

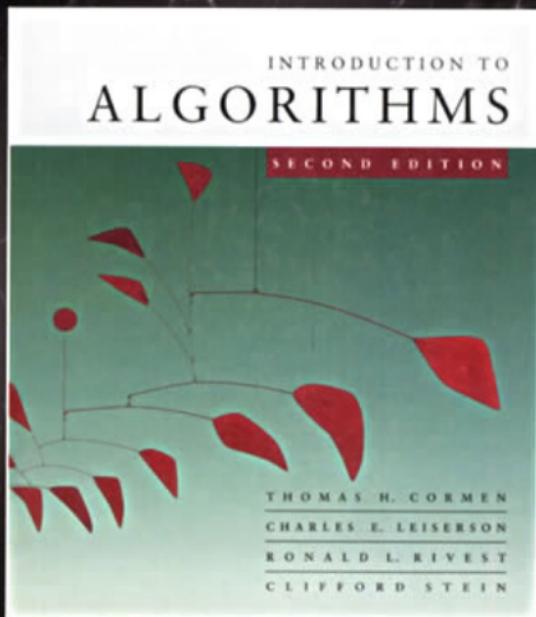


计算机科学丛书

原书第2版

# 算法导论

(美) Thomas H. Cormen Charles E. Leiserson 著 潘金贵 顾铁成 李成法 叶惠 译  
Ronald L. Rivest Clifford Stein



Introduction to Algorithms  
Second Edition



机械工业出版社  
China Machine Press

# 目 录

出版者的话	5.4.4 在线雇用问题	66
专家指导委员会		
译者序		
前言		
<b>第一部分 基础知识</b>		
引言	1	
第1章 算法在计算中的作用	3	
1.1 算法	3	
1.2 作为一种技术的算法	6	
第2章 算法入门	9	
2.1 插入排序	9	
2.2 算法分析	12	
2.3 算法设计	16	
2.3.1 分治法	16	
2.3.2 分治法分析	20	
第3章 函数的增长	26	
3.1 演近记号	26	
3.2 标准记号和常用函数	31	
第4章 递归式	38	
4.1 代换法	38	
4.2 递归树方法	40	
4.3 主方法	43	
*4.4 主定理的证明	45	
4.4.1 取正合幕时的证明	45	
4.4.2 上取整函数和下取整函数	48	
第5章 概率分析和随机算法	54	
5.1 雇用问题	54	
5.2 指示器随机变量	56	
5.3 随机算法	58	
*5.4 概率分析和指示器随机变量的进一步使用	62	
5.4.1 生日悖论	62	
5.4.2 球与盒子	64	
5.4.3 序列	64	
<b>第二部分 排序和顺序统计学</b>		
引言	71	
第6章 堆排序	73	
6.1 堆	73	
6.2 保持堆的性质	74	
6.3 建堆	76	
6.4 堆排序算法	78	
6.5 优先级队列	80	
第7章 快速排序	85	
7.1 快速排序的描述	85	
7.2 快速排序的性能	88	
7.3 快速排序的随机化版本	90	
7.4 快速排序分析	91	
7.4.1 最坏情况分析	91	
7.4.2 期望的运行时间	92	
第8章 线性时间排序	97	
8.1 排序算法时间的下界	97	
8.2 计数排序	98	
8.3 基数排序	100	
8.4 桶排序	102	
第9章 中位数和顺序统计学	108	
9.1 最小值和最大值	108	
9.2 以期望线性时间做选择	109	
9.3 最坏情况线性时间的选择	112	
<b>第三部分 数据结构</b>		
引言	117	
第10章 基本数据结构	119	
10.1 栈和队列	119	
10.2 链表	121	
10.3 指针和对象的实现	124	
10.4 有根树的表示	127	
第11章 散列表	132	

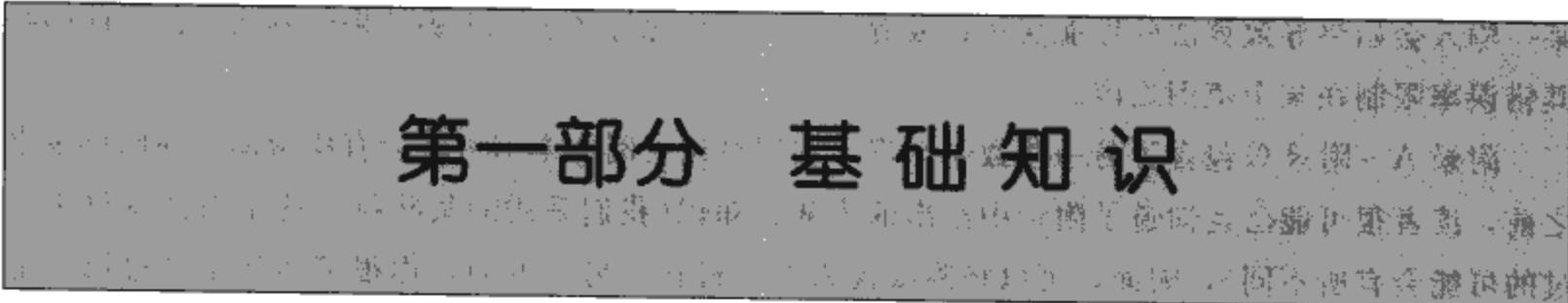
11.1 直接寻址表 .....	132	17.3 势能方法 .....	249
11.2 散列表 .....	133	17.4 动态表 .....	251
11.3 散列函数 .....	137	17.4.1 表扩张 .....	251
11.3.1 除法散列法 .....	138	17.4.2 表扩张和收缩 .....	253
11.3.2 乘法散列法 .....	138		
*11.3.3 全域散列 .....	139		
11.4 开放寻址法 .....	142	<b>第五部分 高级数据结构</b>	
*11.5 完全散列 .....	146	概述 .....	261
<b>第 12 章 二叉查找树</b> .....	151	第 18 章 B 树 .....	263
12.1 二叉查找树 .....	151	18.1 B 树的定义 .....	265
12.2 查询二叉查找树 .....	153	18.2 对 B 树的基本操作 .....	267
12.3 插入和删除 .....	155	18.3 从 B 树中删除关键字 .....	272
*12.4 随机构造的二叉查找树 .....	158	<b>第 19 章 二项堆</b> .....	277
<b>第 13 章 红黑树</b> .....	163	19.1 二项树与二项堆 .....	278
13.1 红黑树的性质 .....	163	19.1.1 二项树 .....	278
13.2 旋转 .....	165	19.1.2 二项堆 .....	279
13.3 插入 .....	167	19.2 对二项堆的操作 .....	281
13.4 删除 .....	172	<b>第 20 章 斐波那契堆</b> .....	291
<b>第 14 章 数据结构的扩张</b> .....	181	20.1 斐波那契堆的结构 .....	291
14.1 动态顺序统计 .....	181	20.2 可合并堆的操作 .....	293
14.2 如何扩张数据结构 .....	184	20.3 减小一个关键字与删除一个结点 .....	299
14.3 区间树 .....	186	20.4 最大度数的界 .....	302
<b>第四部分 高级设计和分析技术</b>		<b>第 21 章 用于不相交集合的数据结构</b> .....	305
<b>导论</b> .....	191	21.1 不相交集合上的操作 .....	305
<b>第 15 章 动态规划</b> .....	192	21.2 不相交集合的链表表示 .....	307
15.1 装配线调度 .....	192	21.3 不相交集合森林 .....	310
15.2 矩阵链乘法 .....	197	*21.4 带路径压缩的按秩合并的分析 .....	312
15.3 动态规划基础 .....	202	<b>第六部分 图 算 法</b>	
15.4 最长公共子序列 .....	208	<b>引言</b> .....	321
15.5 最优二叉查找树 .....	212	<b>第 22 章 图的基本算法</b> .....	322
<b>第 16 章 贪心算法</b> .....	222	22.1 图的表示 .....	322
16.1 活动选择问题 .....	222	22.2 广度优先搜索 .....	324
16.2 贪心策略的基本内容 .....	228	22.3 深度优先搜索 .....	330
16.3 赫夫曼编码 .....	231	22.4 拓扑排序 .....	336
*16.4 贪心法的理论基础 .....	236	22.5 强连通分支 .....	338
*16.5 一个任务调度问题 .....	239	<b>第 23 章 最小生成树</b> .....	344
<b>第 17 章 平摊分析</b> .....	244	23.1 最小生成树的形成 .....	345
17.1 聚集分析 .....	244	23.2 Kruskal 算法和 Prim 算法 .....	348
17.2 记账方法 .....	247	<b>第 24 章 单源最短路径</b> .....	357

24.1 Bellman-Ford 算法 .....	362	第 31 章 有关数论的算法 .....	522
24.2 有向无回路图中的单源最短路径 .....	364	31.1 初等数论概念 .....	522
24.3 Dijkstra 算法 .....	366	31.2 最大公约数 .....	526
24.4 差分约束与最短路径 .....	370	31.3 模运算 .....	529
24.5 最短路径性质的证明 .....	373	31.4 求解模线性方程 .....	533
<b>第 25 章 每对顶点间的最短路径 .....</b>	<b>381</b>	31.5 中国余数定理 .....	535
25.1 最短路径与矩阵乘法 .....	382	31.6 元素的幂 .....	538
25.2 Floyd-Warshall 算法 .....	386	31.7 RSA 公钥加密系统 .....	540
25.3 稀疏图上的 Johnson 算法 .....	391	*31.8 素数的测试 .....	544
<b>第 26 章 最大流 .....</b>	<b>396</b>	*31.9 整数的因子分解 .....	550
26.1 流网络 .....	396	<b>第 32 章 字符串匹配 .....</b>	<b>557</b>
26.2 Ford-Fulkerson 方法 .....	400	32.1 朴素的字符串匹配算法 .....	558
26.3 最大二分匹配 .....	408	32.2 Rabin-Karp 算法 .....	560
*26.4 压入与重标记算法 .....	411	32.3 利用有限自动机进行字符串匹配 .....	563
*26.5 重标记与前移算法 .....	419	*32.4 Knuth-Morris-Pratt 算法 .....	568
<b>第七部分 算法研究问题选编</b>			
<b>引言 .....</b>	<b>431</b>	<b>第 33 章 计算几何学 .....</b>	<b>575</b>
<b>第 27 章 排序网络 .....</b>	<b>433</b>	33.1 线段的性质 .....	575
27.1 比较网络 .....	433	33.2 确定任意一对线段是否相交 .....	580
27.2 0-1 原理 .....	436	33.3 寻找凸包 .....	584
27.3 双调排序网络 .....	438	33.4 寻找最近点对 .....	591
27.4 合并网络 .....	440	<b>第 34 章 NP 完全性 .....</b>	<b>597</b>
27.5 排序网络 .....	442	34.1 多项式时间 .....	600
<b>第 28 章 矩阵运算 .....</b>	<b>446</b>	34.2 多项式时间的验证 .....	605
28.1 矩阵的性质 .....	446	34.3 NP 完全性与可归约性 .....	608
28.2 矩阵乘法的 Strassen 算法 .....	451	34.4 NP 完全性的证明 .....	615
28.3 求解线性方程组 .....	455	34.5 NP 完全问题 .....	620
28.4 矩阵求逆 .....	464	34.5.1 团问题 .....	620
28.5 对称正定矩阵与最小二乘逼近 .....	467	34.5.2 顶点覆盖问题 .....	622
<b>第 29 章 线性规划 .....</b>	<b>473</b>	34.5.3 哈密顿回路问题 .....	623
29.1 标准型和松弛型 .....	477	34.5.4 旅行商问题 .....	626
29.2 将问题表达为线性规划 .....	482	34.5.5 子集和问题 .....	627
29.3 单纯形算法 .....	485	<b>第 35 章 近似算法 .....</b>	<b>633</b>
29.4 对偶性 .....	495	35.1 顶点覆盖问题 .....	634
29.5 初始基本可行解 .....	498	35.2 旅行商问题 .....	636
<b>第 30 章 多项式与快速傅里叶变换 .....</b>	<b>506</b>	35.2.1 满足三角不等式的旅行商 问题 .....	636
30.1 多项式的表示 .....	507	35.2.2 一般旅行商问题 .....	638
30.2 DFT 与 FFT .....	511	35.3 集合覆盖问题 .....	640
30.3 有效的 FFT 实现 .....	516	35.4 随机化和线性规划 .....	643
		35.5 子集和问题 .....	646

**第八部分 附录：数学基础知识**

引言 .....	653
A 求和 .....	654
A.1 求和公式及其性质 .....	654
A.2 确定求和时间的界 .....	656
B 集合等离散数学结构 .....	661
B.1 集合 .....	661
B.2 关系 .....	664
B.3 函数 .....	665
B.4 图 .....	667
B.5 树 .....	670
B. 5.1 自由树 .....	670
B. 5.2 有根树和有序树 .....	671
B. 5.3 二叉树与位置树 .....	672
C 计数和概率 .....	676
C.1 计数 .....	676
C.2 概率 .....	679
C.3 离散随机变量 .....	683
C.4 几何分布与二项分布 .....	686
*C.5 二项分布的尾 .....	689
参考文献 .....	694
索引 .....	711





# 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入门指南的风格来编写的。4

# 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  $\langle a_1, a_2, \dots, a_n \rangle$ 。

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

例如，给定一个输入序列  $\langle 31, 41, 59, 26, 41, 58 \rangle$ ，一个排序算法返回的输出序列是  $\langle 26, 31, 41, 41, 58, 59 \rangle$ 。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。

5

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)。
- 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  $\langle A_1, A_2, \dots, A_n \rangle$ ，希望确定其乘积  $A_1 A_2 \cdots A_n$ 。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： $(A_1 (A_2 (A_3 A_4))), (A_1 ((A_2 A_3) A_4)), ((A_1 A_2) (A_3 A_4)), ((A_1 (A_2 A_3)) A_4), (((A_1 A_2) A_3) A_4)$ 。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。
- 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  $x = 0, 1, \dots, n-1$  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长不了多少的总体距离。第 35 章讨论了这种“近似算法”。

9

## 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），但往往会选择最容易实现的方法。

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。

10

## 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]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。

14

## 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 $\langle a_1, a_2, \dots, a_n \rangle$ 。

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

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。

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

15



图2-1 利用插入排序方法来排序手中的牌

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。

```
1  for  $j \leftarrow 2$  to  $\text{length}[A]$ 
2    do  $\text{key} \leftarrow A[j]$ 
3       $\triangleright$  Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ .
4       $i \leftarrow j-1$ 
5      while  $i > 0$  and  $A[i] > \text{key}$ 
6        do  $A[i+1] \leftarrow A[i]$ 
7         $i \leftarrow i-1$ 
8     $A[i+1] \leftarrow \text{key}$ 
```

### 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。

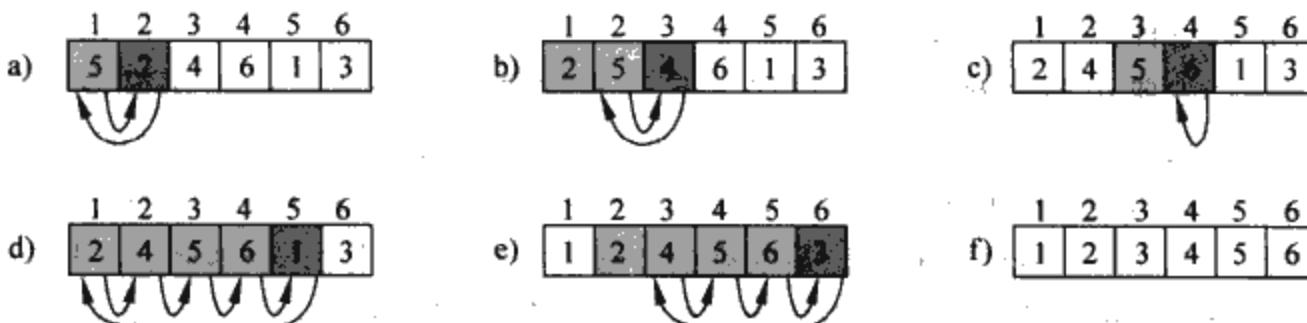


图 2-2 INSERTION-SORT 在数组  $A = \langle 5, 2, 4, 6, 1, 3 \rangle$  上的处理过程。数组下标出现在矩形的上方，数组各个位置中存储的数字出现在矩形中。a)~e) 过程第 1~8 行 **for** 循环各次迭代的情况。在每次迭代中，黑色方框里的值是取自  $A[j]$  的关键字值，在过程第 5 行的测试中，要将它与左边阴影框中的各个值进行比较。阴影箭头示出了过程第 6 行中，将一数组中的各个值向右移一个位置这一动作；黑色箭头示出了要将关键字值移至何处(过程中的第 8 行)。f) 中示出了最终已排好序的数组。

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：

**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， $j=2$ ，<sup>①</sup>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。<sup>②</sup>

2) **while**、**for**、**repeat** 等循环结构和 **if**、**then**、**else** 条件结构与 Pascal 中相同。<sup>③</sup>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。

- 
- ① 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。
  - ② 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。
  - ③ 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。

19 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  $A[1], A[2], \dots, A[j]$ 。

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“ $x \neq \text{NIL}$  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  $A = \langle a_1, a_2, \dots, a_n \rangle$  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。21

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  $c \geq 1$ ，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。22

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 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。<sup>①</sup>

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  $j=2, 3, \dots, n$ ,  $n = \text{length}[A]$ ，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 <b>for</b> $j \leftarrow 2$ <b>to</b> $\text{length}[A]$	$c_1$	$n$
2 <b>do</b> $key \leftarrow A[j]$	$c_2$	$n-1$
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 <b>while</b> $i > 0$ and $A[i] > key$	$c_5$	$\sum_{j=2}^n t_j$
6 <b>do</b> $A[i+1] \leftarrow A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
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  $c_i n$ 。<sup>②</sup> 为计算总运行时间  $T[n]$ ，对每一对 cost 与 times 之积求和，得：

$$\begin{aligned} T(n) = & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) \\ & + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1) \end{aligned}$$

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

① 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以变的。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。

② 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  $j=2, 3, \dots, n$  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  $A[i] \leq key$ 。于是，对  $j=2, 3, \dots, n$ ，有  $t_j=1$ ，最佳的运行时间为：

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

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  $j=2, 3, \dots, n$ ，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

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2}-1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

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：

- 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，低阶项相对来说不太重要。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  $\Theta$  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  $\Theta$  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  $\Theta$  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  $\Theta$  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  $A[1..j-1]$  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  $p \leq q < r$ 。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  $\infty$  来作为哨兵值，这样每当露出一张值为  $\infty$  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  $n_1 \leftarrow q - p + 1$ 
2  $n_2 \leftarrow r - q$ 
3 create arrays  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$ 
4 for  $i \leftarrow 1$  to  $n_1$ 
5   do  $L[i] \leftarrow A[p + i - 1]$ 
6 for  $j \leftarrow 1$  to  $n_2$ 
7   do  $R[j] \leftarrow A[q + j]$ 
8  $L[n_1 + 1] \leftarrow \infty$ 
9  $R[n_2 + 1] \leftarrow \infty$ 
10  $i \leftarrow 1$ 
11  $j \leftarrow 1$ 
12 for  $k \leftarrow p$  to  $r$ 
13   do if  $L[i] \leq R[j]$ 
14     then  $A[k] \leftarrow L[i]$ 
15        $i \leftarrow i + 1$ 
16     else  $A[k] \leftarrow R[j]$ 
17        $j \leftarrow j + 1$ 

```

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  $n_1+1$  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。

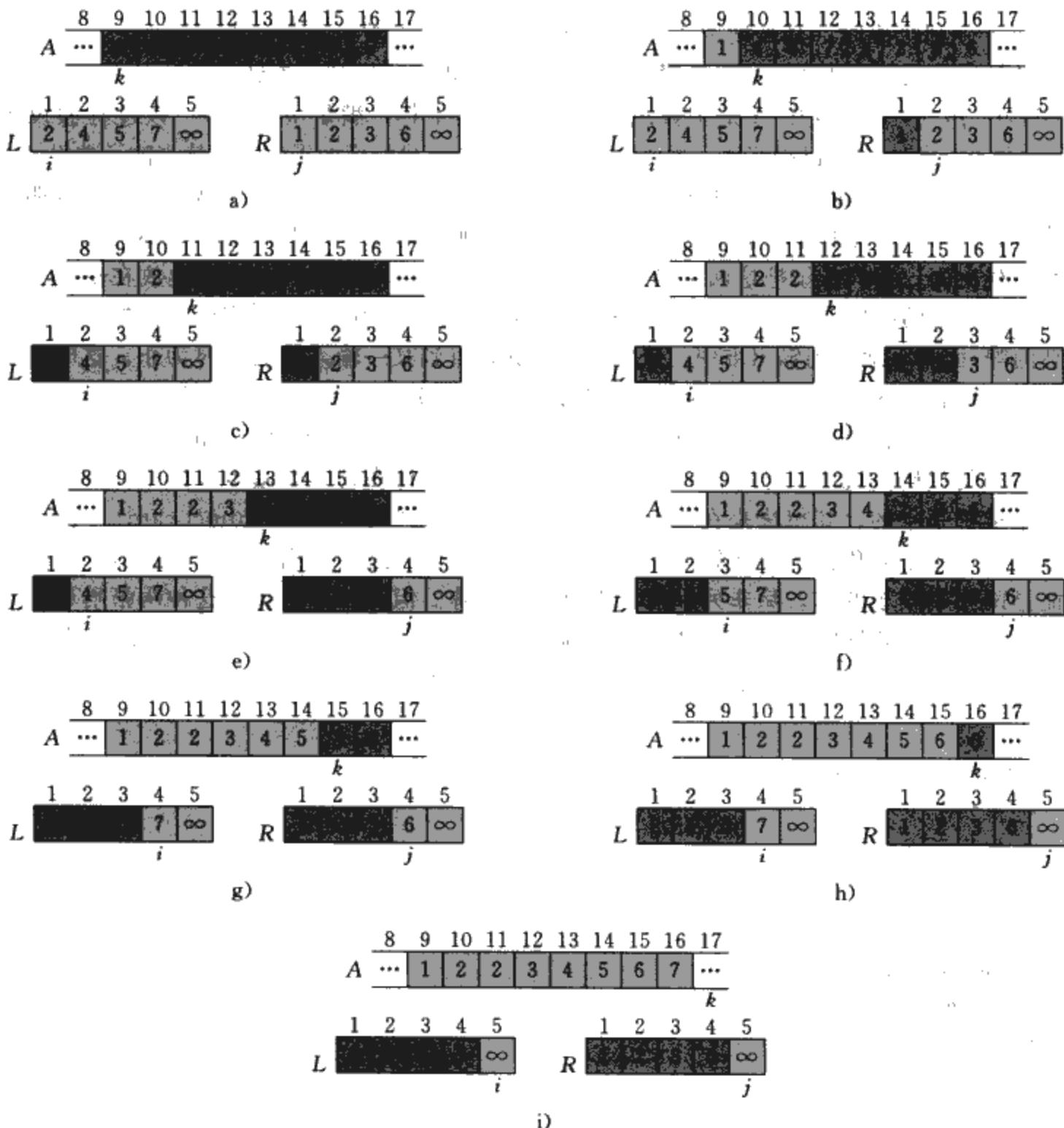


图 2-3 在调用  $\text{MERGE}(A, 9, 12, 16)$  的第 10~17 行的操作中，当子数组  $A[9..16]$  包含序列  $(2, 4, 5, 7, 1, 2, 3, 6)$  时的情况。在复制和插入了哨兵后，数组  $L$  包含了  $(2, 4, 5, 7, \infty)$ ，数组  $R$  包含了  $(1, 2, 3, 6, \infty)$ 。 $A$  中的浅阴影位置包含了它们的最终值， $L$  和  $R$  中的浅阴影位置包含了有待于被复制回  $A$  中的值。合起来看，浅阴影的位置始终包含了最初在  $A[9..16]$  中的值，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  $L[i] \leq R[j]$ 。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  $L[i] \geq R[j]$ ，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。

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  $\Theta(n_1+n_2) = \Theta(n)$ ，<sup>①</sup> 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  $p \geq r$ ，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。<sup>②</sup>

```
MERGE-SORT( $A, p, r$ )
1  if  $p < r$ 
2    then  $q \leftarrow \lfloor (p+r)/2 \rfloor$ 
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  $A = (A[1], A[2], \dots, A[n])$  进行排序，首先要调用  $\text{MERGE-SORT}(A, 1, \text{length}[A])$ ，其中  $\text{length}[A] = n$ 。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。

30  
l  
31

① 我们将在第 3 章中介绍如何来形式化地解释包含  $\Theta$  记号的方程。

②  $\lceil x \rceil$  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。

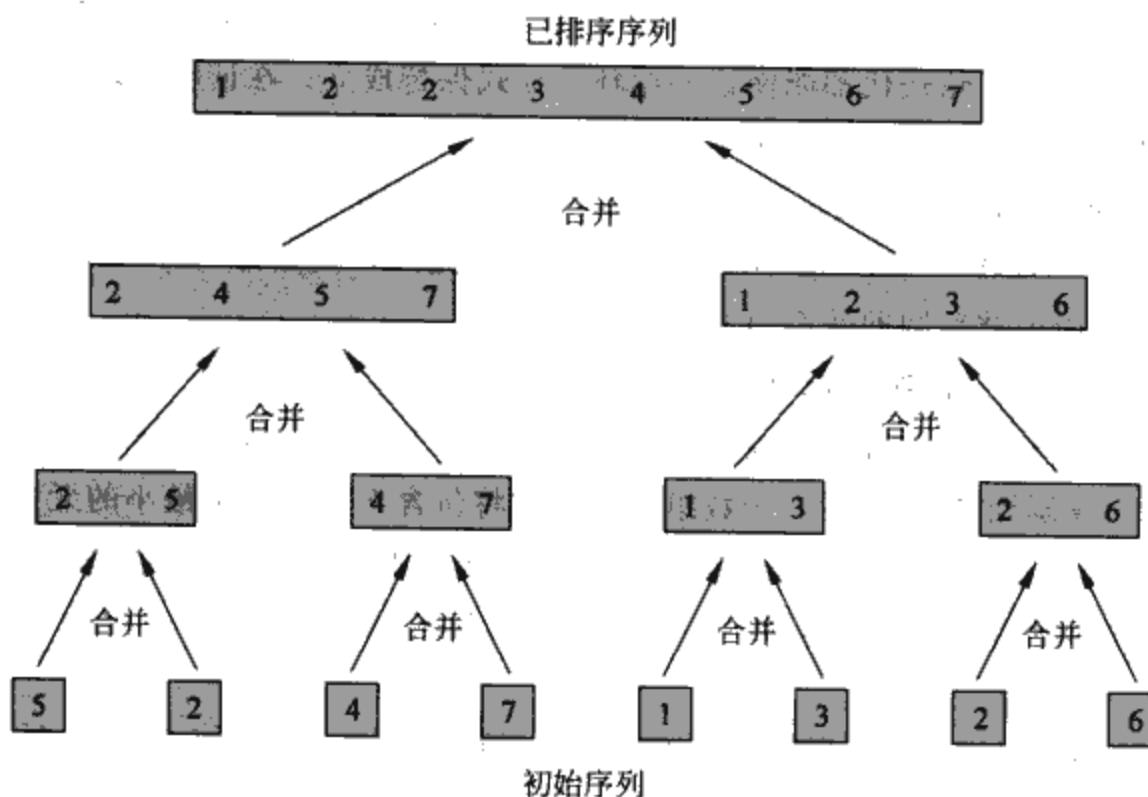


图 2-4 合并排序在数组  $A = \{5, 2, 4, 7, 1, 3, 2, 6\}$  上的处理过程。随着算法的处理过程由底向上进展，待合并的已排序序列长度不断增加

### 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  $n \leq c$  ( $c$  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：

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

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：

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

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：

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

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。<sup>①</sup>

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，最顶层之后的第  $i$  层有  $2^i$  个结点，每一个的代价都是  $c(n/2^i)$ ，于是，顶层之后的第  $i$  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  $\Theta(n \lg n)$ 。

<sup>①</sup> 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  $\Theta(n \lg n)$ 。

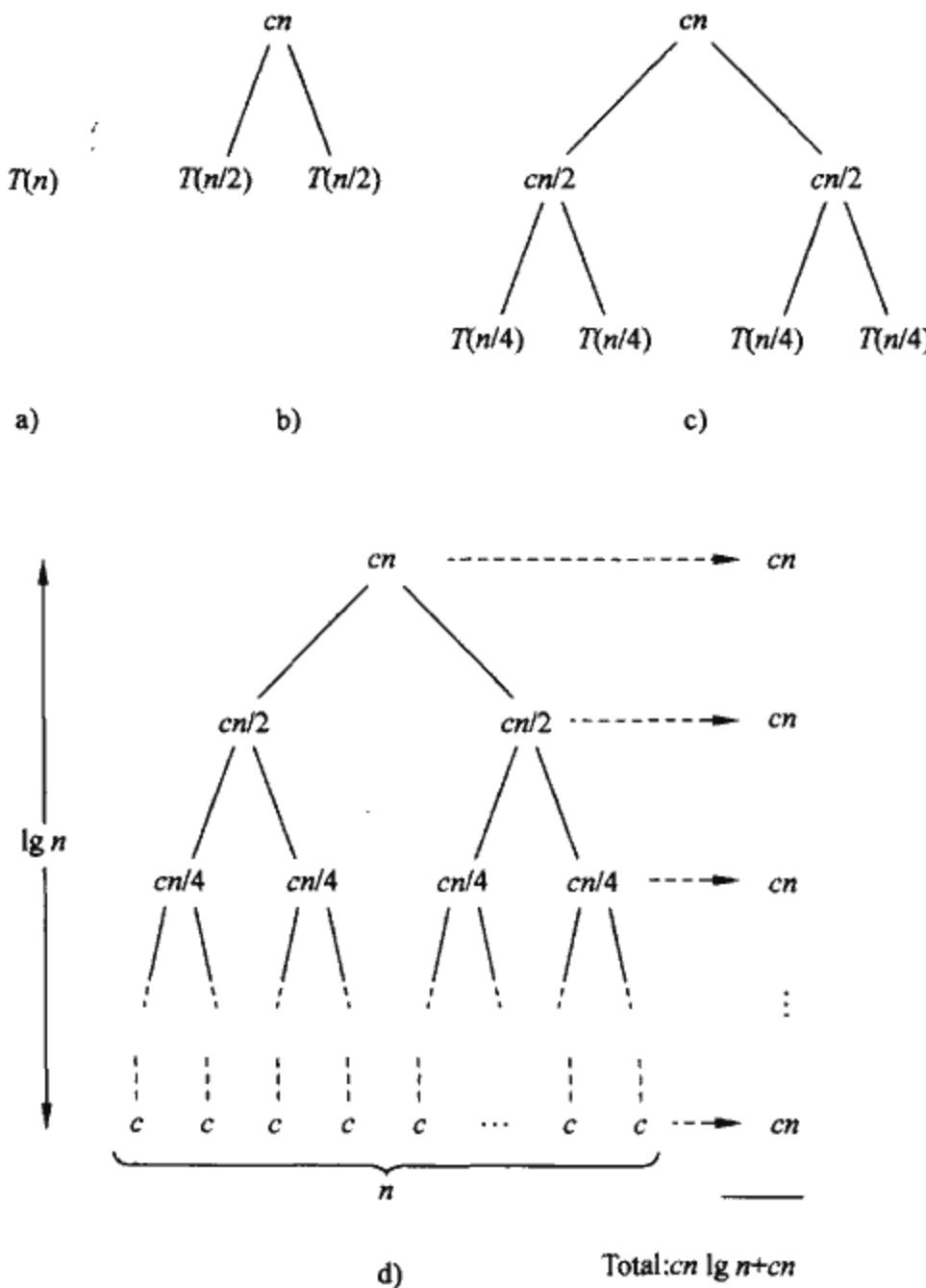


图 2-5 为递归式  $T(n)=2T(n/2)+cn$  构造一棵递归树。a) 图示出了  $T(n)$ , 它被逐步地在 b)~d) 图中进行了扩展以形成递归树。在 d) 图中, 完全扩展了的递归树共有  $\lg n+1$  层(亦即, 其深度为  $\lg n$ , 如图所示), 每一层的总代价都是  $cn$ 。因此, 总的代价就是  $cn \lg n+cn$ , 为  $\Theta(n \lg n)$

## 练习

- 2.3-1 以图 2-4 为模型, 说明合并排序在输入数组  $A=\langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$  上的执行过程。
- 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  $\Theta$  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  $i \leftarrow 1$  to  $\text{length}[A]$ 
2      do for  $j \leftarrow \text{length}[A]$  down to  $i+1$ 
3          do if  $A[j] < A[j-1]$ 
4              then exchange  $A[j] \leftrightarrow A[j-1]$ 
```

- 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：

$$A'[1] \leq A'[2] \leq \dots \leq A'[n] \quad (2.3)$$

其中  $n = \text{length}[A]$ 。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

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

的霍纳规则(Horner's rule)。

给定系数  $a_0, a_1, \dots, a_n$  以及  $x$  的值, 有:

```

1   y ← 0
2   i ← n
3   while i ≥ 0
4       do y ←  $a_i + x \cdot y$ 
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  $a_0, a_1, \dots, a_n$  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  $\langle 2, 3, 8, 6, 1 \rangle$  的 5 个逆序。

b) 如果数组的元素取自集合  $\{1, 2, \dots, n\}$ , 那么, 怎样的数组含有最多的逆序对? 它包含多少个逆序对?

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-Khowārizmī”，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画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  $\Theta$  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  $\mathbb{N} = \{0, 1, 2, \dots\}$  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。41 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。

#### $\Theta$ 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：

$\Theta(g(n)) = \{f(n) : \text{存在正常数 } c_1, c_2 \text{ 和 } n_0, \text{ 使对所有的 } n \geq n_0, \text{ 有 } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$ <sup>○</sup> 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  $c_1 g(n)$  和  $c_2 g(n)$  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  $c_1 g(n)$  和  $c_2 g(n)$  之间。换句话说，对所有的  $n \geq n_0$ ， $f(n)$  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  $f(n) \in \Theta(g(n))$  都是渐近非负的，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  $\Theta$  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”。

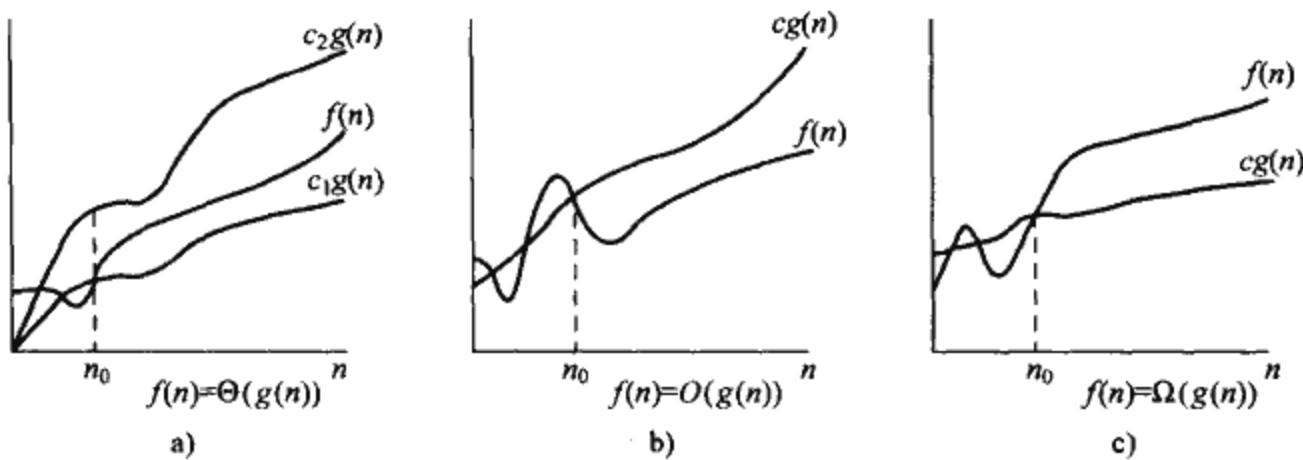


图 3-1 记号  $\Theta$ ,  $O$  和  $\Omega$  的图例。在每个部分中,  $n_0$  是最小的可能值; 大于  $n_0$  的值也有效。a)  $\Theta$  记号限制一个函数在常数因子内。如果存在正常数  $n_0$ ,  $c_1$  和  $c_2$  使得在  $n_0$  右边  $f(n)$  的值永远在  $c_1 g(n)$  与  $c_2 g(n)$  之间, 那么可以写成  $f(n)=\Theta(g(n))$ 。b)  $O$  记号给出一个函数在常数因子内的上限。如果存在正常数  $n_0$  和  $c$  使得在  $n_0$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

在第 2 章中, 引进了非正式的  $\Theta$  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  $n \geq n_0$ , 有

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

成立。用  $n^2$  除不等式得

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

右边的不等式在  $c_2 \geq 1/2$  时对所有的  $n \geq 1$  成立。同样, 左边的不等式可以让  $c_1 \leq 1/14$  时对所有的  $n \geq 7$  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  $n \geq n_0$ , 有  $6n^3 \leq c_2 n^2$ ; 这样就有  $n \leq c_2/6$ , 这对任意大的  $n$  是不可能成立的, 因为  $c_2$  是常数。

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  $\Theta$  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  $n \geq n_0$ ,  $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$  成立。一般来说, 对任何一个多项式  $p(n) = \sum_{i=0}^d a_i n^i$ , 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。<sup>⊖</sup> 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

$\Theta$  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

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

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  $\Theta$  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。

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  $\Theta$  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,  $\Omega$  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

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

读作  $g(n)$  的大  $\Omega$ 。图 3-1c 说明了  $\Omega$  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  $cg(n)$ 。

<sup>⊖</sup> 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  $\lambda$  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)。

**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  $\Omega$  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界限时，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

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

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  $O(1)+O(2)+\dots+O(n)$  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。

### $\text{o}$ 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  $2n = O(n^2)$  却不是的。我们使用  $\text{o}$  记号来表示非渐近紧确的上界。 $\text{o}(g(n))$  的形式定义为集合

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

例如， $2n = \text{o}(n^2)$ ，但是  $2n^2 \neq \text{o}(n^2)$ 。

$O$  记号与  $\text{o}$  记号的定义是类似的。主要区别在于对  $f(n) = O(g(n))$ ，界  $0 \leq f(n) \leq cg(n)$  对某个常数  $c > 0$  成立；但对  $f(n) = \text{o}(g(n))$ ，界  $0 \leq f(n) \leq cg(n)$  对所有常数  $c > 0$  成立。从直觉上看，在  $\text{o}$  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

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \quad (3.1)$$

某些作者将这个极限作为  $\text{o}$  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。

### $\omega$ 记号

$\omega$  记号与  $\Omega$  记号的关系就好像  $\text{o}$  记号与  $O$  记号的关系一样。我们用  $\omega$  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

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

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

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

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

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 $f(n) = \Omega(h(n))$
$f(n) = \text{o}(g(n))$ 和 $g(n) = \text{o}(h(n))$	蕴含 $f(n) = \text{o}(h(n))$
$f(n) = \omega(g(n))$ 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：

$$f(n) = O(g(n)) \approx a \leq b$$

$$f(n) = \Omega(g(n)) \approx a \geq b$$

$$f(n) = \Theta(g(n)) \approx a = b$$

$$f(n) = o(g(n)) \approx a < b$$

$$f(n) = \omega(g(n)) \approx a > b$$

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  $\Theta$  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

$$(n+a)^b = \Theta(n^b) \quad (3.2)$$

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

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  $m \leq n$  蕴含  $f(m) \leq f(n)$ 。类似地，函数  $f(n)$  是单调递减

的，若  $m \leq n$  蕴含  $f(m) \geq f(n)$ 。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  $x$ ，

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 \quad (3.3)$$

对于任何整数  $n$ ，

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

对任意实数  $n \geq 0$  和整数  $a, b > 0$ ，

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil \quad (3.4)$$

$$\lfloor \lfloor n/a \rceil / b \rfloor = \lfloor n/ab \rfloor \quad (3.5)$$

$$\lceil a/b \rceil \leq (a + (b - 1))/b \quad (3.6)$$

$$\lfloor a/b \rceil \geq ((a - (b - 1))/b) \quad (3.7)$$

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：

$$a \bmod n = a - \lfloor a/n \rfloor n \quad (3.8)$$

**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  $a_0, a_1, \dots, a_d$  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  $a \geq 0$ ，函数  $n^a$  单调递增；对  $a \leq 0$ ，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  $a \geq 1$ ，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  $a > 1$ ，

$$\lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0 \quad (3.9)$$

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  $2.71828\cdots$ , 即自然对数函数的底, 对任意实数  $x$

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

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

$$e^x \geq 1 + x \quad (3.11)$$

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

$$1 + x \leq e^x \leq 1 + x + x^2 \quad (3.12)$$

当  $x \rightarrow 0$  时, 用  $1+x$  来近似  $e^x$  的效果相当好:

$$e^x = 1 + x + O(x^2)$$

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

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

## 对数

我们将用到下列记号:

$$\lg n = \log_2 n \quad (\text{以 } 2 \text{ 为底的对数})$$

$$\ln n = \log_e n \quad (\text{自然对数})$$

$$\lg^k n = (\lg n)^k \quad (\text{取幂})$$

$$\lg \lg n = \lg(\lg n) \quad (\text{复合})$$

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  $n$ ,

$$\begin{aligned} a &= b^{\log_b a}, \quad \log_c(ab) = \log_c a + \log_c b \\ \log_b a^n &= n \log_b a, \quad \log_b a = \frac{\log_c a}{\log_c b} \end{aligned} \quad (3.14) \quad [53]$$

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

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:

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

当  $x > -1$  时, 还有以下不等式成立:

$$\frac{x}{1+x} \leq \ln(1+x) \leq x \quad (3.16)$$

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:

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

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  $n \geq 0$ ，

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

54 因此， $n! = 1 \cdot 2 \cdot 3 \cdots n$ 。

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

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

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）

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

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

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

其中

$$\frac{1}{12n+1} < a_n < \frac{1}{12n} \quad (3.20)$$

### 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

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$$

多重函数是一种增长很慢的函数：

$$\lg^* 2 = 1 \quad \lg^* 4 = 2 \quad \lg^* 16 = 3 \quad \lg^* 65536 = 4 \quad \lg^*(2^{65536}) = 5$$

宇宙中可以观察到的原子数目估计约有  $10^{80}$ ，远远小于  $2^{65536}$ ，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：

$$F_0 = 0 \quad F_1 = 1 \quad F_i = F_{i-1} + F_{i-2} \quad \text{当 } i \geq 2 \quad (3.21)$$

因此每一个数都是前两个数的和，产生的序列为

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

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

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

特别地，有

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} \quad (3.23)$$

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  $\phi$  是黄金分割律， $\hat{\phi}$  是共轭数。

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

### 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  $k \geq d$ ，则  $p(n) = O(n^k)$ 。
- b) 若  $k \leq d$ ，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  $p(n) = \omega(n^k)$ 。

[57]

#### 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  $O$ ,  $o$ ,  $\Omega$ ,  $\omega$ , 或  $\Theta$  中的哪种关系。假设  $k \geq 1$ ,  $\epsilon > 0$ ,  $c > 1$  都是常数。在表格的空格内填入“是”或“否”即可。

	A	B	$O$	$o$	$\Omega$	$\omega$	$\Theta$
a)	$\lg^k n$	$n^\epsilon$					
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  $g_1, g_2, \dots, g_{30}$ ，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  $\Omega(g_i(n))$ 。

### 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  $\lg(g(n)) \geq 1$  且  $f(n) \geq 1$  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 $\Omega$ 的一些变形

某些作者定义  $\Omega$  的方式和我们略有不同，可以用  $\tilde{\Omega}$ （读做“ $\Omega$  无穷大”）来表示这种定义。若存在正常数  $c$  使  $f(n) \geq cg(n) \geq 0$  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  $\Omega$  代替  $\tilde{\Omega}$  时并不成立。

b) 请描述用  $\tilde{\Omega}$  代替  $\Omega$  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  $\Omega$ ，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  $O$ 。

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

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  $c \in \mathbb{R}$ , 定义叠函数  $f_c^*$  为

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

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。 $\Omega$  和  $\Theta$  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  $\Theta$  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。 60

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

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

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“ $\Theta$ ”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  $a \geq 1$ ,  $b > 1$ ,  $f(n)$  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：

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

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

$$T(n) = 2T(n/2) + \Theta(n) \quad (4.3)$$

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

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \quad (4.4)$$

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  $T(n) \leq cn \lg n$ ，其中  $c > 0$  是某个常数。先假设这个界对  $\lfloor n/2 \rfloor$  成立，即  $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$ 。对递归式作替换，得：

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

最后一步只要  $c \geq 1$  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  $T(n) \leq cn \lg n$  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  $T(n) \leq cn \lg n$  也就是  $T(1) \leq c1 \lg 1 = 0$ ，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  $n \geq n_0$ ，证明  $T(n) \leq cn \lg n$ ，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  $T(n) \leq cn \lg n$  正确性时，其中常量  $c \geq 1$ ，只要  $c$  取足够大的常数使  $T(2) \leq c2 \lg 2$  和  $T(3) \leq c3 \lg 3$  即可完成证明。实际上，选取任何  $c \geq 2$ ， $n=2$  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。

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  $T(n) \leq cn$ 。用所猜测的界对递归式作替换，得

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

由此引不出  $T(n) \leq cn$ , 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  $T(n) \leq cn - b$ ,  $b \geq 0$  是个常数。现在有

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

只要  $b \geq 1$ 。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  $T(n) \leq cn$  并证明

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

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  $T(n) \leq cn$ 。

### 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，

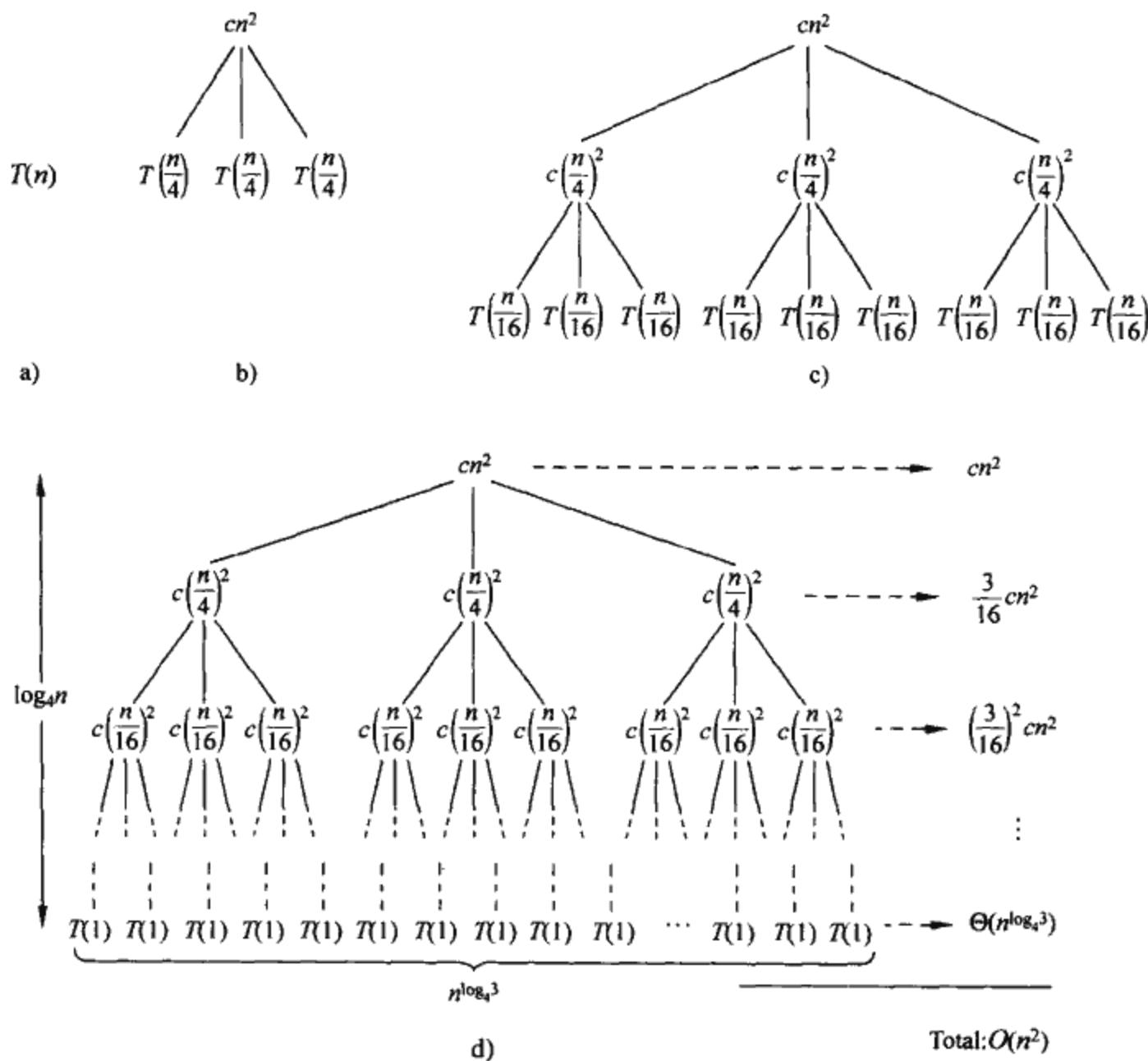


图 4-1 与递归式  $T(n) = 3T(n/4) + cn^2$  对应的递归树的构造。a) 示出了  $T(n)$ ，它在 b)~d) 中被不断地扩展而形成了递归树。d) 完全扩展了的递归树深度为  $\log_4 n$  (它有  $\log_4 n + 1$  层)

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 ( $0, 1, 2, \dots, \log_4 n$ )。

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  $i=0, 1, 2, \dots, \log_4 n - 1$ ，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：

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

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

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = O(n^2) \end{aligned}$$

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  $d > 0$ ， $T(n) \leq dn^2$  成立。适用与前面相同的常数  $c > 0$ ，有

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \leq 3d\lfloor n/4 \rfloor^2 + cn^2 \leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \leq dn^2 \end{aligned}$$

只要  $d \geq (16/13)c$ ，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  $n \rightarrow (2/3)n \rightarrow (2/3)^2 n \rightarrow \dots \rightarrow 1$ 。因为当  $k =$

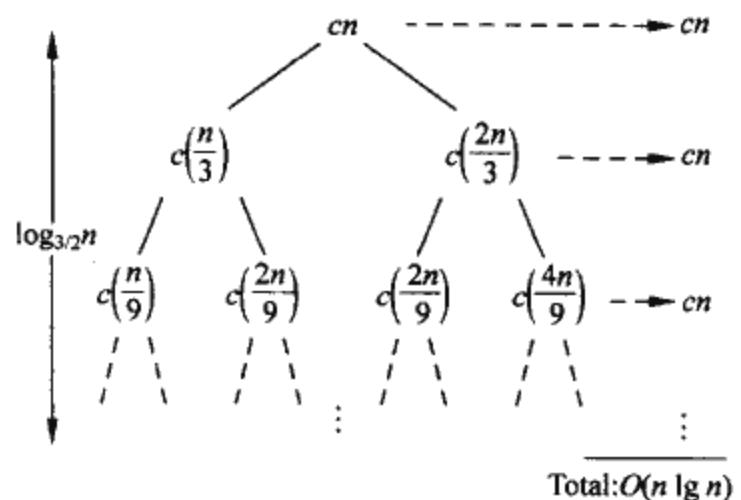


图 4-2 与递归式  $T(n) = T(n/3) + T(2n/3) + cn$  对应的递归树

$\log_{3/2} n$  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  $T(n) \leq dn \lg n$ , 当  $d$  是一个合适的正值常数, 有

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

71

上式成立的条件是  $d \geq c/(\lg 3 - 2/3)$ 。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  $a \geq 1$  且  $c > 0$  是常数。
- 4.2-5 利用递归树来找出递归式  $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$  的渐近紧确解, 其中  $\alpha$  是  $0 < \alpha < 1$  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:

$$T(n) = aT(n/b) + f(n) \quad (4.5)$$

其中  $a \geq 1$  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  $a \geq 1$  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  $\epsilon > 0$ , 有  $f(n) = O(n^{\log_b a - \epsilon})$ , 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  $\epsilon > 0$ , 有  $f(n) = \Omega(n^{\log_b a + \epsilon})$ , 且对常数  $c < 1$  与所有足够大的  $n$ , 有  $af(n/b) \leq c f(n)$ , 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  $\epsilon > 0$ ,  $f(n)$  必须渐近地小于  $n^{\log_b a}$ , 两者差一个因子  $n^\epsilon$ 。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  $af(n/b) \leq c f(n)$ 。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  $f(n) = O(n^{\log_3 9 - \epsilon})$ , 其中  $\epsilon = 1$ , 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  $a = 3$ ,  $b = 4$ ,  $f(n) = n \lg n$ ,  $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$ 。  
因为  $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ , 其中  $\epsilon \approx 0.2$ , 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  $n$ ,  $af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n)$ ,  $c = 3/4$ , 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  $f(n) = n \lg n$  漂近大于  $n^{\log_b a} = n$ , 但并不是多项式大于。对任意正常数  $\epsilon$ , 比值  $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$  漂近小于  $n^\epsilon$ 。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精确的漂近界:

- 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? 给出此递归式的漂近上界。

\*4.3-5 考虑在某个常数  $c < 1$  时的规则性条件  $af(n/b) \leq c f(n)$ , 此条件是主定理第三种情况的一部分。举一个常数  $a \geq 1$ ,  $b > 1$  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  $b > 1$  的整数幕上, 即  $n = 1, b, b^2, \dots$ 。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, 我们将用漂近记号来描述定义在  $b$  的整数幕上的函数的性态。这是有点“活用”漂近记号了。回忆一下, 漂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  $b$  的幕成立。因为可以构造出适合于集合  $\{b^i : i = 0, 1, \dots\}$ (而不是非负整数)上的漂近记号, 故这种“活用”是没有关系的。

要注意当在一个有限域上运用漂近记号时, 不能引起不合适的结论。例如, 证明当  $n$  是 2 的整数幕时,  $T(n) = O(n)$  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

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

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, 在有限域上用漂近记号时一定要有上下文说明。

### 4.4.1 取正合幕时的证明

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  $b > 1$  的正合幕, 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  $b$  的正合幕时主定理成立。

76

**引理 4.2** 设  $a \geq 1$ ,  $b > 1$  为常数,  $f(n)$  为定义在  $b$  的正合幕上的非负函数。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

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

**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。

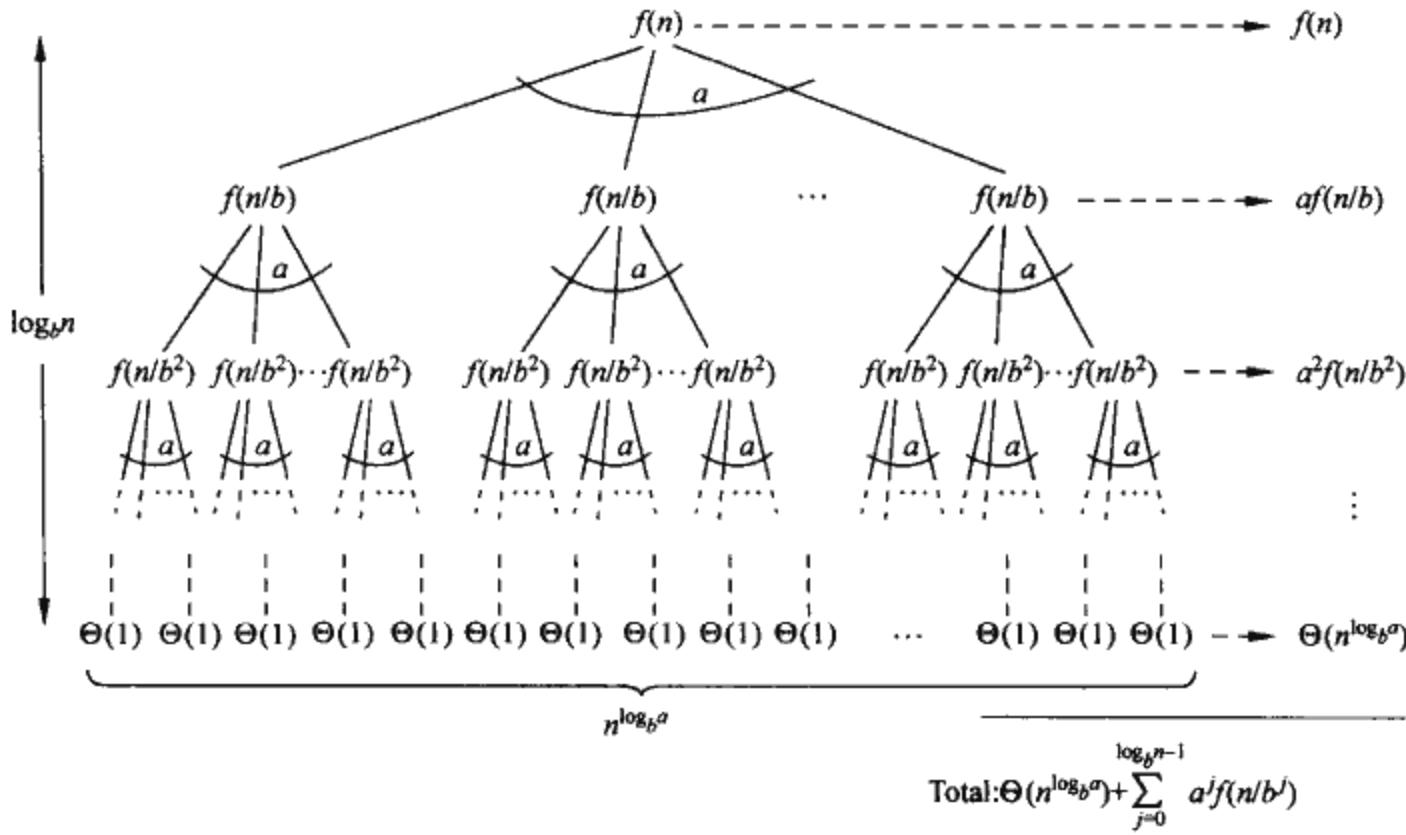


图 4-3 由递归式  $T(n)=aT(n/b)+f(n)$  生成的递归树。该树是一棵完全  $a$  叉树，共有  $n^{\log_b a}$  个叶结点，高度为  $\log_b n$ 。树中每一层的代价都在右边示出，其和由(4.6)式给出

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  $a \geq 1$ ,  $b > 1$  为常数,  $f(n)$  为定义在  $b$  的整数幕上的非负函数。函数  $g(n)$  由下式定义

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

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  $n \geq b$ ,  $a f(n/b) \leq c f(n)$ , 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

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) \quad (4.8) \quad [78]$$

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:

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

因为  $b$  与  $\epsilon$  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

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

对  $\Theta$  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  $n \geq b$ ,  $a f(n/b) \leq c f(n)$ , 有  $f(n/b) \leq (c/a) f(n)$ 。 $j$  次迭代后, 有  $f(n/b^j) \leq (c/a)^j f(n)$ , 或等价地, 有  $a^j f(n/b^j) \leq c^j f(n)$ 。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。

$$\begin{aligned} g(n) &= \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \leq \sum_{j=0}^{\log_b n - 1} c^j f(n) \leq f(n) \sum_{j=0}^{\infty} c^j \\ &= f(n) \left(\frac{1}{1-c}\right) = O(f(n)) \end{aligned}$$

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  $a \geq 1$ ,  $b > 1$  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  $f(n) = O(n^{\log_b a - \epsilon})$ ,  $\epsilon > 0$ , 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  $f(n) = \Omega(n^{\log_b a + \epsilon})$ ,  $\epsilon > 0$ , 且若对常数  $c < 1$ , 和所有足够大的  $n$ ,  $a f(n/b) \leq c f(n)$ , 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  $f(n) = \Omega(n^{\log_b a + \epsilon})$ 。■

#### 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

$$T(n) = aT(\lceil n/b \rceil) + f(n) \quad (4.10)$$

给出下界与对递归式

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \quad (4.11)$$

给出上界都是很容易的事, 因为由界  $\lceil n/b \rceil \geq n/b$  可通过情况 1) 得到所需结果, 界  $\lfloor n/b \rfloor \leq n/b$  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:

$$n, \lceil n/b \rceil, \lceil \lceil n/b \rceil/b \rceil, \lceil \lceil \lceil n/b \rceil/b \rceil/b \rceil, \dots$$

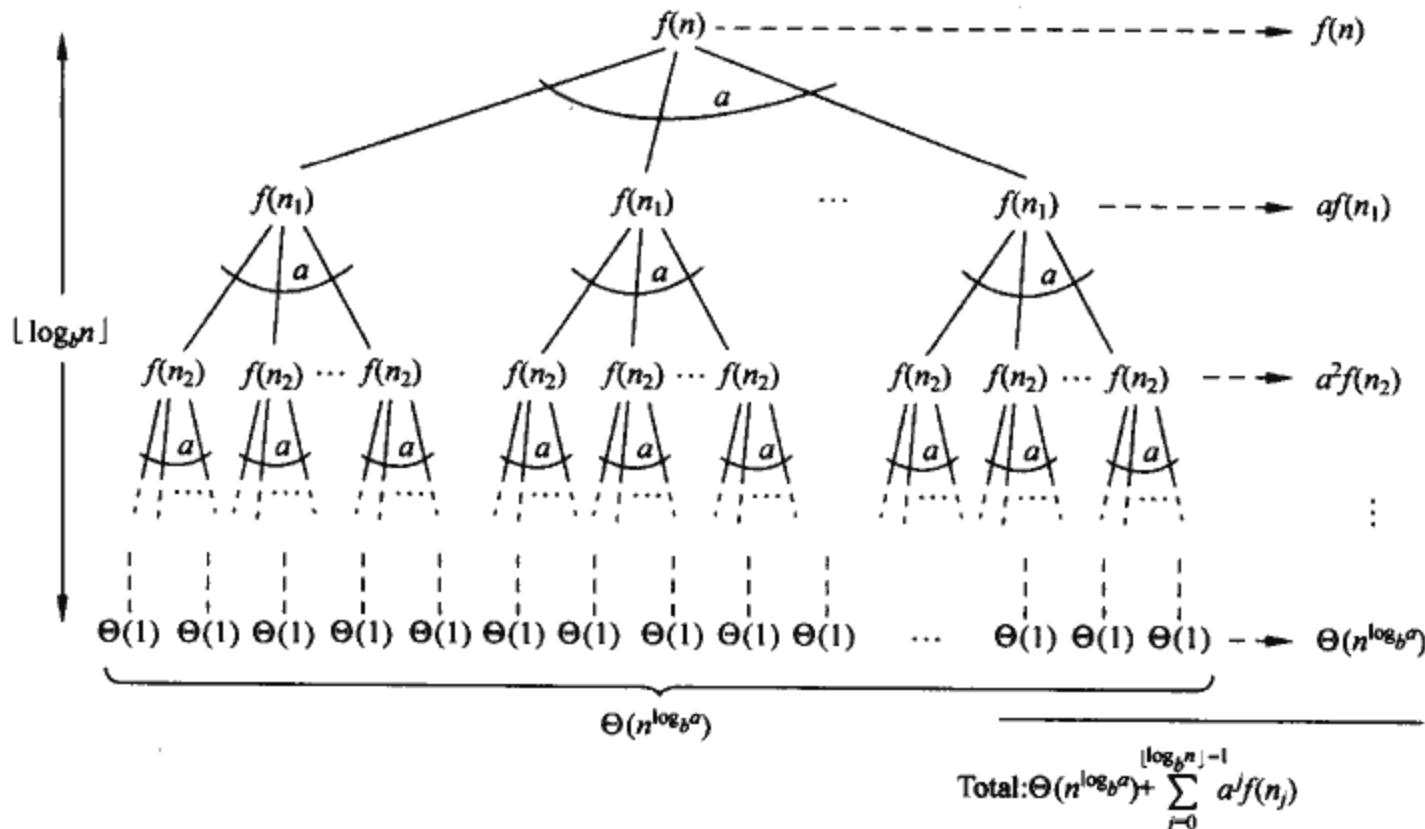


图 4-4 由  $T(n) = aT(\lceil n/b \rceil) + f(n)$  产生的递归树。递归参数  $n_j$  由(4.12)式给出

用  $n_j$  来表示序列中第  $j$  个元素，即

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

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  $\lceil x \rceil \leq x+1$  得：

$$n_0 \leq n, \quad n_1 \leq \frac{n}{b} + 1, \quad n_2 \leq \frac{n}{b^2} + \frac{1}{b} + 1, \quad n_3 \leq \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1, \quad \dots \quad [82]$$

一般地，

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

当  $j = \lfloor \log_b n \rfloor$  时，有

$$\begin{aligned} n_{\lfloor \log_b n \rfloor} &\leq \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} \leq \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} = \frac{n}{n/b} + \frac{b}{b-1} \\ &= b + \frac{b}{b-1} = O(1) \end{aligned}$$

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

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

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

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

先看情况 3)，如果对  $n > b + b/(b-1)$ ，有  $a f(\lceil n/b \rceil) \leq c f(n)$ ， $c < 1$  为常数，则  $a^j f(n_j) \leq c^j f(n)$ 。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  $j \leq \lfloor \log_b n \rfloor$  隐含着  $b^j/n \leq 1$ 。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

$$\begin{aligned} f(n_j) &\leq c \left( \frac{n}{b^j} + \frac{b}{b-1} \right)^{\log_b a} = c \left( \frac{n}{b^j} \left( 1 + \frac{b^j}{n} \cdot \frac{b}{b-1} \right) \right)^{\log_b a} \\ &= c \left( \frac{n^{\log_b a}}{a^j} \right) \left( 1 + \left( \frac{b^j}{n} \cdot \frac{b}{b-1} \right) \right)^{\log_b a} \\ &\leq c \left( \frac{n^{\log_b a}}{a^j} \right) \left( 1 + \frac{b}{b-1} \right)^{\log_b a} = O\left( \frac{n^{\log_b a}}{a^j} \right) \end{aligned}$$

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  $f(n_j) = O(n^{\log_b a - \epsilon})$ ，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  $k \geq 0$ ，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  $a f(n/b) \leq c f(n)$ ， $c < 1$  隐含着存在常数  $\epsilon > 0$  使  $f(n) =$

$\Omega(n^{\log_b a + \epsilon})$ 。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， $n \leq 2$ 。使所给出的界尽量精确。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，即使所传递的是一个  $N$  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，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。使所给出的界尽量精确，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

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

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， $\phi = \frac{1+\sqrt{5}}{2} = 1.61803\dots$ ,  $\hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803\dots$ 。

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  $i \geq 0$ ,  $F_{i+2} \geq \phi^i$ 。

#### 4-6 VLSI 芯片测试

Diogenes 教授有  $n$  个被认为是完全相同的 VLSI<sup>①</sup> 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

① 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  $l$ ,  $1 \leq i < k \leq m$  和  $1 \leq j < l \leq n$ , 有

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

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  $i=1, 2, \dots, m-1$  和  $j=1, 2, \dots, n-1$ , 有

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

(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  $f(1) \leq f(2) \leq \dots \leq f(m)$ 。

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

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

其中  $k \geq 1$ ; 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  $n \geq n_0$ , 有  $f(n/c) \geq d f(n)$ 。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

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

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。

## 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。

91  
HIRE-ASSISTANT( $n$ )  
1  $best \leftarrow 0$   $\triangleright$  candidate 0 is a least-qualified dummy candidate  
2 **for**  $i \leftarrow 1$  **to**  $n$   
3     **do** interview candidate  $i$   
4         **if** candidate  $i$  is better than candidate  $best$   
5             **then**  $best \leftarrow i$   
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。在任何情况下，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率度建立模型。

#### 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。92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。

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  $\langle rank(1), rank(2), \dots, rank(n) \rangle$  是序列  $\langle 1, 2, \dots, n \rangle$  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。

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。）93

## 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 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  $I\{A\}$  定义为

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

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：

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

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：

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

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  $X_A = I\{A\}$ ，则  $E[X_A] = \Pr\{A\}$ 。

**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

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

其中  $\bar{A}$  表示  $A$  的补  $S - A$ 。 ■

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  $X_i = I\{Y_i = H\}$ 。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

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

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：

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

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

$$E[X] = \sum_{x=1}^n x \Pr\{X = x\}$$

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  $E[X]$ ，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，

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

并且

$$X = X_1 + X_2 + \cdots + X_n \quad (5.3)$$

由引理 5.1，有

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

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

$$E[X_i] = 1/i \quad (5.4)$$

现在可以计算  $E[X]$  了：

$$E[X] = E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i] \quad (5.5)$$

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

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

97

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，正好雇用一次的概率是多少？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？
- 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  $\{1, 2, \dots, n\}$  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  $\langle \text{rank}(1), \text{rank}(2), \dots, \text{rank}(n) \rangle$ 。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  $A_3$ )。

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  $best \leftarrow 0$   $\triangleright$  candidate 0 is a least-qualified dummy candidate
3 for  $i \leftarrow 1$  to  $n$ 
4   do interview candidate  $i$ 
5   if candidate  $i$  is better than candidate  $best$ 
6     then  $best \leftarrow i$ 
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  $n \leftarrow length[A]$ 
2 for  $i \leftarrow 1$  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唯一。（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 可以产生输入的均匀随机排列。  
[101]

**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  $i=1, 2, \dots, n$ , 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

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

利用练习 C.2-6, 这个概率等于

$$\begin{aligned} & \Pr\{X_1\} \cdot \Pr\{X_2 | X_1\} \cdot \Pr\{X_3 | X_2 \cap X_1\} \cdot \Pr\{X_4 | X_3 \cap X_2 \cap X_1\} \\ & \cdots \Pr\{X_i | X_{i-1} \cap X_{i-2} \cap \dots \cap X_1\} \cdots \Pr\{X_n | X_{n-1} \cap \dots \cap X_1\} \end{aligned}$$

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  $\Pr\{X_2 | X_1\}=1/(n-1)$ , 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  $i=2, 3, \dots, n$ , 有  $\Pr\{X_i | X_{i-1} \cap X_{i-2} \cap \dots \cap X_1\}=1/(n-i+1)$ 。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

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

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  $\{1, 2, \dots, n\}$  的任何一个确定的排列  $\sigma=(\sigma(1), \sigma(2), \dots, \sigma(n))$ 。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   $n \leftarrow \text{length}[A]$ 
2  for  $i \leftarrow 1$  to  $n$ 
3    do swap  $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$ 
```

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 $\langle x_1, x_2, \dots, x_i \rangle$ 来表示其中的元素。这个排列包含一个 $(i-1)$ 排列 $\langle x_1, x_2, \dots, x_{i-1} \rangle$ ，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 $\langle x_1, x_2, \dots, x_i \rangle$ 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

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

概率 $\Pr\{E_2 | E_1\}$ 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

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

**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)]
```

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)
```

[103]

[104]

```

1  n  $\leftarrow \text{length}[A]$ 
2  for i  $\leftarrow 1$  to n
3    do swap A[i]  $\leftrightarrow$  A[\text{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  $\leftarrow \text{length}[A]$ 
2  offset  $\leftarrow \text{RANDOM}(1, n)$ 
3  for i  $\leftarrow 1$  to n
4    do dest  $\leftarrow i + \text{offset}$ 
5      if dest  $> n$ 
6        then dest  $\leftarrow \text{dest} - n$ 
7      B[dest]  $\leftarrow A[i]$ 
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。  
[105] 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。

## \*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, …, *k*，令 *b<sub>i</sub>* 表示 *i* 的生日，且  $1 \leq b_i \leq n$ 。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, …, *k*，和 *r*=1, 2, …, *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

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

更直观地来看，一旦选定 *b<sub>i</sub>* 后，*b<sub>j</sub>* 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

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

$$\Pr\{B_k\} = \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} \quad (5.8)$$

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

如果  $b_1, b_2, \dots, b_{k-1}$  互异，条件概率  $b_k \neq b_i$  ( $i=1, 2, \dots, k-1$ ) 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)， $1+x \leq e^x$ ，得

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

当  $-k(k-1)/2n \leq \ln(1/2)$  时成立。当  $k(k-1) \geq 2n \ln 2$  时或者解二次方程  $k \geq (1 + \sqrt{1 + (8 \ln 2)n})/2$ ， $k$  个人生日均不同的概率至少为  $1/2$ 。当  $n=365$  时，必须有  $k \geq 23$ 。因此，如果至少有 23 个人在一个房间里，那么至少有两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。[107]

### 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  $(i, j)$ ,  $1 \leq i < j \leq k$ , 定义指示器随机变量  $X_{ij}$  如下

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

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

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

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

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

因此当  $k(k-1) \geq 2n$  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

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

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  $i, i+1, \dots, i+k-1$  得到的都是正面，此处  $1 \leq k \leq n$  且  $1 \leq i \leq n-k+1$ 。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

$$\Pr\{A_{ik}\} = 1/2^k \quad (5.9)$$

对  $k=2 \lceil \lg n \rceil$

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

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 [110]

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

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  $j=0, 1, 2, \dots, n$ ，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，

$$E[L] = \sum_{j=0}^n j \Pr\{L_j\} \quad (5.11)$$

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  $j \geq 2\lceil \lg n \rceil$  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  $j=0, 1, 2, \dots, n$ ，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  $\sum_{j=0}^{2\lceil \lg n \rceil-1} \Pr\{L_j\} \leq 1$ 。于是，有：

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

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  $r \geq 1$ ，由  $r\lceil \lg n \rceil$  个正面组成的序列开始于位置  $i$  的概率为

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

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  $1/n^2 = 1/1000000$ 。

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  $\lfloor n/(\lg n)/2 \rfloor$  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

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

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

$$\begin{aligned} (1 - 1/\sqrt{n})^{\lfloor n / \lfloor (\lg n)/2 \rfloor \rfloor} &\leq (1 - 1/\sqrt{n})^{n / \lfloor (\lg n)/2 \rfloor - 1} \leq (1 - 1/\sqrt{n})^{2n / \lg n - 1} \\ &\leq e^{-(2n / \lg n - 1) / \sqrt{n}} = O(e^{-\lg n}) = O(1/n) \end{aligned}$$

112

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

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

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

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：

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

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  $X_{ik} = I\{A_{ik}\}$  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

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

113

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：

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

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  $E[X] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$ ，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  $bestscore \leftarrow -\infty$ 
2 for  $i \leftarrow 1$  to  $k$ 
3   do if  $score(i) > bestscore$ 
4     then  $bestscore \leftarrow score(i)$ 
5 for  $i \leftarrow k+1$  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  $M(j) = \max_{1 \leq i \leq j} \{score(i)\}$  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  $\Pr\{S_i\} = 0, i=1, 2, \dots, k$ 。于是得到

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

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  $k+1 \leq j \leq i-1$  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

$$\int_k^n \frac{1}{x} dx \leq \sum_{i=k}^{n-1} \frac{1}{i} \leq \int_{k-1}^{n-1} \frac{1}{x} dx$$

解这些定积分可以得到下面的界：

$$\frac{k}{n}(\ln n - \ln k) \leq \Pr\{S\} \leq \frac{k}{n}(\ln(n-1) - \ln(k-1))$$

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

$$\frac{1}{n}(\ln n - \ln k - 1)$$

令此导数等于 0，我们看到当  $\ln k = \ln n - 1 = \ln(n/e)$ ，或等价地，当  $k = n/e$  时，概率的下界最大化。因此，如果用  $k = n/e$  来实现我们的策略，则可以以至少为  $1/e$  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  $1/n$ 。

## 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  $i=0, 1, \dots, 2^b - 1$ ，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  $i \geq 0$ ，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  $i \geq 0$ ,  $n_i = 100i$ 。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  $k \geq 1$  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  $A[1], A[2], A[3], \dots, A[n]$ ，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) 假定刚好有一个下标  $i$  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  $k \geq 1$  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  $k \geq 1$  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、Megiddo 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

第 1 章

### 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  $\langle a_1, a_2, \dots, a_n \rangle$ 。

输出：输入序列的一个重排  $\langle a'_1, a'_2, \dots, a'_n \rangle$ ，使  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ 。

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，还要用它实现优先级队列。  
124

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  $\{0, 1, 2, \dots, k\}$ 。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  $\{0, 1, 2, \dots, k\}$ ，则基数排序就可以在  $\Theta(d(n+k))$  时间内完成排序。当  $d$  是一个常数， $k$  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  $i$ th 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。  
125  
126

## 第6章 堆 排 序

本章要介绍另一种排序算法，即堆排序(heap sort)。像合并排序而不像插入排序，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：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： $\text{length}[A]$ 是数组中的元素个数， $\text{heap-size}[A]$ 是存放在A中的堆的元素个数。就是说，虽然 $A[1.. \text{length}[A]]$ 中都可以包含有效值，但 $A[\text{heap-size}[A].. \text{length}[A]]$ 之后的元素都不属于相应的堆，此处 $\text{heap-size}[A] \leq \text{length}[A]$ 。树的根为 $A[1]$ ，给定了某个结点的下标*i*，其父结点 $\text{PARENT}(i)$ 、左儿子 $\text{LEFT}(i)$ 和右儿子 $\text{RIGHT}(i)$ 的下标可以简单地计算出来：

$\text{PARENT}(i)$

return  $\lfloor i/2 \rfloor$

$\text{LEFT}(i)$

return  $2i$

$\text{RIGHT}(i)$

return  $2i+1$

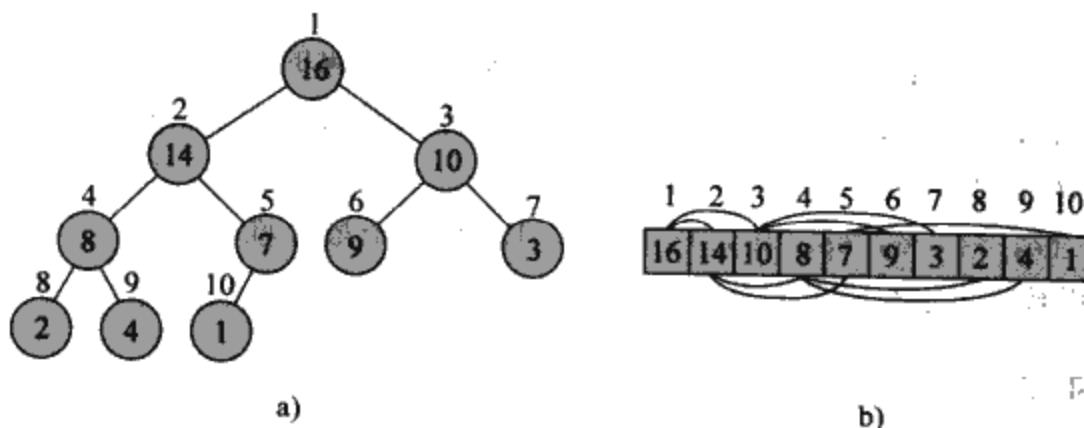


图6-1 一个最大堆(大根堆)可被看作a)一棵二叉树和b)一个数组。圆圈中的数字表示树中每个结点存储的值，结点上方的数字表示对应的数组下标。数组上下的连线表示父子关系，且父结点总在子结点的左边。这棵树的高度为3，存储值为8的4号结点的高度为1

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

$$A[\text{PARENT}(i)] \geq A[i]$$

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

$$A[\text{PARENT}(i)] \leq A[i]$$

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  $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$  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  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ 。

## 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  $\text{LEFT}(i)$  和  $\text{RIGHT}(i)$  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   $l \leftarrow \text{LEFT}(i)$ 
2   $r \leftarrow \text{RIGHT}(i)$ 
3  if  $l \leq \text{heap-size}[A]$  and  $A[l] > A[i]$ 
4    then  $\text{largest} \leftarrow l$ 
5  else  $\text{largest} \leftarrow i$ 
6  if  $r \leq \text{heap-size}[A]$  and  $A[r] > A[\text{largest}]$ 
7    then  $\text{largest} \leftarrow r$ 
8  if  $\text{largest} \neq i$ 
9    then exchange  $A[i] \leftrightarrow A[\text{largest}]$ 
10   MAX-HEAPIFY(A,  $\text{largest}$ )
```

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  $\text{largest}$  中。如果  $A[i]$  是最大的，则以  $i$  为根的子树已是最大堆，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  $A[\text{largest}]$ ，从而使  $i$  及其子女满足堆性质。下标为  $\text{largest}$  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。130

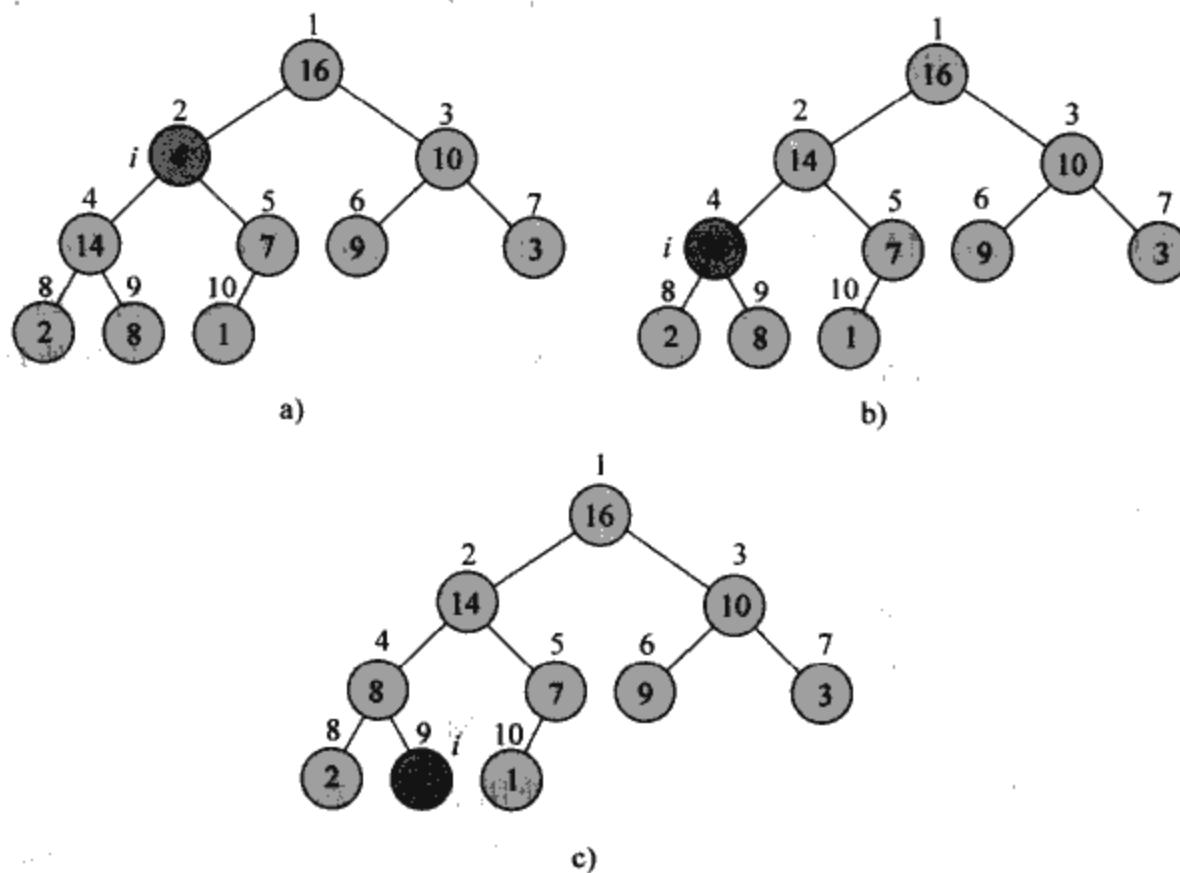


图 6-2 当  $\text{heap-size}[A]=10$  时，MAX-HEAPIFY( $A, 2$ )的作用过程。a) 初始构造，在结点  $i=2$  处  $A[2]$  违反了最大堆性质，因为它不大于它的两个子女。在 b) 中通过交换  $A[2]$  与  $A[4]$ ，在结点 2 处恢复了最大堆性质，但又在结点 4 处违反了最大堆性质。现在递归调用 MAX-HEAPIFY( $A, 4$ )，置  $i=4$ ，c) 中交换了  $A[4]$  和  $A[9]$ ，结点 4 的最大堆性质得到恢复，递归调用 MAX-HEAPIFY( $A, 9$ ) 对该数据结构不会再引起任何变化

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

$$T(n) \leq T(2n/3) + \Theta(1)$$

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  $i > \text{heap-size}[A]/2$ ，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  $n = \text{length}[A]$ )变成一个最大堆。132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  $\text{heap-size}[A] \leftarrow \text{length}[A]$ 
2 for  $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$  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  $i+1, i+2, \dots, n$  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  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$  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  $i+1, i+2, \dots, n$  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。

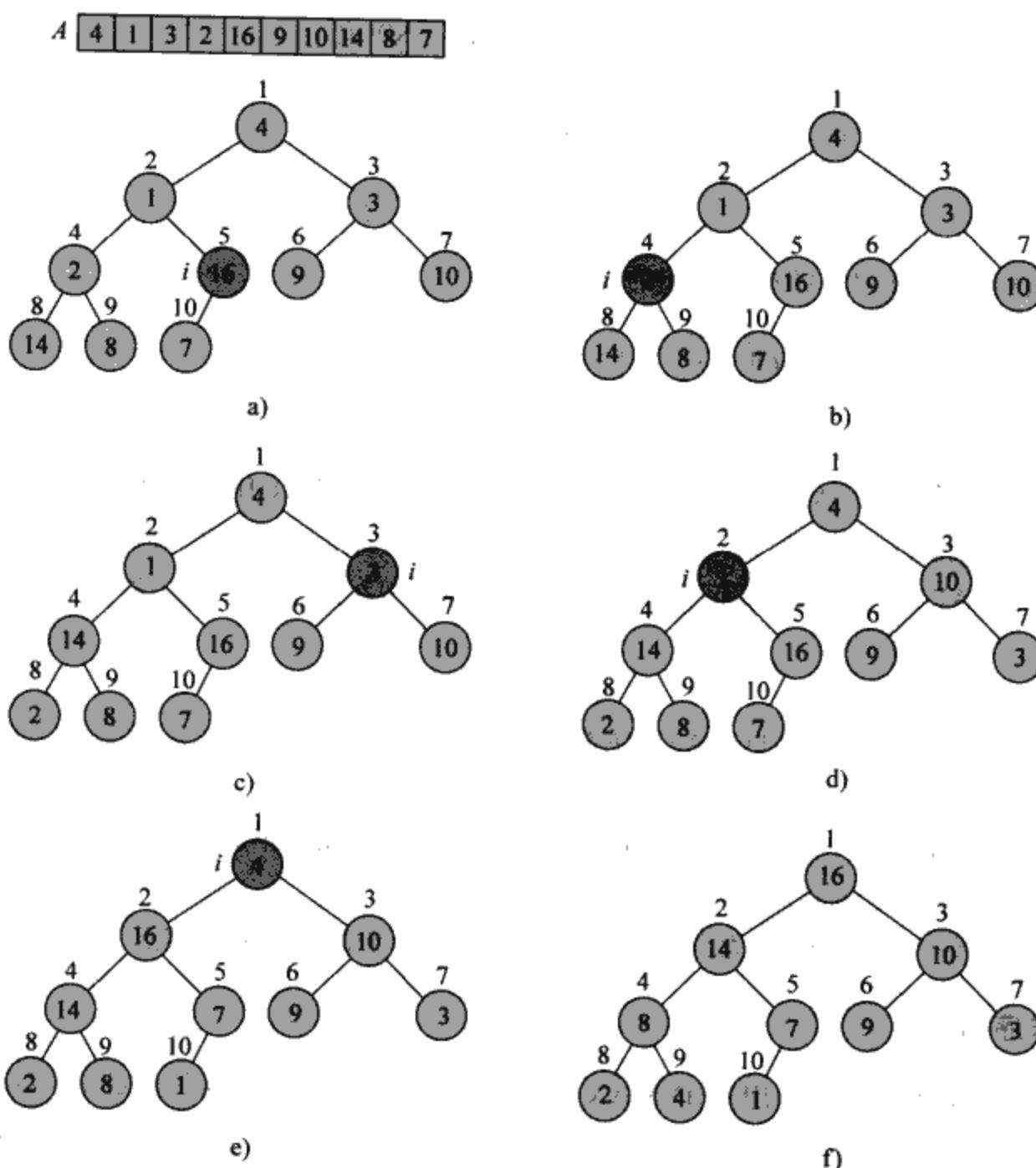


图 6-3 BUILD-MAX-HEAP 的执行过程，图中示出了在 BUILD-MAX-HEAP 的第 3 行调用 MAX-HEAPIFY 之前的数据结构。a)一个包含 10 个元素的输入数组 A 及其所表示的二叉树。图中示出了调用 MAX-HEAPIFY(A, i) 之前循环下标 i 指向结点 5。b)结果所得的数据结构。循环下标 i 在下一轮执行中指向结点 4。c)~e) BUILD-MAX-HEAP 中 for 循环的后续执行过程。注意当对某结点调用 MAX-HEAPIFY 时，该结点的两棵子树都已是最大堆。f) BUILD-MAX-HEAP 执行完毕后的最大堆

**终止：**过程终止时， $i=0$ 。根据循环不变式，我们知道结点  $1, 2, \dots, n$  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：

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

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  $\lfloor \text{length}[A]/2 \rfloor$  降到 1，而不是从 1 升到  $\lfloor \text{length}[A]/2 \rfloor$ ？
- 6.3-3 证明：在任一含  $n$  个元素的堆中，至多有  $\lfloor n/2^{h+1} \rfloor$  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  $n = \text{length}[A]$ ）构成一个最大堆。[135]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罝。现在，如果从堆中“去掉”结点  $n$ （通过减小  $\text{heap-size}[A]$ ），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  $O(\lg n)$ 。

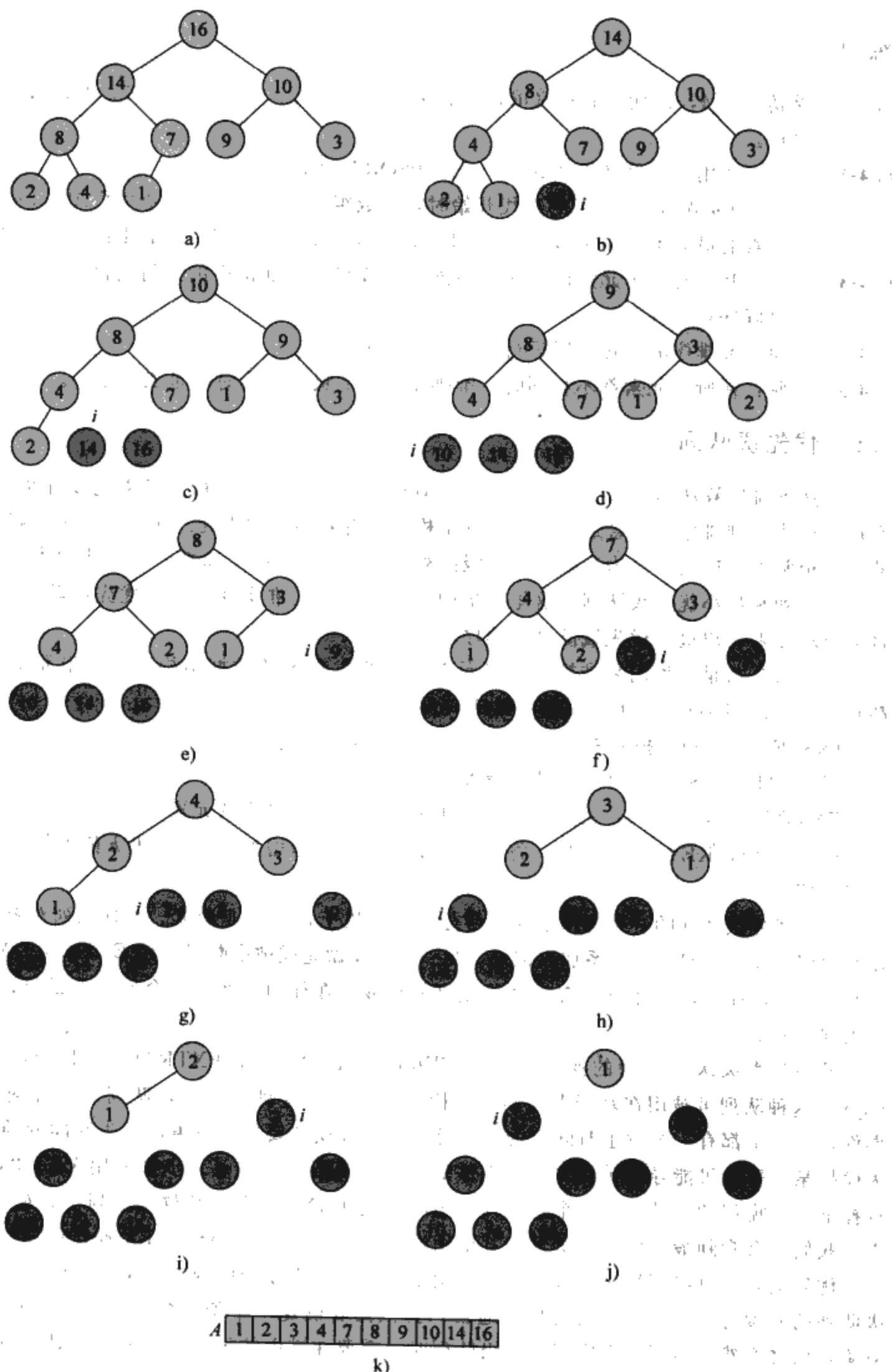


图 6-4 HEAPSORT 的操作过程。a)用 BUILD-MAX-HEAP 构造所得的最大堆的数据结构。b)~j)每次在第 5 行调用 MAX-HEAPIFY 后的最大堆，同时示出  $i$  的值。仅浅阴影结点仍然留在堆中。k)结果的排序数组 A。

## 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。

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: 最大优先级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)

139

```
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)
```

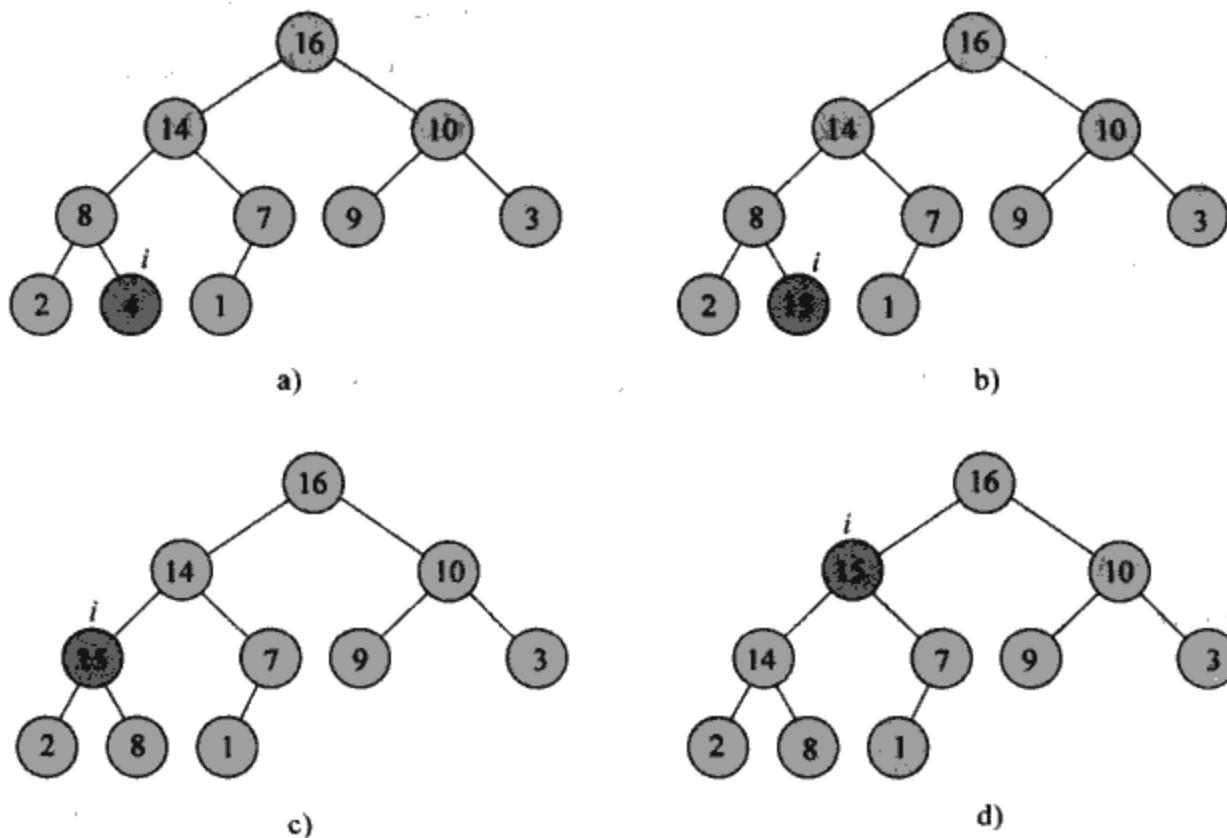


图 6-5 HEAP-INCREASE-KEY 的操作过程。a) 图 6.4a 中的最大堆，其中下标为  $i$  的结点以深阴影表示。b) 该结点将关键字的值增大到 15。c) 经过第 4~6 行的 while 循环的一次迭代，该结点与其父结点交换了关键字，而且下标  $i$  上移到父亲结点。d) while 循环的又一次迭代后的最大堆。此时， $A[\text{PARENT}(i)] \geq A[i]$ ，最大堆性质成立，程序终止。

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。
- 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  $A[1.. \text{heap-size}[A]]$  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  $i \leftarrow 2$  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 $\infty$ 数据项，表示不存在的元素。所以，Young氏矩阵可以用来存放 $r \leq mn$ 个有限的数。

- a)画一个包含元素{9, 16, 3, 2, 4, 8, 5, 14, 12}的 $4 \times 4$ 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  $\{1, 2, \dots, C\}$ 。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  $1, 2, \dots, C$  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  $O(\lg^{1/3+\epsilon} C)$  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  $O(\min(\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n))$ ，其中任意固定值  $\epsilon > 0$ 。这些结论的更多细节讨论可以在 Raman [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。

145

下面的过程实现快速排序：

```
QUICKSORT( $A, p, r$ )
1  if  $p < r$ 
2    then  $q \leftarrow \text{PARTITION}(A, p, r)$ 
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  $\text{QUICKSORT}(A, 1, \text{length}[A])$ 。

### 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   $x \leftarrow A[r]$ 
2   $i \leftarrow p-1$ 
3  for  $j \leftarrow p$  to  $r-1$ 
4    do if  $A[j] \leq x$ 
5      then  $i \leftarrow i+1$ 
6          exchange  $A[i] \leftrightarrow A[j]$ 
7  exchange  $A[i+1] \leftrightarrow A[r]$ 
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：

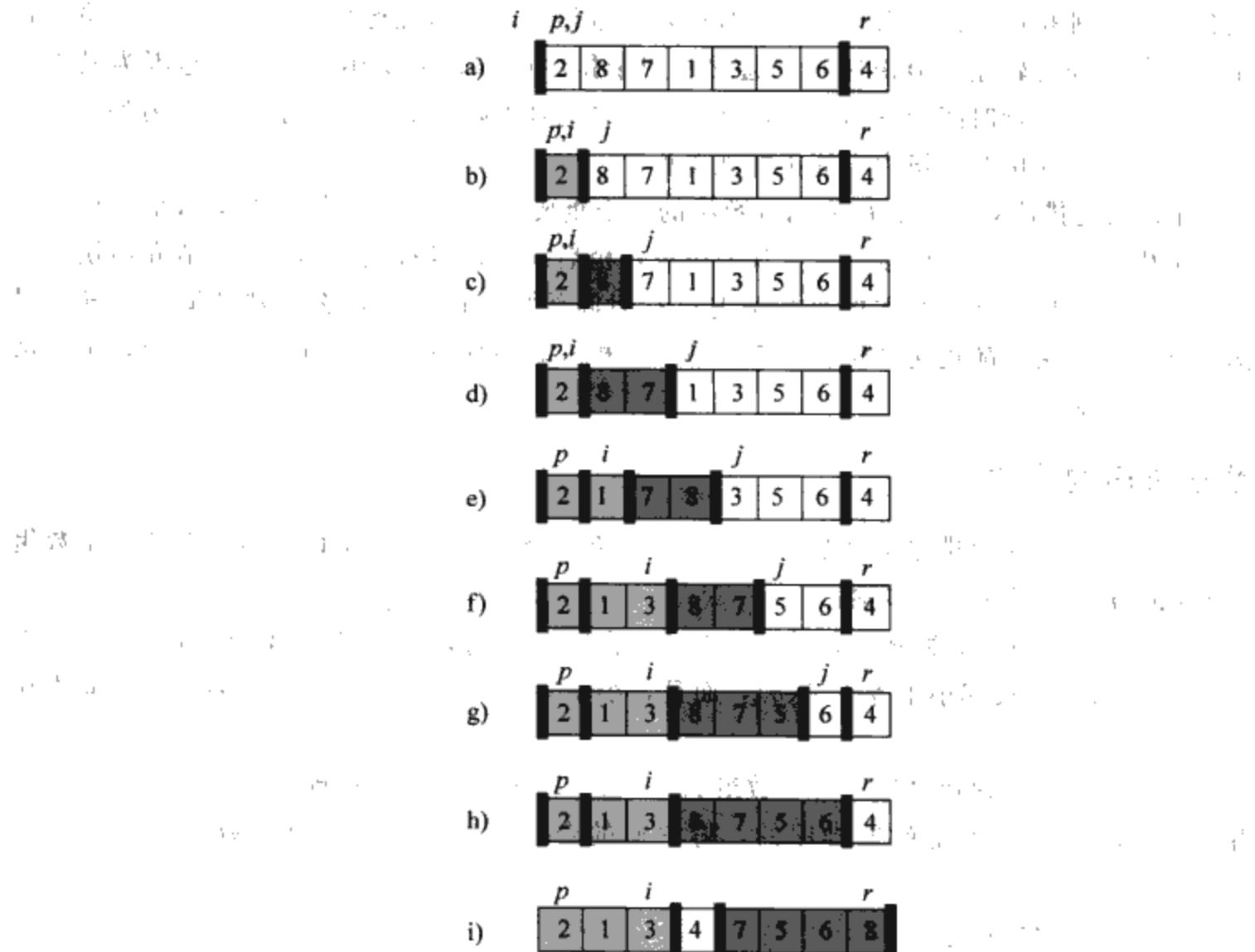


图 7-1 PARTITION 在一个样例数组上的运行过程。浅阴影部分的数组元素都在划分的第一部分，其值都不大于  $x$ 。深阴影的元素在划分第二部分，其值都大于  $x$ 。无阴影的元素尚未被放入前两个部分中的任何一个，白色元素为主元。a) 初始的数组和变量设置。数组元素均未被放入前两个部分中的任何一个；b) 数值 2 被“与它自身交换”，并被放入了元素值较小的那个部分；c)~d) 数值 8 和 7 被加到具有较大值的那个部分中；e) 数值 1 和 8 交换，较小的部分增加；f) 数值 3 和 8 交换，较小的部分增加；g)~h) 较大的部分增加，包含 5 和 6，循环结束；i) 在第 7~8 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  $p \leq k \leq i$ ，则  $A[k] \leq x$ 。
- 2) 如果  $i+1 \leq k \leq j-1$ ，则  $A[k] > x$ 。
- 3) 如果  $k=r$ ，则  $A[k]=x$ 。

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



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。

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  $A[j] \leq x$  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  $A[i] \leq x$ ，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。

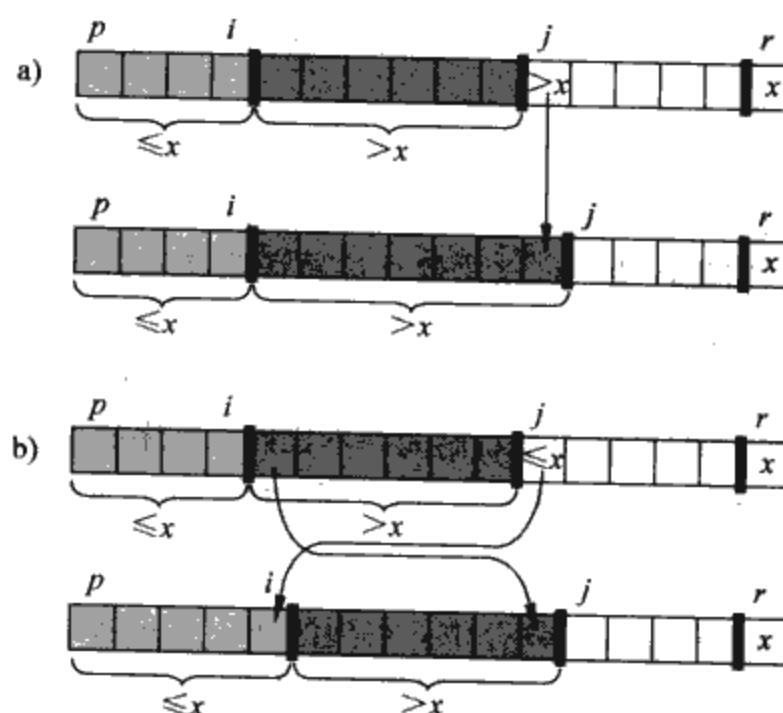


图 7-3 过程 PARTITION 的一次迭代中可能发生的两种情况。a) 如果  $A[j] > x$ ，需要做的唯一操作就是使  $j$  增加 1，从而使循环不变式保持成立；b) 如果  $A[j] \leq x$ ，则将下标  $i$  增加 1，并交换  $A[i]$  和  $A[j]$ ，再使  $j$  增加 1。此时，循环不变式仍然保持成立

**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  
l  
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

$$T(n) \leq 2T(n/2) + \Theta(n)$$

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画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

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

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。  
150  
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  $O(n \lg n)$ 。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  $O(n \lg n)$ 。

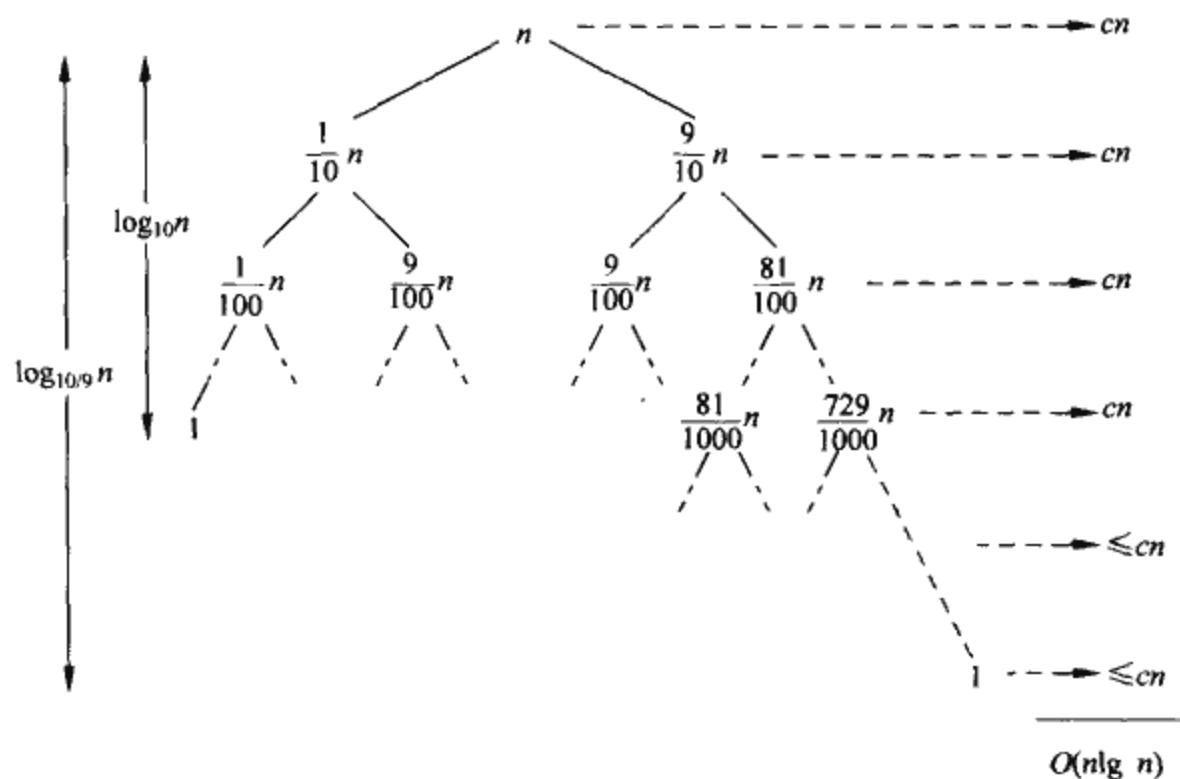


图 7-4 QUICKSORT 的一棵递归树，其中 PARTITION 总是产生 9：1 的划分，总的运行时间为  $O(n \lg n)$ 。各结点中示出了子问题的规模，每一层的代价在右边显示。每一层的代价包含了  $\Theta(n)$  项中所隐含的常数  $c$

### 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。151

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。

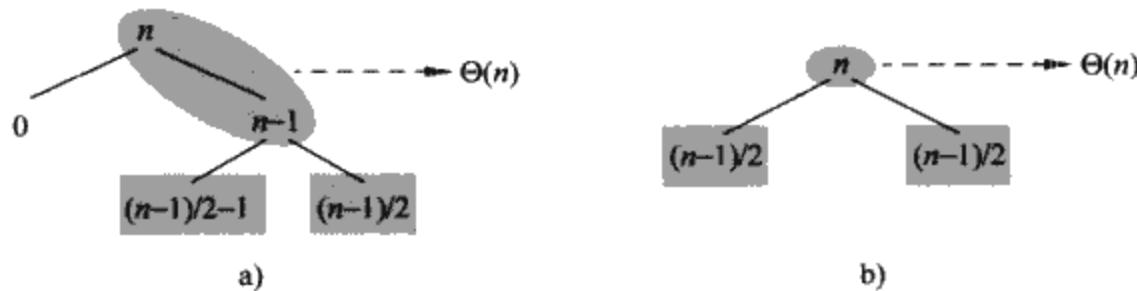


图 7-5 a) 快速排序的递归树中的两层。树根处划分的代价为  $n$ , 且得到是“很差的”划分: 两个子数组的大小分别为 0 和  $n-1$ 。对大小为  $n-1$  的子数组的划分代价为  $n-1$ , 产生一个“较好的”划分: 两个子数组的大小分别为  $(n-1)/2-1$  和  $(n-1)/2$ ; b) 一棵非常平衡的递归树中的一层。在它的两个部分中, 子问题的划分代价以椭圆形阴影示出, 为  $\Theta(n)$ 。但是, a) 图中矩形阴影示出的、待解决的子问题并不大于 b) 图中对应的、待解决的子问题

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  $0 < \alpha \leq 1/2$  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  $0 < \alpha \leq 1/2$ ，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

153

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。

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  $p, \dots, r$  这一范围内随机取样的，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  $i \leftarrow \text{RANDOM}(p, r)$ 
2 exchange  $A[r] \leftrightarrow A[i]$ 
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  $q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$ 
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  $\Theta$  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：

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n) \quad (7.1)$$

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  $T(n) \leq cn^2$  成立， $c$  为某个常数。将此式代入递归式(7.1)，得：

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) + \Theta(n) \end{aligned}$$

表达式  $q^2 + (n-q-1)^2$  在参数的取值区间  $0 \leq q \leq n-1$  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  $\max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) \leq (n-1)^2 = n^2 - 2n + 1$ 。对  $T(n)$  就有：

$$T(n) \leq cn^2 - c(2n-1) + \Theta(n) \leq cn^2$$

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  $z_1, z_2, \dots, z_n$ , 其中  $z_i$  是数组 A 中第  $i$  个最小的元素。此外, 我们还定义  $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$  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? 这回答这个问题, 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

$$X_{ij} = I\{z_i \text{ 与 } z_j \text{ 进行比较}\}$$

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:

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\} \quad (7.2)$$

在上式中,  $\Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\}$  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  $z_i$  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  $z_j$ 。[157]

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:

$$\begin{aligned}\Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\} &= \Pr\{z_i \text{ 或 } z_j \text{ 是从 } Z_{ij} \text{ 中首先选出的主元}\} \\ &= \Pr\{z_i \text{ 是从 } Z_{ij} \text{ 中首先选出的主元}\} \\ &\quad + \Pr\{z_j \text{ 是从 } Z_{ij} \text{ 中首先选出的主元}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}\end{aligned}\tag{7.3}$$

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:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1}$$

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:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n)\tag{7.4}$$

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:

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

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  $q=0, 1, \dots, n-1$  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  $\alpha$  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  
设计的划分算法：

```

HOARE-PARTITION( $A, p, r$ )
1    $x \leftarrow A[p]$ 
2    $i \leftarrow p-1$ 
3    $j \leftarrow r+1$ 
4   while TRUE
5     do repeat  $j \leftarrow j-1$ 
6       until  $A[j] \leq x$ 
7       repeat  $i \leftarrow i+1$ 
8         until  $A[i] \geq x$ 
9       if  $i < j$ 
10      then exchange  $A[i] \leftrightarrow A[j]$ 
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  $p \leq j < r$ 。  
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  $p \leq j < r$ ，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  $1/n$ 。利用这个结论来定义指示器随机变量  $X_i = I\{\text{第 } i \text{ 个最小的元素被选作主元}\}$ 。 $E[X_i]$  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：

$$E[T(n)] = E\left[\sum_{q=1}^n X_q(T(q-1) + T(n-q) + \Theta(n))\right] \quad (7.5)$$

c) 证明公式(7.5)可以简化成

$$E[T(n)] = \frac{2}{n} \sum_{q=0}^{n-1} E[T(q)] + \Theta(n) \quad (7.6)$$

d) 证明：

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \quad (7.7)$$

(提示：将该和式划分成两个部分，一个针对  $k=1, 2, \dots, \lceil n/2 \rceil - 1$ ，另一个针对  $k=\lceil n/2 \rceil, \dots, n-1$ 。)

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  $E[T(n)] \leq an \lg n - bn$ ）

### 7-3 Stooge 排序

Howard、Fine 等教授提出了下面的“漂亮的”排序算法：

```

STOOGESORT(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  STOOGESORT(A, i, j-k)      ▷ First two-thirds.
7  STOOGESORT(A, i+k, j)      ▷ Last two-thirds.
8  STOOGESORT(A, i, j-k)      ▷ First two-thirds again.

```

a) 证明：如果  $n = \text{length}[A]$ ，那么 STOOGESORT(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 STOOGESORT 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精确的渐近( $\Theta$ 记号)界。[161]

c) 比较 STOOGESORT 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

```