第一部分　基础知识

引　言

这一部分将引导读者开始思考算法的设计和分析问题，简单介绍算法的表达方法、将在本书中用到的一些设计策略，以及算法分析中用到的许多基本思想。本书后面的内容都是建立在这些基础知识之上的。

第 1 章是对算法及其在现代计算系统中地位的一个综述。本章给出了算法的定义和一些算法的例子。它还说明了算法是一项技术，就像快速的硬件、图形用户界面、面向对象系统和网络一样。

在第 2 章中，我们给出了书中的第一批算法，它们解决的是对 n 个数进行排序的问题。这些算法是用一种伪代码形式给出的，这种伪代码尽管不能直接翻译为任何常规的程序设计语言，但足够清晰地表达了算法的结构，以便任何一位能力比较强的程序员都能用自己选择的某种语言将算法实现出来。我们分析的排序算法是插入排序，它采用了一种增量式的做法；另外还分析了合并排序算法，它采用了一种递归技术，称为“分治法”。尽管这两种算法所需的运行时间都随 n 的值而增长，但增长的速度是不同的。我们在第 2 章中分析了这两种算法的运行时间，并给出了一种有用的表示方法来表达这些运行时间。

第 3 章给出了这种表示法的准确定义，称为渐近表示。在第 3 章的一开始，首先定义了几种渐近记号，它们主要用于表示算法运行时间的上界和/或下界。第 3 章余下的部分主要给出了一些数学表示方法。这一部分的作用更多的是为了确保读者所用的记号能与本书中的记号体系相匹配，而不主要是教授新的数学概念。

第 4 章更深入地讨论了第 2 章引入的分治方法。特别地，第 4 章包含了解决递归式的方法。递归式主要用于描述递归算法的运行时间。“主方法”(master method)是一种功能很强的技术，它可以用于解决分治算法中出现的递归式。第 4 章中的相当一部分内容都是在证明主方法的正确性，如果跳过这一部分证明内容的话，也没有什么太大的影响。

第 5 章介绍了概率分析和随机化算法。概率分析一般用于确定一些算法的运行时间，在这些算法中，由于同一规模的不同输入可能有着内在的概率分布，因而在这些不同输入之下，算法的运行时间可能有所不同。在有些情况下，我们假定算法的输入符合某种已知的概率分布，于是，算法的运行时间就是在所有可能的输入之下，运行时间的平均值。在其他情况下，概率分布不是来自于输入，而是来自于算法执行过程中所做出的随机选择。如果一个算法的行为不仅由其输入决定，还要由一个随机数生成器所生成的值来决定的话，它就是一个随机化算法(randomized algorithm)。我们可以利用随机化算法，强行使算法的输入符合某种概率分布，从而确保不会有

某一输入会始终导致算法的性能变坏；或者，对于那些允许产生不正确结果的算法，甚至能够将其错误率限制在某个范围之内。

附录A～附录C包含了另一些数学知识，它们对读者阅读本书可能会有所帮助。在阅读本书之前，读者很可能已经知道了附录中给出的大部分知识(我们采用的某些符号约定与读者过去见过的可能会有所不同)，因而，可以将附录视为参考材料。另一方面，你很可能从未见过第一部
4 分中给出的内容。第一部分中的所有各章和附录都是以一种人门指南的风格来编写的。

第 1 章　算法在计算中的作用

什么是算法？为什么要对算法进行研究？相对于计算机中使用的其他技术来说，算法的作用是什么？在本章中，我们就要来回答这些问题。

1.1　算法

简单来说，所谓算法(algorithm)就是定义良好的计算过程，它取一个或一组值作为输入，并产生出一个或一组值作为输出。亦即，算法就是一系列的计算步骤，用来将输入数据转换成输出结果。

我们还可以将算法看作是一种工具，用来解决一个具有良好规格说明的计算问题。有关该问题的表述可以用通用的语言，来规定所需的输入/输出关系。与之对应的算法则描述了一个特定的计算过程，用于实现这一输入/输出关系。

例如，假设需要将一列数按非降顺序进行排序。在实践中，这一问题经常出现。它为我们引入许多标准的算法设计技术和分析工具提供了丰富的问题场景。下面是有关该排序问题的形式化定义：

输入：由 n 个数构成的一个序列$(a_1, a_2, \cdots, a_n)$。

输出：对输入序列的一个排列(重排)$(a'_1, a'_2, \cdots, a'_n)$，使得 $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$。

例如，给定一个输入序列(31，41，59，26，41，58)，一个排序算法返回的输出序列是(26，31，41，41，58，59)。这样的一个输入序列称为该排序问题的一个实例(instance)。一般来说，某一个问题的实例包含了求解该问题所需的输入(它满足有关该问题的表述中所给出的任何限制)。

在计算机科学中，排序是一种基本的操作(很多程序都将它用作一种中间步骤)，因此，迄今为止，科研人员提出了多种非常好的排序算法。对于一项特定的应用来说，如何选择最佳的排序算法要考虑多方面的因素，其中最主要的是考虑待排序的数据项数、这些数据项已排好序的程度、对数据项取值的可能限制、打算采用的存储设备的类型(内存、磁盘、磁带)等。

如果一个算法对其每一个输入实例，都能输出正确的结果并停止，则称它是正确的。我们说一个正确的算法解决了给定的计算问题。不正确的算法对于某些输入来说，可能根本不会停止，或者停止时给出的不是预期的结果。然而，与人们对不正确算法的看法相反，如果这些算法的错误率可以得到控制的话，它们有时也是有用的。关于这一点，在第 31 章中研究用于寻找大质数的算法时介绍了一个例子。但是，一般而言，我们还是仅关注正确的算法。

算法可以用英语、以计算机程序或甚至是硬件设计等形式来表达。不论采用哪种形式，唯一的要求就是算法的规格说明必须提供关于待执行的计算过程的精确描述。

算法可以解决哪些类型的问题？

研究人员并不仅仅是针对排序这一计算问题设计了大量的算法(读者在看到本书的厚度时可能也会这么猜想的)。算法的实际应用面很广，例如：

- 人类基因项目的目标是找出人类 DNA 中的所有 100 000 种基因，确定构成人类 DNA 的 30 亿种化学基对的各种序列，将这些信息存储在数据库中，并开发出用于进行这方面数据分析的工具。这些步骤中的每一个都需要复杂的算法。该项目所涉及的各个问题的解

决方案已超出了本书的范围，但本书中有好几章中的思想在解决这些生物问题时都用到了，这样就使得科学家们可以有效地利用已有资源来完成任务，并且，当利用实验室技术可以提取出更多的信息时，就可以带来人、财、物、时间等方面的节约。

- 因特网使得全世界的人们都能够快速地访问和检索大量的信息。为了能实现这一目的，人们采用了巧妙的算法来管理和操纵大量的数据。这方面必须解决的问题包括寻找好的数据传输路径(第 24 章将介绍解决这些问题的技术)、利用搜索引擎来快速地找到包含特定信息的网页等(有关技术将在第 11 章和第 32 章中介绍)。

6

- 电子商务使得商品和服务可以以电子的形式进行谈判和交易。然而，电子商务要想得到广泛应用的话，非常重要的一点就是保持信用卡号、密码、银行结单等信息的私密性。公共密钥加密技术和数字签名技术(将在第 31 章中介绍)是这一领域内所使用的核心技术，它们的基础就是数值算法和数论理论。
- 在制造业和其他商业应用中，是否能有效地分配稀有资源常常是非常重要的。例如，石油公司可能希望确定该在何处打井，以求最大化预期效益。美国总统候选人可能希望确定该把竞选宣传的资金花在何处，以使赢得竞选胜利的可能性最大。航空公司可能希望以尽可能小的代价来将机组人员分配到不同的航班上，以便做到既到考虑到每一个航班，又不会违反政府有关航空人员调度的规定。因特网服务提供商可能希望确定该把额外的资源置于何处，以便能够更有效地服务其客户。所有这些都是可以利用线性规划求解问题的例子，这一技术将在第 29 章中介绍。

尽管这些例子中某些细节已经超出了本书的范围，我们仍给出了适用于这些问题和问题领域的底层支撑技术。此外，在本书中，我们还说明了如何解决许多具体的问题，例如：

- 给定一幅道路交通图，上面标注出了每一对相邻交叉路口之间的距离。我们的目标就是确定一个交叉路口到另一个交叉路口之间的最短路线。即使不允许每一条路线自我交叉，可能的路线数量也会是巨大的。在所有可能的路线中，该如何来选出最短的路线呢？这里，用一个图来对道路交通图进行建模(前者本身就是对实际道路的一种建模；有关图的内容将在第 10 章和附录 B 中介绍)，希望在图中找出一个顶点到另一个顶点之间的最短路径。在第 24 章中，将看到如何来有效地解决这一问题。
- 给定由 n 个矩阵所组成的一个序列$\langle A_1, A_2, \cdots, A_n\rangle$，希望确定其乘积 $A_1A_2\cdots A_n$。因为矩阵乘法是可以结合的，因而存在着若干合法的乘法顺序。例如，如果 $n=4$，可以按照以下几种顺序来执行矩阵乘法：$(A_1(A_2(A_3A_4)))$，$(A_1((A_2A_3)A_4))$，$((A_1A_2)(A_3A_4))$，$((A_1(A_2A_3))A_4)$，$(((A_1A_2)A_3)A_4)$。如果这些矩阵都是正方矩阵(因而其大小都是一样的)，乘法的顺序对矩阵乘法将花多少时间是没有影响的。然而，如果这些矩阵的大小不同的话(但其大小对矩阵乘法来说是相容的)，那么，乘法的顺序如何就会带来很大的差别了。可能的乘法结合顺序的数量是 n 的指数级的，因此，要尝试所有可能顺序的话，可能会花很长的时间。在第 15 章中，我们将会看到，如何用一种称为动态规划的技术来更为有效地解决这一问题。

7

- 给定一个方程 $ax\equiv b \pmod n$，其中 a、b 和 n 都是整数，希望找出所有(在模 n 时)满足该方程的整数 x。方程的解可能有零个、一个或多个。可以简单地尝试依次用 $x=0$，1，…，$n-1$ 来代入该方程，但第 31 章中给出了一种更为有效的方法。
- 给定平面上 n 个点，希望找出这些点的凸壳，即包含这些点的最小凸多边形。从直观上看，可以将每一个点看成是由一块板上突起的一个钉子表示的。因而，包围这些点的凸壳可以看成是一根包围了所有这些钉子的绷紧的橡皮绳。每一个令橡皮绳发生方向变化

的钉子都是该凸壳的一个顶点(33.3 节的图 33-6 给出了一个例子)。这些点的 2^n 个子集中的任何一个都可能是该凸壳的顶点。知道哪些点是该凸壳的顶点还不够，还需要知道它们出现的顺序。因此，该凸壳的顶点构成有多种选择。第 33 章给出了两种用于寻找凸壳的好方法。

上面这些例子远远没有穷尽所有可能的情况(单从本书的重量就能看出这一点了)，但也体现了许多有趣算法的两个共同特征：

1)有很多候选的解决方案，其中大部分都不是我们所需要的。找到真正需要的解决方案往往不是一件容易的事。

2)有着实际的应用。在上面列出的问题中，最短路径问题提供了最简单的例子。运输公司(如汽车货运或铁路公司)对于如何在公路或铁路网中找出最短路径，有着经济方面的利益，因为选取更短的路线可以降低劳动力和燃料成本。还有，在因特网上，一个路由结点也需要在网络中寻找最短路径，以便快速地路由消息。

数据结构

本书还包含了几种数据结构。数据结构是存储和组织数据的一种方式，以便于对数据进行访问和修改。没有哪一种数据结构可以适用于所有的用途和目的，因而，了解几种数据结构的长处和局限性是相当重要的。

8

技术

尽管可以将本书当作一本有关算法的“菜谱”(cookbook)来使用，但是，仍然可能在将来的某一天，碰到一个问题后，一时不太容易找到一个已有的算法来解决它(例如，本书的练习和思考题中有很多就是这样的情况)。本书将教给读者一些算法设计和分析的技术，以便读者自行设计算法、证明其正确性和理解其效率。

一些比较难的问题

本书的大部分都是关于一些比较高效的算法的。衡量算法效率的常用标准是速度，即一个算法得到最后结果所需要的时间。然而，有一些问题至今还没有已知的有效解法。第 34 章研究了这些问题的一个有趣的子集，即 NP 完全问题。

为什么说 NP 完全问题有趣呢？首先，尽管迄今为止都没有谁能找出 NP 完全问题的有效解法，但也没有人能够证明 NP 完全问题的有效解法是不存在的。换句话说，NP 完全问题是否存在有效算法是未知的。其次，NP 完全问题集有一个显著的特点，即如果该集合中的任何一个问题存在有效的算法，则该集合中的其他所有问题都存在有效算法。NP 完全问题之间的这种关系，使得缺乏有效的算法变得更为令人着急。第三，有几个 NP 完全问题类似于(但又不完全同于)一些有着已知有效算法的问题。对一个问题陈述的一点小小的改动，就会对其已知最佳算法的效率带来很大的变化。

对 NP 完全问题有所了解是很有价值的，因为有些 NP 完全问题会时不时地在实际应用中冒出来。例如，如果要求你找出有关某一 NP 完全问题的有效算法，你很可能会花费大量时间去探寻，结果却是徒劳无益的。如果你能证明该问题是 NP 完全的，就可以把时间花在设计一个有效的算法上，该算法可以给出比较好的、但不一定是最佳可能的结果。

我们来看一个具体的例子。假设有一个货车运输公司，它有一个中央仓库。每一天，它都要在仓库中将货物装满货车，并让它驶往若干个地方去送货。在一天结束时，这辆货车必须最后回到仓库，以便下一天再装货物。为了降低成本，该公司希望选择一条送货车行驶距离最短的送货顺序。这一问题即著名的“旅行商人问题”，并且是个 NP 完全问题。对于该问题，尚没有已知的

有效算法。但是，在特定的假设下，该问题存在着有效的算法，它们能够给出比最短可能的距离
9 长不了多少的总体距离。第 35 章讨论了这种“近似算法”。

练习

1.1-1　给出一个真实世界的例子，其中包含着下列的某种计算问题：排序，确定多矩阵相乘的最佳顺序，或者找出凸壳。

1.1-2　除了运行速度以外，在真实世界问题背景中，还可以使用哪些效率指标？

1.1-3　选择你原来见过的某种数据结构，讨论一下其长处和局限性。

1.1-4　上文中给出的最短路径问题和旅行商人问题有哪些相似之处？有哪些不同之处？

1.1-5　举出一个现实世界的问题例子，它只能用最佳解决方案来解决。再举出另一个例子，在其中“近似”最优解决也足以解决问题。

1.2　作为一种技术的算法

假设计算机无限快，并且计算机存储器是免费的，那么你还有任何理由来研究算法吗？如果你没有别的理由，就是仍然希望证明你的解决方案能够终止并给出正确的结果的话，那么答案就是“是的”。

如果计算机无限快，那么对于某一个问题来说，任何一个可以解决它的正确方法都是可以的。你很可能希望自己的实现能够符合良好的软件工程实践要求(亦即，具有良好的设计和文档说明)，但往往会采用最容易实现的方法。

当然，计算机可以做得很快，但还不能是无限快。存储器可以做到很便宜，但不会是免费
10 的。因此，计算时间是一种有限的资源，存储空间也是一种有限的资源。这些有限的资源必须被有效地使用，那些时间和空间上有效的算法就有助于做到这一点。

效率

解决同一问题的各种不同算法的效率常常相差很大。这种效率上差距的影响往往比硬件和软件方面的差距还要来得大。

例如，在第 2 章中，我们将介绍两个排序算法。第一个称为插入排序算法，对 n 个数据项进行排序的时间大约等于 $c_1 n^2$，其中 c_1 是一个不依赖于 n 的常量。亦即，该算法所需的时间大致正比于 n^2。第二个是合并排序算法，它排序 n 个数据项所需的时间大约是 $c_2 n \lg n$，其中 $\lg n$ 表示 $\log_2 n$，c_2 是另一个同样也不依赖于 n 的常量。插入排序算法与合并排序算法相比，通常有着更小的常量因子，即 $c_1 < c_2$。后面我们还将看到，与对输入规模 n 的依赖相比，常量因子对运行时间的影响要小得多。合并排序算法的运行时间中有个因子 $\lg n$，而插入排序算法的运行时间中有个因子 n，它要比前者大得多了。尽管对于较小的输入规模来说，插入排序要比合并排序更快些，但是，一旦输入规模 n 变得足够大了以后，合并排序的 $\lg n$ 与 n 相比的优势就远远不止是弥补两者之间常量因子上的差距了。无论 c_1 比 c_2 小多少，总会有一个转折点，过了这个点以后，合并排序就会比插入排序运行得更快了。

我们来看一个具体的例子。让一台更快的、运行插入排序的计算机(计算机 A)与一台较慢的、运行合并排序的计算机(计算机 B)进行比较。两者都要对一个大小为一百万个数的数组进行排序。假设计算机 A 每秒能执行 10 亿条指令，而计算机 B 每秒只能执行一千万条指令。因此，在计算能力方面，计算机 A 要比计算机 B 快 100 倍。为了使这一差距更为明显，假设让世界上最能干的程序员采用机器语言，来为计算机 A 编写插入排序算法的代码，所得到的代码需要 $2n^2$

条指令来排序 n 个数(此处，$c_1=2$)。另一方面，让一位平均熟练水平的程序员，采用某种具有低效编译器的高级语言来为计算机 B 编写合并排序算法的代码，所得到的代码有 $50n\lg n$ 条指令(这里 $c_2=50$)。为了排序一百万个数，计算机 A 花的时间为：

$$\frac{2\cdot(10^6)^2 \text{ 条指令}}{10^9 \text{ 条指令/秒}} = 2000 \text{ 秒}$$

计算机 B 花的时间为：

$$\frac{50\cdot 10^6 \lg 10^6 \text{ 条指令}}{10^7 \text{ 条指令/秒}} \approx 100 \text{ 秒}$$

计算机 B 由于采用了一个运行时间增长得更为缓慢的算法，尽管它用的是效率较低的编译器，运行速度也比计算机 A 快了 20 倍！当对一千万个数据进行排序时，合并排序的优势就更为明显了：这时，插入排序要花大约 2.3 天的时间，合并排序只需不到 20 分钟的时间。一般来说，随着问题规模的增长，合并排序的相对优势也会愈加明显。

11

算法和其他技术

上面给出的例子说明了算法就像计算机硬件一样，是一种技术。总体的系统性能不仅依赖于选择快速的硬件，还依赖于选择有效的算法。算法领域的研究和其他计算机技术领域一样，正不断地取得飞速的进展。

读者可能会想，与其他先进的计算机技术(如以下列出的)相比，算法对于当代的计算机是否真的那么重要？

- 具有高时钟主频、流水线技术、超级标量结构的硬件
- 易于使用的、直观的图形用户界面(GUI)
- 面向对象的系统
- 局域网和广域网技术

答案是“是的”。尽管对于有些应用来说，在应用这一层面上没有什么特别明显的算法方面的要求(例如，一些简单的 Web 应用就是这样的)，但大多数问题对算法还是有一定程度要求的。例如，假设有一种基于 Web 的服务，它用于确定如何从一个地方旅行至另一个地方。(在写作本书时，已经有了好几种这样的服务了。)其实现将依赖于快速的计算机硬件、图形用户界面、广域网技术，甚至还可能要依赖于面向对象技术。然而，除此之外，它还需要为某些操作设计算法，如寻找路由(很可能采用最短路径算法)、显示地图、插入地址等。

此外，即使是那些在应用层面上对算法性内容没什么要求的应用，其实也是相当依赖于算法的。该应用要依赖于快速的计算机硬件吧？硬件的设计就要用到算法。该应用要用到图形用户界面吧？任何 GUI 的设计也要依赖于算法。该应用要依赖于网络技术吧？网络路由对算法也有着很大的依赖。该应用是采用某种非机器代码的语言编写而成的吧？那么，它就要由编译器、解释器或汇编器来处理，所有这些软件都要大量用到各种算法。算法是当代计算机中用到的大部分技术的核心。

随着计算机性能的不断增长，可以利用计算机来解决比以往更大的问题。从上文有关插入排序与合并排序的比较中可以看出，正是对于更大的问题规模，不同算法在效率方面的差异才会变得特别显著。

是否拥有扎实的算法知识和技术基础，是区分真正熟练的程序员与新手的一项重要特征。利用当代的计算技术，无需了解很多算法方面的东西，也可以完成一些任务。但是，有了良好的算法基础和背景的话，可以做的事就要多得多了。

12

练习

1.2-1　给出一个实际应用的例子，它在应用这一层次上要求有算法性的内容。讨论其中所涉及算法的功能。

1.2-2　假设我们要比较在同一台计算机上插入排序和合并排序的实现。对于规模为 n 的输入，插入排序要运行 $8n^2$ 步，而合并排序要运行 $64n\lg n$ 步。当 n 取怎样的值时，插入排序的性能要优于合并排序？

1.2-3　对于一个运行时间为 $100n^2$ 的算法，要使其在同一台机器上，比一个运行时间为 2^n 的算法运行得更快，n 的最小取值是多少？

思考题

1-1　算法运行时间的比较

对于下表中的每一个函数 $f(n)$ 和时间 t，求出可以在时间 t 内被求解出来的问题的最大规模 n。假设解决该问题的算法解决该问题需要 $f(n)$ 微秒。

	1 秒	1 分钟	1 小时	1 天	1 个月	1 年	1 个世纪
$\lg n$							
$\sqrt{n}$							
n							
$n\lg n$							
n^2							
n^3							
2^n							
$n!$							

13

本章注记

有许多全面介绍算法这一主题的书都是非常不错的，如 Aho、Hopcroft 和 Ullman[5，6]，Baase 和 Van Gelder[26]，Brassard 和 Bratley[46，47]，Goodrich 和 Tamassia[128]，Horowitz，Sahni 和 Rajasekaran[158]，Kingston[179]，Knuth[182，183，185]，Kozen[193]，Manber[210]，Mehlhorn[217，218，219]，Purdom 和 Brown[252]，Reingold，Nievergelt 和 Deo[257]，Sedgewick[269]，Skiena[280]，以及 Wilf[315]等著作。Bentley[39，40]和 Gonnet[126]对算法设计中一些更具体实际的问题进行了讨论。对算法这一领域的综述可以参见《Handbook of Theoretical Computer Science，Volume A》[302]，以及《CRC Handbook on Algorithms and Theory of Computation》[24]。Gusfield[136]、Pevzner[240]、Setubal 和 Medinas[272]、Waterman[309]

14 等教材则对计算生物学中用到的各种算法做了概述性的介绍。

第2章 算法入门

在本章中，我们要向读者介绍一个贯穿本书的框架，后续的算法设计和分析都是在这个框架中进行的。这一部分内容基本上是完全独立的，也有对第3章和第4章中一些内容的引用。（本章还给出了几个求和的式子，具体如何求解这几个式子可以参见附录A。）

首先，要分析一下如何用插入排序算法解决第1章中提出的排序问题。我们定义了一种“伪代码”，它对于编写过计算机程序的读者来说应该是熟悉的。我们要用这种伪代码来说明应该如何描述算法。在描述了算法之后，再证明它能正确地完成排序任务，并对其运行时间进行分析。在分析过程中，引入了一种记号，它侧重于表达算法的运行时间是如何随着待排序的数据项数而增加的。在讨论过插入排序算法后，还要介绍算法设计中的分治法（divide-and-conquer），并利用该方法来设计一个称为合并排序的算法。在本章的最后，对合并算法的运行时间进行了分析。

2.1 插入排序

我们要分析的第一个算法是插入排序算法，它解决的是第1章中介绍的排序问题：

输入：n个数$\langle a_1, a_2, \cdots, a_n\rangle$。

输出：输入序列的一个排列（即重新排序）$\langle a'_1, a'_2, \cdots, a'_n\rangle$，使得$a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$。

待排序的数也称为关键字（key）。

在本书中，主要用伪代码书写的程序形式来表达算法，这种伪代码在很多方面都与C、Pascal或Java等语言比较类似。如果熟悉这几种语言的话，阅读书中的算法时应该不会有什么困难。伪代码与真实代码的不同之处在于，在伪代码中，可以采用最具表达力的、最简明扼要的方法，来表达一个给定的算法。有时，最清晰的方法就是英语，因此，当遇到在一段“真正的”代码中嵌入了一个英语短语或句子时，不要感到惊讶。在伪代码和真正的代码之间还有一点区别，就是伪代码一般不关心软件工程方面的问题。亦即，数据抽象、模块化和错误处理等问题往往都被忽略掉了，以便更简练地表达算法的核心内容。

我们首先以插入排序算法开始，这是一个对少量元素进行排序的有效算法。插入排序的工作机理与很多人打牌时，整理手中牌时的做法差不多。在开始摸牌时，我们的左手是空的，牌面朝下放在桌上。接着，一次从桌上摸起一张牌，并将它插入到左手一把牌中的正确位置上。为了找到这张牌的正确位置，要将它与手中已有的每一张牌从右到左地进行比较，如图2-1中所示。无论在什么时候，左手中的牌都是排好序的，而这些牌原先都是桌上那副牌里最顶上的一些牌。

图2-1 利用插入排序方法来排序手中的牌

插入排序算法的伪代码是以一个过程的形式给出的，称为INSERTION-SORT，它的参数是一个数组$A[1..n]$，包含了n个待排序的数。（在代码中，A中元素个数n用$length[A]$来表示。）输入的各个数字是原地排序的（sorted in place），意即这些数字就是在数组A中进行重新排序的，在任何时刻，至多只有

其中的常数个数字是存储在数组之外的。当过程 INSERTION-SORT 执行完毕后，输入数组 A 中就包含了已排好序的输出序列。

16

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j]into the sorted sequence A[1..j−1].
4          i ← j−1
5          while i>0 and A[i]>key
6               do A[i+1] ← A[i]
7                  i ← i−1
8          A[i+1] ← key
```

循环不变式与插入算法的正确性

图 2-2 中示出了这个算法在数组 $A=\langle 5, 2, 4, 6, 1, 3\rangle$ 上的工作过程。下标 j 指示了待插入到手中的“当前牌”。在外层 for 循环(循环变量为 j)的每一轮迭代的开始，包含元素 $A[1..j-1]$ 的子数组构成了左手中当前已排好序的一手牌，元素 $A[j+1..n]$ 对应于仍然在桌上的那堆牌。实际上，元素 $A[1..j-1]$ 是最初在位置 1 到 $j-1$ 上的那些元素，但现在已经排好序了。下面，我们以循环不变式(loop invariant)的形式，来形式化地表达 $A[1..j-1]$ 的这些性质：

在过程第 1～8 行的 for 循环中，在每一轮迭代的开始，子数组 $A[1..j-1]$ 中包含了最初位于 $A[1..j-1]$、但目前已排好序的各个元素。

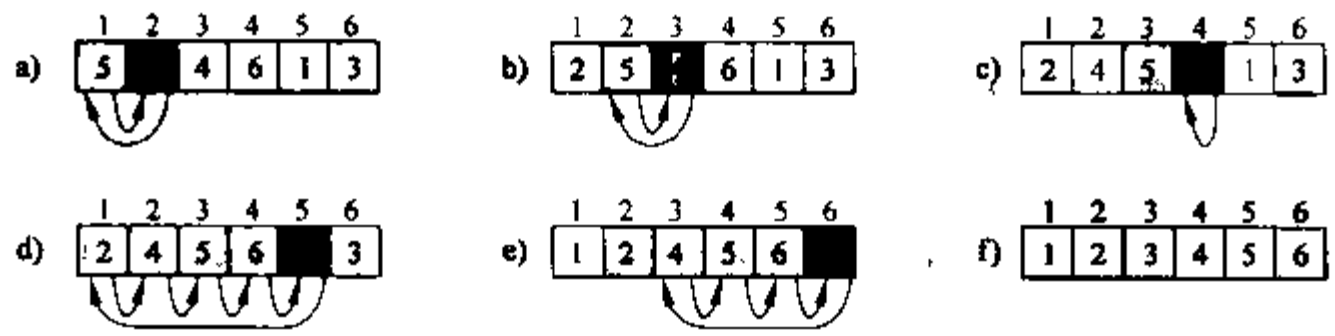


图 2-2　INSERTION-SORT 在数组 $A=\langle 5, 2, 4, 6, 1, 3\rangle$ 上的处理过程。数组下标出现在矩形的上方，数组各个位置中存储的数字出现在矩形中。a)～e)过程第 1～8 行 for 循环各次迭代的情况。在每次迭代中，黑色方框里的值是取自 $A[j]$ 的关键字值，在过程第 5 行的测试中，要将它与左边阴影框中的各个值进行比较。阴影箭头示出了过程第 6 行中，将一数组中的各个值向右移一个位置这一动作；黑色箭头示出了要将关键字值移至何处(过程中的第 8 行)。f)中示出了最终已排好序的数组

循环不变式主要用来帮助我们理解算法的正确性。对于循环不变式，必须证明它的三个性质：

17

初始化：它在循环的第一轮迭代开始之前，应该是正确的。

保持：如果在循环的某一次迭代开始之前它是正确的，那么，在下一次迭代开始之前，它也应该保持正确。

终止：当循环结束时，不变式给了我们一个有用的性质，它有助于表明算法是正确的。

当头两个性质成立时，就能保证循环不变式在循环的每一轮迭代开始之前，都是正确的。请注意这儿的推理与数学归纳法的相似性。在数学归纳法中，要证明某一性质是成立的，必须首先证明其基本情况和一个归纳步骤都是成立的。这儿，证明不变式在第一轮迭代开始之前是成立的，就有点类似于归纳法中对基本情况的证明；证明不变式在各次迭代之间保持成立，就有点类

似于归纳法中对归纳步骤的证明。

有关循环不变式的第三项性质可能是最重要的，因为我们主要是用不变式来证明算法正确性的。此外，它与数学归纳法的常见用法也是不同的：在归纳法中，归纳步骤是无穷地使用的；在这儿，当循环结束时，即终止"归纳"。

下面，我们就来看看对于插入排序算法而言，如何证明这些性质是成立的。

初始化：首先，先证明在第一轮迭代开始之前，循环不变式是成立的。此时，$j=2$,[⊖]而子数组为 $A[1..j-1]$。亦即，它只包含一个元素 $A[1]$，实际上就是最初在 $A[1]$中的那个元素。这个子数组是已排序的(这一点是显然的)，这样就证明了循环不变式在循环的第一轮迭代开始之前是成立的。

保持：接下来，我们来考虑第二个性质：证明每一轮循环都能使循环不变式保持成立。从非形式化的意义上来看，在外层 for 循环的循环体中，要将 $A[j-1]$、$A[j-2]$、$A[j-3]$等元素向右移一个位置，直到找到 $A[j]$的适当位置时为止(第4～7行)，这时将 $A[j]$的值插入(第8行)。如果要更形式化地证明第二个性质成立的话，就需要陈述并证明对内层 while 循环也有一个循环不变式成立。但是，此处我们更倾向于暂时不陷入过于形式化的细节之中，而是依赖于非形式化的分析，来证明第二个性质对于外层循环是成立的。

终止：最后，分析一下循环结束时的情况。对插入排序来说，当 j 大于 n 时(即当 $j=n+1$ 时)，外层 for 循环结束。在循环不变式中，将 j 替换为 $n+1$，就有子数组 $A[1..n]$包含了原先 $A[1..n]$中的元素，但现在已排好序了。但是，子数组 $A[1..n]$其实就是整个数组！因此，整个数组就排好序了，这意味着算法是正确的。 18

在本章中稍后的部分及其他一些章节中，将利用这种循环不变式的方法来证明算法的正确性。

伪代码中的约定

在伪代码的使用中有以下一些约定：

1)书写上的"缩进"表示程序中的分程序(程序块)结构。例如，从第1行开始的 for 循环的体包括了第2～8行，从第5行开始的 while 循环的体包括第6～7行。这种缩进风格也适用于 if-then-else 语句。用缩进取代传统的 begin 和 end 语句来表示程序的块结构，可以大大提高代码的清晰性。[⊜]

2)while，for，repeat 等循环结构和 if，then，else 条件结构与 Pascal 中相同。[⊜]然而，对 for 循环来说有一点小小的不同：在 Pascal 中，循环计数器变量在退出循环时是未定义的，但在本书中，在退出循环后，循环计数器的值仍然保持。于是，紧接着一个 for 循环之后，循环计数器的值就是第一个超出 for 循环终值的那个数字。我们在对插入排序的正确性证明中，也用到了这一性质。在第1行中，for 循环的头为 for $j \leftarrow 2$ to $length[A]$，因此，当此循环结束时，$j=length[A]+1$(或者可以说，$j=n+1$，因为 $n=length[A]$)。

3)符号"▷"表示后面部分是个注释。

4)多重赋值 $i \leftarrow j \leftarrow e$ 是将表达式 e 的值赋给变量 i 和 j；等价于赋值 $j \leftarrow e$，再进行赋值 $i \leftarrow j$。

5)变量(如 i，j 和 key 等)是局部于给定过程的。在没有显式说明的情况下，我们不使用全局变量。

⊖ 当循环是个 for 循环时，在循环第一次迭代开始之前，我们是在对循环计数变量进行赋值之后、在循环首部的第一次测试之前，对循环不变式进行检查的。在 INSERTION-SORT 算法中，这个时间点就是在将2赋给变量 j 之后、在首次测试 $j \leqslant length[A]$是否成立之前。

⊜ 在真正的程序设计语言里，一般不建议单独使用缩进来表示分程序结构，这是因为当代码块跨页时，缩进的层次很难确定。

⊜ 多数分程序结构的语言中都有与此等价的语言构造，具体的语法与 Pascal 中的可能有所不同。

6)数组元素是通过“数组名[下标]”这样的形式来访问的。例如，$A[i]$表示数组 A 的第 i 个
[19] 元素。符号“..”用来表示数组中的一个取值范围，例如，$A[1..j]$就表示 A 的一个子数组，它包含了 j 个元素 $A[1]$，$A[2]$，…，$A[j]$。

7)复合数据一般组织成对象，它们是由属性(attribute)或域(field)所组成的。域的访问是由域名后跟由方括号括住的对象名形式来表示。例如，数组可以被看作是一个对象，其属性有 *length*，表示数组中元素的个数，如 *length*$[A]$就表示数组 A 中的元素个数。在表示数组元素和对象属性时，都要用到方括号，一般来说，通过上下文就可以看出其含义。

用于表示一个数组或对象的变量被看作是指向表示数组或对象的数据的一个指针。对于某个对象 x 的所有域 f，赋值 $y \leftarrow x$ 就使得 $f[y]=f[x]$。更进一步，如果有 $f[x]\leftarrow 3$，则不仅有 $f[x]=3$，同时 $f[y]=3$。换言之，在赋值 $y \leftarrow x$ 后，x 和 y 指向同一个对象。

有时，一个指针不指向任何对象。这时，我们赋给它 NIL。

8)参数采用按值传递方式：被调用的过程会收到参数的一份副本。如果它对某个参数赋值的话，主调过程是看不见这一变动的。当对象被传递时，实际传递的是一个指向该对象数据的指针，而对象的各个域则不被拷贝。例如，如果 x 是某个被调用过程的参数，在被调过程中的赋值 $x \leftarrow y$ 对主调过程来说是不可见的。但是，赋值 $f[x]\leftarrow 3$ 却是可见的。

9)布尔运算符“and”和“or”都具有短路能力。亦即，当我们求表达式“x and y”的值时，首先计算 x 的值。如果 x 的值为 FALSE，那么整个表达式的值就不可能为 TRUE 了，因而就无需再对 y 求值了。但是，如果 x 的值为 TRUE 的话，就必须进一步计算出 y 的值，才能确定整个表达式的值。类似地，在计算表达式“x or y”的值时，仅当 x 的值为 FALSE 时，才需要计算子表达式 y 的值。短路运算符允许我们写出如“$x\neq$NIL and $f[x]=y$”这样的布尔表达式，而不用担心当我们试图在 x 为 NIL 时计算 $f[x]$，会发生怎样的情况。

练习

2.1-1 以图 2-2 为模型，说明 INSERTION-SORT 在数组 $A=\langle 31, 41, 59, 26, 41, 58\rangle$上的执
[20] 行过程。

2.1-2 重写过程 INSERTION-SORT，使之按非升序(而不是按非降序)排序。

2.1-3 考虑下面的查找问题：

输入：一列数 $A=\langle a_1, a_2, \cdots, a_n\rangle$和一个值 v。

输出：下标 i，使得 $v=A[i]$，或者当 v 不在 A 中出现时为 NIL。

写出针对这个问题的线性查找的伪代码，它顺序地扫描整个序列以查找 v。利用循环不变式证明算法的正确性。确保所给出的循环不变式满足三个必要的性质。

2.1-4 有两个各存放在数组 A 和 B 中的 n 位二进整数，考虑它们的相加问题。两个整数的和以二进制形式存放在具有$(n+1)$个元素的数组 C 中。请给出这个问题的形式化描述，并写出伪代码。

2.2　算法分析

算法分析即指对一个算法所需要的资源进行预测。内存、通信带宽或计算机硬件等资源偶尔会是我们主要关心的，但通常，资源是指我们希望测度的计算时间。一般来说，对于一个给定的问题，通过分析几种候选算法，可以很容易地从中选出一个最有效的算法。这种分析的结果可能是找出了不止一个的候选算法，但在这一过程中，通常都要去掉几个较差的算法。

在分析一个算法之前，要建立有关实现技术的模型，包括描述所用资源的及代价的模型。本书主要采用一种通用的单处理器、随机存取机(random-access machine，RAM)计算模型来作为我

们的实现技术，算法可以用计算机程序来实现。在RAM模型中，指令一条接一条地执行，没有并发操作。在后面几章中，将讨论有关数字化计算机硬件的模型。

严格来说，应当精确地定义RAM模型的指令及其代价。然而，这么做比较单调，对理解算法的设计和分析也不会带来多大的作用。要注意不能滥用RAM模型，例如，如果某一RAM具 [21]
有一条能进行排序的指令会怎样呢？这时，我们只用一条指令就可以完成排序了。这样的RAM是不现实的，因为真实的计算机中并没有这种指令。因此，我们的依据就是真实计算机的设计方式。RAM模型包含了真实计算机中常见的指令：算术指令(加法、减法、乘法、除法、取余、向下取整、向上取整指令)、数据移动指令(装入、存储、复制指令)和控制指令(条件和非条件转移、子程序调用和返回指令)。其中每条指令所需的时间都为常量。

RAM模型中的数据类型有整数类型和浮点实数类型。在本书中，一般不关心数据的精度问题，但在一些应用中，精度还是非常关键的。我们还假设数据的每一个字(word)有着最大长度限制。例如，当处理规模为 n 的输入时，一般假定整数是由 $c\lg n$ 位表示的，此处 c 为大于等于1的常量。之所以要求有 $c\geqslant 1$，是因为这样一个字就能容纳下 n 的值，从而可以引用每一个输入元素；之所以要求 c 为常量，是因为这样字长不会任意地增长。(如果字长可以任意地增长的话，就能在一个字中存储巨量的数据，并可以在常量时间内对它进行处理了，这显然是一种不真实的场景。)

真实的计算机包含了一些以上未列出的指令，这些指令代表了RAM模型中的一个灰色区域。例如，指数运算是否是一种常量时间的指令呢？在一般的情况下，不是的；计算 x^y(x 和 y 都是实数)需要若干条指令。然而，在受限的情况下，指数运算是一种常量时间的操作。很多计算机上都有一种"左移"指令，它在常量时间内，将一个整数的各位向左移 k 位。在多数计算机中，将一个整数的各位向左移一位相当于将该数乘上2。将该整数的各位向左移 k 位，相当于将该数乘上 2^k。于是，这样的计算机可以用一条常量时间的指令来计算 2^k，即将整数1向左移 k 位，只要 k 不大于一个计算机字中的位数即可。我们将努力避免在RAM模型中出现这样的灰色区域，但是，当 k 是一个足够小的正整数时，我们将把 2^k 的计算当作一个常量时间的操作。

在RAM模型中，我们没有试图对当代计算机常见的存储器层次进行建模。亦即，并不对高速缓存和虚拟内存(它通常采用请页机制实现)进行建模。有几种计算模型试图反映出存储层次的效果，因为这种效果有时在真实计算机上的真实程序中是非常重要的。本书中有一些问题分析了存储层次的效果，但是本书中的分析大多对它不予考虑。与RAM模型相比，包含了存储层次的模型往往要复杂得多，因而比较难以运用。此外，RAM模型分析通常能够很好地预测实际计算机上的性能。

在RAM模型中，即使是分析一个简单的算法，也往往是比较困难的。在许多算法的分析 [22]
中，所需的数学工具包括组合数学、概率论、代数、还要求能够识别出一个公式中的最重要的部分。因为一个算法的行为在每一种可行的输入下可能会有所不同，因此，我们需要一种方法和手段，以简单明了的公式的形式来总结这些行为。

即使仅选择一种机器模型来分析某一给定的算法，在表示分析时，仍然会面对多种选择。我们希望能有一种表示方法，它书写和处理起来都比较简单，能够显示出算法资源需求的重要特征，摒弃掉琐碎的细节。

插入排序算法的分析

INSERTION SORT过程的时间开销与输入有关：排序1000个数的时间比排序三个数的时间要长。还有，即使排序两个相同长度的输入序列，所需的时间也可能不同。这取决于它们已排序的程度。一般来说，算法所需时间是与输入规模同步增长的，因而常常将一个程序的运行时间表示为其输入的函数。这就要求对术语"运行时间"和"输入规模"更仔细地加以定义。

输入规模的概念与具体问题有关。对许多问题来说(如排序或计算离散傅里叶变换)，最自然的度量标准是输入中的元素个数，例如，待排序数组的大小 n。对另一些问题(如两个整数相乘)，其输入规模的最佳度量是输入数在二进制表示下的位数。有时，用两个数(而不是一个)来表示输入可能更合适。例如，某一算法的输入是个图，则输入规模可以由图中顶点数和边数来表示。在下面讨论的每一个问题中，我们都将指明所用的度量标准。

算法的运行时间是指在特定输入时，所执行的基本操作数(或步数)。可以很方便地定义独立于具体机器的"步骤"概念。目前，先采用以下观点，每执行一行伪代码都要花一定量的时间。虽然每一行所花的时间可能不同，但我们假定每次执行第 i 行所花的时间都是常量 c_i。这种观点与
23 RAM 模型是一致的，同时也反映出了伪代码在多数真实计算机上是如何实现的。⊖

在下面的讨论中，我们由繁到简地给出 INSERTION-SORT 运行时间的表达式。简单的表达式使得更容易从众多的算法中，选出最为有效的一个算法。

首先给出 INSERTION-SORT 过程中，每一条指令的执行时间及执行次数。对 $j=2, 3, \cdots, n$，$n=length[A]$，设 t_j 为第 5 行中 while 循环所做的测试次数。当 for 或 while 循环以通常方式退出(即，因为循环头部的测试条件不满足而退出)时，测试要比循环体多执行 1 次。另外，还假定注解部分是不可执行的，因而不占运行时间。

INSERTION-SORT(A)		*cost*	*times*
1	for $j \leftarrow 2$ to $length[A]$	c_1	n
2	do $key \leftarrow A[j]$	c_2	$n-1$
3	▷ Insert $A[j]$into the sorted sequence $A[1..j-1]$.	0	$n-1$
4	$j \leftarrow j-1$	c_4	$n-1$
5	while $i>0$ and $A[i]>key$	c_5	$\sum_{j=2}^{n} t_j$
6	do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j-1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

该算法总的运行时间是每一条语句执行时间之和。如果执行一条语句需要 c_i 步，又共执行了 n 次这条语句，那么它在总运行时间中占 $c_i n$。⊜为计算总运行时间 $T[n]$，对每一对 cost 与 times 之积求和，得：

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

24 即使是对给定规模的输入，一个算法的运行时间也有可能要依赖于给定的是该规模下的哪种输入。例如，在 INSERTION-SORT 中，如果输入数组已经是排好序的话，就会出现最佳情

⊖ 这儿有些细微的东西要解释一下。对于所需计算时间不为常量的过程来说，我们用英语表示的计算步骤通常是可变的。例如，在本书中稍后部分中，我们可能会说"根据 x 坐标对各个点进行排序"，它需要多于常量的时间。还有，请注意一个调用了某个子程序的语句需要常量的时间，尽管该子程序一旦被调用后，可能需要更多的时间。亦即，我们将调用该子程序、向它传递参数等的过程，与执行该子程序的过程区分开来。

⊜ 这一特性对内存这样的资源来说未必成立。假设一条语句引用了 m 个内存字的语句，并执行了 n 次，则它总共需要的内存不一定是 mn 个字。

况。对 $j=2, 3, \cdots, n$ 中的每一个值，我们发现，在第 5 行中，当 i 取其初始值 $j-1$ 时，都有 $A[i] \leqslant key$。于是，对 $j=2, 3, \cdots, n$，有 $t_j=1$，最佳的运行时间为：

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

这一运行时间可以表示为 $an+b$，常量 a 和 b 依赖于语句的代价 c_i；因此，它是 n 的一个线性函数。

如果输入数组是按照逆序排序的(亦即，是按递降顺序排序的)，那么就会出现最坏情况。我们必须将每个元素 $A[j]$ 与整个已排序的子数组 $A[1..j-1]$ 中的每一个元素进行比较，因而，对 $j=2, 3, \cdots, n$，有 $t_j=j$。请注意

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad \text{和} \quad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(有关这两个求和式的解法可以参见附录 A。)我们发现，在最坏情况下，INSERTION-SORT 的运行时间为

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

这一最坏情况运行时间可以表示为 an^2+bn+c，常量 a、b 和 c 仍依赖于语句的代价 c_i；因此，这是一个关于 n 的二次函数。

一般来说，如插入排序中一样，一个算法的运行时间对某一给定的输入来说，往往是固定的。在后续的各章中，可以看到一些有意思的“随机化”算法，其行为即使对于固定的输入，也是可以变化的。

最坏情况和平均情况分析

在分析插入排序时，我们既考察了最佳情况，也考察了最坏情况。在最佳情况下，输入数组是已经排好序的；在最坏情况下，输入数组是按逆序排序的。在本书的余下部分里，一般考察算法的最坏情况运行时间，亦即，对于规模为 n 的任何输入，算法的最长运行时间。这样做的理由有三点：

25

- 一个算法的最坏情况运行时间是在任何输入下运行时间的一个上界。知道了这一点，就能确保算法的运行时间不会比这一时间更长。也就是说，我们不需要对运行时间做某种复杂的猜测，并期望它不会变得更坏了。
- 对于某些算法来说，最坏情况出现得还是相当频繁的。例如，当在数据库中检索一条信息时，当要找的信息不在数据库中时，检索算法的最坏情况就会经常出现。在有些检索应用中，要检索的信息常常是数据库中没有的。
- 大致上看来，“平均情况”通常与最坏情况一样差。假定我们随机地选择 n 个数，并利用插入排序算法对它们进行排序。要决定应该在在子数组 $A[1..j-1]$ 中的哪一个位置上插入元素 $A[j]$，需要多长时间？在平均情况下，$A[1..j-1]$ 中的一半元素小于 $A[j]$，一半的元素大于 $A[j]$。于是，在平均情况下，要检查子数组 $A[1..j-1]$ 中一半的元素，因而有 $t_j=j/2$。如果求一下平均情况下运行时间的话，会发现它是输入规模的一个二次函数，这与最坏情况下的运行时间是一样的。

在某些特定的情况下，我们可能会对一个算法的平均情况或期望的运行时间感兴趣；在第5章中，将介绍概率分析(probabilistic analysis)技术，它可以用来确定一个算法期望的运行时间。但是，进行平均情况分析存在着一个问题，即对于某个特定的问题来说，“平均”输入是由哪些东西构成的可能不一定很明显。通常，我们会假定具有某一规模的所有输入都是等可能的。在实践中，这一假设可能不一定成立，但是我们有时可以采用随机化算法(randomized algorithm)，这种算法可以做出随机的选择，从而允许对算法进行概率分析。

增长的量级

为了简化对INSERTION-SORT过程的分析，我们做了某些简化抽象。首先，忽略了每条语句的真实代价，而用常量 c_i 来表示。其次，还可以更加简单：最坏情况运行时间是 an^2+bn+c，a，b，c 是依赖于 c_i 的常量。这样，就不仅忽略了真实的代价，也忽略了抽象代价 c_i。

现在再做进一步的抽象，即运行时间的增长率(rate of growth)，或称增长的量级(order of growth)。这样，我们就只考虑公式中的最高次项(例如，an^2)，因为当 n 很大时，低阶项相当来
26 说不太重要。另外，还忽略最高次项的常数系数，因为在考虑较大规模输入下的计算效率时，相对于增长率来说，系数是次要的。例如，插入排序的最坏情况时间代价为 $\Theta(n^2)$。本章中，我们先非正式地用这种 Θ 表示记号，在第3章中，还要给出它的准确定义。

如果一个算法的最坏情况运行时间要比另一个算法的低，我们就常常认为它的效率更高。在输入的规模较小时，由于常数项和低次项的影响，这种看法有时可能是不对的，但是对规模足够大的输入来说，一个具有 $\Theta(n^2)$ 的算法在最坏情况下，比 $\Theta(n^3)$ 的算法运行得更快。

练习

2.2-1　用 Θ 形式表示函数 $n^3/1000-100n^2-100n+3$。

2.2-2　考虑对数组 A 中的 n 个数进行排序的问题：首先找出 A 中的最小元素，并将其与 $A[1]$ 中的元素进行交换。接着，找出 A 中的次最小元素，并将其与 $A[2]$ 中的元素进行交换。对 A 中头 $n-1$ 个元素继续这一过程。写出这个算法的伪代码，该算法称为选择排序(selection sort)。对这个算法来说，循环不变式是什么？为什么它仅需要在头 $n-1$ 个元素上运行，而不是在所有 n 个元素上运行？以 Θ 形式写出选择排序的最佳和最坏情况下的运行时间。

2.2-3　再次考虑线性查找问题(见练习2.1-3)。在平均情况下，需要检查输入序列中的多少个元素？假定待查找的元素是数组中任何一个元素的可能性是相等的。在最坏情况下又怎样呢？用 Θ 形式表示的话，线性查找的平均情况和最坏情况运行时间怎样？对你的答案加以说明。

2.2-4　应如何修改任何一个算法，才能使之具有较好的最佳情况运行时间？

2.3　算法设计

算法设计有很多方法。插入排序使用的是增量(incremental)方法：在排好子数组 $A[1..j-1]$
27 后，将元素 $A[j]$ 插入，形成排好序的子数组 $A[1..j]$。

在本节中，要介绍另一设计策略，叫做“分治法”(divide-and-conquer)。下面要用分治法来设计一个排序算法，使其性能比插入排序好得多。学过第4章就可以知道，分治算法的优点之一是可以利用在第4章中介绍的技术，很容易地确定其运行时间。

2.3.1　分治法

有很多算法在结构上是递归的：为了解决一个给定的问题，算法要一次或多次地递归调用其自身来解决相关的子问题。这些算法通常采用分治策略：将原问题划分成 n 个规模较小而结构与原问题相似的子问题；递归地解决这些子问题，然后再合并其结果，就得到原问题的解。

分治模式在每一层递归上都有三个步骤：

分解(Divide)：将原问题分解成一系列子问题；

解决(Conquer)：递归地解各子问题。若子问题足够小，则直接求解；

合并(Combine)：将子问题的结果合并成原问题的解。

合并排序(merge sort)算法完全依照了上述模式，直观地操作如下：

分解：将 n 个元素分成各含 $n/2$ 个元素的子序列；

解决：用合并排序法对两个子序列递归地排序；

合并：合并两个已排序的子序列以得到排序结果。

在对子序列排序时，其长度为1时递归结束。单个元素被视为是已排好序的。

合并排序的关键步骤在于合并步骤中的合并两个已排序子序列。为做合并，引入一个辅助过程MERGE(A, p, q, r)，其中 A 是个数组，p、q 和 r 是下标，满足 $p \leqslant q < r$。该过程假设子数组 $A[p..q]$ 和 $A[q+1..r]$ 都已排好序，并将它们合并成一个已排好序的子数组代替当前子数组 $A[p..r]$。

MERGE过程的时间代价为 $\Theta(n)$，其中 $n=r-p+1$ 是待合并的元素个数。下面来说明该算法 [28]
的工作过程。再举扑克牌这个例子，假设有两堆牌面朝上地放在桌上，每一堆都是已排序的，最小的牌在最上面。我们希望把这两堆牌合并成一个排好序的输出堆，面朝下地放在桌上。基本步骤包括在面朝上的两堆牌中，选取顶上两张中较小的一张，将其取出后(它所在堆的顶端又会露出一张新的牌)面朝下地放到输出堆中。重复这个步骤，直到某一输入堆为空时为止。这时，把输入堆中余下的牌面朝下地放入输出堆中即可。从计算的角度来看，每一个基本步骤所花时间是个常量，因为我们只是查看并比较顶上的两张牌。又因为至多进行 n 次比较，所以合并排序的时间为 $\Theta(n)$。

下面的伪代码实现了上述想法，但有一个小的变化，即在每一个基本步骤中，避免了检查是否每一个堆都是空的。其想法是在每一堆的底部放上一张“哨兵牌”(sentinel card)，它包含了一个特殊的值，用于简化代码。此处，利用∞来作为哨兵值，这样每当露出一张值为∞的牌时，它不可能是两张中较小的牌，除非另一堆也露出了哨兵牌。但是，一旦发生这种两张哨兵牌同时出现的情况时，说明两堆牌中的所有非哨兵牌都已经被放到输出堆中去了。因为我们预先知道只有 $r-p+1$ 张牌会被放到输出堆中去，因此，一旦执行了 $r-p+1$ 个基本步骤后，算法就可以停止下来了。

```
MERGE(A, p, q, r)
 1  n1 ← q−p+1
 2  n2 ← r−q
 3  create arrays L[1..n1+1]and R[1..n2+1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p+i−1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q+j]
 8  L[n1+1] ← ∞
 9  R[n2+1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i]≤R[j]
14            then A[k] ← L[i]
15                 i ← i+1
16            else A[k] ← R[j]
17                 j ← j+1
```

具体来说，MERGE过程是这样工作的：第1行计算子数组 $A[p..q]$ 的长度 n_1，第2行计算

子数组 $A[q+1..r]$的长度 n_2。在第 3 行中，创建了数组 L 和 R(表示"left"和"right")，长度各为
29 n_1+1 和 n_2+1。第 4～5 行中的 for 循环将子数组 $A[p..q]$复制到 $L[1..n_1]$中去；第 6～7 行中的 for 循环将子数组 $A[q+1..r]$复制到 $R[1..n_2]$中去。第 8～9 行将哨兵置于数组 L 和 R 的末尾。第 10～17 行(如图 2-3 中所示)通过维护以下的循环不变式，执行了 $r-p+1$ 个基本步骤：

在第 12～17 行 for 循环每一轮迭代的开始，子数组 $A[p..k-1]$包含了 $L[1..n_1+1]$和$R[1..n_2+1]$中的 $k-p$ 个最小元素。并且是排好序的。此外，$L[i]$和 $R[j]$是各自所在数组中，未被复制回数组 A 中的最小元素。

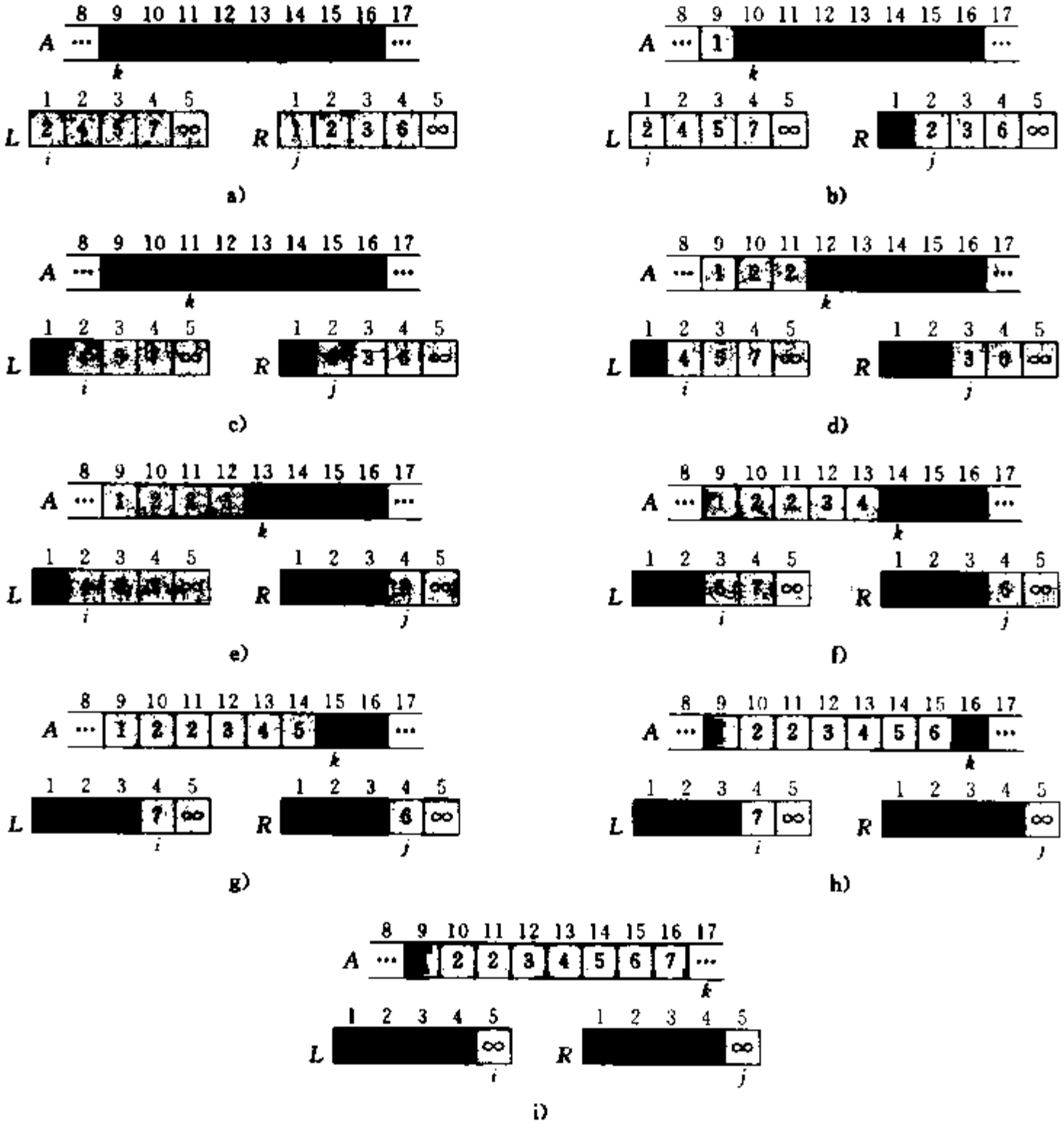


图 2-3　在调用 MERGE(A，9，12，16)的第 10～17 行的操作中，当子数组 $A[9..16]$包含序列(2，4，5，7，1，2，3，6)时的情况。在复制和插入了哨兵后，数组 L 包含了(2，4，5，7，∞)，数组 R 包含了(1，2，3，6，∞)。A 中的浅阴影位置包含了它们的最终值，L 和 R 中的浅阴影位置包含了有待于被复制回 A 中的值。合起来看，浅阴影的位置始终包含了最初在 $A[9..16]$中的值，再加上两个哨兵。A 中的深阴影位置包含了将被覆盖的值，而 L 和 R 中深阴影位置包含了已被复制回 A 中的值。a)～h)在第 12～17 行中循环的每一轮迭代开始之前，数组 A、L 和 R 以及它们各自的下标 k、i 和 j 的情况。i)终止时各数组及其下标的情况。此时，$A[9..16]$中的子数组已排好序了，L 和 R 中的两个哨兵是这两个数组中，仅有的两个未被复制回 A 中的元素

我们必须证明，在第12～17行中for循环的第一轮迭代开始之前，这个循环不变式是成立的，并且，该循环的每一轮迭代都能使这一循环不变式保持成立。在证明循环终止时算法的正确性方面，该循环不变式提供了一个有用的性质。

初始化：在for循环的第一轮迭代开始之前，有$k=p$，因而子数组$A[p..k-1]$是空的。这个空的子数组包含了L和R中$k-p=0$个最小的元素。此外，又因为$i=j=1$，$L[i]$和$R[j]$都是各自所在数组中，尚未被复制回数组A中的最小元素。

保持：为了说明每一轮迭代都能使循环不变式保持成立，首先假设$L[i]\leqslant R[j]$。那么，$L[i]$就是未被复制回数组A的最小元素。由于$A[p..k-1]$包含了$k-p$个最小的元素，因此，在第14行将$L[i]$复制到$A[k]$中后，子数组$A[p..k]$将包含$k-p+1$个最小的元素。增加k的值(在for循环中更新计数器变量的值时)和i的值(在第15行中)，会为下一轮迭代重新建立循环不变式的值。如果这次有$L[i]\geqslant R[j]$，则第16～17行就会执行适当的操作，以使循环不变式保持成立。

终止：在终止时，$k=r+1$。根据循环不变式，子数组$A[p..k-1]$(此时即$A[p..r]$)包含了$L[1..n_1+1]$和$R[1..n_2+1]$中$k-p=r-p+1$个最小元素，并且是已排好序的。数组L和R合起来，包含了$n_1+n_2+2=r-p+3$个元素。除了两个最大的元素外，其余的所有元素都已被复制回数组A中，这两个最大元素都是哨兵。

30
/
31

要理解为什么MERGE过程的运行时间是$\Theta(n)$，此处$n=r-p+1$，请注意第1～3行和第8～11行中的每一行的运行时间都是常量，第4～7行中的for循环所需时间为$\Theta(n_1+n_2)=\Theta(n)$,[⊖]并且，第12～17行的for循环共有$n$轮迭代，其中的每一轮迭代所需时间都是常量。

现在，就可以将MERGE过程作为合并排序中的一个子程序来使用了。下面的过程MERGE-SORT(A，p，r)对子数组$A[p..r]$进行排序。如果$p\geqslant r$，则该子数组中至多只有一个元素，当然就是已排序的。否则，分解步骤就计算出一个下标q，将$A[p..r]$分成$A[p..q]$和$A[q+1..r]$，各含$\lceil n/2\rceil$个元素。[⊖]

```
MERGE-SORT(A, p, r)
1  if p<r
2     then q ← ⌊(p+r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q+1, r)
5          MERGE(A, p, q, r)
```

为了对整个序列$A=(A[1]，A[2]，\cdots，A[n])$进行排序，首先要调用MERGE-SORT($A$，1，$length[A]$)，其中$length[A]=n$。图2-4自底向上地示出了当$n$为2的幂时，整个过程中的操作。算法将两个长度为1的序列合并成已排好序的、长度为2的序列，接着又将长度为2的序列合并成长度为4的有序序列，等等，一直进行到将两个长度为$n/2$的序列合并成最终排好序的、长度为n的序列。

⊖ 我们将在第3章中介绍如何来形式化地解释包含Θ记号的方程。

⊖ $\lceil x\rceil$记号表示大于或等于x的最小整数，$\lfloor x\rfloor$记号表示小于或等于x的最大整数。这两种表示将在第3章中定义。要验证将q置为$\lfloor(p+r)/2\rfloor$，能产生大小分别为$\lceil n/2\rceil$和$\lfloor n/2\rfloor$的子数组$A[p..q]$和$A[q+1..r]$，最简单的方法就是根据p和r为奇数还是偶数，分析四种可能出现的情况。

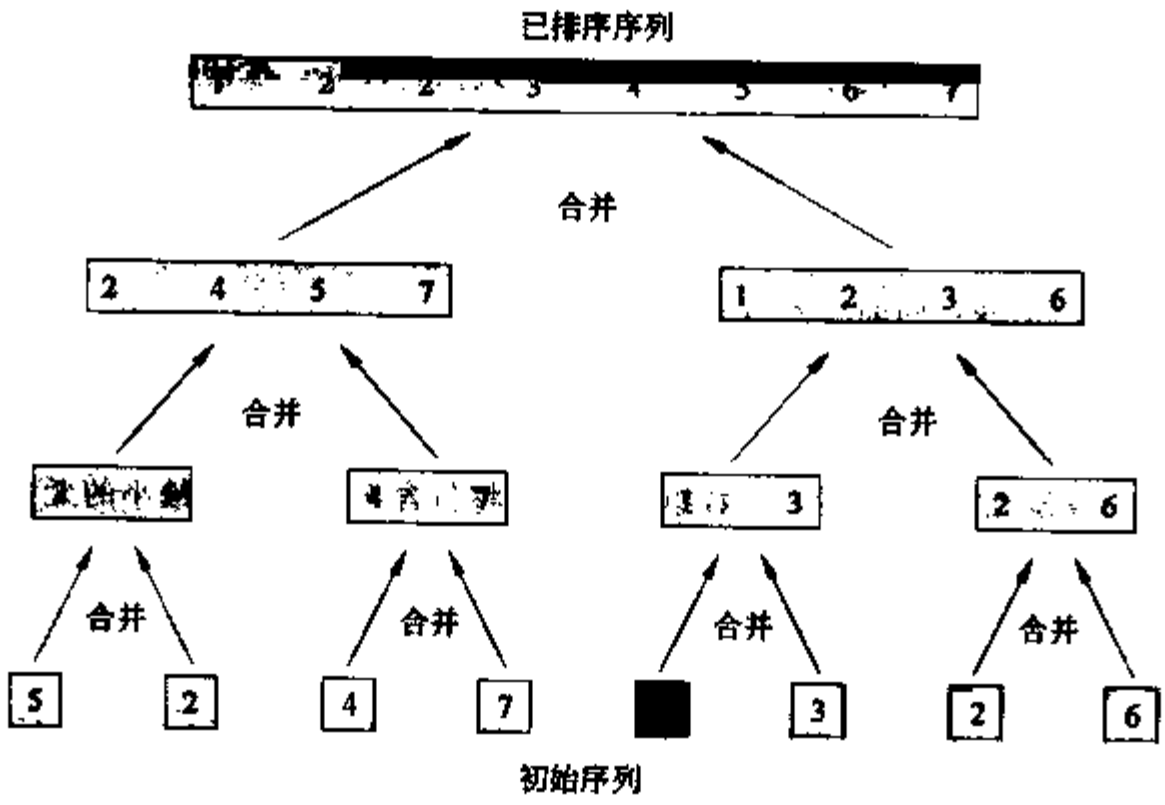


图 2-4　合并排序在数组 $A=(5, 2, 4, 7, 1, 3, 2, 6)$ 上的处理过程。随着算法的处理过程由底向上进展，待合并的已排序序列长度不断增加

2.3.2　分治法分析

当一个算法中含有对其自身的递归调用时，其运行时间可以用一个递归方程(或递归式)来表示。该方程通过描述子问题与原问题的关系，来给出总的运行时间。我们可以利用数学工具来解递归式，并给出算法性能的界。

分治算法中的递归式是基于基本模式中的三个步骤的。如先前一样，设 $T(n)$ 为一个规模为 n 的问题的运行时间。如果问题的规模足够地小，如 $n \leqslant c$(c 为一个常量)，则得到它的直接解的时间为常量，写作 $\Theta(1)$。假设我们把原问题分解成 a 个子问题，每一个的大小是原问题的 $1/b$。(对于合并排序，a 和 b 都是 2，但在许多分治法中，$a \neq b$。)如果分解该问题和合并解的时间各为 $D(n)$ 和 $C(n)$，则得到递归式：

$$T(n) = \begin{cases} \Theta(1) & \text{如果 } n \leqslant c \\ aT(n/b) + D(n) + C(n) & \text{否则} \end{cases}$$

在第 4 章中，将介绍如何解这类常见的递归式。

合并排序算法的分析

32
~
33

MERGE-SORT 的伪代码在元素为奇数个时能正确地工作，而此处我们为了简化对基于递归的算法的分析，就假定原问题的规模是 2 的幂次，这样每一次分解所产生的子序列的长度就恰好为 $n/2$。在第 4 章中，将会看到这一假设不影响递归式解的增长量级。

以下给出递归形式的 $T(n)$ 即最坏情况下合并排序 n 个数的运行时间。合并排序一个元素的时间是个常量。当 $n>1$ 时，将运行时间如下分解：

分解：这一步仅仅是计算出子数组的中间位置，需要常量时间，因而 $D(n)=\Theta(1)$。

解决：递归地解两个规模为 $n/2$ 的子问题，时间为 $2T(n/2)$。

合并：我们已经注意到，在一个含有 n 个元素的子数组上，MERGE 过程的运行时间为 $\Theta(n)$，则 $C(n)=\Theta(n)$。

当我们在合并排序算法的分析中，将函数 $D(n)$ 和 $C(n)$ 相加时，我们是在将一个 $\Theta(1)$ 函数

与另一个 $\Theta(n)$ 函数进行相加。相加的和是 n 的一个线性函数，即 $\Theta(n)$。将它与“解决”步骤中所得的项 $2T(n/2)$ 相加，即得到合并排序的最坏情况运行时间 $T(n)$ 的递归表示：

$$T(n)=\begin{cases}\Theta(1) & \text{如果 } n=1\\ 2T(n/2)+\Theta(n) & \text{如果 } n>1\end{cases} \tag{2.1}$$

第 4 章将介绍“主定理”(master theorem)，它可以用来证明 $T(n)$ 为 $\Theta(n\lg n)$，此处 $\lg n$ 代表 $\log_2 n$。由于对数函数的增长速度比任何线性函数增长得都要慢，因此，当输入规模足够大时，合并排序(其运行时间为 $\Theta(n\lg n)$)在最坏情况下要比插入排序(其运行时间为 $\Theta(n^2)$)好。

此处，无需主定理，也可以直观地理解递归式(2.1)的解为什么会是 $T(n)=\Theta(n\lg n)$。递归式(2.1)重写如下：

$$T(n)=\begin{cases}c & \text{如果 } n=1\\ 2T(n/2)+cn & \text{如果 } n>1\end{cases} \tag{2.2}$$

其中常量 c 代表规模为 1 的问题所需的时间，也是在“解决”和“合并”步骤中处理每个数组元素所需的时间。⊖

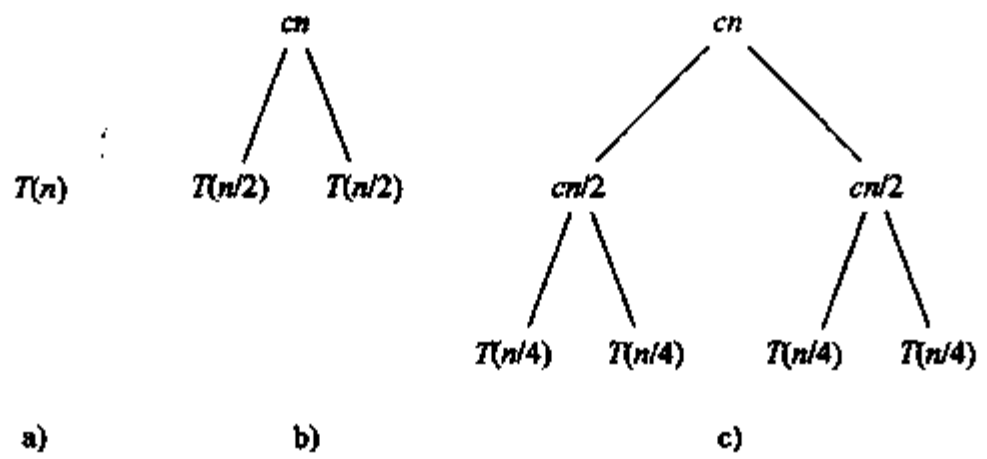
图 2-5 说明了如何解递归式(2.2)。出于方便性的考虑，假设 n 是 2 的整数幂。图 2-5a 图示出 $T(n)$，它在图 2-5b 中被扩展成递归式的一种等价树形表示。cn 项是树根(即顶层递归的代价)，根的两棵子树是两个更小一点的递归式 $T(n/2)$。图 2-5c 示出了这一过程在将 $T(n/2)$ 扩展 34
后的情况。在第二层递归的两个子结点中，每一个结点的代价都是 $cn/2$。继续在树中扩展每个结点，即将其分解成由递归式所决定的各个组成部分，直到问题的规模降到了 1，这时每个问题的代价为 c。图 2-5d 示出了最终的树。

接下来给这棵树的每一层加上代价。最顶层的总代价为 cn，下一层的总代价为 $c(n/2)+c(n/2)=cn$，再往下去一层，总代价为 $c(n/4)+c(n/4)+c(n/4)+c(n/4)=cn$，等等。一般来说，最顶层之下的第 i 层有 2^i 个结点，每一个的代价都是 $c(n/2^i)$，于是，顶层之下的第 i 层的总代价为 $2^i c(n/2^i)=cn$。在最底层，共有 n 个结点，每一个结点的代价为 c，该层的总代价为 cn。

在图 2-5 中，“递归树”中总的层数为 $\lg n+1$。只要做一个非正式的归纳推理，就很容易理解这一事实。$n=1$ 为归纳的基本前提情况，此时递归树中只有一层。由于 $\lg 1=0$，有 $\lg n+1$ 给出了正确的层数。现在，我们进行归纳假设，即假设一棵有 2^i 个结点的递归树中，共有 $\lg 2^i+1=i+1$ 层(由于对任何 i 值，都有 $\lg 2^i=i$)。因为我们假设初始的输入规模是 2 的整数次幂，因而，下一个要考虑的输入规模应该是 2^{i+1}。一棵具有 2^{i+1} 个结点的树比具有 2^i 个结点的树要多一层，因此，其总的层数为 $(i+1)+1=\lg 2^{i+1}+1$。

要计算递归式(2.2)给出的总代价，只要将递归树中各层的代价加起来就可以了。在该树中，总共有 $\lg n+1$ 层，每一层的代价都是 cn，于是，整棵树的总代价就是 $cn(\lg n+1)=cn\lg n+cn$。忽略低阶项和常量 c，即得到结果 $\Theta(n\lg n)$。

⊖ 同一个常量恰好既能表示解决规模为 1 的问题的时间，又能表示解决及合并步骤中处理每个数组元素的时间，这一点一般不太可能。可以这样来绕过这一问题，即设 c 为这两个时间中较大者，从而可以认为递归式给出的是算法运行时间的一个上界。或者，也可以设 c 为这两个时间中的较小者，从而可以认为递归式给出的是算法运行时间的一个下界。这两种界都是 $n\lg n$。将两种情况合在一起，就可以得到运行时间为 $\Theta(n\lg n)$。



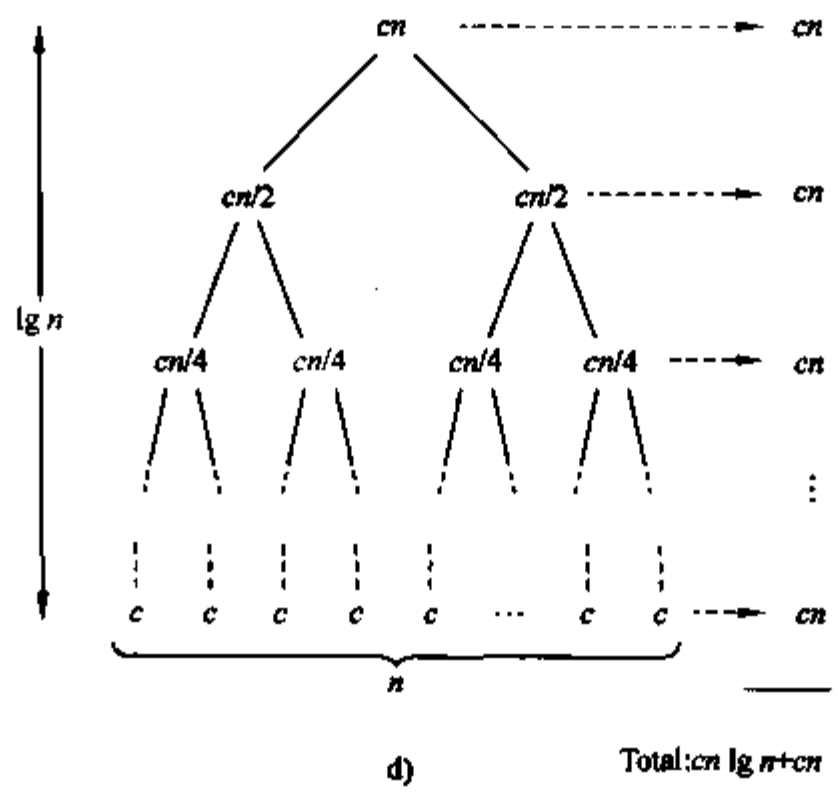


图 2-5　为递归式 $T(n)=2T(n/2)+cn$ 构造一棵递归树，a)图示出了 $T(n)$，它被逐步地在 b)～d)图中进行了扩展以形成递归树。在 d)图中，完全扩展了的递归树共有 $\lg n+1$ 层(亦即，其深度为 $\lg n$，如图所示)，每一层的总代价都是 cn。因此，总的代价就是 $cn\lg n+cn$，为 $\Theta(n\lg n)$

练习

2.3-1　以图 2-4 为模型，说明合并排序在输入数组 $A=(3, 41, 52, 26, 38, 57, 9, 49)$ 上的执行过程。

35
↓
36

2.3-2　改写 MERGE 过程，使之不使用哨兵元素，而是在一旦数组 L 或 R 中的所有元素都被复制回数组 A 后，就立即停止，再将另一个数组中余下的元素复制回数组 A 中。

2.3-3　利用数学归纳法证明：当 n 是 2 的整数次幂时，递归式

$$T(n)=\begin{cases}2 & \text{如果 } n=2\\ 2T(n/2)+n & \text{如果 } n=2^k,\text{对于 } k>1\end{cases}$$

的解为 $T(n)=n\lg n$。

2.3-4　插入排序可以如下改写成一个递归过程：为排序 $A[1..n]$，先递归地排序 $A[1..n-1]$，然后再将 $A[n]$插入到已排序的数组 $A[1..n-1]$中去。对于插入排序的这一递归版本，为它的运行时间写一个递归式。

2.3-5　回顾一下练习 2.1-3 中提出的查找问题，注意如果序列 A 是已排序的，就可以将该序列的中点与 v 进行比较。根据比较的结果，原序列中有一半就可以不用再做进一步的考虑了。二分查找(binary search)就是一个不断重复这一查找过程的算法，它每次都将序列余下的部分分成两半，并只对其中的一半做进一步的查找。写出二分查找算法的伪代码，可以是迭代的，也可以是递归的。说明二分查找算法的最坏情况运行时间为什么是 $\Theta(\lg n)$。

2.3-6　观察一下 2.1 节中给出的 INSERTION-SORT 过程，在第 5～7 行的 while 循环中，采用了一种线性查找策略，在已排序的子数组 $A[1..j-1]$中(反向)扫描。是否可以改用二分查找策略(见练习 2.3-5)，来将插入排序的总体最坏情况运行时间改善至 $\Theta(n\lg n)$?

***2.3-7**　请给出一个运行时间为 $\Theta(n\lg n)$的算法，使之能在给定一个由 n 个整数构成的集合 S 和另一个整数 x 时，判断出 S 中是否存在有两个其和等于 x 的元素。

思考题

2-1　在合并排序中对小数组采用插入排序

尽管合并排序的最坏情况运行时间为 $\Theta(n\lg n)$，插入排序的最坏情况运行时间为 $\Theta(n^2)$，但插入排序中的常数因子使得它在 n 较小时，运行得要更快一些。因此，在合并排序算法中，当子问题足够小时，采用插入排序就比较合适了。考虑对合并排序做这样的修改，即采用插入排序策略，对 n/k 个长度为 k 的子列表进行排序，然后，再用标准的合并机制将它们合并起来，此处 k 是一个待定的值。

a)证明在最坏情况下，n/k 个子列表(每一个子列表的长度为 k)可以用插入排序在 $\Theta(nk)$时间内完成排序。

37

b)证明这些子列表可以在 $\Theta(n\lg(n/k))$最坏情况时间内完成合并。

c)如果已知修改后的合并排序算法的最坏情况运行时间为 $\Theta(nk+n\lg(n/k))$，要使修改后的算法具有与标准合并排序算法一样的渐近运行时间，k 的最大渐近值(即 Θ 形式)是什么(以 n 的函数形式表示)?

d)在实践中，k 的值应该如何选取?

2-2　冒泡排序算法的正确性

冒泡排序(bubblesort)算法是一种流行的排序算法，它重复地交换相邻的两个反序元素。

```
BUBBLESORT(A)
1  for i ← 1 to length[A]
2      do for j ← length[A] downto i+1
3          do if A[j]<A[j-1]
4              then exchange A[j] ↔ A[j-1]
```

a)设 A′表示 BUBBLESORT(A)的输出。为了证明 BUBBLESORT 是正确的，需要证明它能够终止，并且有：

$$A'[1] \leqslant A'[2] \leqslant \cdots \leqslant A'[n] \tag{2.3}$$

其中 $n=length[A]$。为了证明 BUBBLESORT 的确能实现排序的效果，还需要证明什么?

下面两个部分将证明不等式(2.3)。

b)对第 2～4 行中的 for 循环，给出一个准确的循环不变式，并证明该循环不变式是成立的。在证明中应采用本章中给出的循环不变式证明结构。

c)利用在 b)部分中证明的循环不变式的终止条件，为第 1～4 行中的 for 循环给出一个循环不变式，它可以用来证明不等式(2.3)。你的证明应采用本章中给出的循环不变式的证明结构。

[38] d)冒泡排序算法的最坏情况运行时间是什么？比较它与插入排序的运行时间。

2-3　霍纳规则的正确性

以下的代码片段实现了用于计算多项式

$$P(x) = \sum_{k=0}^{n} a_k x^k = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots))$$

的霍纳规则(Horner's rule)。

给定系数 a_0，a_1，…，a_n 以及 x 的值，有：

```
1  y ← 0
2  i ← n
3  while i≥0
4       do y ← a_i + x · y
5          i ← i−1
```

a)这一段实现霍纳规则的代码的渐近运行时间是什么？

b)写出伪代码以实现朴素多项式求值(naive polynomial-evaluation)算法，它从头开始计算多项式的每一个项。这个算法的运行时间是多少？它与实现霍纳规则的代码段的运行时间相比怎样？

c)证明以下给出的是针对第 3～5 行中 while 循环的一个循环不变式：

在第 3～5 行中 while 循环每一轮迭代的开始，有：

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

不包含任何项的和视为等于 0。你的证明应遵循本章中给出的循环不变式的证明结构，并应证明在终止时，有 $y = \sum_{k=0}^{n} a_k x^k$。

d)最后证明以上给出的代码片段能够正确地计算由系数 a_0，a_1，…，a_n 刻划的多项式。

2-4　逆序对

设 $A[1..n]$是一个包含 n 个不同数的数组。如果在 $i<j$ 的情况下，有 $A[i]>A[j]$，则 (i, j)就称为 A 中的一个*逆序对*(inversion)。

a)列出数组〈2，3，8，6，1〉的 5 个逆序。

b)如果数组的元素取自集合{1，2，…，n}，那么，怎样的数组含有最多的逆序对？它

[39] 包含多少个逆序对？

c)插入排序的运行时间与输入数组中逆序对的数量之间有怎样的关系？说明你的理由。

d)给出一个算法，它能用 $\Theta(n \lg n)$的最坏情况运行时间，确定 n 个元素的任何排列中逆序对的数目。(提示：修改合并排序)

本章注记

1968 年，Knuth 发表了三卷题为《计算机程序设计艺术》(The Art of Computer Programming)[182，183，185]的著作中的第一卷。第一卷开创了现代计算机算法研究，并侧重于对算法运行

时间的分析。对于本书中涉及的许多主题，这三卷著作都始终是有吸引力的和有价值的参考书。根据 Knuth 的说法，“算法”(algorithm)一词源自名字“al Khowârizmî”，这是一位 9 世纪的波斯数学家。

Aho，Hopcroft 和 Ullman[5]主张将算法的渐近分析作为比较不同算法间相对性能的一种方法。他们还使利用递归关系来刻划递归算法运行时间的做法变得流行起来。

Knuth[185]以百科全书似的方式，分析了多种排序算法。在他对各种排序算法的比较(16.2 节)中，包括了精确的、比较执行步数这样的分析，类似我们这儿对插入排序所做的分析一样。Knuth 对插入排序的讨论包括了这一算法的几种变形，其中最重要的是 Shell 排序算法，由 D. L. Shell 提出，它对输入序列的周期性子序列采用插入排序，从而可以得到一种更快的排序算法。

Knuth 也对合并排序算法进行了分析。他提到了在 1938 年，有人发明了一种机械式校对机，它在一趟之内，就可以将两堆穿孔卡片合并成一堆。J. von Neumann 是计算机科学的先驱之一，他于 1945 年在 EDVAC 机上，特意为合并排序编写了一个程序。

Gries[133]介绍了程序正确性证明的早期历史，他提到了 P. Naur 在这一领域内的第一篇论文，并指出循环不变式是由 R. W. Floyd 提出的。Mitchell[222]编写的教材中介绍了程序正确性证明方面一些更新的进展。

第 3 章　函数的增长

第 2 章中定义了算法运行时间增长的阶，它给出算法效率的简明特征，并且可以用来比较各种算法的相对性能。例如，合并排序的最坏情况运行时间为 $\Theta(n\lg n)$，当输入的规模 n 足够大时，就要优于最坏情况运行时间为 $\Theta(n^2)$的插入排序。虽然有时候能够精确地确定一个算法的运行时间，如我们在第 2 章中对插入排序的运行时间所做的分析那样，但通常没必要花那么大的力气去算出额外的精确度。对于足够大的输入规模来说，在精确表示的运行时间中，常系数和低阶项是由输入规模所决定的。

当输入规模大到使只有运行时间的增长量级有关时，就是在研究算法的渐近效率。也就是说，从极限角度看，我们只关心算法运行时间如何随着输入规模的无限增长而增长。通常，对不是很小的输入规模而言，从渐近意义上说更有效的算法是最佳的选择。

本章将介绍几种标准方法来简化算法的渐近分析。下一节首先要定义几种渐近记号，其中的 Θ 记号我们已经见过了。然后，给出全书使用的几种记号约定。最后，还要回顾一下在算法分析中经常见到的一些函数的性质。

3.1　渐近记号

用来表示算法的渐近运行时间的记号是用定义域为自然数集 $\mathbf{N}=\{0, 1, 2, \cdots\}$的函数来定义的。这些记号便于用来表示最坏情况运行时间 $T(n)$，因为 $T(n)$一般仅定义于整数的输入规模

上。有时候，以某些不同的方式越规使用这些记号也是很方便的。例如，这些记号的定义域可以很容易地扩充至实数域或缩小到自然数的某个受限子集上。但要注意搞清楚这些表示法的准确含义，方能做到越规使用而不误用。本节定义几种基本的渐近记号和一些常见的扩充用法。

Θ 记号

在第 2 章中，我们知道插入排序的最坏情况下运行时间是 $T(n)=\Theta(n^2)$。现在来定义这种记号的含义。对一个给定的函数 $g(n)$，用 $\Theta(g(n))$来表示函数集合：

$\Theta(g(n))=\{f(n)$：存在正常数 c_1，c_2 和 n_0，使对所有的 $n\geqslant n_0$，有 $0\leqslant c_1g(n)\leqslant f(n)\leqslant c_2g(n)\}$[⊖]对任一个函数 $f(n)$，若存在正常数 c_1，c_2，使当 n 充分大时，$f(n)$能被夹在 $c_1g(n)$和 $c_2g(n)$中间，则 $f(n)$属于集合 $\Theta(g(n))$。因为 $\Theta(g(n))$是一个集合，可以写成“$f(n)\in\Theta(g(n))$”表示 $f(n)$是 $\Theta(g(n))$的元素。不过，通常写成“$f(n)=\Theta(g(n))$”来表示相同的意思。这种用于表示集合成员关系的等号越规使用初看起来可能有点令人迷惑，但在本节稍候就会看到它的好处了。

图 3-1a 给出函数 $f(n)$与 $g(n)$的直观图示，其中 $f(n)=\Theta(g(n))$。对所有位于 n_0 右边的 n 值，$f(n)$的值落在 $c_1g(n)$和 $c_2g(n)$之间。换句话说，对所有的 $n\geqslant n_0$，$f(n)$在一个常因子范围内与 $g(n)$相等。我们说 $g(n)$是 $f(n)$的一个渐近确界。

$\Theta(g(n))$的定义需要每个成员 $f(n)\in\Theta(g(n))$都是渐近非负，也就是说当 n 足够大时 $f(n)$是非负值(渐近正函数则是当 n 足够大时总为正值)。这就要求函数 $g(n)$本身也必须是渐近非负的，否则集合 $\Theta(g(n))$就是空集。因此，假定 Θ 记号中用到的每个函数都是渐近非负的。这个假设对本章中定义的其他渐近记号也是成立的。

⊖ 在集合表示法中，“:”应读作“满足……的特性”。

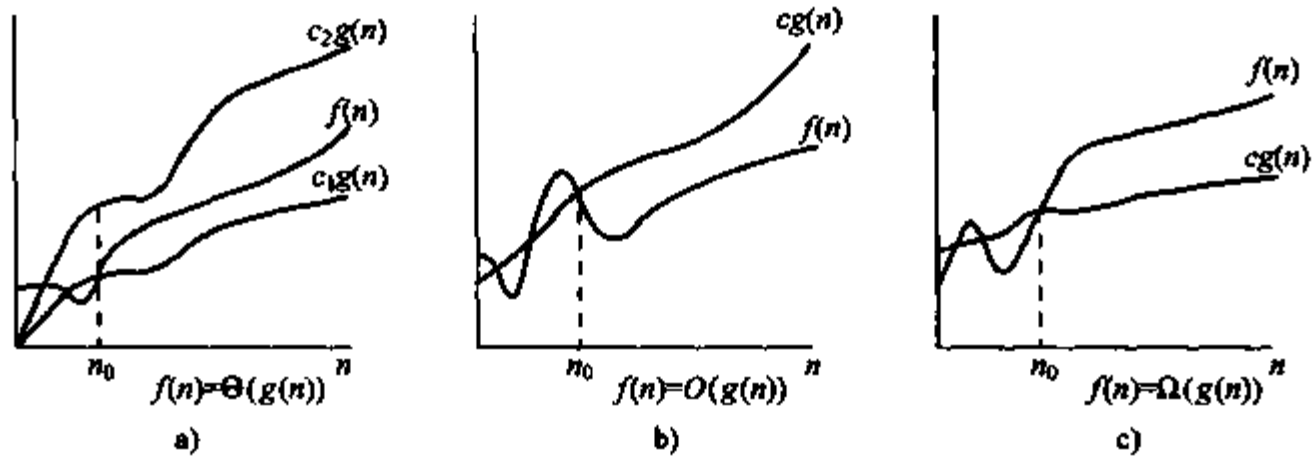


图 3-1　记号 Θ，O 和 Ω 的图例。在每个部分中，n_0 是最小的可能值；大于 n_0 的值也有效。a)Θ 记号限制一个函数在常数因子内。如果存在正常数 n_0，c_1 和 c_2 使得在 n_0 右边 $f(n)$ 的值永远在 $c_1g(n)$ 与 $c_2g(n)$ 之间，那么可以写成 $f(n)=\Theta(g(n))$。b)O 记号给出一个函数在常数因子内的上限。如果存在正常数 n_0 和 c 使得在 n_0 右边 $f(n)$ 的值永远等于或小于 $cg(n)$，那么可以写成 $f(n)=O(g(n))$。c)Ω 符号给出一个函数在常数因子内的下限。如果存在正常数 n_0 和 c 使得在 n_0 右边 $f(n)$ 的值永远等于或大于 $cg(n)$，那么可以写成 $f(n)=\Omega(g(n))$

在第 2 章中，引进了非正式的 Θ 记号，其效果相当于舍弃了低阶项和忽略了最高阶项的系数。为了说明这一点，下面利用正式定义来证明 $\frac{1}{2}n^2-3n=\Theta(n^2)$。首先要确定常数 c_1，c_2 和 n_0，使得对所有的 $n\geqslant n_0$，有

$$c_1n^2\leqslant\frac{1}{2}n^2-3n\leqslant c_2n^2$$

成立。用 n^2 除不等式得

$$c_1\leqslant\frac{1}{2}-\frac{3}{n}\leqslant c_2$$

右边的不等式在 $c_2\geqslant 1/2$ 时对所有的 $n\geqslant 1$ 成立。同样，左边的不等式可以让 $c_1\leqslant 1/14$ 时对所有的 $n\geqslant 7$ 成立。这样，通过选择 $c_1=1/14$，$c_2=1/2$，以及 $n_0=7$，可以证明 $\frac{1}{2}n^2-3n=\Theta(n^2)$。当然，还存在其他的常数选择，而重要的是确实存在某个选择。注意这些常数依赖于函数 $\frac{1}{2}n^2-3n$；$\Theta(n^2)$ 中不同的函数通常需要不同的常数。

我们还可以使用正式定义来证明 $6n^3\neq\Theta(n^2)$。为引出矛盾，假设存在常数 c_2 和 n_0，使对所有的 $n\geqslant n_0$，有 $6n^3\leqslant c_2n^2$；这样就有 $n\leqslant c_2/6$，这对任意大的 n 是不可能成立的，因为 c_2 是常数。

42
~
43

从直观上看，一个渐近正函数中的低阶项在决定渐近确界时可以被忽略，因为当 n 很大时它们就相对地不重要了。最高阶项很小的一部分就足以超越所有的低阶项。因此，若将 c_1 置成略小于最高阶项系数的值，而将 c_2 置成稍大于最高阶项系数的值，就可以满足 Θ 记号定义中的不等式。同样，最高阶项的系数也可以忽略，因为它只是将 c_1 和 c_2 改变成等于该系数的常数因子。

例如，考虑一个任意的二次函数 $f(n)=an^2+bn+c$，其中 a，b 和 c 都是常数，且 $a>0$。舍掉低阶项并忽略常数项就得出 $f(n)=\Theta(n^2)$。为了形式地说明相同的结果，取常数 $c_1=a/4$，$c_2=7a/4$ 以及 $n_0=2\cdot\max((|b|/a),\sqrt{(|c|/a)})$。读者可以证明，对所有的 $n\geqslant n_0$，$0\leqslant c_1n^2\leqslant an^2+bn+c\leqslant c_2n^2$ 成立。一般来说，对任何一个多项式 $p(n)=\sum_{i=0}^{d}a_in^i$，其中 a_i 是常数并且

$a_d>0$，有 $p(n)=\Theta(n^d)$（见思考题 3-1）。

因为任意一个常数都是 0 次的多项式，故可以把任何常函数表示成 $\Theta(n^0)$ 或 $\Theta(1)$。后一种表示略有点越规使用了，因为从中看不出什么变量趋于无穷。[⊖] 我们将经常使用记号 $\Theta(1)$ 表示一个常数或某变量的常函数。

O 记号

Θ 记号渐近地给出一个函数的上界和下界。当只有渐近上界时，使用 O 记号。对一个函数 $g(n)$，用 $O(g(n))$ 表示一个函数集合

$$O(g(n))=\{f(n)：\text{存在正常数 } c \text{ 和 } n_0，\text{使对所有的 } n\geqslant n_0，\text{有 } 0\leqslant f(n)\leqslant cg(n)\}$$

读作 $g(n)$ 的大 O。O 记号在一个常数因子内给出某函数的一个上界。图 3-1b 说明 O 记号的直观意义。对所有位于 n_0 右边的 n 值，函数 $f(n)$ 的值在 $g(n)$ 之下。

为表示一个函数 $f(n)$ 是集合 $O(g(n))$ 的一个元素，记 $f(n)=O(g(n))$。注意 $f(n)=\Theta(g(n))$ 隐含着 $f(n)=O(g(n))$，因为 Θ 记号强于 O 记号。按集合论中的写法，有 $\Theta(g(n))\subseteq O(g(n))$。我们已经证明任意的二次函数 an^2+bn+c（其中 $a>0$）属于 $\Theta(n^2)$，这也就证明任意二次函数也属于 $O(n^2)$。更令人惊讶的是任一个线性函数 $an+b$ 也在 $O(n^2)$ 中，这可由设 $c=a+|b|$ 和 $n_0=1$ 来证明。

44

有些式子如 $n=O(n^2)$，对某些曾见过 O 记号的读者来说可能有些奇怪。在文献中，O 记号有时会被非正式地用来描述渐近确界，而这在前面是用 Θ 记号来定义的。在本书中，当我们写 $f(n)=O(g(n))$ 时，只是说明 $g(n)$ 的某个常数倍是 $f(n)$ 的渐近上界，而不反映该上界如何接近。现今有关算法的文献中，都将渐近上界与渐近确界加以区分。

利用 O 记号，我们常常能通过查看算法的总体结构来描述算法的运行时间。例如，在第 2 章中，插入排序算法中的二重循环结构立即能引出其最坏情况运行的一个上界为 $O(n^2)$：内循环每一轮迭代的代价以 $O(1)$（常量）为上界；下标 i 和 j 最大可以取到 n；对于 n^2 个 i 值和 j 值对中的每一对，内循环至多只执行一次。

O 记号是用来表示上界的，当用它作为算法的最坏情况运行时间的上界时，就有对任意输入的运行时间的上界。因此，插入排序在最坏情况下运行时间的上界 $O(n^2)$ 也适用于每个输入的运行时间。但是，插入排序最坏情况运行时间的界 $\Theta(n^2)$ 并不是对每种输入都适用。例如，在第 2 章已经看到，当输入已经排好序时，插入排序的运行时间为 $\Theta(n)$。

从技术上看，说插入排序的运行时间是 $O(n^2)$ 有点不合适，因为对给定的输入规模 n，实际运行时间与具体输入有关。当我们说“运行时间是 $O(n^2)$”时，是指存在一个 $O(n^2)$ 的函数 $f(n)$，对于任意的 n，运行时间的界是 $f(n)$，而与具体的输入无关。这也就是说最坏情况下运行时间是 $O(n^2)$。

Ω 记号

正如 O 记号给出一个函数的渐近上界，Ω 记号给出函数的渐近下界。给定一个函数 $g(n)$，用 $\Omega(g(n))$ 表示一个函数集合

$$\Omega(g(n))=\{f(n)：\text{存在正常数 } c \text{ 和 } n_0，\text{使对所有的 } n\geqslant n_0，\text{有 } 0\leqslant cg(n)\leqslant f(n)\}$$

读作 $g(n)$ 的大 Ω。图 3-1c 说明了 Ω 记号的直观意义。对所有在 n_0 右边的 n 值，函数 $f(n)$ 的数值等于或大于 $cg(n)$。

⊖ 真正的问题在于，常见的函数表示法并不将函数与值区别开来。在 λ 演算中，函数的参数是明确指定的：函数 n^2 可被写作 $\lambda n.n^2$，或者甚至是 $\lambda r.r^2$。但是，采用更为严格的表示法会使得代数运算变得复杂化，因此我们选择容忍了这种活用。

根据到目前为止我们所见过的各渐近记号的定义，容易证明下面重要的定理(见练习 3.1-5)。 45

定理 3.1　对任意两个函数 $f(n)$和 $g(n)$，$f(n)=\Theta(g(n))$当且仅当 $f(n)=O(g(n))$和 $f(n)=\Omega(g(n))$。 ■

作为应用本定理的一个例子，证明 $an^2+bn+c=\Theta(n^2)$(其中 a，b，c 为常数且 $a>0$)就立即能得出 $an^2+bn+c=\Omega(n^2)$和 $an^2+bn+c=O(n^2)$。实际上在应用定理 3.1 时，一般不是像本例的做法这样用渐近确界导出渐近上界和渐近下界，而是用渐近上界和渐近下界证明出渐近确界。

因为 Ω 记号描述了渐近下界，当它用来对一个算法最佳情况运行时间限界时，也隐含给出了在任意输入下运行时间的界。例如，插入排序的最佳情况运行时间是 $\Omega(n)$，这隐含着该算法的运行时间是 $\Omega(n)$。

插入排序的运行时间介于 $\Omega(n)$和 $O(n^2)$之间，因为它处于 n 的线性函数和二次函数的范围内。更进一步，这两个界从渐近意义上来说是尽可能紧确的：例如，插入排序的运行时间不是 $\Omega(n^2)$，因为存在一个输入(例如，当输入已经排好序时)，使得插入排序的运行时间为 $\Theta(n)$。然而，我们说该算法的最坏情况运行时间为 $\Omega(n^2)$，两者并不矛盾，因为存在一个输入，使得算法的运行时间为 $\Omega(n^2)$。当我们说一个算法的运行时间(无修饰语)是 $\Omega(g(n))$时，是指对每一个 n 值，无论取该规模下什么样的输入，该输入上的运行时间都至少是一个常数乘上 $g(n)$(当 n 足够大时)。

等式和不等式中的渐近记号

我们已经见过了渐近记号在数学公式中的应用。例如，在前面介绍 O 记号时，有"$n=O(n^2)$"。我们还可以写 $2n^2+3n+1=2n^2+\Theta(n)$。那么如何解释这些公式呢？

当渐近记号只出现在等式(或不等式)的右边，如 $n=O(n^2)$，我们已经定义过了等号表示集合的成员关系：$n\in O(n^2)$。但一般来说，当渐近符号出现在某个公式中时，我们将其解释为一个不在乎其名称的匿名函数。例如，公式 $2n^2+3n+1=2n^2+\Theta(n)$即表示 $2n^2+3n+1=2n^2+f(n)$，其中 $f(n)$是某个属于集合 $\Theta(n)$的函数。在本例中，$f(n)=3n+1$，确实在 $\Theta(n)$中。

渐近记号的这种用法有助于略去一个等式中无关紧要的细节。例如，在第 2 章中，我们将合并排序的最坏情况运行时间表示为递归式

$$T(n) = 2T(n/2) + \Theta(n) \qquad 46$$

如果仅对 $T(n)$的渐近行为感兴趣，则没有必要写出所有低阶项，它们都被包含在由$\Theta(n)$表示的匿名函数中了。

一个表达式中的匿名函数的个数与渐近记号出现的次数是一致的。例如，在表达式

$$\sum_{i=1}^{n} O(i)$$

中，只有一个匿名函数(一个 i 的函数)。这个表达式与 $O(1)+O(2)+\cdots+O(n)$不同，后者没有明确的含义。

有时，渐近记号出现在等式的左边，例如

$$2n^2 + \Theta(n) = \Theta(n^2)$$

这时，我们使用以下规则来解释这种等式：*无论等号左边的匿名函数如何选择，总有办法选取等号右边的匿名函数使等式成立。*这样，上例的含义及对任意函数 $f(n)\in\Theta(n)$，存在函数 $g(n)\in\Theta(n^2)$，使对所有的 n，$2n^2+f(n)=g(n)$成立。换言之，等式右边提供了较左边更少的细节。

一组这样的关系可以链起来，例如

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$$

我们可以用上述的规则对每一个等式分别解释。第一个等式说明存在函数 $f(n)\in\Theta(n)$，使对所有的 n 有 $2n^2+3n+1=2n^2+f(n)$。第二个等式说明对任意函数 $g(n)\in\Theta(n)$（例如前述的 $f(n)$），存在某个函数 $h(n)\in\Theta(n^2)$，使对所有的 n 有 $2n^2+g(n)=h(n)$。注意这个解释隐含着 $2n^2+3n+1=\Theta(n^2)$，也就是等式链接的直观结论。

o 记号

O 记号所提供的渐近上界可能是也可能不是渐近紧确的。界 $2n^2=O(n^2)$ 是渐近紧确的，但
47 $2n=O(n^2)$ 却不是的。我们使用 o 记号来表示非渐近紧确的上界。$o(g(n))$ 的形式定义为集合

$o(g(n))=\{f(n)$：对任意正常数 c，存在常数 $n_0>0$，使对所有的 $n\geqslant n_0$，有 $0\leqslant f(n)\leqslant cg(n)\}$

例如，$2n=o(n^2)$，但是 $2n^2\neq o(n^2)$。

O 记号与 o 记号的定义是类似的。主要区别在于对 $f(n)=O(g(n))$，界 $0\leqslant f(n)\leqslant cg(n)$ 对某个常数 $c>0$ 成立；但对 $f(n)=o(g(n))$，界 $0\leqslant f(n)\leqslant cg(n)$ 对所有常数 $c>0$ 成立。从直觉上看，在 o 表示中当 n 趋于无穷时，函数 $f(n)$ 相对于 $g(n)$ 来说就不重要了，即

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0 \tag{3.1}$$

某些作者将这个极限作为 o 记号的定义；本书中给出的定义同样也限定匿名函数必须是渐近非负的。

ω 记号

ω 记号与 Ω 记号的关系就好像 o 记号与 O 记号的关系一样。我们用 ω 记号来表示非渐近紧确的下界。它的一种定义为

$$f(n)\in\omega(g(n))\text{ 当且仅当 }g(n)\in o(f(n))$$

$\omega(g(n))$ 的形式定义为集合

$\omega(g(n))=\{f(n)$：对任意正常数 $c>0$，存在常数 $n_0>0$，使对所有的 $n\geqslant n_0$，有 $0\leqslant cg(n)<f(n)\}$。

例如，$n^2/2=\omega(n)$，但 $n^2/2\neq\omega(n^2)$。关系 $f(n)=\omega(g(n))$ 意味着

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$$

如果这个极限存在。也就是说当 n 趋于无穷时，$f(n)$ 相对 $g(n)$ 来说变得任意大了。

函数间的比较

48 实数的许多关系属性可以应用于渐近比较。下面假设 $f(n)$ 和 $g(n)$ 是渐近正值函数。

传递性：

$f(n)=\Theta(g(n))$ 和 $g(n)=\Theta(h(n))$	蕴含 $f(n)=\Theta(h(n))$
$f(n)=O(g(n))$ 和 $g(n)=O(h(n))$	蕴含 $f(n)=O(h(n))$
$f(n)=\Omega(g(n))$ 和 $g(n)=\Omega(h(n))$	蕴含 $f(n)=\Omega(h(n))$
$f(n)=o(g(n))$ 和 $g(n)=o(h(n))$	蕴含 $f(n)=o(h(n))$
$f(n)=\omega(g(n))$ 和 $g(n)=\omega(h(n))$	蕴含 $f(n)=\omega(h(n))$

自反性：

$$f(n)=\Theta(f(n))\quad f(n)=O(f(n))\quad f(n)=\Omega(f(n))$$

对称性：

$$f(n)=\Theta(g(n))\text{ 当且仅当 }g(n)=\Theta(f(n))$$

转置对称性：

$$f(n) = O(g(n)) \text{ 当且仅当 } g(n) = \Omega(f(n))$$
$$f(n) = o(g(n)) \text{ 当且仅当 } g(n) = \omega(f(n))$$

因为这些性质对渐近记号也成立，我们可以将两个函数 f 与 g 的渐近比较和两个实数 a 与 b 的比较作一类比：

$$\begin{aligned} f(n) &= O(g(n)) \quad \approx \quad a \leqslant b \\ f(n) &= \Omega(g(n)) \quad \approx \quad a \geqslant b \\ f(n) &= \Theta(g(n)) \quad \approx \quad a = b \\ f(n) &= o(g(n)) \quad \approx \quad a < b \\ f(n) &= \omega(g(n)) \quad \approx \quad a > b \end{aligned}$$

如果 $f(n)=o(g(n))$，则 $f(n)$ 比 $g(n)$ 渐近较小；如果 $f(n)=\omega(g(n))$，则 $f(n)$ 比 $g(n)$ 渐近较大。

不过，实数集上有一个属性却不能应用到渐近记号上：

三分性：对两个实数 a 和 b，下列三种情况恰有一种成立：$a<b$，$a=b$，或 $a>b$。 49

虽然任何两个实数都可以做比较，但并不是所有的函数都是渐近可比较的。亦即，对于两个函数 $f(n)$ 和 $g(n)$，可能 $f(n)=O(g(n))$ 和 $f(n)=\Omega(g(n))$ 都不成立。例如，函数 n 和 $n^{1+\sin n}$ 无法利用渐近记号来比较，因为 $n^{1+\sin n}$ 中的指数值在 0 与 2 之间变化。

练习

3.1-1　设 $f(n)$ 与 $g(n)$ 都是渐近非负函数。利用 Θ 记号的基本定义来证明 $\max(f(n), g(n)) = \Theta(f(n)+g(n))$。

3.1-2　证明对任意实常数 a 和 b，其中 $b>0$，有

$$(n+a)^b = \Theta(n^b) \tag{3.2}$$

3.1-3　解释为什么“算法 A 的运行时间至少是 $O(n^2)$”这句话是无意义的。

3.1-4　$2^{n+1}=O(2^n)$ 成立吗？$2^{2n}=O(2^n)$ 成立吗？

3.1-5　证明定理 3.1。

3.1-6　证明：一个算法的运行时间是 $\Theta(g(n))$ 当且仅当其最坏情况运行时间是 $O(g(n))$，且最佳情况运行时间是 $\Omega(g(n))$。

3.1-7　证明 $o(g(n)) \cap \omega(g(n))$ 是空集。

3.1-8　可以将我们的表示法扩展到有两个参数 n 和 m 的情形，其中 n 和 m 的值可以以不同的速率，互相独立地趋于无穷。对给定的函数 $g(n, m)$，$O(g(n, m))$ 为函数集

$O(g(n, m)) = \{f(n, m)$：存在正整数 c，n_0 和 m_0，使对所有 $n \geqslant n_0$ 及 $m \geqslant m_0$，有 $0 \leqslant f(n, m) \leqslant cg(n, m)\}$。

给出对应的 $\Omega(g(n, m))$ 和 $\Theta(g(n, m))$ 的定义。 50

3.2　标准记号和常用函数

本节回顾一些标准的数学函数和记号，并给出它们之间的关系。另外，还要举例说明渐近记号的用法。

单调性

一个函数 $f(n)$ 是单调递增的，若 $m \leqslant n$ 蕴含 $f(m) \leqslant f(n)$。类似地，函数 $f(n)$ 是单调递减

的，若 $m \leqslant n$ 蕴含 $f(m) \geqslant f(n)$。函数 $f(n)$ 是严格递增的，若 $m < n$ 蕴含 $f(m) < f(n)$。函数 $f(n)$ 是严格递减的，若 $m < n$ 蕴含 $f(m) > f(n)$。

下取整(floor)和上取整(ceiling)

对任一个实数 x，小于或等于 x 的最大整数表示为 $\lfloor x \rfloor$（读作 x 的下取整），大于或等于 x 的最小整数表示为 $\lceil x \rceil$（读作 x 的上取整）。对于所有实数 x，

$$x - 1 < \lfloor x \rfloor \leqslant x \leqslant \lceil x \rceil < x + 1 \tag{3.3}$$

对于任何整数 n，

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$$

对任意实数 $n \geqslant 0$ 和整数 a，$b > 0$，

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil \tag{3.4}$$

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor \tag{3.5}$$

$$\lceil a/b \rceil \leqslant (a + (b-1))/b \tag{3.6}$$

$$\lfloor a/b \rfloor \geqslant ((a - (b-1))/b \tag{3.7}$$

下取整函数 $f(x) = \lfloor x \rfloor$ 是单调递增的，上取整函数 $f(x) = \lceil x \rceil$ 亦然。

取模运算(modular arithmetic)

对任意整数 a 和任意正整数 n，$a \bmod n$ 的值即 a/n 的余数：

$$a \bmod n = a - \lfloor a/n \rfloor n \tag{3.8}$$

51 给定一个整数除以另一个整数的余数的良定义后，可以方便地引入表示余数相等性的特殊记号。如果 $(a \bmod n) = (b \bmod n)$，则可以写作 $a \equiv b \pmod n$，并称在模 n 时，a 等于 b。换言之，如果 a 与 b 在被 n 除时有相同的余数，则 $a \equiv b \pmod n$。等价地，$a \equiv b \pmod n$ 当且仅当 n 是 $b - a$ 的一个约数。若模数为 n 时 a 不等于 b，则写作 $a \not\equiv b \pmod n$。

多项式

给定一个正整数 d，n 的 d 次多项式是具有如下形式的函数 $p(n)$：

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

其中常数 a_0，a_1，…，a_d 是多项式的系数，且 $a_d \neq 0$。一个多项式是渐近正的，当且仅当 $a_d > 0$。对于一个 d 次的渐近正多项式 $p(n)$，有 $p(n) = \Theta(n^d)$。对任意实常数 $a \geqslant 0$，函数 n^a 单调递增；对 $a \leqslant 0$，函数 n^a 单调递减。若对于某个常数 k 有 $f(n) = O(n^k)$，则称函数 $f(n)$ 有多项式界。

指数式

对任意实数 $a > 0$、m 和 n，有下列恒等式：

$$a^0 = 1, \quad a^1 = a, \quad a^{-1} = 1/a$$

$$(a^m)^n = a^{mn}, \quad (a^m)^n = (a^n)^m, \quad a^m a^n = a^{m+n}$$

对任意 n 和 $a \geqslant 1$，函数 a^n 关于 n 单调递增；方便的情况下，我们将假设 $0^0 = 1$。

多项式和指数式的增长率可通过下列事实相关联。对任意实数 a 和 b，且 $a > 1$，

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 \tag{3.9}$$

根据此式可得出

$$n^b = o(a^n)$$

52 因此，任何底大于 1 的指数函数比任何多项式函数增长得更快。

用 e 表示 2.718 28…，即自然对数函数的底，对任意实数 x

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{3.10}$$

其中"!"表示阶乘函数，本节稍后将定义它。对任意实数 x，有不等式

$$e^x \geqslant 1 + x \tag{3.11}$$

只有当 $x=0$ 时等号才成立。当 $|x| \leqslant 1$，有近似式：

$$1 + x \leqslant e^x \leqslant 1 + x + x^2 \tag{3.12}$$

当 $x \to 0$ 时，用 $1+x$ 来近似 e^x 的效果相当好：

$$e^x = 1 + x + \Theta(x^2)$$

(在这个等式中，渐近记号是用来描述当 $x \to 0$ 而不是 $x \to \infty$ 时的极限行为)。对任意的 x，有

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x \tag{3.13}$$

对数

我们将用到下列记号：

$$\begin{aligned} \lg n &= \log_2 n && \text{(以 2 为底的对数)} \\ \ln n &= \log_e n && \text{(自然对数)} \\ \lg^k n &= (\lg n)^k && \text{(取幂)} \\ \lg\lg n &= \lg(\lg n) && \text{(复合)} \end{aligned}$$

我们将采用一个重要的记号约定，即对数函数仅作用于公式中的下一项，因此 $\lg n + k$ 就表示 $(\lg n) + k$ 而不是 $\lg(n+k)$。如果常数 b 大于 1，那么函数 $\log_b n$ 关于 $n > 0$ 严格递增。

对任意的实数 $a>0$，$b>0$，$c>0$ 和 n，

$$a = b^{\log_b a}, \quad \log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a, \quad \log_b a = \frac{\log_c a}{\log_c b} \tag{3.14}$$ 53

$$\log_b(1/a) = -\log_b a, \quad \log_b a = \frac{1}{\log_b b}, \quad a^{\log_b c} = c^{\log_b a} \tag{3.15}$$

在上面每个公式中，对数的底不为 1。

由公式(3.14)，改变一个对数的底只是把对数的值改变了一个常数倍，所以当不在意这些常数因子时我们将经常采用"$\lg n$"记号，就像 O 记号一样。计算机工作者常常认为对数的底取 2 最自然，因为很多算法和数据结构都涉及到对问题进行二分。

当 $|x| < 1$ 时，$\ln(1+x)$ 的一个简单级数展开为：

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots$$

当 $x > -1$ 时，还有以下不等式成立：

$$\frac{x}{1+x} \leqslant \ln(1 + x) \leqslant x \tag{3.16}$$

只有当 $x=0$ 时等号才成立。

如果对常数 k，函数 $f(n) = O(\lg^k n)$，则称 $f(n)$ 是多项对数有界的(polylogarithmically bounded)。在公式(3.9)中，通过用 $\lg n$ 替代 n 和用 2^a 替代 a，可以将多项式的增长率和多项对数的增长率联系起来：

$$\lim_{n \to \infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n \to \infty} \frac{\lg^b n}{n^a} = 0$$

由此极限式可以得出结论：对于任意常数 $a>0$，有：

$$\lg^b n = o(n^a)$$

因此，任意正的多项式函数都比多项对数函数增长得快。

阶乘

记号 $n!$（读作 n 的阶乘）定义为对所有整数 $n\geqslant 0$，

$$n!\begin{cases}1 & \text{如果 } n=0\\ n\cdot(n-1)! & \text{如果 } n>0\end{cases}$$

54 因此，$n! = 1\cdot 2\cdot 3\cdots n$。

阶乘函数的一个弱上界是 $n!\leqslant n^n$，因为在阶乘中每一项至多是 n。斯特林近似公式

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right) \tag{3.17}$$

给出了一个更紧确的上界和下界，其中 e 是自然对数的底。可以证明（见练习 3.2-3）

$$n! = o(n^n)\quad n! = \omega(2^n)\quad \lg(n!) = \Theta(n\lg n) \tag{3.18}$$

其中斯特林近似公式有助于证明公式(3.18)。对于任意的 $n\geqslant 1$，下列等式也成立：

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{3.19}$$

其中

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n} \tag{3.20}$$

函数迭代

我们用记号 $f^{(i)}(n)$ 表示函数 $f(n)$ 重复 i 次作用于一个初始值 n 上。形式上，令 $f(n)$ 为实数域上的一个函数。对非负整数 i，递归定义

$$f^{(i)}(n) = \begin{cases}n & \text{如果 } i=0\\ f(f^{(i-1)}(n)) & \text{如果 } i>0\end{cases}$$

例如当 $f(n)=2n$ 时，$f^{(i)}(n)=2^i n$。

多重对数函数

我们用记号“$\lg^* n$”（读作“n 的 log 星”）来表示多重对数。其定义如下。设 $\lg^{(i)} n$ 如上定义，其中 $f(n)=\lg n$。由于非正数的对数无定义，故 $\lg^{(i)} n$ 有定义仅当 $\lg^{(i-1)} n>0$。务必要区分 $\lg^{(i)} n$（从参数 n 开始，连续应用对数函数 i 次）和 $\lg^i n$（n 的对数的 i 次方）。多重对数函数的定义为

55 $$\lg^* n = \min\{i\geqslant 0 : \lg^{(i)} n\leqslant 1\}$$

多重函数是一种增长很慢的函数：

$$\lg^* 2 = 1\quad \lg^* 4 = 2\quad \lg^* 16 = 3\quad \lg^* 65\,536 = 4\quad \lg^*(2^{65\,536}) = 5$$

宇宙中可以观察到的原子数目估计约有 10^{80}，远远小于 $2^{65\,536}$，因此我们很少会遇到一个使 $\lg^* n>5$ 的输入规模 n。

斐波那契数

斐波那契数递归地定义为：

$$F_0 = 0\quad F_1 = 1\quad F_i = F_{i-1}+F_{i-2}\quad \text{当 } i\geqslant 2 \tag{3.21}$$

因此每一个数都是前两个数的和，产生的序列为

$$0,1,1,2,3,5,8,13,21,34,55,\cdots$$

斐波那契数和黄金分割律 ϕ 及其共轭数 $\hat{\phi}$ 有关系，它们由如下公式给出：

$$\phi = \frac{1+\sqrt{5}}{2} = 1.61803\cdots, \quad \hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803\cdots \tag{3.22}$$

特别地，有

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} \tag{3.23}$$

上式可以用归纳法证明(习题 3.2-6)。因为 $|\hat{\phi}|<1$，得公式 $|\hat{\phi}^i|/\sqrt{5}<1/\sqrt{5}<1/2$，因此第 i 个斐波那契数等于和 $\phi^i/\sqrt{5}$ 最接近的整数。所以斐波那契数以指数形式增长。

56

练习

3.2-1 证明：若 $f(n)$ 和 $g(n)$ 是单调递增的函数，则 $f(n)+g(n)$ 和 $f(g(n))$ 也是单调递增的；另外，若 $f(n)$ 和 $g(n)$ 是非负的，那么 $f(n)\cdot g(n)$ 是单调递增的。

3.2-2 证明等式(3.15)。

3.2-3 证明等式(3.18)。并证明 $n! = \omega(2^n)$ 和 $n! = o(n^n)$。

***3.2-4** 函数 $\lceil \lg n \rceil !$ 是否多项式有界？函数 $\lceil \lg \lg n \rceil !$ 呢？

***3.2-5** 哪一个在渐近上更大些：$\lg(\lg^* n)$ 还是 $\lg^*(\lg n)$？

3.2-6 用归纳法证明：第 i 个斐波那契数满足等式

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$$

其中 ϕ 是黄金分割律，$\hat{\phi}$ 是共轭数。

3.2-7 证明：对于 $i\geqslant 0$，第 $(i+2)$ 个斐波那契数满足 $F_{i+2}\geqslant \phi^i$。

思考题

3-1 多项式的渐近性质

设 $p(n)=\sum_{i=0}^{d} a_i n^i$ 为一个 n 的 d 次多项式，其中 $a_d>0$，令 k 为一个常数。利用渐近记号的定义来证明如下性质：

a)若 $k\geqslant d$，则 $p(n)=O(n^k)$。

57

b)若 $k\leqslant d$，则 $p(n)=\Omega(n^k)$。

c)若 $k=d$，则 $p(n)=\Theta(n^k)$。

d)若 $k>d$，则 $p(n)=o(n^k)$。

e)若 $k<d$，则 $p(n)=\omega(n^k)$。

3-2 相对渐近增长

在下表中，对每一对表达式 (A, B)，指出 A 是 B 的 O，o，Ω，ω，或 Θ 中的哪种关系。假设 $k\geqslant 1$，$\epsilon>0$，$c>1$ 都是常数。在表格的空格内填入“是”或“否”即可。

	A	B	O	o	Ω	ω	Θ
a)	$\lg^k n$	n^ϵ					
b)	n^k	c^n					
c)	$\sqrt{n}$	$n^{\sin n}$					
d)	2^n	$2^{n/2}$					
e)	$n^{\lg c}$	$c^{\lg n}$					
f)	$\lg(n!)$	$\lg(n^n)$					

3-3　根据渐近增长率排序

a)根据增长率来对下列函数排序；即找出函数的一种排列 g_1，g_2，…，g_{30}，使 $g_1=\Omega(g_2)$，$g_2=\Omega(g_3)$，…，$g_{29}=\Omega(g_{30})$。将该序列划分成等价类，使 $f(n)$和 $g(n)$在同一个等价类中当且仅当 $f(n)=\Theta(g(n))$。

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$\left(\frac{3}{2}\right)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln\ln n$	$\lg^* n$	$n\cdot 2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n\lg n$	$2^{2^{n+1}}$

b)给出非负函数 $f(n)$的一个例子，使对任何在(a)中的 $g_i(n)$，$f(n)$既不是 $O(g_i(n))$也不是 $\Omega(g_i(n))$。

58

3-4　渐近记号的性质

设 $f(n)$和 $g(n)$为渐近正函数。证明或否定以下的假设：

a)$f(n)=O(g(n))$蕴含 $g(n)=O(f(n))$。

b)$f(n)+g(n)=\Theta(\min(f(n),g(n)))$。

c)$f(n)=O(g(n))$蕴含 $\lg(f(n))=O(\lg(g(n))$，其中 $\lg(g(n))\geqslant 1$ 且 $f(n)\geqslant 1$ 对足够大的 n 成立。

d)$f(n)=O(g(n))$蕴含 $2^{f(n)}=O(2^{g(n)})$。

e)$f(n)=O((f(n))^2)$。

f)$f(n)=O(g(n))$蕴含 $g(n)=\Omega(f(n))$。

g)$f(n)=\Theta(f(n/2))$。

h)$f(n)+o(f(n))=\Theta(f(n))$。

3-5　O和 Ω的一些变形

某些作者定义 Ω 的方式和我们略有不同，可以用 $\tilde{\Omega}$(读做“Ω无穷大”)来表示这种定义。若存在正常数 c 使 $f(n)\geqslant cg(n)\geqslant 0$ 对无穷多的整数 n 成立，则说 $f(n)=\tilde{\Omega}(g(n))$。

a)说明渐近非负的两个函数 $f(n)$和 $g(n)$，要么 $f(n)=O(g(n))$，要么 $f(n)=\tilde{\Omega}(g(n))$，要么二者都成立，然而在用 Ω 代替 $\tilde{\Omega}$ 时并不成立。

b)请描述用 $\tilde{\Omega}$ 代替 Ω 以刻画程序运行时间的潜在优点和缺点。

某些作者定义的 O 也略有不同，设为 O'。称 $f(n)=O'(g(n))$当且仅当 $|f(n)|=O(g(n))$。

c)如果我们用 O' 代替 O 而仍然使用 Ω，定理 3.1 的“当且仅当”的两个方向各有什么变化？

有些作者定义的 $\tilde{O}$ 表示略去了对数因子的 O。

$\tilde{O}(g(n))=\{f(n)$：存在正常数 c、k、和 n_0 使得 $0\leqslant f(n)\leqslant cg(n)\lg^k(n)$，对所有 $n\geqslant n_0$ 成立$\}$。

59

d)请类似地定义 $\tilde{\Omega}$ 和 $\tilde{\Theta}$，并证明与定理 3.1 的类似关系。

3-6　叠函数

在 $\lg^*$ 函数中用到的重复操作符“$*$”可用在实数域内任何单调递增的函数 $f(n)$上。对

一个给定常数 $c \in \mathbf{R}$，定义叠函数 f_c^* 为

$$f_c^*(n) = \min\{i \geqslant 0 : f^{(i)}(n) \leqslant c\}$$

该函数不必针对所有情况定义。换言之，$f_c^*(n)$是为使其自变量小于或等于 c 而重复应用 f 的次数。

对下列每一个函数 $f(n)$和常数 c，给出 $f_c^*(n)$的尽可能紧确的界。

	$f(n)$	c	$f_c^*(n)$
a)	$n-1$	0	
b)	$\lg n$	1	
c)	$n/2$	1	
d)	$n/2$	2	
e)	$\sqrt{n}$	2	
f)	$\sqrt{n}$	1	
g)	$n^{1/3}$	2	
h)	$n/\lg n$	2	

本章注记

根据 Knuth[182]，O 记号的起源可以追溯到 P. Bachmann 在 1892 年发表的一篇数论文章。o 记号是 E. Landau 在 1909 年发明的，用来讨论素数的分布。Ω 和 Θ 记号是由 Knuth[186]所提倡，用来修正在描述上界和下界时都使用 O 记号的常犯的技术错误，尽管技术上 Θ 记号较为准确，但许多人仍然继续使用 O 记号。对渐近记号的历史和发展的进一步讨论可以在 Knuth[182，186]和 Brassard 以及 Bratley[46]内找到。

不是所有的作者都以相同的形式来定义渐近记号，但不同的定义在大多数情况下都比较一 60
致。有些定义包含不是渐近非负的函数，条件是它们的绝对值落在适当的界内。

等式(3.19)由 Robbins[260]而来。其他基本数学函数的性质可以在任何出色的数学参考资料中找到，例如 Abramowitz 和 Stegun[1]或 Zwillinger[320]，或是在微积分的书籍里，例如 Apostol[18]或 Thomas 和 Finney[296]。Knuth[182]和 Graham，Knuth 和 Patashnik[132]包含了许多用在计算机科学上的离散数学的宝贵材料。 61

第4章 递 归 式

在2.3.2节中我们已经知道，当一个算法包含对自身的递归调用时，其运行时间通常可以用递归式(recurrence)来表示。递归式是一组等式或不等式，它所描述的函数是用在更小的输入下该函数的值来定义的。例如，在2.3.2节中，MERGE-SORT过程的最坏情况运行时间$T(n)$可由下面的递归式

$$T(n)=\begin{cases}\Theta(1) & \text{如果 } n=1\\ 2T(n/2)+\Theta(n) & \text{如果 } n>1\end{cases} \tag{4.1}$$

表示，其解为$T(n)=\Theta(n\lg n)$。

本章介绍了三种解递归式的方法，即找出解的渐近"Θ"或"O"界的方法。在*代换法*(substitution method)中，我们先猜有某个界存在，然后再用数学归纳法证明该猜测的正确性。*递归树方法*(recursion-tree method)将递归式转换成树形结构，树中的结点代表在不同递归层次付出的代价。最后，再利用对和式限界的技术来解出递归式。*主方法*(master method)给出递归形式

$$T(n)=aT(n/b)+f(n)$$

的界，其中$a\geqslant 1$，$b>1$，$f(n)$是给定的函数；这种方法要记忆三种情况，但一旦做到了这点，确定很多简单递归式的界就很容易了。

技术细节

在实践中，在表达和解递归式时常常略去一些技术性细节。例如，常常假设函数的自变量为整数。通常，一个算法的运行时间$T(n)$在定义时都假设n为整数，因为对大多数算法来说输入

的规模都是整数。例如，表达MERGE-SORT运行时间的递归式实际上应该是：

$$T(n)=\begin{cases}\Theta(1) & \text{如果 } n=1\\ T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+\Theta(n) & \text{如果 } n>1\end{cases} \tag{4.2}$$

我们常常忽略的另一类细节是边界条件，因为对于固定规模的输入来说，算法的运行时间为常量，故对足够小的n来说，表示算法运行时间的递归式一般为$T(n)=\Theta(1)$。据此，为了方便起见，就常忽略递归式的边界条件，并且假设对小的n值$T(n)$是常量。例如，一般将递归式(4.1)表达成

$$T(n)=2T(n/2)+\Theta(n) \tag{4.3}$$

而并不明确给出当n很小时$T(n)$的值。其原因在于，虽然递归式的解会随着$T(1)$值的改变而改变，但此改变不会超过常数因子，因而增长的阶没有变化。

在表示并解递归式时，常忽略上取整、下取整以及边界条件。进行分析时先假设没有这些细节，而后再确定它们重要与否。它们通常并不重要，但是重要的是要知道它们在什么情况下是事关紧要的。经验以及已知的一些定理告诉我们：这些细节不会影响算法分析中遇到的许多递归式渐近界(见定理4.1)。然而，在这一章中，我们将讨论其中的一些细节，以展示递归式解法的微妙之处。

4.1 代换法

用代换法解递归式需要两个步骤：

1)猜测解的形式。

2)用数学归纳法找出使解真正有效的常数。

"代换法"这一名称源于当归纳假设用较小值时，用所猜测的值代替函数的解。这种方法很有

效，但是只能用于解的形式很容易猜的情形。

代换法可用来确定一个递归式的上界或下界。作为例子，让我们确定递归式

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{4.4}$$

的一个上界，这个式子与递归式(4.2)和式(4.3)类似。我们猜其解为 $T(n)=O(n\lg n)$。我们的方法是证明 $T(n)\leqslant cn\lg n$，其中 $c>0$ 是某个常数。先假设这个界对 $\lfloor n/2 \rfloor$ 成立，即 $T(\lfloor n/2 \rfloor)\leqslant c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$。对递归式作替换，得：

63

$$\begin{aligned} T(n) &\leqslant 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \leqslant cn\lg(n/2) + n \\ &= cn\lg n - cn\lg 2 + n = cn\lg n - cn + n \leqslant cn\lg n \end{aligned}$$

最后一步只要 $c\geqslant 1$ 就成立。

接下来应用数学归纳法就要求解对边界条件成立。一般来说，可以通过证明边界条件符合归纳证明的基本情况来说明它的正确性。对于递归式(4.4)，必须证明能够选择足够大的常数 c，使界 $T(n)\leqslant cn\lg n$ 也对边界条件成立。这些要求有时会导致问题。假设 $T(1)=1$ 是递归式唯一的边界条件。那么对于 $n=1$ 时，界 $T(n)\leqslant cn\lg n$ 也就是 $T(1)\leqslant c1\lg 1=0$，与 $T(1)=1$ 不符。因此，归纳证明的基本情况不能满足。

对特殊边界条件证明归纳假设中的这种困难很容易解决。例如，对递归式(4.4)，利用渐近记号，只要求对 $n\geqslant n_0$，证明 $T(n)\leqslant cn\lg n$，其中 n_0 是常数。这样做的思想是在归纳证明中，对困难的边界条件 $T(1)=1$ 不加考虑。我们可以把 $T(2)$ 和 $T(3)$ 作为归纳证明中的边界条件代替 $T(1)$，让 $n_0=2$，这是因为对 $n>3$，递归不直接依赖于 $T(1)$。我们将递归式的基本情况($n=1$)和归纳证明的基本情况($n=2$ 和 $n=3$)区别开了。通过递归式，可以求解出 $T(2)=4$ 和 $T(3)=5$。归纳证明 $T(n)\leqslant cn\lg n$ 正确性时，其中常量 $c\geqslant 1$，只要 c 取足够大的常数使 $T(2)\leqslant c2\lg 2$ 和 $T(3)\leqslant c3\lg 3$ 即可完成证明。实际上，选取任何 $c\geqslant 2$，$n=2$ 和 $n=3$ 都可满足这个要求。对后面将要讨论的大部分递归式，可以直接扩展边界条件，使递归假设对很小的 n 也成立。

做一个好的猜测

不幸的是，并不存在通用的方法来猜测递归式的正确解。这种猜测需要经验，有时甚至是创造性的。值得庆幸的是，还有一些试探法可以帮助做出好的猜测。将在 4.2 节介绍的递归树也可以用来帮助猜测。

64

如果某个递归式与先前见过的类似，则可猜测该递归式有类似的解。例如，递归式

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$$

看起来比较难解，因为右式 T 的自变量中加了 17。但是，我们的直觉是这个多出来的项对解的影响可能不大。当 n 很大时，$T(\lfloor n/2 \rfloor)$ 与 $T(\lfloor n/2 \rfloor + 17)$ 之间的差别并不大：两者都将 n 分为均匀的两半。因而，我们猜 $T(n)=O(n\lg n)$。这个结论可用代换法来验证(见练习 4.1-5)。

猜测答案的另一种方法是先证明递归式的较松的上下界，然后再缩小不确定性区间。例如，对递归式(4.4)，可以先假设其下界为 $T(n)=\Omega(n)$，因为递归式中有 n，而我们可以证明初始上界为 $T(n)=O(n^2)$。然后，逐步降低其上界，提高其下界，直到达到正确的渐近确界 $T(n)=\Theta(n\lg n)$。

一些细微问题

有时，我们或许能够猜出递归式解的渐近界，但却会在归纳证明时出现一些问题。通常，问题出在归纳假设不够强，无法证明其准确的界。遇到这种情况时，可以去掉一个低阶项来修改所猜的界，以使证明顺利进行。

考虑下面的递归式：

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

我们猜测其解为 $O(n)$，即要证明对适当选择的 c，有 $T(n)\leqslant cn$。用所猜测的界对递归式作替换，得

$$T(n)\leqslant c\lfloor n/2\rfloor+c\lceil n/2\rceil+1=cn+1$$

由此引不出 $T(n)\leqslant cn$，无论 c 为何值。读者可能会猜一个更大的界，如 $T(n)=O(n^2)$，虽然这确实是上界，但事实上，我们所猜测的解 $T(n)=O(n)$ 是正确的。为了证明这一点，要做一个更强的归纳假设。

从直觉上说，我们的猜测几乎是正确的：只是差了一个常数 1，即一个低阶项。然而，就因为差了一项，数学归纳法就无法证明出期望的结果。从所作的猜测中减去一个低阶项，即 $T(n)$
65 $\leqslant cn-b$，$b\geqslant 0$ 是个常数。现在有

$$T(n)\leqslant(c\lfloor n/2\rfloor-b)+(c\lceil n/2\rceil-b)+1=cn-2b+1\leqslant cn-b$$

只要 $b\geqslant 1$。像先前一样，c 要选择的足够大，以便能够处理边界条件。

不少人都会觉得从所作的猜测中减去一项有点与直觉不符。为什么不是增加一项来解决问题呢？关键在于要理解我们是在用数学归纳法：通过对更小的值做更强的假设，就可以证明对某个给定值的更强的结论。

避免陷阱

在运用渐近表示时很容易出错。例如，在递归式(4.4)中，由假设 $T(n)\leqslant cn$ 并证明

$$T(n)\leqslant 2(c\lfloor n/2\rfloor)+n\leqslant cn+n=O(n)\qquad\Leftarrow \text{错!!!}$$

因为 c 是常数，因而错误地证明了 $T(n)=O(n)$。错误在于没有证明归纳假设的准确形式，即 $T(n)\leqslant cn$。

改变变量

有时，对一个陌生的递归式作一些简单的代数变换，就会使之变成读者较熟悉的形式。作为一个例子，考虑

$$T(n)=2T(\lfloor\sqrt{n}\rfloor)+\lg n$$

这个式子看上去较难，但可以对它进行简化，方法是改动变量。为了方便起见，不考虑数的截取整数问题，如将 $\sqrt{n}$ 化为整数。设 $m=\lg n$，得

$$T(2^m)=2T(2^{m/2})+m$$

再设 $S(m)=T(2^m)$，得到新的递归式

$$S(m)=2S(m/2)+m$$

这个式子看起来与式(4.4)就非常像了，这个新的递归式的界是：$S(m)=O(m\lg m)$。将 $S(m)$ 代
66 回 $T(n)$，有 $T(n)=T(2^m)=S(m)=O(m\lg m)=O(\lg n\lg\lg n)$。

练习

4.1-1 证明 $T(n)=T(\lceil n/2\rceil)+1$ 的解为 $O(\lg n)$。

4.1-2 证明 $T(n)=2T(\lfloor n/2\rfloor)+n$ 的解为 $O(n\lg n)$。证明这个递归的解也是 $\Omega(n\lg n)$。得到解为 $\Theta(n\lg n)$。

4.1-3 证明：通过作不同的递归假设，对递归式(4.4)我们可以克服在证明边界条件 $T(1)=1$ 时的困难，同时无需调整归纳证明中的边界情况。

4.1-4 证明合并排序算法的“准确”递归式(4.2)的解为 $\Theta(n\lg n)$。

4.1-5 证明 $T(n)=2T(\lfloor n/2\rfloor+17)+n$ 的解为 $O(n\lg n)$。

4.1-6 通过改变变量求解递归式 $T(n)=2T(\sqrt{n})+1$。得到的解应当是渐近紧确的。不必担心值是否为整数。

4.2　递归树方法

虽然代换法给递归式的解的正确性提供了一种简单的证明方法，但是有的时候很难得到一

个好的猜测。像我们在 2.3.2 节分析合并排序递归式那样，画出一个递归树是一种得到好猜测的直接方法。在递归树中，每一个结点都代表递归函数调用集合中一个子问题的代价。我们将树中每一层内的代价相加得到一个每层代价的集合，再将每层的代价相加得到递归是所有层次的总代价。当用递归式表示分治算法的运行时间时，递归树的方法尤其有用。

递归树最适合用来产生好的猜测，然后用代换法加以验证。但使用递归树产生好的猜测时，通常可以容忍小量的"不良量"(sloppiness)，因为稍后就会证明所做的猜测。如果画递归树时非常地仔细，并且将代价都加了起来，那么就可以直接用递归树作为递归式解的证明。在本节中，我 67
们将使用递归树产生好的猜测；在 4.4 节中，我们将使用递归树直接证明形成主方法基础的定理。

例如，来看看递归树如何为递归式 $T(n)=3T(\lfloor n/2\rfloor)+\Theta(n^2)$ 提供良好的猜测。一开始，我们先解决找到解上界的问题。因为知道底和顶函数在解决递归式问题时并不重要(这个例子的不良量是可以容忍的)，所以建立了一颗关于递归式 $T(n)=3T(n/4)+cn^2$ 的递归树，常系数 $c>0$。

图 4-1 显示了 $T(n)=3T(n/4)+cn^2$ 的递归树的演化过程。为了方便，不妨假设 n 是 4 的幂(另一个可容忍量的例子)。图 4-1a 显示了 $T(n)$，在图 4-1b 中被扩展成一个等价的用来表示递归的树。根部的 cn^2 项表示递归在顶层时所花的代价，而根部以下的三棵子树表示这些 $n/4$ 大小的子问题所需的代价。图 4-1c 展示了对图 4-1b 作进一步处理的过程，将图 4 1b 中代价为 $T(n/4)$ 的结点进行了扩展。三棵子树的根的代价分别是 $c(n/4)^2$。我们继续将树中的每个结点进行扩展，

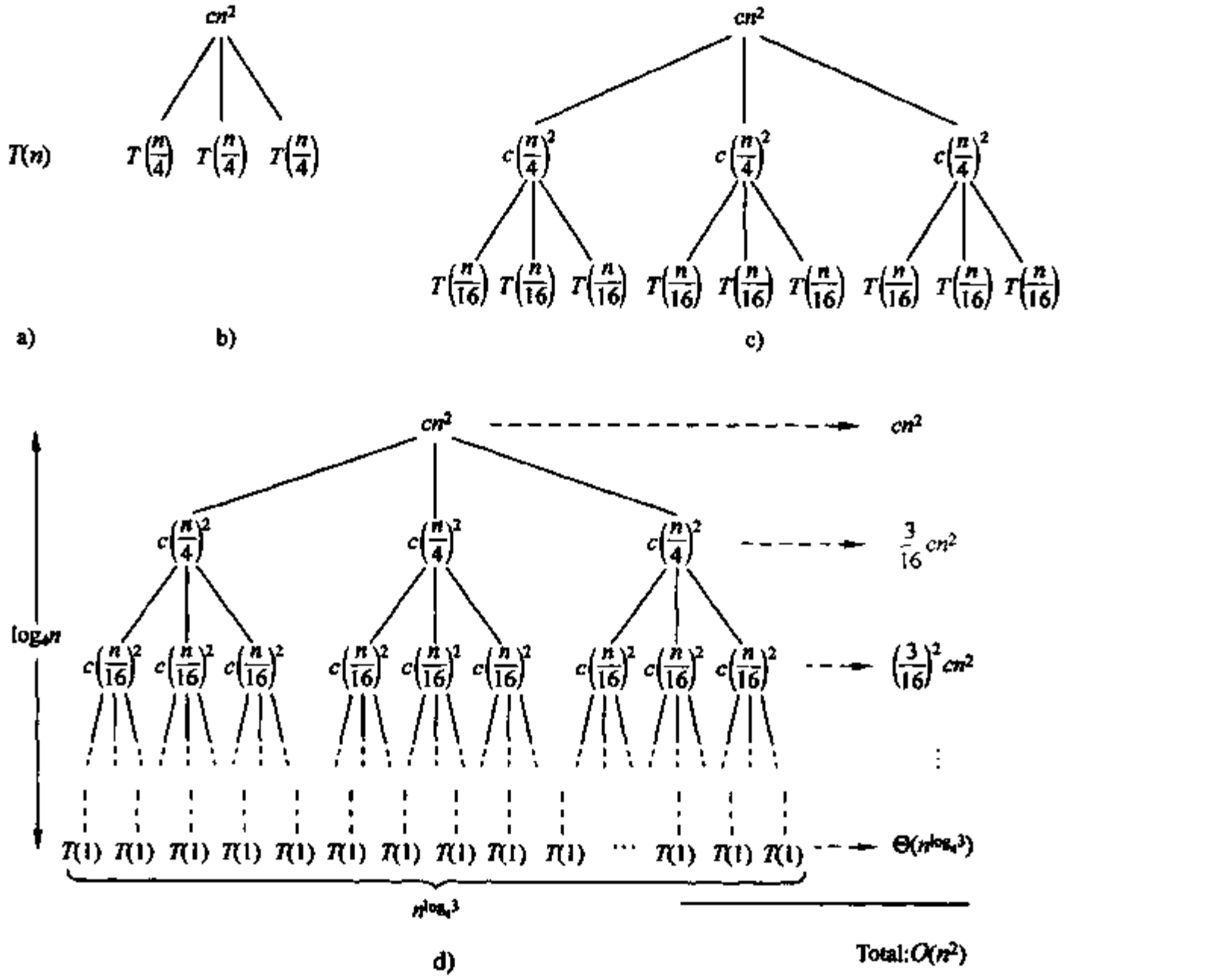


图 4-1　与递归式 $T(n)=3T(n/4)+cn^2$ 对应的递归树的构造。a)示出了 $T(n)$，它在 b)～d)中被不断地扩展而形成了递归树。d)完全扩展了的递归树深度为 $\log_4 n$(它有 $\log_4 n+1$ 层)

直到递归结束。

子问题的大小将会随着离树根越来越远而变得越来越小，最终一定会达到一个边界条件。到达边界条件时离树根有多远呢？对于深度为 i 的结点，其子问题大小为 $n/4^i$。那么，当 $n/4^i=1$，或是当 $i=\log_4 n$ 时子问题的大小即达到 1。因此，这棵树有 $\log_4 n+1$ 层（0，1，2，…，$\log_4 n$）。

下面计算树中每一层所需的代价。每一层都有它的上一层的 3 倍的结点，所以在深度 i 时结点的数目是 3^i。当从根部往下时，每一层的子问题大小是以 4 的因子在缩小，在深度 i 时，每个结点的代价为 $c(n/4^i)^2$，其中 $i=0,1,2,\cdots,\log_4 n-1$，则第 i 层的所有结点的总代价是 $3^i c(n/4^i)^2=(3/16)^i cn^2$。在最后一层，也就是深度 $\log_4 n$ 时，有 $3^{\log_4 n}=n^{\log_4 3}$ 个结点，每个结点的代价是 $T(1)$，总代价为 $n^{\log_4 3}T(1)$，也就是 $\Theta(n^{\log_4 3})$。

现在将所有层次的代价相加得到整棵树的代价：

$$T(n)=cn^2+\frac{3}{16}cn^2+\left(\frac{3}{16}\right)^2cn^2+\cdots+\left(\frac{3}{16}\right)^{\log_4 n-1}cn^2+\Theta(n^{\log_4 3})$$

$$=\sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2+\Theta(n^{\log_4 3})=\frac{(3/16)^{\log_4 n}-1}{(3/16)-1}cn^2+\Theta(n^{\log_4 3})$$

68 ~ 69

最后一个式子看上去有些乱，不过可以将量适当放松，用无限递减等比级数作为上界。回到上一步，应用方程式(A.6)，有

$$T(n)=\sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2+\Theta(n^{\log_4 3})<\sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2+\Theta(n^{\log_4 3})$$

$$=\frac{1}{1-(3/16)}cn^2+\Theta(n^{\log_4 3})=\frac{16}{13}cn^2+\Theta(n^{\log_4 3})=O(n^2)$$

于是，我们得到了原来的递归式 $T(n)=3T(\lfloor n/4\rfloor)+\Theta(n^2)$ 的一个猜测。在这个例子里，cn^2 系数形成一个递减的等比级数，由方程式(A.6)，可知这些系数的总和的上界是常数 16/13。由于树根所需的代价为 cn^2，所以根部的代价占总代价的一个常数部分。换句话说，整棵树的总代价是由根部的代价所决定的。

事实上，如果 $O(n^2)$ 确实是此递归式的上界（稍后将会证明），那么它一定是确界（tight bound）。为什么呢？第一个递归调用需要的代价是 $\Theta(n^2)$，所以 $\Omega(n^2)$ 一定是此递归式的下界。

现在可以使用代换法来验证猜测的正确性，$T(n)=O(n^2)$ 是递归式 $T(n)=3T(\lfloor n/4\rfloor)+\Theta(n^2)$ 的一个上界。我们需要证明，当某常数 $d>0$，$T(n)\leqslant dn^2$ 成立。适用与前面相同的常数 $c>0$，有

$$T(n)\leqslant 3T(\lfloor n/4\rfloor)+cn^2\leqslant 3d\lfloor n/4\rfloor^2+cn^2\leqslant 3d(n/4)^2+cn^2$$

$$=\frac{3}{16}dn^2+cn^2\leqslant dn^2$$

只要 $d\geqslant(16/13)c$，最后一步都会成立。

现在来看另一个较为复杂的例子，如图 4-2 显示的递归树：

$$T(n)=T(n/3)+T(2n/3)+O(n)$$

（为了简化起见，此处还是省略了下取整函数和上取整函数）。与前面一样，我们使用 c 来代表 $O(n)$ 项的常数因子。当将递归树内各层的数值加起来时，可以得到每一层的 cn 值。从根部到叶子的最长路径是 $n\to(2/3)n\to(2/3)^2n\to\cdots\to1$。因为当 $k=$

70

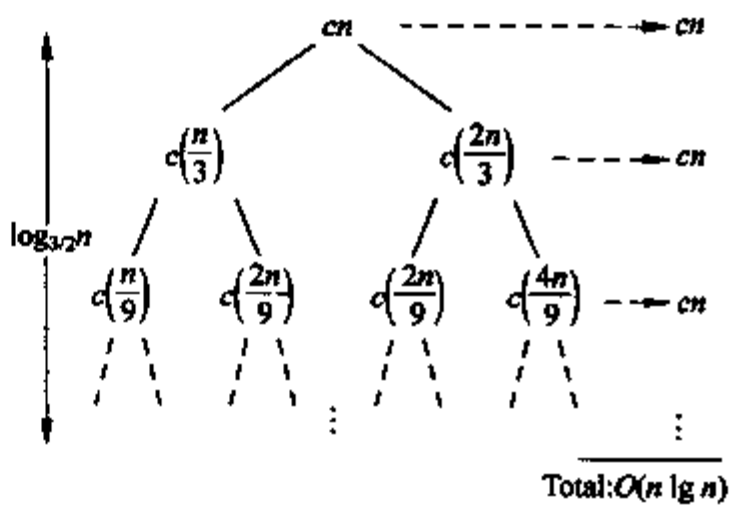


图 4-2　与递归式 $T(n)=T(n/3)+T(2n/3)+cn$ 对应的递归树

$\log_{3/2}n$ 时，$(2/3)^k n=1$，所以树的深度是 $\log_{3/2}n$。

直觉上，我们预期递归式的解至多是层数乘以每层的代价，也就是 $O(cn\log_{3/2}n)=O(n\lg n)$。总代价被均匀地分布到递归树内的每一层上。这里还有一个复杂点：我们还没有考虑叶子的代价。如果这棵树是高度为 $\log_{3/2}n$ 的完整二叉树，那么有 $2^{\log_{3/2}n}=n^{\log_{3/2}2}$ 个叶子。由于叶子代价是常数，因此所有叶子代价的总和为 $\Theta(n^{\log_{3/2}2})$，或者说 $\omega(n\lg n)$。然而，这棵递归树并不是完整的二叉树，少于 $n^{\log_{3/2}2}$ 个叶子，而且从树根往下的过程中，越来越多的内部结点在消失。因此，并不是所有层次都刚好需要 cn 代价；越靠近底层，需要的代价越少。我们可以计算出准确的总代价，但记住我们只是想要找出一个猜测来使用到代换法中。让我们容忍这些误差，而来证明上界为 $O(n\lg n)$ 的猜测是正确的。

事实上，可以用代换法来证明 $O(n\lg n)$ 是递归式解的上界。下面证明 $T(n)\leqslant dn\lg n$，当 d 是一个合适的正值常数。有 71

$$\begin{aligned}
T(n) &\leqslant T(n/3)+T(2n/3)+cn\\
&\leqslant d(n/3)\lg(n/3)+d(2n/3)\lg(2n/3)+cn\\
&=(d(n/3)\lg n-d(n/3)\lg 3)+(d(2n/3)\lg n-d(2n/3)\lg(3/2))+cn\\
&=dn\lg n-d((n/3)\lg 3+(2n/3)\lg(3/2))+cn\\
&=dn\lg n-d((n/3)\lg 3+(2n/3)\lg 3-(2n/3)\lg 2)+cn\\
&=dn\lg n-dn(\lg 3-2/3)+cn\leqslant dn\lg n
\end{aligned}$$

上式成立的条件是 $d\geqslant c/(\lg 3-(2/3))$。因此，没有必要去更准确地计算递归树中的代价。

练习

4.2-1 利用递归树来猜测递归式 $T(n)=3T(\lfloor n/2\rfloor)+n$ 的一个好的渐近上界，并利用代换法来证明你的猜测。

4.2-2 利用递归树来证明递归式 $T(n)=T(n/3)+T(2n/3)+cn$ 的解是 $\Omega(n\lg n)$，其中 c 是一个常数。

4.2-3 画出 $T(n)=4T(\lfloor n/2\rfloor)+cn$ 的递归树，并给出其解的渐近确界，其中 c 是一个常数。然后，用代换法证明你给出的界。

4.2-4 利用递归树来找出递归式 $T(n)=T(n-a)+T(a)+cn$ 的渐近紧确解，其中 $a\geqslant 1$ 且 $c>0$ 是常数。

4.2-5 利用递归树来找出递归式 $T(n)=T(an)+T((1-a)n)+cn$ 的渐近紧确解，其中 a 是 $0<a<1$ 的常数，且 c 是大于 0 的常数。 72

4.3 主方法

主方法(master method)给出求解如下形式的递归式的“食谱”方法：

$$T(n)=aT(n/b)+f(n) \tag{4.5}$$

其中 $a\geqslant 1$ 和 $b>1$ 是常数，$f(n)$ 是一个渐近正的函数。主方法要求记忆三种情况，但这样可很容易确定许多递归式的解，且不需用笔和纸。

递归式(4.5)描述了将规模为 n 的问题划分为 a 个子问题的算法的运行时间，每个子问题规模为 n/b，a 和 b 是正常数。a 个子问题被分别递归地解决，时间各为 $T(n/b)$。划分原问题和合并答案的代价由函数 $f(n)$ 描述。(即使用 2.3.2 节中的记号，$f(n)=D(n)+C(n)$。)如，MERGE-SORT 过程的递归式中有 $a=2$，$b=2$，$f(n)=\Theta(n)$。

从技术正确性角度看，递归式实际上没有得到很好的定义，因为 n/b 可能不是个整数。但用

$T(\lfloor n/b \rfloor)$或 $T(\lceil n/b \rceil)$来代替 a 项 $T(n/b)$并不影响递归式的渐近行为(我们将在下节对此证明)。因而，我们在写分治算法时略去下取整和上取整函数会带来很大的方便。

主定理

主方法依赖于下面的定理：

定理 4.1(主定理)　设 $a \geqslant 1$ 和 $b > 1$ 为常数，设 $f(n)$为一函数，$T(n)$由递归式

$$T(n) = aT(n/b) + f(n)$$

对非负整数定义，其中 n/b 指 $\lfloor n/b \rfloor$ 或 $\lceil n/b \rceil$。那么 $T(n)$可能有如下的渐近界：

1)若对于某常数 $\varepsilon > 0$，有 $f(n) = O(n^{\log_b a - \varepsilon})$，则 $T(n) = \Theta(n^{\log_b a})$；

2)若 $f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a} \lg n)$；

3)若对某常数 $\varepsilon > 0$，有 $f(n) = \Omega(n^{\log_b a + \varepsilon})$，且对常数 $c < 1$ 与所有足够大的 n，有 $af(n/b) \leqslant$
73 $cf(n)$，则 $T(n) = \Theta(f(n))$。 ■

在运用该定理之前，先来看看它包含哪些内容。在以上三种情况的每一种中，都把函数 $f(n)$与函数 $n^{\log_b a}$进行比较。我们的直觉是解由两个函数中较大的一个决定。例如在第一种情况中，函数 $n^{\log_b a}$更大，则解为 $T(n) = \Theta(n^{\log_b a})$。在第三种情况下，$f(n)$是较大的函数，则解为 $T(n) = \Theta(f(n))$。在第二种情况中，两种函数同样大，乘以对数因子，则解为 $T(n) = \Theta(n^{\log_b a} \lg n)$。

这只是我们的直觉，另外还有一些技术问题要加以理解。在第一种情况中，不仅要有 $f(n)$小于 $n^{\log_b a}$，还必须是多项式地小于，即对某个常量 $\varepsilon > 0$，$f(n)$必须渐近地小于 $n^{\log_b a}$，两者差一个因子 n^{ε}。在第三种情况中，$f(n)$不仅要大于 $n^{\log_b a}$，且要多项式地大于，还要满足"规则性"条件 $af(n/b) \leqslant cf(n)$。后面将碰到的大部分多项式有界的函数都满足这个条件。

要注意三种情况并没有覆盖所有可能的 $f(n)$。当 $f(n)$只是小于 $n^{\log_b a}$但不是多项式地小于时，在第一种情况和第二种情况之间就存在一条"沟"。类似情况下，当 $f(n)$大于 $n^{\log_b a}$，但不是多项式地大，第二种情况和第三种情况之间就会存在一条"沟"。如果 $f(n)$落在任一条"沟"中，或是第三种情况中规则性条件不成立，则主方法就不能用于解递归式。

主方法的应用

在应用此方法时，先决定要选取定理中的哪一种情况(如果有情况可满足的话)，然后即可简单地写下答案。

先看第一个例子：

$$T(n) = 9T(n/3) + n$$

在这个递归式中，$a = 9$，$b = 3$，$f(n) = n$，则 $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$。因为 $f(n) = O(n^{\log_3 9 - \varepsilon})$，其中 $\varepsilon = 1$，这对应于主定理中的第一种情况，答案为 $T(n) = \Theta(n^2)$。

再看一个例子：$T(n) = T(2n/3) + 1$

其中 $a = 1$，$b = 3/2$，$f(n) = 1$，$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$。第二种情况成立，因为 $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$，故递归式的解为 $T(n) = \Theta(\lg n)$。

对递归式 $T(n) = 3T(n/4) + n \lg n$，有 $a = 3$，$b = 4$，$f(n) = n \lg n$，$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$。
74 因为 $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$，其中 $\varepsilon \approx 0.2$，如果能证明对 $f(n)$第三种情况中的规则性条件成立，则选用定理中的第三种情况。对足够大的 n，$af(n/b) = 3(n/4)\lg(n/4) \leqslant (3/4)n \lg n = cf(n)$，$c = 3/4$，则递归式的解为 $T(n) = \Theta(n \lg n)$。

对下面的递归式主方法不适用：

$$T(n) = 2T(n/2) + n \lg n$$

这个递归式在形式上是合适的：$a = 2$，$b = 2$，$f(n) = n \lg n$，$n^{\log_b a} = n$。看上去可选择第三种情

况，因为 $f(n)=n\lg n$ 渐近大于 $n^{\log_b a}=n$，但并不是多项式大于。对任意正常数 ϵ，比值 $f(n)/n^{\log_b a}=(n\lg n)/n=\lg n$ 渐近小于 n^{ϵ}。因此，该递归式落在情况二与情况三之间。(练习 4.4-2 给出了解答)

练习

4.3-1 用主方法来给出下列递归式紧确的渐近界：

a) $T(n)=4T(n/2)+n$

b) $T(n)=4T(n/2)+n^2$

c) $T(n)=4T(n/2)+n^3$

4.3-2 某个算法 A 的运行时间由递归式 $T(n)=7T(n/2)+n^2$ 表示；另一个算法 A' 的运行时间为 $T'(n)=aT'(n/4)+n^2$。若要 A' 比 A 更快，那么 a 的最大整数值是多少？

4.3-3 用主方法证明二分查找递归 $T(n)=T(n/2)+\Theta(1)$ 的解是 $T(n)=\Theta(\lg n)$。(二分查找的描述见练习 2.3-5)

4.3-4 主方法能否应用于递归式 $T(n)=4T(n/2)+n^2\lg n$？为什么？给出此递归式的渐近上界。

***4.3-5** 考虑在某个常数 $c<1$ 时的规则性条件 $af(n/b)\leqslant cf(n)$，此条件是主定理第三种情况的一部分。举一个常数 $a\geqslant 1$，$b>1$ 以及一个函数 $f(n)$，满足主定理第三种情况中的除了规则性条件之外的所有条件的例子。

75

*4.4 主定理的证明

本节包含主定理(定理 4.1)的证明。在应用这个定理时，不一定需要理解其证明。

证明分为两部分。第一部分分析"主"递归式(4.5)，并作了简化假设 $T(n)$ 仅定义在 $b>1$ 的整数幂上，即 $n=1, b, b^2, \cdots$。这部分从直觉上说明了该定理为什么是正确的。第二部分说明如何将分析扩展至对所有的正整数 n 都成立，主要是应用数学技巧来解决下取整函数和上取整函数的处理问题。

本节中，我们将用渐近记号来描述定义在 b 的整数幂上的函数的性态。这是有点"活用"渐近记号了。回忆一下，渐近记号的定义要求证明界对足够大的数成立，而不仅仅是对 b 的幂成立。因为可以构造出适合于集合 $\{b^i: i=0, 1, \cdots\}$(而不是非负整数)上的渐近记号，故这种"活用"是没有关系的。

要注意当在一个有限域上运用渐近记号时，不能引起不合适的结论。例如，证明当 n 是 2 的整数幂时，$T(n)=O(n)$ 成立，但不能保证 $T(n)=O(n)$ 对所有 n 成立。函数 $T(n)$ 可定义成

$$T(n)=\begin{cases} n & \text{如果 } n=1,2,4,8,\cdots, \\ n^2 & \text{否则} \end{cases}$$

由该定义得出的最佳上界可证明是 $T(n)=O(n^2)$，与 $T(n)=O(n)$ 相差很大。因而，在有限域上用渐近记号时一定要有上下文说明。

4.4.1 取正合幂时的证明

主定理证明的第一部分是分析递归式(4.5)

$$T(n)=aT(n/b)+f(n)$$

此时的假设是 n 为 $b>1$ 的正合幂，且 b 不必是整数。分析可分成三个引理说明。第一个引理是将解原递归式的问题归约为对一个含和式的求值的问题。第二个引理决定含和式的界。第三个引理把前两个合在一起，证明当 n 为 b 的正合幂时主定理成立。

76

引理 4.2　设 $a\geqslant 1$，$b>1$ 为常数，$f(n)$ 为定义在 b 的正合幂上的非负函数。定义 $T(n)$ 如下：

$$T(n)=\begin{cases}\Theta(1) & \text{如果 } n=1\\ aT(n/b)+f(n) & \text{如果 } n=b^i\end{cases}$$

其中 i 是正整数。则有

$$T(n)=\Theta(n^{\log_b a})+\sum_{j=0}^{\log_b n-1}a^j f(n/b^j) \tag{4.6}$$

证明：我们使用图 4-3 中的递归树。根结点的代价为 $f(n)$，它有 a 个子女，每个的代价是 $f(n/b)$。(为方便起见可将 a 视为整数，但这对数学推导没什么影响。)每个子女又各有 a 个子女，代价为 $f(n/b^2)$。这样就有 a^2 个结点离根的距离为 2。一般地，距根为 j 的结点有 a^j 个，每一个的代价为 $f(n/b^j)$。每一个叶结点的代价为 $T(1)=\Theta(1)$，每一个都距根 $\log_b n$，因为 $n/b^{\log_b n}=1$。树中共有 $a^{\log_b n}=n^{\log_b a}$ 个叶结点。

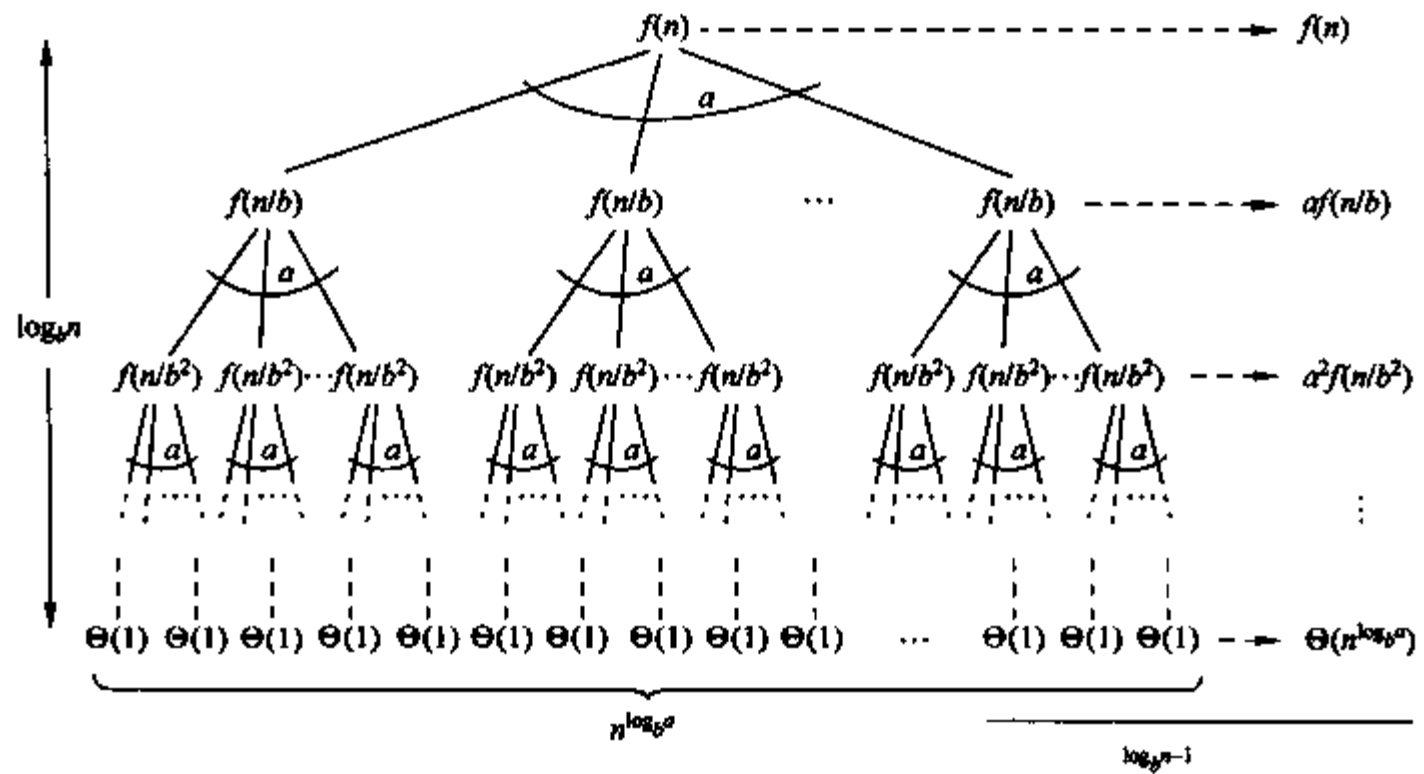


图 4-3　由递归式 $T(n)=aT(n/b)+f(n)$ 生成的递归树。该树是一棵完全 a 叉树，共有 $n^{\log_b a}$ 个叶结点，高度为 $\log_b n$。树中每一层的代价都在右边示出，其和由(4.6)式给出

77

可以将树中各层上的代价加起来而得到方程(4.6)。第 j 层上内部结点的代价为 $a^j f(n/b^j)$，故各层内部结点的总代价和为

$$\sum_{j=0}^{\log_b n-1}a^j f(n/b^j)$$

在其所基于的分治算法中，这个和值表示了将问题分解成为子问题并将子问题的解合并时所花的代价，所有叶子的代价(即解 $n^{\log_b a}$ 个规模为 1 的子问题的代价)为 $\Theta(n^{\log_b a})$。■

根据递归树，主定理的三种情况对应于树中总代价的三种情况：1)由所有叶结点的代价决定；2)均匀地分布在各层上；3)由根结点的代价决定。

方程(4.6)中的和式表示了分治算法中分解和合并步骤的代价。下一个引理给出了该和式的渐近界。

引理 4.3　设 $a\geqslant 1$，$b>1$ 为常数，$f(n)$ 为定义在 b 的整数幂上的非负函数。函数 $g(n)$ 由下式定义

$$g(n)=\sum_{j=0}^{\log_b n-1}a^j f(n/b^j) \tag{4.7}$$

对 b 的整数幂，该函数可被渐近限界为：

1)若对常数 $\epsilon>0$，有 $f(n)=O(n^{\log_b a-\epsilon})$，则 $g(n)=O(n^{\log_b a})$。

2)若 $f(n)=\Theta(n^{\log_b a})$，则 $g(n)=\Theta(n^{\log_b a}\lg n)$。

3)若对常数 $c<1$ 及所有的 $n\geqslant b$，$af(n/b)\leqslant cf(n)$，则 $g(n)=\Theta(f(n))$。

证明：对情况 1，有 $f(n)=O(n^{\log_b a-\epsilon})$，这隐含着 $f(n/b^j)=O((n/b^j)^{\log_b a-\epsilon})$。用它对方程(4.7)作代换，得

$$g(n)=O\left(\sum_{j=0}^{\log_b n-1}a^j\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) \tag{4.8}$$

78

对 O 标记内的式子限界，方法是提出不变项并作简化，得一上升几何级数：

$$\sum_{j=0}^{\log_b n-1}a^j\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}=n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}\left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j=n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}(b^\epsilon)^j$$

$$=n^{\log_b a-\epsilon}\left(\frac{b^{\epsilon\log_b n}-1}{b^\epsilon-1}\right)=n^{\log_b a-\epsilon}\left(\frac{n^\epsilon-1}{b^\epsilon-1}\right)$$

因为 b 与 ϵ 都是常数，最后的表达式可化简为 $n^{\log_b a-\epsilon}O(n^\epsilon)=O(n^{\log_b a})$。用此表达式对方程(4.8)作替换，得

$$g(n)=O(n^{\log_b a})$$

情况 1)得证。

为证情况 2)，假设 $f(n)=\Theta(n^{\log_b a})$，有 $f(n/b^j)=\Theta((n/b^j)^{\log_b a})$。用此式对方程(4.7)作替换，得

$$g(n)=\Theta\left(\sum_{j=0}^{\log_b n-1}a^j\left(\frac{n}{b^j}\right)^{\log_b a}\right) \tag{4.9}$$

对 Θ 记号中的式子作类似情况 1)中的限界，但所得并非一几何级数，而是每项都是相同的：

$$\sum_{j=0}^{\log_b n-1}a^j\left(\frac{n}{b^j}\right)^{\log_b a}=n^{\log_b a}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a}}\right)^j=n^{\log_b a}\sum_{j=0}^{\log_b n-1}1=n^{\log_b a}\log_b n$$

用此式对方程(4.9)中的和式作替换，有

$$g(n)=\Theta(n^{\log_b a}\log_b n)=\Theta(n^{\log_b a}\lg n)$$

则情况 2)得证。

79

情况 3)也可作类似的证明。因为 $f(n)$ 在 $g(n)$ 的定义式(4.7)中出现，且 $g(n)$ 的所有项都是非负的，可以得出 $g(n)=\Omega(f(n))$ 对 b 的整数幂成立。假设对常数 $c<1$ 和所有 $n\geqslant b$，$af(n/b)\leqslant cf(n)$，有 $f(n/b)\leqslant(c/a)f(n)$。j 次迭代后，有 $f(n/b^j)\leqslant(c/a)^j f(n)$，或等价地，有 $a^j f(n/b^j)\leqslant c^j f(n)$。用此式对方程(4.7)作替换并化简，可以得到一个几何级数，与情况 1)不同的是，这个级数是下降的。

$$g(n)=\sum_{j=0}^{\log_b n-1}a^j f(n/b^j)\leqslant\sum_{j=0}^{\log_b n-1}c^j f(n)\leqslant f(n)\sum_{j=0}^{\infty}c^j$$

$$=f(n)\left(\frac{1}{1-c}\right)=O(f(n))$$

因为 c 是常量。如此可得对 b 的整数幂，有 $g(n)=\Theta(f(n))$。情况 3)得证。整个引理证明完毕。

■

现在就可证在 n 为 b 的整数幂时的主定理成立。

引理 4.4　设 $a\geqslant 1$，$b>1$ 是常量，$f(n)$ 是定义在 b 的整数幂上的非负函数。定义 $T(n)$ 如下：

$$T(n)=\begin{cases}\Theta(1) & \text{如果 } n=1\\ aT(n/b)+f(n) & \text{如果 } n=b^i\end{cases}$$

其中 i 是正整数。对于 b 的整数幂，$T(n)$可有如下渐近界：

1)若 $f(n)=O(n^{\log_b a-\varepsilon})$，$\varepsilon>0$，则 $T(n)=\Theta(n^{\log_b a})$

2)若 $f(n)=\Theta(n^{\log_b a})$，则 $T(n)=\Theta(n^{\log_b a}\lg n)$

3)若 $f(n)=\Omega(n^{\log_b a+\varepsilon})$，$\varepsilon>0$，且若对常数 $c<1$，和所有足够大的 n，$af(n/b)\leqslant cf(n)$，则

80 $T(n)=\Theta(f(n))$。

证明：用引理 4.3 中给出的界来对引理 4.2 中和式(4.6)求值。对情况 1)，有

$$T(n)=\Theta(n^{\log_b a})+O(n^{\log_b a})=\Theta(n^{\log_b a})$$

对情况 2)，有

$$T(n)=\Theta(n^{\log_b a})+\Theta(n^{\log_b a}\lg n)=\Theta(n^{\log_b a}\lg n)$$

对情况 3)，有

$$T(n)=\Theta(n^{\log_b a})+\Theta(f(n))=\Theta(f(n))$$

因为 $f(n)=\Omega(n^{\log_b a+\varepsilon})$。 ■

4.4.2　上取整函数和下取整函数

为使主定理的证明更完整，还要将分析扩展到主递归式中含上取整函数和下取整函数的情形，从而使递归式定义在所有的整数上，而不仅仅是定义在 b 的整数幂上。

对递归式

$$T(n)=aT(\lceil n/b\rceil)+f(n) \tag{4.10}$$

给出下界与对递归式

$$T(n)=aT(\lfloor n/b\rfloor)+f(n) \tag{4.11}$$

给出上界都是很容易的事，因为由界 $\lceil n/b\rceil\geqslant n/b$ 可通过情况 1)得到所需结果，界 $\lfloor n/b\rfloor\leqslant n/b$ 可通过情况 2)得到。找出(4.11)的下界与给出(4.10)的上界需要类似的技术，故只给出后一个界。

我们对图 4-3 中的递归树进行了修改，产生了如图 4-4 的递归树。从树根向下走时，我们得到对自变量的一系列递归调用：

81 $$n,\quad \lceil n/b\rceil,\quad \lceil\lceil n/b\rceil/b\rceil,\quad \lceil\lceil\lceil n/b\rceil/b\rceil/b\rceil,\quad \cdots$$

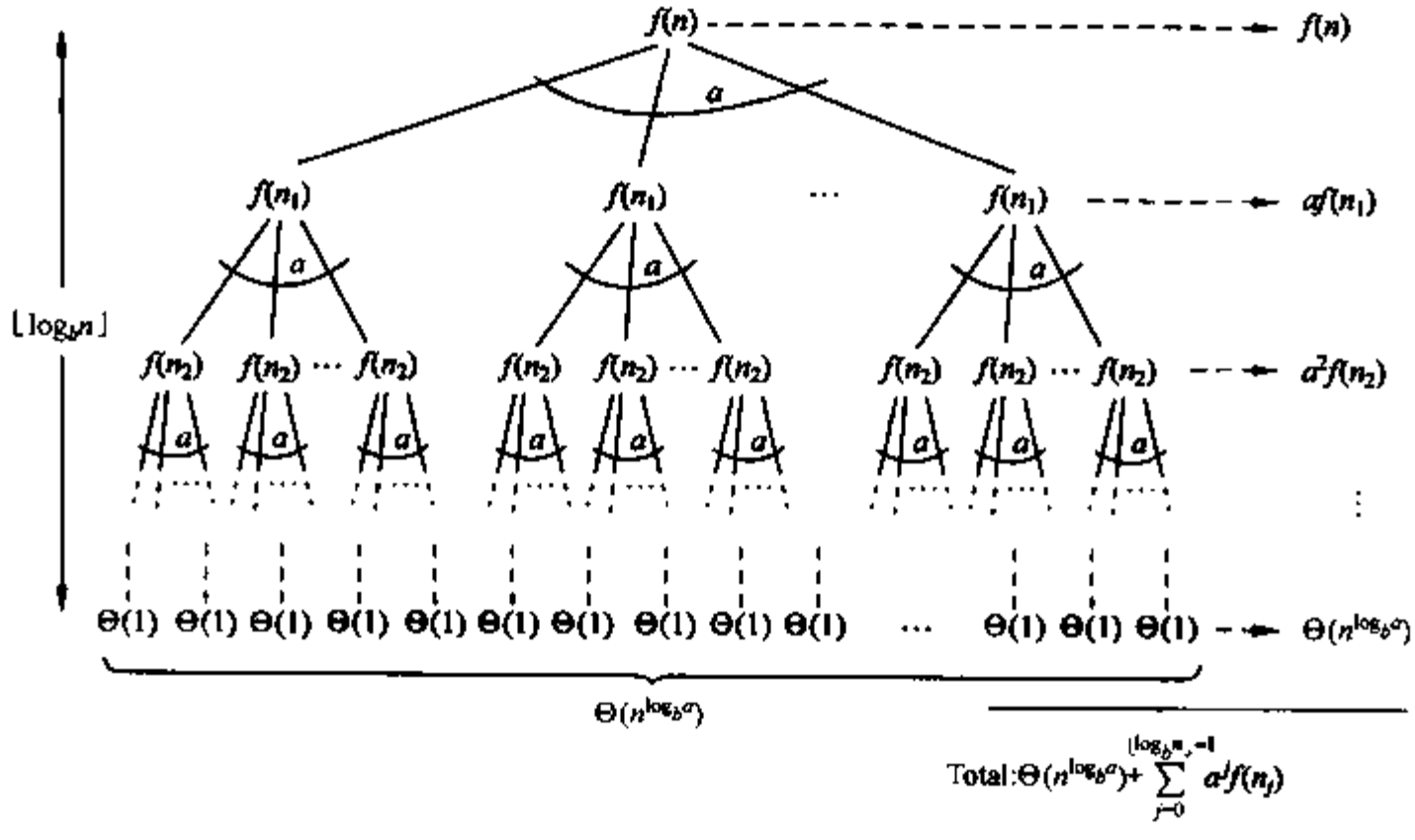


图 4-4　由 $T(n)=aT(\lceil n/b\rceil)+f(n)$产生的递归树。递归参数 n_j 由(4.12)式给出

用 n_j 来表示序列中第 j 个元素，即

$$n_j = \begin{cases} n & \text{如果 } j = 0 \\ \lceil n_{j-1}/b \rceil & \text{如果 } j > 0 \end{cases} \tag{4.12}$$

我们的第一个目标是决定使 n_k 为常数的深度 k。利用不等式 $\lceil x \rceil \leqslant x+1$ 得：

$$n_0 \leqslant n, \quad n_1 \leqslant \frac{n}{b}+1, \quad n_2 \leqslant \frac{n}{b^2}+\frac{1}{b}+1, \quad n_3 \leqslant \frac{n}{b^3}+\frac{1}{b^2}+\frac{1}{b}+1, \quad \cdots$$

82

一般地，

$$n_j \leqslant \frac{n}{b^j}+\sum_{i=0}^{j-1}\frac{1}{b^i} < \frac{n}{b^j}+\sum_{i=0}^{\infty}\frac{1}{b^i} = \frac{n}{b^j}+\frac{b}{b-1}$$

当 $j=\lfloor \log_b n \rfloor$ 时，有

$$n_{\lfloor \log_b n \rfloor} < \frac{n}{b^{\lfloor \log_b n \rfloor}}+\frac{b}{b-1} \leqslant \frac{n}{b^{\log_b n-1}}+\frac{b}{b-1} = \frac{n}{n/b}+\frac{b}{b-1}$$

$$= b+\frac{b}{b-1} = O(1)$$

所以，我们发现在深度为 $\lfloor \log_b n \rfloor$ 时，问题大小至多为常数。

从图 4-4 中，我们得到

$$T(n) = \Theta(n^{\log_b a})+\sum_{j=0}^{\lfloor \log_b n \rfloor -1} a^j f(n_j) \tag{4.13}$$

上式与方程(4.6)很相似，只是此处 n 可以是任意整数，而不限于 b 的整数幂。

现在利用类似于引理 4.3 的证明方法对公式(4.13)中的和式(4.14)求值

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor -1} a^j f(n_j) \tag{4.14}$$

先看情况 3)，如果对 $n>b+b/(b-1)$，有 $af(\lceil n/b \rceil) \leqslant cf(n)$，$c<1$ 为常数，则 $a^j f(n_j) \leqslant c^j f(n)$。这样，(4.14)中的和式就可如引理 4.3 一样地求值。对情况 2)，有 $f(n)=\Theta(n^{\log_b a})$。如果我们能证明 $f(n_j)=O(n^{\log_b a}/a^j)=O((n/b^j)^{\log_b a})$，则套用引理 4.3 中情况 2)的证明即可。注意这时有 $j \leqslant \lfloor \log_b n \rfloor$ 隐含着 $b^j/n \leqslant 1$。界 $f(n)=O(n^{\log_b a})$ 隐含着存在常数 $c>0$，使对足够大的 n_j，有

83

$$f(n_j) \leqslant c\left(\frac{n}{b^j}+\frac{b}{b-1}\right)^{\log_b a} = c\left(\frac{n}{b^j}\left(1+\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a}$$

$$= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\left(\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a}$$

$$\leqslant c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\frac{b}{b-1}\right)^{\log_b a} = O\left(\frac{n^{\log_b a}}{a^j}\right)$$

此处 $c(1+b/(b-1))^{\log_b a}$ 是个常数，如此情况 2)得证。情况 1)的证明几乎一样，关键在于证明界 $f(n_j)=O(n^{\log_b a-\varepsilon})$，这与情况 2)中对应部分的证明类似，但代数运算要更复杂。

到这里我们证明了对于整数 n 主定理的上界，其下界的证明也类似。

练习

*4.4-1　在方程(4.12)中，若 b 是个正整数而不是任意实数时，请给出 n_j 的一个简单而准确的表达式。

*4.4-2　证明：若 $f(n)=\Theta(n^{\log_b a}\lg^k n)$，其中 $k \geqslant 0$，则主定理中递归式的解为 $T(n)=\Theta(n^{\log_b a}\lg^{k+1} n)$。为简单起见，可以只对 b 的整数幂分析。

*4.4-3　在主定理的情况 3)中，条件 $af(n/b) \leqslant cf(n)$，$c<1$ 隐含着存在常数 $\varepsilon>0$ 使 $f(n)=$

84　$\Omega(n^{\log_b a+\varepsilon})$。据此请证明定理中的陈述过强了。

思考题

4-1　递归式的例子

给出下列递归式的渐近上下界。假设 $T(n)$是个常数，$n\leqslant 2$。使所给出的界尽量紧确。并给出证明。

a) $T(n)=2T(n/2)+n^3$

b) $T(n)=T(9n/10)+n$

c) $T(n)=16T(n/4)+n^2$

d) $T(n)=7T(n/3)+n^2$

e) $T(n)=7T(n/2)+n^2$

f) $T(n)=2T(n/4)+\sqrt{n}$

g) $T(n)=T(n-1)+n$

h) $T(n)=T(\sqrt{n})+1$

4-2　找出所缺的整数

某数组 $A[1..n]$含有所有从 0 到 n 的整数，但其中有一个整数不在数组中。通过利用一个辅助数组 $B[0..n]$来记录 A 中出现的整数，很容易在 $O(n)$时间内找出所缺的整数。但在这个问题中，我们却不能由一个单一操作来访问 A 中的一个完整整数，因为 A 中的元素是以二进制表示的。我们所能用的唯一操作就是“取 $A[i]$的第 j 位”，这个操作所花时间为常数。

证明：如果仅用此操作，仍能在 $O(n)$时间内找出所缺的整数。

4-3　参数传递的代价

整个这本书中，我们都假定过程调用中的参数传递所花时间为常数，即使所传递的是个 N 个元素的数组也是一样。这个假设对大多数系统都是有效的，因为当参数为数组时，
85　所传递的只是指向该数组的指针，而不是该数组本身。本题讨论三种参数传递策略：

1)数组由一个指针来传递。时间$=\Theta(1)$。

2)参数数组通过复制而传递。时间$=\Theta(N)$，N 是该数组的大小。

3)一个数组在被传递时，仅拷贝被调用过程可能引用的数组的子域。若传递的是子数组$A[p..q]$。时间$=\Theta(p-q+1)$。

a)考虑在一个已排序的数组中找一个数的递归二叉查找算法(见练习 2.3-5)。针对上面的三种参数传递策略，给出最坏情况运行时间的递归式，并给出其解的上界。可以设 N 为原问题的规模，n 为子问题的规模。

b)重做 2.3.1 节中 MERGE-SORT 的 a)部分。

4-4　更多递归式的例子

给出下列递归式 $T(n)$的渐近上下界。假设对足够小的 n，$T(n)$是常量。使所给出的界尽量紧确，并加以证明。

a) $T(n)=3T(n/2)+n\lg n$

b) $T(n)=5T(n/5)+n/\lg n$

c) $T(n)=4T(n/2)+n^2\sqrt{n}$

d) $T(n)=3T(n/3+5)+n/2$

e) $T(n)=2T(n/2)+n/\lg n$

f) $T(n)=T(n/2)+T(n/4)+T(n/8)+n$

g) $T(n)=T(n-1)+1/n$

h) $T(n)=T(n-1)+\lg n$

i) $T(n)=T(n-2)+2\lg n$

j) $T(n)=\sqrt{n}T(\sqrt{n})+n$

4-5　斐波那契数

我们已在递归式(3.21)中定义了斐波那契数，现在进一步介绍它们的性质。我们将用 86
生成函数技术来解斐波那契递归式。定义生成函数(或形式幂级数) $\mathcal{F}$ 如下

$$\mathcal{F}(z)=\sum_{i=0}^{\infty}F_i z^i=0+z+z^2+2z^3+3z^4+5z^5+8z^6+13z^7+21z^8+\cdots$$

其中，F_i 是第 i 个斐波那契数。

a)证明：$\mathcal{F}(z)=z+z\mathcal{F}(z)+z^2\mathcal{F}(z)$。

b)证明

$$\mathcal{F}(z)=\frac{z}{1-z-z^2}=\frac{z}{(1-\phi z)(1-\hat{\phi}z)}=\frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z}-\frac{1}{1-\hat{\phi}z}\right)$$

其中，$\phi=\dfrac{1+\sqrt{5}}{2}=1.61803\cdots$，$\hat{\phi}=\dfrac{1-\sqrt{5}}{2}=-0.61803\cdots$。

c)证明

$$\mathcal{F}(z)=\sum_{i=0}^{\infty}\frac{1}{\sqrt{5}}(\phi^i-\hat{\phi}^i)z^i$$

d)证明：对 $i>0$，$F_i=\phi^i/\sqrt{5}$ 截至最近的整数。(提示：$|\hat{\phi}|<1$)。

e)证明：对 $i\geqslant 0$，$F_{i+2}\geqslant\phi^i$。

4-6　VLSI 芯片测试

Diogenes 教授有 n 个被认为是完全相同的 VLSI⊖芯片，原则上它们是可以互相测试的。教授的测试装置一次可测二片，当该装置中放有两片芯片时，每一片就对另一片作测试并 87
报告其好坏。一个好的芯片总能够报告另一片的好坏，但一个坏的芯片的结果是不可靠的。这样，每次测试的四种可能结果如下：

A 芯片报告	B 芯片报告	结论
B 是好的	A 是好的	都是好的，或都是坏的
B 是好的	A 是坏的	至少一片是坏的
B 是坏的	A 是好的	至少一片是坏的
B 是坏的	A 是坏的	至少一片是坏的

a)证明若多于 $n/2$ 的芯片是坏的，在这种成对测试方式下，使用任何策略都不能确定哪个芯片是好的。假设坏的芯片可以联合起来欺骗教授。

b)假设有多于 $n/2$ 的芯片是好的，考虑从 n 片中找出一片好芯片的问题。证明 $\lfloor n/2\rfloor$ 对测试就足以使问题的规模降至近原来的一半。

c)假设多于 $n/2$ 片芯片是好的，证明好的芯片可用 $\Theta(n)$ 对测试找出。给出并解答表达

⊖ VLSI 代表“超大规模集成”，当今用于制作大多数微处理器的集成电路芯片技术。

测试次数的递归式。

4-7　Monge 矩阵

一个 $m\times n$ 的实数矩阵 A，如果对所有 i，j，k 和 l，$1\leqslant i<k\leqslant m$ 和 $1\leqslant j<l\leqslant n$，有

$$A[i,j]+A[k,l]\leqslant A[i,l]+A[k,j]$$

那么，此矩阵 A 为 Monge 矩阵。换句话说，每当我们从 Monge 矩阵中挑出两行与两列，并且考虑行列交叉处的 4 个元素，左上角与右下角元素的和小于或等于左下角与右上角元素的和。例如下面的矩阵是一个 Monge 阵。

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a)证明一个矩阵为 Monge 阵，当且仅当对所有 $i=1$，2，…，$m-1$ 和 $j=1$，2，…，$n-1$，有

88

$$A[i,j]+A[i+1,j+1]\leqslant A[i,j+1]+A[i+1,j]$$

（提示：在“仅当”部分，对行、列分别使用归纳法。）

b)下面的矩阵不是 Monge 阵。改变一个元素，把它变成 Monge 阵（提示：利用 a)的结论）

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c)假设 $f(i)$是第 i 行包含最左端最小值的列的索引值。证明对任何的 $m\times n$ Monge 矩阵，有 $f(1)\leqslant f(2)\leqslant\cdots\leqslant f(m)$。

d)以下是一段关于分治算法的描述，用来计算 $m\times n$ Monge 矩阵 A 的每一行的最左端最小值：

构造一个包含所有 A 的偶数行的子矩阵 A'。递归地计算 A'中每一行的最左端最小值。然后计算 A 中奇数行的最左端最小值。

解释如何能在 $O(m+n)$时间内计算出 A 的奇数行的最左端最小值？（假设偶数行的最左端最小值已知）

e)写出 d)部分所描述算法的运行时间的递归式，并证明其解为 $O(m+n\log m)$。

本章注记

递归式最早在 1202 年由 L. Fibonacci 进行了研究，斐波那契数就是用他的名字命名的。A. De Moivre 引入了生成函数（见思考题 4-5）来解递归式。主方法取自 Bentley，Haken 和 Saxe [41]，它提供了练习 4.4-2 所证明的扩展方法。Knuth[182]和 Liu[205]证明如何使用生成函数来解现行递归式。Purdom 和 Brown[252]，Graham，Knuth 和 Patashnik[132]包含解决递归式的扩展讨论。

包括 Akra 和 Bazzi[13]，Roura[262]，以及 Verma[306]在内的一些研究人员，都提供解分治递归式的方法，这些递归式比主方法能解决的要更一般些。我们在这里描述 Akra 和 Bazzi 的结论，它应用于如下形式的递归式

89

$$T(n) = \sum_{i=1}^{k} a_i T(\lfloor n/b_i \rfloor) + f(n) \tag{4.15}$$

其中 $k \geqslant 1$；所有的系数 a_i 都是正数，且它们的和至少为 1；所有的 b_i 至少为 2；$f(n)$有界，正值且非递减；对所有的常数 $c>1$，存在常数 n_0，$d>0$ 使得对任意的 $n \geqslant n_0$，有 $f(n/c) \geqslant df(n)$。这个方法可以应用在主方法不能应用的递归式上，例如 $T(n)=T(\lfloor n/3 \rfloor)+T(\lfloor 2n/3 \rfloor)+O(n)$。为解递归式(4.15)，首先找出 p 的值让 $\sum_{i=1}^{k} a_i b_i^{-p} = 1$。(这样的 p 总是存在，并且是唯一的正值。)于是，此递归式的解是

$$T(n) = \Theta(n^p) + \Theta\left(n^p \int_{n'}^{n} \frac{f(x)}{x^{p+1}} \mathrm{d}x\right)$$

其中 n'是个足够大的常数。Akra-Bazzi 方法有时候比较难以使用，但是它可以用来解将问题分割成不同大小的子问题的递归式。主方法用起来比较简单，但是只能应用在各个子问题的大小相同的情形。

90

第 5 章　概率分析和随机算法

本章介绍概率分析(probabilistic analysis)和随机算法(randomized algorithm)。读者如果不熟悉概率论的基本知识，应先阅读附录 C，其中复习这部分材料。概率分析和随机算法将在本书中多次提到。

5.1　雇用问题

假设你需要雇用一名新的办公室助理。你先前的雇用尝试都以失败告终，所以你决定找一个雇用代理。雇用代理每天给你推荐一个应聘者。你会面试这个人，然后决定要不要雇用他。你必须付给雇用代理一小笔费用来面试应聘者。要真正地雇用一个应聘者则要花更多的钱，因为你必须辞掉目前的办公室助理，还要付一大笔中介费给雇用代理。你的诺言是在任何时候，都要找到最佳人选来担任这项职务。因此，你决定在面试完每个应聘者后，如果这个应聘者比目前的办公助理更有资格，你就会辞掉目前的办公室助理，然后聘请这个新的应聘者。你愿意为这种策略而付出费用，但希望能够预测这种费用会是多少。

下面给出的 HIRE-ASSISTANT 过程以伪代码表达这种雇用策略。它假设应聘办公室助理工作的人编号为 1 到 n。此过程假设你能够在面试完应聘者 i 后，决定应聘者 i 是否是你见过的最适当人选。为了初始化，此程序建立一个虚拟的应聘者，编号为 0，他的应聘条件比所有其他的应聘者都差。


```
HIRE-ASSISTANT(n)
1  best ← 0    ▷ candidate 0 is a least-qualified dummy candidate
2  for i ← 1 to n
3      do interview candidate i
4         If candidate i is better than candidate best
5           then best ← i
6                hire candidate i
```

这个问题的费用模型与第 2 章描述的模型不同。我们所关心的不是 HIRE-ASSISTANT 的执行时间，而是面试和雇用所花的费用。表面上看，分析这个算法的费用看起来与分析合并排序等的执行时间有很大的不同，但是，不管我们是在分析费用或是执行时间，所使用的分析技术却是相同的。在任何情况中，我们都是在计算特定基本操作的执行次数。

面试的费用较低，譬如说为 c_i，而雇用的费用则较高，设为 c_h。假设 m 是已雇用的人数。那么这个算法的总费用就是 $O(nc_i+mc_h)$。不管雇用多少人，我们永远会面试 n 个应聘者，所以面试的费用永远是 nc_i。因此，我们只专注于分析 mc_h，即雇用的费用上。这个量在算法的每次执行中都会改变。

这个场景用来当作一般计算范式的模型。通常情况下我们需要检查序列中的每个成员，并且维护一个目前的“获胜者”，来找出序列中的最大或最小值。雇用问题是对哪一个成员当前获胜的更新频繁度建立模型。

最坏情况分析

在最坏情况下，我们雇用了每个面试的应聘者。当应聘者的资质逐渐递增时，就会出现这种情况，此时我们雇用了 n 次，总的费用是 $O(nc_h)$。

但是，较为合理的预期是应聘者并非总是以资质递增的次序出现的。事实上，我们既不能得知他们的出现次序，也不能控制这个次序。因此，通常我们预期的是一般或平均情况。

概率分析

概率分析是在问题的分析中应用概率技术。大多数情况下，我们使用概率分析来分析一个算法的运行时间。有时候也用它分析其他的量，例如程序 HIRE-ASSISTANT 中的雇用费用问题。为了进行概率分析，必须使用关于输入分布的知识或者对其做的假设。然后分析算法，计算出一个期望的运行时间。这个期望值通过对所有可能的输入分布算出。因此，实际上是将所有可能输入的运行时间作平均。 92

在确定输入的分布时必须非常小心。对于有些问题，我们对所有可能的输入集合可以作某种假定，也可以将概率分析作为一种手段来设计高效的算法，并加深对问题的认识。对于其他的一些问题，可能无法描述一个合理的输入分布，此时就不能使用概率分析方法。

在雇用问题中，可以假设应聘者是以随机顺序出现的。这一假设对于这个问题来说意味着什么呢？假定可以对任何两个应聘者进行比较，并决定哪一个更有资格；换言之，在所有应聘者的资格之间，存在着一个全序关系(全序的定义可参见附录 B)。因此可以使用从 1 到 n 的唯一号码来将应聘者排列名次，用 $rank(i)$ 表示应聘者 i 的名次，并约定较高的名次对应较有资格的应聘者。这个有序序列$\langle rank(1), rank(2), \cdots, rank(n)\rangle$是序列$\langle 1, 2, \cdots, n\rangle$的一个排列。说应聘者以随机的顺序出现，就等于说这个排名列表是数字 1 到 n 的 $n!$ 种排列中的任何一个。或者，也可以称这些排名构成一个均匀的随机排列；亦即，在 $n!$ 种可能的组合中，每一种都以相等的概率出现。

5.2 节包含对雇用问题的一个概率分析。

随机算法

为了利用概率分析，需要了解关于输入分布的一些情况。在许多情况下，我们对输入分布知之甚少。即使知道关于输入分布的某些信息，从计算上来说，可能也无法对这种分布知识建立模型。然而，通过使一个算法中某些部分的行为随机化，就常常可以利用概率和随机性作为算法设计和分析的工具。

在雇用问题中，看起来应聘者好像是以随机的顺序出现的，但是我们无法知道这是否正确。因此为了设计雇用问题的一个随机算法，必须对面试应聘者的次序有更大控制。所以要稍微改变这个模型。假设雇用代理有 n 个应聘者，而且事先给我们一份应聘者的名单。每天我们随机选择其中一个来面试。虽然我们并不了解任何关于应聘者的事项(除了他们的名字)，我们已经做了一个显著的改变。我们控制了应聘者的来到过程且强加了随机次序，而不是依赖于随机次序到达这个猜测。 93

更一般地，如果一个算法的行为不只是由输入决定，同时也由随机数生成器所产生的数值决定，则称这个算法是随机的。我们将假定有一个可以自由使用的随机数生成器 RANDOM。调用 RANDOM(a, b)将返回一个介于 a 与 b 之间的整数，而每个整数出现的机会相等。例如，RANDOM(0, 1)产生 0 的概率是 1/2，产生 1 的概率也是 1/2。调用 RANDOM(3, 7)将返回 3, 4, 5, 6, 7 中的任意一个，且每个出现的概率都是 1/5。每一次 RANDOM 返回的整数独立于上一次调用的返回值。可以将 RANDOM 想象成掷一个$(b-a+1)$面的骰子得到它出现的点数。(实际上，大多数编程环境都会提供一个伪随机数生成器，它是一个确定性的算法，但其返回值看起来是统计上随机的。)

练习

5.1-1　证明：假设在程序 HIRE-ASSISTANT 的第 4 行中，我们总是能够决定哪一个应聘者最佳，这就蕴含我们知道应聘者排名的总次序。

***5.1-2**　描述 RANDOM(a，b)过程的一种实现，它只调用 RANDOM(0，1)。作为 a 和 b 的函数，你的程序的期望运行时间是多少？

***5.1-3**　假设你希望以各 1/2 的概率输出 0 和 1。你可以自由使用一个输出 0 或 1 的过程 BIASED-RANDOM。它以概率 p 输出 1，以概率 $1-p$ 输出 0，其中 $0<p<1$，但是你并不知道 p 的值。给出一个利用 BIASED-RANDOM 作为子程序的算法，返回一个无偏向的结果，即以概率 1/2 返回 0，以概率 1/2 返回 1。作为 p 的函数，你的算法的期望运行时间是多少？

5.2　指示器随机变量

为了分析包括雇用问题在内的许多算法，我们将利用指示器随机变量(indicator random
94 variable)。它为概率与期望之间的转换提供了一个便利的方法。给定一个样本空间 S 和事件 A，那么事件 A 对应的指示器随机变量 $\mathrm{I}\{A\}$定义为

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{如果 } A \text{ 发生的话} \\ 0 & \text{如果 } A \text{ 不发生的话} \end{cases} \tag{5.1}$$

举一个简单的例子，确定在抛一枚均匀硬币时正面朝上的期望次数。样本空间为 $S=\{H, T\}$，定义一个随机变量 Y，取值 H 和 T 的概率各为 1/2。接下来定义指示器随机变量 X_H，它对应于硬币正面朝上的情况即事件 H。这个变量计算抛硬币时正面朝上的次数，如果正面朝上则其值为 1，否则为 0。写作：

$$X_H = \mathrm{I}\{Y = H\} = \begin{cases} 1 & \text{如果 } H \text{ 发生} \\ 0 & \text{如果 } T \text{ 发生} \end{cases}$$

在抛硬币时正面朝上的期望次数，就是指示器变量 X_H 的期望值：

$$\begin{aligned} \mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{Y = H\}] = 1 \cdot \Pr\{Y = H\} + 0 \cdot \Pr\{Y = T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) = 1/2 \end{cases}$$

因此抛一枚均匀的硬币时，正面朝上的期望次数是 1/2。如以下的引理所示，事件 A 对应的指示器随机变量的期望值等于事件 A 发生的概率。

引理 5.1　给定样本空间 S 和 S 中的事件 A，令 $X_A=\mathrm{I}\{A\}$，则 $\mathrm{E}[X_A]=\Pr\{A\}$。

证明：由公式(5.1)指示器随机变量的定义和期望值的定义，有

$$\mathrm{E}[X_A] = \mathrm{E}[\mathrm{I}\{A\}] = 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} = \Pr\{A\}$$

其中 $\overline{A}$ 表示 A 的补 $S-A$。 ■

95　在统计抛掷一枚硬币时正面朝上的期望次数这样的应用中，虽然指示器随机变量看起来很麻烦，但是它在分析重复随机试验中的情况时是比较有用的。例如，随机变量给我们一个求公式(C.36)结果的简单方法。在这个公式中，我们分别考虑出现 0 个、1 个、2 个、…正面朝上的概率，以便计算抛 n 次硬币时正面朝上的次数。但是，公式(C.37)给出的简单方法隐含使用了指示器随机变量。为让此论点更清楚，我们令指示器随机变量 X_i 对应于第 i 次抛硬币时正面朝上的事件：令 Y_i 表示第 i 次抛硬币输出结果的随机变量，有 $X_i=\mathrm{I}\{Y_i=H\}$。假设随机变量 X 表示 n 次抛硬币中出现正面的总次数，于是

$$X = \sum_{i=1}^{n} X_i$$

我们希望计算正面朝上的期望次数，所以我们使用对上面的等式两边取期望，得

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right]$$

等式左边是 n 个随机变量总和的期望值。由引理 5.1，容易计算出每个随机变量的期望值。根据反映了期望的线性性质的公式(C.20)，容易计算出总和的期望值：它等于 n 个随机变量期望值的总和。期望的线性性质利用了指示器随机变量作为有力的分析技术；即使随机变量之间存在依赖关系也成立。现在我们可以很容易地计算正面出现次数的期望值：

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} 1/2 = n/2$$

因此，和公式(C.36)中用到的方法相比，指示器随机变量极大地简化了计算过程。我们将在本书中一直使用指示器随机变量。

[96]

利用指示器随机变量分析雇用问题

下面再回到雇用问题上来。此时，我们希望计算雇用一个新的办公助理的期望次数。为了利用概率分析，假设应聘者以随机的顺序出现，如前一节所述。(5.3 节中去除这个假设。)令 X 作为一个随机变量，其值等于雇用一个新的办公助理的次数。然后，将应用公式(C.19)中的期望值的定义，得到

$$\mathrm{E}[X] = \sum_{x} x\Pr\{X = x\}$$

但是这一计算会很麻烦。我们将改用指示器随机变量来大大简化计算。

为了利用指示器随机变量，我们不是通过定义与雇用一个新的办公助理的次数对应的变量来计算 $\mathrm{E}[X]$，而是定义 n 个和每个应聘者是否被雇用对应的变量。特别地，令 X_i 对应于第 i 个应聘者被雇用这个事件的指示器随机变量。所以，

$$X_i = \mathrm{I}\{\text{第 } i \text{ 位应聘者被雇用}\} = \begin{cases} 1 & \text{如果第 } i \text{ 位应聘者被雇用} \\ 0 & \text{如果第 } i \text{ 位应聘者没有被雇用} \end{cases} \tag{5.2}$$

并且

$$X = X_1 + X_2 + \cdots + X_n \tag{5.3}$$

由引理 5.1，有

$$\mathrm{E}[X_i] = \Pr\{\text{应聘者 } i \text{ 被雇用}\}$$

于是，我们必须计算过程 HIRE-ASSISTANT 中第 5～6 行被执行的概率。

在第 5 行中，如果应聘者 i 胜过从 1 到 $i-1$ 的每一个应聘者，则应聘者 i 会被雇用。由于已经假设应聘者以随机的顺序出现，所以前 i 个应聘者也是以随机的顺序出现的。这些前 i 个应聘者中的任何一个都等可能地是目前最有资格的。应聘者 i 比从应聘者 1 到 $i-1$ 更有资格的概率是 $1/i$，因此也以 $1/i$ 的概率被雇用。由引理 5.1，可以得出结论

$$\mathrm{E}[X_i] = 1/i \tag{5.4}$$

现在可以计算 $\mathrm{E}[X]$ 了：

[97]

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] \tag{5.5}$$

$$= \sum_{i=1}^{n} 1/i = \ln n + O(1) \tag{5.6}$$

即使面试了 n 个人，平均起来看，实际上大约只雇用他们之中的 $\ln n$ 个人。我们用下面的引理来

总结这个结果。

引理 5.2　假设应聘者以随机的次序出现，算法 HIRE-ASSISTANT 总的雇用费用为 $O(c_h \ln n)$。

证明：由对雇用费用的定义及公式(5.6)，可以立即得到这个界。■

期望的雇用费用比最坏情况下的雇用费用 $O(nc_h)$有了显著的改善。

练习

5.2-1　在 HIRE-ASSISTANT 中，假设应聘者以随机的顺序出现，正好雇用 1 次的概率是多少？正好雇用 n 次的概率又是多少？

5.2-2　在 HIRE-ASSISTANT 中，假设应聘者以随机的顺序出现，正好雇用两次的概率是多少？

5.2-3　利用指示器随机变量来计算掷 n 次骰子总和的期望值。

5.2-4　利用指示器随机变量来解帽子保管问题(hat-check problem)：有 n 位顾客，他们每个人给餐厅负责保管帽子的服务生一顶帽子。服务生以随机的顺序将帽子归还给顾客。请问拿到自己帽子的客户的期望数目是多少？

98

5.2-5　假设 $A[1..n]$是由 n 个不同的数构成的数组。如果 $i<j$ 且 $A[i]>A[j]$，则称(i, j)对为 A 的逆序对(inversion)。(思考题 2-4 中有更多关于逆序对的例子。)假设 A 的元素选自$(1, 2, \cdots, n)$上的一个均匀随机排列。利用指示器随机变量来计算 A 中逆序对的期望数目。

5.3　随机算法

在上一节中，说明了了解输入的分布是如何有助于分析算法平均情况行为的。但是，许多时候我们无法得到有关输入分布的信息，因而不可能进行平均情况分析。如 5.1 节中所提到的，在这些情况下，可以考虑采用随机算法。

对于诸如雇用问题之类的问题，假设输入的所有排列都是等可能的往往是有益的。在这样的问题中，通过概率分析可以设计出随机算法。我们不是假设输入的一个分布，而是给定一个分布。特别地，在算法运行之前，我们先随机地排列应聘者，以强加所有排列都是等可能的这个特性。这个修改并没有改变雇用一个新的办公助理大约需要 $\ln n$ 次这个期望值。然而，这意味着对于所有的输入，我们都期望它是这种情况，而不只是对于那些具有特定分布的输入才有这个值。

现在，我们来进一步揭示概率分析和随机算法之间的区别。在 5.2 节中，我们宣称如果应聘者是以随机顺序出现的话，则雇用一个新的办公室助理的期望次数大约是 $\ln n$。注意这个算法是确定性的；对于任何特定的输入，雇用一个新的办公室助理的次数始终相同。此外，这个次数将随输入的变化而改变，而且依赖于各种应聘者的排名。既然它仅依赖于应聘者的排名，可以使用应聘者排名的有序序列来代表一个特定的输入，例如$(rank(1), rank(2), \cdots, rank(n))$。给定排名序列 $A_1=\langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle$，总是会雇用 10 次新的办公助理，因为每一个后来的应聘者都优于前一个，而且在算法的每次迭代中第 5～6 行都要被执行。给定排名序列 $A_2=\langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1\rangle$，总是会只雇用 1 次新的办公助理，即在第一次迭代中。给定排名序列 $A_3=\langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6\rangle$，会雇用 3 次新的办公助理，即在面试排名为 5、8 和 10 的 3 位应聘者的时候。回顾一下算法的费用，它依赖于雇用新的办公助理的次数，可以看到，有昂贵的输入(如 A_1)、不贵的输入(例如 A_2)、适中贵的输入(例如 A_3)。

99

下面再来考虑一下先对应聘者进行排列、再确定最佳应聘者的随机算法。此时随机发生在算法上，而不是发生在输入分布上。给定一个输入，例如上述的 A_3，我们无法说出最大值会被更新多少次，因为它在每次运行此算法时都不同。第一次在 A_3 上运行此算法时，可能会产生排

列 A_1 且执行 10 次更新；而第二次运行可能会产生排列 A_2 且只执行 1 次更新。第三次执行时，可能会产生其他次数的更新。每次运行这个算法，执行依赖于随机的选择，而且很可能和上一次算法的执行不同。对于这个算法以及许多其他的随机算法，没有特定的输入会引出它的最坏情况行为。即使你最坏的敌人也无法产生最糟的输入数列，因为随机的排列使得输入次序不相关。只有在随机数生成器产生一个“不幸运”的置换时，随机算法才运行得不好。

对于雇用问题，代码中唯一需要改动的地方是随机地排列应聘者数列。

```
RANDOMIZED-HIRE-ASSISTANT(n)
1  randomly permute the list of candidates
2  best ← 0   ▷ candidate 0 is a least-qualified dummy candidate
3  for i ← 1 to n
4       do interview candidate i
5          if candidate i is better than candidate best
6            then best ← i
7                 hire candidate i
```

凭借这个简单的改变，我们建立了一个随机算法，它的性能和假设应聘者以随机次序出现所得到的结果是一致的。

引理 5.3　过程 RANDOMIZED-HIRE-ASSISTANT 的期望雇用费用是 $O(c_h \ln n)$。

证明：在对输入数列加以排列之后，已经和 HIRE-ASSISTANT 的概率分析达到了相同的情况。 ■

引理 5.2 和引理 5.3 的比较显示出概率分析和随机算法的差别。在引理 5.2 中，我们在输入上做了假设。在引理 5.3 中，尽管随机化输入排列会花费一些额外的时间，但我们也没有做这种假设。在本节的余下部分里，要讨论关于随机地排列输入的一些事宜。 100

随机排列数组

许多随机算法通过排列给定的输入数组来使输入随机化。（还有其他使用随机的方式。）在这里我们将讨论两种随机化方法。不失一般性，假设给定一个数组 A，它包含元素 1 到 n。我们的目标是构造这个数组的一个随机排列。

一个常用的方法是为数组的每个元素 $A[i]$ 赋一个随机的优先级 $P[i]$，然后依据优先级对数组 A 中的元素进行排序。例如，如果初始数组 $A=\langle 1, 2, 3, 4\rangle$ 且选择随机的优先级 $P=\langle 36, 3, 97, 19\rangle$，将得出数列 $B=\langle 2, 4, 1, 3\rangle$，因为第 2 个优先级最小，接着是第 4 个，然后第 1 个，最后是第 3 个。称这个过程为 PERMUTE-BY-SORTING：

```
PERMUTE-BY-SORTING(A)
1  n ← length[A]
2  for i ← 1 to n
3       do P[i]=RANDOM(1, n^3)
4  sort A, using P as sort keys
5  return A
```

其中第 3 行选取一个在 1 到 n^3 之间的随机数。使用范围 1 到 n^3，是为了让 P 中的所有优先级尽可能是唯一的。（练习 5.3-5 让你证明所有元素都唯一的概率至少是 $1-1/n$，练习 5.3-6 则问你在有两个或更多的优先级相同的情况下，如何来实现这个算法。）假设所有的优先级都唯一。

此过程中耗时的步骤是第 4 行中的排序。在第 8 章中将看到，如果使用基于比较的排序，排序将花费 $\Omega(n \lg n)$ 的时间。这个下界可以达到，因为我们已经知道合并排序的时间代价为 $\Theta(n \lg n)$。

(在第二部分可以看到时间代价为 $\Theta(n \lg n)$的其他基于比较的排序算法。)在排序之后，如果 $P[i]$是第 j 个最小的优先级，那么 $A[i]$将在输出的位置 j 上。用这种方式我们得到了一个排列。还有待于进一步证明的是这个过程能产生均匀的随机排列，亦即，数字 1 到 n 的每一种排列都是等可能被产生的。

引理 5.4　假设所有的优先级都是唯一的，过程 PERMUTE-BY-SORTING 可以产生输入的
101 均匀随机排列。

证明：我们从考虑每个元素 $A[i]$得到第 i 个最小优先级的特殊排列开始，并证明这个排列发生的概率正好是 $1/n!$。对 $i=1, 2, \cdots, n$，令 X_i 代表元素 $A[i]$得到第 i 个最小优先级的事件。我们想计算对所有的 i，事件 X_i 发生的概率，也就是

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\}$$

利用练习 C.2-6，这个概率等于

$$\Pr\{X_1\} \cdot \Pr\{X_2 \mid X_1\} \cdot \Pr\{X_3 \mid X_2 \cap X_1\} \cdot \Pr\{X_4 \mid X_3 \cap X_2 \cap X_1\}$$
$$\cdots \Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} \cdots \Pr\{X_n \mid X_{n-1} \cap \cdots \cap X_1\}$$

因为 $\Pr\{X_1\}$是从 n 个元素的集合中随机选取的优先级是最小的概率，故 $\Pr\{X_1\}=1/n$。接下来有 $\Pr\{X_2 \mid X_1\}=1/(n-1)$，因为假定元素 $A[1]$有最小的优先级，则余下来的 $n-1$ 个元素有相等的可能成为第 2 小的优先级。一般地，对于 $i=2, 3, \cdots, n$，有 $\Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\}=1/(n-i+1)$。这是因为从 $A[1]$到 $A[i-1]$按顺序有前 $i-1$ 小的优先级，余下的 $n-(i-1)$元素中，每一个具有第 i 小优先级的可能性都相同。所以有

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} = \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\cdots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right) = \frac{1}{n!}$$

这样就证明了得到同样排列的概率是 $1/n!$。

可以扩展这个证明，使其对任何优先级的排列都起作用。考虑集合$\{1, 2, \cdots, n\}$的任何一个确定的排列 $\sigma=\langle\sigma(1), \sigma(2), \cdots, \sigma(n)\rangle$。用 r_i 代表赋予元素 $A[i]$的优先级的排名，其中第 j 小的元素的优先级名次为 j。如果定义 X_i 为元素 $A[i]$得到第 $\sigma(i)$优先级的事件，或$r_i=\sigma(i)$，那么同样的证明仍然适用。因此，如果要计算得到任何特殊排列的概率，这个计算和先前的计算完全相同，所以得到这个排列的概率也是 $1/n!$。■

有人可能会有这样的想法，即要证明一个排列是均匀随机排列，只要证明对于每个元素 $A[i]$，其处于位置 j 的概率是 $1/n$ 就足够了。练习 5.3-4 证明这个弱的条件实际上是不充分的。

102 产生随机排列的一个更好方法是原地排列给定的数列。程序 RANDOMIZE-IN-PLACE 在 $O(n)$时间内完成。在第 i 次迭代时，元素 $A[i]$是从元素 $A[i]$到 $A[n]$中随机选取的。第 i 次迭代之后，$A[i]$保持不变。

```
RANDOMIZE-IN-PLACE(A)
1  n ← length[A]
2  for i ← 1 to n
3       do swap A[i] ↔ A[RANDOM(i, n)]
```

我们将使用循环不变式来证明程序 RANDOMIZE-IN-PLACE 是能产生均匀随机排列的。给定包含 n 个元素的一个集合，k 排列是包含这 n 个元素中 k 个元素的序列(参见附录 B)。共有 $n!/(n-k)!$ 种可能的 k 排列。

引理 5.5　过程 RANDOMIZE-IN-PLACE 可以计算出一个均匀随机排列。

证明：我们使用如下的循环不变式：

在第 2～3 行 for 循环的第 i 次迭代之前，对每个可能的$(i-1)$排列，子数组 $A[1..i-1]$包含

这个$(i-1)$排列的概率是$(n-i+1)!/n!$。

我们需要证明这个不变式在第 1 次迭代之前为真，循环的每次迭代能够保持此不变式，并且在循环结束时，此不变式提供提供一个有用的属性来证明循环结束时的正确性。

初始化：考虑刚好在第 1 次循环迭代之前的情况，此时 $i=1$。由循环不变式可知，对每个可能的 0 排列，子数组 $A[1..0]$包含这个 0 排列的概率为$(n-i+1)!/n! = 1$。子数列 $A[1..0]$是空的子数组，且 0 排列也没有任何元素。所以 $A[1..0]$包含所有 0 排列的概率为 1，在第 1 次循环迭代之前循环不变式成立。

保持：假设刚好在第 $i-1$ 次迭代之前，每种可能的$(i-1)$排列出现在子数组 $A[1..i-1]$中的概率是$(n-i+1)!/n!$，我们要证明在第 i 次迭代之后，每种可能的 i 排列出现在子数组 $A[1..i]$中的概率是$(n-i)!/n!$。下一次迭代对 i 加 1 后，还将保持这个循环不变式。

我们再来看一看第 i 次迭代。考虑一个特殊的 i 排列，以$(x_1, x_2, \cdots, x_i)$来表示其中的元素。这个排列包含一个$(i-1)$排列$(x_1, x_2, \cdots, x_{i-1})$，其后接着算法在 $A[i]$里放置的值 x_i。用 E_1 为一个事件，它表示前 $i-1$ 迭代已经在 $A[1..i-1]$中构造了此特殊的$(i-1)$排列。由循环不变式，$\Pr\{E_1\}=(n-i+1)!/n!$。用 E_2 表示第 i 次迭代在 $A[i]$里放置值 x_i 的事件。当 E_1 和 E_2 正好都发生时，i 排列$(x_1, x_2, \cdots, x_i)$在 $A[1..i]$中形成，因此我们希望计算 $\Pr\{E_2 \cap E_1\}$。利用公式(C.14)，有 103

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\}\Pr\{E_1\}$$

概率 $\Pr\{E_2 \mid E_1\}$等于 $1/(n-i+1)$，因为在第 3 行中算法从位置 $A[i..n]$的 $n-i+1$ 个值中随机选取 x_i。因此有

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\}\Pr\{E_1\} = \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} = \frac{(n-i)!}{n!}$$

终止：在结束时，$i=n+1$，得子数组 $A[1..n]$是一个给定 n 排列的概率为$(n-n)!/n! = 1/n!$。因此，RANDOMIZE-IN-PLACE 是可以计算出一个均匀随机排列的。 ■

随机算法通常是解决问题的最简单也是最有效的方法。本书中有几处就要用到随机算法。

练习

5.3-1 Marceau 教授对引理 5.5 证明过程中使用的循环不变式表示异议。他对在第 1 次迭代之前循环不变式是否为真提出质疑。他的理由是人们可以容易地宣称空数组不包含 0 排列。因此空数组包含 0 排列的概率应该是 0，所以在第 1 次迭代之前循环不变式无效。请改写过程 RANDOMIZE-IN-PLACE，使其相关的循环不变式在第 1 次迭代之前对非空数组仍适用，并为你的过程修改引理 5.5 的证明。

5.3-2 Kelp 教授决定写一个过程来随机产生非同一排列(identity permutation)的任意排列。他提出了如下的过程：

```
PERMUTE-WITHOUT-IDENTITY(A)
1  n ← length[A]
2  for i ← 1 to n-1
3       do swap A[i] ↔ A[RANDOM(i+1, n)]
```

104

这段代码实现了 Kelp 教授的意图了吗？

5.3-3 假设不是将元素 $A[i]$与子数组 $A[i..n]$中的随机一个元素相交换，而是将它与数组任何位置上的随机元素相交换：

```
PERMUTE-WITH-ALL(A)
```

```
1  n ← length[A]
2  for i ← 1 to n
3       do swap A[i] ↔ A[RANDOM(1, n)]
```

这段代码会产生均匀随机排列吗？为什么会？或为什么不会？

5.3-4　Armstrong 教授建议使用下列过程来产生均匀随机排列：

```
PERMUTE-BY-CYCLIC(A)
1  n ← length[A]
2  offset ← RANDOM(1, n)
3  for i ← 1 to n
4       do dest ← i+offset
5          if dest>n
6            then dest ← dest−n
7          B[dest] ← A[i]
8  return B
```

证明任意元素 $A[i]$出现在 B 中任何特定位置的概率都是 $1/n$。然后通过证明其结果不是均匀随机排列来表明 Armstrong 教授错了。

*5.3-5　证明程序 PERMUTE-BY-SORTING 的数组 P 中，所有元素都唯一的概率至少为 $1-1/n$。

5.3-6　解释如何实现算法 PERMUTE-BY-SORTING，来处理两个或更多优先级相同的情况。亦即，即使有两个或更多的优先级相同，你的算法也必须产生一个均匀随机排列。

105

*5.4　概率分析和指示器随机变量的进一步使用

本节包含了较为高级的内容，它通过 4 个范例来进一步解释概率分析。第 1 个例子确定在一个有 k 个人的房间中，某两个人生日相同的概率。第 2 个例子讨论把球随机投入盒子的问题。第 3 个例子研究在抛硬币中出现连续正面的情况。最后一个例子分析雇用问题的一个变种，其中你必须在面试所有的应聘者之前做出决定。

5.4.1　生日悖论

我们的第一个例子是生日悖论。一个房间里的人数必须要达到多少，才能使有两个人生日相同的机会达到 50%？这个问题的答案少的有点让人吃惊。下面我们将看到，所出现的悖论就在于这个数目事实上远小于一年中的天数，甚至不足年内天数的一半。

为了回答这个问题，我们用整数 1，2，…，k 对房间里的人编号，其中 k 是房间里的总人数。另外，我们不考虑闰年的情况，且假设所有年份都有 $n=365$ 天。对于 $i=1, 2, \cdots, k$，令 b_i 表示 i 的生日，且 $1 \leqslant b_i \leqslant n$。同时，还假设各人的生日均匀分布在一年的 n 天中，因此 $\Pr\{b_i=r\}=1/n$，对 $i=1,2, \cdots k$，和 $r=1, 2, \cdots, n$ 成立。

两个人 i 和 j 的生日正好相同的概率依赖于生日的随机选择是否是独立的。从现在开始假设生日是独立的，则 i 和 j 的生日都落在同一天 r 上的概率为

$$\Pr\{b_i = r \text{ and } b_j = r\} = \Pr\{b_i = r\}\Pr\{b_j = r\} = 1/n^2$$

这样，他们的生日落在同一天的概率为

$$\Pr\{b_i = b_j\} = \sum_{r=1}^{n} \Pr\{b_i = r \text{ 和 } b_j = r\} = \sum_{r=1}^{n} (1/n^2) = 1/n \tag{5.7}$$

更直观地来看，一旦选定 b_i 后，b_j 被选在同一天的概率是 $1/n$。因而 i 和 j 有相同生日的概率与他们其中一个的生日落在给定一天的概率相同。注意这个巧合依赖于各人的生日独立这个

106

假设。

可以通过考察一个事件的补的方法，来分析 k 个人中至少有两人生日相同的概率。至少有两个人生日相同的概率等于 1 减去所有人的生日都互不相同的概率。k 个人各有互不相同生日的事件为

$$B_k = \bigcap_{i=1}^{k} A_i$$

其中 A_i 是指对所有 $j<i$，i 与 j 生日不同的事件。既然可将 B_k 写成 $B_k = A_k \cap B_{k-1}$，由公式(C.16)可得递归式

$$\Pr\{B_k\} = \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} \tag{5.8}$$

其中初始条件 $\Pr\{B_1\}=\Pr\{A_1\}=1$。换言之，对于 $i=1, 2, \cdots, k-1$，假设 $b_1, b_2, \cdots, b_{k-1}$ 互异，那么 $b_1, b_2, \cdots, b_k$ 是互异的生日的概率等于 $b_1, b_2, \cdots, b_{k-1}$ 互异的概率乘以在 $b_1, b_2, \cdots, b_{k-1}$ 互异的条件下 $b_k \neq b_i$ 的概率。

如果 $b_1, b_2, \cdots, b_{k-1}$ 互异，条件概率 $b_k \neq b_i (i=1, 2, \cdots, k-1)$ 为 $\Pr\{A_k \mid B_{k-1}\}=(n-k+1)/n$，这是因为 n 天中有 $n-(k-1)$ 天没被占用。对递归式(5.8)作迭代，得，

$$\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} = \Pr\{B_{k-2}\}\Pr\{A_{k-1} \mid B_{k-2}\}\Pr\{A_k \mid B_{k-1}\} \\
&\vdots \\
&= \Pr\{B_1\}\Pr\{A_2 \mid B_1\}\Pr\{A_3 \mid B_2\}\cdots\Pr\{A_k \mid B_{k-1}\} \\
&= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right) = 1 \cdot \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right)
\end{aligned}$$

由不等式(3.11)，$1+x \leqslant e^x$，得

$$\Pr\{B_k\} \leqslant e^{-1/n}e^{-2/n}\cdots e^{-(k-1)/n} = e^{-\sum_{i=1}^{k-1} i/n} = e^{-k(k-1)/2n} \leqslant 1/2$$

当 $-k(k-1)/2n \leqslant \ln(1/2)$ 时成立。当 $k(k-1) \geqslant 2n\ln 2$ 时或者解二次方程 $k \geqslant (1+\sqrt{1+(8\ln 2)n})/2$，107
k 个人生日均不同的概率至少为 1/2。当 $n=365$ 时，必须有 $k \geqslant 23$。因此，如果至少有 23 个人在一个房间里，那么至少有两个人生日相同的概率至少是 1/2。在火星上，一年有 669 个火星日，所以要达到相同效果必须有 31 个火星人。

利用指示器随机变量进行分析

利用指示器随机变量，可以给出生日悖论的一个简单而近似的分析。对房间里 k 个人中的每一对 (i, j)，$1 \leqslant i < j \leqslant k$，定义指示器随机变量 X_{ij} 如下

$$X_{ij} = \mathrm{I}\{i \text{ 和 } j \text{ 生日相同}\} = \begin{cases} 1 & \text{如果 } i \text{ 和 } j \text{ 生日相同} \\ 0 & \text{否则} \end{cases}$$

由公式(5.7)，两个人有相同生日的概率是 $1/n$，因此根据引理 5.1，有

$$\mathrm{E}[X_{ij}] = \Pr\{i \text{ 和 } j \text{ 生日相同}\} = 1/n$$

令 X 表示计数具有相同生日的两人对数目的随机变量，得

$$X = \sum_{i=1}^{k}\sum_{j=i+1}^{k} X_{ij}$$

对两边取期望并应用期望的线性性质，得到

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{k}\sum_{j=i+1}^{k} X_{ij}\right] = \sum_{i=1}^{k}\sum_{j=i+1}^{k}\mathrm{E}[X_{ij}] = \binom{k}{2}\frac{1}{n} = \frac{k(k-1)}{2n}$$

因此当 $k(k-1) \geqslant 2n$ 时，有相同生日的两人对的对子期望数目至少是 1 个。如果房间里至少有 $\sqrt{2n}+1$ 个人，就可以期望至少有两个人生日相同。对于 $n=365$，如果 $k=28$，具有相同生日的人的对子数期望值为 $(28\times 27)/(2\times 365) \approx 1.0356$。因此，如果至少有 28 个人，则可以期望至少 108

有一对人的生日相同。在火星上，一年有669个火星日，至少要有38个火星人。

第一种分析仅利用了概率，给出了为使存在至少一对人生日相同的概率大于1/2所需的人数；第二种分析使用了指示器随机变量，给出了所期望的相同生日数为1时的人数。虽然两种情况下人的准确数目不等，但它们在渐近意义上是相等的，都是$\Theta(\sqrt{n})$。

5.4.2　球与盒子

现在我们来考虑这样一个过程，即把相同的球随机投到b个盒子里的过程，其中盒子编号为1，2，…，b。每次投球都是独立的，所投的球等可能落在每一个盒子中。球落在任一个盒子中的概率为$1/b$。因此，投球的过程是一组伯努利试验（参见附录C.4），每次成功的概率为$1/b$，此处成功是指球落入指定的盒子中。这个模型对分析散列技术（参见第11章）特别有用，而且我们可以回答关于这个投球过程的各种有趣的问题。（思考题C-1提出了关于球和盒子的另外一些问题。）

*有多少球落在给定的盒子里？*落在给定盒子里的球数服从二项分布$b(k; n, 1/b)$。如果投n个球，公式(C.36)告诉我们，落在给定盒子中的球数的期望值是n/b。

*在给定的盒子里至少有一个球之前，平均至少要投多少个球？*要投的个数服从几何分布，概率为$1/b$，根据公式(C.31)，成功前的期望个数是$1/(1/b)=b$。

*在每个盒子里至少有一个球之前，要投多少个球？*我们称在一次投球中球落在空盒子里为“击中”。我们想知道的是为了取得b次击中所需的期望投球次数n。

击中次数可以用来将n次投球划分为几个阶段。第i个阶段包括从第$(i-1)$次击中到i次击中之间的投球。第1阶段包含第1次投球，因为第1次投球时所有的盒子都是空的，肯定是一次击中。对第i阶段的每一次投球，有$i-1$个盒子有球，$b-i+1$个盒子是空的。这样，对第i阶段的所有投球，得到一次击中的概率为$(b-i+1)/b$。

用n_i表示第i阶段中的投球次数。于是，为得到b次击中所需的投球次数为$n=\sum_{i=1}^{b} n_i$。每个随机变量n_i都服从几何分布，成功的概率是$(b-i+1)/b$，根据公式(C.31)有

109

$$E[n_i]=\frac{b}{b-i+1}$$

由期望的线性性质，可得：

$$E[n]=E\left[\sum_{i=1}^{b} n_i\right]=\sum_{i=1}^{b} E[n_i]=\sum_{i=1}^{b}\frac{b}{b-i+1}=b\sum_{i=1}^{b}\frac{1}{i}=b(\ln b+O(1))$$

最后一行是根据调和级数的界(A.7)得来的。因此，在我们期望每个盒子里都有一个球之前，大约要投$b\ln b$次。这个问题也称为*赠券收集者问题*（coupon collector's problem），意思是一个人如果想要集齐b种不同赠券中的每一种，大约要有$b\ln b$张随机得到的赠券才能成功。

5.4.3　序列

设想你抛一枚均匀硬币n次。你期望看到连续正面的最长序列有多长？答案是$\Theta(\lg n)$，如以下分析所示。

首先证明出现正面的最长序列的期望长度为$O(\lg n)$。每次抛硬币时出现正面的概率是1/2。令A_{ik}为这样的事件：长度至少为k的正面序列开始于第i次抛掷，或更准确地说，k次连续的硬币抛掷i，$i+1$，…，$i+k-1$得到的都是正面，此处$1\leqslant k\leqslant n$且$1\leqslant i\leqslant n-k+1$。因为每次抛硬币是相互独立的，对任何给定事件$A_{ik}$，所有$k$次投掷都是正面的概率是

$$\Pr\{A_{ik}\}=1/2^k \tag{5.9}$$

对$k=2\lceil\lg n\rceil$

$$\Pr\{A_{i,2\lceil\lg n\rceil}\} = 1/2^{2\lceil\lg n\rceil} \leqslant 1/2^{2\lg n} = 1/n^2$$

因此，长度至少为 $2\lceil\lg n\rceil$ 的一个正面序列起始于位置 i 的概率是很小的。这种序列开始的位置至 110
多有 $n-2\lceil\lg n\rceil+1$ 个。长度至少为 $2\lceil\lg n\rceil$ 的正面序列开始于任一位置的概率为

$$\Pr\left\{\bigcup_{i=1}^{n-2\lceil\lg n\rceil+1} A_{i,2\lceil\lg n\rceil}\right\} \leqslant \sum_{i=1}^{n-2\lceil\lg n\rceil+1} 1/n^2 < \sum_{i=1}^{n} 1/n^2 = 1/n \tag{5.10}$$

因为根据布尔不等式(C.18)，一组事件的并集的概率至多是各个事件的概率之和。(注意即使对不是独立的事件，布尔不等式也成立。)

现在利用不等式(5.10)来给出最长序列长度的界。对 $j=0, 1, 2, \cdots, n$，令 L_j 表示最长正面序列的长度正好是 j 的事件，并且假设最长序列的长度是 L。由期望值的定义，

$$\mathrm{E}[L] = \sum_{j=0}^{n} j\Pr\{L_j\} \tag{5.11}$$

我们可以尝试用类似于不等式(5.10)所计算的每个 $\Pr\{L_j\}$ 的上界来求这个和。不幸的是，这个方法将导致弱的界。不过，可以利用从上述分析得到的一些直观知识来得到一个好的界。从非形式化的角度来看，我们观察到在公式(5.11)的总和中，没有任何一项同时让 j 和 $\Pr\{L_j\}$ 因子都是大的。为什么呢？当 $j \geqslant 2\lceil\lg n\rceil$ 时，$\Pr\{L_j\}$ 就会很小；当 $j < 2\lceil\lg n\rceil$ 时，j 就很小。更形式化一点，注意到对于 $j=0, 1, 2, \cdots, n$，事件 L_j 是不相交的，因此长度至少为 $2\lceil\lg n\rceil$ 的正面序列开始于任一位置的概率为 $\sum_{j=2\lceil\lg n\rceil}^{n} \Pr\{L_j\}$。根据不等式(5.10)，得 $\sum_{j=2\lceil\lg n\rceil}^{n} \Pr\{L_j\} < 1/n$。同时注意到 $\sum_{j=0}^{n} \Pr\{L_j\} = 1$，有 $\sum_{j=0}^{2\lceil\lg n\rceil-1} \Pr\{L_j\} \leqslant 1$。于是，有：

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j\Pr\{L_j\} = \sum_{j=0}^{2\lceil\lg n\rceil-1} j\Pr\{L_j\} + \sum_{j=2\lceil\lg n\rceil}^{n} j\Pr\{L_j\} \\
&< \sum_{j=0}^{2\lceil\lg n\rceil-1} (2\lceil\lg n\rceil)\Pr\{L_j\} + \sum_{j=2\lceil\lg n\rceil}^{n} n\Pr\{L_j\} \qquad \boxed{111} \\
&= 2\lceil\lg n\rceil \sum_{j=0}^{2\lceil\lg n\rceil-1} \Pr\{L_j\} + n\sum_{j=2\lceil\lg n\rceil}^{n} \Pr\{L_j\} \\
&< 2\lceil\lg n\rceil \cdot 1 + n \cdot (1/n) = O(\lg n)
\end{aligned}$$

正面序列的长度超过 $r\lceil\lg n\rceil$ 的机会随着 r 而很快地减少。对 $r \geqslant 1$，由 $r\lceil\lg n\rceil$ 个正面组成的序列开始于位置 i 的概率为

$$\Pr\{A_{i,r\lceil\lg n\rceil}\} = 1/2^{r\lceil\lg n\rceil} \leqslant 1/n^r$$

因此，最长的序列至少长 $r\lceil\lg n\rceil$ 的概率至多是 $n/n^r = 1/n^{r-1}$，或等价地，最长序列的长度小于 $r\lceil\lg n\rceil$ 的概率至少为 $1-1/n^{r-1}$。

看一个例子，对 $n=1000$ 次硬币抛掷，出现一系列最少为 $2\lceil\lg n\rceil = 20$ 次正面的概率至多是 $1/n = 1/1000$。出现长度超过 $3\lceil\lg n\rceil = 30$ 的正面序列的概率至多是 $1/n^2 = 1/1\,000\,000$。

现在我们证明一个补充的下界：在抛 n 次硬币试验中，最长的正面序列的期望长度为 $\Omega(\lg n)$。为证明这个界，通过将 n 次投掷划分成大约 n/s 个组的 s 次抛掷，看长度为 s 的序列。如果选择 $s = \lfloor(\lg n)/2\rfloor$，可以证明这些组中至少有一组全是正面，因此，很可能最长序列的长度至少是 $s = \Omega(\lg n)$。然后，我们将证明最长的序列的期望长度是 $\Omega(\lg n)$。

将 n 次硬币抛掷划分成至少 $\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor$ 个组的 $\lfloor(\lg n)/2\rfloor$ 连续抛掷，并约束没有组都是正面的概率。根据公式(5.9)，从位置 i 开始的组都是正面的概率为

$$\Pr\{A_{i,\lfloor(\lg n)/2\rfloor}\} = 1/2^{\lfloor(\lg n)/2\rfloor} \geqslant 1/\sqrt{n}$$

因此长度至少为 $\lfloor(\lg n)/2\rfloor$ 的正面序列不从位置 i 开始的概率是 $1-1/\sqrt{n}$。既然 $\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor$ 个组是由彼此互斥、独立的抛掷硬币形成的，其中每一个组都不是长为 $\lfloor(\lg n)/2\rfloor$ 的序列的概率至多为

$$(1-1/\sqrt{n})^{\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor}\leqslant(1-1/\sqrt{n})^{n/\lfloor(\lg n)/2\rfloor-1}\leqslant(1-1/\sqrt{n})^{2n/\lg n-1}$$

112 $$\leqslant e^{-(2n/\lg n-1)/\sqrt{n}}=O(e^{-\lg n})=O(1/n)$$

此证明用到了不等式(3.11)，即 $1+x\leqslant e^x$，也用到了你可能想验证的一个事实：对足够大的 n 有 $(2n/\lg n-1)/\sqrt{n}\geqslant\lg n$。

因此，最长的序列超过 $\lfloor(\lg n)/2\rfloor$ 的概率为

$$\sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n}\Pr\{L_j\}\geqslant 1-O(1/n) \tag{5.12}$$

现在就可以计算最长序列的期望长度的下界了，从公式(5.11)开始使用类似于上界分析的方式：

$$\begin{aligned}\mathrm{E}[L]&=\sum_{j=0}^{n}j\Pr\{L_j\}=\sum_{j=0}^{\lfloor(\lg n)/2\rfloor}j\Pr\{L_j\}+\sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n}j\Pr\{L_j\}\\&\geqslant\sum_{j=0}^{\lfloor(\lg n)/2\rfloor}0\cdot\Pr\{L_j\}+\sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n}\lfloor(\lg n)/2\rfloor\Pr\{L_j\}\\&=0\cdot\sum_{j=0}^{\lfloor(\lg n)/2\rfloor}\Pr\{L_j\}+\lfloor(\lg n)/2\rfloor\sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n}\Pr\{L_j\}\\&\geqslant 0+\lfloor(\lg n)/2\rfloor(1-O(1/n))=\Omega(\lg n)\qquad\text{(根据不等式(5.12))}\end{aligned}$$

和生日悖论一样，我们可以利用指示器随机变量来得到一个简单而近似的分析。令 $X_{ik}=\mathrm{I}\{A_{ik}\}$ 表示对应于序列长度至少为 k 的序列开始于第 i 次抛硬币的指示器随机变量。为了统计这些序列的总数，定义

$$X=\sum_{i=1}^{n-k+1}X_{ik}$$

取期望并利用期望的线性性质，有

113 $$\mathrm{E}[X]=\mathrm{E}\left[\sum_{i=1}^{n-k+1}X_{ik}\right]=\sum_{i=1}^{n-k+1}\mathrm{E}[X_{ik}]=\sum_{i=1}^{n-k+1}\Pr\{A_{ik}\}=\sum_{i=1}^{n-k+1}1/2^k=\frac{n-k+1}{2^k}$$

通过代入不同的 k 值，可以计算出长为 k 的序列的期望数目。如果这个数较大(远大于 1)，那么很多长为 k 的序列期望会出现而且发生的概率很高。如果这个数较小(远小于 1)，那么很少长为 k 的序列期望会出现而且发生的概率很低。如果 $k=c\lg n$，对某个正常数 c，有：

$$\mathrm{E}[X]=\frac{n-c\lg n+1}{2^{c\lg n}}=\frac{n-c\lg n+1}{n^c}=\frac{1}{n^{c-1}}-\frac{(c\lg n-1)/n}{n^{c-1}}=\Theta(1/n^{c-1})$$

如果 c 较大，长为 $c\lg n$ 的序列的期望数量将很少，而且结论是它们不大可能发生。另一方面，如果 $c<1/2$，那么得 $\mathrm{E}[X]=\Theta(1/n^{1/2-1})=\Theta(n^{1/2})$，而且期望会有大量长为 $(1/2)\lg n$ 的序列。因此，这种长度的序列很可能发生。通过这些粗略的估计，可以得出结论：最长序列的期望长度是 $\Theta(\lg n)$。

5.4.4　在线雇用问题

作为最后一个例子，我们来考虑雇用问题的一个变形。假设现在我们不希望面试所有的应聘者来找到最好的一个，也不希望因为不断有更好的申请者出现而不停地雇用新人解雇旧人。我们愿意雇用接近最好的应聘者，只雇用一次。我们必须遵守公司的一个要求：在每次面试后，必须或者立即提供职位给应聘者，或者告诉应聘者他们将无法得到这份工作。在最小化面试次

数和最大化雇用应聘者的质量两方面如何取得平衡?

可以通过以下方式来对这个问题建模。在面试一个应聘者之后，我们能够给他一个分数；令 $score(i)$ 表示给第 i 为应聘者的分数，并且假设没有两个应聘者的分数相同。在面试 j 个应聘者之后，我们知道其中哪一个分数最高，但是不知道在剩余的 $n-j$ 个应聘者中会不会有更高分数的应聘者。我们决定采用这样一个策略：选择一个正整数 $k<n$，面试前 k 个应聘者然后拒绝他们，再雇用其后比前面的应聘者有更高分数的第一个应聘者。如果结果是最好的应聘者在前 k 个面试的之中，那么我们将雇用第 n 个应聘者。这个策略形式化地表示在如下所示的过程 ON-LINE-MAXIMUM(k, n)中。该过程返回的是我们希望雇用的应聘者的下标值。

```
ON-LINE-MAXIMUM(k, n)
1  bestscore ← −∞
2  for i ← 1 to k
3       do if score(i)>bestscore
4            then bestscore ← score(i)
5  for i ← k+1 to n
6       do if score(i)>bestscore
7            then return i
8  return n
```

对每一个可能的 k 值，我们希望确定雇用到最好的应聘者的概率。然后选择最佳的 k 值，并用此值来实现这个策略。先假设 k 是固定的。令 $M(j)=\max_{1\leqslant i\leqslant j}\{score(i)\}$ 表示应聘者 1 到 j 中的最高分数。令 S 表示我们成功选择最好的应聘者的事件，S_i 表示当最好的应聘者是第 i 个面试者时成功的事件。由于不同的 S_i 不相交，有 $\Pr\{S\}=\sum_{i=1}^{n}\Pr\{S_i\}$。注意到当最好的应聘者是前 k 个应聘者中的一个时，我们不会成功，有 $\Pr\{S_i\}=0$，$i=1,2,\cdots,k$。于是得到

$$\Pr\{S\}=\sum_{i=k+1}^{n}\Pr\{S_i\} \tag{5.13}$$

现在我们来计算 $\Pr\{S_i\}$。为了当第 i 个应聘者是最好的时成功，有两件事情必须发生。首先，最好的应聘者必须在位置 i 上，用事件 B_i 表示。其次，算法不能选择在位置 $k+1$ 到 $i-1$ 中的任何一个应聘者，而这个选择仅发生在当 j 满足 $k+1\leqslant j\leqslant i-1$ 时，程序第 6 行有 $score(j)<bestscore$。(因为分数是唯一的，可以忽略 $score(j)=bestscore$ 的可能性。)换言之，所有 $score(k+1)$ 到 $score(i-1)$ 的值都必须小于 $M(k)$；如果其中有大于 $M(k)$ 的数，将返回第一个大于 $M(k)$ 的数的下标。用 O_i 表示在位置 $k+1$ 到 $i-1$ 中没有任何应聘者被选取的事件。幸运的是，事件 B_i 和 O_i 是独立的。事件 O_i 只依赖于位置 1 到 $i-1$ 中数值的相对次序，而 B_i 只依赖于位置 i 的数值是否大于所有其他位置的数值。位置 1 到 $i-1$ 中各数值的相对次序如何，并不应影响位置 i 的值是否大于位置 1 到 $i-1$ 中的所有数值，而且位置 i 的值也不会影响位置 1 到 $i-1$ 中值的次序。因此，应用公式(C.15)得

$$\Pr\{S_i\}=\Pr\{B_i\cap O_i\}=\Pr\{B_i\}\Pr\{O_i\}$$

$\Pr\{B_i\}$ 的概率显然是 $1/n$，因为最大值等可能地是 n 个位置中的任何一个。如果事件 O_i 要发生，在位置 1 到 $i-1$ 中的最大值必须在前 k 个位置的一个，而且最大值等可能地是 $i-1$ 个位置中的任何一个。因此，$\Pr\{O_i\}=k/(i-1)$，$\Pr\{S_i\}=k/(n(i-1))$。利用公式(5.13)，有

$$\Pr\{S\}=\sum_{i=k+1}^{n}\Pr\{S_i\}=\sum_{i=k+1}^{n}\frac{k}{n(i-1)}=\frac{k}{n}\sum_{i=k+1}^{n}\frac{1}{i-1}=\frac{k}{n}\sum_{i=k}^{n-1}\frac{1}{i}$$

我们利用积分来近似约束这个和数的上界和下界。根据不等式(A.12)，有

$$\int_k^n \frac{1}{x}\mathrm{d}x \leqslant \sum_{i=k}^{n-1} \frac{1}{i} \leqslant \int_{k-1}^{n-1} \frac{1}{x}\mathrm{d}x$$

解这些定积分可以得到下面的界：

$$\frac{k}{n}(\ln n - \ln k) \leqslant \Pr\{S\} \leqslant \frac{k}{n}(\ln(n-1) - \ln(k-1))$$

这提供了 $\Pr\{S\}$的一个相当紧确的界。由于我们希望最大化成功的概率，因而主要关注如何选取 k 的值，使其能够最大化 $\Pr\{S\}$的下界。(除此之外，下界表达式比上界表达式容易最大化。)将表达式$(k/n)(\ln n - \ln k)$对 k 求导，得

116

$$\frac{1}{n}(\ln n - \ln k - 1)$$

令此导数等于 0，我们看到当 $\ln k = \ln n - 1 = \ln(n/\mathrm{e})$，或等价地，当 $k = n/\mathrm{e}$ 时，概率的下界最大化。因此，如果用 $k = n/\mathrm{e}$ 来实现我们的策略，则可以以至少为 $1/\mathrm{e}$ 的概率，成功地雇用到最有资格的应聘者。

练习

5.4-1　一个房间里必须要有多少人，才能让某人和你生日相同的概率至少为 1/2？必须要有多少人，才能让至少两个人生日为 7 月 4 日的概率大于 1/2？

5.4-2　假设将球投入到 b 个盒子里。每一次投掷都是独立的，并且每个球落入任何盒子的机会都相等。在至少有一个盒子包含两个球之前，期望的投球次数是多少？

***5.4-3**　在生日悖论的分析中，要求各生日彼此独立这一点是否是很重要的？或者，是不是只要每两个人的生日互相独立就足够了？证明你的答案。

***5.4-4**　一个聚会需要邀请多少人，才能让其中很可能有 3 个人的生日相同？

***5.4-5**　在大小为 n 的集合中，一个 k 串实际上是一个 k 排列的概率为多少？这个问题和生日悖论有什么关系？

***5.4-6**　假设将 n 个球投入 n 个盒子里，每次投球都是独立的，并且每个球落入任何盒子的机会都相等。空盒子的期望数量是多少？正好有一个球的盒子的期望数量又是多少？

***5.4-7**　为使序列长度的下界变得更加准确，请证明在 n 次均匀硬币的抛掷中，不出现比 $\lg n - 2\lg\lg n$ 更长的连续正面序列的概率小于 $1/n$。

117

思考题

5-1　概率计数

利用一个 b 位的计数器，一般只能计数到 $2^b - 1$，而用 R. Morris 的概率计数法，则可以计到一个大得多的值，但代价是精度有所损失。

对 $i = 0, 1, \cdots, 2^b - 1$，令计数器的值 i 表示计数 n_i，且各 n_i 构成了一个非负的递增数列。假设计数器的初值为 0，代表计数 $n_0 = 0$。作用于计数器上的 INCREMENT 操作以概率的方式包含值 i。如果 $i = 2^b - 1$，则报告溢出错误。否则，计数器以概率 $1/(n_{i+1} - n_i)$ 增加 1，以概率 $1 - 1/(n_{i+1} - n_i)$保持不变。

对于所有的 $i \geqslant 0$，如果选择 $n_i = i$，则此计数器就是一个平常的计数器。如果选择 $n_i = 2^{i-1}(i > 0)$，或 $n_i = F_i$(第 i 个斐波那契数，见 3.2 节)，则会出现一些有趣的情况。

对这个问题，假设 n_{2^b-1}已足够大，从而使得发生溢出错误的概率可以忽略。

a)证明在执行了 n 次 INCREMENT 操作之后，计数器所表示的数的期望值正好是 n。

b)对有计数器所表示的计数的方差的分析要依赖于 n_i 的序列。让我们来看一个简单情况，对所有 $i \geqslant 0$，$n_i = 100i$。在执行了 n 次 INCREMENT 操作之后，估计计数器所表示的数的方差。

5-2　搜索无序数组

这个问题将分析三个算法，它们在一个包含 n 个元素的无序数组 A 中查找一个值 x。

考虑如下的随机算法：挑选 A 中一个随机的下标 i。如果 $A[i] = x$，则终止；否则继续挑选一个新的随机下标。重复挑选随机下标，直到找到一个下标 j 使 $A[j] = x$，或者我们已经检查过 A 中的每一个元素。注意每次都是从下标的整个集合中挑选，所以有可能会不止一次地检查某个元素。

a)写出过程 RANDOM-SEARCH 的伪代码来实现上述策略。确保当 A 的所有下标都被挑选过时，你的算法即应终止。 118

b)假定刚好有一个下标 i 使 $A[i] = x$。在找到 x 或算法 RANDOM-SEARCH 结束之前，必须挑选 A 的下标的期望数目是多少？

c)假设有 $k \geqslant 1$ 个下标 i 使 $A[i] = x$，推广你对(b)部分的解答。在找到 x 或算法 RANDOM-SEARCH 结束之前，必须挑选 A 的下标的期望数目是多少？你的解答应该是 n 与 k 的函数。

d)假设没有下标 i 使 $A[i] = x$。在检查完 A 的所有元素或算法 RANDOM-SEARCH 结束之前，必须挑选 A 的下标的期望数目是多少？

现在考虑一个确定性的线性查找算法，称之为 DETERMINISTIC-SEARCH。此算法在 A 中顺序地查找 x，即顺序地检查 $A[1]$，$A[2]$，$A[3]$，…，$A[n]$，直到找到 $A[i] = x$，或者已经到达数组的末尾而仍未找到 x 时为止。假设输入数组的所有排列都是等可能的。

e)假设刚好有一个下标 i 使 $A[i] = x$。DETERMINISTIC-SEARCH 的期望运行时间是多少？DETERMINISTIC-SEARCH 的最坏情况运行时间又是多少？

f)假设有 $k \geqslant 1$ 个下标 i 使 $A[i] = x$，推广你对(e)部分的解答。DETERMINISTIC-SEARCH 的期望运行时间是多少？DETERMINISTIC-SEARCH 的最坏情况运行时间又是多少？你的解答应该是 n 与 k 的函数。

g)假设没有下标 i 使 $A[i] = x$。DETERMINISTIC-SEARCH 的期望运行时间是多少？DETERMINISTIC-SEARCH 的最坏情况运行时间又是多少？

最后，考虑一个随机算法 SCRAMBLE-SEARCH，它先将输入数组随机排列，然后在排列的结果数组上，运行上述的确定性线性搜索算法。

h)假设 k 为使 $A[i] = x$ 的下标 i 的值，给出在 $k = 0$ 和 $k = 1$ 情况下，算法 SCRAMBLE-SEARCH 的最坏情况运行时间和期望运行时间。推广你的解答以处理 $k \geqslant 1$ 的情况。

i)你会使用 3 种搜索算法中的哪一个？给出你的答案。 119

本章注记

Bollobás[44]、Hofri[151]和 Spencer[283]介绍了大量的高等概率技术。随机算法的优点在 Karp[174]和 Rabin[253]中有讨论和综述。Motwani 和 Raghavan[228]的教科书中给出有关随机算法的详细论述。

雇用问题的一些变种已经得到广泛研究。这些问题通常被称为“秘书问题”。Ajtai、Meggido 和 Waarts[12]的论文给出了这一领域中的一个例子。 120

第二部分　排序和顺序统计学

引　言

这一部分将给出几个解决以下排序问题的算法：

输入：n 个数的序列$\langle a_1, a_2, \cdots, a_n\rangle$。

输出：输入序列的一个重排$\langle a'_1, a'_2, \cdots, a'_n\rangle$，使 $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$。

输入序列通常是一个 n 元数组，但也可能由其他形式来表示，如链表。

输入数据的结构

在实际中，待排序的数很少是孤立的值，它们通常是一个称为记录的数据集的一部分。每一个记录有一个关键字 key，它是待排序的值。记录的其他数据称为卫星数据，即它们通常以 key 为中心传送。在一个排序的算法中，当交换关键字时，卫星数据也必须交换。如果记录都很大，我们可以交换一组指向各个记录的指针而不是记录本身，以求将数据移动量减少到最小。

在一定意义上，正是这些实现细节才使得一个完整的程序不同于算法。不管我们要排序的是单个的数值还是包含数值的大型记录，就排序的方法来说它们都是一样的。因而，为了集中考虑排序问题，我们一般都假设输入仅由数值构成。将对数字的排序算法转换为对记录排序的程序是很直接的。当然，在具体的工程条件下，实际的程序设计可能还会遇到其他难以捉摸的挑战。

为什么要研究排序

许多计算机科学家认为，排序算法是算法学习中最基本的问题。原因有以下几个：

- 有时候应用程序本身就需要对信息进行排序。例如：为了准备客户账目，银行需要根据支票的号码对支票排序。
- 许多算法通常把排序作为关键子程序。例如，在一个绘制互相重叠的图形对象的程序中，可能需要根据一个“在上方”(above)关系将各对象排序，以便自下而上地绘出对象。在本书的许多算法中，都将排序作为子程序来使用。
- 现在已经有很多的排序算法，它们采用各种技术。事实上，在算法设计中使用的很多重要技术在已经发展多年的排序算法中早已用到了。所以，排序是一个具有历史意义的问题。
- 排序是一个我们可以证明其非平凡下界的问题(在第 8 章中会看到)。我们的最佳上界能够与这个非平凡下界渐近地相等，这就意味着我们的排序算法是渐近最优的。此外，可以利用排序过程的下界来证明其他一些问题的下界。
- 在实现排序算法时很多工程问题即浮出水面。对于某个特定的应用场景来说，最快的排序算法可能与许多因素有关：譬如关键字值和卫星数据的先验知识，主机存储器层次结构(高速缓存和虚拟存储)以及软件环境。很多的这些问题最好在设计算法时解决，而不

是和编码混在一起。

排序算法

在第2章中，介绍了两种对 n 个实数进行排序的算法。插入排序的最坏情况运行时间为 $\Theta(n^2)$，但其算法的内循环是紧密的，对小规模输入来说是一个快速的原地排序算法。(在排序输入数组时，只有常数个元素被存放到数组以外的空间中去。)合并排序有着较好的渐近运行时间 $\Theta(n\lg n)$，但其中的 MERGE 程序不在原地操作。

在本部分中，我们要介绍另外两种对任意实数进行排序的算法。第6章介绍堆排序，它可以在 $O(n\lg n)$ 时间内对 n 个数进行原地排序。这个算法用到了一种重要的称为堆的数据结构，还要
124 用它实现优先级队列。

第7章介绍快速排序，它是另一种对 n 个数进行原地排序的算法，但是它的最坏情况运行时间为 $\Theta(n^2)$。它的平均运行时间是 $\Theta(n\lg n)$，在实际中常常优于堆排序算法。像插入排序算法一样，快速排序的代码也比较紧凑，所以它的运行时间中隐含的常数因子就很小。对于大输入数组的排序来说，这是一个很常用的算法。

插入排序、合并排序、堆排序和快速排序都是比较排序(comparison sort)：它们通过对数组中的元素进行比较来实现排序。为研究比较排序算法的性能极限，第8章一开始就介绍决策树模型。利用这个模型，我们证明任何对 n 个输入比较排序算法的最坏情况运行时间的下界为 $\Omega(n\lg n)$，说明堆排序和合并排序都是渐近最优的比较排序算法。

第8章接下去说明如果我们能通过利用非比较的其他方法来获得有关输入数组中的排序信息，则可以突破 $\Omega(n\lg n)$ 的下界。例如，计数排序算法假定输入数取自集合 $\{0, 1, 2, \cdots, k\}$。通过利用数组下标来确定元素的相对次序，该算法可在 $\Theta(k+n)$ 时间内完成对 n 个数的排序。这样，当 $k=O(n)$ 时，计数排序的运行时间就与输入数组的规模成线性关系。另一个相关的算法是基数排序算法，它可以用来扩大计数使用的范围。如果有 n 个整数要排序，每个整数都有 d 位，且每位都取自集合 $\{0, 1, 2, \cdots, k\}$，则基数排序就可以在 $\Theta(d(n+k))$ 时间内完成排序。当 d 是个常数，k 是 $O(n)$ 时，基数排序就以线性时间运行。第三种算法是桶排序，它要求对各个数在输入数组中的概率分布有所了解。它可以对均匀分布在半开区间 $[0, 1)$ 上的 n 个实数以平均情况时间 $O(n)$ 进行排序。

顺序统计学

在一个由 n 个数构成的集合上，第 i 个顺序统计(the ith order statistic)是集合中第 i 小的数。当然，我们也可通过对输入进行排序，并标出排序输出结果中的第 i 个元素来选择第 i 个顺序统计。如果对输入的分布不作任何假设，这个方法的运行时间就是如第8章证明的下界 $\Omega(n\lg n)$。

在第9章中读者将会看到，即使输入数组中的各元素为任意实数，我们仍能在 $O(n)$ 时间内找到第 i 小的元素。我们将给出一个算法，伪代码很紧凑，最坏情况运行时间为 $\Theta(n^2)$，但平均情况下为线性时间。另外，还将给出一个更复杂的算法，其最坏情况运行时间为 $O(n)$。

背景知识

虽然本部分的大多数内容都不依赖于艰深的数学知识，但是有几节需要用到一定的数学技
125 巧。特别是在对快速排序、桶排序和顺序统计量算法进行平均情况分析时，用到了概率论方面的
~ 知识(相关内容可参见附录C和第5章中的概率分析和随机算法的材料)。顺序统计学的最坏情况
126 线性时间算法的分析，包含了比本篇其他最坏情况分析更复杂的数学内容。

第6章 堆 排 序

本章要介绍另一种排序算法，即堆排序(heapsort)。像合并排序而不像插入顺序，堆排序的运行时间为$O(n\lg n)$。像插入排序而不像合并排序，它是一种原地(in place)排序算法：在任何时候，数组中只有常数个元素存储在输入数组以外。这样，堆排序就把我们讨论过的两种排序算法的优点结合起来。

堆排序还引入另一种算法设计技术：利用某种数据结构(在此算法中为"堆")来管理算法执行中的信息。堆数据结构不只是在堆排序中有用，还可以构成一个有效的优先队列。堆数据结构在后续章节的算法中还将重复出现。

"堆"这个词最初是在堆排序中提出的，但后来就逐渐指"废料收集存储区"，就像程序设计语言 Lisp 和 Java 中所提供的设施那样。我们这里的堆数据结构不是废料收集存储区；本书中以后任何地方提到堆结构，都是指本章定义的结构。

6.1 堆

(二叉)堆数据结构是一种数组对象，如图 6-1 所示，它可以被视为一棵完全二叉树(见 B.5.3 节)。树中每个结点与数组中存放该结点值的那个元素对应。树的每一层都是填满的，最后一层可能除外(最后一层从一个结点的左子树开始填)。表示堆的数组 A 是一个具有两个属性的对象：$length[A]$是数组中的元素个数，$heap\text{-}size[A]$是存放在 A 中的堆的元素个数。就是说，虽然 $A[1..length[A]]$中都可以包含有效值，但 $A[heap\text{-}size[A]]$之后的元素都不属于相应的堆，此处 $heap\text{-}size[A] \leqslant length[A]$。树的根为 $A[1]$，给定了某个结点的下标 i，其父结点 PARENT(i)、左儿子 LEFT(i)和右儿子 RIGHT(i)的下标可以简单地计算出来：

127

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i+1
```

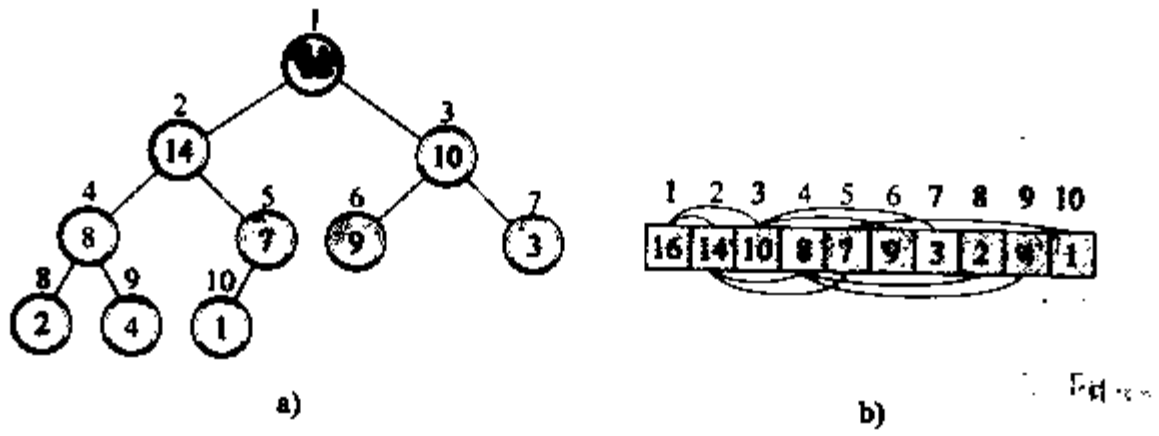


图 6-1 一个最大堆(大根堆)可被看作 a)一棵二叉树和b)一个数组。圆圈中的数字表示树中每个结点存储的值，结点上方的数字表示对应的数组下标。数组上下的连线表示父子关系，且父结点总在子结点的左边。这棵树的高度为 3，存储值为 8 的 4 号结点的高度为 1

在大多数计算机上，LEFT 过程可以在一条指令内计算出 $2i$，方法是将 i 的二进制表示左移 1 位。类似地，RIGHT 过程也可以通过将 i 的二进制表示左移 1 位并在低位中加 1，快速计算出 $2i+1$。PARENT 过程则可以通过把 i 右移 1 位而得到 $\lfloor i/2 \rfloor$。在一个好的堆排序的实现中，这三个过程通常是用"宏"过程或是"内联"过程实现的。

二叉堆有两种：最大堆和最小堆(小根堆)。在这两种堆中，结点内的数值都要满足堆特性，其细节则视堆的种类而定。在最大堆中，最大堆特性是指除了根以外的每个结点 i，有

128　$A[\text{PARENT}(i)] \geqslant A[i]$

即某个结点的值至多是和其父结点的值一样大。这样，堆中的最大元素就存放在根结点中；并且，在以某一个结点为根的子树中，各结点的值都不大于该子树根结点的值。最小堆的组织方式则刚好相反；最小堆特性是指除了根以外的每个结点 i，有

$$A[\text{PARENT}(i)] \leqslant A[i]$$

最小堆的最小元素是在根部。

在堆排序算法中，我们使用的是最大堆。最小堆通常在构造优先队列时使用，具体将在 6.5 节中讨论。对于某个特定的应用，我们将确切地指明需要的是最大堆还是最小堆。当某一性质既适合于最大堆也适合于最小堆时，我们就只使用"堆"这个词。

堆可以被看成是一棵树，结点在堆中的高度定义为从本结点到叶子的最长简单下降路径上边的数目；定义堆的高度为树根的高度。因为具有 n 个元素的堆是基于一棵完全二叉树的，因而其高度为 $\Theta(\lg n)$(见练习 6.1-2)。我们将看到，堆结构上的一些基本操作的运行时间至多与树的高度成正比，为 $O(\lg n)$。本章的下面部分将给出五个基本过程，并说明它们在排序算法和优先级队列数据结构中如何使用。

- MAX-HEAPIFY 过程，其运行时间为 $O(\lg n)$，是保持最大堆性质的关键
- BUILD-MAX-HEAP 过程，以线性时间运行，可以在无序的输入数组基础上构造出最大堆
- HEAPSORT 过程，运行时间为 $O(n \lg n)$，对一个数组原地进行排序
- MAX-HEAP-INSERT，HEAP-EXTRACT-MAX，HEAP-INCREASE-KEY 和 HEAP-MAXIMUM 过程的运行时间为 $O(\lg n)$，可以让堆结构作为优先队列使用。

练习

6.1-1　在高度为 h 的堆中，最多和最少的元素个数是多少？

129　6.1-2　证明：含 n 个元素的堆的高度为 $\lfloor \lg n \rfloor$。

6.1-3　证明：在一个最大堆的某棵子树中，最大元素在该子树的根上。

6.1-4　在一个最大堆中，假设其所有元素都不相同，那么其最小元素可能存在于堆的哪些地方？

6.1-5　一个已排好序的数组是一个最小堆吗？

6.1-6　序列(23，17，14，6，13，10，1，5，7，12)是一个最大堆吗？

6.1-7　证明：当用数组表示存储了 n 个元素的堆时，叶子结点的下标是 $\lfloor n/2 \rfloor+1$，$\lfloor n/2 \rfloor+2$，…，n。

6.2　保持堆的性质

MAX-HEAPIFY 是对最大堆进行操作的重要的子程序。其输入为一个数组 A 和下标 i。当 MAX-HEAPIFY 被调用时，我们假定以 LEFT(i)和 RIGHT(i)为根的两棵二叉树都是最大堆，但这时 $A[i]$可能小于其子女，这样就违反了最大堆性质。MAX-HEAPIFY 让 $A[i]$在最大堆中

"下降"，使以 i 为根的子树成为最大堆。

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l≤heap-size[A]and A[l]>A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r≤heap-size[A]and A[r]>A[largest]
 7     then largest ← r
 8  if largest≠i
 9     then exchange A[i]↔A[largest]
10          MAX-HEAPIFY(A, largest)
```

图 6 2 描述了 MAX-HEAPIFY 的过程。在算法的每一步里，从元素 $A[i]$，$A[\text{LEFT}(i)]$和 $A[\text{RIGHT}(i)]$中找出最大的，并将其下标存在 *largest* 中。如果 $A[i]$是最大的，则以 i 为根的子 130
树已是最大堆，程序结束。否则，i 的某个子结点中有最大元素，则交换 $A[i]$和 $A[largest]$，从而使 i 及其子女满足堆性质。下标为 *largest* 的结点在交换后的值是 $A[i]$，以该结点为根的子树又有可能违反最大堆性质。因而要对该子树递归调用 MAX-HEAPIFY。

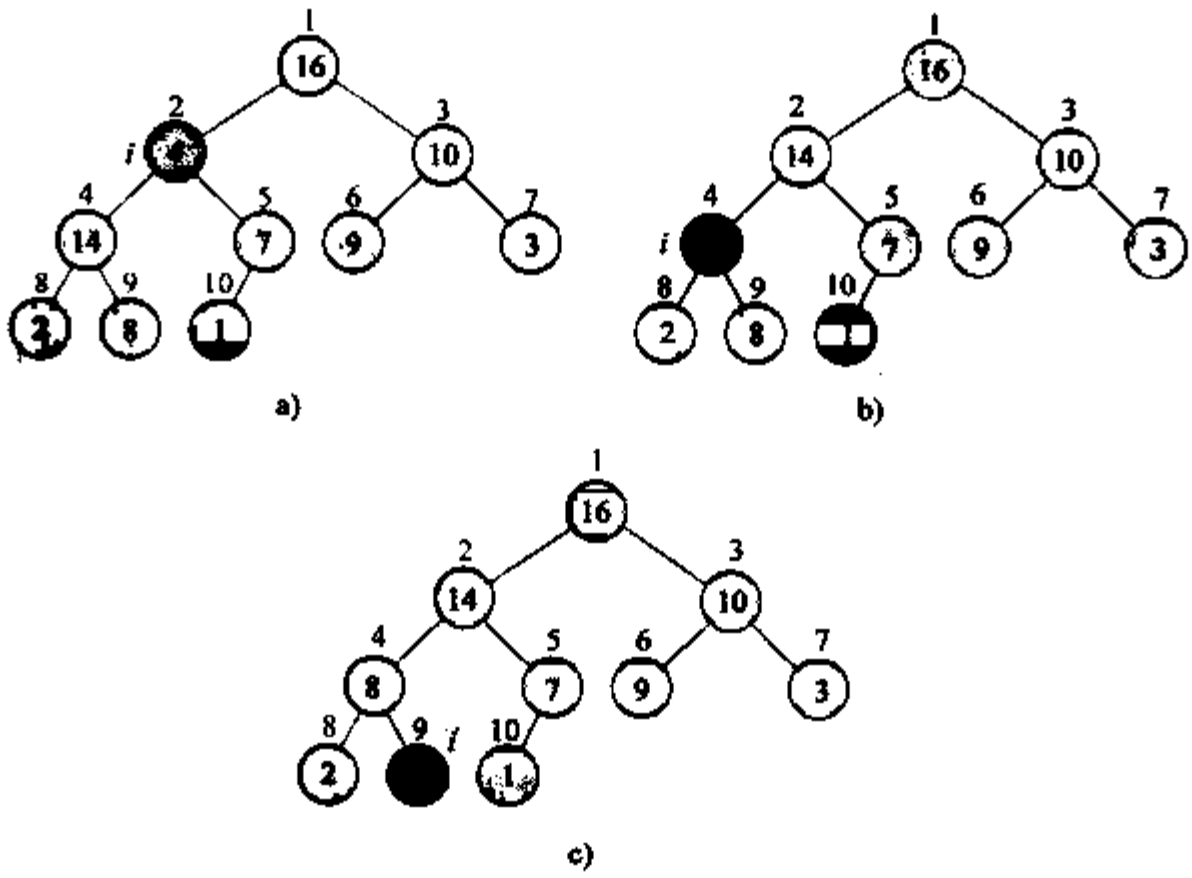


图 6-2 当 *heap-size*[A]=10 时，MAX-HEAPIFY(A, 2)的作用过程。a)初始构造，在结点 $i=2$ 处 $A[2]$违反了最大堆性质，因为它不大于它的两个子女。在 b)中通过交换$A[2]$与 $A[4]$，在结点 2 处恢复了最大堆性质，但又在结点 4 处违反了最大堆性质。现在递归调用 MAX-HEAPIFY(A, 4)，置 $i=4$，c)中交换了 $A[4]$和 $A[9]$，结点 4 的最大堆性质得到恢复，递归调用 MAX-HEAPIFY(A, 9)对该数据结构不会再引起任何变化

当 MAX-HEAPIFY 作用在一棵以结点 i 为根的、大小为 n 的子树上时，其运行时间为调整元素 $A[i]$、$A[\text{LEFT}(i)]$和 $A[\text{RIGHT}(i)]$的关系时所用时间 $\Theta(1)$，再加上对以 i 的某个子结点为根的子树递归调用 MAX-HEAPIFY 所需的时间。i 结点的子树大小至多为 $2n/3$(最坏情况发生在最底层恰好半满的时候)，那么 MAX-HEAPIFY 的运行时间可由下式描述： 131

$$T(n) \leqslant T(2n/3) + \Theta(1)$$

根据主定理的情况2(定理4.1)，该递归式的解为 $T(n)=O(\lg n)$。或者说，MAX-HEAPIFY 作用于一个高度为 h 的结点所需的运行时间为 $O(h)$。

练习

6.2-1 利用图 6-2 作为范例，图示出 MAX-HEAPIFY(A，3)作用于数组 $A=\langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$的过程。

6.2-2 由过程 MAX-HEAPIFY 开始，写出进行对应的最小堆操作的 MIN-HEAPIFY(A，i)过程的伪代码，并比较 MIN-HEAPIFY 与 MAX-HEAPIFY 的运行时间。

6.2-3 当元素 $A[i]$比其两子女的值都大时，调用 MAX-HEAPIFY(A，i)的效果是什么？

6.2-4 对 $i>heap\text{-}size[A]/2$，调用 MAX-HEAPIFY(A，i)的结果怎样？

6.2-5 MAX-HEAPIFY 的代码效率较高，但第 10 行中的递归调用可能例外，它可能使某些编译程序产生出低效的代码。请用迭代的控制结构(循环)取代递归结构，从而写一个更为高效的 MAX-HEAPIFY。

6.2-6 证明：对一个大小为 n 的堆，MAX-HEAPIFY 的最坏运行时间为 $\Omega(\lg n)$。(提示：对于 n 个结点的堆，恰当地设置每个结点的值，使得从根结点到叶结点的路径上的每个结点都递归调用 MAX-HEAPIFY)

6.3　建堆

我们可以自底向上地用 MAX-HEAPIFY 来将一个数组 $A[1..n]$(此处 $n=length[A]$)变成一

132 个最大堆。由练习 6.1-7 可以知道，子数组 $A[(\lfloor n/2\rfloor+1)..n]$中的元素都是树中的叶子，因此每个都可看作是只含一个元素的堆。过程 BUILD-MAX-HEAP 对树中的每一个其他结点都调用一次 MAX-HEAPIFY。

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3        do MAX-HEAPIFY(A, i)
```

图 6-3 给出了 BUILD-MAX-HEAP 作用过程的一个例子。

为了证明 BUILD-MAX-HEAP 的正确性，我们使用如下的循环不变式：

在第 2～3 行中 for 循环的每一次迭代开始时，结点 $i+1$，$i+2$，…，n 都是一个最大堆的根。

我们需要证明在第一次循环迭代之前，这个不变式已为真。每次循环迭代都能保持此不变式，并且在循环结束时，这个不变式会提供一个很有用的属性来显示程序的正确性。

初始化：在第一轮循环迭代之前，$i=\lfloor n/2\rfloor$。结点$\lfloor n/2\rfloor+1$，$\lfloor n/2\rfloor+2$，…，n 都是叶结点，也是平凡最大堆的根。

保持：要证明每次迭代都保持了循环不变式，注意到结点 i 的子结点的编号均比 i 大。于是，根据循环不变式，这些子结点都是最大堆的根。这也是调用函数 MAX-HEAPIFY(A，i)，以使结点 i 成为最大堆的根的前提条件。此外，MAX-HEAPIFY 的调用保持了结点 $i+1$，$i+2$，…，n 为最大堆的根的性质。在 for 循环中递减 i，即为下一次迭代重新建立了循环不变式。

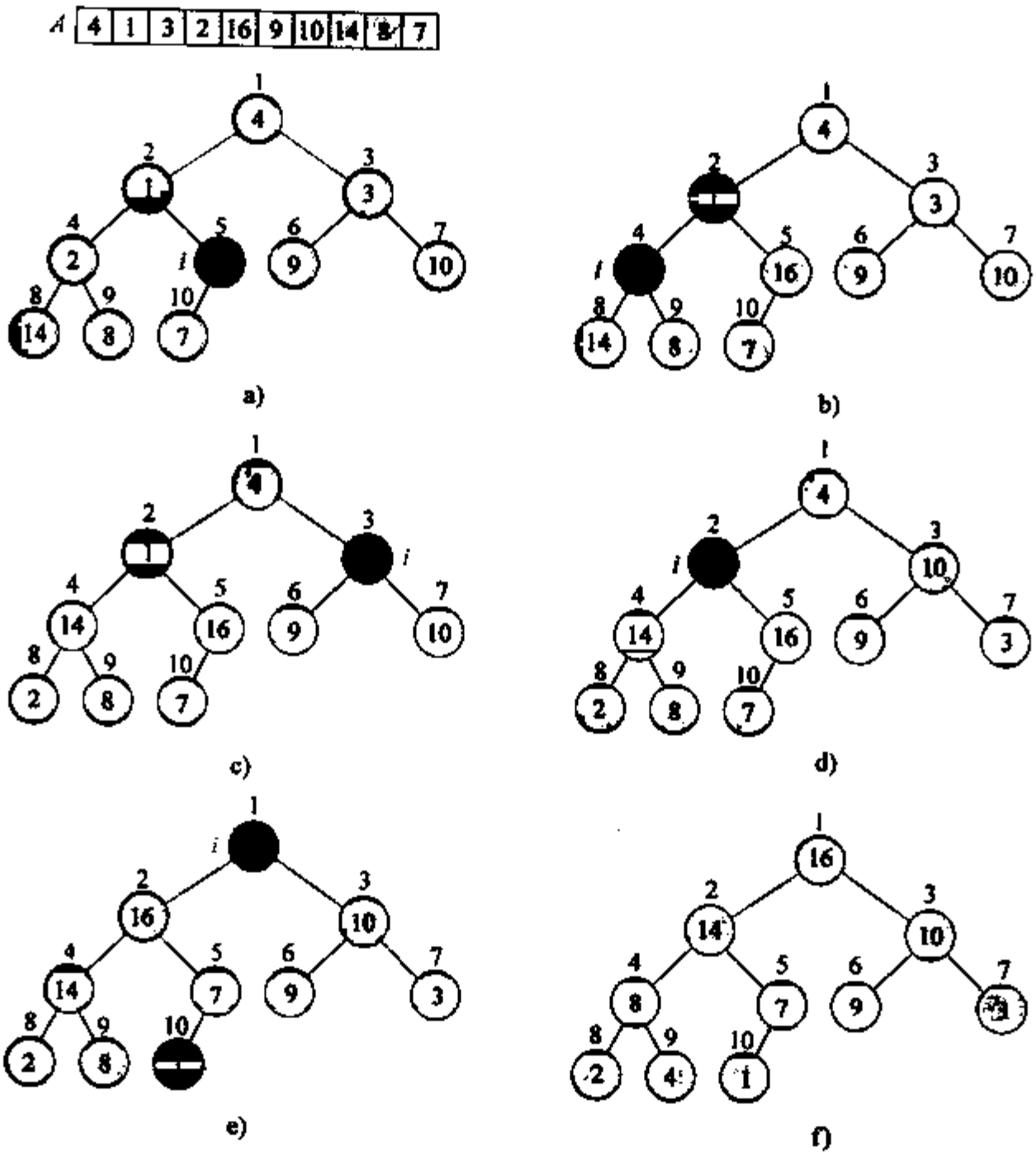


图 6-3 BUILD-MAX-HEAP 的执行过程．图中示出了在 BUILD-MAX-HEAP 的第 3 行调用 MAX-HEAPIFY 之前的数据结构。a)一个包含 10 个元素的输入数组 A 及其所表示的二叉树。图中示出了调用 MAX-HEAPIFY(A, i)之前循环下标 i 指向结点 5。b)结果所得的数据结构。循环下标 i 在下一轮执行中指向结点 4。c)～e)BUILD-MAX-HEAP 中 for 循环的后续执行过程。注意当对某结点调用 MAX-HEAPIFY 时，该结点的两棵子树都已是最大堆。f)BUILD-MAX-HEAP 执行完毕后的最大堆

终止：过程终止时，$i=0$。根据循环不变式，我们知道结点 1，2，…，n 中，每个都是最大堆的根，特别地，结点 1 就是一个最大堆的根。

我们可以这样来计算 BUILD-MAX-HEAP 运行时间的一个简单上界：每次调用 MAX-HEAPIFY 的时间为 $O(\lg n)$，共有 $O(n)$ 次调用，故运行时间是 $O(n \lg n)$。这个界尽管是对的，但从渐近意义上讲不够紧确。

实际上，我们可以得到一个更加紧确的界．这是因为，在树中不同高度的结点处运行 MAX-HEAPIFY 的时间不同，而且大部分结点的高度都较小。关于更紧确界的分析依赖于这样的性质：一个 n 元素堆的高度为 $\lfloor \lg n \rfloor$(见练习 6.1-2)，并且，在任意高度 h 上，至多有 $\lceil n/2^{h+1} \rceil$ 个结点(见练习 6.3-3)。

MAX-HEAPIFY 作用在高度为 h 的结点上的时间为 $O(h)$，我们可以将 BUILD-MAX-HEAP 的代价表达为：

133
~
134

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

上式右边的和式可以用公式(A.8)中的 $x=1/2$ 代入来计算，则

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

于是，BUILD-MAX-HEAP 的运行时间的界为

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

这说明可以在线性时间内，将一个无序数组建成一个最大堆。

我们可以用与 BUILD-MAX-HEAP 类似的方式，利用过程 BUILD-MIN-HEAP 来建立最小堆，只是在第 3 行要用调用 MIN-HEAPIFY(见练习 6.2-2)来替代调用 MAX-HEAPIFY。BUILD-MIN-HEAP 可以在线性时间内，将一个无序线性数组建成一个最小堆。

练习

6.3-1 模仿图 6-3，示出 BUILD-MAX-HEAP 作用于数组 $A=\langle 5, 3, 17, 10, 84, 19, 6, 22, 9\rangle$ 的过程。

6.3-2 在 BUILD-MAX-HEAP 的第 2 行代码中，为什么希望循环下标 i 从 $\lfloor length[A]/2 \rfloor$ 降到 1，而不是从 1 升到 $\lfloor length[A]/2 \rfloor$？

6.3-3 证明：在任一含 n 个元素的堆中，至多有 $\lceil n/2^{h+1} \rceil$ 个高度为 h 的结点。

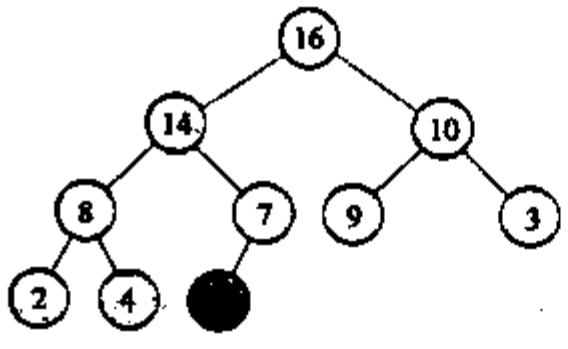
6.4　堆排序算法

开始时，堆排序算法先用 BUILD-MAX-HEAP 将输入数组 $A[1..n]$(此处 $n=length[A]$)构
135 造成一个最大堆。因为数组中最大元素在根 $A[1]$，则可以通过把它与 $A[n]$互换来达到最终正确的位置。现在，如果从堆中"去掉"结点 n(通过减小 $heap\text{-}size[A]$)，可以很容易地将$A[1..n-1]$建成最大堆。原来根的子女仍是最大堆，而新的根元素可能违背了最大堆性质。这时调用 MAX-HEAPIFY(A, 1)就可以保持这一性质，在 $A[1..(n-1)]$中构造出最大堆。堆排序算法不断重复这个过程，堆的大小由 $n-1$ 一直降到 2。(准确的循环不变式见练习 6.4-2)

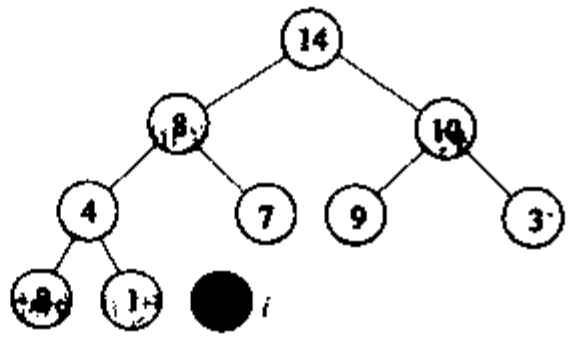
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3      do exchange A[1]↔A[i]
4         heap-size[A] ← heap-size[A]−1
5         MAX-HEAPIFY(A, 1)
```

图 6-4 给出了在初始最大堆建立后堆排序的一个例子。图中的每个最大堆与算法第 2～5 行的 for 循环的每一次迭代的开始对应。

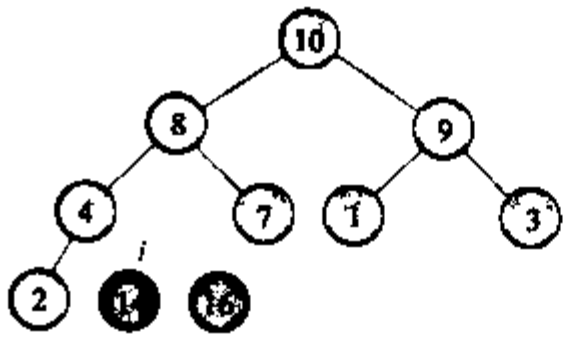
HEAPSORT 过程的时间代价为 $O(n \lg n)$。其中调用 BUILD-MAX-HEAP 的时间为 $O(n)$，$n-1$ 次HEAP-MAX-HEAPIFY 调用中每一次的时间代价为 $O(\lg n)$。



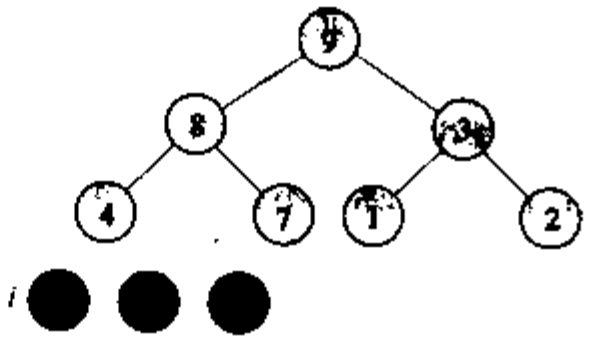
a)



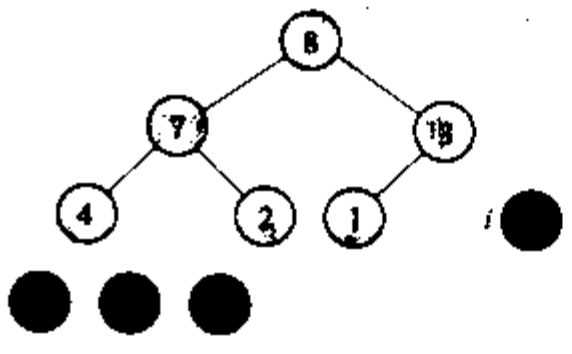
b)



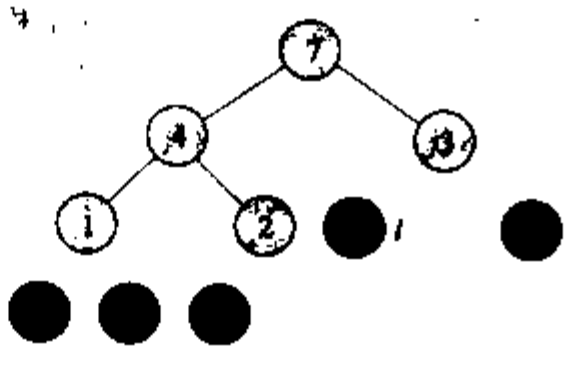
c)



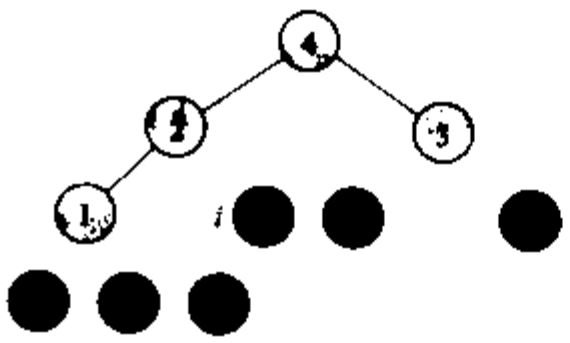
d)



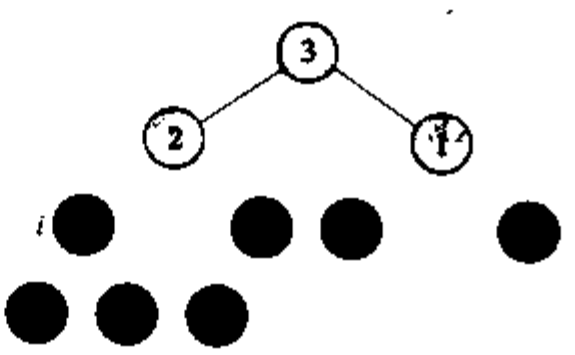
e)



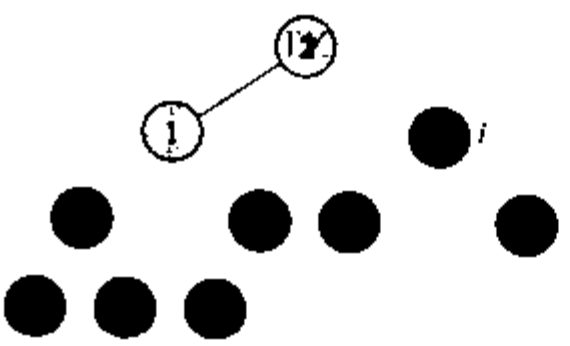
f)



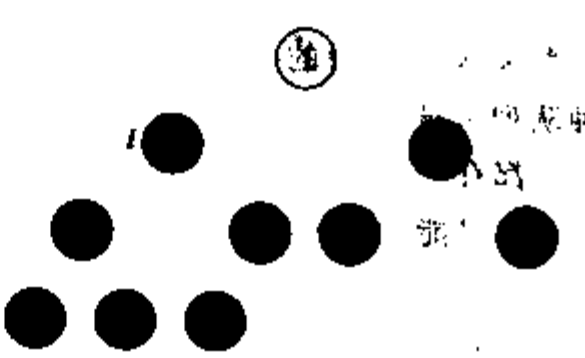
g)



h)



i)



j)

k)

图 6-4　HEAPSORT 的操作过程。a)用 BUILD-MAX-HEAP 构造所得的最大堆的数据结构。b)~j)每次在第 5 行调用 MAX-HEAPIFY 后的最大堆，同时示出 i 的值。仅浅阴影结点仍然留在堆中。k)结果的排序数组 A

练习

6.4-1　模仿图 6-4，说明 HEAPSORT 在数组 $A=\langle 5, 13, 2, 25, 7, 17, 20, 8, 4\rangle$ 上的操作过程。

6.4-2　讨论在使用如下循环不变式时，HEAPSORT 的正确性：

在每次 for 循环 2～5 行的迭代开始时，子数组 $A[1..i]$ 是一个包含了 $A[1..n]$ 中的 i 个最小元素的最大堆；而子数组 $A[i+1..n]$ 包含了已排序的 $A[1..n]$ 中的 $n-i$ 个最大元素。

136～137

6.4-3　对一个其所有 n 个元素已按递增序排列的数组 A，堆排序的运行时间是多少？若 A 的元素呈降序排列呢？

6.4-4　证明：堆排序的最坏情况运行时间为 $\Omega(n \lg n)$。

*6.4-5　证明：在所有元素都不相同时，堆排序算法的最佳运行时间是 $\Omega(n \lg n)$。

6.5　优先级队列

虽然堆排序算法是一个很漂亮的算法，但在实际中，快速排序(第 7 章将要介绍)的一个好的实现往往优于堆排序。尽管这样，堆数据结构还是有着很大的用处。在这一节中，我们要介绍堆的一个很常见的应用：作为高效的优先级队列(priority queue)。如堆一样，队列也有两种：最大优级先队列和最小优先级队列。这里将集中讨论基于最大堆实现的最大优先级队列。练习 6.5-3 将会要求读者写出最小优先级队列的程序。

优先级队列是一种用来维护由一组元素构成的集合 S 的数据结构，这一组元素中的每一个都有一个关键字 key。一个最大优先级队列支持以下操作：

INSERT(S, x)：把元素 x 插入集合 S。这一操作可写为 $S \leftarrow S \cup \{x\}$。

MAXIMUM(S)：返回 S 中具有最大关键字的元素。

EXTRACT-MAX(S)：去掉并返回 S 中的具有最大关键字的元素。

INCREASE-KEY(S, x, k)：将元素 x 的关键字的值增加到 k，这里 k 值不能小于 x 的原关键字的值。

最大优先级队列的一个应用是在一台分时计算机上进行作业调度。这种队列对要执行的各作业及它们之间的相对优先关系加以记录。当一个作业做完或被中断时，用 EXTRACT-MAX 操作从所有等待的作业中，选择出具有最高优先级的作业。在任何时候，一个新作业都可以用 INSERT 加入到队列中去。

最小优先级队列支持的操作包括 INSERT，MINIMUM，EXTRACT-MIN 和 DECREASE-KEY。这种队列可被用在基于事件驱动的模拟器中。在这种应用中，队列中的各项是要模拟的事件，每一个都有一个发生时间作为其关键字。事件模拟要按照各事件发生时间的顺序进行，因为模拟某一事件可能导致稍后对其他事件的模拟。模拟程序在每一步都使用 EXTRACT-MIN 来选择下一个模拟的事件。当一个新事件产生时，使用 INSERT 将其放入队列中。在第 23、24 章

138　中，我们将会看到最小优先级队列的其他用途，特别是 DECREASE-KEY 操作的使用。

优先级队列可以用堆来实现。在一个给定的，诸如作业调度或事件驱动的模拟应用中，优先级队列的元素对应着应用中的对象。通常，我们需要确定一个给定的队列中元素所对应的应用对象，反之亦然。当用堆来实现优先级队列时，需要在堆中的每个元素里存储对应的应用对象的柄(handle)。对象柄的准确表示到底怎样(如，一个指针，一个整型数等等)还取决于具体的应用。同样地，我们需要将堆中元素的柄存储到其对应的应用对象中。这里，对象柄用数组下标表示。因为在堆操作过程中，堆元素会改变在数组中的位置，所以，在具体实现中，为了能够重新定位

堆元素，我们需要更新应用对象中的数组下标。由于应用对象的访问细节与应用本身及其实现有很大的关系，在这里我们不再讨论，要注意的是在实践中，这些对象柄确实需要被正确地维护。

现在来讨论如何实现最大优先级队列的操作。程序 HEAP-MAXIMUM 用 $\Theta(1)$时间实现了 MAXIMUM 操作。

```
HEAP-MAXIMUM(A)
1  return A[1]
```

程序 HEAP-EXTRACT-MAX 实现了 EXTRACT-MAX 操作。它与 HEAPSORT 程序中的 for 循环体(第 3～5 行)很相似。

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A]<1
2     then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A]−1
6  MAX-HEAPIFY(A, 1)
7  return max
```

HEAP-EXTRACT-MAX 的运行时间为 $O(\lg n)$，因为它除了时间代价为 $O(\lg n)$的 MAX-HEAPIFY 外，只有很少的固定量的工作。

程序 HEAP-INCREASE-KEY 实现了 INCREASE-KEY 操作。在优先级队列中，关键字值需要增加的元素由对应数组的下标 i 来标识。该过程首先将元素 $A[i]$的关键字值更新为新的值。由于增大 $A[i]$的关键字可能会违反最大堆性质，所以过程中采用了类似于 2.1 节 INSERTION-SORT 的插入循环(第 5～7 行)的方式，在从本结点往根结点移动的路径上，为新增大的关键字寻找合适的位置。在移动的过程中，此元素不断地与其父母相比，如果此元素的关键字较大，则 139
交换它们的关键字且继续移动。当元素的关键字小于其父母时，此时最大堆性质成立，因此程序终止。(准确的循环不变式见练习 6.5-5)

```
HEAP-INCREASE-KEY(A, i, key)
1  if key<A[i]
2     then error "new key is smaller than current key"
3  A[i] ← key
4  while i>1 and A[PARENT(i)]<A[i]
5        do exchange A[i]↔A[PARENT(i)]
6           i ← PARENT(i)
```

图 6-5 是 HEAP-INCREASE-KEY 操作的一个示例。在 n 元堆上，HEAP-INCREASE-KEY 的运行时间为 $O(\lg n)$，因为在第 3 行被更新的结点到根结点的路径长度为 $O(\lg n)$。

下面的 MAX-HEAP-INSERT 实现了 INSERT 操作。它的输入是要插入到最大堆 A 中的新元素的关键字。这个程序首先加入一个关键字值为 $-\infty$的叶结点来扩展最大堆，然后调用 HEAP-INCREASE-KEY 来设置新结点的关键字的正确值，并保持最大堆性质。

```
MAX-HEAP-INSERT(A, key)
1  heap-size[A] ← heap-size[A]+1
2  A[heap-size[A]] ← −∞
3  HEAP-INCREASE-KEY(A, heap-size[A], key)
```

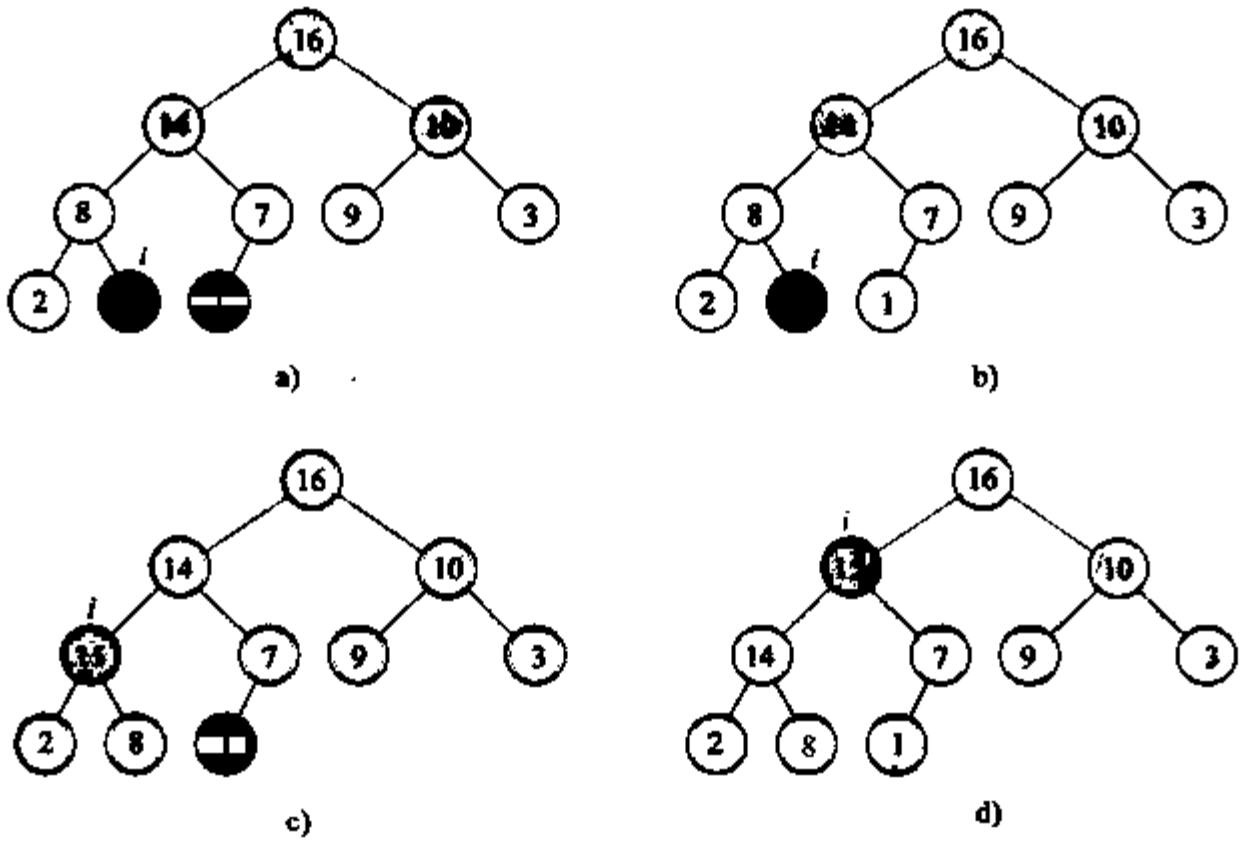


图 6-5　HEAP-INCREASE-KEY 的操作过程。a)图 6.4a 中的最大堆，其中下标为 i 的结点以深阴影表示。b)该结点将关键字的值增大到 15。c)经过第 4～6 行的 while 循环的一次迭代，该结点与其父结点交换了关键字，而且下标 i 上移到父亲结点。d)while 循环的又一次迭代后的最大堆。此时，$A[\text{PARENT}(i)] \geqslant A[i]$，最大堆性质成立，程序终止

在包含 n 个元素的堆上，MAX-HEAP-INSERT 的运行时间为 $O(\lg n)$。

总之，一个堆可以在 $O(\lg n)$时间内，支持大小为 n 的集合上的任意优先队列操作。

练习

6.5-1　描述对堆 $A=\langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$执行 HEAP-EXTRACT-MAX 操作的过程。

6.5-2　利用图 6-5 的堆作为 HEAP-INCREASE-KEY 的模型，描述在堆 $A=\langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$上执行 MAX-HEAP-INSERT$(A, 10)$操作的过程。

6.5-3　使用最小堆实现最小优先级队列，用伪代码写出 HEAP-MINIMUM，HEAP-EXTRACT-MIN、HEAP-DECREASE-KEY 和 MIN-HEAP-INSERT 过程。

140
~
141

6.5-4　为什么我们在 MAX-HEAP-INSERT 的第 2 行先将关键字的值设置为$-\infty$，而后又将此关键字增大到所要求的值呢？

6.5-5　使用以下的循环不变式来论证 HEAP-INCREASE-KEY 的正确性：

在第 4～6 行 while 循环的每次迭代之初，数组 $A[1..heap\text{-}size[A]]$满足最大堆性质，除了一个可能的例外，$A[i]$可能大于 $A[\text{PARENT}(i)]$。

6.5-6　说明如何使用优先级队列来实现一个先进先出队列，另请说明如何用优先级队列来实现栈。（队列和栈的定义见 10.1 节）

6.5-7　HEAP-DELETE(A, i)操作将结点 i 中的项从堆 A 中删去。对含 n 个元素的最大堆，请给出时间为 $O(\lg n)$的 HEAP-DELETE 的实现。

6.5-8　请给出一个时间为 $O(n \lg k)$、用来将 k 个已排序链表合并为一个排序链表的算法。此处 n 为所有输入链表中元素的总数。（提示：用一个最小堆来做 k 路合并）

思考题

6-1 用插入方法建堆

第6.3节中的BUILD-MAX-HEAP过程也可以通过反复调用MAX-HEAP-INSERT，将各元素插入堆中来实现。考虑如下实现：

```
BUILD-MAX-HEAP′(A)
1  heap-size[A] ← 1
2  for i ← 2 to length[A]
3      do MAX-HEAP-INSERT(A, A[i])
```

a)当输入数组相同时，过程BUILD-MAX-HEAP和BUILD-MAX-HEAP′产生的堆是否总是一样的？若读者认为是的，请给出证明；否则，请给出一个反例。

b)证明：在最坏情况下，BUILD-MAX HEAP′要用$\Theta(n \lg n)$时间来建成一个含n个元素的堆。

142

6-2 对d叉堆的分析

d叉堆有与二叉堆很类似，但(一个可能的例外是)其中的每个非叶结点有d个子女，而不是2个。

a)如何在一个数组中表示一个d叉堆？

b)含n个元素的d叉堆的高度是多少？(用n和d表示)

c)给出d叉最大堆的EXTRACT-MAX的一个有效实现，并用d和n表示出它的运行时间。

d)给出d叉最大堆的INSERT的一个有效实现，并用d和n表示出它的运行时间。

e)给出INCREASE-KEY(A, i, k)的一个有效实现，该过程首先执行$A[i] \leftarrow \max(A[i], k)$，并相应地更新$d$叉最大堆的结构。请用$d$和$n$表示出它的运行时间。

6-3 Young氏矩阵

一个$m \times n$的Young氏矩阵(Young tableau)是一个$m \times n$的矩阵，其中每一行的数据都从左到右排序，每一列的数据都从上到下排序。Young氏矩阵中可能会有一些∞数据项，表示不存在的元素。所以，Young氏矩阵可以用来存放$r \leqslant mn$个有限的数。

a)画一个包含元素{9, 16, 3, 2, 4, 8, 5, 14, 12}的4×4的Young氏矩阵。

b)讨论一个$m \times n$的Young氏矩阵，如果$Y[1, 1] = \infty$，则Y为空；如果$Y[m, n] < \infty$，则Y是满的(包含$m \times n$个元素)。

c)给出一个在非空$m \times n$的Young氏矩阵上实现EXTRACT-MIN的算法，使其运行时间为$O(m+n)$。你的算法应该使用一个递归子过程，它通过递归地解决$(m-1) \times n$或$m \times (n-1)$子问题来解决$m \times n$的问题。(提示：考虑一下MAX-HEAPIFY。)定义$T(p)$为EXTRACT-MIN在任何$m \times n$ Young氏矩阵上的最大运行时间，其中$p = m+n$。给出表达$T(p)$的、界为$O(m+n)$的递归式，并解该递归式。

d)说明如何在$O(m+n)$时间内，将一个新元素插入到一个未满的$m \times n$ Young氏矩阵中。

e)在不用其他排序算法帮助的情况下，说明利用$n \times n$ Young氏矩阵对n^2个数排序的运行时间为$O(n^3)$。

143

f)给出一个运行时间为$O(m+n)$的算法，来决定一个给定的数是否存在于一个给定的$m \times n$ Young氏矩阵内。

本章注记

堆排序算法是由 Williams[316]所发明的，他同时描述了如何使用堆来实现一个优先级队列。BUILD-MAX-HEAP 程序是由 Floyd[90]提出的。

在第 16、23、24 章中，我们利用最小堆实现了最小优先级队列。在第 19、20 章中我们还给出了一个实现，在该实现中，某些操作的运行时间界得到了改善。

优先级队列在整型数据上更快的实现是有可能的。van Emde Boas[301]提出了一种数据结构，它支持 MINIMUM，MAXIMUM，INSERT，DELETE，SEARCH，EXTRACT-MIN，EXTRACT-MAX，PRE-DECESSOR，SUCCESSOR 操作。这些操作最坏情况运行时间为 $O(\lg\lg C)$，要求关键字值的空间为集合$\{1, 2, \cdots, C\}$。如果数据是 b 位整型数，而且计算机存储器是由可寻址的 b 位字所构成的，Fredman 和 Willard[99]展示了如何实现 $O(1)$时间的 MINIMUM 和 $O(\sqrt{\lg n})$时间的 EXTRACT-MIN 操作。Thorup[299]将 $O(\sqrt{\lg n})$的界提高到 $O((\lg\lg n)^2)$，它的提高是以超过线性存储空间为代价的，但是可用随机散列方法在线性空间中实现。

当 EXTRACT-MIN 操作序列是单调的，也就是说，连续的 EXTRACT-MIN 操作的返回值会随着时间单调增大时，即出现了优先级队列的一种特别重要的情况。这一情况在若干重要的应用中都会出现，如第 24 章讨论的 Dijkstra 单源最短路径算法和离散事件模拟。在 Dijkstra 算法中，DECREASE-KEY 操作实现的高效性至关重要。在单调的例子中，对于数据是 1，2，…，C 范围内的整数，Ahuja，Melhorn，Orlin 和 Tarjan[8]利用叫作基数堆(radix heap)的数据结构，实现了 $O(\lg C)$平摊时间(amortized time)的 EXTRACT-MIN 和 INSERT(平摊分析的更多内容请参见第 17 章)，以及 $O(1)$运行时间的 DECREASE-KEY。同时使用斐波那契堆(见第 20 章)以及基数堆，界 $O(\lg C)$可以被提高到$O(\sqrt{\lg C})$。后来，这个界被 Cherkassky，Goldberg 和 Silverstein[58]进一步提高到 $O(\lg^{1/3+\varepsilon} C)$的期望时间。他们将 Denardo 与 Fox[72]提出的多层桶结构(multilevel bucketing structure)和上面提到的 Thorup 堆结合了起来。Raman[256]又进一步地将结果改进到 $O(\min(\lg^{1/4+\varepsilon} C, \lg^{1/3+\varepsilon} n))$，其中任意固定值 $\varepsilon>0$。这些结论的更多细节讨论可以在 Raman
144 [256]和 Thorup[299]的论文中找到。

第7章 快速排序

快速排序是一种排序算法，对包含 n 个数的输入数组，最坏情况运行时间为 $\Theta(n^2)$。虽然这个最坏情况运行时间比较差，但快速排序通常是用于排序的最佳的实用选择，这是因为其平均性能相当好：期望的运行时间为 $\Theta(n\lg n)$，且 $\Theta(n\lg n)$记号中隐含的常数因子很小。另外，它还能够进行就地排序，在虚存环境中也能很好地工作。

7.1 节介绍快速排序算法及它用来划分数组的一个重要子程序。这个算法的运行情况比较复杂，在 7.2 节中，我们先对其性能进行直观的讨论，稍后再给出准确的分析。7.3 节介绍快速排序使用随机抽样的变形。这一版本的平均情况运行时间较好，也没有什么特殊的输入会导致最坏情况运行状态。这两个随机化算法将在 7.4 节中分析，其最坏情况运行时间为 $\Theta(n^2)$，平均情况运行时间为 $O(n\lg n)$。

7.1 快速排序的描述

像合并排序一样，快速排序也是基于 2.3.1 节介绍的分治模式的。下面是对一个典型子数组 $A[p..r]$排序的分治过程的三个步骤：

分解：数组 $A[p..r]$被划分成两个(可能空)子数组 $A[p..q-1]$和 $A[q+1..r]$，使得 $A[p..q-1]$中的每个元素都小于等于 $A(q)$，而且，小于等于 $A[q+1..r]$中的元素。下标 q 也在这个划分过程中进行计算。

解决：通过递归调用快速排序，对子数组 $A[p..q-1]$和 $A[q+1..r]$排序。

合并：因为两个子数组是就地排序的，将它们的合并不需要操作：整个数组 $A[p..r]$已排序。

下面的过程实现快速排序：

```
QUICKSORT(A, p, r)
1  if p<r
2    then q ← PARTITION(A, p, r)
3         QUICKSORT(A, p, q−1)
4         QUICKSORT(A, q+1, r)
```

为排序一个完整的数组 A，最初的调用是 QUICKSORT(A, 1, $length[A]$)。

数组划分

快速排序算法的关键是 PARTITION 过程，它对子数组 $A[p..r]$进行就地重排：

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p−1
3  for j ← p to r−1
4       do if A[j]≤x
5             then i ← i+1
6                  exchange A[i]↔A[j]
7  exchange A[i+1]↔A[r]
8  return i+1
```

图 7-1 展示 PARTITION 在一个包含 8 个元素的数组上的操作过程。PARTITION 总是选择一个$x=A[r]$作为主元(pivot element)，并围绕它来划分子数组 $A[p..r]$。随着该过程的执行，数组被划分成四个(可能有空的)区域。在第 3～6 行中 for 循环每一轮迭代的开始，每一个区域都满足特定的性质，这些性质可以作为循环不变式表述如下：

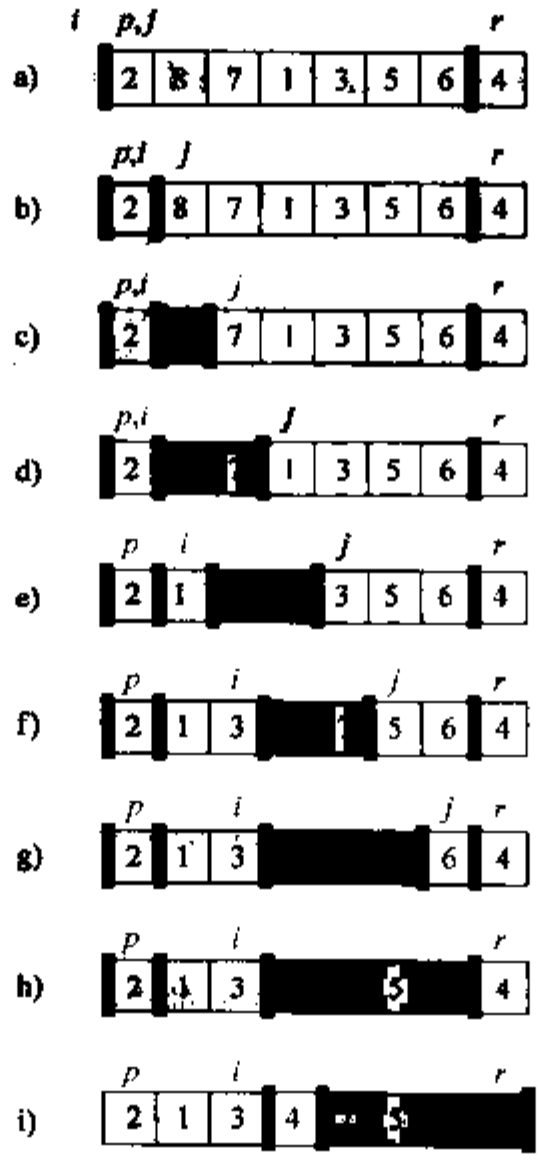


图 7-1　PARTITION 在一个样例数组上的运行过程。浅阴影部分的数组元素都在划分的第一部分，其值都不大于 x。深阴影的元素在划分第二部分，其值都大于 x。无阴影的元素尚未被放入前两个部分中的任何一个，白色元素为主元。a)初始的数组和变量设置。数组元素均未被放入前两个部分中的任何一个；b)数值 2 被"与它自身交换"，并被放入了元素值较小的那个部分；c)～d)数值 8 和 7 被加到具有较大值的那个部分中；e)数值 1 和 8 交换，较小的部分增加；f)数值 3 和 8 交换，较小的部分增加；g)～h)较大的部分增加，包含 5 和 6，循环结束；i)在第 7～8 行中，主元被交换，这样就位于两个部分之间

在第 3～6 行中循环的每一轮迭代的开始，对于任何数组下标 k，有：

1)如果 $p \leqslant k \leqslant i$，则 $A[k] \leqslant x$。

2)如果 $i+1 \leqslant k \leqslant j-1$，则 $A[k] > x$。

3)如果 $k=r$，则 $A[k]=x$。

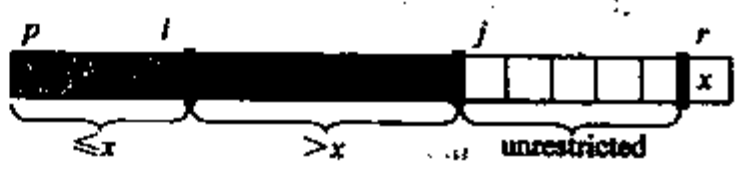


图 7-2　过程 PARTITION 作用于子数组 $a[p..r]$后得到的四个区域。$A[p..i]$中的各个值都小于或等于 x，$A[i+1..j-1]$中的值都大于 x，$A[r]=x$，$A[j..r-1]$中的值可以取任何值

图 7-2 总结这一结构。j 和 $r-1$ 之间的下标在三种情况中都没有涉及，对应元素的值与主元 x 没有特别关系。

我们需要证明这个循环不变式在第一轮迭代之前是成立的，循环的每一轮迭代都能使之

保持成立，并且，该循环不变式提供一个有用的属性，用以证明循环结束时的正确性。

初始化：在循环的第一轮迭代开始之前，有 $i=p-1$ 和 $j=p$。在 p 和 i 之间没有值，在 $i+1$ 和 $j-1$ 之间也没有值，因此，循环不变式的头两个条件显然满足。第1行中的赋值操作满足第三个条件。

保持：如图7-3中所示，要考虑两种情况，具体取决于第4行中测试的结果。图7-3a显示当 $A[j]>x$ 时所做的处理：循环中的唯一操作是 j 增加1。在 j 增加1后，条件2对 $A[j-1]$ 成立，且所有其他项保持不变。图7-3b显示当 $A[j]\leqslant x$ 时所做的处理：将 i 增加1，交换 $A[i]$ 和 $A[j]$，再将 j 增加1。因为进行了交换，现在有 $A[i]\leqslant x$，因而条件1满足。类似地，还有 $A[j-1]>x$，因为根据循环不变式，被交换进 $A[j-1]$ 的项目是大于 x 的。

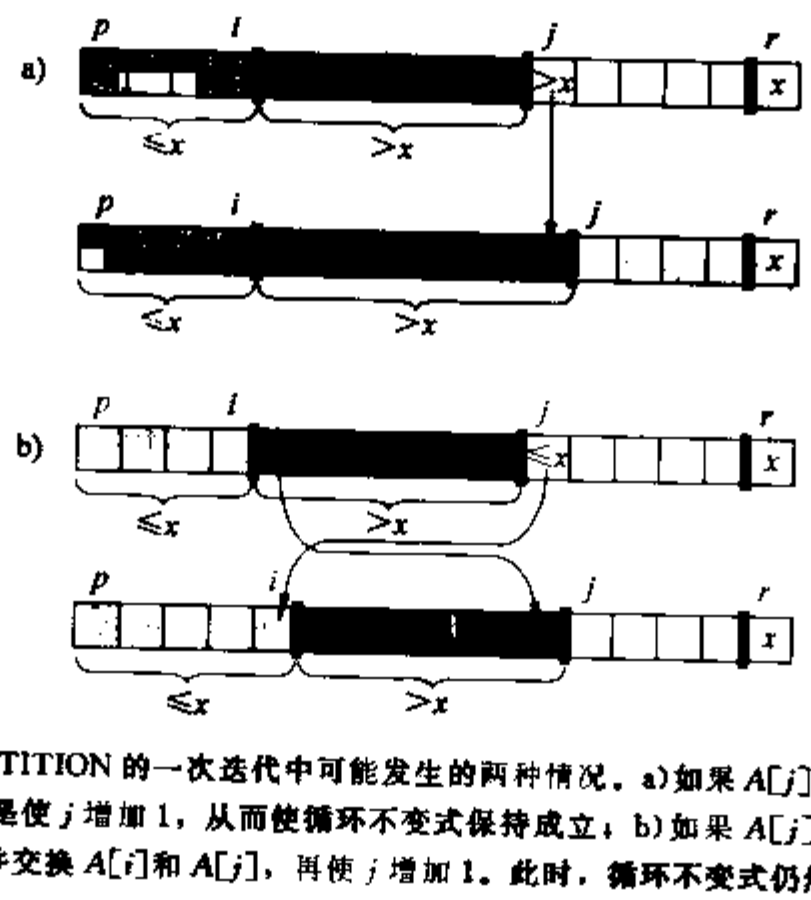


图7-3 过程PARTITION的一次迭代中可能发生的两种情况。a)如果 $A[j]>x$，需要做的唯一操作就是使 j 增加1，从而使循环不变式保持成立；b)如果 $A[j]\leqslant x$，则将下标 i 增加1，并交换 $A[i]$ 和 $A[j]$，再使 j 增加1。此时，循环不变式仍然保持成立

终止：当终止时，$j=r$。于是，数组中的每个元素都在循环不变式所描述的三个集合的某一个之中，亦即，我们已将数组中的所有元素划分成了三个集合：一个集合中包含了小于等于 x 的元素，第二个集合中包含了大于 x 的元素，还有一个只包含了 x 的集合。

在PARTITION过程的最后两行中，通过将主元与最左的、大于 x 的元素进行交换，就将它移到了它在数组中间的位置上。此时，PARTITION的输出满足分解步骤所做规定的要求。

PARTITION在子数组 $A[p..r]$ 上的运行时间为 $\Theta(n)$，其中 $n=r-p+1$（见练习7.1-3）。

练习

7.1-1 仿照图7-1，说明PARTITION过程作用于输入数组 $A=\langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\rangle$ 上的过程。

7.1-2 当数组 $A[p..r]$ 中的元素均相同时，PARTITION返回的 q 值是什么？修改PARTITION，使得当数组 $A[p..r]$ 中所有元素的值相同时，$q=(p+r)/2$。

7.1-3 简要地证明在大小为 n 的子数组上，PARTITION的运行时间为 $\Theta(n)$。

7.1-4 应如何修改QUICKSORT，才能使其按非增序进行排序？

7.2　快速排序的性能

快速排序的运行时间与划分是否对称有关，而后者又与选择了哪一个元素来进行划分有关。如果划分是对称的，那么本算法从渐近意义上来讲，就与合并算法一样快；如果划分是不对称的，那么本算法渐近上就和插入算法一样慢。在本节中，我们要讨论当划分为对称或非对称时快速排序的性能。

最坏情况划分

146 ~ 149

快速排序的最坏情况划分行为发生在划分过程产生的两个区域分别包含 $n-1$ 个元素和 1 个 0 元素的时候(证明见 7.4.1 节)。假设算法的每一次递归调用中都出现了这种不对称划分。划分的时间代价为 $\Theta(n)$。因为对一个大小为 0 的数组进行递归调用后，返回 $T(0)=\Theta(1)$，故算法的运行时间可以递归地表示为：

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

从直观上来看，如果将每一层递归的代价加起来，就可以得到一个算术级数(等式(A.2))，其和值的量级为 $\Theta(n^2)$。利用代换法，可以比较直接地证明递归式 $T(n)=T(n-1)+\Theta(n)$ 的解为 $T(n)=\Theta(n^2)$(见练习 7.2-1)。

因此，如果在算法的每一层递归上，划分都是最大程度不对称的，那么算法的运行时间就是 $\Theta(n^2)$。亦即，快速排序算法的最坏情况运行时间并不比插入排序的更好。此外，当输入数组已经完全排好序时，快速排序的运行时间为 $\Theta(n^2)$，而在同样情况下，插入排序的运行时间为 $O(n)$。

最佳情况划分

在 PARTITION 可能做的最平衡的划分中，得到的两个子问题的大小都不可能大于 $n/2$，因为其中一个子问题的大小为 $\lfloor n/2 \rfloor$，另一个子问题的大小为 $\lceil n/2 \rceil - 1$。在这种情况下，快速排序运行的速度要快得多。这时，表达其运行时间的递归式为

$$T(n) \leqslant 2T(n/2) + \Theta(n)$$

根据主定理(即定理 4.1)的情况 2，该递归式的解为 $T(n)=O(n\lg n)$。由于在每一层递归上，划分的两边都是对称的，因此，从渐近意义上来看，算法运行得就更快了。

平衡的划分

快速排序的平均情况运行时间与其最佳情况运行时间很接近，而不是非常接近于其最坏情况运行时间，这一点可以从 7.4 节的分析中看到。要理解这一点，就要理解划分的平衡性是如何在刻划运行时间的递归式中反映出来的。

例如，假设划分过程总是产生 9∶1 的划分，乍一看这种划分很不平衡，这时，快速排序运行时间的递归式为

$$T(n) \leqslant T(9n/10) + T(n/10) + cn$$

此处，我们显式地写出了 $\Theta(n)$ 项中所隐含的常数 c。图 7-4 示出了与这一递归式对应的递归树。请注意该树每一层的代价都是 cn，直到在深度 $\log_{10} n=\Theta(\lg n)$ 处达到边界条件时为止，在此之下各层的代价至多为 cn。递归于深度 $\log_{10/9} n=\Theta(\lg n)$ 处终止。这样，快速排序的总代价为

150 $O(n\lg n)$。在递归的每一层上是按照 9∶1 的比例进行划分的，直观上看上去好像应该是相当不平衡的，但在这种情况下，快速排序的运行时间为 $O(n\lg n)$，从渐近意义上来看，这与划分是在正中间进行的效果是一样的。事实上按 99∶1 划分运行时间为 $O(n\lg n)$。其原因在于，任何一种按常数比例进行的划分都会产生深度为 $\Theta(\lg n)$ 的递归树，其中每一层的代价为 $O(n)$，因而，每当按照常数比例进行划分时，总的运行时间都是 $O(n\lg n)$。

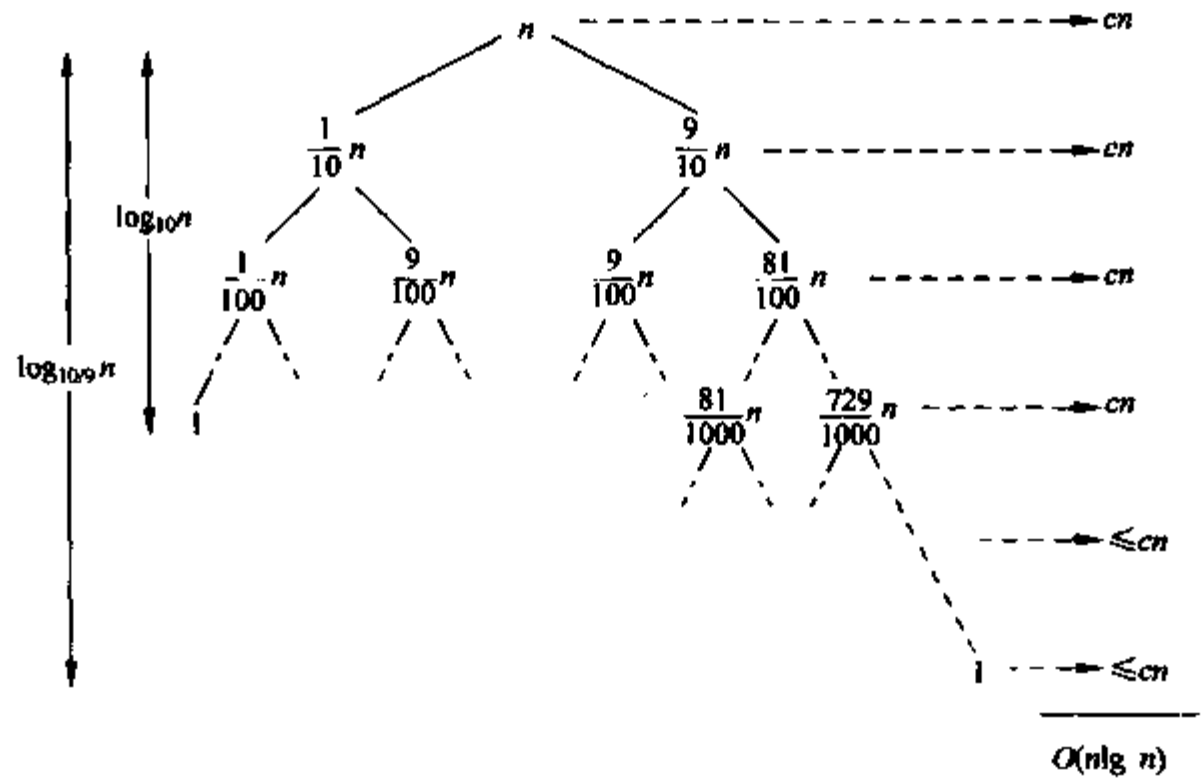


图 7-4　QUICKSORT 的一棵递归树，其中 PARTITION 总是产生 9∶1 的划分，总的运行时间为 $O(n\lg n)$。各结点中示出了子问题的规模，每一层的代价在右边显示。每一层的代价包含了 $\Theta(n)$ 项中所隐含的常数 c

关于平均情况的直觉考虑

要想对快速排序的平均情况有个较为清楚的认识，我们就要对各种输入的出现频率做个假设。快速排序的行为是由作为输入给出的数组中，各元素值的相对顺序来决定的，而不是由数组中特定的值所决定的。如我们在第 5.2 节中对雇佣问题(hiring problem)所做的概率分析那样，此处我们假设输入数据的所有排列都是等可能的。

当对一个随机的输入数组应用快速排序时，要想如我们在非形式化分析中所假设的那样，在每一层上都有同样的划分是不太可能的。我们所能期望的是某些划分比较对称，另一些则很 151
不对称。例如，练习 7.2-6 就要求读者说明 PARTITION 所产生的划分 80%以上都比 9∶1 更对称，而另 20%则比 9∶1 差。

在平均情况下，PARTITION 所产生的划分中既有“好的”，又有“差的”。这时，与 PARTITION 执行过程对应的递归树中，好、差划分是随机地分布在树的各层上的。为与我们的直觉相一致，假设好、差划分交替出现在树的各层上，且好的划分是最佳情况划分，而差的划分是最坏情况下的划分，图 7-5a 示出了递归树的连续两层上的划分情况。在根结点处，划分的代价为 n，划分出来的两个子数组的大小为 $n-1$ 和 0，即最坏情况。在根的下一层处，大小为 $n-1$ 的子数组按最佳情况划分成大小各为 $(n-1)/2-1$ 和 $(n-1)/2$ 的两个子数组。这儿我们假设大小为 0 的子数组的边界条件代价为 1。

在一个差的划分后接一个好的划分后，产生出三个子数组，大小各为 0、$(n-1)/2-1$ 和 $(n-1)/2$，总的划分代价为 $\Theta(n)+\Theta(n-1)=\Theta(n)$，该代价并不比图 7-5b 中的要差。在该图中，一层划分就产生出大小为 $(n-1)/2$ 的两个子数组，代价为 $\Theta(n)$。但是，后者是对称的！从直觉上看，差的划分的代价 $\Theta(n-1)$ 可以被吸收到好的划分的代价 $\Theta(n)$ 中去，结果是一个好的划分。这样，当好、差划分交替分布在各层中时，快速排序的运行时间就如全是好的划分时一样，仍然是 $O(n\lg n)$，但是，O 记号中隐含的常数因子要略大一些。关于平均情况的严格分析将在 7.4.2 中给出。

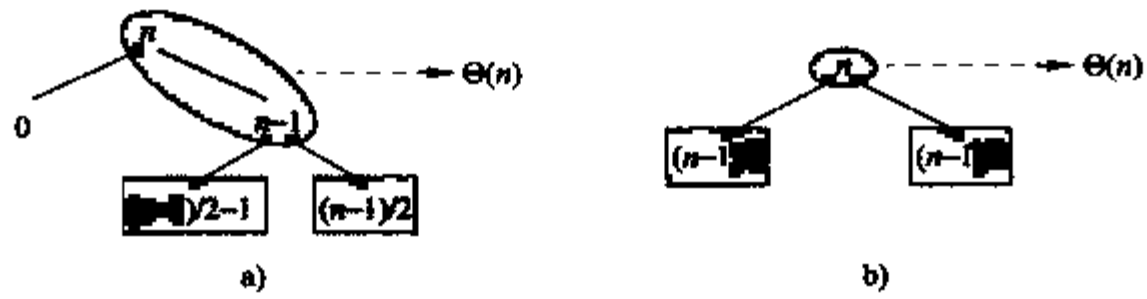


图 7-5　a)快速排序的递归树中的两层。树根处划分的代价为 n，且得到是"很差的"划分：两个子数组的大小分别为 0 和 $n-1$。对大小为 $n-1$ 的子数组的划分代价为 $n-1$，产生一个"较好的"划分：两个子数组的大小分别为 $(n-1)/2-1$ 和 $(n-1)/2$；b)一棵非常平衡的递归树中的一层。在它的两个部分中，子问题的划分代价以椭圆形阴影示出，为 $\Theta(n)$。但是，a)图中矩形阴影示出的、待解决的子问题并不大于 b)图中对应的、待解决的子问题

152

练习

7.2-1　利用代换法证明：递归式 $T(n)=T(n-1)+\Theta(n)$ 的解为 $T(n)=\Theta(n^2)$，如第 7.2 节开头提到的那样。

7.2-2　当数组 A 的所有元素都具有相同值时，QUICKSORT 的运行时间是什么？

7.2-3　证明：当数组 A 包含不同的元素、且按降序排序时，QUICKSORT 的运行时间为 $\Theta(n^2)$。

7.2-4　银行经常按照交易时间，来记录有关某一账户的交易情况，但是，很多人却喜欢按照票据号来收到其银行对账单。这是因为，一般人通常都是按照支票的顺序编号来开出支票的，而商人们也通常都是根据支票编号的顺序来送货的。因此，如何将按交易时间排序转换成按支票编号排序，就成为了一个对几乎已排好序的输入进行排序的问题。证明在这个问题上，过程 INSERTION-SORT 的性能往往要优于过程 QUICKSORT。

7.2-5　假设快速排序的每一层，所做划分的比例都是 $1-\alpha:\alpha$，其中 $0<\alpha\leqslant 1/2$ 是个常数。证明：在对应的递归树中，叶子结点的最小深度大约是 $-\lg n/\lg\alpha$，最大深度大约是 $-\lg n/\lg(1-\alpha)$。（无需考虑整数的取整舍入问题）

***7.2-6**　证明：对于任何常数 $0<\alpha\leqslant 1/2$，在一个随机输入数组上，过程 PARTITION 产生比 $1-\alpha:\alpha$ 更对称的划分的概率约为 $1-2\alpha$。

7.3　快速排序的随机化版本

在探讨快速排序的平均性态过程中，我们已假定输入数据的所有排列都是等可能的，但在工程中，这个假设就不会总是成立(见练习 7.2-4)。正如我们在第 5.3 节中所看到的，有时，我们可以向一个算法中加入随机化的成分，以便对于所有输入，它均能获得较好的平均情况性能。很多人都认为，快速排序的随机化版本是对足够大的输入的理想选择。

153

在 5.3 节中，我们通过显式地对输入进行排列而使算法随机化了。对快速排序也可以这么做，但是，如果采用一种不同的、称为随机取样(random sampling)的随机化技术的话，可以使得分析更加简单。在这种方法中，不是始终采用 $A[r]$作为主元，而是从子数组 $A[p..r]$中随机选择一个元素，即将 $A[r]$与从 $A[p..r]$中随机选出的一个元素交换。在这一修改中，我们是从 $p,\cdots,r$ 这一范围中随机取样的，这么做确保了在子数组的 $r-p+1$ 个元素中，主元元素 $x=A[r]$等可能地取其中的任何一个。因为主元元素是随机选择的，我们期望在平均情况下，对输入数组的划分能够比较对称。

对 PARTITION 和 QUICKSORT 所做的改动比较小。在新的划分过程中，我们在真正进行

划分之前实现交换：

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  exchange A[r]↔A[i]
3  return PARTITION(A, p, r)
```

新的快速排序过程不再调用 PARTITION，而是调用 RANDOMIZED-PARTITION。

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p<r
2     then q ← RANDOMIZED-PARTITION(A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q−1)
4          RANDOMIZED-QUICKSORT(A, q+1, r)
```

我们将在下一节中分析这一算法。

练习

7.3-1 我们为什么要分析一个随机化算法的平均情况性能，而不是其最坏情况性能呢？

7.3-2 在过程 RANDOMIZED-QUICKSORT 的运行过程中，最坏情况下对随机数产生器 RANDOM 调用了多少次？最佳情况下调用了多少次？以 Θ 记号形式给出你的答案。 154

7.4 快速排序分析

7.2 节从直觉上对快速排序的最坏情况性态、它为何运行得较快等作了一些讨论。在本节中，我们要给出对快速排序性能的严格分析。先进行最坏情况分析，这对 QUICKSORT 和 RANDOMIZED-QUICKSORT 都一样。然后，再分析 RANDOMIZED-QUICKSORT 的平均情况性能。

7.4.1 最坏情况分析

在 7.2 节中我们看到，如果快速排序中每一层递归上所做的都是最坏情况划分，则运行时间为 $\Theta(n^2)$。从直觉上看，这就是最坏情况运行时间。下面来证明。

利用代换法（见 4.1 节），可以证明快速排序的运行时间为 $O(n^2)$。设 $T(n)$ 是过程 QUICKSORT 作用于规模为 n 的输入上的最坏情况时间，则有：

$$T(n) = \max_{0\leqslant q\leqslant n-1}(T(q)+T(n-q-1))+\Theta(n) \tag{7.1}$$

其中参数 q 由 0 变到 $n-1$，这是因为过程 PARTITION 产生两个子问题，总的大小为 $n-1$。我们猜测 $T(n)\leqslant cn^2$ 成立，c 为某个常数。将此式代入递归式(7.1)，得：

$$T(n)\leqslant \max_{0\leqslant q\leqslant n-1}(cq^2+c(n-q-1)^2)+\Theta(n)$$

$$=c\cdot\max_{0\leqslant q\leqslant n-1}(q^2+(n-q-1)^2)+\Theta(n)$$

表达式 $q^2+(n-q-1)^2$ 在参数的取值区间 $0\leqslant q\leqslant n-1$ 的某个端点上取得最大值，因为该式关于 q 的二阶导数是正的（见练习 7.4-3）。这样，就有界 $\max_{0\leqslant q\leqslant n-1}(q^2+(n-q-1)^2)\leqslant(n-1)^2=n^2-2n+1$。对 $T(n)$ 就有：

$$T(n)\leqslant cn^2-c(2n-1)+\Theta(n)\leqslant cn^2$$

因为我们可以选择足够大的常数 c，使得项 $c(2n-1)$ 能支配 $\Theta(n)$。这样，$T(n)=O(n^2)$ 在 7.2 节中，我们看到了一个快速排序的运行时间为 $\Omega(n^2)$ 的特定情况：当划分为非对称的时候。此外，练习 7.4-1 也要求读者证明递归式(7.1)有解 $T(n)=\Omega(n^2)$。于是，快速排序的（最坏情

155 况)运行时间为 $\Theta(n^2)$。

7.4.2　期望的运行时间

我们已经从直觉上说明了为什么 RANDOMIZED-QUICKSORT 的平均情况运行时间为 $O(n\lg n)$：如果在递归的每一层上，RANDOMIZED-PARTITION 所做出的划分使任意固定量的元素偏向划分的某一边，则算法的递归树深度为 $\Theta(\lg n)$，且在每一层上所做的工作量都为 $O(n)$。在这些层次之间，即使我们增加了新的、具有最不对称的划分的层次，总的运行时间仍然是 $O(n\lg n)$。要准确地分析 RANDOMIZED-QUICKSORT 的期望运行时间，就要首先理解划分过程是如何进行的。然后，在此基础之上，导出有关期望的运行时间的一个 $O(n\lg n)$界。有了关于期望运行时间的这个上界，再加上我们已在 7.2 节中见到过的 $\Theta(n\lg n)$最佳情况界，就能得出 $\Theta(n\lg n)$这一期望运行时间了。

运行时间和比较

QUICKSORT 的运行时间是由花在过程 PARTITION 上的时间所决定的。每当 PARTITION 过程被调用时，就要选出一个主元元素。后续对 QUICKSORT 和 PARTITION 的各次递归调用中，都不会包含该元素。于是，在快速排序算法的整个执行过程中，至多只可能调用 PARTITION 过程 n 次。调用一次 PARTITION 的时间为 $O(1)$再加上一段时间，这段时间与第 3～6 行中 for 循环中迭代的次数成正比。这一 for 循环的每一轮迭代都要在第 4 行中进行一次比较，即将主元元素与数组 A 中另一个元素进行比较。因此，如果我们可以数清楚第 4 行执行的总次数，就能够给出在 QUICKSORT 的执行过程中，for 循环所花时间的界了。

引理 7.1　设当 QUICKSORT 在一个包含 n 个元素的数组上运行时，PARTITION 在第 4 行中所做比较的次数为 X。那么，QUICKSORT 的运行时间为 $O(n+X)$。

证明：根据上面的讨论，对 PARTITION 的调用共有 n 次。每一次调用都需做固定量的工作，再执行若干次 for 循环。在 for 循环的每一轮迭代中，都要执行第 4 行。 ■

我们的目标是计算出 X，即在对 PARTITION 的所有调用中，所执行的总的比较次数。我们并不打算分析在每一次 PARTITION 调用中做了多少次比较，而是希望导出关于总的比较次数的一个界。为了达到这一目的，我们必须了解算法在何时要对数组中的两个元素进行比较，何时
156 不进行比较。为了便于分析，我们将数组 A 的各个元素重新命名为 z_1，z_2，…，z_n，其中 z_i 是数组 A 中第 i 个最小的元素。此外，我们还定义 $Z_{ij}=\{z_i, z_{i+1}, \cdots, z_j\}$为 z_i 与 z_j 之间(包含这两个元素)的元素集合。

那么，算法何时会比较 z_i 与 z_j 呢？为了回答这个问题，我们首先观察到每一对元素至多比较一次。这是为什么呢？因为各个元素仅与主元元素进行比较，并且，在某一次 PARTITION 调用结束之后，该次调用中所用到的主元元素就再也不会与任何其他元素进行比较了。

我们的分析要用到指示器随机变量(见 5.2 节)。我们定义

$$X_{ij} = \mathrm{I}\{z_i \text{ 与 } z_j \text{ 进行比较}\}$$

我们要考虑的是在算法的执行过程中，是否有任何的比较发生，而不是在循环的一次迭代或对 PARTITION 的一次调用中是否有比较发生。因为每一对元素至多被比较一次，因而，我们可以很容易地刻划算法所执行的总的比较次数：

$$X = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}$$

对上式两边取期望值，再利用期望值的线性特性和引理 5.1，可以得到：

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\} \qquad (7.2)$$

在上式中，$\Pr\{z_i$ 与 z_j 进行比较$\}$还有待于进一步计算。

考虑一下两个元素何时不进行比较也是有好处的。考虑快速排序的一个输入，它由数字 1 到 10 所构成(顺序可以是任意的)，并假设第一个主元元素是 7。那么，对 PARTITION 的第一次调用就将这些输入数字划分成两个集合：{1，2，3，4，5，6}和{8，9，10}。在划分的过程中，主元元素 7 要与所有其他元素进行比较，但是，第一个集合中任何一个元素(例如 2)都没有(后面也不会)与集合 2 中的任何元素(例如 9)进行比较。

一般而言，一旦一个满足 $z_i<x<z_j$ 的主元 x 被选择后，我们知道，z_i 与 z_j 以后是再也不可能进行比较了。另一方面，如果 z_i 在 Z_{ij} 中的所有其他元素之前被选为主元，那么 z_i 与就将与 Z_{ij} 中的、除了它自己以外的所有元素进行比较。类似地，如果 z_j 在 Z_{ij} 中其他元素之前被选为主元，那么 z_j 将与 Z_{ij} 中除自身以外的每项进行比较。在我们的例子中，7 和 9 要进行比较，因 157
为 7 是 $Z_{7,9}$ 中将被选为主元的第一个元素。2 和 9 则始终不会被放到一起进行比较，这是因为从 $Z_{2,9}$ 中选出的头一个主元元素为 7。由此我们知道，z_i 会与 z_j 进行比较，当且仅当 Z_{ij} 中将被选作主元的第一个元素是 z_i 或 z_j。

我们现在来计算这一事件发生的概率。在 Z_{ij} 中的某一元素被选为主元之前，集合 Z_{ij} 整个都是在同一划分中的。于是，Z_{ij} 中的任何元素都会等可能地被首先选为主元。因为集合 Z_{ij} 中共有 $j-i+1$ 个元素，所以，任何元素被首先选为主元的概率是 $1/(j-i+1)$。于是，我们有：

$$\begin{aligned}\Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\} &= \Pr\{z_i \text{ 或 } z_j \text{ 是从 } Z_{ij} \text{ 中首先选出的主元}\}\\ &= \Pr\{z_i \text{ 是从 } Z_{ij} \text{ 中首先选出的主元}\}\\ &\quad + \Pr\{z_j \text{ 是从 } Z_{ij} \text{ 中首先选出的主元}\}\\ &= \frac{1}{j-i+1}+\frac{1}{j-i+1}=\frac{2}{j-i+1} \qquad (7.3)\end{aligned}$$

上式中的第二行成立是因为其中涉及的两个事件是互斥的。将等式(7.2)和等式(7.3)综合起来，有：

$$\mathrm{E}[X]=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1}$$

在求这个和式时，可以将变量作个变换($k=j-i$)，并利用等式(A.7)中给出的有关调和级数的界：

$$\mathrm{E}[X]=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1}=\sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k+1}<\sum_{i=1}^{n-1}\sum_{k=1}^{n}\frac{2}{k}=\sum_{i=1}^{n-1}O(\lg n)=O(n\lg n) \qquad (7.4)$$

于是，我们可以得出结论，即利用 RANDOMIZED-PARTITION，快速排序算法期望的运行时间为 $O(n\lg n)$。 158

练习

7.4-1　证明：在递归式：

$$T(n)=\max_{0\le q\le n-1}(T(q)+T(n-q-1))+\Theta(n)$$

中，$T(n)=\Omega(n^2)$。

7.4-2　证明：快速排序的最佳情况运行时间为 $\Omega(n\lg n)$。

7.4-3　证明：在 $q=0$，1，…，$n-1$ 区间上，当 $q=0$ 或 $q=n-1$ 时，$q^2+(n-q-1)^2$ 取得最大值。

7.4-4　证明：RANDOMIZED-QUICKSORT 算法期望的运行时间为 $\Omega(n\lg n)$。

7.4-5　对插入排序来说，当其输入已"几乎"排好序时，运行时间是很快的。在实践中，可以充分利用这一特点来改善快速排序的运行时间。当在一个长度小于 k 的子数组上调用快速排序时，让它不做任何排序就返回。当顶层的快速排序调用返回后，对整个数组运行插

入排序来完成排序过程。证明这一排序算法的期望运行时间为 $O(nk+n\lg(n/k))$。在理论上和实践中，应如何选择 k?

*7.4-6　考虑对 PARTITION 过程做这样的修改：从数组 A 中随机地选出三个元素，并围绕这三个数的中数(即这三个元素的中间值)对它们进行划分。求出以 α 的函数形式表示的、最坏情况中 $\alpha:(1-\alpha)$ 划分的近似概率。

思考题

7-1　Hoare 划分的正确性

本章中给出的 PARTITION 算法并不是其最初的版本。下面给出的是最初由 T. Hoare
159 设计的划分算法：

```
HOARE-PARTITION(A, p, r)
 1  x ← A[p]
 2  i ← p-1
 3  j ← r+1
 4  while TRUE
 5      do repeat j ← j-1
 6           until A[j]≤x
 7         repeat i ← i+1
 8           until A[i]≥x
 9         if i<j
10           then exchange A[i]↔A[j]
11           else return j
```

a)说明 HOARE-PARTITION 算法在数组 $A=\langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\rangle$ 上的运行过程，并说明在第 4～11 行中 for 循环的每一轮迭代后，数组中各元素的值和辅助值的情况。

下面的三个问题要求读者具体地证明过程 HOARE-PARTITION 是正确的。证明以下几点都是正确的：

b)下标 i 和 j 满足这样的特点，即我们从不会访问数组 A 的、在子数组 $A[p..r]$ 之外的元素。

c)当过程 HOARE-PARTITION 结束时，它返回的 j 值满足 $p\leqslant j<r$。

d)当过程 HOARE-PARTITION 结束时，$A[p..r]$ 中的每个元素都小于或等于 $A[j+1..r]$ 中的每一个元素。

在 7.1 节中给出的 PARTITION 过程中，将主元值(原来位于 $A[r]$ 中)与围绕它划分形成的两个部分分隔了开来。与此相反，HOARE-PARTITION 过程则总是将主元值(原先是在 $A[p]$ 中的)放入两个划分 $A[p..j]$ 和 $A[j+1..r]$ 的某一个之中。因为 $p\leqslant j<r$，故这种划分总是非平凡的。

e)利用 HOARE-PARTITION，重写 QUICKSORT 过程。

7-2　对快速排序算法的另一种分析

对随机化快速排序算法的运行时间还有一种分析方法，它着重关注每一次 QUICKSORT 递归调用的期望运行时间，而不是执行的比较次数。

a)证明：给定一个大小为 n 的数组，任何特定元素被选作主元的概率为 $1/n$。利用这个
160 结论来定义指示器随机变量 $X_i=\mathrm{I}\{$第 i 个最小的元素被选作主元$\}$。$\mathrm{E}[X_i]$ 是多少?

b)设 $T(n)$是一个表示快速排序在一个大小为 n 的数组上的运行时间的随机变量。证明：

$$E[T(n)] = E\left[\sum_{q=1}^{n} X_q(T(q-1)+T(n-q)+\Theta(n))\right] \tag{7.5}$$

c)证明公式(7.5)可以简化成

$$E[T(n)] = \frac{2}{n}\sum_{q=0}^{n-1} E[T(q)] + \Theta(n) \tag{7.6}$$

d)证明：

$$\sum_{k=1}^{n-1} k\lg k \leqslant \frac{1}{2}n^2\lg n - \frac{1}{8}n^2 \tag{7.7}$$

(提示：将该和式划分成两个部分，一个针对 $k=1, 2, \cdots, \lceil n/2\rceil - 1$，另一个针对 $k=\lceil n/2\rceil, \cdots, n-1$。)

e)利用式(7.7)中给出的界，证明式(7.6)中的递归式有解 $E[T(n)]=\Theta(n\lg n)$。(提示：通过替换法证明对于某正常数 a 和 b，有 $E[T(n)]\leqslant an\log n - bn$)

7-3 Stooge 排序

Howard、Fine 等教授提出了下面的"漂亮的"排序算法：

```
STOOGE-SORT(A, i, j)
1  if A[i]>A[j]
2     then exchange A[i]↔A[j]
3  if i+1≥j
4     then return
5  k ← ⌊(j-i+1)/3⌋          ▷Round down.
6  STOOGE-SORT(A, i, j-k)    ▷First two-thirds.
7  STOOGE-SORT(A, i+k, j)    ▷Last two-thirds.
8  STOOGE-SORT(A, i, j-k)    ▷First two-thirds again.
```

a)证明：如果 $n=length[A]$，那么 STOOGE-SORT(A, 1, $length[A]$)能正确地对输入数组 $A[1..n]$进行排序。

b)给出一个表达 STOOGE-SORT 最坏情况运行时间的递归式，以及关于最坏情况运行时间的一个紧确的渐近(Θ 记号)界。 161

c)比较 STOOGE-SORT 与插入排序、合并排序、堆排序和快速排序的最坏情况运行时间。这几位终生教授是否真的名符其实呢？

7-4 快速排序中的堆栈深度

7.1 节中的 QUICKSORT 算法包含有两个对其自身的递归调用。在调用 PARTITION 后，左边的子数组和右边的子数组分别被递归排序。QUICKSORT 中的第二次递归调用并不是必须的；可以用迭代控制结构来代替它。这种技术称作*尾递归*，大多数的编译程序都加以了采用。考虑下面这个快速排序的版本，它模拟了尾递归：

```
QUICKSORT'(A, p, r)
1  while p<r
2     do ▷Partition and sort left subarray.
3        q ← PARTITION(A, p, r)
4        QUICKSORT'(A, p, q-1)
5        p ← q+1
```

a)论证 QUICKSORT′(A，1，*length*[A])能正确地对数组 A 进行排序。

编译程序在做递归过程时，常常要用堆栈来存放有关信息，如每一次递归调用的参数等。有关最近一次调用的信息在栈的顶部，而有关第一次调用的信息则在栈的底部。当一个过程被调用时，其信息被压入栈；当它结束时，其信息则被弹出。因为我们假设数组参数是用指针来表示的，则每个过程的信息在栈中只需要 $O(1)$的栈空间。堆栈深度是在一次计算中用到的堆栈空间的最大值。

b)请给出一种在含有 n 个元素的输入数组上 QUICKSORT′的栈深度为 $\Theta(n)$的情况。

c)修改 QUICKSORT′的代码，使其最坏情况栈深度为 $\Theta(\lg n)$。保持算法的 $O(n\lg n)$期望运行时间不变。

7-5　"三数取中"划分

有一种改进 RANDOMIZED-QUICKSORT 的方法，就是根据从子数组中更仔细地选择的(而不是随机选择的)元素作为主元来划分。常用的做法是三数取中：从子数组中随机选出三个元素，取其中间数作为主元。(见练习 7.4-6。)针对我们这个问题，假设数组$A[1..n]$
162 中的元素都不相同，并且有 $n\geqslant 3$。用 $A'[1..n]$表示已排好序的数组。用"三数取中"方法来选择主元元素 x，并定义 $p_i=\Pr\{x=A'[i]\}$。

a)对 $i=2$，3，…，$n-1$，给出 p_i 的以 n 和 i 表示的准确表达式。(注意，$p_1=p_n=0$。)

b)与一般实现比较，这种实现中取 $A[1..n]$的中值 $x=A'[\lfloor(n+1)/2\rfloor]$的可能性增加了多少？假设 $n\to\infty$，请给出这两个概率比值的极限。

c)如果定义一个"好"的划分是选择了 $x=A'[i]$，其中 $n/3\leqslant i\leqslant 2n/3$，则与一般实现相比，这时得到一个好的划分的可能性增加了多少？(提示：用积分来近似和式。)

d)论证对快速排序而言，"三数取中"方法仅影响其运行时间 $\Omega(n\lg n)$中的常数因子。

7-6　对区间的模糊排序

考虑这样的一种排序问题，即无法准确地知道待排序的各个数字到底是多少。对于其中的每个数字，我们只知道它落在实轴上的某个区间内。亦即，给定的是 n 个形如$[a_i, b_i]$的闭区间，其中 $a_i\leqslant b_i$。算法的目标是对这些区间进行模糊排序(fuzzy-sort)，亦即，产生各区间的一个排列$(i_1, i_2, \cdots, i_n)$，使得存在一个 $c_j\in[a_{i_j}, b_{i_j}]$，满足 $c_1\leqslant c_2\leqslant\cdots\leqslant c_n$。

a)为 n 个区间的模糊排序设计一个算法。你的算法应该具有算法的一般结构，它可以快速排序左部端点(即各 a_i)，也要能充分利用重叠区间来改善运行时间。(随着各区间重叠得越来越多，对各区间进行模糊排序的问题会变得越来越容易。你的算法应能充分利用这种重叠。)

b)证明：在一般情况下，你的算法的期望运行时间为 $\Theta(n\lg n)$，但当所有的区间都重叠时，期望的运行时间为 $\Theta(n)$(亦即，当存在一个值 x，使得对所有的 i，都有 $x\in[a_i,$
163 $b_i]$)。你的算法不应显式地检查这种情况，而是应当随着重叠量的增加，性能自然地有所改善。

本章注记

快速排序是由 Hoare[147]提出的；Hoare 的版本可以见本章的思考题 7-1。7.1 节中给出的 PARTITION 过程是由 N. Lomuto 提出的。7.4 节中的分析是由 Avrim Blum 给出的。Sedgewick [268]和 Bentley[40]都对实现的细节及影响给出了很好的参考。

McIlroy[216]说明了如何设计出一个无敌的对手(killer adversary)，它能够产生一个数组，在这个数组上，快速排序的几乎任何实现都能够以 $\Theta(n^2)$时间运行。如果实现是随机化的，该对
164 手可以在看到快速排序算法的随机选择之后，再产生出该数组。

第 8 章 线性时间排序

到目前为止，我们已经介绍了几种能在 $O(n\lg n)$时间内排序 n 个数的算法。合并排序和堆排序在最坏情况下达到此上界，快速排序在平均情况下达到此上界。对于这些算法中的每一个，我们都能给出 n 个输入数值，使算法以 $\Omega(n\lg n)$时间运行。

这些算法都有一个令人感兴趣的性质：排序结果中，各元素的次序基于输入元素间的比较。我们把这类排序算法称为比较排序。到目前为止，我们介绍的所有排序算法都是比较排序算法。

在 8.1 节里，我们将证明对含 n 个元素的一个输入序列，任何比较排序在最坏情况下都要用 $\Omega(n\lg n)$次比较来进行排序。由此可知，合并排序和堆排序是渐近最优的。

8.2 节、8.3 节和 8.4 节讨论三种以线性时间运行的算法：计数排序、基数排序和桶排序。这些算法都用非比较的一些操作来确定排序顺序。因此，下界 $\Omega(n\lg n)$对它们是不适用的。

8.1 排序算法时间的下界

在一个比较排序算法中，仅用比较来确定输入序列$\langle a_1, a_2, \cdots, a_n\rangle$的元素间次序。就是说，给定两个元素 a_i 和 a_j，测试 $a_i<a_j$，$a_i\leqslant a_j$，$a_i=a_j$，$a_i\geqslant a_j$ 或 $a_i>a_j$ 中的哪一个成立，以确定 a_i 和 a_j 之间的相对次序。用任何别的方法都无法得到次序信息。 165

在本节中，不失一般性，我们假设所有输入元素都是不同的。给定了这个假设后，则 $a_i=a_j$ 形式的比较就无用了，因而又可以假设不做这种形式的比较。另外，注意 $a_i\leqslant a_j$，$a_i\geqslant a_j$，$a_i>a_j$ 和 $a_i<a_j$ 是等价的，因为通过它们所得到的关于 a_i 和 a_j 相对次序的信息都是相同的。这样，又可以进一步假设所有的比较都是 $a_i\leqslant a_j$ 的形式。

决策树模型

比较排序可以被抽象地视为决策树。一棵决策树是一棵满二叉树，表示某排序算法作用于给定输入所做的所有比较，而控制结构、数据移动等都被忽略了。图 8-1 中是对应于 2.1 节中插入排序算法作用于含三个元素的输入序列上的决策树。

在决策树中，对每个内结点都注明 $i:j$，其中 $1\leqslant i, j\leqslant n$，$n$ 是输入序列中的元素个数。对每个叶结点都注明排列$\langle\pi(1), \pi(2), \cdots, \pi(n)\rangle$。（有关排列方面的背景知识参阅 C.1 节。）排序算法的执行对应于遍历一条从树的根到叶结点的路径。在每个内节结点处要做比较 $a_i\leqslant a_j$。内结点的左子树决定着 $a_i\leqslant a_j$ 以后的比较，而其右子树则决定着 $a_i>a_j$ 以后的比较。当到达一个叶结点时，排序算法就已确定了顺序 $a_{\pi(1)}\leqslant a_{\pi(2)}\leqslant\cdots\leqslant a_{\pi(n)}$。要使排序算法能正确地工作，其必要条件是，$n$ 个元素的 $n!$ 种排列中的每一种都要作为决策树的一个叶子而出现。因为任何正确的排序算法都必须能够产生其输入的每一种排列，对根结点来说，每

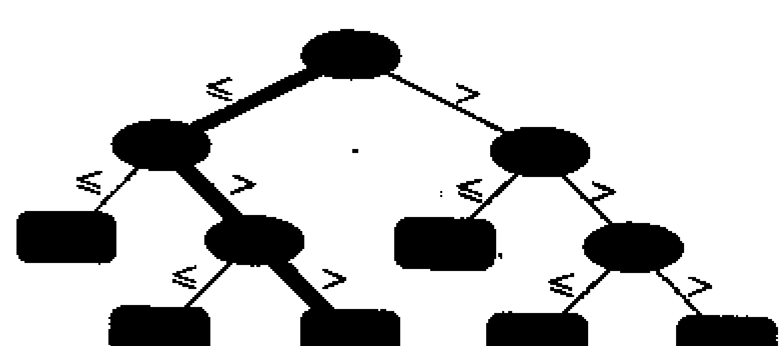


图 8-1 插入排序算法作用于 3 个元素时的决策树。标注为 $i:j$ 的内结点表示在 a_i 和 a_j 之间的比较。排列为$\langle\pi(1), \pi(2), \cdots, \pi(n)\rangle$的叶子表示顺序 $a_{\pi(1)}\leqslant a_{\pi(2)}\leqslant\cdots\leqslant a_{\pi(n)}$。加了阴影的路径表示在对输入序列$\langle a_1=6, a_2=8, a_3=5\rangle$进行排序时所做的决策；叶子上的排列$\langle 3, 1, 2\rangle$表示已排好的顺序 $a_3=5\leqslant a_1=6\leqslant a_2=8$。对输入的元素，共有 $3!=6$ 种可能的排列，因而决策树必须至少有 6 个叶子

一个叶子都必须是可以经由某条路径达到的，该路径对应于比较排序算法的一次实际执行过程。
166 （称这些叶子为“可达的”。）下面将只考虑这样的决策树，即每一种排列都作为可达的叶子出现。

最坏情况下界

在决策树中，从根到任意一个可达叶结点之间最长路径的长度，表示对应的排序算法中最坏情况下的比较次数。这样，一个比较排序算法中的最坏情况比较次数就与其决策树的高度相等。同时，如果决策树中每种排列以可达叶子的形式出现，那么关于其决策树高度的下界也就是关于比较排序算法运行时间的下界。下面的定理给出这样的一个下界。

定理 8.1　任意一个比较排序算法在最坏情况下，都需要做 $\Omega(n \lg n)$ 次的比较。

证明：对于一棵每个排列都作为一个可达叶结点出现的决策树，根据前面的讨论即可确定其高度。考虑一棵高度为 h 的、具有 l 个可达叶结点的决策树，它对应于对 n 个元素所做的比较排序。因为 n 个输入元素共有 $n!$ 种排列，每一种都作为一个叶子出现在树中，故有 $n! \leqslant l$。又由于在一棵高为 h 的二叉树中，叶子的数目不多于 2^h，则有

$$n! \leqslant l \leqslant 2^h$$

对该式取对数，得到

$$\begin{aligned} h &\geqslant \lg(n!) && \text{（因为 lg 函数是单调递增的）} \\ &= \Omega(n \lg n) && \text{（根据(3.18)式）} \end{aligned}$$

■

推论 8.2　堆排序和合并排序都是渐近最优的比较排序算法。

证明：堆排序和合并排序的运行时间上界 $O(n \lg n)$ 与定理 8.1 给出的最坏情况下界 $\Omega(n \lg n)$ 是一致的。

■

练习

8.1-1　在与某种比较排序算法对应的决策树中，一个叶结点最小可能的深度是多少？

167 **8.1-2**　不用斯特林近似公式，给出 $\lg(n!)$ 渐近紧确界。利用 A.2 节中介绍的技术来求和式 $\sum_{k=1}^{n} \lg k$。

8.1-3　证明：对于长度为 n 的 $n!$ 种输入中的至少一半而言，不存在具有线性运行时间的比较排序算法。对 $n!$ 中的 $1/n$ 部分而言又怎样呢？$1/2^n$ 部分呢？

8.1-4　现有 n 个元素要排序。该输入序列包含 n/k 个子序列，每一个包含 k 个元素。每个子序列中的元素都小于后继子序列中的元素，大于前趋子序列中的元素。这样，只要对各 n/k 子序列中的 k 元素排序，就可以得到对整个输入序列长度为 n 的排序结果。证明：这个排序问题中所需的比较次数有一个下界 $\Omega(n \lg k)$。（提示：简单地将每个子序列的下界合并起来的做法是不严格的。）

8.2　计数排序

计数排序假设 n 个输入元素中的每一个都是介于 0 到 k 之间的整数，此处 k 为某个整数。当 $k = O(n)$ 时，计数排序的运行时间为 $\Theta(n)$。

计数排序的基本思想就是对每一个输入元素 x，确定出小于 x 的元素个数。有了这一信息，就可以把 x 直接放到它在最终输出数组中的位置上。例如，如果有 17 个元素小于 x，则 x 就属于第 18 个输出位置。当有几个元素相同时，这个方案要略做修改，因为不能把它们放在同一个输出位置上。

在计数排序算法的代码中，我们假定输入是个数组 $A[1..n]$，$length[A] = n$。另外还需要两个数组：存放排序结果的 $B[1..n]$，以及提供临时存储区的 $C[0..k]$。

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]]+1
 5  ▷C[i]包含等于 i 的元素个数
 6  for i ← 1 to k
 7       do C[i] ← C[i]+C[i−1]
 8  ▷ C[i]包含小于或等于 i 的元素个数
 9  for j ← length[A]downto 1
10       do B[C[A[j]]] ← A[j]
11          C[A[j]] ← C[A[j]]−1
```

168

图 8-2 示出了计数排序的运行过程。在第 1～2 行中的 for 循环初始化操作之后，在第 3～4 行要检查每个输入元素。如果一个输入元素的值为 i，即增加 $C[i]$的值。于是，在第 4 行之后，$C[i]$中存放了等于 i 的元素的个数(整数 $i=0, 1, \cdots, k$)。在第 6～7 行中，通过在数组 C 中记录计数和，可以确定对每一个 $i=0, 1, \cdots, k$，有多少输入元素是小于或等于 i 的。

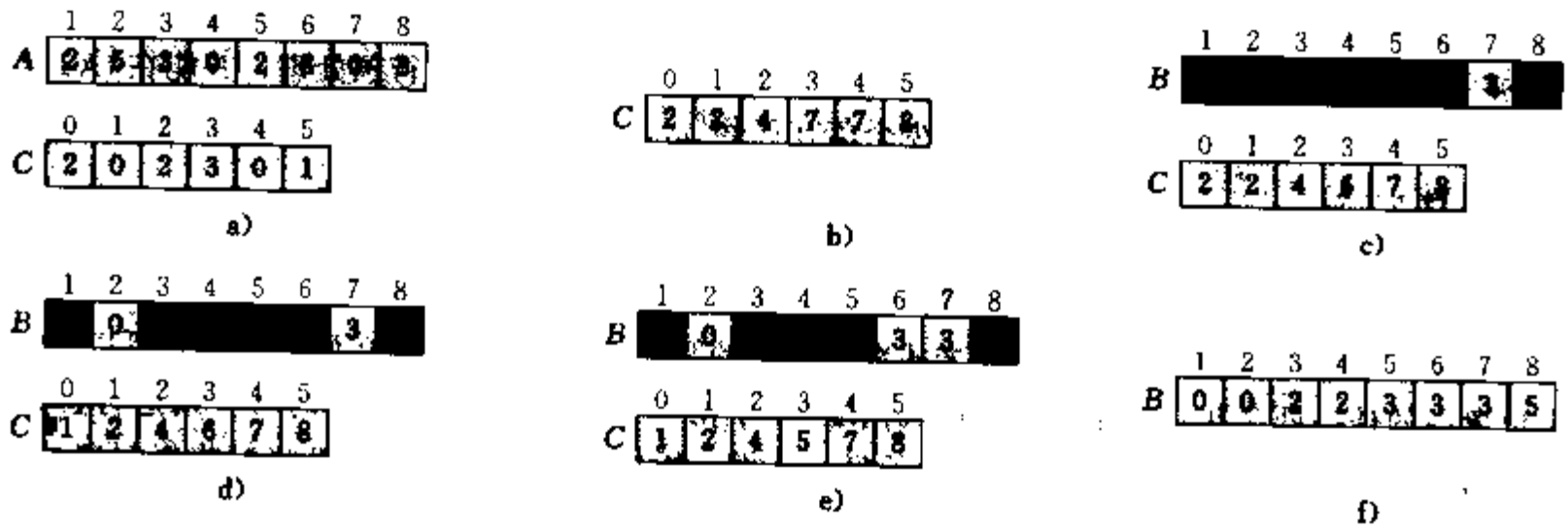


图 8-2　COUNTING-SORT 在一个输入数组 $A[1..8]$上的处理过程，A 中的每一个元素都是不大于 $k=5$ 的非负整数。a)第 4 行执行后，数组 A 和辅助数组 C 的情况；b)第 7 行执行后，数组 C 的情况；c)～e)分别示出了在第 9～11 行中的循环迭代了一次、两次和三次后，输出数组 B 和辅助数组 C 的情况。只有数组 B 中的浅阴影元素被填入了值；f)最终排好序的输出数组 B

最后，在第 9～11 行中的 for 循环部分，把每个元素 $A[j]$放在输出数组 B 中与其相应的最终位置上。如果所有 n 个元素都不相同，则当第一次执行到第 9 行时，对每个 $A[j]$，值 $C[A[j]]$即为 $A[j]$在输出数组中的最终正确位置，因为共有 $C[A[j]]$个元素小于等于 $A[j]$。由于各个元素可能不一定是不同的，因此，每当将一个值 $A[j]$放入数组 B 中时，都要减小 $C[A[j]]$的值。这会使得下一个其值等于 $A[j]$的输入元素(如果存在这样的一个元素的话)直接进入输出数组 B 中 $A[j]$的前一个位置上。

计数排序的时间代价是多少？第 1～2 行的 for 循环所花时间为 $\Theta(k)$。第 3～4 行中 for 循环所花时间为 $\Theta(n)$，第 6～7 行的 for 循环所花时间为 $\Theta(k)$，第 9～11 行的 for 循环所花时间为 $\Theta(n)$。这样，总的时间就是 $\Theta(k+n)$。在实践中，当 $k=O(n)$时，我们常常采用计数排序，这时其运行时间为 $\Theta(n)$。

计数排序的下界优于我们在 8.1 节中证明的 $\Omega(n\lg n)$，因为它不是个比较排序算法。事实上，其代码中根本就不出现输入元素之间的比较。相反，计数排序是用了输入元素的实际值来确定它

169

们在数组中的位置。当我们采用的不是比较排序模型时，排序算法的下界 $\Omega(n \lg n)$ 就不适用了。

计数排序的一个重要性质就是它是稳定的：具有相同值的元素在输出数组中的相对次序与它们在输入数组中的次序相同。亦即，两个相同数之间的顺序是这样来规定的，即在输入数组中先出现的，在输出数组中也位于前面。一般而言，仅当卫星数据随被排序的元素一起移动时，稳定性才显得比较重要。之所以说计数排序的稳定性非常重要，还有另一个原因，即计数排序经常用作基数排序算法的一个子过程。我们将在下一节中看到，计数排序的稳定性对基数排序的正确性来说，是非常关键的。

练习

8.2-1 利用图 8-2 作为模型，说明 COUNTING-SORT 在数组 $A=\langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2\rangle$ 上的处理过程。

8.2-2 证明 COUNTING-SORT 是稳定的。

8.2-3 在 COUNTING-SORT 过程中，假设第 9 行中 for 循环的首部改写成如下形式：

```
9  for j ← 1 to length[A]
```

证明该算法仍能正常地工作。修改后的算法是稳定的吗？

8.2-4 请给出一个算法，使之对于给定的介于 0 到 k 之间的 n 个整数进行预处理，并能在 $O(1)$ 时间内，回答出输入的整数中有多少个落在区间 $[a..b]$ 内。你给出的算法的预处理时间应是 $\Theta(n+k)$。

8.3 基数排序

基数排序(radix sort)是一种用在老式穿卡机上的算法。一张卡片有 80 列，每一列可以在 12 个位置中的任一处穿孔。排序器可以被机械地“程序化”，以便对一叠卡片中的每一列进行检查，
170 再根据穿孔的位置将它们分放入 12 个盒子里。这样，操作员就可以逐个地将它们收集起来，其中第一个位置穿孔的放在最上面，第二个位置穿孔的其次，等等。

对十进制数字来说，每列中只用到 10 个位置。(另两个位置用于编码非数值字符。)一个 d 位数占用 d 个列。因为卡片排序器一次只能查看一个列，因此，要对 n 张卡片上的 d 位数进行排序，就需要用到排序算法。

从直觉上来看，人们可能会觉得应该按最高有效位进行排序，然后再对每个盒子中的数递归地排序，最后再把结果合并起来。不幸的是，为排序每一个盒子中的数，10 个盒子中的 9 个必须先放在一边，这个过程产生了许多要加以记录的中间卡片堆(见练习 8.3-5)。

与人们的直觉相反，基数排序是首先按最低有效位数字进行排序，以解决卡片排序问题。同样，把各堆卡片收集成一叠，其中 0 号盒子中的在 1 号盒子中的前面，后者又在 2 号盒子中的前面，等等。然后，对整个一叠卡片按次低有效位排序，并把结果同样地合并起来。重复这个过程，直到对所有的 d 位数字都进行了排序。所以，仅需要 d 遍就可以将一叠卡片排好序。图 8-3 说明了基数排序作用于“一叠”共 7 个 3 位数的过程。

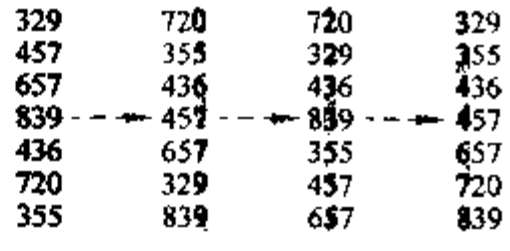


图 8-3　基数排序算法作用于一个由 7 个 3 位数组成的表的过程。最左端的一列为输入。其余各列示出了对各个不断递增的有效位连续排序后表的情况。阴影指示出当前正被排序的数位

关于这个算法，很重要的一点就是按位排序要稳定。由卡片排序器所做的排序是稳定的，

但操作员在把卡片从盒子里拿出来时不能改变它们的次序，即使某一盒子中所有卡片在给定列上的穿孔位置都相同也要注意这一点。

在一台典型的顺序随机存取计算机上，有时采用基数排序来对有多个关键字域的记录进行排序。例如，假设我们想根据三个关键字年、月、日来对日期进行排序。对这个问题，可以用带有比较函数的排序算法来做。给定两个日期，先比较年份，如果相同，再比较月份。如果还是相同，就比较日。此处，我们可以采用另一种方法，即用一种稳定的排序方法对所给信息进行三次 171
排序：先以日，其次对月，再对年。

基数排序算法的代码是很直观的。在下面的过程中，假设长度为 n 的数组 A 中，每个元素都有 d 位数字，其中第 1 位是最低位，第 d 位是最高位。

```
RADIX-SORT(A, d)
1  for i ← 1 to d
2      do use a stable sort to sort array A on digit i
```

引理 8.3　给定 n 个 d 位数，每一个数位可以取 k 种可能的值。基数排序算法能以 $\Theta(d(n+k))$ 的时间正确地对这些数进行排序。

证明：基数排序的正确性可以通过对正在被排序的列进行归纳而加以证明（见练习 8.3-3）。对本算法时间代价的分析要取决于选择哪一种稳定的中间排序算法。当每位数字都界于 0 到 $k-1$ 之间（这样它取 k 种可能的值），且 k 不太大时，可以选择计数排序。对 n 个 d 位数的每一遍处理的时间为 $\Theta(n+k)$，共有 d 遍，故基数排序的总时间为 $\Theta(d(n+k))$。 ■

当 d 为常数、$k=O(n)$ 时，基数排序有线性运行时间。更一般地，在如何将每个关键字分解成若干数位方面，我们有了一定的灵活性。

引理 8.4　给定 n 个 b 位数和任何正整数 $r \leqslant b$，RADIX-SORT 能在 $\Theta((b/r)(n+2^r))$ 时间内正确地对这些数进行排序。

证明：对于一个值 $r \leqslant b$，将每个关键字看作是有 $d=\lceil b/r \rceil$ 个数字，每个数字都包含 r 位。每一个数字都是界于 0 到 2^r-1 之间的一个整数，这样就可以采用计数排序，此处 $k=2^r-1$。（例如，可以将一个 32 位的字视为有 4 个 8 位的数字，于是有 $b=32$，$r=8$，$k=2^r-1=255$，$d=b/r=4$。）计数排序的每一遍所需时间为 $\Theta(n+k)=\Theta(n+2^r)$，共有 d 遍，总的运行时间为 $\Theta(d(n+2^r))=\Theta((b/r)(n+2^r))$。 ■

对于给定的 n 值和 b 值，我们希望所选择的 r 值（$r \leqslant b$）能够最小化表达式 $(b/r)(n+2^r)$。如果 $b<\lfloor \lg n \rfloor$，则对于任何满足 $r \leqslant b$ 的 r 值，都有 $(n+2^r)=\Theta(n)$。于是，选择 $r=b$ 得到的运行时间为 $(b/b)(n+2^b)=\Theta(n)$，从渐近意义上来看，这一时间是最优的。如果 $b \geqslant \lfloor \lg n \rfloor$，则选择 $r=\lfloor \lg n \rfloor$ 可以给出在某一常数因子内的最佳时间，这一点从下面的说明中就能看出来。选择 $r=\lfloor \lg n \rfloor$ 得到的运行时间为 $\Theta(bn/\lg n)$。随着将 r 增大到 $\lfloor \lg n \rfloor$ 之上时，分子中的 2^r 项增加得比分母 172
中的 r 项更快，因此，将 r 增大到 $\lfloor \lg n \rfloor$ 之上时得到的时间为 $\Omega(bn/\lg n)$。如果不是这么做，而是将 r 减小到 $\lfloor \lg n \rfloor$ 之下，则 b/r 项即会变大，而 $n+2^r$ 项仍保持为 $\Theta(n)$。

基数排序是否要比基于比较的排序算法（如快速排序）更好呢？如果根据常见的情况有 $b=O(\lg n)$，并选择 $r \approx \lg n$，则基数排序的运行时间为 $\Theta(n)$，这看上去要比快速排序的平均情况时间 $\Theta(n \lg n)$ 更好一些。然而，在这两个时间中，隐含在 Θ 记号中的常数因子是不同的。对于要处理的 n 个关键字，尽管基数排序执行的遍数可能比快速排序要少，但每一遍所需的时间都要长得多。哪一个排序算法更好取决于底层机器的实现特性（例如，快速排序通常可以比基数排序更为有效地利用硬件缓存），同时还取决于输入的数据。此外，利用计数排序作为中间稳定排序的基数排序不是原地排序，而很多 $\Theta(n \lg n)$ 时间的比较排序算法则可以做到原地排序。因此，当内存

容量比较宝贵时，像快速排序这样的原地排序算法可能是更为可取的。

练习

8.3-1 仿照图 8-3，示出基数排序 RADIX-SORT 作用于下列英语单词上的过程：COW，DOG，SEA，RUG，ROW，MOB，BOX，TAB，BAR，EAR，TAR，DIG，BIG，TEA，NOW，FOX。

8.3-2 下面的算法中哪些是稳定的：插入排序，合并排序，堆排序和快速排序？给出一个能使任何排序算法都稳定的方法。所给出的方法带来的额外时空开销是多少？

8.3-3 利用归纳法来证明基数排序算法能正常工作。在所给出的证明中，哪个地方要假设中间排序是稳定的？

8.3-4 说明如何在 $O(n)$时间内，对 0 到 n^2-1 之间的 n 个整数进行排序。

***8.3-5** 在本节的第一个卡片排序算法中，为排序 d 位十进数，在最坏情况下需要排序几遍？最坏情况下操作员要看管几堆卡片？

173

8.4　桶排序

当桶排序(bucket sort)的输入符合均匀分布时，即可以以线性时间运行。与计数排序类似，桶排序也对输入作了某种假设，因而运行得很快。具体来说，计数排序假设输入是由一个小范围内的整数构成，而桶排序则假设输入由一个随机过程产生，该过程将元素均匀地分布在区间[0，1)上(见 C.2 节中均匀分布的含义)。

桶排序的思想就是把区间[0，1)划分成 n 个相同大小的子区间，或称桶。然后，将 n 个输入数分布到各个桶中去。因为输入数均匀分布在[0，1)上，所以，一般不会有很多数落在一个桶中的情况。为得到结果，先对各个桶中的数进行排序，然后按次序把各桶中的元素列出来即可。

在桶排序算法的代码中，假设输入的是一个含 n 个元素的数组 A，且每个元素满足 $0 \leqslant A[i]<1$。另外，还需要一个辅助数组 $B[0..n-1]$来存放链表(桶)，并假设可以用某种机制来维护这些表。(10.2 节介绍了如何实现关于链表的一些基本操作。)

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3       do insert A[i] into list B[⌊nA[i]⌋]
4  for i ← 0 to n−1
5       do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], …,
   B[n−1] together in order
```

图 8-4 示出了桶排序作用于有 10 个数的输入数组上的操作过程。

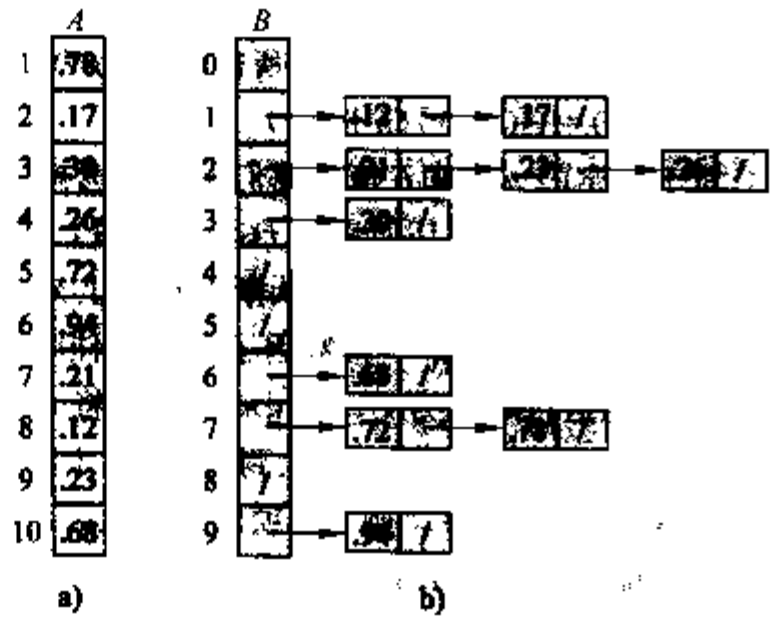


图 8-4　BUCKET-SORT 的操作过程。a)输入数组 $A[1..10]$；b)在算法的第 5 行之后，由已排序链表(桶)组成的数组 $B[0..9]$。第 i 个桶中存放了半开区间$[i/10, (i+1)/10)$中的值。已排好序的输出由各链表按 $B[0]$，$B[1]$，…，$B[9]$的顺序拼接而成

为了说明这个算法能正确地工作，我们来看两个元素 $A[i]$和 $A[j]$。不失一般性，假设 $A[i] \leqslant A[j]$。由于 $\lfloor nA[i] \rfloor \leqslant \lfloor nA[j] \rfloor$，元素 $A[i]$或者被放入 $A[j]$所在桶中，或者被放入一个下标更小的桶中。如果 $A[i]$和 $A[j]$落在同

一个桶中，则第 4～5 行中的 for 循环会将它们按适当的顺序排列。如果 $A[i]$和 $A[j]$落入了不同的桶中，则第 6 行会将它们按适当的顺序排列。因此，桶排序是能正确地工作的。

现在来分析桶排序算法的运行时间。除第 5 行外，所有其他各行在最坏情况下的时间都是 $O(n)$。仍然需要平衡对第 5 行中插入排序的 n 次调用所花的总时间。

为分析调用插入排序的时间代价，设 n_i 为表示桶 $B[i]$中元素个数的随机变量。因为插入排序以二次时间运行(见 2.2 节)，因而桶排序的运行时间：

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

对上式两边取期望，并利用期望的线性性质，我们有：

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] && \text{根据期望的线性性质} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) && \text{根据方程(C.21)} \end{aligned} \tag{8.1}$$

下式：

$$\mathrm{E}[n_i^2] = 2 - 1/n \tag{8.2}$$

对 $i=0, 1, \cdots, n-1$ 是成立的。每一个桶 i 有相同的值 $\mathrm{E}[n_i^2]$这一点是不足为奇的，因为输入数组 A 中的每一个值都是等可能地落在任何桶内的。为了证明公式(8.2)，我们定义指示器随机变量

$$X_{ij} = I\{A[j] \text{ 落在桶 } i \text{ 中}\}$$

其中，$i=0, 1, \cdots, n-1$，$j=1, 2, \cdots, n$。于是，

174
⌇
175

$$n_i = \sum_{j=1}^{n} X_{ij}$$

为了计算 $\mathrm{E}[n_i^2]$，我们将平方项加以扩展，并对各个项进行重新组合：

$$\begin{aligned} \mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] = \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] = \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij}X_{ik}\right] \\ &= \sum_{j=1}^{n} \mathrm{E}[X_{ij}^2] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} \mathrm{E}[X_{ij}X_{ik}] \end{aligned} \tag{8.3}$$

其中，最后一行是根据数学期望的线性性质而得出的。我们分别地计算两个和式。指示器随机变量 X_{ij} 为 1 的概率为 $1/n$，其他情况下为 0。于是有：

$$\mathrm{E}[X_{ij}^2] = 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n}$$

当 $k \ne j$ 时，变量 X_{ij} 和 X_{ik} 是独立的，因而有：

$$\mathrm{E}[X_{ij}X_{ik}] = \mathrm{E}[X_{ij}]\mathrm{E}[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$

将这两个期望值替换进式(8.3)中，有：

$$\mathrm{E}[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} \frac{1}{n^2} = n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n} \tag{8.4}$$

176

于是式(8.2)得证。

在式(8.1)中利用这个期望值，可以得出这样一个结论，即桶排序的期望运行时间为

$\Theta(n)+n\cdot O(2-1/n)=\Theta(n)$。于是，整个桶排序算法以线性期望时间运行。

即使输入不符合均匀分布，桶排序也仍然可以以线性时间运行。只要输入满足这样一个性质，即各个桶尺寸的平方和与总的元素数呈线性关系，那么，通过式(8.1)就可以知道，桶排序仍然能以线性时间运行。

练习

8.4-1　仿照图 8-4，说明 BUCKET-SORT 作用于数组 A＝⟨0.79，0.13，0.16，0.64，0.39，0.20，0.89，0.53，0.71，0.42⟩上的过程。

8.4-2　桶排序的最坏情况运行时间是什么？如果要在保持其线性运行时间的同时，使最坏情况时间为 $O(n\lg n)$，要对算法做什么样的修改？

8.4-3　设 X 是一个随机变量，用于表示在将一枚硬币抛掷两次时，正面朝上的次数。$E[X^2]$是多少？$E^2[X]$是多少？

*8.4-4　在单位圆中有 n 个点，$p_i=(x_i, y_i)$，使得 $0<x_i^2+y_i^2\leqslant 1$，$i=1, 2, \cdots, n$。假设所有点是均匀分布的，亦即，某点落在圆的任一区域中的概率与该区域的面积成正比。请设计一个 $\Theta(n)$期望时间的算法，来根据点到原点之间的距离 $d_i=\sqrt{x_i^2+y_i^2}$ 对 n 个点排序。(提示：在 BUCKET-SORT 算法中，设计适当的桶尺寸，以反映各个点在单位圆中的均匀分布。)

*8.4-5　一个随机变量 X 的概率分布函数 $P(x)$定义为 $P(x)=\Pr\{X\leqslant x\}$。假设 n 个随机变量 X_1，X_2，…，X_n 符合一个连续概率分布函数 P，它可以在 $O(1)$时间内计算。说明如何在线性期望时间内排序这 n 个数。

177

思考题

8-1　比较排序的平均情况下界

在本问题中，我们来证明给定 n 个不同的输入元素，对于任何确定或随机的比较排序算法，其期望运行时间都有下界 $\Omega(n\lg n)$。首先来分析一个确定的比较排序算法 A，其决策树为 T_A。假设 A 的输入的每一种排列都是等可能的。

a)假设 T_A 的每个叶结点都标以在给定的随机输入下到达该结点的概率。证明：恰有 $n!$ 个叶结点标有 $1/n!$，其他的标有 0。

b)设 $D(T)$表示一棵决策树 T 的外路径长度，也就是说，$D(T)$是 T 的所有叶结点深度的和。设 T 为一棵决策树，其叶子数 $k>1$，并设 RT 和 LT 为 T 的右子树和左子树。证明：$D(T)=D(RT)+D(LT)+k$。

c)设 $d(k)$为所有具有 $k>1$ 个叶结点的决策树 $D(T)$的最小值。证明：$d(k)=\min\limits_{1\leqslant i\leqslant k-1}\{d(i)+d(k-i)+k\}$。(提示：考虑一棵能取此最小值的、有 k 个叶结点的决策树 T。设 i_0 为 LT 中的叶结点数，$k-i_0$ 为 RT 中的叶结点数。)

d)证明：对 k 的某一给定值($k>1$)和范围 $1\leqslant i\leqslant k-1$ 内的 i 值，函数 $i\lg i+(k-i)\lg(k-i)$在 $i=k/2$ 处取得最小值。总结 $d(k)=\Omega(k\lg k)$。

e)证明：$D(T_A)=\Omega(n!\lg(n!))$，并给出排序 n 个元素的期望时间为 $\Omega(n\lg n)$的结论。

现在来考虑一个随机化的比较排序算法 B。我们可以将决策树模型加以扩展来处理随机化的情形，方法是采用两类结点：普通的比较结点和“随机化”结点。后一种结点模拟了算法 B 所做的形如 RANDOM(1，r)的随机选择：该类结点有 r 个子女，其中的每一个在算

法的执行过程中，被选择的可能性相同。

f)证明：对任何随机化比较排序算法 B，存在一个确定的比较排序算法 A，平均情况下它所做的比较次数不多于 B。

8-2　以线性时间原地置换排序

假设有一个由 n 个数据记录组成的数组要排序，且每个记录的关键字的值为 0 或 1。排序这样一组记录的一个算法应具备如下三种特性中的一部分：

1)算法的运行时间为 $O(n)$。

2)算法是稳定的。

3)算法是原地进行排序的，它可以使用除输入数组之外的固定量的存储空间。

a)给出一个满足上述条件 1 和条件 2 的算法。

b)给出一个满足上述条件 1 和条件 3 的算法。

c)给出一个满足上述条件 2 和条件 3 的算法。

d)在 a)～c)中给出的算法能否用来在 $O(bn)$时间内，对有 b 位关键字的 n 个记录进行基数排序？如果行，说明如何做；如果不行，说明原因。

e)假设 n 个记录中每一个的关键字都界于 1 到 k 之间。说明如何修改计数排序，使得可以在 $O(n+k)$时间内对 n 个记录原地排序。除输入数组外，可以另用 $O(k)$的存储空间。你给出的算法是稳定的吗？(提示：当 $k=3$ 时应该如何做？)

8-3　排序不同长度的数据项

a)给定一个整数数组，其中不同的整数中包含的数字个数可能不同，但该数组中，所有整数中总的数字数为 n。说明如何在 $O(n)$时间内对该数组进行排序。

b)给定一个字符串数组，其中不同的串包含的字符数可能不同，但所有串中总的字符个数为 n。说明如何在 $O(n)$时间内对该数组进行排序。

(注意此处的顺序是指标准的字母顺序，例如，a<ab<b。)

8-4　水壶

假设给定了 n 个红色的水壶和 n 个蓝色的水壶，它们的形状和尺寸都不相同。所有红色水壶中所盛水的量都不一样，蓝色水壶也是一样。此外，对于每一个红色的水壶，都有一个对应的蓝色水壶，两者所盛的水量是一样的。反之亦然。

你的任务是将所盛水量一样的红色水壶和蓝色水壶找出来。为了达到这一目的，可以执行如下操作：挑选出一对水壶，其中一个是红色的，另一个是蓝色的；将红色水壶中倒满水；再将水倒入蓝色的水壶中。通过这个操作，可以判断出来这两只水壶的容量哪一个大，或者是一样大。假设这样的比较需要一个时间单位。你的目标是找出一个算法，它通过执行最少次数的比较，来确定分组和配对问题。记住不能直接比较两个红色的或两个蓝色的水壶。

a)给出一个确定型的算法，它利用 $\Theta(n^2)$次比较来完成水壶的配对。

b)证明：对于一个可以解决本问题的算法，必须执行的比较次数的下界为 $\Omega(n\lg n)$。

c)给出一个随机化的算法，其期望的比较次数为 $O(n\lg n)$，并证明这个界是正确的。对你的算法来说，最坏情况下的比较次数是什么？

8-5　平均排序

假设我们不是要排序一个数组，而只是要求数组中的元素一般情况下都是呈递增序的。更准确地说，称一个包含 n 个元素的数组 A 为 k 排序的(k-sorted)，如果对所有的 $i=1, 2, \cdots, n-k$，有下式成立：

$$\frac{\sum_{j=i}^{(i+k-1)} A[j]}{k} \leqslant \frac{\sum_{j=(i+1)}^{(i+k)} A[j]}{k}$$

a)说一个数组是 1 排序的是什么意思？

b)给出数字 1，2，…，10 的一个排列，它是 2 排序的，但不是完全排序的。

c)证明：一个 n 元素的数组是 k 排序的，当且仅当对所有的 $i=1, 2, \cdots, n-k$，有 $A[i] \leqslant A[i+k]$。

d)给出一个算法，它能在 $O(n\lg(n/k))$时间内，对一个 n 元素的数组进行 k 排序。

e)说明一个长度为 n 的 k 排序的数组可以在 $O(n\lg k)$内排序。(提示：可以利用练习 6.5-8 的结果。)

f)说明当 k 是一个常量时，需要 $\Omega(n\lg n)$时间来 k 排序一个 n 元素的数组。(提示：可以利用前一部分的结果及比较排序的下界。)

8-6　合并已排序列表的下界

合并两个已排序列表这样的问题是经常出现的。它是 MERGE-SORT 的一个子过程，
180 而合并两个已排序列表的过程在 2.3.1 节中是作为 MERGE 给出的。在本问题中，我们将说明对于两个排序列表(每个表中有 n 个项目)的合并问题，在最坏情况下，所需的比较次数有一个下界 $2n-1$。

首先，利用决策树来说明比较次数有一个下界 $2n-o(n)$。

a)说明给定 $2n$ 个数，共有 $\binom{2n}{n}$ 种可能的方式来将它们划分成两个已排序的列表，每个列表中有 n 个数。

b)利用决策树，说明任何能正确合并两个已排序列表的算法都至少要进行 $2n-o(n)$次比较。

现在我们就来说明还有一个更紧确的 $2n-1$ 界。

c)说明如果两个元素在已排好的顺序中是连续的，并且是来自相对的列表中的，则它们必须进行比较。

d)利用你对前一部分的回答，说明合并两个已排序列表时，比较次数有着下界 $2n-1$。

本章注记

用于研究比较排序的决策树模型是由 Ford 和 Johnson[94]提出的。Knuth 有关排序的综合性论述[185]中涉及了排序问题的很多变形，包括本章所给出的排序问题复杂性的信息理论下界。Ben-Or[36]全面研究了利用泛化的决策树模型进行排序的下界。

根据 Knuth，计数排序是由 H. H. Seward 于 1954 年提出的，此外，他还提出了将计数排序与基数排序结合起来的思想。基数排序是从最低位开始进行的，是一种比较通俗的算法，为机械式卡片排序机的操作员们所广泛采用。根据 Knuth，L. J. Comrie 于 1929 年公开发表第一篇有关该算法的文章，介绍有关穿孔卡片设备的情况。桶排序自从 1956 年就开始使用，当时该算法的基本想法是由E. J. Issac 和 R. C. Singleton 提出来的。

Munro 和 Raman[229]给出一个稳定的排序算法，它在最坏情况下执行 $O(n^{1+\varepsilon})$次比较，其中 $0<\varepsilon\leqslant 1$是任何固定的常量。尽管任何 $O(n\lg n)$时间的算法所做的比较要更少一些，但算法 Munro 和 Raman 仅将数据移动 $O(n)$次，而且是原地排序的。

许多研究人员对在 $o(n\lg n)$时间内排序 n 个 b 位整数这一问题进行研究，并获得一些有益的

成果，其中每一项成果都对计算模型做了略有不同的假设，对算法的限制也略有不同。所有的成果都假设计算机内存被划分成一系列可寻址的 b 位字。Fredman 和 Willard[99]引入融合树 [181]
(fusion tree)数据结构，并利用它在 $O(n \lg n/\lg\lg n)$时间内，对 n 个整数进行排序。Andersson [16]后来将这个界改善到$O(n\sqrt{\lg n})$。这些算法要用到乘法和几个预先计算好的常量。Anderson、Hagerup、Nilsson 和 Raman[17]说明如何在 $O(n\lg\lg n)$时间内，不用乘法即可对 n 个整数进行排序。但是，他们的方法所用到的存储空间可能无法以 n 的界的形式来表示。利用乘法散列技术(multiplicative hashing)，可以将所需的存储空间降至 $O(n)$，但运行时间的最坏情况界 $O(n\lg\lg n)$成为期望时间界。Thorup[297]将 Andersson 提出的指数搜索树[16]加以推广，给出一个 $O(n(\lg\lg n)^2)$时间的排序算法，它不使用乘法或随机化，而是用到了线性空间。Han[137]将这些技术与一些新的想法结合起来，将排序算法的界改善至$O(n\lg\lg n\lg\lg\lg n)$时间。尽管这些算法在理论上有着重要的突破，但都相当复杂，在目前来看，不太可能与现有的、正在实践中使用的排序算法竞争。 [182]

第 9 章　中位数和顺序统计学

在一个由 n 个元素组成的集合中，第 i 个顺序统计量(order statistic)是该集合中第 i 小的元素。例如，在一组元素所组成的集合中，最小值是第 1 个顺序统计量($i=1$)，最大值是第 n 个顺序统计量($i=n$)。非形式地说，一个中位数(median)是它所在集合的"中点元素"。当 n 为奇数时，中位数是唯一的，出现在 $i=(n+1)/2$ 处。当 n 为偶数时，存在两个中位数，分别出现在 $i=n/2$ 和 $i=n/2+1$ 处。因此，不考虑 n 的奇偶性，中位数总是出现在 $i=\lfloor(n+1)/2\rfloor$ 处(下中位数)和 $i=\lceil(n+1)/2\rceil$ 处(上中位数)。为简单起见，本书中所用的"中位数"总是指下中位数。

本章讨论从一个由 n 个不同数值构成的集合中选择其第 i 个顺序统计量的问题。为方便起见，假设集合中的数互异，但实际上我们做的所有处理都可以推广到集合中包含重复数值的情形。可以如下形式化地定义选择问题(selection problem)：

输入：一个包含 n 个(不同的)数的集合 A 和一个数 i，$1\leqslant i\leqslant n$。

输出：元素 $x\in A$，它恰大于 A 中其他的 $i-1$ 个元素。

选择问题可在 $O(n\lg n)$ 时间内解决，因为可以用堆排序或合并排序对输入数据进行排序，然后在输出数组中标出第 i 个元素即可。但是还有其他更快的算法。

在 9.1 节中，要讨论从一个集合中选择最大元素和最小元素的问题。更有意思的是一般选择问题，这类问题在接下来的两节中进行讨论。9.2 节分析一个实用的算法，它在平均情况下的运行时间界为 $O(n)$。9.3 节包含一个更具有理论意义的算法，它在最坏情况下的运行时间为 $O(n)$。

9.1　最小值和最大值

在一个有 n 个元素的集合中，要做多少次比较才能确定其最小元素呢？可以很容易地给出 $n-1$ 次比较这个上界：依次查看集合中的每个元素，并记录比较过程中的最小元素。在下面的过程中，假设集合存放于数组 A 中，且 $length[A]=n$。

```
MINIMUM(A)
1  min ← A[1]
2  for i ← 2 to length[A]
3        do if min>A[i]
4              then min ← A[i]
5  return min
```

当然，最大值也可以通过 $n-1$ 次比较找出来。

这是我们能做到的最好结果吗？是的，因为对确定最小值的问题，可以得到 $n-1$ 次比较这一下界。对于任意一个确定最小值的算法，可以把它看成是在各元素之间进行的一场锦标赛。每次比较都是锦标赛中的一场比赛，两个元素中较小的一个获胜。有一点很关键，就是除了最终获胜者之外，每个元素都要输掉至少一场比赛。因此，为确定最小值而做 $n-1$ 次比较是必须的。从所执行的比较次数来看，算法 MINIMUM 是最优的。

同时找出最小值和最大值

在某些应用中，必须找出 n 个元素集合中的最小值和最大值。例如，一个图形程序可能会要求变换一组数据(x, y)，使之能适合一个矩形显示屏幕或其他图形输出装置。为做到这一点，程序必须首先确定每个坐标中的最小值和最大值。

要设计出一个算法，使之通过渐近最优的 $\Theta(n)$ 次比较，就能从 n 个元素中找出最小值和最大值，做到这一点并不困难。只要独立地找出最小值和最大值，各用 $n-1$ 次比较，共有 $2n-2$ 次比较。

事实上，至多 $3\lfloor n/2 \rfloor$ 次比较就足以同时找到最小值和最大值。做法是记录比较过程中遇到的最小值和最大值。我们并不是将每一个输入元素与当前的最小值和最大值进行比较(这样做的代价是每个元素需要 2 次比较)，而是成对地处理元素。先将一对输入元素互相比较，然后把较 184
小者与当前最小值比较，把较大者与当前最大值比较，因此每两个元素需要 3 次比较。

如何设定当前最小值和最大值的初始值依赖于 n 是奇数还是偶数。如果 n 是奇数，就将最小值和最大值都设为第一个元素的值，然后成对地处理余下的元素。如果 n 是偶数，就对前两个元素做一次比较，以决定最小值和最大值的初值，然后如同 n 是奇数的情形一样，成对地处理余下的元素。

下面来分析一下总的比较次数。如果 n 是奇数，那么总共做了 $3\lfloor n/2 \rfloor$ 次比较。如果 n 是偶数，我们是先做一次初始比较，接着做 $3(n-2)/2$ 次比较，总计 $3n/2-2$ 次比较。因此，不管是哪一种情况，总的比较次数至多是 $3\lfloor n/2 \rfloor$。

练习

9.1-1　证明：在最坏情况下，利用 $n+\lceil \lg n \rceil - 2$ 次比较，即可找到 n 个元素中的第 2 小元素。(提示：同时找最小元素。)

***9.1-2**　证明：在最坏情况下，同时找到 n 个数字中的最大值和最小值需要 $\lceil 3n/2 \rceil - 2$ 次比较。(提示：考虑有多少个数字可能会是最大值或最小值，然后分析一下每一次比较会如何影响这些计数。)

9.2　以期望线性时间做选择

一般选择问题看起来要比找最小值的简单选择问题更难。但令人惊奇的是，两种问题的渐近运行时间却是相同的：都是 $\Theta(n)$。本节将介绍一种用来解决选择问题的分治算法，即 RANDOMIZED-SELECT 算法，它以第 7 章的快速排序算法为模型。如同在排速排序中一样，此算法的思想也是对输入数组进行递归划分。但和快速排序不同的是，快速排序会递归处理划分的两边，而 RANDOMIZED-SELECT 只处理划分的一边。这一差异在分析中就体现出来了：快速排序的期望运行时间是 $\Theta(n \lg n)$，而 RANDOMIZED-SELECT 的期望时间为 $\Theta(n)$。

RANDOMIZED-SELECT 利用了 7.3 节介绍的 RANDOMIZED-PARTITION 程序。所以，与 RANDOMIZED-QUICKSORT 一样，本算法是一个随机算法，因为它的行为部分地由随机数 185
生成器的输出来决定。以下是 RANDOMIZED-SELECT 的伪代码，它返回数组 $A[p..r]$ 中的第 i 小的元素。

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p=r
2     then return A[p]
3  q ← RANDOMIZED-PARTITION(A, p, r)
4  k ← q-p+1
5  if i=k              ▷ the pivot value is the answer
6     then return A[q]
7  elseif i<k
8     then return RANDOMIZED-SELECT(A, p, q-1, i)
9  else return RANDOMIZED-SELECT(A, q+1, r, i-k)
```

在算法第3行的RANDOMIZED-PARTITION执行之后，数组$A[p..r]$被划分成两个(可能是空的)子数组$A[p..q-1]$和$A[q+1..r]$，使得$A[p..q-1]$中的每个元素都小于或等于$A[q]$，进而小于$A[q+1..r]$中的每个元素。与快速排序中一样，称$A[q]$为主元元素(pivot element)。RANDOMIZED-SELECT的第4行计算子数组$A[p..q]$内的元素个数k，即处于划分低区的元素的个数加上一个主元元素。然后，第5行检查$A[q]$是不是第i小的元素。如果是，就返回$A[q]$。否则，算法要确定第i小的元素落在两个子数组$A[p..q-1]$和$A[q+1..r]$中的哪一个之中。如果$i<k$，则要找的元素落在划分的低区中，于是，第8行就在低区的子数组中进一步递归选择。如果$i>k$，则要找的元素落在划分的高区子数组中。因为我们已经知道了有k个值(即$A[p..q]$中的所有元素)小于$A[p..r]$中的第i小元素，故所求元素必是$A[q+1..r]$中的第$(i-k)$小元素，它在第9行中被递归地选取。这个程序看起来允许递归调用含有0个元素的子数组，但在练习9.2-1中，会要求读者证明这种情况不可能发生。

RANDOMIZED-SELECT的最坏情况运行时间为$\Theta(n^2)$，即使是要选择最小元素也是如此，因为在每次划分时可能极不走运，总是按余下的元素中最大的进行划分，而划分操作需要$\Theta(n)$的时间。该算法的平均情况性能较好，又因为它是随机化的，故没有哪一种特别的输入会导致其最坏情况的发生。

当RANDOMIZED-SELECT作用于一个含有n个元素的输入数组$A[p..r]$上时，所需时间是一个随机变量，记为$T(n)$，我们如下来得到$E[T(n)]$的一个上界。程序RANDOMIZED-
186 PARTITION以相等的可能性返回任何元素作为主元。因此，对每一个$k(1\leqslant k\leqslant n)$，子数组$A[p..q]$有$k$个元素(全部小于或等于主元元素)的概率是$1/n$。对$k=1, 2, \cdots, n$，定义指示器随机变量$X_k$为：

$$X_k = \mathrm{I}\{\text{子数组 } A[p..q] \text{ 中恰有 } k \text{ 个元素}\}$$

因此有：

$$\mathrm{E}[X_k] = 1/n \tag{9.1}$$

当调用RANDOMIZED-SELECT，并且选择$A[q]$作为主元元素的时候，事先并不知道是否会立即得到正确答案而结束，或者在子数组$A[p..q-1]$上递归，或者在子数组$A[q+1..r]$上递归。这个决定依第i小元素相对于$A[q]$的位置而定。假设$T(n)$是单调递增的，我们可以将递归调用所需时间的界限设定为最大可能输入的递归调用所需的时间。换言之，为得到一个上界，我们假定第i个元素总是在划分的较大的一边。对一个给定的RANDOMIZED-SELECT，指示器随机变量X_k刚好在一个k值上取值1，在其他的k值时都是0。当$X_k=1$时，可能要递归处理的两个子数组的大小分别为$k-1$和$n-k$。因此，可以得到递归式：

$$\begin{aligned} T(n) &\leqslant \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} (X_k \cdot T(\max(k-1, n-k)) + O(n)) \end{aligned}$$

取期望值，得到

$$\begin{aligned} \mathrm{E}[T(n)] &\leqslant \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\ &= \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n) && \text{(根据线性期望)} \\ &= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(根据(C.23))} \end{aligned}$$

$$= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) \qquad \text{(根据式(9.1))}$$

为了能应用等式(C.23)，我们依赖于 X_k 和 $T(\max(k-1, n-k))$是独立的随机变量。练习 9.2-2 要求读者证明这个命题。

下面来考虑一下表达式 $\max(k-1, n-k)$。我们有 187

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{如果 } k > \lceil n/2 \rceil \\ n-k & \text{如果 } k \leqslant \lceil n/2 \rceil \end{cases}$$

如果 n 是偶数，从 $T(\lceil n/2 \rceil)$到 $T(n-1)$的每个项在总和中刚好出现两次，如果 n 是奇数，所有这些项都会出现两次，而 $T(\lfloor n/2 \rfloor)$出现一次。因此有：

$$\mathrm{E}[T(n)] \leqslant \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(k)] + O(n)$$

我们用替换法来解上面的递归式。假设对满足这个递归式初始条件的某个常数 c，有 $T(n) \leqslant cn$。假设对于小于某个常数的 n，$T(n) = O(1)$；我们稍后再来选取这个常数。同时，还要选择一个常数 a，使得对所有的 $n > 0$，由上式中 $O(n)$项(用来描述这个算法的运行时间中非递归的部分)所描述的函数可由 an 从上方限界。利用这个归纳假设，可以得到：

$$\mathrm{E}[T(n)] \leqslant \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an = \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k\right) + an$$

$$= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2}\right) + an$$

$$\leqslant \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an$$

$$= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2}\right) + an$$

$$= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an = c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an$$

$$\leqslant \frac{3cn}{4} + \frac{c}{2} + an = cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right)$$

为了完成证明，还需要证明对足够大的 n，上面最后一个表达式至多是 cn，或等价地说，$cn/4 - c/2 - an \geqslant 0$。如果在两边加上 $c/2$，并且提取因子 n，就可以得到 $n(c/4 - a) \geqslant c/2$。只要我 188
们选择的常数 c 能满足 $c/4 - a > 0$，即 $c > 4a$，就可以将两边同除以 $c/4 - a$，得到：

$$n \geqslant \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}$$

所以，如果假设对 $n < 2c/(c-4a)$，有 $T(n) = O(1)$，就有 $T(n) = O(n)$。可以得出这样的结论：在平均情况下，任何顺序统计量(特别是中位数)都可以在线性时间内得到。

练习

9.2-1　证明：在 RANDOMIZED-SELECT 中，对长度为 0 的数组，不会进行递归调用。

9.2-2　证明：指示器随机变量 X_k 和 $T(\max(k-1, n-k))$是独立的。

9.2-3　写出 RANDOMIZED-SELECT 的一个迭代版本。

9.2-4　假设要用 RANDOMIZED-SELECT 来选择数组 $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$中的最小元素。给出在 RANDOMIZED-SELECT 的最坏情况性能下的一个划分序列。

9.3　最坏情况线性时间的选择

现在来看一个最坏情况运行时间为 $O(n)$ 的选择算法 SELECT。像 RANDOMIZED-SELECT 一样，SELECT 通过对输入数组的递归划分来找出所求元素，但是，该算法的基本思想是要保证对数组的划分是个好的划分。SELECT 采用了取自快速排序的确定性划分算法 PARTITION（见 7.1 节），并作了修改，把划分主元元素作为其参数。

算法 SELECT 通过执行下列步骤来确定一个有 $n>1$ 个元素的输入数组中的第 i 小的元素。（如果 $n=1$，则 SELECT 只是返回它的唯一输入数值来当作第 i 个最小值。）

1）将输入数组的 n 个元素划分为 $\lfloor n/5 \rfloor$ 组，每组 5 个元素，且至多只有一个组由剩下的
189 $n \bmod 5$ 个元素组成。

2）寻找 $\lceil n/5 \rceil$ 个组中每一组的中位数。首先对每组中的元素（至多为 5 个）进行插入排序，然后从排序过的序列中选出中位数。

3）对第 2 步中找出的 $\lceil n/5 \rceil$ 个中位数，递归调用 SELECT 以找出其中位数 x。（如果有偶数个中位数，根据约定，x 是下中位数。）

4）利用修改过的 PARTITION 过程，按中位数的中位数 x 对输入数组进行划分。让 k 比划分低区的元素数目多 1，所以 x 是第 k 小的元素，并且有 $n-k$ 个元素在划分的高区。

5）如果 $i=k$，则返回 x。否则，如果 $i<k$，则在低区递归调用 SELECT 以找出第 i 小的元素，如果 $i>k$，则在高区找第 $(i-k)$ 个最小元素。

为分析 SELECT 的运行时间，先来确定大于划分主元元素 x 的元素数的一个下界。图 9-1 给出了一些形象的说明。第 2 步中找出的中位数中，至少有一半大于或等于[⊖] 中位数的中位数 x。因此，
190 在 $\lceil n/5 \rceil$ 个组中，除了那个所含元素可能少于 5 的组和包含 x 的那个组之外，至少有一半的组有 3 个元素大于 x。不计这两个组，大于 x 的元素个数至少为

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geqslant \frac{3n}{10} - 6$$

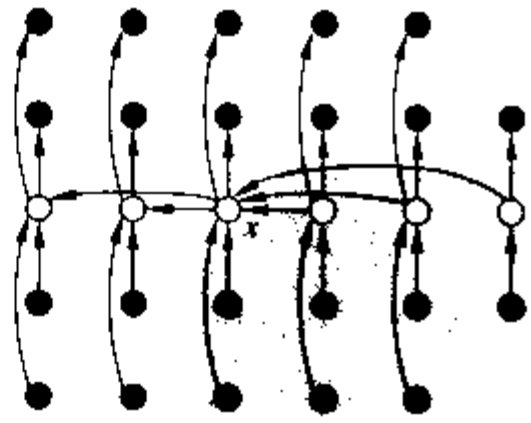


图 9-1　对算法 SELECT 的分析。n 个元素由小圆圈来表示，并且每一个组占一纵列。组的中位数用白色表示，而各中位数的中位数 x 也被标出。（当寻找偶数数目元素的中位数时，使用下中位数。）箭头从较大的元素指向较小的元素，从中可以看出，在 x 的右边，每一个包含 5 个元素的组中都有 3 个元素大于 x；在 x 的左边，每一个包含 5 个元素的组中有 3 个元素小于 x。大于 x 的元素以阴影背景来表示

类似地，小于 x 的元素至少有 $3n/10-6$ 个。因此，在最坏情况下，在第 5 步中最多有 $7n/10+6$ 个元素递归调用 SELECT。

现在就可以来建立一个关于算法 SELECT 的最坏情况运行时间 $T(n)$ 的递归式了。步骤 1、2、4 需要 $O(n)$ 的时间。（步骤 2 对大小为 $O(1)$ 的集合要调用 $O(n)$ 次插入排序。）步骤 3 花时间 $T(\lceil n/5 \rceil)$，步骤 5 所需时间至多为 $T(7n/10+6)$，假设 T 是单调递增的。我们还要作如下假设（这一假设初看起来似乎没有什么动机），即任何等于或少于 140 个元素的输入需要 $O(1)$ 的时间；这个魔力常数 140 的起源很快就会变得清晰了。在此假设下，可以得到递归式：

⊖　由于我们假设这些数字是不相同的，因此在说“大于”或“小于”时，可以不考虑相等的情况。

$$T(n) \leqslant \begin{cases} \Theta(1) & \text{如果 } n \leqslant 140 \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n) & \text{如果 } n > 140 \end{cases}$$

我们使用替换法来证明这个执行时间是线性的。更确切地，我们将证明对某个适当大的常数 c 和所有的 $n>0$，有 $T(n) \leqslant cn$。首先，假设对某个适当大的常数 c 和所有的 $n \leqslant 140$，有 $T(n) \leqslant cn$；如果 c 足够大，这个假设就能成立。同时，还要挑选一个常数 a，使得对所有的 $n>0$，由上述 $O(n)$项(用来描述这个算法运行时间中的非递归部分)所描述的函数由 an 从上方限界。将这个归纳假设代入递归式的右边，得到：

$$\begin{aligned} T(n) &\leqslant c\lceil n/5 \rceil + c(7n/10+6) + an \leqslant cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an = cn + (-cn/10 + 7c + an) \end{aligned}$$

它最多是 cn，如果：

$$-cn/10 + 7c + an \leqslant 0 \tag{9.2}$$

当 $n>70$ 时，不等式(9.2)等价于不等式 $c \geqslant 10a(n/(n-70))$。由于我们假设了 $n \geqslant 140$，有 $n/(n-70) \leqslant 2$，所以选择 $c \geqslant 20a$ 就会满足不等式(9.2)。(注意常数 140 并没有什么特别之处，可以用任何大于 70 的整数来替换它，再相应地选择 c。)因此，SELECT 的最坏情况运行时间是线性的。 191

与比较排序(见 8.1 节)中一样，SELECT 和 RANDOMIZED-SELECT 仅通过元素间的比较来确定它们之间的相对次序。在第 8 章中，我们知道在比较模型中，即使是在平均情况下，排序仍然需要 $\Omega(n \lg n)$的时间(见思考题 8-1)。第 8 章的线性时间排序算法在输入上作了假设。相反地，本章的线性时间选择算法不需要关于输入的任何假设。它们不受下界 $\Omega(n \lg n)$的约束，因为它们没有使用排序就解决了选择问题。

所以，本章中选择算法之所以具有线性运行时间，是因为这些算法没有进行排序；线性时间的行为并不是因为对输入做假设所得到的结果，第 8 章中的排序算法就是这么做的。在比较模型中，即使是在平均情况下，排序仍然需要 $\Omega(n \lg n)$的时间(见思考题 8-1)，所以从渐近意义上来看，本章引言部分提出的排序和索引方法的效率是不高的。

练习

9.3-1 在算法 SELECT 中，输入元素被分为每组 5 个元素。如果它们被分为每组 7 个元素，该算法会仍然以线性时间工作吗？证明如果分成每组 3 个元素，SELECT 无法在线性时间内运行。

9.3-2 分析 SELECT，并证明如果 $n \geqslant 140$，则至少有 $\lceil n/4 \rceil$ 个元素大于中位数的中位数 x，并且至少有 $\lceil n/4 \rceil$ 个元素小于 x。

9.3-3 说明如何才能使快速排序在最坏情况下以 $O(n \lg n)$时间运行。

***9.3-4** 假设对一个含有 n 个元素的集合，某算法只用比较来确定第 i 小的元素。证明：无需另外的比较操作，它也能找到比 i 小的 $i-1$ 个元素和比 i 大的 $n-i$ 个元素。

9.3-5 假设已经有了一个用于求解中位数的“黑箱”子程序，它在最坏情况下需要线性运行时间。写出一个能解决任意顺序统计量的选择问题的线性时间算法。

9.3-6 对一个含有 n 个元素的集合来说，所谓 k 分位数(the kth quantile)，就是能把已排序的集合分成 k 个大小相等的集合的 $k-1$ 个顺序统计量。给出一个能列出某一集合的 k 分位数的 $O(n \lg k)$时间的算法。 192

9.3-7 给出一个 $O(n)$时间的算法，在给定一个有 n 个不同数字的集合 S 以及一个正整数 $k \leqslant n$ 后，它能确定出 S 中最接近其中位数的 k 个数。

9.3-8 设 X[1..n]和 Y[1..n]为两个数组，每个都包含 n 个已排好序的数。给出一个求数组 X

和 Y 中所有 $2n$ 个元素的中位数的、$O(\lg n)$时间的算法。

9.3-9　Olay 教授是一家石油公司的顾问，这家公司正在计划建造一条由东向西的大型管道，它穿过一个有 n 口油井的油田。从每口井中都有一条喷油管道沿最短路径与主管道直接连接(或南或北)，如图 9-2 所示。给定各口井的 x 坐标和 y 坐标，应如何选择主管道的最优位置(使得各喷管长度总和最小的位置)？证明最优位置可在线性时间内确定。

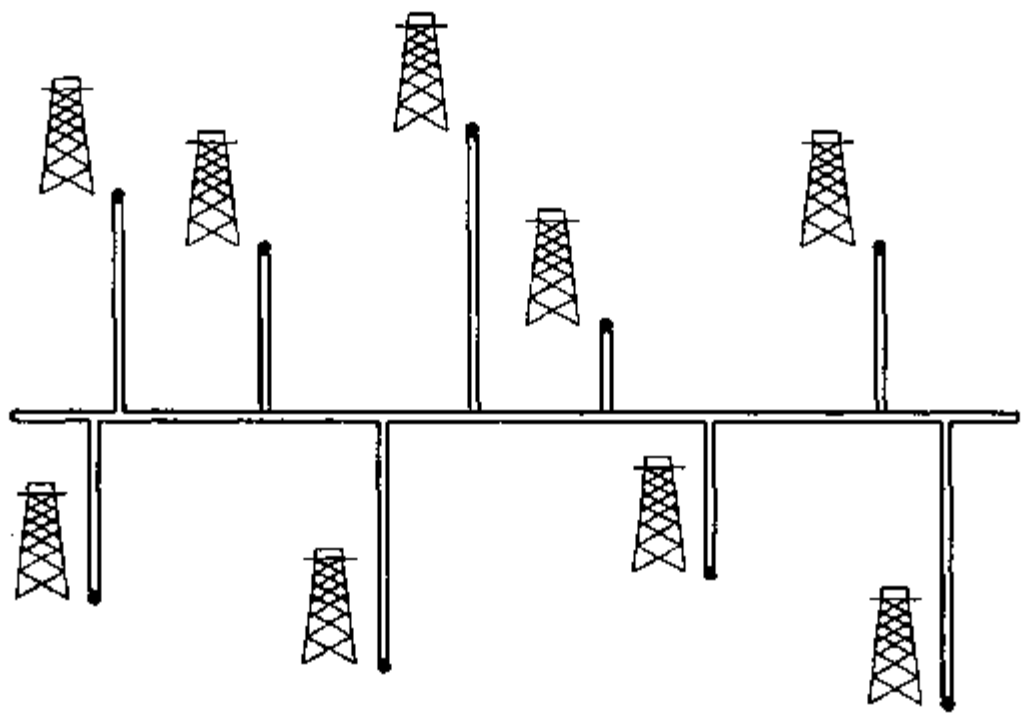


图 9-2　Olay 教授需要确定由东向西管道的位置来让南北向的支线总长度最小

193

思考题

9-1　已排序的 i 个最大数

给定一个含 n 个元素的集合，我们希望能用一个基于比较的算法来找出按顺序排列的 i 个最大元素。请找出能实现下列每一种方法的、具有最佳的渐近最坏情况运行时间的算法，并分析各种方法的运行时间(用 n 和 i 表示)。

a)对输入数排序，并列出 i 个最大的数。

b)对输入数建立一个优先级队列，并调用 EXTRACT-MAX 过程 i 次。

c)利用一个顺序统计量算法来找到第 i 个最大元素，然后划分输入数组，再对 i 个最大数排序。

9-2　带权中位数

对分别具有正的权重 w_1，w_2，…，w_n 且 $\sum_{i=1}^{n} w_i = 1$ 的 n 个不同元素 x_1，x_2，…，x_n，带权(下)中位数是满足如下条件的元素 x_k

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

和

$$\sum_{x_i > x_k} w_i \leqslant \frac{1}{2}$$

a)论证 x_1，x_2，…，x_n 的中位数即各 x_i 的带权中位数，此处权值 $w_i = 1/n$，$i = 1$，2，…，n。

b)说明如何通过排序，在 $O(n \lg n)$的最坏情况时间内求出 n 个元素的带权中位数。

c)说明如何利用一个线性时间的中位数算法(如 9.3 节中的 SELECT)，来在最坏情况 $\Theta(n)$时间内求出 n 个数的带权中位数。

邮局位置问题(post-office location problem)定义如下：已知 n 个点 p_1，p_2，…，p_n 及与它们相联系的权重 w_1，w_2，…，w_n。我们希望能找到一点 p(不一定是输入点中的一个)，使和式 $\sum_{i=1}^{n} w_i d(p, p_i)$ 最小，此处 $d(a, b)$表示点 a 与 b 之间的距离。

d)证明带权中位数是一维邮局位置问题的最佳解决方案，其中所有的点都是实数，并且点 a 与点 b 之间的距离是 $d(a, b)=|a-b|$。

194

e)找出二维邮局位置问题的最佳解答，其中所有的点都是(x, y)坐标对，并且点 $a(x_1, y_1)$与点 $b(x_2, y_2)$之间的距离是 Manhattan 距离：$d(a, b)=|x_1-x_2|+|y_1-y_2|$。

9-3　小型顺序统计量

为从 n 个数字中选出第 i 个顺序统计量，SELECT 在最坏情况下所使用的比较次数 $T(n)$满足 $T(n)=\Theta(n)$，不过隐含在 Θ 记号中的常数相当大。当 i 相对 n 来说较小时，我们可以实现一个不同的程序，它以 SELECT 作为子过程，但最坏情况下所做的比较次数更少。

a)描述一个能用 $U_i(n)$次比较来找出 n 个元素中第 i 小元素的算法，其中

$$U_i(n)=\begin{cases} T(n) & \text{如果 } i \geqslant n/2 \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{否则} \end{cases}$$

(提示：从 $\lfloor n/2 \rfloor$对不相交的两两比较开始，然后对由每对中的较小元素构成的集合作递归处理。)

b)证明：如果 $i<n/2$，则 $U_i(n)=n+O(T(2i)\lg(n/i))$。

c)证明：如果 i 是个小于 $n/2$ 的常数，则 $U_i(n)=n+O(\lg n)$。

d)证明：如果对 $k \geqslant 2$ 有 $i=n/k$，那么 $U_i(n)=n+O(T(2n/k)\lg k)$。

本章注记

最坏情况线性时间寻找中位数的算法是由 Blum，Floyd，Pratt，Rivest 和 Tarjan[43]所发明的。快速的平均时间版本则是由 Hoare[146]提出的。Floyd 与 Rivest[92]设计一个改进的平均时间版本，它递归地从少量元素中选取一个元素，并用这个元素做划分。

确定中位数所需的精确比较次数仍然是未知的。Bent 与 John[38]给出一个寻找中位数的 $2n$ 次比较这一下界。Schonhage，Paterson 和 Pippenger[265]给出了一个 $3n$ 上界。Dor 和 Zwick[79]改进这两个界；他们的上界稍小于 $2.95n$，而下界稍大于 $2n$。Paterson[239]描述这些结果以及其他相关的工作。

195

第三部分　数据结构

引　言

如同在数学中一样，集合也是计算机科学的基础。不过数学中的集合是不变的，而算法所操作的集合却可以随着时间的改变而增大、缩小或产生其他变化。我们称这种集合是动态的。接下来的五章要介绍在计算机上表示和操纵有穷动态集合的一些基本技术。

不同的算法可能需要对集合执行不同的操作。例如，许多算法只要求能将元素插入集合、从集合中删除元素以及测试元素是否属于集合。支持这些操作的动态集合称为字典。另一些算法需要一些更复杂的操作。例如，第6章在介绍堆数据结构时引入了最小优先队列，它支持将一个元素插入一个集合和取出集合中最小元素的操作。实现动态集合的最好方案取决于要支持什么样的集合操作。

动态集合的元素

在动态集合的典型实现中，每个元素都由一个对象来表示，如果有指向对象的指针，就可以对对象的各个域进行检查和处理。(10.3节讨论在对象和指针不是基本数据类型的程序设计环境中如何来实现它们的问题。)某些类型的动态集合假设对象的一个是标识关键的字域。如果关键字互不相同，就可以把此动态集合看成是一个关键字的集合。一个对象中可以有卫星数据，它们随对象的其他域一起移动，但在集合的实现中的其他情况下用不到。对象也可以有由集合运算操作的域；这些域可能包含数据或指向集合中其他对象的指针。

某些动态集合事先假定所有的关键字都取自一个全序集，例如实数或所有按字母顺序排列的单词组成的集合。(全序集满足3.1节中定义的三分性。)全序使我们可以定义集合中的最小元素，或确定比集合中已知元素大的下一个元素等。

动态集合上的操作

动态集合上的操作可分为两类：查询操作，返回有关集合的信息；修改操作，对集合进行修改。下面给出一些典型的操作。任何具体应用常常只是用到其中的一部分。

SEARCH(S, k)

给定一个集合S和关键字值k，返回指向S中一个元素的指针x，使$key[x]=k$，或者，当S中不存在这样的元素时返回NIL。

INSERT(S, x)

是一个修改操作，将由x指向的元素添加到S中去。通常都假设元素x中由集合实现的各个域都已经初始化。

DELETE(S, x)

是一个修改操作，当给定一个指向S中元素的指针x时，将x从S中删除。(注意这个操作

使用一个指向元素 x 的指针，而不是一个关键字的值。）

MINIMUM(S)

是一个全序集 S 上的查询，返回指向 S 中具有最小关键字的元素的指针。

MAXIMUM(S)

是一个全序集 S 上的查询，返回指向 S 中具有最大关键字的元素的指针。

SUCCESSOR(S, x)

是一个查询，给定关键字属于全序集 S 的一个元素 x，返回 S 中比 x 大的下一个元素的指针；当 x 为最大元素时，返回 NIL。

PREDECESSOR(S, x)

是一个查询，给定关键字属于全序集 S 的一个元素 x，返回 S 中比 x 小的前一个元素的指
198 针；当 x 为最小元素时，返回 NIL。

查询 SUCCESSOR 和 PREDECESSOR 常常被推广应用到具有相同关键字的集合上。对包含 n 个关键字的集合，通常的假设是调用 MINIMUM 一次后再调用 $n-1$ 次 SUCCESSOR，就可以按序地枚举出该集合中的所有元素。

执行一个集合操作的时间通常是通过作为参数给出的集合的大小来度量的。例如，第 13 章介绍一种数据结构，它能支持以上列出的每一种操作，且对于大小为 n 的集合的时间为 $O(\lg n)$。

第三部分概述

第 10～14 章描述可以用来实现动态集合的几种数据结构；本书后面将使用其中多种构造解决各类问题的高效算法。另外一种重要的堆数据结构在第 6 章中已经介绍过了。

第 10 章讨论处理简单数据结构的基本问题，如栈、队列、链表以及有根树。这一章还要说明在不把对象和指针作为基本类型支持的程序设计环境中如何来实现它们。这部分内容对接触过几门程序设计课程的读者来说，应该都是熟悉的。

第 11 章介绍散列表，这种结构支持字典操作 INSERT，DELETE 和 SEARCH。在最坏情况下，为做一次散列 SEARCH 操作需要 $\Theta(n)$ 时间；散列表操作的期望时间是 $O(1)$。对散列操作的分析要用到概率论，不过本章的大部分内容并不需要这方面的背景知识。

第 12 章介绍二叉查找树，它支持前面列出的所有动态集合操作。在最坏情况下，在一个有 n 个元素的树上每个操作所需时间为 $\Theta(n)$；但对一棵随机构造的二叉查找树，每个操作的期望时间是 $O(\lg n)$。二叉查找树是很多其他数据结构的基础。

第 13 章介绍红黑树，这是二叉查找树的一种变形。和一般的二叉查找树不同，红黑树始终有着良好的性能：在最坏情况下，它所支持的操作有 $O(\lg n)$ 的运行时间。红黑树是平衡查找树；第 18 章将介绍另一类平衡查找树，称为 B 树。红黑树的机制有点复杂，但不用深入研究其工作机制也能知道其大部分性质。总体来说，浏览一下本章中的代码还是有益处的。

第 14 章介绍如何增强红黑树，使其能支持不在上述基本操作中的一些操作。首先，对红黑树进行增强，使其能支持动态维护一个关键字集合的顺序统计量。然后，对它们做另一种增强，
199 使之支持对实数区间的动态维护。

第 10 章　基本数据结构

在本章中，要讨论如何通过使用了指针的简单数据结构来表示动态集合。有很多复杂的数据结构可以用指针来构造，本章只介绍几种基本的结构，包括栈、队列、链表，以及有根树。另外，还要讨论一种可以利用数组来构造对象和指针的方法。

10.1　栈和队列

栈和队列都是动态集合，在这种结构中，可以用 DELETE 操作去掉的元素是预先规定好的。在栈中，可以去掉的元素是最近插入的那一个：栈实现了一种后进先出(last-in，first-out，缩写为 LIFO)的策略。类似地，在队列中，可以去掉的元素总是在集合中存在时间最长的那一个：队列实现了先进先出(first-in，first-out，缩写为 FIFO)的策略。栈和队列可以用几种方法有效地在计算机上实现，本节介绍如何用数组来实现这两种结构。

栈

作用于栈上的 INSERT 操作称为压入(PUSH)，而无参数的 DELETE 操作常称为弹出(POP)。这两个名字会使人联系到实际生活中的栈，例如在餐馆中用来放盘子的、里面安装了弹簧的栈。在这种栈中，盘子被弹出的次序和它们被压入的次序正好相反，因为在每一时刻，只有最顶上的那只盘子才是可以拿到的。

可以用一个数组 $S[1..n]$来实现一个至多有 n 个元素的栈，如图 10-1 中所示。数组 S 有个属性 $top[S]$，它指向最近插入的元素。由 S 实现的栈包含元素 $S[1..top[S]]$，其中 $S[1]$是栈底元素，$S[top[S]]$是栈顶元素。

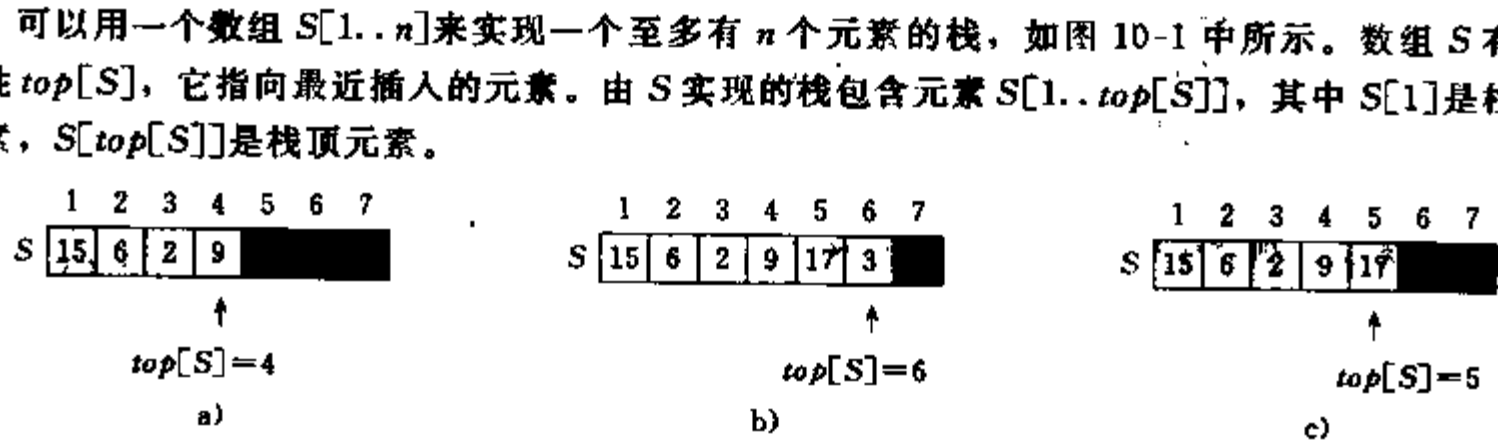


图 10-1　栈 S 的数组实现。栈元素只出现在浅阴影位置。a)栈 S 有四个元素，顶端元素为 9；b)在调用 PUSH(S，17)和 PUSH(S，3)后的栈 S；c)在调用 POP(S)返回最近被压入的元素 3 后的栈 S。虽然元素 3 仍出现于数组中，但它已不在栈中；顶端元素为 17

当 $top[S]=0$ 时，栈中不包含任何元素，因而是空的。要检查一个栈是否为空，可以用查询操作 STACK-EMPTY。如果试图对一个空栈作弹出操作，则称栈下溢。在通常情况下，这是一个错误。如果 $top[S]$超过了 n，则称栈上溢。(在我们的伪代码实现中，不考虑栈的溢出问题。)

有关栈的几种操作可以分别由下面的几行代码实现。

```
STACK-EMPTY(S)
1  if top[S]=0
2     then return TRUE
3     else return FALSE

PUSH(S, x)
1  top[S] ← top[S]+1
2  S[top[S]] ← x
```

```
POP(S)
1  if STACK-EMPTY(S)
2     then error "underflow"
3     else top[S] ← top[S]−1
4          return S[top[S]+1]
```

图 10-1 示出了修正的 PUSH 操作和 POP 操作的结果。以上三种栈操作的时间均为 $O(1)$。

队列

我们把作用于队列上的 INSERT 操作称为入队(ENQUEUE)，把作用于队列上的 DELETE 操作称为出队(DEQUEUE)。像栈操作 POP 一样，DEQUEUE 也不需要取队列的元素作为参数。

201

队列具有 FIFO 性质，看起来就好像在注册办公室里排队的一排人一样。队列有头和尾。当一个元素入队时，将排在队尾，就像刚到的学生站到队尾一样。出队的元素总是队首元素，就像队首等待时间最长的学生一样。(幸运的是我们不用担心元素有插队的情况。)

图 10-2 说明了用一个数组 $Q[1..n]$ 来实现一个至多含 $n-1$ 个元素的队列的方法。队列具有属性 $head[Q]$，它指向队列的头，它的另一个属性为 $tail[Q]$，它指向新元素将会被插入的地方。队列中各元素的位置为 $head[Q]$，$head[Q]+1$，…，$tail[Q]-1$，在最后一个位置要进行"卷绕"，即队列中的 n 个元素排成环形，位置 1 接在位置 n 之后。当 $head[Q]=tail[Q]$ 时，队列为空。刚开始的时候，有 $head[Q]=tail[Q]=1$。当队列为空时，如果试图从中删除一个元素，就会导致队列下溢。当 $head[Q]=tail[Q]+1$ 时，队列是满的。这时，如果试图向其中插入一个元素，就会引起队列上溢。

202

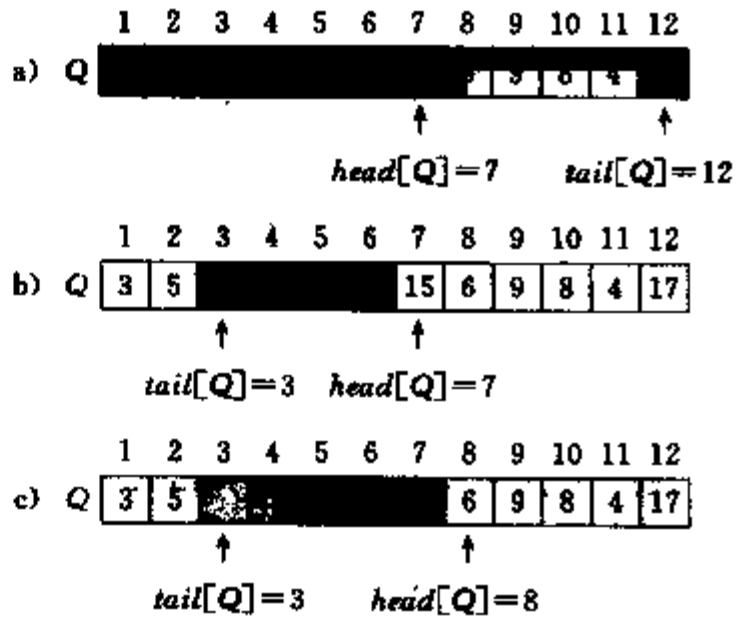


图 10-2　一个用数组 $Q[1..12]$ 实现的队列。队列元素只出现在浅阴影位置。a)该队列有五个元素，在位置 $Q[7..11]$ 上；b)在调用了 ENQUEUE(Q, 17)、ENQUEUE(Q, 3)和 ENQUEUE(Q, 5)后的队列构造；c)在调用 DEQUEUE(Q)返回原队首的关键字 15 后的队列构造。新头的关键字为 6

以下的过程 ENQUEUE 和 DEQUEUE 中，省略了溢出检查部分(练习 10.1-4 要求读者写出用于检测这两种溢出的代码)。

```
ENQUEUE(Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q]=length[Q]
3     then tail[Q] ← 1
4     else tail[Q] ← tail[Q]+1

DEQUEUE(Q)
1  x ← Q[head[Q]]
2  if head[Q]=length[Q]
3     then head[Q] ← 1
4     else head[Q] ← head[Q]+1
5  return x
```

图 10-2 示出了 ENQUEUE 和 DEQUEUE 操作，它们的时间都是 $O(1)$。

练习

10.1-1 仿照图 10-1，说明对一个存储在数组 $S[1..6]$中的、初始为空的栈 S，依次执行 PUSH $(S, 4)$、PUSH$(S, 1)$、PUSH$(S, 3)$、POP(S)、PUSH$(S, 8)$以及 POP(S)操作后的结果。

10.1-2 说明如何用一个数组 $A[1..n]$来实现两个栈，使得两个栈中的元素总数不到 n 时，两者都不会发生上溢。注意 PUSH 和 POP 操作的时间应为 $O(1)$。

10.1-3 仿照图 10-2，说明对一个存储在数组 $Q[1..6]$中的、初始为空的队列 Q，依次执行 ENQUEUE$(Q, 4)$，ENQUEUE$(Q, 1)$，ENQUEUE$(Q, 3)$，DEQUEUE(Q)，ENQUEUE$(Q, 8)$以及 DEQUEUE(Q)操作后的结果。

10.1-4 重写 ENQUEUE 和 DEQUEUE，使之能处理队列的下溢和上溢。 203

10.1-5 栈的插入和删除操作都是在一端进行的，而队列的插入和删除却是在两头进行的。另有一种双端队列(deque)，其两端都可以做插入和删除操作。对于一个用数组构造的双端队列，请写出四个在两端进行插入和删除操作的过程，要求运行时间都为 $O(1)$。

10.1-6 说明如何用两个栈来实现一个队列，并分析有关队列操作的运行时间。

10.1-7 说明如何用两个队列来实现一个栈，并分析有关栈操作的运行时间。

10.2　链表

在链表这种数据结构中，各对象按线性顺序排序。链表与数组不同，数组的线性序是由数组下标决定的，而链表中的顺序是由各对象中的指针所决定的。链表可以用来简单而灵活地表示动态集合，且支持 10.1 节中所列出的所有操作，但效率可能不一定很高。

如图 10-3 所示，双链表 L 的每一个元素都是一个对象。每个对象包含一个关键字域和两个指针域：*next* 和 *prev*。当然，对象中还可能包含一些其他的卫星数据。对链表中的某个元素 x，$next[x]$指向链表中 x 的后继元素，而 $prev[x]$则指向链表中 x 的前驱元素。如果 $prev[x]=$NIL，则元素 x 没有前驱结点，即它是链表的第一个元素，也就是头(head)。如果 $next[x]=$NIL，则元素 x 没有后继结点，即它是链表的最后一个元素，也就是尾(tail)。属性 $head[L]$指向表的第一个元素。如果$head[L]=$NIL，则该链表为空。

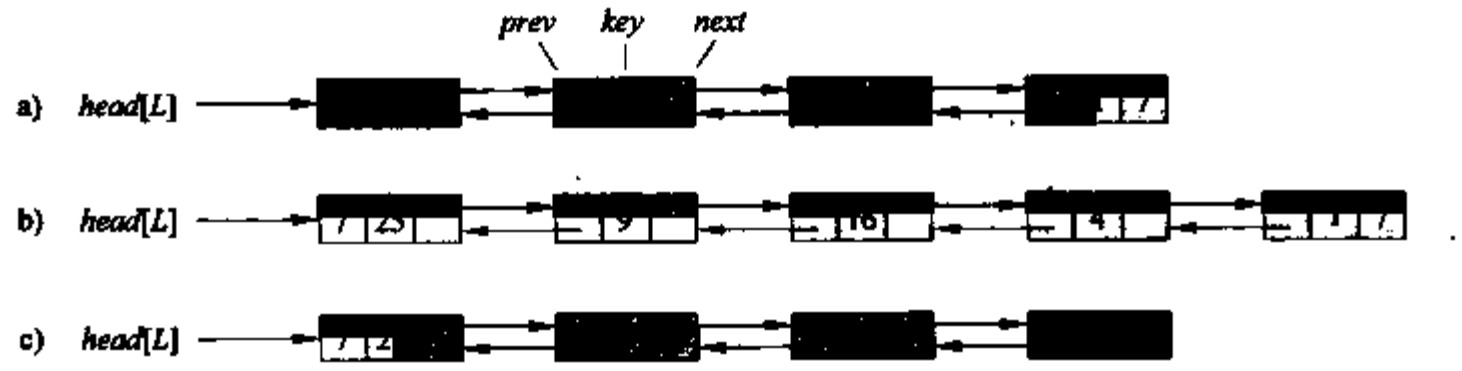


图 10-3　a)表示动态集合{1，4，9，16}的一个双链表 L。链表中每个元素是一个对象，包含关键字域与指向前后对象的指针域(以箭头表示)。表尾的 *next* 域与表头的 *prev* 域都为 NIL，以一个斜杠表示。属性 $head[L]$指向表头；b)执行了 LIST-INSERT(L, x)，其中 $key[x]=25$ 之后，链表以具有关键字 25 的新对象作为新头。这个新对象指向关键字为 9 的旧表头；c)接着调用 LIST-DELETE(L, x)的结果，这时 x 指向关键字为 4 的对象

一个链表可以呈现为好几种形式。它可以是单链接的或双链接的，已排序的或未排序的，环形的或非环形的。如果一个链表是单链接的，则每个元素中没有指向前驱的 *prev* 指针。如果一个链表是已排序的，则该链表的线性顺序就对应着表中各元素关键字的线性顺序。如果链表是未排序的，则元素可以按任意顺序出现。在一个环形链表中，表头元素的 *prev* 指针指向表尾元素，而表尾的 *next* 指针指向表头元素。这样，表中的所有元素就形成了一个环形。在本节余下的部分，假定所处理的链表都是无序的和双向链接的。

链表的搜索操作

下面给出的过程 LIST-SEARCH(L, k)采用了简单的线性查找方法，来找出链表 L 中的第一个具有关键字 k 的元素，并返回指向该元素的指针。如果表中没有包含关键字 k 的对象，则返回 NIL。对图 10-3a 中的链表，调用 LIST-SEARCH(L, 4)返回指向该链表的第三个元素的指针，而调用 LIST-SEARCH(L, 7)就返回 NIL。

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x≠NIL and key[x]≠k
3      do x ← next[x]
4  return x
```

在对一个含 n 个对象的链表进行查找时，过程 LIST-SEARCH 的最坏情况运行时间为 $\Theta(n)$，因为可能需要查找整个链表。

204 ~ 205

链表的插入

给定一个已设置了关键字的新元素 x，过程 LIST-INSERT 将 x 插到链表的前端，如图 10-3b 所示。

```
LIST-INSERT(L, x)
1  next[x] ← head[L]
2  if head[L]≠NIL
3     then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

对一个含 n 个元素的链表，LIST-INSERT 的运行时间为 $O(1)$。

链表的删除

下面的 LIST-DELETE 过程从链表 L 中删除一个元素 x，它需要指向 x 的指针作为参数。它通过对有关元素的指针进行修改，将 x 从表中删除。如果希望删除具有给定关键字的元素，则要先调用 LIST-SEARCH 以得到该元素的指针。

```
LIST-DELETE(L, x)
1  if prev[x]≠NIL
2     then next[prev[x]] ← next[x]
3     else head[L] ← next[x]
4  if next[x]≠NIL
5     then prev[next[x]] ← prev[x]
```

图 10-3c 说明了从链表中删除一个元素的操作。LIST-DELETE 的运行时间为 $O(1)$，但是，如果希望删除一个具有给定关键字的元素，则要先调用 LIST-SEARCH 过程，因而在最坏情况

下的时间为 $\Theta(n)$。

哨兵

对于 LIST-DELETE，如果我们忽视在表头和表尾的边界条件，则该过程的代码会更简单。

```
LIST-DELETE'(L, x)
1  next[prev[x]] ← next[x]
2  prev[next[x]] ← prev[x]
```

哨兵(sentinel)是个哑(dummy)对象，可以简化边界条件。例如，假设有链表 L 和一个对象 $nil[L]$，后者表示 NIL，但也包含和其他元素一样的各个域。现在将链表算法代码中出现的每次对 NIL 的引用，用对哨兵元素 $nil[L]$的引用来代替。这样，就可以将一个一般的双向链表变成一个带哨兵的环形双向链表，哨兵元素 $nil[L]$介于头和尾之间，如图 10-4 所示。域 $next[nil[L]]$指向链表的头，而 $prev[nil[L]]$指向表尾。同样地，表尾的 $next$ 域和表头的 $prev$ 域都指向 $nil[L]$。因为$next[nil[L]]$指向表头，我们可以去掉属性 $head[L]$，把对它的引用换成对 $next[nil[L]]$的引用。一个空链表仅含哨兵元素，因为这时 $next[nil[L]]$和 $prev[nil[L]]$都可以被置成 $nil[L]$。

206

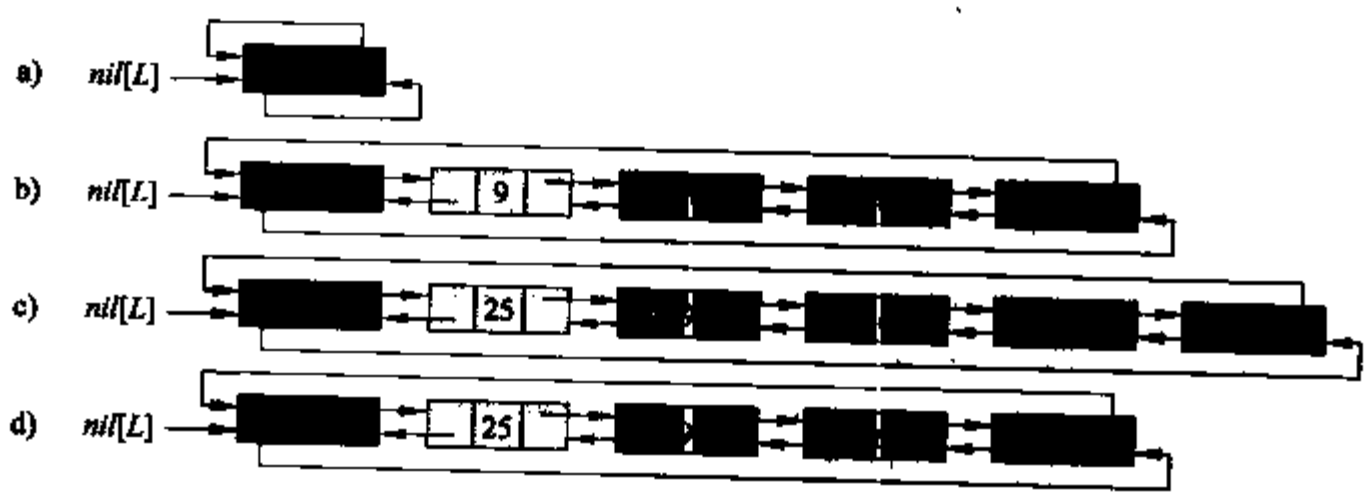


图 10-4　一个带哨兵的环形双向链表．哨兵元素 $nil[L]$介于头和尾之间．属性 $head[L]$不再需要了，因为可以通过 $next[nil[L]]$来访问表头．a)一个空表；b)图 10-3a)中的链表，表头关键字为 9，表尾关键字为 1；c)在执行 LIST-INSERT'(L, x)，将关键字为 25 的新对象插入后的表．新对象成为新表头；d)删除关键字为 1 的对象后的表．新的表尾为具有关键字 4 的对象

LIST-SEARCH 的代码和原来差不多，但对 NIL 和 $head[L]$的引用如上所述地加以改变。

```
LIST-SEARCH'(L, k)
1  x ← next[nil[L]]
2  while x≠nil[L] and key[x]≠k
3        do x ← next[x]
4  return x
```

现在对链表元素的删除用仅含两行代码的过程 LIST-DELETE'即可完成，而元素的插入则用以下的过程进行：

```
LIST-INSERT'(L, x)
1  next[x] ← next[nil[L]]
2  prev[next[nil[L]]] ← x
3  next[nil[L]] ← x
4  prev[x] ← nil[L]
```

207

图 10-4 示出了 LIST-INSERT′和 LIST-DELETE′作用于一个链表上的结果。

哨兵元素基本上不能降低施于有关数据结构上的各种操作的渐近时间界，但它们可以降低常数因子。在循环结构中，使用哨兵的好处主要是使得代码更加简洁，但对于速度则没有什么帮助。例如，采用了哨兵后链表的代码被简化了，但在 LIST-INSERT′和 LIST-DELETE′过程中只节省了 $O(1)$时间。在另一些情况下，采用哨兵则可以使循环部分的代码更加紧凑，因而可以降低运行时间中项 n 或项 n^2 的系数。

要注意不能不加区别地使用哨兵。如果有很多较短的链表，则使用哨兵后就会造成存储的浪费。在本书中，当哨兵真正能简化代码时，我们才加以采用。

练习

10.2-1　动态集合上的操作 INSERT 能否用一个单链表在 $O(1)$时间内实现？对 DELETE 操作呢？

10.2-2　用一单链表 L 实现一个栈，要求 PUSH 和 POP 操作的时间仍为 $O(1)$。

10.2-3　用一单链表 L 实现一个队列，要求 ENQUEUE 和 DEQUEUE 操作的时间仍为 $O(1)$。

10.2-4　如前文所写，LIST-SEARCH′过程的每一次循环迭代都需要做两个测试：一个检查 $x \neq nil[L]$，一个检查 $key[x] \neq k$。说明如何能够在每次迭代中，省去对 $x \neq nil[L]$的检查。

10.2-5　用环形单链表来实现字典操作 INSERT、DELETE 和 SEARCH，并给出它们的运行时间。

10.2-6　动态集合操作 UNION 以两个不相交的集合 S_1 和 S_2 作为输入，输出集合 $S = S_1 \cup S_2$ 包含了 S_1 和 S_2 的所有元素。该操作常常会破坏 S_1 和 S_2。说明应如何选用一种合适的表数据结构，以便支持在 $O(1)$时间内的 UNION 操作。

208

10.2-7　请给出一个 $\Theta(n)$时间的非递归过程，它对含 n 个元素的单链表的链进行逆转。除了链表本身占用的空间外，该过程应仅使用固定量的存储空间。

***10.2-8**　说明如何对每个元素仅用一个指针 $np[x]$(而不是两个指针 $next$ 和 $prev$)来实现双链表。假设所有指针值都是 k 位的整型数，且定义 $np[x] = next[x]$ XOR $prev[x]$，即 $next[x]$和 $prev[x]$的 k 位异或(NIL 用 0 表示)。注意要说明访问表头所需的信息，以及如何实现在该表上的 SEARCH、INSERT 和 DELETE 操作。如何在 $O(1)$时间内实现这样的表。

10.3　指针和对象的实现

有些语言(例如 FORTRAN)中不提供指针与对象数据类型，那么如何实现它们呢？本节将介绍如何在没有显式的指针数据类型时实现链表数据结构的两种方法。我们将使用数组和数组下标来构造对象及指针。

对象的多重数组表示

对一组具有相同域的对象，每一个域都可以用一个数组来表示。图 10-5 说明了如何用三个数组来实现图 10-3a 中的链表。动态集合现有的关键字存储在数组 key 中，而指针存储在数组 $next$ 和 $prev$ 中。对于某一给定的数组下标 x，$key[x]$、$next[x]$和 $prev[x]$就共同表示链表中的一个对象。在这种解释下，一个指针 x 即为指向数组 key、$next$ 和 $prev$ 的共同下标。

在图 10-3a 的链表中，关键字为 4 的对象在关

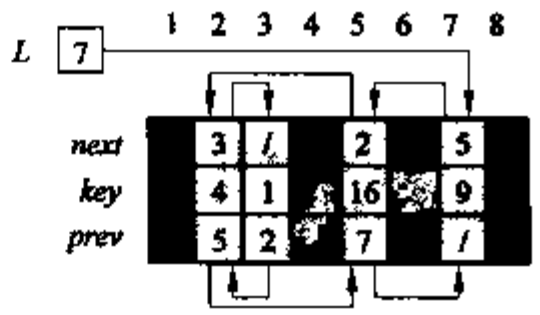


图 10-5　用数组 key，$next$ 和 $prev$ 表示图 10-3a 中的链表。每个垂直的数组切片表示一个单独的对象。存储的指针对应着顶端所示的数组下标；箭头给出了具体的解释。浅阴影对象位置存储了链表元素，而变量 L 则保存了表头元素的下标

键字为 16 的对象后面。对应地，在图 10-5 中，关键字 4 出现在 *key*[2]上，关键字 16 出现在 *key*[5]上，故有 *next*[5]=2，*prev*[2]=5。虽然常数 NIL 出现在表尾的 *next* 域和表头的 *prev* 域中，我们通常用一个不指向数组中任何一个位置的整数(例如 0 或 −1)来表示之。另外，变量 L 中存放了表头元素的下标。

在我们给出的伪代码中，方括号既可以表示数组的下标，又可以表示对象的某个域(属性)。不管怎样，*key*[x]、*next*[x]和 *prev*[x]的含义都与实现是一致的。

对象的单数组表示

计算机存储器中的字是用整数 0 到 $M-1$ 来寻址的，此处 M 是个足够大的整数。在许多程序设计语言中，一个对象占据存储中的一组连续位置，指针即指向某对象所占存储区的第一个位置，后续位置可以通过加上相应的偏移量进行寻址。

对不提供显式指针数据类型的程序设计环境，可以采取同样的策略来实现对象。例如，图 10-6 说明了如何用一个数组 A 来存放图 10-3a 和图 10-5 中的链表。一个对象占用一个连续的子数组$A[j..k]$。对象的每一个域对应着 0 到 $k-j$ 之间的一个偏移量，而对象的指针是下标 j。在图 10-6 中，*key*、*next* 和 *prev* 对应的偏移量为 0、1 和 2。给定指针 i，为了读 *prev*[i]，将指针值 i 与偏移量 2 相加，即读$A[i+2]$。

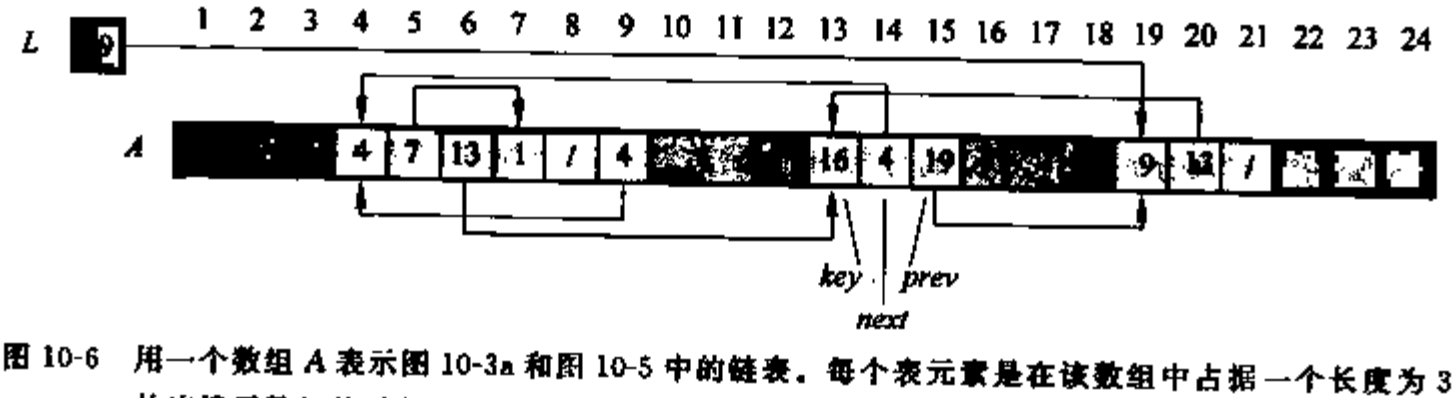


图 10-6　用一个数组 A 表示图 10-3a 和图 10-5 中的链表。每个表元素是在该数组中占据一个长度为 3 的连续子数组的对象。三个域 *key*、*next* 和 *prev* 对应于偏移量 0，1 和 2。指向一个对象的指针是该对象的第一个元素的下标。包含表元素的对象加了浅阴影，箭头示出了表序

这种单数组表示比较灵活，它允许在同一数组中存放不同长度的对象。要操纵一组异构对象要比操纵一组同构对象(各对象具有相同的域)更困难。因为我们考虑的大多数数据结构都是由同构元素所组成的，故用多重数组表示就可以了。

分配和释放对象

为向一个用双链表表示的动态集合中插入一个关键字，需要分配一个指向链表表示中当前未被利用的对象的指针。这样，有必要对链表中未被占用的对象空间加以管理，从而方便分配。在某些系统中，是用废料收集器来确定哪些对象是未用的。有许多应用非常简单，它们可以将未使用的对象返回存储管理器。下面，我们通过一个由多重数组表示的双链表的例子，来讨论对同构对象的分配和释放(或称去配)的问题。

假设多重数组表示中数组长度为 m，且在某一时刻，动态数组包含 $n \leqslant m$ 个元素。这样，n 个对象即表示目前在动态集合中的元素，而另 $m-n$ 个元素是自由的，它们可以用来表示将要插入动态集合中的元素。

我们把自由对象安排成一个单链表，称为自由表。自由表仅用到 *next* 数组，其中存放着表中的 *next* 指针。该自由表的头被置于全局变量 *free* 中。当链表 L 表示的动态集合非空时，自由表将与表 L 交错在一起，如图 10-7 所示。注意每个对象或在表 L 中，或在自由表中，但不能同时在两个表之中。

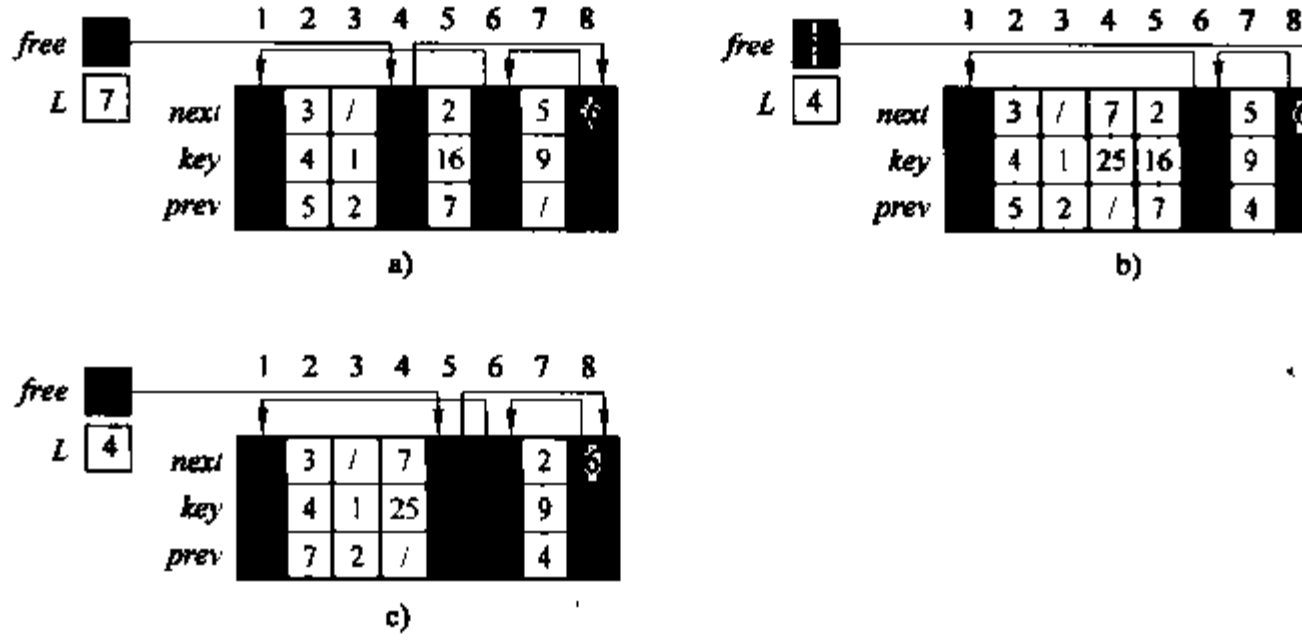


图 10-7　ALLOCATE-OBJECT 和 FREE-OBJECT 过程的效果。a)图10-5 中的表(浅阴影)和一个自由表(重阴影)。箭头示出了自由表的结构；b)调用 ALLOCATE-OBJECT()(它返回下标 4)，置 *key*[4]为 25 和调用 LIST-INSERT(L，4)的结果。新的自由表头为对象 8，它原先为自由表上的 *next*[4]；c)在执行 LIST-DELETE(L，5)后，调用 FREE-OBJECT(5)。对象 5 成为自由表新的表头，它在自由表上的后继为对象 8

自由表是一个栈：下一个分配的对象是最近被释放的那个。我们可以用栈操作 PUSH 和 POP 的表实现方式来分别实现对象的分配和去配过程。在下面的过程中，假设全局变量 *free* 指向自由表的第一个元素。

```
ALLOCATE-OBJECT()
1  if free = NIL
2     then error "out of space"
3     else x ← free
4          free ← next[x]
5          return x

FREE-OBJECT(x)
1  next[x] ← free
2  free ← x
```

开始时，自由表中包含 n 个未分配的对象。当自由表变空时，过程 ALLOCATE-OBJECT 发出出错信号。一般来说，一个自由表可以被几个链表所共用。图 10-8 示出了两个链表和一个自由表通过数组 *key*、*next* 和 *prev* 交叠在一起的情形。

上面两个过程的时间代价为 $O(1)$，因而是两个很实用的过程。要想使它们对任一组同构对象都适用，需要对其进行修改。具体方法是让对象中的任一个域作为自由表中的 *next* 域来使用。

图 10-8　两个链表 L_1(浅阴影)、L_2(重阴影)和一个自由表(黑的)交叠在一起

练习

10.3-1　请画出序列⟨13，4，8，19，5，11⟩存储在以多重数组表示的双链表中的形式。另画出在单数组表示下的形式。

10.3-2 对一组用单数组表示实现的同构对象，写出其过程 ALLOCATE-OBJECT 和 FREE-OBJECT。

10.3-3 在过程 ALLOCATE-OBJECT 和 FREE-OBJECT 的实现中，为什么不需要置或重置对象的 *prev* 域？

10.3-4 我们常常希望一个双链表中的所有元素在存储器中能够紧凑地排列在一起，例如使用多重数组表示中的前 m 个下标位置(在一个分页的虚拟计算机环境中情况就是这样)。假设链表以外没有指向链表元素的指针，请说明应如何实现过程 ALLOCATE-OBJECT 和 FREE-OBJECT，才能使这种表示比较紧凑。(提示：使用栈的数组实现。)

10.3-5 设 L 是一长度为 m 的双链表，存储在长度为 n 的数组 *key*、*next* 和 *prev* 中。假设这些数组由维护双链自由表 F 的两个过程 ALLOCATE-OBJECT 和 FREE-OBJECT 来操纵。进一步假设在数组的 n 个元素中，有 m 个在表 L 中，$n-m$ 个在自由表中。请写出一个过程 COMPACTIFY-LIST(L, F)，对给定的表 L 和自由表 F，移动 L 中的元素，使它们占有数组中的 1，2，…，m 位置，同时调节自由表 F 使之保持正确，并占有数组位置 $m+1, m+2$，…，n。所给出过程的运行时间应该是 $\Theta(m)$，且只能使用固定量的额外空间。请仔细论证你所给出的过程的正确性。

209
~
213

10.4　有根树的表示

前一节中链表的表示方法可以推广至任意同构的数据结构上。在这一节里，我们来讨论用链接数据结构表示有根树的问题。首先要讨论二叉树，然后提出一种适用于结点子女数任意的有根树表示方法。

我们用一个对象来表示树的一个结点。对链表，假设每个结点都有一个关键字域，其余域包括指向其他结点的指针，而且要根据不同类型的树而发生变化。

二叉树

如图 10-9 所示，用域 p、*left* 和 *right* 来存放指向二叉树 T 中的父亲、左儿子和右儿子的指针。如果 $P[x]=$NIL，则 x 为根。如果结点 x 无左儿子，则 $left[x]=$NIL，对右儿子也类似。整个树 T 的根由属性 $root[T]$ 指向。如果 $root[T]=$NIL，则树为空。

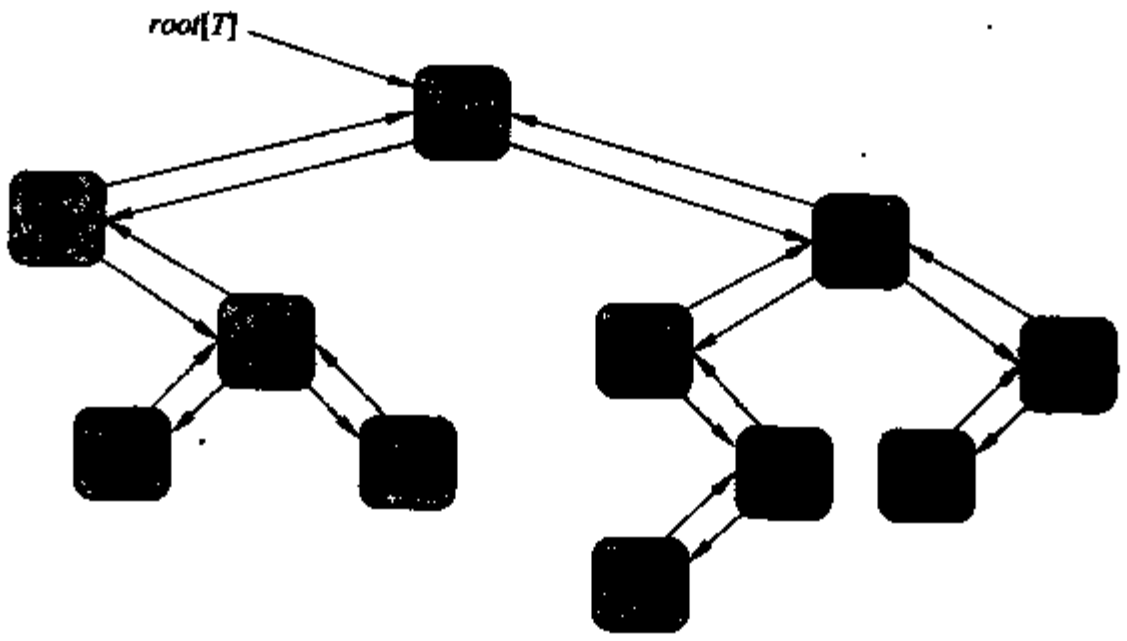


图 10-9　一棵二叉树 T 的表示。每个结点 x 都包含域 $p[x]$(顶端)，$left[x]$(左下)和 $right[x]$(右下)，关键字域没有示出

分支数无限制的有根树

上面二叉树的表示方法可以推广至每个结点的子女数至多为常数 k 的任意种类的树：用 $child_1$，$child_2$，…，$child_k$ 来取代 *left* 和 *right* 域。如果树中结点的子女数是无限制的，那么这种方法就不适用了，因为我们无法事先知道有多少域(多重数组表示的数组)要加以分配。此外，即使结点的子女数 k 以一个很大的常数为界，但多数结点只有少量子女，则会浪费大量的存储空间。

值得庆幸的是，可以用二叉树很方便地表示具有任意子女数的树。这种方法的优点是对任意含 n 个结点的有根树仅用 $O(n)$ 的空间。这种*左孩子、右兄弟*的表示如图 10-10 中所示。像先前一样，每个结点都包含一个父亲指针 p，$root[T]$ 指向树 T 的根。每个结点 x 不再包含指向每个孩子结点的指针，而仅包含两个指针：

1) *left-child*$[x]$ 指向结点 x 的最左孩子。

2) *right-sibling*$[x]$ 指向结点 x 紧右边的兄弟。

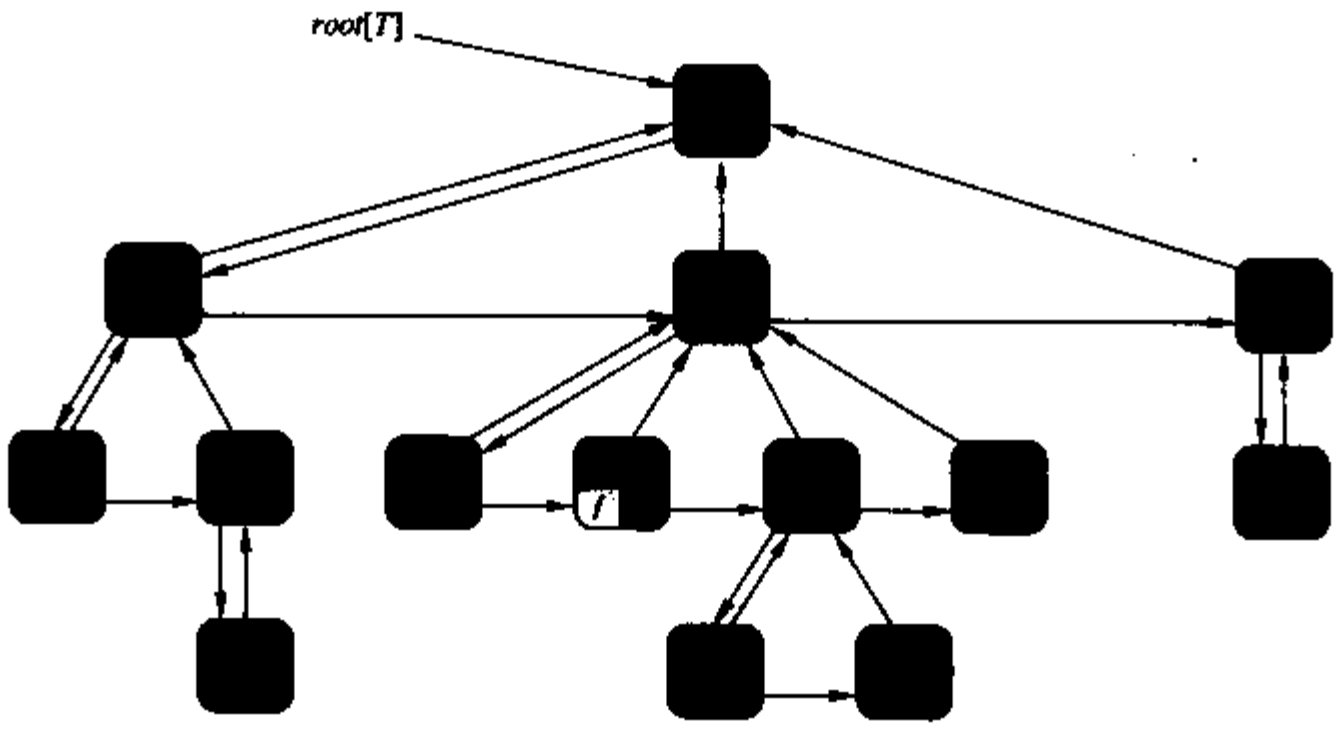


图 10-10　一棵树 T 的左孩子，右兄弟的表示。每个结点 x 都包含域 $p[x]$(顶端)，*left-child*$[x]$(左下)和 *right-sibling*$[x]$(右下)，关键字没有示出

214
~
215

如果 x 没有孩子，则 *left-child*$[x]$ = NIL；如果 x 是其父结点的最右孩子，则 *right-sibling*$[x]$ = NIL。

树的其他表示

有时，可以用另外一些方法来表示有根树。例如，在第 6 章中，我们用了一个数组加上下标的形式来表示基于完全二叉树的堆。将在第 21 章中出现的树只可由叶向根的方向遍历，故只用到父指针，而没有指向子女的指针。另外，还有很多其他的方法，哪一种最好要视具体的应用而定。

练习

10.4-1　画出由下列域表示的、根在下标 6 处的二叉树。

index	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4-2　请写出一个 $O(n)$时间的递归过程，在给定含 n 个结点的二叉树后，它可以将树中每个结点的关键字输出来。

10.4-3　请写出一个 $O(n)$时间的非递归过程，将给定的 n 结点二叉树中每个结点的关键字输出来。可以利用栈作为辅助数据结构。

10.4-4　对于任意的用左孩子、右兄弟表示存储的、含 n 个结点的有根树，写出一个 $O(n)$时间过程来输出每个结点的关键字。

***10.4-5**　写出一个 $O(n)$时间的非递归过程，输出给定的含 n 个结点的二叉树中每个结点的关键字。要求只能使用除树本身以外固定量的额外存储空间，而且在过程中不能修改该树，哪怕是暂时的。

216

***10.4-6**　在任意有根树的左孩子，右兄弟表示中，每个结点有三个指针：*left-child*，*right-sibling* 和 *parent*。从任意结点出发，都可以在常数时间到达其父亲结点；可以在与子女数成线性关系的时间内到达其孩子。说明如何在每个结点内用两个指针和一个布尔值，在与子女数成线性关系的时间内到达其父亲或所有孩子。

思考题

10-1　列表之间的比较

对下表中的四种列表，每一种动态集合操作的渐近最坏情况运行时间是什么？

	未排序的单链表	已排序的单链表	未排序的双链表	已排序的双链表
SEARCH(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

10-2　用链表实现的可合并堆

一个可合并堆支持这样几种操作：MAKE-HEAP(创建一个空的可合并堆)，INSERT，MINIMUM，EXTRACT-MIN 和 UNION[⊖]。说明在下列每一情况中，如何用链表实现可

⊖ 因为我们已经定义了一个可合并堆来支持 MINIMUM 和 EXTRACT-MIN，所以也可以称它为可合并的最小堆。或者，如果它支持 MAXIMUM 和 EXTRACT- MAX 的话，它就是一个可合并的最大堆。

合并堆。尽量使每种操作高效。另分析对动态集合执行的每种操作通过集合规模表示的运
217 行时间。

a)链表是排序的

b)链表是不排序的

c)链表是不排序的，且待合并的各动态集合是不相交的。

10-3　在已排序的紧凑链表中搜索

练习 10.3-4 要求我们回答如何在一个数组的前 n 个位置中紧凑地维护一个含 n 个元素的表。假设所有关键字均不相同，且紧凑表是排序的、即对 $i=1, 2, \cdots, n$，若 $next[i] \neq$ NIL，有 $key[i] < key[next[i]]$。在这些假设下，试说明如下随机算法能在 $O(\sqrt{n})$ 期望时间内完成链表搜索。

```
COMPACT-LIST-SEARCH(L, n, k)
 1  i ← head[L]
 2  while i≠NIL and key[i]<k
 3       do j ← RANDOM(1, n)
 4          if key[i]<key[j]and key[j]≤k
 5             then i ← j
 6                  if key[i]=k
 7                     then return i
 8          i ← next[i]
 9  if i=NIL or key[i]>k
10     then return NIL
11     else return i
```

如果上面代码中的第 3～7 行被略去，则就得到常规的搜索排序链表的算法，其中下标 i，逐次指向表的各个位置。当下标 i 落到表尾或 $key[i] \geq k$ 时，搜索过程即告结束。在后一种情况下，如果 $key[i]=k$，显然我们找到了值为 k 的关键字。如果 $key[i]>k$，那么不可能找到值为 k 的关键字，因此终止查找是正确的。

第 3～7 行试图向前跳到一个随机选择的位置 j 上。如果 $key[j]$ 大于 $key[i]$、但又不大于 k 的话，则这一跳是有好处的。在这样的情况下，j 标记为正常链表搜索中下标 i 会到达的位置。因为链表是紧凑的，所以我们知道，选择 j 为 1 到 n 间的任意一个值时，就指向表中的某个对象，而不会指到自由表中去。

在直接分析 COMPACT-LIST-SEARCH 性能之前，先来分析一个相关算法，分别执行两个循环的 COMPACT-LIST-SEARCH′。这个算法还用参数 t 来控制第一个循环的迭代
218 次数的上界。

```
COMPACT-LIST-SEARCH′(L, n, k, t)
 1  i ← head[L]
 2  for q ← 1 to t
 3       do j ← RANDOM(1, n)
 4          if key[i]<key[j]and key[j]≤k
 5             then i ← j
 6                  if key[i]=k
 7                     then return i
 8  while i≠NIL and key[i]<k
 9       do i ← next[i]
```

```
10  if i=NIL or key[i]>k
11      then return NIL
12      else return i
```

为了比较算法 COMPACT-LIST-SEARCH(L, k)和 COMPACT-LIST-SEARCH′(L, k, t)的执行过程，假设调用 RANDOM(1, n)所返回的整数序列在两个算法中是相同的。

a)假设 COMPACT-LIST-SEARCH(L, k)进行了第 2～8 行循环的 t 次迭代。试说明 COMPACT-LIST-SEARCH′(L, k, t)返回相同的结果，而且 COMPACT-LIST-SEARCH′中的 for 和 while 循环迭代的次数总和至少为 t。

在调用 COMPACT-LIST-SEARCH′(L, k, t)中，设 X_t 为一个随机变量，它刻划了经过第 2～7 行中循环的 t 次迭代后，链表中从位置 i(通过 *next* 指针链)到所需关键字 k 值之间的距离。

b)论证：COMPACT-LIST-SEARCH′(L, k, t)的期望运行时间为 $O(t+\mathrm{E}[X_t])$。

c)证明：$\mathrm{E}[X_t] \leqslant \sum_{r=1}^{n}(1-r/n)^t$。(提示：利用等式(C.24)。)

d)证明：$\sum_{r=0}^{n-1} r^t \leqslant n^{t+1}/(t+1)$。

e)证明：$\mathrm{E}[X_t] \leqslant n/(t+1)$

f)证明：COMPACT-LIST-SEARCH′(L, k, t)以 $O(t+n/t)$的期望时间运行。

g)证明：COMPACT-LIST-SEARCH 以 $O(\sqrt{n})$的期望时间运行。

h)在 COMPACT-LIST-SEARCH 中，为什么要假设每个关键字是不同的？论证当表中有重复的关键字时，随机地向前跳位从渐近意义上来看不一定会有作用。 219

本章注记

Aho，Hopcroft 和 Ullman[6]及 Knuth[182]是非常好的基本数据结构方面的参考文献。许多其他课本都介绍了各种基本的数据结构和它们在各种具体编程语言下的实现。这类课本的例子包括：Goodrich 和 Tamassia[128]，Main[209]，Shaffer[273]和 Weiss[310，312，313]。Gonnet[126]提供了许多数据结构操作性能的实验数据。

栈和队列作为计算机科学中的数据结构，它们的起源已经不得而知。因为在数字计算机引入之前，在数学和书面形式的商业应用中已有相应的记载了。Knuth[182]提到 A. M. Turing 在 1947 年为解决子程序链接而发明了栈。

基于指针的数据结构看上去似乎也是一项“来自民间的发明”。根据 Knuth 的说法，指针很明显地在早期的磁鼓存储器计算机中就已经被使用了。G. M. Hopper 在 1951 年发明的 A-1 语言可以用二叉树来表示代数公式。Knuth 认为，A. Newell，J. C. Shaw 和 H. A. Simon 在 1951 年发明的 IPL-III 语言真正认识到了指针的重要性，并且促进了指针的使用。他们在 1957 年发明的 IPL-III 语言就包含了栈操作。 220

第11章 散 列 表

在很多应用中，都要用到一种动态集合结构，它仅支持 INSERT，SEARCH 和 DELETE 字典操作。例如，计算机程序设计语言的编译程序需要维护一个符号表，其中元素的关键字值为任意字符串，与语言中的标识符对应。实现字典的一种有效数据结构为散列表(hash table)。在最坏情况下，在散列表中，查找一个元素的时间与在链表中查找一个元素的时间相同，在最坏情况下都是 $\Theta(n)$，但在实践中，散列技术的效率是很高的。在一些合理的假设下，在散列表中查找一个元素的期望时间为 $O(1)$。

散列表是普通数组概念的推广。因为可以对数组进行直接寻址，故可以在 $O(1)$时间内访问数组的任意元素。11.1 节进一步讨论直接寻址的问题。如果存储空间允许，我们可以提供一个数组，为每个可能的关键字保留一个位置，就可以应用直接寻址技术。

当实际存储的关键字数比可能的关键字总数较小时，这时采用散列表就会较直接数组寻址更为有效，因为散列表通常采用的数组尺寸与所要存储的关键字数是成比例的。在散列表中，不是直接把关键字用作数组下标，而是根据关键字计算出下标。11.2 节介绍这种技术的主要思想，着重介绍解决“碰撞”(collision)的“链接”(chaining)技术。所谓碰撞，就是指多个关键字映射到同一个数组下标位置。11.3 节介绍如何利用散列函数，根据关键字计算出数组的下标。另外，还将讨论散列技术的几种变形。11.4 节介绍“开放寻址法”(open addressing)，它是处理碰撞的另一种方法。散列是一种极其有效和实用的技术：基本的字典操作只需要 $O(1)$的平均时间。11.5 节解释当待排序的关键字集合是静态的(即当关键字集合一旦存入后不再改变)，“完全散列”(perfect hashing)如何能够在 $O(1)$最坏情况时间内支持关键字查找。

11.1 直接寻址表

当关键字的全域 U 比较小时，直接寻址是一种简单而有效的技术。假设某应用要用到一个动态集合，其中每个元素都有一个取自全域 $U=\{0, 1, \cdots, m-1\}$的关键字，此处 m 是一个不很大的数。另假设没有两个元素具有相同的关键字。

为表示动态集合，我们用一个数组(或称直接寻址表)$T[0..m-1]$，其中每个位置(或称槽)对应全域 U 中的一个关键字。图 11-1 说明这个方法：槽 k 指向集合中一个关键字为 k 的元素。如果该集合中没有关键字为 k 的元素，则 $T[k]=$NIL。

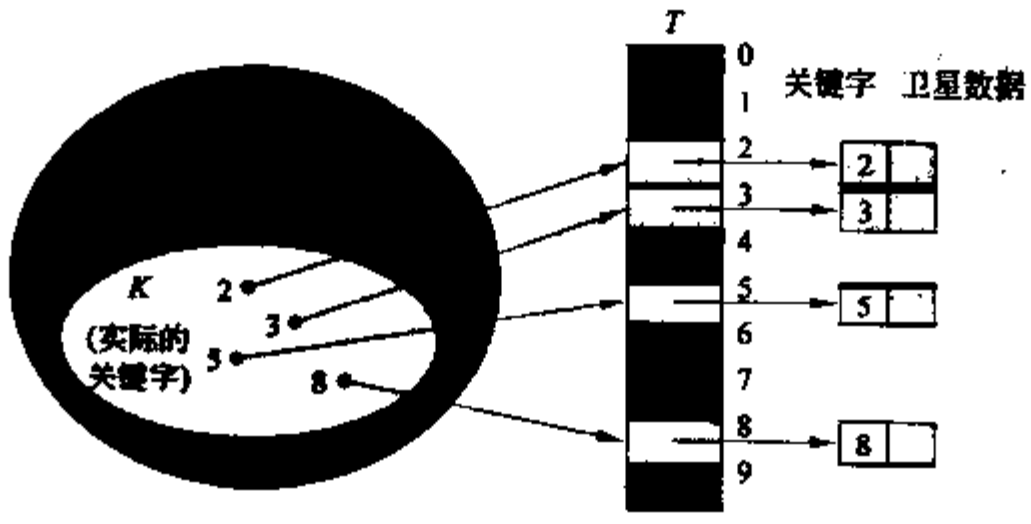


图 11-1 用一个直接寻址表 T 实现动态集合。全域 $U=\{0, 1, \cdots, 9\}$中的每个关键字都对应于表中的一个下标值。由实际关键字构成的集合 $K=\{2, 3, 5, 8\}$决定表中哪些槽包含指向元素的指针。其他带深阴影的槽包含 NIL

几个字典操作实现起来比较简单。

```
DIRECT-ADDRESS-SEARCH(T, k)
    return T[k]

DIRECT-ADDRESS-INSERT(T, x)
    T[key[x]] ← x

DIRECT-ADDRESS-DELETE(T, x)
    T[key[x]] ← NIL
```

上面这些操作执行起来很快，只需 $O(1)$的时间。

对于某些应用，动态集合中的元素可以放在直接寻址表中。亦即不把每个元素的关键字及其卫星数据都放在直接寻址表外部的一个对象中，再由表中某个槽的指针指向该对象，而是直接把对象放在表的槽中，从而节省了空间。此外，通常并不需要存储对象的关键字域，因为如果我们有了一个对象在表中的下标，就可以得到它的关键字。但是，如果不存储关键字，我们就必须有某种办法来确定某个槽是否为空。

练习

11.1-1 考虑由一个长度为 m 的直接寻址表 T 表示的动态集合 S。给出一个查找 S 的最大元素的算法过程。所给的过程在最坏情况下的运行时间是什么？

11.1-2 位向量(bit vector)是一种仅包含 0 和 1 的数组。长度为 m 的位向量所占空间要比包含 m 个指针的数组少得多。请说明如何用一个位向量来表示一个包含不同元素(无卫星数据)的动态集合。字典操作的运行时间应是 $O(1)$。

11.1-3 说明如何实现一个直接寻址表，使各元素的关键字不必都不相同，且各元素可以有卫星数据。所有三种字典操作(INSERT，DELETE 和 SEARCH)的时间应为 $O(1)$。(不要忘记 DELETE 指向要删除的对象的指针变量，而不是关键字。)

***11.1-4** 我们希望通过利用在一个非常大的数组上直接寻址的方式来实现字典。开始时，该数组中可能包含废料，但要对整个数组进行初始化是不实际的，因为该组的规模太大。请给出在大数组上实现直接寻址字典的方案。每个存储的对象占用 $O(1)$空间；操作 SEARCH，INSERT 和 DELETE 的时间为 $O(1)$；对数据结构初始化的时间为 $O(1)$。(提示：可以利用另外一个栈，其大小等于实际存储在字典中的关键字数目，以帮助确定大型数组中某个给定的项是否是有效的。)

222 ~ 223

11.2 散列表

直接寻址技术存在着一个明显的问题：如果域 U 很大，在一台典型计算机的可用内存容量限制下，要在机器中存储大小为 $|U|$ 的一张表 T 就有点不实际，甚至是不可能的了。还有，实际要存储的关键字集合 K 相对 U 来说可能很小，因而分配给 T 的大部分空间都要浪费掉。

当存储在字典中的关键字集合 K 比所有可能的关键字域 U 要小得多时，散列表需要的存储空间要比直接寻址表少很多。特别地，在保持仅需 $O(1)$时间即可在散列表中查找一个元素的好处的情况下，存储要求可以降至 $\Theta(|K|)$。唯一的问题是这个界是针对平均时间的，而对直接寻址来说，它对最坏情况来说也是成立的。

在直接寻址方式下，具有关键字 k 的元素被存放在槽 k 中。在散列方式下，该元素处于 $h(k)$

中，亦即，利用散列函数 h，根据关键字 k 计算出槽的位置。函数 h 将关键字域 U 映射到散列表 $T[0..m-1]$ 的槽位上：

$$h: U \to \{0,1,\cdots,m-1\}$$

这时，可以说一个具有关键字 k 的元素是被散列到槽 $h(k)$ 上，或说 $h(k)$ 是关键字 k 的散列值。图 11-2 给出了形象的说明。采用散列函数的目的就在于缩小需要处理的下标范围，即我们要处理的值就从 $|U|$ 降到 m 了，从而相应地降低了空间开销。

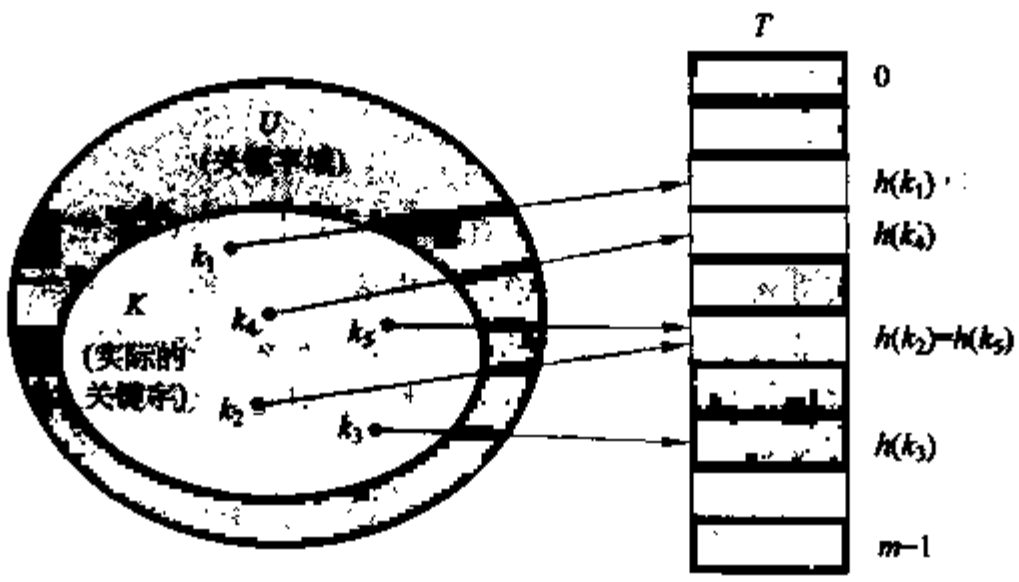


图 11-2　用一个散列函数 h 将关键字映射到散列表空位中。关键字 k_2 和 k_5 映射到同一个空位中，因而发生碰撞

这样做有一个小问题：两个关键字可能映射到同一个槽上。我们将这种情形称为发生了*碰撞*(collision)。幸运的是，能找到有效的方法来解决碰撞。

当然，最理想的解决方法是完全避免碰撞。要做到这一点，可以考虑选用合适的散列函数 h。在选择时有一个主导思想，就是使 h 尽可能地"随机"，从而避免或者至少最小化碰撞。实际上，术语"散列"即体现了这种精神。(当然，一个散列函数 h 必须是确定的，即某一给定的输入 k 应始终产生相同的结果 $h(k)$。)但是，我们知道，$|U|>m$，故必有两个关键字其散列值相同，所以要想完全避免碰撞是不可能的。那么，我们一方面可以通过精心设计的随机散列函数来尽量减少碰撞，另一方面仍需要有解决可能出现的碰撞的办法。

本节余下的部分要介绍一种最简单的碰撞解决技术，称为链接法(chaining)。11.4 节还要介绍另一种碰撞解决方法，称为开放寻址法。

通过链接法解决碰撞

在*链接法*中，把散列到同一槽中的所有元素都放在一个链表中，如图 11-3 所示。槽 j 中有一个指针，它指向由所有散列到 j 的元素构成的链表的头；如果不存在这样的元素，则 j 中为 NIL。

在采用链接法解决碰撞后，散列表 T 上的字典操作就很容易实现。

CHAINED-HASH-INSERT(T, x)
　insert x at the head of list $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)
　search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
　delete x from the list $T[h(key[x])]$

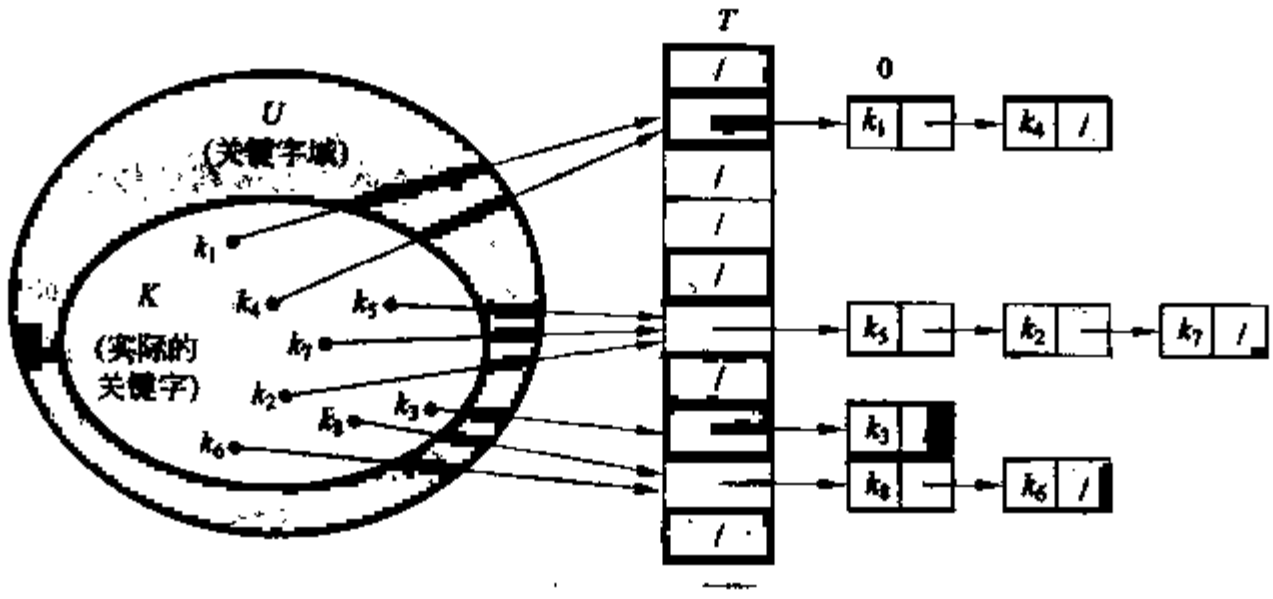


图 11-3　通过链接法解决碰撞。散列表中的每个空位 $T[j]$都包含一个链表，其中所有关键字的散列值均为 j。例如，$h(k_1)=h(k_4)$，还有 $h(k_5)=h(k_2)=h(k_7)$

224
~
225

插入操作的最坏情况运行时间为 $O(1)$。插入过程要快一些，因为假设要插入的元素 x 没有出现在表中；如果需要，在插入前执行搜索，可以检查这个假设(付出额外代价)。查找操作的最坏情况运行时间与表的长度成正比。下面还要做更详细的分析。如果问题中的链表是双向链接的，则删除一个元素 x 的操作可以在 $O(1)$时间内完成(注意，CHAINED-HASH-DELETE 以元素 x 而不是它的关键字 k 作为输入，所以，无需先搜索 x。如果表是单链，用元素 x 而不用关键字 k 作为输入，将不会有很大帮助。我们依然必须寻找表 $T[h(key[x])]$中的 x，所以，通过适当地设置 x 的前趋 *next* 链，把 x 排除在连接之外。在这种情况下，搜索和插入的运行时间基本相同。

对用链接法散列的分析

采用链接法后散列的性能怎样呢？特别地，要查找一个具有给定关键字的元素需要多长时间呢？

给定一个能存放 n 个元素的、具有 m 个槽位的散列表 T，定义 T 的装载因子(load factor)α 为n/m，即一个链中平均存储的元素数。我们的分析以 α 来表达，α 可以小于、等于或大于 1。

用链接法散列的最坏情况性能很差：所有的 n 个关键字都散列到同一个槽中，从而产生出一个长度为 n 的链表。这时，最坏情况下查找的时间为 $\Theta(n)$，再加上计算散列函数的时间，这么一来就和用一个链表来链接所有的元素差不多了。显然，我们并不是因为散列表的最坏情况性能差才用它的。(第 11.5 节中介绍的完全散列能够在关键字集合为静态时，提供比较好的最坏情况性能。)

散列方法的平均性态依赖于所选取的散列函数 h 在一般情况下，将所有的关键字分布在 m 个槽位上的均匀程度。11.3 节中讨论了这些问题，这会儿我们先假定任何元素散列到 m 个槽中每一个的可能性是相同的，且与其他元素已被散列到什么位置上是独立无关的。称这个假设为简单一致散列(simple uniform hashing)。

226

对于 $j=0, 1, \cdots, m-1$，列表 $T[j]$的长度用 n_j 表示，这样有：

$$n = n_0 + n_1 + \cdots + n_{m-1} \tag{11.1}$$

n_j 的平均值为 $E[n_j]= \alpha=n/m$。

假定可以在 $O(1)$时间内计算出散列值 $h(k)$，从而查找具有关键字为 k 的元素的时间线性地依赖于表 $T[h(k)]$的长度 $n_{h(k)}$。先不考虑计算散列函数和寻址槽 $h(k)$的 $O(1)$时间，我们来看看

查找算法期望查找的元素数，即为比较元素的关键字是否为 k 而检查的表 $T[h(k)]$ 中的元素数。分两种情况来考虑。在第一种情况中，查找不成功，表中没有一个元素的关键字为 k。在第二种情况中，查找成功地找到关键字为 k 的元素。

定理 11.1　对一个用链接技术来解决碰撞的散列表，在简单一致散列的假设下，一次不成功查找的期望时间为 $\Theta(1+\alpha)$。

证明：在简单一致散列的假设下，任何尚未被存储在表中的关键字 k 都是等可能地被散列到 m 个槽的任一个之中。因而，当查找一个关键字 k 时，在不成功的情况下，查找的期望时间就是查找至链表 $T[h(k)]$ 末尾的期望时间，这一时间的期望长度为 $E[n_{h(k)}]=\alpha$。于是，一次不成功的查找平均要检查 α 个元素，所需的总时间(包括计算 $h(k)$ 的时间)为 $\Theta(1+\alpha)$。■

对于成功的查找来说，情况略有不同，这是因为每个链表并不是等可能地被查找到的。某个链表被查找到的概率与它所包含的元素数成正比。然而，期望的查找时间仍然是 $\Theta(1+\alpha)$。

定理 11.2　在简单一致散列的假设下，对于用链接技术解决碰撞的散列表，平均情况下一次成功的查找需要 $\Theta(1+\alpha)$ 时间。

证明：假定要查找的关键字是表中存放的 n 个关键字中任何一个的可能性是相同的。在对元素 x 的一次成功的查找中，所检查的元素数比 x 所在的链表中，出现在 x 前面的元素数多 1。在该链表中，出现在 x 之前的元素都是在 x 之后插入的，这是因为新的元素都是在表头插入的。为
227 了确定所查找元素的期望数目，对 x 所在的链表中，在 x 之后插入到表中的期望元素数加 1，再对表中的 n 个元素 x 取平均。设 x_i 表示插入到表中的第 i 个元素，$i=1,2,\cdots,n$，并设 $k_i=key[x_i]$。对关键字 k_i 和 k_j，定义指示器随机变量 $X_{ij}=I\{h(k_i)=h(k_j)\}$。在简单一致散列的假设下，有 $\Pr\{h(k_i)=h(k_j)\}=1/m$，从而根据引理 5.1，有 $E[X_{ij}]=1/m$。于是，在一次成功的查找中，所检查元素的期望数目为

$$
\begin{aligned}
E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) && \text{(根据期望线性)}\\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)\\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) = 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) && \text{(根据式(A.1))}\\
&= 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}
$$

于是，一次成功的查找所需的全部时间(包括计算散列函数的时间)为 $\Theta(2+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$ ■

这一结论说明了什么呢？如果散列表中槽数至少与表中的元素数成正比，则有 $n=O(m)$，从而 $\alpha=n/m=O(m)/m=O(1)$。这说明平均来说，查找操作需要常数量的时间。我们又知道插入操作在最坏情况下需要 $O(1)$ 时间，删除操作最坏情况下需要 $O(1)$ 时间，因而，全部的字典操作平均情况下都可以在 $O(1)$ 时间内完成。

练习

228 **11.2-1**　假设用一个散列函数 h，将 n 个不同的关键字散列到一个长度为 m 的数组 T 中。假定采用的是简单一致散列法，那么期望的碰撞数是多少？更准确地，集合 $\{\{k,l\}:k\neq l$，且 $h(k)=h(l)\}$ 的期望的基是多少？

11.2-2　对于一个利用链接法解决碰撞的散列表，说明将关键字 5、28、19、15、20、33、12、

17、10 插入到该表中的过程。设该表中有 9 个槽位，并设散列函数为 $h(k)=k \bmod 9$。

11.2-3 Marley 教授做了这样一个假设，即如果将链接模式改动一下，使得每个链表都能保持已排序顺序，散列的性能就可以有很大的提高。Marley 教授的改动对成功查找、不成功查找、插入和删除操作的运行时间有什么影响？

11.2-4 说明在散列表内部，如何通过将所有未占用的槽位链接成一个自由链表，来分配和去配元素的存储空间。假定一个槽位可以存储一个标志、一个元素加上一个或两个指针。所有的字典和自由链表操作应具有 $O(1)$ 的期望运行时间。该自由链表需要是是双链表吗？或者，是不是单链表就足够了？

11.2-5 证明：如果 $|U|>nm$，有一个 U 的大小为 n 的子集，它包含了均散列到同一槽位中的关键字，这样对于带链接的散列表，最坏情况下的查找时间为 $\Theta(n)$。

11.3　散列函数

在这一节里，我们要讨论一些有关如何设计出好的散列函数的问题，并介绍三种设计方案。其中两种方案(用除法进行散列和用乘法进行散列)从本质上来看，都是启发式的方法。第三种方案(全域散列)则利用了随机化的技术，来提供可证明的良好性能。

好的散列函数的特点

一个好的散列函数应(近似地)满足简单一致散列的假设：每个关键字都等可能地散列到 m 个槽位的任何一个之中去，并与其他的关键字已被散列到哪一个槽位中无关。不幸的是，一般情况下不太可能检查这一条件是否成立，因为人们很少能知道关键字所符合的概率分布，而各关键字可能并不是完全互相独立的。

有时，我们偶尔也能知道关键字的概率分布。例如，如果已知各关键字都是随机的实数 k，它们独立地、一致地分布于范围 $0 \leqslant k < 1$ 中，那么散列函数：

$$h(k)=\lfloor km \rfloor$$

就能满足简单一致散列这一假设条件。

在实践中，常常可以运用启发式技术来构造好的散列函数。在设计过程中，可以利用有关关键字分布的限制性信息。例如，在一个编译器的符号表中，关键字都是字符串，表示程序中的标识符。在同一个程序中，经常会出现一些很相近的符号，如 pt 和 pts。一个好的散列函数应能最小化将这些相近符号散列到同一个槽中的可能性。

一种好的做法是以独立于数据中可能存在的任何模式的方式导出散列值。例如，“除法散列”(在 11.3.1 节中讨论)用一个特定的质数来除所给的关键字，所得的余数即为该关键字的散列值。假定所选择的质数与关键字分布中的任何模式都是无关的，这种方法常常可以给出很好的结果。

最后，请注意散列函数的某些应用可能会要求比简单一致散列更强的性质。例如，我们可能希望某些很近似的关键字具有截然不同的散列值(当使用 11.4 节中将定义的线性探查技术时，这一性质是特别有用的)。11.3.3 节中将介绍的全域散列(universal hashing)通常能够提供这些性质。

将关键字解释为自然数

多数散列函数都假定关键字域为自然数集 $N=\{0, 1, 2, \cdots\}$。如果所给关键字不是自然数，则必须有一种方法来将它们解释为自然数。例如，一个字符串关键字可以被解释为按适当的基数记号表示的整数。这样，标识符 pt 可以被解释为十进制整数对(112，116)，因为在 ASCII 字符集中，p=112，t=116。然后，按 128 为基数来表示，pt 即为 $(112 \times 128)+116=14\,452$。在

任一给定的应用中，通常都比较容易设计出类似的方法，来将每个关键字解释为一个(可能是很大的)自然数。在后面的内容中，假定所给的关键字都是自然数。

11.3.1　除法散列法

在用来设计散列函数的除法散列法中，通过取 k 除以 m 的余数，来将关键字 k 映射到 m 个
230 槽的某一个中去。亦即，散列函数为：

$$h(k) = k \bmod m$$

例如，如果散列表的大小为 $m=12$，所给关键字为 $k=100$，则 $h(k)=4$。这种方法只要一次除法操作，所以比较快。

当应用除法散列时，要注意 m 的选择。例如，m 不应是 2 的幂，因为如果 $m=2^p$，则 $h(k)$ 就是 k 的 p 个最低位数字。除非我们事先知道，关键字的概率分布使得 k 的各种最低 p 位的排列形式的可能性相同，否则在设计散列函数时，最好考虑关键字的所有位的情况。练习 11.3-3 要求读者证明，当 k 是一个按基数 2^p 解释的字符串时，选 $m=2^p-1$ 可能是个比较糟糕的选择，因为将 k 的各字符进行排列并不会改变其散列值。

可以选作 m 的值常常是与 2 的整数幂不太接近的质数。例如，假设我们要分配一张散列表，并用链接法解决碰撞，表中大约要存放 $n=2000$ 个字符串，每个字符有 8 位。一次不成功的查找大约要检查 3 个元素，但我们并不在意，故分配散列表的大小为 $m=701$。之所以选择 701 这个数，是因为它是个接近 $\alpha=2000/3$、但又不接近 2 的任何幂次的质数。把每个关键字 k 视为一个整数，则我们有散列函数：

$$h(k) = k \bmod 701$$

11.3.2　乘法散列法

构造散列函数的乘法方法包含两个步骤。第一步，用关键字 k 乘上常数 $A(0<A<1)$，并抽取出 kA 的小数部分。然后，用 m 乘以这个值，再取结果的底(floor)。总之，散列函数为

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

其中"$kA \bmod 1$"即 kA 的小数部分，亦即，$kA - \lfloor kA \rfloor$。

乘法方法的一个优点是对 m 的选择没有什么特别的要求，一般选择它为 2 的某个幂次($m=2^p$，p 为某个整数)，这是因为我们可以在大多数计算机上，按如下所示的方法，很方便地实现散列函数。假设某计算机的字长为 w 位，而 k 正好可容于一个字中。限制 A 为形如 $s/2^w$ 的一个分数，其中 s 是一个取自范围 $0<s<2^w$ 中的整数。参照图 11-4，先用 w 位整数 $s=A\cdot 2^w$ 乘上 k，其结果是一个 $2w$ 位的值 $r_1 2^w + r_0$，其中 r_1 为乘积的高位字，r_0 是乘积的低位字。所求的 p 位散列值中，包含了 r_0 的 p 个最高有效位。

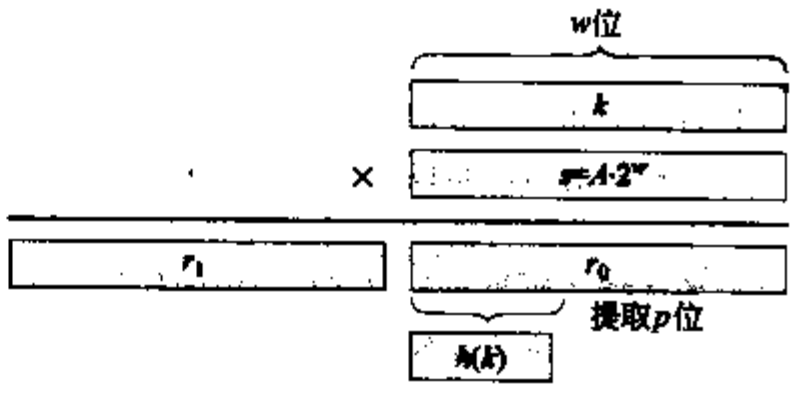


图 11-4　散列的乘法方法。关键字 k 的 w 位表示乘上 w 位值 $s=A\cdot 2^w$。在乘积的低 w 位中，p 个最高位构成了所需的散列值 $h(k)$

虽然这个方法对任何的 A 值都适用，但对某些值效果更好。最佳的选择与待散列的数据的特征有关。Knuth[185]认为

$$A \approx (\sqrt{5}-1)/2 = 0.618\,033\,988\,7\cdots \qquad (11.2)$$

就是个比较理想的值。

下面来看一个例子，假设有 $k=123\,456$，$p=14$，$m=2^{14}=16\,384$，且 $w=32$。根据 Knuth

的建议，取 A 为形如 $s/2^{32}$ 的分数，它与 $(\sqrt{5}-1)/2$ 最为接近，于是 $A=2\,654\,435\,769/2^{32}$。$k\times s=327\,706\,022\,297\,664=(76\,300\times 2^{32})+17\,612\,864$，从而有 $r_1=76\,300$，$r_0=17\,612\,864$。r_0 的 14 个最高有效位产生了散列值 $h(k)=67$。

*11.3.3　全域散列

如果让某个与你作对的人来选择要散列的关键字，那么他会选择全部散列到同一槽中的 n 个关键字，使得平均的检索时间为 $\Theta(n)$。任何一个特定的散列函数都可能出现这种最坏情况性态：唯一有效的改进方法是随机地选择散列函数，使之独立于要存储的关键字。这种方法称作全域散列(universal hashing)，不管对手选择了怎样的关键字，其平均性态都很好。

全域散列的基本思想是在执行开始时，就从一族仔细设计的函数中，随机地选择一个作为散列函数。就像在快速排序中一样，随机化保证了没有哪一种输入会始终导致最坏情况性态。同时，随机化使得即使是对同一个输入，算法在每一次执行时的性态也都不一样。这样就可以确保对于任何输入，算法都具有较好的平均情况性态。再来看看编译器中符号表的例子。我们发现，在全域散列方法中，程序员对标识符的选择就不会一致地导致较差的散列性能了。仅当编译器选择了一个随机的散列函数，使得标识符的散列效果较差时，才会出现较差的性能。但是，出现这种情况的概率很小，并且这一概率对任何相同大小的标识符集来说都是一样的。

231 ~ 232

设 $\mathcal{H}$ 为有限的一组散列函数，它将给定的关键字域 U 映射到 $\{0, 1, \cdots, m-1\}$ 中。这样的一个函数组称为是全域的(universal)，如果对每一对不同的关键字 $k, l\in U$，满足 $h(k)=h(l)$ 的散列函数 $h\in\mathcal{H}$ 的个数至多为 $|\mathcal{H}|/m$。换言之，如果从 $\mathcal{H}$ 中随机地选择一个散列函数，当关键字 $k\neq l$ 时，两者发生碰撞的概率不大于 $1/m$，这也正好是从集合 $\{0, 1, \cdots, m-1\}$ 中随机地、独立地选择 $h(k)$ 和 $h(l)$ 时发生碰撞的概率。

从下面的定理中可以知道，全域散列函数类的平均性态是比较好的。注意 n_i 表示链表 $T[i]$ 的长度。

定理 11.3　如果 h 选自一组全域的散列函数，并用于将 n 个关键字散列到一个大小为 m 的、用链接法解决碰撞的表 T 中。如果关键字 k 不在表中，则 k 被散列至其中的链表的期望长度 $E[n_{h(k)}]$ 至多为 α。如果关键字 k 在表中，则包含关键字 k 的链表的期望长度 $E[n_{h(k)}]$ 至多为 $1+\alpha$。

证明：注意到此处的期望值是有关散列函数的选择的，不依赖于任何有关关键字分布的假设。对于每一对不同的关键字 k 和 l，定义指示器随机变量 $X_{kl}=I\{h(k)=h(l)\}$。因为根据定义，一对关键字发生碰撞的概率为至多为 $1/m$，则有 $\Pr\{h(k)=h(l)\}\leqslant 1/m$，从而根据引理 5.1 有 $\mathrm{E}[X_{kl}]\leqslant 1/m$。

接下来，对于每一个关键字 k，定义一个随机变量 Y_k，它等于非 k 的、与 k 散列到同一槽位中的其他关键字的数目。于是有：

$$Y_k=\sum_{\substack{l\in T\\ l\neq k}}X_{kl}$$

从而有：

$$\mathrm{E}[Y_k]=\mathrm{E}\left[\sum_{\substack{l\in T\\ l\neq k}}X_{kl}\right]=\sum_{\substack{l\in T\\ l\neq k}}\mathrm{E}[X_{kl}]\qquad\text{(根据期望线性)}$$

$$\leqslant\sum_{\substack{l\in T\\ l\neq k}}\frac{1}{m}$$

证明的余下部分与关键字 k 是否在表 T 中有关：

233

- 如果 $k\notin T$，则 $n_{h(k)}=Y_k$，并且 $|\{l:\ l\in T\text{ 且 }l\neq k\}|=n$。于是，$\mathrm{E}[n_{h(k)}]=\mathrm{E}[Y_k]\leqslant$

$n/m=\alpha$。

- 如果 $k\in T$，则由于关键字 k 出现在链表 $T[h(k)]$中，且计数 Y_k 中并没有包括关键字 k，因而有 $n_{h(k)}=Y_k+1$，并且有 $|\{l: l\in T \text{且} l\neq k\}|=n-1$。于是，$E[n_{h(k)}]=E[Y_k]+1\leqslant(n-1)/m+1=1+\alpha-1/m<1+\alpha$。 ■

下面的推论说明了全域散列的确达到了应有的效果：现在，一个对手已无法通过选择一个操作序列来强制达到最坏情况运行时间了。通过在运行时刻巧妙地随机选择散列函数，就可以确保每一个操作序列都可以被处理成具有良好的期望运行时间。

推论 11.4　对于一个具有 m 个槽位的表，利用全域散列和链接法解决碰撞，需要 $\Theta(n)$的期望时间来处理任何包含了 n 个 INSERT、SEARCH 和 DELETE 操作的操作序列，该序列中包含了 $O(m)$个 INSERT 操作。

证明：由于插入操作的数目为 $O(m)$，有 $n=O(m)$，从而有 $\alpha=O(1)$。INSERT 操作和 DELETE 操作需要常量时间，根据定理 11.3，每一个 INSERT 操作的期望时间为 $O(1)$。于是，根据期望值的线性性质，整个操作序列的期望时间为 $O(n)$。 ■

设计一个全域散列函数类

很容易设计出一个全域散列函数类，这一点只需一点点数论方面的知识即可加以证明。读者如果对数论不熟悉的话，可以先看一看第 31 章。

首先，选择一个足够大的质数 p，使得每一个可能的关键字 k 都落在 0 到 $p-1$ 的范围内(包括 0 和 $p-1$)。设 $\mathbf{Z}_p$ 表示集合$\{0, 1, \cdots, p-1\}$，设 $\mathbf{Z}_p^*$ 表示集合$\{1, 2, \cdots, p-1\}$。由于 p 是一个质数，故可以用第 31 章中给出的方法，来解决模 p 的方程。因为我们假定了关键字域的大小大于散列表中的槽位数，故有 $p>m$。

现在，对于任何 $a\in\mathbf{Z}_p^*$ 和任何 $b\in\mathbf{Z}_p$，定义散列函数 $h_{a,b}$。利用一次线性变换，后跟模 p、再模 m 的归约，有：

$$h_{a,b}(k)=((ak+b)\bmod p)\bmod m \tag{11.3}$$

例如，如果有 $p=17$ 和 $m=6$，则有 $h_{3,4}(8)=5$。所有这样的散列函数构成的函数簇为：

234

$$\mathcal{H}_{p,m}=\{h_{a,b}: a\in\mathbf{Z}_p^* \text{ 和 } b\in\mathbf{Z}_p\} \tag{11.4}$$

每一个散列函数 $h_{a,b}$都将 $\mathbf{Z}_p$ 映射到 $\mathbf{Z}_m$。这一类散列函数具有一个良好的性质，即输出范围的大小 m 是任意的，它不一定非得是一个质数不可。第 11.5 节中就将用到这一特性。由于对于 a 来说有$p-1$种选择，对于 b 来说有 p 种选择，因而，$\mathcal{H}_{p,m}$中共有 $p(p-1)$个散列函数。

定理 11.5　由公式(11.3)和公式(11.4)定义的散列函数类$\mathcal{H}_{p,m}$是全域的。

证明：考虑 $\mathbf{Z}_p$ 中的两个不同的关键字 k 和 l，即 $k\neq l$。对于某一给定的散列函数 $h_{a,b}$，设

$$r=(ak+b)\bmod p,\quad s=(al+b)\bmod p$$

首先注意到有 $r\neq s$。这是为什么呢？因为

$$r-s\equiv a(k-l) \pmod p$$

可以导出 $r\neq s$，这是因为 p 为质数，且 a 和$(k-l)$模 p 的结果均不为 0，因而根据定理 31.6，它们的积模 p 后也不为 0。于是，在计算$\mathcal{H}_{p,m}$中任何$h_{a,b}$的过程中，不同的输入 k 和 l 即映射至不同的值 r 和 s(模 p)；在模 p 这一层次上，尚不存在冲突。此外，数对$(a, b)(a\neq 0)$有 $p(p-1)$种可能的选择，其中的每一种都会产生一个不同的数对(r, s)，$r\neq s$，这是因为给定 r 和 s 后，可以解出 a 和 b：

$$a=((r-s)((k-l)^{-1}\bmod p))\bmod p,\quad b=(r-ak)\bmod p$$

其中$((k-l)^{-1} \bmod p)$表示 $k-l$ 模 p 后的倒数。因为仅有 $p(p-1)$种可能的数对$(r, s)(r\neq s)$，

所以在数对$(a, b)(a\neq 0)$与数对$(r, s)(r\neq s)$之间，存在着一种一一对应关系。于是，对任何给定的输入对 k 和 l，如果从 $\mathbf{Z}_p^* \times \mathbf{Z}_p$ 中一致随机地选择(a, b)，所得到的数对(r, s)就等可能地为任何不同的数值对(模 p)。

接下来可以得出这样的结论，即当 r 和 s 为随机选择的不同的值(模 p)时，不同的 k 和 l 发生碰撞的概率等于 $r\equiv s(\bmod m)$。对于某个给定的 r 值，s 的可能取值为余下的 $p-1$ 种，其中满足 $s\neq r$ 且 $s\equiv r(\bmod m)$的 s 值的数目至多为：

$$\begin{aligned}\lceil p/m\rceil - 1 &\leqslant ((p+m-1)/m) - 1 \qquad \text{(根据式(3.7))}\\ &= (p-1)/m\end{aligned}$$

当在模 m 的情况下进行归约时，s 与 r 发生碰撞的概率至多为$((p-1)/m)/(p-1)=1/m$。 235

于是，对于任何不同的值对 k，$l\in\mathbf{Z}_p$，有：

$$\Pr\{h_{a,b}(k) = h_{a,b}(l)\} \leqslant 1/m$$

由此可以看出，$\mathcal{H}_{p,m}$的确是全域的。 ■

练习

11.3-1 假设我们希望查找一个长度为 n 的链表，其中每一个元素都包含一个关键字 k 和一个散列值$h(k)$。每一个关键字都是长字符串。在表中查找具有给定关键字的元素时，如何利用各元素中的散列值？

11.3-2 假设一个长度为 r 的字符串被散列到 m 个槽中，方法是将其视为一个以 128 为基数的数，然后应用除法方法。很容易把数 m 表示为一个 32 位的机器字，但对长度为 r 的字符串，由于它被当作以 128 为基数的数来处理，就要占用若干个机器字。假设应用除法来计算一个字符串的散列值，那么如何才能除了该串本身占用的空间外，只利用常数个机器字？

11.3-3 考虑除法方法的另一种版本，其中 $h(k)=k \bmod m$，$m=2^p-1$，k 为按基数 2^p 解释的字符串。证明：如果串 x 可由串 y 通过其自身的置换排列导出，则 x 和 y 具有相同的散列值。给出一个应用的例子，其中这一特性在散列函数中是不希望出现的。

11.3-4 考虑一个大小为 $m=1000$ 的散列表和一个对应的散列函数 $h(k)=\lfloor m(kA \bmod 1)\rfloor$，$A=(\sqrt{5}-1)/2$。计算关键字 61、62、63、64 和 65 被映射到的位置。

***11.3-5** 定义一个从有限集合 U 到有限集合 B 上的散列函数簇 $\mathcal{H}$ 为 ϵ 全域的，如果对 U 中所有的不同元素对 k 和 l，都有：

$$\Pr\{h(k) = h(l)\} \leqslant \epsilon$$

其中概率是相对从函数簇 $\mathcal{H}$ 中随机抽取的散列函数 h 而言的。证明：一个 ϵ 全域的散列函数簇必定满足：

$$\epsilon \geqslant \frac{1}{|B|} - \frac{1}{|U|}$$ 236

***11.3-6** 设 U 为由取自 $\mathbf{Z}_p$ 中的值构成的 n 元组集合，并设 $B=\mathbf{Z}_p$，其中 p 为质数。当输入为一个取自 U 的输入 n 元组$\langle a_0, a_1, \cdots, a_{n-1}\rangle$时，定义其上的散列函数 $h_b: U\to B(b\in\mathbf{Z}_p)$为：

$$h_b(\langle a_0, a_1, \cdots, a_{n-1}\rangle) = \sum_{j=0}^{n-1} a_j b^j$$

并且，设 $\mathcal{H}=\{h_b: b\in\mathbf{Z}_p\}$。根据练习 11.3-5 中 ϵ 全域的定义，证明 $\mathcal{H}$ 是$((n-1)/p)$全域的。(提示：见练习 31.4-4)

11.4　开放寻址法

在开放寻址法(open addressing)中，所有的元素都存放在散列表里。亦即，每个表项或包含动态集合的一个元素，或包含 NIL。当查找一个元素时，要检查所有的表项，直到找到所需的元素，或者最终发现该元素不在表中。不像在链接法中，这儿没有链表，也没有元素存放在散列表外。在这种方法中，散列表可能会被填满，以致于不能插入任何新的元素、但装载因子 α 是绝不会超过 1 的。

当然，也可以将用作链接的链表存放在散列表未用的槽中(见练习 11.2-4)，但开放寻址法的好处就在于它根本不用指针，而是计算出要存取的各个槽。这样一来，由于不用存储指针而节省了空间，从而可以用同样的空间来提供更多的槽，其潜在的效果就是可以减少碰撞，提高查找速度。

在开放寻址法中，当要插入一个元素时，可以连续地检查(或称探查)散列表的各项，直到找到一个空槽来放置待插入的关键字时为止。检查的顺序不一定是 0，1，…，$m-1$(这种顺序下的查找时间为 $\Theta(n)$)，而是要依赖于待插入的关键字。为了确定要探查哪些槽，我们将散列函数加以扩充，使之包含探查号(从 0 开始)以作为其第二个输入参数。这样，散列函数就变为：

$$h: U \times \{0,1,\cdots,m-1\} \rightarrow \{0,1,\cdots,m-1\}$$

对开放寻址法来说，要求对每一个关键字 k，探查序列

$$\langle h(k,0), h(k,1), \cdots, h(k,m-1) \rangle$$

必须是〈0，1，…，$m-1$〉的一个排列，使得当散列表逐渐填满时，每一个表位最终都可以被视为用来插入新关键字的槽。在下面的伪代码中，假设散列表 T 中的元素为无卫星数据的关键字；关键字 k 与包含关键字 k 的元素完全相同。每个槽或包含一个关键字，或包含 NIL(如果该槽为空的话)。

237

```
HASH-INSERT(T, k)
1  i ← 0
2  repeat j ← h(k, i)
3         if T[j]=NIL
4           then T[j] ← k
5                return j
6           else i ← i+1
7    until i=m
8  error"hash table overflow"
```

查找关键字 k 的算法的探查序列与将 k 插入时的插入算法是一样的。当在查找过程中碰到一个空槽时，查找算法就(非成功地)停止，因为如果 k 确实在表中的话，也应该在该处，而不是探查序列的稍后位置上(之所以这样说，是因为我们假定了关键字不会被删除)。过程 HASH-SEARCH 的输入为一个散列表 T 和一个关键字 k，如果槽 j 中包含关键字 k 则返回 j；如果 k 不在表 T 中，则返回 NIL。

```
HASH-SEARCH(T, k)
1  i ← 0
2  repeat j ← h(k, i)
3         if T[j]=k
4           then return j
5         i ← i+1
```

```
6      until T[j]=NIL or i=m
7   return NIL
```

在开放寻址法中，对散列表元素的删除操作执行起来比较困难。当我们从槽 i 中删除关键字时，不能仅将 NIL 置于其中来标识它为空。如果这样做的话，就会有个问题：在插入某关键字 k 的探查过程中，发现 i 被占用了，则 k 就被插到后面的位置上。在将槽 i 中的关键字删除后，就无法检索关键字 k 了。有一个解决办法，就是在槽 i 中置一个特定的值 DELETED，而不用 NIL。这样就要对过程 HASH-INSERT 作相应的修改，使之将这样的一个槽当作一个空槽，从而仍然可以插入新的元素。对 HASH-SEARCH 无需做什么改动，因为它在搜索时会绕过 DELETED 标识。但是，当我们使用特殊的值 DELETED 时，查找时间就不再依赖于装载因子 α 了。就因为这个原因，在必须删除关键字的应用中，往往采用链接法来解决碰撞。 238

在我们的分析中，作一个一致散列的假设，即假设每个关键字的探查序列是〈0，1，…，$m-1$〉的 $m!$ 种排列中的任一种的可能性是相同的。一致散列将前面定义过的简单一致散列的概念加以了一般化，推广到散列函数的结果不只是一个数，而是一个完整的探查序列的情形。然而，真正的一致散列是很难实现的，在实践中，常常采用它的一些近似方法（如下面要定义的双重散列等）。

有三种技术常用来计算开放寻址法中的探查序列：线性探查，二次探查，以及双重探查。这几种技术都能保证对每个关键字 k，〈$h(k,0)$，$h(k,1)$，…，$h(k,m-1)$〉都是〈0，1，…，$m-1$〉的一个排列。但是，这些技术都不能实现一致散列的假设，因为它们能产生的不同探查序列数都不超过 m^2 个（一致散列要求有 $m!$ 个探查序列）。在这三种技术中，双重散列能产生的探查序列数最多，因而能给出最好的结果。

线性探查

给定一个普通的散列函数 h'：U→{0，1，…，$m-1$}（称为辅助散列函数），线性探查（linear probing）方法采用的散列函数为：

$$h(k,i) = (h'(k)+i) \bmod m, i = 0,1,\cdots,m-1$$

给定一个关键字 k，第一个探查的槽是 $T[h'(k)]$，亦即，由辅助散列函数所给出的槽。接下来探查的是槽 $T[h'(k)+1]$，…，直到槽 $T[m-1]$，然后又卷绕到槽 $T[0]$，$T[1]$，…，直到最后探查槽 $T[h'(k)-1]$。在线性探查方法中，初始探查位置确定了整个序列，故只有 m 种不同的探查序列。

线性探查方法比较容易实现，但它存在着一个问题，称作一次群集（primary clustering）。随着时间的推移，连续被占用的槽不断增加，平均查找时间也随着不断增加。群集现象很容易出现，这是因为当一个空槽前有 i 个满的槽时，该空槽为下一个将被占用槽的概率是 $(i+1)/m$。连续被占用槽的序列将会变得越来越长，因而平均查找时间也会随之增加。

二次探查

二次探查（quadratic probing）采用如下形式的散列函数：

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \qquad (11.5)$$

其中 h' 是一个辅助散列函数，c_1 和 c_2（$\neq 0$）为辅助常数，$i=0$，1，…，$m-1$。初始的探查位置为 $T[h'(k)]$，后续的探查位置要在此基础上加上一个偏移量，该偏移量以二次的方式依赖于探查号 i。这种探查方法的效果要比线性探查好很多，但是，为了能够充分利用散列表，c_1、c_2 和 m 的值要受到限制。思考题 11-3 给出了一种选择这几个参数的方法。此外，如果两个关键字的初始探查位置相同，那么它们的探查序列也是相同的，这是因为 $h(k_1,0) = h(k_2,0)$ 蕴含着 239

$h(k_1, i) = h(k_2, i)$。这一性质可导致一种程度较轻的群集现象，称为二次群集(secondary clustering)。如在线性探查中一样，初始探查决定了整个序列，因此，只有 m 个不同的探查序列被用到了。

双重散列

双重散列是用于开放寻址法的最好方法之一，因为它所产生的排列具有随机选择的排列的许多特性。它采用如下形式的散列函数：

$$h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$$

其中 h_1 和 h_2 为辅助散列函数。初始探查位置为 $T[h_1(k)]$，后续的探查位置在此基础上加上偏移量 $h_2(k)$ 模 m。与线性探查或二次探查不同的是，这里的探查序列以两种方式依赖于关键字 k，因为初始探查位置、偏移量都可能发生变化。图 11-5 给出了一个用双重散列法进行插入的例子。

为能查找整个散列表，值 $h_2(k)$ 要与表的大小 m 互质(见练习 11.4-3)。确保这个条件成立的一种方法是取 m 为 2 的幂，并设计一个总产生奇数的 h_2。另一种方法是取 m 为质数，并设计一个总是产生较 m 小的正整数的 h_2。例如，我们可以取 m 为质数，并取

$$h_1(k) = k \bmod m, \quad h_2(k) = 1 + (k \bmod m')$$

其中 m' 略小于 m(如 $m-1$)。例如，如果 $k=123\,456$，$m=701$，$m'=700$，则有 $h_1(k)=80$，$h_2(k)=257$，可知第一个探查位置为 80，然后检查每第 257 个槽(模 m)，直到找到该关键字，或查遍了所有的槽。

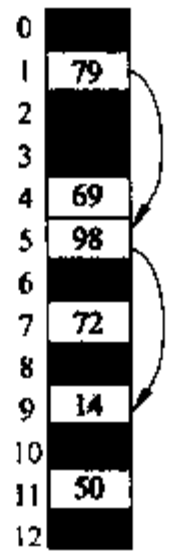


图 11-5　双重散列法下的插入。此处，散列表的大小为 13，$h_1(k) = k \bmod 13$，$h_2(k) = 1 + (k \bmod 11)$。因为 $14 \equiv 1 \pmod{13}$，且 $14 \equiv 3 \pmod{11}$，故在探查了槽 1 和槽 5，并发现它们已被占用后，关键字 14 被插入到空槽 9 中

双重散列法中用了 $\Theta(m^2)$ 种探查序列，而线性探查或二次探查中用了 $\Theta(m)$ 种，故前者是对两种的一种改进。这种改进的原因在于，每一对可能的 $(h_1(k), h_2(k))$ 都产生了一个不同的探查序列。因而，双重散列的性能与“理想的”一致散列的性能看起来就很接近了。

对开放寻址散列的分析

像在分析链接法时一样，对开放寻址法的分析是以散列表的装载因子 $\alpha = n/m$ 来表达的，n 和 m 趋向于无穷。当然，在开放寻址法中，每个槽中至多只有一个元素，因而 $n \leqslant m$，这意味着 $\alpha \leqslant 1$。

现假设采用的是一致散列法。在这种理想的方法中，用于插入或查找每一个关键字 k 的探查序列 $\langle h(k, 0), h(k, 1), \cdots, h(k, m-1)\rangle$ 为 $\langle 0, 1, \cdots, m-1\rangle$ 的任一种排列的可能性是相同的。当然，每一个给定的关键字有唯一固定的探查序列。我们这里想说的是，考虑到关键字空间上的概率分布及散列函数施于这些关键字上的操作，每一种探查序列都是等可能的。

下面就来分析一下在一致散列的假设下，用开放寻址法进行散列时预期的探查数。先来分析一次不成功查找中的探查数。

240

定理 11.6　给定一个装载因子为 $\alpha = n/m < 1$ 的开放寻址散列表，在一次不成功的查找中，期望的探查数至多为 $1/(1-\alpha)$。假设散列是一致的。

证明：在一次不成功的查找中，除了最后一次探查之外，每一次探查都要检查一个被占用

的，但并不包含所求关键字的槽，最后检查的槽是空的。现定义随机变量 X 为在一次不成功的查找中所做的探查数，再定义事件 $A_i(i=1, 2, \cdots)$为进行了第 i 次探查，且探查到的是一个已 241
被占用的槽的事件。那么，事件$\{X \geqslant i\}$即为事件 $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$的交集。下面，我们通过给出 $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$的界来得到 $\Pr\{X \geqslant i\}$的界。根据练习 C.2-6，有：

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}$$

由于共有 n 个元素和 m 个槽，因而 $\Pr\{A_1\} = n/m$。对于 $j>1$，在前 $j-1$ 次探查到的都是已占用槽的前提下，有第 j 次探查且探查到的是已占用槽的概率为$(n-j+1)/(m-j+1)$。之所以会得出这一概率，是因为我们要在$(m-(j-1))$个未探查的槽中，查找余下的$(n-(j-1))$个元素中的某一个。并且，根据一致散列的假设，这一概率为这几个量的比值。注意到 $n<m$ 蕴含着对于所有满足 $0 \leqslant j < m$ 的 j 来说，有$(n-j)/(m-j) \leqslant n/m$，于是，对所有满足 $1 \leqslant i \leqslant m$ 的 i，有：

$$\Pr\{X \geqslant i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \leqslant \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

现在，再利用公式(C.24)来得出期望探查数的界：

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr\{X \geqslant i\} \leqslant \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$ ■

关于以上 $1+\alpha+\alpha^2+\alpha^3+\cdots$的界，可以给出一种比较直观的解释。无论情况怎样，始终至少要做一次探查。当概率大约为 α 时，第一次探查发现的是一个已占用的槽，因而必须做第二次探查。当概率约为 α^2 时，头两次探查的槽是已占用的，因而需要做第三次探查，等等。

如果 α 是一个常数，根据定理 11.6，一次不成功查找的运行时间为 $O(1)$。例如，如果散列表是半满的，在一次不成功的查找中，平均的探查数至多是 $1/(1-0.5)=2$。如果散列表是 90%满的，则平均的探查数至多为 $1/(1-0.9)=10$。 242

根据定理 11.6，几乎直接可以得出 HASH-INSERT 过程的性能。

推论 11.7　平均情况下，向一个装载因子为 α 的开放寻址散列表中插入一个元素时，至多需要做 $1/(1-\alpha)$次探查。假设采用的是一致散列。

证明：只有当表中有空槽时，才可以插入新的元素，故 $\alpha<1$。插入一个关键字要先做一次不成功的查找，然后将该关键字置入第一个遇到的空槽中。所以，期望的探查数为 $1/(1-\alpha)$次。

下面给出一次成功的查找中的期望探查次数。 ■

定理 11.8　给定一个装载因子为 $\alpha<1$ 的开放寻址散列表，一次成功查找中的期望探查数至多为：

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

假定散列是一致的，且表中的每个关键字被查找的可能性是相同的。

证明：查找关键字 k 的探查序列与插入关键字为 k 的元素的探查序列是相同的。根据推论 11.7，如果 k 是第$(i+1)$个插到表中的关键字，则在对 k 的一次查找中，期望的探查次数至多是 $1/(1-i/m)=m/(m-i)$。对散列表中所有 n 个关键字求平均，则得一次成功的查找中平均的探查次数为：

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha}(H_m - H_{m-n})$$

其中 $H_i = \sum_{j=1}^{i} 1/j$ 是第 i 级调和数(如在公式(A.7)中定义的一样)。通过用积分求出和式的界这

243 一技术(如 A.2 节介绍的那样)，可以得到：

$$\frac{1}{\alpha}(H_m - H_{m-n}) = \frac{1}{\alpha}\sum_{k=m-n+1}^{m} 1/k \leqslant \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\mathrm{d}x \qquad (式(A.12))$$

$$= \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

这样就得到了一次成功的查找中期望探查次数的界了。 ■

如果散列表是半满的，则一次成功的查找中，期望的探查数小于 1.387。如果散列表为 90% 满，则期望的探查数小于 2.559。

练习

11.4-1 考虑将关键字 10、22、31、4、15、28、17、88、59 用开放寻址法插入到一个长度为 $m=11$ 的散列表中，主散列函数为 $h(k)=k \bmod m$。说明用线性探查、二次探查($c_1=1$，$c_2=3$)以及双重散列 $h_2(k)=1+(k \bmod (m-1))$将这些关键字插入散列表的结果。

11.4-2 请写出 HASH-DELETE 的伪代码，修改 HASH-INSERT，使之能处理特殊值 DELETED。

***11.4-3** 假设采用双重散列来解决碰撞；亦即，所用的散列函数为 $h(k,i)=(h_1(k)+i\,h_2(k)) \bmod m$。证明：如果对某个关键字 k，m 和 $h_2(k)$有最大公约数 $d \geqslant 1$，则在对关键字 k 的一次不成功的查找中，在回到槽 $h_1(k)$之前，要检查散列表的 $1/d$。于是，当 $d=1$ 时，m 与 $h_2(k)$互质，查找操作可能要检查整个散列表。(提示：见第 31 章)

11.4-4 考虑一个采用了一致散列的开放寻址散列表。给出当装载因子为 3/4 和 7/8 时，在一次
244 不成功查找中期望探查数的上界，以及一次成功查找中期望探查数的上界。

***11.4-5** 考虑一个装载因子为 α 的开放寻址散列表。找出一个非 0 值 α，使得在一次不成功的查找中，期望的探查数等于成功查找中期望探查数的两倍。此处的两个期望探查数上界可以根据定理 11.6 和定理 11.8 得到。

*11.5 完全散列

人们之所以使用散列技术，主要是因为它有着出色的期望性能。其实，当关键字集合是静态的时，散列技术还可以用来获得出色的最坏情况性能。所谓静态就是指一旦各关键字存入表中后，关键字集合就不再变化了。有些应用很自然地有着静态的关键字集合，如一门程序设计语言中的保留字集合，或者一张 CD-ROM 上的文件名集合等。如果某一种散列技术在进行查找时，其最坏情况内存访问次数为 $O(1)$的话，则称其为完全散列(perfect hashing)。

设计完全散列方案的基本想法是比较简单的。我们利用一种两级的散列方案，每一级上都采用全域散列。图 11-6 说明了这一做法。

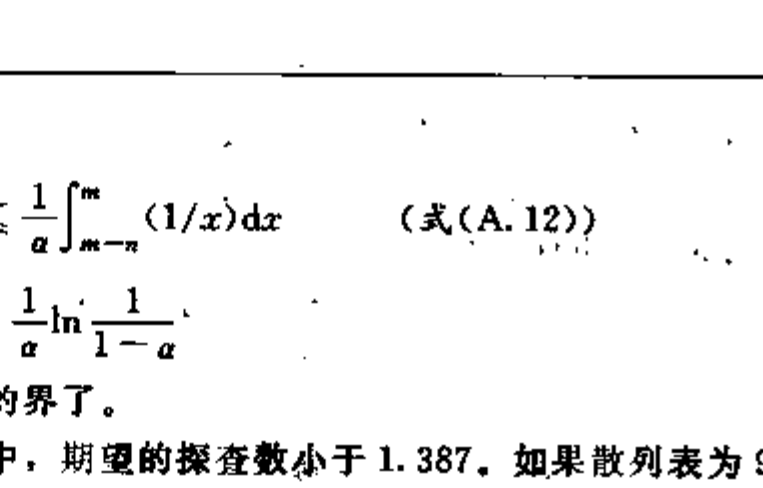


图 11-6　利用完全散列技术来存储关键字集合 $K=\{10, 22, 37, 40, 60, 70, 75\}$。外层的散列函数为 $h(k)=((ak+b) \bmod p) \bmod m$，其中 $a=3$，$b=42$，$p=101$，$m=9$。例如，$h(75)=2$，因此，关键字 75 即散列到表 T 的槽 2 中。二个二次散列表 S_j 中存储了所有散列到槽 j 中的关键字。散列表 S_j 的大小为 m_j，相关的散列函数为 $h_j(k)=((a_jk+b_j) \bmod p) \bmod m_j$。因为 $h_2(75)=1$，故关键字 75 被存储在二次散列表 S_2 的槽 1 中。二次散列表中没有碰撞，因而查找操作在最坏情况下所需的时间为常量

第一级与带链接的散列基本上是一样的：利用从某一全域散列函数簇中仔细选出的一个散列函数 h，将 n 个关键字散列到 m 个槽中。

然而，我们不是对散列到槽 j 中的所有关键字建立一个链表，而是采用了一个较小的二次散列表 S_j，与其相关的散列函数为 h_j。通过仔细地选取散列函数 h_j，可以确保在第二级上不出现碰撞。

但是，为了能真正确保在第二级上不出现碰撞，需要让散列表 S_j 的大小 m_j 为散列到槽 j 中的关键字数 n_j 的平方。m_j 对 n_j 的这种二次依赖关系看上去可能使得总体存储需求很大，后面我们会说明，通过适当地选择第一次散列函数，预期使用的总存储空间仍然为 $O(n)$。

我们要采用的散列函数选自 11.3.3 节中的全域散列函数类。第一次散列函数选自类 $\mathcal{H}_{p,m}$，其中 p 是一个大于任何一个关键字值的质数(见第 11.3.3 节)。那些散列到槽 j 中的关键字通过利用一个从类 $\mathcal{H}_{p,m_j}$ 中选出的散列函数 h_j，被重新散列到一个大小为 m_j 的二次散列表 S_j 中。[⊖]

下面的工作分两步进行。首先，要确定如何才能确保二次散列表中不出现碰撞。其次，要说明期望使用的总体存储空间(即主散列表和所有的二次散列表所占的空间)为 $O(n)$。

定理 11.9　如果利用从一个全域散列函数类中随机选出的散列函数 h，将 n 个关键字存储在一个大小为 $m=n^2$ 的散列表中，那么出现碰撞的概率小于 1/2。

证明：共有 $\binom{n}{2}$ 对关键字可能发生碰撞；如果 h 是从一个全域散列函数类 $\mathcal{H}$ 中随机选出的话，每一对关键字碰撞的概率为 $1/m$。设 X 为一个随机变量，它统计了碰撞的次数。当 $m=n^2$ 时，期望的碰撞次数为：

245
~
246

$$\mathrm{E}[X]=\binom{n}{2}\cdot\frac{1}{n^2}=\frac{n^2-n}{2}\cdot\frac{1}{n^2}<1/2$$

(注意这一分析类似于 5.4.1 节中关于生日悖论的分析。)再运用马尔可夫不等式(Markov's inequality，见 C.29)，$\Pr\{X\geqslant t\}\leqslant\mathrm{E}[X]/t$，将 $t=1$ 代入，即完成证明。■

在定理 11.9 中所描述的情形(即 $m=n^2$)中，对于一个从 $\mathcal{H}$ 中随机选出的散列函数 h，更大的可能是不发生碰撞。给定待散列的包含 n 个关键字的集合 K(注意 K 是静态的)，只需几次随机的尝试，即能比较容易找出一个没有碰撞的散列函数 h。

但是，当 n 比较大时，一个大小为 $m=n^2$ 的散列表还是很大的。因此，我们采用二次散列的方法，并利用定理 11.9 中所叙述的做法，对每个槽中的关键字进行仅一次散列。一个外层的(或称第一级的)散列函数 h 用于将各关键字散列到 $m=n$ 个槽中。那么，如果有 n_j 个关键字被散列到了槽 j 中的话，可以用一个大小为 $m_j=n_j^2$ 的二次散列表 S_j 来提供无碰撞的常量时间查找。

现在再来看看如何确保所用总体存储空间为 $O(n)$的问题。由于第 j 个二次散列表的大小 m_j 以所存储的关键字数 n_j 的平方方式增长，因而存在着这样一种风险，即所需的总体存储空间量可能会很大。

如果第一级表的大小为 $m=n$，则用于存储主散列表、大小为 m_j 的二次散列表、定义取自类 $\mathcal{H}_{p,m_j}$(见 11.3.3 节)的二次散列函数 h_j 的参数 a_j 和 b_j 的存储空间量为 $O(n)$。对于 h_j 而言，当 $n_j=1$ 时，就采用 $a=b=0$。下面要给出的一个定理和一个推论，它们提供了一个关于所有二次散列表的大小加起来后，其期望大小的界。另外还要给出一个推论，它提供了所有二次散列表总大小为超线性的概率的界。

⊖　当 $n_j=m_j=1$ 时，我们并不是真的需要为槽 j 选择一个散列函数；当为这样的槽选择一个散列函数 $h_{a,b}(k)=((ak+b) \bmod p) \bmod m_j$ 时，我们只是用了 $a=b=0$。

定理 11.10　如果利用从某一全域散列函数类中随机选出的散列函数 h，来将 n 个关键字存储到一个大小为 $m=n$ 的散列表中，则有：

$$\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$$

其中，n_j 为散列到槽 j 中的关键字的数目。

247　**证明**：我们从下面的恒等式开始，它对任何非负的整数 a 都成立：

$$a^2 = a + 2\binom{a}{2} \tag{11.6}$$

于是，有：

$$\begin{aligned}
\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] &= \mathrm{E}\left[\sum_{j=0}^{m-1}\left(n_j + 2\binom{n_j}{2}\right)\right] && \text{（由式(11.6)）}\\
&= \mathrm{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{（由线性期望）}\\
&= \mathrm{E}[n] + 2\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{（由式(11.1)）}\\
&= n + 2\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{（因为 } n \text{ 不是随机变量）}
\end{aligned}$$

为了计算和式 $\sum_{j=0}^{m-1}\binom{n_j}{2}$，注意到它只是总的碰撞数。根据全域散列的性质，这一和式的期望值至多为

$$\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}$$

这是因为 $m=n$。于是，有：

$$\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \leqslant n + 2\,\frac{n-1}{2} = 2n-1 < 2n$$ ■

推论 11.11　如果利用从某一全域散列函数类中随机选出的散列函数 h，来将 n 个关键字存储到一个大小为 $m=n$ 的散列表中，并将每个二次散列表的大小置为 $m_j = n_j^2$（$j=0, 1, \cdots, m-1$），则在一个完全散列方案中，存储所有二次散列表所需的存储总量的期望值小于 $2n$。

248　**证明**：因为 $m_j = n_j^2$（$j=0, 1, \cdots, m-1$），根据定理 11.10 有

$$\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right] = \mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n \tag{11.7}$$

证毕。 ■

推论 11.12　如果利用从某一全域散列函数类中随机选出的散列函数 h，来将 n 个关键字存储到一个大小为 $m=n$ 的散列表中，并将每个二次散列表的大小置为 $m_j = n_j^2$（$j=0, 1, \cdots, m-1$），则用于存储所有二次散列表的存储总量超过 $4n$ 的概率小于 $1/2$。

证明：此处再次利用马尔可夫不等式(C.29)，即 $\Pr\{X \geqslant t\} \leqslant \mathrm{E}[X]/t$。这一次是将其作用于不等式(11.7)，并将 $X = \sum_{j=0}^{m-1} m_j$ 和 $t=4n$ 代入：

$$\Pr\left\{\sum_{j=0}^{m-1} m_j \geqslant 4n\right\} \leqslant \frac{\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right]}{4n} < \frac{2n}{4n} = 1/2$$ ■

根据推论 11.12 可以看出，只需尝试从全域散列函数类中随机选出的几个散列函数，即可迅速地找到一个所需存储量较为合理的函数。

练习

11.5-1 假设要将 n 个关键字插入到一个大小为 m、采用了开放寻址法和一致散列技术的散列表中。设 $p(n, m)$ 为没有碰撞发生的概率。证明：$p(n, m) \leqslant e^{-n(n-1)/2m}$。（提示：见公式(3.11)。）论证当 n 超过 $\sqrt{m}$ 时，不发生碰撞的概率迅速趋于 0。

思考题

11-1　最长探查的界

用一个大小为 m 的散列表来存储 n 个数据项目，并且有 $n \leqslant m/2$。采用开放寻址法来解决碰撞问题。

a)假设采用了一致散列，证明对于 $i=1, 2, \cdots, n$，第 i 次插入需要严格多于 k 次探查的概率至多为 2^{-k}。 249

b)证明：对于 $i=1, 2, \cdots, n$，第 i 次插入需要多于 $2\lg n$ 次探查的概率至多是 $1/n^2$。

设随机变量 X_i 表示第 i 次插入所需的探查数。在上面 b)中已证明 $\Pr\{X_i > 2\lg n\} \leqslant 1/n^2$。设随机变量 $X = \max_{1 \leqslant i \leqslant n} X_i$ 表示 n 次插入中所需探查数的最大值。

c)证明：$\Pr\{X > 2\lg n\} \leqslant 1/n$。

d)证明：最长探查序列的期望长度为 $\mathrm{E}[x] = O(\lg n)$。

11-2　链接法中槽大小的界

假设有一个含 n 个槽的散列表，并用链接法来解决碰撞问题。另假设向表中插入 n 个关键字。每个关键字被等可能地散列到每个槽中。设在所有关键字被插入后，M 是各槽中所含关键字数的最大值。读者的任务是证明 M 的期望值 $\mathrm{E}[M]$ 的一个上界为 $O(\lg n/\lg\lg n)$。

a)论证 k 个关键字被散列到某一特定槽中的概率 Q_k 为：

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}$$

b)设 P_k 为 $M=k$ 的概率，也就是包含最多关键字的槽中 k 个关键字的概率。证明：$P_k \leqslant nQ_k$

c)应用斯特林近似公式(3.17)来证明 $Q_k < e^k/k^k$。

d)证明：存在常数 $c>1$，使得 $Q_{k_0} < 1/n^3$ 对 $k_0 = c\lg n/\lg\lg n$ 成立，并总结：$P_k < 1/n^2$ 对 $k \geqslant k_0 = c\lg n/\lg\lg n$ 成立。

e)论证：

$$\mathrm{E}[M] \leqslant \Pr\left\{M > \frac{c\lg n}{\lg\lg n}\right\} \cdot n + \Pr\left\{M \leqslant \frac{c\lg n}{\lg\lg n}\right\} \cdot \frac{c\lg n}{\lg\lg n}$$

并总结：$\mathrm{E}[M] = O(\lg n/\lg\lg n)$。

11-3　二次探查

假设要在一个散列表（表中的各个位置为 0, 1, …, $m-1$）中查找关键字 k，并假设有一个散列函数 h 将关键字空间映射到集合 $\{0, 1, \cdots, m-1\}$ 上，查找方法如下： 250

1)计算值 $i \leftarrow h(k)$，置 $j \leftarrow 0$。

2)探查位置 i，如果找到了所需的关键字 k，或如果这个位置是空的，则结束查找。

3)置 $j \leftarrow (j+1) \bmod m$，$i \leftarrow (i+j) \bmod m$，返回步 2)。

设 m 是 2 的幂。

a)通过给出等式(11.5)中 c_1 和 c_2 的适当值，来证明这个方案是一般的"二次探查"法的一个实例。

b)证明：在最坏情况下，这个算法要检查表中的每一个位置。

11-4　k 全域散列和认证

设 $\mathcal{H}=\{h\}$ 为一个散列函数类，其中每个 h 将关键字域 U 映射到 $\{0, 1, \cdots, m-1\}$ 上。称 $\mathcal{H}$ 是 k 全域的，如果对每个由 k 个不同的关键字 $\langle x^{(1)}, x^{(2)}, \cdots, x^{(k)}\rangle$ 构成的固定序列，以及从 $\mathcal{H}$ 中随机选出的任意 h，序列 $\langle h(x^{(1)}), h(x^{(2)}), \cdots, h(x^{(k)})\rangle$ 是 m^k 个长度为 k 的序列(其元素取自 $\{0, 1, \cdots, m-1\}$)中任一个的可能性相同。

a)证明：如果 $\mathcal{H}$ 是 2 全域的，则它是全域的。

b)设 U 为取自 $\mathbf{Z}_p$ 中的数值的 n 元组集合，并设 $B=\mathbf{Z}_p$，此处 p 为质数。对于任何由取自 $\mathbf{Z}_p$ 中的值所构成的 n 元组 $a=\langle a_0, a_1, \cdots, a_{n-1}\rangle$ 和任何 $b\in\mathbf{Z}_p$，在一个取自 U 的输入 n 元组 $x=\langle x_0, x_1, \cdots, x_{n-1}\rangle$ 上定义散列函数 $h_{a,b}: U\to B$ 为

$$h_{a,b}(x)=\left(\sum_{j=0}^{n-1}a_jx_j+b\right)\bmod p$$

并设 $\mathcal{H}=\{h_{a,b}\}$。论证 $\mathcal{H}$ 是 2 全域的。

c)假设 Alice 和 Bob 悄悄地约定了一个取自 2 全域散列函数簇 $\mathcal{H}$ 中的散列函数 $h_{a,b}$。后来，Alice 通过因特网向 Bob 发送了一个消息 m，其中 $m\in U$。她还通过同时发送一个认证标记 $t=h_{a,b}(m)$ 来向 Bob 认证这一消息，而 Bob 则要检查他所接收到的 (m, t) 对是否满足 $t=h_{a,b}(m)$。假设某一对手半路中截获了 (m, t)，并试图将该值对替换成另一个值对 (m', t') 来欺骗 Bob。论证无论该对手的计算机性能多好，他成功地欺骗 Bob 接受 (m', t') 的概率至多为 $1/p$。

251

本章注记

有关散列函数的分析，Knuth[185]和 Gonnet[126]都是很好的参考。Knuth 认为，是 H. P. Luhn(1953 年)首先提出了散列表这一技术，以及用于解决碰撞的链接方法。在差不多同一时间，G. M. Amdahl 首先提出了开放寻址法的思想。

Carter 和 Wegman[52]于 1979 年引入了全域散列函数类的概念。

Fredman、Komlós 和 Szemerédi[96]针对静态关键字集合(见 11.5 节)，提出了完全散列方案。Dietzfelbinger 等人[73]后来又将这一方法扩展至动态关键字集合上，其处理插入和删除操作的平摊期望时间为 $O(1)$。

252

第 12 章　二叉查找树

查找树（search tree）是一种数据结构，它支持多种动态集合操作，包括 SEARCH，MINIMUM，MAXIMUM，PREDECESSOR，SUCCESSOR，INSERT 以及 DELETE。它既可以用作字典，也可以用作优先队列。

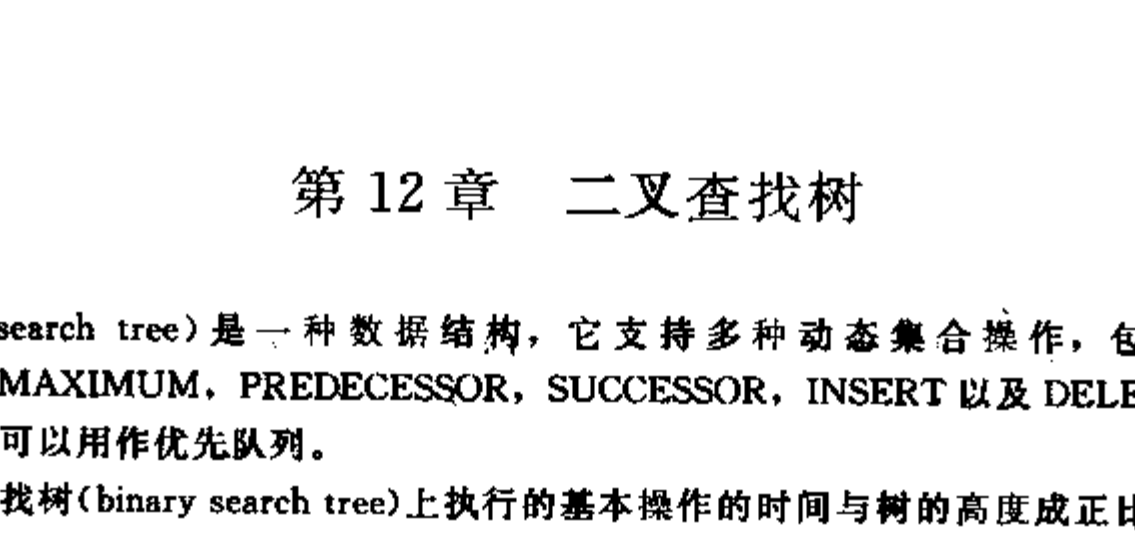
在二叉查找树（binary search tree）上执行的基本操作的时间与树的高度成正比。对于一棵含 n 个结点的完全二叉树，这些操作的最坏情况运行时间为 $\Theta(\lg n)$。但是，如果树是含 n 个结点的线性链，则这些操作的最坏情况运行时间为 $\Theta(n)$。在 12.4 节中可以看到，一棵随机构造的二叉查找树的期望高度为 $O(\lg n)$，从而这种树上基本动态集合操作的平均时间为 $\Theta(\lg n)$。

在实际中，并不总能保证二叉查找树是随机构造成的，但对于有些二叉查找树的变形来说，各基本操作的最坏情况性能却能保证是很好的。第 13 章中给出这样的一种变形，即红黑树，其高度为 $O(\lg n)$。第 18 章介绍 B 树，这种结构对维护随机访问的二级（磁盘）存储器上的数据库特别有效。

在介绍过二叉查找树的基本性质后，随后几节讨论如何遍历二叉查找树以按序输出各个值，如何在树中查找一个值，如何在树中找出最大元素或最小元素，如何找出某一元素的前趋或后继，以及如何对二叉查找树进行插入和删除。树的一些基本数学性质在附录 B 中介绍。

12.1　二叉查找树

如图 12-1 所示，一棵二叉查找树是按二叉树结构来组织的。这样的树可以用链表结构表示，其中每一个结点都是一个对象。结点中除了 *key* 域和卫星数据外，还包含域 *left*，*right* 和 *p*，它们分别指向结点的左儿子、右儿子和父结点。如果某个儿子结点或父结点不存在，则相应域中的值即为 NIL。根结点是树中唯一的父结点域为 NIL 的结点。



a)　　b)

图 12-1　二叉查找树。对任何结点 x，其左子树中的关键字最大不超过 $key[x]$，其右子树中的关键字最小不小于 $key[x]$。不同的二叉查找树可以表示同一组值。在有关查找树的操作中，大部分操作的最坏情况运行时间与树的高度是成正比的。a）一棵包含 6 个结点、高度为 2 的二叉查找树。b）一棵效率较低的二叉查找树，它包含同样的关键字，但高度为 4

二叉查找树中关键字的存储方式总是满足以下的二叉查找树性质：

设 x 为二叉查找树中的一个结点。如果 y 是 x 的左子树中的一个结点，则 $key[y] \leqslant key[x]$。如果 y 是 x 的右子树中的一个结点，则 $key[x] \leqslant key[y]$。

在图 12-1a 中，根结点的关键字为 5，其左子树中的关键字 2，3 和 5 都不大于 5，其右子树中的关键字 7 和 8 都不小于 5。这个性质对树中其他各结点均成立。例如，图 12-1a 中关键字 3 不小于其左子树中的关键字 2，且不大于其右子树中的关键字 5。

根据二叉查找树的性质，可以用一个递归算法按排列顺序输出树中的所有关键字。这种算法称为*中序遍历算法*，因为一子树根的关键字在输出时介于左子树和右子树的关键字之间(类似地，*前序遍历*中根的关键字在其左右子树中的关键字之前输出，而*后序遍历*中根的关键字在其左右子树中的关键字之后输出)。只要调用 INORDER-TREE-WALK(root[T])，就可以输出一棵
254 二叉查找树 T 中的全部元素。

```
INORDER-TREE-WALK(x)
1  if x≠NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

例如，当这个过程应用于图 12-1 中的两棵二叉查找树时，按中序输出关键字序列 2，3，5，5，7，8。要证明本算法的正确性，直接对二叉查找树性质做归纳即可。

遍历一棵含有 n 个结点的二叉查找树所需时间为 $\Theta(n)$，因为在第一次调用遍历过程后，对树中的每个结点，该过程都要被递归调用两次，一次是对左儿子，另一次是对右儿子。关于中序树遍历需要线性时间这一事实，下面的定理给出了一个更为形式化的证明。

定理 12.1　如果 x 是一棵包含 n 个结点的子树的根，则调用 INORDER-TREE-WALK(x)过程的时间为 $\Theta(n)$。

证明：用 $T(n)$表示在一棵包含 n 个结点的子树的根上，调用 INORDER-TREE-WALK 过程所需的时间。对于一棵空子树，INORDER-TREE-WALK 只需很少的一段常量时间(测试 $x\neq$ NIL)，因而有 $T(0)=c$，c 为某一正常数。

对 $n>0$，假设 INORDER-TREE-WALK 是在一个结点 x 上被调用的，该结点的左子树有 k 个结点，其右子树中有 $n-k-1$ 个结点。执行 INORDER-TREE-WALK(x)的时间是 $T(n)=T(k)+T(n-k-1)+d$，d 为某个正常数，它反映了执行 INORDER-TREE-WALK(x)的时间中，扣除花在递归调用上的时间以后的部分。

下面利用替代方法来证明 $T(n)=(c+d)n+c$，从而证明 $T(n)=\Theta(n)$。对 $n=0$，有 $(c+d)\cdot 0+c=c=T(0)$。对 $n>0$，有：

$$\begin{aligned}T(n) &= T(k)+T(n-k-1)+d=((c+d)k+c)+((c+d)(n-k-1)+c)+d\\ &=(c+d)n+c-(c+d)+c+d=(c+d)n+c\end{aligned}$$

255 证毕。 ■

练习

12.1-1　基于关键字集合{1，4，5，10，16，17，21}，画出高度为 2、3、4、5、6 的二叉查找树。

12.1-2　二叉查找树性质与最小堆性质(见 6.1 节)之间有什么区别？能否利用最小堆性质在 $O(n)$时间内，按序输出含有 n 个结点的树中的所有关键字？行的话，解释该怎么做；不行的话，说明原因。

12.1-3　给出一个非递归的中序树遍历算法。(提示：有两种方法，在较容易的方法中，可以采用栈作为辅助数据结构；在较为复杂的方法中，不采用栈结构，但假设可以测试两个指

针是否相等。)

12.1-4　对一棵含有 n 个结点的树，给出能在 $\Theta(n)$ 时间内，完成前序遍历和后序遍历的递归算法。

12.1-5　论证：在比较模型中，最坏情况下排序 n 个元素的时间为 $\Omega(n \lg n)$，则为从任意的 n 个元素中构造出一棵二叉查找树，任何一个基于比较的算法在最坏情况下，都要花 $\Omega(n \lg n)$ 的时间。

12.2　查询二叉查找树

对于二叉查找树，最常见的操作是查找树中的某个关键字。除了 SEARCH 操作外，二叉查找树还能支持诸如 MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 等查询。本节就来讨论这些操作，并说明对高度为 h 的树，它们都可以在 $O(h)$ 时间内完成。

查找

我们用下面的过程在树中查找一个给定的关键字。给定指向树根的指针和关键字 k，过程 TREE-SEARCH 返回指向包含关键字 k 的结点(如果存在的话)的指针；否则，返回 NIL。 256

```
TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH(left[x], k)
5     else return TREE-SEARCH(right[x], k)
```

该过程从树的根结点开始进行查找，并沿着树下降，如图 12-2 中所示。对碰到的每个结点 x，就比较 k 和 $key[x]$。如果这两个关键字相同，则查找结束。如果 k 小于 $key[x]$，则继续查找 x 的左子树，因为由二叉查找树性质可知，k 不可能在 x 的右子树中。对称地，如果 k 大于 $key[x]$，则继续在 x 的右子树中查找。在递归查找过程中遇到的结点即构成了一条由树根下降的路径，故 TREE-SEARCH 算法的运行时间为 $O(h)$，h 是树的高度。

也可以用 while 循环来代替本过程中的递归。在大多数计算机上，非递归版本运行得要更快一些。

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else x ← right[x]
5  return x
```

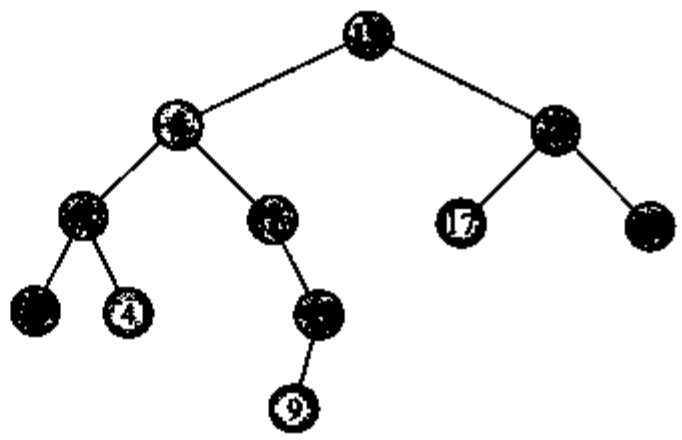


图 12-2　二叉查找树上的查询。为了在树中查找关键字 13，要沿着从根开始的路径 15→6→7→13 进行查找。树中的最小关键字为 2，它可以通过从根开始沿 *left* 指针寻找而找到。最大关键字 20 可以从根开始，沿 *right* 指针下降而找到。关键字为 15 的结点的后继是关键字为 17 的结点，因为它是 15 的右子树中的最小关键字。关键字为 13 的结点没有右子树，故它的后继为其最低的祖先，该祖先的左孩子也是个祖先。在这种情况下，关键字为 15 的结点就是它的后继 257

最大关键字元素和最小关键字元素

要查找二叉树中具有最小关键字的元素，只要从根结点开始，沿着各结点的 *left* 指针查找下去，直至遇到 NIL 时为止，如图 12-2 中所示。下面的过程返回一个指向以给定结点 x

为根的子树中最小元素的指针。

```
TREE-MINIMUM(x)
1  while left[x]≠NIL
2        do x ← left[x]
3  return x
```

二叉查找树性质保证了 TREE-MINIMUM 的正确性。如果一个结点 x 无左子树，其右子树中的每个关键字都至少和 $key[x]$一样大，则以 x 为根的子树中，最小关键字就是 $key[x]$。如果结点 x 有左子树，因其左子树中的关键字都不大于 $key[x]$，而其右子树中的关键字都不小于 $key[x]$，因此，在以 x 为根的子树中，最小关键字可在以 $left[x]$为根的左子树中找到。

过程 TREE-MAXIMUM 的伪代码是对称的：

```
TREE-MAXIMUM(x)
1  while right[x]≠NIL
2        do x ← right[x]
3  return x
```

对高度为 h 的树，这两个过程的运行时间都是 $O(h)$。这是因为，如在 TREE-SEARCH 过程中一样，所遇到的结点序列构成了一条沿着根结点向下的路径。

前趋和后继

给定一个二叉查找树中的结点，有时候要求找出在中序遍历顺序下它的后继。如果所有的关键字均不相同，则某一结点 x 的后继即具有大于 $key[x]$中的关键字中最小者的那个结点。根据二叉查找树的结构，不用对关键字做任何比较，就可以找到某个结点的后继。对于二叉查找树中的某个结点 x，下面的过程返回其后继(如果存在的话)，或返回 NIL(如果 x 具有树中最大关键字的话)。

258

```
TREE-SUCCESSOR(x)
1  if right[x]≠NIL
2     then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y≠NIL and x=right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

TREE-SUCCESSOR 的代码中包含两种情况。如果结点 x 的右子树非空，则 x 的后继即右子树中的最左结点，在第 2 行中通过调用 TREE-MINIMUM($right[x]$)可以找到这个结点。例如，在图 12-2 中，包含关键字 15 的结点的后继为关键字 17 的结点。

另一方面，正如练习 12.2-6 中要求读者证明的那样，如果结点 x 的右子树为空，且 x 有一个后继 y，则 y 是 x 的最低祖先结点，且 y 的左儿子也是 x 的祖先。在图 12-2 中，包含关键字 13 的结点的后继为包含关键字 15 的结点。为找到 y，可以从 x 开始向上查找，直到遇到某个是其父结点的左儿子的结点时为止。TREE-SUCCESSOR 中的第 3～7 行即完成这一工作。

对高度为 h 的一棵树，TREE-SUCCESSOR 算法的运行时间为 $O(h)$，因为我们或者是沿着树中的一条路径向上查找，或者是向下查找。过程 TREE-PREDECESSOR 与 TREE-SUCCESSOR 对称，其运行时间也是 $O(h)$。

即使各关键字不完全是不相同的，也可以通过分别调用 TREE-SUCCESSOR(x)和 TREE-

PREDECESSOR(x)，来将调用返回的结点定义为某一结点 x 的后继和前趋。

总之，通过以上的内容，我们其实已经证明了下面这个定理：

定理 12.2　对一棵高度为 h 的二叉查找树，动态集合操作 SEARCH、MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 等的运行时间均为 $O(h)$。 ■

练习

12.2-1　假设在某二叉查找树中，有 1 到 1000 之间的一些数，现要找出 363 这个数。下列的结点序列中，哪一个不可能是所检查的序列？

a)2，252，401，398，330，344，397，363

b)924，220，911，244，898，258，362，363

c)925，202，911，240，912，245，363

d)2，399，387，219，266，382，381，278，363

e)935，278，347，621，299，392，358，363

259

12.2-2　写出 TREE-MINIMUM 和 TREE-MAXIMUM 过程的递归版本。

12.2-3　写出 TREE-PREDECESSOR 过程。

12.2-4　Bunyan 教授认为他发现了二叉查找树的一个重要性质。假设在二叉查找树中，对某关键字 k 的查找在一个叶结点处结束，考虑三个集合：A，包含查找路径左边的关键字；B，包含查找路径上的关键字；C，包含查找路径右边的关键字。Bunyan 教授宣称，任何三个关键字 $a\in A$、$b\in B$、$c\in C$，必定满足 $a\leqslant b\leqslant c$。请给出该命题的一个最小可能的反例。

12.2-5　证明：如果二叉查找树中的某个结点有两个子女，则其后继没有左子女，其前趋没有右子女。

12.2-6　考虑一棵其关键字各不相同的二叉查找树 T。证明：如果 T 中某个结点 x 的右子树为空，且 x 有一个后继 y，那么 y 就是 x 的最低祖先，且其左孩子也是 x 的祖先。（注意每个结点都是它自己的祖先。）

12.2-7　对于一棵包含 n 个结点的二叉查找树，其中序遍历可以这样来实现：先用 TREE-MINIMUM 找出树中的最小元素，然后再调用 $n-1$ 次 TREE-SUCCESSOR。证明这个算法的运行时间为 $\Theta(n)$。

12.2-8　证明：在一棵高度为 h 的二叉查找树中，无论从哪一个结点开始，连续 k 次调用 TEEE-SUCCESSOR 所需的时间都是 $O(k+h)$。

12.2-9　设 T 为一棵其关键字均不相同的二叉查找树，并设 x 为一个叶子结点，y 为其父结点。证明：$key[y]$ 或者是 T 中大于 $key[x]$ 的最小关键字，或者是 T 中小于 $key[x]$ 的最大关键字。

260

12.3　插入和删除

插入和删除操作会引起以二叉查找树表示的动态集合的变化。要反映出这种变化，就要修改数据结构，但在修改的同时，还要保持二叉查找树性质。我们将看到，为插入一个新元素而修改树结构相对来说比较简单，但在执行删除操作时情况要复杂一些。

插入

为将一个新值 v 插入到二叉查找树 T 中，可以调用过程 TREE-INSERT。传给该过程的参数是个结点 z，并且有 $key[z]=v$，$left[z]=$NIL，$right[z]=$NIL。该过程修改 T 和 z 的某些域，并把 z 插入到树中的适当位置上。

```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x≠NIL
 4        do y ← x
 5           if key[z]<key[x]
 6              then x ← left[x]
 7              else x ← right[x]
 8  p[z] ← y
 9  if y=NIL
10     then root[T] ← z                ▷Tree T was empty
11     else if key[z]<key[y]
12             then left[y] ← z
13             else right[y] ← z
```

图 12-3 示出了 TREE-INSERT 的工作过程。像 TREE-SEARCH 和 ITERATIVE-TREE-SEARCH 一样，TREE-INSERT 从根结点开始，并沿树下降。指针 x 跟踪了这条路径，而 y 始终指向 x 的父结点。在初始化后，第 3～7 行中的 while 循环使这两个指针沿树下降，根据 $key[z]$与 $key[x]$的比较结果，可以决定向左或向右转。直到 x 成为 NIL 时为止。这个 NIL 所占位置即我们想插入项 z 的地方。第 8～13 行置有关 z 的指针。

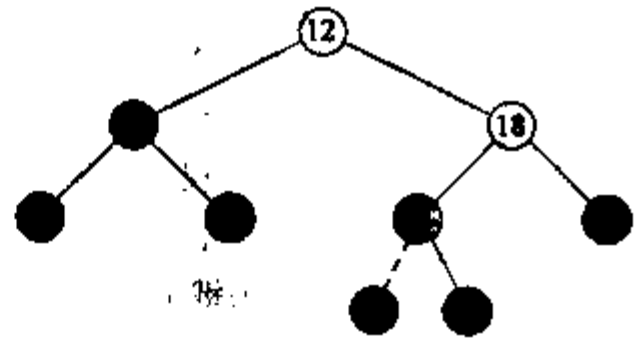


图 12-3　将关键字为 13 的数据项插入一棵二叉查找树。浅阴影结点表示从树根结点开始向下至该数据项插入位置的路径。虚线表示为在树中插入该数据项而加入的链接

和其他查找树上的原始操作一样，过程 TREE-INSERT 的运行时间为 $O(h)$，h 为树的高度。

删除

将给定结点 z 从二叉查找树中删除的过程以指向 z 的指针为参数，并考虑了如图 12-4 所示的三种情况。如果 z 没有子女，则修改其父结点 $p[z]$，使 NIL 为其子女；如果结点 z 只有一个子女，则可以通过在其子结点与父结点间建立一条链来删除 z。最后，如果结点 z 有两个子女，先删除 z 的后继 y(它没有左子女，见练习 12.2-5)，再用 y 的内容来替代 z 的内容。

过程 TREE-DELETE 中，这三种情况的组织略有不同。

```
TREE-DELETE(T, z)
 1  if left[z]=NIL or right[z]=NIL
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y]≠NIL
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x≠NIL
 8     then p[x] ← p[y]
 9  if p[y]=NIL
10     then root[T] ← x
11     else if y=left[p[y]]
12             then left[p[y]] ← x
```

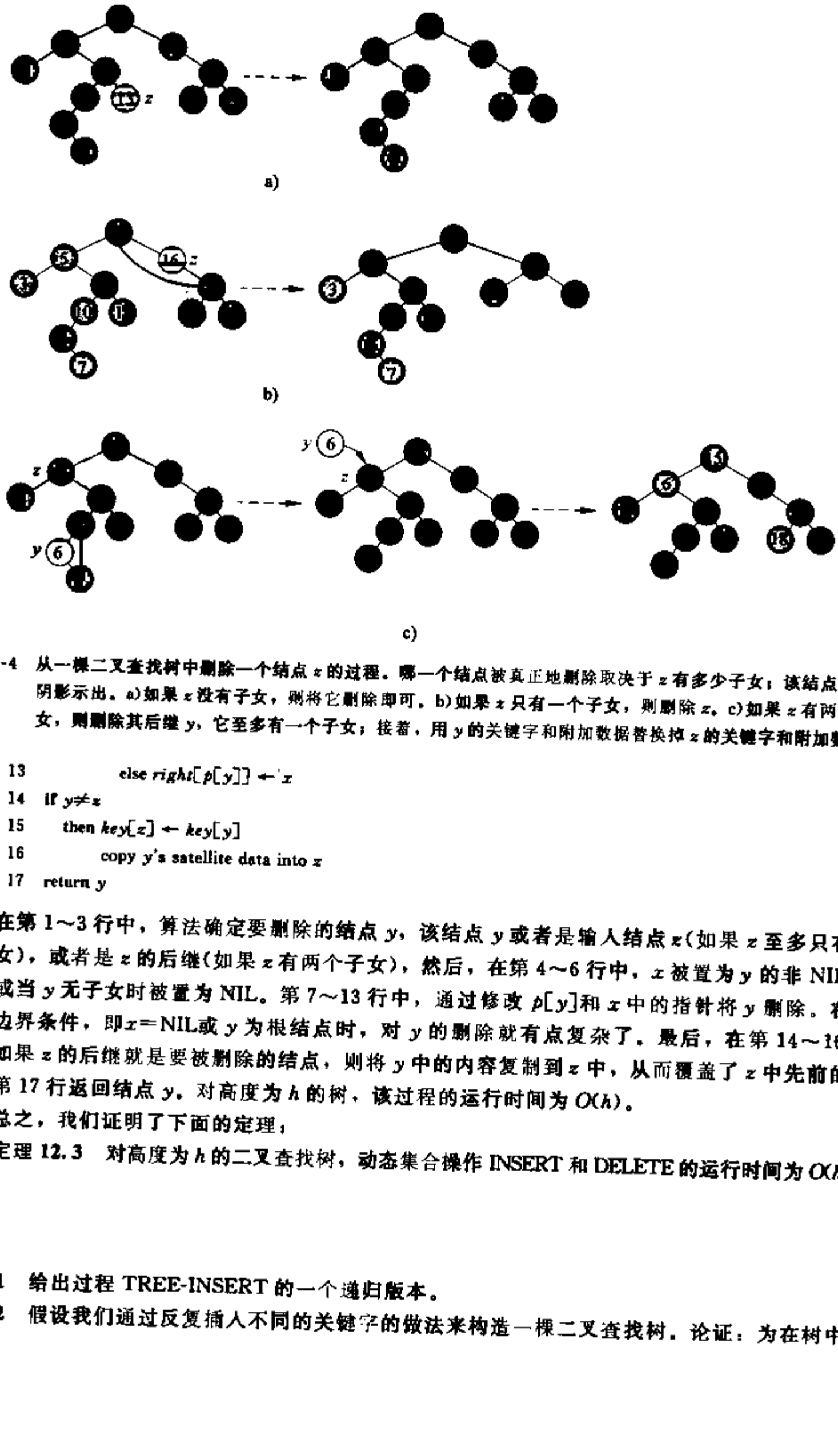


图 12-4　从一棵二叉查找树中删除一个结点 z 的过程。哪一个结点被真正地删除取决于 z 有多少子女：该结点以浅阴影示出。a)如果 z 没有子女，则将它删除即可。b)如果 z 只有一个子女，则删除 z。c)如果 z 有两个子女，则删除其后继 y，它至多有一个子女；接着，用 y 的关键字和附加数据替换掉 z 的关键字和附加数据

```
13          else right[p[y]] ← x
14  if y≠z
15    then key[z] ← key[y]
16         copy y's satellite data into z
17  return y
```

在第 1～3 行中，算法确定要删除的结点 y，该结点 y 或者是输入结点 z(如果 z 至多只有一个子女)，或者是 z 的后继(如果 z 有两个子女)，然后，在第 4～6 行中，x 被置为 y 的非 NIL 子女，或当 y 无子女时被置为 NIL。第 7～13 行中，通过修改 $p[y]$和 x 中的指针将 y 删除。在考虑到边界条件，即x=NIL或 y 为根结点时，对 y 的删除就有点复杂了。最后，在第 14～16 行中，如果 z 的后继就是要被删除的结点，则将 y 中的内容复制到 z 中，从而覆盖了 z 中先前的内容。第 17 行返回结点 y。对高度为 h 的树，该过程的运行时间为 $O(h)$。

261
|
263

总之，我们证明了下面的定理：

定理 12.3　对高度为 h 的二叉查找树，动态集合操作 INSERT 和 DELETE 的运行时间为 $O(h)$。

■

练习

12.3-1　给出过程 TREE-INSERT 的一个递归版本。

12.3-2　假设我们通过反复插入不同的关键字的做法来构造一棵二叉查找树。论证：为在树中查

找一个关键字，所检查的结点数等于插入该关键字时所检查的结点数加 1。

12.3-3 可以这样来对 n 个数进行排序：先构造一棵包含这些数的二叉查找树（重复应用 TREE-INSERT 来逐个地插入这些数），然后按中序遍历来输出这些数。这个排序算法的最坏情况和最好情况运行时间怎样？

12.3-4 假设另有一种数据结构中包含指向二叉查找树中某结点 y 的指针，并假设用过程 TREE-DELETE 来删除 y 的前趋 z。这样做会出现哪些问题？如何改写 TREE-DELETE 来解决这些问题？

12.3-5 删除操作是可交换的吗？（也就是说，先删除 x，再删除 y 的二叉查找树与先删除 y 再删除 x 的一样）说明为什么是的，或者给出一个反例。

12.3-6 当 TREE-DELETE 中的结点 z 有两个子结点时，可以将其前趋（而不是后继）拼接掉。有些人提出了一种公平的策略，即为前趋和后继结点赋予相同的优先级，从而可以得到更好的经验性能。那么，应如何修改 TREE-DELETE 来实现这样一种公平的策略？

264

*12.4 随机构造的二叉查找树

我们已经知道，二叉查找树上各基本操作的运行时间都是 $O(h)$，h 为树的高度。但是，随着元素的插入或删除，树的高度会发生变化。例如，如果各元素是按严格增长的顺序插入的，那么构造出来的树就是一个高度为 $n-1$ 的链。另一方面，练习 B.5-4 说明了 $h \geqslant \lfloor \lg n \rfloor$。如在快速排序中那样，我们可以证明，其平均情况下的行为更接近于最佳情况下的行为，而不是接近最坏情况下的行为。

不幸的是，如果在构造二叉查找树时，既用到了插入操作，又用到了删除，那么就很难确定树的平均高度到底是多少。如果仅用插入操作来构造树，则分析相对容易些。我们可以定义在 n 个不同的关键字上的一棵随机构造的二叉查找树，它是通过按随机的顺序，将各关键字插入一棵初始为空的树而形成的，并且各输入关键字的 $n!$ 种排列是等可能的。（练习 12.4-3 要求读者证明，这一概念不同于假定 n 个关键字上的每棵二叉查找树都是等可能的。）这一节要证明对于在 n 个关键字上随机构造的二叉查找树，其期望高度为 $O(\lg n)$。假设所有的关键字都是不同的。

首先来定义三个随机变量，它们有助于测度一棵随机构造的二叉查找树的高度。对于一棵在 n 个关键字上随机构造的二叉查找树，用 X_n 来表示其高度，并定义其指数高度为 $Y_n = 2^{X_n}$。当在 n 个关键字上构造一棵二叉查找树时，可以选择其中的一个作为树根结点的关键字，并设 R_n 表示一个随机变量，它存放了该关键字在这 n 个关键字中的序号。R_n 的值取集合 $\{1, 2, \cdots, n\}$ 中任何元素的可能性都是相同的。如果 $R_n = i$，则根的左子树是一棵在 $i-1$ 个关键字上随机构造的二叉查找树，其右子树是一棵在 $n-i$ 个关键字上随机构造的二叉查找树。因为一棵二叉树的高度比根结点的两棵子树中较高的那棵子树大 1，因此，一棵二叉树的指数高度就是根的两棵子树中指数高度较大者的两倍。如果知道 $R_n = i$，则有：

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i})$$

作为基础情况，有 $Y_1 = 1$，因为一棵只含一个结点的树的指数高度为 $2^0 = 1$，并且，为了方便起见，定义 $Y_0 = 0$。

接下来，定义指示器随机变量 $Z_{n,1}$，$Z_{n,2}$，$\cdots$，$Z_{n,n}$，其中

$$Z_{n,i} = \mathrm{I}\{R_n = i\}$$

因为 R_n 为集合 $\{1, 2, \cdots, n\}$ 中任何元素的可能性都是相同的，因而有 $\Pr\{R_n = i\} = 1/n$，$i = 1, 2, \cdots, n$，于是，根据引理 5.1，有：

$$\mathrm{E}[Z_{n,i}] = 1/n \tag{12.1}$$

其中 $i=1, 2, \cdots, n$。因为恰好有一个 $Z_{n,i}$的值为 1，所有其他的值都为 0，因而还有： 265

$$Y_n = \sum_{i=1}^{n} Z_{n,i}(2 \cdot \max(Y_{i-1}, Y_{n-i}))$$

下面还将证明，$E[Y_n]$是 n 的一个多项式，这又最终蕴含着 $E[X_n]=O(\lg n)$。

指示器随机变量 $Z_{n,i}=I\{R_n=i\}$独立于 Y_{i-1}和 Y_{n-i}的值。在选择了 $R_n=i$ 后，左子树的指数高度为 Y_{i-1}，是在 $i-1$ 个序号小于 i 的关键字上随机构造而成的。这一子树就象任何其他在 $i-1$ 个关键字上随机构造的二叉查找树一样。除了所包含的关键字数目外，这棵子树的结构根本不受选择 $R_n=i$ 的影响。因此，随机变量 Y_{i-1}和 $Z_{n,i}$是互相独立的。类似地，右子树的指数高度为 Y_{n-i}，是在 $n-i$ 个其序号大于 i 的关键字上随机构造而成的。其结构独立于 R_n 的值，因而随机变量 Y_{n-i}和 $Z_{n,i}$是互相独立的。于是，有：

$$\begin{aligned}
E[Y_n] &= E\left[\sum_{i=1}^{n} Z_{n,i}(2 \cdot \max(Y_{i-1}, Y_{n-i}))\right] \\
&= \sum_{i=1}^{n} E[Z_{n,i}(2 \cdot \max(Y_{i-1}, Y_{n-i}))] \quad \text{（根据线性期望）} \\
&= \sum_{i=1}^{n} E[Z_{n,i}]E[2 \cdot \max(Y_{i-1}, Y_{n-i})] \quad \text{（根据独立）} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot E[2 \cdot \max(Y_{i-1}, Y_{n-i})] \quad \text{（根据式(12.1)）} \\
&= \frac{2}{n}\sum_{i=1}^{n} E[\max(Y_{i-1}, Y_{n-i})] \quad \text{（根据式(C.21)）} \\
&\leqslant \frac{2}{n}\sum_{i=1}^{n}(E[Y_{i-1}] + E[Y_{n-i}]) \quad \text{（由练习 C.3-4）}
\end{aligned}$$

在上面最后一个和式中，每个项 $E[Y_0]$，$E[Y_1]$，…，$E[Y_{n-1}]$都出现了两次，一次是作为 $E[Y_{i-1}]$，另一次是作为 $E[Y_{n-i}]$，从而有递归式：

$$E[Y_n] \leqslant \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i] \tag{12.2}$$

利用替代方法，我们将证明对所有的正整数 n，递归式(12.2)有解

$$E[Y_n] \leqslant \frac{1}{4}\binom{n+3}{3}$$

在求解的过程中，要用到恒等式 266

$$\sum_{i=0}^{n-1}\binom{i+3}{3} = \binom{n+3}{4} \tag{12.3}$$

（练习 12.4-1 要求读者证明这一恒等式。）

对于基础情况，我们验证界

$$1 = Y_1 = E[Y_1] \leqslant \frac{1}{4}\binom{1+3}{3} = 1$$

成立。对于所做的替换，有：

$$\begin{aligned}
E[Y_n] &\leqslant \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i] = \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} \quad \text{（由规约假设）} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} = \frac{1}{n}\binom{n+3}{4} \quad \text{（由式(12.3)）}
\end{aligned}$$

$$= \frac{1}{n} \cdot \frac{(n+3)!}{4!(n-1)!} = \frac{1}{4} \cdot \frac{(n+3)!}{3!n!} = \frac{1}{4}\binom{n+3}{3}$$

上面，我们已给出了 $E[Y_n]$的界，但我们的最终目标是找出 $E[X_n]$的界。练习 12.4-4 中会要求读者证明，函数 $f(x)=2^x$ 是一个凸函数(见附录 C 的 C.3 节)。因此，可以应用 Jensen 不等式(C.25)，该不等式是这样的：

$$2^{E[X_n]} \leqslant E[2^{X_n}] = E[Y_n]$$

据此可以推导出：

$$2^{E[X_n]} \leqslant \frac{1}{4}\binom{n+3}{3} = \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} = \frac{n^3+6n^2+11n+6}{24}$$

267　对两边取对数，可得 $E[X_n]=O(\lg n)$。于是，这就证明了以下定理：

定理 12.4　一棵在 n 个关键字上随机构造的二叉查找树的期望高度为 $O(\lg n)$。 ■

练习

12.4-1　证明等式(12.3)。

12.4-2　请描述这样的一棵二叉查找树：其中每个结点的平均深度为 $\Theta(\lg n)$，但树的深度为 $\omega(\lg n)$。对于一棵含 n 个结点的二叉查找树，如果其中每个结点的平均深度为 $\Theta(\lg n)$，给出其高度的一个渐近上界。

12.4-3　说明基于 n 个关键字的随机选择二叉查找树概念(每棵包含 n 个结点的树被选到的可能性相同)，与本节中介绍的随机构造二叉查找树的概念是不同的(提示：列出 $n=3$ 时的各种可能)。

12.4-4　证明：$f(x)=2^x$ 是凸函数。

***12.4-5**　现对 n 个输入数调用 RRANDOMIZED-QUICKSORT。证明：对任何常数 $k>0$，输入数的所有 $n!$ 种排列中，除了其中的 $O(1/n^k)$种排列之外，都有 $O(n\lg n)$的运行时间。

思考题

12-1　具有相同关键字的二叉查找树

相同关键字的存在，给二叉查找树的实现带来了一些问题。

a)当用 TREE-INSERT 将 n 个具有相同关键字的数据项插入到一棵初始为空的二叉查找树中时，该算法的渐近性能如何？

我们可以对 TREE-INSERT 做一些改进，即在第 5 行的前面测试 $key[z]=key[x]$，在第 11 行前面测试 $key[z]=key[y]$。如果等式成立，我们对下列策略中的某一种加以实现。对每一种策略，请给出将 n 个具有相同关键字的数据项插入一棵初始为空的二叉查找树中

268　的渐近性能(以下的策略是针对第 5 行的，比较的是 z 和 x 的关键字。将 x 换成 y 即可用于第 11 行)。

b)在结点 x 处设一个布尔标志 $b[x]$，并根据 $b[x]$的不同值，置 x 为 $left[x]$或 $right[x]$。每当插入一个与 x 具有相同关键字的结点时，$b[x]$取 TRUE 或 FALSE。

c)在结点 x 处设置一个列表，其中所有结点都具有与 x 相同的关键字，并将 z 插入到该列表中。

d)随机地将 x 置为 $left[x]$或 $right[x]$。(给出最坏情况性能，并非形式地导出平均情况性能。)

12-2　基数树

给定两个串 $a=a_0a_1\cdots a_p$ 和 $b=b_0b_1\cdots\cdots b_q$，其中每一个 a_i 和每一个 b_j 都属于某有序字符集，如果下面两条规则之一成立，则说串 a 按字典序小于串 b：

1)存在一个整数 j，$0\leqslant j\leqslant\min(p,q)$，使 $a_i=b_i$，$i=0,1,\cdots,j-1$，且 $a_j<b_j$；

2)$p<q$，且 $a_i=b_i$，对所有 $i=0,1,\cdots,p$ 成立。

例如，如果 a 和 b 是位串，则根据规则 1)(设 $j=3$)，有 10100<10110；根据规则 2)，有 10100<101000。这与英语字典中的排序很相似。

图 12-5 中示出的是基数树(radix tree)数据结构，其中存储了位串 1011、10、011、100 和 0。当查找某关键字 $a=a_0a_1\cdots a_p$ 时，在深度为 i 的一个结点处，若 $a_i=0$ 则向左转；若 $a_i=1$ 则向右转。设 S 为一组不同的二进串构成的集合，各串的长度之和为 n。说明如何利用基数树，在 $\Theta(n)$时间内将 S 按字典序排序。例如，对图 12-5 中每个结点的关键字，排序的输出应该是序列 0、011、10、100、1011。

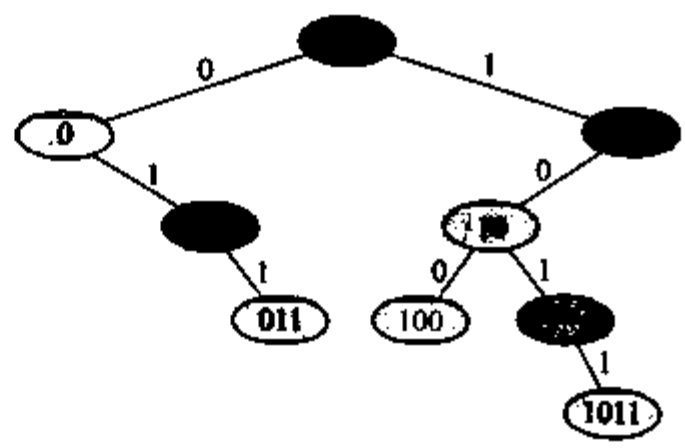


图 12-5　一棵存储了位串 1011、10、011、100 和 0 的基数树。每个结点的关键字可以通过遍历从根至该结点的路径而确定。因而，没有必要将关键字存储在结点中，图中示出的关键字仅为说明之用。如果某一结点对应的关键字不在树中，该结点即以深阴影表示；这样的结点的存在仅是为了建立与其他结点之间的通路

269

12-3　随机构造的二叉查找树中的平均结点深度

在这个问题里，我们要证明在一棵随机构造的二叉查找树中，n 个结点的平均深度为 $O(\lg n)$。虽然这个结果弱于定理 12.4，但我们将采用的技术却会揭示出构造二叉查找树与 7.3 节中 RANDOMIZED-QUICKSORT 的运行机制之间的令人惊奇的相似性。

定义一棵二叉树 T 的总路径长度 $P(T)$为 T 中所有结点 x 的深度之和，表示为 $d(x,T)$。

a)论证：T 中结点的平均深度为

$$\frac{1}{n}\sum_{x\in T}d(x,T)=\frac{1}{n}P(T)$$

进而，我们希望证明 $P(T)$的期望值为 $O(n\lg n)$。

b)设 T_L 和 T_R 分别表示树 T 的左右子树，论证：如果 T 有 n 个结点，则有：

$$P(T)=P(T_L)+P(T_R)+n-1$$

c)设 $P(n)$表示一棵包含 n 个结点的随机构造二叉树中的平均总路径长度。证明：

$$P(n)=\frac{1}{n}\sum_{i=0}^{n-1}(P(i)+P(n-i-1)+n-1)$$

d)证明：$P(n)$可以被重写为

$$P(n)=\frac{2}{n}\sum_{k=1}^{n-1}P(k)+\Theta(n)$$

e)回忆在思考题 7-2 中对快速排序的随机化版本所做的分析，总结 $P(n)=O(n\lg n)$

在对快速排序算法的每一次递归调用中，都是随机地选择一个支点元素来对待排序元素集合进行划分。二叉查找树中的每个结点也对以该结点为根的子树中的所有元素进行划分。

270

f) 请给出快速排序的一种实现，使其中为排序一组元素而做的比较，与将这些元素插入一棵二叉查找树所做的比较是正好一样的(所做的各次比较的次序可以不同，但所做的比较必须相同)。

12-4　不同的二叉树数目

设 b_n 表示包含 n 个结点的不同的二叉树的数目。在本问题里，要给出关于 b_n 的公式和一个渐近估计。

a)证明：$b_0=1$，且对 $n\geqslant 1$，有：

$$b_n=\sum_{k=0}^{n-1} b_k b_{n-1-k}$$

b)参见思考题 4-5 中给出的有关生成函数的定义，设 $B(x)$为生成函数

$$B(x)=\sum_{n=0}^{\infty} b_n x^n$$

证明 $B(x)=xB(x)^2+1$，因而表达 $B(x)$的一种方式是

$$B(x)=\frac{1}{2x}(1-\sqrt{1-4x})$$

在点 $x=a$ 处 $f(x)$的泰勒展式为

$$f(x)=\sum_{k=0}^{\infty}\frac{f^{(k)}(a)}{k!}(x-a)^k$$

其中 $f^{(k)}(x)$是在点 x 处 f 的 k 阶导数。

c)通过在 $x=0$ 处使用 $\sqrt{1-4x}$的泰勒展式，证明 $b_n=\frac{1}{n+1}\binom{2n}{n}$
(即第 n 个 Catalan 数。读者如果愿意的话，可以不用泰勒展式，而是可以将二项展开式(C.4)推广应用于非整数的指数 n 上，即对于任何实数 n 和任何整数 k，当 $k\geqslant 0$ 时，将
271 $\binom{n}{k}$解释为$n(n-1)\cdots(n-k+1)/k!$，否则为 0。)

d)证明：$b_n=\frac{4^n}{\sqrt{\pi}n^{3/2}}(1+O(1/n))$

本章注记

Knuth[185]对简单二叉查找树及其多种变形进行了很好的讨论。二叉查找树似乎是由一些人于 20 世纪 50 年代后期各自独立地提出的。基数树通常被称为检索树，这一英文名称来自于英文单词“retrieval”中间的几个字母。Knuth[185]中对它们也进行了讨论。

15.5 节还将介绍在构造之前，如果预先知道查找频率的话，该如何构造一棵最优的二叉查找树。亦即，给定每个关键字的查找频率和查找落在树中各关键字之间值的频率，我们可以这样来构造一棵二叉查找树，使得遵循这些频率的一组查找操作所检查的结点数最少。

12.4 节中的证明给出了一棵随机构造的二叉查找树期望高度的界，这一证明是由 Aslam[23]提出的。Martínez 和 Roura[211]给出了二叉查找树中插入和删除的随机化算法，其中这两种操作的结果是都是随机二叉查找树。但是，他们对随机二叉查找树的定义与本章中关于随机构造
272 二叉查找树的定义略有不同。

第13章 红 黑 树

由第12章我们知道，一棵高度为 h 的二叉查找树可以实现任何一种基本的动态集合操作，如 SEARCH，PREDECESOR，SUCCESSOR，MINIMUM，MAXIMUM，INSERT 和 DELETE 等，其时间都是 $O(h)$。这样，当树的高度较低时，这些操作就会执行得较快；但是，当树的高度较高时，这些操作的性能可能不比用链表好。红黑树(red-black tree)是许多“平衡的”查找树中的一种，它能保证在最坏情况下，基本的动态集合操作的时间为 $O(\lg n)$。

13.1 红黑树的性质

红黑树是一种二叉查找树，但在每个结点上增加一个存储位表示结点的颜色，可以是 RED 或 BLACK。通过对任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其他路径长出两倍，因而是接近平衡的。

树中每个结点包含五个域：*color*，*key*，*left*，*right* 和 p。如果某结点没有一个子结点或父结点，则该结点相应的指针(p)域包含值 NIL。我们将把这些 NIL 视为指向二叉查找树的外结点(叶子)的指针，而把带关键字的结点视为树的内结点。

一棵二叉查找树如果满足下面的红黑性质，则为一棵红黑树：

1)每个结点或是红的，或是黑的。

2)根结点是黑的。

3)每个叶结点(NIL)是黑的。

4)如果一个结点是红的，则它的两个儿子都是黑的。

5)对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

图 13-1a 给出一棵红黑树的例子。

为了便于处理红黑树代码中的边界条件，我们采用一个哨兵来代表 NIL(参见 10.2 节)。对一棵红黑树 T 来说，哨兵 $nil[T]$ 是一个与树内普通结点有相同域的对象。它的 *color* 域为 BLACK，而它的其他域——p，*left*，*right* 和 *key*——可以设置成任意允许的值。如图 13-1b 所示，所有指向 NIL 的指针都被替换成指向哨兵 $nil[T]$ 的指针。

使用哨兵后，就可以将结点 x 的 NIL 孩子视为一个其父结点为 x 的普通结点。虽然我们可以在树内的每一个 NIL 上新增一个不同的哨兵结点，来让每个 NIL 的父结点都有这样的定义，但是这种做法会浪费空间。我们的做法是采用一个哨兵 $nil[T]$ 来代表所有的 NIL——所有的叶子以及根部的父结点。哨兵的域 p，*left*，*right* 以及 *key* 的取值如何并不重要，为了方便起见，也可以在程序中设定它们。

通常我们将注意力放在红黑树的内部结点上，因为它们存储了关键字的值。在本章的其余部分，在画红黑树时都将忽略叶子，如图 13-1c 所示。

从某个结点 x 出发(不包括该结点)到达一个叶结点的任意一条路径上，黑色结点的个数称为该结点 x 的黑高度，用 $\mathrm{bh}(x)$ 表示。根据上面的性质 5)，黑高度概念是明确定义的，因为从该结点出发的所有下降路径都有相同的黑结点个数。红黑树的黑高度定义为其根结点的黑高度。

下面的引理说明为什么红黑树是一种好的查找树。

引理 13.1 一棵有 n 个内结点的红黑树的高度至多为 $2\lg(n+1)$。

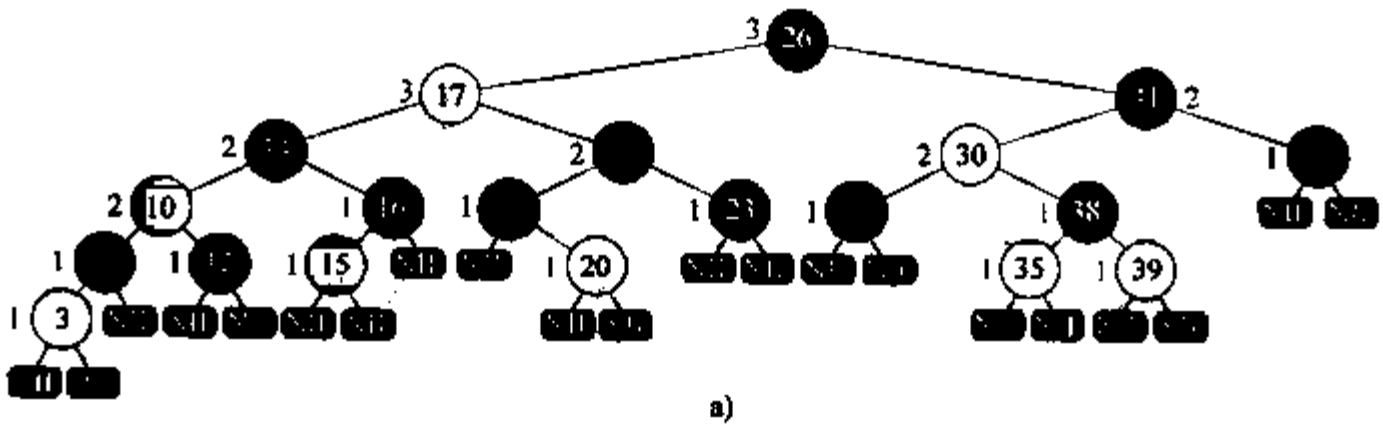


图 13-1　一棵红黑树，其中黑结点用黑色表示，红结点以浅阴影表示。在一棵红黑树内，每个结点或红或黑，红结点的两个儿子都是黑色，并且从每个结点到其子孙叶结点的每条简单路径上，都包含相同数目的黑色结点。a)每个显示成 NIL 的叶子都是黑色。每个非 NIL 的结点都会标上它的黑高度；NIL 的黑高度是 0。b)相同的红黑树，但每个 NIL 都改用一个永远是黑色的哨兵 $nil[T]$ 来代替，并且忽略黑高度。根部的父结点也是哨兵。c) 相同的红黑树，但是叶子与根部的父结点全部被忽略。本章的其余部分也将采用这种绘图方式

证明：先来证明以某一结点 x 为根的子树中至少包含 $2^{bh(x)}-1$ 个内结点。我们通过对 x 的高度进行归纳来证明这一点。如果 x 的高度是 0，则 x 必为一叶结点($nil[T]$)，这时以 x 为根的子树确实至少包含 $2^{bh(x)}-1=2^0-1=0$ 个内结点。对于归纳步骤，考虑一个其高度为正值的结点 x，它是个内结点，并且有两个子女。每个儿子根据其自身的颜色是红或黑，对应有黑高度 $bh(x)$ 或 $bh(x)-1$。因为 x 的儿子的高度小于 x 自身的高度，利用归纳假设，可以得出每个儿子至少包含 $2^{bh(x)-1}-1$ 个内结点。这样，以 x 为根的子树至少包含 $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$ 个内结点。这就证明了前面的推断。

274
~
275

为完成这个引理的证明，设 h 为树的高度。根据性质 4)，从根到叶结点(不包括根)的任一条简单路径上至少有一半的结点必是黑色的。从而，根的黑高度至少是 $h/2$；所以，

$$n \geqslant 2^{h/2} - 1$$

把 1 移到不等号左边，再对两边取对数，得 $\lg(n+1) \geqslant h/2$，或 $h \leqslant 2\lg(n+1)$。 ■

由这个引理可知，动态集合操作 SEARCH、MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 可用红黑树在 $O(\lg n)$时间内实现，因为这些操作在一棵高度为 h 的二叉查找树上的运行时间为 $O(h)$(见第 12 章)，而包含 n 个结点的红黑树又是高度为 $O(\lg n)$的查找树。(当然，在第 12 章的算法中，NIL 的引用必须用 $nil[T]$来代替。)虽然当给定一棵红黑树时，第 12 章的算法 TREE-INSERT 和 TREE-DELETE 的运行时间为 $O(\lg n)$，但是这两个算法并不直接支持动态集合操作 INSERT 和 DELETE，因为它们并不能保证被这些操作修改过的二叉查找树是棵红黑树。然而，我们将在 13.3 节和 13.4 节中看到，这两个操作确实可以在 $O(\lg n)$时间内完成。

练习

13.1-1 使用图 13-1a 的格式，画出在关键字集合{1，2，…，15}上高度为 3 的完全二叉查找树。以三种不同方式，向图中加入 NIL 叶结点并对各结点着色，使所得的红黑树的黑高度分别为 2，3 和 4。

13.1-2 对图 13-1 中的红黑树，画出调用 TREE-INSERT 插入关键字 36 后的结果。如果插入的结点被标为红色，所得的树是否还是一棵红黑树？如果该结点被标为黑色呢？

13.1-3 定义松弛红黑树为满足红黑性质 1，3，4 和 5 的二叉查找树。换言之，根部可以是红色或是黑色。考虑一棵根是红色的松弛红黑树 T。如果将 T 的根部标为黑色而其他都不变，则所得到的是否还是一棵红黑树？

13.1-4 假设将一棵红黑树的每一个红结点"吸收"到它的黑色父结点中，来让红结点的子女变成黑色父结点的子女(忽略关键字的变化)。当一个黑结点的所有红色子女都被吸收后，其可能的度是多少？此结果树的叶子深度怎样？

276

13.1-5 证明：在一棵红黑树中，从某结点 x 到其后代叶结点的所有简单路径中，最长的一条是最短一条的至多两倍。

13.1-6 在一棵黑高度为 k 的红黑树中，内结点最多可能有多少个？最少可能有多少个？

13.1-7 请描述出一棵在 n 个关键字上构造出来的红黑树，使其中红的内结点数与黑的内结点数的比值最大。这个比值是多少？具有最小可能比例的树又是怎样？此比值是多少？

13.2　旋转

当在含 n 个关键字的红黑树上运行时，查找树操作 TREE-INSERT 和 TREE-DELETE 的时间为 $O(\lg n)$。由于这两个操作对树作了修改，结果可能违反 13.1 节中给出的红黑性质。为保持这些性质，就要改变树中某些结点的颜色以及指针结构。

指针结构的修改是通过旋转来完成的，这是一种能保持二叉查找树性质的查找树局部操作。图 13-2中给出了两种旋转：左旋和右旋。当在某个结点 x 上做左旋时，我们假设它的右孩子 y 不是 $nil[T]$；x 可以为树内任意右孩子不是 $nil[T]$的结点。左旋以 x 到

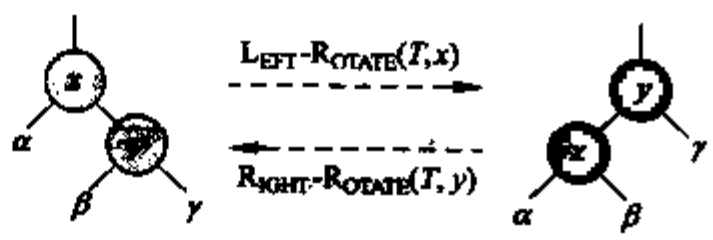


图 13-2　二叉查找树中的旋转操作。操作 LEFT-ROTATE(T, x)通过改变常数个指针来将左边两个结点的结构转变成右边的结构。右边的结构可以使用相反的操作 RIGHT-ROTATE(T, x)来转变成左边的结构。字母 α，β 以及 γ 代表任意的子树。旋转操作保留二叉查找树的属性：α 的关键字在 $key[x]$之前，$key[x]$又在 β 的关键字之前，β 的关键字在 $key[y]$之前，$key[y]$在 γ 的关键字之前

y 之间的链为“支轴”进行。它使 y 成为该子树新的根，x 成为 y 的左孩子，而 y 的左孩子则成为 x 的右孩子。

在 LEFT-ROTATE 的伪代码中，假设 $right[x] \neq nil[T]$，并且根的父结点是 $nil[T]$。

```
LEFT-ROTATE(T, x)
 1  y ← right[x]            ▷Set y.
 2  right[x] ← left[y]      ▷Turn y's left subtree into x's right subtree.
 3  p[left[y]] ← x
 4  p[y] ← p[x]             ▷Link x's parent to y.
 5  if p[x]=nil[T]
 6    then root[T] ← y
 7    else if x=left[p[x]]
 8           then left[p[x]] ← y
 9           else right[p[x]] ← y
10  left[y] ← x             ▷Put x on y's left.
11  p[x] ← y
```

图 13-3 显示了 LEFT-ROTATED 的操作过程。RIGHT-ROTATE 的程序是对称的。LEFT-ROTATE 和 RIGHT-ROTATE 都是在 $O(1)$ 时间内执行。在旋转时只有指针被改变；而结点中的所有其他域都保持不变。

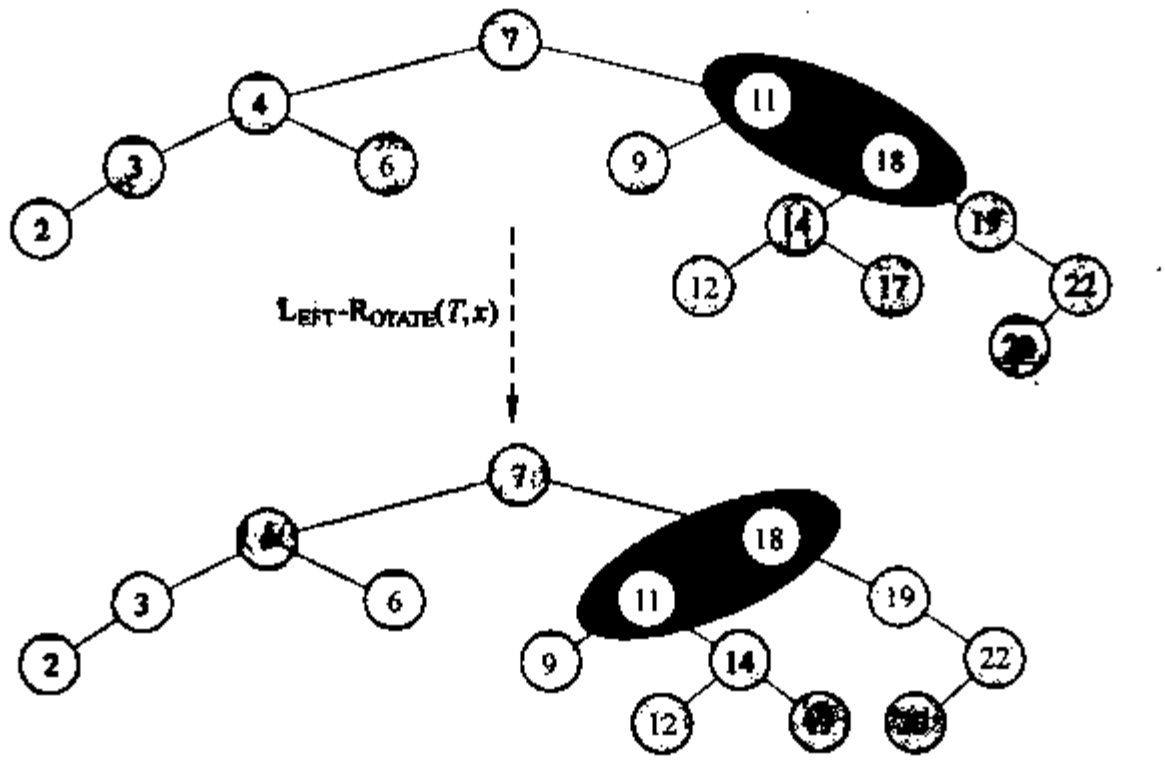


图 13-3　程序 LEFT-ROTATE(T, x)是如何修改一棵二叉查找树的示例。输入的树和修改过的树的中序遍历产生相同的关键字值列表

练习

13.2-1　写出 RIGHT-ROTATE 的伪代码。

13.2-2　证明：在一棵有 n 个结点的二叉查找树中，刚好有 $n-1$ 种可能的旋转。

13.2-3　设在图 13-2 的左边一棵树中，a，b 和 c 分别为子树 α，β 和 γ 中的任意结点。如果将结点 x 左旋，则 a，b 和 c 的深度会如何变化？

13.2-4　证明：任何一棵含 n 个结点的二叉查找树，可以通过 $O(n)$ 次旋转，转变为另一棵含 n 个结点的二叉查找树。

***13.2-5**　如果能够使用一系列的 RIGHT-ROTATE 调用来把一个二叉查找树 T_1 变为二叉查找树 T_2，则说 T_1 可以*右转成* T_2。请给出一个两棵树的例子，其中 T_1 不能够右转成 T_2。然后证明如果 T_1 可以右转成 T_2，则它可以使用 $O(n^2)$ 次 RIGHT-ROTATE 调用来右转。

277 ~ 279

13.3　插入

向一棵含 n 个结点的红黑树中插入一个新结点的操作可在 $O(\lg n)$ 时间内完成。我们利用 TREE-INSERT 过程(见 12.3 节)的一个略作修改的版本，来将结点 z 插入树 T 内，就好像 T 是一棵普通的二叉查找树一样，然后将 z 着为红色。为保证红黑性质能继续保持，我们调用一个辅助程序 RB-INSERT-FIXUP 来对结点重新着色并旋转。调用 RB-INSERT(T, z)会将 z 插入红黑树 T 内，假设 z 的 *key* 域已经事先被赋值。

```
RB-INSERT(T, z)
1  y ← nil[T]
2  x ← root[T]
3  while x≠nil[T]
4        do y ← x
5           if key[z]<key[x]
6              then x ← left[x]
7              else x ← right[x]
8  p[z] ← y
9  if y=nil[T]
10       then root[T] ← z
11       else if key[z]<key[y]
12               then left[y] ← z
13               else right[y] ← z
14 left[z] ← nil[T]
15 right[z] ← nil[T]
16 color[z] ← RED
17 RB-INSERT-FIXUP(T, z)
```

过程 TREE-INSERT 和 RB-INSERT 之间有四处不同。首先，在 TREE-INSERT 内的所有 NIL 都被 *nil*[T]代替。其次，在 RB-INSERT 的第 14、15 行中，设置 *left*[z]和 *right*[z]为 nil[T]，来保持正确的树结构。第三，在第 16 行将 z 着为红色。第四，因为将 z 着为红色可能违反某一条红黑性质，所以在 RB-INSERT 的第 17 行中，调用 RB-INSERT-FIXUP(T, z)来保持红黑性质。

280

```
RB-INSERT-FIXUP(T, z)
 1  while color[p[z]]=RED
 2        do if p[z]=left[p[p[z]]]
 3              then y ← right[p[p[z]]]
 4                   if color[y]=RED
 5                      then color[p[z]] ← BLACK                    ▷Case 1
 6                           color[y] ← BLACK                       ▷Case 1
 7                           color[p[p[z]]] ← RED                   ▷Case 1
 8                           z ← p[p[z]]                            ▷Case 1
 9                      else if z=right[p[z]]
10                              then z ← p[z]                       ▷Case 2
```

```
11                    LEFT-ROTATE(T, z)                          ▷Case 2
12                color[p[z]] ← BLACK                            ▷Case 3
13                color[p[p[z]]] ← RED                           ▷Case 3
14                RIGHT-ROTATE(T, p[p[z]])                       ▷Case 3
15          else (same as then clause with "right" and "left" exchanged)
16  color[root[T]] ← BLACK
```

为了理解 RB-INSERT-FIXUP 的工作过程，下面分三个主要步骤来分析其代码。首先，确定当结点 z 被插入并着为红色后，红黑性质中有哪些不能继续保持。其次，对第 1～15 行中 while 循环的总目标加以分析。最后，具体分析 while 循环中的三种情况[⊖]，看看它们是如何完成循环部分的目标的。图 13-4 显示了在一棵红黑树上 RB-INSERT-FIXUP 如何操作的一个例子。

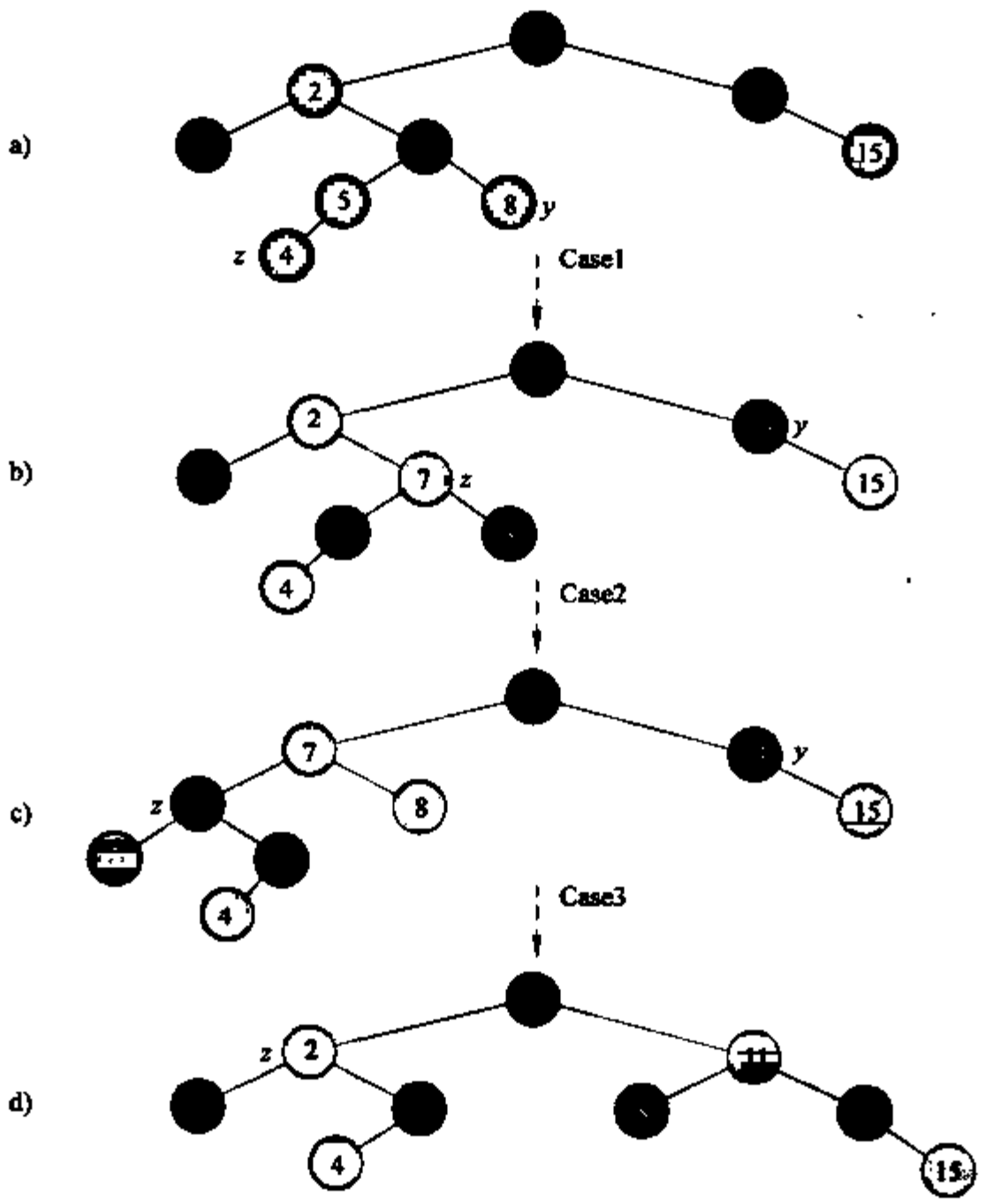


图 13-4　RB-INSERT-FIXUP 的操作。a)在插入后的结点 z。由于 z 和它的父结点 $p[z]$都是红色，所以违反了性质 4)。由于 z 的叔父结点 y 是红色，所以可以应用程序中的情况 1)。结点被重新着色，并且指针 z 沿树上升，所得的树如 b)。这样一来，z 及其父结点又都为红色，但 z 的叔父结点 y 是黑的。因为 z 是 $p[z]$的右孩子，可以应用情况 2)。在执行一个左旋之后，所得结果树见 c)。现在 z 是其父结点的左孩子，则可以应用情况 3)。执行一次右旋后得 d)中的树，它是一棵合法的红黑树

⊖　情况 2 落在情况 3 内，所以这两种情况不是相互排斥的。

在调用 RB-INSERT-FIXUP 时，哪些红黑性质可能会被破坏呢？性质 1)和性质 3)当然继续成立，因为新插入的结点的子女都是哨兵 $nil[T]$。性质 5)即从一个指定结点开始的每条路径上黑结点的个数都是相等的，也会成立，因为结点 z 代替了(黑色)哨兵，并且结点 z 本身是具有哨兵子女的红结点。因此，唯一可能被破坏的就是根结点需要为黑色的性质 2)，以及一个红结点不能有红子女的性质 4)。这两个可能的破坏是因为 z 被着为红色。如果 z 是根结点则破坏了性质 2)，如果 z 的父结点是红色就破坏了性质 4)。图 13-4a 显示在结点 z 被插入后性质 4)被破坏。 281 ~ 282

第 1～15 行中的 while 循环维持下列 3 个部分的不变式：

在循环的每一次迭代的开头，

a)结点 z 是红色。

b)如果 $p[z]$是根，则 $p[z]$是黑色。

c)如果有红黑性质被破坏，则至多只有一个被破坏，并且不是性质 2)就是性质 4)。如果违反性质 2)，则发生的原因是 z 是根而且是红的。如果违反性质 4)，则原因是 z 和 $p[z]$都是红的。

c)部分处理红黑性质的破坏，在说明 RB-INSERT-FIXUP 能保持红黑性质这一点上，比a)部分和 b)部分还要重要，我们用它来了解程序中的情况。因为我们把注意力集中在结点 z 以及树中靠近它的结点，所以从 a)部分知道 z 是红色会有帮助。当在第 2、3、7、8、13、14 行中引用 $p[p[z]]$时，我们使用 b)部分来表明结点 $p[p[z]]$的存在。

我们需要证明在循环的第一次迭代之前循环不变式为真，每次迭代都会使这个循环不变式保持成立，并且在循环结束时，这个循环不变式会给我们一个有用的性质。

我们从初始化和终止的推断开始。然后，在更详细地检查循环体如何工作时，我们将证明循环在每次迭代中都保持这个循环不变式。同时我们还说明循环的每次迭代会有两种可能的结果：指针 z 沿着树上移，或者执行某些旋转然后循环结束。

初始化：在循环的第一次迭代之前，我们从一棵正常的红黑树开始，新增一个红色结点 z。我们证明在调用 RB-INSERT-FIXUP 时，不变式的每一部分都成立。

a)在调用 RB-INSERT-FIXUP 时，z 是新增的红色结点。

b)如果 $p[z]$是根，则 $p[z]$从黑色开始，并且在调用 RB-INSERT-FIXUP 之前不变。

c)我们已经看到在调用 RB-INSERT-FIXUP 时性质 1)、3)、5)成立。

如果性质 2)被违反了，则红色的根必定是新增的结点 z，它是树中唯一的内结点。由于 z 的父结点和两个子女都是黑色的哨兵，没有违反性质 4)。所以，这一对性质 2)的违反是整棵树中，对红黑性质的唯一违反之处。

如果出现了对性质 4)的违反，则由于 z 的子女是黑色的哨兵，并且这棵树在 z 新增之前没有其他的违反，所以这一违反必定是因为 z 和 $p[z]$都是红色。而且，没有其他红黑性质的违反。 283

终止：循环结束是因为 $p[z]$是黑的。(如果 z 是根，则 $p[z]$是黑的哨兵 $nil[T]$。)所以，在循环结束时没有破坏性质 4)。根据循环不变式，唯一可能会不成立的性质是性质 2)。第 16 行会保持这个性质。所以当 RB-INSERT-FIXUP 结束的时候，所有的红黑性质都成立。

保持：实际上，在 while 循环中要考虑六种情况，但其中三种与另外三种是相互对称的，它由在第 2 行中确定的 z 的父结点 $p[z]$是 z 的祖父 $p[p[z]]$的左孩子还是右孩子而定。我们只给出 $p[z]$是左孩子时的代码。结点 $p[p[z]]$存在，因为根据循环不变式的 b)部分，如果 $p[z]$是根部，则 $p[z]$是黑的。由于我们只有在 $p[z]$是红色的时候才进入一次循环的迭代，所以我们知道 $p[z]$不可能是根。所以，$p[p[z]]$存在。

情况 1)与情况 2)、3)的区别在于 z 的父亲的兄弟(或叔叔)的颜色有所不同。第 3 行使 y 指向 z 的叔叔 $right[p[p[z]]]$，并且在第 4 行测试其颜色。如果 y 是红色，则执行情况 1)。否则，

控制转移到情况 2)和情况 3)上。在所有三种情况中，z 的祖父 $p[p[z]]$是黑的，因为它的父结点 $p[z]$是红的，故性质 4)只在 z 和 $p[z]$之间被破坏了。

情况 1)：z 的叔叔 y 是红色的

图 13-5 显示情况 1)(第 5～8 行)的状况。只有在 $p[z]$和 y 都是红色的时候才会执行情况 1)。既然 $p[p[z]]$是黑的，我们可以将 $p[z]$和 y 都着为黑色以解决 z 和 $p[z]$都是红色的问题，将 $p[p[z]]$着为红色以保持性质 5)。然后把 $p[p[z]]$当作新增的结点 z 来重复 while 循环。指针 z 在树中上移两层。

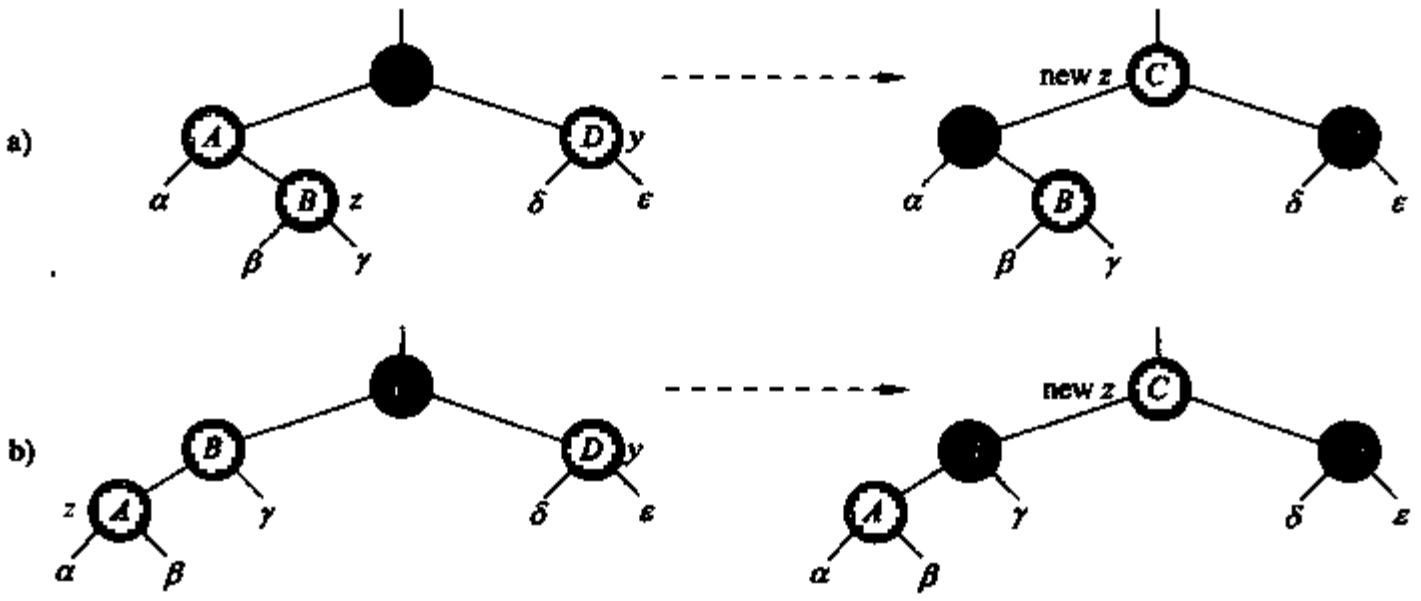


图 13-5　程序 RB-INSERT 中的情况 1)．性质 4)被违反，因为 z 和它的父结点 $p[z]$都是红的。相同的情形发生在 a)z 是一个右孩子或 b)z 是一个左孩子，每一棵子树 α、β、γ 和 δ 都有一个黑根，而且具有相同的黑高度。情况 1)的代码改变了某些结点的颜色，但保持了性质 5)：从一个结点向下到一个叶结点的所有路径都有相同数目的黑结点。while 循环将结点 z 的祖父 $p[p[z]]$作为新的 z 以继续迭代。现在性质 4)的破坏只能发生在新的红 z 和它的父结点之间，条件是如果父结点也是红的

现在来证明情况 1)在下一次迭代的开头会保持这个循环不变式。用 z 来表示当前迭代中的结点 z，而用 $z'=p[p[z]]$来表示在第 1 行中下一次迭代的测试时的结点 z。

a)因为这次迭代把 $p[p[z]]$着为红色，所以结点 z在下一次迭代的开始是红色。

b)在此次迭代中结点 $p[z']$是 $p[p[p[z]]]$，而且这个结点的颜色不变。如果它是根，则在此迭代之前它是黑的，并且在下一次迭代开始前仍然是黑的。

c)我们已经证明了情况 1)保持性质 5)，而且显然它也不违反性质 1)或性质 3)。

如果结点 z'在下一次迭代的开始是根，则在这次迭代中情况 1)修正了唯一被破坏的性质 4)。由于 z'是红的而且是根，所以性质 2)成为唯一被违反的性质，而且这是由 z'导致的。如果结点 z'在下一次迭代的开始不是根，则情况 1)不会导致性质 2)的破坏。情况 1)修正了在这次迭代的开始唯一违反的性质 4)。然后它把 z'着为红色而 $p[z']$不变。如果 $p[z']$是黑的，则没有违反性质 4)。如果 $p[z']$是红的，则把 z'着为红色会在 z'与 $p[z']$之间造成性质 4)的违反。

情况 2)：z 的叔叔 y 是黑色的，而且 z 是右孩子

情况 3)：z 的叔叔 y 是黑色的，而且 z 是左孩子

在情况 2)和情况 3)中，z 的叔叔 y 是黑色的。这两种情况是通过 z 是 $p[z]$的左孩子还是右孩子来区别。第 10、11 行构成了情况 2)，它和情况 3)一起在图 13-6 中显示。在情况 2)中，结点 z 是它的父亲的右孩子。我们立即使用一个左旋来将此状况转变为情况 3)(第 12～14 行)，此时结点 z 成为左孩子。因为 z 和 $p[z]$都是红色的，所以所作的旋转对结点的黑高度和性质 5)都无影响。无论是直接地或间接地通过情况 2)进入情况 3)，z 的叔叔 y 总是黑色的，因为否则的

话我们就应该执行情况 1)。另外，结点 $p[p[z]]$存在，因为我们已经推断它在执行第 2 行和第 3 行时存在，而且在第 10 行将 z 往上移一层、然后在第 11 行将 z 往下移一层之后，$p[p[z]]$的身份保持不变。在情况 3)中，要改变某些结点的颜色，并作一次右旋以保持性质 5)。这样，由于在一行中不再有两个连续的红色结点，因而，所有的处理到此完毕。因为此时 $p[z]$是黑色的，所以无需再执行一次 while 循环。

284
~
285

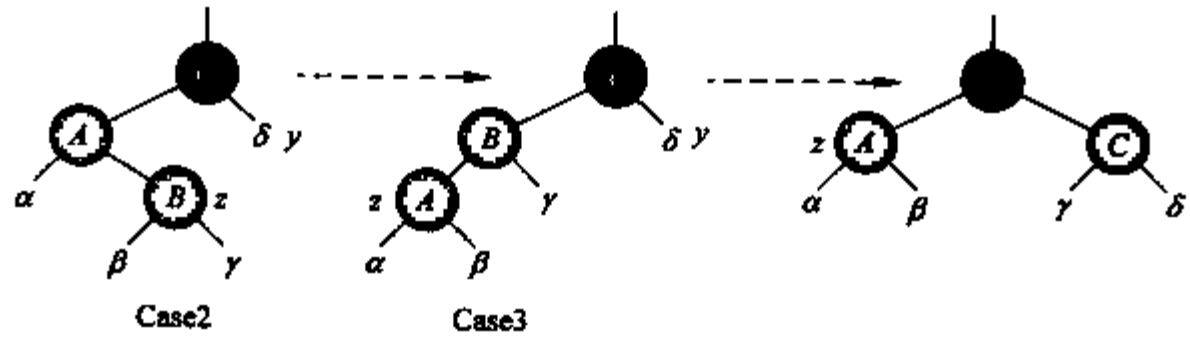


图 13-6　程序 RB-INSERT 的情况 2)、3)。如同情况 1)，性质 4)在情况 2)或情况 3)中被破坏是因为 z 和它的父亲 $p[z]$都是红的。每一棵子树 α、β、γ 和 δ 都有一个黑根(α、β 和 γ 是由性质 4)而来，δ 也有黑根是因为否则将导致情况 1))，而且具有相同的黑高度。情况 2)通过左旋转变为情况 3)，以保持性质 5)；从一个结点向下到一个叶结点的所有路径都有相同数目的黑结点。情况 3)引起某些结点颜色的改变，以及一个同样为了保持性质 5)的右旋。然后 while 循环结束，因为性质 4)已经得到了满足：一行中不再有两个连续的红色结点了

现在来证明情况 2)和情况 3)保持了循环不变式。(正如我们已经推断的，在第 1 行中 $p[z]$下一次测试会是黑色，循环体不会再执行。)

a)情况 2)让 z 指向红色的 $p[z]$，在情况 2)、3)中 z 或 z 的颜色都不再改变。

b)情况 3)把 $p[z]$变成黑色，所以如果 $p[z]$在下一次迭代的开始是根，则它是黑的。

c)如同情况 1)，性质 1)、3)、5)在情况 2)、3)中得以保持。

由于结点 z 在情况 2)、3)中都不是根，所以性质 2)没有被破坏。情况 2)、3)不会引起性质 2)的违反，因为唯一着为红色的结点在情况 3)中通过旋转成为一个黑色结点的子女。

情况 2)、3)修正了对性质 4)的违反，也不会导致违反其他的红黑性质。

在证明循环的每一次迭代都会保持循环不变式之后，就证明了 RB-INSERT-FIXUP 能够正确地保持红黑性质。

286

分析

RB-INSERT 的运行时间怎样呢？含 n 个结点的红黑树的高度为 $O(\lg n)$，因而 RB-INSERT 的第 1～16 行要花 $O(\lg n)$时间。在 RB-INSERT-FIXUP 中，仅当情况 1)被执行，然后指针 z 沿着树上升 2 层时，while 循环才会重复。while 循环可能被执行的总次数就为 $O(\lg n)$。所以 RB-INSERT 总共花费 $O(\lg n)$时间。有趣的是，该过程所作的旋转从不超过两次，因为只要执行了情况 2)或情况 3)之后 while 循环就结束了。

练习

13.3-1　在 RB-INSERT 的第 16 行中，假设新插入的结点 z 是红的。注意如果将 z 着为黑色，则红黑树的性质 4)就不会被破坏。那么我们为什么没有选择将 z 着为黑色呢？

13.3-2　在将关键字 41，38，31，12，19，8 插入一棵初始为空的红黑树中之后，结果树是什么样子？

13.3-3　假设图 13-5 和图 13-6 中子树 α、β、γ、δ 和 ε 的黑高度都是 k。标上各个结点的黑高度，

以验证图中所示的各种转换能保持性质5)。

13.3-4　Teach教授担心RB-INSERT-FIXUP可能将$color[nil[T]]$设置为RED，这时当z为根时第1行的测试就不会让循环结束。通过证明RB-INSERT-FIXUP永远不会将$color[nil[T]]$设置为RED，来说明这位教授的担心是没有必要的。

13.3-5　考虑用RB-INSERT插入n个结点而成的一棵红黑树。证明：如果$n>1$，则该树至少有一个红结点。

287　13.3-6　说明如果红黑树的表示中不提供父指针的话，应当如何有效地实现RB-INSERT。

13.4　删除

和n个结点的红黑树上的其他基本操作一样，对一个结点的删除要花$O(\lg n)$时间。与插入操作相比，删除操作只是稍微复杂些。

程序RB-DELETE是对TREE-DELETE程序(第12.3节)略作修改得来的。在删除一个结点后，该程序就调用一个辅助程序RB-DELETE-FIXUP，用来改变结点的颜色并做旋转，从而保持红黑树性质。

```
RB-DELETE(T, z)
 1  if left[z]=nil[T] or right[z]=nil[T]
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y]≠nil[T]
 5     then x ← left[y]
 6     else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y]=nil[T]
 9     then root[T] ← x
10     else if y=left[p[y]]
11             then left[p[y]] ← x
12             else right[p[y]] ← x
13  if y≠z
14     then key[z] ← key[y]
15          copy y's satellite data into z
16  if color[y]=BLACK
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

过程TREE-DELETE和RB-DELETE之间有三点不同。首先，TREE-DELETE中所有对NIL的引用在RB-DELETE中都被替换成对哨兵$nil[T]$的引用。其次，TREE-DELETE的第7行中判断x是否为NIL的测试被去掉了，取而代之的是在RB-DELETE的第7行中，无条件地执行赋值$p[x]\leftarrow p[y]$。这样，如果x是哨兵$nil[T]$，其父指针就指向被删除的结点y的父亲。第三，在第16、17行如果y是黑色的，则调用RB-DELETE-FIXUP。如果y是红色的，则当y被删除后，红黑性质仍然得以保持，理由如下：

- 树中各结点的黑高度都没有变化。
- 不存在两个相邻的红色结点。

288　- 因为如果y是红的，就不可能是根，所以根仍然是黑色的。

传递给RB-DELETE-FIXUP的结点x是两个结点中的一个：在y被删除之前，如果y有个

不是哨兵 $nil[T]$的孩子，则 x 为 y 的唯一孩子；如果 y 没有孩子，则 x 为哨兵 $nil[T]$。在后一种情况中，第 7 行中的无条件赋值保证了无论 x 是有关键字的内结点或哨兵 $nil[T]$，x 现在的父结点都为先前 y 的父结点。

现在看看过程 RB-DELETE-FIXUP 是如何恢复查找树的红黑性质的。

```
RB-DELETE-FIXUP(T, x)
 1  while x≠root[T] and color[x]=BLACK
 2      do if x=left[p[x]]
 3           then w ← right[p[x]]
 4                if color[w]=RED
 5                  then color[w] ← BLACK                          ▷Case 1
 6                       color[p[x]] ← RED                         ▷Case 1
 7                       LEFT-ROTATE(T, p[x])                      ▷Case 1
 8                       w ← right[p[x]]                           ▷Case 1
 9                if color[left[w]]=BLACK and color[right[w]]=BLACK
10                  then color[w] ← RED                            ▷Case 2
11                       x ← p[x]                                  ▷Case 2
12                  else if color[right[w]]=BLACK
13                         then color[left[w]] ← BLACK             ▷Case 3
14                              color[w] ← RED                     ▷Case 3
15                              RIGHT-ROTATE(T, w)                 ▷Case 3
16                              w ← right[p[x]]                    ▷Case 3
17                       color[w] ← color[p[x]]                    ▷Case 4
18                       color[p[x]] ← BLACK                       ▷Case 4
19                       color[right[w]] ← BLACK                   ▷Case 4
20                       LEFT-ROTATE(T, p[x])                      ▷Case 4
21                       x ← root[T]                               ▷Case 4
22           else (same as then clause with "right" and "left"exchanged)
23  color[x] ← BLACK
```

在 RB-DELETE 中，如果被删除的结点 y 是黑色的，则会产生三个问题。首先，如果 y 原来是根结点，而 y 的一个红色的孩子成为了新的根，这就违反了性质 2)。其次，如果 x 和 $p[y]$(现在也是 $p[x]$)都是红的，就违反了性质 4)。第三，删除 y 将导致先前包含 y 的任何路径上黑结点个数少 1。因此，性质 5)被 y 的一个祖先破坏了。补救这个问题的一个办法就是把结点 x 视为还 289
有额外的一重黑色。也就是说，如果将任意包含结点 x 的路径上黑结点个数加 1，则在这种假设下，性质 5)成立。当将黑结点 y 删除时，将其黑色"下推"至其子结点。现在问题变为结点 x 可能既不是红，又不是黑，从而违反了性质 1)。结点 x 是双重黑色或红黑，这就分别给包含 x 的路径上黑结点个数贡献 2 个或 1 个。x 的 *color* 属性仍然是 RED(如果 x 是红黑的)或 BLACK(如果 x 是双重黑色)。换言之，一个结点额外的黑色反映在 x 指向它，而不是它的 *color* 属性。

过程 RB-DELETE-FIXUP 恢复性质 1)、2)、4)。练习 13.4-1、13.4-2 要求读者说明这个过程是如何恢复性质 2)、4)的，在本节的其余部分将专注于性质 1)。第 1～22 行中 while 循环的目标是将额外的黑色沿树上移，直到：

1)x 指向一个红黑结点，此时在第 23 行中，将 x(单独)着为黑色；

2)x 指向根，这时可以简单地消除那个额外的黑色，或者

3)做必要的旋转和颜色修改。

在 while 循环中，x 总是指向具有双重黑色的那个非根结点。在第 2 行中要判断 x 是其父亲

$p[x]$的左孩子或是右孩子。(已经给出了 x 为左孩子时的代码；x 为右孩子时(第 22 行)是对称的。)对 x 的兄弟，用指针 w 加以记录。因为 x 是双重黑色的，故 w 不能是 $nil[T]$；否则，从 $p[x]$至(单黑色)叶结点 w 的路径上的黑结点个数就会小于从 $p[x]$到 x 的路径上的黑结点数。

算法中的四种情况⊖在图 13-7 中加以说明。在具体研究每一种情况之前，先看看每种情况中的变换是如何保持性质 5)的。关键思想是在每种情况中，从(且包括)子树的根到每棵子树 α，β，…，ζ之间的黑结点个数(包括 x 的额外黑色)并不被变换所改变。因此，如果性质 5)在变换之前成立，那么之后仍然成立。图 13-7a 说明了情况 1)，其中根至任一子树 α 或 β 之间的黑结点数都是 3，这在变换前后是一样的。(再次记住，结点 x 增加了额外一重黑色。)类似地，在变换
290 前后根至子树 γ，δ，ε，ζ 中的任一者之间的黑结点数都是 2。在图 13-7b 中，计数时还要包括

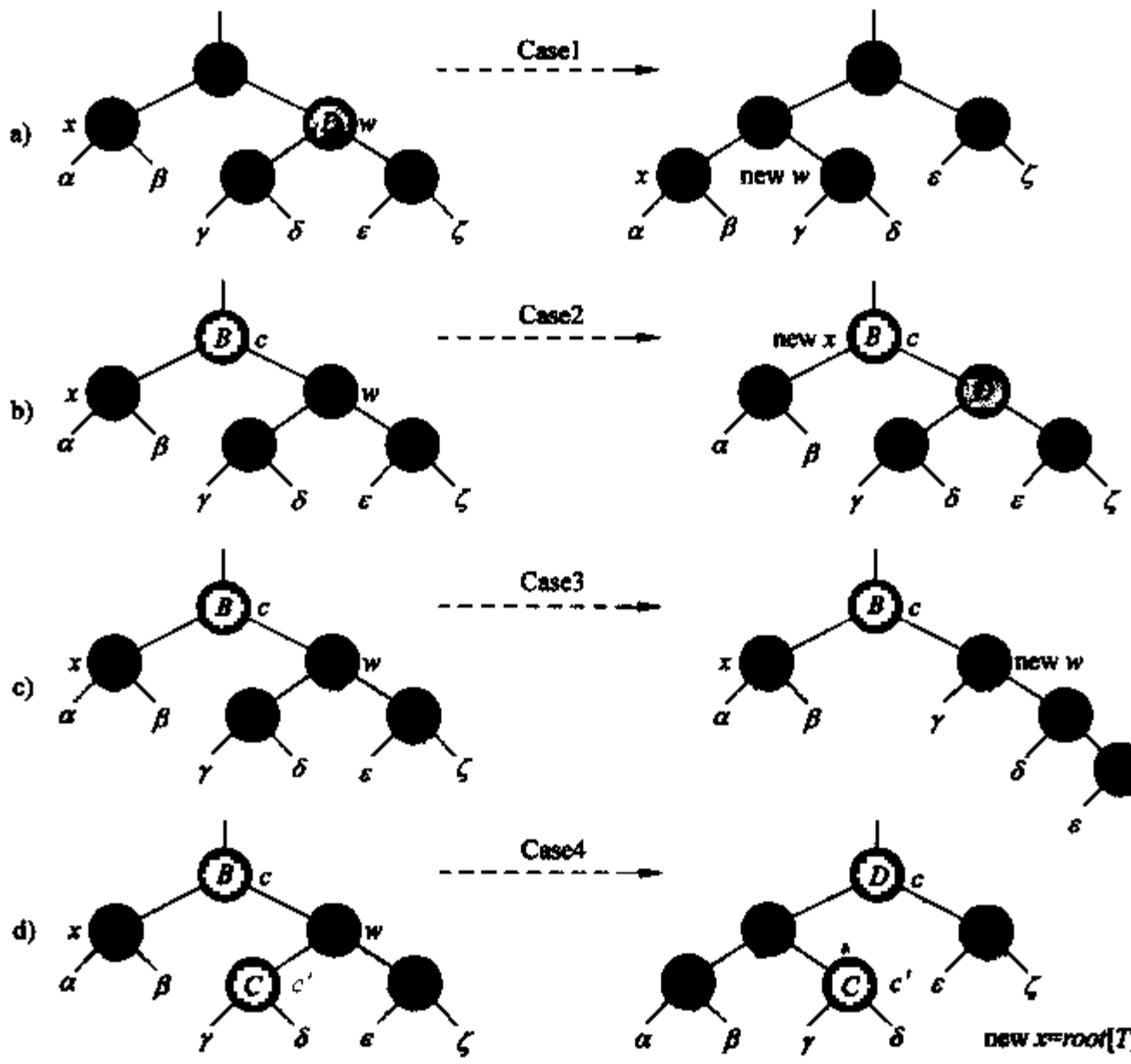


图 13-7　过程 RB-DELETE-FIXUP 中 while 循环的各种情况。加黑的结点 *color* 属性为 BLACK，深阴影的结点 *color* 属性为 RED，浅阴影的结点 *color* 属性用 c 和 c' 表示，也可为 RED 或者为 BLACK。字母 α，β，…，ζ 代表任意的子树。在每种情况中，通过改变某些结点的颜色及/或一次旋转，可以将左边的形式转化为右边的形式。x 指向的任何结点都具有额外的一重黑色而成为双重黑或红黑。引起循环重复的唯一情况即情况 2)。a)通过交换结点 B 和 D 的颜色以及执行一次左旋可将情况 1)转化为情况 2)、3)、4)。b)在情况 2)中，由指针 x 所表示的额外黑色在将结点 D 着为红色，并将 x 置为指向结点 B 后沿树上升。如果通过情况 1)进入情况 2)，则 while 循环结束，因为新的结点 x 是红黑的，因此其 *color* 属性 c 是 RED。c)通过交换结点 C 和 D 的颜色并执行一次右旋，可以将情况 3)转换成情况 4)。d)在情况 4)中，通过改变某些结点的颜色并执行一次左旋(不违反红黑性质)，可
291 以将由 x 表示的额外黑色去掉，然后循环结束

⊖ 和在 RB-INSERT-FIXUP 中一样，在 RB-DELETE-FIXUP 中的情况不是相互排斥的。

所示子树的根的 *color* 属性的值 c，它或是 RED，或是 BLACK。如果定义 count(RED)＝0 以及 count(BLACK)＝1，则根至 a 的黑结点数为 2＋count(c)，变换前后都一样。在此情况中，在变换之后新结点 x 的 *color* 属性值为 c，但是 x 或者是红黑的(如果 c＝RED)或者是双重黑色的(如果 c=BLACK)。其他情况可以类似地加以验证(见练习 13.4-5)。

情况 1)：x 的兄弟 w 是红色的

情况 1)(见 RB-DELETE-FIXUP 的第 5～8 行和图 13-7a 当结点 x 的兄弟 w 为红色时发生。因为 w 必须有黑色孩子，我们可以改变 w 和 $p[x]$颜色，再对 $p[x]$做一次左旋，而且红黑性质得以继续保持。现在，x 的新兄弟是旋转之前 w 的某个孩子，其颜色为黑色。这样，我们已经将情况 1)转换为情况 2)，3)或情况 4)。

当结点 w 为黑色时，情况 2)，3)和情况 4)发生。根据 w 的子结点的颜色对它们加以区分。

情况 2)：x 的兄弟 w 是黑色的，而且 w 的两个孩子都是黑色的

在情况 2)(见 RB-DELETE-FIXUP 的第 10、11 行和图 13-7b 中，w 的两个孩子都是黑色的。因为 w 也是黑色的，故从 x 和 w 上去掉一重黑色，从而 x 只有一重黑色而 w 为红色。为了补偿从 x 和 w 中去掉一重黑色，我们想在原来是红色或黑色的 $p[x]$内新增一重额外黑色。通过以 $p[x]$为新结点 x 来重复 while 循环。注意如果通过情况 1)进入情况 2)，则新结点 x 是红黑色的，因为原来的 $p[x]$是红色的。因此，新结点 x 的 *color* 属性的值 c 为 RED，并且在测试循环条件后循环结束。然后新结点 x 在第 23 行中被(单独)着为黑色。

情况 3)：x 的兄弟 w 是黑色的，w 的左孩子是红色的，右孩子是黑色的

情况 3)(见第 13～16 行和图 13-7c 在 w 为黑色且其左孩子为红色，右孩子为黑色时发生。可以交换 w 和其左孩子 $left[w]$的颜色，并对 w 进行右旋，而红黑性质仍然保持。现在 x 的新兄弟 w 是一个有红色右孩子的黑结点，这样我们已经将情况 3)转换成了情况 4)。

情况 4)：x 的兄弟 w 是黑色的，而且 w 的右孩子是红色的

情况 4)(见第 17～21 行和图 13-7d 当结点 x 的兄弟 w 为黑色且 w 的右孩子为红色时发生。通过作某些颜色修改并对 $p[x]$做一次左旋，可以去掉 x 的额外黑色来把它变成单独黑色，而不破坏红黑性质。将 x 置为根后，当 while 循环测试其循环条件时循环便结束。 292

分析

RB-DELETE 的运行时间怎样呢？含 n 个结点的红黑树的高度为 $O(\lg n)$，不调用 RB-DELETE-FIXUP 时该程序的总时间代价为 $O(\lg n)$。在 RB-DELETE-FIXUP 中，情况 1)，3)和情况 4)在各执行一定次数的颜色修改和至多三次旋转后便结束。情况 2)是 while 循环唯一可以重复的情况，其中指针 x 沿树上升的次数至多为 $O(\lg n)$次，且不执行任何旋转。所以，过程 RB-DELETE-FIXUP 要花 $O(\lg n)$时间，做至多三次旋转，从而 RB-DELETE 的总时间为 $O(\lg n)$。

练习

13.4-1　证明：在执行 RB-DELETE-FIXUP 之后，树根总是黑色的。

13.4-2　证明：在 RB-DELETE 中，如果 x 和 $p[y]$都是红色的，则性质 4)可以通过调用 RB-DELETE-FIXUP(T, x)来恢复。

13.4-3　在练习 13.3-2 中，我们将关键字 41，38，31，12，19，8 连续插入一棵初始为空的树中，从而得到一棵红黑树。请给出从该树中连续删除关键字 8，12，19，31，38，41 后的结果。

13.4-4　在 RB-DELETE-FIXUP 的哪些行中，可能会检查或修改哨兵 $nil[T]$？

13.4-5　在图 13-7 的每种情况中，给出所示子树的根至每个子树 α，β，…，ζ 之间的黑结点个数，并验证它们在转换之后保持不变。当一个结点的 *color* 属性为 c 或 c' 时，在计数中可用记号 count(c)或 count(c')来表示。

13.4-6　Skelton 和 Baron 教授担心在 RB-DELETE-FIXUP 的情况 1)的开头，结点 $p[x]$可能不是黑色。如果这两位教授是对的，则第 5、6 行就是错的。证明 $p[x]$在情况 1)的开头必是
293 黑色的，因此这两位教授没有必要担心。

13.4-7　假设用 RB-INSERT 来将一个结点 x 插入一棵红黑树，紧接着又用 RB-DELETE 将它从树中删除。结果的红黑树与初始的红黑树是否相同？对你的答案加以说明。

思考题

13-1　持久动态集合

有时，在算法的执行过程中，会发现在更新一个动态集合时，需要维护其过去的版本。这样的集合被称为是持久的。实现持久集合的一种方法是每当该集合被修改时，就将其整个地复制下来，但是这种方法会降低一个程序的执行速度，而且占用过多的空间。有时候，可以做得更好。

考虑一个有 INSERT、DELETE 和 SEARCH 操作的持久集合 S，我们使用如图 13-8a 所示的二叉查找树来实现。对集合的每一个版本都维护一个单独的根。为把关键字 5 插入到集合中去，就要创建一个具有关键字 5 的新结点。该结点成为具有关键字 7 的新结点的左孩子，因为我们不能修改具有关键字 7 的既存结点。类似地，具有关键字 7 的新结点成为具有关键字 8 的新结点的左孩子，后者的右孩子为具有关键字 10 的既存结点。关键字为 8 的新结点又成为关键字为 4 的结点的新根 r' 的右孩子，而 r' 的左孩子是关键字为 3 的既存结点。这样，我们只是复制了树的一部分，新树和老树之间共享一些结点，如图 13-8b 所示。

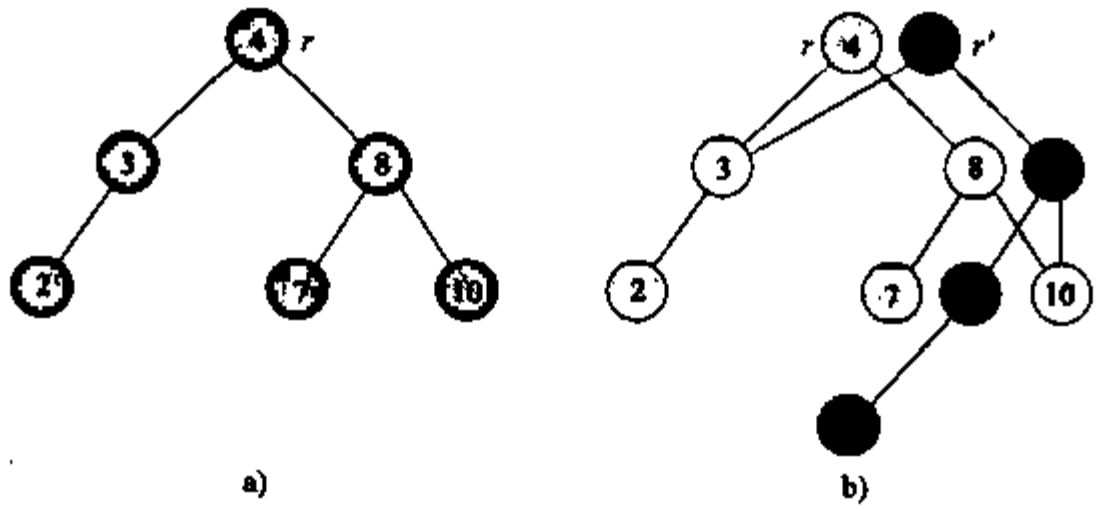


图 13-8　a)包含关键字 2，3，4，7，8，10 的一棵二叉查找树。b)插入关键字 5 后的持久二叉查找树。该集合的最新版本包括由根 r' 出发可到达的结点，而前一个版本包括由根 r 可到达的结点。当关键字 5 插入时就增加那些深阴影的结点

假设树中每个结点都有域 *key*，*left* 和 *right*，但是没有父结点的域。(参见练习 13.3-6。)

a)对一棵一般的持久二叉查找树，为插入一个关键字 k 或删除一个结点 y，确定需要改变哪些结点。

b)请写出一个程序 PERSISTENT-TREE-INSERT，使得在给定一棵持久树 T 和一个要插入的关键字 k 时，它返回将 k 插入 T 后新的持久树 T'。

c)如果持久二叉查找树 T 的高度为 h，所实现的 PERSISTENT-TREE-INSERT 的时间和空间要求分别是多少？(空间要求与新分配的结点数成正比。)

d)假设我们在每个结点中增加一个父亲结点域。这样一来，PERSISTENT-TREE-INSERT 需要做一些额外的复制工作。证明在这种情况下，PERSISTENT-TREE-INSERT 的时空要求为 $\Omega(n)$，其中 n 为树中的结点个数。

e)说明如何利用红黑树来保证每次插入或删除的最坏情况运行时间为 $O(\lg n)$。

13-2　红黑树上的连接操作

连接操作以两个动态集合 S_1 和 S_2 和一个元素 x 为参数，使对任何 $x_1 \in S_1$ 和 $x_2 \in S_2$，有 $key[x_1] \leqslant key[x] \leqslant key[x_2]$。该操作返回一个集合 $S = S_1 \cup \{x\} \cup S_2$。在这个问题中，讨论如何在红黑树上实现连接操作。

a)给定一棵红黑树 T，其黑高度被存放在域 $bh[T]$中。证明在不需要树中结点的额外存储空间和不增加渐近运行时间的前提下，可以用 RB-INSERT 和 RB-DELETE 来维护这个域。并证明当沿 T 下降时，可对每个被访问的结点在 $O(1)$时间内确定其黑高度。

我们希望实现操作 RB-JOIN(T_1，x，T_2)，它删除 T_1 和 T_2，并返回一棵红黑树 $T = T_1 \cup \{x\} \cup T_2$。设 n 为 T_1 和 T_2 中的总结点数。

294
~
295

b)假设 $bh[T_1] \geqslant bh[T_2]$。请描述一个 $O(\lg n)$时间的算法，使之能在 T_1 中从黑高度为$bh[T_2]$的结点中选出具有最大关键字的黑结点 y。

c)设 T_y 是以 y 为根的子树。说明如何在不破坏二叉查找树性质的前提下，在 $O(1)$时间里用 $T_y \cup \{x\} \cup T_2$ 来取代 T_y。

d)要保持红黑性质 1)，3)和性质 5)，应将 x 着什么颜色？说明如何能在 $O(\lg n)$时间内恢复性质 2)、4)。

e)论证 b)部分作的假设不失一般性。描述当 $bh[T_1] \leqslant bh[T_2]$时所出现的对称情况。

f)证明 RB-JOIN 的运行时间是 $O(\lg n)$。

13-3　AVL 树

AVL 树是一种高度平衡的二叉查找树：对每一个结点 x，x 的左子树与右子树的高度差至多为 1。要实现一棵 AVL 树，我们在每个结点内维护一个额外的域：$h(x)$，即结点的高度。至于任何其他的二叉查找树 T，假设 $root[T]$指向根结点。

a)证明一棵有 n 个结点的 AVL 树其高度为 $O(\lg n)$。(提示：证明在一个高度为 h 的 AVL 树中，至少有 F_h 个结点，其中 F_h 是第 h 个斐波那契数。)

b)为把结点插入到一棵 AVL 树中，首先以二叉查找树的顺序把结点放在适当的位置上。在插入之后，这棵树可能就不再是高度平衡了。具体地，某些结点的左子树与右子树的高度差可能会到 2。请描述一个程序 BALANCE(x)，输入一棵以 x 为根的子树，其左子树与右子树都是高度平衡的，而且它们的高度差至多是 2，即 $|h[right[x]] - h[left[x]]| \leqslant 2$，然后将以 x 为根的子树转变为高度平衡的。(提示：使用旋转。)

c)利用 b)来描述一个递归程序 AVL-INSERT(x，z)，输入 AVL 树中的一个结点 x 以及一个新创建的结点 z(其关键字已经填入)，然后把 z 添加到以 x 为根的子树中，并保持 x 是 AVL 树的根的性质。和第 12.3 节中的 TREE-INSERT 一样，假设 $key[z]$已经填入，且 $left[z] = \text{NIL}$，$right[z] = \text{NIL}$；并假设 $h[z] = 0$。这样，为把结点 z 插入到 AVL 树 T 中，我们调用 AVL-INSERT($root[T]$，z)。

d)请给出一个有 n 个结点的 AVL 树的例子，其中一个 AVL-INSERT 操作将执行 $\Omega(\lg n)$次旋转。

13-4 Treap

如果将一个含 n 个元素的集合插入到一棵二叉查找树中，所得到的树可能会非常不平
296 衡，从而导致查找时间很长。然而在 12.4 节中看到，随机构造的二叉查找树往往是平衡的。因此，一般来说，要为一组固定的元素建立一棵平衡树，可以采用的一种策略就是先随机排列这些元素，然后按照排列的顺序将它们插入到树中。

如果没法同时得到所有的元素的话，应该怎样处理呢？如果一次收到一个元素，是否能仍然用它们来随机建立一棵二叉查找树？

我们将通过考察一个数据结构来正面回答这个问题。一棵 treap 是一棵修改了结点顺序的二叉查找树。图 13-9 显示了一个例子。通常树内的每个结点 x 都有一个关键字值 $key[x]$。另外，还要为结点分配 $priority[x]$，它是一个独立选取的随机数。假设所有的优先级都是不同的，而且所有的关键字也是不同的。treap 的结点排列成让关键字遵循二叉查找树性质，并且优先级遵循最小堆顺序性质：

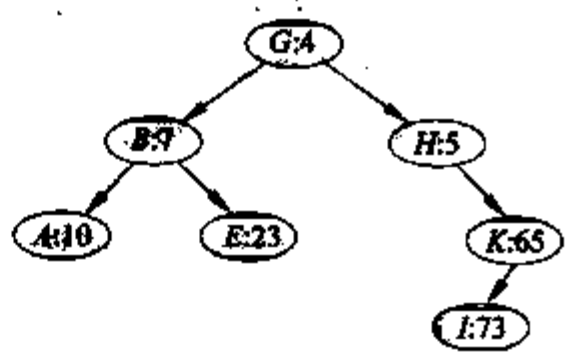


图 13-9　一个 treap。每个结点 x 都被标以 $key[x]$：$priority[x]$。例如，根的关键字是 G，优先级是 4

- 如果 v 是 u 的左孩子，则 $key[v] < key[u]$。
- 如果 v 是 u 的右孩子，则 $key[v] > key[u]$。
- 如果 v 是 u 的孩子，则 $priority[v] > priority[u]$。

（这两个性质的结合就是为什么这种树被称为“treap”的原因：它同时具有二叉查找树和堆的特征。）

用以下方式考虑 treap 会有帮助。假设插入关联关键字的结点 x_1，x_2，…，x_n 到一棵 treap 内。结果的 treap 是将这些结点以它们的优先级（随机选取）的顺序插入一棵正常的二叉查找树形成的，亦即 $priority[x_i] < priority[x_j]$ 表示 x_i 在 x_j 之前被插入。

a）证明：给定一个结点集合 x_1，x_2，…，x_n，它们关联了关键字和优先级（互异），
297 存在唯一的一棵 treap 与这些结点相关联。

b）证明：treap 的期望高度是 $\Theta(\lg n)$，因此在 treap 内查找一个值所花的时间为 $\Theta(\lg n)$。

让我们看看如何将一个新的结点插入到一个既存的 treap 中。第一件事就是指定一个随机的优先级给这个新结点。然后调用称为 TREAP-INSERT 的插入算法，其操作如图 13-10 所示。

c）解释 TREAP-INSERT 如何工作。解释其思想并给出伪代码。（提示：执行通常的二叉查找树插入程序然后做旋转来恢复最小堆顺序的性质。）

d）证明：TREAP-INSERT 的期望运行时间是 $\Theta(\lg n)$。

TREAP-INSERT 先执行一个查找然后做一系列旋转。虽然这两个操作的期望运行时间相等，但是实际的代价不同。查找操作从 treap 中读取信息而不做修改。相反地，旋转会改变 treap 内父结点和孩子指针。在大部分的计算机中，读取操作比写入操作快很多。所以我们希望 TREAP-INSERT 执行很少量的旋转。我们将说明所执行的旋转的期望次数有一个常数界。

为此，需要做一些定义，如图 13-11 所示。一棵二叉查找树 T 的左脊柱是从根部到有最小关键字的结点的路径。换言之，左脊柱是从根部开始只包含左边缘的路径。对称地，T 的右脊柱是从根部开始只包含右边缘的路径。一个脊柱的长度是它所包含的结点数目。

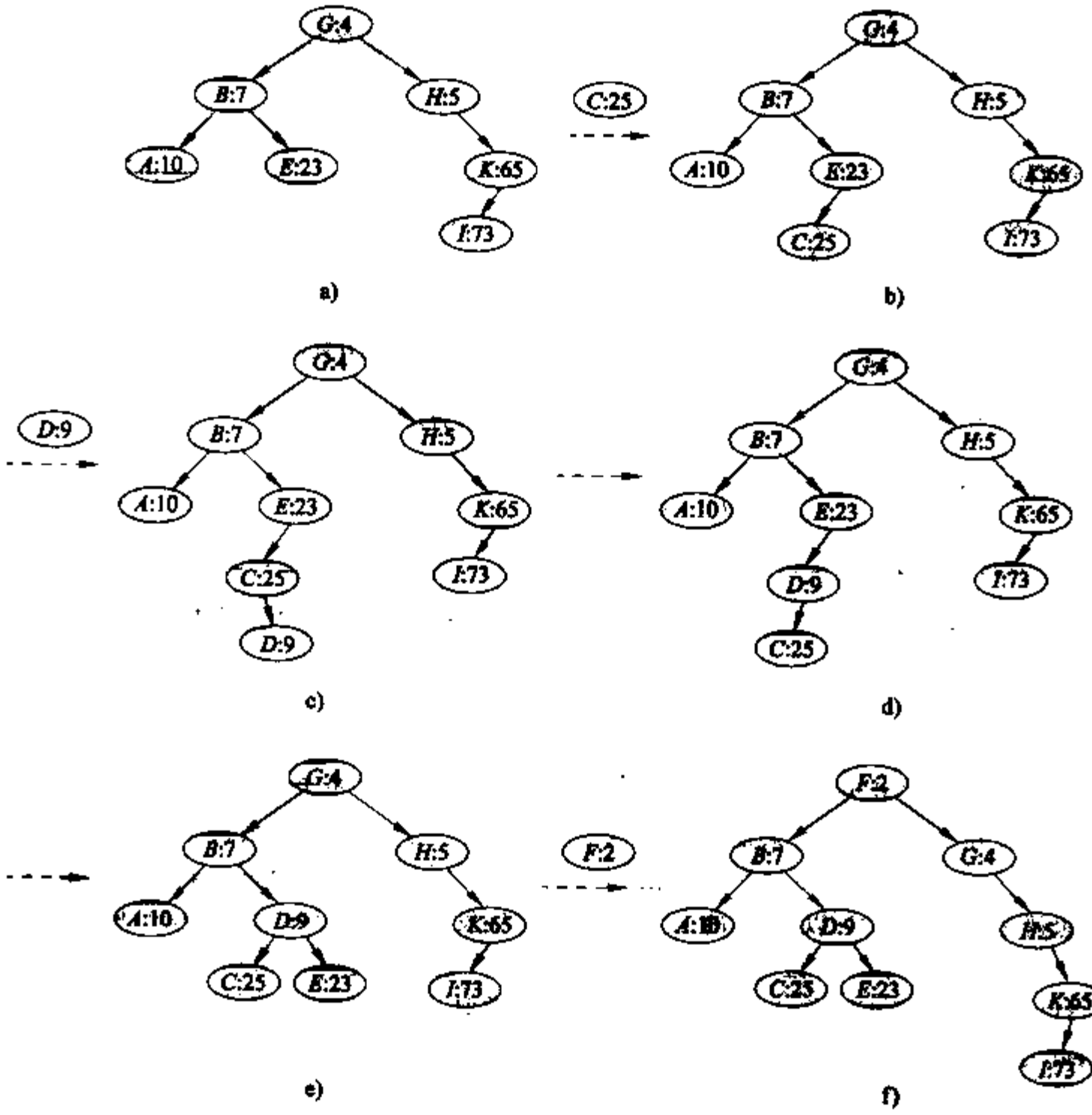


图 13-10　TREAP-INSERT 的操作。a)在插入之前的原始 treap。b)插入一个关键字为 C，优先级为 25 的结点之后的 treap。c)～d)插入一个关键字为 D，优先级为 9 的结点时的中间阶段。e)在 c)和 d)部分的插入完成后的 treap。f)在插入一个关键字为 F，优先级为 2 的结点后的 treap

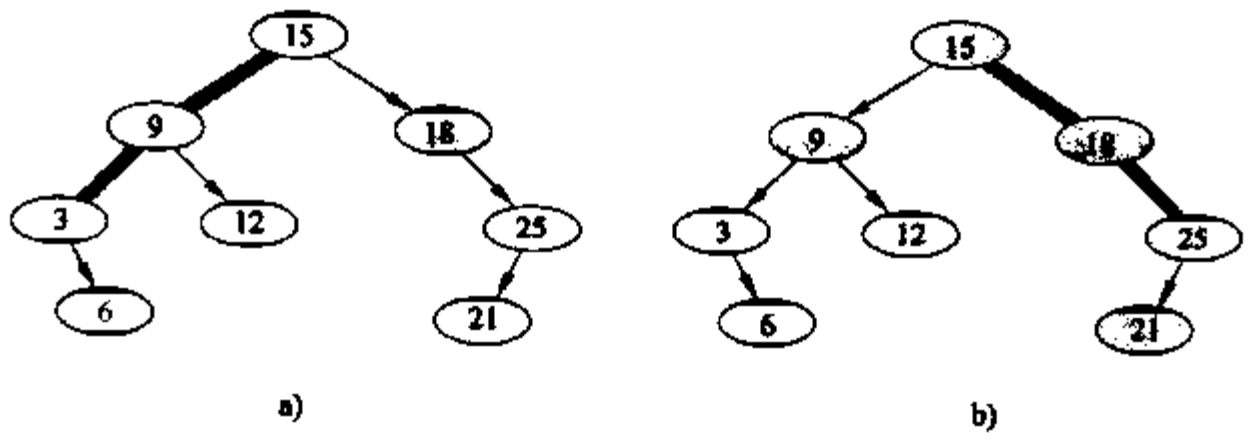


图 13-11　一棵二叉查找树的脊柱。左脊柱在 a)中用阴影来表示，右脊柱在 b)中用阴影来表示

e)考虑利用 TREAP-INSERT 来插入 x 后的 treap T。令 C 为 x 左子树的右脊柱的长度。令 D 为 x 右子树的左脊柱的长度。证明：在插入 x 的期间所执行的旋转的总次数等于 $C+D$。

现在来计算 C 和 D 的期望值。不失一般性，假设关键字为 1，2，…，n，因为只是将它们两两比较。

对结点 x 和 y，$y \neq x$，令 $k=key[x]$ 以及 $i=key[y]$。定义指示器随机变量

$X_{i,k}=I\{y$ 是(T 内)x 左子树的右脊柱$\}$

f)证明 $X_{i,k}=1$ 当且仅当 $priority[y]>priority[x]$，$key[y]<key[x]$，而且对于每个满足 $key[y]<key[z]<key[x]$ 的 z，有 $priority[y]<priority[z]$。

g)证明：$\Pr\{X_{i,k}=1\}=\dfrac{(k-i-1)!}{(k-i+1)!}=\dfrac{1}{(k-i+1)(k-i)}$

h)证明：$E[C]=\sum_{j=1}^{k-1}\dfrac{1}{j(j+1)}=1-\dfrac{1}{k}$

i)利用对称性来证明：$E[D]=1-\dfrac{1}{n-k+1}$

j)总结在将一个结点插入一棵 treap 内时，所执行的旋转的期望次数小于 2。

本章注记

使查找树平衡的想法源自 Adel'son-Vel'skii 和 Landis[2]，他们在 1962 年提出了一类称为“AVL 树”的平衡查找树，如思考题 13-3 所描述。另外一类称为“2-3 树”的查找树由 J. E. Hopcroft 在 1970 年给出(未发表)。在 2-3 树中树的平衡是通过操纵结点的度数来维持的。
298 ~ 300
Bayer 和 McCreight[32]介绍了 2-3 树的一种推广，称为 B 树，有关内容将在第 18 章中介绍。

红黑树是由 Bayer[31]以“对称的二叉 B 树”的名字发明的。Guibas 和 Sedgewick[135]详细研究了它们的性质，并引入了红/黑颜色的约定称呼。Andersson[15]给出了红黑树的简单变种。Weiss[311]把这种变种称为 AA 树。AA 树和红黑树类似，只是左边的孩子永远不能为红色。

treap 是由 Seidel 和 Aragon[271]提出的。它们是 LEDA 内字典的默认实现，LEDA 是一组精心实现的数据结构和算法。

平衡的二叉树有很多其他的变种，包括带权平衡树[230]，k 邻居树[213]以及替罪羊树[108]。或许最有趣的是 Sleator 和 Tarjan[282]所介绍的“伸展树”，它可以“自我调整”。(Tarjan[292]中给出了有关伸展树的详细描述。)伸展树不需要明确的平衡条件如颜色来维持平衡。反之，每次存取时“伸展操作”(包括旋转)在树内执行。在一棵有 n 个结点的树上每个操作的平摊代价(参见第 17 章)是 $O(\lg n)$。

跳表[251]是另外一种平衡的二叉树。跳表是扩充了一些额外指针的链表。在一个包含 n 个
301
元素的跳表上，每一种字典操作都在 $O(\lg n)$ 期望时间内执行。

第 14 章　数据结构的扩张

在有些工程应用环境中，需要一些标准的数据结构，如双链表、散列表或二叉查找树，同时，也有许多应用要求在现有数据结构上有所创新，但很少需要创造出全新的数据结构。通常情况下，只要向标准的数据结构中增加一些信息即可。可以对数据结构编入新的操作，以支持所需的应用。但是，数据结构的扩张并不总是轻而易举的，因为附加的信息还要能为该数据结构上的常规操作所更新和维护。

这一章讨论两种通过扩充红黑树构造的数据结构。14.1 节介绍一种支持一般的动态集合上顺序统计操作的数据结构。有了这种结构，我们就可快速找到一个集合中第 i 小的数，或给出某个元素在集合的全序中的排名。14.2 节对数据结构的扩张过程进行抽象，并提供一个用来简化红黑树扩张的定理。14.3 节利用这个定理帮助设计一种用于维护由区间(如时间区间)构成的动态集合的数据结构。给定一个查询区间，我们可以很快地找到集合中覆盖它的区间。

14.1　动态顺序统计

第 9 章中介绍了顺序统计的概念。例如，在包含 n 个元素的集合中，第 i 个顺序统计量($i=1, 2, \cdots, n$)即为该集合中具有第 i 小关键字的元素。在一个无序的集合中，任意的顺序统计量都可以在 $O(n)$时间内找到。在这一节里，将介绍如何修改红黑树的结构，使得任意的顺序统计量都可以在$O(\lg n)$时间内确定。还将看到，一个元素的排序(即它在集合的线性序中的位置)可以同样地在 $O(\lg n)$时间内确定。

图 14-1 中显示一种能支持快速顺序统计量操作的数据结构。一棵顺序统计量树 T 通过简单地在红黑树的每个结点存入附加信息而成。在一个结点 x 内，除了包含通常的红黑树的域 $key[x]$，$color[x]$，$p[x]$，$left[x]$和 $right[x]$，还包括域 $size[x]$。这个域中包含以结点 x 为根的子树的(内部)结点数(包括 x 本身)，即子树的大小。如果定义哨兵为 0，也就是设置 $size[nil[T]]$为 0，则有等式

$$size[x] = size[left[x]] + size[right[x]] + 1$$

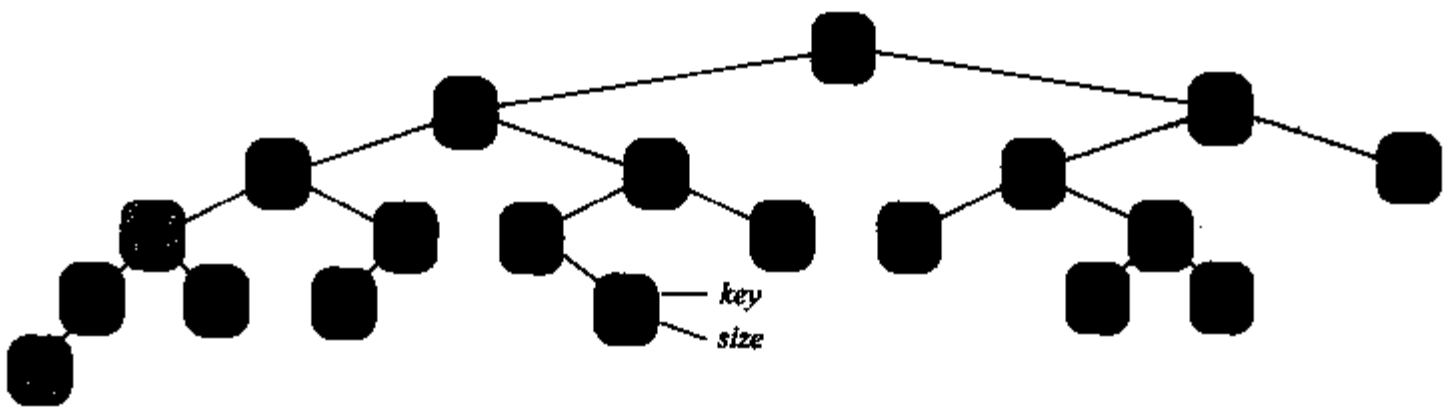


图 14-1　一棵顺序统计树，它是一棵扩充的红黑树。阴影结点为红色的，深色结点为黑色的。除了通常的一些域外，每个结点 x 还有域 $size[x]$，即以 x 为根的子树中结点的个数

在一个顺序统计树中，并不要求关键字互不相同(例如，在图 14-1 的树中，包含两个值为 14 的关键字和两个值为 21 的关键字)。在出现相等关键字的情况下，先前排序的定义不再适用。定义排序为按中序遍历树时输出的结点位置，以此消除顺序统计树原定义的不确定性。如图 14-1

所示，存储在黑色结点的关键字 14 排序为 5，而存储在红色结点的关键字 14 排序为 6。

检索具有给定排序的元素

在说明插入和删除过程中如何来维持规模信息前，先来看看利用这种附加的信息来实现的两个顺序统计查询。先检索具有给定秩的元素的操作。过程 OS-SELECT(x, i)返回一个指向以 x 为根的子树中包含第 i 小关键字的结点的指针。为找出顺序统计树 T 中的第 i 小关键字，调用
303 过程 OS-SELECT($root[T]$, i)。

```
OS-SELECT(x, i)
1  r ← size[left[x]]+1
2  if i=r
3     then return x
4  elseif i<r
5     then return OS-SELECT(left[x], i)
6  else return OS-SELECT(right[x], i−r)
```

OS-SELECT 算法的基本思想与第 9 章中介绍的选择算法的思想差不多。值 $size[left[x]]$表示在对以 x 为根的子树进行中序遍历时排在 x 之前的结点个数。这样，$size[left[x]+1]$即为以 x 为根的子树中 x 的排序。

OS-SELECT 的第 1 行计算以 x 为根的子树中结点 x 的排序 r。如果 $i=r$，结点 x 就是第 i 小元素，返回第 3 行的 x。如果 $i<r$，则第 i 小元素就在 x 的左子树中，故在第 5 行中对 $left[x]$进行递归调用。如果 $i>r$，则第 i 小元素在 x 的右子树中。因为在对以 x 为根的子树进行中序遍历时，共有 r 个元素在 x 的右子树前，故在以 x 为根的子树中第 i 小元素即为以 $right[x]$为根的子树中第$(i-r)$小元素。第 6 行进一步递归地确定这个元素。

为搞清楚 OS-SELECT 的操作过程，考虑如何找出图 14-1 中的顺序统计树中第 17 小元素。开始时以 x 为根，其关键字为 26，$i=17$。因为 26 的左子树的大小为 12，故它的排序为 13。这样，可以断定排序为 17 的结点是 26 的右子树中第 $17-13=4$ 小的元素。在递归调用后，x 为具有关键字 41 的结点，$i=4$。因为 41 的左子树大小为 5，故它的排序为 6。于是，具有排序 4 的结点是 41 的左子树中第 4 小的元素。在递归调用后，x 为具有关键字 30 的结点，在其子树中 x 的排序为 2。这样，再做一次递归调用，就能找到以关键字 38 的结点为根的子树中第 $4-2=2$ 小的元素。现求出它的左子树的大小为 1，这意味着它就是次最小的元素。到此为止，该过程返回一个指向关键字为 38 的结点的指针。

因为每一次递归调用都在顺序统计树中下降了一层，故最坏情况下，OS-SELECT 的总时间与树的高度成正比。又因为该树是棵红黑树，其高度为 $O(\lg n)$，其中 n 为结点个数。所以，对含 n 个元素的动态集合，OS-SELECT 的运行时间为 $O(\lg n)$。

确定一个元素的秩

给定指向一顺序统计树 T 中结点 x 的指针，过程 OS-RANK 返回在对 T 进行中序遍历后得
304 到的线性序中 x 的位置。

```
OS-RANK(T, x)
1  r ← size[left[x]]+1
2  y ← x
3  while y≠root[T]
4       do if y=right[p[y]]
5             then r ← r+size[left[p[y]]]+1
```

```
6        y ← p[y]
7  return r
```

这个算法的工作过程是这样的，x 的秩可以视为在对树的中序遍历中，排在 x 之前的结点个数再加上 1(代表 x 自身)。OS-RANK 保持了以下的循环不变式：

在第 3～6 行中 while 循环的每一次迭代开始时，r 为以结点 y 为根的子树中 $key[x]$的秩。

下面，我们就用这一循环不变式来说明 OS-RANK 能正确地工作：

初始化：在第一次迭代之前，在第 1 行中，置 r 为以 x 为根的子树中 $key[x]$的秩。第 2 行中置$y \leftarrow x$,使得首次执行第 3 行中的测试时，循环不变式为真。

保持：在每一次 while 循环迭代的最后，都要置 $y \leftarrow p[y]$。这样，就必须证明如果 r 是循环体顶部以 y 为根的子树中 $key[x]$的秩，那么 r 也是循环体底部以 $p[y]$为根的子树中 $key[x]$的秩。我们已经对以结点 y 为根的子树在中序遍历下先于 x 的结点个数进行了计数，故要加上以 y 的兄弟为根的子树在中序遍历下先于 x 的结点数；另外，如果 $p[y]$也前于 x，则该计数还要加上 1。如果 y 是个左孩子，则 $p[y]$或 $p[y]$的右子树中的所有结点都不先于 x，r 保持不动；否则，y 是个右孩子，$p[y]$和 $p[y]$的左子树中的所有结点都前于 x。于是在第 5 行中，将当前的 r 值在加上 $size[left[p[y]]]+1$。

终止：当 $y=root[T]$时循环终止，此时以 y 为根的子树是棵完整树。因此，r 值就是这棵整树中 $key[x]$的秩。

现在来看一个例子。当在图 14-1 中的顺序统计树上运行 OS-RANK，以便确定关键字为 38 的结点的秩时，在 while 循环的顶部 $key[y]$和 r 的一系列值如右：

迭代	$key[y]$	r
1	38	2
2	30	4
3	41	4
4	26	17

返回的秩为 17。

while 循环的每一次迭代要花 $O(1)$时间，且 y 在每次迭代中沿树上升一层，故最坏情况下，OS-RANK 的运行时间与树的高度成正比：对含 n 个结点的顺序统计树时间为 $O(\lg n)$。

对子树规模的维护

给定每个结点的 $size$ 域后，OS-SELECT 和 OS-RANK 能迅速计算出所需的顺序统计信息。然而，除非能用红黑树上基本的修改操作对这些 $size$ 域加以有效地维护，否则，就达不到期望的目标。下面就来说明如何在不影响插入和删除的渐近时间的前提下维护子树的规模。

由 13.3 节知道，红黑树上的插入操作包括两个阶段。第一个阶段从根开始，沿着树下降，将新结点插入为某个已有结点的孩子。第二阶段沿树上升，做一些颜色修改和旋转以保持红黑性质。

为在第一阶段中保持子树的规模，对由根至叶子的路径上遍历的每个结点 x，都要增加其 $size[x]$。新增加的结点的 $size$ 为 1。因为所经过的路径上共有 $O(\lg n)$个结点，故维护 $size$ 域的额外代价为 $O(\lg n)$。

在第二阶段中，对红黑树结构上的改变仅是由旋转所致，旋转次数至多为 2。此外，旋转是一种局部操作：它仅会使两个结点的 $size$ 域失效，而旋转操作的支撑链正是与这两个结点关联的。在 13.2 节中给出了 LEFT-ROTATE(T, x)的代码，现增加下面两行：

```
12  size[y] ← size[x]
13  size[x] ← size[left[x]]+size[right[x]]+1
```

图 14-2 说明了 $size$ 域是如何被更新的。对 RIGHT-ROTATE 所做的改动是对称的。

因为向红黑树插入的过程中至多执行两次旋转，所以在第二阶段更新 $size$ 域只需 $O(1)$时间。

305

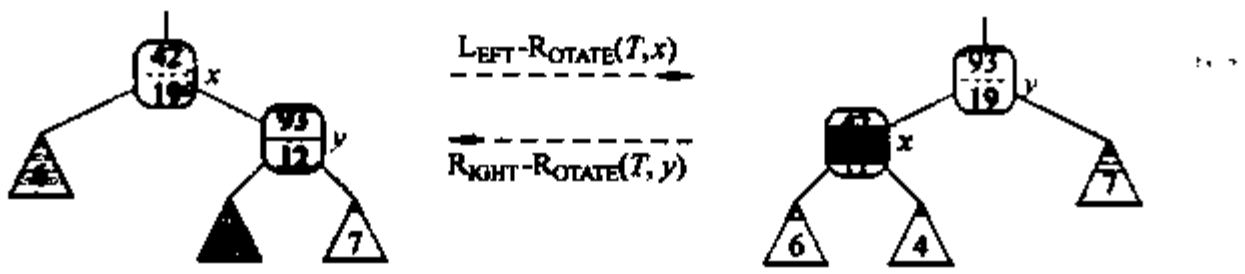


图 14-2　在旋转过程中修改子树大小。要修改的两个 *size* 域与旋转发生时所围绕的链关联。所做的修改是局部的，仅需存储在 x 和 y 中的 *size* 信息，以及图中为三角形的子树的根

306

于是，向一个含 n 个结点的顺序统计树中插入所需的总时间为 $O(\lg n)$，从渐近意义上来看，这与一般的红黑树是相同的。

红黑树上的删除操作同样包含两个阶段：第一阶段对查找树进行操作；第二阶段做至多三次旋转，除此之外，不做任何其他结构上的改动(见 13.4 节)。第一阶段要删除结点 y，为更新子树的规模，可以遍历一条由结点 y 至根的路径，并减小路径上每个结点的 *size* 域。因为在含 n 个结点的红黑树中，这样一条路径的长度为 $O(\lg n)$，故第一阶段中为维护 *size* 域所花的额外时间为 $O(\lg n)$。第二阶段中的 $O(1)$次旋转可以采用与插入同样的方式来处理。这样，对一棵含 n 个结点的顺序统计树，插入操作和删除操作，包括维护 *size* 域，都需 $O(\lg n)$时间。

练习

14.1-1　说明 OS-SELECT(T, 10)作用于图 14-1 中红黑树 T 的过程。

14.1-2　说明 OS-RANK(T, x)作用于图 14-1 中红黑树 T 及关键字 $key[x]$为 35 的结点 x 上的过程。

14.1-3　写出 OS-SELECT 的非递归形式。

14.1-4　写出一个递归过程 OS-KEY-RANK(T, k)，使之以一棵顺序统计树 T 和某个关键字 k 为输入，返回在由 T 表示的动态集合中 k 的秩。假设 T 的所有关键字都是不同的。

14.1-5　给定含 n 个元素的顺序统计树中的一个元素 x 和一个自然数 i，如何在 $O(\lg n)$时间内，确定 x 在该树的线性序中第 i 个后继？

14.1-6　在 OS-SELECT 或 OS-RANK 中，每次引用结点的 *size* 域都仅是为了计算在以结点为根的子树中该结点的秩。假设我们将每个结点的秩(对于以该结点为根的子树而言)存于该结点自身之中。说明在插入和删除时如何来维护这个信息(注意这两种操作可能引起旋转)。

307

14.1-7　说明如何在 $O(n \lg n)$的时间内，利用顺序统计树对大小为 n 的数组中的逆序对(见思考题 2-4)进行计数。

***14.1-8**　现有一个圆上的 n 条弦，每条弦都是按其端点来定义的。请给出一个能在 $O(n \lg n)$时间内确定圆内相交弦的对数(例如，如果 n 条弦都是直径，它们相交于圆心，则正确的答案为 $\binom{n}{2}$)。假设任意两条弦都不会共享端点。

14.2　如何扩张数据结构

在算法设计过程中，常常需要对基本的数据结构进行扩张，以便支持一些新的功能。在下一节中，我们将再一次通过对数据结构进行扩张的方法，来设计一种支持区间操作的数据结构。本节就来讨论这种扩张过程的各个步骤，证明一个定理；在许多情况下，该定理允许我们很容易地扩张红黑树。

对一种数据结构的扩张过程可分为四个步骤：

1)选择基础数据结构

2)确定要在基础数据结构中添加哪些信息

3)验证可用基础数据结构上的基本修改操作来维护这些新添加的信息

4)设计新的操作

以上只是给出了一个一般模式，读者不应盲目地按照上面给出的顺序来执行这些步骤。许多设计过程采用的是试探法，所有的步骤通常都并行地进展。例如，如果不能有效地维护附加信息的话，就没必要确定附加信息以及设计新操作(步骤 2 和步骤 4)。然而，这四步方法使读者在扩张数据结构时能集中注意力，并使思路清晰。

在 14.1 节中设计顺序统计树时，我们就依照了这些步骤。在步骤 1 中，我们选择红黑树作为基础数据结构。之所以选择这种数据结构，是因为它能有效地支持一些基于全序的动态集合 308
操作，如 MINIMUM，MAXIMUM，SUCCESSOR 和 PREDECESSOR。

在步骤 2 中，我们提供了 *size* 域，它可以存放各结点 x 子树的规模信息。一般地，附加信息可使得各类操作更加有效。例如，我们本可以仅用树中存储的关键字来实现 OS-SELECT 和 OS-RANK，但这种实现的运行时间就不止是 $O(\lg n)$了。有些时候，附加信息是指针类信息，而不是具体的数据(见练习 14.2-1)。

在步骤 3 中，我们保证了插入和删除操作仍能在 $O(\lg n)$时间内维护 *size* 域。比较理想的是对原有数据结构几个元素略作改动，就足以维护附加的信息。例如，如果把每个结点的秩存在树中，则 OS-SELECT 和 OS-RANK 过程能较快地运行，但要插入一个新的最小元素，则会使树中每个结点的秩发生变化。如果我们存储的是子树的规模，则插入一个新元素的操作仅影响到 $O(\lg n)$个结点信息改变。

在步骤 4 中，设计了新操作 OS-SELECT 和 OS-RANK。之所以要对数据结构进行扩张，首要原因就是需要新的操作。有时，我们并不设计新的操作，而是利用附加信息来加速已有操作的执行速度(见练习 14.2-1)。

对红黑树的扩张

当红黑树被选作基础数据结构时，可以证明，某些类型的附加信息总是可以用插入和删除来进行有效地维护，从而使得第 3 步非常易于实现。下面定理的证明与 14.1 节用顺序统计树来维护 *size* 域的思路类似。

定理 14.1(红黑树的扩张)　设域 f 对含 n 个结点的红黑树进行扩张的域，且假设某结点 x 的域 f 的内容可以仅用结点 x，$left[x]$和 $right[x]$中的信息计算，包括 $f[left[x]]$和 $f[right[x]]$。这样，在插入和删除操作中，我们可以在不影响这两个操作 $O(\lg n)$渐近性能的情况下，对 T 的所有结点的 f 值进行维护。

证明：证明的主要思想是对树中某结点 x 的 f 域的改动会波及 x 在树中的祖先。亦即，修改 $f[x]$可能导致 $f[p[x]]$的更新，改变 $f[p[x]]$也可能引起 $f[p[p[x]]]$的改变；沿树继续下去。当 $f[root[T]]$被更新时，不再有别的结点要依赖其新值，这个过程就结束了。因为一棵红黑树 309
的高度为 $O(\lg n)$，改变某结点的 f 域就要再花 $O(\lg n)$的时间来更新被该修改所影响的结点。

将一个结点 x 插入 T 的过程包括两个阶段(见 13.3 节)。在第一阶段中，x 被插入为一个现存结点 $p[x]$的孩子。$f[x]$的值可在 $O(1)$时间内计算出。因为根据假设，$f[x]$仅依赖于 x 本身的其他域中的信息和 x 的子结点中的信息，但这时 x 的两个子结点都是标志 $nil[T]$。一旦 $f[x]$被求出后，这个变化就沿树向上传播。这样，插入的第一阶段的总时间为 $O(\lg n)$。在第二阶段中，

对树结构的仅有改变来源于旋转操作。在一次旋转中只有两个结点发生变化，故每次旋转中更新 f 域的总时间为 $O(\lg n)$。又因为插入操作中的旋转次数至多为 2，故插入的总时间为 $O(\lg n)$。

和插入一样，删除操作也包括两个阶段(见 13.4 节)。在第一阶段中，如果被删除的结点由其后继替代，当删除的结点或其后继被删掉时，则树发生变化。这些变化波及到 f 的代价至多为 $O(\lg n)$，因为这些变化对树影响是局部的。第二阶段对红黑树的修复至多需要三次旋转，且每次旋转至多需要 $O(\lg n)$ 时间就可将变化传播到 f。这样，删除的总时间为 $O(\lg n)$。(证毕) ■

在许多情况中(例如在顺序统计树中对 *size* 域的维护)，在一次旋转后进行更新的代价是 $O(1)$，而并不是定理 14.1 中所给出的 $O(\lg n)$。练习 14.2-4 给出了一个例子。

练习

14.2-1 说明如何能在扩张的顺序统计树上，以最坏情况 $O(1)$ 的时间来支持动态集合查询操作 MINIMUM，MAXIMUM，SUCCESSOR 和 PREDECESSOR。顺序统计树上的其他操作的渐近性能应不受影响。(提示：为结点增加指针。)

14.2-2 能否在不影响任何红黑树操作性能的前提下，将结点的黑高度作为一个域来维护？如果可以的话，说明怎样做；如果不能，说明为什么。

14.2-3 能否将红黑树中结点的深度作为一个域来进行有效地维护？如果可以的话，说明怎样做；如果不能，说明为什么。

310

*14.2-4 设$\otimes$为一个满足结合律的二元运算符，另外，设 a 为一棵红黑树的每一结点中的一个域。假设在每个结点 x 中增加一个域 f，使 $f[x]=a[x_1]\otimes a[x_2]\otimes\cdots\otimes a[x_m]$，其中 $x_1, x_2, \cdots, x_m$ 是以 x 为根的子树中，按中序排列的所有结点。证明在一次旋转后，可以在 $O(1)$ 时间里对 f 域作出合适的修改。对扩张稍作修改，证明顺序统计树 *size* 域的每次旋转的维护时间为 $O(1)$。

*14.2-5 希望通过增加操作 RB-ENUMERATE(x, a, b)来扩张红黑树。该操作输出所有的关键字 k，使在以 x 为根的红黑树中有 $a\leqslant k\leqslant b$。描述如何在 $\Theta(m+\lg n)$ 时间里实现 RB-ENUMERATE，其中 m 为输出的关键字数，n 为树中的内部结点。(提示：不需要向红黑树中增加新的域。)

14.3　区间树

在这一节里，我们将扩张红黑树以支持由区间构成的动态集合上的操作。一个闭区间是一个实数的有序对 $[t_1, t_2]$，其中 $t_1\leqslant t_2$。区间 $[t_1, t_2]$ 表示了集合 $\{t\in\mathbf{R}: t_1\leqslant t\leqslant t_2\}$。开区间和半开区间分别略去了集合的两个或一个端点。在这一节里，我们假设区间都是闭的(将结果推广至开和半开区间上比较简单)。

区间可以很方便地表示各占用一段连续时间的一些事件。例如，查询一个由时间区间数据构成的数据库，以找出特定的区间内发生了什么事件。这一节中介绍的数据结构可用来有效地维护这样一个区间数据库。

我们可以把一个区间 $[t_1, t_2]$ 表示成一个对象 i，其各个域为 $low[i]=t_1$(低端点)，$high[i]=t_2$(高端点)。我们说区间 i 和 i' 重叠，如果 $i\cap i'\neq\varnothing$，亦即，如果 $low[i]\leqslant high[i']$，且 $low[i']\leqslant high[i]$。任意两个区间 i 和 i' 满足区间三分法，也就是下列三种性质之一：

a) i 和 i' 重叠

b) i 在 i' 左边(也就是 $high[i]<low[i']$)

c) i 在 i' 右边(也就是 $high[i']<low[i]$)

图 14-3 示出了这三种情况。

311

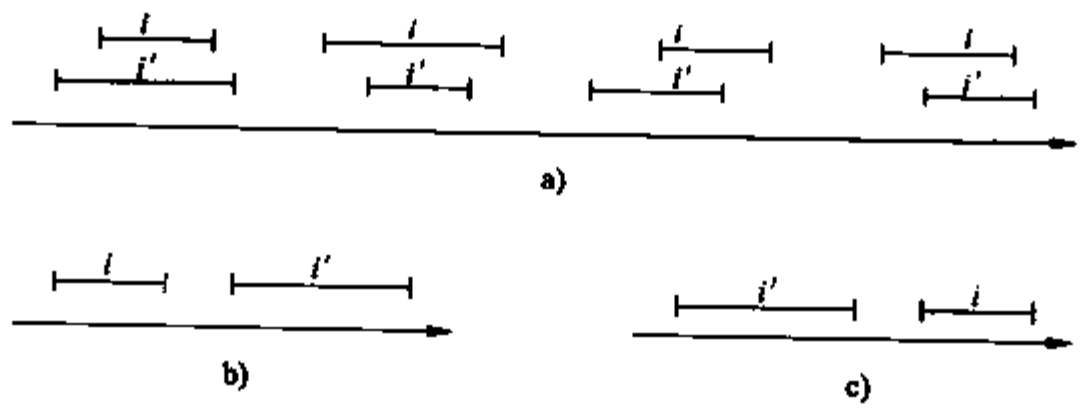


图 14-3　两个闭区间 i 和 i' 的区间三分法。a)如果 i 和 i' 重叠，则共有四种情况，每种情况下，$low[i] \leqslant high[i']$，且 $low[i'] \leqslant high[i]$。b)区间没有重叠且 $high[i] < low[i']$。c)区间没有重叠且 $high[i'] < low[i]$

区间树是一种对动态集合进行维护的红黑树，该集合中的每个元素 x 都包含一个区间 $int[x]$。区间树支持下列操作：

INTERVAL-INSERT(T, x)：将包含区间域 int 的元素 x 插入到区间树 T 中。

INTERVAL-DELETE(T, x)：从区间树 T 中删除元素 x。

INTERVAL-SEARCH(T, i)：返回一个指向区间树 T 中元素 x 的指针，使 $int[x]$ 与 i 重叠；若集合中无此元素存在，则返回 $nil[T]$。

图 14-4 说明了区间树是如何表示一个区间集合的。下面要按 14.2 节中的四步方法，来分析区间树以及区间树上各种操作的设计。

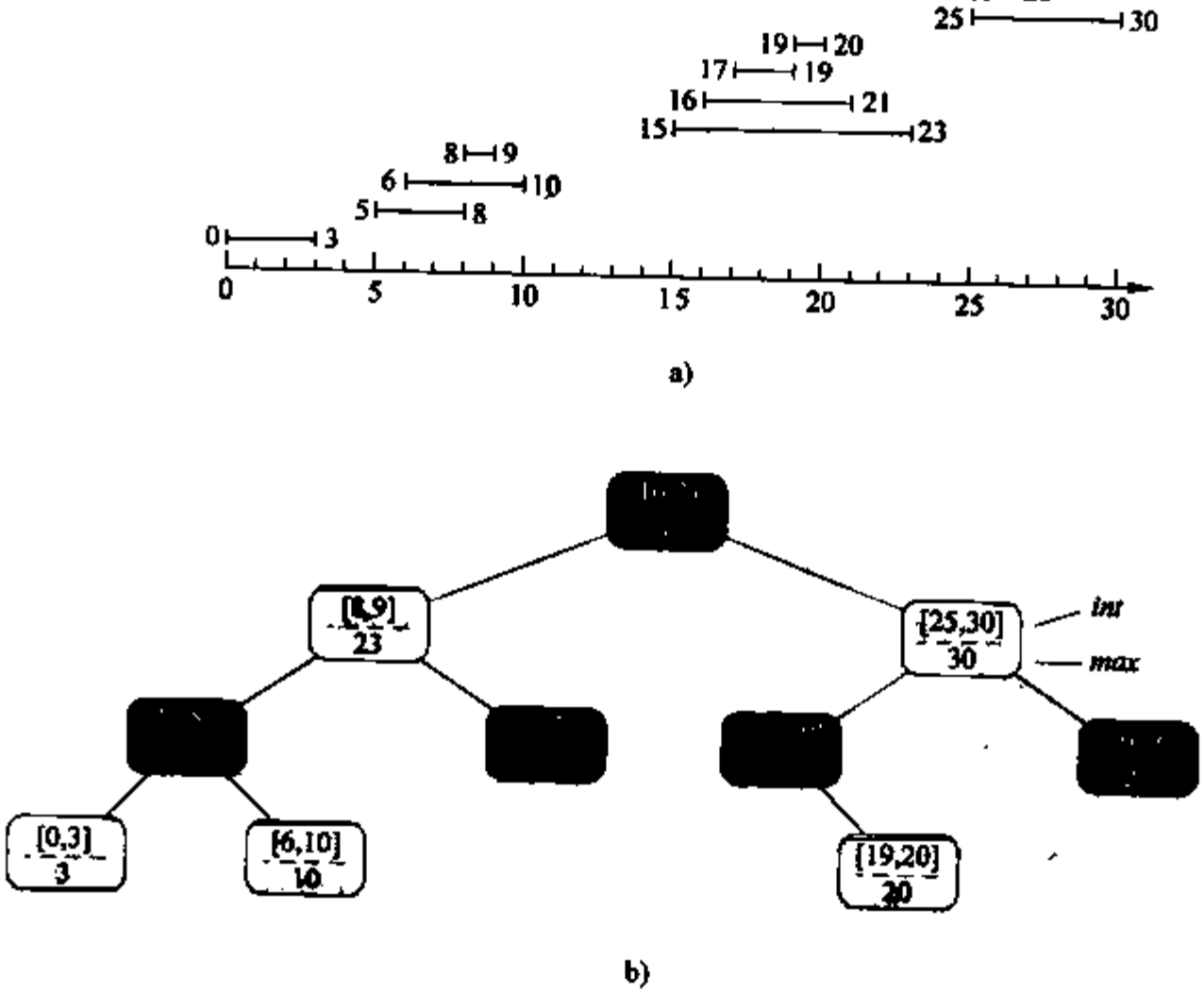


图 14-4　一棵区间树。a)十个区间按左端点自底向上顺序示出。b)表示它们的区间树。对该树做中序遍历，即可按左端点顺序列出各个结点

步骤 1：基础数据结构

我们选择的基础数据结构为这样一种红黑树，其中每个结点 x 包含一个区间域 $int[x]$，x 的关键字为区间的低端点 $low[int[x]]$。这样，对树进行中序遍历就可按低端点的次序列出各区间。

步骤 2：附加信息

每个结点 x 中除了区间信息外，还包含一个值 $max[x]$，即以 x 为根的子树中所有区间的端点的最大值。

步骤 3：对信息的维护

必须验证对含 n 个结点的区间树的插入和删除能在 $O(\lg n)$ 时间内完成。给定区间 $int[x]$ 和 x 的子结点的 max 值，可以确定 $max[x]$：

$$max[x] = \max(high[int[x]], max[left[x]], max[right[x]])$$

312 / 313

这样，根据定理 14.1 可知，插入和删除操作的运行时间为 $O(\lg n)$。事实上，在一次旋转后，更新 max 域只需 $O(1)$ 时间，如练习 14.2-4 和练习 14.3-1 中所说明的那样。

步骤 4：设计新的操作

我们唯一需要的新操作是 INTERVAL-SEARCH(T, i)，它用来找出树 T 中覆盖区间 i 的那个结点。如果树中覆盖 i 的区间不存在，则返回指向哨兵 $nil[T]$ 的指针。

```
INTERVAL-SEARCH(T, i)
1  x ← root[T]
2  while x≠nil[T] and i does not overlap int[x]
3       do if left[x]≠nil[T] and max[left[x]]≥low[i]
4            then x ← left[x]
5            else x ← right[x]
6  return x
```

查找与 i 交叠的区间 x 的过程先以 x 为树根开始，逐步下降。当找到一个重叠区域或 x 指向了 $nil[T]$ 时，该过程结束。因为基本循环的每次迭代要花 $O(1)$ 时间，又含 n 个结点的红黑树的高度为 $O(\lg n)$，故 INTERVAL-SEARCH 过程的时间为 $O(\lg n)$。

在说明 INTERVAL-SEARCH 的正确性之前，先来看看它作用于图 14-4 中区间树上的过程。假设想找到一个与区间 i=[22，25]重叠的区间。开始时 x 为根结点，它包含[16，21]，并不与 i 重叠。因为 $max[left[x]]$=23 大于 $low[i]$=22，所以把 x 作为根的左孩子继续循环。现在 x 包含[8，9]，仍不与 i 重叠。这一次，$max[left[x]]$=10 小于 $low[i]$=22，故将 x 的右孩子作为新的 x 继续循环。这个结点所包含的区间[15，23]与 i 重叠，则过程结束并返回这个结点。

现在来看一个不成功查找的例子。假设想在图 14-4 的区间树中找出与 i=[11，14]重叠的区间，开始时以 x 为根。因为根的区间[16，21]不与 i 重叠，又 $max[left[x]]$=23 大于 $low[i]$=11，则转向左面的包含[8，9]的结点(注意右子树中没有一个区间与 i 重叠，其原因将在后面说明)。区间[8，9]不与 i 重叠，且 $max[left[x]]$=10 小于 $low[i]$=11，因而转向右子树(注意其左子树中没有一个区间与 i 重叠)。区间[15，23]也不与 i 重叠，且它的左孩子为 $nil[T]$，故向右转，循环结束，返回 $nil[T]$。

要理解 INTERVAL-SEARCH 的正确性，就要理解为什么只检查一条由根开始的路径就足够了。该过程的基本思想是在任意结点 x 上，如果 $int[x]$ 不与 i 重叠，则查找总是沿着一个安全的方向前进的：如果树中确有一个与 i 重叠的区间，则该区间必定会被找到。下面的定理更准确

314

地陈述了这个性质。

定理 14.2　INTERVAL-SEARCH(T, i)的任意一次执行都将或者返回一个其区间覆盖了 i

的结点，或者返回 $nil[T]$，此时树 T 中没有哪一个结点的区间覆盖了 i。

证明：当 $x=nil[T]$或 i 覆盖 $int[x]$，第 2～5 行的循环终止。在后一种情况，返回 x 当然是正确的。那么，我们主要考虑第一种情况，当 $x=nil[T]$时 while 循环终止的情况。

对第 2～5 行的 while 循环使用如下循环不变式：

如果树 T 包含覆盖 i 的区间，则以 x 为根的子树必包含此区间。

循环不变式使用如下：

初始化：在第一次迭代之前，第 1 行将 x 设为 T 的根，因此循环不变式是成立的。

保持：在 while 循环的每次迭代中，第 4 行或第 5 行将被执行。将证明循环不变式在各种情况下都能保持成立。

如果第 5 行执行，由于第 3 行的分支条件，有 $left[x]=nil[T]$或 $max[left[x]]<low[i]$。如果 $left[x]=nil[T]$，则以 $left[x]$为根的子树显然不包含覆盖 i 的区间，所以置 x 为 $right[x]$以维护这个不变式。所以，假设 $left[x]\neq nil[T]$且 $max[left[x]]<low[i]$。如图 14-5a 所示，对 x 左子树的任一区间 i'，有

$$high[i'] \leqslant max[left[x]] < low[i]$$

根据区间三分法，i'和 i 不重叠。因此，x 的左子树不包含覆盖 i 的区间，置 x 为 $right[x]$可以使循环不变式保持成立。

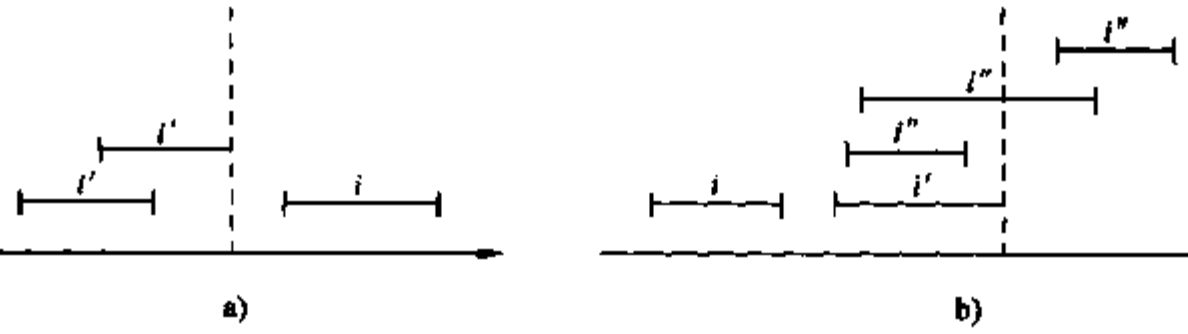


图 14-5　在定理 14.2 的证明中用到的各个区间。在每种情况下，$max[left[x]]$的值用虚线表示。a)向右查找。在 x 的左子树中没有与 i 重叠的区间 i'。b)向左查找。x 的左子树中包含与 i 重叠的区间（此状态未显示），或者 x 左子树中有一个区间满足 $high[i']=max[left[x]]$。既然 i 与 i'不重叠，则与 x 右子树任意区间 i''都不重叠，因为 $low[i']\leqslant low[i'']$

315

如果是第 4 行被执行，我们将证明循环不变式的对换性。也就是说，如果在以 $left[x]$为根的子树中没有覆盖 i 的区间，则树的其他部分也不会包含这样的区间。当第 4 行被执行时，由于第 3 行的分支条件，有 $max[left[x]]\geqslant low[i]$。根据 max 域的定义，在 x 的左子树中必定存在某区间 i'满足

$$high[i'] = max[left[x]] \geqslant low[i]$$

（图 14-5b 示出了这种情况）。因为 i 和 i'不重叠，又 $high[i']<low[i]$不成立，根据区间三分法有 $high[i]<low[i']$。区间树是以区间的低端点为关键字的，所以查找树性质隐含了对 x 的右子树中的任何区间 i''，有

$$high[i] < low[i'] \leqslant low[i'']$$

根据区间三分法，i 和 i''不重叠。得出这样的结论，即不管 x 的左子树中是否存在覆盖 i 的区间，置 x 为 $left[x]$将使循环不变式保持成立。

终止：如果循环在 $x=nil[T]$时终止，则表明在以 x 为根的子树中，没有覆盖 i 的区间。循环不变式的对换式说明 T 中不包含覆盖 i 的区间，所以返回 $x=nil[T]$是正确的。（证毕）■

因此，INTERVAL-SEARCH 过程是正确的。

练习

14.3-1　写出作用于区间树的结点、并于 $O(1)$ 时间内更新 max 域的 LEFT-ROTATE 的伪代码。

316 **14.3-2**　重写 INTERVAL-SEARCH 代码，使得当所有的区间都是开区间时，它也能正确地工作。

14.3-3　请给出一个有效的算法，使对给定的区间 i，它返回一个与 i 重叠的、具有最小低端点的区间；或者，当这样的区间不存在时返回 $nil[T]$。

14.3-4　给定一个区间树 T 和一个区间 i，请描述如何能在 $O(\min(n, k\lg n))$ 时间内，列出 T 中所有与 i 重叠的区间，此处 k 为输出区间数。（可选：找出一种不修改树的方法。）

14.3-5　请说明要对前面介绍的有关区间树的过程作哪些修改，才能支持操作 INTERVAL-SEARCH-EXACTLY(T, i)，它返回一个指向区间树 T 中结点 x 的指针，使 $low[int[x]]=low[i]$，$high[int[x]]=high[i]$，或当 T 不包含这样的结点时返回 $nil[T]$。所有操作对于 n 结点树（包括 INTERVAL-SEARCH-EXACTLY）的运行时间应为 $O(\lg n)$。

14.3-6　请说明如何来维护一个支持操作 MIN-GAP 的动态数集 Q，使该操作能给出 Q 中最近的两个数之间的差幅。例如，如 $Q=\{1, 5, 9, 15, 18, 22\}$，则 MIN-GAP($Q$)返回 $18-15=3$，因为 15 和 18 为 Q 中最近的两数。使操作 INSERT，DELETE，SEARCH 和 MIN-GAP 尽可能高效，并分析它们的运行时间。

*14.3-7　VLSI 数据库通常将一块集成电路表示成一组矩形。假设每个矩形的边都平行于 x 轴或 y 轴，因而矩形的表示中有最小和最大的 x 和 y 坐标。请给出一个能在 $O(\lg n)$ 时间里确定一组矩形中是否有两个重叠的算法。你给出的算法不一定要输出所有相交的矩形，但
317 要能在一个矩形完全被另一个覆盖时给出正确的判断。（提示：将一条线移过所有的矩形）

思考题

14-1　最大重叠点

假设希望对一组区间记录一个最大重叠点，亦即覆盖它的区间最多的那个点。

a）证明：最大重叠点总存在于某段的端点上。

b）设计一数据结构，能有效地支持操作 INTERVAL-INSERT，INTERVAL-DELETE 和返回最大重叠点操作 FIND-POM。（提示：将所有端点组织成红黑树。左端点关联 $+1$ 值，而右端点关联 -1 值。附加一些维护最大重叠点的信息以扩张树中结点。）

14-2　Josephus 排列

Josephus 问题的定义如下：假设 n 个人排成环形，且有一正整数 $m\leqslant n$。从某个指定的人开始，沿环报数，每遇到第 m 个人就让其出列，且报数进行下去。这个过程一直进行到所有人都出列为止。每个人出列的次序定义了整数 1，2，…，n 的(n, m)-*Josephus* 排列。例如，(7, 3)-Josephus 排列为$\langle 3, 6, 2, 7, 5, 1, 4\rangle$。

a）假设 m 为常数。请描述一个 $O(n)$ 时间的算法，使之对给定的整数 n，输出(n, m)-Josephus 排列。

b）假设 m 不是个常数。请描述一个 $O(n\lg n)$ 时间的算法，使给定的整数 n 和 m，输出(n, m)-Josephus 排列。

本章注记

在 Preparata 和 Shamos[247]的书中，描述了一些出现在 H. Edelsbrunner（1980 年）和 E. M. Mc-Creight(1981 年)所著文献内的区间树。该书详细介绍了一种区间树，当给定包含 n 个
318 区间的静态数据库后，能在 $O(k+\lg n)$ 时间内，将所有与指定查询区间重叠的 k 个区间列出。

第四部分　高级设计和分析技术

导　论

这一部分将介绍设计和分析高效算法的三种重要技术：动态规划(第 15 章)、贪心算法(第 16 章)和平摊分析(第 17 章)。本书前面三部分介绍了一些可以普遍应用的技术，如分治法、随机化和递归求解。这一部分的新技术要更复杂一些，但它们对有效地解决很多计算问题来说很有用。这部分的主题在本书的后面还要作进一步讨论。

动态规划通常应用于最优化问题，即要做出一组选择以达到一个最优解。在做选择的同时，经常出现同样形式的子问题。当某一特定的子问题可能出自于多于一种选择的集合时，动态规划是很有效的；关键技术是存储这些子问题每一个的解，以备它重复出现。第 15 章说明如何利用这种简单思想，将指数时间的算法转化为多项式时间的算法。

像动态规划算法一样，贪心算法通常也是应用于最优化问题。在这种算法中，要做出一组选择以达到一个最优解。该算法的思想是以局部最优的方式来做每一个选择。一个简单的例子是找换硬币的问题：在找换一定数额的硬币时，为使所需的硬币数最少，只要重复地选择不大于不足数额的最大面值的硬币。对许多问题来说，采用贪心法可以比用动态规划更快地给出一个最优解。但是，不容易判断贪心法是否一定有效。第 16 章回顾拟阵理论，它通常可以用来帮助做出这种判断。

319 ~ 321

平摊分析是一种用来分析执行一系列类似操作的算法的工具。在这种方法中，不是通过分别计算每一次操作的真实代价界确定一系列操作代价的界，而是对整个操作序列的真实代价限界。这种方法之所以奏效，原因之一是在一个操作序列中，不可能每一个都以其已知的最坏情况时间界运行。可能某些操作的代价高些，而其他的则可能低些。平摊分析不仅仅是一种分析工具，它也是算法设计的一种思维方式，因为算法的设计和对其运行时间的分析经常是紧密相连的。第 17 章介绍对算法进行平摊分析的三种方法。

322

第 15 章　动 态 规 划

和分治法一样，动态规划(dynamic programming)是通过组合子问题的解而解决整个问题的。(此处"programming"是指一种规划，而不是指写计算机代码。)从第 2 章已经知道，分治法算法是指将问题划分成一些独立的子问题，递归地求解各子问题，然后合并子问题的解而得到原问题的解。与此不同，动态规划适用于子问题不是独立的情况，也就是各子问题包含公共的子子问题。在这种情况下，若用分治法则会做许多不必要的工作，即重复地求解公共的子子问题。动态规划算法对每个子子问题只求解一次，将其结果保存在一张表中，从而避免每次遇到各个子问题时重新计算答案。

动态规划通常应用于最优化问题。此类问题可能有很多种可行解。每个解有一个值，而我们希望找出一个具有最优(最大或最小)值的解。称这样的解为该问题的"一个"最优解(而不是"确定的"最优解)，因为可能存在多个取最优值的解。

动态规划算法的设计可以分为如下 4 个步骤：

1)描述最优解的结构。

2)递归定义最优解的值。

3)按自底向上的方式计算最优解的值。

4)由计算出的结果构造一个最优解。

第 1～3 步构成问题的动态规划解的基础。第 4 步在只要求计算最优解的值时可以略去。如果
323 的确做了第 4 步，则有时要在第 3 步的计算中记录一些附加信息，使构造一个最优解变得容易。

接下来的各节利用动态规划方法来求解一些最优化问题。15.1 节分析包括两个汽车装配线的调度问题，在经过每个装配站后，组装中的汽车可以留在同一条装配线上，或者移动到另外一条装配线。15.2 节讨论如何做一连串的矩阵乘法，使得所做的标量乘法总次数最少。在给出这些动态规划的例子之后，15.3 节讨论为使动态规划成为可行的求解技术，一个问题必须具备的两个关键特征。然后，15.4 节介绍如何找出两个序列的最长公共子序列。最后，15.5 节介绍在已知待搜索的关键字分布的情况下，如何利用动态规划构造最优的二叉查找树。

15.1　装配线调度

第一个动态规划的例子是求解一个制造问题。Colonel 汽车公司在有两条装配线的工厂内生产汽车，如图 15-1 所示。一个汽车底盘在进入每一条装配线后，在一些装配站中会在底盘上安装部件，然后，完成的汽车在装配线的末端离开。每一条装配线上有 n 个装配站，编号为 $j=1, 2, \cdots, n$。将装配线 i(i 为 1 或 2)的第 j 个装配站表示为 $S_{i,j}$。装配线 1 的第 j 个站($S_{1,j}$)和装配线 2 的第 j 个站($S_{2,j}$)执行相同的功能。然而，这些装配站是在不同的时间建造的，并且采用了不同的技术，因此，每个站上所需的时间是不同的，即使是在两条不同装配线相同位置的装配站上也是这样。我们把在装配站 $S_{i,j}$ 上所需的装配时间记为 $a_{i,j}$。如图 15-1 所示，一个汽车底盘进入其中一条装配线，然后从每一站进行到下一站。底盘进入装配线 i 的进入时间为 e_i，装配完的汽车离开装配线 i 的离开时间为 x_i。

在正常情况下，一旦一个底盘进入一条装配线后，它只会经过该条装配线。在相同的装配线中，从一个装配站到下一个装配站所花的时间可以忽略。偶尔会来一个特别急的订单，客户要求尽可能快地制造这些汽车。对这些加急的订单，底盘仍然依序经过 n 个装配站，但是工厂经理可以将部分完成的汽车在任何装配站上从一条装配线移到另一条装配线上。把已经通过装配站 $S_{i,j}$

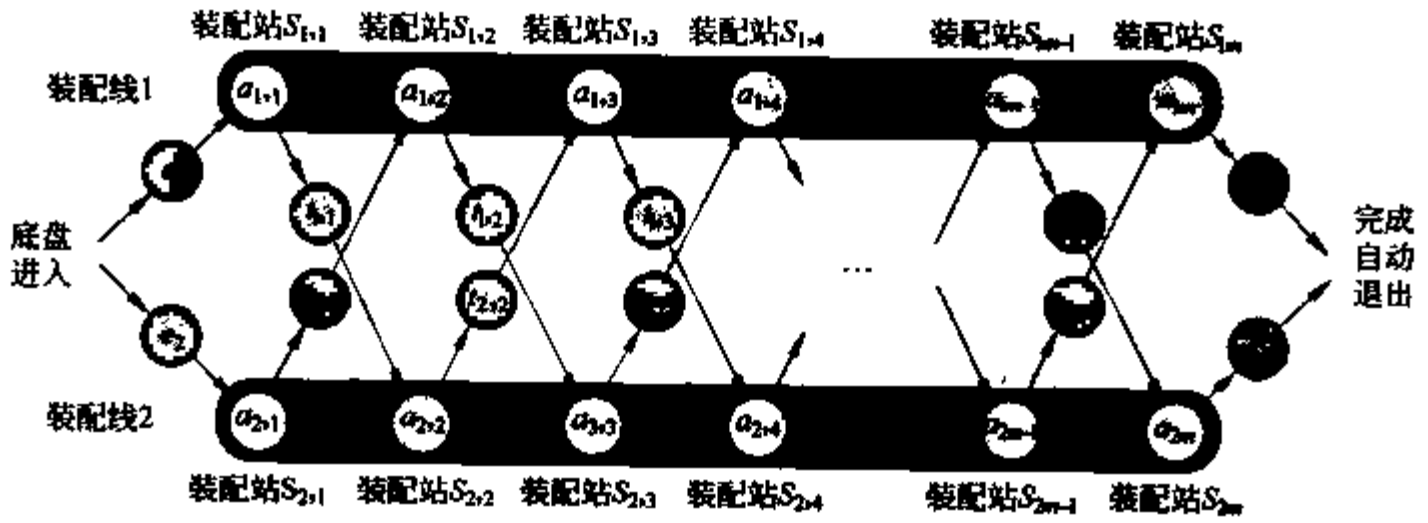
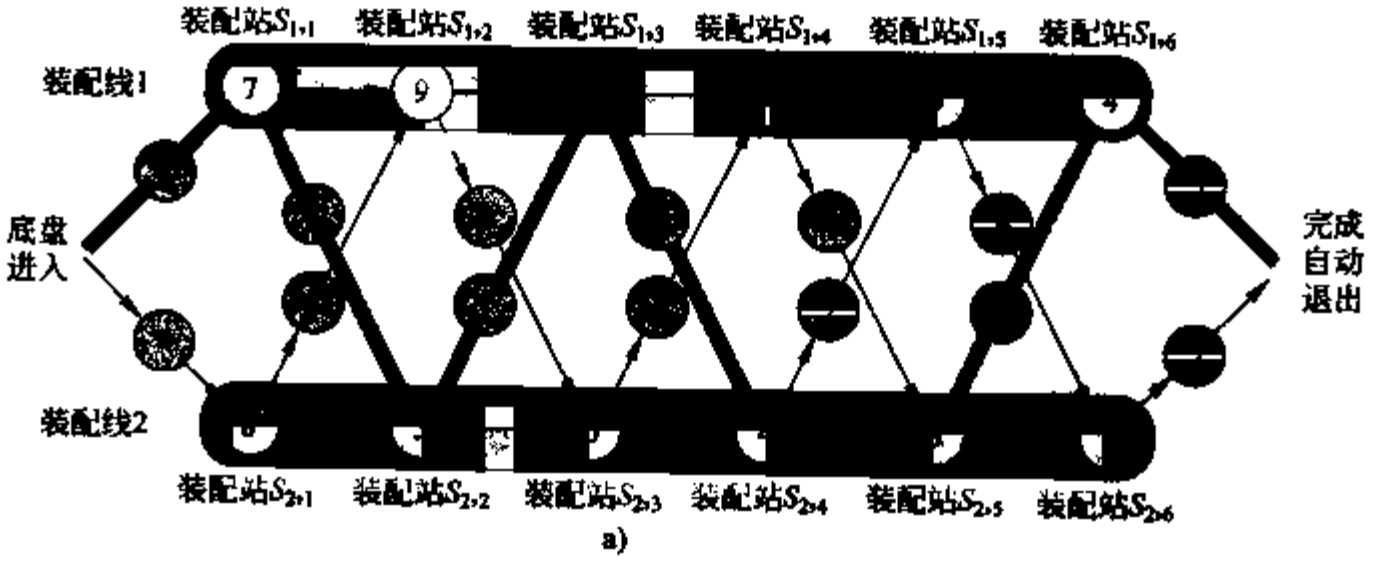


图 15-1　一个找出通过工厂装配线的最快方式的制造问题。共有两条装配线，每条有 n 个装配站；装配线 i 的第 j 个装配站表示为 $S_{i,j}$，在该站的装配时间是 $a_{i,j}$。一个汽车底盘进入工厂，然后进入装配线 i(i 为 1 或 2)，花费时间 e_i。在通过一条线的第 j 个装配站后，这个底盘来到任一条线的第(j +1)个装配站。如果它留在相同的装配线，则没有移动的开销；但是，如果在装配站 $S_{i,j}$ 后，它移动到了另一条线上，则花费时间 $t_{i,j}$。在离开一条线的第 n 个装配站后，完成的汽车花费时间 x_i 离开工厂。待求解的问题是确定应该在装配线 1 内选择哪些站、在装配线 2 内选择哪些站，才能使汽车通过工厂的总时间最小

的一个底盘从装配线 i 移走所花的时间为 $t_{i,j}$，其中 $i=1$，2，而 $j=1$，2，…，$n-1$(因为在第 n 个装配站后，装配已经完成)。问题是要确定在装配线 1 内选择哪些站以及在装配线 2 内选择哪些站，以使汽车通过工厂的总时间最小。在图 15-2a 所示的例子中，最快的总时间是选择装配线 1 的装配站 1，3 和 6，以及装配线 2 的装配站 2，4 和 5。

显然，当有很多个装配站时，用强力法(brute force)来极小化通过工厂装配线的时间是不可行的。如果给定一个序列，在装配线 1 上使用哪些站，在装配线 2 上使用哪些站，则可以在$\Theta(n)$时间内，很容易计算出一个底盘通过工厂装配线要花的时间。不幸的是，选择装配站的可能方式有 2^n 种；可以把装配线 1 内使用的装配站集合看作{1，2，…，n}的一个子集，并注意到有 2^n 个这样的子集。因此，要通过穷举所有可能的方式、然后计算每种方式花费的时间来确定最快通过工厂的路线，需要 $\Omega(2^n)$时间，这在 n 很大时是不可行的。



a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

b)

图 15-2　a)一个装配线问题的实例，代价标示为 e_i、$a_{i,j}$、$t_{i,j}$ 以及 x_i。深阴影的路径表示通过工厂的最快方式。b) a)中的 $f_i[j]$，f^*，$l_i[j]$ 以及 l^* 实例的值

步骤 1：通过工厂最快路线的结构

动态规划方法的第一个步骤是描述最优解的结构的特征。对于装配线调度问题，可以如下执行。考虑底盘从起始点到装配站 $S_{1,j}$ 的最快可能路线。如果 $j=1$，则底盘能走的只有一条路线，所以很容易就可以确定它到装配站 $S_{1,j}$ 花费了多少时间。对于 $j=2, 3, \cdots, n$，则有两种选择：这个底盘可能从装配站 $S_{1,j-1}$ 直接到装配站 $S_{1,j}$，在相同的装配线上，从装配站 $j-1$ 到 j 的时间是可以忽略的。或者，这个底盘可能来自装配站 $S_{2,j-1}$，然后再移动到装配站 $S_{1,j}$，移动的代价是 $t_{2,j-1}$。我们将分别考虑这两种可能性，后面可以看到，它们之间其实是有很多共性的。

首先，假设通过装配站 $S_{1,j}$ 的最快路线通过了装配站 $S_{1,j-1}$。关键的一点是这个底盘必定是利用了最快的路线从开始点到装配站 $S_{1,j-1}$ 的。这是为什么呢？如果存在一条更快的路线通过 $S_{1,j-1}$，我们就可以采用这条更快的路线，从而得到通过装配站 $S_{1,j}$ 的更快的路线：这就形成了矛盾。

类似地，假设通过装配站 $S_{1,j}$ 的最快路线就是通过装配站 $S_{2,j-1}$。现在，我们注意到这个底盘必定是利用了最快的路线从开始点到装配站 $S_{2,j-1}$ 的。理由是相同的：如果有一条更快的通过装配站 $S_{2,j-1}$ 的路线，就可以采用这条更快的路线，从而得到通过装配站 $S_{1,j}$ 的更快的路线，这是一个矛盾。

324 ~ 326

更一般地，对于装配线调度问题，一个问题的（找出通过装配站 $S_{i,j}$ 的最快路线）最优解包含了子问题（找出通过 $S_{1,j-1}$ 或 $S_{2,j-1}$ 的最快路线）的一个最优解。我们称这个性质为**最优子结构**，这是是否可以应用动态规划方法的标志之一，具体会在 15.3 节中看到。

下面利用最优子结构来说明，可以利用子问题的最优解来构造原问题的一个最优解。对于装配线调度问题，推理如下。观察一条通过装配站 $S_{1,j}$ 的最快路线，会发现它必定是经过装配线 1 或 2 上的装配站 $j-1$。因此，通过装配站 $S_{1,j}$ 的最快路线只能是以下二者之一：

- 通过装配站 $S_{1,j-1}$ 的最快路线，然后直接通过装配站 $S_{1,j}$；
- 通过装配站 $S_{2,j-1}$ 的最快路线，从装配线 2 移动到装配线 1，然后通过装配站 $S_{1,j}$。

利用对称的推理思想，通过装配站 $S_{2,j}$ 的最快路线也只能是以下二者之一：

- 通过装配站 $S_{2,j-1}$ 的最快路线，然后直接通过装配站 $S_{2,j}$；
- 通过装配站 $S_{1,j-1}$ 的最快路线，从装配线 1 移动到装配线 2，然后通过装配站 $S_{2,j}$。

为了解决这个问题，即寻找通过任一条装配线上的装配站 j 的最快路线，我们解决它的子问题，即寻找通过两条装配线上的装配站 $j-1$ 的最快路线。

所以，对于装配线调度问题，通过建立子问题的最优解，就可以建立原问题某个实例的一个最优解了。

步骤 2：一个递归的解

在动态规划方法中，第二个步骤是利用子问题的最优解来递归定义一个最优解的值。对于装配线的调度问题，我们选择在两条装配线上通过装配站 j 的最快路线的问题来作为子问题，$j=1, 2, \cdots, n$。令 $f_i[j]$ 表示一个底盘从起点到装配站 $S_{i,j}$ 的最快可能时间。

我们的最终目标是确定底盘通过工厂的所有路线的最快时间，记为 f^*。底盘必须一路经由装配线 1 或 2 通过装配站 n，然后到达工厂的出口。由于这些路线的较快者就是通过整个工厂的最快路线，有：

327

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2) \tag{15.1}$$

要对 $f_1[1]$ 和 $f_2[1]$ 进行推理也是很容易的。不管在哪一条装配线上通过装配站 1，底盘都是直接到达该装配站的。于是，

$$f_1[1] = e_1 + a_{1,1} \tag{15.2}$$

$$f_2[1] = e_2 + a_{2,1} \tag{15.3}$$

现在来考虑如何计算 $f_i[j]$，其中 $j=2, 3, \cdots, n(i=1, 2)$。先来看一看 $f_1[j]$，前面说过，通过装配站 $S_{1,j}$ 的最快路线或者是通过装配站 $S_{1,j-1}$，然后直接通过装配站 $S_{1,j}$ 的最快路线，或者是通过装配站 $S_{2,j-1}$，从装配线 2 移动到装配线 1，然后通过装配站 $S_{1,j}$ 的最快路线。在第一种情况中，有 $f_1[j]=f_1[j-1]+a_{1,j}$，而在后一种情况中，$f_1[j]=f_2[j-1]+t_{2,j-1}+a_{1,j}$。所以：

$$f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) \tag{15.4}$$

其中 $j=2, 3, \cdots, n$。对称地，有：

$$f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) \tag{15.5}$$

其中 $j=2, 3, \cdots, n$。合并公式(15.2)～公式(15.5)，得到递归公式：

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{如果 } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{如果 } j \geqslant 2 \end{cases} \tag{15.6}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{如果 } j = 1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{如果 } j \geqslant 2 \end{cases} \tag{15.7}$$

图 15-2b 示出了图 15-2a 例子中的由等式(15.6)和等式(15.7)计算出的 $f_i[j]$值，以及 f^* 的值。

$f_i[j]$的值就是子问题最优解的值。为了有助于跟踪最优解的构造过程，我们定义 $l_i[j]$为装配线的编号(1 或 2)，其中的装配站 $j-1$ 被通过装配站 $S_{i,j}$ 的最快路线所使用。这里，$i=1, 2$ 且 $j=2, 3, \cdots, n$。(我们避免定义 $l_i[1]$，因为在任一条装配线上都没有一个装配站在站 1 前面)。此外，还定义 l^* 为这样的装配线，其内的装配站 n 被通过整个工厂的最快路线所使用。$l_i[j]$的值可以帮助找到一个最快的路线。利用图 15-2b 中所示的 l^* 的值和 $l_i[j]$，可以如下找到如图 15-2a 所示通过工厂的一条最快路线。从 $l^*=1$ 开始，使用装配站 $S_{1,6}$。现在看到 $l_1[6]$值为 2，所以使用装配站 $S_{2,5}$。接着，可以看到$l_2[5]=2$(使用装配站 $S_{2,4}$)，$l_2[4]=1$(装配站 $S_{1,3}$)，$l_1[3]=2$(装配站 $S_{2,2}$)，以及 $l_2[2]=1$(装配站 $S_{1,1}$)。

步骤 3：计算最快时间

此时，写出一个递归算法来计算通过工厂的最快路线是一件简单的事情，它基于公式(15.1)以及递归式(15.6)和式(15.7)。这种递归算法有一个问题：它的执行时间是关于 n 的指数形式。 328
要知道为什么，令 $r_i(j)$为递归算法中引用 $f_i[j]$的次数。由公式(15.1)，有

$$r_1(n) = r_2(n) = 1 \tag{15.8}$$

由递归式(15.6)和式(15.7)得到

$$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1) \tag{15.9}$$

其中 $j=1, 2, \cdots, n-1$。练习 15.1-2 会要求读者证明 $r_i[j]=2^{n-j}$。这样，单是 $f_1[1]$就被引用了 2^{n-1}次！如练习 15.1-3 要求读者证明的那样，引用所有 $f_i[j]$值的总次数为 $\Theta(2^n)$。

如果在递归的方式中以不同的顺序来计算 $f_i[j]$的值，能做得更好。注意对于 $j \geqslant 2$，$f_i[j]$的每一个值仅依赖于 $f_1[j-1]$和 $f_2[j-1]$的值。通过以递增装配站编号 j 的顺序来计算 $f_i[j]$的值，即在图 15-2b 中从左到右，可以在 $\Theta(n)$时间内计算出通过工厂的最快路线，以及其所花的时间。FASTEST-WAY 程序以值 $a_{i,j}$，$t_{i,j}$，e_i 和 x_i，以及在每条装配线中装配站的数目 n 作为输入。

```
FASTEST-WAY(a, t, e, x, n)
```

```
1  f1[1] ← e1 + a1,1
2  f2[1] ← e2 + a2,1
3  for j ← 2 to n
4      do if f1[j−1] + a1,j ≤ f2[j−1] + t2,j−1 + a1,j
5           then f1[j] ← f1[j−1] + a1,j
6                l1[j] ← 1
7           else f1[j] ← f2[j−1] + t2,j−1 + a1,j
8                l1[j] ← 2
9         if f2[j−1] + a2,j ≤ f1[j−1] + t1,j−1 + a2,j
10          then f2[j] ← f2[j−1] + a2,j
11               l2[j] ← 2
12          else f2[j] ← f1[j−1] + t1,j−1 + a2,j
13               l2[j] ← 1
14 if f1[n] + x1 ≤ f2[n] + x2
15    then f* ← f1[n] + x1
16         l* ← 1
17    else f* ← f2[n] + x2
18         l* ← 2
```

FASTEST-WAY 的工作方式如下。第 1～2 行利用公式(15.2)和公式(15.3)来计算 $f_1[1]$和 $f_2[1]$。然后第 3～13 行的 for 循环计算 $f_i[j]$和 $l_i[j]$，$i=1$，2，且 $j=2$，3，…，n。第 4～8 行利用公式(15.4)来计算 $f_1[j]$和 $l_1[j]$，而第 9～13 行利用公式(15.5)来计算 $f_2[j]$和 $l_2[j]$。最后，第 14～18 行利用公式(15.1)来计算 f^* 和 l^*。因为第 1～2 行与第 14～18 行花费常数时间，而且第 3～13 行的 for 循环的 $n-1$ 次迭代中的每一个也花费常数时间，所以整个过程花费 $\Theta(n)$ 时间。

329

观察 $f_i[j]$和 $l_i[j]$值的计算过程的一种方式是在表格中填入记录。在图 15.2b 中，我们在表格内从左到右填入 $f_i[j]$和 $l_i[j]$的数值(在每一列中从上到下)。要填入一个记录 $f_i[j]$，需要 $f_1[j-1]$和 $f_2[j-1]$的值，由于已经计算并保存了它们，只需简单地查表来确定它们的值。

步骤 4：构造通过工厂的最快路线

计算出 $f_i[j]$，f^*，$l_i[j]$，l^* 的值之后，需要构造在通过工厂的最快路线中使用的装配站的序列。我们在上面已经讨论了如何在图 15-2 的例子中做到这一点。

下面的过程以站号的递减顺序，输出所使用的各个装配站。练习 15.1-1 要求读者修改这个过程，使它按站号的递增顺序输出各个装配站。

```
PRINT-STATIONS(l, l* , n)
1  i ← l*
2  print "line" i ", station" n
3  for j ← n downto 2
4      do i ← li[j]
5           print "line" i ", station" j−1
```

在图 15-2 的例子中，PRINT-STATIONS 将产生以下的输出：

```
line 1, station 6
line 2, station 5
line 2, station 4
line 1, station 3
```

line 2，station 2

line 1，station 1

练习

15.1-1　说明应如何修改程序 PRINT-STATIONS，让它以站号的递增顺序输出各装配站。（提示：利用递归。）

330

15.1-2　利用公式(15.8)、(15.9)及替换法来证明：在递归算法中引用 $f_i[j]$的次数 $r_i(j)$等于 2^{n-j}。

15.1-3　利用练习 15.1-2 的结果，证明所有引用 $f_i[j]$的总次数(即 $\sum_{i=1}^{2}\sum_{j=1}^{n} r_i(j)$)等于 $2^{n+1}-2$。

15.1-4　包含 $f_i[j]$和 $l_i[j]$值的表格共含有 $4n-2$ 个表项。说明如何把空间需求缩减到共 $2n+2$ 个表项，仍然能够计算出 f^*，并且仍然能够输出通过工厂的最快路线上的所有装配站。

15.1-5　Canty 教授猜测存在着某些 e_i，$a_{i,j}$以及 $t_{i,j}$的值，使得 FASTEST-WAY 程序在某个装配站 j 上，产生出满足 $l_1[j]=2$ 且 $l_2[j]=1$ 的 $l_i[j]$值。假设所有的移动代价 $t_{i,j}$是非负值，说明 Canty 教授的猜测是不正确的。

15.2　矩阵链乘法

我们用来说明动态规划的下一个例子是解决矩阵链相乘问题的一个算法。给定由 n 个要相乘的矩阵构成的序列(链)$\langle A_1, A_2, \cdots, A_n\rangle$，要计算乘积

$$A_1 A_2 \cdots A_n \tag{15.10}$$

为计算式(15.10)，可将两个矩阵相乘的标准算法作为一个子程序，根据括号给出的计算顺序做全部的矩阵乘法。一组矩阵的乘积是*加全部括号的*(fully parenthesized)，如果它是单个的矩阵，或是两个加全部括号的矩阵的乘积外加括号而成。矩阵的乘法满足结合率，故无论怎样加括号都会产生相同的结果。例如，如果矩阵链为$\langle A_1, A_2, A_3, A_4\rangle$，乘积 $A_1A_2A_3A_4$ 可用五种不同方式加全部括号：

$(A_1(A_2(A_3A_4)))$，

$(A_1((A_2A_3)A_4))$，

$((A_1A_2)(A_3A_4))$，

$((A_1(A_2A_3))A_4)$，

$(((A_1A_2)A_3)A_4)$。

331

矩阵链加括号的顺序对求积运算的代价有很大的影响。先来看看两个矩阵相乘的代价。标准的算法由下面的伪代码给出。属性 *rows* 和 *columns* 表示矩阵的行数和列数。

```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2     then error "incompatible dimensions"
3     else for i ← 1 to rows[A]
4             do for j ← 1 to columns[B]
5                   do C[i, j] ← 0
6                      for k ← 1 to columns[A]
7                          do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8          return C
```

仅当两个矩阵 A 和 B 相容(即 A 的列数等于 B 的行数)时，才可以进行相乘运算。如果 A 是

个 $p\times q$ 矩阵，B 是个 $q\times r$ 矩阵，则结果矩阵 C 是一个 $p\times r$ 矩阵。计算 C 的时间由第 7 行中标量乘法运算的次数决定，这个次数等于 pqr。接下来对运行时间的分析都将以标量乘法次数来表示。

为说明由不同的加全部括号顺序所带来的矩阵乘积的代价不同，考虑三个矩阵的链$\langle A_1, A_2, A_3\rangle$的问题。假设三个矩阵的维数分别为 10×100，100×5，5×50。如果按$((A_1A_2)A_3)$规定的次序来做乘法，求 10×5 的矩阵乘积 A_1A_2 要做 $10\times 100\times 5=5000$ 次的标量乘法运算，再乘上 A_3 还要做$10\times 5\times 50=2500$次标量乘法，总共 7500 次标量乘法运算。如果按$(A_1(A_2A_3))$的次序来计算，则为求 100×50 的矩阵乘积 A_2A_3 要做 $100\times 5\times 50=25\,000$ 次标量乘法运算，再乘上 A_1 还要 $10\times 100\times 50=50\,000$ 次标量乘法，总共 75 000 次标量乘法运算。因此，按第一种运算次序进行计算就要快到 10 倍。

矩阵链乘法问题可表述如下：给定 n 个矩阵构成的一个链$\langle A_1, A_2, \cdots, A_n\rangle$，其中 $i=1, 2, \cdots, n$，矩阵 A_i 的维数为 $p_{i-1}\times p_i$，对乘积 $A_1A_2\cdots A_n$ 以一种最小化标量乘法次数的方式进行加全部括号。

注意在矩阵链乘法问题中，实际上并没有把矩阵相乘。目的仅是确定一个具有最小代价的矩阵相乘顺序。通常，与真正在执行矩阵乘法时所节省的时间相比，在确定最优顺序上花费的时
332 间会得到更多的回报(例如只做 7500 次标量乘法而不是 75 000 次)。

计算全部括号的重数

在利用动态规划来解矩阵链乘法问题之前，应认识到靠穷尽所有可能的加全部括号方案不会得到一个有效的算法。设 $P(n)$表示一串 n 个矩阵可能的加全部括号方案数。当 $n=1$ 时，只有一个矩阵，因此只有一种方式来加全部括号矩阵乘积。当 $n\geqslant 2$ 时，一个加全部括号的矩阵乘积，就是两个加全部括号的矩阵子乘积的乘积，而且这两个子乘积之间的分裂可能发生在第 k 个和第 $(k+1)$个矩阵之间，其中 $k=1, 2, \cdots, n-1$。因此，可得递归式

$$P(n)=\begin{cases}1 & \text{如果 } n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & \text{如果 } n\geqslant 2\end{cases} \tag{15.11}$$

思考题 12-4 曾要求读者证明一个类似递归式的解是 *Catalan* 数序列，其增长的形式为 $\Omega(4^n/n^{3/2})$。一个较简单的练习(参见练习 15.2-3)将证明递归式(15.11)的解为 $\Omega(2^n)$。所以解的个数是 n 的指数形式，而且对确定矩阵链的最优加全部括号来说，穷尽搜索不是一个好的策略。

步骤 1：最优加全部括号的结构

动态规划方法的第一步是寻找最优的子结构，然后，利用这一子结构，就可以根据子问题的最优解构造出原问题的一个最优解。对于矩阵链乘法问题，可以如下执行这个步骤。为方便起见，用记号 $A_{i..j}$表示对乘积 $A_iA_{i+1}\cdots A_j$ 求值的结果，其中 $i\leqslant j$。注意如果这个问题是非平凡的，即 $i<j$，则对乘积 $A_iA_{i+1}\cdots A_j$ 的任何加全部括号形式都将乘积在 A_k 与 A_{k+1}之间分开，此处 k 是范围 $1\leqslant k<j$ 之内的一个整数。这就是说，对某个 k 值，首先计算矩阵 $A_{i..k}$和 $A_{k+1..j}$，然后把它们相乘就得到最终乘积 $A_{i..j}$。这样，加全部括号的代价就是计算 $A_{i..k}$和 $A_{k+1..j}$的代价之和，再加上两者相乘的代价。

这个问题的最优子结构如下。假设 $A_iA_{i+1}\cdots A_j$ 的一个最优加全部括号把乘积在 A_k 与 A_{k+1} 之间分开，则对 $A_iA_{i+1}\cdots A_j$ 最优加全部括号的“前缀”子链 $A_iA_{i+1}\cdots A_k$ 的加全部括号必须是
333 $A_iA_{i+1}\cdots A_k$ 的一个最优加全部括号。为什么呢？如果对 $A_iA_{i+1}\cdots A_k$ 有一个代价更小的加全部括号，那么把它替换到 $A_iA_{i+1}\cdots A_j$ 的最优加全部括号中去就会产生 $A_iA_{i+1}\cdots A_j$ 的另一种加全

部括号，而它的代价小于最优代价：产生了矛盾。类似的观察也成立，即 $A_iA_{i+1}\cdots A_j$ 的最优加全部括号的子链 $A_{k+1}A_{k+2}\cdots A_j$ 的加全部括号：必须是 $A_{k+1}A_{k+2}\cdots A_j$ 的一个最优加全部括号。

现在，就可以利用所得到的最优子结构，来说明能够根据子问题的最优解来构造原问题的一个最优解。已经看到，一个矩阵链乘法问题的非平凡实例的任何解法都需要分割乘积，而且任何最优解都包含了子问题实例的最优解。所以，可以把问题分割为两个子问题(最优加全部括号 $A_iA_{i+1}\cdots A_k$ 和 $A_{k+1}A_{k+2}\cdots A_j$)，寻找子问题实例的最优解，然后合并这些子问题的最优解，来构造一个矩阵链乘法问题实例的一个最优解。必须保证在寻找一个正确的位置来分割乘积时，我们已经考虑过所有可能的位置，从而确保已检查过了最优的一个。

步骤 2：一个递归解

接下来，根据子问题的最优解来递归定义一个最优解的代价。对于矩阵链乘法问题，子问题即确定 $A_iA_{i+1}\cdots A_j$ 的加全部括号的最小代价问题，此处 $1\leqslant i\leqslant j\leqslant n$。设 $m[i, j]$为计算矩阵 $A_{i..j}$所需的标量乘法运算次数的最小值；对整个问题，计算 $A_{1..n}$的最小代价就是 $m[1, n]$。

我们这样来递归定义 $m[i, j]$。如果 $i=j$，则问题是平凡的；矩阵链只包含一个矩阵 $A_{i..i}=A_i$，故无需做任何标量乘法来计算乘积。因此 $m[i, i]=0$，对 $i=1, 2, \cdots, n$。当 $i<j$ 时，为计算 $m[i, j]$，可以利用步骤 1 中得出的最优解的结构。假设最优加全部括号将乘积 $A_iA_{i+1}\cdots A_j$ 从 A_k 和 A_{k+1}之间分开，其中 $i\leqslant k<j$。因此 $m[i, j]$就等于计算子乘积 $A_{i..k}$和 $A_{k+1..j}$的代价，再加上这两个矩阵相乘的代价的最小值。回忆起每个矩阵 A_i 是 $p_{i-1}\times p_i$ 的，可以看出，计算 $A_{i..k}A_{k+1..j}$要做 $p_{i-1}p_kp_j$ 次标量乘法。所以得到：

$$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$$

这个递归公式假设我们已知 k 的值，而实际上我们并不知道。但是，k 只有 $j-i$ 个可能的值，即$k=i, i+1, \cdots, j-1$。最优加全部括号必然要用到其中之一的 k 值，故只需逐个检查这些值以找到最佳值。这样，关于对乘积 $A_iA_{i+1}\cdots A_j$ 的加全部括号的最小代价的递归定义为 334

$$m[i,j]=\begin{cases}0 & \text{如果 } i=j\\ \min\limits_{i\leqslant k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{如果 } i<j\end{cases} \tag{15.12}$$

各 $m[i, j]$的值给出了子问题的最优解的代价。为了有助于跟踪如何构造一个最优解，定义 $s[i, j]$为这样的一个 k 值：在该处分裂乘积 $A_iA_{i+1}\cdots A_j$ 后可得一个最优加全部括号。亦即 $s[i, j]$等于使 $m[i, j]=m[i, k]+m[k+1, j]+p_{i-1}p_kp_j$ 的 k 值。

步骤 3：计算最优代价

现在，可以很容易地根据递归式(15.12)，来写一个计算乘积 $A_1A_2\cdots A_n$ 的最小代价 $m[1, n]$的递归算法。然而，在 15.3 节中可以看到，这个算法具有指数时间，它与检查每一种加全部括号乘积的强力法差不多。

可以注意到原问题只有相当少的子问题：每一对满足 $1\leqslant i\leqslant j\leqslant n$ 的 i 和 j 对应一个问题，总共$\binom{n}{2}+n=\Theta(n^2)$。一个递归算法在其递归树的不同分支中可能会多次遇到同一个子问题。子问题重叠这一性质是是否可以采用动态规划的第二个标志(第一个标志是最优子结构)。

我们不是递归地解递归式(15.12)，而是执行动态规划方法的第三个步骤，使用自底向上的表格法来计算最优代价。下面的伪代码假设矩阵 A_i 的维数是 $p_{i-1}\times p_i$，$i=1, 2, \cdots, n$。输入是一个序列 $p=\langle p_0, p_1, \cdots, p_n\rangle$，其中 $length[p]=n+1$。此程序使用一个辅助表 $m[1..n, 1..n]$来保存 $m[i, j]$的代价，使用一个辅助表 $s[1..n, 1..n]$来记录计算 $m[i, j]$时取得最优代价处 k 的值。我们将利用表格 s 来构造一个最优解。

为了正确实现自底向上的方法，必须确定哪些表项要被用来计算 $m[i, j]$。公式(15.12)说明计算一个有 $j-i+1$ 个矩阵的矩阵链乘积时，其代价 $m[i, j]$仅依赖于计算一个有少于 $j-i+1$ 个矩阵的矩阵链乘积的代价。也就是说，对 $k=i, i+1, \cdots, j-1$，矩阵 $A_{i..k}$是 $k-i+1<j-i+1$ 个矩阵的乘积，矩阵 $A_{k+1..j}$是 $j-k<j-i+1$ 个矩阵的乘积。因此，该算法填表 m 的方式
335 对应于求解按长度递增的矩阵链上的加全部括号问题。

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p]−1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n              ▷ l is the chain length
 5       do for i ← 1 to n−l+1
 6            do j ← i+l−1
 7               m[i, j] ← ∞
 8               for k ← i to j−1
 9                  do q ← m[i, k]+m[k+1, j]+p_{i−1} p_k p_j
10                     if q<m[i, j]
11                       then m[i, j] ← q
12                            s[i, j] ← k
13  return m and s
```

该算法首先在第 2～3 行中对 $i=1, 2, \cdots, n$ 置 $m[i, i]$←0(长度为 1 的链的最小代价)。在第 4～12 行循环的第一次执行中，利用递归式(15.12)来计算 $m[i, i+1]$，$i=1, 2, \cdots, n-1$(长度 l 等于 2 的链的最小代价)。在循环的第二次执行中，计算 $m[i, i+2]$，$i=1, 2, \cdots, n-2$(长度 l 等于 3 的链的最小代价)，这样一直继续下去。在每一步中，第 9～12 行中计算出的 $m[i, j]$仅依赖于已经计算出的表项 $m[i, k]$和 $m[k+1, j]$。

图 15-3 演示了对包含 $n=6$ 个矩阵的链该算法的执行过程。因为我们仅对 $i \leqslant j$ 定义了 $m[i, j]$，故表 m 中仅主对角线以上的部分被用到。该图已将表进行了旋转，使主对角线水平放置。矩阵链沿表的底部排列。用这样一种安排，在经过 A_i 的东北方向的直线与经过 A_j 的西北方向的直线的交点上，可以找到子矩阵链 $A_iA_{i+1}\cdots A_j$ 相乘的最小代价 $m[i, j]$。表中的每一水平行包含相同长度的矩阵链的表项。过程 MATRIX-CHAIN-ORDER 自底向上地计算每一行，在每一行中又从左到右进行计算。表项 $m[i, j]$是根据乘积 $p_{i-1}p_kp_j(k=i, i+1, \cdots, j-1)$与所有在 $m[i, j]$西南向和东南向的表项计算出来的。

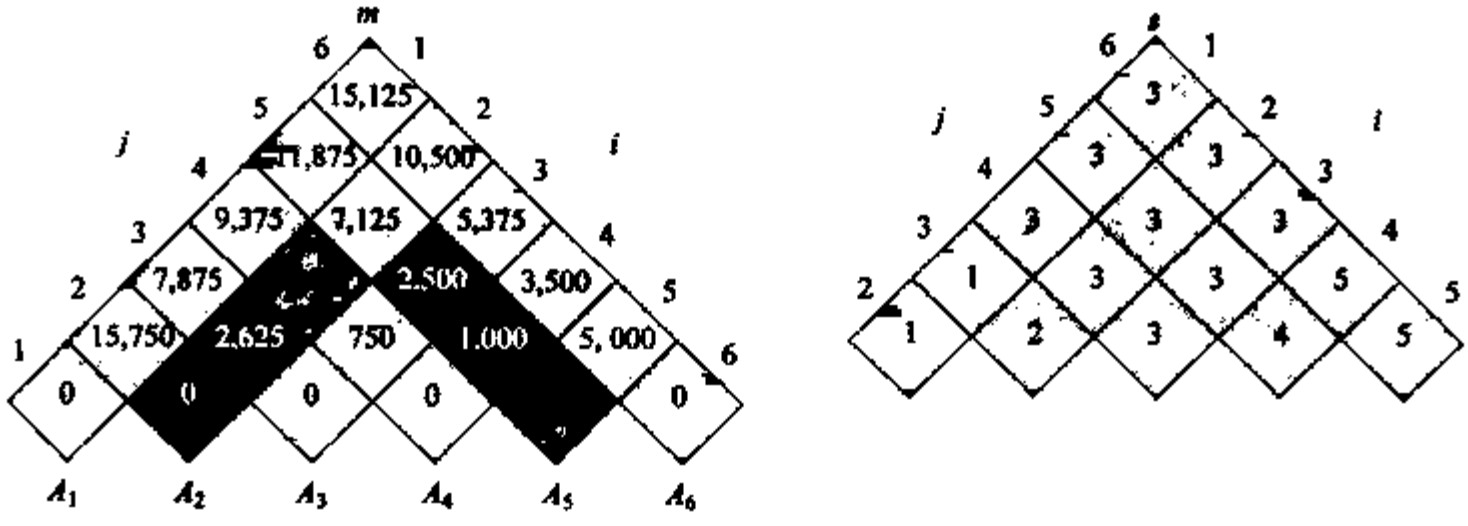


图 15-3　对 $n=6$ 及下面各矩阵维数由 MATRIX-CHAIN-ORDER 计算出的表 m 和 s

矩阵	维数
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

MATRIX-CHAIN ORDER 具有嵌套循环结构，运行时间为 $O(n^3)$。循环的嵌套层数为三层，每一层循环的下标(l，i 和 k)可至多取 $n-1$ 个值。练习 15.2-4 要求读者证明这个算法的运行时间实际上是 $\Omega(n^3)$。另外，该算法还需要 $\Theta(n^2)$的空间来保存 m 和 s。这样，与穷尽各种可能的加全部括号形式的指数时间方法相比，MATRIX-CHAIN-ORDER 就要有效得多了。

对图中的两张表进行旋转，使其主对角线处于水平方向。在表 m 中仅用到了主对角线与上三角，而表 s 中仅用到了上三角。六个矩阵相乘所需的标量乘法的最少次数为 $m[1, 6]=15\,125$。在加了阴影的表项中，在第 9 行中计算下式时，具有相同阴影程度的对被放在一起。

$$m[2,5]=\min\begin{cases} m[2,2]+m[3,5]+p_1p_2p_5 = 0+2500+35\cdot 15\cdot 20 = 13\,000 \\ m[2,3]+m[4,5]+p_1p_3p_5 = 2625+1000+35\cdot 5\cdot 20 = 7125 \\ m[2,4]+m[5,5]+p_1p_4p_5 = 4375+0+35\cdot 10\cdot 20 = 11\,375 \end{cases}$$
$$= 7125$$

步骤 4：构造一个最优解

虽然 MATRIX-CHAIN-ORDER 确定了计算矩阵链乘积所需的最优标量乘法次数，但它没有直接说明如何对这些矩阵进行相乘。利用保存在表格 $s[1..n, 1..n]$内的、经过计算的信息来构造一个最优解并不难。在每一个表项 $s[i, j]$中，记录了对乘积 $A_iA_{i+1}\cdots A_j$ 在 A_k 与 A_{k+1}之间，进行分裂以取得最优加全部括号时的 k 值。由此可知，按最优方式计算 $A_{1..n}$时，最终矩阵相乘次序是 $A_{1..s[1,n]}A_{s[1,n]+1..n}$。之前的矩阵乘法可以递归地进行，因为 $s[1, s[1, n]]$决定计算 $A_{1..s[1,n]}$中最后的矩阵乘法，而 $s[s[1, n]+1, n]$则决定计算 $A_{s[1,n]+1..n}$中最后的矩阵乘法。下面的递归过程在给定由 MATRIX-CHAIN-ORDER 计算出的表 s 以及下标 i 和 j 后，输出(A_i，A_{i+1}，…，A_j)的一个最优加全部括号形式。初始调用 PRINT-OPTIMAL-PARENS(s，1，n)输出(A_1，A_2，…，A_n)的一个最优加全部括号形式。

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i=j
2    then print "A"_i
3    else print "("
4         PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5         PRINT-OPTIMAL-PARENS(s, s[i, j]+1, j)
6         print ")"
```

在图 15-3 的例子中，调用 PRINT-OPTIMAL-PARENS(s，1，6)输出加全部括号$((A_1(A_2A_3))((A_4A_5)A_6))$。

练习

15.2-1　对维数为序列(5，10，3，12，5，50，6)的各矩阵，找出其矩阵链乘积的一个最优加全部括号。

15.2-2　请给出一个递归算法 MATRIX-CHAIN-MULTIPLY(A，s，i，j)，使之在给出矩阵序列(A_1，A_2，…，A_n)、和由 MATRIX-CHAIN-ORDER 计算出的表 s，以及下标 i 和 j 后，能得出一个最优的矩阵链乘法。(初始调用为 MATRIX-CHAIN-MULTIPLY(A，s，1，n))。

336 ~ 337

15.2-3　使用替换法证明递归公式(15.11)的解为 $\Omega(2^n)$。

15.2-4　设 $R(i, j)$表示在调用 MATRIX-CHAIN-ORDER 中计算其他表项时，表项 $m[i, j]$被引用的次数。证明：对整个表总的引用次数为

$$\sum_{i=1}^{n}\sum_{j=i}^{n}R(i,j)=\frac{n^3-n}{3}$$

338　(提示：可以利用等式(A.3)。)

15.2-5　证明：一个含 n 个元素的表达式的加全部括号中恰有 $n-1$ 对括号。

15.3　动态规划基础

虽然我们刚刚讨论过动态规划方法的两个例子，但读者对什么时候可以应用这种方法可能仍然不一定很清楚。从工程的角度来看，什么时候才需要寻找一个问题的动态规划解？在本节中，我们要介绍适合采用动态规划方法的最优化问题中的两个要素：最优子结构和重叠子问题。另外，还要分析一种不同的方法，称为做备忘录(memoization)[⊖]，以充分利用重叠子问题性质。

最优子结构

用动态规划求解优化问题的第一步是描述最优解的结构。回顾一下，如果问题的一个最优解中包含了子问题的最优解，则该问题具有最优子结构。当一个问题具有最优子结构时，提示我们动态规划可能会适用。(注意，在这种情况下，贪心策略可能也是适用的，参见第 16 章)。在动态规划中，我们利用子问题的最优解来构造问题的一个最优解。因此，必须小心以确保在我们所考虑的子问题范围中，包含了用于一个最优解中的那些子问题。

到目前为止在本章讨论的两个子问题中，都发现了最优子结构。在 15.1 节中，可以看到在任何一条装配线上，通过装配站 j 的最快路线包含了在其中一条装配线上通过装配站 $j-1$ 的最快路线。在 15.2 节，我们看到 $A_iA_{i+1}\cdots A_j$ 的一个最优加全部括号把乘积在 A_k 和 A_{k+1} 之间分裂，它包含了加全部括号 $A_iA_{i+1}\cdots A_k$ 和 $A_{k+1}A_{k+2}\cdots A_j$ 问题的最优解。

339　在找寻最优子结构时，可以遵循一种共同的模式：

1)问题的一个解可以是做一个选择。例如，选择一个前一个装配线装配站；或者选择一个下标以在该位置分裂矩阵链。做这种选择会得到一个或多个有待解决的子问题。

2)假设对一个给定的问题，已知的是一个可以导致最优解的选择。不必关心如何确定这个选择，尽管假定它是已知的。

3)在已知这个选择后，要确定哪些子问题会随之发生，以及如何最好地描述所得到的子问题空间。

4)利用一种"剪贴"(cut-and-paste)技术，来证明在问题的一个最优解中，使用的子问题的解本身也必须是最优的。通过假设每一个子问题的解都不是最优解，然后导出矛盾，即可做到这一点。特别地，通过"剪除"非最优的子问题解再"贴上"最优解，就证明了可以得到原问题的一个更好的解，因此，这与假设已经得到一个最优解相矛盾。如果有多于一个的子问题的话，由于它们通常非常类似，所以只要对其中一个子问题的"剪贴"处理略加修改，即可很容易地用于其他子问题。

为了描述子问题空间，可以遵循这样一条有效的经验规则，就是尽量保持这个空间简单，然后在需要时再扩充它。例如，在装配线调度问题中，我们所考虑的子问题空间就是从工厂入口通过装配站 $S_{1,j}$ 和 $S_{2,j}$ 的最快路线。这个子问题空间很合适，因而没有必要再去尝试一个更具一般

⊖　这并不是拼写错误。这个单词确实是 memoization，而不是 memorization。memoization 来源于 memo，因为这个方法主要是记录一个值，以便以后可以检索它。

性的子问题空间了。

相反地，在矩阵链乘积问题中，假设我们已经试图将子问题空间约束为形如 $A_1A_2\cdots A_j$ 的矩阵乘积。如前所述，一个最优的加全部括号必定把乘积在 A_k 和 A_{k+1} 之间分开，对某个 $1\leqslant k\leqslant j$。除非我们能保证 k 总是等于 $j-1$，不然会发现有形如 $A_1A_2\cdots A_k$ 和 $A_{k+1}A_{k+2}\cdots A_j$ 的子问题，而且后者不是 $A_1A_2\cdots A_j$ 的形式。对这个问题，有必要允许子问题在"两端"变化，亦即允许 i 和 j 在子问题$A_iA_{i+1}\cdots A_j$ 中可以变化。

最优子结构在问题域中以两种方式变化：

1)有多少个子问题被使用在原问题的一个最优解中，以及

2)在决定一个最优解中使用哪些子问题时有多少个选择。

在装配线调度问题中，一个最优解只使用了一个子问题，但是，为确定一个最优解，我们必须考虑两种选择。为找出通过装配站 $S_{i,j}$ 的最快路线，我们使用通过 $S_{1,j-1}$ 或 $S_{2,j-1}$ 的最快路线；不管采用了哪一种，它都代表了必须最优解决的子问题。子链 $A_iA_{i+1}\cdots A_j$ 的矩阵链乘法可作为有两个子问题和 $j-i$ 个选择的一个例子。对一个在该处分离乘积的给定的矩阵 A_k，可以分解出两个子问题，即加全部括号 $A_iA_{i+1}\cdots A_k$ 和加全部括号 $A_{k+1}A_{k+2}\cdots A_j$，我们必须最优求解这两个子问题。一旦确定了子问题的最优解后，就从 $j-i$ 个候选者中选出那个下标 k。 340

非正式地，一个动态规划算法的运行时间依赖于两个因素的乘积：子问题的总个数和每一个子问题中有多少种选择。在装配线调度中，总共有 $\Theta(n)$ 个子问题，并且只有两个选择来检查每个子问题，所以执行时间为 $\Theta(n)$。对于矩阵链乘法，总共有 $\Theta(n^2)$ 个子问题，在每个子问题中又至多有 $n-1$ 个选择，因此执行时间为 $O(n^3)$。

动态规划以自底向上的方式来利用最优子结构。也就是说，首先找到子问题的最优解，解决子问题，然后找到问题的一个最优解。寻找问题的一个最优解需要在子问题中做出选择，即选择将用哪一个来求解问题。问题解的代价通常是子问题的代价加上选择本身带来的开销。例如，在装配线调度问题中，首先要解决寻找通过装配站 $S_{1,j-1}$ 或 $S_{2,j-1}$ 的最快路线这一子问题，然后，选择其中一个装配站作为装配站 $S_{i,j}$ 的前一站。选择本身所带来的开销依赖于是否在装配站 $j-1$ 与 j 之间转换装配线；如果停留在原装配线上，则代价为 $a_{i,j}$；如果转换了装配线，则为 $t_{i',j-1}+a_{i,j}$，其中 $i'\neq i$。在矩阵链乘法问题中，首先要确定子链 $A_iA_{i+1}\cdots A_j$ 的最优加全部括号，然后选择矩阵 A_k 在该位置分裂乘积。选择本身所带来的开销为 $p_{i-1}p_kp_j$ 项。

第 16 章中将研究"贪心算法"，它与动态规划有着很多相似之处。特别地，贪心算法适用的问题也具有最优子结构。贪心算法与动态规划有一个最著的区别，就是在贪心算法中，是以自顶向下的方式使用最优子结构的。贪心算法会先做选择，在当时看起来是最优的选择，然后再求解一个结果子问题，而不是先寻找子问题的最优解，然后再做选择。

一些细微之处

要注意在不能应用最优子结构的时候，就一定不能假设它能够应用。考虑下面两个问题，已知一个有向图 $G=(V, E)$和结点 $u, v\in V$。 341

无权最短路径[⊖]：找出一条从 u 到 v 的包含最少边数的路径。这样的一条路径必须是简单路径，因为从路径中去掉一个回路后，会产生边数更少的路径。

无权最长简单路径：找出一条从 u 到 v 的包含最多边数的简单路径。我们需要加入简单性的要求，因为否则的话，就可以随意地遍历一个回路任意多次，来得到有任意多的边数的路径。

⊖ 我们使用名词"无权"(unweighted)与寻找带权边的最短路径问题相区别；第 24、25 章中将讨论这个问题。不带权值的问题可以利用第 22 章中的广度优先搜索技术来求解。

无权最短路径问题具有如下的最优子结构。假设 $u \neq v$，因此这个问题是非平凡的。这样任何从 u 到 v 的路径 p 必定包含一个中间顶点，譬如 w。（注意 w 可以是 u 或是 v）。于是，可以将路径 $u \overset{p}{\rightsquigarrow} v$ 分解为子路径 $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$。显然，$p$ 上边的数目等于 p_1 上边的数目加上 p_2 上边的数目。我们断言，如果 p 是从 u 到 v 的最优(亦即最短)路径，那么 p_1 必定是从 u 到 w 的一条最短路径。为什么呢？我们可以使用“剪贴”方法来进行论证：如果有另一条路径，例如 p_1'，从 u 到 w 有比 p_1 更少的边，那么就可以剪下 p_1 然后贴上 p_1'，以构造比 p 有更少边的一条路径 $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$，而这与 p 是最优的相矛盾。对称地，p_2 必定是从 w 到 v 的一条最短路径。所以，通过考虑所有的中间顶点 w，找出从 u 到 w 的一条最短路径和从 w 到 v 的一条最短路径，然后选择一个会产生整体最短路径的中间顶点 w，来找出从 u 到 v 的一条最短路径。在 25.2 节中，我们将使用这个最优子结构的一个变形，来找出在带权有向图中每一对顶点之间的最短路径。

对无权最长简单路径的问题，假设它具有最优子结构。毕竟，如果将一条最长简单路径 $u \overset{p}{\rightsquigarrow} v$ 分解成子路径 $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$，则 p_1 一定不是从 u 到 w 的最长简单路径，p_2 一定不是从 w 到 v 的最长简单路径吗？答案是否定的！图 15-4 给出了一个例子。考虑路径 $q \to r \to t$，这是从 q 到 t 的一条最长简单路径。$q \to r$ 是从 q 到 r 的一条最长简单路径吗？不是，因为 $q \to s \to t \to r$ 是一条更长的简单路径。$r \to t$ 是一条从 r 到 t 的最长简单路径吗？也不是，因为 $r \to q \to s \to t$ 是一条更长的简单路径。

这个例子说明对于最长简单路径，不仅缺乏最优子结构，而且无法根据子问题的解来构造
342 问题的一个“合法”解。如果把最长简单路径 $q \to s \to t \to r$ 和 $r \to q \to s \to t$ 合并，得到路径 $q \to s \to t \to r \to q \to s \to t$，这不是简单路径。确实，在寻找一条无权最长简单路径的问题中，没有显示其具有任何类型的最优子结构。这个问题还没有找到高效率的动态规划算法。事实上，这个问题是 NP 完全的(这一点将在第 34 章中看到)，即意味着它不可能在多项式时间内解决。

为什么最长简单路径的子结构与最短路径的子结构有如此不同呢？虽然在最长和最短路径问题的解中都使用两个子问题，但是在寻找最长简单路径中子问题不是独立的，而在最短路径问题中它们是独立的。子问题独立是什么意思？意思是一个子问题的解不会影响同一问题中另外一个子问题的解。对于图 15-4 中的例子，找出从 q 到 r 的最长简单路径的问题有两个问题：找出从 q 到 r 以及从 r 到 t 的最长简单路径。对第一个子问题，我们选择路径 $q \to s \to t \to r$，因此使用了顶点 s 和 t。在第二个子问题中就不能再使用这些顶点了，因为合并两个子问题的解会得到一个非简单的路径。如果在第二个子问题中不能使用顶点 t，那我们就根本无法求解这个子问题，因为 t 必须在我们要寻找的路径上，而且它不是“接合”两个子问题的解的顶点(此顶点为 r)。在一个子问题解中使用顶点 s 和 t，会避免它们在另一个子问题中被使用。但是，我们必须使用这两个顶点中的至少一个，才能求解另外一个子问题，并且，只有同时使用它们，才能求得最优解。因此，我们说这些子问题不是相互独立的。用另外一种方式看，在求解一个子问题时，我们对资源的使用(资源即顶点)使得它们无法被另一个子问题所使用。

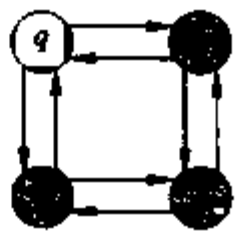


图 15-4　一个有向图，它说明了在一个无权有向图中寻找一条最长简单路径的问题没有最优子结构。路径 $q \to r \to t$ 是从 q 到 t 的一条最长简单路径，但是子路径 $q \to r$ 不是从 q 到 r 的一条最长简单路径，子路径 $r \to t$ 也不是从 r 到 t 的一条最长简单路径

那么为什么在寻找最短路径中子问题是独立的呢？回答是子问题本来就没有共享资源。我们断言，如果顶点 w 在从 u 到 v 的一条最短路径 p 上，那么我们可以接合 $u \overset{}{\rightsquigarrow} w$ 的任意最短路

径和 $w \overset{p_2}{\rightsquigarrow} v$ 的任意最短路径来产生一条从 u 到 v 的最短路径。我们确信，除了 w 之外，没有顶 343
点可以同时出现在路径 p_1 和 p_2 上。为什么呢？假设有某个顶点 $x \neq w$ 同时出现在路径 p_1 和 p_2 上，则可以把 p_1 分解为 $u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xw}}{\rightsquigarrow} w$，以及将 p_2 分解为 $w \overset{p_{wx}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$。根据这个问题的最优子结构，路径 p 的边数和 p_1 与 p_2 合起来的边数相等，假设 p 有 e 条边。现在我们构造一条从 u 到 v 的路径 $u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$。这条路径至多有 $e-2$ 条边，这与 p 是一条最短路径的假设相矛盾。所以，我们确信最短路径问题中的子问题是独立的。

在 15.1 节和 15.2 节中讨论的两个问题都有独立的子问题。在矩阵链乘法中，子问题是把子链 $A_iA_{i+1}\cdots A_k$ 和 $A_{k+1}A_{k+2}\cdots A_j$ 相乘。这些子链是不相交的，因此没有矩阵会同时被包含在它们两个之中。在装配线调度中，为确定通过装配站 $S_{i,j}$ 的最快路线，我们分析了通过装配站 $S_{1,j-1}$ 与 $S_{2,j-1}$ 的最快路线。因为通过装配站 $S_{i,j}$ 的最快路线的解只包含其中一个子问题的解，这个子问题自动与解中使用的所有其他的子问题相独立。

重叠子问题

适用于动态规划求解的最优化问题必须具有的第二个要素是子问题的空间要“很小”，也就是用来解原问题的递归算法可反复地解同样的子问题，而不是总在产生新的子问题。典型地，不同的子问题数是输入规模的一个多项式。当一个递归算法不断地调用同一问题时，我们说该最优问题包含重叠子问题[⊖]。相反地，适合用分治法解决的问题往往在递归的每一步都产生全新的问题。动态规划算法总是充分利用重叠子问题，即通过每个子问题只解一次，把解保存在一个在需要时就可以查看的表中，而每次查表的时间为常数。

在 15.1 节中，我们简要分析了装配线调度问题的一个递归解是如何引用 2^{n-j} 次 $f_i[j]$ 的，$j=1,2,\cdots,n$。这种表格化的解法把一个指数时间的递归算法降为了线性时间的算法。

为了更详细地说明重叠子问题的性质，再来看一下矩阵链乘法问题。回顾一下图 15-3，注 344
意 MATRIX-CHAIN-ORDER 在解较高行中的子问题时，要反复查看较低行中的子问题的解。例如，表项 $m[3,4]$ 被引用了 4 次：在计算 $m[2,4]$，$m[1,4]$，$m[3,5]$ 和 $m[3,6]$ 中被引用。如果 $m[3,4]$ 每次都被重新计算，而不是被查看，则所增加的运行时间将相当可观。为了搞清楚这一点，考虑下面确定 $m[i,j]$ 的(低效的)递归程序，计算矩阵链乘积 $A_{i..j}=A_iA_{i+1}\cdots A_j$ 所需的标量乘法的最小次数。该程序直接基于递归式(15.12)。

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i=j
2     then return 0
3  m[i, j] ← ∞
4  for k ← i to j−1
5        do q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
                  +RECURSIVE-MATRIX-CHAIN(p, k+1, j)+p_{i−1} p_k p_j
6           if q<m[i, j]
7              then m[i, j] ← q
8  return m[i, j]
```

⊖ 动态规划要求其子问题既要独立又要重叠，这看上去似乎有些奇怪。虽然这两点要求听起来可能是矛盾的，但它们描述了两种不同的概念，而不是同一个问题的两个方面。如果同一问题的两个子问题不共享资源，则它们就是独立的。对两个子问题来说，如果它们确实是相同的子问题，只是作为不同问题的子问题出现的话，是重叠的，则它们是重叠的。

图 15-5 演示了由调用 RECURSIVE-MATRIX-CHAIN(p，1，4)所产生的递归树。每个结点上都标有参数 i 和 j 的值。注意某些成对的值出现了多次。

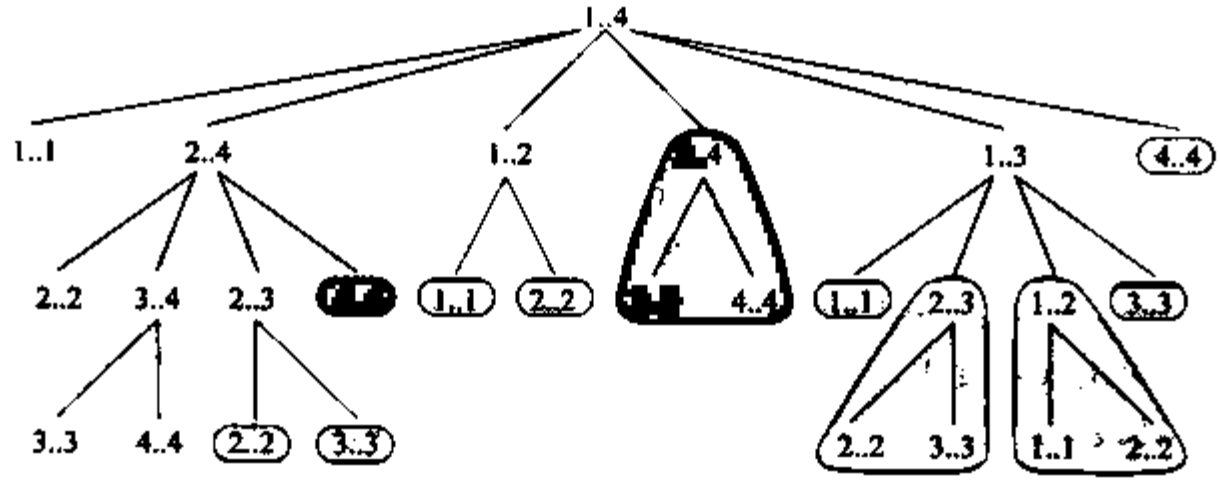


图 15-5　计算 RECURSIVE-MATRIX-CHAIN(p，1，4)的递归树。每个结点包含参数 i 与 j。在加了阴影的子树中，所作的计算可用 MEMOIZED-MATRIX-CHAIN(p，1，4)中的一次查表代替

事实上，可以证明由此递归程序来计算 $m[1, n]$的运行时间至少为 n 的指数。令 $T(n)$表示
345 调用 RECURSIVE-MATRIX-CHAIN 来计算 n 个矩阵的链的一个最优加全部括号要花费的时间。如果假设执行第 1～2 行和 6～7 行各花至少一个单位时间，那么分析该程序可得递归式

$$T(1) \geqslant 1 \quad T(n) \geqslant 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{对于 } n > 1$$

注意对 $i=1, 2, \cdots, n-1$，每一项 $T(i)$一次以 $T(k)$出现，另一次以 $T(n-k)$出现。在这个总和中，提取出 $n-1$ 个 1 以及最前面的 1，可以把递归式重写为

$$T(n) \geqslant 2\sum_{i=1}^{n-1} T(i) + n \tag{15.13}$$

下面利用替换法来证明 $T(n)=\Omega(2^n)$。具体地，我们将证明对所有的 $n \geqslant 1$，有 $T(n) \geqslant 2^{n-1}$。基底很容易证明，因为 $T(1) \geqslant 1 = 2^0$。对 $n \geqslant 2$ 有

$$T(n) \geqslant 2\sum_{i=1}^{n-1} 2^{i-1} + n = 2\sum_{i=0}^{n-2} 2^i + n = 2(2^{n-1} - 1) + n = (2^n - 2) + n \geqslant 2^{n-1}$$

结论得证。所以调用 RECURSIVE-MATRIX-CHAIN(p，1，n)的工作总量至少为 n 的指数。

请读者把这个自顶向下的递归算法与自底向上的动态规划算法作个比较。可以看出后者更加有效，因为它利用了重叠子问题的性质。只有 $\Theta(n^2)$个不同的子问题，动态规划算法对每一个只解一次。另一方面，递归算法对在递归树中重复出现的每个子问题都要重复解一次。当某个问题的自然递归解的递归树中反复包含同一个子问题，而且不同的子问题个数很小，可以考虑能否用动态规划来解这个问题。

重新构造一个最优解

346 在实际应用中，我们通常把每一个子问题中所作的选择保存在一个表格中，这样在需要时，就不必根据已经存储下来的代价信息来重构这方面的信息了。在装配线调度问题中，我们在 $l_i[j]$中保存通过 $S_{i,j}$的最快路线中 $S_{i,j}$的前一站。或者，在填满表格 $f_i[j]$后，只要一些额外的工作，就能确定在通过 $S_{1,j}$的最快路线中哪一站是 $S_{1,j}$的前一站。如果 $f_1[j] = f_1[j-1] + a_{1,j}$，那么 $S_{1,j-1}$是通过 $S_{1,j}$的最快路线中 $S_{1,j}$的前一站。否则就是情况 $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$，所以 $S_{2,j-1}$是 $S_{1,j}$的前一站。对于装配线调度，即使没有表格 $l_i[j]$，重构前一个装配站也只需每个站 $O(1)$的时间。

然而，对于矩阵链乘法问题来说，在重构一个最优解时，有了表格 $s[i, j]$就可以少做不少的工作。假设我们没有表格 $s[i, j]$，而只是在包含最优子问题代价的表格 $m[i, j]$中填写信息。在加全部括号 $A_iA_{i+1}\cdots A_j$ 的一个最优解中，确定使用哪些子问题时有 $j-i$ 个选择，而且 $j-i$ 不是常量。因此，在求出一个给定问题的解时，重构前面所选择过的子问题需要 $\Theta(j-i)=\omega(1)$ 时间。通过把在该处分裂乘积 $A_iA_{i+1}\cdots A_j$ 的矩阵的下标保存在 $s[i, j]$中，就可以在 $O(1)$时间内重构每一个选择。

做备忘录

动态规划有一种变形，它既具有通常的动态规划方法的效率，又采用了一种自顶向下的策略。其思想就是备忘(memoize)原问题的自然但低效的递归算法。像在通常的动态规划中一样，维护一个记录了子问题解的表，但有关填表动作的控制结构更像递归算法。

加了备忘的递归算法为每一个子问题的解在表中记录一个表项。开始时，每个表项最初都包含一个特殊的值，以表示该表项有待填入。当在递归算法的执行中第一次遇到一个子问题时，就计算它的解并填入表中。以后每次遇到该子问题时，只要查看并返回表中先前填入的值即可⊖。

下面是 RECURSIVE-MATRIX-CHAIN 的做备忘录版本：　347

```
MEMOIZED-MATRIX-CHAIN(p)
1  n ← length[p]−1
2  for i ← 1 to n
3      do for j ← i to n
4             do m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)

LOOKUP-CHAIN(p, i, j)
1  if m[i, j]<∞
2    then return m[i, j]
3  if i = j
4      then m[i, j] ← 0
5      else for k ← i to j−1
6             do q ← LOOKUP-CHAIN (p, i, k)+LOOKUP-CHAIN(p, k+1, j)+p_{i−1} p_k p_j
7               if q<m[i, j]
8                 then m[i, j] ← q
9  return m[i, j]
```

像 MATRIX-CHAIN-ORDER 一样，MEMOIZED-MATRIX-CHAIN 对计算出来的 $m[i, j]$值(为计算矩阵 $A_{i..j}$所需的标量乘法的最少次数)维护一张表 $m[1..n, 1..n]$。每个表项初始时包含值∞，以表示该表项有待填入。当执行 LOOKUP-CHAIN(p, i, j)时，在第 1 行中如果 $m[i, j]<\infty$，则该程序就返回先前计算出的代价 $m[i, j]$(第 2 行)。否则，如同在 RECURSIVE-MATRIX-CHAIN 一样计算代价，保存在 $m[i, j]$中，然后返回。(在一个未填入的表项中使用值∞比较方便，因为它是 RECURSIVE-MATRIX-CHAIN 的第 3 行中用来初始化 $m[i, j]$的值)。这样，LOOKUP-CHAIN(p, i, j)总是返回值 $m[i, j]$，但是仅当该程序是第一次以参数 i 和 j 调用时才计算这个值。

图 15-5 说明了 MEMOIZED-MATRIX-CHAIN 是如何较 RECURSIVE-MATRIX-CHAIN 节

⊖ 这种方法预先假设已经知道所有可能的子问题参数，而且已经建立表格位置和子问题之间的关系。另一种方法是把子问题参数当作散列的关键字来记忆。

省时间的。阴影部分的子树表示被查看的(而不是被计算的)值。

像动态规划算法 MATRIX-CHAIN-ORDER 一样，过程 MEMOIZED-MATRIX-CHAIN 的运行时间为 $O(n^3)$。$\Theta(n^2)$个表项中的每一个都由 MEMOIZED-MATRIX-CHAIN 中第 4 行进行一次初始化。我们可以把 LOOKUP-CHAIN 的调用分为两类：

1)在其中 $m[i, j]=\infty$的调用，因此执行第 3～9 行，以及

[348] 2)在其中 $m[i, j]<\infty$的调用，因此 LOOKUP-CHAIN 简单地在第 2 行返回。

第一种类型的调用共有 $\Theta(n^2)$次，每个表项一次。所有第二类调用都是由第一类调用以递归调用的方式发出的。当给定一个 LOOKUP-CHAIN 的调用做递归调用时，它会做 $O(n)$个递归调用。因此，共有 $O(n^3)$个第 2 类调用。每一个第 2 类的调用花费 $O(1)$时间，而每一个第 1 类的调用花费 $O(n)$时间，再加上花在它的递归调用中的时间。所以总的运行时间为 $O(n^3)$。因此，做备忘录将一个 $\Omega(2^n)$时间的算法变为一个 $O(n^3)$时间的算法。

总之，矩阵链乘法问题可以在 $O(n^3)$时间内，用自顶向下的做备忘录算法或自底向上的动态规划算法解决。两种方法都利用了重叠子问题的性质。原问题共有 $\Theta(n^2)$个不同的子问题，这两种方法对每个子问题都只计算一次。如果不使用做备忘录，则自然递归算法就要以指数时间运行，因为它要反复解已经解过的子问题。

在实际应用中，如果所有的子问题都至少要被计算一次，则一个自底向上的动态规划算法通常要比一个自顶向下的做备忘录算法好出一个常数因子，因为前者无需递归的代价，而且维护表格的开销也小些。此外，在有些问题中，还可以用动态规划算法中的表存取模式来进一步减少时间或空间上的需求。或者，如果子问题空间中的某些子问题根本没有必要求解，做备忘录方法有着只解那些肯定要求解的子问题的优点。

练习

15.3-1 在确定矩阵链乘法中最优乘法次数时，下面哪种方法更为有效：枚举对乘积所有可能的加全部括号并逐一计算其乘法的次数，或者运行 RECURSIVE-MATRIX-CHAIN？对你给出的回答加以说明。

15.3-2 画出 2.3.1 节的过程 MERGE-SORT 作用于一个包含 16 个元素的数组上的递归树。请解释在加速一个好的分治算法如 MERGE-SORT 方面，做备忘录方法为什么没有什么效果？

15.3-3 考虑矩阵链乘法问题的一个变形，其目标是加全部括号矩阵序列以最大化而不是最小化

[349] 标量乘法的次数。这个问题是否具有最优子结构？

15.3-4 描述装配线调度问题如何具有重叠子问题。

15.3-5 在动态规划中，我们先求解各个子问题，然后再来决定该选择它们中的哪一个来用在原问题的一个最优解中。Capulet 教授宣称，在寻找一个最优解时，并非总是需要解决所有的子问题。她认为，矩阵链乘法问题的一个最优解总可以在求解子问题之前，通过选择一个矩阵 A_k 来在该位置分裂子乘积 $A_iA_{i+1}\cdots A_j$(通过选择 k 使 $p_{i-1}p_kp_j$ 最小化)。找出一个矩阵链乘法问题的实例，使这个贪心方法得到一个次最优(suboptimal)解。

15.4　最长公共子序列

在生物学应用中，经常要比较两个(或更多)不同有机体的 DNA。一个 DNA 螺旋由一串被称为基的分子组成，可能的基包括腺嘌呤(adenine)、鸟嘌呤(guanine)、胞嘧啶(cytosine)和胸腺嘧啶(thymine)。分别以它们的首字母来代表这些基，一个 DNA 螺旋可以表示为在有穷集合{A, C, G, T}上的一个串。(串的定义参见附录 C)。例如，一个有机体的 DNA 可能为 $S_1=$

ACCGGTCGAGTGCGCGGAAGCCGGCCGAA，而另一个有机体的 DNA 可能为 S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA。将两个 DNA 螺旋作比较的一个目的就是要确定这两个螺旋有多么"相似"，也就是关于这两个有机体的某些度量有多么接近。相似度可以而且已经用很多不同方式来定义。例如，我们可以说两个 DNA 螺旋相似，如果其中一个是另一个的子串。（第 32 章将研究求解这个问题的算法）。在我们的例子中，S_1 或 S_2 都不是另一方的子串。或者，我们可以说两个螺旋相似，如果把其中一个转换成另一个所需改变的数量很小。（思考题 15-3 涉及了这个概念）。另外一个度量螺旋 S_1 和 S_2 相似度的方式是找出第三个螺旋 S_3，在 S_3 中的基也都出现在 S_1 和 S_2 中；而且这些基必须是以相同的顺序出现，但是不必要是连续的。能找到的 S_3 越长，S_1 和 S_2 就越相似。在我们的例子中，最长的螺旋 S_3 是 GTCGTCGGAAGCCGGCCGAA。

我们把最后的这个相似度概念形式化为最长公共子序列问题。一个给定序列的子序列就是该给定序列中去掉零个或者多个元素。以形式化的方式来说，给定一个序列 $X=\langle x_1, x_2, \cdots, x_m\rangle$，另一个序列 $Z=\langle z_1, z_2, \cdots, z_k\rangle$ 是 X 的一个子序列，如果存在 X 的一个严格递增下标序列 $\langle i_1, i_2, \cdots, i_k\rangle$，使得对所有的 $j=1, 2, \cdots, k$，有 $x_{i_j}=z_j$。例如，$Z=\langle B, C, D, B\rangle$ 是 350
$X=\langle A, B, C, B, D, A, B\rangle$ 的一个子序列，相应的下标序列为 $\langle 2, 3, 5, 7\rangle$。

给定两个序列 X 和 Y，称序列 Z 是 X 和 Y 的公共子序列，如果 Z 既是 X 的一个子序列又是 Y 的一个子序列。例如，如果 $X=\langle A, B, C, B, D, A, B\rangle$，$Y=\langle B, D, C, A, B, A\rangle$，则序列 $\langle B, C, A\rangle$ 即为 X 和 Y 的一个公共子序列。但是序列 $\langle B, C, A\rangle$ 不是 X 和 Y 的一个最长公共子序列（Longest-Common-Subsequence，LCS），因为它的长度等于 3，而同为 X 和 Y 的公共子序列 $\langle B, C, B, A\rangle$ 其长度等于 4。序列 $\langle B, C, B, A\rangle$ 是 X 和 Y 的一个 LCS，序列 $\langle B, D, A, B\rangle$ 也是，因为没有长度为 5 或更大的公共子序列。

在最长子序列问题中，给定了两个序列 $X=\langle x_1, x_2, \cdots, x_m\rangle$ 和 $Y=\langle y_1, y_2, \cdots, y_n\rangle$，希望找出 X 和 Y 的最大长度公共子序列。这一节说明 LCS 问题可用动态规划来有效地解决。

步骤 1：描述一个最长公共子序列

解决 LCS 问题的一种强力方法是枚举出 X 的所有子序列，然后逐一检查看其是否为 Y 的子序列，并随时记录所发现的最长子序列。X 的每个子序列对应于 X 的下标集 $\{1, 2, \cdots, m\}$ 的一个子集。X 共有 2^m 个子序列，因此这种方法需要指数时间，这对长序列来说是不实际的。

然而，LCS 问题具有最优子结构性质，下面的定理说明了这一点。我们将看到，子问题的自然类对应于两个输入序列的成对"前缀"。准确地说，给定一个序列 $X=\langle x_1, x_2, \cdots, x_m\rangle$，对 $i=0, 1, \cdots, m$，定义 X 的第 i 个前缀为 $X_i=\langle x_1, x_2, \cdots, x_i\rangle$。例如，如果 $X=\langle A, B, C, B, D, A, B\rangle$，则 $X_4=\langle A, B, C, B\rangle$，而 X_0 是个空序列。

定理 15.1（LCS 的最优子结构） 设 $X=\langle x_1, x_2, \cdots, x_m\rangle$ 和 $Y=\langle y_1, y_2, \cdots, y_n\rangle$ 为两个序列，并设 $Z=\langle z_1, z_2, \cdots, z_k\rangle$ 为 X 和 Y 的任意一个 LCS。

1）如果 $x_m=y_n$，那么 $z_k=x_m=y_n$ 而且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个 LCS。

2）如果 $x_m\neq y_n$，那么 $z_k\neq x_m$ 蕴含 Z 是 X_{m-1} 和 Y 的一个 LCS。

3）如果 $x_m\neq y_n$，那么 $z_k\neq y_n$ 蕴含 Z 是 X 和 Y_{n-1} 的一个 LCS。

证明：1）如果 $z_k\neq x_m$，则可以添加 $x_m=y_n$ 到 Z 中，以得到 X 和 Y 的一个长度为 $k+1$ 的公共子序列，这与 Z 是 X 和 Y 的最长公共子序列的假设相矛盾。因此，必有 $z_k=x_m=y_n$。此时前 351
缀 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的长度为 $(k-1)$ 的公共子序列。证明它就是 LCS，为导出矛盾，假设 X_{m-1} 和 Y_{n-1} 有一个长度大于 $k-1$ 的公共子序列 W，那么将 $x_m=y_n$ 添加到 W 上就会产生一个 X 和 Y 的长度大于 k 的公共子序列，得到矛盾。

2）如果 $z_k\neq x_m$，那么 Z 是 X_{m-1} 和 Y 的一个公共子序列。如果 X_{m-1} 和 Y 有一个长度大于 k

的公共子序列 W，则 W 也应该是 X_m 和 Y 的一个公共子序列，这与 Z 为 X 和 Y 的一个 LCS 的假设矛盾。

3)证明和 2)对称。(证毕) ■

定理 15.1 的特征说明两个序列的一个 LCS 也包含了两个序列的前缀的一个 LCS。这就说明 LCS 问题具有最优子结构性质。稍后我们还会看到，递归解还具有重叠子问题性质。

步骤 2：一个递归解

由定理 15.1 可以知道，在找 $X=\langle x_1, x_2, \cdots, x_m\rangle$ 和 $Y=\langle y_1, y_2, \cdots, y_n\rangle$ 的一个 LCS 时，可能要检查一个或两个子问题。如果 $x_m=y_n$，必须找出 X_{m-1} 和 Y_{n-1} 的一个 LCS。将 $x_m=y_n$ 添加到这个 LCS 上，可以产生 X 和 Y 的一个 LCS。如果 $x_m\neq y_n$，就必须解决两个子问题：找出 X_{m-1} 和 Y 的一个 LCS，以及找出 X 和 Y_{n-1} 的一个 LCS。这两个 LCS 中，较长的就是 X 和 Y 的一个 LCS，因为这些情况涉及了所有的可能，其中一个最优的子问题解必被使用在 X 和 Y 的一个 LCS 中。

可以很容易地看出 LCS 问题中的重叠子问题性质。为找出 X 和 Y 的一个 LCS，可能需要找出 X 和 Y_{n-1} 的一个 LCS 以及 X_{m-1} 和 Y 的一个 LCS。但这两个子问题都包含着找 X_{m-1} 和 Y_{n-1} 的一个 LCS 的子子问题。还有许多其他的子问题共享子子问题。

像在矩阵链乘法问题中一样，LCS 问题的递归解涉及到建立一个最优解的值的递归式。定义 $c[i, j]$ 为序列 X_i 和 Y_j 的一个 LCS 的长度。如果 $i=0$ 或 $j=0$，其中一个的序列长度为 0，因此 LCS 的长度为 0。由 LCS 问题的最优子结构可得递归式

$$c[i,j]=\begin{cases}0 & \text{如果 } i=0 \text{ 或 } j=0\\ c[i-1,j-1]+1 & \text{如果 } i,j>0 \text{ 和 } x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & \text{如果 } i,j>0 \text{ 和 } x_i\neq y_j\end{cases} \tag{15.14}$$

观察这个递归公式，问题中的一个条件限制了我们可能考虑的子问题。当 $x_i=y_j$ 时，可以
352 而且应该考虑寻找 X_{i-1} 和 Y_{j-1} 的 LCS 的子问题。否则，应另外考虑寻找 X_i 和 Y_{j-1} 以及 X_{i-1} 和 Y_j 的 LCS 的两个子问题。在前面已经讨论的动态规划算法(装配线调度和矩阵链乘法)中，没有任何子问题因为原问题的条件而被排除。寻找 LCS 不是唯一的因为问题的条件而排除子问题的动态规划算法。例如，编辑距离(参见思考题 15-3)也具有这个特征。

步骤 3：计算 LCS 的长度

根据(15.14)式，可以很容易地写出一个指数时间的递归算法，来计算两个序列的 LCS 的长度。因为只有 $\Theta(mn)$ 个不同的子问题，所以可以用动态规划来自底向上计算解。

过程 LCS-LENGTH 以两个序列 $X=\langle x_1, x_2, \cdots, x_m\rangle$ 和 $Y=\langle y_1, y_2, \cdots, y_n\rangle$ 为输入。它把 $c[i, j]$ 值填入一个按行计算表项的表 $c[0..m, 0..n]$ 中。(亦即，c 的第一行从左到右填入，然后开始第二行，等等)。它还维护表 $b[1..m, 1..n]$ 以简化最优解的构造。从直觉上看，$b[i, j]$ 指向一个表项，这个表项对应于与在计算 $c[i, j]$ 时所选择的最优子问题的解。该程序返回表 b 和 c；$c[m, n]$ 包含 X 和 Y 的一个 LCS 的长度。

```
LCS-LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
```

```
 8      do for j ← 1 to n
 9           do if x_i = y_j
10                then c[i, j] ← c[i−1, j−1]+1
11                     b[i, j] ← "↖"
12                else if c[i−1, j] ≥ c[i, j−1]
13                        then c[i, j] ← c[i−1, j]
14                             b[i, j] ← "↑"
15                        else c[i, j] ← c[i, j−1]
16                             b[i, j] ← "←"
17   return c and b
```

图 15-6 给出了在序列 $X=\langle A, B, C, B, D, A, B\rangle$ 和 $Y=\langle B, D, C, A, B, A\rangle$ 上，由 LCS-LENGTH 计算出的表。这个程序的运行时间为 $O(mn)$，因为每个表项的计算时间为 $O(1)$。

353

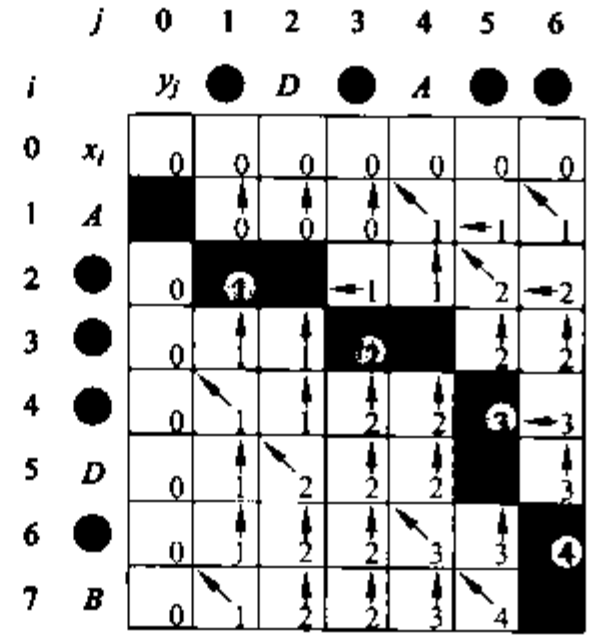


图 15-6　在序列 $X=\langle A, B, C, B, D, A, B\rangle$ 和 $Y=\langle B, D, C, A, B, A\rangle$ 上，由 LCS-LENGTH 计算出的表 c 和 b。第 i 行和第 j 列中的方块包含了 $c[i, j]$ 的值以及指向 $b[i, j]$ 值的箭头。在 $c[7, 6]$ 的项 4，表的右下角为 X 和 Y 的一个 LCS$\langle B, C, B, A\rangle$ 的长度。对 $i, j>0$，项 $c[i, j]$ 仅依赖于是否有 $x_i=y_j$ 及项 $c[i-1, j]$，$c[i, j-1]$ 和 $c[i-1, j-1]$ 的值，这几个项都在 $c[i, j]$ 之前计算。为了重构一个 LCS 的元素，从右下角开始跟踪 $b[i, j]$ 箭头即可；这条路径标示为阴影，这条路径上的每一个"↖"对应于一个使 $x_i=y_j$ 为一个 LCS 的成员的项(高亮)

354

步骤 4：构造一个 LCS

由 LCS-LENGTH 返回的表 b 可以被用来快速构造 $X=\langle x_1, x_2, \cdots, x_m\rangle$ 和 $Y=\langle y_1, y_2, \cdots, y_n\rangle$ 的一个 LCS。首先从 $b[m, n]$ 处开始，沿着箭头在表格中跟踪下去。每当在表项 $b[i, j]$ 中遇到一个"↖"时，即意味着 $x_i=y_j$ 是LCS的一个元素。这种方法是按照反序来找 LCS 的每一个元素的。下面的递归过程按正常的前序输出 X 和 Y 的一个 LCS。初始调用为 PRINT-LCS(b, X, $length[X]$, $length[Y]$)。

```
PRINT-LCS(b, X, i, j)
1  if i = 0 or j = 0
2     then return
3  if b[i, j] = "↖"
4     then PRINT-LCS(b, X, i−1, j−1)
5          print x_i
6  elseif b[i, j] = "↑"
7     then PRINT-LCS (b, X, i−1, j)
8  else PRINT-LCS (b, X, i, j−1)
```

对图 15-6 中的表 b，此程序输出"$B\ C\ B\ A$"。因为在递归的每个阶段 i 和 j 至少有一个要减小，故该过程的运行时间为 $O(m+n)$。

改进代码

一旦设计出某个算法之后，常常可以在时间或空间上对该算法作些改进。对直观的动态规划算法来说尤其如此。有些改变可以简化代码并改进一些常数因子，但并不会带来算法性能方面的渐近改善。其他一些改变则可以在时间和空间上有相当大的渐近节省。

例如，我们可以完全去掉表 b。每个表项 $c[i, j]$ 仅依赖于另外三个 c 表项：$c[i-1, j-1]$，

$c[i-1, j]$和$c[i, j-1]$。给定$c[i, j]$的值，我们可在$O(1)$时间内确定这三个值中的哪一个被用来计算$c[i, j]$的，而不检查表b。这样，利用一个类似于 PRINT-LCS 的过程，在$O(m+n)$时间内即可重构一个 LCS。(练习 15.4-2 要求读者写出此过程的伪代码)。虽然用这种方法节省了$\Theta(mn)$空间，但计算一个 LCS 时所需要的辅助空间并没有渐近地减少，因为表c总是需要占据$\Theta(mn)$空间的。

然而，我们能减少 LCS-LENGTH 的渐近空间需求，因为它一次只需表c的两行：正在被计算的一行和前面一行(实际上，仅需略多于表c一行的空间就可计算一个 LCS 的长度。参见练习 15.4-4)。如果仅要求出一个 LCS 的长度，则这种改进是有用的；如果要重构一个 LCS 的元素，则小的表无法包含足够的信息来使我们在$O(m+n)$时间内重新执行以前各步。

练习

355

15.4-1　确定$\langle 1, 0, 0, 1, 0, 1, 0, 1\rangle$和$\langle 0, 1, 0, 1, 1, 0, 1, 1, 0\rangle$的一个 LCS。

15.4-2　说明如何在不用表b的情况下，通过已计算出的表c和原始序列$X=\langle x_1, x_2, \cdots, x_m\rangle$与$Y=\langle y_1, y_2, \cdots, y_n\rangle$，在$O(m+n)$时间内重构一个 LCS。

15.4-3　请给出一个 LCS-LENGTH 的运行时间为$O(mn)$的做备忘录版本。

15.4-4　说明如何仅用表c中的$2 \cdot \min(m, n)$项以及$O(1)$的额外空间来计算一个 LCS 的长度。然后，说明如何用$\min(m, n)$项以及$O(1)$的额外空间来做到这一点。

15.4-5　请给出一个$O(n^2)$时间的算法，使之能找出一个n个数的序列中最长的单调递增子序列。

***15.4-6**　请给出一个$O(n \lg n)$时间的算法，使之能找出一个n个数的序列中最长的单调递增子序列。(提示：观察长度为i的一个候选子序列的最后一个元素，它至少与长度为$i-1$的一个候选子序列的最后一个元素一样大。通过把候选子序列与输入序列相连接来维护它们)。

15.5　最优二叉查找树

假设我们正在设计一个程序，用于将文章从英语翻译为法语。对于出现在文章内的每一个英文单词，我们需要查看与它等价的法语。执行这些搜索操作的一种方式是建立一棵二叉查找树，它以n个英文单词作关键字以及法语等价词作卫星数据。因为要为文章中的每个单词搜索这棵树，故希望搜索所花费的总时间尽量地小。我们可以使用红黑树或任何其他的平衡二叉查找树，来保证每个单词$O(\lg n)$的搜索时间。但是每个单词出现的频率并不同，而且可能情况是经常使用的单词例如"the"出现在离根部很远的地方，而一个很少使用的单词例如"mycophagist"出

356

现在离根部很近的地方。如此的组织将减慢翻译的速度，因为在二叉查找树中搜索一个关键字时，访问的结点个数等于 1 加上包含该关键字的结点的深度。我们希望文章中出现频繁的单词被放置在距离根部较近的地方[⊖]。而且，文章中可能会有些单词没有法语的翻译，这些单词可能根本就不会出现在二叉查找树中。假设我们知道每个单词出现的频率，应如何组织一棵二叉查找树，使得所有的搜索访问的结点数目最小呢？

我们需要的是一棵*最优二叉查找树*。形式地，给定一个由n个互异的关键字组成的序列$K=\langle k_1, k_2, \cdots, k_n\rangle$，且关键字有序(因此有$k_1<k_2<\cdots<k_n$)，我们想从这些关键字中构造一棵二叉查找树。对每个关键字k_i，一次搜索为k_i的概率是p_i。某些搜索的值可能不在K内，因此还有$n+1$个"虚拟键"$d_0, d_1, d_2, \cdots, d_n$代表不在$K$内的值。具体地，$d_0$代表所有小于$k_1$的值，$d_n$代表所有大于$k_n$的值，而对于$i=1, 2, \cdots, n-1$，虚拟键$d_i$代表所有位于$k_i$和$k_{i+1}$之

⊖　如果这篇文章的主题是可食用的蘑菇，我们就会希望"mycophagist"(食菌者)这个单词出现在离根较近的地方。

间的值。对每个虚拟键 d_i，一次搜索对应于 d_i 的概率是 q_i。图 15-7 显示了在 $n=5$ 个关键字的集合上的两棵二叉查找树。每个关键字 k_i 是一个内部结点，每个虚拟键 d_i 是一个叶子。每次搜索要么成功(找到某个关键字 k_i)，要么失败(找到某个虚拟键 d_i)，因此有

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 \tag{15.15}$$

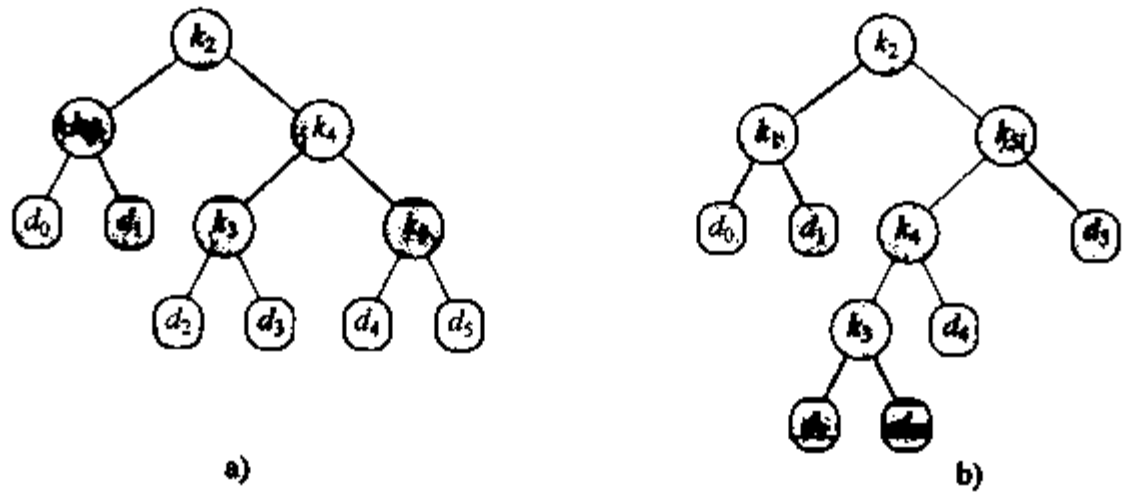


图 15-7　在 $n=5$ 个关键字的集合上的两棵二叉查找树，且概率如下表所示

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

a)一棵期望搜索代价为 2.80 的二叉查找树。b)一棵期望搜索代价为 2.75 的二叉查找树。这棵树是最优的

因为已知了每个关键字和每个虚拟键被搜索的概率，因而可以确定在一棵给定的二叉查找树 T 内一次搜索的期望代价。假设一次搜索的实际代价为检查的结点个数，亦即，在 T 内搜索所发现的结点的深度加上 1。所以在 T 内一次搜索的期望代价为

$$\begin{aligned} \mathrm{E}[T\text{ 的搜索代价}] &= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n} (\mathrm{depth}_T(d_i)+1)\cdot q_i \\ &= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i)\cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i)\cdot q_i \end{aligned} \tag{15.16}$$

其中 depth_T 代表树 T 内一个结点的深度。最后一个等式由等式(15.15)而来。在图 15-7a 中，可以逐个结点计算期望的搜索代价：

结点	深度	概率	贡献
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
合计			2.80

对给定的一组概率，我们的目标是构造一个期望搜索代价最小的二叉查找树。把这种树称作最优二叉查找树。图 15-7b 显示了一棵最优二叉查找树，其概率在图的标题中给出：期望代价是 2.75。这个例子说明一棵最优二叉查找树不一定是一棵整体高度最小的树。我们也不一定总是把有最大概率的关键字放在根部来构造一棵最优二叉查找树。在例子中，关键字 k_5 有所有键的最大搜索概率，而最优二叉查找树的根部是 k_2。(任何以关键字 k_5 为根的二叉查找树的最低期望代价为 2.85)。

357
~
358

如同矩阵链乘法，穷举地检查所有的可能性不会得到一个有效的算法。我们可以将任何 n 个结点的二叉树的结点以关键字 k_1，k_2，…，k_n 来标示，构造一棵最优二叉查找树，然后添加虚拟键作叶子。在思考题 12-4 中，我们曾看到 n 个结点的二叉树共有 $\Omega(4^n/n^{3/2})$个，所以在一个穷举搜索中，我们必须检查指数个数的二叉查找树。毫无疑问，我们将使用动态规划方法来解这个问题。

步骤 1：一棵最优二叉查找树的结构

为描述一棵最优二叉查找树的最优子结构，首先来考察它的子树。一棵二叉查找树的任意一棵子树必定包含在连续范围内的关键字 k_i，…，k_j，对某个 $1 \leqslant i \leqslant j \leqslant n$。另外，一棵含有关键字 k_i，…，k_j 的子树必定也含有虚拟键 d_{i-1}，…，d_j 作为叶子。

现在我们可以陈述最优子结构：如果一棵最优二叉查找树 T 有一棵包含关键字 k_i，…，k_j 的子树 T'，那么这棵子树 T' 对于关键字 k_i，…，k_j 和虚拟键 d_{i-1}，…，d_j 的子问题也必定是最优的。可以应用通常的剪贴思想。如果有一棵子树 T'' 其期望代价比 T' 的小，那么我们可以把 T' 从 T 中剪下，然后贴上 T''，而产生一个期望代价比 T 小的二叉查找树，这与 T 的最优性相矛盾。

使用最优子结构来说明可以根据子问题的最优解，来构造原问题的一个最优解。给定关键字 k_i，…，k_j，其中之一假设是 $k_r(i \leqslant r \leqslant j)$，将是包含这些键的一棵最优子树的根。根 k_r 的左子树包含关键字 k_i，…，k_{r-1}(和虚拟键 d_{i-1}，…，d_{r-1})，右子树包含关键字 k_{r+1}，…，k_j(和虚拟键 d_r，…，d_j)。只要我们检查所有的候选根 k_r，其中 $i \leqslant r \leqslant j$，而且确定所有包含关键字 k_i，…，k_{r-1} 和 k_{r+1}，…，k_j 的最优二叉查找树，就保证可以找到一棵最优的二叉查找树。

要注意的是关于“空”子树。假设在一棵包含关键字 k_i，…，k_j 的子树中，选取 k_i 作为根。根据上面的推论，k_i 的左子树包含关键字 k_i，…，k_{i-1}。很自然地会把这个序列解释成不含有任何关键字。记住，这些子树同时也包含虚拟键。我们采用一种约定，即一棵包含关键字的子树没有真实的关键字但包含单一的虚拟键 d_{i-1}。对称地，如果我们选择 k_j 来当作根，则 k_j 的右子树包含关键字 k_{j+1}，…，k_j；这棵右子树不含有真实的关键字，但其确实包含虚拟键 d_j。

步骤 2：一个递归解

现在就可以来递归地定义一个最优解的值了。选取子问题域为找一个包含关键字 k_i，…，k_j 的最优二叉查找树，其中 $i \geqslant 1$，$j \leqslant n$ 而且 $j \geqslant i-1$。(当 $j=i-1$ 时没有真实的关键字；只有虚拟

359

键 d_{i-1})。定义 $e[i, j]$为搜索一棵包含关键字 k_i，…，k_j 的最优二叉查找树的期望代价。最终，我们要计算 $e[1, n]$。

当 $j=i-1$ 时出现简单情况。此时只有虚拟键 d_{i-1}。期望的搜索代价是 $e[i, i-1]=q_{i-1}$。

当 $j \geqslant i$ 时，需要从 k_i，…，k_j 中选择一个根 k_r，然后用关键字 k_i，…，k_{r-1} 来构造一棵最优二叉查找树作为其左子树，并用关键字 k_{r+1}，…，k_j 来构造一棵最优二叉查找树作为其右子树。当一棵树成为一个结点的子树时，它的期望搜索代价怎么变化？子树中每个结点的深度增加 1。由公式(15.16)，这个子树的期望搜索代价增加为子树中所有概率的总和。对一棵有关键字 k_i，…，k_j 的子树，定义概率的总和为

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \tag{15.17}$$

因此，如果 k_r 是一棵包含关键字 k_i，…，k_j 的最优子树的根，则有

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$$

注意

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

将 $e[i, j]$重写为

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j) \tag{15.18}$$

递归式(15.18)假设我们知道该采用哪一个结点 k_r 作为根。我们选择有最低期望搜索代价的结点作为根，从而得到最终的递归公式：

$$e[i,j] = \begin{cases} q_{i-1} & \text{如果 } j = i-1 \\ \min\limits_{i \leqslant r \leqslant j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{如果 } i \leqslant j \end{cases} \tag{15.19}$$

$e[i, j]$的值是在最优二叉查找树中的期望搜索代价。为有助于我们记录最优二叉查找树的结构，定义 $root[i, j]$为 k_r 的下标 r，对 $1 \leqslant i \leqslant j \leqslant n$，$k_r$ 是包含关键字 k_i，…，k_j 的一棵最优二叉查找树的根。虽然后面将看到如何计算 $root[i, j]$的值，我们把根据这些值构造最优二叉查找树的工作留作练习 15.5-1。

步骤 3：计算一棵最优二叉查找树的期望搜索代价

此时，读者可能已经注意到了最优二叉查找树与矩阵链乘法的特征之间有一些相似。在二者的问题域中，子问题由连续的下标子范围组成。公式(15.19)直接递归的实现和直接递归的矩阵链乘法算法一样低效。反之，我们把 $e[i, j]$保存在表 $e[1..n+1, 0..n]$中。第一维的下标需要达到 $n+1$ 而不是 n，原因是为了有一个只包含虚拟键 d_n 的子树，我们需要计算和保存 $e[n+1, n]$。第二维的下表需要从 0 开始，是为了保存 $e[1, 0]$。使用 $j \geqslant i-1$ 的表项 $e[i, j]$和表 $root[i, j]$来记录包含关键字 k_i，…，k_j 的子树的根。这个表只使用 $1 \leqslant i \leqslant j \leqslant n$ 的表项。 360

为了提高效率，还需要一个表格。不是每当计算 $e[i, j]$时都从头开始计算 $w(i, j)$(这将用 $\Theta(j-i)$步加法)，而是把这些值保存在表 $w[1..n+1, 0..n]$中。对基础情况，计算 $w[i, i-1] = q_{i-1}$，其中 $1 \leqslant i \leqslant n$。对 $j \geqslant i$，计算

$$w[i,j] = w[i,j-1] + p_j + q_j \tag{15.20}$$

因此，可以计算出 $\Theta(n^2)$个 $w[i, j]$的值，每一个值需要 $\Theta(1)$的计算时间。

下面的伪代码以概率 p_1，…，p_n 和 q_1，…，q_n 以及规模 n 为输入、返回表 e 和 $root$。

```
OPTIMAL-BST(p, q, n)
 1  for i ← 1 to n+1
 2       do e[i, i-1] ← q_{i-1}
 3          w[i, i-1] ← q_{i-1}
 4  for l ← 1 to n
 5       do for i ← 1 to n-l+1
 6               do j ← i+l-1
 7                  e[i, j] ← ∞
 8                  w[i, j] ← w[i, j-1]+p_j+q_j
 9                  for r ← i to j
10                       do t ← e[i, r-1]+e[r+1, j]+w[i, j]
11                          if t<e[i, j]
12                            then e[i, j] ← t
```

```
13                    root[i, j] ← r
14  return e and root
```

根据上面的描述以及与 15.2 节 MATRIX-CHAIN-ORDER 过程的相似性，此过程的操作应该是相当直观的。1～3 行中的 for 循环初始化 $e[i, i-1]$和 $w[i, i-1]$的值。4～13 行的 for 循环利用递归式(15.19)和式(15.20)来计算 $e[i, j]$和 $w[i, j]$，对所有的 $1 \leqslant i \leqslant j \leqslant n$。在第一次迭代
361 中，此时 $l=1$，循环计算 $e[i, i]$和 $w[i, i]$，对 $i=1, 2, \cdots, n$。第二次迭代，$l=2$，计算 $e[i, i+1]$和 $w[i, i+1]$，对 $i=1, 2, \cdots, n-1$，等等。最内层的 for 循环，在 9～13 行，尝试每个下标 r 以确定使用哪个关键字 k_r 来作为包含关键字 $k_i, \cdots, k_j$ 的最优二叉查找树的根。无论何时发现一个更好的关键字来作为根，这个 for 循环在 $root[i, j]$中保存下标 r 的当前值。

图 15-8 显示在图 15-7 中给出的关键字分布上，程序 OPTIMAL-BST 计算出的表 $e[i, j]$，$w[i, j]$和 $root[i, j]$。和在矩阵链乘法例子中一样，表格被旋转以使对角线保持水平。OPTIMAL-BST 自底向上计算行，在每行中从左到右计算。

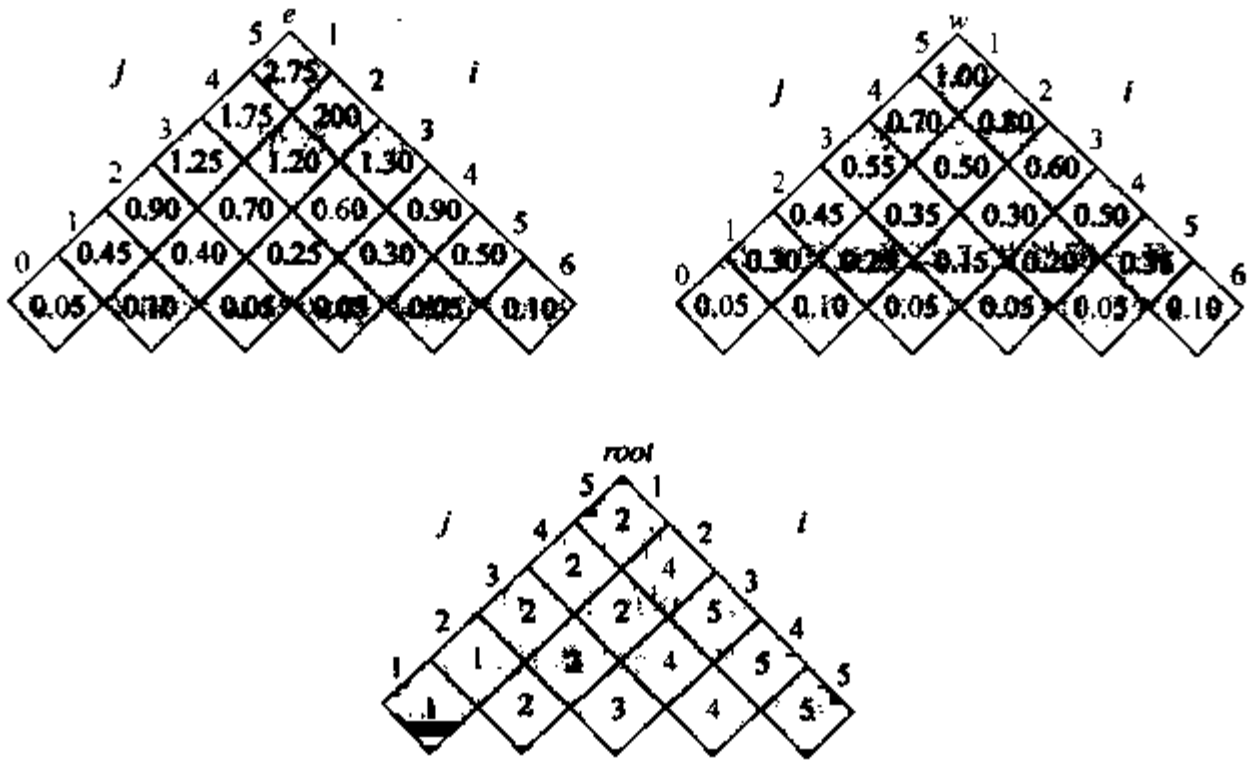


图 15-8　在图 15-7 中显示的关键字分布上，OPTIMAL-BST 计算出的表 $e[i, j]$，$w[i, j]$和 $root[i, j]$。这些表经过旋转，以使对角线是水平的

OPTIMAL-BST 过程需要 $\Theta(n^3)$的运行时间，这与 MATRIX-CHAIN-ORDER 是一样的。要看出执行时间为 $O(n^3)$是比较容易的，因为 for 循环有三层嵌套，而且每个循环的下标有至多 n 个值。OPTIMAL-BST 和 MATRIX-CHAIN-ORDER 中的循环下标并没有刚好相同的界，但是它们在所有方向上都至多差 1。所以，如同 MATRIX-CHAIN-ORDER，OPTIMAL-BST 过程也需
362 要 $\Omega(n^3)$的运行时间。

练习

15.5-1　写出过程 CONSTRUCT-OPTIMAL-BST(*root*)的伪代码，给定表 *root*，输出一棵最优二叉查找树的结构。对图 15-8 中的例子，你的伪代码应该输出结构

k_2 是根

k_1 是 k_2 的左孩子

d_0 是 k_1 的左孩子

d_1 是 k_1 的右孩子
k_5 是 k_2 的右孩子
k_4 是 k_5 的左孩子
k_3 是 k_4 的左孩子
d_2 是 k_3 的左孩子
d_3 是 k_3 的右孩子
d_4 是 k_4 的右孩子
d_5 是 k_5 的右孩子
和图 15-7b 所示的最优二叉查找树相一致。

15.5-2 对有 $n=7$ 个关键字以及如下概率的集合，确定一棵最优二叉查找树的代价和结构：

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

15.5-3 假设不维护表 $w[i, j]$，我们在 OPTIMAL-BST 的第 8 行直接从公式(15.17)计算 $w(i, j)$的值，并在第 10 行是用这个计算的值。这个改变对 OPTIMAL-BST 的渐近执行时间有什么影响?

***15.5-4** Knuth[184]已经证明对于所有的 $1 \leqslant i < j \leqslant n$，总存在最优子树的根使得 $root[i, j-1] \leqslant root[i, j] \leqslant root[i+1, j]$。利用这个事实来修改 OPTIMAL-BST 程序，使其在 $\Theta(n^2)$时间内执行。

363

思考题

15-1　双调欧几里得旅行商问题

欧几里得旅行商问题是对平面上给定的 n 个点确定一条连接各点的最短闭合旅程的问题。图 15-9a 给出了一个 7 个点问题的解。这个问题的一般形式是 NP 完全的，故其解需要多于多项式的时间(参见第 34 章)。

J. L. Bentley 建议通过只考虑双调旅程来简化问题，这种旅程即为从最左点开始，严格地从左到右直至最右点，然后严格地从右到左直至出发点。图 15-9b 显示了同样的 7 个点问题的最短双调路线。在这种情况下，多项式时间的算法是可能的。

描述一个确定最优双调路线的 $O(n^2)$时间的算法。可以假设任何两点的 x 坐标都不相同。(提示：从左到右扫描，保持路线两部分的最优概率)。

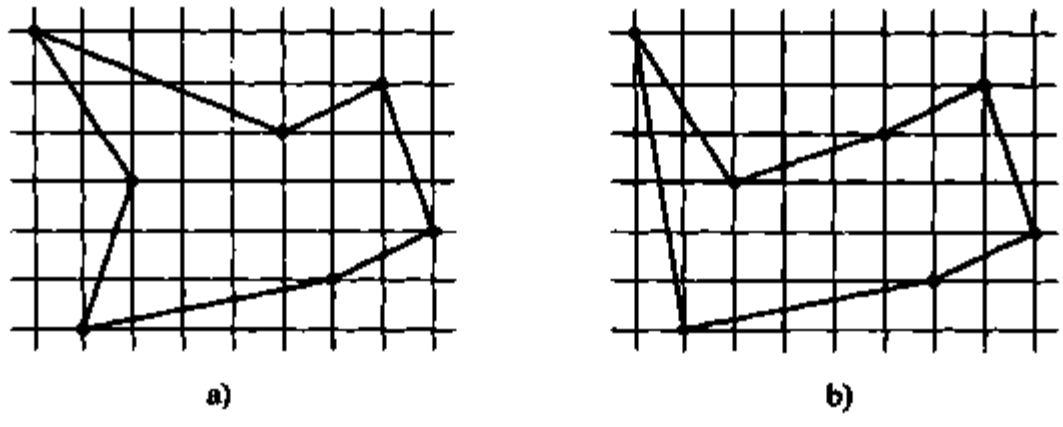


图 15-9　在一个单位栅格上显示的平面上七个点。a)最短闭合路线，长度大约是 24.89。这个路线不是双调的。b)相同点集合上的最短双调闭合路线。长度大约是 25.58

15-2　整齐打印

考虑在一个打印机上整齐地打印一段文章的问题。输入的正文是 n 个长度分别为 l_1，l_2，…，l_n（以字符个数度量）的单词构成的序列。我们希望将这个段落在一些行上整齐地打印出来，每行至多 M 个字符。“整齐度”的标准如下。如果某一行包含从 i 到 j 的单词，$i<j$，且单词之间只留一个空，则在行末多余的空格字符个数为 $M-j+i-\sum_{k=i}^{j} l_k$，它必须是非负值才能让该行容纳这些单词。我们希望所有行（除了最后一行）的行末多余空格字符个数的立方的总和最小。请给出一个动态规划算法，来在打印机上整齐地打印一段有 n 个单词的文章。分析所给算法的执行时间和空间需求。

15-3　编辑距离

为了把一个源文本串 $x[1..m]$ 转换成一个目标串 $y[1..n]$，可以执行各种转换操作。目的是，给定 x 和 y，产生一系列的转换把 x 变为 y。我们使用一个数组 z，假设其容量足以存储需要的所有字符来保存中间结果。z 初始为空，结束时应该有 $z[j]=y[j]$，对于 $j=1$，2，…，n。保持 x 的当前下标 i 和 z 的当前下标 j，允许操作改变 z 以及这些下标。初始时，$i=j=1$。我们需要在转换期间检查 x 的每个字符，这表示在序列的转换操作的结尾，必有 $i=m+1$。

共有 6 种转换操作：

1）从 x 中复制（copy）一个字符到 z 内，通过设置 $z[j]\leftarrow x[i]$，然后将 i 和 j 都增加 1。这个操作检查 $x[i]$。

2）把 x 中的一个字符替换（replace）为另一个字符 c，通过设置 $z[j]\leftarrow c$，然后将 i 和 j 都增加 1。这个操作检查 $x[i]$。

3）删除（delete）x 中的一个字符，通过增加 i 而保持 j 不变。这个操作检查 $x[i]$。

4）插入（insert）一个字符 c 到 z 内，通过设置 $z[j]\leftarrow c$，然后增加 j 但保持 i 不变。这个操作不检查 x 中的任何字符。

5）交换（twiddle，即 exchange）下两个字符，把它们从 x 中复制到 z 中，但以相反的顺序复制；通过设置 $z[j]\leftarrow x[i+1]$ 和 $z[j+1]\leftarrow x[i]$，然后设置 $i\leftarrow i+2$ 和 $j\leftarrow j+2$ 来实现它。这个操作检查 $x[i]$ 和 $x[i+1]$。

364
~
365

6）通过设置 $i\leftarrow m+1$ 消灭（kill）x 中余下的字符。这个操作检查 x 中所有尚未被检查的字符。如果执行这个操作，则它必是最后的操作。

作为一个例子，把原字符串 algorithm 转换为目标字符串 altruistic 的一种方式是使用如下的操作序列，其中带下划线的字符是操作执行后的 $x[i]$ 和 $z[j]$。

操作	x	z
初始串	algorithm	_
copy	algorithm	a_
copy	algorithm	al_
replace by t	algorithm	alt_
delete	algorithm	alt_
copy	algorithm	altr_
insert u	algorithm	altru_
insert i	algorithm	altrui_
insert s	algorithm	altruis_
twiddle	algorithm	altruisti_
insert c	algorithm	altruistic_
kill	algorithm_	altruistic_

注意还有其他一些转换操作序列可以把 algorithm 转换成 altruistic。

每一个转换操作都有一个相关的代价。一个操作的代价依赖于特定的应用，但是我们假设每个操作的代价都是已知的常数。此外，还假设复制和替换的单个代价小于删除和插入操作的合并代价；否则，复制和替换操作就不会被使用。一个给定的转换操作序列的代价等于序列中单个操作的代价之和。对于上面的序列，把 algorithm 转换成 altruistic 的代价为

(3 · cost(copy))+cost(replace)+cost(delete)+(4 · cost(insert))+cost(twiddle)+cost(kill)

a)给定两个序列 $x[1..m]$和 $y[1..n]$和转换操作代价的集合，从 x 到 y 的编辑距离是把 x 转换成 y 的代价最低的操作序列的代价。请描述一个动态规划算法，它找出从 $x[1..m]$到 $y[1..n]$的编辑距离，并输出一个最优的操作序列。分析所述算法的运行时间和空间需求情况。

编辑距离问题是对齐两个 DNA 序列问题(例如，参考 Setubal 和 Meidanis[272，3.2 节])的一般化。有许多通过对齐的方法来度量两个 DNA 序列的相似度。其中一种对齐序列 x 和 y 的方法是在两个序列的任意位置上插入空格(包括两端)，使得结果序列 x' 和 y' 有相同的长度而且不在同一位置均为空格(也就是，没有位置 j 让 $x'[j]$和 $y'[j]$都是空格)。然后，给每个位置一个“分数”。位置 j 以如下方式得到一个分数：

- +1，如果 $x'[j]=y'[j]$而且都不是空格。
- −1，如果 $x'[j]\neq y'[j]$而且都不是空格。
- −2，如果 $x'[j]$或 $y'[j]$是空格。

一个对齐的分数是单个位置的分数之和。例如，给定序列 $x=$ GATCGGCAT 和 $y=$ CAATGTGAATC，一个对齐为：

```
G  ATCG  GCAT
CAAT  GTGAATC
-*++*+*+-++*
```

位置下的+表示该位置的分数是+1，−表示分数是−1，*表示分数是−2，因此这个对齐的总分数为 $6\cdot 1-2\cdot 1-4\cdot 2=-4$。

b)解释如何使用转换操作复制、替换、删除、插入、交换和消灭的一个子集，把找一个最优对齐的问题转换成一个编辑距离问题。

15-4　计划一个公司聚会

Stewart 教授是一家公司总裁的顾问，这家公司计划一个公司聚会。这个公司有一个层次式的结构；也就是，管理关系形成一棵以总裁为根的树。人事部给每个雇员以喜欢聚会的程度来排名，这是一个实数。为了使每个参加者都喜欢这个聚会，总裁不希望一个雇员和他(她)的直接上司同时参加。

Stewart 教授面对一棵描述公司结构的树，使用了 10.4 节描述的左子女、右兄弟表示法。树中每个结点除了包含指针，还包含雇员的名字以及该雇员喜欢聚会的排名。描述一个算法，它生成一张客人列表，使得客人喜欢聚会的程度的总和最大。分析你的算法的执行时间。

15-5　Viterbi 算法

我们可用一个有向图 $G=(V, E)$上的动态规划做语音识别。每条边 $(u, v)\in E$ 上标以选自有限的声音集 Σ 中的一种声音 $\sigma(u, v)$。这种标记图是一个人说一种有限语言的形式化模型。图中从某一特别顶点 $v_0\in V$ 开始的一条路径对应于该模型产生的一个可能声音序

367 列。某一有向路径的标记定义为该路径上所有边的标记的连接。

a)请描述一个有效的算法，对给定的边标记图 G(其中有一个特别顶点 v_0)和 Σ 中的一个字符序列 $s=(\sigma_1, \sigma_2, \cdots, \sigma_k)$，返回 G 的一条始于 v_0、标记为 s 的路径，如果这样的路径存在的话。否则，算法返回 NO-SUCH-PATH。分析算法的执行时间。(提示：可参考第 22 章的有关概念)。

现在，假设每一条边 $(u, v)\in E$ 还被赋予相关联的非负概率 $p(u, v)$，它表示从顶点 u 开始遍历边 (u, v) 并因此产生相应的声音的可能性。自任一顶点出发的边的概率之和等于 1。一条路径的概率定义为其上所有边的概率的乘积。我们可把开始于 v_0 的一条路径的概率视为从 v_0 开始的一次"随机遍历"沿指定路径的概率，其中在顶点 u 上选择走哪条边是根据离开 u 的可用边的概率随机确定的。

b)扩展(a)部分的答案，使得当返回一条路径时，它是从 v_0 开始并具有标记 s 的最可能路径。分析所给算法的执行时间。

15-6　在棋盘上移动

假设有一张 $n\times n$ 个方格的棋盘以及一个棋子。必须根据以下的规则把棋子从棋盘的底边移动到棋盘的顶边。在每一步你可以把棋子移动到三个方格中的一个：

1)正上方的方格，

2)左上方的方格(只能当这个棋子不在最左列的时候)，

3)右上方的方格(只能当这个棋子不在最右列的时候)。

每次从方格 x 移动到方格 y，会得到 $p(x, y)$ 块钱。已知所有对 (x, y) 的 $p(x, y)$，只要从 x 到 y 的移动是合法的。不要假设 $p(x, y)$ 是正值。

请给出一个计算移动方式集合的算法，把棋子从棋盘底边的某个地方移动到棋盘顶边的某个地方，同时收集尽可能多的钱。你的算法可以自由选择底边的任意方格作为起始点，顶边上的任意方格作为目的点，来最大化一路上收集到的钱数。你的算法的执行时间
368 是多少？

15-7　达到最高效益的调度

假设有一台机器，以及在此机器上处理的 n 个作业 $a_1, a_2, \cdots, a_n$ 的集合。每个作业 a_j 有一个处理时间 t_j，效益 p_j，以及最后期限 d_j。机器在一个时刻只能处理一个作业，而且作业 a_j 必须在 t_j 连续时间单位内不间断地运行。如果作业 a_j 在最后期限 d_j 之前完成，则获得效益 p_j，但如果在最后期限之后才完成，则没有效益。请给出一个算法，来寻找能获得最大量效益的调度，假设所有的处理时间都是 1 到 n 之间的整数。你的算法的执行时间是多少？

本章注记

R. Bellman 在 1955 年开始系统地研究动态规划。单词"programming"在这里以及线性规划中，都是指使用一种表格化的解法。虽然在之前已经知道最优化技术含有动态规划的元素，Bellman 给这个领域提供了坚实的数学基础[34]。

Hu 和 Shing[159, 160]给出了矩阵链乘法的一个 $O(n \lg n)$ 时间的算法。

最长公共子序列问题的 $O(mn)$ 时间算法是一个一般的算法。Knuth[63]提出了 LCS 问题的次二次方算法是否存在的问题。Masek 和 Paterson[212]给出一个在 $O(mn/\lg n)$ 时间内执行的算法

来肯定地回答这个问题，其中 $n \leqslant m$ 而且此序列是从一个有界集合中而来。在输入序列中没有元素出现超过一次的特殊情况中，Szymanski[288]说明这个问题可在 $O((n+m)\lg(n+m))$ 时间内解决。这些结果中有许多延伸到了计算字符串编辑距离的问题上(思考题 15-3)。

一篇由 Gilbert 和 Moore[114]撰写的关于可变长二元编码的早期论文有这样的应用：在所有的概率 p_i 都是 0 的情况下构造最优二叉查找树；这篇论文包含一个 $O(n^3)$ 时间的算法。Aho、Hopcroft 和 Ullman[5]给出了 15.5 节中的算法。练习 15.5-4 来自 Knuth[184]。Hu 和 Tucker[161]设计了一个算法，它在所有的概率 p_i 都是 0 的情况下，使用 $O(n^2)$ 的时间和 $O(n)$ 的空间；随后，Knuth[185]把时间降到 $O(n\lg n)$。 369

第16章 贪心算法

适用于最优化问题的算法往往包含一系列步骤，每一步都有一组选择。对许多最优化问题来说，采用动态规划方法来决定最佳选择就有点“杀鸡用牛刀”了，只要采用另一些更简单有效的算法就行了。贪心算法是使所做的选择看起来都是当前最佳的，期望通过所做的局部最优选择来产生出一个全局最优解。这一章讨论可由贪心算法解决的最优化问题。在阅读本章之前，读者应先看一看第15章中有关动态规划的内容，尤其是第15.3节。

贪心算法对大多数优化问题来说能产生最优解，但也不一定总是这样的。在16.1节中，要首先看一个简单而不可轻视的问题，即活动选择问题。利用贪心算法可以很有效地计算出它的解。在讨论贪心算法之前，首先考虑动态规划方法，然后，我们证明总能够用贪心的选择得到其最优解。第16.2节回顾贪心方法的基本要素，给出一种证明贪心算法正确性的方法，这种方法比16.1节中沿用动态规划方法来证明贪心算法正确性的那套做法更为直接。16.3节要给出贪心技术的一个重要应用，即数据压缩(Huffman)编码的设计。在16.4节中，要研究一类称为“拟阵”(matroid)的组合结构某些理论。对这种组合结构，贪心算法总能产生出最优解。最后，16.5节通过带期限和惩罚的单位时间作业调度问题来说明拟阵的应用。

贪心方法是一种很有效的方法，适用于一大类问题。本书后面各章将给出许多可被视为贪心法的应用的算法，如最小生成树(第23章)，Dijkstra的单源最短路径(第24章)以及Chvátal的贪心集合覆盖启发式(第35章)。最小生成树算法是贪心法的一个经典例子。虽然本章可以独立于第23章来阅读，但若把它们结合起来看是有益处的。

16.1 活动选择问题

第一个例子是对几个互相竞争的活动进行调度，它们都要求以独占的方式使用某一公共资源。调度的目标是找出一个最大的相互兼容活动集合。假设有一个需要使用某一资源(如教室等)的n个活动组成的集合$S=\{a_1, a_2, \cdots, a_n\}$。该资源一次只能被一个活动占用。每个活动$a_i$有个开始时间$s_i$和结束时间$f_i$，且$0 \leqslant s_i < f_i < \infty$。一旦被选择后，活动$a_i$就占据半开时间区间$[s_i, f_i)$。如果区间$[s_i, f_i)$和$[s_j, f_j)$互不重叠，称活动$a_i$和$a_j$是兼容的(亦即，如果$s_i \geqslant f_j$或者$s_j \geqslant f_i$)，活动$a_i$和$a_j$是兼容的。活动选择问题就是要选择出一个由互相兼容的问题组成的最大子集合。例如，讨论下面的活动集合S，其中各活动已按结束时间的单调递增顺序进行了排序：

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

(后面将会简短地介绍按排序考虑活动的好处。)对这个例子来说，子集$\{a_3, a_9, a_{11}\}$由相互兼容的活动组成。然而，它不是最大的子集，子集$\{a_1, a_4, a_8, a_{11}\}$更大。事实上，$\{a_1, a_4, a_8, a_{11}\}$是一个最大的相互兼容活动子集；另外一个最大子集是$\{a_2, a_4, a_9, a_{11}\}$。

下面分几个步骤来解决此问题。此问题通过动态规划方法被分为两个子问题，然后将两个

子问题的最优解整合成原问题的一个最优解。在确定该将哪些子问题用于最优解时，要考虑几种选择。读者稍后会发现，贪心算法只需考虑一个选择(亦即，贪心的选择)；在做贪心选择时，子问题之一必是空的，因此只留下一个非空子问题。基于这些观察，我们将找到一种递归贪心算法来解决活动调度问题，并将递归算法转化为迭代算法，以完成贪心方法的过程。虽然本节中介绍的步骤比典型的贪心算法的设计过程更为复杂，但它们说明贪心算法和动态规划之间的关系。

活动选择问题的最优子结构

如上文所述，首先为活动选择问题找到一个动态规划解。与在第 15 章中一样，我们首先找 371
到问题的最优子结构，然后利用这一结构，根据子问题的最优解来构造出原问题的一个最优解。

在第 15 章中我们知道，需要定义一个合适的子问题空间。首先定义如下的集合：

$$S_{ij} = \{a_k \in S : f_i \leqslant s_k < f_k \leqslant s_j\}$$

也就是说，S_{ij} 是 S 中活动的子集，其中，每个活动都在活动 a_i 结束之后开始，且在活动 a_j 开始之前结束。实际上，S_{ij} 包含了所有与 a_i 和 a_j 兼容的活动，并且与不迟于 a_i 结束和不早于 a_j 开始的活动兼容。为了能表示完整的问题，我们加入了虚构活动 a_0 与 a_{n+1}，用惯例 $f_0 = 0$ 以及 $S_{n+1} = \infty$。于是，$S = S_{0,n+1}$，i 和 j 的范围为 $0 \leqslant i, j \leqslant n+1$。

我们可以按如下方法对 i 和 j 的范围作进一步的限制。假设活动已按照结束时间的单调递增顺序排序：

$$f_0 \leqslant f_1 \leqslant f_2 \leqslant \cdots \leqslant f_n < f_{n+1} \tag{16.1}$$

我们断言，当 $i \geqslant j$ 时，$S_{ij} = \varnothing$。为什么呢？假设存在一个活动 $a_k \in S_{ij}$，其中 $i \geqslant j$，则在已排的序中 a_i 在 a_j 后面。因为有 $f_i \leqslant s_k < f_k \leqslant s_j < f_j$，则 $f_i < f_j$，这与在已排的序中 a_i 在 a_j 后面的假设相矛盾。我们可以得出一个结论：设活动按结束时间的单调递增顺序排序，子问题空间被用来从 S_{ij} 中选择最大兼容活动子集，其中 $0 \leqslant i < j \leqslant n+1$，而且所有其他的 S_{ij} 是空的。

要看出活动选择问题的子结构，可以考虑一下某个非空子问题 S_{ij}[⊖]，并假设 S_{ij} 的解包含某活动 a_k，其中 $f_i \leqslant s_k < f_k \leqslant s_j$。用活动 a_k 生成两个子问题：S_{ik}(在 a_i 结束后开始且在 a_k 开始前结束的活动)，以及 S_{kj}(在 a_k 结束后开始且在 a_j 开始前结束的活动)，它们都是 S_{ij} 的子集。S_{ij} 的解是连同活动 a_k 在内的 S_{ik} 和 S_{kj} 解的并集。因此，S_{ij} 解中的活动数等于 S_{ik} 解的大小加上 S_{kj} 解的大小，再加上 1(因为 a_k)。

现在来讨论此问题的最优子结构问题。假设现在已知 S_{ij} 的最优解 A_{ij} 包含活动 a_k，则包含在 S_{ij} 最优解中的针对 S_{ik} 的解 A_{ik} 和针对 S_{kj} 的解 A_{kj} 必定也是最优的。通常的“剪切和粘贴”技术在这里很有用。如果 S_{ik} 有一个解 A'_{ik} 比 A_{ik} 包含更多的活动，则将 A_{ik} 从 A_{ij} 中剪切掉，并粘贴到 372
A'_{ik} 中，所以产生出 S_{ij} 的另一个解，该解比 A_{ij} 包含更多的活动。由于我们假设 A_{ij} 是最优解，则出现了矛盾。类似地，如果 S_{kj} 有一个解 A'_{kj} 比 A_{kj} 包含更多的活动，则将 A_{kj} 换成 A'_{kj}，以产生一个比 A_{ij} 包含更多活动的解。

现在利用最优子结构来证明，可以根据子问题的最优解构造出原问题的一个最优解。我们知道，一个非空子问题 S_{ij} 的任意解中必包含了某项活动 a_k，而 S_{ij} 的任一最优解中都包含了其子问题实例 S_{ik} 和 S_{kj} 的最优解。因此，我们可以构造出 S_{ij} 中的最大兼容活动子集。将问题分成两

⊖ 我们有时会将集合 S_{ij} 称为子问题，而不只是称其为活动的集合。从上下文中，就可以很清楚地看出 S_{ij} 到底是指活动集，还是指其输入为该集合的子问题。

个子问题(找出 S_{ik} 和 S_{kj} 的最大兼容活动子集)，找出这些子问题的最大兼容活动子集 A_{ik} 和 A_{kj}，而后形成最大兼容活动子集 A_{ij} 如下：

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \tag{16.2}$$

整个问题的一个最优解也是 $S_{0,n+1}$ 的一个解。

一个递归解

动态规划方法的第二步是递归地定义最优解的值。对于活动选择问题，设 $c[i, j]$ 为 S_{ij} 中最大兼容子集中的活动数。当 $S_{ij}=\varnothing$ 时，有 $c[i, j]=0$；特别地，当 $i \geqslant j$ 时，$c[i, j]=0$。

现在考虑一个非空子集 S_{ij}。我们知道，如果 a_k 在 S_{ij} 的最大兼容子集中被使用，则子问题 S_{ik} 和 S_{kj} 的最大兼容子集也被使用。使用等式(16.2)，有递归式：

$$c[i,j] = c[i,k] + c[k,j] + 1$$

此递归式假设 k 值已知，然而我们并不知道。k 的取值有 $j-i-1$ 种，它们是 $k=i+1, \cdots, j-1$。由于 S_{ij} 的最大子集一定使用了 k 值中的某一个，故检查过所有的 k 值后，就可以找到最好的一个。因此，$c[i, j]$ 的完整递归定义变为：

$$c[i,j] = \begin{cases} 0 & \text{如果 } S_{ij} = \varnothing \\ \max\limits_{i<k<j}\{c[i,k] + c[k,j] + 1\} & \text{如果 } S_{ij} \neq \varnothing \end{cases} \tag{16.3}$$

将动态规划解转化为贪心解

373 到此为止，要写出一个表格化的、自底向上的、基于递归式(16.3)的动态规划算法是不难的。事实上，练习 16.1-1 的要求就是这样的。然而，有两个关键的发现可以帮助简化我们的解。

定理 16.1　对于任意非空子问题 S_{ij}，设 a_m 是 S_{ij} 中具有最早结束时间的活动：

$$f_m = \min\{f_k : a_k \in S_{ij}\}$$

那么，

1)活动 a_m 在 S_{ij} 的某最大兼容活动子集中被使用。

2)子问题 S_{im} 为空，所以选择 a_m 将使子问题 S_{mj} 为唯一可能非空的子问题。

证明：因为第 2 部分较简单，所以首先证明第 2 部分。假设 S_{im} 非空，因此有某活动 a_k 满足 $f_i \leqslant s_k < f_k \leqslant s_m < f_m$。$a_k$ 同时也在 S_{ij} 中，且具有比 a_m 更早的结束时间，这与 a_m 的选择相矛盾。所以推出 S_{im} 为空。

现证明第 1 部分：设 A_{ij} 为 S_{ij} 的最大兼容活动子集，且将 A_{ij} 中的活动按结束时间单调递增排序。又设 a_k 为 A_{ij} 的第一个活动。如果 $a_k=a_m$，则得到结论，因为我们已经证明 a_m 在 S_{ij} 的某个最大兼容活动子集中。如果 $a_k \neq a_m$，则构造子集 $A'_{ij}=A_{ij}-\{a_k\}\cup\{a_m\}$。因为在活动 A_{ij} 中，a_k 是第一个结束的活动，而 $f_m \leqslant f_k$，所以 A'_{ij} 中的活动是不相交的。注意到 A'_{ij} 具有与 A_{ij} 相同数目的活动，因此 A'_{ij} 是包含 a_m 的 S_{ij} 的最大兼容活动集合。 ■

为什么定理 16.1 如此有价值呢？回顾 15.3 节，我们知道，最优子结构会随着原问题最优解的子问题数目以及确定子问题时的选择数目变化。在动态规划方法中，最优解有两个子问题；在求解子问题 S_{ij} 时，有 $j-i-1$ 种选择。定理 16.1 将两者的数量大大减少：在最优解中只使用一个子问题(另一个子问题一定为空)；在解决子问题 S_{ij} 时，我们只需考虑一种选择：在 S_{ij} 中有着最早结束时间的那个活动。幸运的是，这个活动可以很容易找出来。

374 在减少子问题数目和选择数目的同时，定理 16.1 还带来另一个好处：可以自顶向下地解决

每一个子问题，而不是像动态规划中通常使用的那种自底向上的方式。为解决子问题 S_{ij}，选择 S_{ij} 中具有最早结束时间的活动 a_m，并将子问题 S_{mj} 的最优解中的活动集合加入到 S_{ij} 的解中。因为选择了 a_m，那么在 S_{ij} 的最优解中定会使用 S_{mj} 的一个解。在解决 S_{ij} 之前，我们不需要解决 S_{mj}。解决 S_{ij} 时，首先选择 S_{ij} 中最先结束的活动 a_m，然后解决 S_{mj}。

还要注意我们所解决的子问题中存在一种模式。原始问题为 $S=S_{0,n+1}$。设 a_{m_1} 是 $S_{0,n+1}$ 中具有最早结束时间的活动。(因为活动已按最早结束时间的单调递增顺序排序，$f_0=0$，所以必定有 $m_1=1$。)下一个子问题是 $S_{m_1,n+1}$。设选择 a_{m_2} 为 $S_{m_1,n+1}$ 中最早结束时间的活动(此时不需要 $m_2=2$)。则下一个子问题是 $S_{m_2,n+1}$。如此继续，对于号码为 m_i 的活动，其子问题的形式为 $S_{m_i,n+1}$。换句话说，每个子问题都包含了最近结束的活动，而活动的数目将随子问题的不同而变化。

我们所选择的活动中也存在着一种模式。因为总是选择 $S_{m_i,n+1}$ 中具有最早结束时间的活动，所有子问题的被选活动时间是严格递增的。而且，按活动结束时间的递增顺序，每个活动只需考虑一次。

在解决子问题时，选择的 a_m 是一个可被合法调度、具有最早结束时间的活动。从直觉上来看，这种活动选择方法是一种"贪婪的"选择方法，它给后面剩下的待调度任务留下了尽可能多的机会。也就是说，此处的贪心选择使得剩下的、未调度的时间最大化。

递归贪心算法

我们已经知道了应如何组织动态规划解，以及如何将其作为自顶向下的方法来看待，下面来介绍一种纯贪心的、自顶向下的算法。以下给出的过程 RECURSIVE-ACTIVITY-SELECTOR 是一个直接的递归解。在该过程中，用数组 s 和 f 表示活动的开始和结束时间，用下标 i 和 j 表示待处理的子问题 $S_{i,j}$。它的返回值为 $S_{i,j}$ 的最大兼容活动集合。假设根据公式(16.1)，n 个输入活动已按照结束时间的单调递增顺序排序。否则，也可以在 $O(n\lg n)$ 时间内将它们以此排序(打破任意关系)。初始调用是 RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, $n+1$)。 375

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, i, j)
1  m ← i+1
2  while m<j and s_m<f_i          ▷ Find the first activity in S_ij
3      do m ← m+1
4  if m<j
5     then return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, j)
6     else return ∅
```

图 16-1 示出了此算法的操作过程。对于一次给定的递归调用 RECURSIVE-ACTIVITY-SELECTOR(s, f, i, j)，while 循环中的第 2～3 行是寻找 S_{ij} 的第一个活动，该循环检查 a_{i+1}，a_{i+2}，…，a_{j-1}，直至找到与 a_i 兼容的第一个活动 a_m，此时 $s_m \geqslant f_i$。如果循环终止时找到了这样的活动，则过程从第 5 行返回 $\{a_m\}$ 和 S_{mj} 的最大子集的并集，其中 S_{mj} 是递归调用 RECURSIVE-ACTIVITY-SELECTOR(s, f, m, j)的返回值。或者，循环终止是因为 $m \geqslant j$，结束时间早于 a_j 的活动都与 a_i 不兼容。在这种情况下，$S_{ij}=\varnothing$，过程在第 6 行返回 $\varnothing$。

假设活动已根据结束时间排序，调用 RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, $n+1$)的运行时间为 $\Theta(n)$。对所有递归调用，每个活动仅仅在 while 循环中的第 2 行测试时被检查过一次。特别地，活动 a_k 是在 $i<k$ 情况下的最后一个调用中检查的。

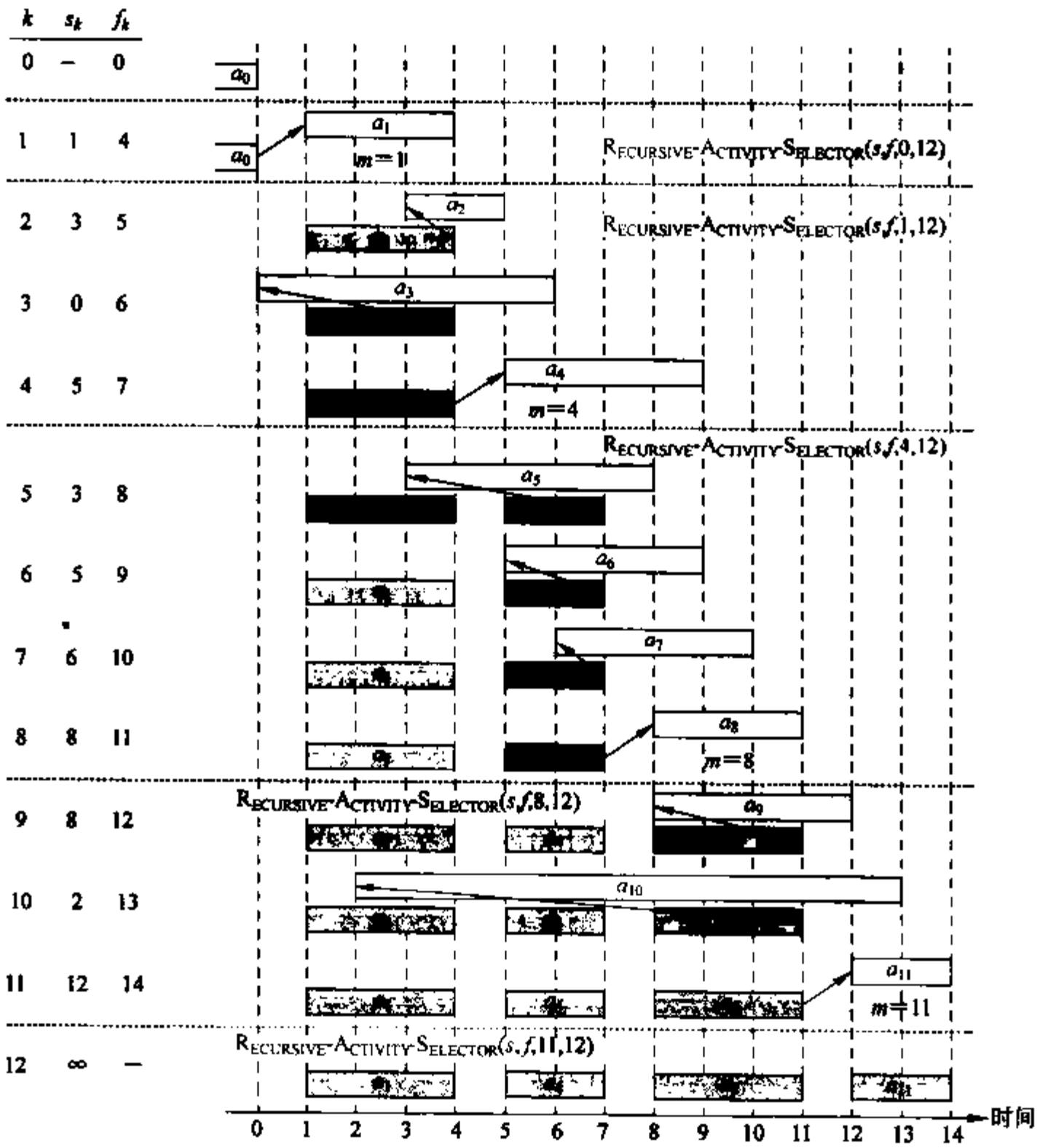


图 16-1　针对前面给出的 11 个活动的例子，RECURSIVE-ACTIVITY-SELECTOR 的操作示意图。出现在水平线之间的是每次递归要考虑的活动。虚构活动 a_0 在时间 0 结束，而在初始过程调用 RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 12)中，活动 a_1 被选中。在每次的递归调用中，已经被选中的活动以阴影标识，而以白色标识的活动表示正在被考虑的对象。如果活动的开始时间先于刚刚被加入活动的结束时间(它们之间的箭头指向左边)，则将被丢弃。否则(箭头指向上或向右)，此活动被选中。最后一个递归调用 RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 12)返回$\varnothing$。被选中活动的结果集为$\{a_1, a_4, a_8, a_{11}\}$

迭代贪心算法

递归算法可以很容易地转换为迭代算法。过程 RECURSIVE-ACTIVITY-SELECTOR 几乎就是个“尾递归”程序(见思考题 7-4)：它以对自己的递归调用的并操作结束。将尾递归过程转换为迭代形式，通常是一项比较直接的工作；事实上，某些语言的编译器会自动完成这一工作。虽然 RECURSIVE-ACTIVITY-SELECTOR 可以作用于任意子问题 S_{ij}，但我们只需考虑 $j=n+1$ 的子问题，亦即，包含最后结束活动的子问题。

过程 GREEDY-ACTIVITY-SELECTOR 是过程 RECURSIVE-ACTIVITY-SELECTOR 的迭代版本，它也假设输入活动已按结束时间的单调递增顺序排序。它将选中的活动放入集合 A，并在结束时将其返回。

376 ~ 377

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n ← length[s]
2  A ← {a_1}
3  i ← 1
4  for m ← 2 to n
5      do if s_m ≥ f_i
6           then A ← A ∪ {a_m}
7                i ← m
8  return A
```

此过程按如下方式工作：用变量 i 标记最近加入 A 的活动，对应着递归版本中的活动 a_i。既然活动按照结束时间的单调递增顺序被考虑，则 f_i 始终是 A 中活动的最大结束时间。即，

$$f_i = \max\{f_k : a_k \in A\} \tag{16.4}$$

第 2～3 行选择活动 a_1，初始化 A 只包含此活动，且初始化 i 标记此活动。第 4～7 行的 for 循环寻找 $S_{i,n+1}$ 中最早结束的活动。此循环按顺序考虑每一个活动 a_m，将与先前所选活动兼容的 a_m 加入到 A 中。此活动也是 $S_{i,n+1}$ 中最早结束的活动。如果 a_m 与 A 中目前的每个活动兼容，则满足公式(16.4)，检查(第 5 行)它的开始时间 s_m 是否不早于最近加入 A 的活动的结束时间 f_i。如果活动 a_m 是兼容的，则在第 6～7 行将活动 a_m 加入到 A 中，并置 i 为 m。调用 GREEDY-ACTIVITY-SELECTOR(s, f)返回的集合 A 也就是调用 RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, $n+1$)返回的集合。

与递归版本相同，GREEDY-ACTIVITY-SELECTOR 也是在 $\Theta(n)$ 时间内调度一个含 n 个活动的集合，其中，假设初始时活动已按照它们的结束时间排序。

练习

16.1-1　根据递归式(16.3)，给出活动选择问题的动态规划算法。要求算法能根据上面的定义计算出 $c[i, j]$ 的大小，而且产生出活动的最大子集 A。假设输入已根据公式(16.1)排序，请比较你的算法的运行时间与 GREEDY-ACTIVITY-SELECTOR 的运行时间。

16.1-2　假设不再总是选择第一个结束的活动，而是选择最后一个开始、且与之前选入活动兼容的活动。试说明这种方式如何成为一个贪心算法，并证明它能得到一个最优解。 378

16.1-3　假设要用很多个教室对一组活动进行调度。我们希望使用尽可能少的教室来调度所有的活动。请给出一个有效的贪心算法，来确定哪一个活动应使用哪一个教室。

（这个问题也被称为区间图着色(interval-graph coloring)问题。我们可作出一个区间图，其顶点为已知的活动，其边连接着不兼容的活动。为使任两个相邻结点的颜色均不相同，所需的最少颜色数对应于找出调度给定的所有活动所需的最少教室数。）

16.1-4　并不是任何用来解决活动选择问题的贪心算法都能给出兼容活动的最大集合。请给出一个例子，说明那种在与已选出的活动兼容的活动中选择生存期最短的方法是行不通的。对那种总是选择与余下的活动重叠最少的活动的做法，以及总是选择开始时间最早且与已选活动兼容的活动的做法，情况又怎么样呢？

16.2　贪心策略的基本内容

贪心算法是通过做一系列的选择来给出某一问题的最优解。对算法中的每一个决策点，做一个当时(看起来是)最佳的选择。这种启发式策略并不是总能产生出最优解，但正像我们在活动选择问题中看到的那样，它常常能给出最优解。这一节讨论贪心方法的某些一般性质。

在16.1节开发一个贪心算法时，我们所遵循的过程比一般情况要更复杂些。经过了如下步骤：

1)决定问题的最优子结构

2)设计出一个递归解

3)证明在递归的任一阶段，最优选择之一总是贪心选择。那么，做贪心选择总是安全的。

379　4)证明通过做贪心选择，所有子问题(除一个以外)都为空。

5)设计出一个实现贪心策略的递归算法。

6)将递归算法转换成迭代算法。

通过这些步骤，可以清楚地发现动态规划是贪心算法的基础。然而，实际在设计贪心算法时，我们经常简化以上步骤。通常直接做出贪心选择来构造子结构，以产生一个待优化解决的子问题。例如，在活动选择问题中，首先定义子问题 S_{ij}，其中 i 和 j 都是可变的。如果总是做出贪心选择，就可以将子问题的形式限制为 $S_{i,n+1}$。

或者，根据贪心选择来构造最优子结构。亦即，将第二个下标丢弃，并定义形如 $S_i=\{a_k \in S: f_i \leqslant s_k\}$ 的子问题。那么，可以证明，将一个贪心选择(S_i 中第一个结束的活动 a_m)与剩余相容活动集合 S_m 的最优解组合起来，就成为 S_i 的一个最优解。更一般地，可以根据如下的步骤来设计贪心算法：

1)将优化问题转化成这样的一个问题，即先做出选择，再解决剩下的一个子问题。

2)证明原问题总是有一个最优解是做贪心选择得到的，从而说明贪心选择的安全。

3)说明在做出贪心选择后，剩余的子问题具有这样的一个性质。即如果将子问题的最优解和我们所作的贪心选择联合起来，可以得出原问题的一个最优解。

在本章后面的小节中，我们将使用这个更加直接的过程。无论如何，在每一个贪心算法的下面，几乎总是会有一个更加复杂的动态规划解。

贪心算法是否能够解决一个特定的最优化问题呢？一般说来不是，但是贪心选择性质和最优子结构是两个关键的特点。如果我们能够证明问题具有这些性质，那么就可以设计出它的一个贪心算法。

贪心选择性质

第一个关键特点是贪心选择性质：一个全局最优解可以通过局部最优(贪心)选择来达到。换

380　句话说，当考虑做何选择时，我们只考虑对当前问题最佳的选择而不考虑子问题的结果。

这一点是贪心算法不同于动态规划之处。在动态规划中，每一步都要做出选择，但是这些选择依赖于子问题的解。因此，解动态规划问题一般是自底向上，从小子问题处理至大子问题。在贪心算法中，我们所做的总是当前看似最佳的选择，然后再解决选择之后所出现的子问题。贪心算法所做的当前选择可能要依赖于已经做出的所有选择，但不依赖于有待于做出的选择或子问题的解。因此，不像动态规划方法那样自底向上地解决子问题，贪心策略通常是自顶向下地做的，一个一个地做出贪心选择，不断地将给定的问题实例归约为更小的问题。

当然，我们必须证明在每一步所做的贪心选择最终能产生一个全局最优解，这也正是需要

技巧的所在。一般来说，如定理 16.1 中的情况一样，在证明中先考察一个全局最优解，然后证明可对该解加以修改，使其采用贪心选择，这个选择将原问题变为一个相似的、但更小的问题。

贪心选择性质在面对子问题做出选择时，通常能帮助我们提高效率。例如，在活动选择问题中，假设我们已将活动按结束时间的单调递增顺序排序，则每个活动只需检查一次。通常对数据进行处理或选用合适的数据结构(优先队列)，能够使贪心选择更加快速，因而产生出一个高效的算法。

最优子结构

对一个问题来说，如果它的一个最优解包含了其子问题的最优解，则称该问题具有最优子结构。这个性质是用来对动态规划以及贪心算法的可应用性进行评价的关键一点。作为最优子结构的一个例子，回顾一下在 16.1 节中，如果子问题 S_{ij} 最优解中包含活动 a_k，则它一定包含子问题 S_{ik} 和 S_{kj} 的最优解。基于这个最优子结构，我们推论如果知道哪个活动将被当成 a_k 使用，就可以选择 a_k 以及子问题 S_{ik} 和 S_{kj} 的最优解中的所有活动，来构造 S_{ij} 的一个最优解。基于对最优子结构的观察，我们能够写出描述一个最优解值的递归式(16.3)。

在贪心算法中使用最优子结构时，通常是用更直接的方式。如前所述，假设在原问题中做了一个贪心选择而得到了一个子问题。真正要做的是证明将此子问题的最优解与所做的贪心选择 381
合并后，的确可以得到原问题的一个最优解。这个方案意味着要对子问题采用归纳法，来证明每个步骤中所做的贪心选择最终会产生出一个最优解。

贪心法与动态规划

因为贪心法与动态规划都利用了最优子结构性质，故人们往往会在贪心解足以解决问题的场合下，给出一个动态规划解，或者在需要动态规划方法的场合下使用贪心方法。为说明这两种技术之间的细微区别，我们来考察一个经典最优化问题的两种变形。

0-1 背包问题是这样的：有一个贼在偷窃一家商店时发现有 n 件物品：第 i 件物品值 v_i 元，重 w_i 磅，此处 v_i 和 w_i 都是整数。他希望带走的东西越值钱越好，但他的背包中至多只能装下 W 磅的东西，W 为一整数。应该带走哪几样东西？(这个问题之所以称为 0-1 背包问题，是因为每件物品或被带走，或被留下；小偷不能只带走某个物品的一部分或带走两次以上的同一物品。)

在部分背包问题中，场景等与上面问题一样，但是窃贼可以带走物品的一部分，而不必做出 0-1 的二分选择。可以把 0-1 背包问题的一件物品想像成一个金锭，而部分背包问题中的一件物品则更像金粉。

两种背包问题都具有最优子结构性质。对 0-1 问题，考虑重量至多为 W 磅的最值钱的一包东西。如果我们从中去掉物品 j，余下的必须是窃贼从除 j 以外的 $n-1$ 件物品中，可以带走的重量至多为 $W-w_j$ 的最值钱的一包东西。对部分背包问题，考虑如果我们从最优货物中去掉某物品 j 的重量 w，则余下的货物必须是窃贼可以从 $n-1$ 件原有物品和物品 j 的 w_j-w 磅中可带走的、重量至多为 $W-w$ 的、最值钱的一包东西。

虽然这两个问题非常相似，但部分背包问题可用贪心策略来解决，而 0-1 背包问题却不行。为解决部分背包问题，先对每件物品计算其每磅的价值 v_i/w_i。按照一种贪心策略，窃贼开始时对具有最大每磅价值的物品尽量多拿一些。如果他拿完了该物品而仍可以取一些其他物品时，他就再取具有次大的每磅价值的物品，一直继续下去，直到不能再取为止。这样，通过按每磅价值来对所有物品排序，贪心算法就可以 $O(n\lg n)$ 时间运行。关于部分背包问题具有贪心选择性质的证明留作练习 16.2-1。 382

为搞清楚为什么这种贪心策略不适用于 0-1 背包问题，让我们来看一个该问题的实例，如

图 16-2a。总共有三件物品，背包可容纳 50 磅重的东西。物品 1 重 10 磅，值 60 元；物品 2 重 20 磅，值 100 元；物品 3 重 30 磅，值 120 元。这样，物品 1 是每磅 6 元，大于物品 2 的每磅 5 元和物品 3 的每磅 4 元。按照贪心策略的话就要取物品 1。然而，从图 16-2b 中的分析可以看出，最优解取的是物品 2 和 3，留下了 1。两种包含物品 1 的可能解都是次优的。

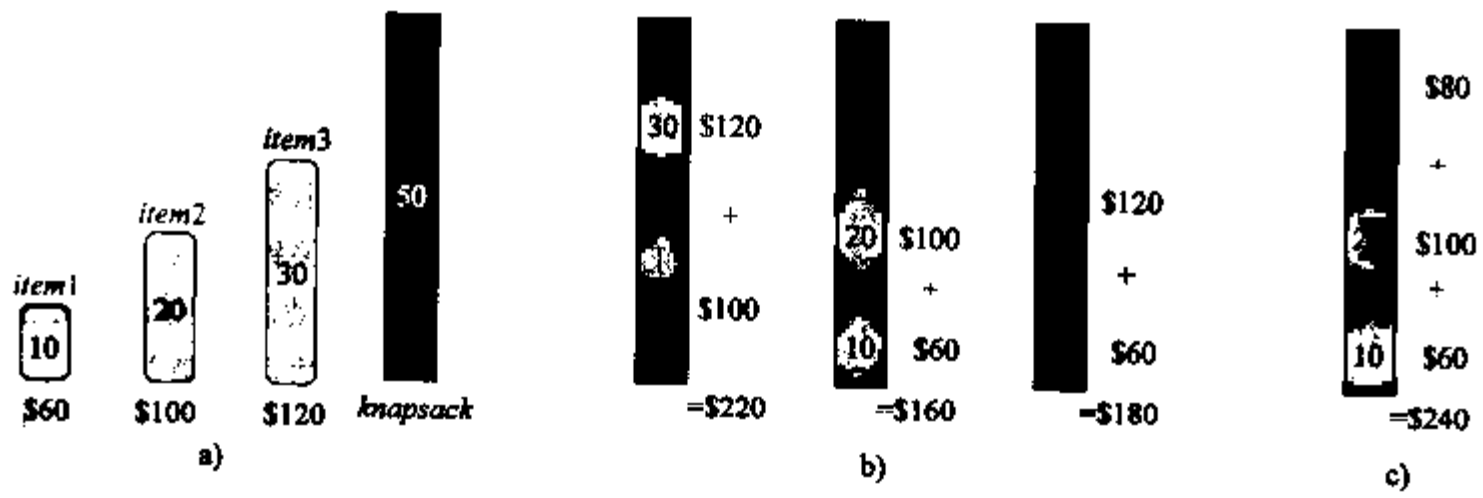


图 16-2　贪心策略对 0-1 背包问题不适用。a)窃贼必须选择三件物品的一个子集，且要求总重量不超过 50 磅。b)最优子集包括物品 2 和 3。任何包含物品 1 的解都是次优的，即使物品 1 具有最大每磅价值。c)对于部分背包问题，按每磅最大价值的顺序带走物品将得到一个最优解

对于部分背包问题，在按照贪心策略先取物品 1 以后，确实可产生一个最优解，如图 16-2c 所示。在 0-1 背包问题中不应取物品 1 的原因在于这样无法将背包填满，空余的空间就降低了他的货物的有效每磅价值。在 0-1 背包问题中，当我们在考虑是否要把一件物品加到背包中时，必须对把该物品加进去的子问题的解与不取该物品的子问题的解进行比较。由这种方式形成的问题导致了许多重叠子问题(这是动态规划的一个特点)，所以，我们可以用动态规划来解决 0-1 背包问题(见练习 16.2-2)。

383

练习

16.2-1　证明部分背包问题具有贪心选择性质。

16.2-2　请给出一个解决 0-1 背包问题的运行时间为 $O(n\,W)$ 的动态规划方法，其中，n 为物品的件数，W 为窃贼可放入他的背包中的物品中的最大重量。

16.2-3　假设在 0-1 背包问题中，按递增重量所排的物品的次序与按递减价值所排的次序一样。请给出此背包问题变体的最优解的有效算法，并说明其正确性。

16.2-4　Midas 教授从 Newark 开一辆车沿着 80 号洲际公路到 Reno。他车子的油箱在满的时候，可以存放够跑 n 英里的汽油，并且他的地图标出了在他的路途中加油站之间的距离。教授希望在路途中尽量地少加油。请给出一个高效的方法，来帮助教授决定哪一处加油站该停下来，并证明你的策略能够得到最优解。

16.2-5　请描述一个有效的算法，使之对给定的一实数轴上的点集 $\{x_1, x_2, \cdots, x_n\}$，能确定包含所有给定点的最小的单位闭区间集合。证明所给出的算法的正确性。

***16.2-6**　假设已知思考题 9-2 的解，说明如何在 $O(n)$ 时间内解决部分背包问题。

16.2-7　假设有两个集合 A 和 B，并且每个集合都包含 n 个正整数。你可以按自己的意愿将集合重新排序。在重新排序后，设 a_i 为集合 A 的第 i 个元素，b_i 为集合 B 的第 i 个元素。那么，你可以得到报酬 $\prod_{i=1}^{n} a_i^{b_i}$。请给出一个能最大化你的报酬的算法。证明它的正确性，并给出其运行时间。

384

16.3 赫夫曼编码

赫夫曼编码是一种被广泛应用而且非常有效的数据压缩技术，根据待压缩数据的特征，一般可压缩掉20%～90%。这里考虑的数据指的是字符串序列。赫夫曼贪心算法使用了一张字符出现频度表，根据它来构造一种将每个字符表示成二进制串的最优方式。

假设有一个包含100 000个字符的数据文件要压缩存储。各字符在该文件中的出现频度见图16-3。仅有6种不同字符出现过，字符a出现了45 000次。

	a	b	c	d	e	f
频度(千字)	45	13	12	16	9	5
固定长代码字	000	001	010	011	100	101
变长代码字	0	101	100	111	1101	1100

图16-3 一个字符编码问题。大小为100 000个字符的一个数据文件仅包含字符a～f，每个字符出现的频度如图中所示。如果对每个字符赋予一个三位的编码，则该文件可被编码为300 000位。如果利用图中的可变长度编码，该文件可被编码为224 000位

可以用很多种方式来表示这样一个文件。我们来考虑设计一种二进制字符编码(或简称为编码)的问题，其中每一个字符都由唯一的二进制串来表示。如果采用固定长度编码，则需要三位二进制数字来表示六个字符：a=000，b=001，…，f=101。这种方法需要300 000位来对整个原文件编码。能不能做得更好一些呢？

可变长编码要比固定长度编码好得多，其特点是对频度高的字符赋以短编码，而对频度低的字符则赋以较长一些的编码。图16-3示出了这样一种编码，其中一位串0表示a，四位串1100表示f。这种编码方式需要

$$(45\times1+13\times3+12\times3+16\times3+9\times4+5\times4)\times1000=224\,000\text{位}$$

来表示整个文件，即可压缩掉约25%。实际上，对这个文件来说，这已是一种最优字符编码了，这一点我们稍后将会看到。

前缀编码

我们这儿考虑的编码方案中，没有一个编码是另一个编码的前缀。这样的编码称为前缀编码[⊖]。可以证明(这儿略去)，由字符编码技术所获得的最优数据压缩总可用某种前缀编码来获得，因此将注意力集中到前缀编码上并不失一般性。

对任何一种二进制字符编码来说编码总是简单的，这只要将文件中表示每个字符的编码并置起来即可。例如，利用图16-3中的可变长度编码，把包含三个字符的文件abc编成0·101·100=0 101 100，其中"·"表示并置。

在前缀编码中解码也是很方便的。因为没有一个码是其他码的前缀，故被编码文件的开始处的编码是确定的。我们只要识别出第一个编码，将它翻译成原文字符，再对余下的编码文件重复这个解码过程即可。在我们的例子中，可将串001 011 101唯一地分析为0·0·101·1101，因而可解码为aabe。

解码过程需要有一种关于前缀编码的方便表示，使得初始编码可以很容易地被识别出来。有一种表示方法就是叶子为给定字符的二叉树。在这种树中，我们将一个字符的编码解释为从根至该字符的路径，其中0表示"转向左子结点"，1表示"转向右子结点"。图16-4给出了与我

⊖ 或许"无前缀编码"(prefix-free code)是个更好的名字，但是"前缀编码"(prefix code)是标准的书面语。

们的例子中两种编码对应的二叉树。注意它们并不是二叉查找树，因为各叶结点无需以排序次序出现，且内结点也不包含关键字。

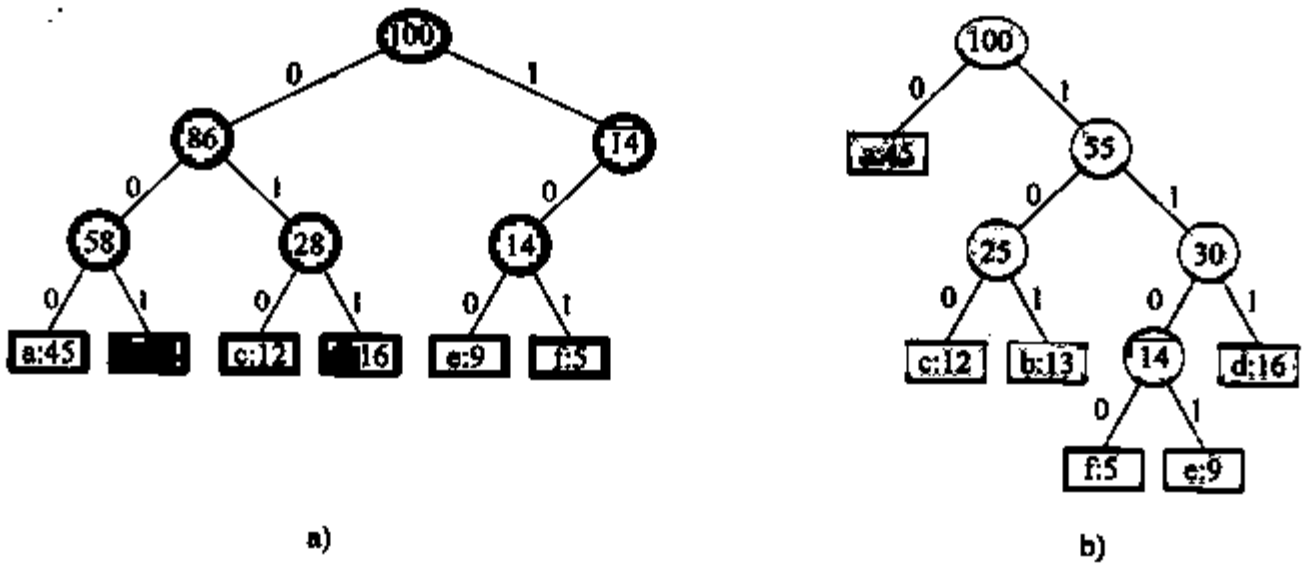


图 16-4　图 16-3 中两种编码策略所对应的二叉树。每个叶子结点被标以一个字符及其出现的频度。每个内结点标以其子树中所有叶子的频度总和。a)与固定长度编码 a＝000，…，f＝101对应的树。b)对应于最优前缀编码 a＝0，b＝101，…，f＝1100 的树

文件的一种最优编码总是由一棵满二叉树来表示的，树中每个非叶结点都有两个子结点(见练习 16.3-1)。在我们的例子中，固定长度编码不是最优编码，因为表示它的树(如图 16-4a 所示)不是满二叉树：有的编码开始于 10…，但没有一个开始于 11…。因为我们现在可以把注意力集中到满二叉树上，故可以说如果 C 是包含待编码字符的字母表且所有字符频度为正，则表示最优前缀编码的树中恰有 $|C|$ 片叶子，每一片表示字母表中的一个字母，另还有 $|C|-1$ 个内结点(见练习 B.5-3)。

给定对应一种前缀编码的二叉树 T，很容易计算出编码一个文件所需的位数。对字母表 C 中的每一个字符 c，设 $f(c)$表示 c 在文件中出现的频度，$d_T(c)$表示 c 的叶子在树中的深度。注意 $d_T(c)$也是字符 c 的编码的长度。这样，编码一个文件所需的位数就是

$$B(T) = \sum_{c \in C} f(c) d_T(c) \tag{16.5}$$

我们把它定义为树 T 的代价。

构造赫夫曼编码

赫夫曼设计了一个可用来构造一种称为赫夫曼编码的最优前缀码的贪心算法。与 16.2 节中的分析一样，该算法的正确性要依赖于贪心选择性质和最优子结构。我们不是先证明这些性质成立、再设计伪代码，而是首先给出伪代码。这样做有助于说明算法是如何做贪心选择的：

385
~
387

在下面的伪代码中，假设 C 是一个包含 n 个字符的集合，且每个字符 $c \in C$ 都是一个出现频度为 $f[c]$的对象。算法以自底向上的方式构造出最优编码所对应的树 T。它从 $|C|$ 个叶结点的集合开始，并依次执行 $|C|-1$ 次“合并”操作来构造最终的树。Q 是一个以 f 为关键字的最小优先级队列，用来识别出要合并的两个频度最低的对象。两个对象合并的结果是一个新对象，其频度为被合并的两个对象的频度之和。

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n−1
4       do allocate a new node z
```

```
5        left[z] ← x ← EXTRACT-MIN(Q)
6        right[z] ← y ← EXTRACT-MIN(Q)
7        f[z] ← f[x]+f[y]
8        INSERT(Q, z)
9  return EXTRACT-MIN(Q)        ▷ Return the root of the tree
```

对我们的例子而言，赫夫曼算法的执行过程如图 16-5 所示。因为字母表中共有 6 个字母，故初始队列的规模为 $n=6$，要构造编码树共需五步合并。最终的树就表示了最优前缀编码，其中某一字母的编码就是从根至该字母的路径上边标记的序列。

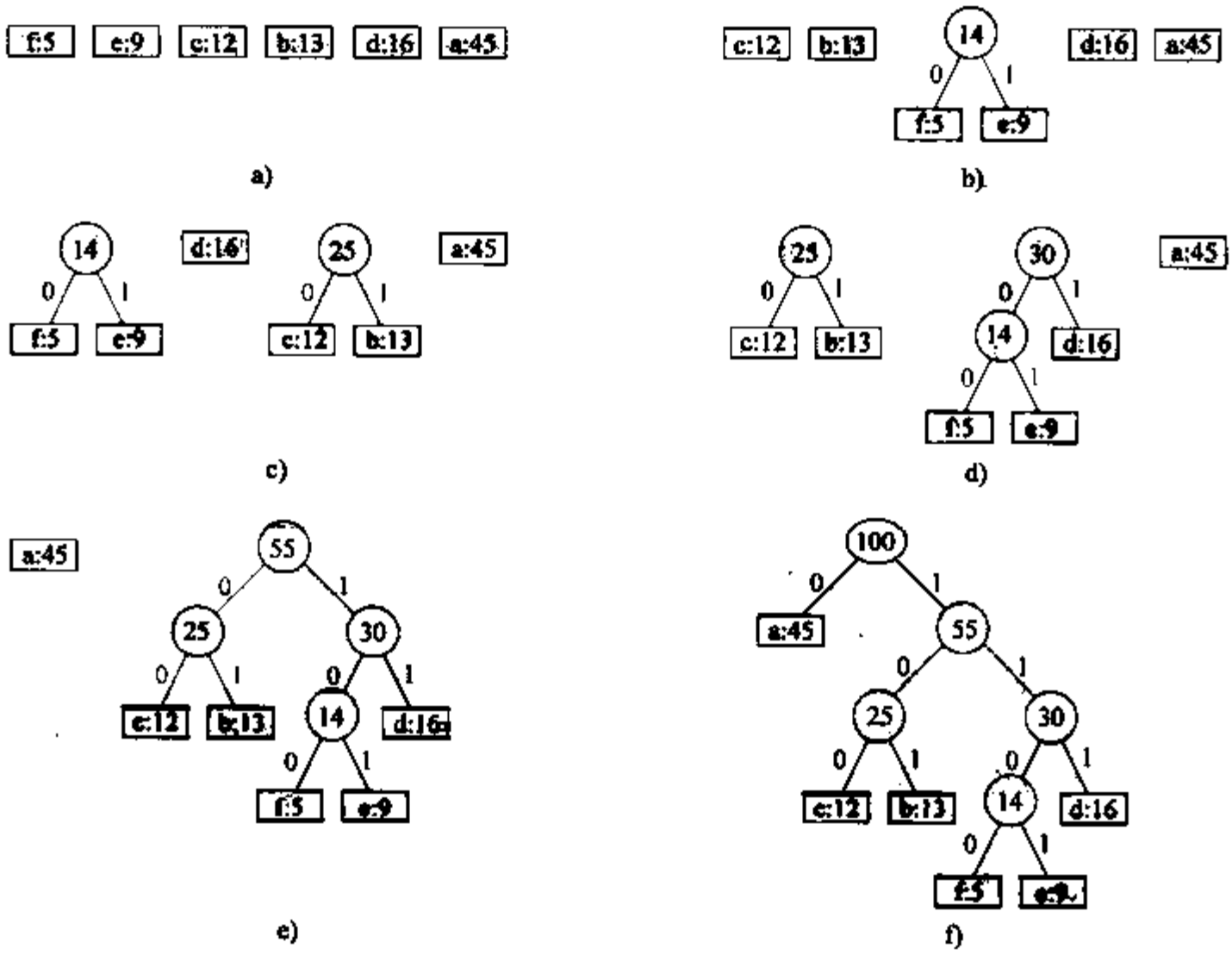


图 16-5　对图 16-3 所给定的频度，赫夫曼算法的执行过程。每一个图都示出了按频度的递增序排序队列的内容。在每一步，合并具有最低频度数的两棵树，叶子以包含字符及其频度的矩形示出。内结点以包含其各子女的频度之和的圆表示。对于连接一个内结点与其子女的边来说，如果它连向左子女则标以 0，连向右子女则标以 1。一个字母编码即为连接根至对应该字母的叶子的所有边上的标记序列。a)包含 $n=6$ 个结点的初始集合，每个结点代表一个字母。b)～e)为中间阶段。f)最终的树

第 2 行以 C 中的字符对最小优先级队列 Q 进行初始化。第 3～8 行的 for 循环反复取出队列中具有最低频度的两个结点 x 和 y，并用将它们合并后所得的 z 插入到队列中以替换它们。z 的频度在第 7 行中计算，为 x 和 y 的频度之和。新结点 z 以 x 为其左结点，以 y 为其右结点。(这个次序是任意的；将某一结点的左、右子结点交换所产生的是一种具有相同代价的不同编码。)在 $n-1$ 次合并后，在队列中剩下的唯一结点，即编码树的根，在第 9 行中返回。

在下面对赫夫曼算法的运行时间的分析中，我们假设 Q 是作为最小二叉堆(见第 6 章)实现的。对包含 n 个字符的集合 C，第 2 行中对 Q 的初始化可用 6.3 节中的 BUILD-MIN-HEAP 过程在 $O(n)$ 时间内完成。第 3～8 行中的 for 循环执行了 $n-1$ 次，又因每一次堆操作需要 $O(\lg n)$ 时

间，故整个循环需要 $O(n\lg n)$时间。这样，作用于 n 个字符集合的 HUFFMAN 的总的运行时间为 $O(n\lg n)$。

赫夫曼算法的正确性

为了证明贪心算法 HUFFMAN 的正确性，我们就要证明确定最优前缀编码的问题具有贪心选择和最优子结构性质。下面这个引理证明了这个问题具有贪心选择性质。

引理 16.2　设 C 为一字母表，其中每个字符 $c\in C$ 具有频度 $f[c]$。设 x 和 y 为 C 中具有最低频度的两个字符，则存在 C 的一种最优前缀编码，其中 x 和 y 的编码长度相同但最后一位不同。

388 ~ 389

证明：证明的主要思想是使树 T 表示任一种最优前缀编码，然后对它进行修改，使之表示另一种最优前缀编码，使得字符 x 和 y 在新树中成为具有最大深度的兄弟叶结点。如果我们能做到这一点，则它们的编码就具有相同长度，而仅仅最后一位不同。

设 a 和 b 为树 T 中具有最大深度的兄弟叶结点。不失一般性，假设 $f[a]\leqslant f[b]$且 $f[x]\leqslant f[y]$。因为 $f[x]$和 $f[y]$是两个最低的频度，而 $f[a]$和 $f[b]$是两个任意的频度，故有 $f[x]\leqslant f[a]$且 $f[y]\leqslant f[b]$。如图 16-6 所示，交换 a 和 x 在树 T 中的位置产生树 T'，然后交换 b 和 y 在树 T'中的位置产生树 T''。根据公式(16.5)，树 T 和 T'之间的代价上相差为：

$$\begin{aligned}B(T)-B(T') &= \sum_{c\in C} f(c)d_T(c)-\sum_{c\in C} f(c)d_{T'}(c)\\ &= f[x]d_T(x)+f[a]d_T(a)-f[x]d_{T'}(x)-f[a]d_{T'}(a)\\ &= f[x]d_T(x)+f[a]d_T(a)-f[x]d_T(a)-f[a]d_T(x)\\ &= (f[a]-f[x])(d_T(a)-d_T(x))\geqslant 0\end{aligned}$$

因为 $f[a]-f[x]$和 $d_T(a)-d_T(x)$都是非负的。更具体一点，$f[a]-f[x]$是非负的，因为 x 是具有最小频度的叶结点；$d_T(a)-d_T(x)$也是非负的，因为 a 为 T 中具有最大深度的叶结点。类似地，交换 y 和 b 并不增加代价，所以 $B(T')-B(T'')$也是非负的。因此，$B(T'')\leqslant B(T)$；又因为 T 是最优的，故 $B(T)\leqslant B(T'')$，这就意味着 $B(T'')=B(T)$。这样，T''就是一棵最优树，其中 x 和 y 为具有最大深度的兄弟叶结点，从而证明了上述引理。 ■

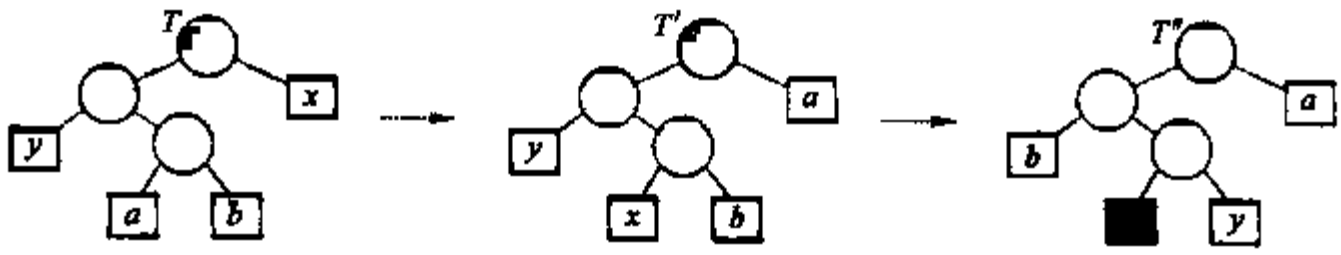


图 16-6　对引理 16.2 的证明中关键步骤的说明。在最优树 T 中，叶子 a 和 b 是最深的叶结点中的两个，并且是兄弟。叶子 x 和 y 为赫夫曼算法首先合并的两个叶子，它们出现于 T 中任意位置上。叶子 a 和 x 交换得到树 T'。然后交换叶子 b 和 y 得到树 T''。因为每次交换并不增加代价，所以所得的树 T''也是个最优树

由引理 16.2 可知，不失一般性，通过合并来构造一棵最优树的过程可以从合并两个频度最低的字符的贪心选择开始。这为什么是个贪心选择呢？我们可以认为一次合并的代价就是被合并的两个字符的频度之和。练习 16.3-3 证明了通过合并构造出的树的总代价就是各次合并的代价之和。在每一步所有可能的合并中，HUFFMAN 选择一个代价最小的合并。

390

下面的引理证明了构造最优前缀编码的问题具有最优子结构性质。

引理 16.3　设 C 为一给定字母表，其中每个字母 $c\in C$ 都定义有频度 $f[c]$。设 x 和 y 是 C 中具有最低频度的两个字母。并设 C' 为字母表移去 x 和 y，再加上(新)字符 z 后的字母表，亦即

$C'=C-\{x, y\}\cup\{z\}$；如 C 一样为 C' 定义 f，其中 $f[z]=f[x]+f[y]$。设 T' 为表示字母表 C' 上最优前缀编码的任意一棵树。那么，将 T' 中的叶子结点 z 替换成具有 x 和 y 孩子的内部结点所得到的树 T，表示字母表 C 上的一个最优前缀编码。

证明：我们先来证明通过考虑公式(16.5)中的各个成分代价来将树 T 的代价 $B(T)$ 以树 T' 的代价 $B(T')$ 来表示。对每一个 $c\in C-\{x, y\}$，我们有 $d_T(c)=d_{T'}(c)$，故 $f[c]d_T(c)=f[c]\ d_{T'}(c)$。又因为 $d_T(x)=d_T(y)=d_{T'}(z)+1$，我们有

$$\begin{aligned} f[x]d_T(x)+f[y]d_T(y) &= (f[x]+f[y])(d_{T'}(z)+1) \\ &= f[z]d_{T'}(z)+(f[x]+f[y]) \end{aligned}$$

根据此式可得：

$$B(T)=B(T')+f[x]+f[y]$$

或等价地，有：

$$B(T')=B(T)-f[x]-f[y]$$

现在用反证法来证明此引理。假设 T 不表示 C 的最优前缀编码，那么存在一棵树 T''，有 $B(T'')<B(T)$。不失一般性地(根据引理 16.2)，T'' 有 x 和 y 两个兄弟结点。设 T''' 是由 T'' 中将 x 和 y 的父亲结点替换为叶子结点 z 而得，其中频度 $f[z]=f[x]+f[y]$。那么，

$$B(T''')=B(T'')-f[x]-f[y]<B(T)-f[x]-f[y]=B(T')$$

推出一个矛盾，因为我们假设 T' 表示 C' 上的最优前缀编码，那么 T 必定表示字母表 C 上的最优前缀编码。(证毕) ■

定理 16.4　过程 HUFFMAN 产生一种最优前缀编码。

证明：由引理 16.2 和引理 16.3 直接可得。(证毕) ■ 391

练习

16.3-1　证明：一棵不满的二叉树不可能与一种最优前缀编码对应。

16.3-2　对下面的频度集合(基于前 8 个斐波那契数)，其最优的赫夫曼编码是什么？

a：1　b：1　c：2　d：3　e：5　f：8　g：13　h：21

能将答案推广到前 n 个斐波那契数吗？

16.3-3　证明：对应于某种编码的树的总代价也能通过计算所有内结点的两子结点的频度之和而得到。

16.3-4　证明：对一字母表的字符按其频度的单调递减顺序排序，则存在一个编码长度单调递增的最优编码。

16.3-5　假设有一个字母表 $C=\{0, 1, \cdots, n-1\}$ 上的最优前缀编码，我们想用尽可能少的位来传输。证明：C 上的任意一个最优前缀编码都可由 $2n-1+n\lceil\lg n\rceil$ 个位的序列来表示。(提示：用 $2n-1$ 位来说明树的结构，通过树的遍历来发现)。

16.3-6　将赫夫曼编码推广至三进制编码(亦即，用符号 0，1，2 来编码)，并证明它能产生最优编码。

16.3-7　假设某一数据文件包含一系列的 8 位字符，且所有 256 个字符的频度都差不多：最大字符频度不到最小字符频度的两倍。证明：这种情况下赫夫曼编码的效率与普通的 8 位固定长度编码就差不多了。

16.3-8　证明：没有一种数据压缩方案能对包含随机选择的 8 位字符的文件作任何压缩。(提示：将文件数与可能的编码文件数进行比较。)　392

*16.4　贪心法的理论基础

关于贪心算法有着一种很漂亮的理论，这一节里我们要大致介绍一下。这种理论在确定贪心方法何时能产生最优解时非常有用，它用到了一种称为“拟阵”的组合结构。虽然这种理论没有覆盖贪心方法所适用的所有情况(例如，它没有包括16.1节中的活动选择问题或16.3节中的赫夫曼编码问题)，但它确实覆盖了许多具有实际意义的情况。另外，这个理论发展很快，并正在覆盖更多的应用(可以参看本章最后的“本章注记”)。

拟阵

一个拟阵是满足下列条件的一个序对 $M=(S, \ell)$：

1) S 是一个有穷非空集合。

2) ℓ 是 S 的一个非空子集族，称为 S 的独立子集，使得如果 $B\in\ell$ 且 $A\subseteq B$，那么 $A\in\ell$。我们说 ℓ 是遗传的，如果它满足这个性质。注意空集 $\varnothing$ 必为 ℓ 的一个成员。

3) 如果 $A\in\ell$，$B\in\ell$，且 $|A|<|B|$，则有某个元素 $x\in B-A$ 使得 $A\cup\{x\}\in\ell$。我们称 M 满足交换性质。

“拟阵”这个词是由 Hassler Whitney 最早开始用的。他曾研究矩阵拟阵，其中 S 的元素是某个给定矩阵的各个行；如果某些行在通常意义下是线性无关的，则它们是独立的。很容易证明这种结构定义了一个拟阵(见练习16.4-2)。

作为另一个例子，来看一看图的拟阵 $M_G=(S_G, \ell_G)$，它是根据一个给定的无向图 $G=(V, E)$ 来定义的：

- 集合 S_G 定义为 E，即 G 的边集。
- 如果 A 是 E 的子集，则 $A\in\ell_G$ 当且仅当 A 是无回路的。亦即，一组边 A 是独立的当且仅当子图 $G_A=(V, A)$ 构成了一个森林。

图的拟阵 M_G 与最小生成树问题有很密切的联系，我们将在第23章中详细介绍。

定理16.5　如果 $G=(V, E)$ 是一个无向图，则 $M_G=(S_G, \ell_G)$ 是个拟阵。

证明：显然，$S_G=E$ 是个有穷集合；并且，ℓ_G 是遗传的，因为森林的子集还是森林。换句话说，从无环的一组边中去掉一些边并不会产生出回路来。

393

现在，要证明 M_G 满足交换性质。假设 $G_A=(V, A)$ 和 $G_B=(V, B)$ 是 G 的森林，且 $|B|>|A|$。A 和 B 都由一组无回路边构成，且 B 中包含比 A 中更多的边。

据定理B.2可知，具有 k 条边的森林恰好包含 $|V|-k$ 棵树。(可以以另一种方式来证明这个定理，即从 $|V|$ 棵树开始，每棵树只含一个结点，且没有边。那么，每向森林中增加一条边就使树的数目减少1。)这样，森林 G_A 包含 $|V|-|A|$ 棵树，而森林 G_B 包含 $|V|-|B|$ 棵树。

因为森林 G_B 中含有的树比森林 G_A 少，故森林 G_B 比包含了某棵树 T 其结点属于森林 G_A 的两颗不同的树。此外，因为 T 是连通的，它必然包含边 (u, v)，使得结点 u 和 v 在森林 G_A 的不同树中。正因为边 (u, v) 连接了森林 G_A 中两棵不同树的结点，故将边 (u, v) 加到森林 G_A 中后不会产生回路。所以，M_G 满足交换性质，从而证明了 M_G 是一个拟阵。■

给定一个拟阵 $M=(S, \ell)$，我们称元素 $x\notin A$ 为 $A\in\ell$ 的一个扩张，如果能在保持独立性的同时将 x 加到 A 中去；亦即，如果 $A\cup\{x\}\in\ell$，x 是 A 的一个扩张。作为一个例子，我们来考虑一个图的拟阵 M_G。如果 A 是一个独立的边集，则边 e 是 A 的一个扩张当且仅当 e 不在 A 中，且将 x 加到 A 中后不产生回路。

如果 A 是拟阵 M 的一个独立子集，且它没有任何扩张，则称 A 是最大的。也就是说，如果 A 不被 M 中比它更大的独立子集所包含，则它是最大的。下面一个性质常常是很有用的。

定理 16.6　某一拟阵中所有最大的独立子集的大小都是相同的。

证明：假设 A 是 M 的一个最大独立子集，且存在 M 的另一个更大的最大独立子集 B 这对矛盾。那么由交换性质可知，A 可以扩张到一个更大的独立集合 $A \cup \{x\}$，$x \in B - A$。这就与 A 是最大的假设相矛盾。(证毕) ■

为说明这个定理，我们来考虑某连通无向图 G 的一个拟阵 M_G。M_G 的每个最大独立子集都应是一棵自由树，且恰好有连接 G 中所有结点的 $|V|-1$ 条边。这样的一棵树称为 G 的生成树。

如果有一个加权函数 w，它对每一个元素 $x \in S$ 都赋予一个正的权值 $w(x)$，则我们称一个拟阵 $M=(S, \ell)$ 是加权的。权函数 w 对于 S 的子集有和式：

$$w(A) = \sum_{x \in A} w(x)$$

394

其中，任意 $A \subseteq S$。例如，如果以 $w(e)$ 表示一个图的拟阵 M_G 中某条边 e 的长度，则 $w(A)$ 即为边集 A 中所有边长的总长度。

关于加权拟阵的贪心算法

适宜用贪心方法来获得最优解的许多问题，都可以归结为在加权拟阵中，找出一个具有最大权值的独立子集的问题。亦即，给定一个加权拟阵 $M=(S, \ell)$，我们希望找出一个独立的且具有最大可能权值 $w(A)$ 的子集 $A \in \ell$，称之为拟阵的最优子集。因为任一元素 $x \in S$ 的权值 $w(x)$ 都是正的，故一个最优子集总是一个最大独立子集——它总是使 A 尽可能地大。

例如，在最小生成树问题中，我们已经知道一个连通无向图 $G=(V, E)$ 和一个长度函数 w，$w(e)$ 是边 e 的(正的)长度。(这儿用长度一词来指示图中原先边的权值，而用"权"这个术语指示相联系的拟阵中的权值。)我们所要做的是找出一个包含着连接所有结点的，且具有最小总长度的边的子集。为将这个问题看作为一个拟阵中找出最优子集的问题，考虑具有权函数 w' 的加权拟阵 M_G，其中 $w'(e)=w_0-w(e)$，且 w_0 大于各边的最大长度。在这个加权拟阵中，所有的权值都是正的，且一个最优子集就是一棵在原图中具有最小总长度的生成树。更具体地说，每一个最大独立子集 A 对应于一棵生成树，又因为

$$w'(A) = (|V|-1)w_0 - w(A)$$

对任意最大独立子集 A 成立，故使得 $w'(A)$ 最大的独立子集必使 $w(A)$ 最小。这样一个可以在某一拟阵中找出最优子集 A 的算法都可用来解决最小生成树问题。

第 23 章给出了解决最小生成树问题的算法，这儿我们所给出的是适用于任何加权拟阵的贪心算法。该算法的输入是一个加权拟阵 $M=(S, \ell)$ 和一个相关的正的权函数 w，返回的是一个最优子集 A。在伪代码中，用 $S[M]$ 和 $\ell[M]$ 表示 M 的组成，用 w 表示权函数。该算法是贪心的，因为它按权值的单调减顺序来依次考虑每个元素 $x \in S$，如果 $A \cup \{x\}$ 是独立的话，它就立即把该元素加到 A 中。

395

```
GREEDY(M, w)
1  A ← ∅
2  sort S[M] into monotonically decreasing order by weight w
3  for each x ∈ S[M], taken in monotonically decreasing order by weight w(x)
4       do if A ∪ {x} ∈ ℓ[M]
5            then A ← A ∪ {x}
6  return A
```

算法按权值的单调减顺序依次考虑 S 的各个元素。如果正在被考虑的元素 x 加到 A 中后能保持 A 的独立性，则将其加入 A 中。否则，丢弃 x。因为根据拟阵的定义空集是独立的，又因为只有在 $A\cup\{x\}$ 是独立的情况下才将 x 加入到 A 中，根据归纳法，子集 A 总是独立的。所以，GREEDY 返回的是一个独立子集 A。过一会儿我们还将看到，A 是一个具有最大可能权值的子集，故 A 是个最优子集。

GREEDY 的运行时间是很容易分析的。设 n 表示 $|S|$，GREEDY 的排序阶段的时间为 $O(n\lg n)$。第 4 行对 S 中的每个元素执行一次，共 n 次。在第 4 行的每次执行中要检查 $A\cup\{x\}$ 是否独立。如果每次检查的时间为 $O(f(n))$，则整个算法的运行时间为 $O(n\lg n+nf(n))$。

下面再来证明 GREEDY 返回的是一个最优子集。

引理 16.7(拟阵具有贪心选择性质)　假设 $M=(S,\ell)$ 是一个具有权函数 w 的加权拟阵，且 S 被按权值的单调减顺序排序。设 x 为 S 的第一个使 $\{x\}$ 独立的元素(如果这样的 x 存在的话)。如果 x 存在，则存在 S 的一个包含 x 的最优子集 A。

证明：如果这样的 x 不存在，则唯一的独立集合为空集，证明结束。否则，设 B 为任意非空的最优子集。假设 $x\notin B$；否则，让 $A=B$，证明结束。

B 中没有一个元素的权值大于 $w(x)$。为说明这点，注意到 $y\in B$ 意味着 $\{y\}$ 是独立的，因为 $B\in\ell$ 且 ℓ 是遗传的。我们选择的 x 就保证了对任意 $y\in B$，$w(x)\geqslant w(y)$。

按照如下的步骤来构造集合 A。开始时 $A=\{x\}$。根据所选择的 x，A 是独立的。利用交换性质，重复地在 B 中找新的可以加至 A 中而同时保持 A 的独立性的元素，直至 $|A|=|B|$。这样，$A=B-\{y\}\cup\{x\}$，其中 $y\in B$，故有

396
$$w(A)=w(B)-w(y)+w(x)\geqslant w(B)$$

因为 B 是最优的，A 也为最优的；又因为 $x\in A$，引理得证。■

下面我们要证明，一个元素如果开始时不被选中，以后也不会被选中。

引理 16.8　设 $M=(S,\ell)$ 为任意一个拟阵。如果 x 是 S 的任意元素，是 S 的独立子集 A 的一个扩张，那么 x 也是 $\varnothing$ 的一个扩张。

证明：因为 x 是 A 的一个扩张，我们有 $A\cup\{x\}$ 是独立的。又因为 ℓ 是遗传的，那么 $\{x\}$ 必定独立。所以，x 是 $\varnothing$ 的一个扩张。■

推论 16.9　设 $M=(S,\ell)$ 为任意一个拟阵。如果集合 S 中元素 x 不是 $\varnothing$ 的扩张，那么 x 也不会是 S 的任意独立子集 A 的一个扩张。

证明：这个推论就是引理 16.8 的简单地对换。■

推论 16.9 说明任何元素如果不能被立即用到，则以后也决不会被用到。所以，在开始时，对 S 中的那些不是 $\varnothing$ 的扩张的元素不予选择决不是漏掉了它们，而是因为它们以后也不会被用到。

引理 16.10(拟阵具有最优子结构性质)　设 x 为 S 中被作用于加权拟阵 $M=(S,\ell)$ 的 GREEDY 第一个选择了的元素。找一个包含 x 的具有最大权值的独立子集的问题，可以归约为找出加权拟阵 $M'=(S',\ell')$ 的一个具有最大权值的独立子集的问题，此处

$$S'=\{y\in S:\{x,y\}\in\ell\}\quad \ell'=\{B\subseteq S-\{x\}:B\cup\{x\}\in\ell\}$$

其中，M' 权函数为(受限于 S' 的)M 的权函数(称 M' 为 M 的由 x 引起的收缩)。

证明：如果 A 为任意包含 x 的 M 的具有最大权值的独立子集，那么 $A'=A-\{x\}$ 就是 M' 的一个独立子集。反之，由 M' 的任意独立子集 A' 可得 M 的一个独立子集 $A=A'\cup\{x\}$。因为在两种情况下，都有 $w(A)=w(A')+w(x)$，所以由 M 中包含的 x 的一个最大权值解可得 M' 中的一个最大权值解，反之亦然。■

397

定理 16.11(拟阵上贪心算法的正确性)　如果 $M=(S, \ell)$为一具有权函数 w 的加权拟阵，则调用 GREEDY(M, w)返回一个最优子集。

证明：根据推论 16.9，在开始时被略去的那些不是∅的扩张的元素可以不予考虑，因为它们不会被用到。一旦选择了第一个元素 x，由引理 16.7 可知在 GREEDY 中将 x 加到 A 中是正确的，因为存在着一个包含 x 的最优子集。最后，引理 16.10 隐含着余下的问题就是一个在 M 的由 x 引起的收缩 M'中寻找一个最优子集问题。在过程 GREEDY 将 A 置为$\{x\}$后，余下的各步骤都可解释为是在拟阵 $M'=(S', \ell')$中进行的，因为 B 在 M'中是独立的，当且仅当 $B \cup \{x\}$在 M 中是独立的，其中 B 为任意属于ℓ'的集合。这样，GREEDY 的后续操作就会找出 M'中的一个具有最大权值的独立子集，而 GREEDY 的全部操作就可找出 M 的一个具有最大权值的独立子集。

■

练习

16.4-1　证明：(S, ℓ_k)为一个拟阵，其中 S 为任一有限集合，ℓ_k 为 S 的所有大小至多为 k 的子集构成的集合，$k \leqslant |S|$。

***16.4-2**　给定一个 $m \times n$ 的某域(如实数)上的矩阵 T，证明(S, ℓ)是个拟阵，其中 S 为 T 的所有列构成的集合，且 $A \in \ell$ 当且仅当 A 中各列是线性无关的。

***16.4-3**　证明：如果(S, ℓ)是一个拟阵，则(S, ℓ')也是一个拟阵。此处

$\ell' = \{A' : S - A'$包含某个最大的 $A \in \ell\}$，

亦即，(S, ℓ')的最大独立子集是(S, ℓ)的最大独立子集的补集。

***16.4-4**　设 S 为一个有穷集合，且 S_1，S_2，…，S_k 为将 S 分成非空的不相交子集的一个划分。结构(S, ℓ)由条件 $\ell = \{A : |A \cap S_i| \leqslant 1, i=1, 2, \cdots, k\}$来定义。请证明$(S, \ell)$为一个拟阵。也就是说，对于包含了划分的每个块中至多一个成员的集合 A，由所有的集合 A 构成的集合决定了一个拟阵的独立集合。 398

16.4-5　说明在最优解为具有最小权值的最大独立子集的加权拟阵问题中，如何改造其权值函数，使之成为一个标准的加权拟阵问题。另请证明所做的改造是正确的。

*16.5　一个任务调度问题

有一个可用拟阵来解决的有趣问题，即在单个处理器上对若干个单位时间任务进行最优调度，其中每个任务都有一个截止期限和超时惩罚。这个问题看起来很复杂，但用贪心算法来解决的话则惊人地简单。

单位时间任务是一个作业(如在计算机上运行的一个程序)，它恰好需要一个单位的时间来运行。给定一个有穷单位时间任务的集合 S，对 S 的一个调度即为 S 的一个排列，它规定了各任务的执行顺序。该调度中的第一个任务开始于时间 0，结束于时间 1；第二个任务开始于时间 1，结束于时间 2，等等。

单处理器上具有期限和惩罚的单位时间任务调度问题的输入如下：

- 包含 n 个单位时间任务的集合 $S=\{a_1, a_2, \cdots, a_n\}$
- n 个取整数值的期限 $d_1, d_2, \cdots, d_n$，每个 d_i 满足 $1 \leqslant d_i \leqslant n$ 且任务 a_i 要求在 d_i 之前完成。
- n 个非负的权(或惩罚)$w_1, w_2, \cdots, w_n$。如果任务 a_i 没在时间 d_i 之前结束，则导致惩罚 w_i；而如果任务在期限之前完成，则没有惩罚。

我们要做的是找出 S 的一个调度，使之最小化因误期而导致的总惩罚。

考虑一个给定的调度。我们说一个任务在调度中迟了，如果它在规定的期限之后完成；否则，这个任务在该调度中是早的。一个任意的调度总可以安排成早任务优先的形式，其中早的任务总是在迟的任务之前。为了搞清楚这一点，请注意如果某个早的任务 a_i 跟在某个迟的任务 a_j 之后，就可以交换 a_i 和 a_j 的位置，并不影响 a_i 是早的 a_j 是迟的状态。

类似地，任意一个调度总可以安排成一个规范形式，其中早的任务先于迟的任务，且按期限的单调递增顺序对早任务进行调度。为了做到这一点，将调度安排成早任务优先形式。然后，只
399 要在该调度中有两个分别完成于时间 k 和 $k+1$ 的早任务 a_i 和 a_j 使得 $d_j<d_i$，就交换 a_i 和 a_j 的位置。因为在交换前任务 j 是早的，$k+1\leqslant d_j$。所以，$k+1<d_i$，则 a_i 在交换之后仍然是早的。任务 a_j 被移到了调度中的更前位置，故它在交换后也仍然是早的。

如此，寻找最优调度问题就成为找一个由最优调度中早的任务构成的集合 A 的问题。一旦 A 被确定后，就可按期限的单调递增顺序列出 A 中的所有元素，从而给出实际的调度，然后按任意顺序列出迟的任务(即 $S-A$)，就可产生出最优调度的一个规范次序。

称一个任务的集合 A 是独立的，如果存在关于 A 中任务的一个调度，使得没有一个任务是迟的。显然，某一调度中早任务的集合就构成了一个独立的任务集。设 ℓ 表示有所有独立的任务集合构成的集合。

考虑确定一给定任务集 A 是否是独立的问题。对 $t=0, 1, 2, \cdots, n$，设 $N_t(A)$ 表示 A 中的期限为 t 或更早的任务的个数。注意，对任意集合 A，$N_0(A)=0$。

引理 16.12　对任意的任务集合 A，下列命题是等价的：

1)集合 A 是独立的；

2)对 $t=0, 1, 2, \cdots, n$，有 $N_t(A)\leqslant t$。

3)如果对 A 中任务按期限的单调递增的顺序进行调度，则没有一个任务是迟的。

证明：显然，如果对某个 t 有 $N_t(A)>t$，则无法为集合 A 做出一个没有迟的任务的调度，这是因为有多于 t 个任务要在时间 t 之前完成。因而，1)蕴含了 2)。如果 2)成立，则 3)也必然成立：当按期限的单调递增的顺序进行调度时，不可能出现任何问题，因为 2)隐含着第 i 大期限至多在时间 i 处。最后，3)蕴含 1)显而易见。■

现在利用引理 16.12 的性质 2)，我们可以很容易地判断一个给定的任务集合是否是独立的(见练习 16.5-2)。

最小化迟任务的惩罚之和的问题与最大化早任务的惩罚之和的问题是一样的。下面的定理保证了我们可以用贪心法来找出具有最大总惩罚的独立任务集 A。

定理 16.13　如果 S 是一个带期限的单位时间任务的集合，且 ℓ 为所有独立的任务集构成的
400 集合，则对应的系统 (S, ℓ) 是一个拟阵。

证明：一个独立的任务集的每个子集肯定是独立的。为证明交换性质，假设 B 和 A 为独立的任务集，且 $|B|>|A|$。设 k 为使 $N_t(B)\leqslant N_t(A)$ 成立的最大的 t(这样的 t 值一定存在，因为 $N_0(A)=N_0(B)=0$)。因为 $N_n(B)=|B|$，$N_n(A)=|A|$，但是 $|B|>|A|$，故必有 $k<n$，且对 $k+1\leqslant j\leqslant n$ 中的所有 j，有 $N_j(B)>N_j(A)$。所以，B 中包含了比 A 中更多的具有期限 $k+1$ 的任务。设 a_i 为 $B-A$ 中具有期限 $k+1$ 的一个任务。另设 $A'=A\cup\{a_i\}$。

现在利用引理 16.12 的性质 2 来证明 A' 必为独立的。对 $0\leqslant t\leqslant k$，有 $N_t(A')=N_t(A)\leqslant t$，因为 A 是独立的。对 $k<t\leqslant n$，有 $N_t(A')\leqslant N_t(B)\leqslant t$，因为 B 是独立的。所以，A' 是独立的，这样就完成了 (S, ℓ) 是拟阵的证明。(证毕)■

根据定理 16.11，我们可用一个贪心算法来找出一个具有最大权值的独立的任务集 A。然后，可以设计出一个以 A 中的任务作为其早任务的最优调度。这种方法对在单一处理器上调度

具有期限和惩罚的单位时间任务来说是很有效的。采用了 GREEDY 后，这个算法的运行时间为 $O(n^2)$，因为算法中 $O(n)$ 次独立性检查的每一次都要花 $O(n)$ 的时间(见练习 16.5-2)。思考题 16-4 给出了一种更快速的实现。

图 16-7 给出了这种在单一处理器上对具有任务期限和惩罚的单位时间任务调度问题的一个例子。在这个例子中，贪心算法选择了任务 a_1，a_2，a_3 和 a_4，放弃了 a_5 和 a_6，最后接受任务 a_7。最终的最优调度为

$$(a_2, a_4, a_1, a_3, a_7, a_5, a_6)$$

总的惩罚数为 $w_5 + w_6 = 50$。

	Task						
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

图 16-7　单处理器上带期限和惩罚的单位时间任务调度问题的一个实例

练习

16.5-1　解图 16-7 中调度问题的实例，但要将每个惩罚 w_i 替换成 $80 - w_i$。 401

16.5-2　说明如何利用引理 16.12 的性质 2 在 $O(|A|)$ 时间内，确定一个给定的任务集 A 是否是独立的。

思考题

16-1　找换硬币

考虑用最少的硬币数来找 n 分钱的问题，假设每个硬币的值都是整数。

a)请给出一个贪心算法，使得所换硬币包括一角的、五分的、二角五分的和一分的。证明所给出的算法能产生最优解。

b)假设可换的硬币的单位是 c 的幂，也就是 c^0，c^1，…，c^k，其中整数 $c>1$，$k \geqslant 1$。证明贪心算法总可以产生一个最优解。

c)请给出一组使贪心算法不能产生最优解的硬币单位集合。所给集合应当包括一分，以便保证对任意 n 值都有解。

d)请给出一种 $O(nk)$ 时间的算法，它能够对任意 k 种不同单位的硬币集合进行找换，假设其中一种硬币单位是一分的。

16-2　最小化平均结束时间的调度

假设给定一任务集合 $S=\{a_1, a_2, \cdots, a_n\}$，其中 a_i 一旦开始，需要 p_i 的单位处理时间来完成。现在只有一台计算机来运行这些任务，而且计算机上每次只能运行一个任务。设 c_i 为任务 a_i 的结束时间，也就是任务 a_i 处理完成的时间。目标是使平均结束时间最小，即最小化 $(1/n)\sum_{i=1}^{n} c_i$。例如，假设有两个任务 a_1 和 a_2，且 $p_1=3$，$p_2=5$。考虑调度时让 a_2 先运行，a_1 跟随其后。那么 $c_2=5$，$c_1=8$，则平均结束时间为 $(5+8)/2=6.5$。

a)请给出一个调度任务的算法，使它能够最小化平均结束时间。每个任务的运行必须是非抢占式的，也就是说，任务 a_i 一旦开始，那么必须连续运行 p_i 单位时间。证明你的算法最小化了平均结束时间，并给出算法的运行时间。 402

b)假设现在并不是所有任务都可立刻获得。亦即，每个任务在被处理之前都有一个松弛时间 r_i。又假设允许抢占，则一个任务后来可被挂起和重新启动。例如，一个处理时间 $p_i=6$ 的任务 a_i 在时间 1 开始运行，在时间 4 被抢占。它在时间 10 恢复，在时间 11 又被抢占，最后在时间 13 恢复，在时间 15 完成。任务 a_i 总共运行了 6 单位的时间，但其运行时间被分成了三段。我们说 a_i 的结束时间为 15。请给出一个任务调度算法，它能够在这种新场景下最小化平均结束时间。证明你的算法的确能够最小化平均结束时间，并给出你的算法的运行时间。

16-3　无环子图

a)设 $G=(V, E)$ 为一个无向图。利用拟阵的定义，证明 (E, ℓ) 为一个拟阵，其中 $A\in\ell$ 当且仅当 A 是 E 的一个无环子集。

b)一个无向图 $G=(V, E)$ 的关联矩阵就是一个 $|V|\times|E|$ 的矩阵 M；如果边 e 与结点 v 关联，则 $M_{ve}=1$，否则 $M_{ve}=0$。论证：M 的若干列的集合在模 2 整数域上是线性无关的，当且仅当对应的边集是无环的。然后利用练习 16.4-2 来提供 a)部分 (E, ℓ) 为拟阵的另一种证明。

c)假设在一个无向图 $G=(V, E)$ 中每条边都有一个非负的权值 $w(e)$ 与之相联系。请给出一个可以找出 E 中具有最大总权值的无环子集的有效算法。

d)设 $G(V, E)$ 为任一有向图，且 (E, ℓ) 定义为 $A\in\ell$，当且仅当 A 不包含任何有向环。请给出一个有向图 G 的例子，使得与之相关的系统 (E, ℓ) 不是拟阵。另请说明拟阵定义中的那个条件不成立。

e)有向图 $G=(V, E)$ 的关联矩阵是一个 $|V|\times|E|$ 的矩阵 M，如果边 e 离开结点 v，则 $M_{ve}=-1$；如果边 e 进入结点 v，则 $M_{ve}=1$；否则 $M_{ve}=0$。论证：如果 M 的部分列是线性无关的，则相应的边集中不包含有向环。

f)由练习 16.4-2 可知，任一矩阵 M 的由线性无关的列集构成的集合是一个拟阵。请解释为什么 d)和 e)的结论不是矛盾的。在一个无环边集概念和与之相联系的关联矩阵中
403 线性无关的列集概念之间，为什么不存在一个完美的对应？

16-4　调度问题的变形

考虑下面给出的用于解决 16.5 节中带期限和惩罚的单位时间任务调度问题的一个算法。设所有 n 个时间槽开始时都是空的，其中时间槽 i 是终止于时间 i 的单位长度时间槽。我们按惩罚的单调递减顺序来考虑任务。在考虑任务 a_j 时，如果有一个恰处于或前于 a_j 的期限 d_j 的时间槽是空的，则将任务 a_j 赋予最近的这样的槽，并填入。如果不存在这样的槽，则将任务 a_j 赋予一个还未被占的、最近的槽。

a)论证：这个算法总能给出一个最优解。

b)利用 21.3 节中的快速不相交集合森林来有效地实现这个算法。假设已按惩罚数的单调递减顺序对输入任务集合进行了排序。请分析所给出的实现的运行时间。

本章注记

在 Lawler[196]及 Papadimitriou 和 Steiglitz[237]中，可以找到更多的关于贪心算法和拟阵的资料。

在 1971 年的 Edmonds[85]的文章中，第一次出现了组合优化问题的贪心算法，拟阵的理论

要追溯到 1935 年 Whitney[314]的文章。

我们的关于活动选择问题的贪心算法的正确性证明是基于 Gavril[112]提出的方法，而任务调度问题在 Lawler[196]，Horowitz 和 Sahni[157]以及 Brassard 和 Bratley[47]中都有研究。

赫夫曼编码是在 1952[162]年被发明出来的；Lelewer 和 Hirschberg[200]总结了在 1987 年已知的数据压缩技术。

拟阵理论的一个延伸——广义拟阵理论，最早是由 Korte 和 Lovász[189，190，191，192]所开创，他们大大地推广了这里的理论。 404

第17章　平摊分析

在平摊分析(amortized analysis)中，执行一系列数据结构操作所需要的时间是通过对执行的所有操作求平均而得出的。平摊分析可以用来证明在一系列操作中，通过对所有操作求平均之后，即使其中单一的操作具有较大的代价，平均代价还是很小的。平摊分析与平均情况分析的不同之处在于它不牵涉到概率；平摊分析保证在最坏情况下，每个操作具有平均性能。

本章的前三节介绍在平摊分析中三种最常用的技术。17.1节首先介绍聚集分析，它用于确定一个n个操作序列的总代价的上界$T(n)$。每个操作的平均代价可表示为$T(n)/n$。我们把平均代价当作每个操作的平摊代价，因此所有的操作有相同的平摊代价。

17.2节介绍记账方法，在其中要确定每个操作的平摊代价。当有一种以上的操作时，每种操作都可有一个不同的平摊代价。这种方法对操作序列中的某些操作先"多记账"，将多记的部分作为对数据结构中的特定对象上"预付的存款"存起来。在该序列中稍后将用到这些存款，以补偿那些对它们记的"账"少于其实际代价的操作。

17.3节讨论"势能方法"，它与记账方法的相似之处在于要确定每个操作的平摊代价，而且可能先对操作多记账以补偿以后的不足记账。这种方法将存款作为数据结构整体的"势能"来维护，而不是将存款与数据结构中的单个对象联系起来。

我们将用两个例子来说明这三种方法。一是带有额外操作MULTIPOP的栈，它一次弹出几个对象。另一个是一个二进制计数器，它利用一个操作INCREMENT从0开始计数。

405　在阅读本章时，读者应该记住在平摊分析中所记的"账"只是为了分析之用。它们不必要也不应该出现在代码中。例如，在采用记账方法时，如果将一项存款值赋予某一对象x，在代码中就没有必要对某个属性$credit[x]$赋相应的值。

通过作平摊分析，可以获得对某种特定数据结构的认识，这种认识有助于优化设计。例如，在17.4节中，我们将用势能方法分析一个动态扩充和收缩的表。

17.1　聚集分析

在聚集分析(aggregate analysis)中，要证明对所有的n，由n个操作所构成的序列的总时间在最坏情况下为$T(n)$。因此，在最坏情况下，每个操作的平均代价(或称平摊代价，amortized cost)为$T(n)/n$。请注意这个平摊代价对每个操作都是成立的，即使当序列中存在几种类型的操作时也是一样的。本章中要研究的另外两种方法，即记账方法和势能方法，可能对不同类型的操作赋予不同的平摊代价。

栈操作

在关于聚集分析的第一个例子中，我们要分析增加了一个新操作后的栈。10.1节中介绍了两种基本的栈操作，每种操作的时间代价都是$O(1)$：

PUSH(S, x)：将对象x压入栈S。

POP(S)：弹出S的栈顶返回弹出的对象。

因为这两个操作的运行时间都为$O(1)$，故可以把每个操作的代价都视为1。因此，含n个PUSH和POP操作的序列的总代价为n，而这n个操作的实际运行时间就是$\Theta(n)$。

现在我们增加一个栈操作MULTIPOP(S, k)，它的作用是弹出栈S的k个栈顶对象，或者，

当栈包含少于 k 个对象时，弹出整个栈中的数据对象。在下面的伪代码中，如果当前栈中没有对象，则操作 STACK-EMPTY 返回 TRUE，否则返回 FALSE。

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S)and k≠0
2      do POP(S)
3         k ← k  1
```

图 17-1 给出 MULTIPOP 的一个例子。

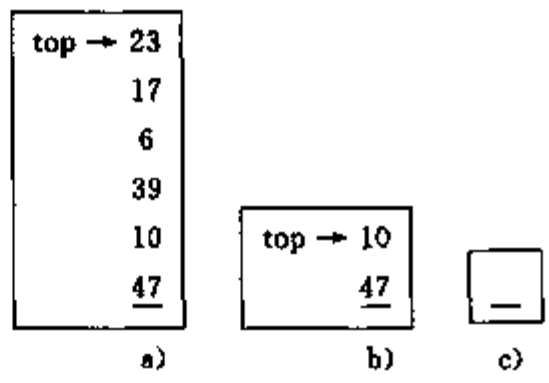


图 17-1　MULTIPOP 在栈 S 上的操作过程，初始情况见 a)。最上面的 4 个对象由 MULTIPOP(S, 4)弹出，其结果如 b)所示。下一个操作是 MULTIPOP(S, 7)，它将栈弹出如 c)中所示，因为余下的对象已不足 7 个

当 MULTIPOP(S, s)对一个包含 s 个对象的栈操作时，运行时间是多少？实际的运行时间与实际执行的 POP 操作数成线性关系，因而只要按 PUSH 和 POP 具有抽象代价 1 来分析 MULTIPOP 就足够了。while 循环迭代的次数是从栈中弹出的对象个数 $\min(s, k)$。对循环的每次迭代，在第 2 行中都要调用一次 POP。因此，MULTIPOP 的总代价是 $\min(s, k)$，而实际运行时间则为这个代价的一个线性函数。

现在来分析一个由 n 个 PUSH，POP 和 MULTIPOP 操作构成的序列，其作用于一个初始为空的栈。序列中一次 MULTIPOP 操作的最坏情况代价为 $O(n)$，因为栈的大小至多为 n。因此，任意栈操作的最坏情况时间就是 $O(n)$，因此 n 个操作的序列的代价是 $O(n^2)$，因为可能会有 $O(n)$个 MULTIPOP 操作，每个的代价都是 $O(n)$。虽然这一分析是正确的，但通过单独地考虑每个操作的最坏情况代价而得到的 $O(n^2)$结论却是不够紧确的。

利用聚集分析，我们可以获得一个考虑到 n 个操作的整个序列的更好的上界。事实上，虽然一次 MULTIPOP 操作的代价可能较高，但当作用于一个初始为空的栈上时，任意一个包含 n 个 PUSH，POP 和 MULTIPOP 操作的序列其代价至多为 $O(n)$。为什么会是这样呢？一个对象在每次被压入栈后，至多被弹出一次。所以，在一个非空栈上，调用 POP 的次数(包括在 MULTIPOP 的调用)至多等于 PUSH 操作的次数，即至多为 n。对任意的 n 值，包含 n 个 PUSH，POP 和 MULTIPOP 操作的序列的总时间为 $O(n)$。每个操作的平均代价为 $O(n)/n=O(1)$。在聚集分析中，我们把每个操作的平摊代价指派为平均代价。所以在这个例子中，三个栈操作的平摊代价都是 $O(1)$。

我们再一次强调，虽然已经说明一个栈操作的平均代价和运行时间是 $O(1)$，但没有用到概率推理。实际上，我们给出的是一列 n 个操作的最坏情况界 $O(n)$。用 n 除这个总代价可得每个操作的平均代价，亦即平摊代价。

二进制计数器递增 1

作为聚集分析的另一个例子，考虑实现一个由 0 开始向上计数的 k 位二进制计数器的问题。我们使用一个位数组 $A[0..k-1]$作为计数器，其中 $length[A]=k$。存储在计数器中的一个二进制数 x 的最低位在 $A[0]$中，最高位在 $A[k-1]$中，故 $x=\sum_{i=0}^{k-1} A[i]\cdot 2^i$。开始时，$x=0$，因此对 $i=0,1,\cdots,k-1$ 均有 $A[i]=0$。为将计数器中的值加 1(模 2^k)，我们使用下面的过程。

```
INCREMENT(A)
```

```
1  i ← 0
2  while i<length[A] and A[i]=1
3       do A[i] ← 0
4          i ← i+1
5  if i<length[A]
6    then A[i] ← 1
```

图 17-2 演示了一个二进制计数器从初值 0 至终值 16 的 16 次增值过程。在第 2～4 行中每次 while 循环的开始，我们希望在位置 i 处加 1。如果 $A[i]=1$，则加 1 后就将位置 i 处的数位变为 0，并产生一个进位 1，它在循环的下一次迭代中加到位置 $i+1$ 上。否则，循环结束。然后，如果 $i<k$，我们知道 $A[i]=0$，因此将 1 加到位置 i 后，使 0 变为 1，这在第 6 行中完成。每次 INCREMENT 操作的代价都与被改变值的位数成线性关系。

计数器值	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	总代价
0	0	0	0	0	0	0	0		0
1	0	0	0	0	0	0			1
2	0	0	0	0	0	0	1		3
3	0	0	0	0	0				4
4	0	0	0	0	0	1	0		7
5	0	0	0	0	0	1			8
6	0	0	0	0	0	1	1		10
7	0	0	0	0					11
8	0	0	0	0	1	0	0		15
9	0	0	0	0	1	0			16
10	0	0	0	0	1	0	1		18
11	0	0	0	0	1				19
12	0	0	0	0	1	1	0		22
13	0	0	0	0	1	1			23
14	0	0	0	0	1	1	1		25
15	0	0	0						26
16	0	0	0	1	0	0	0		31

图 17-2　一个 8 位的二进制计数器，经过 16 次的 INCREMENT 操作，其值从 0 增长到 16。发生翻转而取得下一个值的位加了阴影。右边给出了位翻转所需的运行代价。注意总代价始终不超过 INCREMENT 操作总次数的两倍

如同栈的例子，大致的分析只能得到一个正确但不紧确的界。在最坏情况下，INCREMENT 的每次执行要花 $\Theta(k)$ 时间，此时数组 A 中包含的全为 1。因此，在最坏情况下，作用于一个初始为零的计数器上的 n 个 INCREMENT 操作的时间就为 $O(nk)$。

注意到在每次调用 INCREMENT 时，并不是所有的位都翻转，我们可以分析得更紧确一些，来得到 n 次 INCREMENT 操作的序列的最坏情况代价为 $O(n)$。如图 17-2 所示，在每次调用 INCREMENT 时，$A[0]$确实都要发生翻转。下一个高位 $A[1]$每隔一次翻转；当作用于初始为零的计数器上时，n 次 INCREMENT 操作会导致 $A[1]$翻转 $\lfloor n/2 \rfloor$ 次。类似地，位 $A[2]$每四次翻转一次，或在 n 次 INCREMENT 操作中翻转 $\lfloor n/4 \rfloor$ 次。一般地，对 $i=0, 1, \cdots, \lfloor \lg n \rfloor$，在一个 n 次 INCREMENT 操作的序列中，位 $A[i]$要翻转 $\lfloor n/2^i \rfloor$ 次，这个序列作用于初始为零的计数器上。对于 $i>\lfloor \lg n \rfloor$，位 $A[i]$始终保持不变。因此根据公式(A.6)，在序列中发生的位翻转的总次数为

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

所以，在最坏情况下，作用于一个初始为零的计数器上的 n 次 INCREMENT 操作的时间为

$O(n)$。每次操作的平均代价(即每次操作的平摊代价)是 $O(n)/n = O(1)$。

练习

17.1-1　如果一组栈操作中包括了一次 MULTIPUSH 操作，它一次把 k 个元素压入栈内，那么栈操作的平摊代价的界 $O(1)$是否还能保持？

408
~
409

17.1-2　证明：在 k 位计数器的例子中，如果包含一个 DECREMENT 操作，n 个操作可能花费 $\Theta(nk)$时间。

17.1-3　对某个数据结构执行 n 个操作的一个序列。如果 i 为 2 的整数幂，则第 i 个操作的代价为 i，否则为 1。请利用聚集分析来确定每次操作的平摊代价。

17.2　记账方法

在平摊分析的记账方法中，我们对不同的操作赋予不同的费用，某些操作的费用比它们的实际代价或多或少。我们对一个操作的收费的数量称为平摊代价。当一个操作的平摊代价超过了它的实际代价时，两者的差值就被当作存款(credit)，并赋予数据结构中的一些特定对象。存款可以在以后用于补偿那些其平摊代价低于其实际代价的操作。这样，我们就可以将一个操作的平摊代价看作两部分：其实际代价与存款(或被储蓄或被用完)。这种方法与聚集分析有着很大的不同，对后者而言，所有操作都具有相同的平摊代价。

在选择操作的平摊代价时必须很小心。如果希望通过对平摊代价的分析来说明每次操作的最坏情况平均代价较小，则操作序列的总的平摊代价就必须是该序列的总的实际代价的一个上界。而且，像在聚集分析中一样，这种关系必须对所有的操作序列都成立。如果把第 i 个操作的实际代价用 c_i 表示，第 i 个操作的平摊代价用 $\hat{c}_i$ 表示，对 n 个操作的所有序列我们需要

$$\sum_{i=1}^{n}\hat{c}_i \geqslant \sum_{i=1}^{n}c_i \tag{17.1}$$

数据结构中储存的总存款等于总的平摊代价和总的实际代价之差，即 $\sum_{i=1}^{n}\hat{c}_i - \sum_{i=1}^{n}c_i$。根据不等式(17.1)可知，与数据结构对应的总存款必须始终是非负的。如果总存款允许为负值的话(开始时对某些操作的收费过低，而许诺稍后将补偿)，则这时总的平摊代价就会低于总的实际代价；对到该时刻为止的操作序列来说，总的平摊代价就不会是总的实际代价的一个上界。所以我们必须注意，数据结构中的总存款不能是负的。

栈操作

为了说明平摊分析的记账方法，再回过头来看看栈的例子。各栈操作的实际代价为　410

PUSH	1，
POP	1，
MULTIPOP	$\min(k, s)$，

其中 k 为 MULTIPOP 的一个参数，s 为调用该操作时栈的大小。现在对它们赋予以下的平摊代价：

PUSH	2，
POP	0，
MULTIPOP	0，

注意 MULTIPOP 的平摊代价是常数(0)，而它的实际代价却是个变量。此处所有三个平摊代价

都是都是 $O(1)$，但一般来说，从渐近的意义上来看，所考虑的各种操作的平摊代价是会发生变化的。

现在来说明通过对平摊代价收费，可以支付任何的栈操作序列。假设用 1 元钱来表示代价的单位。开始时栈是空的。回顾 10.1 节中的类比，栈数据结构与餐厅中一堆送放的盘子类似。当把一个盘子压入栈时，我们用 1 元来支付该压入动作的实际代价，还有(2 元收费中的)1 元的存款，我们将这 1 元存款放在盘子的顶上。在任何一个时间点上，堆中每个盘子的上面都有 1 元钱的存款。

盘子上面所存的钱是用来预付将盘子从栈中弹出所需代价的。当执行了一个 POP 操作时，对该操作不收取任何费用，只要用盘中所放的存款来支付其实际代价即可。为弹出一个盘子，就拿掉该盘子上的 1 元存款，并用它来支付弹出操作的实际代价。这样，在对 PUSH 操作多收一点费用后，就无需对 POP 操作收取任何费用。

进一步地，我们也无需对 MULTIPOP 操作收取任何费用。为弹出第一个盘子，可取出其中的 1 元存款，并用它支付一次 POP 操作的实际代价。为弹出第二个盘子，再取出该盘上的 1 元存款来支付第二次 POP 操作等。因此，我们预先收取了足够多的费用来支付 MULTIPOP 操作。换言之，因为栈上的每个盘子上面都有 1 元钱，而且栈中总有非负个数的盘子，这就保证了存款的总量总是非负的。这样，对任意的包含 n 次 PUSH、POP 和 MULTIPOP 操作的序列，总的平摊代价就是其总的实际代价的一个上界。又因为总的平摊代价为 $O(n)$，故总的实际代价也为 $O(n)$。

二进制计数器递增 1

为了进一步解释记账方法，我们再来分析作用于一个初始为零的二进制计数器上的 411 INCREMENT 操作。前面已经看到，这个操作的运行时间与发生翻转的位数是成正比的，而位数在本例中将被用作代价。我们还是用 1 元钱来表示单位代价(此例中即为某一位的翻转)。

对于平摊分析，规定对把某一位设为 1 的操作收取 2 元的平摊费用。当某数位被设置后，用(2 元收费中)1 元来支付置位操作的实际代价，而将另 1 元钱存在该位上作为存款，以在后面把该位翻转为 0 时使用。在任何时间点上，计数器中每个 1 上都有 1 元存款。这样在将某位复位成 0 时，不用支付任何费用；只要取出该位上的 1 元存款来支付复位。

现在就可以来确定 INCREMENT 的平摊代价了。在 while 循环中，复位操作的代价是由有关位上的余款来支付的。在 INCREMENT 的第 6 行中至多有一位被设置，所以一次 INCREMENT 操作的平摊代价至多为 2 元。又因为计数器中为 1 的位数始终是非负的，故存款额总是非负的。因此对 n 次 INCREMENT 操作，总的平摊代价为 $O(n)$，这就给出了总的实际代价的一个界。

练习

17.2-1 对一个大小始终不超过 k 的栈上执行一系列的栈操作。在每 k 个操作后，复制整个栈的内容以留作备份。证明：在对各种栈操作赋予合适的平摊代价后，n 个栈操作(包括复制栈的操作)的代价为 $O(n)$。

17.2-2 利用记账方法的分析重做练习 17.1-3。

17.2-3 假设我们希望不仅能使一个计数器增值，也能使之复位至零(即，使其中所有的位都为 0)。请说明如何将一个计数器实现为一个位数组，使得对一个初始为零的计数器，任一个包含 n 个 INCREMENT 和 RESET 操作的序列的时间为 $O(n)$。(提示：保持一个指针

指向高位 1。）

17.3　势能方法

在平摊分析中，势能方法(potential method)不是将已预付的工作作为存储在数据结构特定对象中的存款来表示，而是表示成一种“势能”或“势”，它在需要时可以释放出来，以支付后面的操作。势是与整个数据结构而不是其中的个别对象发生联系的。 412

势能方法的工作过程是这样的：开始时，先对一个初始数据结构 D_0 执行 n 个操作。对每个 $i=1, 2, \cdots, n$，设 c_i 为第 i 个操作的实际代价，D_i 为对数据结构 D_{i-1} 作用第 i 个操作的结果。势函数 Φ 将每个数据结构 D_i 映射为一个实数 $\Phi(D_i)$，即与数据结构 D_i 相联系的势。第 i 个操作的平摊代价 $\hat{c}_i$ 根据势函数 Φ 定义为

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{17.2}$$

每个操作的平摊代价为其实际代价加上由于该操作所增加的势。根据等式(17.2)，n 个操作的总的平摊代价为：

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \tag{17.3}$$

第二步推导由等式(A.9)所得，因为 $\Phi(D_i)$ 是重叠的。

如果我们能定义一个势函数 Φ 使得 $\Phi(D_n) \geqslant \Phi(D_0)$，则总的平摊代价 $\sum_{i=1}^{n} \hat{c}_i$ 就是总的实际代价 $\sum_{i=1}^{n} c_i$ 的一个上界。实际上，我们并不总是知道要执行多少个操作。所以，如果要求对所有的 i 有 $\Phi(D_i) \geqslant \Phi(D_0)$，则就像在记账方法中一样，我们保证预先支付。通常为了方便起见，定义 $\Phi(D_0)$ 为 0，然后证明对所有的 i，有 $\Phi(D_i) \geqslant 0$。（练习 17.3-1 中给出了一种处理 $\Phi(D_0) \neq 0$ 时的各种情况的简单方法。）

从直觉上看，如果第 i 个操作的势差 $\Phi(D_i) - \Phi(D_{i-1})$ 是正的，则平摊代价 $\hat{c}_i$ 表示对第 i 个操作多收了费，同时数据结构的势也随之增加了。如果势差是负值，则平摊代价就表示对第 i 个操作的不足收费，这是通过减少势来支付该操作的实际代价。

由等式(17.2)和等式(17.3)定义的平摊代价依赖于所选择的势函数 Φ。不同的势函数可能会产生不同的平摊代价，但它们都是实际代价的上界。在选择一个势函数时常要做一些权衡；可选用的最佳势函数的选择要取决于所需的时间界。

栈操作

为了说明势能方法，我们再一次回到栈操作 PUSH、POP 和 MULTIPOP 的实例。定义栈上的势函数 Φ 为栈中对象的个数。开始时要处理的是空栈 D_0，且 $\Phi(D_0)=0$。因为栈中的对象数 413
始终是非负的，故在第 i 个操作之后，栈 D_i 就具有非负的势，且有

$$\Phi(D_i) \geqslant 0 = \Phi(D_0)$$

以 Φ 表示的 n 个操作的平摊代价的总和就表示了实际代价的一个上界。

现在我们来计算各个栈操作的平摊代价。如果作用于一个包含 s 个对象的栈上的第 i 个操作是个 PUSH 操作，则势差为

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$$

根据等式(17.2)，这个 PUSH 操作的平摊代价为

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$

假设栈上的第 i 个操作是 MULTIPOP(S, k)，且弹出了 $k' = \min(k, s)$ 个对象。该操作的实际代

价为 k'，势差为

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'$$

因此，MULTIPOP 操作的平摊代价为

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$$

类似地，一个普通 POP 操作的平摊代价也是 0。

三种栈操作中每一种的平摊代价都是 $O(1)$，这样包含 n 个操作的序列的总平摊代价就是 $O(n)$。因为我们已经证明了 $\Phi(D_i) \geqslant \Phi(D_0)$，故 n 个操作的总平摊代价即为总的实际代价的一个上界。所以 n 个操作的最坏情况代价为 $O(n)$。

二进制计数器递增 1

作为说明势能方法的另一个例子，再来看二进制计数器的增值问题。这一次，定义第 i 次
414 INCREMENT 操作后计数器的势为 b_i，即第 i 次操作后计数器中 1 的个数。

我们来计算一个 INCREMENT 操作的平摊代价。假设第 i 次 INCREMENT 操作对 t_i 个位进行了复位。该操作的实际代价至多是 t_i+1，因为除了将 t_i 个位复位外，它至多将 1 个位设为 1。如果 $b_i=0$，那么第 i 个操作复位 k 个位，则 $b_{i-1}=t_i=k$。如果 $b_i>0$，则 $b_i=b_{i-1}-t_i+1$。在这两种情况中，都有 $b_i \leqslant b_{i-1}-t_i+1$，而且势差为

$$\Phi(D_i) - \Phi(D_{i-1}) \leqslant (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$$

因此平摊代价为

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leqslant (t_i + 1) + (1 - t_i) = 2$$

如果计数器开始时为 0，则 $\Phi(D_0)=0$。因为对所有的 i 有 $\Phi(D_i) \geqslant 0$，故 n 次 INCREMENT 操作的序列的总平摊代价就为总的实际代价的一个上界，且 n 次 INCREMENT 操作的最坏情况代价为 $O(n)$。

势能方法给我们提供了一个简单方法来分析一个计数器，即使它开始时不为零。开始时有 b_0 个 1，在 n 次 INCREMENT 操作之后有 b_n 个 1，此处 $0 \leqslant b_0, b_n \leqslant k$。（$k$ 是计数器的位数。）我们可将等式(17.3)重写为

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) \tag{17.4}$$

对于所有的 $1 \leqslant i \leqslant n$，我们有 $\hat{c}_i \leqslant 2$。因为 $\Phi(D_0)=b_0$ 而且 $\Phi(D_n)=b_n$，n 次 INCREMENT 操作总的实际代价等于

$$\sum_{i=1}^{n} c_i \leqslant \sum_{i=1}^{n} 2 - b_n + b_0 = 2n - b_n + b_0$$

特别要注意，因为 $b_0 \leqslant k$，只要 $k=O(n)$，总的实际代价就是 $O(n)$。换言之，如果我们执行了至少 $n=\Omega(k)$ 次 INCREMENT 操作，则无论计数器中包含什么样的初始值，总的实际代价都是 $O(n)$。

练习

17.3-1　假设有势函数 Φ，使得对所有的 i 都有 $\Phi(D_i) \geqslant \Phi(D_0)$，但是 $\Phi(D_0) \neq 0$。证明：存在一个势函数 Φ' 使得 $\Phi'(D_0)=0$，$\Phi'(D_i) \geqslant 0$，对所有 $i \geqslant 1$，且用 Φ' 表示的平摊代价与用 Φ
415 表示的平摊代价相同。

17.3-2　用势能方法的分析重做练习 17.1-3。

17.3-3　考虑一个包含 n 个元素的普通二叉最小堆数据结构，它支持最坏情况时间代价为 $O(\lg n)$ 的操作 INSERT 和 EXTRACT-MIN。请给出一个势函数 Φ，使得 INSERT 的平摊代价为 $O(\lg n)$，EXTRACT-MIN 的平摊代价为 $O(1)$，并证明函数确实是有用的。

17.3-4 假设某个栈在执行 n 个栈操作 PUSH、POP 和 MULTIPOP 之前包含 s_0 个对象，结束后包含 s_n 个对象，则这 n 个栈操作的总代价是多少？

17.3-5 假设一个计数器的二进制表示中在开始时有 b 个 1，而不是 0。证明：如果 $n=\Omega(b)$，则执行 n 次 INCREMENT 操作的代价为 $O(n)$。（不能假设 b 是常数。）

17.3-6 说明如何用两个普通的栈来实现一个队列（练习 10.1-6），使得每个 ENQUEUE 和 DEQUEUE 操作的平摊代价都为 $O(1)$。

17.3-7 设计一个数据结构来支持整数集合 S 上的下列两个操作：

INSERT(S, x) 将 x 插入 S 中。
DELETE-LARGER-HALF(S) 删除 S 中最大的$\lceil S/2 \rceil$个元素。

解释如何实现这个数据结构，使得任意 m 个操作的序列在 $O(m)$ 时间内运行。

17.4　动态表

在某些应用中，无法预知要在表中存储多少个对象。为表分配了一定的空间，但后来发现空间并不够用。这样就要为该表重新分配一个更大的空间，而原表中的对象就要复制到新的更大的表中。类似地，如果有许多对象被从表中删去了，就应该为原表重新分配一个更小的空间。在这一节中，我们要研究表的动态扩张和收缩的问题。利用平摊分析，我们要证明插入和删除操作 [416] 的平摊代价仅为 $O(1)$，即使当它们引起了表的扩张和收缩时具有较大的实际代价也如此。此外，还将看到如何来保证动态表中未用的空间始终不超过整个空间的一个常量部分。

假设动态表支持 TABLE-INSERT 和 TABLE-DELETE 操作。TABLE-INSERT 将某一元素插入表中，该元素占据一个槽(slot)，即一个元素占据的空间。同样地，TABLE-DELETE 可以被认为是从表中去除一个元素，从而释放了一个槽。用来组织这种表的数据结构方法的细节不重要；我们可以用栈(10.1 节)，堆(第 6 章)，或者散列表(第 11 章)。我们还可以用一个数组或一组数组来实现对象存储，如同我们在 10.3 节的做法。

我们将发现采用在分析散列技术时(第 11 章)引入的一个概念很方便。定义一个非空表 T 的装载因子 $\alpha(T)$ 为表中存储的元素个数除以表的大小(槽的个数)后的结果。对一个空表(其中没有元素)定义其大小为 0，其装载因子为 1。如果某一个动态表的装载因子以一个常数为上界，则表中未使用的空间就始终不会超过整个空间的一个常数部分。

我们先开始分析只做插入操作的动态表，然后再考虑既允许插入又允许删除的更一般情况。

17.4.1　表扩张

假设一个表的存储空间分配为一个槽的数组。当所有的槽都被占用时，或者等价地，当其装载因子为 1 时[⊖]，一个表就被填满了。在某些软件环境中，如果试图向一个满的表中插入一个项，就只会导致错误而终止。此处假设我们的软件环境(像许多现代的软件环境一样)提供了存储管理系统，它能根据请求来分配或释放存储块。这样，当向一个满的表中插入一个项时，就能对原表进行扩张，即分配一个包含比原表更多的槽的新表。因为我们总是需要这个表驻留在一段连续的内存中，必须为更大的表分配一个新的数组，然后把旧表中的项复制到新表中。

一个常用的启发式方法是分配一个是原表两倍槽数的新表。如果只执行插入操作，则表的 [417] 装载因子始终至少为 1/2，浪费的空间就始终不会超过表空间的一半。

⊖ 在某些情况下，例如在一个开放地址散列表中，如果装载因子等于某个严格小于 1 的常数，则可以认为这个表是满的。（参见练习 17.4-1。）

在下面的伪代码中，假设对象 T 表示一个表。域 *table*[T]包含一个指向表的存储块的指针。域 *num*[T]包含表的项数，域 *size*[T]为表中总的槽数。开始时，表是空的，*num*[T]＝*size*[T]＝0。

```
TABLE-INSERT(T, x)
 1  if size[T]=0
 2    then allocate table[T] with 1 slot
 3         size[T] ← 1
 4  if num[T]=size[T]
 5    then allocate new-table with 2 · size[T] slots
 6         insert all items in table[T] into new-table
 7         free table[T]
 8         table[T] ← new-table
 9         size[T] ← 2 · size[T]
10  insert x into table[T]
11  num[T] ← num[T]+1
```

请注意，这里有两个“插入”过程：TABLE-INSERT 程序本身和第 6 行、第 10 行中表的基本插入(elementary insertion)。可以根据基本插入的操作数来分析 TABLE-INSERT 的运行时间，令每个基本插入操作的代价为 1。假设 TABLE-INSERT 的实际运行时间与插入项的时间成线性关系，使得在第 2 行中分配初始表的开销为常数，而第 5 行与第 7 行中分配和释放存储的开销由第 6 行中转移所有表项的代价决定。称第 5～9 行中执行 then 语句的事件为一次扩张。

现在来分析一下作用于一个初始为空的表上的 n 次 TABLE-INSERT 操作的序列。第 i 次操作的代价 c_i 怎样？如果在当前的表中还有空间(或者如果该操作是第一个操作)，则 $c_i=1$，因为这时只需在第 10 行中执行一次基本插入操作即可。如果当前的表是满的，则发生一次扩张，这时 $c_i=i$：第 10 行中基本插入操作的代价 1，再加上第 6 行中将原表中的项复制到新表中的代价 $i-1$。如果执行了 n 次操作，则一次操作的最坏情况代价为 $O(n)$，由此可得 n 次操作的总的运行时间的上界 $O(n^2)$。

这个界不很紧确，因为在执行 n 次 TABLE INSERT 操作的过程中，并不经常包括扩张表的
418 代价。特别地，仅当 $i-1$ 为 2 的整数幂时，第 i 次操作才会引起一次表的扩张。实际上，一次操作的平摊代价为 $O(1)$。这一点我们可以用聚集分析来证明。第 i 次操作的代价是

$$c_i=\begin{cases} i & \text{如果 } i-1 \text{ 是 2 的整数次幂} \\ 1 & \text{否则} \end{cases}$$

由此，n 次 TABLE-INSERT 操作的总代价为

$$\sum_{i=1}^{n} c_i \leqslant n+\sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n+2n=3n$$

因为至多有 n 次操作的代价为 1，而余下的操作的代价就构成了一个几何级数。因为 n 次 TABLE-INSERT 操作的总代价为 $3n$，故每一次操作的平摊代价为 3。

通过采用记账方法，我们可以对为什么一次 TABLE-INSERT 操作的平摊代价会是 3 有一些认识。从直觉上看，每一项要支付三次基本插入操作：将其自身插入当前表中，当表扩张时其自身的移动，以及对另一个在扩张表时已经移动过的另一项的移动。例如，假设刚刚完成扩张后某一表的大小为 m。那么表中共有 $m/2$ 项，且没有“存款”。对每一次插入操作收费 3 元。随即发生的基本插入的代价为 1 元。另有 1 元放在刚插入的元素上作为存款。余下的 1 元放在已在表中的 $m/2$ 个项上的某一个作为存款。填满该表另需要 $m/2-1$ 次插入，因此，到该表包含了 m 个项(已满)时，每一项都有 1 元钱来支付在表扩张期间的再次插入。

势能方法也可以用来分析一系列 n 个 TABLE-INSERT 操作，我们还将在 17.4.2 节设计一个平摊代价为 $O(1)$ 的 TABLE-DELETE 操作。开始时先定义一个势函数 Φ，在刚完成扩张时它为 0，当表满时它也达到表的大小，这样下一次扩张的代价就可由势来支付了。函数

$$\Phi(T) = 2 \cdot num[T] - size[T] \tag{17.5}$$

是一种可能的选择。在刚刚完成一次扩张后，我们有 $num[T] = size[T]/2$，于是有 $\Phi(T) = 0$，这正是所希望的。在将要扩张前，有 $num[T] = size[T]$，于是 $\Phi(T) = num[T]$，这也正是所希望的。势的初值为 0，又因为表总是至少为半满，$num[T] \geqslant size[T]/2$，这就意味着 $\Phi(T)$ 总是非负的。所以，n 次 TABLE-INSERT 操作的总的平摊代价就是总的实际代价的一个上界。 419

为了分析第 i 次 TABLE-INSERT 操作的平摊代价，用 num_i 表示在第 i 次操作后表中所存放的项数，$size_i$ 表示在第 i 次操作后表的总大小，Φ_i 表示在第 i 次操作之后的势。开始时，有 $num_0 = 0$，$size_0 = 0$ 和 $\Phi_0 = 0$。

如果第 i 次 TABLE-INSERT 操作没有触发表扩张，则 $size_i = size_{i-1}$，且该操作的平摊代价为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} = 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) = 3\end{aligned}$$

如果第 i 次操作确实触发了一次扩张，则 $size_i = 2 \cdot size_{i-1}$，且 $size_{i-1} = num_{i-1} = num_i - 1$，这意味着 $size_i = 2 \cdot (num_i - 1)$。因此，操作的平摊代价为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} = num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) = 3\end{aligned}$$

图 17-3 画出了 num_i，$size_i$ 和 Φ_i 的各个值。注意势是如何增长来支付表的扩张的。

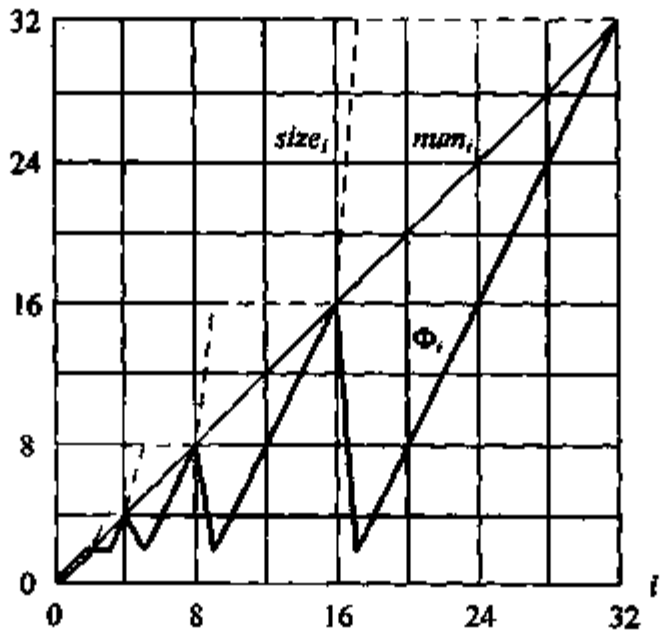


图 17-3　一个含 n 个 TABLE-INSERT 操作的序列作用于表中项目数 num_i，表中的槽数 $size_i$，以及势 $\Phi_i = 2 \cdot num_i - size_i$ 上的效果，这些量在第 i 次操作后加以计算。图中细线表示 num_i，粗线表示 Φ_i，虚线表示 $size_i$。注意在每一次扩张前，势已增长到等于表中的项目数，因而可以支付所有元素移到新表中去的代价。此后，势降为 0，但一旦引起扩张的项目被插入时其值就立即增加 2

17.4.2　表扩张和收缩

为了实现 TABLE-DELETE 操作，只要将指定的项从表中去掉即可。但是当表的装载因子过小时，我们希望对表进行收缩，使得浪费的空间不致太大。表收缩与表扩张是类似的：当表中

的项数降得过低时，就要分配一个新的、更小的表，而后将旧表的各项复制到新表中。旧表所占用的存储空间则可被释放，归还到存储管理系统中去。在理想情况下，我们希望下面的两个性质成立：

- 动态表的装载因子由一个常数从下方限界，以及。
- 表操作的平摊代价由一个常数从上方限界。

另外，我们假设用基本插入和删除操作来度量代价。

关于表收缩和扩张的一个自然的策略是当向满表中插入一个项时，将表的规模扩大一倍，而当从表中删除一个项就导致表不足半满时，则将表缩小一半。这个策略保证了表的装载因子始终不低于 1/2，但不幸的是，这样又会导致各种表操作具有较大的平摊代价。考虑一下下面的这种情况。对某一表 T 执行 n 次操作，此处 n 是 2 的整数幂。前 $n/2$ 个操作是插入，由前面的分析可知其总代价为 $\Theta(n)$。在这一系列插入操作的结束处，$num[T]=size[T]=n/2$。对后面的 $n/2$ 个操作，我们执行下面这样一个序列：

I, D, D, I, I, D, D, I, I, …

其中 I 表示插入，D 表示删除。第一次插入导致表扩张至规模 n。紧接的两次删除又将表的大小收缩至 $n/2$。紧接的两次插入又导致表的另一次扩张等。每次扩张和收缩的代价为 $\Theta(n)$，共有 $\Theta(n)$ 次扩张和收缩。因此，n 次操作的总代价为 $\Theta(n^2)$，而每一次操作的平摊代价为 $\Theta(n)$。

420 ~ 421

这种策略的难度是显而易见的：在一次扩张之后，我们没有做足够的删除来支付一次收缩的代价。类似地，在一次收缩后，我们也没有做足够的插入来支付一次扩张的代价。

我们可以对这个策略加以改进，即允许装载因子低于 1/2。具体来说，当向满的表中插入一项时，仍然将表扩大一倍，但当删除一项而引起表不足 1/4 满时（而不是先前的 1/2），就将表缩小为原来的一半。这样，表的装载因子就由常数 1/4 从下方限界。这种做法的基本思想是使扩张以后表的装载因子为 1/2。因而，在发生一次收缩前要删除表中一半的项，因为只有当装载因子低于 1/4 时才会发生收缩。同理，在收缩之后，表的装载因子也是 1/2。这样，在发生扩张前，要通过插入将表中的项数增加一倍，因为只有当表的装载因子超过 1 时，才能发生扩张。

我们略去了 TABLE-DELETE 的代码，因为它与 TABLE-INSERT 的代码是类似的。但是，为了方便分析，假定如果表中的项数降至 0，就释放该表占据的存储空间。亦即，如果 $num[T]=0$，则 $size[T]=0$。

现在，就可以用势能方法来分析由 n 个 TABLE-INSERT 和 TABLE-DELETE 操作构成的序列的代价了。先定义一个势函数 Φ，它在刚完成一次扩张或收缩时值为 0，并随着装载因子增至 1 或降至 1/4 而变化。我们用 $\alpha(T)=num[T]/size[T]$ 来表示一个非空表 T 的装载因子。因为对一个空表，有 $num[T]=size[T]=0$ 且 $\alpha(T)=1$，故总有 $num[T]=\alpha(T)\cdot size[T]$，无论该表是否为空。我们采用的势函数为

$$\Phi(T)=\begin{cases}2\cdot num[T]-size[T] & \text{如果 } \alpha(T)\geqslant 1/2\\ size[T]/2-num[T] & \text{如果 } \alpha(T)<1/2\end{cases} \tag{17.6}$$

请注意空表的势为 0 且势总是非负的。因此，以 Φ 表示的一列操作的总平摊代价即为其实际代价的一个上界。

在进行详细分析之前，先来看看势函数的某些性质。当装载因子为 1/2 时，势为 0。当它为 1 时，有 $size[T]=num[T]$，这就意味着 $\Phi(T)=num[T]$，因此当因插入一项而引起一次扩张时，就可用势来支付其代价。其装载因子为 1/4 时，有 $size[T]=4\cdot num[T]$，它意味着 $\Phi(T)=$

$num[T]$，因此当因删除一项而引起一次收缩时，就可用势来支付其代价。图 17-4 说明了对一系列操作，势是如何变化的。

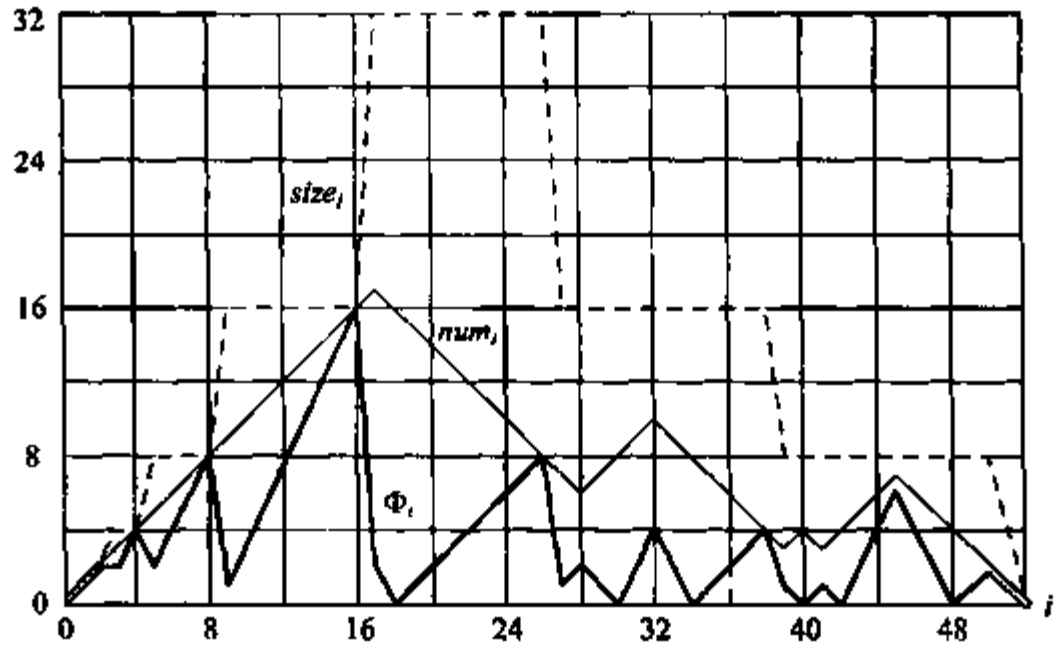


图 17-4　一个含 n 个 TABLE-INSERT 操作的序列作用于表中项目数 num_i，表中的槽数 $size_i$，以及势 $\Phi_i = \begin{cases} 2 \cdot num_i - size_i & \text{如果 } \alpha_i \geqslant 1/2 \\ size_i/2 - num_i & \text{如果 } \alpha_i < 1/2 \end{cases}$ 上的效果。每个值都是在第 i 次操作之后测量。细线表示 num_i，虚线表示 $size_i$，粗线表示 Φ_i。注意在一次扩张之前，势增加到等于表中的项目数，故它能支付将所有元素移到新表中的代价。类似地，在一次收缩之前，势也增大到等于表中的项目数

为分析 n 个 TABLE-INSERT 和 TABLE-DELETE 的操作序列，我们用 c_i 表示第 i 次操作的实际代价，$\hat{c}_i$ 表示其关于 Φ 的平摊代价，num_i 表示在第 i 次操作之后表中存储的项数，$size_i$ 表示在第 i 次操作之后表的大小，α_i 表示在第 i 次操作之后表的装载因子，Φ_i 表示在第 i 次操作之后表的势。开始时，$num_0 = 0$，$size_0 = 0$，$\alpha_0 = 1$，$\Phi_0 = 0$。

我们从第 i 次操作是 TABLE-INSERT 的情况开始分析。如果 $\alpha_{i-1} \geqslant 1/2$，则所要做的分析就与 17.4.1 节对表扩张的分析完全一样。无论表是否进行了扩张，该操作的平摊代价 $\hat{c}_i$ 都至多是 3。如果 $\alpha_{i-1} < 1/2$，则表不会因该操作而扩张，因为仅当 $\alpha_{i-1} = 1$ 时才发生扩张。如果还有 $\alpha_i < 1/2$，则第 i 个操作的平摊代价为

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) = 0 \end{aligned}$$

422 ~ 423

如果 $\alpha_{i-1} < 1/2$ 但 $\alpha_i \geqslant 1/2$，那么

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} = 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\ &= 3 \cdot num_{i-1} - \frac{3}{2} size_{i-1} + 3 = 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\ &< \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3 = 3 \end{aligned}$$

因此，一次 TABLE-INSERT 操作的平摊代价至多为 3。

现在再来分析一下第 i 个操作是 TABLE-DELETE 的情形。这时，$num_i = num_{i-1}$。如果 $\alpha_{i-1} < 1/2$，就必须考虑该操作是否会引起一次收缩。如果没有，则 $size_i = size_{i-1}$，这个操作的

平摊代价为

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1})$$
$$= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) = 2$$

如果 $\alpha_{i-1} < 1/2$ 且第 i 个操作确实引起一次收缩，则该操作的实际代价为 $c_i = num_i + 1$，因为我们删除了一项，移动了 num_i 项。这时，$size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$，该操作的平摊代价为

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1})$$
$$= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) = 1$$

当第 i 次操作为 TABLE-DELETE 且 $\alpha_{i-1} \geqslant 1/2$ 时，其平摊代价仍由一常数从上方限界。有关分析留作练习 17.4-2。

总之，因为每个操作的平摊代价都有一常数上界，所以作用于一动态表上的 n 个操作的实际时间为 $O(n)$。

练习

17.4-1 假设我们希望实现一个动态开放地址散列表。当表的装载因子达到某个严格小于 1 的数 α 时，就可以认为表满了，这是为什么？请简要说明如何对一个动态的开放地址散列表进行插入，使得每个插入操作的平摊代价的期望值为 $O(1)$。为什么每个插入操作的实际代价的期望值不必是 $O(1)$？

424

17.4-2 证明：如果作用于一动态表上的第 i 个操作是 TABLE-DELETE，且 $\alpha_{i-1} \geqslant 1/2$，则以势函数(17.6)表示的每个操作的平摊代价由一个常数从上方限界。

17.4-3 假设当某个表的装载因子降至 1/4 以下时，我们不是通过将其大小缩小一半来收缩，而是在表的装载因子低于 1/3 时，通过将其大小乘以 2/3 来进行收缩。利用势函数

$$\Phi(T) = |2 \cdot num[T] - size[T]|$$

来证明采用这种策略的 TABLE-DELETE 操作的平摊代价由一个常数从上方限界。

思考题

17-1　位反向的二进制计数器

第 30 章要介绍一种重要的算法，称为快速傅里叶变换，或 FFT。FFT 算法的第一步在输入数组 $A[0..n-1]$ 上执行一次位反向置换，其中数组长度 $n = 2^k$（k 为某个非负整数）。这个置换将某些元素进行交换，这些元素的下标的二进制表示彼此相反。

可以将每个下标 a 表示成一个 k 位的序列$\langle a_{k-1}, a_{k-2}, \cdots, a_0\rangle$，其中 $a = \sum_{i=0}^{k-1} a_i 2^i$。定义

$$\mathrm{rev}_k(\langle a_{k-1}, a_{k-2}, \cdots, a_0\rangle) = \langle a_0, a_1, \cdots, a_{k-1}\rangle$$

因此，

$$\mathrm{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i$$

例如，如果 $n = 16$（或等价地，$k = 4$），则 $\mathrm{rev}_k(3) = 12$，因为数值 3 的 4 位表示为 0011，将其反向得 1100，即 12 的 4 位表示。

425

a)给定一个运行时间为 $\Theta(k)$ 的函数 rev_k，请写出一个能在 $O(nk)$ 时间内完成对一个长度为 $n = 2^k$ 的数组的位反向置换的算法。

我们可以用一个基于平摊分析的算法来改善位反向置换的运行时间，方法是采用一个“位反向计数器”和一个程序 BIT-REVERSED-INCREMENT，该程序在给定一个位反向计数器的值 a 后，给出 $\text{rev}_k(\text{rev}_k(a)+1)$。例如，如果 $k=4$，且计数器的初始值为 0，则连续调用 BIT-REVERSED-INCREMENT 就产生序列

0000，1000，0100，1100，0010，1010，… = 0，8，4，12，2，10，…

b)假设计算机中的一个字可以存储 k 位的值，且机器可以在单位时间内，完成对这些二进制值的诸如左移或右移任意位、按位与、按位或等操作。请给出过程 BIT-REVERSED-INCREMENT 的一个实现，使之能在 $O(n)$ 时间内，完成一个含 n 个元素的位反向置换。

c)假设可以在单位时间内对一个字左移或右移一位。这时是否仍可能实现一个 $O(n)$ 时间的位反向置换？

17-2　使二叉查找动态化

对一个已排序数组的二叉查找要花对数的时间，而插入一个新元素的时间则与数组的大小成线性关系。通过使用若干排好序的数组，就可以改善插入的时间。

具体来说，假设希望在一个包含 n 个元素的集合上支持 SEARCH 和 INSERT 操作。设 $k=\lceil \lg(n+1) \rceil$，且 n 的二进制表示为 $\langle n_{k-1}, n_{k-2}, \cdots, n_0 \rangle$。我们有 k 个已排序的数组 $A_0, A_1, \cdots, A_{k-1}$，其中对 $i=0, 1, \cdots, k-1$，数组 A_i 的长度为 2^i。每个数组或者是满的，或者是空的，取决于 $n_i=1$ 还是 $n_i=0$。k 个数组中共有元素 $\sum_{i=0}^{k-1} n_i 2^i = n$ 个。虽然每个数组都是已排序的，但属于不同的数组中的元素之间却没有什么特别的关系。

a)请描述如何在这种数据结构上做 SEARCH 操作，并分析其最坏情况运行时间。

b)请说明如何向这种数据结构中插入一个新元素，并分析插入操作的最坏情况和平摊的运行时间。

c)讨论如何实现 DELETE。

426

17-3　平摊加权平衡树

假设在一棵普通的二叉查找树中，对每个结点 x 增加一个域 $size[x]$，它表示在以 x 为根的子树中关键字的个数。设 α 是在范围 $1/2 \leqslant \alpha < 1$ 之间的一个常数。我们说某一给定的结点 x 是 α 平衡的，如果

$$size[left[x]] \leqslant \alpha \cdot size[x]$$

且

$$size[right[x]] \leqslant \alpha \cdot size[x]$$

如果树中每个结点都是 α 平衡的，则整棵树就是 α 平衡的。下面的维护权平衡树的平摊方法是由 G. Varghese 提出的。

a)从某种程度上来说，一棵 1/2 平衡的树已达到了最大可能的平衡程度。给定任意一棵二叉查找树中的某个结点 x，说明如何重建以 x 为根的子树，使之为 1/2 平衡。所给出的算法的运行时间应为 $\Theta(size[x])$，且可以利用 $O(size[x])$ 的辅助存储空间。

b)证明：在最坏情况下，对一棵有 n 个结点、α 平衡的二叉查找树做一次查找要花 $O(\lg n)$ 时间。

在这个问题的余下部分，假设常数 α 严格大于 1/2。另假设 INSERT 和 DELETE 如在通常的 n 个结点的二叉查找树上一样实现，只是在每次操作之后，如果树中有任何结点不

再是 α 平衡的了，则重建以树中最高的这样的结点为根的子树，使之成为 1/2 平衡的。

我们用势能方法对这种重建方法进行分析。对二叉查找树 T 中的一个结点 x，定义

$$\Delta(x) = | size[left[x]] - size[right[x]] |$$

再定义 T 的势为

$$\Phi(T) = c \sum_{x \in T, \Delta(x) \geqslant 2} \Delta(x)$$

其中 c 是依赖于 α 的足够大的常数。

c)论证：任何二叉查找树都具有非负的势；一棵 1/2 平衡树的势为 0。

d)假设可用 m 单位的势来支付重建一棵包含 m 个结点的子树。若要在 $O(1)$ 平摊时间内重建一棵非 α 平衡的子树，则 c 应为多大(以 α 表示)?

e)证明：对一棵包含 n 个结点的 α 平衡树，插入一个结点或删除一个结点需要 $O(\lg n)$ 平摊时间。

427

17-4　重构红黑树的代价

在红黑树上，有 4 种基本操作会做结构性的修改，它们是结点插入、结点删除、旋转以及颜色修改操作。我们知道，RB INSERT 与 RB-DELETE 只使用 $O(1)$ 次旋转、结点插入和结点删除来保持红黑树的性质，但是它们可能会做更多的颜色修改。

a)描述一棵有 n 个结点的合法红黑树，使得调用 RB-INSERT 来插入第 $(n+1)$ 个结点会引起 $\Omega(\lg n)$ 次颜色修改。然后描述一棵有 n 个结点的合法红黑树，使得在一个特殊结点上调用 RB-DELETE 会引起 $\Omega(\lg n)$ 次颜色修改。

虽然每个操作的颜色修改的最坏情况次数可能是对数的，我们将证明在一棵初始为空的红黑树上，在最坏情况下，任何有 m 个 RB-INSERT 和 RB-DELETE 操作的序列会引起 $O(m)$ 次结构修改。

b)由 RB-INSERT-FIXUP 和 RB-DELETE-FIXUP 的代码中的主循环来处理的某些情况是终止的：一旦遇到时，它们在一个常数次数的额外操作后会引起这个循环终止。对 RB-INSERT-FIXUP 和 RB-DELETE-FIXUP 的每一种情况，指出哪一些是终止的，哪些不是的。(提示：参见图 13-5、图 13-6 和图 13-7。)

我们先分析只执行插入的结构修改。令 T 是一棵红黑树，并且定义 $\Phi(T)$ 是在 T 内红色结点的个数。假设 1 个单位的势可用来支付 RB-INSERT-FIXUP 的三种情况中的任一个所执行的结构修改的代价。

c)让 T' 是应用 RB-INSERT-FIXUP 的情况 1 到 T 上得到的结果。论证：$\Phi(T') = \Phi(T) - 1$。

d)使用 RB-INSERT 来将结点插入到一棵红黑树中可以分为三个部分。列出从 RB-INSERT 的第 1～16 行，从 RB-INSERT-FIXUP 的非终止情况，以及从 RB-INSERT-FIXUP 的终止情况所产生的结构修改和势的改变。

e)利用 d)部分，来论证任何 RB-INSERT 的调用所执行的结构修改的平摊次数是 $O(1)$。

现在我们希望证明当同时有插入和删除时会有 $O(m)$ 个结构修改。对每个结点 x，我们定义

$$w(x) = \begin{cases} 0 & \text{如果 } x \text{ 是红的} \\ 1 & \text{如果 } x \text{ 是黑的而且没有红子女} \\ 0 & \text{如果 } x \text{ 是黑的而且有一个红子女} \\ 2 & \text{如果 } x \text{ 是黑的而且有两个红子女} \end{cases}$$

428

现在，重新定义一棵红黑树 T 的势为

$$\Phi(T) = \sum_{x \in T} w(x)$$

并且设 T' 是应用 RB-INSERT-FIXUP 或 RB-DELETE-FIXUP 的任何非终止情况到 T 上所得到的结果树。

f)证明对 RB-INSERT-FIXUP 的所有非终止情况，有 $\Phi(T') \leqslant \Phi(T) - 1$。论证任何 RB-INSERT-FIXUP 的调用所执行的结构修改的平摊次数是 $O(1)$。

g)证明对 RB-DELETE-FIXUP 的所有非终止情况，有 $\Phi(T') \leqslant \Phi(T) - 1$。论证任何 RB-DELETE-FIXUP 的调用所执行的结构修改的平摊次数是 $O(1)$。

h)完成这个证明：在最坏情况下，任何有 m 个 RB-INSERT 和 RB-DELETE 操作的序列执行 $O(m)$ 次结构修改。

本章注记

Aho，Hopcroft 和 Ullman [5]中用到了聚集分析。Tarjan [293]研究平摊分析中的记账和势能方法，并给出了一些应用。他将记账方法归功于若干作者，包括 M. R. Brown，R. E. Tarjan，S. Huddleston 和 K. Mehlhorn。他将势能方法归功于 D. D. Sleator。术语“平摊”归功于 D. D. Sleator 和 R. E. Tarjan。

势函数也可以用来证明某些类型问题的下界。对问题的每个设定，定义一个势函数来将这个设定映射到一个实数。然后决定最初设定的势 Φ_{init}，最后设定的势 Φ_{final}，以及由任一步引起的最大的势改变 $\Delta\Phi_{max}$。因此步骤的数目必须至少是 $|\Phi_{final} - \Phi_{init}| / |\Delta\Phi_{max}|$。使用势函数来证明 I/O 复杂性的下界的例子出现在这些工作中：Cormen [71]，Floyd [91]，Aggarwal 和 Vitter [4]。Krumme，Cybenko 和 Venkataraman [194]应用势函数来证明传播消息的下界：图中的各个结点告知其他每一个结点一个唯一的消息。

第五部分　高级数据结构

概　　述

这一部分再回过头来讨论支持动态集合上的操作的数据结构，但是在比第三部分更高级的程度上进行。例如，其中有两章大量地利用我们在第 17 章中介绍的平摊分析技术。

第 18 章介绍 B 树，这是一种被设计成专门存储在磁盘上的平衡查找树。因为磁盘的操作速度要大大慢于随机存取存储器，所以在分析 B 树的性能时，不仅要看动态集合操作花了多少计算时间，还要看执行了多少次磁盘存取操作。对每一种 B 树操作，磁盘存取的次数随 B 树高度的增加而增加，而各种 B 树操作又能使 B 树保持较低的高度。

第 19 章和第 20 章给出可合并堆的几种实现。这种堆支持操作 INSERT，MINIMUM，EXTRACT-MIN 和 UNION[⊖]。UNION 操作用于合并两个堆。这两章中出现的数据结构还支持 DELETE 和 DECREASE-KEY 操作。

第 19 章中出现的二项堆结构能在 $O(\lg n)$ 最坏情况时间内支持以上各种操作，此处 n 为输入堆中的总元素数(在 UNION 情况下为两个输入堆中的总的元素数)。当必须支持 UNION 操作时，二项堆优越于在第 6 章中介绍的二叉堆，因为后者在最坏情况下，合并两个二叉堆要花 $\Theta(n)$ 时间。

第 20 章介绍的斐波那契堆对二项堆有改进，至少从理论上说是这样的。我们将用平摊时间界来度量斐波那契堆的性能。对这种堆，操作 INSERT，MINIMUM 和 UNION 仅花 $O(1)$ 的实际和平摊时间，而操作 EXTRACT-MIN 和 DELETE 要花 $O(\lg n)$ 的平摊时间。然而，斐波那契堆的最重要的优点在于 DECREASE-KEY 仅花 $O(1)$ 的平摊时间。正是由于该操作具有这样低的平摊时间，才使得在某些迄今为止渐近最快的图问题算法中，斐波那契堆为其核心部分的原因。

最后，第 21 章介绍用于不相交集合的一些数据结构。由 n 个元素构成的全域被划分成若干动态集合。开始时，每个元素属于由其自身所构成的单元集合。操作 UNION 将两个集合加以合并，而查询 FIND-SET 则可确定给定的元素当前所属的那个集合。通过用一棵简单的有根树来表示每个集合，就可以得到惊人的快速的操作：一个由 m 个操作构成的序列的运行时间为 $O(m\alpha(n))$，其中 $\alpha(n)$ 是一个增长得极慢的函数——在任何可想像的应用中，$\alpha(n)$ 至多为 4。这个问题的数据结构简单，但用来证明这个时间界的平摊分析却比较复杂。

这一部分介绍的主题绝对不是仅有的“高级”数据结构的例子。其他的高级数据结构，如：

- *动态树*(由 Sleator 和 Tarjan [281]提出，并由 Tarjan [292]所论述)，维护一个不相交的有根树的森林。每棵树内的每条边都有一个实数值的代价。动态树支持寻找双亲、根、边的代价，以及从一个结点到根的路径上的最小边代价的查询。可以通过下述手段来操纵

⊖ 如思考题 10-2 中一样，我们已经定义了支持 MINIMUM 和 EXTRACT-MIN 的可合并堆，因此可以把它称为可合并的最小堆。或者，如果它支持 MAXIMUM 和 EXTRACT-MAX，则它是一个可合并的最大堆。除非我们另外指定，可合并的堆默认就是可合并的最小堆。

树：割边，更新从一个结点到根的路径上所有边的代价，把一个根链接到另外一棵树，以及把一个结点变为所在树的根。在动态树的一种实现中，每个操作具有 $O(\lg n)$的平摊时间界；在另一种更复杂的实现中，最坏情况时间界为 $O(\lg n)$。动态树常用在一些渐近最快的网络流算法中。

- 伸展树(splay trees，由 Sleator 和 Tarjan [282]引入，并由 Tarjan [292]所论述)，是一种二叉查找树，标准的查找树操作在其上以 $O(\lg n)$的平摊时间运行。伸展树的一个应用是简化动态树。
- 持久的数据结构允许在过去版本的数据结构上做查询，甚至是有时候做更新。Driscoll，Sarnak，Sleator 和 Tarjan [82]提出只需很小的时空代价，就可以使链式数据结构持久化
432 的技术。思考题 13-1 给出持久动态集的一个简单示例。
- 一些数据结构允许在关键字的一个带限制全域做字典操作(INSERT，DELETE 和 SEARCH)的一个更快的实现。利用这些限制的优点，它们能够得到比基于比较的数据结构更好的最坏情况渐近运行时间。在关键字全域为集合$\{1, 2, \cdots, n\}$的限制条件下，van Emde Boas [301]发明的一个数据结构以最坏情况时间 $O(\lg\lg n)$支持下列操作：MINIMUM，MAXIMUM，INSERT，DELETE，SEARCH，EXTRACT-MIN，EXTRACT-MAX，PREDECESSOR 以及 SUCCESSOR。Fredman 和 Willard [99]引入了聚合树(fusion tree)，它是当限制全域为整数时，第一个允许更快的字典操作的数据结构。他们说明了如何在 $O(\lg n/\lg\lg n)$时间内实现这些操作。包括指数查找树[16]在内的一些数据结构，也给出某些或全部字典操作的改进的界，本书的章节注记中会提到它们。
- 动态图数据结构在允许通过插入或删除顶点或边的操作来改变图的结构的同时，还支持各种查询。支持查询的例子包括顶点连通性[144]，边连通性，最小生成树[143]，双连通性，以及传递闭包[142]等。

433 本书的章节注记中还会提到另外的一些数据结构。

第 18 章　B　树

B 树是为磁盘或其他直接存取辅助存储设备而设计的一种平衡查找树。B 树与第 13 章介绍的红黑树类似，但在降低磁盘 I/O 操作次数方面要更好一些。许多数据库系统使用 B 树或 B 树的变形来储存信息。

B 树与红黑树的主要不同在于，B 树的结点可以有许多子女，从几个到几千个。这就是说，B 树的“分支因子”可能很大，这一因子常常是由所使用的磁盘特性所决定的。B 树与红黑树的相似之处在于，每棵含 n 个结点的 B 树的高度为 $O(\lg n)$，但可能要比一棵红黑树的高度小许多，因为它的分支因子较大。所以，B 树也可以被用来在 $O(\lg n)$时间内，实现许多动态集合操作。

B 树以自然的方式推广二叉查找树。图 18-1 给出一棵简单的 B 树。如果 B 树的内结点 x 包含 $n[x]$个关键字，则 x 就有 $n[x]+1$ 个子女。结点 x 中的关键字是用来将 x 所处理的关键字域划分成 $n[x]+1$ 个子域的分隔点，每个子域都由 x 中的一个子女来处理。当在一棵 B 树中查找某个关键字时，通过对储存在结点 x 中的 $n[x]$个关键字的比较，而做出一个$(n[x]+1)$路的决定。叶结点的结构不同于内部结点的结构；我们将在 18.1 节中讨论这些差别。

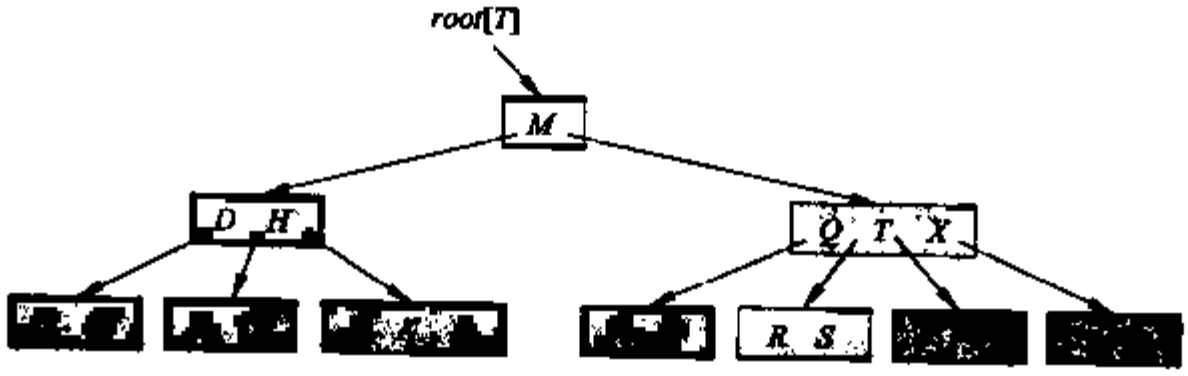


图 18-1　一棵关键字为英语中辅音字母的 B 树。包含 $n[x]$个关键字的内结点 x 有 $n[x]+1$ 个子女。所有的叶结点都处于相同的深度。带浅阴影的结点是在查找字母 R 时要检查的结点

18.1 节给出 B 树的准确定义，并证明 B 树的高度仅随它所包含的结点数按对数增长。18.2 节介绍如何在 B 树中查找或插入一个关键字，18.3 节讨论删除操作。在开始下面内容之前，我们需要弄清楚为什么针对磁盘而设计的数据结构不同于针对随机存取的主存而设计的数据结构。

辅存上的数据结构

有许多不同的技术可用来在计算机系统中提供存储能力。主存一般是由硅制的存储芯片组成。这种技术的每一位代价要比磁存储技术(例如磁带或磁盘)昂贵上两个数量级。许多计算机系统还有基于磁盘的辅存；辅存的容量通常比主存的容量大至少两个数量级。

图 18-2a 显示一个典型的磁盘驱动器。这个驱动器包含若干盘片，它们以固定速度绕共用的主轴旋转；每个盘的表面覆盖一层可磁化的物质。每个磁盘通过磁臂末端的磁头来读写。磁臂是物理连接在一起的，它们可以将磁头移近或远离主轴。当一个给定的磁头静止时，由它底下经过的盘表面称为一个磁道。读/写磁头始终是垂直对齐的，因此它们下面的一组磁道被同时存取。图 18-2b 显示这样的一组磁道，称作柱面。

虽然磁盘比主存便宜而且有较高的容量，但是它们的速度很慢，因为它们有机械移动的部分。有两种机械移动的成分：盘旋转和磁臂移动。在本书写作时，商用的磁盘以每分钟 5400～15 000 转(RPM)的速度旋转，而 7200 RPM 是最常见的。虽然 7200 RPM 看起来很快，但是旋转

一圈要花 8.33 毫秒，比硅存储的常见存取时间 100 纳秒几乎比大 5 个数量级。换句话说，如果要等候一个磁盘旋转完整的一圈，让一个特定的数据项到达读/写磁头下方，我们在这个时间可以存取主存几乎100 000 次!平均来讲只需要等候半圈，但是硅存储体与磁盘的存取时间之间的差异仍然是巨大的。移动磁臂也要花费一些时间。在本书写作时，商用磁盘的平均存取时间在 3～9 毫秒的范围内。

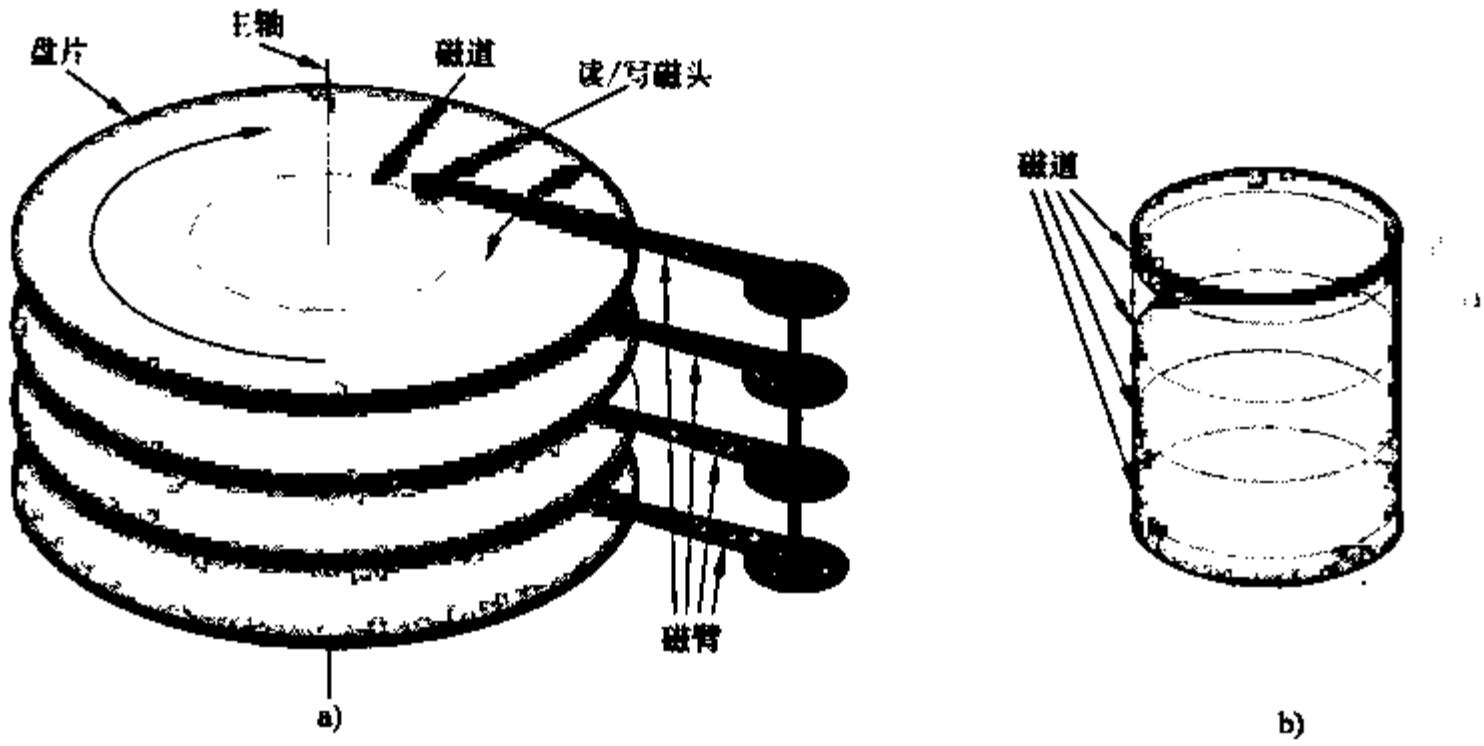


图 18-2　a)一个典型的磁盘驱动器。它由若干绕主轴旋转的盘片组成。每个盘片通过磁臂末端的磁头来读写。这些磁臂连在一起同时移动磁头。在这里，这些磁臂绕着一个共用的传动轴旋转。当读/写磁头静止时，由它下方经过的磁盘表面称为一个磁道。b)柱面由一组垂直的磁道构成

为了平摊等待机械移动所花费的时间，磁盘会一次存取多个数据项，而不仅仅是一个数据项。信息被分割成一些在柱面内连续出现的相等大小的位的页面，每次磁盘读写的都是一个或多个完整页面。对一个典型的磁盘来说，一页的长度可能为 2^{11}到 2^{14}个字节。一旦读/写磁头正确定位，并且盘片已经旋转到所希望页面的开头时，对磁盘的读或写就是完全电子化的(除了盘片的旋转)，大量数据的读或写可以很快地完成。

通常，存取一页的信息并将其从磁盘中读出的时间，要比计算机对所读出的数据进行检查的时间要多。由于这个原因，在这一章里，我们对运行时间的两个主要组成部分分别加以考虑：

- 磁盘存取的次数；
- CPU(计算)时间。

434～436

磁盘存取的次数是按需要从盘中读出或向盘中写入的信息的页数来度量的。我们注意到磁盘的存取时间不是常量 ——它依赖于当前磁道与所需磁道之间的距离以及磁盘的初始旋转状态。但是，我们将用读或写的页数来作为存取磁盘的总时间的主要近似值。

在一个典型的 B 树应用中，要处理的数据量很大，因此无法一次都装入主存。B 树算法将所需的页选择出来复制到主存中去，而后将修改过的页再写回到磁盘上去。因为在任何时刻，B 树算法在主存中都只需要一定量的页数，故主存的大小并不限制可被处理的 B 树的大小。

我们在如下的伪代码中对磁盘操作建模。设 x 为指向某一对象的指针。如果该对象当前在计算机的主存中，则可以如平常一样引用该对象的各个域：例如 $key[x]$。但是，如果 x 指向的对象驻留在磁盘上，则在引用该对象的各个域之前，要先执行操作 DISK-READ(x)，将其读到主存中来。(假定如果 x 已经在主存中，则 DISK-READ(x)就无需任何磁盘访问，即它是一个空

操作。)类似地，操作 DISK-WRITE(x)用来保存对对象 x 的域所作的修改。下面是对对象进行操作的典型模式：

```
x ← a pointer to some object
DISK-READ(x)
operations that access and/or modify the fields of x
DISK-WRITE(x)      ▷ Omitted if no fields of x were changed.
other operations that access but do not modify fields of x
```

在任何时刻，这个系统只能在主存中维持有限的页数。我们将假定不再被使用的页将由系统从主存中换出；后面的 B 树算法将忽略这一点。

在大多数的系统中，B 树算法的运行时间主要由它所执行的 DISK-READ 和 DISK-WRITE 操作的次数所决定，因而应该有效地使用这两种操作，即让它们读或写尽可能多的信息。由于这个原因，在 B 树中，一个结点的大小通常相当于一个完整的磁盘页。因此，一个 B 树结点可以拥有的子女数就由磁盘页的大小所决定。

对储存在磁盘上的一棵大的 B 树，通常采用的分支因子为 50 到 2000，具体要取决于一个关键字相对于一页的大小。选取一个大的分支因子，可以大大地降低树的高度，以及寻找任意关键字时所需的磁盘存取次数。图 18-3 显示了一棵分支因子为 1001、高度为 2 的 B 树，它可以储存超过 10 亿个关键字；尽管如此，因为根结点可以持久地保留在主存中，故在这棵树中，寻找某一关键字至多只需两次磁盘存取！

437

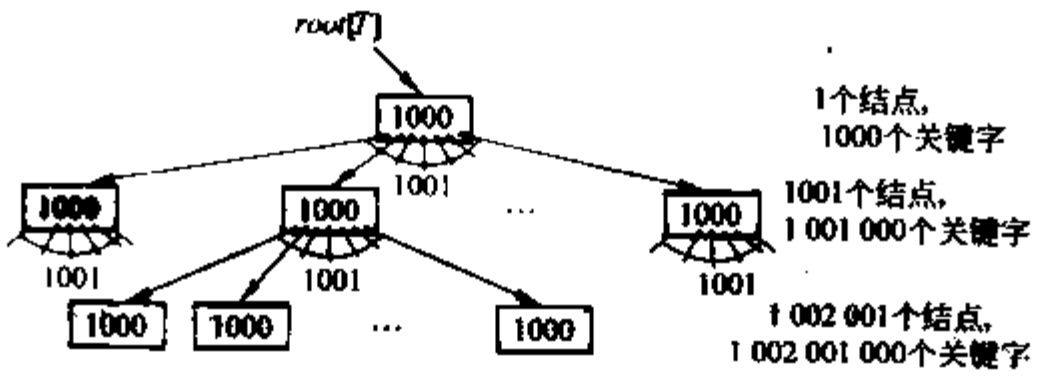


图 18-3 一棵高度为 2 的 B 树包含多于 10 亿个关键字。每个内结点及叶结点包含 1000 个关键字。在深度 1 上共有 1001 个结点，而在深度 2 上有超过 100 万个叶结点。显示在每个结点 x 内的是在 x 内的关键字数目 $n[x]$

18.1 B 树的定义

为简单起见，我们假定，像在二叉查找树和红黑树中一样，和关键字相联系的"附属数据"作为关键字存放在同一结点。在实践中，在每个关键字处可能只是存放了一个指向包含该关键字的"附属数据"的另一个磁盘页的指针。这一章的伪代码都隐含地假设了当关键字从一个结点移动到另一个结点时，无论是与关键字相联系的附属数据，还是指向这样的数据的指针，都随关键字一起移动。一个常见的 B 树的变形，称作 B^+ 树，所有的附属数据都保存在叶结点中，只将关键字和子女指针保存于内结点里，因此最大化了内结点的分支因子。

一棵 B 树 T 是具有如下性质的有根树(根为 $root[T]$)：

1)每个结点 x 有以下域：

a)$n[x]$，当前存储在结点 x 中的关键字数，

b)$n[x]$个关键字本身，以非降序存放，因此 $key_1[x] \leqslant key_2[x] \leqslant \cdots \leqslant key_{n[x]}[x]$，

c)$leaf[x]$，是一个布尔值，如果 x 是叶子的话，则它为 TRUE，如果 x 为一个内结点，则为 FALSE。

2)每个内结点 x 还包含 $n[x]+1$ 个指向其子女的指针 $c_1[x]$，$c_2[x]$，…，$c_{n[x]+1}[x]$。叶结
438 点没有子女，故它们的 c_i 域无定义。

3)各关键字 $key_i[x]$对储存在各子树中的关键字范围加以分隔：如果 k_i 为存储在以 $c_i[x]$为根的子树中的关键字，则

$$k_1 \leqslant key_1[x] \leqslant k_2 \leqslant key_2[x] \leqslant \cdots \leqslant key_{n[x]}[x] \leqslant k_{n[x]+1}$$

4)每个叶结点具有相同的深度，即树的高度 h。

5)每一个结点能包含的关键字数有一个上界和下界。这些界可用一个称作 B 树的最小度数的固定整数 $t \geqslant 2$ 来表示。

a)每个非根的结点必须至少有 $t-1$ 个关键字。每个非根的内结点至少有 t 个子女。如果树是非空的，则根结点至少包含一个关键字。

b)每个结点可包含至多 $2t-1$ 个关键字。所以一个内结点至多可有 $2t$ 个子女。我们说一个结点是满的，如果它恰好有 $2t-1$ 个关键字[⊖]。

$t=2$ 时的 B 树是最简单的。这时每个内结点有 2 个、3 个或 4 个子女，亦即一棵 2-3-4 树。然而在实际中，通常是采用大得多的 t 值。

B 树的高度

B树上大部分操作所需的磁盘存取次数与B树的高度成正比。下面来分析B树的最坏情况高度。

定理 18.1　如果 $n \geqslant 1$，则对任意一棵包含 n 个关键字、高度为 h、最小度数 $t \geqslant 2$ 的 B 树 T，有：

$$h \leqslant \log_t \frac{n+1}{2}$$

证明：如果一棵 B 树的高度为 h，其根结点包含至少一个关键字而其他结点包含至少 $t-1$ 个关键字。这样，在深度 1 至少有两个结点，在深度 2 至少有 $2t$ 个结点，在深度 3 至少有 $2t^2$ 个结点，等等，直到深度 h 至少有 $2t^{h-1}$ 个结点。图 18-4 给出了 $h=3$ 时的一棵树。因此，关键字的
439 个数 n 满足不等式

$$n \geqslant 1+(t-1)\sum_{i=1}^{h} 2t^{i-1} = 1+2(t-1)\left(\frac{t^h-1}{t-1}\right) = 2t^h-1$$

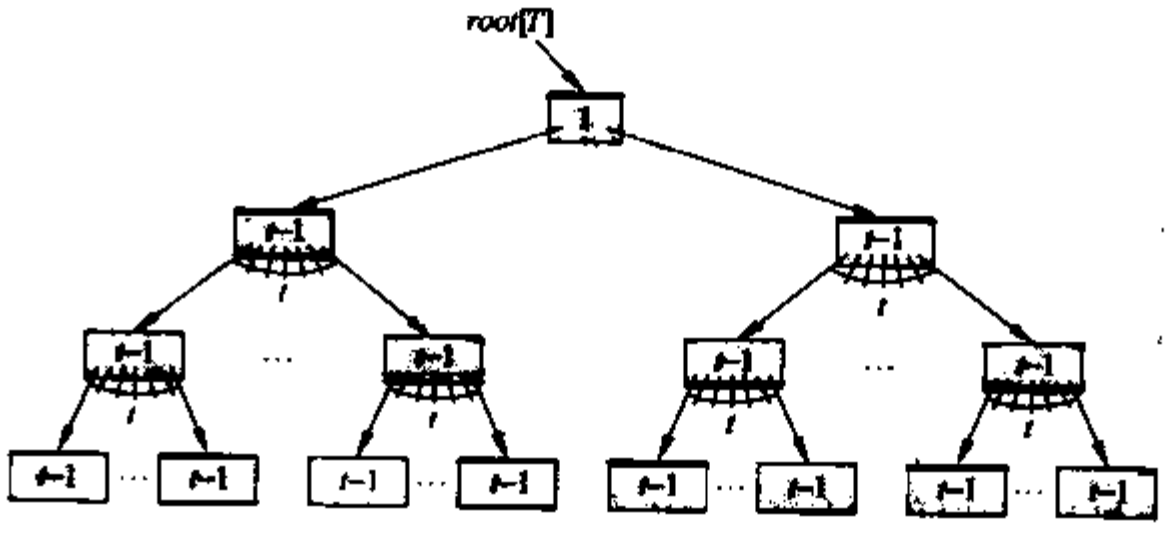


图 18-4　一棵高度为 3 的 B 树，它包含最小可能的关键字数。在每个结点 x 内显示的是 $n[x]$

⊖ B* 树是 B 树的另一种常用变形，在这种树中，要求每个内结点至少为 2/3 满，而不是像 B 树要求的至少半满。

利用简单的代数处理，可以得到 $t^h \leqslant (n+1)/2$。两边取以 t 为底的对数就证明了定理。 ■

与红黑树相比，这里我们看到了 B 树的能力。虽然二者的高度都以 $O(\lg n)$ 的速度增长(注意 t 是个常数)，对 B 树来说对数的底要大很多倍。对大多数的树操作来说，要查找的结点数在 B 树中要比在红黑树中少大约 $\lg t$ 的因子。因为在树中查找任意一个结点通常都需要一次磁盘存取，所以磁盘存取的次数大大地减少了。

练习

18.1-1 为什么不允许 B 树的最小度数 t 为 1?

18.1-2 t 应取什么样的值，才能使图 18-1 中的树是棵合法的 B 树?

18.1-3 请给出表示{1，2，3，4，5}的最小度数为 2 的所有合法的 B 树。 440

18.1-4 一个高度为 h 的 B 树中，可以存储最多多少个关键字? 用最小度数 t 的函数表示。

18.1-5 如果红黑树中的每个黑结点吸收它的红子女，并把它们的子女并入自身，描述这个结果的数据结构。

18.2　对 B 树的基本操作

在这一节里，要给出操作 B-TREE-SEARCH，B-TREE-CREATE 和 B-TREE-INSERT 的细节。在这些过程中，采用两个约定：

- B 树的根结点始终在主存中，因而无需对根做 DISK-READ；但是，每当根结点被改变后，都需要对根结点做一次 DISK-WRITE。
- 任何被当作参数的结点被传递之前，要先对它们做一次 DISK-READ。

我们给出的过程都是"单向"算法，即它们沿树的根下降，没有任何回溯。

搜索 B 树

搜索 B 树与搜索二叉查找树很相似，只是在每个结点所做的不是个二叉或者"两路"分支决定，而是根据该结点的子女数所做的多路分支决定。更准确地说，在每个内结点 x 处，要做 $(n[x]+1)$ 路的分支决定。

B-TREE-SEARCH 是定义在二叉查找树上的 TREE-SEARCH 过程的一个直接推广。它的输入是一个指向某子树的根结点 x 的指针，以及要在该子树中搜索的一个关键字 k。顶层调用的形式为 B-TREE-SEARCH($root[T]$, k)。如果 k 在 B 树中，B-TREE-SEARCH 就返回一个由结点 y 和使 $key_i[y]=k$ 成立的下标 i 组成的有序对 (y, i)。否则返回值 NIL。 441

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ⩽ n[x] and k>key_i[x]
3       do i ← i+1
4  if i ⩽ n[x] and k=key_i[x]
5       then return(x, i)
6  if leaf[x]
7     then return NIL
8     else DISK-READ(c_i[x])
9          return B-TREE-SEARCH(c_i[x], k)
```

利用一个线性搜索过程，第 1～3 行找出使 $k \leqslant key_i[x]$ 成立的最小的下标 i，若找不到就将 i 置为 $n[x]+1$。第 4～5 行检查是否已经找到该关键字，如果找到了则返回。第 6～9 行或者以失败

结束查找(如果 x 是个叶结点)，或者在对子女执行必要的 DISK-READ 后，递归搜索 x 的适当子树。

图 18-1 演示了 B-TREE-SEARCH 的操作过程；浅阴影的结点是在搜索关键字 R 的过程中被检查的结点。

像在二叉查找树的 TREE-SEARCH 过程中那样，在递归过程中所遇到的结点构成了一条从树根下降的路径。因此由 B-TREE-SEARCH 存取的磁盘页面数为 $\Theta(h)=\Theta(\log_t n)$，其中 h 为数的高度，n 为 B 树中所含的关键字个数。因为 $n[x]<2t$，故第 2～3 行中在每个结点处的 while 循环的时间为 $O(t)$，总的 CPU 时间为 $O(th)=O(t\log_t n)$。

创建一棵空的 B 树

为构造一棵 B 树 T，先用 B-TREE-CREATE 来创建一个空的根结点，再调用 B-TREE-INSERT 来加入新的关键字。这两个过程都用到了一个辅助过程 ALLOCATE-NODE，它在 $O(1)$ 时间内为一个新结点分配一个磁盘页。我们可以假定由 ALLOCATE-NODE 所创建的结点无需做 DISK-READ，因为磁盘上还没有关于该结点的有用的信息。

```
B-TREE-CREATE(T)
1  x ← ALLOCATE-NODE()
2  leaf[x] ← TRUE
3  n[x] ← 0
4  DISK-WRITE(x)
5  root[T] ← x
```

442　B-TREE-CREATE 需要 $O(1)$ 次的磁盘操作和 $O(1)$ 的 CPU 时间。

向 B 树插入关键字

与向二叉查找树中插入一个关键字相比，向 B 树中插入一个关键字就要复杂得多了。像在二叉查找树中一样，要查找插入新关键字的叶子位置。但是，在 B 树中，不能简单地创建一个新的叶结点，然后将其插入，因为这样得到的树不再是一棵有效的 B 树。相反地，我们将新关键字插入到一个已存在的叶结点上。因为不能把关键字插入到一个满的叶结点上，故引入一个操作，将一个满的结点 y(有 $2t-1$ 个关键字)按其中间关键字 $key_t[y]$ 分裂成两个各含 $t-1$ 个关键字的结点。中间关键字被提升到 y 的双亲结点，以标识两棵新的树的划分点。但是，如果 y 的双亲也是满的，则它必须在新关键字可以插入之前被分裂，注意满结点的分裂动作会沿着树向上传播。

如同一棵二叉查找树一样，可以从根部沿着树下降到叶子，以便将一个关键字插入到 B 树中。为做到这一点，我们不是等到发现是否真的需要分裂一个满结点时才能做插入。相反地，当沿着树往下查找新关键字所属位置时，就分裂沿途遇到的每个满结点(包含叶结点自身)。因此，每当要分裂一个满结点 y 时，就能确保它的双亲不是满的。

B 树中结点的分裂

过程 B-TREE-SPLIT-CHILD 的输入是一个非满的内结点 x(假定在主存中)，下标 i，以及一个结点 y(同样假设其在主存中)，$y=c_i[x]$ 是 x 的一个满子结点。该过程把这个孩子分裂成两个，并调整 x 使之有一个新增的孩子。(要分裂一个满的根，首先让根成为一个新的空根结点的孩子，这样才能够使用 B-TREE-SPLIT-CHILD。树的高度因此增加 1；分裂是树长高的唯一途径。)

图 18-5 演示了这个过程。满结点 y 按其中间关键字 S 进行分裂，S 则被提升到 y 的双亲结

443　点 x 中。y 中大于中间关键字的那些关键字都置于新结点 z 中，它成为 x 的一个新孩子。

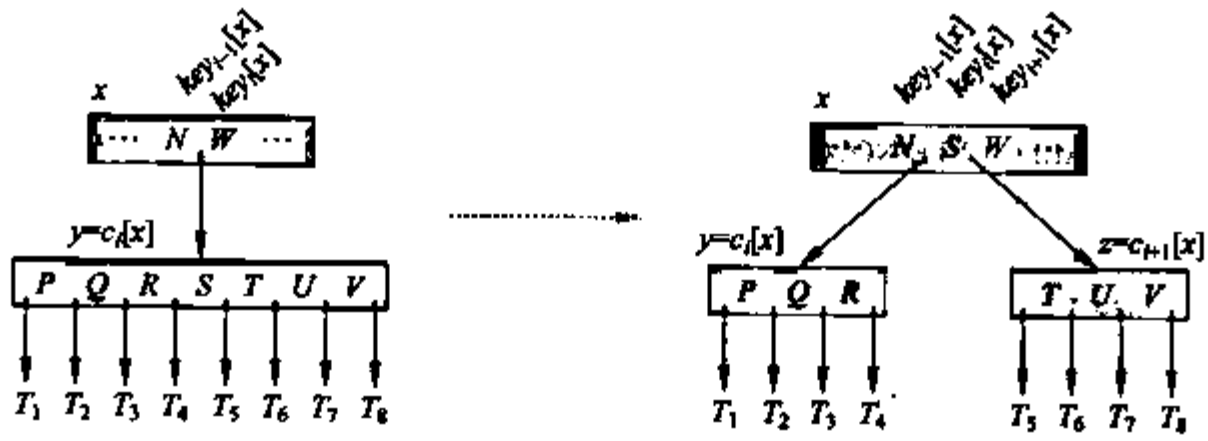


图 18-5　用 $t=4$ 分裂一个结点。结点 y 被分裂为两个结点 y 和 z，而 y 的中间关键字 S 则被提升到 y 的父结点中

```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t−1
 4  for j ← 1 to t−1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7    then for j ← 1 to t
 8              do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t−1
10  for j ← n[x]+1 downto i+1
11       do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14       do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_i[y]
16  n[x] ← n[x]+1
17  DISK-WRITE(y)
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```

B-TREE SPLIT-CHILD 以简单的“剪贴”方式工作。这里 y 是 x 的第 i 个孩子，也是要被分裂的结点。开始时，结点 y 有 $2t$ 个子女($2t-1$ 个关键字)，在分裂后减少至 t 个子女($t-1$ 个关键字)。结点 z“收养”了 y 的 t 个最大的子女($t-1$ 个关键字)，并成为 x 的一个新的孩子，它在 x 的子女表中仅位于 y 之后。y 的中间关键字上升到 x 中，成为分隔 y 和 z 的关键字。 444

第 1～8 行创建结点 z，并将 y 的 $t-1$ 个最大的关键字以及相应的 t 个子女给它。第 9 行对 y 调整关键字计数。最后，第 10～16 行将 z 插入为 x 的一个孩子，提升 y 的中间关键字到 x 来分隔 y 和 z，并调整 x 的关键字计数。第 17～19 行将所有修改的磁盘页面写出。B-TREE-SPLIT-CHILD 占用的 CPU 时间为 $\Theta(t)$，由第 4～5 行和第 7～8 行的循环引起。(其他的循环执行 $O(t)$ 次迭代。)这个过程执行 $O(1)$ 次磁盘操作。

对 B 树用单程下行遍历树方式插入关键字

在一棵高度为 h 的 B 树 T 中，插入一个关键字 k 的操作是在沿树下降的过程中一次完成的，共需要 $O(h)$ 次磁盘存取，所需的 CPU 时间为 $O(th)=O(t\log_t n)$。过程 B-TREE-INSERT 利用 B-TREE-SPLIT-CHILD 保证递归始终不会降至一个满结点上。

```
B-TREE-INSERT(T, k)
 1  r ← root[T]
 2  if n[r]=2t−1
 3    then s ← ALLOCATE-NODE()
 4         root[T] ← s
 5         leaf[s] ← FALSE
 6         n[s] ← 0
 7         c1[s] ← r
 8         B-TREE-SPLIT-CHILD(s, 1, r)
 9         B-TREE-INSERT-NONFULL(s, k)
10    else B-TREE-INSERT-NONFULL(r, k)
```

第 3～9 行处理了根结点 r 为满的情况：对根进行分裂，一个新结点 s(有两个子女)成为根。对根进行分裂是增加 B 树高度的唯一途径。图 18-6 说明了这种情况。与二叉查找树不同，B 树的高度增加是在顶部而不是底部发生。通过调用 B-TREE-INSERT-NONFULL 将关键字 k 插入到以该非满的根结点为根的树中，此程序结束。B-TREE-INSERT-NONFULL 在需要时沿树向下递归，必要时调用 B-TREE-SPLIT-CHILD，在任何时刻保证它所递归处理的结点是非满的。

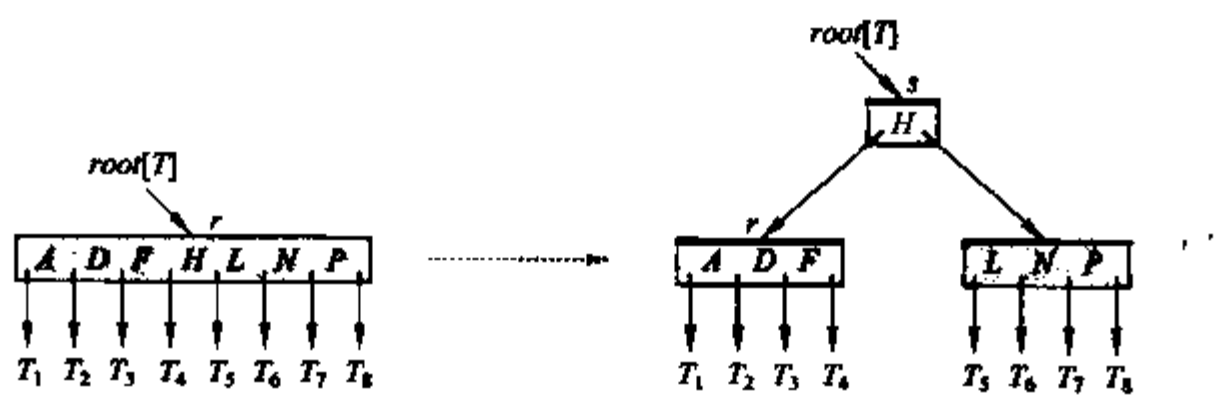


图 18-6　以 $t=4$ 来分裂根部。根结点 r 一分为二，并创建了一个新结点 s。新的根包含了中间关键字 r，且以 r 的两半作为孩子。当根被分裂时，B 树的高度增加 1

辅助的递归过程 B-TREE-INSERT-NONFULL 把关键字 k 插入结点 x，假定在调用该过程时 x 是非满的。操作 B-TREE-INSERT 和递归操作 B-TREE-INSERT-NONFULL 保证了这个假设的正确性。

```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i≥1 and k<keyi[x]
 4               do keyi+1[x] ← keyi[x]
 5                  i ← i−1
 6          keyi+1[x] ← k
 7          n[x] ← n[x]+1
 8          DISK-WRITE(x)
 9     else while i≥1 and k<keyi[x]
10               do i ← i−1
11          i ← i+1
12          DISK-READ(ci[x])
13          if n[ci[x]]=2t−1
14            then B-TREE-SPLIT-CHILD(x, i, ci[x])
15                 if k>keyi[x]
```

```
16              then i ← i+1
17          B-TREE-INSERT-NONFULL(c_i[x], k)
```

B-TREE-INSERT-NONFULL 程序的工作方式如下。第 3～8 行处理 x 是叶子的情况，将关键字 k 插入 x。如果 x 不是叶结点，则应将 k 插入到以内结点 x 为根的子树中适当的叶结点中去。在这种情况下，第 9～11 行确定向 x 的哪个子结点递归下降。第 13 行检查递归是否将降至一个满子结点上，若是，则在第 14 行中用 B-TREE-SPLIT-CHILD 将该子结点分裂成两个非满的孩子，第 15～16 行确定向两个孩子中的哪一个下降是正确的。(注意在第 16 行中增加 i 后无需再做一次 DISK-READ($c_i[x]$)，因为在这种情况下递归要降至一个刚刚由 B-TREE-SPLIT-CHILD 创建的子结点上。)第 13～16 行的实际效果就是保证了该程序始终不会降至一个满结点上。第 17 行递归地将 k 插入到合适的子树中去。图 18-7 说明了向 B 树中插入关键字的各种情况。

445 ～ 446

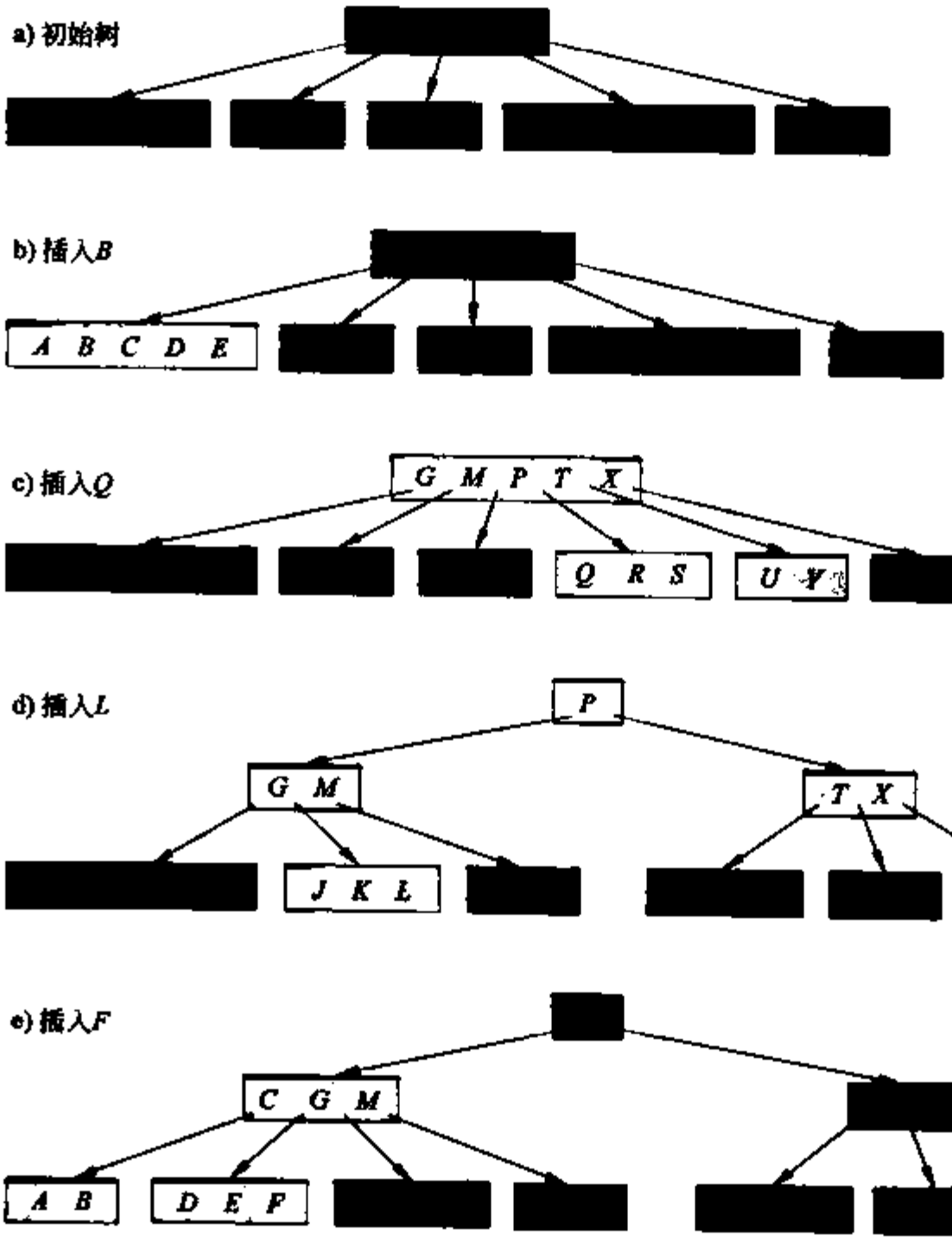


图 18-7 向 B 树中插入关键字。这棵 B 树的最小度数 t 为 3，故一个结点至多可包含 5 个关键字。在插入过程中被修改的结点加了浅阴影。a)这个例子中初始的树。b)将关键字 B 插入初始树后的结果；这是一个对叶结点的简单插入。c)将关键字 Q 插入前一棵树中的结果。结点 $RSTUV$ 被分裂成两个分别包含 RS 和 UV 的结点，关键字 T 被提升到根中，Q 被插入到两半的最左边(RS 结点)中。d)将 L 插入到前一棵树的结果。根结点立即被分裂，因为它是满的，同时 B 树的高度增加 1。L 被插入到包含 JK 的叶结点中。e)将 F 插入到前一棵树的结果。在将 F 插入到两半的最右边(DE 结点)之前，$ABCDE$ 结点要进行分裂

对一棵高度为 h 的 B 树，B-TREE-INSERT 要做的磁盘存取次数为 $O(h)$，因为在调用 B-TREE-INSERT-NONFULL 之间，只做了 $O(1)$ 次 DISK-READ 和 DISK-WRITE 操作。所占用的总的 CPU 时间为 $O(th)=O(t\log_t n)$。因为 B-TREE-INSERT-NONFULL 是尾递归的，故它也可用一个 while 循环来实现，说明了在任何时刻，需要留在主存中的页面数为 $O(1)$。

练习

18.2-1 请给出将关键字 F，S，Q，K，C，L，H，T，V，W，M，R，N，P，A，B，X，Y，D，Z，E 依序插入一棵最小度数为 2 的空的 B 树的结果。只要画出在某些结点分裂之前以及最终的结构。

18.2-2 请解释在什么情况下(如果有的话)，在调用 B-TREE-INSERT 的过程中，会执行冗余的 DISK-READ 或 DISK-WRITE 操作。(所谓冗余的 DISK-READ 是指对已在主存中的某页做 DISK-READ，冗余的 DISK-WRITE 是指将已经存在于磁盘上的某页又完全相同地重写一遍。)

18.2-3 请解释如何在一棵 B 树中，寻找最小关键字和树中某一给定关键字的前驱。

447 ~ 448

***18.2-4** 假设关键字$\{1, 2, \cdots, n\}$被插入一个最小度数为 2 的空 B 树中。最终的 B 树有多少个结点?

18.2-5 因为叶结点无需指向子女的指针，对同样大小的磁盘页，可选用一个与内结点不同的(更大的)t 值。请说明如何修改 B 树的创建和插入的程序来处理这个变形。

18.2-6 假设 B-TREE-SEARCH 的实现是在每个结点处采用二叉查找，而不是线性查找。证明无论怎样选择 t(t 为 n 的函数)，这种实现所需的 CPU 时间都为 $O(\lg n)$。

18.2-7 假设磁盘硬件允许我们任意选择磁盘页的大小，但读取磁盘页的时间为 $a+bt$，其中 a 和 b 为规定的常数，t 为确定磁盘页的大小后，B 树的最小度数。请描述如何选择 t 以(近似地)最小化 B 树的查找时间。对 $a=5$ 毫秒和 $b=10$ 微秒，请给出 t 的一个最优值。

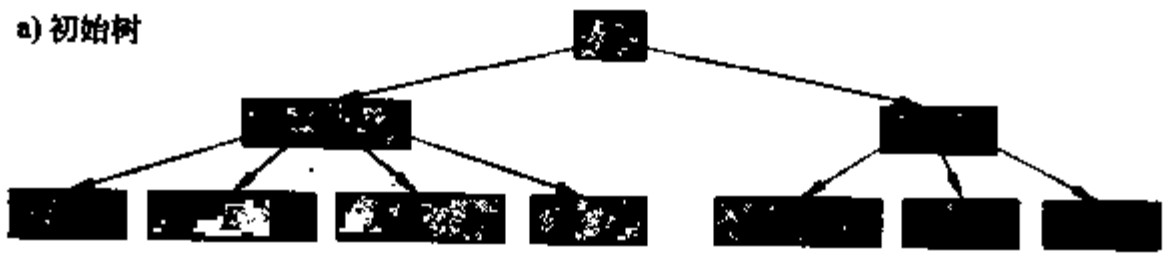
18.3　从 B 树中删除关键字

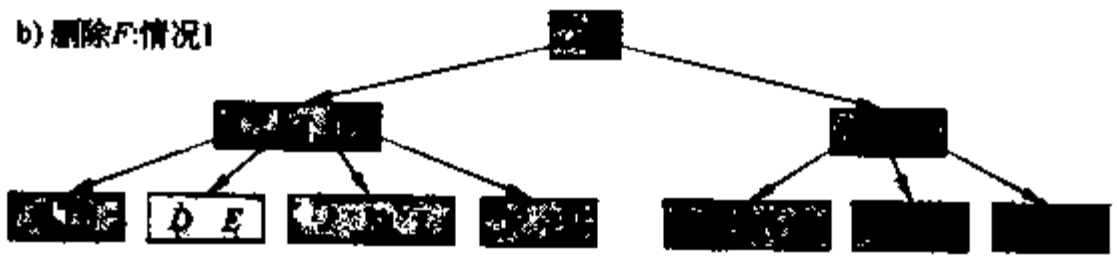
B 树上的删除操作与插入操作类似，只是稍微有点复杂，因为一个关键字能够从任意一个结点中删除，而不只是可以从叶子中删除。此外，从一个内部结点中删除关键字时，需要重新安排这个结点的子女。如同在插入中一样，必须防止因删除操作而导致树的结构违反 B 树性质。就像我们必须保证一个结点不会因为插入而变得太大一样，必须保证一个结点不会在删除期间变得太小(只有根结点除外，它允许有比最小关键字个数 $t-1$ 还少的关键字数，但也不允许有多于最大关键字个数 $2t-1$)。在一个简单的插入算法中，当要插入关键字的路径上的结点是满的时候，可能需要回溯；与此类似，在一个简单的删除算法中，当要删除的关键字的路径上的结点(不是根)有最小的关键字个数时，也可能需要回溯。

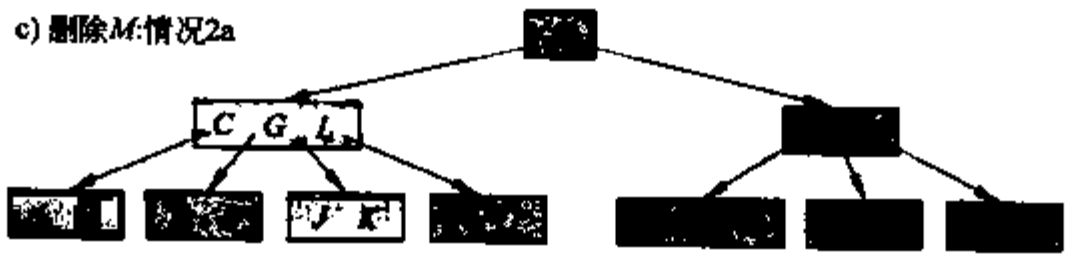
假设用过程 B-TREE-DELETE 从以 x 为根的子树中删除关键字 k。这个过程的结构保证了无论在何时，对结点 x 递归调用 B-TREE-DELETE 后，x 的关键字个数都至少等于最小度数 t。注意这个条件要求比通常的 B 树中最少的关键字数多 1 个的关键字，使得有时在递归降至某结点的一个子结点之前，一个关键字必须移到那个子结点内。这个加强的条件允许我们在一趟下降的过程中，就可以将一个关键字从树中删除，无需任何回溯(只有一个例外，后面我们将解释)。下面对 B 树上删除操作各步骤的规定应当这样来理解，如果根结点 x 成为一个不含任何关键字的内结点(这种

449 情况可能发生在下面的情况 2c 和情况 3b，则 x 就要被删除，而 x 的仅有的孩子 $c_1[x]$ 就成为树的新根，从而树的高度降低了 1，同时也保证了树根必须包含至少一个关键字(除非树是空的)的性质。

下面，我们来大致描述一下删除操作的工作过程，而不是给出其伪代码。图 18-8 给出了从 B 树中删除关键字的各种情况。







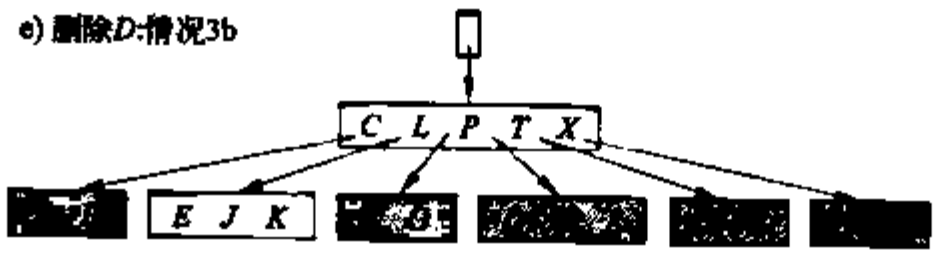


图 18-8　从一棵 B 树中删除关键字。这个 B 树的最小度数 $t=3$，因此一个结点(不是根)包含的关键字不能少于两个。被修改了的结点都加以浅阴影。a)图 18-7e 中的 B 树。b)删除 F。这是情况 1：对一个叶结点中关键字的简单删除。c)删除 M。这是情况 2a：M 的前驱 L 被提升以取代 M 的位置。d)删除 G。这是情况 2c：G 下降以构成结点 $DEGJK$，然后再将 G 从这个叶结点中删除(情况 1)。e)删除 D。这是情况 3b：递归不能降至结点 CL，因为它仅有两个关键字，故 P 下降并与 CL 和 TX 合并以构成 $CLPTX$；然后，将 D 从一个叶结点中删除(情况 1)。e')在 d)之后，根结点被删除，树的高度减小了 1。f)删除 B。这是情况 3a：移动 C 以填补 B 的位置，移动 E 以填补 C 的位置

1)如果关键字 k 在结点 x 中而且 x 是个叶结点，则从 x 中删除 k。

2)如果关键字 k 在结点 x 中而且 x 是个内结点，则作如下操作。

a)如果结点 x 中前于 k 的子结点 y 包含至少 t 个关键字，则找出 k 在以 y 为根的子树中的前驱 k'。递归地删除 k'，并在 x 中用 k' 取代 k。(找到 k' 并删除它可在沿树下降的一趟过程中完成。)

b)对称地，如果结点 x 中位于 k 之后的子结点 z 包含至少 t 个关键字，则找出 k 在以 z 为根的子树中的后继 k'。递归地删除 k'，并在 x 中用 k' 取代 k。(找到 k' 并删除它可在沿树下降的一趟过程中完成。)

c)否则，如果 y 和 z 都只有 $t-1$ 个关键字，则将 k 和 z 中所有关键字合并进 y，使得 x 失去 k 和指向 z 的指针，这使 y 包含 $2t-1$ 个关键字。然后，释放 z 并将 k 从 y 中递归删除。

450 ~ 451

3)如果关键字 k 不在内结点 x 中，则确定必包含 k 的正确的子树的根 $c_i[x]$(如果 k 确实在树中的话)。如果 $c_i[x]$ 只有 $t-1$ 个关键字，执行步骤 3a 或 3b 以保证我们降至一个包含至少 t 个关键字的结点。然后，通过对 x 的某个合适的子结点递归而结束。

a)如果 $c_i[x]$ 只包含 $t-1$ 个关键字，但它的一个相邻兄弟包含至少 t 个关键字，则将 x 中的某一个关键字降至 $c_i[x]$ 中，将 $c_i[x]$ 的相邻左兄弟或右兄弟中的某一关键字升至 x，将该兄弟中合适的子女指针移到 $c_i[x]$ 中，这样使得 $c_i[x]$ 增加一个额外的关键字。

b)如果 $c_i[x]$ 以及 $c_i[x]$ 的所有相邻兄弟都包含 $t-1$ 个关键字，则将 $c_i[x]$ 与一个兄弟合并，即将 x 的一个关键字移至新合并的结点，使之成为该结点的中间关键字。

因为一棵 B 树中的大部分关键字都在叶结点中，我们可以预期在实际中，删除操作主要是用于从叶结点中删除关键字。这样 B-TREE-DELETE 过程只要沿树下降一趟即可，而无需任何回溯。但是，当删除某内结点中的一个关键字时，该程序也沿树下降一趟，但可能还要返回删除了关键字的那个结点，以用其前驱或后继来取代被删除的关键字(情况 2a 和情况 2b)。

这个过程虽然看似复杂，但对一棵高度为 h 的 B 树，它只需 $O(h)$ 次磁盘存取操作，因为在递归调用该过程之间，仅需 $O(1)$ 次对 DISK-READ 和 DISK-WRITE 的调用。所需 CPU 时间为 $O(th)=O(t\log_t n)$。

练习

18.3-1　请说明按顺序从图 18-8f 中删除 C，P 和 V 后的结果。

18.3-2　请写出 B-TREE-DELETE 的伪代码。

思考题

18-1　辅存上的栈

考虑在一个有着相对少量的快速主存，但有着相对大量的慢速磁盘存储空间的计算机上实现一个栈的问题。操作 PUSH 和 POP 支持的对象为单个字的值。我们所希望支持的栈可以增长得很大而无法装入主存，因而它的大部分都要放在磁盘上。

一种简单但是低效的栈实现方法是将整个栈放在磁盘上。在主存中保持一个栈指针，它指向栈顶元素的磁盘地址。如果该指针的值为 p，则栈顶元素是磁盘的 $\lfloor p/m \rfloor$ 页上的第 $(p \bmod m)$ 个字，此处 m 为每页所含的字数。

452

为实现 PUSH 操作，我们增加一个栈指针，从磁盘中将适当的页读到主存中后，复制要被压入该页上适当字里的元素，最后将该页写回到磁盘上。POP 操作的实现是类似的。先减小栈指针，从磁盘上读入所需的页，再返回栈顶元素。在这种情况下，无需将读

人的页写回，因为它没有被修改。

因为磁盘操作的代价相对较高，我们计算实现的两部分代价：总的磁盘存取次数和总的 CPU 时间。任何对一个包含 m 个字的页面的磁盘存取，都会引起一次磁盘存取和 $\Theta(m)$的 CPU 时间。

a)从渐近意义上来看，在最坏情况下，利用这种简单实现的 n 个栈操作的磁盘存取次数是多少？n 个栈操作的 CPU 时间是多少？(用 m 和 n 来表示这个问题以及后面几个问题的答案。)

现在考虑栈的另一种实现，即始终将栈的一页放在主存中。(我们还维护少量的主存来记录当前哪一页在主存中。)只有相关的磁盘页面驻留在主存时，我们才能执行一次栈操作。如果需要的话，当前主存中的页可被写回磁盘，并从磁盘上向主存中读入新的一页。如果相关的磁盘页已在主存中，则无需任何磁盘存取。

b)n 个 PUSH 操作所需的最坏情况磁盘存取次数是多少？CPU 时间是多少？

c)n 个栈操作所需的最坏情况磁盘存取次数是多少？CPU 时间是多少？

假设一种栈的实现方式是保持栈的两个页在主存中(除了小量的存储用来记录哪些页在主存中之外)。

d)请描述如何管理栈页使得任何栈操作的平摊磁盘存取次数为 $O(1/m)$，平摊 CPU 时间为 $O(1)$。

18-2 连接与分裂 2-3-4 树

*连接*操作的输入为两个动态集合 S'和 S''，以及一个元素 x，使得对任何 $x' \in S'$，$x'' \in S''$，有 $key[x'] < key[x] < key[x'']$。它返回一个集合 $S = S' \cup \{x\} \cup S''$。分裂操作有点像一个"逆"连接操作：给定一个动态集 S 和一个元素 $x \in S$，它创建一个集合 S'，其中包含了 $S - \{x\}$中所有关键字小于 $key[x]$的元素；同时创建了一个集合 S''，其中包含了 $S - \{x\}$中所有关键字大于 $key[x]$的元素。在这个问题中，我们来讨论如何在 2-3-4 树上实现这些操作。为方便起见，假定所有元素都只包含关键字，而且所有的关键字都不相同。 453

a)对 2-3-4 树中的每个结点 x，说明如何将以 x 为根的子树的高度作为一个域 $height[x]$来维护。要注意所给出的实现不应影响查找、插入和删除操作的渐近运行时间。

b)说明如何实现联结操作。给定两个 2-3-4 树 T'和 T'' 以及一个关键字 k，连接操作的运行时间应是 $O(1 + |h' - h''|)$，其中 h'和 h''分别为 T'和 T''的高度。

c)考试从一棵 2-3-4 树 T 的根至一个给定的关键字 k 的路径 p，包含 T 中小于 k 的关键字集合 S'，以及包含 T 中大于 k 的关键字的集合 S''。证明 p 将 S'分裂成一个树的集合 $\{T'_0, T'_1, \cdots, T'_m\}$和一个关键字的集合 $\{k'_1, k'_2, \cdots, k'_m\}$，此处对每个 $i = 1, 2, \cdots, m$，以及任何关键字 $y \in T'_{i-1}$和 $z \in T'_i$，都有 $y < k'_i < z$。T'_{i-1}和 T'_i 的高度之间有什么关系？请说明 p 是如何将 S''分裂成树的集合和关键字的集合的。

d)请说明如何实现 T 上的分裂操作。利用联结操作将 S'中的关键字拼成一棵 2-3-4 树 T'，将 S''中的关键字拼成一棵 2-3-4 树 T''。分裂操作的运行时间应为 $O(\lg n)$，此处 n 为 T 中的关键字个数。(提示：连接的代价应该是套选的。)

本章注记

Knuth [185]，Aho，Hopcroft 和 Ullman [5]，以及 Sedgewick [269]给出了平衡树方案和 B 树的进一步讨论。Comer [66]提供了 B 树的一个综述。Guibas 和 Sedgewick [135]讨论了包括红黑树和 2-3-4 树在内的各种平衡树方案之间的关系。

在 1970 年，J. E. Hopcroft 发明了 2-3 树，它是 B 树和 2-3-4 树的前身，其中每个内结点有两个或三个子女。在 1972 年 Bayer 和 McCreight [32]提出了 B 树；他们没有解释为什么要取这个名字。

Bender，Demaine 和 Farach-Colton [37] 研究了在存储器层次的影响下，如何让 B 树操作有效地执行。他们提出了一个缓存忘却(cache-oblivious)算法，该算法在不需要明确知道存储器层
454 次中数据传输规模的情况下，也可以高效地工作。

第19章 二 项 堆

本章和第20章要介绍称为可合并堆(mergeable heap)的数据结构，这些数据结构支持下面五种操作：

MAKE-HEAP()：创建并返回一个不包含任何元素的新堆。

INSERT(H，x)：将结点 x(其关键字域中已填入了内容)插入堆 H 中。

MINIMUM(H)：返回一个指向堆 H 中包含最小关键字的结点的指针。

EXTRACT-MIN(H)：将堆 H 中包含最小关键字的结点删除，并返回一个指向该结点的指针。

UNION(H_1，H_2)：创建并返回一个包含堆 H_1 和 H_2 中所有结点的新堆。同时，H_1 和 H_2 被这个操作"删除"。

另外，这两章里的数据结构还支持下面两种操作：

DECREASE-KEY(H，x，k)：将新关键字值 k(假定它不大于当前的关键字值)赋给堆 H 中的结点 x。[⊖]

DELETE(H，x)：从堆 H 中删除结点 x。

如图19-1中的表格所示，如果不需要UNION操作，则普通的二叉堆(如第6章)堆排序中用到的性能就很好。在一个二叉堆上，非UNION操作的最坏情况运行时间为 $O(\lg n)$(或更好)。但是，如果某个应用中一定要用UNION操作，则二叉堆就不能令人满意了。UNION操作先将包含两个要合并的堆的数组并置，然后运行MIN-HEAPIFY(见练习6.2-2)；在最坏情况下，UNION操作的运行时间为 $\Theta(n)$。

过程	二叉堆(最坏情况)	二项堆(最坏情况)	斐波那契堆(平摊)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Omega(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Omega(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Omega(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

图19-1 可合并堆的三种实现中各操作的运行时间。在执行某一操作时堆中的项目数用 n 表示

在这一章中，我们要讨论"二项堆"(binomial heap)，其最坏情况时间界也如图19-1所示。特别地，UNION操作只要 $O(\lg n)$ 时间就可完成包含 n 个元素的两个二项堆的合并。

在第20章中，我们将讨论斐波那契堆，对某些操作它具有更好的时间界。但请注意，图19-1中斐波那契堆的运行时间是平摊时间界，而不是每个操作的最坏情况时间界。

本章略去在插入前分配结点和删除后释放结点的问题。我们假定这些细节由调用堆过程的程序来处理。

⊖ 正如第五部分导言中所提到的，默认的可合并堆为可合并最小堆。因此，操作MINIMUM，EXTRACT-MIN和DECREASE-KEY都能够应用。或者，也可以定义一个具有操作MAXIMUM，EXTRACT-MAX和INCREASE-KEY的可合并最大堆。

从对 SEARCH 操作的支持方面来看，二叉堆、二项堆和斐波那契堆都是低效的。在这些结构中，要找到一个包含给定关键字的结点可能花一些时间。因为这个原因，在 DECREASE-KEY 和 DELETE 等涉及一个给定结点的操作需要一个指向该结点的指针作为输入的一部分。如 6.5 节关于优先队列的讨论中一样，当在一个应用中使用可合并堆时，我们通常将对应应用对象的柄(handle)存入每个可合并堆的元素中，同时也将对应可合并堆元素的柄存入每个应用对象。这些柄的确切性质取决于具体的应用及其实现。

19.1 节先定义二项树，再定义二项堆。另外，还要介绍二项堆的一种特别表示。19.2 节说
456 明如何以图 19-1 给出的时间界实现二项堆上的操作。

19.1　二项树与二项堆

一个二项堆由一组二项树所构成，故这一节先定义二项树并证明它们的一些关键性质，然后定义二项堆，并说明如何表示它们。

19.1.1　二项树

二项树 B_k 是一种递归定义的有序树(见 B.5.2 节)。如图 19-2a 所示，二项树 B_0 只包含一个结点。二项树 B_k 由两棵二项树 B_{k-1} 连接而成：其中一棵树的根是另一棵树的根的最左孩子。图 19.2b显示从 B_0 到 B_4 的二项树。

下面的引理给出二项树的一些性质。

引理 19.1(二项树的性质)　二项树 B_k 具有以下的性质：

1)共有 2^k 个结点，

2)树的高度为 k，

3)在深度 i 处恰有 $\binom{k}{i}$ 个结点，其中 $i=0, 1, 2, \cdots, k$，

4)根的度数为 k，它大于任何其他结点的度数；并且，如果根的子女从左到右编号为 $k-1, k-2, \cdots, 0$，子女 i 是子树 B_i 的根。

证明：对 k 进行归纳。对每一个性质，基都是二项树 B_0。验证每个性质对 B_0 成立是很容易的。

对归纳步骤，假设本引理对 B_{k-1} 成立。

1)二项树 B_k 包含两个 B_{k-1}，故 B_k 有 $2^{k-1}+2^{k-1}=2^k$ 个结点。

2)根据由两个 B_{k-1} 连接成 B_k 的方式，可知 B_k 结点的最大深度比 B_{k-1} 中的最大深度大 1。根据归纳假设，这个最大深度为 $(k-1)+1=k$。

3)设 $D(k, i)$ 表示二项树 B_k 在深度 i 处的结点数。因为 B_k 由两个 B_{k-1} 连接而成，故 B_{k-1}
457 中深度 i 处的某个结点在 B_k 中深度 i 和 $i+1$ 处各出现一次。换句话说，在 B_k 中，深度 i 处的结点个数为 B_{k-1} 中深度 i 处结点个数与 B_{k-1} 中深度 $i-1$ 处结点个数之和。这样，

$$\begin{aligned} D(k,i) &= D(k-1,i)+D(k-1,i-1) \quad \text{(根据递归假设)} \\ &= \binom{k-1}{i}+\binom{k-1}{i-1} \quad \text{(根据练习 C.1-7)} \\ &= \binom{k}{i} \end{aligned}$$

4)在 B_k 中，度数大于在 B_{k-1} 中的度数的唯一结点就是根，它在 B_k 中的子女比在 B_{k-1} 中多一个。因为 B_{k-1} 的根的度数为 $k-1$，故 B_k 的根的度数为 k。根据归纳假设，同时也像图 19-2c 所示的那样，从左至右看，B_{k-1} 的根的各子女分别为 B_{k-2}，B_{k-3}，…，B_0 的根。所以，当将 B_{k-1} 与另一 B_{k-1} 连接时，所得的根的子女为 B_{k-1}，B_{k-2}，…，B_0 的根。 ■

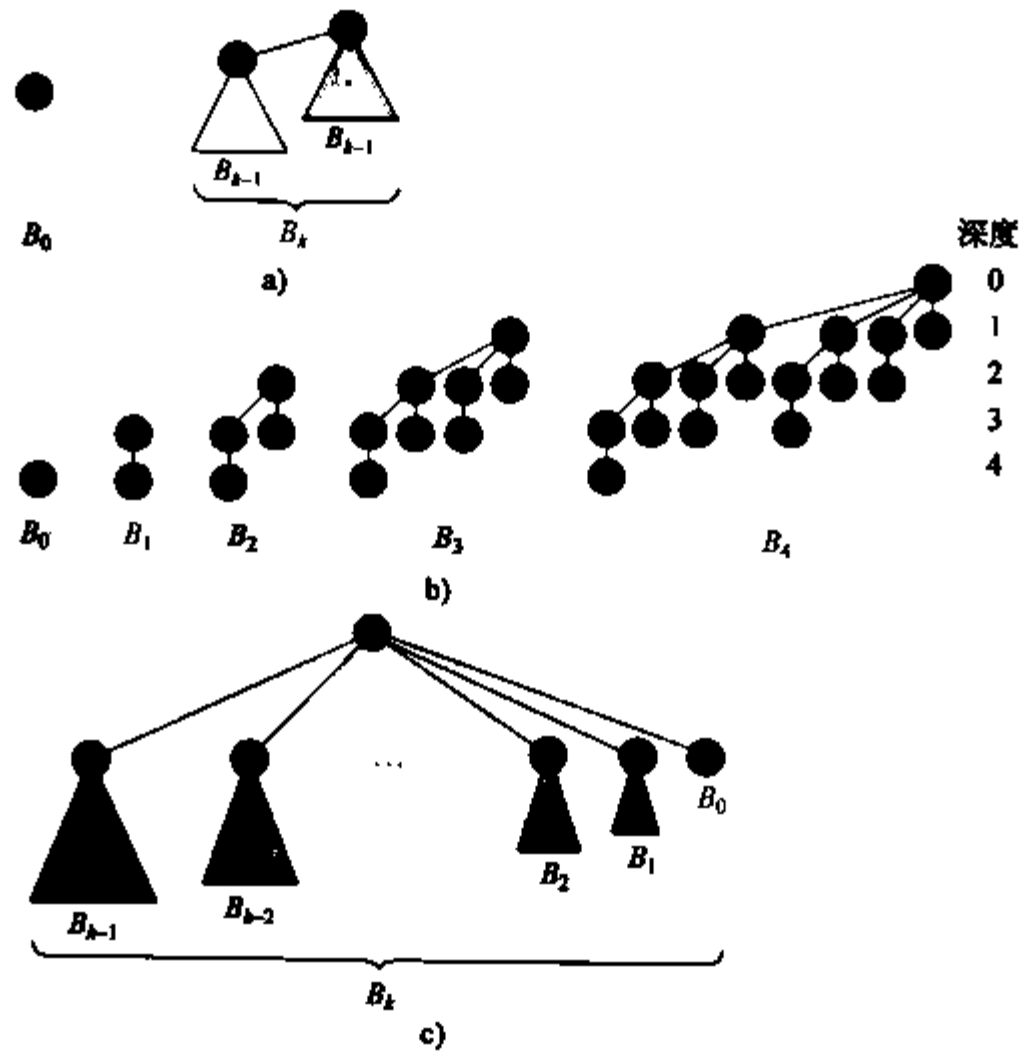


图 19-2　a)二项树 B_k 的递归定义，三角形表示有根的子树。b)二项树 B_0 至 B_4，B_4 中示出了各结点的深度。c)以另一种方式来看二项树 B_k

推论 19.2　在一棵包含 n 个结点的二项树中，任意结点的最大度数为 $\lg n$。

证明：由引理 19.1 的性质 1 和性质 4 直接可得。■

术语"二项树"是从引理 19.1 的性质 3 而来，因为项 $\binom{k}{i}$ 为二项系数。练习 19.1-3 对此术语作了进一步的说明。

19.1.2　二项堆

二项堆 H 由一组满足下面的二项堆性质的二项树组成。

1)H 中的每个二项树遵循最小堆性质：结点的关键字大于或等于其父结点的关键字。我们说这种树是最小堆有序的。

2)对任意非负整数 k，在 H 中至多有一棵二项树的根具有度数 k。

第一个性质告诉我们，在一棵最小堆有序的二项树中，其根包含了树中最小的关键字。

根据第二个性质可知，在包含 n 个结点的二项堆 H 中，包含至多 $\lfloor \lg n \rfloor + 1$ 棵二项树。为搞清楚这一点，注意 n 的二进制表示中有 $\lfloor \lg n \rfloor + 1$ 位，例如 $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \cdots, b_0 \rangle$，从而 $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$。所以，根据引理 19.1 中的性质 1 可知，二项树 B_i 出现于 H 中，当且仅当位 $b_i = 1$。这样，二项堆 H 包含至多 $\lfloor \lg n \rfloor + 1$ 棵二项树。

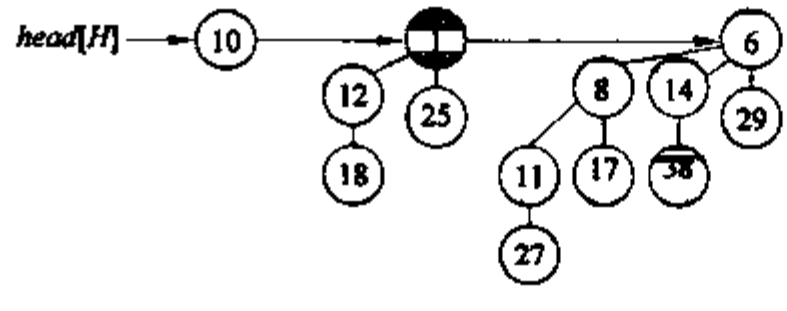
图 19-3a 示出了包含 13 个结点的二项堆 H。13 的二进制表示为〈1101〉，故 H 包含了最小堆有序二项树 B_3，B_2 和 B_0，它们分别有 8，4 和 1 个结点，即共有 13 个结点。

二项堆的表示

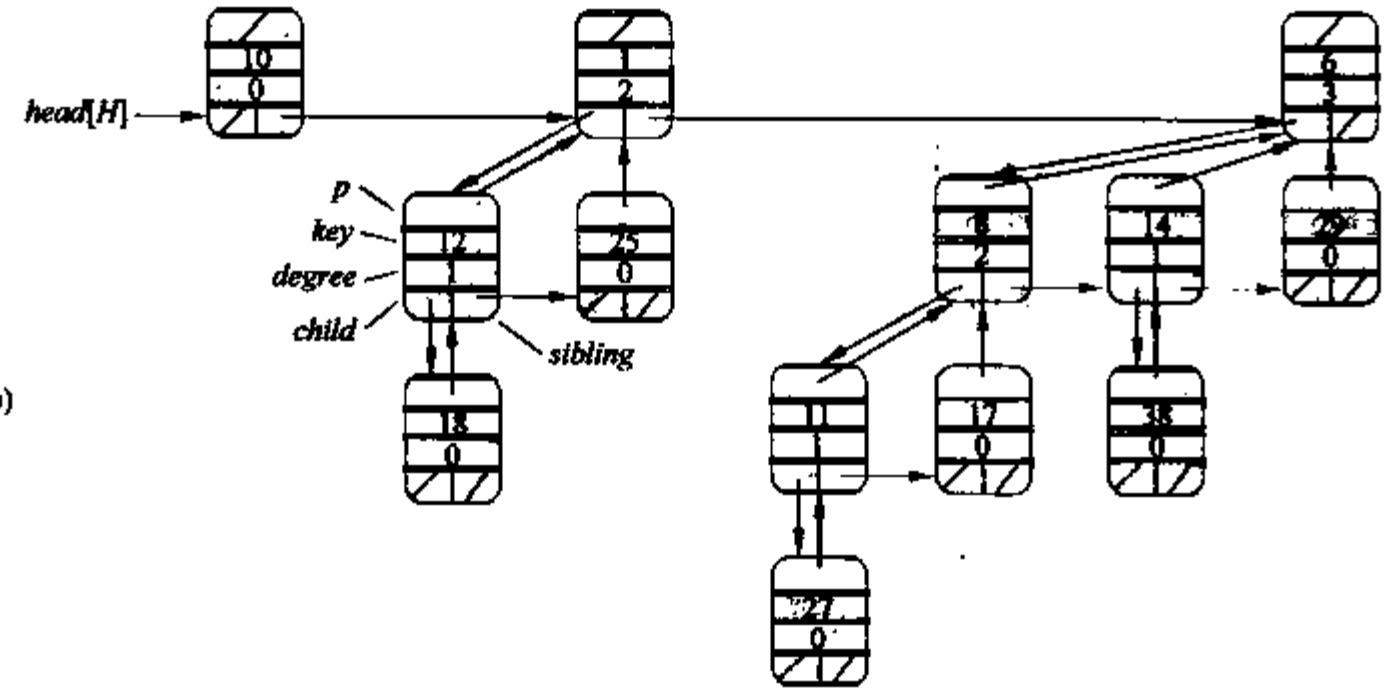
如图 19-3b 所示，二项堆中的每棵二项树都按 10.4 节中左孩子，右兄弟的表示方式存储。在每个结点中，都有一个关键字域及其他依应用要求而定的卫星数据。另外，每个结点 x 还包含了指向其父结点的指针 $p[x]$，指向其最左孩子的指针 $child[x]$，以及指向 x 的紧右兄弟的指针 $sibling[x]$。如果结点 x 是根，则 $p[x]=$NIL。如果结点 x 没有子女，则 $child[x]=$NIL。如果 x 是其父结点的最右孩子，则 $sibling[x]=$NIL。每个结点 x 都包含域 $degree[x]$，即 x 的子女个数。

459 如图 19-3 所示，一个二项堆中的各二项树的根被组织成一个链表，我们称之为根表。在遍历根表时，各根的度数是严格递增的。根据第二个二项堆性质，在一个 n 结点的二项堆中各根的度数构成了$\{0, 1, \cdots, \lfloor \lg n \rfloor\}$的一个子集。对根结点与非根结点来说，*sibling* 域的含义是不同的。如果 x 为根，则 $sibling[x]$指向根表中下一个根(像通常一样，如果 x 为根表中最后一个根，则 $sibling[x]=$NIL)。

一个给定的二项堆 H 可通过域 $head[H]$来存取接；这个域即指向 H 的根表中第一个根的指针。如果二项堆 H 中没有元素，则 $head[H]=$NIL。



a)



b)

图 19-3　一个包含 13 个结点的二项堆。a)一个二项堆包含了二项树 B_0，B_2 和 B_3，它们分别有 8，4 和 1 个结点，即共有 13 个结点。由于每棵二项树都是最小堆有序的，所以任意结点的关键字都不小于其父结点的关键字。图中还示出了根表，它是一个按根的度数递增排序的链表。b)二项堆 H 的一个更具体的表示。每棵二项树按左孩子、右兄弟表示方式存储，每个结点

460 存储自身的度数

练习

19.1-1　假设 x 为一个二项堆中，某棵二项树中的一个结点，并假定 $sibling[x]\neq$NIL。如果 x 不是根，则 $degree[sibling[x]]$与 $degree[x]$相比怎样？如果 x 是个根呢？

19.1-2　如果 x 是二项堆的某棵二项树的一个非根结点，$degree[p[x]]$与 $degree[x]$相比怎样？

19.1-3　如图 19-4 所示，假设按后序遍历顺序，将二项树 B_k 中的结点标为二进制形式。考虑深度 i 处标为 l 的一个结点 x，且设 $j=k-i$。证明：在 x 的二进制表示中共有 j 个 1。恰包含 j 个 1 的二进制 k 串共有多少？证明 x 的度数与 l 的二进制表示中，最右 0 的右边的 1 的个数相同。

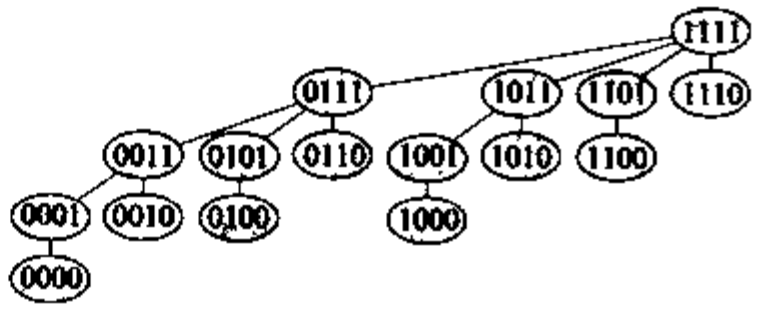


图 19-4　二项树 B_4，其各结点按后序遍历次序标以一个二进制数

19.2　对二项堆的操作

这一节里，要介绍如何在图 19-1 中给出的时间界内，执行对二项堆的各种操作。我们将只给出上界，下界留作练习 19.2-10。

创建一个新二项堆

为了构造一个空的二项堆，过程 MAKE-BINOMIAL-HEAP 分配并返回一个对象 H，且 $head[H]=$NIL。该过程的运行时间为 $\Theta(1)$。

461

寻找最小关键字

过程 BINOMIAL-HEAP-MINIMUM 返回一个指针，它指向包含 n 个结点的二项堆 H 中具有最小关键字的结点。这个实现假设没有一个关键字为∞(见练习 19.2-5)。

```
BINOMIAL-HEAP-MINIMUM(H)
1  y ← NIL
2  x ← head[H]
3  min ← ∞
4  while x≠NIL
5      do if key[x]<min
6            then min ← key[x]
7                 y ← x
8         x ← sibling[x]
9  return y
```

因为一个二项堆是最小堆有序的，故最小关键字必在根结点中。过程 BINOMIAL-HEAP MINIMUM 检查所有的根(至多有 $\lfloor \lg n \rfloor + 1$)，将当前最小者存于 min 中，而将指向当前最小者的指针存于 y 之中。当对图 19-3 中二项堆调用时，BINOMIAL-HEAP-MINIMUM 返回一个指向具有关键字 1 的结点的指针。

因为至多要检查 $\lfloor \lg n \rfloor + 1$ 个根，故 BINOMIAL-HEAP-MINIMUM 的运行时间为 $O(\lg n)$。

合并两个二项堆

合并两个二项堆的操作可用作后面大部分操作的一个子程序。过程 BINOMIAL-HEAP-UNION 反复连接根结点的度数相同的各二项树。在下面的过程，将以结点 y 为根的 B_{k-1} 树与以结点 z 为根的 B_{k-1} 树连接起来；亦即，它使得 z 成为 y 的父结点，并成为一棵 B_k 树的根。

```
BINOMIAL-LINK(y, z)
1  p[y] ← z
2  sibling[y] ← child[z]
3  child[z] ← y
4  degree[z] ← degree[z]+1
```

过程 BINOMIAL-LINK 在 $O(1)$时间内，使得结点 y 成为结点 z 的子女链表的新头。这个过程之所以能正确地运行，是因为每棵二项树的左孩子、右兄弟表示与树的排序性质正好匹配：在
462 一棵 B_k 树中，根的最左孩子是一棵 B_{k-1} 树的根。

下面的过程合并二项堆 H_1 和 H_2，并返回结果堆。在合并过程中，它也同时破坏了 H_1 和 H_2 的表示。除了 BINOMIAL-LINK 之外，这个过程还使用了一个辅助过程 BINOMIAL-HEAP-MERGE，来将 H_1 和 H_2 的根表合并成一个按度数的单调递增次序排列的链表。BINOMIAL-HEAP-MERGE(其伪代码留作练习 19.2-1)与 2.3.1 节中的 MERGE 过程是类似的。

```
BINOMIAL-HEAP-UNION(H1, H2)
 1  H ← MAKE-BINOMIAL-HEAP()
 2  head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
 3  free the objects H1 and H2 but not the lists they point to
 4  if head[H]=NIL
 5    then return H
 6  prev-x ← NIL
 7  x ← head[H]
 8  next-x ← sibling[x]
 9  while next-x≠NIL
10      do if(degree[x]≠degree[next-x])or
              (sibling[next-x]≠NIL and degree[sibling[next-x]]=degree[x])
11           then prev-x ← x                                   ▷ Cases 1 and 2
12                x ← next-x                                   ▷ Cases 1 and 2
13           else if key[x]≤key[next-x]
14                  then sibling[x] ← sibling[next-x]          ▷ Case 3
15                       BINOMIAL-LINK(next-x, x)              ▷ Case 3
16           else if prev-x=NIL                                ▷ Case 4
17                  then head[H] ← next-x                      ▷ Case 4
18                  else sibling[prev-x] ← next-x              ▷ Case 4
19                BINOMIAL-LINK(x, next-x)                     ▷ Case 4
20                x ← next-x                                   ▷ Case 4
21        next-x ← sibling[x]
22  return H
```

图 19-5 示出了 BINOMIAL-HEAP-UNION 的一个例子，其中出现了代码中给出的所有四种情况。

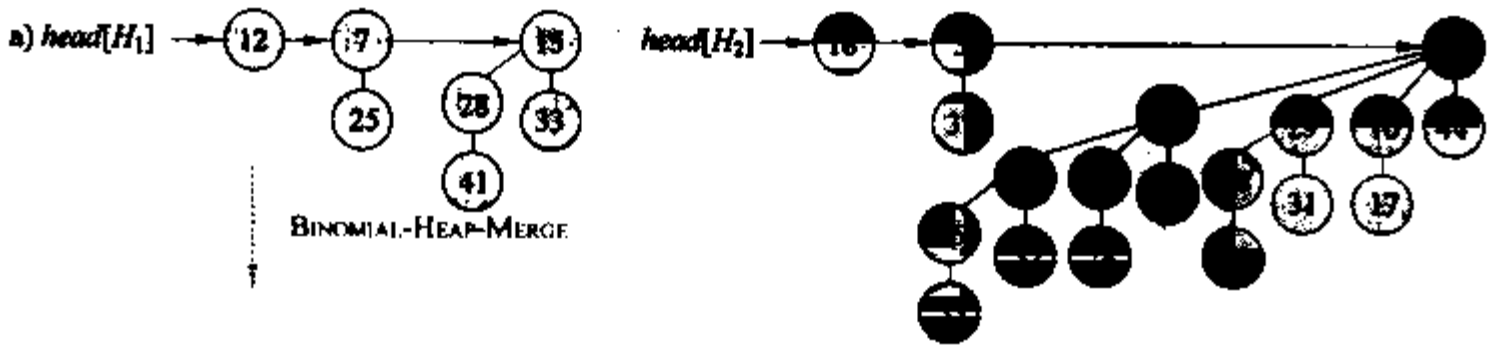


图 19-5　BINOMIAL-HEAP-UNION 的执行过程。a)二项堆 H_1 和 H_2。b)二项堆 H 是 BINOMIAL-HEAP-MERGE(H_1, H_2)的输出。开始时，x 是 H 的根表上的第一个根。因为 x 和 $next\text{-}x$ 的度数都为 0，且 $key[x]<key[next\text{-}x]$，这与情况 3 对应。c)在链接发生后，$x$ 是具有相同度数的三个根中的第一个，这与情况 2 对应。d)在根表中所有指针都向下移动一个位置后，情况 4 适用，因为 x 是两个具有相同度数的根中的第一个。e)在发生链接后，情况 3 适用。f)在另一次链接后，情况 1 适用，因为 x 的度数为 3，$next\text{-}x$ 的度数为 4。While 循环的这一次迭代也是最后一次，因为在根表中的各指针向下移动一个位置后，$next\text{-}x=$NIL

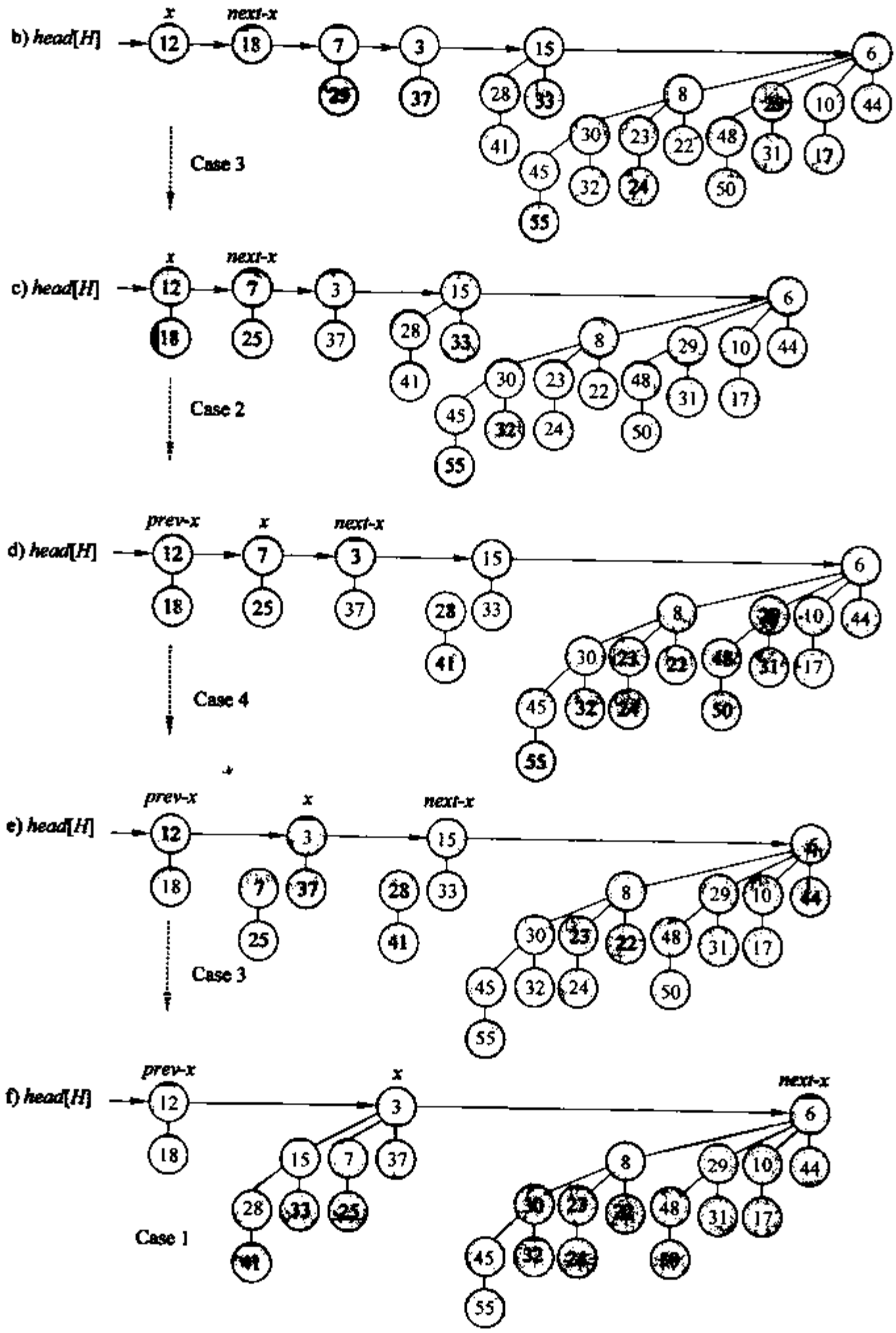


图 19-5　(续)

BINOMIL-HEAP-UNION 过程有两个阶段。第一个阶段中执行对 BINOMIAL-HEAP-MERGE 的调用，将二项堆 H_1 和 H_2 的根表合并成一个链表 H，它按度数排序成单调递增次序。然而，对于每一个度数值，可能有两个根(但不可能更多了)与其对应，所以第二阶段将相等

度数的根连接起来，直到每个度数至多有一个根时为止。因为链表 H 是按度数排序的，所以可以很快地做各种链表操作。

具体说来，该过程的工作过程是这样的。第 1～3 行先将二项堆 H_1 和 H_2 的根表合并成一个根表 H。H_1 和 H_2 的根表是按严格递增的度数来排序的，而 BINOMIAL-HEAP-MERGE 返回一个按单调递增度数排序的根表 H。如果 H_1 和 H_2 的根表共有 m 个根，则 BINOMIAL-HEAP-MERGE 可在 $O(m)$ 时间内，反复检查两个根表头上的根，并将度数较低的根添加至输出根表中，同时将其从输入根表中去掉。

过程 BINOMIAL-HEAP-UNION 接着对指向 H 的根表的某些指针进行初始化。首先，在第 4～5 行中，如果正好是在合并两个空二项堆，则返回。那么，从第 6 行开始，H 至少有一个根。在整个过程中，一共维护了三个指向根表的指针：

- x 指向目前被检查的根
- *prev-x* 指向根表中 x 前一个根，即 $sibling[prev\text{-}x]=x$（因为初始时 x 没有前驱，我们开始将 *prev-x* 置为 NIL）
- *next-x* 指向根表中 x 的后一个根，即 $sibling[x]=next\text{-}x$

463～465

开始时，对一个给定度数的根表 H 上至多有两个根：因为 H_1 和 H_2 为两个二项堆，对一个给定的度数，它们都各只含有一个根。另外，BINOMIAL-HEAP-MERGE 保证了如果 H 中的两个根具有相同的度数，则在根表中是邻居。

实际上，在 BINOMIAL-HEAP-UNION 的执行过程中，在有的时刻，H 根表可能会出现三个根具有相同的度数。待会儿我们将会看到这种情况是怎么发生的。在第 9～21 行中的 while 循环的每一次迭代中，要根据 x 和 *next-x*（甚至可能 $sibling[next\text{-}x]$）的度数，来确定是否把两者连接起来。该循环的一种不变式是每进入循环体时，x 和 *next-x* 都是非 NIL 的（具体的循环不变式可参见练习 19.2-4）。

情况 1，如图 19-6a 所示，发生的条件是 $degree[x]\neq degree[next\text{-}x]$；亦即，$x$ 为 B_k 树的根，而 *next-x* 为一棵 B_l 树的根，且 $l>k$。第 11～12 行处理了这种情况。我们并不连接 x 和 *next-x*，只是将指针指向表中的下一个位置。在第 21 行中更新 *next-x*，使之指向新结点 x 之后的结点。这个处理对所有情况都是一样的。

情况 2，如图 19-6b 所示，当 x 为具有相同度数的三个根中的第一个时发生；亦即，当 $degree[x]=degree[next\text{-}x]=degree[sibling[next\text{-}x]]$ 时发生。对这种情况的处理与情况 1 相同：将指针又一次移向表中下一个位置。下一次迭代将执行情况 3 或情况 4，把三个相等度数的根中第二个和第三个联合起来。第 10 行测试情况 1 和情况 2，第 11～12 行对两种情况都作了处理。

情况 3 和情况 4 当 x 为具有相同度数的两个根中的第一个时发生；亦即，当 $degree[x]=degree[next\text{-}x]\neq degree[sibling[next\text{-}x]]$ 时发生。这些情况在任一次迭代中都可能发生，但是其中之一总会紧随情况 2 而发生。在情况 3 和情况 4 中，我们将 x 与 *next-x* 相连接。这两种情况根据 x 和 *next-x* 哪一个具有更小的关键字来区分；这个区分条件决定在这两个根连接后哪一个结点将成为根。

在情况 3 中，如图 19-6c 所示，$key[x]\leqslant key[next\text{-}x]$，故将 *next-x* 连接到 x 上。第 14 行将 *next-x* 从根表中去掉，第 15 行使 *next-x* 成为 x 的最左孩子。

在情况 4 中，如图 19-6d 所示，*next-x* 具有更小的关键字，故 x 被连接到 *next-x* 上。第 16～18 行将 x 从根表中去掉；又有两种情况取决于 x 是（第 17 行）还是不是（第 18 行）根表中的第一个根。第 19 行使 x 成为 *next-x* 的最左孩子，第 20 行更新 x 以进入下一轮迭代。

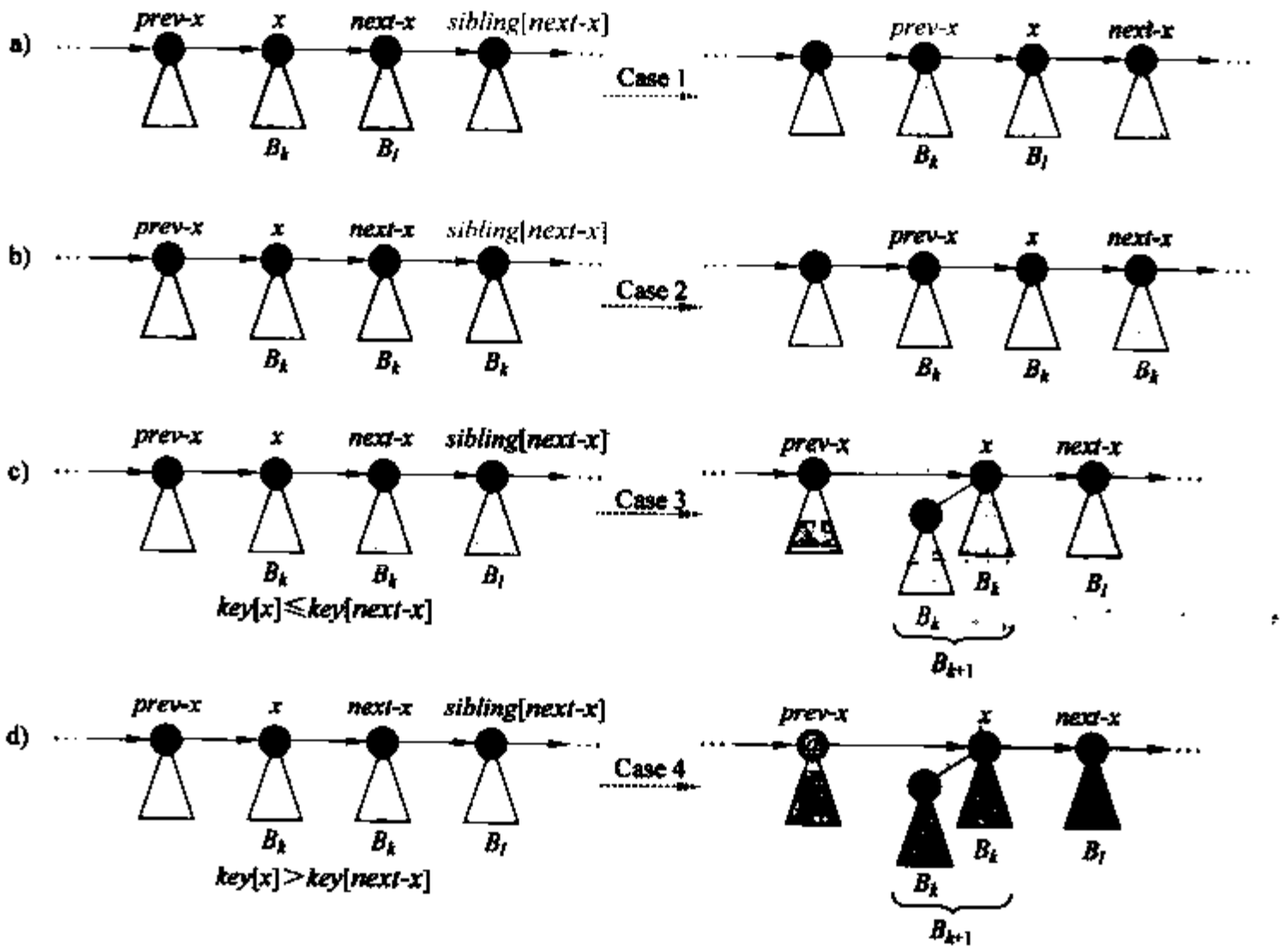


图 19-6　在 BINOMIAL-HEAP-UNION 中发生的四种情况，标记 a，b，c 和 d 仅用于区别所牵涉到的根，它们并不表示这些根的关键字或度数。在每种情况中，x 是一棵 B_k 树的根，且 $l>k$。a)情况 1：$degree[x]\neq degree[next\text{-}x]$，指针移向根表中的下一个位置。b)情况 2：$degree[x]=degree[next\text{-}x]=degree[sibling[next\text{-}x]]$。指针再一次移向根表中的下一个位置，而且下一次迭代将执行情况 3 或情况 4。c)情况 3：$degree[x]=degree[next\text{-}x]\neq degree[sibling[next\text{-}x]]$，且 $key[x]\leqslant key[next\text{-}x]$。将 $next\text{-}x$ 从根表中去掉，并将其连接到 x 上，构造一棵 B_{k+1} 树。d)情况 4：$degree[x]=degree[next\text{-}x]\neq degree[sibling[next\text{-}x]]$，且 $key[next\text{-}x]\leqslant key[x]$。我们将 x 从根表中去掉，并将其连接到 $next\text{-}x$ 上，也构造一棵 B_{k+1} 树

在情况 3 或情况 4 之后，为 while 循环的下一次执行所做的设置是相同的。我们刚刚连接了两个 B_k 树以形成一棵 B_{k+1} 树，即现在 x 所指向的树。在 BINOMIAL-HEAP-MERGE 输出的根表中已有 0，1 或 2 个其他的 B_{k+1} 树，故现在 x 就为根表上 1，2 或 3 棵 B_{k+1} 树中的第一个。如果仅有 x 一个，则在下一轮循环中进入情况 1：$degree[x]\neq degree[next\text{-}x]$。如果 x 是两个中的第一个，则在下一轮循环中进入情况 3 或情况 4。当 x 为三个中的第一个时，才在下一轮循环中进入情况 2。

BINOMIA-HEAP-UNION 的运行时间为 $O(\lg n)$，其中 n 为二项堆 H_1 和 H_2 中总的结点数。设 H_1 包含 n_1 个结点，H_2 包含了 n_2 个结点，故 $n=n_1+n_2$。又因为 H_1 至多包含 $\lfloor \lg n_1 \rfloor +1$ 个根，而 H_2 至多包含 $\lfloor \lg n_2 \rfloor +1$ 个根，故在调用了 BINOMIAL-HEAP-MERGE 后，H 包含至多 $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor +2 \leqslant 2\lfloor \lg n \rfloor +2=O(\lg n)$ 个根。因此，执行 BINOMIAL-HEAP-MERGE 的时间为 $O(\lg n)$。while 循环的每次迭代需要 $O(1)$ 时间，又因为在每次循环中，或者将指针指向 H 根表中的下一位，或从根表中去掉一个根，于是就总共要执行至多 $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor +2$ 次迭代。所以，总的时间为 $O(\lg n)$。

466
~
467

插入一个结点

下面的过程将结点 x 插入二项堆 H 中，假定结点 x 已被分配，且 $key[x]$也已填有内容。

```
BINOMIAL-HEAP-INSERT(H, x)
1  H' ← MAKE-BINOMIAL-HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
4  sibling[x] ← NIL
5  degree[x] ← 0
6  head[H'] ← x
7  H ← BINOMIAL-HEAP-UNION(H, H')
```

这个过程先在 $O(1)$时间内，构造一个只包含一个结点的二项堆 H'，再在 $O(\lg n)$时间内，将其与包含 n 个结点的二项堆 H 合并。对 BINOMIAL-HEAP-UNION 的调用还负责释放临时二项堆 H'。(另一种不调用 BINOMIAL-HEAP-UNION 的直接实现在练习 19.2-8 中给出。)

抽取具有最小关键字的结点

下面的过程从二项堆 H 中抽取具有最小关键字的结点，并返回一个指向该结点的指针。

```
BINOMIAL-HEAP-EXTRACT-MIN(H)
1  find the root x with the minimum key in the root list of H,
       and remove x from the root list of H
2  H' ← MAKE-BINOMIAL-HEAP()
3  reverse the order of the linked list of x's children,
       and set head[H'] to point to the head of the resulting list
4  H ← BINOMIAL-HEAP-UNION(H, H')
5  return x
```

这个过程的工作如图 19-7 所示。输入二项堆 H 如图 19-7a 所示；图 19-7b 示出了第 1 行后的情形：具有最小关键字的根 x 被从 H 的根表中去掉。如果 x 为一棵 B_k 树的根，则根据引理 19.1的性质 4，x 的各子女从左到右分别为 B_{k-1}，B_{k-2}，…，B_0 树的根。图 19-7c 说明了通
468 过在第 3 行中逆转 x 的子女表，得到一个包含 x 树中除 x 而外的每个结点的二项堆 H'。因为在第 1 行中 x 树已从 H 中去掉，故第 4 行中合并 H 和 H'所得的结果二项堆(如图 19-7d 所示)包含原先 H 中除 x 以外的所有结点。最后，第 5 行返回 x。

因为如果 H 有 n 个结点，则第 1～4 行每次需时间 $O(\lg n)$。所以，BINOMIAL-HEAP-EXTRACT-MIN 的运行时间为 $O(\lg n)$。

减小关键字的值

下面的过程将二项堆 H 中的某一结点 x 的关键字减小为一个新值 k。如果 k 大于 x 的当前关键字值，这个过程就引发一个错误。

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
 1  if k > key[x]
 2    then error "new key is greater than current key"
 3  key[x] ← k
 4  y ← x
 5  z ← p[y]
 6  while z ≠ NIL and key[y]<key[z]
 7      do exchange key[y] ↔ key[z]
 8         ▷ If y and z have satellite fields, exchange them, too.
 9         y ← z
10         z ← p[y]
```

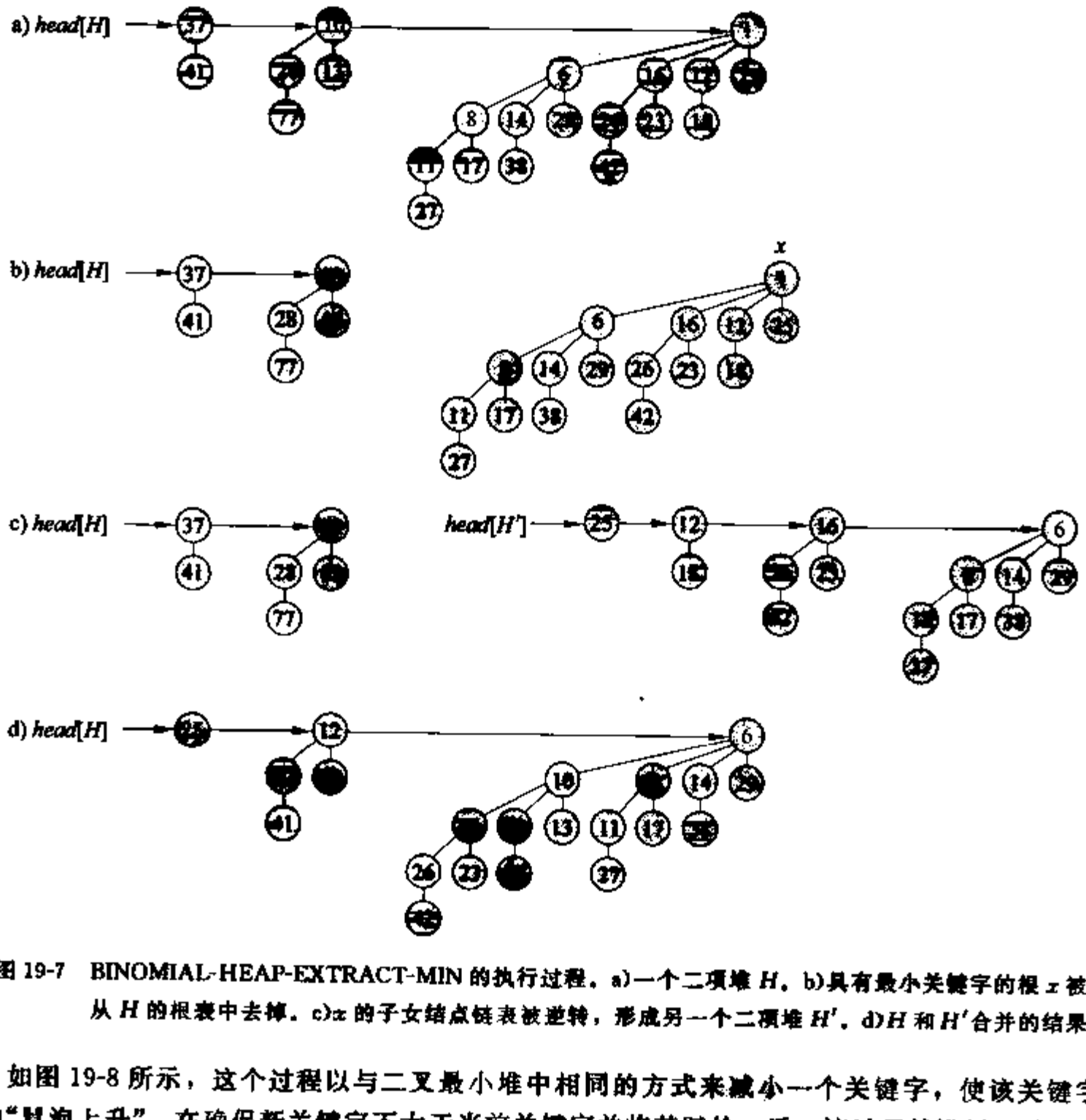


图 19-7　BINOMIAL-HEAP-EXTRACT-MIN 的执行过程。a)一个二项堆 H。b)具有最小关键字的根 x 被从 H 的根表中去掉。c)x 的子女结点链表被逆转，形成另一个二项堆 H'。d)H 和 H' 合并的结果

如图 19-8 所示，这个过程以与二叉最小堆中相同的方式来减小一个关键字，使该关键字在堆中“冒泡上升”。在确保新关键字不大于当前关键字并将其赋给 x 后，该过程就沿树上升。开始时 y 指向结点 x。在第 6～10 行 while 循环的每次迭代中，将 $key[y]$与 y 的父结点 z 的关键字作比较。如果 y 为根或 $key[y] \geqslant key[z]$，则该二项树已是最小堆有序。否则，结点 y 就违反了最小堆有序，故将其关键字与其父结点 z 的关键字相交换，同时还要交换其他的卫星数据。然后，这个过程将 y 置为 z，在树中上升一层，并继续下一次循环。

BINOMIAL-HEAP-DECREASE-KEY 过程的时间为 $O(\lg n)$。根据引理 19.1 的性质 2，x 的最大深度为 $\lfloor \lg n \rfloor$，故第 6～10 行的while 循环迭代至多 $\lfloor \lg n \rfloor$ 次。

删除一个关键字

很容易在 $O(\lg n)$时间内从二项堆 H 中删除一个结点 x 的关键字及卫星数据。在下面的实现中，假定当前在二项堆中的所有结点的关键字都不为 $-\infty$。

```
BINOMIAL-HEAP-DELETE(H, x)
1  BINOMIAL-HEAP-DECREASE-KEY(H, x, -∞)
2  BINOMIAL-HEAP-EXTRACT-MIN(H)
```

这个过程使结点 x 在整个二项堆中具有唯一最小的关键字，即给其一个关键字 $-\infty$。(练

习 19.2-6处理了$-\infty$不能作为关键字出现的情形，包括暂时作为关键字的情形。)然后，通过调用 BINOMIAL-HEAP-DECREASE-KEY，来使该关键字及其卫星数据信息冒泡上升至树根。接着，再通过调用 BINOMIAL-HEAP-EXTRACT-MIN 将根从 H 中去掉。

过程 BINOMIAL-HEAP-DELETE 的时间为 $O(\lg n)$。

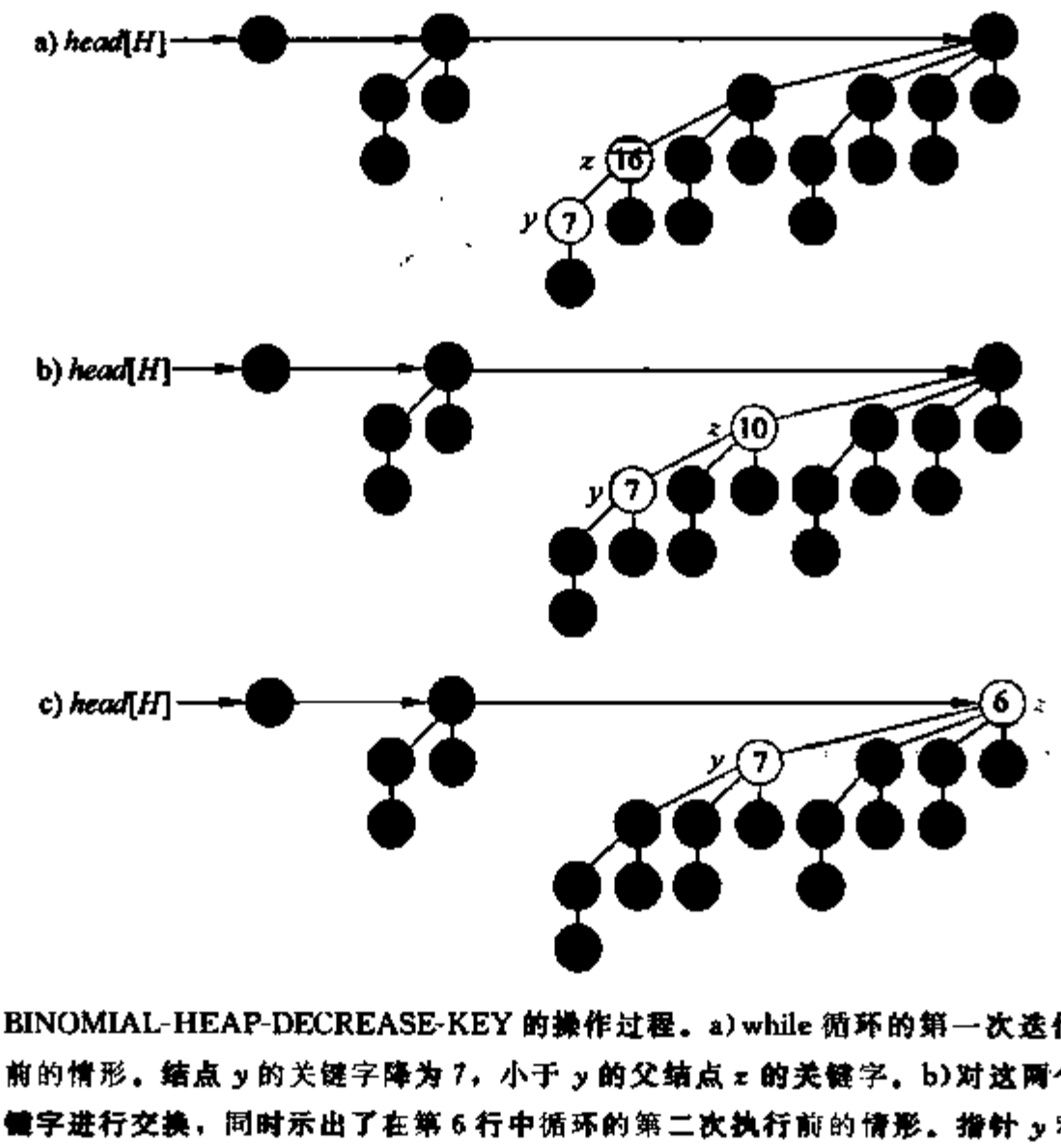


图 19-8　BINOMIAL-HEAP-DECREASE-KEY 的操作过程。a)while 循环的第一次迭代(第 6 行)前的情形。结点 y 的关键字降为 7，小于 y 的父结点 z 的关键字。b)对这两个结点的关键字进行交换，同时示出了在第 6 行中循环的第二次执行前的情形。指针 y 和 z 移向树中的上一层，但仍然违反最小堆序要求。c)再做一次交换，并将指针 y 和 z 再向上移一层，就可以满足最小堆序的要求，故 while 循环结束

练习

471　**19.2-1**　请写出 BINOMIAL-HEAP-MERGE 的伪代码。

19.2-2　请给出将关键字为 24 的结点插入如图 19-7d 中所示的二项堆后，所得的结果二项堆。

19.2-3　请给出将关键字为 28 的结点从图 19-8c 中的二项堆中删除后的结果。

19.2-4　讨论使用如下循环不变式时 BINOMIAL-HEAP-UNION 的正确性。

在第 9～21 行每一次 while 循环迭代开始时，x 指向下列之一的根：

- 该度数下唯一的根；
- 该度数下仅有两根中的第一个；
- 该度数下仅有三个根中的第一或第二个。

而且，根表中在 x 的前驱之前的所有根结点都有唯一度数，如果 x 的前驱的度数不同于 x，则它在根表中的度数也是唯一的。最后，根表结点的顺序是按结点度数单调递增的。

19.2-5　请解释，如果关键字的值可以是∞，为什么过程 BINOMIAL-HEAP-MINIMUM 可能无法正确工作？重写这个过程的伪代码，使之在这种情况下也能正常工作。

19.2-6　假设无法表示出关键字$-\infty$。重写 BINOMIAL-HEAP-DELETE 过程，使之在这种情况下能正确地工作。运行时间仍应为 $O(\lg n)$。

19.2-7　讨论二项堆上的插入与一个二进制数增值的关系，以及合并两个二项堆与将两个二进制数相加之间的关系。

19.2-8　根据练习 19.2-7，在不调用 BINOMIAL-HEAP-UNION 的前提下，重写 BINOMIAL-HEAP-INSERT，将一个结点直接插入一个二项堆中。

472

19.2-9　证明：如果将根表按度数排成严格递减序（而不是严格递增序）保存，仍可以在不改变渐近运行时间的前提下实现每一种二项堆操作。

19.2-10　请找出使 BINOMIAL-HEAP-EXTRACT-MIN、BINOMIAL-HEAP-DECREASE-KEY 以及 BINOMIAL-HEAP DELETE 的运行时间为 $\Omega(\lg n)$的输入。解释为什么 BINOMIAL-HEAP-INSERT，BINOMIAL-HEAP-MINIMUM 以及 BINOMIAL-HEAP-UNION 的最坏运行时间是 $\tilde{\Omega}(\lg n)$，而不是 $\Omega(\lg n)$（见思考题 3-5）。

思考题

19-1　2-3-4 堆

在第 18 章中介绍了 2-3-4 树，其中每个内结点（非根可能）有两个、三个或四个子女，且所有的叶结点的深度相同。在这个问题里，我们来实现 2-3-4 堆，它支持可合并堆的操作。

2-3-4 堆与 2-3-4 树有些不同之处。在 2-3-4 堆中，关键字仅存在于叶结点中，且每个叶结点 x 仅包含一个关键字于其 $key[x]$域中。另外，叶结点中的关键字之间没有什么特别的次序；亦即，从左至右来看，各关键字可以排成任何次序。每个内结点 x 包含一个值 $small[x]$，它等于以 x 为根的子树的各叶结点中所存储的最小关键字。根 r 包含了一个 $height[r]$域，即树的高度。最后，2-3-4 堆主要是在主存中的，故无需任何磁盘读写。

请实现下面各 2-3-4 堆操作。对包含 n 个元素的 2-3-4 堆，a)～e)中的每个操作的运行时间都应是 $O(\lg n)$。f)中的 UNION 操作的运行时间应是 $O(\lg n)$，其中 n 为两个输入堆中的元素个数。

a) MINIMUM，返回一个指向最小关键字的叶结点的指针。

b) DECREASE-KEY，它将某一给定叶结点 x 的关键字减小为一个给定的值 $k \leqslant key[x]$。

c) INSERT，插入具有关键字 k 的叶结点 x。

d) DELETE，删除一给定叶结点 x。

e) EXTRACT-MIN，抽取具有最小关键字的叶结点。

473

f) UINION，合并两个 2-3-4 堆，返回一个 2-3-4 堆并破坏输入堆。

19-2　采用二项堆的最小生成树算法

第 23 章要介绍两个在无向图中寻找最小生成树的算法。这里我们可以看到如何利用二项堆来设计一个不同的最小生成树算法。

给定一个连通的无向图 $G=(V, E)$，以及权值函数 $w: E \to \mathbf{R}$。称 $w(u, v)$为边(u, v)的权。希望找出 G 的最小生成树：一个无环子集 $T \subseteq E$ 连接 V 中所有结点，且总的权值

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

为最小。

下面的伪代码构造一个最小生成树 T，可用 23.1 节的技术来证明这个过程是正确的。对 V 的结点保持一个划分$\{V_i\}$，且对每组 V_i，都有一个与 V_i 中结点关联的边的集合

$$E_i \subseteq \{(u,v) : u \in V_i \text{ 或 } v \in V_i\}$$

```
MST(G)
 1  T ← ∅
 2  for each vertex v_i ∈ V[G]
 3       do V_i ← {v_i}
 4          E_i ← {(v_i, v) ∈ E[G]}
 5  while there is more than one set V_i
 6       do choose any set V_i
 7          extract the minimum-weight edge(u, v) from E_i
 8          assume without loss of generality that u ∈ V_i and v ∈ V_j
 9          if i ≠ j
10            then T ← T ∪ {(u, v)}
11                 V_i ← V_i ∪ V_j, destroying V_j
12                 E_i ← E_i ∪ E_j
```

请说明如何用二项堆来实现此算法，以便管理点集和边集。需要对二项堆的表示做改变吗？需要增加图 19-1 中所没有的可合并堆操作吗？给出你的实现的运行时间。

474

本章注记

二项堆是在 1978 年由 Vuillemin[307]提出的。其后，Brown[49，50]对它们的性质做了深入的研究。

475

第 20 章　斐波那契堆

在第 19 章中，我们看到了二项堆如何以 $O(\lg n)$ 的最坏情况时间支持可合并堆操作 INSERT，MINIMUM，EXTRACT-MIN 和 UNION，以及操作 DECREASE-KEY 和 DELETE。在这一章里，我们将讨论斐波那契堆，它也支持这些操作，但有一个长处，即不涉及删除元素的操作仅需 $O(1)$ 的平摊运行时间。

从理论上来看，当相对于其他操作的数目来说，EXTRACT-MIN 与 DELETE 操作的数目较小时，斐波那契堆是很理想的。在许多应用中都会出现这个情况。例如，某些图问题的算法对每条边都调用一次 DECREASE-KEY。对有许多边的稠密图来说，每一次 DECREASE-KEY 调用的 $O(1)$ 平摊时间加起来，就是对二叉或二项堆的 $\Theta(\lg n)$ 最坏情况时间的一个很大改善。用于解决诸如最小生成树（第 23 章）和寻找单源最短路径（第 24 章）等问题的快速算法都要用到斐波那契堆。

但是，从实际上看，对大多数应用来说，由于斐波那契堆的常数因子以及程序设计上的复杂性，使得它不如通常的二叉（或 k 叉）堆合适。因此，斐波那契堆主要是具有理论上的意义。如果能设计出一种与斐波那契堆有相同的平摊时间界但又简单得多的数据结构，那么它就会有很大的实用价值了。

和二项堆一样，斐波那契堆由一组树构成。实际上，这种堆松散地基于二项堆。如果不对斐波那契堆做任何 DECREASE-KEY 或 DELETE 操作，则堆中的每棵树就和二项树一样。两种堆相比，斐波那契堆的结构比二项堆更松散一些，从而可以改善渐近时间界。对结构的维护工作可被延迟到方便时再做。

像 17.4 节中的动态表一样，斐波那契堆也是以平摊分析为指导思想来设计数据结构的很好的例子。在这一章余下部分对斐波那契堆操作的分析中，大量应用 17.3 节中的势能方法。 476

在这一章，我们假定读者已经阅读过第 19 章中有关二项堆的内容。有关各种堆操作的说明，以及对二叉堆、二项堆和斐波那契堆上各种操作的时间界的总结，都已经在第 19 章的图 19-1 中出现过了。我们给出的斐波那契堆的结构依赖于二项堆的结构。对斐波那契堆的有些操作与对二项堆是类似的。

和二项堆一样，斐波那契堆不能有效地支持 SEARCH 操作。因此，对于那些作用于某一给定结点的操作，就需要以一个指向该结点的指针作为输入的一部分。在一个应用中使用斐波那契堆时，通常将每个对应的应用对象的柄存入每个斐波那契堆元素中，同时也将每个对应的斐波那契堆元素的柄存入其应用对象中。

20.1 节定义斐波那契堆，讨论它们的表示方式，并给出用来对其进行平摊分析的势函数。20.2 节说明如何实现可合并堆操作，并取得图 19-1 中所给出的各平摊时间界。另外两个操作 DECREASE-KEY 与 DELETE 在 20.3 节中介绍。最后，20.4 节完成分析过程的一个关键部分，并解释此数据结构名称的来历。

20.1　斐波那契堆的结构

与二项堆一样，斐波那契堆是由一组最小堆有序树构成，但堆中的树并不一定是二项树。图 20-1a 给出了一个斐波那契堆的例子。

与二项堆中树都是有序的不同，斐波那契堆中的树都是有根而无序的。如图 20-1b 所示，每个结点 x 包含一个指向其父结点的指针 $p[x]$，以及一个指向其任一子女的指针 $child[x]$。x 的

所有子女被链接成一个环形双链表，称为 x 的子女表。子女表中的每个孩子 y 有指针 $left[y]$和 $right[y]$，分别指向其左、右兄弟。如果 y 结点是独子，则 $left[y]=right[y]=y$。各兄弟在子女表中出现的次序是任意的。

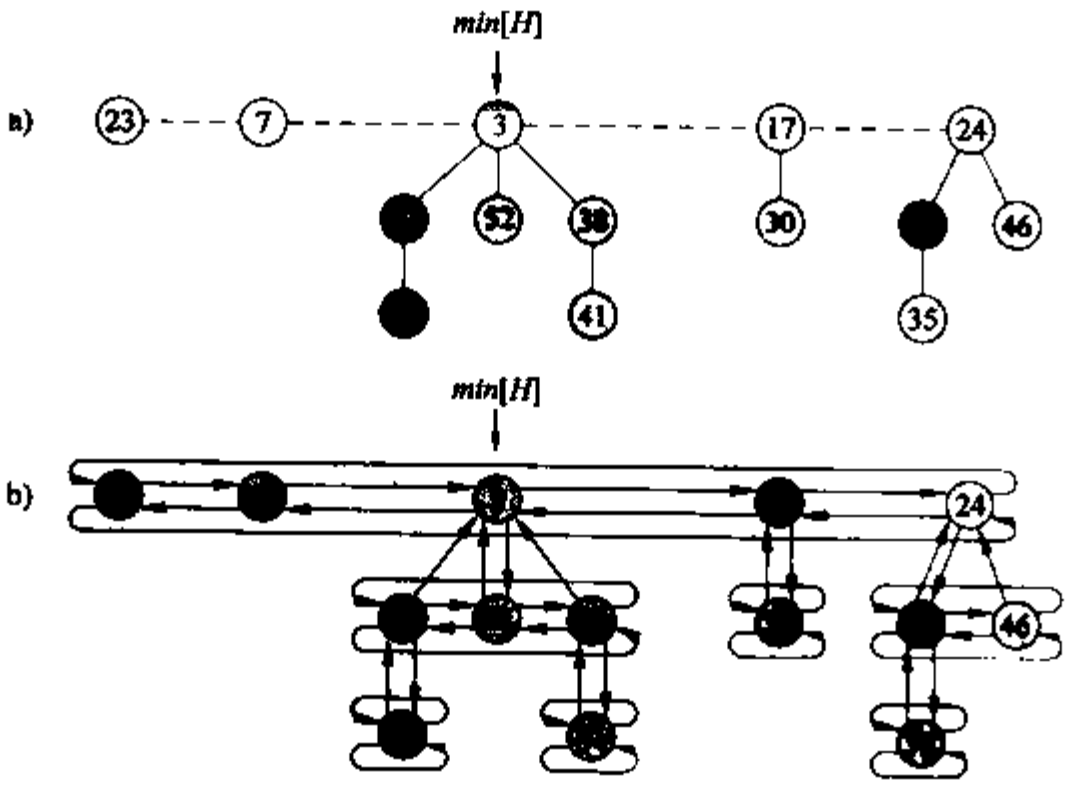


图 20-1　a)由 5 棵最小堆有序树和 14 个结点构成的一个斐波那契堆。虚线指示根表。堆中最小结点为包含关键字 3 的结点。三个被标记的结点都被加黑了。这个斐波那契堆的势为 5+2×3=11。b)一个显示指针 p(向上的箭头)、$child$(向下的箭头)、$left$ 和 $right$(侧向箭头)的更完全的表示。在本章余下的插图中略去了这些细节，因为这里所示的所有信息都可从 a)中得到

在斐波那契堆中采用环形双链表(见 10.2 节)有两个好处。首先，可以在 $O(1)$时间内将某结点从环形双链表中去掉。其次，给定两个这样的表，可以在 $O(1)$时间内将它们连接为一个环形双链表。在对斐波那契堆操作的描述中，我们将非形式地提及这两种操作，实现的细节留给读者去补充。

每一个结点中的另外两个域也很有用。结点 x 的子女表中子女的个数存储于 $degree[x]$中；布尔值域 $mark[x]$指示自从 x 上一次成为另一个结点子女以来，它是否失掉了一个孩子。新创建的结点是没有标记的，且当结点 x 成为另一结点的孩子时也是没有标记的。直到 20.3 节的 DECREASE-KEY 操作，我们才会置所有 $mark$ 域为 FALSE。

对于一个给定的斐波那契堆 H，可以通过指向包含最小关键字的树根的指针 $min[H]$来访问，这个结点被称为斐波那契堆中的*最小结点*。如果一个斐波那契堆 H 是空的，则 $min[H]=$NIL。

在一个斐波那契堆中，所有树的根都通过用其 $left$ 和 $right$ 指针链接成一个环形的双链表，称为该堆的*根表*。于是，指针 $min[H]$就指向根表中具有最小关键字的结点。在根表中各树的顺序可以是任意的。

477
~
478

我们还要用到另一个有关斐波那契堆 H 的属性：H 中目前所包含的结点个数为 $n[H]$。

势函数

前面已经说过，我们将用 17.3 节中的势能方法来分析斐波那契堆操作的性能。对一个给定的斐波那契堆 H，用 $t(H)$来表示 H 的根表中树的棵数，用 $m(H)$来表示 H 中有标记结点的个数。斐波那契堆 H 的势定义为

$$\Phi(H) = t(H) + 2m(H) \tag{20.1}$$

(20.3 节中还将给出这一势函数的直观解释。)例如，图 20-1 中所示的斐波那契堆的势为 5+2×3=11。

一组斐波那契堆的势为各成分斐波那契堆的势之和。假定一个单位的势可以支付常数量的工作，此处该常数足够大，可以覆盖我们可能遇到的任何常数时间的工作。

此外，假定斐波那契堆应用在开始时，都没有堆。于是，初始的势就为 0，且根据方程(20.1)，势始终是非负的。根据(17.3)式可知，对某一操作序列来说，其总的平摊代价的一个上界也就是这个操作序列总的实际代价的一个上界。

最大度数

在这一章余下几节里要做的平摊分析中，都假定在包含 n 个结点的斐波那契堆中，结点的最大度数有一个已知的上界 $D(n)$。练习 20.2-3 说明当斐波那契堆仅支持可合并堆操作时，$D(n) \leqslant \lfloor \lg n \rfloor$。在 20.3 节中，我们将看到当斐波那契堆还需支持 DECREASE-KEY 和 DELETE 操作时，$D(n) = O(\lg n)$。

20.2　可合并堆的操作

在这一节里，我们要介绍并分析斐波那契堆所实现的各种可合并堆操作。如果仅需支持 MAKE-HEAP，INSERT，MINIMUM，EXTRACT-MIN 和 UNION 操作，则每个斐波那契堆就只是一组"无序的"二项树。无序的二项树和二项树一样，也是递归定义的。无序的二项树 U_0 包含一个结点，一棵无序的二项树 U_k 包含两棵无序二项树 U_{k-1}，其中一棵的根结点成为另一棵的根结点的任意子结点。引理 19.1 中给出的二项树的性质对无序二项树仍然成立，但性质 4 要做如下变化(见练习 20.2-2)：

479

4′. 对无序二项树 U_k，根的度数为 k，它大于任何其他结点的度数。根的子女按某种顺序分别为子树 U_0，U_1，…，U_{k-1} 的根。

于是，如果一个有 n 个结点的斐波那契堆由一组无序二项树构成，则 $D(n) = \lg n$。

对斐波那契堆上的各种可合并堆操作来说，其关键思想就是尽可能久地将工作推后。在各操作的实现之间有一个性能权衡的问题。如果斐波那契堆中树的棵数较少，则在一次 EXTRACT-MIN 操作中，可以很快地在余下的结点中确定新的最小结点。然而，正如我们已在练习 19.2-10 中见过的二项堆的情形那样，为确保树的数目较小，就要付出一定的代价：可能要花 $\Omega(\lg n)$ 时间向一个二项堆中插入一个结点或合并两个二项堆。下面将看到，当向斐波那契堆中插入新结点或合并两个斐波那契堆时，并不去合并树，而是将这个工作留给 EXTRACT-MIN 操作，那时就真正需要找出新的最小结点了。

创建一个新的斐波那契堆

创建一个空的斐波那契堆，过程 MAKE-FIB-HEAP 分配并返回一个斐波那契堆对象 H，且 $n[H] = 0$，$min[H] = \text{NIL}$；此时 H 中还没有树。因为 $t(H) = 0$，$m(H) = 0$，该空斐波那契堆的势 $\Phi(H) = 0$。因此，MAKE-FIB-HEAP 的平摊代价就等于其 $O(1)$ 的实际代价。

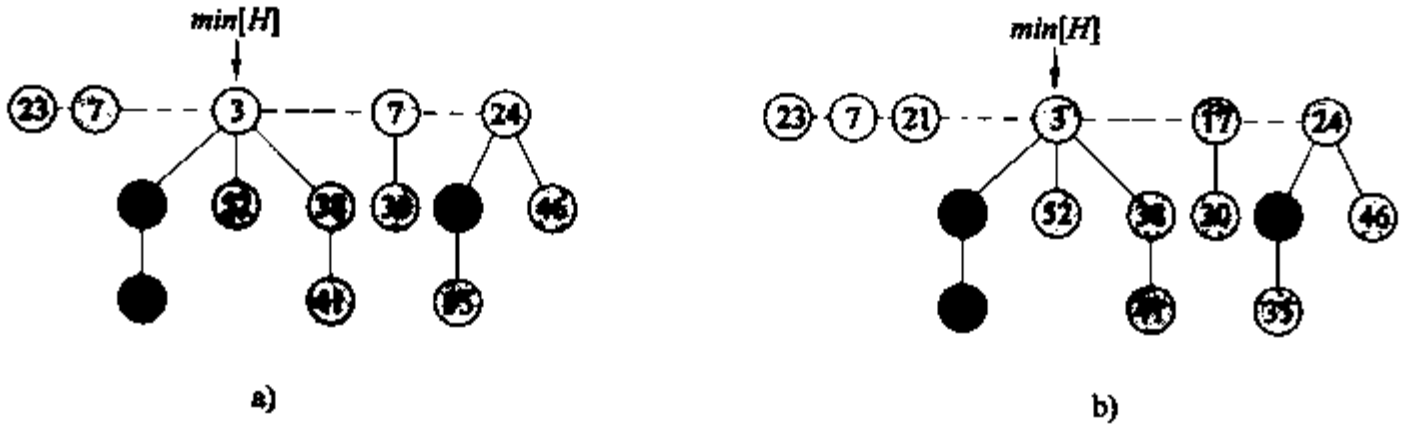
插入一个结点

下面的过程将结点 x 插入斐波那契堆 H，并假定该结点已被分配，且其 $key[x]$ 已填有内容。

```
FIB-HEAP-INSERT(H, x)
1  degree[x] ← 0
2  p[x] ← NIL
3  child[x] ← NIL
4  left[x] ← x
5  right[x] ← x
```

```
 6  mark[x] ← FALSE
 7  concatenate the root list containing x with root list H
 8  if min[H] = NIL or key[x] < key[min[H]]
 9     then min[H] ← x
480 10  n[H] ← n[H]+1
```

在第 1～6 行对结点 x 的各域进行初始化，并构造其自身的环形双向链表后，第 7 行在 $O(1)$ 的实际时间内将 x 加入到 H 的根表中。于是，结点 x 成为一棵单结点的最小堆有序树，同时也就是斐波那契堆中的一棵无序二项树。它没有任何子女，也没被加过标记。第 8～9 行在必要时对指向斐波那契堆 H 中最小结点的指针进行更新。最后，第 10 行增加 $n[H]$ 来反映出新增加了一个结点。图 20-2 示出将一个具有关键字 21 的结点插入图 20-1 中的斐波那契堆后的结果。



a)　　b)

图 20-2　将一个结点插入斐波那契堆。a)一个斐波那契堆 H。b)插入关键字为 21 的结点后的斐波那契堆 H。该结点自成一棵最小堆有序树，从而被加入到根表中，成为根的左兄弟

与 BINOMIAL-HEAP-INSERT 过程不同，FIB-HEAP-INSERT 并不对斐波那契堆中的树进行合并。如果连续执行了 k 次 FIB-HEAP-INSERT 操作，则 k 棵单结点的树被加到了根表中。

为了确定 FIB-HEAP-INSERT 的平摊代价，设 H 为输入的斐波那契堆，H' 为结果斐波那契堆。于是，$t(H')=t(H)+1$，$m(H')=m(H)$，且势的增加为

$$((t(H)+1)+2m(H))-(t(H)+2m(H))=1$$

因为实际的代价为 $O(1)$，故平摊的代价为 $O(1)+1=O(1)$。

寻找最小结点

在一个斐波那契堆 H 中，最小的结点由指针 $min[H]$ 指示，故可以在 $O(1)$ 实际时间内找到最小结点。又因为 H 的势没有变化，所以这个操作的平摊代价就等于其 $O(1)$ 的实际代价。

合并两个斐波那契堆

下面的过程合并斐波那契堆 H_1 和 H_2，同时也破坏了 H_1 和 H_2。它仅简单地将 H_1 和 H_2
481 的两根表并置，然后确定一个新的最小结点。

```
FIB-HEAP-UNION(H1, H2)
1  H ← MAKE-FIB-HEAP()
2  min[H] ← min[H1]
3  concatenate the root list of H2 with the root list of H
4  if (min[H1] = NIL) or (min[H2] ≠ NIL and min[H2] < min[H1])
5     then min[H] ← min[H2]
6  n[H] ← n[H1]+n[H2]
7  free the objects H1 and H2
8  return H
```

第 1～3 行将 H_1 和 H_2 的根表拼接成一个新的根表 H，第 2，4 行和第 5 行设置 H 的最小结点，第 6 行将 $n[H]$置为总结点数，第 7 行中释放斐波那契堆对象 H_1 和 H_2，第 8 行返回结果斐波那契堆 H。与过程 FIB-HEAP-INSERT 中一样，并没有对树进行合并。

势的改变为

$$\Phi(H)-(\Phi(H_1)+\Phi(H_2))=(t(H)+2m(H))-((t(H_1)+2m(H_1))+(t(H_2)+2m(H_2)))=0,$$

这是因为 $t(H)=t(H_1)+t(H_2)$，且 $m(H)=m(H_1)+m(H_2)$。所以，FIB-HEAP-UNION 的平摊代价与其 $O(1)$的实际代价相等。

抽取最小结点

抽取最小结点的过程是这一节所介绍的操作中最复杂的。先前，对根表中的树进行合并这项工作是被推后的，那么，到了这儿，最终要由这个操作完成。下面的伪代码完成了抽取最小结点的工作。为方便起见，代码中假定从链表中删除一个结点时，仍在表中的指针被更新，而被抽取结点的指针则无变化。该过程还用到辅助过程 CONSOLIDATE(稍后给出)。 482

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z ← min[H]
 2  if z ≠ NIL
 3      then for each child x of z
 4               do add x to the root list of H
 5                  p[x] ← NIL
 6           remove z from the root list of H
 7           if z = right[z]
 8             then min[H] ← NIL
 9             else min[H] ← right[z]
10                  CONSOLIDATE(H)
11          n[H] ← n[H]-1
12  return z
```

如图 20-3 所示，FIB-HEAP-EXTRACT-MIN 先使最小结点的每个子女都成为一个根，并将最小结点从根表中去掉。然后，通过将度数相同的根链接起来，直至对应每个度数至多只有一个根来调整根表。

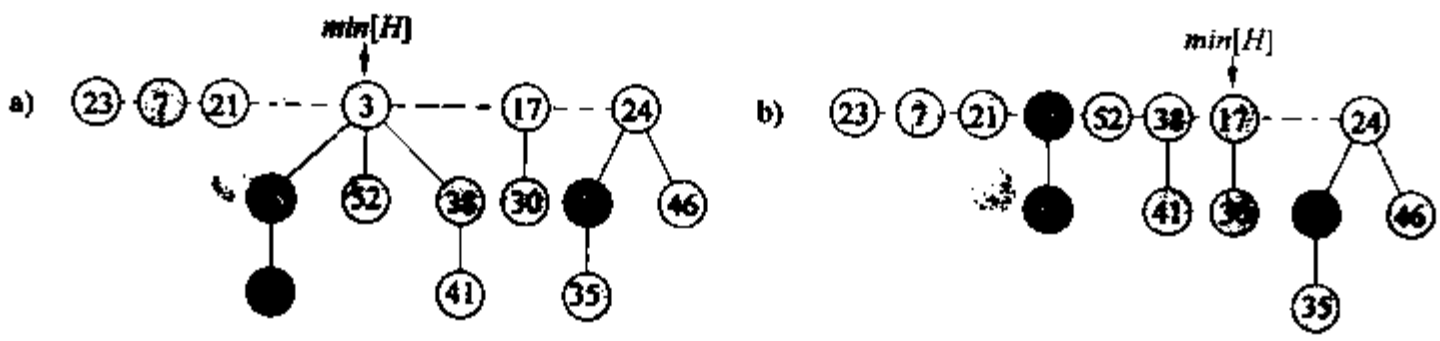


图 20-3　FIB-HEAP-EXTRACT-MIN 的操作过程。a)斐波那契堆 H。b)将最小结点 z 从根表中去掉，并将其子女加到根表后的情形。c)至 e)在过程 CONSOLIDATE 的第 3～13 行 for 循环中，头三次迭代的每一次以后的树组 A 和各棵树。对根表的处理是从用 $min[H]$指向的结点开始，并遵循 $right$ 指针而进行的。每个图都示出了在一次迭代结束时 w 和 x 的值。f)-h)for 循环的下一轮迭代，同时还示出了在第 6～12 行中 while 循环的每次迭代结束时 w 和 x 的值。f)图示出了 while 循环第一次执行时的情形。关键字 23 的结点被链接向关键字 7 的结点，后者由 x 指向。g)关键字为 17 的结点被链向关键字为 7 的结点，它仍由 x 指向。h)关键字为 24 的结点被链向关键字为 7 的结点。因为先前并没有一个结点由 A[3]所指向，故在 for 循环迭代结束时，A[3]被设为指向结果树的根。i)-l)for 循环的后四次迭代执行中每一次以后的情形。m)在通过数组 A 以及确定新的 $min[H]$指针而重构根表后的斐波那契堆 H

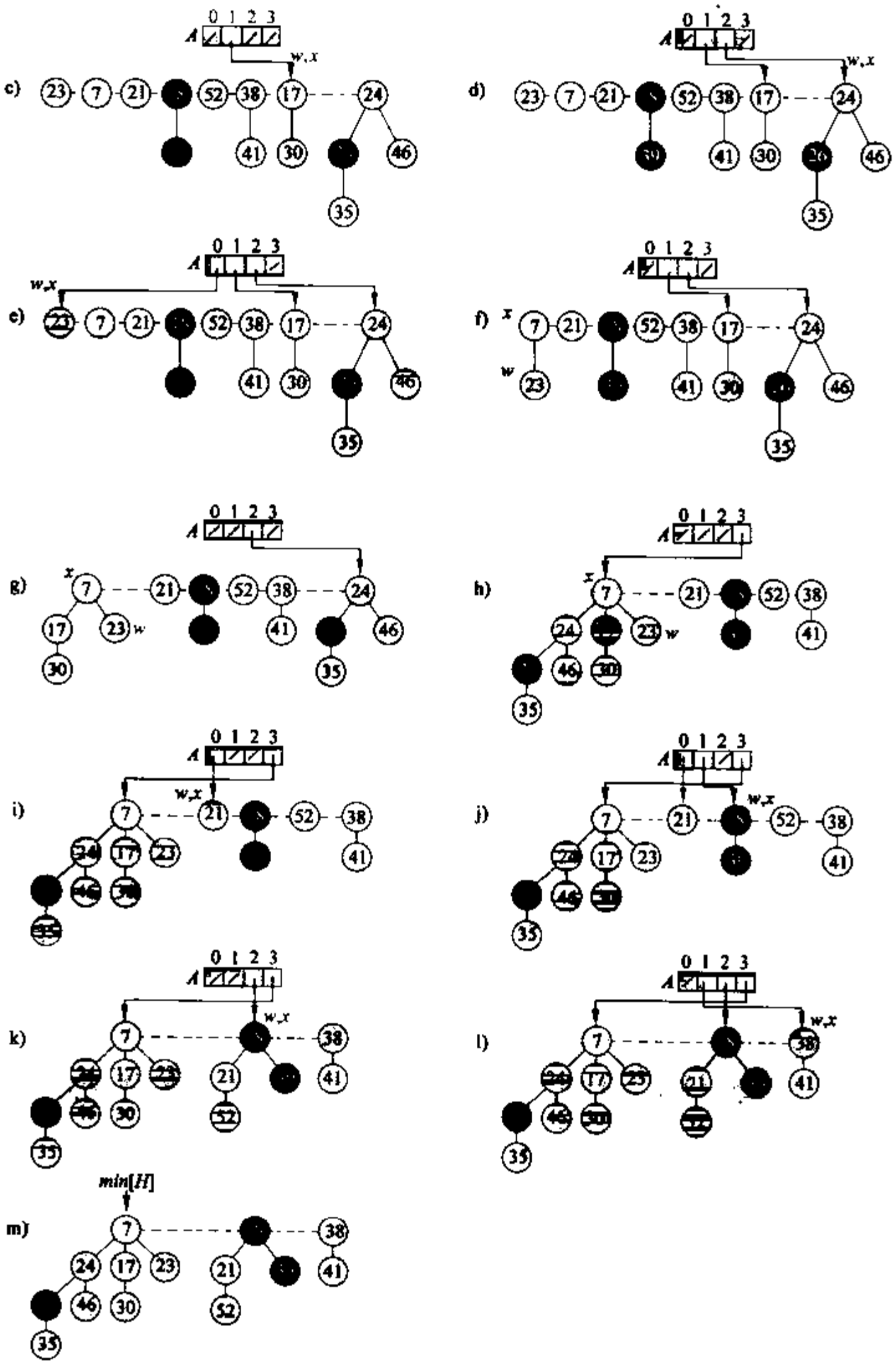


图 20-3　（续）

第 1 行中先保存指向最小结点的指针 z，该指针在最后返回。如果 z=NIL，则斐波那契堆 H 已经为空，结束；否则，向在过程 BINOMIAL HEAP-EXTRACT-MIN 中一样，通过在第 3～5 行中使 z 的所有子女成为根(将它们放入根表)来从 H 中删除结点 z，并在第 6 行中将 z 从根表中去掉。如果第 6 行后 $z=right[z]$，则 z 为根表中唯一的结点且没有子女，于是所有余下的工作就是在返回 z 之前在第 8 行中使斐波那契堆为空。否则，让指针 $min[H]$指向根表中的一个非 z 的结点(在这个情况里，即 $right[z]$)，当然这个结点并不一定就是调用完 FIB-HEAP-EXTRACT-MIN 后新的最小结点。图 20-3b 示出了图 20-3a 中的斐波那契堆在执行了第 9 行后的结果。

下一步要合并 H 的根表，即减少斐波那契堆中树的数目，这由调用 CONSOLIDATE(H)来完成。对根表的合并过程即反复执行下面的步骤，直到根表中的每个根都有一个不同的 $degree$ 值时为止：

1)在根表中找出两个具有相同度数的根 x 和 y，且 $key[x] \leqslant key[y]$。

2)将 y 链接到 x：将 y 从根表中移出，成为 x 的一个孩子。这个操作由 FIB-HEAP-LINK 过程完成。域 $degree[x]$增值，且如果 y 上有标记的话也被去掉。

过程 CONSOLIDATE 使用了一个辅助数组 $A[0..D(n[H])]$；如果 $A[i]=y$，则当前的 y 是个 $degree[y]=i$ 的根。

483
|
485

```
CONSOLIDATE(H)
 1  for i ← 0 to D(n[H])
 2       do A[i] ← NIL
 3  for each node w in the root list of H
 4       do x ← w
 5          d ← degree[x]
 6          while A[d]≠NIL
 7             do y ← A[d]        ▷ Another node with the same degree as x.
 8                if key[x]>key[y]
 9                  then exchange x ↔ y
10                FIB-HEAP-LINK(H, y, x)
11                A[d] ← NIL
12                d ← d+1
13             A[d] ← x
14  min[H] ← NIL
15  for i ← 0 to D(n[H])
16       do if A[i]≠NIL
17             then add A[i] to the root list of H
18                  if min[H]=NIL or key[A[i]]<key[min[H]]
19                    then min[H] ← A[i]

FIB-HEAP-LINK(H, y, x)
1  remove y from the root list of H
2  make y a child of x, incrementing degree[x]
3  mark[y] ← FALSE
```

具体地说，CONSOLIDATE 的工作过程如下：在第 1～2 行中对 A 进行初始化，置每一项为 NIL。第 3～13 行的 for 循环对根表中的每一个根 w 进行处理。在完成对每个根 w 的处理后，它结束于以某结点 x 为根的树中，x 可能(也可能不)与 w 相同。在被处理的根结点中，没有度数与 x 是相同的，所以让 $A[degree[x]]$指向 x。当 for 循环终止时，每个度数下至多有一个根，且数

组 A 指向每一个留下的根。

在第 6～12 行的 while 循环中，要反复地将包含结点 w 的树的根 x 链接到与其相同度数的其他树根上，直到没有其他度数相同的根时为止。这个 while 循环维持了如下的循环不变式：

在 while 循环每次迭代之初，$d=degree[x]$。

我们按如下方式使用此循环不变式：

486

初始化：第 5 行确保了在第一次进入循环时循环不变式成立。

保持：在 while 循环的每一次迭代中，$A[d]$指向某根 y。因为 $d=degree[x]=degree[y]$，所以将 x 和 y 链接。在链接操作之后，x 和 y 中具有较小关键字者就成为另一个的父结点，故如有必要，则在第 8～9 行交换 x 和 y 的指针。然后，在第 10 行中，通过调用 FIB-HEAP-LINK (H, y, x)将 y 链到 x 上。这个调用使 $degree[x]$变大，却使 $degree[y]$仍为 d。因为结点 y 不再是个根，故在第 11 行中，从数组 A 中去掉指向它的指针。$degree[x]$的值因为调用 FIB-HEAP-LINK 而增加，故第 12 行恢复不变式 $d=degree[x]$。

终止：重复 while 循环直到 $A[d]=$NIL，此时没有别的根的度数与 x 的相同。

在 while 循环终止后，第 13 行中将 $A[d]$置为 x，并执行 for 循环的下一轮迭代。

在图 20-3c-e 中，示出了数组 A 以及第 3～13 行 for 循环的前三次迭代后的结果树。在 for 循环下一次迭代的执行中，发生了三次链接，其结果如图 20-3f-h 所示。图 20-3i-l 示出了 for 循环的下四次迭代执行的结果。

所有余下的就是清理工作。当第 3～13 行的 for 循环结束后，第 14 行清空根表，第 15～19 行根据数组 A 重新构造根表，所得的斐波那契堆如图 20-3m 所示。在调整根表后，FIB-HEAP-EXTRACT-MIN 通过在第 11 行中减小 $n[H]$，并在第 12 行中返回一个指向被删除结点 z 的指针而结束。

请注意，如果在执行 FIB-HEAP-EXTRACT-MIN 前，斐波那契堆中所有的树都是无序二项树，则在此以后，它们也都是无序的二项树。树发生变化的方式有两种。首先，在 FIB-HEAP-EXTRACT-MIN 的第 3～5 行中，根 z 的每个孩子 x 成为一个根。根据练习 20.2-2，每棵新的树都是无序的二项树。其次，仅当若干棵树有相同度数时，才用 FIB-HEAP-LINK 把它们链接起来。因为在链接前，所有的树都是无序的二项树，若两棵树的根都各有 k 个子女，则它们必具有 U_k 的结构。于是，链接所得的树就具有 U_{k+1}的结构。

现在来证明从一个包含 n 个结点的斐波那契堆中，抽取最小结点的平摊代价为 $O(D(n))$。设 H 表示执行 FIB-HEAP-EXTRACT-MIN 操作前的斐波那契堆。

抽取最小结点的实际代价可如下计算。因为在 FIB-HEAP-EXTRACT-MIN 中，至多处理最小结点的 $D(n)$个子女，再加上第 1～2 行和第 14～19 行中 CONSOLIDATE 所作的工作，合起来的时间代价为 $O(D(n))$。再来分析一下第 3～13 行中的 for 循环。在调用 CONSOLIDATE 时，

487

根表的大小至多为 $D(n)+t(H)-1$，因为它包含原来的 $t(H)$个根表结点，再减去被抽取的根结点，加上被抽取结点的子女(至多有 $D(n)$个)。在第 6～12 行的 while 循环的每次执行中，其中一个根要被链接到另一个上，则 for 循环中所做的工作量至多与 $D(n)+t(H)$成正比。这样，抽取最小结点总的实际工作量为 $O(D(n)+t(H))$。

在抽取最小结点之前的势为 $t(H)+2m(H)$，而在此之后的势至多为 $(D(n)+1)+2m(H)$，因为该操作之后至多留下 $D(n)+1$ 个根，且操作中没有任何结点被加标记。所以总的平摊代价至多为

$$\begin{aligned}&O(D(n)+t(H))+((D(n)+1)+2m(H))-(t(H)+2m(H))\\&=O(D(n))+O(t(H))-t(H)=O(D(n))\end{aligned}$$

这是因为，我们可以通过扩大势的单位来支配 $O(t(H))$中隐藏的常数。从直觉上看，执行每一次链接的代价是由势的减少来支付的，而势的减少又是由于链接操作使根的数目减少 1 而引起的。在 20.4 节中，我们将看到 $D(n)=O(\lg n)$，所以抽取最小结点的平摊代价为 $O(\lg n)$。

练习

20.2-1 请给出对图 20-3m 中所示的斐波那契堆调用 FIB-HEAP-EXTRACT-MIN 后得到的斐波那契堆。

20.2-2 证明：引理 19.1 对无序二项树也成立，但要将性质 4 换成性质 4′。

20.2-3 证明：如果仅需支持可合并堆操作，则在包含 n 个结点的斐波那契堆中结点的最大度数 $D(n)$至多为 $\lfloor \lg n \rfloor$。

20.2-4 McGee 教授设计了一种新的基于斐波那契堆的数据结构。McGee 堆与斐波那契堆具有相同的结构，也支持可合并堆操作。各操作的实现与斐波那契堆中的相同，只是插入和合并在最后的步骤中做合并调整。McGee 堆的操作的最坏情况运行时间是多少？教授所设计的数据结构到底有多少新意？

488

20.2-5 论证：如果对关键字的唯一操作是比较两个关键字(如本章的所有实现中的情况)，则并非所有的可合并堆操作都有 $O(1)$的平摊运行时间。

20.3　减小一个关键字与删除一个结点

这一节里，要介绍如何在 $O(1)$的平摊时间里，减小斐波那契堆中某结点的关键字值，以及如何在 $O(D(n))$的平摊时间内，从包含 n 个结点的斐波那契堆中删除一个结点。这些操作不保持斐波那契堆中的所有树都是无序二项树的性质。但它们非常接近，因而可用 $O(\lg n)$来限界最大度数 $D(n)$。证明这个界(我们将在 20.4 节中证明)隐含 FIB HEAP-EXTRACT-MIN 和 FIB HEAP DELETE 的平摊运行时间为 $O(\lg n)$。

减小一个关键字

在下面关于 FIB-HEAP-DECREASE-KEY 操作的伪代码中，假定从链表中删除一个结点并不改变被删除结点的任何结构域。

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k>key[x]
2     then error "new key is greater than current key"
3  key[x] ← k
4  y ← p[x]
5  if y≠NIL and key[x]<key[y]
6     then CUT(H, x, y)
7          CASCADING-CUT(H, y)
8  if key[x]<key[min[H]]
9     then min[H] ← x

CUT(H, x, y)
1  remove x from the child list of y, decrementing degree[y]
2  add x to the root list of H
3  p[x] ← NIL
4  mark[x] ← FALSE

CASCADING-CUT(H, y)
```

489

```
1  z ← p[y]
2  if z≠NIL
3    then if mark[y]=FALSE
4           then mark[y] ← TRUE
5           else CUT(H, y, z)
6                CASCADING-CUT(H, z)
```

FIB-HEAP-DECREASE-KEY 过程是这样工作的：第 1～3 行确保新关键字不大于 x 的当前关键字，并将新关键字赋给 x。如果 x 为根或 $key[x] \geqslant key[y]$，此处 y 为 x 的父结点，则无需发生任何结构上的变化，因为没有违反最小堆序。第 4～5 行测试这个条件。

如果违反了最小堆序，则要发生很多变化。先在第 6 行切断 x。过程 CUT"切断"x 与其父结点 y 之间的连接，使 x 成为一个根。

我们用 *mark* 域来获得期望时间界。这个域记录了每个结点的一小段历史。假设结点 x 经历了如下的事件：

1)在某个时刻，x 是个根。

2)然后，x 被链接到另一个结点上。

3)再通过切断来去除 x 的两个子女。

一旦第二个孩子也失掉后，x 与其父结点之间的联系就被切断了，并成为一个新根。如果发生了第 1 步和第 2 步，且 x 的一个孩子被切割掉了，则域 $mark[x]$为 TRUE。于是，CUT 过程在第 4 行清掉 $mark[x]$，因为它执行了第 1 步。(现在我们就能搞清楚为什么 FIB-HEAP-LINK 的第 3 行清掉了 $mark[y]$：结点 y 被链向另一个结点，故执行第 2 步。下一次去掉 y 的一个孩子时，$mark[y]$将被置为 TRUE)。

但事情还没有就此结束，因为 x 可能是其父结点 y 被链到另一结点后被切掉的第二个孩子。所以，FIB-HEAP-DECREASE-KEY 的第 7 行对 y 执行一次级联切断操作。如果 y 是个根，则 CASCADING-CUT 过程在第 2 行中的测试就返回。如果 y 是未标记的，则该过程在第 4 行对其加标记，因为它的第一个孩子刚被除去，然后返回。然而，如果 y 是有标记的，则说明 y 刚失去了其第二个孩子，在第 5 行中将 y 切掉，且 CASCADING-CUT 在第 6 行对 y 的父结点 z 再递归调用其自身。CASCADING-CUT 一直沿树递归上去，直至找到一个根结点或未加标记的结点。

一旦所有的级联删除都执行后，在必要的情况下，FIB-HEAP-DECREASE-KEY 的第 8～9 行更新 $min[H]$。关键字变化的结点也就是关键字被减小的结点 x。因此，新的最小结点也许是原最小结点，或是结点 x。

490

图 20-4 说明了两次调用 FIB-HEAP-DECREASE-KEY 的执行过程，开始时为图 20-4a 中所示的斐波那契堆。图 20-4b 中所示的第一次调用不涉及任何级联切断。图 20-4c-e 中所示的第二次调用引发了两次级联切断。

现在来证明 FIB-HEAP-DECREASE-KEY 的平摊代价仅为 $O(1)$。先来确定其实际代价。FIB-HEAP-DECREASE-KEY 过程要花 $O(1)$时间，再加上级联切断的时间。假设在一次给定的 FIB-HEAP-DECREASE-KEY 的调用中，要递归调用 c 次 CASCADING-CUT。每一次 CASCADING-CUT 的调用(不包括递归调用)的时间为 $O(1)$。这样，FIB-HEAP-DECREASE-KEY 的实际代价，包括所有递归调用为 $O(c)$。

下一步来计算势的变化。设 H 表示 FIB-HEAP-DECREASE-KEY 操作之前的斐波那契堆。对 CASCADING-CUT 的每次递归调用(除了最后一次外)切断一个被标记结点并清除标记位。在此之后，共有 $t(H)+c$ 棵树(原来的 $t(H)$棵树，由级联切断所产生的 $c-1$ 棵树，以及以 x 为根

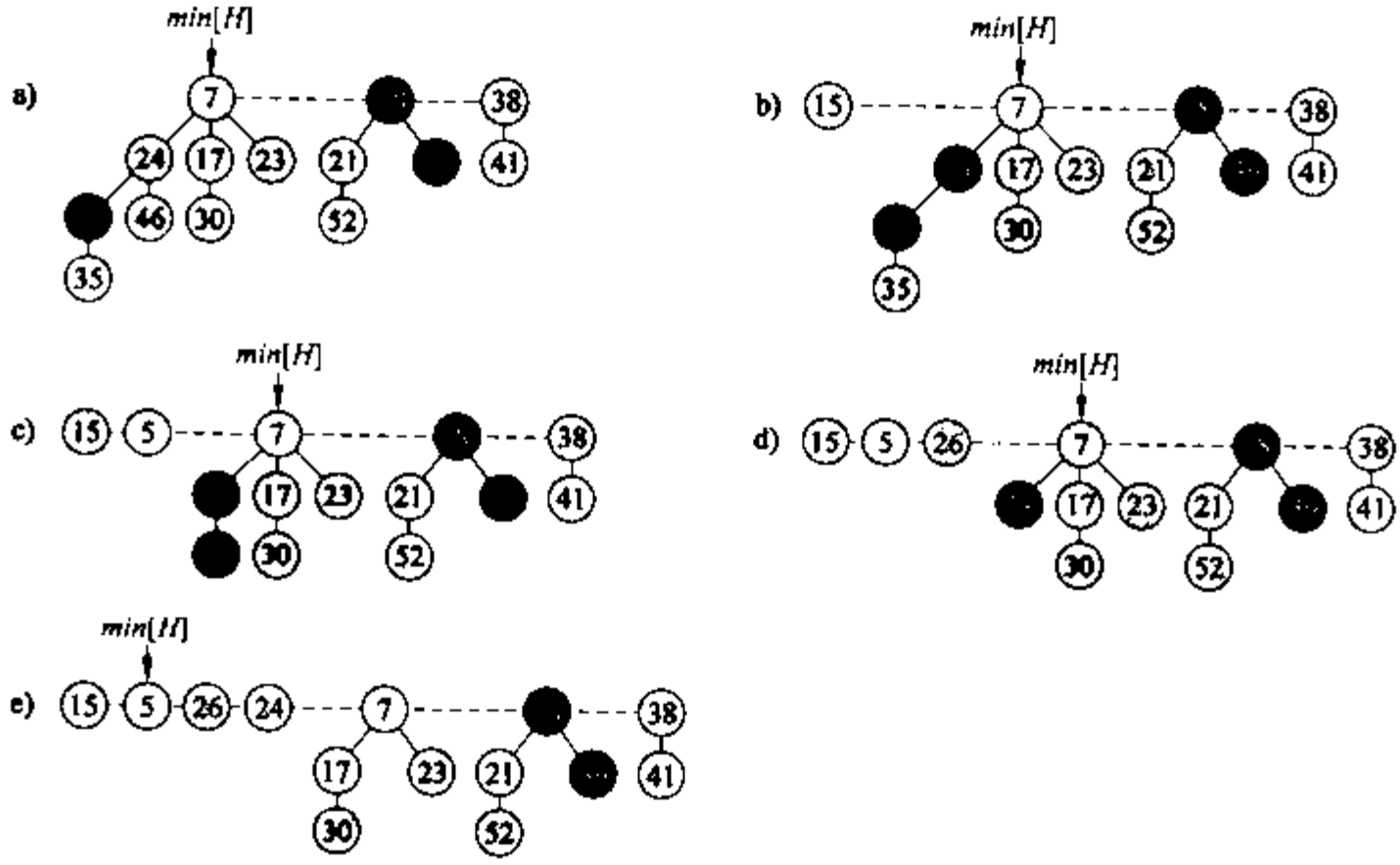


图 20-4　对 FIB-HEAP-DECREASE-KEY 的两次调用。a)初始斐波那契堆。b)关键字为 46 的结点的关键字降为 15。该结点成为一个根，且其先前为被标记的父结点(关键字为 24)也被加上了标记。c)-e)关键字为 35 的结点的关键字被降为 5。在 c)中，现具有关键字 5 的结点成为一个根，其父结点(关键字为 26)被标记，故发生级联切断。在 d)中，关键字为 26 的结点与其父结点间的联系被切断，成为一个未加标记的根。这样就发生另一次级联切断，因为关键字为 24 的结点也被标记了。在 e)中，这个结点与其父结点的联系被切断，成为一个无标记的根。在这一点上级联切断停止，因为关键字为 7 的结点是个根。(即使这个结点不是个根，级联切断也要停止，因为它未被标记。)FIB-HEAP-DECREASE-KEY 操作的结果示于 e)中，其中 $min[H]$指向新的最小结点

491

的树)，和至多 $m(H)-c+2$ 个加标记的结点($c-1$ 个结点被级联切断消除标记，而最后一次调用 CASCADING-CUT 则可能给某一结点加上标记)。因此，势的改变至多为

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H)) = 4-c$$

这样，FIB-HEAP-DECREASE-KEY 的平摊代价至多为

$$O(c)+4-c = O(1)$$

因为我们可以扩大势的单位以支配 $O(c)$中隐含的常数。

读者到现在就应该清楚了为什么在定义势函数时，要包括一个标记结点数两倍的项了。当一个有标记的结点 y 在级联切断中被切断时，它的标记位也被清掉了，故势就减少了 2。一个单位的势支付切断和标记位的清除，另一单位势补偿了因结点 y 成为根而增加的势。

删除一个结点

如以下伪代码所示，很容易在 $O(D(n))$的平摊时间内，从一个包含 n 个结点的斐波那契堆中删除一个结点。假定在斐波那契堆中任何关键字的当前值都不为$-\infty$。

```
FIB-HEAP-DELETE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, -∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE 与 BINOMIAL-HEAP-DELETE 是类似的。它将唯一的很小的关键字 $-\infty$ 赋给 x，使其成为斐波那契堆中的最小结点。然后，由 FIB-HEAP-EXTRACT MIN 过程将结点 x 从斐波那契堆中删除。FIB-HEAP-DELETE 的平摊时间为 FIB-HEAP-DECREASE-KEY 的 $O(1)$ 平摊时间与 FIB-HEAP-EXTRACT-MIN 的 $O(D(n))$ 的平摊时间之和。因为在 20.4 节将
492 看到 $D(n)=O(\lg n)$，FIB-HEAP-DELETE 的平摊时间是 $O(\lg n)$。

练习

20.3-1 假设一个斐波那契堆中的某个根 x 是有标记的。请解释 x 是如何成为有标记的根。另说明 x 有无标记对分析来说没有影响，即使它不是先被链接到另一个结点，然后又失去一个子结点的根。

20.3-2 使用聚集分析来证明 FIB-HEAP-DECREASE-KEY 的平摊时间 $O(1)$ 是每个操作的平均代价。

20.4 最大度数的界

为证明 FIB-HEAP-EXTRACT-MIN 和 FIB-HEAP-DELETE 的平摊时间为 $O(\lg n)$，我们必须证明，在包含 n 个结点的斐波那契堆中，任意结点的度数的上界 $D(n)$ 为 $O(\lg n)$。根据练习 20.2-3，当斐波那契堆中的所有树均为无序二项树时，$D(n)=\lfloor \lg n \rfloor$。但是，FIB-HEAP-DECREASE-KEY 中发生的切割可能引起斐波那契堆中的树违反无序二项树性质。这一节里，要证明因为一旦某结点失去两个孩子后，就将它与它的父结点之间切断，故 $D(n)$ 为 $O(\lg n)$。特别地，我们要证明 $D(n) \leqslant \lfloor \log_\phi n \rfloor$，其中 $\phi=(1+\sqrt{5})/2$。

分析的关键如下：对斐波那契堆的每个结点 x，定义 size(x) 为以 x 为根的子树中包括 x 在内的所有结点个数（请注意 x 无需在根表中——它可以是任何结点）。我们将证明 size(x) 为 $degree[x]$ 的幂。请记住，$degree[x]$ 始终是 x 的度数的准确计数。

引理 20.1　设 x 为斐波那契堆中任一结点，且假设 $degree[x]=k$。设 y_1，y_2，…，y_k 表示按与 x 链接的次序排列的 x 的子女，从最早的到最迟的，则对 $i=2$，3，…，k，有 $degree[y_1] \geqslant 0$ 和 $degree[y_i] \geqslant i-2$。

证明：很明显，$degree[y_1] \geqslant 0$。

对 $i \geqslant 2$，注意到当 y_i 链接到 x 上时，y_1，y_2，…，y_{i-1} 都是 x 的子女，故必有 $degree[x]=i-1$。又仅当 $degree[x]=degree[y_i]$ 时，才将结点 y_i 链接到 x 上，故这时又必有 $degree[y_i]=i-1$。从此之后，结点 y_i 至多失去了一个孩子，因为如果它失去了两个孩子，它将被从 x 处切
493 断。所以有 $degree[y_i] \geqslant i-2$。（证毕）■

最后一部分的分析说明了术语"斐波那契堆"的来源。回顾在 3.2 节通过递归定义的第 k 个斐波那契数，其中 $k=0$，1，2，…：

$$F_k = \begin{cases} 0 & \text{如果 } k=0 \\ 1 & \text{如果 } k=1 \\ F_{k-1}+F_{k-2} & \text{如果 } k \geqslant 2 \end{cases}$$

下面的引理给出了 F_k 的另一种表示。

引理 20.2　对所有整数 $k \geqslant 0$，

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

证明：对 k 进行归纳。当 $k=0$ 时，

$$1+\sum_{i=0}^{0} F_i = 1+F_0 = 1+0 = 1 = F_2$$

做归纳假设 $F_{k+1} = 1+\sum_{i=0}^{k-1} F_i$，有

$$F_{k+2} = F_k + F_{k+1} = F_k + \left(1+\sum_{i=0}^{k-1} F_i\right) = 1+\sum_{i=0}^{k} F_i$$ ■

下面的引理及其推论完成了全部分析。它们用到了不等式(在练习 3.2-7 中证明)

$$F_{k+2} \geqslant \phi^k$$

其中 ϕ 为黄金分割率，在(3.22)式中定义为 $\phi=(1+\sqrt{5})/2=1.618\,03\cdots$。 494

引理 20.3　设 x 为斐波那契堆的任一结点，且 $k=degree[x]$。那么，$size(x)\geqslant F_{k+2}\geqslant\phi^k$，其中 $\phi=(1+\sqrt{5})/2$。

证明：设 s_k 为所有满足 $degree[z]=k$ 的结点 z 中，$size(z)$ 的最小可能值。显然，$s_0=1$，$s_1=2$ 以及 $s_2=3$。s_k 至多为 $size(x)$，而且，显然 s_k 的值随着 k 单调递增。和在引理 20.1 中一样，设 y_1，y_2，…，y_k 表示 x 的各子女，且按它们与 x 链接的次序排列。为计算 $size(x)$ 的下界，x 本身和第一个孩子 y_1($size(y_1)\geqslant 1$)各算一个，有

$$size(x) \geqslant s_k = 2+\sum_{i=2}^{k} s_{degree[y_i]} \geqslant 2+\sum_{i=2}^{k} s_{i-2}$$

其中，最后一行应用了引理 20.1($dgree[y_i]\geqslant i-2$)以及 s_k 的单调性($s_{degree[y_i]}\geqslant s_{i-2}$)。

现在我们通过对 k 的归纳，来证明对所有非负整数 k，$s_k\geqslant F_{k+2}$。基是 $k=0$ 和 $k=1$，显然成立。对归纳步骤，假定 $k\geqslant 2$ 和 $s_i\geqslant F_{i+2}$，其中 $i=0$，1，…，$k-1$，我们有

$$s_k \geqslant 2+\sum_{i=2}^{k} s_{i-2} \geqslant 2+\sum_{i=2}^{k} F_i = 1+\sum_{i=0}^{k} F_i = F_{k+2} \qquad \text{(根据引理 20.2)}$$

这样，我们就证明了 $size(x)\geqslant s_k\geqslant F_{k+2}\geqslant\phi^k$。 ■

推论 20.4　在一个包含 n 个结点的斐波那契堆中，结点的最大度数 $D(n)$ 为 $O(\lg n)$。

证明：设 x 为含 n 个结点的斐波那契堆中的任意结点，$k=degree[x]$。根据引理 20.3，有 $n\geqslant size(x)\geqslant\phi^k$。取以 ϕ 为底的对数，得 $k\leqslant\log_\phi n$。(实际上，因为 k 为整数，有 $k\leqslant\lfloor\log_\phi n\rfloor$。)因此，任意结点的最大度数 $D(n)$ 为 $O(\lg n)$。 ■ 495

练习

20.4-1　Pinocchio 教授声称，包含 n 个结点的斐波那契堆的高度为 $O(\lg n)$。请证明他是错的，即对于任意正整数 n，给出一个斐波那契堆操作序列，它创建一个仅包含一棵树的堆，该树是 n 个结点的线性链。

20.4-2　假设我们将级联切断规则加以推广，使得当某个结点 x 一失去其第 k 个孩子时，就将其与父结点的联系切断，此处 k 为常整数(20.3 节的规则中 $k=2$)。k 取什么值时，有 $D(n)=O(\lg n)$?

思考题

20-1　删除的另一种实现

Pisano 教授提出了 FIB-HEAP-DELETE 过程的如下变形，并声称当要删除的结点不是由 $min[H]$ 所指向的结点时，该算法运行得更快。

```
PISANO-DELETE(H, x)
1  if x=min[H]
2     then FIB-HEAP-EXTRACT-MIN(H)
3     else y ← p[x]
4          if y≠NIL
5             then CUT(H, x, y)
6                  CASCADING-CUT(H, y)
7          add x's child list to the root list of H
8          remove x from the root list of H
```

a)教授之所以宣称这个过程可以运行得更快，部分原因是他假设可以在 $O(1)$ 实际时间内执行第 7 行。他这个假设错在哪里？

b)请给出当 x 不是 $min[H]$ 时，PISANO-DELETE 的实际时间的一个好的上界。所给出的界应以 $degree[x]$ 以及对 CASCADING-CUT 过程的调用次数 c 来表示。

c)设 H' 为执行一次 PISANO-DELETE(H, x)后得到的斐波那契堆。假定结点 x 不是
496 一个根，请用 $degree[x]$，c，$t(H)$ 和 $m(H)$ 来表示 H' 的势的界。

d)证明即使当 $x\neq min[H]$ 时，PISANO-DELETE 的平摊时间在渐进上看也不比 FIB-HEAP-DELETE 好。

20-2　其他斐波那契堆的操作

我们希望增强斐波那契堆 H，使之支持两种新的操作，同时，还不改变其他斐波那契堆操作的平摊运行时间。

a)操作 FIB-HEAP-CHANGE-KEY(H, x, k)将结点 x 的关键字改变为 k。请给出这个操作的一个高效的实现。另根据 k 大于、小于或等于 $key[x]$ 等不同情况，来分析实现的平摊运行时间。

b)操作 FIB-HEAP-PRUNE(H, r)将 $\min(r, n[H])$ 个结点从 H 中删除。请给出这个操作的一个高效的实现。删除哪些结点是任意的。请分析所给出的实现的平摊运行时间。(提示：可能需要修改数据结构和势函数。)

本章注记

斐波那契堆最早是由 Fredman 和 Tarjan[98]提出的，在他们的文章中还描述了一些关于应用斐波那契堆的问题，如单源最短路径，每对最短路径，带权二部图匹配和最小生成树问题等。

随后，Driscoll，Gabow，Shrairman 和 Tarjan[81]设计出有别于斐波那契堆的“松散堆”。该堆有两种变形。一种与斐波那契堆具有相同的平摊时间界；另一种的 DECREASE-KEY 操作的最坏情况(不是平摊时间)运行时间为 $O(1)$，而且 EXTRACT-MIN 和 DELETE 的最坏情况运行时间为 $O(\lg n)$。松散堆在并行算法中，也要比斐波那契堆更优越一些。

在第 6 章的“本章注记”中，也提到了一些数据结构。当 EXTRACT-MIN 调用的返回值序列随时间单调递增，而且数据是某一特定范围内的整数时，这些数据结构也支持快速的
497 DECREASE-KEY 操作。

第 21 章　用于不相交集合的数据结构

在某些应用中，要将 n 个不同的元素分成一组不相交的集合。不相交集合上有两个重要操作，即找出给定的元素所属的集合和合并两个集合。为了使某种数据结构能够支持这两种操作，就需要对该数据结构进行维护；本章就要来讨论各种维护方法。

21.1 节描述不相交集合数据结构所支持的各种操作，并给出这种数据结构的一个简单应用。在 21.2 节中，介绍不相交集合的一种简单的链表实现。另一种更为有效的、采用有根树的表示方法将在 21.3 节中给出。采用树表示的运行时间在实践上来说是线性的，但从理论上来说是超线性的。21.4 节定义并讨论一种增长极快的函数及其增长极为缓慢的逆函数。在基于树的实现中，各操作的运行时间中都出现了该反函数。然后，再利用平摊分析方法，证明运行时间的一个上界是超线性的。

21.1　不相交集合上的操作

不相交集合数据结构(disjoint-set data structure)保持一组不相交的动态集合 $S=\{S_1, S_2, \cdots, S_k\}$。每个集合通过一个代表来识别，代表即集合中的某个成员。在某些应用中，哪一个成员被选作代表是无所谓的；我们关心的是如果寻找某一动态集合的代表两次，并且在两次寻找之间不修改集合，两次得到的答案应该是相同的。在另一些应用中，关于如何选择代表可能存在着预先说明的规则，例如选择集合中的最小元素(当然假定集合中的元素是可以排序的)。

如我们已经研究过的动态集合的其他实现中一样，集合中的每一个元素是由一个对象表示的。设 x 表示一个对象，我们希望支持以下操作：

MAKE-SET(x)：建立一个新的集合，其唯一成员(因而其代表)就是 x。因为各集合是不相交的，故要求 x 没有在其他集合中出现过。

UNION(x, y)：将包含 x 和 y 的动态集合(比如说 S_x 和 S_y)合并为一个新的集合(即这两个集合的并集)。假定在这个操作之前两个集合是不相交的。在经过此操作后，所得集合的代表可以是 $S_x \cup S_y$ 中的任何成员，但在 UNION 的很多实现中，都选择 S_x 或 S_y 的代表作为新的代表。由于要求各集合是不相交的，故我们“消除了”集合 S_x 和 S_y，把它们从 S 中删去。

FIND-SET(x)：返回一个指针，指向包含 x 的(唯一)集合的代表。

在本章中，将根据两个参数来分析不相交集合数据结构的运行时间。一个参数是执行 MAKE-SET 操作的次数 n，另一个参数即执行 MAKE-SET、UNION 和 FIND-SET 操作的总次数 m。因为各集合是不相交的，故每一个 UNION 操作就使得集合的个数减少 1。于是，在 $n-1$ 次 UNION 操作之后，仅留下了一个集合。也就是说，UNION 操作的次数至多为 $n-1$。请注意在总的操作次数 m 中，包括 MAKE-SET 操作的次数，因而有 $m \geqslant n$。假定在所有执行的操作中，n 个 MAKE-SET 操作是最先执行的。

不相交集合数据结构的一个应用

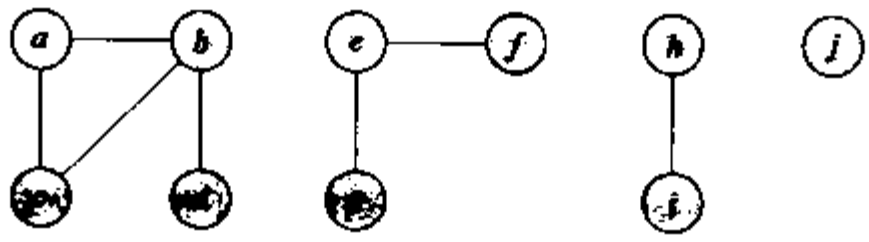
不相交集合数据结构有多种应用，其中之一是用于确定一个无向图中连通子图的个数(见 B.4 节)。例如，图 21-1a 显示一个包含四个连通子图的图。

在下面给出的过程 CONNECTED-COMPONENTS 中，利用了不相交集合操作来计算一个图的连通子图。一旦 CONNECTED-COMPONENTS 作为预处理步骤执行后，过程 SAME-

COMPONENT 回答两个顶点是否在同一连通子图的查询.[⊖](图 G 的顶点集用 $V[G]$表示，其边
499 集用 $E[G]$表示。)

```
CONNECTED-COMPONENTS(G)
1  for each vertex v∈V[G]
2       do MAKE-SET(v)
3  for each edge(u, v)∈E[G]
4       do if FIND-SET(u)≠FIND-SET(v)
5            then UNION(u, v)

SAME-COMPONENT(u, v)
1  if FIND-SET(u)=FIND-SET(v)
2    then return TRUE
3    else return FALSE
```



a)

处理的边	不相交集合组									
初始集合	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e, g)	{a}	{b, d}	{c}		{e, g}	{f}		{h}	{i}	{j}
(a, c)	{a, c}	{b, d}			{e, g}	{f}		{h}	{i}	{j}
(h, i)	{a, c}	{b, d}			{e, g}	{f}		{h, i}		{j}
(a, b)	{a, b, c, d}				{e, g}	{f}		{h, i}		{j}
(e, f)	{a, b, c, d}				{e, f, g}			{h, i}		{j}
(b, c)	{a, b, c, d}				{e, f, g}			{h, i}		{j}

b)

图 21-1　a)一个包含四个连通子图的图。这些子图是{a, b, c, d}，{e, f, g}，{h, i}和{j}。
b)处理每条边后的不相交集合组

过程 CONNECTED-COMPONENTS 开始时，将每个顶点 v 置于各自的集合中。然后，对每一条边(u, v)，它将包含 u 和包含 v 的集合进行合并。根据练习 21.1-2，在所有的边都被处理后，两个顶点在同一个连通子图中，当且仅当与之对应的对象在同一个集合中。这样，
500 CONNECTED-COMPONENTS 计算集合的方式，使得过程 SAME-COMPONENT 可以确定两个顶点是否在同一个连通子图中。图 21-1b 说明了 CONNECTED-COMPONENTS 是如何计算不相交集合的。

在连通子图算法的实际实现中，图和不相交集合数据结构的表示是需要互相引用的。亦即，

⊖ 当图的边集是"静态的"时(即不随时间而变化)，用深度优先搜索算法(见练习 22.3-1)可以更快地计算出各连通子图。然而，有时图中各边是"动态"加入的，因而，随着各条边的加入，就需要维护各连通子图。在这种情况下，与针对每一条新边运行一次新的深度优先搜索相比，此处给出的实现会更为有效。

表示一个顶点的对象会包含一个指向对应不相交集合对象的指针，反之亦然。这些编程细节与具体的实现语言有关，此处不再进一步讨论了。

练习

21.1-1 假设 CONNECTED-COMPONENTS 作用于一个无向图 $G=(V, E)$，此处 $V=\{a, b, c, d, e, f, g, h, i, j, k\}$，并且以如下次序对 E 的边进行处理：(d, i)，(f, k)，(g, i)，(b, g)，(a, h)，(i, j)，(d, k)，(b, j)，(d, f)，(g, j)，(a, e)，(i, d)。请列出在每次执行第 3～5 行后各连通子图中的顶点。

21.1-2 证明：在 CONNECTED-COMPONENTS 处理了所有的边后，两个顶点在同一个连通子图中，当且仅当它们在同一个集合中。

21.1-3 在 CONNECTED-COMPONENTS 作用于一个包含 k 个连通子图的无向图 $G=(V, E)$ 的过程中，要调用 FIND-SET 多少次？要调用 UNION 多少次？用 $|V|$、$|E|$ 和 k 来表达答案。

21.2　不相交集合的链表表示

要实现不相交集合数据结构，一种简单的方法是每一个集合都用一个链表来表示。每个链表中的第一个对象作为它所在集合的代表。链表中的每一个对象都包含一个集合成员、一个指向包含下一个集合成员的对象的指针，以及指向代表的指针。每个链表都含 *head* 指针和 *tail* 指针，*head* 指向链表的代表，*tail* 指向链表中最后的对象。图 21-2a 示出了两个集合的链表表示。在每个链表中，对象可以以任何次序出现(但要保证每个表中的第一个对象是所在集合的代表这一假设成立)。

在这种链表表示中，MAKE-SET 操作和 FIND-SET 操作都比较容易实现了，只需 $O(1)$ 的时间。为执行 MAKE-SET(x)操作，我们创建一个新的链表，其仅有对象为 x。对 FIND-SET(x)操作，只要返回由 x 指向代表的指针即可。 501

合并的一个简单实现

在 UNION 操作的实现中，最简单的是采用链表集合表示的实现，这种实现要比 MAKE-SET 或 FIND-SET 多不少的时间。如图 21-2b 所示，我们是这样来执行 UNION(x, y)的，就是将 x 所在的链表拼接到 y 所在链表的表尾。利用 y 所在链表的 *tail* 指针，可以迅速地找到应该在何处拼接 x 所在的链表。新集合的代表为原先包含 y 的集合的代表。不幸的是，对于原先 x 所在链表中的每一个对象，都需更新其指向代表的指针，这一更新过程所需时间与 x 所在表的长度成线性关系。

实际上，不难给出一个作用于 n 个对象上的、包含 m 个操作的序列，它需要 $\Theta(n^2)$ 时间。假设有对象 x_1，x_2，…，x_n。在这些对象上，如图 21-3 所示，执行 n 个 MAKE-SET 操作，后跟 $n-1$ 个 UNION 操作，因而有 $m=2n-1$。执行 n 个 MAKE-SET 操作所需时间为 $\Theta(n)$。因为第 i 个 UNION 操作更新了 i 个对象，故 $n-1$ 个 UNION 操作所更新的对象总数为

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

总的操作数为 $2n-1$，因而，平均来看，每个操作需要 $\Theta(n)$ 的时间。亦即，一个操作的平摊时间为 $\Theta(n)$。

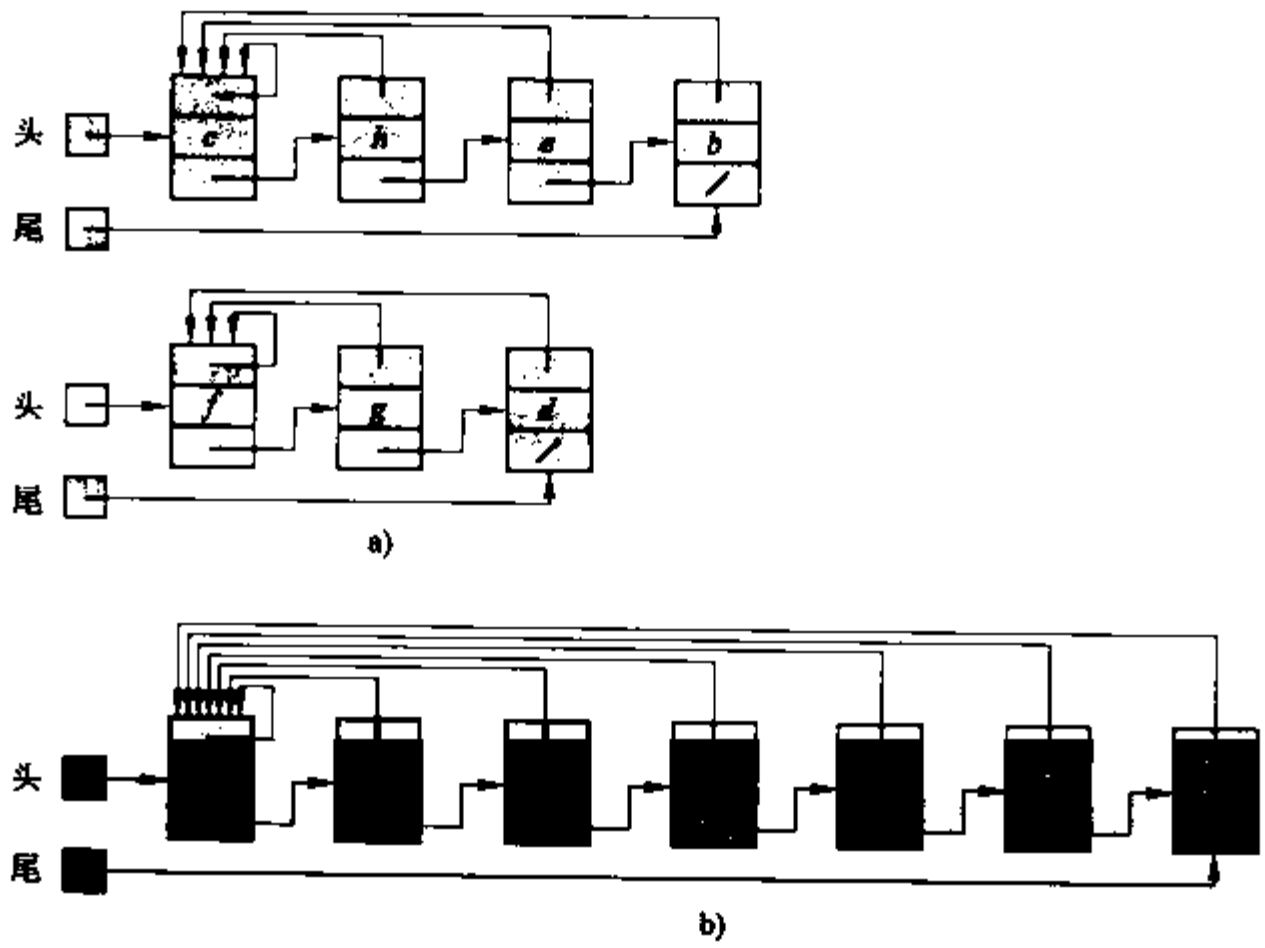


图 21-2　a)两个集合的链表表示。其中一个集合包含对象 b、c、e 和 h，代表为 c；另一个集合包含对象 d、f 和 g，代表为 f。链表中的每一个对象都包含一个集合成员、一个指向表中下一对象的指针，以及指向表中第一个对象(即代表)的指针。每一个链表都有指针 *head* 和 *tail*，它们分别指向链表中的第一个和最后一个对象。b)UNION(e，g)的结果。在结果集合中，代表为 f

操作	更新的对象数
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_1，x_2)	1
UNION(x_2，x_3)	2
UNION(x_3，x_4)	3
⋮	⋮
UNION(x_{n-1}，x_n)	$n-1$

图 21-3　利用链表集合表示和 UNION 操作的简单实现。在 n 个对象上的 $2n-1$ 个操作需要 $\Theta(n^2)$的时间，即平均每个操作需要 $\Theta(n)$ 时间

一种加权合并启发式策略

在最坏情况下，根据上面给出的 UNION 过程的实现，每次调用这一过程都平均需要 $\Theta(n)$ 的时间，这是因为我们可能是将一个较长的表拼到一个较短的表上；在这种情况下，还必须更新较长表中每个对象的指向代表的指针。现在，假设每个表中还包括了表的长度(这很容易维护)，并且总是把较短的表拼到较长的表上去；如果两个表一样长的话，可以以任意顺序拼接。利用这种简单的*加权合并启发式策略*(weighted-union heuristic)，如果两个集合都有 $\Omega(n)$个成员的话，

一次 UNION 操作仍会需要 $\Theta(n)$时间。然而，根据下面的定理，m 个 MAKE-SET、UNION 和 FIND-SET 操作的一个序列(其中有 n 个是 MAKE-SET 操作)要花 $O(m+n\lg n)$的时间。

502 ~ 503

定理 21.1　利用不相交集合的链表表示和加权合并启发式，一个包括 m 个 MAKE-SET、UNION 和 FIND-SET 操作(其中有 n 个是 MAKE-SET 操作)的序列所需时间为 $O(m+n\lg n)$。

证明：首先，对一个大小为 n 的集合中的每个对象，计算该对象指向代表的指针被更新次数的一个上界。考虑一个固定的对象 x。我们知道，每当 x 的代表指针被更新时，x 必是从更小的集合中开始。因此，x 的代表指针被第一次更新后，结果集合中必至少含有两个元素。类似地，下一次 x 的代表指针被更新后，结果集合中必至少含有四个元素。继续下去，可以注意到，对任意 $k\leqslant n$，在 x 的代表指针被更新 $\lceil\lg k\rceil$ 次后，结果集合中必至少含有 k 个元素。又由于最大的集合至多包含 n 个元素，故在所有的 UNION 操作被执行后，每个对象的代表指针至多被更新了 $\lceil\lg k\rceil$ 次。另外，还必须将更新链表的 *head* 和 *tail* 指针及链表长度的时间代价考虑在内，每一 UNION 操作仅需 $\Theta(1)$的时间。因而，更新 n 个对象所用的总时间为 $O(n\lg n)$。

m 个操作构成的序列所需时间也可以很容易地求出。每一个 MAKE-SET 和 FIND-SET 操作需要 $O(1)$时间，共有 $O(m)$次这样的操作，故整个操作序列所需的总时间为 $O(m+n\lg n)$。 ■

练习

21.2-1　请写出采用链表表示和加权合并启发式策略时，MAKE-SET、FIND-SET 和 UNION 操作的伪代码。假定每个对象 x 都有一个属性 $rep[x]$，它指向 x 所在集合的代表；每个集合 S 都有属性 $head[S]$、$tail[S]$和 $size[S]$($size[S]$等于链表的长度)。

21.2-2　请给出下面的程序中，执行 FIND-SET 操作后的数据结构及返回的答案。采用加权合并启发式策略的链表表示。

504

```
1  for i ← 1 to 16
2      do MAKE-SET(x_i)
3  for i ← 1 to 15 by 2
4      do UNION(x_i, x_{i+1})
5  for i ← 1 to 13 by 4
6      do UNION(x_i, x_{i+2})
7  UNION(x_1, x_5)
8  UNION(x_11, x_13)
9  UNION(x_1, x_10)
10 FIND-SET(x_2)
11 FIND-SET(x_9)
```

假定如果包含 x_i 的集合与包含 x_j 的集合大小一样，则操作 UNION(x_i，x_j)将 x_i 所在的表拼接到 x_j 所在的表上。

21.2-3　对定理 21.1 的证明加以改造，使得 MAKE-SET 和 FIND-SET 操作有平摊时间界 $O(1)$。对于采用了链表表示和加权合并启发式策略的 UNION 操作，有界 $O(\lg n)$。

21.2-4　对于图 21-3 中操作序列的运行时间，给出其紧确的渐近界。假定采用的是链表表示和加权合并启发式策略。

21.2-5　当采用链表表示时，给出对 UNION 的一个简单改动，使得无需让 *tail* 指针指向每个链表中的最后一个对象。无论是否采用了加权合并启发式策略，你所做的修改不应改变 UNION 过程的渐近运行时间。(提示：不要将一个表整个儿地拼接到另一个表的后面，而可以让两个表中的元素交叉地合并起来。)

21.3　不相交集合森林

在不相交集合的另一种更快的实现中，用有根树来表示集合，树中的每个结点都包含集合的一个成员，每棵树表示一个集合。在一个如图 21-4a 所示的不相交集合森林(disjoint-set forest)中，每个成员仅指向其父结点。每棵树的根包含了代表，并且是它自己的父结点。下面将看到，尽管
505 采用了这种表示的直观算法并不比采用链表表示的算法更快，但是，通过引入两种启发式策略("按秩合并"和"路径压缩")，就可以获得目前已知的、渐近意义上最快的不相交集合数据结构了。

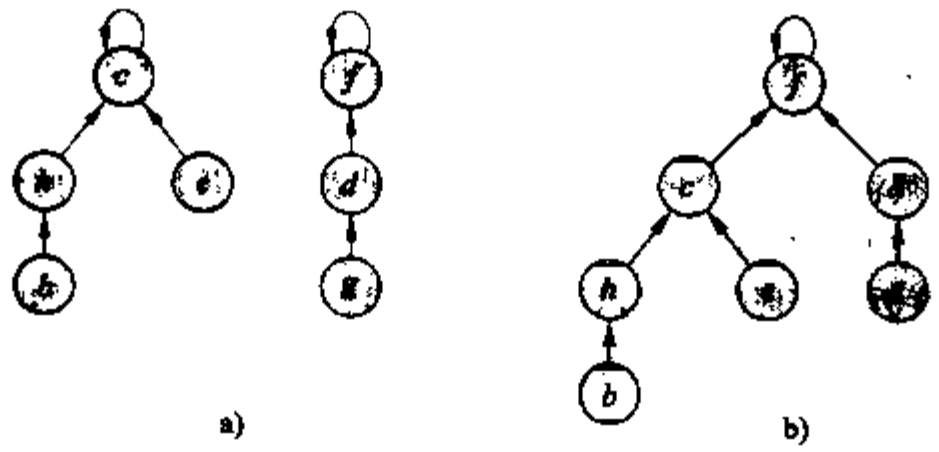


图 21-4　一个不相交集合森林。a)两棵表示图 21-2 中两个集合的树。左边的树表示集合$\{b, c, e, h\}$，其中 c 为代表；右边的树表示集合$\{d, f, g\}$，其中 f 为代表。b)UNION(e, g)的结果

执行以下三种不相交集合操作。MAKE-SET 创建一棵仅包含一个结点的树。在执行 FIND-SET 操作时，要沿着父结点指针一直找下去，直至找到树根为止。在这一查找路径上访问过的所有结点构成了查找路径(find path)。UNION 操作(如图 21-4b 中所示)使得一棵树的根指向另一棵树的根。

改进运行时间的启发式策略

到目前为止，我们还没有对链表实现做出改进。一个包含 $n-1$ 次 UNION 操作的序列可能会构造出一棵为 n 个结点的线性链的树。但是，通过采用两种启发式策略，可以获得一个几乎与总的操作数 m 成线性关系的运行时间。

第一种启发式是按秩合并(union by rank)，它与我们用于链表表示中的加权合并启发式是相似的。其思想是使包含较少结点的树的根指向包含较多结点的树的根。我们并不显式地记录以每个结点为根的子树的大小，而是采用了一种能够简化分析的方法。对每个结点，用秩表示结点高度的一个上界。在按秩合并中，具有较小秩的根在 UNION 操作中要指向具有较大秩的根。

第二种启发式即路径压缩(path compression)，它非常简单而有效。如图 21-5 中所示，在 FIND-SET 操作中，利用这种启发式策略，来使查找路径上的每个结点都直接指向根结点。路径
506 压缩并不改变结点的秩。

不相交集合森林的伪代码

为了实现一个按秩合并启发式的不相交集合森林，要记录下秩的变化。对每个结点 x，有一个整数 $rank[x]$，它是 x 的高度(从 x 到其某一后代叶结点的最长路径上边的数目)的一个上界。当由 MAKE-SET 创建了一个单元集时，对应的树中唯一结点的初始秩为 0，每个 FIND-SET 操作不改变任何秩。当对两棵树应用 UNION 时，有两种情况，具体取决于根是否有相等的秩。当两个秩不相等时，我们使具有较高秩的根成为具有较低秩的根的父结点，但秩本身保持不变。当两个秩相同时，任选一个根作为父结点，并增加其秩的值。

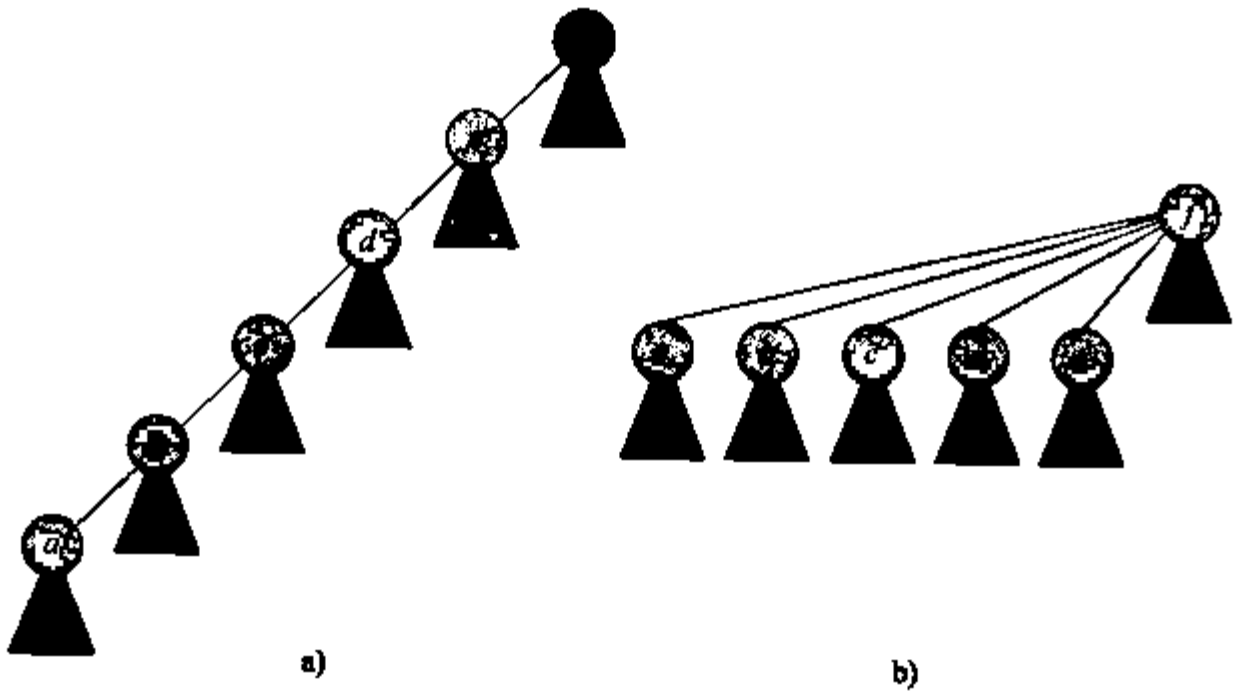


图 21-5　操作 FIND-SET 过程中的路径压缩。箭头和根结点处的自回路被略去了。a)一棵表示执行 FINS-SET(a)前某一集合的树。其中三角表示子树，其根为所示结点。每个结点都有一个指向其父结点的指针。b)在执行 FIND-SET(a)之后的同一集合。此时，查找路径上的每个结点都直接指向根

这种方法可以用以下的伪代码来表示。其中，$p[x]$表示 x 的父结点，LINK 过程是由 UNION 调用的一个子过程，它以指向两个根的指针作为输入。

507

```
MAKE-SET(x)
1  p[x] ← x
2  rank[x] ← 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  If rank[x]>rank[y]
2      then p[y] ← x
3      else p[x] ← y
4          if rank[x]=rank[y]
5              then rank[y] ← rank[y]+1
```

带路径压缩的 FIND-SET 过程也是相当简单的：

```
FIND-SET(x)
1  if x≠p[x]
2     then p[x] ← FIND-SET(p[x])
3  return p[x]
```

过程 FIND-SET 是一种两趟方法(two-pass method)：一趟是沿查找路径上升，直至找到根；第二趟是沿查找路径下降，以便更新每个结点，使之直接指向根。对 FIND-SET(x)的每一次调用，都会在第 3 行返回 $p[x]$。如果 x 为根，则不执行第 2 行，返回 $p[x]=x$。这种情况下递归结束。否则，执行第 2 行，且参数为 $p[x]$的递归调用返回一个指向根的指针。第 2 行更新结点 x，使之直接指向根，并在第 3 行返回这个指针。

启发式策略对运行时间的影响

如果将按秩合并或路径压缩分开来使用的话，它们都能改善不相交集合森林操作的运行时间；如果将这两种启发式合起来使用，则改善的幅度更大。单独来看，按秩合并产生的运行时间为 $O(m\lg n)$(见练习 21.4-4)，这个界是紧确的(见练习 21.3-3)。如果有 n 个 MAKE-SET 操作(因而至多有 $n-1$ 个 UNION 操作)和 f 个 FIND-SET 操作，则单独应用路径压缩启发式的话，可以得到最坏情况运行时间 $\Theta(n+f\cdot(1+\log_{2+f/n}n))$。此处对这一结论不加以证明。

508 当同时使用按秩合并和路径压缩时，最坏情况运行时间为 $O(m\alpha(n))$，其中 $\alpha(n)$是一个增长极其缓慢的函数，其定义将在 21.4 节中给出。在任意可想像的不相交集合数据结构的应用中，都会有 $\alpha(n)\leqslant 4$。因此，在各种实际情况中，可以把这个运行时间看作与 m 成线性关系。在 21.4 节中，要给出关于这个上界的证明。

练习

21.3-1 用按秩合并和路径压缩启发式的不相交集合森林来重做练习 21.2-2。

21.3-2 写出 FIND-SET 的路径压缩的非递归版本。

21.3-3 请给出一个包含 m 个 MAKE-SET、UNION 和 FIND-SET 操作的序列(其中 n 个是 MAKE-SET 操作)，使得采用按秩合并时，这一操作序列的时间代价为 $\Omega(m\lg n)$。

***21.3-4** 证明：在采用了按秩合并和路径压缩时，任意一个包含 m 个 MAKE-SET、FIND-SET 和 LINK 操作的序列(其中所有 LINK 操作出现于 FIND-SET 操作之前)需要 $O(m)$的时间。在同样情况下，如果仅用路径压缩启发式呢?

*21.4　带路径压缩的按秩合并的分析

21.3 节中提到过，对作用于 n 个元素上的 m 个不相交集合操作，联合使用按秩合并和路径压缩启发式的运行时间为 $O(m\alpha(n))$。本节要看看函数 α 增长到底多慢。然后，利用平摊分析中的势方法来证明这一运行时间。

一个增长极快的函数及其增长极慢的逆函数

对整数 $k\geqslant 0$ 和 $j\geqslant 1$，定义函数 $A_k(j)$为

509
$$A_k(j)=\begin{cases}j+1, & \text{如果 } k=0\\ A_{k-1}^{(j+1)}(j), & \text{如果 } k\geqslant 1\end{cases}$$

其中表达式 $A_{k-1}^{(j+1)}(j)$采用了 3.2 节中给出的函数迭代表示记号。具体来说，有 $A_{k-1}^{(0)}(j)=j$ 和 $A_{k-1}^{(i)}(j)=A_{k-1}(A_{k-1}^{(i-1)}(j))$，$i\geqslant 1$。称参数 k 为函数 A 的级(level)。

函数 $A_k(j)$随 j 和 k 严格地递增。为了搞清楚这个函数的增长速度有多快，我们首先来获得 $A_1(j)$和 $A_2(j)$的闭合形式表示。

引理 21.2　对任意整数 $j\geqslant 1$，有 $A_1(j)=2j+1$。

证明：首先，通过对 i 归纳来证明 $A_0^{(i)}(j)=j+i$。对于归纳的基本情况，有 $A_0^{(0)}(j)=j=j+0$。在归纳步骤中，假设 $A_0^{(i-1)}(j)=j+(i-1)$。这样就有 $A_0^{(i)}(j)=A_0(A_0^{(i-1)}(j))=(j+(i-1))+1=j+i$。最后，有 $A_1(j)=A_0^{(j+1)}(j)=j+(j+1)=2j+1$。 ■

引理 21.3　对任意整数 $j\geqslant 1$，有 $A_2(j)=2^{j+1}(j+1)-1$。

证明：首先，通过对 i 进行归纳来证明 $A_1^{(i)}(j)=2^i(j+1)-1$。对归纳的基本情况，有 $A_1^{(0)}(j)=j=2^0(j+1)-1$。在归纳步骤中，假设 $A_1^{(i-1)}(j)=2^{i-1}(j+1)-1$，则有 $A_1^{(i)}(j)=$

$A_1(A_1^{(i-1)}(j))=A_1(2^{i-1}(j+1)-1)=2\cdot(2^{i-1}(j+1)-1)+1=2^i(j+1)-2+1=2^i(j+1)-1$。

最后，有 $A_2(j)=A_1^{(j+1)}(j)=2^{j+1}(j+1)-1$。■

现在，只要在 $k=0$，1，2，3，4 这几个级别上检查 $A_k(1)$，就可以看出 $A_k(j)$增长得有多快了。根据 $A_0(k)$的定义和上面的引理，有 $A_0(1)=1+1=2$，$A_1(1)=2\times1+1=3$，$A_2(1)=2^{1+1}\times(1+1)-1=7$。此外，还有：

$$A_3(1)=A_2^{(2)}(1)=A_2(A_2(1))=A_2(7)=2^8\cdot8-1=2^{11}-1=2047$$

以及：

$$A_4(1)=A_3^{(2)}(1)=A_3(A_3(1))=A_3(2047)=A_2^{(2048)}(2047)\gg A_2(2047) \qquad 510$$

$$=2^{2048}\cdot2048-1>2^{2048}=(2^4)^{512}=16^{512}\gg10^{80}$$

这就是可观察到的宇宙中估计的原子数量。

对整数 $n\geqslant0$，定义函数 $A_k(n)$的逆函数如下：

$$\alpha(n)=\min\{k:A_k(1)\geqslant n\}$$

也就是说，$\alpha(n)$是使得函数 $A_k(1)$至少为 n 的最低级别 k。根据上面 $A_k(1)$的值，可以知道：

$$\alpha(n)=\begin{cases}0, & \text{对 } 0\leqslant n\leqslant2\\ 1, & \text{对 } n=3\\ 2, & \text{对 } 4\leqslant n\leqslant7\\ 3, & \text{对 } 8\leqslant n\leqslant2047\\ 4, & \text{对 } 2048\leqslant n\leqslant A_4(1)\end{cases}$$

只有对那些大得不切实际的 n 值(大于 $A_4(1)$，这是一个巨大的数)，才会有 $\alpha(n)>4$，因而对所有实际的应用来说，都有 $\alpha(n)\leqslant4$。

秩的性质

在本节余下的内容里，对于按秩合并和路径压缩的不相交集合操作，我们要证明其运行时间的一个 $O(m\alpha(n))$界。为了证明这个界，首先要证明秩的一些简单性质。

引理 21.4　对所有的结点 x，有 $rank[x]\leqslant rank[p[x]]$，如果 $x\neq p[x]$则不等号严格成立。$rank[x]$的初始值为 0，并随时间而增长，直到 $x\neq p[x]$；从此以后，$rank[x]$就不再变化。$rank[p[x]]$的值是时间的单调递增函数。

证明：利用 21.3 节给出的 MAKE-SET、UNION 和 FIND-SET 的实现，对操作的次数进行归纳即可证明本引理。这一证明过程留作练习 21.4-1。■

推论 21.5　在从任何一个结点指向根的路径上，结点的秩是严格递增的。■ 511

引理 21.6　每个结点的秩至多为 $n-1$。

证明：每个结点的秩从 0 开始，且仅当执行了 LINK 操作时才会增加。由于 UNION 操作至多有 $n-1$ 个，因而 LINK 操作也至多只有 $n-1$ 个。又由于每个 LINK 操作或者不改变任何的秩，或者只将某个结点的秩增加 1，因此，所有的秩至多为 $n-1$。■

引理 21.6 提供了一个关于结点秩的较弱的界。事实上，每个结点的秩至多为 $\lfloor\lg n\rfloor$(见练习 21.4-2)。但是，引理 21.6 所给出的这一较松的界对我们来说足够了。

时间界的证明

我们将利用平摊分析中的势方法(potential method，见 17.3 节)来证明 $O(m\alpha(n))$时间界。在做平摊分析时，为方便起见，假定调用的是 LINK 操作，而不是 UNION 操作。亦即，因为 LINK 过程的参数是指向两个根的指针，故假定 FIND-SET 操作是独立执行的。下面的引理说明了即使将因 UNION 调用而导致的额外 FIND-SET 操作也算进来，渐近的运行时间还是一样的。

引理 21.7　假定有一个由 m' 个 MAKE-SET、UNION 和 FIND-SET 操作构成的操作序列 S'，将其转换成一个新的操作序列 S，它由 m 个 MAKE-SET、LINK 和 FIND-SET 操作所构成，转换方法是将每一个 UNION 操作转换成两个 FIND-SET 操作，后跟一个 LINK 操作。那么，如果操作序列 S 的运行时间为 $O(m\alpha(n))$的话，操作序列 S'的运行时间即为 $O(m'\alpha(n))$。

证明：由于操作序列 S'中的每一个 UNION 操作都被转换成了 S 中的三个操作，因而有 $m' \leqslant m \leqslant 3m'$。因为 $m = O(m')$，如果转换后的序列 S 有界 $O(m\alpha(n))$的话，就意味着原序列 S'有界 $O(m'\alpha(n))$。 ■

在本节的余下部分里，将假定由 m'个 MAKE-SET、UNION 和 FIND-SET 操作所构成的初始序列已经被转换成了由 m 个 MAKE-SET、LINK 和 FIND-SET 操作所构成的序列。现在来证明转换后序列的一个 $O(m\alpha(n))$时间界，并利用引理 21.7 来证明包含 m'个操作的原序列的
512 $O(m'\alpha(n))$运行时间。

势函数

我们所采用的势函数(potential function)在 q 个操作之后，为不相交集合森林中的每个结点 x 都分配一个势 $\phi_q(x)$。将所有结点的势加起来，即可得到整个森林的势：$\Phi_q = \sum_x \phi(x)$，其中 Φ_q 表示整个森林在 q 次操作之后的势。在第一次操作之前，森林是空的，此时可以任意地设 $\Phi_0 = 0$。Φ_q 始终都是非负的。

$\phi_q(x)$的值取决于在第 q 次操作之后，x 是否是一个树根。如果是的，或者，如果 $rank[x] = 0$，则有 $\phi_q(x) = \alpha(n) \cdot rank[x]$。

现在，假定在第 q 次操作之后，x 并不是一个树根，且 $rank[x] \geqslant 1$。在可以定义 $\phi_q(x)$之前，需要定义两个关于 x 的辅助函数。首先，定义

$$\text{level}(x) = \max\{k : rank[p[x]] \geqslant A_k(rank[x])\}$$

亦即，level(x)是一个最大的级别 k，使得当将 A_k 作用于 x 的秩时，不大于 x 的父结点的秩。

我们称下式

$$0 \leqslant \text{level}(x) < \alpha(n) \tag{21.1}$$

是成立的，它可以如下推出：

$$\begin{aligned} rank[p[x]] &\geqslant rank[x] + 1 && \text{(根据引理 21.4)} \\ &= A_0(rank[x]) && \text{(根据 } A_0(j) \text{ 的定义)} \end{aligned}$$

这意味着 level(x)$\geqslant$0，从而有：

$$\begin{aligned} A_{\alpha(n)}(rank[x]) &\geqslant A_{\alpha(n)}(1) && \text{(因为 } A_k(j) \text{ 是严格递增的)} \\ &\geqslant n && \text{(根据 } \alpha(n) \text{ 的定义)} \\ &> rank[p[x]] && \text{(根据引理 21.6)} \end{aligned}$$

这蕴含着 level(x)$<\alpha(n)$。注意因为 $rank[p[x]]$随时间单调递增，故 level(x)也是随时间单调递增的。

第二个辅助函数是

$$\text{iter}(x) = \max\{i : rank[p[x]] \geqslant A^{(i)}_{\text{level}(x)}(rank[x])\}$$

亦即，iter(x)是首先将 $A_{\text{level}(x)}$应用于 x 的秩，并在获得一个大于 x 的父结点的秩的值之前，可以迭代地应用 $A_{\text{level}}(x)$的最大次数。

我们称下式

$$1 \leqslant \text{iter}(x) \leqslant rank[x] \tag{21.2}$$

是成立的，它可以如下导出：

$$\begin{aligned} rank[p[x]] &\geqslant A_{level(x)}(rank[x]) && \text{（根据 level}(x)\text{ 的定义）} \\ &= A^{(1)}_{level(x)}(rank[x]) && \text{（根据函数迭代的定义）} \end{aligned}$$

513

这蕴含着 $iter(x) \geqslant 1$，并且有：

$$\begin{aligned} A^{(rank[x]+1)}_{level(x)}(rank[x]) &= A_{level(x)+1}(rank[x]) && \text{（根据 } A_k(j) \text{ 的定义）} \\ &> rank[p[x]] && \text{（根据 level}(x)\text{ 的定义）} \end{aligned}$$

这意味着 $iter(x) \leqslant rank[x]$。注意因为 $rank[p[x]]$是随时间单调递增的，为了使 $iter(x)$能够递减，$level(x)$就必须增加。只要 $level(x)$保持不变，$iter(x)$就必须增加，或者保持不变。

在定义了这些辅助函数后，就可以来定义在 q 次操作之后结点 x 的势了：

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot rank[x] & \text{如果 } x \text{ 是一个根或 } rank[x] = 0 \\ (\alpha(n) - level(x)) \cdot rank[x] - iter(x) & \text{如果 } x \text{ 不是根且 } rank[x] \geqslant 1 \end{cases}$$

接下来的两个引理给出了结点势的一些有用性质。

引理 21.8　对每个结点 x 和所有操作的计数 q，有

$$0 \leqslant \phi_q(x) \leqslant \alpha(n) \cdot rank[x]$$

证明：如果 x 是个根或 $rank[x]=0$，则根据定义，有 $\phi_q(x)=\alpha(n) \cdot rank[x]$。现在，假设 x 不是根，且 $rank[x] \geqslant 1$。通过最大化 $level(x)$和 $iter(x)$，可以得到 $\phi_q(x)$的一个较低的界。根据界(21.1)，$level(x) \leqslant \alpha(n)-1$，根据界(21.2)，有 $iter(x) \leqslant rank[x]$。于是，

$$\phi_q(x) \geqslant (\alpha(n) - (\alpha(n) - 1)) \cdot rank[x] - rank[x] = rank[x] - rank[x] = 0$$

类似地，通过最小化 $level(x)$和 $iter(x)$，可以得到 $\phi_q(x)$的一个上界。根据界(21.1)可知 $level(x) \geqslant 0$，根据界(21.2)可知 $iter(x) \geqslant 1$。于是有：

$$\phi_q(x) \leqslant (\alpha(n) - 0) \cdot rank[x] - 1 = \alpha(n) \cdot rank[x] - 1 < \alpha(n) \cdot rank[x]$$ ■

势的变化和操作的平摊代价

现在来分析不相交集合操作是如何影响结点的势的。在理解了每个操作所引起的势的变化后，就可以确定每个操作的平摊代价。

514

引理 21.9　设 x 是一个非根结点，并假设第 q 次操作是 LINK 操作或 FIND-SET 操作。在第 q 次操作之后，$\phi_q(x) \leqslant \phi_{q-1}(x)$。此外，如果 $rank[x] \geqslant 1$，且 $level(x)$或 $iter(x)$因为第 q 次操作而发生了变化，则有 $\phi_q(x) \leqslant \phi_{q-1}(x) - 1$。亦即，$x$ 的势是不会增加的，并且，如果它有正的秩，同时 $level(x)$或 $iter(x)$发生了变化，则 x 的势至少要减少 1。

证明：因为 x 不是一个根，故第 q 次操作不会改变 $rank[x]$，又由于 n 在前 n 次 MAKE-SET 操作之后没有发生变化，因而 $\alpha(n)$也保持不变。于是，在计算 x 的势的公式中，在第 q 次操作之后，这些成分均保持不变。如果 $rank[x]=0$，则 $\phi_q(x)=\phi_{q-1}(x)=0$。现在，假设 $rank[x] \geqslant 1$。

前面说过，$level(x)$是随时间而单调递增的。如果第 q 次操作使得 $level(x)$保持不变，那么 $iter(x)$或者增加，或者保持不变。如果 $level(x)$和 $iter(x)$都保持不变，则 $\phi_q(x)=\phi_{q-1}(x)$。如果 $level(x)$没有变化，而 $iter(x)$增加了，则它至少要增加 1，因而有 $\phi_q(x) \leqslant \phi_{q-1}(x)-1$。

最后，如果第 q 次操作使 $level(x)$的值增加了，则增加的量至少为 1，因而$(\alpha(n) - level(x)) \cdot rank[x]$至少减少 $rank[x]$。由于 $level(x)$增加了，$iter(x)$的值可能会下降，但根据界(21.2)，这一下降至多为 $rank[x]-1$。于是，由于 $iter(x)$的变化而带来的势的增加，就小于由于 $level(x)$的变化而导致的势的减少，因而可以得出结论 $\phi_q(x) \leqslant \phi_{q-1}(x)-1$。 ■

下面要给出最后三个引理，它们说明了 MAKE-SET、LINK 和 FIND-SET 操作中每一个的平摊代价都是 $O(\alpha(n))$。回顾一下公式(17.2)可以知道，每个操作的平摊代价是其真实代价，再加上由于该操作而导致的势的增加量。

引理 21.10　每个 MAKE-SET 操作的平摊代价是 $O(1)$。

证明：假设第 q 个操作是 MAKE-SET(x)。这个操作生成一个秩为 0 的结点 x，因此有 $\phi_q(x)=0$。除此之外，没有别的秩或势方面的变化，因而有 $\Phi_q=\Phi_{q-1}$。注意到 MAKE-SET 操作的实际代价为 $O(1)$，从而完成对本引理的证明。■

515　引理 21.11　每个 LINK 操作的平摊代价是 $O(\alpha(n))$。

证明：假设第 q 个操作是 LINK(x, y)。LINK 操作的实际代价是 $O(1)$。不失一般性，假设这一 LINK 操作使 y 成为了 x 的父结点。

为了确定由于 LINK 操作而导致的势的变化，我们注意到势可能发生变化的结点只有 x 和 y，以及该操作之前 y 的子结点。下面我们将证明，由于 LINK 操作而导致势增加的结点只可能是 y，并且最多增加 $\alpha(n)$：

- 根据引理 21.9，对于那些在 LINK 操作之前为 y 的子女的任何一个结点来说，其势都不会由于该 LINK 操作而有所增加。
- 根据 $\phi_q(x)$的定义可以看出，由于 x 是第 q 次操作之前的一个根，故 $\phi_{q-1}(x)=\alpha(n)\cdot rank[x]$。如果 $rank[x]=0$，则 $\phi_q(x)=\phi_{q-1}(x)=0$。否则，

$$\begin{aligned}\phi_q(x)&=(\alpha(n)-\text{level}(x))\cdot rank[x]-\text{iter}(x)\\&<\alpha(n)\cdot rank[x]\quad\text{（根据不等式(21.1)和不等式(21.2)）}\end{aligned}$$

因为最后一个量是 $\phi_{q-1}(x)$，可以看出，x 的势是减少了。

- 由于在 LINK 操作之前 y 是个根，因而有 $\phi_{q-1}(y)=\alpha(n)\cdot rank[y]$。该 LINK 操作使 y 仍保持为根，并且，它或者保持 y 的秩不变，或者使 y 的秩增加 1。于是，有 $\phi_q(y)=\phi_{q-1}(y)$，或者有 $\phi_q(y)=\phi_{q-1}(y)+\alpha(n)$。

因此，由于 LINK 操作而带来的势的增加至多为 $\alpha(n)$。LINK 操作的平摊代价为

$$O(1)+\alpha(n)=O(\alpha(n))$$ ■

引理 21.12　每个 FIND-SET 操作的平摊代价为 $O(\alpha(n))$。

证明：假设第 q 次操作是一个 FIND-SET 操作，且查找路径上包含 s 个结点。FIND-SET 操作的实际代价是 $O(s)$。下面要证明没有结点的势会因为 FIND-SET 操作而增加，且在查找路径上，至少有 $\max(0,\ s-(\alpha(n)+2))$个结点的势会下降至少 1。

为了说明任何结点的势都没有增加，首先对除了根之外的其他所有结点应用引理 21.9。如果 x 为根结点，则它的势为 $\alpha(n)\cdot rank[x]$，它不会发生变化。

现在来证明至少有 $\max(0,\ s-(\alpha(n)+2))$个结点的势下降至少 1。设 x 为查找路径上的一个满足 $rank[x]>0$ 的结点，且在查找路径中的某处，x 后跟另一个非根结点 y，它在 FIND-SET 操作之前，满足 $\text{level}(y)=\text{level}(x)$。（在查找路径上，结点 y 并不一定要紧跟在结点 x 的后面。）在

516　查找路径上，除了至多 $\alpha(n)+2$ 个结点而外，其他的结点都满足关于 x 的这些限制。那些不满足这些限制的结点是查找路径上的第一个结点（如果该结点的秩为 0）、路径上的最后一个结点（即根结点）或路径上最后一个满足条件 $\text{level}(w)=k$ 的结点 w，此处 $k=0,\ 1,\ \cdots,\ \alpha(n)-1$。

为了证明 x 的势下降了至少 1，可以先将这样的一个结点 x 固定下来。设 $k=\text{level}(x)=\text{level}(y)$。在由 FIND-SET 操作引起的路径压缩之前，有：

$$rank[p[x]]\geqslant A_k^{\text{iter}(x)}(rank[x])\quad\text{（根据 iter}(x)\text{ 的定义）}$$

$$rank[p[y]]\geqslant A_k(rank[y])\quad\text{（根据 level}(y)\text{ 的定义）}$$

$$rank[y]\geqslant rank[p[x]]\quad\text{（根据推论 21.5，并且在查找路径上 }y\text{ 跟在 }x\text{ 的后面）}$$

将这些不等式组合到一起，并设 i 为路径压缩之前 $\text{iter}(x)$的值，有：

$$\begin{aligned} rank[p[y]] &\geqslant A_k(rank[y]) \\ &\geqslant A_k(rank[p[x]]) \quad \text{（因为 } A_k(j) \text{ 是严格递增）} \\ &\geqslant A_k(A_k^{\text{iter}(x)}(rank[x])) = A_k^{i+1}(rank[x]) \end{aligned}$$

因为路径压缩会使得 x 和 y 拥有同一父结点，我们知道，在路径压缩之后，$rank[p[x]]=rank[p[y]]$，且路径压缩不会使 $rank[p[y]]$下降。由于 $rank[x]$没有发生变化，在路径压缩之后，有 $rank[p[x]]\geqslant A_k^{i+1}(rank[x])$。于是，路径压缩将使得 iter($x$)增加(到至少 $i+1$)，或者使 level(x)增加(这在 iter(x)增加到至少 $rank[x]+1$ 时发生)。不管发生的是这两种情况中的哪一种，根据引理 21.9，都有 $\phi_q(x)\leqslant\phi_{q-1}(x)-1$。亦即，$x$ 的势下降了至少 1。

FIND-SET 操作的平摊代价为其实际代价加上势的变化量。这一操作的实际代价为 $O(s)$，我们在前面也已经证明了其势的减少量至少为 $\max(0, s-(\alpha(n)+2))$。于是，该操作的平摊代价至多为 $O(s)-(s-(\alpha(n)+2))=O(s)-s+O(\alpha(n))=O(\alpha(n))$，因为我们可以放大势的单位，以便忽略 $O(s)$中隐含的常数。 ■

将上面给出的几个引理综合在一起，就可以得到下面的定理。

定理 21.13　对于一个由 m 个 MAKE-SET、UNION 和 FIND-SET 操作所组成的操作序列(其中 n 个为 MAKE-SET 操作)，当在一个带按秩合并和路径压缩的不相交集合森林上执行时，最坏情况下的执行时间为 $O(m\alpha(n))$。

证明：根据引理 21.7、21.10、21.11 和引理 21.12 立即得证。 ■ 517

练习

21.4-1　证明引理 21.4。

21.4-2　证明：每个结点的秩都至多为 $\lfloor \lg n \rfloor$。

21.4-3　根据练习 21.4-2 的结论，对每个结点 x，存储 $rank[x]$需要多少位(bit)?

21.4-4　利用练习 21.4-2 的结果，对于带按秩合并、但不带路径压缩的不相交集合上的操作，简要地证明其运行时间为 $O(m\lg n)$。

21.4-5　Dante 教授认为，因为各结点的秩在一条指向根的路径上是严格递增的，故各结点的级别在该路径上应该是单调递增的。换句话说，如果 $rank(x)>0$，并且 $p[x]$不是一个根，则 level(x)$\leqslant$level($p[x]$)。Dante 教授的推理正确吗?

21.4-6　考虑函数 $\alpha'(n)=\min\{k: A_k(1)\geqslant\lg(n+1)\}$。证明：对于 n 的所有实际取值，有 $\alpha'(n)\leqslant 3$，并利用练习 21.4-2 的结果，说明应如何修改势函数参数，以便证明一个包含 m 个 MAKE-SET、UNION 和 FIND-SET 操作(其中 n 个为 MAKE-SET 操作)的序列，当在一个带按秩合并和路径压缩的不相交集合森林上执行时，其最坏情况运行时间为 $O(m\alpha'(n))$。

思考题

21-1　脱机最小值

脱机最小值问题(off-line minimum problem)是对 INSERT 和 EXTRACT-MIN 操作所作用的一个其元素取自域$\{1, 2, \cdots, n\}$的动态集合 T 加以维护。已知的是一个包含 n 个 INSERT 和 m 个 EXTRACT-MIN 调用的序列 S，其中$\{1, 2, \cdots, n\}$中的每一个关键字恰被插入一次。我们希望确定每次 EXTRACT-MIN 调用返回的是哪个关键字。特别地，我们希望对一个数组 $extracted[1..m]$进行填充，其中对 $i=1, 2, \cdots, m$，$extracted[i]$是由第 i 次 EXTRACT-MIN 调用所返回的关键字。该问题是“脱机”的，意即可以在确定任何返回的关键字之前处理整个序列 S。 518

a)在下面的脱机最小值问题的例子中，每个 INSERT 由一个数字表示，每个 EXTRACT-MIN 由字母 E 表示：

4，8，E，3，E，9，2，6，E，E，E，1，7，E，5

将正确的值填入 *extracted* 数组。

为了设计出解决此问题的算法，将序列 S 分成若干个同构的子序列。亦即，将 S 表示成：

$$I_1，E，I_2，E，I_3，\cdots，I_m，E，I_{m+1}$$

其中每个 E 表示一次 EXTRACT-MIN 调用，每个 I_j 表示一个(可能为空的)INSERT 调用序列。对每个子序列 I_j，开始时把由这些操作插入的关键字放入一个集合 K_j，如果 I_j 为空，则它也是空的。然后执行下面的过程。

```
OFF-LINE-MINIMUM(m, n)
1  for i ← 1 to n
2       do determine j such that i∈K_j
3          if j≠m+1
4            then extracted[j] ← i
5                 let l be the smallest value greater than j
                        for which set K_l exists
6                 K_l ← K_l ∪ K_j, destroying K_j
7  return extracted
```

b)证明由 OFF-LINE-MINIMUM 返回的数组 *extracted* 是正确的。

c)说明如何用不相交集合数据结构来有效地实现 OFF-LINE-MINIMUM。另给出该实现的最坏情况运行时间的一个紧确的界。

21-2　深度确定

在深度确定问题中，我们对以下三个操作所作用的一个有根树的森林 $\mathcal{F}=\{T_i\}$ 加以维护：

519

MAKE-TREE(v)：创建一棵包含唯一结点 v 的树。

FIND-DEPTH(v)：返回结点 v 在树中的深度。

GRAFT(r，v)：使结点 r(假定为某棵树的根)成为结点 v 的子结点(假定结点 v 在另一棵树中，它本身可能是、也可能不是一个根)。

a)假设我们采用的是类似于不相交集合森林的树表示：$p[v]$为结点 v 的父亲；如果 v 是根的话，$p[v]=v$。如果通过置 $p[r]\leftarrow v$ 来实现 GRAFT(r，v)，通过沿查找路径上升至根并返回所遇到的非 v 结点个数来实现 FIND-DEPTH(v)，证明一个包含 m 次 MAKE-TREE，FIND-DEPTH 和 GRAFT 操作序列的最坏情况运行时间为 $\Theta(m^2)$。

通过采用按秩合并和路径压缩启发式，可以缩短最坏情况运行时间。我们采用不相交集合森林 $S=\{S_i\}$，其中每个集合 S_i(它本身是棵树)与森林 $\mathcal{F}$ 中的一棵树 T_i 对应。然而，集合 S_i 中的树结构并不一定与 T_i 的结构对应。实际上，S_i 的实现并没有记录准确的父-子关系，但它使我们可以确定 T_i 中任意结点的深度。

这种表示的主要思想就是在每个结点 v 中记录一个“伪距离”$d[v]$，它被定义成使集合 S_i 中沿从 v 至根的路径上所有的伪距离之和等于 v 在 T_i 中的深度。也就是说，如果 S_i 中从 v 至根的路径为 v_0，v_1，$\cdots$，v_k，此处 $v_0=v$，且 v_k 为 S_i 的根，则 v 在 T_i 中的深度为 $\sum_{j=0}^{k} d[v_j]$。

b)给出 MAKE-TREE 的一种实现。

c)说明应如何修改 FIND-SET 以实现 FIND-DEPTH。所给出的实现应做路径压缩，且其运行时间应与查找路径长度成线性关系。要保证该实现能正确地更新伪距离。

d)说明应如何修改 UNION 和 LINK 过程以实现 GRAFT(r, v)过程，它合并分别包含 r 的 v 的集合。要确保所给出的实现能正确地更新伪距离。注意某一集合 S_i 的根不必为对应的树 T_i 的根。

e)对于一个包含 m 个 MAKE-TREE、FIND-DEPTH 和 GRAFT 操作(其中 n 个是 MAKE-TREE 操作)的序列，给出其在最坏情况下运行时间的一个紧确的界。 520

21-3　Tarjan 的脱机最小公共祖先算法

在一棵有根树 T 中，两个结点 u 和 v 的最小公共祖先(least common ancestor)是指这样的一个结点 w，它是 u 和 v 的祖先，并且在树 T 中具有最大深度。在脱机最小公共祖先问题中，给定的是一棵有根树 T 和一个由 T 中结点的无序对构成的任意集合 $P=\{(u, v)\}$，我们希望确定 P 中每个对的最小公共祖先。

为了解决脱机最小公共祖先问题，下面的过程通过对 LCA(*root*[T])初始调用，来执行对 T 的树遍历。在遍历之前，假定每个结点都着色为 WHITE。

```
LCA(u)
1  MAKE-SET(u)
2  ancestor[FIND-SET(u)] ← u
3  for each child v of u in T
4       do LCA(v)
5          UNION(u, v)
6          ancestor[FIND-SET(u)] ← u
7  color[u] ← BLACK
8  for each node v such that [u, v]∈P
9       do if color[v]=BLACK
10            then print "The least common ancestor of"
                        u "and" v "is" ancestor[FIND-SET(v)]
```

a)证明：对每一对$\{u, v\}\in P$，第 10 行恰执行一次。

b)证明：在调用 LCA(u)时，不相交集合数据结构中的集合数等于 u 在树 T 中的深度。

c)证明：对每一对$\{u, v\}\in P$，LCA 能正确地输出 u 和 v 的最小公共祖先。

d)假定采用 21.3 节中的不相交集合数据结构实现，分析 LCA 的运行时间。

本章注记

不相交集合数据结构方面的许多重要结果至少应部分地归功于 R. E. Tarjan。Tarjan[290, 292]利用聚集分析，给出了第一个紧确的上界，它是用 Ackermann 函数的增长极慢的逆函数 $\tilde{a}(m, n)$。来表达的(21.4 节中给出的函数 $A_k(j)$)类似于 Ackermann 函数，而函数 $\alpha(n)$则类似于其逆函数。对于所有能想像到的 m 和 n 值，$\alpha(n)$和 $\tilde{a}(m, n)$都至多为 4。)在更早一些的时候， 521
Hopcraft 和 Ullman[5, 155]证明了一个 $O(m\lg^* n)$上界。21.4 节中的处理是根据 Tarjan[294]后来所做的分析而改造的，而 Tarjan 的工作又是基于 Kozen[193]的分析而做出的。对于 Tarjan 早先给出的上界，Harfst 和 Reingold[139]给出了一个基于势的版本。

Tarjan 和 van Leeuwen[295]讨论了路径压缩启发式的各种变化，包括“一趟方法”，这种方

法与两趟的方法相比，有时在其性能表示中，可以给出更好的常数因子。与 Tarjan 早先对基本路径压缩启发式的分析一样，Tarjan 和 van Leeuewen 所给出的分析是聚集分析。Harfst 和 Reingold[139]后来证明了应如何对势函数做一个小小的改动，以便将其关于路径压缩的分析运用到这些一趟方法上来。Gabow 和 Tarjan[103]证明了在某些应用中，不相交集合操作可以做到在 $O(m)$时间内运行。

Tarjan[291]证明了对于任何满足特定技术条件的不相交集合数据结构上的操作，其运行时间的下界为 $\Omega(m\hat{a}(m, n))$。这一下界后来由 Fredman 和 Saks[97]加以了推广，他们证明了在最
522 坏情况下，必须访问 $\Omega(m\hat{a}(m, n))(\lg n)$位的内存字。

第六部分 图 算 法

引 言

图是计算机科学中常用的一类数据结构，有关图的算法也是计算机科学中基础性的算法。有许多有趣的计算问题都是用图来定义的。在本部分中，要介绍一些更为重要的图和图算法。

第 22 章介绍图在计算机中的表示，并讨论基于广度优先或深度优先图搜索的算法。在该章中，给出两种深度优先搜索的应用：根据拓扑结构对有向无回路图进行排序，以及将有向图分解为强连通子图。

第 23 章介绍如何求图的最小权生成树(minimum-weight spanning tree)问题。这种树是这样定义的，即当图中的每一条边都有一个相关的权值时，这种树由连接了图中所有顶点的、且权值最小的路径所构成。计算最小生成树的算法是贪心算法的很好的例子(见第 16 章)。

第 24 章和第 25 章考虑当图中的每条边都有一个相关的长度或“权重”时，如何计算顶点之间的最短路径问题。第 24 章讨论如何计算从一个给定的源顶点至所有其他顶点的最短路径问题。第 25 章考虑每一对顶点之间最短路径的计算问题。

最后，第 26 章介绍在物流网络(有向图)中，物流的最大流量计算问题。在这样的网络中，有着指定的物流源和指定的目的地，每条有向边也有着指定的容量(即每条有向边上可以通过的物流量)。这是个一般性的问题，会以多种形式出现；一个好的计算最大流量的算法可以用来有效地解决多种相关的问题。

523 ~ 525

在描述某一给定图 $G=(V, E)$ 上的一个图算法的运行时间时，通常以图中的顶点数 $|V|$ 和边数 $|E|$ 来度量输入的规模。亦即，用于描述输入规模的相关参数有两个，而不只是一个。本书中采用了一种通用的表示法来表示这两个参数。在渐近表示(如 O 记号或 Θ 记号)中，且仅在这些表示中，符号 V 代表 $|V|$，符号 E 代表 $|E|$。例如，我们可以这样说：“该算法的运行时间为 $O(VE)$”，意思就是该算法的运行时间为 $O(|V|\,|E|)$。这种约定使得表达运行时间的公式更容易阅读，而不会导致歧义。

我们采用的另一种约定出现在伪代码中。我们用 $V[G]$ 来表示一个图 G 的顶点集，用 $E[G]$ 来表示其边集。亦即，伪代码将顶点集和边集看作是图的属性。

526

第 22 章　图的基本算法

本章阐述图的表示方法和图的搜索方法。搜索一个图是有序地沿着图的边访问所有顶点。图的搜索算法可以使我们发现图的很多结构信息。许多图的算法在开始时，都是通过搜索输入的图来获取图结构信息。另外还有一些图的算法实际上是由基本的图搜索算法经过简单扩充而成的。因此，图的搜索技术是图算法领域的核心。

22.1 节讨论图在计算机中的两种最普遍的表示法：邻接表表示和邻接矩阵表示。22.2 节介绍一种简便的称为广度优先搜索的图搜索算法，并展示如何建立一个图的广度优先树。22.3 节介绍深度优先搜索算法，并证明一些根据深度优先搜索访问顶点次序的标准结论。22.4 节提供深度优先搜索的第一个应用：有向无回路图的拓扑排序。22.5 节给出深度优先搜索的另一个应用，即寻找一个有向图的强连通子图。

22.1　图的表示

要表示一个图 $G=(V, E)$，有两种标准的方法，即邻接表和邻接矩阵。这两种表示法既可以用于有向图，也可以用于无向图。通常采用邻接表表示法，因为用这种方法表示稀疏图(图中 $|E|$ 远小于 $|V|^2$)比较紧凑。本书中大部分图算法都假定输入的图的存储结构是邻接表形式。但是，当遇到稠密图($|E|$ 接近于 $|V|^2$)或必须很快判别两个给定顶点是否存在连接边时，通常采用邻接矩阵表示法。例如，第 25 章中讨论的两种每对顶点间最短路径算法中，假定输入图采用邻接矩阵表示法。

图 $G=(V, E)$ 的邻接表表示由一个包含 $|V|$ 个列表的数组 Adj 所组成。其中每个列表对应于 V 中的一个顶点。对于每一个 $u \in V$，邻接表 $Adj[u]$ 包含所有满足条件 $(u, v) \in E$ 的顶点 v。亦即，$Adj[u]$ 包含图 G 中所有和顶点 u 相邻的顶点。(或者，它也可能包含指向这些顶点的指针。)每个邻接表中的顶点一般以任意顺序存储。图 22-1b 是图 22-1a 中无向图的邻接表表示。类似地，图 22-2b 是图 22-2a 中有向图的邻接表表示。

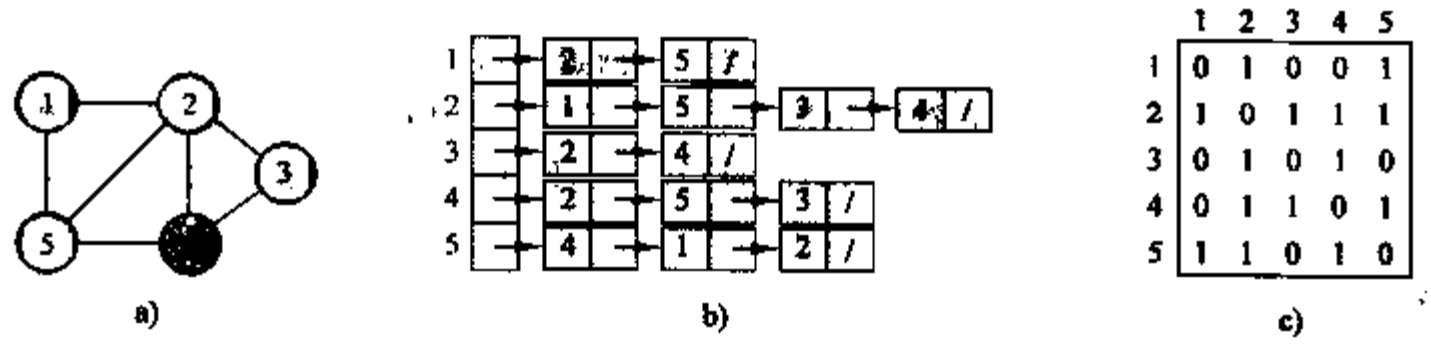


图 22-1　无向图的两种表示法。a)一个有 5 个顶点和 7 条边的无向图 G。b)G 的邻接表表示。c)G 的邻接矩阵表示

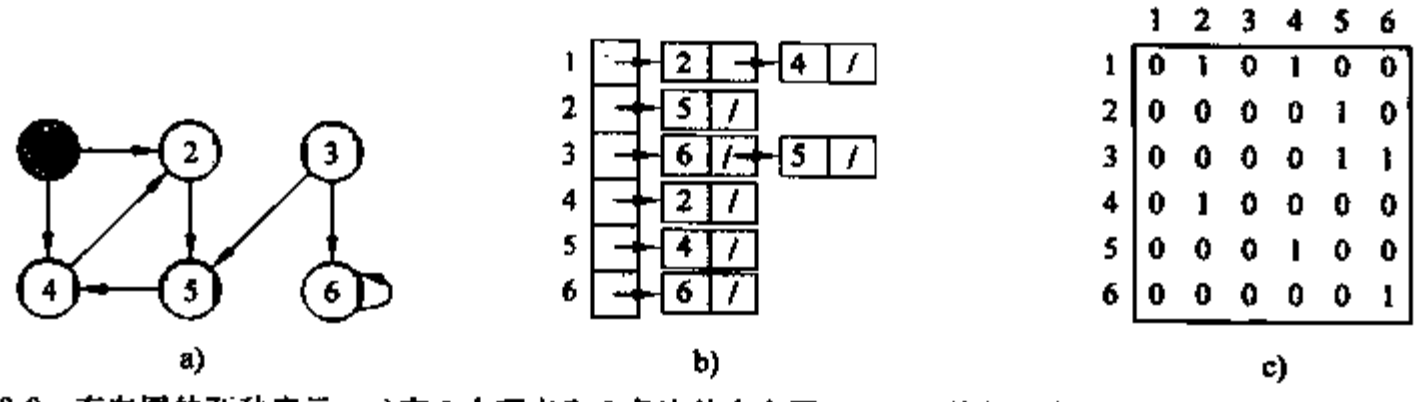


图 22-2　有向图的两种表示。a)有 6 个顶点和 8 条边的有向图 G。b)G 的邻接表表示。c)G 的邻接矩阵表示

如果 G 是一个有向图，则所有邻接表的长度之和为 $|E|$，这是因为一条形如 (u, v) 的边是通过让 v 出现在 $Adj[u]$ 中来表示的。如果 G 是一个无向图，则所有邻接表的长度之和为 $2|E|$，因为如果 (u, v) 是一条无向边，那么 u 就会出现在 v 的邻接表中，反之亦然。不论是有向图还是无向图，邻接表表示法都有一个很好的特性，即它所需要的存储空间为 $\Theta(V+E)$。 528

邻接表稍作变动，即可用来表示加权图，即每条边都有着相应权值的图。权值通常由加权函数 $w: E \to \mathbf{R}$ 给出。例如，设 $G=(V, E)$ 是一个加权函数为 w 的加权图。对每一条边 $(u, v) \in E$，权值 $w(u, v)$ 和顶点 v 一起存储在 u 的邻接表中。邻接表表示法稍作修改就能支持其他多种图的变体，因而有着很强的适应性。

邻接表表示法也有着潜在的不足之处，即如果要确定图中边 (u, v) 是否存在，只能在顶点 u 的邻接表 $Adj[u]$ 中搜索 v，除此之外，没有其他更快的方法。这一不足可以通过图的邻接矩阵表示法来弥补，但要（在渐近意义下）以占用更多的存储空间作为代价。（练习 22.1-8 中给出一些支持快速边查找功能的邻接表变形。）

在图 $G=(V, E)$ 的邻接矩阵表示法中，假定各顶点按某种任意的方式编号为 $1, 2, \cdots, |V|$，那么 G 的邻接矩阵为一个 $|V| \times |V|$ 的矩阵 $A=(a_{ij})$，它满足：

$$a_{ij} = \begin{cases} 1 & \text{如果}(i,j) \in E \\ 0 & \text{否则} \end{cases}$$

图 22-1c 和图 22-2c 分别是图 22-1a 和图 22-2a 中无向图和有向图的邻接矩阵表示。一个图的邻接矩阵表示需要占用 $\Theta(V^2)$ 的存储空间，它与图中的边数多少是无关的。

观察一下图 22-1c 中的邻接矩阵，会发现它是沿主对角线对称的。定义一个矩阵 $A=(a_{ij})$ 的转置矩阵为矩阵 $A^{\mathrm{T}}=(a_{ij}^{\mathrm{T}})$，其中 $a_{ij}^{\mathrm{T}}=a_{ji}$。因为在一个无向图中，$(u, v)$ 和 (v, u) 表示的是同一条边，故无向图的邻接矩阵 A 就是它自己的转置矩阵：$A=A^{\mathrm{T}}$。在某些应用中，可以只存储邻接矩阵的对角线及对角线以上的部分，这样一来，图所占用的存储空间几乎可以减少一半。

正如图的邻接表表示一样，邻接矩阵也可以用来表示加权图。例如，如果 $G=(V, E)$ 是一个加权图，其权值函数为 w，对于边 $(u, v) \in E$，其权值 $w(u, v)$ 就可以简单地存储在邻接矩阵的第 u 行第 v 列的元素中。如果边不存在，则可以在矩阵的相应元素中存储一个 NIL 值，在很多问题中，对这样的元素赋 0 或 ∞ 会更为方便一些。

邻接表表示和邻接矩阵表示在渐近意义下至少是一样有效的，但由于邻接矩阵简单明了，因而当图较小时，更多地采用邻接矩阵来表示。另外，如果一个图不是加权的，采用邻接矩阵的存储形式还有一个优越性：在存储邻接矩阵的每个元素时，可以只用一个二进位，而不必用一个字的空间。 529

练习

22.1-1 给定一个有向图的邻接表表示，计算该图中每个顶点的出度需要多少时间？计算每个顶点的入度需要多少时间？

22.1-2 给出一个包含 7 个顶点的完全二叉树的邻接表表示，写出其等价的邻接矩阵表示。假设各个顶点如在一个二项堆中一样，从 1 到 7 进行编号。

22.1-3 有向图 $G=(V, E)$ 的转置是图 $G^{\mathrm{T}}=(V, E^{\mathrm{T}})$，其中 $E^{\mathrm{T}}=\{(v, u) \in V \times V: (u, v) \in E\}$，因此 G^{T} 就是将 G 中所有的边反向后形成的图。写出根据 G 计算出 G^{T} 的有效算法（针对邻接表和邻接矩阵两种表示形式分别写出），并分析所给出算法的运行时间。

22.1-4 给定一个多重图 $G=(V, E)$ 的邻接表表示，给出一个具有 $O(V+E)$ 时间的算法，来计

算其"等价"的无向图 $G'=(V, E')$的邻接表表示，其中 E'包含 E 中所有的边，且将两个顶点之间的所有多重边用一条边代表，并去掉 E 中所有的环。

22.1-5 有向图 $G=(V, E)$的平方是图 $G^2=(V, E^2)$，该图满足下列条件：$(u, w)\in E^2$ 当且仅当对 $v\in V$，有 $(u, v)\in E$，且 $(v, w)\in E$。亦即，如果图 G 中顶点 u 和 w 之间存在着一条恰包含两条边的路径时，则 G^2 必包含该边 (u, w)。针对图 G 的邻接表和邻接矩阵两种表示法，分别写出根据 G 产生 G^2 的有效算法，并分析所给出算法的运行时间。

22.1-6 当采用邻接矩阵表示时，大多数图算法需要的时间都是 $\Omega(V^2)$，但也有一些例外。证明在给定了一个有向图 G 的邻接矩阵后，可以在 $O(V)$时间内，确定 G 中是否包含一个通用的汇(universal sink)，即入度为 $|V|-1$、出度为 0 的顶点。

530

22.1-7 有向图 $G=(V, E)$的关联矩阵 $B=(b_{ij})$是一个 $|V|\times|E|$ 的矩阵，它满足下列条件：

$$b_{ij}=\begin{cases}-1 & \text{如果边 } j \text{ 离开顶点 } i\\ 1 & \text{如果边 } j \text{ 进入顶点 } i\\ 0 & \text{其他情况}\end{cases}$$

试述矩阵乘积 BB^T 中各元素的含义，其中 B^T 为 B 的转置矩阵。

22.1-8 假设每个数组元素 $Adj[u]$采用的不是链表，而是一个包含了所有满足 $(u, v)\in E$ 的顶点 v 的散列表。如果所有的边查找都是等可能的，则确定某条边是否在图中所需的期望时间是多少？这种方案的不足是什么？请给出另一种表示每一个边列表的数据结构，以便解决这些不足。与散列表方案相比，你给出的方案有没有什么不足之处？

22.2　广度优先搜索

广度优先搜索(breadth-first search)是最简单的图搜索算法之一，也是很多重要的图算法的原型。在 Prim 最小生成树算法(23.2 节)和 Dijkstra 单源最短路径算法(24.3 节)中，都采用了与广度优先搜索类似的思想。

在给定图 $G=(V, E)$和一个特定的源顶点 s 的情况下，广度优先搜索系统地探索 G 中的边，以期"发现"可从 s 到达的所有顶点，并计算 s 到所有这些可达顶点之间的距离(即最少的边数)。该搜索算法同时还能生成一棵根为 s、且包括所有 s 的可达顶点的广度优先树。对从 s 可达的任意顶点 v，广度优先树中从 s 到 v 的路径对应于图 G 中从 s 到 v 的一条最短路径，即包含最少边数的路径。该算法对有向图和无向图同样适用。

该搜索算法之所以称为广度优先搜索，是因为它始终是将已发现和未发现顶点之间的边界，沿其广度方向向外扩展。亦即，算法首先会发现和 s 距离为 k 的所有顶点，然后才会发现和 s 距离为 $k+1$ 的其他顶点。

为了记录搜索的轨迹，广度优先搜索将每个顶点都着色为白色、灰色或黑色。算法开始时，所有顶点都是白色的；随着搜索的进行，各顶点会逐渐变成灰色，然后成为黑色。在搜索中第一

531 次碰到某一顶点时，称该顶点被发现了，此时该顶点变为非白色。因此，灰色和黑色顶点都是已被发现的，但广度优先搜索还是对它们加以了区分，以确保搜索是以一种广度优先的顺序进行的。如果 $(u, v)\in E$ 且顶点 u 为黑色，则顶点 v 就或者是灰色，或者是黑色；也就是说，与黑色顶点相邻的所有顶点都是已经被发现的。灰色顶点可能会有一些白色的相邻顶点，它们代表了已发现与未发现顶点之间的边界。

广度优先搜索构造了一棵广度优先树，它在开始时只包含一个根顶点，即源顶点 s。在扫描某个已发现顶点 u 的邻接表的过程中，每当发现一个白色顶点 v，该顶点 v 及边 (u, v)就被添加到树中。在广度优先树中，称 u 是 v 的先辈或父母。由于一个顶点至多只能被发现一次，因此，

它最多只能有一个父母顶点。在广度优先树中，祖先和后裔关系的定义和通常一样，是相对于根 s 来定义的：如果 u 处于树中从根 s 到顶点 v 的路径中，那么 u 称为 v 的祖先，v 是 u 的后裔。

下面的广度优先搜索过程 BFS 假定输入图 $G=(V, E)$采用邻接表表示，对于图中的每个顶点，还采用了几种另外的数据结构，对每个顶点 $u\in V$，其色彩存储于变量 $color[u]$中，u 的父母存于变量 $\pi[u]$中。如果 u 没有父母(例如 $u=s$ 或 u 尚未被发现)，则 $\pi[u]=$NIL。由该算法计算出来的源顶点 s 和顶点 u 之间的距离存于变量 $d[u]$中。该算法中还使用了一个先进先出队列 Q(见 10.1 节)来管理所有的灰色顶点。

```
BFS(G, s)
 1  for each vertex u∈V[G]−{s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q≠∅
11       do u ← DEQUEUE(Q)
12          for each v∈Adj[u]
13              do if color[v]=WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u]+1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18       color[u] ← BLACK
```

532

图 22-3 展示 BFS 作用于样例图上的搜索过程。

过程 BFS 的执行过程如下：第 1～4 行置每个顶点为白色，置每个顶点 u 的 $d[u]$为无穷大，置每个顶点的父母为 NIL。第 5 行将源顶点 s 为置为灰色，这是因为在过程开始时，源顶点已被发现。第 6 行将 $d[s]$初始化为 0，第 7 行将源顶点的父顶点置为 NIL。第 8～9 行初始化队列 Q，使其仅含源顶点 s。

只要队列 Q 中还有灰色顶点(即那些已被发现、但还没有完全搜索其邻接表的顶点)，第 10～18 行中的 **while** 循环将一直进行下去。这个 **while** 循环可以保持如下的不变式：

在第 10 行进行测试时，队列 Q 中包含了一组灰色顶点。

尽管下面不用这个循环不变式来证明算法的正确性，但也容易看出，它在循环的第一次迭代之前是成立的，并且循环的每一次迭代都能保持该不变式。在第一次迭代之前，唯一的一个灰色顶点(也是 Q 中的唯一顶点)是源顶点 s。第 11 行确定队列 Q 头部的灰色顶点 u，并将其从 Q 中去掉。第 12～17 行中的 **for** 循环考察 u 的邻接表中的每个顶点 v。如果 v 是白色的，则表明该顶点尚未被发现，此时算法通过执行第 14～17 行来发现该顶点。它先被置为灰色，其距离 $d[v]$被置为 $d[u]+1$。接着，u 被记为该顶点的父母。最后，它被置于队列 Q 的尾部。当 u 的邻接表中的所有顶点都被检查完后，第 11～18 行将 u 置为黑色。此时，循环不变式得到了保持，因为每当一个顶点(在第 14 行中)被置为灰色时，它还同时被插入到队列中(第 17 行)，而每当一个顶点出队(第 11 行)后，它还同时被置为黑色(第 18 行)。

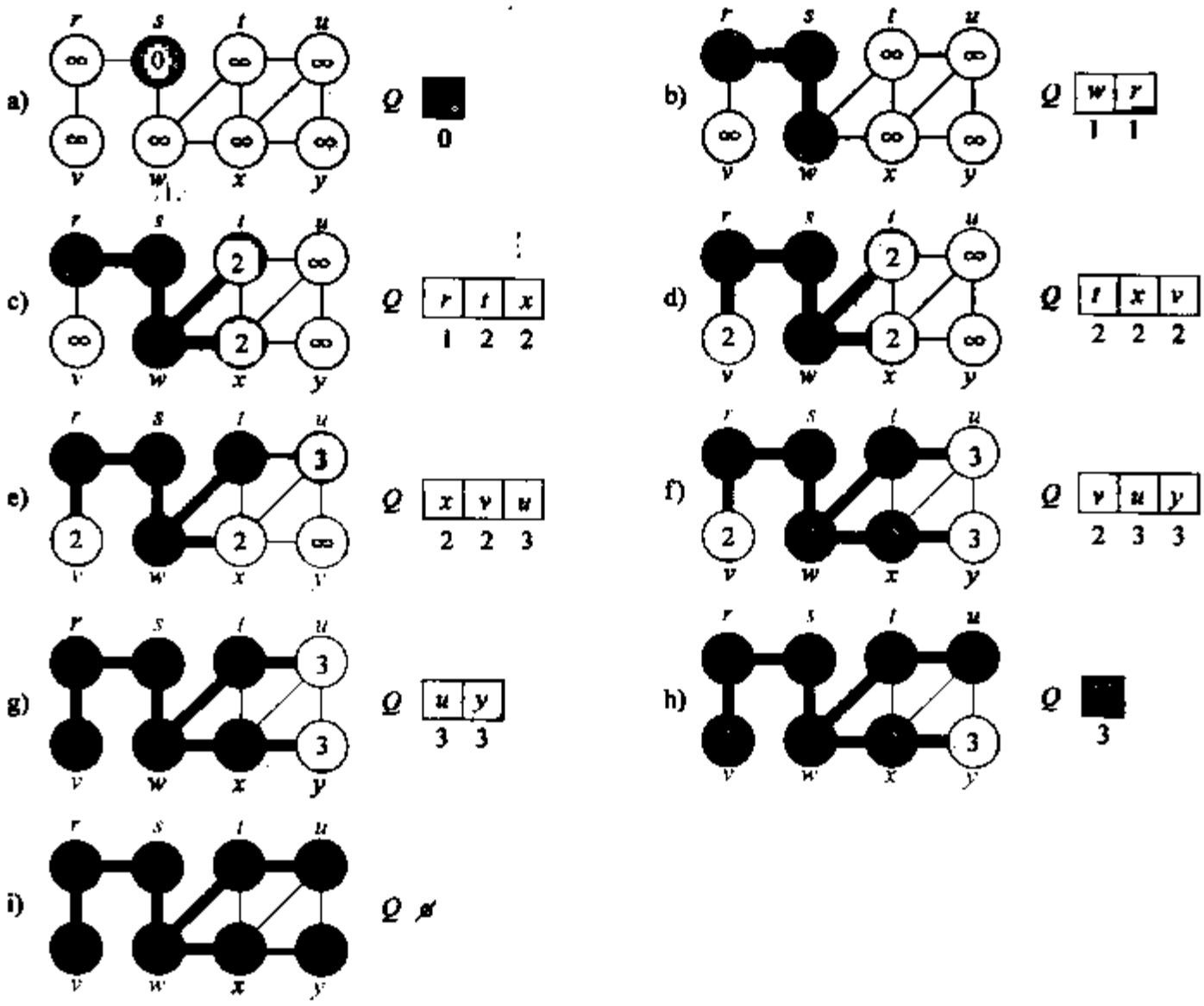


图 22-3　BFS 在一个无向图上的处理过程。树中的边在由 BFS 产生时即用阴影显示。在每个顶点 u 内示出了 $d[u]$。在过程的第 10～18 行中，在 while 循环的每一轮迭代的开始时显示队列 Q。顶点间距离在队列中顶点的旁边示出

广度优先搜索的结果与第 12 行中，某一给定顶点的邻居被访问的顺序有关：产生的广度优先树可能会有所不同，而由算法计算出来的距离 d 则都是一样的。(见练习 22.2-4。)

分析

在证明广度优先搜索的各种性质之前，先来做一项相对简单一些的工作，即分析该算法在输入图 $G=(V, E)$ 上的运行时间。此处采用聚集分析(aggregate analysis)技术，如 17.1 节中一样。在初始化后，再没有任何顶点被置为白色。因此，第 13 行中的测试保证了每个顶点至多只进入队列一次，因而至多只从队列中出来一次。入队和出队操作需要 $O(1)$ 的时间，因此队列操作所需的全部时间为 $O(V)$。因为只有当每个顶点将出队时，才会扫描其邻接表，因而每个顶点的邻接表至多被扫描一次。由于所有邻接表的长度和为 $\Theta(E)$，故扫描所有邻接表所花费的全部时间为 $O(E)$。初始化操作的开销为 $O(V)$，于是，过程 BFS 的总运行时间为 $O(V+E)$。由此可见，广度优先搜索的运行时间是图 G 的邻接表大小的一个线性函数。

最短路径

533
~
534

在本节的开始我们讲过，对于一个图 $G=(V, E)$，广度优先搜索算法可以得到从已知源顶点 $s \in V$ 到每个可达顶点的距离。定义从顶点 s 到 v 之间的最短路径距离 $\delta(s, v)$ 为从 s 到 v 的任何路径中最少的边数；如果从 s 到 v 之间没有通路，则 $\delta(s, v)=\infty$。具有这一距离 $\delta(s, v)$ 的路

径即为从 s 到 v 的最短路径[⊖]，在证明广度优先搜索计算出来的就是最短路径距离之前，先来看一下最短路径距离的一个重要性质。

引理 22.1　设 $G=(V, E)$是一个有向图或无向图，$s\in V$ 为 G 的任意一个顶点，则对任意边 $(u, v)\in V$，有：

$$\delta(s,v)\leqslant\delta(s,u)+1$$

证明：如果从顶点 s 可达顶点 u，则从 s 也可达 v。在这种情况下，从 s 到 v 的最短路径不可能比从 s 到 u 的最短路径加上边 (u, v)更长，因此不等式成立。如果从 s 不可达顶点 u，则 $\delta(s, u)=\infty$，不等式仍然成立。■

我们希望证明对每个顶点 $v\in V$，BFS 过程都能正确地计算出 $d[v]=\delta(s, v)$。下面，先来证明 $d[v]$是 $\delta(s, v)$的上界。

引理 22.2　设 $G=(V, E)$是一个有向或无向图，并假设算法 BFS 从 G 中某一给定源顶点 $s\in V$ 开始执行。在执行终止时，对每个顶点 $v\in V$，BFS 所计算出来的 $d[v]$的值满足 $d[v]\geqslant\delta(s, v)$。

证明：对一个顶点进入队列 Q(ENQUEUE)的操作次数进行归纳。归纳假设为对所有顶点 $v\in V$，$d[v]\geqslant\delta(s, v)$成立。

归纳的基础是 BFS 过程第 9 行中当顶点 s 被插入队列 Q 后的情形。这时归纳假设成立，因为对于任意顶点 $v\in V-\{s\}$，有 $d[s]=0=\delta(s, s)$，且 $d[v]=\infty\geqslant\delta(s, v)$。

在归纳步骤中，考虑从顶点 u 开始进行的搜索过程中发现的一个白色顶点 v。归纳假设蕴含了 $d[u]\geqslant\delta(s, u)$。根据 BFS 过程第 15 行中的赋值语句以及引理 22.1 可知：

$$d[v]=d[u]+1\geqslant\delta(s,u)+1\geqslant\delta(s,v)$$

535

然后，顶点 v 被插入到队列 Q 中，以后它不会再次被插入队列，因为它已被置为灰色了，而第 14～17 行的 then 子句只对白色顶点进行操作。这样，$d[v]$的值就不会再有所改变了，所以归纳假设成立。■

为了证明 $d[v]=\delta(s, v)$，首先必须更准确地展示在 BFS 的执行过程中，是如何对队列进行操作的。下面一个引理说明无论何时，队列中的顶点至多有两个不同的 d 值。

引理 22.3　假设过程 BFS 在图 $G=(V, E)$上的执行过程中，队列 Q 包含顶点$(v_1, v_2, \cdots, v_r)$，其中 v_1 是队列 Q 的头，v_r 是队列的尾，则 $d[v_r]\leqslant d[v_1]+1$，且 $d[v_i]\leqslant d[v_{i+1}]$，$i=1, 2, \cdots, r-1$。

证明：证明过程是对队列操作的次数进行归纳。开始时，队列中仅包含顶点 s，引理自然成立。

在归纳步骤中，必须证明在队列中插入和去掉一个顶点后，引理仍然成立。如果队列头 v_1 被从队列中去掉了，v_2 即成为新的队头。(如果此时队列为空，引理无疑是成立的)。根据归纳假设，应该有 $d[v_1]\leqslant d[v_2]$，但接着就有 $d[v_r]\leqslant d[v_1]+1\leqslant d[v_2]+1$，余下的不等式不受影响。因此，$v_2$ 为队头时引理依然成立。

对于将顶点插入队列中的情况，需要更仔细地分析一下过程 BFS。在 BFS 的第 17 行中，当将顶点 v 插入队列时，它即成为 v_{r+1}。此时，顶点 u 已从队列 Q 中去掉，其邻接表则正在被扫描。根据归纳假设，对新的队头 v_1，有 $d[v_1]\geqslant d[u]$。于是，$d[v_{r+1}]=d[v]=d[u]+1\leqslant d[v_1]+1$。根据归纳假设，还可以得出 $d[v_r]\leqslant d[u]+1$，因而有 $d[v_r]\leqslant d[u]+1=d[v]=d[v_{r+1}]$，余下的不等

⊖ 在第 24 章和第 25 章中，我们将把对最短路径的研究推广至加权图，即图中的每一条边都有一个实型的权值。一条路径的权值是组成该路径的所有边的权值之和。本章中所讨论的图都不是加权图，或者说，图中所有边都有单位权值。

式不受影响。因此，当顶点 v 被插入到队列中时，引理同样正确。■

下面的推论表明，在顶点被插入时的 d 值是随时间而单调递增的。

推论 22.4　假设在 BFS 的执行过程中将顶点 v_i 和 v_j 插入了队列，且 v_i 先于 v_j 入队。那么，当 v_j 入队时，有 $d[v_i] \leqslant d[v_j]$。

证明：在 BFS 的执行过程中，每个顶点至多接收一个有限的 d 值一次；根据这一性质及引理 22.3，立即可得本推论。■

536　现在就可以来证明广度优先搜索算法能够正确地计算出最短路径距离了。

定理 22.5(广度优先搜索的正确性)　设 $G=(V, E)$ 是一个有向图或无向图，并假设过程 BFS 从 G 上某个给定的源顶点 $s \in V$ 开始运行。那么，在执行过程中，BFS 可以发现源顶点 s 可达的每一个顶点 $v \in V$。在运行终止时，对所有 $v \in V$，都有 $d[v]=\delta(s, v)$。此外，对任意从 s 可达的顶点 $v \neq s$，从 s 到 v 的最短路径之一是从 s 到 $\pi[v]$ 的最短路径再加上边 $(\pi[v], v)$。

证明：为了引出矛盾，假设某个顶点接收了一个不等于其最短路径距离的 d 值。设 v 为一个接收了这种不正确 d 值的、具有最小 $\delta(s, v)$ 的顶点；显然，$v \neq s$。根据引理 22.2，有 $d[v] \geqslant \delta(s, v)$，从而有 $d[v] > \delta(s, v)$。顶点 v 必定是从 s 可达的，否则的话，就会有 $\delta(s, v)=\infty \geqslant d[v]$。设 u 为从 s 到 v 的一条最短路径中 v 的直接前趋顶点，于是有 $\delta(s, v)=\delta(s, u)+1$。因为 $\delta(s, u)<\delta(s, v)$，并且，根据我们选取 v 的方式，有 $d[u]=\delta(s, u)$。将这些性质综合在一起，有：

$$d[v] > \delta(s,v) = \delta(s,u) + 1 = d[u] + 1 \tag{22.1}$$

现在来考虑在第 11 行中，BFS 选择使顶点 u 从 Q 中出队的时间。在这一时刻，顶点 v 可能是白色的、灰色的或黑色的。我们将证明，在这几种情况的每一种中，都能导出与不等式(22.1)的矛盾来。如果 v 是白色的，则第 15 行就会置 $d[v]=d[u]+1$，这与不等式(22.1)矛盾。如果 v 是黑色的，则它已经被从队列中去掉了，于是，根据推论 22.4，有 $d[v] \leqslant d[u]$，这又与不等式(22.1)矛盾。如果 v 是灰色的，就会在某个顶点 w 出队时被置为灰色；w 的出队时间要早于 u，且满足 $d[v]=d[w]+1$。然而，根据推论 22.4 有 $d[w] \leqslant d[u]$，因而可以导出 $d[v] \leqslant d[u]+1$，这再一次与不等式(22.1)矛盾。

于是可以得出这样的一个结论，即对于所有的 $v \in V$，都有 $d[v]=\delta(s, v)$。所有从 s 可达的顶点都必定会被发现，因为如果它们没有被发现的话，就会有无穷大的 d 值。在证明的最后，注意到如果 $\pi[v]=u$，则 $d[v]=d[u]+1$。于是，通过取一条从 s 至 $\pi[v]$ 的最短路径，接着再访问边 $(\pi[v], v)$ 的做法，就可以得到一条从 s 至 v 的最短路径了。■

广度优先树

过程 BFS 在搜索图的同时，也建立了一棵广度优先树，如图 22.3 中所示。这棵树是由每个顶点中的 π 域所表示的。对于图 $G=(V, E)$ 及给定的源顶点 s，可以更为形式化地定义其前趋子图(predecessor subgraph) $G_\pi=(V_\pi, E_\pi)$，其中：

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

537　且

$$E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$$

如果 V_π 由从 s 可达的顶点所构成，则前趋子图 G_π 是一棵广度优先树，并且对于所有的 $v \in V_\pi$，在 G_π 中都有唯一的从 s 到 v 的简单路径，该路径也同样是 G 中从 s 到 v 的一条最短路径。广度优先树事实上就是一棵树，因为它是连通的，且 $|E_\pi| = |V_\pi| - 1$(见定理 B.2)。E_π 中的边称为树的边。

下面的引理表明，当 BFS 从图 G 中的某个源顶点 s 开始执行后，前趋子图即构成一棵广度优先树。

引理 22.6　当过程 BFS 应用于某一有向图或无向图 $G=(V, E)$时，同时要构造出 π 域，使得前趋子图 $G_\pi=(V_\pi, E_\pi)$是一棵广度优先树。

证明：过程 BFS 的第 16 行置 $\pi[v]=u$，当且仅当$(u, v)\in E$ 且 $\delta(s, v)<\infty$(亦即，v 是从 s 可达的)，因而，V_π 是由 V 中从 s 可达的顶点所组成。由于 G_π 形成了一棵树，所以，根据定理 B.2，它包含从 s 到 V_π 中每一个顶点的唯一路径。通过归纳地应用定理 22.5，可知每一条这样的路径都是最短路径。(证毕) ■

下面的过程将输出从 s 到 v 的最短路径上的所有顶点，假定已经运行了 BFS 来计算最短路径树。

```
PRINT-PATH(G, s, v)
1  if v=s
2     then print s
3     else if π[v]=NIL
4             then print "no path from" s "to" v "exists"
5             else PRINT-PATH(G, s, π[v])
6                  print v
```

因为每次递归调用时的路径都比前一次调用中的路径少一个顶点，所以该过程的运行时间是关于所输出路径上顶点数的一个线性函数。

练习

22.2-1　对于图 22-2a 中所示的有向图，当指定源顶点为 3 时运行广度优先搜索算法，说明所得的 d 和 π 值是什么。

22.2-2　对于图 22-3 中所示的无向图，当指定源顶点为 u 时运行广度优先搜索算法，说明所得的 d 和 π 值是什么。 538

22.2-3　如果 BFS 的输入图是用邻接矩阵表示的，且对该算法加以修改以处理这种输入图表示，那么该算法的运行时间如何？

22.2-4　试证明在广度优先搜索算法中，赋给顶点 u 的值 $d[u]$与顶点在邻接表中的次序无关。利用图 22-3 作为例子，说明由 BFS 计算出来的广度优先树与邻接表中的顺序是有关的。

22.2-5　举例说明，在有向图 $G=(V, E)$中，源顶点 $s\in V$，且树边集合 $E_\pi\subseteq E$ 满足对每一顶点 $v\in V$，图(V, E_π)中从 s 到 v 的唯一路径是 G 中的一条最短路径；然而，不论在每个邻接表中各顶点如何排列，都不能通过在 G 上运行 BFS 而产生边集 E_π。

22.2-6　有两种类型的职业摔跤手：一种是"好选手"，另一种是"差选手"。对于任何一对职业摔跤手来说，他们中可能有、也可能没有比赛。假定有 n 位职业摔跤手，并且有一份清单，上面列出了 r 对参加比赛的摔跤手。试给出一个 $O(n+r)$时间的算法，它能够确定是否可能指定某些摔跤手为好选手，而将余下的摔跤手指定为坏选手，从而使得每一场比赛都是在一个好选手与一个差选手之间进行。如果有可能做出这样的指定，你的算法就应该将它产生出来。

***22.2-7**　树 $T=(V, E)$的直径(diameter)定义为$\max_{u,v\in V}\delta(u, v)$，亦即，树的直径是树中所有最短路径长度中的最大值。试写出计算树的直径的有效算法，并分析算法的运行时间。

22.2-8　设 $G=(V, E)$是一个连通的无向图。请给出一个 $O(V+E)$时间的算法，以计算图 G 中

的一条路径，对于 E 中的每一条边，该路径都恰好在每一个方向上遍历一次。如果你身处一个迷宫之中，说明如何才能找到出路。

539

22.3　深度优先搜索

正如"深度优先搜索"这一名称所暗示的那样，这种搜索算法所遵循的搜索策略是尽可能"深"地搜索一个图。在深度优先搜索中，对于最新发现的顶点，如果它还有以此为起点而未探测到的边，就沿此边继续探测下去。当顶点 v 的所有边都已被探寻过后，搜索将回溯到发现顶点 v 有起始点的那些边。这一过程一直进行到已发现从源顶点可达的所有顶点时为止。如果还存在未被发现的顶点，则选择其中一个作为源顶点，并重复以上过程。整个过程反复进行，直到所有的顶点都已被发现时为止。

与广度优先搜索类似，在深度优先搜索中，每当扫描已发现顶点 u 的邻接表，从而发现新顶点 v 时，就将置 v 的先辈域 $\pi[v]$ 为 u。与广度优先搜索不同的是，其先辈子图形成一棵树，深度优先搜索产生的先辈子图可以由几棵树所组成，因为搜索可能由多个源顶点开始重复进行。[⊖] 因此，在深度优先搜索中，先辈子图的定义也和广度优先搜索中稍有所不同：$G_\pi=(V, E_\pi)$，其中

$$E_\pi = \{(\pi[v], v) : v \in V \quad 且 \quad \pi[v] \neq \text{NIL}\}$$

深度优先搜索的先辈子图形成了一个由数棵深度优先树所组成的深度优先森林。E_π 中的边称为树边。

与广度优先搜索类似，在深度优先搜索过程中，也通过对顶点进行着色来表示顶点的状态。开始时，每个顶点均为白色，搜索中被发现时即置为灰色，结束时又被置成黑色(即当其邻接表被完全检索之后)。这一技巧可以保证每一个顶点在搜索结束时，只存在于一棵深度优先树中，因此，这些树是不相交的。

除了创建一个深度优先森林外，深度优先搜索同时为每个顶点加盖时间戳。每个顶点 v 有两个时间戳：当顶点 v 第一次被发现(并置成灰色)时，记录下第一个时间戳 $d[v]$，当结束检查 v 的邻接表(并置 v 为黑色)时，记录下第二个时间戳 $f[v]$。许多图的算法中都用到了时间戳，它们对推算深度优先搜索的进行情况有很大的帮助。

540

下面给出的过程 DFS 记录了何时在变量 $d[u]$ 中发现顶点 u，以及何时在变量 $f[u]$ 中完成对顶点 u 的检索。这些时间戳为 1 到 $2|V|$ 之间的整数，因为对 $|V|$ 个顶点中的每一个，都对应一个发现事件和一个完成事件。对每一个顶点 u，有：

$$d[u] < f[u] \tag{22.2}$$

顶点 u 在时刻 $d[u]$ 之前为白色(WHITE)，在时刻 $d[u]$ 和 $f[u]$ 之间为灰色(GRAY)，以后就变为黑色(BLACK)了。

下面的伪代码就是一个基本的深度优先搜索算法，输入图 G 可以是有向图或无向图，变量 *time* 是一个全局变量，用于记录时间戳。

```
DFS(G)
1  for each vertex u∈V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
```

⊖ 广度优先搜索只能有一个源顶点，而深度优先却可以从多个源顶点开始搜索，这一点看上去似乎有些随意。尽管从概念上说，广度优先搜索也可以从多个源顶点开始进行，而深度优先也可以仅从一个源顶点开始，但我们的做法反映了这两种搜索的结果一般情况是如何使用的。广度搜索通常用于从某个源顶点开始，寻找最短路径距离(以及相关的先辈子图)。深度优先搜索通常作为另一个算法中的一个子程序，这一用法在本章稍后就能看到。

```
4  time ← 0
5  for each vertex u∈V[G]
6       do if color[u]=WHITE
7            then DFS-VISIT(u)

DFS-VISIT(u)
1  color[u] ← GRAY          ▷White vertex u has just been discovered.
2  time ← time+1
3  d[u] ← time
4  for each v∈Adj[u]        ▷Explore edge(u, v).
5       do if color[v]=WHITE
6            then π[v] ← u
7                 DFS-VISIT(v)
8  color[u] ← BLACK         ▷Blacken u; it is finished.
9  f[u] ← time ← time+1
```

图 22-4 说明了 DFS 在图 22-2 所示图上的执行过程。

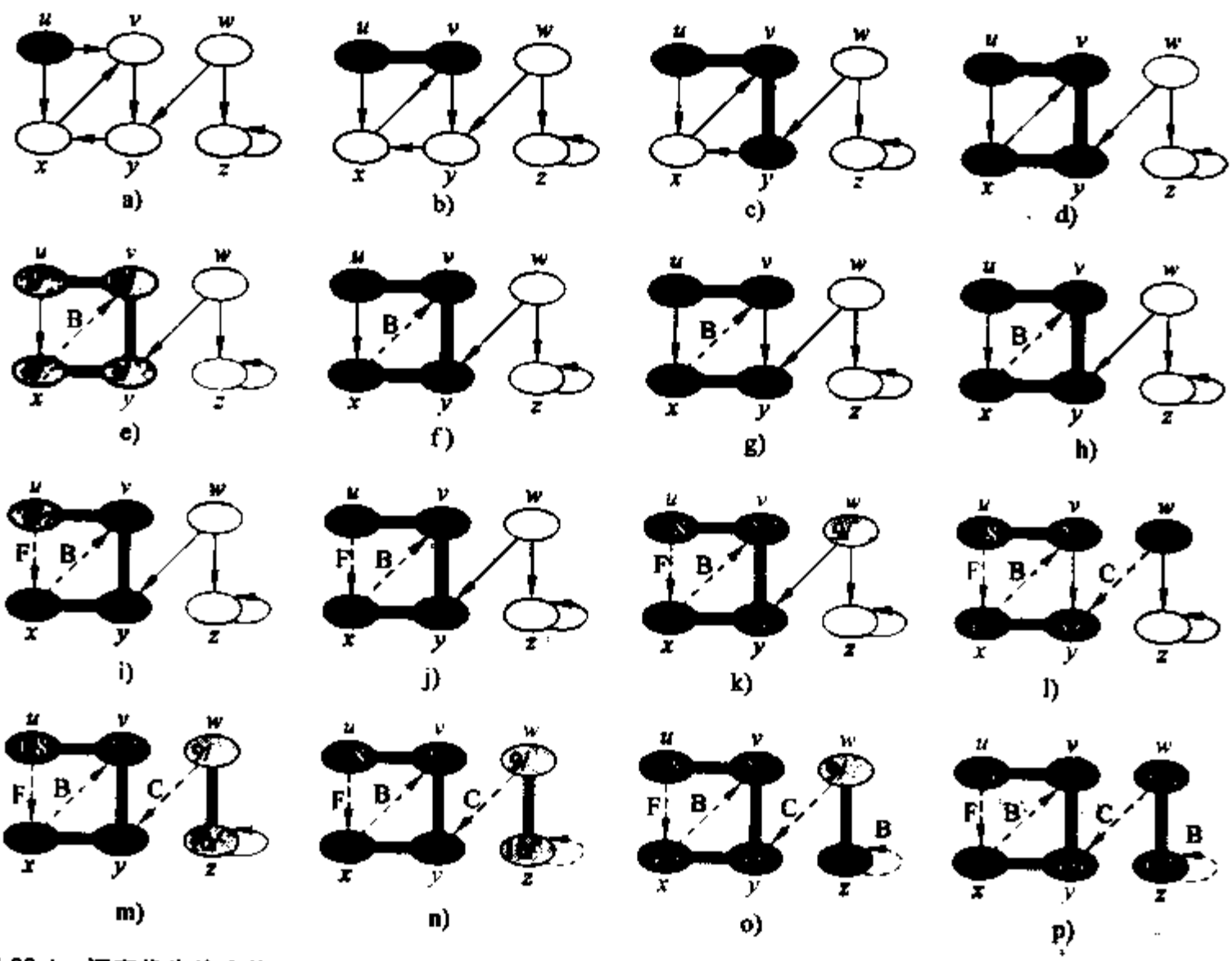


图 22-4　深度优先搜索算法 DFS 在有向图上的工作过程。当各条边被算法探索到时，它们或者被加以阴影（如果它们是树边），或者被标以虚线（否则的话）。对于非树边，根据它们是反向、交叉或前向边等情况，将它们标记为 B、C 或 F。对于各个顶点，根据它们的发现/完成时间，给它们加上时间戳

过程 DFS 的执行过程如下：第 1～3 行把所有顶点置为白色，所有 π 域初始化为 NIL。第 4 行复位全局时间计数器，第 5～7 行依次检索 V 中的顶点，发现白色顶点时，调用 DFS-VISIT 访问该顶点。每次通过第 7 行调用 DFS-VISIT(u)时，顶点 u 就成为深度优先森林中一棵新树的根，

当 DFS 返回时，每个顶点 u 都对应于一个发现时刻 $d[u]$和一个完成时刻 $f[u]$。

每次调用 DFS-VISIT(u)时，顶点 u 开始时为白色，第 1 行置 u 为灰色，第 2 行使全局变量 *time* 增值，第 3 行将 *time* 的新值记录为发现时间 $d[u]$。第 4～7 行检查和 u 相邻接的每个顶点 v，并且如果 v 为白色顶点，则递归访问顶点 v。在第 4 行语句中，在检查每一个顶点 $v \in Adj[u]$ 时，就称边(u，v)已被深度优先搜索探寻。最后，当以 u 为起点的所有边都被探寻后，第 8～9 行语句置 u 为黑色，并将完成时间记录在 $f[u]$中。

注意深度优先搜索的结果可能依赖于在 DFS 的第 5 行中各顶点被访问的顺序，也依赖于在 DFS-VISIT 的第 4 行中一个顶点的邻居顶点被访问的顺序。在实践中，这些不同的访问顺序往往不会引起什么问题，因为任何深度优先搜索结果通常都可以被有效地利用，最终结果基本上都是等价的。

541
|
542

算法 DFS 的运行时间怎样呢？DFS 中第 1～3 行和第 5～7 行中的循环占用的时间为 $\Theta(V)$，这不包括调用 DFS-VISIT 的时间。就像我们在处理广度优先搜索时一样，采用聚集分析。对于每个顶点 $v \in V$，过程 DFS-VISIT 仅被调用一次，因为只有对白色顶点才会调用该过程，且该过程所做的第一件事就是将顶点置为灰色。在 DFS-VISIT(v)的一次执行过程中，第 4～7 行中的循环被执行了 $|Adj[v]|$ 次。因为有

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

故执行过程 DFS-VISIT 中第 4～7 行的总代价为 $\Theta(E)$。因此，DFS 的运行时间为 $\Theta(V+E)$。

深度优先搜索的性质

利用深度优先搜索，可以获得有关图结构的有价值的信息。深度优先搜索最基本的特征也许是它的先辈子图 G_π 形成了一个由树所组成的森林，这是因为深度优先树的结构准确反映了递归调用 DFS-VISIT 的过程。亦即，$u=\pi[v]$当且仅当在搜索 u 的邻接表的过程中，调用了过程 DFS-VISIT(v)。此外，在深度优先森林中，顶点 v 是顶点 u 的后裔，当且仅当 v 是 u 为灰色时发现的。

深度优先搜索的另一个重要特性是发现和完成时间具有括号结构(parenthesis structure)。如果把发现顶点 u 用左括号“(u”表示，完成用右括号“u)”表示，那么，在各级括号正确嵌套的前提下，发现与完成时间的记载就是一个完善的表达式。例如，图 22-5a 中的深度优先搜索就对应着图 22-5b 中的括号结构。下面的定理给出了另一种表述括号结构条件的方式。

定理 22.7(括号定理)　在对一个(有向或无向)图 $G=(V, E)$的任何深度优先搜索中，对于图中任意两个顶点 u 和 v，下述三个条件中仅有一个成立：

- 区间$[d[u], f[u]]$和区间$[d[v], f[v]]$是完全不相交的，且在深度优先森林中，u 或 v 都不是对方的后裔。
- 区间$[d[u], f[u]]$完全包含于区间$[d[v], f[v]]$中，且在深度优先树中，u 是 v 的后裔。

543

- 区间$[d[v], f[v]]$完全包含于区间$[d[u], f[u]]$中，且在深度优先树中，v 是 u 的后裔。

证明：先讨论 $d[u]<d[v]$的情形。根据 $d[v]$是否小于 $f[u]$，又可以进一步分为两种子情况，第一种子情况是若 $d[v]<f[u]$，这样 v 已被发现时，u 顶点依然是灰色的，这就说明 v 是 u 的后裔。再者，因为顶点 v 比 u 发现得晚，所以在搜索返回顶点 u 并完成之前，所有从 v 出发的边都已被探寻并已完成。因此，在这种条件下，区间$[d[v], f[v]]$必然完全包含于区间$[d[u], f[u]]$中。第二种子情况是 $f[u]<d[v]$，根据不等式(22.2)，区间$[d[u], f[u]]$和区间$[d[v], f[v]]$必然是不相交的。因为这两个区间是不相交的，对于这两个顶点中的任意一个而言，当另一个为灰色时，当前顶点一定是未被发现的，因而它们都不是对方顶点的后裔。

$d[v]<d[u]$ 的情形是类似的，只要把上述证明中 u 和 v 对调一下即可。(证毕)　■

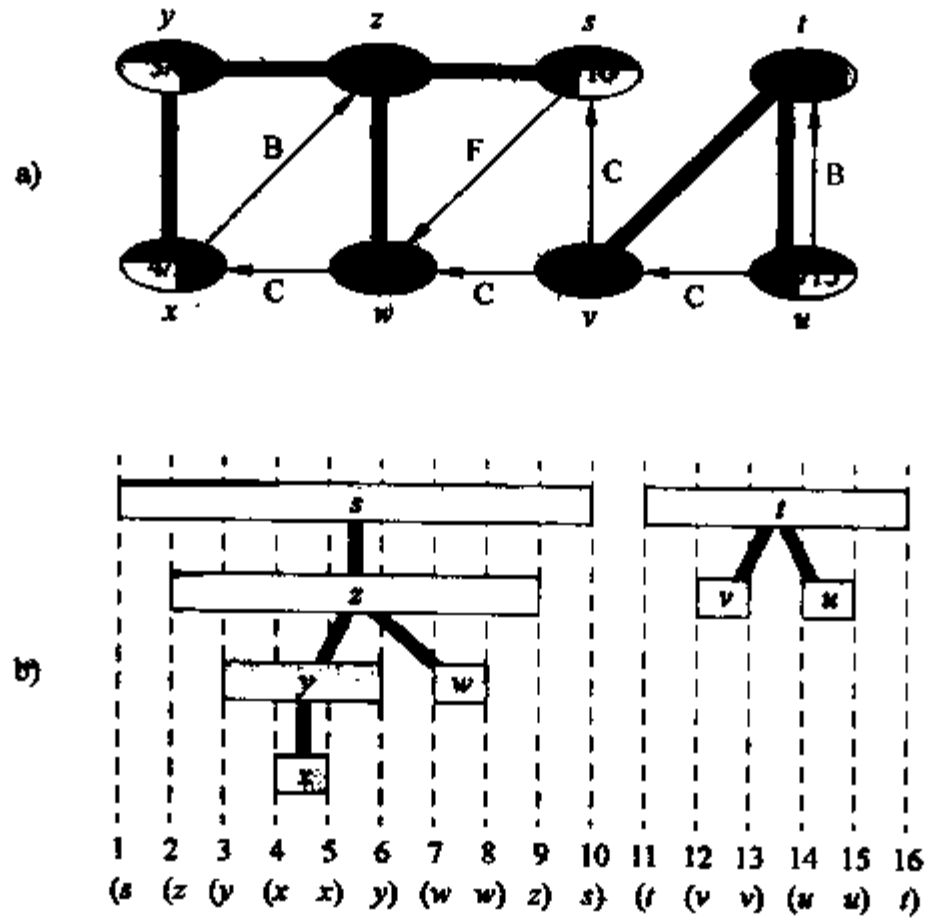


图 22-5　深度优先搜索的性质。a)对一个有向图进行深度优先搜索的结果。各顶点都如图 22-4 中所示加盖了时间戳，各边的类型也已标出。b)每个顶点的发现时间和完成时间区间与图中示出的括号化顺序对应。每个矩形跨越由对应顶点的发现时间和完成时间所构成的区间。树边也在图中示出。如果两个区间重叠，则一个嵌套在另一个之内，并且，与较小区间对应的顶点是与较大区间对应的顶点的后裔。c)重新绘制的子图 a)，在深度优先树中，所有的树和前向边都指向下方，且所有的反向边都从一个后裔回指向某一祖先

推论 22.8(后裔区间的嵌套)　在一个(有向或无向)图 G 中的深度优先森林中，顶点 v 是顶点 u 的后裔，当且仅当 $d[u]<d[v]<f[v]<f[u]$。

证明：根据定理 22.7 立即可得。(证毕) ■

在深度优先森林中，如果一个顶点是另一个顶点的后裔，下面的定理给出了另一个重要的特征。

定理 22.9(白色路径定理)　在一个(有向或无向)图 $G=(V, E)$ 的深度优先森林中，顶点 v 是顶点 u 的后裔，当且仅当在搜索过程中于时刻 $d[u]$ 发现 u 时，可以从顶点 u 出发，经过一条完全由白色顶点组成的路径到达 v。

证明：

⇒：假设 v 是 u 的后裔。设 w 是深度优先树中，u 和 v 之间的通路上的任意顶点，这样 w 就是 u 的后裔。根据推论 22.8 可知 $d[u]<d[w]$，因此在时刻 $d[u]$，w 是白色的。

⇐：假设在时刻 $d[u]$，顶点 v 沿着一条仅由白色顶点所组成的通路可达 u，但在深度优先树中，v 没有成为 u 的后裔。不失一般性，假定该通路上的其他每一个顶点都成为了 u 的后裔(否则可以设 v 是该通路中最接近 u 的、且不为 u 的后裔的顶点)。设 w 为该通路上 v 的前趋，这样 w 就是 u 的后裔(实际上，w 和 u 可以是同一个顶点)，根据推论 22.8 得 $f[w]\geqslant f[u]$。注意，v 必须在 u 被发现之后、在 w 完成之前被发现，于是，有 $d[u]<d[v]<f[w]\leqslant f[u]$。由定理 22.7可知，区间$[d[v], f[v]]$完全包含于区间$[d[u], f[u]]$中。根据推论 22.8，$v$ 必定是 u 的一个后裔。(证毕) ■

544
~
545

边的分类

深度优先搜索另一个令人感兴趣的性质就是可以通过搜索对输入图 $G=(V, E)$的边进行归类，这种归类可以用来收集有关图的很多重要信息。例如，在一下节中可以看到，一个有向图是无回路的，当且仅当对该图的深度优先搜索没有产生"反向"边(引理 22.11)。

根据在图 G 上进行深度优先搜索所产生的深度优先森林 G_π，可以把图的边分为四种类型：

1)树边(tree edge)，是深度优先森林 G_π 中的边。如果顶点 v 是在探寻边(u, v)时被首次发现的，那么(u, v)就是一条树边。

2)反向边(back edge)是深度优先树中，连接顶点 u 到它的某一祖先顶点 v 的那些边。有向图中可能出现的自环也被认为是反向边。

3)正向边(forward edge)是指深度优先树中，连接顶点 u 到它的某个后裔 v 的非树边(u, v)。

4)交叉边(cross edge)是其他类型的边，存在于同一棵深度优先树中的两个顶点之间，条件是其中一个顶点不是另一个顶点的祖先。交叉边也可以在不同的深度优先树的顶点之间。

在图 22-4 和图 22-5 中，都对边进行了类型标识。图 22-5c 还说明了如何重新绘制图 22-5a，以使深度优先树中的树边和正向边向下绘制，使反向边向上绘制，任何图都可以按这种方式重新绘制。

可以对算法 DFS 做一些修改，使之遇到图中的边时，对其进行分类。算法的核心思想在于对于每条边(u, v)，当该边被第一次探寻到时，即根据所到达的顶点 v 的颜色，来对该边进行分类(只是对正向边和交叉边不作区分)：

1)白色(WHITE)表明它是一条树边。

2)灰色(GRAY)表明它是一条反向边。

3)黑色(BLACK)表明它是一条正向边或交叉边。

第一种情形由算法的说明立即可以推知。对于第二种情形，可以发现灰色顶点总是形成一条线性的后裔链，它对应于活动的 DFS-VISIT 调用栈；灰色顶点的数目等于最近发现的顶点在深度优先森林中的深度加 1。探寻总是从深度最深的灰色顶点开始，因而，到达另一个灰色顶点的边所达到的必是它的祖先。第三种情形处理其余的可能性；可以证明，如果 $d[u]<d[v]$，则
546 (u, v)就是一条正向边；如果 $d[u]>d[v]$，则(u, v)就是一条交叉边(见练习 22.3-4)。

在无向图中，由于(u, v)和(v, u)实际上是同一条边，上述的边分类可能会产生歧义。当出现这种情况时，边被被归为分类表中适用的第一种类型，对应地(见练习 22.3-5)，我们将根据算法的执行过程中，首先遇到的边是(u, v)还是(v, u)来对其进行分类。

下面来说明在对一个无向图进行深度优先搜索时，不会出现正向边和交叉边。

定理 22.10　在对一个无向图 G 进行深度优先搜索的过程中，G 的每一条边要么是树边，要么是反向边。

证明：设(u, v)为 G 中的任意一条边，并不失一般性，假定 $d[u]<d[v]$。于是，v 由于是在 u 的邻接表中，故必定在完成 u 之前(当 u 为灰色时)，就已经被发现并完成了。如果边(u, v)是首先沿着从 u 到 v 的方向被探寻到的，那么在该时刻之前，v 都是未被发现的(白色)，因为否则的话，必定已经沿 v 到 u 的方向探寻到了该边。于是，(u, v)即成为了一条树边。如果(u, v)是首先沿从 v 到 u 的方向被探寻到的，则由于该边被第一次探寻时，u 依然是灰色顶点，所以(u, v)是一条反向边。(证毕) ■

下面的几节中还将多次给出这些定理的应用。

练习

22.3-1　画一个 3×3 的图，图中行和列分别标上白(WHITE)、灰(GRAY)、黑(BLACK)。对每个单元(i, j)，试说明在对一个有向图进行深度优先搜索过程中的任何时刻，是否可能存在一条边，从颜色为 i 的顶点到颜色为 j 的顶点。对每一条可能的边，说明它可以属于哪种边类型。对无向图进行深度优先搜索的情形，也请画出这样的一张图。

22.3-2　说明深度优先搜索在图 22-6 上是如何进行的。假定 DFS 过程的第 5～7 行 for 循环按字母表顺序考察各个顶点，并假定每个邻接表都是按字母顺序排序的。说明每个顶点的发现和完成时间，并给出每条边的分类。

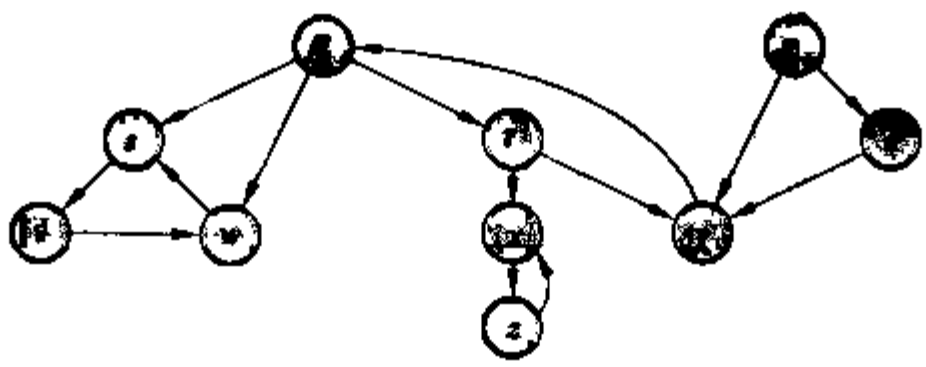


图 22-6　用于练习 22.3-2 和练习 22.5-2 的一个有向图

22.3-3　对图 22-4 中所示的深度优先搜索，给出其括号结构。

22.3-4　证明：边(u, v)是一条：

a)树边或前向边，当且仅当 $d[u]<d[v]<f[v]<f[u]$；

b)反向边，当且仅当 $d[v]<d[u]<f[u]<f[v]$；

c)交叉边，当且仅当 $d[v]<f[v]<d[u]<f[u]$。

22.3-5　证明：在一个无向图中，如果是根据在深度优先搜索中，(u, v)和(v, u)哪一个首先被遇到作为标准来将(u, v)归类为树边或反向边的话，就等价于根据边分类方案中的各类型的优先级来对它进行分类。

22.3-6　重写过程 DFS，利用一个栈来消除递归。

22.3-7　对于“在一个有向图 G 中，如果有一条从 u 到 v 的路径，并且，在对 G 的深度优先搜索中，如果有 $d[u]<d[v]$，则在所得到的深度优先森林中，v 是 u 的一个后裔”这一推测，给出它的一个反例。

22.3-8　对于“在一个有向图 G 中，如果有一条从 u 到 v 的路径，则任何深度优先搜索都必定能得到 $d[v]\leqslant f[u]$”这一推测，给出它的一个反例。

547
l
548

22.3-9 修改深度优先搜索的伪代码，输出有向图 G 中的每一条边及其类型。如果 G 是无向图，说明需要做出哪些改动(如果需要的话)。

22.3-10 解释在有向图中，对于一个顶点 u(即使 u 在 G 中既有入边又有出边)，是如何会最终落到一棵仅包含 u 的深度优先树中的。

22.3-11 证明：对无向图 G 的深度优先搜索可以用来识别出 G 的连通分支，且深度优先森林中所包含的树的数量与 G 中的连通分支的数量一样多。更精确地，说明如何修改深度优先搜索算法，使得每个顶点 v 都被赋予一个 1 到 k 之间的整型标记 $cc[v]$，其中 k 是 G 中连通分支的数目，从而使得 $cc[u]=cc[v]$，当且仅当 u 和 v 是在同一个连通分支中。

***22.3-12** 在一个有向图 $G=(V, E)$ 中，如果 $u \rightsquigarrow v$ 蕴含着对所有顶点 u、$v \in V$，至多有一条从 u 到 v 的简单路径，则称 G 是单连通的。给出一个有效的算法来判定一个有向图是否是单连通的。

22.4　拓扑排序

本节说明了如何运用深度优先搜索，对一个有向无回路图(有时称为 dag)进行拓扑排序。对有向无回路图 $G=(V, E)$ 进行拓扑排序后，结果为该图所有顶点的一个线性序列，满足如果 G 包含边 (u, v)，则在该序列中，u 就出现在 v 的前面(如果图是有回路的，就不可能存在这样的线性序列)。一个图的拓扑排序可以看成是图中所有顶点沿水平线排列而成的一个序列，使得所有的有向边均从左指向右。因此，拓扑排序不同于在第二部分中讨论的通常意义上的排序。

在很多应用中，有向无回路图用于说明事件发生的先后次序，图 22-7 即给出了一个实例，说明 Bumstead 教授早晨穿衣的过程。他必须先穿好某些衣服，才能再穿其他衣服(如先穿袜子后穿鞋)，其他的一些衣服则可以按任意次序穿戴(如袜子和裤子)；在图 22-7a 的有向无回路图中，有向边 (u, v) 表示衣服 u 必须先于衣服 v 穿戴。因此，该图的拓扑排序给出了一个穿衣的顺序。图 22-7b 说明了对该图进行拓扑排序后，将沿水平线方向形成一个顶点序列，使得图中所有有向
549 边均从左指向右。

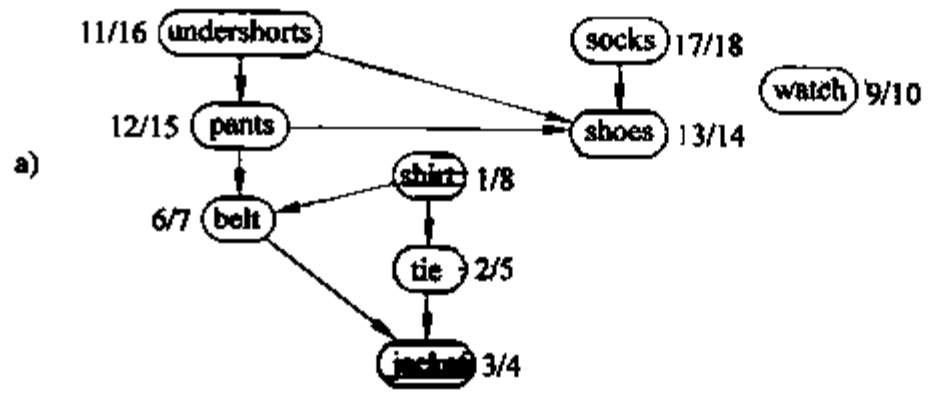


图 22-7　a) Bumstead 教授对他所穿的衣服进行了拓扑排序。每一条有向边 (u, v) 都意味着必须先穿衣服 u，再穿衣服 v。深度优先搜索中的发现和完成时间都在每个顶点的旁边示出。b) 经过拓扑排序后的同一图形。图中的各个顶点按照完成时间的递降顺序，至左向右排列。注意所有的有向边都是从左指向右的

下面给出的简单算法可以对一个有向无回路图进行拓扑排序。

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times f[v] for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

图 22-7b 说明了经拓扑排序的顶点以与其完成时刻相反的顺序出现。

因为深度优先搜索的运行时间为 $\Theta(V+E)$，而将 $|V|$ 个顶点中的每一个顶点插入链表所需时间为 $O(1)$，因此，执行拓扑排序的运行时间为 $\Theta(V+E)$。

为了证明上面算法的正确性，我们运用了下面有关有向无回路图的重要引理。

引理 22.11　一个有向图 G 是无回路的，当且仅当对 G 进行深度优先搜索时没有得到反向边。

证明：⇒：假设有一条反向边 (u, v)，那么在深度优先森林中，顶点 v 就是顶点 u 的祖先，因此，G 中从 v 到 u 必存在一条通路，这一通路和反向边 (u, v) 构成了一个回路。

⇐：假设 G 中包含一个回路 c。下面证明对 G 的深度优先搜索将产生一条反向边。设 v 是回路 c 中第一个被发现的顶点，且边 (u, v) 是 c 中通向 v 的边。在时刻 $d[v]$，c 中的顶点形成了一条从 v 到 u 的、由白色顶点所组成的通路。根据白色路径定理可知，在深度优先森林中，顶点 u 必是顶点 v 的后裔。因而，(u, v) 是一条反向边。（证毕）■

定理 22.12　TOPOLOGICAL-SORT(G)算法产生一个有向无回路图 G 的拓扑排序。

证明：假设对某一已知有向无回路 $G=(V, E)$ 运行过程 DFS，以便确定其顶点的完成时刻。只要证明对任一对不同顶点 u、$v \in V$，若 G 中存在着一条从 u 到 v 的边，则 $f[v]<f[u]$。考虑过程 DFS(G)所探寻的任何边 (u, v)，当探寻到该边时，顶点 v 不可能为灰色，否则 v 将成为 u 的祖先，(u, v) 将是一条反向边，和引理 22.11 矛盾。因此，v 必定是白色或黑色顶点。若 v 是白色，它就成为 u 的后裔，因此有 $f[v]<f[u]$。若 v 是黑色，则已经完成了探索，且 $f[v]$ 已设置。因为仍在探寻 u，还要为 $f[v]$ 赋时间戳。一旦这么做了后，就同样有 $f[v]<f[u]$，这样一来，对于有向无回路图中任意边 (u, v)，都有 $f[v]<f[u]$，从而定理得证。（证毕）■

练习

22.4-1　在练习 22.3-2 的假设下，说明 TOPOLOGICAL-SORT 运行于图 22-8 中所示的有向无回路图上时，所产生的顶点顺序。

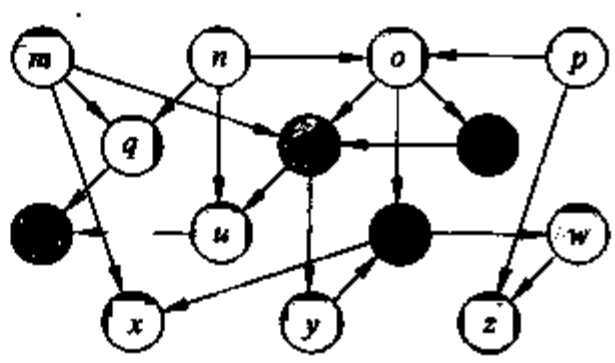


图 22-8　一个用于拓扑排序的有向无回路图

22.4-2　给出一个线性时间的算法，其输入为一个有向无回路图 $G=(V, E)$ 和两个顶点 s 和 t，返回 G 中从 s 到 t 的通路数目。例如，在图 22.8 中的有向无回路图中，从顶点 p 到顶点 v 之间恰有 4 条通路：pov，$poryv$，$posryv$ 以及 $psryv$。（读者所给出的算法仅需要给出通路的数目，不需要将它们一一列出。）

22.4-3　给出一个算法，用它来确定一个给定的无向图 $G=(V, E)$ 中是否包含一个回路。所给出算法的运行时间应为 $O(V)$，这一时间独立于 $|E|$。

22.4-4　证明或反证：如果一个有向图 G 包含回路，则 TOPOLOGICAL-SORT(G)能产生一个顶点的排序序列，它可以最小化“坏”边的数目。所谓坏边，即那些与所生成的顶点序列不一致的边。

22.4-5 在一个有向无回路图 $G=(V, E)$上，执行拓扑排序的另一种方法是重复地寻找一个入度为 0 的顶点，将该顶点输出，并将该顶点及其所有的出边从图中删除。解释如何来实现这一想法，才能使得它的运行时间为 $O(V+E)$。如果 G 中包含回路的话，这个算法在运行时会发生什么？

22.5　强连通分支

现在来考虑深度优先搜索的一种经典应用：把一个有向图分解为各强连通分支(strongly connected component)，本节将介绍如何使用两种深度优先搜索过程来进行这种分解。很多有关有向图的算法都是从这种分解步骤开始的。在分解之后，算法即在每一个强连通分支上独立地运行。最后，再根据各个分支之间的关系，将所有的解组合起来。

根据附录 B 中给出的有关知识可以知道，有向图 $G=(V, E)$的一个强连通分支就是一个最大的顶点集合 $C\subseteq V$，对于 C 中的每一对顶点 u 和 v，有 $u \rightsquigarrow v$ 和 $v \rightsquigarrow u$；亦即，顶点 u 和 v 是互相可达的。图 22-9 示出了一个例子。

在寻找图 $G=(V, E)$的强连通分支的算法中，用到了 G 的转置，其定义已经在练习 22.1-3 中给出：$G^{\mathrm{T}}=(V, E^{\mathrm{T}})$，$E^{\mathrm{T}}=\{(u, v): (v, u)\in E\}$。亦即，由 E^{T} 是由 G 中的边改变方向后所组成的。在给定图 G 的邻接表表示的情况下，建立 G^{T} 所需时间为 $O(V+E)$。有意思的是，G 和
552 G^{T} 有着完全相同的强连通分支，亦即，在 G 中，u 和 v 互为可达，当且仅当在 G^{T} 中它们互为可达。图 22-9b 示出了图 22-9a 中所示的图的转置，其中各强连通分支都着上了阴影。

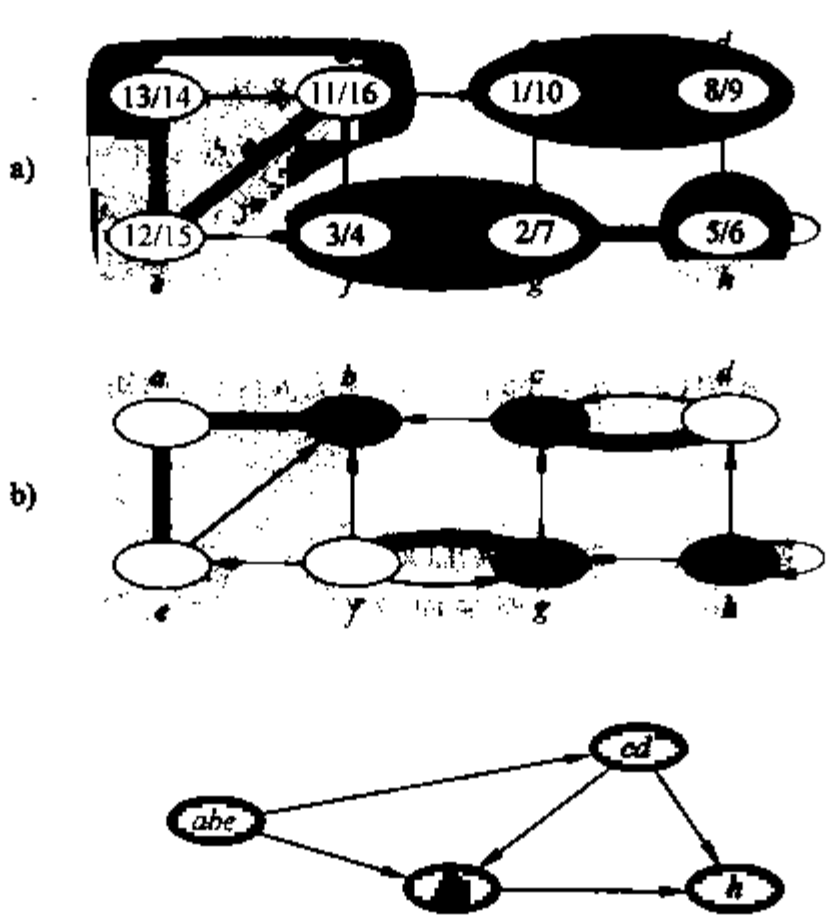


图 22-9　a)一个有向图 G。图中的阴影区域就是 G 的各强连通子图，对每个顶点都标出了其发现时刻与完成时刻，阴影覆盖的边为树边。b)G 的转置图 G^{T}。图中示出了 STRONGLY-CONNECTED-COMPONENTS 第 3 行计算出的深度优先森林，其中阴影覆盖的边是树边，每个强连通子图对应于一棵深度优先树。图中涂上深阴影的顶点 b、c、g 和 h 是对 G^{T} 进行深度优先搜索所产生的深度优先树的树根。c)把 G 的每个强连通子图缩减为每个分支只有单个顶点所得到的无回路子图 G^{SCC}

下列线性时间(即运行时间为 $\Theta(V+E)$)算法可以得出有向图 $G=(V, E)$强连通分支。该算法使用了两次深度优先搜索，一次在图 G 上进行，另一次在图 G^T 上进行。 553

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G)to compute finishing times f[u] for each vertex u
2  compute G^T
3  call DFS(G^T). but in the main loop of DFS, consider the vertices
       in order of decreasing f[u](as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

本算法背后的思想来自于分支图 $G^{SCC}=(V^{SCC}, E^{SCC})$的一个重要性质，下面就来定义这一性质。假设 G 的强连通分支为 C_1，C_2，…，C_k。顶点集 V^{SCC}为$\{v_1, v_2, \cdots, v_k\}$，对于 G 的每一个强连通分支 C_i，它都包含有一个顶点 v_i。如果对于某个 $x\in C_i$ 以及某个 $y\in C_j$，G 中包含了一条有向边(x, y)的话，则就有一条边$(v_i, v_j)\in E^{SCC}$。从另一个方面来看，收缩那些其关联的顶点都处于 G 的同一强连通分支内的边，即可得到图 G^{SCC}。图 22-9c 示出了图 22-9a 的分支图。

关键的性质是分支图是一个有向无回路图，下面的引理蕴含了这一性质。

引理 22.13　设 C 和 C' 是有向图 $G=(V, E)$中两个不同的强连通分支，设 u、$v\in C$，u'、$v'\in C'$，并假设 G 中存在着一条通路 $u \rightsquigarrow u'$。那么，G 中不可能还同时存在通路 $v' \rightsquigarrow v$。

证明：如果 G 中存在着一条通路 $v' \rightsquigarrow v$，则 G 中就有通路 $u \rightsquigarrow u' \rightsquigarrow v'$ 和通路 $v' \rightsquigarrow v \rightsquigarrow u$。于是，$u$ 和 v' 是互相可达的，这与 C 和 C' 是不同的强连通分支这一假设矛盾。 ■

下面将看到，在第二次深度优先搜索中，对于在第一次深度优先搜索中计算出来的顶点完成时间，当按照时间的降序来考虑各个顶点时，实际上就相当于按拓扑排序顺序访问分支图中各顶点(其中每一个都与 G 的一个强连通分支对应)。

因为 STRONGLY-CONNECTED-COMPONENTS 执行了两次深度优先搜索，当我们讨论 $d[u]$或$f[u]$时，就存在着歧义的可能性。在本节中，这些值始终是指第一次调用 DFS 时(第 1 行中)计算出来的发现和完成时间。

我们将发现和完成时间表示记号扩展至顶点集。如果 $U\subseteq V$，则定义 $d(U)=\min_{u\in U}\{d[u]\}$和 $f(U)=\max_{u\in U}\{f[u]\}$。亦即，$d(U)$和 $f(U)$分别是 U 中任何顶点的最早的发现时间和最晚的完成时间。 554

下面的引理及其推论给出了有关强连通分支和第一次深度优先搜索中完成时间的一个关键性质。

引理 22.14　设 C 和 C' 为有向图 $G=(V, E)$中的两个不同的强连通分支。假设有一条边 $(u, v)\in E$,其中 $u\in C$，$v\in C'$，则 $f(C)>f(C')$。

证明：根据在深度优先搜索中，强连通分支 C 和 C' 中的哪一个有着第一个被发现的顶点，可以区分出两种情况。

如果 $d(C)<d(C')$，设 x 为 C 中第一个被发现的顶点。在时刻 $d[x]$，C 和 C' 中的所有顶点都是白色的。在 G 中，有一条从 x 到 C 中每一个顶点的、仅由白色顶点所组成的通路。因为 $(u, v)\in E$,对任何顶点 $w\in C'$，在时刻 $d[x]$，G 中还有一条从 x 到 w 的、仅由白色顶点所组成的通路：$x \rightsquigarrow u \rightsquigarrow v \rightsquigarrow w$。根据白色路径定理，在深度优先树中，$C$ 和 C' 中的所有顶点都成为了 x 的后裔。根据推论 22.8，$f[x]=f(C)>f(C')$。

如果有 $d(C)>d(C')$，设 y 为 C' 中第一个被发现的顶点。在时刻 $d[y]$，C' 中的所有顶点都是白色的，且 G 中有一条从 y 到 C' 中每一个顶点的、仅由白色顶点所组成的通路．根据白色路

径定理，在深度优先树中，C'中的所有顶点都成为了 y 的后裔，并且，根据推论 22.8，$f[y]=f(C')$。在时刻 $d[y]$，C 中的所有顶点都是白色的。由于存在着一条从 C 到 C' 的边 (u, v)，根据引理 22.13 可知，不可能有一条从 C' 到 C 的路径。于是，C 中没有从 y 可达的顶点。因此，在时刻 $f[y]$，C 中的所有顶点仍然是白色的。于是，对于任何顶点 $w\in C$，有 $f[w]>f[y]$，这蕴含着 $f(C)>f(C')$。（证毕）■

下面的推论说明，G^T 中的每一条连接了两个不同强连通分支的边都是从（第一个在深度优先搜索中）有着更早的完成时间的分支指向有着较晚完成时间的分支。

推论 22.15　设 C 和 C' 为有向图 $G=(V, E)$ 中两个不同的强连通分支，假设存在着一条边 $(u, v)\in E^T$，其中 $u\in C$，$v\in C'$，则有 $f(C)<f(C')$。

证明：因为 $(u, v)\in E^T$，所以有 $(v, u)\in E$。又由于 G 和 G^T 的强连通分支是相同的，根据引理 22.14 可知，$f(C)<f(C')$。（证毕）■

要理解过程 STRONGLY-CONNECTED-COMPONENTS 为什么能正常工作，推论 22.15 是非常关键的。我们来看一看第二次在 G^T 上执行深度优先搜索时会发生些什么。首先从强连通分
555 支 C 开始，它的完成时间 $f(C)$ 是最大的。搜索从某个顶点 $x\in C$ 开始，访问 C 中的所有顶点。根据推论 22.15，在 G^T 中，没有从 C 到任何其他连通分支的边，因而从 x 开始的搜索不会访问任何其他分支中的顶点。于是，根为 x 的树仅包含了 C 中的顶点。在访问完 C 中的所有顶点后，第 3 行中的搜索选择另外某个强连通分支 C' 中的一个顶点作为根，C' 的完成时间 $f(C')$ 在所有除 C 之外的强连通分支中是最大的。接着，搜索又访问 C' 的所有顶点，但根据推论 22.15，在 G^T 中，从 C' 仅有的到其他分支的边必定是指向 C 的，而后者是已经被访问过的。一般来说，当在算法第 3 行中对 G^T 进行深度优先搜索时，如果访问了任何强连通分支，则从该分支出来的任何边都必定是指向那些已经被访问过的分支的。因此，每棵深度优先树中都恰好是一个强连通分支。下面的定理形式化了这一论证过程。

定理 22.16　过程 STRONGLY-CONNECTED-COMPONENTS(G) 可以正确地计算出有向图 G 的强连通分支。

证明：通过对在第 3 行中对 G^T 进行深度优先搜索中发现的深度优先树的数目进行归纳，可以证明每棵树中的顶点都形成了一个强连通分支。归纳假设是在第 3 行中产生的头 k 棵树都是强连通分支。归纳的基础（即 $k=0$ 的情况）是显而易见的。

在归纳步骤中，假定第 3 行中产生的头 k 棵深度优先树都是强连通分支，在此基础上来考虑产生的第 $(k+1)$ 树。设这棵树的根为顶点 u，并设 u 位于强连通分支 C 中。根据在第 3 行的深度优先搜索中选择根的方式，对于除 C 而外的任何其他进一步访问的强连通分支 C'，都有 $f[u]=f(C)>f(C')$。根据归纳假设，当搜索访问到 u 时，C 中所有其他顶点都是白色的。于是，根据白色路径定理，在 C 的深度优先树中，C 中的所有其他顶点都是 u 的后裔。此外，根据归纳假设和推论 22.15，G^T 中的任何离开 C 的边必定是指向已被访问过的强连通分支。于是，在对 G^T 进行深度优先搜索的过程中，对于除 C 之外的任何其他强连通分支中来说，其顶点都不可能是 u 的后裔。因此，在 G^T 中，根为 u 的深度优先树中的顶点恰好形成了一棵强连通分支，这样完成了归纳步骤，从而定理得证。（证毕）■

还可以这样来分析第二次深度优先搜索操作是如何进行的。考虑一下 G^T 的分支图 $(G^T)^{SCC}$。如果将第二次深度优先搜索过程中访问的每个强连通分支都映射至 $(G^T)^{SCC}$ 的一个顶点上，则
556 $(G^T)^{SCC}$ 的各个顶点会以一种与拓扑排序相反的顺序被访问。如果将 $(G^T)^{SCC}$ 的所有边掉一个方向，就可以得到图 $((G^T)^{SCC})^T$。因为 $((G^T)^{SCC})^T=G^{SCC}$（见练习 22.5-4），第二次深度优先搜索就按照拓扑排序的顺序来访问 G^{SCC} 的各个顶点。

练习

22.5-1　当在一个图中加入一条新的边后，其强连通分支的数目会如何变化？

22.5-2　说明过程 STRONGLY-CONNECTED-COMPONENTS 在图 22-6 上的工作过程。特别地，说明第 1 行中计算出来的完成时间和第 3 行中产生的森林是怎样的。假定 DFS 的第 5～7行中的循环是按字母顺序来考虑各个顶点的，且邻接表也是按字母顺序排序的。

22.5-3　Deaver 教授声称，用于强连通分支的算法可以这样简化，即在第二次深度优先搜索中使用原图(而不是其转置图)，并按完成时间递增的顺序来扫描各个顶点。这位教授的说法正确吗？

22.5-4　证明：对于任何有向图 G，都有 $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}}=G^{\mathrm{SCC}}$。亦即，$G^{\mathrm{T}}$ 的分支图的转置与 G 的分支图是一样的。

22.5-5　给出一个 $O(V+E)$时间的算法，以计算一个有向图 $G=(V, E)$的分支图。注意在算法产生的分支图中，两个顶点之间至多只能有一条边。

22.5-6　给定一个有向图 $G=(V, E)$，解释如何生成另一个图 $G'=(V, E')$，使得(a)G'有着与 G 相同的强连通分支；(b)G'有着与 G 相同的分支图；(c)E'应尽可能地小。给出一个能计算出 G'的快速算法。

22.5-7　在一个有向图 $G=(V, E)$，如果对所有顶点对 u、$v\in V$，都有 $u\rightsquigarrow v$ 或 $v\rightsquigarrow u$，则图 G 称为半连通(semiconnected)。给出一个有效的算法，确定一个图 G 是否是半连通的。证明你所给出的算法是正确的，并分析其运行时间。

557

思考题

22-1　通过广度优先搜索对边进行分类

深度优先森林把图的边分为树边、正向边、反向边和交叉边四种类型。应用广度优先树，同样可以把从搜索源顶点可达的边分为同样的四种类型。

a)证明在对无向图的广度优先搜索中，存在下列性质：

1)不存在正向边和反向边

2)对于每条树边(u, v)，有 $d[v]=d[u]+1$

3)对于每条交叉边(u, v)，有 $d[v]=d[u]$或 $d[v]=d[u]+1$

b)证明在对有向图的广度优先搜索中，下列性质成立：

1)不存在正向边

2)对于每一条树边(u, v)，有 $d[v]=d[u]+1$

3)对于每一条交叉边(u, v)，有 $d[v]\leqslant d[u]+1$

4)对于每一条反向边(u, v)，有 $0\leqslant d[v]\leqslant d[u]$

22-2　挂接点、桥以及双连通分支

设 $G=(V, E)$是一个无向连通图，如果去掉 G 的某个顶点后 G 就不再是连通图了，这样的顶点称为挂接点(articulation point)。如果去掉某一边后，G 就不再成为连通图了，这样的边称为桥(bridge)。G 的双连通分支(biconnected component)是满足以下条件的一个最大边集，即该集合中的任意两条边都位于同一个公共简单回路上。图 22-10 给出了这几个定义的图示。可以用深度优先搜索来确定挂接点、桥以及双连通分支。设 $G_\pi=(V, E_\pi)$是 G 的一棵深度优先树。

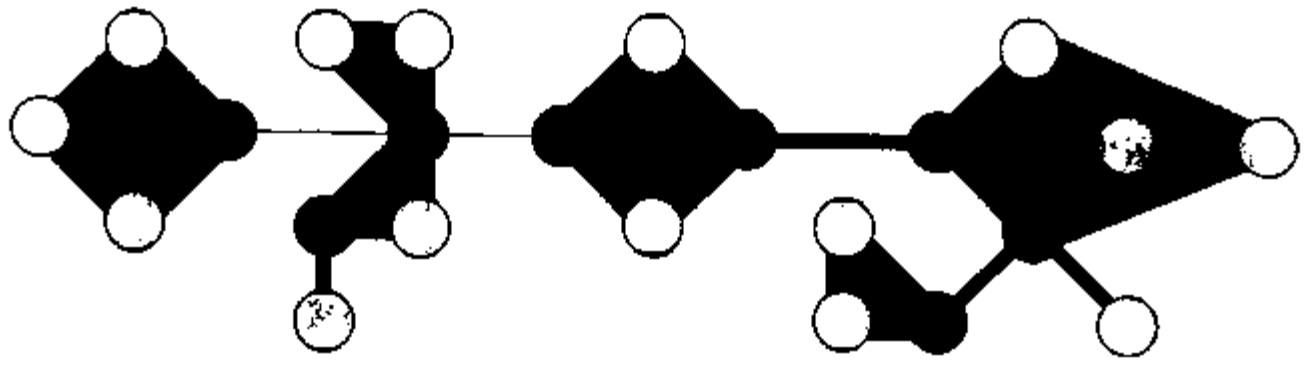


图 22-10　思考题 22-2 中所用到的无向连通图的挂接点、桥以及双连通分支。图中深阴影的顶点为挂接点，深阴影的边为桥，图中阴影覆盖的区域中的边(旁边示出了一个 *bcc* 编号)表示双连通分支

a)证明 G_π 的根是 G 的挂接点，当且仅当在 G_π 中该根顶点至少有两个子女。

b)设 v 是 G_π 中的某一非根顶点，证明 v 是 G 的挂接点当且仅当 v 有一个子顶点 s，使得不存在从 s 或 s 的任何后裔顶点指向 v 的某个真祖先顶点的反向边。

c)设

$$low[v] = \min\begin{cases} d[v], \\ d[w]：对\ v\ 的后裔\ u,(u,w)\ 是反向边 \end{cases}$$

试说明对所有顶点 $v\in V$，如何在 $O(E)$时间内计算出 $low[v]$。

d)说明如何在 $O(E)$时间内计算出所有的挂接顶点。

e)证明：G 的某边是桥，当且仅当它不属于 G 的任何简单回路。

f)试说明如何在 $O(E)$时间内计算出 G 中所有的桥。

g)证明 G 的双连通分支划分了 G 的非桥边。

h)给出一个运行时间为 $O(E)$的算法，它对 G 的每条边 e 标示一个正整数 $bcc[e]$，使得 $bcc[e]=bcc[e']$当且仅当 e 和 e'在同一个双连通分支中。

22-3　欧拉回路

有向强连通图 $G=(V, E)$的欧拉回路是指通过 G 中每条边仅一次(但可以访问某个顶点多次)的一个回路。

a)证明：图 G 具有欧拉回路，当且仅当每一个顶点 $v\in V$ 的入度和出度都相等；

b)给出一个 $O(E)$时间的算法，它能够在图 G 中存在着欧拉回路的情况下，找出这一回路。(提示：将边不相交的回路进行合并。)

558
~
559

22-4　可达性

设 $G=(V, E)$是一个有向图，图中每个顶点 $u\in V$ 都标记有唯一的整数 $L(u)$，该整数取自集合 $=\{1, 2, \cdots, |V|\}$。对每个顶点 $v\in V$，设 $R(u)=\{v\in V: u \rightsquigarrow v\}$为从 u 可达的顶点集合。定义 $\min(u)$为 $R(u)$中标记值最小的顶点。亦即，$\min(u)$是这样的一个顶点 v，使得 $L(v)=\min\{L(w), w\in R(u)\}$。请给出一个 $O(V+E)$时间的算法，对所有的顶点 $u\in V$，该算法可以计算出 $\min(u)$。

本章注记

Even[87]和 Tarjan[292]是图算法方面很好的参考文献。

广度优先搜索是由 Moore[226]在如何在迷宫中寻找出路这一背景中提出的。Lee[198]在线路板上布线这一背景中也独立地提出了相同的算法。

Hopcroft 和 Tarjan[154]建议，对于稀疏矩阵，与邻接矩阵表示相比，采用邻接表表示法要更好一些。是他们首先认识到了深度优先搜索在算法上的重要性。20 世纪 50 年代后期以来，深度优先搜索得到了广泛的应用，尤其是用在人工智能程序中。

Tarjan[289]给出了一个用于寻找强连通分支的线性时间算法。22.5 节中有关强连通分支的算法是根据 Aho、Hopcroft 和 Ullman[6]而改编的，他们将这算法归功于 S. R. Kosaraju(未发表)和 M. Sharir[276]。Gabow[101]也提出了一个有关强连通分支的算法，它的做法是收缩回路，并利用两个栈来使得它以线性时间运行。Knuth[182]首先提出了一个用于实现拓扑排序的线性时间算法。

560

第 23 章　最小生成树

在设计电子线路时，常常要把数个元件的引脚连在一起，使其电位相同。要使 n 个引脚互相连通，可以使用 $n-1$ 条连接线，每条连接线连接两个引脚。在各种连接方案中，通常希望找出连接线最少的接法。

可以把这一接线问题模型化为一个无向连通图 $G=(V, E)$，其中 V 是引脚集合，E 是每对引脚之间可能互联的集合。对图中每一条边 $(u, v)\in E$，都有一个权值 $w(u, v)$ 表示连接 u 和 v 的代价(需要的接线数目)。我们希望找出一个无回路的子集 $T\subseteq E$，它连接了所有的顶点，且其权值之和

$$w(T)=\sum_{(u,v)\in T} w(u,v)$$

为最小。因为 T 无回路且连接所有的顶点，所以它必然是一棵树，称为生成树(spanning tree)，因为它"生成"了图 G。把确定树 T 的问题称为最小生成树问题。[⊖]图 23-1 展示一个连通图及其最小生成树的实例。

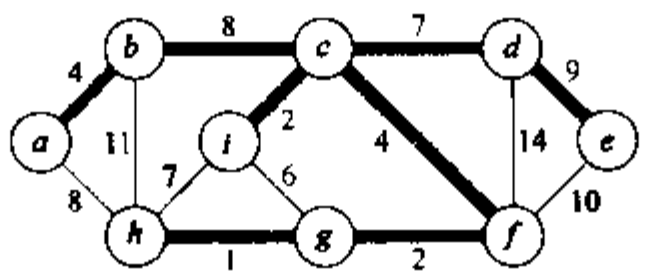


图 23-1　一个连通图的最小生成树。图中显示各条边的权值，带阴影的边为最小生成树的边。树中各边的权值之和为 37。最小生成树并不是唯一的：用边 (a, h) 替代边 (b, c) 得到另外一棵最小生成树，其中各边的权值之和也是 37

在本章中，要介绍解决最小生成树问题的两种算法：Kruskal 算法和 Prim 算法。这两种算法中都使用普通的二叉堆，都很容易达到 $O(E\lg V)$ 的运行时间。通过采用斐波那契堆，Prim 算法的运行时间可以减少到 $O(E+V\lg V)$；如果 $|V|$ 远小于 $|E|$ 的话，这将是对该算法的较大改进。

这两个算法都是贪心算法，有关贪心算法的介绍可见第 16 章。在算法的每一步中，都必须在几种可能性中选择一种。贪心策略的思想是选择当时最佳的可能性。一般来说，这种策略不一定能保证找到全局最优解。然而，对于最小生成树问题来说，却可以证明某些贪心策略的确可以获得具有最小权值的生成树。本章与第 16 章是独立的，但本章中给出的贪心方法是第 16 章中所介绍的理论思想的一种经典应用。

23.1 节介绍一种"通用"的最小生成树的算法，该算法通过每次加入一条边来逐渐形成一棵生成树。23.2 节说明实现上述通用算法的两种途径。第一种算法由 Kruskal 提出，它类似于 21.1 节中的连通分支算法。另一种算法由 Prim 提出，它类似于 24.3 节中讨论的 Dijkstra 最短路径算法。

⊖ 术语"最小生成树"其实是"最小权值生成树"的缩略。我们并不是要使 T 中边的数目最小化，因为根据定理 B.2，所有生成树中的边数都恰好是 $|V|-1$。

23.1　最小生成树的形成

假设已知一个无向连通图 $G=(V, E)$，其权值函数为 $w: E \to \mathbf{R}$。我们的目的是找到图 G 的一棵最小生成树。本章所讨论的两种算法都运用贪心方法，但在如何运用这种方法上有所不同。

下面给出的"通用"算法采用这种贪心策略，它在每一个步骤中都形成最小生成树的一条边。算法维护一个边的集合 A，保持以下的循环不变式：

在每一次循环迭代之前，A 是某个最小生成树的一个子集。

在算法的每一步中，确定一条边 (u, v)，使得将它加入集合 A 后，仍然不违反这个循环不变式；亦即，$A \cup \{(u, v)\}$ 仍然是某一个最小生成树的子集。称这样的边为 A 的安全边(safe edge)，因为可以安全地把它添加到 A 中，而不会破坏上述的循环不变式。

561
~
562

```
GENERIC-MST(G, w)
1  A ← ∅
2  while A does not form a spanning tree
3       do find an edge(u, v) that is safe for A
4          A ← A∪{(u, v)}
5  return A
```

以上算法是这样来使用循环不变式的：

初始化：在第 1 行之后，集合 A 显然满足循环不变式。

保持：第 2～4 行的循环中加入的仅是安全边，从而保持循环不变式。

终止：所有加入到 A 中的边都位于一棵最小生成树中，因而，第 5 行返回的集合 A 必定是一棵最小生成树。

在这个算法中，最棘手的部分自然是第 3 行的寻找安全边。这样的一条安全边是必定存在的，因为在执行第 3 行时，根据循环不变式可知，存在着一棵生成树 T，满足 $A \subseteq T$。在 while 循环体内，A 必定是 T 的一个真子集，于是，必定存在着一条边 $(u, v) \in T$，满足 $(u, v) \notin A$，且 (u, v) 对 A 来说是安全的。

在本节的余下部分中，将提出一条识别安全边的规则(定理 23.1)，下一节将讨论运用这一规则来寻找安全边的两种有效算法。

首先来定义几个概念。无向图 $G=(V, E)$ 的一个割 $(S, V-S)$ 是对 V 的一个划分。图 23-2 中示出了这一概念。当一条边 $(u, v) \in E$ 的一个端点属于 S，而另一个端点属于 $V-S$ 时，则称边 (u, v) 通过割 $(S, V-S)$。如果一个边的集合 A 中没有边通过某一割，则说该割不妨害边集 A。如果某条边的权值是通过一个割的所有边中最小的，则称该边为通过这个的割的一条轻边(light edge)。要注意在若干条边的权值相等的情况下，可能会存在着多条通过某一割的轻边。从更一般的来讲，如果一条边是满足某一性质的所有边中具有最小权值的边，就称该边为满足该性质的一条轻边。

下面的定理给出了识别安全边的一条规则。

定理 23.1　设图 $G=(V, E)$ 是一个无向连通图，并且在 E 上定义了一个具有实数值的加权函数 w。设 A 是 E 的一个子集，它包含于 G 的某个最小生成树中。设割 $(S, V-S)$ 是 G 的任意一个不妨害 A 的割，且边 (u, v) 是通过割 $(S, V-S)$ 的一条轻边，则边 (u, v) 对集合 A 来说是安全的。

证明：设 T 是包含 A 的一棵最小生成树，并假定 T 不包含轻边 (u, v)，因为若包含的话，就完成证明了。我们将运用"剪切-粘贴技术"来建立另一棵包含 $A \cup \{(u, v)\}$ 的最小生成树 T'，从而证明 (u, v) 是 A 的一条安全边。

563

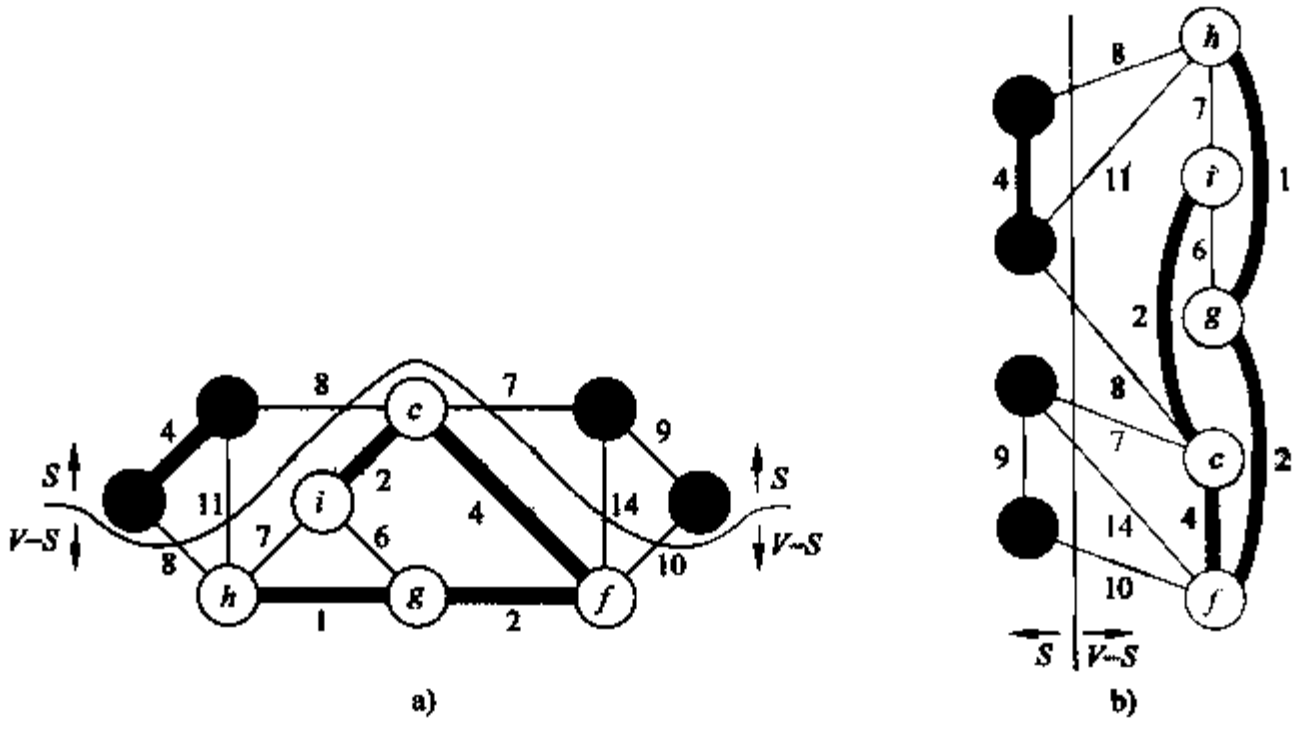


图 23-2　从两种途径来观察图 23-1 所示图的割$(S, V-S)$。a)集合 S 中的顶点以黑色示出，$V-S$ 中的顶点以白色示出。连接白色和黑色顶点的那些边为通过该割的边。边(d, c)为通过该割的唯一一条轻边。子集 A 包含阴影覆盖的那些边，注意，由于 A 中没有边通过割，所以割$(S, V-S)$不妨害 A。b)在同一张图中，将集合 S 中的顶点放在图的左边，集合 $V-S$ 中的顶点放在图的右边。如果某条边使左边的顶点与右边的顶点相连，则称该边通过该割

在 T 中，边(u, v)利用从 u 到 v 通路 p 上的边形成了一个回路，如图 23-3 中所示。由于 u 和 v 处于割$(S, V-S)$的相对的边上，因此在 T 中的通路 p 上，至少存在一条边也通过割。设 (x, y)为满足此条件的边。因为割不妨害 A，所以边(x, y)不属于 A；又因为(x, y)处于 T 中从 u 到 v 的唯一通路上，所以，去掉边(x, y)就会把 T 分成两个子图。这时，加入边(u, v)就可把它们重新连接起来，以形成一棵新的生成树 $T' = T - \{(x, y)\} \cup \{(u, v)\}$。

下一步来证明 T'是一棵最小生成树。因为(u, v)是通过割$(S, V-S)$的一条轻边，且边(x, y)也通过该割，所以有 $w(u, v) \leqslant w(x, y)$，从而有：

$$w(T') = w(T) - w(x, y) + w(u, v) \leqslant w(T)$$

564　但是，T 是一棵最小生成树，所以有 $w(T) \leqslant w(T')$，因此，T'必定也是一棵最小生成树。

接下来，还要证明(u, v)实际上是 A 的一条安全边。由于 $A \subseteq T$ 且$(x, y) \notin A$，故有 $A \subseteq T'$，则 $A \cup \{(u, v)\} \subseteq T'$，而 T'是最小生成树，因而，(u, v)对 A 是安全的。(证毕) ■

定理 23.1 使我们可以更好地理解算法 GENERIC-MST 在连通图 $G=(V, E)$上的执行流程。在算法的执行过程中，集合 A 始终是无回路的，否则，包含 A 的最小生成树将包含一个环，这样一来就形成了矛盾。在算法执行过程中的任何一个时刻，图 $G_A=(V, A)$是一个森林，G_A 的每一个连通分支都是一棵树(其中的某些树可能只包含一个顶点，例如在算法开始时，A 为空集，森林中包含 $|V|$ 棵树，每个顶点对应于一棵树)。此外，对 A 安全的任何边(u, v)都连接了 G_A 中不同的连通分支，这是由于 $A \cup \{(u, v)\}$必须是无回路的。

随着最小生成树的 $|V|-1$ 条边相继被确定，GENERIC-MST 中第 2～4 行的循环也随之要执行 $|V|-1$ 次。初始状态下，$A=\varnothing$，G_A 中有 $|V|$ 棵树，每次迭代过程均将减少一棵树，当森林中只包含一棵树时，算法的执行终止。

下面给出的是定理 23.1 的一个推论，23.2 节中要介绍的两种算法都用到了这一推论。

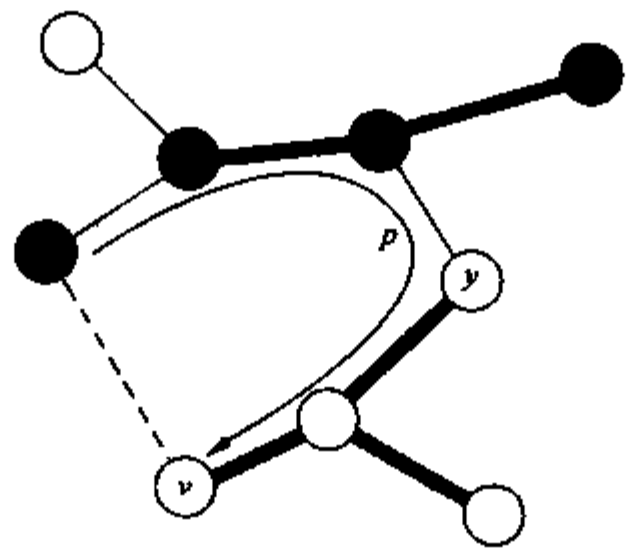


图 23-3　定理 23.1 的证明。S 中的顶点为黑色，$V-S$ 中的顶点为白色。图中示出了最小生成树 T 中的边，但未示出图 G 中的边。A 中的边加了阴影，(u, v)是一条通过割$(S, V-S)$的轻边。边(x, y)是 T 中从 u 到 v 的唯一路径 p 上的一条边。通过将边(x, y)从 T 中去掉，并加入边(u, v)，就形成了一棵最小生成树 T'，它包含了边(u, v)

推论 23.2　设 $G=(V, E)$是一个无向连通图，并且在 E 上定义了相应的实数值加权函数 w。设 A 是 E 的子集，且包含于 G 的某一最小生成树中。设 $C=(V_C, E_C)$为森林 $G_A=(V, A)$的一个连通分支(树)。如果边(u, v)是连接 C 和 G_A 中其他某连通分支的一条轻边，则(u, v)对集合 A 来说是安全的。

565

证明：因为割$(V_C, V-V_C)$不妨害 A，(u, v)是该割的一条轻边。因此，(u, v)对 A 来说是安全的。(证毕)　■

练习

23.1-1　设(u, v)是图 G 中的最小权边。证明：(u, v)属于 G 的某一最小生成树。

23.1-2　Sabatier 教授推测定理 23.1 有下面这一逆定理：设 $G=(V, E)$是一个连通无向图，并在 E 上定义了一个实值的权值函数 w。设 A 为 E 的一个子集，它包含于 G 的某个最小生成树中。设$(S, V-S)$为 G 的任意一个不妨害 A 的割，并设(u, v)是 A 的一条通过$(S, V-S)$的安全边。那么，(u, v)就是该割的一条轻边。请给出一个反例来证明教授的推测是不正确的。

23.1-3　证明：如果一条边(u, v)被包含在某一最小生成树中，那么它就是通过图的某个割的轻边。

23.1-4　给出一个图的例子，使得边集$\{(u, v)$：存在着一个割$(S, V-S)$，使得(u, v)是一条通过$(S, V-S)$的轻边$\}$不会形成一棵最小生成树。

23.1-5　设 e 是图 $G=(V, E)$的某个回路上的一条最大权边。证明：存在着 $G'=(V, E-\{e\})$的一棵最小生成树，它也是 G 的最小生成树。亦即，存在着 G 的不包含 e 的最小生成树。

23.1-6　证明：一个图有唯一的最小生成树，如果对于该图的每一个割，都存在着通过该割的唯一一条轻边。另外，给出一个反例来证明其逆命题不成立。

23.1-7　论证：如果图中所有边的权值都是正的，那么，任何连接所有顶点、且有着最小总权值的边的子集必为一棵树。请给出一个例子来说明如果允许某些权值非正的话，这一结论就不成立了。

566

23.1-8　设 T 是图 G 的一棵最小生成树，L 是 T 中各边权值的一个已排序的列表。证明：对于 G 的任何其他最小生成树 T'，L 也是 T'中各边权值的一个已排序的列表。

23.1-9　设 T 是图 $G=(V, E)$的一棵最小生成树，V'是 V 的一个子集。设 T'为 T 的一个基于 V'

的子图，G'为G的一个基于V'的子图。证明：如果T'是连通的，则T'是G'的一棵最小生成树。

23.1-10 给定一个图G和一棵最小生成树T，假定减小了T中某一边的权值。证明：T仍然是G的一棵最小生成树。更为形式地，设T是G的一棵最小生成树，其各边的权值由权值函数w给出。选择一条边$(x, y) \in T$和一个正数k，并用下式来定义权值函数w'：

$$w'(u,v) = \begin{cases} w(u,v) & \text{如果}(u,v) \neq (x,y) \\ w(x,y) - k & \text{如果}(u,v) = (x,y) \end{cases}$$

证明：T是G的一棵最小生成树，其各边的权值由w'给出。

***23.1-11** 给定一个图G和一棵最小生成树T，假定减小了不在T中的某条边的权值。请给出一个算法，来寻找经过修改的图中的最小生成树。

23.2 Kruskal 算法和 Prim 算法

本节所介绍的两种最小生成树算法是对上一节所介绍的通用算法的细化。它们均采用了一个特定的规则来确定 GENERIC-MST 算法第 3 行所描述的安全边。在 Kruskal 算法中，集合A是一个森林，加入集合A中的安全边总是图中连接两个不同连通分支的最小权边。在 Prim 算法中，集合
567 A仅形成单棵树，添加入集合A的安全边总是连接树与一个不在树中的顶点的最小权边。

Kruskal 算法

Kruskal 算法直接基于第 23.1 节中给出的通用最小生成树算法的基础。该算法找出森林中连接任意两棵树的所有边中，具有最小权值的边(u, v)作为安全边，并把它添加到正在生长的森林中。设C_1和C_2表示边(u, v)连接的两棵树，因为(u, v)必是连接C_1和其他某棵树的一条轻边，所以由推论 23.2 可知，(u, v)对C_1来说是安全边。Kruskal 算法同时也是一种贪心算法，因为在算法的每一步中，添加到森林中的边的权值都是尽可能小的。

Kruskal 算法的实现类似于第 21.1 节中计算连通分支的算法，它采用了一种不相交集合数据结构，以维护几个互相不相交的元素集合。每一个集合包含了当前森林中某棵树的顶点，操作 FIND-SET(u)返回包含u的集合中的一个代表元素。于是，通过测试 FIND-SET(u)是否等同于 FIND-SET(v)，就可以确定顶点u和v是否属于同一棵树。通过过程 UNION，可以实现树与树的合并。

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v∈V[G]
3       do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge(u, v)∈E, taken in nondecreasing order by weight
6       do if FIND-SET(u)≠FIND-SET(v)
7            then A ← A∪{(u, v)}
8                 UNION(u, v)
9  return A
```

Kruskal 算法的工作流程如图 23-4 中所示。第 1～3 行将集合A初始化为空集，并建立$|V|$棵树，每棵树都包含了图的一个顶点。在第 4 行中，根据权值的非递减顺序，对E中的边进行排序。在第 5～8 行的 for 循环中，首先检查每条边(u, v)，其端点u和v是否属于同一棵树。如果是，把(u, v)加入森林中就会形成一个回路，所以放弃边(u, v)。否则，说明两个顶点分属于不同的树，由第 7 行把边加入集合A中，第 8 行对两棵树中的顶点进行合并。

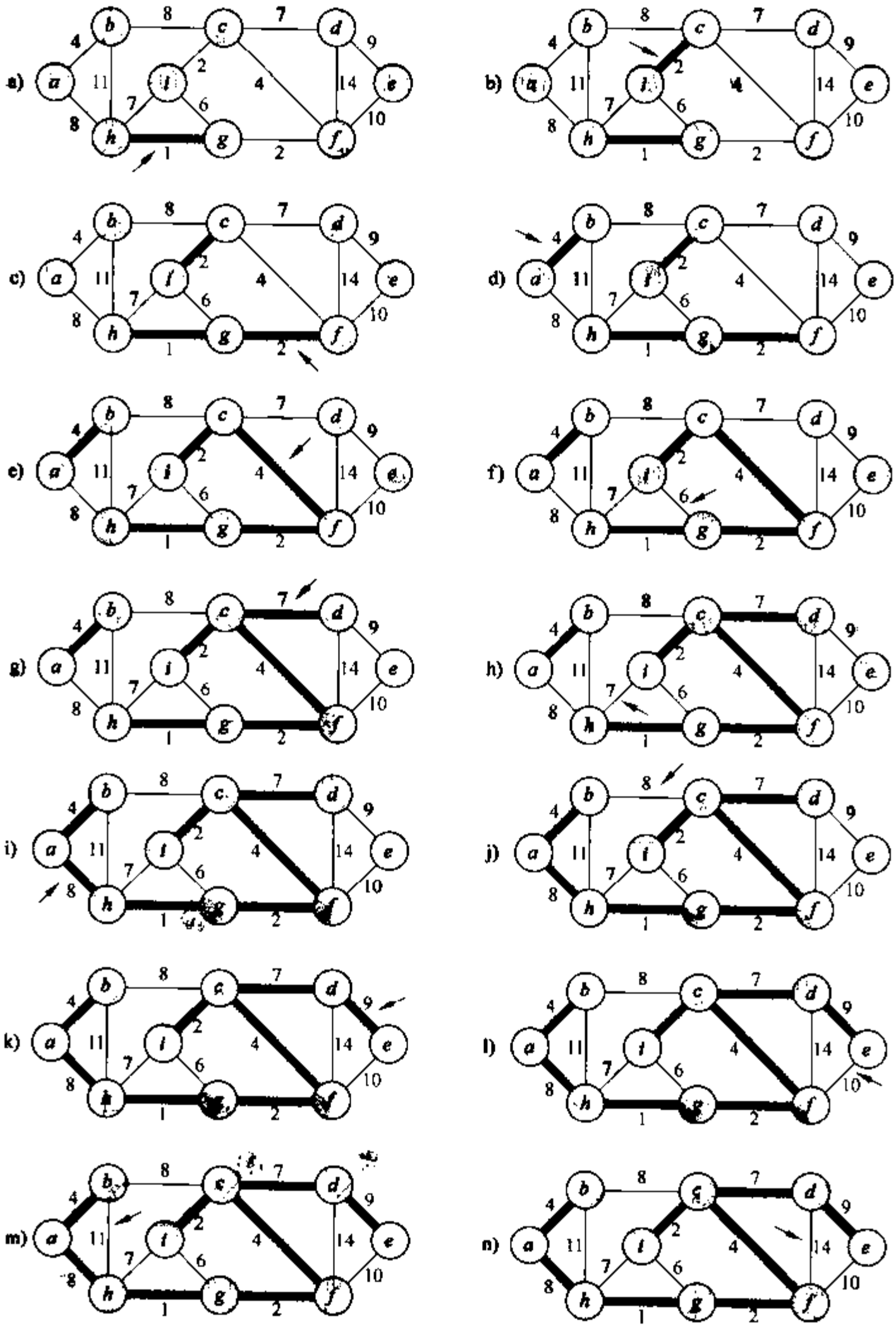


图 23-4　Kruskal 算法在图 23-1 上的执行过程。阴影边属于正在生长的森林 A。算法根据权值的排序顺序来考察各条边。在算法的每一步中，都有一个箭头指向当前正在被考察的边。如果该边将森林中的两棵树合并了起来，它就被添加到森林中，从而完成对两棵树的合并

Kruskal算法在图 $G=(V, E)$上的运行时间取决于不相交集合数据结构是如何实现的。假设采用第21.3节中所述的按秩结合和路径压缩的启发式方法，来实现不相交集合森林。由于从渐近意义上来说，这是最快的实现方法。第1行中对 A 的初始化需要占用 $O(1)$时间，第4行中对各条边进行排序需要的运行时间为 $O(E \lg E)$；(过一会儿，再来解释在第2～3行的 for 循环中，$|V|$ 个 MAKE-SET 操作的代价。)第5～8行中的 for 循环要在不相交集合森林上，执行 $O(E)$次 FIND-SET 和 UNION 操作。和 $|V|$ 个 MAKE-SET 操作一起，这些操作所需的总时间为 $O((V+E)\alpha(V))$，其中 α 函数为21.4节中定义的一个增长极为缓慢的函数。因为假定 G 是连通的，故有 $|E| \geqslant |V|-1$，因而不相交集合操作所需时间为 $O(E\alpha(V))$。此外，由于 $\alpha(|V|) \leqslant O(\lg V) = O(\lg E)$，故 Kruskal 算法总的运行时间为 $O(E \lg E)$。由于 $|E| < |V|^2$，因而有 $\lg |E| = O(\lg V)$，于是，也可以将 Kruskal 算法的运行时间重新表述为 $O(E \lg V)$。

Prim 算法

如 Kruskal 算法一样，Prim 算法也是第23.1节讨论的通用最小生成树算法的特例。Prim 算法的执行非常类似于寻找图的最短路径的 Dijkstra 算法(详见第24.3节)。Prim 算法的特点是集合 A 中的边总是形成单棵树。如图23-5所示，树从任意根顶点 r 开始形成，并逐渐生成，直至该树覆盖了 V 中的所有顶点。在每一步，一条连接了树 A 与 $G_A=(V, A)$中某孤立顶点的轻边被加入到树 A 中。由推论23.2可知，该规则仅加入对 A 安全的边，因此当算法终止时，A 中的边就形成了一棵最小生成树。因为每次添加到树中的边都是使树的权尽可能小的边，因此，上述策略也是“贪心”的。

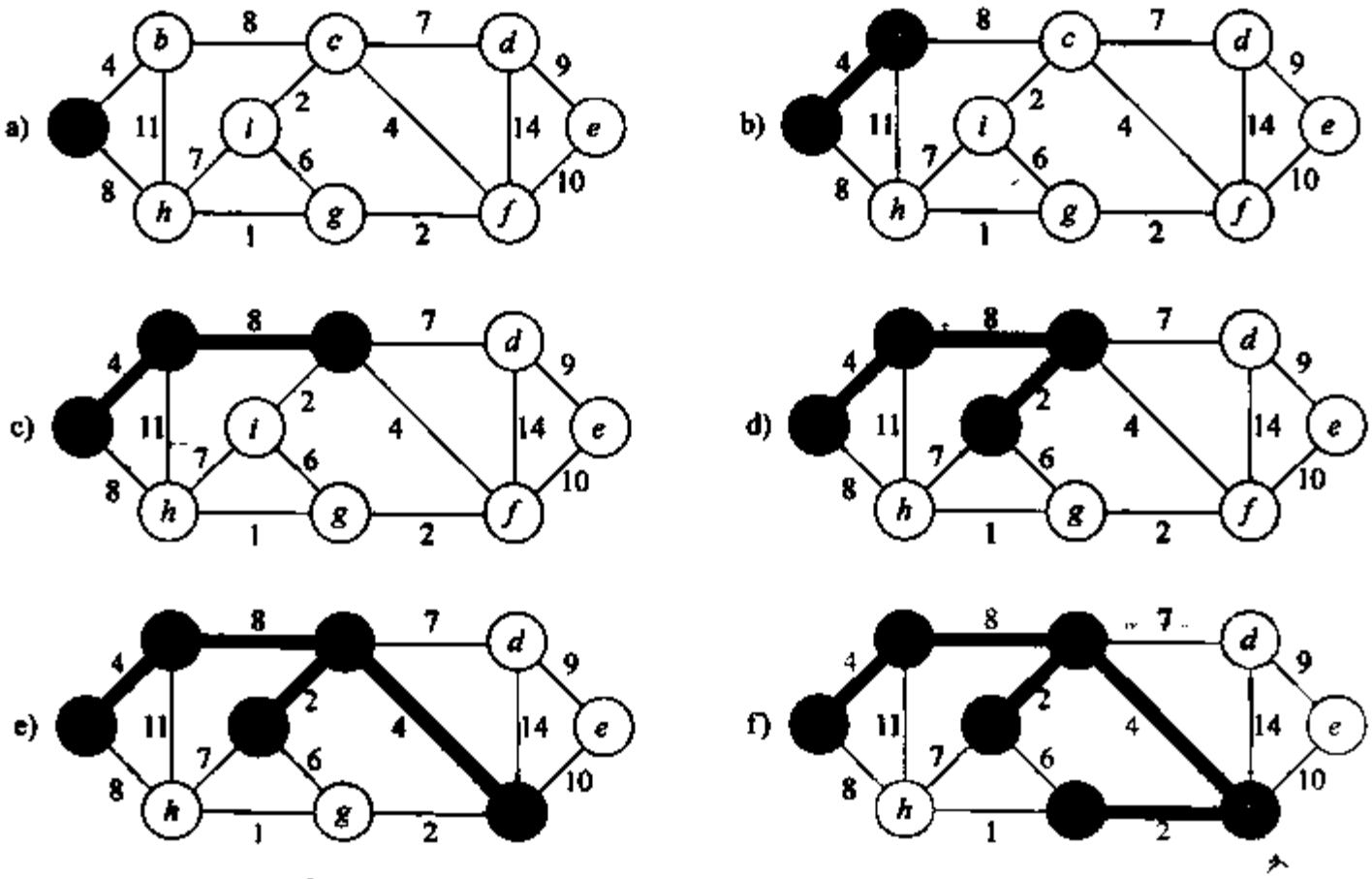


图23-5　Prim 算法在图23-1所示图上的执行过程。根顶点为 a。阴影边处于正在生长的树中，树中的顶点以黑色示出。在算法执行过程的每一步中，树中的各顶点构成了图的一个割，而通过该割的一条轻边被添加到树中。例如，在第二步中，算法可以选择将边 (b, c)或边 (a, h)加入到树中，因为两者都是通过割的轻边

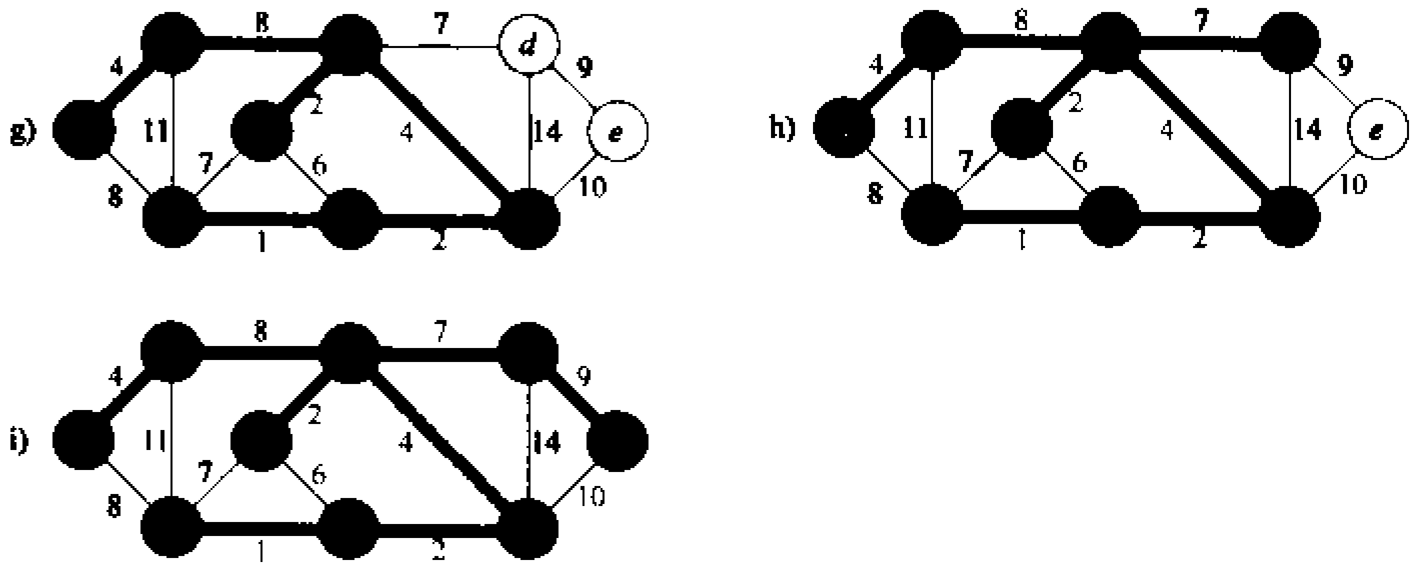


图 23-5 （续）

有效实现 Prim 算法的关键是设法较容易地选择一条新的边，将其添加到由 A 的边所形成的树中。在下面的伪代码中，算法的输入是连通图 G 和待生成的最小生成树的根 r。在算法的执行过程中，不在树中的所有顶点都放在一个基于 *key* 域的最小优先级队列 Q 中。对每个顶点 v 来说，$key[v]$是所有将 v 与树中某一顶点相连的边中的最小权值；按据约定，如果不存在这样的边，则 $key[v]=\infty$。域$\pi[v]$指树中 v 的“父母”。在算法的执行过程中，GENERIC-MST 的集合 A 隐含地满足： 568～571

$$A=\{(v,\pi[v]):v\in V-\{r\}-Q\}$$

当算法终止时，最小优先级队列 Q 是空的，而 G 的最小生成树 A 则满足：

$$A=\{(v,\pi[v]):v\in V-\{r\}\}$$

```
MST-PRIM(G, w, r)
 1  for each u ∈ V[G]
 2       do key[u] ← ∞
 3          π[u] ← NIL
 4  key[r] ← 0
 5  Q ← V[G]
 6  while Q ≠ ∅
 7       do u ← EXTRACT-MIN(Q)
 8          for each v ∈ Adj[u]
 9              do if v ∈ Q and w(u, v) < key[v]
10                    then π[v] ← u
11                         key[v] ← w(u, v)
```

Prim 算法的工作流程如图 23-5 所示。第 1～5 行置每个顶点的 *key* 域为∞(根顶点 r 除外，r 的 *key* 域被置为 0，这样它就会成为第一个被处理的顶点)，将每个顶点的父母设置为 NIL，并初始化最小优先级队列 Q，使之包含所有的顶点。该算法还要维护以下的包含了三个部分的循环不变式：

在第 6～11 行中 while 循环的每一次迭代之前，有：

1)$A=\{(v,\pi[v]):v\in V-\{r\}-Q\}$；

2)已经被放入最小生成树中的结点都是 $V-Q$ 中的顶点；

3)对所有的结点 $v\in Q$ 来说，如果 $\pi[v]\neq$NIL，则 $key[v]<\infty$，且 $key[v]$是一条轻边$(v,\pi[v])$的权值，该边连接了 v 与已在最小生成树中的某个顶点。

第 7 行找出与通过割$(V-Q, Q)$的一条轻边相关联的顶点$u \in Q$(第一次迭代例外，根据第 4
572 行，这时有$u=r$)。从集合 Q 中去掉 u 后，把它加入到树的结点集合 $V-Q$ 中，从而也就将$(u, \pi[u])$加入到了 A 中。第 8～11 行中的 for 循环对与 u 邻接、且不在树中的每个顶点 v 的 *key* 域和 π 域进行更新。这一更新可以使循环不变式的第三部分保持成立。

Prim 算法的性能取决于优先队列 Q 是如何实现的。如果用二叉最小堆来实现 Q(见第 6 章)，则可以用过程 BUILD-MIN-HEAP 来实现第 1～5 行的初始化部分，其运行时间为 $O(V)$。while 循环的循环体需要执行 $|V|$ 次，且由于每次 EXTRACT-MIN 操作需要 $O(\lg V)$时间，所以对 EXTRACT-MIN 的全部调用所占用的时间为 $O(V \lg V)$。第 8～11 行中的 for 循环总共要执行 $O(E)$次，这是因为所有邻接表的长度和为 $2|E|$。在 for 循环内部，第 9 行对队列 Q 的成员条件进行测试可以在常数时间内完成，这是由于为每个顶点留出 1 位(bit)的空间来记录该结点是否在队列 Q 中，在顶点移出队列时，对该位进行更新。第 11 行的赋值语句隐含了一个对最小堆进行的 DECREASE-KEY 操作，该操作在二叉最小堆上可以用 $O(\lg V)$时间完成。因此，Prim 算法的整个运行时间为 $O(V \lg V + E \lg V) = O(E \lg V)$。从渐近意义上来说，它和 Kruskal 算法的运行时间是相同的。

通过使用斐波那契堆，Prim 算法的渐近运行时间可得到进一步改善。在第 20 章，已经说明如果 $|V|$ 个元素组织成斐波那契堆，可以在 $O(\lg V)$的平摊时间内完成 EXTRACT-MIN 操作，在 $O(1)$的平摊时间里完成 DECREASE-KEY 操作(为实现第 11 行的代码)。因此，如果用斐波那契堆来实现最小优先队列 Q，Prim 算法的运行时间可以改进为 $O(E+V \lg V)$。

练习

23.2-1　根据对边进行排序时处理的不同方式，即使对同一输入图 G，Kruskal 算法也可能得出不同的生成树。证明对 G 的每一棵最小生成树 T，Kruskal 算法中都存在一种方法来对边进行排序，使得算法返回的最小生成树为 T。

23.2-2　假定图 $G=(V, E)$用邻接矩阵表示，在这种条件下，给出 Prim 算法的运行时间为 $O(V^2)$的实现。

23.2-3　从渐近意义上来说，对于稀疏图 $G=(V, E)$，$|E|=\Theta(V)$，用斐波那契堆来实现
573 Prim 算法是否要比用二叉堆来实现其运行速度要更快些？对于稠密图($|E|=\Theta(V^2)$)的情形又如何呢？从渐近意义来看，$|E|$和$|V|$有怎样的关系才会使得用斐波那契堆比用二叉堆来实现其算法执行起来较快？

23.2-4　假设在某个图中，所有边的权值均为 1 到 $|V|$ 之间的整数。在这一条件下，你能使 Kruskal 算法的运行时间达到多快？如果各边的权值都是 1 到 W(W 是某个常数)之间的整数，情况又怎样呢？

23.2-5　假设在某个图中，所有边的权值都是 1 到 $|V|$ 之间的整数，在这一条件下，你能使 Prim 算法的运行时间达到多快？如果各边的权值都是 1 到 W(W 是某个常数)之间的整数，情况又怎样呢？

***23.2-6**　假设在某个图中，所有边的权值都一致分布在半开区间$[1, 0)$之间。对于 Kruskal 和 Prim 这两个算法，你可以让哪一个运行得更快些？

***23.2-7**　假设某个图 G 有一棵已经计算出来的最小生成树。如果一个新的顶点及其关联的边被加入到了 G 中，该最小生成树可以在多快的时间内被更新呢？

23.2-8　Toole 教授提出了一种新的分治算法来计算最小生成树，该算法是这样的：给定一个图 $G=(V, E)$，将顶点集合 V 划分成两个集合 V_1 和 V_2，使得$|V_1|$和$|V_2|$至多差 1。

设 E_1 为一个边集，其中的边都与 V_1 中的顶点关联，E_2 为另一个边集，其中的边都与 V_2 中的顶点关联。在两个子图 $G_1=(V_1, E_1)$和 $G_2=(V_2, E_2)$上，分别递归地解决最小生成树问题。最后，从 E 中选择一条通过割(V_1, V_2)的最小权边，并利用该边，将所得的两棵最小生成树合并成一棵完整的生成树。

请论证该算法能正确地计算出 G 的最小生成树，或者给出一个使该算法不能正确工作的例子。

574

思考题

23-1　次最优的最小生成树

设 $G=(V, E)$是一个无向连通图，在其上定义了权值函数 w：$E\to\mathbf{R}$，并假设 $|E|\geqslant|V|$，且所有边的权值都是不同的。

所谓次最优的最小生成树是这样定义的：设 T 为 G 的所有生成树的集合，并设 T'为 G 的一棵最小生成树。那么，次最优的最小生成树就是这样的一棵最小生成树 T，它满足 $w(T)=\min_{T\in T-\{T'\}}\{w(T')\}$。

a)证明最小生成树是唯一的，但次最优最小生成树未必一定是唯一的。

b)设 T 是 G 的一棵最小生成树，证明存在边$(u, v)\in T$ 和$(x, y)\notin T$，使得 $T-\{(u, v)\}\cup\{(x, y)\}$是 G 的一棵次最优最小生成树。

c)设 T 是 G 的一棵生成树，且对任意两个顶点 u、$v\in V$，设 $\max[u, v]$是 T 中 u 和 v 之间唯一通路上的具有最大权值的边。请给出一个运行时间为 $O(V^2)$的算法，在给定 T 和所有顶点 u、$v\in V$ 以后，它可以计算出 $\max[u, v]$。

d)写出一个有效的算法来计算 G 的次最优最小生成树。

23-2　稀疏图的最小生成树

对于一个非常稀疏的连通图 $G=(V, E)$，可以对 G 进行“预处理”，以便在运行 Prim 算法前减少结点的数目，这样，对使用了斐波那契堆的 Prim 算法，就能对其原有的运行时间 $O(E+V\lg V)$做进一步的改进。特别地，对于每个结点 u，我们都选择与 u 关联的最小权边(u, v)，并将(u, v)添加到正在构造的最小生成树中。接着，收缩所有被选中的边。(见 B.4 节)。在收缩这些边时，不是一次收缩一条，而是首先找出被组合成同一个新结点的那组结点。接着，生成一次一条地收缩这些边而得到的图，根据这些边的端点所在的集合“重命名”这些边来实现。原图中的若干条边可能会被重命名成相同的名称。在这种情况下，仅会保留其中的一条边，且其权值是原来那些边中最小的。

开始时，将待构造的最小生成树 T 置为空，并且，对于每一条边$(u, v)\in E$，都置 $orig[u, v]=(u, v)$和 $c[u, v]=w(u, v)$。我们利用 $orig$ 属性来引用原图中的一条边，它与收缩后的图中的某一边对应。c 属性中存放的是一条边的权值，并且，随着各条边不断地被收缩，该属性也根据上述选择边的权值的模式来更新。过程 MST-REDUCE 的输入为 G、$orig$、c 和 T，返回的是一个经过收缩的图 G'，以及更新过的、针对图 G'的属性 $orig'$ 和 c'。该过程还逐渐将 G 中的边添加到最小生成树 T 中。

575

```
MST-REDUCE(G, orig, c, T)
1  for each v∈V[G]
2       do mark[v] ← FALSE
3          MAKE-SET(v)
4  for each u ∈ V[G]
```

```
 5      do if mark[u]=FALSE
 6           then choose v∈Adj[u]such that c[u, v]is minimized
 7                UNION(u, v)
 8                T ← T∪{orig[u, v]}
 9                mark[u] ← mark[v] ← TRUE
10  V[G'] ← {FIND-SET(v) : v∈V[G]}
11  E[G'] ← ∅
12  for each(x, y)∈E[G]
13      do u ← FIND-SET(x)
14         v ← FIND·SET(y)
15         if(u, v)∉E[G']
16           then E[G'] ← E[G']∪{(u, v)}
17                orig'[u, v] ← orig[x, y]
18                c'[u, v] ← c[x, y]
19           else if c[x, y]<c'[u, v]
20                  then orig'[u, v] ← orig[x, y]
21                       c'[u, v] ← c[x, y]
22  construct adjacency lists Adj for G'
23  return G', orig', c', and T
```

a)设 T 是 MST-REDUCE 所返回的边集，A 是图 G' 的最小生成树，它是通过调用 MST-PRIM(G', c', r)而形成的，其中 r 是 $V[G']$中的任意顶点。证明：$T\cup\{orig'[x, y]: (x, y)\in A\}$是 G 的一棵最小生成树。

b)论证：$|V[G']|\leqslant|V|/2$

c)说明如何实现 MST-REDUCE，以使其运行时间为 $O(E)$。(提示：使用简单的数据结构。)

d)假设我们运行了 MST-REDUCE 的 k 个阶段，其中每一个阶段产生的输出 G'、$orig'$ 和 c' 都用作下一个阶段的输入 G、$orig$ 和 c，并且逐渐将各条边添加到 T 中。论证 k 个阶段的总体运行时间为 $O(kE)$。

576

e)假设在运行 MST-REDUCE 的 k 个阶段后，如在上面 d)小题中一样，通过调用 MST-PRIM(G', c', r)来运行 Prim 算法，其中 G' 和 c' 是由最后一个阶段返回，而 r 是 $V[G']$中的任意顶点。试说明应如何选择 k，才能使得总体运行时间为 $O(E\lg\lg V)$。证明你所选择的 k 能使总体的渐近运行时间为最小。

f)从渐近意义上来看，$|E|$ 为何值(用 $|V|$ 表示)时，可以使带预处理功能的 Prim 算法优于不带预处理的 Prim 算法？

23-3　瓶颈生成树

无向图 G 的一棵瓶颈生成树(bottleneck spanning tree) T 是这样的一棵生成树，它最大的边权值在 G 的所有生成树中是最小的。瓶颈生成树的值为 T 中最大权值边的权。

a)论证：最小生成树也是瓶颈生成树。

这说明寻找一棵瓶颈生成树并不难于寻找一棵最小生成树。在下面的两个小题中，将说明可以在线性时间内找到这样的一棵生成树。

b)给出一个线性时间的算法，它在给定一个图 G 和一个整数 b 的情况下，确定瓶颈生成树的值是否最大不超过 b。

c)利用你在 b)小题中给出的算法作为一个线性时间的、用于求解瓶颈生成树问题的算

法的子例程。(提示：可以利用一个子例程来收缩各条边，如在思考题 23-2 中描述的 MST-REDUCE 过程中那样。)

23-4　其他的最小生成树算法

在本问题中，我们给出三个算法的伪代码。每一个算法都取一个图作为输入，并返回一个边集 T。对于这三个算法中的每一个，读者要证明 T 是一棵最小生成树，或者证明 T 不是一棵最小生成树。此外，对于每个算法，无论它是否能计算出一棵最小生成树，都要给出其最有效的实现。

577

a. MAYBE-MST-A(G, w)

```
1  sort the edges into nonincreasing order of edge weights w
2  T ← E
3  for each edge e, taken in nonincreasing order by weight
4       do if T−{e} is a connected graph
5            then T ← T−e
6  return T
```

b. MAYBE-MST-B(G, w)

```
1  T ← ∅
2  for each edge e, taken in arbitrary order
3       do if T∪{e} has no cycles
4            then T ← T∪e
5  return T
```

c. MAYBE-MST-C(G, w)

```
1  T ← ∅
2  for each edge e, taken in arbitrary order
3     do T ← T∪{e}
4          if T has a cycle c
5             then let e' be the maximum-weight edge on c
6                  T ← T−{e'}
7  return T
```

本章注记

Tarjan[292]对最小生成树问题进行了综述，并提供了非常好的材料。Graham 和 Hell[131]描述了最小生成树问题的历史。

根据 Tarjan，第一篇有关最小生成树算法的论文是在 1926 年由 O. Borůvka 发表的。Borůvka 算法包含了对思考题 23-2 中 MST-REDUCE 过程的 $O(\lg V)$ 次迭代。Kruskal 算法是由 Kruskal[195]于 1956 年公开发表的。被大家称为 Prim 算法的算法的确是由 Prim[250]提出的，但 V. Jarník 在更早一些的时候(1930 年)就提出这一算法了。

578

在寻找最小生成树这一问题中，贪心算法之所以会比较有效，是因为一个图的森林集合形成了一个图形拟阵(见 16.4 节)。

当 $|E| = \Omega(V \lg V)$ 时，用斐波那契堆实现的 Prim 算法的运行时间为 $O(E)$。对于更为稀疏的图来说，Fredman 和 Tarjan[98]综合利用了 Prim 算法、Kruskal 算法和 Borůvka 算法的思想，并采用了高级数据结构，给出了一个运行时间为 $O(E \lg^* V)$ 的算法。Gabow、Galil、Spencer 和

Tarjan[102]对这一算法进行了改进，使其运行时间达到了 $O(E\lg\lg^* V)$。Chazelle[53]给出了一个运行时间为 $O(E\hat{\alpha}(E, V))$ 的算法，其中 $\hat{\alpha}(E, V)$ 是 Ackermann 函数的逆函数。（有关 Ackermann 函数及其逆函数的简单讨论，可见第 21 章的“本章注记”。）Chazelle 算法与先前的最小生成树算法有所不同，它并没有采用贪心方法。

与最小生成树相关的一个问题是生成树的验证(spanning tree verification)问题，即给定一个图 $G=(V, E)$和一棵树 $T\subseteq E$，我们希望确定 T 是否是 G 的一棵最小生成树。King[177]给出了一个线性时间的算法来解决生成树验证问题，该算法是建立在更早一些的 Komlós[188]和 Dixon, Rauch 和 Tarjan[77]等工作基础之上的。

以上提到的算法都是确定性的算法，都可以归入基于比较的算法模型(见第 8 章)。Karger, Klein 和 Tarjan[169]给出了一个随机化的最小生成树算法，其期望运行时间为 $O(V+E)$。该算法利用了递归技术，且递归的方式与 9.3 节介绍的线性时间选择算法类似：通过对一个辅助问题进行递归调用，来找出边集的一个子集 E'，它不可能出现在任何最小生成树中。另外还要对 $E-E'$进行一次递归调用，用以找出最小生成树。在生成树验证方面，该算法还借鉴了 Borůvka 算法和 King 算法的思想。

Fredman 和 Willard[100]说明了应如何通过一个确定性的、非基于比较的算法，在 $O(V+E)$ 时间内找出一棵最小生成树。他们提出的算法假定数据是 b 位的整数，且计算机内存由可寻址的
579 b 位字所组成。

第 24 章　单源最短路径

假设有一个开车的人希望找出从芝加哥到波士顿之间可能的最短路线。他有一张美国公路图，该公路图上标出了每一对相邻的公路交叉点之间的距离，他应该如何找出这一最短路线呢？

一种可能的方法就是枚举出所有从芝加哥到波士顿之间的路线，并对每条路线的长度求和，然后选择最短的一条。很容易看出，即使不考虑包含回路的路线，依然存在着数以百万计的行车路线，而其中绝大多数是不值得考虑的。例如，从芝加哥途经休斯敦到波士顿就是一个糟糕的选择，因为休斯敦大约有一千英里远。

在本章和第 25 章中，我们将阐述如何有效地解决这类问题。在最短路径问题中，给出的是一个带权有向图 $G=(V, E)$，加权函数 w：$E \to \mathbf{R}$ 为从边到实型权值的映射。路径 $p=\langle v_0, v_1, \cdots, v_k\rangle$ 的权是指其组成边的所有权值之和：

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

定义从 u 到 v 间的最短路径的权为：

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{如果存在着一条从 } u \text{ 到 } v \text{ 的路径} \\ \infty & \text{否则} \end{cases}$$

从顶点 u 到顶点 v 的最短路径定义为权 $w(p)=\delta(u, v)$ 的任何路径。

在芝加哥到波士顿的例子中，我们可以把公路图模型化为一个图：图的顶点表示公路交叉点，边表示公路交叉点之间的路段，边的权表示公路长度。我们的目标是从芝加哥的一个给定的公路交叉点（如克拉克街区（Clark St.）和爱迪逊大街（Addison Ave.））到波士顿的一个给定的公路交叉点（如布鲁克林大街（Brookline Ave.）和约克路）。

边的权值还可以被解释为其他的某种度量标准，而不一定是距离。它常常被用来表示时间、费用、罚款、损失或者任何其他沿一条路径线性积累的和我们试图将其最小化的某个量。

第 22.2 节中介绍的广度优先搜索算法就是一种在无权图上执行的最短路径算法，即在图的边都具有单位权值的图上的一种算法。因为有关广度优先搜索的许多概念都产生于对有权图中最短路径的研究，因此，读者可以在继续本章学习之前，先复习 22.2 节。

单源最短路径的变体

在本章中，我们将重点讨论单源最短路径问题：已知图 $G=(V, E)$，我们希望找出从某给定源顶点 $s \in V$ 到每个顶点 $v \in V$ 的最短路径。很多其他问题都可用单源问题的算法来解决，其中包括下列变体：

单终点最短路径问题：找出从每个顶点 v 到指定终点 t 的最短路径。把图中的每条边反向，就可以把这一问题变为单源最短路径问题。

单对顶点最短路径问题：对于某给定顶点 u 和 v，找出从 u 和 v 的一条最短路径。如果我们解决了源点为 u 的单源问题，则这一问题也就获得解决。即使在最坏的情况下，从渐近意义上来看，目前还没有比最好的单源算法更快的算法来解决这一问题。

每对顶点间最短路径问题：对于每对顶点 u 和 v，找出从 u 到 v 的最短路径。虽然将每个顶点作为源点，运行一次单源算法就可以解决这一问题，但通常可以更快地解决这一问题，它自己的结构本身就是重要的。下一章将详细讨论这个问题。

最短路径的最优子结构

最短路径算法通常依赖于一种性质，也就是一条两顶点间的最短路径包含路径上其他的最短路径。(第 26 章 Edmonds-Karp 的最大流算法也依赖于这一性质。)这种最优子结构性质是动态规划(第 15 章)和贪心算法(第 16 章)方法是否适用的一个标记。24.3 节的 Dijkstra 算法是一个贪心算法，而找出所有顶点对之间的最短路径的 Floyd-Warshall 算法(见第 25 章)是一个动态规划
581 算法。下面的引理更加准确地陈述最短路径的最优子结构性质。

引理 24.1(最短路径的子路径是最短路径)　对于一给定的带权有向图 $G=(V, E)$，所定义的权函数为 $w: E \to \mathbf{R}$。设 $p=\langle v_1, v_2, \cdots, v_k\rangle$是从 v_1 到 v_k 的最短路径。对于任意 i，j，其中 $1 \leqslant i \leqslant j \leqslant k$，设 $p_{ij}=\langle v_i, v_{i+1}, \cdots, v_j\rangle$为 p 中从顶点 v_i 到顶点 v_j 的子路径。那么，p_{ij} 是从 v_i 到 v_j 的最短路径。

证明：如果将路径 p 分解为 $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$，则有 $w(p)=w(p_{1i})+w(p_{ij})+w(p_{jk})$。假设存在一条从 v_i 到 v_j 的路径 p'_{ij}，其权值 $w(p'_{ij})<w(p_{ij})$。那么，$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ 是从 v_1 到 v_k 的一条路径，它的权值 $w(p_{1i})+w(p'_{ij})+w(p_{jk})$小于 $w(p)$，这与 p 是 v_1 至 v_k 的最短路径相矛盾。■

负权值边

在单源最短路径问题的某些实例中，可能存在着权值为负值的边。如果图 $G=(V, E)$不包含从源 s 可达的负权回路，则对所有 $v \in V$，最短路径的权的定义 $\delta(u, v)$依然正确，即使它是一个负值也是如此。但是，如果存在一条从 s 可达的负权回路，那么最短路径的权的定义就不能成立了。从 s 到该回路上的顶点之间就不存在最短路径，因为我们总是可以顺着已找出的"最短"路径，再穿过负权值回路而获得一条权值更小的路径。因此，如果从 s 到 v 的某路径中存在一条负权回路，就定义 $\delta(u, v)=-\infty$。

图 24-1 说明了负的权值以及负权回路对最短路径权的影响。因为从 s 到 a 只有一条路径(路径 $\langle s, a\rangle$)，$\delta(s, a)=w(s, a)=3$。类似地，从 s 到 b 也只有一条路径，所以 $\delta(s, b)=w(s, a)+w(a, b)=3+(-4)=-1$。从 s 到 c 有无限多路径：$\langle s, c\rangle$，$\langle s, c, d, c\rangle$，$\langle s, c, d, c, d, c\rangle$等等。因为回路$\langle c, d, c\rangle$的权为 $6+(-3)=3>0$，所以从 s 到 c 的最短路径为$\langle s, c\rangle$，其权为 $\delta(s, c)=5$。类似地，从 s 到 d 的最短路径为$\langle s, c, d\rangle$，其权为 $\delta(s, d)=w(s, c)+w(c, d)=11$。同样，从 s 到 e 存在无数条路径：$\langle s, e\rangle$，$\langle s, e, f, e\rangle$，$\langle s, e, f, e, f, e\rangle$等等。由于回路$\langle e, f, e\rangle$的权为 $3+(-6)=-3<0$，所以从 s 到 e 没有最短路径。只要穿越负权回路$\langle e, f, e\rangle$任意次，我们就可以发现从 s 到 e 的路径可以有任意小的负权值。所以 $\delta(s, e)=-\infty$。类似地，$\delta(s, f)=-\infty$。因为 g 是从 f 可达的，所以我们可以找到任意小的负权值的从 s 到 g 的路径，则 $\delta(s, g)=-\infty$。顶点 h，i 和 j 也形成了一个负权回路，但因为它们从 s 不可达，因此 $\delta(s, h)=\delta(s, i)=\delta(s, j)=\infty$。

一些最短路径的算法，例如 Dijkstra 算法，假定输入图中的所有边的权值都是非负的，如公
582 路地图的例子。另一些算法，如 Bellman-Ford 算法，允许输入图中存在负权边，只要不存在从源点可达的负权回路，它能给出正确的解答。特别地，如果存在负权回路，算法还可以检测并报告出这种回路的存在。

回路

一条最短路径能包含回路吗？正如刚才所见到的，它不能包含负权回路。它也不会包含正权回路，因为从路径上移去回路后，可以产生一个具有相同源点和终点、权值更小的路径。也就是说，如果 $p=\langle v_0, v_1, \cdots, v_k\rangle$是一条路径，而 $c=\langle v_i, v_{i+1}, \cdots, v_j\rangle$是这条路径上的一个正权

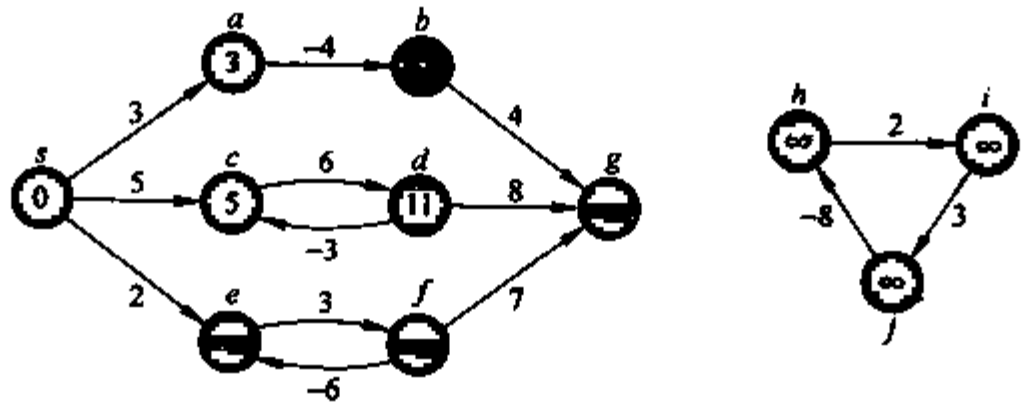


图 24.1　有向图的负权边。每个顶点内部显示的是从源点 s 到本顶点的最短路径的权值。因为顶点 e 和 f 形成了一个从 s 可达的负权环，所以最短路径的权为 $-\infty$。因为顶点 g 是一个具有 $-\infty$ 权的最短路径顶点可达的，所以最短路径的权也为 $-\infty$。像 h，i 和 j 顶点都不是从 s 可达的，所以它们的最短路径的权为 ∞，即使它们在一个负权回路上

回路(那么 $v_i=v_j$，且 $w(c)>0$)。因而，路径 $p'=\langle v_0, v_1, \cdots, v_i, v_{j+1}, v_{j+2}, \cdots, v_k\rangle$ 的权 $w(p')=w(p)-w(c)<w(p)$，p 不可能是从 v_0 到 v_k 的最短路径。

现在只剩下 0 权回路了。我们可以从任意一条路径上移去 0 权回路，产生具有相同权值的另一条路径。因此，如果从源点 s 到终点 v 存在着一条包含 0 权回路的最短路径，那么就一定还存在着一条从 s 到 v 的无回路最短路径。只要最短路径包含 0 权回路，我们可以不断地将这些回路从路径中移除，直到产生出一个无环的最短路径为止。所以，不失一般性，可以假设找到的最短路径为无环的。因为图 $G=(V, E)$ 中的任意一条无环路径包含至多 $|V|$ 个不同的顶点，同时包含至多 $|V|-1$ 条边。因此，我们可以把注意力集中在至多 $|V|-1$ 条边的最短路径上。 583

最短路径的表示

我们通常不仅希望算出最短路径的权，而且也希望得到最短路径上的顶点。我们所采用的最短路径的表示方法类似于第 22.2 节中广度优先树的表示方法。已知图 $G=(V, E)$，对每个顶点 $v\in V$，设置其前趋(predecessor)顶点 $\pi[v]$为另一顶点或 NIL。本章中的最短路径算法设置 π 属性，以便使源于顶点 v 的前辈链表沿着从 s 到 v 的最短路径的相反方向排列。因此，对于一给定的 $\pi[v]\neq$NIL 的顶点 v，可以运用第 22.2 节中的过程 PRINT-PATH(G, s, v)输出从 s 到 v 的一条最短路径。

不过，在最短路径算法的执行过程中，无需用 π 的值来指明最短路径。正如广度优先搜索一样，我们感兴趣的是由 π 的值导出的前趋子图 $G_\pi=(V_\pi, E_\pi)$。这里，我们定义顶点集 V_π 为 G 中所有具有非空前趋的顶点集合，再加上源点 s。

$$V_\pi=\{v\in V:\pi[v]\neq \text{NIL}\}\cup\{s\}$$

有向边集 E_π 是由 V_π 中的顶点的 π 值导出的边集：

$$E_\pi=\{(\pi[v],v)\in E:v\in V_\pi-\{s\}\}。$$

我们将证明本章中算法得出的 π 值有如下性质：在算法结束时，G_π 就是“最短路径树”，非形式地说是一棵有根树，它包含了从源点 s 到 s 可达的每个顶点之间的一条最短路径。最短路径树和第 22.2 节讨论的广度优先树相似，但是它所包含的从源点出发的最短路径是用边的权(而不是用边的数目)来表示的。更准确地说，设图 $G=(V, E)$ 是带权有向图，其加权函数为 $w: E\to \mathbf{R}$，并假定 G 中不包含从源点 $s\in V$ 可达的权值为负的回路，那么最短路径是良定义的。以 s 为根的最短路径树是有向子图 $G'=(V', E')$，其中 $V'\subseteq V$，$E'\subseteq E$，那么：

1) V' 是 G 中从 s 可达的顶点集合

2)G'形成了一棵以s为根的有根树

3)对所有$v\in V'$，G'中从s到v的唯一简单路径是G中从s到v的最短路径。

584 最短路径并不一定是唯一的，最短路径树亦是如此。例如，图24-2示出了一个加权有向图，以及具有相同树根的两棵最短路径树。

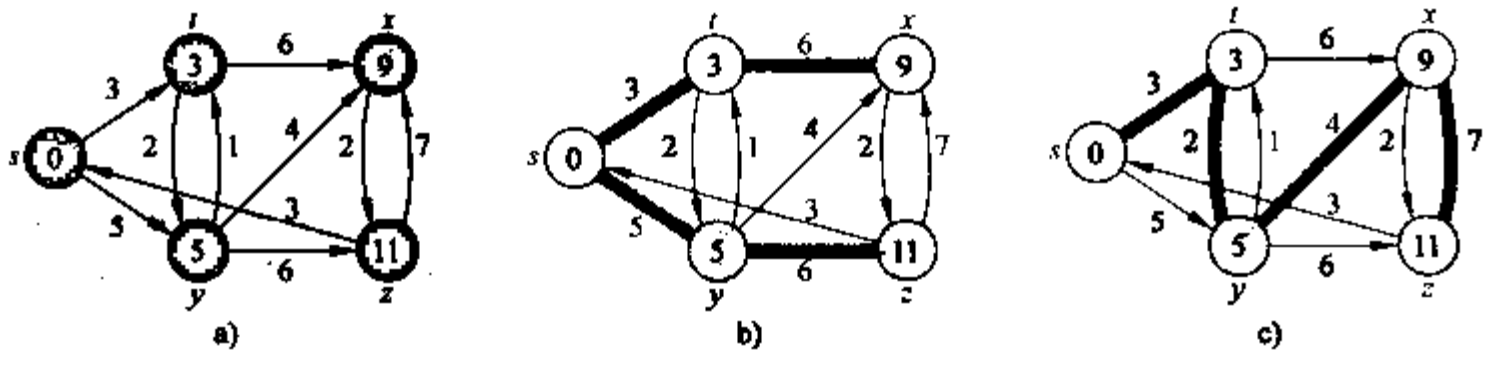


图24-2　a)标记了以s为源点的最短路径权值的加权有向图。b)阴影覆盖的边形成了一棵以源点s为根的最短路径树。c)具有相同根的另一棵最短路径树

松弛技术

本章的算法使用了松弛(relaxation)技术。对每个顶点$v\in V$，都设置一个属性$d[v]$，用来描述从源点s到v的最短路径上权值的上界，称为最短路径估计(shortest-path estimate)。我们用下面的$\Theta(V)$时间的过程来对最短路径估计和前趋进行初始化。

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v∈V[G]
2        do d[v] ← ∞
3           π[v] ← NIL
4  d[s] ← 0
```

经过初始化以后，对所有$v\in V$，$\pi[v]=$NIL，对$v\in V-\{s\}$，有$d[s]=0$以及$d[v]=\infty$。

在松弛[Θ]一条边(u, v)的过程中，要测试是否可以通过u，对迄今找到的到v的最短路径进行改进；如果可以改进的话，则更新$d[v]$和$\pi[v]$。一次松弛操作可以减小最短路径估计的值
585 $d[v]$，并更新v的前趋域$\pi[v]$。下面的伪代码对边(u, v)进行了一步松弛操作。

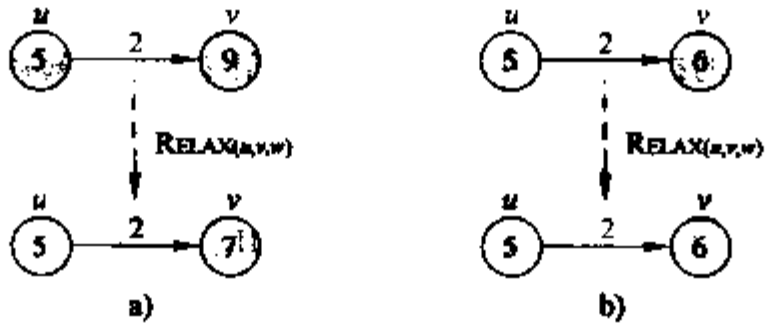


图24-3　对权$w(u, v)=2$的边(u, v)进行松弛。每个顶点的最短路径估计被标在顶点内。a)因为在松弛前$d[v]>d[u]+w(u, v)$，所以$d[v]$的值减小。b)这里，在松弛之前$d[v]\leqslant d[u]+w(u, v)$，那么松弛不会改变$d[v]$

Θ 用"松弛"这个名词来表示紧缩上界的操作看似有些奇怪，这个名词的使用是有来历的。一步松弛操作的结果可以看作是对约束$d[v]\leqslant d[u]+w(u, v)$的松弛，其中如果$d[u]=\delta(s, u)$以及$d[v]=\delta(s, v)$，三角不等式(定理24.10)必须满足。也就是说，如果$d[v]\leqslant d[u]+w(u, v)$，满足此约束并没有"压力"，所以约束是"松弛的"。

```
RELAX(u, v, w)
1  if d[v]>d[u]+w(u, v)
2     then d[v] ← d[u]+w(u, v)
3          π[v] ← u
```

图 24-3 示出了松弛一条边的两个实例，在其中一个例子中，最短路径估计值减小，而在另一个实例中，最短路径估计值不变。

本章中的每个算法都会调用 INITIALIZE-SINGLE-SOURCE，然后重复对边进行松弛的过程。另外，松弛是改变最短路径和前趋的唯一方式。本章中的算法之间的区别在于对每条边进行松弛操作的次数，以及对边执行松弛操作的次序有所不同。在 Dijkstra 算法以及关于有向无回路图的最短路径算法中，对每条边执行一次松弛操作。在 Bellman-Ford 算法中，对每条边要执行多次松弛操作。

最短路径以及松弛的性质

要证明本章算法的正确性，需要求助于最短路径以及松弛的一些性质。在这里先陈述一下这些性质，24.5 节里将会对它们进行形式化的证明。为了便于参考，这里陈述的每个性质都包含了 24.5 节里的引理或推论的标号。最后的五个性质是关于最短路径估计和前趋子图的，它们隐含地假设了图是调用 INITIALIZE-SINGLE-SOURCE(G, s)进行初始化的，而且最短路径估计和前趋子图唯一的变化途径就是一系列的松弛步骤。 586

三角不等式(引理 24.10)

对任意边$(u, v)\in E$，有 $\delta(s, v)\leqslant\delta(s, u)+w(u, v)$

上界性质(引理 24.11)

对任意顶点 $v\in V$，有 $d[v]\geqslant\delta(s, v)$，而且一旦 $d[v]$达到 $\delta(s, v)$值就不再改变。

无路径性质(推论 24.12)

如果从 s 到 v 不存在路径，则总是有 $d[v]=\delta(s, v)=\infty$。

收敛性质(引理 24.14)

如果 $s\rightsquigarrow u\rightarrow v$ 是图 G 某 $u, v\in V$ 的最短路径，而且在松弛边(u, v)之前的任何时间 $d[u]=\delta(s, u)$，则在操作过后总有 $d[v]=\delta(s, v)$。

路径松弛性质(引理 24.15)

如果 $p=\langle v_0, v_1, \cdots, v_k\rangle$是从 $s=v_0$ 到 v_k 的最短路径，而且 p 的边按照$(v_0, v_1)(v_1, v_2), \cdots,(v_{k-1}, v_k)$的顺序进行松弛，那么 $d[v_k]=\delta(s, v_k)$。这个性质的保持并不受其他松弛操作的影响，即使它们与 p 的边上的松弛操作混合在一起也是一样的。

前趋子图性质(引理 24.17)

一旦对于所有 $v\in V$，$d[v]=\delta(s, v)$，前趋子图就是一个以 s 为根的最短路径树。

本章概述

第 24.1 节提出了 Bellman-Ford 算法，该算法用来解决一般(边的权值可以为负)的单源最短路径问题。Bellman-Ford 算法非常简单，可以检测是否有从源点可达的负权回路。第 24.2 节中给出了在一个有向无环图中，在线性时间内计算出单源最短路径的算法。第 24.3 节给出了 Dijkstra 算法，它的运行时间比 Bellman-Ford 算法低，但要求所有边的权值为非负。第 24.4 节说明了如何使用 Bellman-Ford 算法来解决“动态规划”的一个特例。最后，第 24.5 节证明了上面所陈述的最短路径和松弛的性质。

下面给出一些关于无穷运算的约定。假设对任何实数 $a\neq-\infty$，有 $a+\infty=\infty+a=\infty$。同时，为了保证我们的证明在出现负权回路情况下的正确性，假定对任何实数 $a\neq\infty$，有 $a+(-\infty)=(-\infty)+a=-\infty$。

本章所有的算法都假设有向图 G 用邻接表的形式存储，而且每条边上还存储了它的权值。因此，当遍历每一个邻接表时，可以对每条边在 $O(1)$ 时间内确定其权值。

587

24.1　Bellman-Ford 算法

Bellman-Ford 算法能在一般的情况(存在负权边的情况)下，解决单源最短路径问题。对于给定的带权有向图 $G=(V, E)$，其源点为 s，加权函数为 w：$E \to \mathbf{R}$，对该图运行 Bellman-Ford 算法后可以返回一个布尔值，表明图中是否存在着一个从源点可达的权为负的回路。若存在这样的回路的话，算法说明该问题无解；若不存在这样的回路，算法将产生最短路径及其权值。

此算法运用松弛技术，对每个顶点 $v \in V$，逐步减小从源 s 到 v 的最短路径的权的估计值 $d[v]$直至其达到实际最短路径的权 $\delta(s, v)$。算法返回布尔值 TRUE，当且仅当图中不包含从源点可达的负权回路。

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| −1
3       do for each edge (u, v) ∈ E[G]
4            do RELAX(u, v, w)
5  for each edge(u, v) ∈ E[G]
6       do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

图 24-4 说明了 Bellman-Ford 算法在 5 个顶点的图上的执行过程。在第 1 行初始化每个顶点的 d 和 π 值后，算法对图中的边进行了 $|V|-1$ 遍操作。每一遍都是第 2～4 行 for 循环的一次迭代。图 24-4b-e 示出了此算法对边进行四遍操作，每一遍过后的状态。在 $|V|-1$ 遍操作过后，第 5～8 行对负权回路进行检查，并返回适当的布尔值(稍后我们将会看到为什么这里的检查是有效的)。

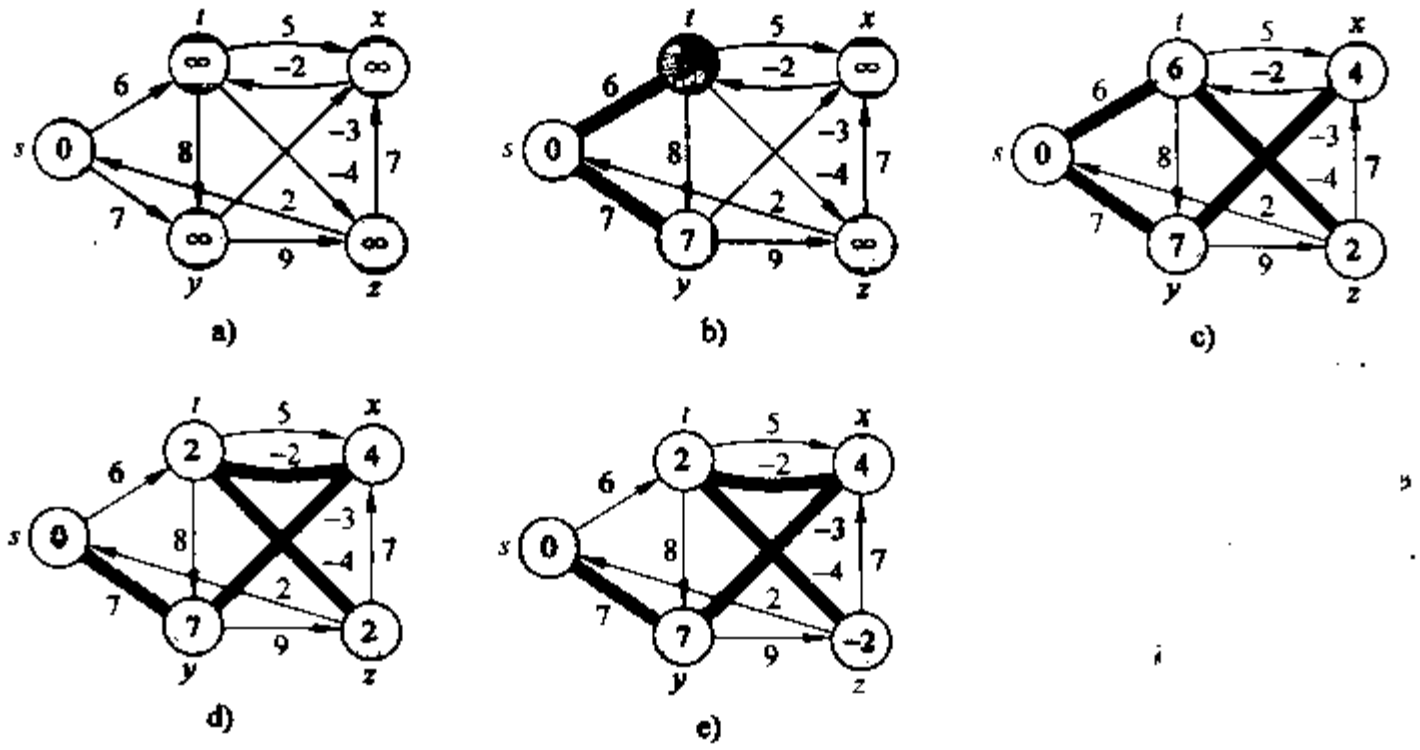


图 24-4　Bellman-Ford 算法的执行过程。源点是顶点 s。d 值被标记在顶点内，阴影覆盖的边指示了前趋值：如果边 (u, v) 被阴影覆盖，则 $\pi[v]=u$。在这个特定的例子中，每一趟按照如下的顺序对边进行松弛：(t, x)，(t, y)，(t, z)，(x, t)，(y, x)，(y, z)，(z, x)，(z, s)，(s, t)，(s, y)。a)示出了对边进行第一趟操作前的情况。b)至 e)示出了每一趟连续对边操作后的情况。e)中 d 和 π 值是最终结果。Bellman-Ford 算法在这个例子中返回的是 TRUE

Bellman-Ford 算法的运行时间为 $O(VE)$，因为第 1 行的初始化占用时间为 $\Theta(V)$，第 2～4 行对边进行 $|V|-1$ 趟操作，每趟操作的运行时间为 $\Theta(E)$。第 5～7 行的 for 循环运行时间为 $O(E)$。

为了证明 Bellman-Ford 算法的正确性，开始时讨论没有负权回路的情况，算法可以正确地计算出从源点出发到所有顶点的最小路径的权。

引理 24.2　设 $G=(V, E)$为带权有向图，其源点为 s，权函数为 $w: E \to \mathbf{R}$. 并且假定 G 中不包含从 s 点可达的负权回路。那么，在 BELLMAN-FORD 第 2～4 行 for 循环的 $|V|-1$ 次迭代后，对任何 s 可达的顶点 v，有 $d[v]=\delta(s, v)$。

证明：证明此引理将使用路径松弛性质。设 v 为从 s 可达的任意顶点，$p=\langle v_0, v_1, \cdots, v_k\rangle$ 是任一条从 s 到 v 的无环最短路径，其中 $v_0=s$ 和 $v_k=v$。路径 p 有至多 $|V|-1$ 条边，所以 $k \leqslant |V|-1$。第 2～4 行的 for 循环的 $|V|-1$ 次迭代，其每一次都松弛了所有 E 边。第 i 次迭代被松弛的边为(v_{i-1}, v_i)，其中 $i=1, 2, \cdots, k$。由路径松弛性质，有 $d[v]=d[v_k]=\delta(s, v_k)=\delta(s, v)$。 ■

588 ~ 589

推论 24.3　设 $G=(V, E)$为带权有向图，源顶点为 s，加权函数为 $w: E \to \mathbf{R}$。对每一顶点 $v \in V$，从 s 到 v 存在一条通路，当且仅当对 G 运行 Bellman-Ford 算法，算法终止时，有 $d[v]<\infty$。

证明：证明留待练习 24.1-2。 ■

定理 24.4(Bellman-Ford 算法的正确性)　设 $G=(V, E)$为带权有向图，源点为 s，权函数为 $w: E \to \mathbf{R}$，对该图运行 Bellman-Ford 算法。若 G 不包含 s 可达的负权回路，则算法返回 TRUE，对所有顶点 $v \in V$，有 $d[v]=\delta(s, v)$成立。前趋子图 G_π 是以 s 为根的最短路径树。如果 G 包含从 s 可达的负权回路，则算法返回 FALSE。

证明：假设图 G 不包含从 s 可达的负权回路，首先证明这样一个结论：在算法终止时，对所有顶点 $v \in V$，有 $d[v]=\delta(s, v)$。如果顶点 v 是从 s 可达的，则可由引理 24.2 证明。如果 v 从 s 不可达，则可由无路径性质得到上述结论。这样就证明了该结论。前趋子图的性质以及这个结论隐含着 G_π 是一棵最短路径树。现在，利用这个结论来说明 BELLMAN-FORD 返回的是 TRUE。在终止时，对所有边$(u, v) \in E$，有

$$\begin{aligned} d[v] &= \delta(s,v) \\ &\leqslant \delta(s,u)+w(u,v) \quad \text{（根据三角不等式）} \\ &= d[u]+w(u,v) \end{aligned}$$

这样，第 6 行的测试不会使得 BELLMAN-FORD 返回 FALSE。则算法必然返回 TRUE。

相反地，假设图 G 包含一个从 s 可达的负权回路；设此回路为 $c=(v_0, v_1, \cdots, v_k)$，其中 $v_0=v_k$。那么，

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \tag{24.1}$$

为了引出矛盾，假设 Bellman-Ford 算法返回 TRUE。因此，对 $i=1, 2, \cdots, k$，有 $d[v_i] \leqslant d[v_{i-1}]+w(v_{i-1}, v_i)$。把回路 c 中所有这样的不等式相加，有：

590

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &\leqslant \sum_{i=1}^{k} (d[v_{i-1}]+w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

由于 $v_0=v_k$，c 中的每个顶点在 $\sum_{i=1}^{k} d[v_i]$ 和 $\sum_{i=1}^{k} d[v_{i-1}]$ 每个和式中都仅出现一次，所以

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

另外，由推论 24.3 可知，对 $i=1, 2, \cdots, k$，$d[v_i]$是有限的。那么，

$$0 \leqslant \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

这就与不等式(24.1)相矛盾。因此，如果图 G 中不包含从源点可达且权值为负的回路，Bellman-Ford 算法返回 TRUE，否则返回 FALSE。 ■

练习

24.1-1 以顶点 z 作为源点，对图 24-4 所给出的有向图运行 Bellman-Ford 算法。每趟操作中，按照图中的相同顺序对边进行松弛，并示出每趟过后 d 与 π 的值。现在，将边(z, x)的权值变为 4，再以 s 为源点运行此算法。

24.1-2 证明推论 24.3。

24.1-3 对于给定的无负权回路的带权有向图 $G=(V, E)$，设在所有 $u, v \in V$ 的顶点对中，m 为所有从 u 到为 v 的最短路径上边数最小值中的最大值(这里，最短路径是根据权值来说的，而不是边的数目)。可以对 Bellman-Ford 算法做简单的修改，则可在 $m+1$ 趟后终止。

24.1-4 对 Bellman-Ford 算法进行修改，对任意顶点 v，当从源点到 v 的某些路径上存在一个负
591 权回路，则置 $d[v]$为$-\infty$。

*24.1-5 设 $G=(V, E)$为一带权有向图，其权函数 $w: E \to \mathbf{R}$。请给出一个 $O(VE)$时间的算法，对每个顶点 $v \in V$，找出值 $\delta^*(v)=\min_{u \in V}\{\delta(u, v)\}$。

*24.1-6 假定一加权有向图 $G=(V, E)$包含一负权回路。请给出一个能够列出此回路上的顶点的高效算法。并证明你算法的正确性。

24.2　有向无回路图中的单源最短路径

按顶点的拓扑序列对某加权 dag 图(有向无回路图)$G=(V, E)$的边进行松弛后，就可以在 $\Theta(V+E)$时间内计算出单源最短路径。在一个 dag 图中最短路径总是存在的，因为即使图中有权值为负的边，也不可能存在负权回路。

算法开始对 dag 图进行拓扑排序(见 22.4 节)，以便获得顶点的线性序列。如果从顶点 u 到顶点 v 存在一条路径，则在拓扑序列中 u 先于 v。在拓扑排序的过程中我们仅对顶点执行了一趟操作。当对每个顶点处理时，松弛从该点出发的所有边。

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       do for each vertex v ∈ Adj[u]
5            do RELAX(u, v, w)
```

图 24-5 给出了此算法的执行过程。

此算法的运行时间很容易分析。如 22.4 节所示，第 1 行的拓扑排序需 $\Theta(V+E)$时间。第 2 行调用的 INITIALIZE-SINGLE-SOURCE 需 $\Theta(V)$时间。第 3～5 行的 for 循环中对每一个顶点有一次迭代。对每一个顶点，其每条出边都只被检测过一次。第 4～5 行的 for 循环内部总共有 $|E|$ 次迭代(我们在这里使用了聚集分析)。因为 for 循环内部的每次迭代需 $\Theta(1)$时间，总的运行时间为
592 $\Theta(V+E)$，这与图的邻接表表示的大小成线性关系。

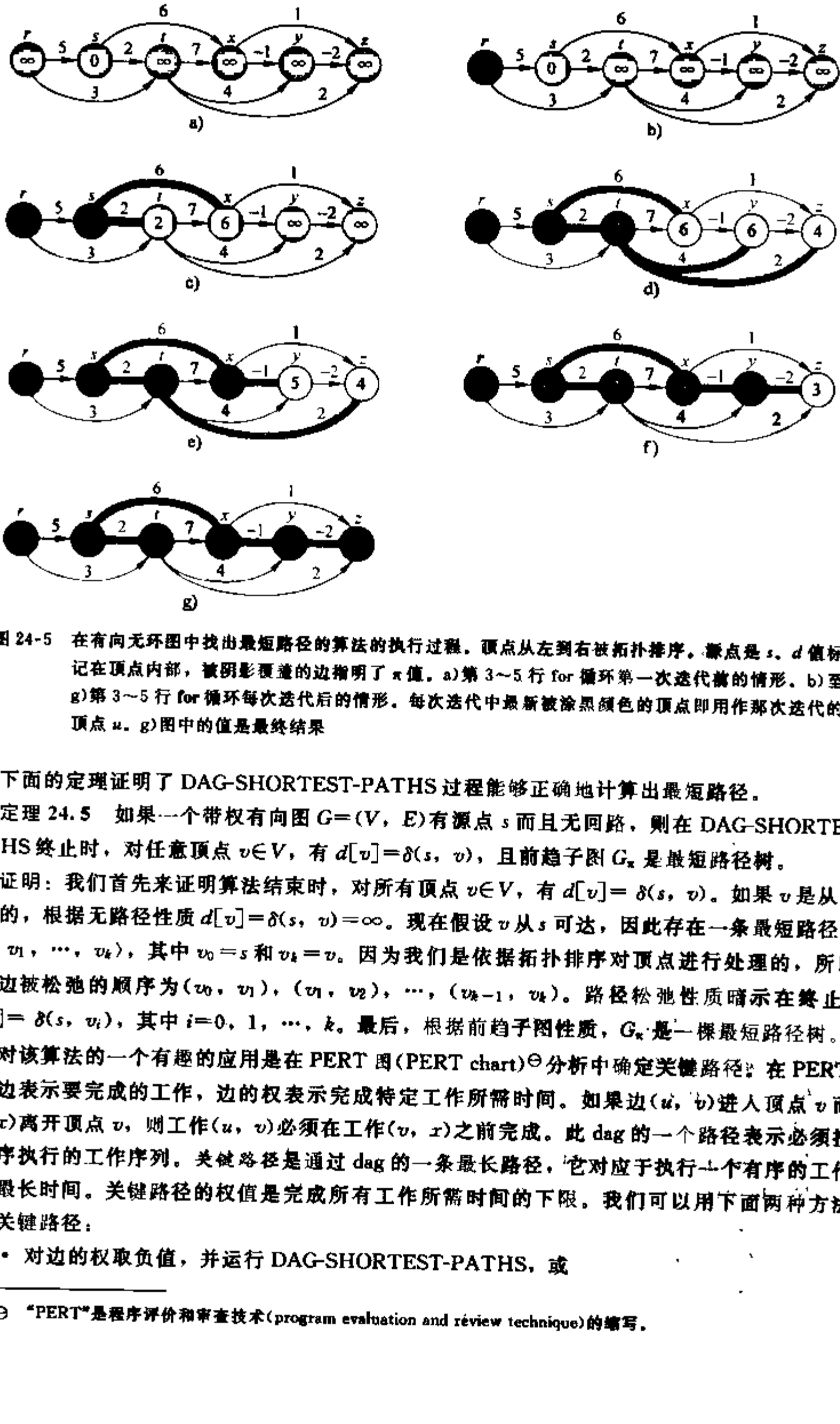


图 24-5　在有向无环图中找出最短路径的算法的执行过程。顶点从左到右被拓扑排序。源点是 s。d 值标记在顶点内部，被阴影覆盖的边指明了 π 值。a)第 3～5 行 for 循环第一次迭代前的情形。b)至 g)第 3～5 行 for 循环每次迭代后的情形。每次迭代中最新被涂黑颜色的顶点即用作那次迭代的顶点 u。g)图中的值是最终结果

下面的定理证明了 DAG-SHORTEST-PATHS 过程能够正确地计算出最短路径。

定理 24.5　如果一个带权有向图 $G=(V, E)$ 有源点 s 而且无回路，则在 DAG-SHORTEST-PATHS 终止时，对任意顶点 $v \in V$，有 $d[v]=\delta(s, v)$，且前趋子图 G_π 是最短路径树。 593

证明：我们首先来证明算法结束时，对所有顶点 $v \in V$，有 $d[v]=\delta(s, v)$。如果 v 是从 s 不可达的，根据无路径性质 $d[v]=\delta(s, v)=\infty$。现在假设 v 从 s 可达，因此存在一条最短路径 $p=\langle v_0, v_1, \cdots, v_k\rangle$，其中 $v_0=s$ 和 $v_k=v$。因为我们是依据拓扑排序对顶点进行处理的，所以 p 上的边被松弛的顺序为 (v_0, v_1)，(v_1, v_2)，…，(v_{k-1}, v_k)。路径松弛性质暗示在终止时，$d[v_i]=\delta(s, v_i)$，其中 $i=0, 1, \cdots, k$。最后，根据前趋子图性质，G_π 是一棵最短路径树。■

对该算法的一个有趣的应用是在 PERT 图(PERT chart)[⊖]分析中确定关键路径。在 PERT 图中，边表示要完成的工作，边的权表示完成特定工作所需时间。如果边 (u, v) 进入顶点 v 而边 (v, x) 离开顶点 v，则工作 (u, v) 必须在工作 (v, x) 之前完成。此 dag 的一个路径表示必须按一定顺序执行的工作序列。关键路径是通过 dag 的一条最长路径，它对应于执行一个有序的工作序列的最长时间。关键路径的权值是完成所有工作所需时间的下限。我们可以用下面两种方法来找出关键路径：

- 对边的权取负值，并运行 DAG-SHORTEST-PATHS，或

⊖ "PERT"是程序评价和审查技术(program evaluation and review technique)的缩写。

- 将 INITIALIZE-SINGLE-SOURCE 的第 2 行“∞”换为“$-\infty$”，并且将 RELAX 过程中的“>”换成“<”，然后运行 DAG-SHORTEST-PATHS。

练习

24.2-1　以顶点 r 作为源点，对图 24-5 中的有向图运行 DAG-SHORTEST-PATHS。

24.2-2　假设我们把 DAG-SHORTEST-PATHS 的第 3 行修改成：

```
3  for the first | V | −1 vertices, taken in topologically sorted order
```

说明此过程仍然正确。

[594] **24.2-3**　上面对 PERT 图的阐述不太自然。如果用顶点代表工作，边代表有序的约束，即边(u, v)表示工作 u 必须在工作 v 之前完成，权值则赋给顶点而不是边。修改过程 DAG-SHORTEST-PATHS，使其能在线性时间内，计算出具有带权顶点的有向无回路图中的最长路径。

24.2-4　给出一个高效算法来统计有向无回路图中的全部路径数。分析所给出的算法。

24.3　Dijkstra 算法

Dijkstra 算法解决了有向图 $G=(V, E)$上带权的单源最短路径问题，但要求所有边的权值非负。因此在本节中，我们假定对每条边$(u, v)\in E$，有 $w(u, v)\geqslant 0$。正如我们所看到的，一个实现得很好的 Dijkstra 算法比 Bellman-Ford 算法的运行时间要低。

Dijkstra 算法中设置了一顶点集合 S，从源点 s 到集合中的顶点的最终最短路径的权值均已确定。算法反复选择具有最短路径估计的顶点 $u\in V-S$，并将 u 加入 S 中，对 u 的所有出边进行松弛。在下面的算法实现中，用到了顶点的最小优先队列 Q，排序关键字为顶点的 d 值。

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q≠∅
5       do u ← EXTRACT-MIN(Q)
6          S ← S∪{u}
7          for each vertex v∈Adj[u]
8               do RELAX(u, v, w)
```

Dijkstra 算法对边的松弛如图 24-6 所示。第 1 行执行通常的 d 和 π 值的初始化。第 2 行将集合 S 初始化为空集。在第 4～8 行 while 循环的每次迭代开始时，算法都保持着循环不变式 $Q=V-S$。第 3 行初始化最小优先队列 Q，使其包含 V 中所有顶点；因为此时 $S=\varnothing$，第 3 行过后循环不变式依然保持。在第 4～8 行中 while 循环的每一轮迭代里，将一顶点 u 从 $Q=V-S$ 中删除，并加入到集合 S 中，因此不变式仍然保持成立。（第一次循环时 $u=s$。）因此，在 $V-S$ 的所有顶

[595] 点中，顶点 u 具有最小的最短路径估计。然后，第 7～8 行对以 u 为起点的每条边(u, v)进行松弛。如果可以经过 u 来改进到顶点 v 的最短路径的话，就对估计值 $d[v]$以及前趋 $\pi(v)$进行更新。注意在第 3 行以后，就不会再插入顶点到 Q 中了，并且每个顶点只能从 Q 中删除并插入 S 一次，因此第 4～8 行中 while 循环的迭代次数为 $|V|$ 次。

因为 Dijkstra 算法总是在 $V-S$ 中选择“最轻”或“最近”的顶点插入集合 S 中，所以我们说它使用了贪心策略。贪心策略已在第 16 章中作了详细阐述；在这儿，读者无需阅读第 16 章也可以

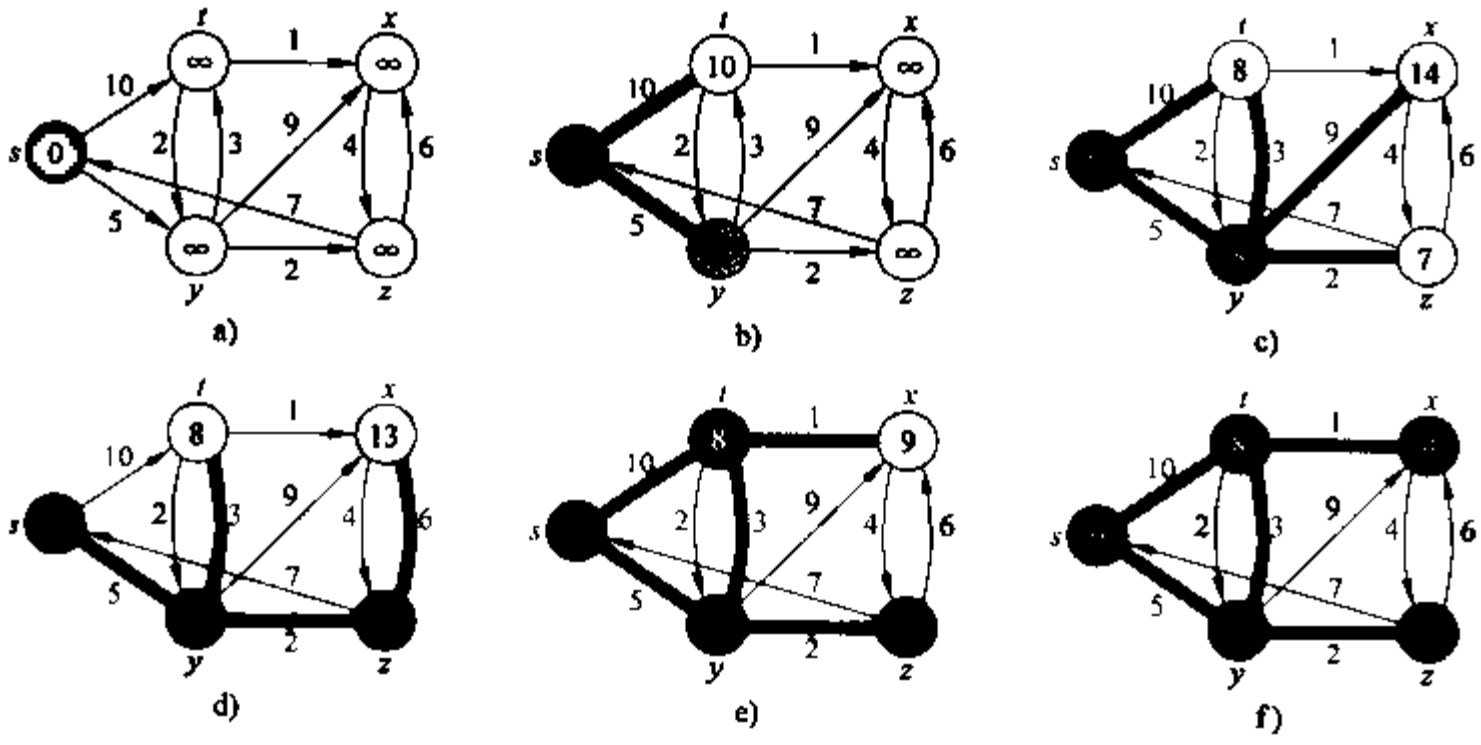


图 24-6　Dijkstra 算法的执行过程。源点 s 为最左端顶点。最短路径估计被标记在顶点内，阴影覆盖的边指出了前趋的值。黑色顶点在集合 S 中，而白色顶点在最小优先队列 $Q=V-S$ 中。a)第 4～8 行 while 循环第一次迭代前的情形。阴影覆盖的顶点具有最小的 d 值，而且在第 5 行被选为顶点 u。b)至 f)while 循环在每一次连续迭代后的情形。每个图中阴影覆盖的顶点被选作下一次迭代第 5 行的顶点 u。f)图中的 d 和 π 值是最终结果

读懂 Dijkstra 算法。贪心策略并非总是能获得全局意义上的最理想结果，但正如下列定理和推论所述的那样，Dijkstra 算法确实计算出了最短路径。关键是要证明每当一个顶点 u 被插入集合 S 中时，有 $d[u]=\delta(s, u)$。

596

定理 24.6(Dijkstra 算法的正确性)　已知一带权有向图 $G=(V, E)$，其加权函数 w 的值为非负，源点为 s。对该图运行 Dijkstra 算法，则在算法终止时，对所有 $u\in V$ 有 $d[u]=\delta(s, u)$。

证明：我们使用如下循环不变式：

在第 4～8 行 while 循环每次迭代开始时，对每个顶点 $v\in S$ 有 $d[v]=\delta(s, v)$。

这足够说明对每个顶点 $u\in V$，当 u 加入集合 S 时 $d[u]=\delta(s, u)$。一旦有 $d[u]=\delta(s, u)$，我们利用上界性质来说明，此后的所有时间里等式仍然成立。

初始化：初始时，$S=\varnothing$，因此循环不变式显然成立。

保持：我们希望说明在每一轮迭代中，对加入到集合 S 中的顶点 u，都有 $d[u]=\delta(s, u)$。利用矛盾思想，设 u 是加入集合 S 中的第一个满足 $d[u]\neq\delta(s, u)$的顶点。我们将集中注意 while 循环的迭代开始时的情形。u 被加入集合 S，通过检查从 s 到 u 的最短路径，推导出此时 $d[u]=\delta(s, u)$，从而与假设矛盾。u 一定不等于 s，因为 s 是第一个被加入集合 S 的顶点，而且 $d[s]=\delta(s, s)=0$。因为$u\neq s$,我们有在 u 加入 S 之前 $S\neq\varnothing$。从 s 到 u 一定存在某条路径，否则根据无路径性质 $d[u]=\delta(s, u)=\infty$，这违背 $d[u]\neq\delta(s, u)$的假设。因为至少存在一条路径，那么从 s 到 u 存在一条最短路径 p。在 u 加入 S 之前，路径 p 连接着 S 中的一个顶点 s 和 $V-S$ 中的一个顶点 u。设顶点 y 为 p 路径上第一个属于$V-S$ 的顶点，而且设 $x\in S$ 为 y 的前趋。因此，如图 24-7 所示，路径 p 可以被分解为 $s \overset{p_1}{\rightsquigarrow} x\rightarrow y \overset{p_2}{\rightsquigarrow} u$(路径 p_1 和 p_2 都可能没有边)。

当 u 加入 S 时，我们要求 $d[y]=\delta(s, y)$。为了证明这一点，考察 $x\in S$。因为 u 是第一个被加入集合 S 时 $d[u]\neq\delta(s, u)$的顶点，则当 x 加入集合 S 时有 $d[x]=\delta(s, x)$。边(x, y)在那时被松弛过，因此此要求是符合收敛性质的。

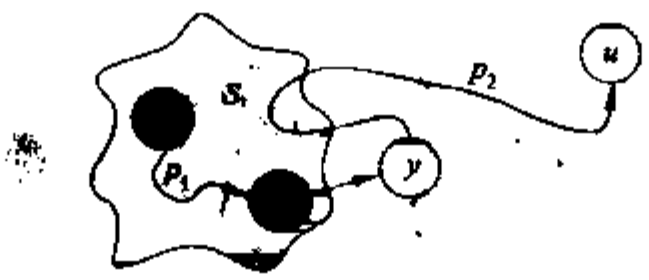


图 24-7　定理 24.6 的证明。在顶点 u 加入之前集合 S 非空。从源点 s 到顶点 u 的最短路径 p 可被分解为 $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$，其中 y 是路径上第一个不在 S 中的顶点，$x \in S$ 是 y 的紧邻前趋。顶点 x 和 y 是不同的，但是可能会有 $s=x$ 或 $y=u$。路径 p_2 不能确定是否会再加入集合 S

现在，我们就能通过推出矛盾来证明 $d[u]=\delta(s, u)$。因为在从 s 到 u 的最短路径上，y 在 u 之前出现，而且所有边的权值都为非负值(注意在 p_2 上也是如此)，因而有 $\delta(s, y) \leqslant \delta(s, u)$，所以

$$d[y] = \delta(s,y) \leqslant \delta(s,u) \leqslant d[u] \quad \text{(根据上界性质)} \tag{24.2}$$

但是，当 u 在第 5 行被选择时，顶点 u 和 y 都在 $V-S$ 中，有 $d[u] \leqslant d[y]$。因此，式(24.2)中的两个不等式实际上是等式，有

$$d[y] = \delta(s,y) = \delta(s,u) = d[u]$$

结果 $d[u]=\delta(s, u)$，这与当初 u 的选择相矛盾。所以，当 u 被加入集合 S 时 $d[u]=\delta(s, u)$，而且在以后的任何时刻，此等式成立。

终止：在终止时，$Q=\varnothing$，联系先前的不变式 $Q=V-S$，说明 $S=V$。因此，对所有顶点 $u \in V$，$d[u]=\delta(s, u)$。■

推论 24.7　已知一加权函数非负且源点为 s 的带权有向图 $G=(V, E)$，若在该图上运行 Dijkstra 算法，则在算法终止时，前趋子图 G_π 是以 s 为根的最短路径树。

证明：由定理 24.6 和前趋子图性质立即得证。■

分析

Dijkstra 算法的执行速度如何呢？它通过调用三种优先队列操作维护最小优先队列 Q：INSERT(第 3 行)，EXTRACT-MIN(第 5 行)和 DECREASE-KEY(在第 8 行调用的 RELAX 中)。同 EXTRACT-MIN 类似，INSERT 也是每顶点调用一次。因为每个顶点 $v \in V$ 被加入集合 S 中仅有一次，所以在算法的过程中，邻接表 $Adj[v]$ 的每条边在第 7～8 行算法中的 for 循环中仅检查一次。由于所有邻接表中的边的总数为 $|E|$，因而 for 循环中有共计 $|E|$ 次迭代，共有至多 $|E|$ 次 DECREASE-KEY 操作。(我们使用聚集分析的方法对其进行再一次的考察。)

597 ~ 598

Dijkstra 算法的运行时间依赖于最小优先队列的具体实现。首先考虑第一种情况，利用从 1 至 $|V|$ 编好号的顶点，简单地将 $d[v]$ 存入一个数组的第 v 项。每一个 INSERT 和 DECREASE-KEY 操作都是 $O(1)$ 时间，而每一个 EXTRACT-MIN 操作为 $O(V)$ 时间(因为需要搜索整个数组)，总计的运行时间为 $O(V^2+E)=O(V^2)$。

特别地，如果是稀疏图的情况，有 $E=o(V^2/\lg V)$。在这种情况下，利用二叉最小堆来实现最小优先队列是很有用的。(正如 6.5 节中所讨论的，一个重要的实现细节就是顶点和对应的堆元素必须维护对方的柄。)每一个 EXTRACT-MIN 操作需要 $O(\lg V)$ 的时间。如前所述，有 $|V|$ 个这样的操作。构建二叉最小堆的时间为 $O(V)$。每一个 DECREASE-KEY 操作需要 $O(\lg V)$ 的时间，而且最多有 $|E|$ 次这种操作。因此总计的运行时间为 $O((V+E)\lg V)$，如果所有顶点都从源点可达的话，则为 $O(E \lg V)$。如果 $E=o(V^2/\lg V)$，这个运行时间是对 $O(V^2)$ 时间的简单实

现的一个改进。

事实上，用斐波那契堆(见第 20 章)来实现优先队列，可以将运行时间提升到 $O(V\lg V+E)$。$|V|$ 个 EXTRACT-MIN 操作的每个平摊代价为 $O(\lg V)$，而且至多 $|E|$ 个 DECREASE-KEY 操作的每个平摊时间为 $O(1)$。从历史的角度看，在 Dijkstra 算法中，DECREASE-KEY 的调用比 EXTRACT-MIN 的调用一般要多得多，所以任何能够在不增加 EXTRACT-MIN 操作的平摊时间的同时，减小每个 DECREASE-KEY 操作的平摊时间到 $o(\lg V)$的方法，从渐近意义上来说，都能获得比二叉堆更快的实现。也正是因为这一点，才推动了斐波那契堆的发展。

Dijkstra 算法和广度优先搜索算法(见 22.2 节)以及计算最小生成树的 Prim 算法(见 23.2 节)都有类似之处。它和广度优先算法的相似性在于，前者的集合 S 相当于后者的黑色顶点集合；正如集合 S 中的顶点有着最终的最短路径的权值，广度优先搜索中的黑色顶点也有着正确的广度优先距离。Dijkstra 算法与 Prim 算法的相似之处在于，两种算法均采用最小优先队列，来找出给定集合(Dijkstra 算法中的集合 S 以及 Prim 算法中的生长树)以外“最轻”的顶点，然后把该点加入到集合中，并相应地调整该集合以外剩余顶点的权。 599

练习

24.3-1 用顶点 s 和顶点 z 分别作为源点，对图 24-2 所示的有向图运行 Dijkstra 算法。利用图 24-6 所示的方式，说明在 while 循环的每次迭代以后的 d 和 π 值，以及集合 S 中的顶点情况。

24.3-2 给出一含有负权边的有向图的简单实例，说明 Dijkstra 算法对其运行时会产生错误的结果。在允许图中边的权值为负时，为什么定理 24.6 的证明不能成立？

24.3-3 假设将 Dijkstra 算法的第 4 行改为

```
4  while |Q| > 1
```

这一修改使得 while 循环执行 $|V|-1$ 次而不是 $|V|$ 次，是否正确？

24.3-4 已知一有向图 $G=(V, E)$，其每条边$(u, v)\in E$ 均对应有一个实数值 $r(u, v)$，表示从顶点 u 到顶点 v 之间的通信线路的可靠性，取值范围为 $0\leqslant r(u, v)\leqslant 1$。定义 $r(u, v)$为从 u 到 v 的线路不中断的概率，并假定这些概率是相互独立的。写出一个有效算法，来找出两个指定顶点间的最可靠的线路。

24.3-5 设 $G=(V, E)$为一个带权有向图，其权函数 $w: E\to\{1, 2, \cdots, W\}$，$W$ 是某正整数，并假设从源点 s 到任意两个顶点之间都不存在相同的最短路径权值。现假设我们定义一个无权有向图 $G'=(V\cup V', E')$，将每条边$(u, v)\in E$ 依次替换为 $w(u, v)$个单位权边。那么，G'中有多少个顶点？假设在 G'上运行广度优先搜索，试说明在对 G'进行广度优先搜索的过程中，V 中顶点被标记成黑色的顺序与 DIJKSTRA 算法运行于 G 上时，算法的第 5 行从优先队列中删除 V 中顶点的顺序相同。

24.3-6 设 $G=(V, E)$为带权有向图，权函数 $w: E\to\{0, 1, \cdots, W\}$，其中 W 为某非负整数。修改 Dijkstra 算法，以使其计算从指定源点 s 的最短路径所需的运行时间为 $O(WV+E)$。 600

24.3-7 修改练习 24.3-6 中的算法，使其运行时间为 $O((V+E)\lg W)$。(提示：在任意时刻，$V-S$中有多少不同的最短路径估计?)

24.3-8 假定有一个带权有向图 $G=(V, E)$，从源点 s 出发的边可能有负权，所有其他的边的权都非负，而且不存在负权回路。论证在这样的图中，Dijkstra 算法可以正确地从 s 找到最短路径。

24.4　差分约束与最短路径

第 29 章研究一般的线性规划问题，在那些问题中，要对由一组线性不等式定义的线性函数进行优化。在本节中，我们将考察可以简化为寻找单源最短路径的线性规划的一种特殊情形。由此引出的单源最短路径问题可以运用 Bellman-Ford 算法来解决，进而也解决了原线性规划问题。

线性规划

在一般的线性规划问题(linear-programming problem)中，给定一个 $m\times n$ 的矩阵 A，一个 m 维向量 b 和一个 n 维向量 c，我们希望找出由 n 个元素组成的向量 x，在由 $Ax\leqslant b$ 所给出的 m 个约束条件下，使目标函数 $\sum_{i=1}^{n} c_i x_i$ 最大。

单纯形法(the simplex algorithm)是第 29 章的重点，它并不总是能在输入规模的多项式时间内运行；但是，还有其他一些线性规划算法是可以以多项式时间运行的。理解线性规划问题的构造非常重要，原因有这样几点：首先，当知道可以将某一给定问题转换成一个多项式规模的线性规划问题时，也就意味着这个问题存在着一个多项式时间的算法。其次，线性规划中有许多特例存在更快速的算法。例如，本节所述的单源最短路径问题就是一个线性规划的特例。其他可以被构造成线性规划的包括单对最短路径问题(练习 24.4-4)以及最大流问题(练习 26.1-8)。

有时，我们并不关心目标函数，仅仅是希望找出一个可行解，即一个满足 $Ax\leqslant b$ 的向量 x，
601 或是确定不存在可行解。下面我们来关注这样的一个可行性问题。

差分约束系统

在一个差分约束系统(system of difference constraints)中，线性规划矩阵 A 的每一行包含一个 1 和一个 −1，A 的所有其他元素都为 0。因此，由 $Ax\leqslant b$ 给出的约束条件是 m 个差分约束集合，其中包含 n 个未知元。每个约束条件为如下形式的简单线性不等式：

$$x_j - x_i \leqslant b_k$$

其中 $1\leqslant i, j\leqslant n$，$1\leqslant k\leqslant m$。

例如，考虑这样一个问题，即寻找一个 5 维向量 $x=(x_i)$ 以满足：

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leqslant \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

这一问题等价于找出未知量 x_i，$i=1, 2, \cdots, 5$，满足下列 8 个差分约束条件：

$$x_1 - x_2 \leqslant 0 \tag{24.3}$$

$$x_1 - x_5 \leqslant -1 \tag{24.4}$$

$$x_2 - x_5 \leqslant 1 \tag{24.5}$$

$$x_3 - x_1 \leqslant 5 \tag{24.6}$$

$$x_4 - x_1 \leqslant 4 \tag{24.7}$$

$$x_4 - x_3 \leqslant -1 \tag{24.8}$$

$$x_5 - x_3 \leqslant -3 \tag{24.9}$$

$$x_5 - x_4 \leqslant -3 \tag{24.10}$$

该问题的一个解为 $x=(-5, -3, 0, -1, -4)$，这可以直接代入每个不等式而得到验证。实际上，该问题有多个解。另一个解为 $x'=(0, 2, 5, 4, 1)$，这两个解是有联系的：x' 中的每个元素比 x 中的相应元素大 5。这一事实并不仅仅是巧合。

引理 24.8　设 $x=(x_1, x_2, \cdots, x_n)$ 是一个差分约束系统 $Ax \leqslant b$ 的一个解，d 为任意常数。则 $x+d=(x_1+d, x_2+d, \cdots, x_n+d)$ 也是该系统 $Ax \leqslant b$ 的解。

602

证明：对于每个 x_i 和 x_j，有 $(x_j+d)-(x_i+d)=x_j-x_i$。因此，若 x 满足 $Ax \leqslant b$，则 $x+d$ 也同样满足。 ■

差分约束系统出现在很多不同的应用领域中。例如，未知量 x_i 可以是事件将要发生的时刻。每个约束条件可以解释为两件事情之间必须至少有一段时间，或者至多有一段时间。也许事件就是装配一件产品需要执行的工作。假设涂上一种粘合剂需 2 小时凝固，从 x_1 时间开始，必须等到 x_2 时间才能装配上一个零件。那么，就有约束 $x_2 \geqslant x_1+2$，或等价地，$x_1-x_2 \leqslant -2$。另一方面，我们可能需要在涂上粘合剂之后，但是在不比这个粘合剂凝固一半的时间还晚的时间安装零件。在这个情况下，我们得到一对约束 $x_2 \geqslant x_1$ 和 $x_2 \leqslant x_1+1$，或等价地，$x_1-x_2 \leqslant 0$ 和 $x_2-x_1 \leqslant 1$。

约束图

用图形理论观点来解释差分约束系统是很有益的。在一理想的差分约束系统 $Ax \leqslant b$ 中，$m \times n$ 的线性规划矩阵 A 可被看作是 n 顶点、m 条边的图的关联矩阵(见练习 22.1-7)的转置。对于 $i=1, 2, \cdots, n$，图中的每一个顶点 v_i 对应着 n 个未知量的一个 x_i。图中的每个有向边对应着关于两个未知量的 m 个不等式的其中一个。

更形式地，给定一个差分约束系统 $Ax \leqslant b$，相应的约束图是一个带权有向图 $G=(V, E)$，其中

$$V = \{v_0, v_1, \cdots, v_n\}$$

而且

$$E = \{(v_i, v_j) : x_j - x_i \leqslant b_k \text{ 是一个约束}\} \cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \cdots, (v_0, v_n)\}$$

引入附加顶点 v_0 是为了保证其他每个顶点均从 v_0 可达。因此，顶点集合 V 由对应于每个未知量 x_i 的顶点 v_i 和附加的顶点 v_0 所组成。边的集合 E 由对应于每个差分约束条件的边与对应于每个未知量 x_i 的边 (v_0, v_i) 所构成。如果 $x_j-x_i \leqslant b_k$ 是一个差分约束，则边 (v_i, v_j) 的权 $w(v_i, v_j)=b_k$。从 v_0 出发的每条边的权值均为 0。图 24-8 给出了式(24.3)～式(24.10)的差分约束系统的约束图。

下面的定理说明，可以通过在相应的约束图中找出最短路径的权的方法，来求得差分约束系统的解。

603

定理 24.9　给定一差分约束系统 $Ax \leqslant b$，设 $G=(V, E)$ 为其相应的约束图。如果 G 不包含负权回路，那么

$$x = (\delta(v_0, v_1), \ \delta(v_0, v_2), \ \delta(v_0, v_3), \ \cdots, \ \delta(v_0, v_n)) \tag{24.11}$$

是此系统的一可行解。如果 G 包含负权回路，那么此系统不存在可行解。

证明：首先来证明如果约束图中不包含负权回路，则由式(24.11)给出一可行解。考察任意边 $(v_i, v_j) \in E$，由三角不等式 $\delta(v_0, v_j) \leqslant \delta(v_0, v_i)+w(v_i, v_j)$ 或 $\delta(v_0, v_j)-\delta(v_0, v_i) \leqslant w(v_i, v_j)$。因此，设 $x_i=\delta(v_0, v_i)$，$x_j=\delta(v_0, v_j)$，满足对应边 (v_i, v_j) 的差分约束 $x_j-x_i \leqslant w(v_i, v_j)$。

接下来证明如果约束图中包含负权回路，则差分约束系统无可行解。不失一般性，设负权回

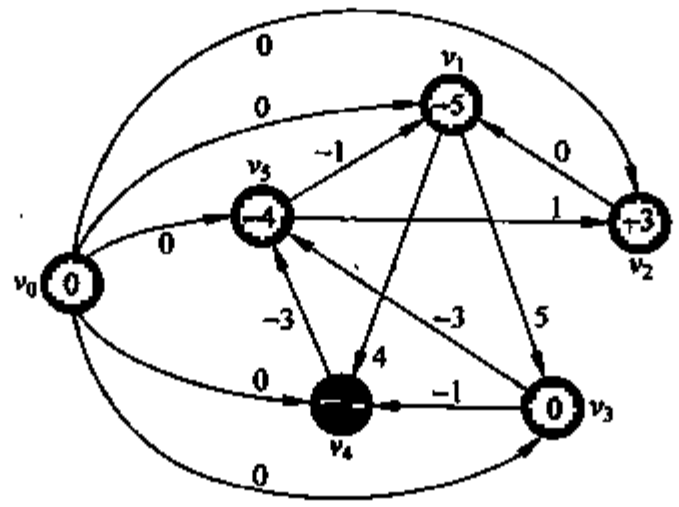


图 24-8　对应式(24.3)～式(24.10)的差分约束系统的约束图，每个顶点 v_i 中显示了 $\delta(v_0, v_i)$ 的值。该系统的一个可行解为 $x=(-5, -3, 0, -1, -4)$

路为 $c=\langle v_1, v_2, \cdots, v_k\rangle$，其中 $v_1=v_k$。(因为顶点 v_0 没有入边，所以它不可能在回路 c 上。)回路 c 对应着如下的差分约束：

$$\begin{aligned} x_2 - x_1 &\leqslant w(v_1, v_2) \\ x_3 - x_2 &\leqslant w(v_2, v_3) \\ &\vdots \\ x_k - x_{k-1} &\leqslant w(v_{k-1}, v_k) \\ x_1 - x_k &\leqslant w(v_k, v_1) \end{aligned}$$

604

假定存在满足 k 个不等式的一个解 x，那么也满足上述 k 个不等式相加所得的不等式。如果把这些不等式相加，就会发现每个顶点 x_i 的正负项相抵，结果左项的和为 0，而右项的和为 $w(c)$，因此有 $0 \leqslant w(c)$。但由于 c 是负权回路，$w(c)<0$，因此得到矛盾 $0 \leqslant w(c)<0$。(证毕) ■

差分约束系统问题的求解

定理 24.9 告诉我们，可以采用 Belllman-Ford 算法对差分约束系统求解。因为在约束图中，从源点 v_0 到所有其他顶点间均存在边，因此约束图中任何负权回路均从 v_0 可达。如果 Bellman-Ford 算法返回 TRUE，则最短路径权给出了此系统的一个可行解。例如在图 24-8 中，最短路径权给出了一个可行解 $x=(-5, -3, 0, -1, -4)$，而由引理 24.8，对任意常数 d，$x=(d-5, d-8, d, d-1, d-4)$ 也是系统的可行解。如果 Bellman-Ford 算法返回 FALSE，则差分约束系统无可行解。

关于 n 个未知量的 m 个约束条件的一个差分约束系统产生出一个具有 $n+1$ 个顶点和 $n+m$ 条边的图，因此采用 Bellman-Ford 算法，可以在 $O((n+1)(n+m))=O(n^2+nm)$ 时间内将系统解决。练习 24.4-5 要求对算法进行修改，以使其运行时间变为 $O(nm)$，即使 m 远小于 n。

练习

24.4-1　对下列差分约束系统找出其可行解，或者说明不存在可行解。

$$x_1 - x_2 \leqslant 1 \quad x_1 - x_4 \leqslant -4 \quad x_2 - x_3 \leqslant 2 \quad x_2 - x_5 \leqslant 7 \quad x_2 - x_6 \leqslant 5$$

605

$$x_3 - x_6 \leqslant 10 \quad x_4 - x_2 \leqslant 2 \quad x_5 - x_1 \leqslant -1 \quad x_5 - x_4 \leqslant 3 \quad x_6 - x_3 \leqslant -8$$

24.4-2　对下列差分约束系统找出其可行解，或说明不存在可行解。

$$x_1 - x_2 \leqslant 4 \quad x_1 - x_5 \leqslant 5 \quad x_2 - x_4 \leqslant -6 \quad x_3 - x_2 \leqslant 1 \quad x_4 - x_1 \leqslant 3$$

$$x_4 - x_3 \leqslant 5 \quad x_4 - x_5 \leqslant 10 \quad x_5 - x_3 \leqslant -4 \quad x_5 - x_4 \leqslant -8$$

24.4-3　在约束图中，从新顶点 v_0 出发的最短路径的权是否可以为正数？试说明之。

24.4-4　试用线性规划方法来表述单对顶点最短路径问题。

24.4-5　试说明如何对 Bellman-Ford 算法稍作修改，使其在解关于 n 个未知量的 m 个不等式所定义的差分约束系统时，运行时间为 $O(nm)$。

24.4-6　假定除一个差分约束系统外，我们还要求处理形如 $x_i = x_j + b_k$ 的相等约束。试说明 Bellman-Ford 算法如何作适当修改，以解决这个约束系统的变形。

24.4-7　试说明如何不用附加顶点 v_0 而对约束图运行类 Bellman-Ford 算法，从而求得差分约束系统的解。

*24.4-8　设 $Ax \leqslant b$ 是关于 n 个未知量的 m 个约束条件的差分约束系统。证明对其相应的约束图运行 Bellman-Ford 算法，可以求得满足 $Ax \leqslant b$，并且对所有的 x_i，有 $x_i \leqslant 0$，式 $\sum_{i=1}^{n} x_i$ 的最大值。

606

*24.4-9　证明 Bellman-Ford 算法在差分约束系统 $Ax \leqslant b$ 的约束图上运行时，使 $(\max\{x_i\} - \min\{x_i\})$ 取得满足 $Ax \leqslant b$ 的最小值。当该算法被用于安排建设工程的进度时，请说明如何应用上述事实。

24.4-10　假设线性规划 $Ax \leqslant b$ 中，矩阵 A 的每一行对应于差分约束条件，即形如 $x_i \leqslant b_k$ 或 $-x_i \leqslant b_k$ 的单变量的约束条件。说明如何修改 Bellman-Ford 算法，以求得这种约束系统的解。

24.4-11　对所有 b 的元素均为实数，且所有未知量 x_i 必须是整数的情形，写出一个有效算法，以求得差分的约束系统 $Ax \leqslant b$ 的解。

*24.4-12　对所有 b 的元素均为实数且部分(并不一定是全部)未知量 x_i 必须是整数的情形，写出一个有效算法，以求得差分的约束系统 $Ax \leqslant b$ 的解。

24.5　最短路径性质的证明

贯穿本章，我们论证的几种正确性均依赖于三角不等式、上界性质、无路径性质、收敛性质、路径松弛性质和前趋子图性质。在本章开始时，我们未加证明地陈述过这些性质，本节就来对它们进行证明。

三角不等式

在广度优先搜索(22.2 节)的研究中，我们以引理 22.1 的形式，证明了无权图上最短距离的一个简单性质。三角不等式将这个性质推广到了带权图。

引理 24.10(三角不等式)　设 $G=(V, E)$ 为一带权有向图，其权函数 $w: E \to \mathbf{R}$，源点为 s。那么，对于所有边 $(u, v) \in E$，有

$$\delta(s, v) \leqslant \delta(s, u) + w(u, v)$$

607

证明：假定从源点 s 到顶点 v 存在一条最短路径 p，那么 p 的权不比从 s 到 v 的其他任何一条路径的权大。特别地，p 的权不大于从源点 s 到顶点 u 的最短路径的权再加上边 (u, v) 的权。

练习 24.5-3 要求处理从 s 到 v 没有最短路径的情况。 ■

对最短路径估计的松弛的效果

一加权有向图由 INITIALIZE-SINGLE-SOURCE 进行初始化，我们在其上对边执行了一系列的松弛步骤。下面一组引理描述最短路径估计是如何受影响的。

引理 24.11(上界性质)　设 $G=(V, E)$ 为有向加权图，其加权函数为 $w: E \to \mathbf{R}$。设 $s \in V$ 为源点，INITIALIZE-SINGLE-SOURCE(G, s) 对图进行了初始化。那么，对于所有 $v \in V$ 有

$d[v]\geqslant\delta(s,v)$，而且这个不变式对图 G 中边的任意系列松弛操作都保持不变。更进一步说，一旦 $d[v]$ 到达下界 $\delta(s,v)$ 将不再改变。

证明：通过对松弛步骤数目的归纳，我们证明对所有 $v\in V$ 的顶点，有不变式 $d[v]\geqslant\delta(s,v)$。

作为基础，在初始化以后 $d[v]\geqslant\delta(s,v)$ 为真，由于 $d[s]=0\geqslant\delta(s,s)$（注意，如果 s 在一个负权回路上，则 $\delta(s,s)$ 为 $-\infty$；否则为 0），而且 $d[v]=\infty$ 隐含着对所有 $v\in V-\{s\}$，有 $d[v]\geqslant\delta(s,v)$。

对归纳步骤，考虑边 (u,v) 的松弛。通过归纳假设，在松弛前，对所有 $x\in V$，有 $d[x]\geqslant\delta(s,x)$。唯一可能变化的 d 值为 $d[v]$。如果变化了，则有

$$\begin{aligned} d[v] &= d[u]+w(u,v) \\ &\geqslant \delta(s,u)+w(u,v) \quad \text{（根据归纳假设）} \\ &\geqslant \delta(s,v) \quad \text{（根据三角不等式）} \end{aligned}$$

因而不变式仍然保持。

一旦 $d[v]=\delta(s,v)$，则 $d[v]$ 的值将不再改变，值得注意的是达到了它的下界，$d[v]$ 值将不会减小了。因为我们刚刚证明了 $d[v]\geqslant\delta(s,v)$，而且松弛步骤也不会增加 d 值。■

推论 24.12（无路径性质） 假定在给定的带权有向图 $G=(V,E)$ 中，权函数为 $w:E\to\mathbf{R}$，
608 从一源点 $s\in V$ 到一给定顶点 $v\in V$ 不存在路径。那么，在 INITIALIZE-SINGLE-SOURCE(G,s) 对图初始化以后，有 $d[v]=\delta(s,v)=\infty$，在对于 G 的边进行任意序列的松弛操作后，这个等式作为循环不变式仍然保持。

证明：由上界性质，总有 $\infty=\delta(s,v)\leqslant d[v]$，那么 $d[v]=\infty=\delta(s,v)$。■

引理 24.13　设 $G=(V,E)$ 为一个带权有向图，其权函数 $w:E\to\mathbf{R}$，且 $(u,v)\in E$。那么，通过执行 RELAX(u,v,w) 松弛边 (u,v) 后，有 $d[v]\leqslant d[u]+w(u,v)$。

证明：如果在松弛边 (u,v) 之前，有 $d[v]>d[u]+w(u,v)$，那么之后就有 $d[v]=d[u]+w(u,v)$。或者，如果在松弛之前有 $d[v]\leqslant d[u]+w(u,v)$，那么 $d[u]$ 和 $d[v]$ 都不会变化，所以之后有 $d[v]\leqslant d[u]+w(u,v)$。■

引理 24.14（收敛性质） 设 $G=(V,E)$ 为一个带权有向图，其权函数为 $w:E\to\mathbf{R}$，$s\in V$ 为一个源点。对某些顶点 $u,v\in V$，设 $s\rightsquigarrow u\to v$ 为图 G 中的最短路径。假定 G 通过调用 INITIALIZE-SINGLE-SOURCE(G,s) 进行初始化，然后在图 G 的边上执行了包括调用 RELAX(u,v,w) 在内的一系列松弛步骤。如果在调用之前 $d[u]=\delta(s,u)$，那么在调用之后的任意时间 $d[v]=\delta(s,v)$。

证明：根据上界性质，如果在松弛边 (u,v) 之前的某一时刻 $d[u]=\delta(s,u)$，那么这个等式以后将一直保持下去。特别地，在松弛边 (u,v) 后，有

$$\begin{aligned} d[v] &\leqslant d[u]+w(u,v) \quad \text{（根据引理 24.13）} \\ &= \delta(s,u)+w(u,v) \\ &= \delta(s,v) \quad \text{（根据引理 24.1）} \end{aligned}$$

根据上界性质 $d[v]\geqslant\delta(s,v)$，可得 $d[v]=\delta(s,v)$，而且这个等式之后将一直保持。■

引理 24.15（路径松弛性质） 设 $G=(V,E)$ 为一带权有向图，权函数 $w:E\to\mathbf{R}$，$s\in V$ 为源点。考虑任意从 $s=v_0$ 到 v_k 的最短路径 $p=\langle v_0,v_1,\cdots,v_k\rangle$。如果 G 通过 INITIALIZE-SINGLE-SOURCE(G,s) 进行初始化，然后按顺序进行了一系列的松弛步骤，包括松弛边 (v_0,v_1)，$(v_1,$
609 $v_2)$，…，(v_{k-1},v_k)，那么，经过这些松弛后以及在以后的任意时刻，都有 $d[v_k]=\delta(s,v_k)$。无论其他边是否发生松弛（包括与 p 的边交错地进行的松弛），这一性质都始终保持。

证明：我们通过归纳：在路径 p 上第 i 条边被松弛后，有 $d[v_i]=\delta(s,v_i)$。作为基础，$i=$

0，也就是在 p 的任意边被松弛之前，根据初始化有 $d[v_0]=d[s]=0=\delta(s, s)$。根据上界性质，$d[s]$的值在初始化后将不再改变。

在归纳过程中，假设 $d[v_{i-1}]=\delta(s, v_{i-1})$，并检查边$(v_{i-1}, v_i)$的松弛。根据收敛性质，在这次松弛后 $d[v_i]=\delta(s, v_i)$，而且此后这个等式将一直成立。■

松弛和最短路径树

现在我们来证明一旦一系列松弛操作使得最短路径估计收敛到最短路径权，那么由 π 值导出的前趋子图 G_π 就是 G 的最短路径树。首先证明如下引理，它表明了前趋子图总是形成一个以源点为根的有根树。

引理 24.16　设 $G=(V, E)$为一带权有向图，其权值函数为 $w: E\to\mathbf{R}$，$s\in V$ 为一个源点，并假定 G 不含从 s 可达的负权回路。那么，在图被 INITIALIZE-SINGLE-SOURCE(G, s)初始化后，前趋子图 G_π 就构成以 s 为根的有根树，在对 G 边任意序列的松弛操作下仍然像不变式一样保持这个性质。

证明：初始时，G_π 中唯一的顶点就是源点，引理显然成立。考虑经过了一系列松弛步骤后的一个前趋子图 G_π，我们首先证明 G_π 是无环的。为了引出矛盾，假设某个松弛操作在图 G_π 中构出了环。设环为 $c=\langle v_0, v_1, \cdots, v_k\rangle$，其中 $v_k=v_0$。那么，对 $i=1, 2, \cdots, k$ 有 $\pi[v_i]=v_{i-1}$，而且不失一般性，可以假设是对边(v_{k-1}, v_k)的松弛造成了 G_π 中的环。

我们断言环 c 上的所有顶点都从源点 s 可达。为什么呢？c 上的每个顶点都有一个非 NIL 前趋，所以当 c 上的每个顶点都被赋予一个非 NIL π 值时，就被赋予了一个有限最短路径估计。根据上界性质，环 c 上的每个顶点都有一个有限最短路径权，这隐含着它们都是从 s 可达的。

在调用 RELAX(v_{k-1}, v_k, w)之前，我们将检查 c 上的最短路径估计，并证明 c 是一个负权环，那么与 G 不包含从 s 可达的负权环的假设矛盾。在调用之前，对 $i=1, 2, \cdots, k-1$，有 $\pi[v_i]=v_{i-1}$。 610
那么，对于 $i=1, 2, \cdots, k-1$，对 $d[v_i]$的最后一次更新是 $d[v_i]\leftarrow d[v_{i-1}]+w(v_{i-1}, v_i)$。如果那时 $d[v_{i-1}]$改变了，它将减小。因此，在调用 RELAX(v_{k-1}, v_k, w)之前，有

$$d[v_i]\geqslant d[v_{i-1}]+w(v_{i-1}, v_i)\quad \text{对于所有 } i=1,2,\cdots,k-1 \tag{24.12}$$

因为调用改变了 $\pi[v_k]$的值，有下列严格的不等式成立：

$$d[v_k]>d[v_{k-1}]+w(v_{k-1}, v_k)$$

把这一严格不等式和 $k-1$ 个不等式(24.12)相加，我们就得到回路 c 的最短路径估计的和：

$$\begin{aligned}\sum_{i=1}^{k} d[v_i] &> \sum_{i=1}^{k}(d[v_{i-1}]+w(v_{i-1}, v_i))\\ &= \sum_{i=1}^{k} d[v_{i-1}]+\sum_{i=1}^{k} w(v_{i-1}, v_i)\end{aligned}$$

但是因为回路 c 中每个顶点在每个和式中仅出现一次，因此

$$\sum_{i=1}^{k} d[v_i]=\sum_{i=1}^{k} d[v_{i-1}]$$

这说明

$$0>\sum_{i=1}^{k} w(v_{i-1}, v_i)$$

所以沿回路 c 的权之和为负，因而产生矛盾。

现在，我们已证明了 G_π 是有向无回路图。要证明它是一棵以 s 为根的有根树，只要证明下列结论就可以了，即对每个顶点 $v\in V_\pi$，G_π 中存在着从 s 到 v 的唯一路径(见练习 B.5-2)。

我们首先说明对 V_π 中的每个顶点存在一条从 s 出发的路径。V_π 中的顶点是其 π 值为非 NIL

的顶点再加上顶点 s。设法通过归纳证明，从 s 到 V_π 中的所有顶点之间均有通路。证明的详细过程留作练习 24.5-6。

为了完成对本引理的证明，必须证明对任意顶点 $v\in V_\pi$，图 G_π 中从 s 到 v 至多存在一条路径。假设相反，即，假定从 s 到某顶点 v 存在两条简单通路，p_1 和 p_2，其中 p_1 可分解为 $s\rightsquigarrow u\rightarrow x\rightarrow z\rightsquigarrow v$，$p_2$ 可分解为 $s\rightsquigarrow u\rightarrow y\rightarrow z\rightsquigarrow v$，$x\neq y$(见图 24-9)，则有 $\pi[z]=x$，$\pi[z]=y$。这 611
说明 $x=y$，与假设相矛盾。由此可得出结论，G_π 中从 s 到 v 仅存在唯一的简单路径，因此 G_π 是一棵以 s 为根的有根树。■

现在我们可以说明：如果在进行了一系列松弛操作后，真正的最短路径的权被赋给了每个顶点，那么前趋子图 G_π 是一棵最短路径树。

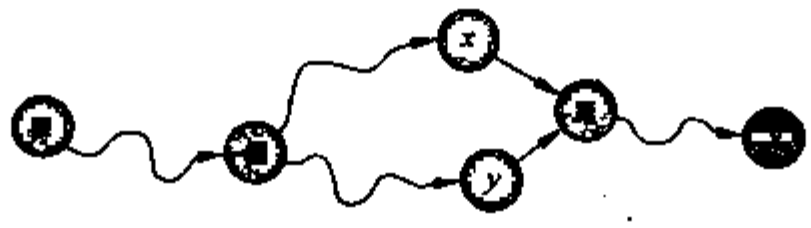


图 24-9　说明 G_π 中从源点 s 到顶点 v 的路径是唯一的。如果存在两条路径 $p_1(s\rightsquigarrow u\rightsquigarrow x\rightarrow z\rightsquigarrow v)$ 和 $p_2(s\rightsquigarrow u\rightsquigarrow y\rightarrow z\rightsquigarrow v)$，其中 $x\neq y$，那么有 $\pi[z]=x$ 和 $\pi[z]=y$，形成一个矛盾

引理 24.17(前趋子图性质)　设 $G=(V, E)$为一带权有向图，其权函数 $w: E\rightarrow\mathbf{R}$，$s\in V$ 为一个源点，而且假定 G 不含 s 可达的负权回路。设调用了 INITIALIZE-SINGLE-SOURCE(G, s)，然后在 G 的边上执行了一系列的松弛操作，得到对所有 $v\in V$ 有 $d[v]=\delta(s, v)$。因此，前趋子图 G_π 是一个以 s 为根的最短路径树。

证明：我们必须证明 G_π 满足本章在前面给出的最短路径树的三个性质。关于第一个性质，必须证明 V_π 是从 s 可达的顶点集合。根据定义，最短路径的权 $\delta(s, v)$为有限值，当且仅当 v 从 s 可达。所以，从 s 可达的顶点实际上就是那些具有有限 d 值的顶点。但对于顶点 $v\in V-\{s\}$，其 $d[v]$被赋予一有限值当且仅当 $\pi[v]\neq$NIL。因此，V_π 中的顶点正是那些从 s 可达的顶点。

由引理 24.16 可知第二条性质成立。

于是，现在剩下的就是证明最短路径树的最后一条性质：对每个顶点 $v\in V_\pi$，G_π 中的唯一简单路径 $s\overset{p}{\rightsquigarrow}v$ 就是 G 中从 s 到 v 的最短路径。设 $p=\langle v_0, v_1, v_2, \cdots, v_k\rangle$，其中 $v_0=s$ 且 612
$v_k=v$。对 $i=1, 2, \cdots, k$，有 $d[v_i]=\delta(s, v_i)$和 $d[v_i]\geqslant d[v_{i-1}]+w(v_{i-1}, v_i)$均成立，由此可得 $w(v_{i-1}, v_i)\leqslant\delta(s, v_i)-\delta(s, v_{i-1})$。沿路径 p 把权相加，得

$$\begin{aligned}w(p)&=\sum_{i=1}^{k}w(v_{i-1},v_i)\leqslant\sum_{i=1}^{k}(\delta(s,v_i)-\delta(s,v_{i-1}))\\&=\delta(s,v_k)-\delta(s,v_0)\quad\text{(由于和量缩)}\\&=\delta(s,v_k)\quad\text{(因为 }\delta(s,v_0)=\delta(s,s)=0)\end{aligned}$$

因此 $w(p)\leqslant\delta(s, v_k)$。因为 $\delta(s, v_k)$是从 s 到 v_k 的任意路径权值的下界，所以有 $w(p)=\delta(s, v_k)$，而且 p 是从 s 到 $v=v_k$ 的最短路径。■

练习

24.5-1　对图 24-2，除图中已画出的两棵树以外，另外再画出两棵图中所示有向图的最短路径树。

24.5-2　举出一个带权有向图 $G=(V, E)$的实例，其加权函数为 $w: E\rightarrow\mathbf{R}$，且源点为 s，要求 G 满足下列性质：对每条边 $(u, v)\in E$，存在包含 (u, v)且以 s 为根的最短路径树，同时存在另一棵以 s 为根，但不包含 (u, v)的最短路径树。

24.5-3　对引理 24.5-3 的证明过程进行修改，使其能在最短路径的权为∞或－∞时，证明引理的正确性。

24.5-4　设 $G=(V, E)$是带权有向图，源点为 s，并设 G 由过程 INITIALIZE-SINGLE-SOURCE(G, s)进行了初始化。证明如果经过一系列松弛操作，$\pi[s]$的值被置为非 NIL，则 G 中包含一个负权回路。

24.5-5　设 $G=(V, E)$为带权有向图且不含负权边。设 $s\in V$ 为源点若 $v\in V-\{s\}$为从 s 可达的顶点，则 $\pi[v]$是从源 s 到 v 的某最短路径中顶点 v 的前趋，否则 $\pi[v]$为 NIL。举出这样的一个图 G 和给 π 赋值的一个例子，说明可以在 G_π 中产生回路。（根据引理 24.16，这样一种赋值不可能由一个松弛操作序列所产生。）

613

24.5-6　设 $G=(V, E)$为带权有向图，其权值函数为 $w: E\to\mathbf{R}$，且图中不包含负权回路。设 $s\in V$ 为源点，且 G 由 INITIALIZE-SINGLE-SOURCE(G, s)进行了初始化。证明对每一顶点 $v\in V_\pi$，G_π 中存在一条从 s 到 v 的通路，且经过任意序列的松弛操作后，这一性质依然保持。

24.5-7　设 $G=(V, E)$为带权有向图且不包含负权回路。设 $s\in V$ 为源点且 G 由 INITIALIZE-SINGLE-SOURCE(G, s)进行了初始化。证明存在 $|V|-1$ 步的松弛序列，使得对所有 $v\in V$，$d[v]=\delta(s, v)$。

24.5-8　设 G 为任意带权有向图，且存在一源点 s 可达负权回路。证明对 G 的边总可以构造一个无限的松弛序列，使得每个松弛步骤都能对最短路径估计进行修改。

思考题

24-1　对 Bellman-Ford 算法的 Yen 氏改进

假设对 Bellman-Ford 算法每一趟中边的松弛顺序作如下安排，在第一趟执行之前，把一任意线性序列 $v_1, v_2, \cdots, v_{|v|}$ 赋值给输入图 $G=(V, E)$的各点。然后把边的集合 E 分为 $E_f\cup E_b$，其中 $E_f=\{(v_i, v_j)\in E: i<j\}$，$E_b=\{(v_i, v_j)\in E: i>j\}$。（假定 G 中不含自环，所以每条边或在 E_f，或在 E_b 中。）定义 $G_f=(V, E_f)$，$G_b=(V, E_b)$。

a）证明对拓扑序列$\langle v_1, v_2, \cdots, v_{|v|}\rangle$，$G_f$ 是无回路图；对拓扑序列$\langle v_{|v|}, v_{|v|-1}, \cdots, v_1\rangle$，$G_b$ 是无回路图。

假设我们对 Bellman-Ford 算法中的每一趟按下列方法实现：按 $v_1, v_2, \cdots, v_{|v|}$ 的顺序访问每个顶点，同时对从该顶点出发的 E_f 中的边进行松弛。然后再按 $v_{|v|}, v_{|v|-1}, \cdots, v_1$ 的顺序访问每个顶点，同时对从该点出发的 E_b 中的边进行松弛。

b）证明在这一方案中，如果 G 不包含从源点 s 可达且权为负的回路，则对边仅执行$\lceil |V|/2\rceil$趟操作后，对所有点 $v\in V$，有 $d[v]=\delta(s, v)$。

614

c）这种方案是否提高了 Bellman-Ford 算法的渐近运行时间？

24-2　嵌套框

如果存在$\{1, 2, \cdots, d\}$上的某一排列 π，满足 $x_{\pi(1)}<y_1, x_{\pi(2)}<y_2, \cdots, x_{\pi(d)}<y_d$，则称一个 d 维框$(x_1, x_2, \cdots, x_d)$嵌入另一个 d 维框$(y_1, y_2, \cdots, y_d)$中。

a）证明嵌套关系具有传递性。

b）描述一个有效算法以确定某 d 维框是否嵌套于另一 d 维框中。

c）假定给出一个由 n 个 d 维框组成的集合$\{B_1, B_2, \cdots, B_n\}$，写出一有效算法以找出满足条件 B_{i_j} 嵌入 $B_{i_{j+1}}$，$j=1, 2, \cdots, k-1$ 的最长嵌套框序列$\langle B_{i_1}, B_{i_2}, \cdots, B_{i_k}\rangle$。用变量 n 和 d 来描述所给出的算法的运行时间。

24-3　套汇问题

套汇是指利用货币兑率的差异，把一个单位的某种货币转换为大于一个单位的同种货币的方法。例如，假定1美元可以买46.4印度卢比，1印度卢比可以买2.5日元，1日元可以买0.0091美元。通过货币兑换，一个商人可以从1美元开始买入，得到46.4×2.5×0.0091=1.0556美元，因而获得5.56%的利润。

假定已知n种货币c_1，c_2，…，c_n和有关兑换率的$n\times n$的表R，一单位货币c_i可以买入$R[i, j]$单位的货币c_j。

a)写出一个有效算法，以确定是否存在一个货币序列$\langle c_{i1}, c_{i2}, \cdots, c_{ik}\rangle$满足

$$R[i_1, i_2]\cdot R[i_2, i_3]\cdots R[i_{k-1}, i_k]\cdot R[i_k, i_1] > 1$$

并分析你的算法的运行时间。

b)写出一个有效的算法来输出该序列(如果存在这样的序列的话)，并分析算法的运行时间。

24-4　关于单源最短路径的Gabow定标算法

定标算法对问题进行求解，开始时仅考虑每个相应输入值(例如边的权)的最高位，接
615 着通过察看最高两位对初始答案进行细微调整，这样逐步查看越来越多的高位信息，同时每次均对前面的解答进行细微调整，直至所有位均被考虑在内，并且正确答案得出为止。

在这一问题中，我们考察一种通过对边的权进行定标操作而计算单源最短路径的算法。给定边的权为非负整数w的有向图$G=(V, E)$，设$W=\max_{(u,v)\in E}\{w(u, v)\}$，我们的目标是设计出一种运行时间为$O(E\lg W)$的算法，假设所有顶点均从源点可达。

该算法对边权的二进制表示，从最高有效位到最低有效位每次检查一部分位。具体地，设$k=\lceil\lg(W+1)\rceil$是W的二进制表示的位数，而且对$i=1, 2, \cdots, k$，$w_i(u, v)=\lfloor w(u, v)/2^{k-i}\rfloor$。这就是说，$w_i(u, v)$是由$w(u, v)$高$i$位有效位对$w(u, v)$“按比例缩小”的描述(因此对所有$(u, v)\in E$，$w_k(u, v)=w(u, v)$)。例如，若$k=5$，$w(u, v)=25$，其相应的二进制数为$\langle 11001\rangle$，则$w_3(u, v)=\langle 110\rangle=6$。又如对$k=5$，若$w(u, v)=\langle 00100\rangle=4$，则$w_3(u, v)=\langle 001\rangle=1$。我们定义$\delta_i(u, v)$为用权函数$w_i$计算出从顶点$u$到顶点$v$的最短路径的权。因此对所有$u$，$v\in V$，有$\delta_k(u, v)=\delta(u, v)$。对于一指定源点$s$，定标算法首先对所有$v\in V$计算出最短路径的权$\delta_1(s, v)$，接着对所有$v\in V$计算出$\delta_2(s, v)$，如此下去，直至对所有$v\in V$计算出$\delta_k(s, v)$。假定在整个计算过程中都有$|E|\geqslant|V|-1$，并且我们将看到从$\delta_{i-1}$计算出$\delta_i$需要运行时间$O(E)$，所以整个算法的运行时间为$O(kE)=O(E\lg W)$。

a)假设对所有顶点$v\in V$，有$\delta(s, v)\leqslant|E|$。证明对所有$v\in V$，可以在$O(E)$的运行时间内计算出$\delta(s, v)$。

b)证明对所有$v\in V$，可以在$O(E)$的运行时间内计算出$\delta_1(s, v)$。

现在我们集中注意力看看如何根据δ_{i-1}计算出δ_i。

c)证明对$i=2, 3, \cdots, k$，要么$w_i(u, v)=2w_{i-1}(u, v)$，要么有$w_i(u, v)=2w_{i-1}(u, v)+1$。然后再证明对所有$v\in V$，有

$$2\delta_{i-1}(s, v)\leqslant\delta_i(s, v)\leqslant 2\delta_{i-1}(s, v)+|V|-1$$

d)对$i=2, 3, \cdots, k$和所有边$(u, v)\in E$，我们定义

$$\hat{w}_i(u, v)=w_i(u, v)+2\delta_{i-1}(s, u)-2\delta_{i-1}(s, v)$$

616 证明对$i=2, 3, \cdots, k$和所有u，$v\in V$，边(u, v)被“再加权”的权值$\hat{w}_i(u, v)$是非负整数。

e)定义$\hat{\delta}_i(s, v)$为使用仅函数$\hat{w}_i$的从s到v的最短路径的权。证明对$i=2, 3, \cdots, k$

及所有的 $v\in V$，有

$$\delta_i(s,v)=\hat{\delta}_i(s,v)+2\delta_{i-1}(s,v)$$

且 $\hat{\delta}_i(s,v)\leqslant|E|$。

f)说明对所有 $v\in V$，如何在 $O(E)$ 运行时间内，根据 $\delta_{i-1}(s,v)$ 计算出 $\delta_i(s,v)$，并推断出对所有的 $v\in V$，可以在 $O(E\lg W)$ 运行时间内计算出 $\delta(s,v)$。

24-5　Karp 最小平均权值回路算法

设 $G=(V,E)$ 是权值函数为 $w:E\to\mathbf{R}$ 的有向图，且 $n=|V|$，定义 E 中某边的回路 $c=\langle e_1,e_2,\cdots,e_k\rangle$ 的平均权值为

$$\mu(c)=\frac{1}{k}\sum_{i=1}^{k}w(e_i)$$

设 $\mu^*=\min_c\mu(c)$，其中 c 的取值范围为 G 中的所有有向回路。满足 $\mu(c)=\mu^*$ 的回路 c 称为最小平均权值回路。本问题中将寻求一有效算法来计算 μ^*。

假定不失一般性，每个顶点 $v\in V$ 均从源点 $s\in V$ 可达。设 $\delta(s,v)$ 为从 s 到 v 的最短路径的权，且 $\delta_k(s,v)$ 为从 s 到 v 仅由 k 条边组成的一最短路径的权。若从 s 到 v 不存在仅由 k 条边组成的通路，则 $\delta_k(s,v)=\infty$。

a)证明若 $\mu^*=0$，则 G 不包含负权回路，且对所有的顶点 $v\in V$，有 $\delta(s,v)=\min_{0\leqslant k\leqslant n-1}\delta_k(s,v)$。

b)证明若 $\mu^*=0$，那么对所有顶点 $v\in V$，有：

$$\max_{0\leqslant k\leqslant n-1}\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}\geqslant 0$$

（提示：利用(a)中的两个性质。）

c)设 c 为权为 0 的回路，且 u 和 v 是回路 c 中的任意两个顶点。假定沿回路上从 u 到 v 的路径的权为 x。证明：$\delta(s,v)=\delta(s,u)+x$。（提示：沿回路从 v 到 u 的路径的权为 $-x$。）

d)证明若 $\mu^*=0$，则在每个最小平均权值回路中存在一个顶点 v，满足：　617

$$\max_{0\leqslant k\leqslant n-1}\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}=0$$

（提示：证明到达最小平均权值回路中任何顶点的一条最短路径可以沿回路被延长，从而取得一条到达该回路中下一个顶点的最短路径。）

e)证明若 $\mu^*=0$，则

$$\min_{v\in V}\max_{0\leqslant k\leqslant n-1}\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}=0$$

f)证明：若把 G 中每条边的权加上常数 t，则 μ^* 也增加 t。运用这一结果证明：

$$\mu^*=\min_{v\in V}\max_{0\leqslant k\leqslant n-1}\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}$$

g)给出一个 $O(VE)$ 时间的算法来计算 μ^*。

24-6　双调最短路径

如果一个序列首先单调递增，然后再单调递减，或者能够通过循环移位来单调递增再单调递减，这样的序列就是双调(bitonic)的。例如序列$\langle 1,4,6,8,3,-2\rangle$，$\langle 9,2,-4,-10,-5\rangle$和$\langle 1,2,3,4\rangle$是双调的，但是$\langle 1,3,12,4,2,10\rangle$不是双调的。（有关双调排序的讨论可参见第 27 章，有关双调欧拉旅行商问题可参见思考题 15-1。）

假设给定一个权值函数为 $w:E\to\mathbf{R}$ 的有向图 $G=(V,E)$，我们希望找到从源点 s 出发的单源最短路径。我们还有另外一个信息：对每个顶点 $v\in V$，任何从 s 到 v 的最短路径

上边的权构成一个双调序列。

请给出你解决此问题的最高效算法，并分析其运行时间。

本章注记

Dijkstra 算法[75]出现在 1959 年，但并未提及优先队列。Bellman-Ford 算法是分别以 Bellman[35]和 Ford[95]两算法为基础的。Bellman 描述了最短路径和差分约束之间的关系。
618 Lawler[196]考虑了民间流行的部分，描述了在 dag 图中解决最短路径的线性时间算法。

当边的权值是相对较小的非负整数时，单源最短路径问题有着更高效的解决算法。在 Dijkstra 算法中，调用 EXTRACT-MIN 后，返回的值序列是根据时间单调递增的。正如第 6 章注记中所讨论到的，在这种情况下，有很多数据结构能够比二叉堆或斐波那契堆更高效地实现各种优先队列的操作。Ahuja，Mehlhorn，Orlin 和 Tarjan[8]给出了一个无负权图上、需要 $O(E+V\sqrt{\lg W})$时间的算法，其中 W 是图中任一边的最大权值。最好的界是由 Thorup[299]和 Raman 所给出的，前者给出了一个运行时间为 $O(E\lg\lg V)$的算法，后者提出了一个运行时间为 $O(E+V\min\{(\lg V)^{1/3+\epsilon}, (\lg W)^{1/4+\epsilon}\})$的算法。这两个算法使用空间的大小依赖于运行主机的字长。虽然使用空间的量可以不受输入规模的限制，利用随机化散列技术，它可以被归约为与输入规模呈线性关系。

对带整数权值的无向图，Thorup[298]给出了一个 $O(V+E)$时间的算法，来求解单源最短路径。与前一段所提及的算法不同，因为调用 EXTRACT-MIN 所返回的值序列不是随时间单调递增的，所以这个算法不是 Dijkstra 算法的实现。

对于带负权边的图，Gabow 和 Tarjan[104]给出了一个运行时间为 $O(\sqrt{V}E\lg(VW))$的算法，Goldberg[118]给出了一个运行时间为 $O(\sqrt{V}E\lg W)$的算法，其中 $W=\max_{(u,v)\in E}\{|w(u,v)|\}$。

619 Cherkassky，Goldberg 和 Radzik[57]对各种最短路径算法进行了细致的实验比较。

第 25 章　每对顶点间的最短路径

在本章中，我们讨论找出图中每对顶点间最短路径的问题。例如，对一张公路图，需要制表说明每对城市间的距离，就可能出现这种问题。与第 24 章中一样，给定一加权有向图 $G=(V, E)$，其加权函数 $w: E \to \mathbf{R}$ 为边到实数权值的映射。对于每对顶点 $u, v \in V$，我们希望找出从 u 到 v 的一条最短(最小权)路径，其中路径的权值是指其组成边的权值之和。我们通常希望以表格形式输出结果：第 u 行第 v 列的元素应是从 u 到 v 的最短路径的权值。

可以把单源最短路径算法运行 $|V|$ 次来解决每对顶点间最短路径问题，每一次运行时，轮流把每个顶点作为源点。如果所有边的权值是非负的，可以采用 Dijkstra 算法。如果采用线性数组来实现最小优先队列，算法的运行时间为 $O(V^3+VE)=O(V^3)$。如果是稀疏图，采用二叉最小堆来实现最小优先队列，就可以把算法的运行时间改进为 $O(VE \lg V)$。或者，采用斐波那契堆来实现最小优先队列，其算法运行时间为 $O(V^2 \lg V+VE)$。

如果允许有负权值的边，就不能采用 Dijkstra 算法。必须对每个顶点运行一次速度较慢的 Bellman-Ford 算法。它的运行时间为 $O(V^2E)$，而在稠密图上的运行时间为 $O(V^4)$。在本章我们将看到如何更好地解决这个问题。此外，也将探讨每对顶点间的最短路径问题与矩阵乘法的关系，并研究其代数结构。

在前面的单源算法中，假定采用图的邻接表表示法。与此不同，本章中的大多数算法均采用邻接矩阵表示法。(在稀疏图的 Johnson 算法中采用邻接表。)为方便起见，假设顶点编号为 1，2，…，$|V|$，于是输入是一个 $n \times n$ 的矩阵 W，其表示 n 个顶点的有向图 $G=(V, E)$ 中边的权值。也就是 $W=(w_{ij})$，其中

$$w_{ij}=\begin{cases}0 & \text{如果 } i=j \\ \text{有向边}(i,j)\text{ 的权} & \text{如果 } i\neq j \text{ 且 } (i,j)\in E \\ \infty & \text{如果 } i\neq j \text{ 且 } (i,j)\notin E\end{cases} \tag{25.1}$$

允许存在权值为负的边，但目前我们假定输入图中不包含权值为负的回路。

本章中每对顶点间最短路径算法的输出是一个 $n \times n$ 的矩阵 $D=(d_{ij})$，其中元素 d_{ij} 是从 i 到 j 的最短路径的权值。就是说，如果用 $\delta(i, j)$ 表示从顶点 i 到顶点 j 的最短路径的权值(如第 24 章所述)，则在算法终止时 $d_{ij}=\delta(i, j)$。

为了求解对输入邻接矩阵的每对顶点间最短路径问题，不仅要算出最短路径的权值，而且要计算出一个前驱矩阵 $\Pi=(\pi_{i,j})$，其中若 $i=j$ 或从 i 到 j 没有通路，则 $\pi_{i,j}$ 为 NIL，否则 π_{ij} 表示从 i 出发的某条最短路径上 j 的前驱顶点。正如在第 24 章中提到的前趋子图 G_π 是一个给定源顶点的一棵最短路径树，由 Π 矩阵的第 i 行导出的子图应是以 i 为根的一棵最短路径树。对每个顶点 $i \in V$，定义 G 对于 i 的前趋子图为 $G_{\pi,i}=(V_{\pi,i}, E_{\pi,i})$，其中

$$V_{\pi,i}=\{j \in V:\pi_{ij}\neq \text{NIL}\}\cup\{i\}$$

和

$$E_{\pi,i}=\{(\pi_{ij}, j): j \in V_{\pi,i}-\{i\}\}$$

如果 $G_{\pi,i}$ 是一棵最短路径树，则下列过程将输出从顶点 i 到顶点 j 的一条最短路径，它是由第 22 章中的过程 PRINT-PATH 修改而成的。

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
1  if i=j
2     then print i
```

```
3    else if π_ij = NIL
4         then print "no path from" i "to" j "exists"
5         else PRINI-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
6              print j
```

为了突出本章中每对顶点算法的本质特征，我们不可能像第 24 章阐述前趋子图那样花大篇
621 幅去讨论前趋矩阵的建立及其性质。基本性质将在某些练习中讨论。

本章概述

25.1 节介绍一个基于矩阵乘法的动态规划算法，求解每对顶点间的最短路径问题。由于采用了"重复平方"(repeated squaring)的技术，算法的运行时间为 $\Theta(V^3 \lg V)$。25.2 节给出另一种动态规划算法，即 Floyd-Warshall 算法。该算法的运行时间为 $\Theta(V^3)$。25.2 节还讨论求有向图传递闭包的问题，这一问题与每对顶点间最短路径有关系。最后，26.3 节介绍 Johnson 算法。与本章中其他算法不同，Johnson 算法采用图的邻接表表示法。该算法求解每对顶点间最短路径问题所需的时间为 $O(V^2 \lg V + VE)$，对大型稀疏图来说这是一个很好的算法。

在讨论之前，有必要对邻接矩阵表示法建立一些约定。首先，一般假设输入图 $G=(V, E)$有 n 个顶点，所以 $n=|V|$。其次，我们按常规用大写字母来表示矩阵，例如 W，L 或 D，用带有下标的小写字母来表示矩阵的元素，如 w_{ij}，l_{ij} 或 d_{ij}。有些矩阵有带括号的上标，如 $L^{(m)}=(l_{ij}^{(m)})$或 $D^{(m)}=(d_{ij}^{(m)})$，用来表示迭代。最后，对给定的 $n \times n$ 矩阵 A，假设 n 的值保存在属性 $rows[A]$中。

25.1　最短路径与矩阵乘法

本节要介绍一个动态规划算法，用来解决有向图 $G=(V, E)$上每对顶点间的最短路径问题。动态规划的每一次主循环都将引发一个与矩阵乘法运算十分相似的操作，因此算法看上去很像是重复的矩阵乘法。开始时，先找到一种运行时间为 $\Theta(V^4)$的算法，来解决每对顶点间的最短路径问题，然后改进这一算法，使其运行时间达到 $\Theta(V^3 \lg V)$。

在继续讨论之前，先来扼要地重述一下第 15 章中给出的设计动态规划算法的几个步骤：

1)描述一个最优解的结构。

2)递归定义一个最优解的值。

3)按自底向上的方式计算一个最优解的值。

622 (第 4 步"从计算出的信息中构造一个最优解"将在练习中讨论。)

最短路径的结构

我们先来描述最优解的结构。对于图 $G=(V, E)$上每对顶点间的最短路径问题，已在引理 24.1 中证明了最短路径的所有子路径也是最短路径。假设图以邻接矩阵 $W=(w_{ij})$来表示。考察从顶点 i 到顶点 j 的一条最短路径 p，假设 p 至多包含 m 条边。假设图中不存在权值为负的回路，则 m 必是有限值。如果 $i=j$，则路径 p 权值为 0 而且没有边。若顶点 i 和顶点 j 是不同顶点，则把路径 p 分解为 $i \overset{p'}{\rightsquigarrow} k \rightarrow j$，其中路径 p'至多包含 $m-1$ 条边。由引理 24.1，p'是从 i 到 k 的一条最短路径，而且 $\delta(i, j)=\delta(i, k)+w_{kj}$。

每对顶点间最短路径问题的一个递归解

现在设 $l_{ij}^{(m)}$是从顶点 i 到顶点 j 的至多包含 m 条边的任何路径的权值最小值。当 $m=0$ 时，从 i 到 j 存在一条不包含边的最短路径当且仅当 $i=j$。因此

$$l_{i,j}^{(0)} = \begin{cases} 0 & \text{若 } i = j \\ \infty & \text{若 } i \neq j \end{cases}$$

对 $m \geqslant 1$，先计算 $l_{ij}^{(m-1)}$（从 i 到 j 的最短路径的权至多包含 $m-1$ 条边），以及从 i 到 j 的至多包含 m 条边的路径的最小权值，后者是通过计算 j 的所有可能前趋 k 而得到的，然后取二者中的最小值作为 $l_{ij}^{(m)}$。因此我们递归定义

$$l_{ij}^{(m)} = \min(l_{ij}^{(m-1)}, \min_{1 \leqslant k \leqslant n}\{l_{ik}^{(m-1)} + w_{kj}\}) = \min_{1 \leqslant k \leqslant n}\{l_{ik}^{(m-1)} + w_{kj}\} \tag{25.2}$$

后一等式成立是因为对所有 j，$w_{jj} = 0$。

实际的最短路径权值 $\delta(i, j)$是多少呢？如果图中不包含权值为负的回路，那么对于所有的顶点对 i 和 j，当 $\delta(i, j) < \infty$时，存在一条从 i 到 j 的最短路径，它是简单路径，从而至多包含 $n-1$ 条边。从顶点 i 到顶点 j 的多于 $n-1$ 条边的路径其权值不可能小于从 i 到 j 的最短路径的权值。因此，实际的最短路径权值由下式给出

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots \tag{25.3}$$

自底向上计算最短路径的权值

把矩阵 $W = (w_{ij})$作为输入，来计算一组矩阵 $L^{(1)}$，$L^{(2)}$，…，$L^{(n-1)}$，其中对 $m = 1$，2，…，$n-1$，有$L^{(m)} = (l_{ij}^{(m)})$。最后矩阵 $L^{(n-1)}$包含实际的最短路径权值。注意，对所有的顶点 i，$j \in V$，$l_{ij}^{(1)} = w_{ij}$，因此 $L^{(1)} = W$。 623

算法的核心如下面的过程所示，给定矩阵 $L^{(m-1)}$ 和 W，返回矩阵 $L^{(m)}$。也就是它把已经计算出来的最短路径延长一条边。

```
EXTEND-SHORTEST-PATHS(L, W)
1  n ← rows[L]
2  let L' = (l'_ij) be an n×n matrix
3  for i ← 1 to n
4       do for j ← 1 to n
5            do l'_ij ← ∞
6               for k ← 1 to n
7                  do l'_ij ← min(l'_ij , l_ik + w_kj)
8  return L'
```

该过程计算出矩阵 $L' = (l'_{ij})$作为返回值。对所有的 i 和 j 计算等式(25.2)，用 L 代表 $L^{(m-1)}$，用 L'代表$L^{(m)}$，来计算这个返回值。(不用上标形式写出，是为了使输入矩阵和输出矩阵独立于 m。)由于算法中存在三个嵌套的 for 循环，因此运行时间为 $\Theta(n^3)$。

现在来讨论算法和矩阵乘法运算的关系。假定我们希望计算两个 $n \times n$ 的矩阵 A 和 B 的矩阵乘积$C = A \cdot B$。于是对 i，$j = 1$，2，…，n，计算

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{25.4}$$

注意如果对等式(25.2)做如下的替换

$l^{(m-1)}$	$\rightarrow a$
w	$\rightarrow b$
$l^{(m)}$	$\rightarrow c$
min	$\rightarrow +$
$+$	$\rightarrow \cdot$

则得到等式(25.4)。因此，如果对过程 EXTEND-SHORTEST-PATHS 进行这样的变换，并用 0
624 (表示+)替换∞(表示 min)，就得到一个直接的 $\Theta(n^3)$时间的矩阵乘法过程：

```
MATRIX-MULTIPLY(A, B)
1  n ← rows[A]
2  let C be an n×n matrix
3  for i ← 1 to n
4       do for j ← 1 to n
5             do c_ij ← 0
6                for k ← 1 to n
7                   do c_ij ← c_ij + a_ik · b_kj
8  return C
```

回到每对顶点的最短路径问题，我们是通过把最短路径逐边延长，而最终计算出最短路径权值的。设 $A \cdot B$ 代表过程 EXTEND-SHORTEST-PATHS(A, B)返回的矩阵"乘积"，我们计算 $n-1$ 个矩阵的序列：

$$
\begin{aligned}
L^{(1)} &= L^{(0)} \cdot W = W \\
L^{(2)} &= L^{(1)} \cdot W = W^2 \\
L^{(3)} &= L^{(2)} \cdot W = W^3 \\
&\vdots \\
L^{(n-1)} &= L^{(n-2)} \cdot W = W^{n-1}
\end{aligned}
$$

如前面的论证一样，矩阵 $L^{(n-1)} = W^{n-1}$ 包含最短路径的权值。下面的过程在 $\Theta(n^4)$时间内计算该序列。

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← rows[W]
2  L^(1) ← W
3  for m ← 2 to n−1
4       do L^(m) ← EXTEND-SHORTEST-PATHS(L^(m−1), W)
5  return L^(n−1)
```

图 25-1 显示了一个图，以及由过程 SLOW-ALL-PAIRS-SHORTEST-PATHS 计算出的矩阵 $L^{(m)}$。

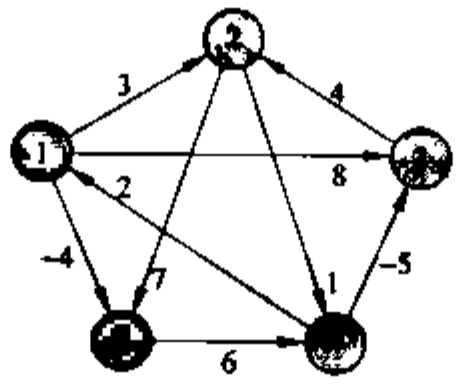


图 25-1　一个有向图，以及由过程 SLOW-ALL-PAIRS-SHORTEST-PATHS 计算出的矩阵序列 $L^{(m)}$。读者可自行验证 $L^{(5)} = L^{(4)}$。W 等于 $L^{(4)}$，因此对所有 $m \geqslant 4$，有 $L^{(m)} = L^{(4)}$

$$L^{(1)}=\begin{pmatrix}0&3&8&\infty&-4\\\infty&0&\infty&1&7\\\infty&4&0&\infty&\infty\\2&\infty&-5&0&\infty\\\infty&\infty&\infty&6&0\end{pmatrix}\quad L^{(2)}=\begin{pmatrix}0&3&8&2&-4\\3&0&-4&1&7\\\infty&4&0&5&11\\2&-1&-5&0&-2\\8&\infty&1&6&0\end{pmatrix}$$

$$L^{(3)}=\begin{pmatrix}0&3&-3&2&-4\\3&0&-4&1&-1\\7&4&0&5&11\\2&-1&-5&0&-2\\8&5&1&6&0\end{pmatrix}\quad L^{(4)}=\begin{pmatrix}0&1&-3&2&-4\\3&0&-4&1&-1\\7&4&0&5&3\\2&-1&-5&0&-2\\8&5&1&6&0\end{pmatrix}$$

改进运行时间

不过，我们的目标并不是计算出全部的 $L^{(m)}$ 矩阵；我们所感兴趣的仅仅是矩阵 $L^{(n-1)}$。回顾不存在负权值回路的情况，等式(25.3)蕴含着对所有整数 $m\geqslant n-1$，有 $L^{(m)}=L^{(n-1)}$。如同传统的矩阵乘法满足结合率，由 EXTEND-SHORTEST-PATHS 定义的矩阵乘法也一样(参考练习 25.1-4)。因此，通过计算下列矩阵序列，我们只需计算$\lceil\lg(n-1)\rceil$个矩阵乘积就能计算出 $L^{(n-1)}$

$$L^{(1)}=W$$

$$L^{(2)}=W^2=W\cdot W$$

$$L^{(4)}=W^4=W^2\cdot W^2$$

$$L^{(8)}=W^8=W^4\cdot W^4$$

$$\vdots$$

$$L^{(2^{\lceil\lg(n-1)\rceil})}=W^{2^{\lceil\lg(n-1)\rceil}}=W^{2^{\lceil\lg(n-1)\rceil-1}}\cdot W^{2^{\lceil\lg(n-1)\rceil-1}}$$

因为 $2^{\lceil\lg(n-1)\rceil}\geqslant n-1$，最终乘积 $L^{(2^{\lceil\lg(n-1)\rceil})}$ 等于 $L^{(n-1)}$。

下面的过程利用重复平方技术计算上述矩阵序列。

625
|
626

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← rows[W]
2  L(1) ← W
3  m ← 1
4  while m<n−1
5       do L(2m) ← EXTEND-SHORTEST-PATHS(L(m), L(m))
6          m ← 2m
7  return L(m)
```

在第 4～6 行 while 循环的每一次迭代中，从 $m=1$ 开始计算 $L^{(2m)}=(L^{(m)})^2$。在每次迭代的最后，m 的值乘以 2。最后一次迭代实际通过对某个 $n-1\leqslant 2m\leqslant 2n-2$ 计算 $L^{(2m)}$，然后计算出 $L^{(n-1)}$。根据等式(25.3)，有 $L^{(2m)}=L^{(n-1)}$。下一次执行第 4 行中的测试语句时，由于 m 已被乘以 2，所以 $m\geqslant n-1$，测试失败，因此过程返回它计算出的最后一个矩阵。

因为$\lceil\lg(n-1)\rceil$个矩阵乘积中的每一个都需要 $\Theta(n^3)$时间，因此 FAST-ALL-PAIRS-SHORTEST-PATHS 的运行时间为$\Theta(n^3\lg n)$。算法中的代码是紧凑的，不包含复杂的数据结构，因此隐含于 Θ 记号中的常数是很小的。

练习

25.1-1　对图 25-2 中的带权有向图执行 SLOW-ALL-PAIRS-SHORTEST-PATHS，说明循环的每次迭代生成的矩阵。然后对 FASTER-ALL-PAIRS-SHORTEST-PATHS 做同样的工作。

25.1-2　为什么对所有的 $1 \leqslant i \leqslant n$，要求 $w_{ii}=0$？

25.1-3　在最短路径算法中使用的矩阵 $L^{(0)}$ 对应于常规矩阵乘法中的什么？

627

$$L^{(0)}=\begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & \infty \end{pmatrix}$$

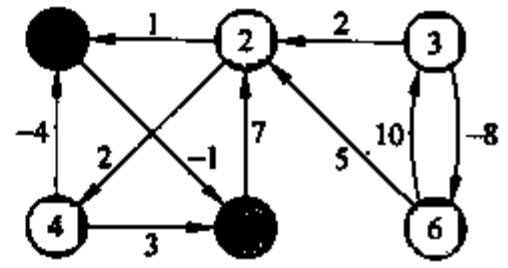


图 25-2　在练习 25.1-1，25.2-1 和练习 25.3-1 中用到的一个带权有向图

25.1-4　说明 EXTEND-SHORTEST-PATHS 所定义的矩阵乘法满足结合律。

25.1-5　说明如何把单源最短路径问题表述为矩阵和向量的乘积。描述对该乘积的计算是如何与类似 Bellman-Ford 这样的算法相一致的。

25.1-6　假设我们希望在本节的算法中得出最短路径上的顶点。说明如何在 $O(n^3)$ 时间内，根据已完成的最短路径权值的矩阵 L 计算出前趋矩阵 Π。

25.1-7　可以用与计算最短路径的权值相同的时间，计算出最短路径上的顶点。定义 $\pi_{ij}^{(m)}$ 为从 i 到 j 至多包含 m 条边的任何最小权值的路径上顶点 j 的前趋。修改 EXTEND-SHORTEST-PATHS 和 SLOW-ALL-PAIRS-SHORTEST-PATHS，使得在矩阵 $L^{(1)}$，$L^{(2)}$，…，$L^{(n-1)}$ 计算好之后，算法可以计算出矩阵 $\Pi^{(1)}$，$\Pi^{(2)}$，…，$\Pi^{(n-1)}$。

25.1-8　如上所述，FASTER-ALL-PAIRS-SHORTEST-PATHS 过程需要我们保存 $\lceil \lg(n-1) \rceil$ 个矩阵，每个矩阵包含 n^2 个元素，总的空间需求为 $\Theta(n^2 \lg n)$。修改这个过程，使其仅使用两个 $n \times n$ 个矩阵，而且需要的空间为 $\Theta(n^2)$。

25.1-9　修改 FASTER-ALL-PAIRS-SHORTEST-PATHS，使其能检测出图中是否存在权值为负的回路。

628　**25.1-10**　写出一个有效的算法来计算图中最短的负权值回路的长度（即所包含的边数）。

25.2　Floyd-Warshall 算法

在本节中，我们将采用另一种动态规划方案，来解决在一个有向图 $G=(V, E)$ 上每对顶点间的最短路径问题。所产生的算法称为 Floyd-Warshall 算法，其运行时间为 $\Theta(V^3)$。和前面一样，允许存在权值为负的边，但我们假设不存在权值为负的回路。如在第 25.1 节中一样，我们将按动态规划的过程来设计算法。在学习完结果算法之后，我们将提供一个类似的方法来找出有向图的传递闭包。

最短路径的结构

在 Floyd-Warshall 算法中，我们利用最短路径结构中的另一个特征，它不同于基于矩阵乘法的每对顶点算法中所用到的特征。该算法考虑最短路径上的中间顶点，其中简单路径 $p=\langle v_1, v_2, \cdots, v_l \rangle$ 上的中间顶点是除 v_1 和 v_l 以外 p 上的任何一个顶点，即任何属于集合 $\{v_2, v_3, \cdots, v_{l-1}\}$ 的顶点。

Floyd-Warshall 算法主要基于以下观察。设 G 的顶点为 $V=\{1, 2, \cdots, n\}$，对某个 k 考虑顶

点的一个子集$\{1, 2, \cdots, k\}$。对任意一对顶点 $i, j \in V$，考察从 i 到 j 且中间顶点皆属于集合$\{1, 2, \cdots, k\}$的所有路径，设 p 是其中的一条最小权值路径。(路径 p 是简单的。) Floyd-Warshall 算法利用了路径 p 与 i 到 j 之间的最短路径(所有中间顶点都属于集合$\{1, 2, \cdots, k-1\}$)之间的联系。这一联系依赖于 k 是否是路径 p 上的一个中间顶点。

- 如果 k 不是路径 p 的中间顶点，则 p 的所有中间顶点皆在集合$\{1, 2, \cdots, k-1\}$中。因此从顶点 i 到顶点 j 且满足所有中间顶点皆属于集合$\{1, 2, \cdots, k-1\}$的一条最短路径，也同样是从 i 到 j 且满足所有中间顶点皆属于集合$\{1, 2, \cdots, k\}$的一条最短路径。
- 如果 k 是路径 p 的中间顶点，那么可将 p 分解为 $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$，如图 25-3 所示。由引理 24.1 可知，$p_1$ 是从 i 到 k 的一条最短路径，且其所有中间顶点均属于集合$\{1, 2, \cdots, k\}$。因为顶点 k 不是路径 p_1 上的一个中间顶点，所以 p_1 是从 i 到 k 的一条最短路径，且其所有中间顶点均属于集合$\{1, 2, \cdots, k-1\}$。类似地，p_2 是从顶点 k 到顶点 j 的一条最短路径，且其所有中间顶点均属于集合$\{1, 2, \cdots, k-1\}$。

629

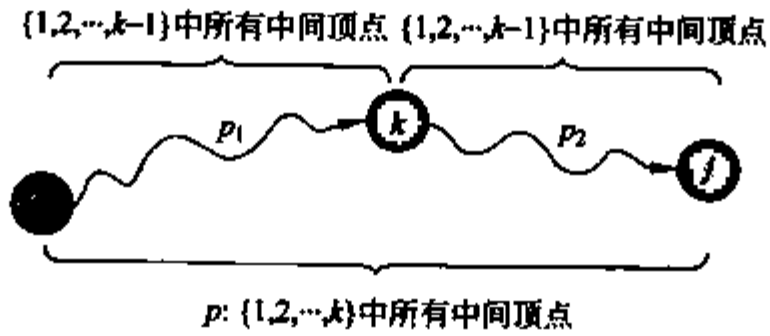


图 25-3　路径 p 是从顶点 i 到顶点 j 的一条最短路径，k 是 p 上编号最高的中间顶点。路径 p 中从顶点 i 到顶点 k 的部分即路径 p_1，其所有中间顶点皆属于集合$\{1, 2, \cdots, k-1\}$。从顶点 k 到顶点 j 的路径 p_2 也具有同样的性质

解决每对顶点间最短路径问题的一个递归解

基于上述观察，我们定义一个最短路径估计的递归公式，它不同于 25.1 节中给出的那个公式。令 $d_{ij}^{(k)}$ 为从顶点 i 到顶点 j、且满足所有中间顶点皆属于集合$\{1, 2, \cdots, k\}$的一条最短路径的权值。当$k=0$ 时，从顶点 i 到顶点 j 的路径中，没有编号大于 0 的中间顶点；亦即，根本不存在中间顶点。这样的路径至多包含一条边，因此 $d_{ij}^{(0)} = w_{ij}$。根据上述讨论，我们用下式给出一个递归式

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{如果 } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{如果 } k \geqslant 1 \end{cases} \tag{25.5}$$

因为对于任意路径，所有的中间顶点都在集合$\{1, 2, \cdots, n\}$内，矩阵 $D^{(n)} = (d_{ij}^{(n)})$给出了最终解答：对所有的 $i, j \in V$，有 $d_{ij}^{(n)} = \delta(i, j)$。

自底向上计算最短路径的权值

基于递归式(25.5)，下面的自底向上的过程按 k 值递增顺序计算 $d_{ij}^{(k)}$ 的值。它的输入是等式(25.1)中定义的 $n \times n$ 矩阵 W。过程返回最短路径权值的矩阵 $D^{(n)}$。

```
FLOYD-WARSHALL(W)
1  n ← rows[W]
2  D(0) ← W
3  for k ← 1 to n
4       do for i ← 1 to n
5            do for j ← 1 to n
```

6　　do $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

630　7　return $D^{(n)}$

图 25-4 显示了利用 Floyd-Warshall 算法，对图 25-1 中的图形计算出的矩阵序列 $D^{(k)}$。

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

图 25-4 利用 Floyd-Warshall 算法对图 25-1 中的图形计算出的矩阵 $D^{(k)}$ 和 $\Pi^{(k)}$

Floyd-Warshall 算法的运行时间是由第 3～6 行的三重嵌套 for 循环所决定的。每次执行第 6 行花费 $O(1)$时间，因此算法的运行时间为 $\Theta(n^3)$。正如 25.1 节最后的算法一样，其代码是紧凑

631 的，而且不包含复杂的数据结构，因此隐含于 Θ 记号中的常数是很小的。因此，即便对于中等规模的输入图来说，Floyd-Warshall 算法仍然是相当实用的。

构造一条最短路径

在 Floyd-Warshall 算法中存在大量不同的方法来建立最短路径。一种途径是计算最短路径权值的矩阵 D，然后根据矩阵 D 构造前驱矩阵 Π。这一方法可以在 $O(n^3)$时间内实现(参见练习 25.1-6)。给定前趋矩阵 Π，可以使用过程 PRINT-ALL-PAIRS-SHORTEST-PATH 来输出一条给定最短路径上的顶点。

可以像 Floyd-Warshall 算法计算矩阵 $D^{(k)}$ 那样，“联机”计算前趋矩阵 Π。具体来说，我们计算一个矩阵序列 $\Pi^{(1)}$，$\Pi^{(2)}$，…，$\Pi^{(n)}$，其中 $\Pi = \Pi^{(n)}$，并定义 $\pi_{ij}^{(k)}$ 为从 i 出发的一条最短路径上顶点 j 的前趋，而这条路径上的中间顶点属于集合$\{1, 2, \cdots, k\}$。

我们可以给出 $\pi_{ij}^{(k)}$ 的递归公式。当 $k=0$ 时，从 i 到 j 的最短路径根本不存在中间顶点。因此，

$$\pi_{ij}^{(0)}=\begin{cases}\text{NIT} & \text{如果 } i=j \text{ 或 } w_{ij}=\infty\\ i & \text{如果 } i\neq j \text{ 和 } w_{ij}<\infty\end{cases}\tag{25.6}$$

对于 $k\geqslant 1$，如果取路径 $i\rightsquigarrow k\rightsquigarrow j$，其中 $k\neq j$，则我们所选择的 j 的前趋相同于我们在从 k 出发，且满足所有中间顶点都属于集合$\{1, 2, \cdots, k-1\}$的最短路径上所选择的 j 的前驱。否则，我们所选择的 j 的前趋就相同于我们在从 i 出发，且满足所有中间顶点都属于集合$\{1, 2, \cdots, k-1\}$的最短路径上所选择的 j 的前趋。正式地，对于 $k\geqslant 1$，

$$\pi_{ij}^{(k)}=\begin{cases}\pi_{ij}^{(k-1)} & \text{如果 } d_{ij}^{(k-1)}\leqslant d_{ij}^{(k-1)}+d_{kj}^{(k-1)}\\ \pi_{ij}^{(k-1)} & \text{如果 } d_{ij}^{(k-1)}>d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\end{cases}\tag{25.7}$$

我们把矩阵 $\Pi^{(k)}$ 的计算合并到 FLOYD-WARSHALL 过程中，这个工作留作练习 25.2-3。图 25-4 显示了由此产生的算法对图 25-1 所示的图计算所得的 $\Pi^{(k)}$ 矩阵序列。练习还提出了更难的任务，即证明前趋子图 $G_{\pi,i}$ 是一棵以 i 为根的最短路径树。另外一种重构最短路径的方法在练习 25.2-7 中给出。

有向图的传递闭包

已知一有向图 $G=(V, E)$，顶点集合 $V=\{1, 2, \cdots, n\}$，我们可能希望确定对所有顶点对 $i, j\in V$，图 G 中是否都存在一条从 i 到 j 的路径。G 的传递闭包定义为图 $G^*=(V, E^*)$，其中

$E^*=\{(i, j)$：图 G 中存在一条从 i 到 j 的路径$\}$。 632

在 $\Theta(n^3)$ 时间内计算出图的传递闭包的一种方法为对 E 中每条边赋以权值 1，然后运行 Floyd-Warshall 算法。如果从顶点 i 到顶点 j 存在一条路径，则 $d_{ij}<n$。否则，有 $d_{ij}=\infty$。

另外还有一个类似的方法，可以在 $\Theta(n^3)$ 时间内计算出图 G 的传递闭包，它在实际中可以节省时空需求。该方法要求把 Floyd-Warshall 算法中的 min 和＋算术运算操作，用相应的逻辑运算 $\vee$（逻辑 OR）和 $\wedge$（逻辑 AND）来代替。对 $i, j, k=1, 2, \cdots, n$，如果图 G 中从顶点 i 到顶点 j 存在一条通路，且其所有中间顶点均属于集合$\{1, 2, \cdots, k\}$，则定义 $t_{ij}^{(k)}$ 为 1，否则 $t_{ij}^{(k)}$ 为 0。我们把边(i, j)加入 E^* 中当且仅当 $t_{ij}^{(n)}=1$。通过这种方式我们来构造传递闭包 $G^*=(V, E^*)$。与递归式(25.5)类似，$t_{ij}^{(k)}$ 的递归定义为

$$t_{ij}^{(0)}=\begin{cases}0 & \text{如果 } i\neq j \text{ 和 } (i,j)\notin E\\ 1 & \text{如果 } i=j \text{ 或 } (i,j)\in E\end{cases}$$

且对 $k\geqslant 1$，

$$t_{ij}^{(k)}=t_{ij}^{(k-1)}\vee(t_{ik}^{(k-1)}\wedge t_{kj}^{(k-1)})\tag{25.8}$$

正如在 Floyd-Warshall 算法中那样，我们按 k 的递增顺序来计算矩阵 $T^{(k)}=(t_{ij}^{(k)})$

```
TRANSITIVE-CLOSURE(G)
1  n ← |V[G]|
2  for i ← 1 to n
3       do for j ← 1 to n
4            do if i=j or (i, j)∈E[G]
5                 then t_ij^(0) ← 1
6                 else t_ij^(0) ← 0
7  for k ← 1 to n
8       do for i ← 1 to n
```

```
9          do for j ← 1 to n
10             do t_ij^(k) ← t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
11 return T^(n)
```

图 25-5 显示了过程 TRANSITIVE-CLOSURE 对一个样图计算所得的矩阵 $T^{(k)}$。过程 TRANSITIVE-CLOSURE 的运行时间与 Floyd-Warshall 算法一样，也是 $\Theta(n^3)$。但是在某些计算机上，对单位的值，
633 逻辑操作的执行速度快于对整数字长数据的算术运算操作。再者，因为直接的传递闭包算法仅使用布尔值而不是整数值，其空间要求与 Floyd-Warshall 算法相比，也要小一个与计算机字长相对应的因子。

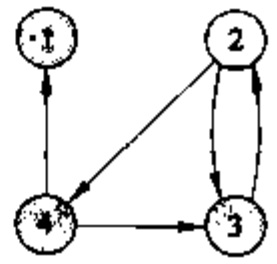


图 25-5　由传递闭包算法计算出的有向图和矩阵 $T^{(k)}$

$$T^{(0)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&0\\1&0&1&1\end{pmatrix}\quad T^{(1)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&0\\1&0&1&1\end{pmatrix}\quad T^{(2)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&1\\1&0&1&1\end{pmatrix}$$

$$T^{(3)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&1\\1&1&1&1\end{pmatrix}\quad T^{(4)}=\begin{pmatrix}1&0&0&0\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{pmatrix}$$

练习

25.2-1　对图 25-2 所示的带权有向图运行 Floyd-Warshall 算法。写出外层循环中每次迭代所生成的矩阵 $D^{(k)}$。

25.2-2　说明如何利用 25.1 节的技术计算传递闭包。

25.2-3　根据等式(25.6)和等式(25.7)，修改过程 FLOYD-WARSHALL 以使其包含对矩阵 $\Pi^{(k)}$ 的计算。严格证明对所有的 $i\in V$，前趋子图 $G_{\pi,i}$ 是以 i 为根的一棵最短路径树。(提示：为了证明 $G_{\pi,i}$ 是无回路图，首先根据 $\pi_{ij}^{(k)}$ 的定义来证明 $\pi_{ij}^{(k)}=l$ 蕴含着 $d_{ij}^{(k)}\geqslant d_{il}^{(k)}+w_{lj}$。然后改写引理 24.16 的证明。)

25.2-4　如上所见，由于我们要计算 $d_{ij}^{(k)}$，i，j，$k=1$，2，$\cdots$，n，因此 Floyd-Warshall 算法的空间要求为 $\Theta(n^3)$。证明：仅仅去掉所有上标所得的如下过程是正确的，因此仅需要
634 $\Theta(n^2)$的空间。

```
FLOYD-WARSHALL'(W)
1 n ← rows[W]
2 D ← W
3 for k ← 1 to n
4     do for i ← 1 to n
5            do for j ← 1 to n
6                   do d_ij ← min(d_ij , d_ik + d_kj)
7 return D
```

25.2-5　假设我们修改等式(26.7)中等号的处理方式

$$\pi_{ij}^{(k)}=\begin{cases}\pi_{ij}^{(k-1)} & \text{如果 } d_{ij}^{(k-1)}<d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\\ \pi_{kj}^{(k-1)} & \text{如果 } d_{ij}^{(k-1)}\geqslant d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\end{cases}$$

选择它作为前趋矩阵 Π 的定义是否正确？

25.2-6　如何利用 Floyd-Warshall 算法的输出来检测是否存在权为负的回路？

25.2-7 Floyd-Warshall 算法中另一种重构最短路径的方法利用了 $\phi_{ij}^{(k)}$ 的值，i，j，$k=1$，2，…，n，其中 $\phi_{ij}^{(k)}$ 是从 i 到 j 的最短路径中，具有最高编号的中间顶点，而且这条路径的所有中间顶点都属于集合$\{1, 2, \cdots, k\}$。请给出 $\phi_{ij}^{(k)}$ 的递归公式，修改过程 FLOYD-WARSHALL 以计算 $\phi_{ij}^{(k)}$ 的值，并用矩阵 $\Phi=(\phi_{ij}^{(n)})$ 作为输入来重写过程 PRINT-ALL-PAIRS-SHORTEST-PATH。矩阵 Φ 与第 15.2 节中的矩阵链乘法问题中的表 s 有何相似之处？

25.2-8 写出一运行时间为 $O(VE)$ 的算法，来计算有向图 $G=(V, E)$ 的传递闭包。

25.2-9 假定一个有向无环图的传递闭包可以在 $f(|V|, |E|)$ 时间内计算，其中 f 是 $|V|$ 和 $|E|$ 的单调递增函数。证明：计算一般有向图 $G=(V, E)$ 的传递闭包 $G^*=(V, E^*)$ 的时间为 $f(|V|, |E|)+O(V+E^*)$。 635

25.3　稀疏图上的 Johnson 算法

Johnson 算法可在 $O(V^2 \lg V+VE)$ 时间内，求出每对顶点间的最短路径。对于稀疏图，该算法在渐近意义上要好于矩阵的重复平方或 Floyd-Warshall 算法。算法执行后，返回一个关于每对顶点间最短路径的权值的矩阵，或者报告输入图中存在一个负权值的回路。Johnson 算法把第 24 章中描述的 Dijkstra 算法和 Bellman-Ford 算法作为其子程序。

Johnson 算法运用了*重赋权技术*，其执行方式如下。如果图 $G=(V, E)$ 中所有边的权 w 均为非负，则把每对顶点依次作为源点来执行 Dijkstra 算法，就可以找出每对顶点间的最短路径；利用斐波那契堆最小优先队列，则该算法的运行时间为 $O(V^2 \lg V+VE)$。如果 G 含有负权边但不含有负权的回路，就只计算一个新的负权边的集合，而这可以采用相同的方法。这个新的边权值 $\hat{w}$ 的集合必须满足两个重要性质：

1）对所有顶点对 u，$v \in V$，路径 p 是利用加权函数 w 从 u 到 v 的一条最短路径，当且仅当 p 也是利用加权函数 $\hat{w}$ 从 u 到 v 的一条最短路径。

2）对于所有的边 (u, v)，新的权 $\hat{w}(u, v)$ 是非负的。

稍后我们将看到，为确定新的加权函数 $\hat{w}$ 而对 G 进行的预处理可在 $O(VE)$ 时间内完成。

通过重赋权值保持最短路径

如下面的引理所述，对边重新赋权以满足上面第一个性质是比较容易的。我们用 δ 表示根据加权函数 w 导出的最短路径的权，而用 $\hat{\delta}$ 表示根据加权函数 $\hat{w}$ 导出的最短路径的权。

引理 25.1（重赋权值不会改变最短路径） 已知带权有向图 $G=(V, E)$，加权函数为 $w: E \to \mathbf{R}$，设 $h: V \to \mathbf{R}$ 是将顶点映射到实数的任意函数。对每条边 $(u, v) \in E$，定义

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v) \tag{25.9}$$

令 $p=\langle v_0, v_1, \cdots, v_k\rangle$ 为从顶点 v_0 到顶点 v_k 的任意一条路径。则 p 是利用加权函数 w 从 v_0 到 v_k 的一条最短路径，当且仅当 p 也是利用加权函数 $\hat{w}$ 的一条最短路径。亦即，$w(p)=\delta(v_0, v_k)$ 当且仅当 $\hat{w}(p)=\hat{\delta}(v_0, v_k)$。另外，使用加权函数 w 时，G 中存在一条负权的回路，当且仅当使用加权函数 $\hat{w}$ 时，G 中存在一条负权的回路。 636

证明：首先证明

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k) \tag{25.10}$$

有

$$\hat{w}(p) = \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i)$$

$$
\begin{aligned}
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \quad \text{（由于求和叠缩）} \\
&= w(p) + h(v_0) - h(v_k)
\end{aligned}
$$

因此，从 v_0 到 v_k 的任意路径 p 上，都有 $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$。如果从 v_0 到 v_k 之间，存在着一条路径短于使用加权函数 w 的路径，则它也短于使用 $\hat{w}$ 的路径。因此，$w(p) = \delta(v_0, v_k)$ 当且仅当 $\hat{w}(p) = \hat{\delta}(v_0, v_k)$。

最后，要证明根据加权函数 w，G 中包含一负权回路，当且仅当根据加权函数 $\hat{w}$，G 中包含一负权回路。考虑任意回路 $c = \langle v_0, v_1, \cdots, v_k \rangle$，其中 $v_0 = v_k$。根据等式(25.10)，

$$\hat{w}(c) = w(c) + h(v_0) - h(v_k) = w(c)$$

因此根据 w，c 的权为负当且仅当根据 $\hat{w}$，它也有负的权。 ■

通过重赋权产生非负的权

我们的下一个目标是保证第二条性质成立：我们希望对于所有边 $(u, v) \in V$，$\hat{w}(u, v)$ 的值非负。给定一带权有向图 $G = (V, E)$，加权函数为 w：$E \to \mathbf{R}$，据此构造一个新图 $G' = (V', E')$，其中对某个新顶点 $s \notin V$，$V' = V \cup \{s\}$，$E' = E \cup \{(s, v) : v \in V\}$，扩展加权函数 w，使得对所有 $v \in V$，有 $w(s, v) = 0$。注意因为不存在进入顶点 s 的边，所以除了以 s 作为源点的路径，G' 中不存在包含 s 的其他最短路径。再者，G' 不包含负权回路，当且仅当 G 不包含负权回路。图 25-6a 显示了图 G'，它相应于如图 25-1 所示的图 G。

637　现在假设 G 和 G' 都不含负权回路。定义对所有 $v \in V'$，$h(v) = \delta(s, v)$。由引理 24.10 的三角不等式，对所有的边 $(u, v) \in E'$，有 $h(v) \leqslant h(u) + w(u, v)$。因此，如果我们根据等式(25.9)定义新权 $\hat{w}$，有 $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geqslant 0$，这样就满足了第二条性质。图 25-6b 显示了对图 25-6a 中的图重赋权后所得的图 G'。

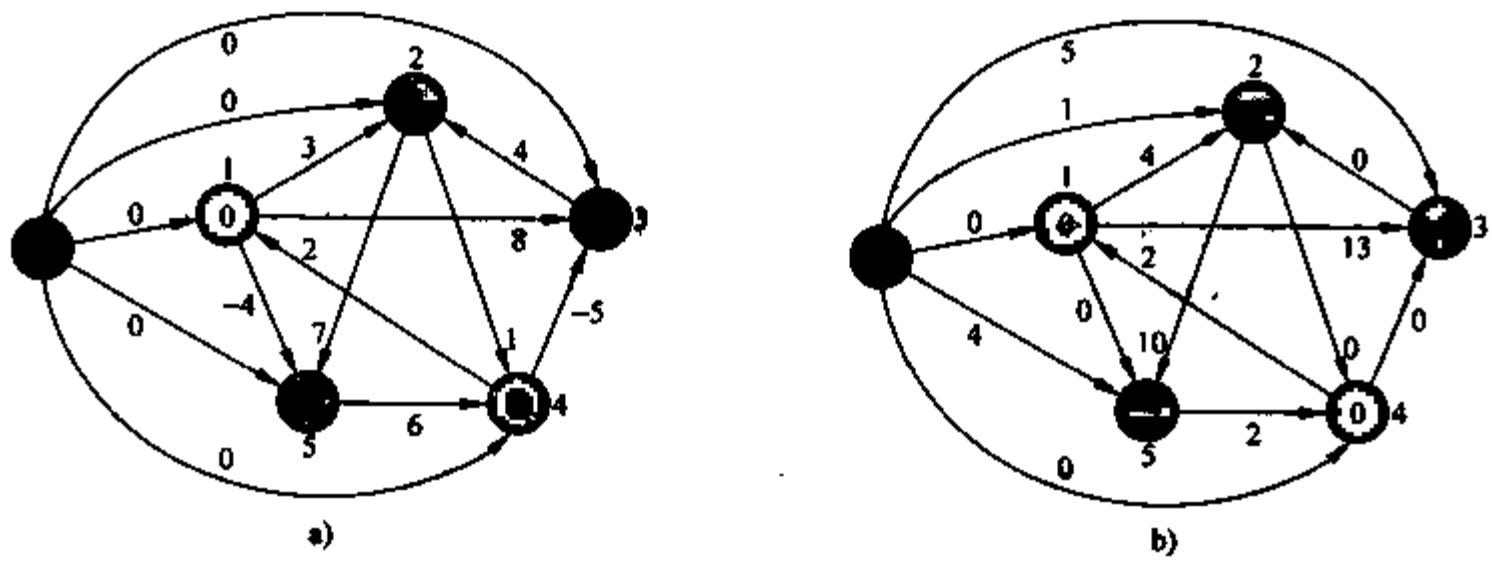


图 25-6　Johnson 每对顶点最短路径算法在图 25-1 所示的图上的运行过程。a)用原始加权函数 w 的图 G'。新顶点 s 是黑色的。在每个顶点 v 内的是 $h(v) = \delta(s, v)$。b)每条边 (u, v) 用加权函数 $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ 重赋权。c)～g)用加权函数 $\hat{w}$ 在图 G 的每个顶点上执行 Dijkstra 算法的结果。在每个子图中，源顶点 u 是黑色的，而阴影边是在算法计算出的最短路径树内。每个顶点 v 内包含值 $\hat{\delta}(u, v)$ 和 $\delta(u, v)$，用一个斜线来分隔。$d_{uv} = \delta(u, v)$ 的值等于 $\hat{\delta}(u, v) + h(v) - h(u)$

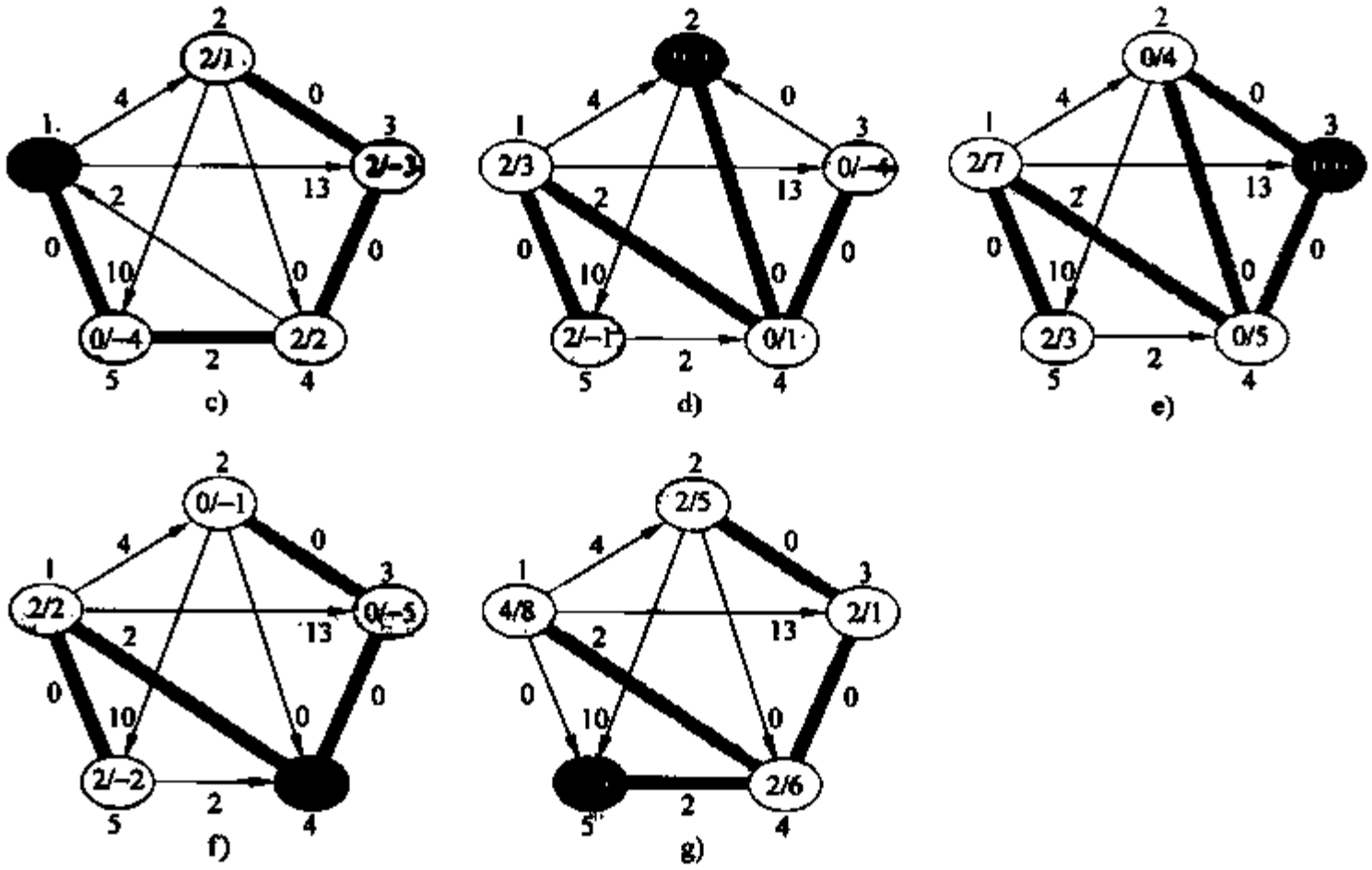


图 25-6　(续)

计算每对顶点间的最短路径

在计算每对顶点间最短路径的 Johnson 算法中，把 Bellman-Ford 算法(24.1 节)和 Dijkstra 算法(24.3 节)作为其子程序。它假设图的边用邻接表形式存储。算法返回通常的 $|V| \times |V|$ 矩阵 $D=d_{ij}$，其中 $d_{ij}=\delta(i, j)$，或者报告输入图中存在一负权的回路。如通常的每对顶点最短路径算法一样，我们假设顶点的编号为从 1 到 $|V|$。

```
JOHNSON(G)
1  compute G', where V[G'] = V[G] ∪ {s},
           E[G'] = E[G] ∪ {(s, v): v ∈ V[G]}, and
           w(s, v) = 0 for all v ∈ V[G]
2  if BELLMAN-FORD(G', w, s) = FALSE
3     then print "the input graph contains a negative-weight cycle"
4     else for each vertex v ∈ V[G']
5             do set h(v) to the value of δ(s, v)
                        computed by the Bellman-Ford algorithm
6          for each edge (u, v) ∈ E[G']
7             do ŵ(u, v) ← w(u, v) + h(u) − h(v)
8          for each vertex u ∈ V[G]
9             do run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ V[G]
10               for each vertex v ∈ V[G]
11                  do d_uv ← δ̂(u, v) + h(v) − h(u)
12         return D
```

这段代码实现了我们前面说明的操作。第 1 行产生 G'。第 2 行对加权函数为 w 源点为 s 的图 G'

执行 Bellman-Ford 算法。如果 G'(因此 G)包含负权回路，第 3 行将报告这一问题。第 4～11 行假定 G'不包含负权回路。第 4～5 行对所有的 $v \in V$，把 $h(v)$置为由 Bellman-Ford 算法计算出来的最短路径的权$\delta(s, v)$。第 6～7 行计算新的权$\hat{w}$。对每对顶点 $u, v \in V$，第 8～11 行的 for 循环把 V 中的每个顶点轮流作为源点调用 Dijkstra 算法，来计算最短路径的权$\hat{\delta}(u, v)$。第 11 行把用等式(25.10)计算出的正确的最短路径的权 $\delta(u, v)$存入相应的矩阵元素 d_{uv}。最后，第 12 行返回计算好的矩阵 D。图 25-6 显示了 Johnson 算法的执行过程。

638
|
639

如果采用斐波那契堆来实现 Dijkstra 算法中的最小优先队列，则 Johnson 算法的运行时间为 $O(V^2 \lg V + VE)$。更简单的二叉堆实现，则可以得到 $O(VE \lg V)$的运行时间。对于稀疏图来说，这在渐近意义上仍然比 Floyd-Warshall 算法快。

练习

25.3-1 利用 Johnson 算法，找出图 25-2 所示图中的每对顶点间最短路径。说明由算法计算出的 h 和$\hat{w}$的值。

25.3-2 把新顶点 s 加入到 V 中得到 V'，其意图是什么？

25.3-3 假定对所有边$(u, v) \in E$，有 $w(u, v) \geqslant 0$。那么加权函数 w 和$\hat{w}$有什么关系？

25.3-4 GreenStreet 教授声称，存在一个比 Johnson 算法中所使用的更简单的重赋权方法。令 $w^* = \min_{(u,v) \in E} \{w(u, v)\}$，只要对所有边$(u, v) \in E$ 定义$\tilde{w}(u, v) = w(u, v) - w^*$。他的重赋权方法错在什么地方？

25.3-5 假设在有向图 G 上用加权函数 w 运行 Johnson 算法。证明：如果 G 包含 0 权的回路 c，则对 c 中的每个边(u, v)，$\hat{w}(u, v) = 0$。

25.3-6 Michener 教授宣称，在 JOHNSON 算法的第 1 行中，不需要建立一个新的源顶点。相反，他主张可以使用 $G' = G$，而且设 s 为 $V[G]$中的任意顶点。给出一个带权有向图 G 的例子，使得当将这位教授的想法用到 JOHNSON 算法中后，会导致错误的答案。然后证明如果 G 是强连通的(每个顶点都可以和其他每个顶点相连接)，那么采用这位教授修改的 JOHNSON 算法的话，所返回的结果是正确的。

640

思考题

25-1 动态图的传递闭包

假设对有向图 $G = (V, E)$，当我们插入边到 E 中时希望保持传递闭包的正确性，即在插入每条边后，希望对迄今为止已插入边的传递闭包进行更新。假设图 G 开始时不含任何边，并且传递闭包用布尔矩阵来表示。

a)说明当插入一条新边到图 $G = (V, E)$时，如何能在 $O(V^2)$时间内对其传递闭包 $G^* = (V, E^*)$进行更新？

b)举出一图 G 和边 e 的例子，当 e 被插入到 G 中后，需要 $\Omega(V^2)$的运行时间对 G 的传递闭包进行更新。

c)描述一个有效的算法，使得在图中插入一条边时，算法能对图的传递闭包进行更新。对任意由 n 次插入操作组成的序列，所给出的算法的运行时间应为 $\sum_{i=1}^{n} t_i = O(V^3)$，其中 t_i 为第 i 条边被插入时更新传递闭包所需的时间。证明你的算法的运行时间在这个时

间界内。

25-2　ε 稠密图中的最短路径

在图 $G=(V, E)$中，如果对某常数 $0<\varepsilon\leqslant 1$，有 $|E|=\Theta(V^{1+\varepsilon})$，则说 G 是 ε 稠密的。通过在 ε 稠密图上的最短路径算法中应用 d 叉最小堆(参见思考题 6-2)，能使算法的运行时间相当于基于斐波那契堆的算法的运行时间，同时无需引入后者所使用的复杂数据结构。

a)在 d 叉最小堆中，INSERT，EXTRACT-MIN 和 DECREASE-KEY 的渐近运行时间是多少？用 d 和元素个数 n 的函数表示。如果选择 $d=\Theta(n^{\alpha})$，α 为 $0<\alpha\leqslant 1$ 的一个常数，则它们的运行时间又是多少？试将这些运行时间与在斐波那契堆上执行这些操作的平摊代价做一比较。

b)说明如何在 $O(E)$时间内，对一个不含负权边的 ε 稠密有向图 $G=(V, E)$，计算出从某单源顶点出发的最短路径。(提示：把 d 看作 ε 的函数。)

c)说明如何在 $O(VE)$的运行时间内，对一个不含负权边的 ε 稠密有向图 $G=(V, E)$，解决其每对顶点间的最短路径问题。

641

d)说明如何在 $O(VE)$的运行时间内，对一个可能包含负权边、但不包含负权回路的 ε 稠密有向图 $G=(V, E)$，解决其每对顶点间的最短路径问题。

本章注记

Lawler[196]详细讨论了每对顶点最短路径问题，但没有分析稀疏图的解。他将矩阵乘法归功于多人的努力。Floyd-Warshall 算法由 Floyd[89]而来，他以 Warshall[308]的描述如何计算布尔矩阵的传递闭包的理论为基础。Johnson 算法来自[168]。

一些研究人员已经给出通过矩阵乘法来计算最短路径的改进算法。Fredman[95]说明每对顶点间最短路径问题可以在边权值的总和之间，使用 $O(V^{5/2})$次比较来解决，而且得到一个在 $O(V^3(\lg\lg V/\lg V)^{1/3})$时间内执行的算法，比 Floyd-Warshall 算法的执行时间稍微好一点。另一类的研究显示快速矩阵乘法(参见第 28 章的章节注记)可以应用到每对顶点间最短路径问题。令 $O(n^{\omega})$为快速算法作用于 $n\times n$ 矩阵相乘所需的运行时间；目前 $\omega<2.376$ [70]。在无向无权图上，Galil 和 Margalit[105，106]以及 Seidel[270]设计的算法用$(V^{\omega}p(V))$时间解决每对顶点间最短路径问题，其中 $p(n)$表示一个以 n 的多项式对数为界的特殊函数。在稠密图中，这些算法比执行 $|V|$ 个广度优先搜索所需的时间$O(VE)$要快。还有一些研究人员扩展了这些结果，来给出解无向图中每对顶点间最短路径问题的算法，其中边的权值是在范围$\{1, 2, \cdots, W\}$内的整数。渐近最快的这种算法，是由 Shoshan 和 Zwick[278]而来，在 $O(WV^{\omega}p(VW))$时间内执行。

Karger，Koller 和 Phillips[170]，以及 McGeoch[215]独立地给出了一个时间界，它依赖于 E^*，即 E 内某条最短路径中的边的集合。已知一个非负边权值的图，他们的算法在 $O(VE^*+V^2\lg V)$时间内运行，而且当 $|E^*|=o(E)$时，它是对运行 $|V|$ 次 Dijkstra 算法的改进。

Aho，Hopcroft 和 Ullman[5]定义了一个叫做“闭合半环”(closed semiring)的代数结构，来作为在有向图中解决路径问题的一般框架。Floyd-Warshall 算法和 25.2 节的传递闭包算法都是基于闭合半环的每对顶点算法的实例。Maggs 和 Plotkin[208]说明了如何利用一个闭合半环来寻找最小生成树。

642

第 26 章　最　大　流

为了求从一点到另一点的最短路径，我们可以把公路地图模型化为有向图。同样，我们可以把一个有向图理解为一个“流网络”(flow network)，并运用它来回答有关物流方面的问题。设想某物质从产生它的源点经过一个系统，流向消耗该物质的汇点(sink)这样一种过程。源点以固定速度产生该物质，而汇点则用同样的速度消耗该物质。从直观上看，系统中任何一点的物质的“流”为该物质在系统中运行的速度。我们可以应用流网络来模型化流经管道的液体、通过装配线的部件、电网中的电流或通讯网络传送的信息等等。

流网络中的每条有向边可以被认为是传输物质的管道。每个管道都有一个固定的容量，可以看作是物质能够流经该管道的最大速度，例如，经过某一管道的每小时 200 加仑的液体，或通过一段电线的 20 安培的电流。顶点是管道间的连接点，除了源点和汇点以外，物质只流经这些顶点，而不聚集在顶点中。换句话说，物质进入某顶点的速度必须等于离开该顶点的速度。我们称这一特性为“流守恒”(flow conservation)。当物质是电流时，流守恒与基尔霍夫电流定律等价。

最大流问题是关于流网络的最简单的问题。它提出这样的问题：在不违背容量限制的条件下，把物质从源点传输到汇点的最大速率是多少？我们在本章中将会看到，这个问题可以由有效的算法来解决。此外，最大流算法中使用的基本技术适合于解决其他的网络流问题。

本章提出解决最大流问题的两种一般方法。26.1 节对流网络和流概念以及最大流问题给出
643 形式化定义。26.2 节描述解决最大流问题的 Ford 和 Fulkerson 经典方法。26.3 节给出这一方法的一种应用，即在无向二分图中寻找最大匹配。26.4 节阐述压入与重标记方法，该方法构成网络流问题的很多快速算法的基础。26.5 节论述重标记与前移算法，该算法是压入与重标记方法的一个特定实现，其运行时间为 $O(V^3)$。虽然这一算法并非是目前最快的算法，但它说明渐近意义上最快的算法中应用的某些技术。在实际应用中，该算法也是非常有效的。

26.1　流网络

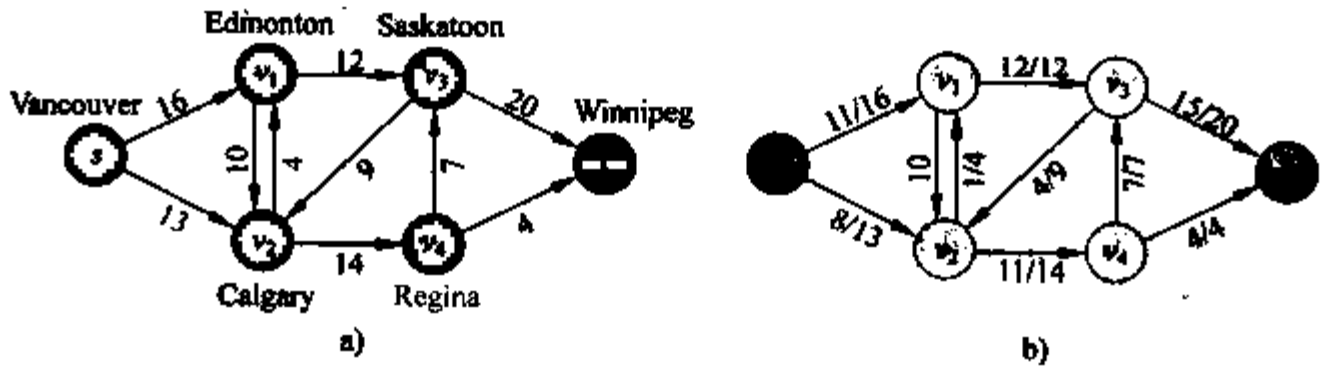
在本节中，我们将给出流网络的图论定义，讨论其性质，并精确地定义最大流问题。我们还要引入几个有用的记号。

流网络和流

流网络 $G=(V, E)$ 是一个有向图，其中每条边 $(u, v)\in E$ 均有一非负容量 $c(u, v)\geqslant 0$。如果 $(u, v)\notin E$，则假定 $c(u, v)=0$。流网络中有两个特别的顶点：源点 s 和汇点 t。为了方便起见，假定每个顶点均处于从源点到汇点的某条路径上，就是说，对每个顶点 $v\in V$，存在一条路径 $s\rightsquigarrow v\rightsquigarrow t$。因此，图 G 为连通图，且 $|E|\geqslant |V|-1$。图 26-1 展示一个流网络的实例。

现在对流作出更为形式化的定义。设 $G=(V, E)$ 是一个流网络，其容量函数为 c。设 s 为网络的源点，t 为汇点。G 的流是一个实值函数 $f: V\times V\rightarrow \mathbf{R}$，且满足下列三个性质：

容量限制：对所有 $u, v\in V$，要求 $f(u, v)\leqslant c(u, v)$。



a)　　b)

图 26-1　a)关于 Lucky Puck 公司卡车运输问题的流网络 $G=(V, E)$。Vancouver 工厂是源点 s，而 Winnipeg 仓库是汇点 t。冰球经由中间城市运送，但是每天只能有 $c(u, v)$箱从城市 u 运到城市 v。每条边都标记它的容量。b)G 中的流 f，其值 $|f|=19$。图中只显示正网络流。如果 $f(u, v)>0$，则标记边(u, v)为 $f(u, v)/c(u, v)$(斜杠记号仅仅用来区分流和容量，不表示相除)。如果 $f(u, v)\leqslant 0$，边(u, v)只标记它的容量

反对称性：对所有 $u, v\in V$，要求 $f(u, v)=-f(v, u)$。

流守恒性：对所有 $u\in V-\{s, t\}$，要求

$$\sum_{v\in V} f(u,v) = 0$$

$f(u, v)$称为从顶点 u 到顶点 v 的流，它可以为正，为零，也可以为负。流 f 的值定义为

$$|f| = \sum_{v\in V} f(s,v) \tag{26.1}$$

即，从源点出发的总流(这里，记号 $|\cdot|$ 表示流的值，并不表示绝对值或势)。在最大流问题中，给出一个具有源点 s 和汇点 t 的流网络 G，希望找出从 s 到 t 的最大值流。

在看一个有关网络流问题的例子前，先来看一下流的三个性质。容量限制只说明从一个顶点到另一顶点的网络流不能超过设定的容量。反对称性说明从顶点 u 到顶点 v 的流是其反向流求负所得。流守恒性说明从非源点或非汇点的顶点出发的总网络流为 0。根据反对称性，对所有 $v\in V-\{s, t\}$，可以把流守恒性重写为

$$\sum_{u\in V} f(u,v) = 0$$

亦即，进入一个顶点的总流为 0。

当(u, v)或(v, u)都不在 E 中，则 u 和 v 之间不可能有网络流，即 $f(u, v)=f(v, u)=0$。(练习 26.1-1要求读者形式化地证明这一性质。)

关于流的性质，最后要说明的一点涉及正的网络流。进入一个顶点 v 的总的正网络流定义为：

$$\sum_{\substack{u\in V \\ f(u,v)>0}} f(u,v) \tag{26.2}$$

离开某顶点的正网络流是对称地进行定义的。定义某个顶点处的总的净流量(total net flow)为离开该顶点的总的正流量，减去进入该顶点的总的正流量。流守恒性的一种解释是这样的，即进入某个非源点非汇点顶点的正网络流，必须等于离开该顶点的正网络流。这个性质(即一个顶点处总的净流量必定为 0)常常被非形式化地称为“流进等于流出”。

644
~
645

网络流的一个例子

用流网络可以把图 26-1a 所示的卡车运输问题模型化。Lucky Puck 公司在 Vancouver 有一家制造冰球的工厂(源点 s)，在 Winnipeg 有一个存储产品的仓库(汇点 t)。Lucky Puck 从另外一家公司租用卡车，把冰球从工厂运到仓库。因为卡车按指定路线(边)在两城市(顶点)间行驶且其容

量有限，所以在图 26-1a 中，Lucky Puck 每天在每城市 u 和 v 之间至多装运 $c(u, v)$ 箱产品。Lucky Puck 公司无权控制运输路线和卡车的运输能力，所以不能改变图 26-1a 所示的流网络。他们的目标是确定每天所能运输的最大箱数 p，并按这一数量进行生产，因为生产出来的产品多于其运输能力是毫无意义的。Lucky Puck 公司并不关心把球从工厂运到仓库需要多少时间，他们关心的仅仅是每天有 p 箱球离开工厂，并且每天有 p 箱球到达仓库。

从表面看，将运输"流"模拟成网络流是非常适合的，因为每天从一个城市运输到另一城市的箱数受到容量限制。此外，必须遵循流守恒特性，在一种稳定状态下，球进入运输网络中间某城市的速度，必须等于它们离开该城市的速度，否则球就会堆积在中间城市中。

然而，在运输和流之间有一点细微的不同。Lucky Puck 可以将球从 Edmonton 运到 Calgary，也可以从 Calgary 运到 Edmonton。假设每天运输 8 箱从 Edmonton(图 26-1 中的 v_1)到 Calgary(v_2)，以及每天有 3 箱从 Calgary 到 Edmonton。将这些运输线路直接表示成网络流看似很自然，但是实际上不能。反对称性质要求 $f(v_1, v_2) = -f(v_2, v_1)$，但这里很清楚不满足这一性质，因为 $f(v_1, v_2) = 8$ 和 $f(v_2, v_1) = 3$。

Lucky Puck 可能会意识到每天从 Edmonton 运 8 箱到 Calgary 以及 3 箱从 Calgary 到 Edmonton 是没有意义的，因为他们可以每天从 Edmonton 运 5 箱到 Calgary，从 Calgar 运送 0 箱到 Edmonton，效果是一样的(在运输过程中可能使用更少的资源)。用流 $f(v_1, v_2) = 5$ 和 $f(v_2, v_1) = -5$ 来表示后一种场景。从效果上看，从 v_1 到 v_2 每天 8 箱中的 3 箱被从 v_2 到 v_1 每
646 天的 3 箱抵消了。

通常，利用抵消处理，可以将两城市间的运输用一个流来表示，该流在两个顶点之间的至多一条边上是正的。也就是说，任何在两城市间相互运输球的情况，都可以通过抵消将其转化成一个相等的情形，球只在一个方向上运输：沿正向流的方向。

给定一个实际运输的网络流 f，不能重构其准确的运输线路。如果我们知道 $f(u, v) = 5$，这也许表示有 5 个单位从 u 运输到了 v，或者表示从 u 到 v 运输了 8 个单位，从 v 到 u 运输了 3 个单位。其实，并不需关心实际的物理运输线路是如何铺设的；对于任意两点，我们只关心它们之间的净流量。如果我们确实关心底层运输情况的话，那么就需要使用另一种模型，以便能够记录两个方向上的运输信息。

在本章的算法中将隐式地利用抵消。假设边(u, v)有流量 $f(u, v)$。在一个算法的过程中，可能对边(u, v)上的流量增加 d。在数学上，这一操作为 $f(u, v)$减 d；从概念上看，可以认为这 d 个单位是对边(u, v)上 d 个单位流量的抵消。

具有多个源点和多个汇点的网络

在一个最大流问题中，可以有几个源点和几个汇点，而并非仅有一个源点和一个汇点。例如，Lucky Puck 公司实际可能拥有 m 个工厂$\{s_1, s_2, \cdots, s_m\}$和 n 个仓库$\{t_1, t_2, \cdots, t_n\}$，如图 26-2a 所示。所幸的是，这一问题并不比普通的最大流问题更难。

在具有多个源点和多个汇点的网络中，确定最大流的问题可以归约为一个普通的最大流问题。图 26-2b说明了图 26-2a 中的网络是如何转换为仅含有单源点和单汇点的一个普通流网络的。我们增加了一个*超级源点* s，并且对每个 $i = 1, 2, \cdots, m$ 加入有向边(s, s_i)，其容量 $c(s, s_i) = \infty$。同时，创建一个新的*超级汇点* t，并且对每个 $j = 1, 2, \cdots, n$ 加入有向边(t_j, t)，其容量 $c(t_j, t) = \infty$。从直观上看，图 26-2a 网络中的任意流均对应于图 26-2b 网络中的一个流，反之亦然。单源点 s 对多个源点 s_i 提供了其所需要的任意大的流。同样，单汇点 t 对多个汇点 t_j 消耗
647 其所需要的任意大的流。练习 26.1-3要求读者形式化地证明这两个问题是等价的。

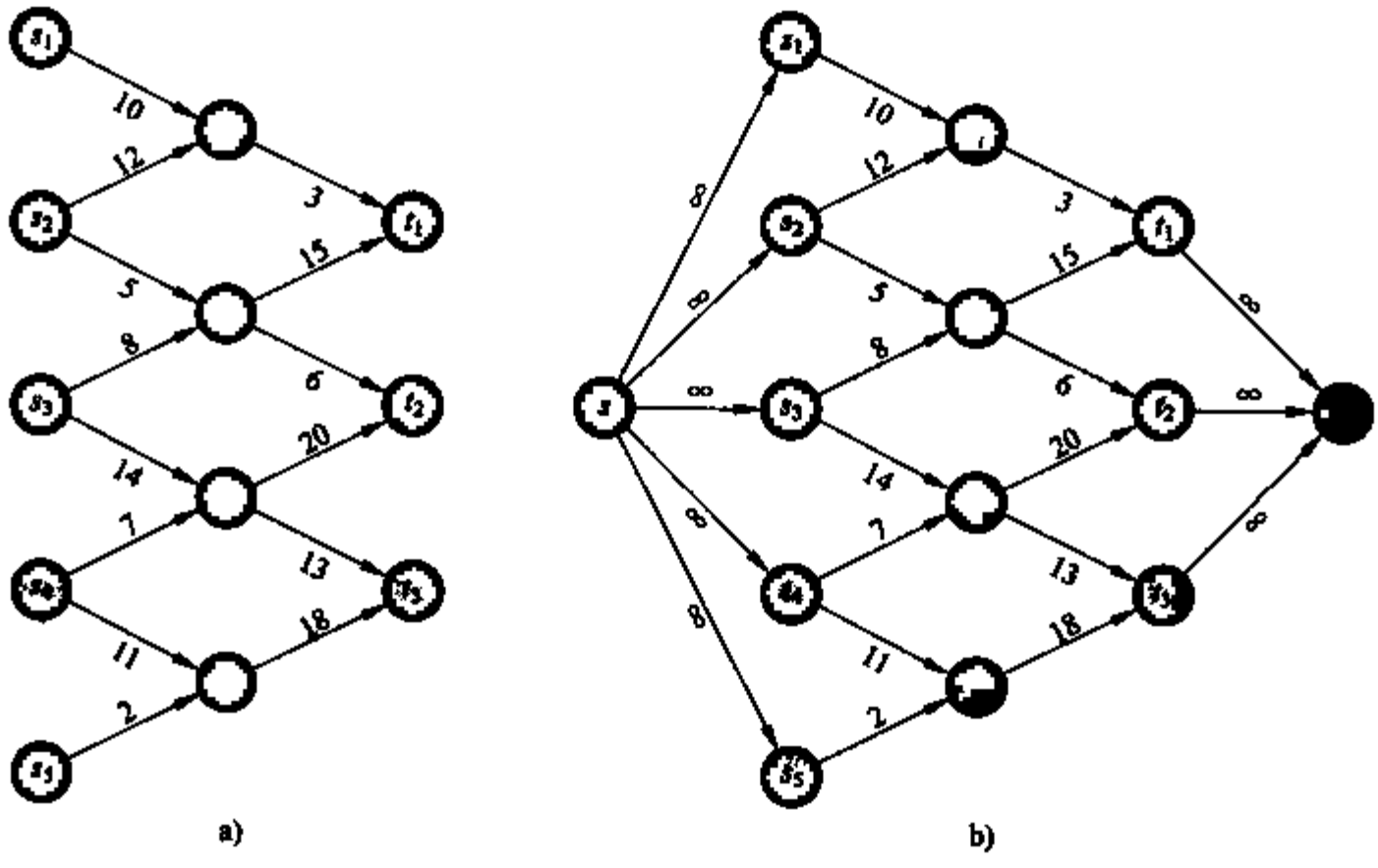


图 26-2　把多源点多汇点最大流问题转换为一个单源点单汇点问题。a)一个具有 5 个源点 $S=\{s_1, s_2, s_3, s_4, s_5\}$ 和 3 个汇点 $T=\{t_1, t_2, t_3\}$ 的流网络。b)一个等价单源点单汇点流网络。我们增加一个超级源点 s，以及从 s 到每个多源的源点具有无限容量的边。同时，增加一个超级汇点 t，以及从每个汇点到 t 的具有无限容量的边

对流的处理

下面来看一些函数(如 f)，它们以流网络中的两个顶点作为自变量。在本章中，我们将使用一种隐含求和记号，其中任何一个自变量或两个自变量可以是顶点的集合，它们所表示的值是对自变量所代表元素的所有可能的情形求和。例如，如果 X 和 Y 是顶点的集合，则

$$f(X,Y) = \sum_{x\in X}\sum_{y\in Y} f(x,y)$$

那么，流守恒限制可以表述为对所有 $u\in V-\{s, t\}$，有 $f(u, V)=0$。同时，为方便起见，在运用隐含求和记法时，我们将省略集合的大括号。例如，在等式 $f(s, V-s)=f(s, V)$ 中，项 $V-s$ 是指集合 $V-\{s\}$。

648

隐含集合记号常可以简化有关流的等式。下列引理给出了有关流和隐含集合记号的几个恒等式，其证明留作练习 26.1-4。

引理 26.1　设 $G=(V, E)$ 是一个流网络，f 是 G 中的一个流。那么下列等式成立：

1)对所有 $X\subseteq V$，$f(X, X)=0$。

2)对所有 $X, Y\subseteq V$，$f(X, Y)=-f(Y, X)$。

3)对所有 $X, Y, Z\subseteq V$，其中 $X\cap Y=\varnothing$，有 $f(X\cup Y, Z)=f(X, Z)+f(Y, Z)$ 且 $f(Z, X\cup Y)=f(Z, X)+f(Z, Y)$。 ■

作为应用隐含求和记法的一个例子，可以证明一个流的值为进入汇点的全部网络流，即

$$|f| = f(V,t) \tag{26.3}$$

直观上看这是正确的，根据流守恒特性，除了源点和汇点以外，对所有顶点来说，进入顶点的总的正流量等于离开该顶点的总的正流量。根据定义，源点顶点总的净流量大于 0；亦即，对源点顶点来说，离开它的正流要比进入它的正流更多。对称地，汇点顶点是唯一一个其总的净流量小于 0 的顶点；亦即，进入它的正流要比离开它的正流更多。形式证明如下：

$$
\begin{aligned}
|f| &= f(s,V) && \text{(定义)}\\
&= f(V,V) - f(V-s,V) && \text{(根据引理 26.1 第 3 点)}\\
&= -f(V-s,V) && \text{(根据引理 26.1 第 1 点)}\\
&= f(V,V-s) && \text{(根据引理 26.1 第 2 点)}\\
&= f(V,t) + f(V,V-s-t) && \text{(根据引理 26.1 第 3 点)}\\
&= f(V,t) && \text{(根据流守恒特性)}
\end{aligned}
$$

在本章的后面，我们将推广这一结论(引理 26.5)。

练习

26.1-1 利用流的定义，证明如果$(u, v)\notin E$且$(v, u)\notin E$，有$f(u, v)=f(v, u)=0$。

649 **26.1-2** 证明：对于任意非源点非汇点的顶点v，进入v的总正向流必定等于离开v的总正向流。

26.1-3 在多个源点和多个汇点的问题中，试对流的定义和特性进行扩展说明。证明在具有多个源点和多个汇点的流网络中，任意流均对应于通过增加一个超级源点和超级汇点所得到的具有相同值的一个单源点单汇点流，反之亦然。

26.1-4 证明引理 26.1。

26.1-5 对于图 26-1b 所示的流网络$G=(V, E)$和流f，找出两个子集合$X, Y\subseteq V$，且满足$f(X, Y)=-f(V-X, Y)$。再找出两个子集合$X, Y\subseteq V$，且满足$f(X, Y)\neq -f(V-X, Y)$。

26.1-6 给定流网络$G=(V, E)$，设f_1和f_2为$V\times V$到$\mathbf{R}$上的函数。流的和f_1+f_2是从$V\times V$到$\mathbf{R}$上的函数，定义如下：对所有$u, v\in V$

$$(f_1+f_2)(u,v) = f_1(u,v) + f_2(u,v) \tag{26.4}$$

如果f_1和f_2为G的流，则f_1+f_2必满足流的三条性质中的哪一条？又可能违反哪一条？

26.1-7 设f为网络中的一个流，α为实数。标量流之积(scalar flow product)αf是一个从$V\times V$到$\mathbf{R}$上的函数，定义为

$$(\alpha f)(u,v) = \alpha \cdot f(u,v)$$

证明网络中的流形成一个凸集。即，证明如果f_1和f_2是流，则对所有$0\leqslant\alpha\leqslant 1$，$\alpha f_1+(1-\alpha)f_2$也是流。

26.1-8 将最大流问题表述为一个线性规划问题。

26.1-9 亚当教授有两个孩子，可不幸的是他们互相不喜欢对方。问题是如此地严重：他们拒绝一同走到学校，而且甚至不愿意走过对方当天踏过的街区(在角落交叉的路径并不会产生问题)。幸运的是，教授的房子和学校都是在角落，但是他并不确定是否该把他的两个孩子送到同一个学校。教授有镇上的一份地图，试说明如何对于决定两个孩子是否可
650 以上同一所学校的问题，如何将其表述为一个最大流问题。

26.2 Ford-Fulkerson 方法

本节将讨论解决最大流问题的 Ford-Fulkerson 方法。之所以称为"方法"而不是"算法"，是由于它包含具有不同运行时间的几种实现。Ford-Fulkerson 方法依赖于三种重要思想，它们与该算法以外很多有关流的算法和问题密切相关。这三种思想是：残留网络(residual network)，增广路径(augmenting path)和割(cut)。这些思想是最大流最小割定理(定理 26.7)的精髓，该定理用流网络的割来描述最大流的值。在结束本节前，我们将给出 Ford-Fulkerson 方法的一种特定实现，并分析它的运行时间。

Ford-Fulkerson 方法是一种迭代方法。开始时，对所有$u, v\in V$有$f(u, v)=0$，即初始状

态时流的值为 0。在每次迭代中，可通过寻找一条“增广路径”来增加流值。增广路径可以看作是从源点 s 到汇点 t 之间的一条路径，沿该路径可以压入更多的流，从而增加流的值。反复进行这一过程，直至增广路径都被找出为止。最大流最小割定理将说明在算法终止时，这一过程可产生出最大流。

```
FORD-FULKERSON-METHORD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p
3        do augment flow f along p
4  return f
```

残留网络

直观上，给定流网络和一个流，其残留网络由可以容纳更多网络流的边所组成。更形式化地，假定有一个流网络 $G=(V, E)$，其源点为 s，汇点为 t。设 f 为 G 中的一个流，并考察一对顶点 $u, v \in V$。在不超过容量 $c(u, v)$ 的条件下，从 u 到 v 之间可以压入的额外网络流量，就是 (u, v) 的残留容量(residual capacity)，由下式定义：

$$c_f(u,v) = c(u,v) - f(u,v) \tag{26.5}$$

例如，如果 $c(u, v)=16$ 且 $f(u, v)=11$，则在不超过边 (u, v) 的容量限制的条件下，可以再传输 $c_f(u, v)=5$ 个单位的流来增加 $f(u, v)$。当网络流 $f(u, v)$ 为负值时，残留容量 $c_f(u, v)$ 大于容量 $c(u, v)$。例如，如果 $c(u, v)=16$ 且 $f(u, v)=-4$，则残留容量 $c_f(u, v)$ 为 20。关于这一点可以做如下解释：从 v 到 u 存在着 4 个单位的网络流，我们可以通过从 u 到 v 压入 4 个单位的网络流来抵消它。然后，在不超过边 (u, v) 的容量限制的条件下，还可以从 u 到 v 压入另外 16 个单位的网络流。因此，从开始时的网络流 $f(u, v)=-4$，共压入了额外的 20 个单位的网络流，并不会超过容量限制。 651

给定一流网络 $G=(V, E)$ 和流 f，由 f 压得的 G 的残留网络是 $G_f=(V, E_f)$，其中

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$$

这就是说，在残留网络中，每条边(或称为残留边)能够容纳一个严格为正的网络流。图 26-3a 再次说明了图 26-1b 中的流网络 G 和流 f。图 26-3b 说明了其相应的残留网络 G_f。

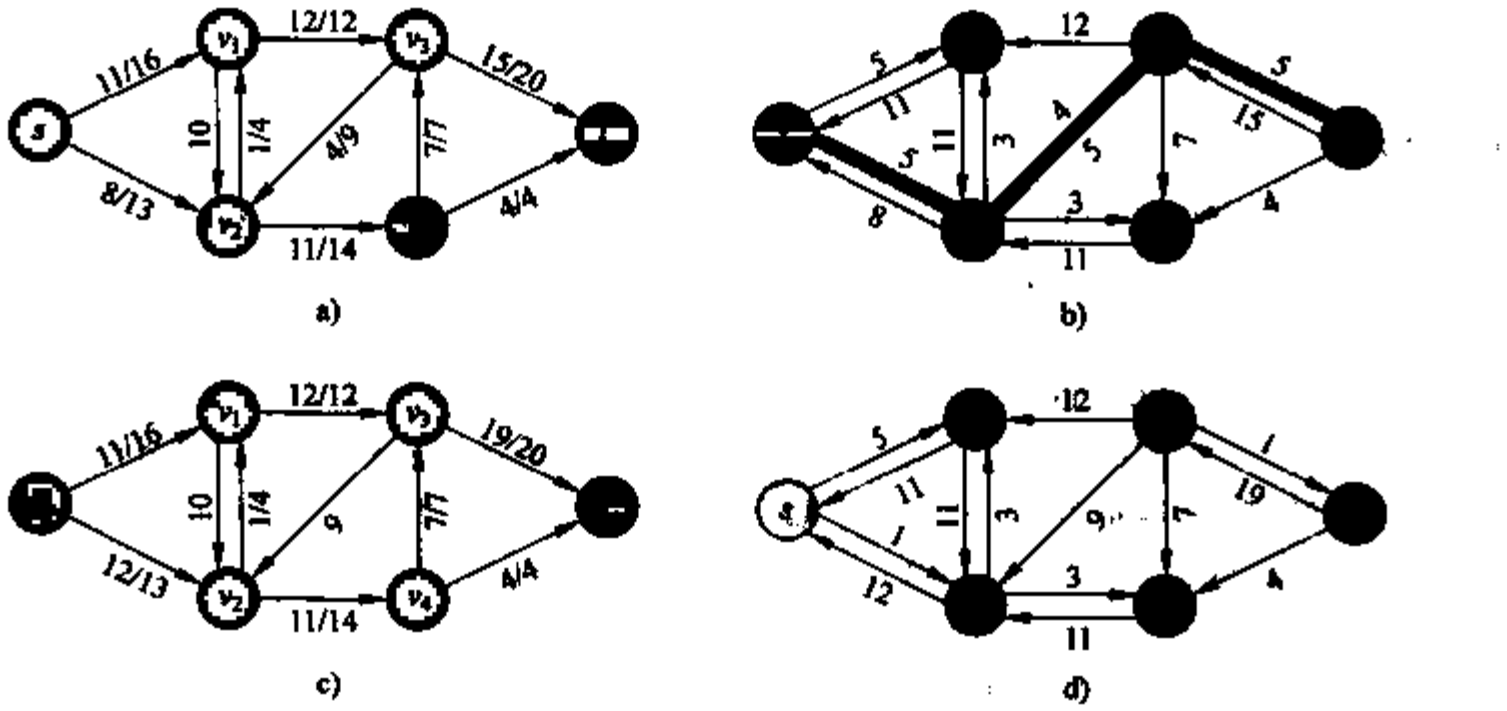


图 26-3　a)图 26-1b 中的流网络 G 和流 f。b)残留网络 G_f。阴影覆盖的边为增广路径 p，其残留容量为 $c_f(p)=c(v_2, v_3)=4$。c)G 中根据残留容量 4 沿路径 p 增加导出的流。d)根据 c)中的流导出的残留网络

E_f 中的边既可以是 E 中的边，也可以是它们的反向边。如果边 $(u, v)\in E$ 有 $f(u, v)<c(u, v)$，那么 $c_f(u, v)=c(u, v)-f(u, v)>0$ 且 $(u, v)\in E_f$。如果对边 $(u, v)\in E$，有 $f(u, v)>0$，那么 $f(v, u)<0$。在这样的情况下，$c_f(v, u)=c(v, u)-f(v, u)>0$，所以 $(v, u)\in E_f$。如果 (u, v) 或 (v, u) 都没有出现在原始的网络里，那么 $c(u, v)=c(v, u)=0$，$f(u, v)=f(v, u)=0$（根据练习 26.1-1），而且 $c_f(u, v)=c_f(v, u)=0$。只有当两条边 (u, v) 和 (v, u) 中，至少有一条边出现于初始网络中时，边 (u, v) 才能够出现在残留网络中，所以有如下限制条件：

652

$$|E_f|\leqslant 2|E|$$

注意，残留网络 G_f 本身也是一个流网络，其容量由 c_f 给出。下列引理说明残留网络中的流与初始网络中的流有何关系。

引理 26.2　设 $G=(V, E)$ 是源点为 s、汇点为 t 的一个流网络，且 f 为 G 中的一个流。设 G_f 是由 f 导出的 G 的残留网络，且 f' 为 G_f 中的一个流。那么，由等式(26.4)所定义的流之和 $f+f'$ 是 G 中的一个流，其值为 $|f+f'|=|f|+|f'|$。

证明：必须验证反对称性、容量限制和流守恒性都得到满足。关于反对称性，对所有 u，$v\in V$，有

$$\begin{aligned}(f+f')(u,v)&=f(u,v)+f'(u,v)=-f(v,u)-f'(v,u)\\&=-(f(v,u)+f'(v,u))=-(f+f')(v,u)\end{aligned}$$

关于容量限制，对所有 u，$v\in V$，$f'(v, u)\leqslant c_f(v, u)$，因此根据等式(26.5)，有：

$$\begin{aligned}(f+f')(u,v)&=f(u,v)+f'(u,v)\leqslant f(u,v)+(c(u,v)-f(u,v))\\&=c(u,v)\end{aligned}$$

关于流守恒性，对所有 $u\in V-\{s, t\}$，有：

$$\begin{aligned}\sum_{v\in V}(f+f')(u,v)&=\sum_{v\in V}(f(u,v)+f'(u,v))=\sum_{v\in V}f(u,v)+\sum_{v\in V}f'(u,v)\\&=0+0=0\end{aligned}$$

最后，有：

$$\begin{aligned}|f+f'|&=\sum_{v\in V}(f+f')(s,v)=\sum_{v\in V}(f(s,v)+f'(s,v))\\&=\sum_{v\in V}f(s,v)+\sum_{v\in V}f'(s,v)=|f|+|f'|\end{aligned}$$

653　■

增广路径

已知一个流网络 $G=(V, E)$ 和流 f，增广路径 p 为残留网络 G_f 中从 s 到 t 的一条简单路径。根据残留网络的定义，在不违反边的容量限制条件下，增广路径上的每条边 (u, v) 可以容纳从 u 到 v 的某额外正网络流。

图 26-3b 中阴影覆盖的路径是一条增广路径。如果把图中的残留网络 G_f 看作为一个流网络，在不违背容量限制的条件下，这条路径上在每条边还能够传输 4 个单位的额外网络流。因为该路径上的最小残留容量为 $c_f(v_2, v_3)=4$。称能够沿一条增广路径 p 的每条边传输的网络流的最大量为 p 的残留容量，由下式定义：

$$c_f(p)=\min\{c_f(u,v):(u,v)\text{ 在 }p\text{ 上}\}$$

下面的引理使上述论断更加准确，其证明留作练习 26.2-7。

引理 26.3　设 $G=(V, E)$ 是一个流网络，f 是 G 的一个流，并设 p 是 G_f 中的一条增广路径。我们用下式定义一个函数：$f_p: V\times V\to\mathbf{R}$

$$f_p(u,v)=\begin{cases}c_f(p) & \text{如果}(u,v)\text{在}p\text{上}\\ -c_f(p) & \text{如果}(v,u)\text{在}p\text{上}\\ 0 & \text{否则}\end{cases} \tag{26.6}$$

则 f_p 是 G_f 上的一个流，其值为 $|f_p|=c_f(p)>0$。■

下列推论说明如果把 f 加上 f_p，则可以得到 G 的另外一个流，其值更接近最大值。图 26-3c 说明把图 26-3a 中的 f 加上图 26-3b 中的 f_p 所得的结果。

推论 26.4　设 $G=(V, E)$是一个流网络，f 是 G 的一个流，p 是 G_f 中的一条增广路径。设 f_p 如等式(26.6)所定义。通过 $f'=f+f_p$ 定义一个函数 f'：$V\times V\to\mathbf{R}$。则 f' 是 G 的一个流，其值 $|f'|=|f|+|f_p|>|f|$。

证明：由引理 26.2 和引理 26.3 立即可得。■

流网络的割

Ford-Fulkerson 方法沿增广路径反复增加流，直至找出最大流时为止。要证明的最大流最小割定理告诉我们：一个流是最大流，当且仅当它的残留网络不包含增广路径。不过，为了证明该定理，我们必须先考察流网络的割这一概念。 654

流网络 $G=(V, E)$的割(S, T)将 V 划分为 S 和 $T=V-S$ 两部分，使得 $s\in S$，$t\in T$。(这一定义与第 23 章中用于最小生成树的"割"的定义相似，只是此处是对有向图而不是无向图进行分割，且满足 $s\in S$，$t\in T$。)如果 f 是一个流，则穿过割(S, T)的净流被定义为 $f(S, T)$。割(S, T)的容量为 $c(S, T)$。一个网络的最小割也就是网络中所有割中具有最小容量的割。

图 26-4 说明了图 26-1b 中的流网络的割$(\{s, v_1, v_2\}, \{v_3, v_4, t\})$。通过该割的净流为：

$$f(v_1,v_3)+f(v_2,v_3)+f(v_2,v_4)=12+(-4)+11=19$$

它的容量为

$$c(v_1,v_3)+c(v_2,v_4)=12+14=26$$

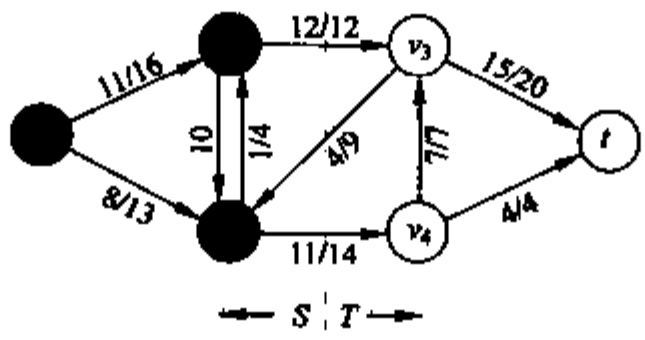


图 26-4　图 26-1b 中流网络的一个割(S, T)，其中 $S=\{s, v_1, v_2\}$，$T=\{v_3, v_4, t\}$。S 中的顶点是黑色，T 中顶点为白色。穿越(S, T)的净流量为 $f(S, T)=19$，容量为 $c(S, T)=26$

注意，通过割的净流可能包括顶点间的负网络流。但割的容量完全由非负值组成。换句话说，穿过割(S, T)的净流由双向的正网络流组成：在加上从 S 到 T 的正网络流的同时，减去从 T 到 S 的正网络流。另一方面，割(S, T)的容量仅由从 S 到 T 的边计算而得。从 T 到 S 的边在计算 $c(S, T)$时是不包括在内的。

下面引理说明流经任意割的净流都是相同的，且与流的值相等。 655

引理 26.5　设 f 是源点为 s，汇点为 t 的流网络 G 中的一个流。并且(S, T)是 G 的一个割。则通过割(S, T)的净流为 $f(S, T)=|f|$。

证明：根据流守恒性，有 $f(S-s, V)=0$，因而有：

$$f(S,T)=f(S,V)-f(S,S)\qquad\text{(根据引理 26.1,第 3) 部分)}$$

$$
\begin{aligned}
&= f(S,V) && \text{(根据引理 26.1,第 1) 部分)}\\
&= f(s,V) + f(S-s,V) && \text{(根据引理 26.1,第 3) 部分)}\\
&= f(s,V) && \text{(因为 } f(S-s,V)=0\text{)}\\
&= |f| &&
\end{aligned}
$$

■

引理 26.5 的一个直接推论是我们早先证明过的结论等式(26.3)，即一个流的值为进入汇点的总网络流量。

引理 26.5 的另一个推论说明如何运用割的容量来限制一个流的值。

推论 26.6　对一个流网络 G 中任意流 f 来说，其值的上界为 G 的任意割的容量。

证明：设 (S, T) 为 G 中的任意割，且 f 为任意流。根据引理 26.5 和容量限制，有

$$|f| = f(S,T) = \sum_{u\in S}\sum_{v\in T} f(u,v) \leqslant \sum_{u\in S}\sum_{v\in T} c(u,v) = c(S,T)$$

■

由推论 26.6 可以得出一个直接的结论，网络的最大流必定不超过此网络最小割的容量。下
656 面就来证明重要的最大流最小割定理，即，最大流的值实际上等于某一最小割的容量。

定理 26.7(最大流最小割定理)　如果 f 是具有源点 s 和汇点 t 的流网络 $G=(V, E)$ 中的一个流，则下列条件是等价的：

1) f 是 G 的一个最大流。

2) 残留网络 G_f 不包含增广路径。

3) 对 G 的某个割 (S, T)，有 $|f| = c(S, T)$。

证明：1)⇒2)：为了引入矛盾，假设 f 是 G 的最大流，但 G_f 中包含一条增广路径 p。由推论 26.4，流的和 $f+f_p$ 为 G 的一个流，其值严格大于 $|f|$ (f_p 由等式(26.6)给出)。这与假设 f 是最大流相矛盾。

2)⇒3)：假设 G_f 中不包含增广路径，即 G_f 不包含从 s 到 v 的路径。定义

$$S = \{v \in V : G_f \text{ 中从 } s \text{ 到 } v \text{ 存在一条通路}\}$$

并且 $T=V-S$。划分 (S, T) 是一个割：$s\in S$，由于 G_f 中不存在从 s 到 t 的路径，所以 $t\notin S$。对每对顶点 $u\in S$，$v\in T$，有 $f(u, v)=c(u, v)$，否则 $(u, v)\in E_f$，v 就属于集合 S。因此由引理 26.5，$|f| = f(S, T)=c(S, T)$。

3)⇒1)：由推论 26.6 可知，对所有的割 (S, T)，有 $|f| \leqslant c(S, T)$。因此条件 $|f| = c(S, T)$ 说明 f 是一个最大流。

■

基本的 Ford-Fulkerson 算法

在 Ford-Fulkerson 方法的每次迭代中，找出任意增广路径 p，并把沿 p 每条边的流 f 加上其残留容量 $c_f(p)$。在 Ford-Fulkerson 方法的以下实现中，通过更新有边相连的每对顶点 u，v 之间的网络流 $f[u, v]$，来计算出图 $G=(V, E)$ 中的最大流。[⊖] 如果 u 和 v 之间在任意方向没有边相连，则隐含地假设 $f[u, v]=0$。$c(u, v)$ 的容量假定已经在图中给出，且如果 $(u, v)\notin E$，有 $c(u, v)=0$。残留容量 $c_f(u, v)$ 按公式(26.5)计算。代码中的符号 $c_f(p)$ 实际上只是存储路径 p
657 的残留容量的一个临时变量。

```
FORD-FULKERSON(G, s, t)
1  for each edge(u, v) ∈ E[G]
2       do f[u, v] ← 0
3          f[v, u] ← 0
4  while there exists a path p from s to t in the residual network G_f
5       do c_f(p) ← min {c_f(u, v): (u, v) is in p}
```

⊖ 使用方括号来标识易变域(如 f)，用圆括号来表示函数。

```
6        for each edge(u, v)in p
7            do f[u, v] ← f[u, v]+c_f(p)
8               f[v, u] ← −f[u, v]
```

FORD-FULKERSON 算法仅仅是对前面给出的 FORD-FULKERSON-METHOD 伪代码的扩充。图 26-5 说明算法在样图上运行中每次迭代的结果。第 1～3 行把流 f 初始化为 0。第 4～8 行的 while 循环反复找出 G_f 中的增广路径 p，并把沿 p 的流 f 加上其残留容量 $c_f(p)$。当不再有增广路径时，流 f 就是一个最大流。

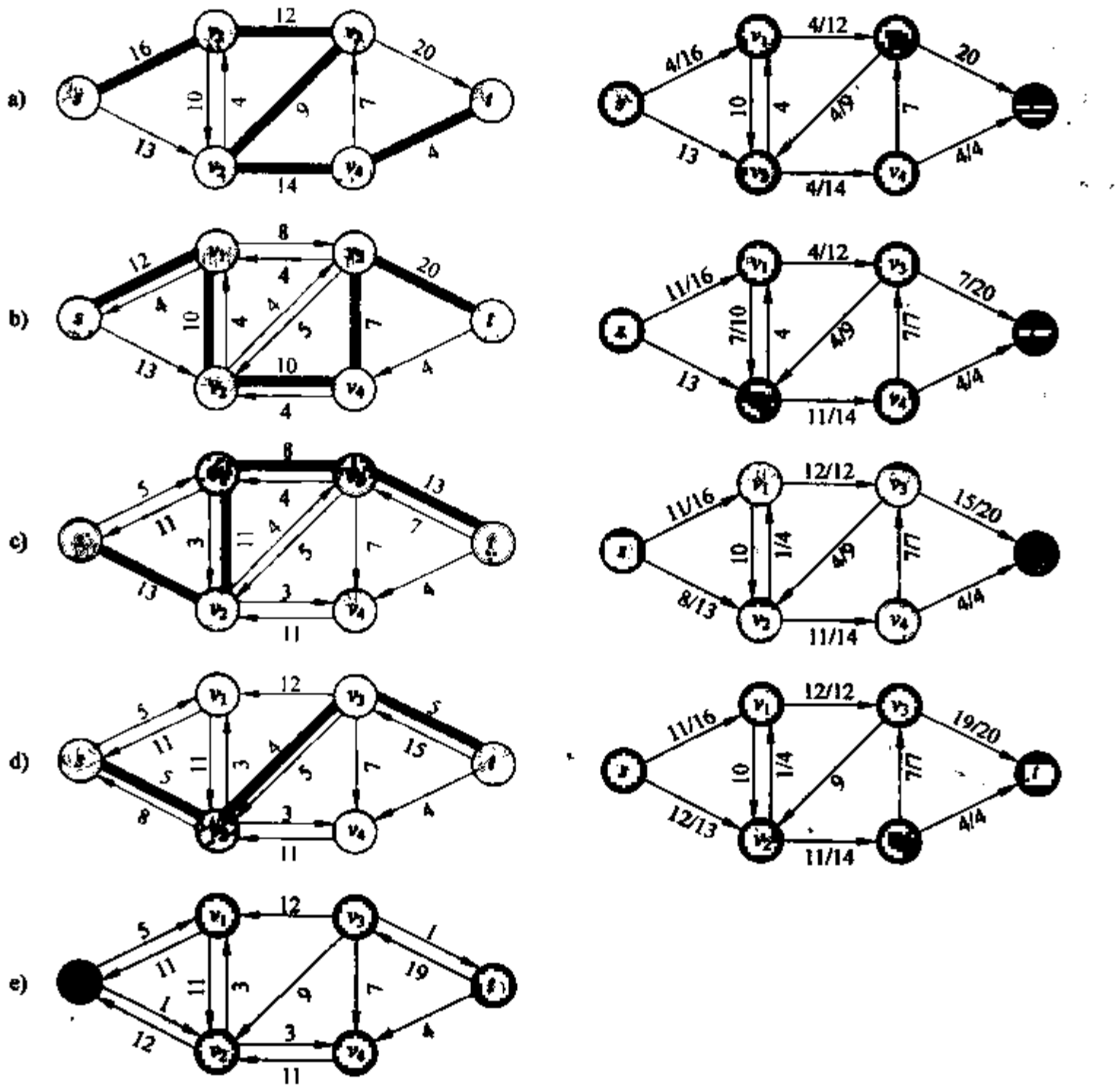


图 26-5　基本 Ford-Fulkerson 算法的执行。a)～d)while 循环相继地迭代过程。每部分的左边表示从第 4 行开始的残留网络 G_f，并用阴影标出了增广路径 p。每部分的右边表示将 f_p 加入到 f 后形成的新流 f。a)中的残留网络是输入网络 G。e)最后在 while 循环测试的残留网络。它没有增广路径，且 d)中显示的流 f 就是一个最大流

Ford-Fulkerson 算法的分析

FORD-FULERSON 过程的运行时间取决于如何确定第 4 行中的增广路径。如果选择不好，算法甚至可能不会终止：流的值随着求和运算将不断增加，但它甚至不会收敛到流的最大

值。[⊖]不过，如果采用广度优先搜索(见第 22.2 节)来选择增广路径，算法的运行时间为多项式时间复杂度。但是，在证明这一点之前，先任意选择增广路径、且所有容量均为整数，取得一个简单的界。

在实际中碰到的大多数最大流的问题中其容量经常为整数。如果容量为有理数，则经过适当的按比例转换，可以使它们都变为整数。在这一假设下，FORD-FULKERSON 的一种简易实现的运行时间为 $O(E \mid f^* \mid)$，其中 f^* 是算法找出的最大流。具体分析如下：第 1～3 行运行时间为 $\Theta(E)$。第 4～8 行的 while 循环至多执行 $\mid f^* \mid$ 次，因为在每次迭代中，流的值至少增加一个单位。

658
~
659

如果能够有效地操纵用于实现网络 $G=(V, E)$ 的数据结构，while 循环的效率就比较高。假定有一种对应于有向图 $G'=(V, E')$ 的数据结构，其中 $E'=\{(u, v): (u, v) \in E$ 或 $(v, u) \in E\}$。网络 G 中的边也同样是 G' 中的边，因此在这一数据结构中，保持其容量和流就非常简单了。如果给定 G 的流 f，则残留网络 G_f 中的边是 G' 中所有满足 $c(u, v)-f(u, v) \neq 0$ 的边 (u, v) 所组成的。因此，如果采用深度优先搜索算法或广度优先搜索，在残留网络中寻找一条路径的运行时间应为 $O(V+E')=O(E)$。所以，while 循环中的每次迭代所占用的时间为 $O(E)$，这就使得 FORD-FULKERSON 过程的整个运行时间为 $O(E \mid f^* \mid)$。

当容量为整数且最佳流的值 $\mid f^* \mid$ 较小时，FORD-FULKERSON 算法的运行时间还是不错的。图 26-6a 举例说明了在 $\mid f^* \mid$ 较大的一个简单流网络上，运行此算法时所产生的结果。在该网络中一个最大流的值为 2 000 000：1 000 000 单位的流通过路径 $s \to u \to t$，另一个 1 000 000 单位的流通过路径 $s \to v \to t$。如图 26-6a 所示，如果由 FORD-FULKERSON 找出的第一条增广路径为 $s \to u \to v \to t$，则在第一次迭代完成后流的值为 1。算法导出的残留网络如图 26-6b 所示。如果在第二次迭代中找出的增广路径为 $s \to v \to u \to t$(如图 26-6b 所示)，则流的值变为 2。图 26-6c 说明产生的残留网络。我们可以在奇数次的迭代中，选择 $s \to u \to v \to t$ 作为增广路径，在偶数次的迭代中，选择 $s \to v \to u \to t$ 作为增广路径，并如此继续下去。因为每次对流的量仅增加 1 个单位，所以总共要执行 2 000 000 次加法运算。

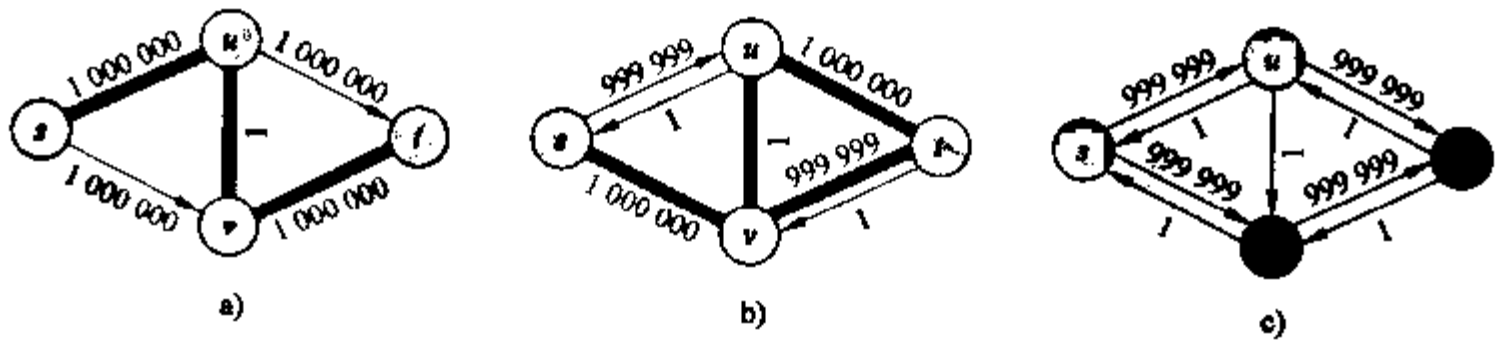


图 26-6　a)一个流网络，其 FORD-FULKERSON 的运行时间为 $\Theta(E \mid f^* \mid)$，其中 f^* 是一个最大流，这里 $\mid f^* \mid = 2\,000\,000$。图中显示了一条残余容量为 1 的增广路径。b)结果残余网络。另一条残余容量为 1 的增广路径。c)结果残余网络

Edmonds-Karp 算法

如果在第 4 行中用广度优先搜索来实现对增广路径 p 的计算，即，如果增广路径是残留网络
660 中从 s 到 t 的最短路径(其中每条边为单位距离，或权)，则能够改进 FORD-FULKERSON 的界。称 Ford-Fulkerson 方法的这种实现为 Edmonds-Karp 算法。现在来证明 Edmonds-Karp 算法的运

⊖ 仅当边容量为非有理数时，Ford-Fulkerson 方法可能不能终止。然而，事实上非有理数不能存储在有限精度的计算机中。

行时间为$O(VE^2)$。

分析依赖于至残留网络G_f中顶点的距离。下列引理使用符号$\delta_f(u, v)$来表示每条边为单位长度的图G_f中，从u到v之间的最短路径长度。

引理 26.8　如果对具有源点s和汇点t的流网络$G=(V, E)$运行 Edmonds-Karp 算法，则对所有顶点$v\in V-\{s, t\}$，残留网络G_f中的最短路径长度$\delta_f(u, v)$随着每个流的增加而单调递增。

证明：假定对某个顶点$v\in V-\{s, t\}$存在着流增加，它引起s到v之间最短路径距离减少，继而导出矛盾。设f为第一次增加以前的流，该增加导致了某个最短路径距离的减小；设f'为增加以后的流。设v是在流增加时最小距离$\delta_{f'}(u, v)$被减小的顶点，因此$\delta_{f'}(s, v)<\delta_f(s, v)$。设$p=s\rightsquigarrow u\rightarrow v$为$G_{f'}$中从$s$到$v$的最短路径，因此$(u, v)\in E_{f'}$且

$$\delta_{f'}(s,u)=\delta_{f'}(s,v)-1 \tag{26.7}$$

因为无论怎样选择v，我们知道顶点u的距离标号都不会减小，亦即：

$$\delta_{f'}(s,u)\geqslant\delta_f(s,u) \tag{26.8}$$

我们断言$(u, v)\notin E_f$，为什么呢？因为如果有$(u, v)\in E_f$，那么有

$$\begin{aligned}\delta_f(s,v)&\leqslant\delta_f(s,u)+1 && \text{(根据引理 24.10 三角不等式)}\\ &\leqslant\delta_{f'}(s,u)+1 && \text{(根据不等式(26.8))}\\ &=\delta_{f'}(s,v) && \text{(根据等式(26.7))}\end{aligned}$$

这与我们的假设$\delta_{f'}(s, v)<\delta_f(s, v)$相矛盾。

如何才有$(u, v)\notin E_f$且$(u, v)\in E_{f'}$？流增加必定将从v到u的流增加。Edmonds-Karp 算法总是沿着最短路径增加流，所以G_f中从s到u的最短路径以(v, u)作为其最后一条边。于是有：

$$\begin{aligned}\delta_f(s,v)&=\delta_f(s,u)-1\\ &\leqslant\delta_{f'}(s,u)-1 && \text{(根据不等式(26.8))}\\ &=\delta_{f'}(s,v)-2 && \text{(根据等式(26.7))}\end{aligned}$$

这与我们开始所做的假设$\delta_{f'}(s, v)<\delta_f(s, v)$相矛盾。因此可以得出结论，我们假设存在这样的顶点$v$是不正确的。（证毕）■ 661

下面一个定理给出了 Edmondes-Karp 算法中的迭代次数的界。

定理 26.9　如果对具有源点s和汇点t的一个流网络$G=(V, E)$运行 Edmonds-Karp 算法，对流进行增加的全部次数为$O(VE)$。

证明：在一残留网络G_f中，如果其增广路径p的残留容量是边(u, v)的残留容量，即，如果$c_f(p)=c_f(u, v)$，则说边(u, v)对增广路径p是关键的。在沿增广路径对流进行增加后，该路径上的任何关键边便从残留网络中消失。此外，任何增广路径上至少有一条边必为关键边。下面将证明$|E|$条边中的每一条都可能至多$|V|/2-1$次地成为关键边。

设u和v为V中的顶点，且它们之间由E中的一条边相连。(u, v)可以多少次作为关键边？由于增广路径是最短路径，所以当(u, v)第一次作为关键边时，有

$$\delta_f(s,v)=\delta_f(s,u)+1$$

一旦对流进行增加后，边(u, v)就从残留网络中消失。以后，也不能重新出现在另一条增广路径上，直到从u到v的网络流减小后为止，并且，只有当(u, v)出现在增广路径上时，这种情况才会发生。如果当这一事件发生时f'是G的流，则有

$$\delta_{f'}(s,u)=\delta_{f'}(s,v)+1$$

因为根据引理 26.8 有 $\delta_f(s, v) \leqslant \delta_{f'}(s, v)$，进而有：

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 \geqslant \delta_f(s,v) + 1 = \delta_f(s,u) + 2$$

所以，从(u, v)成为关键边的时刻到它再次成为关键边的时刻，从源点到 u 的距离至少增加了 2。初始时从源点到 u 的距离至少为 0。从 s 到 u 的最短路径上的中间顶点不包含 s，u 或 t（因为关键路径上的(u, v)暗含了 $u \neq t$）。因此直到 u 变为从源点不可达(如果可能的话)之前，其距离至多为 $|V|-2$。因此边(u, v)至多$(|V|-2)/2 = |V|/2-1$ 次成为关键边。因为在残留网络中，可能有 $O(E)$对顶点间有边相连，所以在 Edmonds-Karp 算法的整个执行过程中，全部关键边的数目为 $O(VE)$。每条增广路径至少存在一条关键边，因此定理成立。(证毕) ■

由于在用广度优先搜索寻找增广路径时，FORD-FULKERSON 中的每次迭代都可以在$O(E)$
662 的运行时间内完成，所以 Edmonds-Karp 算法的全部运行时间为 $O(VE^2)$。我们将会看到，压入与重标记算法能够达到更好的界。第 26.4 节中的算法给出了达到 $O(V^2E)$运行时间的方法，它同时是第 26.5 节$O(V^3)$时间算法的基础。

练习

26.2-1 在图 26-1b 中，通过割$(\{s, v_2, v_4\}, \{v_1, v_3, t\})$的流是多少？该割的容量是多少？

26.2-2 试说明 Edmonds-Karp 算法在图 26-1a 所示的流网络上执行的过程。

26.2-3 在图 26-5 的实例中，对应于图中最大流的最小割是多少？在例中出现的增广路径中，哪两条路径抵消了先前被传输的流？

26.2-4 证明对任意一对顶点 u 和 v、任意的容量和流函数 c 和 f，有：

$$c_f(u,v) + c_f(v,u) = c(u,v) + c(v,u)$$

26.2-5 在 26.1 节中，我们通过增加具有无限容量的边，把一个多源点多汇点的流网络转换为单源点单汇点的流网络。证明：如果初始的多源点多汇点网络中的边具有有限的容量，则转换后所得的网络中的任意流均为有限值。

26.2-6 假定在多源点多汇点问题中，每个源点 s_i 产生 p_i 单位的流，即 $f(s_i, V) = p_i$。同时假定每个汇点 t_j 消耗 q_j 单位的流，即 $f(V, t_j) = q_j$，其中 $\sum_i p_i = \sum_j p_j$。说明如何把寻找一个流 f 以满足这些附加限制的问题，转化为在一个单源点单汇点流网络中寻找最大流的问题。

663 **26.2-7** 证明引理 26.3。

26.2-8 证明：一个网络 $G=(V, E)$的最大流总可以被至多由 $|E|$ 条增广路径所组成的序列发现。(提示：找出最大流后再确定路径。)

26.2-9 无向图边连通度(edge connectivity)是指为了使图不连通而必须去掉的最少边数 k。例如，树的边连通度为 1，顶点的循环链的边连通度是 2。说明如何对至多 $|V|$ 个流网络(每个流网络有 $O(V)$个顶点和 $O(E)$条边)运行最大流算法，就可确定无向图 $G=(V, E)$的边连通度。

26.2-10 假定一个流网络 $G=(V, E)$中有对称边，也就是$(u, v) \in E$ 当且仅当$(v, u) \in E$，试证明 Edmonds-Karp 算法在至多进行 $|V||E|/4$ 次迭代后将终止执行。(提示：对任意边(u, v)，考虑(u, v)为关键边的时刻之间，$\delta(s, u)$和 $\delta(v, t)$是如何变化的。)

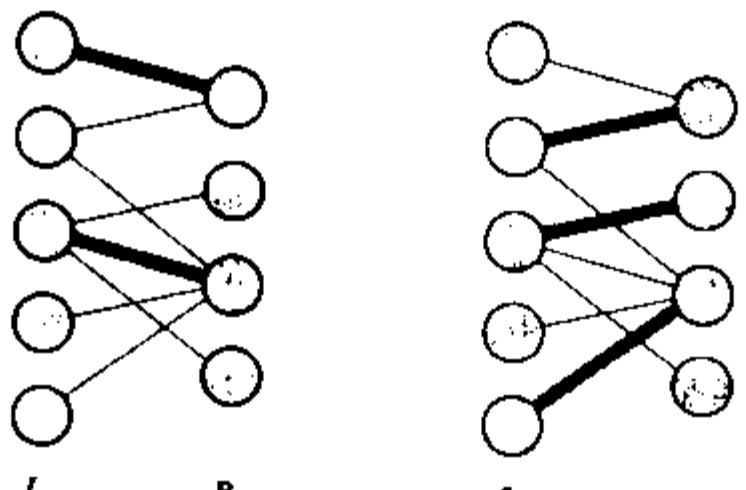
26.3　最大二分匹配

一些组合问题可以很容易地转换为最大流问题。26.1 节中的多源点、多汇点最大流问题就是一个例子。其他一些组合问题从表面上看，似乎与流网络没有什么联系，但实际上也可以转

换为最大流问题。本节就来提出这样的一个问题：在一个二分图(见 B.4 节)中寻找最大匹配。为了解决这一问题，将利用由 Ford-Fulkerson 方法提供的完整性性质(integrality property)。还将看到，可以应用 Ford-Fulkerson 方法在 $O(VE)$ 时间内解决图 $G=(V, E)$ 的最大二分匹配问题。

最大二分匹配问题

给定一个无向图 $G=(V, E)$，一个匹配(matching)是一个边的子集合 $M \subseteq E$，且满足对所有顶点 $v \in V$，M 中至多有一条边与 v 关联。如果 M 中某条边与 v 关联，则说顶点 $v \in V$ 被匹配，否则说 v 是无匹配的。最大匹配是最大势的匹配，也就是说，是满足对任意匹配 M'，有 $|M| \geqslant |M'|$ 的匹配 M。在本节中，我们将把注意力集中在寻找二分图的最大匹配上。假定顶点集合可被划分为 $V=L \cup R$，其中 L 和 R 是不相交的，且 E 中的所有边的一个端点在 R 中，另一端点在 L 中。进一步假设 V 中的每个顶点至少有一条关联的边。图 26-7 说明了匹配的概念。 664



a)　　b)

图 26-7　顶点划分为 $V=L \cup R$ 的二分图 $G=(V, E)$。a)势为 2 的一个匹配。b)势为 3 的最大匹配

二分图的最大匹配问题有着许多实际的应用。例如，把一个机器集合 L 和要同时执行的任务集合 R 相匹配。E 中有边 (u, v)，就说明一台特定机器 $u \in L$ 能够完成一项特定任务 $v \in R$。最大匹配可以为尽可能多的机器提供任务。

寻找最大二分匹配

利用 Ford-Fulkerson 方法可以在关于 $|V|$ 和 $|E|$ 的多项式时间内，找出无向二分图 $G=(V, E)$ 的最大匹配。解决这一问题的关键技巧在于建立一个流网络，其中流对应于匹配，如图 26-8 所示。对二分图 G 的相应流网络 $G'=(V', E')$ 定义如下。设源点 s 和汇点 t 是不属于 V 的新顶点，$V'=V \cup \{s, t\}$。如果 G 的顶点划分为 $V=L \cup R$，G' 的有向边为 E 的边，从 L 指向 R，再加上 V 条新边：

$$\begin{aligned} E' = & \{(s,u) : u \in L\} \\ & \cup \{(u,v) : u \in L, v \in R, 和 (u,v) \in E\} \\ & \cup \{(v,t) : v \in R\} \end{aligned}$$

在结束构造工作之前，对 E' 中的每条边赋予单位容量。因为 V 中的每个顶点至少有一条关联边，$|E| \geqslant |V|/2$。因此，$|E| \leqslant |E'| = |E| + |V| \leqslant 3|E|$，则 $|E'| = \Theta(E)$。 665

下列引理说明了 G 的匹配直接对应于 G 的相应流网络 G' 中的流。如果对所有 $(u, v) \in V \times V$，$f(u, v)$ 是一个整数，则说流网络 $G=(V, E)$ 上的流 f 是具有整数值的。

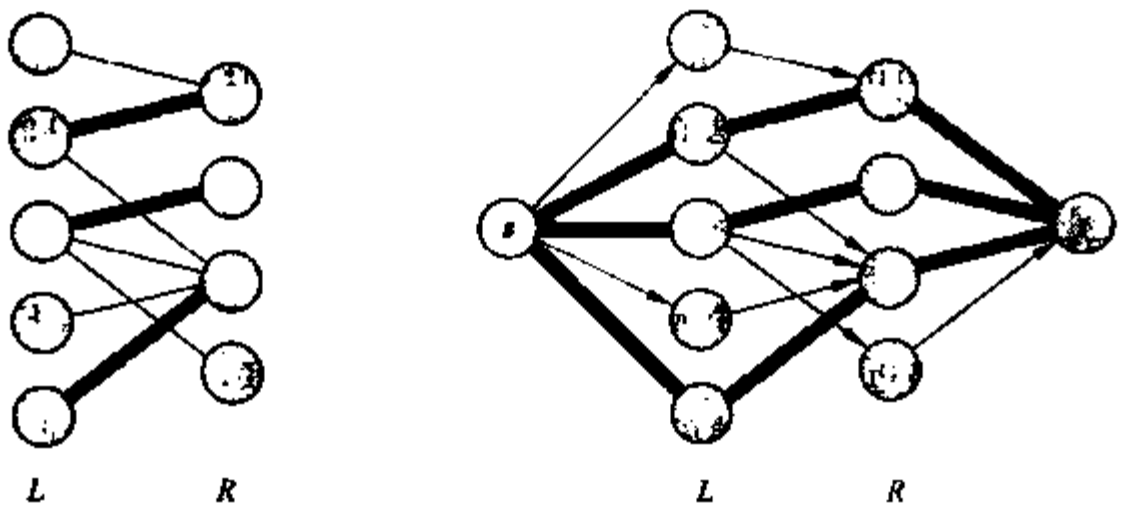


图 26-8　与一个二分图对应的流网络。a)图 26-7 中的二分图 $G=(V, E)$，其顶点划分为 $V=L\cup R$。图中阴影边示出的是一个最大匹配。b)对应的流网络 G'，图中示出了一个最大流。每条边具有单位容量。阴影边的流量为 1，所有其他的边上没有流量。从 L 指向 R 的阴影边对应于二分图的最大匹配中的边

引理 26.10　设 $G=(V, E)$是一个二分图，其顶点划分为 $V=L\cup R$，设 $G'=(V', E')$是它相应的流网络。如果 M 是 G 的匹配，则 G'中存在一个整数值的流 f，且 $|f|=|M|$。相反地，如果 f 是 G'中的整数值流，则 G 中存在一匹配 M 满足 $|M|=|f|$。

证明：先来证明 G 的匹配 M 对应于 G'中一整数值的流。定义 f 如下，如果 $(u, v)\in M$，则 $f(s, u)=f(u, v)=f(v, t)=1$ 且 $f(u, s)=f(v, u)=f(t, v)=-1$。对所有其他边 $(u, v)\in E'$，定义 $f(u, v)=0$。可以很容易地证明 f 满足反对称性、容量限制和流守恒性。

从直观上看，每条边 $(u, v)\in M$ 相应于 G'中经过路径 $s\to u\to v\to t$ 的 1 个单位的流。此外，
666 除了 s 和 t 外，由 M 中的边引出的各路径上，顶点都是各不相同的。穿过割 $(L\cup\{s\}, R\cup\{t\})$的净流与 $|M|$ 相等；因此，根据引理 26.5，流的值为 $|f|=|M|$。

为了证明定理的后一部分成立，设 f 为 G'中一个具有整数值的流，并设

$$M=\{(u,v): u\in L, v\in R, \text{和 } f(u,v)>0\}$$

对每个顶点 $u\in L$，仅有一条进入该顶点的边，即 (s, u)，其容量为 1。因此对每个顶点 $u\in L$，至多有 1 个单位的正网络流进入该顶点，并且，如果的确有 1 个单位的正网络流流进了该顶点，则根据流守恒性，必定有 1 个单位的正网络流流出。此外，由于 f 的值为整数，所以对每个顶点 $u\in L$，1 单位的流可以至多进入一条边，且至多从一条边流出。因此 1 个单位的网络流进入 u 当且仅当恰有一个顶点 $v\in R$，满足 $f(u, v)=1$，而且每个顶点 $u\in L$ 至多有一个出边带有正网络流。对每个顶点 $v\in R$ 也有一个对称的结论。因此，引理中定义的集合 M 是一个匹配。

为证明 $|M|=|f|$，注意到对每一个被匹配的顶点 $u\in L$，有 $f(s, u)=1$，并且对每条边 $(u, v)\in E-M$，有 $f(u, v)=0$。因此，根据引理 26.1 有：

$$|M|=f(L,R)=f(L,V')-f(L,L)-f(L,s)-f(L,t) \quad \text{（根据引理 26.1）}$$

对于上式做进一步化简。流守恒性暗示了 $f(L, V')=0$；引理 26.1 说明 $f(L, L)=0$；反对称性表明 $-f(L, s)=f(s, L)$；而且从 L 到 t 没有边，有 $f(L, t)=0$。因此，

$$\begin{aligned}|M| &= f(s,L)\\ &= f(s,V') && \text{（因为所有 } s \text{ 的出边都进入 } L)\\ &= |f| && \text{（根据 } |f| \text{ 的定义）}\end{aligned}$$

■

根据引理 26.10，可以得出结论，在一个二分图 G 中，一个最大匹配对应于流网络 G' 中的一个最大流。因此，可以通过对 G' 运行最大流算法来计算出 G 的最大匹配。这一推理过程中存在的唯一障碍就是最大流算法可能返回一个 G' 的非整数量的流 $f(u, v)$，即使流的值 $|f|$ 应该为一个整数。下列定理说明，如果采用 Ford-Fulkerson 方法，这一困难就不会出现。

定理 26.11(完整性定理)　如果容量函数 c 只取整数值，则由 Ford-Fulkerson 方法得出的最大流 f 满足 $|f|$ 为整数的性质。此外，对所有顶点 u 和 v，$f(u, v)$ 的值为整数。 667

证明：通过对迭代次数归纳来进行证明。具体证明留作练习 26.3-2。 ■

下面来证明引理 26.10 的一个推论。

推论 26.12　二分图 G 的一个最大匹配 M 的势是其相应的流网络 G' 中某一最大流 f 的值。

证明：下面的证明采用引理 26.10 中的术语。假定 M 是 G 的最大匹配，且其相应的 G' 中的流 f 不是最大流。则 G' 中必有一最大流 f' 满足 $|f'| > |f|$。由于 G' 中的容量均为整数，则由定理 26.11 可知 f' 的值也为整数。因此，f' 对应于 G 中的匹配 M'，且其势 $|M'| = |f'| > |f| = |M|$，这与 M 是最大匹配的假设相矛盾。用类似的方法可以证明：如果 f 是 G' 的最大流，其相应的匹配是 G 上的最大匹配。 ■

因此，对于一个无向二分图 G，可以用下列方法找出其最大匹配：先建立流网络 G'，对它运行 Ford-Fulkerson 方法，根据求得的具有整数值的最大流 f，就可直接获得最大匹配 M。因为二分图上的任何匹配的势至多为 $\min(L, R) = O(V)$，所以 G' 中最大流的值为 $O(V)$，因此，我们可以在 $O(VE') = O(VE)$ 的时间内，找出一个二分图的最大匹配，因为 $|E'| = \Theta(E)$。

练习

26.3-1　对图 26-8b 中所示的流网络运行 Ford-Fulkerson 算法。并指出每次对流增加以后所得的残留网络。对 L 中的顶点从上到下编为 1～5 号，对 R 中的顶点从上到下编为 6～9 号。在每次迭代中，找出按辞典顺序排列时，最小的一条增广路径。

26.3-2　证明定理 26.11。

26.3-3　设 $G=(V, E)$ 为二分图，其顶点划分为 $V=L\cup R$，且 G' 是其相应的流网络，在 FORD-FULKERSON 执行过程中，对在 G' 中找出的任意增广路径的长度给出一个适当的上界。 668

***26.3-4**　完全匹配(perfect matching)是指图中每个顶点均被匹配。设 $G=(V, E)$ 是其顶点分划为 $V=L\cup R$ 的一个无向二分图，其中 $|L| = |R|$。对任意 $X\subseteq V$，定义 X 的邻居为：

$$N(X) = \{y \in V : (x,y) \in E, \text{对某个 } x \in X\}$$

即，由与 X 的某元素相邻的顶点所构成的集合。证明 Hall 定理：G 中存在一个完全匹配，当且仅当对每个子集 $A\subseteq L$，有 $|A| \leqslant |N(A)|$。

***26.3-5**　在二分图 $G=(V, E)$ 中，$V=L\cup R$，如果每个顶点 $v\in V$ 的度均为 d，则说该图是 d 正则的(d-regular)。每个 d 正则二分图有 $|L| = |R|$。证明：对每个 d 正则二分图，其相应流网络的最小割的容量为 $|L|$。运用该结论证明：每个 d 正则二分图均有一个势为 $|L|$ 的匹配。

*26.4　压入与重标记算法

在本节中，我们要介绍计算最大流的压入与重标记(push-relabel)方法。目前，许多关于最大流问题的渐近最快速算法就是压入与重标记算法，最大流算法最快速的实际实现都是基于压入与重标记方法的。其他有关流的问题，如最小代价流问题，也可以有效地用压入与重标记方法来解决。本节中要介绍 Goldberg 的“一般性”最大流算法，该算法有一种简单的实现，其运行时

间为 $O(V^2E)$，这是对 Edmonds-Karp 算法的 $O(VE^2)$时间的一种改进。26.5 节中对一般性算法进行进一步的精化，得到另外一种运行时间为 $O(V^3)$的压入与重标记算法。

相对于 Ford-Fulkerson 方法来说，压入与重标记采用的是一种更局部化的方法。它不是检查整个残留网络 $G=(V, E)$来找出增广路径，而是每次仅对一个顶点进行操作，并且仅检查残留网络中该顶点的相邻顶点。此外，与 Ford-Fulkerson 方法不同，压入与重标记算法在执行过程中，并不能保持流守恒特性。但是，该算法保持了一个"前置流(preflow)"，它是一个函数 $f: V\times V\rightarrow \mathbf{R}$，它满足反对称性、容量限制和下列放宽条件的流守恒特性：对所有顶点 $u\in V-\{s\}$，有 $f(V, u)\geqslant 0$。亦即，进入除源点顶点以外的顶点的总净流为非负值。进入顶点 u 的总净流称为进入 u 的余流(excess flow)，由下式给出：

669

$$e(u) = f(V,u) \tag{26.9}$$

对一个顶点 $u\in V-\{s, t\}$，如果 $e(u)>0$，则称顶点 u 溢出。

我们将先描述压入与重标记方法所包含的直观思想，然后再讨论该方法使用的两种操作："压入"(pushing)前置流和"重标记"(relabeling)顶点。最后，我们将给出一般性压入与重标记算法，并分析其正确性和运行时间。

直观思想

如果用液体流的形式来描述的话，压入与重标记方法所包含的直观思想大概最容易懂了：可以把一个流网络 $G=(V, E)$看成是由具有给定容量、且互相连接的管道所组成的系统。把这个比方应用到 Ford-Fulkerson 方法中，可以说网络中的每一条增广路径均引发出一条无分支点、从源点到汇点的额外液体流。Ford-Fulkerson 方法以迭代的方式加入更多的流，直至不能加入时为止。

从直观上来看，一般性压入与重标记算法的思想在某种程度上来说有所不同。和先前一样，图的有向边对应于管道。而作为管道结合点的顶点却有着两个有趣的特性。第一，为了容纳余流，每个顶点均有一个排出管道，它导向能积聚液体的任意大容量水库。第二，每个顶点和它的水库以及所有的管道连接点都处于一个平台上，当算法向前压入时，平台随之逐渐升高。

顶点的高度决定了如何压入流：我们仅仅把流向下压，即从较高顶点向较低顶点压。从较低顶点到较高顶点可能存在一正向网络流，但是对流的压入总是向下压。源点的高度固定为 $|V|$，汇点的高度固定为 0。所有其他顶点的高度开始时都是 0，并逐步增加。算法首先从源点输送尽可能多地流到汇点。它输送的量恰好足够填满从源点出发的每条管道，即它输送的量为割$(s, V-s)$的容量。当流第一次进入一个中间顶点时，它就聚集在该顶点的水库中，并且最终将从那里被继续向下压入。

最终可能会发生下列情况，即离开某顶点 u、且还未被充满的仅有管道所连接的顶点与 u 等高或高于 u。在这种情况下，为了使某溢出顶点 u 摆脱其余流，就必须增加它的高度，即称为"重标记"顶点 u 的操作。我们把 u 的高度增加到比其最低的相邻顶点的高度高 1 个单位(从 u 到该顶点必有一条未充满的管道)。因此，当一个顶点被重标记后，至少存在着一条流出管道，并且可以通过它压入更多的流。

最终，有可能达到汇点的所有流均到达汇点。因为管道服从容量限制，并且通过任何割的流
670 量依然受到割的容量限制，所以这时再没有流能到达汇点了。为了使前置流成为"合法"流，算法继续把顶点重标记到高于源点的固定高度 $|V|$，以便把汇点聚在溢出顶点的水库中的余流送回给源。正如我们将要看到的那样，一旦所有的水库均为空后，前置流不仅是"合法"的流，而且也是最大流了。

基本的操作

从前面的讨论中，我们知道压入与重标记算法中要执行两种基本操作：把流的余量从一顶点压入到它的一个相邻顶点，以及重标记一个顶点。采用这两种操作中的哪一种取决于顶点的高度。现在，我们来给出顶点高度的准确定义。

设 $G=(V, E)$是一个流网络，其源点为 s，汇点为 t。设 f 是 G 的一个前置流。如果函数 $h: V \to \mathbf{N}$ 满足 $h(s)=|V|$，$h(t)=0$，且对每条残留边$(u, v)\in E_f$，有

$$h(u) \leqslant h(v)+1$$

则该函数为高度函数[⊖]。我们立即可得下列引理。

引理 26.13　设 $G=(V, E)$是一个流网络，f 是 G 的前置流，h 是定义在 V 上的高度函数。对任意两顶点 u，$v\in V$，如果 $h(u)>h(v)+1$，则(u, v)不是残留图中的边。 ■

压入操作

如果 u 是某溢出顶点，$c_f(u, v)>0$ 且 $h(u)=h(v)+1$，则可以应用基本操作 PUSH(u, v)。下列伪代码对隐式给出的网络 $G=(V, E)$中的前置流 f 进行更新。它假定对给定的 c 和 f 可以在常数时间内计算出残留容量。存储于顶点 u 的余流用 $e[u]$表示，u 的高度用 $h[u]$表示。符号 $d_f(u, v)$是存储能够从 u 压入到 v 的流量的一个临时变量。 671

```
PUSH(u, v)
1  ▷ Applies when: u is overflowing, c_f(u, v)>0, and h[u]=h[v]+1.
2  ▷ Action: Push d_f(u, v)=min(e[u], c_f(u, v)) units of flow from u to v.
3  d_f(u, v) ← min(e[u], c_f(u, v))
4  f[u, v] ← f[u, v]+d_f(u, v)
5  f[v, u] ← −f[u, v]
6  e[u] ← e[u]− d_f(u, v)
7  e[v] ← e[v]+d_f(u, v)
```

PUSH 的代码执行如下。假定顶点 u 有一个正的余量 $e[u]$，且(u, v)的残留容量为正。我们能够从 u 传输 $d_f(u, v)=\min(e[u], c_f(u, v))$单位的流到顶点 v，并不会使 $e[u]$变成负值或超出容量 $c(u, v)$。$d_f(u, v)$的值是在第 3 行中计算的。通过第 4～5 行中更新 f 和第 6～7 行中更新 e。因此，如果在调用 PUSH 之前 f 是前置流，则调用后依然为前置流。

注意，PUSH 的代码中没有涉及 u 和 v 的高度，我们将避免涉及它们，除非 $h[u]=h[v]+1$。因此，余流仅在高度差为 1 时才被向下压入。由引理 26.13 可知，两个顶点的高度差大于 1 时，它们之间不可能存在残留边，所以在高度差大于 1 时，只要属性 h 确实是一个高度函数，允许将流向下压入则毫无意义。

我们称操作 PUSH(u, v)是一个从 u 到 v 的压入。如果压入操作适用于离开顶点 u 的某边(u, v)，则也可以说压入操作适用于 u。如果边(u, v)变为饱和(压入后 $c_f(u, v)=0$)，则该压入是饱和压入，否则就是一个不饱和压入。如果一条边是饱和的，则它不出现在残留网络中。一个简单的引理说明了不饱和压入的结果。

引理 26.14　在一次从 u 到 v 的不饱和压入操作之后，顶点 u 不再溢出。

证明：因为压入是不饱和的，实际压入的流量 $d_f(u, v)$必定等于压入之前的 $e[u]$。由于要

⊖ 高度函数一般称作“距离函数”，而且一个顶点的高度叫做“距离标号”。使用“高度”一词，是因为它更能够体现算法背后的直观含义。我们仍然用“relabel”一词来表示增加某一顶点的高度。一个顶点的高度是与其到汇点 t 的距离相关的，对转置 G^T 进行广度优先搜索可以找到。

从 $e[u]$中减去这个量，在压入后，它即变为 0。 ■

重标记操作

如果 u 是溢出顶点，且对所有边 $(u, v) \in E_f$ 有 $h[u] \leqslant h[v]$，则可以应用基本操作 RELABEL(u)。换句话说，已知溢出顶点 u，如果对从 u 到 v 还存在残留容量的每一个顶点 v，
672 由于 v 的高度不在 u 之下而使我们不能把流从 u 压入到 v，则此时可以重标记溢出顶点 u。(回忆一下定义可知，源点 s 或汇点 t 都不可能是溢出顶点，因此 s 和 t 都不能被重标记。)

```
RELABEL(u)
1  ▷ Applies when: u is overflowing and for all v∈V such that(u, v)∈E_f,
                   we have h[u]≤h[v].
2  ▷ Action: Increase the height of u.
3  h[u] ← 1+min{h[v] : (u, v)∈E_f}
```

当我们调用操作 RELABEL(u)时，说顶点 u 被重标记了。注意当 u 被重标记时，E_f 必须至少包含一条离开 u 的边，以使上述代码中的最小化运算不会在空集上执行。从 u 为溢出顶点的假设可知，这一情况是成立的。由于 $e[u]>0$，有 $e[u]=f[V, u]>0$，因而至少存在一个顶点 v 满足 $f[v, u]>0$。但是

$$c_f(u,v) = c(u,v) - f[u,v] = c(u,v) + f[v,u] > 0$$

这说明$(u, v)\in E_f$。因此，操作 RELABEL(u)使 u 在高度函数约束下，具有所允许的最大高度。

一般性算法

一般性压入与重标记算法使用下列子程序，在流网络中建立一个初始前置流。

```
INITIALIZE-PREFLOW(G, s)
1  for each vertex u ∈V[G]
2       do h[u] ← 0
3          e[u] ← 0
4  for each edge(u, v)∈E[G]
5       do f[u, v] ← 0
6          f[v, u] ← 0
7  h[s] ← |V[G]|
8  for each vertex u ∈Adj[s]
9       do f[s, u] ← c(s, u)
10         f[u, s] ← −c(s, u)
11         e[u] ← c(s, u)
12         e[s] ← e[s]−c(s, u)
```

673 INITIALIZE-PREFLOW 建立的初始前置流 f 定义为

$$f[u,v] = \begin{cases} c(u,v) & \text{如果 } u = s \\ -c(v,u) & \text{如果 } v = s \\ 0 & \text{否则} \end{cases} \tag{26.10}$$

即每条离开源点 s 的边被充满，而其他所有边不运载任何流。对每个与源点邻接的顶点 v，开始时有$e[v]=c(s, v)$，$e[s]$被初始化为这些容量的和的负值。一般性算法中也从高度函数 h 开始。h 由下式给出：

$$h[u] = \begin{cases} |V| & \text{如果 } u = s \\ 0 & \text{否则} \end{cases}$$

这是一个高度函数，因为满足 $h[u]>h[v]+1$ 的边 (u, v) 仅是那些满足 $u=s$ 的边，并且那些边是饱和的，这意味着它们不在残留网络中。

在对流进行初始化，后跟一系列的压入和重标记操作(这些操作可以以任意的顺序执行)后，即可得到 GENERIC-PUSH-RELABEL 算法：

```
GENERIC-PUSH-RELABEL(G)
1  INITIALIZE-PREFLOW(G, s)
2  while there exists an applicable push or relabel operation
3       do select an applicable push or relabel operation and perform it
```

下面的引理告诉我们：只要存在着溢出顶点，那么两个基本操作中至少有一个是适用的。

引理 26.15(溢出的顶点可以被压入或重标记)　设 $G=(V, E)$ 是一个流网络，源点为 s，汇点为 t，f 为一前置流。设 h 是 f 的任意高度函数。如果 u 是任意溢出顶点，则压入操作或重标记操作适用于该顶点。

证明：对任意残留边 (u, v)，由于 h 是高度函数，所以有 $h(u)\leqslant h(v)+1$。如果压入操作不适用于 u，则对所有残留边 (u, v)，必定有 $h(u)<h(v)+1$ 成立。这说明 $h(u)\leqslant h(v)$，因此重标记操作适用于 u。■

压入与重标记方法的正确性

为了说明一般性压入与重标记算法解决了最大流问题，我们将首先证明如果算法终止，则前置流 f 为一个最大流。接着，再证明算法能够终止。下面先来看看高度函数 h。 674

引理 26.16(顶点高度不会减小)　GENERIC-PUSH-RELABEL 在流网络 $G=(V, E)$ 上的执行过程中，对每个顶点 $u\in V$，其高度 $h[u]$ 不会减小。此外，对顶点 u 应用重标记操作时，其高度 $h[u]$ 至少增加 1。

证明：因为顶点的高度仅在重标记操作中变化，所以只要证明引理的第二个论断就足够了。如果顶点 u 将被重标记，则对所有满足 $(u, v)\in E_f$ 的顶点 v，有 $h[u]\leqslant h[v]$；这说明 $h[u]<1+\min\{h[v]: (u, v)\in E_f\}$，因此该操作必使 $h[u]$ 增加。■

引理 26.17　设 $G=(V, E)$ 是一个流网络，其源点为 s，汇点为 t。在 GENERIC-PUSH-RELABEL 对 G 的执行过程中，属性 h 始终为高度函数。

证明：通过对执行的基本操作次数进行归纳来证明。开始时，h 是一个高度函数，这一点已经在上面证明过了。

我们断言如果 f 是高度函数，则执行操作 RELABEL(u)后，h 仍然是高度函数。如果我们观察一下离开 u 的残留边 $(u, v)\in E_f$，可以发现操作 RELABEL(u)保证其后有 $h[u]\leqslant h[v]+1$。现在来考虑一下进入 u 的残留边 (w, u)。由引理 26.16 可知，在操作 RELABEL(u)之前，$h[w]\leqslant h[u]+1$ 蕴涵着在在该操作之后，有 $h[w]<h[u]+1$。因此操作 RELABEL(u)使 h 仍为高度函数。

现在来考虑一个操作 PUSH(u, v)。这个操作可能会向 E_f 中加入边 (v, u)，也可能会从 E_f 中删除边 (u, v)。在前一种情况中，有 $h[v]=h[u]-1<h[u]+1$，因而 h 仍然是一个高度函数。在后一种情况中，从残留网络中删除边 (u, v)就删除了相应的约束，此时 h 仍然为一个高度函数。■

下列引理给出了高度函数的一个重要性质。

引理 26.18　设 $G=(V, E)$ 是一个流网络，其源点为 s，汇点为 t。设 f 是 G 的一前置流，h 是定义在 V 上的高度函数。则在残留网络 G_f 中不存在从源点 s 到汇点 t 的路径。

证明：为了引入矛盾，假定在 G_f 中，存在着一条从 s 到 t 的路径 $p=\langle v_0, v_1, \cdots, v_k\rangle$，其

中$v_0=s$，$v_k=t$。不失一般性，假设p是一条简单路径，所以$k<|V|$。对$i=0, 1, \cdots, k-1$，边$(v_i, v_{i+1})\in E_f$。因为h是高度函数，所以对$i=0, 1, \cdots, k-1$，有$h(v_i)\leqslant h(v_{i+1})+1$。把
675 路径p上的这些不等式联合起来，可得$h(s)\leqslant h(t)+k$。但是，由于$h(t)=0$，有$h(s)\leqslant k<|V|$，这与我们对高度函数的要求$h(s)=|V|$相矛盾。■

现在我们可以证明，如果一般性压入与重标记算法终止，则它计算出来的前置流为最大流。

定理 26.19(一般性压入与重标记算法的正确性)　当算法 GENERIC-PUSH-RELABEL 在具有源点s和汇点t的流网络$G=(V, E)$上运行时，若算法终止，则它计算出的前置流f为G的最大流。

证明：我们使用如下的循环不变式：

在 GENERIC-PUSH-RELABEL 中，每次执行第 2 行的 while 循环测试时，f都是一个前置流。

初始化：INITIALIZATION-PREFLOW 初始化f为前置流。

保持：第 2～3 行的 while 循环内部只有压入和重标记操作。重标记操作只影响高度属性，对流值无作用；因此，它们并不影响f是否为前置流。正如前面讨论过的那样，如果f在压入操作之前为前置流，那么操作结束后它仍然保持为前置流。

终止：在终止时，$V-\{s, t\}$中的每个顶点的余流必定为 0，根据引理 26.15 和引理 26.17 以及f始终为前置流的循环不变式，网络中没有溢出顶点。所以，f是一个流。因为h是个高度函数，根据引理 26.18 可知，在残留网络G_f中，从s到t之间没有路径。根据最大流最小割定理(定理 26.7)可知，f是最大流。■

对压入与重标记方法的分析

为了说明一般性压入与重标记算法确实会终止，我们将给出它执行的操作次数的界。对于三种类型的操作(重标记、饱和压入和不饱和压入)中的每一种，分别给出其界。有了这些界后，构造一个运行时间为$O(V^2E)$的算法就成为一个简单的问题了。在开始分析之前，先来证明一个重要的引理。

引理 26.20　设$G=(V, E)$是源点为s，汇点为t的一个流网络，且f是G的前置流。则对
676 任意溢出顶点u，在残留网络G_f中存在着一条从u到s的简单路径。

证明：对任意溢出顶点u，设$U=\{v: G_f$中存在一条从u到v的简单路径$\}$，且为了引入矛盾，假定$s\notin U$。设$\overline{U}=V-U$。

我们认为，对于每对顶点$v\in U$，$w\in\overline{U}$，有$f(w, v)\leqslant 0$。为什么呢？这是因为，如果$f(w, v)>0$的话，则$f(v, w)<0$。这说明有$c_f(v, w)=c(v, w)-f(v, w)>0$。因此存在一条边$(v, w)\in E_f$，因而$G_f$中必有一条形如$u\rightsquigarrow v\to w$的简单路径，这与我们对$w$的选择相矛盾。

因此，必有$f(\overline{U}, U)\leqslant 0$，这是由于在这一隐含的求和式中，每一项都为非正值，因而可以得出结论：

$$\begin{aligned} e(U) &= f(V,U) && \text{(根据等式(26.9))} \\ &= f(\overline{U},U)+f(U,U) && \text{(根据引理 26.1,第 3) 部分)} \\ &= f(\overline{U},U) && \text{(根据引理 26.1,第 1) 部分)} \\ &\leqslant 0 \end{aligned}$$

对所有属于$V-\{s\}$的顶点，其余流为非负；因为我们假定$U\subseteq V-\{s\}$，因此对所有顶点$u\in U$，必有$e(u)=0$。特别地，$e(u)=0$，这与我们假定u是溢出顶点相矛盾。■

下个引理给出了顶点高度的界，其推论给出了执行全部重标记操作的次数的界。

引理 26.21　设 $G=(V, E)$是一个流网络，其源点为 s，汇点为 t。GENERIC-PUSH-RELABEL 在 G 上执行过程中的任何时刻，对所有的顶点 $u\in V$，都有 $h[u]\leqslant 2|V|-1$。

证明：因为根据定义，源点 s 和汇点 t 均不是溢出顶点，所以它们的高度不会改变。因此总是有$h[s]=|V|$ 和 $h[t]=0$，都不会大于 $2|V|-1$。

现在来考虑任意顶点 $u\in V-\{s, t\}$。初始时，$h[u]=0\leqslant 2|V|-1$。下面将证明在每次重标记操作后，仍然有 $h[u]\leqslant 2|V|-1$。当 u 被重标记时，正处于溢出状态，由引理 26.20 可知，G_f 中从 u 到 s 存在一条简单路径 p。设 $p=\langle v_0, v_1, \cdots, v_k\rangle$，其中 $v_0=u$，$v_k=s$，且 $k\leqslant |V|-1$，因为 p 是简单路径。对 $i=0, 1, \cdots, k-1$，有$(v_i, v_{i+1})\in E_f$，则根据引理 26.17，有 $h[v_i]\leqslant h[v_{i+1}]+1$。在路径 p 上扩展这些不等式，可得 $h[u]=h[v_0]\leqslant h[v_k]+k\leqslant h[s]+(|V|-1)=2|V|-1$。■

推论 26.22(关于重标记操作的界)　设 $G=(V, E)$是一个流网络，其源点为 s，汇点为 t。在 GENERIC PUSH-RELABEL 对 G 的执行过程中，对每个顶点执行重标记操作的次数至多为 $2|V|-1$，全部重标记操作执行次数至多为$(2|V|-1)(|V|-2)<2|V|^2$。 677

证明：只有属于 $V-\{s, t\}$中的 $|V|-2$ 个顶点可能被重标记。设 $u\in V-\{s, t\}$。操作 RELABEL(u)增加了 $h[u]$的值。$h[u]$的初始值为 0，并且由引理 26.21 可知，其值不超过 $2|V|-1$。因此，每个顶点 $u\in V-\{s, t\}$至多被重标记 $2|V|-1$ 次，因此，所执行的全部重标记操作次数为$(2|V|-1)(|V|-2)<2|V|^2$。■

引理 26.21 也有助于限制饱和压入的次数。

引理 26.23(关于饱和压入的界)　在 GENERIC-PUSH-RELABEL 对任意流网络 $G=(V, E)$的执行过程中，饱和压入的次数至多为 $2|V||E|$。

证明：对任意顶点对 $u, v\in V$，统计从 u 到 v 以及从 v 到 u 的饱和压入次数，称为 u，v 之间的饱和压入。如果存在这样的压入，那么(u, v)和(v, u)至少有一边实际存在于 E 中。现在，假定发生了从 u 到 v 的饱和压入。此时，$h[v]=h[u]-1$。为使以后产生从 u 到 v 的另一次压入，算法必须首先将流从 v 压入到 u，但直到 $h[v]=h[u]+1$ 才会发生。因为 $h[u]$是非递减的，所以为了达到$h[v]=h[u]+1$，$h[v]$的值至少增加 2。同样，在从 v 到 u 的饱和压入之间，$h[u]$必须至少增加 2。高度在开始时为 0，并且，根据引理 26.21 可知，高度始终不会超过 $2|V|-1$，这说明了任意一顶点的高度增加 2 的次数少于 $|V|$ 次。因为 u 和 v 之间的两次饱和压入中，$h[u]$和 $h[v]$必定至少有一个增加了 2，在 u 和 v 之间的饱和压入的次数少于 $2|V|$。乘以边数就得到饱和压入总次数为少于 $2|V||E|$ 的界。■

下列引理给出了一般性压入与重标记算法中不饱和压入次数的界。

引理 26.24(关于不饱和压入的界)　在 GENERIC-PUSH-RELABEL 对任意流网络 $G=(V, E)$的执行过程中，不饱和压入的次数至多为：$4|V|^2(|V|+|E|)$。

证明：定义一个势函数 $\Phi=\sum_{v:e(v)>0} h[v]$。初始时，$\Phi=0$，且 Φ 值在重标记、饱和压入和不饱和压入后都可能发生变化。我们将给出饱和压入和重标记操作在增加 Φ 时所提供量的界，然后说明每次不饱和压入至少使 Φ 减少 1。利用这些界，可以推导出不饱和压入操作次数的上界。

下面来分析一下 Φ 的值可能增加的两种方式。第一种，重标记一个顶点 u 时 Φ 至多增加 $2|V|$，这是因为求和所依赖的集合是相同的，并且 u 不能被重标记到超过其可能的最大高度(由引理 26.21 可知，该高度至多为 $2|V|-1$)。第二种，从顶点 u 到顶点 v 的饱和压入，可以给 678
Φ 少于 $2|V|$ 的增加，因为没有哪个顶点的高度发生了变化(除了 v，其高度不超过 $2|V|-1$，可能溢出)。

现在，我们来说明从 u 到 v 的不饱和压入对 Φ 至少减小 1。为什么呢？在不饱和压入前，u 是溢出的，v 可能也可能不溢出。根据引理 26.14，u 在压入后不再溢出。而且，在压入之后 v 必定是溢出的(除非它是源点)。因而，势函数 Φ 减去了 $h[u]$，而增加了 0 或 $h[v]$。因为 $h[u]-h[v]=1$，则净效果为势函数至少减小 1。

因此，在算法的执行过程中，由于根据推论 26.22 和引理 26.23 可知，重标记和饱和压入对 Φ 总量的增加不超过$(2|V|)(|2V|^2)+(2|V|)(2|V||E|=4|V|^2(|V|+|E|)$。由于 $\Phi\geqslant 0$，所以全部的减小量，即非饱和压入的全部执行次数至多为 $4|V|^2(|V|+|E|)$。 ■

给出了重标记、饱和压入以及不饱和压入的界后，我们已经为下面分析 GENERIC-PUSH-RELABEL 过程以及基于压入与重标记方法的任何算法作好了充分的准备。

定理 26.25　在 GENERIC-PUSH-RELABEL 对任意流网络 $G=(V, E)$的执行过程中，基本操作的执行次数为 $O(V^2E)$。

证明：根据推论 26.22 和引理 26.23 以及引理 26.24 立即可得。 ■

因而，算法在 $O(V^2E)$次操作后终止。余下的就是给出每个操作高效的实现方法和选择合适的操作执行。

推论 26.26　对任意流网络 $G=(V, E)$，存在一种压入与重标记算法的实现，其运行时间为 $O(V^2E)$。

证明：练习 26.4-1 要求说明如何实现一般性算法，使其开销为：每个重标记操作运行时间为$O(V)$，每个压入操作为 $O(1)$。而且，要求设计出一种数据结构，以便能够在 $O(1)$时间内选
679 择适用的操作。由此可知推论成立。 ■

练习

26.4-1　说明如何实现一般性压入与重标记算法，使得每次重标记操作需要 $O(V)$时间，每次压入操作需要 $O(1)$时间，选择可应用的操作需要 $O(1)$时间，而整个算法的时间为 $O(V^2E)$。

26.4-2　证明：在执行全部的 $O(V^2)$次重标记操作时，一般性压入与重标记算法所需的全部运行时间仅为 $O(VE)$。

26.4-3　假定运用压入与重标记算法找出了流网络 $G=(V, E)$中的最大流。试给出一种快速算法来找出 G 的最小割。

26.4-4　写出一个有效的压入与重标记算法，以找出一个二分图的最大匹配。

26.4-5　假定在流网络 $G=(V, E)$中，所有边的容量都属于集合$\{1, 2, \cdots, k\}$。试用 $|V|$，$|E|$ 和 k 来分析一般性压入与重标记算法的运行时间(提示：一条边在成为饱和边前能承受多少次不饱和压入?)

26.4-6　证明 INITIALIZE-PREFLOW 的第 7 行可以改为 $h[s]\leftarrow|V[G]|-2$，这并不影响一般性压入与重标记算法的正确性或其渐近意义上的性能。

26.4-7　设 $\delta_f(u, v)$为残留网络 G_f 中从 u 到 v 的距离(边数)。证明 GENERIC-PUSH-RELABEL 保持下列性质：若 $h[u]<|V|$，则 $h[u]\leqslant\delta_f(u, t)$；并且 $h[u]\geqslant|V|$，则 $h[u]-|V|\leqslant\delta_f(u, s)$。

*26.4-8　如上一个练习题中一样，设 $\delta_f(u, v)$为残留网络 G_f 中从 u 到 v 的距离。说明如何修改一般性压入与重标记算法以保持下列性质：若 $h[u]<|V|$，则 $h[u]=\delta_f(u, t)$；并且
680 若 $h[u]\geqslant|V|$，则 $h[u]-|V|=\delta_f(u, s)$。为保持这一性质，算法的全部运行时间为 $O(VE)$。

26.4-9　证明对于 $|V|\geqslant 4$，在流网络 $G=(V, E)$上运行 GENERIC-PUSH-RELABEL 所执行的非饱和压入操作的次数至多为 $4|V|^2|E|$。

*26.5　重标记与前移算法

压入与重标记方法允许我们以任意次序执行基本操作。但是，通过仔细选择执行次序和有效安排网络的数据结构，可以用比推论 26.26 给出的 $O(V^2E)$更少的运行时间来解决最大流问题。下面就来研究重标记与前移算法，这是一种运行时间为 $O(V^3)$的压入与重标记算法。从渐近意义上来看，对于稠密网络它至少不弱于 $O(V^2E)$。

重标记与前移算法设置了一张网络中的顶点表，算法从表的前端开始扫描表，反复选出溢出顶点 u，然后"排除"它，即反复执行压入和重标记操作，直至顶点 u 不再存在正的余流。当某个顶点被重标记时，它就被移到表的前端(所以算法名为"重标记与前移")，算法又重新开始扫描。

重标记与前移算法的正确性及其性能分析与概念"容许"边(admissible edge)有关：即残留网络中压入的流经过的那些边。在证明关于容许网络的几条性质后，我们将讨论排除操作，最后给出并分析重标记与前移算法。

容许边和容许网络

设 $G=(V, E)$是一个流网络，其源点为 s，汇点为 t。f 是 G 的前置流，h 是高度函数，则如果 $c_f(u, v)>0$ 且 $h(u)=h(v)+1$，就说(u, v)是容许边。否则，(u, v)是非容许边。容许网络为 $G_{f,h}=(V, E_{f,h})$，其中 $E_{f,h}$为容许边的集合。

容许网络由能被压入的流所通过的那些边组成。下列引理说明这种网络是一个有向无回路图(dag)。

引理 26.27(容许网络中不包含回路)　如果 $G=(V, E)$是一个流网络，f 是 G 的一个前置流，且 h 是 G 上的高度函数，则容许网络 $G_{f,h}=(V, E_{f,h})$中不包含回路。 681

证明：通过引入矛盾来证明该引理。假定 $G_{f,h}$包含一个回路 $p=\langle v_0, v_1, \cdots, v_k\rangle$，其中 $v_0=v_k$，且 $k>0$。由于 p 中的每条边均为容许边，所以对 $i=1, 2, \cdots, k$，有 $h(v_{i-1})=h(v_i)+1$。对这些等式沿回路求和，得：

$$\sum_{i=1}^{k} h(v_{i-1}) = \sum_{i=1}^{k} (h(v_i)+1) = \sum_{i=1}^{k} h(v_i) + k$$

因为回路 p 中的每个顶点在每个求和式中仅出现一次，由此推得 $0=k$，与假设矛盾。 ■

下面两条引理说明执行压入和重标记操作对容许网络的影响。

引理 26.28　设 $G=(V, E)$是一个流网络，f 是 G 的一个前置流，且 h 是 G 上的高度函数。如果顶点 u 是溢出顶点且(u, v)是容许边，则可采用 PUSH(u, v)。该操作不会建立任何新的容许边，但它可能使(u, v)变为非容许边。

证明：根据容许边的定义可知，能够把流从 u 压入到 v。由于 u 是溢出顶点，所以操作 PUSH(u, v)适用。把流从 u 压入到 v 这一操作唯一可能建立的新的残留边是边(v, u)。但由于 $h(v)=h(u)-1$，所以边(v, u)不可能变成容许边。如果该操作是饱和压入，则操作执行后，$c_f(u, v)=0$ 且(u, v)成为非容许边。 ■

引理 26.29　设 $G=(V, E)$是一个流网络，f 是 G 的一个前置流，且 h 是 G 上的高度函数。如果顶点 u 是溢出顶点，并且不存在离开 u 的容许边。则此时 RELABEL(u)适用。在执行重标记操作后，至少存在一条离开 u 的容许边，但不会有进入 u 的容许边。

证明：如果 u 是溢出顶点，则由引理 26.15 可知，要么有压入操作、要么有重标记操作适用于

该顶点。如果不存在离开 u 的容许边，就不可能有从 u 出发向前压入的流，此时 RELABEL(u)适用。在重标记操作后，$h[u]=1+\min\{h[v]:(u,v)\in E_f\}$。因此，如果 v 是集合中满足最小值的顶点，则边(u,v)成为容许边。这样，在重标记操作后，至少存在一条离开 u 的容许边。

为了证明在重标记操作后，不会有进入 u 的容许边，假定存在一顶点 v，使得(u,v)为容许边。那么在执行重标记操作后，$h[v]=h[u]+1$。所以在重标记前有 $h[v]>h[u]+1$。但根据引
682 理 26.13 可知，在高度差大于 1 的两个顶点之间，不可能存在残留边。此外，重标记某顶点并不会改变残留网络。因此，(v,u)不属于残留网络，因而它不可能属于容许网络。■

相邻表

重标记与前移算法中的边都被放入"相邻表"中。如果给定流网络 $G=(V,E)$，对顶点 $u\in V$，其相邻表 $N[u]$是一个关于 G 中 u 的相邻顶点的单链表。因此，如果$(u,v)\in E$ 或$(v,u)\in E$，则顶点 v 出现在表 $N[u]$中。相邻表 $N[u]$仅仅包含那些可能存在残留边(u,v)的顶点 v。$N[u]$中的第一个顶点由 $head[N[u]]$指出。相邻表中 v 的下一个顶点由指针 $next\text{-}neighbor[v]$指出；如果 v 是相邻表中的最后一个顶点，则该指针为 NIL。

重标记与前移算法在其执行过程中，按某确定的顺序循环访问每个相邻表。对每个顶点 u，域 $current[u]$指向 $N[u]$中当前被考察的顶点。$current[u]$开始时被置为 $head[N[u]]$。

溢出顶点的排除

一个溢出顶点 u 通过下列方式排除：把该顶点的所有余流通过容许边压入到相邻顶点，必要时重标记顶点 u，使离开 u 的边变成容许边。代码如下：

```
DISCHARGE(u)
1  while e[u]>0
2      do v ← current[u]
3         if v=NIL
4            then RELABEL(u)
5                 current[u] ← head[N[u]]
6         elseif c_f(u, v)>0and h[u]=h[v]+1
7           then PUSH(u, v)
8         else current[u] ← next-neighbor[v]
```

图 26-9 描述了第 1～8 行 while 循环中的几次迭代。只要 u 还有正的余流，循环将一直进行下去。在每一次迭代中，都将根据相邻表 $N[u]$中当前顶点 v 的情况，选择三个操作中的一个加以执行：

1)如果 v 为 NIL，说明运行到了 $N[u]$的末顶点。第 4 行重标记顶点 u，第 5 行把 u 的当前邻接点复位为 $N[u]$中的第一个顶点(下面的引理 26.30 将说明重标记操作适用于这种情况)。

2)如果 v 不是 NIL，且(u,v)是一条容许边(由第 6 行中的测试所决定)，则第 7 行把 u 的一
683 些(也可能是全部)余流压入到顶点 v。

3)如果 v 不是 NIL，且(u,v)是非容许边，则第 8 行使 $current[u]$在相邻表 $N[u]$中向前压入一个顶点。

注意，如果对一个溢出顶点 u 调用 DISCHARGE，则由 DISCHARGE 执行的最后一个操作必定是从 u 出发的一个压入操作。为什么呢？因为只有当 $e[u]$变为 0 时算法才会终止，而提取操作和使指针 $current[u]$增值这一操作都不会影响 $e[u]$的值。

我们必须保证当 DISCHARGE 调用 PUSH 或 RELABEL 时，相应的操作适用。下面的引理证明了这一事实。

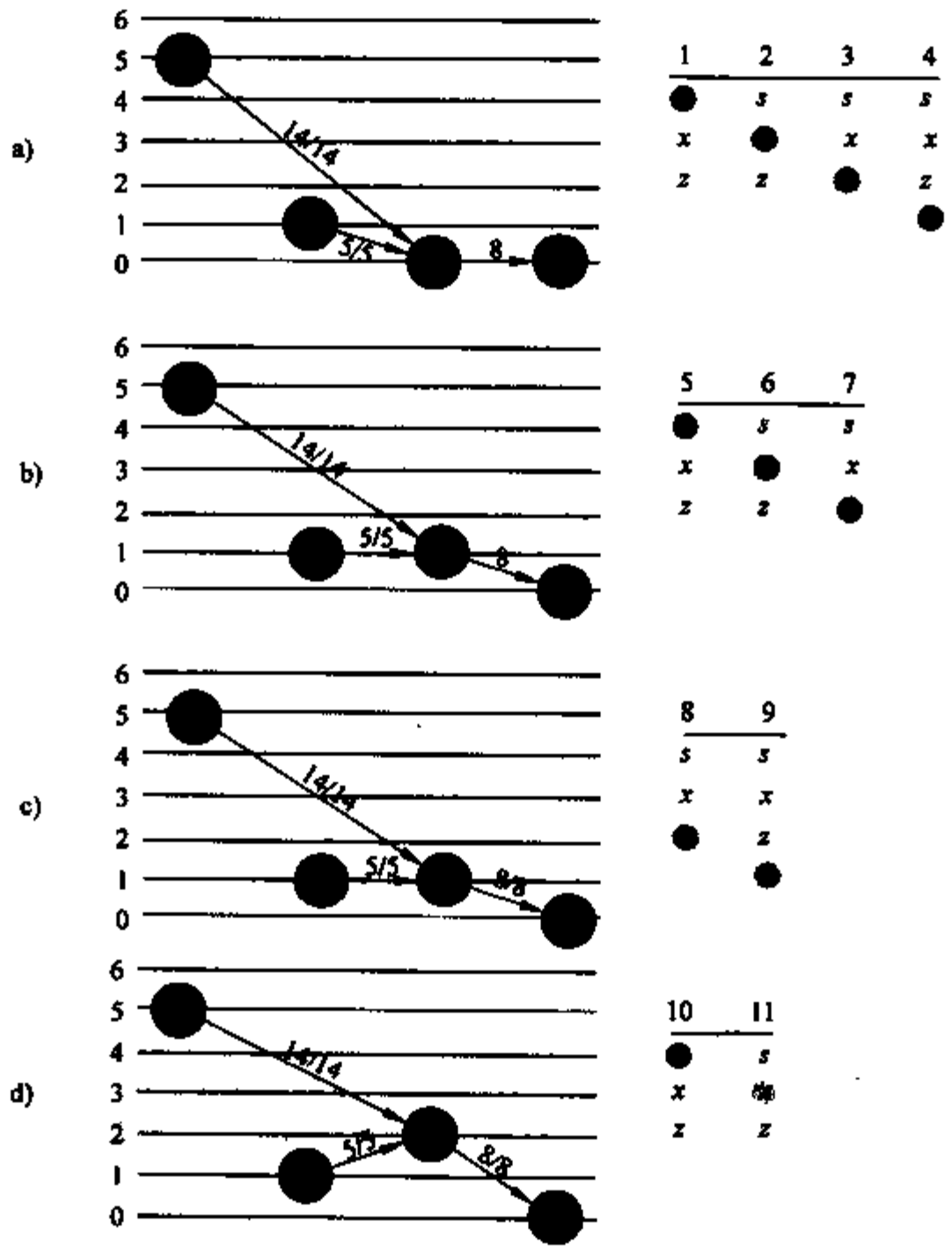


图 26-9　排除顶点 y，DISCHARGE 的 while 循环通过 15 次迭代，将所有余流从 y 压出。图中只显示了 y 的邻接点以及进入或离开 y 的边。在每一个子图中，顶点中的数字显示了第一次迭代开始时的余流，而且每个顶点都显示在标有其高度的位置上。右边部分显示的是每次迭代开始是的邻接表 $N[y]$，迭代次数被标在顶端，标有阴影的邻接点是 $current[y]$。a)初始时，有 19 个单位的余流需从 y 顶点压出，且 $current[y]=s$。1、2、3 轮迭代因为没有离开 y 的容许边只是压入 $current[y]$。在第 4 轮迭代中，$current[y]=$NIL(在邻接表下用阴影标出)，因而 y 被重标记了，而且 $current[y]$被重新复位到邻接表的头部。b)被重标记以后，y 的高度为 1。在第 5～6 次迭代中，边(y, s)和边(y, x)为非容许，但是 8 单位的余流在第 7 轮迭代中从 y 压入到 x。因为这次压入，在迭代中 $current[y]$将不被压入。c)因为第 7 轮迭代的压入饱和了边(y, x)，在第 8 轮迭代中它将变为非容许的。在第 9 轮迭代中，$current[y]=$NIL，因而顶点 y 被再次重标记且 $current[y]$被复位。d)在第 10 轮迭代中(y, s)为非容许边，但是 5 单位的余流在第 11 轮的迭代中从 y 压入到 x。e)因为在第 11 轮的迭代中 $current[y]$没有被压入，所以在第 12 轮迭代时(y, x)为非容许的。在第 13 轮迭代中，(y, z)为非容许边，而第 14 次迭代重标记顶点 y 且复位 $current[y]$。f)在第 15 次迭代中，6 单位的余流从 y 压入到 s。g)顶点 y 现在没有余流，因而 DISCHARGE 终止。在这个例子中，DISCHARGE 开始和结束时，将当前指针指向邻接表的头部，然而通常都不需如此

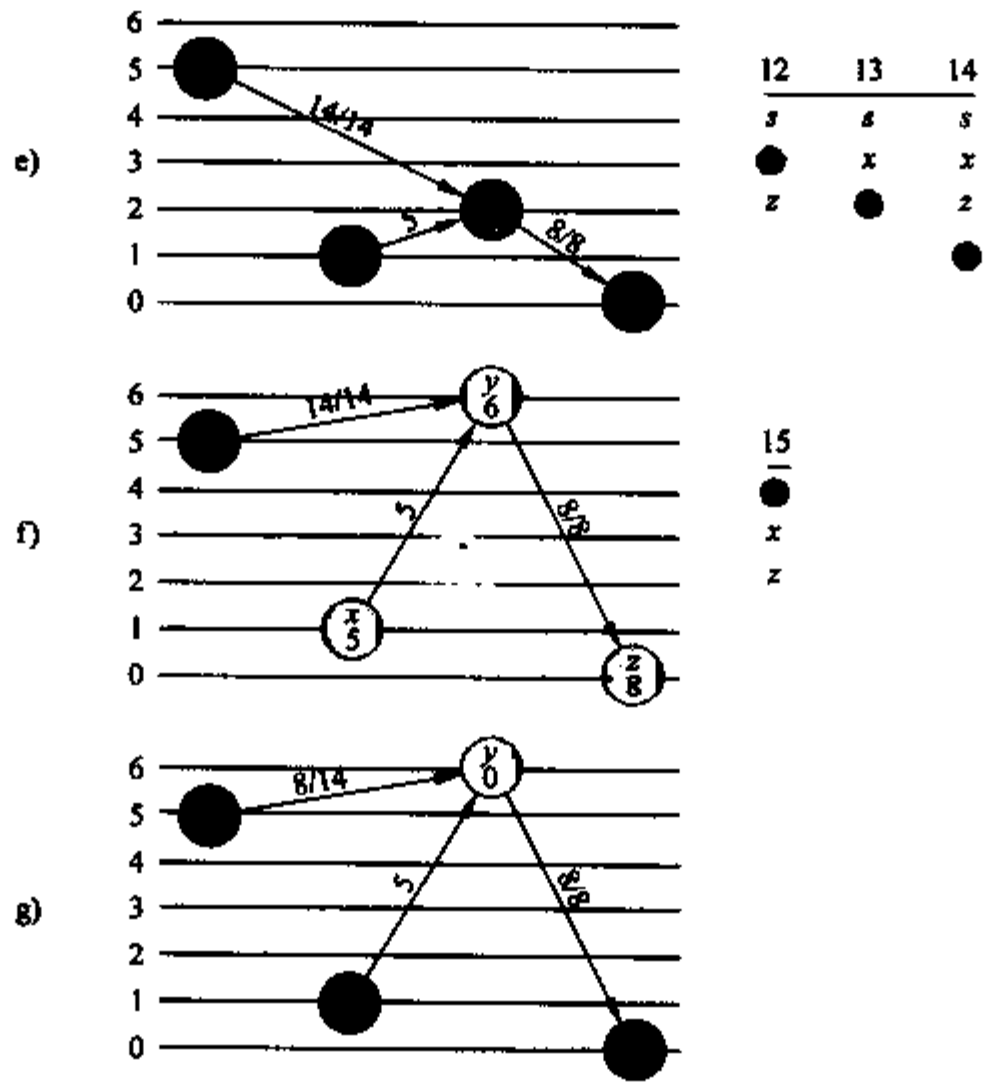


图 26-9　（续）

引理 26.30　如果 DISCHARGE 在第 7 行中调用 PUSH(u, v)，则此时压入操作适用于(u, v)。如果 DISCHARGE 在第 4 行调用 RELABEL(u)，则此时重标记操作适用于顶点 u。

证明：第 1 行和第 6 行中的测试保证了仅在压入操作适用时，才会发生压入操作。这样我们就证明了引理的第一个结论。

为了证明第二个结论，根据第 1 行中的测试和引理 26.29，仅需要证明所有离开 u 的边为非容许边。注意，当反复调用 DISCHARGE(u)时，指针 $current[u]$沿表 $N[u]$向前移动。每一"趟"开始于 $N[u]$的头并在 $current[u]$=NIL 时结束，这时 u 被重标记，然后又开始新的一趟扫描。在某趟中当指针 $current[u]$经过一顶点 $v \in N[u]$时，由第 6 行中的测试可知边(u, v)一定被看作非容许边。因此在该趟扫描完成时，每条离开 u 的边都在该趟中某个时刻被确定为非容许边。最关键的是，在该趟结束时，每条离开 u 的边依然是非容许边。为什么呢？因为由引理 26.28 可知，压入并不能生成任何容许边，更不用说生成离开 u 的边了。因此，任何容许边必定由一个重标记操作所生成。但是，顶点 u 在该趟中并没有被重标记，根据引理 26.29，在该趟中被重标记的任何其他顶点 v 都没有进入该顶点的容许边。因此，在该趟结束时，所有离开 u 的边依然为非容许边，定理得证。 ■

重标记与前移算法

在重标记与前移算法中，我们设置了包含 $V-\{s, t\}$中所有顶点的链表 L。该链表的一个重要性质是根据容许网络对表中的所有顶点进行了拓扑排序。这在下面的循环不变式中可以看到。(引理 26.27 已证明了容许网络是一个有向无回路图)

684
~
686

在重标记与前移算法的伪码中，假设对每个顶点 u 已经建立了相邻表 $N[u]$，并假定 $next[u]$指向 L 中 u 的后继顶点。若 u 是表中的最后一个顶点，则 $next[u]$=NIL。

```
RELABEL-TO-FRONT(G, s, t)
 1  INITIALIZE-PREFLOW(G, s)
```

```
2  L ← V[G]−{s, t}, in any order
3  for each vertex u∈V[G]−{s, t}
4       do current[u] ← head [N[u]]
5  u ← head[L]
6  while u≠NIL
7       do old-height ← h[u]
8          DISCHARGE(u)
9          if h[u]>old-height
10            then move u to the front of list L
11         u ← next[u]
```

重标记与前移算法的执行如下。第 1 行把前置流和高度初始化为与一般性前置流压入算法相同的值。第 2 行对表 L 进行初始化，使其包含所有潜在的溢出顶点(以任何顺序)。第 3～4 行对每个顶点 u 的 *current* 指针进行初始化，使其指向 u 的相邻表中的第一个顶点。

如图 26-10 所示，第 6～11 行的 while 循环对表 L 进行扫描，并同时排除顶点。第 5 行使扫描从表中第一个顶点开始。每次执行循环体时，在第 8 行中的一个顶点 u 被排除。如果 u 由过程 DISCHARGE 重标记，第 10 行就把它移到表 L 的前端。在排除操作执行之前，先把顶点 u 的高度存入变量 *old-height* 中(第 7 行)，并把它与执行排除操作以后的 u 的高度进行比较(第 9 行)。第 11 行使得 while 循环的下一次迭代使用表 L 中 u 的后继顶点。如果 u 被移到表的前端，则下次迭代中使用的顶点为 u 的处于表中新的位置的后继顶点。

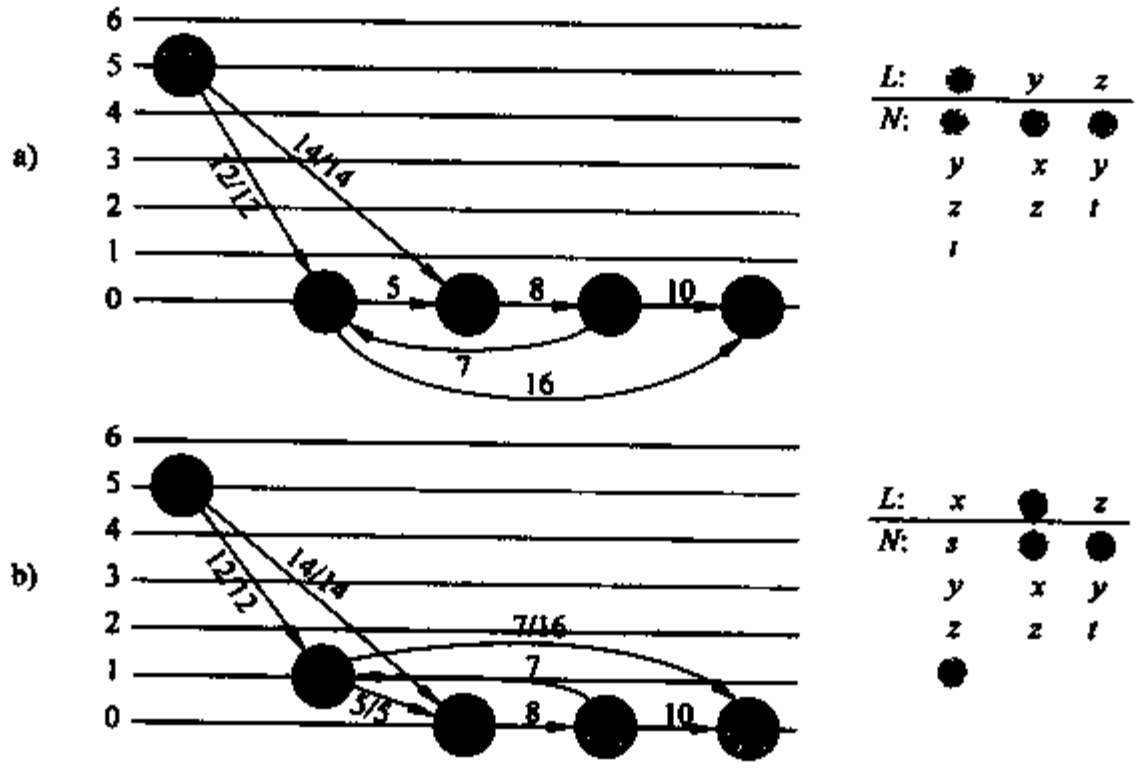


图 26-10　RELABEL-TO-FRONT 的过程。a)while 循环第一次迭代之前的流网络。初始时，26 单位流从 s 流出。右边显示了初始链表 $L=\langle x, y, z\rangle$，其中初始 $u=x$。L 中每个顶点下面保存了其邻接表，并且用阴影标出了当前的邻居。顶点 x 被排除，其高度被重标记到 1，5 个单位的流被压入 y，7 单位的流被压入汇点 t。因为 x 被重标记，需要移至 L 的头部(然而，这次并不改变 L 的结构)。b)L 中继 x 被排除的顶点为 y。图 26-9 给出了这一情形下 y 排除的每一个细节。因为 y 被重标记了，被移到了 L 的头部。c)现在 L 中 x 紧随着 y，因此它再一次被排除，将所有 5 单位的余流压入了 t。因为顶点 x 在这次排除操作中没有被重标记，因而在 L 中仍为原来位置。d)在 L 中 z 为 x 的下一顶点，它被排除。它的高度被重标记了 1，并且将所有 8 单位的余流压入 t。因为 z 是被重标记的，所以被移到了 L 的头部。e)在 L 中顶点 y 紧随着顶点 z，因而被排除。但是因为 y 没有余流，DISCHARGE 立即返回，且 y 在 L 中仍处于原位置，接着顶点 x 被排除。又因为它同样没有余流，DISCHARGE 立即返回，且 x 在 L 中仍处于原位置。RELABEL-TO-FRONT 到达链表 L 的末端并终止。此时没有溢出顶点，此前置流即为最大流

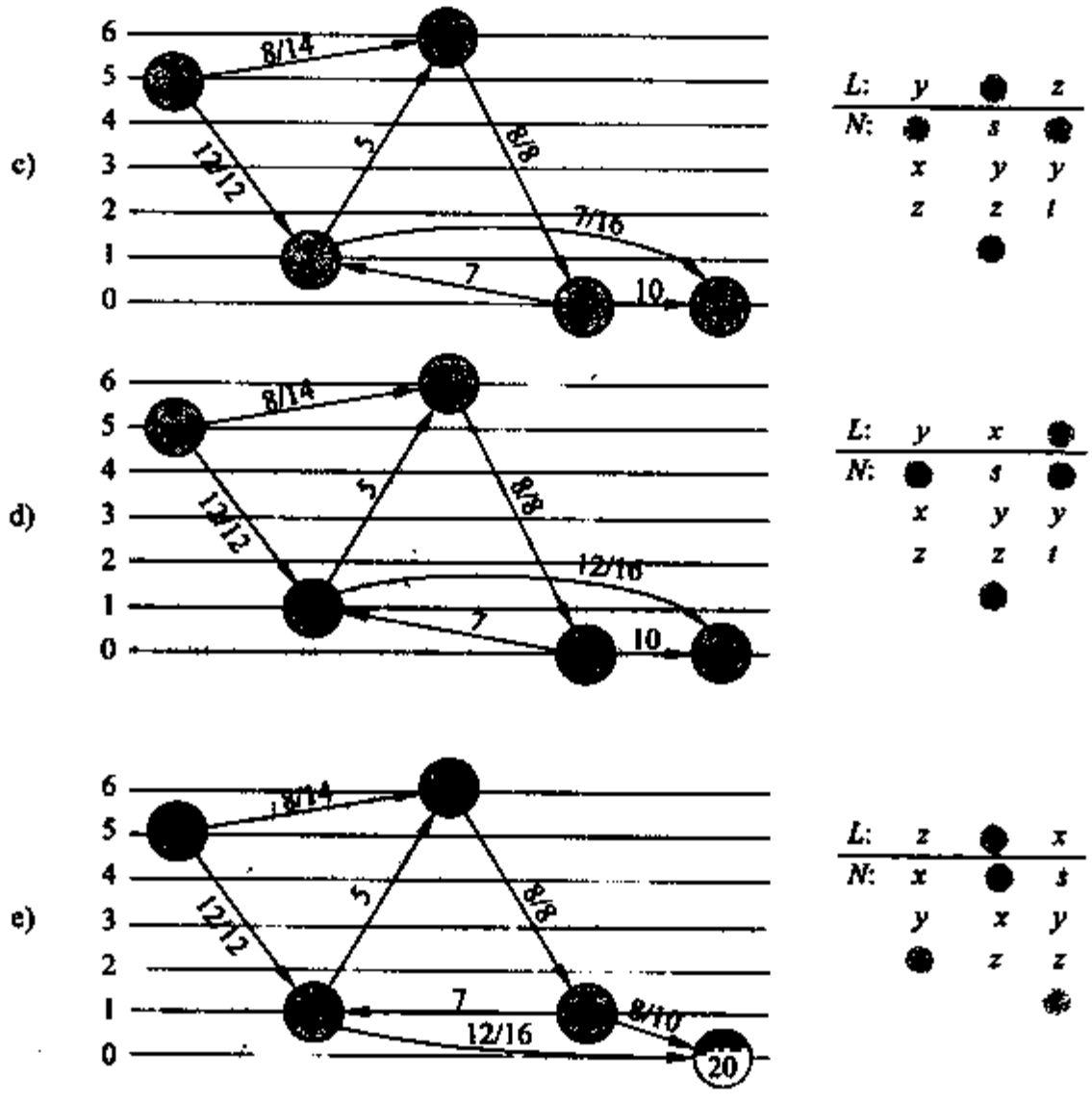


图 26-10 （续）

为了证明 RELABEL-TO-FRONT 计算出的是一个最大流，我们先证明它是一般压入与重标记算法的一种实现。首先，注意该算法仅在压入或重标记操作适用时才执行它们，因为引理 26.30 保证了 DISCHARGE 在适合的时候才会应用。还需要证明的是当 RELABEL-TO-FRONT 终止时，没有基本操作适用。正确性讨论的剩下部分将依赖于下面的循环不变式：

在 RELABEL-TO-FRONT 第 6 行的每次测试中，链表 L 存放了容许网络 $G_{f,h}=(V, E_{f,h})$ 中被拓扑排序的顶点，而且链表中在 u 之前的顶点都没有余流。

初始化：运行了 INITIALIZE-PREFLOW 后，$h[s]=|V|$ 以及对所有 $v\in V-\{s\}$，有 $h[v]=0$。因为 $|V|\geqslant 2$（V 至少包含 s 和 t），没有容许边。因而，$E_{f,h}=\varnothing$，$V-\{s, t\}$ 的任意序列都是 $G_{f,h}$ 的一个拓扑排序。

因为 u 初始化为链表 L 的头，没有顶点在其之前，所以没有在其之前的具有余流的顶点。

保持：拓扑排序是由 while 循环的每次迭代保持的，我们发现，容许网络只会因为压入和重标记操作改变。根据引理 26.28，压入操作不能将边变成容许边。因而，容许边只能由重标记操作产生。然而，顶点 u 被重标记之后，根据引理 26.29 所述，不会产生进入 u 的容许边，但可能产生离开 u 的容许边。因而，将 u 移到 L 的前端，算法保证任意离开 u 的容许边都满足拓扑排序。

要看出 L 中在 u 之前的顶点都没有余流，我们用 u' 来表示下次迭代的 u 顶点。下一次迭代中，先于 u' 的顶点包括当前的 u（根据第 11 行），以及无其他任何顶点（如果 u 被重标记）或者与先前一样的顶点（如果 u 未被重标记）。因为如果 u 被排除，那么此后它将没有余流。因此，如果 u 在排除过程中被重标记，则 u' 之前的顶点都没有余流；如果 u 在排除过程中没有被重标记，链

表中在它之前的顶点都没有获得余流，因为在排除的过程中，L 仍然保持拓扑排序(正如前面指出的，容许边仅由重标记产生，而非压入)，那么每次的压入操作仅仅将余流移到了链表中更下面的顶点(s 或 t)。先于 u' 的顶点都不会有余流。

687 ~ 689

终止：当循环终止时，u 恰好到 L 末端，因而循环不变式确保每个顶点的余流都为 0。因此，不需应用基本操作。

分析

现在来证明对任意流网络 $G=(V, E)$，过程 RELABEL-TO-FRONT 的运行时间为 $O(V^3)$。因为该算法是一般性压入与重标记方法的一种实现，所以利用推论 26.22，该推论对每个顶点执行的重标记操作次数给出了 $O(V)$ 的界，并对全部重标记操作的执行次数给出了 $O(V^2)$ 的界。此外，练习 26.4-2 给出了重标记操作所耗费的运行时间 $O(VE)$，引理 26.23 也对饱和压入操作的全部执行次数给出了 $O(VE)$ 的界。

定理 26.31　RELABEL-TO-FRONT 对任意流网络 $G=(V, E)$ 的运行时间为 $O(V^3)$。

证明：让我们考察重标记与前移算法的"阶段"，即其两次连续重标记操作之间的时间段。由于存在 $O(V^2)$ 次重标记操作，所以算法有 $O(V^2)$ 个阶段。每个阶段至多包含 $|V|$ 次对 DISCHARGE 的调用，这一点从以下陈述中可以看出。如果 DISCHARGE 没有执行重标记操作，那么 DISCHARGE 的下一次调用将沿着链表 L 继续，而 L 的长度小于 $|V|$。如果 DISCHARGE 执行了重标记操作，下一次的 DISCHARGE 调用属于不同的阶段。因为每个阶段包含至多 $|V|$ 个对 DISCHARGE 的调用，且共有 $O(V^2)$ 个阶段，所以 RELABEL-TO-FRONT 的第 8 行调用 DISCHARGE 的次数为 $O(V^3)$。因而，RELABEL-TO-FRONT 的 while 循环的工作总量至多为 $O(V^3)$(不包括 DISCHARGE 内部的执行)。

现在我们必须对算法执行中在 DISCHARGE 内部完成的工作给出一个界。在 DISCHARGE 内部，while 循环的每次迭代会执行三种操作中的一种。下面对执行每一种操作时所包含的全部工作量进行分析。

首先来看看重标记操作(第 4～5 行)。练习 26.4-2 对执行的全部 $O(V^2)$ 次重标记操作给出一个 $O(VE)$ 的时间界。

现在，我们考虑在第 8 行中对指针 $current[u]$ 进行更新的操作。每次当顶点 u 被重标记时，这一操作出现 $O(degree(u))$ 次，并且对于该顶点总共出现该操作 $O(V \cdot degree(u))$ 次。因此，对所有顶点，由握手引理(练习 B.4-1)可知，在相邻表中压入指针所完成的全部工作量为 $O(VE)$。 690

DISCHARGE 完成的第三种类型的操作是压入操作(第 7 行)。我们已经知道饱和压入操作的全部执行次数为 $O(VE)$。注意，如果执行不饱和压入操作，则由于该压入操作使余流变为 0，所以 DISCHARGE 立即返回。因此，每次对 DISCHARGE 的调用中，至多有一次不饱和压入操作。正如我们已经注意到的，DISCHARGE 被调用了 $O(V^3)$ 次，因此执行不饱和压入所需要的全部运行时间为 $O(V^3)$。

因此，RELABEL-TO-FRONT 的运行时间为 $O(V^3+VE)=O(V^3)$。 ■

练习

26.5-1　用图 26-10 所示的方法，说明 RELABEL-TO-FRONT 对图 26.1a 中的流网络执行的过程。假定 L 中的初始顶点顺序为 (v_1, v_2, v_3, v_4)，相邻表为：

$$N[v_1] = \langle s, v_2, v_3 \rangle$$
$$N[v_2] = \langle s, v_1, v_3, v_4 \rangle$$

$$N[v_3] = \langle v_1, v_2, v_4, t \rangle$$
$$N[v_4] = \langle v_2, v_3, t \rangle$$

*26.5-2　我们希望通过对溢出顶点设置一个先进先出队列的方法来实现压入与重标记算法。算法反复排除处于队头的顶点，任何在排除前不为溢出但排除后变为溢出的顶点被放在队列末尾。当队头的顶点被排除后，就把它从队列中去掉。当队列为空时，算法终止。证明可以实现这一算法，使其能在 $O(V^3)$ 的时间内计算出最大流。

26.5-3　证明：如果 RELABEL 仅通过计算 $h[u] \leftarrow h[u]+1$ 来更新 $h[u]$，一般性算法依然正确。这一变化对 RELABEL-TO-FRONT 的性能有何影响？

*26.5-4　证明：如果总是排除最高的溢出顶点，则可以使压入与重标记方法的运行时间变为 $O(V^3)$。

691　26.5-5　假定在压入与重标记算法执行的某一时刻，存在一个整数 $0 < k \leqslant |V|-1$，没有顶点满足 $h[v]=k$。证明所有 $h[v]>k$ 的顶点都在最小割的源点一边。如果这样的 k 存在，间隙启发式(gap heuristic)对 $h[v]>k$ 的 $v \in V-s$ 的每个顶点 v 进行更新，置 $h[v] \leftarrow \max(h[v], |V|+1)$。证明所得到的属性 h 为高度函数。(间隙启发式对压入与重标记算法的良好运行来说非常关键。)

思考题

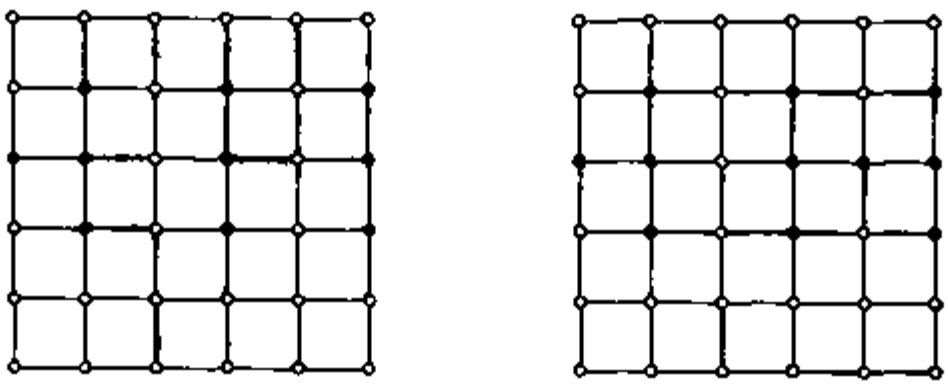
26-1　逃脱问题

一个 $n \times n$ 栅格是由 n 行和 n 列顶点组成的一个无向图，如图 26-11 所示。用 (i, j) 表示处于第 i 行第 j 列的顶点。除了边界顶点(即满足 $i=1$，$i=n$，$j=1$ 或 $j=n$ 的顶点 (i, j))，栅格中的所有其他顶点都有四个相邻顶点。

给定栅格中的 $m \leqslant n^2$ 个起始点 (x_1, y_1)，(x_2, y_2)，…，(x_m, y_m)，逃脱问题即确定从起始顶点到边界上的任何 m 个相异的顶点之间，使否存在 m 条其顶点不相交的路径。例如，图 26-11a 中的栅格就包含了一个逃脱，而图 26-11b 中的栅格中则没有逃脱。

a)考察顶点和边都具有容量的流网络，即，进入某指定顶点的正网络流受到一定的容量限制。证明：在边和顶点具有容量的网络中，确定最大流的问题可以转化为类似规模的流网络中的普通最大流问题。

b)描述一种有效的算法解决逃脱问题，并分析其运行时间。



a)　　b)

图 26-11　逃脱问题的栅格。开始点为黑色，其他栅格点为白色。a)一个包含逃脱的栅格，并用阴影显示出路径。b)不包含逃脱的栅格

26-2　最小路径覆盖

在有向图 $G=(V, E)$ 中，路径覆盖是一个其顶点不相交的路径的集合 P，满足 V 中

的每一个顶点仅包含于 P 中的一条路径中。路径可以从任意顶点开始和结束，且长度也为任意值，包括 0。G 的一个*最小路径覆盖*是指包含尽可能少的路径的路径覆盖。

a)写出一个有效算法，以找出有向无回路图 $G=(V, E)$ 的一个最小路径覆盖(提示：假设 $V=\{1, 2, \cdots, n\}$，构造图 $G'=(V', E')$，其中

$$V'=\{x_0, x_1, \cdots, x_n\} \cup \{y_0, y_1, \cdots, y_n\}$$

$$E'=\{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\}$$

然后对其运行最大流算法。)

b)所给的算法是否适用于包含回路的有向图？试作说明。

26-3　航天飞机实验

NASA 正在为航天飞机计划一系列的太空飞行，在每次飞行中必须决定进行何种商业性实验和配载何种仪器设备，Spock 教授正在研究这一问题。对每次飞行，NASA 考察一个实验集合 $E=\{E_1, E_2, \cdots, E_n\}$，实验 E_j 的商业赞助人已同意为该实验的结果向 NASA 支付 p_j 美元。实验使用的全部仪器为集合 $I=\{I_1, I_2, \cdots, I_m\}$；每个实验 E_j 所需要用到仪器为子集 $R_j \subseteq I$。运送仪器 I_k 的费用为 c_k 美元。教授的任务就是找出一个有效算法，确定在一次指定飞行中要进行哪些实验并运送哪些仪器才能使净收益最大，净收益指进行实验所获得的全部收入与运送仪器的全部费用的差额。

考察下面的网络 G。网络包含一个源点顶点 s，顶点 $I_1, I_2, \cdots, I_m$，顶点 $E_1, E_2, \cdots, E_n$ 和一个汇点顶点 t。对 $k=1, 2, \cdots, m$，存在一条容量为 c_k 的边 (s, I_k)，且对 $j=1, 2, \cdots, n$，存在一条容量为 p_j 的边 (E_j, t)。对 $k=1, 2, \cdots, m$ 和 $j=1, 2, \cdots, n$，如果 $I_k \in R_j$，则存在一条无限容量的边 (I_k, E_j)。

a)证明：对 G 的一个有限容量的割 (S, T)，如果 $E_j \in T$，则对每个 $I_k \in R_j$，有 $I_k \in T$。

b)说明如何根据 G 的最小割的容量和给定的 p_j 值来确定最大净收益。

c)给出一个有效的算法，确定应该进行哪些实验和运送哪些仪器。利用 m、n 和 $r=\sum_{j=1}^{n} |R_j|$ 来分析算法的运行时间。

692 | 693

26-4　最大流的更新

设 $G=(V, E)$ 是源点为 s、汇点为 t 并且具有整数容量的一个流网络。假定已知 G 中的一个最大流。

a)假定把一条边 $(u, v) \in E$ 的容量增加 1。试写出一个运行时间为 $O(V+E)$ 的算法以更新最大流。

b)假定把一条边 $(u, v) \in E$ 的容量减小 1。试写出一个运行时间为 $O(V+E)$ 的算法以更新最大流。

26-5　用定标法计算最大流

设 $G=(V, E)$ 是源点为 s、汇点为 t 的一个流网络。其每条边 $(u, v) \in E$ 的容量 $c(u, v)$ 为整数。设 $C=\max_{(u,v) \in E} c(u, v)$。

a)论证 G 的最小割的容量至多为 $C|E|$。

b)证明：对一给定的数 K，如果存在一条容量至少为 K 的增广路径，则可以在 $O(E)$ 的时间内找出该路径。

下面是对 FORD-FULKERSON-METHOD 修改后的算法，可以用于计算 G 的最大流。

```
MAX-FLOW-BY-SCALING(G, s, t)
1  C ← max_{(u,v)∈E} c(u, v)
```

```
2  initialize flow f to 0
3  K ← 2⌊lgC⌋
4  while K≥1
5      do while there exists an augmenting path p of capacity at least K
6             do augment flow f along p
7         K ← K/2
8  return f
```

c)论证过程 MAX-FLOW-BY-SCALING 返回一个最大流。

d)证明：每次执行第 4 行时，残留图 G_f 的最小割的容量最大为 $2K|E|$。

e)论证对每个 K 值，第 5～6 行的 while 内循环将执行 $O(E)$次。

694 f)证明：MAX-FLOW-BY-SCALING 过程的运行时间为 $O(E^2 \lg C)$。

26-6　具有负容量的最大流

假定允许一个流网络有负容量边(可以有正容量边)。在这样的网络中，可以不存在可行流。

a)考虑流网络 $G=(V, E)$中 $c(u, v)<0$ 的边(u, v)。用 u 和 v 之间的流来简单地解释负容量的含义。

设 $G=(V, E)$为一个具有负边容量的流网络，且 s 和 t 分别表示 G 的源点和汇点。构造一个一般的流网络 $G'=(V', E')$，其容量函数为 c'，源点为 s'，汇点为 t'，其中

$$V' = V \cup \{s', t'\}$$

且，

$$E' = F \cup \{(u,v):(v,u) \in E\} \cup \{(s',v):v \in V\} \cup \{(u,t'):v \in V\} \cup \{(s,t),(t,s)\}$$

按如下方式给边容量赋值。对每条边$(u, v)\in E$，设

$$c'(u,v) = c'(v,u) = (c(u,v) + c(v,u))/2$$

对每个顶点 $u\in V$，设置：

$$c'(s',u) = \max(0,(c(V,u) - c(u,V))/2)$$

且

$$c'(u,t') = \max(0,(c(u,V) - c(V,u))/2)$$

同时，置 $c'(s, t)=c'(t, s)=\infty$。

b)证明：如果 G 中存在一个可行流，则 G'中的所有容量均为非负值，而且 G'中存在一个最大流，其所有进入汇点 t'的边都饱和了。

c)证明 b)小题的逆命题。给出的证明应该是构造性的，也就是，给定 G'中的流，进
695 入 t'的边都饱和，需要说明如何得到 G 中的一个可行流。

d)给出一个能够找出 G 中最大可行流的算法。$MF(|V|, |E|)$表示 $|V|$ 个顶点、$|E|$ 条边的图上普通最大流算法的最坏运行时间。试用 MF 来分析你的算法在具有负容量边流网络上计算最大流的时间。

26-7　Hopcroft-Karp 二分图匹配算法

在本问题中，为了找到一个二分图的最大匹配，我们将描述一个由 Hopcroft 和 Karp 提出的更快速算法。此算法的运行时间为 $O(\sqrt{V}E)$。给定一个无向二分图 $G=(V, E)$，其中 $V=L\cup R$ 且所有边仅有一个端点在 L 中，设 M 为 G 的一个匹配。称简单路径 P 为 G 中关于 M 的增广路径，如果此路径从 L 中未匹配顶点出发，终止于 R 中的未匹配顶点，并且路径上的边交替地属于 M 和 $E-M$(这个增广路径的定义与流网络中的增广路径的定义

有联系，但不相同。)在本问题中，我们认为一条路径为一系列的边，而非一系列的点。关于匹配 M 的最短增广路径就是具有最小边数的增广路径。

给定两集合 A 和 B，对称差 $A \oplus B$ 定义为 $(A-B) \cup (B-A)$，也就是说只出现在两集合中的任意一个集合的那些元素。

a)证明：如果 M 是一个匹配和 P 是关于 M 的一个增广路径，那么对称差 $M \oplus P$ 是一个匹配而且 $|M \oplus P| = |M| + 1$。说明如果 P_1，P_2，…，P_k 是关于 M 的顶点不相交的增广路径，那么对称差 $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ 是一个匹配，其势为 $|M| + k$。

我们算法的一般结构如下：

```
HOPCROFT-KARP(G)
1  M ← ∅
2  repeat
3        let 𝒫 ← {P1, P2, …, Pk} be a maximum set of vertex-disjoint
               shortest augmenting paths with respect to M
4        M ← M ⊕ (P1 ∪ P2 ∪ … ∪ Pk)
5    until 𝒫 = ∅
6  return M
```

余下的问题要求分析出此算法的迭代次数(也就是 repeat 循环的迭代次数)，并且描述出第 3 行的一种实现。

696

b)给定 G 中的两个匹配 M 和 M^*，说明图 $G' = (V, M \oplus M^*)$ 中的每个顶点的度数至多为 2。证明 G' 是由一些不相交的简单路径或环组成。试说明在这样的简单路径或环上的每条边交替地属于 M 或 M^*。证明如果 $|M| \leqslant |M^*|$，那么 $M \oplus M^*$ 包含关于 M 的至少 $|M^*| - |M|$ 条顶点不相交增广路径。

设 l 为关于匹配 M 的最短增广路径的长度，并且 P_1，P_2，…，P_k 为关于 M 的长度为 l 的顶点不相交的增广路径最大集合。设 $M' = M \oplus (P_1 \cup \cdots \cup P_k)$，并假设 P 是关于 M' 的最短增广路径。

c)说明如果 P 与 P_1，P_2，…，P_k 顶点不相交，那么 P 有多于 l 条边。

d)现在假定 P 并不是与 P_1，P_2，…，P_k 点不相交。设 A 为边 $(M \oplus M') \oplus P$ 的集合。说明 $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$ 且 $|A| \geqslant (k+1)l$。证明 P 具有多于 l 条边。

e)证明如果关于 M 的最短增广路径长度为 l，那么最大匹配的大小至多为 $|M| + |V|/l$。

f)说明此算法 repeat 循环的迭代次数至多为 $2\sqrt{V}$。(提示：考虑经过 $\sqrt{V}$ 次迭代，M 能增长多少?)

g)给定一算法，能在 $O(E)$ 时间内找到关于给定匹配 M 的点不相交最短增广路径 P_1，P_2，…，P_k 的最大集合。证明 HOPCROFT-KARP 总的运行时间为 $O(\sqrt{V}E)$。

本章注记

Ahuja，Magnanti 和 Orlin[7]，Even[87]，Lawler[196]，Papadimitriou 和 Steiglitz[237]以及 Tarjan[292]都是关于网络流和相关算法非常好的参考资料。Goldberg，Tardos 和 Tarjan[119]给出了关于网络流问题算法的非常好的综述，Schrijver[267]写出了一篇网络流领域历史发展的有趣的综述。

Ford-Fulkerson 方法是由 Ford 和 Fulkerson[93]提出的，他们首次形式化地研究了网络流的

诸多问题，包括最大流以及二分图匹配的问题。早先 Ford-Fulkerson 方法的实现大都使用广度优先搜索来寻找增广路径；Edmonds 和 Karp[86]，以及 Dinic[76]分别独立地证明了这种策略是一
697 种多项式时间算法。一个类似的想法是使用"阻塞流(blocking flow)"，它是首先由 Dinic[76]提出的。Karzanov[176]第一次提出了前置流的思想。压入与重标记是由 Goldberg[117]以及 Goldberg 和 Tarjan[121]提出的。Goldberg 和 Tarjan 给出了一个使用队列来保存溢出顶点集合的 $O(V^3)$ 运行时间的算法，以及一个用动态树达到 $O(VE\lg(V^2/E+2))$ 时间的算法。许多其他的研究者也提出了压入与重标记最大流的算法。Ahuja 和 Orlin[9]以及 Ahuja，Orlin 和 Tarjan[10]给出了使用标量的算法。Cheriyan 和 Maheshwari[55]提出了从最大高度的溢出顶点上将余流压出。Cheriyan 和 Hagerup[54]提出随机置换邻接表，以及一些研究者[14，178，24]提出了基于这一思想，聪明的、非随机的一系列更快速算法。King，Rao 和 Tarjan[178]的算法是最快速的算法，其运行时间为 $O(VE\log_{E/(V/\lg V)}V)$。

到目前为止，最大流问题在渐近度上最快速的算法是由 Goldberg 和 Rao[120]提出的，其运行时间为 $O(\min(V^{2/3}, E^{1/2})E\lg(V^2/E+2)\lg C)$，其中 $C=\max_{(u,v)\in E}c(u,v)$。这样的算法不使用压入与重标记方法，而是基于求出阻塞流。所有先前的最大流算法，包括本章的算法，使用了距离标记(压入与重标记算法使用了类似的高度的标记)，长度 1 被隐式地赋给每条边。这样一个新的算法使用了不同的方式，给高容量边赋值为 0，给低容量边赋值为 1。非形式地，根据这些长度，从源点到汇点的最短路径往往具有较高的容量，也就意味着需要执行更少的迭代次数。

实际上，在最大流问题的增广路径或基于线性规划的算法中，压入与重标记算法占据了主导地位。Cherkassky 和 Goldberg[56]的一项研究突出了在实现压入与重标记算法时，两个启发式方法的重要。第一个启发式方法是周期性地在残留网络中运行广度优先搜索，以得到更准确的高度值。第二个启发式方法是间隙启发式(gap heuristic)，在练习 26.5-5 中已有描述。他们得出压入与重标记变量的最好选择是排除具有最大高度的溢出顶点。

目前最好的最大二分图匹配的算法是由 Hopcroft[152]提出的，其运行时间为 $O(\sqrt{V}E)$，在
698 思考题 26-7 中有描述。Lovász 和 Plummer[207]的书是关于匹配问题的一本非常好的参考书。

第七部分　算法研究问题选编

引　言

本篇主要是对一些算法课题进行讨论，这些课题是对本书前面一部分材料的扩展和补充。在一些章节中介绍新的算法，如组合电路或并行计算机，其他部分主要论述算法应用的特殊领域，如计算几何学和数论。最后两章对设计有效算法所受的一些限制进行探讨，并介绍克服这些限制的相应技术。

第 27 章将给出一种并行计算模型，即比较网络。粗略地说，比较网络是允许同时进行很多比较的一种算法。这一章说明如何建立比较网络，使其在 $O(\lg^2 n)$ 运行时间内对 n 个数进行排序。

第 28 章研究矩阵操作的高效算法。通过考察矩阵的一些基本性质，讨论 Strassen 算法，它可以在 $O(n^{2.81})$ 时间内将两个 $n \times n$ 矩阵相乘。然后，该章给出两种通用方法，即 LU 分解和 LUP 分解，在利用高斯消去法在 $O(n^3)$ 时间内解线性方程时要用到这两种方法。该章也说明了矩阵转置和矩阵乘法的执行能够同样地快速。在该章的最后，说明了当一组线性方程没有精确解时，如何计算最小二乘近似解。

第 29 章研究线性规划。在给定资源限制和竞争限制下，希望得到最大或最小的目标。线性规划产生于多种实践应用领域。该章讨论的是线性规划的形式化和解法。解法中介绍的是单纯形法，它是线性规划中一种最古老的解法。与本书中众多算法不同，单纯形法的最坏情况运行时间不是多项式时间，但是在实际中却被高效而广泛地应用着。

第 30 章中介绍有关多项式的操作和一种有名的信号处理技术——快速傅里叶变换(FFT)，可用于在 $O(n \lg n)$ 运行时间内，计算两个 n 次多项式的乘积。本章也讨论了 FFT 的有效实现方法，包括并行电路。

第 31 章介绍有关数论的算法。在对于数论的基本知识进行简单回顾后，该章介绍了计算最大公因数的欧几里得算法，接着给出了求解模运算线性方程组的算法，以及求解一个数的幂对另一个数的模的算法。本章还介绍了数论算法的一个重要的应用实例：RSA 公用密钥加密系统。这一密钥系统不仅可以用于信息加密，以使敌方不能读懂信息内容，而且可用于提供数字签名，本章阐述了 Miller-Rabin 随机算法素数测试，应用它可以有效地找出大的素数——这正是 RSA 系统的基本要求。本章最后论述了用于把整数分解因数的 Pollard 的“rho”启发式方法，并讨论了整数分解因数的技术现状。

第 32 章将讨论这样一个问题：在一段给定的正文字符串中，找出给定模式的字符串的全部出现位置。这一问题在文本编辑程序中经常出现。该章检验朴素方法后，首先考虑由 Rabin 和 Karp 发明的一种很精致的方法。接着，在考察了基于有限自动机的一种有效的解决方法后，论述了 Knuth-Morris-Pratt 算法，该算法对模式进行预处理，从而获得很高的效率。

计算几何学是第 33 章所讨论的课题。在讨论了计算几何学的基础性知识后，本章说明了“扫

除"(sweeping)方法如何有效地确定一个线段集合中是否包含交叉点。可找出一个点集合的凸包的两种算法——Graham 扫描算法和 Jarvis 步进算法，同样说明了扫除方法的巨大作用。本章最后阐述了一种在平面上的一组给定点中，找出最近的一对结点的有效算法。

第 34 章涉及 NP 完全问题。许多有趣的计算问题都是 NP 完全的，但目前还没有解决这一问题的多项式时间的算法。本章阐述了确定 NP 完全性的技术，介绍了几种经典 NP 完全的问题，如确定一个图中是否有汉密顿回路，确定一个布尔表达式是否是可满足的，以及确定一个给定的数的集合是否包含一个其和为给定目标值的子集。本章还证明了著名的旅行商问题是 NP 完全的。

第 35 章说明如何运用近似算法有效地找出 NP 完全问题的近似解。对一些 NP 完全的问题，
[702] 接近最优解的近似解还是比较容易获得的，但对其他一些问题，即使目前已知的最好的近似算法，其性能随着问题规模的增加而明显降低。当然，也有一些问题，如果我们增加其计算时间，就可能获得更好的近似解。在本章中通过对结点覆盖问题(不加权重的和加权重的)、旅行商问
[703] 题、集合覆盖问题以及子集求和问题的讨论，阐述了这种可能性。

第 27 章　排 序 网 络

在本书的第二部分中，我们学习了串行计算机(随机存取计算机或称 RAM)上的排序算法，这类计算机每次只能执行一个操作。本章中所讨论的排序算法基于计算的一种比较网络模型。在这种网络模型中，可以同时执行多个比较操作。

比较网络与 RAM 的区别主要在于两个方面。首先，前者只能执行比较。因此，像计数排序(见第 8.2 节)这样的算法就不能在比较网络上实现。其次，在 RAM 模型中，各操作是串行执行的，即一个操作紧接着另一个操作；在比较网络中，操作可以同时发生，或者"以并行方式"发生。如我们将会见到的那样，这一特点使得我们能够构造出一种在次线性的运行时间内对 n 个值进行排序的比较网络。

在 27.1 节的开始，我们给出比较网络和排序网络的定义。此外，还通过网络深度这一概念，对比较网络的"运行时间"给出一个自然的定义。27.2 节证明"0-1 原理"，这一原理大大减轻分析排序网络的正确性所要做的工作。

实质上，我们将要设计的有效排序网络是 2.3.1 节中合并排序算法的一个并行形式。设计过程分为三个步骤。27.3 节阐述将成为其他算法基础的"双调"排序程序的设计。27.4 节中对双调排序程序稍作修改以生成一个合并网络，它能够把两个排序序列合并为一个排序序列。最后，在 27.5 节中，我们把这些合并网络组成一个排序网络，使其能在 $O(\lg^2 n)$ 的运行时间内对 n 个值进行排序。

27.1　比较网络

排序网络(sorting network)是总能对其他输入进行排序的比较网络(comparison network)，所 704
以现在有必要先讨论比较网络及其特点。比较网络仅由线路和比较器构成。如图 27-1a 中所示，比较器是具有两个输入 x 和 y 以及两个输出 x' 和 y' 的一个装置。它执行下列函数：

$$x' = \min(x, y)$$
$$y' = \max(x, y)$$

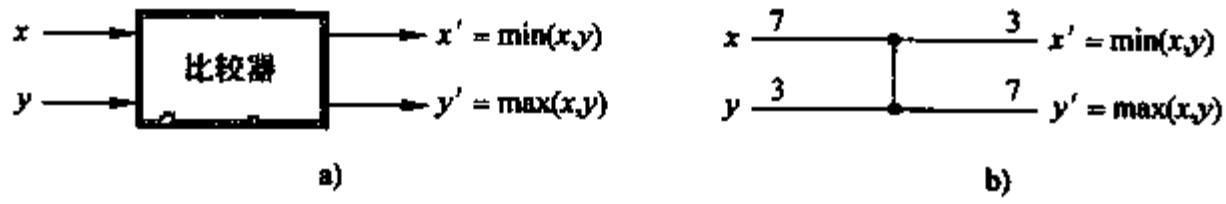


图 27-1　a)输入为 x 和 y 与输出为 x' 和 y' 的比较器。b)用一根垂直线画出的一个相同的比较器，输入为 $x=7$，$y=3$ 和输出为 $x'=3$，$y'=7$

因为图 27-1a 中给出的比较器的图形表示太大而不方便，所以我们将按常规把比较器画为一根垂直线，如图 27-1b 所示。输入在左面，输出在右面，较小的输入值在输出端的上部，较大的输入值在输出端的下部。因此，可以认为比较器对两个输入进行排序。

假设每个比较器操作占用的时间为 $O(1)$。换句话说，假定出现输入值 x 和 y 与产生输出值 x' 和 y' 之间的时间为常数。

一条线路把一个值从一处传输到另一处。可以把一个比较器的输出端与另一个比较器的输入端相连，在其他情况下，它要么是网络的输入线，要么是网络的输出线。在本章中都假定比较

网络含 n 条输入线 a_1，a_2，…，a_n，以及 n 条输出线 b_1，b_2，…，b_n。需要排序的值通过输入线进入网络，由网络计算出的结果通过输出线输出。同样，我们所说的输入序列$\langle a_1, a_2, \cdots, a_n\rangle$和输出序列$\langle b_1, b_2, \cdots, b_n\rangle$分别指输入线和输出线中的值。也就是说，我们用同一名称表示线路及其运载的值。表达的含义可根据上下文来判断。

图 27-2 展示一个比较网络，它是一个由线路互联的一组比较器的集合。我们把具有 n 个输入的比较网络画成一个由 n 条水平线和垂直伸展的比较器的组合。需要注意的是图中的一条线并不是仅代表一条线路，而是连接到各个比较器上的不同线路的一个序列。例如在图 27-2 中，顶端的一条线代表三条线路：一条是连接到比较器 A 的输入端上的输入线路 a_1；一条是连接比较
705 器 A 的顶端输出与比较器 C 的一个输入的线路；还有一条是来自比较器 C 的顶端输出的线路 b_1。每个比较器的输入端要么与网络的 n 条输入线路 a_1，a_2，…，a_n 中的一条相连接，要么与另一个比较器的输出端相连接。类似地，每个比较器的输出端要么与网络的 n 条输出线路 b_1，b_2，…，b_n 中的一条相连接，要么与另一个比较器的输入端相连接。互相连接的比较器主要应满足如下要求：互相连接的图中必须没有回路。如果我们沿图中一个给定比较器的输出端到达另一个比较器的输入端，再到输出端，再从这个输出端到达第三个比较器的输入端……如此下去，我们通过的路径必须不包含返回自身环和两次经过同一个比较器。因此，如图 27-2 所示，在画出一个比较网络时，可以把网络输入端画在左边，而把网络输出端画在右边，数据从网络左边向右边移动通过网络。

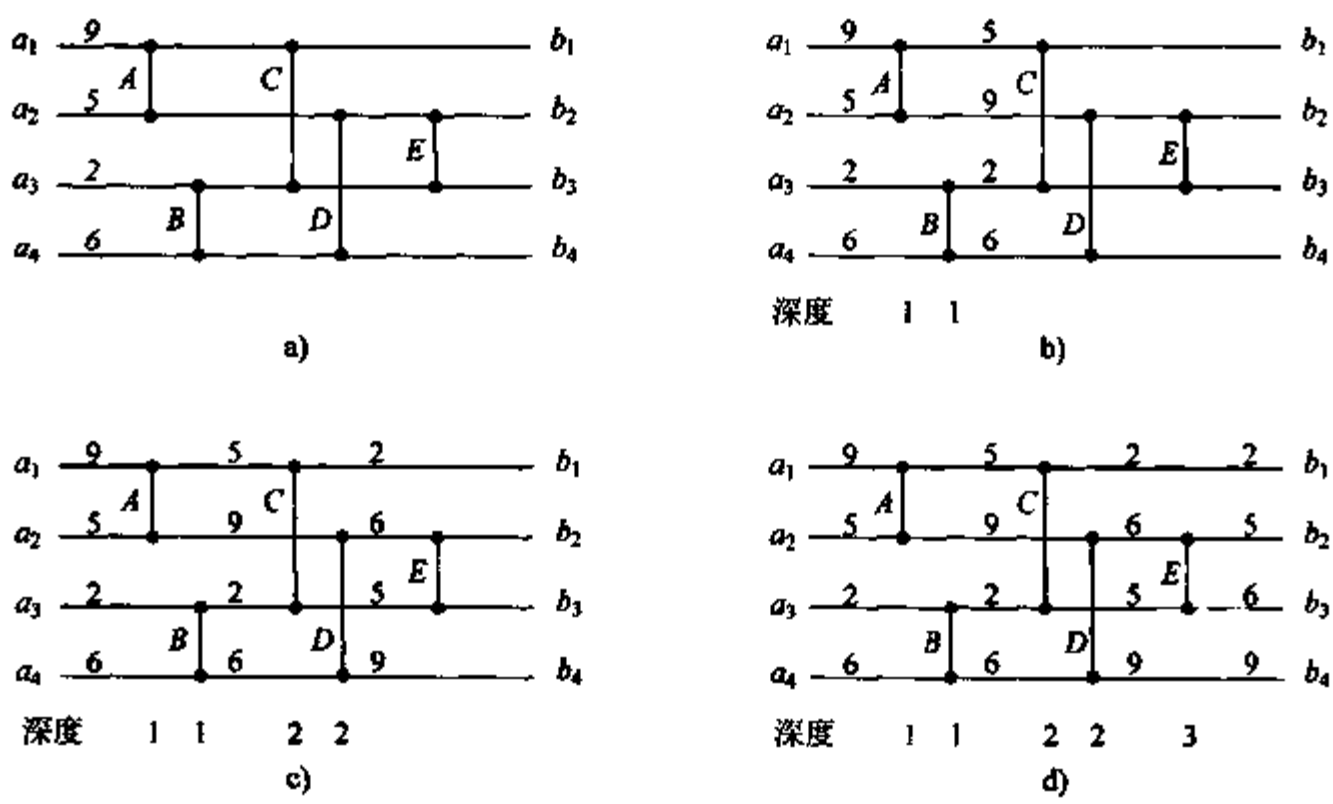


图 27-2　a)一个四输入、四输出的比较网络，事实上，它是一个排序网络。在时间 0，输入显示在四条输入线上。b)在时间 1 处，各个值显示在深度为 1 的比较器 A 和 B 的输出处。c)在时间 2 处，各个值显示在深度为 2 的比较器 C 和 D 的输出处。输出线 b_1 和 b_4 的值为最终值，但输出线 b_2 和 b_3 的值不是最终值。d)在时间 3 处，各个值显示在深度为 3 的比较器 E 的输出处。输出线 b_2 和 b_3 的值为最终的值

只有当同时有两个输入时，比较器才能产生输出值。例如，在图 27-2a 中，假设在时间 0 输入线路上出现了一个输入序列$\langle 9, 5, 2, 6\rangle$。则在时刻 0，只有比较器 A 和 B 同时存在两个输入
706 值。假定每个比较器要花 1 个单位的时间来计算出输出值，则比较器 A 和 B 在时刻 1 产生输出值，其结果如图 27-2b所示。注意，比较器 A 和 B 是同时或并行地产生其输出值的。现在，在时

刻 1，比较器 C 和 D 都有两个输入值。一个单位的时间以后，即在时刻 2，C 和 D 产生输出值，如图 27-2c 所示。比较器 C 和 D 同样也是并行操作，比较器 C 上面的输出端和比较器 D 下面的输出端分别连接着比较网络的输出线路 b_1 和 b_4，因此这些网络输出线路在时刻 2 运载着其最终值。同时，在时刻 2，比较器 E 具有有效的输入值，图 27-2d 说明了它在时刻 3 产生其输出值。这些值由网络输出线路 b_2 和 b_3 运载，这样我们就得到输出序列⟨2，5，6，9⟩。

在每个比较器均运行单位时间的假设下，我们可以对比较网络的"运行时间"作出定义，这就是从输入线路接收到其值的时刻，到所有输出线路收到其值所花费的时间。非形式地说，这一运行时间就是任何输入元素从输入线路到输出所经过的比较器数目的最大值。一条线路的深度可以更为形式化地定义如下：比较网络的输入线路深度为 0。如果一个比较器有两条深度分别为 d_x 和 d_y 的输入线路，则其输出线路的深度为 $\max(d_x+d_y)+1$。由于比较网络中没有比较器回路，所以线路的深度有明确定义，并且定义比较器的深度为其输出线路的深度。图 27-2 说明了比较器深度的概念，一个比较网络的深度是它的输出线路的最大深度，或者等价地，是其比较器的最大深度。例如，图 27-2 中的比较网络的深度为 3，这是因为比较器 E 的深度为 3。如果每个比较器产生其输出值需要 1 个单位的时间，且网络的输入出现在时刻 0，则一个深度为 d 的比较器应该在时刻 d 产生其输出值。因此网络的深度应等于网络的所有输出线均产生输出值的时刻。

排序网络是指对每个输入序列，其输出序列均为单调递增（即 $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$）的一种比较网络。当然，并非每个比较网络都是排序网络，不过图 27-2 中的网络是排序网络。为了解释这一点，请注意在时刻 1 后，四个输入值中的最小值不是产生于比较器 A 上面的一个输出端，就是产生于比较器 B 上面的一个输出端。因此，在时刻 2 后，该最小值必定处于比较器 C 上面的一个输出端。同样，可以利用对称性证明在时刻 2 后，四个输入值中的最大值必定由比较器 D 下面的一个输出端输出。因此对比较器 E 来说，剩下的工作就是保证其中间两个值处于正确的输出端，这一工作在时刻 3 完成。

比较网络与过程的相似之处在于它指定如何进行比较，其不同之处在于其实际规模决定于输入和输出的数目。因此，我们实际是在描述比较网络的"家族"。例如，本章的目标就是开发一 707
个关于有效排序网络的家族排序程序 SORTER。我们通过一个家族名和输入数目（等于输出数目）来说明一个家族中的某个指定网络。例如，在家族 SORTER 中，具有 n 个输入和 n 个输出的排序网络定义为 SORTER[n]。

练习

27.1-1　给定一输入序列⟨9，6，5，2⟩，说明图 27-2 中网络的所有线路上出现的值。

27.1-2　设 n 为 2 的幂，试说明如何构造一个具有 n 个输入、n 个输出、且深度为 $\lg n$ 的比较网络，其顶部的输出线路总是输出最小的输入值，而底部的输出线路则总是输出最大的输入值。

27.1-3　向一个排序网络中加入一个比较器后，所得的比较网络可能不再是排序网络了。说明如何对图 27-2 中所示的网络增加一个比较器，使其不能对所有输入的置换进行排序。

27.1-4　证明任何具有 n 个输入的排序网络的深度至少为 $\lg n$。

27.1-5　证明任何排序网络中的比较器的数目至少为 $\Omega(n \lg n)$。

27.1-6　考察图 27-3 所示的比较网络。证明它实际上是一个排序网络，并说明其结构和插入排序（见 2.1 节）有何联系。

27.1-7　我们可以把具有 C 个比较器和 n 个输入的比较网络表示为取值范围是从 1 到 n 的、c 对

整数组成的一张表。如果两对整数中包含同一整数，则在网络中相应的比较器的排序次序由表中整数对的次序决定。给出这种表示法，并描述一个运行时间为 $O(n+c)$ 的(串行)算法来计算比较网络的深度。

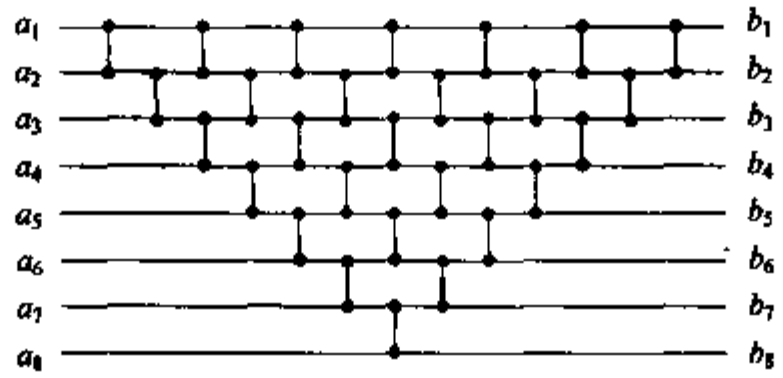


图 27-3　基于练习 27.1-6 中插入排序的排序网络

*27.1-8　假定除了上述标准类型的比较器外，我们引进一种"颠倒的"比较器，这种比较器在其底部线路中产生最大输出值。试说明如何把由 c 个标准的或颠倒的比较器组成的任意网络，转换为仅包含 c 个标准比较器的排序网络。证明所给出的转换方法是正确的。

27.2　0-1 原理

0-1 原理(0-1 principle)认为：如果对于属于集合{0，1}的每个输入值，排序网络都能正确运行，则对任意的输入值，它也能正确运行(输入值可以为整数、实数或任意线性排序的值的集合)。当我们构造排序网络和其他比较网络时，0-1 原理使得我们可以把注意力集中于对由 0 和 1 组成的输入序列进行相应操作。一旦构造好排序网络，并证明它能对所有的 0-1 序列进行排序时，就可以运用 0-1 原理，来说明它能对任意值的序列进行正确的排序。

0-1 原理的证明依赖于单调递增函数的概念(见 3.2 节)。

引理 27.1　如果比较网络把输入序列 $a=\langle a_1, a_2, \cdots, a_n\rangle$ 转化为输出序列 $b=\langle b_1, b_2, \cdots, b_n\rangle$，则对任意单调递增函数 f，该网络把输入序列 $f(a)=\langle f(a_1), f(a_2), \cdots, f(a_n)\rangle$ 转化为输出序列 $f(b)=\langle f(b_1), f(b_2), \cdots, f(b_n)\rangle$。

708
~
709

证明：我们先来证明一个断言：如果 f 是一个单调递增的函数，则输入为 $f(x)$ 和 $f(y)$ 的比较器产生的输出为 $f(\min(x, y))$ 和 $f(\max(x, y))$。然后，通过归纳来证明本引理。

为了证明上面的断言，我们来考察一个输入值为 x 和 y 的比较器。该比较器的上端输出为 $\min(x, y)$，其下端输出为 $\max(x, y)$。假定现在把 $f(x)$ 和 $f(y)$ 作为该比较器的输入，如图 27-4 所示，则该比较器的上端输出为 $\min(f(x), f(y))$，下端输出为 $\max(f(x), f(y))$。由于 f 是单调递增函数，若 $x \leqslant y$，则 $f(x) \leqslant f(y)$。因此有下列等式：

$$\min(f(x), f(y)) = f(\min(x, y))$$
$$\max(f(x), f(y)) = f(\max(x, y))$$

所以当把 $f(x)$ 和 $f(y)$ 作为比较器的输入时，所得出的值就是 $f(\min(x, y))$ 和 $f(\max(x, y))$，这样就证明了上述断言是正确的。

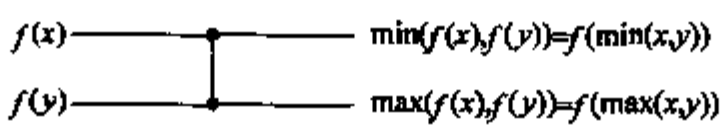
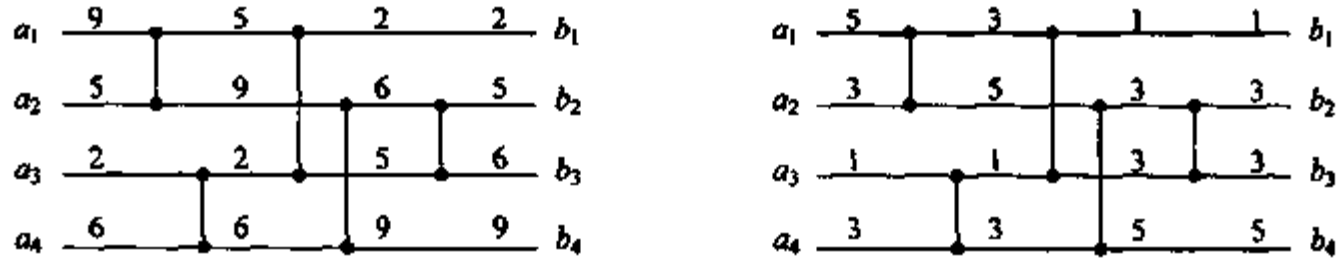


图 27-4　引理 27.1 的证明过程中比较器的操作，f 为单调递增函数

下面来对一般的比较网络中每条线路的深度进行归纳，从而证明一个比上述引理更强的结论：当把序列 a 作为网络的输入时，如果每条线路的值为 a_i，则当把序列 $f(a)$作为网络的输入时，该线路的值为 $f(a_i)$。因为输出线路包含于上述结论中，所以证明了该结论，也就证明了引理。

作为归纳的基础，考察一下深度为 0 的线路，即输入线路 a_i。此时结论显然成立：当 $f(a)$被应用于网络时，输入线路运载的值为 $f(a_i)$。下面进行归纳。考虑深度为 d 的一条线路，其中 $d \geqslant 1$。该线路是深度为 d 的比较器的输出线路，且该比较器的输入线路的深度严格小于 d。因此根据归纳假设，当序列 a 作为输入时，如果该比较器的输入线路上运载的值为 a_i 和 a_j，则当用序列 $f(a)$作为输入时，该输入线路上运载的值应为 $f(a_i)$和 $f(a_j)$。根据我们先前证明的结论，该比较器的输出线路必然运载值 $f(\min(a_i, a_j))$和 $f(\max(a_i, a_j))$。因为当输入序列为 a 时，线路运载的值为 $\min(a_i, a_j)$和$\max(a_i, a_j)$，所以定理得证。 ■

作为引理 27.1 的一个应用实例，图 27-5b 说明了使单调递增函数 $f(x)=\lceil x/2 \rceil$作用于图 27-2 中网络的输入时，所得到的排序网络。图 27-5 中每条线路上的值就是把 f 作用于图 27-2 中同一条线路上的值后所得的值。



a)　　b)

图 27-5　a)图 27-2 中输入序列为⟨9，5，2，6⟩的排序网络。b)对输入使用单调递增函数 $f(x)=f(\lceil x/2 \rceil)$的相同的排序网络。网络中的每条线路具有对应着 a)中 f 应用后得出的值

当一个比较网络是排序网络时，引理 27.1 使得我们能够证明下面的重要结论。

定理 27.2(0-1 原理)　如果一个具有 n 个输入的比较网络能够对所有可能存在的 2^n 个 0 和 1 组成的序列进行正确的排序，则对所有任意数组成的序列，该比较网络也可能对其正确排序。

证明：为了引出矛盾，假定网络能对所有 0-1 序列进行排序，但存在一个由任意数组成的序列，网络不能对该序列正确地排序。这就是说，存在一个输入序列，其中两个元素 a_i 和 a_j 满足 $a_i < a_j$，但在输出序列中 a_j 被排在 a_i 之前。我们定义一个单调递增函数 f 为：

$$f(x) = \begin{cases} 0 & \text{如果 } x \leqslant a_i \\ 1 & \text{如果 } x > a_i \end{cases}$$

因为当$\langle a_1, a_2, \cdots, a_n \rangle$作为输入序列时，其输出序列中 a_j 位于 a_i 之前，所以由引理 27.1 可知，当序列$\langle f(a_1), f(a_2), \cdots, f(a_n) \rangle$作为输入序列时，其输出序列中 $f(a_j)$必位于 $f(a_i)$之前。但由于$f(a_j)=1$，$f(a_i)=0$，这样就推出网络不能正确地对 0-1 序列$\langle f(a_1), f(a_2), \cdots, f(a_n) \rangle$进行排序，这与前面的假设相矛盾。 ■

练习

27.2-1　证明：把一个单调递增函数作用于一个已排序序列后，得到的仍然是一个排序序列。

27.2-2　证明：当且仅当能正确地对如下 $n-1$ 个 0-1 序列进行排序：$\langle 1, 0, 0, \cdots, 0, 0 \rangle$，$\langle 1, 1, 0, \cdots, 0, 0 \rangle$，$\cdots$，$\langle 1, 1, 1, \cdots, 1, 0 \rangle$，具有 n 个输入的比较网络才能够正确地对输入序列$\langle n, n-1, \cdots, 1 \rangle$进行排序。

710
~
711

27.2-3　证明：运用0-1原则，证明图27-6中所示的比较网络为一个排序网络。

27.2-4　对判定树模型(decision-tree model)阐述并证明与0-1原理类似的结论。(提示：注意要正确地处理等式。)

27.2-5　证明：对所有$i=1, 2, \cdots, n-1$，在一个具有n个输入的排序网络中，第i条线与第$i+1$条线之间必至少有一个比较器。

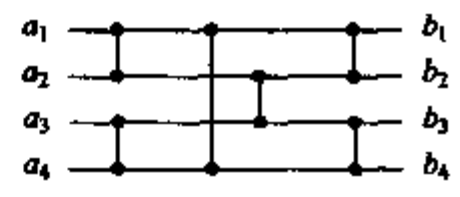


图27-6　一个对四个数进行排序的排序网络

27.3　双调排序网络

要构造有效的排序网络，第一步是构造一个能对任意双调序列(bitonic sequence)进行排序的比较网络。所谓双调序列，是指序列要么先单调递增后再单调递减，或者循环移动成为先单调递增后再单调递减。例如序列$\langle 1, 4, 6, 8, 3, 2\rangle$，$\langle 6, 9, 4, 2, 3, 5\rangle$和$\langle 9, 8, 3, 2, 4, 6\rangle$都是双调的。对于边界情况，任何1个和2个数的序列都是双调序列。双调的0-1序列的结构比较简单，其形式为$0^i1^j0^k$或$1^i0^j1^k$，其中i、j、$k\geqslant 0$。必须注意：单调递增或单调递减的序列也是单调的。

我们将要构造的双调排序程序是一个能对0和1的双调序列进行排序的比较网络。练习27.3-6将要求读者证明，双调排序程序可以对任意数组成的双调序列进行排序。

半清洁器

双调排序由一些阶段组成，其中每一个阶段称为一个半清洁器(half-cleaner)。每个半清洁器
712 是一个深度为1的比较网络，其中输入线i与输入线$i+n/2$进行比较，$i=1, 2, \cdots, n/2$(假设n为偶数)。图27-7说明了一个具有8个输入和8个输出的半清洁器HALF-CLEANER[8]。

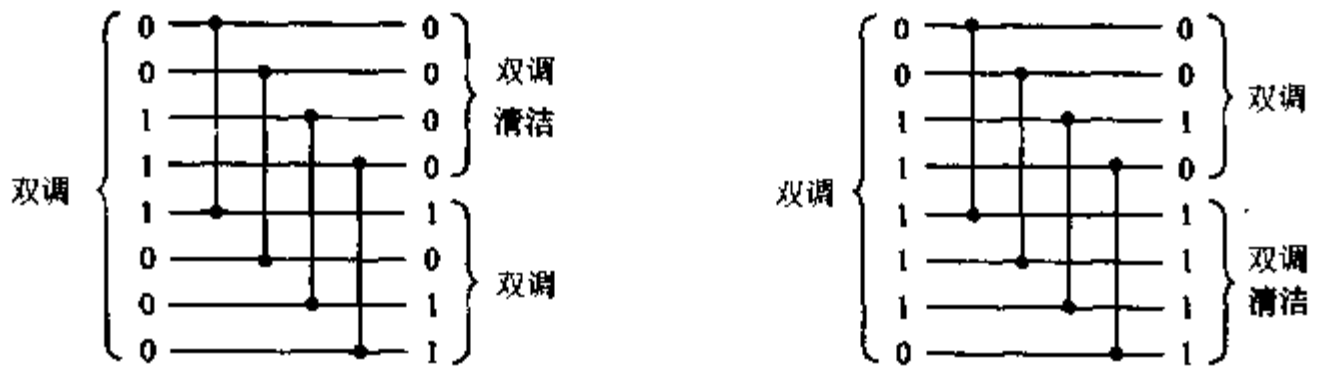


图27-7　比较网络HALF-CLEANER[8]。两种不同的0-1输入和输出。假定输入为双调的，半清洁器保证了上半部分的输出元素至少与下半部分的输出元素一样小。而且，两半部分都是双调的，且至少有一半是清洁的

当由0和1组成的双调序列作用于半清洁器输入时，半清洁器产生一个满足下列条件的输出序列：较小的值位于输出的上半部，较大的值位于输出的下半部，并且两部分序列仍然是双调的。事实上，两部分序列中至少有一个部分是清洁的——全由0或全由1组成。正是由于这一性质，我们才称其为“半清洁器”。(注意，所有的清洁序列都是双调的。)下面一个引理证明了半清洁器的这些性质。

引理27.3　如果半清洁器的输入是一个由0和1组成的双调序列，则其输出满足如下性质：输出的上半部分与下半部分都是双调的，上半部分输出的每一个元素至少与下半部分输出的每个元素一样小，并且两部分中至少有一个部分是清洁的。

证明：比较网络HALF-CLEANER[n]把第i个输入与第$i+n/2$个输入进行比较，$i=1, 2, \cdots, n/2$。不失一般性，假设输入形如：$00\cdots011\cdots100\cdots0$(输入为$11\cdots100\cdots011\cdots1$的情形与

上述情形是对称的)。根据序列的中点 $n/2$ 落在序列中连续的“0”段或连续“1”段的不同位置，可以分为三种情形，并且这些情况中有一种(中点处于连续的“1”段的情形)又可继续分为两种情况。图 27-8 说明了这四种情形。在每一种情形下，引理都成立。■ 713

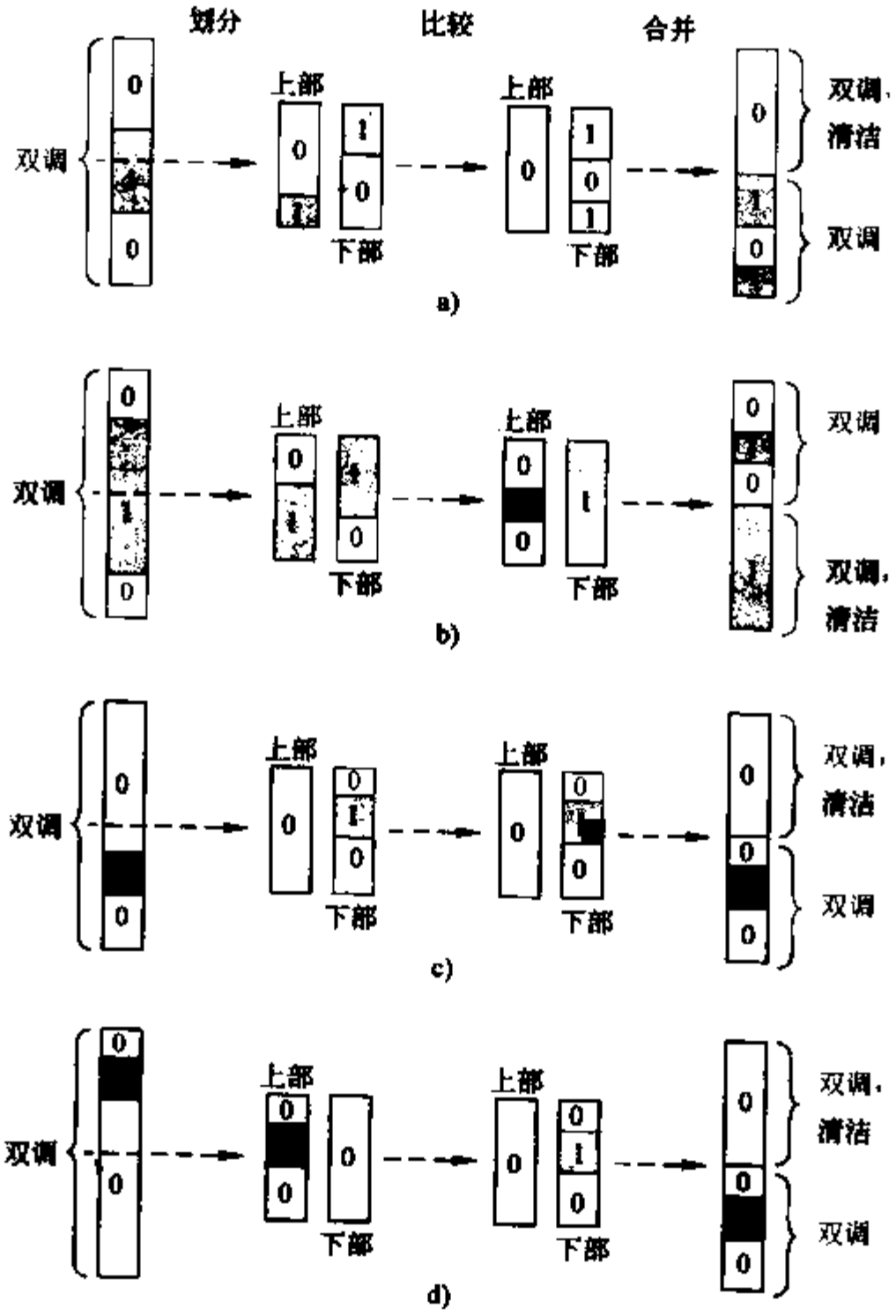


图 27-8　在 HALF-CLEANER[n]可能的比较。输入序列假定为 0 和 1 的双调序列，不失一般性，假设它形如 00…011…100…0。0 子串为白色，而 1 子串为灰色。可以认为 n 个输入被分成两半，对于 $i=1, 2, \cdots, n/2$，输入 i 与 $i+n/2$ 相比较。a)至 b)展示了划分出现在中间 1 子串的情况。c)至 d)展示了划分出现在 0 子串的情况。对于所有情况，输出的上半部分与下半部分都是双调的，上半部分输出的每一个元素至少与下半部分输出的每个元素一样小，并且两部分中至少有一个部分是清洁的

714

双调排序器

如图 27-9 所示，通过递归地连接半清洁器，就可以建立一个双调排序器。它是一个对双调序列进行排序的网络。BITONIC-SORTER[n]的第一个阶段由 HALF-CLEANER[n]组成。由引理 27.3可知，HALF-CLEANER[n]产生两个规模缩小一半的双调序列，且满足上半部分的每个元素至少与下半部分的每个元素一样小。因此，我们可以运用两个 BITONIC-SORTER[$n/2$]

分别对两部分递归地进行排序，从而完成整个排序工作。在图 27-9a 中，已经清楚地说明了递归的应用。在图 27-9b 中，对递归进行了展开以说明程序的其余部分。BITONIC-SORTER[n]的深度 $D(n)$由下列递归式给出：

$$D(n)=\begin{cases}0 & \text{如果 } n=1\\ D(n/2)+1 & \text{如果 } n=2^k \text{ 且 } k\geqslant 1\end{cases}$$

可推得其解为：$D(n)=\lg n$。

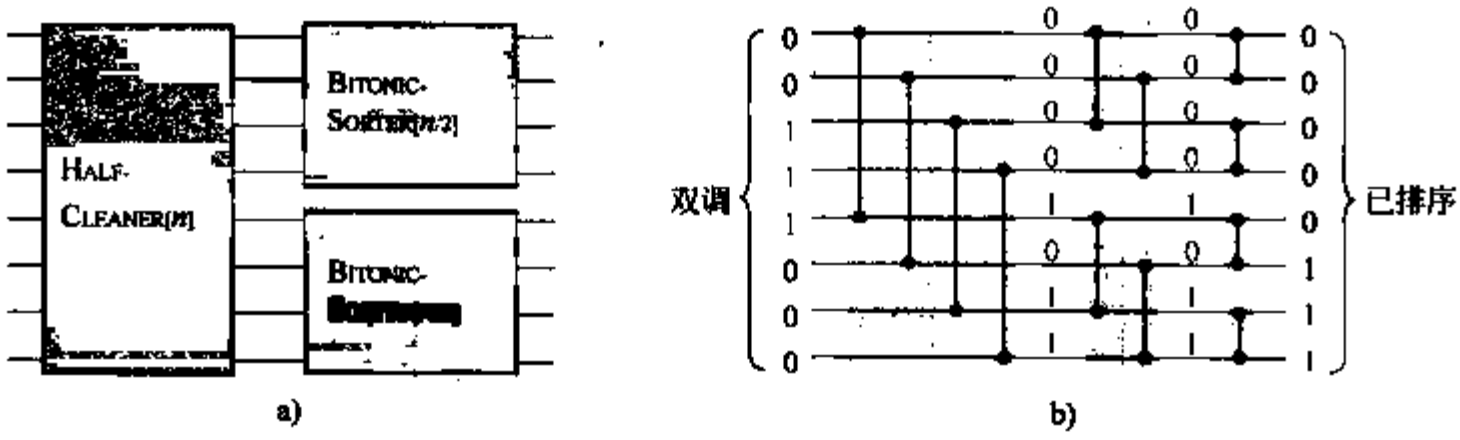


图 27-9　比较网络 BITONIC-SORTER[n]，这里 $n=8$。a)递归构造：HALF-CLEANER[n]紧跟着两个并行操作的 BITONIC-SORTER[$n/2$]。b)分开递归构造后的网络。每一个半清洁器被涂上了阴影。0、1 的样例值被标在线上

因此，可以用 BITONIC-SORTER 对深度为 $\lg n$ 的 0-1 双调序列进行排序。由类似于 0-1 原则的结论可知：该网络能对由任意数组成的双调序列进行排序。这一断言的证明留作练习(见练习 27.3-6)。

练习

27.3-1　存在多少个由 0 和 1 组成的双调序列？

715　27.3-2　证明当 n 为 2 的幂时，BITONIC-SORTER[n]包含 $\Theta(n\lg n)$个比较器。

27.3-3　说明当输入数 n 不是 2 的幂时，如何构造一个深度为 $O(\lg n)$的双调排序器。

27.3-4　如果某半清洁器的输入是一个由任意数组成的双调序列，证明输出端满足下列性质：输出的上半部分和下半部分都是双调的，上半部分中的每个元素至少与下半部分中的每个元素一样小。

27.3-5　考察两个由 0 和 1 组成的序列。证明如果其中一个序列的每个元素至少和另一个序列中每个元素一样小，则两个序列中有一个序列是清洁的。

27.3-6　证明下列与 0-1 原则类似的关于双调排序网络的结论：一个能对任何由 0 和 1 组成的双调序列进行排序的比较网络，也能够对任何由任意数字组成的双调序列进行排序。

27.4　合并网络

我们将用合并网络(merging network)来构造排序网络。所谓合并网络，就是指能把两个已排序的输入序列合并为一个有序的输出序列的网络。我们将对 BITONIC-SORTER[n]加以修改，以生成合并网络 MERGER[n]。和前一节中的双调排序类似，下面仅对输入为 0-1 序列的情况来证明合并网络的正确性。练习 27.4-1 要求读者把这一证明扩充到任意输入值的情形。

合并网络基于下列直觉思想：已知两个有序序列，如果把第二个序列的顺序颠倒，再把两个序列连接在一起，所得的序列应为双调序列。例如，已知两个有序的 0-1 序列：$X=00000111$ 和

Y=00001111，我们把 Y 的顺序颠倒，得 Y^R=11110000，再把 X 和 Y^R 相连接就得到双调序列 0000011111110000。因此要合并两个输入序列 X 和 Y，只要对 X 和 Y^R 连接成的序列执行双调排序就可以了。

716

我们通过修改 BITONIC-SORTER[n]的第一半清洁器来构造 MERGER[n]。构造过程的关键是隐含地对输入的第二个部分执行颠倒次序的操作。已知需要进行合并的两个有序序列$\langle a_1, a_2, \cdots, a_{n/2}\rangle$和$\langle a_{n/2+1}, a_{n/2+2}, \cdots, a_n\rangle$，希望的双调排序序列为$\langle a_1, a_2, \cdots, a_{n/2}, a_n, a_{n-1}, \cdots, a_{n/2+1}\rangle$。因为对 i=1，2，…，$n/2$，BITONIC-SORTER[n]的半清洁器的输入 i 与输入 $n/2+i$ 进行比较，所以构造合并网络的第一步是比较输入 i 与输入 $n-i+1$。图 27-10 说明了其对应关系。唯一的细微之处在于和通常的半清洁器的输出次序相比，在 MERGER(n)的第一步中，上端和下端输出的次序与通常的相反。由于双调序列次序颠倒后仍然是双调序列，所以合并网络第一步中的上端和下端输出满足引理 27.3 中的性质，因此，可以对上端和下端并行地进行双调排序，以产生合并网络的有序输出。

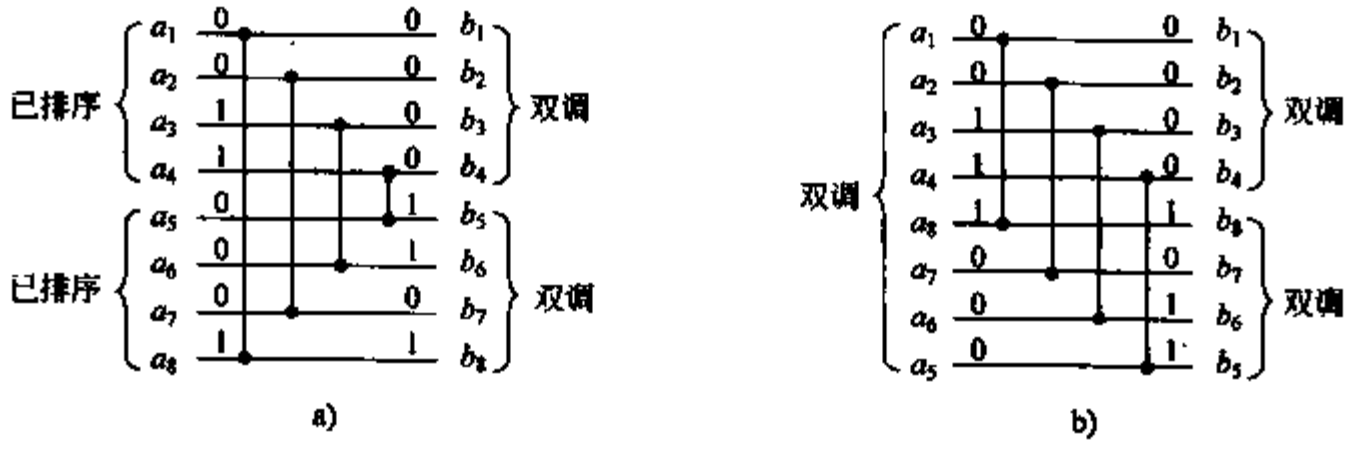


图 27-10　比较 MERGER[n]与 HALF-CLEANER[n]的第一步，其 n=8。a)MERGER[n]的第一步将两个单调序列$\langle a_1, a_2, \cdots, a_{n/2}\rangle$和$\langle a_{n/2+1}, a_{n/2+2}, \cdots, a_n\rangle$转化成两个双调序列$\langle b_1, b_2, \cdots, b_{n/2}\rangle$和$\langle b_{n/2+1}, b_{n/2+2}, \cdots, b_n\rangle$。b)对于 HALF-CLEANER[$n$]的等同操作。双调输入序列$\langle a_1, a_2, \cdots, a_{n/2-1}, a_{n/2}, a_n, a_{n-1}, \cdots, a_{n/2+2}, a_{n/2+1}\rangle$被转化为两个双调序列$\langle b_1, b_2, \cdots, b_{n/2}\rangle$和$\langle b_n, b_{n-1}, \cdots, b_{n/2+1}\rangle$

所得的合并网络如图 27-11 所示。MERGER[n]中只有第一步与 BITONIC-SORTER[n]不同，其他均相同，因此，MERGER[n]的深度与 BITONIC-SORTER[n]的深度一样，也是 $\lg n$。

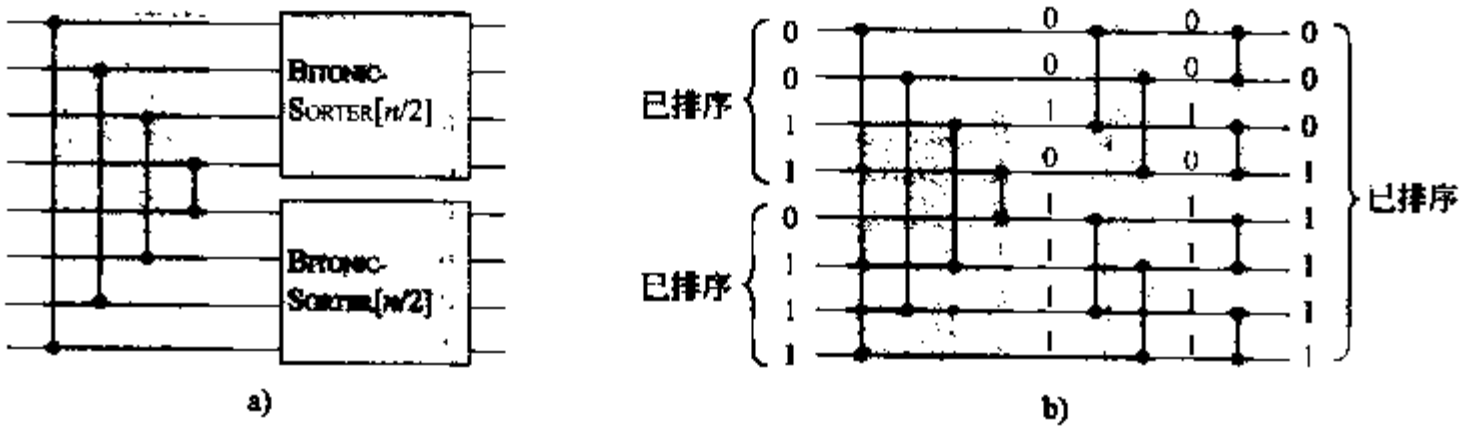


图 27-11　一个将两个已排序的输入序列合并为一个排序输出序列的网络。网络 MERGER[n]可以看作是第一个半清洁器变体将输入 i 和 $n-i+1$ 进行比较的网络，其中 i=1，2，…，$n/2$，且 n=8。a)网络被分解为第一步，其后紧跟着两个并行的 BITONIC-SORTER[$n/2$]。b)递归分解的相同的网络。0-1 样例值被标在线上，用阴影表示每一步

练习

27.4-1　对于合并网络，证明一个与0-1原则类似的结论。特别地，证明一个能对任何两个由0和1组成的单调递增序列进行合并的比较网络，也能对任何两个任意数组成的单调递增序列进行合并。

27.4-2　要把多少个不同的0-1输入序列作为一个比较网络的输入，才能验证该网络是一个合并网络？

27.4-3　证明：对任何能把1与$n-1$项合并，以产生一个长度为n的排序序列的网络，其深度至少为$\lg n$。

717
~
718

***27.4-4**　考察一个输入为$a_1, a_2, \cdots, a_n$的合并网络，n为2的幂，其中包含两个需要合并的单调序列$\langle a_1, a_3, \cdots, a_{n-1}\rangle$和$\langle a_2, a_4, \cdots, a_n\rangle$。证明：在这种合并网络中，比较器的数目为$\Omega(n \lg n)$。这为什么是一个有意义的下界呢？（提示：将比较器分解为3个集合）。

***27.4-5**　证明：不论输入次序如何，任何合并网络都需要$\Omega(n \lg n)$个比较器。

27.5　排序网络

我们现在已掌握了所有必要的工具，可以构造一个能对任意输入序列进行排序的网络了。排序网络SORTER[n]运用合并网络，实现了对第2.3.1节中的合并排序算法的并行化。图27-12说明了排序网络的构造及其操作。

图27-12a说明了SORTER[n]的递归构造。给定n个输入元素，用两个SORTER[$n/2$]，递归地对两个长度为$n/2$的子序列（并行地）进行排序，然后，再用MERGER[n]对得到的两个序列进行合并。递归的边界情况是$n=1$，此时，可以只用一条线路来对1个元素组成的序列进行排序，因为1个元素的序列已排好序了。图27-12b说明了递归展开后所得的结果，而图27-12c说明了用实际的合并网络代替图27-12b的MERGER框所得到的实际网络。

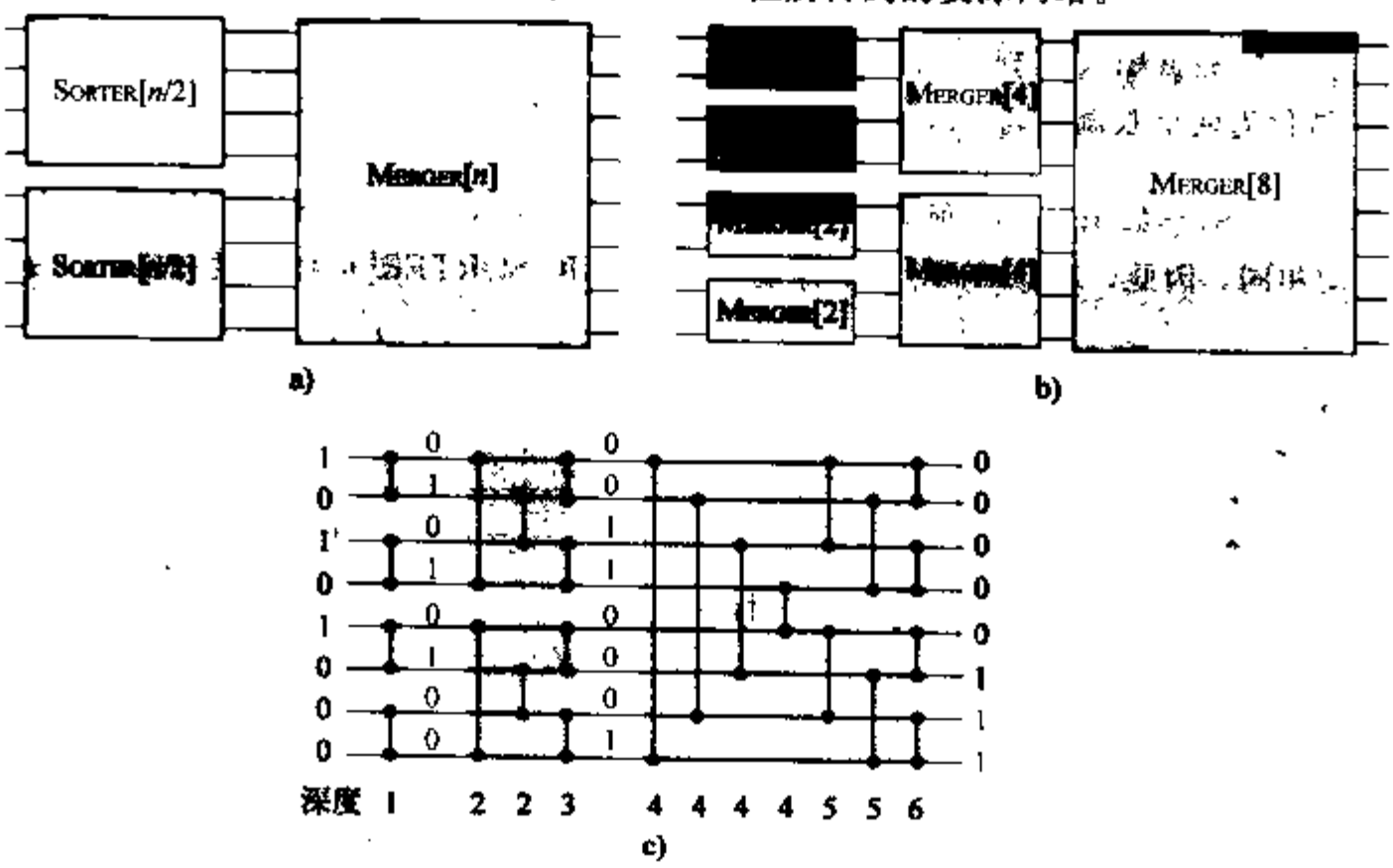


图27-12　递归地将合并网络合并而构造出的排序网络SORTER[n]。a）递归构造。b）说明了递归展开后所得的结果。c）说明了用实际的合并网络代替MERGER框所得到的实际网络。每个比较器的深度被显式地标出，且0-1值也标在线上

在网络 SORTER[n]中，数据要通过 $\lg n$ 个阶段。网络的每一个独立的输入已经是由 1 个元素组成的一个有序序列。SORTER[n]的第一个阶段包含 $n/2$ 个 MERGER[2]，它们并行地对每对由 1 个元素组成的序列进行合并，以产生长度为 2 的排序序列。第二个阶段包含 $n/4$ 个 MERGER[4]，它们把每对由 2 个元素组成的排序序列进行合并，以产生长度为 4 的排序序列。一般来说，对于 $k=1, 2, \cdots, \lg n$，第 k 个阶段包含 $n/2^k$ 个 MERGER[2^k]，它们把每对由 2^{k-1} 个元素组成的排序序列进行合并，结果是长度为 2^k 的排序序列。在最后一个阶段，只产生由全部输入值组成的一个排序序列。我们可以用归纳法来证明这一排序网络能对 0-1 序列进行排序，因此由 0-1 原则(定理 27.2)可知，它也同样能对任意输入值进行排序。

我们可以递归地分析排序网络的深度。SORTER[n]的深度 $D(n)$ 就是 SORTER[$n/2$]的深度 $D(n/2)$(存在两个相同的 SORTER[$n/2$]，它们并行地操作)加上 MERGER[n]的深度 $\lg n$。因此，SORTER[n]的深度可由下列递归式定义：

$$D(n) = \begin{cases} 0 & \text{如果 } n = 1 \\ D(n/2) + \lg n & \text{如果 } n = 2^k \text{ 且 } k \geqslant 1 \end{cases}$$

可以推出其解为 $D(n) = \Theta(\lg^2 n)$(要用到练习 4.4-2 中给出的主定理版本)。因此，可以在 $O(\lg^2 n)$的时间内并行地对 n 个数进行排序。

练习

27.5-1　SORTER[n]中有多少个比较器？

27.5-2　证明 SORTER[n]的深度恰好为$(\lg n)(\lg n+1)/2$。

27.5-3　假定有 2^n 个元素$(a_1, a_2, \cdots, a_{2^n})$，我们希望把该序列划分为两个序列，其中一个包含 n 个最小值，另一个包含 n 个最大值。证明：分别对序列$\langle a_1, a_2, \cdots, a_n\rangle$和$\langle a_{n+1}, a_{n+2}, \cdots, a_{2n}\rangle$进行排序后再在一定的深度内就可达到上述要求。

***27.5-4**　设 $S(k)$为具有 k 个输入的排序网络的深度，$M(k)$为具有 $2k$ 个输入的合并网络的深度。假定我们对一个由 n 个数组成的序列进行排序，并且已知每个数与其在结果序列中的正确位置相差不超过 k 个数的位置。证明：能够在深度 $S(k)+2M(k)$内对这 n 个数进行排序。

719 ~ 720

***27.5-5**　可以通过反复执行下列过程 k 次，来对一个 $m \times m$ 矩阵中的元素进行排序：

1. 把每个奇数行的元素排列成单调递增序列。
2. 把每个偶数行的元素排列成单调递减序列。
3. 把每列元素排列成单调序列。

对上述排序过程需要进行多少次迭代？排序后输出的结果序列是什么样的模式？

思考题

27-1　排序网络的转置

在一个比较网络中，如果每一个比较器仅连接相邻的两根线，如图 27-3 所示，则称这种网络为*转置网络*(transposition network)。

a)证明：任何具有 n 个输入的转置网络都包括 $\Omega(n^2)$个比较器。

b)证明：当且仅当能对序列$\langle n, n-1, \cdots, 1\rangle$进行排序时，具有 n 个输入的转置网络为排序网络。(提示：运用与引理 27.1 的证明相类似的归纳法来证明。)

一个具有 n 个输入$(a_1, a_2, \cdots, a_n)$的*奇偶排序网络*(odd-even sorting network)具有 n 级比较器。图 27-13 说明了一个 8 输入的奇偶转置网络。从图中可以看出，对 $i=2, 3, \cdots, n$

和$d=1, 2, \cdots, n$,线 i 通过深度为 d 的比较器与线 $j=i+(-1)^{i+d}$相连接，其中 $1\leqslant j\leqslant n$。

c)证明：奇偶排序网络的确能进行排序。

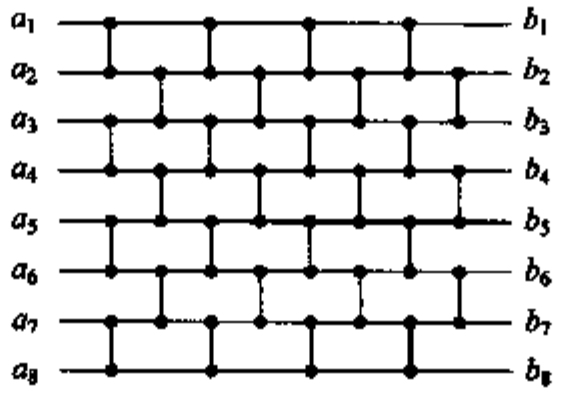


图 27-13　一个 8 输入的奇偶排序网络

27-2　Batcher 奇偶合并网络

27.4 节中介绍了如何基于双调排序来构造合并网络。在本题中，我们将构造一个奇偶合并网络。假设 n 为 2 的幂，我们希望对线路上的排序序列$\langle a_1, a_2, \cdots, a_n\rangle$和序列$\langle a_{n+1}, a_{n+2}, \cdots, a_{2n}\rangle$进行合并。如果 $n=1$，在线 a_1 和 a_2 之间放置一个比较器。否则，就递归地构造两个并行操作的奇偶合并网络。第一个合并对线路上的序列$\langle a_1, a_3, \cdots, a_{n-1}\rangle$和序列$\langle a_{n+1}, a_{n+3}, \cdots, a_{2n-1}\rangle$(序号为奇数的元素)进行合并，第二个合并网络把序列$\langle a_2, a_4, \cdots, a_n\rangle$与序列$\langle a_{n+2}, a_{n+4}, \cdots, a_{2n}\rangle$(序号为偶数的元素)进行合并，为使两个排序序列相连接，我们把一个比较器放在 a_{2i}和 a_{2i+1}之间，$i=1, 2, \cdots, n-1$。

a)对 $n=4$，画出一个 $2n$ 个输入的合并网络。

b)Corrigan 教授提出，可以通过合并两个用递归合并排序的子串的方式来进行排序。在 a_{2i-1}和 a_{2i}之间放置一个比较器($i=1, 2, \cdots, n$)，而不是在 a_{2i}和 a_{2i+1}之间放置比较器($i=1, 2, \cdots, n-1$)。对于 $n=4$，画出 $2n$ 个输入的网络，并给出一个反例以说明教授的产生合并网络的思想有错误。说明你的例子在 a 部分所画的 $2n$ 个输入的合并网络上运行得很好。

c)运用 0-1 原则证明：任意的 $2n$ 个输入的奇偶合并网络是合并网络。

d)$2n$ 个输入的奇偶合并网络的深度是多少？规模有多大？

27-3　排列网络

具有 n 个输入和 n 个输出的排列网络(permutation network)中存在着一些开关，用来根据 $n!$ 种可能的排列，把网络的输入和输出进行各种可能的连接。图 27-14a 说明了一个两输入、两输出的排列网络 P_2，该网络只包含一个开关，对输入输出存在两种连接方式：直接相连或交叉相连。

721
|
722

a) 证明：如果把排序网络中的每一个比较器换成图 27-14a 所示的开关，所得的网络就是一个排列网络。也就是说，对任意排列 π，网络中存在一种置开关的方式可以使输入 i 与输出$\pi(i)$相连。

图 27-14b 说明了一个 8 输入、8 输出排列网络 P_8 的递归构造过程，其中使用了两个相同的 P 和 8 个开关。开关的状态可以实现排列 $\pi=\langle\pi(1), \pi(2), \cdots, \pi(8)\rangle=\langle 4, 7, 3, 5, 1, 6, 8, 2\rangle$，而它又递归地要求上面的 P_4 实现排列$\langle 4, 2, 3, 1\rangle$，下面的 P_4 实现排列$\langle 2, 3, 1, 4\rangle$。

b)说明如何实现排列网络 P_8 上的排列$\langle 5, 3, 4, 6, 1, 8, 2, 7\rangle$，画出开关的位置

状态和两个排列网络 P_4 所实现的排列。

设 n 为 2 的幂，用与定义 P_8 类似的方式，用两个 $P_{n/2}$ 来递归地对 P_n 进行定义。

c)描述一个运行时间(普通 RAM)为 $O(n)$ 的算法，该算法可以对 P_n 中连接输入和输出的 n 个开关进行置位，并能说明每个 $P_{n/2}$ 所必须实现的排列，以此来完成任意给定的 n 元素的排列。证明所给出的算法是正确的。

d)P_n 的深度是多少？规模有多大？利用普通的随机存取计算机(RAM)，要花多长时间才能够计算出所有的开关置位？(包括 $P_{n/2}$ 中的开关置位)。

e)证明：对 $n>2$，任何排列网络(不只是 P_n)必须通过开关置位的两种连接实现某一排列。

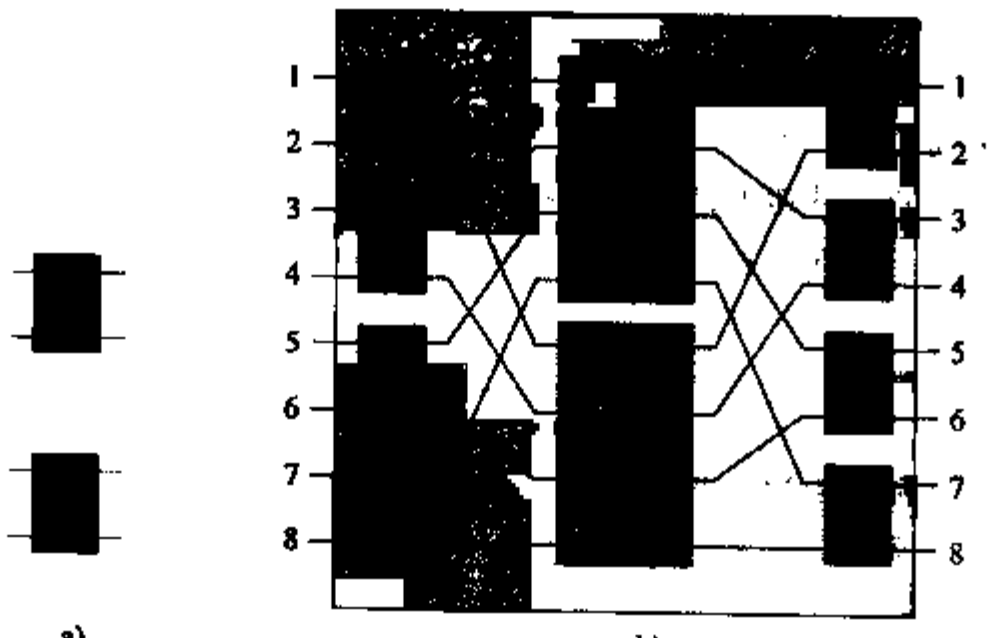


图 27-14　排列网络。a)排列网络 P_2，一个开关可以两种方式连接。b)由 8 个开关和两个 P_4 递归构造 P_8。开关和 P_4 的连接考虑到了排列 $\pi=\langle 4, 7, 3, 5, 1, 6, 8, 2\rangle$

本章注记

Knuth[185]包含了关于排序网络的讨论及其历史的话题。它们第一次是由 P. N. Armstrong、R. J. Nelson 和 D. J. O'Connor 于 1954 年提出的。在 1960 年早期，K. E. Batcher 发现了第一个能够在 $O(\lg n)$ 时间内将两个序列合并的网络。他是用了奇偶合并(见思考题 27-2)，并且说明了如何使用这一技术在 $O(\lg^2 n)$ 的时间内将 n 个数排序。过后不久，他又提出了与 27.3 节中类似的 $O(\lg n)$ 深度的双调排序器。Knuth 将 0-1 原则归功于 W. G. Bouricius(1954 年)，后者证明了其在决策树中的正确性。

有相当一段时间，关于是否存在深度为 $O(\lg n)$ 的排序网络的问题一直没有解决。在 1983 年，答案被找到了、那就是"是的"，但该答案不太令人满意。AKS 排序网络(根据它的发明者，Ajtai、Komlós 和 Szemerédi[11]命名的)可以使用 $O(n\lg n)$ 个比较器，在 $O(\lg n)$ 深度上将 n 个数排序。遗憾的是，隐藏在 O 记号下的常数是非常大的(数千)，因而被认为是不实用的。

723
~
724

第28章 矩阵运算

矩阵运算在科学计算中非常重要。因此，在实际应用中，关于矩阵的有效算法非常值得注意。本章简要介绍矩阵论和矩阵运算，着重介绍矩阵乘法问题与求解联立线性方程组问题。

在28.1节介绍矩阵的基本概念与记号后，28.2节给出矩阵相乘的Strassen算法，该算法能令人惊奇地在$\Theta(n^{\lg 7})=\Theta(n^{2.81})$时间内，计算出两个$n\times n$矩阵的乘积。28.3节说明如何使用LUP分解求解线性方程组。28.4节讨论矩阵乘法问题与求逆矩阵问题之间的密切联系。最后，28.5节讨论一类重要的矩阵，即对称正定矩阵，并说明如何用它们求超定线性方程组的最小二乘解。

在实践中产生的一个重要问题是数值的稳定性(numerical stability)。由于在实际的计算机中，浮点数表示的精度有限，因此，在数值计算的过程中，舍入误差可能会被放大，从而导致不正确的结果；这样的计算是数值不稳定的。在本章中，除了会偶尔简要地提到数值稳定性外，不会着重来讨论这个问题。建议读者参考Golub和Van Loan[125]所著的优秀教材来全面了解有关数值稳定性方面的知识。

28.1 矩阵的性质

本节，首先复习矩阵论中的一些基本概念和矩阵的一些基本性质，重点是复习后面几节需
725 要用到的那些概念和性质。

矩阵和向量

矩阵(matrix)是数字的一个矩形阵列。例如

$$A=\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}=\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \tag{28.1}$$

是一个2×3的矩阵$A=(a_{ij})$，其中$i=1,2$，$j=1,2,3$，矩阵中处于第i行、第j列的元素为a_{ij}。用大写字母表示矩阵，用相应的带下标的小写字母表示矩阵的元素。元素为实数的所有$m\times n$矩阵组成的集合用$\mathbf{R}^{m\times n}$表示。一般来说，元素属于集合S的所有$m\times n$矩阵的集合用$S^{m\times n}$表示。

矩阵A的转置(transpose)A^{T}是矩阵A的行和列互相交换而产生的矩阵。对于式(28.1)中的矩阵A，

$$A^{\mathrm{T}}=\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

向量(vector)是数字的一维阵列。例如，

$$x=\begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \tag{28.2}$$

是一个包含3个数的向量。我们用小写字母表示向量，当$i=1,2,\cdots,n$时，n维向量x中的第i个元素表示为x_i。可以把向量的标准形式列向量看成是一个$n\times 1$矩阵。其对应的行向量可通过转置获得：

$$x^{\mathrm{T}}=(2 \quad 3 \quad 5)$$

单位向量 e_i 是第 i 个元素为 1 而所有其他元素均为 0 的向量。通常，从上下文可以清楚地看出单位向量的大小。

零矩阵是指所有元素都是 0 的矩阵。这样的矩阵通常用 0 来表示，因为从上下文就可以把它与数字 0 区分开来。如果是一个元素为 0 的矩阵，则该矩阵的大小也需要从上下文得出。

我们经常会遇到 $n\times n$ 方阵。方阵的几种特殊情形非常值得我们重视。

1)对角矩阵：当 $i\neq j$ 时，有 $a_{ij}=0$。因为所有的非对角线上的元素均为 0，所以可以通过列出对角线上的元素表示对角矩阵：

$$\operatorname{diag}(a_{11},a_{22},\cdots,a_{nn})=\begin{pmatrix}a_{11}&0&\cdots&0\\0&a_{22}&\cdots&0\\\vdots&\vdots&\ddots&\vdots\\0&0&\cdots&a_{nn}\end{pmatrix}$$

2)$n\times n$ 单位矩阵(identity matrix)I_n 是对角线上的元素都是 1 的对角矩阵：

$$I_n=\operatorname{diag}(1,1,\cdots,1)=\begin{pmatrix}1&0&\cdots&0\\0&1&\cdots&0\\\vdots&\vdots&\ddots&\vdots\\0&0&\cdots&1\end{pmatrix}$$

当 I 不带下标时，它的规模可以从上下文推出。单位矩阵的第 i 列就是单位向量 e_i。

3)三对角矩阵(tridiagonal matrix)T 是满足若 $|i-j|>1$，则元素 $t_{ij}=0$ 的矩阵。非零元素仅出现在主对角线上，仅靠主对角线上面($t_{i,i+1}$，$i=1,2,\cdots,n-1$)和仅靠对角线下面($t_{i+1,i}$，$i=1,2,\cdots,n-1$)：

$$T=\begin{pmatrix}t_{11}&t_{12}&0&0&\cdots&0&0&0\\t_{21}&t_{22}&t_{23}&0&\cdots&0&0&0\\0&t_{32}&t_{33}&t_{34}&\cdots&0&0&0\\\vdots&\vdots&\vdots&\vdots&\ddots&\vdots&\vdots&\vdots\\0&0&0&0&\cdots&t_{n-2,n-2}&t_{n-2,n-1}&0\\0&0&0&0&\cdots&t_{n-1,n-2}&t_{n-1,n-1}&t_{n-1,n}\\0&0&0&0&\cdots&0&t_{n,n-1}&t_{nn}\end{pmatrix}$$

4)上三角矩阵(upper-triangular matrix)U 是满足若 $i>j$，则 $u_{ij}=0$ 的矩阵。对角线下面的所有元素均为 0：

$$U=\begin{pmatrix}u_{11}&u_{12}&\cdots&u_{1n}\\0&u_{22}&\cdots&u_{2n}\\\vdots&\vdots&\ddots&\vdots\\0&0&\cdots&u_{nn}\end{pmatrix}$$

如果其对角线元素都为 1，则这样的上三角矩阵称为单位上三角矩阵(unit upper-triangular)。

5)下三角矩阵(lower-triangular matrix)L 是满足若 $i<j$，则 $l_{ij}=0$ 的矩阵。对角线上面的所有元素均为 0：

$$L=\begin{pmatrix}l_{11}&0&\cdots&0\\l_{21}&l_{22}&\cdots&0\\\vdots&\vdots&\ddots&\vdots\\l_{n1}&l_{n2}&\cdots&l_{nn}\end{pmatrix}$$

如果其对角线元素都为1，则这样的下三角矩阵称为单位下三角矩阵。

6）置换矩阵（permutation matrix）P 的每一行或列中都仅含一个1，其他元素都为0。下列矩阵是置换矩阵的一个例子：

$$P=\begin{pmatrix}0&1&0&0&0\\0&0&0&1&0\\1&0&0&0&0\\0&0&0&0&1\\0&0&1&0&0\end{pmatrix}$$

这个矩阵之所以叫置换矩阵，是因为把一个向量 x 和一个置换矩阵相乘所得的结果就是向量 x 中的元素的一种置换。

7）对称矩阵（symmetric matrix）A 满足条件 $A=A^{\mathrm{T}}$。例如，

$$\begin{pmatrix}1&2&3\\2&6&4\\3&4&5\end{pmatrix}$$

就是一个对称矩阵。

关于矩阵的运算

矩阵或向量的元素来源于一个数字系统，例如实数、复数或整数对某质数所取的模。数字系统定义了数的加法和乘法规则。我们可以把这些定义推广到矩阵的加法和乘法运算中来。

定义矩阵的加法如下。如果 $A=(a_{ij})$，$B=(b_{ij})$ 是 $m\times n$ 矩阵，则它们的矩阵和 $C=(c_{ij})=A+B$ 是 $m\times n$ 矩阵，定义为

728

$$c_{ij}=a_{ij}+b_{ij}$$

$i=1,2,\cdots,m$；$j=1,2,\cdots n$。亦即，矩阵加法是通过对各矩阵的对应元素分别相加所获得的。零矩阵是矩阵加法运算的单位元：

$$A+0=A=0+A$$

如果 λ 是一个常数，$A=(a_{ij})$ 是一个矩阵，则矩阵 A 的标量乘积 $\lambda A=(\lambda a_{ij})$ 是用 λ 乘以 A 中各元素形成的矩阵。作为一个特例，定义矩阵 $A=(a_{ij})$ 的负矩阵为 -1。$A=-A$，因此 $-A$ 的第 ij 个元素为 $-a_{ij}$。所以，

$$A+(-A)=0=(\quad A)+A$$

有了上述定义后，就可以把两个矩阵的差定义为一个矩阵与另一个矩阵的负矩阵的和：$A-B=A+(-B)$。

定义矩阵的乘积如下。首先，两个可以相乘的矩阵 A 和 B 必须是相容的，即 A 的列数必须等于 B 的行数（一般来说，在一个包含矩阵乘积 AB 的表达式中，通常已隐含地假设矩阵 A 和矩阵 B 是相容的）。如果 $A=(a_{ij})$ 是一个 $m\times n$ 矩阵，$B=(b_{jk})$ 是一个 $n\times p$ 矩阵，则它们的乘积 $C=AB$ 是 $m\times p$ 矩阵 $C=(c_{ik})$，其中

$$c_{ik}=\sum_{j=1}^{n}a_{ij}b_{jk} \tag{28.3}$$

$i=1,2,\cdots,m$；$k=1,2,\cdots p$。25.1节中的过程 MATRIX-MULTIPLY 就是用基于式(28.3)的简单方法来实现矩阵乘法的。它假定所有矩阵都是方阵，即 $m=n=p$。为了求出两个 $n\times n$ 方阵的乘积，过程 MATRIX-MULTIPLY 执行 n^3 次乘法和 $n^2(n-1)$ 次加法，因此它的运行时间为 $\Theta(n^3)$。

矩阵的许多（但不是全部）代数性质与数字的相同。单位矩阵是矩阵乘法的单位元：

$$I_m A = A I_n = A$$

其中 A 为任意 $m \times n$ 矩阵。任何一个矩阵与零矩阵的积就是零矩阵：

$$A0 = 0$$

矩阵乘法满足结合律，即对相容的矩阵 A，B，C，有

$$A(BC) = (AB)C \tag{28.4}$$

矩阵乘法对加法满足分配律：

$$A(B+C) = AB + AC$$

$$(B+C)D = BD + CD \tag{28.5}$$

729

但是，对于 $n > 1$，$n \times n$ 矩阵乘法不满足交换律。

例如，若 $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ 和 $B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$，则 $AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$，而 $BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$。

定义矩阵与向量的乘积或向量与向量的乘积如下：可以把向量看作 $n \times 1$ 矩阵(或把行向量看作是 $1 \times n$ 矩阵)。因此，如果 A 是一个 $m \times n$ 矩阵，x 是一个 n 维向量，则 Ax 是一个 m 维向量。如果 x 和 y 都是 n 维向量，则

$$x^{\mathrm{T}} y = \sum_{i=1}^{n} x_i y_i$$

是一个称为 x 和 y 的内积的数(实际上是一个 1×1 矩阵)。xy^{T} 是一个 $n \times n$ 的矩阵 Z，称为 x 和 y 的外积，其中 $z_{ij} = x_i y_j$。n 维向量 x 的(欧氏)范数(euclidean norm)由下式定义：

$$\|x\| = (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2} = (x^{\mathrm{T}} x)^{1/2}$$

因此，x 的范数是其在 n 维欧氏空间中的长度。

矩阵的逆、秩和行列式

定义 $n \times n$ 矩阵 A 的逆矩阵(inverse)为一个 $n \times n$ 矩阵，用 A^{-1} 来表示(如果存在)，它满足 $AA^{-1} = I_n = A^{-1}A$。例如：

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$$

许多非零 $n \times n$ 矩阵并没有逆矩阵。没有逆矩阵的矩阵称为不可逆矩阵(noninvertible)或奇异矩阵(singular)。下面的矩阵就是一个奇异矩阵：

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$$

如果一个矩阵有逆矩阵，则称其可逆，或非奇异矩阵。如果矩阵的逆矩阵存在，则必定是唯一的(见练习 28.1-3)。如果 A 和 B 是非奇异的 $n \times n$ 矩阵，则 730

$$(BA)^{-1} = A^{-1} B^{-1} \tag{28.6}$$

求逆运算与转置运算交换：

$$(A^{-1})^{\mathrm{T}} = (A^{\mathrm{T}})^{-1}$$

设 x_1，x_2，…，x_n 是 n 个向量，若存在不全为零的常系数 c_1，c_2，…，c_n，使 $c_1 x_1 + c_2 x_2 + \cdots + c_n x_n = 0$ 成立，就称 x_1，x_2，…，x_n 为线性相关。例如，行向量 $x_1 = (1\ 2\ 3)$、$x_2 = (2\ 6\ 4)$和 $x_3 = (4\ 11\ 9)$是线性相关的，因为 $2x_1 + 3x_2 - 2x_3 = 0$。如果向量不是线性相关的，它们就是线性无关的。例如，单位矩阵中的各列向量就是线性无关的。

非零 $m \times n$ 矩阵 A 的列秩(column rank)是指 A 的极大线性无关列向量组中向量的个数。类似地，A 的行秩(row rank)是指 A 的极大线性无关行向量组中向量的个数。任意矩阵 A 的一个基本

性质就是其行秩与列秩总是相等，因此可以简单地说成 A 的秩。$m\times n$ 矩阵的秩是 0 和 $\min(m, n)$ 之间的一个整数，包含二者在内(零矩阵的秩为 0，$n\times n$ 单位矩阵的秩为 n)。还有一种更有用的关于矩阵的秩的等价定义。非零 $m\times n$ 矩阵 A 的秩是满足下列条件的最小的数 r：存在 $m\times r$ 矩阵 B 和 $r\times n$ 矩阵 C 且有 $A=BC$。如果一个 $n\times n$ 方阵的秩为 n，则它是满秩(full rank)。如果一个 $m\times n$ 矩阵的秩为 n，则它是列满秩。下列定理指出了秩的一个基本性质。

定理 28.1　一个方阵满秩当且仅当它为非奇异矩阵。 ■

矩阵 A 的空向量(null vector)是指满足 $Ax=0$ 的一个非零向量 x。下列定理与其推论找出了列秩和奇异性的概念与空向量之间的联系，其证明过程留作练习 28.1-9。

定理 28.2　当且仅当 A 无空向量时，矩阵 A 为列满秩。 ■

731　**推论 28.3**　当且仅当 A 具有空向量时，方阵 A 是奇异的。 ■

对于 $n>1$，$n\times n$ 矩阵 A 的第 ij 个余子式(minor)是把 A 的第 i 行和第 j 列的元素去掉后所形成的一个 $(n-1)\times(n-1)$ 矩阵 $A_{[ij]}$。$n\times n$ 矩阵 A 的行列式(determinant)可用其余子式递归定义如下：

$$\det(A)=\begin{cases}a_{11} & \text{如果 } n=1\\ \sum_{j=1}^{n}(-1)^{1+j}a_{1j}\det(A_{[1j]}) & \text{如果 } n>1\end{cases} \tag{28.7}$$

其中项 $(-1)^{i+j}\det(A_{[ij]})$ 称为元素 a_{ij} 的余子式。

下列定理说明了行列式的一些基本性质，其证明从略。

定理 28.4(行列式的性质)　方阵 A 的行列式具有如下性质：

- 如果 A 的任何行或列的元素为 0，则 $\det(A)=0$。
- 用常数 λ 乘 A 的行列式任意一行(或任意一列)的诸元素，等于用 λ 乘 A 的行列式。
- A 的行列式中一行(或一列)元素加上另一行(或另一列)中的相应元素，行列式的值不变。
- A 的行列式的值与其转置矩阵 A^{T} 的行列式的值相等。
- 行列式的任意两行(或两列)互换，则其值异号。

另外，对任意方阵 A 和 B，我们有 $\det(AB)=\det(A)\det(B)$。 ■

定理 28.5　当且仅当 $\det(A)=0$ 时，一个 $n\times n$ 方阵 A 是奇异的。 ■

正定矩阵

正定矩阵(positive-definite matrix)有很多重要应用。对于一个 $n\times n$ 矩阵 A，如果对所有 n 维向量 $x\neq 0$ 都有 $x^{\mathrm{T}}Ax>0$，则称矩阵 A 为正定矩阵。例如，单位矩阵是正定矩阵，因为对任何非零向量

$$x=(x_1 \quad x_2 \quad \cdots \quad x_n)^{\mathrm{T}}$$

732

$$x^{\mathrm{T}}I_n x=x^{\mathrm{T}}x=\sum_{i=1}^{n}x_i^2>0$$

我们将看到，由于有下列定理成立，实际应用中遇到的矩阵常常是正定矩阵。

定理 28.6　对任意列满秩矩阵 A，矩阵 $A^{\mathrm{T}}A$ 是正定的。

证明：必须证明对任意非零向量 x，都有 $x^{\mathrm{T}}(A^{\mathrm{T}}A)x>0$ 成立。对任意向量 x，

$$x^{\mathrm{T}}(A^{\mathrm{T}}A)x=(Ax)^{\mathrm{T}}(Ax) \quad (\text{见练习 } 28.1\text{-}2)=\|Ax\|^2$$

注意，$\|Ax\|^2$ 恰好是向量 Ax 中各元素的平方和。因此，如果 $\|Ax\|^2=0$，则 Ax 的每个元素都是 0，即 $Ax=0$。由于 A 是一个列满秩矩阵，$Ax=0$ 就蕴含着 $x=0$。因此，由定理 28.2 可知，$A^{\mathrm{T}}A$ 是正定矩阵。 ■

我们将在 28.5 节中讨论正定矩阵的一些其他性质。

练习

28.1-1　证明：如果 A 和 B 是 $n\times n$ 对称矩阵，则 $A+B$ 和 $A-B$ 也是对称的。

28.1-2　证明：$(AB)^{\mathrm{T}}=B^{\mathrm{T}}A^{\mathrm{T}}$，而且 $A^{\mathrm{T}}A$ 总是一个对称矩阵。

28.1-3　证明：矩阵的逆是唯一的，即如果 B 和 C 都是 A 的逆矩阵，则 $B=C$。

28.1-4　证明：两个下三角矩阵的乘积仍然是一个下三角矩阵。证明：一个下三角（或上三角）矩阵的行列式的值是其对角线上的元素之积。证明：一个下三角矩阵如果存在逆矩阵，则逆矩阵也是一个下三角矩阵。

28.1-5　证明：如果 P 是一个 $n\times n$ 置换矩阵，A 是一个 $n\times n$ 矩阵，则可以把 A 的各行进行置换以得到 PA，而把 A 的各列进行置换可得到 AP。证明：两个置换矩阵的乘积仍然是一个置换矩阵。证明：如果 P 是一个置换矩阵，则 P 是可逆矩阵，其逆矩阵是 P^{T}，且 P^{T} 也是一个置换矩阵。

733

28.1-6　设 A 和 B 是 $n\times n$ 矩阵，且有 $AB=I$。证明：如果把 A 的第 j 行加到第 i 行而得到 A'，则可以通过把 B 的第 j 列减去第 i 列而获得 A' 的逆矩阵 B'。

28.1-7　设 A 是一个非奇异的 $n\times n$ 复数矩阵。证明：当且仅当 A 的每个元素都是实数时，A^{-1} 的每个元素都是实数。

28.1-8　证明：如果 A 是一个 $n\times n$ 阶非奇异的对称矩阵，则 A^{-1} 也是一个对称矩阵。证明：如果 B 是一个任意的 $m\times n$ 矩阵，则由乘积 BAB^{T} 给出的 $m\times m$ 矩阵是对称的。

28.1-9　证明定理 28.2。亦即，证明如果矩阵 A 为列满秩当且仅当若 $Ax=0$，则说明 $x=0$。（提示：把列向量线性相关表示为矩阵向量等式。）

28.1-10　证明：对任意两个相容矩阵 A 和 B，$\mathrm{rank}(AB)\leqslant\min(\mathrm{rank}(A),\mathrm{rank}(B))$，其中等号仅当 A 或 B 是非奇异方阵时成立。（提示：利用矩阵秩的另一种等价定义。）

28.1-11　已知数 x_0，x_1，…，x_{n-1}，证明范德蒙德（Vandermonde）矩阵

$$V=(x_0,x_1,\cdots,x_{n-1})=\begin{pmatrix}1 & x_0 & x_0^2 & \cdots & x_0^{n-1}\\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1}\end{pmatrix}$$

的行列式

$$\det(V(x_0,x_1,\cdots,x_{n-1}))=\prod_{0\leqslant j<k\leqslant n-1}(x_k-x_j)$$

（提示：把第 i 列乘以 $-x_0$ 加到第 $i+1$ 列上，$i=n-1$，$n-2$，…，1，然后再使用归纳法。）

734

28.2　矩阵乘法的 Strassen 算法

本节要讨论求两个 $n\times n$ 矩阵乘积的著名的 Strassen 递归算法，其运行时间为 $\Theta(n^{\lg 7})=\Theta(n^{2.81})$。对足够大的 n，该算法在性能上超过了在 25.1 节中介绍的运行时间为 $\Theta(n^3)$ 的简易矩阵乘法算法 MATRIX-MULTIPLY。

算法概述

Strassen 算法可以看作是我们熟知的一种设计技巧——分治法的一种应用。假设我们希望计算乘积 $C=AB$，其中 A、B 和 C 都是 $n\times n$ 方阵。假定 n 是 2 的幂，把 A、B 和 C 都划分为四个

$n/2\times n/2$ 矩阵，等式 $C=AB$ 可改写如下：

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix}=\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & f \\ g & h \end{pmatrix} \tag{28.8}$$

（练习 28.2-2 处理了 n 不是 2 的幂的情形。）式(28.8)对应于以下四个等式

$$r=ae+bg \tag{28.9}$$

$$s=af+bh \tag{28.10}$$

$$t=ce+dg \tag{28.11}$$

$$u=cf+dh \tag{28.12}$$

上面每个等式都包含了两次 $n/2\times n/2$ 矩阵的乘法运算和两次乘积所得 $n/2\times n/2$ 矩阵的加法运算。利用这些等式来定义一个简单的分治策略，可以推出下列关于两个 $n\times n$ 矩阵相乘所需时间 $T(n)$的递归式：

$$T(n)=8T(n/2)+\Theta(n^2) \tag{28.13}$$

遗憾的是，递归式(28.13)的解为 $T(n)=\Theta(n^3)$，因此这种方法并不比普通的方法执行得快。

Strassen 发现了另外一种不同的递归方法，该方法只需要执行 7 次递归的 $n/2\times n/2$ 的矩阵乘法运算和 $\Theta(n^2)$次标量加法与减法运算，从而得到递归式

$$T(n)=7T(n/2)+\Theta(n^2)=\Theta(n^{\lg 7})=O(n^{2.81}) \tag{28.14}$$

735 Strassen 方法分为以下四个步骤：

1）如式(28.8)那样，把输入矩阵 A 和 B 划分为 $n/2\times n/2$ 的子矩阵。

2）运用 $\Theta(n^2)$次标量加法与减法运算，计算出 14 个 $n/2\times n/2$ 的矩阵 A_1，B_1，A_2，B_2，…，A_7，B_7。

3）递归计算出 7 个矩阵的乘积 $P_i=A_iB_i$，$i=1$，2，…，7。

4）仅使用 $\Theta(n^2)$次标量加法与减法运算，对 P_i 矩阵的各种组合进行求和或求差运算，从而获得结果矩阵 C 的四个子矩阵 r，s，t，u。

上述过程满足递归式(28.14)。现在我们所需要做的就是对上述步骤进行细化。

确定子矩阵的乘积

现在我们还不清楚 Strassen 当时是如何找出算法正常运行的关键——子矩阵乘积的。在此，我们重新构造一种似乎可能的发现方法。

我们猜想每个矩阵的积 P_i 可以写成如下形式

$$P_i=A_iB_i=(a_{i1}a+a_{i2}b+a_{i3}c+a_{i4}d)\cdot(\beta_{i1}e+\beta_{i2}f+\beta_{i3}g+\beta_{i4}h) \tag{28.15}$$

其中系数 a_{ij}、β_{ij} 都属于集合$\{-1,0,1\}$。也就是说，我们猜想：对矩阵 A 的某些子矩阵进行加减运算，并对矩阵 B 的某些矩阵进行加减运算，再把所得的结果相乘，从而计算出每个子矩阵积。当然还有一些策略也是可行的，但这种简单策略已被证明是行之有效的。

如果用这种方法得到所要求的乘积，则可以继续递归地运用这种方法，而无需假定乘法满足交换率，这是由于在每个乘积中，A 的所有子矩阵都在其左边，而 B 的所有子矩阵都在其右边。这种性质对能够递归地运用这种方法来说是很关键的，因为矩阵乘法不满足交换率。

为了方便起见，我们将用 4×4 矩阵来表示子矩阵乘积的线性组合，其中每个乘积如式(28.15)中那样，把 A 的一个子矩阵和 B 的一个子矩阵进行组合。例如，可以把式(28.9)改写为：

$$r = ae + bg = (a \quad b \quad c \quad d)\begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{matrix} \\ a \\ b \\ c \\ d \end{matrix}\begin{matrix} \begin{matrix} e & f & g & h \end{matrix} \\ \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{matrix}$$

上面最后的表达式使用了一种简略记号，其中“+”表示+1，“·”表示 0，“−”表示−1(从此以后，我们省略行和列的标号)。如果使用这种记号，就有如下结果矩阵 **C** 的其他子矩阵：

$$s = af + bh = \begin{pmatrix} \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$t = ce + dg = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \end{pmatrix}$$

$$u = cf + dh = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}$$

现在开始寻找一种有关矩阵乘法的快速算法。通过观察，我们发现子矩阵 s 可以由式 $s = P_1 + P_2$ 计算，其中分别使用一次矩阵乘法就可以计算出 P_1 和 P_2。

$$P_1 = A_1 B_1 = a \cdot (f - h) = af - ah = \begin{pmatrix} \cdot & + & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

736
⌇
737

$$P_2 = A_2 B_2 = (a + b) \cdot h = ah + bh = \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

矩阵 t 可用类似的方式 $t = P_3 + P_4$ 求得，其中

$$P_3 = A_3 B_3 = (c + d) \cdot e = ce + de = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}$$

和

$$P_4 = A_4 B_4 = d \cdot (g - e) = dg - de = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{pmatrix}$$

定义基本项为出现于式(28.9)～式(28.12)右端的八项中的一项。我们现在已使用了四个积

来计算两个基本项为 af、bh、ce 和 dg 的子矩阵 s 和 t。注意，P_1 计算基本项 af，P_2 计算基本项 bh，P_3 计算基本项 ce，P_4 计算基本项 dg。因此，现在所需要做的就是使用另外不多于 3 个的积，计算出剩余的两个子矩阵 r 和 u，其基本项为对角项 ae、bg、cf 和 dh。为了一次计算出两个基本项，现在用新方法计算 P_5：

$$P_5 = A_5 B_5 = (a+d)\cdot(e+h) = ae + ah + de + dh = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}$$

除了计算出基本项 ae 和 dh 外，P_5 还计算出非基本项 ah 和 de，从某种程度来说这两项是不常需要的。可以使用 P_4 和 P_2 来消去它们，但是又会产生另外两个非基本项：

$$P_5 + P_4 - P_2 = ae + dh + dg - bh = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \end{pmatrix}$$

通过加入另外一个积

$$P_6 = A_6 B_6 = (b-d)\cdot(g+h) = bg + bh - dg - dh = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{pmatrix}$$

但是，却得到

$$r = P_5 + P_4 - P_2 + P_6 = ae + bg = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

可以采用类似方法，通过运用 P_1 和 P_3 把 P_5 的非基本项移到不同的方向，来从 P_5 得到 u：

738
|
739

$$P_5 + P_1 - P_3 = ae + af - ce + dh = \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}$$

通过减去另外一个积

$$P_7 = A_7 B_7 = (a-c)\cdot(e+f) = ae + af - ce - cf = \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & - & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

可以得到

$$u = P_5 + P_1 - P_3 - P_7 = cf + dh = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}$$

这样，7 个子矩阵 P_1，P_2，…，P_7 就可以用于计算积 $C = AB$，至此就完成了对 Strassen 方法的论述。

讨论

从实用的观点来看，Strassen 算法通常不是矩阵乘法所选择的方法，原因有以下四点：

1)在 Strassen 算法的运行时间中，隐含的常数因子比简单的 $\Theta(n^3)$ 方法中的常数因子要大。

2)当矩阵是稀疏的时候，为稀疏矩阵设计的方法更快。

3)Strassen 算法不像简单方法那样具有数值稳定性。

4)在递归层次中生成的子矩阵要消耗空间。

后两个理由在 1990 年左右被缓解。Higham[145]显示了数值稳定性的差异被过度强调了；虽然 Strassen 算法在某些应用上是非常数值不稳定的，但对其他应用来说，则是在可接受的范围内。Bailey 等人[30]讨论了减少 Strassen 算法所需存储空间的技术。

在实际中，稠密矩阵上的快速矩阵乘法实现一般采取这样的做法，即对于矩阵规模超过一个"交叉点"时采用 Strassen 算法，一旦子问题的大小缩小到交叉点以下时，则改用简单方法。交叉点的准确数值与系统有很大的关系。在统计操作的数目、但忽略缓存和流水线效果的分析中，得到了低到 $n=8$(Higham[145])或 $n=12$(Huss-Lederman 等人[163])的交叉点。实验测量通常得到较高的交叉点，有时会低到 $n=20$。对任何指定的系统，通常是直接以实验来决定交叉点。 740

通过使用本书没有涉及的先进技术，实际上可以获得运行时间优于 $\Theta(n^{\lg 7})$ 的 $n\times n$ 矩阵乘法。目前，运行时间的最理想上界约为 $O(n^{2.376})$。最好的上界显然就是 $\Omega(n^2)$(说显然是因为我们必须在矩阵积中填满 n^2 个元素)。因此，目前我们还不知道矩阵乘法究竟有多困难。

练习

28.2-1　运用 Strassen 算法计算矩阵积

$$\begin{pmatrix}1 & 3\\ 5 & 7\end{pmatrix}\begin{pmatrix}8 & 4\\ 6 & 2\end{pmatrix}$$

说明计算过程。

28.2-2　如果 n 不是 2 的整数幂，应如何修改 Strassen 算法，以求出两个 $n\times n$ 矩阵的积？证明修改后的算法的运行时间为 $\Theta(n^{\lg 7})$。

28.2-3　如果使用 k 次乘法(假定乘法不满足交换律)就能计算出两个 3×3 矩阵的积，就能在 $o(n^{\lg 7})$ 时间内计算出两个 $n\times n$ 矩阵的积，满足上述条件的最大的 k 值是多少？这一算法的运行时间又将是多少？

28.2-4　V. Pan 发现了一种使用了 132 464 次乘法的求 68×68 矩阵积的方法，一种使用了 143 640 次乘法的求 70×70 矩阵积的方法，以及一种使用了 155 424 次乘法求 72×72 矩阵积的方法。当用于分治的矩阵乘法算法时，哪一种方法可以获得渐近意义上最好的运行时间？把它与 Strassen 算法的运行时间进行比较。

28.2-5　用 Strassen 算法作为子程序，能在多长时间内计算出一个 $kn\times n$ 矩阵与一个 $n\times kn$ 矩阵的乘积？如果把输入矩阵的次序颠倒一下则情况又如何？

28.2-6　说明如何仅用三次实数乘法运算，就可以计算复数 $a+bi$ 与 $c+di$ 的乘积。该算法应该把 a，b，c 和 d 作为输入，并分别生成实部 $ac-bd$ 和虚部 $ad+bc$ 的值。 741

28.3　求解线性方程组

对一组同时成立的线性方程求解是很多应用中都会出现的基本问题。一个线性系统可以表述为一个矩阵方程，其中每个矩阵或向量元素都属于一个域，如实数域 R。在本节中，我们讨论

任何运用 LUP 分解来求解线性方程组。

我们先来看看一组具有 n 个未知量 x_1，x_2，…，x_n 的线性方程：

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{28.16}$$

满足式(28.16)中所有方程的一组关于 x_1，x_2，…，x_n 的值称为方程组的一个解。在本节中，我们只讨论存在 n 个未知量的 n 个方程的情况。

式(28.16)中的方程可以方便地重写如下：

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

或是等价地，设 $\boldsymbol{A}=(a_{ij})$，$x=(x_i)$，$b=(b_i)$，有

$$Ax = b \tag{28.17}$$

如果 A 是非奇异矩阵，则它有逆矩阵 A^{-1}，并且

$$x = A^{-1}b \tag{28.18}$$

就是解向量。我们可以证明 x 是式(28.18)的唯一解。如果存在两个解 x 和 x'，则 $Ax = Ax' = b$，且有

$$x = (A^{-1}A)x = A^{-1}(Ax) = A^{-1}(Ax') = (A^{-1}A)x' = x'$$

742 在本节中，我们主要讨论 A 为非奇异矩阵，或者等价地(由定理 28.1)A 的秩等于未知量的个数 n 的情形。不过，对其他可能的情形，我们也作了简要讨论。如果方程的数目少于未知量数目(或更一般地，A 的秩小于 n)，则该线性方程组为欠定方程组。一个欠定方程组具有无穷多组解，虽然如果方程组不相容也可能无解。如果方程的数目超过未知量个数，则该方程组为超定方程组，并且方程组可能没有解。如何找出超定线性方程组的好的近似解将在 28.5 节中详细讨论。

现在让我们回到求解关于 n 个未知量的线性方程组 $Ax = b$ 的问题上来。一种方法是计算出 A^{-1}，然后方程组两边都乘以 A^{-1}，得 $A^{-1}Ax = A^{-1}b$，或 $x = A^{-1}b$。在实际应用中，这种方法的缺点是数值不稳定。幸运的是还有另一种方法——LUP 分解，该方法具有数值稳定性，且有运行速度要比前者快三倍的优点。

LUP 分解概述

LUP 分解的思想就是找出三个 $n \times n$ 矩阵 L、U 和 P，满足

$$PA = LU \tag{28.19}$$

其中

- L 是一个单位下三角矩阵
- U 是一个上三角矩阵
- P 是一个置换矩阵

称满足式(28.19)的矩阵 L、U 和 P 为矩阵 A 的一个 LUP 分解。我们将要证明每一个非奇异矩阵 A 都有这样一种分解。

对矩阵 A 进行 LUP 分解的优点是当相应矩阵为三角矩阵(如矩阵 L 和 U)时，更容易求解线性系统。在计算出 A 的 LUP 分解后，就可以用如下方式对三角形线性系统进行求解，也就获得

式(28.17)中 $Ax=b$ 的解。对 $Ax=b$ 的两边同时乘以 P，就得到等价的方程组 $PAx=Pb$，根据练习 28.1-5 可知，这相当于对式(28.16)的方程进行排列。运用式(28.19)的分解，得到 $LUx=Pb$。

现在，通过对两个三角形线性系统求解就可以得到该等式的解。定义 $y=Ux$，其中 x 就是要求的解向量。首先，用一种称为"正向替换"的方法求解下列下三角线性系统 743

$$Ly=Pb \tag{28.20}$$

得到未知向量 y。然后再用一种称为"逆向替换"的方法求解下列上三角线性系统

$$Ux=y \tag{28.21}$$

来得到未知量 x。由于置换矩阵 P 是可求逆的矩阵(练习 28.1-5)，因此向量 x 就是 $Ax=b$ 的解：

$$Ax=P^{-1}LUx=P^{-1}Ly=P^{-1}Pb=b$$

下一步将先说明正向替换与逆向替换是如何进行的，然后再解决如何计算 LUP 分解的问题。

正向替换与逆向替换

如果已知 L、P 和 b，用正向替换可以在 $\Theta(n^2)$ 的时间内求解下三角线性系统，见式(28.20)。为方便起见，用一个数组 $\pi[1..n]$ 来表示置换 P。对 $i=1,2,\cdots,n$，$\pi[i]$ 说明 $P_{i,\pi[i]}=1$，并且对 $j\neq\pi[i]$，有 $P_{ij}=0$。因此，PA 的位于第 i 行、第 j 列的元素为 $a_{\pi[i],j}$，Pb 的第 i 个元素为 $b_{\pi[i]}$。由于 L 是单位下三角矩阵，所以式(28.20)可以改写为：

$$\begin{aligned}
y_1 &= b_{\pi[1]}\\
l_{21}y_1+y_2 &= b_{\pi[2]}\\
l_{31}y_1+l_{32}y_2+y_3 &= b_{\pi[3]}\\
&\vdots\\
l_{n1}y_1+l_{n2}y_2+l_{n3}y_3+\cdots+y_n &= b_{\pi[n]}
\end{aligned}$$

由第一个等式我们可以直接求出 y_1 的值：$y_1=b_{\pi[1]}$。求出 y_1 的值后，把它代入第二个方程得到

$$y_2=b_{\pi[2]}-l_{21}y_1$$

现在把 y_1 和 y_2 的值代入第三个方程中，得到

$$y_3=b_{\pi[3]}-(l_{31}y_1+l_{32}y_2)$$

一般地，把 y_1，y_2，$\cdots$，y_{i-1}"正向"替换到第 i 个方程中就可求出 y_i 的解。 744

$$y_i=b_{\pi[i]}-\sum_{j=1}^{i-1}l_{ij}y_j$$

逆向替换类似于正向替换。如果已知 U 和 y，就能求出第 n 个方程的解，然后逆向逐次求解到第 1 个方程的解。和正向替换一样，这一过程的运行时间也是 $\Theta(n^2)$。由于 U 是一个上三角矩阵，我们可以把系统(式(28.21))改写为如下这样：

$$\begin{aligned}
u_{11}x_1+u_{12}x_2+\cdots+u_{1,n-2}x_{n-2}+u_{1,n-1}x_{n-1}+u_{1n}x_n &= y_1\\
u_{22}x_2+\cdots+u_{2,n-2}x_{n-2}+u_{2,n-1}x_{n-1}+u_{2n}x_n &= y_2\\
&\vdots\\
u_{n-2,n-2}x_{n-2}+u_{n-2,n-1}x_{n-1}+u_{n-2,n}x_n &= y_{n-2}\\
u_{n-1,n-1}x_{n-1}+u_{n-1,n}x_n &= y_{n-1}\\
u_{n,n}x_n &= y_n
\end{aligned}$$

因此，可以按如下方式顺序求出 x_n，x_{n-1}，$\cdots$，x_1 的解：

$$\begin{aligned}
x_n &= y_n/u_{n,n}\\
x_{n-1} &= (y_{n-1}-u_{n-1,n}x_n)/u_{n-1,n-1}
\end{aligned}$$

$$x_{n-2} = (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2}$$
$$\vdots$$

或者，一般地有，

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j\right)/u_{ii}$$

如果已知 P、L、U 和 b，下面的过程 LUP-SOLVE 通过把正向替换与逆向替换结合起来而求出 x 的解。代码中假定维数 n 出现在属性 $rows[L]$ 中，置换矩阵 P 用数组 π 表示。

```
LUP-SOLVE(L, U, π, b)
1  n ← rows[L]
2  for i ← 1 to n
3      do y_i ← b_π[i] − Σ_{j=1}^{i−1} l_ij y_j
4  for i ← n downto 1
5      do x_i ← (y_i − Σ_{j=i+1}^{n} u_ij x_j)/u_ii
6  return x
```

745

过程 LUP-SOLVE 在第 2～3 行中使用正向替换求出了 y 的解，然后在第 4～5 行中用逆向替换求出 x 的解。由于在每个 for 循环中都隐含了一个求和的循环过程，所以算法的运行时间为 $\Theta(n^2)$。

作为这两种方法的应用实例，我们来考察下列线性系统：

$$\begin{pmatrix}1 & 2 & 0\\ 3 & 4 & 4\\ 5 & 6 & 3\end{pmatrix}x = \begin{pmatrix}3\\ 7\\ 8\end{pmatrix}$$

其中

$$A = \begin{pmatrix}1 & 2 & 0\\ 3 & 4 & 4\\ 5 & 6 & 3\end{pmatrix} \quad b = \begin{pmatrix}3\\ 7\\ 8\end{pmatrix}$$

并且我们希望求得未知量 x 的解。其 LUP 分解如下：

$$L = \begin{pmatrix}1 & 0 & 0\\ 0.2 & 1 & 0\\ 0.6 & 0.5 & 1\end{pmatrix} \quad U = \begin{pmatrix}5 & 6 & 3\\ 0 & 0.8 & -0.6\\ 0 & 0 & 2.5\end{pmatrix} \quad P = \begin{pmatrix}0 & 0 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{pmatrix}$$

（读者可以验证 $PA = LU$。）用正向替换法求出 $Ly = Pb$ 的解 y：

$$\begin{pmatrix}1 & 0 & 0\\ 0.2 & 1 & 0\\ 0.6 & 0.5 & 1\end{pmatrix}\begin{pmatrix}y_1\\ y_2\\ y_3\end{pmatrix} = \begin{pmatrix}8\\ 3\\ 7\end{pmatrix}$$

通过先计算 y_1，然后计算 y_2，最后计算出 y_3，得到

$$y = \begin{pmatrix}8\\ 1.4\\ 1.5\end{pmatrix}$$

746

用逆向替换法求出 $Ux = y$ 的解 x

$$\begin{pmatrix}5 & 6 & 3\\ 0 & 0.8 & -0.6\\ 0 & 0 & 2.5\end{pmatrix}\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix} = \begin{pmatrix}8\\ 1.4\\ 1.5\end{pmatrix}$$

这样就获得所需的解(按 x_3，x_2，x_1 的顺序计算)：

$$x = \begin{pmatrix} -1.4 \\ 2.2 \\ 0.6 \end{pmatrix}$$

LU 分解的计算

现在我们已经证明对于一个非奇异矩阵 A，如果能计算出其 LUP 分解，则可以运用正向替换与反向替换来求出线性方程组 $Ax = b$ 的解。下面来说明如何有效地找出矩阵 A 的 LUP 分解。首先考虑 A 是 $n \times n$ 非奇异矩阵并且 P 默认(或等价地，$P = I_n$)的情形。在这种情况下，必须找出因式分解 $A = LU$。矩阵 L 和 U 称为 A 的 LU 分解。

把执行 LU 分解的过程称为高斯消元法。先从其他方程中减去第一个方程的倍数，以便把那些方程中的第一个变量消去。然后，再从第三个以后的方程中减去第二个方程的倍数，以把这些方程的第一个和第二个变量都消去。继续上述过程，直至系统变为一个上三角矩阵形式，这个矩阵就是 U。矩阵 L 是由使得变量被消去的行的乘数所组成。

采用递归方法可以实现这一策略。我们希望构造出 $n \times n$ 非奇异矩阵 A 的一个 LU 分解。如果 $n = 1$，则完成构造，因为可以指定 $L = I_1$，$U = A$。对于 $n > 1$，把 A 拆成四个部分：

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}$$

其中 v 是一个 $n-1$ 维列向量，w^{T} 是一个 $n-1$ 维行向量，A' 是一个 $(n-1) \times (n-1)$ 矩阵。 747

然后，运用矩阵代数(简单地使用乘数来验证方程式)，可以把 A 分解为：

$$A = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} \tag{28.22}$$

上述分解的第一个矩阵与第二个矩阵中的 0 分别表示 $n-1$ 维行向量与 $n-1$ 维列向量。项 vw^{T}/a_{11} 是一个 $(n-1) \times (n-1)$ 矩阵，它是向量 v 与 w 的外积矩阵的每一个元素除以 a_{11} 所得到的矩阵，与矩阵 A' 是相容的。所得 $(n-1) \times (n-1)$ 矩阵

$$A' - vw^{\mathrm{T}}/a_{11} \tag{28.23}$$

称为矩阵 A 对于 a_{11} 的 Schur 补。

如果 A 是非奇异的，则 Schur 补也是非奇异的。为什么？假设 $(n-1) \times (n-1)$ Schur 补是奇异的，则根据定理 28.1，它的行秩严格小于 $n-1$。因为在矩阵的第一列的底部 $n-1$ 个元素

$$\begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}$$

都是 0，这个矩阵的底部 $n-1$ 行的行秩必须严格小于 $n-1$。因此这个矩阵的行秩严格小于 n。应用练习 28.1-10 到式(28.22)，A 的行秩严格小于 n，且根据定理 28.1，我们得出 A 是奇异的矛盾。

因为 Schur 补是非奇异的，现在我们可以用递归方法来找出它的 LU 分解。我们说

$$A' - vw^{\mathrm{T}}/a_{11} = L'U'$$

其中 L' 是一个单位下三角矩阵，U' 是一个上三角矩阵。运用矩阵代数可得：

$$A = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} = LU$$

因此就找出了 A 的 LU 分解。(注意，因为 L' 是单位下三角矩阵，所以 L 也是单位下三角矩阵，
748 又因为 U' 是上三角矩阵，所以 U 也是上三角矩阵。)

当然，如果 $a_{11}=0$，就不能运用这种方法，否则就会产生除数为 0 的情形。如果 Schur 补 $A'-vw^{\mathrm{T}}/a_{11}$ 的左上角元素为 0，这种方法也行不通，因为在下一次递归过程中就要除以该元素。在 LUP 分解中，所除以的元素称为主元(pivot)，它们处于矩阵 U 的对角线上。在 LUP 分解中要包含一个置换矩阵 P 的原因，就是为了避免把 0 作为除数。利用置换来避免除数为 0(或一个很小的数)的过程称为选主元。

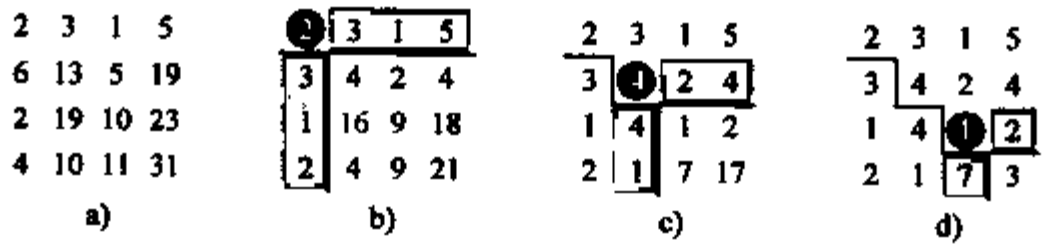
保证 LU 分解总能进行的一类重要矩阵就是对称正定矩阵。对这一类矩阵无需选主元，因此可以从容地运用上述递归策略，而无需担心除数为 0。我们将在 28.5 节中证明这一结论和其他一些结论。

我们对矩阵进行 LU 分解的代码就是根据上述递归策略设计的，只不过用迭代循环代替了递归过程(这一转化是对"尾递归"过程进行标准的最优化处理，所谓"尾递归"过程是指过程的最后一个操作是对其自身进行递归调用的过程)。代码假定 A 的规模保存在属性 $rows[A]$ 中。因为我们知道输出矩阵 U 的对角线以下的元素均为 0，并且过程 LU-SOLVE 并没有查看这些元素，所以代码也没有填入这些元素。同样，因为输出矩阵 L 的对角线上的元素都是 1，对角线以上的元素都是 0，所以也没有把这些元素填入矩阵。因此，下列代码仅对 L 和 U 的"有意义"的元素进行了计算。

```
LU-DECOMPOSITION(A)
1  n ← rows[A]
2  for k ← 1 to n
3       do u_kk ← a_kk
4          for i ← k+1 to n
5               do l_ik ← a_ik / u_kk ▷ l_ik holds v_i
6                  u_ki ← a_ki        ▷ u_ki holds w_i^T
7          fori ← k+1 to n
8               do for j ← k+1 to n
9                     do a_ij ← a_ij − l_ik u_kj
10 return L and U
```

从第 2 行开始的外层 for 循环对每个递归步迭代一次。在该循环内的第 3 行确定出主元 $u_{kk}=a_{kk}$。在第 4～6 行的 for 循环内(当 $k=n$ 时，该循环不执行)，用 v 和 w^{T} 两个向量对 L 和 U 进行更新。在第 5 行中确定出向量 v 的各元素，并把 v_i 存放于 l_{ik} 中，第 6 行确定出向量 w^{T} 的各元素，并把 w_i^{T} 存放于 u_{ki} 中。最后，在第 7～9 行计算 Schur 补中的各元素，并把它们存放在矩阵
749 A 的相应元素中(在第 9 行我们无需除以 a_{kk}，因为在第 5 行中计算 l_{ik} 时已经做过了)。因为第 9 行语句处于三层嵌套之中，所以 LU-DECOMPOSITION 的运行时间为 $\Theta(n^3)$。

图 28-1 说明了 LU-DECOMPOSITION 的操作过程。其中 L 和 U 的每一个有意义的元素都存放在矩阵 A 的适当位置上。也就是说，可以在每个元素 a_{ij} 与 l_{ij}(若 $i>j$)或 u_{ij}(若 $i\leqslant j$)之间建立某种对应关系，更新矩阵 A，使得过程终止执行时矩阵 A 包含 L 和 U 中的有意义的元素。要获得这一优化的代码，只要把上述代码中的 l 或 u 用 a 取代就可以了；不难证明这一转化方法仍然使算法保持其正确性。



$$\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix} \begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

A　　L　　U

图 28-1　LU-DECOMPOSITION 的执行过程。a)矩阵 A。b)在黑色圆圈内的元素 $a_{11}=2$ 是主元，阴影列是 v/a_{11}，阴影行是 w^{T}。目前计算好的 U 的元素在水平线之上，而 L 的元素是在竖直线的左边。Schur 补矩阵 $A'-vw^{\mathrm{T}}/a_{11}$ 占据了右下方。c)现在我们在图 b)部分所产生的 Schur 补矩阵上操作。在黑色圆圈内的元素 $a_{22}=4$ 是主元，而阴影列和阴影行分别是 v/a_{22} 和 w^{T}(在 Schur 补的划分中)。线段将这个矩阵划分成目前所计算的 U 的元素(上)、目前所计算的 L 的元素(左)以及新的 Schur 补(右下)。d)下一个步骤完成分解(当递归终止时，新的 Schur 补中的元素 3 成为 U 的一部分)。e)分解 $A=LU$

LUP 分解的计算

一般情况下，为了求线性方程组 $Ax=b$ 的解，必须在 A 的非对角线元素中选主元以避免除数为 0。除数不仅不能为 0，也不能很小(即使 A 是非奇异的)，否则就会在计算中导致数值不稳定。因此，所选的主元必须是一个较大的值。

750

LUP 分解的数学基础与 LU 分解相似。回顾上面的内容，我们已知一个 $n\times n$ 非奇异矩阵 A，并希望计算出一个置换矩阵 P、一个单位下三角矩阵 L 和一个上三角矩阵 U，并满足条件 $PA=LU$。在如 LU 分解中那样对 A 进行分解以前，先把一个非零元素，如 a_{k1}，从第一列中某个位置移到矩阵(1，1)的位置上(如果第 1 列仅包含 0 元素，则矩阵 A 是奇异矩阵，因为由定理 28.4 和定理 28.5 可知其行列式的值为 0)。为了使方程组仍然保持成立，把第 1 行与第 k 行互换。这实际上等价于用一个置换矩阵 Q 作乘以 A 的乘法(练习 28.1-5)。因此，可以把 QA 写成

$$QA=\begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}$$

其中 $v=(a_{21}, a_{31}, \cdots, a_{n1})^{\mathrm{T}}$，但用 a_{11} 代替了 a_{k1}；$w^{\mathrm{T}}=(a_{k2}, a_{k3}, \cdots, a_{kn})$；$A'$ 是一个 $(n-1)\times(n-1)$矩阵。因为 $a_{k1}\neq 0$，所以可以执行与 LU 分解相同的线性代数运算，但能保证除数不会为 0：

$$QA=\begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}=\begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix}\begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A'-vw^{\mathrm{T}}/a_{k1} \end{pmatrix}$$

如同在 LU 分解中一样，如果 A 是非奇异的，则 Schur 补 $A'-vw^{\mathrm{T}}/a_{k1}$ 也是非奇异的。所以，我们可以推导地找出它的一个 LUP 分解，包括单位下三角矩阵 L'、上三角矩阵 U' 和置换矩阵 P'，满足

$$P'(A'-vw^{\mathrm{T}}/a_{k1})=L'U'$$

定义

$$P=\begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix}Q$$

它是一个置换矩阵，因为它是两个置换矩阵的乘积(练习 28.1-5)。现在有

$$PA=\begin{pmatrix}1 & 0\\ 0 & P'\end{pmatrix}QA=\begin{pmatrix}1 & 0\\ 0 & P'\end{pmatrix}\begin{pmatrix}1 & 0\\ v/a_{kl} & I_{n-1}\end{pmatrix}\begin{pmatrix}a_{kl} & w^{\mathrm{T}}\\ 0 & A'-vw^{\mathrm{T}}/a_{kl}\end{pmatrix}$$

$$=\begin{pmatrix}1 & 0\\ P'v/a_{kl} & P'\end{pmatrix}\begin{pmatrix}a_{kl} & w^{\mathrm{T}}\\ 0 & A'-vw^{\mathrm{T}}/a_{kl}\end{pmatrix}=\begin{pmatrix}1 & 0\\ P'v/a_{kl} & I_{n-1}\end{pmatrix}\begin{pmatrix}a_{kl} & w^{\mathrm{T}}\\ 0 & P'(A'-vw^{\mathrm{T}}/a_{kl})\end{pmatrix}$$

$$=\begin{pmatrix}1 & 0\\ P'v/a_{kl} & I_{n-1}\end{pmatrix}\begin{pmatrix}a_{kl} & w^{\mathrm{T}}\\ 0 & L'U'\end{pmatrix}=\begin{pmatrix}1 & 0\\ P'v/a_{kl} & L'\end{pmatrix}\begin{pmatrix}a_{kl} & w^{\mathrm{T}}\\ 0 & U'\end{pmatrix}=LU$$

这样就得到了 LUP 分解。因为 L' 是一个单位下三角矩阵，所以 L 也是单位下三角矩阵，又因为 U' 是个上三角矩阵，所以 U 也是一个上三角矩阵。

注意在上述推导过程中，与 LU 分解中不同的是，列向量 v/a_{kl} 和 Schur 补 $A'-vw^{\mathrm{T}}/a_{kl}$ 都必须与置换矩阵 P' 相乘。

与 LU-DECOMPOSITION 一样，我们为 LUP 分解设计的伪代码也采用循环迭代来代替递归过程。作为对直接实现递归的一种改进，我们设置了数组 π 来表示置换矩阵 P，$\pi[i]=j$ 说明 P 的第 i 行、第 j 列的元素为 1。实现代码时，在 A 中"原地"计算 L 和 U。因此，当该过程终止执行时，有

$$a_{ij}=\begin{cases}l_{ij} & \text{如果 } i>j\\ u_{ij} & \text{如果 } i\leqslant j\end{cases}$$

```
LUP-DECOMPOSITION(A)
 1  n←rows[A]
 2  for i ← 1 to n
 3       do π[i]← i
 4  for k ← 1 to n
 5      do p ← 0
 6         for i ← k to n
 7            do if | a_ik | >p
 8                 then p ← | a_ik |
 9                      k' ← i
10         if p=0
11           then error "singular matrix"
12         exchange π[k]↔ π[k']
13         for i ← 1 to n
14            do exchange a_ki ↔a_k'i
15         for i ← k+1 to n
16            do a_ik ← a_ik /a_kk
17               for j ← k+1 to n
18                  do a_ij ← a_ij −a_ik a_kj
```

751 ~ 752

图 28-2 说明了过程 LUP-DECOMPOSITION 是如何对矩阵进行分解的。数组 π 在第 2～3 行被初始化以表示单位矩阵。从第 4 行开始的外层 for 循环实现了递归过程。每执行一次外层循环，第 5～9 行确定要找出其 LU 分解的 $(n-k+1)\times(n-k+1)$ 矩阵中当前第一列(列 k)的绝对值最大的元素 $a_{k'k}$。如果在当前第一列中的所有元素都是 0，则第 10～11 行报告该矩阵为奇异矩阵。为了选主元，在第 12 行中把 $\pi[k']$ 与 $\pi[k]$ 交换。在第 13～14 行中，把矩阵 A 的第 k 行与第 k' 行进行交换，这样就选出了主元 a_{kk}(要对整个行进行交换，是因为在上述方法的推导过程中，不仅 $A'-vw^{\mathrm{T}}/a_{kl}$ 与 P' 相乘，而且向量 v/a_{kl} 也与 P' 相乘)。最后，在第 15～18 行中计算出 Schur 余式，所用的方法与 LU-DECOMPOSITION 中第 4～9 行的计算方法相同，不过这里的计算过程是在 A 中"原地"进行的。

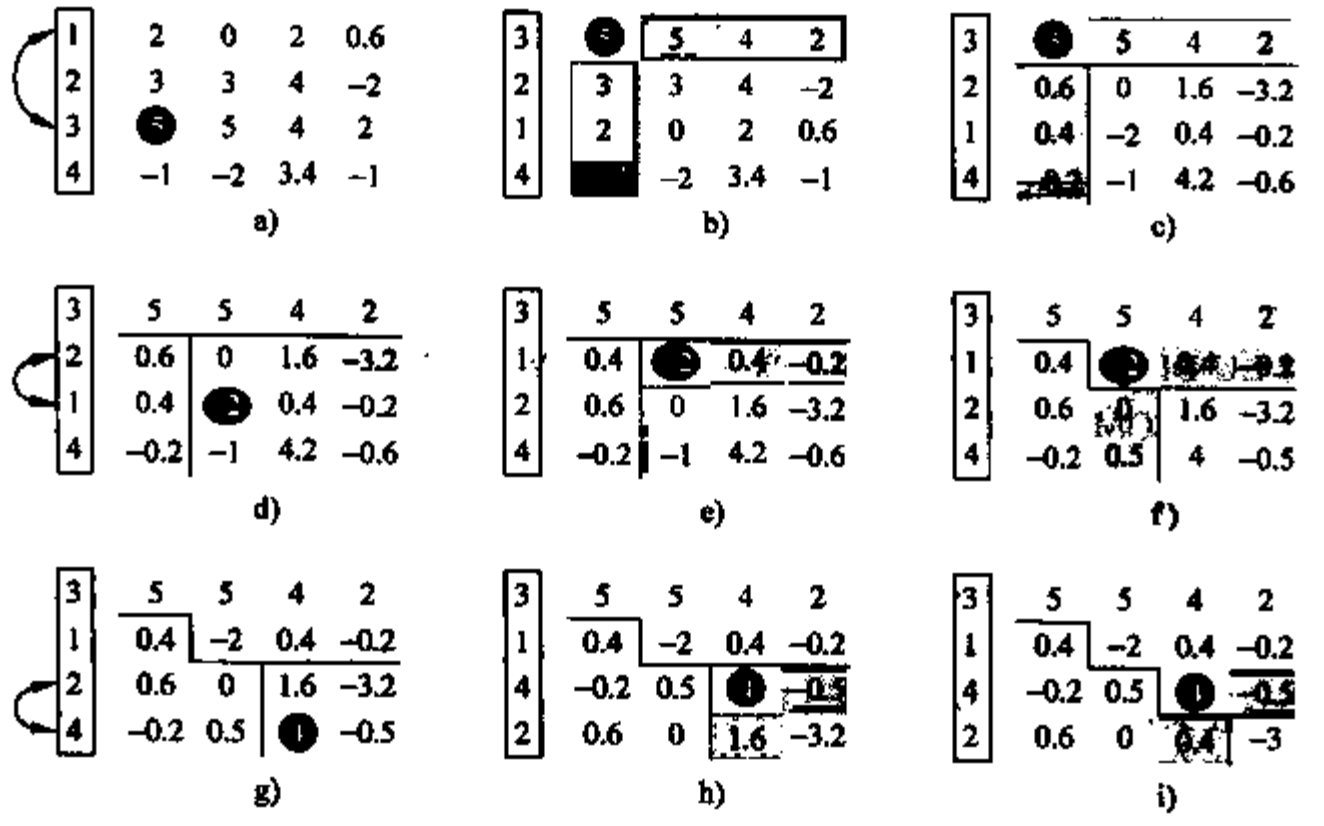


图 28-2　LUP-DECOMPOSITION 的操作过程。a)输入矩阵 A，行的单位置换在其左方。算法的第一步确定在第 3 行黑色圆圈中的元素 5 是第一列的主元。b)第 1 行与第 3 行互换，并且更新了置换。阴影列和阴影行分别表示 v 和 w^T。c)向量 v 被 $v/5$ 所代替，且矩阵的右下方使用 Schur 补来更新。线条将矩阵分割成 3 个区域：U 的元素(上)、L 的元素(左)、Schur 补的元素(右下)。d)至 f)为第二步。g)至 i)为第三步。没有变化发生在第四步以及其后。j)LUP 分解 $PA = LU$

因为上述过程中的三重嵌套结构，所以过程 LUP-DECOMPOSITION 的运行时间为 $\Theta(n^3)$，与 LU-DECOMPOSITION 的运行时间相同。因此，选主元的过程在运行时间中至多占一个常数因子的部分。

练习

28.3-1　运用正向替换法求解下列方程组：

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

28.3-2　求出下列矩阵的 LU 分解：

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}$$

28.3-3　运用 LUP 分解来求解下列方程组：

$$\begin{pmatrix}1 & 5 & 4\\ 2 & 0 & 3\\ 5 & 8 & 2\end{pmatrix}\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix}=\begin{pmatrix}12\\ 9\\ 5\end{pmatrix}$$

753 ~ 754

28.3-4 试描述一个对角矩阵的 LUP 分解。

28.3-5 试描述一个置换矩阵 A 的 LUP 分解，并证明它是唯一的。

28.3-6 证明：对所有 $n\geqslant 1$，存在具有 LU 分解的奇异的 $n\times n$ 矩阵。

28.3-7 在 LU-DECOMPOSITION 中，当 $k=n$ 时是否有必要执行最外层的 for 循环迭代？在 LUP-DECOMPOSITION 中情况又是怎样？

28.4　矩阵求逆

在实际应用中，一般并不使用逆矩阵来求线性方程组的解，而是运用一些更具数值稳定性的技术，如 LUP 分解来求解线性方程组，但是，有时候仍然需要计算一个矩阵的逆矩阵。在本节中，我们论述如何利用 LUP 分解来计算逆矩阵。此外，还将证明矩阵乘法和计算逆矩阵问题是具有相同难度的两个问题，即(在技术条件限制下)可以使用一个算法在相同渐近时间内解决另外一个问题。所以，可以使用 Strassen 矩阵乘法算法来求一个矩阵的逆。确实，正是由于要证明可以用比通常的办法更快的算法来求解线性方程组，才推动了最初的 Strassen 算法的产生。

根据 LUP 分解计算逆矩阵

假设有一个矩阵 A 的 LUP 分解，包括三个矩阵 L、U 和 P，并满足 $PA=LU$。如果运用 LU-SOLVE，则可以在 $\Theta(n^2)$的运行时间内，求出形如 $Ax=b$ 的线性系统的解。由于 LUP 分解仅取决于 A 而不取决于 b，所以就能够再用 $\Theta(n^2)$的运行时间，求出形如 $Ax=b'$的另一个线性方程组的解。一般地，一旦得到了 A 的 LUP 分解，就可以在 $\Theta(kn^2)$的运行时间内，求出 k 个形如 $Ax=b$的线性方程组的解，这 k 个方程组中只有 b 不相同。

方程

$$AX=I_n \tag{28.24}$$

可以看作形如 $Ax=b$ 的 n 个不同的方程组的集合。这些方程定义矩阵 X 为 A 的逆矩阵。更准确地说，设 X_i 表示 X 的第 i 列，并且单位向量 e_i 是 I_n 的第 i 列。则可以通过用 LUP 分解分别求出每个方程组的 X_i 的方法，来对 X 求解得式(28.24)

755

$$AX_i=e_i$$

可以在 $\Theta(n^2)$的运行时间内求出 n 列中的每个 X_i 的解，因此根据 A 的 LUP 分解来计算 X 的运行时间为 $\Theta(n^3)$。由于在 $\Theta(n^3)$的时间内可以计算出 A 的 LUP 分解，所以也可以在 $\Theta(n^3)$的时间内确定出 A 的逆矩阵 A^{-1}。

矩阵乘法与逆矩阵

现在我们证明，对矩阵乘法可以获得的的理论上的加速，可以相应地加速求逆矩阵的运算。事实上，可以证明更强的结论：从下面的意义上说，求逆矩阵运算等价于矩阵乘法运算。如果 $M(n)$表示求两个 $n\times n$ 矩阵的乘积所需要的时间，则有在 $O(M(n))$时间内对一个 $n\times n$ 矩阵求逆的方法。如果 $I(n)$表示对一个非奇异的 $n\times n$ 矩阵求逆所需的时间，则有在 $O(I(n))$时间内对两个 $n\times n$ 矩阵相乘的方法。下面用两个分开的定理来证明这些结果。

定理 28.7(矩阵乘法不比求逆矩阵困难)　如果能在 $I(n)$时间内求出一个 $n\times n$ 矩阵的逆矩阵，其中 $I(n)=\Omega(n^2)$且满足正则条件 $I(3n)=O(I(n))$，则可以在 $O(I(n))$时间内求出两个 $n\times n$ 矩阵的乘积。

证明：设 A 和 B 为两个 $n\times n$ 矩阵，我们希望计算出其乘积 C。定义 $3n\times 3n$ 矩阵 D 如下所示：

$$D=\begin{pmatrix} I_n & A & 0\\ 0 & I_n & B\\ 0 & 0 & I_n \end{pmatrix}$$

D 的逆矩阵为

$$D^{-1}=\begin{pmatrix} I_n & -A & AB\\ 0 & I_n & -B\\ 0 & 0 & I_n \end{pmatrix}$$

因此可以利用 D^{-1} 的右上角的 $n\times n$ 子矩阵计算出乘积 AB。

我们能够在 $\Theta(n^2)=O(I(n))$ 的时间内构造出矩阵 D，然后用 $O(I(3n))=O(I(n))$ 的运行时间求出 D 的逆矩阵（根据关于 $I(n)$ 的正则条件）。因此有 $M(n)=O(I(n))$。■

注意，对任意常数 $c>0$ 和 $d\geqslant 0$，只要 $I(n)=\Theta(n^c\lg^d n)$，$I(n)$ 就能满足正则条件。

求逆矩阵运算并不比矩阵乘法运算更困难。对这一问题的证明依赖于对称正定矩阵的一些性质，这些性质将在 28.5 节中证明。 756

定理 28.8（求逆矩阵运算并不比矩阵乘法运算更困难） 如果能在 $M(n)$ 的时间内计算出两个 $n\times n$ 实矩阵的乘积，其中 $M(n)=\Omega(n^2)$ 且 $M(n)$ 满足两个正则条件：对任意的 $0\leqslant k\leqslant n$ 有 $M(n+k)=O(M(n))$，以及对某个常数 $c<1/2$ 有 $M(n/2)\leqslant cM(n)$。则可以在 $O(M(n))$ 时间内求出任何一个 $n\times n$ 非奇异实矩阵的逆矩阵。

证明：可以假定 n 是 2 的幂，这是因为对任意 $k>0$，有

$$\begin{pmatrix} A & 0\\ 0 & I_k \end{pmatrix}^{-1}=\begin{pmatrix} A^{-1} & 0\\ 0 & I_k \end{pmatrix}$$

因此，通过挑选 k 以使 $n+k$ 为 2 的幂，可以使矩阵的规模扩大到 2 的下一个幂，并且从这一规模扩大的答案中获得我们要求的 A^{-1}。$M(n)$ 的第一个正则条件保证这一扩展对运行时间的影响不会超过一个常数因子。

目前，假定 $n\times n$ 矩阵 A 是对称正定矩阵。把 A 划分为 4 个 $n/2\times n/2$ 的子矩阵：

$$A=\begin{pmatrix} B & C^{\mathrm{T}}\\ C & D \end{pmatrix} \tag{28.25}$$

那么，如果设

$$S=D-CB^{-1}C^{\mathrm{T}} \tag{28.26}$$

是 A 关于 B 的 Schur 补（我们将在 28.5 节看到更多的关于这种形式的 Schur 补），则有

$$A^{-1}=\begin{pmatrix} B^{-1}+B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1}\\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix} \tag{28.27}$$

可以用矩阵乘法来验证上式的正确性（利用 $AA^{-1}=I_n$）。如果 A 是对称正定矩阵，根据引理 28.9、引理 28.10 和引理 28.11 可知，B^{-1} 和 S^{-1} 都存在，因为 B 和 S 都是对称正定矩阵。由练习 28.1-2 可知，$B^{-1}C^{\mathrm{T}}=(CB^{-1})^{\mathrm{T}}$，$B^{-1}C^{\mathrm{T}}S^{-1}=(S^{-1}CB^{-1})^{\mathrm{T}}$。因此式(28.26)和式(28.27)可用于说明包含 4 个 $n/2\times n/2$ 矩阵乘法的递归算法，

$$C\cdot B^{-1}$$
$$(CB^{-1})\cdot C^{\mathrm{T}}$$
$$S^{-1}\cdot(CB^{-1})$$

757

$$(CB^{-1})^{\mathrm{T}} \cdot (S^{-1}CB^{-1})$$

所以，通过对两个 $n/2\times n/2$ 矩阵（B 和 S）进行求逆，执行 $n/2\times n/2$ 矩阵乘法（可以使用 $n\times n$ 矩阵的一个算法来做），加上从 A 内提取子矩阵以及在这些 $n/2\times n/2$ 矩阵上执行常数次数的加法与减法的一个额外代价 $O(n^2)$，可以对一个 $n\times n$ 的对称正定矩阵求逆。得到递归式：

$$I(n)\leqslant 2I(n/2)+4M(n)+O(n^2)=2I(n/2)+\Theta(M(n))=O(M(n))$$

第二行成立是因为 $M(n)=\Omega(n^2)$，而第三行成立是因为在这个定理中，表述的第二个正则条件允许我们应用主定理（定理 4.1）的情况 3。

现在还需证明，当 A 可逆但不是对称正定矩阵时，对矩阵求逆运算，也可以达到和矩阵乘法一样的渐近运行时间。基本思想是，对任意的非奇异矩阵 A，矩阵 $A^{\mathrm{T}}A$ 是对称的（练习 28.1-2）正定矩阵（见定理 28.6）。主要技巧在于把求 A 的逆矩阵的问题转化为求 $A^{\mathrm{T}}A$ 的逆矩阵的问题。

这一转化是基于下列观察的基础之上：当 A 为一个非奇异的 $n\times n$ 矩阵时，有

$$A^{-1}=(A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}$$

这是因为 $((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})A=(A^{\mathrm{T}}A)^{-1}(A^{\mathrm{T}}A)=I_n$，并且一个矩阵的逆矩阵是唯一的。因此，可以这样来计算 A^{-1}：先把 A^{T} 与 A 相乘获得 $A^{\mathrm{T}}A$，然后运用上面的分治算法求出对称正定矩阵的逆矩阵，最后再把结果乘以 A^{T}。这三步中每一步的运行时间都是 $O(M(n))$，因此可以在 $O(M(n))$ 的运行时间内求出任意非奇异实矩阵的逆矩阵。■

在定理 28.8 的证明过程中，对 A 是非奇异矩阵的情形，提出了一种通过无需选主元的 LU 分解，就能求解方程组 $Ax=b$ 的方法。把方程 $Ax=b$ 的两边同时乘以 A^{T}，得到 $(A^{\mathrm{T}}A)x=A^{\mathrm{T}}b$。由于 A^{T} 是可逆的，所以这一变换不会影响解向量 x，这样就可以通过计算 LU 分解，来对对称正定矩阵 $A^{\mathrm{T}}A$ 进行分解。然后通过运用正向替换和逆向替换，就可以求得方程右端是 $A^{\mathrm{T}}b$ 的解向量 x。尽管这种方法在理论上是正确的，但实际上过程 LUP-DECOMPOSITION 执行得更快。LUP 分解所需的算术运算次数要比前者少一个常数因子，并且从某种程度来说，LUP 分解有着
758 更好的数值性质。

练习

28.4-1 设 $M(n)$ 是求 $n\times n$ 矩阵的乘积所需的时间，$S(n)$ 表示求 $n\times n$ 矩阵的平方所需的时间。证明：求矩阵乘积运算与求矩阵平方运算实质上难度相同；一个 $M(n)$ 时间的矩阵相乘算法蕴含着一个 $O(M(n))$ 时间的矩阵平方算法，一个 $S(n)$ 时间的矩阵平方算法蕴含着一个 $O(S(n))$ 时间的矩阵相乘算法。

28.4-2 设 $M(n)$ 是求 $n\times n$ 矩阵的乘积所需的时间，$L(n)$ 为计算一个 $n\times n$ 矩阵的 LUP 分解所需要的时间。证明：求矩阵乘积运算与计算矩阵 LUP 分解实质上难度相同：一个 $M(n)$ 时间的矩阵相乘算法蕴含着一个 $O(M(n))$ 时间的矩阵 LUP 分解算法，一个 $L(n)$ 时间的矩阵 LUP 分解算法蕴含着一个 $O(L(n))$ 时间的矩阵相乘算法。

28.4-3 设 $M(n)$ 是求 $n\times n$ 矩阵的乘积所需的时间，$D(n)$ 表示求 $n\times n$ 矩阵的行列式的值所需要的时间。证明：求矩阵乘积运算与求行列式的值实质上难度相同：一个 $M(n)$ 时间的矩阵相乘算法蕴含着一个 $O(M(n))$ 时间的行列式算法，一个 $D(n)$ 时间的行列式算法蕴含着一个 $O(D(n))$ 时间的矩阵相乘算法。

28.4-4 设 $M(n)$ 是求 $n\times n$ 布尔矩阵的乘积所需的时间，$T(n)$ 为找出 $n\times n$ 布尔矩阵的传递闭包所需要的时间（参见 25.2 节）。证明：一个 $M(n)$ 时间的布尔矩阵相乘算法蕴含着一个 $O(M(n)\lg n)$ 时间的传递闭包算法，一个 $T(n)$ 时间的传递闭包算法蕴含着一个 $O(T(n))$ 时间的布尔矩阵相乘算法。

28.4-5　当矩阵元素属于整数模 2 所构成的域时，基于定理 28.8 的求逆矩阵算法是否能够运行？试解释其原因。

*28.4-6　推广基于定理 28.8 的求逆矩阵算法，使之能处理复矩阵的情形，并证明所给出的推广方法是正确的。（提示：用 A 的共轭转置矩阵 A^* 来代替 A 的转置矩阵 A^T，把 A^T 中的每个元素用其共轭复数来取代就得到 A^*。考虑用 Hermitian 矩阵来代替对称矩阵，所谓 Hermitian 矩阵就是满足 $A=A^*$ 的矩阵 A。）

28.5　对称正定矩阵与最小二乘逼近

对称正定矩阵有许多有趣而很理想的性质。例如，它们都是非奇异矩阵，并且可以对其进行 LU 分解而无需担心出现除数为 0 的情况。在本节中，我们将证明其他几条关于对称正定矩阵的性质，并给出一个用最小二乘来进行曲线拟合的有趣应用实例。

我们要证明的第一条性质大概也是最基本的一条性质。

引理 28.9　任意对称正定矩阵都是非奇异矩阵。

证明：假设矩阵 A 是奇异的，则由推论 28.3 可知存在一个向量 x，满足 $Ax=0$。因此，$x^TAx=0$，这样 A 就不可能是正定矩阵。■

要证明可以对一个对称正定矩阵 A 进行 LU 分解而不会出现除数为 0 的情况，还要涉及另外一些知识。先来证明关于 A 的某些子矩阵的性质。定义 A 的第 k 个前导子矩阵 A_k 为 A 的前 k 行和前 k 列交叉的元素组成的矩阵。

引理 28.10　如果 A 是一个对称正定矩阵，则 A 的每一个主子式都是对称正定的。

证明：每个主子式 A_k 显然都是对称的。为了证明 A_k 是正定的，假设其不是正定然后导出矛盾。如果 A_k 不是正定的，则存在一个 k 维向量 $x_k\neq 0$ 使得 $x_k^TA_kx_k\leqslant 0$。设 A 是 $n\times n$ 矩阵，定义 n 维向量 $x=(x_k^T\ 0)^T$，其中 x_k 之后有 $n-k$ 个 0。因此我们有

$$x^TAx=\begin{pmatrix}x_k^T & 0\end{pmatrix}\begin{pmatrix}A_k & B^T\\ B & C\end{pmatrix}\begin{pmatrix}x_k\\ 0\end{pmatrix}=\begin{pmatrix}x_k^T & 0\end{pmatrix}\begin{pmatrix}A_kx_k\\ Bx_k\end{pmatrix}=x_k^TA_kx_k\leqslant 0$$

这与 A 是正定矩阵相矛盾。■

现在我们来讨论 Schur 补的几条重要性质。设 A 是一个对称正定矩阵，A_k 是 A 的 $k\times k$ 主子式。把 A 划分为

$$A=\begin{pmatrix}A_k & B^T\\ B & C\end{pmatrix} \tag{28.28}$$

推广定义（式(28.23)）以定义矩阵 A 关于 A_k 的 Schur 补为

$$S=C-BA_k^{-1}B^T \tag{28.29}$$

（根据引理 28.10，A_k 是对称正定的；因此由引理 28.9 可知 A_k^{-1} 存在，且 S 是良定义的。）注意，若设 $k=1$，则先前对 Schur 补的定义（式(28.23)）与定义（式(28.29)）是一致的。

下面的一条引理说明，对称正定矩阵的 Schur 补本身也是对称正定的。我们在定理 28.8 中用到了该结论，并且还要用其推论来证明对称正定矩阵的 LU 分解的正确性。

引理 28.11（Schur 补引理）　如果 A 是一个对称正定矩阵，A_k 是 A 的 $k\times k$ 主子式。则 A 关于 A_k 的 Schur 补也是对称正定的。

证明：因为 A 是对称的，所以子式 C 也是对称的。由练习 28.1-8 可知 $BA_k^{-1}B^T$ 是对称的，再根据练习 28.1-1 可知 S 是对称矩阵。

现在还要证明 S 是正定的。考察式(28.28)中对 A 的划分。对任何非零向量 x，根据 A 是正

定矩阵的假设有 $x^{\mathrm{T}}Ax>0$。把 x 拆成两个子向量 y 和 z，分别与 A_k 和 C 相容。因为 A_k^{-1} 存在，所以有

$$x^{\mathrm{T}}Ax=\begin{pmatrix}y^{\mathrm{T}} & z^{\mathrm{T}}\end{pmatrix}\begin{pmatrix}A_k & B^{\mathrm{T}}\\ B & C\end{pmatrix}\begin{pmatrix}y\\ z\end{pmatrix}=\begin{pmatrix}y^{\mathrm{T}} & z^{\mathrm{T}}\end{pmatrix}\begin{pmatrix}A_ky+B^{\mathrm{T}}z\\ By+Cz\end{pmatrix}=y^{\mathrm{T}}A_ky+y^{\mathrm{T}}B^{\mathrm{T}}z+z^{\mathrm{T}}By+z^{\mathrm{T}}Cz$$

$$=(y+A_k^{-1}B^{\mathrm{T}}z)^{\mathrm{T}}A_k(y+A_k^{-1}B^{\mathrm{T}}z)+z^{\mathrm{T}}(C-BA_k^{-1}B^{\mathrm{T}})z \tag{28.30}$$

(可以用矩阵乘法来验证。)最后这个式子相当于二次型的"全平方"(参见练习 28.5-2)。

因为 $x^{\mathrm{T}}Ax>0$ 对任意非零向量 x 都成立，我们可以挑选一个任意的非零向量 z，然后选择
761 $y=-A_k^{-1}B^{\mathrm{T}}z$，这样就把式(28.30)中的第一项消去，剩下 $z^{\mathrm{T}}(C-BA_k^{-1}B^{\mathrm{T}})z=z^{\mathrm{T}}Sz$ 作为表达式的值。对任意 $z\neq 0$，我们有 $z^{\mathrm{T}}Sz=x^{\mathrm{T}}Ax>0$，所以 S 是正定的。■

推论 28.12　对一个对称正定矩阵进行 LU 分解不会出现除数为 0 的情形。

证明：设 A 是一个对称正定矩阵。我们将证明一个比推论更强的结论：每个主元都严格为正。第一个主元为 a_{11}。设 e_1 是第一个单位向量，由此我们得到 $a_{11}=e_1^{\mathrm{T}}Ae_1>0$。因为 LU 分解的第一步产生的是 A 关于 $A_1=(a_{11})$ 的 Schur 补，所以引理 28.11 说明由归纳可知所有的主元都是正值。■

最小二乘逼近

对给定一组数据的点进行曲线拟合是对称正定矩阵的一个重要应用。假设给定 m 个数据点 $(x_1, y_1), (x_2, y_2), \cdots, (x_m, y_m)$，其中已知 y_i 受到测量误差的影响。我们希望找出一个函数 $F(x)$，满足对 $i=1, 2, \cdots, m$，有

$$y_i = F(x_i) + \eta_i \tag{28.31}$$

其中近似误差 η_i 是很小的。函数 $F(x)$ 的形式依赖于我们所遇到的问题。在此，假定它的形式为线性加权和：

$$F(x) = \sum_{j=1}^{n} c_j f_j(x)$$

其中和项的个数和特定的基函数 f_j 取决于我们对问题的了解。一种选择是 $f_j(x)=x^{j-1}$，这说明

$$F(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1}$$

是一个 x 的 $n-1$ 次多项式。

通过选择 $n=m$，就能确切地计算出式(28.31)中的每一个 y_i。这样一个高次函数 F 尽管容易处理数据，但也容易对数据产生"干扰"，并且一般在对未预见到的 x 预测其相应的 y 值时，其
762 精确性也是很差的。通常，较好的做法是选择比 m 小得多的 n，并且通过选择系数 c_j，使我们获得的函数 F 能够发现数据点的显著模式。从理论上讲，选择 n 要满足一些原则，但这超出了本书所讨论的范围。在任何情况下，一旦选定了 n，就得到了我们希望尽可能好地对其求解的一个超定方程组。现在来说明这个过程。

设

$$A=\begin{pmatrix}f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1)\\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2)\\ \vdots & \vdots & \ddots & \vdots\\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m)\end{pmatrix}$$

表示基函数在给定点的值的矩阵。即 $a_{ij}=f_j(x_i)$。设 $c=(c_k)$ 表示所求系数组成的 n 维向量。则

$$\boldsymbol{Ac}=\begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix}\begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}=\begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix}$$

是由 y 的“被预见的值”组成的 m 维向量。因此，

$$\eta = \boldsymbol{Ac} - \boldsymbol{y}$$

是逼近误差的 m 维向量。

为了使逼近误差最小，我们选定使误差向量 η 的范数最小，就得到一个最小二乘解，因为

$$\|\eta\| = \left(\sum_{i=1}^{m}\eta_i^2\right)^{1/2}$$

又因为

$$\|\eta\|^2 = \|\boldsymbol{Ac}-\boldsymbol{y}\|^2 = \sum_{i=1}^{m}\left(\sum_{j=1}^{n}a_{ij}c_j - y_i\right)^2$$

可以通过对每个 c_k 求 $\|\eta\|^2$ 的微分并令结果为 0，来求出 $\|\eta\|$ 的最小值。 763

$$\frac{\mathrm{d}\|\eta\|^2}{\mathrm{d}c_k} = \sum_{i=1}^{m}2\left(\sum_{j=1}^{n}a_{ij}c_j - y_i\right)a_{ik} = 0 \qquad (28.32)$$

对 $k=1, 2, \cdots n$，式(28.32)中的 n 个方程等价于下面的矩阵方程

$$(\boldsymbol{Ac}-\boldsymbol{y})^{\mathrm{T}}\boldsymbol{A} = 0$$

或等价地(利用练习 28.1-2)，

$$\boldsymbol{A}^{\mathrm{T}}(\boldsymbol{Ac}-\boldsymbol{y}) = 0$$

这意味着

$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{Ac} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{y} \qquad (28.33)$$

在统计学中，该式称为正态方程(normal equation)。由练习 28.1-2 可知 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ 是对称矩阵，并且如果 $\boldsymbol{A}$ 是列满秩，则根据定理 28.6 可知 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ 也是正定矩阵。因此 $(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})^{-1}$ 存在，并且式(28.33)的解为

$$c = ((\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})^{-1}\boldsymbol{A}^{\mathrm{T}})\boldsymbol{y} = \boldsymbol{A}^{+}\boldsymbol{y} \qquad (28.34)$$

其中矩阵 $\boldsymbol{A}^{+}=((\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})^{-1}\boldsymbol{A}^{\mathrm{T}})$ 称为矩阵 $\boldsymbol{A}$ 的伪逆矩阵。伪逆矩阵是逆矩阵概念在 $\boldsymbol{A}$ 不是方阵的情形中的自然推广。(比较：式(28.34)作为 $\boldsymbol{Ac}=\boldsymbol{y}$ 的近似解，$\boldsymbol{A}^{-1}b$ 作为 $\boldsymbol{Ax}=b$ 的精确解。)

作为最小二乘拟合的一个实例，假定有五个点的数据：

$$(x_1, y_1) = (-1, 2)$$
$$(x_2, y_2) = (1, 1)$$
$$(x_3, y_3) = (2, 1)$$
$$(x_4, y_4) = (3, 0)$$
$$(x_5, y_5) = (5, 3)$$

如图 28-3 中的黑点所示。我们希望用一个二次多项式对这些点进行拟合

$$F(x) = c_1 + c_2 x + c_3 x^2$$

首先构造出基函数值的矩阵

$$\boldsymbol{A} = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix}$$

764

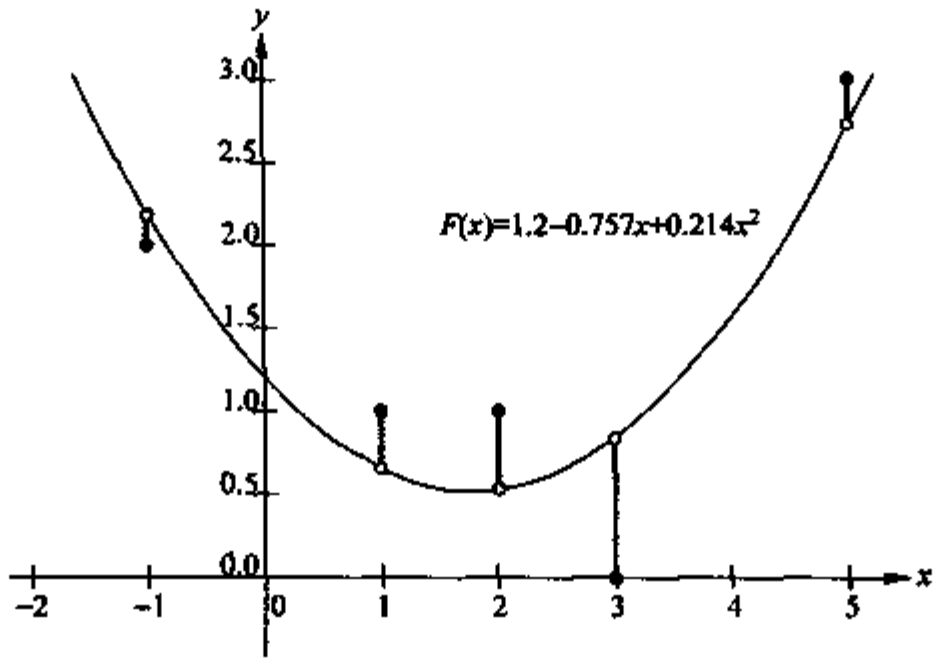


图 28-3　对五个点数据的集合{(−1，2)，(1，1)，(2，1)，(3，0)，(5，3)}用二次多项式进行最小二乘拟合。黑点是数据点，而白点是由多项式 $F(x)=1.2-0.757x+0.214x^2$ 所预测到的估计值，这个二次多项式使得平方误差之和最小。每个数据点的误差以一条阴影线来表示

其伪逆矩阵为

$$A^+ = \begin{pmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{pmatrix}$$

再把 A^+ 与 y 相乘，就得到系数向量

$$c = \begin{pmatrix} 1.200 \\ -0.757 \\ 0.214 \end{pmatrix}$$

其对应于二次多项式

$$F(x) = 1.200 - 0.757x + 0.214x^2$$

在最小二乘意义上，上式是对已知数据的最接近的二次拟合。

在实际应用中，按如下方式求正态方程(式(28.33))的解：先把 A^T 与 y 相乘，然后求出 A^TA 的 LU 分解。如果 A 为列满秩，则可以保证矩阵 A^TA 为非奇异矩阵，这是因为它是对称正
765 定矩阵(参见练习 28.1-2 和定理 28.6)。

练习

28.5-1　证明：对称正定矩阵的对角线上的每一个元素都是正值。

28.5-2　设 $A=\begin{pmatrix} a & b \\ b & c \end{pmatrix}$ 是一个 2×2 的对称正定矩阵。运用引理 28.11 的证明过程中用过的"全平方"来证明其行列式的值 $ac-b^2$ 是正的。

28.5-3　证明：一个对称正定矩阵中值最大的元素处于其对角线上。

28.5-4　证明：一个对称正定矩阵的每一个主子式的行列式的值都是正的。

28.5-5　设 A_k 表示对称正定矩阵 A 的第 k 个主子式。证明在 LU 分解中，$\det(A_k)/\det(A_{k-1})$ 是第 k 个主元，为方便起见，设 $\det(A_0)=1$。

28.5-6　找出形如 $F(x)=c_1+c_2x\lg x+c_3e^x$ 的函数，使其为下列数据点的最优最小二乘拟合：(1，1)，(2，1)，(3，3)，(4，8)。

28.5-7　证明：伪逆矩阵 A^+ 满足下列四个等式：

$$AA^+A=A$$
$$A^+AA^+=A^+$$
$$(AA^+)^T=AA^+$$
$$(A^+A)^T=A^+A$$

766

思考题

28-1　三对角线性方程组

考察三对角矩阵：

$$A=\begin{pmatrix}1&-1&0&0&0\\-1&2&-1&0&0\\0&-1&2&-1&0\\0&0&-1&2&-1\\0&0&0&-1&2\end{pmatrix}$$

a)求出矩阵 A 的 LU 分解。

b)通过运用正向替换与逆向替换求方程 $Ax=(1\quad 1\quad 1\quad 1\quad 1)^T$ 的解。

c)求 A 的逆矩阵。

d)证明：对任意的 $n\times n$ 对称正定的三对角矩阵 A 和任意 n 维向量 b，通过进行 LU 分解可以在 $O(n)$ 的时间内求出方程 $Ax=b$ 的解。论证在最坏情况下，从渐近意义上来看，基于求出 A^{-1} 的任何方法都要花费更多的时间。

e)证明：对任意 $n\times n$ 非奇异的三对角矩阵 A 和任意 n 维向量 b，通过进行 LUP 分解，可以在 $O(n)$ 的运行时间内求出方程 $Ax=b$ 的解。

28-2　样条

把一组点插值到一个曲线中的一种实用方法是运用三次样条。已知 $n+1$ 个点-值的对组成的集合 $\{(x_i, y_i): i=0, 1, \cdots, n\}$，其中 $x_0<x_1<\cdots<x_n$。我们希望拟合出这些点的分段三次曲线(样条) $f(x)$。也就是说，曲线 $f(x)$ 由 n 个三次多项式 $f_i(x)=a_i+b_ix+c_ix^2+d_ix^3$ 组成，$i=0, 1, \cdots, n-1$，其中如果 $x_i\leqslant x\leqslant x_{i+1}$，则曲线的值由式 $f(x)=f_i(x-x_i)$ 给出。把三次多项式"粘"在一起的点 x_i 称为结(knot)。为了简单起见，假定 $x_i=i$，$i=0, 1, \cdots, n$。

为了保证 $f(x)$ 的连续性，要求当 $i=0, 1, \cdots, n-1$ 时

$$f(x_i)=f_i(0)=y_i$$
$$f(x_{i+1})=f_i(1)=y_{i+1}$$

为了保证 $f(x)$ 足够光滑，我们同时要求当 $i=0, 1, \cdots, n-1$ 时，每个结的一阶导数是连续的：

767

$$f'(x_{i+1})=f_i'(1)=f_{i+1}'(0)$$

a)假定当 $i=0, 1, \cdots, n$ 时，不仅已知点-值对 $\{(x_i, y_i)\}$，而且已知每个结的一阶导数 $D_i=f'(x_i)$。试用值 y_i，y_{i+1}，D_i 和 D_{i+1} 来表示每个系数 a_i，b_i，c_i 和 d_i(注意 $x_i=i$)。根据点-值对和一阶导数来计算出 $4n$ 个系数需要多少时间？

如何选择 $f(x)$ 在每个结的一阶导数依然是个问题。一种方法是要求二阶导数在每个

结保持连续(当 $i=0, 1, \cdots, n-1$ 时)

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

在第一个结和最后一个结，我们假定 $f''(x_0)=f_0''(0)=0$ 以及 $f''(x_n)=f_{n-1}''(1)=0$；这些假设使得 $f(x)$成为一个自然三次样条。

b)利用二阶导数的连续性来证明当 $i=1, 2, \cdots, n-1$ 时，

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) \tag{28.35}$$

c)证明

$$2D_0 + D_1 = 3(y_1 - y_0) \tag{28.36}$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}) \tag{28.37}$$

d)把式(28.35)～式(28.37)改写为包含关于未知量的向量 $D=\langle D_0, D_1, \cdots, D_n\rangle$的一个矩阵方程。在所给的方程中矩阵具有什么属性?

e)论证用自然三次样条可以在 $O(n)$时间内对一组 $n+1$ 个点-值对进行插值(参见思考题 28-1)。

f)试说明即使当 x_i 不一定等于 i 时，如何确定出一个自然三次样条对一组 $n+1$ 个满足 $x_0<x_1<\cdots<x_n$ 的点(x_i, y_i)进行插值。需要求解什么样的矩阵方程? 所给出的算法的运行速度有多快?

本章注记

768　在很多优秀的教科书中，对数值和科学计算的介绍都比本章所述的内容要详细得多。下列参考材料特别值得阅读：George 和 Liu [113]；Golub 和 Van Loan [125]；Press、Flannery、Teukolsky 和 Vetterling [248, 249]；Strang [285, 286]。

Golub 和 Van Loan [125]讨论了数值稳定性。他们说明了为什么 det(A)不一定是一个矩阵 A 稳定性的好的指标，并提议改用$\|A\|_\infty \|A^{-1}\|_\infty$，其中$\|A\|_\infty = \max\limits_{1\leqslant i\leqslant n}\sum\limits_{j=1}^{n}|a_{ij}|$。他们还同时探讨了如何计算这个数值而不实际计算 A^{-1}。

1969 年出版的 Strassen 算法[287]引起了非常大的震撼。在那之前，很难想像简单算法可以被改进。矩阵乘法的难度的渐近上界从那时开始被显著改进。到目前为止，$n\times n$ 矩阵相乘的渐近效率最高的算法是由 Coppersmith 和 Winograd [70]提出的，其运行时间为 $O(n^{2.376})$。Strassen 算法的图形展示由 Paterson[238]提出的。

高斯消元法是 LU 和 LUP 分解的基础，是第一个解决线性方程组的系统方法。它也是最早的数值算法之一。在较早之前人们就已经知道这一算法了，它的发现一般是归功于 C. F. Gauss (1777—1855)。在他的著作[287]中，Strassen 同时还展示了可以在 $O(n^{\lg 7})$时间内对一个 $n\times n$ 矩阵求逆。Winograd[317]最早证明矩阵乘法不会比矩阵求逆困难，而反向的证明则来自于 Aho、Hopcroft 和 Ullman [5]。

另外一种重要的矩阵分解是奇异值分解(singular value decomposition，SVD)。在 SVD 中，一个$m\times n$矩阵 A 被分解为 $A=Q_1\Sigma Q_2^{\mathrm{T}}$，其中$\Sigma$是一个只在对角线上有非零元素的 $m\times n$ 矩阵，Q_1 是一个列相互正交的 $m\times m$ 矩阵，Q_2 是一个列相互正交的 $n\times n$ 矩阵。如果两个向量的内积为 0 并且范数都是 1，则它们是正交的。Strang 的著作[285, 286]以及 Golub 和 Van Loan[125]均含有对 SVD 很好的处理。

769　Strang[286]有一个关于对称正定矩阵和一般线性代数的很好介绍。

第29章 线性规划

在给定有限的资源和竞争约束情况下，很多问题都可以表述为最大化或最小化某个目标。如果可以把目标指定为某些变量的一个线性函数，而且如果可以将资源的约束指定为这些变量的等式或不等式，则得到一个线性规划问题(linear-programming problem)。线性规划出现在许多实际应用中。我们从一次选举政治策略的应用开始研究它。

一个政治问题

假设你是一位政治家，试图赢得一场选举。你的选区包括三种不同类型的区域——市区、郊区和乡村。这些区域分别有 100 000、200 000 和 50 000 个登记选民。为了有效控制选举局面，你希望在这三个选区中的每一个选区都赢得大多数的选票。你是正直的，并且从不支持你不相信的政策。然而你认识到，在某些地方，某些政策对赢取选票来说会更为有效。你的主要政策是修筑更多的道路、枪械管制、农业补贴以及增加公共运输的汽油税。根据你的竞选班子的研究，你可以估计通过在每项政策上花费 1000 美元做广告，在每个居民区可以赢取或输掉多少选票。图 29-1中的表格给出这种信息。在此表格中，每项描述通过花费 1000 美元广告费支持某个特定政策，在市区、郊区或乡村可以赢得选民的千人数。负数项表示丢失的选民数。你的任务是计算你需要花费最少的钱，去赢得 50 000 张市区选票、100 000 张郊区选票和 25 000 张乡村选票。

政策	市区	郊区	乡村
修路	−2	5	3
枪械管制	8	2	−5
农业补贴	0	0	10
汽油税	10	0	−2

图 29-1 政策对选民的影响。每项表示通过花费 1000 美元广告费支持一项特定政策，可以赢得的市区、郊区或乡村的选民的千人数。负数项表示将丢失的选票数

通过反复试验，有可能得到赢得所需选票的一种策略，但这种策略可能不是花费最少的。例如，你可以支出 20 000 美元广告费给修筑道路、0 美元给枪械管制、4000 美元给农业补贴，9000 美元给汽油税。在这种情况下，你将赢得 20(−2)+0(8)+4(0)+9(10)=50 千张市区选票，20(5)+0(2)+4(0)+9(0)=100 千张郊区选票，以及 20(3)+0(−5)+4(10)+9(−2)=82 千张乡村选票。你将在市区和郊区赢得你想要的准确票数，而在乡村地区则赢得超过所需的票数。(事实上，在乡村地区，得到的选票比选民数量还多!)为了累积这些选票，你需要付出 20+0+4+9=33 千美元的广告费。

自然你可能怀疑你的策略是否是最好的。也就是说，你是否能花费更少的广告费来达到你的这些目标？额外的反复试验可以帮助你回答这个问题，但是你宁愿有一个系统化的方法来回答这样的问题。为此，我们将以数学的语言来表述这个问题。我们引入四个变量：

- x_1 是支出在修筑道路广告上的千美元数。
- x_2 是支出在枪械管制广告上的千美元数。
- x_3 是支出在农业补贴广告上的千美元数。
- x_4 是支出在汽油税广告上的千美元数。

可以将赢得至少 50 000 张市区选票的需求写成

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geqslant 50 \tag{29.1}$$

类似地，可以将赢得至少 100 000 张郊区选票和 25 000 张乡村选票的需求写成

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geqslant 100 \tag{29.2}$$

和

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geqslant 25 \tag{29.3}$$

变量 x_1、x_2、x_3 和 x_4 满足不等式(29.1)～不等式(29.3)的任何一组值都是赢得足够数量的每种票数的一个策略。为了使花费尽量地小，我们希望最小化广告的费用。也就是说，要最小化表达式

770
～
771

$$x_1 + x_2 + x_3 + x_4 \tag{29.4}$$

虽然在政治竞选活动中负面的广告宣传时常发生，但是不可能有负的广告费用。相应地，我们需要限定

$$x_1 \geqslant 0, x_2 \geqslant 0, \quad x_3 \geqslant 0 \text{ 和 } x_4 \geqslant 0 \tag{29.5}$$

将最小化式(29.4)的目标和不等式(29.1)至不等式(29.3)以及不等式(29.5)联立，得到一个称为"线性规划"的问题。将这个问题格式化为以下形式：

最小化

$$x_1 + x_2 + x_3 + x_4 \tag{29.6}$$

满足约束条件

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geqslant 50 \tag{29.7}$$

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geqslant 100 \tag{29.8}$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geqslant 25 \tag{29.9}$$

$$x_1, x_2, x_3, x_4 \geqslant 0 \tag{29.10}$$

由这个线性规划的解即可得到这个政治家的一个最优策略。

一般线性规划

在一般线性规划的问题中，我们希望最优化一个满足一组线性不等式约束的线性函数。已知一组实数 a_1，a_2，…，a_n 和一组变量 x_1，x_2，…，x_n，在这些变量上的一个线性函数 f 定义为

$$f(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = \sum_{j=1}^{n} a_jx_j$$

如果 b 是一个实数而 f 是一个线性函数，则等式

$$f(x_1, x_2, \cdots, x_n) = b$$

是一个线性等式，而不等式

$$f(x_1, x_2, \cdots, x_n) \leqslant b$$

和

$$f(x_1, x_2, \cdots, x_n) \geqslant b$$

都是线性不等式。我们使用名词线性约束来表示线性等式或线性不等式。在线性规划中，我们不
772 允许严格的不等式。正式地说，线性规划问题是这样的一种问题，它要最小化或最大化一个受限于一组有限的线性约束的线性函数。如果是要最小化，则称此线性规划为最小化线性规划；如果是要最大化，则称此线性规划为最大化线性规划。

本章的余下部分将介绍线性规划的形式化和求解。虽然有一些线性规划的多项式时间算法，但在本章中并不研究它们。相反地，我们研究单纯形算法，它是最古老的线性规划算法。单纯形算法在最坏的情况下不是在多项式时间内运行，但是它相当有效，而且在实际中被广泛使用。

线性规划概述

为了描述线性规划的性质和算法，使用规范的形式来表示它们是很方便的。在本章中使用两种形式：标准的和松弛的。29.1 节中将给出它们的精确定义。非正式地，在标准型中的线性规划是约束为线性不等式的线性函数的最大化，而松弛型的线性规划是约束为线性等式的线性函数的最大化。通常使用标准型来表示线性规划，但当描述单纯形算法的细节时，使用松弛形式会比较方便。从现在开始，我们将注意力集中在受 m 个线性不等式约束的 n 个变量上的线性函数的最大化。

首先考虑下面包含两个变量的线性规划：

最大化

$$x_1 + x_2 \tag{29.11}$$

满足约束

$$4x_1 - x_2 \leqslant 8 \tag{29.12}$$

$$2x_1 + x_2 \leqslant 10 \tag{29.13}$$

$$5x_1 - 2x_2 \geqslant -2 \tag{29.14}$$

$$x_1, x_2 \geqslant 0 \tag{29.15}$$

则称满足所有约束(式(29.12)～式(29.15))的变量 x_1 和 x_2 的任何设定为线性规划的一个可行解。如果在(x_1，x_2)——笛卡儿坐标系统中画出这些约束的图形，如图 29-2a 所示，可以看到可行解的集合(在图形中是阴影的)在二维空间中构成一个凸形区域[⊖]，称这个凸形区域为可行区域。要最大化的函数称为目标函数。在概念上，可以在可行区域内的每个点上评估目标函数 x_1+x_2；将目标函数在一个特定点上的值称为目标值。我们可以识别出一个有最大目标值的点作为最优解。在这个例子中(以及大多数的线性规划中)，可行区域包含无限个数的点，所以我们希望找出一个有效的方式来找到一个取最大目标值的点，而无需在可行区域的每个点上评估目标函数。 773

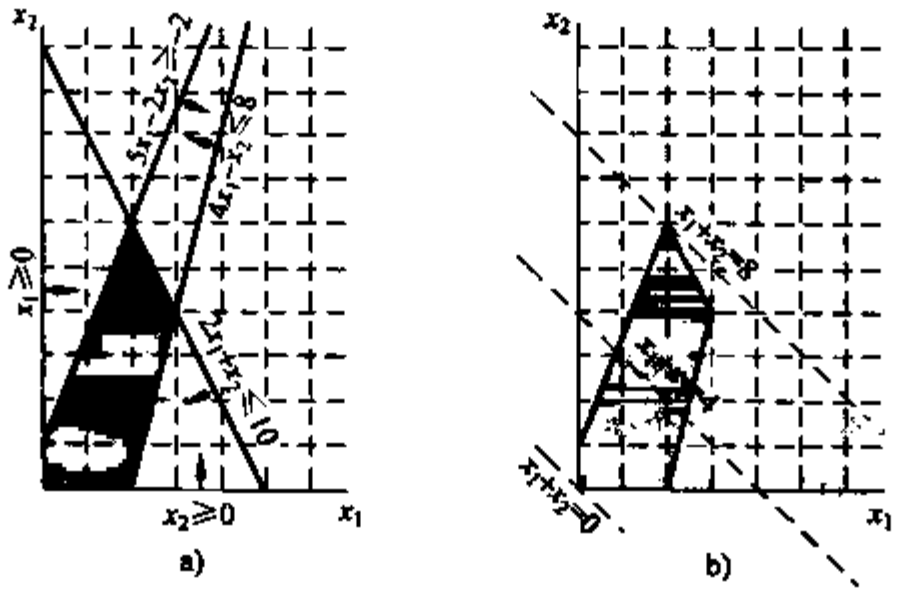


图 29-2　a)在式(29.12)～式(29.15)中给出的线性规划。每个约束以一条直线和一个方向来表示。约束的交集即可行区域以阴影表示。b)虚线分别表示目标值为 0、4 和 8 的点。线性规划的最优解是 $x_1=2$，$x_2=6$，目标值是 8

在二维中，可以利用一个图形程序来最优化。对任意的 z，$x_1+x_2=z$ 的点的集合是斜率为 1 的一条直线。如果画出 $x_1+x_2=0$，则得到通过原点的斜率为 -1 的直线，如图 29-2b 所示。

⊖　凸形区域的一个直觉定义是它满足需求：区域内的任何两点之间连线上的点都属于这个区域。

这条直线与可行区域的交集是目标值为 0 的可行解的集合。在这个情况下，直线与可行区域的交集是点(0，0)。更一般地，对任意的 z，直线 $x_1+x_2=z$ 与可行区域的交集是目标值为 z 的可行解的集合。图 29-2b 显示了直线 $x_1+x_2=0$、$x_1+x_2=4$ 以及 $x_1+x_2=8$。因为在图 29-2 中可行区域是有界的，所以必定存在某个最大值 z 使得直线 $x_1+x_2=z$ 和可行区域的交集非空。任何让此情况出现的点都是线性规划的一个最优解，在这个情况下，最优解是 $x_1=2$，$x_2=6$，目标值
774 是 8。线性规划的最优解出现在可行区域的一个顶点上并不是偶然的。使得直线 $x_1+x_2=z$ 与可行区域相交的 z 的最大值必定是在可行区域的边界上，因此这条直线与可行区域的边界的交集或是一个顶点，或是一条线段。如果交集是一个顶点，那么只有一个最优解，而且它是一个顶点。如果交集是一条线段，那么线段上的每一点都有相同的目标值；特别地，线段的两个端点都是最优解。因为线段的每个端点都是一个顶点，所以在这个情况下最优解也在一个顶点上。

虽然我们无法容易地画出用图形表示超过两个变量的线性规划，但是同样的直观认识仍然成立。如果有三个变量，则每个约束以三维空间的一个半空间来描述。三个半空间的交集构成了可行区域。目标函数取目标值 z 的点集合是一个平面。如果目标函数的系数都是非负的，而且如果原点是线性规划的一个可行解，那么当把这个平面移离原点时，就得到递增的目标值的点(如果原点不是可行解，或者如果目标函数的某些系数是负值，则直观的图形将变得稍微复杂一点)。如同在二维空间一样，因为可行区域是凸的，取得最优目标值的点的集合必然包含可行区域的一个顶点。类似地，如果有 n 个变量，每个约束定义了 n 维空间中的一个半空间。这些半空间的交集形成的可行区域称作*单纯形*(simplex)。目标函数现在成为一个超平面，而且因为它的凸性，故仍然有一个最优解在单纯形的一个顶点上取得。

单纯形算法以一个线性规划作为输入，输出它的一个最优解。它从单纯形的某个顶点开始，执行一系列的迭代。在每次迭代中，它沿着单纯形的一条边从当前定点移动到一个目标值不小于(通常是大于)当前顶点的相邻顶点。当达到一个局部的最大值，即一个顶点的目标值大于其所有相邻顶点的目标值时，单纯形算法终止。因为可行区域是凸的而且目标函数是线性的，所以局部最优事实上是全局最优的。在 29.4 节中，将使用一个称作“对偶性”的概念来说明单纯形算法输出的解确实是最优的。

虽然几何观察给出了单纯形算法操作过程的一个很好的直观观察，但在 29.3 节讨论单纯形算法的细节时，并不显式地引用它。相反地，我们采用一种代数方法。首先将已知的线性规划写成松弛型，即线性等式的集合。这些线性等式将表示某些变量，称作“基本变量”，而其他变量称作“非基本变量”。从一个顶点移动到另一个顶点伴随着将一个基本变量变为非基本变量，以及将
775 一个非基本变量变为基本变量。这个操作称作一个“主元”，而且从代数的观点来看，它只不过是将线性规划重写成等价的松弛型而已。

上述的二维的例子是特别简单的一个。我们将在本章中讨论几个更详细的例子。这些论题包括识别无解的线性规划，没有有限最优解的线性规划，以及原点不是可行解的线性规划。

线性规划的应用

线性规划有大量的应用。任何一本运筹学的教科书上都充满了线性规划的例子，而且现在大多数商学院中将其作为标准的工具教授给学生。前面所述的选举的场景是一个典型的例子。下列是其他两个线性规划的例子：

- 一家航空公司希望调度他的飞行机组。美国联邦航空委员会提出了许多限制，例如每个机组成员可以工作的连续小时数，以及要求每个机组在每个月内只能在一种机型上工作。这家航空公司想要使用尽量少的机组成员，来调度其所有航班的机组。

- 一家石油公司想要确定在何处钻井采油。在一个特定位置钻井有一个对应的费用，而且根据地理勘探，还有一定数量(桶数)油的回报。这家公司有一个有限的预算来钻新的油井，并且想要在这个预算内让预期找到的油量最大。

线性规划在建模和求解图和组合问题时也很有用，例如出现在本书中的那些问题。我们在 24.4 节已经看到一个用来求解差分约束系统的线性规划的特殊例子。在 29.2 节中，将研究如何将一些图和网络流问题形式化为线性规划。在 35.4 节中，将利用线性规划作为工具，来找出另一个图问题的近似解。

线性规划的算法

本章研究单纯形算法。当它被精心实现时，在实际中通常能够快速地解决一般的线性规划。然而对于某些刻意仔细设计的输入，单纯形算法可能需要指数时间。线性规划的第一个多项式时间算法是椭圆算法，它在实际中运行缓慢。第二类指数时间的算法称为内点法。与单纯形算法(即沿着可行区域的外部移动，并在每次迭代中维护一个为单纯形的顶点的可行解)相比，这些算法在可行区域的内部移动。中间解尽管是可行的，但未必是单纯形的顶点，但最终的解是一个顶点。第一个这样的算法是由 Karmarkar 发现的。对于大型输入，内点法的性能可与单纯形算法相媲美，有时甚至更快。 776

如果在线性规划中加入一个额外的要求，即所有的变量都取整数值，就得到了整数线性规划。练习 34.5-3 要求读者证明，仅找出这个问题的一个可行解就是 NP-难度的；因为还没有已知的多项式时间的算法能解 NP-难度问题，所以没有已知的整数线性规划的多项式时间算法。相反地，一般的线性规划问题可以在多项式时间内求解。

在本章中，如果有一个线性规划其变量为 $x=(x_1, x_2, \cdots, x_n)$，而且希望引用这些变量的一个特定设定，我们将使用记号 $\bar{x}=(\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n)$。

29.1　标准型和松弛型

本节描述两种将在求解线性规划中有用的格式：标准型和松弛型。在标准型中所有的约束都是不等式，而在松弛型中约束都是等式。

标准型

在标准型中，已知 n 个实数 $c_1, c_2, \cdots, c_n$；m 个实数 $b_1, b_2, \cdots, b_m$；以及 mn 个实数 a_{ij}，其中 $i=1, 2, \cdots, m$，而 $j=1, 2, \cdots n$。我们希望找出 n 个实数 $x_1, x_2, \cdots, x_n$ 来

最大化

$$\sum_{j=1}^{n} c_j x_j \tag{29.16}$$

满足约束

$$\sum_{j=1}^{n} a_{ij} x_j \leqslant b_i, \quad i = 1,2,\cdots,m \tag{29.17}$$

$$x_j \geqslant 0, \quad j = 1,2,\cdots,n \tag{29.18}$$

概括为两个变量的线性规划而引入的术语，我们称表达式(29.16)为目标函数，式(29.17)和式(29.18)行中的 $n+m$ 个不等式为约束。式(29.18)行中的 n 个约束称为非负性约束。一个任意的线性规划不需要有非负性约束，但是标准型需要。有时将一个线性规划表示成一个更紧凑的形式会比较方便。如果构造一个 $m \times n$ 矩阵 $A=(a_{ij})$，一个 m 维的向量 $b=(b_i)$，一个 n 维的向量 $c=(c_j)$，以及一个 n 维向量 $x=(x_j)$，那么可以重写式(29.16)至式(29.18)中定义的线性规划为 777

最大化　　$c^{\mathrm{T}}x$　　(29.19)

满足约束

$$Ax \leqslant b \tag{29.20}$$

$$x \geqslant 0 \tag{29.21}$$

在式(29.19)中，$c^{\mathrm{T}}x$ 是两个向量的内积。在式(29.20)中，Ax 是一个矩阵向量乘积，而且在式(29.21)中 $x \geqslant 0$ 表示向量 x 的每个元素都必须是非负的。我们看到可以用一个元组(A, b, c)以标准型来表示一个线性规划，而且我们将采用 A、b 和 c 总是拥有如上所述的维数的约定。

现在来介绍描述线性规划解的术语。其中有些名词在先前两个变量的线性规划的例子中已经使用过了。称满足所有约束的的变量$\bar{x}$的设定为可行解，而不满足至少一个约束的变量$\bar{x}$的设定为不可行解。称一个解x拥有目标值 $c^{\mathrm{T}}\bar{x}$。在所有可行解中其目标值最大的一个可行解$\bar{x}$是一个最优解，且称其目标值 $c^{\mathrm{T}}\bar{x}$为最优目标值。如果一个线性规划没有可行解，则称此线性规划不可行；否则它是可行的。如果一个线性规划有一些可行解但没有有限的最优目标值，则称此线性规划是无界的。练习 29.1-9 要求读者说明即使可行区域无界，线性规划仍然可以有一个有限的最优目标值。

将线性规划转换为标准型

已知一个最小化或最大化的线性函数受若干线性约束，总可以将这个线性规划转换为标准型。一个线性规划可能由于如下 4 个原因之一而不是标准型：

1)目标函数可能是一个最小化，而不是最大化。

2)可能有的变量不具有非负性约束。

3)可能有等式约束，即有一个等号而不是小于等于号。

4)可能有不等式约束，但不是小于等于号，而是一个大于等于号。

当把一个线性规划 L 转换为另一个线性规划 L' 时，我们希望有性质：从 L' 的最优解能得到
778 L 的最优解。为解释这个思想，我们说两个最大化线性规划 L 和 L' 是等价的，如果对于 L 的目标值为 x 的每个可行解$\bar{x}$，存在一个相应的 L' 的目标值为 z 的可行解$\bar{x}'$，而且对于 L' 的目标值为 z 的每个可行解$\bar{x}'$，存在一个相应的 L 的目标值为 z 的可行解$\bar{x}$(这个定义并不意味着可行解之间一对一的对应关系)。一个最小化线性规划 L 和一个最大化线性规划 L' 是等价的，如果对于 L 的目标值为 z 的每个可行解$\bar{x}$，存在一个相应的 L' 的目标值为 $-z$ 的可行解$\bar{x}'$，而且对于 L' 的目标值为 z 的每个可行解$\bar{x}'$，存在一个相应的 L 的目标值为 $-z$ 的可行解$\bar{x}$。

现在来说明如何逐一消除如上所列各项中的每个可能的问题。在消除每一个问题之后，我们将指出这个新的线性规划和原来的是等价的。

为将一个最小化线性规划 L 转换成一个等价的最大化线性规划 L'，简单地对目标函数中的系数取负即可。因为 L 和 L' 有相同的可行解集合，而且对任意的可行解，L 的目标值是 L' 的目标值的负值。所以这两个线性规划是等价的。例如，如果有线性规划

最小化　　$-2x_1 + 3x_2$

满足约束

$$\begin{aligned} x_1 + x_2 &= 7 \\ x_1 - 2x_2 &\leqslant 4 \\ x_1 &\geqslant 0 \end{aligned}$$

将目标函数的系数取负，得

最大化　　$2x_1 - 3x_2$

满足约束

$$\begin{aligned} x_1 &+ x_2 = 7 \\ x_1 &- 2x_2 \leqslant 4 \\ x_1 &\geqslant 0 \end{aligned}$$

接下来说明如何将某些变量不具有非负性约束的线性规划，转换成每个变量都有非负性约束的线性规划。假设某个变量 x_j 不具有非负性约束。则把 x_j 的每次出现都以 $x_j'-x_j''$ 来替代，并增加非负性约束 $x_j'\geqslant 0$ 和 $x_j''\geqslant 0$。因此，如果目标函数有一个项为 c_jx_j，则将其替代为 $c_jx_j'-c_jx_j''$，而且如果约束 i 有一个项为 $a_{ij}x_j$，则将其替代为 $a_{ij}x_j'-a_{ij}x_j''$。新的线性规划的任意可行解 $\bar{x}$ 对应于原来的线性规划的一个可行解，其中 $\bar{x}_j=\bar{x}_j'-\bar{x}_j''$，而且拥有相同的目标值，因此这两个线性规划是等价的。把这个转换方案应用到每一个不具有非负性约束的变量上，得到一个等价的线性规划，其中所有的变量都具有非负性约束。 779

继续这个例子，我们想确认每个变量都具有一个相应的非负性约束。变量 x_1 具有非负性约束，但变量 x_2 不具有。因此，用两个变量 x_2' 和 x_2'' 来替换 x_2，且修改线性规划以得到

最大化　$2x_1-3x_2'+3x_2''$

满足约束

$$\begin{aligned} x_1 &+ x_2' - x_2'' = 7 \\ x_1 &- 2x_2' + 2x_2'' \leqslant 4 \\ x_1, x_2', x_2'' &\geqslant 0 \end{aligned} \tag{29.22}$$

接下来，我们将等式约束转换为不等式约束。假设一个线性规划有一个等式约束 $f(x_1, x_2, \cdots, x_n)=b$。因为当且仅当 $x\geqslant y$ 和 $x\leqslant y$ 时 $x=y$，所以可以将这个等式约束用一对不等式约束 $f(x_1, x_2, \cdots, x_n)\leqslant b$ 和 $f(x_1, x_2, \cdots, x_n)\geqslant b$ 来代替。在每个等式约束上重复这个替换，就得到全是不等式约束的线性规划。

最后，可以通过将大于等于的约束乘以 -1 来把它们转换成小于等于的约束。也就是说，任何形式为

$$\sum_{j=1}^{n} a_{ij}x_j \geqslant b_i$$

的不等式等价于

$$\sum_{j=1}^{n} -a_{ij}x_j \leqslant -b_i$$

因此，通过用 $-a_{ij}$ 来替换每个系数 a_{ij}，用 $-b_i$ 来替换每个值 b_i，我们得到一个等价的小于等于的约束。

结束我们的例子，通过用两个不等式取代约束式(29.22)中的等式，得到

最大化　$2x_1-3x_2'+3x_2''$

满足约束

$$\begin{aligned} x_1 &+ x_2' - x_2'' \leqslant 7 \\ x_1 &+ x_2' - x_2'' \geqslant 7 \\ x_1 &- 2x_2' + 2x_2'' \leqslant 4 \\ x_1, x_2', x_2'' &\geqslant 0 \end{aligned} \tag{29.23}$$

780

最后，对约束式(29.23)取负。为了变量的名称保持一致，将名称 x_2' 用 x_2 代替，将名称 x_2'' 用 x_3 代替，得到标准型

最大化　$2x_1-3x_2+3x_3$　(29.24)

满足约束

$$x_1 + x_2 - x_3 \leqslant 7 \tag{29.25}$$

$$-x_1 - x_2 + x_3 \leqslant -7 \tag{29.26}$$

$$x_1 - 2x_2 + 2x_3 \leqslant 4 \tag{29.27}$$

$$x_1, x_3, x_3 \geqslant 0 \tag{29.28}$$

将线性规划转换为松弛型

为了利用单纯形算法高效地求解线性规划，我们更喜欢将它表示成其中某些约束是等式约束的形式。更准确地说，我们将把它转换成非负约束都只是不等式约束，而余下的约束都是等式约束的形式。令

$$\sum_{j=1}^{n} a_{ij}x_j \leqslant b_i \tag{29.29}$$

是一个不等式约束。引入一个新的变量 s，并重写不等式(29.29)为两个约束

$$s = b_i - \sum_{j=1}^{n} a_{ij}x_i \tag{29.30}$$

$$s \geqslant 0 \tag{29.31}$$

称 s 是一个松弛变量(slack variable)，因为它度量了等式(29.29)左边和右边之间的松弛或差别。因为当且仅当等式(29.30)和不等式(29.31)都为真时不等式(29.29)为真，所以可以对线性规划的每个不等式约束应用这个转换，得到一个等价的线性规划，其中只有不等式是非负约束。当从标准型转换到松弛型时，我们将使用 x_{n+i}(而不是 s)来表示与第 i 个不等式关联的松弛变量。因此第 i 个约束是

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij}x_i \tag{29.32}$$

以及非负约束 $x_{n+i} \geqslant 0$。

781　将这个转换应用到标准型的线性规划的每个约束中，得到一个不同形式的线性规划。例如，对式(29.24)至式(29.28)描述的线性规划，引入松弛变量 x_4、x_5 和 x_6，得到

最大化

$$2x_1 - 3x_2 + 3x_3 \tag{29.33}$$

满足约束

$$x_4 = 7 - x_1 - x_2 - x_3 \tag{29.34}$$

$$x_5 = -7 + x_1 + x_2 - x_3 \tag{29.35}$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \tag{29.36}$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geqslant 0 \tag{29.37}$$

在这个线性规划中，除了非负约束外的所有约束都是等式，而且每个变量都具有非负约束。把每个等式约束写成其中一个变量在等式左边，而其余变量在等式右边的形式。而且每个等式右边都有相同的变量集合，且这些变量同时也是出现在目标函数中的仅有的变量。等式左边的变量称为基本变量，而等式右边的变量称为非基本变量。

对于满足这些条件的线性规划，有时会省略词语“最大化”(maximize)和“满足约束”，且省略明显的非负约束。我们也会使用变量 z 来表示目标函数的值。称这个结果形式为松弛型。如果重写式(29.33)至式(29.37)中的线性规划为松弛型，得到

$$z = 2x_1 - 3x_2 + 3x_3 \tag{29.38}$$

$$x_4 = 7 - x_1 - x_2 + x_3 \tag{29.39}$$

$$x_5 = -7 + x_1 + x_2 - x_3 \tag{29.40}$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \tag{29.41}$$

如同标准型一样，使用更简明的记号来描述一个松弛型会比较方便。如即将在 29.3 节看到的那样，在单纯形算法执行时，基本变量和非基本变量集合将发生变化。用 N 来表示非基本变量的下标的集合，用 B 来表示基本变量下标的集合。总是有 $|N| = n$，$|B| = m$，以及 $N \cup B = \{1, 2, \cdots, n+m\}$。等式将被 B 的元素索引，等式右边的变量将被 N 的元素索引。和标准型一样，用 b_i、c_j 和 a_{ij} 来表示常项和系数。我们还使用 v 来表示目标函数的自由常项。因此可以简明地定义一个松弛型，即用一个元组 (N, B, A, b, c, v) 来表示松弛型

$$z = v + \sum_{j \in N} c_j x_j \tag{29.42}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for} \quad i \in B \tag{29.43}$$

782

其中所有的变量 x 受非负约束。这是因为在式(29.43)中减去和式 $\sum_{j \in N} a_{ij} x_j$，$a_{ij}$ 的值，实际上是它们“出现”在松弛型中时系数的负值。

例如，在下面的松弛型中，

$$\begin{aligned}
z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}
\end{aligned}$$

有 $B = \{1, 2, 4\}$，$N = \{3, 5, 6\}$)，

$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix} \quad b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix}$$

$c = (c_3\ c_5\ c_6)^{\mathrm{T}} = (-1/6 - 1/6 - 2/3)^{\mathrm{T}}$，以及 $v = 28$。注意 A、b 和 c 的下标值不必要是连续整数的集合；它们依赖于索引值集合 B 和 N。作为一个例子，A 的元素是它们出现在松弛型中的系数的负值，观察到 x_1 的等式包含项 $x_3/6$，而系数 a_{13} 实际上是 $-1/6$，而不是 $+1/6$。

练习

29.1-1　如果将式(29.24)～式(29.28)中的线性规划表示成式(29.19)～式(29.21)中的简洁记号形式，n、m、A、b 和 c 分别是什么？

29.1-2　给出式(29.24)至式(29.28)的线性规划的 3 个可行解。每个解的目标值是多少？

29.1-3　在式(29.38)至式(29.41)的松弛型中，N、B、A、b、c 和 v 分别是什么？ 783

29.1-4　将下列线性规划转换为标准型：

最小化　$2x_1 + 7x_2$

满足约束

$$\begin{aligned}
x_1 &= 7 \\
3x_1 + x_2 &\geqslant 24 \\
x_2 &\geqslant 0 \\
x_3 &\leqslant 0
\end{aligned}$$

29.1-5　将下列线性规划转换为松弛型：

最大化

$$2x_1 - 6x_3$$

满足约束

$$\begin{aligned} x_1 + x_2 - x_3 &\leqslant 7 \\ 3x_1 - x_2 &\geqslant 8 \\ -x_1 + 2x_2 + 3x_3 &\geqslant 0 \\ x_1, x_2, x_3 &\geqslant 0 \end{aligned}$$

基本变量和非基本变量分别是什么？

29.1-6　说明下列线性规划是不可行的：

最大化

$$3x_1 - 2x_2$$

满足约束

$$\begin{aligned} x_1 + x_2 &\leqslant 2 \\ -2x_1 - 2x_2 &\leqslant -10 \\ x_1, x_2 &\geqslant 0 \end{aligned}$$

29.1-7　说明下列线性规划是无界的：

最大化

$$x_1 - x_2$$

满足约束

$$\begin{aligned} -2x_1 + x_2 &\leqslant -1 \\ -x_1 - 2x_2 &\leqslant -2 \\ x_1, x_2 &\geqslant 0 \end{aligned}$$

784

29.1-8　假设有一个 n 个变量和 m 个约束的线性规划，且假设将其转换成标准型。请给出所得线性规划中变量和约束个数的一个上界。

29.1-9　请给出一个线性规划的例子，其中可行区域是无界的，但最优解的值是有界的。

29.2　将问题表达为线性规划

虽然本章的重点在单纯形算法上，但是识别出一个问题是否可以形式化为一个线性规划也是很重要的。一旦一个问题被形式化成一个多项式规模的线性规划，它可以用椭圆法或内点法在多项式时间内解决。一些线性规划的软件包可以高效地解决问题，因此，一旦问题被表示成一个线性规划后，实际上它可以用这种软件包来求解。

下面来看一些线性规划问题的实际例子。首先从前面已经研究过的两个问题开始：单源最短路径问题（参见第 24 章）和最大流问题（参见第 26 章）。然后，再介绍最小费用流问题。最小费用流问题有一个不是基于线性规划的多项式时间算法，但我们不讨论它。最后，还要介绍多商品流问题，它的唯一已知的多项式时间算法是基于线性规划的。

最短路径

在第 24 章叙述的单源最短路径问题可以形式化为一个线性规划。在这一节中，我们着重介绍单对最短路径问题的形式化，而把到更一般的单源最短路径问题的扩展留作练习 29.2-3。

在单对最短路径问题中，已知有一个带权有向图 $G=(V, E)$，加权函数 $w: E \to \mathbf{R}$ 将边映射到实数值的权值、一个源顶点 s、一个目的顶点 t。我们希望计算从 s 到 t 的一条最短路径的权值 $d[t]$。为把这个问题表示成线性规划，需要确定变量和约束的集合来定义何时有从 s 到 t 的一条
785 最短路径。幸运的是，Bellman-Ford 算法做的就是这个。当 Bellman-Ford 算法终止时，对每个顶

点 v，它计算了一个值 $d[v]$，使得对每条边 $(u, v)\in E$，有 $d[v]\leqslant d[u]+w(u, v)$。源顶点初始得到一个值 $d[s]=0$，以后也不会改变。因此我们得到如下的线性规划，来计算从 s 到 t 的最短路径的权值：

最小化

$$d[t] \tag{29.44}$$

满足约束

$$d[v]\leqslant d[u]+w(u,v) \quad \text{对每条边}(u,v)\in E \tag{29.45}$$

$$d[s]=0 \tag{29.46}$$

在这个线性规划中，有 $|V|$ 个变量 $d[v]$，每个顶点 $v\in V$ 各有一个。有 $|E|+1$ 个约束，每条边各有一个再加上源顶点总是有值 0 的额外约束。

最大流

最大流问题也可以表示成线性规划。回顾我们已知一个有向图 $G=(V, E)$，其中每条边 $(u, v)\in E$ 有一个非负的容量 $c(u, v)\geqslant 0$，以及两个特别的顶点：源 s 和汇 t。如同在 26.1 节中定义的一样，流是一个实数值的函数 $f: V\times V\rightarrow \mathbf{R}$，它满足三个性质：容量限制，斜对称性、流守恒性。最大流是满足这些约束和最大化流量值的流，其中流量值是从源流出的总流量。因此，流满足线性约束，且流的值是一个线性函数。我们还假设了如果 $(u, v)\notin E$，则 $c(u, v)=0$，可以将最大流问题表示为线性规划：

最大化

$$\sum_{v\in V} f(s,v) \tag{29.47}$$

满足约束

$$f(u,v)\leqslant c(u,v) \quad \text{对每个 } u,v\in V \tag{29.48}$$

$$f(u,v)=-f(v,u) \quad \text{对每个 } u,v\in V \tag{29.49}$$

$$\sum_{v\in V} f(u,v)=0 \quad \text{对每个 } u\in V-\{s,t\} \tag{29.50}$$

这个线性规划有 $|V|^2$ 个变量，对应于每一对顶点之间的流，且有 $2|V|^2+|V|-2$ 个约束。

通常求解一个较小规模的线性规划更加有效。为了方便记号表示，式(29.47)～式(29.50)中的线性规划有一个流和每对 $(u, v)\notin E$ 的顶点 u、v 的容量为 0。把这个线性规划重写成有 $O(V+E)$ 个约束的形式会更有效。练习 29.2-5 将要求读者完成这个任务。 786

最小费用流

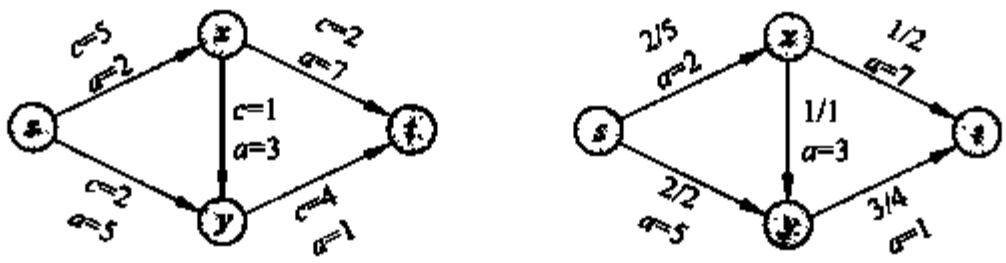
在这一节中，我们已经使用线性规划来求解已知有效算法的问题。事实上，为一个问题特别设计的一个有效的算法，譬如用于单源最短路径问题的 Dijkstra 算法，或者最大流的 push-relabel 方法，经常在理论和实践中都比线性规划更加有效。

线性规划的真正力量来自其求解新问题的能力。回顾在本章开始政治家所面对的问题。得到足够数量的选票而不用花费太多钱的问题，任何一个本书已经研究过的算法都无法求解，但是可以被线性规划求解。教科书中充满了线性规划可以解决的真实世界中的问题。线性规划同时也对各种我们没有已知的有效算法的问题特别有用。

例如，考虑最大流问题的如下一般化。假设每条边 (u, v) 除了有一个容量 $c(u, v)$ 外，还有一个实数值的费用 $a(u, v)$。如果通过边 (u, v) 传送 $f(u, v)$ 个单位的流，那么发生了一个费用 $a(u, v)f(u, v)$。同时还给定了一个流目标 d。我们希望从 s 发送 d 个单位的流到 t，使得流上发生的总费用 $\sum_{(u,v)\in E} a(u, v)f(u, v)$ 最小。这个问题被称为最小费用流问题。

图 29-3a 显示了最小费用流的一个例子。我们希望从 s 发送 4 个单位的流到 t，同时产生最小的总费用。任何一个特定的合法流，亦即一个满足约束(式(29.48)～式(29.50))的函数 f，产

生的总费用为 $\sum_{(u,v)\in E} a(u,v)f(u,v)$。我们希望找到一个特殊的 4 个单位的流，使其最小化这个费用。图 29-3b 给出了一个最优解，其总费用 $\sum_{(u,v)\in E} a(u,v)f(u,v) = (2\cdot 2)+(5\cdot 2)+(3\cdot 1)+(7\cdot 1)+(1\cdot 3) = 27$。



a)　　b)

图 29-3　a)一个最小费用流问题的例子。用 c 表示容量、a 表示花费。顶点 s 是源，顶点 t 是汇(sink)，而且希望从 s 发送 4 个单位的流到 t。b)这个最小费用流的一个解，其中从 s 发送 4 个单位的流到 t。对于每条边，流和容量写成流/容量的形式

787

有特别为最小费用流设计的多项式时间算法，但它们超出了本书的范畴。然而，可以将最小费用流问题表示成一个线性规划。这个线性规划看去来和最大流问题的那个相类似，它有流量为准确的 d 个单位的额外约束，以及最小化费用的新的目标函数：

最小化

$$\sum_{(u,v)\in E} a(u,v)f(u,v) \tag{29.51}$$

满足约束

$$f(u,v) \leqslant c(u,v) \qquad \text{对每个 } u,v \in V \tag{29.52}$$

$$f(u,v) = -f(v,u) \qquad \text{对每个 } u,v \in V \tag{29.53}$$

$$\sum_{v\in V} f(u,v) = 0 \qquad \text{对每个 } u \in V-\{s,t\} \tag{29.54}$$

$$\sum_{v\in V} f(s,v) = d \tag{29.55}$$

多商品流

举最后一个例子，考虑另外一个流问题。假设 26.1 节的 Lucky Puck 公司决定多样化它的产品线，并且不只运送冰上曲棍球的圆盘，同时还运送球棒和头盔。每个装备都是在它自己的工厂内制造，有它自己的仓库，而且每天必须从工厂运送到仓库。球棒是在 Vancouver 制造，要运送到 Saskatoon，而头盔是在 Edmonton 制造，要运送到 Regina。但是，运输网络的容量并没有改变，不同的项目或者是商品必须共享同一个网络。

这个例子是多商品流问题的一个实例。在这个问题中，仍然给定一个有向图 $G=(V, E)$，其中每条边 $(u, v)\in E$ 有一个非负的容量 $c(u, v)\geqslant 0$。如同在最大流问题中一样，我们隐含地假设对于 $(u, v)\notin E$ 有 $c(u, v)=0$。另外，还已知 k 种不同的商品，K_1，K_2，…，K_k，其中商品 i 用元组 $K_i=(s_i, t_i, d_i)$ 来指定。这里，s_i 是商品 i 的源；t_i 是商品 i 的汇；d_i 是需求，即商品 i 从 s_i 到 t_i 所需的流量值。将商品 i 的流用 f_i 表示(因此 $f_i(u, v)$ 是商品 i 从顶点 u 到顶点 v 的流)，定义为一个满足流量守恒、斜对称性和容量约束的实数值函数。现在定义汇聚流 $f(u, v)$ 为各种商品流的总和，因此 $f(u, v)=\sum_{i=1}^{k} f_i(u,v)$。在边 (u,v) 上的汇聚流不能超过边 (u,v) 的容

788 量。这个约束包含了每个商品的容量约束。以这样的方式来描述这个问题，没有东西要最小化；只需要确定是否能找到这样的一个流。因此，用一个“空”的目标函数来写这个线性规划：

最小化

0

满足约束

$$\sum_{i=1}^{k} f_i(u,v) \leqslant c(u,v) \quad \text{对每个 } u,v \in V$$

$$f_i(u,v) = -f_i(v,u) \quad \text{对每个 } i = 1,2,\cdots,k\text{;并且对每个 } u,v \in V\text{,}$$

$$\sum_{v \in V} f_i(u,v) = 0 \quad \text{对每个 } i = 1,2,\cdots,k\text{;并且对每个 } u \in V - \{s_i,t_i\}$$

$$\sum_{v \in V} f_i(s,v) = d_i \quad \text{对每个 } i = 1,2,\cdots,k$$

这个问题唯一已知的多项式时间算法是将它表示成一个线性规划，然后用一个多项式时间线性规划算法来解决。

练习

29.2-1　将单对最短路径线性规划从式(29.44)～式(29.46)转换成标准型。

29.2-2　详细写出在图 24-2a 中，和寻找从结点 s 到结点 y 的最短路径相对应的线性规划。

29.2-3　在单源最短路径问题中，我们想要找出从源顶点 s 到所有顶点 $v \in V$ 的最短路径权值。给定一个图 G，写出一个线性规划，它的解具有性质：对每个顶点 $v \in V$，$d[v]$是从 s 到 v 的最短路径的权值。

29.2-4　详细写出在图 26-2a 中寻找最大流所对应的线性规划。

29.2-5　重写最大流式(29.47)～式(29.50)的线性规划，使它只使用 $O(V+E)$个约束。

789

29.2-6　写出一个线性规划，已知一个二分图 G，求解最大二分匹配问题。

29.2-7　在最小费用多商品流问题中，给定有向图 $G=(V,E)$，其中每条边 $(u,v) \in E$ 有一个非负的容量 $c(u,v) \geqslant 0$，以及一个费用 $a(u,v)$。如同在多商品流问题中一样，已知 k 种不同的商品，$K_1,K_2,\cdots,K_k$，其中商品 i 用元组 $K_i=(s_i,t_i,d_i)$ 来指定。如同在多商品流问题中一样，为商品 i 定义流 f_i，在边 (u,v) 上定义汇聚流 $f(u,v)$。一个可行流是在每条边 (u,v) 上的汇聚流不超过边 (u,v) 的容量。一个流的费用是 $\sum_{u,v \in V} a(u,v) f(u,v)$，目标是寻找最小费用的可行流。将这个问题表示为一个线性规划。

29.3　单纯形算法

单纯形算法是求解线性规划的古典方法。和本书其他算法相反的是，它的执行时间在最坏的情况下并不是多项式。然而，它确实使我们加深了对线性规划的理解，而且在实际中通常相当快速。

除了本章稍早描述的几何解释外，单纯形算法与 28.3 节讨论的高斯消元法有些类似的地方。高斯消元法从解未知的一个线性等式系统开始。在每次迭代中，将这个系统重写为具有一些额外结构的等价形式。经过一定次数的迭代后，我们已经重写这个系统，使得它的解很容易得到。单纯形算法以一个相似的方式进行，而且可以将其看作是在不等式上的高斯消元法。

现在我们描述在单纯形算法迭代背后的主要思想。和每次迭代关联的“基本解”可以很容易地从线性规划的松弛型中得到：将每个非基本变量设为 0，并从等式约束中计算基本变量的值。一个基本解总是对应于单纯形的一个顶点。在代数上，一次迭代将一个松弛型转换成一个等价的松弛型。相应的基本可行解的目标值不小于前一次迭代中的目标值(通常是大于)。为了达到目标值的这种递增，我们选择一个非基本变量，使得如果是要从 0 开始增加这个变量的值，则目标值也会增加。可以在变量上增加的数值受其他约束限制。特别是，要增加它直到某个基本变量变

790

为0为止。然后重写松弛型，将这个基本变量和所选的非基本变量的角色互换。虽然我们使用了变量的一个特殊设定来指导这个算法，而且还将在证明中使用它，但这个算法并没有明显地维护这个解。它只是重写线性规划直到最优解变得"明显"为止。

单纯形算法的一个例子

我们从一个扩展的例子开始。考虑下列标准型的线性规划：

$$\text{最大化} \quad 3x_1 + x_2 + 2x_3 \tag{29.56}$$

满足约束

$$x_1 + x_2 + 3x_3 \leqslant 30 \tag{29.57}$$

$$2x_1 + 2x_2 + 5x_3 \leqslant 24 \tag{29.58}$$

$$4x_1 + x_2 + 2x_3 \leqslant 36 \tag{29.59}$$

$$x_1, x_2, x_3 \geqslant 0 \tag{29.60}$$

为了利用单纯形算法，必须将线性规划转换成松弛型；我们在29.1节看到了如何做这个转换。除了是一个代数操作外，松弛也是一个有用的算法概念。回顾在29.1节中每个变量有一个对应的非负约束；我们称一个等式约束对于它的非基本变量的一个特殊设定是紧(tight)的(如果它们导致这个约束的基本变量变为0的话)。类似地，导致一个基本变量变为负值的非基本变量的设定违反了这个约束。所以，松弛变量显式地维护每个约束距离紧的程度，来让它们帮助我们确定可以增加多少非基本变量的值而不违反任何约束。

将松弛变量 x_4、x_5 和 x_6 分别关联于不等式(29.57)～不等式(29.59)，并且将线性规划写成松弛型，得到

$$z = 3x_1 + x_2 + 2x_3 \tag{29.61}$$

$$x_4 = 30 - x_1 - x_2 - 3x_3 \tag{29.62}$$

$$x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \tag{29.63}$$

$$x_6 = 36 - 4x_1 - x_2 - 2x_3 \tag{29.64}$$

约束系统式(29.62)～式(29.64)有3个等式和6个变量。变量 x_1、x_2、x_3 的任意设定定义了 x_4、x_5 和 x_6 的值；因此这个等式系统有无限个数的解。如果所有的 $x_1, x_2, \cdots, x_6$ 都是非负
791 的，则这个解是可行的，而且也有无限个数的可行解。像这样的一个系统的无限个数的可能解在稍后的证明中会有用处。我们将集中注意力于基本解：把等式右边的所有(非基本)变量设为0，然后计算等式左边(基本)变量的值。在这个例子中，基本解为 $(\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_6) = (0, 0, 0, 30, 24, 36)$，其目标值为 $z = (3 \cdot 0) + (1 \cdot 0) + (2 \cdot 0) = 0$。注意到这个基本解对每个 $i \in B$ 设定 $\bar{x}_i = b_i$。单纯形算法的每次迭代会重写等式集合和目标函数，来将一个不同的变量集合放在右边。因此，重写过的问题会有一个不同的基本解。我们强调重写不会改变基本的线性规划；在一次迭代中的问题与前一次迭代中的问题有着相同的可行解集合。然而，问题确实会与前一次迭代的问题有着不同的基本解。

如果一个基本解也是可行的，则称其为基本可行解(basic feasible solution)。在单纯形算法的执行过程中，基本解几乎总是基本可行解。然而我们将在29.5节看到，在单纯形算法的前一次迭代中，基本解可能不是可行的。

在每次迭代中，目标是重新表达线性规划，来让基本解有一个更大的目标值。选择一个在目标函数中系数为正值的非基本变量 x_e，而且尽可能增加 x_e 的数值而不违反任何约束。变量 x_e 成为基本变量，某个其他变量 x_l 成为非基本变量。其他基本变量和目标函数的值都可能改变。

继续这个例子，让我们来考虑增加 x_1 的值。当增加 x_1 时，x_4、x_5、x_6 的值随之减小。因

为对每个变量有一个非负的约束，所以不能允许它们之中的任何一个变成负值。如果 x_1 增加到 30 以上，则 x_4 成为负值，而当 x_1 分别增加到 12 和 9 时，x_5 和 x_6 也成为负值。第三个约束(式(29.64))是最紧的约束，它限制了可以增加 x_1 多少。因此我们将互换 x_1 和 x_6 的角色。解 x_1 的方程(式(29.64))，得到

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \tag{29.65}$$

为了重写右边带有 x_6 的其他等式，用等式(29.65)来取代 x_1。在等式(29.62)上做这个替换，得到

$$\begin{aligned} x_4 &= 30 - x_1 - x_2 - 3x_3 \\ &= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - x_2 - 3x_3 \\ &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \end{aligned} \tag{29.66}$$

同样地，可以联立等式(29.65)和约束式(29.63)以及目标函数式(29.61)，以重写线性规划为如下的形式：

792

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \tag{29.67}$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \tag{29.68}$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \tag{29.69}$$

$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} \tag{29.70}$$

称这个操作为主元(pivot)。正如上面所展示的，一个主元选取一个非基本变量 x_e(称为换入变量)和一个基本变量 x_l(称为换出变量)，然后交换二者的角色。

式(29.67)至式(29.70)所描述的线性规划等价于等式(29.61)至等式(29.64)所描述的线性规划。我们在单纯形算法中所执行的操作是重写等式，来让变量在等式的左边与右边之间移动，并且用一个等式来替换为另外一个等式。第一个操作只是建立一个等价的问题，而第二个操作通过简单的线性代数也建立了一个等价的问题。

为了展示这个等价性，观察到初始的基本解(0，0，0，30，24，36)满足新的等式即式(29.68)至式(29.70)且拥有目标值 27＋(1/4)·0＋(1/2)·0－(3/4)·36＝0。和新的线性规划相关联的基本解将非基本变量的值设为 0，即(9，0，0，21，6，0)，且目标值 z 为 27。简单的算术证明这个解也满足等式(29.61)～等式(29.64)，并且当插入目标函数式(29.61)时，有目标值(3·9)＋(1·0)＋(2·0)＝27。

继续这个例子，我们希望找出一个想增加其值的新的变量。我们不想增加 x_6，因为当它的值增加时，目标值将减小。可以尝试增加 x_2 或 x_3；我们选择 x_3。能增加 x_3 到多少而不会违反任何约束？约束(式(29.69))将其限制为 18，约束(式(29.69))将其限制为 42/5，而约束(式(29.70))将其限制为 3/2。第三个约束又是最紧的，且我们将重写第三个约束，使得 x_3 在等式的左边而 x_5 在等式的右边。然后，将这个新的等式替换入等式(29.67)～等式(29.69)中，得到新的、但是等价的系统

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \tag{29.71}$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \tag{29.72}$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \tag{29.73}$$

793

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} \tag{29.74}$$

这个系统相关联的基本解为(33/4，0，3/2，69/4，0，0)，目标值为 111/4。现在增加目标值的唯一途径是增加 x_2。这三个约束分别给出 132、4、∞的上界(来自约束式(29.74))的上界∞，是因为当增加 x_2 时，基本变量 x_4 的值也增加。因此这个约束对 x_2 能增加多少没有限制)。将 x_2 增加到 4，它变成非基本的。然后为 x_2 解方程(式(29.73))，并替换入其他等式，得到

$$z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \tag{29.75}$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \tag{29.76}$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \tag{29.77}$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \tag{29.78}$$

此时，目标函数中所有的系数都是负的。正如我们将在本章稍后看到的，这种情况只发生在当我们已经重写线性规划使得基本解就是一个最优解的时候。因此，对于这个问题，解(8，4，0，18，0，0)有目标值 28 是最优的。现在可以回到在式(29.56)至式(29.60)中给出的初始的线性规划。在这个初始的线性规划中仅有的变量是 x_1、x_2 和 x_3，因此解是 $x_1=8$，$x_2=4$，$x_3=0$，且目标值为(3·8)+(1·4)+(2·0)=28。注意在最终解中松弛变量的值度量了每个不等式的松弛是多少。松弛变量 x_4 等于 18，且在不等式(29.57)中左边的值为 8+4+0=12，比右边的值 30 小了 18。松弛变量 x_5 和 x_6 都是 0，而且事实上在不等式(29.58)和不等式(29.59)中左右两边是相等的。还注意到即使在初始的松弛型中系数是整数，在其他线性规划中的系数却不一定是整数，而且中间解也不一定是可行的。进一步讲，线性规划的最终解不需要是整数；这个例子有一个整数解纯属巧合。

主元

现在我们来制定主元的过程。过程 PIVOT 以一个松弛型为输入，给定元组(N, B, A, b, c, v)，换出变量 x_l 的下标 l，以及换入变量 x_e 的下标 e。它返回描述新松弛型的元组$(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$。(回顾前面可知矩阵 A 和$\hat{A}$的元素实际上都是出现在松弛型中的系数的负值。)

794

```
PIVOT(N, B, A, b, c, v, l, e)
1  ▷Compute the coefficients of the equation for new basic variable x_e.
2  b̂_e ← b_l / a_le
3  for each j ∈ N − {e}
4      do â_ej ← a_lj / a_le
5  â_el ← 1 / a_le
6  ▷Compute the coefficients of the remaining constraints.
7  for each i ∈ B − {l}
8      do b̂_i ← b_i − a_ie b̂_e
```

```
 9        for each j∈N−{e}
10            do â_ij ← a_ij − a_ie â_ej
11        â_il ← −a_ie â_ll
12    ▷ Compute the objective function.
13    v̂ ← v + c_e b̂_e
14    for each j∈N−{e}
15        do ĉ_j ← c_j − c_e â_ej
16    ĉ_l ← −c_e â_ll
17    ▷ Compute new sets of basic and nonbasic variables.
18    N̂ = N−{e}∪{l}
19    B̂ = B−{l}∪{e}
20    return(N̂, B̂, Â, b̂, ĉ, v̂)
```

PIVOT 的执行过程如下。第 2～5 行通过重写有 x_l 在左边的等式来让 x_e 在等式的左边，以计算 x_e 的新的等式中的系数。第 7～11 行用这个新等式的右边替换 x_e 的每次出现，来更新余下的等式。第 13～16 行对目标函数施行相同的替换，第 18～19 行更新非基本变量和基本变量的集合。第 20 行返回新的松弛型。当 $a_{le}=0$ 时，PIVOT 将产生除 0 的错误，但如我们将在定理 29.2 和定理 29.12 中看到的那样，PIVOT 只有当 $a_{le}\neq 0$ 时才会被调用。

现在我们总结 PIVOT 对基本解中变量的值的影响。

引理 29.1　考虑当 $a_{le}\neq 0$ 时对 PIVOT(N, B, A, b, c, v, l, e)的调用。令调用返回的值为$(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$，令$\bar{x}$表示调用结束后的基本解。则

1)对每个 $j\in\hat{N}$，$\bar{x}_j=0$

2)$\bar{x}_e=b_l/a_{le}$。

3)对每个 $i\in\hat{B}-\{e\}$，$\bar{x}_i=b_i-a_{ie}\hat{b}_e$。

795

证明：第一个陈述成立是因为基本解总是将所有的非基本变量设为 0。当在一个约束中把每个非基本变量设为 0 时

$$\bar{x}_i=\hat{b}_i-\sum_{j\in N}\hat{a}_{ij}\bar{x}_j$$

对每个 $i\in\hat{B}$ 有$\bar{x}_i=\hat{b}_i$。因为 $e\in\hat{B}$，根据 PIVOT 的第 2 行，有

$$\bar{x}_e=\hat{b}_e=b_l/a_{le}$$

这就证明了第二个陈述。同样地，对每个 $i\in\hat{B}-\{e\}$，利用第 8 行，有

$$\bar{x}_i=\hat{b}_i=b_i-a_{ie}\hat{b}_e$$

这证明了第三个陈述。■

正式的单纯形算法

现在就可以对单纯形算法进行形式化了，先前已经用例子说明了这一方法。那个例子是特别好的一个，通过那个例子，其实还可以进一步讨论另外的几个问题：

- 如何确定一个线性规划是不是可行的？
- 如果线性规划是可行的，但初始基本解是不可行的，该做什么？

- 应如何确定一个线性规划是无界的？
- 应如何选择换入变量和换出变量？

在 29.5 节中，我们将说明如何确定一个问题是否是可行的，且如果可行，如何找出一个初始基本解可行的松弛型。因此假设有一个程序 INITIALIZE-SIMPLEX(A, b, c)，输入为一个标准型的线性规划，即一个 $m \times n$ 矩阵 $A=(a_{ij})$，一个 m 维的向量 $b=(b_i)$，一个 n 维的向量 $c=(c_j)$。如果问题是不可行的，它返回一个消息说明线性规划不可行，然后终止。否则，它返回一个初始基本解可行的松弛型。

过程 SIMPLEX 以一个标准型的线性规划作为输入，如前所述。它返回一个 n 维向量
796 $\bar{x}=(\bar{x}_j)$，即在式(29.19)～式(29.21)中描述的线性规划的一个最优解。

```
SIMPLEX(A, b, c)
 1  (N, B, A, b, c, v) ← INITIALIZE-SIMPLEX(A, b, c)
 2  while some index j∈N has c_j>0
 3      do choose an index e∈N for which c_e>0
 4        for each index i∈B
 5            do if a_ie>0
 6                then Δ_i ← b_i/a_ie
 7                else Δ_i ← ∞
 8        choose an index l∈B that minimizes Δ_i
 9        if Δ_l=∞
10          then return "unbounded"
11          else (N, B, A, b, c, v) ← PIVOT(N, B, A, b, c, v, l, e)
12  for i ← 1 to n
13      do if i∈B
14            then x̄_i ← b_i
15            else x̄_i ← 0
16  return (x̄_1, x̄_2, ..., x̄_n)
```

过程 SIMPLEX 的执行过程如下。在第 1 行，它调用上面所述的过程 INITIALIZE-SIMPLEX(A, b, c)，要么确定这个线性规划是不可行的，要么返回一个初始基本解可行的松弛型。算法的主要部分在第 2～11 行的 while 循环中给出。如果目标函数中所有的系数都是负值，则 while 循环终止。否则在第 3 行选择一个在目标函数中系数为正值的变量 x_e 作为换入变量。尽管可以自由地选择任意一个这样的变量作为换入变量，但还是假定使用某个预先制定的确定性的规则。下一步，在第 4～8 行，检查每个约束，然后挑选出最严格限制 x_e 能增加的数量而又不违反任何的非负约束的那个约束；和这个约束关联的基本变量是 x_l。我们可以再自由选择这些变量中的任意一个作为换出变量，但还是假设使用某个预先制定的确定性的规则。如果没有一个约束限制换入变量能够增加的量，则算法在第 10 行返回"无界"。否则，第 11 行通过调用上面描述的 PIVOT(N, B, A, b, c, v, l, e)子过程来互换换入变量与换出变量的角色。第 12～15 行通过把所有的非基本变量设为 0 以及把每个基本变量 $\bar{x}_i$ 设为 b_i，来计算初始线性规划变量 $\bar{x}_1$，$\bar{x}_2$，…，$\bar{x}_n$ 的一个解。在定理 29.10 中，我们将看到这个解是线性规划的一个最优解。最后，第 16 行返回初始线性规划变量的计算值。

为了证明 SIMPLEX 是正确的，首先证明如果 SIMPLEX 有一个初始可行解且最终会结束，
797 则它要么返回一个可行解，要么确定线性规划是无界的。然后，说明 SIMPLEX 会终止。最后，在 29.4 节中要说明返回的解是最优的。

引理 29.2　给定一个线性规划(A, b, c)，假设在 SIMPLEX 的第 1 行中 INITIALIZE-SIMPLEX 的调用返回一个基本解可行的松弛型。那么如果 SIMPLEX 在第 16 行返回一个解，则此解是线性规划的可行解。如果 SIMPLEX 在第 10 行返回"无界"，则线性规划是无界的。

证明：我们使用下面三部分的循环不变式：

在第 2～11 行 while 循环的每次迭代开始，

1)松弛型等价于调用 INITIALIZE-SIMPLEX 所返回的松弛型。

2)对每个$i \in B$，有$b_i \geqslant 0$。

3)和松弛型相关联的基本解是可行的。

初始化：松弛型的等价性对于第一次迭代是平凡的。在引理的陈述中，假设在 SIMPLEX 的第 1 行调用 INITIALIZE-SIMPLEX 返回一个基本解可行的松弛型。因此，不变式的第三部分成立。进一步，因为每一个基本变量x_i在基本解中都被设置为b_i，而且基本解的可行性隐含着每个基本变量x_i是非负的，所以有$b_i \geqslant 0$。因此，不变式的第二部分也成立。

保持：假设第 10 行的 return 声明没有被执行，我们来说明循环不变式会被维护。在讨论终止性时，我们将处理第 10 行被执行的情形。

while 循环的一次迭代交换基本变量与非基本变量的角色。所执行的唯一操作包括解方程式和将一个方程替换为另一个方程，因此其松弛型等价于前一次迭代中的松弛性，而根据循环不变式，它又等价于初始的松弛型。

现在来说明循环不变式的第二个部分。假设在 while 循环每次迭代的开始，对每个$i \in B$有$b_i \geqslant 0$，我们来说明在第 11 行调用 PIVOT 之后这些不等式仍然成立。因为变量b_i和基本变量的集合B的唯一改变发生在这个赋值中，这就足以说明第 11 行保持了这部分的不变式。令b_i、a_{ij}和B表示调用 PIVOT 之前的值，令$\hat{b}_i$表示从 PIVOT 返回的值。 798

首先，因为根据循环不变式有$b_l \geqslant 0$，根据 SIMPLEX 的第 5 行有$a_{le} > 0$，根据 PIVOT 的第 2 行有$\hat{b}_e = b_l / a_{le}$，因此有$\hat{b}_e \geqslant 0$。

对于剩下的下标$i \in B - l$，有

$$\begin{aligned} \hat{b}_i &= b_i - a_{ie}\hat{b}_e \quad \text{(根据 PIVOT 的第 8 行)} \\ &= b_i - a_{ie}(b_l / a_{le}) \quad \text{(根据 PIVOT 的第 2 行)} \end{aligned} \tag{29.79}$$

有两种情况要考虑，视$a_{ie} > 0$或$a_{ie} \leqslant 0$而定。如果$a_{ie} > 0$，则因为我们选择l使得

$$b_l / a_{le} \leqslant b_i / a_{ie} \text{(对所有的 } i \in B\text{)} \tag{29.80}$$

有

$$\begin{aligned} \hat{b}_i &= b_i - a_{ie}(b_l / a_{le}) \quad \text{(由式(29.79)可得)} \\ &\geqslant b_i - a_{ie}(b_i / a_{ie}) \quad \text{(由式(29.80)可得)} \\ &= b_i - b_i = 0 \end{aligned}$$

所以$\hat{b}_i \geqslant 0$。如果$a_{ie} \leqslant 0$，则因为a_{le}、b_i和b_l都是非负的，式(29.79)隐含$\hat{b}_i$也必须为非负。

现在我们论证基本解是可行的，也就是说，所有的变量都是非负值。非基本变量被设为 0，因此是非负的。每个基本变量x_i由如下的等式定义：

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j$$

基本解设置$\bar{x}_i = b_i$。利用循环不变式的第二部分，得到结论：每个基本变量$\bar{x}_i$都是非负的。

终止：while 循环以两种方式之一来结束。如果它因为第 2 行中的条件而终止，则当前的基

本解是可行的，而且在第 16 行中返回这个解。另外一个终止的方式是在第 10 行返回“无界”。在这种情况下，对于第 4～7 行中 for 循环的每次迭代，当第 5 行被执行时，我们发现 $a_{ie} \leqslant 0$。令 x 表示返回“无界”的那次迭代的开始处的松弛型所属的基本解。考虑如下定义的解 $\bar{x}$：

799

$$\bar{x}_i = \begin{cases} \infty & \text{若 } i = e \\ 0 & \text{若 } i \in N - \{e\} \\ b_i - \sum_{j \in N} a_{ij}\, \bar{x}_j & \text{若 } i \in B \end{cases}$$

现在我们说明这个解是可行的，也就是说，所有的变量非负。除了 $\bar{x}_e$ 之外的非基本变量都是 0，而 $\bar{x}_e$ 是正的；因此所有的非基本变量是非负的。对于每个基本变量 $\bar{x}_i$，有

$$\bar{x}_i = b_i - \sum_{j \in N} a_{ij}\, \bar{x}_j = b_i - a_{ie}\, \bar{x}_e$$

循环不变式隐含着 $b_i \geqslant 0$，而且有 $a_{ie} \leqslant 0$ 以及 $\bar{x}_e = \infty > 0$。所以，$\bar{x}_i \geqslant 0$。

现在来证明解 $\bar{x}$ 的目标值是无界的。目标值为

$$z = v + \sum_{j \in N} c_j\, \bar{x}_j = v + c_e \bar{x}_e$$

因为 $c_e > 0$（根据第 3 行）和 $\bar{x}_e = \infty$，目标值为 ∞，因此这个线性规划是无界的。 ■

在每次迭代中，除了集合 N 和 B 外，SIMPLEX 还要维护 A、b、c 和 v。虽然显式地维护 A、b、c 和 v 对有效实现单纯形算法是非常重要的，但它不是严格必需的。换句话说，松弛型是由基本变量和非基本变量的集合唯一决定的。在证明这个事实之前，先证明一个有用的代数引理。

引理 29.3　令 I 表示一个下标的集合。对每个 $i \in I$，令 α_i 和 β_i 表示实数，令 x_i 表示一个实数值的变量。令 γ 表示任意的实数。假设对于 x_i 的任何取值，有

$$\sum_{i \in I} \alpha_i x_i = \gamma + \sum_{i \in I} \beta_i x_i \tag{29.81}$$

则对于每个 $i \in I$，$\alpha_i = \beta_i$，且 $\gamma = 0$。

证明：因为式(29.81)对 x_i 的任何取值都成立，所以可以使用特殊的值来得出关于 α、β 和 γ 的结论。如果对每个 $i \in I$，令 $x_i = 0$，则得出结论 $\gamma = 0$。现在挑选任意一个下标 $i \in I$，令 $x_i = 1$ 且对 $k \neq i$ 令 $x_k = 0$。则必然有 $\alpha_i = \beta_i$。因为 i 是 I 内的任意一个下标，所以结论为对每个 $i \in I$ 都有 $\alpha_i = \beta_i$。 ■

800　现在我们来说明线性规划的松弛型是由基本变量的集合唯一决定的。

引理 29.4　令 (A, b, c) 表示一个线性规划的标准型。已知基本变量的集合 B，则所属的松弛型被唯一确定。

证明：为了导出矛盾，假设有两个不同的松弛型有相同的基本变量集合 B。松弛型也必须有相同的 $N = \{1, 2, \cdots, n+m\} - B$ 个非基本变量的集合。把第一个松弛型写作

$$z = v + \sum_{j \in N} c_j x_j \tag{29.82}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{当 } i \in B \text{ 时} \tag{29.83}$$

第二个为

$$z = v' + \sum_{j \in N} c'_j x_j \tag{29.84}$$

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{当 } i \in B \text{ 时} \tag{29.85}$$

考虑将式(29.85)中的每个等式减去式(29.83)中对应的等式所形成的等式系统。所产生的系

统为

$$0 = (b_i - b'_i) - \sum_{j \in N}(a_{ij} - a'_{ij})x_j \quad \text{当 } i \in B \text{ 时}$$

或等价地，

$$\sum_{j \in N} a_{ij}x_j = (b_i - b'_i) + \sum_{j \in N} a'_{ij}x_j \quad \text{当 } i \in B \text{ 时}$$

现在，对每个 $i \in B$，以 $a_i = a_{ij}$，$\beta_i = a'_{ij}$ 和 $\gamma = b_i - b'_i$ 来应用引理 29.3。因为 $a_i = \beta_i$，所以对每个 $j \in N$ 有 $a_{ij} = a'_{ij}$，而且因为 $\gamma = 0$，所以有 $b_i = b'_i$。因此，对这两个松弛型，A 和 b 与 A' 和 b' 是相同的。利用类似的论证，练习 29.3-1 说明必定也有 $c = c'$ 和 $v = v'$，所以这两个松弛型必定是相同的。■

还需要说明 SIMPLEX 会终止，而且当它终止时，它返回的解是最优的。29.4 节将研究最优性。现在我们来讨论终止性。

终止性

在本节开头所给的例子中，单纯形算法的每次迭代会增加和基本解相关联的目标值。如练习 29.3-2 要求读者所证明的那样，SIMPLEX 的任何迭代都不会减小和基本解相关联的目标值。 801
遗憾的是，可能会有一次迭代维持目标值不变。这个现象叫做退化，我们现在开始详细地研究它。

目标值通过在 PIVOT 的第 13 行中的赋值 $\hat{v} \leftarrow v + c_e\hat{b}_e$ 被改变。因为只有在 $c_e > 0$ 时，SIMPLEX 才会调用 PIVOT，让目标值保持不变(即 $\hat{v} = v$)的唯一途径就是让 $\hat{b}_e$ 为 0。这个值是在 PIVOT 的第 2 行由 $\hat{b}_e \leftarrow b_l / a_{le}$ 赋值的。因为总是以 $a_{le} \neq 0$ 来调用 PIVOT，所以可以看到为了让 $\hat{b}_e$ 等于 0 来让目标值保持不变，必须有 $b_l = 0$。

确实，这个情况可能发生。考虑线性规划

$$\begin{aligned} z &= & & & x_1 &+ x_2 &+ x_3 \\ x_4 &= 8 &- x_1 &- x_2 & & \\ x_5 &= & & x_2 &- x_3 & \end{aligned}$$

假设选择 x_1 来作为换入变量，选择 x_4 作为换出变量。在旋转之后，得到

$$\begin{aligned} z &= 8 & &+ x_3 &- x_4 \\ x_1 &= 8 - x_2 & &- x_4 \\ x_5 &= & x_2 &- x_3 & \end{aligned}$$

此时唯一的选择是以 x_3 作为换入变量、x_5 作为换出变量来进行旋转。因为 $b_5 = 0$，在旋转后目标值 8 保持不变：

$$\begin{aligned} z &= 8 + x_2 &- x_4 &- x_5 \\ x_1 &= 8 - x_2 &- x_4 & \\ x_3 &= & x_2 &- x_5 \end{aligned}$$

目标值没有变化，但是表现形式变了。幸运地，如果再一次旋转，以 x_2 为换入变量、x_1 为换出变量，目标值将增加，且单纯形算法可以继续执行。

现在我们说明退化是唯一可能让单纯形算法不终止的方式。回顾我们的假设：根据某个确定性的规则，SIMPLEX 在第 3 行和第 8 行分别选择下标 e 和 l。如果在两次不同迭代中的松弛型相同，则称 SIMPLEX 发生了循环。在这种情况下，因为 SIMPLEX 是个确定性的算法，它会永远循环同一序列的松弛型。

引理 29.5　如果 SIMPLEX 在至多$\binom{n+m}{m}$次迭代内不能中止，则它是循环的。

证明：根据引理 29.4，基本变量集合 B 唯一确定了一个松弛型。有 $n+m$ 个变量，且 $|B|=m$，所以共有$\binom{n+m}{m}$种选择 B 的方式。因此只有$\binom{n+m}{m}$个互异的松弛型。所以，如果 SIMPLEX 执行超过$\binom{n+m}{m}$次的迭代，则它必然循环。 ■

802

循环在理论上是可能的，但是非常罕见。更小心地选择换入变量和换出变量可以避免其发生。一种选择是稍微扰动输入，使得不可能有两个解的目标值相等。第二种选择是按字典顺序打破一样的目标值，而第三种选择是通过总是选择具有最小下标的变量来打破这种相等的局面。最后一个策略叫做 Bland 规则。有关这些策略会避免循环的证明在此就省略了。

引理 29.6　如果在 SIMPLEX 的第 3 行和第 8 行，总是选择具有最小下标的变量来打破一样的目标值，那么 SIMPLEX 必定会终止。 ■

我们以下面的引理来总结这一节。

引理 29.7　假设 INITIALIZE-SIMPLEX 返回一个基本解可行的松弛型，则 SIMPLEX 要么报告线性规划是无界的，要么它在至多$\binom{n+m}{m}$次迭代内得到一个可行解来终止。

证明：引理 29.2 和引理 29.6 说明如果 INITIALIZE-SIMPLEX 返回一个基本解可行的松弛型，则 SIMPLEX 要么报告线性规划是无界的，要么以一个可行解来终止。根据引理 29.5 的反命题，如果 SIMPLEX 以一个可行解来终止，则它必然是在至多$\binom{n+m}{m}$次迭代内终止。 ■

练习

29.3-1　说明必定有情况 $c=c'$ 和 $v=v'$，来完成引理 29.4 的证明。

29.3-2　说明在 SIMPLEX 的第 11 行调用 PIVOT 永远不会减小 v 的值。

29.3-3　假设将一个标准型的线性规划(A, b, c)转换为松弛型。说明当且仅当 $b_i \geqslant 0$ 时$(i=1, 2, \cdots, m)$，基解是可行的。

803

29.3-4　利用 SIMPLEX 求解下面的线性规划：

最大化
$$18x_1 + 12.5x_2$$
满足约束
$$\begin{aligned} x_1 + x_2 &\leqslant 20 \\ x_1 &\leqslant 12 \\ x_2 &\leqslant 16 \\ x_1, x_2 &\geqslant 0 \end{aligned}$$

29.3-5　利用 SIMPLEX 求解下面的线性规划：

最大化
$$-5x_1 - 3x_2$$
满足约束
$$\begin{aligned} x_1 - x_2 &\leqslant 1 \\ 2x_1 + x_2 &\leqslant 2 \\ x_1, x_2 &\geqslant 0 \end{aligned}$$

29.3-6　利用 SIMPLEX 求解下面的线性规划：

最小化　　$x_1 + x_2 + x_3$

满足约束

$$\begin{aligned} 2x_1 + 7.5x_2 + 3x_3 &\geqslant 10\,000 \\ 20x_1 + 5x_2 + 10x_3 &\geqslant 30\,000 \\ x_1, x_2, x_3 &\geqslant 0 \end{aligned}$$

29.4　对偶性

前面已经证明了在某些假设下，SIMPLEX 是会终止的。然而，我们还没有说明它确实能找到线性规划的一个最优解。为此，我们引入一个有效的概念，叫做线性规划对偶性(linear-programming duality)。

对偶性是个非常重要的性质。在一个最优化问题中，一个对偶问题的识别几乎总是伴随着一个多项式时间算法的发现。对偶性也可以用来证明某个解的确是最优解。 804

例如，假设已知一个最大流问题的实例，要寻找的是一个值为 $|f|$ 的流 f。如何才能知道 f 是否是一个最大流呢？根据最大流最小割定理(定理 26.7)，如果可以找到一个割的值也是 $|f|$，就证实了 f 确实是一个最大流。这是对偶性的一个例子：已知一个最大化问题，定义一个相关的最小化问题，来让这两个问题有相同的最优目标值。

已知一个目标是最大化的线性规划，要描述如何制定一个对偶线性规划，它的目标是最小化，而且最优值与原始线性规划的相同。在表示对偶线性规划时，称原始的线性规划为原(primal)。

给定一个标准型的原线性规划，如式(29.16)～式(29.18)所示，定义其对偶线性规划为

最小化　　$\sum_{i=1}^{m} b_i y_i$　　(29.86)

满足约束

$$\sum_{i=1}^{m} a_{ij} y_i \geqslant c_j \text{ 对 } j = 1,2,\cdots,n \tag{29.87}$$

$$y_i \geqslant 0 \text{ 对 } i = 1,2,\cdots,m \tag{29.88}$$

为了构造对偶问题，我们将最大化改成最小化，将约束右边的与目标函数的系数的角色互换，并且将小于等于号改成大于等于号。在原问题的 m 个约束中，每一个在对偶问题中都有一个对应的变量 y_i；在对偶问题的 n 个约束中，每一个在原问题中都有一个对应的变量 x_j。例如，考虑式(29.56)至式(29.60)中给出的线性规划。这个线性规划的对偶问题为

最小化　　$30y_1 + 24y_2 + 36y_3$　　(29.89)

满足约束

$$y_1 + 2y_2 + 4y_3 \geqslant 3 \tag{29.90}$$

$$y_1 + 2y_2 + y_3 \geqslant 1 \tag{29.91}$$

$$3y_1 + 5y_2 + 2y_3 \geqslant 2 \tag{29.92}$$

$$y_1, y_2, y_3 \geqslant 0 \tag{29.93}$$

在定理 29.10 中，我们将证明对偶线性规划的最优值总是等于原线性规划的最优值。而且，单纯形算法实际上隐含地同时解决了原线性规划和对偶线性规划，因此提供了最优性的一个证明。

我们从说明弱对偶性开始，它表示原线性规划的任意可行解的数值不大于对偶问题的任意可行解的值。 805

引理 29.8(线性规划弱对偶性)　令 $\bar{x}$ 表示式(29.16)～式(29.18)中的原线性规划的任意一个

可行解，令 $\bar{y}$ 表示式(29.86)～式(29.88)中的对偶问题的任意一个可行解。则

$$\sum_{j=1}^{n} c_j \bar{x}_j \leqslant \sum_{i=1}^{m} b_i \bar{y}_i$$

证明：有

$$\begin{aligned}\sum_{j=1}^{n} c_j \bar{x}_j &\leqslant \sum_{j=1}^{n}\left(\sum_{i=1}^{m} a_{ij}\bar{y}_i\right)\bar{x}_j \quad \text{(见式(29.87))} \\ &= \sum_{i=1}^{m}\left(\sum_{j=1}^{n} a_{ij}\bar{x}_j\right)\bar{y}_i \\ &\leqslant \sum_{i=1}^{m} b_i \bar{y}_i \quad \text{(见式(29.17))}\end{aligned}$$

■

推论 29.9　令 $\bar{x}$ 表示原线性规划 (A, b, c) 的一个可行解，且令 $\bar{y}$ 表示相应的对偶问题的一个可行解。如果

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i$$

则 $\bar{x}$ 和 $\bar{y}$ 分别是原线性规划和对偶线性规划的最优解。

证明：根据引理 29.8，原问题可行解的目标值不会超过对偶问题的可行解的目标值。原线性规划是一个最大化问题，而对偶线性规划是一个最小化问题。因此，如果可行解 $\bar{x}$ 和 $\bar{y}$ 有相同的目标值，那么没有一个可以被改进。■

在证明总有对偶解的值等于原问题最优解的值之前，先来描述如何找出这样的一个解。当在式(29.56)至式(29.60)中的线性规划上执行单纯形算法时，最后的迭代得到松弛型式(29.75)～式(29.78)，$B=\{1, 2, 4\}$ 和 $N=\{3, 5, 6\}$。如下面将看到的那样，和最终松弛型对应的基本解是线性规划的一个最优解；因此线性规划式(29.56)～式(29.60)的一个最优解是 $(\bar{x}_1, \bar{x}_2, \bar{x}_3)=(8, 4, 0)$，目标值为 $(3 \cdot 8)+(1 \cdot 4)+(2 \cdot 0)=28$。如下面将看到的那样，可以读出一个最优对偶解：原目标函数的系数的负值是对偶变量的值。更准确地说，假设原问题的最后一个松弛型为

806

$$z = v' + \sum_{j \in N} c'_j x_j \quad x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{当 } i \in B \text{ 时}$$

则一个最优对偶解是设定

$$\bar{y}_i = \begin{cases} -c'_{n+i} & \text{若 } (n+i) \in N \\ 0 & \text{其他情况下} \end{cases} \tag{29.94}$$

所以，式(29.89)～式(29.93)定义的对偶线性规划的一个最优解是 $\bar{y}_1=0$（因为 $n+1=4 \in B$），$\bar{y}_2=-c'_5=1/6$，以及 $\bar{y}_3=-c'_6=2/3$。求对偶目标函数(式(29.89))的值，得到目标值为 $(30 \cdot 0)+(24 \cdot (1/6))+(36 \cdot (2/3))=28$，这证实了原问题的目标值确实等于对偶问题的目标值。综合这些计算与引理 29.8，得到原线性规划的最优目标值是 28 的证明。现在来证明在一般情况下，对偶问题的一个最优解及原问题解的最优性的证明，都可以通过这个方式来获得。

定理 29.10（线性规划对偶性）　假设 SIMPLEX 在原线性规划 (A, b, c) 上返回值 $\bar{x}=(\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n)$。令 N 和 B 表示最终松弛型的非基本变量和基本变量的集合，令 c' 表示最终松弛型中的系数，令 $\bar{y}=(\bar{y}_1, \bar{y}_2, \cdots, \bar{y}_m)$ 由式(29.94)定义。则 $\bar{x}$ 是原线性规划的一个最优解，$\bar{y}$ 是对偶线性规划的一个最优解，而且

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i \tag{29.95}$$

证明：根据推论 29.9，如果可以找到满足式(29.95)的可行解$\bar{x}$和$\bar{y}$，则$\bar{x}$和$\bar{y}$必定是最优的原解和对偶解。现在要证明在定理叙述中描述的$\bar{x}$和$\bar{y}$满足式(29.95)。

假设在一个原始线性规划上执行 SIMPLEX，如式(29.16)至式(29.18)中所给出的那样。这个算法处理一系列的松弛型，直到以一个最终松弛型终止，此时目标函数为

$$z = v' + \sum_{j \in N} c'_j x_j \tag{29.96}$$

807

因为 SIMPLEX 以一个解来终止，由第 2 行的条件得知

$$c'_j \leqslant 0 \quad \text{对于所有的 } j \in N \tag{29.97}$$

如果定义

$$c'_j = 0 \quad \text{对于所有的 } j \in B \tag{29.98}$$

则可以重写式(29.96)为

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ &= v' + \sum_{j \in N} c'_j x_j + \sum_{j \in B} c'_j x_j \quad (\text{因为 } c'_j = 0, \text{若 } j \in B \text{ 的话}) \\ &= v' + \sum_{j=1}^{n+m} c'_j x_j \quad (\text{因为 } N \cup B = \{1,2,\cdots,n+m\}) \end{aligned} \tag{29.99}$$

对于属于这个最终松弛型的基本解$\bar{x}$，对于所有的$j \in N$有$\bar{x}_j = 0$，且$z = v'$。因为所有的松弛型都是等价的，所以，如果在$\bar{x}$上计算原始的目标函数，也必定会得到同样的目标值，即

$$\sum_{j=1}^{n} c_j \bar{x}_j = v' + \sum_{j=1}^{n+m} c'_j \bar{x}_j \tag{29.100}$$

$$\begin{aligned} &= v' + \sum_{j \in N} c'_j \bar{x}_j + \sum_{j \in B} c'_j \bar{x}_j \\ &= v' + \sum_{j \in N} (c'_j \cdot 0) + \sum_{j \in B} (0 \cdot \bar{x}_j) \\ &= v' \end{aligned} \tag{29.101}$$

现在要证明由式(29.94)所定义的$\bar{y}$对于对偶线性规划来说是可行的，而且其目标值$\sum_{i=1}^{m} b_i \bar{y}_i$等于$\sum_{j=1}^{n} c_j \bar{x}_j$。式(29.100)说明了在$\bar{x}$上计算的第一个和最后一个松弛型是相等的。更一般地，所有松弛型的等价意味着对任意变量集合$x = (x_1, x_2, \cdots, x_n)$，有

$$\sum_{j=1}^{n} c_j x_j = v' + \sum_{j=1}^{n+m} c'_j x_j$$

所以，对任何变量x的集合，有

808

$$\begin{aligned} \sum_{j=1}^{n} c_j x_j &= v' + \sum_{j=1}^{n+m} c'_j x_j = v' + \sum_{j=1}^{n} c'_j x_j + \sum_{j=n+1}^{n+m} c'_j x_j \\ &= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} c'_{n+i} x_{n+i} \\ &= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} (-\bar{y}_i) x_{n+i} \quad (\text{见式(29.94)}) \\ &= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} (-\bar{y}_i)\left(b_i - \sum_{j=1}^{n} a_{ij} x_j\right) \quad (\text{见式(29.32)}) \\ &= v' + \sum_{j=1}^{n} c'_j x_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{i=1}^{m} \sum_{j=1}^{n} (a_{ij} x_j) \bar{y}_i \end{aligned}$$

$$= v' + \sum_{j=1}^{\dot{n}} c'_j x_j - \sum_{i=1}^{\dot{m}} b_i \bar{y}_i + \sum_{j=1}^{\dot{n}} \sum_{i=1}^{\dot{m}} (a_{ij} \bar{y}_i) x_j$$

$$= \left(v' - \sum_{i=1}^{\dot{m}} b_i \bar{y}_i\right) + \sum_{j=1}^{\dot{n}} \left(c'_j + \sum_{i=1}^{\dot{m}} a_{ij} \bar{y}_i\right) x_j$$

因此

$$\sum_{j=1}^{\dot{n}} c_j x_j = \left(v' - \sum_{i=1}^{\dot{m}} b_i \bar{y}_i\right) + \sum_{j=1}^{\dot{n}} \left(c'_j + \sum_{i=1}^{\dot{m}} a_{ij} \bar{y}_i\right) x_j \tag{29.102}$$

应用引理 29.3 到式(29.102)上，得

$$v' - \sum_{i=1}^{\dot{m}} b_i \bar{y}_i = 0 \tag{29.103}$$

$$c'_j + \sum_{i=1}^{\dot{m}} a_{ij} \bar{y}_i = c_j \quad \text{如果} \quad j = 1,2,\cdots,n \tag{29.104}$$

根据式(29.103)，有 $\sum_{i=1}^{\dot{m}} b_i \bar{y}_i = v'$，因此对偶问题的目标值 $\left(\sum_{i=1}^{\dot{m}} b_i \bar{y}_i\right)$ 等于原问题的目标值(v')。还
809 需证明解 $\bar{y}$ 对于对偶问题是可行的。根据式(29.97) 和式(29.98)，对于所有 $j = 1,2,\cdots,n+m$，有 $c'_j \leqslant 0$。因此，对任意的 $i = 1,2,\cdots,m$，式(29.104) 隐含着

$$c_j = c'_j + \sum_{i=1}^{\dot{m}} a_{ij} \bar{y}_i \leqslant \sum_{i=1}^{\dot{m}} a_{ij} \bar{y}_j$$

这满足对偶性的约束式(29.87)。最后，因为对每个 $j \in N \cup B$ 有 $c'_j \leqslant 0$，当依据式(29.94)来设置 $\bar{y}$ 时，每个 $\bar{y}_i \geqslant 0$，并因此也会满足非负约束。 ■

至此，我们已经证明了给定一个可行的线性规划，如果 INITIALIZE-SIMPLEX 返回一个可行解，而且如果 SIMPLEX 终止时不是返回“无界”，那么返回的解确实是一个最优解。我们也说明了如何构造对偶线性规划的一个最优解。

练习

29.4-1 写出练习 29.3-4 中给出的线性规划的对偶问题。

29.4-2 假设有一个不是标准型的线性规划。可以通过先将其转换成标准型，再取对偶来产生它的对偶问题。然而，更方便的是能够直接产生对偶问题。说明已知一个任意的线性规划，如何直接取该线性规划的对偶。

29.4-3 写出式(29.47)～式(29.50)行中给出的最大流线性规划的对偶问题。说明如何将这个形式理解成一个最小割问题。

29.4-4 写出式(29.51)～式(29.55)行中给出的最小费用流线性规划的对偶问题。说明如何将这个问题用图和流来解释。

29.4-5 证明：线性规划的对偶的对偶是原线性规划。

810 **29.4-6** 第 26 章的哪一个结果可以解释成最大流的弱对偶？

29.5　初始基本可行解

在本节，首先讲解如何测试一个线性规划是否是可行的，如果它是，如何产生一个基本解可行的松弛型。通过证明线性规划的基本定理来结束，即 SIMPLEX 过程永远产生正确的结果。

找出一个初始解

在 29.3 节，我们假设有一个过程 INITIALIZE-SIMPLEX，它确定线性规划是否有可行解，如果有，则给出一个基本解可行的松弛型。在这里叙述这个过程。

一个线性规划可能是可行的，但是初始基本解可能不是可行的。例如，考虑下列的线性规划：

最大化

$$2x_1 - x_2 \tag{29.105}$$

满足约束

$$2x_1 - x_2 \leqslant 2 \tag{29.106}$$

$$x_1 - 5x_2 \leqslant -4 \tag{29.107}$$

$$x_1, x_2 \geqslant 0 \tag{29.108}$$

如果要将这个线性规划转换成松弛型，基本解将设 $x_1 = 0$ 和 $x_2 = 0$。这个解违反了约束(式(29.107))，因此它不是一个可行解。所以 INITIALIZE-SIMPLEX 无法仅返回明显的松弛型。根据检查，这个线性规划是否有可行解还不清楚。为了确定它是否有可行解，可以指定一个辅助线性规划。对这个辅助线性规划，(稍加努力)即可找到一个基本解可行的松弛型。进一步地，这个辅助线性规划的解将决定初始的线性规划是否是可行的；如果是，则它将提供一个可行解来让我们初始化 SIMPLEX。 811

引理 29.11　令 L 是一个标准型的线性规划，如式(29.16)～式(29.18)中所给出的那样。令 L_{aux} 是下面带有 $n+1$ 个变量的线性规划：

最大化

$$-x_0 \tag{29.109}$$

满足约束

$$\sum_{j=1}^{n} a_{ij}x_j - x_0 \leqslant b_i \text{ 对于 } i = 1,2,\cdots,m \tag{29.110}$$

$$x_j \geqslant 0 \text{ 对于 } j = 0,1,\cdots,n \tag{29.111}$$

则当且仅当 L_{aux} 的最优目标值为 0 时，L 是可行的。

证明：假设 L 有一个可行解 $\bar{x} = (\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n)$。则解 $\bar{x}_0 = 0$ 并上 $\bar{x}$ 是 L_{aux} 的一个可行解，目标值为 0。因为 $\bar{x}_0 \geqslant 0$ 是 L_{aux} 的一个约束，而且目标函数是去最大化 $-x_0$，所以这个解对于 L_{aux} 肯定是最优的。

相反地，假设 L_{aux} 的最优目标值是 0。则 $\bar{x}_0 = 0$，而且其余的变量 $\bar{x}$ 的值满足 L 的约束。 ■

现在来描述找出标准型线性规划的一个初始可行解的策略。

```
INITIALIZE-SIMPLEX(A, b, c)
 1  let l be the index of the minimum b_i
 2  if b_l ≥ 0        ▷ Is the initial basic solution feasible?
 3       then return({1, 2, ..., n}, {n+1, n+2, ..., n+m}, A, b, c, 0)
 4  form L_aux by adding -x_0 to the left-hand side of each equation and setting the objective function to -x_0
 5  let(N, B, A, b, c, v) be the resulting slack form for L_aux
 6  ▷ L_aux has n+1 nonbasic variables and m basic variables.
 7  (N, B, A, b, c, v) ← PIVOT(N, B, A, b, c, v, l, 0)
 8  ▷ The basic solution is now feasible for L_aux.
 9  iterate the while loop of lines 2-11 of SIMPLEX until an optimal solution to L_aux is found
10  if the basic solution sets x̄_0 = 0
11       then return the final slack form with x_0 removed and the original objective function restored
12       else return "infeasible"
```

INITIALIZE-SIMPLEX 的执行过程如下。在 1～3 行，隐含地测试 L 的初始松弛型的基本解，而它由这些给定：$N = \{1, 2, \cdots, n\}$；$B = \{n+1, n+2, \cdots, n+m\}$；对于所有 $i \in B$， 812

$\bar{x}_i = b_i$；以及对于所有 $j \in N$，$\bar{x}_j = 0$（由于 A、b 和 c 的值在松弛型中与标准型中相同，建立松弛型不需要费什么力气）。如果这个基本解是可行的，亦即 $\bar{x}_i \geqslant 0$（对于所有 $i \in N \cup B$），则返回这个松弛型。否则，在第 4 行像在引理 29.11 中一样构造辅助线性规划 L_{aux}。因为 L 的初始基本解是不可行的，所以 L_{aux} 的松弛型的初始基本解也不是可行的。因此在第 7 行执行一个对 PIVOT 的调用，以 x_0 为换入变量，x_l 为换出变量，其中下标 l 在第 1 行中被选择为最小的 b_i 的下标。稍后会看到，由 PIVOT 的调用所产生的基本解是可行的。现在有一个基本解可行的松弛型，可在第 9 行重复调用 PIVOT 来完全解决辅助线性规划。正如第 10 行中的测试所表明的那样，如果找到了一个目标值为 0 的 L_{aux} 的最优解，那么就可以在第 11 行中为 L 构造一个基本解可行的松弛型。为了做到这一点，从约束中删除所有 x_0 的项，并且恢复 L 的初始目标函数。初始目标函数可能同时包含基本变量和非基本变量。因此，在这个目标函数中，将每个基本变量用其关联的约束的右边来替换。另一方面，如果在第 10 行中发现初始线性规划 L 是不可行的，那么在第 12 行中就返回这一信息。

现在来说明 INITIALIZE-SIMPLEX 在线性规划（式(29.105)～式(29.108)）上的操作。如果可以找到 x_1 和 x_2 的非负值以满足不等式(29.106)至不等式(29.107)，则线性规划是可行的。利用引理 29.11，制定辅助线性规划为：

最大化

$$-x_0 \tag{29.112}$$

满足约束

$$2x_1 - x_2 - x_0 \leqslant 2 \tag{29.113}$$

$$x_1 - 5x_2 - x_0 \leqslant -4 \tag{29.114}$$

$$x_1, x_2, x_0 \geqslant 0$$

根据引理 29.11，如果这个辅助线性规划的最优目标值是 0，则初始线性规划有一个可行解。如果这个辅助线性规划的最优目标值是正数，则初始线性规划没有可行解。

将这个线性规划写成松弛型，可得：

$$\begin{aligned} z &= -x_0 \\ x_3 &= 2 - 2x_1 + x_2 + x_0 \\ x_4 &= -4 - x_1 + 5x_2 + x_0 \end{aligned}$$

我们还没有走出“迷雾”，因为将设定 $x_4 = -4$ 的基本解对这个辅助线性规划不是可行的。然
813 而，可以利用对 PIVOT 的调用将此松弛型转换成基本解可行的。如第 7 行指出的那样，选择 x_0 作为换入变量。在第 1 行，选择 x_4 作为换出变量，而它是基本解中最小的基本变量。在旋转后，得到松弛型

$$\begin{aligned} z &= -4 - x_1 + 5x_2 - x_4 \\ x_0 &= 4 + x_1 - 5x_2 + x_4 \\ x_3 &= 6 - x_1 - 4x_2 + x_4 \end{aligned}$$

对应的基本解是 $(x_0, x_1, x_2, x_3, x_4) = (4, 0, 0, 6, 0)$，这是可行的。现在重复调用 PIVOT，直到得到 L_{aux} 的一个最优解。在这种情况下，以 x_2 为换入变量、x_0 为换出变量的 PIVOT 调用得到

$$\begin{aligned} z &= -x_0 \\ x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \end{aligned}$$

这个松弛型是辅助线性规划的最优解。因为这个解有 $x_0=0$，我们知道初始问题是可行的。而且，因为 $x_0=0$，所以可以将其从约束集合中删除。然后可以使用原始的目标函数，包含适当的替换使得它只包含非基本变量。在这个例子中，得到目标函数

$$2x_1-x_2=2x_1-\left(\frac{4}{5}-\frac{x_0}{5}+\frac{x_1}{5}+\frac{x_4}{5}\right)$$

取 $x_0=0$，然后再进行简化，得到目标函数

$$\frac{4}{5}+\frac{9x_1}{5}-\frac{x_4}{5}$$

以及松弛型

$$\begin{aligned} z &= \frac{4}{5}+\frac{9x_1}{5}-\frac{x_4}{5} \\ x_2 &= \frac{4}{5}+\frac{x_1}{5}+\frac{x_4}{5} \\ x_3 &= \frac{14}{5}-\frac{9x_1}{5}+\frac{x_4}{5} \end{aligned}$$

这个松弛型有一个可行的基本解，可以将它返回给过程 SIMPLEX。

现在正式地证明 INITIALIZE-SIMPLEX 的正确性。

引理 29.12　如果线性规划 L 没有可行解，则 INITIALIZE-SIMPLEX 返回“不可行”。否则，它返回一个基本解可行的合法松弛型。

814

证明：首先假设线性规划 L 没有可行解。则根据定理 29.11，在式(29.109)至式(29.111)中定义的 L_{aux} 的最优目标值不是零，而且根据 x_0 上的非负约束，最优解必然有一个负的目标值。而且，这个目标值必定是有限的，因为对 $i=1,2,\cdots,n$ 设定 $x_i=0$，且 $x_0=|\min_{i=1}^{m}\{b_i\}|$ 是可行的，这个解的目标值为 $-|\min_{i=1}^{m}\{b_i\}|$。因此，INITIALIZE-SIMPLEX 的第 9 行将找到一个有负的目标值的解。令 $\bar{x}$ 表示和最终松弛型相关联的基本解。不能有 $\bar{x}_0=0$，因为 L_{aux} 将会有目标值为 0，与目标值为负值的事实相矛盾。因此第 10 行的测试产生第 12 行所返回的“不可行”。

现在假设线性规划 L 确实有一个可行解。从练习 29.3-3 得知如果对于 $i=1,2,\cdots,m$ 有 $b_i\geqslant 0$，则和初始松弛型相关联的基本解是可行的。在这种情况下，第 2～3 行将返回和输入相关联的松弛型。(将标准型转换为松弛型并没有太多工作要做，因为 A、b 和 c 在二者中都是相同的。)

在余下的证明中，要处理线性规划是可行的但是在第 3 行中并不返回的情况。我们要论证在这种情况下，第 4～9 行找到 L_{aux} 的一个可行解，其目标值为 0。首先，由第 1～2 行，必定有

$$b_l<0$$

和

$$b_l\leqslant b_i \quad \text{对每个 } i\in B \tag{29.115}$$

在第 7 行中，执行一个选主元操作，其中换出变量 x_l 是具有最小的 b_i 的等式的左边，而换入变量是额外加入的变量 x_0。现在来证明在这个主元之后，b 的所有元素都是非负的，因此 L_{aux} 的基本解是可行的。令 $\bar{x}$ 表示调用 PIVOT 之后的基本解，令 $\hat{b}$ 和 $\hat{B}$ 表示 PIVOT 返回的值，引理 29.1 隐含着

$$\bar{x}_i=\begin{cases} b_i-a_{ie}\hat{b}_e & \text{若 } i\in\hat{B}-\{e\} \\ b_l/a_{le} & \text{若 } i=e \end{cases} \tag{29.116}$$

第 7 行中 PIVOT 的调用有 $e=0$，且根据式(29.110)，有

$$a_{i0} - a_{ie} = -1 \quad \text{对每个 } i \in B \tag{29.117}$$

(注意 a_{i0} 是 x_0 出现在式(29.110)时的系数，而不是此系数的负值，因为 L_{aux} 是在标准型中而不是在松弛型中)因为 $l \in B$，也有 $a_{le} = -1$。因此，$b_l/a_{le} > 0$，于是 $\bar{x}_e > 0$。对于剩下的基本
815 变量，有

$$\begin{aligned}\bar{x}_i &= b_i - a_{ie}\hat{b}_e \quad \text{(等式(29.116))} \\ &= b_i - a_{ie}(b_l/a_{le}) \quad \text{(PIOVT 的第 2 行)} \\ &= b_i - b_l \quad \text{(等式(29.117)和 } a_{le} = -1\text{)} \\ &\geqslant 0 \quad \text{(不等式(29.115))}\end{aligned}$$

这意味着现在每个基本变量都是非负的。因此在第 7 行调用 PIVOT 后的基本变量都是可行的。接下来执行第 9 行来解 L_{aux}。因为已经假设 L 有一个可行解，引理 29.11 暗示着 L_{aux} 有一个目标值为 0 的最优解。因为所有的松弛型都是等价的，L_{aux} 的最终基本解必定有 $\bar{x}_0 = 0$，而且当把 x_0 从线性规划中删除后，得到一个 L 的可行的松弛型。然后这个松弛型在第 10 行返回。 ■

线性规划的基本定理

下面，通过证明过程 SIMPLEX 确实有作用来结束本章。特别地，任何线性规划都可能是不可行的，或是无界的，或有一个有限目标值的最优解；在每种情况下，SIMPLEX 都能正确地起作用。

定理 29.13(线性规划的基本定理)　以标准型给出任意的线性规划 L 可以是以下三者之一：

1)有一个有限目标值的最优解。

2)不可行。

3)无界。

如果 L 是不可行的，SIMPLEX 返回“不可行”。如果 L 是无界的，SIMPLEX 返回“无界”。否则，SIMPLEX 返回一个有限目标值的最优解。

证明：根据引理 29.12，如果线性规划 L 是不可行的，则 SIMPLEX 返回“不可行”。现在假设线性规划 L 是可行的。根据引理 29.12，INITIALIZE-SIMPLEX 返回一个基本解可行的松弛型。因此，根据引理 29.7，SIMPLEX 或者返回“无界”，或者以一个可行解终止。如果它以一个可行解终止，则定理 29.10 告诉我们这个解是最优的。另一方面，如果 SIMPLEX 返回“无界”，则引理 29.2 告诉我们线性规划 L 的确是无界的。因为 SIMPLEX 永远以这些方式之中的一种来
816 终止，所以完成证明。 ■

练习

29.5-1　写出详细的伪代码以实现 INITIALIZE-SIMPLEX 的第 5 行和第 11 行。

29.5-2　证明：当 INITIALIZE-SIMPLEX 执行 SIMPLEX 的主循环时，永远不会返回“无界”。

29.5-3　假设已知一个标准型的线性规划 L，且假设对于 L 与 L 的对偶问题，其对应于初始松弛型的基本解都是可行的。证明：L 的最优目标值是 0。

29.5-4　假设在线性规划中允许严格的不等式。证明：在这种情况下，线性规划的基本定理不再成立。

29.5-5　用 SIMPLEX 解下列线性规划：

最大化　　$x_1 + 3x_2$

满足约束

$$\begin{aligned} -x_1 + x_2 &\leqslant -1 \\ -2x_1 - 2x_2 &\leqslant -6 \\ -x_1 + 4x_2 &\leqslant 2 \\ x_1, x_2 &\geqslant 0 \end{aligned}$$

29.5-6　解在式(29.6)～式(29.10)中给出的线性规划。

29.5-7　考虑下面被称作 P 的 1 个变量的线性规划：

最大化　　tx

满足约束

$$\begin{aligned} rx &\leqslant s \\ x &\geqslant 0 \end{aligned}$$

其中 r、s 和 t 是任意的实数。令 D 表示 P 的对偶问题。 817

陈述什么样值的 r、s 和 t 可以让

1)P 和 D 都是拥有有限目标值的最优解。

2)P 是可行的，但是 D 是不可行的。

3)D 是可行的，但是 P 是不可行的。

4)P 和 D 都是不可行的。

思考题

29-1　线性不等式的可行性

已知一个在 n 个变量 x_1，x_2，…，x_n 上的 m 个线性不等式的集合，线性不等式可行性问题询问是否有变量的一个设定能够同时满足每个不等式。

a)证明：如果有一个线性规划的算法，则可以利用它来解线性不等式可行性问题。在线性规划问题中用到的变量和约束的个数应该是 n 和 m 的多项式。

b)证明：如果有一个线性不等式可行性问题的算法，则可以用它来解线性规划问题。在线性不等式可行性问题中，用到的变量和线性不等式的个数应该是 n 和 m 的多项式，即线性规划中变量和约束的个数。

29-2　互补松弛性

互补松弛描述原变量的值与对偶约束、对偶变量的值与原约束之间的关系。令 $\bar{x}$ 表示式(29.16)至式(29.18)中给出的原线性规划的一个最优解，令 $\bar{y}$ 表示式(29.86)～式(29.88)中给出的对偶线性规划的最优解。互补松弛表明下面的条件是 $\bar{x}$ 和 $\bar{y}$ 为最优的充分必要条件：

$$\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j \quad \text{或} \quad \bar{x}_j = 0 \quad \text{对于 } j = 1,2,\cdots,n$$

和

$$\sum_{j=1}^{n} a_{ij}\bar{x}_j = b_i \quad \text{或} \quad \bar{y}_i = 0 \quad \text{对于 } i = 1,2,\cdots,m$$

818

a)对式(29.56)至式(29.60)行中的线性规划验证保持互补松弛性。

b)证明：对任意的原线性规划和它相应的对偶问题，保持互补松弛性。

c)证明：在式(29.16)至式(29.18)中所给出的原线性规划的一个可行解$\bar{x}$是最优的，当且仅当有值$\bar{y}=(\bar{y}_1, \bar{y}_2, \cdots, \bar{y}_m)$使得

- $\bar{y}$是式(29.86)～式(29.88)中给出的对偶线性规划的一个可行解。
- 每当$\bar{x}_j>0$时，有$\sum_{i=1}^{m} a_{ij}\bar{y}_i=c_j$。
- 每当$\sum_{j=1}^{n} a_{ij}\bar{x}_j<b_i$时，有$y_i=0$。

29-3　整数线性规划

一个整数线性规划问题是一个加上变量x必须在整数上取值的额外约束的线性规划问题。练习34.5-3说明仅确定一个整数线性规划是否有可行解是NP-难度的，这表示这个问题不大可能有一个多项式时间的算法。

a)证明弱对偶性(引理29.8)对整数线性规划成立。

b)证明对偶性(定理29.10)对整数线性规划不总是成立。

c)已知一个标准型的原线性规划，定义P为原线性规划的最优目标值，D为其对偶问题的最优目标值，IP为整数版本的原问题(亦即，原问题加上变量取整数值的约束)的最优目标值，ID为整数版本的对偶问题的最优目标值。证明$IP\leqslant P=D\leqslant ID$。

29-4　Farkas引理

819

令A为一个$m\times m$矩阵，b为一个m维向量。则Farkas引理说明正好有一个系统

$$Ax\leqslant 0,\quad bx>0$$

和

$$yA=b,\quad y\geqslant 0$$

是可解的，其中x是一个n维向量、y是一个m维向量。证明Farkas引理。

本章注记

本章只是刚开始研究线性规划的广阔领域。有许多书籍都对线性规划作了详细的讨论，包括Chvátal [62]、Gass [111]、Karloff [171]、Schrijver [266]和Vanderbei [304]。其他许多书籍中都包含了有关线性规划的很好的内容，包括Papadimitriou和Steiglitz [237]，Ahuja、Magnanti和Orlin [7]。本章所采用的是Chvátal的方法。

线性规划的单纯形算法是在1947年由G. Dantzig所发明的。不久之后，发现在各种领域中的很多问题都可以制定成线性规划，并且使用单纯形算法来解。这个发现导致了线性规划以及数个算法的广泛使用。单纯形算法的变种仍然是解线性规划问题的最受欢迎的方法。这段历史在很多地方都有描述，包括[62]和[171]中的注释。

椭圆法是线性规划的第一个多项式时间算法，在1979年由L. G. Khachian而来；它以N. Z. Shor、D. B. Judin和A. S. Nemirovskii的早期工作为基础。在组合优化中使用椭圆法来解各种问题在Grötschel、Lovász和Schrijver [134]中有描述。到目前为止，在实际中椭圆法还无法与单纯形算法相竞争。

Karmarkar的论文[172]包含了他的内点法的描述。在他之后，许多研究者都设计出了各种内点法。有关这方面的内容，在Goldfarb和Todd [122]的文章以及Ye [319]的书中都有好的综述。

单纯形算法的分析是研究界的一个活跃领域。Klee和Minty构造了单纯形算法执行2^n-1次迭代的一个例子。单纯形算法在实际中通常执行得非常好，而且许多研究者尝试给出这个实验观察

的理论推断。有一类研究是由 Borgwardt 开始，然后许多其他人也都在进行，它显示了在输入的某些概率假设下，单纯形算法在期望的指数时间内收敛。在这个领域中目前的进展由 Spielman 和 Teng [284]做出，他们引入“算法的平滑分析”并将它应用到单纯形算法上。 820

已知单纯形算法在某些特定情况下会执行得更有效。特别值得注意的是网络单纯形算法，即特定于网络流问题上的单纯形算法。对某些网络问题，包括最短路径、最大流和最小费用流问题，网络单纯形算法的变种可以在多项式时间内执行。参考诸如 Orlin [234]的文章以及里面的引用。 821

第 30 章　多项式与快速傅里叶变换

两个 n 次多项式相加的简单方法所需的时间为 $\Theta(n)$，而相乘的简单方法所需的时间为 $\Theta(n^2)$。在本章中，我们将讨论快速傅里叶变换(Fast Fourier Transform，FFT)方法是如何使多项式相乘的运行时间降低为 $\Theta(n\lg n)$的。

傅里叶变换的最常见用途是信号处理，这也是快速傅里叶变换的最常见用途。在时间域内给定的信号把时间映射到振幅的一个函数。傅里叶分析允许将信号表示成各种频率的相移正弦曲线的一个加权总和。和频率相关联的权重和相位在频率域中刻画出信号的特性。信号处理是一个有着丰富内容的领域，这一方面有不少很好的参考书；在本章末的"本章注记"中将列出其中一些书。

多项式

在一个代数域 F 上，关于变量 x 的多项式(polynomial)定义为形式和(formal sum)形式表示的函数 $A(x)$：

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

称值 a_0，a_1，…，a_{n-1}为多项式的系数。所有系数都属于域 F，典型的情况是复数集合 C。如果一个多项式 $A(x)$的最高次的非零系数为 a_k，则称 $A(x)$的次数(degree)是 k。任何严格大于一个多项式次数的整数都是这个多项式的次数界(degree-bound)。因此，对于次数界为 n 的多项式来说，其次数可以是 0 到 $n-1$ 之间的任何整数，也包括 0 和 $n-1$ 在内。

在多项式上可以定义各种运算。在多项式加法中，如果 $A(x)$和 $B(x)$是次数界为 n 的多项式，那么它们的和也是一个次数界为 n 的多项式 $C(x)$，并满足对所有属于定义域的 x，都有
822 $C(x)=A(x)+B(x)$。就是说，如果

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

并且

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

则

$$C(x) = \sum_{j=0}^{n-1} c_j x^j$$

其中 $c_j=a_j+b_j$，$j=0$，1，…，$n-1$。例如，如果 $A(x)=6x^3+7x^2-10x+9$，$B(x)=-2x^3+4x-5$，则 $C(x)=4x^3+7x^2-6x+4$。

在多项式乘法中，如果 $A(x)$和 $B(x)$都是次数界为 n 的多项式，则说它们的乘积是一个次数界为 $2n-1$ 的多项式积 $C(x)$，并满足对所有属于定义域的 x，都有 $C(x)=A(x)B(x)$。读者可能以前也学过多项式乘法，其方法是把 $A(x)$中的每一项与 $B(x)$中的每一项相乘，然后再把同类项合并。例如，可以通过如下方式，对两个多项式 $A(x)=6x^3+7x^2-10x+9$ 和 $B(x)=-2x^3+4x-5$ 进行乘法运算：

$$
\begin{array}{r}
6x^3 + \quad 7x^2 - \ 10x + 9 \\
\underline{-\ 2x^3 \qquad\quad + \ 4x - \ 5} \\
-\ 30x^3 - 35x^2 + 50x - 45 \\
24x^4 + 28x^3 - 40x^2 + 36x \\
\underline{-\ 12x^6 - 14x^5 + 20x^4 - 18x^3 \qquad\qquad} \\
-\ 12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45
\end{array}
$$

另一种表示积 $C(x)$的方法是：

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \tag{30.1}$$

其中

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} \tag{30.2}$$

注意 degree(C) = degree(A) + degree(B)蕴含

$$\begin{aligned}\text{degree-bound}(C) &= \text{degree-bound}(A) + \text{degree-bound}(B) - 1 \\ &\leqslant \text{degree-bound}(A) + \text{degree-bound}(B)\end{aligned}$$

但是，我们不说 C 的次数界为 A 的次数界与 B 的次数界的和，这是因为如果一个多项式的次数界为 k，也可以说该多项式的次数界为 $k+1$。

823

本章概述

30.1 节介绍两种表示多项式的方法，即系数表示法和点值表示法。当用系数表示法表示多项式时，多项式(如式(30.1)和式(30.2))乘法的简单算法所需的运行时间为 $\Theta(n^2)$，但采用点值表示法后，其运行时间仅为 $\Theta(n)$。不过，通过对这两种表示法进行转换，采用系数表示法来求两个多项式的乘积，可以使运行时间变为 $\Theta(n \lg n)$。为了弄清这种方法为什么可行，我们将在 30.2 节中学习单位复根。然后，运用 FFT 和它的逆变换(也在 30.2 节中介绍)来执行上述转换。30.3 节讨论了如何在串行模型与并行模型上快速实现 FFT。

本章中大量使用了复数，符号 i 专用于表示 $\sqrt{-1}$。

30.1　多项式的表示

从某种意义上说，多项式的系数表示法与点值表示法是等价的，即用点值形式表示的多项式都对应唯一一个系数形式的多项式。在本节中，将介绍这两种表示方法，并阐述如何把这两种表示结合起来，从而使这两个次数界为 n 的多项式乘法运算在 $\Theta(n \lg n)$时间内完成。

系数表示法

对一个次数界为 n 的多项式 $A(x) = \sum_{j=0}^{n-1} a_j x^j$ 来说，其*系数表示法*(coefficient representation)就是一个由系数组成的向量 $a = (a_0, a_1, \cdots, a_{n-1})$。在本章所涉及的矩阵方程中，一般将它作为列向量看待。

采用系数表示法对于某些多项式的运算是很方便的。例如，对多项式 $A(x)$在给定点 x_0 的求值运算就是计算 $A(x_0)$的值。如果使用霍纳(Horner)法则，则求值运算的运行时间为 $\Theta(n)$：

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\cdots))$$

类似地，对两个分别用系数向量 $a = (a_0, a_1, \cdots, a_{n-1})$和 $b = (b_0, b_1, \cdots, b_{n-1})$表示的多项式进行相加时，所需的时间是 $\Theta(n)$：仅输出系数向量 $c = (c_0, c_1, \cdots, c_{n-1})$，其中对 $j =$

0，1，…，$n-1$，有 $c_j=a_j+b_j$。

现在来考虑两个用系数形式表示的、次数界为 n 的多项式 $A(x)$ 和 $B(x)$ 的乘法运算，如果用
824 式(30.1)和式(30.2)所描述的方法，完成多项式乘法所需要的时间就是 $\Theta(n^2)$，因为向量 a 中的每个系数必须与向量 b 中的每个系数相乘。当用系数形式表示时，多项式乘法运算似乎要比求多项式的值和多项式加法困难得多。由式(30.2)给出的结果系数向量 c 也称为输入向量 a 和 b 的卷积(convolution)，表示成 $c=a\otimes b$。因为多项式乘法与卷积的计算都是最基本的计算问题，在实践中非常重要，所以本章将重点讨论有关的高效算法。

点值表示法

一个次数界为 n 的多项式 $A(x)$ 的点值表示(point-value representation)就是 n 个点值对所形成的集合：

$$\{(x_0,y_0),(x_1,y_1),\cdots,(x_{n-1},y_{n-1})\}$$

其中所有 x_k 各不相同，并且当 $k=0,1,\cdots,n-1$ 时有

$$y_k = A(x_k) \tag{30.3}$$

一个多项式可以有很多不同的点值表示，这是由于任意 n 个相异点 x_0，x_1，…，x_{n-1} 组成的集合，都可以作为这种表示法的基础。

对于一个用系数形式表示的多项式来说，在原则上计算其点值表示是简单易行的，因为我们所要做的就是选取 n 个相异点 x_0，x_1，…，x_{n-1}，然后对 $k=0,1,\cdots,n-1$，求出 $A(x_k)$。根据霍纳法则，求出这 n 个点的值所需要的时间为 $\Theta(n^2)$。在稍后可以看到，如果巧妙地选取 x_k 的话，就可以加速这一计算过程，使其运行时间变为 $\Theta(n\lg n)$。

求值计算的逆(从一个多项式的点值表示确定其系数表示中的系数)称为插值(interpolation)。下列定理说明插值具有良定义，假设插值多项式的次数界等于已知的点值对的数目。

定理 30.1(多项式插值的唯一性)　对于任意 n 个点值对组成的集合$\{(x_0,y_0),(x_1,y_1),\cdots,(x_{n-1},y_{n-1})\}$，存在唯一的次数界为 n 的多项式 $A(x)$，满足 $y_k=A(x_k)$，$k=0,1,\cdots,n-1$。

825 **证明**：证明过程的基础是某个矩阵存在逆矩阵。式(30.3)等价于矩阵方程

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \cdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \cdots \\ y_{n-1} \end{pmatrix} \tag{30.4}$$

左边的矩阵是范德蒙德矩阵，表示为 $V(x_0,x_1,\cdots,x_{n-1})$。根据练习 28.1-11，该矩阵的行列式的值为

$$\prod_{0\le j<k\le n-1}(x_k-x_j)$$

因此，由定理 28.5 可知，如果 x_k 相异，则该矩阵是可逆的(即非奇异的)。因此，对给定的唯一的点值表示，就能够求出系数 a_j 的解：

$$a = V(x_0,x_1,\cdots,x_{n-1})^{-1}y$$ ■

定理 30.1 的证明过程同时描述了基于求解线性方程组(式(30.4))的一种插值算法。利用第 28 章描述的 LU 分解算法，可以在 $O(n^3)$ 的运行时间内求出该线性方程组的解。

要对 n 个点进行插值，还可以用另一种更快的算法，它是基于以下拉格朗日公式的：

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j\ne k}(x-x_j)}{\prod_{j\ne k}(x_k-x_j)} \tag{30.5}$$

读者可以验证一下等式(30.5)的右端是一个次数界为 n 的多项式，并满足对所有 k，$A(x_k)=y_k$。练习 30.1-5 要求读者说明如何在 $\Theta(n^2)$ 的运行时间内，运用拉格朗日公式来计算 A 的所有系数。

因此，n 个点的求值运算与插值运算是良定义的互逆运算，它们将多项式的系数表示与点值表示进行相互转化[⊖]。关于这些问题，上述算法的运行时间为 $\Theta(n^2)$。

对许多关于多项式的操作来说，点值表示法是很方便的。对于加法来说，如果 $C(x)=A(x)+B(x)$，则对任意点 x_k，有 $C(x_k)=A(x_k)+B(x_k)$。更准确地说，如果已知 A 的点值表示：

$$\{(x_0,y_0),(x_1,y_1),\cdots,(x_{n-1},y_{n-1})\}$$

826

和 B 的点值表示：

$$\{(x_0,y_0'),(x_1,y_1'),\cdots,(x_{n-1},y_{n-1}')\}$$

(注意，A 和 B 对相同的 n 个点进行求值)，则 C 的点值表示为

$$\{(x_0,y_0+y_0'),(x_1,y_1+y_1'),\cdots,(x_{n-1},y_{n-1}+y_{n-1}')\}$$

因此，对两个点值形式表示的次数界为 n 的多项式相加，所需时间为 $\Theta(n)$。

类似地，点值表示法对于多项式乘法也是方便的。如果 $C(x)=A(x)B(x)$，则对任意点 x_k，有 $C(x_k)=A(x_k)B(x_k)$，并且对 A 的点值表示和 B 的点值表示进行逐点相乘，就可以得到 C 的点值表示。不过，我们也必须面对这样一个问题，即 C 的次数界是 A 的次数界与 B 的次数界的和。A 和 B 的标准点值表示法是由每个多项式的 n 个点值对所组成的。把这些点值对相乘，就得到 C 的 n 个点值对，但是由于 C 的次数界为 $2n$，因此要获得 C 的点值表示需要 $2n$ 个点值对(参见练习 30.1-4。)因此，必须对 A 和 B 的点值表示进行“扩充”，使每个多项式都包含 $2n$ 个点值对。如果已知 A 的扩充点值表示：

$$\{(x_0,y_0),(x_1,y_1),\cdots,(x_{2n-1},y_{2n-1})\}$$

和 B 的相应扩充点值表示：

$$\{(x_0,y_0'),(x_1,y_1'),\cdots,(x_{2n-1},y_{2n-1}')\}$$

则 C 的点值表示为：

$$\{(x_0,y_0y_0'),(x_1,y_1y_1'),\cdots,(x_{2n-1},y_{2n-1}y_{2n-1}')\}$$

如果已知两个扩充点值形式的输入多项式，使其相乘而得到点值形式的结果需要 $\Theta(n)$ 的时间，这要比采用系数形式的两个多项式相乘所需的时间少得多。

最后来考虑对一个点值表示的多项式，如何求其在某个新点上的值这一问题。对这个问题来说，最简单不过的方法显然就是先把该多项式转化为其系数形式，然后再求其在新点处的值。

系数形式表示的多项式的快速乘法

能否利用关于点值形式表示的多项式的线性时间乘法算法，来加快系数形式表示的多项式乘法运算的速度呢？答案依赖于能否快速把一个多项式从系数形式转换为点值形式(求值)，和从点值形式转换为系数形式(插值)。

827

可以采用需要的任何点作为求值点，但经过精心地挑选求值点，可以把两种表示法之间转化所需的时间压缩为 $\Theta(n\lg n)$。如将在 30.2 节看到的那样，如果选择“单位复根”作为求值点，则可以通过对系数向量进行离散傅里叶变换(或 DFT)，得到相应的点值表示。同样，也可以通过对点值对执行“逆 DFT”运算，而获得相应的系数向量，这样就完成了求值运算的逆运算——插值。30.2 节将说明 FFT 如何在 $\Theta(n\lg n)$ 的时间内执行 DFT 和逆 DFT 运算。

⊖ 从数值稳定性的角度看，插值是一个众所周知的难处理的问题。虽然这里所描述的方法在数学上是正确的，但是在计算期间输入的小误差，或是四舍五入的误差都可能造成结果的大差异。

图 30-1 用图的方式说明了这种策略，其中涉及了次数界问题。两个次数界为 n 的多项式的积是一个次数界为 $2n$ 的多项式。因此，在对输入多项式 A 和 B 进行求值之前，首先通过增加 n 个值为 0 的高阶系数，使其次数界增加到 $2n$。因为向量包含 $2n$ 个元素，所以用到了“$2n$ 次单位复根”，它在图 30-1 中由项 ω_{2n} 表示。

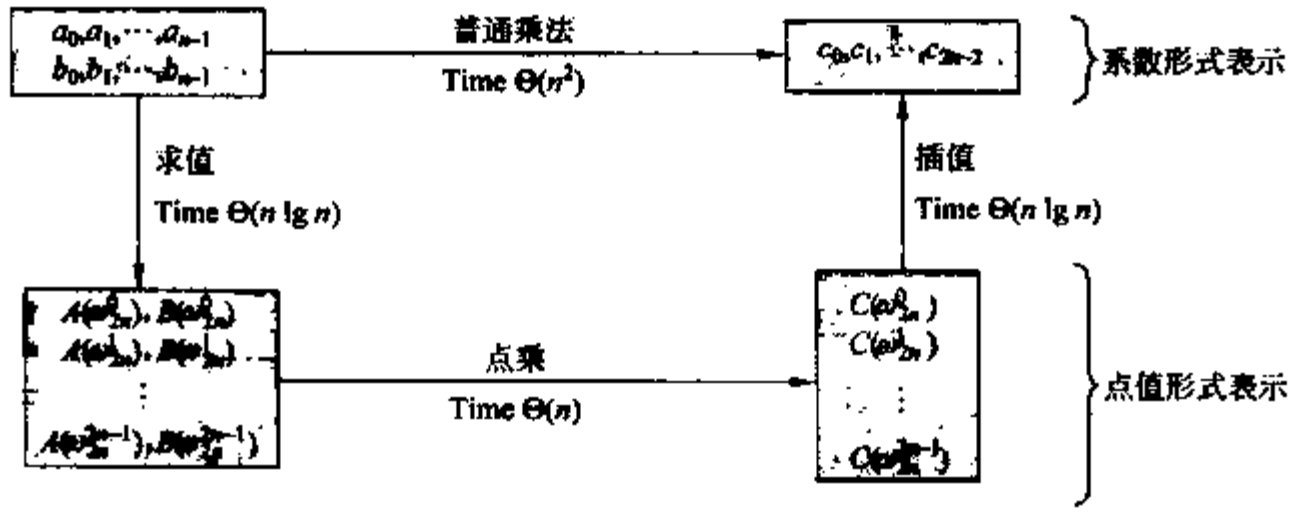


图 30-1　一种高效的多项式乘法过程的图示。上方代表系数形式，下方代表点值形式。从左到右的箭头对应于乘法操作。项 ω_{2n} 是 $2n$ 次单位复根

如果已知 FFT，就有下列运行时间为 $\Theta(n\lg n)$ 的过程，该过程把两个次数界为 n 的多项式 $A(x)$ 和 $B(x)$ 进行乘法运算，其中输入与输出均采用系数表示法。假定 n 为 2 的幂，通过加入值为 0 的高阶系数，这个要求总能被满足。

1)使次数界增加一倍：通过加入 n 个值为 0 的高阶系数，把多项式 $A(x)$ 和 $B(x)$ 扩充为次数界为 $2n$ 的多项式并构造其系数表示。

2)求值：两次应用 $2n$ 阶的 FFT 计算出 $A(x)$ 和 $B(x)$ 的长度为 $2n$ 的点值表示。这两个点值
828 表示中包含了两个多项式在 $2n$ 次单位根处的值。

3)点乘：把 $A(x)$ 的值与 $B(x)$ 的值逐点相乘，就可以计算出多项式 $C(x)=A(x)B(x)$ 的点值表示，这个表示中包含了 $C(x)$ 在每个 $2n$ 次单位根处的值。

4)插值：只要对 $2n$ 个点值对应用一次 FFT 以计算出其逆 DFT，就可以构造出多项式 $C(x)$ 的系数表示。

执行第 1 步和第 3 步所需时间为 $\Theta(n)$，执行第 2 步和第 4 步所需时间为 $\Theta(n\lg n)$。因此，一旦说明了如何运用 FFT，就已经证明了下列定理。

定理 30.2　当输入与输出都采用系数形式来表示多项式时，就能够在 $\Theta(n\lg n)$ 的时间内，计算出两个次数界为 n 的多项式的积。 ■

练习

30.1-1　运用式(30.1)和式(30.2)把下列两个多项式相乘：$A(x)=7x^3-x^2+x-10$，$B(x)=8x^3-6x+3$。

30.1-2　求一个次数界为 n 的多项式 $A(x)$ 在某已知点 x_0 的值也可以用以下方法获得：把多项式 $A(x)$ 除以多项式 $(x-x_0)$，得到一个次数界为 $n-1$ 的商多项式 $q(x)$ 和余项 r，并满足

$$A(x)=q(x)(x-x_0)+r$$

显然 $A(x_0)=r$。试说明如何根据 x_0 和 A 的系数，在 $\Theta(n)$ 的时间内计算出余项 r 以及 $q(x)$ 中的系数。

30.1-3　根据 $A(x)=\sum_{j=0}^{n-1} a_j x^j$ 的点值表示推导出 $A^{rev}(x)=\sum_{j=0}^{n-1} a_{n-1-j}x^j$ 的点值表示，假定没有一个点是 0。

30.1-4　证明：为了唯一确定一个次数界为 n 的多项式，n 个相互不同的点值对是必需的，也就是说，如果给定少于 n 对不同的点值，它们就无法唯一确定一个次数界为 n 的多项式。(提示：利用定理 30.1，加入一个任意选择的点值对到一个有 $n-1$ 个点值对的集合，你对这个集合有什么看法?)

829

30.1-5　说明如何使用等式(30.5)在 $\Theta(n^2)$ 的时间内进行插值运算。(提示：先计算多项式 $\prod_j (x-x_j)$ 的系数表示，然后把每个项的分子除以 $(x-x_k)$；参见练习 30.1-2。n 个分母中的每一个可以在 $O(n)$ 时间内计算。)

30.1-6　试解释在采用点值表示法时，用“显然”的方法来进行多项式除法错误在何处，即除以相应的 y 值。试对除法有确切结果与无确切结果两种情况分别进行讨论。

30.1-7　考察两个集合 A 和 B，每个集合包含取值范围在 0 到 $10n$ 之间的 n 个整数。要计算出 A 与 B 的笛卡儿和，它的定义如下：

$$C=\{x+y: x\in A \text{ 且 } y\in B\}$$

注意，C 中整数的取值范围在 0 到 $20n$ 之间。我们希望计算出 C 中的元素，并且求出 C 的每个元素可为 A 与 B 中元素和的次数。证明：解决这个问题需要 $\Theta(n\lg n)$ 的时间。(提示：用 $10n$ 次多项式来表示 A 和 B。)

30.2　DFT 与 FFT

在 30.1 节中说过，如果使用单位复根的话，就可以在 $\Theta(n\lg n)$ 时间内完成求值与插值运算。在本节中，要给出单位复根的定义及其性质，定义 DFT，并说明 FFT 如何仅用 $\Theta(n\lg n)$ 的时间，就可以计算出 DFT 与它的逆。

单位复根

n 次单位复根是满足 $\omega^n=1$ 的复数 ω。n 次单位复根刚好有 n 个，它们是 $e^{2\pi ik/n}$，$k=0,1,\cdots,n-1$。为了解释这一式子，我们利用复数的幂的定义：

$$e^{iu}=\cos(u)+i\sin(u)$$

图 30-2 说明 n 个单位复根均匀地分布在以复平面的原点为圆心的单位半径的圆周上。值

$$\omega_n=e^{2\pi i/n} \qquad (30.6)$$

称为主 n 次单位根；所有的其他 n 次单位复根都是 ω^n 的幂[⊖]。

n 个 n 次单位复根 $\omega_n^0,\omega_n^1,\cdots,\omega_n^{n-1}$ 在乘法运算下形成了一个群(参见 31.3 节)。该群的结构与加法群 $(\mathbb{Z}_n,+)$ 模 n 相同，这是因为 $\omega_n^n=\omega_n^0=1$ 蕴含着 $\omega_n^j\omega_n^k=\omega_n^{j+k}=\omega_n^{(j+k)\bmod n}$。类似地有 $\omega_n^{-1}=\omega_n^{n-1}$。

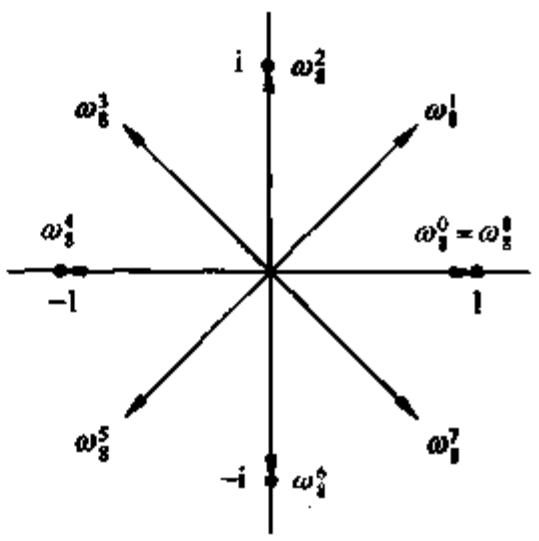


图 30-2　在复平面上 $\omega_8^0,\omega_8^1,\cdots,\omega_8^7$ 的值，其中 $\omega_8=e^{2\pi i/8}$ 是主 8 次单位根

830

⊖　许多其他作者对 ω^n 有不同的定义：$\omega^n=e^{-2\pi i/n}$。这个定义一般用在信号处理应用中，不管是 ω^n 的哪一种定义，其背后的数学含义基本上是相同的。

下面的引理给出了 n 次单位复根的重要性质。

引理 30.3(相消引理)　对任何整数 $n \geqslant 0$，$k \geqslant 0$，$d > 0$，

$$\omega_{dn}^{dk} = \omega_n^k \tag{30.7}$$

证明：由式(30.6)可直接推得引理成立，因为

831

$$\omega_{dn}^{dk} = (e^{2\pi i/dn})^{dk} = (e^{2\pi i/n})^k = \omega_n^k$$ ■

推论 30.4　对任意偶数 $n > 0$，有

$$\omega_n^{n/2} = \omega_2 = -1$$

证明：证明过程留作练习 30.2-1。 ■

引理 30.5(折半引理)　如果 $n > 0$ 为偶数，n 个 n 次单位复根的平方等于 $n/2$ 个 $n/2$ 次单位复根。

证明：根据相消引理，对任意非负整数 k，有 $(\omega_n^k)^2 = \omega_{n/2}^k$。注意，如果对所有 n 次单位复根进行平方，每个 $n/2$ 次单位根要刚好获得两次，因为

$$(\omega_n^{k+n/2})^2 = \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = \omega_n^{2k} = (\omega_n^k)^2$$

因此，ω_n^k 与 $\omega_n^{k+n/2}$ 的平方值相等。这条性质也可以由推论 30.4 来证明，因为 $\omega^{n/2} = -1$ 说明 $\omega_n^{k+n/2} = -\omega_n^k$，所以 $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$。 ■

我们将会看到，折半引理对于运用分治法来对多项式的系数与点值表示进行相互转换是非常重要的，这是因为它能够保证递归的子问题的规模是递归调用前的一半。

引理 30.6(求和引理)　对任意整数 $n \geqslant 1$ 和不能被 n 整除的非负整数 k，有

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

832　**证明**：因为等式(A.5)除了适用于实数，也适用于复数，故有

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} = \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} = \frac{(1)^k - 1}{\omega_n^k - 1} = 0$$

要求 k 不能被 n 整除就可以保证分母不为 0，因为只有当 k 能被 n 整除时才有 $\omega_n^k = 1$。 ■

DFT

回顾一下我们希望计算次数界为 n 的多项式

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

在 ω_n^0，ω_n^1，…，ω_n^{n-1} 处的值(即在 n 个 n 次单位复根处)[⊖]。不失一般性地假定 n 是 2 的幂，因为给定的次数界总可以增大；如果需要，总可以添加值为零的新的高阶系数[⊜]。假定已知 A 的系数形式：$a = (a_0, a_1, \cdots, a_{n-1})$。对 $k = 0, 1, \cdots, n-1$，定义结果 y_k 如下：

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj} \tag{30.8}$$

向量 $y = (y_0, y_1, \cdots, y_{n-1})$ 是系数向量 $a = (a_0, a_1, \cdots, a_{n-1})$ 的离散傅里叶变换

833　(Discrete Fourier Transform，DFT)，也写作 $y = \mathrm{DFT}_n(a)$。

⊖　长度 n 实际上就是在 30.1 节所指的 $2n$，因为在求值之前把给定多项式的次数加倍，因此在多项式乘法的上下文中，实际上是使用 $2n$ 次单位复根。

⊜　当在信号处理中利用 FFT 时，添加零系数以达到 2 的幂大小通常是不好的，因为它会引入高频率的噪声。在信号处理中加零来达到 2 的幂大小所用的一个技术是镜像。设 n' 是大于 n 的最小的 2 的整数次方，镜像的一个方式是令 $a_{n+j} = a_{n-j-1}$ $(j = 0, 1, \cdots, n'-n-1)$。

FFT

通过使用一种称为快速傅里叶变换(FFT)的方法，就可以在 $\Theta(n \lg n)$ 的时间内计算出 $\mathrm{DFT}_n(a)$，而直接的方法所需时间为 $\Theta(n^2)$。FFT 主要是利用了单位复根的特殊性质。

FFT 方法运用了分治策略，它用 $A(x)$ 中偶数下标的系数与奇数下标的系数，分别定义了两个新的次数界为 $n/2$ 的多项式 $A^{[0]}(x)$ 和 $A^{[1]}(x)$：

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$$

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$

注意，$A^{[0]}$ 包含 A 中所有偶数下标的系数(下标的相应二进制数的最后一位为 0)，而 $A^{[1]}$ 包含 A 中所有奇数下标的系数(下标的相应二进制数的最后一位为 1)。有下式

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \tag{30.9}$$

这样，求 $A(x)$ 在 ω_n^0，ω_n^1，$\cdots$，ω_n^{n-1} 处的值的问题就转换为：

1)求次数界为 $n/2$ 的多项式 $A^{[0]}$ 与 $A^{[1]}$ 在点

$$(\omega_n^0)^2, (\omega_n^1)^2, \cdots, (\omega_n^{n-1})^2 \tag{30.10}$$

的值，然后

2)根据式(30.9)把上述结果进行组合。

根据折半引理，值的序列(30.10)并不是由 n 个不同的值组成的，而是仅由 $n/2$ 个 $n/2$ 次单位复根所组成，每个根均出现两次。因此次数界为 $n/2$ 的多项式 $A^{[0]}$ 和 $A^{[1]}$ 递归地在 $n/2$ 个 $n/2$ 次单位复根处进行求值。这些子问题与原始问题形式相同，但规模缩小一半。现在，我们已经成功地把一个 n 个元素的 DFT_n 计算过程划分为两个 $n/2$ 个元素的 $\mathrm{DFT}_{n/2}$ 计算过程。这一分解是下列递归的 FFT 算法的基础，该算法计算出一个由 n 个元素组成的向量 $a=(a_0, a_1, \cdots, a_{n-1})$ 的 DFT，其中 n 是 2 的幂。 834

```
RECURSIVE-FFT(a)
 1  n ← length[a]                ▷n 是 2 的幂
 2  if n=1
 3       then return a
 4  ω_n ← e^{2πi/n}
 5  ω ← 1
 6  a^[0] ←(a_0, a_2, ···, a_{n-2})
 7  a^[1] ←(a_1, a_3, ···, a_{n-1})
 8  y^[0] ←RECURSIVE-FFT(a^[0])
 9  y^[1] ←RECURSIVE-FFT(a^[1])
10  for k ← 0 to n/2−1
11       do y_k ← y_k^[0] + ω y_k^[1]
12            y_{k+(n/2)} ← y_k^[0] − ω y_k^[1]
13            ω ← ω ω_n
14  return y                     ▷y 是一个列向量
```

RECURSIVE-FFT 的执行过程如下。第 2～3 行代表递归的基础：一个元素的 DFT 就是该元素自身，因为在这种情况下

$$y_0 = a_0 \omega_1^0 = a_0 \cdot 1 = a_0$$

第 6～7 行定义了多项式 $A^{[0]}$ 和 $A^{[1]}$ 的系数向量。第 4、5 和 13 行保证可以对 ω 进行适当的更新，以便每当第 11～12 行被执行时，有 $\omega=\omega_n^k$(在一次次迭代中使 ω 的值不断变化可以节约每次通过 for 循环从零开始计算 ω_n^k 的时间)。第 8～9 行执行了递归计算 $\mathrm{DFT}_{n/2}$ 的过程。对 $k=0$，

1，…，$n/2-1$，置：

$$y_k^{[0]} = A^{[0]}(\omega_{n/2}^k)$$

$$y_k^{[1]} = A^{[1]}(\omega_{n/2}^k)$$

或者，根据相消引理有 $\omega_{n/2}^k = \omega_n^{2k}$，有：

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k})$$

$$y_k^{[1]} = A^{[1]}(\omega_n^{2k})$$

第 11～12 行把递归计算 $\mathrm{DFT}_{n/2}$ 所得的结果进行组合。对 y_0，y_1，…，$y_{n/2-1}$，执行第 11 行得到：

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \quad \text{（由式(30.9)可得）} \end{aligned}$$

835

对 $y_{n/2}$，$y_{n/2+1}$，…，y_{n-1}，设 $k=0$，1，…，$n/2-1$，执行第 12 行得到：

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_k^{k+(n/2)} y_k^{[1]} \quad \text{（由于 } \omega_n^{k+(n/2)} = -\omega_n^k\text{）} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \quad \text{（由于 } \omega_n^{2k+n} = \omega_n^{2k}\text{）} \\ &= A(\omega_n^{k+(n/2)}) \quad \text{（由式(30.9)可得）} \end{aligned}$$

因此，由过程 RECURSIVE-FFT 所返回的向量 y 确实是输入向量 a 的 DFT。

在第 10～13 行的 for 循环中，每个值 $y_k^{[1]}$ 被乘以 ω_n^k，$k=0$，1，…，$n/2-1$。这个乘积既被加到 $y_k^{[0]}$ 上，又被从 $y_k^{[0]}$ 中减去。因为每个因子 ω_n^k 同时用在它的正数与负数形式中，所以因子 ω_n^k 叫做旋转因子(twiddle factor)。

为了确定过程 RECURSIVE-FFT 的运行时间，注意除了递归调用外，每条命令执行所需的时间为 $\Theta(n)$，n 为输入向量的长度。因此，关于运行时间有下列递归式：

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$$

因此，运用快速傅里叶变换，可以在 $\Theta(n \lg n)$的时间内，求出次数界为 n 的多项式在 n 次单位复根处的值。

对单位复根进行插值

现在来说明如何在单位复根处进行插值，以便把一个多项式从点值表示转化成系数表示，进而完成多项式乘法方案。按如下方式进行插值：把 DFT 写成一个矩阵方程，然后再检查其逆矩阵的形式。

根据式(30.4)，可以把 DFT 写成矩阵积 $y = V_n a$，其中 V_n 是由 ω_n 的适当幂组成的一个范德蒙德矩阵：

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

对 j，$k=0$，1，…，$n-1$，V_n 的$(k,\ j)$处的元素为 ω_n^{kj}，并且 V_n 中元素的幂形成一张乘法运算表。

对于逆运算 $a=\mathrm{DFT}_n^{-1}(y)$，通过把 y 乘以逆矩阵 V_n^{-1} 来进行处理。

定理 30.7　对 j，$k=0$，1，…，$n-1$，V_n^{-1} 的$(j,\ k)$处的元素为 ω_n^{-kj}/n。

证明：我们证明 $V_n^{-1}V_n=I_n$(即 $n\times n$ 单位矩阵)。考察 $V_n^{-1}V_n$ 中的元素$(j,\ j')$：

$$[V_n^{-1}V_n]_{jj'}=\sum_{k=0}^{n-1}(\omega_n^{-kj}/n)(\omega_n^{kj'})=\sum_{k=0}^{n-1}\omega_n^{k(j'-j)}/n$$

如果 $j=j'$，则和式的值为 1，由求和引理(引理 30.6)可知在其他情况下和式的值为 0。注意，我们依赖于 $-(n-1)<j'-j<n-1$，以便 $j'-j$ 不能被 n 整除，这样才能应用求和引理。■

在求出逆矩阵 V_n^{-1} 后，$\mathrm{DFT}_n^{-1}(y)$可由下式给出：

$$a_j=\frac{1}{n}\sum_{k=0}^{n-1}y_k\omega_n^{-kj} \tag{30.11}$$

其中 $j=0$，1，…，$n-1$。通过比较式(30.8)与式(30.11)，我们发现通过对 FFT 算法进行如下修改：把 a 与 y 的角色互换，用 ω_n^{-1} 来代替 ω_n，并且将每个结果元素除以 n，就可以计算出逆 DFT(参见练习 30.2-4)。因此，同样可以在 $\Theta(n\lg n)$的时间内计算出 DFT_n^{-1}。

于是，通过运用 FFT 与逆 FFT，就可以在 $\Theta(n\lg n)$的时间内，把次数界为 n 的多项式在其系数表示与点值表示之间来回进行转换。这样，在矩阵乘法的上下文中，我们已经证明了下列结论。

定理 30.8(卷积定理)　对任意两个长度为 n 的向量 a 和 b，其中 n 是 2 的幂，

$$a\otimes b=\mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a)\cdot\mathrm{DFT}_{2n}(b))$$

其中向量 a 和 b 用 0 扩充使其长度达到 $2n$，“$\cdot$”表示 2 个 $2n$ 个元素组成的向量的点乘。■

练习

30.2-1　证明推论 30.4。

30.2-2　计算向量(0，1，2，3)的 DFT。

30.2-3　使用运行时间为 $\Theta(n\lg n)$的方案重做练习 30.1-1。

30.2-4　写出在 $\Theta(n\lg n)$的运行时间内计算出 DFT_n^{-1} 的伪代码。

30.2-5　试把 FFT 过程推广到 n 是 3 的幂的情形，写出其运行时间的递归式并求解该式。

***30.2-6**　假定我们不是在复数域上执行 n 个元素的 FFT(n 为偶数)，而是在整数模 m 所生成的环 Z_m 上执行 FFT，其中 $m=2^{tn/2}+1$，并且 t 是任意正整数。对模 m，用 $\omega=2^t$ 来代替 ω_n 作为主 n 次单位根。证明：在该系统中 DFT 与逆 DFT 有良定义。

30.2-7　已知一组值 z_0，z_1，…，z_{n-1}(可能有重复)，说明如何求出仅在 z_0，z_1，…，z_{n-1} 处(可能有重复)值为 0 的次数界为 n 的多项式 $P(x)$的系数。所给出的过程的运行时间应该是 $O(n\lg^2 n)$。(提示：当且仅当 $P(x)$是$(x-z_j)$的倍数时，多项式 $P(x)$在 z_j 处值为 0。)

***30.2-8**　向量 $a=(a_0,a_1,\cdots,a_{n-1})$ 的 chirp 变换是向量 $y=(y_0,y_1,\cdots,y_{n-1})$，其中 $y_k=\sum_{j=0}^{n-1}a_jz^{kj}$，$z$ 是任意复数。因此，DFT 是线性调频变换的一种特殊情况($z=\omega_n$)。证明：对任意复数 z，可以在 $O(n\lg n)$ 的时间内求出线性调频变换的值。(提示：利用等式

$$y_k=z^{k^2/2}\sum_{j=0}^{n-1}(a_jz^{j^2/2})(z^{-(k-j)^2/2})$$

838　可以把线性调频变换看作为卷积。)

30.3　有效的 FFT 实现

由于 DFT 的实际应用(如信号处理)需要极高的速度，所以本节将讨论两种有效的 FFT 实现方法。首先来讨论一种运行时间为 $\Theta(n\lg n)$ 的 FFT 算法的迭代实现方法。不过，在其运行时间的 Θ 记号中，隐含的常数要比 30.2 节中论述的递归实现方法中的常数小。然后，我们将深入分析迭代实现方法，设计出一个有效的并行 FFT 电路。

FFT 的一种迭代实现

在过程 RECURSIVE-FFT 中，第 10～16 行的 for 循环中两次计算了值 $\omega_n^k y_k^{[1]}$。在编译术语中，该值称为公用子表达式。我们可以改变循环使其仅计算一次，并把它存放在临时变量 t 中。

```
for k ← 0 to n/2−1
    do t ← ω y_k^[1]
       y_k ← y_k^[0] + t
       y_{k+(n/2)} ← y_k^[0] − t
       ω ← ω ω_n
```

在这个循环中，把 $\omega=\omega_n^k$ 乘以 $y_k^{[1]}$，把所得的积存入 t 中，然后从 $y_k^{[0]}$ 中增加 t 和减去 t，整个这一串操作称为一个蝴蝶操作(bufferfly operation)，图 30-3 说明其执行步骤。

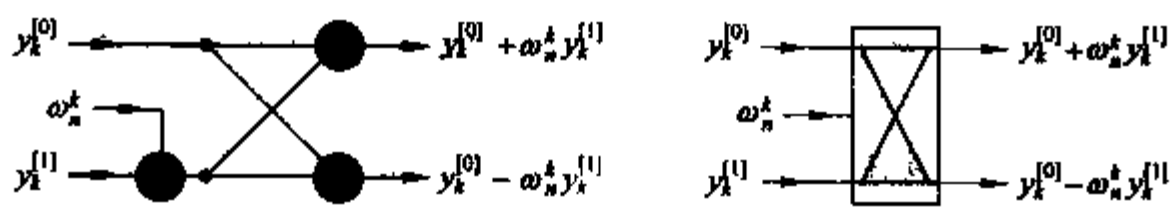


图 30-3　蝴蝶操作。a)两个输入向量从左边进入，旋转因子 ω_n^k 乘以 $y_k^{[1]}$，其和与差在右边输出。b)蝴蝶操作的一个简化图形。我们将在一个并行 FFT 电路中使用这个表示方式

现在来说明如何使 FFT 采用迭代结构而不是递归结构。在图 30-4 中，对一次 RECURSIVE-FFT 调用请求中所发出的各次递归调用，将其输入向量安排成一种树形结构，在初始调用时有 $n=8$。树中的每一个结点相应于每次对递归的调用，由对应的输入向量进行标记。每次调用 RECURSIVE-FFT 都同时产生两个递归调用，除非该过程接收到了 1 个元素组成的向量。我们把其第一次调用作为左子女，第二次调用作为右子女。

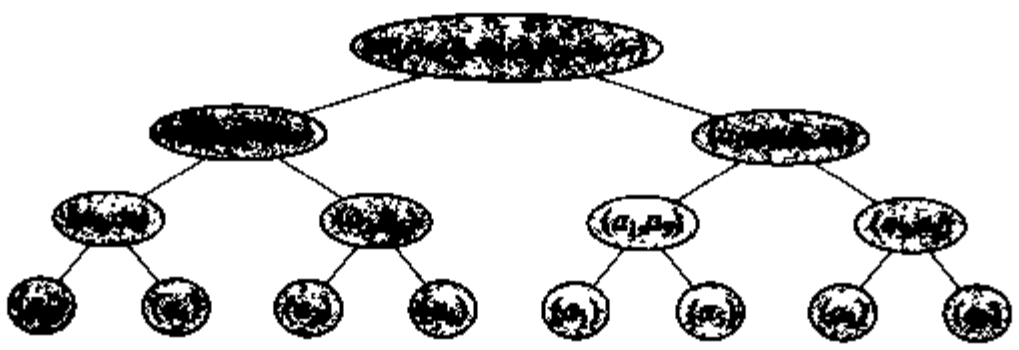


图 30-4　过程 RECURSIVE-FFT 的递归调用产生的输入向量树，初始调用时 $n=8$

在对树进行观察时，我们注意到如果把初始向量 a 中的元素按其在叶中出现的次序进行排列，就可以对过程 RECURSIVE-FFT 的执行过程模拟如下。首先，按对来取出元素，运用一次蝴蝶操作计算出每对的 DFT，并且用其 DFT 来取代该对元素。这样向量中就包含了 $n/2$ 个二元素的 DFT。下一步，按对取出这 $n/2$ 个 DFT，通过执行两次蝴蝶操作计算出四个向量元素的 DFT，并用这四个元素的 DFT 取代相对应的两个二元素的 DFT。于是向量中包含了 $n/4$ 个四元素的 DFT。继续进行这一过程，直至向量包含两个 $n/2$ 元素的 DFT，这时，再使用 $n/2$ 次蝴蝶操作，就可以把它们组合成最终的 n 个元素的 DFT。

为了把这一观察到的结果变为代码，我们利用了一个数组 $A[0..n-1]$，初始时该数组包含输入向量 a 中的元素，其顺序为它们在图 30-4 中的树的叶出现的顺序(我们在后面将说明如何确定这个顺序，它也被称为位反转置换)。因为在树的每一层都要进行组合，所以引入一个变量 s 以计算树的层次，其取值范围为从 1(在最底层，这时我们组合对来形成二元素 DFT)到 $\lg n$(在最顶层上，要对两个 $n/2$ 元素的 DFT 进行组合，以产生最后结果)。因此，该算法有如下结构：

```
1  for s ← 1 to lg n
2      do for k ← 0 to n−1 by 2^s
3             do combine the two 2^(s−1)-element DFT's in
                    A[k..k+2^(s−1)−1]and A[k+2^(s−1)..k+2^s−1]
                    into one 2^s-element DFT in A[k..k+2^s−1]
```

839
~
840

第 3 行中的循环体可以用更详细的伪代码来描述。我们从过程 RECURSIVE-FFT 中复制 for 循环，使 $y^{[0]}$ 与 $A[k..k+2^{s-1}-1]$ 相一致，$y^{[1]}$ 与 $A[k+2^{s-1}..k+2^s-1]$ 相一致。在每次蝴蝶操作中用到的旋转因子依赖于 s 的值；它是 ω_m 的幂，其中 $m=2^s$(引入变量 m 仅仅是为了使代码易读)。此外又引入另一个临时变量 u，以便能在本地执行蝴蝶操作。当用循环体来取代第 3 行的整个结构时，就得到下列伪代码，它是稍后将说明的并行实现的基础。这个代码首先调用辅助过程 BIT-REVERSE-COPY(a, A)，以便把向量 a 按需要的初始顺序复制到数组 A 中。

```
ITERATIVE-FFT(a)
 1  BIT-REVERSE-COPY(a, A)
 2  n ← length[a]          ▷n 是 2 的幂
 3  for s ← 1 to lg n
 4      do m ← 2^s
 5         ω_m ← e^(2πi/m)
 6         for k ← 0 to n−1 by m
 7             do ω ← 1
 8                forj ← 0 to m/2−1
 9                   do t ← ωA[k+j+m/2]
10                      u ← A[k+j]
11                      A[k+j]← u+t
12                      A[k+j+m/2]← u−t
13                      ω← ωω_m
```

过程 BIT-REVERSE-COPY 是怎样把输入向量 a 中的元素按要求的顺序放入数组 A 的呢？在图 30-4 中，叶出现的顺序是一个位反转置换。亦即，如果令 rev(k)为把 k 的二进制表示的各位反转所形成的 $\lg n$ 位的整数，则我们希望把向量中的元素 a_k 放在数组 $A[\text{rev}(k)]$的位置上。例如，在图 30-4 中，叶出现的次序为 0，4，2，6，1，5，3，7；这个序列用二进制表示为 000，100，010，110，001，101，011，111，把二进制各位反转之后，得到序列 000，001，010，011，100，101，110，111。为了说明我们希望获得一般情况下的位反转置换，注意到在树的最顶层、最低位为 0 的下标被放在左子树中，而最低位为 1 的下标被放在右子树中。在每一层去

841 掉最低位后，沿着树下降继续这一过程，直至在叶得到由位反转置换给定的顺序。

由于函数 rev(k)的值很容易计算，所以过程 BIT-REVERSE-COPY 有如下代码：

```
BIT-REVERSE-COPY(a, A)
1  n ← length[a]
2  for k ← 0 to n−1
3       do A[rev(k)]← a_k
```

这种迭代的 FFT 实现方法的运行时间为 $\Theta(n\lg n)$。调用 BIT-REVERSE-COPY 的运行时间当然是 $O(n\lg n)$，因为我们迭代了 n 次，并且可以在 $O(\lg n)$时间内，对一个在 0 到 $n-1$ 之间的 $\lg n$ 位整数进行反向操作(在实际应用中，通常事先就知道了 n 的初值，所以可以计算出一张表，求出每个 k 的rev(k)，使 BIT-REVERSE-COPY 的运行时间为 $\Theta(n)$，且该式中隐含的常数也较小。此外，也可以采用思考题 17-1 中描述的精巧的平摊反转二进制计数器方案)。为了完成对命题 ITERATIVE-FFT 的运行时间为 $\Theta(n\lg n)$的证明，我们证明最内层循环体(第 8～13 行)被执行的次数 $L(n)$为 $\Theta(n\lg n)$。对 s 的每个值，第 6～13 行的 for 循环迭代 $n/m=n/2^s$ 次，第 8～13 行的最内层循环迭代 $m/2=2^{s-1}$次。因此，有：

$$L(n)=\sum_{s=1}^{\lg n}\frac{n}{2^s}\cdot 2^{s-1}=\sum_{s=1}^{\lg n}\frac{n}{2}=\Theta(n\lg n)$$

并行 FFT 电路

可以利用使得能够有效实现迭代 FFT 算法的许多性质，来产生一个有效的并行 FFT 算法。我们将并行 FFT 算法表示成一个与第 27 章的比较网络相似的电路。FFT 电路使用蝴蝶操作而不是比较器，如图 30-3b 所示。在第 27 章所设计的深度的记号在这里也能用到。图 30-5 说明了 $n=8$时，计算出关于 n 个输入的 FFT 的 PARALLEL-FFT 电路。电路一开始就对输入进行位反

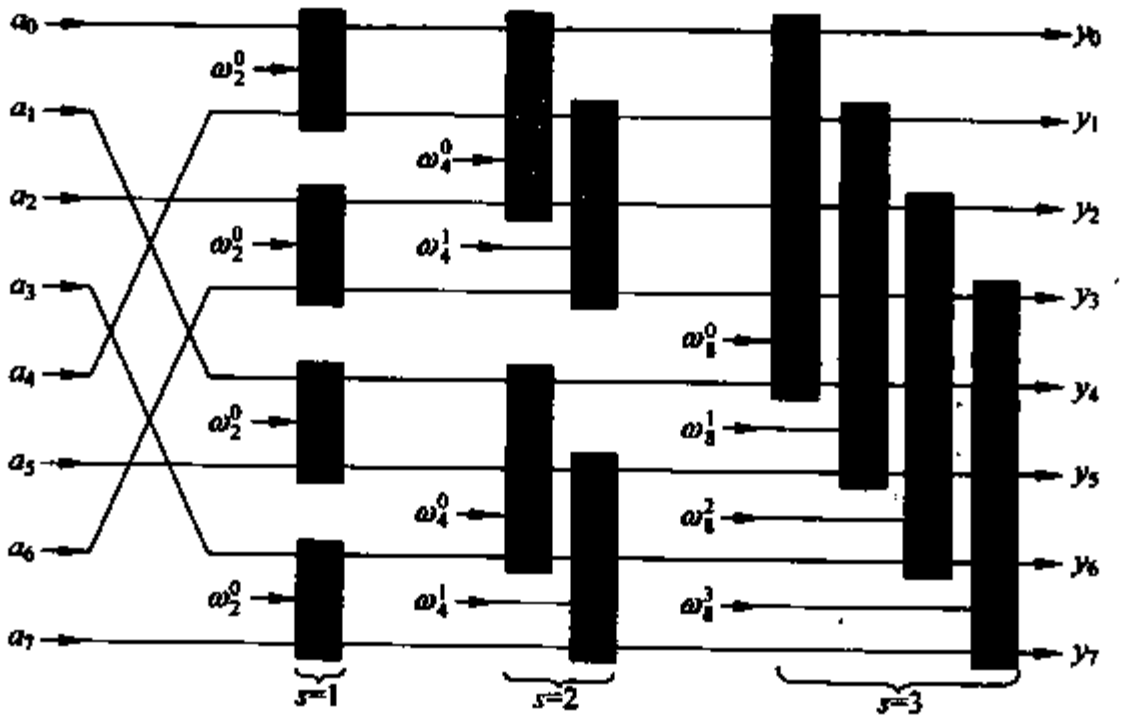


图 30-5　一个计算 FFT 的电路 PARALLEL-FFT，这里的输入为 $n=8$。每个蝴蝶操作采用两条线路上的数值和一个旋转因子来当作输入，它产生两条线路上的数值作为输出。蝴蝶的级被标示以对应于 ITERATIVE-FFT 过程的最外层循环的迭代。只有顶端和底端通过蝴蝶的线路才与蝴蝶接触；而通过蝴蝶中间的线路不会影响蝴蝶，它们的值也不会被蝴蝶改变。例如，在第 2 级顶端蝴蝶与线路 1(输出标示为 y_1 的线路)无影响，它的输入与输出只在线路 0 和 2 上(分别标示为 y_0 和 y_2)。一个 FFT 在 n 的输入上可以用 $\Theta(n\lg n)$个蝴蝶操作在 $\Theta(\lg n)$深度内计算出来

转置换，其后的电路分为 $\lg n$ 级，每一级由 $n/2$ 个并行执行的蝴蝶操作所组成。因此电路的深度为 $\Theta(\lg n)$。

电路 PARALLEL-FFT 的最左边的部分执行位反转置换，其余部分模拟迭代的 ITERATIVE FFT 过程。因为最外层 for 循环的每次迭代均执行 $n/2$ 次独立的蝴蝶操作，所以电路并行地执行它们。在过程 ITERATIVE-FFT 中每次迭代的值 s 对应于图 30-5 中的一级蝴蝶。在第 s 级中($s=1, 2, \cdots, \lg n$)，有 $n/2^s$ 组蝴蝶(对应于 ITERATIVE-FFT 中 k 的每个值)，每组中有 2^{s-1} 个蝴蝶(对应于 ITERATIVE-FFT 中 j 的每个值)。图 30-5 所示的蝴蝶对应于最内层循环(ITERATIVE-FFT 的第 9～12 行)中的蝴蝶操作。还要注意，蝴蝶中用到的旋转因子对应于 ITERATIVE-FFT 中用到的那些旋转因子：在第 s 级，我们使用了 $\omega_m^0, \omega_m^1, \cdots, \omega_m^{m/2-1}$，其中 $m=2^s$。

练习

842
～
843

30.3-1 试说明如何用过程 ITERATIVE-FFT 计算出输入向量(0，2，3，−1，4，5，7，9)的 DFT。

30.3-2 试说明如何把位反转置换放在计算过程的最后而不是在开始处，以实现 FFT 算法。(提示：考虑逆 DFT。)

30.3-3 ITERATIVE-FFT 在每级中计算旋转因子多少次？重写 ITERATIVE-FFT，使其在阶段 s 中只计算旋转因子 2^{s-1} 次。

30.3-4 假设 FFT 电路中的加法器有时会发生错误：不论输入如何，它们的输出总是为 0。假定确有一个加法器发生上述情况，但读者并不知道是哪一个加法器。描述如何能够通过给整个 FFT 电路提供输入值并观察其输出，来找到那个产生错误的加法器？你的方法的效率如何？

思考题

30-1　分治乘法

a)说明如何仅用三次乘法，就能求出线性多项式 $ax+b$ 与 $cx+d$ 的乘积。(提示：有一个乘法运算是 $(a+b)\cdot(c+d)$。)

b)试写出两种分治算法，使其在 $\Theta(n^{\lg 3})$ 的运行时间内，求出两个次数界为 n 的多项式的乘积。第一个算法把输入多项式的系数分成高阶系数与低阶系数各一半，第二个算法根据其系数下标的奇偶性来进行划分。

c)证明：用 $O(n^{\lg 3})$ 步可以计算出两个 n 比特的整数的乘积，其中每一步至多对固定数量 1 比特的值进行操作。

30-2　Toeplitz 矩阵

Toeplitz 矩阵是一个满足如下条件的 $n\times n$ 矩阵 $A=(a_{ij})$：$a_{ij}=a_{i-1,j-1}$；$i=2, 3, \cdots, n$；$j=2, 3, \cdots, n$。

a)两个 Toeplitz 矩阵的和是否一定是 Toeplitz 矩阵？其积又如何？

b)试说明如何表示 Toeplitz 矩阵，才能在 $O(n)$ 时间内求出两个 $n\times n$Toeplitz 矩阵的和。

844

c)写出一个运行时间为 $O(n\lg n)$ 的算法，使其能够计算出 $n\times n$Toeplitz 矩阵与 n 维向量的乘积。请在算法中运用 b)中的 Toeplitz 矩阵表示法。

d)写出一个高效算法，使其能够计算出两个 $n\times n$Toeplitz 矩阵的乘积，并分析算法的运行时间。

30-3　多维快速傅里叶变换

可以将式(30.8)所定义的1维离散傅里叶变换推广到d维上。输入是一个d维的数组$A=(a_{i_1,i_2,\cdots,i_d})$，维数分别为$n_1$，$n_2$，…，$n_d$，其中$n_1n_2\cdots n_d=n$。用下面的等式($0\leqslant k_1<n_1$，$0\leqslant k_2<n_2$，…，$0\leqslant k_d<n_d$)来定义$d$维的离散傅里叶变换：

$$y_{k_1,k_2,\cdots,k_d}=\sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1}\cdots\sum_{j_d=0}^{n_d-1}a_{j_1,j_2,\cdots,j_d}\omega_{n_1}^{j_1k_1}\omega_{n_2}^{j_2k_2}\cdots\omega_{n_d}^{j_dk_d}$$

a)证明可以通过依次在每个维上计算1维的DFT，来计算一个d维的DFT。也就是说，首先沿着第1维计算n/n_1个独立的1维DFT。然后，把沿着第1维的DFT的结果来作为输入，计算沿着第2维的n/n_2个独立的1维的DFT。利用这个结果作为输入，计算沿着第3维的n/n_3个独立的1维的DFT，等等，直到第d维。

b)证明维的次序并无影响，因此可以通过在任意次序的d维中计算1维DFT来计算一个d维的DFT。

c)证明如果通过计算快速傅里叶变换来计算每1维的DFT，则计算一个d维的DFT的总时间是$O(n\lg n)$，与d无关。

30-4　求多项式在某一点的所有阶导数的值

已知一个次数界为n的多项式$A(x)$，其t阶导数定义如下：

$$A^{(t)}(x)=\begin{cases}A(x) & \text{若 } t=0\\ \dfrac{\mathrm{d}}{\mathrm{d}x}A^{(t-1)}(x) & \text{若 } 1\leqslant t\leqslant n-1\\ 0 & \text{若 } t\geqslant n\end{cases}$$

根据$A(x)$的系数表示$(a_0,a_1,\cdots,a_{n-1})$和一个已知点x_0，我们希望计算出

845　$A^{(t)}(x_0)$，$t=0$，1，…，$n-1$。

a)已知系数b_0，b_1，…，b_{n-1}满足

$$A(x)=\sum_{j=0}^{n-1}b_j(x-x_0)^j$$

说明如何在$O(n)$时间内计算出$A^{(t)}(x_0)$，$t=0$，1，…，$n-1$。

b)说明如何在$O(n\lg n)$的时间内找到b_0，b_1，…，b_{n-1}，已知$A(x_0+\omega_n^k)$，$k=0$，1，…，$n-1$。

c)证明：

$$A(x_0+\omega_n^k)=\sum_{r=0}^{n-1}\left(\frac{\omega_n^{kr}}{r!}\sum_{j=0}^{n-1}f(j)g(r-j)\right)$$

其中$f(j)=a_j\cdot j!$，并且

$$g(l)=\begin{cases}x_0^{-l}/(-l)! & \text{若 } -(n-1)\leqslant l\leqslant 0\\ 0 & \text{若 } 1\leqslant l\leqslant(n-1)\end{cases}$$

d)试说明如何在$O(n\lg n)$的时间内求出$A(x_0+\omega_n^k)$的值，$k=0$，1，…，$n-1$。证明：可以在$O(n\lg n)$的时间内，求出$A(x)$的所有非平凡导数在x_0处的值。

30-5　多项式在多个点的求值

我们已经注意到，运用霍纳(Horner)法则，就能够在$O(n)$的时间内，求出次数界为$n-1$的多项式在单个点的值。同时也发现，运用FFT也能够在$O(n\lg n)$的时间内，求出多项式在所有n个单位复根处的值。现在我们就来说明如何在$O(n\lg^2 n)$的时间内，求出一个次数界为n的多项式在任意n个点的值。

为了做到这一点，我们将不加证明地运用下列结论：当一个多项式除以另一个多项式时，可以在 $O(n\lg n)$的时间内计算出其多项式余式。例如，多项式 $3x^3+x^2-3x+1$ 除以多项式 x^2+x+2 所得的余式为

$$(3x^3+x^2-3x+1)\bmod(x^2+x+2)=-7x+5$$

已知多项式 $A(x)=\sum_{k=0}^{n-1}a_kx^k$ 的系数表示和 n 个点 $x_0,x_1,\cdots,x_{n-1}$，我们希望计算出 n 个值 $A(x_0),A(x_1),\cdots,A(x_{n-1})$。对 $0\leqslant i\leqslant j\leqslant n-1$，定义多项式 $P_{ij}(x)=\prod_{k=i}^{j}(x-x_k)$，$Q_{ij}=A(x)\bmod P_{ij}(x)$。注意多项式 $Q_{ij}(x)$ 的次数界至多为 $j-i$。 846

a)证明：对任意点 z，有 $A(x)\bmod(x-z)=A(z)$。

b)证明：$Q_{kk}(x)=A(x_k)$，且 $Q_{0,n-1}(x)=A(x)$。

c)证明：对 $i\leqslant k\leqslant j$，有 $Q_{ij}(x)=Q_{ij}(x)\bmod P_{ik}(x)$，且 $Q_{kj}(x)=Q_{ij}(x)\bmod P_{kj}(x)$。

d)给出一个运行时间为 $O(n\lg^2 n)$的算法以求出 $A(x_0)$，$A(x_1)$，…，$A(x_{n-1})$的值。

30-6　运用模运算的 FFT

如其定义所述，离散傅里叶变换(DFT)要求使用复数，因此，由于舍入误差而导致精确性下降。对某些问题来说，我们已知其答案仅包含整数，并且为了保证准确地计算出答案，要求我们利用基于模运算的一种 FFT 的变异。例如，求两个整系数的多项式的积的问题就属于这类问题。练习 30.2-6 说明了这类问题的一种解决方法，即运用一个长度为 $\Omega(n)$位的模来处理 n 个点的 DFT。下面给出了另一种解决方法，即运用一个更为合理的长度 $O(\lg n)$的模；它要求读者事先理解第 31 章的内容。设 n 为 2 的幂。

a)假定我们寻找最小的 k，使 $p=kn+1$ 为质数。试给出下列结论的简单而有启发性的证明：我们预计 k 约为 $\lg n$(k 的值可能比 $\lg n$ 大一些或小一些，但我们能够合理地预计出 k 的 $O(\lg n)$个候选值的平均值)。p 的预计长度与 n 的长度有什么关系？

设 g 是 $\mathbf{Z}_p^*$ 的生成元，并且设 $w=g^k\bmod p$。

b)说明 DFT 与逆 DFT 对模 p 来说是有良定义的逆运算，其中 w 用作主 n 次单位根。

c)论证对模 p 来说，FFT 与其逆可以在 $O(n\lg n)$的时间内运行，假定算法已知 p 和 w，并且在长度为 $O(\lg n)$位的字上的操作仅需单位时间。

d)对模 $p=17$，计算出向量(0，5，3，7，7，2，1，6)的 DFT。注意，$g=3$ 是 $\mathbf{Z}_{17}^*$ 的生成元。

本章注记

VanLoan 的书[303]对快速傅里叶变换作了非常好的讲解。Press、Flannery、Teukolsky 和 Vetterling [248，249]中有关于快速傅里叶变换及其应用的很好的描述。对于信号处理这个流行 847
的 FFT 应用领域，可以参考 Oppenheim 和 Schafer [232]，以及 Oppenheim 和 Willsky [233]的教科书。Oppenheim 和 Schafer 的书同时也介绍了如何处理 n 不是 2 的整数次方时的情况。

傅里叶分析并不局限于 1 维的数据。在图像处理中，分析 2 维或更多维数据时它也被广泛使用。Gonzalez 与 Woods [127]和 Pratt[246]讨论了多维的傅里叶变换以及它们在图像处理中的应用，而 Tolimieri、An 与 Lu [300]和 Van Loan [303]的书讨论了多维的快速傅里叶变换的数学。

Cooley 与 Tukey[68]因为在 20 世纪 60 年代发明 FFT 而备受推崇。事实上 FFT 在之前已经被发现了好几次，但是它的重要性在现代数字计算机出现之前并没有完全被了解。虽然 Press、Flannery、Teukolsky 与 Vetterling 将这个方法归功于 1924 年的 Runge 和 König，但一篇 Heideman、Johnson 与 Burrus 的论文[141]将 FFT 的历史追溯到 1805 年的 C. F. Gauss。 848

第 31 章　有关数论的算法

数论一度被认为是漂亮的但却没什么大用处的纯数学学科。今天，有关数论的算法被广泛使用，部分是因为基于大素数的密码系统的发明。这些系统的可行之处在于我们容易求出大素数，而系统的安全性在于大素数的积难以分解。本章介绍一些初等数论知识和相关的算法，它们是这样一些应用的基础。

31.1 节介绍数论的基本概念，例如整除性、同模和唯一因子分解等。31.2 节要研究一个世界上很古老的算法：关于计算两个整数的最大公约数的欧几里得算法。31.3 节回顾模运算的概念。31.4 节讨论一个已知数 a 的倍数模 n 所得到的集合，并说明如何利用欧几里得算法求出方程 $ax \equiv b (\mathrm{mod}\, n)$ 的所有解。31.5 节阐述中国余数定理。31.6 节考察已知数 a 的幂模 n 所得的结果，并阐述一种已知 a，b 和 n，可以有效计算 a^b 模 n 的反复平方算法。这一运算是有效地进行素数基本测试和很多现代加密系统的核心。31.7 节描述 RSA 公开密钥加密系统。31.8 节主要讨论随机性素数基本测试，它可以用于有效地找出大素数，这是我们为 RSA 加密系统构造密钥的过程中所必须完成的基本任务。最后，31.9 节回顾一种把小整数分解因子的简单而有效的启发性方法。令人惊奇的是，分解因子是人们可能想到的一个难于处理的问题。这也许是因为 RSA 系统的安全性取决于对大整数进行因子分解的困难程度。

输入的规模与算术运算的代价

因为我们将处理一些大整数，所以需要调整一下如何看待输入规模和基本算术运算的代价的看法。

849 在本章中，一个“大的输入”意味着输入包含“大的整数”，而不是输入中包含“许多整数”（如排序的情况）。因此，我们将根据表示输入数所要求的位数来衡量输入的规模，而不是仅根据输入中包含的整数的个数。我们说，具有整数输入 a_1，a_2，…，a_k 的算法是多项式时间算法，仅当其运行时间表示为 $\lg a_1$，$\lg a_2$，…，$\lg a_k$ 的多项式，即它是转换为二进制的输入长度的多项式。

在本书大部分章节中，我们发现把基本算术运算（乘法、除法或余数的计算）看作仅需一个单位时间的原语操作是很方便的。通过计算一个算法所执行的这种算术运算的次数，以此为基础，合理地估算出算法在计算机上的实际运行时间。但是，当输入值很大时，基本操作也可能是费时的。因此，衡量一个数论算法所要求的位操作的次数将是比较适宜的。在这种模型中，用普通的方法进行两个 β 位整数的乘法，需要进行 $\Theta(\beta^2)$ 次位操作。类似地，一个 β 位整数除以一个短整数的运算，或者求一个 β 位整数除以一个短整数所得的余数的运算，也可以用简单算法在 $\Theta(\beta^2)$ 的时间内完成（参见练习 31.1-11）。目前也有更快的算法。例如，关于两个 β 位整数相乘这一运算，一种简单分治算法的运行时间为 $\Theta(\beta^{\lg 3})$，而目前已知的最快算法的运行时间为 $\Theta(\beta \lg \beta \lg \lg \beta)$。在实际应用中，$\Theta(\beta^2)$ 的算法常常是最好的算法，我们将用这个界作为分析的基础。

在本章中，我们在分析算法时一般既考虑算术运算的次数，也考虑它们所要求的位操作的次数。

31.1　初等数论概念

本节简单地回顾有关整数集合 $\mathbf{Z} = \{\cdots, -2, -1, 0, 1, 2, \cdots\}$ 和自然数集合 $\mathbf{N} = \{0, 1, 2, \cdots\}$ 的初等数论概念。 ■

整除性和约数

一个整数能被另一个整数整除的概念是数论中的一个中心概念。记号 $d \mid a$(读作"d 整除 a")意味着对某个整数 k，有 $a=kd$。0 可被任何整数整除。如果 $a>0$ 且 $d \mid a$，则 $|d| \leqslant |a|$。如果 $d \mid a$，则也可以说 a 是 d 的倍数。如果 a 不能被 d 整除，则写作 $d \nmid a$。

如果 $d \mid a$ 并且 $d \geqslant 0$，则我们说 d 是 a 的约数。注意，$d \mid a$ 当且仅当 $-d \mid a$，因此定义约数为非负整数不会失去一般性，只要明白 a 的任何约数的相应负数同样能整除 a。一个整数 a 的 850
约数最小为 1，最大为 $|a|$。例如，24 的约数有 1，2，3，4，6，8，12 和 24。

每个整数 a 都可以被其平凡约数 1 和 a 整除，a 的非平凡约数也称为 a 的因子。例如，20 的因子有 2，4，5 和 10。

素数和合数

对于某个整数 $a>1$，如果它仅有平凡约数 1 和 a，则称 a 为素数(或质数)。素数具有许多特殊性质，在数论中起着关键作用。按顺序看，前 20 个素数为：

2，3，5，11，13，17，19，23，29，31，37，41，43，47，53，59，61，67，71

练习 31.1-1 要求读者证明有无穷多个素数。不是素数的整数 $a>1$ 称为合数。例如，因为有 $3 \mid 39$，所以 39 是合数。整数 1 被称为基数，它既不是素数也不是合数。类似地，整数 0 和所有负整数既不是素数，也不是合数。

除法定理，余数和同模

已知一个整数 n，所有整数都可以划分为是 n 的倍数的整数，以及不是 n 的倍数的整数。对于不是 n 的倍数的那些整数，又可以根据它们除以 n 所得的余数来进行分类，数论的大部分理论都是基于上述划分的。下列定理是进行这种划分的基础。此处不给出该定理的证明过程(参见 Niven 和 Zuckerman[231])。

定理 31.1(除法定理)　对任意整数 a 和任意正整数 n，存在唯一的整数 q 和 r，满足 $0 \leqslant r < n$，并且 $a=qn+r$。 ■

值 $q=\lfloor a/n \rfloor$ 称为除法的商。值 $r=a \bmod n$ 称为除法的余数。$n \mid a$ 当且仅当 $a \bmod n=0$。

根据整数模 n 所得的余数，可以把整数分成 n 个等价类。包含整数 a 的模 n 等价类为：

$$[a]_n = \{a+kn : k \in \mathbf{Z}\}$$

例如，$[3]_7=\{\cdots, -11, -4, 3, 10, 17, \cdots\}$；该集合还有其他记法 $[-4]_7$ 和 $[10]_7$。利用 3.2 节里的概念，可以说写法 $a \in [b]_n$ 等同于 $a \equiv b \pmod n$。所有这样的等价类的集合为： 851

$$Z_n = \{[a]_n : 0 \leqslant a \leqslant n-1\} \tag{31.1}$$

我们经常见到定义

$$Z_n = \{0,1,\cdots,n-1\} \tag{31.2}$$

如果用 0 表示 $[0]_n$，用 1 表示 $[1]_n$ 等等，每一类均用其最小的非负元素来表示，则上述两个定义式(31.1)和式(31.2)是等价的。但是，我们必须记住相应的等价类。例如，提到 Z_n 的元素 -1 就是指 $[n-1]_n$，因为 $-1 \equiv n-1 \pmod n$。

公约数与最大公约数

如果 d 是 a 的约数并且也是 b 的约数，则 d 是 a 与 b 的公约数。例如，30 的约数为 1，2，3，5，6，10，15 和 30，因此 24 与 30 的公约数为 1，2，3 和 6。注意，1 是任意两个整数的公约数。

公约数的一条重要性质为：

$$d \mid a \text{ 并且 } d \mid b \text{ 蕴含着 } d \mid (a+b) \text{ 并且 } d \mid (a-b) \tag{31.3}$$

更一般地，对任意整数 x 和 y，有

$$d \mid a \text{ 并且 } d \mid b \text{ 蕴含着 } d \mid (ax + by) \tag{31.4}$$

同样，如果 $a \mid b$，则，或者 $|a| \leqslant |b|$，或者 $b=0$，这说明：

$$a \mid b \text{ 且 } b \mid a \text{ 蕴含着 } a = \pm b \tag{31.5}$$

两个不同时为 0 的整数 a 与 b 的最大公约数表示成 $\gcd(a, b)$。例如，$\gcd(24, 30)=6$，$\gcd(5, 7)=1$，$\gcd(0, 9)=9$。如果 a 与 b 不同时为 0，则 $\gcd(a, b)$是一个在 1 与 $\min(|a|, |b|)$之间的整数。定义 $\gcd(0, 0)=0$；对于使 gcd 函数的一般性质(如下面的等式(31.9))普遍成立来说，是必不可少的。

下列性质是 gcd 函数的基本性质：

$$\gcd(a,b) = \gcd(b,a) \tag{31.6}$$

$$\gcd(a,b) = \gcd(-a,b) \tag{31.7}$$

$$\gcd(a,b) = \gcd(|a|,|b|) \tag{31.8}$$

$$\gcd(a,0) = |a| \tag{31.9}$$

$$\gcd(a,ka) = |a| \quad \text{对任何 } k \in \mathbf{Z} \tag{31.10}$$

852 下列定理给出了 $\gcd(a, b)$的另外一个有用的性质。

定理 31.2　如果 a 和 b 是不都为 0 的任意整数，则 $\gcd(a, b)$是 a 与 b 的线性组合集合$\{ax+by: x, y \in \mathbf{Z}\}$中的最小正元素。

证明：设 s 是 a 与 b 的线性组合集中的最小正元素，并且对某个 x，$y \in \mathbf{Z}$，有 $s=ax+by$。设 $q=\lfloor a/s \rfloor$。则式(3.8)说明

$$a \bmod s = a - qs = a - q(ax + by) = a(1 - qx) + b(-qy)$$

因此，$a \bmod s$ 也是 a 与 b 的一种线性组合。但由于 $a \bmod s < s$，所以有 $a \bmod s = 0$，因为 s 是满足这样的线性组合的最小正数。因此有 $s \mid a$，并且类似可推得 $s \mid b$。因此，s 是 a 与 b 的公约数，所以 $\gcd(a, b) \geqslant s$。因为 $\gcd(a, b)$能同时被 a 与 b 整除，并且 s 是 a 与 b 的一个线性组合，所以由式(31.4)可知 $\gcd(a, b) \mid s$。但由 $\gcd(a, b) \mid s$ 和 $s > 0$，可知 $\gcd(a, b) \leqslant s$。将上面已证明的 $\gcd(a, b) \geqslant s$ 与 $\gcd(a, b) \leqslant s$ 结合起来，得到 $\gcd(a, b) = s$，因此证得 s 是 a 与 b 的最大公约数。 ■

推论 31.3　对任意整数 a 与 b，如果 $d \mid a$ 并且 $d \mid b$，则 $d \mid \gcd(a, b)$。

证明：根据定理 31.2，$\gcd(a, b)$是 a 与 b 的一个线性组合，所以从式(31.4)可推得该推论成立。 ■

推论 31.4　对所有整数 a 和 b 以及任意非负整数 n，

$$\gcd(an,bn) = n \gcd(a,b)$$

证明：如果 $n=0$，该推论显然成立。如果 $n>0$，则 $\gcd(an, bn)$是集合$\{anx+bny\}$中的最小正元素，即为集合$\{ax+by\}$中的最小正元素的 n 倍。 ■

推论 31.5　对所有正整数 n，a 和 b，如果 $n \mid ab$ 并且 $\gcd(a, n)=1$，则 $n \mid b$。

853 **证明**：证明过程留作练习 31.1-4。 ■

互质数

如果两个整数 a 与 b 仅有公因数 1，即如果 $\gcd(a, b)=1$，则 a 与 b 称为*互质数*。例如，8 和 15 是互质数，因为 8 的约数为 1，2，4，8，而 15 的约数为 1，3，5，15。下列定理说明如果两个整数中每一个数都与一个整数 p 互为质数，则它们的积与 p 互为质数。

定理 31.6　对任意整数 a，b 和 p，如果 $\gcd(a, p)=1$ 且 $\gcd(b, p)=1$，则 $\gcd(ab, p)=1$。

证明：由定理 31.2 可知，存在整数 x，y，x'，y' 满足

$$ax + py = 1$$
$$bx' + py' = 1$$

把上面两个等式两边相乘，整理得

$$ab(xx') + p(ybx' + y'ax + pyy') = 1$$

因为 1 是 ab 与 p 的一个正线性组合，所以运用定理 3 就可以证明所需结论。 ■

对于整数 n_1，n_2，…，n_k，如果对任何 $i \neq j$ 都有 $\gcd(n_i, n_j) = 1$，则说整数 n_1，n_2，…，n_k 两两互质。

唯一的因子分解

下列结论说明了关于素数整除性的一个基本而重要的事实。

定理 31.7　对所有素数 p 和所有整数 a，b，如果 $p \mid ab$，则 $p \mid a$ 或 $p \mid b$（或两者都成立）。

证明：为了引入矛盾，假设 $p \mid ab$，但 $p \nmid a$ 并且 $p \nmid b$。因此，$\gcd(a, p) = 1$ 且 $\gcd(b, p) = 1$，这是因为 p 的约数只有 1 和 p。又因为假设了 a、b 都不能被 p 整除。据定理 31.6 可知，$\gcd(ab, p) = 1$；又由假设 $p \mid ab$ 可知 $\gcd(ab, p) = p$，于是产生矛盾，从而证明定理成立。 ■

从定理 31.7 可推断出，一个整数分解为素数的因子分解式是唯一的。 854

定理 31.8（唯一质因子分解）　合数 a 仅能以一种方式，写成如下的乘积形式：

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$

其中 p_i 为素数，$p_1 < p_2 < \cdots < p_r$，且 e_i 为正整数。

证明：证明过程留作练习 31.1-10。 ■

例如，数 6000 可以唯一地分解为 $2^4 \cdot 3 \cdot 5^3$。

练习

31.1-1　证明有无穷多个素数。（提示：证明素数 p_1，p_2，…，p_k 都不能整除 $(p_1 p_2 \cdots p_k) + 1$。）

31.1-2　证明：如果 $a \mid b$ 且 $b \mid c$，则 $a \mid c$。

31.1-3　证明：如果 p 是素数并且 $0 < k < p$，则 $\gcd(k, p) = 1$。

31.1-4　证明推论 31.5。

31.1-5　证明：如果 p 是素数且 $0 < k < p$，则 $p \mid \binom{p}{k}$。证明对所有整数 a，b 和素数 p，

$$(a + b)^p \equiv a^p + b^p \pmod{p}$$

31.1-6　证明：如果 a 和 b 是任意整数，且满足 $a \mid b$ 和 $b > 0$，则对任意 x 有

$$(x \bmod b) \bmod a = x \bmod a$$

在相同的假设下，证明对任意整数 x 和 y，如果 $x \equiv y \pmod{b}$，则 $x \equiv y \pmod{a}$。

31.1-7　对任意整数 $k > 0$，如果存在一个整数 a 满足 $a^k = n$，则说整数 n 为 k 次幂。如果对于某个整数 $k > 1$，$n > 1$ 是一个 k 次幂，则说 n 是非平凡幂。说明如何在关于 β 的多项式时间内，确定出一个 β 位整数 n 是非平凡幂。 855

31.1-8　证明等式(31.6)～式(31.10)。

31.1-9　证明：gcd 运算满足结合律。亦即，证明对所有整数 a，b 和 c，有：

$$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c)$$

*31.1-10　证明定理 31.8。

31.1-11　试写出一个 β 位整数除以一个短整数的有效算法，以及求一个 β 位整数除以一个短整

数的余数的有效算法。所给出的算法的运行时间应为 $O(\beta^2)$。

31.1-12　写出一个能把一个 β 位二进制整数转化为相应的十进制表示的有效算法。论证：如果长度至多为 β 的整数的乘法与除法运算所需时间为 $M(\beta)$，则执行二进制到十进制转换所需的时间为 $\Theta(M(\beta)\lg\beta)$。（提示：运用分治法，把数分为顶部和底部两部分，分别进行递归操作而获得所需结果。）

31.2　最大公约数

在本节中，要运用欧几里得算法有效地计算出两个整数的最大公约数。在对其运行时间的分析中，我们会惊奇地发现它与斐波那契数存在着联系，由此可获得欧几里得算法在最坏情况下的输入。

在本节中我们仅限于对非负整数的情况进行讨论。这一限制是有道理的，因为由式(31.8)可知 $\gcd(a, b)=\gcd(|a|, |b|)$。

原则上讲，可以根据 a 和 b 的素数因子分解，求出正整数 a 和 b 的最大公约数 $\gcd(a, b)$。的确，如果

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} \tag{31.11}$$

856

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r} \tag{31.12}$$

其中使用了零指数，使得素数集合 $p_1, p_2, \cdots, p_r$ 对于 a 和 b 相同，正如练习 31.2-1 要求读者证明的，

$$\gcd(a,b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)} \tag{31.13}$$

我们将在 31.9 节中说明，目前已知的最好的分解因子算法也不能达到多项式运行时间。因此，根据这种方法来计算最大公约数，不大可能获得一种有效的算法。

计算最大公约数的欧几里得算法基于下面的定理。

定理 31.9(GCD 递归定理)　对任意非负整数 a 和任意正整数 b，

$$\gcd(a,b) = \gcd(b, a \bmod b)$$

证明：我们将证明 $\gcd(a, b)$ 与 $\gcd(b, a \bmod b)$ 可以互相整除，这样由等式(31.5)可知，它们一定相等(因为它们都是非负整数)。

先来证明 $\gcd(a, b) \mid \gcd(b, a \bmod b)$。如果设 $d=\gcd(a, b)$，则 $d \mid a$ 并且 $d \mid b$。由等式(3.8)可知，$(a \bmod b)=a-qb$，其中 $q=\lfloor a/b \rfloor$。这样，因为 $(a \bmod b)$ 是 a 与 b 的线性组合，所以由等式(31.4)可知 $d \mid (a \bmod b)$。因此，因为 $d \mid b$ 并且 $d \mid (a \bmod b)$，由推论 31.3 可得 $d \mid \gcd(b, a \bmod b)$，或者等价地，有

$$\gcd(a,b) \mid \gcd(b, a \bmod b) \tag{31.14}$$

证明 $\gcd(b, a \bmod b) \mid \gcd(a, b)$ 的过程几乎与上面的过程一样。如果设 $d=\gcd(b, a \bmod b)$，则 $d \mid b$ 并且 $d \mid (a \bmod b)$。由于 $a=qb+(a \bmod b)$，其中 $q=\lfloor a/b \rfloor$，所以 a 是 b 和 $(a \bmod b)$ 的一个线性组合。根据等式(31.4)可得 $d \mid a$。由于 $d \mid b$ 并且 $d \mid a$，所以根据推论 31.3，有 $d \mid \gcd(a, b)$，或者等价地有

$$\gcd(b, a \bmod b) \mid \gcd(a,b) \tag{31.15}$$

运用式(31.5)，再根据式(31.14)与式(31.15)，就可以完成对本定理的证明。 ■

欧几里得算法

欧几里得(约公元前 300 年)的《几何原本》描述了下列 gcd 算法，但实际上，这一算法出现的时间可能还要更早些。它被表示为直接基于定理 31.9 的一个递归程序。输入 a 和 b 都是任意

非负整数。

857

```
EUCLID(a, b)
1  if b=0
2     then return a
3     else return EUCLID(b, a mod b)
```

下面来举例说明 EUCLID 的运行过程。考虑 gcd(30, 21)的计算过程：

$$\text{EUCLID}(30,21) = \text{EUCLID}(21,9) = \text{EUCLID}(9,3) = \text{EUCLID}(3,0) = 3$$

在这个计算过程中，三次递归调用了 EUCLID。

过程 EUCLID 的正确性可从定理 31.9 以及下列事实推出：如果算法在第 2 行返回 a，则 $b=0$，因此由式(31.9)可知 $\gcd(a, b)=\gcd(a, 0)=a$。因为在递归调用中第二个自变量的值严格递减且始终是非负的，所以算法不可能无限递归下去。因此，EUCLID 在运行终止时，总能求出正确的答案。

欧几里得算法的运行时间

下面来分析一下 EUCLID 在最坏情况下的运行时间，可以把它看成输入 a 与 b 的大小的函数。不失一般性，假定 $a>b\geqslant 0$。这个假设的合理性是基于下述观察的：如果 $b>a\geqslant 0$，则 EUCLID(a, b)立即会递归调用 EUCLID(b, a)，即如果第一个自变量小于第二个自变量，则 EUCLID 进行一次递归调用以使两个自变量对换，然后继续往下执行。类似地，如果 $b=a>0$，则过程在进行一次递归调用后就终止执行，因为 $a \bmod b=0$。

过程 EUCLID 的运行时间与其递归调用的次数成正比。在我们的分析过程中，用到了由递归式(3.21)定义的斐波那契数 F_k。

引理 31.10　如果 $a>b\geqslant 1$ 并且 EUCLID(a, b)执行了 $k\geqslant 1$ 次递归调用，则 $a\geqslant F_{k+2}$，$b\geqslant F_{k+1}$。

证明：通过对 k 进行归纳来证明引理。作为归纳的基础，设 $k=1$，则 $b\geqslant 1=F_2$。又由于 $a>b$，则必有 $a\geqslant 2=F_3$。因为 $b>(a \bmod b)$，即在每次调用中，第一个变量严格大于第二个变量；因此对每次递归调用，假设 $a>b$ 都成立。

现在进行归纳，假设执行 $k-1$ 次递归调用时引理成立；我们将证明若执行 k 次递归调用，引理同样成立。因为 $k>0$，所以有 $b>0$，并且 EUCLID(a, b)递归调用 EUCLID(b, $a \bmod b$)，其轮流调用 $k-1$ 次递归调用。根据归纳假设，可知 $b\geqslant F_{k+1}$(因此也就证明了引理的一部分)，并且有 $(a \bmod b)\geqslant F_k$。我们有 858

$$b+(a \bmod b)=b+(a-\lfloor a/b\rfloor b)\leqslant a$$

因为由 $a>b>0$ 可知 $\lfloor a/b\rfloor\geqslant 1$，所以

$$a\geqslant b+(a \bmod b)\geqslant F_{k+1}+F_k=F_{k+2}$$ ■

下面的定理是这个引理的一个直接推论。

定理 31.11(Lamé 定理)　对任意整数 $k\geqslant 1$，如果 $a>b\geqslant 1$ 且 $b<F_{k+1}$，则 EUCLID(a, b)的递归调用次数少于 k 次。 ■

可以证明定理 31.11 中的上界是所有上界中最好的。连续的斐波那契数是 EUCLID 最坏情形下的一种输入。因为 EUCLID(F_3, F_2)仅作一次递归调用，并且对 $k>2$，有 $F_{k+1} \bmod F_k=F_{k-1}$，所以也有下式成立：

$$\gcd(F_{k+1},F_k)=\gcd(F_k,(F_{k+1} \bmod F_k))=\gcd(F_k,F_{k-1})$$

因此 EUCLID(F_{k+1}, F_k)恰好进行了 $k-1$ 次递归调用，达到定理 31.11 中的上界。

由于 F_k 约为 $\phi^k/\sqrt{5}$，其中 ϕ 是由式(3.22)定义的黄金分割率$(1+\sqrt{5})/2$，所以 EUCLID 执行中的递归调用次数为 $O(\lg b)$。(更严格的紧确界请参见练习 31.2-5。)因此，如果过程 EUCLID 作用于两个 β 位数，则它将执行 $O(\beta)$次算术运算和 $O(\beta^3)$次位操作(假设 β 位数的乘法和除法运算要执行 $O(\beta^2)$次位操作。)思考题 31-2 要求读者证明位操作次数的界为 $O(\beta^2)$。

欧几里得算法的推广形式

现在来重写欧几里得算法以计算出其它的有用信息。特别地，我们推广该算法，使它能计算
859 出满足下列条件的整系数 x 和 y：

$$d = \gcd(a,b) = ax + by \tag{31.16}$$

注意，x 与 y 可能为 0 或负数。以后会发现这些系数对计算模乘法的逆是非常有用的。过程 EXTENDED-EUCLID 的输入为一对非负整数，返回一个满足式(31.16)的三元式(d, x, y)。

图 31-1 用计算 gcd(99，78)的例子说明了 EXTENDED-EUCLID 的执行过程。

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	—	3	1	0

图 31-1　EXTENDED-EUCLID 对输入 99 与 78 操作的一个例子。每一行显示一层递归：输入 a 与 b 的数值、计算数值$\lfloor a/b \rfloor$，以及返回值 d，x 与 y。返回的三元式(d, x, y)成为在下一个较高层递归中计算所使用的元组(d', x', y')。调用 EXTENDED-EUCLID(99，78)返回(3，−11，14)，因此 gcd(99，78)=3，而且 gcd(99，78)=3=99·(−11)+78·14

```
EXTENDED-EUCLID(a, b)
1  if b=0
2     then return(a, 1, 0)
3  (d', x', y') ← EXTENDED-EUCLID(b, a mod b)
4  (d, x, y) ← (d', y', x'−⌊a/b⌋y')
5  return(d, x, y)
```

过程 EXTENDED-EUCLID 是过程 EUCLID 的一个变形。第 1 行等价于 EUCLID 第 1 行中的测试"$b=0$"。如果 $b=0$，则 EXTENDED-EUCLID 不仅返回第 2 行中的 $d=a$，而且返回系数 $x=1$ 和 $y=0$，以使 $a=ax+by$。如果 $b\neq 0$，则 EXTENDED-EUCLID 首先计算出满足 $d'=\gcd(b, a \bmod b)$和

$$d' = bx' + (a \bmod b)y' \tag{31.17}$$

的(d', x', y')。对过程 EUCLID 来说，在这种情况下，有 $d=\gcd(a, b)=d'=\gcd(b, a \bmod b)$。为了得到满足 $d=ax+by$ 的 x 和 y，我们利用等式 $d=d'$和等式(3.8)来改写式(31.17)为

860 $$d = bx' + (a - \lfloor a/b \rfloor b)y' = ay' + b(x' - \lfloor a/b \rfloor y')$$

因此，当选择 $x=y'$和 $y=x'-\lfloor a/b \rfloor y'$时，就可以满足等式 $d=ax+by$。这样就证明了过程 EXTENDED-EUCLID 的正确性。

由于在 EUCLID 中，所执行的递归调用次数与在 EXTENDED-EUCLID 中所执行的递归调

用次数相等，因此，EUCLID 与 EXTENDED-EUCLID 的运行时间相同，两者相差不超过一个常数因子。亦即，对 $a>b>0$，递归调用的次数为 $O(\lg b)$。

练习

31.2-1　证明由等式(31.11)和式(31.12)可推得等式(31.13)。

31.2-2　计算过程 EXTENDED-EUCLID(899，493)的输出值(d, x, y)。

31.2-3　证明对所有整数 a，k 和 n

$$\gcd(a,n) = \gcd(a+kn,n)$$

31.2-4　仅用常量大小的存储空间(即仅存储常数个整数值)把过程 EUCLID 改写成迭代形式。

31.2-5　如果 $a>b\geqslant 0$，证明 EUCLID(a, b)至多执行了 $1+\log_\phi b$ 次递归调用。把这个界改进为 $1+\log_\phi(b/\gcd(a, b))$。

31.2-6　过程 EXTENDED-EUCLID(F_{k+1}, F_k)返回什么值？证明所给出的答案是正确的。

31.2-7　用递归等式 $\gcd(a_0, a_1, \cdots, a_n)=\gcd(a_0, \gcd(a_1, \cdots, a_n))$来定义多于两个变量的 gcd 函数。证明 gcd 函数的返回值与其自变量的次序无关。说明如何找出满足 $\gcd(a_0, a_1, \cdots, a_n)=a_0x_0+a_1x_1+\cdots+a_nx_n$ 的整数 $x_0, x_1, \cdots, x_n$。证明所给出的算法执行的除法运算次数为 $O(n+\lg(\max\{a_0, a_1, \cdots, a_n\}))$。

31.2-8　定义 $\mathrm{lcm}(a_1, a_2, \cdots, a_n)$是 n 个整数 $a_1, a_2, \cdots, a_n$ 的最小公倍数，即每个 a_i 的倍数中的最小非负整数。说明如何用(两个自变量的)gcd 函数作为子程序以有效地计算出 $\mathrm{lcm}(a_1, a_2, \cdots, a_n)$。 861

31.2-9　证明 n_1，n_2，n_3 和 n_4 是两两互质的当且仅当

$$\gcd(n_1n_2, n_3n_4) = \gcd(n_1n_3, n_2n_4) = 1$$

更一般地，证明 $n_1, n_2, \cdots, n_k$ 是两两互质的，当且仅当从 n_i 中导出的$\lceil \lg k \rceil$对数互为质数。

31.3　模运算

可以把模运算非正式地与通常的整数运算一样看待，如果执行模 n 运算，则每个结果值 x 都由集合$\{0, 1, \cdots, n-1\}$中的某个元素所取代，该元素在模 n 的意义下与 x 等价(即用 $x \bmod n$ 来取代 x)。如果仅限于运用加法、减法和乘法运算，则用这样的非正式模型就足够了。模运算模型最适合于用群论结构来进行描述，下面就给出这一更为形式化的模型。

有限群

群$(S, \oplus)$是一个集合 S 和定义在 S 上的二进制运算$\oplus$，它满足下列性质：

1)封闭性：对所有 $a, b\in S$，有 $a\oplus b\in S$。

2)单位元：存在一个元素 $e\in S$，称为群的单位元，满足对所有 $a\in S$，$e\oplus a=a\oplus e=a$。

3)结合律：对所有 $a, b, c\in S$，有$(a\oplus b)\oplus c=a\oplus(b\oplus c)$。

4)逆元：对每个 $a\in S$，存在唯一的元素 $b\in S$，称为 a 的逆元，满足 $a\oplus b=b\oplus a=e$。

例如，考察一个熟知的在加法运算下的整数 $\mathbf{Z}$ 所构成的群$(\mathbf{Z}, +)$：0 是单位元，a 的逆元为 $-a$。如果群$(S, \oplus)$满足交换律，即对所有 $a, b\in S$，有 $a\oplus b=b\oplus a$，则它是一个交换群。如果群$(S, \oplus)$满足 $|S|<\infty$，则它是一个有限群。

根据模加法与模乘法所定义的群

通过对模 n 运用加法与乘法运算，可以得到两个有限可交换群，其中 n 为一个正整数。这些

862 群基于在 31.1 节中定义的整数模 n 所形成的等价类。

为了定义 $\mathbf{Z}_n$ 上的群，需要一种合适的二进制运算。可以通过重新定义普通的加法运算与乘法运算，来获取。$\mathbf{Z}_n$ 上的加法与乘法运算很容易定义，因为两个整数的等价类唯一决定了其和或积的等价类。亦即，如果 $a\equiv a'(\mathrm{mod}n)$ 和 $b\equiv b'(\mathrm{mod}n)$，则

$$a+b\equiv a'+b' \quad (\text{模 } n)$$
$$ab\equiv a'b' \quad (\text{模 } n)$$

因此，定义模 n 加法与模 n 乘法如下：(分别用 $+_n$ 和 $\cdot_n$ 表示)

$$\begin{aligned}[a]_n+_n[b]_n&=[a+b]_n\\ [a]_n\cdot_n[b]_n&=[ab]_n\end{aligned}\tag{31.18}$$

($\mathbf{Z}_n$ 上的减法可类似定义为 $[a]_n-_n[b]_n=[a-b]_n$，但下面将会看到，除法的定义要复杂一些。)这些事实说明在 $\mathbf{Z}_n$ 中进行计算时，用每个等价类的最小非负元素作为其代表很方便，也很普遍。我们可以像通常那样，对这些代表元素执行加法、减法与乘法，但每个结果 x 都由该类的代表元素来代替(即用 $x \bmod n$ 来代替)。

运用这一模 n 加法的定义，定义模 n 加法群 $(\mathbf{Z}_n, +_n)$，它的规模为 $|\mathbf{Z}_n|=n$。图 31-2a 给出了群 $(\mathbf{Z}_6, +_6)$ 的运算表。

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

a)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

b)

图 31-2　两个有限群。其等价类由代表元素来表示。a)群 $(\mathbf{Z}_6, +_6)$。b)群 $(\mathbf{Z}_{15}^*, \cdot_{15})$

863 **定理 31.12**　系统 $(\mathbf{Z}_n, +_n)$ 是一个有限可交换群。

证明：式(31.18)表明 $(\mathbf{Z}_n+_n)$ 是封闭的。由 $+$ 满足交换律与结合律可以推出 $+_n$ 满足交换律与结合律：

$$\begin{aligned}([a]_n+_n[b]_n)+_n[c]_n&=[a+b]_n+_n[c]_n=[(a+b)+c]_n=[a+(b+c)]_n\\ &=[a]_n+_n[b+c]_n=[a]_n+_n([b]_n+_n[c]_n)\\ [a]_n+_n[b]_n&=[a+b]_n=[b+a]_n=[b]_n+_n[a]_n\end{aligned}$$

$(\mathbf{Z}_n, +_n)$ 的单位元是 0(即 $[0]_n$)。元素 a(即 $[a]_n$)的(加法)逆元是元素 $-a$(即 $[-a]_n$ 或 $[n-a]_n$)，因为 $[a]_n+_n[-a]_n=[a-a]_n=[0]_n$。■

运用模 n 乘法的定义，可以定义模 n 乘法群 $(\mathbf{Z}_n^*, \cdot_n)$。该群中的元素是 $\mathbf{Z}_n$ 中与 n 互为质数的元素组成的集合 $\mathbf{Z}_n^*$：

$$\mathbf{Z}_n^*=\{[a]_n\in\mathbf{Z}_n:\gcd(a,n)=1\}$$

为了表明 $\mathbf{Z}_n^*$ 是确切定义的，注意对 $0\leqslant a<n$，对所有整数 k，有 $a\equiv(a+kn)(\mathrm{mod}\ n)$。因此根据练习 31.2-3，因为 $\gcd(a, n)=1$，所以对所有整数 k，$\gcd(a+kn, n)=1$。因为 $[a]_n=$

$\{a+kn: k\in\mathbf{Z}\}$，所以集合 $\mathbf{Z}_n^*$ 是确切定义的。下面是这种群的一个例子：

$$\mathbf{Z}_{15}^* = \{1,2,4,7,8,11,13,14\}$$

其中定义在群上的运算是模 15 乘法运算(这里把元素$[a]_{15}$表示成 a。例如，把$[7]_{15}$表示为 7。)图 31-2b 显示了群$(\mathbf{Z}_{15}^*, \cdot_{15})$。例如，在 $\mathbf{Z}_{15}^*$中，$8\cdot 11\equiv 13 \pmod{15}$。该群的单位元为 1。

定理 31.13　系统$(\mathbf{Z}_n^*, \cdot_n)$是一个有限可交换群。

证明：定理 31.6 说明$(\mathbf{Z}_n^*, \cdot_n)$是封闭的。与定理 31.12 证明过程中对$+_n$的证明类似，可以证明$\cdot_n$也满足交换律和结合律。其单位元为$[1]_n$。为了证明逆元存在，设 a 是 $\mathbf{Z}_n^*$ 中的一个元素，并设(d, x, y)为 EXTENDED-EUCLID(a, n)的输出结果，则 $d=1$，因为 $a\in\mathbf{Z}_n^*$，而且 864

$$ax + ny = 1$$

或者等价地

$$ax \equiv 1 \pmod{n}$$

因此，$[x]_n$是$[a]_n$对模 n 乘法的逆元。关于逆元的唯一性证明留到推论 31.26。 ■

作为计算乘法逆元的一个例子，假设 $a=5$ 且 $n=11$。则 EXTENDED-EUCLID(a, n)返回$(d, x, y)=(1, -2, 1)$，于是 $1=5\cdot(-2)+11\cdot 1$。因此 -2(亦即 9 mod 11)是模 11 乘法下 5 的乘法逆元。

在本章的后面部分遇到群$(\mathbf{Z}_n, +_n)$和$(\mathbf{Z}_n^*, \cdot_n)$时，为了方便起见，仍然用代表元素来表示等价类，并且分别用通常的运算记号 + 和 ·(或并置)来表示运算$+_n$和$\cdot_n$。另外，对模 n 等价也可以用 $\mathbf{Z}_n$ 中的方程来说明。例如，下列两种表示是等价的：

$$ax \equiv b \pmod{n}$$

$$[a]_n \cdot_n [x]_n = [b]_n$$

为了更加方便，当从上下文能看出所采用的运算时，有时仅用 S 来表示群$(S, \oplus)$。因此可以用 $\mathbf{Z}_n$ 和 $\mathbf{Z}_n^*$ 来表示群$(\mathbf{Z}_n, +_n)$和$(\mathbf{Z}_n^*, \cdot_n)$。

一个元素 a 的(乘法)逆元表示为$(a^{-1} \bmod n)$。$\mathbf{Z}_n^*$ 中的除法由式 $a/b \equiv ab^{-1} \pmod{n}$定义。例如，在 $\mathbf{Z}_{15}^*$中，有 $7^{-1}\equiv 13 \pmod{15}$，因为 $7\cdot 13\equiv 91\equiv 1 \pmod{15}$，这样就有 $4/7\equiv 4\cdot 13\equiv 7 \pmod{15}$。

$\mathbf{Z}_n^*$ 的规模表示为 $\phi(n)$。这个函数称为欧拉 phi 函数，满足下式：

$$\phi(n) = n\prod_{p\mid n}\left(1-\frac{1}{p}\right) \tag{31.19}$$

其中 p 包括能整除 n 的所有素数(如果 n 是素数，则也包括 n 本身)。在此不对此公式作出证明。从直观上看，开始时有一张 n 个余数组成的表$\{0, 1, \cdots, n-1\}$，然后对每个能整除 n 的素数 p，在表中划掉所有是 p 的倍数的数。例如，由于 45 的素数约数为 5 和 3，所以， 865

$$\phi(45) = 45\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right) = 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right) = 24$$

如果 p 是素数，则 $\mathbf{Z}_p^* = \{1, 2, \cdots, p-1\}$，并且

$$\phi(p) = p-1 \tag{31.20}$$

如果 n 是合数，则 $\phi(n)<n-1$。

子群

如果$(S, \oplus)$是一个群，$S'\subseteq S$，并且$(S', \oplus)$也是一个群，则$(S', \oplus)$称为$(S, \oplus)$的子群。例如，在加法运算下，偶数形成一个整数的子群。下列定理提供了识别子群的一个有用的工具。

定理 31.14(一个有限群的非空封闭子集是一个子群)　如果$(S, \oplus)$是一个有限群，S'是 S 的一个任意非空子集并满足：对所有 $a, b\in S'$，有 $a\oplus b\in S'$，则$(S', \oplus)$是$(S, \oplus)$的一个

子群。

证明：证明过程留作练习 31.3-2。 ■

例如，集合{0，2，4，6}形成 $\mathbf{Z}_8$ 的一个子群，这是因为它是非空的，而且在＋运算下它具有封闭性(即在 $+_8$ 下它是封闭的。)

下列定理对子群的规模作出了一个非常有用的限制，证明在此略去。

定理 31.15(拉格朗日定理) 如果$(S, \oplus)$是一个有限群，$(S', \oplus)$是$(S, \oplus)$的一个子群，则 $|S'|$ 是 $|S|$ 的一个约数。 ■

对一个群 S 的子群 S'，如果 $S' \neq S$，则子群 S' 称为群 S 的真子群。在第 31.8 节中对 Miller-Rabin 素数测试过程将用到下列推论。

866 **推论 31.16** 如果 S' 是有限群 S 的真子群，则 $|S'| \leqslant |S|/2$。 ■

由一个元素生成的子群

定理 31.14 给出了一种生成一个有限群$(S, \oplus)$的子群的有趣方法：选择一个元素 a，并取出根据群上的运算由 a 所能生成的所有元素。具体地，对 $k \geqslant 1$ 定义 $a^{(k)}$ 如下：

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \cdots \oplus a}_{k}$$

例如，如果取群 $\mathbf{Z}_6$ 中的元素 $a=2$，序列 $a^{(1)}$，$a^{(2)}$，…为

$$2,4,0,2,4,0,2,4,0,\cdots$$

在群 $\mathbf{Z}_n$ 中，有 $a^{(k)} = ka \bmod n$。在群 $\mathbf{Z}_n^*$ 中，有 $a^{(k)} = a^k \bmod n$。由 a 生成的子群用$\langle a \rangle$或$(\langle a \rangle, \oplus)$来表示，其定义如下：

$$\langle a \rangle = \{a^{(k)} : k \geqslant 1\}$$

称 a 生成子群$\langle a \rangle$，或者 a 是$\langle a \rangle$的生成元。因为 S 是有限集，所以$\langle a \rangle$是 S 的有限子集，它可能包含 S 中的所有元素。由$\oplus$满足结合律可知

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)}$$

所以$\langle a \rangle$具有封闭性，根据定理 31.14，$\langle a \rangle$是 S 的一个子群。例如，在 $\mathbf{Z}_6$ 中，有

$$\langle 0 \rangle = \{0\}$$
$$\langle 1 \rangle = \{0,1,2,3,4,5\}$$
$$\langle 2 \rangle = \{0,2,4\}$$

同样地，在 $\mathbf{Z}_7^*$ 中，有

$$\langle 1 \rangle = \{1\}$$
$$\langle 2 \rangle = \{1,2,4\}$$
$$\langle 3 \rangle = \{1,2,3,4,5,6\}$$

在群 S 中 a 的阶用 $\mathrm{ord}(a)$来表示，定义为满足 $a^{(t)} = e$ 的最小正整数 t。

定理 31.17 对任意有限群$(S, \oplus)$和任意 $a \in S$，一个元素的阶等于它所生成的子群的规模，即 $\mathrm{ord}(a) = |\langle a \rangle|$。

证明：设 $t = \mathrm{ord}(a)$。因为 $a^{(t)} = e$ 并且对 $k \geqslant 1$ 有 $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$，如果 $i > t$，则对某个 $j < i$，有 $a^{(i)} = a^{(j)}$。因此，在 $a^{(t)}$ 后面不会出现新元素，于是$\langle a \rangle = \{a^{(1)}, a^{(2)}, \cdots, a^{(t)}\}$，而且

867 $|\langle a \rangle| \leqslant t$。为了证明 $|\langle a \rangle| \geqslant t$，假设为了引起矛盾，对某个满足 $1 \leqslant i < j \leqslant t$ 的 i 和 j 有 $a^{(i)} = a^{(j)}$。则对 $k \geqslant 0$，有 $a^{(i+k)} = a^{(j+k)}$。但这样就说明 $a^{(i+(t-j))} = a^{(j+(t-j))} = e$，因为 $i + (t-j) < t$，但 t 为满足 $a^{(t)} = e$ 的最小正值，这样就产生了矛盾。因此，序列 $a^{(1)}$，$a^{(2)}$，…，$a^{(t)}$ 中的每个元素都是不同的，$|\langle a \rangle| \geqslant t$。于是得出结论 $\mathrm{ord}(a) = |\langle a \rangle|$。 ■

推论 31.18　序列 $a^{(1)}$，$a^{(2)}$，…是周期性序列，其周期为 $t=\mathrm{ord}(a)$；即 $a^{(i)}=a^{(j)}$ 当且仅当 $i\equiv j \pmod t$。 ■

对所有整数 i，定义 $a^{(0)}$ 为 e，和定义 $a^{(i)}$ 为 $a^{(i \bmod t)}$，其中 $t=\mathrm{ord}(a)$，与上述推论是一致的。

推论 31.19　如果$(S, \oplus)$是一个具有单位元 e 的有限群，则对所有 $a\in S$，

$$a^{(|S|)} = e$$

证明：由拉格朗日定理可知 $\mathrm{ord}(a) \mid |S|$，因此 $|S| \equiv 0 \pmod t$，其中 $t=\mathrm{ord}(a)$。所以，

$$a^{(|S|)} = a^{(0)} = e$$ ■

练习

31.3-1　画出群$(\mathbf{Z}_4, +_4)$和群$(\mathbf{Z}_5^*, \cdot_5)$的运算表。通过找这两个群的元素间的、满足 $a+b\equiv c \pmod 4$当且仅当 $\alpha(a)\cdot\alpha(b)\equiv\alpha(c) \pmod 5$的一一对应关系 α，来证明这两个群是同构的。

31.3-2　证明定理 31.14。

31.3-3　证明：如果 p 是素数且 e 是正整数，则

$$\phi(p^e) = p^{e-1}(p-1)$$

31.3-4　证明：对任意 $n>1$ 和任意 $a\in\mathbf{Z}_n^*$，由式 $f_a(x)=ax \bmod n$ 所定义的函数 $f_a : \mathbf{Z}_n^* \to \mathbf{Z}_n^*$ 是 $\mathbf{Z}_n^*$ 的一个置换。

31.3-5　列举出 $\mathbf{Z}_9$ 和 $\mathbf{Z}_{13}^*$的所有子群。

868

31.4　求解模线性方程

现在来考虑求解下列方程的问题：

$$ax \equiv b \pmod n \tag{31.21}$$

其中 $a>0$，$n>0$。这个问题有若干种应用；例如，在 31.7 节中我们将它作为在 RSA 公钥密码系统中，寻找密钥过程的一部分。假设已知 a，b 和 n，希望求出所有满足式(31.21)的对模 n 的 x 值。可能没有解，也可能有一个或多个解。

设$\langle a\rangle$表示由 a 生成的 $\mathbf{Z}_n$ 的子群。由于$\langle a\rangle=\{a^{(x)} : x>0\}=\{ax \bmod n : x>0\}$，所以方程(31.21)有一个解当且仅当 $b\in\langle a\rangle$。拉格朗日定理(定理 31.15)告诉我们，$|\langle a\rangle|$ 必定是 n 的约数。下列定理准确地刻划了$\langle a\rangle$的特性。

定理 31.20　对任意正整数 a 和 n，如果 $d=\gcd(a, n)$，则在 $\mathbf{Z}_n$ 中

$$\langle a\rangle = \langle d\rangle = \{0, d, 2d, \cdots, ((n/d)-1)d\} \tag{31.22}$$

因此有

$$|\langle a\rangle| = n/d$$

证明：首先证明 $d\in\langle a\rangle$。注意到 EXTENDED-EDCLID(a, n)可生成满足 $ax'+ny'=d$ 的整数 x'和 y'。因此，$ax'\equiv d \pmod n$，所以 $d\in\langle a\rangle$。

由于 $d\in\langle a\rangle$，所以 d 的所有倍数均属于$\langle a\rangle$，这是因为 a 的倍数的倍数仍然是 a 的倍数。所以，$\langle a\rangle$包含了集合$\{0, d, 2d, \cdots, ((n/d)-1)d\}$中的每一个元素。亦即，$\langle d\rangle\subseteq\langle a\rangle$。

现在来证明$\langle a\rangle\subseteq\langle d\rangle$。如果 $m\in\langle a\rangle$，则对某个整数 x 有 $m=ax \bmod n$，所以对某个整数 y 有 $m=ax+ny$。但是，$d \mid a$ 并且 $d \mid n$，所以根据式(31.4)有 $d \mid m$。因此，$m\in\langle d\rangle$。

由以上这些结论，得到$\langle a\rangle=\langle d\rangle$。为了说明 $|\langle a\rangle| = n/d$，请注意在 0 和 $n-1$ 之间(包括 0 和 $n-1$)恰有 n/d 个 d 的倍数。 ■

869 **推论 31.21**　方程 $ax \equiv b \pmod{n}$对于未知量 x 有解，当且仅当 $\gcd(a, n) \mid b$。 ■

推论 31.22　方程 $ax \equiv b \pmod{n}$或者对模 n 有 d 个不同的解，其中 $d = \gcd(a, n)$，或者无解。

证明：如果 $ax \equiv b \pmod{n}$有一个解，则 $b \in \langle a \rangle$。根据定理 31.17，$\mathrm{ord}(a) = |\langle a \rangle|$，而且推论 31.18 和定理 31.20 蕴含着对 $i = 0, 1, \cdots$ 序列 $ai \bmod n$ 是一个周期性序列，其周期为 $|\langle a \rangle| = n/d$。如果 $b \in \langle a \rangle$，则 b 在序列 $ai \bmod n$ 中恰好出现 d 次，对 $i = 0, 1, \cdots, n-1$，因为当 i 从 0 增加到 $n-1$ 时，长度为 n/d 的一组值 $\langle a \rangle$ 恰好重复了 d 次，这 d 个位置的、满足 $ax \bmod n = b$ 的下标 x 就是方程 $ax \equiv b \pmod{n}$的解。 ■

定理 31.23　设 $d = \gcd(a, n)$，假定对整数 x' 和 y'，有 $d = ax' + ny'$（例如 EXTENDED-EUCLID 所计算出的结果）。如果 $d \mid b$，则方程 $ax \equiv b \pmod{n}$有一个解的值为 x_0，满足

$$x_0 = x'(b/d) \bmod n$$

证明：有

$$\begin{aligned} ax_0 &\equiv ax'(b/d) && \pmod{n} \\ &\equiv d(b/d) && \pmod{n} && (\text{由于 } ax' \equiv d \pmod{n}) \\ &\equiv b && \pmod{n} \end{aligned}$$

因此 x_0 是 $ax \equiv b \pmod{n}$的一个解。 ■

定理 31.24　假设方程 $ax \equiv b \pmod{n}$有解（即有 $d \mid b$，其中 $d = \gcd(a, n)$）x_0 是该方程的任意一个解，则该方程对模 n 恰有 d 个不同的解，分别为：$x_i = x_0 + i(n/d)\ (i = 1, 2, \cdots, d-1)$。

证明：因为 $n/d > 0$ 并且对 $i = 0, 1, \cdots, d-1$，有 $0 \leqslant i(n/d) < n$，所以对模 n，值 $x_0, x_1, \cdots, x_{d-1}$ 都是不相同的。因为 x_0 是 $ax \equiv b \pmod{n}$的一个解，有 $ax_0 \bmod n = b$。因此，对 $i = 0, 1, \cdots, d-1$，有

$$\begin{aligned} ax_i \bmod n &= a(x_0 + in/d) \bmod n \\ &= (ax_0 + ain/d) \bmod n \\ &= ax_0 \bmod n && (\text{由于 } d \mid a) \\ &= b \end{aligned}$$

因此，x_i 也是一个解。根据推论 31.22 可知方程且有 d 个解，因此 $x_0, x_1, \cdots, x_{d-1}$ 必定
870 是方程的全部解。 ■

现在已经为求解方程 $ax \equiv b \pmod{n}$做好了数学上的准备工作；下列算法可输出该方程的所有解。输入 a 和 n 为任意正整数，b 为任意整数。

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
1  (d, x', y') ← EXTENDED-EUCLID(a, n)
2  if d | b
3     then x0 ← x'(b/d) mod n
4        for i ← 0 to d-1
5           do print (x0 + i(n/d)) mod n
6     else print "no solutions"
```

下面通过一个例子来说明该过程的操作，考察方程 $14x \equiv 30 \pmod{100}$（这里，$a = 14$，$b = 30$，$n = 100$）。在第 1 行中调用 EXTENDED-EUCLID 后得到 $(d, x, y) = (2, -7, 1)$。因为 $2 \mid 30$，所以执行第 3～5 行。在第 3 行，计算 $x_0 = (-7)(15) \bmod 100 = 95$。第 4～5 行的循环输出两个解：95 和 45。

过程 MODULAR-LINEAR-EQUATION-SOLVER 执行过程如下。第 1 行计算出 $d = \gcd(a,$

n)和两个满足 $d=ax'+ny'$ 的值 x' 和 y'，同时表明 x' 是方程 $ax'\equiv d \pmod n$ 的一个解。如果 d 不能整除 b，则由推论 31.21 可知方程 $ax\equiv b \pmod n$ 没有解。第 2 行检查是否有 $d \mid b$；如果否，则第 6 行报告方程无解。否则，第 3 行将根据定理 31.23，计算出方程 $ax\equiv b \pmod n$ 的一个解 x_0。已知一个解后，由定理 31.24 可知，其他 $d-1$ 个解可以通过对模 n 加上 (n/d) 的倍数来得到。第4～5 行的 for 循环输出所有 d 个解，对模 n 从 x_0 开始，每两个解之间相差 (n/d)。

MODULAR-LINEAR-EQUATION-SOLVER 执行 $O(\lg n + \gcd(a, n))$ 次算术运算。因为 EXTENDED-EUCLID 需要执行 $O(\lg n)$ 次算术运算，并且第 4～5 行 for 循环中的每次迭代均要执行常数次算术运算。

定理 31.24 的下述推论说明了若干特殊情形。

推论 31.25　对任意 $n>1$，如果 $\gcd(a, n)=1$，则方程 $ax\equiv b \pmod n$ 对模 n 有唯一解。■

如果 $b=1$，这是通常遇到的一种重要情形，则要求的 x 是 a 的对模 n 乘法的逆元。 871

推论 31.26　对任意 $n>1$，如果 $\gcd(a, n)=1$，则方程 $ax\equiv 1 \pmod n$ 对模 n 有唯一解。否则方程无解。■

推论 31.26 使得在 a 和 n 互质时，可以用记号 $(a^{-1} \bmod n)$ 来表示 a 对模 n 乘法的逆元。如果 $\gcd(a, n)=1$，则方程 $ax\equiv 1 \pmod n$ 的一个解就是 EXTENDED-EUCLID 所返回的整数 x，因为方程

$$\gcd(a,n) = 1 = ax + ny$$

蕴含着 $ax\equiv 1 \pmod n$。因此，运用 EXTENDED-EUCLID 可以有效地计算出 $(a^{-1} \bmod n)$。

练习

31.4-1　求出方程 $35x\equiv 10 \pmod{50}$ 的所有解。

31.4-2　证明：当 $\gcd(a, n)=1$，由方程 $ax\equiv ay \pmod n$ 可得 $x\equiv y \pmod n$。通过一个反例 $\gcd(a, n)>1$ 的情况来证明条件 $\gcd(a, n)=1$ 是必要的。

31.4-3　考察下列对过程 MODULAR-LINEAR-EQUATION-SOLVER 的第 3 行的修改：

3 then $x_0 \leftarrow x'(b/d) \bmod (n/d)$

修改后算法是否能正确运行？试说明能或者不能的原因。

***31.4-4**　设 $f(x)\equiv f_0+f_1x+\cdots+f_tx^t \pmod p$ 是一个 t 次多项式，其系数 f_i 均属于 $\mathbf{Z}_p$，其中 p 为素数。如果 $f(a)\equiv 0 \pmod p$，则说明 $a\in\mathbf{Z}_p$ 为 f 的零元。证明：如果 a 是 f 的零元，则对某个 $t-1$ 次的多项式 $g(x)$，有 $f(x)\equiv(x-a)g(x) \pmod p$。通过对 t 进行归纳，来证明 t 次多项式 $f(x)$ 对模 p（p 为素数）至多有 t 个不同的零元。 872

31.5　中国余数定理

大约在公元 100 年，中国的数学家孙子解决了以下问题：找出被 3，5 和 7 除时余数分别为 2，3 和 2 的所有整数 x。有一个解为 $x=23$；所有的解是形如 $23+105k$（k 为任意整数）的整数。“中国余数定理”提出，对一组两两互质的模数（如 3，5 和 7）来说，其取模运算的方程组与对其积（如 105）取模运算的方程之间存在着一种对应关系。

中国余数定理有两个主要作用。设整数 n 因式分解为 $n=n_1n_2\cdots n_k$，其中因子 n_i 两两互质。首先，中国余数定理是一个描述性的“结构定理”，它说明 $\mathbf{Z}_n$ 的结构等同于笛卡儿积 $\mathbf{Z}_{n_1}\times\mathbf{Z}_{n_2}\times\cdots\times\mathbf{Z}_{n_k}$ 的结构，其中，第 i 个组元定义了对模 n_i 的组元之间的加法与乘法运算。其次，用这种描述常常可以获得有效的算法，因为处理 $\mathbf{Z}_{n_i}$ 系统中的每个系统可能比处理模 n 运算效率更高（从位操作次数看）。

定理 31.27(中国余数定理)　设 $n=n_1n_2\cdots n_k$，其中因子 n_i 两两互质。考虑下列对应关系：

$$a \leftrightarrow (a_1, a_2, \cdots, a_k) \tag{31.23}$$

其中 $a\in\mathbf{Z}_n$，$a_i\in\mathbf{Z}_{n_i}$，而且对 $i=1, 2, \cdots, k$

$$a_i = a \bmod n_i$$

则映射(31.23)是一个在 $\mathbf{Z}_n$ 与笛卡儿积 $\mathbf{Z}_{n_1}\times\mathbf{Z}_{n_2}\times\cdots\times\mathbf{Z}_{n_k}$ 之间的一一对应(双射)。对 $\mathbf{Z}_n$ 中元素所执行的运算可以等价地作用于对应的 k 元组，即在适当的系统中独立地对每个坐标位置执行所需的运算。亦即，如果

$$a \leftrightarrow (a_1, a_2, \cdots, a_k)$$
$$b \leftrightarrow (b_1, b_2, \cdots, b_k)$$

那么

$$(a+b) \bmod n \leftrightarrow ((a_1+b_1) \bmod n_1, \cdots, (a_k+b_k) \bmod n_k) \tag{31.24}$$

$$(a-b) \bmod n \leftrightarrow ((a_1-b_1) \bmod n_1, \cdots, (a_k-b_k) \bmod n_k) \tag{31.25}$$

873

$$(ab) \bmod n \leftrightarrow (a_1b_1 \bmod n_1, \cdots, a_kb_k \bmod n_k) \tag{31.26}$$

证明：两种表示之间的变换是非常简单的。从 a 转换为 $(a_1, a_2, \cdots, a_k)$，仅需执行 k 次除法运算。从输入 $(a_1, a_2, \cdots, a_k)$ 计算出 a 要复杂一点，它以如下的方式来计算。定义 $m_i=n/n_i$ $(i=1, 2, \cdots, k)$；因此 m_i 是除了 n_i 的所有 n_j 的乘积：$m_i=n_1n_2\cdots n_{i-1}n_{i+1}\cdots n_k$。接着对 $i=1, 2, \cdots, k$，定义

$$c_i = m_i(m_i^{-1} \bmod n_i) \tag{31.27}$$

等式(31.27)总是良定义的：因为 m_i 和 n_i 互质(根据定理 31.6)，推论 31.26 保证 $m_i^{-1} \bmod n_i$ 存在。最后，作为 $a_1, a_2, \cdots, a_k$ 的函数以如下方式计算 a：

$$a \equiv (a_1c_1 + a_2c_2 + \cdots + a_kc_k) \pmod n \tag{31.28}$$

现在来证明式(31.28)能保证 $a\equiv a_i \pmod{n_i}$ $(i=1, 2, \cdots, k)$。注意，如果 $j\neq i$，则 $m_j\equiv 0 \pmod{n_i}$，这蕴含着 $c_j\equiv m_j\equiv 0 \pmod{n_i}$。同时注意到，由式(31.27)知 $c_i\equiv 1 \pmod{n_i}$。因此得到这个吸引人且有用的对应关系

$$c_i \leftrightarrow (0, 0, \cdots, 0, 1, 0, \cdots, 0)$$

这是一个除第 i 个坐标为 1 外，其余坐标均为 0 的向量；因此在某种意义上，说 c_i 是这一表示的“基”。所以对每个 i，有

$$\begin{aligned} a &\equiv a_ic_i && \pmod{n_i} \\ &\equiv a_im_i(m_i^{-1} \bmod n_i) && \pmod{n_i} \\ &\equiv a_i && \pmod{n_i} \end{aligned}$$

这正是我们要证明的：我们从 a_i 计算 a 的方法得到结果 a，其满足约束 $a\equiv a_i \pmod{n_i}$ $(i=1, 2, \cdots, k)$。由于能够进行双向变换，所以为一一对应关系。对于任何 x 和 $i=1, 2, \cdots, k$，有 $x \bmod n_i = (x \bmod n) \bmod n_i$，所以根据练习 31.1-6，可以直接推出式(31.24)～式(31.26)成立。 ■

本章后面的部分将用到下面几条推论。

推论 31.28　如果 $n_1, n_2, \cdots, n_k$ 两两互质，$n=n_1n_2\cdots n_k$，则对任意整数 $a_1, a_2, \cdots, a_k$ $(i=1, 2, \cdots, k)$，方程组

$$x \equiv a_i \pmod{n_i}$$

874　关于未知量 x 对模 n 有唯一解。 ■

推论 31.29　如果 n_1，n_2，…，n_k 两两互质，$n=n_1n_2\cdots n_k$，则对所有整数 x 和 a（$i=1$，2，…，k），

$$x \equiv a \pmod{n_i}$$

当且仅当

$$x \equiv a \pmod{n}$$ ■

现在来举例说明中国余数定理的应用，假设已知两个方程：

$$a \equiv 2 \pmod 5$$

$$a \equiv 3 \pmod{13}$$

这样 $a_1=2$，$n_1=m_2=5$，$a_2=3$，$n_2=m_1=13$，而 $n=65$，所以我们希望计算出 $a \bmod 65$。因为 $13^{-1}\equiv 2 \pmod 5$ 和 $5^{-1}\equiv 8 \pmod{13}$，所以有

$$c_1 = 13(2 \bmod 5) = 26$$

$$c_2 = 5(8 \bmod 13) = 40$$

以及

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 && \pmod{65} \\ &\equiv 52 + 120 && \pmod{65} \\ &\equiv 42 && \pmod{65} \end{aligned}$$

图 31-3 说明了对模 65 的中国余数定理的应用。

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

图 31-3　对于 $n_1=5$ 和 $n_2=13$，中国余数定理的一个应用实例。对这个例子，$c_1=26$，$c_2=40$。在第 i 行，第 j 列显示的是 a 的值，模 65，使得 $(a \bmod 5)=i$ 和 $(a \bmod 13)=j$。注意第 0 行第 0 列的值为 0。类似地，第 4 行，第 12 列包含 64（等价于 -1）。因为 $c_1=26$，往下移动一行让 a 增加 26。类似地，$c_2=40$ 表示往右移动一列让 a 增加 40。让 a 增加 1 对应于沿着对角线往右下移动，从底端折返到顶端，以及从右端折返到左端

875

因此，如果要执行模 n 运算，则既可以直接对模 n 进行计算，当方便时，也可以把模 n 表示进行变换，再分别对模 n_i 进行运算处理。这两种计算是完全等价的。

练习

31.5-1　计算出使方程 $x\equiv 4 \pmod 5$ 和 $x\equiv 5 \pmod{11}$ 同时成立的所有解。

31.5-2　试找出被 9，8，7 除时，余数分别为 1，2，3 的所有整数 x。

31.5-3　论证：在定理 31.27 的定义下，如果 $\gcd(a, n)=1$，则

$$(a^{-1} \bmod n) \leftrightarrow (a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \cdots, (a_k^{-1} \bmod n_k))$$

31.5-4　在定理 31.27 的定义下，证明：对于任意的多项式 f，方程 $f(x)\equiv 0 \pmod n$ 的根的数目等于每个方程 $f(x)\equiv 0 \pmod{n_1}$，$f(x)\equiv 0 \pmod{n_2}$，…，$f(x)\equiv 0 \pmod{n_k}$ 的根的数目的积。

31.6　元素的幂

正如我们考虑一个已知元素 a 对模 n 的倍数一样，常常自然地考虑对模 n 的 a 的幂组成的序列，其中 $a\in\mathbf{Z}_n^*$：

$$a^0,a^1,a^2,a^3,\cdots \tag{31.29}$$

模 n。对其从 0 开始编号，则该序列中的第 0 个值为 $a^0 \bmod n=1$，第 i 个值为 $a^i \bmod n$。例如，对模 7，3 的幂为

i	0	1	2	3	4	5	6	7	8	9	10	11	…
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	…

而对模 7 来说 2 的幂为

i	0	1	2	3	4	5	6	7	8	9	10	11	…
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	…

876

在本节中，设$\langle a\rangle$表示由 a 反复相乘生成的 $\mathbf{Z}_n^*$ 的子群，$\mathrm{ord}_n(a)$(对模 n，a 的阶)表示 a 在 $\mathbf{Z}_n^*$ 中的阶。例如，在 $\mathbf{Z}_7^*$ 中，$\langle 2\rangle=\{1,2,4\}$，$\mathrm{ord}_7(2)=3$。用欧拉 phi 函数 $\phi(n)$作为 $\mathbf{Z}_n^*$ 的规模的定义(参见 31.3 节)，就可以把推论 31.19 转化为用 $\mathbf{Z}_n^*$ 表示，从而得到欧拉定理，再具体用 $\mathbf{Z}_p^*$ 来表示(p 是素数)，就得到费马定理。

定理 31.30(欧拉定理)　对于任意整数 $n>1$，$a^{\phi(n)}\equiv 1 \pmod n$对所有 $a\in\mathbf{Z}_n^*$ 都成立。 ■

定理 31.31(费马定理)　如果 p 是素数，则 $a^{p-1}\equiv 1 \pmod p$对所有 $a\in\mathbf{Z}_p^*$ 都成立。

证明：根据等式(31.20)，如果 p 是素数，$\phi(p)=p-1$。 ■

这个推论对 $\mathbf{Z}_p$ 中除了 0 以外的每一个元素都适用，因为 $0\notin\mathbf{Z}_p^*$。但是，对所有 $a\in\mathbf{Z}_p^*$，如果 p 是素数，则有 $a^p\equiv a \pmod p$。

如果 $\mathrm{ord}_n(g)=|\mathbf{Z}_n^*|$，则对模 n，$\mathbf{Z}_n^*$ 中的每个元素都是 g 的幂，且称 g 是 $\mathbf{Z}_n^*$ 的原根(primitive root)或生成元(generator)。例如，对模 7，3 是原根，但 2 不是。如果 $\mathbf{Z}_n^*$ 包含一个原根，就称群 $\mathbf{Z}_n^*$ 为循环群。下列定理是由 Niven 和 Zuckerman[231]首先证明的，在此略去证明过程。

定理 31.32　对所有的素数 $p>2$ 和所有正整数 e，满足 $\mathbf{Z}_n^*$ 为循环群的 $n(n>1)$值为 2，4，p^e 和 $2p^e$。 ■

如果 g 是 $\mathbf{Z}_n^*$ 的一个原根且 a 是 $\mathbf{Z}_n^*$ 中的任意元素，则存在一个 z 满足 $g^z\equiv a \pmod n$。这个 z 称为对模 n 到基 g 上的 a 的离散对数或指数；其值表示为 $\mathrm{ind}_{n,g}(a)$。

定理 31.33(离散对数定理)　如果 g 是 $\mathbf{Z}_n^*$ 的一个原根，则等式 $g^x\equiv g^y \pmod n$成立，当且仅当等式 $x\equiv y \pmod{\phi(n)}$成立。

877

证明：首先假设 $x\equiv y \pmod{\phi(n)}$。则对某个整数 k 有 $x=y+k\phi(n)$。因此，

$$\begin{aligned} g^x &\equiv g^{y+k\phi(n)} && \pmod n \\ &\equiv g^y\cdot(g^{\phi(n)})^k && \pmod n \\ &\equiv g^y\cdot 1^k && \pmod n \quad \text{(根据欧拉定理)} \\ &\equiv g^y && \pmod n \end{aligned}$$

相反地，再假设 $g^x\equiv g^y \pmod n$。因为 g 的幂的序列生成$\langle g\rangle$中的每一个元素，且 $|\langle g\rangle|=\phi(n)$，由推论 31.18 可知，$g$ 的幂的序列是一个周期为 $\phi(n)$的周期性序列。所以，如果 $g^x\equiv g^y \pmod n$，必有 $x\equiv y \pmod{\phi(n)}$。 ■

利用离散对数，有时可以简化对模运算方程的讨论，这一点从如下定理的证明中可以看出。

定理 31.34　如果 p 是一个奇素数且 $e \geqslant 1$，则方程

$$x^2 \equiv 1 \pmod{p^e} \tag{31.30}$$

仅有两个解：$x=1$ 和 $x=-1$。

证明：设 $n=p^e$。由定理 31.32 可知 $\mathbf{Z}_n^*$ 有一个原根 g，从而方程(31.30)可以写成

$$(g^{\mathrm{ind}_{n,g}(x)})^2 \equiv g^{\mathrm{ind}_{n,g}(1)} \pmod{n} \tag{31.31}$$

注意到 $\mathrm{ind}_{n,g}(1)=0$，由定理 31.33 可知方程(31.31)等价于

$$2 \cdot \mathrm{ind}_{n,g}(x) \equiv 0 \pmod{\phi(n)} \tag{31.32}$$

为了求解这个关于未知量 $\mathrm{ind}_{n,g}(x)$ 的方程，运用 31.4 节中的方法。根据式(31.19)，有 $\phi(n)=p^e(1-1/p)=(p-1)p^{e-1}$。设 $d=\gcd(2, \phi(n))=\gcd(2, (p-1)p^{e-1})=2$，且注意到 $d \mid 0$，所以根据定理 31.24，可以看出方程(31.32)恰有 $d=2$ 个解。因此，方程(31.30)也恰有两个解，通过观察可知，它们应为 $x=1$ 和 $x=-1$。■

如果一个数 x 满足方程 $x^2 \equiv 1 \pmod{n}$，但 x 不等于对模 n 来说 1 的两个“平凡”平方根：1 或 -1，则 x 是对模 n 来说 1 的非平凡平方根。例如，6 是对模 35 来说 1 的非平凡平方根。下面给出定理 31.34 的一个推论，它将用于在 31.8 节中讨论的 Miller-Rabin 素数测试过程的正确性证明中。

推论 31.35　如果对模 n 存在 1 的非平凡平方根，则 n 是合数。　878

证明：根据定理 31.34 的逆否命题，如果对模 n 存在 1 的非平凡平方根，则 n 不可能是奇素数或者奇素数的幂。如果 $x^2 \equiv 1 \pmod{2}$，则 $x \equiv 1 \pmod{2}$，故 1 的所有模 2 的平方根都是平凡的。因此，n 不能是素数。最后，为了让 1 的非平凡平方根存在，必须有 $n>1$。因此，n 必定是合数。■

运用反复平方法求数的幂

数论计算中经常出现一种运算，就是求一个数的幂对另外一个数的模的运算，也称为模取幂。更准确地说，我们希望找出一种有效的方法来计算 $a^b \bmod n$ 的值，其中 a，b 为非负整数，n 为正整数。在许多素数测试子程序和 RSA 公开密钥加密系统中，模取幂运算是一种很重要的运算。当用二进制来表示 b 时，采用反复平方法，可以有效地解决这个问题。

设$\langle b_k, b_{k-1}, \cdots, b_1, b_0\rangle$是 b 的二进制表示。(亦即，二进制表示有 $k+1$ 位长，b_k 为最高有效位，b_0 为最低有效位。)下列过程随着 c 的值从 0 到 b 成倍增长，最终计算出 $a^c \bmod n$。

```
MODULAR-EXPONENTIATION(a, b, n)
1  c ← 0
2  d ← 1
3  let⟨b_k, b_{k-1}, ..., b_0⟩be the binary representation of b
4  for i ← k downto 0
5       do c ← 2c
6          d ← (d · d) mod n
7          if b_i = 1
8            then c ← c+1
9                 d ← (d · a) mod n
10 return d
```

在每次迭代中，第 6 行中平方操作的使用解释了“反复平方”名称的由来。举一个例子，对 $a=7$，$b=560$，以及 $n=561$，这个算法计算图 31-4 中给出的序列的值模 561；所用到的指数序列显示在表格的行中，以 c 标示。

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

图 31-4　当 $a=7$，$b=560=\langle 1000110000\rangle$，$n=561$ 时，MODULAR-EXPONENTIATION 计算 $a^b \pmod n$ 的结果。数值在每次 for 循环执行后显示。最终结果是 1

这个算法并不真的需要变量 c，只是用它作解释；算法维持下面两部分的循环不变式：

在第 4～9 行的 for 循环的每次迭代之前，

1. c 的值与 b 的二进制表示的前缀 $\langle b_k, b_{k-1}, \cdots, b_{i+1}\rangle$ 相同，

2. $d=a^c \bmod n$。

下面使用这个循环不变式：

初始化：在最初，$i=k$，因此前缀 $\langle b_k, b_{k-1}, \cdots, b_{i+1}\rangle$ 是空的，这对应于 $c=0$。而且，$d=1=a^0 \bmod n$。

保持：令 c' 和 d' 表示在 for 循环的一次迭代的结束处 c 和 d 的值，因此它们是下一次迭代之前的值。每次迭代更新 $c' \leftarrow 2c$（如果 $b_i=0$）或者 $c' \leftarrow 2c+1$（如果 $b_i=1$），来让 c 在下一次迭代之前是正确的。如果 $b_i=0$，则 $d'=d^2 \bmod n=(a^c)^2 \bmod n=a^{2c} \bmod n=a^{c'} \bmod n$。如果 $b_i=1$，则 $d'=d^2 a \bmod n=(a^c)^2 a \bmod n=a^{2c+1} \bmod n=a^{c'} \bmod n$。不管在哪种情况下，在下一次迭代之前 $d=a^c \bmod n$。

终止：在结束时，$i=-1$。因此，$c=b$，因为 c 的值等于 b 的二进制表示的前缀 $\langle b_k, b_{k-1}, \cdots, b_0\rangle$ 的值。因此，$d=a^c \bmod n=a^b \bmod n$。

如果输入 a，b 与 n 都是 β 位的数，则所需的算术运算的总次数是 $O(\beta)$，并且所需的位操作的总次数是 $O(\beta^3)$。

练习

879～880

31.6-1　试画出一张表以说明 $\mathbf{Z}_{11}^*$ 中每个元素的阶。找出最小的原根 g 并计算出一张表，要求写出对所有 $x \in \mathbf{Z}_{11}^*$，相应的 $\mathrm{ind}_{11,g}(x)$ 的值。

31.6-2　写出一个模取幂算法，要求该算法检查 b 的各位的顺序为从右向左，而不是从左向右。

31.6-3　假设已知 $\phi(n)$，试说明如何运用过程 MODULAR-EXPONENTIATION 计算出对任意 $a \in \mathbf{Z}_n^*$，$a^{-1} \bmod n$ 的值。

31.7　RSA 公钥加密系统

公钥加密系统（public-key cryptosystem）可以对传输于两个通信单位之间的消息进行加密，这样即使窃听者窃听到被加密的消息，也不能对加密消息进行破译。公钥加密系统还能够使通信的一方，在电子报文的末尾附加一个无法伪造的“数字签名”。这种签名是对文件上写的手写签名的电子模拟。任何人都可以容易地核对这一签名，但却无人能够伪造，如果这一消息中的任何位有所变化，整个签名就失去了效力。因此，数字签名可以作为确认签名者和其签署的信息内容的一种证明。这对于电子签署的商业性合同、电子支票、电子购买订货单和其他一些必须经过认证的电子信息来说，是一种理想的工具。

RSA 公钥加密系统主要基于以下事实：寻求大素数是很容易的，但要把一个数分解为两个大素数的积确实相当困难的。31.8 节中讲述一个能有效地找出大素数的过程，31.9 节中将讨论

大整数的分解问题。

公钥加密系统

在公钥加密系统中，每个参与者都用一把公钥和一把密钥。每把密钥都是一条信息。例如，在 RSA 公开密钥加密系统中，每个密钥均是由一对整数组成。在密码学中常把参与者“Alice”和“Bob”作为例子：用 P_A 和 S_A 分别表示 Alice 的公钥和密钥，用 P_B 和 S_B 分别表示 Bob 的公钥和密钥。

每个参与者均自己创建起公钥和密钥。密钥需要保密，但公钥则可以对任何人公开或干脆公之于众。事实上，如果每个参与者的公钥都能在一个公开目录中查到的话是很方便的，这样能使任何参加者容易地获得任何其他参加者的公钥。

公钥和密钥指定可适用于任何信息的功能函数。设$\mathcal{D}$表示允许的信息集合。例如，$\mathcal{D}$可能是所有有限长度的位序列的集合。在最简单的、原始的公钥密码学中，要求公钥与密钥说明一种从$\mathcal{D}$到其自身的一一对应的函数。对应于 Alice 的公钥 P_A 的函数用$P_A()$表示，对应于她的密钥 S_A 的函数表示成 $S_A()$，因此 $P_A()$与 $S_A()$函数都是$\mathcal{D}$的排列。假定如果已知密钥 P_A 或 S_A，就能够有效地计算出函数 $P_A()$和 $S_A()$。 881

任何参与者的公钥和密钥都是一个“匹配对”，它们指定的函数互为反函数。亦即，对任何消息 $M \in \mathcal{D}$，有

$$M = S_A(P_A(M)) \tag{31.33}$$

$$M = P_A(S_A(M)) \tag{31.34}$$

无论用哪一种次序，运用两把密钥 P_A 和 S_A 对 M 相继进行变换后，最后仍然得到消息 M。

在公钥加密系统中，重要的是除 Alice 外，没有人能在较短的时间内计算出函数 $S_A()$。送给 Alice 加密邮件的保密程度与 Alice 的数字签名的真实性均依赖于以下假设：只有 Alice 能够计算出 $S_A()$。这个要求也是 Alice 要对 S_A 保密的原因；如果她不能做到这一点，就会失去她的唯一性特性，因而加密系统也就不能把唯一性能力赋予她。即使每个人都知道 P_A，并且能够有效地计算出 $S_A()$的反函数 $P_A()$，我们依然必须保证只有 Alice 能够计算出 $S_A()$的假设成立。设计一个可行的公钥加密系统的主要困难在于解决如下问题：如何创建一个系统，在该系统中我们可以公开其变换 $P_A()$，而不至于因此而公开计算其相应的逆变换 $S_A()$的方法。

在公钥加密系统中，加密的工作方式如图 31-5 所示。假定 Bob 要发送一条加密的消息 M，使得该消息对于窃密者看起来像一串无意义的乱码。发送消息的方案如下：

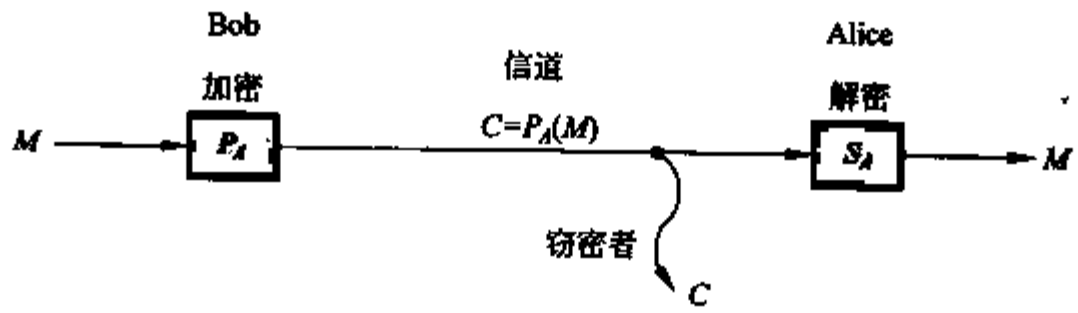


图 31-5　公钥系统的加密过程。Bob 使用 Alice 的公钥 P_A 来加密消息 M，然后传送结果的密文 $C=P_A(M)$给 Alice。一个截获传送的密文的窃密者无法得到关于 M 的信息。Alice 收到 C，并且使用密钥来解密，以得到原始消息 $M=S_A(C)$

- Bob 取得 Alice 的公钥 P_A（根据一个公开的目录或直接向 Alice 索取）。 882
- Bob 计算出相应于 M 的密文 $C=P_A(M)$，并把 C 发送给 Alice。

• 当 Alice 收到密文 C 后，她运用自己的密钥 S_A 恢复出原始信息：$M=S_A(C)$。

因为 $S_A()$和 $P_A()$互为反函数，所以 Alice 能够根据 C 计算出 M。因为只有 Alice 能够计算出 $S_A()$，所以也只有 Alice 能根据 C 计算出 M。运用 $P_A()$对 M 进行加密，可以使 M 的内容不会泄露给除 Alice 以外的任何人。

类似地，在公钥系统中可以很容易地实现数字签名。(注意到有其他方式来解决构造数字签名的问题，在这里我们不作讨论。)假设现在 Alice 希望把一个数字签署的回应 M' 发送给 Bob。数字签名的过程如图 31-6 所示。

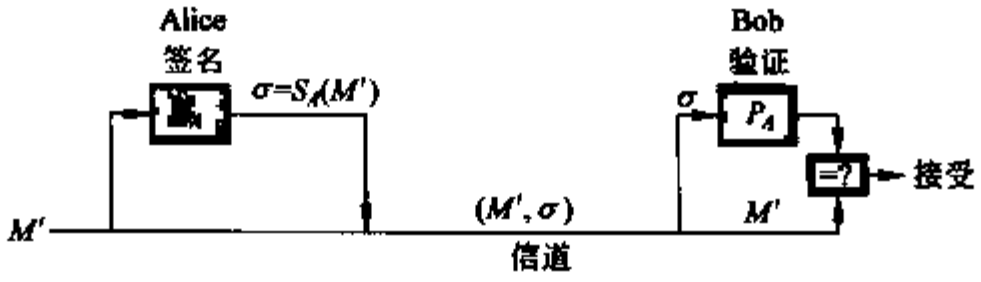


图 31-6　公钥系统的数字签名过程。Alice 通过将她的数字签名 $\sigma=S_A(M')$附加到消息 M'上，来对消息 M'签名。她将消息/签名对(M', σ)发送给 Bob，而 Bob 通过检查等式 $M'=P_A(\sigma)$来验证它。如果等式成立，则他接受(M', σ)作为 Alice 已经签名的一个消息

• Alice 运用她的密钥 S_A 计算出信息 M'的数字签名 $\sigma=S_A(M')$。

• Alice 把该消息/签名对(M', σ)发送给 Bob。

• 当 Bob 收到(M', σ)时，他可以利用 Alice 的公钥通过验证等式 $M'=P_A(\sigma)$来证实该消息的确是 Alice 发出的。(假设 M'包含有 Alice 的名字，这样 Bob 就知道应该使用谁的公钥。)如果等式成立，则 Bob 知道消息 M'确实是 Alice 签名的。如果等式不成立，那么 Bob 就得出结论，要么是信息 M'或数字签名 σ 在信息传输过程中有误，要么信息对(M', σ)就是一个有意的伪造。

883

因为数字签名既是对签署者身份的证明，也是对所签署的信息内容的证明，所以它是对文件末尾的手写签名的一种模拟。

数字签名有一条重要性质，就是它可以被任何能取得签署者的公钥的人所验证。一条签署过的信息可以被一方确认后再传送到其他各方，他们也同样能对该签名进行验证。例如，这条消息可能是 Alice 发给 Bob 的一张电子支票。当 Bob 确认了支票上 Alice 的签名后，他可以把这张支票送交银行，而银行也可以对签名进行验证，然后调拨相应的资金。

注意，被签署的信息并没有加密；该信息是“公开的”，没有受到保护。如果把上述有关加密和签名的两种方案结合起来使用，就可以构造出同时被签署和加密的消息。签署者首先把他的数字签名附加在消息的后面，然后再用他希望的接收者的公钥对得到的消息/签名对进行加密。接收者运用他的密钥对收到的消息进行解密，以同时获得原始消息和数字签名，然后，他可以用签署者的公钥对签名进行验证。在日常的纸张文件系统相应的过程为：对文件签名后，把文件封入一个纸质信封内，该信封只能由希望的接收者打开。

RSA 加密系统

在 RSA 公钥加密系统中，一个参加者按下列过程来创建他的公钥与密钥。

1. 随机选取两个大素数 p 和 q，且 $p\neq q$，例如素数 p 和 q 可能各有 512 位(二进制)。
2. 根据式 $n=pq$ 计算出 n 的值。
3. 选取一个与 $\phi(n)$互质的小奇数 e，其中根据等式(31.19)知 $\phi(n)=(p-1)(q-1)$。

4. 对模 $\phi(n)$，计算出 e 的乘法逆元 d 的值。（推论 31.26 保证 d 存在并且唯一。已知 e 和 $\phi(n)$，可以利用 31.4 节中的技术来计算 d）。

5. 输出对 $P=(e, n)$，把它作为 RSA 公钥。

6. 把对 $S=(d, n)$保密，并把它作为 RSA 密钥。

对这个方案来说，域 D为集合 $\mathbf{Z}_n$。与公钥 $P=(e, n)$相关联的消息 M 的变换为

$$P(M) = M^e \pmod{n} \tag{31.35}$$

与密钥 $S=(d, n)$相关联的密文 C 的相应变换为 884

$$S(C) = C^d \pmod{n} \tag{31.36}$$

这两个等式对加密与签名都适用。为了创建一个签名，签署人把其密钥应用于签署的消息而不是密文。为了确认签名，可以对签名而不是对被加密的消息应用签署人的公钥。

我们运用 31.6 节中描述的过程 MODULAR-EXPONENTIATION，来实现上述公钥与密钥的有关操作。为了分析这些操作的运行时间，假定公钥(e, n)和密钥(d, n)满足 $\lg e = O(1)$，$\lg d \leqslant \beta$ 以及 $\lg n \leqslant \beta$。则应用公钥操作，需要执行 $O(1)$次模乘法运算和 $O(\beta^2)$次位操作。应用密钥操作，需要执行 $O(\beta)$次模乘法运算和 $O(\beta^3)$次位操作。

定理 31.36（RSA 的正确性） RSA 等式(31.35)和式(31.36)定义了满足等式(31.33)和式(31.34)的 $\mathbf{Z}_n$ 上的逆变换。

证明：根据等式(31.35)和式(31.36)，对任意 $M \in \mathbf{Z}_n$，有

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n}$$

因为 e 和 d 是关于模 $\phi(n)=(p-1)(q-1)$的乘法的逆，所以对某个整数 k，有

$$ed = 1 + k(p-1)(q-1)$$

但是，如果 $M \not\equiv 0 \pmod{p}$，有

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} && \pmod{p} \\ &\equiv M(1)^{k(q-1)} && \pmod{p} \qquad \text{（根据定理 31.31）} \\ &\equiv M && \pmod{p} \end{aligned}$$

同样，如果 $M \equiv 0 \pmod{p}$，有 $M^{ed} \equiv M \pmod{p}$。因此，对所有 M，

$$M^{ed} \equiv M \pmod{p}$$

类似地，对所有 M，

$$M^{ed} \equiv M \pmod{q}$$

因此，根据中国余数定理的推论 31.29，对所有 M，有

$$M^{ed} \equiv M \pmod{n}$$

■

RSA 加密系统的安全性主要来源于对大整数进行因子分解的困难性。如果对方能对公钥中的模 n 进行分解，他就可以根据公钥推导出密钥，并可以与公钥的创建者用同样的方法，来运用 885
因子 p 和 q。因此，如果能够容易地分解大整数，也就能够容易地打破 RSA 加密系统。这一命题的逆命题就是如果分解大整数是难的，则打破 RSA 也是难的，经过二十多年的研究，人们还没有发现比分解模 n 更容易的方法来打破 RSA 加密系统。并且正如我们将在 31.9 节中看到的那样，对大整数进行分解的困难程度令人惊异。通过随机地选取两个 512 位的素数并求出它们的积，就可以创建出一把用现行技术无法在可行的时间内“破解”的公钥。在目前数论算法的设计方法还没有根本突破的情况下，当细心地遵循所建议的标准来实现时，RSA 加密系统可以为实际应用提供高度的安全性。

为了通过 RSA 加密系统获得安全性，建议在数百位长度的整数上操作，以防御因子分解技术可能的突破。在 2001 年，RSA 模数通常是在 768 到 2048 位的范围内。要建立这样大小的模

数，必须能够有效地找出大素数。31.8 节将讨论这个问题。

为了效率，通常运用一种“混合的”或“密钥管理”模式的 RSA，来实现快速的无公钥加密系统。在这样一个系统中，加密密钥与解密密钥是相同的。如果 Alice 希望私下把一条很长的消息 M 发送给 Bob，在无公钥加密系统中，她选取一把随机密钥 K，然后运用 K 对 M 进行加密，得到密文 C。这里 C 和 M 一样长，但 K 是相当短的。然后，她利用 Bob 的公开 RSA 密钥对 K 进行加密。因为 K 很短，所以计算 $P_B(K)$的速度也很快(比计算 $P_B(M)$的速度快很多)。然后，她把$(C, P_B(K))$传送给 Bob，Bob 对 $P_B(K)$解密后得到 K，然后再用 K 对 C 进行解密从而得到 M。

类似地，我们也经常使用一种“混合的”方法来提高数字签名的执行效率。在这种方法中，我们使 RSA 与一个公开的单向散列函数 h 相结合，其中的单向散列函数是易于计算的，但是对这种函数来说，要找出两条消息 M 和 M'满足：$h(M)=h(M')$，这在计算上是不可行的。$h(M)$的值是消息 M 的一个短的(譬如 160 位)“指纹”。如果 Alice 希望签署一条消息 M，她首先把函数 h 作用于 M 得到指纹$h(M)$，然后，她用密钥来加密 $h(M)$。她把$(M, S_A(h(M)))$作为她签署的 M 的版本发送给 Bob。Bob 可以通过计算 $h(M)$，然后验证用 P_A 作用于他收到的 $S_A(h(M))$，看是否等于 $h(M)$来验证签名的真实性。因为没有人能够构造出具有相同指纹的两条消息，所以不可能既改变了签署的消息，又能保持签名的合法性。

最后，我们注意到利用证书可以更容易地分配公钥。例如，假设存在一个“可信的权威”T，
886 每个人都知道他的公钥。Alice 可以从 T 获取一条签署的消息(她的证书)，“Alice 的公钥是 P_A”。由于每个人都知道 P_T，所以这个证书是一个“自我认证”。Alice 可以将她的证书包含在签名信息中，这样就可以使接收者立即得到 Alice 的公钥，以便验证她的签名。因为她的密钥是被 T 签署的，所以接收者就知道 Alice 的密钥确实是 Alice 本人的密钥。

练习

31.7-1 考察一个 RSA 密钥集合，其中 $p=11$，$q=29$，$n=319$，$e=3$。在密钥中用到的 d 值应当是多少？对消息 $M=100$ 加密后得到什么消息？

31.7-2 证明：如果 Alice 的公开指数 e 等于 3，并且对方获得了 Alice 的秘密指数 d。则对方能够在关于 n 的位数的多项式时间内对 Alice 的模 n 进行分解。(尽管我们不要证明下列结论，但读者也许会对下列事实感兴趣：即使不知道条件 $e=3$，上述结论仍然成立。参见 Miller[221]。)

***31.7-3** 证明：在如下意义上说，RSA 是乘法的：

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n}$$

利用这个事实证明：如果对方有一个过程，对从 $\mathbf{Z}_n$ 中随机选取的并用 P_A 加密的消息，它能够有效地解密出其中的百分之一，那么他就可以运用一种概率性算法，以一个较大的概率，对用 P_A 加密的每一条信息进行解密。

*31.8 素数的测试

在本节中，我们要考虑寻找大素数的问题。首先讨论素数的密度，接着讨论一种似乎可行的(但不完全)测试素数的方法，然后，介绍由 Miller 和 Rabin 发现的有效的随机素数测试算法。

素数的密度

887 在很多应用领域(如密码学)中，需要找出大的“随机”素数。幸运的是，大素数并不算太少，

因此测试适当的随机整数，直至找到素数的过程也不是太费时的。素数分布函数 $\pi(n)$描述了小于或等于 n 的素数的数目。例如 $\pi(10)=4$，因为小于或等于 10 的素数有 4 个，分别为 2，3，5，7。素数定理给出了 $\pi(n)$的一个有用的近似。

定理 31.37(素数定理)

$$\lim_{n\to\infty}\frac{\pi(n)}{n/\ln n}=1 \qquad \blacksquare$$

即使对于较小的 n，近似计算式 $n/\ln n$ 可以给出 $\pi(n)$相当精确的估计值。例如，当 $n=10^9$ 时，其误差不超过 6%，这时 $\pi(n)=50\ \ 847\ \ 534$，且 $n/\ln n\approx 48\ \ 254\ \ 942$。(对研究数论的人来说，$10^9$ 是一个小数字。)

运用素数定理，可以估计出一个随机选取的整数 n 是素数的概率为 $1/\ln n$。因此，为了找出一个长度与 n 相同的素数，大约要检查在 n 附近随机选取的 $\ln n$ 个整数。例如，为了找出一个 512 位长的素数，大约需要对 $\ln 2^{512}\approx 355$ 个随机选取的 512 位长的整数进行素数测试。(通过只选择奇数，就可以把这个数字减少一半。)

在本节的余下部分，我们来考虑确定一个大的奇数 n 是否是素数的问题。为了表示上的方便，假定 n 具有下列素数分解因子：

$$n=p_1^{e_1}p_2^{e_2}\cdots p_r^{e_r} \tag{31.37}$$

其中 $r\geqslant 1$，p_1，p_2，…，p_r 是 n 的素数因子，且 e_1，e_2，…，e_r 是正整数。当然，n 是素数当且仅当 $r=1$ 并且 $e_1=1$。

解决这个素数测试问题的一种简便方法是试除。我们试着用每个整数 2，3，…，$\lfloor\sqrt{n}\rfloor$分别去除 n。(大于 2 的偶数可以跳过)。很容易看出，n 是素数当且仅当没有一个试除数能整除 n。假定每次试除需要常数时间，则最坏情况下运行时间是 $\Theta(\sqrt{n})$，这是 n 的长度的幂。(回顾一下，如果 n 表示成 β 位的二进制数，则 $\beta=\lceil\lg(n+1)\rceil$，因此$\sqrt{n}=\Theta(2^{\beta/2})$。)因此，只有当 n 很小或者 n 恰好有一个较小的素数因子时，试除法才能较好地执行。当执行试除法时，它的优点是它不仅能确定 n 是素数还是合数，而且当 n 是合数时，它确定出 n 的一个素数因子。

在本节中，我们所感兴趣的仅仅是确定一个指定的数 n 是否是素数；如果 n 是一个合数，我们将不关心找出其素数因子。正如将在 31.9 节中看到的那样，计算一个数的素数因子分解的计算过程的代价是高昂的。让人惊讶的是，确定一个数是否是素数，要比确定一个合数的素数因子分解容易得多。 888

伪素数测试过程

现在来考察一种“几乎可行”的素数测试方法，事实上，对很多实际应用来说，这种方法已经是一种相当好的方法了。后面还将对这个方法作精心的改进，以消除其中存在的小的缺陷。设 $\mathbf{Z}_n^+$ 表示 $\mathbf{Z}_n$ 中的非零元素：

$$\mathbf{Z}_n^+=\{1,2,\cdots,n-1\}$$

如果 n 是素数，则 $\mathbf{Z}_n^+=\mathbf{Z}_n^*$。

如果 n 是一个合数，而且

$$a^{n-1}\equiv 1 \pmod n \tag{31.38}$$

则说 n 是一个基为 a 的伪素数。费马定理(定理 31.31)蕴含着如果 n 是一个素数，则对 $\mathbf{Z}_n^+$ 中的每一个 a，n 满足等式(31.38)。因此，如果能找出任意的 $a\in\mathbf{Z}_n^+$，使得 n 不满足等式(31.38)，那么 n 当然就是合数。让人吃惊的是，这个命题的逆命题也几乎成立，因此，这一衡量标准几乎是素数测试的正确标准。对 $a=2$，测试看 n 是否满足等式(31.38)。如果不满足，则可以说 n 是

合数。否则，输出一个猜测：n是素数(实际上，此时我们所知道的只是n或者是素数，或者是基于2的伪素数)。

下列过程就是用这种方法测试素数n的过程。它利用31.6节中的过程MODULAR-EXPONETIATION。假设输入n是一个大于2的整数。

```
PSEUDOPRIME(n)
1  if MODULAR-EXPONENTIATION(2, n−1, n)≠1 (mod n)
2     then return COMPOSITE          ▷ Definitely.
3     else return PRIME              ▷ We hope!
```

这个过程可能会产生错误，但是只有一种类型。亦即，如果它判定n是合数，则结果总是正确的。但如果它判定n是素数，那么只有当n是基于2的伪素数时过程才出错。

这个过程出错的概率有多大？机会非常少。在小于10 000的n值中，只有在其中22个值上会产生错误。前面四个这样的值分别为341，561，645和1 105。可以证明当$\beta\to\infty$时，该过程对随机选取的β位数进行测试时，发生错误的概率趋于0。如果像Pomerance[244]那样，对给定规模的基于2的伪素数的数目能做出更加精确的估计，就可以得到被上述过程称为素数的一个随机选取的512位数，是基于2的伪素数的概率不到$1/10^{20}$，而被上述过程称为素数的一个随机选取
889 的1024位数，是基于2的伪素数的概率不到$1/10^{41}$。因此，如果只是试着为某个应用找到一个大的素数，通过随机选取大的数字，直到它们其中之一使得PSEUDOPRIME输出PRIME，在所有实际的用途中几乎永远不会出错。但是当测试素数的数字不是随机选取的时候，就需要一个更好的方法来进行素数测试。后面可以看到，如果能稍微巧妙一点，再加上一些随机性，就会得到一个在所有输入上都工作良好的素数测试程序。

遗憾的是，我们不能仅通过选取另外一个基数(例如$a=3$)检查等式(31.38)的方法，来消除所有的出错机会，因为对所有$a\in\mathbb{Z}_n^*$，总存在满足等式(31.38)的合数n。这些整数被称为Carmichael数。前三个Carmichael数是561，1 105和1 729。Carmichael数是非常少的；例如，在小于100 000 000的数中，只有255个Carmichael数。练习31.8-2解释了这种数很少的原因。

下一步来说明如何对素数测试方法进行改进，以使测试过程不会把Carmichael数当成素数。

Miller-Rabin随机性素数测试方法

Miller-Rabin素数测试方法对简单测试过程PSEUDOPRIME做了两点改进，从而解决了其中存在的问题：

- 它试验了数个随机选取的基值a，而不是仅仅试验一个基值。
- 当计算每个模取幂的值时，注意在最后一组平方里是否发现了对模n来说1的非平凡平方根。如果发现这样的根存在，终止执行并输出结果COMPOSITE。推论31.35说明了用这种方法检测合数是正确的。

下面是Miller-Rabin素数测试的代码。输入$n>2$是一个奇数，该过程测试它是否为素数，s是从$\mathbb{Z}_n^+$中随机选取的要进行试验的基值的个数。代码中运用5.1节中的随机数生成程序RANDOM：RANDOM(1，$n-1$)返回一个$1\leqslant a\leqslant n-1$的随机选取的整数$a$。代码中还使用一个辅助过程WITNESS，WITNESS(a，n)为TRUE当且仅当a是合数n的“证据”，即，用a来证明(其证明方法将在后面给出)n是合数是可能的。测试WITNESS(a，n)是对作为过程PSEUDOPRIME的基础(用$a=2$)的测试

$$a^{n-1}\not\equiv 1 \pmod n$$

的一个扩展，但是更加有效。我们首先要介绍并证明一下WITNESS的构造过程，然后说明如何

把它应用于 Miller-Rabin 素数测试过程。令 $n-1=2^t u$，其中 $t \geqslant 1$ 且 u 是奇数；亦即，$n-1$ 的二进制表示是奇数 u 的二进制表示后面跟上 t 个零。因此，$a^{n-1}=(a^u)^{2^t} \pmod n$，所以可以通过先计算 $a^u \bmod n$，然后对结果连续平方 t 次来计算 $a^{n-1} \bmod n$。 890

```
WITNESS(a, n)
1  let n-1=2^t u, where t≥1 and u is odd
2  x_0 ← MODULAR-EXPONENTIATION(a, u, n)
3  for i ← 1 to t
4       do x_i ← x_{i-1}^2 mod n
5          if x_i = 1 and x_{i-1} ≠ 1 and x_{i-1} ≠ n - 1
6             then return TRUE
7  if x_i ≠ 1
8    then return TRUE
9  return FALSE
```

WITNESS 的这个伪代码通过首先在第 2 行计算值 $x_0=a^u \bmod n$，然后在第 3～6 行的 for 循环的一行中对结果平方 t 次，来计算 $a^{n-1} \bmod n$。通过在 i 上归纳，所计算的序列 x_0，x_1，…，x_t 的值满足等式 $x_i \equiv a^{2^i u} \pmod n$ $(i=0, 1, \cdots, t)$，所以特别地，$x_t \equiv a^{n-1} \pmod n$。然而每当在第 4 行执行一个平方步骤时，如果第 5～6 行侦测到 1 的一个非平凡平方根刚刚被发现，则循环可能提前结束。如果这样，则算法终止并返回 TRUE。如果 $x_t \equiv a^{n-1} \pmod n$ 所计算的值不等于 1，则第 7～8 行返回 TRUE，如同在这个情况中 PSEUDOPRIME 返回 COMPOSITE 一样。如果我们在第 6 行或第 8 行没有返回 TRUE，则第 9 行返回 FALSE。

现在来论证如果 WITNESS(a, n)返回 TRUE，则可以用 a 构造出 n 是合数的证明。

如果 WITNESS 从第 8 行返回 TRUE，则它已经发现了 $x_t=a^{n-1} \bmod n \neq 1$。然而，如果 n 是素数，则根据费马定理(定理 31.31)有对任何 $a \in \mathbf{Z}_n^+$，$a^{n-1} \equiv 1 \pmod n$ 成立。因此，n 不可能为素数，并且等式 $a^{n-1} \bmod n \neq 1$ 就是对这一事实的证明。

如果 WITNESS 从第 6 行返回 TRUE，则它已经发现了对模 n 来说，x_{i-1} 是 x_i-1 的一个非平凡平方根，这是因为有 $x_{i-1} \neq \pm 1 \pmod n$，但 $x_i \equiv x_{i-1}^2 \equiv 1 \pmod n$。推论 31.35 说明仅当 n 是合数时，对模 n 来说才可能存在 1 的非平凡平方根，因此证明了对模 n 来说 x_{i-1} 是 1 的非平凡平方根，也就证明了 n 是合数。

这样就完成了有关 WITNESS 正确性的证明。如果调用 WITNESS(a, n)得到输出为 TRUE，则 n 必为合数，并且可以很容易地从 a 和 n 确定 n 是合数的证明。

在这里，我们以序列 $X=\langle x_0, x_1, \cdots, x_t \rangle$ 的函数形式简短展示 WITNESS 的行为的另一种描述，稍后在分析 Miller-Rabin 素数测试的效率时可能会发现它很有用。注意如果对某个 $0 \leqslant i < t$ 有 $x_i=1$，则 WITNESS 可能不会计算序列的余下部分。然而如果计算，x_{i+1}，x_{i+2}，…，x_t 的值都将是 1，而且我们考虑序列 X 中这些位置开始时都是 1。有四种情况： 891

1) $X=\langle \cdots, d \rangle$，其中 $d \neq 1$：序列 X 不是以 1 结尾。返回 TRUE；a 是 n 为合数的证据(由费马定理)。

2) $X=\langle 1, 1, \cdots, 1 \rangle$：序列 X 全都是 1。返回 FALSE；a 不是 n 为合数的证据。

3) $X=\langle \cdots, -1, 1, \cdots, 1 \rangle$：序列 X 以 1 结尾，而且最后一个非 1 的数是 -1。返回 FALSE；a 不是 n 为合数的证据。

4) $X=\langle \cdots, d, 1, \cdots, 1 \rangle$，其中 $d \neq \pm 1$：序列 X 以 1 结尾，但最后一个非 1 的数不是 -1。返回 TRUE；a 是 n 为合数的证据，因为 d 是 1 的一个非平凡平方根。

现在来看利用 WITNESS 的 Miller-Rabin 素数测试过程。再一次，假设 n 是大于 2 的一个奇整数。

```
MILLER-RABIN(n, s)
1  for j ← 1 to s
2    do a ← RANDOM(1, n−1)
3       if WITNESS (a, n)
4          then return COMPOSITE    ▷确定.
5  return PRIME                     ▷几乎肯定.
```

过程 MILLER-RABIN 是为了证明 n 是合数所进行的概率性搜索过程。主循环(从第 1 行开始)从 $\mathbf{Z}_n^+$ 中挑选 s 个 a 的随机数值(第 2 行)。如果所挑选的一个 a 值是 n 为合数的证据，则过程 MILLER-RABIN 在第 4 行输出 COMPOSITE。这样的输出总是正确的，这一点由 WITNESS 的正确性证明可以看出。如果在 s 次试验中没有发现证据，则 MILLER-RABIN 假定这是因为证据没被发现，因此假设 n 为素数。我们将看到如果 s 足够大，则这个输出结果很可能是正确的，但也存在这样一种很小的可能性，即过程在选择 a 时运气不佳，因为过程虽然没有发现证据，但证据却确实存在。

为了说明 MILLER-RABIN 的操作过程，设 n 为 Carmichael 数 561，因此 $n-1=560=2^4 \cdot 35$。假定选择 $a=7$ 被选作基，图 31-4 说明 WITNESS 计算 $x_0 \equiv a^{35} \equiv 241 \pmod{561}$，因此计算序列 $X=\langle 241, 298, 166, 67, 1\rangle$。因此，在最后一次平方时发现了 1 的非平凡平方根，因为 $a^{280} \equiv 67 \pmod{n}$，$a^{560} \equiv 1 \pmod{n}$。因此 $a=7$ 是 n 为合数的证据，WITNESS(7, n)返回 TRUE，因而 MILLER-RABIN 返回 COMPOSITE。

892

如果 n 是一个 β 位数，则 MILLER-RABIN 需要执行 $O(s\beta)$ 次算术运算和 $O(s\beta^3)$ 次位操作，这是因为从渐近意义上说，它需要执行的工作仅是 s 次模取幂运算。

Miller-Rabin 素数测试的出错率

如果 Miller-Rabin 输出 PRIME，则它仍有一种很小的可能性会产生错误。但是，与 PSEUDOPRIME 不同的是，出错的可能性并不依赖于 n；对该过程也不存在坏的输入。相反地，它取决于 s 的大小和在选取基值 a 时"抽签的运气"。同时，由于每次测试比对等式(31.38)作简单的检查更严格，因此从总的原则上，对随机选取的整数 n，其出错率应该是很小的。下列定理阐述了一个更精确的论点。

定理 31.38　如果 n 是一个奇合数，则 n 为合数的证据的数目至少为 $(n-1)/2$。

证明：证明过程说明了非证据的数目不多于 $(n-1)/2$，蕴含着定理成立。

首先，我们断言任何非证据都必须是 $\mathbf{Z}_n^*$ 的一个成员。为什么呢？考虑任意的非证据 a。它必须满足 $a^{n-1} \equiv 1 \pmod{n}$，或是等价地，$a \cdot a^{n-2} \equiv 1 \pmod{n}$。因此，方程 $ax \equiv 1 \pmod{n}$有一个解，即 a^{n-2}。由推论 31.21 可知，$\gcd(a, n) \mid 1$，这反过来意味着 $\gcd(a, n)=1$。因此，a 是 $\mathbf{Z}_n^*$ 的一个成员；所有的非证据都属于 $\mathbf{Z}_n^*$。

为了完成证明，我们说明不只是所有的非证据都包含在 $\mathbf{Z}_n^*$ 内，而且它们都包含在 $\mathbf{Z}_n^*$ 的一个真子群 B 中(回顾一下，如果 B 是 $\mathbf{Z}_n^*$ 的一个子群但 B 不等于 $\mathbf{Z}_n^*$，则说 B 是 $\mathbf{Z}_n^*$ 的一个真子群)。根据推论 31.16，有 $|B| \leqslant |\mathbf{Z}_n^*|/2$。因为 $|\mathbf{Z}_n^*| \leqslant n-1$，所以有 $|B| \leqslant (n-1)/2$。因此，非证据的数目至多是 $(n-1)/2$，所以证据的数目必须至少有 $(n-1)/2$。

下面来说明如何找出 $\mathbf{Z}_n^*$ 的包含所有非证据的真子群 B。具体分两种情况来讨论。

情况 1：存在一个 $x \in \mathbf{Z}_n^*$，使得

$$x^{n-1} \not\equiv 1 \pmod{n}$$

换句话说，n 不是一个 Carmichael 数。如我们早先所注意到的，因为 Carmichael 数是非常少的，情况 1 是由"实际"所产生的主要情况(例如，当 n 已经被随机选取，而且被测试其素数性)。

设 $B=\{b\in \mathbf{Z}_n^* : b^{n-1}\equiv 1 \pmod{n}\}$。显然 B 是非空的，因为 $1\in B$。因为 B 在模 n 的乘法下是封闭的，根据定理 31.14 可知，B 是 $\mathbf{Z}_n^*$ 的一个子群。注意每个非证据都属于 B，因为非证据 a 满足 $a^{n-1}\equiv 1 \pmod{n}$。因为 $x\in \mathbf{Z}_n^* - B$，所以 B 是 $\mathbf{Z}_n^*$ 的一个真子群。 893

情况 2：对所有的 $x\in \mathbf{Z}_n^*$，

$$x^{n-1} \equiv 1 \pmod{n} \tag{31.39}$$

换句话说，n 是一个 Carmichael 数。这个情况实际上非常稀少。然而正如我们现在要说明的，MILLER-RABIN 测试(不同于伪素数测试)可以有效地确定 Carmichael 数的合数性。

在这种情况下，n 不可能是一个素数幂。为搞清楚为什么，反之我们假设 $n=p^e$，其中 p 是一个素数，$e>1$。按如下方式推导出矛盾。因为 n 假设是奇数，故 p 也必须是奇数。定理 31.32 蕴含着 $\mathbf{Z}_n^*$ 是一个循环群：它包含一个生成元 g，满足 $\mathrm{ord}_n(g)=|\mathbf{Z}_n^*|=\phi(n)=p^e(1-1/p)=(p-1)p^{e-1}$。根据式(31.39)，有 $g^{n-1}\equiv 1 \pmod{n}$。则离散对数定理(定理 31.33，取 $y=0$)意味着 $n-1\equiv 0 \pmod{\phi(n)}$，或者

$$(p-1)p^{e-1} \mid p^e-1$$

这对 $e>1$ 是一个矛盾，因为 $(p-1)p^{e-1}$ 可以被素数 p 整除，但是 p^e-1 不能。因此，n 不是一个素数幂。

因为奇合数 n 不是一个素数幂，我们把它分解成一个积 n_1n_2，其中 n_1 和 n_2 都是大于 1 的奇数且互质。(有数种方法来做这个分解，选择哪一种并没有关系。例如，如果 $n=p_1^{e_1}p_2^{e_2}\cdots p_r^{e_r}$，则我们可以选择 $n_1=p_1^{e_1}$，而 $n_2=p_2^{e_2}p_3^{e_3}\cdots p_r^{e_r}$。)

回顾一下，我们定义 t 和 u 使得 $n-1=2^tu$，其中 $t\geqslant 1$，u 是奇数，且对于输入 a，过程 WITNESS 计算序列

$$X=\langle a^u, a^{2u}, a^{2^2u}, \cdots, a^{2^tu}\rangle$$

(所有的计算都是根据模 n 计算的)。

让我们称一对整数(v, j)为可接受的，如果 $v\in \mathbf{Z}_n^*$，$j\in\{0, 1, \cdots, t\}$且

$$v^{2^ju}\equiv -1 \pmod{n}$$

可接受的对是肯定存在的，因为 u 是奇数；可以选择 $v=n-1$ 和 $j=0$，来让$(n-1, 0)$是一个可接受对。现在挑选最大可能的 j，使得存在一个可接受对(v, j)，调整 v 来让(v, j)是一个可接受对。设

$$B=\{x\in \mathbf{Z}_n^* : x^{2^ju}\equiv \pm 1 \pmod{n}\}$$

因为 B 是在模 n 的乘法下封闭的，故它是 $\mathbf{Z}_n^*$ 的一个子群。因此由推论 31.16，$|B|$ 整除 $|\mathbf{Z}_n^*|$。每一个非证据都必定是 B 的成员，因为由一个非证据产生的序列 X 必须全部为 1，或者在第 j 个位置之前，包含一个 -1(根据 j 的最大性)。(如果(a, j')是可接受的，其中 a 是一个非证据，则根据我们选择 j 的方式，必有 $j'\leqslant j$。) 894

现在，利用 v 的存在性来说明存在一个 $w\in \mathbf{Z}_n^* - B$。因为 $v^{2^ju}\equiv -1 \pmod{n}$，根据中国余数定理的推论 31.29，有 $v^{2^ju}\equiv -1 \pmod{n_1}$。根据推论 31.28，同时存在一个 w 满足

$$w\equiv v \pmod{n_1}$$
$$w\equiv 1 \pmod{n_2}$$

于是

$$w^{2^i u} \equiv -1 \pmod{n_1}$$

$$w^{2^i u} \equiv 1 \pmod{n_2}$$

由推论 31.29 可知，$w^{2^i u} \not\equiv 1 \pmod{n_1}$意味着 $w^{2^i u} \not\equiv 1 \pmod{n}$，而且 $w^{2^i u} \not\equiv -1 \pmod{n_2}$意味着 $w^{2^i u} \not\equiv -1 \pmod{n}$。因此，$w^{2^i u} \not\equiv \pm 1 \pmod{n}$，所以 $w \notin B$。

接下来还要证明 $w \in \mathbb{Z}_n^*$。首先分别对模 n_1 和模 n_2 进行处理。对模 n_1，注意到因为 $v \in \mathbb{Z}_n^*$，有 $\gcd(v, n)=1$，因此 $\gcd(v, n_1)=1$；如果 v 与 n 没有任何公约数，它当然不会与 n_1 有任何公约数。因为 $w \equiv v \pmod{n_1}$，所以 $\gcd(w, n_1)=1$。对模 n_2，$w \equiv 1 \pmod{n_2}$意味着 $\gcd(w, n_2)=1$。合并这些结果，利用推论 31.6，即 $\gcd(w, n_1 n_2)=\gcd(w, n)=1$。亦即 $w \in \mathbb{Z}_n^*$。

因此，$w \in \mathbb{Z}_n^* - B$，而且我们以 B 是 $\mathbb{Z}_n^*$ 的一个真子群的结论完成情况 2。

在两种情况中的任何一个，我们看出 n 为合数的证据的数目都至少为$(n-1)/2$。 ■

定理 31.39　对任意奇数 $n>2$ 和正整数 s，MILLER-RABIN(n, s)出错的概率至多为 2^{-s}。

证明：利用定理 31.38，我们看到如果 n 是合数，则每次执行第 1～4 行的 for 循环，发现 n 为合数的证据 x 的概率至少为 1/2。只有当 MILLER-RABIN 运气太差，在主循环总共 s 次迭代中，每一次都没能发现 n 为合数的证据时，过程才会出错。而这种每次都错过发现证据的概率至多为 2^{-s}。 ■

因此，对可以想象得到的几乎所有的应用，选取 $s=50$ 应该是足够的。如果试图对随机选取
895 的大整数，应用 MILLER-RABIN 来找出大素数，则可以论证(在此不做证明)选取较小的 s 值(如 3)也未必导致错误的结论。即对一个随机选取的奇合数 n，n 为合数的非证据的预计数目可能要比$(n-1)/2$ 少得多。但是，如果整数 n 是随机选取的，则运用经过改进的定理 31.39，所能证明的最佳结论是非证据数目至多为$(n-1)/4$。而且确实存在整数 n，使得非证据的数目就是$(n-1)/4$。

练习

31.8-1　证明：如果一个奇整数 $n>1$ 不是素数或素数的幂，则对模 n 存在一个 1 的非平凡平方根。

***31.8-2**　可以稍稍把欧拉定理加强为以下形式，对所有 $a \in \mathbb{Z}_n^*$，有

$$a^{\lambda(n)} \equiv 1 \pmod{n}$$

其中 $n=p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$，且 $\lambda(n)$定义为

$$\lambda(n) = \operatorname{lcm}(\phi(p_1^{e_1}), \cdots, \phi(p_r^{e_r})) \tag{31.40}$$

证明 $\lambda(n) \mid \phi(n)$。如果 $\lambda(n) \mid n-1$，则合数 n 为 Carmichael 数。最小的 Carmichael 数为 $561=3 \cdot 11 \cdot 17$；这里，$\lambda(n)=\operatorname{lcm}(2, 10, 16)=80$，它可以整除 560。证明 Carmichael 数必须既是“无平方数”(不能被任何素数的平方所整除)，又是至少三个素数的积。因为这个原因，所以 Carmichael 数不是很普遍的。

31.8-3　证明：如果对模 n，x 是 1 的非平凡平方根，则 $\gcd(x-1, n)$和 $\gcd(x+1, n)$都是 n 的非平凡约数。

*31.9　整数的因子分解

假设我们希望将一个整数 n 分解为素数的积。通过上面一节所讨论的素数测试，可以知道 n 是否是合数，但如果 n 是合数，它并不能指出 n 的素数因子。对一个大整数 n 进行因子分解，似

乎要比仅确定 n 是素数还是合数困难得多。即使用当今的超级计算机和现行的最佳算法，要对一个 1024 位的数进行因子分解也还是不可行的。

896

Pollard 的 rho 启发式方法

对到 B 的所有整数进行试除，可以保证完全获得到 B^2 的任意数的因子分解。下列过程做相同的工作量，就能对到 B^4 的任意数进行因子分解(除非运气不佳)。由于该过程仅仅是一种启发性方法，因此既不能保证其运行时间也不能保证其运行成功，不过在实际应用中该过程还是非常有效的。POLLARD-RHO 过程的另一个优点是，它只使用固定量的存储空间。(可以很容易地在一个可编程的掌上计算器上实现 POLLARD-RHO，来找出小数字的因子。)

```
POLLARD-RHO(n)
 1  i ← 1
 2  x_1 ← RANDOM(0, n−1)
 3  y ← x_1
 4  k ← 2
 5  while TRUE
 6    do i ← i+1
 7       x_i ← (x_{i−1}^2 − 1) mod n
 8       d ← gcd(y−x_i, n)
 9       if d≠1 and d≠n
10         then print d
11       if i=k
12         then y ← x_i
13              k ← 2k
```

该过程的执行过程如下。第 1～2 行把 i 初始化为 1，把 x_1 初始化为 $\mathbb{Z}_n$ 中随机选取的一个值。第 5 行开始的 while 循环将一直进行迭代，来搜索 n 的因子。在 while 循环的每一次迭代中，递归式

$$x_i \leftarrow (x_{i-1}^2 - 1) \bmod n \tag{31.41}$$

在第 7 行中用于计算下列无穷序列

$$x_1, x_2, x_3, x_4, \cdots \tag{31.42}$$

中 x_i 的下一个值，i 的值在第 6 行中进行相应的增加。为清楚起见，代码中使用了下标变量 x_i，但即使去掉所有的下标，程序也以同样的方式执行，因为仅需要保留最近计算出的 x_i 的值。使用这个修改，此过程只使用了一个常量的存储空间。

程序不时地把最近计算出的 x_i 值存入变量 y 中。具体来说，存储的值是那些下标为 2 的幂的变量：

897

$$x_1, x_2, x_4, x_8, x_{16}, \cdots$$

第 3 行中保存值 x_1，每当 $i=k$ 时，第 12 行就保存值 x_k。在第 4 行中变量 k 被初始化为 2，并且每当更新 y 后，第 13 行中就把 k 的值增加一倍。因此，k 值的序列为 1，2，4，8，…，并且总是给出要存入 y 的下一个值 x_k 的下标。

第 8～10 行试图用存入 y 的值和 x_i 的当前值找出 n 的一个因子。特别是，第 8 行计算出最大公约数 $d=\gcd(y-x_i, n)$。如果 d 是 n 的非平凡约数(在第 9 行中进行检查)，则第 10 行输出 d 的值。

起初，这一寻找因子分解的过程似乎有点神秘。但是注意，POLLARD-RHO 决不会输出错误的答案；它输出的任何数都是 n 的非平凡约数。当然，POLLARD-RHO 也可能不输出任何信

息，并没有保证它能产生任何结果。不过，我们将看到，我们有充分的理由预计，POLLARD-RHO 在 while 循环执行大约 $\Theta(\sqrt{p})$ 次迭代后，会输出 n 的一个因子 p。因此，如果 n 是合数，则在大约经 $n^{1/4}$ 次的更新操作后，可以预计该过程已经找到了要把 n 完全分解因子所需要的足够的约数，这是由于除了可能有的最大的一个素因子外，n 的每一个素因子 p 均小于 $\sqrt{n}$。

分析一下要经过多久，模 n 的随机序列中才会重复出现一个值，就可以了解这个过程的性能。由于 $\mathbb{Z}_n$ 是有限的，并且序列(31.42)中的每一个值仅仅取决于前一个值，所以序列(31.42)最终将产生自身重复。一旦到达一个 x_i，满足对某个 $j<i$ 有 $x_i=x_j$，则已经处在一个回路中，因为 $x_{i+1}=x_{j+1}$，$x_{i+2}=x_{j+2}$，等等。该过程取名为“rho 启发式方法”，原因就在于(如图 31-7 所示)序列 x_1，x_2，…，x_{j-1} 可以画成 rho(ρ)的“尾”，而回路 x_j，x_{j+1}，…，x_i 可以画成 rho 的“体”。

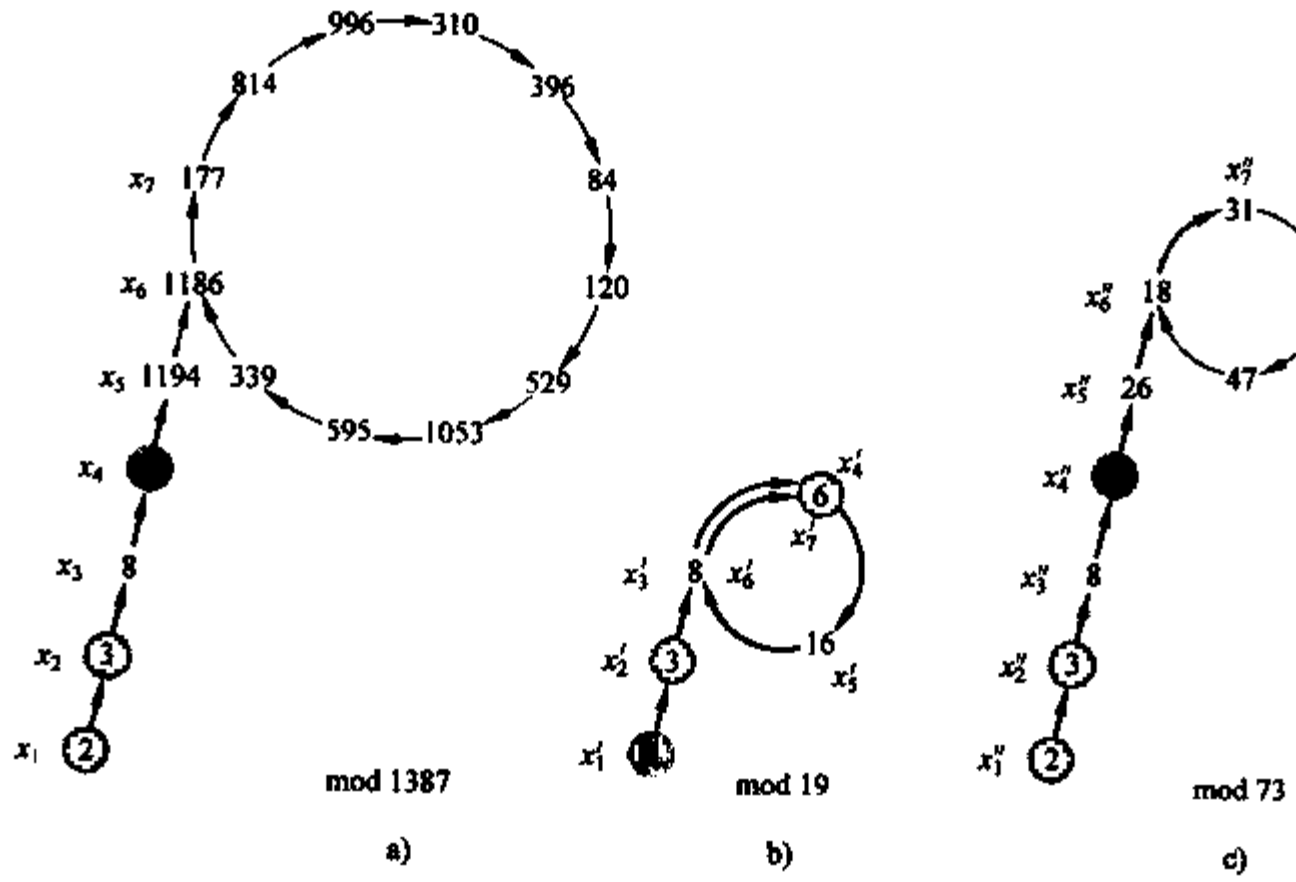


图 31-7　Pollard 的 rho 启发式方法 a) 由递归式 $x_{i+1}\leftarrow(x_i^2-1)\bmod 1387$ 所产生的值，从 $x_1=2$ 开始。1387 的素数因子分解是 $19\cdot 73$。粗箭头指出在因子 19 被发现之前所执行的迭代步骤。细箭头指出在迭代中未到达的值，来画出“rho”的形状。阴影数值是由 POLLARD-RHO 保存的 y 的值。因子 19 是在到达 $x_7=177$ 时被发现，此时计算 gcd(63−177，1387)=19。第一个要重复的 x 数值是 1186，但是因子 19 是在这个数值重复之前被发现的。b)相同递归式产生的值，模 19。a)给出的每个值 x_i 对模 19 来说等于这里显示的 x'_i。例如，$x_4=63$ 和 $x_7=177$ 都等于 6，模 19。c)相同递归式产生的值，模 73。a)给出的每个值 x_i 对模 73 来说等于这里显示的 x''_i。由中国余数定理，a)中每个结点对应于一对结点，即 b)中一个和 c)中的一个

下面来考虑一个问题，x_i 的序列发生重复需要多久？实际上，这个问题的答案并不是我们所需要的，但我们将会看到如何修改这个论点。

为了进行估算，假定函数

$$f_n(x)=(x^2-1)\bmod n$$

像一个“随机”函数那样进行计算。当然，它并不是一个真正的随机函数，但根据这个假设所获得的结论，与我们对 POLLARD-RHO 的执行情况进行的观察是一致的。因而，可以把每个 x_i

看作按在 $\mathbf{Z}_n$ 上均匀分布的方式从 $\mathbf{Z}_n$ 中独立选取的。根据 5.4.1 节中对生日悖论的分析，在序列出现回路之前预计要执行的步数为 $\Theta(\sqrt{n})$。

现在根据要求作适当修改。设 p 是满足 $\gcd(p, n/p)=1$ 的 n 的一个非平凡因子。例如，如果 n 的因子分解为 $n=p_1^{e_1}p_2^{e_2}\cdots p_r^{e_r}$，则可以取 p 的值为 $p_1^{e_1}$。(如果 $e_1=1$，则 p 就是 n 的最小素数因子，这是一个可以牢记的好例子。)

序列$\langle x_i\rangle$包含一个相应的对模 p 的序列$\langle x_i'\rangle$，其中对于所有 i

$$x_i' = x_i \bmod p$$

更进一步，因为 f_n 是仅使用算术操作(平方和减法)模 n 定义的，所以我们将看到可以用 x_i' 来计算 x_{i+1}'；序列从"模 p"的角度看是从"模 n"的角度看的一个较小版本。

$$\begin{aligned} x_{i+1}' &= x_{i+1} \bmod p = f_n(x_i) \bmod p = ((x_i^2-1)\bmod n)\bmod p \\ &= (x_i^2-1)\bmod p = ((x_i \bmod p)^2-1) \bmod p \\ &= ((x_i')^2-1) \bmod p = f_p(x_i') \qquad \text{(根据练习 31.1-6)} \end{aligned}$$

因此，虽然我们没有显式地计算序列$\langle x_i'\rangle$，但这个序列是良定义的，而且与序列$\langle x_i\rangle$有相同的递归式。

如前面一样进行推理，可以发现在序列$\langle x_i'\rangle$重复出现之前，预计执行的步数是 $\Theta(\sqrt{p})$。如果与 n 相比，p 是一个很小的数，则序列$\langle x_i'\rangle$可能比序列$\langle x_i\rangle$重复的频率要快得多。的确，一旦序列$\langle x_i\rangle$中的两个元素仅需对模 p 等价，而无需对模 n 等价，序列$\langle x_i'\rangle$就发生重复。图 31-7b、c 说明了这一点。

设 t 表示$\langle x_i'\rangle$序列中第一个重复出现的值的下标，并且 $u>0$ 表示所产生的循环回路的长度。亦即 t 和 $u>0$ 是对所有 $i\geqslant 0$，满足条件 $x_{t+i}'=x_{t+u+i}'$的最小值。根据上面的论证，t 和 u 的期望值都是 $\Theta(\sqrt{p})$。注意，如果 $x_{t+i}'=x_{t+u+i}'$，则 $p \mid (x_{t+u+i}-x_{t+i})$。因此，$\gcd(x_{t+u+i}-x_{t+i}, n)>1$。

所以，一旦 POLLARD-RHO 把满足 $k\geqslant t$ 的任何值 x_k 存入变量 y，则 $y \bmod p$ 总在模 p 的回路中(如果把一个新值存入 y，则该值也在对模 p 的回路中。)最终，赋给 k 的值将大于 u，然后过程就沿着模为 p 的回路，完成整个一次循环而不改变 y 的值。当 x_i"遇到"先前存储的对模 p 的 y 值时，即当 $x_i \equiv y \pmod p$时，就找到了 n 的一个因子。

发现的因子可以假定是因子 p，但偶尔也可能是 p 的倍数。由于 t 和 u 的期望值都是 $\Theta(\sqrt{p})$，所以产生因子 p 所要求的期望执行步数为 $\Theta(\sqrt{p})$。

该算法不会如期望的那样执行，原因有两条。第一，对于运行时间的启发式分析并不严格，也有可能对模 p 的值的回路要比$\sqrt{p}$大得多。在这种情况下，算法的执行虽然仍然正确，但其执行速度要比期望低的多。在实际应用中，这似乎还可以讨论。第二，这一算法得出 n 的约数可能总是平凡因子 1 或 n。例如，假定 $n=pq$，其中 p 和 q 为素数。可能会发生这样的情况：关于 p 的 t 和 u 的值与关于 q 的 t 和 u 的值相等，因此因子 p 和因子 q 总是在相同的 gcd 运算中被揭示出。由于两个因子同时被揭示，所以也就揭示出无用的平凡因子 $pq=n$。在实际应用中，这似乎也没意义。如果需要，可以用一个不同形式的递归式 $x_{i+1}\leftarrow(x_i^2-c)\bmod n$ 来重新开始运行该启发式过程(值 $c=0$ 和 $c=2$ 应该被避免，其原因这里不作说明，但对其它值没有问题。)

当然，上述分析过程是启发式的，不是严格的，因为递归式并不真是"随机的"。然而，在实际应用中该过程可以良好地运行，并且似乎和我们在上面的启发式分析中所说明的一样有效。它是一种找出大整数的小素数因子的可供选择的方法。为了对一个 β 位合数 n 完全分解因子，仅需找出所有小于$\lfloor n^{1/2}\rfloor$的素数因子就可以了，因此，可以期望 POLLARD-RHO 需执行的算术运

898 ≀ 899
900

算至多为 $n^{1/4}=2^{\beta/4}$ 次，位操作至多为 $n^{1/4}\beta^2=2^{\beta/4}\beta^2$。POLLARD-RHO 最具有吸引力的特点就是它可以在 $\Theta(\sqrt{p})$ 次期望的算术运算内，找出 n 的一个小因子 p。

练习

31.9-1　在图 31-7a 所示的执行过程中，过程 POLLARD-RHO 在何时输出 1387 的因子 73？

31.9-2　假设已知函数 f：$\mathbf{Z}_n\to\mathbf{Z}_n$ 和一个初值 $x_0\in\mathbf{Z}_n$。定义 $x_i=f(x_{i-1})$，$i=1$，2，…。设 t 和 $u>0$ 是满足 $x_{t+i}=x_{t+u+i}$，$i=0$，1，…的最小值。在 Pollard 的 rho 算法的术语中，t 为 rho 的尾的长度，u 是 rho 的回路的长度。试写出一个计算 t 和 u 的值的有效算法，并分析其运行时间。

31.9-3　要发现形如 p^e 的数（其中 p 是素数，$e>1$）的一个因子，POLLARD-RHO 要执行多少步？

*31.9-4　POLLARD-RHO 的一个缺陷是在其递归过程的每一步，都要计算一个 gcd。有人建议对 gcd 的计算进行批处理：累计一行中数个连续的 x_i 的积，然后在 gcd 计算中使用该积而不是 x_i。请说明如何实现这一设计思想，以及为什么它是正确的，在处理一个 β 位数 n 时，所选取的最有效的批处理规模是多大？

901

思考题

33-1　二进制的 gcd 算法

在大多数计算机上，与计算余数的执行速度相比，减法运算、测试一个二进制整数的奇偶性运算以及折半运算的执行速度都要更快些。本问题所讨论的二进制 gcd 算法中避免了欧几里得算法中对余数的计算过程。

a）证明：如果 a 和 b 都是偶数，则 $\gcd(a,b)=2\gcd(a/2,b/2)$。

b）证明：如果 a 是奇数，b 是偶数，则 $\gcd(a,b)=\gcd(a,b/2)$。

c）证明：如果 a 和 b 都是奇数，则 $\gcd(a,b)=\gcd((a-b)/2,b)$。

d）设计一个有效的二进制 gcd 算法，输入为整数 a 和 $b(a\geqslant b)$ 并且算法的运行时间为 $O(\lg a)$。假定每个减法运算、测试奇偶性运算以及折半运算都能在单位时间内执行。

31-2　对欧几里得算法中位操作的分析

a）证明：用普通的“纸和笔”算法来进行长除法运算：用 a 除以 b，得到商 q 和余数 r，需要执行 $O((1+\lg q)\lg b)$ 次位操作。

b）定义 $\mu(a,b)=(1+\lg a)(1+\lg b)$。证明：过程 EUCLID 在把计算 $\gcd(a,b)$ 的问题转化为计算 $\gcd(b,a \bmod b)$ 的问题时，所执行的位操作次数至多为 $c(\mu(a,b)-\mu(b,a \bmod b))$，其中 $c>0$ 为某一个足够大的常数。

c）证明：在一般情况下 EUCLID(a,b) 需要执行的位操作次数为 $O(\mu(a,b))$；当其输入为两个 β 位数时需要执行的位操作次数为 $O(\beta^2)$。

31-3　关于斐波那契数的三个算法

在已知 n 的情况下，本问题对计算第 n 个斐波那契数 F_n 的三种算法的效率进行了比较。假定两个数的加法、减法和乘法的代价都是 $O(1)$，与数的大小无关。

a）证明：用递归式(3.21)来计算 F_n 的直接递归方法的运行时间为 n 的幂。

902　b）试说明如何运用记忆法在 $O(n)$ 的时间内计算 F_n。

c）试说明如何仅用整数加法和乘法运算，就可以在 $O(\lg n)$ 的时间内计算 F_n。（提示：考虑矩阵

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

和它的幂。)

d)现在假设对两个 β 位数相加需要 $\Theta(\beta)$ 的时间，对两个 β 位数相乘需要 $\Theta(\beta^2)$ 的时间。如果这样来更合理地估计基本算术运算的代价，则这三种方法的运行时间又是多少？

31-4　二次余数

设 p 是一个奇素数。如果关于未知量 x 的方程 $x^2 \equiv a \pmod{p}$ 有解，则数 $a \in \mathbf{Z}_p^*$ 就是一个二次余数。

a)证明：对模 p，恰有 $(p-1)/2$ 个二次余数。

b)如果 p 是素数，对 $a \in \mathbf{Z}_p^*$，定义勒让德符号 $\left(\frac{a}{p}\right)$ 等于 1，如果 a 是对模 p 的二次余数；否则其值等于 -1。证明：如果 $a \in \mathbf{Z}_p^*$，则

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}$$

试写出一个有效的算法，使其能确定一个给定的数 a 是否是对模 p 的二次余数。分析所给算法的效率。

c)证明：如果 p 是形如 $4k+3$ 的素数，且 a 是 $\mathbf{Z}_p^*$ 中一个二次余数，则 $a^{k+1} \bmod p$ 是对模 p 的 a 的平方根。找出一个对模 p 的二次余数 a 的平方根需要多长时间？

d)试描述一个有效的随机算法，来找出一个以任意素数 p 为模的非二次余数，以及 $\mathbf{Z}_p^*$ 中不是二次余数的成员。所给出的算法平均需要执行多少次算术运算？

本章注记

Niven 和 Zuckerman[231]给出了有关基本数论的优秀介绍。Knuth[183]中包含了找出最大公约数的算法，以及其他基本数论算法的一个很好的讨论。Bach[28]和 Riesel[258]提供了计算数论的较新的综述。Dixon[78]给出了因子分解和素数测试的一个概论。Pomerance[245]编辑的会议文集包含了数篇优秀的综述文章。在最近，Bach 和 Shallit[29]提供了计算数论基础的一个特别的概论。 903

Knuth[183]讨论了欧几里得算法的来源。它出现在希腊数学家欧几里得在公元前 300 年所写的《几何原本》的第 7 册中的命题 1 和命题 2。欧几里得的描述可能来源于大约公元前 375 年的 Eudoxus 的一个算法。欧几里得的算法可以说是最早的非平凡算法；只有古埃及人所知的一个乘法算法可以与之匹敌。Shallit[274]编撰了欧几里得算法的分析历史。

Knuth 将中国余数定理(定理 31.27)的一个特殊情况归功于中国数学家孙子，他生活在约公元前 200 年到公元后 200 年(这个日期很不确定)。相同的特殊情况是由约公元后 100 年的希腊数学家 Nichomachus 所给出的。在 1247 年它被秦九韶一般化。中国余数定理由 L. Euler 在 1734 年以最完全的方式做了最后的陈述和证明。

在这里展示的随机素数测试算法归功于 Miller[221]和 Rabin[254]；它是已知的最快速的随机素数测试算法，在常数因子内。定理 31.39 的证明稍微采纳了 Bach[27]提出的建议。MILLER-RABIN 的一个更强结果的证明由 Monier[224，225]给出。随机化在得到一个多项式时间的素数测试算法时是必要的。已知的最快的确定性素数测试算法是 Adleman，Pomerance 和 Rumely[3]的素数测试的 Cohen-Lenstra 版本[65]。当测试一个长度为 $\lceil \lg(n+1) \rceil$ 的数字 n 的素数性时，它在 $(\lg n)^{O(\lg\lg\lg n)}$ 时间内运行，只是稍微超多项式。

找出大的"随机"素数的问题在 Beauchemin，Brassard，Crépeau，Goutier 和 Pomerance[33]的一篇论文中有好的讨论。

公钥加密系统的概念归功于 Diffie 和 Hellman[74]。RSA 加密系统于 1977 年由 Rivest，Shamir 和 Adleman[259]提出。从那时开始，密码学领域开始蓬勃发展。我们对 RSA 加密系统的了解已经加深，而现代的实现是用展示在这里的基本技术的显著改进。另外，许多新技术已经发展，证明加密系统是安全的。例如，Goldwasser 和 Micali[123]说明随机化在安全的公钥加密方案的设计中是一个有效的工具。对于签名方案，Goldwasser，Micali 和 Rivest[124]展示了一个
904 数字签名方案，其中每一种可想象到的伪造行为可以证明与因子分解一样困难。Menezes 等人[220]提供了应用密码学的一个概况。

整数因子分解的 rho 启发式方法是由 Pollard[242]提出的。展示在这里的版本是 Brent[48]所提议的一个变种。

对大数因子分解的最佳算法来说，其运行时间大概是指数增长于要因子分解的数 n 的长度的 3 次根。一般数域的筛选因子分解算法可能是对于一般的大输入最有效的算法，它是由 Buhler 等人[51]提出的，旨在对数域筛选因子分解算法进行扩展，后者是由 Pollard[243]和 Lenstra 等人[201]提出的、并由 Coppersmith[69]以及其他人加以了改善。虽然很难给出这个算法的严格的分析，在合理的假设下我们可以得到 $L(1/3, n)^{1.902+o(1)}$ 的一个运行时间，其中 $L(\alpha, n) = e^{(\ln n)^{\alpha}(\ln \ln n)^{1-\alpha}}$。

椭圆曲线方法是由 Lenstra[202]提出的，它对于某些输入比数域筛选方法更有效，因为，与 Pollard 的 rho 方法一样，它可以相当快速地找到一个小素数因子 p。使用这个方法，找到 p 的时
905 间预计是 $L(1/2, p)^{\sqrt{2}+o(1)}$。

第 32 章　字符串匹配

在文本编辑程序中，经常出现要在一段文本中找出某个模式的全部出现位置这一问题。典型情况是，一段文本是正在编辑的文件，所搜寻的模式是用户提供的一个特定单词。解决这个问题的有效算法能极大地提高文本编辑程序的响应性能。字符串匹配算法也常常用于其他方面，例如在 DNA 序列中搜寻特定的模式。

字符串匹配问题的形式定义是这样的：假设文本是一个长度为 n 的数组 $T[1..n]$，模式是一个长度为 $m \leqslant n$ 的数组 $P[1..m]$。进一步假设 P 和 T 的元素都是属于有限字母表 Σ 表中的字符。例如，可以有 $\Sigma=\{0, 1\}$ 或 $\Sigma=\{a, b, \cdots, z\}$，字符数组 P 和 T 常称为字符串。

如果 $0 \leqslant s \leqslant n-m$，并且 $T[s+1..s+m]=P[1..m]$（即对 $1 \leqslant j \leqslant m$，有 $T[s+j]=P[j]$），则说模式 P 在文本 T 中出现且位移为 s。（或者等价地，模式 P 在文本 T 中从位置 $s+1$ 开始出现。）如果 P 在 T 中出现且位移为 s，则称 s 是一个有效位移，否则称 s 为无效位移。这样一来，字符串匹配问题就变成一个在一段指定的文本 T 中，找出某指定模式 P 出现的所有有效位移的问题。图 32-1 说明这个定义。

图 32-1　字符串匹配问题。目标是找出所有在文本 $T=$abcabaabcabac 中模式 $P=$abaa 的所有出现。该模式在文本中仅出现了一次，在位移 $s=3$ 处。位移 $s=3$ 是有效位移。模式中的每个字符都用一条竖线与文本中的匹配字符联系，且所有匹配的字符都涂了阴影

除了在 32.1 节我们将回顾关于字符串匹配问题的朴素的算法外，本章的每个字符串匹配算法都对模式进行了一些预处理，然后找寻所有有效位移；我们称第二步为“匹配”。图 32-2 给出本章所有算法的预处理和匹配的时间。每个算法的总运行时间为预处理和匹配时间的总和。32.2 节介绍由 Rabin 和 Karp 发现的一种有趣的字符串匹配算法。该算法在最坏情况下的运行时间为 $\Theta((n-m+1)m)$，虽然这一时间并不比朴素的算法好，但是在平均情况和实际情况中，该算法的效果要好得多。这种算法也可以很好地推广到解决其他的模式匹配问题。32.3 节中描述另一种字符串匹配算法，该算法构造一个特别设计的有限自动机，用来搜寻某给定模式 P 在文本中的出现位置。此算法用 $O(m|\Sigma|)$ 的预处理时间，但只用 $\Theta(n)$ 的匹配时间。32.4 节介绍与其类似但更巧妙的 Knuth-Morris-Pratt（或 KMP）算法。该算法的匹配时间同样为 $\Theta(n)$，但是将预处理时间降至 $\Theta(m)$。

算法	预处理时间	匹配时间
朴素算法	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
有限自动机算法	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

图 32-2　本章的字符串匹配算法以及它们的预处理和匹配时间

记号与术语

我们将用Σ^*表示用字母表Σ中的字符形成的所有有限长度的字符串的集合。在本章中，我们仅考虑长度有限的字符串。长度为0的空字符串用ϵ表示，它也属于Σ^*。字符串x的长度用$|x|$表示。两个字符串x和y的连接表示为xy，其长度为$|x|+|y|$，由x的字符接y的字符组成。

如果对某个字符串$y\in\Sigma^*$，有$x=wy$，就说字符串w是字符串x的前缀，表示为$w\sqsubset x$。注意，如果$w\sqsubset x$，则$|w|\leqslant|x|$。类似地，如果对某个字符串$y\in\Sigma^*$，有$x=yw$，就说字符串w是字符串x的后缀，表示为$w\sqsupset x$，由$w\sqsupset x$推知$|w|\leqslant|x|$。空字符串ϵ既是每个字符串的前缀，也是每个字符串的后缀。例如，有ab$\sqsubset$abcca，cca$\sqsupset$abcca。对任意字符串x和y
907 以及任意字符a，$x\sqsupset y$当且仅当$xa\sqsupset ya$。请注意，$\sqsubset$和$\sqsupset$都是传递关系。下列引理在后面将会用到。

引理32.1(重叠后缀引理)　假设x，y和z是满足$x\sqsupset z$和$y\sqsupset z$的三个字符串。如果$|x|\leqslant|y|$，则$x\sqsupset y$；如果$|x|\geqslant|y|$，则$y\sqsupset x$；如果$|x|=|y|$，则$x=y$。

证明：见图32-3的图形证明。 ■

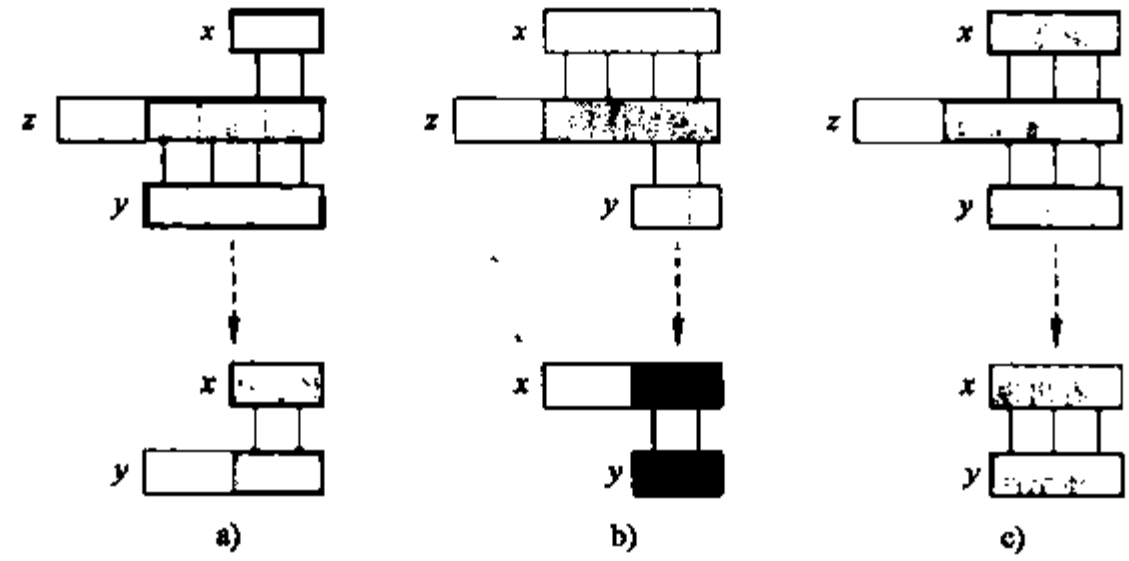


图32-3　引理32.1的图形证明。假定$x\sqsupset z$和$y\sqsupset z$。图的三个部分分别说明引理的三种情况。竖线连接字符串的匹配区域(用阴影表示)。a)如果$|x|\leqslant|y|$，则$x\sqsupset y$。b)如果$|x|\geqslant|y|$，则$y\sqsupset x$。c)如果$|x|=|y|$，则$x=y$

为了使记号简洁，我们将把模式$P[1..m]$的由k个字符组成的前缀$P[1..k]$用P_k表示。因此，$P_0=\epsilon$，$P_m=P=P[1..m]$。类似地，我们把文本T的由k个字符组成的前缀表示为T_k。采用这种记号，可以把字符串匹配问题表述为：找出范围为$0\leqslant s\leqslant n-m$并满足$P\sqsupset T_{s+m}$的所有位移$s$。

在我们的伪代码中，允许把比较两个等长的字符串是否相等的操作当作原语操作。如果对字符串的比较是从左往右进行，并且发现一个不匹配字符时比较就终止，则假设这样一个测试过程所需的时间是关于所发现的匹配字符数目的线性函数。更准确地说，假设测试"$x=y$"需要的时间为$\Theta(t+1)$，其中t是满足$z\sqsubset x$且$z\sqsubset y$的最长字符串z的长度。(之所以写$\Theta(t+1)$而不
908 是$\Theta(t)$，是考虑到$t=0$的情况；前面的字符不匹配，但是仍需要一定量的时间。)

32.1　朴素的字符串匹配算法

下面要介绍一个朴素的字符串匹配算法，它用一个循环来找出所有有效位移，该循环对$n-m+1$个可能的每一个s值检查条件$P[1..m]=T[s+1..s+m]$。

```
NAIVE-STRING-MATCHER(T, P)
1  n ← length[T]
2  m ← length[P]
3  for s ← 0 to n−m
4      do if P[1..m]=T[s+1..s+m]
5          then print"Pattern occurs with shift"s
```

这种朴素的字符串匹配过程可以形象地看成用一个包含模式的"模板"沿文本滑动，同时对每个位移注意模板上的字符是否与文本中的相应字符相等，如图 32-4 所示。从第 3 行开始的 for 循环显式地考察每一个可能的位移。第 4 行的测试代码确定当前的位移是否有效，该测试隐含着一个循环。该循环逐个对相应位置上的字符进行检查，直至所有位置都能成功地进行匹配，或者发现一个不匹配的位置时为止。第 5 行输出每个有效的位移 s。

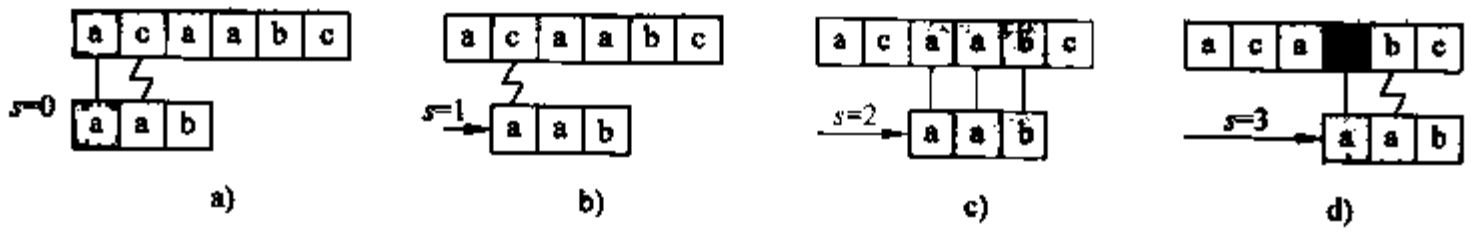


图 32-4　朴素字符串匹配对模式 $P=$aab 和文本 $T=$acaabc 的操作。可以把 P 想像成一个沿着正文滑动的"模板"。a)～d)为 4 个连续的朴素字符串匹配。图中竖线连接相应匹配区域(阴影部分)，折线连接先错误匹配的字符，如果是的话。在位移 $s=2$ 时，找到匹配的模式，见图中 c)

在最坏情况下，过程 NAIVE-STRING-MATCHER 的运行时间为 $\Theta((n-m+1)m)$。例如，考察文本字符串 a^n(n 个 a 组成的字符串)和模式 a^m。对 $n-m+1$ 个可能的位移值 s 中的每一个值，第 4 行中比较相应字符的循环必须执行 m 次，以证明位移的有效性。因此，最坏情况下的运行时间为 $\Theta((n-m+1)m)$，如果 $m=\lfloor n/2 \rfloor$，则为 $\Theta(n^2)$。由于没有预处理，NAIVE-STRING-MATCHER 的运行时间与它的匹配时间相等。

我们将看到，NAIVE-STRING-MATCHER 并不是关于这个问题的最优过程。在本章中，还要介绍一种算法，它的最坏情况预处理时间为 $\Theta(m)$，最坏情况匹配时间为 $\Theta(n)$。这种朴素字符串匹配算法的效率不高，其原因在于对于 s 的一个值，我们获得的关于文本的信息在考虑 s 的其他值时完全被忽略了。这样的信息可能是非常有用的，例如，如果 $P=$aaab，并且我们发现 $s=0$ 是有效的，则位移 1，2，3 都不是有效位移，因为 $T[4]=$b。后面我们将考察能够有效地利用这部分信息的几种方法。

练习

32.1-1　试说明当模式 $P=0001$，文本 $T=000010001010001$ 时，朴素的字符串匹配所执行的比较。

32.1-2　假设模式 P 中的所有字符都是不同的。试说明如何对一段 n 个字符的文本 T 加速过程 NAIVE-STRING-MATCHER 的执行速度，使其运行时间达到 $O(n)$。

32.1-3　假设模式 P 和文本 T 是长度分别为 m 和 n 的随机选取的字符串，其字符属于 d 个元素的字母表 $\sum_d = \{0,1,\cdots,d-1\}$，其中 $d \geqslant 2$。证明朴素算法第 4 行中隐含的循环所执行的字符比较的预计次数为：

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leqslant 2(n-m+1)$$

(假定一旦发现一个不匹配字符或整个模式已被匹配时，朴素算法就终止对于给定位移的字符比较过程。)这个结论说明，对随机选取的字符串来说，朴素算法还是相当有效的。

32.1-4 假设允许模式 P 中包含一个间隔字符◇，该字符可以与任意的字符串匹配(甚至可以与长度为 0 的字符串匹配。)例如，模式 ab◇ba◇c 在文本 cabccbacbacab 中的出现为

$$\mathrm{c}\ \underbrace{\mathrm{ab}}_{\mathrm{ab}}\ \underbrace{\mathrm{cc}}_{\diamond}\ \underbrace{\mathrm{ba}}_{\mathrm{ba}}\ \underbrace{\mathrm{cba}}_{\diamond}\ \underbrace{\mathrm{c}}_{\mathrm{c}}\ \mathrm{ab}$$

和

$$\mathrm{c}\ \underbrace{\mathrm{ab}}_{\mathrm{ab}}\ \underbrace{\mathrm{ccbac}}_{\diamond}\ \underbrace{\mathrm{ba}}_{\mathrm{ba}}\ \underbrace{\ }_{\diamond}\ \underbrace{\mathrm{c}}_{\mathrm{c}}\ \mathrm{ab}$$

909 ~ 910

注意，间隔字符可以在模式中出现任意次，但假定它不会出现在文本中。试给出一个多项式运行时间的算法，以确定这样的模式 P 是否出现于给定的文本 T 中，并分析你的算法的运行时间。

32.2 Rabin-Karp 算法

在实际应用中，Rabin 和 Karp 所建议的字符串匹配算法能够较好地运行，我们还可以从中归纳出有关问题的其他算法，如二维模式匹配。Rabin-Karp 算法预处理时间为 $\Theta(m)$，在最坏情况下的运行时间为 $O((n-m+1)m)$，但基于某种假设，它的平均情况运行时间还是比较好的。

该算法中利用了一些初等数论概念，如两个数对于第三个数的模等价。要了解有关的定义，请参照 31.1 节的内容。

为了便于说明，假定 $\Sigma=\{0, 1, 2, \cdots, 9\}$，这样每个字符都是一个十进制数字。(一般情况下，可以假定每个字符都是基数为 d 的表示法中的一个数字，$d=|\Sigma|$。)可以用一个长度为 k 的十进制数来表示由 k 个连续字符组成的字符串。因此，字符串 31415 就对应于十进制数 31 415。如果输入字符既可以看作图形符号，也可以看作数字，我们就会发现在本节中的标准文本字体中，采用这种数字表示是很方便的。

已知一个模式 $P[1..m]$，设 p 表示其相应的十进制数的值。类似地，对于给定的文本 $T[1..n]$，用 t_s 来表示其长度为 m 的子字符串 $T[s+1..s+m]$($s=0, 1, \cdots, n-m$)相应的十进制数的值。当然，$t_s=p$ 当且仅当 $T[s+1..s+m]=P[1..m]$，因此 s 是有效位移当且仅当 $t_s=p$。如果能够在 $\Theta(m)$ 的时间内计算出 p 的值，并在总共 $\Theta(n-m+1)$ 的时间[⊖]内计算出所有 t_s 的值，那么通过把 p 值与每个 t_s 值进行比较，就能够在 $\Theta(m)+\Theta(n-m+1)=\Theta(n)$ 的时间内，求出所有有效位移 s(目前暂不考虑 p 和 t_s 可能是很大的数的情况)。

我们可以运用霍纳法则(Horner's rule，参见 30.1 节)在 $\Theta(m)$ 的时间内计算 p 的值：

$$p = P[m]+10(P[m-1]+10(P[m-2]+\cdots+10(P[2]+10P[1])\cdots))$$

911 类似地，也可以在 $\Theta(m)$ 的时间内，根据 $T[1..m]$ 计算出 t_0 的值。

为了在 $\Theta(n-m)$ 的时间内计算出剩余的值 t_1，t_2，…，t_{n-m}，可以在常数时间内根据 t_s 计算出 t_{s+1}，这是因为

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \qquad (32.1)$$

例如，如果 $m=5$，$t_s=31\,415$，我们希望去掉高位数字 $T[s+1]=3$，再加入一个低位数字(假定它是 $T[s+5+1]=2$)就得到：

⊖ 写 $\Theta(n-m+1)$ 而不是 $\Theta(n-m)$，是因为 s 具有 $n-m+1$ 不同的值。"+1"是为了突显 $m=n$ 时的渐近意义，单计算 t_s 的值需要 $\Theta(1)$ 时间，而不是 $\Theta(0)$ 时间。

$$t_{s+1} = 10(31\,415 - 10\,000 \cdot 3) + 2 = 14\,152$$

减去 $10^{m-1}T[s+1]$就从 t_s 中去掉了高位数字，再把结果乘以 10 就使数位向左移了一位。然后再加上 $T[s+m+1]$，就加入一个适当的低位数。如果能够预先计算出常数 10^{m-1}(用 31.6 节中介绍的技术，就可以在 $O(\lg m)$的时间内完成这一计算过程，但对这个应用问题，用一种简便的运行时间为 $O(m)$的方法就可以进行计算)，则每次执行式(32.1)时，需要执行的算术运算次数为常数。因此，可以在 $\Theta(m)$时间内计算出 p，在 $\Theta(n-m+1)$时间内计算出 t_0，t_1，…，t_{n-m}，因而能够用 $\Theta(m)$的预处理时间和 $\Theta(n-m+1)$的匹配时间，找出模式 $P[1..m]$在文本 $T[1..n]$中的所有出现。

该过程的唯一问题就是 p 和 t_i 的值可能太大，导致不能方便地对其进行处理。如果 P 包含 m 个字符，那么，关于在 p(m 数位)上的每次算术运算需要"常数"时间这一假设就不合理了。幸运的是，对这一问题还存在一种简单的补救方法，如图 32-5 所示：对一个合适的模 q 来计算 p 和 t_i 的模。每个字符是一个十进制数，因为 p，t_0 以及递归式(32.1)计算过程都可以对模 q 进行，所以可以在 $\Theta(m)$时间内计算出模 q 的 p 值，在 $\Theta(n-m+1)$时间内计算出模 q 的所有 t_i 值。通常选模 q 为一个素数，使得 $10q$ 正好为一个计算机字长，用单精度算术运算就可以执行所有必

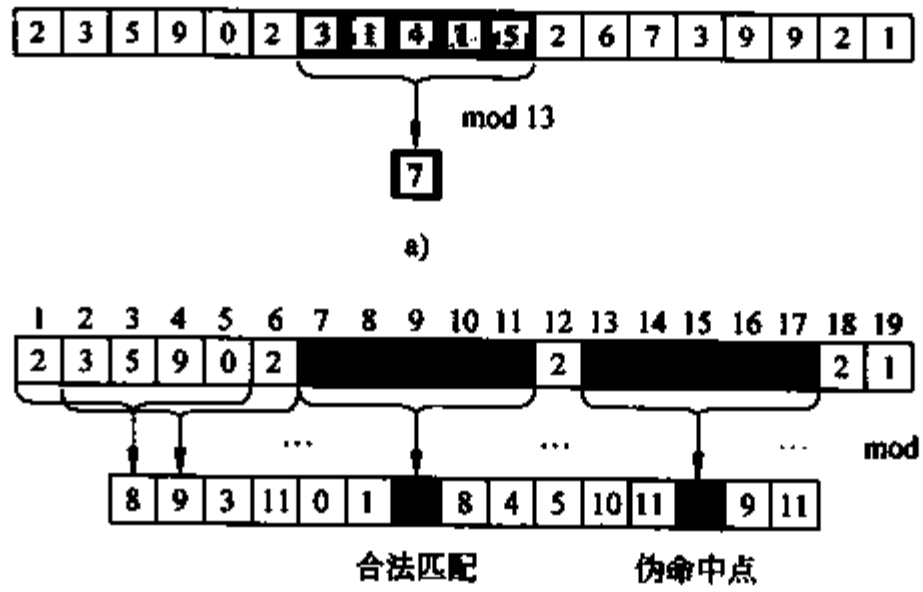


图 32-5　Rabin-Karp 算法。每一个字符是一个 10 进制数字，用模 13 计算值。a)一个文本字符串。长度为 5 的窗口被标上了阴影。阴影数字的数值模 13 计算出的值为 7。b)一个相同的文本字符串，以及其所有可能的长度为 5 的窗口的每一个位置所计算出的模 13 的值。假定模式 $P = 31\,415$，寻找模 13 为 7 的窗口，因为 $31\,415 \equiv 7 \pmod{13}$。找到了两个这样的窗口，在图中标上阴影。第一个在文本位置 7 开始的，确实是模式的出现；而第二个在文本位置 13 开始的是一个伪命中。c)已知前一个窗口的值，在常数时间内计算出某窗口的值。第一个窗口的值 31 415。去掉高位数字 3，左移(乘以 10)，然后加入低位数字 2 得到一个新值 14 152。但是，所有的计算都是模 13，因此第一个窗口的值为 7，而新窗口的值为 8

要的运算过程。在一般情况下，采用 d 进制的字母表$\{0, 1, \cdots, d-1\}$时，所选取的 q 要满足使 dq 的值在一个计算机字长内，并调整递归式(32.1)以使对模 q 进行运算，使其成为：

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q \tag{32.2}$$

其中 $h \equiv d^{m-1} \pmod q$是一个 m 数位文本窗口中高位数位上的数字"1"的值。

但是，加入模 q 后，美中不足的是由 $t_s \equiv p \pmod q$并不能说明 $t_s = p$。另一方面，如果 $t_s \not\equiv p \pmod q$，则可以肯定 $t_s \neq p$，因此位移 s 是非法的。因此，可以把测试 $t_s \equiv p \pmod q$作为一种快速的启发式测试方法，以排除非法的位移 s。对任何满足 $t_s \equiv p \pmod q$的位移 s，都必须进一步进行测试，以决定 s 是真正的有效位移还是一个伪命中点。我们可以通过显式地检查条件 $P[1..m] = T[s+1..s+m]$来完成这种测试。如果 q 足够大，就可以期望伪命中点很少出现，这样进行额外检查的代价就足够低了。

912
~
913

下列过程是对上述设计思想的准确细化。过程的输入为文本 T，模式 P，使用的基数 d(它的典型值为 $|\Sigma|$)以及使用的素数 q。

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n ← length[T]
 2  m ← length[P]
 3  h ← d^(m-1) mod q
 4  p ← 0
 5  t_0 ← 0
 6  for i ← 1 to m                    ▷ Preprocessing.
 7       do p ← (dp+P[i]) mod q
 8          t_0 ← (dt_0 + T[i]) mod q
 9  for s ← 0 to n-m                  ▷ Matching.
10       do if p = t_s
11            then if P[1..m] = T[s+1..s+m]
12                    then print "Pattern occurs with shift" s
13          if s < n-m
14            then t_{s+1} ← (d(t_s - T[s+1]h) + T[s+m+1]) mod q
```

过程 RABIN-KARP-MATCHER 的执行过程如下。所有的字符都是 d 进制的数字。仅为了说明清楚，我们给 t 添加了下标；去掉所有的下标后，程序照样能够正确运行。第 3 行把 h 初始化为 m 数位窗口中高位数字的值。第 4～8 行计算出 p 的值为 $P[1..m] \bmod q$，t_0 的值为 $T[1..m] \bmod q$。第 9～14 行的 for 循环迭代了所有可能的位移 s，保持着如下的循环不变量：

无论第 10 行何时执行，$t_s = T[s+1..s+m] \bmod q$。

如果在第 10 行中有 $p = t_s$(一个"命中点")，则在第 11 行中判断是否有 $P[1..m] = T[s+1..s+m]$，以排除它是伪命中点的可能性。所发现的任何有效位移都在第 12 行中输出。如果$s < n-m$(第 13 行中检查该条件)，则至少再执行一次 for 循环，这时首先执行第 14 行，以保证再次执行到第 10 行时循环不变式依然成立。第 14 行直接利用等式(32.2)，就可在常数时间内由 $t_s \bmod q$ 的值计算出 $t_{s+1} \bmod q$ 的值。

RABIN-KARP-MATCHER 的预处理时间为 $\Theta(m)$，其匹配时间在最坏情况下为 $\Theta((n-m+1)m)$，因为 Rabin-Karp 算法与朴素的字符串匹配算法一样，对每个有效位移进行显式验证。如果 $P = a^m$ 并且 $T = a^n$，则验证所需的时间为 $\Theta((n-m+1)m)$，因为 $n-m+1$ 个可能的位移中每一个都是有效位移。

914

在许多实际应用中，有效位移数很少(如只有常数 c 个)，因此，算法的期望匹配时间为

$O((n-m+1)+cm)=O(n+m)$，再加上处理伪命中点所需的时间。假设减少模 q 的值就像是从 Σ^* 到 Z_q 上的一个随机映射，基于这种假设，进行启发性分析。（参见 11.3.1 节中对散列除法的讨论。要正式证明这个假设是比较困难的，但是有一种可行的方法，就是假定 q 是从适当大的整数中随机得出的。我们在此将不形式化定义。）于是，可以预计伪命中的次数为 $O(n/q)$，因为可以估计出任意的 t_s 对模 q 等价于 p 的概率为 $1/q$。因为第 10 行中的测试会在 $O(n)$ 个位置上失败，且每次命中的时间代价为 $O(m)$，因此，Rabin-Karp 算法的期望运行时间为：

$$O(n)+O(m(v+n/q))$$

其中 v 是有效位移数。如果选取 $q \geqslant m$ 且 $v=O(1)$，则这一算法的运行时间为 $O(n)$。就是说，如果期望的有效位移数很少（$O(1)$），而我们选取的素数 q 要比模式的长度大得多，则可以预计 Rabin-Karp 算法的匹配时间为 $O(n+m)$，因为当 $m \leqslant n$ 时，期望的匹配时间为 $O(n)$。

练习

32.2-1　如果取模 $q=11$，那么当 Rabin-Karp 匹配算法在文本 $T=3\,141\,592\,653\,589\,793$ 中搜寻模式 $P=26$ 时，会遇到多少个伪命中点？

32.2-2　如何扩展 Rabin-Karp 方法，使其能解决这样的问题：如何在文本字符串中搜寻出给定的 k 个模式中任何一个出现？起初假定所有 k 个模式都是等长的。然后扩展你的算法以允许不同长度的模式。

32.2-3　试说明如何扩展 Rabin-Karp 方法以处理下列问题：在一个 $n \times n$ 二维字符组成中搜寻一个给定的 $m \times m$ 模式。（可以使该模式在水平方向和垂直方向移动，但不可以把模式旋转。）

32.2-4　Alice 有一份很长的 n 位文件的复印件 $A=\langle a_{n-1}, a_{n-2}, \cdots, a_0\rangle$，Bob 也有一份类似的文件 $B=\langle b_{n-1}, b_{n-2}, \cdots, b_0\rangle$。Alice 和 Bob 都希望知道他们的文件是否一样。为了避免传送整个文件 A 或 B。他们运用下列快速概率检查手段，他们一起选择一个素数 $q>1000n$，并从 $\{0, 1, \cdots, q-1\}$ 中随机选取一个整数 x，然后，Alice 求出 915

$$A(x)=\left(\sum_{i=0}^{n} a_i x^i\right) \bmod q$$

的值，Bob 也用类似方法计算出 $B(x)$。证明：如果 $A \neq B$，则 $A(x)=B(x)$ 的概率至多为 1/1000；如果两个文件相同，则 $A(x)$ 的值必定等于 $B(x)$ 的值。（提示：参见练习 31.4-4）

32.3　利用有限自动机进行字符串匹配

很多字符串匹配算法都要建立一个有限自动机，它通过对文本字符串 T 进行扫描的方法，找出模式 P 的所有出现位置。本节将介绍一种建立这样的自动机的方法。用于字符串匹配的自动机都是非常有效的：它们只对每个文本字符检查一次，并且检查每个文本字符的时间为常数。因此，在建立好自动机后所需要的时间为 $\Theta(n)$。但是，如果 Σ 很大，建立自动机所花的时间也可能是很多的。32.4 节描述了解决这个问题的一种巧妙方法。

在本节的开头先来定义有限自动机概念。然后，我们要考察一种特殊的字符串匹配自动机，并说明如何利用它找出一个模式在文本中的出现位置。对这个问题的讨论中，包括对一段给定的文本，如何模拟出字符串匹配自动机的执行步骤的一些细节。最后，我们将说明对一个给定的输入模式，如何构造相应的字符串匹配自动机。

有限自动机

一个有限自动机 M 是一个 5 元组 $(Q, q_0, A, \Sigma, \delta)$，其中：

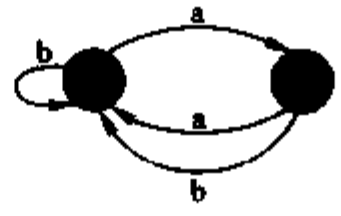
- Q 是状态的有限集合，
- $q_0 \in Q$ 是初始状态，
- $A \subseteq Q$ 是一个接受状态集合，
- Σ 是有限的输入字母表，
- δ 是一个从 $Q \times \Sigma$ 到 Q 的函数，称为 M 的转移函数。

有限自动机开始于状态 q_0，每次读入输入字符串的一个字符。如果有限自动机在状态 q 时读入了输入字符 a，则它从状态 q 变为状态 $\delta(q, a)$（进行了一次转移）。每当其当前状态 q 属于 A 时，就说自动机 M 接受了迄今为止所读入的字符串。没有被接收的输入称为被拒绝的输入。图 32-6 用一个简单的两状态自动机说明了上述定义。

916

状态	输入 a	输入 b
0	1	0
1	0	0

a)



b)

图 32-6　一个简单的两状态自动机，其状态集 $Q=\{0, 1\}$，开始状态 $q_0=0$，输入字母表 $\Sigma=\{a, b\}$。a)用表格表示的转移函数 δ。b)一个等价的状态转换图。状态 1 是仅有的接受状态(涂黑了)。有向边表示转移。例如，从状态 1 到状态 0，标有 b 的有向边表示 $\delta(1, b)=0$。此状态机接受那些在最后具有奇数个 a 的字符字符串。更准确地说，一个字符串 x 被接收，当且仅当 $x=yz$，其中 $y=\varepsilon$ 或 y 以一个 b 结尾，$z=a^k$，其中 k 是奇数。例如，对于输入 abaaa，此状态机的状态序列(包括初始状态)为$\langle 0, 1, 0, 1, 0, 1\rangle$，因而它接受了这个输入；对于输入 abbaa，状态序列是$\langle 0, 1, 0, 0, 1, 0\rangle$，因而它拒绝了这个输入

有限自动机 M 可以推导出一个函数 ϕ，称为终态函数，它是从 Σ^* 到 Q 的函数，并满足：$\phi(w)$ 是 M 在扫描字符串 w 后终止时的状态。因此，M 接受字符串 w 当且仅当 $\phi(w) \in A$。函数 ϕ 由下列递归关系定义：

$$\phi(\varepsilon) = q_0$$

$$\phi(wa) = \delta(\phi(w), a) \quad \text{对于 } w \in \Sigma^*, a \in \Sigma$$

字符串匹配自动机

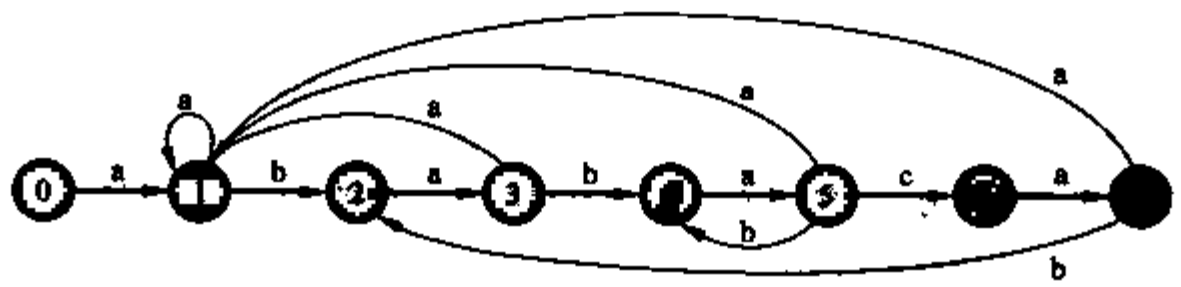
对每个模式 P 都存在一个字符串匹配自动机，必须在预处理阶段，根据模式构造出相应的自动机后，才能利用它来搜寻文本字符串。图 32-7 说明了关于模式 P=ababaca 的有限自动机的构造过程。从现在开始，假定 P 是一个已知的固定模式。为了使说明上的简洁，在下面的概念中将不特别指出对 P 的依赖关系。

为了详细说明与给定模式 $P[1..m]$ 相应的字符串匹配自动机，首先定义一个辅助函数 σ，称为相应 P 的后缀函数。函数 σ 是一个从 Σ^* 到 $\{0, 1, \cdots, m\}$ 上定义的映射，$\sigma(x)$ 是 x 的后缀 P 的最长前缀的长度：

$$\sigma(x) = \max\{k : P_k \sqsupset x\}$$

因为空字符串 $P_0=\varepsilon$ 是每一个字符串的后缀，所以后缀函数是有完备定义的。例如，对模式 P=ab，有 $\sigma(\varepsilon)=0$，$\sigma(\text{ccaca})=1$，$\sigma(\text{ccab})=2$。对一个长度为 m 的模式 P 来说，$\sigma(x)=m$ 当且

917



a)

状态	输入 a	b	c	P
0		0	0	a
1	1	2	0	b
2	3	0	0	a
3	1		0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
状态$\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

c)

图 32-7　a)一个字符串匹配自动机的状态转换图，它可以接收所有以字符串 ababaca 结尾的字符串。状态 0 是初始状态，状态 7(被涂黑)是仅有的接受状态。从状态 i 到状态 j，标有 a 的有向边表示 $\delta(i, a)=j$。形成自动机"脊"的右向边，在图中加重了颜色，对应着模式和输入字符串之间的成功匹配。向左指的边对应着失败的匹配。一些表示匹配失败的边并没有标出来；通常，如果状态 i 对某个 $a \in \Sigma$，没有对应 a 的出边，则 $\delta(i, a)=0$。b)对应的变迁函数 δ 和模式字符串 P=ababaca。模式和输入之间的成功匹配被标上了阴影。c)自动机在文本 T=abababacaba 上的操作。在处理了前缀 T_i 后，在文本的字符 $T[i]$下，给出了它在自动机内的状态 $\phi(T_i)$。找到了一个模式的出现，以位置 9 结尾

仅当 $P \sqsupset x$。根据后缀函数的定义有：如果 $x \sqsupset y$，则 $\sigma(x) \leqslant \sigma(y)$。

给定模式 $P[1..m]$，其对应的字符串匹配自动机定义如下：

- 状态集 Q 为$\{0, 1, \cdots, m\}$，初始状态 q_0 为 0 状态，状态 m 是唯一的接受状态。
- 对任意状态 q 和字符 a，变迁函数 δ 由如下等式定义

$$\delta(q,a) = \sigma(P_q a) \tag{32.3}$$

直觉上定义 $\delta(q, a)=\sigma(P_q a)$是合理的。自动机的操作中保持如下条件不变：

$$\phi(T_i) = \sigma(T_i) \tag{32.4}$$

下面的定理 32.4 将证明这个结论。这意味着对文本字符串 T 的前面 i 个字符进行扫描后，自动机的状态为 $\phi(T_i)=q$，其中 $q=\sigma(T_i)$是最长后缀 T_i 的长度，T_i 是模式 P 的一个前缀。如果下面扫描到的字符为 $T[i+1]=a$，则自动机的状态应转换为 $\sigma(T_{i+1})=\sigma(T_i a)$。该定理的证明过程说明 $\sigma(T_i a)=\sigma(P_q a)$。也就是说，为了计算 P 的前缀 $T_i a$ 的最长后缀的长度，可以先计算出 P 的前缀 $P_q a$ 的最长后缀。在每一种状态下，自动机仅需知道迄今已读入的字符串的后缀 P 的最长前缀的长度。因此，置 $\delta(q, a)=\sigma(P_q a)$可以保持所需的不变式(32.4)成立。对于上述这些非形式化的说明，我们将在稍后给出更为严格的论证。

例如，在图 32-7 所示的字符串匹配自动机中，有 $\delta(5, b)=4$。这个结论是从以下事实推导出的：如果在状态 $q=5$ 时自动机读入一个 b，则 $P_q b$=ababab，于是同时也是 ababab 的后缀 P

918

的最长前缀为 P_4=abab。

为了清楚地说明字符串匹配自动机的操作过程，我们给出一个简单而有效的程序，用来模拟这样一个自动机(用它的变迁函数 δ 来表示)，在输入文本 $T[1..n]$中，寻找长度为 m 的模式 P 的出现位置的过程。对于长度为 m 的模式的任意字符串匹配自动机来说，状态集 Q 为{0，1，…，m}，初始状态为 0，唯一的接收态是状态 m。

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1  n ← length[T]
2  q ← 0
3  for i ← 1 to n
4       do q ← δ(q, T[i])
5          if q=m
6             then print"Pattern occurs with shift"i−m
```

由 FINITE-AUTOMATON-MATCHER 的简单循环结构可以看出，对于一个长度为 n 的文本字符串，它的匹配时间为 $\Theta(n)$。但是，这一匹配时间没有包括计算变迁函数 δ 所需要的预处理时间。我们将在证明 FINITE-AUTOMATON-MATCHER 的正确性以后，再来讨论这一问题。

考察自动机在输入文本 $T[1..n]$上进行的操作。我们将证明自动机扫过字符 $T[i]$后，其状
919 态为 $\sigma(T_i)$。因为 $\sigma(T_i)=m$ 当且仅当 $P \sqsupset T_i$，所以自动机处于接受状态 m，当且仅当模式 P 已经被扫描过。为了证明这个结论，要用到下面两条关于后缀函数 σ 的引理。

引理 32.2(后缀函数不等式)　对任意字符串 x 和字符 a，有 $\sigma(xa)\leqslant\sigma(x)+1$。

证明：参照图 32-8，设 $r=\sigma(xa)$。如果 $r=0$，则根据 $\sigma(x)$非负，可知结论 $\sigma(xa)=r\leqslant\sigma(x)+1$ 显然成立，于是假定 $r>0$。现在，根据 σ 的定义有 $P_r \sqsupset xa$。所以，把 a 从 P_r 与 xa 的末尾去掉后，就得到 $P_{r-1} \sqsupset x$。因此，$r-1\leqslant\sigma(x)$，因为 $\sigma(x)$是满足 $P_k \sqsupset x$ 的最大的 k 值。 ■

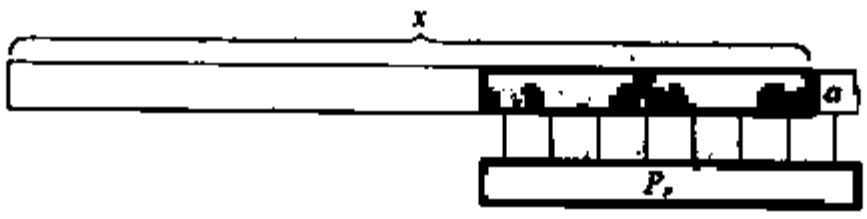
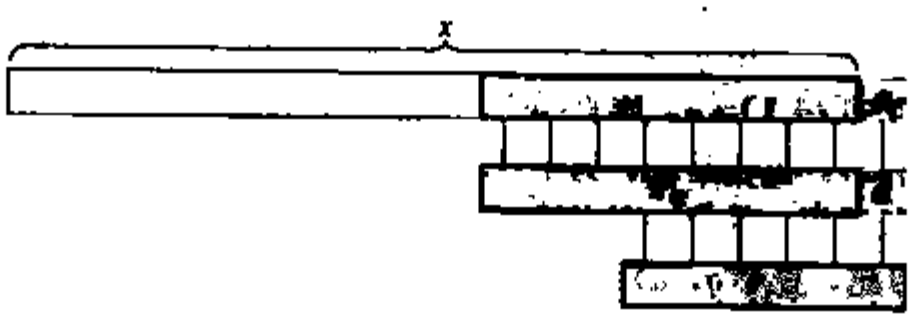


图 32-8　描述了引理 32.2 的证明。图中显示 $r\leqslant\sigma(x)+1$，其中 $r=\sigma(xa)$

引理 32.3(后缀函数递归引理)　对任意 x 和字符 a，如果 $q=\sigma(x)$，则 $\sigma(xa)=\sigma(P_qa)$。

证明：根据 σ 的定义，有 $P_q \sqsupset x$。如图 32-9 所示，$P_qa \sqsupset xa$。如果设 $r=\sigma(xa)$，由引理 32.2，得 $r\leqslant q+1$。因为 $P_qa \sqsupset xa$，$P_r \sqsupset xa$，并且 $|P_r|\leqslant|P_qa|$，所以由引理 32.1 可知 $P_r \sqsupset P_qa$。由此可得 $r\leqslant\sigma(P_qa)$，即 $\sigma(xa)\leqslant\sigma(P_qa)$。但由于 $P_qa \sqsupset xa$，所以同时有 $\sigma(P_qa)\leqslant\sigma(xa)$，这就说明 $\sigma(xa)=\sigma(P_qa)$。 ■



920 图 32-9　描述了引理 32.3 的证明。图中显示 $r=\sigma(P_qa)$，其中 $q=\sigma(x)$和 $r=\sigma(xa)$

现在，就可以来证明描述字符串匹配自动机在给定的输入文本上操作过程的主要定理了。如上所述，这个定理说明了自动机在每一步中，仅仅是记录迄今为止所读入字符串的后缀的模式的最长前缀。换句话说，自动机保持着不变式(32.4)。

定理 32.4　如果 ϕ 是字符串匹配自动机关于给定模式 P 的终态函数，$T[1..n]$是自动机的输入文本，对 $i=0, 1, \cdots, n$，有 $\phi(T_i)=\sigma(T_i)$。

证明：对 i 进行归纳。对 $i=0$，因为 $T_0=\epsilon$，定理显然为真，因此 $\phi(T_0)=0=\sigma(T_0)$。

现在假设 $\phi(T_i)=\sigma(T_i)$成立，要证明的是 $\phi(T_{i+1})=\sigma(T_{i+1})$。设 q 表示 $\phi(T_i)$，a 表示 $T[i+1]$，那么

$$\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) && \text{(根据 } T_{i+1} \text{ 和 } a \text{ 的定义)}\\
&= \delta(\phi(T_i), a) && \text{(根据 } \phi \text{ 的定义)}\\
&= \delta(q, a) && \text{(根据 } q \text{ 的定义)}\\
&= \sigma(P_q a) && \text{(根据式(32.3) 关于 } \delta \text{ 的定义)}\\
&= \sigma(T_i a) && \text{(根据引理 32.3 和归纳)}\\
&= \sigma(T_{i+1}) && \text{(根据 } T_{i+1} \text{ 的定义)}
\end{aligned}$$

根据定理 32.4，如果自动机在第 4 行进入状态 q，则 q 是满足 $P_q \sqsupset T_i$ 的最大值。因此，在第 5 行有 $q=m$，当且仅当刚刚扫描过模式 P 在文本中的一次出现位置。于是可以得出结论，FINITE-AUTOMATON-MATCHER 可以正确地运行。 ■

计算变迁函数

下列过程根据一个给定模式 $P[1..m]$来计算变迁函数 δ。 921

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1  m ← length[P]
2  for q ← 0 to m
3     do for each character a∈Σ
4           do k ← min(m+1, q+2)
5              repeat k ← k−1
6                 until P_k ⊐ P_q a
7              δ(q, a) ← k
8  return δ
```

这个过程根据定义直接计算 $\delta(q, a)$。在从第 2 行和第 3 行开始的嵌套循环中，要考察所有的状态 q 和字符 a。第 4～7 行把 $\delta(q, a)$置为满足 $P_k \sqsupset P_q a$ 的最大的 k。代码开始时 k 为最大可能的值 $\min(m, q+1)$。随着过程的执行，k 逐渐递减至 $P_k \sqsupset P_q a$。

COMPUTE-TRANSITION-FUNCTION 的运行时间为 $O(m^3 |\Sigma|)$，这是因为外循环提供了因子 $m|\Sigma|$，内层的 repeat 循环至多执行 $m+1$ 次，而第 6 行的测试 $P_k \sqsupset P_q a$ 可能需要比较 m 个字符，还存在速度更快的过程。如果能够利用精心计算出的有关模式 P 的信息，则根据 P 计算出 δ 所需要的时间可以改进为 $O(m|\Sigma|)$(参见练习 32.4-6)。如果用改进后的过程来计算 δ，则对字母表 Σ，为了找出长度为 m 的模式在长度为 n 的文本中的所有出现位置，需要 $O(m|\Sigma|)$的预处理时间和 $\Theta(n)$的匹配时间。

练习

32.3-1　对模式 $P=$ aabab 构造出相应的字符串匹配自动机，并说明它在文本字符串 $T=$ aaababaabaababaab 上的操作过程。

32.3-2　对字母表 $\Sigma=\{a, b\}$，画出与模式 ababbabbababbababbabb 相应的字符串匹配自动机的

状态转换图。

32.3-3　如果由 $P_k \sqsupset P_q$ 蕴含 $k=0$ 或 $k=q$，则称模式 P 是不可重叠的。试描述与不可重叠模式相应的字符串匹配自动机的状态转换图。

922

***32.3-4**　已知两个模式 P 和 P'，试描述如何构造一个有限自动机，使之能确定其中任意一个模式的所有出现位置。要求尽量使自动机的状态数最小。

32.3-5　已知一个包括间隔字符(参见练习 32.1-4)的某模式 P，说明如何构造一个有限自动机，使其在 $O(n)$的时间内，找出 P 在文本 T 中的一次出现位置，其中 $n=|T|$。

*32.4　Knuth-Morris-Pratt 算法

现在来介绍一种由 Knuth、Morris 和 Pratt 三人设计的线性时间字符串匹配算法。这个算法不用计算变迁函数 δ，匹配时间为 $\Theta(n)$，只用到辅助函数 $\pi[1, m]$，它是在 $\Theta(m)$时间内，根据模式预先计算出来的。数组 π 使得我们可以按需要，“现场”有效地计算(在平摊意义上来说)变迁函数 δ。粗略地说，对任意状态 $q=0, 1, \cdots, m$ 和任意字符 $a \in \Sigma$，$\pi[q]$的值包含了与 a 无关但在计算$\delta(q, a)$时需要的信息。(稍后就对这一说法加以说明。)由于数组 π 只有 m 个元素，而 δ 有 $\Theta(m|\Sigma|)$个值，所以通过预先计算 π 而不是 δ，使得时间减少了一个 Σ 因子。

关于模式的前缀函数

模式的前缀函数 π 包含有模式与其自身的位移进行匹配的信息。这些信息可用于避免在朴素的字符串匹配算法中，对无用位移进行测试，也可以避免在字符串匹配自动机中，对 δ 的预先计算过程。

考察一下朴素的字符串匹配算法的操作过程。图 32-10a 说明在模式 $P=$ababaca 和文本 T 的匹配过程中，模板的一个特定位移 s。在这个例子中，$q=5$ 个字符已经匹配成功，但模式的第 6 个字符不能与相应的文本字符匹配。q 个字符已经匹配成功的信息确定了相应的文本字符。知道

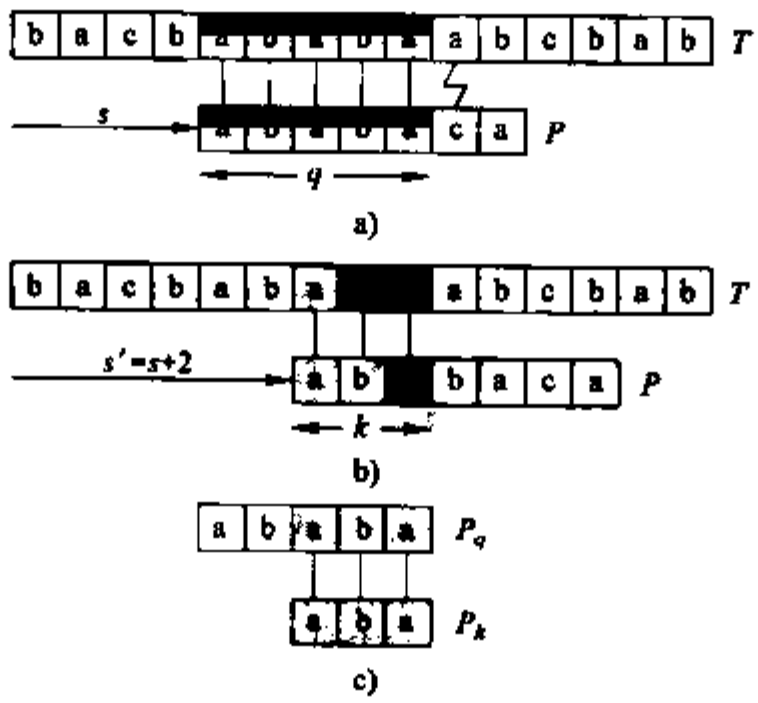


图 32-10　前缀函数 π。a)模式 $P=$ababaca 和文本 T 平行摆放，前 $q=5$ 个字符匹配。匹配的字符被打上阴影且用垂直线连接。b)根据 5 个匹配字符的已有信息，可以推知 $s+1$ 的位移是无效的。但是 $s'=s+2$ 的位移与我们对文本的了解是一致的，因而可能是有效的。c)推导中使用的有用信息可以通过模式自身的比较来预计算。这里，我们发现 P 的最长前缀同时也是 P_5 的一个真后缀是 P_3。这些信息被预先计算，并用数组 π 来表示，即 $\pi[5]=3$。在位移 s 有 q 个字符成功匹配，则下一个可能有效的位移为 $s'=s+(q-\pi[q])$

这 q 个文本字符，就使我们能够立即确定某些位移是非法的。在该图的实例中，位移 $s+1$ 是非法的，因为模式的第一个字符 a 将与已经知道与模式的第二个字符 b 匹配的文本字符进行匹配。但是，图 32-10b 所示的位移 $s'=s+2$，使模式的前面三个字符和相应的三个文本字符对齐后必定会匹配。在一般情况下，知道下列问题的答案将是很有用的：

已知模式字符 $P[1..q]$与文本字符 $T[s+1..s+q]$匹配，那么满足

$$P[1..k] = T[s'+1..s'+k] \tag{32.5}$$

其中 $s'+k=s+q$ 的最小位移 $s'>s$ 是多少？这样的位移 s'是大于 s 的未必非法的第一个位移，因为已知 $T(s+1..s+q)$。在最好的情况下，有 $s'=s+q$，因此立刻能排除位移 $s+1$，$s+2$，…，$s+q-1$。在任何情况下，对于新的位移 s'，无需把 P 的前 k 个字符与 T 中相应的字符进行比较，因为等式(32.5)已经保证它们肯定匹配。

可以用模式与其自身进行比较，以预先计算出这些必要的信息，如图 32-10c 所示。由于 $T[s'+1..s'+k]$是文本中已经知道的部分，所以它是字符串 P_q 的一个后缀。可以把等式(32.5)解释为要求满足 $P_k \sqsupset P_q$ 的最大的 $k<q$。于是，$s'=s+(q-k)$就是下一个可能的有效位移。已经证明，把已匹配字符的数目 k 存储在新的位移 s'(而不是 $s'-s$)中是比较方便的。这些信息可用于加快朴素的字符串匹配算法与有限自动机匹配器的执行速度。

923
~
924

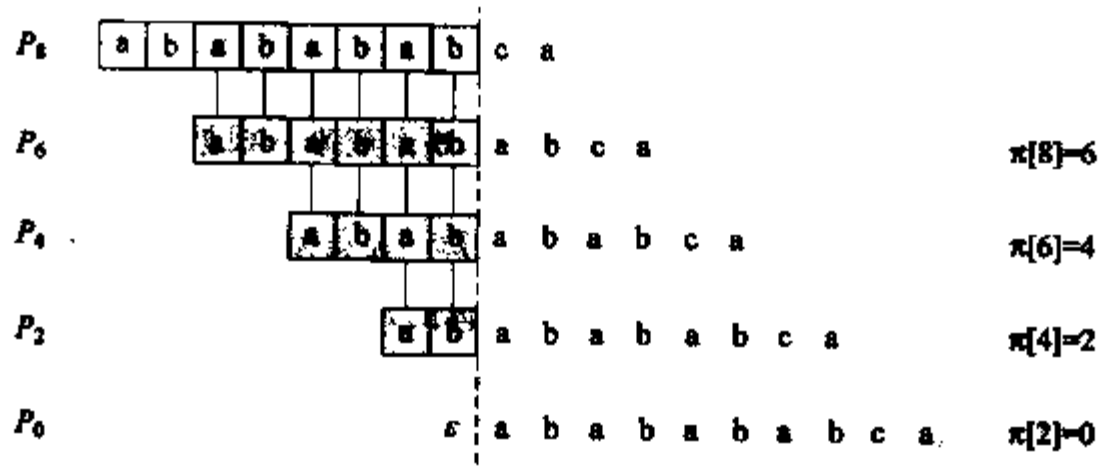
以下是预计算过程的形式化说明。已知一个模式 $P[1..m]$，模式 P 的**前缀函数**是函数 $\pi: \{1, 2, \cdots, m\} \to \{0, 1, \cdots, m-1\}$并满足

$$\pi[q] = \max\{k : k < q \text{ 且 } P_k \sqsupset P_q\}$$

即 $\pi[q]$是 P_q 的真后缀 P 的最长前缀的长度。又例如，图 32-11a 中给出了关于模式 abababababca 的完整前缀函数 π。

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

a)



b)

图 32-11　对模式 $P=$abababababca 和 $q=8$ 应用引理 32.5 的描述。a)给定模式的 π 函数。因为 $\pi[8]=6$，$\pi[6]=4$，$\pi[4]=2$ 和 $\pi[2]=0$，通过迭代 π 得到 $\pi^*[8]=\{6, 4, 2, 0\}$。b)将包含模式 P 的模板向右移动，并注意何时 P 的某前缀 P_k 与 P_8 的某后缀匹配；在 $k=6$，4，2，0 时发生。图中第一行给出了 P，点垂直线就画在 P_8 后。相继的几行都显示了 P 的位移，使得 P 的某前缀 P_k 与 P_8 的某后缀匹配。成功匹配的字符被打上了阴影。垂直线连接了并列的匹配字符。因此，$\{k : k<q \text{ 且 } P_k \sqsupset P_q\}=\{6, 4, 2, 0\}$。引理要求对所有 q 有 $\pi^*[q]=\{k : k<q \text{ 且 } P_k \sqsupset P_q\}$

925

下面给出的过程 KMP-MATCHER 的伪代码就是 Knuth-Morris-Pratt 匹配算法。我们将看到，它大部分都是在模仿过程 FINITE-AUTOMATON-MATCHER，KMP-MATCHER 调用了一个辅助过程 COMPUTE-PREFIX-FUNCTION 来计算 π。

```
KMP-MATCHER(T, P)
 1  n ← length[T]
 2  m ← length[P]
 3  π ← COMPUTE-PREFIX-FUNCTION(P)
 4  q ← 0                              ▷Number of characters matched.
 5  for i ← 1 to n                     ▷Scan the text from left to right.
 6      do while q>0 and P[q+1]≠T[i]
 7          do q ← π[q]                ▷Next character does not match.
 8        if P[q+1]=T[i]
 9          then q ← q+1               ▷Next character matches.
10        if q=m                       ▷Is all of P matched?
11          then print "Pattern occurs with shift" i−m
12            q ← π[q]                 ▷Look for the next match.
```

```
COMPUTE PREFIX FUNCTION(P)
 1  m ← length[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5      do while k>0 and P[k+1]≠P[q]
 6            do k ← π[k]
 7        if P[k+1]=P[q]
 8          then k ← k+1
 9        π[q] ← k
10  return π
```

下面先来分析这两个过程的运行时间，对其正确性的证明要复杂一些。

运行时间分析

运用平摊分析方法(参见 17.3 节)进行分析后可知，过程 COMPUTE-PREFIX-FUNCTION 的运行时间为 $\Theta(m)$。我们把 k 的势与算法中当前状态 k 联系起来。根据第 3 行，该势的初始值为 0。因为 $\pi[k]<k$，所以每当执行第 6 行时 k 值递减。但是，因为对所有 k，$\pi[k]\geqslant 0$，所以 k 不可能变成负值。对 k 值产生影响的另一行代码就是第 8 行，在每次执行 for 循环体时，该行至多使 k 增加 1。因为进行 for 循环时 $k<q$，并且在 for 循环体的每次迭代过程中，q 的值都增加，所以 $k<q$ 总成立。(根据第 9 行，这也同时说明 $\pi[q]<q$ 是正确的。)由于 $\pi[k]<k$，所以每次执行第 6
926 行中 while 循环体时，得到的结果就是势函数的相应减少。第 8 行至多把势函数增加 1，因此第 5～9 行循环体的平摊代价为 $O(1)$。由于外层循环迭代的次数为 $\Theta(m)$，并且最终的势函数至少与初始势函数一样大，所以实际最坏情况下，COMPUTE-PREFIX-FUNCTION 的全部运行时间为 $\Theta(m)$。

在类似的平摊分析中，如果用 q 的值作为势函数，则 KMP-MATCHER 的匹配时间为 $\Theta(n)$。

与 FINITE-AUTOMATON-MATCHER 相比，通过运用 π 而不是 δ，可使对模式进行预处理所需的时间由 $O(m|\Sigma|)$下降为 $\Theta(m)$，同时保持实际的匹配时间为 $\Theta(n)$。

前缀函数计算的正确性

我们先证明一个重要引理，它说明通过对前缀函数 π 进行迭代，就能够列举出是某给定前缀

P_q 的后缀的所有前缀 P_k。设

$$\pi^*[q] = \{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], \cdots, \pi^{(t)}[q]\}$$

其中 $\pi^{(t)}[q]$是按函数迭代的概念来定义的，$\pi^{(0)}[q]=q$，对 $i\geqslant 1$，$\pi^{(i+1)}[q]=\pi[\pi^{(i)}[q]]$，并且当达到 $\pi^{(t)}[q]=0$ 时，$\pi^*[q]$中的序列就终止。下面的引理描述了 $\pi^*[q]$，如图 32-11 所示。

引理 32.5(前缀函数迭代引理)　设 P 是长度为 m 的模式，其前缀函数为 π，对 $q=1, 2, \cdots, m$，有 $\pi^*[q]=\{k: k<q$ 且 $P_k \supset P_q\}$。

证明：首先证明

$$i \in \pi^*[q] \text{ 蕴含着 } P_i \supset P_q \tag{32.6}$$

若 $i\in\pi^*[q]$，则对某个 $u>0$，有 $i=\pi^{(u)}[q]$。我们通过对 u 进行归纳来证明式(32.6)成立。对 $u=1$，有 $i=\pi[q]$，因为 $i<q$ 且 $P_{\pi[q]}\supset P_q$，此断言成立。利用关系 $\pi[i]<i$ 和 $P_{\pi[i]}\supset P_i$，以及 $<$ 和 $\supset$ 的传递性，就可以证明对所有 $i\in\pi^*[q]$，因此 $\pi^*[q]\subseteq\{k: k<q$ 且 $P_k\supset P_q\}$。

我们通过引入矛盾来证明 $\{k: k<q$ 且 $P_k\supset P_q\}\subseteq\pi^*[q]$，假定相反地存在一个整数属于集合 $\{k: k<q$ 且 $P_k\supset P_q\}-\pi^*[q]$，且设 j 是最大的这样一个值。因为 $\pi[q]$是 $\{k: k<q$ 且 $P_k\supset P_q\}$中的最大值且 $\pi[q]\in\pi^*[q]$，必定有 $j<\pi[q]$。因而可以设 j' 表示 $\pi^*[q]$中比 j 大的最 927
小整数。(如果 $\pi^*[q]$中没有其他数比 j 大，则可以选取 $j'=\pi[q]$。)我们有 $P_j\supset P_q$，因为 $j\in\{k: k<q$ 且 $P_k\supset P_q\}$，另外还有 $P'_j\supset P_q$，因为 $j'\in\pi^*[q]$。因此，根据引理 32.1，$P_j\supset P'_j$，而且根据此性质 j 是小于 j' 的最大值。因而必定有 $\pi[j']=j$，且因为 $j'\in\pi^*[q]$，同样必定有 $j\in\pi^*[q]$。这就产生了矛盾，所以引理得证。 ■

算法 COMPUTE-PREFIX-FUNCTION 根据 $q=1, 2, \cdots, m$ 的顺序计算 $\pi[q]$的值。COMPUTE-PREFIX-FUNCTION 的第 2 行中的计算 $\pi[1]=0$ 当然是正确的，因为对所有 q，$\pi[q]<q$。下列引理和其推论将用于证明对 $q>1$，COMPUTE-PREFIX-FUNCTION 能正确地计算出 $\pi[q]$。

引理 32.6　设 P 是长度为 m 的模式，π 是 P 的前缀函数。对 $q=1, 2, \cdots, m$，如果 $\pi[q]>0$，则 $\pi[q]-1\in\pi^*[q-1]$。

证明：如果 $r=\pi[q]>0$，那么 $r<q$ 且 $P_r\supset P_q$；因此 $r-1<q-1$ 且 $P_{r-1}\supset P_{q-1}$(把 P_r 和 P_q 中的最后一个字符去掉)。因此，根据引理 32.5，可得 $\pi[q]-1=r-1\in\pi^*[q-1]$。 ■

对 $q=2, 3, \cdots, m$，定义子集 $E_{q-1}\subseteq\pi^*[q-1]$为：

$$\begin{aligned} E_{q-1} &= \{k\in\pi^*[q-1] : P[k+1]=P[q]\} \\ &= \{k: k<q-1 \text{ 和 } P_k\supset P_{q-1} \text{ 和 } P[k+1]=P[q]\} \quad \text{(根据引理 32.5)} \\ &= \{k: k<q-1 \text{ 和 } P_{k+1}\supset P_q\} \end{aligned}$$

集合 E_{q-1} 由满足 $P_k\supset P_{q-1}$ 和 $P_{k+1}\supset P_q$ 的 $k<q-1$ 个值组成，因为 $P[k+1]=P[q]$。因而，E_{q-1} 是由 $k\in\pi^*[q-1]$中的值组成，可以将 P_k 扩展到 P_{k+1}，并得到 P_q 合适的后缀。

推论 32.7　设 P 是长度为 m 的模式，π 是 P 的前缀函数，对 $q=2, 3, \cdots, m$，

$$\pi[q]=\begin{cases} 0 & \text{如果 } E_{q-1}=\varnothing \\ 1+\max\{k\in E_{q-1}\} & \text{如果 } E_{q-1}\neq\varnothing \end{cases}$$

证明：如果 E_{q-1} 为空，则没有用于扩展 P_k 到 P_{k+1}及得到合适 P_q 后缀的 $k\in\pi^*[q-1]$(包括 $k=0$)。因此 $\pi[q]=0$。

如果 E_{q-1}非空，那么对每一个 $k\in E_{q-1}$，有 $k+1<q$ 且 $P_{k+1}\supset P_q$。因此，根据 $\pi[q]$的定义有 928

$$\pi[q]\geqslant 1+\max\{k\in E_{q-1}\} \tag{32.7}$$

注意到 $\pi[q]>0$。设 $r=\pi[q]-1$，那么 $r+1=\pi[q]$。因为 $r+1>0$，则有 $P[r+1]=P[q]$。而且

根据引理 32.6 有 $r\in\pi^*[q-1]$。因此，$r\in E_{q-1}$，即 $r\leqslant\max\{k\in E_{q-1}\}$或等价地

$$\pi[q]\leqslant 1+\max\{k\in E_{q-1}\} \tag{32.8}$$

联合等式(32.7)和式(32.8)则推论得证。■

我们现在来完成对 COMPUTE-PREFIX-FUNCTION 计算 π 的正确性证明。在过程 COMPUTE-PREFIX-FUNCTION 中，第 4～9 行 for 循环的每次迭代开始时，有 $k=\pi[q-1]$。当第一次进入循环时，该条件由第 2 行和第 3 行实现，并且因为第 9 行的执行，使得该条件在下面的每次迭代中均保持成立。第 5～8 行调整 k 的值，使它变为现在的 $\pi[q]$的正确值。第 5～6 行的循环搜索所有 $k\in\pi^*[q-1]$的值，直至找到一个 $P[k+1]=P[q]$；此时，k 是集合 E_{q-1}中的最大值，根据推论 32.7，可以置 $\pi[q]$为 $k+1$。如果找不到这样的 k，则在第 7 行 $k=0$。如果 $P[1]=P[q]$，那么应该将 k 和 $\pi[q]$都置为 1；否则只需将 $\pi[q]$置为 0 而不管 k。第 7～9 行完成在任意条件下 k 和 $\pi[q]$的设置。这样就完成了对 COMPUTE-PREFIX-FUNCTION 正确性的证明。

KMP 算法的正确性

过程 KMP-MATCHER 可以看作是过程 FINITE-AUTOMATON-MATCHER 的一次重新实现。我们将证明 KMP MATCHER 的第 6～9 行的代码与 FINITE-AUTOMATON-MATCHER 的第 4 行代码(把 q 的值置为 $\delta(q,T[i])$)是等价的。在 KMP 算法中，并不是利用存储的值 $\delta(q,T[i])$，而是在需要时，根据 π 重新计算出该值。一旦证明了 KMP-MATCHER 模拟了 FINITE-AUTOMATON-MATCHER 的操作过程，自然也就可以由 FINITE-AUTOMATON-MATCHER 的正确性，推出 KMP-MATCHER 也是正确的(但是下面将看到为什么 KMP-MATCHER 中的第 12 行代码是必需的)。

KMP-MATCHER 的正确性可由以下断言推得：$\delta(q,T[i])=0$，或者 $\delta(q,T[i])-1\in\pi^*[q]$。为了检查该断言的正确性，设 $k=\delta(q,T[i])$。根据 δ 和 σ 的定义有 $P_k\sqsupset P_qT[i]$. 因此，去掉 P_k 和 $P_qT[i]$的最后一个字符后(这时 $k-1\in\pi^*[q]$)，或者有 $k=0$ 或者有 $k\geqslant 1$ 并且 $P_{k-1}\sqsupset P_q$。所以，或者 $k=0$，或者 $k-1\in\pi^*[q]$，断言得证。

现在按如下方法来运用该断言。设 q' 表示进入第 6 行时 q 的值。依据引理 32.5 中的等式
929 $\pi^*[q]=\{k:k<q\text{ 且 }P_k\sqsupset P_q\}$，调整迭代 $q\leftarrow\pi[q]$以枚举出$\{k:P_k\sqsupset P_{q'}\}$中的元素。第 6～9 行通过按递减顺序检查 $\pi^*[q']$中的元素，以便决定 $\delta(q',T[i])$的值。运用上述断言，代码开始时有 $q=\phi(T_{i-1})=\sigma(T_{i-1})$，并且执行迭代操作 $q\leftarrow\pi[q]$，直至找到一个 q 满足 $q=0$ 或者 $P[q+1]=T[i]$。在前一种情况下，有 $\delta(q',T[i])=0$；在后一种情况下，q 是 $E_{q'}$中的最大元素，因此由推论 32.7 有 $\delta(q',T[i])=q+1$。

在 KMP-MATCHER 中，之所以一定要有第 12 行代码，是为了避免在找出 P 的一次出现后，第 6 行中可能出现 $P[m+1]$的情形。(由练习 32.4-6 的提示：对任意 $a\in\Sigma$，有 $\delta(m,a)=\delta(\pi[m],a)$，或者等价地有 $\sigma(P_a)=\sigma(P_{\pi[m]a})$，可以推得在下一次执行第 6 行代码时，$q=\sigma(T_{i-1})$依然保持有效。)关于 Knuth-Morris-Pratt 算法的正确性证明中，其余的部分可以从 FINITE-AUTOMATON-MATCHER 的正确性推得，因为现在可以看出 KMP-MATCHER 模拟了 FINITE-AUTOMATON-MATCHER 的操作过程。

练习

32.4-1　当字母表为$\Sigma=\{a,b\}$时，计算相应于模式 ababbabbabbababbabb 的前缀函数 π。

32.4-2　给出关于 q 的函数 $\pi^*[q]$的规模的上界。举例说明所给出的上界是严格的。

32.4-3 试说明如何通过检查字符串 PT 的 π 函数，来确定模式 P 在文本 T 中的出现位置（由 P 和 T 并置形成的长度为 $m+n$ 的字符串）。

32.4-4 试说明如何通过以下方式对过程 KMP-MATCHER 进行改进：把第 7 行（不是第 12 行中）出现的 π 替换为 π'。对 $q=1, 2, \cdots, m$ 的递归定义如下：

$$\pi'[q]=\begin{cases}0 & \text{如果 } \pi[q]=0\\ \pi'[\pi[q]] & \text{如果 } \pi[q]\neq 0 \text{ 和 } P[\pi[q]+1]=P[q+1]\\ \pi[q] & \text{如果 } \pi[q]\neq 0 \text{ 和 } P[\pi[q]+1]\neq P[q+1]\end{cases}$$

试说明修改后的算法为什么是正确的，并说明在何种意义来说，这一修改是对原算法的改进。

32.4-5 写出一个线性时间的算法，以确定文本 T 是否是另一个字符串 T' 的循环旋转。例如，arc 和 car 是彼此的循环旋转。

930

***32.4-6** 给出一个有效算法，计算出相应于某给定模式 P 的字符串匹配自动机的变迁函数 δ。所给出的算法的运行时间应该是 $O(m|\Sigma|)$。（提示：证明，如果 $q=m$ 或 $P[q+1]\neq a$，则 $\delta(q, a)=\delta(\pi[q], a)$）。

思考题

32-1 基于重复因子的字符串匹配

设 y^i 表示字符串 y 与其自身并置 i 次所得的结果。例如 $(ab)^3=ababab$。如果对某个字符串 $y\in\Sigma^*$ 和某个 $r>0$ 有 $x=y^r$，则称字符串 $x\in\Sigma^*$ 具有重复因子 r。设 $\rho(x)$ 表示满足 x 具有重复因子 r 的最大值 r。

a）写出一有效算法以计算出 $\rho(P_i)$（$i=1, 2, \cdots, m$），算法的输入为模式 $P[1..m]$。算法的运行时间是多少？

b）对任何模式 $P[1..m]$，设 $\rho^*(P)$ 定义为 $\max_{1\leqslant i\leqslant m}\rho(P_i)$。证明：如果从长度为 m 的所有二进制字符串所组成的集合中随机地选择模式 P，则 $\rho^*(P)$ 的期望值是 $O(1)$。

c）论证下列字符串匹配算法可以在 $O(\rho^*(P)n+m)$ 的运行时间内，正确地找出模式 P 在文本 $T[1..n]$ 中的所有出现位置。

```
REPETITION-MATCHER(P, T)
 1  m ← length[P]
 2  n ← length[T]
 3  k ← 1+ρ*(P)
 4  q ← 0
 5  s ← 0
 6  while s≤n−m
 7       do if T[s+q+1]=P[q+1]
 8            then q ← q+1
 9                 if q=m
10                   then print "Pattern occurs with shift" s
11         if q=m or T[s+q+1]≠P[q+1]
12           then s ← s+max(1, ⌈q/k⌉)
13                q ← 0
```

931

该算法是 Galil 和 Seiferas 提出的。通过对这些设计思想进行大量地扩充，他们得到了一个线性时间的字符串匹配算法，该算法除了 P 和 T 所要求的存储空间外，仅需 $O(1)$ 的

存储空间。

本章注记

Aho，Hopcroft 和 Ullman[5]中讨论了字符串匹配与有限自动机理论的关系。Knuth-Morris-Pratt 算法[187]是由 Knuth，Pratt 和 Morris 各自独立提出的；他们合作公布了其工作成果。Rabin-Karp 算法是由 Rabin 和 Karp[175]提出的。Galil 和 Seiferas[107]给出了一个有趣的确定性
932 线性时间的字符串匹配算法，除存储模式和文本外只需用 $O(1)$ 的空间。

第 33 章　计算几何学

计算几何学是计算机科学的一个分支，专门研究那些用来解决几何问题的算法。在现代工程与数学中，计算机图形学、机器人学、VLSI 设计/计算机辅助设计以及统计学等领域中，都要用到计算几何学。计算几何学问题的输入一般是关于一组几何对象的描述，如一组点、一组线段，或者一个多边形的按逆时针顺序排列的一组顶点。输出常常是对有关这些对象的问题的回答，如是否直线相交，是否为一个新的几何对象，如顶点集合的凸包(convex hull，即最小封闭的凸多边形)。

在本章中，我们将学习一些二维的(即平面上的)计算几何学算法。在这些算法中，每个输入对象都用一组点$\{p_1, p_2, p_3, \cdots\}$来表示，其中每个 $p_i=(x_i, y_i)$，$x_i, y_i \in \mathbf{R}$。例如，一个 n 个顶点的多边形 P 可以用一组点$\langle p_0, p_1, p_2, \cdots, p_{n-1}\rangle$来表示，这些点按照在 P 的边界上出现的顺序排列。计算几何学也可以用来求解三维空间，甚至是高维空间中的问题，但这样的问题及其解决方案是很难视觉化的。不过，即使是在二维平面上，也能够看到应用计算几何学技术的一些很好的例子。

33.1 节说明如何有效而准确地回答有关线段的一些基本问题：一条线段是在与其共享一个端点的另一条线段的顺时针方向，还是在其逆时针方向？当转动两条邻接的线段时，该转向哪个方向？两条线段是否相交？33.2 节介绍一种称为“扫除”(sweeping)的技术，利用该技术设计一种运行时间为$O(n \lg n)$的算法，用来确定 n 条线段中是否包含相交的线段。33.3 节给出两种“旋转扫除”(rotational-sweep)的算法，用于计算 n 个点的凸包(最小封闭的凸多边形)。这两个算法分别是运行时间为$O(n \lg n)$的 Graham 扫描法和运行时间为 $O(nh)$的 Jarvis 步进法(h 是凸包中顶点的数目)。最后，33.4 节介绍一种运行时间为 $O(n \lg n)$的分治算法，用于在平面上的 n 个点中找出距离最近的一个点对。

33.1　线段的性质

在本章中，有好几个计算几何学的算法都要涉及线段的性质。两个不同的点 $p_1=(x_1, y_1)$ 和 $p_2=(x_2, y_2)$的凸组合是满足下列条件的任意点 $p_3=(x_3, y_3)$：对某个 α，$(0 \leqslant \alpha \leqslant 1)$，有 $x_3=\alpha x_1+(1-\alpha)x_2$，$y_3=\alpha y_1+(1-\alpha)y_2$。也可以写作 $p_3=\alpha p_1+(1-\alpha)p_2$。从直观上看，$p_3$ 是位于穿过 p_1 和 p_2 的直线上、并处于 p_1 和 p_2 之间(也包括 p_1 和 p_2 两点)的任意点。在给定两个不同的点 p_1 和 p_2 的情况下，线段$\overline{p_1 p_2}$是 p_1 和 p_2 的凸组合(convex combination)的集合。我们称 p_1 和 p_2 为线段$\overline{p_1 p_2}$的端点。有时，还要考虑到 p_1 和 p_2 之间的顺序，这时，可以说有向线段$\overrightarrow{p_1 p_2}$。如果 p_1 是原点(0，0)，则可以把有向线段$\overrightarrow{p_1 p_2}$看作为向量 p_2。

本节将讨论以下问题：

1)已知两条有向线段$\overrightarrow{p_0 p_1}$和$\overrightarrow{p_0 p_2}$，相对于它们的公共端点 p_0 来说，$\overrightarrow{p_0 p_1}$是否在$\overrightarrow{p_0 p_2}$的顺时针方向上？

2)已知两条线段$\overline{p_0 p_1}$和$\overline{p_1 p_2}$，如果先通过$\overline{p_0 p_1}$再通过$\overline{p_1 p_2}$，在点 p_1 处是不是要向左转？

3)线段$\overline{p_1 p_2}$和$\overline{p_3 p_4}$是否相交？

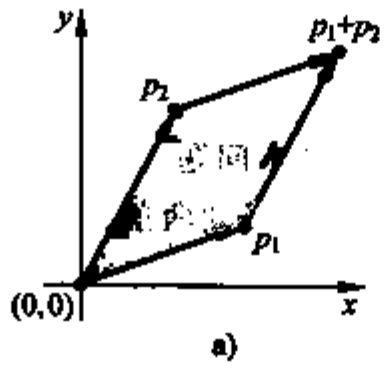
对给定的点没有任何限制。

我们可以在 $O(1)$ 时间内回答以上的每一个问题，这一点不会使人惊讶，因为每个问题的输入规模都是 $O(1)$。此外，我们将采用的方法仅限于加法、减法、乘法和比较运算。我们既不需要除法运算，也不需要三角函数，这两者的计算代价都比较高昂，并且容易产生舍入误差等问题。例如，要确定两条线段是否相交，一种"直接的"方法就是对这两条线段，都计算出形如 $y=mx+b$ 的直线方程(其中 m 为斜率，b 为 y 轴截距)，找出两条直线的交点，并检查交点是否同时在两条线段上。在这一方法中，用除法求出交点。当线段接近于平行时，算法对实际计算机中除法运算的精度非常敏感。本节中的方法避免使用除法，因而要精确得多。

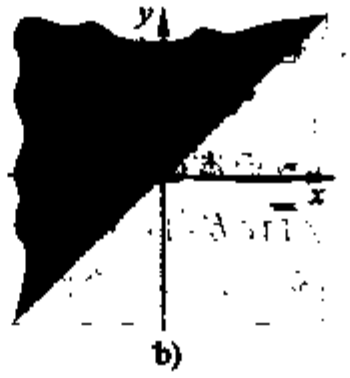
叉积

叉积(cross product)的计算是关于线段算法的中心。考察如图 33-1a 所示的向量 p_1 和 p_2。
934 可以把叉积 $p_1 \times p_2$ 看作是由点(0，0)，p_1，p_2 和 $p_1+p_2=(x_1+x_2, y_1+y_2)$ 所形成的的平行四边形的面积。另一种等价而更有用的定义是把叉积定义为一个矩阵的行列式：⊖

$$p_1 \times p_2 = \det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = -\, p_2 \times p_1$$



a)



b)

图 33-1　a)向量 p_1 和 p_2 的叉积是平行四边形的面积。b)浅阴影区域中包含了 p 的顺时针方向上的向量，深阴影区域中包含了 p 的逆时针方向上的向量

如果 $p_1 \times p_2$ 为正数，则相对于原点(0，0)来说，p_1 在 p_2 在顺时针方向上；如果 $p_1 \times p_2$ 为负数，则 p_1 在 p_2 的逆时针方向上。图 33-1b 示出了一个向量 p 的顺时针区域与逆时针区域。如果叉积为 0 的话，即出现边界情况；这时，两个向量是共线的，即它们指向同一个方向或相反的方向。

为了确定相对于公共端点 p_0，有向线段 $\overrightarrow{p_0p_1}$ 是否在有向线段 $\overrightarrow{p_0p_2}$ 的顺时针方向，只需要把 p_0 作为原点就可以了。亦即，可以用 p_1-p_0 表示向量 $p_1'=(x_1', y_1')$，其中 $x_1'=x_1-x_0$，$y_1'=y_1-y_0$。类似地可以定义 p_2-p_0，然后计算叉积：

$$(p_1-p_0)\times(p_2-p_0)=(x_1-x_0)(y_2-y_0)-(x_2-x_0)(y_1-y_0)$$

如果该叉积为正，则 $\overrightarrow{p_0p_1}$ 在 $\overrightarrow{p_0p_2}$ 的顺时针方向上；如果为负，则 $\overrightarrow{p_0p_1}$ 在 $\overrightarrow{p_0p_2}$ 的逆时针方
935 向上。

确定连续线段是向左转还是向右转

我们讨论的下一个问题是在点 p_1 处，两条连续的线段 $\overline{p_0p_1}$ 和 $\overline{p_1p_2}$ 是向左转还是向右转。亦即，我们希望找出一种方法，以确定一个给定的角 $\angle p_0p_1p_2$ 的转向。运用叉积，使得无需对角进行计算，就可以回答这个问题。如图 33-2 所示，只需要检查一下有向线段 $\overrightarrow{p_0p_2}$ 是在有向线段

⊖ 实际上，叉积是一个三维的概念。根据"右手规则"，它是一个与 p_1 和 p_2 都垂直的向量，其量值为 $|x_1y_2-x_2y_1|$。然而，在本章中，将叉积简单地看作是 $x_1y_2-x_2y_1$ 要更方便一些。

$\overrightarrow{p_0p_1}$的顺时针方向，还是在其逆时针方向。在做这一判断时，要计算出叉积$(p_2-p_0)\times(p_1-p_0)$。如果该叉积的符号为负，则$\overrightarrow{p_0p_2}$在$\overrightarrow{p_0p_1}$的逆时针方向，因此，在 p_1 点要向左转。如果叉积为正，就说明$\overrightarrow{p_0p_2}$在$\overrightarrow{p_0p_1}$的顺时针方向，因此，在点 p_1 处要向右转。叉积为 0 说明点 p_0、p_1 和 p_2 共线。

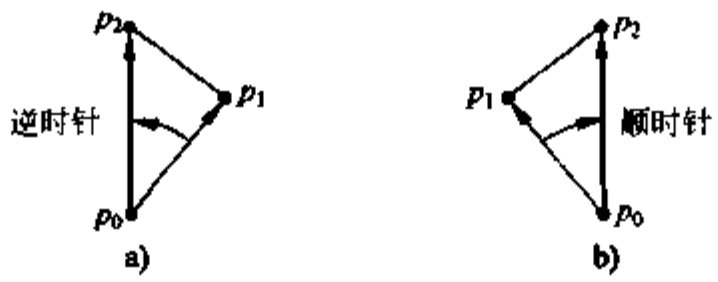


图 33-2　利用叉积来确定连续线段$\overline{p_0p_1}$和$\overline{p_1p_2}$在点 p_1 处的转向。相对于有向线段$\overrightarrow{p_0p_1}$，检查有向线段$\overrightarrow{p_0p_2}$是在其顺时针方向还是逆时针方向。a)如果是在逆时针方向，则说明在点 p_1 处向左转。b)如果是在顺时针方向，则说明向右转

确定两个线段是否相交

为了确定两个线段是否相交，要检查每个线段是否跨越(straddle)了包含另一线段的直线。给定一个线段$\overline{p_1p_2}$，如果点 p_1 位于某一直线的一边，而点 p_2 位于该直线的另一边，则称线段$\overline{p_1p_2}$跨越了该直线。如果 p_1 和 p_2 就落在该直线上的话，即出现边界情况。两个线段相交，当且仅当下面两个条件中有一个成立，或同时成立：

1)每个线段都跨越包含了另一线段的直线。

2)一个线段的某一端点位于另一线段上。(这一情况来自于边界情况。)

下面的过程即实现了这一思想。如果线段$\overline{p_1p_2}$和$\overline{p_3p_4}$相交的话，SEGMENTS-INTERSECT 即返回 TRUE(真)；如果它们不相交，则返回 FALSE(假)。它调用了子过程 DIRECTION，后者利用上面的叉积方法，计算出线段的相对方位；此外，还要利用子过程 ON-SEGMENT，用于确定一个与某一线段共线的点是否位于该线段上。 936

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
1  d1 ← DIRECTION(p3, p4, p1)
2  d2 ← DIRECTION(p3, p4, p2)
3  d3 ← DIRECTION(p1, p2, p3)
4  d4 ← DIRECTION(p1, p2, p4)
5  if((d1>0 and d2<0)or(d1<0 and d2>0))and
          ((d3>0 and d4<0)or(d3<0 and d4>0))
6    then return TRUE
7  elseif d1=0 and ON-SEGMENT(p3, p4, p1)
8    then return TRUE
9  elseif d2=0 and ON-SEGMENT(p3, p4, p2)
10   then return TRUE
11 elseif d3=0 and ON-SEGMENT(p1, p2, p3)
12   then return TRUE
13 elseif d4=0 and ON-SEGMENT(p1, p2, p4)
14   then return TRUE
15 else return FALSE
```

```
DIRECTION(p_i, p_j, p_k)
1  return(p_k − p_i)×(p_j − p_i)

ON-SEGMENT(p_i, p_j, p_k)
1  if min(x_i, x_j)≤x_k≤max(x_i, x_j)and min(y_i, y_j)≤y_k≤max(y_i, y_j)
2     then return TRUE
3     else return FALSE
```

SEGMENTS-INTERSECT 的处理流程是这样的：第 1～4 行计算每个端点 p_i 相对于另一线段的相对方位 d_i。如果所有的相对方位都非 0，则可以很容易地确定线段 $\overline{p_1p_2}$ 和 $\overline{p_3p_4}$ 是否相交，具体做法如下。如果相对于 $\overrightarrow{p_3p_4}$ 来说，有向线段 $\overrightarrow{p_3p_1}$ 和 $\overrightarrow{p_3p_2}$ 的方向相反的话，则线段 $\overline{p_1p_2}$ 跨越包含线段 $\overline{p_3p_4}$ 的直线。在这种情况下，d_1 和 d_2 的符号是不同的。类似地，如果 d_3 和 d_4 的符号不同的话，线段 $\overline{p_3p_4}$ 就跨越包含 $\overline{p_1p_2}$ 的直线。如果第 5 行中的测试结果为真的话，则两个线段互相跨越，SEGMENTS-INTERSECT 即返回 TRUE。图 33-3a 示出了这种情况。否则，两个线段不跨越对方所在的直线，但可能会出现边界情况。如果所有相对的方向都是非 0 的，则不会出现边界情况，此时第 7～13 行中所有关于是否为 0 的测试都会失败，SEGMENTS-INTERSECT 即在第 15 行返回 FALSE。图 33-3b 示出了这一情形。

如果任何相对方向 d_k 为 0 的话，则出现边界情况。此处，我们知道 p_k 与另一线段是共线
937 的。它直接位于另一线段上，当且仅当它位于另一线段的两个端点之间。过程 ON-SEGMENT 返回 p_k 是否在线段 $\overline{p_ip_j}$ 的端点之间；$\overline{p_ip_j}$ 是在第 7～13 行中调用 ON-SEGMENT 过程时的另一线段；该子过程假定是 p_k 与线段 $\overline{p_ip_j}$ 是共线的。图 33-3c、d 示出了共线点的情况。在图 33-3c 中，p_3 位于 $\overline{p_1p_2}$ 上，因而在第 12 行中，SEGMENTS-INTERSECT 返回 TRUE。在图 33-3d 中，没有哪一个端点位于另一线段上，故 SEGMENTS-INTERSECT 在第 15 行中返回 FALSE。

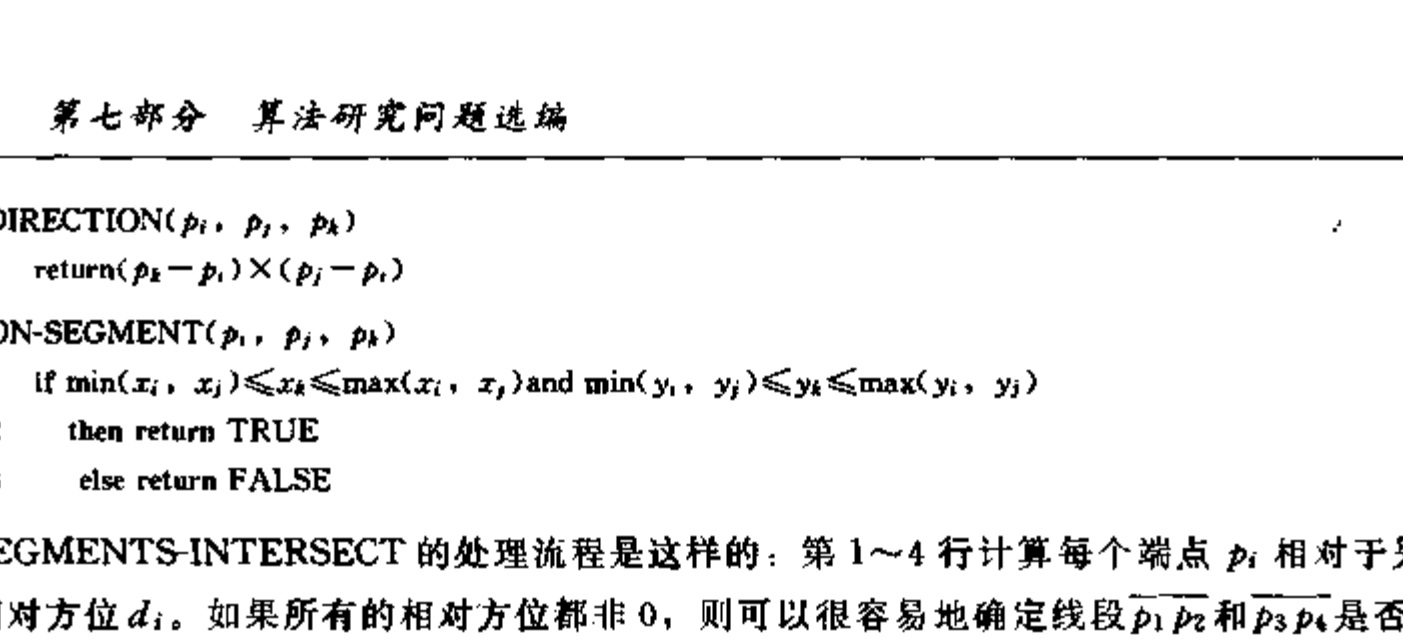


图 33-3　过程 SEGMENTS-INTERSECT 的各种情况。a)线段 $\overline{p_1p_2}$ 和 $\overline{p_3p_4}$ 互相跨越对方所在的直线。因为 $\overline{p_3p_4}$ 跨越包含 $\overline{p_1p_2}$ 的直线，故叉积 $(p_3-p_1)\times(p_2-p_1)$ 和 $(p_4-p_1)\times(p_2-p_1)$ 的符号是不同的。因为 $\overline{p_1p_2}$ 跨越包含 $\overline{p_3p_4}$ 的直线，故叉积 $(p_1-p_3)\times(p_4-p_3)$ 和 $(p_2-p_3)\times(p_4-p_3)$ 的符号是不同的。b)线段 $\overline{p_3p_4}$ 跨越包含 $\overline{p_1p_2}$ 的直线，但 $\overline{p_1p_2}$ 不跨越包含 $\overline{p_3p_4}$ 的直线。叉积 $(p_1-p_3)\times(p_4-p_3)$ 和 $(p_2-p_3)\times(p_4-p_3)$ 的符号是相同的。c)点 p_3 与 $\overline{p_1p_2}$ 共线，且位于 p_1 和 p_2 之间。d)点 p_3 与 $\overline{p_1p_2}$ 共线，但不在 p_1 和 p_2 之间。两个线段不相交

叉积的其他应用

本章后面的几小节介绍了叉积在其他方面的应用。在 33.3 节中，根据相对于某一指定原点的极角大小来对一组点进行排序。正如练习 33.1-3 将要求读者证明的那样，叉积在排序过程中，可以用于执行比较操作。在 33.2 节中，将运用红黑树来保持一组线段的垂直顺序。在这种方法中，并不是显式地记录关键字值，而是将红黑树代码中的每一次关键字值比较替换为叉积计算，以便确定与某指定垂直线相交的两条线段中，相互的上下位置。 938

练习

33.1-1 证明：如果 $p_1 \times p_2$ 为正，则相对于原点(0，0)，向量 p_1 在向量 p_2 的顺时针方向；如果叉积为负，则 p_1 在 p_2 的逆时针方向。

33.1-2 Powell 教授指出，在过程 ON-SEGMENT 的第 1 行中，只需要检测 x 坐标的值。试说明教授为什么错了。

33.1-3 一个点 p_1 相对于原点 p_0 的极角(polar angle)即在常规的极坐标系统中，向量 $p_1 - p_0$ 的极角。例如，相对于(2，4)而言，点(3，5)的极角即为向量(1，1)的极角，即 45 度或 $\pi/4$ 弧度。相对于(2，4)而言，(3，3)的极角为向量(1，−1)的极角，即 315 度或 $7\pi/4$ 弧度。请编写一段伪代码，根据相对于某个给定原点 p_0 的极角，对一个由 n 点组成的序列$\langle p_1, p_2, \cdots, p_n\rangle$进行排序。所给出过程的运行时间应为 $O(n \lg n)$，并要求用叉积来比较极角的大小。

33.1-4 试说明如何在 $O(n^2 \lg n)$的时间内，确定 n 个点中的任意三点是否共线。

33.1-5 多边形(polygon)是平面上由一系列线段构成的、封闭的曲线。亦即，它是一条首尾相连的曲线，由一系列直线段所形成。这些直线段称为多边形的边(side)。一个连接了两条连续边的点称为多边形的顶点。如果多边形是简单的(一般情况下都会作此假设)，它自身内部不会发生交叉。在平面上，由一个简单多边形包围的一组点形成了该多边形的内部(interior)，落在该多边形上的所有点形成了其边界(boundary)，而包围该多边形的所有点则形成了其外部(exterior)。对一个简单多边形来说，如果给定其边界上或内部的任意两个点，连接这两个点的线段上的所有点都被包含在该多边形的边界上或内部的话，则该多边形为凸(convex)多边形。

Amundsen 教授提出，对于由 n 个点组成的序列$\langle p_0, p_1, \cdots, p_{n-1}\rangle$，可以用下面的方法来确定它们是否能形成一个凸多边形的各连续顶点。如果集合$\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \cdots, n-1\}$(其中下标加法是模 n 的)不是既包含左转又包含右转，则输出“Yes”；否则，输出“No”。试说明虽然这种方法的运行时间为线性时间，但它并不总是能得出正确的结果。对这位教授的方法进行修改，使其总是能在线性时间内得出正确的答案。 939

33.1-6 已知一个点 $p_0 = (x_0, y_0)$，p_0 的右水平射线(right horizontal ray)是点的集合$\{p_i = (x_i, y_i) : x_i \geqslant x_0$，且 $y_i = y_0\}$，亦即，它是 p_0 点正右方的点的集合，包含 p_0 本身。试说明如何通过把问题转化为确定两条线段是否相交的问题，从而可以在 $O(1)$的时间内，确定给定的 p_0 的右水平射线是否与线段 $\overline{p_1 p_2}$ 相交。

33.1-7 要确定一个点 p_0 是否是在一个简单多边形 P(不一定是凸多边形)内部，一种方法是检查由 p_0 发出的任何射线，看它是否与多边形 P 的边界相交奇数次，但 p_0 本身不能处于 P 的边界上。试说明如何在 $\Theta(n)$的时间内，计算出点 p_0 是否在一个由 n 个顶点组成的多边形 P 的内部。(提示：利用练习 33.1-6 的结论。要确保使所给出的算法在射线与多

边形的边界相交于某顶点、在射线遮盖住多边形的一条边时，都能得出正确的结果。）

33.1-8　试说明如何在 $\Theta(n)$ 的时间内，计算出由 n 个顶点所组成的简单多边形（但不一定是凸多边形）的面积。（与多边形有关的定义可见练习 33.1-5。）

33.2　确定任意一对线段是否相交

本节要讨论用于确定一组线段中，任意两个线段是否相交的一种算法。该算法使用了一种称为“扫除”（sweeping）的技术，这种技术在许多计算几何学算法中都用到了。此外，本节末尾的练习说明了这一算法（或其简单变体）还可以用于解决其他计算几何学问题。

该算法的运行时间为 $O(n \lg n)$，其中 n 是已知的线段数目。它仅确定是否存在相交的线段，但并不输出所有的相交点。（根据练习 33.2-1，在最坏情况下，要找出 n 个线段中的所有相交点，所需的时间为 $\Omega(n^2)$。）

在扫除过程中，一条假想的垂直扫除线（sweep line）穿过已知的一组几何物体，并且通常是从左到右依次移动扫除线。扫除线移动的空间方向（在这种情况下为 x 轴方向）可以看作是一种
940 时间上先后顺序的度量。扫除技术提供了一种对一组几何物体进行排序的方法，它通常先把它们放入一个动态数据结构中，并且利用它们之间的关系对其进行排序。本节中确定线段相交的算法按从左向右的次序考察所有的线段端点，每遇到一个端点就核查是否是相交点。

为了描述确定 n 条线段中任意两条是否相交的算法并证明其正确性，我们做出了如下两条简化性假设：第一，假定没有一条输入线段是垂直的。第二，假定没有三条输入线段相交于同一点。练习 33.2-8、33.2-9 要求读者说明这一算法是足够可靠的，即只需对它略做改动，使得即使这两条假设不成立，算法也能正常工作。的确，如果去掉上面的两条简化性假设后，在为计算几何学算法编程并证明其正确性时，对边界条件的处理就常常是最棘手的部分了。

排序线段

因为上面假定了不存在垂直线段，所以任何与给定垂直扫除线相交的输入线段与其只能有一个交点。因此，可以根据交点的 y 坐标对与一给定垂直扫除线相交的线段进行排序。

更准确地说，考察两条线段 s_1 和 s_2。如果一条横坐标为 x 的垂直扫除线与这两条线段都相交，则说这两条线段在 x 是可比的。如果 s_1 和 s_2 在 x 处是可比的，并且在 x 处，s_1 与扫除线的交点比 s_2 与同一条扫除线的交点高，则说在 x 处 s_1 位于 s_2 之上，写作 $s_1 >_x s_2$。例如，在图 33-4a 中，有下列关系：$a >_r c$，$a >_t b$，$b >_t c$，$a >_t c$，$b >_u c$。线段 d 与其他任何线段都不可比。

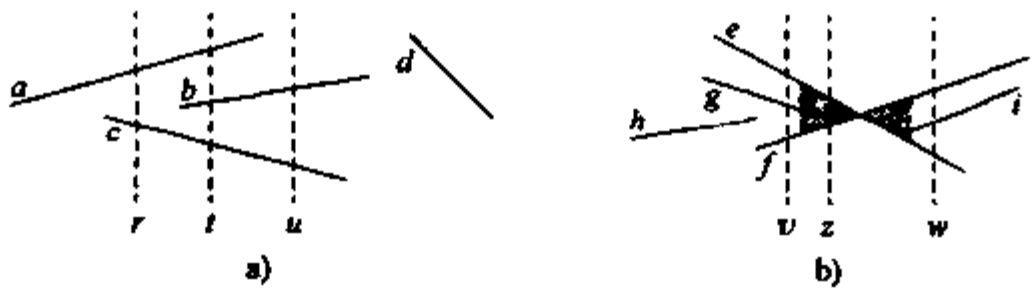


图 33-4　根据各垂直扫除线确定线段的顺序。a）图中有如下关系成立：$a >_r c$，$a >_t b$，$b >_t c$，$a >_z c$，$b >_u c$。线段 d 与其他任何线段都不可比。b）当线段 e 和 f 相交时，它们的次序颠倒了：$e >_v f$，但 $f >_w e$。任何穿过阴影区域的扫除线（如 z）都使得 e 和 f 在其全序中连续

对任意给定的 x，关系“$>_x$”是定义在那些在 x 处与扫除线相交的线段上的全序关系（参见
941 B.2 节）。但是，当线段进入和离开该排序时，随着 x 值的不同，线段的次序也可以不同。当线

段的左端点遇到扫除线时，线段就进入排序。当其右端点遇到扫除线时，就离开排序。

当扫除线穿过两条线段的交点时，会发生怎样的情况呢？如图 33-4b 所示，它们在全序中的位置被颠倒了。扫除线 v 和 w 分别位于线段 e 和 f 的交点的左边和右边，有 $e >_v f$，$f >_w e$。注意，因为我们假定没有三条线段相交于同一点，所以必有某条垂直扫除线 x，使得相交线段 e 和 f 在全序 $>_x$ 中是连续的。任何穿过图 33-4b 中阴影区域的扫除线(如 z)都使得 e 和 f 在其全序中连续。

扫除线的移动

典型的扫除算法要维护下列的两组数据：

1)扫除线状态(sweep-line status)给出了与扫除线相交的物体之间的关系。

2)事件点调度(event-point schedule)是一个从左向右排列的 x 坐标的序列，它定义了扫除线的暂停位置。称每个这样的暂停位置为事件点。扫除线状态仅在事件点处才会发生变化。

对于某些算法(例如练习 33.2-7 要求读者给出的算法)，事件点调度是随算法执行而动态地确定的。我们现在讨论的算法仅是基于输入数据的简单性质静态地确定事件点。特别地，每条线段的端点都是事件点。我们通过增加 x 坐标，并从左向右执行来对线段的端点进行排序。(如果两个或多个端点位于同一条垂直线上，亦即，它们有着相同的 x 坐标，就通过将所有共垂线的左端点置于共垂直线的右端点之前来打破"平局"。在一组共垂线的左端点中，将 y 坐标较小的放在前面，对共垂线的右端点也做同样的处理。)当遇到线段的左端点时，就把该线段插入到扫除线状态中，并且当遇到其右端点时，就把它从扫除线状态中删去。每当两条线段在全序中第一次变为连续时，就检查它们是否相交。

扫除线状态是一个全序 T，在 T 上要执行下列操作：

- INSERT(T, s)：把线段 s 插入到 T 中。
- DELETE(T, s)：把线段 s 从 T 中删除。
- ABOVE(T, s)：返回 T 中紧靠线段 s 上面的线段。
- BELOW(T, s)：返回 T 中紧靠线段 s 下面的线段。

942

如果输入中有 n 条线段，则可以运用红黑树，在 $O(\lg n)$ 时间内执行上述每个操作。读者可以回顾一下，第 13 章中所介绍的红黑树操作涉及了关键字的比较。此处可以用叉积比较来取代关键字比较，以确定两个线段的相对次序(见练习 33.2-2)。

求线段交点的伪代码

下列算法的输入是由 n 个线段组成的集合 S，如果 S 中的任何一对线段相交，算法就返回布尔值 TRUE，否则就返回布尔值 FALSE。全序 T 是由一棵红黑树来实现的。

```
ANY-SEGMENTS-INTERSECT(S)
1  T ← ∅
2  sort the endpoints of the segments in S from left to right
         breaking ties by putting left endpoints before right endpoints
         and breaking further ties by putting points with lower
         y-coordinates first
3  for each point p in the sorted list of endpoints
4       do if p is the left endpoint of a segment s
5            then INSERT(T, s)
6                 if(ABOVE(T, s)exists and intersects s)
                        or (BELOW(T, s)exists and intersects s)
7                   then return TRUE
```

```
8       if p is the right endpoint of a segment s
9         then if both ABOVE(T, s)and BELOW(T, s)exist
                    and ABOVE(T, s)intersects BELOW(T, s)
10              then return TRUE
11            DELETE(T, s)
12  return FALSE
```

图 33-5 说明了这一算法的执行过程。第 1 行初始化全序为空。第 2 行通过对 $2n$ 个线段端点从左向右进行排序，对 x 坐标相同的端点，把 y 坐标较小的点放在前面，最后确定事件点调度。注意，可以通过在(x, e, y)上对端点按词典顺序进行排序，来执行第 2 行的操作，此处 x、y 为通常的坐标，$e=0$ 表示左端点，$e=1$ 表示右端点。

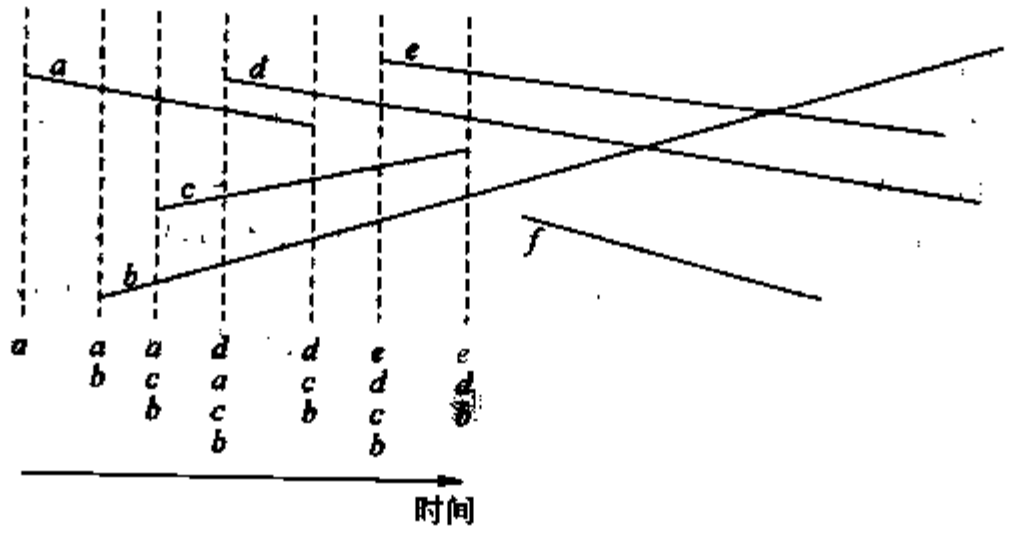


图 33-5　ANY-SEGMENTS-INTERSECT 的执行过程。每条虚线都是一个事件点处的扫除线，每条扫除线下的线段名排序为 for 循环结束时的全序 T，在该循环中，要对各对应的事件点进行处理。当线段 c 被删除时，即可发现线段 d 和 b 相交

在第 3～11 行的 for 循环中，每次迭代均对一个事件点 p 进行处理。如果 p 是线段 s 的左端点，则第 5 行把 s 加到全序中；如果 s 与在由经过 p 的扫除线定义的全序中与其连续的两条线段中的任何一条相交，则第 6～7 行返回 TRUE。(如果 p 位于另一线段 s' 上，则出现边界情形。在这种情况下，仅要求将 s 和 s' 连续地放入 T 中。)如果 p 是线段 s 的右端点，则把 s 从全序中删除。在由穿过 p 的扫除线所定义的全序中，如果 s 旁边的两条线段间有交点，则第 9～10 行返回 TRUE；当 s 被删除后，这些线段在全序中就变为连续。如果这些线段不相交，则第 11 行把线段 s 从全序中删除。最后，如果在处理完全部 $2n$ 个事件点后没有发现线段相交，第 12 行就返回 FASLE。

正确性

为了说明 ANY-SEGMENTS-INTERSECT 是正确的，我们将证明当且仅当 S 中的线段有一个交点时，对 ANY-SEGMENTS-INTERSECT(S)的调用返回 TRUE。

很容易看出，仅当 ANY-SEGMENTS-INTERSECT 发现两个输入线段之间的一个交点时，它才返回 TRUE(在第 7 行和第 10 行中)。于是，如果它返回 TRUE，就说明存在一个交点。

另外，还需要说明以上结论的逆结论：如果存在一个交点，ANY-SEGMENTS-INTERSECT 就会返回 TRUE。假定至少有一个交点。设 p 为最左边的交点，当出现两个最左边的顶点时，可以选择 y 坐标最小的那个交点。设 a 和 b 为相交于 p 的两个线段。因为在 p 的左边没有线段相交，因而，对于在 p 左边的所有点来说，由 T 给出的顺序是正确的。又因为没有三条线段相交

943
~
944

于同一点，故存在一条扫除线 z，在该扫除线上线段 a 和 b 成为全序中的连续线段。[⊖]此外，z 位于 p 的左边，或者穿过 p。在扫除线 z 上，存在一个线段端点 q，它是 a 和 b 在全序中成为连续线段的事件点。如果 p 在扫除线 z 上，则 $q=p$；如果 p 不在扫除线 z 上，q 就位于 p 的左边。不管是这两种情况中的哪一种，在 q 被遇到之前，T 给出的顺序都是正确的。(正是在这儿，算法按字典顺序对事件点进行了排序。因为 p 是所有最左边的交点中位置最低的，故即使 p 位于扫除线 z 上，且有另一个交点 p' 也位于 z 上，事件点 $q=p$ 也能在另一个交点 p' 对全序 T 产生干扰之前得到处理。此外，即使 p 是某一线段(例如 a)的左端点，同时是另一线段(例如 b)的右端点，因为左端点事件发生在右端点事件之前，故当线段 a 被首次遇到之前，线段 b 已经在全序 T 中了。)事件点 q 或者被 ANY-SEGMENTS-INTERSECT 处理，或者不被它处理。

如果 q 由 ANY-SEGMENTS-INTERSECT 进行了处理，处理时采取的动作有两种可能：

1)a 或 b 被插入到 T 中，则另一线段在全序中位于它的上面或下面。第 4～7 行可以发现这种情况。

2)线段 a 和 b 都已经在 T 中，在全序中，它们之间的一个线段被删除，使得 a 和 b 成为连续线段。第 8～11 行可以发现这种情况。

无论是这两种情况中的哪一种，都能够发现交点 p，且 ANY-SEGMENTS-INTERSECT 返回 TRUE。

如果事件点 q 没有被 ANY-SEGMENTS-INTERSECT 处理，该过程必定在处理所有的事件点之前已经返回。仅当 ANY-SEGMENTS-INTERSECT 已经找到了一个交点并返回 TRUE 时，才会出现这种情况。

于是，如果存在着一个交点，ANY-SEGMENTS-INTERSECT 就返回 TRUE。我们已经看到，如果 ANY-SEGMENTS-INTERSECT 返回 TRUE，则必定存在一个交点。因此可以说，ANY-SEGMENTS-INTERSECT 始终能返回正确的结果。 945

运行时间

如果集合 S 中有 n 条线段，则 ANY-SEGMENTS-INTERSECT 的运行时间为 $O(n\lg n)$。第 1 行的运行需要 $O(1)$的时间。如果使用合并排序或堆排序，则执行第 2 行所需的时间为 $O(n\lg n)$。由于总共有 $2n$ 个事件点，所以第 3～11 行的 for 循环至多迭代 $2n$ 次。每次迭代所需的时间为 $O(\lg n)$，这是因为每个红黑树操作所需的时间为 $O(\lg n)$。运用 33.1 节中的方法，则可以使每次相交测试所需的时间为 $O(1)$。因此，总的运行时间为 $O(n\lg n)$。

练习

33.2-1　证明：在 n 条线段的集合中，可能有 $\Theta(n^2)$个交点。

33.2-2　已知两条在 x 处可比的线段 a 和 b，试说明如何在 $O(1)$时间内确定 $a>_x b$ 和 $b>_x a$ 中哪一个成立。假定这两条线段都不是垂直的。(提示：如果 a 和 b 不相交，利用叉积即可。如果 a 和 b 相交(当然也可以用叉积来确定)，仍然可以只利用加、减、乘这几种运算，而无需用到除法。当然，在应用 $>_x$ 关系时，如果 a 和 b 相交，就可以停下来并声明已找到了一个交点。)

33.2-3　Maginot 教授建议修改过程 ANY-SEGMENTS-INTERSECT，使其不是找出一个交点后

⊖ 如果允许三个线段相交于同一点上，就可能会有一个干扰的线段 c，它与 a 和 b 都交叉于点 p 处。亦即，对所有位于 p 左边的、满足 $a<_w b$ 的扫除线 w，都可能有 $a<_w c$ 和 $c<_w b$。练习 33.2-8 要求读者证明即使三个线段的确相交在同一点上，ANY-SEGMENTS-INTERSECT 也是正确的。

就返回，而是输出相交的线段，再继续进行 for 循环的下一次迭代。他称这样得到的过程为 PRINT-INTERSECTING-SEGMENTS，并声称该过程能够按照线段在集合中出现的次序，从左到右输出所有的交点。试说明这位教授的说法有两点是错的，即举出一组线段，使得运用过程 PRINT-INTERSECTING-SEGMENTS 所找出的第一个相交点不是最左相交点，再举出一组线段，使过程 PRINT-INTERSECGING-SEGMENTS 不能找出所有的相交点。

33.2-4 写出一个运行时间为 $O(n \lg n)$ 的算法，以确定由 n 个顶点组成的多边形是否是简单多边形。

33.2-5 写出一个运行时间为 $O(n \lg n)$ 的算法，以确定总共有 n 个顶点的两个简单多边形是否相交。

946

33.2-6 一个圆面是由一个圆加上其内部所组成，并且用圆心和半径表示。如果两个圆面有任何公共点，则称这两个圆面相交。写出一个运行时间为 $O(n \lg n)$ 的算法，以确定 n 个圆面中是否有任何两个圆面相交。

33.2-7 已知 n 条线段中总共包含 k 个交点，试说明如何在 $O((n+k)\lg n)$ 时间内，输出全部 k 个交点。

33.2-8 论证即使有三条或更多的线段相交于同一点，过程 ANY-SEGMENTS-INTERSECT 也能正确执行。

33.2-9 证明在有垂直线段的情况下，如果某一垂直线段的底部端点被当作是左端点来处理，其顶部端点被当作是右端点来处理，则过程 ANY-SEGMENTS-INTERSECT 也能正确执行。如果允许有垂直线段的话，你对练习 33.2-2 的回答应如何修改？

33.3　寻找凸包

点集 Q 的凸包(convex hull)是一个最小的凸多边形 P，满足 Q 中的每个点或者在 P 的边界上，或者在 P 的内部。(凸多边形的准确定义可见练习 33.1-5。)我们用 $CH(Q)$ 来表示 Q 的凸包。形象一点说，可以把 Q 中的每个点都想像成是露在一块板外的铁钉，那么凸包就是包围了所有这些铁钉的一条拉紧了的橡皮绳所构成的形状。图 33-6 示出了一组点及其凸包。

本节要介绍两种算法，它们都用于计算包含 n 个点的点集的凸包。两种算法都按逆时针方向的顺序输出凸包的各个顶点。第一种算法称为 Graham 扫描法(Graham's scan)，运行时间为 $O(n \lg n)$。第二种算法称为 Jarvis 步进法(Jarvis march)，其运行时间为 $O(nh)$，h 为凸包中的顶点数。从图 33-6 中可以看出，$CH(Q)$ 的每一个顶点都是 Q 中的点。两种算法都利用了这一性质，来决定应该保留 Q 中的哪些点作为凸包的顶点，以及应该去掉 Q 中的哪些点。

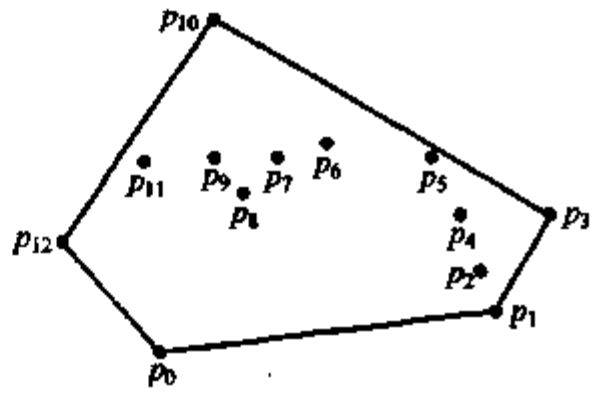


图 33-6　点集 $Q=\{p_0, p_1, \cdots, p_{12}\}$ 及其以灰色显示的凸包 $CH(Q)$

事实上，有好几种方法都能在 $O(n \lg n)$ 时间内计算凸包。Graham 扫描法和 Jarvis 步进法都运用了一种称为“旋转扫除”(rotational sweep)的技术，根据每个顶点对一个参照顶点的极角的大小，依次进行处理。其他方法有以下几种：

- 在增量方法(incremental method)中，对点从左到右进行排序后，得到一个序列 $\langle p_1, p_2, \cdots, p_n \rangle$，在第 i 步，根据左起第 i 个点，对 $i-1$ 个最左边的点的凸包 $CH(\{p_1, p_2, \cdots, p_{i-1}\})$ 进行更新，从而形成 $CH(\{p_1, p_2, \cdots, p_i\})$。练习 33.3-6 要求读者证

明实现这种方法所需的全部时间为 $O(n \lg n)$。

- 在分治法(divide-and-conquer method)中，在 $\Theta(n)$时间内，n 个点组成的集合被划分为两个子集，分别包含最左边的$\lceil n/2 \rceil$和最右边的$\lfloor n/2 \rfloor$个点，并对子集的凸包进行递归计算，然后利用一种巧妙的办法，在 $O(n)$时间内对计算出来的凸包进行组合。这种方法的运行时间用大家熟悉的递归式 $T(n)=2T(n/2)+O(n)$来表示，因此，这一分治法的运行时间为 $O(n \lg n)$。
- 剪枝-搜索方法(prune-and-search method)类似于 9.3 节中讨论的最坏情况下线性时间的中值算法。它通过反复丢弃剩余点中固定数量的点，来寻找凸包的上部(或称"上链")，直至仅剩下凸包的上链。然后，再执行同样的操作以找出下链。从渐近意义上来看，这种方法速度最快，如果凸包包含 h 个顶点的话，则该方法的运行时间仅为 $O(n \lg h)$。

计算一组点的凸包本身就是一个有趣的问题。其他一些关于计算几何学问题的算法都始于对凸包的计算。例如，考虑二维的最远点对问题：已知平面上的 n 个点的集合，希望找出它们中彼此之间距离最远的两个点。练习 33.3-3 要求读者证明这两个点必定是凸包的顶点。尽管在此不作证明，但我们知道，要找出 n 个顶点的凸多边形中最远顶点对，需要 $O(n)$的时间。因此，通过在 $O(n \lg n)$时间内计算出 n 个输入点的凸包，然后再找出得到的凸多边形中的最远顶点对，就可以在 $O(n \lg n)$的时间内，找出任意 n 个点组成的集合中距离最远的点对。

947
~
948

Graham 扫描法

Graham 扫描法(Graham's scan)通过设置一个关于候选点的堆栈 S 来解决凸包问题。输入集合 Q 中的每个点都被压入栈一次，非 CH(Q)中顶点的点最终将被弹出堆栈。当算法终止时，堆栈 S 中仅包含 CH(Q)中的顶点，其顺序为各点在边界上出现的逆时针方向排列的顺序。

过程 GRAHAM-SCAN 的输入为点集 Q，$|Q| \geq 3$，它调用函数 TOP(S)，以便在不改变堆栈 S 的情况下，返回处于栈顶的点，并调用函数 NEXT-TO-TOP(S)来返回处于堆栈顶部下面的那个点，且不改变栈 S。我们稍后将证明：过程 GRAHAM-SCAN 返回的堆栈 S 从底部到顶部，依次是按逆时针方向排列的 CH(Q)中的顶点。

```
GRAHAM-SCAN(Q)
1  let p0 be the point in Q with the minimum y-coordinate,
          or the leftmost such point in case of a tie
2  let(p1, p2, ···, pm)be the remaining points in Q,
          sorted by polar angle in counterclockwise order around p0
          (if more than one point has the same angle, remove all but
          the one that is farthest from p0)
3  PUSH(p0, S)
4  PUSH(p1, S)
5  PUSH(p2, S)
6  for i ← 3 to m
7      do while the angle formed by points NEXT-TO-TOP(S), TOP(S),
                  and pi makes a nonleft turn
8             do POP(S)
9          PUSH(pi, S)
10 return S
```

图 33-7 说明了 GRAHAM-SCAN 的执行过程。第 1 行选取 p_0 作为 y 坐标最小的点，如果有数个这样的点，则选取最左边的点作为 p_0。由于 Q 中没有其他点比 p_0 更低，并且与其有相同 y 坐标的点都在它的右边，所以 p_0 是 CH(Q)的一个顶点。第 2 行根据 Q 中剩余的点相对于 p_0 的极角对它们进行排序，所用的方法(比较叉积)与练习 33.1-3 中相同。如果有两个或更多的点相

949 对 p_0 的极角相同，那么除了与 p_0 距离最远的点以外，其余各点都是 p_0 与该最远点的凸组合。因此，我们可以完全不考虑这些点。设 m 表示除 p_0 以外剩余的点的数目。Q 中每个点关于 p_0 的极角(用弧度表示)属于半开区间$[0, \pi)$中。由于对各个点是按其极角来排序的，故可以把这些点按相对于 p_0 的逆时针方向进行排序。我们将这一由点组成的有序序列表示为$\langle p_1, p_2, \cdots, p_m\rangle$。注意，点 p_1 和 p_m 都是 CH(Q)中的顶点(参见练习 33.3-1)。图 33-7a 说明了图 33-6 中的点按相对于 p_0 的极角进行递增排序得到的序列。

过程中的剩余部分运用了堆栈 S。第 3～5 行对堆栈进行初始化，使其从底到顶依次包含前面三个点 p_0、p_1 和 p_2。图 33-7a 说明了初始的堆栈 S。第 6～9 行的 for 循环对序列$\langle p_3, p_4, \cdots, p_m\rangle$中的每一个点进行一次迭代。算法的意图是在对点 p_i 进行处理后，在栈 S 中，按由底到顶的的顺序，包含 CH($\{p_0, p_1, \cdots, p_i\}$)中按逆时针方向排列的各个顶点。第 7～8 行的 while 循环把发现不是凸包中的顶点的点从堆栈中移去。沿逆时针方向通过凸包时，在每个顶点处应该向左转。因此，while 循环每次发现在一个顶点处没有向左转时，就把该顶点从堆栈中弹出。(仅检查不向左转的情况，而不是对向右转进行检查，这样的测试就排除了在所形成的凸包的某个顶点处为直角的可能性。这正是我们所希望的，因为一个凸多边形的每个顶点不能是该多边形中其他顶点的凸组合。)当算法向点 p_i 推进、在已经弹出了所有非左转的顶点后，就把 p_i 压入堆栈中。图 33-7b～k 示出了 for 循环的每次迭代后，堆栈 S 的状态。最后，GRAHAM-SCAN 在第 10 行返回堆栈 S。图 33-7l 示出了相应的凸包。

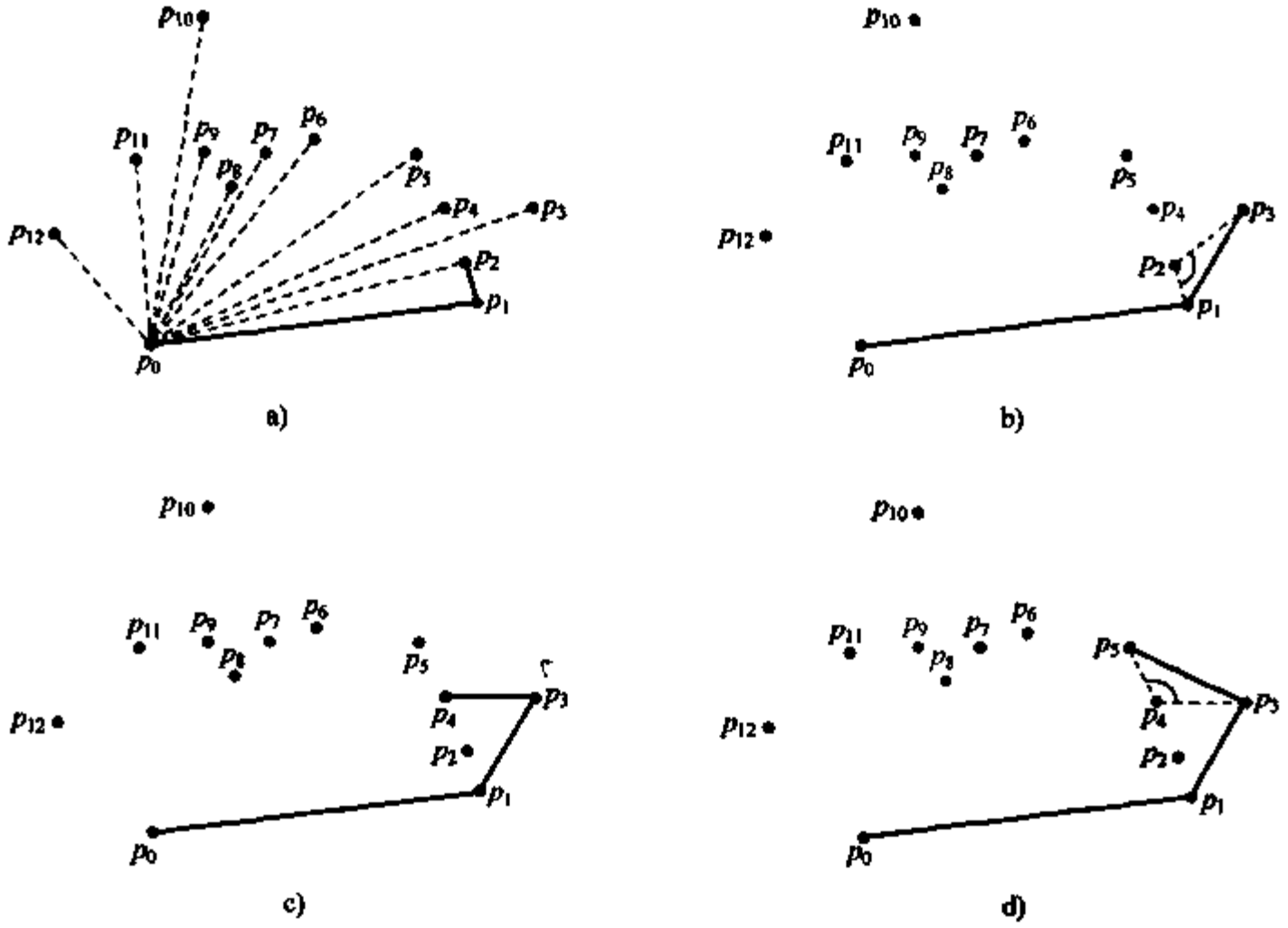
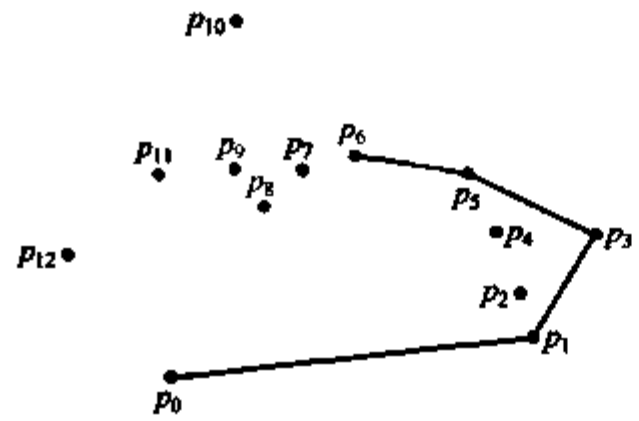
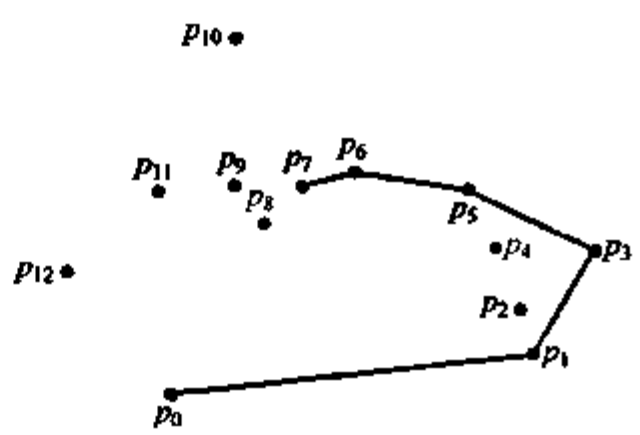


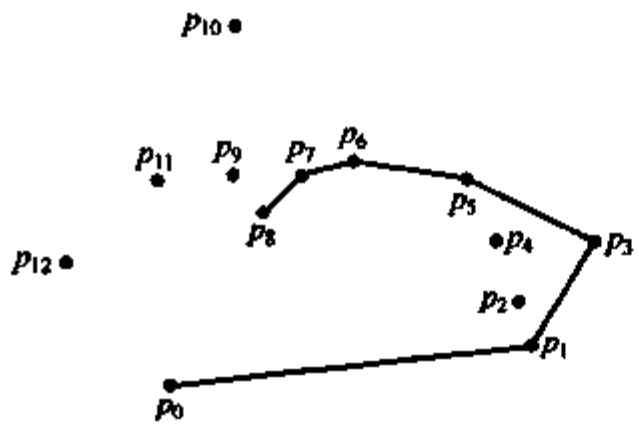
图 33-7　GRAHAM-SCAN 在图 33-6 所示的集合 Q 上的执行过程。在每一步中，栈 S 中包含的当前凸包以灰色示出。a)一组点的序列$\langle p_1, p_2, \cdots, p_{12}\rangle$，按相对于点 p_0 的极角大小的递增顺序编号，初始的栈 S 中包含 p_0、p_1 和 p_2。b)至 k)中第 6～9 行中 for 循环的每一轮迭代后栈 S 的情况。虚线示出的是非左转，它们会引发将点从栈中弹出。例如，h)中，角 $\angle p_7 p_8 p_9$ 处的右转使得 p_8 被弹出，接着角 $\angle p_6 p_7 p_9$ 处的右转使得 p_7 被弹出。l)过程返回的凸包与图 33-6 中匹配



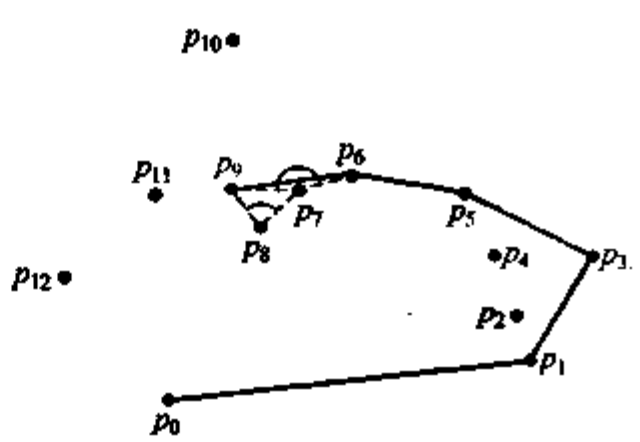
e)



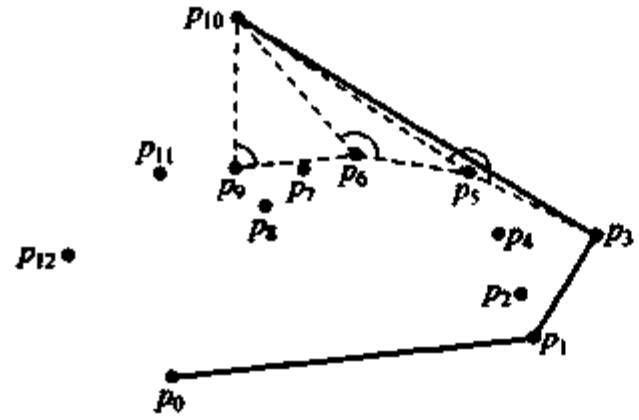
f)



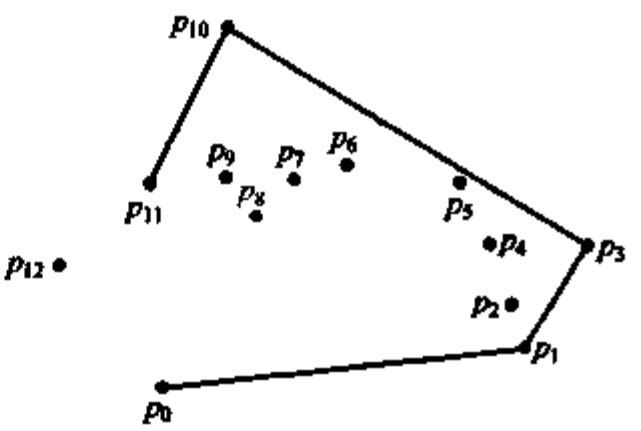
g)



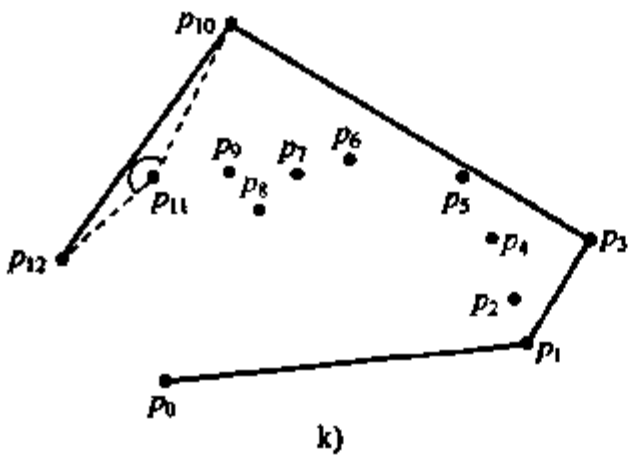
h)



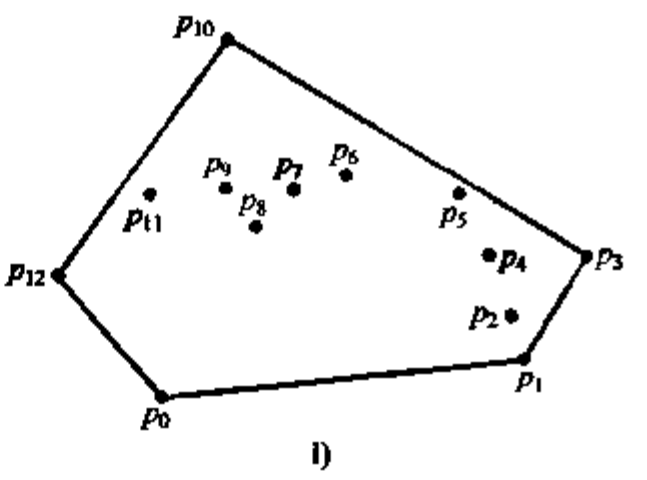
i)



j)



k)



l)

图 33-7　（续）

下面的定理给出了 GRAHAM-SCAN 算法正确性的形式化证明。

定理 33.1(Graham 扫描法的正确性)　如果在一个点集 Q 上运行 GRAHAM-SCAN，其中 $|Q| \geqslant 3$，则在过程终止时，栈 S 从底到顶，按逆时针方向顺序包含了 $CH(Q)$ 中的各个顶点。

证明：在第 2 行之后，有点序列$\langle p_1, p_2, \cdots, p_m\rangle$。对 $i=2, 3, \cdots, m$，定义一个子点集 $Q_i=\{p_0, p_1, \cdots, p_i\}$。包含在 $Q-Q_m$ 中的是那些被删除的点，因为它们与 Q_m 中的某个点有着相对于 p_0 的相同极角；这些点不在 CH(Q)中，因而 CH(Q_m)=CH(Q)。于是，我们只要证明当 GRAHAM-SCAN 终止时，栈 S 中包含了 CH(Q_m)中的顶点，且这些顶点是按照逆时针顺序，从底至顶在栈中排列。请注意正如 p_0、p_1 和 p_m 是 CH(Q)中的顶点一样，点 p_0、p_1 和 p_i 也都是 CH(Q_i)中的顶点。

950
l
952

证明中要用到如下的循环不变式：

在第 6～9 行中 for 循环的每一轮迭代开始时，栈 S 中从底至顶，恰包含了 CH(Q_{i-1})中按逆时针顺序排列的各个顶点。

初始化：在首次执行第 6 行时，这个循环不变式是保持的，因为此时栈 S 中恰包含了 $Q_2=Q_{i-1}$ 中的顶点，这三个顶点形成了它们自己的凸包。此外，它们按逆时针顺序，从底至顶地出现在栈 S 中。

保持：进行 for 循环的一轮迭代后，栈 S 顶上的点为 p_{i-1}，它是在上一迭代最后(或在第一次迭代开始之前，当 $i=3$ 时)被压入栈的。设第 7～8 行中 while 循环执行后、第 9 行将 p_i 压入栈之前，栈 S 顶的点为 p_j，设 p_k 为栈 S 中紧靠着 p_j 之下的点。在 p_j 成为栈顶点、且尚未将 p_i 压入栈的那个时刻，栈 S 中包含了与 for 循环的第 j 轮迭代后一样的点。因此，根据循环不变式，在该时刻，栈 S 中恰包含了 CH(Q_j)中的顶点，它们按逆时针顺序，自底向上地出现地栈中。

我们继续关注当 p_i 被压入栈之前的这一时刻。参考图 33-8a，因为 p_i 相对于 p_0 的极角大于 p_j 的极角，并且，由于角$\angle p_k p_j p_i$ 是向左转的(否则 p_j 就应该已经被弹出了)，可以看出，由于 S 恰包含了 CH(Q_j)中的顶点，因此，一旦压入 p_i，栈 S 中恰包含了 CH($Q_j \cup \{p_i\}$)中的顶点，并且，这些点仍然是按逆时针顺序、自底向上地出现在栈中的。

现在，我们已经证明了 CH($Q_j \cup \{p_i\}$)与 CH(Q_i)是同一个点集。考虑任意一个点 p_t 在 for 循环的第 i 轮迭代中被弹出，另设 p_r 为 p_t 被弹出时，栈 S 中紧靠在 p_t 下面的点(p_r 可以是 p_j)。角$\angle p_r p_t p_i$ 所做的是非左转向，而 p_t 相对于是 p_0 的极角大于 p_r 的极角。如图 33-8b 中所示，p_t 必定在由 p_0、p_r 和 p_i 构成的三角形内部，或者在这个三角形的某一条边上(但它不是该三角形的一个顶点)。显然，由于 p_t 是在一个由 Q_i 的其他三个点所构成的三角形内部，故不可能是 CH(Q_i)的一个顶点。正是因为 p_t 不是 CH(Q_i)的一个顶点，因而有：

$$\mathrm{CH}(Q_i - \{p_t\}) = \mathrm{CH}(Q_i) \tag{33.1}$$

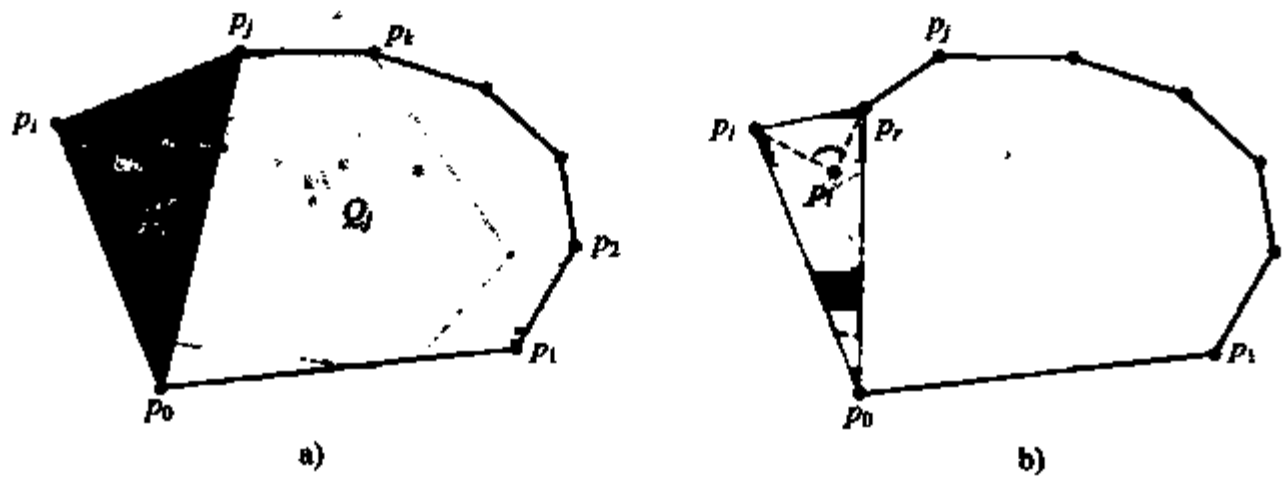


图 33-8　GRAHAM-SCAN 的正确性的证明过程 a)因为 p_i 相对于 p_0 的极角大于 p_j 的极角，又因为角$\angle p_k p_j p_i$ 是向左转的，故将 p_i 加入 CH(Q_j)中就得到 CH($Q_j \cup \{p_i\}$)中的各顶点。b)如果角$\angle p_r p_t p_i$ 执行的是非左的转向，则 p_t 或者是在由 p_0、p_r 和 p_i 所构成的三角形内部，或者是在该三角形的一条边上，它不可能是 CH(Q_i)的一个顶点

设 P_i 为 for 循环的第 i 轮迭代中弹出的点的集合。由于等式(33.1)适用于 P_i 中的所有点，因此，可以反复地应用它，来说明 $CH(Q_i-P_i)=CH(Q_i)$。但是，$Q_i-P_i=Q_j\cup\{p_i\}$，于是，可以得出这样的结论，即 $CH(Q_j\cup\{p_i\})=CH(Q_i-P_i)=CH(Q_i)$。

上面已经证明了一旦将 p_i 压入栈后，栈 S 中就恰包含 $CH(Q_i)$中的顶点，并且是按逆时针顺序、自栈底向上排列的。增加 i 的值将使得循环不变式对循环的下一轮迭代也保持成立。

终止：当循环终止时，有 $i=m+1$，因而，循环不变式蕴含着栈 S 中恰包含了 $CH(Q_m)$(即 $CH(Q)$)中的顶点，并且，这些顶点是按逆时针顺序、从栈底向上排列的。证毕。 ■

现在来证明 GRAHAM-SCAN 的运行时间为 $O(n\lg n)$，其中 $n=|Q|$。执行第 1 行代码需要$\Theta(n)$的时间。如果运用合并排序或堆排序对极角进行排序，并用 33.1 节中的叉积方法对极角进行比较，那么执行第 2 行代码所需的时间为 $O(n\lg n)$。(对极角相同的点，除最远点以外的其他点都被去掉，完成这一操作所需的时间为 $O(n)$。)第 3～5 行执行时间为 $O(1)$。因为 $m\leqslant n-1$，所以第 6～9 行的 for 循环至多执行 $n-3$ 次。因为 PUSH 的执行时间为 $O(1)$，所以，除了花在第 7～8 行的 while 循环上的时间外，每一轮迭代还需要 $O(1)$的时间。于是，除去执行嵌套 while 循环所需的时间外，整个 for 循环的执行时间为 $O(n)$。

下面运用聚集方法来证明执行整个 while 循环所需的时间为 $O(n)$。对 $i=0,1,\cdots,m$，每个点 p_i 恰好都被压入堆栈 S 中一次。如在 17.1 节中对过程 MULTIPOP 的分析那样，我们注意到对每个 PUSH 操作，至多存在一个 POP 操作。至少有三个点(p_0，p_1 和 p_m)不会从堆栈中弹出，所以，事实上总共至多执行 $m-2$ 次 POP 操作。While 循环中每次迭代时执行一次 POP 操作，因此，while 循环总共至多执行 $m-2$ 次迭代。由于第 7 行的测试所需时间为 $O(1)$，每次调用 POP 所需的时间为 $O(1)$，并且由于 $m\leqslant n-1$，所以执行 while 循环所需的全部时间为 $O(n)$。因此，过程 GRAHAM-SCAN 的运行时间为 $O(n\lg n)$。

953
~
954

Jarvis 步进法

Jarvis 步进法(Jarvis march)运用了一种称为打包(package wrapping)的技术来计算一个点集 Q 的凸包。算法的运行时间为 $O(nh)$，其中 h 是 $CH(Q)$中的顶点数。当 h 为 $o(\lg n)$时，Jarvis 步进法在渐近意义上比 Graham 扫描法的速度更快些。

从直观上看，可以把 Jarvis 步进法想像成在集合 Q 的外面紧紧地包了一层纸。开始时，把纸的末端粘在集合中最低的点上，即粘在与 Graham 扫描法开始时相同的点 p_0 上。该点为凸包的一个顶点。把纸拉向右边使其绷紧，然后再把纸拉高一些，直到碰到一个点。该点也必定是凸包中的一个顶点。使纸保持绷紧状态，用这种方法继续围绕顶点集合，直至回到原始点 p_0。

更形式地说，Jarvis 步进法构造了 $CH(Q)$的顶点序列 $H=(p_0,p_1,\cdots,p_{h-1})$，其中 p_0 为起始点。如图 33-9 所示，下一个凸包顶点 p_1 具有相对于 p_0 的最小极角。(如果有数个这样的点，就选取距离 p_0 最远的点作为 p_1。)类似地，p_2 具有相对于 p_1 的最小极角，等等。当到达最高顶点，如 p_k(如果有数个这样的点，则选取距离最远的点)时，我们已经构造好了 $CH(Q)$的右链了，如图 33-9 所示。为了构造其左链，从 p_k 开始选取相对于 p_k 具有最小极角的点作为 p_{k+1}，但这时的 x 轴是原 x 轴的反方向。如此继续下去，根据负 x 轴的极角逐渐形成左链，直至回到初始顶点 p_0。

可以用围绕凸包的一次概念性扫除来实现 Jarvis 步进法，即无需分别构造左链和右链。在这样一种典型的实现方法中，要随时记录上一次选取的凸包的边的角度，并要求凸包边的角度序列严格递增(在 0 到 2π 弧度范围内)。分别构造左右链的优点是无需显式地计算角度，33.1 节中

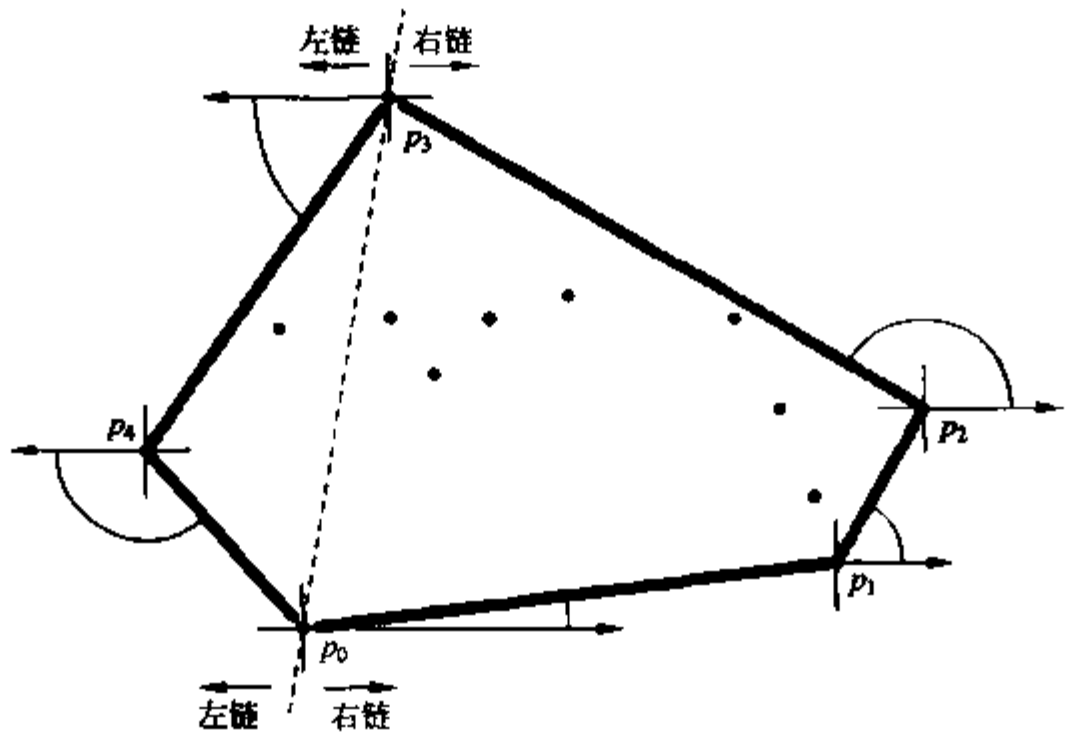


图 33-9　Jarvis 步进法的操作。选出的第一个顶点是最低的点 p_0。下一个顶点 p_1 与其他点相比，有着相对于 p_0 的最小极角。接着，p_2 有着相对于 p_1 的最小极角。右链最高达到最高点 p_3。接着，通过找相对负 x 轴的最小极角将左链构造出来

介绍的技术就足以用来对角度进行比较了。

如果有适当的实现方法，Jarvis 步进法的运行时间就会是 $O(nh)$。对 $CH(Q)$ 的 h 个顶点中的每一个顶点，都找出具有最小极角的顶点。如果采用 33.1 节中讨论过的技术，则每次极角比较操作所需的时间为 $O(1)$。正如 9.1 节中说明的那样，如果每次比较操作所需时间为 $O(1)$，则可以在 $O(n)$时间内计算出 n 个值中的最小值。因此，Jarvis 步进法的运行时间为 $O(nh)$。

练习

33.3-1　证明：在过程 GRAHAM-SCAN 中，点 p_1 和 p_m 必定是 $CH(Q)$的顶点。

33.3-2　考虑一个能支持加法、比较和乘法的计算模型，用该模型对 n 个数进行排序时，存在一个下界 $\Omega(n \lg n)$。证明：当在这样一个模型中有序地计算出 n 个点组成的集合的凸包时，其下界为 $\Omega(n \lg n)$。

955
~
956

33.3-3　已知一组点的集合 Q，证明彼此间距离最远的点对必定是 $CH(Q)$中的顶点。

33.3-4　对给定的一个多边形 P 和在其边界上的一个点 q，q 的阴影是满足线段 $\overline{qr}$ 完全在 P 的边界上或内部的点 r 的集合。如果在 P 的内部存在一个点 p，它处于 P 的边界上每个点的阴影中，则多边形 P 是星形多边形。所有满足这种条件的点 p 的集合称为 P 的内核(参见图 33-10。)给定一个 n 个顶点的星形多边形 P 按逆时针方向排序的各个顶点，试说明如何在 $O(n)$的时间内计算出 $CH(Q)$。

33.3-5　在联机凸包问题(on-line convex-hull problem)中，每次只给出 n 个点组成的集合 Q 中的一个点。在接收到每个点后，就计算出目前所见到的点的凸包。显然，可以对每个点运行一次 Graham 扫描算法，总的运行时间为 $O(n^2 \lg n)$。试说明如何在 $O(n^2)$时间内解决联机凸包问题。

***33.3-6**　试说明如何实现增量方法，使其在 $O(n \lg n)$的时间内，计算出 n 个点的凸包。

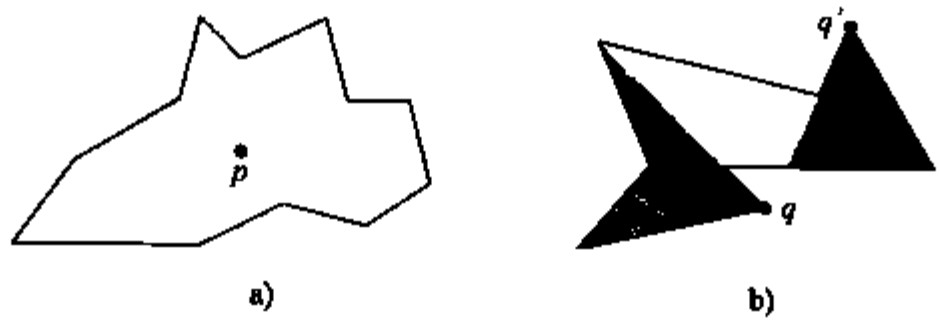


图 33-10　练习 33.3-4 中用到的星形多边形的定义。a)一个星形多边形。从 p 至边界上任何点 q 的线段仅在 q 处与边界相交。b)一个非星形多边形。左边阴影区域为 q 的阴影，而右边的阴影区域为 q' 的阴影。由于这些区域是不相交的，故内核为空

33.4　寻找最近点对

现在来考虑一下在 $n \geqslant 2$ 个点的集合 Q 中寻找最近点对的问题。“最近”是指通常意义下的欧几里得距离：即，点 $p_1=(x_1, y_1)$ 和 $p_2=(x_2, y_2)$ 之间的距离为 $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$。集合 Q 中的两个点可能重合，在这种情况下，它们之间的距离为 0。这一问题可以应用于交通控制等系统中。在空中或海洋交通控制系统中，需要发现两个距离最近的交通工具，以便检测出可能发生的相撞事故。

957

在最简单的蛮力搜索最近点对的算法中，要查看所有 $\binom{n}{2}=\Theta(n^2)$ 个点对。在本节中，将介绍一种解决该问题的分治算法，其运行时间可以用大家熟悉的递归式 $T(n)=2T(n/2)+O(n)$ 来描述。因此，该算法的运行时间仅为 $O(n \lg n)$。

分治算法

算法的每一次递归调用的输入为子集 $P \subseteq Q$ 和数组 X 和 Y，每个数组均包含输入子集 P 的所有点。对数组 X 中的点，按其 x 坐标单调递增的顺序进行排序。类似地，对数组 Y 中的点按其 y 坐标单调递增的顺序进行排序。注意，为了获得 $O(n \lg n)$ 的时间界，不能在每次递归调用中都进行排序。如果每次递归调用都进行排序的话，运行时间的递归式就变为 $T(n)=2T(n/2)+O(n \lg n)$，其解为 $T(n)=O(n \lg^2 n)$。(利用练习 4.4-2 中给出的主方法。)我们稍后将会看到如何运用“预排序”来保持这种排序性质，无需在每次递归调用中都进行排序。

输入为 P、X 和 Y 的递归调用首先检查是否 $|P| \leqslant 3$。如果是，则仅执行上述的简单方法：对所有 $\binom{|P|}{2}$ 点对进行检查，并返回最近点对。如果 $|P|>3$，则递归调用执行如下分治法模式：

分解：找出一条垂直线 l，它把点集 P 划分为满足下列条件的两个集合 P_L 和 P_R：$|P_L|=\lceil |P|/2 \rceil$，$|P_R|=\lfloor |P|/2 \rfloor$，$P_L$ 中的所有点在线 l 上或在 l 的左侧，P_R 中的所有点在线 l 上或在 l 的右侧。数组 X 被划分为两个数组 X_L 和 X_R，分别包含 P_L 和 P_R 中的点，并按 x 坐标单调递增的顺序进行排序。类似地，数组 Y 被划分为两个数组 Y_L 和 Y_R，分别包含 P_L 和 P_R 中的点，并按 y 坐标单调递增的顺序进行排序。

解决：把 P 划分为 P_L 和 P_R 后，再进行两次递归调用，一次找出 P_L 中的最近点对，另一次找出 P_R 中的最近点对。第一次调用的输入为子集 P_L，数组 X_L 和 Y_L；第二次调用的输入为子集 P_R、X_R 和 Y_R。设对 P_L 和 P_R，返回的最近点对的距离分别为 δ_L 和 δ_R，则置 $\delta=\min(\delta_L, \delta_R)$。

合并：最近点对要么是某次递归调用找出的距离为 δ 的点对，要么是 P_L 中的一个点与 P_R 中的一个点组成的点对，算法确定是否存在其距离小于 δ 的一个点对。注意，如果存在这样的一个点对，则点对中的两个点必定都在距离直线 l 的 δ 单位之内。因此，如图 33-11a 所示，它们必定都处于以直线 l 为中心、宽度为 2δ 的垂直带形的区域内。为了找出这样的点对(如果存在的
958 话)，算法要做如下工作：

1)建立一个数组 Y'，它是把数组 Y 中所有不在宽度为 2δ 的垂直带形区域内的点去掉后所得的数组。数组 Y' 与 Y 一样，是按 y 坐标顺序排序的。

2)对数组 Y' 中的每个点 p，算法试图找出 Y' 中距离 p 在 δ 单位以内的点。下面将会看到，在 Y' 中仅需考虑紧随 p 后的 7 个点。算法计算出从 p 到这 7 个点的距离，并记录下 Y' 的所有点对中，最近点对的距离 δ'。

3)如果 $\delta'<\delta$，则垂直带形区域内，的确包含比根据递归调用所找出的最近距离更近的点对，于是返回该点对及其距离 δ'。否则，就返回递归调用中发现的最近点对及其距离 δ。

上述描述中，省略了一些对获得 $O(n\lg n)$ 运行时间非常必要的实现细节。在证明算法的正确性以后，将说明如何实现算法，才能获得要求的运行时间范围。

正确性

除以下两方面外，这种最近点对算法的正确性是显而易见的。第一，当 $|P|\leqslant 3$ 时，递归调用过程到底，就能保证不会解仅含一个点的子问题进行划分。第二，仅需检查数组 Y' 中紧随每个点 p 后的 7 个点。现在就来证明这条性质。

假定在某一级递归调用中，最近点对为 $p_L\in P_L$，$p_R\in P_R$，则 p_L 和 p_R 间的距离 δ' 严格小于 δ。点 p_L 必定在直线 l 上，或者在 l 左边 δ 单位以内。类似地，p_R 必定在直线 l 上，或者在 l 右边 δ 单位以内。此外，p_L 和 p_R 的垂直距离也小于 δ 单位。因此，如图 33-11a 所示，p_L 和 p_R 在以直线 l 为中心线的 $\delta\times 2\delta$ 矩形区域内。(在该矩形内也可能有其他点。)

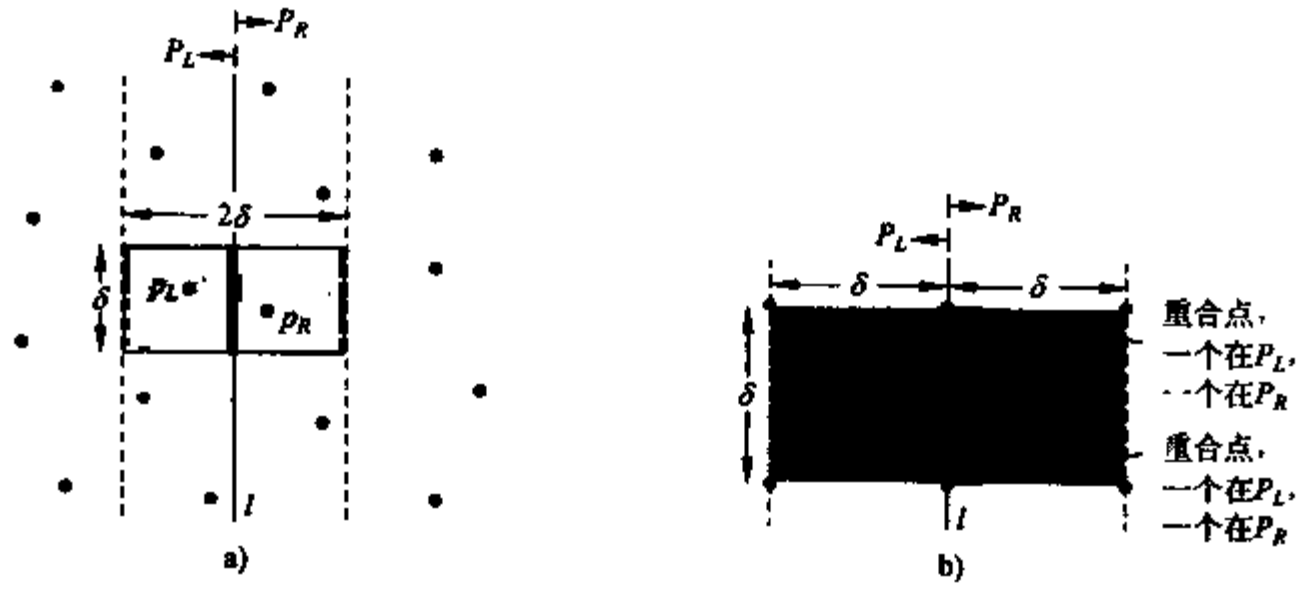


图 33-11　证明最近顶点对算法需要检查数组 Y' 中每个点后面的 7 个点时涉及的一些关键概念。a)如果 $p_L\in P_L$，$p_R\in P_R$，且 p_L 和 p_R 间的距离小于 δ，则它们必定位于一个以直线 l 为中心线的 $\delta\times 2\delta$ 矩形区域内。b)4 个两两之间的距离至少为 δ 的点是如何位于同一个 $\delta\times\delta$ 正方形内的。左边为 P_L 中的 4 个点，右边为 P_R 中的 4 个点。在 $\delta\times 2\delta$ 的矩形区域内，如果直线 l 上示出的点实际上都是重合的点对，其中一个点在 P_L 中，另一个点在 P_R 中的话，可能就会有 8 个点

下面来证明 P 中至多有 8 个点可能处于该 $\delta\times 2\delta$ 矩形区域内。考察该矩形左半边的 $\delta\times\delta$ 正方形。因为 P_L 中的所有点之间的距离至少为 δ 单位，所以至多有 4 个点可能位于该正方形内，

图 33-11b 说明了原因。类似地，P_R 中至多有 4 个点可能位于该矩形右半边的 $\delta\times\delta$ 正方形内。因此，P 中至多有 8 个点可能位于该 $\delta\times 2\delta$ 矩形内。(注意，由于直线上的点可能属于 P_L，也可能属于 P_R，所以直线 l 上最多可以有 4 个点。如果有两对重合的点，每对包含一个 P_L 中的点和一个 P_R 中的点，一对在直线 l 与矩形上面一条边的交点处，另一对在直线 l 与矩形下面一条边的交点处，就会达到上述限制。)

在说明了 P 中至多有 8 个点可能位于该矩形中后，就很容易看出仅需检查数组 Y' 中每个点之后的 7 个点。仍假设最近的点对为 p_L 和 p_R，并(不失一般性)假设在数组 Y' 中，p_L 位于 p_R 之前。那么，即使 p_L 在 Y' 中尽可能早出现而 p_R 尽可能晚出现，p_R 也一定是跟随 p_L 的 7 个位置中的一个。因此，就证明了最近点对算法是正确的。

算法的实现与运行时间

前面说过，我们的目标是取得关于运行时间的递归式 $T(n)=2T(n/2)+O(n)$，其中 $T(n)$ 是在 n 个点的集合上算法的运行时间。主要困难在于保证传递给递归调用的数组 X_L、X_R、Y_L 和 Y_R 能按适当的坐标进行排序，并且能使 Y' 按 y 坐标进行排序。(注意，如果某次递归调用接收到的数组 X 已经是排好序的，则很容易在线性时间内，完成把 P 划分为 P_L 和 P_R 的操作。)

959
~
960

在每次调用中，我们希望形成一个已排序数组的有序子集。例如，给某个特定调用的输入为子集 P 和按 y 坐标排序的数组 Y。把 P 划分为 P_L 和 P_R 后，需要形成按 y 坐标排序的数组 Y_L 和 Y_R。这些数组必须在线性时间内形成。我们所用的方法可以看作与 2.3.1 节中介绍的合并排序过程 MERGE 相反：把一个已排序数组分成两个有序数组。下面的伪代码给出了这种思想的实现。

```
1  length[Y_L] ← length[Y_R] ← 0
2  for i ← 1 to length[Y]
3      do if Y[i] ∈ P_L
4          then length[Y_L] ← length[Y_L]+1
5               Y_L[length[Y_L]] ← Y[i]
6          else length[Y_R] ← length[Y_R]+1
7               Y_R[length[Y_R]] ← Y[i]
```

我们仅仅是按次序检查数组 Y 中的点。如果一个点 $Y[i]$ 在 P_L 中，则把它添加到数组 Y_L 的末端；否则，就把它添加到数组 Y_R 的末端。用类似的伪代码也可以形成数组 X_L、X_R 和 Y'。

剩下的问题只是首先，如何对点进行排序。仅需对其进行预排序，即在第一次递归调用前，对所有的点进行排序。这些排序数组被传递到第一次递归调用中，在那里，根据需要在通过递归调用时再对其进行削减。预排序使运行时间增加了 $O(n\lg n)$，但这样一来，除递归调用外，递归过程的每一步仅需线性时间。如果设 $T(n)$ 为每个递归步的运行时间，$T'(n)$ 为整个算法的运行时间，我们就得到了 $T'(n)=T(n)+O(n\lg n)$，以及

$$T(n)=\begin{cases}2T(n/2)+O(n) & \text{如果 } n>3\\ O(1) & \text{如果 } n\leqslant 3\end{cases}$$

因此，$T(n)=O(n\lg n)$，$T'(n)=O(n\lg n)$。

练习

33.4-1　Smothers 教授提出了一个方案，即在最近点对算法中，只检查数组 Y' 中每个点后面的 5 个点，其思想是总是把直线 l 上的点放入集合 P_L 中。那么，直线 l 上就不可能有一个点属于 P_L，另一个点属于 P_R 的重合点对。因此，至多可能有 6 个点处于 $\delta\times 2\delta$ 的矩形内。这种方案的缺陷何在？

33.4-2 在不增加算法渐近运行时间的前提下，试说明如何保证传递给第一次递归调用的点集中不包含重合的点。证明这样一来，只需要检查数组 Y' 中跟随每个点后的 5 个数组位置就足够了？

961

33.4-3 两个点之间的距离除欧几里得距离外，还有其他定义方法。在平面上，点 p_1 和 p_2 之间的 L_m 距离由下式给出：$(|x_1-x_2|^m+|y_1-y_2|^m)^{1/m}$。因此，欧几里得距离实际上是 L_2 距离。修改最近点对算法，使其利用 L_1 距离，也称为曼哈顿距离。

33.4-4 已知平面上的两个点 p_1 和 p_2，它们之间的 L_∞ 距离为 $\max(|x_1-x_2|, |y_1-y_2|)$。修改最近点对算法，使其能利用 L_∞ 距离。

33.4-5 对最近点对算法进行修改，使其能避免对数组 Y 进行预排序，但仍然能使算法的运行时间保持为 $O(n\lg n)$。（提示：将已排序的数组 Y_L 和 Y_R 加以合并，以形成有序的数组 Y。）

思考题

33-1　凸层

已知平面上的点集 Q，我们用归纳法来定义 Q 的凸层(convex layer)。Q 的第一凸层是由 Q 中是 $CH(Q)$ 顶点的那些点组成。对 $i>1$，定义 Q_i 由把 Q 中所有在凸层 1，2，…，$i-1$ 中的点去除后剩余的点所构成。如果 $Q_i\neq\phi$，那么 Q 的第 i 凸层为 $CH(Q_i)$；否则，第 i 凸层无定义。

a)写出一个运行时间为 $O(n^2)$ 的算法，以找出 n 个点所组成的集合的各凸层。

b)证明：在对 n 个实数进行排序所需时间为 $\Omega(n\lg n)$ 的任何计算模型上，要计算出 n 个点的凸层需要 $\Omega(n\lg n)$ 时间。

33-2　最大层

设 Q 是平面上 n 个点所组成的集合。如果有 $x\geqslant x'$ 且 $y\geqslant y'$，则称点 (x, y) 支配点 (x', y')。Q 中不被其中任何其他点支配的点称为最大点。注意，Q 可以包含许多最大点，可以把这些最大点组织成如下的最大层。第一最大层 L_1 是 Q 中最大点构成的集合。对 $i>1$，第 i 最大层 L_i 是 $Q-\bigcup_{j=1}^{i-1}L_j$ 中的最大点构成的集合。

962

假设 Q 包含 k 个非空的最大层，并设 y_i 是 L_i 中最左边的点的 y 坐标($i=1, 2, \cdots, k$)。假定 Q 中没有两个点有相同的 x 坐标或 y 坐标。

a)证明 $y_1>y_2>\cdots>y_k$。

考察一个点 (x, y)，它在 Q 中任何点的左边，并且其 y 坐标与 Q 中任何点的 y 坐标都不相同。设 $Q'=Q\cup\{(x, y)\}$。

b)设 j 是满足 $y_j<y$ 的最小下标，除非 $y<y_k$，在这种情况下，设 $j=k+1$。证明 Q' 的最大层如下：

- 如果 $j\leqslant k$，则 Q' 的最大层与 Q 的最大层相同，只是 L_j 也包含 (x, y) 作为其新的最左点。
- 如果 $j=k+1$，则 Q' 的前 k 个最大层与 Q 的相同，但此外，Q' 有一个非空的第 $k+1$ 最大层：$L_{k+1}=\{(x, y)\}$。

c)描述一种时间为 $O(n\lg n)$ 的算法，以便计算出 n 个点的集合 Q 的各最大层。（提示：把一条扫除线从右向左移动。）

d)如果允许输入点有相同的 x 坐标或 y 坐标，会不会出现问题？如果会，提出一种方

法来解决这一问题。

33-3　魑魅和鬼问题

有 n 个巨人正与 n 个鬼战斗。每个巨人的武器是一个质子包，它可以用一串质子流射中鬼而把鬼消灭。质子流沿直线行进，在击中鬼时就终止。巨人决定采取下列策略。他们各自寻找一个鬼形成 n 个巨人-鬼对，然后每个巨人同时向各自选取的鬼射出一串质子流。我们知道，质子流互相交叉是很危险的，因此，巨人选择的配对方式应该使质子流都不会交叉。

假定每个巨人和每个鬼的位置都是平面上一个固定的点，并且没有三个位置共线。

a)论证存在一条通过一个巨人和一个鬼的直线，使得直线一边的巨人数与同一边的鬼数相等。试说明如何在 $O(n \lg n)$ 时间内找出这样一条直线。

b)写出一个运行时间为 $O(n^2 \lg n)$ 的算法，使其按不会有质子流交叉的条件把巨人与鬼配对。

33-4　拾取棍子问题

Charon 教授有 n 根小棍子，它们以某种方式，互相叠放在一起。每根棍子都用其端点来指定，每个端点都是一个有序的三元组，给出了其 (x, y, z) 坐标。所有棍子都不是垂直的。他希望拾取所有的棍子，但要满足条件，一次一根地当一根棍子上面没有压着其他棍子时，他才可以挑起该棍子。 963

a)给出一个过程，取两根棍子 a 和 b 作为参数，报告 a 是在 b 的上面、下面还是与 b 无关。

b)给出一个有效的算法，它应能确定是否有可能拾取所有的棍子。如果能，提供一个拾取所有棍子的合法顺序。

33-5　稀疏包分布

考虑计算平面上点的集合的凸包问题，但这些点是根据某已知的随机分布取得的。有时，从这样一种分布中取得的 n 个点的凸包的期望规模为 $O(n^{1-\varepsilon})$，其中 ε 为大于 0 的某个常数。称这样的分布为*稀疏包分布*。稀疏包分布包括以下几种：

- 点是均匀地从一个单位半径的圆面中取得的，凸包的期望规模为 $\Theta(n^{1/3})$。
- 点是均匀地从一个具有 k 条边的凸多边形内部取得的(k 为任意常数)。凸包的期望规模为 $\Theta(\lg n)$。
- 点是根据二维正态分布取得的。凸包的期望规模为 $\Theta(\sqrt{\lg n})$。

a)已知两个分别有 n_1 和 n_2 个顶点的凸多边形，说明如何在 $O(n_1 + n_2)$ 时间内，计算出全部 $n_1 + n_2$ 个点的凸包(多边形可以重叠)。

b)证明：对于根据稀疏包分布独立取得的一组 n 个点，其凸包可以在 $O(n)$ 的期望时间内计算出来。(*提示*：采用递归方法分别求出前 $n/2$ 个点和后 $n/2$ 个点的凸包，然后再对结果进行合并。)

本章注记

本章只是刚刚撩开了计算几何学算法和技术这一"神秘面纱"的一角。有关计算几何学的参考书很多，如 Preparata 和 Shamos[247]，Edelsbrunner[83]，以及 O'Rourke[235]。

尽管几何学从古代就有人研究了，但用于解决几何问题的算法方面的发展相对来说却是比较新的。Preparata 和 Shamos 指出，最早的用于描述问题复杂性的记号表示是由 E. Lemoine 于 1902 提出的。当时，他正致力于研究欧几里得构造，即用指南针和尺子所做的构造，并设计出 964

了5条原语：将指南针的一个指针放在某一给定点上，将指南针的一个指针放在某一给定的线上，画一个圆，使尺子的边通过某一给定的点，画一条直线。Lemoine对激活某一构造所需的原语数目比较感兴趣；他称这一数目为该构造的“简单性”。

33.2节中用于确定任何线段是否相交的算法是由Shamos和Hoey[275]提出的。

Graham扫描算法的原始版本是由Graham[130]给出的。打包算法是由Jarvis[165]提出的。Yao[318]利用决策树计算模型，证明了凸包算法运行时间的一个下界$\Omega(n\lg n)$。当将凸包中顶点数目h考虑在内时，Kirkpatrick和Seidel[180]的剪枝-搜索算法是渐近最优的，其运行时间为$O(n\lg h)$。

用于寻找最近点对的、运行时间为$O(n\lg n)$的分治算法是由Shamos提出的，并出现于Preparata和Shamos[247]中。Preparata和Shamos还证明了在决策树模型下，该算法是渐近最
965 优的。

第 34 章　NP 完全性

迄今为止，几乎我们学习过的所有算法都是多项式时间的算法：对规模为 n 的输入，它们在最坏情况下的运行时间为 $O(n^k)$，其中 k 为某个常数。人们很自然地会想到这样一个问题：是否所有的问题都能在多项式时间内解决？答案是否定的。例如，存在着一些问题，如图灵著名的"停机问题"，任何计算机不论耗费多少时间也不能解决。还有一些问题是可以解决的，但对任意常数 k，它们都不能在 $O(n^k)$ 的时间内得到解答。一般来说，我们把可以由多项式时间的算法解决的问题看作是易处理的问题，而把需要超多项式时间才能解决的问题看作是难处理的问题。

本章的主题是一类称为"NP 完全"(NP-complete)的有趣问题，它的状态是未知的。迄今为止，既没有人找出求解 NP 完全问题的多项式时间的算法，也没有人能够证明对这类问题不存在多项式时间算法。这一所谓的 P≠NP 问题自 1971 年被提出以后，已经成为了理论计算机科学研究领域中，最深奥和最错综复杂的开放问题之一了。

NP 完全问题有一个方面特别诱惑人，就是在这一类问题中，有几个问题在表面上看来与有着多项式时间算法的问题非常相似。在下面列出的每一对问题中，一个是可以在多项式时间内求解的，另一个是 NP 完全的，但从表面上来看，两个问题之间的差别很小：

最短与最长简单路径：在第 24 章中，我们看到了即使是在边有负的权值的情况下，也能在一个有向图 $G=(V, E)$ 中，在 $O(VE)$ 的时间内，从一个源顶点开始找出最短的路径。然而，寻找两个顶点间最长简单路径问题是 NP 完全的。事实上，即使所有边的权值都是 1，它也是 NP 完全的。

欧拉游程与哈密顿回路：对一个连通的有向图 $G=(V, E)$ 的欧拉游程是一个回路，它遍历图 G 中每条边一次，但可能不止一次地访问同一个顶点。根据思考题 22-3，可以在 $O(E)$ 时间内，确定一个图中是否存在着一个欧拉游程，并且，事实上，能够在 $O(E)$ 时间内找出这一欧拉游程中的各条边。一个有向图 $G=(V, E)$ 的哈密顿回路是一种简单回路，它包含 V 中的每个顶点。确定一个有向图中是否存在哈密顿回路的问题是 NP 完全的。(在本章的后面，我们还要证明，确定一个无向图中是否存在哈密顿回路的问题也是 NP 完全的。)

2-CNF 可满足性与 3-CNF 可满足性：在一个布尔公式中，可以包含这样一些成分：布尔变量，其取值可以是 0 或者 1；布尔连接词如 ∧(AND)、∨(OR)以及¬(NOT)；括号。对一个布尔公式来说，如果存在着对其变量的某种 0 和 1 赋值，使得它的值为 1 的话，则称它是可满足的。在本章的稍后部分，还将更为形式地定义有关的术语。但是，非形式地，称一个布尔公式为 k 合取范式(conjunctive normal form)或 k-CNF，如果它是用 AND 连接若干个 OR 子句，且每个子句中恰有 k 个布尔变量或其否定形式的话。例如，布尔公式 $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ 就是一个 2-CNF。(它有一个可满足性赋值 $x_1=1$，$x_2=0$，$x_3=1$。)存在着一个多项式时间的算法来确定一个 2-CNF 公式是否是可满足的，但我们在本章稍后就会看到，要确定一个 3-CNF 公式是否是可满足的这一问题是 NP 完全的。

NP 完全性与 P 类和 NP 类

在本章中，我们将谈及三类问题：P，NP 和 NPC，其中最后一类是指 NP 完全问题。此处，我们只是非形式地对它们进行描述，后面将给出更为形式的定义。

P 类中包含的是在多项式时间内可解的问题。更具体一点，它们都是一些可以在 $O(n^k)$ 时间

内求解的问题，此处 k 为某个常数，n 是问题的输入规模。在本书前面几章分析过的问题中，绝大部分都属于 P 类。

NP 类中包含的是在多项式时间内"可验证"的问题，此处是指某一解决方案的"证书"，就能够在问题输入的规模的多项式时间内，验证该证书是正确的。例如，在哈密顿回路问题中，给定一个有向图 $G=(V, E)$，证书可以是一个包含 $|V|$ 个顶点的序列 $\langle v_1, v_2, v_3, \cdots, v_{|V|}\rangle$。很容易在多项式时间内判断出对 $i=1, 2, 3, \cdots, |V|-1$，是否有 $(v_i, v_{i+1})\in E$ 和 $(v_{|V|}, v_1)\in E$。再看另一个例子，对于 3-CNF 可满足性问题，一个证书可以是对一组变量的一个赋值。我们可以很容易地在多项式时间内，检查这一赋值是否满足布尔表达式。

P 中的任何问题也都属于 NP，这是因为如果某一问题是属于 P 的，则可以在不给出证书的情况下，在多项式时间内解决它。在本章的后面，将把这一概念形式化，但现在我们可以暂且相
967 信有 P⊆NP。至于 P 是否是 NP 的一个真子集，在目前是一个开放的问题。

从非形式的意义上来说，如果一个问题属于 NP，且与 NP 中的任何问题是一样"难的"(hard)，则说它属于 NPC 类，也称它为 NP 完全的(NP-complete)。在本章中后面，我们将形式化地定义什么叫做与 NP 中的任何问题是一样难的。同时，我们还将不加证明地宣称如果任何 NP 完全的问题可以在多项式时间内解决，则每一个 NP 完全的问题都有一个多项式时间的算法。多数搞理论研究的计算机科学家认为，NP 完全问题是难处理的，因为迄今为止研究过的 NP 完全问题非常之多，但还没有任何人发现其中任何一个问题的多项式时间解决方案。因此，如果这些问题都能在多项式时间内得到解决的话，那将是非常惊人的。然而，尽管迄今为止，人们付出了大量的努力来证明 NP 完全问题是不易解决的，但仍然没有任何最终结果，因此，我们也不能排除 NP 完全问题事实上可以在多项式时间内求解的可能性。

要成为一名优秀的算法设计者，就一定要懂得关于 NP 完全问题的基本原理。如果读者能确定一个 NP 完全问题，就可以提供充分的论据说明其难处理性。作为一名工程师，更好的做法是花时间开发一种近似算法(见第 35 章)或解决某种易处理的问题特例，而不是寻找求得问题确切解的一种快速算法。此外，从表面上看，很多自然而有趣的问题并不比排序、图的搜索或网络流问题更困难，但事实上，它们却是 NP 完全问题。因此，熟悉这类问题是非常重要的。

证明 NP 完全问题概述

在证明某一特定问题是 NP 完全问题时，我们所采用的技术与本书中大部分内容里设计和分析算法时用到的技术都有所不同。之所以会有这样的差别，有一个根本的原因：在证明一个问题是 NP 完全问题时，我们是在陈述它是一个多么困难的问题(或至少我们认为它有多难)。我们并不是要证明存在某个有效的算法，而是要证明不太可能存在一个有效的算法。从这一角度来看，NP 完全性证明有点类似于 8.1 节中，对任何比较排序算法的运行时间下界 $\Omega(n\lg n)$ 的证明；但是，用于证明 NP 完全性所用到的特殊技巧与 8.1 节中所用到的决策树方法是不同的。

在证明一个问题为 NP 完全问题时，要依赖于三个关键概念：

判定问题与最优化问题

很多有趣的问题都是最优化问题(optimization problem)，其中每一种可能的解(亦即，"合法的")解都有一个相关的值，我们的目标是找出一个具有最佳值的可行解。例如，在一个称为 SHORTEST-PATH 的问题中，已知的是一个无向图 G 及顶点 u 和 v，要找出 u 到 v 之间的经过
968 最少边的路径。(换句话说，SHORTEST-PATH 是在一个无权、无向图中的单点对间最短路径问题。)然而，NP 完全性不直接适合于最优化问题，但适合于判定问题(decision problem)，因为这种问题的答案是简单的"是"或"否"(或者，更为形式地，答案是"1"或"0")。

尽管证明一个问题是 NP 完全问题会将我们的目光局限于判定问题，但在最优化问题与判定问题之间有着很方便的关系。通常，通过对待优化的值强加一个界，就可以将一个给定的最优化问题转化为一个相关的判定问题了。例如，对 SHORTEST-PATH 问题来说，它有一个相关的判定问题(我们称其为 PATH)，就是要判定给定的有向图 G、顶点 u 和 v、一个整数 k，在 u 和 v 之间是否存在一条包含至多 k 条边的路径。

当我们试图证明最优化问题是一个“困难的”问题时，就可以利用该问题与相关的判定问题之间的关系。这是因为，从某种意义上来说，判定问题要“更容易一些”，或至少“不会更难”。例如，我们可以先解决 SHORTEST-PATH 问题，再将找出的最短路径上边的数目与相关判定问题中参数 k 进行比较，从而可以解决 PATH 问题。换句话说，如果某个最优化问题比较容易的话，那么其相关的判定问题也会是比较容易的。按照与 NP 完全性更为有关的方式来说，就是如果我们能够提供证据表明某个判定问题是个困难问题的话，就等于提供了证据表明其相关的最优化问题也是困难的。因此，即使 NP 完全性理论限制了人们对判定问题的关注，但它对最优化问题常常还是有一定意义的。

归约

上述有关证明一个问题不难于或不简单于另一个问题的说法，对两个问题都是判定问题也是适用的。在几乎每一个 NP 完全性证明中，我们都利用了这一思想，做法如下。我们来考虑一个判定问题(称为 A)，希望在多项式时间内解决该问题。称某一特定问题的输入为该问题的一个实例(instance)，例如，PATH 问题的一个实例可以是某一特定的图 G、G 中特定的点 u 和 v 和一个特定的整数 k。现在，假设有另一个不同的判定问题(称为 B)，我们知道如何在多项式时间内来解决它。最后，假设有一个过程，它能将 A 的任何实例 α 转换成 B 的、具有以下特征的某个实例 β：

1)转换操作需要多项式时间；

2)两个实例的答案是相同的。亦即，α 的答案是“是”，当且仅当 β 的答案也是“是”。 969

称这样的一种过程为多项式时间的归约算法(reduction algorithm)，并且，如图 34-1 中所示，它提供了一种在多项式时间内解决问题 A 的方法：

图 34-1　对于一个判定问题 A，在给定另一个问题 B 的多项式时间判定算法后，利用一个多项式时间的归约算法，即可在多项式时间内解决问题 A。在多项式时间内，将 A 的一个实例 α 转换成 B 的一个实例 β，在多项式时间内解决 B，再将 β 的答案用作 α 的答案即可

1)给定问题 A 的一个实例 α，利用多项式时间归约算法，将它转换为问题 B 的一个实例 β。

2)在实例 β 上，运行 B 的多项式时间判定算法。

3)将 β 的答案用作 α 的答案。

只要上述步骤中的每一步只需多项式时间，则所有三步合起来也只需要多项式时间，这样，我们就有了一种对 α 进行判断的方法了。换句话说，通过将对问题 A 的求解“归约”为对问题 B 的求解，就可以利用 B 的“易求解性”来证明 A 的“易求解性”。

前面说过，NP 完全性是为了反映一个问题有多难，而不是为了反映它是多容易的，因此，我们以相反的方式来利用多项式时间归约，从而说明某一问题是 NP 完全的。可以将这一思想再

往前推进一步，并说明如何利用多项式时间的归约来表明对某一特定的问题 B 而言，不存在多项式时间的算法。假设有一个判定问题 A，我们已经知道它不可能存在多项式时间的算法。(此时可以先不必顾虑如何才能找到这样的一个问题 A。)进一步假设有一个多项式时间的归约，它将 A 的一个实例转换为 B 的实例。现在，可以利用反证法来简单地证明 B 不可能存在多项式时间的算法。先做一个相反的假设，即，假设 B 有一个多项式时间的算法。那么，利用图 34-1 中示出的方法，我们就应该有某种方法能在多项式时间内解决问题 A，而这是与 A 没有多项式时间算法这一假设矛盾的。

至于 NP 完全性，我们不能假设问题 A 绝对没有多项式时间的算法。然而，证明的方法是类
970 似的，即，假设问题 A 是 NP 完全的前提下，来证明问题 B 是 NP 完全的。

第一个 NP 完全问题

应用归约技术要有个前提，即已知一个 NP 完全的问题，这样才能证明另一个问题也是 NP 完全的。因此，需要找到"第一个"NP 完全问题。我们将使用的这第一个问题就是电路可满足性问题(circuit satisfiability problem)，在这个问题中，已知的是一个布尔组合电路，它由 AND、OR 和 NOT 门所组成，我们希望知道这个电路是否存在一组布尔输入，能够使它的输出为 1。34.3 节中要证明，这第一个问题是 NP 完全的。

本章内容概要

本章主要研究 NP 完全性对算法分析有着最直接影响的几个方面。在 34.1 节中，要对"问题"这一概念作形式化定义，还要定义复杂类 P，它包含了在多项式时间内可解的判定问题。同时，还要看看这些概念是如何对应到形式语言的理论结构中去的。34.2 节中定义了关于判定问题的 NP 类，这些问题的解可以在多项式的时间内进行验证。在该节中，还要正式地提出 P≠NP 问题。

34.3 节主要讨论如何通过多项式时间的"归约"来研究问题之间的关系。它定义了 NP 完全性，并概述了一个称为"电路可满足性"的问题是 NP 完全问题的证明过程。在找出一个 NP 完全问题之后，在 34.4 节中要讨论如何利用归约方法，更简便地证明其他一些问题也是 NP 完全问题。通过证明两个公式可满足性问题是 NP 完全问题，对归约方法加以说明。34.5 节中给出了其他一些 NP 完全问题。

34.1　多项式时间

在开始研究 NP 完全性之前，先来形式化地定义一下多项式时间可解的问题。这些问题一般都被看作是易处理的，其原因是哲学方面的，而不是数学方面的。我们可以提供三点论据：

第一，虽然把所需运行时间为 $\Theta(n^{100})$ 的问题作为难处理问题也有其合理之处，但在实际中，需要如此高次的多项式时间的问题是非常少的。在实际中，所遇到的典型多项式时间可解问题所需的时间要少得多。经验表明，一旦某一问题的一个多项式时间算法被发现后，往往就会跟着发现一些更为有效的算法。即使对某个问题来说，当前最佳算法的运行时间为 $\Theta(n^{100})$，很有
971 可能在很短的时间内，就能找到一个运行时间要好得多的算法。

第二，对很多合理的计算模型来说，在一个模型上用多项式时间可解的问题，在另一个模型上也可以在多项式时间内获得解决。例如，用本书中大部分所使用的串行随机存取计算机在多项式时间内可求解的问题类，与抽象的图灵机上在多项式时间内可求解的问题类是相同的，[⊖]它也与利用并行计算机在多项式时间内可求解的问题类相同，即使处理器数目随输入规模以多项

⊖ 有关图灵机模型的详细讨论，可见 Hopcroft 和 Ullman[156]或 Lewis 和 Papadimitriou[204]。

式增加也是这样。

第三，多项式时间可解问题类具有很好的封闭性，这是因为在加法、乘法和组合运算下，多项式是封闭的。例如，如果一个多项式时间算法的输出馈送给另一个多项式时间算法作为输入，则得到的组合算法也是多项式时间的算法。如果另外一个多项式时间算法对一个多项式时间的子程序进行常数次调用，那么组合算法的运行时间也是多项式的。

抽象问题

为了理解多项式时间可解问题类，首先必须对"问题"这一概念进行形式化定义。定义抽象问题 Q 为在问题实例集合 I 和问题解法集合 S 上的一个二元关系。例如，SHORTEST-PATH 的一个实例是由一个图和两个顶点所组成的三元组，其解为图中的顶点序列，序列可能为空，表示两个顶点间不存在通路。问题 SHORTEST-PATH 本身就是一个关系，它把图的每个实例和两个顶点与图中联系这两个顶点的最短路径联系在了一起。因为最短路径不一定是唯一的，因此，一个给定的问题实例可能有多个解。

抽象问题的这一形式定义对我们的要求来说显得太笼统。为了简单起见，NP 完全性理论把注意力集中在判定问题(decision problem)上，即那些解为"是"或"否"的问题。于是，我们可以把抽象的判定问题看作是从实例集 I 映射到解集 $\{0, 1\}$ 上的一个函数。例如，与 SHORTEST-PATH 有关的判定问题是我们先前见到过的 PATH 问题，它是这样的：如果 $i=\langle G, u, v, k\rangle$ 是决策问题 PATH 的一个实例，那么，如果从 u 到 v 的最短路径的长度至多为 k 条边，则 $\text{PATH}(i)=1$(是)，否则 $\text{PATH}(i)=0$(否)。许多抽象问题并不是判定问题，而是最优化问题。在这些问题中，必须最大化或最小化某个量。然而，如我们在上面看到的，将最优化问题转化为一个并不更难的判定问题通常是比较简单的。

编码

如果要用一个计算机程序来求解一个抽象问题，就必须用一种程序能理解的方式来表示问题实例。抽象对象集合 S 的编码是从 S 到二进制串集合的映射 e。[⊖]例如，我们都熟悉把自然数 $\mathbf{N}=\{0, 1, 2, 3, 4, \cdots\}$ 编码为串 $\{0, 1, 10, 11, 100, \cdots\}$。在这种编码方案中，$e(17)=10001$。任何看过计算机键盘上字符的表示法的人都会熟知 ASCII 码或 EBCDIC 码。在 ASCII 码中，A 的编码为 1 000 001。即使是一个复合对象，也可以把其组成部分的表示进行组合，从而把它编码为一个二进制串。多边形、图、函数、有序对、程序等所有这些都可以编码为二进制串。

因此，"求解"某个抽象判定问题的计算机算法实际上是把一个问题实例的编码作为其输入。我们把实例集为二进制串的集合的问题称为具体问题。如果当提供给一个算法的是长度为 $n=|i|$ 的一个问题实例 i 时，算法可以在 $O(T(n))$ 时间内产生问题的解，我们就说该算法在时间 $O(T(n))$ 内解决了该具体问题。[⊜]因此，如果对某个常数 k，存在一个算法能在时间 $O(n^k)$ 内求解出某具体问题，就说该具体问题是多项式时间可解的。

现在，可以正式地定义复杂类 P 是在多项式时间内可解的具体判定问题的集合。

利用编码，可以将抽象问题映射到具体问题上。给定一个抽象判定问题 Q，其映射为实例集合 I 到 $\{0, 1\}$。利用一种编码 e：$I\to\{0, 1\}^*$，可以导出与该问题相关的具体判定问题，用 $e(Q)$ 来表示。[⊝]如果一个抽象问题实例 $i\in I$ 的解为 $Q(i)\in\{0, 1\}$，则该具体问题实例 $e(i)\in\{0, 1\}^*$

⊖ e 的陪域不一定是二进制串；定义在一个有限字母表(其中至少有两个符号)上的任何串集都是可以的。

⊜ 假定该算法的输出是独立于其输入的。因为需要至少一个时间步来产生输出的每一位，且共有 $O(T(n))$ 个时间步，故输出的规模为 $O(T(n))$。

⊝ 下面很快会介绍，$\{0, 1\}^*$ 表示所有由集合 $\{0, 1\}$ 中符号构成的串的集合。

的解也是 $Q(i)$。在技术上，可能存在一些表示无意义的抽象问题实例的二进制串，为了方便起见，假定任何这样的串都映射到 0。因此，对表示抽象问题实例的编码的二进制串实例，具体问题与抽象问题产生同样的解。

973 我们希望将编码作为桥梁，把多项式时间可解性的定义从具体问题扩展到抽象问题，但同时也希望这一定义与任何特定的编码无关。亦即，求解一个问题的效率不应依赖于问题的编码。不幸的是，实际上这种依赖性是相当严重的。例如，假定把一个整数 k 作为一个算法的唯一输入，并假设算法的运行时间为 $\Theta(k)$。如果提供的整数 k 是一元的(即由 k 个 1 组成的串)，那么，对长度为 n 的输入，该算法的运行时间为 $O(n)$，它是多项式时间。但是，如果采用更自然的二进制来表示整数 k，则输入长度为 $n=\lfloor \lg k \rfloor+1$。在这种情况下，该算法的运行时间为 $\Theta(k)=\Theta(2^n)$，它与输入规模成指数关系。因此，根据编码的不同，算法的运行时间可以是多项式时间或超多项式时间。

因此，对一个抽象问题如何编码，对理解多项式时间是相当重要的。如果不先指定编码，就不可能真正谈及对一个抽象问题的求解。然而，在实际应用中，如果不采用"代价高昂的"编码(如一元编码)，则问题的实际编码形式对问题是否能在多项式时间内求解的影响是微不足道的。例如，以 3 代替 2 为基数来表示整数，对问题是否能在多项式时间内求解没有任何影响，因为可以在多项式时间内，将以基数 3 表示的整数可以转换为以基数 2 表示的整数。

对一个函数 $f:\{0,1\}^* \to \{0,1\}^*$，如果存在一个多项式时间的算法 A，它对任意给定的输入 $x\in\{0,1\}^*$，都能产生输出 $f(x)$，则称该函数 f 是一个多项式时间可计算的(polynomial-time computable)函数。对某个问题实例集 I，如果存在两个多项式时间可计算的函数 f_{12} 和 f_{21}，满足对任意 $i\in I$，有 $f_{12}(e_1(i))=e_2(i)$，且 $f_{21}(e_2(i))=e_1(i)$，我们就说这两种编码 e_1 和 e_2 是多项式相关的。[⊖] 亦即，$e_2(i)$ 可以由一个多项式时间的算法根据编码 $e_1(i)$ 求出，反之亦然。如果某一抽象问题的两种编码 e_1 和 e_2 是多项式相关的，则如下面引理所述，该问题本身是否是多项式时间可解与选用这两种编码中的哪一种无关。

引理 34.1　设 Q 是定义在一个实例集 I 上的一个抽象判定问题，e_1 和 e_2 是 I 上多项式相关
974 的编码，则 $e_1(Q)\in P$ 当且仅当 $e_2(Q)\in P$。

证明：只需要证明一个方向(正向)，因为反向与正向是对称的。假定对某常数 k，$e_1(Q)$ 能在 $O(n^k)$ 的时间内求解。此外，再假定对任意问题实例 i 和某个常数 c，根据编码 $e_2(i)$，可以在时间 $O(n^c)$ 内计算出编码 $e_1(i)$，其中 $n=|e_2(i)|$。为了在输入 $e_2(i)$ 上求解问题 $e_2(Q)$，可以先计算出 $e_1(i)$，然后在输入 $e_1(i)$ 上运行关于 $e_1(Q)$ 的算法。这需要多长时间呢？代码变换所需的时间为 $O(n^c)$，因此 $|e_1(i)|=O(n^c)$，这是因为串行计算机的输出不可能比其运行时间更长。所以，求解关于 $e_1(i)$ 的问题所需的时间为 $O(|e_1(i)|^k)=O(n^{ck})$，因为 c 和 k 都是常数，所以这也是多项式时间。■

由此可见，对一个抽象问题的实例，采用二进制或三进制来进行编码，对其"复杂性"都没有影响，也就是说，对其是否为多项式时间可解是没有影响的。但是，如果对实例进行一元编码，则其复杂性可能会变化。为了能够用一种与编码无关的方式进行描述，一般都假定用合理的、简洁的方式对问题实例进行编码，除非我们特别另外指明。更准确地说，我们将假定一个整数的编码与其二进制表示是多项式相关的，并且一个有限集合的编码与其相应的括在括号中的、元素

⊖ 从技术上来说，我们还要求函数 f_{12} 和 f_{21}"将非实例映射至非实例"。某种编码 e 的非实例是指这样的一个串 $x\in\{0,1\}^*$，使得没有任何一个实例 i 能满足 $e(i)=x$。我们要求对编码 e_1 的每个非实例 x，都有 $f_{12}(x)=y$，其中 y 是 e_2 的某个非实例。并且，对 e_2 的每个非实例 x'，都有 $f_{21}(x')=y'$，其中 y' 是 e_1 的某个非实例。

间用逗号隔开的列表的编码是多项式相关的(ASCII 码就是这样的一种编码方案)。有了这样一种"标准的"编码，就可以合理地推导出其他数学对象如元组、图和和公式等的编码了。为了表示一个对象的标准编码，把该对象用尖括号括起来，如$\langle G\rangle$表示图 G 的标准编码。

只要隐式地使用与标准编码多项式相关的编码，就可以直接讨论抽象问题，而无需参照任何特定的编码，因为我们已经知道，选取哪一种编码对该问题是否多项式时间可解没有任何影响。今后，我们一般假设所有问题实例都是采用标准编码的二进制串，除非显式地指明其他情况。此外，我们也将忽略抽象问题与具体问题之间的差别。但是，读者也应该注意对实际中产生的某些问题，其标准编码并非是显而易见的，并且，选择编码方式对问题的求解会带来不同的影响。

形式语言体系

关注判定问题有一个方便之处，就是它们使得形式语言理论的使用变得比较容易了。先来回顾一下形式语言理论中的一些定义。字母表Σ是符号的有限集合。字母表Σ上的语言 L 是由Σ中符号组成的串的任意集合。例如，如果$\Sigma=\{0, 1\}$，集合 $L=\{10, 11, 101, 111, 1011, 1101, 10001, \cdots\}$是关于素数的二进制表示的语言。用 ϵ 表示空串，用 ϕ 表示空语言。Σ 上所有串构成的语言表示为 Σ^*。例如，如果 $\Sigma=\{0, 1\}$，则 $\Sigma^*=\{\epsilon, 0, 1, 00, 01, 10, 11, 000, \cdots\}$就是所有二进制串的集合。$\Sigma$上的每个语言 L 都是Σ^*的一个子集。 975

有多种运算都可以作用于语言。集合论中的运算(如并与交)，其直接来自于集合论定义。定义 L 的补为$\overline{L}=\Sigma^*-L$。两种语言 L_1 和 L_2 的并置是语言

$$L=\{x_1x_2 : x_1\in L_1 \text{ 且 } x_2\in L_2\}$$

语言 L 的闭包(Kleene 星)为语言

$$L^*=\{\epsilon\}\cup L\cup L^2\cup L^3\cup\cdots$$

其中 L^k 是 L 与其自身进行 k 次并置运算后得到的语言。

从语言理论的观点来看，任何判定问题 Q 的实例集即集合Σ^*，其中$\Sigma=\{0, 1\}$。因为 Q 完全是由那些答案为 1(是)的问题实例来描述的，因而可以把 Q 看作是定义在$\Sigma=\{0, 1\}$上的一个语言 L，其中

$$L=\{x\in\Sigma^* : Q(x)=1\}$$

例如，与判定问题 PATH 对应的语言为

PATH$=\{(G, u, v, k)$：$G=(V, E)$是一个无向图，$u, v\in V$，$k\geqslant 0$ 是一个整数，图 G 中从 u 到 v 存在一条长度至多为 k 的路径$\}$

(在方便的时候，有时用同一个名称(如上面的 PATH)来表示一个判定问题和与其相应的语言。)

形式语言体系可以用来表述判定问题与求解这些问题的算法之间的关系。如果对给定输入 x，算法输出 $A(x)=1$，我们就说算法 A 接受串 $x\in\{0, 1\}^*$。被算法 A 接受的语言是串的集合 $L=\{x\in\{0, 1\}^* : A(x)=1\}$，即为算法所接受的串的集合。如果 $A(x)=0$，则说算法 A 拒绝串 x。

即使语言 L 被算法 A 所接受，该算法也不一定会拒绝一个输入串 $x\notin L$。例如，算法可能会永远循环下去。如果 L 中的每个二进制串要么被算法 A 接受，要么被其拒绝，则说语言 L 由算法 A 判定。如果对任意长度为 n 的串 $x\in L$ 和某个常数 k，算法 A 在时间 $O(n^k)$ 内接受 x，则语言 L 在多项式时间内被算法 A 接受。如果对于任意长度为 n 的串 $x\in\{0, 1\}^*$ 和某个常数 k，算法 A 可以在时间 $O(n^k)$ 内正确地判定 $x\in L$，则说语言 L 在多项式时间内被算法 A 判定。因此， 976

要接受一个语言，只需关心 L 中的字符串，但是要判定某一语言，算法必须正确地接受或者拒绝 $\{0,1\}^*$ 中的每一个串。

例如，语言 PATH 就能够在多项式时间内被接受。一个多项式时间的接受算法要验证 G 是否编码一个无向图 G，u 和 v 是否是 G 中的顶点，利用广度优先搜索计算出 G 中从 u 到 v 的最短路径，然后把得到的最短路径上的边数与 k 进行比较。如果 G 编码了无向图，且从 u 到 v 的路径中至多有 k 条边，则算法输出 1 并停机；否则，该算法永远运行下去。但是，这一算法并没有对 PATH 问题进行判定，因为对最短路径长度多于 k 条边的实例，算法并没有显式地输出 0。PATH 的判定算法必须显式地拒绝不属于 PATH 的二进制串。对 PATH 这样的判定问题来说，很容易设计出这样的一种判定算法：当不存在从 u 到 v 的、包含至多 k 条边的路径时，算法不是永远地运行下去，而是输出 0 并停机。对于其他的一些问题(如图灵停机问题)，只存在接受算法，而不存在判定算法。

我们可以非形式地定义一个*复杂性类*(complexity class)为语言的一个集合，某一语言是否属于该集合，可以通过某种*复杂性度量*(complexity measure)来确定，比如一个算法的运行时间，该算法可以确定某个给定的串 x 是否属于语言 L。复杂性类的实际定义要更专业一些，有兴趣的读者可以进一步参考 Hartmanis 和 Stearns[140]的那篇开创性的文章。

运用上述的形式语言理论体系，可以提出关于复杂性类 P 的另外一种定义：

$P=\{L\subseteq\{0,1\}^*$：存在一个算法 A 能在多项式时间内判定 $L\}$

事实上，P 也是能在多项式时间内被接受的语言类。

定理 34.2　$P=\{L$：L 能被一个多项式时间的算法所接受$\}$。

证明：因为由多项式时间算法判定的语言类是多项式时间算法接受的语言类的一个子集，所以，我们仅需要证明如果 L 能被一个多项式时间的算法所接受，它也能够被一个多项式时间的算法所判定。设 L 是被某个多项式时间算法 A 所接受的语言。我们将运用经典的"模拟"论证方法，来构造另一个能够判定 L 的多项式时间算法 A'。因为对某个常数 k，A 能在 $O(n^k)$ 时间内接受 L，所以也存在一个常数 c，使 A 至多在 $T=cn^k$ 步内就能接受 L。对任意输入的串 x，算法
977 A' 模拟 A 在时间 T 内的操作状态。在时间 T 结束时，算法 A' 即检查 A 的行为。如果 A 接受了 x，则 A' 通过输出 1 来接受 x；如果 A 没有接受 x，则 A' 通过输出 0 来拒绝 x。A' 模拟 A 的开销对运行时间的影响不会大于一个多项式因子，因此，A' 是一个能判定 L 的多项式时间算法。 ■

注意，定理 34.2 的证明过程是非构造性的。对于一个给定的语言 $L\in P$，我们也许实际上并不知道接受 L 的算法 A 的运行时间界，但是，我们知道这样的一个界是存在的。因此，我们知道存在着能够检查该界的算法 A'，只是算法 A' 不容易找到而已。

练习

34.1-1　定义最优化问题 LONGEST-PATH-LENGTH 为一个关系，它将一个无向图的每个实例、两个顶点与这两个顶点间最长路径中所包含的边数联系了起来。定义判定问题 LONGEST-PATH$=\{(G,u,v,k)$：$G=(V,E)$ 为一个无向图，u、$v\in V$，$k\geqslant 0$ 是一个整数，且 G 中存在着一条从 u 到 v 的简单路径，它包含了至少 k 条边$\}$。证明最优化问题 LONGEST-PATH-LENGTH 可以在多项式时间内解决，当且仅当 LONGEST-PATH$\in$P。

34.1-2　对于在无向图中寻找最长简单回路这一问题，给出其形式化的定义及相关的判定问题。另外，给出与该判定问题对应的语言。

34.1-3　给出一种形式化的编码，它利用邻接矩阵表示形式，将有向图编码为二进制串。另外，

再给出利用邻接表表示的编码。论证这两种表示是多项式时间相关的。

34.1-4　练习 16.2-2 中曾要求读者给出 0-1 背包问题的动态规划算法，它是个多项式时间的算法吗？对你的回答进行解释。

34.1-5　证明：对于一个多项式时间的算法，当它调用一个多项式时间的子例程时，如果至多调用常数次，则此算法以多项式时间运行，但是，当调用子例程的次数为多项式时，此算法就可能变成一个指数时间的算法。

978

34.1-6　证明：类 P 在被看作是一个语言集合时，在并集、交集、并置、补集和 Kleene 星运算下是封闭的。亦即，如果 L_1、$L_2 \in \mathrm{P}$，则 $L_1 \cup L_2 \in \mathrm{P}$，等等。

34.2　多项式时间的验证

现在来看看对语言成员进行“验证”的算法。例如，假定对判定问题 PATH 的一个给定实例 $\langle G, u, v, k\rangle$，同时也给定了一条从 u 到 v 的路径 p。我们可以检查 p 的长度是否至多为 k。如果是的，就可以把 p 看作是该实例的确属于 PATH 的“证书”(certificate)。对于判定问题 PATH 来说，这一证书并没有使我们得益多少。毕竟，PATH 属于 P，事实上，PATH 可以在线性时间内求解。因此，根据指定的证书来验证成员所需的时间与从头开始解决问题的时间一样长。现在来考虑一个问题，我们知道它没有多项式时间的判定算法，但是对于指定的证书，验证却是比较容易的。

哈密顿回路

在无向图中找出哈密顿回路(Hamiltonian cycle)这一问题已被研究 100 多年了。形式地说，无向图 $G=(V, E)$ 中的一个哈密顿回路是通过 V 中每个顶点一次的简单回路。具有这种回路的图称为哈密顿图，否则称为非哈密顿图。Bondy 和 Murty[45]曾引述过 W. R. Hamilton 写的一封信，信中描述了一个在正十二面体上做的数学游戏，如图 34-2a 所示，一个游戏者在任意 5 个连续顶点上钉上 5 个图钉。另一个游戏者必须完成路径，以形成一个包含所有顶点的回路。正十二面体是哈密顿图，图 34-2a 显示了一条哈密顿回路。但是，并不是所有的图都是哈密顿图。例如，图 34-2b 示出了一个具有奇数个顶点的二分图。练习 34.2-2 中将要求读者证明所有这样的图都是非哈密顿图。

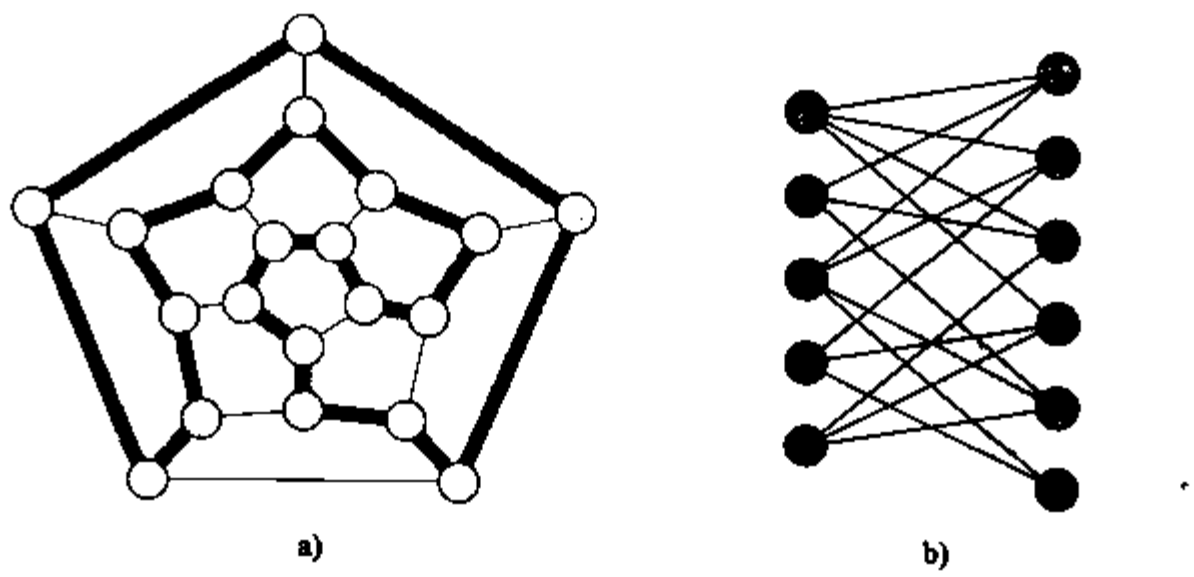


图 34-2　a)一个表示了正十二面体中顶点、边和面的图，其中的哈密顿回路以阴影边示出。b)一个包含奇数个顶点的二分图。任何这样的图都是非哈密顿图

我们可以用下列形式语言来定义哈密顿回路问题："图 G 中是否具有一条哈密顿回路?"

HAM-CYCLE=$\{\langle G\rangle: G$ 是一个哈密顿图$\}$

用算法如何来判定语言 HAM-CYCLE? 已知一个问题实例$\langle G\rangle$，一种可能的判定算法就是罗列出 G 的顶点的所有排列，然后对每一种排列进行检查，以确定它是否是一条哈密顿回路。那么，该算法的运行时间是多少呢? 如果我们"合理地"把图编码为其邻接矩阵，图中顶点数 m 为 $\Omega(\sqrt{n})$，其中 $n=|\langle G\rangle|$ 是 G 的编码长度，则总共会有 $m!$ 种可能的顶点排列，因此，算法的运行时间为 $\Omega(m!)=\Omega(\sqrt{n}!)=\Omega(2^{\sqrt{n}})$，它并非是 $O(n^k)$ 的形式(k 为任意常数)。因此，这种朴素算法的运行时间并不是多项式时间。事实上，哈密顿问题是 NP 完全的问题，我们将在 34.5 节中证明这一结论。

验证算法

现在来考虑一个稍为容易一些的问题。假设有个人说某给定图 G 是哈密顿图，并提出可以通过给出沿哈密顿回路排列的顶点来证明他的话。证明当然是非常容易的：仅仅需要检查所提供的回路是否是 V 中顶点的一个排列，以及沿回路的每条连续的边是否在图中存在，这样就可以验证所提供的回路是否是哈密顿回路。当然，该验证算法可以在 $O(n^2)$ 的时间内实现，其中 n 是 G 的编码的长度。因此，我们可以在多项式时间内验证图中存在一条哈密顿回路和证明过程。

979
~
980

我们把验证算法定义为含两个自变量的算法 A，其中一个自变量是普通输入串 x，另一个是称为"证书"的二进制串 y。如果存在一个证书 y 满足 $A(x, y)=1$，则该含两个自变量的算法 A 验证了输入串 x。由一个验证算法 A 所验证的语言是：

$$L=\{x\in\{0,1\}^*: \text{存在 } y\in\{0,1\}^*, \text{满足 } A(x,y)=1\}$$

从直观上来看，如果对任意串 $x\in L$，都存在一个证书 y，且 A 可以用 y 来证明 $x\in L$，则算法 A 就验证了语言 L。此外，对任意串 $x\notin L$，必须不存在能证明 $x\in L$ 的证书。例如，在哈密顿回路问题中，证书是哈密顿回路中顶点的列表。如果一个图是哈密顿图，哈密顿回路本身就提供了足够的信息来验证这一事实。相反地，如果某个图不是哈密顿图，那么也不存在这样的顶点列表能使验证算法认为该图是哈密顿图，因为验证算法会仔细地检查所提供的"回路"是否不是哈密顿回路。

复杂类 NP

复杂类 NP 是能被一个多项式时间算法验证的语言类。[⊖]更准确地说，一个语言 L 属于 NP，当且仅当存在一个两输入的多项式时间算法 A 和常数 c 满足：

$$L=\{x\in\{0,1\}^*: \text{存在一个证书 } y(|y|=O(|x|^c)) \text{ 满足 } A(x,y)=1\}$$

我们说算法 A 在多项式时间内验证了语言 L。

根据先前我们对哈密顿回路问题的讨论，可知 HAM-CYCLE$\in$NP。(能知道某个重要的集合是非空的总是件好事。)此外，如果 $L\in$P，则 $L\in$NP，如果存在一个多项式时间的算法来判定 L，那么只要忽略任何证书，并接受那些它确定属于 L 的输入串，就可以很容易地把该算法转化为一个两参数的验证算法。因此，P$\subseteq$NP。

目前还不知道是否有 P=NP，但大多数研究人员认为 P 和 NP 不是同一个类。从直觉上看，类 P 由可以很快解决的问题组成，而类 NP 由可以很快验证其解的问题组成。从实际经验中读者

⊖ "NP"这一名称代表"非确定性多项式时间"(nondeterministic polynomial time)。NP 类最初是在非确定性这一上下文中得到研究的，但本书采用了一种更为简单但等价的验证表示记号。Hopcroft 和 Ullman[156]利用非确定计算模型，给出了 NP 完全性的一种很好的表述。

也许已经知道，从头开始解决一个问题常常要比验证一个明确给出的解要困难得多，在有时间限制的情况下更是如此。从事理论研究的计算机科学家一般都认为，这一类比可以延伸到类 P 和 NP 上，因此 NP 包括了不属于 P 的语言。

此外，还有更令人信服的证据能说明 P≠NP，即存在着“NP 完全”的语言。34.3 节将对这类语言进行研究。

在 P≠NP 问题之外，还有许多其他基本问题没有解决，尽管很多研究人员做了大量的工作，但还没有人知道 NP 类在补运算下是否是封闭的，亦即，$L \in \text{NP}$ 是否说明了 $\overline{L} \in \text{NP}$ 的语言 L 的集合。我们可以定义复杂类 co-NP 为满足 $\overline{L} \in \text{NP}$ 的语言 L 构成的集合。这样一来，NP 在补运算下是否封闭的问题就可以重新表示为是否有 NP＝co-NP。由于 P 在补运算下具有封闭性(见练习 34.1-6)，所以有 $\text{P} \subseteq \text{NP} \cap \text{co-NP}$。但是，我们仍然不知道是否有 $\text{P} = \text{NP} \cap \text{co-NP}$，或者在 $\text{NP} \cap \text{co-NP}$ 一 P 中是否存在某种语言。图 34-3 说明了四种可能的方案。

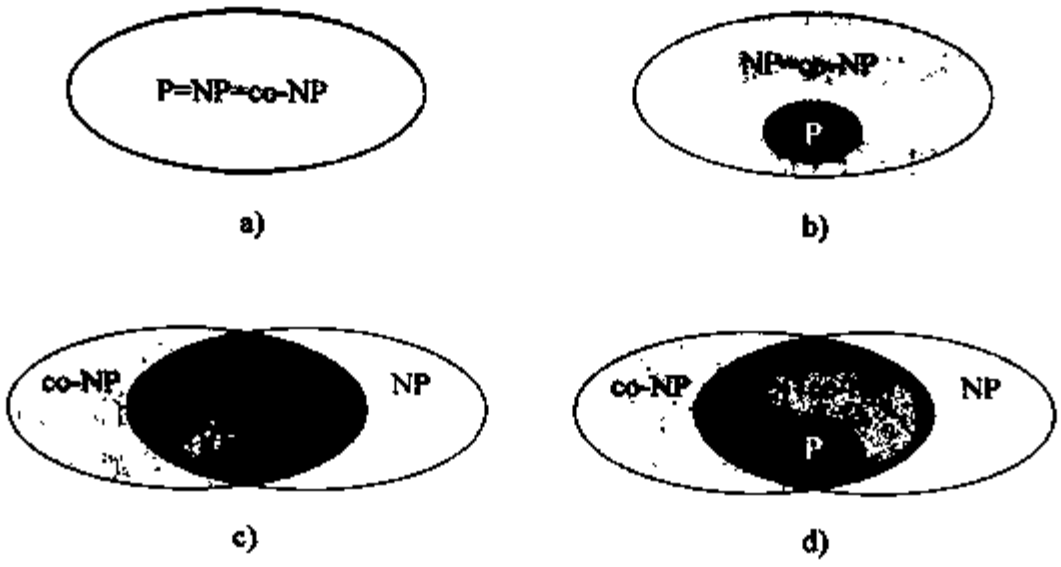


图 34-3　复杂类之间的四种可能关系。在每一个图中，一个区域包围另一个区域表明一种真子集关系。a)P＝NP＝co-NP。多数研究人员都认为这种情况是最不可能的。b)如果 NP 在补集运算下是封闭的，则 NP＝co-NP，但不一定有 P＝NP。c)P＝NP∩co-NP，但 NP 在补集运算下不封闭。d)NP≠co-NP，且 P≠ NP∩co-NP。多数研究人员认为这种情况的可能性最大

令人遗憾的是，我们对 P 与 NP 之间确切关系的理解是很不完全的。然而，通过探讨 NP 完全性理论，从实用角度来看，我们在证明问题是难处理的过程中，所遇到的不利因素也许会比想像中的少一些。

练习

34.2-1　考虑语言 GRAPH-ISOMORPHISM＝{⟨G_1，G_2⟩：G_1 和 G_2 是同构图}。通过描述一个可以在多项式时间内验证该语言的算法，来证明 GRAPH-ISOMORPHISM∈NP。

34.2-2　证明：如果 G 是一个无向的二分图，且有着奇数个顶点，则 G 是非哈密顿图。

34.2-3　证明：如果 HAM-CYCLE∈P，则按序列出一个哈密顿回路中各个顶点的问题是多项式时间可解的。

34.2-4　证明：由语言构成的 NP 类在并集、交集、并置和 Kleene 星运算下是封闭的。讨论一下 NP 在补集运算下的封闭性。

34.2-5　证明：NP 中的任何语言都可以用一个运行时间为 $2^{O(n^k)}$(其中 k 为常数)的算法来加以判定。

34.2-6 图中的哈密顿路径是一种简单路径，它经过图中每个顶点一次。证明：语言 HAM-PATH=$\{\langle G, u, v\rangle$：图 G 中存在一条从 u 到 v 的哈密顿路径$\}$属于 NP。

34.2-7 证明：在有向无回路图中，哈密顿路径问题可以在多项式时间内求解。给出解决该问题的一个有效算法。

34.2-8 设 ϕ 为一个布尔公式，它由布尔输入变量 x_1，x_2，…，x_k，非(¬)、AND(∧)、OR(∨)和括号组成。如果对公式 ϕ 的输入变量的每一种 1 和 0 赋值，公式的结果都为 1，则称其为重言式(tautology)。定义 TAUTOLOGY 为由重言布尔公式所组成的语言。证明：TAUTOLOGY∈co-NP。

34.2-9 证明：P⊆co-NP。

34.2-10 证明：如果 NP≠co-NP，则 P≠NP。

34.2-11 设 G 为一个包含至少 3 个顶点的连通无向图，并设对 G 中所有由长度至多为 3 的路径连接起来的点对，将它们直接连接后所形成的图为 G^3。证明：G^3 是一个哈密顿图。（提示：为 G 构造一棵生成树，并采用归纳法进行证明。）

983

34.3　NP 完全性与可归约性

从事理论研究的计算机科学家们之所以会相信 P≠NP，最令人信服的理由可能就是存在着一类“NP 完全”问题。该类问题有一种令人惊奇的性质，即如果一个 NP 完全问题能在多项式时间内得到解决，那么，NP 中的每一个问题都可以在多项式时间内求解，即 P=NP。但是，尽管进行了多年的研究，目前还没有找出关于任何 NP 完全性问题的多项式时间的算法。

语言 HAM-CYCLE 就是一个 NP 完全问题。如果我们能够在多项式时间内判定 HAM-CYCLE，就能够在多项式时间内求解 NP 中的每一个问题。事实上，如果能证明 NP−P 为非空集合，就可以肯定地说 HAM-CYCLE∈NP−P。

在某种意义上说，NP 完全语言是 NP 中“最难”的语言。在本节中，要说明如何运用称为“多项式时间可归约性”的确切概念，来比较各种语言的相对“难度”。首先，我们将正式定义 NP 完全语言，然后，再简要地证明一种称为 CIRCUIT-SAT 的语言是 NP 完全语言。在 34.4 节和 34.5 节中，将运用可归约性概念来证明很多其他问题都是 NP 完全的。

可归约性

从直觉上看，一个问题 Q 可以被归约为另一个问题 Q'，如果 Q 的任何实例都可以被“容易地重新描述为”Q' 的实例，而 Q' 的实例的解也是 Q 的实例的解。例如，求解关于未知量 x 的线性方程问题可以转化为求解二次方程问题。已知一个实例 $ax+b=0$，可以把它变换为 $0x^2+ax+b=0$，其解也是方程 $ax+b=0$ 的解。因此，如果一个问题 Q 可以转化为另一个问题 Q'，则从某种意义上来说，Q 并不比 Q' 更难解决。

回到关于判定问题的形式语言体系中，我们说语言 L_1 在多项式时间内可以归约为语言 L_2，写作 $L_1 \leqslant_P L_2$，如果存在一个多项式时间可计算的函数 f：$\{0,1\}^* \to \{0,1\}^*$，满足对所有的 $x \in \{0,1\}^*$，都有：

$$x \in L_1 \text{ 当且仅当 } f(x) \in L_2 \tag{34.1}$$

称函数 f 为归约函数，计算 f 的多项式时间算法 F 称为归约算法。

图 34-4 说明了关于从语言 L_1 到另一种语言 L_2 的多项式时间归约的思想。每一种语言都是 $\{0,1\}^*$ 的子集，归约函数 f 提供了一个多项式时间的映射，使得如果 $x \in L_1$，则 $f(x) \in L_2$。

984

如果 $x \notin L_1$，则 $f(x) \notin L_2$。因此，归约函数提供了从语言 L_1 表示的判定问题的任意实例 x 到语

言 L_2 表示的判定问题的实例 $f(x)$上的映射。如果能提供是否有 $f(x)\in L_2$ 的答案，也就直接提供了是否有 $x\in L_1$ 的答案。

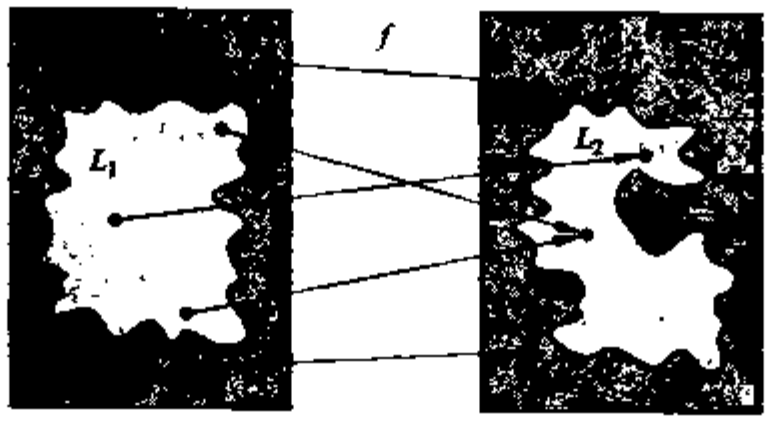


图 34-4　通过归约函数 f，在多项式时间内将语言 L_1 归约为语言 L_2。对任何输入 $x\in\{0,1\}^*$，是否有 $x\in L_1$ 这一问题与是否有 $f(x)\in L_2$ 的答案是一样的

多项式时间归约对证明各种语言属于 P 提供了一种有力的工具。

引理 34.3　如果 L_1、$L_2\subseteq\{0,1\}^*$ 是满足 $L_1\leqslant_P L_2$ 的语言，则 $L_2\in$ P 蕴含着 $L_1\in$ P。

证明：设 A_2 是一个判定问题 L_2 的多项式时间算法，F 是计算归约函数 f 的多项式时间归约算法。下面来构造一个判定 L_1 的多项式时间算法 A_1。

图 34-5 说明了 A_1 的构造过程。对给定的输入 $x\in\{0,1\}^*$，算法 A_1 利用 F 把 x 变换为 $f(x)$，然后它利用 A_2 测试是否有 $f(x)\in L_2$。A_2 的输出值提供给 A_1 作为输出。

图 34-5　引理 34.3 的证明。算法 F 是一个归约算法，它在多项式时间内计算出从 L_1 到 L_2 的归约函数 f，A_2 是一个能判定 L_2 的多项式时间算法。图中示出的是一个算法 A_1 通过利用 F 将任何输入 x 转换成 $f(x)$，再利用 A_2 来判定是否有 $f(x)\in L_2$，最终判定是否有 $x\in L_1$ 的过程

985

根据条件(34.1)可以推导出 A_1 的正确性。该算法的运行时间为多项式时间，因为 F 和 A_2 的运行时间都是多项式时间(参见练习 34.1-5)。■

NP 完全性

多项式时间归约提供了一种形式方法，用来证明一个问题在一个多项式时间因子内至少与另一个问题一样难。亦即，如果 $L_1\leqslant_P L_2$，则 L_1 大于 L_2 的难度不会超过一个多项式时间因子，这就是我们采用“小于或等于”来表示归约记号的原因。现在，我们就可以定义 NP 完全语言的集合，这些都是 NP 中最难的问题。

语言 $L\subseteq\{0,1\}^*$ 是 NP 完全的，如果：

1. $L\in$ NP
2. 对每一个 $L'\in$ NP，有 $L'\leqslant_P L$

如果一种语言 L 满足性质 2，但不一定满足性质 1，则称 L 是 NP 难度(NP-hard)的。我们也定义 NPC 为 NP 完全语言类。

正如下列定理所述，NP 完全性是判定 P 是否等于 NP 的关键。

定理 34.4　如果任何 NP 完全问题是多项式时间可求解的，则 P=NP。等价地，如果 NP 中的任何问题不是多项式时间可求解的，则所有 NP 完全问题都不是多项式时间可求解的。

证明：假定 $L \in P$ 并且 $L \in NPC$。对任意 $L' \in NP$，由 NP 完全性定义中的性质 2，有 $L' \leqslant_P L$。这样，根据引理 34.3，就有 $L' \in P$，这样就证明了本定理的第一个结论。

第二个结论是第一个结论的对换句，因此第二个结论也得证。 ■

正是因为这个原因，对 P≠NP 问题的研究都是以 NP 完全问题为中心的。大部分从事理论研究的计算机科学家们都认为 P≠NP，据此可以导出图 34-6 中所示的 P、NP 与 NPC 之间的关系。但是，我们都知道，也许有人会找出关于一个 NP 完全问题的多项式时间算法，这样就能证明 P=NP。然而，由于迄今为止还没有找出任何 NP 完全问题的多项式时间算法，所以在目前，证明了一个问题具有 NP 完全性，也就可以提供其难处理性的极好证明。

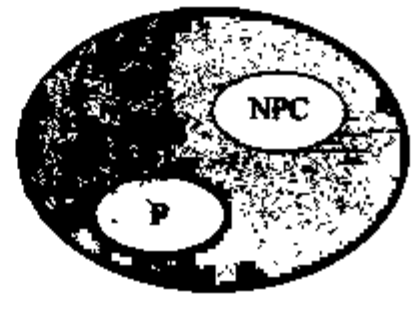


图 34-6　大多数理论计算机科学家们眼里的 P、NP 和 NPC 三者之间的关系。P 和 NPC 都完全包含在 NP 内，且 P∩NPC=∅

986

电路可满足性

前面已经定义了 NP 完全问题这一概念，但到现在为止，我们实际上还没有证明任何问题是 NP 完全问题。一旦我们证明了至少有一个问题是 NP 完全问题，就可以用多项式时间可归约性作为工具，来证明其他问题也具有 NP 完全性。因此，下面来着重证明存在一个 NP 完全问题：即电路可满足性(circuit-satisfiability)问题。

不幸的是，在电路可满足性问题的形式化证明中，需要一些超出本书范围的技术细节。因此，我们将非正式地描述一种基于基本的布尔组合电路知识的证明过程。

布尔组合电路是由布尔组合元素通过电路互连后构造而成的。布尔组合元素是指任何一种电路元素，它有着固定数目的输入和输出，执行的是某种良定义的函数功能。布尔值取自集合 {0，1}，其中 0 代表 FALSE(假)，1 代表 TRUE(真)。

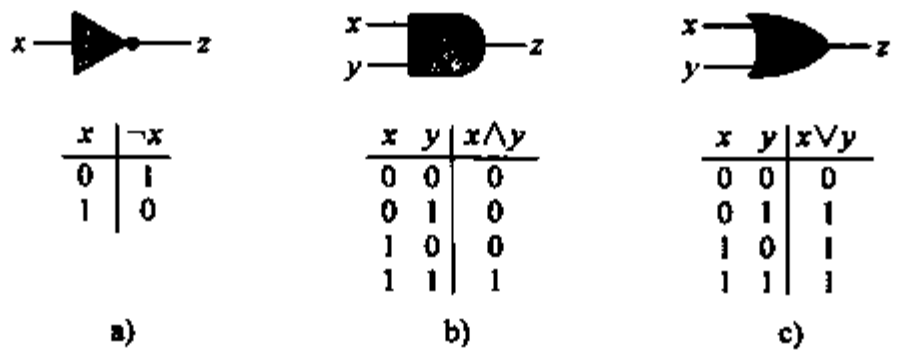
在电路可满足性问题中，所用到的布尔组合元素计算的是一个简单的布尔函数，这些元素称为逻辑门。图 34-7 示出了在电路可满足性问题中用到的三种基本的逻辑门：NOT 门(非门，也称为反向器)、AND 门(与门)和 OR 门(或门)。NOT 门只有一个二进输入 x，它的值为 0 或 1，产生的是二进输出 z，其值与输入值相反。另外的两种门都取两个二进输入 x 和 y，产生一个二进输出 z。

每一种门及任何一种布尔组合元素的操作都可以用一个真值表来描述，如图 34-7 中所示。真值表给出了对于输入组合的每一种可能取值，组合元素的输出情况。例如，OR 门的真值表告诉我们，当输入为 $x=0$ 和 $y=1$ 时，输出值为 $z=1$。我们用符号¬来表示 NOT 函数，用∧来表示 AND 函数，用∨来表示 OR 函数。例如，0∨1=1。

987

我们可以将 AND 门和 OR 门加以推广，使其可以有多于两个的输入。对 AND 门来说，如果其所有输入均为 1，则其输出为 1，否则其输出为 0。对 OR 门来说，如果其任何一个输入为 1，则其输出为 1，否则为 0。

布尔组合电路由一个或多个布尔组合元素通过线路连接而成。一个电路将某一元素的输出与另一个元素的输入连接起来，即将第一个元素的输出值提供给第二个元素作为其输入值。图 34-8 示出了两个类似的布尔组合电路，它们仅在一个门上有所不同。图 34-8a 示出了当输入为



x	$\neg x$
0	1
1	0

a)

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

b)

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

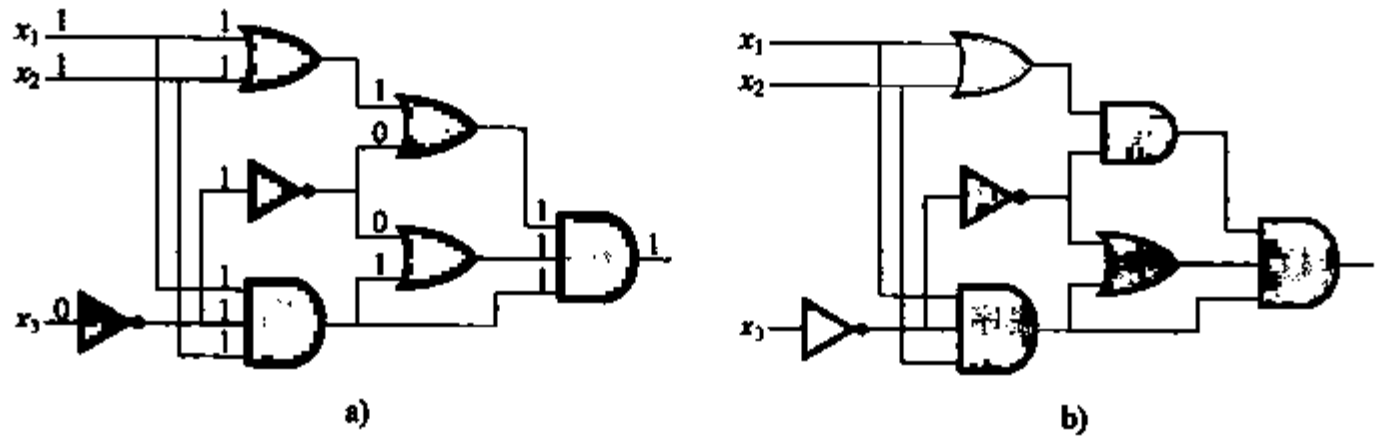
c)

图 34-7　三种基本的逻辑门，它们都具有二进制形式的输入和输出。在每一种门下面，是描述该门电路操作的真值表。a)NOT 门，b)AND 门，c)OR 门

$\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$时，每根接线上的值。虽然一根线上，不可能有多于一个的布尔元素的输出与其相连，但它可以作为其他几个元素的输入。由一个接线提供输入的元素的个数，称为该接线的扇出。如果没有哪一个元素的输出是接到某根接线上的话，则称该线是电路的输入，它接受来自外部的数据。如果没有哪一个元素的输入连接到某根接线上的话，则称该接线为电路的输出，它将电路的计算结果提供给外部。(一根内部接线也可以扇出至电路的输出上。)为了定义电路可满足性问题，我们将电路的输出限制为 1；在实际的硬件设计中，布尔组合电路是可以有多个输出的。

布尔组合电路不包含回路。换句话说，假设我们创建了一个有向图 $G=(V, E)$，其中每一个顶点代表一个组合元素，k 条有向边代表每一根扇出为 k 的接线；如果某一接线将一个元素 u 的输出与另一个元素 v 的输入连接了起来，图中就会有一条有向边(u, v)。那么，G 必定是无回路的。

一个布尔组合电路的真值赋值是指一组布尔输入值。如果一个单输出布尔组合电路具有一个可满足性赋值(即使得电路的输出为 1 的一个真值赋值)，就称该布尔组合电路是可满足的。例如，图 34-8a 中的电路具有可满足性赋值$\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$，因此它是可满足的。如练习 34.3-1 中要求读者说明的那样，不存在对 x_1、x_2 和 x_3 的赋值，使得图 34-8b 中的电路产生输出为 1，它总是输出 0，因此，它是不可满足的。



a)　　b)

图 34-8　电路可满足性问题的两个实例。a)对此电路的输入赋值$(x_1 = 1, x_2 = 1, x_3 = 0)$，使得电路的输出为 1。因而，此电路是可满足的。b)对此电路输入的任何一种赋值都不能使得电路的输出为 1。因而，此电路是不可满足的

电路可满足性问题就是："给定一个由与、或和非门构成的一个布尔组合电路，它是可满足电路吗?"为了给出这一问题的形式定义，必须对电路的编码有一个统一的标准。布尔组合电路的规模是指其中布尔组合元素的个数，再加上电路中接线的数目。我们可以设计出一种像图形那样的编码，使其可以把任何给定电路 C 映射为一个二进制串$\langle C \rangle$，该串的长度与电路本身的规模

呈多项式关系。于是，作为一种形式语言，我们可以定义：

CIRCUIT-SAT＝{⟨C⟩：C 是一个可满足的布尔组合电路}

电路可满足性问题在计算机辅助硬件优化领域中极其重要。如果一个子电路总是输出 0，就可以用一个更为简单的子电路来取代原电路，该子电路省略了所有逻辑门，并提供常数值 0 作为其输出。如果能够开发关于该问题的多项式时间算法，那将具有很大的实际应用价值。

988
~
989

给定一个电路 C，通过检查输入的所有可能赋值来确定它是否是可满足性电路。遗憾的是，如果有 k 个输入，就会有 2^k 种可能的赋值。当电路 C 的规模为 k 的多项式时，对每个电路进行检查需要 $\Omega(2^k)$的时间，这与电路的规模呈超多项式关系。[⊖]事实上，如前面所述，有很强的证据表明，不存在能解决电路可满足性问题的多项式时间算法，因为该问题是 NP 完全的。根据 NP 完全性定义中的两个部分，把对这一事实的证明过程也分为两部分。

引理 34.5　电路可满足性问题属于 NP 类。

证明：我们将提出一个能验证 CIRCUIT-SAT 的、两输入的多项式时间算法 A。A 的一个输入是布尔组合电路 C(的标准编码)。另一个输入是一个相应于 C 中线路的一个布尔型赋值的证书。(练习 34.3-4 中提供了一个更小的证书。)

对算法 A 的构造如下：对电路中的每个逻辑门，算法要检查输出线路上证书所提供的值，看它是否是根据输入线路值正确计算出的一个函数值。然后，如果整个电路的输出为 1，则算法输出 1，因为对电路 C 的输入所赋的值提供了一种可满足性赋值。否则，算法 A 输出 0。

每当将一个可满足的电路 C 输入算法 A 时，必会有一个证书，其长度为 C 的规模的多项式，并使 A 输出 1。每当将一个不可满足的电路作为 A 的输入时，则不存在这样的证书让 A 认为该电路是可满足的。算法 A 的运行时间为多项式时间：如果运用较好的实现方法，可以达到线性时间。因此，CIRCUIT-SAT 可以在多项式时间内被验证，从而有 CIRCUIT-SAT∈NP。 ■

证明 CIRCUIT-SAT 是 NP 完全问题的第二部分，就是要证明该语言是 NP 难度的，即必须证明 NP 中的每一种语言，都可以在多项式时间内归约为 CIRCUIT-SAT。这一事实的实际证明过程比较复杂，因此，我们将基于计算机硬件的工作机理，来给出一个简要的证明过程。

计算机程序是作为一个指令序列存储于计算机存储器中的。一条典型的指令包含操作代码，操作数在存储器中的地址以及结果的存储地址。一个特定的称为程序计数器的存储器单元记录
990 了将被执行的下一条指令的地址。每当取出一条指令时，程序计数器即自动地增值，这样就可以使计算机按顺序执行指令。但是，一条指令执行后，可以使一个值被写入程序计数器中，于是正常的执行顺序发生改变，以使计算机可以执行循环或条件分支语句。

在程序执行过程中的任一时刻，计算过程的整个状态都表示于计算机的存储器里。(此处所说的存储器包括程序自身、程序计数器、工作存储器以及计算内务操作所设置的各种状态位。)我们把计算机存储的任何一种特定状态称为一个配置。执行一条指令可以看作使一个配置映射为另一个配置。重要的是，实现这种映射关系到的计算机硬件可以用一个布尔组合电路来实现，在下列引理的证明过程中，用 M 来表示该布尔组合电路。

引理 34.6　电路可满足性问题是 NP 难度的。

证明：设 L 是 NP 中的任意语言。我们将描述一个多项式时间的算法 F 来计算归约函数 f，该函数把每个二进制串 x 都映射为一个电路 $C=f(x)$，使得 $x\in L$ 当且仅当 $C\in$CIRCUIT-SAT。

⊖ 另一方面，如果电路 C 的规模为 $\Theta(2^k)$，则对运行时间为 $O(2^k)$的算法来说，其运行时间与电路规模是呈多项式关系的。即使 P≠NP，这种情况也不会与该问题是 NP 完全的这一事实矛盾；对某种特例存在多项式时间的算法，并不意味着对所有情况都存在多项式时间的算法。

由于 $L \in \mathrm{NP}$，所以必定存在着一个算法 A，它可以在多项式时间内验证 L。我们将构造的算法 F 将使用两输入的算法 A 来计算归约函数 f。

设 $T(n)$表示算法 A 对长度为 n 的输入串在最坏情况下的运行时间，并设 $k \geqslant 1$ 为一个常数，满足 $T(n) = O(n^k)$，且证书的长度为 $O(n^k)$。（A 的运行时间实际上是关于整个输入规模的一个多项式，既包括输入串也包括证书，但由于证书的长度是关于输入串长度 n 的多项式，所以运行时间也是关于 n 的多项式。）

证明的基本思想是把 A 的计算过程表示成一个配置序列，如图 34-9 所示。每个配置可以被划分为数个部分，包括 A 的程序、程序计数器、辅助机器状态、输入 x、证书 y 和工作存储器。从初始配置 c_0 开始，每个配置 c_i 都由实现计算机硬件的组合电路 M 映射到其随后的配置 c_{i+1}。当算法 A 终止执行时，其输出(0 或 1)被写入到工作存储器的某个指定单元中，并且，如果我们假定此后 A 会停止，则该值不会改变。因此，如果算法至多执行 $T(n)$步，则输出出现于 $c_{T(n)}$ 中的一位上。

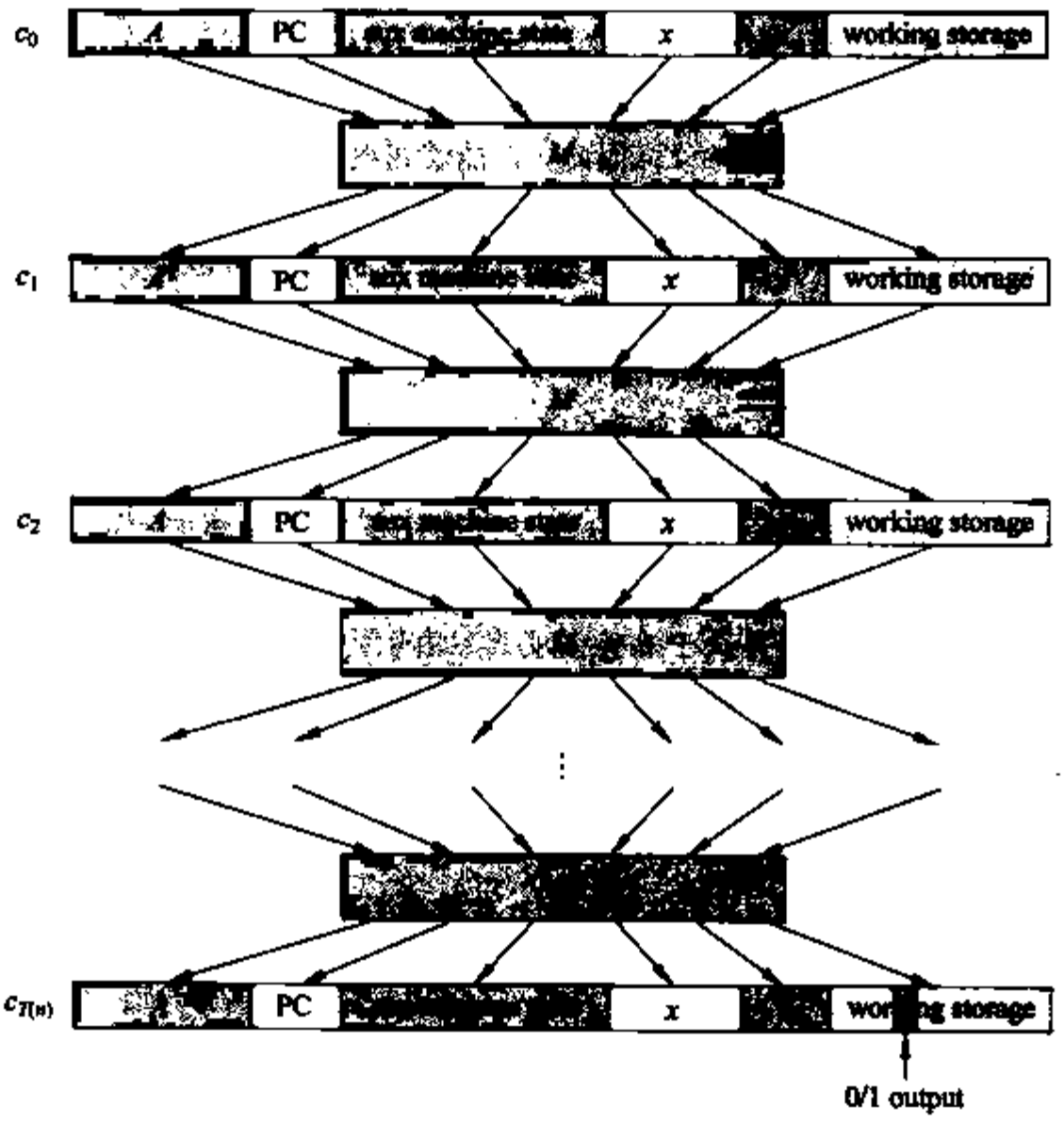


图 34-9　算法 A 在输入 x 和证书 y 上运行时所产生的配置序列。每个配置都表示了计算机在一步计算之后的状态，除了 A、x 和 y 之外，还包括了程序计数器(PC)、辅助机器状态和工作存储器。除了证书 y 外，初始配置 c_0 是固定的。通过一个布尔组合电路 M，每个配置被映射到下一配置上。输出是工作存储器中一个特别的位

归约算法 F 构造出一个组合电路，它根据给定的初始配置计算出产生的全部配置。其设计思想为复制 $T(n)$个电路 M 的拷贝，并把它们粘贴在一起。产生配置 c_i 的第 i 个电路的输出直接馈送作为第$(i+1)$个电路的输入。因此，这些配置并非终止于一个状态寄存器中，而是仅仅驻留于连接 M 的拷贝之间的线路上。

991
~
992

我们来回顾一下多项式时间归约算法 F 必须做的工作。给定一个输入 x，它必须计算出一个电路 $C=f(x)$，C 是可满足电路，当且仅当存在一个证书 y 满足 $A(x, y)=1$。当 F 获得一个输入 x 时，它首先计算出 $n=|x|$，然后构造出一个由 $T(n)$ 个 M 的拷贝组成的组合电路 C'，C' 的输入是对应于对 $A(x, y)$ 进行计算的初始配置，输出为配置 $c_{T(n)}$。

F 所计算出的电路 $C=f(x)$ 是对 C' 稍作修改而得到的。首先，相应于 A 的程序的 C' 的输入，初始的程序计数器、输入 x 和存储器的初始状态的线路直接与这些已知值相连。因此，电路剩下的唯一输入对应于证书 y。其次，电路的所有输出都被忽略，但对应于 A 的输出的 $c_{T(n)}$ 中的一位除外。这样构造成的电路 C 对长度为 $O(n^k)$ 的任意输入计算出 $C(y)=A(x, y)$。当我们给归约算法 F 提供一个输入串 x 时，它就计算出这样一个电路 C 并输出。

接下来还要证明两条性质。第一，必须证明 F 能够正确地计算出归约函数 f，即必须证明 C 是可满足的当且仅当存在一个证书 y，满足 $A(x, y)=1$。第二，必须证明 F 的运行时间为多项式时间。

为了证明 F 能够正确地计算出归约函数，假设存在一个长度为 $O(n^k)$ 的证书 y 满足 $A(x, y)=1$。那么，如果把 y 的各位作为 C 的输入，则 C 的输出为 $C(y)=A(x, y)=1$。因此，如果有一个证书存在，则 C 是可满足电路。另一方面，假定 C 是可满足的，则对 C 存在一个输入 y 满足 $C(y)=1$，据此，可以得到 $A(x, y)=1$。因此，F 能够正确地计算出一个归约函数。

为了完成证明过程，仅需证明 F 的运行时间是关于 $n=|x|$ 的多项式时间。首先注意到表示一个配置所需的位数是关于 n 的多项式。A 的程序本身的规模为常数，与其输入 x 的长度无关。输入 x 的长度为 n，证书 y 的长度为 $O(n^k)$。由于算法至多运行 $O(n^k)$ 步，所以 A 所要求的工作存储器数量也是 n 的多项式。（假定该存储器单元是连续的；练习 34.3-5 要求读者把证明扩展到下列情况：A 所存取的存储单元散布于存储器的一个大范围内，对每个输入 x，其特定的散布方式也可能不同。）

实现计算机硬件的组合电路 M 的规模是关于配置的长度的多项式，即为 $O(n^k)$ 的多项式，因而也是关于 n 的多项式。（这个电路的大部分实现了存储系统的逻辑。）电路 C 至多由 $t=O(n^k)$ 个 M 的拷贝所组成，因此其规模是关于 n 的多项式。由于构造过程的每一步都需要多项式时间，所以用归约算法 F 可以在多项式时间内，完成从 x 构造电路 C 的过程。 ■

993

综上所述，语言 CIRCUIT-SAT 至少与 NP 中的任何语言具有同样的难度，并且又因为它是属于 NP 的，故它是 NP 完全的。

定理 34.7　电路可满足性问题是 NP 完全的。

证明：根据引理 34.5 和引理 34.6 以及 NP 完全性的定义，可以直接推得结论。 ■

练习

34.3-1　验证图 34-8b 中的电路是不可满足的。

34.3-2　证明：$\leqslant_P$ 关系是语言上的一种传递关系。亦即，要证明如果有 $L_1\leqslant_P L_2$，且 $L_2\leqslant_P L_3$，则有 $L_1\leqslant_P L_3$。

34.3-3　证明：$L\leqslant_P \bar{L}$，当且仅当 $\bar{L}\leqslant_P L$。

34.3-4　证明：在对引理 34.5 的另一种证明中，可满足性赋值可以当作证书来使用。哪一个证书可以使证明过程更容易些？

34.3-5　在引理 34.6 的证明中，假定算法 A 的工作存储占用的是一片具有多项式大小的连续存储空间。在该证明中的什么地方用到了这一假设？论证这一假设是不失一般性的。

34.3-6　相对于多项式时间的归约来说，一个语言 L 对语言类 C 是完全的，如果对所有 $L'\in C$，有 $L\in C$，且 $L'\leqslant_P L$。证明：相对于多项式时间的归约来说，ϕ 和 $\{0, 1\}^*$ 是 P 中仅有

的对 P 不完全的语言。

34.3-7　证明：L 对 NP 是完全的，当且仅当 $\bar{L}$ 对 co-NP 是完全的。 994

34.3-8　在引理 34.6 的证明中，归约算法 F 基于有关 x、A 和 k 的信息，构造了电路 $C=f(x)$。Sartre 教授观察到串 x 是 F 的输入，但只有 A、k 的存在性和运行时间 $O(n^k)$ 中所隐含的常数因子对 F 来说是已知的（因为语言 L 属于 NP），但其实际值对 F 来说却是未知的。因此，这位教授就得出了这样的结论，即 F 不可能构造出电路 C，且语言 CIRCUIT-SAT 不一定是 NP 难度的。说明在这位教授的推理中存在哪些问题。

34.4　NP 完全性的证明

电路可满足性问题的 NP 完全性依赖于直接证明对每一种语言 $L\in$ NP，有 $L\leqslant_P$CIRCUIT-SAT。在本节中，将说明如何在不把 NP 中的每一种语言直接归约为给定语言的前提下，证明一种语言是 NP 完全的。我们将通过证明各类公式可满足性问题是 NP 完全问题来说明这种方法。34.5 节中提供了用于说明这一方法的更多实例。

下面的引理是证明一种语言是 NP 完全语言的方法的基础。

引理 34.8　如果 L 是一种满足对某个 $L'\in$ NPC，有 $L'\leqslant_P L$ 的语言，则 L 是 NP 难度的。此外，如果 $L\in$ NP，则 $L\in$ NPC。

证明：由于 L' 是 NP 完全语言，所以对所有 $L''\in$ NP，有 $L''\leqslant_P L'$。根据假设，$L'\leqslant_P L$，因此根据传递性（练习 34.3-2），有 $L''\leqslant_P L$，这说明 L 是 NP 难度的。如果 $L\in$ NP，则也有 $L\in$ NPC。 ■

换句话说，通过把一个已知为 NP 完全的语言 L' 归约为 L，就可以把 NP 中的每一种语言都隐式地归约为 L。因此，引理 34.8 提供了证明某种语言 L 是 NP 完全语言的一种方法：

1）证明 $L\in$ NP。

2）选取一个已知的 NP 完全语言 L'。

3）描述一种算法来计算一个函数 f，它把 L' 中的每个实例 $x\in\{0,1\}^*$ 都映射为 L 中的一个实例 $f(x)$。

4）证明对所有 $x\in\{0,1\}^*$，函数 f 满足 $x\in L'$ 当且仅当 $f(x)\in L$。

5）证明计算函数 f 的算法具有多项式运行时间。 995

（第 2 步到第 5 步用于证明 L 是 NP 难度的。）这种根据一种已知的 NP 完全语言进行归约的方法，比说明如何直接根据 NP 中的每一种语言进行归约这一复杂的过程要简单得多。证明了 CIRCUIT-SAT$\in$NPC 已经使我们有了一个“立足点”。知道了电路可满足性问题是 NP 完全问题之后，再进一步证明其他问题是 NP 完全问题就要容易得多了。此外，当我们逐渐积累起一份已知 NP 完全问题的目录时，选择根据哪一种语言进行归约的余地就更大了。

公式可满足性

下面，对于确定布尔公式（而不是电路）是否是可满足这一问题，通过给出一个 NP 完全性证明，来说明上面提到的归约方法。在算法历史上，这个问题是第一个被证明为 NP 完全的。

（公式）可满足性问题可以根据语言 SAT 描述如下：SAT 的一个实例就是一个由下列成分组成的布尔公式 ϕ：

1. n 个布尔变量：x_1，x_2，…，x_n；

2. m 个布尔连接词：布尔连接词是任何具有一个或两个输入和一个输出的布尔函数，如 $\wedge$（与），$\vee$（或），$\neg$（非），$\rightarrow$（蕴含），$\leftrightarrow$（当且仅当）；

3. 括号。(不失一般性，假定没有冗余的括号；亦即，每个布尔连接词至多有一对括号)

很容易对一个布尔公式 ϕ 进行编码，使其长度为 $n+m$ 的多项式。如在布尔组合电路中一样，关于一个布尔公式 ϕ 的真值赋值是为 ϕ 中各变量所取的一组值；可满足性赋值是指使公式 ϕ 的值为 1 的真值赋值。具有可满足性赋值的公式就是可满足公式。可满足性问题提出了如下问题："一个给定的布尔公式是不是可满足的?"用形式语言的术语来表达，有：

$$\text{SAT} = \{\langle\phi\rangle : \phi \text{ 是一个可满足布尔公式}\}$$

例如，公式

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

具有可满足性赋值 $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$，这是因为

$$\begin{aligned}\phi &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 = (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 = 1\end{aligned} \tag{34.2}$$

996

因此，该公式 ϕ 属于 SAT。

确定一个任意的布尔公式是否是可满足的朴素算法不具有多项式运行时间。在一个具有 n 个变量的公式 ϕ 中，有 2^n 种可能的赋值。如果 $\langle\phi\rangle$ 的长度是关于 n 的多项式，则检查每一种可能的赋值需要 $\Omega(2^n)$ 时间，这是 $\langle\phi\rangle$ 长度的一个超多项式。正如下列定理所述，不太可能存在多项式时间的算法。

定理 34.9　布尔公式的可满足性问题是 NP 完全的。

证明：我们首先论证 SAT $\in$ NP，然后通过证明 CIRCUIT-SAT $\leqslant_P$ SAT，来证明 CIRCUIT-SAT 是 NP 难度的；根据引理 34.8 可知，这将证明定理成立。

为了证明 SAT 属于 NP，我们来证明对于输入公式 ϕ，由它的一个可满足性赋值所组成的证书可以在多项式时间内得到验证。验证算法将公式中的每个变量替换为其对应的值，再对表达式进行求值，这一做法与上面式(34.2)中的做法非常类似。这一任务很容易在多项式时间内完成。如果表达式的值为 1，则说明公式是可满足的。于是，引理 34.8 中有关 NP 完全性的第一个条件就成立了。

为了证明 SAT 是 NP 难度的，我们来证明 CIRCUIT-SAT $\leqslant_P$ SAT。换句话说，电路可满足性问题的任何实例可以在多项式时间内，归约为公式可满足性问题的一个实例。利用归纳法，可以将任意布尔组合电路表示为一个布尔公式。观察一下产生电路输出的逻辑门，并归纳地将每个逻辑门的输入表示为公式。于是，通过写出一个表达式，它将逻辑门的功能作用于其输入的公式，即可获得与电路对应的公式了。

遗憾的是，用这种直接的方法并不能构成一个多项式时间的归约过程。共享的子公式(它们源自于那些输出线的扇出为 2 或更多的逻辑门)会使得所生成的公式以指数的规模增长(参见练习 34.4-1)。因此，从某种意义上来说，我们必须采用更"聪明的"归约算法。

图 34-10 说明了在图 34-8a 中的电路上，由 CIRCUIT-SAT 归约为 SAT 的基本思想。对电路 C 中的每一根线 x_i，公式 ϕ 中都有一个变量 x_i。逻辑门的适当操作就可以表述为关于其附属线路变量的公式。例如，输出"与"门的操作为 $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$。

此归约算法产生的公式为 ϕ，它是电路输出变量与描述每个门的操作的子句合取式的"与"。

997　对图中的电路，相应的公式为：

$$\begin{aligned}\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_3) \\ &\wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\ &\wedge (x_6 \leftrightarrow \neg x_4)\end{aligned}$$

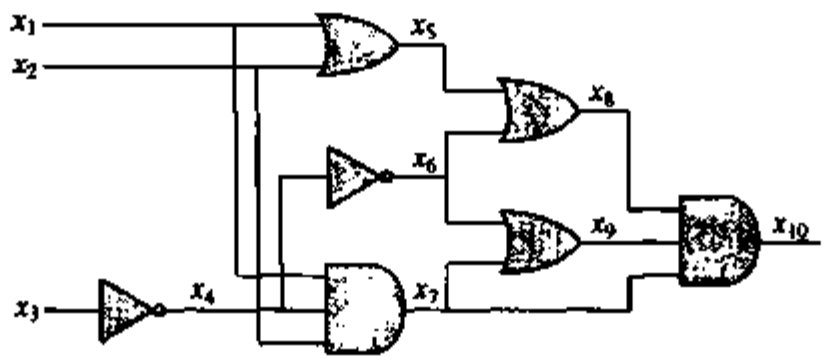


图 34-10　把电路可满足性归约为公式可满足性。在归约算法所产生的公式中，电路中的每根线都有着一个对应的变量

$$\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$$
$$\wedge (x_8 \leftrightarrow (x_5 \vee x_6))$$
$$\wedge (x_9 \leftrightarrow (x_6 \vee x_7))$$
$$\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

给定一个电路 C，就可以直接地在多项式时间内产生这样的一个公式 ϕ。

为什么只有当公式 ϕ 可满足时，电路 C 才是可满足的呢？如果 C 具有一个可满足性赋值，则电路的每条线路都有一个良定义的值，并且电路的输出为 1。因此，用线路的值对 ϕ 中的每个变量赋值后，就使得 ϕ 中每个子句的值为 1，因而，所有子句的合取值也为 1。反之，如果存在一个赋值 ϕ 的值为 1，则类似可证电路 C 是可满足的。这样，我们就已经证明了 CIRCUIT-SAT $\leqslant_P$SAT，这样就完成了整个证明过程。 ■

3-CNF 可满足性

根据公式可满足性进行归约，可以证明很多问题是 NP 完全问题。归约算法必须能够处理任何输入公式，但这样一来，就必须考虑大量的情况。因此，常常需要根据布尔公式的一种限制性语言来进行归约，使需要考虑的情况较少。当然，不能由于对该语言的限制过多，而使其成为多项式时间可解的语言。3-CNF 可满足性(或 3-CNF-SAT)就是这样一种方便的语言。 998

我们运用下列术语来定义 3-CNF 可满足性。布尔公式中的一个文字(literal)是指一个变量或变量的“非”。如果一个布尔公式可以表示为所有子句(clause)的“与”，且每个子句都是一个或多个文字的“或”，则称该布尔公式为合取范式，或 CNF(conjunctive normal form)。如果公式中每个子句恰好都有三个不同的文字，则称该布尔公式为 3 合取范式，或 3-CNF。

例如，布尔公式

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

就是一个 3 合取范式，其三个子句中的第一个为 $(x_1 \vee \neg x_1 \vee \neg x_2)$，它包含三个文字 x_1、$\neg x_1$ 和 $\neg x_2$。

在 3-CNF-SAT 中，有这样的问题：3-CNF 形式的一个给定布尔公式 ϕ 是否可满足？下列定理说明，即便当布尔公式表述为这种简单范式时，也不可能存在多项式时间的算法以确定其可满足性。

定理 34.10　3 合取范式形式的布尔公式的可满足性问题是 NP 完全的。

证明：我们在证明定理 34.9 时，为证明 SAT$\in$NP 所采用的证明方法同样也可以用来证明 3-CNF-SAT$\in$NP。于是，根据引理 34.8，我们仅需要证明 SAT$\leqslant_P$3-CNF-SAT。

归约算法可以分为三个基本步骤。每一步骤都逐渐使输入公式 ϕ 向所要求的 3 合取范式接近。

第一步类似于在定理 34.9 中用于证明 CIRCUIT-SAT$\leqslant_P$SAT 的过程。首先，为输入公式 ϕ 构造一棵二叉“语法分析”树，文字作为树叶，连接词作为内部顶点。图 34-11 说明了公式

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \tag{34.3}$$

的一棵语法分析树。如果输入公式中有包含数个文字的“或”的子句，就可以利用结合律对表达式加上括号，以使在所产生的树中的每一个内部顶点上均有 1 个或两个子女。现在，就可以把二叉语法分析树看作是计算该函数的一个电路了。

仿照定理 34.9 的证明中的归约过程，我们引入一个变量 y_i 作为每个内部顶点的输出。然后，把原始公式 ϕ 改写为根变量与描述每个顶点操作的子句的合取的“与”。公式(34.3)经改写后
999 所得的表达式为：

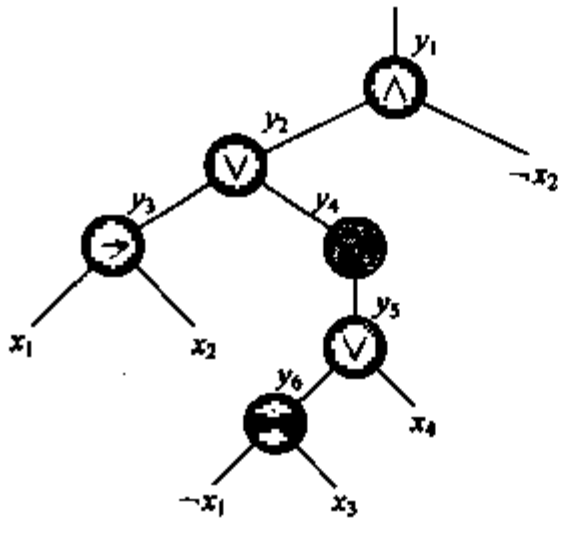
$$\begin{aligned}\phi' = {} & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_4 \leftrightarrow \neg y_5) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\end{aligned}$$

注意，这样得到的公式 ϕ' 是各子句 ϕ'_i 的合取式，每一个 ϕ'_i 至多有 3 个文字。此外，唯一的要求是每个子句都是文字的“或”。

归约的第二步是把每个子句 ϕ'_i 变换为合取范式。通过对 ϕ'_i 中变量的所有可能的赋值进行计算，可以构造出 ϕ'_i 的真值表。真值表中的每一行由子句变量的一种可能的赋值和根据这一赋值所计算出来的子句的值所组成。如果运用真值表中值为 0 的项，就可以构造出公式的析取范式(disjunctive normal form，DNF)，就是“与”的“或”，它等价于 $\neg\phi'_i$。然后，运用德·摩根定律(等式(B.2))把所有文字取补，并把“或”变成“与”、“与”变成“或”，就可以把公式变换为 CNF 公式 ϕ''_i。

在我们所举的例子中，按以下方式把子句 $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ 变换为 CNF。图 34-12 中给出了 ϕ'_1 的真值表，与 $\neg\phi'_1$ 等价的 DNF 公式为：

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$



1000 图 34-11　与公式 $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$ 对应的树

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

图 34-12　子句 $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ 的真值表

应用德·摩根定律，得到 CNF 公式：

$\phi_1''' = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$

它等价于原始子句 ϕ_1'。

现在，公式 ϕ' 的每个子句 ϕ_i' 已经被转换为一个 CNF 公式 ϕ_i''，因此，ϕ' 等价于由 ϕ_i'' 的合取式组成的 CNF 公式 ϕ''。此外，ϕ'' 的每个子句至多包含 3 个文字。

归约的第三步(也是最后一步)就是继续对公式进行变换，使每个子句恰好有三个不同的文字。最后的 3-CNF 公式 ϕ''' 是根据 CNF 公式 ϕ'' 的子句构造出来的，其中使用了两个辅助变量 p 和 q。对 ϕ'' 中的每个子句 C_i，使 ϕ''' 中包含下列子句：

- 如果 C_i 中有三个不同的文字，则直接把 C_i 作为 ϕ''' 中的一个子句；
- 如果 C_i 中有两个不同的文字，即，如果 $C_i = (l_1 \vee l_2)$，其中 l_1 和 l_2 为文字，则把 $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ 作为 ϕ''' 的子句。加入文字 p 和 $\neg p$ 仅仅是为了满足每个子句必须恰有三个不同的文字这一语法要求：不论 $p=0$ 或 $p=1$，$(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ 都等价于 $(l_1 \vee l_2)$。
- 如果 C_i 中有一个文字 l，则把 $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ 作为 ϕ''' 中的子句。注意 p 和 q 的每一种组合都使四个子句的合取式的值为 1。

现在，我们可以看出 3-CNF 公式 ϕ''' 是可满足的，当且仅当上述三个步骤的每一步中，ϕ 都是可满足的。像从 CIRCUIT-SAT 归约为 SAT 的过程一样，第一步根据 ϕ 构造 ϕ' 的过程保持可满足性。第二步产生的 CNF 公式 ϕ'' 在代数上与 ϕ' 等价。第三步产生的 3-CNF 公式 ϕ''' 也等价于 ϕ''，这是因为对变量 p 和 q 的任意赋值所产生的公式在代数上与 ϕ'' 等价。

1001

此外，还必须证明归约可以在多项式时间内完成。从 ϕ 构造 ϕ' 中的每个连接词至多引入一个变量和一个子句。从 ϕ' 构造 ϕ'' 的过程对 ϕ' 中的每一个子句至多在 ϕ'' 中引入 8 个子句，这是因为 ϕ' 中的每个子句至多有 3 个变量，因此每个子句的真值表中至多有 $2^3 = 8$ 行。从 ϕ'' 构造 ϕ''' 的过程对 ϕ'' 中的每个子句至多在 ϕ''' 中引入 4 个子句。因此，所产生的公式 ϕ''' 的规模是关于原始公式长度的多项式。我们可以容易地在多项式时间内完成每一步构造过程。■

练习

34.4-1 考虑一下在定理 34.9 的证明中的直接(非多项式时间)归约。描述一个规模为 n 的电路，当用这种归约方法将其转换为一个公式时，能产生出一个规模为 n 的指数的公式。

34.4-2 给出将定理 34.10 中的方法用于公式(34.3)时所得到的 3-CNF 公式。

34.4-3 Jagger 教授提出，在定理 34.10 的证明中，可以通过仅利用真值表技术(无需其他步骤)，就能证明 SAT$\leqslant_P$3-CNF-SAT。亦即，这位教授的意思是取布尔公式 ϕ，形成有关其变量的真值表，根据该真值表导出一个 3-DNF 形式的、等价于 $\neg\phi$ 的公式，再对其取反，并运用德·摩根定律，从而可以得到一个等价于 ϕ 的 3-CNF 公式。证明：这一策略不能产生多项式时间的归约。

34.4-4 证明：确定某一布尔公式是否是重言式这一问题对 co-NP 来说是完备的。(提示：见练习 34.3-7)。

34.4-5 证明：确定析取范式形式的布尔公式的可满足性这一问题是多项式时间可解的。

34.4-6 假设某人给出了一个判定公式可满足性的多项式时间算法。请说明如何利用这一算法在多项式时间内找出可满足性赋值。

1002

34.4-7 设 2-CNF-SAT 为 CNF 形式的、每个子句中恰有两个文字的可满足公式的集合。证明：

2-CNF-SAT∈P。你所给出的算法应尽可能地高效。（提示：注意 $x \vee y$ 与 $\neg x \rightarrow y$ 是等价的。将 2-CNF-SAT 归约为一个在有向图上高效可解的问题。）

34.5　NP 完全问题

NP 完全问题产生于各种不同领域：布尔逻辑，图论，算术，网络设计，集合与划分，存储与检索，排序与调度，数学程序设计，代数与数论，游戏与趣味难题，自动机与语言理论，程序优化，生物学，化学，物理，等等。在本节中，我们将运用归约方法，对从图论到集合划分的各种问题进行 NP 完全证明。

图 34-13 给出了在本节和 34.4 节中进行的 NP 完全证明的结构。图中每种语言的 NP 完全性都是根据对指向它的语言进行归约而证明的。其根为 CIRCUIT-SAT，我们在定理 34.7 中已经证明了它是 NP 完全语言。

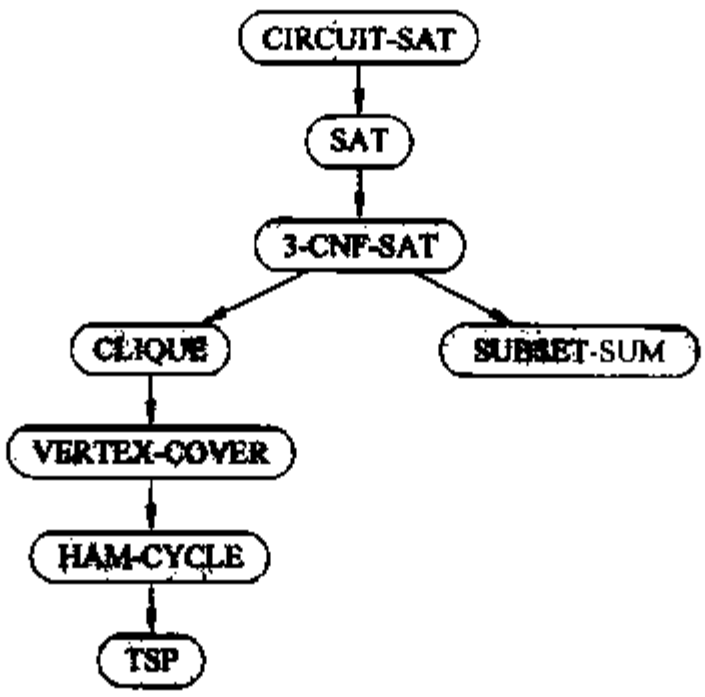


图 34-13　34.4 节和 34.5 节中 NP 完全性证明的结构。所有的证明最终都是通过对 CIRCUIT-SAT 的 NP 完全性的归约而得到的

34.5.1　团问题

无向图 $G=(V, E)$ 中的团（或团集）(clique) 是一个顶点子集 $V' \subseteq V$，其中每一对顶点之间都由 E 中的一条边相连。换句话说，一个团是 G 的一个完全子图。团的规模是指它所包含的顶点数。团问题是关于寻找图中规模最大的团的最优化问题。作为判定问题时，我们仅仅问这样的一个问题：在图中是否存在一个给定规模为 k 的团？其形式定义为：

$$\text{CLIQUE} = \{\langle G, k\rangle : G \text{ 是具有规模为 } k \text{ 的团的图}\}$$

要确定具有 $|V|$ 个顶点的图 $G=(V, E)$ 是否包含一个规模为 k 的团，一种朴素的算法是列出 V 的所有 k 子集，并对其中的每一个进行检查，看它是否形成了一个团。该算法的运行时间为 $\Omega\left(k^2\binom{|V|}{k}\right)$，如果 k 为常数，则它是多项式时间。但是，在一般情况下，k 可能接近于 $|V|/2$，这样一来，算法的运行时间就是超多项式时间。因此，人们猜想不大可能存在关于团问题的有效算法。

定理 34.11　团问题是 NP 完全的。

证明：为了证明 CLIQUE∈NP，对给定的图 $G=(V, E)$，用团中顶点集 $V' \subseteq V$ 作为 G 的证

书。对每一对顶点 u，$v \in V'$，通过检查边(u, v)是否属于 E，就可以在多项式时间内，完成对 V'是否是团的检查。

下一步来证明 3-CNF-SAT$\leqslant_P$CLIQUE，以此来说明团问题是 NP 难度的。从某种意义上来说，我们能证明这一结论是令人惊奇的，因为从表面上看、逻辑公式与图似乎没有什么联系。

从 3-CNF-SAT 的一个实例开始说明归约算法。设 $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ 是一个具有 k 个子句的 3-CNF 形式的布尔公式。对 $r = 1, 2, \cdots, k$，每个子句 C_r 恰好有三个不同的文字 l_1^r、l_2^r 和 l_3^r。我们将构造一个图 G，使得 ϕ 是可满足的，当且仅当 G 包含一个规模为 k 的团。

图 $G = (V, E)$构造如下，对 ϕ 中的每个子句 $Cr = (l_1^r \vee l_2^r \vee l_3^r)$，我们把三个顶点 v_1^r、v_2^r 和 v_3^r 组成的三元组放入 V 中。如果下列两个条件同时满足，就用一条边连接两个顶点 v_i^r 和 v_j^s。

- v_i^r 和 v_j^s 处于不同的三元组中，即 $r \neq s$。
- 它们的相应文字是一致的，即 l_i^r 不是 l_j^s 的非。

根据 ϕ 可以很容易地在多项式时间内计算出该图。通过以下实例来说明这一构造过程。如果有：

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

则 G 就是图 34-14 所示的图。

1003
~
1004

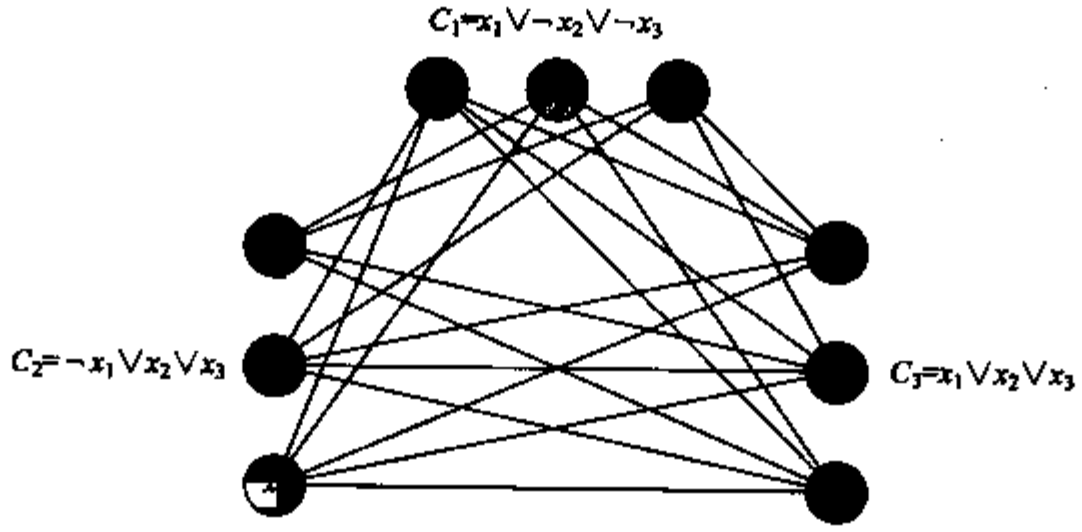


图 34-14　在由 3-CNF-SAT 归约至 CLIQUE 的过程中，由 3-CNF 公式 $\phi = C_1 \wedge C_2 \wedge C_3$ 导出的图 G，其中 $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$，$C_2 = (\neg x_1 \vee x_2 \vee x_3)$，$C_3 = (x_1 \vee x_2 \vee x_3)$。该公式的一个可满足赋值为 $x_2 = 0$，$x_3 = 1$，x_1 可以是 0 或 1。这一赋值以 $\neg x_2$ 满足 C_1，以 x_3 满足 C_2 和 C_3，与浅阴影顶点所构成的团集对应

必须证明这一从 ϕ 到 G 的变换是一种归约过程。首先，假定 ϕ 有一个可满足性赋值，那么，每个子句 C_r 中至少有一个文字 l_i^r 被赋值为 1，并且，每个这样的文字对应于一个顶点 v_i^r。从每个子句中挑选出一个这样的“真”文字，就得到 k 个顶点组成的集合 V'。可以断言，V'是一个团。对任意两个顶点 v_i^r，$v_j^s \in V'(r \neq s)$，根据给定的可满足性赋值，两个顶点相应的文字 l_i^r 和 l_j^s 都被映射为 1，这两个文字不可能是互补的关系。因此，由 G 的构造过程可知，边(v_i^r, v_j^s)属于 E。

反之，假定 G 有一个规模为 k 的团 V'。在 G 中没有连接同一个三元组中顶点的边，因此，V'包含每个三元组中的一个顶点。我们可以对每个满足 $v_i^r \in V'$的相应文字 l_i^r 赋值为 1，而不必担心会出现对一个文字与其补同时赋值为 1 的情形，这是因为在 G 中，没有连接不一致的文字的边。由于每个子句都是可满足的，所以 ϕ 也是可满足的(不与团中的顶点相对应的任何变量可以任意设置)。(证毕) ■

在图 34-14 所示的例子中，ϕ 的一个可满足性赋值为 $x_2 = 0$，$x_3 = 1$，规模 $k = 3$ 的相应团集

由对应于第一个子句中的$\neg x_2$、第二个子句中的x_3和第三个子句中的x_3的顶点所组成。由于该
1005 团不包含对应于x_1或$\neg x_1$的顶点，因此，在这个可满足性赋值中，可以将x_1设置为0或1。

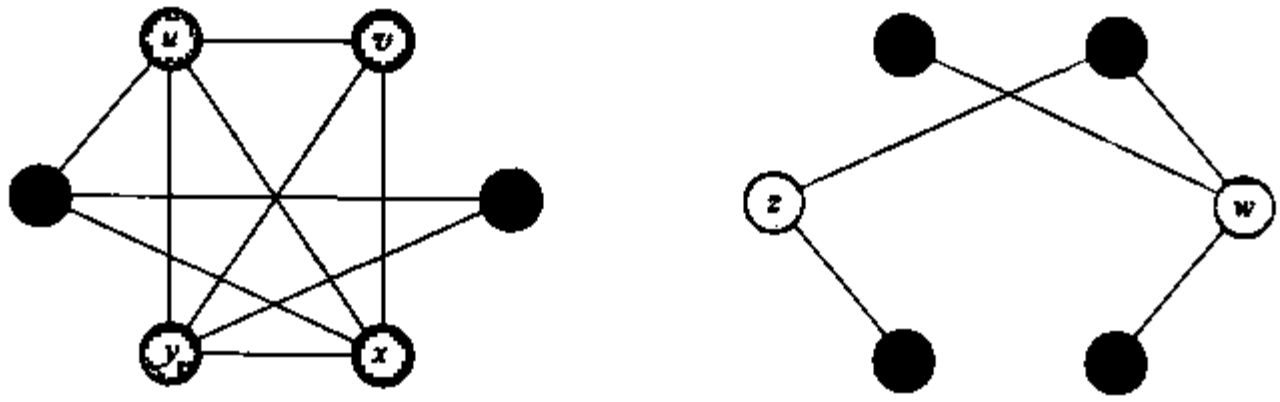
注意在定理34.11的证明中，我们将3-CNF-SAT的任意一个实例归约成了带某种特定结构的CLIQUE的实例。看上去，似乎是我们仅证明了CLIQUE在有些图中是NP难度的，在这些图中，顶点被限制为以三元组形式出现，且同一元组的顶点之间没有边。我们的确仅证明了CLIQUE在这种受限的情况下才是NP难度的，但是，这一证明足以证明在一般的图中，CLIQUE也是NP难度的。这是为什么呢？如果我们有一个多项式时间的算法，它能在一般的图上解决CLIQUE问题，那么它就能在受限的图上解决这一问题。

然而，将带有某种特殊结构的3-CNF-SAT的实例归约为通用的CLIQUE实例还不够。为什么这么说呢？有可能出现这样的情况，选择来进行归约的3-CNF-SAT的实例比较"容易"，因而无法将一个NP难度的问题归约为CLIQUE。

另外，还要注意一下3-CNF-SAT的实例中所用到的归约，而不仅是它的解决方案。如果认为多项式时间的归约的前提是已经知道公式ϕ是否是可满足的话，则会导致错误，因为我们并不知道该如何在多项式时间内来确定这一信息。

34.5.2　顶点覆盖问题

无向图$G=(V, E)$的*顶点覆盖*(vertex cover)是指子集$V' \subseteq V$，满足如果$(u, v) \in E$，则$u \in V'$或$v \in V'$(或两者成立)。亦即，每个顶点"覆盖"与其关联的边。G的顶点覆盖是覆盖E中所有边的顶点组成的集合。顶点覆盖的规模是指它所包含的顶点数目。例如，图34-15b中的图有一个规模为2的顶点覆盖$\{w, z\}$。



a)　　b)

图34-15　把CLIQUE归约为VERTEX-COVER。a)一个包含团$V'=\{u, v, x, y\}$的无向图$G=(V, E)$。b)由归约算法所产生的图$\bar{G}$，其中包含顶点覆盖$V-V'=\{w, z\}$

顶点覆盖问题(vertex cover problem)是指在给定的图中，找出具有最小规模的顶点覆盖。把这一最优化问题重新表述为判定问题，即确定一个图是否具有一个给定规模k的顶点覆盖。作为一种语言，我们定义

VERTEX-COVER$=\{\langle G, k\rangle$：图G具有规模为k的顶点覆盖$\}$

下面的定理说明了该问题是一个NP完全问题。

定理34.12　顶点覆盖问题是NP完全的。

证明：先来证明VERTEX-COVER$\in$NP。假定已知一个图$G=(V, E)$和整数k，我们选取的证书是顶点覆盖$V' \subseteq V$自身。验证算法证实$|V'|=k$，然后对每条边$(u, v) \in E$，检查是否有$u \in V'$或$v \in V'$。这一验证可以简单地在多项式时间内进行。

我们通过说明CLIQUE$\leqslant_P$VERTEX-COVER来证明顶点覆盖问题是NP难度的。这一归约

过程是以图的“补图”概念为基础的。已知一个无向图 $G=(V, E)$，定义 G 的补图 $\overline{G}=(V, \overline{E})$，其中 $\overline{E}=\{(u, v): u, v\in V, u\neq v, 且(u, v)\notin E\}$。换句话说，$\overline{G}$ 是包含不在 G 中的那些边的图。图 34-15 显示出了一个图与其补图，并说明了从 CLIQUE 到 VERTEX-COVER 的归约过程。

归约算法的输入是团问题的实例(G, k)。它计算出补图 $\overline{G}$，这很容易在多项式时间内完成。归约算法的输出是顶点覆盖问题的实例$(\overline{G}, |V|-k)$。为了完成证明，下面来说明该变换的确是一个归约过程：图 G 具有一个规模为 k 的团，当且仅当图 $\overline{G}$ 有一个规模为 $|V|-k$ 的顶点覆盖。

假设 G 包含一个团 $V'\subseteq V$，且 $|V'|=k$。我们断言，$V-V'$ 是 $\overline{G}$ 中的一个顶点覆盖。设 (u, v) 是 $\overline{E}$ 中的任意边，则有 $(u, v)\notin E$，这说明 u 或 v 至少有一个不属于 V'，因为 V' 中的每一对顶点间都有一条 E 中的边相连。等价地，u 或 v 至少有一个属于 $V-V'$，这意味着边 (u, v) 被 $V-V'$ 所覆盖。由于 (u, v) 是从 $\overline{E}$ 中任意选取的边，所以 $\overline{E}$ 的每条边都被 $V-V'$ 中的一个顶点所覆盖。因此，规模为 $|V|-k$ 的集合 $V-V'$ 形成了 $\overline{G}$ 的一个顶点覆盖。

反之，假设 $\overline{G}$ 具有一个顶点覆盖 $V'\subseteq V$，其中 $V'=|V|-k$。那么，对所有 $u, v\in V$，如果 $(u, v)\in\overline{E}$，则 $u\in V'$ 或 $v\in V'$ 或两者都成立。与此相对，对所有 $u, v\in V$，如果 $u\notin V'$ 且 $v\notin V'$，则有 $(u, v)\in E$。换句话说，$V-V'$ 是一个团，其规模为 $|V|-|V'|=k$。（证毕） ■

1006
~
1007

由于 VERTEX-COVER 是 NP 完全的，所以我们并不期望能找出一种多项式时间的算法，来寻找最小规模的顶点覆盖。不过，35.1 节中介绍了一种多项式时间的“近似算法”，它可以产生顶点覆盖问题的“近似”解。该算法所产生的顶点覆盖的规模至多为顶点覆盖最小规模的两倍。

因此，不能因为某个问题是 NP 完全的，就对它彻底放弃希望了。对这样的问题，尽管寻找其最优解是 NP 完全问题，但依然可能存在某种多项式时间的近似算法，来获得这一问题的近似最优解。第 35 章介绍了几个 NP 完全问题的近似算法。

34.5.3　哈密顿回路问题

现在，我们再回过头来讨论一下 34.2 节中定义的哈密顿回路问题。

定理 34.13　哈密顿回路问题是 NP 完全问题。

证明：我们先来说明 HAM-CYCLE 属于 NP。已知一个图 $G=(V, E)$，我们选取的证书是形成哈密顿回路的 $|V|$ 个顶点组成的序列。验证算法检查这一序列恰好包含 V 中每个顶点一次（只有第一个顶点在末尾重复出现一次），并且它们在 G 中形成一个回路。亦即，它要检查每一对连续顶点及首、尾顶点之间是否都存在着一条边。该验证算法可以在多项式时间内执行。

现在，我们来证明 VERTEX-COVER$\leqslant_P$HAM-CYCLE，从而证明 HAM-CYCLE 是 NP 完全的。给定一个无向图 $G=(V, E)$和一个整数 k，构造一个无向图 $G'=(V', E')$，使得它包含一个哈密顿回路，当且仅当 G 中有一个大小为 k 的顶点覆盖。

构造过程是基于附件图(widget)的，它是一个图的一部分，往往加上了某些特性。图 34-16a 中示出了我们用到的附件图。对于每条边 $(u, v)\in E$，我们所构造的图 G' 都将包含这一附件图的一份拷贝，用 W_{uv} 来表示。对 W_{uv} 中的每个顶点，用$[u, v, i]$或$[v, u, i]$来表示，其中 $1\leqslant i\leqslant 6$，这样，每个 W_{uv} 包含 12 个顶点。附件图 W_{uv} 还包含了图 34-16a 中所示的 14 条边。

除了附件图的内部结构外，我们还通过限制附件图与构造出来的图 G' 其余部分之间的连接，来强加一些有用的特性。具体来说，只有顶点$[u, v, 1]$、$[u, v, 6]$、$[v, u, 1]$和$[v, u, 6]$有边与 W_{uv} 的外界相连。G' 中的任何哈密顿回路都必须以图 34-16b-d 中所示三种方法中的某一种，来遍历 W_{uv} 中的边。如果回路由顶点$[u, v, 1]$进入，则必定由顶点$[v, u, 6]$退出，且它或者访问附件图中的 12 个顶点（图 34-16b），或者访问从$[u, v, 1]$到$[v, u, 6]$的 6 个顶点（图 34-16c）。 1008

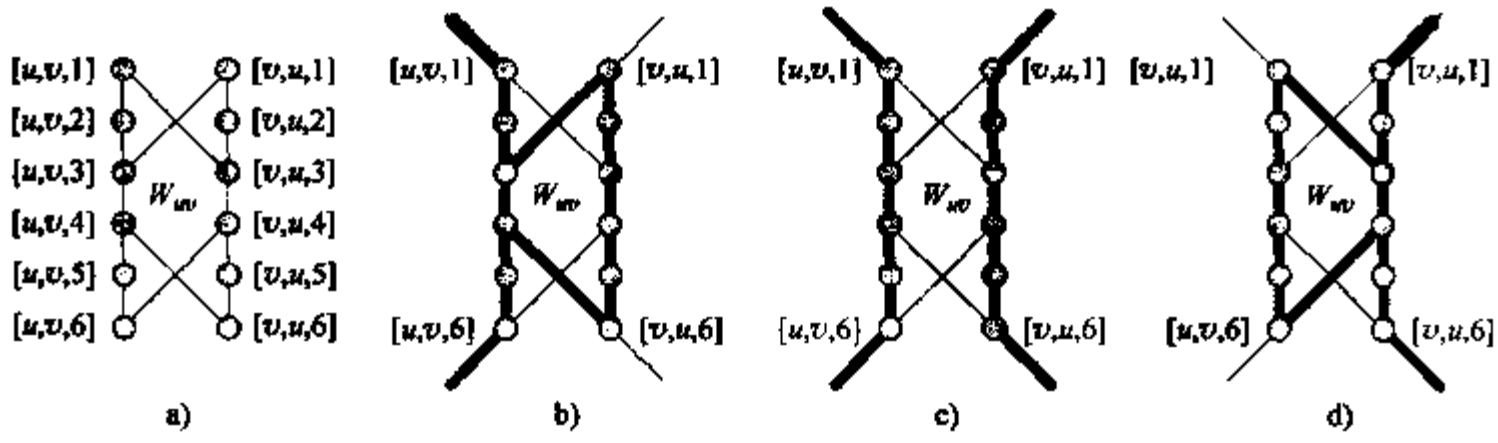


图 34-16　在将顶点覆盖问题归约为哈密顿回路问题的过程中所用到的附件图。图 G 的一条边 (u, v)对应于归约过程中所生成的图 G' 中的附件 W_{uv}。a)附件图，其中各个顶点都加上了标记。b)～d) 加了阴影的路径是通过附件图的、包含了所有顶点的仅有的可能路径。假定从该附件图至 G' 的其余部分的仅有连接是通过顶点$[u, v, 1]$、$[u, v, 6]$、$[v, u, 1]$和$[v, u, 6]$完成的

在后一情况中，回路必须重新进入附件图，以便访问顶点$[u, v, 1]$到$[v, u, 6]$。类似地，如果回路是从顶点$[v, u, 1]$进入的，则必须从顶点$[v, u, 6]$退出，且它或者访问附件图中的所有12个顶点(图 34-16d)，或者访问从$[v, u, 1]$到$[v, u, 6]$的6个顶点(图 34-16c)。除此以外，不可能有其他的、能访问附件图中所有 12 个顶点的路径了。特别地，不可能构造出一个两顶点的不相交路径，其中一条连接了$[u, v, 1]$与$[v, u, 6]$，另一条连接了$[v, u, 1]$和$[u, v, 6]$，使得两条路径的并包含了附件图中的所有顶点。

除了附件图中的那些顶点外，V' 中仅有的其他顶点为选择器顶点(selector vertex) s_1，s_2，…、s_k。我们利用与 G' 中选择器顶点关联的边，来选择 G 中的那些能实现 k 个顶点覆盖的顶点。

除了附件图中的边之外，E' 中还有另外两类边，如图 34-17 中所示。首先，对每个顶点 $u \in V$，都加入一些边来连接一对一对的附件图，从而形成一条路径，它包含了所有对应于 G 中与 u 关联的边的附件图。对于与每个顶点 $u \in V$ 相邻的所有顶点，将其任意地排序为 $u^{(1)}$，$u^{(2)}$，…，$u^{(\text{degree}(u))}$，其中 degree(u)是与 u 相邻的顶点的数目。通过将边$(([u, u^{(i)}, 6), [u, u^{(i+1)}, 1))$：$1 \leqslant i \leqslant \text{degree}(u)-1$)加入到 E' 中，即可在 G' 中生成一条穿越所有附件图的路径，这些附件图与那些关联于顶点 u 上的边对应。例如，在图 34-17 中，我们将与顶点 w 相邻的顶点排序为 x、y、z，这样图 34-17b 中的图 G' 就包含了边$([w, x, 6], [w, y, 1])$和$([w, y, 6], [w, z, 1])$。对于每个顶点 $u \in V$，G' 中的这些边形成了一条路径，它包含了一系列的附件图，这些附件图都与 G 中关联于顶点 u 上的边对应。

这些边给我们的直觉就是，如果选择了 G 的顶点覆盖中的某个顶点 $u \in V$，就可以在 G' 中构造出一条从$[u, u^{(1)}, 1]$到$[u, u^{(\text{degree}(u))}, 6]$的路径，它“覆盖”了所有与关联于顶点 u 的边对应的附件图。亦即，对于这些附件图中的每一个(如 $W_{u,u^{(i)}}$)，该路径或者包含所有的 12 个顶点(如果 u 在顶点覆盖中，但 $u^{(i)}$ 不在顶点覆盖中)，或者只是 6 个顶点$[u, u^{(i)}, 1]$，$[u, u^{(i)}, 2]$，…，
1009 $[u, u^{(i)}, 6]$(如果 u 和 $u^{(i)}$ 都在顶点覆盖中)。

E' 中的最后一类边将这些路径中的第一个顶点$[u, u^{(1)}, 1]$及最后一个顶点$[u, u^{(\text{degree}(u))}, 6]$与每一个选择器顶点联接起来。亦即，包含了以下的边：

$$\{(s_j, [u, u^{(1)}, 1]) : u \in V \text{ 和 } 1 \leqslant j \leqslant k\} \cup \{(s_j, [u, u^{(\text{degree}(u))}, 6]\} : u \in V \text{ 和 } 1 \leqslant j \leqslant k\}$$

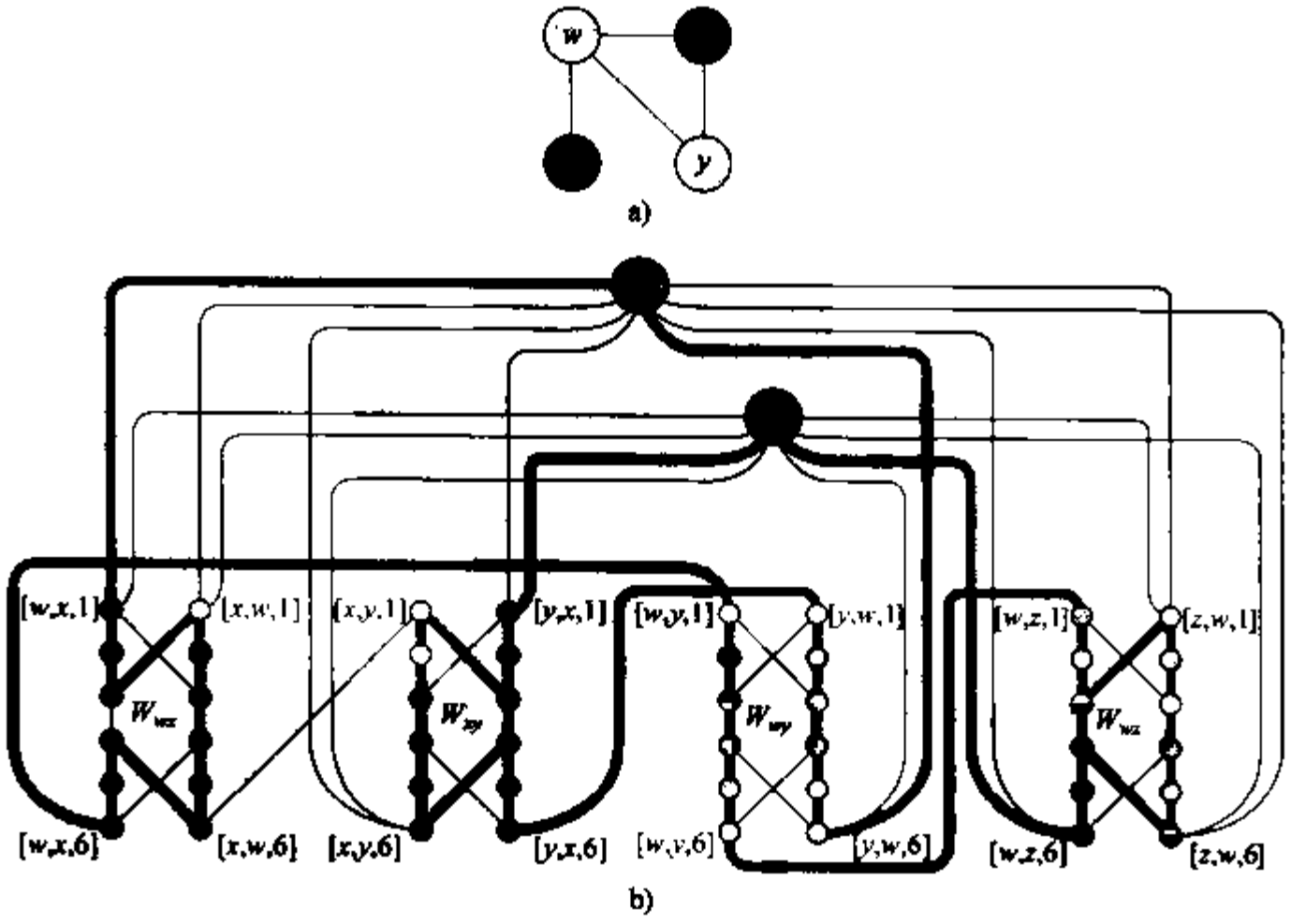


图 34-17　顶点覆盖问题的一个实例向哈密顿回路问题的一个实例的归约过程。a) 一个无向图 G，它有一个规模为 2 的顶点覆盖，由图中浅阴影的顶点 w 和 y 所组成。b)归约过程所产生的无向图 G'，其中与顶点覆盖对应的哈密顿回路以阴影示出。顶点覆盖 $\{w, y\}$ 对应于出现在哈密顿回路中的边 $(s_1, [w, x, 1])$ 和 $(s_2, [y, x, 1])$

接着，我们要证明 G' 的大小是 G 的大小的多项式，因而可以在 G 的大小的多项式时间内构造出 G'。G' 的顶点由附件图中的顶点及选择器顶点所构成。每一个附件图包含 12 个顶点，另外共有 $k \leqslant |V|$ 个选择器顶点，因此，共有：

$$|V'| = 12|E| + k \leqslant 12|E| + |V| \qquad (1010)$$

个顶点。G' 中的边包括附件图中的边、连接不同附件图的边和连接选择器顶点与附件图的边。每一个附件图中共有 14 条边，所有附件图加起来有 $14|E|$ 条边。对于每一个顶点 $u \in V$，在附件图之间有 $\text{degree}(u) - 1$ 条边，于是，对 V 中所有的顶点求和，则附件图之间共有 $\sum_{u \in V} (\text{degree}(u) - 1) = 2|E| - |V|$ 条边。最后，每一对附件图之间有两条边，它们由一个选择器顶点和 V 中的一个顶点所构成，共有 $2k|V|$ 条这样的边。因此，G' 中边的总数为：

$$\begin{aligned} |E'| &= (14|E|) + (2|E| - |V|) + (2k|V|) = 16|E| + (2k - 1)|V| \\ &\leqslant 16|E| + (2|V| - 1)|V| \end{aligned}$$

现在，我们来证明从 G 到 G' 的变换是一个归约。亦即，我们必须证明 G 中有一个规模为 k 的顶点覆盖，当且仅当 G' 中有一个哈密顿回路。

假设 $G = (V, E)$ 中有一个规模为 k 的顶点覆盖 $V^* \subseteq V$。设 $V^* = \{u_1, u_2, \cdots, u_k\}$。如图 34-17 中所示，通过为每个顶点 $u_j \in V^*$ 包含以下边[⊖]，即在 G 中形成了一个哈密顿回路。包

⊖ 从技术上来说，是根据顶点而不是根据边来定义回路的(见 B.4 节)。出于清晰性的考虑，此处"误用"了这种定义，即根据边来定义哈密顿回路。

含边$\{([u_j, u_j^{(i)}, 6], [u_j, u_j^{(i+1)}, 1]): 1 \leqslant i \leqslant \text{degree}(u_j)\}$，它们连接了所有与关联于$u_j$的边对应的附件图。此外，还要包含如图 34-16b-d 所示的附件图中的边，具体取决于某条边是否被V^*中的一个或两个顶点所覆盖。哈密顿回路还包含了边

$$\{(s_j, [u_j, u_j^{(1)}, 1]) : 1 \leqslant j \leqslant k\}$$
$$\cup \{(s_{j+1}, [u_j, u_j^{\{\text{degree}(u_j)\}}, 6] : 1 \leqslant j \leqslant k-1)\}$$
$$\cup \{(s_1, [u_k, u_k^{\{\text{degree}(u_k)\}}, 6])\}$$

只要仔细看一看图 34-17，就可以验证这些边确实形成了一个回路，从s_1开始，访问与所有关联于u_1的边对应的附件图，再访问s_2，访问与所有关联于u_2的边对应的附件图，等等，直到它返回s_1时为止。每个附件图都被访问了一次或两次，具体取决于V^*中的一个还是两个顶点覆盖了其对应的边。因为V^*是图G的一个顶点覆盖，因此，E中的每一条边都与V^*中的某个顶点关联，故回路访问要访问G'中每个附件图中的每一个顶点。因为该回路还要访问每一个选择
1011 器顶点，因此，它是哈密顿回路。

反之，假设$G'=(V', E')$中包含了一个哈密顿回路$C \subseteq E'$。我们断言集合

$$V^* = \{u \in V : (s_j, [u, u^{(1)}, 1]) \in C, \text{对于 } 1 \leqslant j \leqslant k\} \tag{34.4}$$

是G的一个顶点覆盖。为了说明为什么是成立的，可以将C划分为一些最大路径，它们从某个选择器顶点s_i开始，遍历一条边$(s_i, [u, u^{(1)}, 1])$(对某个$u \in V$)，再终止于某个选择器顶点s_j，而不会经过任何其他的选择器顶点。我们称每一条这样的路径为"覆盖路径"。根据G'的构造方式，每一条覆盖路径都必须从某个顶点s_i开始，对某个顶点$u \in V$取边$(s_i, [u, u^{(1)}, 1])$，再经过所有与E中关联于u的边对应的附件图，最后终止于某个选择器顶点s_j。称这一覆盖路径为p_u，根据式(34.4)，我们将u放入V^*。p_u所访问的每个附件图都必定是W_{uv}或W_{vu}，$v \in V$。对于p_u所访问的每个附件图，其顶点都会被一条或两条覆盖路径所访问。如果被一条覆盖路径所访问，则边$(u, v) \in E$在G中就由顶点u所覆盖。如果有两条覆盖路径访问了该附件图，则另一条覆盖路径必定为p_v，这就蕴含着$v \in V^*$，因而边$(u, v) \in E$被顶点u和v所覆盖。因为每一个附件图中的每一个顶点都要被某条覆盖路径所访问，因此可以看出，E中的每一条边都由V^*中的某个顶点所覆盖。(证毕) ■

34.5.4　旅行商问题

旅行商问题(traveling-salesman problem)与哈密顿问题有着密切的联系。在该问题中，一个售货员必须访问n个城市。如果把该问题模型化为一个具有n个顶点的完全图，就可以说这个售货员希望进行一次巡回旅行，或经过哈密顿回路，恰好访问每个城市一次，并最终回到出发的城市。从城市i到城市j的旅行费用为一个整数$c(i, j)$，这个售货员希望使整个旅行的费用最低，而所需的全部费用是他旅行经过的各边费用之和。例如，在图 34-18 中，费用最低的旅行线路是$\langle u, w, v, x, u \rangle$，其费用为 7。与旅行商问题对应的判定问题的形式语言是：

TSP=$\{\langle G, c, k \rangle : G=(V, E)$是一个完全图，$c$是$V \times V \to \mathbf{Z}$上的一个函数，$k \in \mathbf{Z}$且$G$包含一个费用至多为$k$的旅行商的旅行回路$\}$

下面的定理说明不大可能存在一种关于旅行商问题的快速算法。

1012 **定理 34.14**　旅行商问题是 NP 完全的。

证明：首先来说明 TSP 属于 NP。给定该问题的一个实例，用回路中n个顶点组成的序列作为证书。验证算法检查该序列是否恰好包含每个顶点一次，并且对边的费用求和后，检查和是否至多为k。当

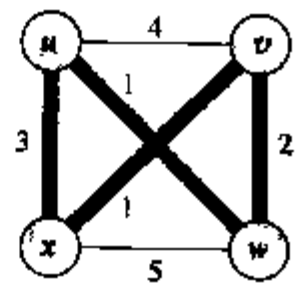


图 34-18　旅行商问题的一个例子。阴影边表示一条最小代价的旅行线路，其费用为 7

然，可以在多项式时间内完成这一过程。

为了证明 TSP 是 NP 难度的，我们来证明 HAM-CYCLE$\leqslant_P$TSP。设 $G=(V, E)$是 HAM CYCLE 的一个实例。构造 TSP 的实例如下。建立一个完全图 $G'=(V, E')$，其中 $E'=\{(i, j): i, j\in V$ 且 $i\neq j\}$，定义费用函数 c 为：

$$c(i,j)=\begin{cases}0 & \text{如果}(i,j)\in E\\ 1 & \text{如果}(i,j)\notin E\end{cases}$$

(注意因为 G 是无向图，它没有自环路，因而对所有顶点 $v\in V$，都有 $c(v, v)=1$。)于是，$(G', c, 0)$就是 TSP 的一个实例，它很容易在多项式时间内产生。

现在来说明图 G 中具有一个哈密顿回路，当且仅当图 G'中有一个费用至多为 0 的回路。假定图 G 中有一个哈密顿回路 h。h 中的每条边都属于 E，因此在 G'中的费用为 0。因此，h 在 G'中是费用为 0 的回路。反之，假定图 G'中有一个费用 h'至多为 0 的回路。由于 E'中边的费用只能是 0 或 1，故回路 h'的费用就是，且回路上每条边的费用必为 0。因此，h'仅包含 E 中的边。这样，我们就得出结论，h'是图 G 中的一个哈密顿回路。(证毕) ■

34.5.5　子集和问题

我们要考虑的下一个 NP 完全问题是一个算术问题，即子集和问题(subset-sum problem)。在子集和问题中，已知一个有限集 $S\subset$N 和一个目标 $t\in$N，问题是是否存在一个子集 $S'\subseteq S$，其元素和为 t。例如，如果 S=$\{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16\,808, 17\,206, 117\,705, 117\,993\}$，$t=138\,457$，则子集 $S'=\{1, 2, 7, 98, 343, 686, 2409, 17\,206, 117\,705\}$就是该问题的一个解。

和通常一样，我们把该问题定义为一种语言：

1013

$$\text{SUBSET-SUM}=\left\{\langle S,t\rangle: \text{存在一个子集 } S'\subseteq S, \text{满足 } t=\sum_{s\in S'}s\right\}$$

与任何算术问题一样，重要的是记住在标准编码中，假定输入的整数都是以二进制形式编码的。在这个假设下，可以证明对于子集和问题，不太可能存在一种快速的算法。

定理 34.15　子集和问题是 NP 完全的。

证明：为了说明 SUBSET-SUM 属于 NP，对该问题的实例$\langle S, t\rangle$，设子集 S'是证书。利用某一验证算法，就可以在多项式时间内完成是否有 $t=\sum_{s\in S'}s$ 的检查。

现在来证明 3-CNF-SAT$\leqslant_P$SUBSET-SUM。给定变量 x_1，x_2，…，x_n 上的一个 3-CNF 公式 ϕ，它由子句 C_1，C_2，…，C_k 构成，每个子句恰好包含 3 个不同的文字，归约算法构造出子集和问题的一个实例$\langle S, t\rangle$，使得 ϕ 是可满足的，当且仅当存在着 S 的一个子集，其元素和恰为 t。不失一般性，下面对 ϕ 做两个简化性的假设。首先，ϕ 的每一个子句都不会既包含某个变量，又包含该变量的非，因为这样的子句对变量的任何赋值来说，都是永真的。其次，每一个变量都出现在至少一个子句中，因为否则的话，对没有出现在任何子句中的变量赋任意值都是没有影响的。

对每个变量 x_i，归约算法都要在 S 中生成两个数；对每个子句 C_j，也要在 S 中生成两个数。所生成的数都是 10 进制的，且每个数都包含 $n+k$ 个数位，每个数位与一个变量或一个子句对应。10 进制(或其他进制)有着我们所需要的、可以避免从低位向高位进位的性质。

如图 34 19 中所示，我们如下构造集合 S 和目标 t。对于每一个数位，都用一个变量或一个子句来标记。最低有效 k 位用子句标记，最高有效 n 位用变量标记。

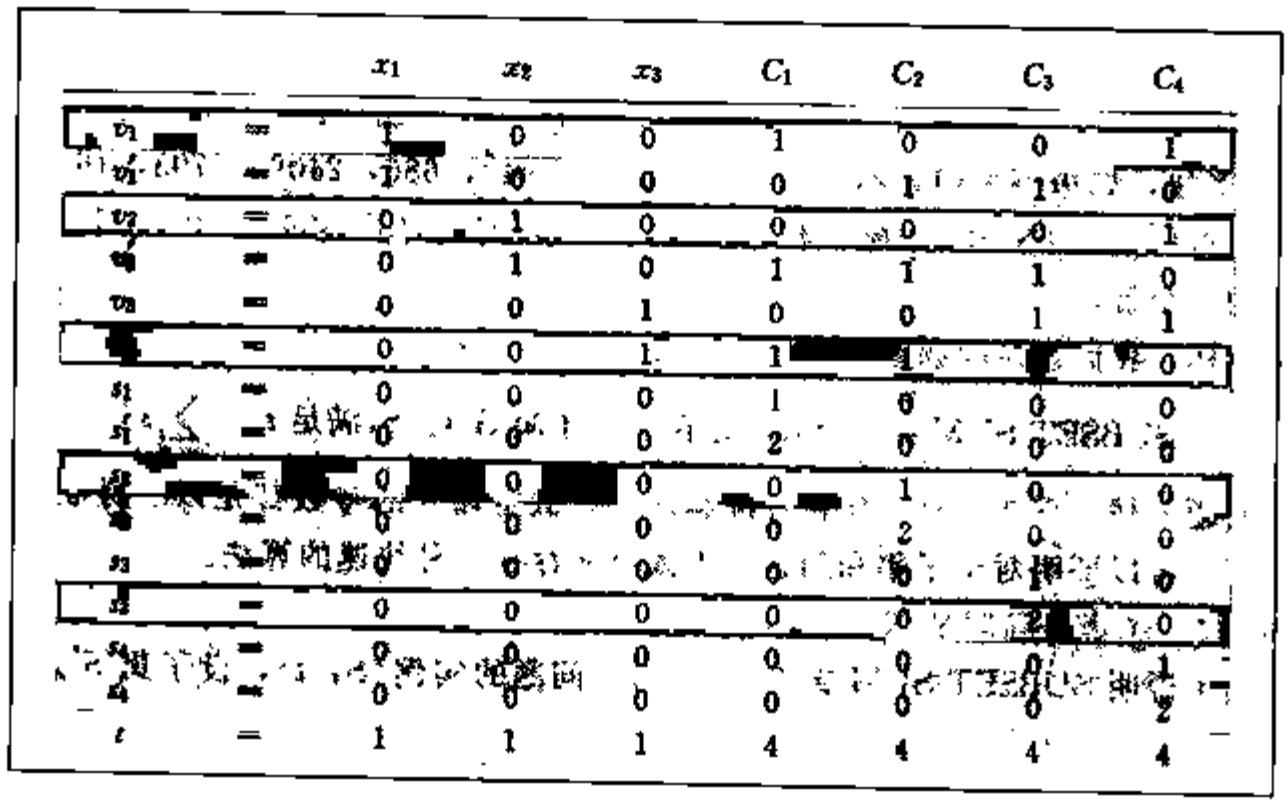
- 目标 t 在每个用变量标记的数位上都有个 1，在每个用子句标记的数位上都有个 4。

- 对每个变量 x_i，在 S 中都对应有两个整数 v_i 和 v'_i。在其中的每一个数中，由 x_i 标记的数位上为 1，由其他变量标记的数位上为 0。如果文字 x_i 出现于子句 C_j 中，则在变量 v_i 中，由 C_j 标记的数位上为 1。如果文字 $\neg x_i$ 出现于子句 C_j 中，则在变量 v'_i 中，由 C_j 标记的数位上为 1。在 v_i 和 v'_i 中，所有其他由子句标记的数位上均为 0。

在集合 S 中，所有 v_i 和 v'_i 值都是唯一的。这是为什么呢？对 $l \neq i$，在最高有效的 n 个数位上，没有 v_l 或 v'_l 的值会与 v_i 和 v'_i 相等。此外，根据上面所做的简化性假设，任意两个变量 v_i 和 v'_i 在最低有效 k 位上都不会完全相等。如果 v_i 和 v'_i 是相等的，则 x_i 和 $\neg x_i$ 将不得不出现于同一
1014 组子句中。但是，我们又假定任何一个子句中都不能同时包含 x_i 和 $\neg x_i$，且假定或者是 x_i、或者是 $\neg x_i$ 必须出现在某个子句中，因而，必定有某个子句 C_j，其中 v_i 和 v'_i 是不同的。

- 对每个子句 C_j，在 S 中都有两个对应的整数 s_j 和 s'_j。在这两个数中，除了由 C_j 所标记的数位外，其他的数位上都是 0。对 s_j 来说，在 C_j 数位上为 1，而对 s'_j 来说，在这个数位上则为 2。这些整数是“松弛变量”，可以用来获取每个子句所标记的数位，从而可以将它们加到目标值 4 中。

只要简单地看一看图 34-19，就会发现所有的 s_j 和 s'_j 值在集合 S 中都是唯一的。



	x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1 =	1	0	0	1	0	0	1
v'_1 =	1	0	0	0	1	1	0
v_2 =	0	1	0	0	0	0	1
v'_2 =	0	1	0	1	1	1	0
v_3 =	0	0	1	0	0	1	1
v'_3 =	0	0	1	1	1	0	0
s_1 =	0	0	0	1	0	0	0
s'_1 =	0	0	0	2	0	0	0
s_2 =	0	0	0	0	1	0	0
s'_2 =	0	0	0	0	2	0	0
s_3 =	0	0	0	0	0	1	0
s'_3 =	0	0	0	0	0	2	0
s_4 =	0	0	0	0	0	0	1
s'_4 =	0	0	0	0	0	0	2
t =	1	1	1	4	4	4	4

图 34-19　从 3-CNF-SAT 到 SUBSET-SUM 的归约。3-CNF 形式的公式为 $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$，其中 $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$，$C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$，$C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$，$C_4 = (x_1 \vee x_2 \vee x_3)$。$\phi$ 的一个可满足性赋值为 $(x_1 = 0, x_2 = 0, x_3 = 1)$。归约过程所产生的集合 S 中包含了一些 10 进制数字：按照从上往下看的顺序，$S = \{1\,001\,001, 1\,000\,110, 100\,001, 101\,110, 10\,011, 11\,100, 1\,000, 2\,000, 100, 200, 10, 20, 1, 2\}$。目标 t 为 1 114 444。子集 $S' \subseteq S$ 以浅阴影示出，它包含了 v'_1、v'_2 和 v_3，与可满足性赋值对应。它还包含了松弛变量 s_1，s'_1，s'_2，s_3，s_4 和 s'_4，以便在由 C_1 到 C_4 所标记的数位上达到目标值 4

注意在任何一个数位上，最大的数位和为 6，这个和值出现在由子句所标记的数位上（来自
1015 v_i 和 v'_i 值的 3 个 1，加上来自 s_j 和 s'_j 值的 1 和 2）。于是，按照 10 进制来对这些数进行解释，就不会出现由低位向高位的进位。⊖

⊖ 事实上，任何满足 $b \geqslant 7$ 的进制 b 都是可以的。这一小节开头给出的实例是如图 34-19 所示的集合 S 和目标 t，它们是按 7 进制来解释的，且 S 是按排序顺序列出的。

以上的归约过程可以在多项式时间内完成。集合 S 中包含了 $2n+2k$ 个值，其中每一个值都有 $n+k$ 个数位，产生每一个数位的时间都是 $n+k$ 的多项式。目标 t 有 $n+k$ 个数位，其中的每一个数位都可以由归约过程在常量时间内产生。

现在，我们来证明 3-CNF 形式的公式 ϕ 是可满足的，当且仅当存在着一个子集 $S' \subseteq S$，其元素之和为 t。首先，假设 ϕ 有一个可满足性赋值。对 $i=1, 2, \cdots, n$，如果在此赋值中有 $x_i=1$，就将 v_i 包含在 S' 中。否则，包含 v'_i。换句话说，我们是把在可满足性赋值中，与值为 1 的文字对应的 v_i 和 v'_i 值包含在了 S' 中。在对所有的 i 包含了 v_i 或 v'_i（但不是同时包含两者）之后，并且将由所有 s_j 和 s'_j 中的变量所标记的数位置 0 后，可以看出，对于每个由变量标记的数位，S' 中元素之和必为 1，这与目标 t 中的那些数位正好是匹配的。因为每个子句是可满足的，故子句中必有某个文字的值为 1。于是，由一个子句所标记的每个数位都至少有一个 1，从而可以在 S' 元素的和值中，贡献出一个 v_i 或 v'_i。事实上，在每个子句中，可能有 1、2 或 3 个文字的值为 1，因此，根据 S' 中所包含的 v_i 或 v'_i 值的情况，每一个由子句所标记的数位的和为 1、2 或 3。（例如，在图 34-19 中，在某一可满足性赋值中，文字 $\neg x_1$、$\neg x_2$ 和 x_3 的值为 1。子句 C_1 和 C_4 都恰包含了这三个文字中的一个，因此，v'_1、v'_2 和 v_3 合起来，为 C_1 和 C_4 中数位的和值贡献了一个 1。子句 C_2 包含了这三个文字中的两个，因而，v'_1、v'_2 和 v_3 合起来，为 C_2 中数位的和值贡献了一个 2。子句 C_3 包含所有这三个文字，因而，v'_1、v'_2 和 v_3 合起来，为 C_3 中数位的和值贡献了一个 3。）在由子句 C_j 所标记的每个数位中，通过将松弛变量 $\{s_j, s'_j\}$ 的一个非空子集包含进 S'，即可达到目标值 4。（在图 34-19 中，S' 包含了 s_1，s'_1，s'_2，s_3，s_4 和 s'_4。）由于我们已经在和值的所有数位中匹配了目标值，且不会发生进位，因而，S' 中元素的和值为 t。

现在，假设有一个子集 $S' \subseteq S$，其元素之和为 t。对每一个 $i=1, 2, \cdots, n$，子集 S' 必定包含了 v_i 和 v'_i 两者中的一个，否则的话，变量所标记的数位加起来就不会为 1 了。如果 $v_i \in S'$，就置 $x_i=1$。否则，有 $v'_i \in S'$，就置 $x_i=0$。我们断言，对 $j=1, 2, \cdots, k$，每个子句 C_j 可以通过此赋值得到满足。为了证明这一断言，注意到为了在 C_j 标记的数位中达到和值 4，子集 S' 必须至少包含 v_i 或 v'_i 值中的一个，它们在 C_j 标记的数位上有个 1，因为松弛变量 s_j 和 s'_j 合起来所做的贡献至多为 3。如果 S' 包含了一个 v_i，它在该位置上有个 1，则文字 x_i 会出现在子句 C_j 中。因为当 $v_i \in S'$ 时，我们已经设置了 $x_i=1$，因而，子句 C_j 得到满足。如果 S' 包含了一个 v'_i，它在 1016
该位置上有个 1，则文字 $\neg x_i$ 会出现在子句 C_j 中。因为当 $v'_i \in S'$ 时，我们已经设置了 $x_i=0$，因而，子句 C_j 又得到满足。于是，ϕ 的所有子句都得到满足，这样就完成了整个证明。（证毕） ■

练习

34.5-1 子图同构问题（subgraph-isomorphism problem）取两个图 G_1 和 G_2，要回答 G_1 是否与 G_2 的一个子图同构这一问题。证明：子图同构问题是 NP 完全的。

34.5-2 给定一个 $m \times n$ 的矩阵 A 和一个整型的 m 维向量 b，0-1 整数规划问题（0-1 integer-programming problem）即是否有一个整型的 n 维向量 x，其元素取自集合 $\{0, 1\}$，满足 $Ax \leqslant b$。证明：0-1 整数规划问题是 NP 完全的。（提示：由 3-CNF-SAT 问题进行归约。）

34.5-3 整数线性规划问题（integer linear-programming problem）与练习 34.5-2 中给出的 0-1 整数规划问题类似，只是向量 x 的值可以取任何整数，而不仅是 0 或 1。假定 0-1 整数规划问题是 NP 难度的，证明整数线性规划问题是 NP 完全的。

34.5-4 证明：如果目标值 t 表示成一元形式，则子集和问题在多项式时间内可解。

34.5-5 集合划分问题（set-partition problem）的输入为一个数字集合 S。问题是这些数字是否能

被划分成两个集合 A 和 $\bar{A}=S-A$，使得 $\sum_{x\in A}x=\sum_{x\in\bar{A}}x$。证明：集合划分问题是 NP 完全的。

34.5-6　证明：哈密顿路径问题是 NP 完全的。

34.5-7　最长简单回路问题(longest-simple-cycle problem)是在一个图中，找出一个具有最大长度的简单回路(即其中没有重复的顶点)。证明：这个问题是 NP 完全的。

1017

34.5-8　在半 3-CNF 可满足性(half 3-CNF satisfiability problem)中，给定一个 3-CNF 形式的公式 ϕ，它包含 n 个变量和 m 个子句，其中 m 是偶数。我们希望确定是否存在着对 ϕ 中变量的一个真值赋值，使得恰有一半的子句为 0，恰有另一半的子句为 1。证明：半 3-CNF 可满足性问题是 NP 完全的。

思考题

34-1　独立集

图 $G=(V, E)$ 的独立集是子集 $V'\subseteq V$，使得 E 中的每条边至多与 V' 中的一个顶点相关联。独立集问题是要找出 G 中具有最大规模的独立集。

a)给出与独立集问题相关的判定问题的形式描述，并证明它是 NP 完全的。(提示：根据团问题进行归约。)

b)假设有一个"黑箱"子程序，它用于解决(a)中定义的判定问题。试写出一个算法，以找出规模最大的独立集。所给出的算法的运行时间应该是关于 $|V|$ 和 $|E|$ 的多项式，其中查询黑箱的工作被看作是一步操作。

尽管独立集判定问题是 NP 完全的，但在某些特殊情况下，该问题是多项式时间可解的。

c)当 G 中的每个顶点的度数均为 2 时，试写出一个有效的算法来求解独立集问题。分析算法的运行时间，并证明算法的正确性。

d)当 G 为二分图时，试写出一个有效的算法以求解独立集问题。分析算法的运行时间，并证明算法的正确性。(提示：利用第 26.3 节中的结论。)

34-2　Bonnie 和 Clyde

Bonnie 和 Clyde 刚刚抢劫了一家银行。他们抢劫到了一袋钱，并打算将钱分光。对于下面的每一种场景，给出一个多项式时间的算法，或者证明该问题是 NP 完全的。每一种情况下的输入是关于袋子里 n 件东西的一份清单，以及每一件东西的价值。

a)共有 n 个硬币，但只有两种不同的面值：一些面值 x 美元，一些面值 y 美元。他俩希望平分掉这笔钱。

1018

b)共有 n 个硬币，它们有着任意数量的不同面值，但每一种面值都是 2 的非负整数次幂，亦即，可能的面值为 1 美元、2 美元、4 美元，等等。他俩希望平分掉这笔钱。

c)共有 n 张支票，十分巧合的是，这些支票恰好是支付给"Bonnie 和 Clyde"的。他俩希望平分掉这些支票，从而可以分得同样数目的钱。

d)与 c)一样，共有 n 张支票，但这一次，他俩愿意接受这样的一种支票分配方案，即两人所分得的钱数差距不大于 100 美元。

34-3　图的着色

无向图 $G=(V, E)$ 的 k 着色(k-coloring)是一个函数 $c: V\to\{1, 2, \cdots, k\}$，对每条边 $(u, v)\in E$，有 $c(u)\neq c(v)$。换句话说，数 1，2，…，k 表示 k 种颜色，并且相邻顶点必须为不同的颜色。图的着色问题(graph-coloring problem)就是确定要对某个给定图着色所

必需的最少的颜色种类。

a)写出一个有效的算法，以找出一个图的 2 着色(如果存在的话)。

b)把图的着色问题描述为一个判定问题。证明：该判定问题在多项式时间内可解，当且仅当图的着色问题在多项式时间内可解。

c)设语言 3-COLOR 是能够进行三着色的图的集合。证明：如果 3-COLOR 是 NP 完全语言，则 b)中的判定问题是 NP 完全的。

为了证明 3-COLOR 具有 NP 完全性，我们根据 3-CNF-SAT 来进行归约。给定一个由 m 个子句组成的关于 n 个变量 x_1，x_2，…，x_n 的公式 ϕ，构造图 $G=(V, E)$ 如下。对每个变量和每个变量的"非"，集合 V 分别包含一个顶点，对每个子句，V 包含 5 个顶点。另外，V 中还有三个特殊的顶点：TRUE，FALSE 和 RED。图的边分为两种类型：与子句无关的"文字"边和依赖于子句的"子句"边。对 $i=1, 2, \cdots, n$，文字边形成一个由特殊顶点构成的三角形，并且还形成了一个由 x_i、$\neg x_i$ 和 RED 构成的三角形。 1019

d)论证在对包含文字边的图的任意一个 3 着色 c 中，一个变量和它的"非"中恰好有一个被着色为 c(TRUE)，另一个被着色为 c(FALSE)。论证对 ϕ 的任何真值赋值，对仅包含文字边的图都有一种 3 着色存在。

图 34-20 所示的附件图用于实现相应于子句 $(x \vee y \vee z)$ 的条件。每个子句都要求复制唯一的图中涂黑的 5 个顶点的一份拷贝；如图所示，它们把子句中的文字与特殊顶点 TRUE 相连。

e)论证如果 x、y 和 z 中每个顶点均着色为 c(TRUE)或 c(FALSE)，则该附件图是 3 着色的，当且仅当 x、y 和 z 中至少有一个被着色为 c(TRUE)。

f)完成 3-COLOR 是 NP 完全问题的证明。

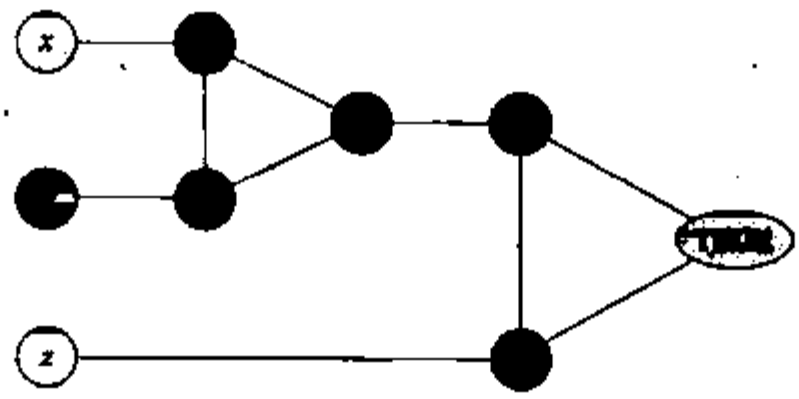


图 34-20 思考题 34-3 中用到的对应于子句 $(x \vee y \vee z)$ 的附件图

34-4 带收益和完工期限的调度

假设有一台机器和 n 项任务 a_1，a_2，…，a_n。每一项任务 a_j 都有着处理时间 t_j、利润 p_j 和完工期限 d_j。这台机器一次只能处理一项任务，而任务 a_j 必须不间断地运行 t_j 个连续的时间单位。如果能赶在期限 d_j 之前完成任务 a_j，就能获取利润 p_j，但是，如果是在期限到了之后完成任务的，就没有任何利润了。作为一个最优化问题，已知的是 n 项任务的处理时间、利润和完工期限，希望找出一种调度方案，以便完成所有的任务，并能获取最大的利润。

a)将这个问题表述为一个判定问题。

b)证明：此判定问题是 NP 完全的。

c)给出此判定问题的一个多项式时间算法，假定所有的处理时间都是 1 到 n 之间的整数。(提示：采用动态规划。) 1020

d)给出此最优化问题的一个多项式时间算法，假定所有的处理时间都是1到n之间的整数。

本章注记

Garey和Johnson撰写的书[110]中提供了有关NP完全性的很好的指南，详细地讨论了这一理论，并列出了一个目录，其中包括了许多在1979年时已知的NP完全问题。本章中定理34.13的证明就是改编自该书，34.5节开头给出的NP完全问题领域列表也是取自该书的目录。Johnson在1981年到1992年之间，在*Journal of Algorithms*上撰写了一系列(共23期)专栏文章，报告NP完全性方面的最新研究进展。Hopcraft，Motwani和Ullman[153]，Lewis和Papadimitriou[204]，Papadimitrou[236]，以及Sipser[279]在复杂性理论这一背景中，很好地处理了NP完全性问题。Aho，Hopcroft和Ullman[5]也涉及了NP完全性问题，并给出了几种归约，包括从哈密顿回路问题到顶点覆盖问题的归约。

P类是在1964年由Cobham[64]、1965年由Edmonds[84]独立地提出的，后者还提出了NP类，并推测有P≠NP。NP完全性概念是在1971年，由Cook[67]提出的，他给出了公式可满足性问题和3-CNF可满足性问题的第一个NP完全性证明。Levin[203]独立地提出了这一概念，并给出了tiling问题的NP完全性证明。Karp[173]在1972年提出了归约方法，并总结了各种NP完全问题。在Karp的论文中，给出了对团问题、顶点覆盖问题、哈密顿回路问题的NP完全性的原创性证明。自那以后，许多研究人员证明了数以百计的问题都是NP完全问题。在1995年庆祝Karp 60岁生日的聚会上，Papadimitriou在发言中提到，“每年，在标题、摘要或关键词中有‘NP完全’这一字眼的论文有大约6000篇。这一数字比有关‘编译器’、‘数据库’、‘专家’、‘神经网络’或‘操作系统’这些术语中每一个的论文数量都要多。”

近来，复杂性理论方面的最新研究成果为计算近似解的复杂性问题带来了希望。这一方面的工作利用“概率意义下可检验的证明”这一概念，给出了NP的新定义。这一新的定义意味着对诸如团、顶点覆盖、旅行商等带有三角不等式的问题，还有许多其他的问题，计算有效的近似解决方案是NP难度的，因而并不比计算最优解更容易。有关这一领域的介绍可以参见Arora的论文[19]；Arora和Lund[149]中的一章；Arora[20]撰写的一篇综述性文章；一本由Mayr，
1021 Promel和Steger编辑的书[214]；以及Johnson的一篇综述性文章[167]。

第35章　近似算法

许多具有实际意义的问题都是NP完全问题，但都非常重要，所以不能仅因为获得其最优解的过程是难以驾驭的而放弃它们。如果一个问题是NP完全的，就不太可能找到一个能给出其准确解的多项式时间算法，但这并不意味着没有希望了。解决NP完全问题至少有三种方法：第一，如果实际输入的规模比较小，则用具有指数运行时间的算法来解决问题就很理想了。第二，或许能将一些重要的、多项式时间可解的特殊情况隔离出来；第三，仍有可能在多项式时间里找到(最坏情况或平均情况)近似最优解(near-optimal solution)。在实践中，近似最优解常常就足够好了。能返回近似最优解的算法称为近似算法(approximation algorithm)。本章就来介绍解决几个NP完全问题的多项式时间近似算法。

近似算法的性能比值

假定我们在解一个最优化问题，该问题的每一个可能解都有正的代价，我们希望找出一个近似最优解。根据所要解决的问题，最优解可以定义成具有最大可能代价的解或具有最小可能代价的解。就是说，该问题可能是一个求最大值的问题或求最小值的问题。

我们说问题的一个近似算法有着近似比(approximation ratio)$\rho(n)$，如果对规模为n的任何输入，由该近似算法产生的解的代价C与最优解的代价C^*只差一个因子$\rho(n)$：

$$\max\left(\frac{C}{C^*},\frac{C^*}{C}\right)\leqslant\rho(n) \tag{35.1}$$

我们也称一个能达到近似比$\rho(n)$的算法为$\rho(n)$近似算法。这个定义对求最大值和求最小值问题都适用。对于一个求最大值的问题，$0<C\leqslant C^*$，而比值C^*/C给出最优解的代价大于近似解的代价的倍数。类似地，对于一个求最小值的问题，$0<C^*\leqslant C$，比值C/C^*给出近似解的代价大于最优解的代价的倍数。因为我们假定所有解的代价都是正的，故这两种比值都是明确定义的。一个近似算法的近似比不会小于1，因为$C/C^*<1$蕴含着$C^*/C>1$。于是，一个1近似算法[⊖]产生的是最优解，而一个有着较大近似比的近似算法可能返回较最优解差很多的解。

对于很多问题来说，已经设计出具有较小的固定近似比的多项式时间近似算法；对于另一些问题来说，在其已知的最佳多项式时间的近似算法中，近似比是作为输入规模n的函数而增长的。35.3节将讨论的集合覆盖问题就是这样的一个问题。

一些NP完全问题允许有多项式时间的近似算法，通过消耗越来越多的计算时间，这些近似算法可以达到不断缩小的近似比。就是说，在计算时间和近似的质量之间可以进行权衡。这类问题的一个例子是35.5节要讨论的子集和问题。这类问题非常重要，值得专门进行研究。

一个最优化问题的近似方案(approximation scheme)是这样的一种近似算法，它的输入除了该问题的实例外，还有一个值$\epsilon>0$，使得对任何固定的ϵ，该方案是个$(1+\epsilon)$近似算法。对一个近似方案来说，如果对任何固定的$\epsilon>0$，该方案都以其输入实例的规模n的多项式时间运行，则称此方案为多项式时间近似方案。

随着ϵ的减小，多项式时间近似方案的运行时间会迅速增长。例如，一个多项式时间近似方案的运行时间可能达到$O(n^{2/\epsilon})$。在理想的情况下，如果ϵ按一个常数因子减小，为了获得希望的近似效果，所增加的运行时间不应超过一个常数因子。换句话说，我们希望运行时间既是

⊖　当近似比独立于n时，我们将使用"近似比ρ"和"ρ近似算法"等术语，以表示与n无关。

$1/\epsilon$ 的多项式，又是 n 的多项式。

对一个近似方案来说，如果其运行时间既为 $1/\epsilon$ 的多项式，又为输入实例的规模 n 的多项式，则称其为完全多项式时间的近似方案。例如，近似方案可能有 $O((1/\epsilon)^2 n^3)$ 运行时间。对于
1023 这样的一种方案，ϵ 的任意常数倍的减少可以由运行时间的相应常数倍增加来弥补。

本章内容的安排

本章的前 4 节介绍一些解决 NP 完全问题的多项式时间近似算法的例子，第 5 节给出一个完全多项式时间近似方案。35.1 节以对顶点覆盖问题的研究开始。这是一个 NP 完全的最小化问题，它有着一个近似比为 2 的近似算法。35.2 节给出了旅行商问题的一个近似比为 2 的近似算法。在这个问题中，代价函数满足三角不等式。这一节还证明如果没有三角不等式，对任意常数 $\rho \geqslant 1$，不可能存在 ρ 近似算法，除非 P＝NP。在 35.3 节里，说明对集合覆盖问题，如何将贪心方法作为有效的近似算法获得一个覆盖，其代价在最差情况下比最优代价大一个对数倍。35.4 节给出另外两个近似算法。首先研究 3-CNF 可满足性问题的最优化形式，并给出一个简单的随机化算法，它给出的解具有预期的近似比 8/7。接着，分析顶点覆盖问题的一个带权值的变形，并说明如何利用线性规划设计一个 2 近似算法。最后，35.5 节给出子集和问题的一个完全多项式时间的近似方案。

35.1　顶点覆盖问题

在 34.5.2 节中，顶点覆盖问题被定义和证明为 NP 完全的。无向图 $G=(V, E)$ 的一个顶点覆盖是一个子集 $V' \subseteq V$，使得如果 (u, v) 是 G 的一条边，则或者 $u \in V'$，或者 $v \in V'$（或者两者都成立）。一个顶点覆盖的规模即其中所包含的顶点数。

顶点覆盖问题要求在一个给定的无向图中，找出一个具有最小规模的顶点覆盖。我们称这样的一个顶点覆盖为一个最优顶点覆盖。这个问题是一个 NP 完全的判定问题的最优化形式。

虽然在一个图 G 中寻找最优顶点覆盖比较困难，但要找出一个近似最优的顶点覆盖不会太难。下面给出的近似算法以一个无向图 G 为输入，并返回一个其规模保证不超过最优顶点覆盖
1024 的规模两倍的顶点覆盖。

```
APPROX-VERTEX-COVER(G)
1  C ← ∅
2  E' ← E[G]
3  while E' ≠ ∅
4      do let(u, v) be an arbitrary edge of E'
5         C ← C ∪ {u, v}
6         remove from E' every edge incident on either u or v
7  return C
```

图 35-1 示出了 APPROX-VERTEX-COVER 的操作过程。变量 C 包含了正在被构造的顶点覆盖。第 1 行将 C 初始化为空集。第 2 行将 E' 置为图 G 的边集 $E[G]$ 的一个副本。第 3～6 行中的循环重复地从 E' 中选出一条边 (u, v)，将其端点 u 和 v 加入 C，并删去 E' 中所有被 u 或 v 覆盖的边。这个算法的运行时间为 $O(V+E)$，E' 以邻接表来表示。

定理 35.1　APPROX-VERTEX-COVER 有一个多项式时间的 2 近似算法。

证明：前面我们已经说明了 APPROX-VERTEX-COVER 的运行时间为多项式。

由 APPROX-VERTEX-COVER 返回的顶点集合 C 是个顶点覆盖，因为这个算法会一直循环，直到 $E[G]$ 中的每条边都被 C 中的某个顶点覆盖为止。

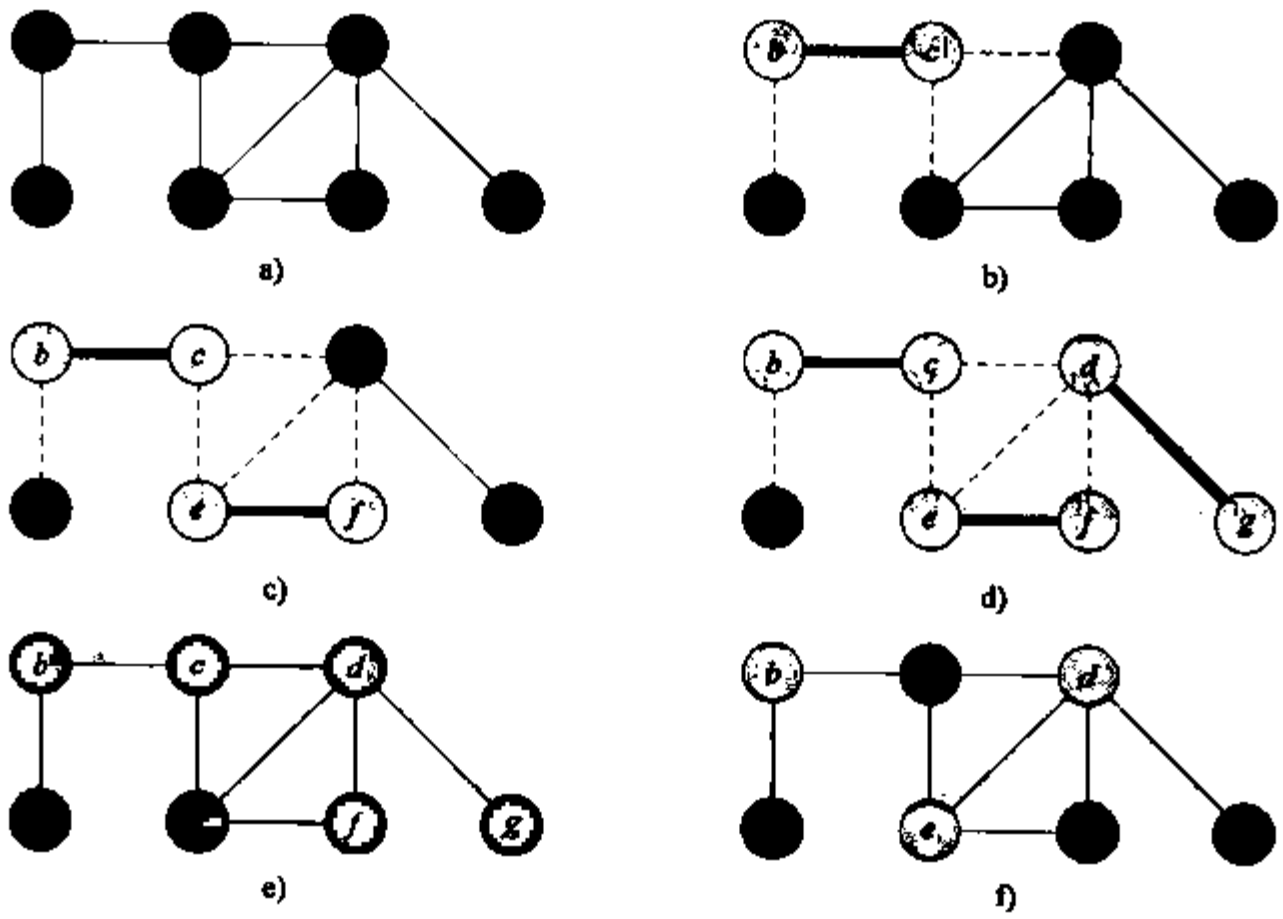


图 35-1　APPROX-VERTEX-COVER 的操作过程：a)具有 7 个顶点和 8 条边的输入图 G。b)以粗线示出的边(b, c)是被 APPROX-VERTEX-COVER 所选择的第一条边。加了浅阴影的顶点 b 和 c 被加入集合 C，它包含了正被构造的顶点覆盖。以虚线示出的边(a, b)、(c, e)和(c, d)被删除，因为现在它们由 C 中的某个顶点所覆盖。c)边(e, f)被选中；顶点 e 和 f 被加到 C 中。d)边(d, g)被选择中；顶点 d 和 g 被加到 C 中。e)集合 C 是由 APPROX-VERTEX-COVER 产生的顶点覆盖，它包含 6 个顶点 b、c、d、e、f、g。f)这个问题的最优顶点覆盖仅包含三个顶点：b，d 和 e

为了说明 APPROX-VERTEX-COVER 所返回的顶点覆盖的规模至多为最优覆盖的两倍，设 A 表示在 APPROX-VERTEX-COVER 的第 4 行中选出的边集。为了覆盖 A 中的边，任意一个顶点覆盖(尤其是最优覆盖 C^*)都必须包含 A 中每条边的至少一个端点。又因为 A 中没有两条边具有共同的端点，因为一旦一条边在第 4 行中被选出后，在第 6 行中就将所有与其端点关联的边从 E' 中去掉。所以，A 中不会有两条边由 C^* 中的同一顶点所覆盖，于是，最优顶点覆盖的规模有着如下的下界：

$$|C^*| \geqslant |A| \tag{35.2}$$

第 4 行的每一次执行都会都会挑选出一条边，其两个端点都不在 C 中，因此，所返回的顶点覆盖的规模上界(实际上，是一个上确界)为：

$$|C| = 2|A| \tag{35.3}$$

将式(35.2)和式(35.3)结合起来，有：

$$|C| = 2|A| \leqslant 2|C^*|$$

从而定理得证。(证毕) ■

我们再来回顾一下上述的证明过程。开始时，人们可能会想，如何才能证明 APPROX-VERTEX-COVER 所返回的顶点覆盖的规模至多为最优顶点覆盖规模的两倍，因为我们并不知道最优顶点覆盖的规模到底是多少。对于这个问题，答案就是我们利用了最优顶点覆盖规模的一个下界。练习 35.1-2 要求读者证明，在 APPROX-VERTEX-COVER 的第 4 行中，挑选出来的边集 A 实际上是图 G 的一个最大匹配。(所谓最大匹配是指这样的一种匹配，它不是任何其他匹配的真子集。)就像定理 35.1 的证明过程中所说的那样，一个最大匹配的大小是最优顶点覆盖大

1025
~
1026

小的下界。该算法返回的是一个顶点覆盖，其大小至多为最大匹配 A 大小的两倍。通过将算法返回的解的规模与下界进行比较，即可得到算法的近似比。后面的几节里，也将采用这一方法来获得近似算法的近似比。

练习

35.1-1 给出一个图的例子，使得 APPROX-VERTEX-COVER 对该图总是产生次最优解。

35.1-2 设 A 表示在 APPROX-VERTEX-COVER 的第 4 行中挑选出来的边集。证明：集合 A 是图 G 中的一个最大匹配。

***35.1-3** Nixon 教授提出了以下的启发式方法来解决顶点覆盖问题：重复地选择度数最高的顶点，并去掉所有邻接边。给出一个例子，说明这位教授的启发式方法达不到近似比 2。（提示：可以考虑一个二分图，其中左图中顶点的度数一样，而右图中顶点的度数不一样。）

35.1-4 给出一个有效的贪心算法，以便在线性时间内，找出一棵树的最优顶点覆盖。

35.1-5 从定理 34.12 的证明中，我们知道了顶点覆盖问题和 NP 完全团问题在某种意义上来说是互补的，即最优顶点覆盖是补图中某一最大规模团的补。这种关系是否意味着存在着一个多项式时间的近似算法，它对团问题有着固定的近似比？请给出你的回答，并对你的回答加以说明。

35.2　旅行商问题

在 34.5.4 节中引入的旅行商问题（traveling-salesman problem）中，给定的是一个完全的无向图 $G=(V, E)$，其中每条边 $(u, v)\in E$ 都有一个非负的整数代价 $c(u, v)$，我们希望找出 G 的一个具有最小代价的哈密顿回路。现在我们把前面所用的这些记号表示略作扩充，设 $c(A)$ 表示子集 $A\subseteq E$ 中所有边的总代价：

1027

$$c(A)=\sum_{(u,v)\in A} c(u,v)$$

在很多实际情况中，从一个地方 u 直接到另一个地方 w 总是代价最小的。经由任何一个中转站 v 的一种路径不可能具有更小的代价了。换种方式说，去掉一个中间站决不会增加代价。将这一概念加以形式化，即如果对所有的顶点 u，v，$w\in V$，有：

$$c(u,w)\leqslant c(u,v)+c(v,w)$$

就称代价函数 c 满足三角不等式。

三角不等式是个很自然的不等式，在许多应用中，它都能自动得到满足。例如，如果图的顶点为平面上的点，且在两个顶点间旅行的代价即为它们之间通常的欧几里得距离，就满足三角不等式。（除了欧几里得距离外，还有许多其他的代价函数能满足此三角不等式。）

如练习 35.2-2 所说明的那样，即使我们强行要求代价函数要满足三角不等式，也不能改变旅行商问题的 NP 完全性。因此，不可能找出一个准确解决这个问题的多项式时间算法，因而就要寻找一些好的近似算法。

在 35.2.1 节中要讨论一个 2 近似算法，用于解决符合三角不等式的旅行商问题。在 35.2.2 节中，要证明如果不符合三角不等式，则不存在具有常数近似比多项式时间的近似算法，除非 P=NP。

35.2.1　满足三角不等式的旅行商问题

利用前一小节的方法，我们首先要计算出一个结构（即最小生成树），其权值是最优旅行商路线长度的下界。接着，要利用这一最小生成树来生成一条遍历线路，其代价不大于最小生成树权

值的两倍，只要代价函数满足三角不等式即可。下面的算法实现了这一过程，其中要将 23.2 节中的最小生成树算法 MST-PRIM 作为子程序加以调用。

```
APPROX-TSP-TOUR(G, c)
1  select a vertex r∈V[G] to be a "root" vertex
2  compute a minimum spanning tree T for G from root r
       using MST-PRIM(G, c, r)
3  let L be the list of vertices visited in a preorder tree walk of T
4  return the hamiltonian cycle H that visits the vertices in the order L
```

1028

回忆我们在 12.1 节中说过，先序树遍历递归地访问树中的每个顶点，在第一次遇到某个顶点时(在访问其子女之前)就列出该顶点。

图 35-2 说明了 APPROX-TSP-TOUR 的操作过程，其中图 35-2a 示出了给定点的集合，图 35-2b 示出了一棵最小生成树 T，它是由 MST-PRIM 计算出来的，根为 a 顶点。图 35-2c 对 T 进行先序遍历时，各顶点的访问顺序。图 35-2d 显示了由 APPROX-TSP-TOUR 返回的对应游程。图 35-2e 示出了一个最优的游程，它比图 35-2d 中的游程要短约 23%。

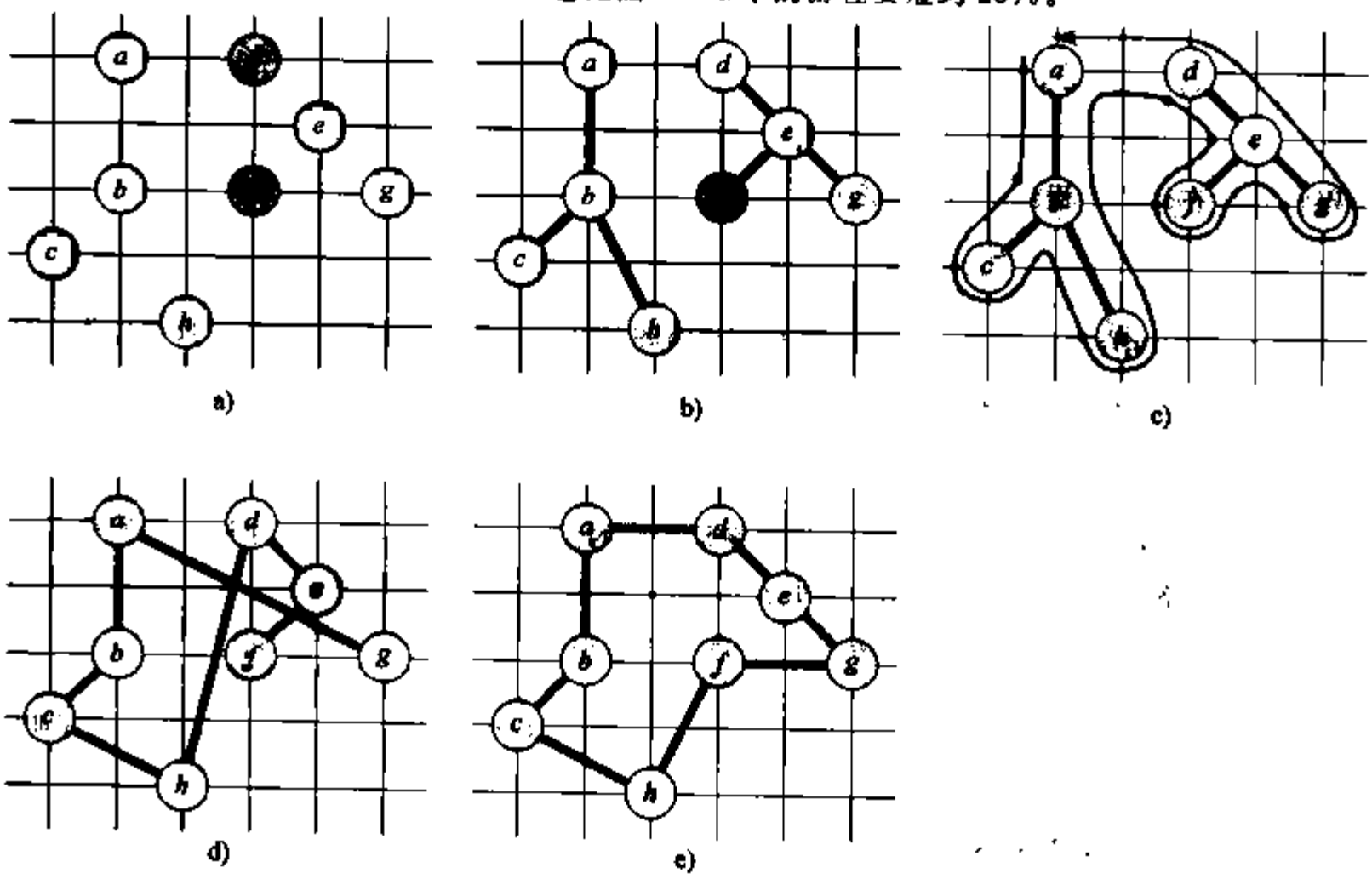


图 35-2　APPROX-TSP-TOUR 的操作过程。a)给定的点集，它们位于一个整数网格的各顶点上。例如，f 位于 h 右边一个单位、上边两个单位处。此处采用通常的欧几里得距离作为两点间的代价函数。b)在这些点的基础上，由 MST-PRIM 计算出来的一棵最小生成树 T。顶点 a 为根顶点。各顶点的标记方式恰好使得它们可以按字典顺序，由 MST-PRIM 加入到主树中。c)从 a 开始对 T 进行的遍历。对树的完整遍历将按 $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$ 的顺序访问各顶点。在对 T 进行先序遍历时，只在第一次遇到一个顶点时，才将该顶点列出，由每个顶点旁边的点表示，从而得到顺序 a, b, c, h, d, e, f, g。d)按先序遍历顺序访问各顶点时，所得到的顶点游程路线。这是由 APPROX-TSP-TOUR 返回的游程 H，其总代价约为 19.074。e)在给定顶点集后，一个最优的游程 H^*，其总代价约为 14.715

1029

根据练习 23.2-2，即使是采用 MST-PRIM 的简单实现，APPROX-TSP-TOUR 的运行时间也是 $\Theta(V^2)$。现在我们来证明：如果旅行商问题的某一实例的代价函数满足三角不等式，则 APPROX-TSP-TOUR 所返回的游程的代价不大于最优游程的代价的两倍。

定理 35.2　APPROX-TSP-TOUR 是一个解决满足三角不等式的旅行商问题的、多项式时间的 2 近似算法。

证明：前面已经证明了 APPROX-TSP-TOUR 的运行时间为多项式。

设 H^* 表示在给定顶点集合上的一个最优游程。因为我们是通过删除一个游程路线中的任一条边而得到一棵生成树的，故最小生成树 T 的权值是最优游程代价的一个下界，亦即：

$$c(T) \leqslant c(H^*) \tag{35.4}$$

对 T 的一个完全遍历在初次访问一个顶点，以及在从访问某个顶点的一棵子树返回后列出该顶点。我们称这个遍历为 W。对例子中的树进行完全遍历，就得到次序

$$a,b,c,b,h,b,a,d,e,f,e,g,e,d,a$$

因为该完全遍历恰经过了 T 的每条边两次，所以有(将代价 c 的定义加以自然的扩展，以处理多个边集的情况)：

$$c(W) = 2c(T) \tag{35.5}$$

式(35.4)和式(35.5)蕴含着

$$c(W) \leqslant 2c(H^*) \tag{35.6}$$

即 W 的代价在最优游程代价的两倍之内。

但是，W 一般来说不是一个游程，因为它对于某些顶点要访问一次以上。然而，根据三角不等式，可以从 W 中去掉一次对任意顶点的访问，代价并不会增加。(如果在对顶点 u 和 w 的访问之间，从 W 中去掉顶点 v，所得的游程顺序就指示了直接从 u 到 w。)反复应用这个操作，可以从 W 中，将对每个顶点的除第一次访问之外的其他各次访问去掉。在我们的例子中，这样的
1030 一个操作过程即可得游程次序：

$$a,b,c,h,d,e,f,g$$

这个次序与对树 T 做先序遍历所得的次序是一样的。设 H 为对应这个先序遍历的回路，它是个哈密顿回路，因为每个顶点仅被访问一次，并且它实际上是由 APPROX-TSP-TPUR 计算出来的回路。因为 H 是通过从完全遍历 W 中删除了某些顶点后得到的，故有：

$$c(H) \leqslant c(W) \tag{35.7}$$

将不等式(35.6)和不等式(35.7)合起来，就有 $c(H) \leqslant 2c(H^*)$，从而完成了对本定理的证明。(证毕) ■

尽管定理 35.2 给出了很好的近似比，但在实践中，APPROX-TSP-TOUR 通常并不是解决旅行商问题的最佳选择。另外有些近似算法的实际性能要比这个算法好得多，具体可见本章末的参考文献。

35.2.2　一般旅行商问题

如果我们去掉关于代价函数 c 满足三角不等式的假设，则不可能在多项式时间内找到好的近似游程路线，除非 P=NP。

定理 35.3　如果 P≠NP 则对任何常数 $\rho \geqslant 1$，一般旅行商问题不存在具有近似比 ρ 的多项式时间近似算法。

证明：用矛盾来证明。假设对某个数 $\rho \geqslant 1$，存在一个近似比为 ρ 的多项式时间近似算法 A。不失一般性，假定 ρ 是一个整数(必要的话，可以对其舍入)。我们来说明如何在多项式时间内，

用 A 来解决哈密顿回路问题(其定义见 34.2 节)的各种实例。根据定理 34.13 可知，哈密顿回路问题是 NP 完全的，因而根据定理 34.4，在多项式时间内解决这个问题就蕴含着 P=NP。

设 $G=(V, E)$为哈密顿回路问题的一个实例。我们希望通过利用假想的近似算法 A 来有效地确定 G 是否包含一个哈密顿回路。我们如下将 G 变为旅行商问题的一个实例。设 $G'=(V, E')$为 V 上的完全图，也就是说，

$$E' = \{(u,v): u,v \in V, \text{且 } u \neq v\}$$

再对 E'中的每条边如下地赋以一个整数代价：

$$c(u,v) = \begin{cases} 1 & \text{如果}(u,v) \in E \\ \rho \mid V \mid + 1 & \text{否则} \end{cases}$$

1031

G'和 c 的表示可以在 $|V|$ 和 $|E|$ 的多项式时间内由 G 的表示构造出来。

现在来考虑旅行商问题(G', c)。如果原图 G 中存在一条哈密顿回路 H，则代价函数 c 对 H 中的每条边赋以代价 1，因而(G', c)中包含了一个代价为 $|V|$ 的游程。另一方面，如果 G 中不包含一条哈密顿回路，那么 G'的任意一个游程必定要用到不在 E 中的某条边。但是，任意一个用到不在 E 中的边的游程的代价至少为

$$(\rho \mid V \mid + 1) + (\mid V \mid - 1) = \rho \mid V \mid + \mid V \mid > \rho \mid V \mid$$

因为不在 G 中的边的代价如此之大，故 G 中哈密顿回路的游程代价(为 $|V|$)与任何其他游程的代价(至少为 $\rho|V|+|V|$)之间相差至少为 $|V|$。

如果我们对旅行商问题(G', c)应用近似算法 A 将会怎样呢？因为 A 能保证使其返回的游程的代价不大于一个最优游程代价的 ρ 倍，如果 G 包含一个哈密顿回路，则 A 必定会返回它。如果 G 不包含哈密顿回路，则 A 就会返回一个代价大于 $\rho|V|$ 的游程。所以，可以用 A 来在多项式时间内，解决哈密顿回路问题。(证毕) ■

定理 35.3 的证明过程演示了一种通用的技术，这种技术可以用来证明某一问题无法被很好地近似。假设给定一个 NP 难度的问题 X，我们可以构造出一个最小化问题 Y，使得 X 的“yes”实例对应于 Y 的、值至多为 k(对于某个 k)的实例，而 X 的“no”实例对应于 Y 的、值大于 ρk 的实例。那么，我们就证明了除非有 P=NP，否则，问题 Y 不存在 ρ 近似算法。

练习

35.2-1 假设有一个完全无向图 $G=(V, E)$，含有至少三个顶点，其代价函数 c 满足三角不等式。证明：对所有的 u，$v \in V$，有 $c(u, v) \geqslant 0$。

35.2-2 说明如何才能在多项式时间内，将旅行商问题的一个实例转换为另一个其代价函数满足三角不等式的实例。两个实例必须有同一组最优游程。请解释为什么这样的一种多项式时间的转换与定理 35.3 并不矛盾，假设 P≠NP。

1032

35.2-3 考虑以下的用于构造近似旅行商游程的*最近点启发式*：从只包含任意选择的某一顶点的平凡回路开始。在每一步中，找出一个顶点 u，它不在回路中，但与回路上任何顶点之间的距离最短。假设回路上距离 u 最近的顶点为 v。将回路加以扩展以包含顶点 u，即将 u 插入在 v 之后。重复这一过程，直到所有的顶点都在回路上时为止。证明：这一启发式返回的游程总代价不大于最优游程代价的两倍。

35.2-4 *瓶颈旅行商问题*中，要找出这样的一条哈密顿回路，使得回路中代价最大的边的代价最小。假设代价函数满足三角不等式，证明：这个问题存在着一个近似比为 3 的多项式时间近似算法。(提示：采用递归证明的方法，证明如问题 23-3 中讨论的那样，通过完

全遍历瓶颈生成树及跳过某些顶点，可以恰访问树中的每个顶点一次，但连续跳过的中间顶点不会多于两个。证明在瓶颈生成树中，代价最大的边的代价至多为瓶颈哈密顿回路中代价最大的边的代价。）

35.2-5 假设与旅行商问题的一个实例对应的顶点是平面上的点，且代价 $c(u, v)$ 是点 u 和 v 之间的欧几里得距离。证明：一条最优游程不会自我交叉。

35.3 集合覆盖问题

集合覆盖问题是一个最优化问题，它模型化了许多资源选择问题，其相应的判定问题推广了 NP 完全的顶点覆盖问题，因而也是 NP 难度的。然而，用于解决顶点覆盖的近似算法在这儿就用不上了，需要尝试其他的一些方法。我们要讨论一种简单的带对数近似比的贪心启发式方法，亦即，随着实例规模逐渐增大，相对于一个最优解的规模来说，近似解的规模也可能增大。但是，由于对数函数增长很慢，故这个近似算法可能会产生出很有用的结果来。

集合覆盖问题的一个实例 $(X, \mathcal{F})$ 由一个有穷集 X 和一个 X 的子集族 $\mathcal{F}$ 构成，且 X 的每一
1033 个元素属于 $\mathcal{F}$ 中的至少一个子集：

$$X = \bigcup_{S \in \mathcal{F}} S$$

我们说一个子集 $S \in \mathcal{F}$ 覆盖了它的元素。这个问题是要找到一个最小规模子集 $C \subseteq \mathcal{F}$，使其所有成员覆盖 X 的所有成员：

$$X = \bigcup_{S \in C} S \tag{35.8}$$

我们说任何满足方程(35.8)的 C 覆盖 X。图 35-3 说明了集合覆盖问题。C 的规模被定义为它所包含的集合数，而不是这些集合中的元素数。在图 35-3 中，最小集合覆盖的规模为 3。

集合覆盖问题是对许多常见的组合问题的一种抽象。来看一个简单的例子：假设 X 表示解决某一问题所需要的各种技巧的集合，另外有一个给定的可用来解决该问题的人的集合。我们希望组成一个人数尽可能少的委员会，使得对 X 中每种必需的技巧，委员会中都有一位成员掌握该技巧。在集合覆盖问题的判定版本中，我们想知道是否存在一个规模至多为 k 的覆盖，k 是在该问题的实例中规定的另一个参数。这个问题的判定版本是 NP 完全的，练习 35.3-2 要求读者证明这一点。

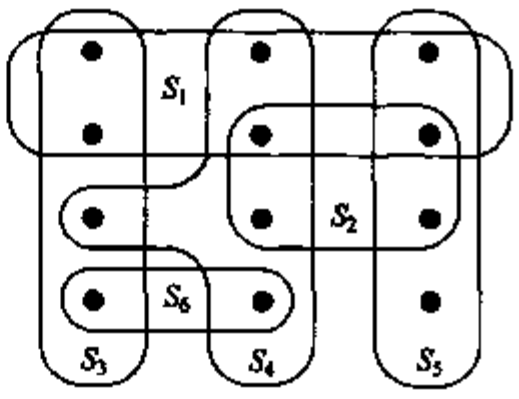


图 35-3　集合覆盖问题的一个实例 $(X, \mathcal{F})$，其中 X 包含 12 个黑点，并且 $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$。一个最小规模集合覆盖为 $C = \{S_3, S_4, S_5\}$。通过按序选择集合 S_1、S_4、S_6 和 S_3，贪心算法产生出一个大小为 4 的覆盖

一个贪心近似算法

1034 贪心方法在每一阶段都选择出能覆盖最多的、未被覆盖的元素的集合 S。

```
GREEDY-SET-COVER(X, F)
1  U ← X
2  C ← ∅
3  while U ≠ ∅
4      do select an S ∈ F that maximizes |S ∩ U|
5         U ← U − S
```

```
6        C ← C∪{S}
7  return C
```

在图 35-3 的例子中，GREEDY-SET-COVER 按序将集合 S_1、S_4、S_5、S_3 加入到 C 中。

这个算法的工作过程是这样的：在每个阶段，集合 U 包含余下的未被覆盖的元素构成的集合；集合 C 包含正在被构造的覆盖。第 4 行是贪心决策步骤，即选出一个子集 S，它能覆盖尽可能多的未被覆盖的元素(如果有两个子集覆盖了一样多的元素，可以任意选择其中的一个)。在 S 被选出后，将其元素从 U 中去掉，并将 S 置于 C 中。当该算法结束时，集合 C 就包含了一个覆盖 X 的 $\mathcal{F}$ 的子族。

很容易实现算法 GREEDY-SET-COVER，使其以 $|X|$ 和 $|\mathcal{F}|$ 的多项式时间运行。因为第 3～6 行间循环的迭代次数至多为 $\min(|X|, |\mathcal{F}|)$，又可以将循环体实现以时间 $O(|X||\mathcal{F}|)$运行，故存在一个运行时间为 $O(|X||\mathcal{F}|\min(|X|, |\mathcal{F}|))$的实现。练习 35.3-3 要求读者给出一个线性时间的算法。

分析

下面来证明以上的贪心算法可以返回一个比最优集合覆盖大不了很多的集合覆盖。为方便起见，在本章中，我们用 $H(d)$来表示第 d 级调和数 $H_d = \sum_{i=1}^{d} 1/i$(见 A.1 节)。作为一个边界条件，定义 $H(0)=0$。

定理 35.4　GREEDY-SET-COVER 是一个多项式时间的 $\rho(n)$近似算法，其中：

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\})$$

证明：我们已经证明了 GREEDY-SET-COVER 是以多项式时间运行的。

为了证明 GREEDY-SET-COVER 是一个 $\rho(n)$近似算法，对每一个由该算法选出的集合赋予一个代价 1，将这一代价分布于初次被覆盖的元素上，再利用这些代价来导出一个最优集合覆盖 C^* 的规模和由该算法返回的集合覆盖 C 的规模之间的关系。设 S_i 表示由 GREEDY-SET-COVER 所选出的第 i 个子集；在将 S_i 加入 C 中时要发生代价 1。将这个选择 S_i 的代价均匀地分布于首次被 S_i 覆盖的元素之上。设 c_x 表示分配给元素 x 的代价，$x \in X$。对每一个元素只分配一次代价，即当它被首次覆盖时分配一个代价。如果 x 首次被 S_i 覆盖，则

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$$

在算法的每一步中，要分配 1 个单位的代价，因此有：

$$|C| = \sum_{x \in X} c_x$$

分配给最优覆盖的代价为：

$$\sum_{S \in C^*} \sum_{x \in S} c_x$$

由于每一个 $x \in X$ 都至少在一个集合 $S \in C^*$ 中，因此有：

$$\sum_{S \in C^*} \sum_{x \in S} c_x \geqslant \sum_{x \in X} c_x$$

将上面的两个不等式组合起来，有：

$$|C| \leqslant \sum_{S \in C^*} \sum_{x \in S} c_x \tag{35.9}$$

证明的余下部分关键在于以下的不等式，我们稍后就要对它进行证明。对属于族 $\mathcal{F}$ 的任何集合 S，有

1035

$$\sum_{x\in S} c_x \leqslant H(|S|) \tag{35.10}$$

根据不等式(35.9)和不等式(35.10)，可得：

$$|C| \leqslant \sum_{S\in C^*} H(|S|) \leqslant |C^*| \cdot H(\max\{|S|: S\in \mathcal{F}\})$$

这就证明了该定理。

现在来证明不等式(35.10)。对任意集合 $S\in\mathcal{F}$ 和 $i=1, 2, \cdots, |C|$，设

$$u_i = |S-(S_1 \cup S_2 \cup \cdots \cup S_i)|$$

为 $S_1, S_2, \cdots, S_i$ 被该算法选出之后，S 中余下的未被覆盖的元素个数。定义 $u_0 = |S|$ 为 S 中元素(开始时它们都未被覆盖)的个数。设 k 为满足 $u_k=0$ 的最小下标，使得 S 的每个元素至少被集合 $S_1, S_2, \cdots, S_k$ 中之一所覆盖。这样，对 $i=1, 2, \cdots, k$，$u_{i-1}\geqslant u_i$，且 S 中共有 $u_{i-1}-u_i$ 个元素首次被 S_i 所覆盖。于是有

1036

$$\sum_{x\in S} c_x = \sum_{i=1}^{k} (u_{i-1}-u_i)\cdot \frac{1}{|S_i-(S_1\cup S_2\cup\cdots\cup S_{i-1})|}$$

注意到

$$|S_i-(S_1\cup S_2\cup\cdots\cup S_{i-1})| \geqslant |S-(S_1\cup S_2\cup\cdots\cup S_{i-1})| = u_{i-1}$$

这是因为对 S_i 的贪心选择保证了 S 不可能比 S_i 覆盖更多的元素(否则，选出的就会是 S，而不是 S_i)。由此可得

$$\sum_{x\in S} c_x \leqslant \sum_{i=1}^{k} (u_{i-1}-u_i)\cdot \frac{1}{u_{i-1}}$$

现在来如下地给出这个量的界：

$$\begin{aligned}
\sum_{x\in S} c_x &\leqslant \sum_{i=1}^{k} (u_{i-1}-u_i)\cdot \frac{1}{u_{i-1}} \\
&= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\
&\leqslant \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \qquad (\text{由于 } j\leqslant u_{i-1}) \\
&= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\
&= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\
&= H(u_0) - H(u_k) \qquad (\text{根据和的叠缩}) \\
&= H(u_0) - H(0) \\
&= H(u_0) \qquad (\text{由于 } H(0)=0) \\
&= H(|S|),
\end{aligned}$$

1037 这就完成了对不等式(35.10)的证明。 ■

推论 35.5　GREEDY-SET COVER 是一个多项式时间的($\ln|X|+1$)近似算法。

证明：利用不等式(A.12)和定理 35.4 即可。 ■

在某些应用中，$\max\{|S|: S\in\mathcal{F}\}$是一个较小的常数，在这种情况下，由 GREEDY-SET-COVER 返回的解就至多比最优解大一个很小的常数倍。例如，对一个其顶点的度至多为 3 的图来说，当利用这种启发式来获取其近似顶点覆盖时，即会出现这样的应用。在这种情况下，由 GREEDY-SET-COVER 找出的解不大于一个最优解的 $H(3)=11/6$ 倍，这个性能保证比 APPROX-VERTEX-COVER 的要略好一些。

练习

35.3-1 将以下的每一个单词都看作是字母的集合：{arid，dash，drain，heard，lost，nose，shun，slate，snare，thread}。说明当出现两个候选集合可供选择时，如果倾向于在词典中先出现的单词，则 GREEDY-SET-COVER 会产生怎样的集合覆盖。

35.3-2 通过从顶点覆盖问题进行归约，证明集合覆盖问题的判定版本是 NP 完全的。

35.3-3 说明如何实现 GREEDY-SET-COVER，使其运行时间为 $O(\sum_{S\in\mathcal{F}}|S|)$。

35.3-4 以下给出的是定理 35.4 的较弱一些的形式，证明它是成立的：

$$|C|\leqslant|C^*|\max\{|S|:S\in\mathcal{F}\}$$

35.3-5 GREEDY-SET-COVER 可以返回一些不同的解，具体取决于在第 4 行中如何打破“平局”(即如何从两个覆盖同样多元素的候选集合中进行选择——译者注)。给出一个过程 BAD-SET-COVER-INSTANCE(n)，它返回集合覆盖问题的一个 n 元素的实例；对这样的实例，具体取决于在第 4 行中如何打破平局，GREEDY-SET-COVER 可以返回一些不同的、呈 n 的指数的解。

1038

35.4　随机化和线性规划

本节要研究两种在设计近似算法时非常有用的技术：随机化和线性规划。我们要给出 3-CNF 可满足性的最优化版本的一个随机化算法，还要利用线性规划，为顶点覆盖问题的一个带权版本设计近似算法。本节只是简单地涉及了这两种极为有用的技术。本章末的“本章注记”给出了有关这两个领域的进一步参考文献。

解决 MAX-3-CNF 可满足性问题的一个随机化近似算法

正如存在着可以计算出准确解的随机算法一样，也存在着能够计算近似解的随机化算法。我们称某一问题的一个随机化算法具有近似比 $\rho(n)$，如果对任何规模为 n 的输入，该随机化算法所产生的解的期望代价 C 是在最优解的代价 C^* 的一个因子 $\rho(n)$之内：

$$\max\left(\frac{C}{C^*},\frac{C^*}{C}\right)\leqslant\rho(n) \tag{35.11}$$

能达到近似比 $\rho(n)$的随机化算法也称为随机化的 $\rho(n)$近似算法。换句话说，随机化的近似算法类似于确定型的算法，只是其近似比为一个期望值。

如在 34.4 节中定义的那样，3-CNF 可满足性问题的一个特定实例可能是可满足的，也可能不是。为了使其可满足，必须找到一种对变量的赋值，使得每个子句的计算结果都为 1。如果某个实例是不可满足的，我们可能希望计算一下它离“可满足”还差多少；亦即，我们希望找出变量的一种赋值，使得有尽可能多的子句得到满足。称这种最大化问题为 MAX-3-CNF-可满足性问题。MAX-3-CNF 可满足性问题的输入与 3-CNF 可满足性问题的输入是一样的，目标是返回变量的一种赋值，它能最大化结果为 1 的子句的数量。下面来说明以概率 1/2 来随机地将每个变量设置为 1，以概率 1/2 来随机地将每个变量设置为 0，即可得到随机化的 8/7 近似算法。根据 34.4 节中 3-CNF 可满足性的定义，我们要求每个子句都恰包含 3 个不同的文字。进一步地假定所有的子句都不会既包含一个变量，又包含其否定形式。(练习 35.4-1 要求读者去掉这后一个假设。)

1039

定理 35.6　给定 MAX-3-CNF 可满足性问题的一个实例，它有 n 个变量 x_1，x_2，…，x_n 和 m 个子句，以概率 1/2 独立地将每个变量设置为 1 和以概率 1/2 独立地将每个变量设置为 0 的随机化近似算法是一个随机化的 8/7 近似算法。

证明：假设我们已经以概率 1/2 独立地将每个变量设置为 1，以概率 1/2 独立地将每个变量设置为 0。对 $i=1, 2, \cdots, n$，定义指示器随机变量

$$Y_i = \mathrm{I}\{\text{子句 } i \text{ 被满足}\}$$

因此，只要第 i 个子句的文字中，至少有一个已被置为 1，就有 $Y_i=1$。由于在同一个子句中，任何一个文字的出现次数都不会多于一次，又由于我们已经假设了同一子句中不会同时出现一个变量及其否定形式，故每个子句中三个文字的设置都是互相独立的。对于一个子句来说，只有当它的三个文字都被置为 0 时，才不会被满足，因此 $\Pr\{\text{子句 } i \text{ 不被满足}\}=(1/2)^3=1/8$。于是，$\Pr\{\text{子句 } i \text{ 被满足}\}=1-1/8=7/8$。根据引理 5.1，$\mathrm{E}[Y_i]=7/8$。设 Y 为得到满足的子句的总数，有 $Y=Y_1+Y_2+\cdots+Y_m$。于是，有：

$$\mathrm{E}[Y]=\mathrm{E}\Big[\sum_{i=1}^{m} Y_i\Big]=\sum_{i=1}^{m}\mathrm{E}[Y_i] \quad \text{（根据期望的线性性质）}$$

$$=\sum_{i=1}^{m} 7/8 = 7m/8$$

显然，m 是得到满足的子句数量的一个上界，因而，近似比至多为 $m/(7m/8)=8/7$。（证毕） ■

利用线性规划来近似带权顶点覆盖

在*最小权值顶点覆盖问题*（minimum-weight vertex-cover problem）中，给定一个无向图 $G=(V, E)$，其中每个顶点 $v\in V$ 都有一个关联的正的权值 $w(v)$。对任意顶点覆盖 $V'\subseteq V$，定义该顶点覆盖的权为 $w(V')=\sum_{v\in V'} w(v)$。目标是找出一个具有最小权值的顶点覆盖。

对于这个问题，不能直接采用面向无权顶点覆盖的算法，也不能采用随机化的解决方案，因为这两种方法给出的解可能远远不是最优的。但是，对于最小权值顶点覆盖，我们将利用线性规划技术，计算出其权值的一个下界。然后，对计算出来的结果进行“舍入”，并利用它来获得一个顶点覆盖。

1040

假设对每一个顶点 $v\in V$，都安排一个变量 $x(v)$ 与之关联，并且，要求对每一个顶点 $v\in V$，有 $x(v)\in[0, 1]$。将 $x(v)=1$ 解释为 v 在顶点覆盖中，将 $x(v)=0$ 解释为 v 不在顶点覆盖中。那么，我们可以写出这样一条限制，即对于任意边 (u, v)，u 和 v 之中至少有一个必须在顶点覆盖中，即 $x(u)+x(v)\geqslant 1$。这样一来，就引出了以下的用于寻找最小权值顶点覆盖的 0-1 整数规划（0-1 integer program）：

$$\text{最小化} \sum_{v\in V} w(v)x(v) \tag{35.12}$$

条件是

$$x(u)+x(v)\geqslant 1 \quad \text{对每一个 } v\in V \tag{35.13}$$

$$x(v)\in\{0,1\} \quad \text{对每一个 } v\in V \tag{35.14}$$

根据练习 34.5-2 可以知道，寻找满足式(35.13)和式(35.14) 的 $x(v)$ 值这一问题是 NP 难度的，因此，上述的表示并没有什么直接的用处。假设去掉了 $x(v)\in\{0, 1\}$ 这一限制，并代之以 $0\leqslant x(v)\leqslant 1$，就可以得到以下的线性规划，称为*线性规划松弛*（linear-program relaxation）：

$$\text{最小化} \sum_{v\in V} w(v)x(v) \tag{35.15}$$

条件是

$$x(u)+x(v)\geqslant 1 \quad \text{对每一个 } v\in V \tag{35.16}$$

$$x(v)\leqslant 1 \quad \text{对每一个 } v\in V \tag{35.17}$$

$$x(v)\geqslant 0 \quad \text{对每一个 } v\in V \tag{35.18}$$

式(35.12)～式(35.14)中 0-1 整数规划的任意可行解也是式(35.15)～式(35.18)中线性规划

的一个可行解。于是，线性规划的一个最优解是 0-1 整数规划最优解的一个下界，从而也是最小权值顶点覆盖问题最优解的下界。

下面的过程利用上述线性规划的解，构造最小权值顶点覆盖问题的一个近似解。

```
APPROX-MIN-WEIGHT-VC(G, w)
1  C ← ∅
2  compute x̄, an optimal solution to the linear program in lines(35.15)—(35.18)
3  for each v∈V
4       do if x̄(v)≥1/2
5            then C ← C∪{v}
6  return C
```

1041

APPROX-MIN-WEIGHT-VC 的工作过程是这样的：第 1 行将顶点覆盖初始化为空。第 2 行形成式(35.15)～式(35.18)中的线性规划，并求解这一线性规划。最优解要给每一个顶点 v 赋一个相关的值 $\bar{x}(v)$，其中 $0 \leqslant \bar{x}(v) \leqslant 1$。在第 3～5 行中，利用这一值来指导该选择将哪些顶点加入到顶点覆盖 C 中。如果 $\bar{x}(v) \geqslant 1/2$，将 v 加入到 C 中；否则，不将其加入。实际上，我们是在对线性规划的解中，将每个带小数的变量"四舍五入"成 0 或 1，以便获得式(35.12)～式(35.14)中 0-1 整数规划的解。最后，第 6 行返回顶点覆盖 C。

定理 35.7　算法 APPROX-MIN-WEIGHT-VC 是解决最小权值顶点覆盖问题的一个多项式时间的 2 近似算法。

证明：因为在第 2 行中，有一个多项式时间的算法来解决线性规划，因为第 3～5 行中的 for 循环以多项式时间运行，因而 APPROX-MIN-WEIGHT-VC 是一个多项式时间的算法。

现在，来证明 APPROX-MIN-WEIGHT-VC 是一个 2 近似算法。设 C^* 是最小权值顶点覆盖问题的一个最优解，并设 z^* 为式(35.15)～式(35.18)中线性规划的一个最优解。由于最优的顶点覆盖也是该线性规划的可行解，因此，z^* 必定是 $w(C^*)$ 的一个下界，即：

$$z^* \leqslant w(C^*) \tag{35.19}$$

接下来，我们断言通过对变量 $\bar{x}(v)$ 的小数部分进行舍入，得到一个集合 C，它是一个顶点覆盖，并且满足 $w(C) \leqslant 2z^*$。为了搞清楚为什么说 C 是一个顶点覆盖，可以考虑任意边 $(u, v) \in E$。根据限制式(35.16)，我们知道，$\bar{x}(u) + \bar{x}(v) \geqslant 1$，这蕴含着在 $\bar{x}(u)$ 和 $\bar{x}(v)$ 中，至少有一个其值至少为 1/2。因此，u 和 v 两者之中，至少有一个将被加入到顶点覆盖中，因而每一条边都将被覆盖。

现在来看覆盖的权值，有：

$$z^* = \sum_{v \in V} w(v)\,\bar{x}(v) \geqslant \sum_{v \in V,\,\bar{x}(v) \geqslant 1/2} w(v)\,\bar{x}(v) \geqslant \sum_{v \in V,\,\bar{x}(v) \geqslant 1/2} w(v) \cdot \frac{1}{2} = \sum_{v \in C} w(v) \cdot \frac{1}{2} \tag{1042}$$

$$= \frac{1}{2}\sum_{v \in C} w(v) = \frac{1}{2} w(C) \tag{35.20}$$

将不等式(35.19)和式(35.20)结合起来，有：

$$w(C) \leqslant 2z^* \leqslant 2w(C^*)$$

这就说明 APPROX-MIN-WEIGHT-VC 是一个 2 近似算法。（证毕）　■

练习

35.4-1　证明：即使允许一个子句既包含变量又包含其否定形式，将每个变量随机地以概率 1/2 设置为 1 和以概率 1/2 设置为 0，它仍然是一个随机化的 8/7 近似算法。

35.4-2 MAX-CNF 可满足性问题与 MAX-3-CNF 可满足性问题类似，只是它并不限制每个子句都恰包含 3 个文字。对 MAX-CNF 可满足性问题，给出它的一个随机化的 2 近似算法。

35.4-3 在 MAX-CUT 问题中，给定一个无权无向图 $G=(V, E)$。如在第 23 章中一样，定义一个割$(S, V-S)$，并定义一个割的权为通过该割的边的数目。问题的目标是找出一个具有最大权值的割。假设对每个顶点 v，随机地且独立地将 v 以概率 1/2 置入 S 中，以概率 1/2 置入 $V-S$ 中。证明：这个算法是一个随机化的 2 近似算法。

35.4-4 证明：式(35.17)中的限制是多余的，意即，如果将它们从式(35.15)～式(35.18)间的线性规划中去掉，所得到线性规划的任何最优解必定满足对每个 $v\in V$，$x(v)\leqslant 1$。

35.5　子集和问题

子集和问题的一个实例是一个对(S, t)，其中 S 是一个正整数的集合$\{x_1, x_2, \cdots, x_n\}$，$t$ 为一个正整数。这个判定问题是判定是否存在 S 的一个子集，使得其中的数加起来恰为目标值 t。这个问题是 NP 完全的(见 34.5.5 节)。

1043

与此判定问题相联系的最优化问题常常出现于实际应用中。在这种最优化问题中，希望找到$\{x_1, x_2, \cdots, x_n\}$的一个子集，使其中元素相加之和尽可能地大，但不能大于 t。例如，假设我们有一辆能装不多于 t 磅重的货的卡车，并有 n 个不同的盒子要装运，其中第 i 个的重量为 x_i 磅。我们希望在不超过重量极限的前提下，将货尽可能地装满卡车。

在这一节里，我们先给出解决这个最优化问题的一个指数时间算法，然后说明如何来修改算法，使之成为一个完全多项式时间的近似方案。(一个完全多项式时间近似方案的运行时间为 $1/\epsilon$ 以及输入规模的多项式。)

一个指数时间的准确算法

假设对 S 的每个子集 S'，都计算出 S' 中所有元素的和。接着，在所有其元素和不超过 t 的子集中，选择其和最接近 t 的那个子集。显然，这一算法将返回最优解，但它可能需要指数级的时间。为了实现这个算法，可以采用一种迭代过程：在第 i 轮迭代中，计算$\{x_1, x_2, \cdots, x_i\}$的所有子集的元素和，计算的基础是$\{x_1, x_2, \cdots, x_{i-1}\}$的所有子集的和。在此计算过程中，一旦某个特定的子集 S' 的和超过了 t，就没有必要再对它进行处理了，因为 S' 的超集都不会成为最优解。下面给出这一策略的一个实现。

过程 EXACT-SUBSET-SUM 的输入为一个集合 $S=\{x_1, x_2, \cdots, x_n\}$和一个目标值 t；其伪代码一会儿给出。这个过程以迭代的方式计算列表 L_i，其中列出了$\{x_1, x_2, \cdots, x_i\}$的所有子集的和，这些和值都不超过目标值 t。接着，它返回 L_n 中的最大值。

如果 L 是一个由正整数所构成的表，x 是一个正整数，我们用 $L+x$ 来表示通过对 L 中每个元素增加 x 而导出的整数列表。例如，如果 $L=\langle 1, 2, 3, 5, 9\rangle$，则 $L+2=\langle 3, 4, 5, 7, 11\rangle$。我们也对集合应用这个记号，因而

$$S+x=\{s+x: s\in S\}$$

我们还用到了一个辅助过程 MERGE-LISTS(L, L')，返回对它的两个已排序的输入列表 L 和 L' 合并及删除重复值后所产生的排序列表。像在合并排序中用到的 MERGE 过程一样(见 2.3.1 节)，MERGE-LISTS 的运行时间为 $O(|L|+|L'|)$(这里省略 MERGE-LISTS 的伪代码)。

1044

```
EXACT-SUBSET-SUM(S, t)
1  n ← |S|
2  L_0 ← ⟨0⟩
3  for i ← 1 to n
```

```
4       do L_i ← MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
5          remove from L_i every element that is greater than t
6  return the largest element in L_n
```

为了搞清楚 EXACT-SUBSET-SUM 的工作过程，设 P_i 表示通过选择$\{x_1, x_2, \cdots, x_i\}$的一个(可能为空的)子集、并将其成员加起来所得到的所有值的集合。例如，如果 $S=\{1, 4, 5\}$，则：

$$P_1 = \{0,1\}$$
$$P_2 = \{0,1,4,5\}$$
$$P_3 = \{0,1,4,5,6,9,10\}$$

给定等式

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) \tag{35.21}$$

可以通过对 i 的归纳来证明(见练习 35.5-1)，表 L_i 是一个包含 P_i 中的所有值不大于 t 的元素的有序表。因为 L_i 的长度可大至 2^i，一般来说 EXACT-SUBSET-SUM 是一个指数时间的算法。在 t 为 $|S|$ 的多项式或 S 中的所有成员均由 $|S|$ 的一个多项式限界的特殊情况下，EXACT-SUBSET-SUM 是一个多项式时间的算法。

一个完全多项式时间近似方案

对子集和问题，我们可以导出一个完全多项式时间的近似方案。在每个列表 L_i 被创建后，对它进行"修整"。具体的思想是如果 L 中的两个值比较接近，那么，出于寻找近似解的目的，没有理由同时保存这两个数。更准确地说，我们采用一个修整参数 δ，满足 $0<\delta<1$。按 δ 来修整一个列表 L 意味着以这样一种方式从 L 中去除尽可能多的元素，即如果 L' 为修整 L 后的结果，则对从 L 中去除的每个元素 y，都存在着一个仍在 L' 中的、近似 y 的元素 z，使得：

$$\frac{y}{1+\delta} \leqslant z \leqslant y \tag{35.22}$$

可以将这样的一个 z 看成是在新表 L' 中"代表" y。每个 y 都由一个满足不等式(35.22)的 z 来代表。例如，如果 $\delta=0.1$，且 1045

$$L = \langle 10,11,12,15,20,21,22,23,24,29 \rangle$$

则可以修整 L 而得

$$L' = \langle 10,12,15,20,23,29 \rangle$$

其中被删除的值 11 由 10 来代表，被删除的值 21 和 22 由 20 代表，被删除的值 24 由 23 代表。表的修整版本中的元素也是原表的元素，对一个表加以修整后，可以大大减少表中的元素，同时还可以在表中为每个被从该表中删除的元素保留一个与其很接近的(且略小一些的)代表值。

下面给出的过程在给定 L 和 δ 的情况下，在时间 $\Theta(m)$ 内修整一个输入表 $L=\langle y_1, y_2, \cdots, y_m \rangle$，假定 L 已排成单调递增序。该过程的输出是一个修整过的有序表。

```
TRIM(L, δ)
1  m ← |L|
2  L' ← ⟨y_1⟩
3  last ← y_1
4  for i ← 2 to m
5       do if y_i > last · (1+δ)        ▷ y_i ≥ last because L is sorted
6             then append y_i onto the end of L'
7                  last ← y_i
8  return L'
```

L 的元素被按照单调递增次序加以扫描，而一个数被加入返回的列表 L' 中，仅当它是 L 的第一个元素，或者如果它不能由最近被放入 L' 中的数来代表。

给定过程 TRIM 后，可以如下构造近似方案。这个过程的输入为包含 n 个整数的集合 $S=\{x_1, x_2, \cdots, x_n\}$（以任意次序放置），一个目标整数 t，以及一个“近似参数”ϵ，此处：

$$0<\epsilon<1 \tag{35.23}$$

它返回一个值 z，该值落在最优解的 $1+\epsilon$ 倍内。

```
APPROX-SUBSET-SUM(S, t, ∈)
1  n ← |S|
2  L0 ← ⟨0⟩
3  for i ← 1 to n
4       do Li ← MERGE-LISTS(Li−1, Li−1 + xi)
5          Li ← TRIM(Li, ∈/2n)
6          remove from Li every element that is greater than t
7  let z* be the largest value in Ln
8  return z*
```

1046

第 2 行将表 L_0 初始化为仅包含元素 0 的一个表。第 3～6 行间 for 循环的效果是这样的，将 L_i 作为一个包含集合 P_i 的适当修整版本（去掉了所有大于 t 的元素）来计算。因为 L_i 是从 L_{i-1} 构造出来的，故必须保证重复的修整不会引入太多的不准确性。下面将看到，APPROX-SUBSET-SUM 返回一个正确的近似，如果存在的话。

作为一个例子，假设有实例

$$S=\langle 104,102,201,101\rangle$$

且 $t=308$，$\epsilon=0.40$。修整参数 δ 为 $\epsilon/8=0.05$。APPROX-SUBSET-SUM 在所指示的各行上计算出如下值：

第 2 行：$L_0=\langle 0\rangle$

第 4 行：$L_1=\langle 0, 104\rangle$

第 5 行：$L_1=\langle 0, 104\rangle$

第 6 行：$L_1=\langle 0, 104\rangle$

第 4 行：$L_2=\langle 0, 102, 104, 206\rangle$

第 5 行：$L_2=\langle 0, 102, 206\rangle$

第 6 行：$L_2=\langle 0, 102, 206\rangle$

第 4 行：$L_3=\langle 0, 102, 201, 206, 303, 407\rangle$

第 5 行：$L_3=\langle 0, 102, 201, 303, 407\rangle$

第 6 行：$L_3=\langle 0, 102, 201, 303\rangle$

第 4 行：$L_4=\langle 0, 101, 102, 201, 203, 302, 303, 404\rangle$

第 5 行：$L_4=\langle 0, 101, 201, 302, 404\rangle$

第 6 行：$L_4=\langle 0, 101, 201, 302\rangle$

该算法返回 $z^*=302$ 作为答案，它在最优答案 $307=104+102+101$ 的 $\epsilon=40\%$ 之内。实际上，它是在其 2% 之内。

定理 35.8　APPROX-SUBSET-SUM 是关于子集和问题的一个完全多项式时间近似方案。

证明：第 5 行修整 L_i 以及从 L_i 中去除每个大于 t 的元素，该操作保持了 L_i 的每个元素同时也是 P_i 的成员这一性质。所以，在第 8 行返回的值 z^* 确实为 S 的某个子集的元素之和。设

$y^* \in P_n$ 表示子集和问题的一个最优解。那么，根据第 6 行可以知道，$z^* \leqslant y^*$。根据不等式(35.1)，我们需要证明 $y^*/z^* \leqslant 1+\varepsilon$。此外，还必须证明这个算法的运行时间既是 $1/\varepsilon$ 的多项式，又是输入规模的多项式。 1047

通过对 i 做归纳，可以证明对 P_i 中每个至多为 t 的元素 y，存在一个 $z \in L_i$，使得

$$\frac{y}{(1+\varepsilon/2n)^i} \leqslant z \leqslant y \tag{35.24}$$

(见练习 35.5-2。)不等式(35.24)对 $y^* \in P_n$ 必成立，因而存在着一个 $z \in L_n$，使得：

$$\frac{y^*}{(1+\varepsilon/2n)^n} \leqslant z \leqslant y^*$$

从而有：

$$\frac{y^*}{z} \leqslant \left(1+\frac{\varepsilon}{2n}\right)^n \tag{35.25}$$

因为存在着一个 $z \in L_n$ 能满足不等式(35.25)，故该不等式对 z^* 必定成立，而 z^* 是 L_n 中的最大值；亦即，

$$\frac{y^*}{z^*} \leqslant \left(1+\frac{\varepsilon}{2n}\right)^n \tag{35.26}$$

接下来还要证明 $y^*/z^* \leqslant 1+\varepsilon$。首先来证明 $(1+\varepsilon/2n)^n \leqslant 1+\varepsilon$。根据方程(3.13)，有 $\lim_{n\to\infty}(1+\varepsilon/2n)^n = e^{\varepsilon/2}$。因为可以证明

$$\frac{d}{dn}\left(1+\frac{\varepsilon}{2n}\right)^n > 0 \tag{35.27}$$

函数 $(1+\varepsilon/2n)^n$ 在接近极限 $e^{\varepsilon/2}$ 的过程中，随 n 的增长而增长，于是有：

$$\begin{aligned}\left(1+\frac{\varepsilon}{2n}\right)^n &\leqslant e^{\varepsilon/2}\\ &\leqslant 1+\varepsilon/2+(\varepsilon/2)^2 && \text{(根据不等式(3.12))}\\ &\leqslant 1+\varepsilon && \text{(根据不等式(35.23))}\end{aligned} \tag{35.28}$$

将不等式(35.26)和不等式(35.28)结合起来，就完成了对近似比的分析。

为了证明 APPROX-SUBSET-SUM 是一个完全多项式时间近似方案，我们要导出一个关于 L_i 的长度的界。在修整后，L_i 中连续的元素 z 和 z' 必有关系 $z'/z > 1+\varepsilon/2n$。也就是说，它们之间相差的倍数必至少为 $1+\varepsilon/2n$。所以，每个列表都包含了值 0，有可能还包含了值 1，以及 1048
$\lfloor \log_{1+\varepsilon/2n} t \rfloor$ 个其他的值。每个列表 L_i 中的元素数至多为

$$\begin{aligned}\log_{1+\varepsilon/2n} t + 2 &= \frac{\ln t}{\ln(1+\varepsilon/2n)} + 2\\ &\leqslant \frac{2n(1+\varepsilon/2n)\ln t}{\varepsilon} + 2 && \text{(根据不等式(3.16))}\\ &\leqslant \frac{4n\ln t}{\varepsilon} + 2 && \text{(根据不等式(35.23))}\end{aligned}$$

这个界是输入规模的多项式。此处的输入规模即表示 t 所需的位数 $\lg t$，再加上表示集合 S 所需的位数，而后者既是 n 的多项式，也是 $1/\varepsilon$ 的多项式。因为 APPROX-SUBSET-SUM 的运行时间为 L_i 长度的多项式，所以 APPROX-SUBSET-SUM 是一个完全多项式时间的近似方案。(证毕) ■

练习

35.5-1 证明等式(35.21)。然后，证明在执行了 EXACT-SUBSET-SUM 的第 5 行之后，L_i 是一个有序表，它包含了 P_i 中所有不大于 t 的元素。

35.5-2　证明不等式(35.24)。

35.5-3　证明不等式(35.27)。

35.5-4　设 t 为给定输入列表的某个子集之和。如果利用本节给出的近似方案，找出不小于 t 的最小值的良好近似，应该如何修改该近似方案？

思考题

35-1　装箱

假设有一组 n 个物体，其中第 i 个物体的大小 s_i 满足 $0<s_i<1$。我们希望把所有的物体都装入最少的箱子中，这些箱子为单位尺寸大小，即每个箱子能容纳所有物体的一个总
1049 尺寸不大于 1 的子集。

a)证明：确定最少所需箱子个数的问题是 NP 难度的(提示：对子集和问题进行归约)。

首先适合启发式依次考察每个物体，将其放入能容纳它的第一个箱子。设 $S=\sum_{i=1}^{n} s_i$。

b)论证：所需箱子的最优个数至少为 $\lceil S \rceil$。

c)论证：首先适合启发式至多使一个箱子不到半满。

d)证明：由首先适合启发式用到的箱子数绝不会大于 $\lceil 2S \rceil$。

e)证明：首先适合启发式具有近似比 2。

f)给出首先适合启发式的一个有效实现，并分析其运行时间。

35-2　对最大团规模的近似

设 $G=(V, E)$ 为一个无向图。对任意 $k\geqslant 1$，定义 $G^{(k)}$ 为无向图 $(V^{(k)}, E^{(k)})$，其中 $V^{(k)}$ 是 V 中顶点的所有有序 k 元组构成的集合，$E^{(k)}$ 被定义成 $(v_1, v_2, \cdots, v_k)$ 与 $(w_1, w_2, \cdots, w_k)$ 邻接，当且仅当对每一个 $i(1\leqslant i\leqslant k)$ G 中或者有 v_i 与 w_i 邻接，或者有 $v_i=w_i$。

a)证明：$G^{(k)}$ 中最大团的大小等于 G 中最大团的大小的 k 次幂。

b)论证：如果有一个寻找最大规模团的近似算法，其近似比为常数，则该问题存在一个完全多项式时间的近似方案。

35-3　带权集合覆盖问题

假设我们将集合覆盖问题加以一般化，使得族 $\mathcal{F}$ 中的每个集合 S_i 都有一个权值 w_i，而一个覆盖 C 的权则为 $\sum_{S_i\in C} w_i$。我们希望确定一个具有最小权值的覆盖。(35.3 节中处理了对所有的 i，$w_i=1$ 的情况。)

证明贪心集合覆盖启发式可以以很自然的方式加以推广，对带权集合覆盖问题的任何实例提供一个近似解。证明该启发式有一个近似比 $H(d)$，其中 d 为任意集合 S_i 的最大
1050 规模。

35-4　最大匹配

在一个无向图 G 中，所谓匹配(matching)是指这样的一组边，其中任意两条边都不关联于同一顶点。在 26.3 节中，我们看到了如何在一个二分图中寻找最大匹配。在本问题中，我们要来考察一般无向图(即不必是二分图的无向图)中的匹配问题。

a)极大匹配(maximal matching)是指不是任何其他匹配的真子集的匹配。通过给出一个无向图 G 和 G 中的一个极大匹配 M(它不是一个最大匹配)，来证明极大匹配未必是最大匹配。(存在着包含 4 个顶点的这样的图。)

b)考虑一个无向图 $G=(V, E)$。给出一个 $O(E)$时间的贪心算法，用于寻找 G 中的极大匹配。

在这个问题中，我们主要关注寻找最大匹配的多项式时间近似算法。目前，这方面已知的最大匹配的最快算法需要超线性(但是是多项式)时间，这儿的近似算法以线性时间运行，读者需要证明(b)中用于寻找极大匹配的线性贪心算法是有关最大匹配的一个 2 近似算法。

c)证明：G 的一个最大匹配的规模是 G 的任何顶点覆盖的规模的下界。

d)考虑 $G=(V, E)$中的一个极大匹配 M。设

$$T=\{v\in V: M \text{ 中的某条边与 } v \text{ 关联}\}$$

对于由 G 的那些不在 T 中的顶点而构成的子图，应如何分析它们?

e)根据 d)得出这样的结论：$2\mid M\mid$ 是 G 的顶点覆盖的规模。

f)利用 c)和 e)，证明 b)中的贪心算法是有关最大匹配的一个 2 近似算法。

35-5　并行机调度

在并行机调度问题(parallel machine scheduling)中，已知 n 项作业 J_1，J_2，…，J_n，其中每一项作业 J_k 都有一个非负的处理时间 p_k 关联。另外，还已知有 m 台完全相同的机器 M_1，M_2，…，M_m。调度规定每一项作业 J_k 在哪一台机器上运行，在哪一个时间段运行。每一项作业 J_k 必须在一台机器 M_i 上运行连续的 p_k 个时间单位，并且，在该时间段 1051
里，其他作业都不能在 M_i 上运行。设 C_k 表示作业 J_k 的完成时间，亦即，作业 J_k 完成处理的时间。给定一个调度后，定义 $C_{\max}=\max_{1\leqslant j\leqslant n}C_k$ 为该调度的跨度(makespan)。问题的目标是找出一个调度，使其跨度最小。

例如，假设有两台机器 M_1 和 M_2，另有四项作业 J_1、J_2、J_3、J_4，分别有 $p_1=2$，$p_2=12$，$p_3=4$，$p_4=5$。那么，一种可能的调度方案就是在机器 M_1 上，先运行作业 J_1，再运行作业 J_2；在机器 M_2 上，先运行作业 J_4，再运行作业 J_3。在这个调度中，$C_1=2$，$C_2=14$，$C_3=9$，$C_4=5$，$C_{\max}=14$。一种最优调度方案是这样的，就是在机器 M_1 上运行 J_2，在机器 M_2 上运行作业 J_1，J_3 和 J_4。在这个调度中，$C_1=2$，$C_2=12$，$C_3=6$，$C_4=11$，$C_{\max}=12$。

给定一个并行机调度问题，用 $C^*_{\max}$ 表示一个最优调度的跨度。

a)证明：最优跨度至少与最大处理时间一样大，亦即：

$$C^*_{\max}\geqslant \max_{1\leqslant k\leqslant n} p_k$$

b)证明：最优跨度至少与平均的机器负载一样大，亦即：

$$C^*_{\max}\geqslant \frac{1}{m}\sum_{1\leqslant k\leqslant n} p_k$$

假设我们利用以下贪心算法来解决并行机调度问题：每当一台机器空闲下来后，就将任何尚未被调度的作业调度到该机器上。

c)编写伪代码来实现这一贪心算法。你给出的算法的运行时间怎样?

d)对贪心算法所返回的调度，证明：

$$C_{\max}\leqslant \frac{1}{m}\sum_{1\leqslant k\leqslant n} p_k+\max_{1\leqslant k\leqslant n} p_k$$

并总结此算法是一个多项式时间的 2 近似算法。

本章注记

1052 有些解决问题的方法给出的未必是准确解，人们知道这些方法已有数千年的时间了(例如，对 π 的值进行近似的方法)，但是，近似算法却是一个非常新的概念。Hochbaum 认为，是 Garey，Graham，以及 Ullman[109]，还有 Johnson[166]形式化了多项式时间近似算法这一概念。通常认为，第一个这样的算法是由 Graham[129]给出的，思考题 35-5 中即讨论了该算法。

自从这项早期的工作以来，人们针对这类问题，提出了数以千计的近似算法，这一领域中也出现了大量的文献。Ausiello 等人[25]、Hochbaum[149]和 Vazirani[305]等人编写的教材是比较新的，它们专门讨论了近似算法方面的问题，Shmoys[277]、Klein 和 Young[181]的综述也涉及了这方面的内容。其他的几本教材，如 Garey 和 Johnson[110]、Papadimitriou 和 Steiglitz[237]，也涉及了许多有关近似算法方面的内容。Lawler，Lenstra，Rinnooy Kan，以及 Shmoys[197]都详尽地讨论了旅行商问题的近似算法。

根据 Papadimitriou 和 Steiglitz，APPROX-VERTEX-COVER 算法是由 F. Gavril 和 M. Yannakakis 提出的。顶点覆盖问题得到了大量的研究(Hochbaum[149]列出了这一问题的 16 种不同的近似算法)，但所有这些算法的近似比都至少是 $2-o(1)$。

算法 APPROX-TSP-TOUR 出现在 Rosenkrantz，Stearns 以及 Lewis 等的一篇论文[261]里。Christofedes 改进了这一算法，给出了旅行商问题的一个满足三角不等式的、3/2 近似算法。Arora[21]和 Mitchell[223]证明了如果各个点位于欧几里得平面上，就存在着一个多项式时间的近似算法。定理 35.3 源自于 Sahni 和 Gonzalez[264]。

对解决集合覆盖问题的贪心启发式策略的分析源自于 Chvátal[61]发表的证明中一个更为一般的结果；本章中给出的基本结果源自于 Johnson[166]和 Lovász[206]。

算法 APPROX-SUBSET-SUM 及其分析大致源自于 Ibarra 和 Kim[164]给出的背包问题和子集和问题的有关近似算法。

MAX-3-CNF 可满足性问题的随机化算法来自 Johnson[166]的工作。带权顶点覆盖算法是由 Hochbaum[148]提出的。35.4 节中仅仅是略显示出了随机化和线性规划技术在设计近似算法方面的强大作用。这两种思想的结合产生了一种称为“随机化舍入”(randomized rounding)的技术。当利用这种技术来解决某一问题时，该问题首先被模型化为一个整数线性规划。接着，求解其线性规划松弛方程，所得到的解中的各个变量被作为概率加以解释。这些概率随即被用来帮助导出原问题的解。这一技术首先由 Raghavan 和 Thompson[255]使用，之后即被人们大量使用。
1053 (Motwani，Naor，以及 Raghavan[227]给出了一个综述。)在近似算法方面，还有其他几种比较突出的新思想与方法，如主对偶(primal dual)方法([116]给出了有关这一技术的综述)，寻找用于分治算法中稀疏割集[199]，半确定型程序设计[115]的使用等。

第 34 章的“本章注记”中提到过，在概率可检验证明方面的最新成果导出了许多问题的可近似性的下界，也包括本章中讨论的几个问题。除了第 34 章中列出的参考文献外，Arora 和 Lund
1054 [22]也对概率可检验证明与近似各种问题的困难性的关系做了很好的分析。

第八部分　附录：数学基础知识

引　言

在做算法分析时，经常需要借助一些数学工具。这些工具中有些和高中的代数一样简单，但也有一些可能是读者不太熟悉的。在本书的第一部分中，已经介绍了如何处理渐近表示和递归式。本部分(附录)介绍了算法分析中用到的另一些概念和方法。正如第一部分引言中所说的那样，读者在阅读这本书之前，可能已经读过了附录部分所介绍的许多内容(本书中的记号约定与读者在其他书中所看到的可能有所不同)。因此，读者应该把附录当成参考材料。和本书的其他部分一样，为了提高读者的能力，附录部分也包括了一些练习题和思考题。

附录A介绍了一些计算和式以及给出其界的方法，这些方法在算法分析中经常出现。这部分的许多公式在任何一本微积分课本中都会被提及，但是，把它们编辑在一起后，使用起来会更加方便。

附录B包括了有关集合、关系、函数、图以及树的基本定义和表示，另外还介绍了这些数学对象的一些基本性质。

在附录C中，首先介绍了计数的基本原理，即排列组合等方面的内容。接着，介绍了初等概率论的定义和性质。本书中绝大部分的算法分析都不需要概率知识，所以在读者第一次阅读时，可以跳过该附录后面的部分，甚至不需浏览。以后当需要更深入理解概率分析时，就会发现附录C的内容组织很适合于作为参考资料使用。

A 求　和

当一个算法包含像 while 或 for 循环这样的迭代控制结构时，其运行时间可以表示成循环体每次执行时间的总和。例如，在 2.2 节中我们已经知道，在最坏情况下，插入排序的第 j 次迭代所需的时间与 j 成正比。通过把每次迭代所需的时间相加，就可以获得和式(或称级数)：$\sum_{j=2}^{n} j$。对这个和式求值，即可以得到算法在最坏情况下运行时间的一个上界 $\Theta(n^2)$。这个例子指出理解如何计算和式及其界的重要性。

A.1 节列举几个包含求和的基本公式。A.2 节提出计算和的界限的有用技术。A.1 节中的公式没有给出证明，不过它们之中有些证明会在 A.2 节给出，以解释那一节中所用到的方法。在任何一本微积分课本中，都可以找到其他大部分的证明。

A.1　求和公式及其性质

给定一个数列 a_1，a_2，…，有限和 $a_1+a_2+\cdots+a_n$(n 是非负整数)可以写成

$$\sum_{k=1}^{n} a_k$$

如果 $n=0$，该和式的值定义为 0。有穷级数的值总是有定义的，并且它的项可以按任何顺序相加。

[1058] 给定一个数列 a_1，a_2，…，无穷和 $a_1+a_2+\cdots$ 可以写成

$$\sum_{k=1}^{\infty} a_k$$

这个公式等价于

$$\lim_{n\to\infty} \sum_{k=1}^{n} a_k$$

如果极限不存在，则这个级数是发散的；反之，它就是收敛的。一个收敛级数的项并不总是可以按任意顺序相加的。我们可以重新排列一个绝对收敛级数 $\sum_{k=1}^{\infty} a_k$(亦即，级数 $\sum_{k=1}^{\infty} |a_k|$ 也收敛)的项。

线性性质

对任意实数 c 和任意有限序列 a_1，a_2，…，a_n 和 b_1，b_2，…，b_n，有

$$\sum_{k=1}^{n} (ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k$$

无穷收敛级数也具有线性性质。

线性特征可以被用来对含有渐近记号的项求和，例如

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right)$$

在这个等式中，左边的 Θ 表示应用于变量 k，而在右边它应用于变量 n。对无穷收敛级数可以做同样的处理。

等差级数

和式

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$$

是一个等差级数，它的值为

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1) \tag{A.1}$$

$$= \Theta(n^2) \tag{A.2}$$

1059

平方和与立方和

平方和与立方和的求和公式如下：

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{A.3}$$

$$\sum_{k=0}^{n} k^3 = \frac{n^2 (n+1)^2}{4} \tag{A.4}$$

几何级数

对于实数 $x \neq 1$，和式

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

是一个几何级数(或称指数级数)，它的值是

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \tag{A.5}$$

当和是无穷的且 $|x| < 1$ 时，即得到无穷递减几何级数

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \tag{A.6}$$

调和级数

对正整数 n，第 n 个调和数是

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1) \tag{A.7}$$

(A.2 节将给出关于这个界的证明。)

级数的积分和微分

可以对上述公式进行积分或求导来获得其他一些公式。例如，对 $|x| < 1$，通过对无穷几何级数(A.6)的两端求导并乘以 x，可以得到

1060

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1 - x)^2} \tag{A.8}$$

套迭级数

对于任何序列 a_0，a_1，…，a_n，

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0 \tag{A.9}$$

因为 a_1，a_2，…，a_{n-1} 中的每一个项都是恰好加进一次又减去一次。我们称这个和压缩了。类似地，有：

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n$$

下面来看一个压缩和式的例子，考虑级数

$$\sum_{k=1}^{n-1}\frac{1}{k(k+1)}$$

因为级数中每个项都可以重写为

$$\frac{1}{k(k+1)}=\frac{1}{k}-\frac{1}{k+1}$$

得到

$$\sum_{k=1}^{n-1}\frac{1}{k(k+1)}=\sum_{k=1}^{n-1}\left(\frac{1}{k}-\frac{1}{k+1}\right)=1-\frac{1}{n}$$

乘积

有限乘积 $a_1a_2\cdots a_n$ 可以写成

$$\prod_{k=1}^{n}a_k$$

如果 $n=0$，乘积的值定义为 1。通过等式

1061

$$\lg\left(\prod_{k=1}^{n}a_k\right)=\sum_{k=1}^{n}\lg a_k$$

可以把一个包含乘积的公式转变为一个包含求和的公式。

练习

A. 1-1　求 $\sum_{k=1}^{n}(2k-1)$ 的简化公式。

***A. 1-2**　利用调和级数性质证明 $\sum_{k=1}^{n}1/(2k-1)=\ln(\sqrt{n})+O(1)$。

A. 1-3　对 $0<|x|<1$，证明 $\sum_{k=0}^{\infty}k^2x^k=x(1+x)/(1-x)^3$。

***A. 1-4**　证明 $\sum_{k=0}^{\infty}(k-1)/2^k=0$。

***A. 1-5**　求 $\sum_{k=1}^{\infty}(2k+1)x^{2k}$ 的值。

A. 1-6　利用求和公式的线性特征证明 $\sum_{k=1}^{n}O(f_k(n))=O\left(\sum_{k=1}^{n}f_k(n)\right)$。

A. 1-7　求 $\prod_{k=1}^{n}2\cdot 4^k$ 的值。

***A. 1-8**　求 $\prod_{k=2}^{n}(1-1/k^2)$ 的值。

A. 2　确定求和时间的界

对于描述了算法运行时间的求和公式，可以利用多种技术来计算其界。这里给出一些常用的方法。

数学归纳法

计算级数最常用的方法是数学归纳法。例如，我们来证明等差级数 $\sum_{k=1}^{n}k$ 的值等于 $\frac{1}{2}n(n+1)$。容易看出，当 $n=1$ 时，这一结论是成立的，因此可以归纳假设对 n 成立，并证明对 $n+1$ 成立。我们有

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1) = \frac{1}{2}n(n+1) + (n+1) = \frac{1}{2}(n+1)(n+2) \qquad 1062$$

在使用数学归纳法时，并不一定要推算出和的准确值。另外，归纳法也可以用来计算和式的界。例如，下面来证明几何级数 $\sum_{k=0}^{n} 3^k$ 的界是 $O(3^n)$。更具体一点，对某个常量 c 证明 $\sum_{k=0}^{n} 3^k \leqslant c3^n$。对初始条件 $n=0$，当 $c \geqslant 1$ 时有 $\sum_{k=0}^{0} 3^k = 1 \leqslant c \cdot 1$。假设这个界限对 n 成立，证明它对 $n+1$ 同样成立。有

$$\sum_{k=0}^{n+1} 3^k = \sum_{k=0}^{n} 3^k + 3^{n+1} \leqslant c3^n + 3^{n+1} = \left(\frac{1}{3} + \frac{1}{c}\right)c3^{n+1} \leqslant c3^{n+1}$$

当 $(1/3+1/c) \leqslant 1$ 或 $c \geqslant 3/2$ 时成立。因此，$\sum_{k=0}^{n} 3^k = O(3^n)$。

当在归纳法中使用渐近表示来证明界限时必须很小心。考虑一个错误的证明 $\sum_{k=1}^{n} k = O(n)$。当然 $\sum_{k=1}^{1} k = O(1)$。假设对这个界对 n 成立，现在来证明它对 $n+1$ 成立：

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1) \\ &= O(n) + (n+1) \qquad \Leftarrow \text{错了！} \\ &= O(n+1)\end{aligned}$$

问题是 O 记号中所隐藏的“常量”是随 n 而增长的，因而是变化的，不是常量。我们没有证明同样的常量对所有的 n 成立。

确定级数各项的界

有时，一个级数的理想上界可以通过对级数中的每个项求界来获得。通常，使用最大的项作为其他项的界就足够了。例如，等差级数(A.1)的一个可快速获得的上界是

$$\sum_{k=1}^{n} k \leqslant \sum_{k=1}^{n} n = n^2 \qquad 1063$$

一般来说，对级数 $\sum_{k=1}^{n} a_k$，如果令 $a_{\max} = \max_{1 \leqslant k \leqslant n} a_k$，则

$$\sum_{k=1}^{n} a_k \leqslant n a_{\max}$$

当一个级数实际上以一个几何级数为界时，选择级数的最大项作为每项的界是很弱的方法。给定一个级数 $\sum_{k=0}^{n} a_k$，假设对所有的 $k \geqslant 0$ 有 $a_{k+1}/a_k \leqslant r$，这时 $0 < r < 1$ 是一个常量。可以把一个无穷递减几何级数作为和的界限，因为 $a_k \leqslant a_0 r^k$，所以

$$\sum_{k=0}^{n} a_k \leqslant \sum_{k=0}^{\infty} a_0 r^k = a_0 \sum_{k=0}^{\infty} r^k = a_0 \frac{1}{1-r}$$

可以利用这个方法来计算和 $\sum_{k=1}^{\infty} k/3^k$ 的界。为了使求和从 $k=0$ 开始，将级数重写为 $\sum_{k=0}^{\infty} (k+1)/3^{k+1}$，第一项($a_0$)是 1/3，连续两项的比率($r$)是

$$\frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} = \frac{1}{3} \cdot \frac{k+2}{k+1} \leqslant \frac{2}{3}$$

对所有的 $k \geqslant 0$ 成立。因此，有

$$\sum_{k=1}^{\infty}\frac{k}{3^k}=\sum_{k=0}^{\infty}\frac{k+1}{3^{k+1}}\leqslant\frac{1}{3}\cdot\frac{1}{1-2/3}=1$$

使用这个方法的一个常见问题是当证明了连续两项的比率小于 1 后，就假设这个和以一个几何级数为界。一个例子是无穷调和级数，它是发散的，因为

1064

$$\sum_{k=1}^{\infty}\frac{1}{k}=\lim_{n\to\infty}\sum_{k=1}^{n}\frac{1}{k}=\lim_{n\to\infty}\Theta(\lg n)=\infty$$

级数中第 $k+1$ 项与第 k 项的比率是 $k/(k+1)<1$，但这个级数并不是以一个递减几何级数为界。要将一个几何级数作为级数的界，必须证明有一个 $r<1$，r 是常量，并且所有连续两项的比率都不能超过 r。在调和级数中，不存在这样的一个 r，因为比率是任意逼近 1 的。

分割求和

在求复杂的求和公式的界时，可以将级数表示为两个或多个级数，按下标的范围进行划分，然后再对每个级数分别求界。例如，要求等差级数 $\sum_{k=1}^{n}k$ 的下界，这个级数已经被证明有上界 n^2。我们可能会尝试使用最小项作为级数中每个项的界，但因为最小项是 1，得到下界为 n，这与上界 n^2 相去甚远。

可以首先分割和式来获得一个更理想的下界。方便起见，假设 n 为偶数。有

$$\sum_{k=1}^{n}k=\sum_{k=1}^{n/2}k+\sum_{k=n/2+1}^{n}k\geqslant\sum_{k=1}^{n/2}0+\sum_{k=n/2+1}^{n}(n/2)=(n/2)^2=\Omega(n^2)$$

因为 $\sum_{k=1}^{n}k=O(n^2)$，这是一个渐近确界。

对于从算法分析中获得的和，我们常常对其进行分割并忽略开始的一些常数项。这种方法通常用在和 $\sum_{k=0}^{n}a_k$ 中的每个项 a_k 都独立于 n 的情况。对任何常数 $k_0>0$，可以写成

$$\sum_{k=0}^{n}a_k=\sum_{k=0}^{k_0-1}a_k+\sum_{k=k_0}^{n}a_k=\Theta(1)+\sum_{k=k_0}^{n}a_k$$

因为和式中开始的一些项都是常量，并且项数也是固定的，可以用其他方法来计算 $\sum_{k=k_0}^{n}a_k$ 的界

1065 限。这种方法同样可以用在无穷项的求和。例如，对

$$\sum_{k=0}^{\infty}\frac{k^2}{2^k}$$

求渐近上界。

注意到当 $k\geqslant 3$ 时，连续两项的比率为：

$$\frac{(k+1)^2/2^{k+1}}{k^2/2^k}=\frac{(k+1)^2}{2k^2}\leqslant\frac{8}{9}$$

因此，上述的和式可以分割成

$$\sum_{k=0}^{\infty}\frac{k^2}{2^k}=\sum_{k=0}^{2}\frac{k^2}{2^k}+\sum_{k=3}^{\infty}\frac{k^2}{2^k}\leqslant\sum_{k=0}^{2}\frac{k^2}{2^k}+\frac{9}{8}\sum_{k=0}^{\infty}\left(\frac{8}{9}\right)^k=O(1)$$

因为第一个和有常数个项，而第二个和是一个递减几何级数。

分割求和的方法可以在更复杂的情况下求渐近界限。例如，可以获得调和级数(A.7)

$$H_n=\sum_{k=1}^{n}\frac{1}{k}$$

的界 $O(\lg n)$。

方法的思想是将范围 1 到 n 划分为$\lfloor\lg n\rfloor$个部分，每个部分的上界为 1。每个部分的项从 $1/2^i$

到$1/2^{i+1}$(不包括$1/2^{i+1}$)，给定

$$\sum_{k=1}^{n}\frac{1}{k}\leqslant\sum_{i=0}^{\lfloor\lg n\rfloor}\sum_{j=0}^{2^i-1}\frac{1}{2^i+j}\leqslant\sum_{i=0}^{\lfloor\lg n\rfloor}\sum_{j=0}^{2^i-1}\frac{1}{2^i}\leqslant\sum_{i=0}^{\lfloor\lg n\rfloor}1\leqslant\lg n+1 \tag{A.10}$$

1066

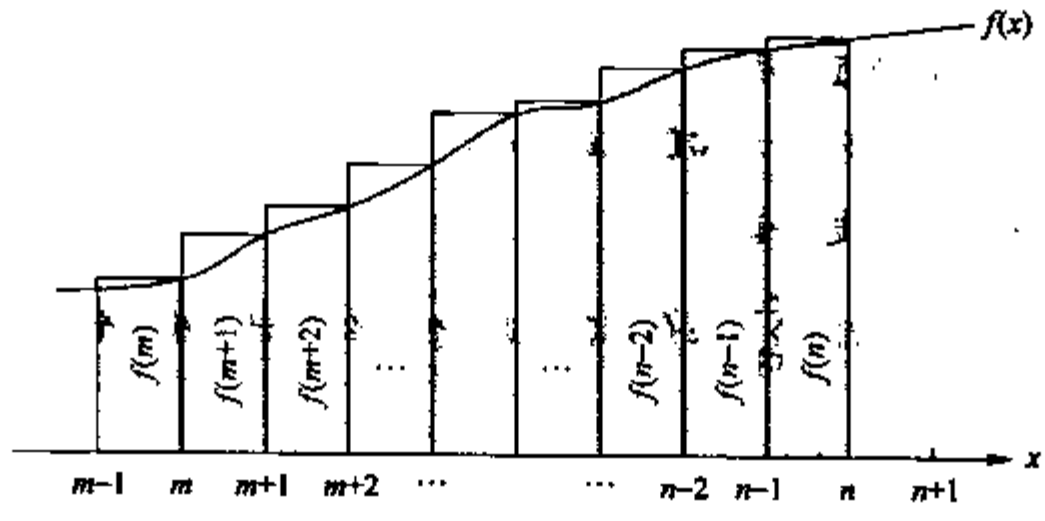
通过积分求近似值

当一个和表示成$\sum_{k=m}^{n}f(k)$，$f(k)$是单调递增函数时，可以通过积分来求它的近似值：

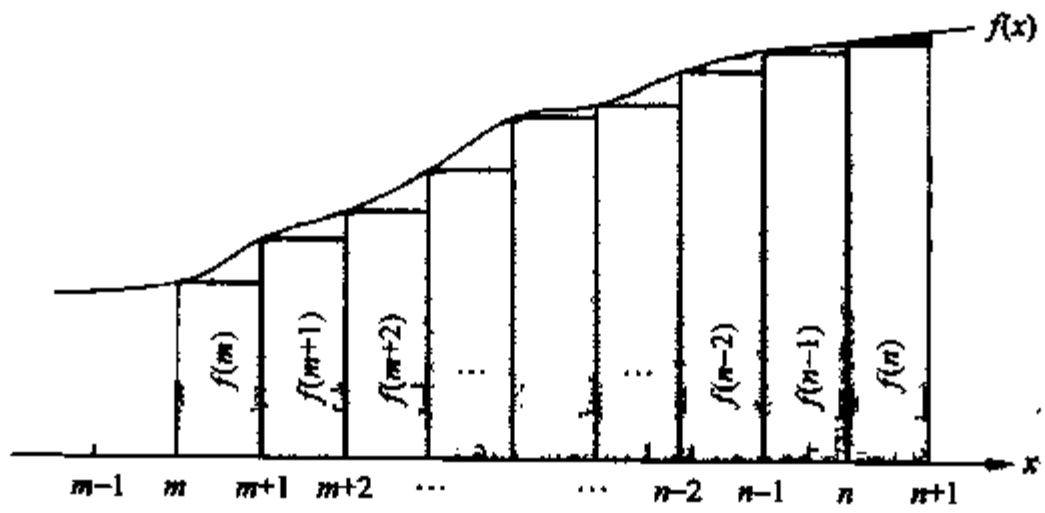
$$\int_{m-1}^{n}f(x)\mathrm{d}x\leqslant\sum_{k=m}^{n}f(k)\leqslant\int_{m}^{n+1}f(x)\mathrm{d}x \tag{A.11}$$

这一近似的理由及含义可见图 A.1。图中矩形部分的面积表示和，曲线下的阴影区域表示积分。当$f(k)$单调递增时，可以使用类似的方法来计算界

$$\int_{m}^{n+1}f(x)\mathrm{d}x\leqslant\sum_{k=m}^{n}f(k)\leqslant\int_{m-1}^{n}f(x)\mathrm{d}x \tag{A.12}$$



a)



b)

图 A.1 通过积分求$\sum_{k=m}^{n}f(k)$的近似值。每个矩形的面积显示在矩形内，矩形面积的总和表示求和的值。曲线下的阴影区域表示积分。通过比较 a) 中的面积，得到不等式$\int_{m-1}^{n}f(x)\mathrm{d}x\leqslant\sum_{k=m}^{n}f(k)$，然后通过把矩形向右移动一个单位，根据 b) 得到不等式$\sum_{k=m}^{n}f(k)\leqslant\int_{m}^{n+1}f(x)\mathrm{d}x$

积分近似(A.12)对第 n 个调和数给出了一个紧确的估计。对于其下界，有

$$\sum_{k=1}^{n} \frac{1}{k} \geqslant \int_{1}^{n+1} \frac{dx}{x} = \ln(n+1) \tag{A.13}$$

对于上界，可以推导出如下的不等式

$$\sum_{k=2}^{n} \frac{1}{k} \leqslant \int_{1}^{n} \frac{dx}{x} = \ln n$$

从它可以导出上界

$$\sum_{k=1}^{n} \frac{1}{k} \leqslant \ln n + 1 \tag{A.14}$$

练习

A.2-1　证明 $\sum_{k=1}^{n} 1/k^2$ 有常量上界。

1067
~
1068

A.2-2　求和 $\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil$ 的渐近上界。

A.2-3　通过分割求和证明第 n 个调和数是 $\Omega(\lg n)$。

A.2-4　通过积分求 $\sum_{k=1}^{n} k^3$ 的近似值。

A.2-5　为什么我们没有直接对 $\sum_{k=1}^{n} 1/k$ 使用积分近似(A.12)来计算第 n 个调和数的上界？

思考题

A-1　求和的界

求下列和式的渐近确界。假设 $r \geqslant 0$，$s \geqslant 0$ 都是常量。

a) $\sum_{k=1}^{n} k^r$

b) $\sum_{k=1}^{n} \lg^s k$

c) $\sum_{k=1}^{n} k^r \lg^s k$

本章注记

Knuth[182]是学习本章时的一本很好的参考书。级数的基本性质可以在任何一本微积分书
1069 中找到，例如 Apostol[18]或 Thomas 与 Finney[296]。

B 集合等离散数学结构

本书中的许多章节涉及到离散数学的内容。本附录更加完整地回顾了集合、关系、函数、图和树的表示、定义和基本性质。读者如果已经非常熟悉这些内容，可以跳过本附录。

B.1 集合

集合包含一组可区分的对象，称为成员或元素。如果一个对象 x 是集合 S 的一个成员，记为 $x \in S$（读作“x 是 S 的成员”，或者更简洁地称“x 在 S 中”）。如果 x 不是 S 的成员，则记为 $x \notin S$。可以通过在一对大括号中显式列出集合的所有成员的方式描述一个集合。例如，可以通过 $S = \{1, 2, 3\}$ 定义一个只包含成员 1，2，3 的集合 S。因为 2 是集合 S 的一个成员，可以记为 $2 \in S$。因为 4 不是 S 的成员，可以记为 $4 \notin S$。集合不能包含一个以上相同的对象，[⊖] 它的元素是无序的。如果两个集合 A 和 B 包含相同的元素，则称集合 A，B 相等，记为 $A = B$。例如，$\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$。

采用特殊的记号表示常用的集合。

- $\varnothing$ 表示空集，也就是说，集合中没有任何成员。
- $\mathbf{Z}$ 表示整数集，也就是集合 $\{\cdots, -2, -1, 0, 1, 2, \cdots\}$。
- $\mathbf{R}$ 表示实数集。
- $\mathbf{N}$ 表示自然数集，即集合 $\{0, 1, 2, \cdots\}$。[⊖]

如果集合 A 中的所有元素都在集合 B 中，即如果 $x \in A$ 就有 $x \in B$，则可以记为 $A \subseteq B$，并称集合 A 是集合 B 的一个子集。如果 $A \subseteq B$ 但 $A \neq B$，则称 A 是 B 的真子集，记为 $A \subset B$。（一些作者使用符号“$\subset$”表示普通子集关系，而不是真子集关系）。对任何集合 A，有 $A \subseteq A$。对两个集合 A 和 B，当且仅当 $A \subseteq B$ 且 $B \subseteq A$ 时，有 $A = B$。对任意三个集合 A，B，C，如果 $A \subseteq B$ 且 $B \subseteq C$，则 $A \subseteq C$。对任意集合 A，有 $\varnothing \subseteq A$。

有时可以通过其他集合来定义集合。对于一个集合 A，为了定义其子集 B，可以通过一个属性区分出 A 中属于 B 的元素。例如，可以通过 $\{x : x \in Z \text{ 且 } x/2 \text{ 是整数}\}$ 定义偶数集。表示中的冒号读作“满足条件”。（一些作者用竖线来代替冒号。）

给定两个集合 A 和 B，可以使用集合操作定义新集合：

- 集合 A 和 B 的交域是集合

$$A \cap B = \{x : x \in A \text{ 且 } x \in B\}$$

- 集合 A 和 B 的并域是集合

$$A \cup B = \{x : x \in A \text{ 或 } x \in B\}$$

- 集合 A 和 B 的差域是集合

$$A - B = \{x : x \in A \text{ 且 } x \notin B\}$$

集合操作服从下列定律。

空集律：

$$A \cap \varnothing = \varnothing$$

⊖ 多重集合是集合的一个变种，它可以包含一个以上相同的对象。

⊖ 一些作者把 1 作为自然数的开始。现在的趋势是从 0 开始。

$$A \cup \varnothing = A$$

幂等律：

$$A \cap A = A$$
$$A \cup A = A$$

交换律：

$$A \cap B = B \cap A$$
$$A \cup B = B \cup A$$

结合律：

$$A \cap (B \cap C) = (A \cap B) \cap C$$
$$A \cup (B \cup C) = (A \cup B) \cup C$$

分配律：

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \tag{B.1}$$

吸收律：

$$A \cap (A \cup B) = A$$
$$A \cup (A \cap B) = A$$

德·摩根律：

$$A - (B \cap C) = (A - B) \cup (A - C)$$
$$A - (B \cup C) = (A - B) \cap (A - C) \tag{B.2}$$

德·摩根律的第一个公式在图 B.1 中通过文氏图来解释。在文氏图中，集合表示成平面的区域。

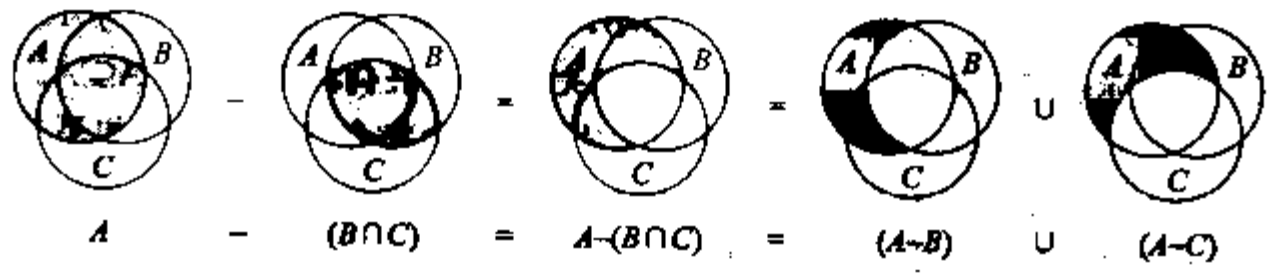


图 B.1　文氏图(Venn diagram)解释了德·摩根律(B.2)的第一个公式。这里集合 A，B，C 用圆形表示

通常，所有被考虑的集合都是一个全域 U 的子集。例如，当考虑各种仅有整数组成的集合时，集合 $\mathbf{Z}$ 就是个合适的全域。给定一个全集 U，可以定义 A 的补集为 $\overline{A} = U - A$。对于任何集合 $A \subseteq U$，有以下定律：

$$\overline{\overline{A}} = A$$
$$A \cap \overline{A} = \varnothing$$
$$A \cup \overline{A} = U$$

德·摩根律(B.2)可以用补集形式表示。对任意两个集合 B，$C \subseteq U$，有

$$\overline{B \cap C} = \overline{B} \cup \overline{C}$$
$$\overline{B \cup C} = \overline{B} \cap \overline{C}$$

如果集合 A 和 B 没有公共的元素，即 $A \cap B = \varnothing$，则称两者不相交。如果一个非空集合 $\mathcal{S} = \{S_i\}$ 满足条件

- 集合两两不相交，即对 S_i，$S_j \in \mathcal{S}$，如果 $i \neq j$，则 $S_i \cap S_j = \varnothing$
- 它们的并是 S，即

$$S = \bigcup_{S_i \in \mathcal{S}} S_i$$

则称集合$\mathcal{S}$是集合 S 的一个划分。换句话说，如果 S 中的每个元素仅出现在一个 $S_i \in \mathcal{S}$，则称$\mathcal{S}$为集合 S 的一个划分。

集合中元素的个数称为集合的势(或大小)，记为$|S|$。如果两个集合的元素可以形成一一对应，则称两者有相同的势。空集的势 $|\varnothing| = 0$。如果一个集合的势是自然数，则称集合是有限的，反之，它是无限的。如果一个无限集合可以跟自然数集合 **N** 形成一一对应，则称其为可数无限；反之，则称其为不可数。整数集 **Z** 是可数的，而实数集 **R** 是不可数的。

对两个有限集合 A 和 B，有等式

$$|A \cup B| = |A| + |B| - |A \cap B| \tag{B.3}$$

从它可以推出不等式

$$|A \cup B| \leqslant |A| + |B|$$

如果 A 和 B 不相交，那么 $|A \cap B| = 0$，从而 $|A \cup B| = |A| + |B|$。如果 $A \subseteq B$，那么 $|A| \leqslant |B|$。

一个具有 n 个元素的有限集合常被称为 n 维集。一个 1 维集也称为单元集。一个集合的 k 个元素的子集称为 k 子集。

集合 S 的所有子集的集合，包括空集和 S 本身表示为 2^S，称为 S 的幂集。例如，$2^{\{a,b\}} = \{\varnothing, \{a\}, \{b\}, \{a, b\}\}$。有限集 S 的幂集的势是 $2^{|S|}$。

有时我们需要关注一些类似集合，但元素是有序的结构。一个包含两个元素 a 和 b 的有序对表示成(a, b)，可以形式上定义成集合$(a, b) = \{a, \{a, b\}\}$。因此有序对(a, b)跟有序对(b, a)不同。 1073

两个集合 A 和 B 的笛卡儿积表示成 $A \times B$，它是所有对中第一个元素是 A 中元素，第二个元素是 B 中元素的有序对的集合。更正式的表示是，

$$A \times B = \{(a,b) : a \in A \text{ 且 } b \in B\}$$

例如，$\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$。当 A 和 B 是有限集合时，它们笛卡儿积的势是

$$|A \times B| = |A| \cdot |B| \tag{B.4}$$

n 个集合 A_1，A_2，…，A_n 的笛卡儿积是 n 元组的集合

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \cdots, a_n) : a_i \in A_i, i = 1, 2, \cdots, n\}$$

如果其中的每一个集合都是有限的话，它的势是

$$|A_1 \times A_2 \times \cdots \times A_n| = |A_1| \cdot |A_2| \cdots |A_n|$$

把一个集合 A 的 n 重笛卡儿积表示成集合

$$A^n = A \times A \times \cdots \times A$$

如果 A 是有限的，则它的势是 $|A^n| = |A|^n$。一个 n 元组也可以看成一个长度为 n 的有限序列(参见 B.3 节)。

练习

B.1-1 利用文氏图解释分配律(B.1)的第一个公式。

B.1-2 证明适用于任意有限个集合的广义德·摩根律：

$$\overline{A_1 \cap A_2 \cap \cdots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_n}$$
$$\overline{A_1 \cup A_2 \cup \cdots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}$$

*B. 1-3　证明等式(B. 3)的推广，称为容斥原理：

$$\begin{aligned}|A_1 \cup A_2 \cup \cdots \cup A_n| = & \\ & |A_1|+|A_2|+\cdots+|A_n| \\ & -|A_1 \cap A_2|-|A_1 \cap A_3|-\cdots \quad \text{(所有对)} \\ & +|A_1 \cap A_2 \cap A_3|+\cdots \quad \text{(所有元组)} \\ & \vdots \\ & +(-1)^{n-1}|A_1 \cap A_2 \cap \cdots \cap A_n|\end{aligned}$$

1074

B. 1-4　证明奇自然数集合是可数的。

B. 1-5　证明对任何有限集合 S，幂集 2^S 有 $2^{|S|}$ 个元素(即集合 S 有 $2^{|S|}$ 个不同的子集)。

B. 1-6　通过扩展有序对的集合理论定义，为 n 元组提供一个归纳定义。

B. 2　关系

两个集合 A 和 B 上的二元关系 R 是笛卡儿积 $A \times B$ 的一个子集。如果 $(a, b) \in R$，可以写作 aRb。R 是集合 A 上的二元关系，也就是说 R 是 $A \times A$ 的一个子集。例如，自然数集上的“小于”关系是集合 $\{(a, b): a, b \in \mathbf{N}, a<b\}$。集合 A_1，A_2，…，A_n 上的 n 元关系是 $A_1 \times A_2 \times \cdots \times A_n$ 的子集。

对所有的 $a \in A$，如果 aRa，则称二元关系 $R \subseteq A \times A$ 是自反的。例如，“=”和“≤”是自然数上的自反关系，但“<”则不是。对所有的 a，$b \in A$，如果当 aRb 时有 bRa，则称关系 R 是对称的。例如，“=”是对称的，而“<”与“≤”都不是。对所有 a，b，$c \in A$，如果当 aRb 且 bRc 时有 aRc，则称关系 R 是传递的。例如，关系“<”，“≤”和“=”都是传递的，但 $R = \{(a, b): a, b \in \mathbf{N}, a=b-1\}$ 不是，因为从 3R4 和 4R5 不能推出 3R5。

如果一个关系是自反的、对称的、传递的，则称其为等价关系。例如，“=”是自然数集上的等价关系，但“<”不是。如果 R 是集合 A 上的等价关系，那么对 $a \in A$，a 的等价类是集合 $[a] = \{b \in A: aRb\}$，即所有跟 a 等价的元素的集合。例如，我们定义 $R = \{(a, b): a, b \in \mathbf{N}, a+b$ 为偶数$\}$，因为 $a+a$ 是偶数(自反)，如果 $a+b$ 是偶数则 $b+a$ 是偶数(对称)，如果 $a+b$ 是偶数且 $b+c$ 是偶数则 $a+c$ 是偶数(传递)，因此 R 是等价关系。4 的等价类是

1075

$[4]=\{0, 2, 4, 6, \cdots\}$，3 的等价类是 $[3]=\{1, 3, 5, 7, \cdots\}$。关于等价类的一个基本定理如下：

定理 B. 1(等价关系相当于划分)　集合 A 上任一等价关系 R 的等价类构成一个对 A 的划分，A 上的任一划分确定 A 上的一个等价关系，划分中的集合是等价类。

证明：作为证明的第一部分，必须证明 R 的等价类是非空的、两两不相交的集合的并是 A。因为 R 是自反的，$a \in [a]$，所以等价类是非空的；更进一步，因为每个元素 $a \in A$ 属于等价类 $[a]$，等价类的并是 A。还要证明等价类是两两不相交的，即如果两个等价类 $[a]$、$[b]$ 有一个公共元素 c，则它们是相同的集合。既然 aRc、bRc，根据对称和传递，可知 aRb。因此，对任意元素 $x \in [a]$，如果 xRa 则 xRb，从而 $[a] \subseteq [b]$。类似可得 $[b] \subseteq [a]$，从而 $[a]=[b]$。

作为证明的第二部分，令 $A=\{A_i\}$ 为 A 的一个划分，并定义关系 $R=\{(a, b)$：存在 i 使 $a \in A_i$ 且 $b \in A_i\}$。令 R 是集合 A 上的等价关系。根据自反性，对 $a \in A_i$ 有 aRa。根据对称性，因为如果 aRb 则 a 和 b 在同一集合 A_i 中，所以 bRa。如果 aRb 且 bRc，那么这三个元素在同一集

合中，从而 aRc，传递性成立。如果 $a\in A_i$，对 $x\in[a]$有 $x\in A_i$，对 $x\in A_i$ 有 $x\in[a]$，可见划分中的集合是 R 的等价类。■

如果对 aRb 且 bRa 有 $a=b$，则集合 A 上的二元关系 R 是反对称的。例如，因为对 $a\leqslant b$ 且 $b\leqslant a$ 有 $a=b$，则自然数上的"$\leqslant$"关系是反对称的。一个具有自反、反对称和传递性质的关系称为偏序。具有偏序的集合称为偏序集。例如，关系"是……的后代"是全体人集合上的一个偏序（如果将个体称为自己的后代）。

在偏序集合 A 中可能没有单独的"最大"元素 a 使得对所有的 $b\in A$ 有 bRa。相反，可能存在多个最大元素 a，使得不存在 $b\in A$ 且 $b\neq a$ 满足 aRb。例如，不同尺寸盒子的集合，可能存在许多不能放进其他任何盒子的最大尺寸盒子，但不存在单独的任何盒子都能放进的"最大"盒子。[⊖] 1076

如果对所有的 a，$b\in A$ 有 aRb 或 bRa，即 A 中的每对元素都满足关系 R，则集合 A 上的偏序 R 是全序的或线性序的。例如，关系"$\leqslant$"是自然数上的全序，但"是……的后代"关系不是全体人集合上的全序，因为存在都不是对方后代的两个个体。

练习

B.2-1 证明：所有 Z 子集上的子集关系"$\subseteq$"是偏序而不是全序。

B.2-2 证明对任意正整数 n，关系"等价模 n"是整数上的等价关系。（如果存在一个整数 q 使得 $a-b=qn$，有 $a\equiv b(\mathrm{mod}\,n)$）。这个关系将整数划分成怎样的等价类？

B.2-3 举出满足下面关系的例子：

a）自反、对称，但不传递，

b）自反、传递，但不对称，

c）对称、传递，但不自反。

B.2-4 令 S 是有穷集，R 是 $S\times S$ 上的等价关系。证明如果 R 又是反对称的，那么根据 R 获得的 S 的等价类是单元集。

B.2-5 Narcissus 教授宣称如果关系 R 是对称、传递的，那么它也是自反的。他给出以下证明。根据对称性，如果 aRb 则 bRa。再根据传递性，有 aRa。这个教授正确吗？

B.3 函数

给定两个集合 A 和 B，函数 f 是 $A\times B$ 上的一个二元关系，且对所有的 $a\in A$ 仅存在一个 $b\in B$ 使得 $(a,b)\in f$。集合 A 称为函数 f 的定义域，B 称为函数 f 的陪域。我们有时写成 $f:A\to B$；当 $(a,b)\in f$，因为 b 由 a 的选择唯一确定，可写成 $b=f(a)$。 1077

直观上，函数 f 给 A 中的每个元素指派一个 B 中的元素。不可以把 B 中两个不同的元素指派给 A 中的同一个元素，但 B 中的同一个元素可以指派给 A 中两个不同的元素。例如，二元关系

$$f=\{(a,b):a,b\in \mathrm{N}\text{ 且 }b=a \bmod 2\}$$

是一个函数 $f:\mathrm{N}\to\{0,1\}$，因为对任一自然数 a，$\{0,1\}$ 中仅存在一个元素使得 $b=a\mathrm{mod}2$。在这个例子中，$0=f(0)$，$1=f(1)$，$0=f(2)$ 等。相反，二元关系

$$g=\{(a,b):a,b\in \mathrm{N}\text{ 且 }a+b\text{ 为偶数}\}$$

不是一个函数，因为(1，3)和(1，5)都在 g 中，因此对于 $a=1$ 存在不止一个 b 使得 $(a,b)\in g$。

给定一个函数 $f:A\to B$，如果 $b=f(a)$，则称 a 为 f 的自变量，b 为函数 f 对应于 a 的值。

⊖ 为准确起见，为了能将"放入"关系看作一个偏序，必须将盒子看作可以放入它本身。

可以通过列出函数定义域每个元素对应的值来定义一个函数。例如，对 $n\in\mathbf{N}$ 定义 $f(n)=2n$，这意味着 $f=\{(n,\ 2n):\ n\in\mathbf{N}\}$。如果两个函数 f，g 有相同的定义域和陪域，且对于定义域中的所有元素 a 有 $f(a)=g(a)$，则称它们相等。

长度为 n 的有穷序列是一个定义域为 n 个整数集合 $\{0,\ 1,\ \cdots,\ n-1\}$ 的函数。我们通常列举值来表示一个有穷序列 $\langle f(0),\ f(1),\ \cdots,\ f(n-1)\rangle$。一个无穷序列是一个定义域为自然数集合 $\mathbf{N}$ 的函数。例如，通过递归(3.21)定义的斐波那契序列是无穷序列 $\langle 0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ \cdots\rangle$。

如果一个函数 f 的定义域是一个笛卡儿积，通常可以省略 f 参数两边的括号。例如，函数 $f:\ A_1\times A_2\times\cdots\times A_n\to B$ 可以写成 $b=f(a_1,\ a_2,\ \cdots,\ a_n)$ 来代替 $b=f((a_1,\ a_2,\ \cdots,\ a_n))$。我们可以将每个 a_i 称为函数 f 的一个参数，尽管实际上 f 的参数是一个 n 元组 $(a_1,\ a_2,\ \cdots,\ a_n)$。

如果 $f:\ A\to B$ 是函数且 $b=f(a)$，则称 b 是 a 在函数 f 中的像。集合 $A'\subseteq A$ 在函数 f 中的像可以定义为

$$f(A')=\{b\in B: b=f(a),\text{对某个 } a\in A'\}$$

函数 f 的值域是它定义域的像，即 $f(A)$。例如，由 $f(n)=2n$ 定义的函数 $f:\ \mathbf{N}\to\mathbf{N}$ 的值域是 $f(\mathbf{N})=\{m:\ m=2n \text{ 对某些 } n\in\mathbf{N}\}$。

如果函数 f 的值域就是它的定义域，则称其为满射函数。例如，函数 $f(n)=\lfloor n/2\rfloor$ 是从 $\mathbf{N}$ 到
1078 $\mathbf{N}$ 的满射，因为对每个 $\mathbf{N}$ 中元素都会作为 f 的某个参数的值出现。相反，函数 $f(n)=2n$ 不是从 $\mathbf{N}$ 到 $\mathbf{N}$ 的满射，因为没有一个参数的值会是 3。函数 $f(n)=2n$ 是从自然数到偶数的满射。满射 $f:\ A\to B$ 有时描述为将 A 映射到 B。有时我们说 f 是映射，是指 f 是满射。

如果函数 $f:\ A\to B$ 对不同的参数有不同的值，即如果 $a\neq a'$ 则 $f(a)\neq f(a')$，那么 f 是单射的。例如，函数 $f(n)=2n$ 从 $\mathbf{N}$ 到 $\mathbf{N}$ 是单射的，因为每个偶数 b 是 f 定义域中至多一个元素的像，即 $b/2$。函数 $f(n)=\lfloor n/2\rfloor$ 不是单射，因为参数 2 和 3 都可以得到值 1。单射有时也称为一对一函数。

如果函数 $f:\ A\to B$ 是单射的和满射的，则其为双射的。例如，函数 $f(n)=(-1)^n\lceil n/2\rceil$ 是从 $\mathbf{N}$ 到 $\mathbf{Z}$ 的双射：

$$\begin{aligned}0&\to 0\\1&\to -1\\2&\to 1\\3&\to -2\\4&\to 2\\&\vdots\end{aligned}$$

因为 $\mathbf{Z}$ 中不存在作为 $\mathbf{N}$ 中超过 1 个元素像的元素，因此函数是单射的。因为 $\mathbf{Z}$ 中每个元素都是 $\mathbf{N}$ 中某个元素的像，因此函数是满射的。从而，函数是双射的。双射有时称为一一对应，因为它使定义域和陪域中的元素配对。从集合 A 到它本身的双射有时也称为置换。

如果一个函数 f 是双射的，它的*逆* f^{-1} 定义为

$$f^{-1}(b)=a \text{ 当且仅当 } f(a)=b$$

例如，函数 $f(n)=(-1)^n\lceil n/2\rceil$ 的逆是

$$f^{-1}(m)=\begin{cases}2m & \text{如果 } m\geqslant 0\\ -2m-1 & \text{如果 } m<0\end{cases}$$

练习

B.3-1 令 A 和 B 为有穷集，$f: A \to B$ 为函数。证明

a)如果 f 是单射的，则 $|A| \leqslant |B|$；

b)如果 f 是满射的，则 $|A| \geqslant |B|$。

B.3-2 当定义域和陪域都是 N 时，函数 $f(x)=x+1$ 是双射吗？当定义域和陪域都是 Z 时，是双射吗？

1079

B.3-3 给出一个二元关系逆的自然定义，使得如果一个关系实际是一个双射函数时，它的关系逆就是它的函数逆。

B.3-4 给出一个从 Z 到 Z×Z 的双射。

B.4 图

这部分讨论两种图：有向的和无向的。某些文献中的定义与本文不同，但大部分差异很小。22.1 节说明了图在计算机内存中是如何表示的。

有向图 G 是一对(V, E)，其中 V 是有穷集，E 是 V 上的二元关系。V 集称为 G 的顶点集合，它的元素称为顶点。E 集称为 G 的边集合，它的元素称为边。图 B.2a 是顶点{1，2，3，4，5，6}上有向图的图示。顶点用圆圈表示，边用箭头表示。注意自身环，即从顶点到自身的边，是可能的。

在一个无向图 $G=(V, E)$中，边集 E 由无序顶点对而不是有序顶点对组成。即边是一个集合$\{u, v\}$，其中 $u, v \in V$ 且 $u \neq v$。我们习惯使用(u, v)而不是$\{u, v\}$来表示一条边，(u, v)和(v, u)视为同一条边。在无向图中不允许自身环，因此每个边由两个不同的顶点组成。图 B.2b 是顶点集{1，2，3，4，5，6}上无向图的图示。

尽管有向图和无向图的某些定义不同，但大多数定义都是一致的。如果(u, v)是有向图 $G=(V, E)$的一条边，则称(u, v)离开顶点 u，进入顶点 v。例如，在图 B.2a 中，离开顶点 2 的边是(2，2)，(2，4)和(2，5)。进入顶点 2 的边是(1，2)和(2，2)。如果(u, v)是无向图 $G=(V, E)$的一条边，则称(u, v)与顶点 u、v 关联。在图 B.2b 中，与顶点 2 关联的边为(1，2)和(2，5)。

如果(u, v)是图 $G=(V, E)$的一条边，则称顶点 v 与顶点 u 相邻。如果图是无向的，则邻接关系是对称的，在有向图中，则邻接关系不一定对称。在有向图中，如果 v 与 u 相邻，有时记为 $u \to v$。在图 B.2a、b，顶点 2 与顶点 1 相邻，因为边(1，2)属于这两幅图。在图 B.2a 中，顶点 1 与顶点 2 不相邻，因为边(2，1)不在图中。

1080

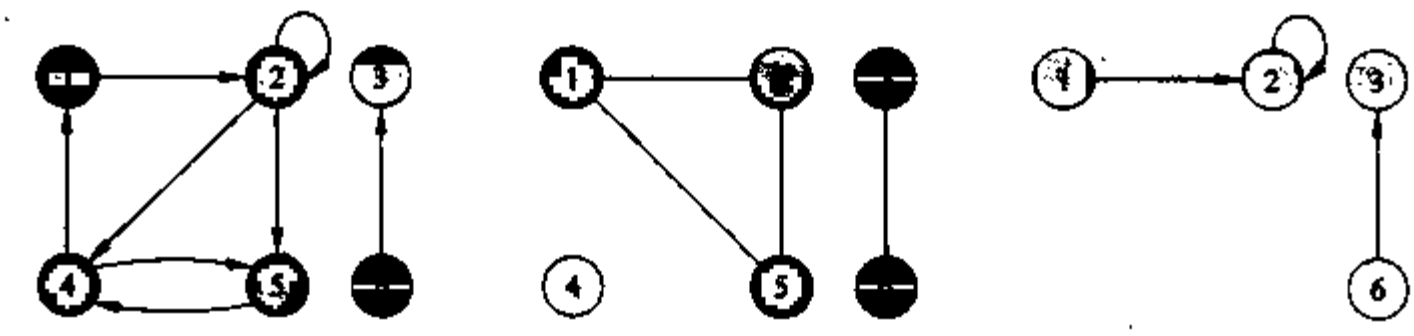
在无向图中，一个顶点的度是指与之关联的边的条数。例如，图 B.2b 中，顶点 2 的度为 2。如果一个顶点的度为 0，则称其是孤立的，如图 B.2b 中的顶点 4。在有向图中，顶点的出度是以它为起点的边的条数，入度是以它为终点的边的条数。在有向图中，顶点的度等于该顶点的入度与出度之和。在图 B.2a 中，顶点 2 的入度是 2，出度是 3，度是 5。

在图 $G=(V, E)$中，从顶点 u 到顶点 u' 且长度为 k 的路径是顶点序列$(v_0, v_1, v_2, \cdots, v_k)$，并满足 $u=v_0$，$u'=v_k$ 且对 $i=1, 2, \cdots, k$ 有$(v_{i-1}, v_i) \in E$。路径的长度是路径中边的条数。路径包含顶点 $v_0, v_1, \cdots, v_k$ 和边(v_0, v_1)，(v_1, v_2)，$\cdots$，(v_{k-1}, v_k)。(存在从 u 到 u 的长度为 0 的路径)。如果存在一条从 u 到 u' 的路径 p，则称 u' 是从 u 经由 p 可达的，如果 G 是有向的，可以记为 $u \overset{p}{\rightsquigarrow} u'$。如果路径上各顶点均不重复，则称这样的路径为简单路径。在图 B.2a 中，路径(1，2，5，4)是长度为 3 的简单路径。路径(2，5，4，5)不是简单路径。

路径 $p=\langle v_0, v_1, \cdots, v_k\rangle$的子路径是它的顶点的一个连续子序列。亦即，对任意 $0\leqslant i\leqslant j\leqslant k$，顶点$\langle v_i, v_{i+1}, \cdots, v_j\rangle$的子序列是 p 的子路径。

在有向图中，如果 $v_0=v_k$ 且路径至少包含一条边，则称路径$\langle v_0, v_1, \cdots, v_k\rangle$形成回路。如果 $v_1, v_2, \cdots, v_k$ 互不相同，则称回路为简单回路。自身环是长度为 1 的回路。如果存在整数 j 使得对$i=0, 1, \cdots, k-1$ 有 $v'_i=v_{(i+j) \bmod k}$，则路径$\langle v_0, v_1, v_2, \cdots, v_{k-1}, v_0\rangle$与路径$\langle v'_0, v'_1, v'_2, \cdots, v'_{k-1}, v'_0\rangle$形成相同的回路。在图 B.2a 中，路径$\langle 1, 2, 4, 1\rangle$与路径$\langle 2, 4, 1, 2\rangle$和路径$\langle 4, 1, 2, 4\rangle$形成相同的回路。这个回路是简单的，而$\langle 1, 2, 4, 5, 4, 1\rangle$则不是简单回路。由边$\langle 2, 2\rangle$形成的回路$\langle 2, 2\rangle$是一个自身环。一个不存在自身环的有向图称为简单图。在无向图中，如果 $k\geqslant 3$，$v_0=v_k$ 且 $v_1, v_2, \cdots, v_k$ 互不相同，则路径$\langle v_0, v_1, \cdots, v_k\rangle$形成(简单)回路。例如，在图 B.2b 中，路径$\langle 1, 2, 5, 1\rangle$是回路。不存在回路的图是无回路图。

如果无向图的每对顶点都有路径相连，则称其为连通图。在“可达”关系下，顶点的等价类称为图的连通分支。图 B.2b 中的图有三个连通分支：$\{1, 2, 5\}$，$\{3, 6\}$和$\{4\}$。$\{1, 2, 5\}$中的每个顶点对其他顶点都是可达的。如果无向图仅有一个连通分支，即每个顶点都是其他顶点可达的，则称其为连通的。



a) b) c)

图 B.2 有向图和无向图。a)有向图 $G=(V, E)$，其中 $V=\{1, 2, 3, 4, 5, 6\}$，$E=\{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$。边$(2, 2)$是自身环。b)无向图 $G=(V, E)$，其中 $V=\{1, 2, 3, 4, 5, 6\}$，$E=\{(1, 2), (1, 5), (2, 5), (3, 6)\}$。顶点 4 是孤立点。c)在 a)中顶点集$\{1, 2, 3, 6\}$构成的子图

如果有向图中的每对顶点都相互可达，则称其为强连通图。在“相互可达”关系下顶点的等价类称为有向图的强连通分支。如果有向图仅有一个强连通分支，则称其是强连通的。图 B.2a 中的图有三个强连通分支：$\{1, 2, 4, 5\}$，$\{3\}$和$\{6\}$。$\{1, 2, 4, 5\}$中的顶点对是相互可达的。因为顶点 6 不能从顶点 3 到达，则顶点集$\{3, 6\}$不形成强连通分支。

如果存在双射 $f: V\to V'$，使得当且仅当$(f(u), f(v))\in E'$时$(u, v)\in E$，则称图 $G=(V, E)$与图 $G'=(V', E)$同构。换句话说，我们可以在保持 G 和 G'中相应边的情况下，将 G 中的顶点重新标注成 G'中的顶点。图 B.3a 显示了一对同构图 G 和 G'，它们的顶点集分别为 $V=\{1, 2, 3, 4, 5, 6\}$和 $V'=\{u, v, w, x, y, z\}$。从 V 到 V'的映射 $f(1)=u$，$f(2)=v$，$f(3)=w$，$f(4)=x$，$f(5)=y$，$f(6)=z$ 是所要求的双射函数。图 B.3b 中的图不是同构的。尽管两个图都有 5 个顶点和 7 条边，但上面的图有度为 4 的顶点，而下面的图没有。

如果 $V'\subseteq V$ 且 $E'\subseteq E$，则称图 $G'=(V', E')$是图 $G=(V, E)$的子图。给定集合 $V'\subseteq V$，G 关于 V'的子图是图 $G'=(V', E')$，其中 $E'=\{(u, v)\in E: u, v\in V'\}$。图 B.2a 关于顶点集$\{1, 2, 3, 6\}$的子图显示在图 B.2c 中，它的边集是$\{(1, 2), (2, 2), (6, 3)\}$。

1081
~
1082

给定一个无向图 $G=(V, E)$，它的有向版本是有向图 $G'=(V, E')$，其中$(u, v)\in E'$当且仅当$(u, v)\in E$。即图 G 中的每条无向边(u, v)在有向图中被两条有向边(u, v)和(v, u)代替。

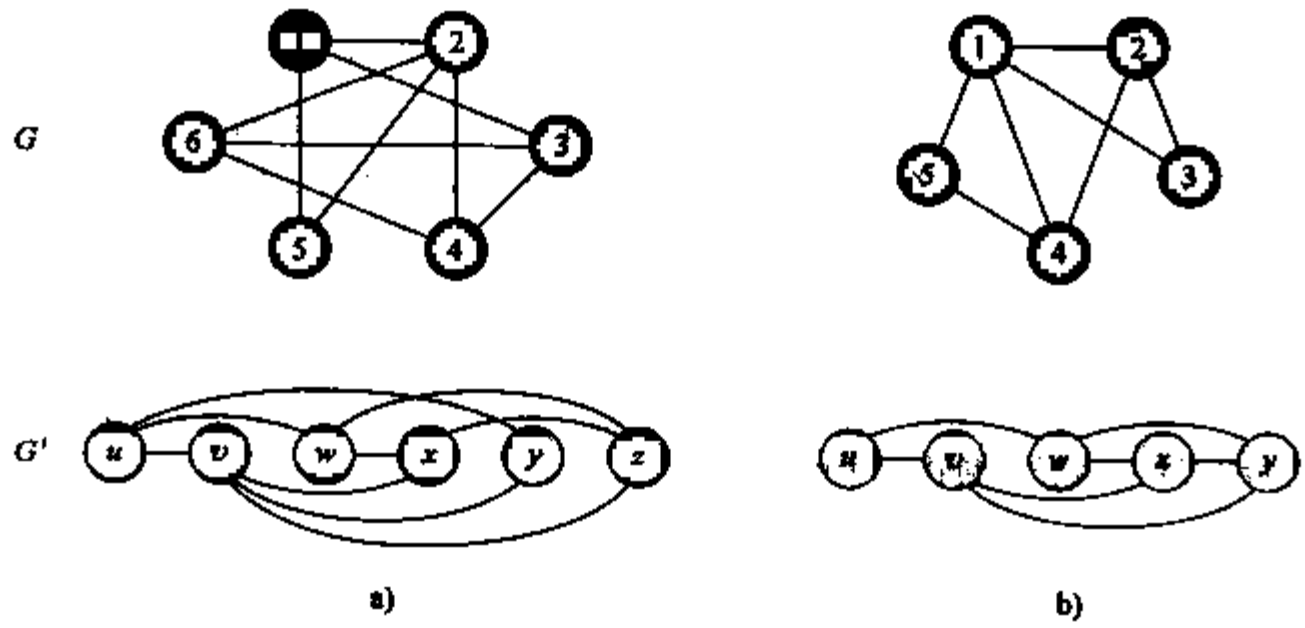


图 B.3 a)两个同构的图。上图中的顶点按以下的关系式映射至下图中的顶点：$f(1)=u$，$f(2)=v$，$f(3)=w$，$f(4)=x$，$f(5)=y$，$f(6)=z$。b)非同构的两个图，因为上图中有一个度为 4 的顶点，而下图中没有这样的顶点

给定一个有向图 $G=(V, E)$，它的无向版本是无向图 $G'=(V, E')$，其中 $(u, v)\in E'$ 当且仅当 $(u, v)\in E$ 且 $u\neq v$。即无向图包含 G 中"去掉方向"的边，并去除了自身环。(因为在无向图中 (u, v) 和 (v, u) 是同一条边，因此即使有向图包含 (u, v) 和 (v, u)，无向图也只能包含它们中的一条。)在有向图 $G=(V, E)$ 中，顶点 u 的邻居是 G 的无向版本中与 u 邻接的顶点。即，如果 $(u, v)\in E$ 或 $(v, u)\in E$ 则 v 是 u 的邻居。在无向图中，如果 u 和 v 邻接，则它们是邻居。

几种图有特殊的名称。每对顶点都邻接的无向图称为完全图。如果无向图 $G=(V, E)$ 的顶点集 V 可以划分成两个集合 V_1，V_2，使得对 $(u, v)\in E$ 有 $u\in V_1$，$v\in V_2$，或者 $u\in V_2$，$v\in V_1$，则称图 G 为二分图。即所有的边都位于两个集合 V_1 和 V_2 之间。无回路的无向图称为森林。连通的、无回路的无向图是(自由)树(见 B.5 节)。通常取"有向无回路图"的单词首字母，称为 dag。

我们偶尔会遇到图的两种变种。多重图与无向图类似，但它可以在顶点之间存在多重边和自身环。超图与无向图类似，但每个超边不止连接两个顶点，而是连接顶点的任意子集。许多为普通有向和无向图而写的算法也可以应用到这些类似图的结构。

沿边 $e=(u, v)$ 收缩无向图 $G=(V, E)$ 得图 $G'=(V', E')$，其中 $V'=V-\{u, v\}\cup\{x\}$，x 是新顶点。边集 E' 通过从 E 中删除边 (u, v)，对每个射到 u 或 v 的顶点 w 删除 (u, w) 和 (v, w)，添加新边 (x, w) 得到。

练习

B.4-1 在一个教职员工的聚会中，与会者相互握手以示敬意。每个教授记得他或她自己握手的次数。会议结束后，系领导将每个教授握手的次数相加。通过证明握手引理说明结果是偶数。握手引理是：如果 $G=(V, E)$ 是无向图，那么 $\sum_{v\in V}\text{degree}(v)=2|E|$。

B.4-2 证明如果有向或无向图在两个顶点 u、v 间有一条路径，则 u、v 之间存在一条简单路径。证明如果有向图有一个回路，则它包含一条简单回路。

B.4-3 证明任意连通的无向图 $G=(V, E)$ 满足 $|E|\geqslant|V|-1$。

B.4-4 证明在无向图中，"可达"关系是图中顶点上的等价关系。等价关系三个性质中的哪些性

1083

质对有向图顶点上的“可达”关系成立？

B.4-5　图 B.2a 中有向图的无向版本是什么？图 B.2b 中无向图的有向版本是什么？

*B.4-6　证明如果令超图中的关联关系对应于二分图中的邻接关系，则超图可以表示成二分图。（提示：令二分图中的一个顶点集对应超图中的顶点集，令二分图中的另一个顶点集对应超边。）

1084

B.5　树

跟图一样，树有许多相关的，但差别很小的概念。这节介绍几种树的定义和数学性质。10.4 节和 22.1 节描述了树在计算机内存中的表示。

B.5.1　自由树

正如 B.4 节的定义，自由树是一个连通的，无回路的无向图。当我们称一个图是树时，通常忽略形容词“自由”。如果一个无向图是无回路的但可能是非连通的，称为森林。许多对树有效的算法对森林同样有效。图 B.4a 是一个自由树，图 B.4b 是一个森林。图 B.4b 中的森林不是树，因为它不是连通的。图 B.4c 既不是树也不是森林，因为它包含回路。

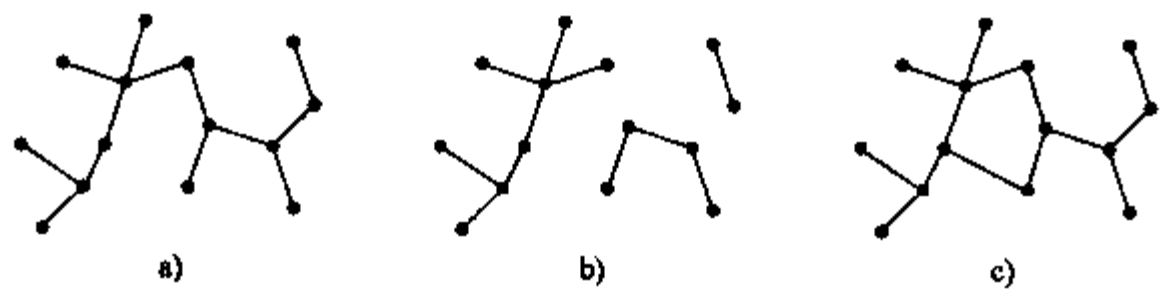


图 B.4　a)自由树　b)森林　c)此图包含回路，因此既不是树也不是森林

下面的定理展现了自由树的许多重要事实。

定理 B.2(自由树性质)

令 $G=(V, E)$为一个无向图。下面的表述是等价的。

1)G 是自由树。

2)G 中任意两个顶点由唯一一条简单路径相连。

3)G 是连通的，但从 E 中去掉任何边后得到的图都是非连通的。

4)G 是连通的，且 $|E| = |V| - 1$。

5)G 是无回路的，且 $|E| = |V| - 1$。

6)G 是无回路的，但添加任何边到 E 中后得到的图包含回路。

证明：1)⇒2)：树是连通的，因此 G 中任意两个顶点由至少一条简单路径相连。如图 B.5 所示，令 u、v 为两个顶点，它们由两条不同的简单路径 p_1、p_2 相连。令 w 为两条路径首次分叉的顶点，即 w 是第一个既在 p_1 又在 p_2 上，且在 p_1 上的后继顶点 x 与它在 p_2 上的后继顶点 y 满足 $x \neq y$ 的顶点。令 z 为两条路径首次交汇的顶点，即 z 是继 w 之后第一个既在 p_1 上又在 p_2 上的顶点。令 p' 是 p_1 从 w 经过 x 到 z 的子路径，p'' 是路径 p_2 从 w 经过 y 到 z 的子路径。p' 和 p'' 除了端点外不共享顶点。因此，通过连接 p' 和反向 p'' 得到的路径是一个回路。这跟假设 G 是树矛盾。因此，如果 G 是树，那么在两个顶点间至多存在一条简单路径。

2)⇒3)：如果 G 中的任意两个顶点由唯一一条简单路径相连，那么 G 是连通的。令 (u, v) 为 E 中的任意边。这条边是从 u 到 v 的一条路径，因此它必须是从 u 到 v 的唯一路径。如果我们将 (u, v) 从 G 中去掉，就不存在从 u 到 v 的路径，从而 G 非连通。

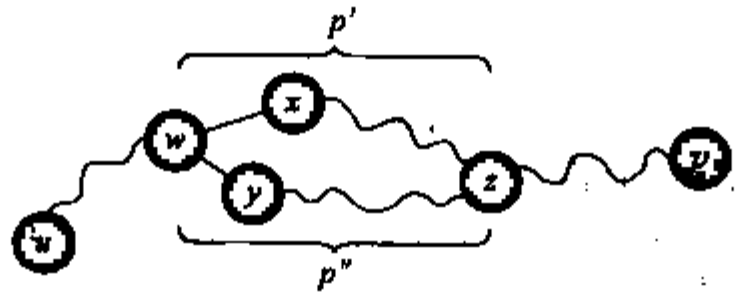


图 B.5 定理 B.2 证明中的一步：如果 1)G 是自由树，那么 2)G 中的任意两个顶点有唯一一条简单路径相连。假设顶点 u 和 v 由两个不同的简单路径 p_1、p_2 相连，两条路径在顶点 w 处首次分叉，在顶点 z 处重新交汇。连接 p' 和反向 p'' 得到的路径是一个回路，这跟假设 G 是树矛盾

3)⇒4)：假设图 G 是连通的，根据练习 B.4-3 有 $|E|\geqslant|V|-1$。我们还需通过归纳证明 $|E|\leqslant|V|-1$。有 $n=1$ 或 $n=2$ 个顶点的连通图有 $n-1$ 条边。假设 G 有 $n\geqslant3$ 个顶点，所有少于 n 个顶点且满足 3)的图也满足 $|E|\leqslant|V|-1$。从 G 中移除任意一条边将图分成 $k\geqslant2$ 个连通分支(实际上 $k=2$)。每个子图都满足 3)，否则 G 就不会满足 3)。因此，通过归纳，所有分支中边的条数至多是 $|V|-k\leqslant|V|-2$。再加上去掉的那条边，可得 $|E|\leqslant|V|-1$。

4)⇒5)：假设 G 是连通的且 $|E|=|V|-1$。我们必须证明 G 是无回路的。假设 G 有一个包含 k 个顶点 v_1，v_2，…，v_k 的回路，不失一般性，假设这是一个简单回路。令 $G_k=(V_k, E_k)$ 为这个回路构成的 G 的子图。注意 $|V_k|=|E_k|=k$。如果 $k<|V|$，因为 G 是连通的，则必然存在一个顶点 $v_{k+1}\in V-V_k$ 跟某个顶点 $v_i\in V_k$ 邻接。定义 $G_{k+1}=(V_{k+1}, E_{k+1})$ 为 G 的子图，且满足 $V_{k+1}=V_k\cup\{v_{k+1}\}$，$E_{k+1}=E_k\cup\{(v_i, v_{k+1})\}$。注意 $|V_{k+1}|=|E_{k+1}|=k+1$。如果 $k+1<|V|$，就继续以同样的方式定义 G_{k+2} 等等，直到得到 $G_n=(V_n, E_n)$ 满足 $n=|V|$，$V_n=V$，$|E_n|=|V_n|=V$。因为 G_n 是 G 的子图，可知 $E_n\subseteq E$，从而 $|E|\geqslant|V|$，这跟假设 $|E|=|V|-1$ 矛盾，从而 G 是无回路的。

5)⇒6)：假设 G 是无回路的且 $|E|=|V|-1$。令 k 为 G 中连通分支的个数。从定义看，每个连通分支都是一棵自由树，并且由 1)可得 5)，G 中所有连通分支的所有边的数目是 $|V|-k$。因此，k 必须为 1，G 实际上是一个树。因为由 1)可得 2)，G 中的任意两个顶点由唯一一条简单路径相连。因此，添加任何边到 G 中都会形成回路。

6)⇒1)：假设 G 是无回路的，但添加任何边到 E 中都会形成回路。我们必须证明 G 是连通的。令 u 和 v 是 G 中的任意顶点。如果 u 和 v 不邻接，则添加边 (u, v) 将形成回路，回路中除 (u, v) 外的其他边都属于 G。从而，u 和 v 之间存在路径。因为 u 和 v 是任意选择的，因此 G 是连通的。 ■

B.5.2 有根树和有序树

有根树是一棵自由树，它有一个与其他点不同的结点。这个特殊的顶点称为树的根。我们通常称有根树的顶点为结点[⊖]。图 B.6a 显示一棵有 12 个结点且根为结点 7 的有根树。

考虑根为 r 的有根树 T 中的一个结点 x。任何在从 r 到 x 的唯一路径上的结点 y 称为 x 的一个祖先。如果 y 是 x 的一个祖先，则 x 是 y 的一个子孙。(每个结点都是它自己的祖先和子孙。)如果 y 是 x 的祖先，且 $x\neq y$，则称 y 是 x 的真祖先，x 是 y 的真子孙。以 x 为根的子树是由 x 的子孙构成的，x 是树根。例如，图 B.6a 以结点 8 为根的子树包含结点 8、6、5、9。

⊖ 在图论中，术语"结点"通常用作"顶点"的同义词。这里我们将它专用来表示有根树中的顶点。

1085
~
1086

1087 从树 T 的根 r 到结点 x 的路径中最后一条边是(y, x)，因此 y 是 x 的双亲，x 是 y 的子女。根是 T 中唯一没有双亲的结点。如果两个结点有同样的双亲，则为兄弟结点。没有子女的结点称为外部结点或叶结点。非叶结点称为内部结点。

有根树 T 中结点 x 的子女数目称为 x 的度。[⊖]从根 r 到结点 x 路径的长度称为 x 在 T 中的深度。结点在树中的高度是从结点向下到某个叶结点最长简单路径中边的条数，树的高度是其根的高度。树的高度也等于树中结点的最大深度。

有序树是子女结点有序的有根树。即，如果一个结点有 k 个子女，那么有第一子女，第二子女，…，第 k 子女。当看成有序树时，图 B.6 中两棵树不同，但如果仅看成有根树，两者是相同的。

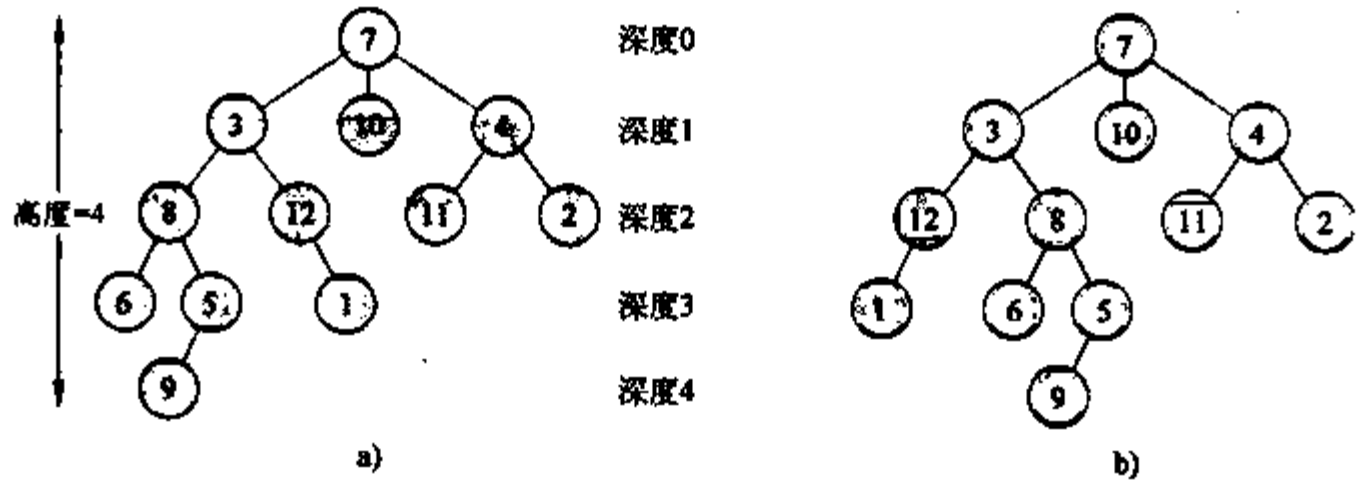


图 B.6　有根树和有序树。a)高度为 4 的有根树。这棵树按标准方式绘制：根(结点 7)在上面，它的子女(深度为 1 的结点)在它下面，根子女的子女(深度为 2 的结点)在根子女的下面，依此类推。对有序树而言，一个结点子女按从左到右顺序是有关联的；反之，不关联。b)另一棵有根树。作为一棵有根树，它跟 a)中的树相同，但作为有序树，它跟 a)不同，因为结点 3 的子女以不同顺序出现

B.5.3　二叉树与位置树

二叉树的定义是以递归形式给出的。二叉树 T 是定义在结点有限集上的结构，它或者不包
1088 含任何结点或者有三个不相交的结点集构成：根结点，一个称为左子树的二叉树和一个称为右子树的二叉树。

不包含任何结点的二叉树称为空树或零树，有时表示成 NIL。如果左子树非空，则它的根称为整棵树的根的左子女。同理，非空右子树的根称为整棵树的根的右子女。如果一个子树是空树 NIL，我们称子女缺失或丢失。图 B.7a 显示了一棵二叉树。

二叉树不是简单的每个结点的度至多为 2 的有序树。例如，在一棵二叉树中，如果一个结点仅有一个子女，那么它是左子女还是右子女是有关系的。在一棵有序树中，一个单独子女在左在右没有区别。因为结点位置不同，图 B.7b 中的二叉树与图 B.7a 中的二叉树不同。如果看成有序树，两者是相同的。

如图 B.7c 所示，二叉树的位置信息可以用有序树的内部结点来表示。思想是将二叉树中缺失的子女用没有子女的结点代替。这些叶结点在图中用正方形表示。这样形成一个满二叉树：每个结点或者是叶结点，或者度数为 2，不存在度为 1 的结点。因此，结点的子女顺序保存了位置信息。

⊖ 注意结点的度取决于 T 是有根树还是自由树。自由树中结点的度跟无向图中一样，是相邻顶点的个数。而在有根树中，度指结点孩子的个数，结点的双亲不包含在内。

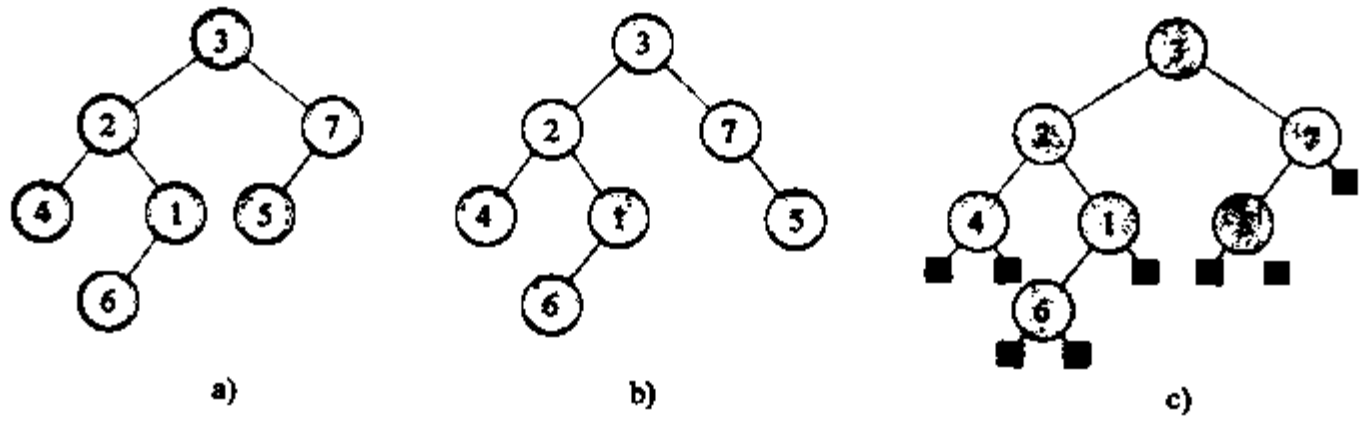


图 B.7 二叉树。a) 一棵按标准方式绘制的二叉树。结点左子女画在结点的左下方。结点的右子女画在结点的右下方。b)一棵与 a)中二叉树不同的二叉树。在 a)中，结点 7 的左子女是 5，右子女缺失。在 b)中，结点 7 的左子女缺失，右子女是 5。作为有序树，两者是相同的，但作为二叉树，它们不相同。c)在 a)中的二叉树由满二叉树的内部结点表示。满二叉树是每个内部结点度为 2 的有序树，树中的叶结点用正方形表示

1089

区分二叉树和有序树的位置信息可以扩展到每个结点有多于 2 个子女的树。在位置树中，结点的子女用不同的正整数标识。如果没有结点被标识成整数 i，则结点的第 i 个子女缺失。k 叉树是每个结点的标识超过 k 的子女均缺失的位置树。因此，二叉树是 $k=2$ 的 k 叉树。

完全 k 叉树是所有的叶结点都有相同深度，并且所有的内部结点度都为 k。图 B.8 显示了一棵高度为 3 的完全二叉树。一棵高度为 h 的完全 k 叉树有多少个叶结点？根结点有 k 个深度为 1 的子女，每个子女有 k 个深度为 2 的子女等等。因此，深度为 h 的叶结点个数是 k^h。从而有 n 个叶结点的完全 k 叉树的高度是 $\log_k n$。根据等式(A.5)高度为 h 的完全 k 叉树内部结点的个数是

$$1+k+k^2+\cdots+k^{h-1}=\sum_{i=0}^{h-1}k^i=\frac{k^h-1}{k-1}$$

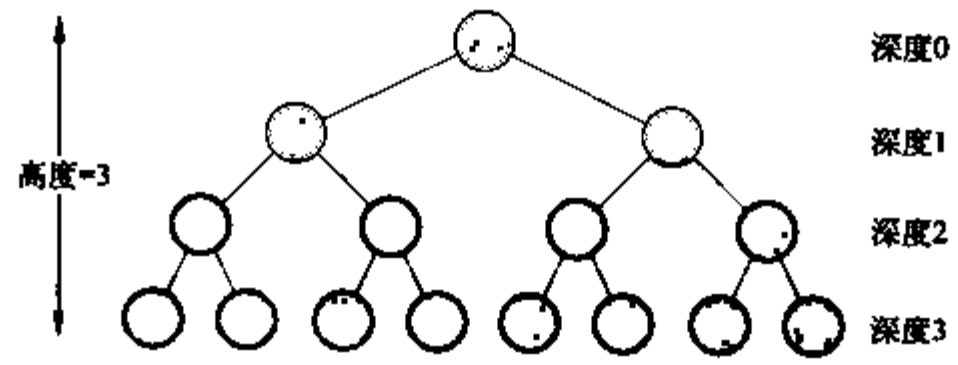


图 B.8 一棵高度为 3 且包含 8 个叶结点和 7 个内部结点的完全二叉树

因此，一棵完全二叉树有 2^h-1 个内部结点。

练习

B.5-1 画出所有由 3 个顶点 A，B，C 组成的自由树。画出所有由 3 个结点 A，B，C 组成且以 A 为根的有根树。画出所有由 3 个结点 A，B，C 组成且以 A 为根的有序树。画出所有由 3 个结点 A，B，C 组成且以 A 为根的二叉树。

1090

B.5-2 令 $G=(V, E)$ 为有向无回路图，且存在一个顶点 $v_0\in V$，使得从 v_0 到每个顶点 $v\in V$ 存在唯一一条路径。证明 G 的无向版本是一棵树。

B.5-3 通过归纳证明：在任何非空二叉树中，度为 2 的结点的个数比叶结点的个数少 1。

B.5-4 通过归纳证明：一棵有 n 个结点的非空二叉树的高度至少为 $\lfloor \lg n \rfloor$。

*B. 5-5　满二叉树的内部路径长度是所有内部结点的深度之和。同样，外部路径长度是所有叶结点的深度之和。考虑一棵有 n 个内部结点，内部路径长度为 i，外部路径长度为 e 的满二叉树。证明 $e=i+2n$。

*B. 5-6　让我们将“权重” $w(x)=2^{-d}$ 跟二叉树 T 中深度为 d 的每个叶结点 x 关联。证明 $\sum_x w(x)\leqslant 1$，这里的求和考虑 T 中的所有叶结点 x。(这个式子称为 Kraft 不等式。)

*B. 5-7　证明每个有 L 个叶结点的二叉树包含一棵有 $L/3$ 到 $2L/3$ 个叶结点的子树。

思考题

B-1　图着色

给定一个无向图 $G=(V, E)$，G 的 k 着色是函数 $c: V\to\{0, 1, \cdots, k-1\}$，满足对任意边 $(u, v)\in E$ 有 $c(u)\neq c(v)$。即数字 0，1，…，$k-1$ 代表 k 种颜色，并且相邻接的顶点必须是不同的颜色。

a)证明任意树是 2 可着色的。

1091　b)证明下列表述等价：

1)G 是二分的。

2)G 是 2 可着色的。

3)G 中没有奇数长度的回路。

c)令 d 是 G 中任意顶点的最大度数。证明 G 可用 $d+1$ 种颜色着色。

d)证明如果 G 有 $O(|V|)$ 条边，那么 G 可以用 $O(\sqrt{|V|})$ 种颜色着色。

B-2　友谊图

将下列表述改写成关于无向图的定理，并证明。假设友谊是对称的但不是自反的。

a)在任意有 $n\geqslant 2$ 个人的组中，有两个人在组内有相同数目的朋友。

b)在每个有 6 个人的组中，或者有 3 个人互为朋友，或者有 3 个人互不相识。

c)每个组中的人可以分成两个子组，满足每个属于某个子组的人有至少一半的朋友在另外一个子组。

d)如果一个组中的每个人都至少是组内一半人的朋友，那么这个组的组员可以围坐在一张桌子旁，且满足每个人都坐在两个朋友之间。

B-3　等分树

许多分治法的图算法都要求将图等分为两个大小相近的子图，这可以通过划分顶点来实现。这个问题研究通过去掉少量的边来等分树。我们要求如果当边去掉后，两个顶点在相同的子树中，那么它们必须在相同的划分中。

a)证明通过移除一条边，可以将 n 个顶点的二叉树的顶点划分成两个集合 A 和 B，且满足 $|A|\leqslant 3n/4$，$|B|\leqslant 3n/4$。

b)通过给出一个例子：一个简单二叉树在去掉一条边后，其最均匀平衡的划分满足

1092　$|A|=3n/4$，来证明在最差情况下(a)中的常数 3/4 是最优的。

c)证明：通过移除最多 $O(\lg n)$ 条边可以将任何 n 个顶点的二叉树划分成两个集合 A 和 B，且满足 $|A|=\lfloor n/2\rfloor$，$|B|=\lceil n/2\rceil$。

本章注记

G. Boole 是符号逻辑发展的先驱，他在一本 1854 年出版的书中引入了许多基本的集合表示。现代集合理论是由 G. Cantor 在 1874～1895 年间创立的。Cantor 主要关注无穷集合。术语“函数”是由 G. W. Leibniz 提出的，他使用“函数”来指代多种数学公式。他局限性的定义已经多次被推广。图论起始于 1736 年，当时 L. Euler 证明不可能经过 Königsberg 城中的七座桥中每座一次且仅一次并回到起点。

Harary[138] 是一本很有用的书，它给出了图论中的许多定义和结论。 1093

C 计数和概率

本附录对初等组合学和概率论方面的相关知识进行了回顾。读者如果在这些方面有着很好的知识背景，可以略过本附录的开始部分，着重阅读后面的几节。本书中大多数章节并不需要概率知识，但在有几章中，这方面的知识是必需的。

C.1 节回顾了计数理论的基本结论，包括计算排列和组合的标准公式。C.2 节介绍概率公理和概率分布的一些基本事实。随机变量和期望值和方差的性质在 C.3 节介绍。C.4 节考察从伯努利试验得到的几何分布和二项分布。C.5 节继续考察二项分布，深入讨论该分布的“尾部”特性。

C.1 计数

计数理论尝试回答“多少”的问题，不需要实际枚举出有多少。例如，我们可能会问，“有多少个不同的 n 位数?”或“n 个不同的元素有多少种排序方式?”本节复习计数理论的内容。由于部分内容需要对集合有初步的了解，因此建议读者首先复习一下 B.1 节中的内容。

加法规则与乘法规则

有时，我们希望计数的某个元素项集合可以表示成一组不相交集合的并或笛卡儿积。

加法规则指出，从两个不相交集合之一中选择一个元素时，方法的数目是这两个集合的势之和。就是说，如果 A 和 B 是两个没有公共成员的有限集合，那么由等式(B.3)可得 $|A \cup B| = |A| + |B|$。例如，汽车牌照的每一位是一个字母或一个数字。因为字母有 26 种选择，数字有 10 种选择，因此每一位可能的数目是 $26+10=36$。

乘法规则指出，选择一个有序对时，方法的数目是选择第一个元素的方法数乘以选择第二个元素的方法数。就是说，如果 A 和 B 是两个有限集合，那么 $|A \times B| = |A| \cdot |B|$，这就是等式(B.4)。例如，如果冰淇淋店提供 28 种口味的冰淇淋和 4 种奶油，则由一勺冰淇淋和一种奶油组成的圣代冰淇淋的数目为 $28 \times 4 = 112$。

串

有限集合 S 的串是 S 中元素的一个序列。例如，有 8 个长度为 3 的二进制串：

$$000, 001, 010, 011, 100, 101, 110, 111$$

有时，将长度为 k 的串称为 k 串。串 s 的子串 s' 是 s 中连续元素的一个有序序列。k 子串是长度为 k 的子串。例如，010 是 01101001 的一个 3 子串(从位置 4 开始的 3 子串)，但 111 不是 01101001 的子串。

集合 S 的 k 串可以被看成笛卡儿积 S^k 的一个元素，它是一个 k 元组。因此，有 $|S|^k$ 个长度为 k 的串。例如，二进制 k 串的个数是 2^k。直观上，为了在 n 元集上构造一个 k 串，有 n 种方法选择第一个元素；对每个这样的选择，有 n 种方法选择第二个元素；这样依次进行 k 次。从而 k 串的数目是 $n \cdot n \cdots n = n^k$。

排列

有限集合 S 的排列是 S 中所有元素的有序序列，且每个元素仅出现一次。例如，如果 $S = \{a, b, c\}$，则 S 有 6 种排列：

$$abc, acb, bac, bca, cab, cba$$

一个有 n 个元素的集合有 $n!$ 种排列，因为序列的第一个元素可以有 n 种选择，第二个元素有

$n-1$ 种选择，第三个元素有 $n-2$ 种选择，等等。

集合 S 的 k 排列是 S 中 k 个元素的有序序列，且每个元素仅出现一次。(通常的排列其实是 n 元集合的 n 排列。)集合 $\{a, b, c, d\}$ 的 12 个 2 排列是

$$ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc$$

1095

n 元集合的 k 排列的数目是

$$n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!} \tag{C.1}$$

因为有 n 种方式选择第一个元素，$n-1$ 种方式选择第二个元素，依此类推直到选出 k 个元素，最后一个是从 $n-k+1$ 个元素中选择。

组合

n 元集 S 的 k 组合就是 S 的 k 子集。例如，4 元集 $\{a, b, c, d\}$ 有 6 个 2 组合：

$$ab, ac, ad, bc, bd, cd$$

(这里我们将 2 元集 $\{a, b\}$ 简写为 ab，其他类似。) 我们可以通过在 n 元集中选取 k 个不同的元素来构成 n 元集的一个 k 组合。

n 元集 k 组合的数目可以用其 k 排列的数目来表示。对每个 k 组合，它的元素恰好有 $k!$ 种排列，每个都是 n 元集的一个不同的 k 排列。因此，n 元集 k 组合的数目是其 k 排列的数目除以 $k!$。根据式(C.1)，这个数量是

$$\frac{n!}{k!(n-k)!} \tag{C.2}$$

对 $k=0$，这个公式告诉我们从 n 元集中选择 0 个元素有 1 种方法，因此 $0! = 1$。

二项式系数

我们用 $\binom{n}{k}$(读作"n 选 k")表示 n 元集 k 组合的个数。根据式(C.2)有

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

这个公式对 k 和 $n-k$ 是对称的：

$$\binom{n}{k} = \binom{n}{n-k} \tag{C.3}$$

这些数也称为二项式系数，因为它们是出现在二项展开式中的：

1096

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} \tag{C.4}$$

二项展开式的一个特例是当 $x=y=1$：

$$2^n = \sum_{k=0}^{n} \binom{n}{k}$$

这个公式对应于通过包含 1 的个数对 2^n 个二进制 n 串计数：一共有 $\binom{n}{k}$ 个二进制 n 串恰好包含 k 个 1，因为有 $\binom{n}{k}$ 种方法从 n 个位置中选择 k 个位置来放 1。

二项式界

有时，需要求出二项式系数的界。对 $1 \leqslant k \leqslant n$，有下界

$$\binom{n}{k}=\frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}=\left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right)\geqslant\left(\frac{n}{k}\right)^k$$

利用从斯特林近似式(3.17)获得的不等式 $k!\geqslant(k/e)^k$，可得上界

$$\binom{n}{k}=\frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}\leqslant\frac{n^k}{k!}\leqslant\left(\frac{en}{k}\right)^k \tag{C.5}$$

对所有 $0\leqslant k\leqslant n$，可以通过归纳(见练习 C.1-12)证明界

$$\binom{n}{k}\leqslant\frac{n^n}{k^k(n-k)^{n-k}} \tag{C.6}$$

为方便起见，这里假设 $0^0=1$。对 $k=\lambda n$，这里 $0\leqslant\lambda\leqslant 1$，这个界可以被改写为

1097

$$\binom{n}{\lambda n}\leqslant\frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}}=\left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^n=2^{nH(\lambda)}$$

这里

$$H(\lambda)=-\lambda\lg\lambda-(1-\lambda)\lg(1-\lambda) \tag{C.7}$$

是(二元)熵函数，这里为方便起见，设 $0\lg 0=0$，从而 $H(0)=H(1)=0$。

练习

C.1-1　一个 n 串有多少个 k 子串？(将不同位置上同样的 k 子串视为不同的子串。)一个 n 串共有多少个子串？

C.1-2　一个有 n 个输入，m 个输出的布尔函数是从 $\{\text{TRUE, FALSE}\}^n$ 到 $\{\text{TRUE, FALSE}\}^m$ 的函数。有多少个有 n 个输入，1 个输出的布尔函数？有多少个有 n 个输入，m 个输出的布尔函数？

C.1-3　n 个教授围着圆形会议桌而坐的方式有多少种？如果一种坐法能从另一个坐法旋转得到，则认为这两个坐法相同。

C.1-4　要从集合{1，2，…，100}中选择三个不同的数，并且使它们和为偶数，共有多少种选择方法？

C.1-5　对 $0<k\leqslant n$，证明恒等式

$$\binom{n}{k}=\frac{n}{k}\binom{n-1}{k-1} \tag{C.8}$$

C.1-6　对 $0\leqslant k<n$，证明恒等式

$$\binom{n}{k}=\frac{n}{n-k}\binom{n-1}{k}$$

1098

C.1-7　为了从 n 中选择 k 个对象，可以使其中一个对象与众不同，并考虑这个对象是否被选中。利用这个方法证明

$$\binom{n}{k}=\binom{n-1}{k}+\binom{n-1}{k-1}$$

C.1-8　对 $n=0$，1，…，6，$0\leqslant k\leqslant n$，利用练习 C.1-7 的结果制作一张关于二项式系数 $\binom{n}{k}$ 的表，$\binom{0}{0}$ 在最上方，$\binom{1}{0}$ 和 $\binom{1}{1}$ 在下一行，依次类推。这样一张关于二项式系数的表称为帕斯卡三角。

C.1-9　证明

$$\sum_{i=1}^{n} i = \binom{n+1}{2}$$

C.1-10 证明对任意 $n \geqslant 0$ 且 $0 \leqslant k \leqslant n$，当 $k = \lfloor n/2 \rfloor$ 或 $k = \lceil n/2 \rceil$ 时，$\binom{n}{k}$ 有最大值。

*C.1-11 论证对任意 $n \geqslant 0$，$j \geqslant 0$，$k \geqslant 0$ 且 $j + k \leqslant n$，有

$$\binom{n}{j+k} \leqslant \binom{n}{j}\binom{n-j}{k} \tag{C.9}$$

提供一个代数证明和一个基于从 n 中选择 $j+k$ 个对象方法的论证。给出一个等式不成立的例子。

*C.1-12 对 $k \leqslant n/2$，利用归纳来证明不等式(C.6)，并使用等式(C.3)将它扩展到所有的 $k \leqslant n$。

*C.1-13 使用斯特林近似式证明

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}(1 + O(1/n)) \tag{C.10}$$

*C.1-14 通过对熵函数 $H(\lambda)$ 求导来证明当 $\lambda = 1/2$ 时，它达到最大值。$H(1/2)$ 是多少？ 1099

*C.1-15 证明对任意整数 $n \geqslant 0$，

$$\sum_{k=0}^{n} \binom{n}{k} k = n2^{n-1} \tag{C.11}$$

C.2 概率

概率是设计和分析概率的和随机的算法的基本工具。本节回顾基本的概率理论。

我们用样本空间 S 定义概率，样本空间是一个集合，它的元素称为基本事件。每个基本事件可被看成是一次试验的一个可能的输出。对于抛两个不同硬币的试验，样本空间可以看作{H，T}上所有可能的 2 串集合：

$$S = \{HH, HT, TH, TT\}$$

一个事件是样本空间 S 的子集[⊖]。例如，在抛两个硬币的试验中，得到一个正面一个反面的事件是{HT，TH}。事件 S 称为必然事件，事件 $\varnothing$ 称为零事件。如果 $A \cap B = \varnothing$，则称事件 A 和 B 互斥。有时我们将基本事件 $s \in S$ 当作事件{s}。根据定义，所有的基本事件是互斥的。

概率公理

在样本空间 S 上，概率分布 Pr{}是从 S 中事件到实数的映射，并满足下列概率公理：

1)对所有事件 A，$\Pr\{A\} \geqslant 0$。

2)$\Pr\{S\} = 1$。 1100

3)对两个互斥事件 A 和 B，$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$。更一般地，对任何两两互斥的(有限或可数无限)事件序列 A_1，A_2，…，

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\}$$

称 $\Pr\{A\}$ 为事件 A 的概率。这里，公理 2 是一个归一化要求：没有一个原则规定将 1 作为

⊖ 对一般概率分布，可能会有一些样本空间 S 的子集不被认为是事件。这种情况通常发生在样本空间是不可数无限的时候。重要的要求是当取事件的补，求有限或可数个事件的并或求有限或可数个事件的交时，样本空间的事件集合要闭合。我们所看到的大部分概率分布都是关于有限或可数样本空间，通常将样本空间的所有子集看作事件。一个需要注意的例外是连续均匀概率分布，我们稍后将讨论它。

必然事件的概率，仅仅是为了自然和方便起见。

从这些公理和基本的集合论(见 B.1 节)可以得到一些结果。零事件$\varnothing$的概率 $\Pr\{\varnothing\}=0$。如果 $A\subseteq B$，则 $\Pr\{A\}\leqslant\Pr\{B\}$。使用 $\bar{A}$ 表示事件 $S-A$(A 的补)，则有 $\Pr\{\bar{A}\}=1-\Pr\{A\}$。对任意两个事件 A 和 B，

$$\Pr\{A\cup B\}=\Pr\{A\}+\Pr\{B\}-\Pr\{A\cap B\} \tag{C.12}$$

$$\leqslant\Pr\{A\}+\Pr\{B\} \tag{C.13}$$

在抛硬币的试验中，设 4 个基本事件中每个的概率为 1/4，那么至少得到一个正面的概率是

$$\Pr\{HH,HT,TH\}=\Pr\{HH\}+\Pr\{HT\}+\Pr\{TH\}=3/4$$

另一种做法是，因为得到少于一个正面的概率 $\Pr\{TT\}=1/4$，则得到至少一个正面的概率是 $1-1/4=3/4$。

离散概率分布

如果概率分布定义在有限或可数无限的样本空间，则称其为离散概率分布。令 S 为样本空间。对任何事件 A，因为 A 中的基本事件是互斥的，所以

$$\Pr\{A\}=\sum_{s\in A}\Pr\{s\}$$

如果 S 是有限的，且每个事件 $s\in S$ 的概率是

$$\Pr\{s\}=1/|S|$$

则得到 S 上的均匀概率分布。在这种情况下，实验常被表述成“从 S 中随机选择一个元素”。

作为一个例子，考虑抛均质硬币的过程，抛这种硬币时得到正面和反面的概率是相同的，都
1101 是 1/2。如果抛 n 次，则得到样本空间 $S=\{H,T\}^n$ 上的均匀概率分布，S 是大小为 2^n 的集合。S 中的每个基本事件都可以表示成集合$\{H,T\}$上长度为 n 的串，并且每个事件的发生概率都是 $1/2^n$。事件

$$A=\{\text{恰好出现 } k \text{ 次正面和 } n-k \text{ 次反面}\}$$

是 S 的一个子集，大小 $|A|=\binom{n}{k}$。因为集合$\{H,T\}$上长度为 n 的串中恰好有$\binom{n}{k}$个包含 k 个 H 的串，因此事件 A 的概率 $\Pr\{A\}=\binom{n}{k}/2^n$。

连续均匀概率分布

连续均匀概率分布是一个不是所有的样本空间子集都可看成事件的概率分布的例子。连续均匀概率分布定义在一个实数闭区间$[a,b]$上，这里 $a<b$。直观上，我们希望区间$[a,b]$中的每个点有“相等的可能”。因为有不可数个点，因此如果给每个点都给予相同的有限、正概率，则不能同时满足公理 2 和公理 3。因为这个原因，我们仅将概率与 S 的某些子集相关，从而使这些事件满足公理。

对任何闭区间$[c,d]$，这里 $a\leqslant c\leqslant d\leqslant b$，定义事件$[c,d]$概率的连续均匀概率分布是

$$\Pr\{[c,d]\}=\frac{d-c}{b-a}$$

注意对任何点 $x=[x,x]$，x 的概率是 0。如果去掉某一区间$[c,d]$的两个端点，就可以得到开区间(c,d)。因为$[c,d]=[c,c]\cup(c,d)\cup[d,d]$，根据公理 3 有 $\Pr\{[c,d]\}=\Pr\{(c,d)\}$。通常，对连续均匀概率分布来说，事件集合是样本空间$[a,b]$的任意子集，它可以通过有限或可数个开和闭区间的并而获得。

条件概率和独立性

有时，对实验结果会有一些先验知识。例如，一个朋友抛两个均质硬币，并告诉你至少一个硬币是正面。那么两个硬币都是正面的概率是多少？给出的信息排除了两个都是反面的可能。剩下的三个基本事件都有相等的可能，因此我们推断每个事件发生的概率是 1/3。因为仅有一个基本事件是两个正面，因此答案是 1/3。

1102

条件概率给出了对试验结果有先验知识的形式化定义。当另一事件 B 发生时，事件 A 的条件概率是

$$\Pr\{A \mid B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}} \tag{C.14}$$

要求 $\Pr\{B\} \neq 0$。（我们将“$\Pr\{A \mid B\}$”读作“在 B 条件下，A 的概率”。）直观上，我们认为 B 发生、A 也发生的事件是 $A \cap B$。即 $A \cap B$ 是 A 和 B 都发生的结果集。因为结果是 B 中的一个基本事件，通过将 B 中所有基本事件的概率除以 $\Pr\{B\}$ 来对它们进行正规化，它们的和是 1。因此，在给定 B 的情况下，A 的条件概率是事件 $A \cap B$ 的概率与事件 B 的概率的比率。在上面例子中，A 是两个正面的事件，B 是至少一个正面的事件。因此 $\Pr\{A \mid B\} = (1/4)/(3/4) = 1/3$。

如果两个事件 A 和 B 满足

$$\Pr\{A \cap B\} = \Pr\{A\}\Pr\{B\} \tag{C.15}$$

则称两个事件独立，如果 $\Pr\{B\} \neq 0$，这等同于

$$\Pr\{A \mid B\} = \Pr\{A\}$$

例如，假设抛两个均质硬币的结果是独立的。那么两个都是正面的概率是 $(1/2)(1/2) = 1/4$。现在假设一个事件是第一个硬币正面向上，另一个事件是两个硬币向上的面不同，每个事件发生的概率都是 1/2，它们同时发生的概率是 1/4，根据独立的定义，这两个事件是独立的，尽管有人认为两个事件都依赖于第一个硬币。最后，假设硬币焊接在一起，因此它们同时正面向上或反面向上，并且两者的概率都相同。从而每个硬币正面向上的概率都是 1/2，但它们同时正面向上的概率是 $1/2 \neq (1/2)(1/2)$。因此，一个硬币正面向上的事件与另一个硬币正面向上的事件不独立。

如果对所有 $1 \leqslant i < j \leqslant n$ 有

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\}$$

则称事件集合 A_1，A_2，…，A_n 为两两独立。对事件集合的每个 k 子集 A_{i_1}，A_{i_2}，…，A_{i_k}，这里 $2 \leqslant k \leqslant n$ 且 $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$，如果满足

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}\} = \Pr\{A_{i_1}\}\Pr\{A_{i_2}\}\cdots\Pr\{A_{i_k}\} \tag{1103}$$

则称这些事件是(相互)独立的。例如，假设抛两个均质硬币。令 A_1 为第一个硬币正面向上的事件，A_2 为第二个硬币正面向上的事件，A_3 为两个硬币面向不同的事件。有

$$\begin{aligned}
\Pr\{A_1\} &= 1/2 \\
\Pr\{A_2\} &= 1/2 \\
\Pr\{A_3\} &= 1/2 \\
\Pr\{A_1 \cap A_2\} &= 1/4 \\
\Pr\{A_1 \cap A_3\} &= 1/4 \\
\Pr\{A_2 \cap A_3\} &= 1/4 \\
\Pr\{A_1 \cap A_2 \cap A_3\} &= 0
\end{aligned}$$

因为对 $1 \leqslant i < j \leqslant 3$，有 $\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} = 1/4$，因而事件 A_1，A_2 和 A_3 是两两独

立的，但不是相互独立的，因为 $\Pr\{A_1 \cap A_2 \cap A_3\}=0$，但 $\Pr\{A_1\}\Pr\{A_2\}\Pr\{A_3\}=1/8\neq 0$。

贝叶斯定理

根据条件概率(C.14)的定义和交换律 $A\cap B=B\cap A$，两个具有非零概率的事件 A 和 B 满足

$$\Pr\{A \cap B\}=\Pr\{B\}\Pr\{A \mid B\}=\Pr\{A\}\Pr\{B \mid A\} \tag{C.16}$$

求解 $\Pr\{A \mid B\}$，可得

$$\Pr\{A \mid B\}=\frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{B\}} \tag{C.17}$$

这称为贝叶斯定理。分母 $\Pr\{B\}$ 是一个规范化常量，可以将其重新表示如下。因为 $B=(B\cap A)\cup(B\cap\bar{A})$，且 $B\cap A$ 和 $B\cap\bar{A}$ 是互斥事件，故有：

$$\Pr\{B\}=\Pr\{B \cap A\}+\Pr\{B \cap \bar{A}\}=\Pr\{A\}\Pr\{B \mid A\}+\Pr\{\bar{A}\}\Pr\{B \mid \bar{A}\}$$

代入式(C.17)，得贝叶斯定理的等价形式：

1104

$$\Pr\{A \mid B\}=\frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\}+\Pr\{\bar{A}\}\Pr\{B \mid \bar{A}\}}$$

贝叶斯定理可以简化条件概率的计算。例如，假设有一个均质硬币和一个总是正面向上的不均匀硬币。我们进行一次包含三个独立事件的实验：随机选择一个硬币，将这个硬币抛一次，将这个硬币再抛一次。如果被选中的硬币两次都是正面向上，那么它是不均匀硬币的概率是多少？

我们使用贝叶斯定理来解决这个问题。令 A 表示事件不均匀硬币被选中，B 表示事件硬币两次都是正面向上。我们需要计算 $\Pr\{A \mid B\}$. 因为有 $\Pr\{A\}=1/2$，$\Pr\{B \mid A\}=1$，$\Pr\{\bar{A}\}=1/2$，$\Pr\{B \mid \bar{A}\}=1/4$，从而

$$\Pr\{A \mid B\}=\frac{(1/2)\cdot 1}{(1/2)\cdot 1+(1/2)\cdot(1/4)}=4/5$$

练习

C.2-1　证明布尔不等式：对任意有限或可数无限事件序列 A_1，A_2，…，

$$\Pr\{A_1 \cup A_2 \cup \cdots\}\leqslant \Pr\{A_1\}+\Pr\{A_2\}+\cdots \tag{C.18}$$

C.2-2　Rosencrantz 教授抛一个均质硬币一次。Guildenstern 教授抛一个均质硬币两次。Rosencrantz 教授比 Guildenstern 教授得到更多正面的概率是多少？

C.2-3　一副 10 张的纸牌，每张带有一个 1 到 10 不同的数字，并通过洗牌将它们彻底打乱。每次从这副牌中抽一张，一共抽 3 张。这三张牌按升序被选中的概率是多少？

***C.2-4**　描述一个以两个整数 a 和 b $(0<a<b)$ 为输入的过程，抛均质硬币使输出为正面向上的概率是 a/b，反面向上的概率是 $(b-a)/b$。给出一个需要抛硬币次数的界限，应该是 $O(1)$ 复杂度的。(提示：将 a/b 以二进制形式表示。)

1105 **C.2-5**　证明 $\Pr\{A \mid B\}+\Pr\{\bar{A}/B\}=1$。

C.2-6　对任意事件集合 A_1，A_2，…，A_n，证明

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_n\}=\Pr\{A_1\}\cdot\Pr\{A_2 \mid A_1\}\cdot\Pr\{A_3 \mid A_1 \cap A_2\}\cdots \Pr\{A_n \mid A_1 \cap A_2 \cap \cdots \cap A_{n-1}\}$$

***C.2-7**　描述怎样构造一个 n 个事件的集合，使得它们两两独立，但不存在包含 $k>2$ 个相互独立元素的子集。

***C.2-8**　已知事件 C，如果

$$\Pr\{A \cap B \mid C\}=\Pr\{A \mid C\}\cdot\Pr\{B \mid C\}$$

则事件 A 和 B 是条件独立的。给出一个关于两个事件的简单但非平凡的例子，使它们不独立，但在给定第三个事件的情况下是条件独立的。

***C.2-9** 如果你是一个游戏的参加者，这个游戏的奖品放在三个幕布的其中一个后面。如果你选择了正确的幕布，你就会赢得奖品。你选择了一个幕布，但在你掀起幕布前，主持人掀起其他两个幕布之一，使你知道那个幕布里面是空的，并询问你是否愿意改选剩下的那个幕布。如果你改选，你的机会将会如何变化？

***C.2-10** 一个监狱看守从三个罪犯中随机选择一个予以释放，其他两个将被处死。警卫知道哪个人是否会被释放，但是不允许给罪犯任何关于其状态的信息。让我们分别称罪犯为 X，Y，Z。罪犯 X 私下问警卫 Y 或 Z 哪个会被处死，因为他已经知道他们中至少一个人会死，警卫不能透露任何关于他本人状态的信息。警卫告诉 X，Y 将被处死。X 感到很高兴，因为他认为他或者 Z 将被释放，这意味着他被释放的概率是 1/2。他正确吗？或者他的机会仍是 1/3？请解释。

C.3 离散随机变量

（离散）随机变量 X 是一个从有限或可数无限样本空间 S 到实数的函数。它将一个实数与一次试验的每个可能结果关联起来，这使得我们可以分析结果实数集上的概率分布。随机变量也 1106
可以被定义在不可数无限样本空间上，但这会引起一些技术问题，而我们没必要去解决它。因此，我们假设随机变量是离散的。

对一个随机变量 X 和一个实数 x，将事件 $X=x$ 定义为 $\{s\in S : X(s)=x\}$，从而

$$\Pr\{X=x\}=\sum_{s\in S:X(s)=x}\Pr\{s\}$$

函数

$$f(x)=\Pr\{X=x\}$$

是随机变量 X 的概率密度函数。根据概率公理，$\Pr\{X=x\}\geqslant 0$ 且 $\sum_x \Pr\{X=x\}=1$。

作为一个例子，考虑抛一对普通的 6 面骰子的实验。样本空间有 36 个可能的基本事件。假设满足均匀概率分布，从而每个基本事件 $s\in S$ 有相同的可能：$\Pr\{s\}=1/36$。定义随机变量 X 为两个骰子值的大者。$\Pr\{X=3\}=5/36$，因为 X 将值 3 赋给 36 个可能基本事件中的 5 个，它们是(1，3)，(2，3)，(3，3)，(3，2)和(3，1)。

在同一个样本空间中定义多个随机变量是很平常的。如果 X 和 Y 是随机变量，函数

$$f(x,y)=\Pr\{X=x \text{ 且 } Y=y\}$$

是 X 和 Y 的联合概率密度函数。对一个固定值 y，

$$\Pr\{Y=y\}=\sum_x \Pr\{X=x \text{ 且 } Y=y\}$$

类似地，对一个固定值 x，

$$\Pr\{X=x\}=\sum_y \Pr\{X=x \text{ 且 } Y=y\}$$

利用条件概率(C.14)的定义，可得

$$\Pr\{X=x \mid Y=y\}=\frac{\Pr\{X=x \text{ 且 } Y=y\}}{\Pr\{Y=y\}}$$

如果对所有 x 和 y，事件 $X=x$ 和 $Y=y$ 是独立的，则称随机变量 X 和 Y 是独立的。一种等价的表述是：如果对所有 x 和 y，有 $\Pr\{X=x \text{ 和 } Y=y\}=\Pr\{X=x\}\Pr\{Y=y\}$，则 X 和 Y 是独立的。

给定同一个样本空间上定义的随机变量集，可以定义作为原变量和、积或其他函数的新随

1107 机变量。

随机变量的期望值

对一个随机变量来说，关于其分布的最简单、最有效的概括是它具有取值的“平均”。一个离散随机变量 X 的**期望值**(期望或中数)是

$$\mathrm{E}[X] = \sum_x x\Pr\{X = x\} \tag{C.19}$$

当上式中的和是有限或绝对收敛时，这个公式是良定义的。有时，将 X 的期望表示成 μ_X，或者，当根据上下文很容易看出随机变量时，简写成 μ。

考虑一个抛两个均质硬币的游戏。有一个正面向上你赢 3 美元，有一个反面向上你输 2 美元。表示收入的随机变量 X 的期望值是

$$\begin{aligned}\mathrm{E}[X] &= 6 \cdot \Pr\{2\mathrm{H'S}\} + 1 \cdot \Pr\{1\mathrm{H},1\mathrm{T}\} - 4 \cdot \Pr\{2\mathrm{T'S}\} \\ &= 6(1/4) + 1(1/2) - 4(1/4) = 1\end{aligned}$$

两个随机变量和的期望是它们期望的和，即

$$\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y] \tag{C.20}$$

在 $\mathrm{E}[X]$和 $\mathrm{E}[Y]$有定义处都成立。将这个特性称为期望的线性性质，即使 X 和 Y 不独立，这个公式也成立。它可以被扩展到期望的有限的、绝对收敛的和。期望的线性性质是使我们能利用指标随机变量(见 5.2 节)来进行概率分析的关键属性。

如果 X 是任意随机变量，任意函数 $g(x)$定义一个新的随机变量 $g(X)$。如果 $g(X)$的期望有定义，则

$$\mathrm{E}[g(X)] = \sum_x g(x)\Pr\{X = x\}$$

令 $g(x) = ax$，对任何常量 a，有

$$\mathrm{E}[aX] = a\mathrm{E}[X] \tag{C.21}$$

因此，期望是线性的：对任意随机变量 X 和 Y 以及常量 a，有

$$\mathrm{E}[aX + Y] = a\mathrm{E}[X] + \mathrm{E}[Y] \tag{C.22}$$

当两个随机变量 X 和 Y 独立且各自都有一个已定义的期望时，有：

$$\begin{aligned}\mathrm{E}[XY] &= \sum_x \sum_y xy\Pr\{X = x \text{ 和 } Y = y\} = \sum_x \sum_y xy\Pr\{X = x\}\Pr\{Y = y\} \\ &= \Big(\sum_x x\Pr\{X = x\}\Big)\Big(\sum_y y\Pr\{Y = y\}\Big) = \mathrm{E}[X]\mathrm{E}[Y]\end{aligned}$$

通常，当 n 个随机变量 X_1，X_2，…，X_n 相互独立时，有：

$$\mathrm{E}[X_1 X_2 \cdots X_n] = \mathrm{E}[X_1]\mathrm{E}[X_2]\cdots\mathrm{E}[X_n] \tag{C.23}$$

当一个随机变量 X 从自然数集 $\mathbf{N} = \{0, 1, 2, \cdots\}$中取值时，可以用一个很好的公式来表示其期望：

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=0}^{\infty} i\Pr\{X = i\} = \sum_{i=0}^{\infty} i\,(\Pr\{X \geqslant i\} - \Pr\{X \geqslant i+1\}) \\ &= \sum_{i=1}^{\infty} \Pr\{X \geqslant i\}\end{aligned} \tag{C.24}$$

因为每个项 $\Pr\{X \geqslant i\}$被加入 i 次，减去 $i-1$ 次(除 $\Pr\{X \geqslant 0\}$，它被加入 0 次，且没有被减去)。

当对随机变量 X 应用凸函数 $f(x)$时，假设期望存在且有限，则有 Jensen 不等式(Jensen's inequality)：

$$\mathrm{E}[f(X)] \geqslant f(\mathrm{E}[X]) \tag{C.25}$$

(如果对任意 x、y 和所有的 $0 \leqslant \lambda \leqslant 1$，有 $f(\lambda x+(1-\lambda)y) \leqslant \lambda f(x)+(1-\lambda)f(y)$，则称函数 $f(x)$ 为凸函数。)

方差和标准差

随机变量的期望值并没有说明随机变量的值是如何"分布的"。例如，如果对随机变量 X 和 Y，有 $\Pr\{X=1/4\}=\Pr\{X=3/4\}=1/2$ 和 $\Pr\{Y=0\}=\Pr\{Y=1\}=1/2$，那么 $\mathrm{E}[X]$和 $\mathrm{E}[Y]$都是 1/2，但 Y 的实际取值比 X 的实际取值离均值要更远一些。

1108
~
1109

方差的概念在数学上表示随机变量的取值距离均值可能会有多远。均值为 $\mathrm{E}[X]$的随机变量 X 的*方差*是

$$\begin{aligned}\mathrm{Var}[X] &= \mathrm{E}\left[(X-\mathrm{E}[X])^2\right] \\ &= \mathrm{E}[X^2-2X\mathrm{E}[X]+\mathrm{E}^2[X]] \\ &= \mathrm{E}[X^2]-2\mathrm{E}[X\mathrm{E}[X]]+\mathrm{E}^2[X] \\ &= \mathrm{E}[X^2]-2\mathrm{E}^2[X]+\mathrm{E}^2[X] \\ &= \mathrm{E}[X^2]-\mathrm{E}^2[X] \end{aligned} \tag{C.26}$$

等式 $\mathrm{E}[\mathrm{E}^2[X]]=\mathrm{E}^2[X]$和 $\mathrm{E}[X\mathrm{E}[X]]=\mathrm{E}^2[X]$成立的理由是 $\mathrm{E}[X]$不是随机变量，而仅仅是一个实数，这意味着等式(C.21)可以被应用($a=\mathrm{E}[X]$)。式(C.26)可以被改写，以获得随机变量平方的期望公式：

$$\mathrm{E}[X^2] = \mathrm{Var}[X]+\mathrm{E}^2[X] \tag{C.27}$$

随机变量 X 的方差和 aX 的方差是相关的(见练习 C.3-10)：

$$\mathrm{Var}[aX] = a^2\mathrm{Var}[X]$$

当 X 和 Y 是独立随机变量时，

$$\mathrm{Var}[X+Y] = \mathrm{Var}[X]+\mathrm{Var}[Y]$$

通常，如果 n 个随机变量 X_1，X_2，…，X_n 是两两独立的，那么

$$\mathrm{Var}\Big[\sum_{i=1}^{n} X_i\Big] = \sum_{i=1}^{n}\mathrm{Var}[X_i] \tag{C.28}$$

随机变量 X 的*标准差*是 X 方差的正平方根。随机变量 X 的标准差有时可以表示成 σ_X，或者，当随机变量 X 可根据上下文理解时，可以简写成 σ。根据这种表示，X 的方差可以表示成 σ^2。

练习

C.3-1 转动两个普通的 6 面骰子。两个骰子值的和的期望是多少？两个值中大值的期望是多少？

C.3-2 有一个包含 n 个不同值的数组 $A[1..n]$，其中的值是随机排序的，且这 n 个数的每种排列都是等可能的。数组中最大值下标的期望是多少？数组中最小值下标的期望是什么？ 1110

C.3-3 狂欢节游戏中使用一个带有 3 个骰子的罩子。参加者可以在 1 到 6 的任意一个数字上押 1 美元。主持人摇动罩子，并按如下方法决定参加者获得的回报。如果任何一个骰子都未出现参加者所押的数字，则他输掉所押的钱。反之，如果他押的数字出现在三个骰子中的第 k 个上，$k=1$，2，3，则他收回所押的钱并额外获得 k 美元。玩一次游戏的收入期望是多少？

C.3-4 证明如果随机变量 X 和 Y 非负，则

$$\mathrm{E}[\max(X,Y)] \leqslant \mathrm{E}[X]+\mathrm{E}[Y]$$

***C.3-5** 令 X 和 Y 是独立随机变量。证明对任何函数 f 和 g，$f(X)$和 $f(Y)$独立。

***C.3-6** 令 X 为非负随机变量，并假设 $\mathrm{E}[X]$有良定义。对任意 $t>0$，证明马尔可夫不等式

(Markov's inequality)：

$$\Pr\{X \geqslant t\} \leqslant \mathrm{E}[X]/t \tag{C.29}$$

•C.3-7　令 S 是样本空间，X 和 X' 是随机变量且满足对所有 $s \in S$ 有 $X(s) \geqslant X'(s)$。证明对任意实常数 t，有

$$\Pr\{X \geqslant t\} \geqslant \Pr\{X' \geqslant t\}$$

C.3-8　哪个更大：随机变量平方的期望，还是它期望的平方？

C.3-9　证明对任何取值为 0、1 的随机变量 X，有

$$\mathrm{Var}[X] = \mathrm{E}[X]\mathrm{E}[1 - X]$$

1111 C.3-10　根据式(C.26)对方差的定义，证明 $\mathrm{Var}[aX] = a^2 \mathrm{Var}[X]$。

C.4　几何分布与二项分布

抛硬币是伯努利试验的例子，它定义只有两种可能结果的试验：成功，概率是 p，失败，概率是 $q = 1 - p$。如果讨论多个伯努利试验的集合，除非特别指出，一般认为各试验是相互独立的，每个试验的成功概率都是 p。两种从伯努利试验引出的重要分布是：几何分布和二项分布。

几何分布

假设进行一系列伯努利试验，每次实验成功的概率是 p，失败的概率是 $q = 1 - p$。在取得一次成功前一共要进行多少次试验？令随机变量 X 为取得一次成功所要进行的试验次数，则 X 的取值范围是 $\{1, 2, \cdots\}$。对 $k \geqslant 1$，因为在取得一次成功前有 $k-1$ 次失败，从而有

$$\Pr\{X = k\} = q^{k-1}p \tag{C.30}$$

满足式(C.30)的概率分布称为几何分布。图 C.1 显示了这种分布。

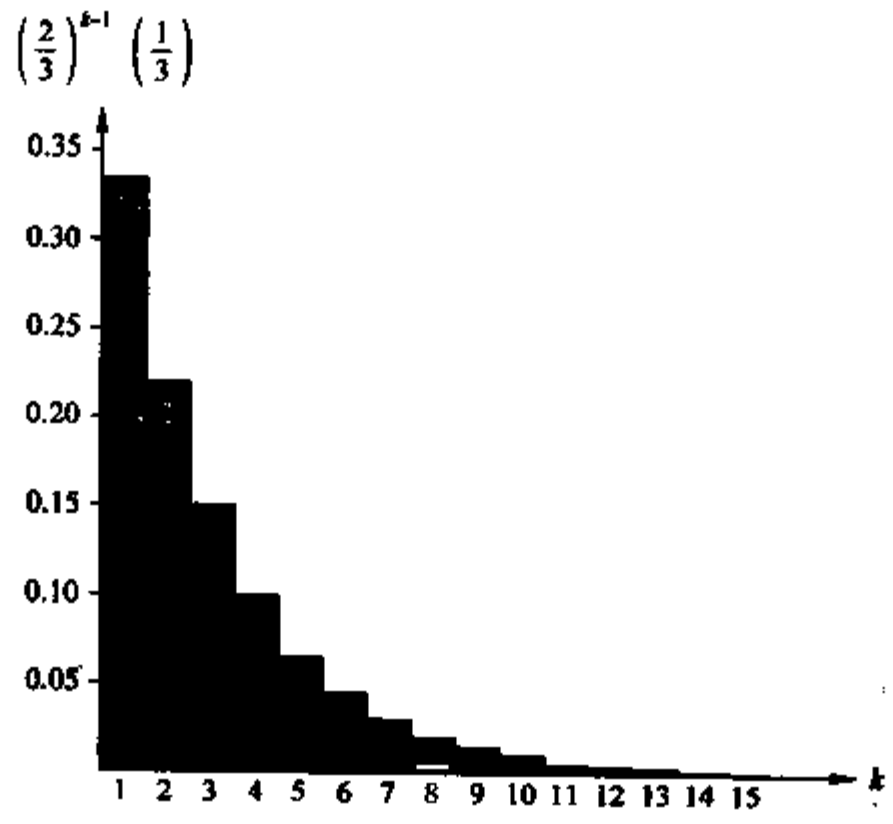


图 C.1　一个成功概率是 $p = 1/3$，失败概率是 $q = 1 - p$ 的几何分布。它的期望是 $1/p = 3$

假设 $q < 1$，几何分布的期望可以通过等式(A.8)计算：

$$\mathrm{E}[X] = \sum_{k=1}^{\infty} kq^{k-1}p = \frac{p}{q}\sum_{k=0}^{\infty} kq^k = \frac{p}{q} \cdot \frac{q}{(1-q)^2} = 1/p \tag{C.31}$$

因此，根据直觉，在取得一次成功前，平均需要进行 $1/p$ 次试验。方差可以类似计算，但使用练习 A.1-3 的公式，方差是

$$\mathrm{Var}[X] = q/p^2 \tag{C.32}$$

作为一个例子，考虑重复抛两个骰子，直到获得 1 个 7 或 1 个 11。在 36 个可能的结果中，6 个产生 7，2 个产生 11。因此，成功的概率是 $p=8/36=2/9$。因此，在取得一个 7 或 11 前，平均要抛 $1/p=9/2=4.5$ 次骰子。

二项分布

当成功的概率是 p，失败的概率是 $q=1-p$ 时，n 次伯努利试验会成功多少次？定义随机变量 X 为 n 次试验中成功的次数，则 X 的取值范围是 $\{1, 2, \cdots, n\}$。对 $k=0, \cdots, n$，因为有 $\binom{n}{k}$ 种方法从 n 次试验中选取 k 次成功的试验，且每次发生的概率是 $p^k q^{n-k}$，从而有

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k} \tag{C.33}$$

满足等式(C.33)的概率分布称为二项分布。方便起见，将二项分布族表示成

$$b(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k} \tag{C.34}$$

图 C.2 显示了一个二项分布。“二项”这个名字由式(C.33)是 $(p+q)^n$ 展开式的第 k 项而来。因为 $p+q=1$，根据概率公理 2 的要求有

$$\sum_{k=0}^{n} b(k; n, p) = 1 \tag{C.35}$$

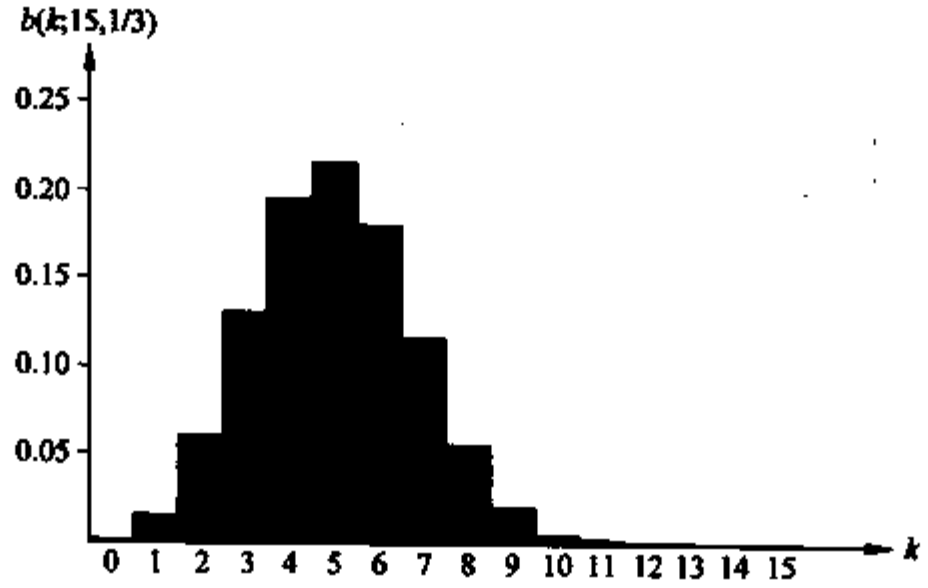


图 C.2 $n=15$ 次伯努利试验，每个都有 $p=1/3$ 的成功概率，所产生的二项分布 $b(k; 15, 1/3)$。分布的期望值为 $np=5$

根据式(C.8)和式(C.35)，可以计算出二项分布随机变量的期望。令 X 为满足二项分布 $b(k; n, p)$ 的随机变量，$q=1-p$。根据期望的定义，有

$$\begin{aligned} \mathrm{E}[X] &= \sum_{k=0}^{n} k \Pr\{X = k\} = \sum_{k=0}^{n} k b(k; n, p) = \sum_{k=1}^{n} k \binom{n}{k} p^k q^{n-k} \\ &= np \sum_{k=1}^{n} \binom{n-1}{k-1} p^{k-1} q^{n-k} = np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} \end{aligned}$$

$$= np\sum_{k=0}^{n-1} b(k;n-1,p) = np \tag{C.36}$$

利用期望的线性特征，可以通过相当少的代数运算获得相同的结果。令随机变量 X_i 表示第 i 次试验中成功的次数。那么 $\mathrm{E}[X_i]=p\cdot 1+q\cdot 0=p$，再根据期望的线性特征(式(C.20))，可知 n 次试验中成功次数的期望是

$$\mathrm{E}[X]=\mathrm{E}\left[\sum_{i=1}^{n} X_i\right]=\sum_{i=1}^{n}\mathrm{E}[X_i]=\sum_{i=1}^{n} p=np \tag{C.37}$$

可以使用同样的方法来计算分布的方差。根据式(C.26)有 $\mathrm{Var}[X_i]=\mathrm{E}[X_i^2]-\mathrm{E}^2[X_i]$。因此 X_i 仅能取 0 和 1，则有 $\mathrm{E}[X_i^2]=\mathrm{E}[X_i]=p$，从而

$$\mathrm{Var}[X_i]=p-p^2=pq \tag{C.38}$$

我们利用 n 次试验的独立性来计算 X 的方差，根据式(C.28)有

$$\mathrm{Var}[X]=\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right]=\sum_{i=1}^{n}\mathrm{Var}[X_i]=\sum_{i=1}^{n} pq=npq \tag{C.39}$$

从图 C.2 可以看出，随着 k 从 0 到 n 增大，二项分布 $b(k;n,p)$ 先增加直到达到均值 np，
1115 然后再减小。可以通过计算后续项的比率，来证明二项分布总是表现成这样：

$$\frac{b(k;n,p)}{b(k-1;n,p)}=\frac{\binom{n}{k}p^k q^{n-k}}{\binom{n}{k-1}p^{k-1}q^{n-k+1}}=\frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} \tag{C.40}$$

$$=\frac{(n-k+1)p}{kq}=1+\frac{(n+1)p-k}{kq}$$

当 $(n+1)p-k$ 为正时，比率大于 1。从而对 $k<(n+1)p$ 有 $b(k;n,p)>b(k-1;n,p)$(上升过程)，对 $k>(n+1)p$ 有 $b(k;n,p)<b(k-1;n,p)$(下降过程)。如果 $k=(n+1)p$ 是整数，则 $b(k;n,p)=b(k-1;n,p)$，因此分布有两个最大值：在 $k=(n+1)p$ 和 $k-1=(n+1)p-1=np-q$ 两点。反之，它在 $np-q<k<(n+1)p$ 间的唯一整数 k 处取得最大值。

下列引理给出二项分布的一个上限。

引理 C.1　令 $n\geqslant 0$，$0<p<1$，$q=1-p$，$0\leqslant k\leqslant n$。有

$$b(k;n,p)\leqslant\left(\frac{np}{k}\right)^k\left(\frac{nq}{n-k}\right)^{n-k}$$

证明：利用式(C.6)，有

$$b(k;n,p)=\binom{n}{k}p^k q^{n-k}\leqslant\left(\frac{n}{k}\right)^k\left(\frac{n}{n-k}\right)^{n-k}p^k q^{n-k}=\left(\frac{np}{k}\right)^k\left(\frac{nq}{n-k}\right)^{n-k}$$ ■

练习

C.4-1　对几何分布，验证概率公理的公理 2。

C.4-2　抛 6 个均质硬币，在获得 3 个正面向上和 3 个反面向上前平均要抛多少次？

1116 C.4-3　证明 $b(k;n,p)=b(n-k;n,q)$，这里 $q=1-p$。

C.4-4　证明对应二项分布 $b(k;n,p)$ 最大值的值约等于 $1/\sqrt{2\pi npq}$，这里 $q=1-p$。

*C.4-5　证明 n 次伯努利试验中没有一次成功的概率约等于 $1/e$，每次试验成功的概率是 $p=1/n$。证明只有一次成功的概率也约等于 $1/e$。

*C.4-6　Rosencrantz 教授和 Guildenstern 教授都抛一个均质硬币 n 次。证明他们获得的正面向上次数相同的概率是 $\binom{2n}{n}/4^n$。(提示：对 Rosencrantz 教授来说，正面向上表示成功；对

Guildenstern 教授来说，反面向上表示成功。）利用你的结论验证等式

$$\sum_{i=0}^{n}\binom{n}{k}^2=\binom{2n}{n}$$

*C.4-7 证明对 $0\leqslant k\leqslant n$，

$$b(k;n,1/2)\leqslant 2^{nH(k/n)-n}$$

这里 $H(x)$是熵函数(C.7)。

*C.4-8 考虑 n 次伯努利试验，这里对 $i=1,2,\cdots,n$，第 i 次试验成功的概率是 p_i，令 X 为表示总成功次数的随机变量。令对所有 $i=1,2,\cdots,n$ 有 $p\geqslant p_i$。证明对 $1\leqslant k\leqslant n$，有

$$\Pr\{X<k\}\leqslant\sum_{i=0}^{k-1}b(i;n,p)$$

*C.4-9 令 X 为表示集合 A 中 n 次伯努利试验总成功次数的随机变量，其中第 i 次试验成功的概率是 p_i，令 X'为表示集合 A'中 n 次伯努利试验总成功次数的随机变量，其中第 i 次试验成功的概率 $p_i'\geqslant p_i$。证明对 $0\leqslant k\leqslant n$ 有

$$\Pr\{X'\geqslant k\}\geqslant\Pr\{X\geqslant k\}$$

（提示：说明如何通过包含 A 中试验的实验来获得 A'中的伯努利试验，并利用练习 C.3-7 的结论）

1117

*C.5 二项分布的尾

对每次成功概率都为 p 的 n 次伯努利试验，至少或至多有 k 次成功的概率比正好有 k 次成功的概率更有意思。本节要讨论二项分布的尾，即二项分布 $b(k;n,p)$的两个远离均值 np 的区域。我们要证明尾部(所有项的和)的几个重要的界。

下面首先给出有关分布 $b(k;n,p)$的右尾的界，关于左尾的界可以通过成功和失败来确定。

定理 C.2 考虑 n 个伯努利试验的序列，这里成功概率为 p。令 X 为表示成功总数的随机变量。则对 $0\leqslant k\leqslant n$，至少成功 k 次的概率是

$$\Pr\{X\geqslant k\}=\sum_{i=k}^{n}b(i;n,p)\leqslant\binom{n}{k}p^k$$

证明：对 $S\subseteq\{1,2,\cdots,n\}$，令 A_S 表示对每个 $i\in S$ 有第 i 次试验成功的事件。很明显如果 $|S|=k$，则 $\Pr\{A_S\}=p^k$。我们有

$$\begin{aligned}\Pr\{X\geqslant k\}&=\Pr\{\text{存在 } S\subseteq\{1,2,\cdots,n\} \text{ 使 } |S|=k \text{ 和 } A_S\}\\&=\Pr\Big\{\bigcup_{S\subseteq\{1,2,\cdots,n\},|S|=k}A_S\Big\}\leqslant\sum_{S\subseteq\{1,2,\cdots,n\},|S|=k}\Pr\{A_S\}=\binom{n}{k}p^k\end{aligned}$$

这里的不等式是由 Boole 不等式(C.18)而来。 ■

下面的推论重申了关于二项分布左尾部的定理。通常，我们将对左尾部的证明应用到右尾部的工作留给读者。

推论 C.3 考虑 n 个伯努利试验的序列，成功的概率是 p。如果 X 是表示成功总数的随机变量，那么对 $0\leqslant k\leqslant n$，最多有 k 次成功的概率是

1118

$$\Pr\{X\leqslant k\}=\sum_{i=0}^{k}b(i;n,p)\leqslant\binom{n}{n-k}(1-p)^{n-k}=\binom{n}{k}(1-p)^{n-k}$$ ■

下面来考虑二项分布左尾部的界。它的推论表明左尾部在远离均值时按指数下降。

定理 C.4 考虑 n 个伯努利试验的序列，这里成功的概率是 p，失败的概率是 $q=1-p$。令

X 为表示成功总数的随机变量。从而对 $0<k<np$，少于 k 次成功的概率是

$$\Pr\{X<k\}=\sum_{i=0}^{k-1}b(i;n,p)<\frac{kq}{np-k}b(k;n,p)$$

证明：利用 A.2 节的方法，通过几何级数为级数 $\sum_{i=0}^{k-1}b(i;n,p)$ 定界。对 $i=1,2,\cdots,k$，从式(C.40)得

$$\frac{b(i-1;n,p)}{b(i;n,p)}=\frac{iq}{(n-i+1)p}<\frac{iq}{(n-i)p}\leqslant\frac{kq}{(n-k)p}$$

如果令

1119

$$x=\frac{kq}{(n-k)p}<\frac{kq}{(n-np)p}=\frac{kq}{nqp}=\frac{k}{np}<1$$

则对 $0<i\leqslant k$，有

$$b(i-1;n,p)<xb(i;n,p)$$

重复应用这个不等式 $k-i$ 次，得到

$$b(i;n,p)<x^{k-i}b(k;n,p)$$

对于 $0\leqslant i<k$，从而有

$$\sum_{i=0}^{k-1}b(i;n,p)<\sum_{i=0}^{k-1}x^{k-i}b(k;n,p)<b(k;n,p)\sum_{i=1}^{\infty}x^i$$

$$=\frac{x}{1-x}b(k;n,p)=\frac{kq}{np-k}b(k;n,p)$$ ■

推论 C.5　考虑 n 个伯努利试验的序列，这里成功的概率是 p，失败的概率是 $q=1-p$。对 $0<k\leqslant np/2$，成功次数少于 k 的概率小于成功次数小于 $k+1$ 次概率的一半。

证明：因为 $k\leqslant np/2$，$q\leqslant 1$，故有

$$\frac{kq}{np-k}\leqslant\frac{(np/2)q}{np-(np/2)}=\frac{(np/2)q}{np/2}\leqslant 1$$

1120 令 X 为表示成功次数的随机变量，定理 C.4 表明成功次数少于 k 的概率是

$$\Pr\{X<k\}=\sum_{i=0}^{k-1}b(i;n,p)<b(k;n,p)$$

从而，因为 $\sum_{i=0}^{k-1}b(i;n,p)<b(k;n,p)$，有

$$\frac{\Pr\{X<k\}}{\Pr\{X<k+1\}}=\frac{\sum_{i=0}^{k-1}b(i;n,p)}{\sum_{i=0}^{k}b(i;n,p)}=\frac{\sum_{i=0}^{k-1}b(i;n,p)}{\sum_{i=0}^{k-1}b(i;n,p)+b(k;n,p)}<1/2$$

因为 $\sum_{i=0}^{k-1}b(i;n,p)<b(k;n,p)$. ■

右尾部可以通过类似方法定界。证明留到练习 C.5-2。

推论 C.6　考虑 n 次伯努利试验的序列，这里成功的概率是 p。令 X 为表示成功总数的随机变量。从而对 $np<k<n$，成功次数多于 k 的概率是

$$\Pr\{X>k\}=\sum_{i=k+1}^{n}b(i;n,p)<\frac{(n-k)p}{k-np}b(k;n,p)$$ ■

推论 C.7　考虑 n 次伯努利试验的序列，这里成功的概率是 p，失败的概率是 $q=1-p$。从而对 $(np+n)/2<k<n$，成功次数多于 k 次的概率小于成功次数多于 $k-1$ 次概率的一半。 ■

下一个定理考虑 n 个伯努利试验，每个成功的概率是 p_i，这里 $i=1, 2, \cdots, n$。正如随后的推论所示，通过设置每个试验的 $p_i=p$，可以利用这个定理来为二项分布的右尾部定界。

定理 C.8 考虑 n 个伯努利试验的序列，这里第 i 次试验成功的概率是 p_i，失败的概率是 $q_i=1-p_i$，$i=1, 2, \cdots, n$。令 X 为表示成功总数的随机变量，$\mu=\mathrm{E}[X]$。对 $r>\mu$ 有

$$\Pr\{X-\mu \geqslant r\} \leqslant \left(\frac{\mu \mathrm{e}}{r}\right)^r \tag{1121}$$

证明：因为对任何 $\alpha>0$，函数 $\mathrm{e}^{\alpha x}$ 随 x 严格递增，

$$\Pr\{X-\mu \geqslant r\} = \Pr\{\mathrm{e}^{\alpha(X-\mu)} \geqslant \mathrm{e}^{\alpha r}\} \tag{C.41}$$

这里的 α 将在后面被确定。使用马尔可夫不等式(C.29)，得到

$$\Pr\{\mathrm{e}^{\alpha(X-\mu)} \geqslant \mathrm{e}^{\alpha r}\} \leqslant \mathrm{E}[\mathrm{e}^{\alpha(X-\mu)}]\mathrm{e}^{-\alpha r} \tag{C.42}$$

证明体包括给 $\mathrm{E}[\mathrm{e}^{\alpha(X-\mu)}]$ 定界和用一个合适的值来替代不等式(C.42)中的 α。首先来估计 $\mathrm{E}[\mathrm{e}^{\alpha(X-\mu)}]$ 的值。利用 5.2 节中的符号，对 $i=1, 2, \cdots, n$，令 $X_i=I\{\text{第 } i \text{ 次伯努利试验成功}\}$，即，如果第 i 次伯努利试验成功则 X_i 的值为 1，反之为 0。因此，

$$X = \sum_{i=1}^{n} X_i$$

根据期望的线性特征，有：

$$\mu = \mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} p_i$$

这意味着

$$X-\mu = \sum_{i=1}^{n} (X_i - p_i)$$

为了估计 $\mathrm{E}[\mathrm{e}^{\alpha(X-\mu)}]$ 的值，我们替换 $X-\mu$，因为随机变量 X_i 相互独立意味着随机变量 $\mathrm{e}^{\alpha(X_i-p_i)}$ 也相互独立(见练习 C.3-5)，则根据式(C.23)得

$$\mathrm{E}[\mathrm{e}^{\alpha(X-\mu)}] = \mathrm{E}\left[\mathrm{e}^{\alpha\sum_{i=1}^{n}(X_i-p_i)}\right] = \mathrm{E}\left[\prod_{i=1}^{n} \mathrm{e}^{\alpha(X_i-p_i)}\right] = \prod_{i=1}^{n} \mathrm{E}[\mathrm{e}^{\alpha(X_i-p_i)}]$$

根据期望的定义有

$$\mathrm{E}[\mathrm{e}^{\alpha(X_i-p_i)}] = \mathrm{e}^{\alpha(1-p_i)}p_i + \mathrm{e}^{\alpha(0-p_i)}q_i = p_i\mathrm{e}^{\alpha q_i} + q_i\mathrm{e}^{-\alpha p_i} \leqslant p_i\mathrm{e}^{\alpha} + 1 \leqslant \exp(p_i\mathrm{e}^{\alpha}) \tag{C.43}$$ 1122

这里 $\exp(x)$ 表示指数函数：$\exp(x)=\mathrm{e}^x$。(不等式(C.43)根据不等式 $\alpha>0$，$q_i \leqslant 1$，$\mathrm{e}^{\alpha q_i} \leqslant \mathrm{e}^{\alpha}$ 和 $\mathrm{e}^{-\alpha q_i} \leqslant 1$ 得到，最后一行由不等式(3.11)得到)。从而，因为 $\mu = \sum_{i=1}^{n} p_i$ 有

$$\mathrm{E}[\mathrm{e}^{\alpha(X-\mu)}] = \prod_{i=1}^{n} \mathrm{E}[\mathrm{e}^{\alpha(X_i-p_i)}] \leqslant \prod_{i=1}^{n} \exp(p_i\mathrm{e}^{\alpha}) = \exp\left(\sum_{i=1}^{n} p_i\mathrm{e}^{\alpha}\right) = \exp(\mu\mathrm{e}^{\alpha}) \tag{C.44}$$

因此，根据等式(C.41)、不等式(C.42)和式(C.44)可知

$$\Pr\{X-\mu \geqslant r\} \leqslant \exp(\mu\mathrm{e}^{\alpha} - \alpha r) \tag{C.45}$$

选择 $\alpha=\ln(r/\mu)$，(见练习 C.5-7)可得

$$\Pr\{X-\mu \geqslant r\} \leqslant \exp\{\mu\mathrm{e}^{\ln(r/\mu)} - r\ln(r/\mu)\} = \exp(r - r\ln(r/\mu)) = \frac{\mathrm{e}^r}{(r/\mu)^r} = \left(\frac{\mu\mathrm{e}}{r}\right)^r \quad \blacksquare$$

当应用到每次试验都有相同成功概率的伯努利试验时，定理 C.8 可产生如下推论，它给出了二项分布右尾部的界。

推论 C.9 考虑 n 次伯努利试验的序列，这里每次试验成功的概率是 p，失败的概率是 $q=1-p$。从而对 $r>np$，有

$$\Pr\{X - np \geqslant r\} = \sum_{k=\lceil np+r \rceil}^{n} b(k;n,p) \leqslant \left(\frac{npe}{r}\right)^r$$

1123　**证明**：根据等式(C.36)，有 $\mu = \mathrm{E}[X] = np$。 ■

练习

***C.5-1**　哪一个可能性更小：抛均质硬币 n 次，但没有一次正面向上，或抛均质硬币 $4n$ 次，得到正面向上的次数少于 n？

***C.5-2**　证明推论 C.6 和推论 C.7。

***C.5-3**　证明对所有 $a>0$ 和所有满足 $0<k<n$ 的 k，有

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na - k(a+1)} b(k;n,a/(a+1))$$

***C.5-4**　证明如果 $0<k<np$，这里 $0<p<1$ 且 $q=1-p$，那么

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np-k}\left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}$$

***C.5-5**　证明定理 C.8 的条件意味着

$$\Pr\{\mu - X \geqslant r\} \leqslant \left(\frac{(n-\mu)e}{r}\right)^r$$

类似地，证明推论 C.9 的条件意味着

$$\Pr\{np - X \geqslant r\} \leqslant \left(\frac{nqe}{r}\right)^r$$

***C.5-6**　考虑 n 次伯努利试验的序列，这里第 i 次试验成功的概率是 p_i，失败的概率是 $q_i = 1 - p_i$，$i=1, 2, \cdots, n$。令 X 是表示成功总数的随机变量，令 $\mu = \mathrm{E}[X]$。证明对 $r \geqslant 0$，

1124

$$\Pr\{X - \mu \geqslant r\} \leqslant e^{-r^2/2n}$$

（提示：证明 $p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leqslant e^{\alpha^2/2}$。然后根据定理 C.8 中的证明思路进行证明，并用这个不等式代替不等式(C.43)）

***C.5-7**　证明当 $a = \ln(r/\mu)$ 时，不等式(C.45)的右部取到最小值。

思考题

C-1　球和盒子

在本问题中，基于不同的假设，考虑一下将 n 个球放入 b 个不同的盒子中这一问题，看看共有多少种不同的放置方法。

a)假设 n 个球是不同的且不考虑它们在盒子中的顺序。证明有 b^n 种方法可将球放入盒子。

b)假设 n 个球是不同的且每个盒子中的球是有顺序的。证明正好有 $(b+n-1)!/(b-1)!$ 种方法将球放入盒子中。（提示：考虑将 n 个不同的球和 $b-1$ 根不可区分的棍子排成 1 排的方法数目）

c)假设球是相同的，从而不许考虑它们在盒子中的顺序。证明将球放入盒子中的方法数是 $\binom{b+n-1}{n}$。（提示：如果球是相同的，那么(b)中的排列有多少是重复的？）

d)假设球是相同的且每个盒子最多能容纳 1 个球。证明将球放入盒子中的方法数是 $\binom{b}{n}$。

e)假设球是相同的且每个盒子都不能为空。证明将球放入盒子中的方法数是$\binom{n-1}{b-1}$。

本章注记

B. Pascal 和 P. de Fermat 在 1654 年的一封著名的书信中，第一次讨论了解决概率问题的一般方法，它出现在 1657 年 C. Huygens 的一本书中。严格的概率理论开始于 J. Bernoulli 在 1713 年和 A. De Moivre 在 1730 年的工作。P. S. de Laplace，S. -D. Poisson 和 C. F. Gauss 继续发展了这一理论。

P. L. Chebyshev 和 A. A. Markov 首先研究了随机变量的和。A. N. Kolmogorov 在 1933 年将概率理论公理化。Chernoff[59]和 Hoeffding[150]提出了分布尾部的界。P. Erdös 在随机组合结构方面做出了开创性的工作。 1125

Knuth[182]和 Liu[205]是基本组合和计数的很好的参考书。像 Billingsley[42]，Chung[60]，Drake[80]，Feller[88]和 Rozanov[263]这样的权威教材对概率论做了全面的介绍。 1126

参 考 文 献

[1] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions*. Dover, 1965.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[3] Leonard M. Adleman, Carl Pomerance, and Robert S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics*, 117:173–206, 1983.

[4] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[6] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[7] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[8] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37:213–223, 1990.

[9] Ravindra K. Ahuja and James B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748–759, 1989.

[10] Ravindra K. Ahuja, James B. Orlin, and Robert E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5):939–954, 1989.

[11] Miklos Ajtai, János Komlós, and Endre Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[12] Miklos Ajtai, Nimrod Megiddo, and Orli Waarts. Improved algorithms and analysis for secretary problems and generalizations. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 473–482, 1995.

[13] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.

[14] Noga Alon. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters*, 35:201–204, 1990.

[15] Arne Andersson. Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, number 709 in Lecture Notes in Computer Science, pages 60–71. Springer-Verlag, 1993.

[16] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 135–141, 1996.

[17] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57:74–93, 1998.

[18] Tom M. Apostol. *Calculus*, volume 1. Blaisdell Publishing Company, second edition, 1967.

[19] Sanjeev Arora. *Probabilistic checking of proofs and the hardness of approximation problems*. PhD thesis, University of California, Berkeley, 1994.

[20] Sanjeev Arora. The approximability of NP-hard problems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 337–348, 1998.

[21] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753–782, 1998.

[22] Sanjeev Arora and Carsten Lund. Hardness of approximations. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 399–446. PWS Publishing Company, 1997.

[23] Javed A. Aslam. A simple bound on the expected height of a randomly built binary search tree. Technical Report TR2001-387, Dartmouth College Department of Computer Science, 2001.

[24] Mikhail J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[25] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, 1999.

[26] Sara Baase and Alan Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, third edition, 2000.

[27] Eric Bach. Private communication, 1989.

[28] Eric Bach. Number-theoretic algorithms. In *Annual Review of Computer Science*, volume 4, pages 119–172. Annual Reviews, Inc., 1990.

[29] Eric Bach and Jeffery Shallit. *Algorithmic Number Theory—Volume I: Efficient Algorithms*. The MIT Press, 1996.

[30] David H. Bailey, King Lee, and Horst D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4):357–371, 1990.

[31] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[32] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[33] Pierre Beauchemin, Gilles Brassard, Claude Crépeau, Claude Goutier, and Carl Pomerance. The generation of random numbers that are probably prime. *Journal of Cryptology*, 1:53–64, 1988.

[34] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[35] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[36] Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 80–86, 1983.

[37] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[38] Samuel W. Bent and John W. John. Finding the median requires $2n$ comparisons. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 213–216, 1985.

[39] Jon L. Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982.

[40] Jon L. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[41] Jon L. Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, 1980.

[42] Patrick Billingsley. *Probability and Measure*. John Wiley & Sons, second edition, 1986.

[43] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[44] Béla Bollobás. *Random Graphs*. Academic Press, 1985.

[45] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. American Elsevier, 1976.

[46] Gilles Brassard and Paul Bratley. *Algorithmics: Theory and Practice*. Prentice-Hall, 1988.

[47] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[48] Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20(2):176–184, 1980.

[49] Mark R. Brown. *The Analysis of a Practical and Nearly Optimal Priority Queue*. PhD thesis, Computer Science Department, Stanford University, 1977. Technical Report STAN-CS-77-600.

[50] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.

[51] J. P. Buhler, H. W. Lenstra, Jr., and Carl Pomerance. Factoring integers with the number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 50–94. Springer-Verlag, 1993.

[52] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[53] Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[54] Joseph Cheriyan and Torben Hagerup. A randomized maximum-flow algorithm. *SIAM Journal on Computing*, 24(2):203–226, 1995.

[55] Joseph Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18(6):1057–1086, 1989.

[56] Boris V. Cherkassky and Andrew V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[57] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[58] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists and monotone priority queues. *SIAM Journal on Computing*, 28(4):1326–1346, 1999.

[59] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

[60] Kai Lai Chung. *Elementary Probability Theory with Stochastic Processes*. Springer-Verlag, 1974.

[61] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[62] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.

[63] V. Chvátal, D. A. Klarner, and D. E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, 1972.

[64] Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30. North-Holland, 1964.

[65] H. Cohen and H. W. Lenstra, Jr. Primality testing and Jacobi sums. *Mathematics of Computation*, 42(165):297–330, 1984.

[66] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[67] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[68] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[69] Don Coppersmith. Modifications to the number field sieve. *Journal of Cryptology*, 6:169–180, 1993.

[70] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[71] Thomas H. Cormen. *Virtual Memory for Data-Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1992.

[72] Eric V. Denardo and Bennett L. Fox. Shortest-route methods: 1. Reaching, pruning, and buckets. *Operations Research*, 27(1):161–186, 1979.

[73] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[74] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[75] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[76] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 11(5):1277–1280, 1970.

[77] Brandon Dixon, Monika Rauch, and Robert E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[78] John D. Dixon. Factorization and primality tests. *The American Mathematical Monthly*, 91(6):333–352, 1984.

[79] Dorit Dor and Uri Zwick. Selecting the median. In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 28–37, 1995.

[80] Alvin W. Drake. *Fundamentals of Applied Probability Theory*. McGraw-Hill, 1967.

[81] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[82] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[83] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1987.

[84] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[85] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126–136, 1971.

[86] Jack Edmonds and Richard M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[87] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[88] William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, third edition, 1968.

[89] Robert W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[90] Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7:701, 1964.

[91] Robert W. Floyd. Permuting information in idealized two-level storage. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109. Plenum Press, 1972.

[92] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975.

[93] Lestor R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[94] Lestor R. Ford, Jr. and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66:387–389, 1959.

[95] Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[96] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[97] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 1989.

[98] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[99] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[100] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[101] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74:107–114, 2000.

[102] Harold N. Gabow, Z. Galil, T. Spencer, and Robert E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[103] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[104] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[105] Zvi Galil and Oded Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2):103–139, 1997.

[106] Zvi Galil and Oded Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54(2):243–254, 1997.

[107] Zvi Galil and Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983.

[108] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 1993.

[109] Michael R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analyis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, pages 143–150, 1972.

[110] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[111] Saul Gass. *Linear Programming: Methods and Applications*. International Thomson Publishing, fourth edition, 1975.

[112] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[113] Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

[114] E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38(4):933–967, 1959.

[115] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

[116] Michel X. Goemans and David P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 144–191. PWS Publishing Company, 1997.

[117] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[118] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.

[119] Andrew V. Goldberg, Éva Tardos, and Robert E. Tarjan. Network flow algorithms. In Bernhard Korte, László Lovász, Hans Jürgen Prömel, and Alexander Schrijver, editors, *Paths, Flows, and VLSI-Layout*, pages 101–164. Springer-Verlag, 1990.

[120] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45:783–797, 1998.

[121] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[122] D. Goldfarb and M. J. Todd. Linear programming. In G. L. Nemhauser, A. H. G. Rinnooy-Kan, and M. J. Todd, editors, *Handbook in Operations Research and Management Science, Vol. 1, Optimization*, pages 73–170. Elsevier Science Publishers, 1989.

[123] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[124] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[125] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[126] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.

[127] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992.

[128] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.

[129] Ronald L. Graham. Bounds for certain multiprocessor anomalies. *Bell Systems Technical Journal*, 45:1563–1581, 1966.

[130] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[131] Ronald L. Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[132] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[133] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[134] M. Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1988.

[135] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

[136] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[137] Yijie Han. Improved fast integer sorting in linear space. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 793–796, 2001.

[138] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[139] Gregory C. Harfst and Edward M. Reingold. A potential-based amortized analysis of the union-find data structure. *SIGACT News*, 31(3):86–95, 2000.

[140] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[141] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the Fast Fourier Transform. *IEEE ASSP Magazine*, pages 14–21, 1984.

[142] Monika R. Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 664–672, 1995.

[143] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[144] Monika R. Henzinger, Satish Rao, and Harold N. Gabow. Computing vertex connectivity: New bounds from old techniques. *Journal of Algorithms*, 34(2):222–250, 2000.

[145] Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transactions on Mathematical Software*, 16(4):352–368, 1990.

[146] C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321–322, 1961.

[147] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[148] Dorit S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Math*, 6:243–254, 1983.

[149] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.

[150] W. Hoeffding. On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics*, 27:713–721, 1956.

[151] Micha Hofri. *Probabilistic Analysis of Algorithms*. Springer-Verlag, 1987.

[152] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[153] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

[154] John E. Hopcroft and Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[155] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[156] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[157] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.

[158] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Computer Algorithms*. Computer Science Press, 1998.

[159] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing*, 11(2):362–373, 1982.

[160] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part II. *SIAM Journal on Computing*, 13(2):228–251, 1984.

[161] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[162] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[163] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *SC96 Technical Papers*, 1996.

[164] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

[165] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2:18–21, 1973.

[166] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[167] David S. Johnson. The NP-completeness column: An ongoing guide—the tale of the second prover. *Journal of Algorithms*, 13(3):502–524, 1992.

[168] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[169] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[170] David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993.

[171] Howard Karloff. *Linear Programming*. Birkäuser, 1991.

[172] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[173] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[174] Richard M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34:165–201, 1991.

[175] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.

[176] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[177] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[178] Valerie King, Satish Rao, and Robert E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17:447–474, 1994.

[179] Jeffrey H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis*. Addison-Wesley, 1990.

[180] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(2):287–299, 1986.

[181] Philip N. Klein and Neal E. Young. Approximation algorithms for NP-hard optimization problems. In *CRC Handbook on Algorithms*, pages 34-1–34-19. CRC Press, 1999.

[182] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968. Second edition, 1973.

[183] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1969. Second edition, 1981.

[184] Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[185] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[186] Donald E. Knuth. Big omicron and big omega and big theta. *ACM SIGACT News*, 8(2):18–23, 1976.

[187] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[188] J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[189] Bernhard Korte and László Lovász. Mathematical structures underlying greedy algorithms. In F. Gecseg, editor, *Fundamentals of Computation Theory*, number 117 in Lecture Notes in Computer Science, pages 205–209. Springer-Verlag, 1981.

[190] Bernhard Korte and László Lovász. Structural properties of greedoids. *Combinatorica*, 3:359–374, 1983.

[191] Bernhard Korte and László Lovász. Greedoids—a structural framework for the greedy algorithm. In W. Pulleybank, editor, *Progress in Combinatorial Optimization*, pages 221–243. Academic Press, 1984.

[192] Bernhard Korte and László Lovász. Greedoids and linear objective functions. *SIAM Journal on Algebraic and Discrete Methods*, 5(2):229–238, 1984.

[193] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992.

[194] David W. Krumme, George Cybenko, and K. N. Venkataraman. Gossiping in minimal time. *SIAM Journal on Computing*, 21(1):111–139, 1992.

[195] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[196] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.

[197] Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[198] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

[199] Tom Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 422–431, 1988.

[200] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[201] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 11–42. Springer-Verlag, 1993.

[202] H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[203] L. A. Levin. Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. In Russian.

[204] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, second edition, 1998.

[205] C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.

[206] László Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[207] László Lovász and M. D. Plummer. *Matching Theory*, volume 121 of *Annals of Discrete Mathematics*. North Holland, 1986.

[208] Bruce M. Maggs and Serge A. Plotkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291–293, 1988.

[209] Michael Main. *Data Structures and Other Objects Using Java*. Addison-Wesley, 1999.

[210] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[211] Conrado Martínez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[212] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[213] H. A. Maurer, Th. Ottmann, and H.-W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1):11–14, 1976.

[214] Ernst W. Mayr, Hans Jürgen Prömel, and Angelika Steger, editors. *Lectures on Proof Verification and Approximation Algorithms*. Number 1367 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[215] C. C. McGeoch. All pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5):426–441, 1995.

[216] M. D. McIlroy. A killer adversary for quicksort. *Software—Practice and Experience*, 29(4):341–344, 1999.

[217] Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

[218] Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

[219] Kurt Mehlhorn. *Multidimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

[220] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[221] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[222] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[223] Joseph S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal on Computing*, 28(4):1298–1309, 1999.

[224] Louis Monier. *Algorithmes de Factorisation D'Entiers*. PhD thesis, L'Université Paris-Sud, 1980.

[225] Louis Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, 12(1):97–108, 1980.

[226] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[227] Rajeev Motwani, Joseph (Seffi) Naor, and Prabakhar Raghavan. Randomized approximation algorithms in combinatorial optimization. In Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 447–481. PWS Publishing Company, 1997.

[228] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge Unveristy Press, 1995.

[229] J. I. Munro and V. Raman. Fast stable in-place sorting with $O(n)$ data moves. *Algorithmica*, 16(2):151–160, 1996.

[230] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[231] Ivan Niven and Herbert S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, fourth edition, 1980.

[232] Alan V. Oppenheim and Ronald W. Schafer, with John R. Buck. *Discrete-Time Signal Processing*. Prentice-Hall, second edition, 1998.

[233] Alan V. Oppenheim and Alan S. Willsky, with S. Hamid Nawab. *Signals and Systems*. Prentice-Hall, second edition, 1997.

[234] James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78(1):109–129, 1997.

[235] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1993.

[236] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[237] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[238] Michael S. Paterson, 1974. Unpublished lecture, Ile de Berder, France.

[239] Michael S. Paterson. Progress in selection. In *Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory*, pages 368–379, 1996.

[240] Pavel A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000.

[241] Steven Phillips and Jeffery Westbrook. Online load balancing and network flow. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 402–411, 1993.

[242] J. M. Pollard. A Monte Carlo method for factorization. *BIT*, 15:331–334, 1975.

[243] J. M. Pollard. Factoring with cubic integers. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 4–10. Springer, 1993.

[244] Carl Pomerance. On the distribution of pseudoprimes. *Mathematics of Computation*, 37(156):587–593, 1981.

[245] Carl Pomerance, editor. *Proceedings of the AMS Symposia in Applied Mathematics: Computational Number Theory and Cryptography*. American Mathematical Society, 1990.

[246] William K. Pratt. *Digital Image Processing*. John Wiley & Sons, second edition, 1991.

[247] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[248] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1986.

[249] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.

[250] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[251] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[252] Paul W. Purdom, Jr. and Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Rinehart, and Winston, 1985.

[253] Michael O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, 1976.

[254] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[255] P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, 1987.

[256] Rajeev Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2):81–87, 1997.

[257] Edward M. Reingold, Jürg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[258] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. Progress in Mathematics. Birkhäuser, 1985.

[259] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. See also U.S. Patent 4,405,829.

[260] Herbert Robbins. A remark on Stirling's formula. *American Mathematical Monthly*, 62(1):26–29, 1955.

[261] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6:563–581, 1977.

[262] Salvador Roura. An improved master theorem for divide-and-conquer recurrences. In *Proceedings of Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, pages 449–459, 1997.

[263] Y. A. Rozanov. *Probability Theory: A Concise Course*. Dover, 1969.

[264] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23:555–565, 1976.

[265] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13(2):184–199, 1976.

[266] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.

[267] Alexander Schrijver. Paths and flows—a historical survey. *CWI Quarterly*, 6:169–183, 1993.

[268] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

[269] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[270] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[271] Raimund Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[272] João Setubal and João Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[273] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall, second edition, 2001.

[274] Jeffrey Shallit. Origins of the analysis of the Euclidean algorithm. *Historia Mathematica*, 21(4):401–419, 1994.

[275] Michael I. Shamos and Dan Hoey. Geometric intersection problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 208–215, 1976.

[276] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7:67–72, 1981.

[277] David B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. In William Cook, László Lovász, and Paul Seymour, editors, *Combinatorial Optimization*, volume 20 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1995.

[278] Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 605–614, 1999.

[279] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[280] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998.

[281] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[282] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[283] Joel Spencer. *Ten Lectures on the Probabilistic Method*. Regional Conference Series on Applied Mathematics (No. 52). SIAM, 1987.

[284] Daniel A. Spielman and Shang-Hua Teng. The simplex algorithm usually takes a polynomial number of steps. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, 2001.

[285] Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.

[286] Gilbert Strang. *Linear Algebra and Its Applications*. Harcourt Brace Jovanovich, third edition, 1988.

[287] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[288] T. G. Szymanski. A special case of the maximal common subsequence problem. Technical Report TR-170, Computer Science Laboratory, Princeton University, 1975.

[289] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[290] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[291] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979.

[292] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[293] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[294] Robert E. Tarjan. Class notes: Disjoint set union. COS 423, Princeton University, 1999.

[295] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

[296] George B. Thomas, Jr. and Ross L. Finney. *Calculus and Analytic Geometry.* Addison-Wesley, seventh edition, 1988.

[297] Mikkel Thorup. Faster deterministic sorting and priority queues in linear space. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555, 1998.

[298] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[299] Mikkel Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[300] Richard Tolimieri, Myoung An, and Chao Lu. *Mathematics of Multidimensional Fourier Transform Algorithms.* Springer-Verlag, second edition, 1997.

[301] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84. IEEE Computer Society, 1975.

[302] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity.* Elsevier Science Publishers and The MIT Press, 1990.

[303] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform.* Society for Industrial and Applied Mathematics, 1992.

[304] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions.* Kluwer Academic Publishers, 1996.

[305] Vijay V. Vazirani. *Approximation Algorithms.* Springer-Verlag, 2001.

[306] Rakesh M. Verma. General techniques for analyzing recursive algorithms with applications. *SIAM Journal on Computing*, 26(2):568–581, 1997.

[307] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[308] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[309] Michael S. Waterman. *Introduction to Computational Biology, Maps, Sequences and Genomes.* Chapman & Hall, 1995.

[310] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++.* Addison-Wesley, 1994.

[311] Mark Allen Weiss. *Algorithms, Data Structures and Problem Solving with C++.* Addison-Wesley, 1996.

[312] Mark Allen Weiss. *Data Structures and Problem Solving Using Java.* Addison-Wesley, 1998.

[313] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-Wesley, 1999.

[314] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509–533, 1935.

[315] Herbert S. Wilf. *Algorithms and Complexity*. Prentice-Hall, 1986.

[316] J. W. J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7:347–348, 1964.

[317] S. Winograd. On the algebraic complexity of functions. In *Actes du Congrès International des Mathématiciens*, volume 3, pages 283–288, 1970.

[318] Andrew C.-C. Yao. A lower bound to finding convex hulls. *Journal of the ACM*, 28(4):780–787, 1981.

[319] Yinyu Ye. *Interior Point Algorithms: Theory and Analysis*. John Wiley & Sons, 1997.

[320] Daniel Zwillinger, editor. *CRC Standard Mathematical Tables and Formulae*. CRC Press, 30th edition, 1996.

索　引

本索引使用下列约定。数字按其英文拼写的字母顺序排序，例如，“2-3-4 树”被当成“two-three-four tree”来索引。当一个索引项涉及的是一个位置而不是正文内容时，页码后面即会跟随一个标记：ex. 表示练习，pr. 表示思考题，fig. 表示图，n. 表示脚注。带标记的页码通常表示一个练习、思考题、图或脚注的第一页，而并不一定是引用真正出现的那个页码。

$\alpha(n)$，511
o-notation(o 记号)，47-48
O-notation(O 记号)，43 fig.，44-45
O'-notation(O' 记号)，59 pr.
$\tilde{O}$-notation($\tilde{O}$ 记号)，59 pr.
ω-notation(ω 记号)，48
Ω-notation(Ω 记号)，43 fig.，45-46
$\overline{\Omega}$-notation($\overline{\Omega}$ 记号)，59 pr.
$\tilde{\Omega}$-notation($\tilde{\Omega}$ 记号)，59 pr.
$\rho(n)$-approximation algorithm(近似算法)，1022
Θ-notation(Θ 记号)，42-44，43 fig.
$\tilde{\Theta}$-notation($\tilde{\Theta}$ 记号)，59 pr.
{}(set)(集合)，1070
$\in$(set member)(集合成员)，1070
$\notin$(not a set member)(非集合成员)，1070
$\varnothing$(empty set)(空集)，1070
$\subseteq$(subset)(子集)，1071
$\subset$(proper subset)(真子集)，1071
：(such that)(满足条件)，1071
$\cap$(set intersection)(集合的交)，1071
$\cup$(set union)(集合的并)，1071
—(set difference)(集合的差)，1071
$|\ |$
(flow value)(流的值)，644
(length of a string)(串的长度)，907
(set cardinality)(集合的势)，1073
$\times$
(Cartesian product)(笛卡儿积)，1074
(cross product)(叉积)，934
<>
(sequence)(序列)，1078
(standard encoding)(标准编码)，975
$\binom{n}{k}$(choose)(选择)，1096
！(factorial)(阶乘)，54
⌈⌉(ceiling)(上取整)，51
⌊⌋(floor)(下取整)，51
Σ(sum)(和)，1058
Π(product)(积)，1061
$\rightarrow$(adjacency relation)，(邻接关系)，1080
$\rightsquigarrow$(reachability relation)(可达关系)，1081
$\wedge$(AND)(与)，633，987
¬(NOT)(非)，987
$\vee$(OR)(或)，633，987
$\oplus$(group operator)(群运算符)，862
$\otimes$(convolution operator)(卷积运算符)，825
* (closure operator)(闭包运算符)，976
| (divides relation)(整除关系)，850
$\nmid$(does-not-divide relation)(非整除关系)，850
$\equiv$(equivalent modulo n)(模 n 等于)，52，1077 ex.
$\not\equiv$(not equivalent modulo n)(模 n 不等于)，52
$[a]_n$(equivalence class modulo n)(模 n 的等价类)，851
$+_n$(addition modulo n)(模 n 加法)，863
$\cdot_n$(multiplication modulo n)(模 n 乘法)，863
$\left(\frac{a}{p}\right)$(Legendre symbol)(勒让德符号)，903 pr.
ε(empty string)(空串)，907，976
$\sqsubset$(prefix relation)(前缀关系)，907
$\sqsupset$(suffix relation)(后缀关系)，907
$>_x$(above relation)(之上关系)，941
▷(comment symbol)(注释符号)，19
$\leqslant_P$(polynomial-time reducibility relation)(多项式时间可归约关系)，984，994 ex.

A

AA-tree(AA 树)，301
abelian group(阿贝尔群或交换群)，862
ABOVE，942
above relation(之上关系)，941
absent child(空孩子)，1089
absolutely convergent series(绝对收敛级数)，1059

absorption laws for sets(集合的吸收律)，1072
abstract problem(抽象问题)，972
acceptable pair of integers(可接受的整数对)，894
acceptance(接受)
  by an algorithm(被一个算法)，976
  by a finite automaton(被一个有穷自动机)，916
accepting state(接受状态)，916
accounting method(记账方法)，410-412
  for binary counters(用于二进制计数器)，411-412
  for dynamic tables(用于动态表)，419
  for stack operations(用于栈操作)，410-411，412 ex.
Ackermann's function(Ackermann 函数)，521
activity-selection problem(活动选择问题)，371-379
acyclic graph(无回路图)，1082
  relation to matroids(和拟阵的关系)，403 pr.
add instruction(加法指令)，22
addition(加法)
  of binary integers(二进制整数)，21 ex.
  of matrices(矩阵)，728
  modulo n ($+_n$)(模 n)，863
  of polynomials(多项式)，822
additive group modulo n(模 n 的加法群)，863
addressing(寻址)，open(开放)，见 open addressing
adjacency-list representation(邻接表表示法)，528
adjacency-matrix representation(邻接矩阵表示法)，529
adjacent vertices(邻接顶点)，1080
admissible edge(容许边)，681
admissible network(容许网络)，681-683
aggregate analysis(聚集分析)，406-410
  for binary counters(二进制计数器)，408-409
  for breadth-first search(广度优先搜索)，534
  for depth-first search(深度优先搜索)，542-543
  for Dijkstra's algorithm(Dijkstra 算法)，598
  for disjoint-set data structures(不相交集合数据结构)，503-504，505 ex.，509 ex.
  for dynamic tables(动态表)，418-419
  for Fibonacci heaps(斐波那契堆)，493 ex.
  for Graham's scan(Graham 扫描)，954-955
  for shortest paths in a dag(有向无环图中的最短路径)，592
  for stack operations(栈操作)，406-408
aggregate flow(汇聚流)，788
Akra-Bazzi method for solving a recurrence(解递归式的 Akra-Bazzi 方法)，89-90
AKS sorting network(AKS 排序网络)，724
algorithm(算法)，5
  correctness of(正确性)，6
  origin of word(字的起源)，40
  running time of(运行时间)，23
  as a technology(作为一项技术)，12
Alice，881
Allocate-Node，442
Allocate-Object，211
allocation of objects(对象的分配)，210-212
all-pairs shortest paths(每对顶点间最短路径)，581，620-642
  in ϵ-dense graphs(ϵ 稠密图)，641 pr.
  Floyd-Warshall algorithm for(Floyd-Warshall 算法)，629-632
  Johnson's algorithm for(Johnson 算法)，636-640
  by matrix multiplication(通过矩阵乘法)，622-629
  by repeated squaring(通过重复平方)，625-627
alphabet(字母表)，916，975
$\alpha(n)$，511
amortized analysis(平摊分析)，405-429
  accounting method of(记账方法)，410-412
  aggregate analysis(聚集分析)，406-410
  for bit-reversal permutation(位反向置换)，425 pr.
  for breadth-first search(广度优先搜索)，534
  for depth-first search(深度优先搜索)，542-543
  for Dijkstra's algorithm(Dijkstra 算法)，598
  for disjoint-set data structures(不相交集合数据结构)，503-504，505 ex.，509 ex.，512-517，518 ex.
  for dynamic tables(动态表)，416-425
  for Fibonacci heaps(斐波那契堆)，479-482，487-488，491-492，493 ex.
  for the generic push-relabel algorithm(一般的 push-relabel 算法)，678-679
  for Graham's scan(Graham 扫描)，954-955
  for the Knuth-Morris-Pratt algorithm(Knuth-Morris-Pratt 算法)，926-927
  for making binary search dynamic(使二分搜索动态化)，426 pr.
  potential method of(势方法)，412-416
  for restructuring red-black trees(重构红黑树)，428 pr.
  for shortest paths in a dag(有向无环图中的最短路径)，592

for stacks on secondary storage(辅存上的栈), 452 pr.
for weight-balanced trees(权平衡树), 427 pr.
amortized cost(平摊代价)
in the accounting method(记账方法), 410
in aggregate analysis(聚集分析), 406
in the potential method(势方法), 413
analysis of algorithms(算法的分析), 21-27
参见 amortized analysis, probabilistic analysis
ancestor(祖先), 1087
least common(最小公共), 521 pr.
AND function ($^{\wedge}$)(AND 函数), 633, 987
AND gate(AND 门), 987
and(而且), in pseudocode(伪码), 20
antisymmetry(反对称性), 1076
ANY-SEGMENTS-INTERSECT, 943
approximation(逼近)
by least squares(通过最小二乘), 762-765
of summation by integrals(通过积分求和), 1067
approximation algorithm(近似算法), 1022-1054
for bin packing(装箱), 1049 pr.
for MAX-CNF satisfiability problem(MAX-CNF 可满足性问题), 1043 ex.
for MAX-CUT problem(MAX-CUT 问题), 1043 ex.
for the maximum-clique problem(最大团问题), 1050 pr.
for maximum matching(最大匹配), 1051 pr.
for MAX-3-CNF satisfiability(MAX-3-CNF 可满足性), 1039-1040
for minimum-weight vertex cover(最小权重顶点覆盖), 1040-1043
for parallel-machine-scheduling problem(并行机调度问题), 1051 pr.
randomized(随机化的), 1039
for the set-covering problem(集合覆盖问题), 1033-1038
for the subset-sum problem(子集和问题), 1043-1049
for the traveling-salesman problem(旅行商问题), 1027-1033
for the vertex-cover problem(顶点覆盖问题), 1024-1027
for the weighted set-covering problem(加权集合覆盖问题), 1050 pr.
approximation ratio(近似率), 1022, 1039
approximation scheme(近似方案), 1023
APPROX-MIN-WEIGHT-VC, 1041
APPROX-SUBSET-SUM, 1046
APPROX-TSP-TOUR, 1028
APPROX-VERTEX-COVER, 1025
arbitrage(套汇), 615 pr.
arc(弧), 见 edge
argument of a function(函数的参数), 1078
arithmetic(算术), modular(模), 51-52, 862-869
arithmetic instructions(算术指令), 22
arithmetic series(等差级数), 1059
arithmetic with infinities(无穷量的计算), 587
arm(臂), 435
array(数组), 19
Monge, 88 pr.
articulation point(挂接点), 558 pr.
assembly-line scheduling(流水线调度), 324-331
assignment(赋值)
multiple(多重), 19
satisfying(可满足性), 988, 996
truth(真值), 988, 996
associative laws for sets(集合的结合律), 1072
associative operation(结合操作), 862
asymptotically larger(渐近大于), 49
asymptotically nonnegative(渐近非负), 42
asymptotically positive(渐近正), 42
asymptotically smaller(渐近小于), 49
asymptotically tight bound(渐近确界), 42
asymptotic efficiency(渐近效率), 41
asymptotic lower bound(渐近下界), 45
asymptotic notation(渐近记号), 41-50, 59 pr.
and graph algorithms(与图算法), 526
and linearity of summations(与和的线性), 1059
asymptotic upper bound(渐近上界), 44
attribute of an object(对象的属性), 20
augmenting data structures(扩张数据结构), 302-318
augmenting path(增广路径), 654, 696 pr.
authentication(鉴别), 251 pr.
automaton(自动机)
finite(有穷的), 916
string-matching(串匹配), 917-922
auxiliary hash function(辅助散列函数), 239
auxiliary linear program(辅助线性规划), 811
average-case running time(平均情况运行时间), 26

AVL-INSERT, 296 pr.
AVL tree(AVL 树), 296 pr.
axioms(公理), for probability(对于概率), 1100

B

back edge(反向边), 546, 550
back substitution(逆向替换), 745
bad guy(坏人), 539 ex.
BAD-SET-COVER-INSTANCE, 1038 ex.
BALANCE, 296 pr.
balanced search tree(平衡搜索树)
  AA-trees(AA 树), 301
  AVL trees(AVL 树), 296 pr.
  B-trees(B 树), 434-454
  k-neighbor trees(k 邻居树), 301
  red-black trees(红黑树), 273-301
  scapegoat trees(替罪羊树), 301
  splay trees(伸展树), 301, 432
  treaps, 296 pr.
  2-3-4 trees(2-3-4 树), 439, 453 pr.
  2-3 trees(2-3 树), 300, 454
  weight-balanced trees(带权平衡树), 301, 427 pr.
balls and bins(球与盒子), 109-110, 1125 pr.
base(基), 350
base-a pseudoprime(以 a 为底的伪素数), 889
base case(基础条件), 64
basic feasible solution(基本可行解), 792
basic solution(基本解), 792
basic variable(基本变量), 782
basis function(基函数), 762
Batcher's odd-even merging network(Batcher 奇偶合并网络), 721 pr.
Bayes's theorem(贝叶斯定理), 1104
BELLMAN-FORD, 588
Bellman-Ford algorithm(Bellman-Ford 算法), 588-592
  for all-pairs shortest paths(每对顶点间最短路径), 620, 639
  in Johnson's algorithm(Johnson 算法中), 639
  and objective functions(与目标函数), 606-607 ex.
  to solve systems of difference constraints(求解差分约束系统), 605
  Yen's improvement to(Yen 氏的改进), 614 pr.
BELOW, 942
Bernoulli trial(伯努利试验), 1112
  and balls and bins(和球与盒子), 109-110
  and streaks(和序列), 110-114
best-case running time(最好情况运行时间), 27 ex., 46
BFS, 532
BIASED-RANDOM, 94 ex.
biconnected component(双连通子图), 558 pr.
big-oh notation(大 O 记号), 43 fig., 44-45
big-omega notation(大 Ω 记号), 43 fig., 45-46
bijective function(双射), 1079
binary character code(二进制字符编码), 385
binary counter(二进制计数器)
  analyzed by accounting method(用记账方法分析), 411-412
  analyzed by aggregate analysis(用汇聚分析做分析), 408-409
  analyzed by potential method(用势方法分析), 414-415
  and binomial heaps(与二项堆), 472 ex.
  bit-reversed(位反向), 425 pr.
binary entropy function(二元熵函数), 1098
binary gcd algorithm(二进制的 gcd 算法), 902 pr.
binary heap(二叉堆), 见 heap
binary relation(二元关系), 1075
binary search(二分搜索), 37 ex.
  with fast insertion(用快速插入), 426 pr.
  in insertion sort(插入排序), 37 ex.
  in searching B-trees(搜索 B 树), 449 ex.
binary search tree(二叉搜索树), 253-272
  AA-trees(AA 树), 301
  AVL trees(AVL 树), 296 pr.
  deletion from(删除), 262-263
  with equal keys(带有相等的关键字), 268 pr.
  insertion into(插入), 261
  k-neighbor trees(k 邻居树), 301
  maximum key of(最大关键字), 258
  minimum key of(最小关键字), 258
  optimal(最优的), 356-363, 369
  predecessor in(前驱), 258-259
  querying(查询), 256-261
  randomly built(随机构造的), 265-268, 270 pr.
  scapegoat trees(替罪羊树), 301
  searching(搜索), 256-258
  for sorting(排序), 264 ex.
  splay trees(伸展树), 301
  successor in(后继), 258-259

and treaps(与 treaps)，296 pr.
weight-balanced trees(加权平衡树)，301
参见 red-black tree
binary-search-tree property(二叉搜索树的性质)，254
vs. min-heap property(与最小堆的性质)，256 ex.
binary tree(二叉树)，1088
full(满的)，1089
number of different ones(不相同的数目)，271 pr.
representation of(表示法)，214
参见 binary search tree
binomial coefficient(二项式系数)，1096-1098
binomial distribution(二项分布)，1113-1116
and balls and bins(和球与盒子)，109
maximum value of(最大值)，1117 ex.
tails of(的尾)，1118-1125
binomial expansion(二项式展开)，1096
binomial heap(二项堆)，455-475
and binary counter and binary addition(与二进制计数器和二进制加法)，472 ex.
creating(构造)，461
decreasing a key in(对其中一个关键字减值)，470
deletion from(从中删除)，470-471
extracting the minimum key from(抽取最小的关键字)，468-469
insertion into(插入)，468
minimum key of(最小关键字)，462
in minimum-spanning-tree algorithm(最小生成树算法)，474 pr.
properties of(性质)，459
running times of operations on(操作的运行时间)，456 fig.
uniting(合并)，462-468
BINOMIAL-HEAP-DECREASE-KEY，470
BINOMIAL-HEAP-DELETE，470
BINOMIAL-HEAP-EXTRACT-MIN，468
BINOMIAL-HEAP-INSERT，468
BINOMIAL-HEAP-MERGE，464
BINOMIAL-HEAP-MINIMUM，462
BINOMIAL-HEAP-UNION，463
BINOMIAL-LINK，462
binomial tree(二项树)，457-459
unordered(无序的)，479
bin packing(装箱)，1049 pr.
bipartite graph(二分图)，1083
corresponding flow network of(对应的流网络)，665
d-regular(d 正则的)，669 ex.
and hypergraphs(与超图)，1084 ex.
bipartite matching(二分匹配)，497，664-669
Hopcroft-Karp algorithm for(Hopcroft-Karp 算法)，696 pr.
birthday paradox(生日悖论)，106-109
biscuit to be kept out of basket(放在篮子外的饼干)，646
bisection of a tree(树的等分)，1092 pr.
bitonic euclidean traveling-salesman problem(双调欧几里得旅行商问题)，364 pr.
bitonic sequence(双调序列)，712
and shortest paths(与最短路径)，618 pr.
BITONIC-SORTER，715
bitonic sorting network(双调排序网络)，712-716
bitonic tour(双调回路)，364 pr.
bit operation(位操作)，850
in Euclid's algorithm(欧几里得算法)，902 pr.
bit-reversal permutation(位反向置换)，425 pr.，841
BIT-REVERSE-COPY，842
bit-reversed binary counter(位反向二进制计数器)，425 pr.
BIT-REVERSED-INCREMENT，425 pr.
bit vector(位向量)，222 ex.
black-height(黑高度)，274
black vertex(黑顶点)，531，540
blocking flow(阻塞流)，697
block structure in pseudocode(伪码中的块结构)，19
Bob，881
Boole's inequality(布尔不等式)，1105 ex.
boolean combinational circuit(布尔组合电路)，988
boolean combinational element(布尔组合元件)，987
boolean connective(布尔连接)，996
boolean formula(布尔公式)，967，983 ex.，996，1002 ex.
boolean function(布尔函数)，1098 ex.
boolean matrix multiplication(布尔矩阵乘法)
and transitive closure(与传递闭包)，759 ex.
Boruvka's algorithm(Boruvka 算法)，578
bottleneck spanning tree(瓶颈生成树)，577 pr.
bottleneck traveling-salesman problem(瓶颈旅行商问题)，1033 ex.
bottom of a stack(栈的底)，200
bound(界)

asymptotically tight(渐近紧确)，42
asymptotic lower(渐近下)，45
asymptotic upper(渐近上)，44
on binomial coefficients(对二项式系数)，1097-1098
on binomial distributions(对二项分布)，1116
polylogarithmic(多项对数的)，54
on the tails of a binomial distribution(对二项分布的尾)，1118-1125
boundary condition(边界条件)，63-64
boundary of a polygon(多边形的边界)，939 ex.
bounding a summation(确定求和时间的界)，1062-1069
box(盒子)，nesting(嵌套)，615 pr.
B^+-tree(B^+ 树)，438
branching factor(分支因子)，in B-tree(在 B 树中)，437
branch instructions(分支指令)，22
breadth-first search(广度优先搜索)，531-539
and shortest paths(与最短路径)，534-537，581
similarity to Dijkstra's algorithm(与 Dijkstra 算法的相似性)，599，600 ex.
breadth-first tree(广度优先树)，532，538
bridge(桥)，558 pr.
B*-tree(B* 树)，439
B-tree(B 树)，434-454
creating(构造)，442
deletion from(删除)，449-452
full node in(满结点)，439
height of(高度)，439-440
insertion into(插入)，443-447
minimum degree of(最小度数)，439
minimum key of(最小关键字)，447 ex.
properties of(性质)，438-441
searching(搜索)，441-442
splitting a node in(分裂结点)，443-445
2-3-4 trees(2-3-4 树)，439
B-TREE-CREATE，442
B-TREE-DELETE，449
B-TREE-INSERT，445
B-TREE-INSERT-NONFULL，446
B-TREE-SEARCH，442，449 ex.
B-TREE-SPLIT-CHILD，444
BUBBLESORT，38 pr.
bucket(桶)，174
bucket sort(桶排序)，174-177
BUCKET-SORT，174
BUILD-MAX-HEAP，133
BUILD-MAX-HEAP′，142 pr.
BUILD-MIN-HEAP，135
butterfly operation(蝴蝶操作)，839

C

cache(高速缓存)，22
cache-oblivious(健忘缓存)，454
call(调用)
a subroutine(子例程)，22，23 n.
by value(传值)，20
cancellation lemma(相消引理)，831
cancellation of flow(流的抵消)，646
canonical form for task scheduling(任务调度的规范型)，399
capacity(容量)
of a cut(切)，655
of an edge(边)，644
residual(剩余)，651，654
capacity constraint(容量约束)，644
cardinality of a set (| |)(集合的势)，1073
Carmichael number(Carmichael 数)，890，896 ex.
Cartesian product (×)(笛卡儿积)，1074
Cartesian sum(笛卡儿和)，830 ex.
cascading cut(连锁切断)，490
CASCADING-CUT，490
Catalan numbers(Catalan 数)，271 pr.，333
ceiling function (⌈ ⌉)(上取整函数)，51
in master theorem(主定理)，81-84
ceiling instruction(取顶指令)，22
certain event(必然事件)，1100
certificate(证书)
in a cryptosystem(加密系统)，886
for verification algorithms(验证算法)，980
CHAINED-HASH-DELETE，226
CHAINED-HASH-INSERT，226
CHAINED-HASH-SEARCH，226
chaining(拉链法)，225-228，250 pr.
chain of a convex hull(凸包链)，955
changing(变化)
a key in a Fibonacci heap(斐波那契堆中的关键字)，497 pr.
variables in the substitution method(替换方法中

的变量)，66
character code(字符编码)，385
child(孩子)，1087，1089
child list in a Fibonacci heap(斐波那契堆中的孩子列表)，477
Chinese remainder theorem(中国余数定理)，873-876
chirp transform(线性调频变换)，838 ex.
choose $\binom{n}{k}$(选择)，1096
chord(弦)，308 ex.
ciphertext(密文)，883
circuit(电路)
  boolean combinational(布尔组合)，988
  for Fast Fourier Transform(快速傅里叶变换)，842-843
CIRCUIT-SAT，989
circuit satisfiability(电路可满足性)，987-994
circular(循环的)，doubly linked list with a sentinel(带哨兵的双向链表)，206
circular linked list(循环链表)，204
  参见 linked list
class(类)
  complexity(复杂性)，977
  equivalence(等价)，1075
classification of edges(边的分类)
  in breadth-first search(广度优先搜索)，558 pr.
  in depth-first search(深度优先搜索)，546-547，548 ex.
clause(子句)，999
clean sequence(清洁序列)，713
clique(团)，1003
CLIQUE，1003
clique problem(团问题)，1003-1006
  approximation algorithm for(近似算法)，1050 pr.
closed interval(闭区间)，311
closed semiring(闭合半环)，642
closest pair(最近的对)，finding(寻找)，957-962
closest-point heuristic(最近点的启发式)，1033 ex.
closure(闭包)
  group property(群的性质)，862
  of a language(语言)，976
  operator(操作符)(*)，976
  transitive(传递的)，见 transitive closure
clustering(群集)，239-240
CNF(conjunctive normal form)(合取范式)，967，999
CNF satisfiability(CNF 可满足性)，1043 ex.
code(编码)，385
  Huffman(赫夫曼)，385-393
codomain(陪域)，1077
coefficient(系数)
  binomial(二项式)，1096
  of a polynomial(多项式)，52，822
  in slack form(松弛型)，783
coefficient representation(系数表示法)，824
  and fast multiplication(与快速乘法)，827-829
cofactor(余子式)，732
coin changing(找换硬币)，402 pr.
collinearity(共线性)，935
collision(碰撞)，224
  resolution by chaining(通过链接解决)，225-228
  resolution by open addressing(通过开放式寻址解决)，237-245
coloring(着色)，1019 pr.，1091 pr.
color of a red-black-tree node(红黑树结点的颜色)，273
column rank(列秩)，731
column vector(列向量)，726
combination(组合)，1096
combinational circuit(组合电路)，988
combinational element(组合元件)，987
comment in pseudocode (▷)(伪码中的注释)，19
commodity(商品)，788
common divisor(公因子)，852
  greatest(最大的)，见 greatest common divisor
common multiple(公倍数)，861 ex.
common subexpression(公共子表达式)，839
common subsequence(公共子序列)，351
  longest(最长的)，350-356，369
commutative laws for sets(集合的交换律)，1071
commutativity under an operator(操作符的可交换性)，862
COMPACTIFY-LIST，213 ex.
compact list(紧凑列表)，218 pr.
COMPACT-LIST-SEARCH，218 pr.
COMPACT-LIST-SEARCH'，219 pr.
comparable line segments(可比较的线段)，941
comparator(比较器)，705
comparison network(比较网络)，704-709
comparison sort(比较排序)，165

and binary search trees(与二叉搜索树), 256 ex.
and mergeable heaps(与可合并堆), 489 ex.
randomized(随机化的), 178 pr.
and selection(与选择), 192
compatible activities(可比较的活动), 371
compatible matrices(可比较的矩阵), 332, 729
complement(补)
of an event(事件), 1101
of a graph(图), 1007
of a language(语言), 976
Schur, 748, 761
of a set(集合), 1072
complementary slackness(互补松弛), 818 pr.
complete graph(完全图), 1083
complete k-ary tree(完全 k 叉图), 1090
参见 heap
completeness of a language(语言的完备性), 994 ex.
completion time(完成时间), 402 pr., 1051 pr.
complexity class(复杂性类), 977
co-NP, 982
NP, 967, 981
NPC, 968, 986
P, 967, 973
complexity measure(复杂性度量), 977
complex numbers(复数), multiplication of(乘法), 741 ex.
complex root of unity(单位复根), 830
interpolation at(插值), 836-837
component(分支)
biconnected(双连通), 558 pr.
connected(连通), 1082
strongly connected(强连通), 1082
component graph(分支图), 554
composite number(合数), 851
witness to(证据), 890
computational geometry(计算几何), 933-965
computational problem(计算问题), 5-6
COMPUTE-PREFIX-FUNCTION, 926
COMPUTE-TRANSITION-FUNCTION, 922
concatenation(串联)
of languages(语言), 976
of strings(字符串), 907
concrete problem(具体问题), 973
conditional branch instruction(条件分支指令), 22
conditional independence(条件独立), 1106 ex.
conditional probability(条件概率), 1103-1104
configuration(配置), 991
conjugate transpose(共轭转置), 759 ex.
conjunctive normal form(合取范式), 967, 999
connected component(连通分支), 1082
identified using depth-first search(用深度优先搜索识别), 549 ex.
identified using disjoint-set data structures(用不相交集合数据结构识别), 499-501
CONNECTED-COMPONENTS, 500
connected graph(连通图), 1082
connective(连接词), 996
co-NP, 982
conservation of flow(流的守恒), 644
consistency of literals(字面上的一致), 1004
CONSOLIDATE, 486
consolidating a Fibonacci-heap root list(合并一个斐波那契堆的根列表), 483
constraint(约束), 777
difference(差分), 602
equality(等式), 606 ex., 778, 780
inequality(不等式), 778, 780
linear(线性), 772
nonnegativity(非负性), 777, 779
tight(紧), 791
violation of(违反), 791
constraint graph(约束图), 603-605
contain(包含), in a path(在路径中), 1081
continuous uniform probability distribution(连续均匀概率分布), 1102
contraction(收缩)
of a dynamic table(动态表的), 420-422
of a matroid(拟阵的), 397
of an undirected graph by an edge(无向图沿一条边的), 1084
control instructions(控制指令), 22
convergence property(收敛性), 587, 609
convergent series(收敛级数), 1059
convex combination of points(点的凸组合), 934
convex function(凸函数), 1109
convex hull(凸包), 947-957, 964 pr.
convex layers(凸层), 962 pr.
convex polygon(凸多边形), 939 ex.
convex set(凸集), 650 ex.
convolution(卷积) ($\otimes$), 825

convolution theorem(卷积定理)，837
copy instruction(复制指令)，22
correctness of an algorithm(算法的正确性)，6
countably infinite set(可数的无穷集合)，1073
counter(计数器)，见 binary counter
counting(计数)，1094-1100
  probabilistic(概率的)，118 pr.
counting sort(计数排序)，168-170
  in radix sort(基数排序中)，172
COUNTING-SORT，168
coupon collector's problem(优惠券收集者问题)，110
cover(覆盖)
  path(路径)，692 pr.
  by a subset(用子集)，1034
  vertex(顶点)，1006，1024，1040-1043
covertical(共垂线的)，942
credit(存款)，410
critical edge(关键边)，662
critical path(关键路径)，594
cross edge(交叉边)，546
crossing a cut(通过割)，563
cross product(叉积)(×)，934
cryptosystem(加密系统)，881-887
cubic spline(三次样条)，767 pr.
curve fitting(曲线拟合)，762-765
cut(割)
  cascading(级联)，490
  of a flow network(流网络)，654-657
  of an undirected graph(无向图)，563
  weight of(权值)，1043 ex.
CUT，489
cutting(分割)，in a Fibonacci heap(斐波那契堆中)，490
cycle of a graph(图的环)，1081-1082
  hamiltonian(哈密顿)，967，979
  minimum mean-weight(最小平均权值)，617 pr.
  negative-weight(负权值)，见 negative-weight cycle
  and shortest paths(与最短路径)，583
cyclic group(循环群)，877
cyclic rotation(循环旋转)，930 ex.
cycling(循环)，of simplex algorithm(单纯形法的)，802
cylinder(柱面)，435

D

dag，见 directed acyclic graph
DAG-SHORTEST-PATHS，592
d-ary heap(*d* 叉堆)，143 pr.
  in shortest-paths algorithms(最短路径算法)，641 pr.
data-movement instructions(数据移动指令)，22
data structure(数据结构)，8，197-318，431-522
  AA-trees(AA 树)，301
  augmentation of(扩张)，302-318
  AVL trees(AVL 树)，296 pr.
  binary search trees(二叉搜索树)，253-272
  binomial heaps(二项堆)，455-475
  bit vectors(位向量)，222 ex.
  B-trees(B 树)，434-454
  deques(双端队列)，204 ex.
  dictionaries(字典)，197
  direct-address tables(直接寻址表)，222-223
  for disjoint sets(不相交集合)，498-522
  for dynamic graphs(动态图)，433
  dynamic sets(动态集合)，197-199
  dynamic trees(动态树)，432
  exponential search trees(指数搜索树)，182，433
  Fibonacci heaps(斐波那契堆)，476-497
  fusion trees(融合树)，182，433
  hash tables(散列表)，224-229
  heaps(堆)，127-144
  interval trees(区间树)，311-317
  k-neighbor trees(*k* 邻居树)，301
  linked lists(链表)，204-209
  order statistic trees(顺序统计树)，302-308
  persistent(持久的)，294 pr.，432
  potential of(势)，413
  priority queues(优先级队列)，138-142
  queues(队列)，200-203
  radix trees(基树)，269 pr.
  red-black trees(红黑树)，273-301
  relaxed heaps(松弛堆)，497
  rooted trees(有根树)，214-217
  scapegoat trees(scapegoat 树)，301
  on secondary storage(辅助存储器)，434-437
  skip lists(跳表)，301
  splay trees(伸展树)，301，432
  stacks(栈)，200-201
  treaps，296 pr.

2-3-4 heaps(2-3-4 堆)，473 pr.
2-3-4 trees(2-3-4 树)，439，453 pr.
2-3 trees(2-3 树)，300，454
van Emde Boas，433
weight-balanced trees(带权平衡树)，301
deadline(最后期限)，399
deallocation of objects(对象的释放)，210-212
decision by an algorithm(由一个算法决定)，976
decision problem(决策问题)，969，972
and optimization problems(与优化问题)，969
decision tree(决策树)，166-167
zero-one principle for(的 0-1 原理)，712 ex.
DECREASE-KEY，138，455
decreasing a key(对一个关键字减值)
in binomial heaps(二项堆)，470
in Fibonacci heaps(斐波那契堆)，489-492
in 2-3-4 heaps(2-3-4 堆)，473 pr.
DECREMENT，409 ex.
degeneracy(退化)，802
degree(度)
of a binomial-tree root(二项树根)，457
maximum(最大)，of a Fibonacci heap(斐波那契堆)，479，488 ex.，493-496
minimum(最小)，of a B-tree(B 树)，439
of a node(结点)，1088
of a polynomial(多项式)，52，822
of a vertex(顶点)，1081
degree-bound(次数界)，822
DELETE，198，455
DELETE-LARGER-HALF，416 ex.
deletion(删除)
from binary search trees(二叉搜索树)，262-263
from binomial heaps(二项堆)，470-471
from B-trees(B 树)，449-452
from chained hash tables(链接的散列表)，226
from direct-address tables(直接寻址表)，222
from dynamic tables(动态表)，422
from Fibonacci heaps(斐波那契堆)，492，496 pr.
from heaps(堆)，142 ex.
from interval trees(区间树)，313
from linked lists(链表)，206
from open-address hash tables(开放式寻址散列表)，238
from order-statistic trees(顺序统计树)，307
from queues(队列)，201
from red-black trees(红黑树)，288-294
from stacks(栈)，200
from sweep-line statuses(扫描线状态)，942
from 2-3-4 heaps(2-3-4 树)，473 pr.
demand paging(请求页面)，22
DeMorgan's laws(德·摩根律)，1072
dense graph(稠密图)，527
ε-dense(ε 稠密)，641 pr.
density of prime numbers(素数的密度)，887-888
dependence(相关)
and indicator random variables(与指示器随机变量)，96
linear(线性)，731
参见 independence
depth(深度)
average(平均)，of a node in a randomly built binary search tree(随机构造的二叉搜索树中结点)，270 pr.
of a comparison network(比较网络)，707
of a node in a rooted tree(有根树的结点)，1088
of quicksort recursion tree(快速排序递归树)，153 ex.
of SORTER，720 ex.
of a sorting network(排序网络)，708 ex.
of a stack(栈)，162 pr.
depth-determination problem(深度确定问题)，519 pr.
depth-first forest(深度优先森林)，540
depth-first search(深度优先搜索)，540-549
in finding articulation points，bridges，and biconnected components(寻找挂接点、桥和双连通子图)，558 pr.
in finding strongly connected components(寻找强连通分支)，552-557
in topological sorting(拓扑排序)，549-552
depth-first tree(深度优先树)，540
deque(双端队列)，204 ex.
DEQUEUE，203
derivative of series(级数的导数)，1060
descendant(子孙)，1087
destination vertex(目的顶点)，581
det，见 determinant
determinant(行列式)，732
and matrix multiplication(与矩阵乘法)，759 ex.
deterministic(确定性的)，99
DETERMINISTIC-SEARCH，118 pr.

DFS, 541
DFS-VISIT, 541
DFT(Discrete Fourier Transform)(离散傅里叶变换), 833
diagonal matrix(对角矩阵), 726
  LUP decomposition of(LUP 分解), 754 ex.
diameter of a tree(树的直径), 539 ex.
dictionary(字典), 197
difference constraints(差分约束), 601-607
difference equation(差分方程), 见 recurrence
difference of sets(集合的差) (-), 1071
  symmetric(对称的), 696 pr.
differentiation of series(级数的微分), 1060
digital signature(数字签名), 883
digraph(有向图), 见 directed graph
DIJKSTRA, 595
Dijkstra's algorithm(Dijkstra 算法), 595-601
  for all-pairs shortest paths(每对顶点间最短路径), 620, 639
  implemented with a Fibonacci heap(用斐波那契堆实现), 599
  implemented with a min-heap(用最小堆实现), 599
  with integer edge weights(带整数边权值), 600-601 ex.
  in Johnson's algorithm(Johnson 算法), 639
  similarity to breadth-first search(与广度优先搜索的相似性), 599, 600 ex.
  similarity to Prim's algorithm(与 Prim 算法的相似性), 570, 599
DIRECT-ADDRESS-DELETE, 222
direct addressing(直接寻址), 222-223
DIRECT-ADDRESS-INSERT, 222
DIRECT-ADDRESS-SEARCH, 222
direct-address table(直接地址表), 222-223
directed acyclic graph (有向无环图)(dag), 1083
  and back edges(与反向边), 550
  and component graphs(与分支图), 554
  and hamiltonian-path problem(与哈密顿路径问题), 983 ex.
  single-source shortest-paths algorithm for(单源最短路径算法), 592-595
  topological sort of(拓扑排序), 549-552
directed graph(有向图), 1080
  all-pairs shortest paths in(每对顶点间最短路径), 620-642
  constraint graph(约束图), 603
  Euler tour of(欧拉回路), 559 pr., 966
  hamiltonian cycle of(哈密顿环), 967
  and longest paths(与最长路径), 966
  path cover of(路径覆盖), 692 pr.
  PERT chart(PERT 图), 594, 594 ex.
  semiconnected(半连通的), 557 ex.
  shortest path in(最短路径), 580
  single-source shortest paths in(单源最短路径), 580-619
  singly connected(单连通图), 549 ex.
  square of(平方), 530 ex.
  transitive closure of(传递闭包), 632
  transpose of(转置), 530 ex.
  参见 circuit, directed acyclic graph, graph, network
directed segment(有向线段), 934-935
directed version of an undirected graph(无向图的有向版本), 1082
DIRECTION, 937
DISCHARGE, 683
discharge of an overflowing vertex(溢出顶点的释放), 683
discovered vertex(被发现的顶点), 531, 540
discovery time(发现时间), in depth-first search(深度优先搜索), 541
Discrete Fourier Transform(离散傅里叶变换), 833
discrete logarithm(离散对数), 877
discrete logarithm theorem(离散对数定理), 877
discrete probability distribution(离散概率分布), 1101
discrete random variable(离散随机变量), 1106-1111
disjoint-set data structure(不相交集合数据结构), 498-522
  analysis of(分析), 512-517, 518 ex.
  in depth determination(深度确定), 519 pr.
  disjoint-set-forest implementation of(不相交集合森林实现), 505 509
  in Kruskal's algorithm(Kruskal 算法), 569
  linear-time special case of(线性时间特例), 522
  linked-list implementation of(链表实现), 501-505
  in off-line least common ancestors(脱机最小公共祖先), 521 pr.
  in off-line minimum(脱机最小), 518 pr.
  in task scheduling(任务调度), 404 pr.

disjoint-set forest(不相交集合森林)，505-509
  analysis of(分析)，512-517，518 ex.
  rank properties of(秩性质)，511-512，518 ex.
  参见 disjoint-set data structure
disjoint sets(不相交集合)，1073
disjunctive normal form(析取范式)，1000
disk(磁盘)，947 ex.
  参见 secondary storage
DISK-READ，437
DISK-WRITE，437
distance(距离)
  edit(编辑)，364 pr.
  euclidean(欧几里得)，957
  L_∞，962 ex.
  Manhattan(曼哈顿)，194 pr.，962 ex.
  of a shortest path(最短路径)，534
distribution(分布)
  binomial(二项)，1113-1116
  geometric(几何)，1112
  of inputs(输入)，93，99
  of prime numbers(素数)，888
  probability(概率)，1100-1102
  sparse-hulled(稀疏包)，964 pr.
distributive laws for sets(集合的分配律)，1072
divergent series(发散级数)，1059
divide-and-conquer method(分治法)，28-33
  analysis of(分析)，32-33
  for binary search(二叉搜索)，37 ex.
  for conversion of binary to decimal(二进制到十进制的转换)，856 ex.
  for Fast Fourier Transform(快速傅里叶变换)，834-836
  for finding the closest pair of points(寻找最近点对)，958-961
  for finding the convex hull(寻找凸包)，948
  for matrix inversion(矩阵的逆)，756-758
  for merge sort(合并排序)，28-36
  for multiplication(乘法)，844 pr.
  for quicksort(快速排序)，145-164
  and recursion trees(与递归树)，67
  relation to dynamic programming(与动态规划的关系)，323
  for selection(选择)，185-193
  solving recurrences for(求解递归式)，62-90
  for Strassen's algorithm(Strassen 算法)，735-742
divide instruction(除法指令)，22
divides relation(整除关系)($|$)，850
division method(除法)，230-231
division theorem(除法定理)，851
divisor(因子)，850-851
  common(公共)，852
  参见 greatest common divisor
DNA，350，364 pr.
DNF (disjunctive normal form)(析取范式)，1000
does-not-divide relation(非整除关系)($\nmid$)，850
domain(域)，1077
dominates relation(支配关系)，962 pr.
double hashing(双散列)，240-241，244 ex.
doubly linked list(双向链表)，204
  参见 linked list
d regular graph(d 正则图)，669 ex.
duality(对偶性)，804-811
  weak(弱)，805
dual linear program(对偶线性规划)，805
dynamic graph(动态图)，499 n.
  data structures for(数据结构)，433
  minimum-spanning-tree algorithm for(最小生成树算法)，574 ex.
  transitive closure of(传递闭包)，641 pr.
dynamic order statistics(动态顺序统计)，302-308
dynamic-programming method(动态规划方法)，323-369
  for activity selection(活动选择)，378 ex.
  for all-pairs shortest paths(每对顶点间最短路径)，622-632
  for assembly-line scheduling(流水线调度)，324-331
  for bitonic euclidean traveling-salesman problem(双调欧几里得旅行商问题)，364 pr.
  compared to greedy algorithms(与贪心算法比较)，341，350 ex.，373-375，380，382-383
  for edit distance(编辑距离)，364 pr.
  elements of(基础)，339-350
  for Floyd-Warshall algorithm(Floyd-Warshall 算法)，629-632
  for longest common subsequence(最长公共子序列)，350-356
  for matrix-chain multiplication(矩阵链乘法)，331-339
  and memoization(与记忆化)，347-349

for optimal binary search trees(最优二叉搜索树), 356-363
optimal substructure in(最优子结构), 339-344
overlapping subproblems in(重叠子问题), 344-346
for printing neatly(整齐输出), 364 pr.
reconstructing an optimal solution in(重新构造一个最优解), 346-347
for scheduling(调度), 369 pr.
for transitive closure(传递闭包), 632-635
for Viterbi algorithm(Viterbi 算法), 367 pr.
for 0-1 knapsack problem(0-1 背包问题), 384 ex.
dynamic set(动态集合), 197-199
参见 data structure
dynamic table(动态表), 416-425
analyzed by accounting method(用记账方法分析), 419
analyzed by aggregate analysis(用汇聚分析做分析), 418-419
analyzed by potential method(用势方法分析), 419-420, 422 424
load factor of(装载因子), 417
dynamic tree(动态树), 432

E

e, 53
E [] (expected value)(期望值), 1108
early-first form(早优先形式), 399
early task(早任务), 399
edge(边), 1080
admissible(容许的), 681
back(反向的), 546
bridge(桥), 558 pr.
capacity of(容量), 644
classification in breadth-first search(广度优先搜索中的分类), 558 pr.
classification in depth-first search(深度优先搜索中的分类), 546-547
critical(关键的), 662
cross(交叉的), 546
forward(正向的), 546
inadmissible(不容许的), 681
light(轻的), 563
negative-weight(负的权), 582-583
residual(剩余的), 652
safe(安全的), 562
saturated(饱和的), 672
tree(树), 538, 540, 546
weight of(权), 529
edge connectivity(边的连通性), 664 ex.
edge set(边集合), 1080
edit distance(编辑距离), 364 pr.
Edmonds-Karp algorithm (Edmonds-Karp 算法), 660-663
elementary event(基本事件), 1100
element of a set(集合的元素) ($\in$), 1070
ellipsoid algorithm(椭圆面算法), 776
elliptic-curve factorization method(椭圆曲线因式分解方法), 905
else(else 子句), in pseudocode(伪码), 19
empty language(空语言) ($\emptyset$), 976
empty set(空集) ($\emptyset$), 1070
empty set laws(空集律), 1071
empty stack(空栈), 201
empty string(空串) (ε), 907, 975
empty tree(空树), 1089
encoding of problem instances(问题实例的编码), 973-975
endpoint(端点)
of an interval(区间), 311
of a line segment(线段), 934
ENQUEUE, 203
entering a vertex(进入一个顶点), 1080
entering variable(换入变量), 793
entropy function(熵函数), 1098
ε-dense graph(ε 稠密图), 641 pr.
equality(相等)
of functions(函数), 1078
linear(线性), 772
of sets(集合), 1070
equality constraint(等式约束), 606 ex., 778
and inequality constraints(与不等式约束), 780
tight(紧确的), 791
violation of(违反), 791
equation(方程)
and asymptotic notation(与渐近记号), 46-47
normal(正态), 764
recurrence(递归), 见 recurrence
equivalence (等价), modular (模) ($\equiv$), 52, 1077 ex.

equivalence class(等价类)，1075
  modulo n(模 n) ($[a]_n$)，851
equivalence relation(等价关系)，1075-1076
  and modular equivalence(与模等价)，1077 ex.
equivalent linear programs(等价的线性规划)，779
escape problem(逃跑问题)，692 pr.
essential term(基本项)，738
EUCLID，858
Euclid's algorithm (欧几里得算法)，856-862，902 pr.
euclidean distance(欧几里得距离)，957
euclidean norm(欧几里得范数)，730
Euler's phi function(欧拉 phi 函数)，865
Euler's theorem(欧拉定理)，877，896 ex.
Euler tour(欧拉回路)，559 pr.，966
  and hamiltonian cycles(与哈密顿环)，966
ϵ-universal hash function(ϵ 全域散列函数)，236 ex.
evaluation of a polynomial(多项式的求值)，39 pr.，824，829 ex.
  and its derivatives(与它的导数)，845 pr.
  at multiple points(多点)，846 pr.
event(事件)，1100
event point(事件点)，942
event-point schedule(事件点调度)，942
EXACT-SUBSET-SUM，1045
excess flow(余流)，669
exchange property(交换性质)，393
exclusion and inclusion(容斥)，1074 ex.
execute a subroutine(执行子例程)，23 n.
expansion of a dynamic table(动态表的扩张)，417-418
expectation(期望)，见 expected value
expected running time(期望运行时间)，26
expected value(期望值)，1108-1109
  of a binomial distribution(二项分布)，1114
  of a geometric distribution(几何分布)，1112
  of an indicator random variable(指示器随机变量)，95
explored vertex(探测过的顶点)，542
exponential function(指数函数)，52-53
exponential height(指数高度)，265
exponential search tree(指数搜索树)，182，433
exponential series(指数级数)，1060
exponentiation(取幂)
  modular(模)，879
exponentiation instruction(取幂指令)，22
EXTENDED-EUCLID，860
EXTEND-SHORTEST-PATHS，624
extension of a set(集合的扩张)，394
exterior of a polygon(多边形的外边界)，939 ex.
external node(外结点)，1088
external path length(外部路径长度)，1091 ex.
extracting the maximum key(抽取最大关键字)
  from d-ary heaps(d 叉堆)，143 pr.
  from max-heaps(最大堆)，139
extracting the minimum key(抽取最小关键字)
  from binomial heaps(二项堆)，468-469
  from Fibonacci heaps(斐波那契堆)，482 488
  from 2-3-4 heaps(2-3-4 堆)，473 pr.
  from Young tableaus(Young 氏矩阵)，143 pr.
EXTRACT-MAX，138-139
EXTRACT-MIN，138，455

F

factor(因子)，851
  twiddle(旋转)，836
factorial function (阶乘函数)(!)，54-55
factorization(因式分解)，896-901，905
  unique(唯一的)，854
failure in a Bernoulli trial(伯努利试验失败)，1112
fair coin(均质硬币)，1101
fan-out(扇出)，988
Farkas's lemma(Farkas 引理)，819 pr.
farthest-pair problem(最远对问题)，948
FASTER-ALL-PAIRS-SHORTEST-PATHS，627，628 ex.
FASTEST-WAY，329
Fast Fourier Transform (FFT)(快速傅里叶变换)
  circuit for(电路)，842-843
  iterative implementation of(迭代实现)，839-842
  multidimensional(多维的)，845 pr.
  recursive implementation of(递归实现)，834-836
  using modular arithmetic(利用取模运算)，847 pr.
feasibility problem(可行性问题)，601，818 pr.
feasible linear program(可行线性规划)，778
feasible region(可行区域)，773
feasible solution(可行解)，601，773，778
Fermat's theorem(费马定理)，877
FFT，见 Fast Fourier Transform

FIB-HEAP-CHANGE-KEY, 497 pr.
FIB-HEAP-DECREASE-KEY, 489
FIB-HEAP-DELETE, 492
FIB-HEAP-EXTRACT-MIN, 483
FIB-HEAP-INSERT, 480
FIB-HEAP-LINK, 486
FIB-HEAP-PRUNE, 497 pr.
FIB-HEAP-UNION, 482
Fibonacci heap(斐波那契堆), 476-497
  changing a key in(改变一个关键字), 497 pr.
  creating(构造), 480
  decreasing a key in(对其中一个关键字减值), 489 492
  deletion from(删除), 492, 496 pr.
  in Dijkstra's algorithm(Dijkstra 算法), 599
  extracting the minimum key from(抽取最小关键字), 482-488
  insertion into(插入), 480-481
  in Johnson's algorithm(Johnson 算法), 640
  maximum degree of(最大度数), 479, 493-496
  minimum key of(最小关键字), 481
  potential function for(势函数), 479
  in Prim's algorithm(Prim 算法), 573
  pruning(剪枝), 497 pr.
  running times of operations on(操作的运行时间), 456 fig.
  uniting(合并), 481-482
Fibonacci numbers(斐波那契数), 56, 86 pr., 494
  computation of(计算), 902 pr.
field of an object(对象的域), 20
FIFO (first-in, first-out)(先进先出), 200
  参见 queue
final-state function(最终状态函数), 917
FIND-DEPTH, 519 pr.
find path(发现路径), 506
FIND-SET, 499
  disjoint-set-forest implementation of(不相交集合森林实现), 508, 522
  linked-list implementation of(链表实现), 501
finished vertex(完成顶点), 540
finishing time(完成时间), in depth-first search(深度优先搜索), 541
  and strongly connected components(与强连通分支), 555
finish time(完成时间), in activity selection(活动选择), 371
finite automaton(有穷自动机), 916
  for string matching(串匹配), 917-922
FINITE-AUTOMATON-MATCHER, 919
finite group(有限群), 862
finite sequence(有穷序列), 1078
finite set(有穷集), 1073
first-fit heuristic(首先适合启发式), 1049 pr.
first-in(先进), first-out(先出), 200
  参见 queue
fixed-length code(固定长度编码), 385
floating-point data type(浮点数据类型), 22
floor function(下取整函数) $\lfloor \rfloor$, 51
  in master theorem(主定理), 81-84
floor instruction(下取整指令), 22
flow(流), 644-650
  aggregate(汇聚), 788
  blocking(阻塞), 697
  excess(溢出), 669
  integer-valued(整数值的), 666
  sum(和), 650 ex.
  total net(总净的), 645
  total positive(总正的), 645
  value of(值), 644
flow conservation(流守恒), 644
flow network(流网络), 644-650
  corresponding to a bipartite graph(相应于二分图), 665
  cut of(割), 654-657
  with negative capacities(负容量), 695 pr.
FLOYD-WARSHALL, 630
FLOYD-WARSHALL', 635 ex.
Floyd-Warshall algorithm (Floyd-Warshall 算法), 629-632, 634 ex.
for(for 循环)
  and loop invariants(与循环不变量), 18 n.
  in pseudocode(伪码), 19
FORD-FULKERSON, 658
Ford-Fulkerson method (Ford-Fulkerson 方法), 651-664
FORD-FULKERSON-METHOD, 651
forest(森林), 1083, 1085
  depth-first(深度优先), 540
  disjoint-set(不相交集合), 505-509
formal power series(形式幂级数), 86 pr.

formula-satisfiability problem(公式可满足性问题), 996-998
forward edge(正向边), 546
forward substitution(正向替换), 744-745
fractional knapsack problem(部分背包问题), 382, 384 ex.
freeing of objects(对象的释放), 210-212
free list(释放列表), 211
FREE-OBJECT, 212
free tree(释放树), 1083, 1085-1087
frequency domain(频率域), 822
full binary tree(满二叉树), 1089, 1091 ex.
  relation to optimal code(和最优编码的关系), 386
full node(满结点), 439
full rank(满秩), 731
full walk of a tree(树的完全遍历), 1030
fully parenthesized(完全括号化), 331
fully polynomial-time approximation scheme(完全多项式时间近似方案), 1023
  for the maximum-clique problem(最大团问题), 1050 pr.
  for the subset-sum problem(子集和问题), 1043-1049
function(函数), 1077-1080
  Ackermann's, 521
  basis(基), 762
  convex(凸的), 1109
  linear(线性的), 25, 772
  objective(目标), 见 objective function
  prefix(前缀), 923-925
  quadratic(二次的), 25
  suffix(后缀), 917
  transition(变换), 916
functional iteration(函数迭代), 55
fundamental theorem of linear programming(线性规划的基本定理), 816
fusion tree(聚合树), 182, 433
fuzzy sorting(模糊排序), 163 pr.

G

Gabow's scaling algorithm for single-source shortest paths(单源最短路径的 Gabow 定标算法), 615 pr.
gap character(间隔字符), 910 ex., 923 ex.
gap heuristic(间隔启发式), 691 ex.
garbage collection(垃圾收集), 127, 210
gate(门), 987
Gaussian elimination(高斯消元), 747
gcd, 852-853, 856 ex.
  参见 greatest common divisor
general number-field sieve(一般数域的筛选), 905
generating function(生成函数), 86 pr.
generator(生成元)
  of a subgroup(子群), 867
  of $\mathbb{Z}_n^*$ ($\mathbb{Z}_n^*$), 877
GENERIC-MST, 563
GENERIC-PUSH-RELABEL, 674
generic push-relabel algorithm(一般性先流推进算法), 673-681
geometric distribution(几何分布), 1112
  and balls and bins(与球和盒子), 109
geometric series(几何级数), 1060
gift wrapping(礼物包装), 955
global variable(全局变量), 19
Goldberg's algorithm(Goldberg 算法), 见 push-relabel algorithm
golden ratio(黄金分割率)(ϕ), 56, 86 pr.
good guy(优秀选手), 539 ex.
gossiping(传布消息), 429
GRAFT, 519 pr.
Graham's scan(Graham 扫描), 949-955
GRAHAM-SCAN, 949
graph(图), 1080-1084
  adjacency-list representation of(邻接表表示法), 528
  adjacency-matrix representation of(邻接矩阵表示法), 529
  algorithms for(算法), 525-698
  breadth-first search of(广度优先搜索), 531-539
  complement of(补), 1007
  component(分支), 554
  constraint(约束), 603-605
  dense(稠密), 527
  depth-first search of(深度优先搜索), 540-549
  dynamic(动态的), 499 n.
  ϵ-dense(ϵ 稠密的), 641 pr.
  hamiltonian(哈密顿), 979
  incidence matrix of(关联矩阵), 403 pr., 531 ex.
  interval(区间), 379 ex.
  nonhamiltonian(非哈密顿), 979
  shortest path in(最短路径), 535

singly connected(单连通的)，549 ex.
sparse(稀疏的)，527
static(静止的)，499 n.
tour of(回路)，1012
weighted(带权的)，529
参见 directed acyclic graph，directed graph，flow network，undirected graph，tree
graph-coloring problem(图着色问题)，1019 pr.
graphic matroid(图拟阵)，393，579
GRAPH-ISOMORPHISM，982 ex.
gray vertex(灰色顶点)，531，540
greatest common divisor (最大公因子)(gcd)，852-853
binary gcd algorithm for(二进制 gcd 算法)，902 pr.
Euclid's algorithm for(欧几里得算法)，856-862
with more than two arguments(多于两个参数)，861 ex.
recursion theorem for(递归定理)，857
greedoid(广义拟阵)，404
GREEDY，396
GREEDY-ACTIVITY-SELECTOR，378
greedy algorithm(贪心算法)，370-404
for activity selection(活动选择)，371-379
for coin changing(找换硬币)，402 pr.
compared to dynamic programming(与动态规划比较)，341，350 ex.，373-375，380，382-383
Dijkstra's algorithm(Dijkstra 算法)，595-601
elements of(原本内容)，379-384
for fractional knapsack problem(用于部分背包问题)，382
greedy-choice property in(贪心选择性质)，380-381
for Huffman code(赫夫曼编码)，385-393
Kruskal's algorithm(Kruskal 算法)，568-570
and matroids(与拟阵)，393-399
for minimum spanning tree(最小生成树)，561
optimal substructure in(最优子结构)，381-382
Prim's algorithm(Prim 算法)，570-573
for the set-covering problem(集合覆盖问题)，1033-1038
for task scheduling(任务调度)，399-402，402 pr.，404 pr.
on a weighted matroid(在带权拟阵上)，395-398
for the weighted set-covering problem(用于带权集合覆盖问题)，1050 pr.
greedy-choice property(贪心选择性质)，380-381
of Huffman codes(赫夫曼编码)，388-390
of a weighted matroid(加权拟阵)，396-397
GREEDY-SET-COVER，1035
grid(网格)，692 pr.
group(群)，862-869
cyclic(循环的)，877

H

half 3-CNF satisfiability(半 3-CNF 可满足性)，1018 ex.
HALF-CLEANER，713
half-open interval(半开区间)，311
Hall's theorem(Hall 定理)，669 ex.
halting problem(停机问题)，966
halving lemma(折半引理)，832
HAM-CYCLE，979
hamiltonian cycle(哈密顿环)，967，979
hamiltonian-cycle problem(哈密顿环问题)，979，1008-1012
hamiltonian graph(哈密顿图)，979
hamiltonian path(哈密顿路径)，983 ex.
hamiltonian-path problem(哈密顿路径问题)，1017 ex.
HAM-PATH，983 ex.
handle(柄)，139，456，477
handshaking lemma(握手引理)，1084 ex.
harmonic number(调和数)，1060
harmonic series(调和级数)，1060
HASH-DELETE，244 ex.
hash function(散列函数)，224，229-237
auxiliary(辅助的)，239
division method for(除法方法)，230-231
e-universal(e 全域)，236 ex.
multiplication method for(乘法方法)，231-232
one-way(单路)，886
universal(全域)，232-236
hashing(散列)，221-252
chaining(链接)，225-228，250 pr.
double(双)，240-241，244 ex.
k-universal(k 全域)，251 pr.
open addressing(开放寻址)，237-245
perfect(完全的)，245-249
universal(全域)，232-236
HASH-INSERT，238，244 ex.

HASH-SEARCH，238，244 ex.
hash table(散列表)，224-229
  dynamic(动态的)，424 ex.
  secondary(副的)，245
  参见 hashing
hash value(散列值)，224
hat-check problem(帽子检查问题)，98 ex.
head(头)
  in a disk drive(磁盘驱动器)，435
  of a linked list(链表)，204
  of a queue(队列)，202
heap(堆)，127-144
  analyzed by potential method(用势方法分析)，416 ex.
  binomial(二项的)，参见 binomial heap
  building(构造)，132-135，142 pr.
  d-ary，143 pr.，641 pr.
  deletion from(删除)，142 ex.
  in Dijkstra's algorithm(Dijkstra 算法)，599
  extracting the maximum key from(抽取最大关键字)，139
  Fibonacci(斐波那契)，参见 Fibonacci heap
  as garbage-collected storage(作为垃圾收集存储)，127
  height of(高度)，129
  in Huffman's algorithm(在赫夫曼算法中)，388
  to implement a mergeable heap(实现可合并的堆)，455
  increasing a key in(增大一个关键字)，139-140
  insertion into(插入)，140
  in Johnson's algorithm(Johnson 算法)，640
  max-heap(最大堆)，128
  maximum key of(最大关键字)，139
  mergeable(可合并的)，见 mergeable heap
  min-heap(最小堆)，129
  in Prim's algorithm(Prim 算法)，573
  as a priority queue(作为一个优先级队列)，138-142
  relaxed(松弛的)，497
  running times of operations on(操作的运行时间)，456 fig.
  and treaps(与 treaps)，296 pr.
  2-3-4，473 pr.
HEAP-DECREASE-KEY，141 ex.
HEAP-DELETE，142 ex.
HEAP-EXTRACT-MAX，139
HEAP-EXTRACT-MIN，141 ex.
HEAP-INCREASE-KEY，140
HEAP-MAXIMUM，139
HEAP-MINIMUM，141 ex.
heap property(堆的性质)，128
  maintenance of(维护)，130-132
  vs. binary-search-tree property(与二叉搜索树的性质)，256 ex.
heapsort(堆排序)，127-144
HEAPSORT，136
height(高度)
  of a binomial tree(二项树)，457
  black(黑)，274
  of a B-tree(B 树)，439-440
  of a *d*-ary heap(*d* 叉堆)，143 pr.
  of a decision tree(决策树)，167
  exponential(指数)，265
  of a heap(堆)，129，129 ex.
  of a node in a heap(堆中结点)，129，135 ex.
  of a node in a tree(树中结点)，1088
  of a red-black tree(红黑树)，274
  of a tree(树)，1088
height-balanced tree(高度平衡树)，296 pr.
height function(高度函数)，in push-relabel algorithms (push-relabel 算法)，671
hereditary family of subsets(子集的遗传族)，393
Hermitian matrix(埃尔米特矩阵)，759 ex.
high endpoint of an interval(区间的高端点)，311
HIRE-ASSISTANT，92
hiring problem(雇佣问题)，91-92
  on-line(联机)，114-117
  probabilistic analysis of(概率分析)，97-98
hit(命中)，spurious(假的)，912
HOARE-PARTITION，160 pr.
HOPCROFT-KARP，696 pr.
Hopcroft-Karp bipartite matching algorithm (Hopcroft-Karp 二分匹配算法)，696 pr.
horizontal ray(水平射线)，940 ex.
Horner's rule(霍纳法则)，39 pr.，824
  in the Rabin-Karp algorithm (Rabin-Karp 算法)，911
HUFFMAN，388
Huffman code(赫夫曼编码)，385-393
hull(包)，convex(凸)，947-957，964 pr.
hyperedge(超边)，1083

hypergraph(超图)，1083
  and bipartite graphs(与二分图)，1084 ex.

I

idempotency laws for sets(集合的幂等律)，1071
identity(单位元)，862
identity matrix(单位矩阵)，727
if(if 子句)，in pseudocode(伪码)，19
image(像)，1078
implicit summation notation(隐式的求和记号)，648
inadmissible edge(不容许的边)，681
incidence(人射)，1080
incidence matrix(关联矩阵)
  and difference constraints(与差分约束)，603
  of a directed graph(有向图)，403 pr.，531 ex.
  of an undirected graph(无向图)，403 pr.
inclusion and exclusion(容斥)，1074 ex.
INCREASE-KEY，138
increasing a key in a max-heap(在最大堆中对一个关键字增值)，139-140
INCREMENT，408
incremental design method(增量设计方法)，27
  for finding the convex hull(寻找凸包)，948
in-degree(入度)，1081
indentation in pseudocode(伪码中的缩进)，19
independence(独立)
  of events(事件)，1103，1106 ex.
  of random variables(随机变量)，1107
  of subproblems in dynamic programming(动态规划中子问题)，343-344
independent family of subsets(子集的独立族)，393
independent set(独立集)，1018 pr.
  of tasks(任务)，400
index of an element of Z_n^* (Z_n^* 中元素的下标)，877
indicator random variable(指示器随机变量)，94-98
  in analysis of expected height of a randomly built binary search tree(随机构造的二叉搜索树的期望高度的分析中)，265-267
  in analysis of inserting into a treap(向 treap 插入的分析中)，296 pr.
  in analysis of streaks(在序列的分析中)，113-114
  in analysis of the birthday paradox(在生日悖论的分析中)，108-109
  in approximation algorithm for MAX-3-CNF satisfiability(在 MAX-3-CNF 可满足性的近似算法中)，1040
  in bounding the right tail of the binomial distribution(二项分布的右尾定界)，1122-1123
  in bucket sort analysis(在桶排序分析中)，175-177
  in hashing analysis(散列分析)，227-228
  in hiring-problem analysis(雇佣问题分析)，97-98
  in quicksort analysis(快速排序分析)，157-158，160 pr.
  in randomized selection analysis(随机选择分析)，187-189，189 ex.
  in universal-hashing analysis(全域散列分析)，233-234
induced subgraph(导出子图)，1082
inequality(不等式)，linear(线性的)，772
inequality constraint(不等式约束)，778
  and equality constraints(与等式约束)，780
infeasible linear program(不可行的线性规划)，778
infeasible solution(不可行解)，778
infinite sequence(无穷序列)，1078
infinite set(无穷集合)，1073
infinite sum(无穷和)，1058
infinity(无穷大)，arithmetic with(算术)，587
INITIALIZE-PREFLOW，673
INITIALIZE-SIMPLEX，796，812
INITIALIZE-SINGLE-SOURCE，585
injective function(单射函数)，1079
inner product(内积)，730
inorder tree walk(中序树遍历)，254，260 ex.，305
INORDER-TREE-WALK，255
input(输入)
  to an algorithm(算法)，5
  to a combinational circuit(组合电路)，988
  distribution of(分布)，93，99
  to a logic gate(逻辑门)，987
  size of(规模)，23
input alphabet(输入字母表)，916
input sequence(输入序列)，705
input wire(输入线)，705
INSERT，138，198，416 ex.，455
insertion(插入)
  into binary search trees(二叉搜索树)，261
  into binomial heaps(二项堆)，468
  into B-trees(B 树)，443-447
  into chained hash tables(链接散列表)，226

into d-ary heaps(d 叉堆)，143 pr.
into direct-address tables(直接地址表)，222
into dynamic tables(动态表)，418
into Fibonacci heaps(斐波那契堆)，480-481
into heaps(堆)，140
into interval trees(区间树)，313
into linked lists(链表)，205-206
into open-address hash tables(至开放式寻址散列表)，237-238
into order-statistic trees(顺序统计树)，306-307
into queues(队列)，201
into red-black trees(红黑树)，280-287
into stacks(栈)，200
into sweep-line statuses(扫除线状态)，942
into 2-3-4 heaps(2-3-4 堆)，473 pr.
into Young tableaus(Young 氏矩阵)，143 pr.
insertion sort(插入排序)，11，15-19，24-25
in bucket sort(桶排序)，174-177
compared to merge sort(与合并排序比较)，13 ex.
compared to quicksort(与快速排序比较)，153 ex.
decision tree for(决策树)，166 fig.
in merge sort(合并排序)，37 pr.
in quicksort(快速排序)，159 ex.
sorting-network implementation of(排序网络实现)，708 ex.
using binary search(用二叉搜索)，37 ex.
INSERTION-SORT，17，24
instance(实例)
of an abstract problem(抽象问题)，969，972
of a problem(问题)，5
instructions of the RAM model(RAM 模型的指令)，21
integer data type(整数数据类型)，22
integer linear-programming problem(整数线性规划问题)，777，819 pr.，1017 ex.
integers(整数) (Z)，1070
integer-valued flow(整数值的流)，666
integral(积分)，to approximate summations(以近似和)，1067
integrality theorem(完整性定理)，667
integration of series(级数的积分)，1060
interior of a polygon(多边形的内部)，939 ex.
interior point method(内点法)，776
intermediate vertex(中间顶点)，629
internal node(内结点)，1088
internal path length(内部路径长度)，1091 ex.
interpolation by a cubic spline(用三次样条插值)，767 pr.
interpolation by a polynomial(用多项式插值)，825，830 ex.
at complex roots of unity(在单位复根处)，836-837
intersection(交叉点)
of chords(弦)，308 ex.
determining(确定)，for a set of line segments(一系列线段)，940-947
determining(确定)，for two line segments(两条线段)，936-938
of languages(语言)，976
of sets(集合)(∩)，1071
interval(区间)，311
fuzzy sorting of(模糊排序)，163 pr.
INTERVAL-DELETE，312
interval-graph coloring problem(区间图着色问题)，379 ex.
INTERVAL-INSERT，312
INTERVAL-SEARCH，312，314
INTERVAL-SEARCH-EXACTLY，317 ex.
interval tree(区间树)，311-317
interval trichotomy(区间三分法)，311
intractability(棘手)，966
invalid shift(非法移位)，906
inverse(逆元)
of a bijective function(双射函数)，1079
in a group(群)，862
of a matrix(矩阵)，730，733 ex.
of a matrix from an LUP decomposition(来自 LUP 分解的矩阵)，755-756
multiplicative(乘法)，modulo n(模 n)，871
inversion in a sequence(序列中的倒置)，39 pr.，99 ex.
inverter(逆变器)，987
invertible matrix(可逆矩阵)，730
isolated vertex(孤立顶点)，1081
isomorphic graphs(同构图)，1082
iterated function(多重函数)，60 pr.
iterated logarithm function(多重对数函数)，55-56
ITERATIVE-FFT，841
ITERATIVE-TREE-SEARCH，257

J

Jarvis's march(Jarvis 步进法)，955

Jensen's inequality(Jensen 不等式)，1109
JOHNSON，639
Johnson's algorithm(Johnson 算法)，636-640
joining(连接)
  of red-black trees(红黑树)，295 pr.
  of 2-3-4 trees(2-3-4 树)，453 pr.
joint probability density function(联合概率密度函数)，1107
Josephus permutation(Josephus 排列)，318 pr.

K

Karmarkar's algorithm(Karmarkar 算法)，777，820
Karp's minimum mean-weight cycle algorithm(Karp 最小平均权回路算法)，617 pr.
k-ary tree(k 叉树)，1090
k-CNF，967
k-coloring(k 着色)，1019 pr.，1091 pr.
k-combination(k 组合)，1096
k-conjunctive normal form(k 合取范式)，967
kernel of a polygon(多边形的内核)，956 ex.
key(关键字)，15，123，138，197
  median(中位数)，of a B-tree node(B 树结点)，443
  public(公开的)，881，884
  secret(秘密的)，881，884
  static(静态的)，245
Kleene star(Kleene 星)(*)，976
KMP algorithm(KMP 算法)，923-931
KMP-MATCHER，926
knapsack problem(背包问题)
  fractional(部分的)，382，384 ex.
  0-1，382，384 ex.
k-neighbor tree(k 邻居树)，301
knot(结)，of a spline(样条)，767 pr.
Knuth-Morris-Pratt algorithm(Knuth-Morris-Pratt 算法)，923-931
k-permutation(k 排列)，1095
Kraft inequality(Kraft 不等式)，1091 ex.
Kruskal's algorithm(Kruskal 算法)，568-570
  with integer edge weights(带整数边权值)，574 ex.
k-sorted(k 有序的)，180 pr.
k-string(k 串)，1095
k-subset(k 子集)，1073
k-substring(k 子串)，1095
kth power(k 次幂)，855 ex.
k-universal hashing(k 全域散列)，251 pr.

L

Lagrange's formula(拉格朗日公式)，826
Lagrange's theorem(拉格朗日定理)，866
Lam's theorem(Lam 定理)，859
language(语言)，975
  completeness of(完备性)，994 ex.
  proving NP-completeness of(NP 完全性证明)，995-996
  verification of(验证)，980
last-in(后进)，first-out(先出)，200
  参见 stack
late task(迟任务)，399
layers(层)
  convex(凸的)，962 pr.
  maximal(最大的)，962 pr.
LCA，521 pr.
lcm (least common multiple)(最小公倍数)，861 ex.
LCS，见 longest common subsequence
LCS-LENGTH，353
leading submatrix(主子式)，760
leaf(叶子)，1088
least common ancestor(最小公共祖先)，521 pr.
least common multiple(最小公倍数)，861 ex.
least-squares approximation(最小二乘逼近)，762-765
leaving a vertex(离开一个顶点)，1080
leaving variable(换出变量)，793
LEFT，128
left child(左孩子)，1089
left-child(左孩子)，right-sibling representation(右兄弟表示法)，214，217 ex.
LEFT-ROTATE，278，316 ex.
left rotation(左旋)，277
left spine(左脊)，296 pr.
left subtree(左子树)，1089
Legendre symbol $\frac{a}{p}$(勒让德符号)，903 pr.
length(长度)
  of a path(路径)，1081
  of a sequence(序列)，1078
  of a spine(脊)，296 pr.
  of a string(串)，907，1095
level of a function(函数的阶)，510

lexicographically less than(字典序小于)，269 pr.
lexicographic sorting(字典排序)，269 pr.
lg (binary logarithm)(2 为底的对数)，53
lg* (iterated logarithm function)(多对数函数)，55-56
lg^k (exponentiation of logarithms)(对数的指数)，53
lg lg (composition of logarithms)(对数复合)，53
LIFO (last-in, first-out)(后进，先出)，200
  参见 stack
light edge(轻边)，563
linear constraint(线性约束)，772
linear dependence(线性相关)，731
linear equality(线性等式)，772
linear equations(线性方程)
  solving modular(求解模)，869-872
  solving systems of(组的求解)，742-755
  solving tridiagonal systems of(三对角方程组求解)，767 pr.
linear function(线性函数)，25，772
linear independence(线性独立)，731
linear inequality(线性不等式)，772
linear-inequality feasibility problem(线性不等式可行性问题)，818 pr.
linearity of expectation(期望的线性性)，1108
linearity of summations(和的线性性)，1059
linear order(线性序)，1077
linear probing(线性探测)，239
linear programming(线性规划)，770-821
  algorithms for(算法)，776-777
  applications of(应用)，776
  duality in(对偶性)，804-811
  finding an initial solution for(寻找一个初始解)，811-816
  fundamental theorem of(基本定理)，816
  interior-point methods for(内点法)，776，820
  Karmarkar's algorithm for (Karmarkar 算法)，777、820
  and maximum flow(与最大流)，786
  and minimum-cost flow(与最小费用流)，787-788
  and minimum-cost multicommodity flow(与最小费用多商品流)，790 ex.
  and multicommodity flow(与多商品流)，788 789
  simplex algorithm for(单纯形法)，790-804
  and single-pair shortest path(与单对最短路径)，785-786
  and single-source shortest paths(与单源最短路径)，601-607
  slack form for(松弛型)，781-783
  standard form for(标准型)，777-781
  参见 integer linear-programming problem，0-1 integer-programming problem
linear-programming relaxation(线性规划松弛)，1041
linear search(线性搜索)，21 ex.
line segment(线段)，934
  determining turn of(确定它的转向)，936
  determining whether any intersect(确定任意线段是否相交)，940-947
  determining whether two intersect(确定两条线段是否相交)，936-938
link(链接)
  of binomial trees(二项树)，457
  of Fibonacci-heap roots(斐波那契堆的根)，483
LINK，508
linked list(链表)，204-209
  compact(紧凑的)，213 ex.，218 pr.
  deletion from(删除)，206
  to implement disjoint sets(实现不相交集合)，501-505
  insertion into(插入)，205-206
  neighbor list(邻居列表)，683
  searching(搜索)，205，236 ex.
list(列表)，见 linked list
LIST-DELETE，206
LIST-DELETE'，206
LIST-INSERT，206
LIST-INSERT'，207
LIST-SEARCH，205
LIST-SEARCH'，207
literal(文字的)，999
little-oh notation(小 o 记号)，47-48
little-omega notation(大 Ω 记号)，48
L_∞-distance(L_∞ 距离)，962 ex.
ln (natural logarithm)(自然对数)，53
load factor(装载因子)
  of a dynamic table(动态表)，417
  of a hash table(散列表)，226
load instruction(装载指令)，22
local variable(局部变量)，19
logarithm function(对数函数) (log)，53-54
  discrete(离散的)，877

iterated(多重的)($\lg^*$), 55-56
logic gate(逻辑门), 987
longest common subsequence(最长公共子序列), 350-356, 369
LONGEST-PATH, 978 ex.
LONGEST-PATH-LENGTH, 978 ex.
longest-simple-cycle problem(最长简单环问题), 1017 ex.
longest simple path(最常简单路径), 966
  in an unweighted graph(在一个无权图中), 342
LOOKUP-CHAIN, 348
looping constructs in pseudocode(在伪码中的循环结构), 19
loop invariant(循环不变式), 17
  for breadth-first search(广度优先搜索), 534
  for building a heap(构造堆), 133
  for consolidating the root list in extracting the minimum node from a Fibonacci heap(从斐波那契堆中抽取最小结点时合并根列表), 486
  for determining the rank of an element in an order-statistic tree(确定顺序统计树中一个元素的秩), 305
  for Dijkstra's algorithm(Dijkstra 算法), 597
  for the generic minimum-spanning-tree algorithm(用于一般的最小生成树算法), 562
  for the generic push-relabel algorithm(用于一般的 push-relabel 算法), 676
  for heapsort(堆排序), 136 ex.
  for Horner's rule(霍纳法则), 39 pr.
  for increasing a key in a heap(增大堆中一个关键字增值), 142 ex.
  initialization of(初始化), 18
  for insertion sort(插入排序), 17-19
  and for loops(与 for 循环), 18 n.
  maintenance of(维护), 18
  for merging(合并), 30
  for modular exponentiation(按模取幂), 879-880
  origin of(起源), 40
  for partitioning(划分), 146
  for Prim's algorithm(Prim 算法), 572
  for the Rabin-Karp algorithm(Rabin-Karp 算法), 914
  for randomly permuting an array(随机排列一个数组), 103
  for red-black tree insertion(红黑树插入), 283
  for the relabel-to-front algorithm(relabel-to-front 算法), 687
  for searching an interval tree(搜索区间树), 315
  for simplex algorithm(单纯形算法), 798
  for string-matching automata(串匹配自动机), 919, 921
  and termination(与终止), 18
  for uniting binomial heaps(合并二项堆), 472 ex.
low endpoint of an interval(区间的低端点), 311
lower bounds(下界)
  for average sorting(平均排序), 180 pr.
  for convex hull(凸包), 956 ex.
  for median finding(中位数寻找), 195
  for merging(合并), 180 pr.
  for minimum-weight vertex cover(最小权顶点覆盖), 1042
  and potential functions(与势函数), 429
  for size of a merging network(合并网络的大小), 718 ex.
  for size of optimal vertex cover(最优顶点覆盖的大小最), 1026
  for sorting(排序), 165-168
lower median(低中位数), 183
lower-triangular matrix(下三角矩阵), 728
LU decomposition(LU 分解), 747-750
LU-DECOMPOSITION, 749
LUP decomposition(LUP 分解), 743
  computation of(计算), 750-754
  of a diagonal matrix(对角矩阵), 754 ex.
  in matrix inversion(矩阵求逆), 755-756
  and matrix multiplication(与矩阵乘法), 759 ex.
  of a permutation matrix(置换矩阵), 754 ex.
  use of(用途), 743-747
LUP-DECOMPOSITION, 752
LUP-SOLVE, 745

M

main memory(主存), 434
MAKE-BINOMIAL-HEAP, 461
MAKE-HEAP, 455
MAKE-SET, 498
  disjoint-set-forest implementation of(不相交集合森林实现), 508
  linked-list implementation of(链表实现), 501
MAKE-TREE, 519 pr.

Manhattan distance（曼哈顿距离），194 pr.，962 ex.
marked node(带标记的结点)，478，490
Markov's inequality(马尔可夫不等式)，1111 ex.
master method for solving a recurrence(解递归式的主方法)，73-76
master theorem(主定理)，73
  proof of(证明)，76-84
matched vertex(匹配的顶点)，664
matching(匹配)
  maximal(最大)，1026，1051 pr.
  maximum(最小)，1051 pr.
  and maximum flow(与最大流)，664-669
  of strings(串)，906-932
  weighted bipartite(加权二分)，497
matric matroid(矩阵拟阵)，393
matrix(矩阵)，725-734
  adjacency(邻接)，529
  conjugate transpose of(共轭转置)，759 ex.
  Hermitian(埃尔米特)，759 ex.
  incidence(关联)，403 pr.，531 ex.
  predecessor(前驱)，621
  pseudoinverse of(伪逆)，764
  sorting the entries of(元素排序)，721 ex.
  symmetric positive-definite(对称正定)，760-762
  Toeplitz，844 pr.
  transpose of(转置)，529，726
  参见 matrix inversion，matrix multiplication
matrix-chain multiplication(矩阵链乘法)，331-339
MATRIX-CHAIN-MULTIPLY
MATRIX-CHAIN-ORDER，336
matrix inversion(矩阵求逆)，755-758
matrix multiplication(矩阵乘法)
  for all-pairs shortest paths(每对顶点间最短路径)，622-629
  boolean(布尔)，759 ex.
  and computing the determinant(与计算行列式)，759 ex.
  and LUP decomposition(与 LUP 分解)，759 ex.
  and matrix inversion(与矩阵求逆)，756-758
  Pan's method for(Pan 方法)，741 ex.
  Strassen's algorithm for(Strassen 算法)，735-742
MATRIX-MULTIPLY，332，625
matroid(拟阵)，393-399，403 pr.，579
MAX-CNF satisfiability（MAX-CNF 可满足性），1043 ex.
MAX-CUT problem(MAX-CUT 问题)，1043 ex.
MAX-FLOW-BY-SCALING，694 pr.
max-flow min-cut theorem（最大流最小切问题)，657
max-heap(最大堆)，128
  building(构造)，132-135
  deletion from(删除)，142 ex.
  extracting the maximum key from(抽取最大关键字)，139
  in heapsort(堆排序)，135-138
  increasing a key in(增大一个关键字)，139-140
  insertion into(插入)，140
  maximum key of(最大关键字)，139
  as a max-priority queue(作为一个最大优先级队列)，138-142
  mergeable(可合并的)，见 mergeable max-heap
MAX-HEAPIFY，130
MAX-HEAP-INSERT，140
  building a heap with(构造堆)，142 pr.
max-heap property(最大堆的性质)，128
  maintenance of(维护)，130-132
maximal element of a partially ordered set(偏序集的最大元)，1076
maximal layers(最大层)，962 pr.
maximal matching(最大匹配)，1026，1051 pr.
maximal point(最大点)，962 pr.
maximal subset in a matroid（拟阵中的最大子集)，394
maximization linear program（最大化线性规划)，773
  and minimization linear programs(与最小化线性规划)，779
maximum(最大值)，183
  in binary search trees(二叉搜索树)，258
  of a binomial distribution(二项分布)，1117 ex.
  finding(寻找)，184-185
  in heaps(堆)，139
  in order-statistic trees(顺序统计树)，310 ex.
  in red-black trees(红黑树)，276
MAXIMUM，138-139，198
maximum bipartite matching(最大二分匹配)，664-669，680 ex.
  Hopcroft-Karp algorithm for（Hopcroft-Karp 算法)，696 pr.

maximum degree in a Fibonacci heap(斐波那契堆中的最大度数)，479，488 ex.，493-496
maximum flow(最大流)，643-698
  Edmonds-Karp algorithm for(Edmonds-Karp 算法)，660-663
  Ford-Fulkerson method for(Ford-Fulkerson 方法)，651-664
  as a linear program(作为一个线性规划)，786
  and maximum bipartite matching(与最大二分匹配)，664-669
  with negative capacities(带有负容量)，695 pr.
  push-relabel algorithms for(先流推进算法)，669-692
  relabel-to-front algorithm for(relabel-to-front 算法)，681-692
  scaling algorithm for(定标算法)，694 pr.
  updating(更新)，694 pr.
maximum matching(最大匹配)，1051 pr.
max-priority queue(最大优先级队列)，138
MAX-3-CNF satisfiability(MAX-3-CNF 可满足性)，1039-1040
MAYBE-MST-A，578 pr.
MAYBE-MST-B，578 pr.
MAYBE-MST-C，578 pr.
mean(均值)，见 expected value
mean weight of a cycle(环的平均权)，617 pr.
median(中位数)，183-195
  of sorted lists(排序表)，193 ex.
  weighted(带权的)，194 pr.
median key of a B-tree node(B 树结点的中位数关键字)，443
median-of-3 method(3 数取中方法)，162 pr.
member of a set(集合的成员)(∈)，1070
memoization(记忆化)，347-349
MEMOIZED-MATRIX-CHAIN，348
memory(存储器)，434
memory hierarchy(存储器分层体系)，22
merge(合并)
  of k sorted lists(k 个排序表)，142 ex.
  lower bounds for(下界)，180 pr.
  of two sorted arrays(两个排序数组)，28
  using a comparison network(用比较网络)，716-718
MERGE，29
mergeable heap(可合并堆)，431，455
  and comparison sorts(与比较排序)，489 ex.
  linked-list implementation of(链表实现)，217 pr.
  relaxed heaps(松弛堆)，497
  running times of operations on(操作的运行时间)，456 fig.
  2-3-4 heaps(2-3-4 堆)，473 pr.
  参见 binomial heap，Fibonacci heap
mergeable max-heap(可合并的最大堆)，217 n.，431 n.，455 n.
mergeable min-heap(可合并的最小堆)，217 n.，431 n.，455
MERGE-LISTS，1044
MERGER，717
merge sort(合并排序)，11，28-36
  compared to insertion sort(与插入排序比较)，13 ex.
  sorting network implementation of(排序网络实现)，719-721
  use of insertion sort in(用插入排序)，37 pr.
MERGE-SORT，32
  recursion tree for(递归树)，349 ex.
merging network(合并网络)，716-718
  odd-even(奇偶)，721 pr.
MILLER-RABIN，892
Miller Rabin primality test(Miller-Rabin 素数测试)，890-896
MIN-GAP，317 ex.
min-heap(最小堆)，129
  analyzed by potential method(用势方法分析)，416 ex.
  building(构造)，132-135
  in Dijkstra's algorithm(Dijkstra 算法)，599
  in Huffman's algorithm(赫夫曼算法)，388
  in Johnson's algorithm(Johnson 算法)，640
  mergeable(可合并的)，见 mergeable min-heap
  as a min-priority queue，141 ex.
  in Prim's algorithm(Prim 算法)，573
MIN-HEAPIFY，132 ex.
MIN-HEAP-INSERT，141 ex.
min-heap-ordered(最小堆顺序的)，459
min-heap property(最小堆性质)，129，459
  maintenance of(维护)，132 ex.
  vs. binary-search-tree property(与二叉搜索树性质)，256 ex.
minimization linear program(最小化线性规划)，773
  and maximization linear programs(与最大化线性

规划)，779
minimum(最小值)，183
in binary search trees(二叉搜索树)，258
in binomial heaps(二项堆)，462
in B-trees(B树)，447 ex.
in Fibonacci heaps(斐波那契堆)，481
finding(寻找)，184-185
off-line(脱机)，518 pr.
in order-statistic trees(顺序统计树)，310 ex.
in red-black trees(红黑树)，276
MINIMUM，138，184，198，455
minimum-cost flow(最小代价流)，787-788
minimum-cost multicommodity flow(最小代价多商品流)，790 ex.
minimum-cost spanning tree(最小代价生成树)，见 minimum spanning tree
minimum cut(最小割)，655
minimum degree of a B-tree(B树的最小度数)，439
minimum key in 2-3-4 heaps(2-3-4 堆中的最小关键字)，473 pr.
minimum mean-weight cycle(最小平均权环)，617 pr.
minimum node of a Fibonacci heap(斐波那契堆的最小结点)，478
minimum path cover(最小路径覆盖)，692 pr.
minimum spanning tree(最小生成树)，561-579
in approximation algorithm for traveling-salesman problem(旅行商问题的近似算法)，1028
Borůvka's algorithm for(Borůvka 算法)，578
constructed using binomial heaps(用二项堆构造)，474 pr.
on dynamic graphs(动态图)，574 ex.
generic algorithm for(一般算法)，562-567
Kruskal's algorithm for(Kruskal 算法)，568-570
Prim's algorithm for(Prim 算法)，570-573
relation to matroids(与拟阵的关系)，393，395
second-best(次优)，575 pr.
minimum-weight spanning tree(最小权生成树)，见 minimum spanning tree
minimum-weight vertex cover(最小权顶点覆盖)，1040-1043
minor of a matrix(矩阵的子式)，732
min-priority queue(最小优先级队列)，138，141 ex.
in constructing Huffman codes(构造赫夫曼编码)，387
in Dijkstra's algorithm(Dijkstra 算法)，598
in Prim's algorithm(Prim 算法)，572-573
mirroring(镜像)，833 n.
missing child(缺孩子)，1089
mod(模)，51，851
modular arithmetic(模运算)，51-52，862 869
modular exponentiation(按模取幂)，879
MODULAR-EXPONENTIATION，879
modular linear equations(模线性方程)，869-872
MODULAR-LINEAR-EQUATION-SOLVER，871
modulo(模)，51，851
Monge array(Monge 数组)，88 pr.
monotone sequence(单调序列)，144
monotonically decreasing(单调递减)，51
monotonically increasing(单调递增)，51
MST，474 pr.
MST-KRUSKAL，569
MST-PRIM，572
MST-REDUCE，576 pr.
multicommodity flow(多商品流)，788-789
minimum-cost(最小代价)，790 ex.
multidimensional Fast Fourier Transform(多维快速傅里叶变换)，845 pr.
multigraph(多重图)，1083
converting to equivalent undirected graph(转换成等价的无向图)，530 ex.
multiple(倍数)，729，850
of an element modulo n(元素模 n)，869-872
least common(最小公共)，861 ex.
multiple assignment(多重赋值)，19
multiple sources and sinks(多个源和汇)，647
multiplication(乘法)
of complex numbers(复数)，741 ex.
divide-and-conquer method for(分治法)，844 pr.
of matrices(矩阵)，729，734 ex.
of a matrix chain(矩阵链)，见 matrix-chain multiplication
modulo n(模 n) ($\cdot_n$)，863
of polynomials(多项式)，823
multiplication method(乘法方法)，231-232
multiplicative group modulo n(模 n 的乘法群)，864
multiplicative inverse(乘法逆元)，modulo n(模 n)，871
multiply instruction(乘法指令)，22
MULTIPOP，406

MULTIPUSH，409 ex.
multiset(多重集合)，1070 n.
mutually exclusive events(互斥事件)，1100
mutually independent events(相互独立事件)，1103

N

N (set of natural numbers)(自然数的集合)，1070
naive algorithm for string matching(串匹配的朴素算法)，909-911
NAIVE-STRING-MATCHER，909
natural cubic spline(自然三次样条)，767 pr.
natural numbers(自然数)(N)，1070
negative of a matrix(矩阵的负)，729
negative-weight cycle(负权的环)
  and difference constraints(与差分约束)，603
  and relaxation(与松弛)，613 ex.
  and shortest paths(与最短路径)，582
negative-weight edges(负权的边)，582-583
neighbor(邻居)，1083
neighborhood(邻近)，669 ex.
neighbor list(邻居列表)，683
nesting boxes(嵌套的盒子)，615 pr.
net flow across a cut(穿过割的净流量)，655
network(网络)
  admissible(容许的)，681-683
  bitonic sorting(双调排序)，712-716
  comparison(比较)，704-709
  flow(流)，见 flow network
  for merging(合并)，716-718
  odd-even merging(奇偶合并)，721 pr.
  odd-even sorting(奇偶排序)，721 pr.
  permutation(置换)，722 pr.
  residual(残留)，651-653
  sorting(排序)，704-724
  transposition(转置)，721 pr.
NEXT-TO-TOP，949
NIL，20
node(结点)，1087
  参见 vertex
nonbasic variable(非基本变量)，782
nondeterministic polynomial time(非确定多项式时间)，981 n.
  参见 NP
nonhamiltonian graph(非哈密顿图)，979
noninstance(非实例)，974 n.
noninvertible matrix(不可逆矩阵)，730
nonnegativity constraint(非负性约束)，777，779
nonoverlappable string pattern(不可重叠的串模式)，922 ex.
nonsaturating push(非饱和的推)，672，678
nonsingular matrix(非奇异矩阵)，730
nontrivial power(非平凡幂)，855 ex.
nontrivial square root of 1(1 的非平凡平方根)，modulo n(模 n)，878
no-path property(无路径性质)，587，608-609
normal equation(正态方程)，764
norm of a vector(向量的范数)，730
NOT function(非函数)(¬)，987
NOT gate(非门)，987
NP (complexity class)(复杂类)，967，981，983 ex.
NPC (complexity class)(复杂性类)，968，986
NP-complete(NP-完全的)，968，986
NP-completeness(NP-完全)，966-1021
  of the circuit-satisfiability problem(电路可满足性问题)，987-994
  of the clique problem(团问题)，1003-1006
  of determining whether a boolean formula is a tautology(确定一个布尔公式是否为重言式)，1002 ex.
  of the formula-satisfiability problem(公式可满足性问题)，996-998
  of the graph-coloring problem(图着色问题)，1019 pr.
  of the half 3-CNF satisfiability problem(半 3-CNF 可满足性问题)，1018 ex.
  of the hamiltonian-cycle problem(哈密顿环问题)，1008-1012
  of the hamiltonian-path problem(哈密顿路径问题)，1017 ex.
  of the independent-set problem(独立集问题)，1018 pr.
  of the integer linear-programming problem(整数线性规划问题)，1017 ex.
  of the longest-simple-cycle problem(最长简单环问题)，1017 ex.
  proving(证明)，of a language(语言)，995-996
  of scheduling with profits and deadlines(带有利润和完成期限的调度)，1020 pr.
  of the set-covering problem(集合覆盖问题)，1038 ex.

of the set-partition problem(集合划分问题), 1017 ex.
of the subgraph-isomorphism problem(子图同构问题), 1017 ex.
of the subset-sum problem(子集和问题), 1013-1017
of the 3-CNF-satisfiability problem(3-CNF-可满足性问题), 998-1002
of the traveling-salesman problem(旅行商问题), 1012-1013
of the vertex-cover problem(顶点覆盖问题), 1006-1008
of the 0-1 integer-programming problem(0-1 整数规划问题), 1017 ex.
NP-hard(NP 难度), 986
n-set(*n* 集), 1073
n-tuple(*n* 元组), 1074
null event(零事件), 1100
null tree(空树), 1089
null vector(空向量), 731
number-field sieve(数域的筛选), 905
numerical stability(数值稳定性), 725, 743, 769

O

o-notation(*o* 记号), 47-48
O-notation(*O* 记号), 43 fig., 44-45
O'-notation(*O'* 记号), 59 pr.
$\tilde{O}$-notation($\tilde{O}$ 记号), 59 pr.
Object(对象), 20
allocation and freeing of(分配与释放), 210-212
array implementation of(数组实现), 209-213
passing as parameter(作为参数传递), 20
objective function(目标函数), 601, 606-607 ex., 773, 777
objective value(目标值), 774, 778
optimal(最优的), 778
occurrence of a pattern(模式的出现), 906
odd-even merging network(奇偶合并网络), 721 pr.
odd-even sorting network(奇偶排序网络), 721 pr.
OFF-LINE-MINIMUM, 519 pr.
off-line problem(脱机问题)
least common ancestors(最小公共祖先), 521 pr.
minimum(最小的), 518 pr.
Omega-notation(Ω 记号), 43 fig., 45-46
1-approximation algorithm(1 近似算法), 1023
one-pass method(一遍扫描法), 522
one-to-one correspondence(一一对应), 1079
one-to-one function(一对一函数), 1079
one-way hash function(单路散列函数), 886
on-line convex-hull problem(联机凸包问题), 957 ex.
on-line hiring problem(联机雇佣问题), 114-117
ON-LINE-MAXIMUM, 115
ON-SEGMENT, 937
onto(映成), 1079
open-address hash table(开放寻址散列表), 237-245
double hashing(双散列), 240-241, 244 ex.
linear probing(线性探测), 239
quadratic probing(二次探测), 239-240, 250 pr.
open interval(开区间), 311
optimal binary search tree(最优二叉搜索树), 356-363, 369
OPTIMAL-BST, 361
optimal objective value(最优目标值), 778
optimal solution(最优解), 778
optimal subset of a matroid(拟阵的最优子集), 395
optimal substructure(最优子结构)
of activity selection(活动选择), 371-373
of assembly-line scheduling(装配线调度), 325-327
of binary search trees(二叉搜索树), 359
in dynamic programming(动态规划), 339-344
of the fractional knapsack problem(部分背包问题), 382
in greedy algorithms(贪心算法), 381-382
of Huffman codes(赫夫曼编码), 391
of longest common subsequences(最长公共子序列), 351-352
of matrix-chain multiplication(矩阵链乘法), 333-334
of shortest paths(最短路径), 581-582, 623, 629
of unweighted shortest paths(无权值最短路径), 342
of weighted matroids(加权拟阵), 397
of the 0-1 knapsack problem(0-1 背包问题), 382
optimal vertex cover(最优顶点覆盖), 1024
optimization problem(最优化问题), 323, 968, 972
approximation algorithms for(近似算法), 1022-1054
and decision problems(与决策问题), 969

OR function(或函数)(∨)，633，987
or(或)，in pseudocode(伪码)，20
order(序)
  of a group(群的)，867
  linear(线性)，1077
  partial(偏)，1076
  total(全)，1077
ordered pair(有序对)，1073
ordered tree(有序树)，1088
order of growth(增长的量级)，26
order statistics(顺序统计)，183-195
  dynamic(动态的)，302-308
order-statistic tree(顺序统计树)，302-308
  querying(查询)，310 ex.
OR gate(或门)，987
origin(源)，934
orthonormal(正交的)，769
OS-KEY-RANK，307 ex.
OS-RANK，305
OS-SELECT，304
out-degree(出度)，1081
outer product(外积)，730
output(输出)
  of an algorithm(算法)，5
  of a combinational circuit(组合电路)，988
  of a logic gate(逻辑门)，987
output sequence(输出序列)，705
output wire(输出线)，705
overdetermined system of linear equations(超定线性方程组)，743
overflow(溢出)
  of a queue(队列)，202
  of a stack(栈)，201
overflowing vertex(溢出顶点)，670
overlapping intervals(重叠区间)，311
  finding all(找出所有的)，317 ex.
  point of maximum overlap(最大重叠的点)，318 pr.
overlapping rectangles(重叠矩形)，317 ex.
overlapping subproblems(重叠子问题)，344-346
overlapping-suffix lemma(重叠后缀引理)，908

P

P (complexity class)(复杂性类)，967，973，977，979 ex.
package wrapping(打包)，955
page on a disk(磁盘上的页)，436，452 pr.
paging(取页)，22
pair(对)，ordered(有序的)，1073
pairwise disjoint sets(两两不相交集合)，1073
pairwise independence(两两独立)，1103
pairwise relatively prime(两两互质的数)，854
Pan's method for matrix multiplication(Pan 的矩阵乘法方法)，741 ex.
parallel-machine-scheduling problem(并行机调度问题)，1051 pr.
parameter(参数)，20
  costs of passing(传递的代价)，85 pr.
parent(双亲)，1087
  in a breadth-first tree(广度优先树)，532
PARENT，128
parenthesis structure of depth-first search(深度优先搜索的括号结构)，543
parenthesis theorem(括号定理)，543
parenthesization of a matrix-chain product(矩阵链乘积的括号化)，331
parse tree(语法分析树)，999
partial order(偏序)，1076
PARTITION，146
partitioning algorithm(划分算法)，146-148
  around median of 3 elements(在 3 个元素的中位数附近)，159 ex.
  randomized(随机的)，154
partition of a set(集合的划分)，1073，1076
Pascal's triangle(Pascal 三角)，1099 ex.
path(路径)，1081
  augmenting(增广)，654，696 pr.
  critical(关键)，594
  find(找出)，506
  Hamiltonian(哈密顿)，983 ex.
  longest(最长的)，342，966
  shortest(最短的)，见 shortest paths
  weight of(权)，580
PATH，969，976
path compression(路径压缩)，506
path cover(路径覆盖)，692 pr.
path length(路径长度)，of a tree(树)，270 pr.，1091 ex.
path-relaxation property(路径松弛性质)，587，609-610

pattern in string matching(串匹配中的模式)，906
  nonoverlappable(不重叠的)，922 ex.
pattern matching(模式匹配)，见 string matching
penalty(罚款)，399
perfect hashing(完全散列)，245-249
perfect matching(完全匹配)，669 ex.
permutation(排列)，1079
  bit-reversal(位反向)，425 pr.，841
  Josephus，318 pr.
  in place(就地)，102
  random(随机)，101-104
  of a set(集合)，1095
  uniform random(均匀随机)，93，101
permutation matrix(置换矩阵)，728，733 ex.
  LUP decomposition of(LUP 分解)，754 ex.
permutation network(置换网络)，722 pr.
PERMUTE-BY-CYCLIC，105 ex.
PERMUTE-BY-SORTING，101
PERMUTE-WITH-ALL，105 ex.
PERMUTE-WITHOUT-IDENTITY，104 ex.
persistent data structure(持久的数据结构)，294 pr.，432
PERSISTENT-TREE-INSERT，294 pr.
PERT chart(PERT 表)，594，594 ex.
phi function(phi 函数)，865
PISANO-DELETE，496 pr.
pivot(主元)
  in linear programming(线性规划)，793-796，803 ex.
  in LU decomposition(LU 分解)，749
  in quicksort(快速排序)，146
PIVOT，795
platter(碟)，435
pointer(指针)，20
  array implementation of(数组实现)，209-213
point-value representation(点值表示法)，825
polar angle(极角)，939 ex.
Pollard's rho heuristic(Pollard 的 rho 启发式)，897-901，901 ex.
POLLARD-RHO，897
polygon(多边形)，939 ex.
  kernel of(核)，956 ex.
  star-shaped(星形)，956 ex.
polylogarithmically bounded(多项对数界的)，54
polynomial(多项式)，52，822
  addition of(加法)，822
  asymptotic behavior of(渐近行为)，57 pr.
  coefficient representation of(系数表示法)，824
  evaluation of(求值)，39 pr.，824，829 ex.，845-846 pr.
  interpolation by(插值)，825，830 ex.
  multiplication of(乘法)，823，827-829，844 pr.
  point-value representation of(点值表示法)，825
polynomially bounded(多项式界的)，52
polynomially related(多项式相关的)，974
polynomial-time acceptance(多项式时间接受)，976
polynomial-time algorithm(多项式时间算法)，850，966
polynomial-time approximation scheme(多项式时间近似方案)，1023
polynomial-time computability(多项式时间可计算性)，974
polynomial-time decision(多项式时间决策)，976
polynomial-time reducibility(多项式时间可归约性)(≤P)，984，994 ex.
polynomial-time solvability(多项式时间可求解性)，973
polynomial-time verification(多项式时间验证)，979-983
pop(出弹出)
  from a run time stack(运行时栈)，162 pr.
  stack operation(栈操作)，201
POP，201
positional tree(位置树)，1090
positive-definite matrix(正定矩阵)，732
positive flow(正流)，645
post-office location problem(邮局选址问题)，194 pr.
postorder tree walk(后序树遍历)，254
potential function(势函数)，413
  for lower bounds(下界)，429
potential method(势方法)，412-416
  for binary counters(二进制计数器)，414-415
  for disjoint-set data structures(不相交集合数据结构)，512-517
  for dynamic tables(动态表)，419-420，422-424
  for Fibonacci heaps(斐波那契堆)，479-482，487-488，491-492
  for the generic push-relabel algorithm(一般的压进与重标记算法)，678-679

for the Knuth-Morris-Pratt algorithm (Knuth-Morris-Pratt 算法), 926-927
for min-heaps(最小堆), 416 ex.
for restructuring red-black trees(重构红黑树), 428 pr.
for stack operations(栈操作), 413-414
potential of a data structure(数据结构的势), 413
power(幂)
of an element(元素), modulo n(模 n), 876-881
kth(k 次), 855 ex.
nontrivial(非平凡的), 855 ex.
power series(幂级数), 86 pr.
power set(幂集), 1073
Pr {} (probability distribution)(概率分布), 1100
predecessor(前驱)
in binary search trees(二叉搜索树), 258-259
in breadth-first trees(广度优先树), 532
in B-trees(B 树), 447 ex.
in linked lists(链表), 204
in order-statistic trees(顺序统计量树), 310 ex.
in red-black trees(红黑树), 276
in shortest-paths trees(最短路径树), 584
PREDECESSOR, 198
predecessor matrix(前驱矩阵), 621
predecessor subgraph(前驱子图)
in all-pairs shortest paths(每对顶点间最短路径), 621
in breadth-first search(广度优先搜索), 537
in depth-first search(深度优先搜索), 540
in single-source shortest paths(单源最短路径), 584
predecessor-subgraph property(前趋子图的性质), 587, 612-613
preemption(抢占), 402 pr.
prefix(前缀)
of a sequence(序列的), 351
of a string (匚)(串), 907
prefix code(前缀码), 385
prefix function(前缀函数), 923-925
prefix-function iteration lemma(前缀函数迭代引理), 927
preflow(先流), 669
preorder tree walk(先序树遍历), 254
presorting(预排序), 961
Prim's algorithm(Prim 算法), 570-573
with an adjacency matrix(邻接矩阵), 573 ex.
in approximation algorithm for traveling-salesman problem(旅行商问题的近似算法), 1028
implemented with a Fibonacci heap(用斐波那契堆实现), 573
implemented with a min-heap(用最小堆实现), 573
with integer edge weights(带整数边权值), 574 ex.
similarity to Dijkstra's algorithm(与 Dijkstra 算法的相似之处), 570, 599
for sparse graphs(稀疏图), 575 pr.
primality testing(素数测试), 887-896, 904
Miller-Rabin test(Miller Rabin 测试), 890-896
pseudoprimality testing(伪素数测试), 889-890
primal linear program(原始线性规划), 805
primary clustering(一次群集), 239
primary memory(主存), 434
prime distribution function(素数分布函数), 888
prime number(素数), 851
density of(密度), 887-888
prime number theorem(素数定理), 888
primitive root of $\mathbb{Z}_n^*$ ($\mathbb{Z}_n^*$ 的原根), 877
principal root of unity(单位主根), 831
principle of inclusion and exclusion(容斥原理), 1074 ex.
PRINT-ALL-PAIRS-SHORTEST-PATH, 621
PRINT-INTERSECTING-SEGMENTS, 946 ex.
PRINT-LCS, 355
PRINT-OPTIMAL-PARENS, 338, 338 ex.
PRINT-PATH, 538
PRINT-STATIONS, 330
priority queue(优先级队列), 138-142
in constructing Huffman codes(构造赫夫曼编码), 387
in Dijkstra's algorithm(Dijkstra 算法), 598
heap implementation of(堆实现), 138-142
max-priority queue(最大优先队列), 138
min-priority queue(最小优先队列), 138, 141 ex.
with monotone extractions(带单调抽取), 144
in Prim's algorithm(Prim 算法), 572-573
参见 binary search tree, binomial heap, Fibonacci heap
probabilistic analysis(概率分析), 92-93, 106-117
of approximation algorithm for MAX-3-CNF satisfiability(MAX-3-CNF 可满足性的逼近

算法)，1040
of average-case lower bound for sorting(排序的平均情况下界)，178 pr.
and average inputs(与平均输入)，26
of average node depth in a randomly built binary search tree(随机构造的二叉搜索树中结点的平均深度)，270 pr.
of balls and bins(球与盒子)，109-110
of birthday paradox(生日悖论)，106-109
of bucket sort(桶排序)，174-177，177 ex.
of collisions(碰撞)，228 ex.，249 ex.
of convex hull over a sparse-hulled distribution(稀疏包分布上的凸包)，964 pr.
of file comparison(文件比较)，915 ex.
of hashing with chaining(链接散列)，226-228
of the height of a randomly built binary search tree(随机构造的二叉搜索树的高度的)，265-268
of the hiring problem(雇佣问题的)，97-98
of insertion into a binary search tree with equal keys(把相等关键字插入二叉搜索树)，268 pr.
of longest-probe bound for hashing(散列的最长探测界)，249 pr.
of the Miller-Rabin primality test(Miller-Rabin 素数测试)，893-896
of open-address hashing(开放式寻址散列)，241-244，244 ex.
of partitioning(划分)，153 ex.，159 ex.，160 pr.，162 pr.
of perfect hashing(完全散列)，246-249
of Pollard's rho heuristic(Pollard 的 rho 启发式)，898-901
of probabilistic counting(概率计数)，118 pr.
of quicksort(快速排序)，156-158，160 pr.，162 pr.，268 ex.
of the Rabin-Karp algorithm (Rabin-Karp 算法)，915
and randomized algorithms(随机算法)，99-101
of randomized selection(随机选择)，186-189
of searching a compact list(搜索紧凑表)，218 pr.
of slot-size bound for chaining(链接槽大小的界)，250 pr.
of sorting points by distance from origin(按距源点距离对点排序)，177 ex.
of streaks(序列)，110-114
of universal hashing(全域散列)，233-236
probabilistic counting(概率计数)，118 pr.
probability(概率)，1100-1106
probability density function(概率密度函数)，1107
probability distribution(概率分布)，1100
probability distribution function(概率分布函数)，177 ex.
probe(探测)，237，249 pr.
probe sequence(探测序列)，237
probing(探测)，见 linear probing，quadratic probing
problem(问题)
  abstract(抽象的)，972
  computational(计算的)，5-6
  concrete(具体的)，973
  decision(决策)，969，972
  intractable(难处理的)，966
  optimization(最优化)，323，968，972
  solution to(解)，6，972-973
  tractable(易处理的)，966
procedure(过程)，6，15-16
product(积)
  Cartesian(笛卡儿)，1074
  cross(叉)，934
  inner(内)，730
  of matrices(矩阵)，729，734 ex.
  outer(外)，730
  of polynomials(多项式)，823
  rule of(规则)，1095
  scalar flow(标量流)，650 ex.
professional wrestler(职业摔跤选手)，539 ex.
program counter(程序计数器)，990
programming(规划)，见 dynamic programming，linear programming
proper ancestor(真祖先)，1087
proper descendant(真子孙)，1087
proper subgroup(真子群)，866
proper subset(真子集)(⊂)，1071
prune-and-search method(剪枝-搜索方法)，948
pruning a Fibonacci heap(斐波那契堆剪枝)，497 pr.
pseudocode(伪码)，15，19-20
pseudoinverse(伪逆)，764
pseudoprime(伪素数)，889-890
PSEUDOPRIME，889
pseudorandom-number generator(伪随机数生成器)，94

public key(公钥)，881，884
public-key cryptosystem(公钥加密系统)，881-887
PUSH
  push-relabel operation(push-relabel 操作)，672
  stack operation(栈操作)，201
push onto a run-time stack(压入运行时栈)，162 pr.
push operation(压入操作)(in push-relabel algorithms)(push-relabel 算法)，671-672
push-relabel algorithm(先流推进算法)，669-692
  basic operations in(基本操作)，671-673
  by discharging an overflowing vertex of maximum height(通过释放一个最大高度的溢出顶点)，691 ex.
  to find a maximum bipartite matching(来找出最大二分匹配)，680 ex.
  gap heuristic for(间隔启发式)，691 ex.
  generic algorithm(一般算法)，673-681
  with a queue of overflowing vertices(带有溢出顶点的队列)，691 ex.
  relabel-to-front algorithm(relabel-to-front 算法)，681-692

Q

quadratic function(二次函数)，25
quadratic probing(二次探测)，239-240，250 pr.
quadratic residue(二次余数)，903 pr.
quantile(分位数)，192 ex.
query(查询)，198
queue(队列)，200-203
  in breadth-first search(广度优先搜索)，532
  implemented by stacks(用栈实现)，204 ex.
  linked-list implementation of(链表实现)，208 ex.
  priority(优先级)，见 priority queue
in push-relabel algorithms(先流推进算法)，691 ex.
quicksort(快速排序)，145-164
  analysis of(分析)，149-153，155-159
  average-case analysis of(平均情况分析)，156-158
  compared to insertion sort(与插入排序比较)，153 ex.
  compared to radix sort(与基数排序比较)，173
  description of(描述)，145-149
  good worst-case implementation of(好的最坏情况实现)，192 ex.
  with median-of-3 method(用 3 数取中方法)，162 pr.
  randomized version of(随机化版本)，153-154，160 pr.
  stack depth of(栈深度)，162 pr.
  tail-recursive version of(尾递归版本)，162 pr.
  use of insertion sort in(用插入排序)，159 ex.
  worst-case analysis of(最坏情况分析)，155
QUICKSORT，146
QUICKSORT′，162 pr.
quotient(商)，851

R

R(set of real numbers)(实数集)，1070
Rabin-Karp algorithm(Rabin-Karp 算法)，911-916
RABIN-KARP-MATCHER，914
radix sort(基数排序)，170-173
  compared to quicksort(与快速排序比较)，173
RADIX-SORT，172
radix tree(基数树)，269 pr.
RAM，见 random-access machine RANDOM，94，94 ex.
random-access machine(随机存取机器)，21-22
  vs. comparison networks(与比较网络)，704
randomized algorithm(随机算法)，93-94，99-105
  and average inputs(与平均输入)，26
  comparison sort(比较排序)，178 pr.
  for the hiring problem(雇佣问题)，100
  for insertion into a binary search tree with equal keys(把相等关键字插入二叉搜索树)，268 pr.
  for MAX-3-CNF satisfiability(MAX-3-CNF 可满足性)，1039-1040
  Miller-Rabin primality test(Miller-Rabin 素数测试)，890-896
  for partitioning(划分)，154，159 ex.，160 pr.，162 pr.
  for permuting an array(排列数组)，101-104
  Pollard's rho heuristic(Pollard 的 rho 启发式)，897-901，901 ex.
  and probabilistic analysis(与概率分析)，99-101
  quicksort(快速排序)，153-154，159 ex.，160 pr.，162 pr.
  randomized rounding(随机舍入)，1053
  for searching a compact list(搜索紧凑表)，218 pr.

for selection(选择)，185-189
universal hashing(全域散列)，232-236
worst-case performance of(最坏情况性能)，154 ex.
RANDOMIZED-HIRE-ASSISTANT，100
RANDOMIZED-PARTITION，154
RANDOMIZED-QUICKSORT，154，268 ex.
relation to randomly built binary search trees(与随机构造的二叉搜索树的关系)，270 pr.
randomized rounding(随机舍入)，1053
RANDOMIZED-SELECT，186
RANDOMIZE-IN-PLACE，103
randomly built binary search tree(随机构造的二叉搜索树)，265-268，270 pr.
random-number generator(随机数生成器)，94
random permutation(随机排列)，101-104
uniform(均匀)，93，101
random sampling(随机取样)，154
RANDOM-SEARCH，118 pr.
random variable(随机变量)，1106-1111
indicator(指示器)，见 indicator random variable
range(范围)，1078
rank(秩，排序)
column(列)，731
full(满)，731
of a matrix(矩阵)，731，734 ex.
of a node in a disjoint-set forest(不相交集合森林中结点)，506，511-512，518 ex.
of a number in an ordered set(有序集合中的数)，302
in order-statistic trees(顺序统计树)，304-306，307 ex.
row(行)，731
rate of growth(增长率)，26
ray(射线)，940 ex.
RB-DELETE，288
RB-DELETE-FIXUP，289
RB-ENUMERATE，311 ex.
RB-INSERT，280
RB-INSERT-FIXUP，281
RB-JOIN，295 pr.
reachability in a graph(图中的可达性)($\rightsquigarrow$)，1081
real numbers(实数)(R)，1070
reconstructing an optimal solution in dynamic programming(动态规划中重构最优解)，346-347
record(记录)(data)(数据)，123
rectangle(矩形)，317 ex.
recurrence(递归)，32，62-90
solution by Akra-Bazzi method(用 Akra-Bazzi 方法求解)，89-90
solution by master method(用主方法求解)，73-76
solution by recursion-tree method(用递归树方法求解)，67-72
solution by substitution method(用替换方法求解)，63-67
recurrence equation(递归方程)，见 recurrence
recursion(递归)，28
recursion tree(递归树)，36，67-72
for merge sort(合并排序)，349 ex.
in proof of master theorem(主定理的证明)，76-78
and the substitution method(与替换方法)，70-72
RECURSIVE-ACTIVITY-SELECTOR，376
RECURSIVE-FFT，835
RECURSIVE-MATRIX-CHAIN，345
red-black tree(红黑树)，273-301
augmentation of(扩张)，309-310
compared to B-trees(与 B 树比较)，440
deletion from(删除)，288-294
in determining whether any line segments intersect(确定任意线段是否相交)，943
for enumerating keys in a range(列举一个范围内的关键字)，311 ex.
height of(高度)，274
insertion into(插入)，280-287
joining of(连接)，295 pr.
maximum key of(最大关键字)，276
minimum key of(最小关键字)，276
predecessor in(前趋)，276
properties of(性质)，273-277
relaxed(松弛的)，276 ex.
restructuring(重构)，428 pr.
rotation in(旋转)，277-279
searching in(搜索)，276
successor in(后继)，276
and 2-3-4 trees(与 2-3-4 树)，441 ex.
参见 interval tree，order-statistic tree

reducibility(可归约性)，984-986
reduction algorithm(归约算法)，970，984
reduction function(归约函数)，984
reflexive relation(自反关系)，1075
reflexivity of asymptotic notation(渐近记号的自反性)，49
region(区域)，feasible(可行的)，773
rejection(拒绝)
  by an algorithm(被算法)，976
  by a finite automaton(被自动机)，917
RELABEL，673
relabeled vertex(重新标记的顶点)，673
relabel operation(重新记号)(in push-relabel algorithms)(先流推进算法)，672-673，677
RELABEL-TO-FRONT，687
relabel-to-front algorithm(relabel-to-front 算法)，681-692
relation(关系)，1075-1077
relatively prime(互质)，854
RELAX，586
relaxation(松弛)
  of an edge(边)，585-587
  linear programming(线性规划)，1041
relaxed heap(松弛堆)，497
relaxed red-black tree(松弛红黑树)，276 ex.
release time(释放时间)，402 pr.
remainder(余数)，51，851
remainder instruction(余数指令)，22
repeat(重复)，in pseudocode(伪码)，19
repeated squaring(重复平方)
  for all-pairs shortest paths(所有顶点间最短路径)，625-627
  for raising a number to a power(求数的幂)，879
repetition factor of a string(字符串的重复因子)，931 pr.
REPETITION-MATCHER，931 pr.
representative of a set(集合的代表)，498
RESET，412 ex.
residual capacity(剩余容量)，651，654
residual edge(剩余边)，652
residual network(剩余网络)，651-653
residue(剩余)，51，851，903 pr.
respect a set of edges(不妨碍边的集合)，563
return instruction(返回指令)，22
reweighting(重新赋权)
  in all-pairs shortest paths(每对顶点间最短路径)，636
  in single-source shortest paths(单源最短路径)，615 pr.
rho heuristic(rho 启发式)，897-901，901 ex.
$\rho(n)$-approximation algorithm ($\rho(n)$ 近似算法)，1022
RIGHT，128
right child(右孩子)，1089
right-convert(右变换)，279 ex.
right horizontal ray(右水平射线)，940 ex.
RIGHT-ROTATE，278
right rotation(右旋转)，277
right spine(右脊)，296 pr.
right subtree(右子树)，1089
root(根)
  of a tree(树)，1087
  of unity(单位)，830-831
  of Z_n^* (Z_n^*)，877
rooted tree(有根树)，1087
  representation of(表示法)，214-217
root list(根列表)
  of a binomial heap(二项堆)，459
  of a Fibonacci heap(斐波那契堆)，478
rotation(旋转)
  cyclic(循环)，930 ex.
  in a red-black tree(红黑树)，277-279
rotational sweep(旋转扫除)，947，949-955
rounding(舍入)，1042
  randomized(随机的)，1053
row rank(行秩)，731
row vector(行向量)，726
RSA public-key cryptosystem(RSA 公钥加密系统)，881-887
rule of product(求积准则)，1095
rule of sum(求和准则)，1094
running time(运行时间)，23
  average-case(平均情况)，26
  best-case(最好情况)，27 ex.，46
  of a comparison network(比较网络)，707
  expected(期望)，26
  of a graph algorithm(图算法)，526
  order of growth(增长的量级)，26

rate of growth(增长率)，26
worst-case(最坏情况)，26，46

S

safe edge(安全边)，562
SAME-COMPONENT，500
sample space(样本空间)，1100
sampling(取样)，154
SAT，996
satellite data(附属数据)，123，197
satisfiability(可满足性)，988，996-998，1039-1040，1043 ex.
satisfiable formula(可满足公式)，967，996
satisfying assignment(满足赋值)，988，996
saturated edge(饱和边)，672
saturating push(饱和推)，672，678
scalar flow product(标量流积)，650 ex.
scalar multiple(标量倍数)，729
scaling(定标)
in maximum flow(最大流)，694 pr.
in single-source shortest paths(单源最短路径)，615 pr.
scapegoat tree(scapegoat 树)，301
schedule(调度)，399，1051 pr.
event-point(事件点)，942
scheduling(调度)，369 pr.，402 pr.，1020 pr.，1051 pr.
Schur complement(Schur 补)，748，761
Schur complement lemma(Schur 补引理)，761
SCRAMBLE-SEARCH，118 pr.
SEARCH，198
searching(搜索)
binary search(二分搜索)，37 ex.
in binary search trees(二叉搜索树)，256-258
in B-trees(B 树)，441-442
in chained hash tables(链接散列表)，226
in compact lists(紧凑列表)，218 pr.
in direct-address tables(直接地址表)，222
for an exact interval(正合区间)，317 ex.
in interval trees(区间树)，314-316
linear search(线性搜索)，21 ex.
in linked lists(链表)，205
in open-address hash tables(开放式寻址散列表)，238
problem of(问题)，21 ex.
in red-black trees(红黑树)，276
an unsorted array(无顺序数组)，118 pr.
search tree(搜索树)，见 balanced search tree，binary search tree，B-tree，exponential search tree，interval tree，optimal binary search tree，order-statistic tree，red-black tree，splay tree，2-3 tree，2-3-4 tree
secondary clustering(二次群集)，240
secondary hash table(辅助散列表)，245
secondary storage(辅存)
search tree for(搜索树)，434-454
stacks on(栈)，452 pr.
second-best minimum spanning tree(次优的最小生成树)，575 pr.
secret key(密钥)，881，884
segment(段)，见 directed segment，line segment
SEGMENTS-INTERSECT，937
SELECT，189-190
selection(选择)
of activities(活动)，见 activity-selection problem
and comparison sorts(与比较排序)，192
in expected linear time(期望线性时间)，185-189
in order-statistic trees(顺序统计量树)，303-304
problem of(问题)，183
in worst-case linear time(最坏情况线性时间)，189-193
selection sort(选择排序)，27 ex.
selector vertex(选择顶点)，1009
self-loop(自身环)，1080
semiconnected graph(半连接图)，557 ex.
sentinel(哨兵)，29，206-208，274
sequence(序列)(( ))
bitonic(双调)，618 pr.，712
clean(清洁)，713
finite(有穷)，1078
infinite(无穷)，1078
input(输入)，705
inversion in(逆)，39 pr.，99 ex.
output(输出)，705
probe(探测)，237
series(级数)，86 pr.，1059-1061
set(集合)({})，1070-1075

convex(凸)，650 ex.
independent(独立)，1018 pr.
set-covering problem(集合覆盖问题)，1033-1038
weighted(加权的)，1050 pr.
set-partition problem(集合划分问题)，1017 ex.
shadow of a point(点的影子)，956 ex.
Shell's sort(Shell 排序)，40
shift in string matching(字符串匹配中的移位)，906
shift instruction(移位指令)，22
short-circuiting operator(短路运算符)，20
SHORTEST-PATH，968
shortest paths(最短路径)，580-642
all-pairs(每对顶点间)，581，620-642
Bellman-Ford algorithm for(Bellman-Ford 算法)，588-592
with bitonic paths(双调路径)，618 pr.
and breadth-first search(与广度优先搜索)，534-537，581
convergence property of(收敛性)，587，609
and difference constraints(与差分约束)，601-607
Dijkstra's algorithm for(Dijkstra 算法)，595-601
in a directed acyclic graph(有向无环图)，592-595
in ε-dense graphs(ε 稠密图)，641 pr.
estimate of(估计)，585
Floyd-Warshall algorithm for(Floyd-Warshall 算法)，629-632
Gabow's scaling algorithm for(Gabow 改变尺度算法)，615 pr.
Johnson's algorithm for(Johnson 算法)，636-640
as a linear program(作为线性规划)，785-786
and longest paths(与最长路径)，966
by matrix multiplication(用矩阵乘法)，622-629
and negative-weight cycles(与负权环)，582
with negative-weight edges(负权值边)，582-583
no-path property of(无路径性质)，587，608-609
optimal substructure of(最优子结构)，581-582
path-relaxation property of(路径松弛性质)，587，609-610
predecessor-subgraph property of(前趋子图性质)，587，612-613
problem variants(问题变体)，581
and relaxation(与松弛)，585-587
by repeated squaring(用重复平方)，625-627
single-destination(单目的地)，581
single-pair(单对)，341，581
single-source(单源)，580-619
tree of(树)，584，610-613
triangle inequality of(三角不等式)，587，607-608
in an unweighted graph(无权值图)，341，535
upper-bound property of(上界性质)，587，608
in a weighted graph(加权图)，580
sibling(兄弟)，1088
side of a polygon(多边形的边)，939 ex.
signature(签名)，883
simple cycle(简单环)，1081
simple graph(简单图)，1082
simple path(简单路径)，1081
longest(最长的)，342，966
simple polygon(简单多边形)，939 ex.
simple uniform hashing(简单均匀散列)，226
simplex(单纯形)，775
SIMPLEX，797
simplex algorithm(单纯形法)，775，790-804，820-821
single-destination shortest paths(单目的地最短路径)，581
single-pair shortest path(单对最短路径)，341，581
as a linear program(作为线性规划)，785-786
single-source shortest paths(单源最短路径)，580-619
Bellman-Ford algorithm for(Bellman-Ford 算法)，588-592
with bitonic paths(带有双调路径)，618 pr.
and difference constraints(与差分约束)，601-607
Dijkstra's algorithm for(Dijkstra 算法)，595-601
in a directed acyclic graph(有向无环图)，592-595
in ε-dense graphs(ε 稠密图)，641 pr.
Gabow's scaling algorithm for(Gabow 改变尺度算法)，615 pr.
and longest paths(与最长路径)，966
singleton(单元集)，1073
singly connected graph(单连通图)，549 ex.
singly linked list(单链表)，204
参见 linked list
singular matrix(奇异矩阵)，730
singular value decomposition(奇异值分解)，769
sink(汇)，530 ex.，644，647
size(规模)

of an algorithm's input(算法输入)，23，849-850，973-975
of a binomial tree(二项树)，457
of a boolean combinational circuit(布尔组合电路)，989
of a clique(团)，1003
of a comparison network(比较网络)，707
of a set(集合)，1073
of a sorting network(排序网络)，708 ex.
of a subtree in a Fibonacci heap(斐波那契堆中子树)，495
of a vertex cover(顶点覆盖)，1006，1024
skew symmetry(斜对称)，644
skip list(跳表)，301
slack(松弛)，781
slack form(松弛型)，773，781-783
uniqueness of(唯一性)，801
slack variable(松弛变量)，781
slot(槽)，222
SLOW-ALL-PAIRS-SHORTEST-PATHS，625
solution(解)
to an abstract problem(抽象问题)，972
basic(基本)，792
to a computational problem(计算问题)，6
to a concrete problem(具体问题)，973
feasible(可行的)，601，773，778
infeasible(不可行的)，778
optimal(最优的)，778
to a system of linear equations(线性方程组)，742
sorted linked list(有序链表)，204
参见 linked list
SORTER，719，720 ex.
sorting(排序)，15-19，28-36，123-182
average-case lower bound for(平均情况下界)，178 pr.
bubblesort(冒泡排序)，38 pr.
bucket sort(桶排序)，174-177
comparison sort(比较排序)，165
counting sort(计数排序)，168-170
fuzzy(模糊)，163 pr.
heapsort(堆排序)，127-144
insertion sort(插入排序)，11，15-19
lexicographic(词典的)，269 pr.
in linear time(线性时间)，168-177，178 pr.
lower bounds for(下界)，165-168
of a matrix(矩阵)，721 ex.
merge sort(合并排序)，11，28-36
network for(网络)，见 sorting network
in place(就地)，16，124
of points by polar angle(用极角的点)，939 ex.
problem of(问题)，5，15，123
quicksort(快速排序)，145-164
radix sort(基数排序)，170-173
selection sort(选择排序)，27 ex.
Shell's sort(Shell 排序)，40
topological(拓扑的)，见 topological sort
using a binary search tree(用二叉搜索树)，264 ex.
using networks(用网络)，见 sorting network
variable-length items(可变长度的项)，179 pr.
sorting network(排序网络)，707
AKS，724
based on insertion sort(基于插入排序)，708 ex.
based on merge sort(基于合并排序)，719-721
bitonic(双调的)，712-716
depth of(深度)，708 ex.，720 ex.
odd-even(奇偶)，721 pr.
size of(规模)，708 ex.
source(源)，531，581，644，647
spanning tree(生成树)，394，561
bottleneck(瓶颈)，577 pr.
verification of(验证)，579
参见 minimum spanning tree
sparse graph(稀疏图)，527
sparse hulled distribution(稀疏包分布)，964 pr.
spindle(轴)，435
spine(脊)，296 pr.
splay tree(伸展树)，301，432
spline(样条)，767 pr.
splitting(分裂)
of B-tree nodes(B 树结点)，443-445
of 2-3-4 trees(2-3-4 树)，453 pr.
splitting summations(分割求和)，1065-1066
spurious hit(假的命中)，912
square matrix(方阵)，726
square of a directed graph(有向图的平方)，530 ex.
square root(平方根)，modulo a prime(模素数)，903 pr.
squaring(平方)，repeated(重复)

for all-pairs shortest paths(每对顶点间最短路径)，625-627
for raising a number to a power(求数的乘幂)，879
stability(稳定性)
numerical(数值)，725，743，769
of sorting algorithms(排序算法)，170，173 ex.
stack(栈)，200-201
in Graham's scan(Graham 扫描)，949
implemented by queues(用队列实现)，204 ex.
linked-list implementation of(链表实现)，208 ex.
operations analyzed by accounting method(用记账方法分析的操作)，410-411
operations analyzed by aggregate analysis(用聚集方法分析的操作)，406-408
operations analyzed by potential method(用势方法分析的操作)，413-414
for procedure execution(过程执行)，162 pr.
on secondary storage(辅助存储)，452 pr.
STACK-EMPTY，201
standard deviation(标准差)，1110
standard form(标准型)，773，777-781
star-shaped polygon(星型多边形)，956 ex.
start state(开始状态)，916
start time(开始时间)，371
state of a finite automaton(有穷自动机的状态)，916
static graph(静止图)，499 n.
static set of keys(关键字的静态集合)，245
Stirling's approximation(斯特林近似)，55
STOOGE-SORT，161 pr.
storage management(存储管理)，127，210-212，213 ex.，229 ex.
store instruction(存储指令)，22
straddle(横跨)，936
Strassen's algorithm(Strassen 算法)，735-742
streaks(序列)，110-114
strictly decreasing(严格递减)，51
strictly increasing(严格递增)，51
string(串)，906，1095
string matching(串匹配)，906-932
based on repetition factors(基于重复因子)，931 pr.
by finite automata(用有穷自动机)，916-923
with gap characters(带分隔符)，910 ex.，923 ex.
Knuth-Morris-Pratt algorithm for(Knuth-Morris-Pratt 算法)，923-931
naive algorithm for(朴素算法)，909-911
Rabin-Karp algorithm for(Rabin-Karp 算法)，911-916
string-matching automaton(串匹配自动机)，917-922，923 ex.
strongly connected component(强连通分支)，1082
decomposition into(分解成)，552-557
STRONGLY-CONNECTED-COMPONENTS，554
strongly connected graph(强连通图)，1082
subgraph(子图)，1082
predecessor(前趋)，见 predecessor subgraph
subgraph-isomorphism problem(子图同构问题)，1017 ex.
subgroup(子群)，866-868
subpath(子路径)，1081
subroutine(子例程)
calling(调用)，20，22，23 n.
executing(执行)，23 n.
subsequence(子序列)，350
subset(子集)
hereditary family of(遗传族)，393
independent family of(独立族)，393
subset(子集)(⊆)，1071
SUBSET-SUM，1013
subset-sum problem(子集和问题)
approximation algorithm for(逼近算法)，1043-1049
NP-completeness of(NP 完全性)，1013-1017
with unary target(带一元目标)，1017 ex.
substitution method(替换方法)，63-67
and recursion trees(与递归树)，70-72
substring(子串)，1095
subtract instruction(减指令)，22
subtraction of matrices(矩阵的减法)，729
subtree(子树)，1087
maintaining sizes of(维护规模)，in order-statistic trees(顺序统计量树)，306-307
success in a Bernoulli trial(伯努利试验中的成功)，1112
successor(后继)
in binary search trees(二叉搜索树)，258-259
finding ith(找出第 i 个)，of a node in an order-statistic tree(顺序统计量树的结点)，307 ex.

in linked lists(链表)，204
in order-statistic trees(顺序统计量树)，310 ex.
in red-black trees(红黑树)，276
SUCCESSOR，198
suffix(后缀)(⊐)，907
suffix function(后缀函数)，917
suffix-function inequality(后缀函数不等式)，920
suffix-function recursion lemma(后缀递归引理)，920
sum(和)
Cartesian(笛卡儿)，830 ex.
flow(流)，650 ex.
infinite(无穷的)，1058
of matrices(矩阵)，728
of polynomials(多项式)，822
rule of(准则)，1094
telescoping(叠缩)，1061
summation(求和)，1058-1069
in asymptotic notation(渐近记号)，47，1059
bounding(界限)，1062-1069
formulas and properties of(公式和性质)，1058-1062
implicit(隐式的)，648
linearity of(线性)，1059
summation lemma(求和引理)，832
superpolynomial time(超多项式时间)，966
supersink(超汇)，647
supersource(超源)，647
surjection(满射)，1078
SVD，769
sweeping(扫除)，940-947，962 pr.
sweep line(扫除线)，940
sweep-line status(扫除线状态)，942-943
symbol table(符号表)，221，230，232
symmetric difference(对称差)，696 pr.
symmetric matrix(对称矩阵)，728，733-734 ex.
symmetric positive-definite matrix(对称正定矩阵)，760-762
symmetric relation(对称关系)，1075
symmetry of Θ-notation(Θ记号的对称性)，49
systems of difference constraints(差分约束系统)，601-607
systems of linear equations(线性方程组)，742-755，767 pr.

T

TABLE-DELETE，422
TABLE-INSERT，418
tail(尾)
of a binomial distribution(二项分布)，1118-1125
of a linked list(链表)，204
of a queue(队列)，202
tail recursion(尾递归)，162 pr.，376
target(目标)，1013
Tarjan's off-line least-common-ancestors algorithm(Tarjan 的脱机最小公共祖先算法)，521 pr.
task(任务)，399
task scheduling(任务调度)，399-402，404 pr.
tautology(重试)，983 ex.，1002 ex.
Taylor series(泰勒级数)，271 pr.
telescoping series(叠缩级数)，1061
telescoping sum(叠缩和式)，1061
testing(测试)
of primality(素数)，887-896，904
of pseudoprimality(伪素数)，889-890
text in string matching(串匹配中的文字)，906
then(then 子句)，in pseudocode(伪码)，19
3-CNF，999
3-CNF-SAT，999
3-CNF satisfiability(3-CNF 可满足性)，998 1002
approximation algorithm for(逼近算法)，1039-1040
and 2-CNF satisfiability(与 2-CNF 可满足性)，967
3-COLOR，1019 pr.
3-conjunctive normal form(3 合取范式)，999
tight constraint(紧约束)，791
time(时间)，见 running time
time domain(时间域)，822
timestamp(时间戳)，540，548 ex.
Toeplitz matrix(Toeplitz 矩阵)，844 pr.
TOP，949
top of a stack(栈顶)，200
topological sort(拓扑排序)，549-552
in computing single-source shortest paths in a dag(计算有向无环图内单源最短路径)，592
TOPOLOGICAL-SORT，550
total net flow(总净流)，645

total order(全序)，1077
total path length(总路径长度)，270 pr.
total positive flow(总正流)，645
tour(回路)
  bitonic(双调)，364 pr.
  Euler(欧拉)，559 pr.，966
  of a graph(图)，1012
track(磁道)，435
tractability(易处理)，966
transition function(变换函数)，916，921-922
transitive closure(传递闭包)，632-635
  and boolean matrix multiplication(与布尔矩阵乘法)，759 ex.
  of dynamic graphs(动态图)，641 pr.
TRANSITIVE-CLOSURE，633
transitive relation(传递关系)，1075
transitivity of asymptotic notation(渐近记号的传递性)，49
transpose(转置)
  conjugate(共轭)，759 ex.
  of a directed graph(有向图)，530 ex.
  of a matrix(矩阵)，529，726
transpose symmetry of asymptotic notation(渐近记号的转置对称性)，49
transposition network(转置网络)，721 pr.
traveling-salesman problem(旅行商问题)
  approximation algorithm for(近似算法)，1027-1033
  bitonic euclidean(双调欧几里得)，364 pr.
  bottleneck(瓶颈)，1033 ex.
  NP-completeness of(NP 完全性)，1012-1013
  with the triangle inequality(带三角不等式)，1028-1031
  without the triangle inequality(不带三角不等式)，1031-1032
traversal of a tree(树的遍历)，见 tree walk
treap，296 pr.
TREAP-INSERT，296 pr.
tree(树)，1085-1091
  AA-trees(AA 树)，301
  AVL，296 pr.
  binary(二叉)，见 binary tree
  binomial，457-459，479
  bisection of(二分)，1092 pr.
  breadth-first(广度优先)，532，538
  B-trees(B 树)，434-454
  decision(决策)，166-167
  depth-first(深度优先)，540
  diameter of(直径)，539 ex.
  dynamic(动态的)，432
  free(释放)，1083，1085-1087
  full walk of(完全遍历)，1030
  fusion(聚合)，182，433
  heap(堆)，127-144
  height-balanced(高度平衡的)，296 pr.
  height of(高度)，1088
  interval(区间)，311-317
  k-neighbor(k 邻居)，301
  minimum spanning(最小生成)，见 minimum spanning tree
  optimal binary search(最优二叉搜索)，356-363，369
  order-statistic(顺序统计量)，302-308
  parse(语法分析)，999
  recursion(递归)，36，67-72
  red-black(红黑)，见 red-black tree
  rooted(有根的)，214-217，1087
  scapegoat，301
  search(搜索)，见 search tree
  shortest-paths(最短路径)，584，610-613
  spanning(生成)，见 minimum spanning tree，spanning tree
  splay(伸展)，301，432
  treap，296 pr.
  2-3，300，454
  2-3-4，439，453 pr.
  walk(遍历)，见 tree walk
  weight-balanced trees(带权平衡树)，301
TREE-DELETE，262，288
tree edge(树的边)，538，540，546
TREE-INSERT，261，280
TREE-MAXIMUM，258
TREE-MINIMUM，258
TREE-PREDECESSOR，259
TREE-SEARCH，257
TREE-SUCCESSOR，259
tree walk(树的遍历)，254，260 ex.，305，1030
trial(试验)，Bernoulli(伯努利)，1112

trial division(试除)，888
triangle inequality(三角不等式)，1028
  and negative-weight edges(与负权值边)，1032 ex.
  for shortest paths(最短路径)，587，607-608
triangular matrix(三角矩阵)，727-728，733 ex.
trichotomy(三分法)，interval(区间)，311
trichotomy property of real numbers(实数的三分性质)，49
tridiagonal linear systems(三对角线性系统)，767 pr.
tridiagonal matrix(三对角矩阵)，727
trie(检索树)，见 radix tree
TRIM，1046
trimming of a list(列表的削减)，1045
trivial divisor(平凡因子)，851
truth assignment(真值赋值)，988，996
truth table(真值表)，987
TSP，1012
tuple(元组)，1074
twiddle factor(旋转因子)，836
2-CNF-SAT，1003 ex.
2-CNF satisfiability(2-CNF 可满足性)，1003 ex.
  and 3-CNF satisfiability(与 3-CNF 可满足性)，967
two-pass method(两遍扫描法)，508
2-3-4 heap(2-3-4 堆)，473 pr.
2-3-4 tree(2-3-4 树)，439
  joining(连接)，453 pr.
  and red-black trees(与红黑树)，441 ex.
  splitting(分裂)，453 pr.
2-3 tree(2-3 树)，300，454

U

unary(一元的)，974
unbounded linear program(无界线性规划)，778
unconditional branch instruction(无条件分支指令)，22
uncountable set(不可数集合)，1073
underdetermined system of linear equations(欠定线性方程组)，743
underflow(下溢)
  of a queue(队列)，202
  of a stack(栈)，201
undirected graph(无向图)，1080
  articulation point of(挂接点)，558 pr.
  biconnected component of(双连通分支)，558 pr.
  bridge of(桥)，558 pr.
  clique in(团)，1003
  coloring of(着色)，1019 pr.，1091 pr.
  computing a minimum spanning tree in(计算最小生成树)，561-579
  converting to(转换成)，from a multigraph(从多图)，530 ex.
  d-regular(d 正则)，669 ex.
  grid(网格)，692 pr.
  hamiltonian(哈密顿)，979
  independent set of(独立集)，1018 pr.
  matching of(匹配)，664
  nonhamiltonian(非哈密顿)，979
  vertex cover of(顶点覆盖)，1006，1024
  参见 graph
undirected version of a directed graph(有向图的无向版本)，1082
uniform hashing(均匀散列)，239
uniform probability distribution(均匀概率分布)，1101-1102
uniform random permutation(均匀随机排列)，93，101
union(并)
  of dynamic sets(动态集合)，见 uniting
  of languages(语言)，976
  of sets(集合)($\cup$)，1071
UNION，455，499
  disjoint-set-forest implementation of(不相交集合森林实现)，508
  linked-list implementation of(链表实现)，502-504，505 ex.
union by rank(按秩合并)，506
unique factorization of integers(整数唯一因数分解)，854
unit(单位)(1)，851
uniting(联合)
  of binomial heaps(二项堆)，462-468
  of Fibonacci heaps(斐波那契堆)，481-482
  of heaps(堆)，455
  of linked lists(链表)，208 ex.
  of 2-3-4 heaps(2-3-4 堆)，473 pr.
unit lower-triangular matrix(单位下三角矩阵)，728

unit-time task(单位时间任务)，399
unit upper-triangular matrix(单位上三角矩阵)，727
unit vector(单位向量)，726
universal hashing(全域散列)，232-236
universal sink(通用汇)，530 ex.
universe(全域)，1072
unmatched vertex(不匹配的顶点)，664
unordered binomial tree(不匹配的二项树)，479
unsorted linked list(无序链表)，204
  参见 linked list
unweighted longest simple paths(无权最长简单路径)，342
unweighted shortest paths(无权最短路径)，341
upper-bound property(上界性质)，587，608
upper median(上中位数)，183
upper-triangular matrix(上三角矩阵)，727

V

valid shift(合法移位)，906
value(值)
  of a flow(流)，644
  of a function(函数)，1078
  objective(目标)，774，778
Vandermonde matrix(范德蒙德矩阵)，734 ex.
van Emde Boas data structure(van Emde Boas 数据结构)，144，433
Var [] (variance)(方差)，1110
variable(变量)
  basic(基本的)，782
  entering(换入)，793
  leaving(换出)，793
  nonbasic(非基本的)，782
  in pseudocode(伪码)，19
  random(随机)，1106-1111
  slack(松弛)，781
  参见 indicator random variable
variable-length code(可变长度编码)，385
variance(方差)，1109
  of a binomial distribution(二项分布)，1115
  of a geometric distribution(几何分布)，1112
vector(向量)，726，730-731
  convolution of(卷积)，825
  cross product of(叉积)，934
  orthonormal(正交的)，769
  in the plane(平面上)，934
Venn diagram(文氏图)，1072
verification(验证)，979-983
  of spanning trees(生成树)，579
  verification algorithm(验证算法)，980
vertex(顶点)
  articulation point(挂接点)，558 pr.
  in a graph(图)，1080
  intermediate(中间的)，629
  isolated(孤立的)，1081
  of a polygon(多边形的)，939 ex.
  selector(选择器)，1009
vertex cover(顶点覆盖)，1006，1024，1040-1043
VERTEX-COVER，1006
vertex-cover problem(顶点覆盖问题)
  approximation algorithm for(逼近算法)，1024-1027
  NP-completeness of(NP 完全性)，1006-1008
vertex set(顶点集合)，1080
violation of an equality constraint(等式约束的违反)，791
virtual memory(虚拟存储)，22
Viterbi algorithm(Viterbi 算法)，367 pr.
VLSI (very large scale integration)(超大规模集成)，87 n.

W

walk of a tree(树的遍历)，见 tree walk
weak duality(弱对偶性)，805
weight(权)
  of a cut(割)，1043 ex.
  of an edge(边)，529
  mean(平均)，617 pr.
  of a path(路径)，580
weight-balanced tree(权平衡树)，301，427 pr.
weighted bipartite matching(带权二分匹配)，497
weighted matroid(带权拟阵)，394-398
weighted median(带权中位数)，194 pr.
weighted set-covering problem(带权集合覆盖问题)，1050 pr.
weighted-union heuristic(带权合并启发式)，503
weighted vertex cover(带权顶点覆盖)，1040-1043