

TURING

图灵计算机科学丛书

Foundations Of Algorithms Fifth Edition

# 算法基础 (第5版)

[美] Richard E. Neapolitan 著  
贾洪峰 译



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS

TURING

图灵计算机科学丛书

Foundations Of Algorithms Fifth Edition

# 算法基础（第5版）

[美] Richard E. Neapolitan 著  
贾洪峰 译

人民邮电出版社  
北京

## 图书在版编目 (C I P ) 数据

算法基础 : 第5版 / (美) 那不勒坦

(Neapolitan, R. E.) 著 ; 贾洪峰译. — 北京 : 人民邮电出版社, 2016. 3

(图灵计算机科学丛书)

ISBN 978-7-115-41657-5

I. ①算… II. ①那… ②贾… III. ①电子计算机—  
算法理论 IV. ①TP301. 6

中国版本图书馆CIP数据核字(2016)第023362号

## 内 容 提 要

本书通过大量示例介绍了算法设计、算法的复杂度分析以及计算复杂度。主要内容有：算法设计与分析、分而治之方法、动态规划方法、贪婪方法、回溯算法、分支定界算法、计算复杂度、难解性和NP理论、遗传算法和遗传编程、数论算法、并行算法等。此外，本书在每章末尾都提供了大量练习，而且还提供了全面的教辅材料及答案，是教授和学习算法设计与分析的理想教材。

本书适合高等院校学生、程序员及算法分析和设计人员。

- 
- ◆ 著 [美] Richard E. Neapolitan
  - 译 贾洪峰
  - 责任编辑 岳新欣
  - 执行编辑 李舒扬
  - 责任印制 杨林杰
  - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
  - 邮编 100164 电子邮件 315@ptpress.com.cn
  - 网址 <http://www.ptpress.com.cn>
  - 三河市海波印务有限公司印刷
  - ◆ 开本: 880×1230 1/16
  - 印张: 25.5
  - 字数: 775千字 2016年3月第1版
  - 印数: 1-3 000册 2016年3月河北第1次印刷
  - 著作权合同登记号 图字: 01-2015-1434号
- 

定价: 99.00元

读者服务热线: (010)51095186转600 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京东工商广字第 8052 号

# 版 权 声 明

Original English language edition published by Jones & Bartlett Learning, LLC. 5 Wall Street, Burlington, MA 01803. *Foundations Of Algorithms, Fifth Edition* by Richard E. Neapolitan. Copyrights © 2015 by Jones & Bartlett Learning, LLC. All Rights Reserved.

Simplified Chinese Edition Copyrights © 2016 by Posts & Telecom Press.

本书中文简体字版由 JONES & BARTLETT LEARNING, LLC 授权人民邮电出版社独家出版。未得书面许可，本书的任何部分和全部不得以任何形式重制。

版权所有，侵权必究。

纪念我的朋友 Jack，他让生活变得充满乐趣！

Richard E. Neapolitan

# 前　　言

本书这一版保留了之前各版赖以成功的特点。和之前的版本一样，这一版仍然使用伪代码，而非真正的C++代码。在呈现复杂算法时，无论使用何种程序设计语言，如果毫无节制地运用该语言的所有细节，都只会模糊学生对算法的理解。此外，伪代码应当能让精通任意高级编程语言的人们都可以看懂，也就是说，它应当尽可能避免使用某种语言特有的细节。1.1节中讨论了明显与C++不一致的内容。本书讨论的是算法设计、算法的复杂度分析和计算复杂度（对问题的分析），没有涉及其他类型的分析，比如正确性分析。之所以要编写本书，是因为我找不出一本教科书，既严密准确地讨论了算法的复杂度分析，又能让主流大学（比如东北伊利诺伊大学）计算机科学专业的学生读懂。东北伊利诺伊大学的大多数学生都还没有学习微积分，这就意味着他们不熟悉抽象数学和数学符号。就我所知，现有教科书中都采用了一些数学符号，这对于精通数学的学生来说完全没有问题，但对东北伊利诺伊大学的多数学生来说，就显得有些过于简练而难以理解了。

为使本书更易于理解，我采取了以下做法：

- 假定学生的数学背景知识仅包括大学代数和离散结构；
- 更多地使用日常语言来解释数学概念；
- 在正式证明中给出更多细节；
- 提供大量范例。

本教科书是为高年级本科生或研究生为时一学期的“算法设计与分析”课程编写的，旨在让学生基本掌握编写和分析算法的方法，并向他们传授一些运用标准的算法设计策略来编写算法的必要技能。过去，这些策略包括分而治之、动态规划、贪婪方法、回溯和分支定界。但近年来，遗传算法的应用对计算机科学家来说越来越重要。而学生只有在学习人工智能的相关课程时才可能接触到此类算法。但并没有什么本质性的特征将遗传算法归入人工智能领域。因此，为了更全面地向学生提供当前流行的有用方法，这一版增加了一章内容，讨论遗传算法和遗传编程。

绝大多数复杂度分析都只需要有限数学的知识，所以在大多数讨论中，可以假定读者只有大学代数和离散结构的背景知识。也就是说，对于大多数内容来说，并不需要依靠那些只会在微积分课上学到的概念。没有微积分背景知识的学生通常会对数学符号感到不适应。因此，我会尽量使用日常语言来介绍数学概念（如“大O”符号），减少数学符号的使用。要在两者之间找到一个最佳平衡点并不那么容易；为使表述清晰，有必要使用一定数量的符号，但使用过多，又会让许多学生感到困惑。根据学生们的反馈情况，我找到了一个合适的平衡点。

这并不是说我不忠于数学的严格性。我对所有结果都给出了正式证明。但在给出这些证明时，我还给出了较平常而言更多的细节，而且提供了大量范例。学生们在看到具体范例时，通常能够更好地掌握理论概念。因此，数学背景知识不够扎实的学生只要愿意付出足够的努力，应当可以理解这些数学论证，从而更深入地掌握相关内容。此外，书中的确包含了一些需要微积分知识才能理解的内容（比如使用极限来确定阶数和证明一些定理）。但是，学生们没掌握这些内容也能理解书中其余部分。对于需要微积分知识的内容，在目录和正文中的空白处都标有一个❶符号；并不需要更多的数学背景知识，但难度高于书中其他部分的内容，则标注了❷符号。

## 预备知识

前面已经说过，本书假定学生的数学背景知识仅限于大学代数和有限数学。真正需要的数学知识在附录 A 中复习。关于计算机科学方面的背景知识，假定学生们已经学习了数据结构课程。因此，通常会在数据结构教科书中出现的内容，本书不再给出。

## 各章内容

本书大部分内容都是根据问题的解决方法而非应用领域来组织的。我感觉这种组织形式可以让算法的设计与分析领域显得更连贯。另外，学生也能更轻松地学到一整套技巧，在遇到新问题时，从中找出也许可行的解决方法。各章内容如下。

- 第 1 章介绍算法设计与分析，其中对阶的概念进行了直观、正式的介绍。
- 第 2 章介绍算法设计的分而治之方法。
- 第 3 章给出动态规划方法，讨论了应当使用动态规划而不是分而治之方法的场景。
- 第 4 章讨论了贪婪方法，并在最后对比了用于解决最优化问题的动态规划和贪婪方法。
- 第 5 章和第 6 章分别介绍回溯算法和分支定界算法。
- 第 7 章从算法分析转到计算复杂度，它是对问题本身的分析。我们通过分析“排序问题”来介绍计算复杂度。之所以选择这一问题，一是因为它重要，二是因为排序算法非常多，还有最重要的一点，是因为一些性能非常出色的排序算法几乎可以达到“排序问题”的时间下限（这里所说的下限，是指仅通过比较键进行排序的算法所花费的时间）。在比较了排序算法之后，分析了通过比较键进行排序的问题。这一章最后讨论了基排序，这种排序算法不是通过比较键来实现排序的。
- 第 8 章通过分析“查找问题”进一步讨论了计算复杂度。这一章分析了在列表中查找键的问题，还分析了“选择问题”，也就是在一个列表中找出第  $k$  小的键的问题。
- 第 9 章专门讨论难解性 (intractability) 和  $NP$  理论。为使本书既通俗易懂又准确严谨，对这一内容的讨论要比一般算法教科书中更为全面。首先明确划分了三类问题之间的界限：一类是已经为其找到多项式时间算法的问题，一类是已经证明为难解的问题，还有一类是尚未证明是难解的，但还从来没有为其找到多项式时间算法的问题。接下来讨论了  $P$  问题、 $NP$  问题、 $NP$  完全问题和  $NP$  等价问题。我发现，如果学生没有真正明白这几类问题之间的关系，经常会糊里糊涂。本章最后讨论了近似算法。
- 第 10 章介绍遗传算法和遗传编程，其中提供了有理论、实践两方面的应用，比如金融贸易算法。
- 第 11 章介绍数论算法，包括欧氏算法和用于判定一个数字是否为质数的新的多项式时间算法。
- 第 12 章简要介绍并行算法，包括并行体系结构和 PRAM 模型。
- 附录 A 复习了理解本书所需要的数学知识。
- 附录 B 介绍了求解递归方程的方法。在第 2 章分析分而治之算法时用到了附录 B 中的结果。
- 附录 C 给出了一种不交集数据结构，在实现第 4 章的两个算法时用到了这一结构。

## 教授方法

为激发学生的兴趣，每一章都以一个与该章内容有关的故事开头。此外，还使用了许多示例，并在各章最后给出大量习题，按节进行分组。在各节的习题之后是补充习题，其挑战性通常要更大一些。

为表明一个问题可以有多种解决方法，有些问题采用了多种解法。例如，在解决“旅行推销员问题”时使用了动态规划、分支定界和近似算法。在解决“0-1 背包问题”时使用了动态规划、回溯和分支定界算法。为

使内容更加完整，我给出了一个横跨多章的题目，其中有一个名为 Nancy 的推销员，她要找出一条最佳推销旅行路线。

## 课程概述

如前所述，本书适用于高年级本科生或研究生算法课程。

在为时一学期的课程中，建议依次讲授以下内容。

第 1 章：全部

附录 B：B.1、B.3 节

第 2 章：2.1~2.5、2.8 节

第 3 章：3.1~3.4、3.6 节

第 4 章：4.1、4.2、4.4 节

第 5 章：5.1、5.2、5.4、5.6、5.7 节

第 6 章：6.1、6.2 节

第 7 章：7.1~7.5、7.7、7.8.1、7.8.2、7.9 节

第 8 章：8.1.1、8.5.1、8.5.2 节

第 9 章：9.1~9.4 节

第 10 章：10.1~10.3.2 节

第 2~6 章均包含若干节，其中每一节都利用该章给出的设计方法解决一个问题。我选择了最感兴趣的几节，你可以选择其中任意一节。

你可能来不及讲授第 11 章和第 12 章。但是，在学习前 10 章之后，学生们应当可以很轻松地理解第 12 章中的内容。数学知识扎实的学生，比如在学习了微积分之后，应当可以自学第 11 章的内容。

## 教师资源

具备教师资格的读者可以申请教师手册、PowerPoint 演示文稿和完整的答案手册。Jones & Bartlett Learning 保留对所有申请进行评估的权利。

## 致谢

我要感谢所有阅读了本书草稿并提出许多有用建议的人们。特别感谢我的同事 William Bultman、Jack Hade、Mary Kenevan、Jim Kenevan、Stuart Kurtz、Don La Budde 和 Miguel Vian，他们欣然全面地审阅了相关内容。还要感谢学术与专业同行审查员们，他们富有深刻见解的批评意见大大提升了本书的质量。他们许多人所做的详尽工作都远远超出了我们的预期。他们是：泽维尔大学的 David D. Berry，瓦尔德斯塔州立大学的 David W. Boyd，圣何塞州立大学的 Vladimir Drobot，加利福尼亚大学欧文分校的 Dan Hirschberg，东北伊利诺伊大学的 Xia Jiang，西弗吉尼亚大学的 Raghu Karinthi，东北伊利诺伊大学的 Peter Kimmel，东北伊利诺伊大学的 C. Donald La Budde，印第安纳大学—普渡大学韦恩堡分校的 Y. Daniel Liang，德雷塞尔大学的 David Magagnosc，南伊利诺伊大学卡本代尔分校的 Robert J. McGlinn，密西西比大学的 Laurie C. Murphy，梅西山学院的 Paul D. Phillips，加州州立理工大学波莫纳分校的 H. Norton Riley，西北大学的 Majid Sarrafzadeh，弗吉尼亚理工学院暨州立大学的 Cliff Shaffer，得克萨斯理工大学的 Nancy Van Cleave，纽约州立大学宾汉姆顿分校的 William L. Ziegler。最后，我还要感谢 Taylor 和 Francis，特别是 Randi Cohen，他们允许将我在 2012 年出版的 *Contemporary*

*Artificial Intelligence*一书中的内容放在本书第 10 章中。

## 勘误

---

在一本如此篇幅的书中，肯定会存在一些错误。如果你发现了任何错误，或有任何改进建议，我非常愿意收到你的来信。请将你的意见发给 Rich Neapolitan。电子信箱：RE-Neapolitan@neiu.edu。谢谢！

R. N.

# 目 录

<b>第1章 算法：效率、分析和阶</b>	1
1.1 算法	1
1.2 开发高效算法的重要性	5
1.2.1 顺序查找与二分查找的对比	6
1.2.2 斐波那契序列	7
1.3 算法分析	10
1.3.1 复杂度分析	10
1.3.2 理论应用	14
1.3.3 正确性分析	15
1.4 阶	15
1.4.1 阶的直观介绍	15
1.4.2 阶数的严谨介绍	17
①1.4.3 利用极限计算阶	23
1.5 本书概要	25
1.6 习题	25
<b>第2章 分而治之</b>	30
2.1 二分查找	30
2.2 合并排序	33
2.3 分而治之方法	38
2.4 快速排序（分割交换排序）	38
2.5 Strassen矩阵乘法算法	42
2.6 大整数的算术运算	46
2.6.1 大整数的表示：加法和其他线性 时间运算	46
2.6.2 大整数的乘法	46
2.7 确定阈值	50
2.8 不应使用分而治之方法的情况	53
2.9 习题	53
<b>第3章 动态规划</b>	58
3.1 二项式系数	58
3.2 Floyd最短路径算法	61
3.3 动态规划与最优化问题	66
3.4 矩阵链乘法	67
3.5 最优二叉查找树	73
3.6 旅行推销员问题	79
3.7 序列对准	84
3.8 习题	88
<b>第4章 贪婪方法</b>	92
4.1 最小生成树	94
4.1.1 Prim算法	96
4.1.2 Kruskal算法	100
4.1.3 Prim算法与Kruskal算法的比较	103
4.1.4 最终讨论	103
4.2 单源最短路径的Dijkstra算法	104
4.3 调度计划	106
4.3.1 使系统内总时间最短	106
4.3.2 带有最终期限的调度安排	108
4.4 霍夫曼编码	112
4.4.1 前缀码	113
4.4.2 霍夫曼算法	114
4.5 贪婪方法与动态规划的比较：背包问题	116
4.5.1 0-1背包问题的一种贪婪方法	116
4.5.2 部分背包问题的贪婪方法	118
4.5.3 0-1背包问题的动态规划方法	118
4.5.4 0-1背包问题动态规划算法的 改进	118
4.6 习题	120
<b>第5章 回溯</b>	124
5.1 回溯方法	124
5.2 n皇后问题	129
5.3 用蒙特卡洛算法估计回溯算法的效率	132
5.4 “子集之和”问题	134
5.5 图的着色	138
5.6 哈密顿回路问题	141
5.7 0-1背包问题	143
5.7.1 0-1背包问题的回溯算法	143
5.7.2 比较0-1背包问题的动态规划 算法与回溯算法	149
5.8 习题	150
<b>第6章 分支定界</b>	153
6.1 用0-1背包问题说明分支定界	154
6.1.1 带有分支定界修剪的宽度优先 查找	154
6.1.2 带有分支定界修剪的最佳优先 查找	158
6.2 旅行推销员问题	161
⑥6.3 漱因推理（诊断）	167
6.4 习题	173
<b>第7章 计算复杂度介绍：排序问题</b>	175
7.1 计算复杂度	175
7.2 插入排序和选择排序	176

7.3 每次比较最多减少一个倒置的算法的下限 .....	179	第 10 章 遗传算法和遗传编程 .....	268
7.4 再谈合并排序 .....	181	10.1 遗传知识复习 .....	268
7.5 再谈快速排序 .....	185	10.2 遗传算法 .....	270
7.6 堆排序 .....	186	10.2.1 算法 .....	270
7.6.1 堆和基本堆例程 .....	186	10.2.2 说明范例 .....	270
7.6.2 堆排序的一种实现 .....	189	10.2.3 旅行推销员问题 .....	272
7.7 合并排序、快速排序和堆排序的比较 .....	193	10.3 遗传编程 .....	278
7.8 仅通过键的比较进行排序的下限 .....	194	10.3.1 说明范例 .....	279
7.8.1 排序算法的决策树 .....	194	10.3.2 人造蚂蚁 .....	281
7.8.2 最差情况下的下限 .....	196	10.3.3 在金融贸易中的应用 .....	283
7.8.3 平均情况下的下限 .....	197	10.4 讨论及扩展阅读 .....	284
7.9 分配排序（基数排序） .....	200	10.5 习题 .....	284
7.10 习题 .....	203		
<b>第 8 章 再谈计算复杂度：查找问题 .....</b>	<b>207</b>	<b>第 11 章 数论算法 .....</b>	<b>286</b>
8.1 仅通过键的比较进行查找的下限 .....	207	11.1 数论回顾 .....	286
8.1.1 最差表现的下限 .....	209	11.1.1 合数与质数 .....	286
8.1.2 平均情况下的下限 .....	210	11.1.2 最大公约数 .....	286
8.2 插值查找 .....	213	11.1.3 质因数分解 .....	288
8.3 树中的查找 .....	215	11.1.4 最小公倍数 .....	289
8.3.1 二叉查找树 .....	215	11.2 计算最大公约数 .....	290
8.3.2 B 树 .....	218	11.2.1 欧氏算法 .....	290
8.4 散列 .....	219	11.2.2 欧氏算法的扩展 .....	292
8.5 选择问题：对手论证 .....	222	11.3 模运算回顾 .....	294
8.5.1 找出最大键 .....	222	11.3.1 群论 .....	294
8.5.2 同时找出最大键和最小键 .....	223	11.3.2 关于 $n$ 同余 .....	295
8.5.3 找出第二大的键 .....	227	11.3.3 子群 .....	299
8.5.4 查找第 $k$ 小的键 .....	230	◆ 11.4 模线性方程的求解 .....	302
8.5.5 选择问题的一种概率算法 .....	236	◆ 11.5 计算模的幂 .....	305
8.6 习题 .....	238	11.6 寻找大质数 .....	307
<b>第 9 章 计算复杂度和难解性：NP 理论简介 .....</b>	<b>241</b>	11.6.1 寻找大质数 .....	307
9.1 难解性 .....	241	11.6.2 检查一个数字是否为质数 .....	307
9.2 再谈输入规模 .....	242	11.7 RSA 公钥密码系统 .....	318
9.3 三类一般问题 .....	244	11.7.1 公钥加密系统 .....	318
9.3.1 已经找到多项式时间算法的问题 .....	244	11.7.2 RSA 加密系统 .....	319
9.3.2 已经证明难解的问题 .....	245	11.8 习题 .....	321
9.3.3 未被证明是难解的，但也从来没有找到多项式时间算法的问题 .....	245		
9.4 NP 理论 .....	245	<b>第 12 章 并行算法简介 .....</b>	<b>324</b>
9.4.1 集合 $P$ 和 $NP$ .....	247	12.1 并行体系结构 .....	325
9.4.2 $NP$ 完全问题 .....	250	12.1.1 控制机制 .....	326
9.4.3 $NP$ 困难、 $NP$ 容易和 $NP$ 等价问题 .....	256	12.1.2 地址空间的组织 .....	326
9.5 处理 $NP$ 困难问题 .....	259	12.1.3 互联网络 .....	328
9.5.1 旅行推销员问题的近似算法 .....	259	12.2 PRAM 模型 .....	330
9.5.2 装箱问题的近似算法 .....	263	12.2.1 为 CREW PRAM 模型设计算法 .....	332
9.6 习题 .....	266	12.2.2 为 CRCW PRAM 模型设计算法 .....	337
		12.3 习题 .....	339
		<b>附录 A 必备数学知识回顾 .....</b>	<b>340</b>
		<b>附录 B 求解递归方程：在递归算法分析中的应用 .....</b>	<b>363</b>
		<b>附录 C 不交集的数据结构 .....</b>	<b>388</b>
		<b>参考文献 .....</b>	<b>395</b>

# 第 1 章

## 算法：效率、分析和阶



本书讨论用计算机解决问题的方法。这里所说的“方法”并不是指一种程序设计风格或者一种程序设计语言，而是用于解决问题的方法或方法学。例如，假设 Barney Beagle 希望在电话簿中找到名字“Collie, Colleen”。一种方法是从第一个名字开始，依次查看每个名字，直到找出“Collie, Colleen”为止。但是，没有人会这样来找一个名字。电话簿中的名字都是有序的，所以 Barney 会利用这一事实，将电话簿翻到他认为 C 字头名字所在的位置。如果他翻过了，可以往回翻一些。他不停地前后翻动，直到找出“Collie, Colleen”所在的页面。你可能看出来了，第二种方法就是一种经过修改的二分查找，而第一种方法是一种顺序查找。1.2 节会进一步讨论这两种查找方法。这里要说的是，这一问题有两种不同的解决方法，而这两种方法与程序设计语言或风格没有任何关系。计算机程序只不过是实现这些方法的一种途径。

第 2 章至第 6 章讨论各种解决问题的方法，并运用这些方法解决各种问题。将某一方法应用于某一问题，会得到解决该问题的一个逐步过程。这个逐步过程称为该问题的算法 (algorithm)。研究这些方法及其应用的目的就是掌握许多方法，以便在遇到新问题时，可以从中找出解决该问题的方法。后面经常会看到，可以使用几种不同的方法来解决同一问题，但其中一种方法得出的算法要远快于其他方法得出的算法。在电话簿中查找姓名时，经过修改的二分查找法当然要快于顺序查找法。因此，我们不仅要判断某个问题能否用某一给定方法解决，还要分析所得到的算法在时间和存储方面的效率如何。当在计算机上实现算法时，时间是指 CPU 周期，存储是指内存。你可能会感到奇怪，计算机变得越来越快，内存变得越来越便宜，为什么还需要考虑效率。本章将讨论一些基本概念，它们是理解本书内容所必需的。在此过程中我会向你说明，为什么无论计算机变得多快、内存变得多便宜，还是要考虑效率。

### 1.1 算法

到目前为止，我们已经提到了“问题”“答案”和“算法”等词。我们大多数人都能很好地理解这些词语的含义。但为了打下一个坚实的基础，还是再具体定义一下这些术语。

计算机程序由完成特定任务（比如排序）的各个模块组成，计算机是可以理解这些模块的。本书关注的不是整个程序的设计，而是这些完成特定任务的各个模块的设计。这些特定的任务称为“问题”。明确地说，问题 (problem) 就是要为其寻求答案的疑问。下面给出问题的一些例子。

**例 1.1** 将一个包含  $n$  个数字的列表  $S$  按非递减顺序排序。其答案就是这些数字排序后的结果。

这里所说的列表 (list) 是指一组按特定顺序排列的项目。例如，

$$S = [10, 7, 11, 5, 13, 8]$$

是一个包含六个数字的列表，其中第一个数字为 10，第二个为 7，等等。例 1.1 中说到要按“非递减顺序”对这个列表排序，而不是按升序排列，这样就存在同一数字在列表中出现多次的可能性。

**例 1.2** 判断数字  $x$  是否在一个包含  $n$  个数字的列表  $S$  中。如果  $x$  在  $S$  中，则答案为“是”，否则为“否”。

一个问题可能会在问题描述中包含一些未指定取值的变量。这些变量称为问题的参数 (parameter)。例 1.1 中有两个参数： $S$  (列表) 和  $n$  ( $S$  中项目的个数)。例 1.2 中有三个参数： $S$ 、 $n$  和数字  $x$ 。在这两个例子中，并不需要让  $n$  成为一个参数，因为它的值由  $S$  唯一确定。但是，让  $n$  成为参数有利于问题的描述。

因为问题中包含参数，所以它代表着一类问题，每对参数进行一次赋值，就得到其中一个问题。对参数的

每次特定赋值就称为该问题的一个实例（instance）。一个问题实例的答案（solution）就是该实例中所提问题的回答。

**例 1.3** 例 1.1 所提问题的一个实例为：

$$S = [10, 7, 11, 5, 13, 8], n = 6$$

这一实例的答案为  $[5, 7, 8, 10, 11, 13]$ 。

**例 1.4** 例 1.2 所提问题的一个实例为：

$$S = [10, 7, 11, 5, 13, 8], n = 6, x = 5$$

这一实例的答案为：“是， $x$  在  $S$  中。”

对于例 1.3 中的实例，只需审视  $S$ ，用一种无法具体描述的认知步骤在大脑中生成有序序列，就能给出该实例的答案。之所以能这样做，是因为  $S$  非常小，在人类的意识水平，大脑可以快速扫描  $S$ ，几乎马上就可以给出答案（因此，我们无法描述大脑用于获得此答案的具体步骤）。但是，如果这个例子中的  $n$  值为 1000，那人们就无法使用这一方法，当然也就不可能将这样一种数字排序方法转换为计算机程序。为了给出能够解答某一问题所有实例的计算机程序，我们必须给出一个通用的逐步过程，用于给出每个实例的答案。这一逐步过程称为算法。我们说算法解决了问题。

**例 1.5** 例 1.2 所示问题的一种算法。从  $S$  中的第一项开始，依次将  $x$  与  $S$  中的每一项对比，直到找出  $x$  或  $S$  中的项目耗尽为止。如果找到了  $x$ ，则答案为“是”；如果没有找到  $x$ ，则答案为“否”。

任何算法都可以像例 1.5 中那样用日常语言来描述。但是，以这种方式来书写算法有两个缺点。第一，这样很难书写复杂算法，即使勉强写出，人们也很难理解。第二，根据算法的日常语言描述，无法清晰地知道如何生成对该算法的计算机语言描述。

因为 C++ 是当前学生比较熟悉的一门语言，所以我们使用一种类似于 C++ 的伪代码来书写算法。只要拥有类 Algol 命令式语言（比如 C、Pascal 或 Java）的编程经验，应当不难看懂这种伪代码。

为演示该伪代码，我们以一种算法为例，该算法可以解决例 1.2 所示问题的一般形式。为简单起见，例 1.1 和例 1.2 的表述都是针对数字的。但是，一般来说，我们希望对来自任意有序集合中的项目进行查找和排序。通常，每一项都唯一地确定了一条记录，因此，这些项目通常称为键（key）。例如，一条记录可能由某个人的个人信息组成，以这个人的社会保障号作为键。为这些项目定义一种数据类型 keytype，在编写查找和排序算法时就使用这一数据类型。这就是说，这些项目来自任意有序集合。

下面的算法用数组表示列表  $S$ ，它返回的不只是“是”或“否”，如果  $x$  在  $S$  中，还会返回  $x$  在数组中的位置；如果不在，则返回 0。这一查找算法并不要求这些项目来自有序集合，但我们仍然使用标准数据类型 keytype。

### 算法 1.1 顺序查找

问题：键  $x$  是否存在于拥有  $n$  个键的数组  $S$  中？

输入（参数）：正整数  $n$ ；键的数组  $S$ ，其索引范围为 1 至  $n$ ；键  $x$ 。

输出：location， $x$  在  $S$  中的位置（若  $x$  不在  $S$  中，则为 0）。

```
void seqsearch (int n,
                const keytype S[ ],
                keytype x,
                index& location)
{
    location = 1;
    while (location <= n && S[location] != x)
        location++;
    if (location > n)
        location = 0;
}
```

这段伪代码与 C++ 非常类似，但不完全等同。一个很明显的不同就是对数组的使用。C++ 中的数组索引只

能是从 0 开始的整数。很多时候，使用以其他整数范围为索引的数组可以更清楚地解释算法；而有时，使用非整数索引可以将算法解释得最为清楚。因此，在伪代码中，允许数组的索引是任意集合。在指明算法的“输入”和“输出”时，总会规定索引的范围。例如，算法 1.1 中指明  $S$  的索引范围为 1 至  $n$ 。人们在计算列表中的项目数时，习惯于从 1 开始，所以为列表使用这一索引范围是很不错的选择。当然，这个算法可以直接用 C++ 实现，声明如下：

```
keytype S[n + 1];
```

跳过  $S[0]$  位置不用。下文不再讨论以任何程序设计语言实现算法。我们的目的只是清楚明了地给出算法，从而使其便于理解和分析。

关于伪代码中的数组，还有另外两点明显不同于 C++ 的地方。第一，允许采用变长两维数组作为例程的参数，比如算法 1.4。第二，我们声明了局部变长数组。例如，如果  $n$  是过程 example 的一个参数，而且需要一个索引范围为 2 到  $n$  的局部数组，可以声明如下：

```
void example (int n)
{
    keytype S[2..n];
    ...
}
```

符号  $S[2..n]$  表示以 2 到  $n$  为索引范围的数组，这完全是伪代码；也就是说，它不是 C++ 语言的组成部分。

如果与使用实际的 C++ 指令相比，使用数学表达式或日常语言可以更简洁、更清楚地描述算法步骤，我们就会这么做。例如，假定仅当变量  $x$  介于取值 low 和 high 之间时，某些指令才会执行。我们会写为：

```
if (low ≤ x ≤ high) {
    ...
}
```

而不是：

```
if (low <= x && x <= high){
    ...
}
```

假定我们希望变量  $x$  取变量  $y$  的值， $y$  取  $x$  的值。我们将写为：

交换  $x$  和  $y$ ；

而不是写为：

```
temp = x;
x = y;
y = temp;
```

除了数据类型 keytype 之外，我们还经常使用以下数据类型，它们也不是预定义的 C++ 数据类型。

数据类型	含    义
index	用作索引的整数变量
number	一个可以定义为整数（int）或实数（float）的变量
bool	一个可以取“true”或“false”的变量

有时数字可以取任意实数，有时只能取整数。如果这一点对算法来说并不重要，我们将使用数据类型 number。

我们有时会使用下面的非标准控制结构：

```
repeat ( n times ) {
    :
}
```

其含义是将代码重复  $n$  次。在 C++ 中，需要另外引入一个控制变量，并编写 `for` 循环。只有当真正需要引用循环中的控制变量时，我们才会使用 `for` 循环。

当算法的名字看起来与其返回值很吻合时，我们就将算法写为函数（function）。否则，就将其写为进程（procedure，C++ 中的 `void` 函数），并使用引用参数（reference parameter，也就是按地址传递的参数）来返回值。如果参数不是数组，在声明它时，会在数据类型名的末尾添加一个`&` 符号。它在这里的意思是：这个参数包含算法的一个返回值。因为数组在 C++ 中是自动按引用传递的，而且在传递数组时，C++ 中也没有使用`&` 符号，所以我们没有使用`&` 来表示数组中包含算法的返回值。相反，由于 C++ 中使用保留字 `const` 来防止对所传递数组进行修改，所以我们使用 `const` 表示数组中不包含算法的返回值。

一般情况下，我们尽量避免使用 C++ 特有的功能，以便那些只了解其他高级语言的人也能读懂伪代码。但是，我们的确会编写一些类似于  $i++$  之类的指令，其含义是指将  $i$  递增 1。

如果不了解 C++，你可能会觉得逻辑运算符和某些关系运算符有些陌生。这些符号列出如下。

运算符	C++ 符号	比 较	C++ 代码
and	<code>&amp;&amp;</code>	$x = y$	<code>(x == y)</code>
or	<code>  </code>	$x \neq y$	<code>(x != y)</code>
not	<code>!</code>	$(x \leq y)$	<code>(x &lt;= y)</code>
		$x \geq y$	<code>(x &gt;= y)</code>

下面给出更多的示例算法。第一个例子演示函数的使用。过程的例程名之前带有关键字 `void`，而函数的例程名之前带有函数返回的数据类型。这个值在函数中通过 `return` 语句返回。

### 算法 1.2 数组成员求和

问题：将包含  $n$  个数字的数组  $S$  中的所有成员加在一起。

输入：正整数  $n$ ；数字数组  $S$ ，其索引范围为 1 至  $n$ 。

输出： $sum$ ， $S$  中的数字之和。

```
number sum (int n, const number S[])
{
    index i;
    number result;

    result = 0;
    for (i = 1; i <= n; i++)
        result = result + S[i];
    return result;
}
```

本书将讨论许多排序算法。下面是其中很简单的一个。

### 算法 1.3 交换排序

问题：按非递减顺序对  $n$  个键排序。

输入：正整数  $n$ ；键的数组  $S$ ，其索引范围为 1 至  $n$ 。

输出：数组  $S$ ，其中的键按非递减顺序排列。

```
void exchangesort (int n; keytype S[])
{
    index i, j;
    for (i=1;i<=n;i++)
        for (j=i+1;j<=n;j++)
```

```

if (S[j] < S[i])
    交换 S[i] 和 S[j];
}

```

指令

交换 S[i] 和 S[j];

的含义是， $S[i]$  将取  $S[j]$  的值， $S[j]$  将取  $S[i]$  的值。这条命令一点都不像 C++ 指令；如果不使用 C++ 指令的细节可以将事情描述得更简单，我们就一定会这么做。“交换排序”是将第  $i$  个位置的数字与第  $i+1$  个位置到第  $n$  个位置的数字进行比较。只要发现给定位置的数字小于第  $i$  个位置的数字，就交换这两个数字。这样，在完成第一遍 `for-i` 循环后，最小的数字将放在第一位，在第二遍循环后，第二小的数字将放在第二位，以此类推。

下面的算法执行矩阵乘法。回想一下，如果有两个  $2 \times 2$  矩阵：

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \text{ 和 } B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

则它们的乘积  $C = A \times B$  为：

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}$$

例如，

$$\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 2 \times 5 + 3 \times 6 & 2 \times 7 + 3 \times 8 \\ 4 \times 5 + 1 \times 6 & 4 \times 7 + 1 \times 8 \end{bmatrix} = \begin{bmatrix} 28 & 38 \\ 26 & 36 \end{bmatrix}$$

一般情况下，如果有两个  $n \times n$  矩阵  $A$  和  $B$ ，则其乘积  $C$  如下：

$$c_{ij} = \sum_{k=1}^n a_{ik}b_{kj} \quad (1 \leq i, j \leq n)$$

由这一定义，可以直接得出矩阵乘法的以下算法。

#### 算法 1.4 矩阵乘法

问题：计算两个  $n \times n$  矩阵的乘积。

输入：一个正整数  $n$ ；二维数字数组  $A$  和  $B$ ，每个数组的行和列都以 1 至  $n$  为索引范围。

输出：一个二维数字数组  $C$ ，包含了  $A$  和  $B$  的乘积，它的行和列都以 1 至  $n$  为索引范围。

```

void matrixmult (int n,
                  const number A[][],
                  const number B[][],
                  number C[][])
{
    index i, j, k;

    for (i = 1; i <=n; i++)
        for (j=1; j<=n; j++){
            C[i][j] = 0;
            for (k=1; k<=n; k++)
                C[i][j] = C[i][j]+A[i][k]*B[k][j];
        }
}

```

## 1.2 开发高效算法的重要性

前面曾经提到，无论计算机变得多么快速，内存变得多么廉价，效率永远都是重要的考虑因素。接下来，我们通过对比同一问题的两种算法来说明其原因。

## 1.2.1 顺序查找与二分查找的对比

前面曾经提到，在电话簿中查找姓名的方法是一种经过修改的二分查找，它通常要远快于顺序查找。下面将对比这两种方法的算法，以说明二分查找法要快多少。

我们已经编写了完成顺序查找的算法，即算法 1.1。在一个非递减顺序数组中进行二分查找的算法类似于用大拇指前后翻动电话簿。也就是说，假定正在查找  $x$ ，算法首先将  $x$  与数组的中间项进行对比。如果相等，则算法完成。如果  $x$  小于中间项，则  $x$  必然在数组的前半部分（如果存在的话），算法将对数组的前半部分重复该查找过程。（也就是说，将  $x$  与数组前半部分的中间项进行比较，等等。）如果  $x$  大于数组的中间项，则对数组的后半部分重复查找过程。一直重复此过程，直到找出  $x$ ，或者确认  $x$  不在数组中为止。这一方法的算法如下。

### 算法 1.5 二分查找

问题：判断  $x$  是否在一个包含  $n$  个键的有序数组  $S$  中。

输入：正整数  $n$ ；有序（非递减顺序）键数组  $S$ ，其索引范围为 1 至  $n$ ；键  $x$ 。

输出：location， $x$  在  $S$  中的位置（如果  $x$  不在  $S$  中，则为 0）。

```
void binsearch (int n,
                const keytype S[],
                keytype x,
                index& location)
{
    index low, high, mid;

    low = 1; high = n;
    location = 0;
    while (low <= high && location == 0){
        mid = [(low + high)/2];
        if (x == S[mid])
            location = mid;
        else if (x < S[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
}
```

我们来对比顺序查找与二分查找所做的工作。重点确定每种算法执行的比较次数。如果数组  $S$  包含 32 项，且  $x$  不在数组中，则算法 1.1（顺序查找）需要将  $x$  与所有 32 项进行比较后，才能确定  $x$  不在数组中。一般情况下，对于一个大小为  $n$  的数组，顺序查找需要完成  $n$  次比较才能确定  $x$  不在该数组中。显然，在搜索一个大小为  $n$  的数组时，这是顺序查找所执行的最大比较次数。也就是说，如果  $x$  在数组中，则比较次数不大于  $n$ 。

下面考虑算法 1.5（二分查找）。每执行一遍 `while` 循环，会将  $x$  与  $S[mid]$  比较两次（找到  $x$  时除外）。在用高效汇编语言实现该算法时，每执行一遍 `while` 循环，仅将  $x$  与  $S[mid]$  比较一次，比较结果将确定条件代码，然后根据条件代码的取值执行适当的跳转。这意味着每执行一遍 `while` 循环，仅将  $x$  与  $S[mid]$  比较一次。我们假定算法就是以这种方式实现的。在这一假设条件下，图 1-1 表明，对于一个大小为 32 的数组，若  $x$  大于其中所有项目，该算法将执行 6 次比较。注意， $6 = \lg 32 + 1$ 。这里的  $\lg$  是指  $\log_2$ 。在算法分析中经常会遇到  $\log_2$ ，所以我们为它保留了特殊符号  $\lg$ 。你应当相信，二分查找最多只需要执行这么多的比较次数。也就是说，如果  $x$  在数组中，或者  $x$  小于数组中的所有项目，或者  $x$  在两个数组项目之间，所执行的比较次数都不会超过  $x$  大于所有数组项目时的次数。



图 1-1 对于一个大小为 32 的数组，当  $x$  大于其中所有项目时，二分查找法用来与  $x$  进行比较的数组项目。这些项目的编号就是它们与  $x$  的比较顺序

假设将数组大小加倍，使其包含 64 个项目。二分查找只会增加一次比较，因为第一次比较就将数组减半，得到一个大小为 32 的待查子数组。因此，对于一个大小为 64 的数组，当  $x$  大于其中所有项目时，二分查找执行 7 次比较。注意， $7 = \lg 64 + 1$ 。一般情况下，数组大小每加倍一次，只需要增加一次比较。因此，若  $n$  是 2 的幂，对于一个大小为  $n$  的数组，若  $x$  大于其中所有项目，则二分查找执行的比较次数为  $\lg n + 1$ 。

表 1-1 对比了当  $x$  大于数组中所有项目时，顺序查找和二分查找对于不同  $n$  值所执行的比较次数。当数组包含大约 40 亿个项目时（大约是世界上的人口数量），二分查找只执行 33 次比较，而顺序查找则将  $x$  与所有 40 亿个项目进行了比较。即使计算机能在 1 纳秒（一秒的十亿分之一）内完成一次 while 循环，顺序查找也需要 4 秒钟才能确定  $x$  不在数组中，而二分查找几乎马上就可以做出判断。对于在线应用程序或者在需要查找许多项目时，这一差别会非常明显。

表 1-1 当  $x$  大于所有数组项目时，顺序查找和二分查找执行的比较次数

数组大小	顺序查找执行的比较次数	二分查找执行的比较次数
128	128	8
1 024	1 024	11
1 048 576	1 048 576	21
4 294 967 296	4 294 967 296	33

为方便起见，在前面对二分查找的讨论中，所考虑数组的大小都是 2 的幂。第 2 章还会再次讨论二分查找法，将其作为该章主题内容——分而治之方法的一个例子。届时将考虑大小可以为任意正整数的数组。

尽管上面这个查找范例给人们留下了很深刻的印象，但其说服力也并不强，因为顺序查找仍然可以在人的有生之年内完成任务。接下来将研究一个更差的算法，它无法在可接受的时间内完成任务。

## 1.2.2 斐波那契序列

现在讨论用于计算斐波那契序列第  $n$  个项目的算法。斐波那契序列的递归定义如下：

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \quad (n \geq 2) \end{aligned}$$

计算序列中的前几个项目，可得：

$$\begin{aligned} f_2 &= f_1 + f_0 = 1 + 0 = 1 \\ f_3 &= f_2 + f_1 = 1 + 1 = 2 \\ f_4 &= f_3 + f_2 = 2 + 1 = 3 \\ f_5 &= f_4 + f_3 = 3 + 2 = 5 \\ &\dots \end{aligned}$$

斐波那契序列在计算机科学和数学中有着多种应用。因为斐波那契序列是以递归形式定义的，所以可由定义中得出以下递归算法。

### 算法 1.6 斐波那契序列的第 $n$ 项（递归）

问题：确定斐波那契序列的第  $n$  项。

输入：一个非负整数  $n$ 。

输出：fib，斐波那契序列的第  $n$  项。

```
int fib (int n)
{
    if (n <= 1)
        return n;
    else
        return fib(n-1)+fib(n-2);
}
```

“非负整数”是指大于或等于 0 的整数，而“正整数”则是指严格大于 0 的整数。我们以这种方式规定算法的输入，明确指出输入可取哪些值。但是，为避免混乱，算法的表达式中直接将  $n$  声明为一个整数。本书中将一直遵循这一约定。

尽管这一算法很容易生成，也很好理解，但其效率却极为低下。图 1-2 给出了在计算  $\text{fib}(5)$  时与该算法对应的递归树。树中一个节点处的调用会递归调用其子节点处的调用。例如，为在顶级获得  $\text{fib}(5)$ ，需要  $\text{fib}(4)$  和  $\text{fib}(3)$ ；为了获得  $\text{fib}(3)$ ，又需要  $\text{fib}(2)$  和  $\text{fib}(1)$ ，以此类推。如树中所示，因为需要一遍又一遍地计算各个取值，所以这个函数的效率很低。例如， $\text{fib}(2)$  被计算了三次。

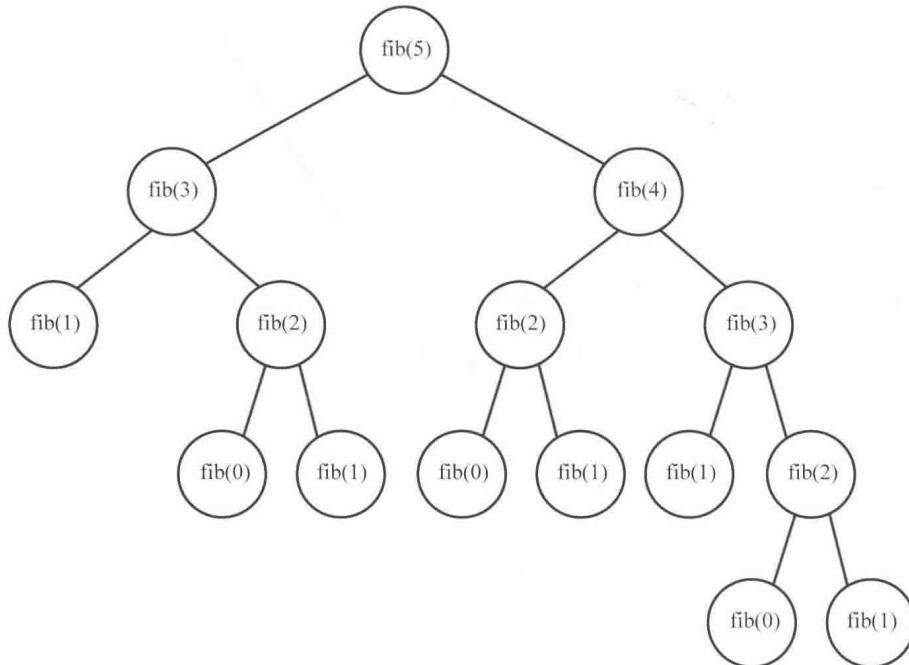


图 1-2 算法 1.6 计算第 5 个斐波那契项时的对应递归树

这个算法的效率有多低呢？图 1-2 中的树显示，为了计算  $\text{fib}(n)$  ( $0 \leq n \leq 6$ )，该算法计算的项数如下：

$n$	计算的项数
0	1
1	1
2	3
3	5
4	9
5	15
6	25

当  $1 \leq n \leq 5$  时，只需要数一数以  $\text{fib}(n)$  为根节点的子树中有多少个节点，即可求出前六个值。而在计算  $\text{fib}(6)$  中的项数时，需要先计算分别以  $\text{fib}(5)$  和  $\text{fib}(4)$  为根节点的树中有多少节点，然后再将两者之和加上  $\text{fib}(6)$  所在的根节点。我们无法由这些数字得到一个类似于二分查找那样的简单表达式。但可以注意到，对于前 7 个数值，每当  $n$  值增加 2 时，树中的项目数将超过原来的 2 倍。例如，当  $n=4$  时，树中有 9 项；当  $n=6$  时，有 25 项。我们将  $T(n)$  称为  $n$  的递归树中的项数。如果  $n$  每增加 2，项数都会超过原来的 2 倍，则对于偶数  $n$ ，可以得到下式：

$$\begin{aligned}
 T(n) &> 2 \times T(n-2) \\
 &> 2 \times 2 \times T(n-4) \\
 &> 2 \times 2 \times 2 \times T(n-6) \\
 &\vdots \\
 &> \underbrace{2 \times 2 \times 2 \times \cdots \times 2}_{n/2 \text{ 项}} \times T(0)
 \end{aligned}$$

因为  $T(0)=1$ , 所以这将意味着  $T(n)>2^{n/2}$ 。我们用归纳法来证明, 即使  $n$  不是偶数, 上式在  $n \geq 2$  时也是成立的。当  $n=1$  时, 该不等式不成立, 这是因为  $T(1)=1$ , 它小于  $2^{1/2}$ 。归纳法在附录 A 的 A.3 节复习。

**定理 1.1** 若  $T(n)$  是与算法 1.6 对应的递归树中的项数, 则当  $n \geq 2$  时,

$$T(n) > 2^{n/2}$$

**证明:** 通过对  $n$  进行归纳来完成证明。

**归纳基础:** 因为归纳部分需要前两种情景的结果, 所以需要两个基础情景。对于  $n=2$  和  $n=3$ , 图 1-2 中的递归表明:

$$\begin{aligned}
 T(2) &= 3 > 2 = 2^{2/2} \\
 T(3) &= 5 > 2.8323 \approx 2^{3/2}
 \end{aligned}$$

**归纳假设:** 做出归纳假设的一种方式是假定该命题对于所有小于  $n$  的  $m$  值都成立。随后在归纳步骤中证明, 由这一假设可以推导出, 上述命题对于  $n$  也必定成立。下面的证明中采用了此方法。假设对于  $2 \leq m < n$  的  $m$  值, 有

$$T(m) > 2^{m/2}$$

**归纳步骤:** 必须证明  $T(n) > 2^{n/2}$ 。 $T(n)$  的值等于  $T(n-1)$  与  $T(n-2)$  之和再加上一个根节点。因此,

$$\begin{aligned}
 T(n) &= T(n-1) + T(n-2) + 1 \\
 &> 2^{(n-1)/2} + 2^{(n-2)/2} + 1 \quad \text{根据归纳假设} \\
 &> 2^{(n-2)/2} + 2^{(n-2)/2} = 2 \times 2^{\left(\frac{n}{2}\right)-1} = 2^{n/2}
 \end{aligned}$$

我们已经确认, 为求出第  $n$  个斐波那契项, 算法 1.6 计算的项数大于  $2^{n/2}$ 。后面会再次讨论这一结果, 以说明该算法是多么低效。但首先来设计一种用于计算第  $n$  个斐波那契项的高效算法。回想一下, 递归算法的问题就在于会一遍又一遍地计算同一数值。如图 1-2 所示, 在计算  $\text{fib}(5)$  时,  $\text{fib}(2)$  被计算了 3 次。如果在计算一个值时, 将其保存在一个数组中, 后面再需要它时, 就不用再重新计算了。下面的迭代算法就使用了这一策略。

**算法 1.7 斐波那契序列的第  $n$  项 (迭代)**

**问题:** 确定斐波那契序列的第  $n$  项。

**输入:** 一个非负整数  $n$ 。

**输出:**  $\text{fib}_2$ , 斐波那契序列的第  $n$  项。

```

int fib2 (int n)
{
    index i;
    int f[0..n];

    f[0]=0;
    if (n > 0)
        f[1]=1;
    for (i=2; i<=n; i++)
        f[i]=f[i-1]+f[i-2];
    return f[n];
}

```

不用数组  $f$  也可以写出算法 1.7，因为循环的每次迭代只需要最近两项。但是，使用数组可以表述得更清楚一些。

以上算法在计算  $\text{fib2}(n)$  时，前  $n+1$  项中的每一项都仅计算一次。因此，为求出第  $n$  个斐波那契项，它只需要计算  $n+1$  个项目。回想一下，算法 1.6 需要计算超过  $2^{n/2}$  个项目来计算第  $n$  个斐波那契项。表 1-2 对比了这些表达式在  $n$  取不同值时的结果。执行时间的计算基于一个简化假设：一个项目可以在  $10^{-9}$  秒内计算得出。该表给出了算法 1.7 在一台假想计算机上计算第  $n$  项时所耗费的时间（该计算机计算每项的时间为 1 纳秒），并给出了执行算法 1.7 所需要的时间下限。当  $n$  为 80 时，算法 1.6 至少需要 18 分钟。当  $n$  为 120 时，它需要超过 36 年，与人类的寿命相比，这是一个无法忍受的时间。即使可以建造一台快 10 亿倍的计算机，算法 1.6 也需要 40 000 年才能计算出第 200 项。将第 200 项的时间除以 10 亿即可得出这一结果。可以看出，无论计算机变得多么快，算法 1.6 仍然需要耗费无法忍受的时间，除非是  $n$  值非常小。另一方面，算法 1.7 几乎马上就可以计算出第  $n$  个斐波那契项。这一比较结果表明，无论计算机变得多么快，算法的效率都依然是一个重要的考虑因素。

表 1-2 算法 1.6 与算法 1.7 的比较

$n$	$n+1$	$2^{n/2}$	使用算法 1.7 的执行时间	使用算法 1.6 的执行时间下限
40	41	1 048 576	41 ns*	1048 $\mu$ s†
60	61	$1.1 \times 10^9$	61 ns	1 秒
80	81	$1.1 \times 10^{12}$	81 ns	18 分
100	101	$1.1 \times 10^{15}$	101 ns	13 天
120	121	$1.2 \times 10^{18}$	121 ns	36 年
160	161	$1.2 \times 10^{24}$	161 ns	$3.8 \times 10^7$ 年
200	201	$1.3 \times 10^{30}$	201 ns	$4 \times 10^{13}$ 年

\*  $1 \text{ ns} = 10^{-9}$  秒。

†  $1 \mu\text{s} = 10^{-6}$  秒。

算法 1.6 是一种分而治之算法。回想一下，分而治之方法为有序数组的查找问题提供了一种非常高效的算法（算法 1.5：二分查找）。如第 2 章所示，分而治之方法会为某些问题给出非常高效的算法，但对其他一些问题，却会给出非常低效的算法。我们为计算第  $n$  个斐波那契项给出的高效算法（算法 1.7）是动态规划方法（dynamic programming approach）的一个例子，这种方法是第 3 章的主题。可以看出，最佳方法的选择才是本质所在。

前面已经表明，算法 1.6 计算的项数非常大，至少为指数级，那还会更糟糕吗？答案是否定的。利用附录 B 中的技术，有可能给出项数的准确公式，该公式是  $n$  的指数函数。有关斐波那契序列的深入讨论，请参阅附录 B 中的例 B.5 和例 B.9。

## 1.3 算法分析

为了判断一种算法在解决某种问题时的效率如何，需要分析算法。在前一节比较算法时，引入了算法的效率分析。但这些分析过程相当随意，不够正式。我们现在讨论算法分析中使用的术语以及标准的分析方法。在本书后续部分将遵循这些标准。

### 1.3.1 复杂度分析

在分析一种算法的时间效率时，并没有计算出实际 CPU 周期数，因为这一数值取决于运行算法的具体计算机。此外，我们甚至不希望计算所执行的指令数，因为指令数取决于用于实现算法的程序设计语言，以及程序员编写程序的方式。我们希望有一种量度，不受计算机、程序设计语言、程序员、算法的所有复杂细节（比

如循环索引的递增、指针设定等)的影响。通过对比两种算法在不同  $n$  值时完成的比较次数, 我们了解到算法 1.5 的效率要远高于算法 1.1, 其中  $n$  是数组中的项数。这是分析算法的标准方法。一般情况下, 算法的运行时间随输入的规模而增加, 总运行时间近似正比于某些基本运算(比如比较指令)的执行次数。因此, 可以推导某些基本运算的执行次数, 将其表示为输入规模的函数, 以此来分析算法的效率。

对于许多算法, 可以轻松找到一个合理的度量, 用来表示输入的大小, 我们称之为输入规模 (input size)。例如, 考虑算法 1.1 (顺序查找)、算法 1.2 (数组元素求和)、算法 1.3 (交换排序) 和算法 1.5 (二分查找)。在所有这些算法中, 数组中的项数  $n$  都是输入大小的一个简单度量。因此, 可以将其称为输入规模。在算法 1.4 (矩阵乘法) 中, 行列数  $n$  是输入大小的一个简单度量。因此, 仍然可以将  $n$  称为输入规模。在一些算法中, 更合适的做法是用两个数字来度量输入规模。例如, 当算法的输入是一幅图时, 经常用顶点数和边数来度量输入的大小。因此, 我们说输入规模由这两个参数组成。

有时, 在将一个参数称为“输入规模”时, 一定要非常小心。比如, 在算法 1.6 (斐波那契序列的第  $n$  项, 递归) 和算法 1.7 (斐波那契序列的第  $n$  项, 迭代) 中, 你可能认为应当将  $n$  称为输入规模。但是,  $n$  是输入, 不是输入的规模。对于这一算法, 输入规模的合理度量是对  $n$  编码所用的符号数。如果采用二进制表示法, 则输入规模应当是对  $n$  编码所用的比特数, 即  $\lceil \lg n \rceil + 1$ 。例如,

$$n = 13 = \underbrace{1101_2}_{4 \text{ 个比特}}$$

因此, 输入  $n=13$  的规模为 4。当时, 我们将这两种算法分别计算的项数表示为  $n$  的函数, 从而深入了解了它们的相对效率, 但  $n$  仍然没有度量输入的规模。这些考虑事项在第 9 章将变得非常重要, 届时将会更详细地讨论输入规模。在此之前, 以一种简单度量(比如数组中的项数)作为输入规模, 通常就足够了。

在确定了输入规模后, 我们选择某条指令或某组指令, 使算法完成的总工作量大体与这条指令或这组指令的执行次数成正比。我们将这条指令或这组指令称为算法的基本运算。例如, 在算法 1.1 和算法 1.5 的每次循环中, 都会将  $x$  与一个项目  $S$  进行比较。因此, 在这两个算法中, 比较指令都非常适合选为基本运算。我们通过针对每个  $n$  值, 计算出算法 1.1 和算法 1.5 执行这一基本运算的次数, 从而深入了解了这两种算法的相对效率。

一般情况下, 算法的时间复杂度分析 (time complexity analysis) 就是针对输入规模的每个取值, 计算该算法执行了多少次基本运算。尽管我们不希望考虑算法的实现细节, 但通常还是假定基本运算的实现是尽可能高效的。例如, 我们假定在算法 1.5 的实现中, 比较运算在每遍 while 循环中仅执行一次。这样, 我们分析的就是基本运算的最高效实现。

基本运算的选择并没有什么一成不变的规则, 大多由个人判断力和经验决定。前面已经提到, 我们一般不会包含构成控制结构的指令。例如, 算法 1.1 中就没有包含为了控制 while 循环而对索引值进行递增和比较的指令。有时, 完全可以将整个循环看作基本运算。而在另一个极端, 对于非常详尽的分析, 可能将每条机器指令看作基本运算。前面曾经提到, 因为我们希望分析结果与计算机无关, 所以本书不会采用这一做法。

有时可能希望考虑两种不同的基本运算。例如, 在通过比较键来完成排序的算法中, 经常希望将比较指令和赋值指令分别看作基本运算。这并不是说这两条指令一起构成了基本运算, 而是有两个不同的基本运算, 一个是比较指令, 另一个是赋值指令。之所以要这样做, 是因为排序算法完成的比较次数和赋值次数通常是不相等的。因此, 分别计算出这两种指令的完成次数, 可以更深入地了解算法的效率。

回想一下, 算法的时间复杂度分析就是针对每个输入规模值, 计算基本运算的执行次数。在某些情况下, 基本运算的执行次数不仅取决于输入规模, 还与输入值有关。算法 1.1 (顺序查找) 就是这种情况。例如, 如果  $x$  是数组中的第一项, 那基本运算仅执行一次, 但如果  $x$  不在数组中, 它就会执行  $n$  次。而在其他情况下, 比如在算法 1.2 (数组元素求和) 中, 确定输入规模  $n$  的取值后, 基本运算的执行次数都是一样的。在这种情况下,  $T(n)$  定义为: 对于输入规模  $n$  的一个具体取值, 该算法执行基本运算的次数。 $T(n)$  称为算法的所有情况时间复杂度 (every-case time complexity), 对  $T(n)$  的计算称为所有情况时间复杂度分析。所有情况时间复杂度分析的例子如下。

**算法 1.2 的分析 所有情况时间复杂度（数组元素求和）**

除控制指令之外，循环中的唯一指令就是将数组中每个项目加到 sum 的指令。因此，将这条指令称为基本运算。

**基本运算：**将数组中的一个项目加到 sum。

**输入规模：** $n$ ，数组中的项目数。

无论数组中的数字取何值，for 循环的执行遍数均为  $n$ 。因此，基本运算总是执行  $n$  次， $T(n) = n$ 。

**算法 1.3 的分析 所有情况时间复杂度（交换排序）**

前面曾经提到，在通过键的比较来完成排序的算法中，可以将比较指令或赋值指令看作基本运算。这里将分析比较指令的次数。

**基本运算：** $S[j]$  与  $S[i]$  的比较。

**输入规模：** $n$ ，要排序的项数。

必须求出要执行多少遍 for- $j$  循环。对于一个给定的  $n$  值，总是执行  $n-1$  遍 for- $i$  循环。在 for- $i$  循环的第一遍执行中，for- $j$  循环执行  $n-1$  遍，在第二遍执行中，for- $j$  循环执行  $n-2$  遍，在第三遍执行中，for- $j$  循环执行  $n-3$  遍……在最后一遍执行中，for- $j$  循环执行 1 遍。因此，for- $j$  循环的总执行遍数给出如下：

$$T(n) = (n-1) + (n-2) + (n-3) + \dots + 1 = \frac{(n-1)n}{2}$$

最后一个等式在附录 A 的例 A.1 中推导。

**算法 1.4 的分析 所有情况时间复杂度（矩阵乘法）**

最内层 for 循环中的唯一指令就是执行乘法和加法的指令。不难看出，可以采用某种方式来实现该算法，使加法运算的执行次数少于乘法运算。因此，我们仅将乘法指令看作基本运算。

**基本运算：**最内层 for 循环中的乘法指令。

**输入规模：** $n$ ，行列数。

for- $i$  循环总是执行  $n$  遍，在每遍执行中，总是执行  $n$  遍 for- $j$  循环，在 for- $j$  循环的每遍执行中，总会执行  $n$  遍 for- $k$  循环。因为基本运算在 for- $k$  循环内部，所以

$$T(n) = n \times n \times n = n^3$$

前文讨论过，在算法 1.1 中，基本运算的执行次数并非对于所有  $n$  值都是相同的。因此，这一算法没有得到所有情况的时间复杂度。对于许多算法都是如此。但是，这并不意味着就不能分析此类算法，因为还有其他三种分析方法可供尝试。第一种是考虑基本运算的最大执行次数。对于给定算法， $W(n)$  的定义为：当输入规模为  $n$  时，该算法执行其基本运算的最大次数。所以  $W(n)$  称为算法的最差情况时间复杂度（worst-case-time complexity）， $W(n)$  的计算称为最差情况时间复杂度分析。如果  $T(n)$  存在，显然  $W(n)=T(n)$ 。下面是当  $T(n)$  不存在时对  $W(n)$  的分析。

**算法 1.1 的分析 最差情况时间复杂度（顺序查找）**

**基本运算：**将数组中的一个项目与  $x$  进行比较。

**输入规模：** $n$ ，数组中的项目数。

基本运算最多执行  $n$  次，当  $x$  是数组中的最后项目或者当  $x$  不在数组时就是这种情况。因此， $W(n) = n$ 。

最差情况分析给出的是绝对的最大耗费时间量，但在某些情况下，可能想知道算法的平均运行情况。对于一种给定算法， $A(n)$  定义为：对于输入规模  $n$ ，该算法执行基本运算的平均次数，也就是均值（期望值）（关于均值的讨论请参阅附录 A 中的 A.8.2 节）。 $A(n)$  称为算法的平均情况时间复杂度（average-case time complexity）， $A(n)$  的计算称为平均情况时间复杂度分析。和  $W(n)$  一样，如果存在  $T(n)$ ，则  $A(n)=T(n)$ 。

为计算  $A(n)$ ，需要为所有规模为  $n$  的不同输入指定相应概率。基于所有可用信息来指定概率是非常重要的。例如，下面的分析是对算法 1.1 的平均情况分析。我们将会假定，如果  $x$  在数组中，则它位于数组中任一位置的概率相等。如果只知道  $x$  可能位于数组中的某一位置，那就没有理由认为某一数组位置会优先于另一位置。

因此，为所有数组位置指定相同概率是合理的。这就是说，我们正在计算的是对所有项目执行相同次查找时所需要的平均查找时间。如果有信息表明不会出现服从这种分布的输入，那就不应当在分析中使用这一分布。例如，如果数组中包含一些人的名字，而我们正在查找从所有美国人口中随机选出的名字，那对常见名“John”所在数组位置的查找频率可能要高于对罕见名字“Felix”所在位置的查找频率（关于随机性的讨论，请参阅附录A中的A.8.1节）。我们不应忽略这一信息，再假定所有位置都是等概率的。

下面的分析表明，平均情况的分析通常要难于最差情况的分析。

#### 算法 1.1 的分析 平均情况时间复杂度（顺序查找）

**基本运算：**将数组中的一个项目与  $x$  进行比较。

**输入规模：** $n$ ，数组中的项数。

首先分析已知  $x$  在  $S$  中的情景， $S$  中的项目各不相同，而且没有理由认为  $x$  在某个数组位置的可能性要高于另一位置。基于这一信息，对于  $1 \leq k \leq n$ ， $x$  在第  $k$  个数组位置的概率为  $1/n$ 。如果在第  $k$  个数组位置中，为找出  $x$  位置（然后退出循环）而执行的基本运算次数为  $k$ 。也就是说，平均情况时间复杂度为：

$$A(n) = \sum_{k=1}^n \left( k \times \frac{1}{n} \right) = \frac{1}{n} \times \sum_{k=1}^n k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

这个四重等式的第三步在附录 A 中的例 A.1 中推导。不出所料，平均来说，将查找数组大约一半的位置。

下面分析  $x$  可能不在数组中的情景。为分析这一情景，必须为“ $x$  在数组中”这一事件指定某一概率  $p$ 。如果  $x$  在数组中，则再次假设它以相同概率分布在从 1 至  $n$  的位置中。因此， $x$  位于第  $k$  个位置的概率就是  $p/n$ ，不在数组中的概率为  $1-p$ 。回想一下，如果在第  $k$  个位置找到了  $x$ ，则执行  $k$  遍循环；如果  $x$  不在数组中，则执行  $n$  遍循环。因此，平均情况时间复杂度为：

$$\begin{aligned} A(n) &= \sum_{k=1}^n \left( k \times \frac{p}{n} \right) + n(1-p) \\ &= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n \left( 1 - \frac{p}{2} \right) + \frac{p}{2} \end{aligned}$$

这一三重等式中的最后一步用代数运算推导得出。若  $p=1$ ，则  $A(n)=(n+1)/2$ ，与前面一样。若  $p=1/2$ ，则  $A(n)=3n/4+1/4$ 。这意味着，平均来说，将大约查找数组位置的  $3/4$ 。

在继续后续讨论之前，要给出一点有关“平均（值）”的提醒。尽管人们经常将“平均”作为典型情景提及，但在这样解读“平均”时一定要非常小心。例如，气象学者可能会说，在芝加哥，一个典型 1 月 25 日的最高温度为  $22^{\circ}\text{F}$ ，因为在过去 80 年里，这一天的平均最高温度就是  $22^{\circ}\text{F}$ 。报纸上可能发表一篇文章，说伊利诺伊州埃文斯通市一个典型家庭的年收入为 50 000 美元，因为这是平均收入。只有当实际情景不会与“平均值”偏离太多时（也就是说标准偏差很小时），才能说这个平均值是“典型值”。1 月 25 日的最高温度可能属于这一情景。但在埃文斯通市，各个家庭的收入相差很大。一个家庭年收入为 20 000 美元或 100 000 美元可能要比 50 000 美元更为典型。回想上面的分析，当已知  $x$  在数组中时， $A(n)=(n+1)/2$ 。这并不是典型查找次数，因为介于 1 至  $n$  之间的所有查找次数都是同等典型的。这些考虑事项在涉及响应时间的算法中非常重要。例如，设想一个监测核电厂的系统。哪怕只有一种情景的响应时间极差，其结果也可能是灾难性的。因此，如果说平均响应时间为 3 秒，那非常重要的一点就是要知道，是因为所有响应时间都大约为 3 秒，还是因为大多数响应时间为 1 秒，而其中一部分为 60 秒。

最后一种时间复杂度分析是计算出基本运算的最小执行次数。对于一种给定算法， $B(n)$  的定义为，对于输入规模  $n$ ，该算法执行基本运算的最小次数。因此， $B(n)$  称为算法的最佳情况时间复杂度（best-case time complexity）， $B(n)$  的求解称为最佳情况时间复杂度分析。和  $W(n)$ 、 $A(n)$  的情景一样，如果  $T(n)$  存在，则  $B(n)=T(n)$ 。下面求算法 1.1 的  $B(n)$ 。

#### 算法 1.1 的分析 最佳情况时间复杂度（顺序查找）

**基本运算：**将数组中的一个项目与  $x$  进行比较。

输入规模： $n$ ，数组中的项目数。

因为  $n \geq 1$ ，所以至少要执行一遍循环。如果  $x=S[1]$ ，无论  $n$  值如何，都将执行一遍循环。因此， $B(n)=1$ 。

对于不存在所有情况时间复杂度的算法，我们执行最差情况分析和平均情况分析的频率要远高于最佳情况分析。平均情况分析的价值在于，它可以告诉我们，在对许多不同输入多次应用某一算法时，该算法将会耗用多长时间。比如，在反复使用一种排序算法对所有可能输入进行排序时，这一分析就非常有用。一般情况下，如果平均排序时间很不错，那偶尔一次的慢速排序是可以忍受的。在 2.4 节将会看到一种名为“快速排序”的算法，它的表现恰好就是这种情况。它是最流行的排序算法之一。前面曾经提到，平均情况分析对于监测核电厂的系统来说是不够的。在这种情况下，最差情况分析会更有用一点，因为它会给出算法耗费时间的上限。对于上面刚刚讨论的这两种情况来说，最佳情况分析都没有什么价值。

我们只讨论了对算法时间复杂度的分析。刚刚讨论的所有考虑事项同样适用于内存复杂度（memory complexity）的分析，内存复杂度分析算法在内存方面的效率。尽管本书中的大多数分析为时间复杂度分析，但我们偶尔会发现，进行内存复杂度分析也很有用。

一般情况下，复杂度函数（complexity function）可以是任何一个将正整数映射为非负实数的函数。如果对于某一具体算法，没有说明是时间复杂度还是内存复杂度，通常会使用标准函数符号来表示复杂度函数，比如  $f(n)$  和  $g(n)$ 。

**例 1.6** 下面的函数都是复杂度函数，因为它们都将正整数映射为非负实数。

$$\begin{aligned}f(n) &= n \\f(n) &= n^2 \\f(n) &= \lg n \\f(n) &= 3n^2 + 4n\end{aligned}$$

### 1.3.2 理论应用

在应用算法分析的理论时，有时需要掌握以下信息：在实现该算法的实际计算机上执行基本运算、开销指令和控制指令所耗费的时间。“开销指令”是指诸如循环之前的初始化指令。这些指令的执行次数并不会随输入规模的增加而增加。“控制指令”是指为控制循环而递增索引之类的指令。这些指令的执行次数会随输入规模的增加而增加。基本运算、开销指令和控制指令都是算法及算法实现的性质。它们不是问题的性质。这就是说，对于同一问题的两个不同算法，这些性质通常是不一样的。

假定同一问题有两种算法，第一种算法的所有情况时间复杂度为  $n$ ，第二种算法为  $n^2$ 。第一种算法的效率看起来更高一些。但假设给定一台计算机，它分别将两种算法的基本运算处理一遍，第一种算法耗费的时间是第二种算法的 1000 倍。这里所说的“处理”是指包含了执行控制指令的时间。因此，如果将第二种算法的基本运算处理一遍需要的时间为  $t$ ，那将第一种算法的基本运算处理一遍将需要  $1000t$ 。为简单起见，假设执行开销指令耗费的时间在两种算法中都可以忽略。这就是说，对于第一种算法来说，这台计算机处理  $n$  的一个实例所需要的时间为  $n \times 1000t$ ，而对于第二种算法则为  $n^2 \times t$ 。必须求解以下不等式，才能确定第一种算法在什么情况更为高效：

$$n^2 \times t > n \times 1000t$$

不等式两边同时除以  $nt$ ，得：

$$n > 1000$$

如果实际应用的输入规模从来不会超过 1000，那就应当实现第二种算法。在继续讨论之前应当指出，要准确判断某种算法何时快于另一算法，并不总是那么容易。有时必须使用近似方法来分析通过对比两种算法而得出的不等式。

回想一下，前面假设处理开销指令耗费的时间是可忽略的。如果事实并非如此，那在判断第一种方法何时更为高效时，这些指令也必须考虑在内。

### 1.3.3 正确性分析

本书中的“算法分析”是指时间或内存方面的效率分析。还有其他类型的分析。例如，可以分析算法的正确性：证明一种算法的确完成了它应当完成的任务。尽管我们有时会非正式地说明算法是正确的，有时也会加以证明，但关于正确性的详尽讨论，还请参阅 Dijkstra（1976 年）、Gries（1981 年）或 Kingston（1990 年）的文献。

## 1.4 阶

前面刚刚阐明，对于时间复杂度分别为  $n$  和  $n^2$  的两种算法，无论两种算法需要多长时间来处理基本运算，当  $n$  值足够大时，第一种算法的效率总是高于第二种。现在假定同一问题有两种算法，而且第一种算法的所有情况时间复杂度为  $100n$ ，第二种算法为  $0.01n^2$ 。根据上面刚刚给出的论述，可以证明，第一种算法的效率最终会高于第二种算法。例如，如果两种算法处理基本运算需要的时间相同，而且开销时间也大致相同，则当

$$0.01n^2 > 100n$$

时，第一种算法更为高效。

两边同除以  $0.01n$ ，得：

$$n > 10\,000$$

如果第一种算法处理基本运算的时间长于第二种算法，第一种算法最终还是会变得更为高效，只是  $n$  值要更大一些。

时间复杂度为  $n$  和  $100n$  的算法称为线性时间算法（linear-time algorithm），因为它们的时间复杂度是输入规模  $n$  的线性函数，而时间复杂度为  $n^2$  和  $0.01n^2$  的算法称为二次时间算法（quadratic-time algorithm），因为它们的时间复杂度是输入规模  $n$  的二次函数。这里有一条基本原理，即，任意线性时间算法的效率最终将高于任意二次时间算法。在算法的理论分析中，我们关心的是最终行为。下面将说明如何根据算法的最终行为对其进行分组。这样就可以轻松地判断一组算法的最终行为是否高于另一组算法。

### 1.4.1 阶的直观介绍

诸如  $5n^2$  和  $5n^2+100$  的函数称为纯二次函数（pure quadratic function），因为它们没有包含线性项，而诸如  $0.1n^2+n+100$  之类的函数则称为完全二次函数（complete quadratic function），因为它们包含有线性项。表 1-3 表明，二次项最终在函数中占据主导地位。也就是说，与这个二次项的取值相比，其他项目的取值最终将变得微不足道。因此，尽管这个函数不是纯二次函数，但可以将它划分为纯二次函数一类。这就是说，某个算法具有这种时间复杂度，就可以将其称为二次时间算法。从直观上来说，在为复杂度函数分类时，应当总可以扔掉低阶项。例如，似乎应当可以将  $0.1n^3+10n^2+5n+25$  划分为纯三次函数。后面马上就会严格证明是可以这样做的。首先让我们直观感受一下，看看如何划分复杂度函数。

表 1-3 二次项最终占主导地位

$n$	$0.1n^2$	$0.1n^2 + n + 100$
10	10	120
20	40	160
50	250	400
100	1000	1200
1000	100 000	101 100

所有可以划分为纯二次函数的复杂度函数构成一个集合，称为  $\Theta(n^2)$ ，其中  $\Theta$  是大写希腊字母“西塔”。如果一个函数是集合  $\Theta(n^2)$  的元素，就说该函数为  $n^2$  阶。例如，由于可以抛弃低阶项，所以

$$g(n) = 5n^2 + 100n + 20 \in \Theta(n^2)$$

这意味着  $g(n)$  是  $n^2$  阶的。作为一个更具体的例子，回想 1.3.1 节中，算法 1.3（交换排序）的时间复杂度由  $T(n)=n(n-1)/2$  给出。因为

$$\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

抛弃低阶项  $n/2$ ，可证明  $T(n) \in \Theta(n^2)$ 。

当一个算法的时间复杂度属于  $\Theta(n^2)$  时，将该算法称为二次时间算法或  $\Theta(n^2)$  算法，也称该算法是  $\Theta(n^2)$  的。交换排序是一种二次时间算法。

与此类似，可以用纯三次函数归类的复杂度函数集合称为  $\Theta(n^3)$ ，并称这个集合中的函数是  $n^3$  阶的，以此类推。我们将这些集合称为复杂度类别（complexity category）。下面是一些最常见的复杂度类别：

$$\Theta(\lg n) \quad \Theta(n) \quad \Theta(n \lg n) \quad \Theta(n^2) \quad \Theta(n^3) \quad \Theta(2^n)$$

根据这一排序，如果  $f(n)$  所在类别位于  $g(n)$  所属类别的左侧，则  $f(n)$  的曲线最终将低于  $g(n)$ 。图 1-3 画出了这些类别中一些最简单成员的曲线： $n$ ， $\lg n$ ， $n \lg n$ ，等等。表 1-4 给出了一些算法的执行时间，这些算法的时间复杂度由上述函数给出。这里做了一个简化假设：处理每种算法的基本运算需要 1 纳秒 ( $10^{-9}$  秒)。表中结果可能会让人感到惊讶。有人可能会想，只要一条算法不是指数时间算法，那应当就够了。但是，即使是二次时间算法，也需要 31.7 年的时间来处理一个输入规模为 10 亿的实例。而  $\Theta(n \lg n)$  算法处理这样一个实例只需要 29.9 秒。一般情况下，一个算法必须是  $\Theta(n \lg n)$  或更佳时，我们才能假定它可以在可容忍时间内处理极大实例。这并不是说，如果时间复杂度属于更高阶类别，那这种算法就没有用了。具有二次、三次甚至更高阶时间复杂度的算法，经常可以处理在许多应用中出现的实例。

表 1-4 具有给定时间复杂度的算法的执行时间

$n$	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 $\mu\text{s}^*$	0.01 $\mu\text{s}$	0.033 $\mu\text{s}$	0.10 $\mu\text{s}$	1.0 $\mu\text{s}$	1 $\mu\text{s}$
20	0.004 $\mu\text{s}$	0.02 $\mu\text{s}$	0.086 $\mu\text{s}$	0.40 $\mu\text{s}$	8.0 $\mu\text{s}$	1 $\text{ms}^\dagger$
30	0.005 $\mu\text{s}$	0.03 $\mu\text{s}$	0.147 $\mu\text{s}$	0.90 $\mu\text{s}$	27.0 $\mu\text{s}$	1 $\text{s}$
40	0.005 $\mu\text{s}$	0.04 $\mu\text{s}$	0.213 $\mu\text{s}$	1.60 $\mu\text{s}$	64.0 $\mu\text{s}$	18.3 $\text{分}$
50	0.006 $\mu\text{s}$	0.05 $\mu\text{s}$	0.282 $\mu\text{s}$	2.50 $\mu\text{s}$	125.0 $\mu\text{s}$	13 $\text{天}$
$10^2$	0.007 $\mu\text{s}$	0.10 $\mu\text{s}$	0.664 $\mu\text{s}$	10.00 $\mu\text{s}$	1.0 ms	$4 \times 10^{13}$ 年
$10^3$	0.010 $\mu\text{s}$	1.00 $\mu\text{s}$	9.966 $\mu\text{s}$	1.00 ms	1.0 s	
$10^4$	0.013 $\mu\text{s}$	10.00 $\mu\text{s}$	130.000 $\mu\text{s}$	100.00 ms	16.7 分	
$10^5$	0.017 $\mu\text{s}$	0.10 ms	1.670 ms	10.00 s	11.6 天	
$10^6$	0.020 $\mu\text{s}$	1.00 ms	19.930 ms	16.70 分	31.7 年	
$10^7$	0.023 $\mu\text{s}$	0.01 s	2.660 s	1.16 天	31 709 年	
$10^8$	0.027 $\mu\text{s}$	0.10 s	2.660 s	115.70 天	$3.17 \times 10^7$ 年	
$10^9$	0.030 $\mu\text{s}$	1.00 s	29.900 s	31.70 年		

\*  $1 \mu\text{s} = 10^{-6}$  秒。

†  $1 \text{ms} = 10^{-3}$  秒。

在结束这一讨论之前要强调一点：与仅知道阶数相比，准确地知道时间复杂度可以掌握更多信息。例如，回想一下前面讨论的假想算法，它们的时间复杂度为  $100n$  和  $0.01n^2$ 。如果需要相同的时间来处理两种算法的基本运算，执行其开销指令，那当实例小于 10 000 时，二次时间算法的效率要更高一些。如果应用中从来不需要超过此数值的实例，那就应当实现二次时间算法。如果只知道时间复杂度分别属于  $\Theta(n)$  和  $\Theta(n^2)$ ，那就无法掌握以上信息。这个例子中的系数属于极端情况，在实践中，它们通常不会这样极端。此外，有时很难准确得出时间复杂度。因此，有时会满足于仅求出阶数。

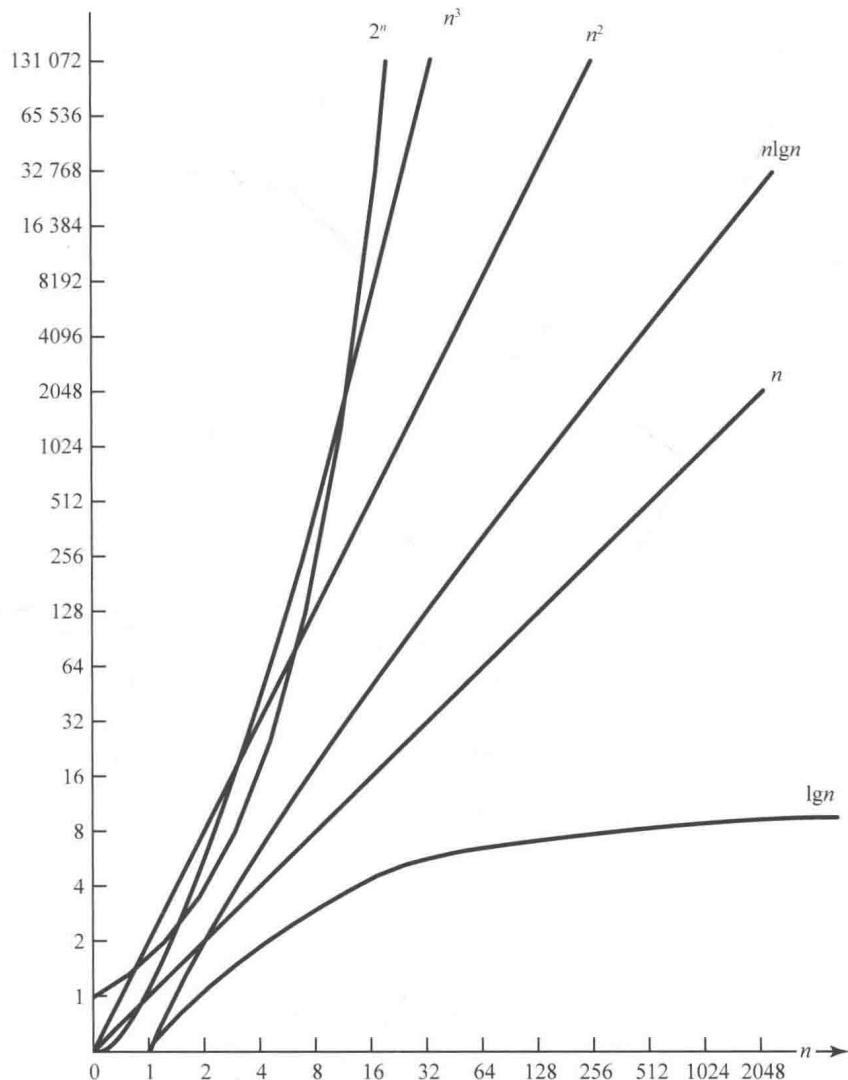


图 1-3 一些常见复杂度函数的增长速率

## 1.4.2 阶数的严谨介绍

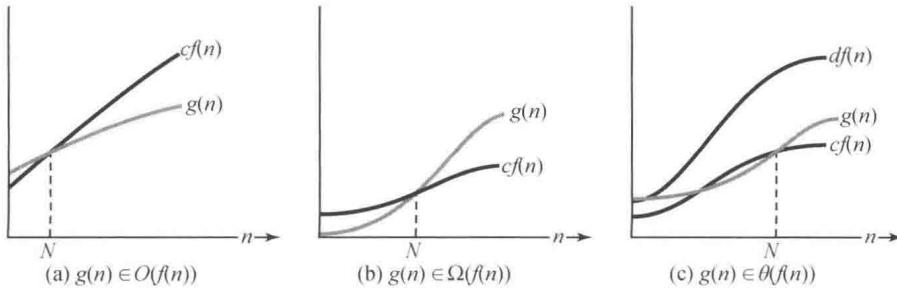
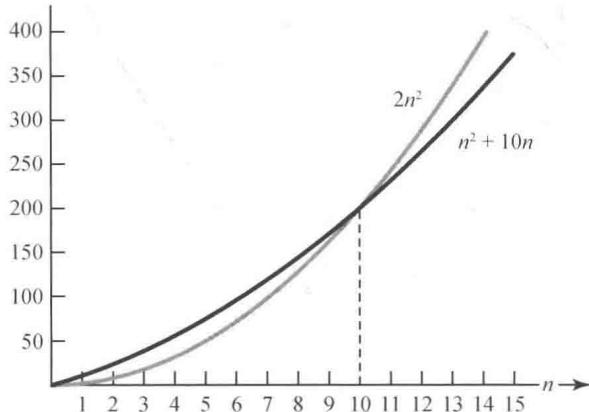
前面的讨论让我们对阶 ( $\Theta$ ) 有了直观的感受。下面将发展一种可以准确定义阶的理论。为此，给出另外两个基础概念。第一个是“大  $O$ ”。

**定义** 对于给定复杂度函数  $f(n)$ ,  $O(f(n))$  是由一些复杂度函数  $g(n)$  组成的集合, 对于其中每个  $g(n)$ , 必存在某一正实常数  $c$  及某一非负整数  $N$ , 使得对于所有  $n \geq N$ , 满足

$$g(n) \leq c \times f(n)$$

如果  $g(n) \in O(f(n))$ , 就说  $g(n)$  是  $f(n)$  的大  $O$ 。图 1-4a 显示了“大  $O$ ”。尽管在该图中,  $g(n)$  的起始位置高于  $c f(n)$ , 但最终落在  $c f(n)$  之下, 并保持这种状态。图 1-5 给出了一个具体例子。尽管  $n^2 + 10n$  在图中的起始位置高于  $2n^2$ , 但对于  $n \geq 10$ , 则有

$$n^2 + 10n \leq 2n^2$$

图 1-4 图解“大 O”、 $\Omega$  和  $\Theta$ 图 1-5 函数  $n^2+10n$  最终落在函数  $2n^2$  之下

因此可以在“大 O”的定义中取  $c=2$  和  $N=10$ ，得出结论：

$$n^2+10n \in O(n^2)$$

例如，如果  $g(n)$  属于  $O(n^2)$ ， $g(n)$  的曲线最终将位于某个纯二次函数  $cn^2$  之下。这就意味着，如果  $g(n)$  是某一算法的时间复杂度，那该算法的运算时间最终将至少与二次函数一样快。从分析的角度来说，可以说  $g(n)$  最终将会与纯二次函数一样好。我们说“大 O”（以及马上将会介绍的其他概念）描述了一个函数的渐近（asymptotic）特性，这是因为它们只与最终特性有关。我们说“大 O”设定了一个函数的渐近上限（asymptotic upper bound）。

下面的例子说明如何证明“大 O”。

**例 1.7** 证明  $5n^2 \in O(n^2)$ 。因为对于  $n \geq 0$ ，有

$$5n^2 \leq 5n^2$$

所以取  $c=5$ ,  $N=0$ ，即可得到所需要的结果。

**例 1.8** 回想一下，算法 1.3（交换排序）的时间复杂度由下式给出：

$$T(n) = \frac{n(n-1)}{2}$$

因为，对于  $n \geq 0$ ，有

$$\frac{n(n-1)}{2} \leq \frac{n(n)}{2} = \frac{1}{2}n^2$$

所以取  $c=1/2$ ,  $N=0$ ，可以得出结论  $T(n) \in O(n^2)$ 。

学生们在学习“大 O”时的难点在于，他们经常错误地认为必须要找到一个唯一的  $c$  和唯一的  $N$ ，才能证明一个函数是另一个函数的“大 O”。事实并非如此。回想一下，图 1-5 使用  $c=2$  和  $N=10$  证明了  $n^2+10n \in O(n^2)$ 。或者，也可以像下面这样证明。

例 1.9 证明  $n^2+10n \in O(n^2)$ 。因为对于  $n \geq 1$ , 有

$$n^2+10n \leq n^2+10n^2=11n^2$$

所以取  $c=11$ ,  $N=1$ , 可以得到想要的结果。

一般情况下, 可以使用看起来最直接的操作来证明 “大  $O$ ”。

例 1.10 证明  $n^2 \in O(n^2+10n)$ 。因为对于  $n \geq 0$ , 有

$$n^2 \leq 1 \times (n^2+10n)$$

所以取  $c=1$ ,  $N=0$ , 可以得到想要的结果。

这个例子是希望表明: “大  $O$ ” 中的函数不一定必须是图 1-3 中绘制的简单函数。它可以是任意复杂度函数。但一般情况下, 我们会取类似于图 1-3 中绘制的简单函数。

例 1.11 证明  $n \in O(n^2)$ 。因为对于  $n \geq 1$ , 有

$$n \leq 1 \times n^2$$

所以取  $c=1$ ,  $N=1$ , 可以得到想要的结果。

最后这个例子阐明了有关 “大  $O$ ” 的一个重要论述。 $O(n^2)$  中的复杂度函数并非一定要有二次项。只要它的曲线最终低于某个纯二次函数即可。因此, 任何对数或线性复杂度函数都属于  $O(n^2)$ 。同样, 任何对数、线性或二次复杂度函数都属于  $O(n^3)$ , 以此类推。图 1-6a 给出了  $O(n^2)$  的一些代表性元素。

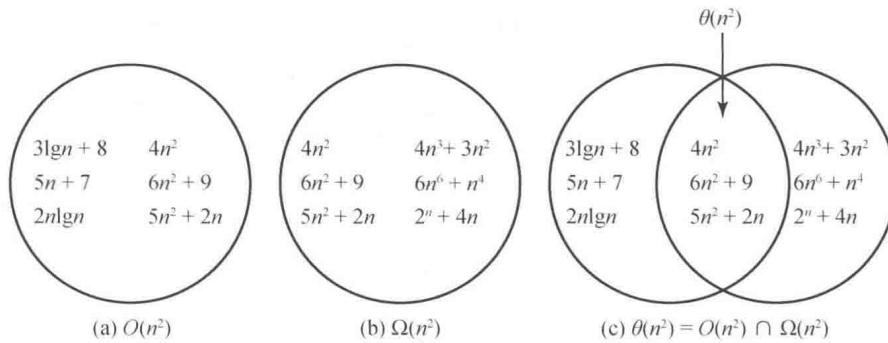


图 1-6 集合  $O(n^2)$ 、 $\Omega(n^2)$ 、 $\Theta(n^2)$ 。这里给出了一些代表性元素

“大  $O$ ” 为复杂度设定了渐近上限, 而下面的概念则为复杂度函数设定了渐近下限 (asymptotic lower bound)。

**定义** 对于给定复杂度函数  $f(n)$ ,  $\Omega(f(n))$  是由一些复杂度函数  $g(n)$  组成的集合, 对于其中每个  $g(n)$ , 必存在某一正实常数  $c$  及某一非负整数  $N$ , 使得对于所有  $n \geq N$ , 满足

$$g(n) \geq c \times f(n)$$

符号  $\Omega$  是大字希腊字母 “欧米伽”。如果  $g(n) \in \Omega(f(n))$ , 就说  $g(n)$  是  $f(n)$  的欧米伽。图 1-4b 演示了  $\Omega$ 。下面给出一些例子。

例 1.12 证明  $5n^2 \in \Omega(n^2)$ 。因为对于  $n \geq 0$ , 有

$$5n^2 \geq 1 \times n^2$$

所以取  $c=1$ ,  $N=0$ , 可以得到想要的结果。

例 1.13 证明  $n^2+10n \in \Omega(n^2)$ 。因为对于所有  $n \geq 0$ , 有

$$n^2+10n \geq n^2$$

所以取  $c=1$ ,  $N=0$ , 可以得到想要的结果。

例 1.14 再次考虑算法 1.3 (交换排序) 的时间复杂度。证明:

$$T(n) = \frac{n(n-1)}{2} \in \Omega(n^2)$$

对于  $n \geq 2$ , 有

$$n-1 \geq \frac{n}{2}$$

因此, 对于  $n \geq 2$ , 有

$$\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4}n^2$$

这就意味着, 取  $c=1/4$ ,  $N=2$ , 可以得到想要的结果。

和“大  $O$ ”中的情景一样, 满足  $\Omega$  定义条件的常数  $c$  和  $N$  并不唯一。可以选择使操作最容易的一对。

如果一个函数属于  $\Omega(n^2)$ , 该函数的曲线最终将位于某一纯二次函数之上。对分析来说, 这意味着它最终至少会像一个纯二次函数一样差。但是, 如下例所示, 这个函数不一定是二次函数。

**例 1.15** 证明  $n^3 \in \Omega(n^2)$ 。因为对于  $n \geq 1$ , 有

$$n^3 \geq 1 \times n^2$$

所以取  $c=1$ ,  $N=1$ , 可以得到想要的结果。

图 1-6b 给出了  $\Omega(n^2)$  的一些代表性元素。

如果一个函数同时属于  $O(n^2)$  和  $\Omega(n^2)$ , 可以得出结论: 这个函数的曲线最终将位于某一纯二次函数之下, 且又最终位于某一纯二次函数之上。也就是说, 它最终至少会像某个纯二次函数一样好, 且又最终至少会像某个纯二次函数一样差。因此可以得出结论, 它的增长速度类似于一个纯二次函数。这正是我们所想要的阶的严格概念的结果。有以下定义。

**定义** 对于一个给定的复杂度函数  $f(n)$ ,

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

这意味着  $\Theta(f(n))$  是由一些复杂度函数  $g(n)$  组成的集合, 对于其中每个  $g(n)$ , 必存在某正实常数  $c$  和  $d$ , 及某一非负整数  $N$ , 使得对于所有  $n \geq N$ , 满足

$$c \times f(n) \leq g(n) \leq d \times f(n)$$

若  $g(n) \in \Theta(f(n))$ , 就说  $g(n)$  是  $f(n)$  的阶 (order)。

**例 1.16** 再次考虑算法 1.3 的时间复杂度。例 1.8 和例 1.14 一同证明了

$$T(n) = \frac{n(n-1)}{2} \text{ 同时属于 } O(n^2) \text{ 和 } \Omega(n^2)。$$

这意味着  $T(n) \in O(n^2) \cap \Omega(n^2) = \Theta(n^2)$ 。

图 1-6c 显示  $\Theta(n^2)$  是  $O(n^2)$  和  $\Omega(n^2)$  的交集, 而图 1-4c 则显示了  $\Theta$ 。在图 1-6c 中注意  $5n+7$  不属于  $\Omega(n^2)$ , 函数  $4n^3+3n^2$  不属于  $O(n^2)$ 。因此, 这两个函数都不属于  $\Theta(n^2)$ 。尽管这在直觉上是正确的, 但我们还没有证明它。下面的例子说明如何进行此种证明。

**例 1.17** 用反证法 (矛盾法) 证明  $n$  不属于  $\Omega(n^2)$ 。在这种证明方法中, 假设某一论述为正确, 在本例中假设  $n \in \Omega(n^2)$ , 然后进行推导, 得出一个不正确的结果。也就是说, 此结果与某个已知正确的论述矛盾。于是, 可以得出结论, 前面做出的假设不可能是正确的。

假设  $n \in \Omega(n^2)$ , 也就是假设存在某个正常数  $c$  和某个非负整数  $N$ , 使得对于  $n \geq N$ , 有

$$n \geq cn^2$$

如果在此不等式两边除以  $cn$ , 则对于  $n \geq N$ , 有

$$\frac{1}{c} \geq n$$

但是，对于任意  $n > 1/c$ ，此不等式都不成立，这就意味着，它不可能对于所有  $n \geq N$  都成立。这一矛盾证明了  $n$  不属于  $\Omega(n^2)$ 。

还有另外一个有关“阶”的定义，它表达了诸如函数  $n$  与函数  $n^2$  之间的关系。

**定义** 对于一个给定的复杂度函数  $f(n)$ ， $o(f(n))$  是由所有满足以下条件的复杂度函数  $g(n)$  组成的集合：对于每个正实常数  $c$ ，必存在一个非负整数  $N$ ，使得对于所有  $n \geq N$ ，有

$$g(n) \leq c \times f(n)$$

如果  $g(n) \in o(f(n))$ ，就说  $g(n)$  是  $f(n)$  的小  $o$ 。回忆一下，“大  $O$ ”是指必然存在某个正实常数  $c$ ，使此上限成立。这一定义是说，该上限必须对于每个正实常数  $c$  成立。因为此上限对于每个正数  $c$  成立，所以它对任意小的  $c$  也成立。例如，如果  $g(n) \in o(f(n))$ ，则存在一个  $N$ ，使得当  $n > N$  时，

$$g(n) \leq 0.000\ 01 \times f(n)$$

可以看到，当  $n$  变得很大时， $g(n)$  相对于  $f(n)$  来说变得微不足道。对于分析来说，如果  $g(n)$  属于  $o(f(n))$ ，则  $g(n)$  最终将远优于诸如  $f(n)$  之类的函数。下面的例子说明了这一点。

### 例 1.18 证明

$$n \in o(n^2)$$

给定  $c > 0$ 。需要找出一个  $N$ ，使得对于  $n \geq N$ ，满足

$$n \leq cn^2$$

如果将此不等式的两边同除以  $cn$ ，得：

$$\frac{1}{c} \leq n$$

因此，选择任意  $N \geq 1/c$  就足够了。

注意， $N$  的值取决于常数  $c$ 。例如，若  $c = 0.000\ 01$ ，必须使  $N$  最小为 100 000。也就是说，对于  $n \geq 100\ 000$ ，有

$$n \leq 0.000\ 01n^2$$

**例 1.19 证明**  $n$  不属于  $o(5n)$ 。下面将使用反证法进行证明。令  $c = \frac{1}{6}$ 。若  $n \in o(5n)$ ，则必然存在某个  $N$  值，

使得对于  $n \geq N$ ，

$$n \leq \frac{1}{6}5n = \frac{5}{6}n$$

这一矛盾证明了  $n$  不属于  $o(5n)$ 。

下面的定理给出了“小  $o$ ”与其他渐近符号之间的关系。

**定理 1.2** 若  $g(n) \in o(f(n))$ ，则

$$g(n) \in O(f(n)) - \Omega(f(n))$$

即， $g(n)$  属于  $O(f(n))$ ，但不属于  $\Omega(f(n))$ 。

证明：因为  $g(n) \in o(f(n))$ ，所以对于每个正实常数  $c$ ，必存在一个  $N$ ，使得对于所有  $n \geq N$ ，有

$$g(n) \leq c \times f(n)$$

这意味着该上限当然对某个  $c$  值成立。因此，

$$g(n) \in O(f(n))$$

我们将使用反证法证明  $g(n)$  不属于  $\Omega(f(n))$ 。如果  $g(n) \in \Omega(f(n))$ ，则存在某一实常数  $c > 0$  和某个  $N_1$ ，使得对于所有  $n \geq N_1$ ，有

$$g(n) \geq c \times f(n)$$

但因为  $g(n) \in o(f(n))$ ，所以存在某个  $N_2$ ，使得对于所有  $n \geq N_2$ ，有

$$g(n) \leq \frac{c}{2} \times f(n)$$

但这些不等式必须对于所有大于  $N_1$  和  $N_2$  的  $n$  值都成立。这一矛盾证明了  $g(n)$  不可能属于  $\Omega(f(n))$ 。

你可能会想  $o(f(n))$  和  $O(f(n)) - \Omega(f(n))$  必然是同一集合。并非如此。有一些不常见的函数，属于  $O(f(n)) - \Omega(f(n))$ ，但不属于  $o(f(n))$ 。下例证实了这一点。

**例 1.20** 考虑下面的函数：

$$g(n) = \begin{cases} n & (n \text{ 为偶数}) \\ 1 & (n \text{ 为奇数}) \end{cases}$$

将以下证明留作练习：

$$g(n) \in O(n) - \Omega(n)，\text{ 但 } g(n) \text{ 不属于 } o(n)。$$

当然，例 1.20 是故意设计的。当复杂度函数表示实际算法的时间复杂度时， $O(f(n)) - \Omega(f(n))$  中的函数通常就是  $o(f(n))$  中的函数。

现在深入讨论  $\Theta$ 。在习题中我们证明了：

$$g(n) \in \Theta(f(n)) \text{ 等价于 } f(n) \in \Theta(g(n))$$

例如，

$$n^2 + 10n \in \Theta(n^2), \quad n^2 \in \Theta(n^2 + 10n)$$

这意味着  $\Theta$  将复杂度函数划分到不交集中。我们将这些集合称为复杂度类别。给定类别中的任何一个函数都可以表示该类别。为方便起见，通常用一个类别中最简单的元素来代表该类别。上一个示例类别由  $\Theta(n^2)$  表示。

某些算法的时间复杂度不随  $n$  递增。例如，回想一下，算法 1.1 的最佳情况时间复杂度  $B(n)$  对于所有  $n$  值都为 1。包含此类函数的复杂度类别可以用任意常数表示，为简便起见，我们用  $\Theta(1)$  表示它。

下面给出“阶”的一些重要性质，利用这些性质，可以轻松地确定许多复杂度函数的阶。这些性质直接给出，未加证明。一些证明会在习题中推导，而另外一些证明则可以由下一小节获得的结果推导得出。第二个结果已在前面讨论过。将它包含在这里只是出于完整性考虑。

### 阶的性质

- (1)  $g(n) \in O(f(n))$ ，当且仅当  $f(n) \in \Omega(g(n))$ 。
- (2)  $g(n) \in \Theta(f(n))$ ，当且仅当  $f(n) \in \Theta(g(n))$ 。
- (3) 若  $b > 1, a > 1$ ，则  $\log_a n \in \Theta(\log_b n)$ 。

这意味着所有对数复杂度函数都属于同一复杂度类别，我们用  $\Theta(\lg n)$  代表这一类别。

- (4) 若  $b > a > 0$ ，则  $a^n \in o(b^n)$ 。

这意味着所有指数复杂度函数都不在同一复杂度类别中。

- (5) 对于所有  $a > 0$ ，有  $a^n \in o(n!)$ 。

这意味着  $n!$  比任何指数复杂度函数都要差。

- (6) 思考复杂度类别的以下顺序：

$$\Theta(\lg n) \quad \Theta(n) \quad \Theta(n \lg n) \quad \Theta(n^2) \quad \Theta(n^j) \quad \Theta(n^k) \quad \Theta(a^n) \quad \Theta(b^n) \quad \Theta(n!)$$

其中  $k > j > 2, b > a > 1$ 。若复杂度函数  $g(n)$  所在类别位于  $f(n)$  所在类别的左侧，则

$$g(n) \in o(f(n))$$

(7) 若  $c \geq 0, d > 0, g(n) \in O(f(n))$ , 且  $h(n) \in \Theta(f(n))$ , 则

$$c \times g(n) + d \times h(n) \in \Theta(f(n))$$

例 1.21 性质 3 表明：所有对数复杂度函数都属于同一复杂度类别。例如，

$$\Theta(\log_4 n) = \Theta(\lg n)$$

这意味着  $\log_4 n$ 、 $\lg n$  之间的关系与  $7n^2 + 5n$ 、 $n^2$  之间的关系相同。

例 1.22 性质 6 表明：任意对数函数最终将优于任意多项式函数，任意多项式函数最终将优于任意指数函数，任意指数函数最终将优于阶乘函数。例如，

$$\lg n \in o(n), n^{10} \in o(2^n), 2^n \in o(n!)$$

例 1.23 性质 6 和性质 7 可重复使用。例如，可以证明  $5n + 3\lg n + 10n\lg n + 7n^2 \in \Theta(n^2)$ , 如下所示。重复应用性质 6 和性质 7, 有

$$7n^2 \in \Theta(n^2)$$

这意味着：

$$10n\lg n + 7n^2 \in \Theta(n^2)$$

这意味着：

$$3\lg n + 10n\lg n + 7n^2 \in \Theta(n^2)$$

这意味着：

$$5n + 3\lg n + 10n\lg n + 7n^2 \in \Theta(n^2)$$

在实践中，我们不会重复求助于这些性质，而是轻松地意识到可以抛弃低阶项。

如果可以得到一个算法的准确时间复杂度，只需抛弃低阶项目就能确定它的阶。如果无法获得这一信息，可以回过头来，借助于“大  $O$ ”和  $\Omega$  的定义来确定阶。例如，假定对于某些算法，无法准确求出  $T(n)$ [或  $W(n)$ ,  $A(n)$ ,  $B(n)$ ]。如果能够直接利用定义证明：

$$T(n) \in O(f(n)) \text{ 及 } T(n) \in \Omega(f(n))$$

就可以得出结论， $T(n) \in \Theta(f(n))$ 。

有时，很容易证明  $T(n) \in O(f(n))$ , 但难以确定  $T(n)$  是否属于  $\Omega(f(n))$ 。在这种情况下，我们可以满足于仅证明  $T(n) \in O(f(n))$ , 因为这意味着  $T(n)$  至少像  $f(n)$  这样的函数一样好。类似地，我们可以满足于仅知道  $T(n) \in \Omega(f(n))$ , 因为这意味着  $T(n)$  至少像  $f(n)$  这样的函数一样差。

在结束讨论之前需要提一下，许多作者会说

$$f(n) = \Theta(n^2), \text{ 而不说 } f(n) \in \Theta(n^2)$$

它们的意思实际是一样的，也就是说， $f(n)$  是集合  $\Theta(n^2)$  的一个元素。同样，人们经常会写为

$$f(n) = O(n^2), \text{ 而不是 } f(n) \in O(n^2)$$

关于“阶”的历史介绍，请参阅 Knuth (1973 年) 的文献；关于本书对阶定义的讨论，请参阅 Brassard (1985 年) 的文献。我们关于“大  $O$ ”、 $\Omega$  和  $\Theta$  的定义大多数是标准的，但“小  $o$ ”还有其他不同的定义方法。将  $\Theta(n)$ 、 $\Theta(n^2)$  等集合称为“复杂度类别”并不是标准做法。有些作者将它们称为“复杂度类”，尽管这一术语经常用来表示第 9 章讨论的问题集。而另外一些作者根本就没有为它们指定任何具体的名字。

### ④1.4.3 利用极限计算阶

下面说明在某些情况下如何用极限计算阶。这一部分是为熟悉极限和导数的读者准备的。本书其他部分并不需要这部分知识。

## 定理 1.3

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \begin{cases} c & \text{导出: 若 } c > 0, \text{ 则 } g(n) \in \Theta(f(n)) \\ 0 & \text{导出: } g(n) \in o(f(n)) \\ \infty & \text{导出: } f(n) \in o(g(n)) \end{cases}$$

证明: 此证明留作练习。

例 1.24 由定理 1.3 可导出:

$$\frac{n^2}{2} \in o(n^3)$$

这是因为

$$\lim_{n \rightarrow \infty} \frac{n^2 / 2}{n^3} = \lim_{n \rightarrow \infty} \frac{1}{2n} = 0$$

在例 1.24 中应用定理 1.3 并不是非常令人激动, 因为这一结果的直接证明也很简单。下面的例子更有趣一些。

例 1.25 由定理 1.3 可导出: 对于  $b > a > 0$ , 有

$$a^n \in o(b^n)$$

这是因为

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

因为  $0 < a/b < 1$ , 所以此极限为 0。

这是阶的性质 4 (在 1.4.2 节接近末尾处)。

例 1.26 由定理 1.3 可导出: 对于  $a > 0$ , 有

$$a^n \in o(n!)$$

若  $a \leq 1$ , 其结果是显而易见的。设  $a > 1$ 。若  $n$  很大, 使得:

$$\left\lceil \frac{n}{2} \right\rceil > a^4$$

则

$$\frac{a^n}{n!} < \frac{a^n}{\underbrace{a^4 a^4 \cdots a^4}_{\lceil n/2 \rceil \text{ 个}}} \leq \frac{a^n}{(a^4)^{n/2}} = \frac{a^n}{a^{2n}} = \left( \frac{1}{a} \right)^n$$

因为  $a > 1$ , 所以可导出:

$$\lim_{n \rightarrow \infty} \frac{a^n}{n!} = 0$$

这是阶的性质 5。

下面的定理扩展了定理 1.3 的用途, 其证明可在大多数微积分教科书中找到。

定理 1.4 (洛必达法则) 若  $f(x)$  和  $g(x)$  均可导, 导数分别为  $f'(x)$  和  $g'(x)$ , 且

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

那么只要  $\lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}$  存在, 则

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

定理 1.4 对于实值函数成立，而复杂度函数是整数值函数。但是，大多数复杂度函数（例如， $\lg n$ 、 $n$  等）也是实值函数。此外，如果函数  $f(x)$  是实变量  $x$  的函数，那么只要  $\lim_{x \rightarrow \infty} g(x)$  存在，则

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

其中  $n$  是整数。因此，定理 1.4 可用于复杂度分析，如下面的例子所示。

**例 1.27** 由定理 1.3 和定理 1.4 可推导出：

$$\lg n \in o(n)$$

这是因为

$$\lim_{x \rightarrow \infty} \frac{\lg x}{x} = \lim_{x \rightarrow \infty} \frac{d(\lg x)/dx}{dx/dx} = \lim_{x \rightarrow \infty} \frac{1/(x \ln 2)}{1} = 0$$

**例 1.28** 由定理 1.3 和定理 1.4 可推导出：对于  $b > 1$ ,  $a > 1$ ,

$$\log_a n \in \Theta(\log_b n)$$

这是因为

$$\lim_{x \rightarrow \infty} \frac{\log_a x}{\log_b x} = \lim_{x \rightarrow \infty} \frac{d(\log_a x)/dx}{d(\log_b x)/dx} = \frac{1/(x \ln a)}{1/(x \ln b)} = \frac{\ln b}{\ln a} > 0$$

这是阶的性质 3。

## 1.5 本书概要

我们现在已经为复杂算法的设计与分析做好了准备。本书大部分内容都是根据技术而非应用领域进行组织的。前面已经提到，这种组织方式的目的是为了建立一个方法库，在遇到新问题时，可从其中选择可能的解决方法。第 2 章讨论“分而治之”方法。第 3 章介绍动态规划，第 4 章讨论“贪婪方法”。第 5 章介绍回溯技术。第 6 章讨论一种与回溯有关的方法，名为“分支定界”。第 7 章和第 8 章从算法的设计与分析转为分析问题本身。这种分析称为计算复杂度分析，主要计算对于一个给定问题，其所有算法的时间复杂度下限。第 7 章分析排序问题，第 8 章分析查找问题。第 9 章专门讨论一种特殊类型的问题。对于这类问题，人们还从来没有开发出一种算法，使其时间复杂度在最差情况下能够优于指数函数。但也从来没有人能够证明不存在这种算法。事实表明，这类问题多达数千个，并且相互之间密切关联。此类问题的研究已经成为一个较新的、令人兴奋的计算机科学领域。第 10 章将再次回到算法的设计。但与第 2 章到第 6 章中给出的算法不同，我们将讨论解决一类特定问题的算法，也就是数论算法。这类算法解决的是涉及整数的问题。前 9 章讨论的所有算法都是为单处理器计算机设计的，这类计算机一次只能执行一个指令序列。由于计算机硬件价格的大幅降低，并行计算机近期的发展非常迅速。这些计算机有多个处理器，所有处理器可同时（并行）执行指令。为这类计算机编写的算法称为“并行算法”。第 11 章介绍此类算法。

## 1.6 习题

### 1.1 节

1. 编写一个算法，在一个包含  $n$  个数字的列表（数组）中找出最大数。
2. 编写一个算法，在一个包含  $n$  个数字的列表中找出最小数。
3. 有一个包含  $n$  个元素的集合，其元素存储在一个列表中。编写一个算法，以该列表为输入，输出该集合的所有三元素子集。
4. 编写一个“插入排序”算法（插入排序在 7.2 节讨论），使用二分查找法找出下一个插入位置。
5. 编写一个算法，计算两个整数的最大公约数。

6. 编写一个算法，在一个包含  $n$  个数字的列表中找出最大数和最小数。尝试找出一种方法，对数组项进行的比较次数不超过  $1.5n$  次。

7. 编写一个算法，确定一个准完全二叉树是否是一个堆。

### 1.2 节

8. 当需要查找操作时，在什么情况下不适合采用顺序查找（算法 1.1）？

9. 给出一个实际例子，你不会在其中应用交换排序（算法 1.3）来完成排序任务。

### 1.3 节

10. 为习题 1 到习题 7 中的算法定义基本运算，并研究这些算法的性能。若某一给定算法存在所有情况时间复杂度，则计算该复杂度。若不存在，则计算最差情况时间复杂度。

11. 为基本“插入排序”和习题 4 中给出的版本（它使用了二分查找）确定最差情况、平均情况和最佳情况时间复杂度。

12. 编写一个  $\Theta(n)$  算法，对  $n$  个互不相同的整数排序，这些整数的大小范围为 1 至  $kn$ （含），其中  $k$  是一个正整数常量。（提示：使用一个包含  $kn$  个元素的数组。）

13. 算法 A 执行  $10n^2$  次基本运算，而算法 B 执行  $300 \ln n$  次基本运算，从哪个  $n$  值开始，算法 B 开始呈现其较佳性能？

14. 对于一个输入规模为  $n$  的问题，有 Alg1 和 Alg2 两种算法。Alg1 的运行需要  $n^2$  微秒，Alg2 的运行需要  $100n \log n$  微秒。Alg1 的实现需要占用 4 小时的程序员时间，占用 2 分钟的 CPU 时间。而 Alg2 则需要 15 小时的程序员时间，6 分钟的 CPU 时间。如果程序员的报酬标准是 20 美元/小时，CPU 为 50 美元/分钟，要使用 Alg2 解决一个规模为 500 的问题实例，必须应用多少次后才能证明其开发成本的正当性。

### 1.4 节

15. 直接证明： $f(n)=n^2+3n^3 \in \Theta(n^3)$ 。即，使用  $O$  和  $\Omega$  的定义来证明  $f(n)$  同时属于  $O(n^3)$  和  $\Omega(n^3)$ 。

16. 利用  $O$  和  $\Omega$  的定义证明： $6n^2+20n \in O(n^3)$ ，但  $6n^2+20n \notin \Omega(n^3)$ 。

17. 利用 1.4.2 节给出的阶的性质，证明： $5n^5+4n^4+6n^3+2n^2+n+7 \in \Theta(n^5)$ 。

18. 设  $p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0$ ，其中  $a_k > 0$ 。利用 1.4.2 节给出的阶的性质，证明  $p(n) \in \Theta(n^k)$ 。

19. 函数  $f(x)=3n^2+10n \log n+1000n+4 \log n+9999$ ，属于下面哪个复杂度类别：

- (a)  $\theta(\lg n)$  (b)  $\theta(n^2 \log n)$  (c)  $\theta(n)$  (d)  $\theta(n \lg n)$  (e)  $\theta(n^2)$  (f) 都不是

20. 函数  $f(x)=(\log n)^2+2n+4n+\log n+50$  属于下面哪个复杂度类别：

- (a)  $\theta(\lg n)$  (b)  $\theta((\lg n)^2)$  (c)  $\theta(n)$  (d)  $\theta(n \lg n)$  (e)  $\theta(n(\lg n)^2)$  (f) 都不是

21. 函数  $f(x)=n+n^2+2^n+n^4$  属于下面哪个复杂度类别：

- (a)  $\theta(n)$  (b)  $\theta(n^2)$  (c)  $\theta(n^3)$  (d)  $\theta(n \lg n)$  (e)  $\theta(n^4)$  (f) 都不是

22. 按复杂度类别为以下函数分组：

$$\begin{aligned} & n \ln n \quad (\lg n)^2 \quad 5n^2+7n \quad n^{5/2} \quad n! \quad 2^{n!} \quad 4^n \quad n^n \quad n^n + \ln n \\ & 5^{\lg n} \quad \lg(n!) \quad (\lg n)! \quad \sqrt{n} \quad e^n \quad 8n+12 \quad 10^n+n^{20} \end{aligned}$$

23. 证明 1.4.2 节“阶的性质”中的性质 1、2、6、7。

24. 讨论渐近比较 ( $O$ 、 $\Omega$ 、 $\Theta$ 、 $o$ ) 的反射性、对称性和传递性。

25. 假定有一台计算机，需要 1 分钟的时间解决一个规模  $n=1000$  的问题实例。假定又购买了一台新计算机，其速度是原计算机的 1000 倍。假定所用算法具有下面的时间复杂度  $T(n)$ ，在 1 分钟内可以解决何种规模的实例？

- (a)  $T(n) = n$   
 (b)  $T(n) = n^3$   
 (c)  $T(n) = 10^n$

26. 推导定理 1.3 的证明过程。

27. 证明以下陈述的正确性。

- (a)  $\lg n \in O(n)$
- (b)  $n \in O(n \lg n)$
- (c)  $n \lg n \in O(n^2)$
- (d)  $2^n \in \Omega(5^{\lg n})$
- (e)  $\lg^3 n \in o(n^{0.5})$

### 补充习题

28. 目前, 使用算法 A 可以在 1 分钟中解决规模为 30 的问题实例, 此算法属于  $\Theta(2^n)$  算法。而我们很快就得在 1 分钟内解决两倍规模的问题实例。你认为购买一台更快 (也更贵) 的计算机是否会有帮助。

29. 考虑以下算法:

```
for (i = 1; i <= 1.5n; i++)
    cout << i;
for (i = n; i >= 1; i--)
    cout << i;
```

- (a) 当  $n=2, n=4, n=6$  时的输出结果是什么?
- (b) 时间复杂度  $T(n)$  是什么? 可以假定规模  $n$  可被 2 整除。

30. 考虑以下算法:

```
j = 1;
while (j <= n/2) {
    i = 1;
    while (i <= j) {
        cout << j << i;
        i++;
    }
    j++;
}
```

- (a) 当  $n=6, n=8, n=10$  时的输出结果是什么?
- (b) 时间复杂度  $T(n)$  是什么? 可以假定规模  $n$  可被 2 整除。

31. 考虑以下算法:

```
for (i = 2; i <= n; i++) {
    for (j = 0; j <= n) {
        cout << i << j;
        j = j + [n/4];
    }
}
```

- (a) 当  $n=4, n=16, n=32$  时的输出结果是什么?
- (b) 时间复杂度  $T(n)$  是什么? 可以假定规模  $n$  可被 4 整除。

32. 下面嵌套循环的时间复杂度  $T(n)$  是什么? 为简单起见, 可以假定  $n$  是 2 的幂。也就是说, 存在某一正整数  $k$ , 使得  $n=2^k$ 。

```
for (i = 1; i <= n; i++){
    j = n;
    while (j >= 1){
        < While 循环体> // 需要  $\Theta(1)$ 
        j = [j/2];
    }
}
```

33. 为以下问题给出一种算法, 并确定其时间复杂度。给定一个由  $n$  个不同正整数组成的列表, 将该列表分为两个大小均为  $n/2$  的子列表, 使两个子列表中的整数和之差最大。可以假定  $n$  是 2 的倍数。

34. 下面嵌套循环的时间复杂度是什么？为简单起见，可以假定  $n$  是 2 的幂。也就是说，对于某一正整数  $k$ ，有  $n=2^k$ 。

```
i = n;
while (i >= 1){
    j = i;
    while (j <= n){
        <while 循环体> // 需要  $\Theta(1)$ 
        j = 2 * j;
    }
    i = [i/2];
}
```

35. 考虑以下算法：

```
int addThem (int n, int A[])
{
    index i, j, k;

    j = 0;
    for (i = 1; i <= n; i++)
        j = j+A[i];
    k = 1;
    for (i = 1; i <= n; i++)
        k = k + k;
    return j + k;
}
```

- (a) 若  $n=5$ ，且数组  $A$  包含 2、5、3、7、8，输出为多少？
- (b) 该算法的时间复杂度  $T(n)$  为多少？
- (c) 尝试提高该算法的效率。

36. 考虑以下算法：

```
int anyEqual (int n, int A[][])
{
    index i, j, k, m;
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            for (k = 1; k <= n; k++)
                for (m = 1; m <= n; m++)
                    if (A[i][j]==A[k][m] && !(i==k && j==m))
                        return 1;
    return 0;
}
```

- (a) 该算法的最佳情况时间复杂度为多少（假定  $n>1$ ）？
- (b) 该算法的最差情况时间复杂度为多少？
- (c) 尝试提高该算法的效率。
- (d) 若该算法返回 0，数组  $A$  的哪种性质得以保持？
- (e) 若该算法返回 1，数组  $A$  的哪种性质得以保持？

37. 给定一个  $\Theta(\lg n)$  算法，计算当  $x^n$  除以  $p$  时的余数。为简单起见，可以假设  $n$  是 2 的幂。也就是说，对于某一正整数  $k$ ，有  $n=2^k$ 。

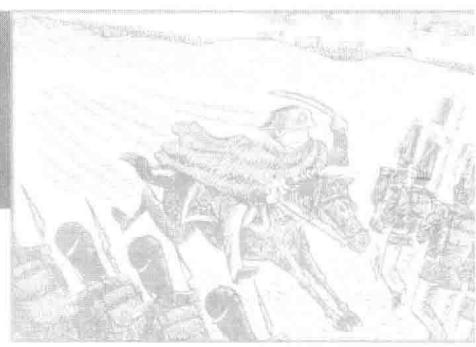
38. 用日常语言解释，以下集合中有哪些函数。

- (a)  $n^{O(1)}$
- (b)  $O(n^{O(1)})$
- (c)  $O(O(n^{O(1)}))$

39. 证明函数  $f(n)=|n^2 \sin n|$  既不属于  $O(n)$ , 也不属于  $\Omega(n)$ 。
40. 假定  $f(n)$  和  $g(n)$  是渐近正函数, 论证以下表述的正确性。
- (a)  $f(n)+g(n) \in O(\max(f(n), g(n)))$
  - (b)  $f^2(n) \in \Omega(f(n))$
  - (c)  $f(n)+o(f(n)) \in \Theta(f(n))$ , 其中  $o(f(n))$  表示任意属于  $o(f(n))$  的函数  $g(n)$
41. 给出下面问题的一个算法。给定一个由  $n$  个不同正整数组成的列表, 将该列表分为两个大小均为  $n/2$  的子列表, 使两个子列表中的整数和之差最小。计算该算法的时间复杂度。可以假定  $n$  是 2 的倍数。
42. 算法 1.7 (斐波那契序列的第  $n$  项, 迭代) 关于  $n$  显然是线性的, 但它是一种线性时间算法吗? 1.3.1 节将输入规模定义为输入的大小。在第  $n$  个斐波那契项的例子中,  $n$  是输入, 可以将用于对  $n$  编码的比特数用作输入规模。利用这一度量, 64 的规模为  $\lg 64=6$ , 1024 的规模为  $\lg 1024=10$ 。证明, 算法 1.7 根据输入规模属于指数时间算法。进一步证明, 任何计算第  $n$  个斐波那契项的算法都必然是一种指数时间算法, 因为输出的大小是输入规模的指数。(关于输入规模的相关讨论, 请参阅 9.2 节。)
43. 确定算法 1.6 (斐波那契序列的第  $n$  项, 递归) 关于其输入规模的时间复杂度(见习题 34)。
44. 能否验证习题 1~7 所得算法的正确性?

# 第 2 章

## 分而治之



我们的第一种算法设计方法——分而治之，模仿了法国皇帝拿破仑在 1805 年 12 月 2 日奥斯特里茨战役中采用的杰出策略。奥地利、俄罗斯联军的人数比拿破仑的军队多出大约 15 000 人。奥俄联军向法军右翼发动大规模进攻。拿破仑预见到这一进攻，他向联军中央部位发起冲击，将他们的军队一分为二。因为这两支较小的军队无法分别与拿破仑匹敌，均遭受严重损失，被迫撤退。拿破仑将一支大型军队分割（divide）为两支较小的军队，再分别治服这两支较小的军队，从而治服（conquer）整个大型军队。

分而治之（divide-and-conquer）方法将同一策略应用于问题的一个实例。也就是说，它将问题的一个实例划分为两个或更多个较小的实例。这些较小的实例通常也是原问题的实例。如果可以轻松获得较小问题实例的答案，那通过合并这些答案，就能得出原实例的答案。如果这些较小的实例还是太大，难以轻松解决，可以将它们划分为再小一些的实例。一直持续这一实例划分过程，直到其规模小到可以轻松获得答案为止。

分而治之方法是一种自顶向下（top-down）方法。也就是，通过向下获得较小实例的答案，以获得一个问题顶级实例的答案。读者可能发现，这就是递归例程使用的方法。回想一下，在编写递归时，人们在解决问题的级别进行思考，而让系统来处理在获取答案过程中的细节（通过栈的操作）。在设计分而治之算法时，我们通常就在这一级别思考，将它编写为递归例程。在此之后，我们有时可以为该算法设计一个更高效的迭代版本。

现在从二分查找入手，用示例来介绍分而治之方法。

### 2.1 二分查找

1.2 节给出了二分查找的一个迭代版本（算法 1.5）。这里给出一个递归版本，因为递归正好说明了分而治之方法所使用的自顶向下方法。用分而治之的术语来说，二分查找在一个有序（非递减顺序）数组中查找键  $x$  的位置时，首先将  $x$  与数组的中间项进行对比。如果它们相等，则算法完成。如果不相等，则将数组分为两个子数组，一个子数组中包含中间项目左侧的所有项目，另一个子数组包含其右侧的所有项目。如果  $x$  小于中间项，则将这一过程再应用于左侧子数组。否则，将其应用于右侧子数组。也就是将  $x$  与适当子数组的中间项进行对比。如果它们相等，则算法完成。如果不相等，则再将子数组划分为两个更小的数组。一直重复这一过程，直到找出  $x$ ，或者最终确认  $x$  不在数组中。

二分查找的步骤可总结如下。

如果  $x$  等于中间项，则退出。否则，

(1) 将数组划分为两个子数组，其大小大约为原数组的一半。如果  $x$  小于中间项，则选择左子数组；如果  $x$  大于中间项，则选择右子数组。

(2) 确定  $x$  是否在该子数组中，以攻克（解决）该子数组。如果子数组不够小，则进行递归处理。

(3) 由子数组的答案获得原数组的答案。

二分查找是最简单的分而治之算法，因为原实例仅被分解为一个较小实例，所以不存在输出结果的组合。原实例的答案就是较小实例的答案。下面的例子演示了二分查找。

**例 2.1** 假定  $x=18$ ，且有以下数组：

10	12	13	14	18	20	25	27	30	35	40	45	47
↑ 中间项												

(1) 划分数组：因为  $x < 25$ ，所以需要查找

10 12 13 14 18 20

(2) 确定  $x$  是否在该子数组中，以攻克该子数组。这一任务通过递归分解该子数组完成。其答案为：  
是， $x$  在此子数组中。

(3) 由此子数组的答案获得原数组的答案：

是， $x$  在此数组中。

步骤(2)中直接假定该子数组的答案是已知的。并没有讨论答案获取过程的细节，这是因为我们希望在解决问题的层面给出答案。在为问题设计递归算法时，需要：

- 找出一种方式，由一个或多个较小实例的答案获得原实例的答案；
- 确定较小实例逐步接近的终止条件；
- 在终止条件下确定答案。

我们不需要关心如何获得答案的细节（在计算机中，是由栈操作完成的）。实际上，操心这些细节有时会妨碍对复杂递归算法的开发。图 2-1 给出了人类在使用二分查找方法时执行的具体步骤。

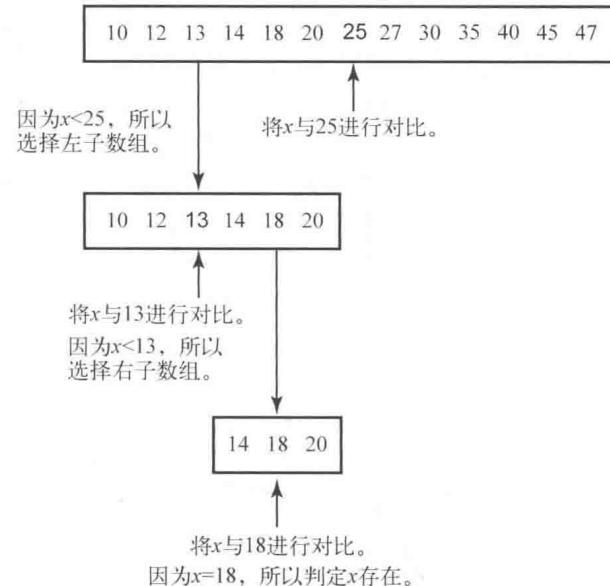


图 2-1 人类在用二分查找法进行查找时完成的步骤（注意： $x=18$ ）

现在给出二分查找的一个递归版本。

### 算法 2.1 二分查找（递归）

问题：判断  $x$  是否在一个大小为  $n$  的有序数组  $S$  中。

输入：正整数  $n$ ；键  $x$ ；有序（非递减顺序）键数组  $S$ ，索引范围为 1 到  $n$ 。

输出：location， $x$  在  $S$  中的位置（若  $x$  不在  $S$  中，则为 0）。

```

index location (index low, index high)
{
    index mid;

    if (low > high)
        return 0;
    else {
        mid = [(low + high)/2];
        if (x == S[mid])
    }
  
```

```

    return mid
else if (x < S[mid])
    return location(low, mid - 1);
else
    return location (mid + 1, high);
}
}

```

注意， $n$ 、 $S$  和  $x$  不是函数 `location` 的参数。因为它们在每次递归调用中保持不变，所以不需要将它们作为参数。在本书中，只有变量（其取值会在递归调用中变化）才会作为递归例程的参数。这样做的原因有两个。第一，降低递归例程表述的混乱程度。第二，在递归例程的实际实现中，每次递归调用都会为传送给例程的所有变量制作一份副本。如果一个变量的值不会改变，那这个副本就是不必要的。如果变量是一个数组，那这种浪费的成本可能会非常高昂。解决这种问题的一个办法是按地址传送变量。实际上，如果实现语言是 C++，数组是自动按地址传送的，使用保留字 `const` 可以保证数组不会被修改。但是，如果在递归算法的伪代码表达式中包含所有这些内容，会使表述变得混乱程度，可能会降低其清晰程度。

每种递归算法都可能用多种方式实现，具体取决于实现语言。例如，C++中的一种实现方式是将所有参数都传送给递归例程；另一种方法是使用类；还有一种方法是定义一些不会在递归调用中发生变化的全局参数。我们将演示如何实现最后一种方式，因为它与我们的算法表述一致。如果将  $S$  和  $x$  定义为全局变量， $n$  是  $S$  中的项数，算法 2.1 中对函数 `location` 的顶级调用将如下所示：

```
locationout = location (1, n);
```

因为二分查找的递归版本采用了尾递归（tail-recursion，也就是说，在递归调用之后不执行操作），所以可以很轻松地生成迭代版本，如 1.2 节所示。如前文所述，我们已经编写了一个递归版本，这是因为递归版本可以清楚地演示将一个实例分为较小实例的分而治之过程。但在诸如 C++ 之类的语言中，用迭代替换尾递归是有好处的。最重要的是，去掉递归调用生成的栈之后，会节省大量内存。回想一下，当一个例程调用另一例程时，有必要将第一个例程中尚未最终确定的结果压入活动记录的栈中，加以保存。如果第二个例程调用另一例程，第二个例程中尚未最终确定的结果也必须压入栈中，以此类推。当控制返回发出调用的例程时，它的活动记录从栈中弹出，计算出原来未最终确定的结果。在递归例程中，压入栈中的活动记录数由递归调用达到的深度决定。在二分查找法中，该栈在最差情况下达到的深度大约为  $\lg n + 1$ 。

用迭代替换尾递归的另一个原因是迭代算法的执行速度要快于递归版本（但仅相差一个常量乘法因子），因为前者不维护栈。因为大多数现代 LISP 方言都将尾递归编译为迭代代码，所以在这些方言中不需要将尾递归替换为迭代。

二分查找不存在所有情况时间复杂度。我们将进行最差情况分析。在 1.2 节我们已经非正式地进行了这一分析，现在将更严格地加以分析。尽管此分析针对的是算法 2.1，但也适用于算法 1.5。如果不熟悉求解递归方程的方法，在继续以下内容之前应当先学习附录 B。

### 算法 2.1 的分析 最差情况时间复杂度（二分查找，递归）

在查找数组的算法中，成本最高的运算通常是将查找项与数组项进行比较。因此，有以下信息。

**基本运算：**将  $x$  与  $S[mid]$  对比。

**输入规模：** $n$ ，数组中的项目数。

首先分析  $n$  为 2 的幂的情景。在对函数 `location` 的所有调用中，只要  $x$  不等于  $S[mid]$ ，就会将  $x$  与  $S[mid]$  进行两次比较。但是，在 1.2 节对二分查找的非正式分析中曾经讨论过，可以假定只有一次比较，因为在高效的汇编语言实现中就是这种情况。回想一下，1.3 节曾经假定，基本运算的实现是尽可能高效的。

如 1.2 节中之讨论，发生最差情况的一种方式就是  $x$  大于所有数组项。如果  $n$  是 2 的幂，而且  $x$  大于所有数组项目，那每次调用生成的实例恰好为原实例的一半。例如，如果  $n=16$ ，则  $mid=\lfloor(1+16)/2\rfloor=8$ 。因为  $x$  大于所有数组项，所有最顶端的八个项目将作为第一次递归调用的输入。同样，顶端四项将作为第二次递归调用的输入，以此类推。可得以下递推式：

$$W(n) = \underbrace{W\left(\frac{n}{2}\right)}_{\text{递归调用的比较}} + \underbrace{1}_{\text{顶级比较}}$$

若  $n=1$ , 且  $x$  大于这个唯一数组项目, 会将  $x$  与此项目进行比较, 后面是在  $\text{low} > \text{high}$  条件下的一次递归调用。这里, 终止条件为真, 这意味着没有更多比较。因此,  $W(1)$  为 1。我们已经确定了递推关系:

$$W(n) = W\left(\frac{n}{2}\right) + 1 \quad (n > 1, n \text{ 为 } 2 \text{ 的幂})$$

$$W(1) = 1$$

这一递推式的求解在附录 B 中完成。其答案为

$$W(n) = \lg n + 1$$

如果  $n$  不仅局限于 2 的幂, 则

$$W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$$

其中  $\lfloor y \rfloor$  表示小于或等于  $y$  的最大整数。我们将在习题中说明如何给出这一结果。

## 2.2 合并排序

与排序有关的一个过程是合并。我们用两路合并 (two-way merging) 表示将两个有序数组合并为一个有序数组。重复应用此合并过程, 可以完成对一个数组的排序。例如, 对于一个包含 16 个项目的数组, 可以将其分为两个大小各为 8 的子数组, 对这两个数组进行排序, 然后合并它们, 生成有序数组。同样, 可以将每个大小为 8 的子数组进一步划分为两个大小为 4 的子数组, 然后对这些子数组进行排序和合并。最终, 子数组的大小将变为 1, 大小为 1 的数组显然是有序的。这一过程称为“合并排序”。给定一个包含  $n$  个项目的数组 (为简单起见, 令  $n$  是 2 的幂), 合并排序涉及以下步骤。

- (1) 将数组划分为两个各包含  $n/2$  个项目的子数组。
- (2) 攻克 (解决) 每个子数组, 对其排序。除非数组足够小, 否则以递推方式完成此任务。
- (3) 将子数组合并为单个有序数组, 以合并这些子数组的答案。

下面的例子演示了这些步骤。

**例 2.2** 假定数组依次包含这些数字:

27 10 12 20 25 13 15 22

- (1) 划分以下数组:

27 10 12 20 和 25 13 15 22

- (2) 对每个子数组排序:

10 12 20 27 和 13 15 22 25

- (3) 合并子数组:

10 12 13 15 20 22 25 27

在步骤(2)中, 我们在解决问题的级别思考, 并假定子数组的答案已知。为了更具体一些, 图 2-2 演示了人类在使用合并排序法时完成的步骤。当数组大小变为 1 时, 出现终止条件; 此时, 合并开始。

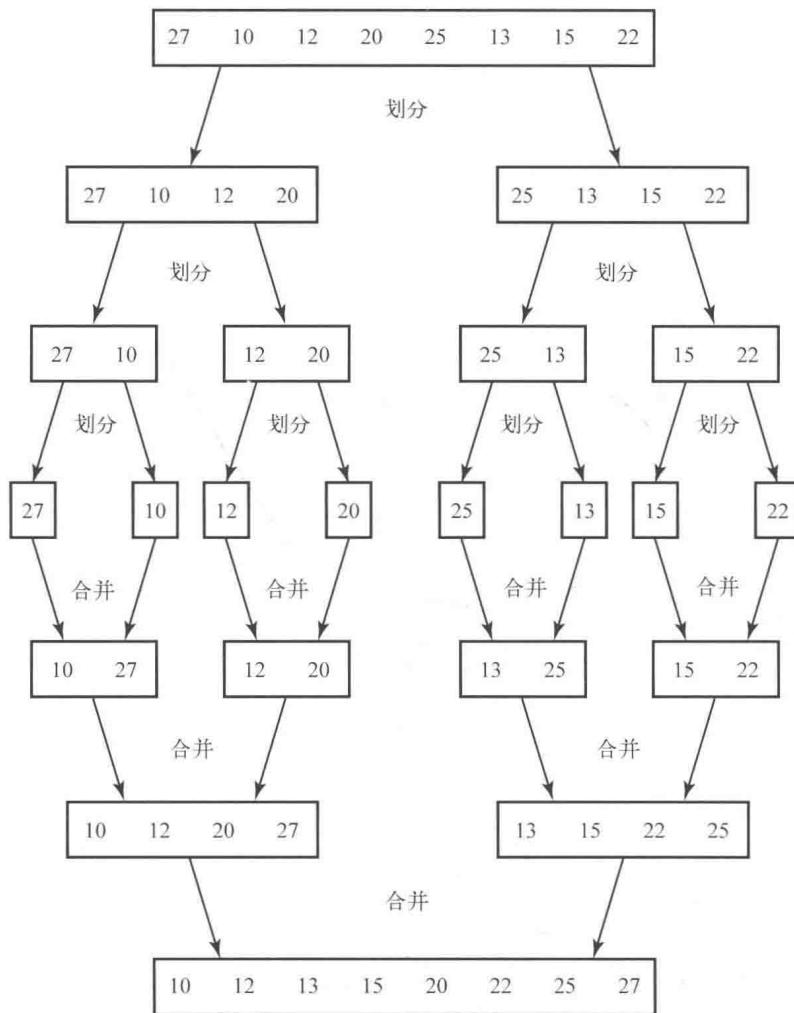


图 2-2 人类应用合并排序时执行的步骤

### 算法 2.2 合并排序

问题：将  $n$  个键排列为非递减顺序。

输入：正整数  $n$ ；键的数组  $S$ ，其索引范围为 1 至  $n$ 。

输出：数组  $S$ ，其中的键按非递减顺序排列。

```
void mergesort (int n, keytype S[])
{
    if (n>1) {
        const int h = [n/2], m = n - h;
        keytype U[1..h], V[1..m];
        将 S[1] 至 S[h] 复制到 U[1] 至 U[h];
        将 S[h+1] 至 S[n] 复制到 V[1] 至 V[m];
        mergesort(h, U);
        mergesort(m, V);
        merge(h, m, U, V, S);
    }
}
```

在分析“合并排序”算法之前，必须编写和分析一个可以合并两个有序数组的算法。

### 算法 2.3 合并

问题：将两个有序数组合并为一个有序数组。

输入：正整数  $h$  和  $m$ ；有序键数组  $U$ ，其索引范围为 1 至  $h$ ；有序键数组  $V$ ，其索引范围为 1 至  $m$ 。  
 输出：数组  $S$ ，其索引范围为 1 至  $h+m$ ，在单个有序数组中包含了  $U$  和  $V$  中的键。

```

void merge (int h, int m, const keytype U[],  

           const keytype V[],  

           keytype S[])
{
    index i, j, k;

    i = 1; j = 1; k = 1;
    while (i <= h && j <= m){
        if (U[i] < V[j]) {
            S[k] = U[i];
            i++;
        }
        else {
            S[k] = V[j];
            j++;
        }
        k++;
    }
    if (i>h)
        将 V[j]至 V[m]复制到 S[k]至 S[h+m];
    else
        将 U[i]至 U[h]复制到 S[k]至 S[h+m];
}

```

表 2-1 演示了在合并两个大小为 4 的数组时，过程 merge 是如何工作的。

表 2-1 将两个数组  $U$  和  $V$  合并为一个数组  $S$  的示例\*

$k$	$U$	$V$	$S$ (结果)
1	<b>10</b> 12 20 27	<b>13</b> 15 22 25	10
2	10 <b>12</b> 20 27	<b>13</b> 15 22 25	10 12
3	10 12 <b>20</b> 27	<b>13</b> 15 22 25	10 12 13
4	10 12 <b>20</b> 27	13 <b>15</b> 22 25	10 12 13 15
5	10 12 <b>20</b> 27	13 15 <b>22</b> 25	10 12 13 15 20
6	10 12 20 <b>27</b>	13 15 <b>22</b> 25	10 12 13 15 20 22
7	10 12 20 <b>27</b>	13 15 22 <b>25</b>	10 12 13 15 20 22 25
-	10 12 20 27	13 15 22 25	10 12 13 15 20 22 25 27 ←最终值

\*进行比较的项目以黑体表示。

### 算法 2.3 的分析 最差情况时间复杂度（合并）

1.3 节曾经提到，对于通过键的比较完成排序的算法，可以分别将比较指令和赋值指令看作基本运算。这里考虑比较指令。在第 7 章深入讨论合并排序时，将考虑赋值的个数。在这个算法中，比较次数依赖于  $h$  和  $m$ 。

**基本运算：**将  $U[i]$  与  $V[j]$  对比。

**输入规模：** $h$  和  $m$ ，两个输入数组中各自包含的项目数。

当循环退出时发生最差情况，因为索引之一（比如  $i$ ）已经到达其退出点  $h+1$ ，而另一个索引  $j$  已经到达  $m$ ，比其退出点小 1。例如， $V$  中的前  $m-1$  个项目首先被放入  $S$  中，然后放入  $U$  中的所有  $h$  个项目，此时，因为  $i$  等于  $h+1$ ，循环退出，于是就发生了上述情况。因此，

$$W(h, m) = h + m - 1$$

现在可以分析合并排序。

### 算法 2.2 的分析 最差情况时间复杂度（合并排序）

基本运算是 merge 过程中进行的比较。因为比较次数随  $h$  和  $m$  的增加而增加，而  $h$  和  $m$  随  $n$  的增加而增

加，所以有：

**基本运算：**merge 过程中进行的比较。

**输入规模：** $n$ ，数组  $S$  中的项目数。

总比较次数等于每次以  $U$  为输入对 mergesort 进行递归调用时发生的比较次数、以  $V$  为输入对 mergesort 进行递归调用时发生的比较次数、对 merge 进行顶级调用时发生的比较次数之和。因此，

$$W(n) = \underbrace{W(h)}_{\text{对 } U \text{ 排序的比较次数}} + \underbrace{W(m)}_{\text{对 } V \text{ 排序的比较次数}} + \underbrace{h+m-1}_{\text{合并中的比较次数}}$$

首先分析当  $n$  为 2 的幂时的情景。在这一情景中：

$$\begin{aligned} h &= \lfloor n/2 \rfloor = \frac{n}{2} \\ m &= n - h = n - \frac{n}{2} = \frac{n}{2} \\ h + m &= \frac{n}{2} + \frac{n}{2} = n \end{aligned}$$

$W(n)$  的表达式变为：

$$\begin{aligned} W(n) &= W\left(\frac{n}{2}\right) + W\left(\frac{n}{2}\right) + n - 1 \\ &= 2W\left(\frac{n}{2}\right) + n - 1 \end{aligned}$$

当输入大小为 1 时，满足终止条件，不再进行合并。因此， $W(1)$  为 0。得到以下递推式：

$$\begin{aligned} W(n) &= 2W\left(\frac{n}{2}\right) + n - 1 \quad (n > 1, n \text{ 为 } 2 \text{ 的幂}) \\ W(1) &= 0 \end{aligned}$$

这一递推式在附录 B 的例 B.19 中求解。其答案为：

$$W(n) = n \lg n - (n-1) \in \Theta(n \lg n)$$

当  $n$  不是 2 的幂时，将在习题中得出：

$$W(n) = W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1$$

其中， $\lceil y \rceil$  和  $\lfloor y \rfloor$  分别是不小于  $y$  的最小整数和不大于  $y$  的最大整数。由于存在下取整函数 ( $\lfloor \cdot \rfloor$ ) 和上取整函数 ( $\lceil \cdot \rceil$ )，很难准确分析这一情景。但是，利用诸如附录 B 中例 B.25 所用的归纳论证，可以证明  $W(n)$  是非递减的。因此，由该附录中的定理 B.4 可推出：

$$W(n) \in \Theta(n \lg n)$$

**原地排序** (in-place sort) 算法使用的空间仅限于输入内容的存储空间。算法 2.2 不是原地排序，因为除了输入数组  $S$  之外，它还使用了数组  $U$  和  $V$ 。如果  $U$  和  $V$  是 merge 中的变量参数 (按地址传送)，在调用 merge 时不会创建这些数组的第二副本。但是，在每次调用 mergesort 时都仍然会创建  $U$  和  $V$  的新数组。在最高级别，这两个数组的项目数之和为  $n$ 。在顶级递归调用中，两个数组中项目数之和大约为  $n/2$ ；在下一级的递归调用中，两个数组的项目数之和大约为  $n/4$ ；一般情况下，在每一递归级别，两个数组中项目数之和大约是上一级和值的一半。因此，额外创建的数组项总数大约为  $n(1+1/2+1/4+\dots)=2n$ 。

算法 2.2 清晰地演示了将一个问题的实例分解为较小实例的过程，因为实际上由输入数组 (原实例) 创建了两个新数组 (较小实例)。因此，这种方式非常适合介绍合并排序和演示分而治之方法。但是，有可能将额外空间缩减至仅有一个包含  $n$  个项目的数组。要实现这一目的，需要在输入数组  $S$  中完成大部分操作。下面的

做法类似于算法 2.1（二分查找，递归）中使用的方法。

#### 算法 2.4 合并排序 2

问题：将  $n$  个键排列为非递减顺序。

输入：正整数  $n$ ；键的数组  $S$ ，索引范围为 1 至  $n$ 。

输出：数组  $S$ ，其中包含按非递减顺序排列的键。

```
void mergesort2 (index low, index high)
{
    index mid;

    if (low < high){
        mid = [(low + high)/2];
        mergesort2(low, mid);
        mergesort2(mid + 1, high);
        merge2(low, mid, high);
    }
}
```

前面曾经约定，仅使变量（其取值会在递归调用中发生变化）成为递归例程的参数，根据这一约定， $n$  和  $S$  不是过程 mergesort2 的参数。如果在实现算法时将  $S$  定义为全局变量， $n$  是  $S$  中的数据项，则对 mergesort2 中的顶级调用如下：

```
mergesort2(1, n);
```

与 mergesort2 一起使用的合并过程如下。

#### 算法 2.5 合并 2

问题：合并 mergesort2 中创建的  $S$  的两个有序子数组。

输入：索引  $low$ 、 $mid$  和  $high$ ，以及  $S$  的子数组，其索引为  $low$  至  $high$ 。数组中从  $low$  至  $mid$  中的键已经排列为非递减顺序，数组中从  $mid + 1$  到  $high$  的键也是如此。

输出： $S$  的索引范围为  $low$  至  $high$  的子数组，其中包含了按非递减顺序排列的键。

```
void merge2 (index low, index mid, index high)
{
    index i, j, k;
    keytype U[low..high]; // 合并过程中需要的局部数组

    i = low; j = mid + 1; k = low;
    while (i ≤ mid && j ≤ high){
        if (S[i] ≤ S[j]){
            U[k] = S[i];
            i++;
        }
        else {
            U[k] = S[j];
            j++;
        }
        k++;
    }
    if (i > mid)
        将 S[j] 至 S[high] 移至 U[k] 至 U[high];
    else
        将 S[i] 至 S[mid] 移至 U[k] 至 U[high];
        将 U[low] 至 U[high] 移至 S[low] 至 S[high];
}
```

## 2.3 分而治之方法

在详细地研究了两种分而治之方法之后，现在应当更好地理解了这一方法的如下一般性描述。分而治之设计策略涉及以下步骤。

- (1) 将问题的一个实例划分为一个或多个较小实例。
- (2) 攻克每个较小实例。除非这些较小实例足够小，否则以递归方式完成这一任务。
- (3) 必要时，合并较小实例的答案，以获得原实例的答案。

步骤(3)中之所以要说“必要时”，是因为在诸如二分查找（递归，算法 2.1）之类的算法中，原实例被缩减为仅有一个较小实例，所以不需要合并答案。

下面给出分而治之方法的更多例子。在这些例子中，不会明确提及前面概括的步骤。应当可以清楚地看出，我们就是在遵循这些步骤。

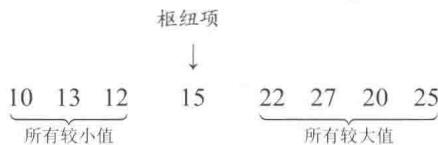
## 2.4 快速排序（分割交换排序）

现在来看一种名为“快速排序”的排序算法，它是由 Hoare 在 1962 年开发的。快速排序在某些地方与合并排序类似，它的完成过程也是将数组分解为两部分，然后分别对每一部分排序。但在快速排序中，在划分数组时，是将所有小于某个枢纽项的项目放到该项目之前，将所有大于该枢纽项的项目放到该项目之后。枢纽项可以是任意项目，为方便起见，直接选择第一个项目。下面的例子演示了快速排序的工作过程。

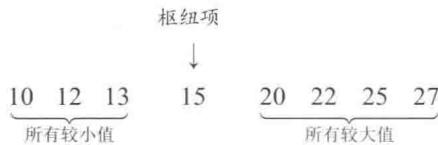
**例 2.3** 假定数组中依次包含以下数字：



(1) 划分数组，使所有小于枢纽项的项目都在该项目的左侧，所有大于它的项目都在其右侧：



(2) 对子数组排序：



在划分之后，子数组中的项目顺序并非人工指定，而是由划分方式的实施结果决定。我们根据划分例程（稍后给出）放置这些项目的方式排定了它们的顺序。重要的是，所有小于枢纽项的项目都在其左侧，所有大于它的项目都在其右侧。然后递归调用快速排序，分别对两个子数组排序。划分子数组，并持续这一过程，直到数组中只有一个项目为止。这样一个数组显然是有序的。例 2.3 给出了解题级别的答案。图 2-3 演示了人类在使用快速排序方法时完成的步骤。算法如下。

### 算法 2.6 快速排序

**问题：**将  $n$  个键排列为非递减顺序。

**输入：**正整数  $n$ ；键的数组  $S$ ，索引范围为 1 至  $n$ 。

**输出：**数组  $S$ ，其中包含按非递减顺序排列的键。

```
void quicksort (index low, index high)
{
```

```

index pivotpoint;

if (high > low){
    partition(low, high, pivotpoint);
    quicksort(low, pivotpoint - 1);
    quicksort(pivotpoint + 1, high);
}
}

```

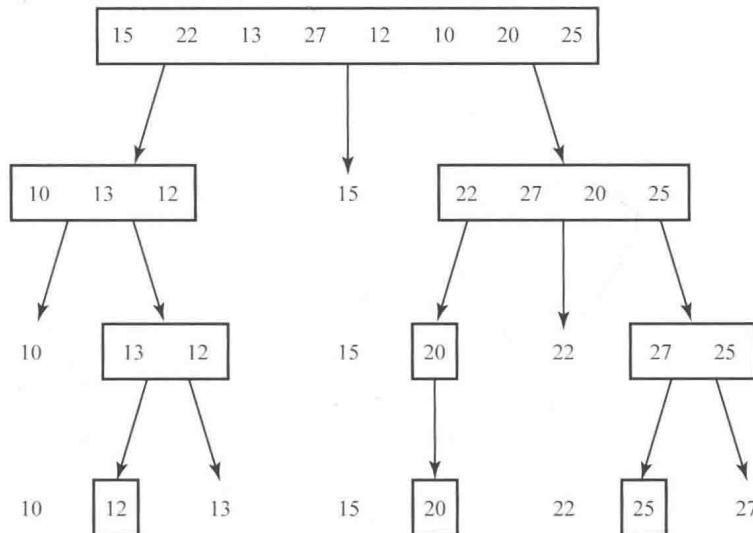


图 2-3 人类在执行快速排序时完成的步骤。子数组放在矩形中，而枢纽点则单独放置

根据我们的惯例， $n$  和  $S$  不是过程 quicksort 的参数。如果在实现算法时，将  $S$  定义为全局变量， $n$  是  $S$  中的项目数，则对 quicksort 的顶级调用将如下所示：

```
quicksort(1, n);
```

数组的划分由过程 partition 完成。下面给出这一过程的算法。

### 算法 2.7 分割

问题：分割用于快速排序的数组  $S$ 。

输入：两个索引， $low$  和  $high$ ;  $S$  的子数组，其索引为  $low$  至  $high$ 。

输出：pivotpoint，索引范围为  $low$  至  $high$  的子数组的枢纽点。

```

void partition (index low, index high,
                index& pivotpoint)
{
    index i, j;
    keytype pivotitem;

    pivotitem = S[low]; // 为 pivotitem 选择第一项
    j = low;
    for (i = low + 1; i <= high; i++)
        if (S[i] < pivotitem){
            i++;
            交换 S[i] 和 S[j];
        }
    pivotpoint = j;
    交换 S[low] 和 S[pivotpoint]; // 将 pivotitem 放在 pivotpoint
}

```

过程 partition 的工作方式是依次检查数组中的每个项目。只要发现一个小于枢纽项的项目，就将其移到数

组的左侧。表 2-2 表明了对于例 2.3 中的数组，partition 是如何进行的。

表 2-2 过程 partition 的一个例子\*

<i>i</i>	<i>j</i>	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	
-	-	15	22	13	27	12	10	20	25	←初始值
2	1	<b>15</b>	<b>22</b>	13	27	12	10	20	25	
3	2	<b>15</b>	22	<b>13</b>	27	12	10	20	25	
4	2	<b>15</b>	<b>13</b>	<b>22</b>	<b>27</b>	12	10	20	25	
5	3	<b>15</b>	13	22	27	<b>12</b>	10	20	25	
6	4	<b>15</b>	13	<b>12</b>	27	<b>22</b>	<b>10</b>	20	25	
7	4	<b>15</b>	13	12	<b>10</b>	22	<b>27</b>	<b>20</b>	25	
8	4	<b>15</b>	13	12	10	22	27	20	<b>25</b>	
-	4	<b>10</b>	13	12	<b>15</b>	22	27	20	25	←最终值

\*进行比较的项目以黑体表示。刚刚交换过的项目放在方框内。

接下来分析“分割”和“快速排序”。

**算法 2.7 的分析** 所有情况时间复杂度（分割）

**基本运算：**将  $S[i]$  与 pivotitem 进行比较。

**输入规模：**  $n = \text{high} - \text{low} + 1$ ，子数组中的项目数。

因为除第一项之外的每个项目都会进行比较，所以：

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

这里用  $n$  表示子数组的大小，而不是数组  $S$  的大小。只有在顶级调用 partition 时，它才表示  $S$  的大小。

快速排序没有所有情况复杂度。我们将分析最差情况和平均情况。

**算法 2.6 的分析** 最差情况时间复杂度（快速排序）

**基本运算：**partition 中将  $S[i]$  与 pivotitem 进行比较。

**输入规模：**  $n$ ，数组  $S$  中的项目数。

非常奇怪的是，当数组已经排列为非递减顺序时发生最差情况。其原因应当清楚了。如果数组已经排列为非递减顺序，那数组中就没有小于第一项的项目，而这一项正是枢纽项。因此，在顶级调用 partition 时，不会在枢纽项的左侧放置项目，partition 指定给 pivotpoint 的值是 1。与此类似，在每次递归调用中，pivotpoint 接受 low 的值。因此，数组被重复分割为左侧一个空子数组，右侧是只减少了一个项目的子数组。对于已经排列为非递减顺序的各个实例，有

$$T(n) = \underbrace{T(0)}_{\text{对左侧子数组排序的时间}} + \underbrace{T(n-1)}_{\text{对右侧子数组排序的时间}} + \underbrace{n-1}_{\text{分割的时间}}$$

我们使用了符号  $T(n)$ ，这是因为当前正在为已排列成非递减顺序的实例计算所有情况复杂度。因为  $T(0)=0$ ，所以有递推关系

$$\boxed{\begin{aligned} T(n) &= T(n-1) + n-1 \quad (n > 0) \\ T(0) &= 0 \end{aligned}}$$

这一递归方程在附录 B 的例 B.16 中求解。其答案为：

$$T(n) = \frac{n(n-1)}{2}$$

我们已经证明了最差情况为至少  $n(n-1)/2$ 。尽管直观上看来它就是所能达到的最差情况，但仍然需要证明这一点。为此，我们使用归纳法证明：对于所有  $n$ ，有

$$W(n) \leq \frac{n(n-1)}{2}$$

归纳基础：当  $n=0$  时，

$$W(0) = 0 \leq \frac{0(0-1)}{2}$$

归纳假设：假设对于  $0 \leq k < n$ ，有

$$W(k) \leq \frac{k(k-1)}{2}$$

归纳步骤：需要证明

$$W(n) \leq \frac{n(n-1)}{2}$$

对于一个给定  $n$  值，存在某一规模为  $n$  的实例，其处理时间为  $W(n)$ 。设  $p$  是处理这一实例时，在顶级调用 `partition` 时返回的 pivotpoint 值。因为在处理规模为  $p-1$  和  $n-p$  的实例时，其各自时间不可能超过  $W(p-1)$  和  $W(n-p)$ ，所以有

$$\begin{aligned} W(n) &\leq W(p-1) + W(n-p) + n - 1 \\ &\leq \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + n - 1 \end{aligned}$$

最后一个不等式是根据归纳假设得出的。经过代数运算可以证明：对于  $1 \leq p \leq n$ ，最后一个表达式为

$$\leq \frac{n(n-1)}{2}$$

这就完成了归纳证明。

我们已经证明了最差时间复杂度由下式给出：

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

当数组已经有序时会发生最差情况，因为我们总是选择第一个项目作为枢纽项。因此，如果有理由相信数组是接近有序的，那它就不适合作为枢纽项。在第 7 章深入讨论快速排序时，将会研究其他选择枢纽项的方法。如果使用这些方法，当数组已经有序时不会发生最差情况。但最差情况时间复杂度仍然是  $n(n-1)/2$ 。

在最差情况下，算法 2.6 不比交换排序（算法 1.3）快。那为什么将这种方法称为“快速排序”呢？后面将会看到，快速排序是因平均情况方面的表现而得名。

#### ◆ 算法 2.6 的分析 平均情况时间复杂度（快速排序）

基本运算：`partition` 中将  $S[i]$  与 `pivotitem` 进行比较。

输入规模： $n$ ，数组  $S$  中的项目数。

我们将假定，没理由认为数组中的数字具有任何特定顺序，因此 `partition` 返回的 pivotpoint 值取 1 至  $n$  中任意数字的概率相同。如果有理由取不同分布，那这一分析将不适用。因此，这里获得的均值是对每种可能顺序进行相同次数的排序时所得到的平均排序时间。在这种情况下，平均情况时间复杂度由以下递推式给出：

$$\begin{aligned} &\text{pivotpoint 为 } p \text{ 的概率} \\ &\downarrow \\ A(n) &= \sum_{p=1}^n \frac{1}{n} \left[ \underbrace{A(p-1) + A(n-p)}_{\substack{\text{当 pivotpoint 为 } p \text{ 时,} \\ \text{对子数组排序的平均时间}}} \right] + \frac{n-1}{n} \quad (2.1) \end{aligned}$$

在习题中将会证明：

$$\sum_{p=1}^n [A(p-1) + A(n-p)] = 2 \sum_{p=1}^n A(p-1)$$

将此等式代入等式 2.1 中, 得:

$$A(n) = \frac{2}{n} \sum_{p=1}^n A(p-1) + n - 1$$

两边乘以  $n$ , 得:

$$nA(n) = 2 \sum_{p=1}^n A(p-1) + n(n-1) \quad (2.2)$$

将等式 2.2 应用于  $n-1$ , 得:

$$(n-1)A(n-1) = 2 \sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2) \quad (2.3)$$

用等式 2.2 减去等式 2.3, 得:

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$$

化简为:

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

如果令

$$a_n = \frac{A(n)}{n+1}$$

得以下递归方程:

$$\begin{aligned} a_n &= a_{n-1} + \frac{2(n-1)}{n(n+1)} \quad (n>0) \\ a_0 &= 0 \end{aligned}$$

和附录 B 中例 B.22 中的递归方程类似, 这一递推方式的近似答案为:

$$a_n \approx 2 \ln n$$

由此可推出:

$$\begin{aligned} A(n) &\approx (n+1)2 \ln n = (n+1)2(\ln 2)(\lg n) \\ &\approx 1.38(n+1)\lg n \in \Theta(n \lg n) \end{aligned}$$

快速排序的平均情况时间复杂度与合并排序的时间复杂度同阶。第 7 章及 Knuth (1973 年) 的文献对合并排序和快速排序进行了深入对比。

## 2.5 Strassen 矩阵乘法算法

回想一下, 算法 1.4 (矩阵乘法) 是严格根据矩阵乘法的定义计算两个矩阵的乘积。我们已经证明了其乘法次数的时间复杂度由  $T(n)=n^3$  给出, 其中  $n$  是矩阵中的行列数。还可以分析加法的次数。在习题中将会看出, 对此算法稍做修改后, 加法次数的时间复杂度为  $T(n)=n^3-n^2$ 。因为这两个时间复杂度都是  $\Theta(n^3)$ , 所以这一算法很快就会变得无法实际应用。1969 年, Strassen 发表了一种算法, 其时间复杂度在乘法和加/减法方面都优于三次函数。下面的例子演示了他的方法。

**例 2.4** 假定要计算两个  $2\times 2$  矩阵  $A$  和  $B$  的乘积  $C$ , 即:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Strassen 推导得出，如果令

$$\begin{aligned}m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\m_2 &= (a_{21} + a_{22})b_{11} \\m_3 &= a_{11}(b_{12} - b_{22}) \\m_4 &= a_{22}(b_{21} - b_{11}) \\m_5 &= (a_{11} + a_{12})b_{22} \\m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\m_7 &= (a_{12} - a_{22})(b_{21} + b_{22})\end{aligned}$$

则乘积  $C$  为：

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

在习题中将证明这一结果是正确的。

为计算两个  $2 \times 2$  矩阵的乘积，Strassen 的方法需要 7 次乘法和 18 次加/减法，而直接计算法则需要 8 次乘法和 4 次加减法。我们以增加 14 次加法或减法的代价节省了一次乘法。这一结果不是那么引人注目，事实上，Strassen 方法的价值也的确没有体现在  $2 \times 2$  矩阵的乘法上。因为 Strassen 的公式中没有应用乘法交换律，所以这些公式适用于可以分别划分为四个子矩阵的大型矩阵。首先来划分矩阵  $A$  和  $B$ ，如图 2-4 所示。假定  $n$  是 2 的幂，以矩阵  $A_{11}$  为例，它表示的是  $A$  的如下子矩阵：

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,n/2} \\ a_{21} & a_{22} & \cdots & a_{2,n/2} \\ \vdots & & & \\ a_{n/2,1} & \cdots & & a_{n/2,n/2} \end{bmatrix}$$

利用 Strassen 的方法，首先计算：

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

其中，我们的运算现在是矩阵加法和乘法。以同种方式计算  $M_2$  至  $M_7$ 。接下来计算：

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

以及  $C_{12}$ 、 $C_{21}$  和  $C_{22}$ 。最后，通过合并四个子矩阵  $C_{ij}$  得到  $A$  和  $B$  的乘积  $C$ 。下例说明了这些步骤。

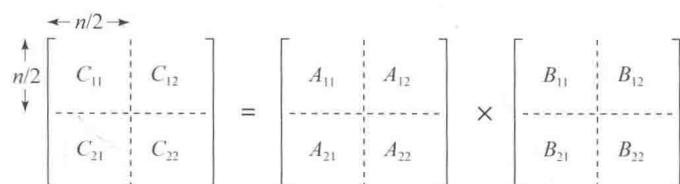


图 2-4 Strassen 算法中的子矩阵划分

例 2.5 设

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

图 2-5 演示了 Strassen 方法中的划分。计算过程如下：

$$\begin{aligned}
 M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\
 &= \left( \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) \times \left( \begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right) \\
 &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix}
 \end{aligned}$$

图 2-5  $n=4$  及矩阵具有所示数值时, Strassen 算法的矩阵划分

当这些矩阵足够小时, 可以采用直接相乘。本例在  $n=2$  时直接相乘。即:

$$\begin{aligned}
 M_1 &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} \\
 &= \begin{bmatrix} 3 \times 17 + 5 \times 7 & 3 \times 10 + 5 \times 9 \\ 11 \times 17 + 13 \times 7 & 11 \times 10 + 13 \times 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}
 \end{aligned}$$

在此之后, 以同样形式计算  $M_2$  至  $M_7$ , 然后计算  $C_{11}$ 、 $C_{12}$ 、 $C_{21}$  和  $C_{22}$ 。然后合并它们, 得出  $C$ 。

下面给出当  $n$  为 2 的幂时, Strassen 方法的一种算法。

### 算法 2.8 Strassen

问题: 计算两个  $n \times n$  矩阵的乘积, 其中  $n$  为 2 的幂。

输入: 一个整数  $n$ , 它是 2 的幂; 两个  $n \times n$  矩阵  $A$  和  $B$ 。

输出:  $A$  和  $B$  的乘积  $C$ 。

```

void strassen (int n
               n × n_matrix A,
               n × n_matrix B,
               n × n_matrix& C)
{
    if (n <= threshold)
        利用标准算法计算 C=A×B;
    else {
        将 A 分为 4 个子矩阵 A11, A12, A21, A22;
        将 B 分为 4 个子矩阵 B11, B12, B21, B22;
        使用 Strassen 方法计算 C=A×B;
        //示例递归调用:
        //strassen(n/2, A11+A22, B11+B22, M1);
    }
}

```

$threshold$  的值是指我们感到使用标准算法要比递归调用过程 strassen 更为高效的转折点。2.7 节将讨论用于确定这一阈值的方法。

### 算法 2.8 的分析 乘法次数的所有情况时间复杂度 (Strassen)

基本运算: 一次初等乘法。

输入规模:  $n$ , 矩阵的行列数。

为简单起见, 我们分析一直将矩阵划分到两个  $1 \times 1$  矩阵的情景, 这时, 只需要将两个矩阵中的数字相乘即可。实际使用的阈值并不会影响阶数。当  $n=1$  时, 只进行一次乘法。当有两个  $n \times n$  ( $n>1$ ) 矩阵时, 该算法恰好被调用 7 次, 每次传送  $(n/2) \times (n/2)$  个矩阵, 在顶级调用中没有乘法。我们已经建立了以下递推关系:

$$\boxed{\begin{aligned} T(n) &= 7T\left(\frac{n}{2}\right) && (n>1, n \text{ 是 } 2 \text{ 的幂}) \\ T(1) &= 1 \end{aligned}}$$

这一递归方程在附录 B 的例 B.2 中求解。答案为：

$$T(n) = n^{\lg 7} \approx n^{2.81} \in \Theta(n^{2.81})$$

### 算法 2.8 的分析 加/减法次数的所有情况复杂度分析 (Strassen)

基本运算：一次初等加法或减法。

输入规模： $n$ ，矩阵中的行列数。

再次假定一直划分到两个  $1 \times 1$  矩阵。当  $n=1$  时，不进行加/减法。当有两个  $n \times n$  矩阵 ( $n>1$ ) 时，该算法恰好被调用 7 次，每次传送  $(n/2) \times (n/2)$  矩阵，对  $(n/2) \times (n/2)$  矩阵执行 18 次矩阵加/减法。在将两个  $(n/2) \times (n/2)$  矩阵相加或相减时，对矩阵中的项目执行了  $(n/2)^2$  次加法或减法。我们已经得到以下递归方程：

$$\boxed{\begin{aligned} T(n) &= 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 && (n>1, n \text{ 是 } 2 \text{ 的幂}) \\ T(1) &= 0 \end{aligned}}$$

这一递归方程在附录 B 的例 B.20 中求解。答案为：

$$T(n) = 6n^{\lg 7} - 6n^2 \approx 6n^{2.81} - 6n^2 \in \Theta(n^{2.81})$$

当  $n$  不是 2 的幂时，必须修改前一算法。一种简单的修改是向原矩阵增加足够数目的行和列，使维数变为 2 的幂。或者，在递归调用中，当行列数为奇数时，仅另外增加一行和一列 0。Strassen (1969 年) 建议采用下面这种更为复杂的修改。将矩阵嵌入拥有  $2^k m$  个行与列的更大矩阵中，其中  $k=\lfloor \lg n - 4 \rfloor$ ， $m=\lfloor n/2^k \rfloor + 1$ 。在到达阈值 (threshold)  $m$  之前，使用 Strassen 方法，之后使用标准算法。可以证明，算术运算 (乘、加、减) 的总数小于  $4.7n^{2.81}$ 。

表 2-3 比较了当  $n$  为 2 的幂时，标准算法与 Strassen 算法的时间复杂度。如果暂时忽略递归调用中涉及的开销，Strassen 算法在乘法方面总是更高效一些，对于较大的  $n$  值，Strassen 的算法在加/减法方面也更高效。2.7 节将会讨论一种分析方法，其中考虑了递归调用花费的时间。

表 2-3 两种实现  $n \times n$  矩阵相乘的算法比较

	标准算法	Strassen 算法
乘法	$n^3$	$n^{2.81}$
加法/减法	$n^3 - n^2$	$6n^{2.81} - 6n^2$

Shmuel Winograd 开发了 Strassen 算法的一种变体，只需要 15 次加/减法，它出现在 Brassard 和 Bratley 的文献中 (1988 年)。对于这一算法，加/减法的时间复杂度为：

$$T(n) \approx 5n^{2.81} - 5n^2$$

Coppersmith 和 Winograd 在 1987 年开发了一种矩阵乘法算法，它的乘法次数时间复杂度属于  $O(n^{2.38})$ 。但是，对应的常数很大，对应的 Strassen 算法的效率通常更高一些。

有可能证明：矩阵乘法所需算法的时间复杂度至少为二次的。矩阵乘法能否在二次时间内完成，仍然是一个悬而未决的问题；还没有人能为矩阵乘法设计一种二次时间算法，但也还没人证明不可能设计这样一种算法。

最后一点，诸如矩阵求逆、矩阵行列式的计算等其他矩阵运算都与矩阵乘法直接相关。因此，可以很轻松地为这些运算设计一些算法，使其与 Strassen 的矩阵乘法算法一样高效。

## 2.6 大整数的算术运算

假定需要对一些整数进行算术运算，而这些整数的大小超出了计算机硬件所能表示的整数范围。如果需要保持结果中的所有有效数字，将其切换为浮点表示也没有什么价值。在这种情况下，唯一的选择就是使用软件来表示和处理这些整数。我们可以在分而治之方法的帮助下完成这一任务。我们主要讨论以基数 10 表示的整数。但是，可以很轻松地修改由此设计得出的方法，以处理其他基数。

### 2.6.1 大整数的表示：加法和其他线性时间运算

表示大整数的一种直接方法就是使用一个整数数组，每个数组位置存储一个数位。例如，整数 543 127 可以在数组  $S$  中表示如下：

$$\begin{array}{ccccccc} 5 & 4 & 3 & 1 & 2 & 7 \\ S[6] & S[5] & S[4] & S[3] & S[2] & S[1] \end{array}$$

要表示正整数和负整数，只需要为符号保留一个高阶数组位置即可。可以在该位置放入一个 0，表示正整数，放入 1 表示负整数。我们将采用这种表示法，并使用定义的数组类型 `large_integer` 表示一个很大的数组，足以表示有关应用中的整数。

不难编写一个线性时间的加减法算法，其中  $n$  为大整数中的数位个数。基本运算由一个十进制数位的运算组成。习题中会要求你编写和分析这些算法。此外，可以轻松编写完成以下运算的线性时间算法：

$$u \times 10^m \quad u \text{ divide } 10^m \quad u \text{ rem } 10^m$$

其中， $u$  表示一个较大的整数， $m$  是非负整数，`divide` 返回整数除法的商，`rem` 返回余数。这些任务也在习题中完成。

### 2.6.2 大整数的乘法

有一种简单的二次时间算法可以实现大整数的乘法，那就是模拟我们在文法学校学到的标准做法。我们将设计一种优于二次时间的算法。这种算法的基础是利用分而治之方法，将  $n$  位整数分为两个大约为  $n/2$  个数位的整数。下面是这种划分的两个例子：

$$\begin{aligned} 567\,832 &= \underbrace{567}_{6 \text{位}} \times 10^3 + \underbrace{832}_{3 \text{位}} \\ 9\,423\,723 &= \underbrace{9423}_{7 \text{位}} \times 10^3 + \underbrace{723}_{3 \text{位}} \end{aligned}$$

一般情况下，如果  $n$  是整数  $u$  的位数，则将该整数分为两个整数，其中一个有  $\lceil n/2 \rceil$  位，另一个有  $\lfloor n/2 \rfloor$  位，如下所示：

$$\underbrace{u}_{n \text{位}} = \underbrace{x}_{\lceil n/2 \rceil \text{位}} \times 10^m + \underbrace{y}_{\lfloor n/2 \rfloor \text{位}}$$

利用这一表示法，10 的指数  $m$  可给出如下：

$$m = \left\lceil \frac{n}{2} \right\rceil$$

如果有两个  $n$  位整数，

$$u = x \times 10^m + y$$

$$v = w \times 10^m + z$$

它们的乘积如下：

$$\begin{aligned} uv &= (x \times 10^m + y)(w \times 10^m + z) \\ &= xw \times 10^{2m} + (xz + wy) \times 10^m + yz \end{aligned}$$

对大约一半位数的整数执行四次乘法，并执行线性时间运算，即可实现  $u$  和  $v$  相乘。下例说明这一方法。

例 2.6 考虑下式：

$$\begin{aligned} 567\,832 \times 9\,423\,723 &= (567 \times 10^3 + 832)(9423 \times 10^3 + 723) \\ &= 567 \times 9423 \times 10^6 + (567 \times 723 + 9423 \times 832) \times 10^3 + 832 \times 723 \end{aligned}$$

采用递归方式，可以再将这些小整数分为更小的整数，以实现这些小整数的乘法。一直持续这一划分过程，直到达到一个阈值后，可以采用标准方式完成乘法。

尽管在演示此方法时使用了位数大体相同的整数，但在位数不同时仍然适用。只需使用  $m=\lfloor n/2 \rfloor$  划分两个整数，其中  $n$  是较大整数的位数。算法如下。一直进行划分，直到其中一个整数为 0，或者达到较大整数的某一阈值，这时，使用计算机的硬件来完成相乘（也就是采用通常的方法）。

### 算法 2.9 大整数乘法

问题：求两个大整数  $u$  和  $v$  的乘积。

输入：大整数  $u$  和  $v$ 。

输出：prod， $u$  和  $v$  的乘积。

```
large_integer prod(large_integer u, large_integer v)
{
    large_integer x, y, w, z;
    int n, m;

    n=maximum(u 的位数, v 的位数)
    if (u == 0 || v == 0)
        return 0;
    else if (n <= threshold)
        return 以通常形式获得的 uxv;
    else {
        m = ⌊n/2⌋;
        x = u divide 10m; y = u rem 10m;
        w = y divide 10m; z = v rem 10m;
        return prod(x,w)×102m+(prod(x,z)+prod(y,z))×10m+prod(y,z);
    }
}
```

注意， $n$  是算法的一个隐含输入，因为它是两个整数中较大整数的位数。别忘了，divide、rem 和  $\times$  表示需要编写的线性时间函数。

### 算法 2.9 的分析 最差情况时间复杂度（大整数乘法）

现在分析两个  $n$  位整数相乘需要花费多长时间。

**基本运算：**在加、减或进行  $\text{divide } 10^m$ 、 $\text{rem } 10^m$  或  $\times 10^m$  时，对一个大整数的一个十进制数位执行的运算。后三者的每次调用都会导致该基本运算被执行  $m$  次。

**输入规模：** $n$ ，两个整数中每个整数的位数。

最差情况是两个整数的所有数位都不等于 0，因为在这种情况下，只有在超过阈值时才会终止递归。我们将分析这种情况。

假定  $n$  是 2 的幂。 $x$ 、 $y$ 、 $w$  和  $z$  都恰好有  $n/2$  位，这就是说，在对 prod 的 4 次递归调用中，每个调用的输入规模都是  $n/2$ 。因为  $m=n/2$ ，所以加、减、 $\text{divide } 10^m$ 、 $\text{rem } 10^m$  和  $\times 10^m$  的线性时间运算都具有关于  $n$  的线性时间复杂度。这些线性运算的最大输入规模并非完全相同，所以要确定准确的时间复杂度并不是那么简单。将所有线性时间运算放到一项  $cn$  中就会容易得多，其中  $c$  是一个正常数。于是有递归方程：

$$\boxed{\begin{aligned} W(n) &= 4W\left(\frac{n}{2}\right) + cn && (n > s, n \text{ 是 } 2 \text{ 幂}) \\ W(s) &= 0 \end{aligned}}$$

当达到数值  $s$  时，我们不再划分实例，其实际取值小于或等于阈值，且为 2 的幂，因为在这种情况下，所有输入都是 2 的幂。

当  $n$  并不局限于 2 的幂时，有可能确定一个与以上递推关系类似，但包含上取整和下取整函数的递推关系。利用类似于附录 B 中例 B.25 中的归纳论述，可以证明  $W(n)$  是最终非递减的。因此，由附录 B 中的定理 B.6 可推出：

$$W(n) \in \Theta(n^{\lg 4}) = \Theta(n^2)$$

上面用于对大整数相乘的算法仍然是二次的。问题在于，该算法对位数为原整数一半的整数执行了四次乘法。如果可以减少这些乘法的次数，就能获得一个优于二次函数的算法。我们采用以下方式实现。回想一下，函数 prod 必须计算

$$xw, \quad xz, \quad yw, \quad yz \quad (2.4)$$

为此，我们递归调用函数 prod 四次，以计算

$$xw, \quad xz, \quad yw, \quad yz$$

如果设

$$r = (x+y)(w+z) = xw + (xz+yw) + yz$$

则

$$xz + yw = r - xw - yz$$

这就是说，通过确定以下三个值，就可以获得式 2.4 中的三个值：

$$r = (x+y)(w+z), \quad xw, \quad yz$$

为获得这三个值，只需要进行三次乘法，但要增加一些线性时间加法和减法。下面的算法实现了这一方法。

### 算法 2.10 大整数乘法 2

问题：求两个大整数  $u$  和  $v$  的乘积。

输入：大整数  $u$  和  $v$ 。

输出：prod2， $u$  和  $v$  的乘积。

```
large_integer prod2(large_integer u, large_integer v)
{
    large_integer x, y, w, z, r, p, q;
    int n, m;

    n = maximum (u 中的位数, v 中的位数);
    if (u == 0 || v == 0)
        return 0;
    else if (n <= threshold)
        return 以通常方式获得的 uxv;
    else {
        m = [n/2];
        x = u divide 10m; y = u rem 10m;
        w = v divide 10m; z = v rem 10m;
        r = prod2(x + y, w + z);
        p = prod2(x, w);
        q = prod2(y, z);
        return p×102m+(r-p-q)×10m+q;
    }
}
```

### ◆算法 2.10 的分析 最差情况时间复杂度（大整数乘法 2）

现在分析两个  $n$  位整数的相乘需要花费多长时间。

基本运算：在加、减或进行  $\text{divide } 10^m$ 、 $\text{rem } 10^m$  或  $\times 10^m$  时，对一个大整数的一个十进制数位执行的运算。后三者的每次调用都会导致该基本运算被执行  $m$  次。

输入规模:  $n$ , 两个整数中每个整数的位数。

当两个整数的所有数位都不等于 0 时发生最差情况, 因为在这种情况下, 只有在超过阈值时才会终止递归。我们分析这种情况。

若  $n$  是 2 的幂, 则  $x$ 、 $y$ 、 $w$  和  $z$  的位数均为  $n/2$ 。因此, 如表 2-4 所示,

$$\frac{n}{2} \leq x+y \text{ 的位数} \leq \frac{n}{2} + 1$$

$$\frac{n}{2} \leq w+z \text{ 的位数} \leq \frac{n}{2} + 1$$

表 2-4 算法 2.10 中  $x+y$  的位数举例

$n$	$x$	$y$	$x+y$	$x+y$ 中的位数
4	10	10	20	$2 = \frac{n}{2}$
4	99	99	198	$3 = \frac{n}{2} + 1$
8	1000	1000	2000	$4 = \frac{n}{2}$
8	9999	9999	19998	$5 = \frac{n}{2} + 1$

这意味着对于给定函数调用可以有以下输入规模:

输入规模	
prod2( $x+y, w+z$ )	$\frac{n}{2} \leq \text{输入规模} \leq \frac{n}{2} + 1$
prod2( $x, w$ )	$\frac{n}{2}$
prod2( $y, z$ )	$\frac{n}{2}$

因为  $m=n/2$ , 所以线性时间运算加、减、divide 10<sup>m</sup>、rem 10<sup>m</sup> 和  $\times 10^m$  相对于  $n$  都具有线性时间复杂度。因此,  $W(n)$  满足:

$$\boxed{3W\left(\frac{n}{2}\right) + cn \leq W(n) \leq 3W\left(\frac{n}{2} + 1\right) + cn \quad (n > s, n \text{ 为 } 2 \text{ 的幂})}$$

$W(s)=0$

其中,  $s$  小于或等于阈值, 且是 2 的幂, 因为在这种情况下, 所有输入都是 2 的幂。当  $n$  并不局限于 2 的幂时, 有可能确定一个与以上递推关系类似, 但包含上取整和下取整函数的递推关系。利用类似于附录 B 中例 B.25 中的归纳论述, 可以证明  $W(n)$  是最终非递减的。因此, 根据此递推式中的左不等式及定理 B.6, 可推出:

$$W(n) \in \Omega(n^{\log_2 3})$$

接下来证明:

$$W(n) \in O(n^{\log_2 3})$$

为此, 令

$$W'(n) = W(n+2)$$

利用递推关系中的右不等式, 有:

$$\begin{aligned}
 W'(n) &= W(n+2) \\
 &\leq 3W\left(\frac{n+2}{2}+1\right) + c[n+2] \\
 &\leq 3W\left(\frac{n}{2}+2\right) + cn + 2c \\
 &\leq 3W\left(\frac{n}{2}\right) + cn + 2c
 \end{aligned}$$

因为  $W(n)$  是非递减的，所以  $W'(n)$  也是如此。因此，根据附录 B 中的定理 B.6，有

$$W'(n) \in O(n^{\log_2 3})$$

所以，

$$W(n) = W'(n-2) \in O(n^{\log_2 3})$$

合并两个结果，得：

$$W(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.58})$$

Borodin 和 Munro (1975 年) 利用快速傅里叶变换开发了一种用于大整数相乘的  $\Theta(n(\lg n)^2)$  算法。综述文章 (Brassard, Monet and Zuffelatto, 1986 年) 介绍了非常大的整数乘法。

还有可能为其他大整数运算 (比如除法、求平方根) 编写一些算法，使其时间复杂度与乘法算法同阶。

## 2.7 确定阈值

如 2.1 节中讨论过的，递归需要相当大的计算机时间开销。例如，仅对八个键进行排序，如果使用  $\Theta(n \lg n)$  算法，而不是  $\Theta(n^2)$  算法，那上述开销是否真的划算呢？或者，对于这样一个小  $n$  值，交换排序 (算法 1.3) 是否比递归合并排序更快一点呢？我们设计一种方法，用来确定当  $n$  取何值时，调用一种替代算法的速度至少与继续细分实例一样快。这些值取决于分而治之算法、替代算法和实现这些算法的计算机。理想情况下，我们希望求出  $n$  的最优阈值。它就是一个输入规模实例，对于任何一个小于它的实例，调用另一方法的速度至少与继续细分该实例一样快。但是，后面将会看到，最优阈值并非总是存在的。即使我们的分析没有给出最优阈值，也可以利用分析结果选择一个阈值。然后修改分而治之算法，一旦  $n$  达到该阈值，就不再细分该实例，而是调用替代算法。在算法 2.8、算法 2.9 和算法 2.10 中已经使用了阈值。

要计算阈值，需要考虑实现算法的计算机。这一技术利用合并排序和交换排序加以说明。我们在这一分析中使用合并排序的最差情况时间复杂度。因此，我们正在尝试优化的实际是最差情况特性。在分析合并排序时，最差情况由以下递推式给出：

$$W(n) = W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1$$

假定我们正在实现合并排序 2 (算法 2.4)。假定在所用计算机上，合并排序 2 需要  $32n \mu\text{s}$  的时间来划分和重新合并规模为  $n$  的实例，其中  $\mu\text{s}$  表示微秒。划分和重新合并实例的时间包括：计算 mid 值的时间、为两次递归调用进行栈操作的时间、合并两个子数组的时间。因为划分与重新合并时间中包括几个不同组成部分，所以总时间不可能就是  $n$  乘以一个常数。但是，如此假定可以让事情变得尽可能简单。因为  $W(n)$  递推关系式中的  $n-1$  项是重新合并时间，所以它包含在时间  $32n \mu\text{s}$  中。因此，对于此计算机，合并排序 2 有以下递归方程：

$$W(n) = W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + 32n \mu\text{s}$$

因为在输入规模为 1 时仅进行一次终止条件校验，所以假定  $W(1)$  基本为 0。为简单起见，我们在开始时仅讨论  $n$  为 2 的幂的情景。在这种情况下，有以下递归方程：

$$\boxed{\begin{aligned} W(n) &= 2W(n/2) + 32n \mu\text{s} \quad (n > 1, n \text{ 为 } 2 \text{ 的幂}) \\ W(1) &= 0 \mu\text{s} \end{aligned}}$$

附录 B 中的方法可用于求解这一递归方程。答案为：

$$W(n) = 32n \lg n \mu\text{s}$$

假定在同一计算机上，利用交换排序对规模为  $n$  的实例进行排序，所需时间恰好为：

$$\frac{n(n-1)}{2} \mu\text{s}$$

有时，学生们会错误地认为，通过求解以下不等式，可以找出“合并排序 2”应当调用“交换排序”的最优点：

$$\frac{n(n-1)}{2} \mu\text{s} < 32n \lg n \mu\text{s}$$

答案为：

$$n < 591$$

学生们有时会认为，最优做法是在  $n < 591$  时调用“交换排序”，而在  $n \geq 591$  时调用“合并排序 2”。因为这一分析的基础是“ $n$  为 2 的幂”，所以它只是近似分析。但更重要的是，它是不正确的，因为它只是告诉我们，如果使用合并排序 2，并一直划分到  $n=1$ ，那当  $n < 591$  时，交换排序更好一些。我们希望使用合并排序 2，并一直划分，直到调用交换排序的效果好于进一步细分实例。这不同于一直划分到  $n=1$ ，因此，开始调用交换排序的  $n$  值应当小于 591。“这个值应当小于 591”，这一点从抽象层面有些难以理解。下面这个例子计算了当  $n$  取何值时，调用交换排序的效率要高于继续细分实例，这应当可以让事情变得清晰起来。从现在开始，我们的讨论不再局限于“ $n$  为 2 的幂”这种情况。

**例 2.7** 计算在调用算法 1.3（交换排序）时，算法 2.5（合并排序 2）的最优阈值。假定我们修改合并排序 2，在  $n$  不大于某一阈值  $t$  时，调用交换排序。假定采用刚刚讨论的假想计算机，对于这个版本的合并排序 2，有

$$W(n) = \begin{cases} \frac{n(n-1)}{2} \mu\text{s} & (n \leq 1) \\ W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + 32n \mu\text{s} & (n > t) \end{cases} \quad (2.5)$$

我们希望计算  $t$  的最优值。这个值就是使式 2.5 中的上下表达式相等的值，因为此时，调用交换排序的效率与继续细分实例的效率相同。因此，要确定  $t$  的最优值，必须求解：

$$W\left(\left\lfloor \frac{t}{2} \right\rfloor\right) + W\left(\left\lceil \frac{t}{2} \right\rceil\right) + 32t = \frac{t(t-1)}{2} \quad (2.6)$$

因为  $\lfloor t/2 \rfloor$  和  $\lceil t/2 \rceil$  都小于或等于  $t$ ，所以当实例取这些输入规模之一时，其执行时间由式 2.5 中的上表达式给出。因此，

$$\begin{aligned} W\left(\left\lfloor \frac{t}{2} \right\rfloor\right) &= \frac{\lfloor t/2 \rfloor (\lfloor t/2 \rfloor - 1)}{2} \\ W\left(\left\lceil \frac{t}{2} \right\rceil\right) &= \frac{\lceil t/2 \rceil (\lceil t/2 \rceil - 1)}{2} \end{aligned}$$

将这两个等式代入式 2.6，得：

$$\frac{\lfloor t/2 \rfloor (\lfloor t/2 \rfloor - 1)}{2} + \frac{\lceil t/2 \rceil (\lceil t/2 \rceil - 1)}{2} + 32t = \frac{t(t-1)}{2} \quad (2.7)$$

一般情况下，在带有下取整函数和上取整函数的公式中，为  $t$  插入奇数值和偶数值会得到不同的答案。这就是并非一定存在最优阈值的原因。这种情况将在下文研究。但在本例中，如果为  $t$  插入一个偶数值（设定  $\lfloor t/2 \rfloor$  和  $\lceil t/2 \rceil$  都等于  $t/2$ ），并求解式 2.7，得到  $t=128$ 。

如果为  $t$  插入一个奇数值（设定  $\lfloor t/2 \rfloor$  等于  $(t-1)/2$ ， $\lceil t/2 \rceil$  等于  $(t+1)/2$ ），并求解式 2.7，得到  $t=128.008$ 。因此，得到最优阈值 128。

接下来，给出一个不存在最优阈值的例子。

**例 2.8** 假设对于在一台特定计算机上运行的给定分而治之算法，我们计算得出：

$$T(n) = 3T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 16n \text{ } \mu\text{s}$$

其中  $16n \mu\text{s}$  是划分和重新组合一个规模为  $n$  的实例时所需要的时间。假定在同一计算机上，一个特定的迭代算法需要  $n^2 \mu\text{s}$  处理规模为  $n$  的实例。为计算应当调用该迭代算法的数值  $t$ ，需要求解：

$$3T\left(\left\lceil \frac{t}{2} \right\rceil\right) + 16t = t^2$$

因为  $\lceil t/2 \rceil \leq t$ ，所以当输入为此规模时，调用迭代算法，也就是说：

$$T\left(\left\lceil \frac{t}{2} \right\rceil\right) = \left\lceil \frac{t}{2} \right\rceil^2$$

因此，需要求解：

$$3\left\lceil \frac{t}{2} \right\rceil^2 + 16t = t^2$$

如果为  $t$  代入一个偶数值（通过设定  $\lceil t/2 \rceil=t/2$ ）并求解，得：

$$t = 64$$

如果为  $t$  代入一个奇数值（通过设定  $\lceil t/2 \rceil=(t+1)/2$ ）并求解，得：

$$t = 70.04$$

因为两个  $t$  值不相等，所以不存在最优阈值。这就是说，如果一个实例的规模是介于 64 到 70 之间的偶整数，那再将实例划分一次的效率较高，而当大小为介于 64 到 70 之间的奇数时，调用迭代算法的效率更高。当规模小于 64 时，总以调用迭代算法的效率为高。当规模大于 70 时，总以继续划分实例的效率为高。表 2-5 印证了上述内容。

表 2-5 各种实例规模，验证了在例 2.8 中， $n$  为偶数时的阈值为 64， $n$  为奇数时的阈值为 70

$n$	$n^2$	$3\left\lceil \frac{n}{2} \right\rceil^2 + 16n$
62	3844	3875
63	3969	4080
64	4096	4096
65	4225	4307
68	4624	4556
69	4761	4779
70	4900	4795
71	5041	5024

## 2.8 不应使用分而治之方法的情况

如果可能的话，应避免在以下两种情况下使用分而治之方法。

- (1) 一个规模为  $n$  的实例被划分为两个或多个实例，而每个实例的规模仍然几乎为  $n$ 。
- (2) 一个规模为  $n$  的实例被划分为差不多  $n$  个规模为  $n/c$  的实例，其中  $c$  为常量。

第一种划分会导致一个指数时间的算法，而第二种划分则会导致一种  $n^{\Theta(\lg n)}$  算法。对于很大的  $n$  值，这两种方法都是不可接受的。从直观上就能看出为什么这些划分会导致较差的性能。例如，第一种情况类似于拿破仑将一支拥有 30 000 名士兵的敌方军队划分为两个拥有 29 999 名士兵的军队（如果可能的话）。他不但没有分解敌人，反而几乎使其数目加倍。如果拿破仑这样做，那他遭遇滑铁卢的时间就要提前许多了。

你现在应当验证，算法 1.6（斐波那契序列的第  $n$  项，递归）是一种分而治之方法，它将计算第  $n$  项的实例划分为分别计算第  $n-1$  项和第  $n-2$  项的两个实例。尽管在此算法中  $n$  不是输入规模，但其情景与前面刚刚讨论的有关输入规模的情景相同。也就是说，算法 1.6 计算的项数是  $n$  的指数函数，而算法 1.7（斐波那契序列的第  $n$  项，迭代）计算的项数是  $n$  的线性函数。

但另一方面，在某些时候，一个问题就需要指数特性，在这种情况下，就没理由不采用简单的分而治之解决方案。考虑汉诺塔问题（在习题 17 中给出）。简单来说，这个问题就是要将  $n$  个盘子从一根柱子移到另一根柱子上，但关于它们的移动方式有一定的限制。在习题中将会证明，根据该问题的标准分而治之算法获得的移动序列，是  $n$  的指数，还会证明，在该问题的给定约束条件下，它是最有效的移动序列。因此，该问题需要很大的移动次数，与  $n$  成指数关系。

## 2.9 习题

### 2.1 节

1. 利用递归二分查找法（算法 2.1）在以下整数列表（数组）中查找整数 120。给出操作步骤。

12 34 37 45 57 82 99 120 134

2. 尽管有些不现实，但假设我们要使用递归二分查找法（算法 2.1）查找一个包含 7 亿项的列表。这一算法最多需要执行多少次比较，才能找出给定项目，或者得出它不在列表中的结论？
3. 假设我们执行的查找总是成功的。也就是说，在算法 2.1 中，总是可以在列表  $S$  中找到项目  $x$ 。请改进算法 2.1，删除其中所有不必要的操作。
4. 证明：当  $n$  不一定是 2 的幂时，二分查找（算法 2.1）的最差情况时间复杂度由下式给出：

$$W(n) = \lfloor \lg n \rfloor + 1$$

提示：首先证明  $W(n)$  的递推式由下式给出：

$$W(n) = 1 + W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \quad (n > 1)$$

$$W(1) = 1$$

为此，分别考虑  $n$  的偶数值和奇数值。然后利用归纳法求解该递推式。

5. 假设在算法 2.1（第 4 行）中，分割函数被改为  $mid = low$ 。试解释这一新的查找策略。分析这一策略的特性，并用阶的符号给出分析结果。
6. 编写一个查找  $n$  项有序列表的算法，它将列表划分为三个子列表，每个子列表差不多有  $n/3$  个项目。此算法找出可能包含给定项目的子列表，并将它分为三个几乎同样大小的更小子列表。此算法一直重复这一过程，直到找出项目，或者得出该项目不在列表的结论。分析你的算法，并用阶的符号给出分析结果。
7. 使用分而治之方法编写一个算法，找出一个  $n$  项列表中的最大项目。分析你的算法，并用阶的符号给出分析

结果。

## 2.2 节

8. 使用合并排序（算法 2.2 和算法 2.4）对以下列表排序。给出操作步骤。

123 34 189 56 150 12 9 240

9. 给出第 8 题的递归调用树。

10. 为下面的问题编写一个递归算法，其最差时间复杂度不差于  $\Theta(n \ln n)$ 。给定一个由  $n$  个不同正整数组成的列表，将该列表分为两个大小各为  $n/2$  的子列表，使两个子列表中的整数和之差最大。可以假定  $n$  是 2 的幂。

11. 为合并排序（算法 2.2 和算法 2.4）编写一个非递归算法。

12. 证明：合并排序（算法 2.2 和算法 2.4）的最差情况时间复杂度的递推式为：

$$W(n) = W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1$$

其中， $n$  并不限于 2 的幂。

13. 编写一个算法，对一个包含  $n$  个项目的列表进行排序，它将该列表划分为三个各有大约  $n/3$  项的子列表，对每个子列表进行递归排序，并合并三个有序子列表。分析你的算法，并用阶的符号给出分析结果。

## 2.3 节

14. 给定以下递推关系：

$$\boxed{T(n) = 7T\left(\frac{n}{5}\right) + 10n \quad (n > 1)} \\ T(1) = 1$$

求  $T(625)$ 。

15. 考虑下面给出的算法 solve。此算法通过求出与任意输入  $I$  相对应的输出（答案） $O$  来解决问题  $P$ 。

```
void solve (input I, output& O)
{
    if (size(I)==1)
        直接求答案 O;
    else {
        将 I 分为 5 个输入 I1、I2、I3、I4、I5,
        其中, size(Ij) = size(I)/3 (j = 1, ..., 5);
        for (j=1; j<=5; j++)
            solve(Ij, Oj);
        合并 O1、O2、O3、O4、O5, 得到输入为 I 时 P 的 O;
    }
}
```

假设划分与合并的基本运算为  $g(n)$ ，对于规模为 1 的实例没有基本运算。

(a) 写出递推关系  $T(n)$ ，表示当输入规模为  $n$  时，为求解  $P$  而执行的基本运算数。

(b) 若  $g(n) \in \Theta(n)$ ，则此递推关系的解是怎样的？（不需要证明。）

(c) 假定  $g(n)=n^2$ ，对  $n=27$  时的递推关系进行严格求解。

(d) 求出当  $n$  为 3 的幂时的一般解。

16. 假设，在一种分而治之算法中，总是将一个问题的规模为 10 的实例分解为 10 个规模为  $n/3$  的子实例，而且分解与合并步骤消耗的时间属于  $\Theta(n^2)$ 。为运行时间  $T(n)$  编写一个递推关系并求解，以得出  $T(n)$ 。

17. 为汉诺塔问题编写一个分而治之算法。汉诺塔由三个柱子和  $n$  个不同大小的圆盘组成。这些圆盘按大小的递减顺序堆叠在一个柱子上。目的是将这些圆盘移到一个新柱子上，其间以第三个柱子为临时柱子。问题的解决应当根据以下规则：(1) 在移动盘子时，必须将其放在三个柱子之一上。(2) 一次只能移动一个盘子，而且这个盘子必须是其中一个柱子上最顶端的盘子；(3) 较大的盘子永远不能放到较小盘子的上方。

(a) 为你的算法证明  $S(n)=2^n-1$ 。（这里的  $S(n)$  表示当输入为  $n$  个盘子时的移动步骤数。）

(b) 证明：任意其他算法执行的移动次数至少与(a)部分中给出的一样多。

18. 当一个分而治之算法将一个问题的规模为  $n$  的实例划分为规模分别为  $n/c$  的子实例时，递推关系通常由下式给出：

$$\boxed{\begin{aligned} T(n) &= aT\left(\frac{n}{c}\right) + g(n) \quad (n > 1) \\ T(1) &= d \end{aligned}}$$

其中， $g(n)$  是划分与合并过程的代价， $d$  是一个常数。令  $n = c^k$ 。

(a) 证明：

$$T(c^k) = d + a \sum_{j=1}^k a^{k-j} \times g(c^j)$$

(b) 若  $g(n) \in \Theta(n)$ ，求解该递推关系。

## 2.4 节

19. 使用快速排序（算法 2.6）对下面的列表排序。给出逐步操作。

123 34 189 56 150 12 9 240

20. 给出第 19 题的递归调用树。

21. 证明：若

$$W(n) \leq \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + n - 1$$

则

$$W(n) \leq \frac{n(n-1)}{2} \quad (1 \leq p \leq n)$$

在讨论算法 2.6（快速排序）的最差情况时间复杂度分析时用到了这一结果。

22. 验证以下恒等式：

$$\sum_{p=1}^n [A(p-1) + A(n-p)] = 2 \sum_{p=1}^n A(p-1)$$

在讨论算法 2.6（快速排序）的平均情况时间复杂度分析时用到了这一结果。

23. 为快速排序（算法 2.6）编写一个非递归算法。分析你的算法，并使用阶的符号给出分析结果。

24. 假定快速排序以列表中的第一项作为枢纽项：

- (a) 给定一个代表最差情况的  $n$  项列表（例如，一个包含 10 个整数的数组）；  
 (b) 给定一个代表最佳情况的  $n$  项列表（例如，一个包含 10 个整数的数组）。

## 2.5 节

25. 证明：在对算法 1.4（矩阵乘法）稍做修改后，可以将其执行的加法数目减少至  $n^3 - n^2$ 。

26. 例 2.4 中给出了两个  $2 \times 2$  矩阵的 Strassen 乘积。验证这一乘积的正确性。

27. 在使用标准算法计算两个  $64 \times 64$  矩阵的乘积时，需要执行多少次乘法？

28. 在使用 Strassen 方法计算两个  $64 \times 64$  矩阵的乘积时，需要执行多少次乘法？

29. Shmuel Winograde 设计了一种经过改进的 Strassen 算法，它执行 15 次加/减法，而不是 18 次。试写出该算法递推式。求解该递推式，并使用 2.5 节末尾给出的时间复杂度验证你的答案。

## 2.6 节

30. 使用算法 2.10（大整数乘法 2）计算 1253 和 23 103 的乘积。

31. 要计算第 30 题中两个整数的乘积，需要执行多少次乘法？

32. 编写执行以下运算的算法：

$$u \times 10^m; u \text{ divide } 10^m; u \text{ rem } 10^m$$

其中， $u$  表示一个大整数， $m$  是一个非负整数，`divide` 返回整数除法的商，`rem` 返回余数。分析你的算法，并证明这些运算可以在线性时间内完成。

33. 修改算法 2.9（大整数乘法），使其将每个  $n$  位整数分为：

- (a) 三个分别有  $n/3$  位的较小整数（可以假设  $n = 3^k$ ）；
- (b) 四个分别有  $n/4$  位的较小整数（可以假设  $n = 4^k$ ）。

分析你的算法，并以阶的符号给出它们的时间复杂度。

## 2.7 节

34. 在你的计算机上实现交换排序和快速排序算法，对一个  $n$  元素项目列表进行排序。试求出当  $n$  最小为何值时，应用快速排序算法时的开销才会物有所值。

35. 在你的计算机上实现标准算法和 Strassen 算法，完成两个  $n \times n$  ( $n = 2^k$ ) 矩阵的相乘。试求出当  $n$  最小为何值时，应用 Strassen 算法时的开销才会物有所值。

36. 假定在一台特定计算机上，要分解和重新合并算法 2.8 (Strassen) 中一个规模为  $n$  的实例，需要  $12n^2 \mu\text{s}$ 。

注意，这一时间中包含了进行所有加法和减法的时间。如果使用标准算法完成两个  $n \times n$  矩阵的相乘需要  $n^3 \mu\text{s}$ ，试确定一个阈值，在超过引线阈值后，不应再继续细分实例而应当开始调用标准算法。是否有唯一的最优阈值？

## 2.8 节

37. 利用分而治之方法编写一个计算  $n!$  的递归算法。定义输出规模（见第 1 章的习题 36），并回答以下问题。

你的函数是否具有指数时间复杂度？这是否违反了 2.8 节所给情景 1 的表述？

38. 假定，在分而治之算法中，我们总是将一个问题的规模为  $n$  的实例划分为  $n$  个规模为  $n/3$  的子实例，而且划分和合并步骤需要线性时间。试写出运行时间  $T(n)$  的递推式，并求解  $T(n)$  的这一递归方程。以阶的符号给出你的解。

## 补充习题

39. 实现斐波那契序列的两个算法（算法 1.6 和算法 1.7）。测试每个算法，以验证它是正确的。试计算，该递推算法可以接受作为参数，而且仍然能在 60 秒内给出答案的最大数字。看看迭代算法给出这一答案需要多长时间。

40. 编写一个高效算法，在一个  $n \times m$  表（二维数组）中查找一个值。这个表按行、列排序，即

```
Table[i][j] ≤ Table[i][j+1]
Table[i][j] ≤ Table[i+1][j]
```

41. 假设在一次淘汰锦标赛中有  $n=2^k$  支队伍，第一轮有  $n/2$  场比赛， $n/2=2^{k-1}$  支获胜队将进入第二轮，以此类推。

- (a) 试为该锦标赛的总轮数推导递归方程。
- (b) 当有 64 支队伍时，该锦标赛中会有多少轮比赛？
- (c) 求解(a)部分的递归方程。

42. tromino 是一组三个排列为 L 形的单位方块。考虑以下铺设瓷砖问题：输入是由单位方块组成的  $m \times m$  阵列，其中  $m$  是 2 的正数幂值，阵列中有一个禁入方块。输出结果是该阵列满足以下条件的一种排列方案：

- 除输入方块之外的所有单位方块都由一个 tromino 覆盖；
- 没有 tromino 覆盖输入方块；
- 没有两个 tromino 重叠；
- 没有 tromino 超出板块之外。

试编写解决这一问题的分而治之算法。

43. 考虑以下问题：

- (a) 假定有九枚外观相同的硬币，其编号为 1 至 9，其中只有一枚硬币的重量大于其他硬币。进一步假设，你有一个天平，而且仅有两次称重机会。试设计一种方法，在这些约束条件下找出那枚较重的赝品硬币。
- (b) 现在假定有一个整数  $n$ （表示  $n$  枚硬币），而且其中只有一枚硬币的重量大于其他硬币。进一步假定  $n$  是 3 的幂，并有  $\log_3 n$  次称重机会，用来找出那枚偏重的硬币。编写一个解决这一问题的算法。确定你所写算法的时间复杂度。
44. 编写一种递归  $\Theta(n \lg n)$  算法，其参数是三个整数  $x$ 、 $n$  和  $p$ ，该算法计算  $x^n$  除以  $p$  时的余数。为简单起见，可以假设  $n$  是 2 的幂，即对于某一正整数  $k$ ，有  $n = 2^k$ 。
45. 有一个给定列表，其中包含  $n$  个实数值，请使用分而治之方法编写一个递推算法，求出该列表任意连续子列表中的最大和值。分析你的算法，并用阶的符号给出分析结果。

# 第3章

## 动态规划



回想一下，为了确定第  $n$  个斐波那契项，分而治之算法（算法 1.6）要计算的项数是  $n$  的指数函数。原因在于分而治之方法在求解一个问题的实例时，是将它划分为较小的实例，然后再盲目地求解这些较小的实例。如第 2 章中的讨论，这是一种自顶向下的方法。它在诸如合并排序这样的问题中是有效的，这类问题中的较小实例是没有关联的，这是因为每个较小实例分别由需要独立排序的键数组构成。但是，在诸如第  $n$  个斐波那契项之类的问题中，这些较小实例是相关的。例如，如 1.2 节所示，为计算第 5 个斐波那契项，需要计算第 4 个和第 3 个斐波那契项。但是，第 4 个和第 3 个斐波那契项的计算是有关联的，因为都需要第二个斐波那契项。因为分而治之算法是独立进行这两次计算的，所以它将不止一次地计算第二个斐波那契项。在较小实例相互关联的问题中，分而治之算法经常会重复求解共同实例，从而得到一种效率非常低下的算法。

本章要讨论的方法——动态规划（dynamic programming）采取了相反的方法。动态规划与分而治之方法的相似之处是都将一个问题的一个实例划分为较小的实例。但是，动态规划首先解决小实例，存储其结果，然后在需要这样一个结果时进行查询，而不是重新计算它。“动态规划”一词来自控制理论，在这里“规划”意味着使用一个在其中存有答案的数组（表）。第 1 章曾经提到，我们用于计算第  $n$  个斐波那契项的高效算法（算法 1.7）就是动态规划的一个例子。回想一下，为了计算第  $n$  个斐波那契项，这一算法首先在一个索引范围为 0 至  $n$  的数组  $f$  中依次构建出前  $n+1$  项。在动态规划算法中，我们在一个数组（或数组序列）中自下而上地构造出一个答案。因此，动态规划是一种自下而上（bottom-up）方法。有时，比如在算法 1.7 的情景中，在使用数组（或数组序列）设计了该算法之后，就能修改该算法，从而不再需要原来分配的大部分空间。

动态规划算法的开发步骤如下。

- (1) 确立一种递归性质，能够给出一个问题实例的解。
- (2) 首先求解较小的实例，以自下而上方法求解问题的原实例。

为说明这些步骤，3.1 节给出了动态规划的另一个简单示例。其余小节将给出动态规划的更高级应用。

### 3.1 二项式系数

二项式系数（binomial coefficient，在附录 A 的 A.7 节讨论）给出如下：

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad (0 \leq k \leq n)$$

对于不太小的  $n$  和  $k$  值，无法直接根据这一定义来计算二项式系数，因为即使对于中等大小的  $n$  值， $n!$  也是非常大的。在习题中将会证明：

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & (0 < k < n) \\ 1 & (k = 0 \text{ 或 } k = n) \end{cases} \quad (3.1)$$

利用这一递推特性，就不再需要计算  $n!$  或  $k!$ 。这会让人想到下面的分而治之算法。

#### 算法 3.1 使用分而治之的二项式系数计算

问题：计算二项式系数。

输入：非负整数  $n$  和  $k$ ， $k \leq n$ 。

输出: bin, 二项式系数  $\binom{n}{k}$ 。

```
int bin (int n, int k)
{
    if (k == 0 || n == k)
        return 1;
    else
        return bin(n-1, k-1)+bin(n-1, k);
}
```

与算法 1.6 (斐波那契序列的第  $n$  项, 递归) 类似, 这个算法的效率也非常低。在习题中将会证明, 为了计算  $\binom{n}{k}$ , 这个算法需要计算

$$2\binom{n}{k}-1$$

个项目。问题在于, 每次递归调用都要求解相同的实例。例如,  $\text{bin}(n-1, k-1)$  和  $\text{bin}(n-1, k)$  都需要  $\text{bin}(n-2, k-1)$  的结果, 而在每个递归调用中, 这一实例是分别求解的。2.8 节曾经提到, 当一个实例被划分为两个几乎与原实例一样大的小实例时, 分而治之方法的效率总是很低下的。

下面使用动态规划开发了一种更高效的算法。式 3.1 中已经推导了递推性质。我们将使用这一性质在数组  $B$  中构建答案, 其中  $B[i][j]$  中将包含  $\binom{i}{j}$ 。为这一问题构建动态规划算法的步骤如下。

(1) 确立一种递归性质。这一任务已经在式 3.1 中完成。用  $B$  表示为:

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & (0 < j < i) \\ 1 & (j = 0 \text{ 或 } j = i) \end{cases}$$

(2) 以自下而上方式求解该问题的一个实例: 从第一行开始, 在  $B$  中依次计算各行。

步骤(2)在图 3-1 中演示。(你可能会发现, 图中的排列就是帕斯卡三角。) 每个后续行都是利用步骤(1)确立的递推性质, 根据其前面的行计算得出。最终计算的  $B[n][k]$  值是  $\binom{n}{k}$ 。例 3.1 演示了这些步骤。注意, 这个例子中仅计算了前两列。原因是, 这个例子中的  $k=2$ , 一般情况下, 只需要计算每行中直到第  $k$  列的值。例 3.1 计算了  $B[0][0]$ , 因为二项式系数对  $n=k=0$  进行了定义。因此, 即使在计算其他二项式系数时不需要这个值, 算法也会执行这一步骤。

	0	1	2	3	4	$j$	$k$
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
$i$							
$n$							

$B[i-1][j-1]B[i-1][j]$

$\downarrow$

$\rightarrow B[i][j]$

图 3-1 用于计算二项式系数的数组  $B$

例 3.1 计算  $B[4][2]=\binom{4}{2}$ 。

计算第 0 行：{执行这一步骤只是为了准确地模拟算法。}

{在后续计算中并不需要  $B[0][0]$ 。}

$$B[0][0]=1$$

计算第 1 行：

$$B[1][0]=1$$

$$B[1][1]=1$$

计算第 2 行：

$$B[2][0]=1$$

$$B[2][1]=B[1][0]+B[1][1]=1+1=2$$

$$B[2][2]=1$$

计算第 3 行：

$$B[3][0]=1$$

$$B[3][1]=B[2][0]+B[2][1]=1+2=3$$

$$B[3][2]=B[2][1]+B[2][2]=2+1=3$$

计算第 4 行：

$$B[4][0]=1$$

$$B[4][1]=B[3][0]+B[3][1]=1+3=4$$

$$B[4][2]=B[3][1]+B[3][2]=3+3=6$$

例 3.1 依次计算逐渐增大的二次项系数值。在每次迭代中，该迭代所需要的值都已经计算得出并存储起来。这一过程是动态规划方法的基础。下面的算法实现了这种计算二项式系数的方法。

### 算法 3.2 使用动态规划的二项式系数计算

问题：计算二项式系数。

输入：非负整数  $n$  和  $k$ ，其中  $k \leq n$ 。

输出：bin2，二项式系数  $\binom{n}{k}$ 。

```
int bin2 (int n, int k)
{
    index i, j;
    int B[0..n][0..k];

    for (i = 0; i <= n; i++)
        for (j = 0; j <= minimum(i, k); j++)
            if (j == 0 || j == i)
                B[i][j]=1;
            else
                B[i][j] = B[i-1][j-1]+B[i-1][j];
    return B[n][k];
}
```

参数  $n$  和  $k$  不是本算法输入的规模，而是它的输入，输入规模是对它们进行编码所需要的符号数。1.3 节讨论关于计算第  $n$  个斐波那契项的算法时，曾经提到一种类似情景。不过，将所需工作量表示为  $n$  和  $k$  的函数，仍然可以深入了解该算法的效率。对于给定的  $n$  和  $k$ ，计算  $\text{for-}j$  循环执行的遍数。下表给出了对于每个  $i$  值执行的遍数：

$i$	0 1 2 3 ... $k$ $k+1$ ... $n$
循环遍数	1 2 3 4 ... $k+1$ $k+1$ ... $k+1$

于是，总遍数为：

$$1 + 2 + 3 + 4 + \dots + k + \underbrace{(k+1) + (k+1) + \dots + (k+1)}_{n-k+1 \text{ 次}}$$

应用附录 A 中例 A.1 的结果，求得此表达式等于：

$$\frac{k(k+1)}{2} + (n-k+1)(k+1) = \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk)$$

上面利用动态规划代替分而治之方法，设计了一种高效得多的算法。前文曾经提到，动态规划与分而治之方法的类似之处在于要找出一种递归性质，将一个实例划分为较小的实例。它们的区别在于，动态规划中是使用递归性质，从最小的实例开始，依次迭代求解各实例，而不是盲目使用递归。这样，每个较小实例只需要求解一次。当分而治之方法会给出一种低效算法时，动态规划是一种值得尝试的好方法。

表示算法 3.2 的最直接方法就是创建整个二维数组  $B$ 。但是，一旦计算了某一行，就不再需要该行之前的各行数值。因此，在编写算法时，可以仅使用一个索引范围为 0 至  $k$  的一维数组。这一修改在习题中研究。对

该算法的另一种改进是利用如下事实： $\binom{n}{k} = \binom{n}{n-k}$ 。

## 3.2 Floyd 最短路径算法

航空旅行者遇到的一个共同问题是，当两个城市之间没有直飞航班时，如何确定由一个城市飞往另一个城市的最短路途。接下来将开发一种算法，用来解决这一问题及类似问题。首先让我们非正式地复习一点图论。图 3-2 是一个加权有向图。回想一下，在一个图的图形表示中，圆圈表示顶点（vertex），圆圈之间的线表示边（edge，也称为弧）。如果每条边都有一个与其相关联的方向，则将这种图称为有向图（directed graph 或 digraph）。在绘制这种图中的一条边时，会使用箭头表示其方向。在有向图中，两个顶点之间可以有两条边，一个方向一条。例如，在图 3-2 中，从  $v_1$  到  $v_2$  有一条边，从  $v_2$  到  $v_1$  有一条边。如果这些边有与它们相关联的值，则将这些值称为权重（weight），这种图称为加权图（weighted graph）。这里假定这些权重是非负数。尽管这些值通常称为权重，但在许多应用中，它们表示距离。因此，我们讨论的是从一个顶点到另一个顶点的路径。在有向图中，路径（path）是指一系列顶点，从每个顶点到其后续顶点都有一条边。例如，在图 3-2 中，序列  $[v_1, v_4, v_3]$  是一条路径，因为从  $v_1$  到  $v_4$  有一条边，从  $v_4$  到  $v_3$  有一条边。序列  $[v_3, v_4, v_1]$  不是路径，因为不存在从  $v_4$  到  $v_1$  的边。从一个顶点到其自身的路径称为环（cycle）。图 3-2 中的路径  $[v_1, v_4, v_5, v_1]$  是一个环。如果图中包含环，那它就是有环的（cyclic）；否则就是无环的（acyclic）。如果一条路径从来不会两次穿过同一顶点，则称该路径为简单路径（simple）。图 3-2 中的路径  $[v_1, v_2, v_3]$  是简单的，但路径  $[v_1, v_4, v_5, v_1, v_2]$  不是简单的。注意，一个简单路径的子路径绝对不会是环。加权图中的路径长度（length）是指该路径上的所有权重之和；在无权图中，它就是路径中的边数。

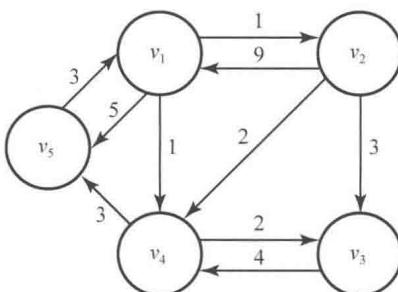


图 3-2 加权有向图

有一个问题的应用场景很多，那就是找出从每个顶点到所有其他顶点的最短路径。显然，最短路径必然是

简单路径。在图 3-2 中, 从  $v_1$  到  $v_3$  有三条简单路径, 即  $[v_1, v_2, v_3]$ 、 $[v_1, v_4, v_3]$  和  $[v_1, v_2, v_4, v_3]$ 。因为:

$$\begin{aligned}\text{length}[v_1, v_2, v_3] &= 1+3=4 \\ \text{length}[v_1, v_4, v_3] &= 1+2=3 \\ \text{length}[v_1, v_2, v_4, v_3] &= 1+2+2=5\end{aligned}$$

所以  $[v_1, v_4, v_3]$  是从  $v_1$  到  $v_3$  的最短路径。前文曾经提到, 最短路径的一个常见应用就是确定城市之间的最短路线。

最短路径问题是一种最优化问题 (optimization problem)。一个最优化问题的一个实例可能会有不止一种候选答案。每个候选答案都有一个与其相关联的值, 该实例的答案是任何拥有最优值的候选答案。根据具体问题, 此最优值可以是最小值, 也可以是最大值。在最短路径问题中, 候选答案是指从一个顶点到另一顶点的路径, 值是路径的长度, 而最优值是这些长度的最小值。

因为从一个顶点到另一顶点可能会有不止一条最短路径, 所以我们的问题是找出最短路径中的任意一条。这一问题有一种显而易见的算法, 那就是对于每个顶点, 计算从该顶点到其他每个顶点所有路径的长度, 并计算这些长度的最小值。但是, 这种算法的性能要低于指数时间。例如, 假定从每个顶点到其他每个顶点都有一条边, 那么从一个顶点到另一顶点的所有路径中, 所有符合以下条件的路径构成一个子集: 始于第一个顶点、止于另一顶点, 并穿过所有其他顶点。因为这样一条路径上的第二个顶点可以是  $n-2$  个顶点中的任意一个, 这样一条路径上的第三个顶点可以是  $n-3$  个顶点中的任意一个……这样一条路径上的倒数第二个顶点只能是一个顶点, 所以从一个顶点到另一个顶点, 并穿过所有其他顶点的路径总数是:

$$(n-2)(n-3)\cdots 1 = (n-2)!$$

它的性能低于指数函数。在许多最优化问题中都会遇到这一相同情景。也就是说, 这种显而易见、考虑所有可能性的算法具有指数时间性能, 甚至会更差。我们的目的是找出一种更高效的算法。

利用动态规划, 我们为最短路径问题设计了一种三次时间算法。首先, 设计一种算法, 仅确定最短路径的长度。然后, 对其进行修改, 使其也给出最短路径。我们用一个数组  $W$  表示一个包含  $n$  个顶点的加权图, 在这个数组中:

$$W[i][j] = \begin{cases} \text{边上的权值} & (\text{若存在从 } v_i \text{ 到 } v_j \text{ 的边}) \\ \infty & (\text{若不存在从 } v_i \text{ 到 } v_j \text{ 的边}) \\ 0 & (\text{若 } i=j) \end{cases}$$

如果存在一条从  $v_i$  到  $v_j$  的边, 就说顶点  $v_j$  与  $v_i$  相邻 (adjacent), 因此, 将这个数组称为图的邻接矩阵 (adjacency matrix) 表示。图 3-2 中的图即以这种方式表示在图 3-3 中。图 3-3 中的数组  $D$  包含了图中最短路径的长度。例如,  $D[3][5]$  是 7, 这是因为 7 是从  $v_3$  到  $v_5$  的最短路径长度。如果可以设计一种方法, 由  $W$  中的值计算  $D$  中的值, 那就找到了解决最短路径问题的算法。为完成这一任务, 创建一个包括  $n+1$  个数组的序列  $D^{(k)}$ , 其中  $0 \leq k \leq n$ , 且  $D^{(k)}[i][j]$  为从  $v_i$  到  $v_j$  的一条最短路径的长度, 该最短路径仅以集合  $\{v_1, v_2, \dots, v_k\}$  中的顶点作为中间顶点。

	1	2	3	4	5		1	2	3	4	5
1	0	1	$\infty$	1	5		0	1	3	1	4
2	9	0	3	2	$\infty$		8	0	3	2	5
3	$\infty$	$\infty$	0	4	$\infty$		10	11	0	4	7
4	$\infty$	$\infty$	2	0	3		6	7	2	0	3
5	3	$\infty$	$\infty$	$\infty$	0		3	4	6	4	0
	$W$						$D$				

图 3-3  $W$  表示图 3-2 中的图,  $D$  包含了最短路径的长度。我们求解最短路径问题的算法就是由  $W$  中的值计算  $D$  中的值

首先说明这些数组中各项的含义，然后再来解释为什么它能让我们从  $W$  计算出  $D$ 。

例 3.2 为图 3-2 中的图计算  $D^{(k)}[i][j]$  的一些典型值。

$$D^{(0)}[2][5] = \text{length}[v_2, v_5] = \infty$$

$$D^{(1)}[2][5] = \text{minimum}(\text{length}[v_2, v_5], \text{length}[v_2, v_1, v_5]) = \text{minimum}(\infty, 14) = 14$$

$$D^{(2)}[2][5] = D^{(1)}[2][5] = 14 \{ \text{对于任意图，它们都是相等的，因为始于 } v_2 \text{ 的最短路径不可能穿过 } v_2 \}$$

$$D^{(3)}[2][5] = D^{(2)}[2][5] = 14 \{ \text{对于本图，它们是相等的，因为包含 } v_3 \text{ 时不会生成从 } v_2 \text{ 到 } v_5 \text{ 的新路径} \}$$

$$\begin{aligned} D^{(4)}[2][5] &= \text{minimum}(\text{length}[v_2, v_1, v_5], \text{length}[v_2, v_4, v_5], \text{length}[v_2, v_1, v_4, v_5], \text{length}[v_2, v_3, v_4, v_5]) \\ &= \text{minimum}(14, 5, 13, 10) = 5 \end{aligned}$$

$$D^{(5)}[2][5] = D^{(4)}[2][5] = 5 \{ \text{对于任意图，它们都是相等的，因为终止于 } v_5 \text{ 的最短路径不可能穿过 } v_5 \}$$

最后计算的值  $D^{(5)}[2][5]$  是从  $v_2$  到  $v_5$ 、允许穿过任意其他顶点的最短路径长度。这意味着，它是一条最短路径的长度。

因为  $D^{(n)}[i][j]$  是从  $v_i$  到  $v_j$ 、允许穿过任意其他顶点的最短路径长度，所以它是从  $v_i$  到  $v_j$  的最短路径的长度。因为  $D^{(0)}[i][j]$  是不允许穿过任意其他顶点的最短路径的长度，所以它是从  $v_i$  到  $v_j$  的边上的权重。我们已经确定：

$$D^{(0)} = W, D^{(n)} = D$$

因此，要由  $W$  确定  $D$ ，只需要找出一种由  $D^{(0)}$  获得  $D^{(n)}$  的方式。用动态规划达成这一目的的步骤如下。

(1) 确立一种可以用来由  $D^{(k-1)}$  计算  $D^{(k)}$  的递归性质（过程）。

(2) 对于  $k=1$  到  $n$ ，重复步骤(1)中的过程，采用自下而上方式求解问题的一个实例。这就创建了序列：

$$\begin{array}{c} D^0, D^1, D^2, \dots, D^n \\ \uparrow \qquad \qquad \qquad \uparrow \\ W \qquad \qquad \qquad D \end{array} \quad (3.2)$$

在完成步骤(1)时考虑以下两种情况。

**情景 1：**仅以  $\{v_1, v_2, \dots, v_k\}$  中的顶点作为中间顶点，则从  $v_i$  到  $v_j$  至少有一条最短路径没有使用  $v_k$ 。因此：

$$D^{(k)}[i][j] = D^{(k-1)}[i][j] \quad (3.3)$$

图 3-2 中有这种情况的一个例子，即：

$$D^{(5)}[1][3] = D^{(4)}[1][3] = 3$$

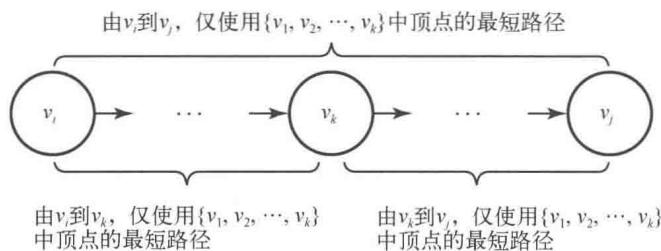
这是因为，在包含顶点  $v_5$  时，从  $v_1$  到  $v_3$  的最短路径仍然是  $[v_1, v_4, v_3]$ 。

**情景 2：**仅以  $\{v_1, v_2, \dots, v_k\}$  中的顶点作为中间顶点，则从  $v_i$  到  $v_j$  的所有最短路径都没有使用  $v_k$ 。在这种情况下，任意最短路径都如图 3-4 所示。因为  $v_k$  不是从  $v_i$  到  $v_k$  子路径上的中间顶点，所以这条子路径仅使用了  $\{v_1, v_2, \dots, v_{k-1}\}$  中的顶点为中间顶点。这意味着这条子路径的长度必然等于  $D^{(k-1)}[i][k]$ ，原因如下：第一，子路径的长度不能更短，因为  $D^{(k-1)}[i][k]$  是从  $v_i$  到  $v_k$  仅使用  $\{v_1, v_2, \dots, v_{k-1}\}$  中顶点的最短路径长度；第二，这些子路径的长度不能更长，因为如果这样的话，就可以用一条最短路径代替图 3-4 的这条路径，这与图 3-4 中的整个路径都是最短路径这一事实矛盾。与此类似，图 3-4 中从  $v_k$  到  $v_j$  的子路径长度必然等于  $D^{(k-1)}[k][j]$ 。因此，在第二种情况下：

$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j] \quad (3.4)$$

图 3-2 中第二种情况的一个例子是：

$$D^{(2)}[5][3] = 7 = 4 + 3 = D^{(1)}[5][2] + D^{(1)}[2][3]$$

图 3-4 使用  $v_k$  的最短路径

因为必然取情景 1 与情景 2 中的一种，所以  $D^{(k)}[i][j]$  是式 3.3、式 3.4 中右侧值的最小值。这意味着，可以像下面这样由  $D^{(k-1)}$  获得  $D^{(k)}$ ：

$$D^{(k)}[i][j] = \min(\underbrace{D^{(k-1)}[i][j]}_{\text{情景1}}, \underbrace{D^{k-1}[i][k] + D^{k-1}[k][j]}_{\text{情景2}})$$

我们已经完成了动态规划算法设计中的第(1)步。为完成第(2)步，我们使用步骤(1)的递归性质来创建如式 3.2 所示的数组序列。下面来完成一个例子，说明这些数组中的每个数组是如何由前一数组计算得到的。

**例 3.3** 给定图 3-2 中的图，它由图 3-3 中的邻接矩阵表示，一些示例计算如下所示（回想一下， $D^{(0)}=W$ ）：

$$\begin{aligned} D^{(1)}[2][4] &= \min(D^{(0)}[2][4], D^{(0)}[2][1]+D^{(0)}[1][4]) \\ &= \min(2, 9+1)=2 \\ D^{(1)}[5][2] &= \min(D^{(0)}[5][2], D^{(0)}[5][1]+D^{(0)}[1][2]) \\ &= \min(\infty, 3+1)=4 \\ D^{(1)}[5][4] &= \min(D^{(0)}[5][4], D^{(0)}[5][1]+D^{(0)}[1][4]) \\ &= \min(\infty, 3+1)=4 \end{aligned}$$

在计算了整个数组  $D^{(1)}$  之后，就来计算数组  $D^{(2)}$ 。一个示例计算是：

$$\begin{aligned} D^{(2)}[5][4] &= \min(D^{(1)}[5][4], D^{(1)}[5][2]+D^{(1)}[2][4]) \\ &= \min(4, 4+2)=4 \end{aligned}$$

在计算了所有  $D^{(2)}$  之后，依次继续计算，直到得出  $D^{(5)}$ 。这个最终数组是  $D$ ——最短路径的长度。它出现在图 3-3 中的右侧。

接下来介绍一种算法由 Floyd 设计（1962 年），被称为 Floyd 算法。在此算法后面，将解释为什么除了输入数组  $W$  之外，它仅使用一个数组  $D$ 。

### 算法 3.3 Floyd 最短路径算法

**问题：**计算在一个加权图中从每一顶点到其他每个顶点的最短路径。这些权值为非负数。

**输入：**一个加权有向图；图中的顶点数  $n$ 。这个图由一个二维数组  $W$  表示，其行、列的索引范围都是 1 至  $n$ ，其中  $W[i][j]$  是从第  $i$  个顶点到第  $j$  个顶点的边上的权重。

**输出：**一个二维数组  $D$ ，其行、列的索引范围都是 1 至  $n$ ，其中  $D[i][j]$  是从第  $i$  个顶点到第  $j$  个顶点的一条最短路径的长度。

```
void floyd (int n
            const number W[][],
            number D[][])
{
    index i, j, k;
    D = W;
    for (k = 1; k <= n; k++)
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
```

```
D[i][j]=minimum(D[i][j], D[i][k]+D[k][j]);
}
```

执行计算时可以仅使用一个数组  $D$ ，因为第  $k$  行、第  $k$  列的值在第  $k$  次循环迭代中不会改变。也就是说，在第  $k$  次迭代中，算法赋值如下：

$$D[i][k] = \min(D[i][k], D[i][k]+D[k][k])$$

它显然等于  $D[i][k]$ ，而且：

$$D[k][j] = \min(D[k][j], D[k][k]+D[k][j])$$

它显然等于  $D[k][j]$ 。在第  $k$  次迭代中， $D[i][j]$  是仅根据其自己的值以及第  $k$  行和第  $k$  列的值计算得出。因为这些值已经保持了来自第  $k-1$  次迭代的值，所以就是我们想要的值。如前所述，有时在开发了一种动态规划算法之后，有可能对其进行修订，提高其空间效率。

接下来分析 Floyd 算法。

### 算法 3.3 的分析 所有情况时间复杂度（Floyd 最短路径算法）

基本运算：for- $j$  循环中的指令。

输入规模： $n$ ，图中的顶点数。

在一个循环的内部循环中有一个循环，每个循环执行  $n$  遍。所以：

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3)$$

对算法 3.3 进行以下修改后可给出最短路径。

### 算法 3.4 Floyd 最短路径算法 2

问题：与算法 3.3 相同，只是还会创建最短路径。

附加输出：数组  $P$ ，其行、列的索引范围都是 1 至  $n$ ，其中：

$$P[i][j] = \begin{cases} \text{从 } v_i \text{ 到 } v_j \text{ 最短路径上一个中间顶点的最高索引 (若至少存在一个中间顶点)} \\ 0 \text{ (若不存在中间顶点)} \end{cases}$$

```
void floyd2 (int n,
             const number W[][],
             number D[][],
             index P[][])

{
    index, i, j, k;

    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            P[i][j]=0;
    D = W;
    for (k = 1; k <= n; k++)
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
                if (D[i][k]+D[k][j] < D[i][j]){
                    P[i][j] = k;
                    D[i][j] = D[i][k] + D[k][j];
                }
}
```

图 3-5 给出了在将该算法应用于图 3-2 中的图时，创建得出的数组  $P$ 。

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

图 3-5 在将算法 3.4 应用于图 3-2 中的图时生成的数组  $P$ 

下面的算法使用数组  $P$  生成了由顶点  $v_q$  到  $v_r$  的一条最短路径。

### 算法 3.5 输出最短路径

问题：输出在一个加权图中，从一个顶点到另一顶点的最短路径上的中间顶点。

输入：由算法 3.4 生成的数组  $P$ ，作为算法 3.4 输入的图中顶点的两个索引  $q$  和  $r$ 。

$$P[i][j] = \begin{cases} \text{从 } v_i \text{ 到 } v_j \text{ 最短路径上一个中间顶点的最高索引 (若至少存在一个中间顶点)} \\ 0 \text{ (若不存在中间顶点)} \end{cases}$$

输出：从  $v_q$  到  $v_r$  的最短路径上的中间顶点。

```
void path (index q, r)
{
    if (P[q][r] != 0){
        path (q, P[q][r]);
        cout << "v" << P[q][r];
        path (P[q][r], r);
    }
}
```

回想第 2 章做出的一项约定：只有在递归调用中会发生取值变化的变量才会作为递归例程的输入。因此，数组  $P$  不是  $\text{path}$  的输入。如果在实现该算法时，将  $P$  定义为全局变量，而且希望得到从  $v_q$  到  $v_r$  的一条最短路径，则对  $\text{path}$  的顶级调用将如下所示：

```
path(q, r);
```

给定图 3-5 中的  $P$  值，如果  $q$  和  $r$  的值分别为 5 和 3，则输出应当是：

```
v1 v4
```

这是从  $v_5$  到  $v_3$  的最短路径上的中间顶点。

我们在习题中推导出算法 3.5 的  $W(n) \in \Theta(n)$ 。

## 3.3 动态规划与最优化问题

回想一下，算法 3.4 不仅确定了最短的长度，还构造了最短路径。最优解的构建是为最优化问题开发动态规划算法的第三步。这就是说，开发此种算法的步骤如下。

(1) 确立一种递归性质，能够给出问题实例的最优解。

(2) 以自下而上方式计算最优解的值。

(3) 以自下而上方式构造最优解。

步骤(2)和步骤(3)是在算法中同时完成的。因为算法 3.2 不是最优化问题，所以不存在第三步。

看起来似乎任何最优化问题都可以使用动态规划方法解决，但事实并非如此。最优性原理必须适用于该问

题。该原理表述如下：

**定义** 如果一个问题的一个实例的最优解中总是包含所有子实例的最优解，就说**最优化原理适用于该问题**。

最优化原理很难表述，通过查看示例可以更好地理解。在最短路径问题中，我们证明了，如果  $v_k$  是从  $v_i$  到  $v_j$  的一条最优路径上的一个顶点，则从  $v_i$  到  $v_k$  和从  $v_k$  到  $v_j$  的子路径也必然是最优的。因此，该实例的最优解包含了所有子实例的最优解，最优化原理适用。

如果最优化原理适用于一个给定问题，就可以确定一个递归性质，用子实例的最优解给出原实例的最优解。之所以能用动态规划构造一个实例的最优解，一个重要而不太明显的原因就是，这些子实例的最优解可能是任意最优解。例如，在最短路径问题中，如果子路径是任意最短路径，那合并后的路径也将是最优的。因此，可以采用自下而上的方法，利用这种递归特性，为越来越大的实例构建最优解。在此过程形成的每个解也都是最优的。

尽管最优化定理看起来可能是显而易见的，但在实践中还是需要先证明该原理适用于相关问题，然后才能假定可以用动态规划获得最优解。下例说明，该原理并非适用于所有最优化问题。

**例 3.4** 考虑最长路径问题，也就是找出从每个顶点到所有其他顶点的最长简单路径。我们将最长路径问题限制为简单路径，是因为如果存在回路，只需要重复穿过该回路，就可以创建出任意长的路径。在图 3-6 中，从  $v_1$  到  $v_4$  的最优（最长）简单路径是  $[v_1, v_3, v_2, v_4]$ 。但是，子路径  $[v_1, v_3]$  不是从  $v_1$  到  $v_3$  的最优（最长）路径，因为：

$$\text{length}[v_1, v_3] = 1 \quad \text{length}[v_1, v_2, v_3] = 4$$

因此，最优化原理不适用。其原因在于，从  $v_1$  到  $v_3$  和从  $v_3$  到  $v_4$  的最优路径不能一同构成一条从  $v_1$  到  $v_4$  的最优路径。这样会生成一个回路，而不是一条最优路径。

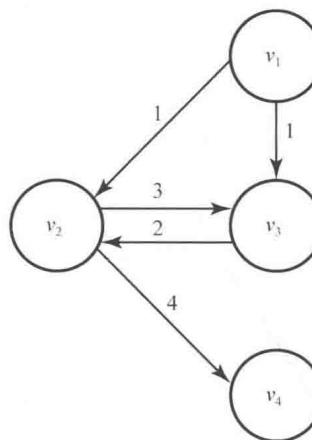


图 3-6 带有回路的加权有向图

本章其余部分涉及最优化问题。在设计这些算法时，不会明确提到前面概括的步骤。但应当清楚，我们是遵循这些步骤的。

## 3.4 矩阵链乘法

假定希望将一个  $2 \times 3$  矩阵乘以一个  $3 \times 4$  矩阵，如下所示：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

所得矩阵是一个 $2 \times 4$ 矩阵。如果使用矩阵相乘的标准方法（也就是说，根据矩阵乘法的定义获得的方法），计算乘积中的每一项时都需要三次初等乘法。例如，第一列的第一项由下式给出：

$$\underbrace{1 \times 7 + 2 \times 2 + 3 \times 6}_{\text{三次乘法}}$$

因为乘积中共有 $2 \times 4 = 8$ 项，所以初等乘法的总次数为：

$$2 \times 4 \times 3 = 24$$

一般来说，要使用标准方法将一个 $i \times j$ 矩阵乘以一个 $j \times k$ 矩阵，需要进行

$$i \times j \times k \text{ 次初等乘法}$$

考虑以下四个矩阵的乘法：

$$\begin{array}{cccccc} A & \times & B & \times & C & \times & D \\ 20 \times 2 & & 2 \times 30 & & 30 \times 12 & & 12 \times 8 \end{array}$$

每个矩阵的下面标出了其维度。矩阵乘法是一种结合性运算，也就是说，相乘顺序无关紧要。例如， $A(B(CD))$ 和 $(AB)(CD)$ 的结果相同。四个矩阵的乘法有五种不同的顺序，每种顺序会产生不同数目的初等乘法。在前面的矩阵中，每一顺序的初等乘法次数如下。

$$\begin{aligned} A(B(CD)) &= 30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 = 3680 \\ (AB)(CD) &= 20 \times 2 \times 30 + 30 \times 12 \times 8 + 20 \times 30 \times 8 = 8880 \\ A((BC)D) &= 2 \times 30 \times 12 + 2 \times 12 \times 8 + 20 \times 2 \times 8 = 1232 \\ ((AB)C)D &= 20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 = 10320 \\ (A(BC))D &= 2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 = 3120 \end{aligned}$$

对四个矩阵的相乘来说，第三种顺序是最优顺序。

我们的目标是设计一种算法，用来确定 $n$ 个矩阵相乘的最优顺序。最优顺序仅取决于矩阵的维度。因此，除了 $n$ 之外，这些维度将是算法仅有的输入。暴力算法就是像前面的做法一样，考虑所有可能顺序，并取最小值。我们将证明，这一算法至少是指数时间的。为此，令 $t_n$ 是对 $n$ 个矩阵 $A_1, A_2, \dots, A_n$ 进行相乘的不同顺序数目。由所有这些顺序组成的集合中，其中一个子集的顺序是在最后再乘以 $A_1$ 。如下所示，这一子集中的不同顺序个数是 $t_{n-1}$ ，因为它就是可以将 $A_2$ 乘到 $A_n$ 的不同顺序数：

$$A_1 \underbrace{(A_2 A_3 \cdots A_n)}_{t_{n-1} \text{ 个不同顺序}}$$

由所有顺序组成的集合还有另外一个子集，其中的顺序是在最后再乘以 $A_n$ 。显然，这一子集的不同顺序数目也是 $t_{n-1}$ 个。因此，

$$t_n \geq t_{n-1} + t_{n-1} = 2t_{n-1}$$

因为将两个矩阵相乘只有一种方式，所以 $t_2 = 1$ 。利用附录B中的方法，可以求解此递推式，证明：

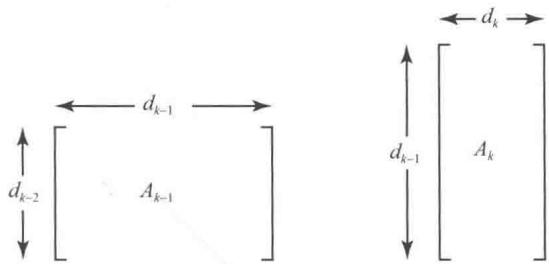
$$t_n \geq 2^{n-2}$$

不难看出，最优化原理适用于这一问题。也就是说， $n$ 个矩阵的最优相乘顺序包含了 $n$ 个矩阵任意子集的最优相乘顺序。例如，如果6个特定矩阵的最优相乘顺序为：

$$A_1 (((A_2 A_3) A_4) A_5) A_6$$

则 $(A_2 A_3) A_4$ 必然是将矩阵 $A_2$ 至 $A_4$ 相乘的最优顺序。这意味着可以使用动态规划构建一个解。

因为我们正在将第 $k-1$ 个矩阵 $A_{k-1}$ 乘以第 $k$ 个矩阵 $A_k$ ，所以 $A_{k-1}$ 中的列数必然等于 $A_k$ 中的行数。例如，在前面讨论的乘积中，第一个矩阵有三列，第二个有三行。如果设 $d_0$ 是 $A_1$ 中的行数， $d_k$ 是 $A_k$ 中的列数 $(1 \leq k \leq n)$ ，则 $A_k$ 的维数就是 $d_{k-1} \times d_k$ 。图3-7对此进行了演示。

图 3-7  $A_{k-1}$  中的列数与  $A_k$  中的行数相同

和上一节一样，将使用一个数组序列来构造解。对于  $1 \leq i \leq j \leq n$ ，设：

$$M[i][j] = \text{将 } A_i \text{ 至 } A_j \text{ 相乘时需要的最小乘法次数 (若 } i < j \text{ )}$$

$$M[i][i] = 0$$

在讨论如何使用这些数组之前，先来说明其中每一项的含义。

**例 3.5** 假设有以下六个矩阵：

$$\begin{array}{ccccccc} A_1 & \times & A_2 & \times & A_3 & \times & A_4 & \times & A_5 & \times & A_6 \\ 5 \times 2 & & 2 \times 3 & & 3 \times 4 & & 4 \times 6 & & 6 \times 7 & & 7 \times 8 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 & d_3 & d_4 & d_4 & d_5 & d_5 & d_6 \end{array}$$

为将  $A_4$ 、 $A_5$  和  $A_6$  相乘，有以下两种顺序及初等乘法次数：

$$(A_4 A_5) A_6 \text{ 乘法次数} = d_3 \times d_4 \times d_5 + d_3 \times d_5 \times d_6 = 4 \times 6 \times 7 + 4 \times 7 \times 8 = 392$$

$$A_4 (A_5 A_6) \text{ 乘法次数} = d_4 \times d_5 \times d_6 + d_3 \times d_4 \times d_6 = 6 \times 7 \times 8 + 4 \times 6 \times 8 = 528$$

因此，

$$M[4][6] = \min(392, 528) = 392$$

六个矩阵相乘的最优顺序必然具有以下因式分解之一：

1.  $A_1 (A_2 A_3 A_4 A_5 A_6)$
2.  $(A_1 A_2) (A_3 A_4 A_5 A_6)$
3.  $(A_1 A_2 A_3) (A_4 A_5 A_6)$
4.  $(A_1 A_2 A_3 A_4) (A_5 A_6)$
5.  $(A_1 A_2 A_3 A_4 A_5) A_6$

其中每个括号里的乘积都是根据括号内矩阵的最优顺序获得的。在这些因式分解中，乘法次数最少的一个必然是最优的。第  $k$  个分解的乘法次数等于获得每个因数所需的最小次数再加上这两个因式相乘所需的次数。这意味着它等于：

$$M[1][k] + M[k+1][6] + d_0 d_k d_6$$

我们已经确定：

$$M[1][6] = \min_{1 \leq k \leq 5} (M[1][k] + M[k+1][6] + d_0 d_k d_6)$$

在上面的论证中，并没有什么原因要求第一个矩阵必须为  $A_1$ ，或者要求最后一个矩阵必须为  $A_6$ 。例如，对于  $A_2$  至  $A_6$  相乘也可以获得类似结果。因此，可以推广这一结果，获得在将  $n$  个矩阵相乘时的以下递归性质。对于  $1 \leq i \leq j \leq n$ ，

$$M[i][j] = \min_{i \leq k \leq j-1} (M[i][k] + M[k+1][j] + d_{i-1} d_k d_j) \quad (\text{若 } i < j) \quad (3.5)$$

$$M[i][i] = 0$$

基于这一性质的分而治之算法为指数时间。我们使用动态规划开发一种更高效的算法，以计算步骤中的  $M[i][j]$  的值。将使用一个类似于帕斯卡三角的网格（见 3.1 节）。这种计算要比 3.1 节的计算稍复杂一些，其依据是式 3.5 具有以下属性： $M[i][j]$  是根据其同行左侧的所有项目、其同列下方的所有项目计算得出的。利用这

一属性，可以计算  $M$  中的项目如下：首先，将主对角线上的所有这些项目都设定为 0；接下来，计算对角线 1 上的所有项目（对角线 1 就是刚好高于主对角线的对角线）；接下来计算对角线 2 中的所有项目；以此类推。继续这一方式，直到计算完成对角线 5 中的唯一项目，它就是最终答案  $M[1][6]$ 。图 3-8 针对例 3.5 中的矩阵演示了这一过程。下面的例子给出了这些计算。

**例 3.6** 假定有例 3.5 中的六个矩阵。动态规划算法中的步骤如下。结果显示在图 3-8 中。

计算对角线 0：

$$M[i][i]=0 \quad (1 \leq i \leq 6)$$

计算对角线 1：

$$\begin{aligned} M[1][2] &= \min_{1 \leq k \leq 1} (M[1][k] + M[k+1][2] + d_0 d_k d_2) \\ &= M[1][1] + M[2][2] + d_0 d_1 d_2 \\ &= 0 + 0 + 5 \times 2 \times 3 = 30 \end{aligned}$$

$M[2][3]$ 、 $M[3][4]$ 、 $M[4][5]$  和  $M[5][6]$  的值都以相同方式计算，在图 3-8 中给出。

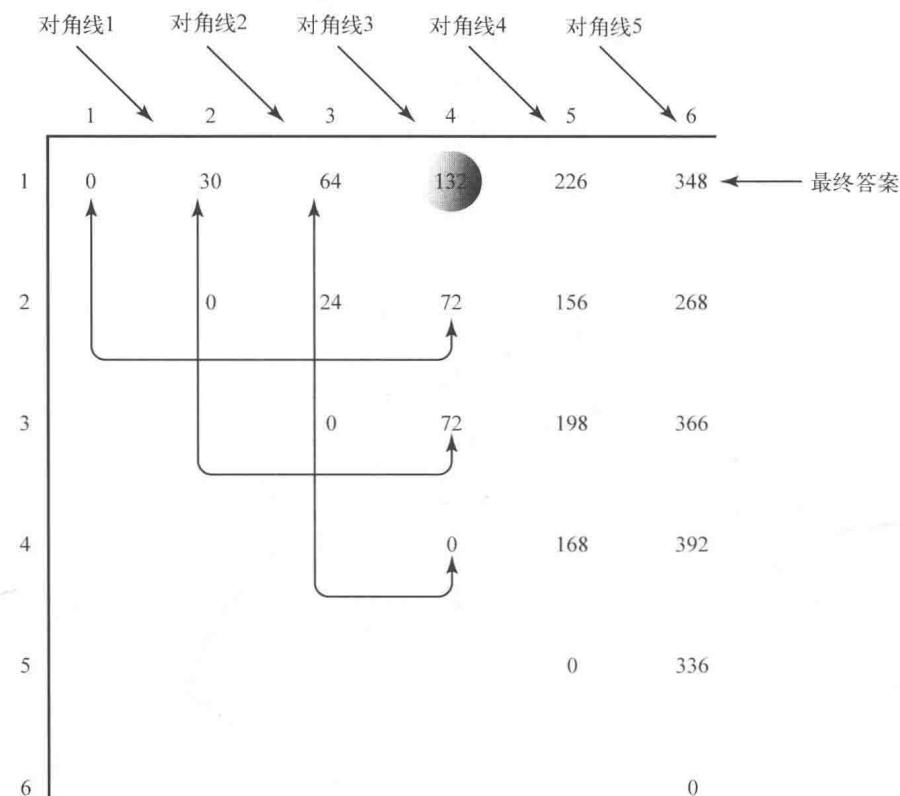


图 3-8 例 3.5 中逐步形成的数组  $M$ 。被圈起来的  $M[1][4]$  是根据标出的一对项目计算得出的

计算对角线 2：

$$\begin{aligned} M[1][3] &= \min_{1 \leq k \leq 2} (M[1][k] + M[k+1][3] + d_0 d_k d_3) \\ &= \min(M[1][1] + M[2][3] + d_0 d_1 d_3, M[1][2] + M[3][3] + d_0 d_2 d_3) \\ &= \min(0 + 24 + 5 \times 2 \times 4, 30 + 0 + 5 \times 3 \times 4) = 64 \end{aligned}$$

$M[2][4]$ 、 $M[3][5]$ 、 $M[4][5]$  和  $M[5][6]$  的值都以相同方式计算，如图 3-8 所示。

计算对角线 3：

$$\begin{aligned}
 M[1][4] &= \min_{1 \leq k \leq 3} (M[1][k] + M[k+1][4] + d_0 d_k d_4) \\
 &= \min(M[1][1] + M[2][4] + d_0 d_1 d_4, \\
 &\quad M[1][2] + M[3][4] + d_0 d_2 d_4, \\
 &\quad M[1][3] + M[4][4] + d_0 d_3 d_4) \\
 &= \min(0 + 72 + 5 \times 2 \times 6, 30 + 72 + 5 \times 3 \times 6, 64 + 0 + 5 \times 4 \times 6) = 132
 \end{aligned}$$

$M[2][5]$ 、 $M[3][5]$ 和 $M[3][6]$ 的值都以相同方式计算，如图 3-8 所示。

计算对角线 4：

对角线 4 的项目以相同方式计算，如图 3-8 所示。

计算对角线 5：

最后，以相同方式计算对角线 5 中的项目。这一项就是实例的解；它是初等乘法的最小数目，它的值由下式给出：

$$M[1][6]=348$$

下面的算法实现了这一方法。 $n$ 个矩阵的维度（即， $d_0$ 至 $d_n$ 的值）是算法仅有的输入。矩阵本身不是输入，因为矩阵的值与本问题无关。由算法生成的数组 $P$ 可用于输出最优顺序。在分析算法 3.6 之后将讨论这一点。

### 算法 3.6 最少量乘法

问题：确定将 $n$ 个矩阵相乘所需初等乘法的最少量，以及生成该最小数的相乘顺序。

输入：矩阵个数 $n$ ；整数数组 $d$ ，其索引范围为 0 至  $n$ ，其中 $d[i-1] \times d[i]$ 是第 $i$ 个矩阵的维度。

输出：minmult，实现 $n$ 个矩阵相乘所需初等乘法的最少量；一个二维数组 $P$ ，由它可以得到最优顺序。 $P$ 中行的索引范围为 1 至  $n-1$ ，列的索引范围为 1 至  $n$ 。 $P[i][j]$ 是按照矩阵相乘的最优顺序，对矩阵 $i$ 至 $j$ 进行划分的位置。

```

int minmult (int n,
             const int d[],
             index P[][])
{
    index i, j, k, diagonal;
    int M[1..n][1..n];

    for (i = 1; i <= n; i++)
        M[i][i] = 0;
    for (diagonal = 1; diagonal <= n - 1; diagonal++) // 对角线 1 刚刚高于主对角线
        for (i = 1; i <= n - diagonal; i++){
            j = i + diagonal;
            M[i][j] =
                minimum(M[i][k]+M[k+1][j]+f[i-1]*d[k]*d[j]);
            i<=k<i-1
            P[i][j]=给出该最小值的 k 值;
        }
    return M[1][n];
}

```

接下来分析算法 3.6。

### 算法 3.6 的分析 所有情况时间复杂度（最少量乘法）

基本运算：可以将对每个 $k$ 值执行的指令看作基本运算。其中包含为判断是否为最小值所做的一次比较。

输入规模： $n$ ，要相乘的矩阵个数。

在一个循环的内部循环中有一个循环。因为对于 $diagonal$  和 $i$ 的给定值，有 $j=i+diagonal$ ，所以 $k$ 循环的执行遍数为：

$$j-1-i+1=i+diagonal-1-i+1=diagonal$$

对于 $diagonal$ 的一个给定值，执行 $for-i$ 循环的遍数为 $n-diagonal$ 。因为 $diagonal$ 由 1 变至  $n-1$ ，所以基本

运算的执行总次数等于

$$\sum_{\text{diagonal}=1}^{n-1} [(n - \text{diagonal}) \times \text{diagonal}]$$

在习题中将会推导出，这一表达式等于

$$\frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

下面说明如何从数组  $P$  获得一种最优顺序。在将该算法应用于例 3.5 中的维度时，该数组的值如图 3-9 所示。例如， $P[2][5]=4$  意味着将  $A_2$  至  $A_5$  相乘的最优顺序具有以下分解：

$$(A_2 A_3 A_4) A_5$$

其中，括号内的矩阵是根据最优顺序相乘。即， $P[2][5]$ （它是 4）是应当划分这些矩阵以获得因子的位置。我们首先查看  $P[1][n]$ ，以确定顶级分解，从而生成一种最优顺序。因为  $n=5$  和  $P[1,6]=1$ ，所以最优顺序中的顶级分解为：

$A_1(A_2 A_3 A_4 A_5 A_6)$					
1	2	3	4	5	6
1	1	1	1	1	1
2		2	3	4	5
3			3	4	5
4				4	5
5					5

图 3-9 在将算法 3.6 应用于例 3.5 中的维度时生成的数组  $P$

接下来查看  $P[2][6]$ ，以确定  $A_2$  至  $A_6$  的最优相乘顺序中的分解。因为  $P[2][6]$  的值为 5，所以该分解为：

$$(A_2 A_3 A_4 A_5) A_6$$

现在知道最优顺序的分解为：

$$A_1((A_2 A_3 A_4 A_5) A_6)$$

其中，仍然必须确定  $A_2$  至  $A_5$  相乘的分解。接下来查阅  $P[2][5]$ ，继续这一方式，直到确定所有分解为止。答案是：

$$A_1(((A_2 A_3) A_4) A_5) A_6$$

下面的算法实现刚刚描述的方法。

### 算法 3.7 输出最优顺序

问题：输出  $n$  个矩阵相乘的最优顺序。

输入：正整数  $n$ ；由算法 3.6 生成的数组  $P$ 。 $P[i][j]$  是在矩阵  $i$  至  $j$  的最优相乘顺序中的划分点。

输出：这些矩阵的最优相乘顺序。

```
void order (index i, index j)
{
    if (i == j)
        cout << "A" << i;
    else {
        k = P[i][j];
        cout << "(";
        order (i, k);
        order (k + 1, j);
        cout << ")";
    }
}
```

根据对递归的一般约定， $P$  和  $n$  不是 order 的输入，而是算法的输入。如果在实现该算法时，将  $P$  和  $n$  定义为全局变量，则对 order 的顶级调用应当如下所示：

```
order (1, n);
```

当维度为例 3.5 中的维度时，则该算法输出以下结果：

$$(A_1(((A_2A_3)A_4)A_5)A_6))$$

因为该算法在每个混合项的外围都放置括号，所以整个表达式外围也有括号。在习题中为算法 3.7 确定了以下关系：

$$T(n) \in \Theta(n)$$

我们为链式矩阵乘法设计的  $\Theta(n^3)$  算法源于 Godbole (1973 年)。Yao (1982 年) 开发了一些用于加速特定动态规划方案的方法。利用这些方法，有可能为链式矩阵乘法创建  $\Theta(n^2)$  算法。Hu 和 Shing (1982 年, 1984 年) 为链接矩阵乘法描述了一种  $\Theta(nlgn)$  算法。

## 3.5 最优二叉查找树

接下来设计一种算法，用来确定二叉查找树中一组项目的最佳组织方式。首先来回顾一下二叉查找树，然后再讨论哪种组织方式是最优的。对于二叉树中的任意节点，以该节点的左子节点为根节点的子树，称为该节点的左子树 (left subtree)。树的根节点的左子树称为树的左子树。右子树 (right subtree) 采用类似定义。

**定义** 二叉查找树 (binary search tree) 是由一些项目组成的二叉树，这些项目通常称为键，来自一个有序集合，满足以下条件：

- (1) 每个节点包含一个键；
- (2) 一个给定节点的左子树中的键小于或等于该节点的键；
- (3) 一个给定节点的右子树中的键大于或等于该节点的键。

图 3-10 显示了两棵二叉查找树，分别拥有相同的键。在左边的树中，查看 “Ralph” 所在节点的右子树。这棵树包含 “Tom” “Ursula” 和 “Wally”，根据字母排序，这些名字均大于 “Ralph”。尽管在一般情况下，一个键可以在二叉查找树中出现多次，但为简单起见，假定这些键是各不相同的。

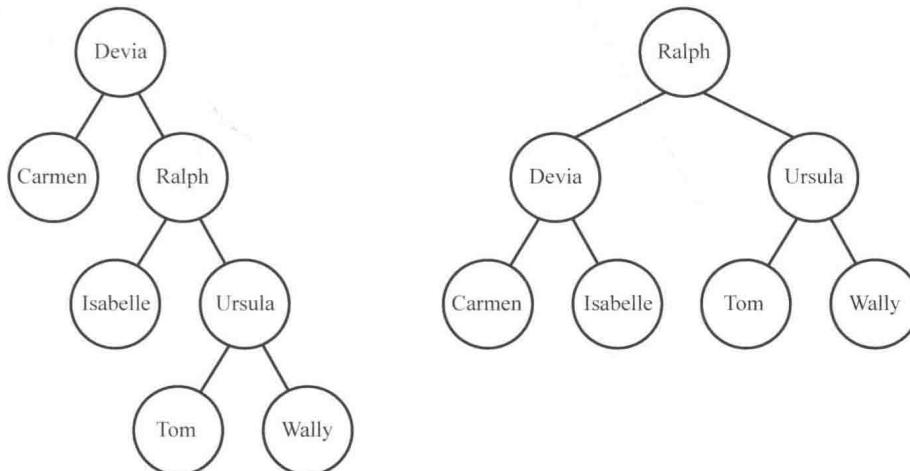


图 3-10 两个二叉查找树

树中一个节点的深度 (depth) 就是从根节点到该节点的唯一路径中的边数，也称为该节点在树中的级别 (level)。通常，我们说一个节点拥有某一深度，位于某一级别。例如，在图 3-10 左边的树中，“Ursula”所在的节点拥有深度 2。或者，也可以说该节点位于级别 2。根节点拥有深度 0，位于级别 0。一棵树的深度是树中所有节点的最大深度。图 3-10 中左边的树的深度为 3，而右边的树的深度为 2。如果一个二叉树中每个节点的两个子树的深度差从来不超过 1，就说该二叉树是平衡的 (balanced)。图 3-10 中左边的树是不平衡的，因为根节点左子树的深度为 0，右子树的深度为 2。图 3-10 中右边的树是平衡树。

一般来说，二叉查找树包含了根据键值检索的记录。我们的目的是组织二叉查找树中的键，使得查找一个键的平均时间降至最短。（关于平均的讨论，请见 A.8.2 节。）以这种方式组织的树称为最优的 (optimal)。不难看出，如果所有键作为查找键 (search key) 的概率相同，那图 3-10 右边的树就是最优的。我们关心的是这些键的概率不相同时的情景。比如，在图 3-10 中的一棵树中查找在美国随机选出的一个人名，就属于上述情景的一个例子。因为“Tom”要比“Ursula”更常见，所以我们会为“Tom”分配一个较大的概率。（关于随机性的讨论请参阅附录 A 中的 A.8.1 节。）

下面讨论已知查找键确实在树中的情景。在习题中将研究这种情况的一般化情景：查找键可能不在树中。为使平均查找时间最短，需要知道查找一个键的时间复杂度。因此，在继续讨论之前，先编写和分析一个在二叉查找树中查找一个键的算法。该算法使用以下数据类型：

```
struct nodetype
{
    keytype key;
    nodetype* left;
    nodetype* right;
};

typedef nodetype* node_pointer;
```

这一声明意味着 `node_pointer` 变量是 `nodetype` 记录的一个指针。也就是说，它的值是这样一个记录的内存地址。

### 算法 3.8 查找二叉树

**问题：**在二叉查找树中找出包含一个键的节点。假定这个键就在树中。

**输入：**指向二叉查找树的指针 `tree` 和一个键 `keyin`。

**输出：**指针 `p`，指向包含这个键的节点。

```
void search ( node_pointer tree,
              keytype keyin,
              node_pointer& p)
{
    bool found;

    p = tree;
    found = false;
    while ( ! found)
        if(p->key == keyin)
            found = true;
        else if (keyin < p->key);
            p = p->left;           //向左子节点前进。
        else
            p = p->right;         //向右子节点前进。
}
```

过程 `search` 为查找一个键而进行的比较次数称为查找时间 (search time)。我们的目的是确定一棵树，使其平均查找时间降至最短。1.2 节曾经讨论过，假定可以高效地实现这些比较。在这一假设下，前面算法中每迭代一次 `while` 循环，只完成一次比较。因此，一个给定键的查找时间为：

$$\text{depth(key)} + 1$$

其中,  $\text{depth(key)}$  是该键所在节点的深度。例如, 因为在图 3-10 左边的树中, “Ursula” 所在节点的深度为 2, 所以“Ursula”的查找时间为:

$$\text{depth(Ursula)} + 1 = 2 + 1 = 3$$

设  $\text{Key}_1, \text{Key}_2, \dots, \text{Key}_n$  依次是  $n$  个键, 并设  $p_i$  是  $\text{Key}_i$  作为查找键的概率。若  $c_i$  是在给定树中查找  $\text{Key}_i$  所需的比较次数, 则这棵树的平均查找时间为:

$$\sum_{i=1}^n c_i p_i$$

这是我们希望最小化的值。

**例 3.7** 图 3-11 给出了当  $n=3$  时的五棵树。键的实际值并不重要。唯一的要求是它们是有序的。如果:

$$p_1=0.7 \quad p_2=0.2 \quad p_3=0.1$$

图 3-11 中树的平均查找时间为:

1.  $1(0.7) + 2(0.2) + 1(0.1) = 2.6$
2.  $2(0.7) + 3(0.2) + 1(0.1) = 2.1$
3.  $2(0.7) + 1(0.2) + 2(0.1) = 1.8$
4.  $1(0.7) + 3(0.2) + 2(0.1) = 1.5$
5.  $1(0.7) + 2(0.2) + 3(0.1) = 1.4$

第五棵树是最优的。

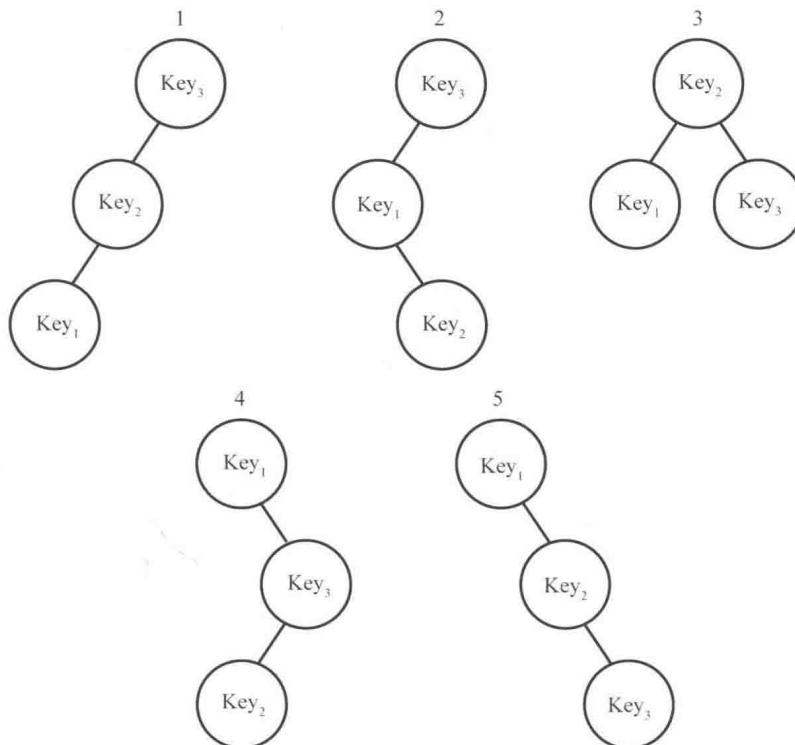


图 3-11 当有三个键时可能的二叉查找树

一般情况下, 我们无法通过考虑所有二叉查找树来找出一种最优二叉查找树, 因为这些树的数量至少是  $n$  的指数函数。可以按如下思路证明。如果我们仅考虑深度为  $n-1$  的所有二叉查找树, 会得到指数数量的树。在深度为  $n-1$  的二叉查找树中, 在  $n-1$  级中每一级中的节点 (除根结点外), 或者是其父节点的左子节点, 或者是其右子节点, 这意味着在这些级别中的每一种都有两种可能性。也就是说, 深度为  $n-1$  的不同二叉查找树的

数量为  $2^{n-1}$ 。

我们将使用动态规划开发一种更高效的算法。为此，假定键  $\text{Key}_i$  至  $\text{Key}_j$  在树中的排列使得

$$\sum_{m=i}^j c_m p_m$$

最小，其中  $c_m$  为了从树中找出  $\text{Key}_m$  所需要的比较次数。我们说这样一棵树对于这些键来说是最优的，并用  $A[i][j]$  表示最优值。因为在仅包含一个键的树中查找一个键只需要一次比较，所以  $A[i][i] = p_i$ 。

**例 3.8** 假定有例 3.7 中的三个键及相关概率。即：

$$p_1=0.7 \quad p_2=0.2 \quad p_3=0.1$$

为确定  $A[2][3]$ ，必须考虑图 3-12 中的两棵树。对于这两棵树，有以下结果：

1.  $1(p_2)+2(p_3)=1(0.2)+2(0.1)=0.4$
2.  $2(p_2)+1(p_3)=2(0.2)+1(0.1)=0.5$

第一棵树是最优的，且：

$$A[2][3]=0.4$$

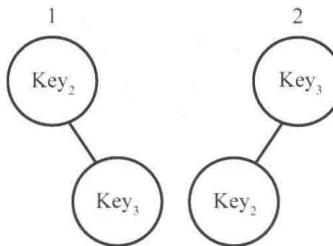


图 3-12 由  $\text{Key}_2$  和  $\text{Key}_3$  组成的二叉查找树

注意，例 3.8 中获得的最优树是例 3.7 最优树根节点的右子树。即使这个树并非与右子树完全相同，在其中进行查找的平均时间也必然相同。否则，就可以用它来代替该子树，得到一棵具有更短平均查找时间的树。一般来说，一棵最优树的任意子树对于该子树中的键都必然是最优的。因此，最优性原理适用。

接下来，设树 1 是当  $\text{Key}_1$  在根节点时的最优树，树 2 是当  $\text{Key}_2$  在根节点时的最优树……树  $n$  是当  $\text{Key}_n$  在根节点时的最优树。对于  $1 \leq k \leq n$ ，树  $k$  的子树必然是最优的，因此，这些子树中的平均查找时间如图 3-13 所示。该图还表明，对于每个  $m \neq k$ ，若在树  $k$  中查找  $\text{Key}_m$  需要  $x$  次比较，在包含该键的子树中查找该键需要  $y$  次比较，则  $x$  比  $y$  恰好大 1（也就是在根节点处进行的比较）。对于树  $k$  中的  $\text{Key}_m$ ，这一次比较增加的平均查找时间为  $1 \times p_m$ 。我们已经确定，树  $k$  的平均查找时间为：

$$\underbrace{A[1][k-1]}_{\text{左子树的平均时间}} + \underbrace{p_1 + \dots + p_{k-1}}_{\text{在根节点处进行比较的附加时间}} + \underbrace{p_k}_{\text{在根节点处进行查找的平均时间}} + \underbrace{A[k+1][n]}_{\text{右子树的平均时间}} + \underbrace{p_{k+1} + \dots + p_n}_{\text{在根节点处进行比较的附加时间}}$$

它等于：

$$A[1][k-1] + A[k+1][n] + \sum_{m=1}^n p_m$$

因为  $k$  棵树之一必然为最优的，所以最优树的平均查找时间为：

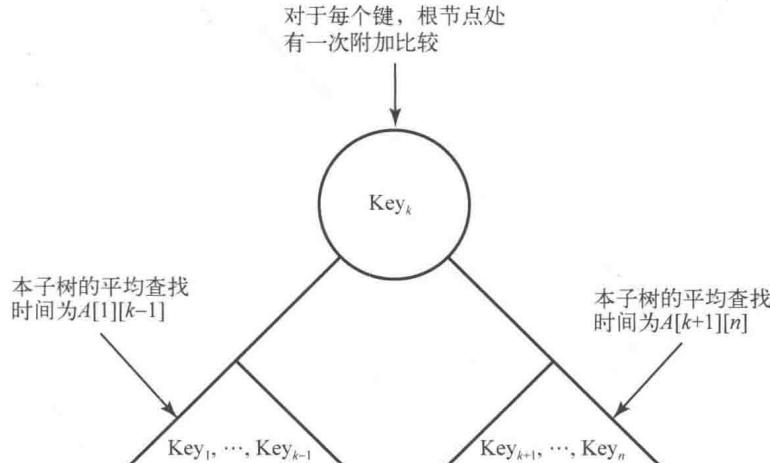
$$A[1][n] = \min_{1 \leq k \leq n} (A[1][k-1] + A[k+1][n] + \sum_{m=1}^n p_m)$$

其中  $A[1][0]$  和  $A[n+1][n]$  被定义为 0。尽管后一表达式中的可能性之和显然为 1，但我们仍然将其写为和式，这是因为现在希望推广该结果。为此，在前面的讨论中，并没有什么约束条件要求这些键必须为  $\text{Key}_1$  至  $\text{Key}_n$ 。也就是说，一般情况下，该讨论适用于  $\text{Key}_i$  至  $\text{Key}_j$ （其中  $i < j$ ）。因此，推导出下式：

$$A[i][j] = \min_{i \leq k \leq j} (A[i][k-1] + A[k+1][j]) + \sum_{m=i}^j p_m \quad (i < j)$$

$$A[i][i] = p_i$$

(3.6)

图 3-13  $Key_k$  在根节点时的最优二叉查找树

利用式 3.6, 可以编写一个确定最优二叉树的算法。因为  $A[i][j]$  是由位于第  $i$  行但在  $A[i][j]$  左侧的项目和位于第  $j$  列但在  $A[i][j]$  下方的项目计算得出的, 所以我们接下来依次计算每个对角线上的值(如算法 3.6 中的做法)。因为算法中的步骤与算法 3.6 非常类似, 所以这里不再给出演示这些步骤的例子, 而是直接给出算法, 再给出一个综合示例, 展示应用该算法的结果。由该算法生成的数组  $R$  包含了在每一步为根节点选择的键的索引。例如,  $R[1][2]$  是一棵最优树根节点处的键的索引, 这棵最优树包含了前两个键。而  $R[2][4]$  同样是一棵最优树根节点处的键的索引, 该最优树中包含了第二、三、四个键。在分析了该算法之后, 将讨论如何由  $R$  构建最优树。

### 算法 3.9 最优二叉查找树

问题: 为一组键确定一棵最优二叉查找树, 每个键都带有作为查找键的给定概率。

输入:  $n$ , 键的个数; 一个实数数组  $p$ , 其索引范围为 1 至  $n$ , 其中  $p[i]$  是查找第  $i$  个键的概率。

输出: 变量  $\text{minavg}$ , 它的值是一棵最优二叉查找树的平均查找时间; 一个二维数组  $R$ , 可以由该数组构造一棵最优树。 $R$  中行的索引范围为 1 至  $n+1$ , 列的索引范围为 0 至  $n$ 。 $R[i][j]$  是一棵最优树根节点处的键的索引, 该最优树中包含了第  $i$  至  $j$  个键。

```

void optsearchtree (int n,
                    const float p[],
                    float& minavg,
                    index R[][])
{
    index i, j, k, diagonal;
    float A[1..n + 1][0..n];
    for (i = 1; i <= n; i++){
        A[i][i - 1] = 0;
        A[i][i] = p[i];
        R[i][i] = i;
        R[i][i - 1] = 0;
    }
    A[n + 1][n] = 0;
    R[n + 1][n] = 0;
    for (diagonal = 1; diagonal <= n - 1; diagonal++)
        for (i = 1; i <= n - diagonal; i++) { //Diagonal-1 恰好位于主对角线之上。
            j = i + diagonal;
            if (A[i][j] > A[i][j - 1] + A[i + 1][j])
                R[i][j] = R[i][j - 1];
            else
                R[i][j] = R[i + 1][j];
            A[i][j] = A[i][j - 1] + A[i + 1][j] + p[j];
        }
}

```

```

A[i][j] = minimum(A[i][k-1] + A[k+1][j]) +  $\sum_{m=i}^j p_m$ 
R[i][j] = 给出最小值的 k 值;
}
minavg = A[1][n];
}

```

### 算法 3.9 的分析 所有情况时间复杂度（最优二叉查找树）

基本运算：为每个  $k$  值执行的指令。其中包含了为判断最小值进行的比较。 $\sum_{m=i}^j p_m$  不需要每次都从头计算。

在习题中将会找出一种计算这些和值的有效方法。

输入规模： $n$ ，键的个数。

这一算法的控制流程几乎与算法 3.6 完全相同。唯一的区别在于，对于 diagonal 和  $i$  的给定值，基本运算将完成  $\text{diagonal}+1$  次。根据类似于对算法 3.6 的分析，可以得出：

$$T(n) = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$$

下面的算法由数组  $R$  构建一个二叉树。回想一下， $R$  中包含了在每一步为根节点选择的键的索引。

### 算法 3.10 生成最优二叉查找树

问题：生成最优二叉查找树。

输入： $n$ ，键的个数；数组 Key，其中依次包含了  $n$  个键；由算法 3.9 生成的数组  $R$ 。 $R[i][j]$  是一棵树的根节点处的键的索引，其中包含了第  $i$  至  $j$  个键。

输出：指针 tree，指向一棵包含  $n$  个键的最优二叉查找树。

```

node_pointer tree (index i, j)
{
    index k;
    node_pointer p;

    k=R[i][j];
    if (k == 0)
        return NULL;
    else {
        p = new nodetype;
        p-> key = Key[k];
        p-> left = tree (i, k - 1);
        p-> right = tree (k + 1, j);
        return p;
    }
}

```

指令  $p=\text{new nodetype}$  获得一个新节点，并将其地址放在  $p$  中。根据我们对递归算法的约定，参数  $n$ 、Key 和  $R$  不是函数 tree 的输入。如果实现算法时将  $n$ 、Key 和  $R$  定义为全局变量，则通过调用 tree，可以获得一个指向最优二叉查找树根节点的指针 root，如下所示：

```
root = tree (1, n);
```

算法 3.9 中没有给出具体步骤，因为它与算法 3.6（最少量乘法）类似。同样，算法 3.10 中也没有给出这些步骤，因为此算法与算法 3.7（输出最优顺序）类似。因此，我们提供了一个综合示例，给出了应用算法 3.9 与算法 3.10 的结果。

例 3.9 假定有数组 Key 的以下值：

Don	Isabelle	Ralph	Wally
Key[1]	Key[2]	Key[3]	Key[4]

以及：

$$p_1 = \frac{3}{8} \quad p_2 = \frac{3}{8} \quad p_3 = \frac{1}{8} \quad p_4 = \frac{1}{8}$$

由算法 3.9 生成的数组  $A$  和  $R$  显示在图 3-14 中，由算法 3.10 创建的树显示在图 3-15 中。最少平均查找时间为  $7/4$ 。

	0	1	2	3	4		0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$	1	0	1	1	2	2
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1	2		0	2	2	2
3			0	$\frac{1}{8}$	$\frac{3}{8}$	3			0	3	3
4				0	$\frac{1}{8}$	4				0	4
5					0	5					0

$A$                      $R$

图 3-14 数组  $A$  和  $R$ ，在将算法 3.9 应用于例 3.9 中的实例时生成

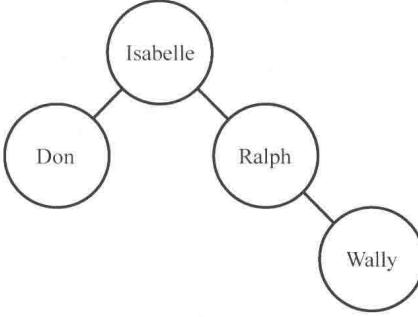


图 3-15 在将算法 3.9 和算法 3.10 应用于例 3.9 中的实例时生成的树

注意， $R[1][2]$  可以是 1 或 2。理由是，在仅包含前两个键的最优树中，上述两个数字都可能是其根节点的索引。因此，这两个索引在算法 3.9 中都可以使  $A[1][2]$  取最小值，也就是说，这两者都可以选作  $R[1][2]$ 。

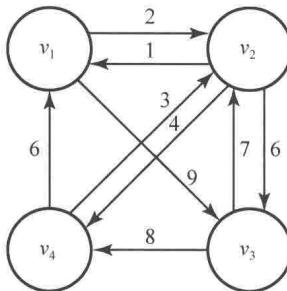
前面用于确定最优二叉树的算法来自 Gilbert 和 Moore (1959 年)。 $\Theta(n^2)$  算法可以使用 Yao 的动态规划加速方法获得 (1982 年)。

## 3.6 旅行推销员问题

假定一位推销员正在计划一次包含 20 个城市的推销旅行。每个城市都由一条道路连接到部分其他城市。为尽量缩短旅行时间，希望确定一条最短路线，从推销员的所在城市出发，每个城市到访一次，最终回到起始城市。确定最短路线的问题称为“旅行推销员”问题。

这一问题的一个实例可以用一个加权图表示，其中的每个顶点表示一个城市。和在 3.2 节中一样，我们推广这一问题，使其包含如下情景：沿某一方向的权重（距离）可不同于沿另一方向的权重。再次假定这些权重是非负数。图 3-2 和图 3-16 显示了这种加权图。有向图中的一个旅程 (tour, 也称为哈密顿回路) 是指从一个顶点到其自身的一条路径，该路径穿过所有其他顶点，且仅穿过一次。在一个加权有向图中，最优旅程是具有最短长度的这种路径。旅行推销员问题就是在加权有向图中至少存在一条旅程时，找出一条最优旅程来。因为起始顶点与最优旅程的长度无关，所以我们将  $v_1$  看作起始顶点。下面是图 3-16 中的三条旅程和对应的长度：

$\text{length}[v_1, v_2, v_3, v_4, v_1] = 22$   
 $\text{length}[v_1, v_3, v_2, v_4, v_1] = 26$   
 $\text{length}[v_1, v_3, v_4, v_2, v_1] = 21$

图 3-16 最优旅程为  $[v_1, v_3, v_4, v_2, v_1]$ 

最后一条旅程是最优的。我们直接通过考虑所有可能旅程就解决了这一实例。一般来说，从每个顶点到每个其他顶点都可以有一条边。如果考虑所有可能存在的旅程，旅程上的第二个顶点可以是  $n-1$  个顶点中的任意一个，旅程上的第三个顶点可以是  $n-2$  个顶点中的任意一个……旅程上的第  $n$  个顶点只能是一个顶点。因此，旅程的总数为：

$$(n-1)(n-2)\cdots 1 = (n-1)!$$

它要劣于指数复杂度。

动态规划能否应用于这一问题？注意，如果  $v_k$  是一条最优旅程上排在  $v_1$  之后的第一个顶点，那么该旅程中由  $v_k$  到  $v_1$  的子路径必然是从  $v_k$  到  $v_1$  且恰好穿过所有其他顶点一次的最短路径。这就是说最优化原理适用于这一问题，可以使用动态规划。为此，我们和 3.2 节中一样，用一个邻接矩阵  $W$  来表示图。图 3-17 给出了图 3-16 中图的邻接矩阵表示。设：

$V$ =所有顶点的集合

$A=V$  的一个子集

$D[v_i][A]$ =一条最短路径的长度，该路径从  $v_i$  到  $v_1$  经过  $A$  中的每个顶点，且恰好经过一次

		1	2	3	4
		1	2	3	4
1	1	0	2	9	$\infty$
	2	1	0	6	4
3	$\infty$	7	0	8	
4	6	3	$\infty$	0	

图 3-17 图 3-16 中图的邻接矩阵表示  $W$ 

例 3.10 对于图 3-17 中的图，

$$V=\{v_1, v_2, v_3, v_4\}$$

注意， $\{v_1, v_2, v_3, v_4\}$  使用大括号来表示集合，而  $[v_1, v_2, v_3, v_4]$  使用中括号表示路径。若  $A=\{v_3\}$ ，则

$$D[v_2][A]=\text{length}[v_2, v_3, v_1]=\infty$$

若  $A=\{v_3, v_4\}$ ，则

$$D[v_2]=\min(\text{length}[v_2, v_3, v_4, v_1], \text{length}[v_2, v_4, v_3, v_1])=\min(20, \infty)=20$$

因为  $V-\{v_1, v_j\}$  包含除  $v_1$  和  $v_j$  之外的所有顶点，而且最优化原理适用，所以有

$$\text{最优旅程的长度} = \min_{2 \leq j \leq n} (W[1][j] + D[v_j][V - \{v_1, v_j\}])$$

一般来说，对于  $i \neq 1$  及不在  $A$  中的  $v_i$ ，

$$\begin{aligned} D[v_i][A] &= \min_{j: v_j \in A} (W[i][j] + D[v_j][A - \{v_j\}]) \quad (\text{若 } A \neq \emptyset) \\ D[v_i][\emptyset] &= W[i][1] \end{aligned} \quad (3.7)$$

利用式 3.7，可以为旅行推销员问题生成一种动态规划算法。但首先来演示该算法将如何进行。

**例 3.11** 为图 3-17 所示的图确定一条最优旅程。首先考虑空集：

$$D[v_2][\emptyset] = 1$$

$$D[v_3][\emptyset] = \infty$$

$$D[v_4][\emptyset] = 6$$

现在考虑所有包含一个元素的集合：

$$\begin{aligned} D[v_3][\{v_2\}] &= \min_{j: v_j \in \{v_2\}} (W[3][j] + D[v_j][\{v_2\} - \{v_j\}]) \\ &= W[3][2] + D[v_2][\emptyset] = 7 + 1 = 8 \end{aligned}$$

同理：

$$D[v_4][\{v_2\}] = 3 + 1 = 4$$

$$D[v_2][\{v_3\}] = 6 + \infty = \infty$$

$$D[v_4][\{v_3\}] = \infty + \infty + \infty$$

$$D[v_2][\{v_4\}] = 4 + 6 = 10$$

$$D[v_3][\{v_4\}] = 8 + 6 = 14$$

接下来考虑所有包含两个元素的集合：

$$\begin{aligned} D[v_4][\{v_2, v_3\}] &= \min_{j: v_j \in \{v_2, v_3\}} (W[4][j] + D[v_j][\{v_2, v_3\} - \{v_j\}]) \\ &= \min(W[4][2] + D[v_2][\{v_3\}], W[4][3] + D[v_3][\{v_2\}]) \\ &= \min(3 + \infty, \infty + 8) = \infty \end{aligned}$$

同理：

$$D[v_3][\{v_2, v_4\}] = \min(7 + 10, 8 + 4) = 12$$

$$D[v_2][\{v_3, v_4\}] = \min(6 + 14, 4 + \infty) = 20$$

最后，计算最优旅程的长度：

$$\begin{aligned} D[v_1][\{v_2, v_3, v_4\}] &= \min_{j: v_j \in \{v_2, v_3, v_4\}} (W[1][j] + D[v_j][\{v_2, v_3, v_4\} - \{v_j\}]) \\ &= \min(W[1][2] + D[v_2][\{v_3, v_4\}], \\ &\quad W[1][3] + D[v_3][\{v_2, v_4\}], \\ &\quad W[1][4] + D[v_4][\{v_2, v_3\}]) \\ &= \min(2 + 20, 9 + 12, \infty + \infty) = 21 \end{aligned}$$

旅行推销员问题的动态规划算法如下。

### 算法 3.11 旅行推销员问题的动态规划算法

**问题：**确定一个加权有向图中的最优旅程。权重为非负数。

**输入：**一个加权有向图； $n$ ，图中的顶点数。该图由一个二维数组  $W$  表示，它的行、列索引范围都是 1 至  $n$ ，

其中  $W[i][j]$  是从第  $i$  个顶点到第  $j$  个顶点的边上的权重。

**输出：**变量  $\text{minlength}$ ，它的值是最优旅程的长度；一个二维数组  $P$ ，可以由其构建一个最优旅程。 $P$  中行的索引范围为 1 至  $n$ ，列的索引范围是  $V - \{v_1\}$  的所有子集。 $P[i][A]$  是从  $v_i$  到  $v_1$  的最短路径上位于  $v_i$  之后第一

个顶点的索引，该最短路径经过  $A$  中的所有顶点，且恰好经过一次。

```

void travel (int n, const number W[][], index P[][],
             number& minlength)
{
    index i, j, k; number D[1..n][V-{vi}的子集];

    for (i = 2; i <= n; i++)
        D[i][∅] = W[i][1];
    for (k = 1; k <= n - 2; k++)
        for (包含 k 个顶点的所有子集 A⊆V-{vi})
            for (满足 i≠1 且 vi 不在 A 中的 i){
                D[i][A]=minimumj:j∈A (W[i][j]+D[j][A-{vj}]);
                P[i][A]=给出最小值的 j 值;
            }
        D[1][V-{vi}]=minimum2≤j≤n (W[i][j]+D[j][V-{vi, vj}]);
        P[1][V-{vi}]=给出最小值的 j 值;
        minlength = D[1][V-{vi}];
}

```

在说明如何由数组  $P$  获得一个最优旅程之前，先来分析该算法。首先需要一个定理：

定理 3.1 对于所有  $n \geq 1$ ,

$$\sum_{k=1}^n k \binom{n}{k} = n 2^{n-1}$$

证明：下式的证明留作练习。

$$k \binom{n}{k} = n \binom{n-1}{k-1}$$

因此，

$$\begin{aligned} \sum_{k=1}^n k \binom{n}{k} &= \sum_{k=1}^n n \binom{n-1}{k-1} \\ &= n \sum_{k=0}^{n-1} \binom{n-1}{k} \\ &= n 2^{n-1} \end{aligned}$$

最后一个等式是应用附录 A 中例 A.10 的结果获得的。

算法 3.11 的分析

所有情况时间与空间复杂度（旅行推销员问题的动态规划算法）

**基本运算：**与中间循环中的时间相比，第一循环和最后循环的时间都是微不足道的，因为中间循环包含了多个嵌套级别。因此，将对于每个  $v_j$  执行的指令作为基本运算。它们包含了一个加法指令。

**输入规模：**  $n$ ，图中的顶点数。

对于每个包含  $k$  个顶点的集合  $A$ ，必须考虑  $n-1-k$  个顶点，对于其中每个顶点，基本运算都会执行  $k$  次。

因为在  $V-\{v_i\}$  的子集中，包含  $k$  个顶点的子集个数等于  $\binom{n-1}{k}$ ，所以基本运算的完成总数为：

$$T(n) = \sum_{k=1}^{n-2} (n-1-k) k \binom{n-1}{k} \quad (3.8)$$

不难证明：

$$(n-1-k) \binom{n-1}{k} = (n-1) \binom{n-2}{k}$$

将此等式代入式 3.8, 得:

$$T(n) = (n-1) \sum_{k=1}^{n-2} k \binom{n-2}{k}$$

最后, 应用定理 3.1, 得:

$$T(n) = (n-1)(n-2)2^{n-3} \in \Theta(n^2 2^n)$$

因为此算法使用的内存也很大, 所以我们还将分析其内存复杂度, 称之为  $M(n)$ 。用于存储数组  $D[v_i][A]$  和  $P[v_i][A]$  的内存显然占内存量的主要部分, 因此, 我们将确定这些数组必须有多大。因为  $V - \{v_i\}$  中包含  $n-1$  个顶点, 所以可应用附录 A 中例 A.10 的结果得出结论: 它有  $2^{n-1}$  个子集  $A$ 。数组  $D$  和  $P$  的第一个索引范围为 1 至  $n$ 。因此,

$$M(n) = 2 \times n 2^{n-1} = n 2^n \in \Theta(n 2^n)$$

这时, 你可能会对我们得到的结果感到奇怪, 因为我们的新算法仍然是  $\Theta(n^2 2^n)$ 。下面的例子说明, 即使是具有这一时间复杂度的算法, 有时也是有用的。

**例 3.12** Ralph 和 Nancy 在竞争同一销售职位。老板在周五告诉他们, 从周一开始, 谁先跑完所有 20 个城市的地域, 谁就获得这个职位。这片地域包含他们所在城市, 而且在完成任务后必须返回所在城市。

从每座城市到每座其他城市都有一条道路。Ralph 认为自己有整个周末的时间来确定路线, 所以他就直接采用暴力算法, 在自己的计算机上考虑所有  $(20-1)!$  种旅程。Nancy 想起自己在算法课上学到的动态规划算法。她认为自己应当发挥所有优势, 于是在自己的计算机上运行了该算法。假定需要 1 微秒的时间来处理 Nancy 算法中的基本运算, 而 Ralph 算法则需要 1 微秒的时间来计算每条旅程的长度, 那么, 每种算法所需的时间大体如下。

暴力算法:  $19! \mu s = 3857$  年

动态程序算法:  $(20-1)(20-2)2^{20-3} \mu s = 45$  秒

可以看出, 如果替代算法的时间复杂度为阶乘函数, 那即使是  $\Theta(n^2 2^n)$  的算法也可能是有用的。本例中动态规划算法使用的内存为:

$$20 \times 2^{20} = 20971520 \text{ 个数组位置。}$$

尽管这个数值很大, 但根据当今的标准, 仍然是切实可行的。

用  $\Theta(n^2 2^n)$  算法来寻找最优旅程之所以可行, 只是因为  $n$  值很小。例如, 如果有 60 个城市, 那这种算法也会需要很多年的时间。

现在讨论如何从数组  $P$  中获得最优旅程。我们不会给出算法, 而是直接演示它是如何做到的。为了确定图 3-16 所示图中的一个最优旅程, 数组  $P$  中需要确定的成员为:

3	4	2
$P[1, \{v_2, v_3, v_4\}]$	$P[3, \{v_2, v_4\}]$	$P[4, \{v_2\}]$

得出的一条最优旅程如下:

$$\text{第一个节点的索引} = P[1][\{v_2, v_3, v_4\}] = 3$$

$$\text{第二个节点的索引} = P[3][\{v_2, v_4\}] = 4$$

$$\text{第三个节点的索引} = P[4][\{v_2\}] = 2$$

因此, 最优旅程为:

$$[v_1, v_3, v_4, v_2, v_1]$$

还从来没有人能够为旅行推销员问题找出一种算法，使其最差情况时间复杂度优于指数复杂度，但也从来没有人证实这种算法不存在。第9章将讨论一大类密切相关的问题，它们都具有这一特性，旅行推销员问题只是其中一个。

### 3.7 序列对准

下面给出动态规划在分子遗传学上的一种应用，即同源 DNA 序列对准。首先简单复习遗传学中的一些概念。

染色体是一种很长的链状大分子，由复合脱氧核糖核酸（DNA）组成。染色体是生物遗传特征的载体。DNA 包括两链互补基因，每一链都由一个核苷酸序列组成。核苷酸由戊糖（脱氧核糖）、一个磷酸基团和一个嘌呤或嘧啶碱基组成。嘌呤——腺嘌呤（A）和鸟嘌呤（G）——的结构类似，嘧啶——胞嘧啶（C）和胸腺嘧啶（T）——也是如此。两链基因由核苷酸对之间的氢键结合在一起。每个核苷酸对被称为一个典范碱基对（bp），A、G、C 和 T 称为碱基。

图 3-18 中绘制了一段 DNA。两链基因实际上相互扭在一起，构成一种右旋双螺旋结构。但为达到我们的目的，只需将它们看作如图所示的字符串即可。人们相信，一个染色体就是一个长的 DNA 分子。



图 3-18 一段 DNA

DNA 序列是一段 DNA，位点（site）就是每个碱基对在序列中的位置。DNA 序列可以经历置换突变（一个核苷酸被另一个置换）、插入突变（在序列中插入一个碱基对）和缺失突变（序列中缺失一个碱基对）。

考虑一个特定种群（物种）每个个体中的相同 DNA 序列。在每个世代，序列中的每个位点都有可能在每个配子中产生突变，产生下一世代的一个个体。一种可能出现的结果是，在一个给定位点，一个碱基由整个种群（或种群大部分个体）的另一碱基置换。另一种可能出现的结果是，最终发生了一个物种生成事件，也就是说，种群中的成员分为两个不同物种。在这种情况下，在其中一个物种中最终发生的总置换远远不同于另一物种中发生的总置换。这意味着分别取自两个物种的个体序列可能会有很大不同。我们说这些序列已经发生了分歧。两个物种的对应序列称为同源序列。在系统树干扰中，我们关心的是对比来自不同物种的同源序列，并估计它们在进化意义上的距离。

在对比来自两个不同物种的两个个体的同源序列时，必须首先对准这些序列，因为这些序列中的一个或两个可能在产生分歧之后经历了插入以及/或者缺失突变。例如，对准人类与夜猴的第一基因内区，会得到一个 196 核苷酸序列，在任一序列中，都有 163 个位点不存在缝隙。

**例 3.13 假定有以下同源序列：**

A	A	C	A	G	T	T	A	C
T	A	A	G	G	T	C	A	

可以用许多方式对准它们。下面给出两种对准方式：

-	A	A	C	A	G	T	T	A	C
T	A	A	-	G	G	T	-	-	C

A	A	C	A	G	T	T	A	C
T	A	-	G	G	T	-	C	A

当在一种对准方式中包含一个破折号（-）时，称之为插入一个缝隙。表示带有缝隙的序列存在一次缺失，或者另一序列存在一次插入。

前面例子中的哪种对准方式更好一点？它们都有 8 个匹配的碱基对。上面的对准方式有两个失配碱基对，但代价是插入了四个缝隙。而下面的对准方式中有三个失配碱基对，但其代价是仅插入了两个缝隙。一般情况下，如果没有事先指定失配和缝隙的罚分，是不可能判定哪种对准方式更好的。例如，假定一个缝隙的罚分为 1，一个失配的罚分为 3。将一种对准方式中所有罚分之和称为该对准方式的代价。给定这些罚分分配方式，例 3.13 上面的对准方式的代价为 10，而下面的代价为 11。因此，上面的要好一些。但如果缝隙的罚分为 2，而失配的罚分为 1，不难看出，下面的对准方式的代价较低，所以也就更好一些。

在为缝隙和失配指定了罚分之后，就有可能确定最优对准方式了。但是，如果为了做出判定而要查看所有可能出现的对准方式，将是一件难以处理的任务。下面为序列对准问题开发一种高效的动态规划算法。

为了让描述更具体一些，做出以下假定：

□ 失配的罚分为 1；

□ 缝隙的罚分为 2。

首先在数组中表示两个序列，如下所示：

A	A	C	A	G	T	T	A	C	C
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]
T	A	A	G	G	T	C	A		
y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]		

设  $\text{opt}(i, j)$  是子序列  $x[i..9]$  与  $y[j..7]$  最优对准的代价，则  $\text{opt}(0, 0)$  是  $x[0..9]$  和  $y[0..7]$  最优对准的代价，这正是我们希望执行的对准。这一最优对准的起始方式必然为下面情况之一。

(1)  $x[0]$  与  $y[0]$  对准。若  $x[0]=y[0]$ ，则第一个对准位点没有罚分，而若  $x[0]\neq y[0]$ ，则罚分为 1。

(2)  $x[0]$  与一个缝隙对准，第一个对准位点处的罚分为 2。

(3)  $y[0]$  与一个缝隙对准，第一个对准位点处的罚分为 2。

假定  $x[0..9]$  和  $y[0..7]$  的最优对准  $A_{\text{opt}}$  将  $x[0]$  与  $y[0]$  对准，则这一对准方式中包含了  $x[1..9]$  与  $y[1..7]$  的一个对准  $B$ 。假定这一对准  $B$  并不是两个子序列的最优对准，那应当还有其他某个对准  $C$ ，其代价要更小一些。这样，对准  $C$  再加上  $x[0]$  与  $y[0]$  的对准，就会得到  $x[0..9]$  和  $y[0..7]$  的一个对准，其代价要小于  $A_{\text{opt}}$ 。因此，对准  $B$  必然是  $x[1..9]$  与  $y[1..7]$  的最优对准。同理，如果  $x[0..9]$  和  $y[0..7]$  的最优对准将  $x[0]$  与一个缝隙对准，那么该对准中也必然包含了  $x[1..9]$  与  $y[0..7]$  的一个最优对准；如果  $x[0..9]$  和  $y[0..7]$  的最优对准将  $y[0]$  与一个缝隙对准，那这一对准中包含了  $x[0..9]$  与  $y[1..7]$  的一个最优对准。

例 3.14 假定下面是  $x[0..9]$  和  $y[0..7]$  的一个最优对准：

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]
A	A	C	A	G	T	T	A	C	C
T	A	-	A	G	G	T	-	C	A
y[0]	y[1]		y[2]	y[3]	y[4]	y[5]		y[6]	y[7]

那下面的对准就必然是  $x[1..9]$  与  $y[1..7]$  的一个最优对准：

x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]
A	C	A	G	T	T	A	C	C
A	-	A	G	G	T	-	C	A
y[1]		y[2]	y[3]	y[4]	y[5]		y[6]	y[7]

如果设  $x[0]=y[0]$  时  $\text{penalty}=0$ ，否则为 1，则可以确立以下递推性质：

$$\text{opt}(0, 0)=\min(\text{opt}(1, 1)+\text{penalty}, \text{opt}(1, 0)+2, \text{opt}(0, 1)+2)$$

我们在演示这一递归性质时是从两个序列的第 0 个位置开始的，但很显然，如果从任意位置开始，它同样成立。因此，一般来说：

$$\text{opt}(i, j)=\min(\text{opt}(i+1, j+1)+\text{penalty}, \text{opt}(i+1, j)+2, \text{opt}(i, j+1)+2)$$

为完成递归算法的开发，需要有终止条件。设  $m$  是序列  $\mathbf{x}$  的长度， $n$  是序列  $\mathbf{y}$  的长度。如果在到达序列  $\mathbf{x}$  的终点 ( $i=m$ ) 时正处于序列  $\mathbf{y}$  的第  $j$  个位置 ( $j < n$ )，那就必须插入  $n-j$  个缝隙。因此，一个终止条件是：

$$\text{opt}(m, j) = 2(n-j)$$

同理，如果在到达序列  $\mathbf{y}$  的终点 ( $j=n$ ) 时正处于序列  $\mathbf{x}$  的第  $i$  个位置 ( $i < m$ )，那就必须插入  $m-i$  个缝隙。因此，另一个终止条件是：

$$\text{opt}(i, n) = 2(m-i)$$

现在有了下面的分而治之算法。

### 算法 3.12 使用分而治之的序列对准

问题：确定两个同源 DNA 序列的一种最优对准方式。

输入：一个长度为  $m$  的 DNA 序列  $\mathbf{x}$  和一个长度为  $n$  的 DNA 序列  $\mathbf{y}$ 。这两个序列在数组中表示。

输出：两个序列最优对准的代价。

```
void opt (int i, int j)
{
    if (i == m)
        opt = 2(n - j);
    else if (j == n)
        opt = 2(m - i);
    else {
        if (x[i] == y[j])
            penalty = 0;
        else
            penalty = 1;
        opt = min(opt(i + 1, j + 1) + penalty, opt(i + 1, j) + 2,
                  opt(i, j + 1) + 2);
    }
}
```

对此算法的顶级调用为：

```
optimal_cost = opt(0, 0);
```

注意，这个算法仅给出了一种最优对准的代价，并没有给出最优对准。可以对其进行修改，使其也给出一种最优对准。但是，这里不会这样做，因为这种算法的效率非常低下，我们实际使用的并不是这种算法。该算法拥有指数时间复杂度，其证明留作习题。

这个算法的问题在于，许多子实例都会被计算多次。例如，为了在顶级计算  $\text{opt}(0,0)$ ，需要计算  $\text{opt}(1,1)$ 、 $\text{opt}(1,0)$  和  $\text{opt}(0,1)$ 。要在第一次递归调用中计算  $\text{opt}(1,0)$ ，需要计算  $\text{opt}(2,1)$ 、 $\text{opt}(2,0)$  和  $\text{opt}(1,1)$ 。对  $\text{opt}(1,1)$  的两次求值并不需要独立完成。

为了使用动态规划解决这一问题，我们为当前实例创建一个  $(m+1) \times (n+1)$  数组，如图 3-19 所示。注意，在每个序列中都包含了一个额外字符，它是一个缝隙。这样做的目的是为上行迭代赋予一个起点。我们希望计算  $\text{opt}(i,j)$ ，并将它存储在这个数组的第  $ij$  个位置。回想一下，有以下等式：

$$\text{opt}(i, j) = \min(\text{opt}(i+1, j+1) + \text{penalty}, \text{opt}(i+1, j) + 2, \text{opt}(i, j+1) + 2) \quad (1)$$

$$\text{opt}(10, j) = 2(8-j) \quad (2)$$

$$\text{opt}(i, 8) = 2(10-i) \quad (3)$$

现在将当前实例中的  $m$  值和  $n$  值代入公式中。如果位于数组的底行，则使用式(2)计算  $\text{opt}(i, j)$ ；如果位于最右列，则使用式(3)；在其他情况下，使用式(1)。注意，在式(1)中，每个数组项的值都可以由其右侧数组项的值、其下方数组值的值和其右下方数据项的值计算得出。例如，图 3-19 中演示了  $\text{opt}(6, 5)$  是由  $\text{opt}(6, 6)$ 、 $\text{opt}(7, 5)$  和  $\text{opt}(7, 6)$  计算得出的。

<i>j</i>	0	1	2	3	4	5	6	7	8
<i>i</i>	T	A	A	G	G	T	C	A	-
0	A								
1	A								
2	C								
3	A								
4	G								
5	T								
6	T								
7	A								
8	C								
9	C								
10	-								

图 3-19 用于找出最优对准方式的数组

因此，首先计算对角线 1 中的所有值，然后计算对象线 2 中的所有值，再计算对角线 3 中的所有值，以此类推，就可以计算出图 3-19 数组中的所有值。下面举例说明前三个对角线的计算。

□ 对角线 1：

$$\text{opt}(10, 8)=2(10-10)=0$$

□ 对角线 2：

$$\text{opt}(9, 8)=2(10-9)=2$$

$$\text{opt}(10, 7)=2(8-7)=2$$

□ 对角线 3：

$$\text{opt}(8, 8)=2(10-8)=4$$

$$\begin{aligned} \text{opt}(9, 7) &= \min(\text{opt}(9+1, 7+1)+\text{penalty}, \text{opt}(9+1, 7)+2, \text{opt}(9, 7+1)+2) \\ &= \min(0+1, 3+2, 2+2)=1 \end{aligned}$$

$$\text{opt}(10, 6)=2(8-6)=4$$

图 3-20 给出了计算出所有值之后的数组。最优对准的值为  $\text{opt}(0, 0)$ ，等于 7。

<i>j</i>	0	1	2	3	4	5	6	7	8
<i>i</i>	T	A	A	G	G	T	C	A	-
0	A	7	8	10	12	13	15	16	18
1	A	6	6	8	10	11	13	14	16
2	C	6	5	6	8	9	11	12	14
3	A	7	5	4	6	7	9	11	12
4	G	9	7	5	4	5	7	9	10
5	T	8	8	6	4	4	5	7	8
6	T	9	8	7	5	3	3	5	6
7	A	11	9	7	6	4	2	3	4
8	C	13	11	9	7	5	3	1	3
9	C	14	12	10	8	6	4	2	1
10	-	16	14	12	10	8	6	4	2

图 3-20 用于找出最优对准的完整数组

接下来说明如何由整个数组来获得最优对准。首先，必须获得通向  $\text{opt}(0, 0)$  的路径。为此，首先从数组的左上角开始，并回溯各个步骤。我们查看三个可能通向  $\text{opt}(0, 0)$  的数组项，并选择给出正确值的一个。然后用选定项目重复这一过程。当出现相同分值时，可任取其一。如此操作，直到抵达右下角。图 3-20 中突出显示了获得的路径。下面说明如何获得路径中的前几个值。首先，用  $[i][j]$  来表示占据第  $i$  行第  $j$  列的数组位置。然后处理如下：

(1) 选择数组位置  $[0][0]$ ；

(2) 找出路径中的第二个数组项。

(a) 检查数组位置  $[0][1]$ 。由于我们由这一数组位置向左移动，以到达数组位置  $[0][0]$ ，所以插入一个缝隙，这意味着代价将增大 2。由此得到：

$$\text{opt}(0, 1) + 2 = 8 + 2 = 10 \neq 7$$

(b) 检查数组位置  $[1][0]$ 。由于我们由这一数组位置向上移动，以到达数组位置  $[0][0]$ ，所以插入一个缝隙，这意味着代价将增大 2。由此得到：

$$\text{opt}(1, 0) + 2 = 6 + 2 = 8 \neq 7$$

(c) 检查数组位置  $[1][1]$ 。由于我们沿对角线由这一数组位置向上移动，以到达数组位置  $[0][0]$ ，所以会使代价增加 penalty 值。因为  $x[0]=A$ ,  $y[0]=T$ , 所以  $\text{penalty}=1$ 。由此得到：

$$\text{opt}(1, 1) + 1 = 6 + 1 = 7$$

因此，路径中的第二个数组位置为  $[1][1]$ 。

或者，也可以在将项目存入数组时创建路径。也就是说，在每次存储数组元素时，都会创建一个指针，回溯指向确定其取值的数组元素。

有了路径之后，就可以像下面这样提取对准方式（注意，这些序列是按逆序生成的）。

(1) 从数组的右下角开始，沿突出显示的路径前进。

(2) 每次沿对角线移动，一到达数组位置  $[i][j]$  时，就将第  $i$  行的字符放入  $x$  序列，将第  $j$  列的字符放入  $y$  序列。

(3) 每次垂直向上移动，一到达数组位置  $[i][j]$  时，就将第  $i$  行的字符放入  $x$  序列，并将一个缝隙放入  $y$  序列。

(4) 每次垂直向左移动，一到达数组位置  $[i][j]$  时，就将第  $j$  行的字符放入  $y$  序列，并将一个缝隙放入  $x$  序列。

如果对图 3-20 中的数组遵循这一步骤，将获得下面的最优对准。

A	A	C	A	G	T	T	A	C	C
T	A	-	A	G	G	T	-	C	A

注意，如果指定不同罚分，则可能会得到不同的最优对准。Li (1997 年) 讨论了罚分的分配。

关于为序列对准问题编写动态规划算法的工作，留作习题。这个序列对准算法是由 Waterman 在 1984 年详细开发的，它是应用最广泛的序列对准方法之一，在诸如 BLAST (Bedell, 2003 年) 和 DASH (Gardner-Stephen 和 Knowles, 2004 年) 等高级序列对准系统中均使用这种方法。本节内容以 Neapolitan (2009 年) 的材料为基础。该文对分子进化遗传的讨论要详细得多。

## 3.8 习题

### 3.1 节

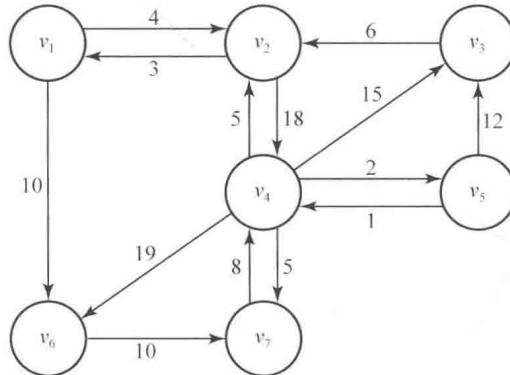
1. 推导本节给出的式 3.1。

2. 利用对  $n$  的归纳法，证明二项式系数问题（算法 3.1）的分而治之算法，基于式 3.1 计算  $\binom{n}{k}$  时，需要计算  $2\binom{n}{k} - 1$  项。

3. 在你的系统上实现二项式系数问题的两个算法(算法 3.1 和算法 3.2), 并使用不同问题实例研究它们的性能。  
 4. 修改算法 3.2(使用动态规划的二项式系数计算), 使它仅使用索引范围为 0 至  $k$  的一维数组。

### 3.2 节

5. 使用 Floyd 最短路径算法 2(算法 3.4), 为下图构建矩阵  $D$ (包含最短路径的长度) 和矩阵  $P$ (包含最短路径上中间顶点的最高索引)。给出操作步骤。



6. 利用第 5 题中得到的矩阵  $P$ , 使用输出最短路径算法(算法 3.5)在第 5 题的图中找出从顶点  $v_7$  到顶点  $v_3$  的最短路径。  
 7. 分析“输出最短路径算法”(算法 3.5), 并证明它具有线性时间复杂度。  
 8. 在你的系统上实现 Floyd 最短路径算法 2(算法 3.4), 并使用不同的图来研究其性能。  
 9. 能否修改 Floyd 最短路径算法 2(算法 3.4), 仅给出从一给定顶点到图中另一指定顶点的最短路径? 说明理由。  
 10. 能否使用 Floyd 最短路径算法 2(算法 3.4), 在一个具有负数权重的图中找出最短路径? 说明理由。

### 3.3 节

11. 找出一个最优化问题, 其中不适用最优化原理, 从而不能使用动态规划获得其最优解。说明理由。

### 3.4 节

12. 列出对五个矩阵  $A$ 、 $B$ 、 $C$ 、 $D$ 、 $E$  进行相乘的所有可能顺序。  
 13. 找出计算  $A_1 \times A_2 \times A_3 \times A_4 \times A_5$  乘积的最优顺序及其代价, 其中:

$A_1$  为  $(10 \times 4)$

$A_2$  为  $(4 \times 5)$

$A_3$  为  $(5 \times 20)$

$A_4$  为  $(20 \times 2)$

$A_5$  为  $(2 \times 50)$

给出由算法 3.6 生成的最终矩阵  $M$  和  $P$ 。

14. 在你的系统上生成实现最少量乘法算法(算法 3.6)和输出最优顺序算法(算法 3.7), 并使用不同问题实例研究其性能。  
 15. 证明: 基于式 3.5 的分而治之算法拥有指数时间复杂度。  
 16. 考虑一个问题, 计算  $n$  个矩阵相乘时有多少种顺序。  
   (a) 编写一个以整数  $n$  为输入的递归算法, 并求解此问题。当  $n$  等于 1 时, 该算法返回 1。  
   (b) 实现该算法, 并给出当  $n=2, 3, 4, 5, 6, 7, 8, 9, 10$  时的算法输出。  
 17. 推导下面的等式:

$$\sum_{\text{diagonal}=1}^{n-1} [(n-\text{diagonal}) \times \text{diagonal}] = \frac{n(n-1)(n+1)}{6}$$

在算法 3.6 的所有情况时间复杂度分析中用到了这一等式。

18. 证明：要为一个拥有  $n$  个矩阵的表达式加上括号，共需要  $n-1$  对括号。
19. 分析算法 3.7，证明它具有线性时间复杂度。
20. 编写一个高效算法，找出对  $n$  个矩阵  $A_1 \times A_2 \times \cdots \times A_n$  相乘的最优顺序，其中每个矩阵的维度为  $1 \times 1$ 、 $1 \times d$ 、 $d \times 1$  或  $d \times d$ ， $d$  为某一正整数。分析你的算法，并用阶的符号给出分析结果。

### 3.5 节

21. 使用六个各不相同的键，可以构建多少个不同的二叉查找树？
22. 为以下各项创建一个最优二叉查找树，其中每个单词的出现概率在括号中给出：CASE(0.05), ELSE(0.15), END(0.05), IF(0.35), OF(0.05), THEN(0.35)。
23. 找出一种计算  $\sum_{m=i}^j p_m$  的高效方法，在最优二叉查找树算法（算法 3.9）中用到了它。
24. 在你的系统上实现最优二叉查找树算法（算法 3.9）和构建最优二叉查找树算法（算法 3.10），并使用不同问题实例研究它们的性能。
25. 分析算法 3.10，并使用阶的符号表示其时间复杂度。
26. 将最优二叉查找树算法（算法 3.9）推广到查找键可能不在树中的情况。也就是说，应当令  $q_i$  ( $i=0, 1, 2, \dots, n$ ) 是一个缺失查找键可能位于  $\text{Key}_i$  和  $\text{Key}_{i+1}$  之间的概率。分析你推广后的算法，并用阶的符号给出分析结果。
27. 证明：基于式 3.6 的分而治之算法拥有指数时间复杂度。

### 3.6 节

28. 为用下面矩阵  $W$  表示的加权有向图找出一个最优回路。给出逐步操作。

$$W = \begin{bmatrix} 0 & 8 & 13 & 18 & 20 \\ 3 & 0 & 7 & 8 & 10 \\ 4 & 11 & 0 & 10 & 7 \\ 6 & 6 & 7 & 0 & 11 \\ 10 & 6 & 2 & 1 & 0 \end{bmatrix}$$

29. 为旅行推销员问题的动态规划算法（算法 3.11）编写一个更详细的版本。
30. 在你的系统上实现第 29 题为算法 3.11 编写的详尽版本，并使用几个问题实例研究其性能。

### 3.7 节

31. 分析 3.7 节算法 opt 的时间复杂度。
32. 为序列对准问题编写动态规划算法。
33. 假定失配的罚分为 1，缝隙的罚分为 2，使用动态规划算法找出以下序列的最优对准：

C C G G G T T A C C A  
G G A G T T C A

### 补充习题

34. 和计算第  $n$  个斐波那契项的算法一样（见第 1 章的第 34 题），算法 3.2（使用动态规划的二项式系数计算）中的输入规模是对数字  $n$  和  $k$  进行编码所需要的符号数。分析该算法相对于其输入规模的性能。
35. 计算对  $n$  个矩阵  $A_1, A_2, \dots, A_n$  进行相乘的可能顺序数。
36. 证明： $n$  键二叉查找树的数目由下式给出：

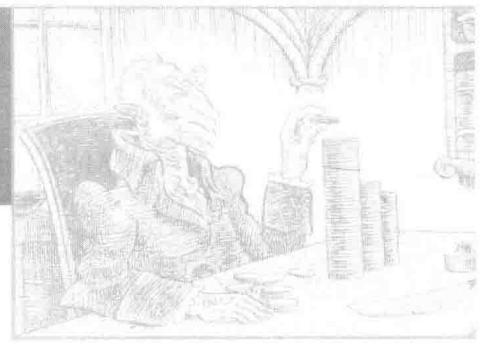
$$\frac{1}{(n+1)} \binom{2n}{n}$$

37. 能否为最优二叉查找树问题（算法 3.9）开发一种二次时间算法？
38. 使用动态规划方法编写一种算法，找出在一个给定的  $n$  实数列表中，任意连续子列表元素的最大和值。分析你的算法，并用阶的符号给出分析结果。

39. 考虑两个字符序列  $S_1$  和  $S_2$ 。例如，可以为  $S_1=A\$CMA*MN$  和  $S_2=AXMC4ANB$ 。假定为了构造一个序列的子序列，可以从任意位置删除任意数量的字符，使用动态规划方法创建一个算法，找出  $S_1$  和  $S_2$  的最长公共子序列。此算法返回每个序列的最长公共子序列。

# 第4章

## 贪婪方法



无论是在小说中，还是在现实生活中，查理·狄更斯笔下的经典人物——埃比尼泽·斯克鲁奇（Ebenezer Scrooge），可能都算得上古往今来最贪婪的人了。他每天唯一的动力就是贪婪地攫取尽可能多的金子。在“圣诞节的过去精灵”让他想起过去、“圣诞节的未来精灵”向他警告未来之后，他改变了自己的贪婪做法。

贪婪算法的方式与斯克鲁奇的做法相同。也就是说，依次获取数组项，每次都获取一个根据某一准则被认为是“最佳”的项目，而不考虑之前或未来做出的选择。千万不要因为斯克鲁奇和“贪婪”一词的负面内涵而认为“贪婪算法存在问题”。这些算法经常会得出非常高效而简单的答案。

和动态规划类似，贪婪算法也经常用于求解最优化问题。但是，贪婪方法要更直接一些。在动态规划中，递归性质用于将一个实例划分为较小实例。在贪婪算法中，不会划分为较小实例。贪婪算法（greedy algorithm）通过进行一系列选择来获得答案，每次选择只考虑当时的最佳值。也就是说，每次选择都是局部最优的。人们希望最终会得到一个全局最优的答案，但并非总能如愿。对于一种给定算法，必须判断这个答案是否总是最优的。

一个简单的例子演示了贪婪方法。Joe 是一位售货员，经常遇到找零钱的问题。顾客通常不希望收到大量硬币。例如，当找零为 0.87 美元时，如果 Joe 找给顾客 87 美分，大多数顾客都会非常恼火。因此，Joe 的目标是不仅要保证零钱数正确，还要尽可能少用硬币。对于 Joe 的找零问题，它的一个答案就是一组硬币，加起来正好等于应当找给顾客的零钱数，而最优答案就是硬币个数最少的一个集合。这一问题的贪婪方法可以按如下进行。最初，在找零中没有硬币。Joe 首先查看他能找到的最大硬币（面值）。也就是说，他判断哪个硬币最好（局部最优）的准则是硬币的面值。这在贪婪算法中被称为选择过程。接下来，他看看将这一硬币加到找零中是否会使找零总值超出应找钱数。这在贪婪算法中称为可行性检查。如果加入这枚硬币后并不会使钱数超出应找零数目，就将其加入零钱中。接下来，检查零钱数值是否等于应找钱数。这是贪婪算法中的答案检查。如果两个数值不相等，则 Joe 利用选择过程获取另一枚硬币，并重复该过程，一直到零钱数等于应找钱数，或者硬币用完为止。在后一种情况下，他无法正好找回正确的找零钱数。下面是这一过程的高级算法。

```
while (有多枚硬币，且实例尚未解决){  
    获取剩余硬币中的面额最大者；           // 选择过程  
    if (加入硬币会使零钱超出应找钱数)      // 可行性检查  
        拒绝该硬币；  
    else  
        将该硬币加入零钱中；  
        if (零钱总值等于应找钱数)            // 答案检查  
            该实例得到解决；  
}
```

在可行性检查中，当我们断定加入一枚硬币会使零钱数超出应找钱数时，我们知道通过加入这枚硬币所得到的集合，不可能给出该实例的答案。因此，这个集合是不可行的，应当拒绝。图 4-1 给出这一算法的一个示例应用。再次说明，此算法被称为“贪婪”，是因为选择过程直接贪婪地选择第二大硬币，而不考虑进行这种选择的潜在缺点。没有重新考虑一项选择的机会：一旦接受了一枚硬币，它就被永久包含在答案中；一旦一枚硬币被拒绝，它就被永久排除在答案之外。这一过程非常简单，但它能否给出一种最优答案呢？也就是说，在找零问题中，当一种答案可行时，该算法提供的答案中，其硬币数目是不是给出正确零钱所需要的最小值呢？如果这些硬币包括美国硬币（1 美分、5 美分、10 美分、25 美分、50 美分），而且每种硬币至少有一个，那只要存在答案，贪婪算法就总能返回一种最优答案。这一点在习题中证明。除了这些涉及标准美国硬币的情景之

外，还有其他一些可以由贪婪算法给出最优答案的情景。习题中研究了其中一些。注意，如果在美国硬币之外增加一枚 12 美分的硬币，那贪婪算法就不会总能给出最优答案了。图 4-2 演示了这一结果。在该图中，贪婪答案中包含五枚硬币，而最优答案中仅包含三枚硬币——10 美分、5 美分和 1 美分。



图 4-1 一种用于找零的贪婪算法

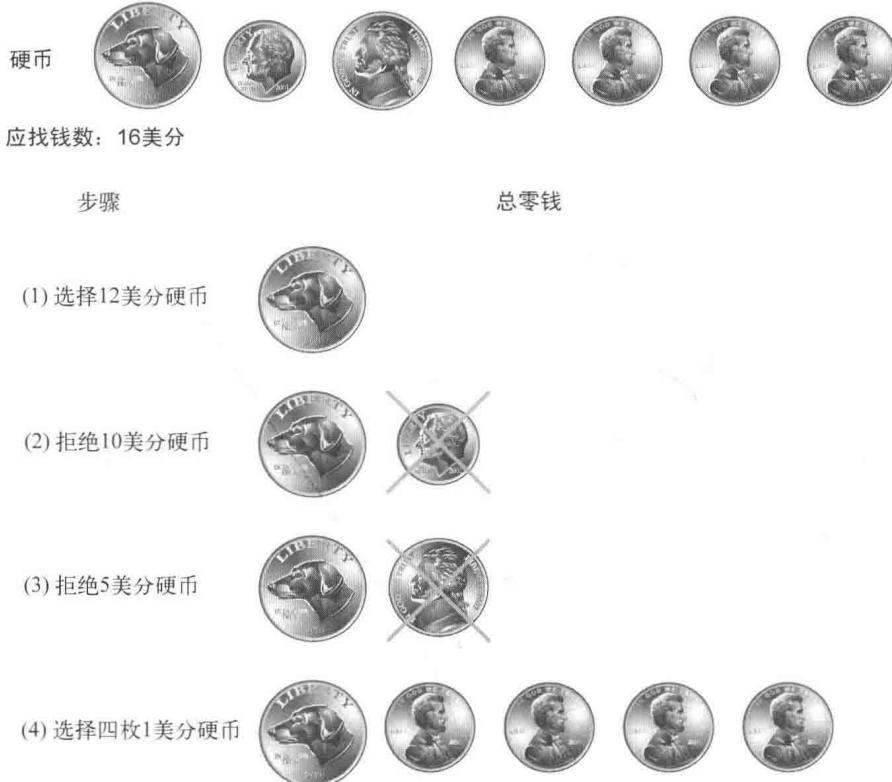


图 4-2 当包含 12 美分硬币时，贪婪算法不是最优的

这个找零问题表明，贪婪算法并不能保证给出最优答案。针对特定的贪婪算法，总得判断其能否给出最优解。4.1 节、4.2 节、4.3 节和 4.4 节讨论了贪婪方法总能给出最优解的问题。4.5 节研究了贪婪方法无法给出最优解的一种问题。该节将贪婪方法与动态规划进行了对比，以说明每种方法可能适用的时机。现在以贪婪方法的一般性概述作为结尾。贪婪算法从一个空集入手，依次向其中添加项目，直到该集合表示一个问题实例的一个答案为止。每次迭代都包括以下部分。

- **选择过程：**选择要添加到该集合中的下一个项目。该选择过程根据一条贪婪准则进行，该准则满足当时的某一局部最优考虑因素。
- **可行性检查：**检查新集合是否可能给出实例的解，以此判断该集合是否可行。
- **答案检查：**判断新集合是否构成了该实例的解。

## 4.1 最小生成树

假想一位城市规划人员希望用道路将特定城市连接起来，使人们可以由任意城市驾车到达任意其他城市。如果存在预算限制，规划人员可能希望以最少量的道路来达到这一目的。我们将设计一种算法来解决本问题及类似问题。首先再复习一些图论知识。图 4-3a 给出了一个无向加权连通图  $G$ 。这里假定这些权重都是非负数。这个图是无向图（undirected graph），因为其中的边都没有方向。在图形表示时，各条边上不带箭头。因为这些边没有方向，所以说一条边位于两个顶点之间。无向图中的一条路径（path）就是由顶点组成的一个序列，每个顶点与其后续者之间都有一条边。因为这些边没有方向，所以存在从顶点  $u$  到顶点  $v$  的路径就等价于存在从  $v$  到  $u$  的路径。因此，对于无向图，我们直接说两个顶点之间存在路径。如果每对顶点之间均存在路径，就说这个无向图是连通的（connected）。图 4-3 中的所有图都是连通的。如果从图 4-3b 的图中删除顶点  $v_2$  与  $v_4$

之间的边，该图就不再是连通图。

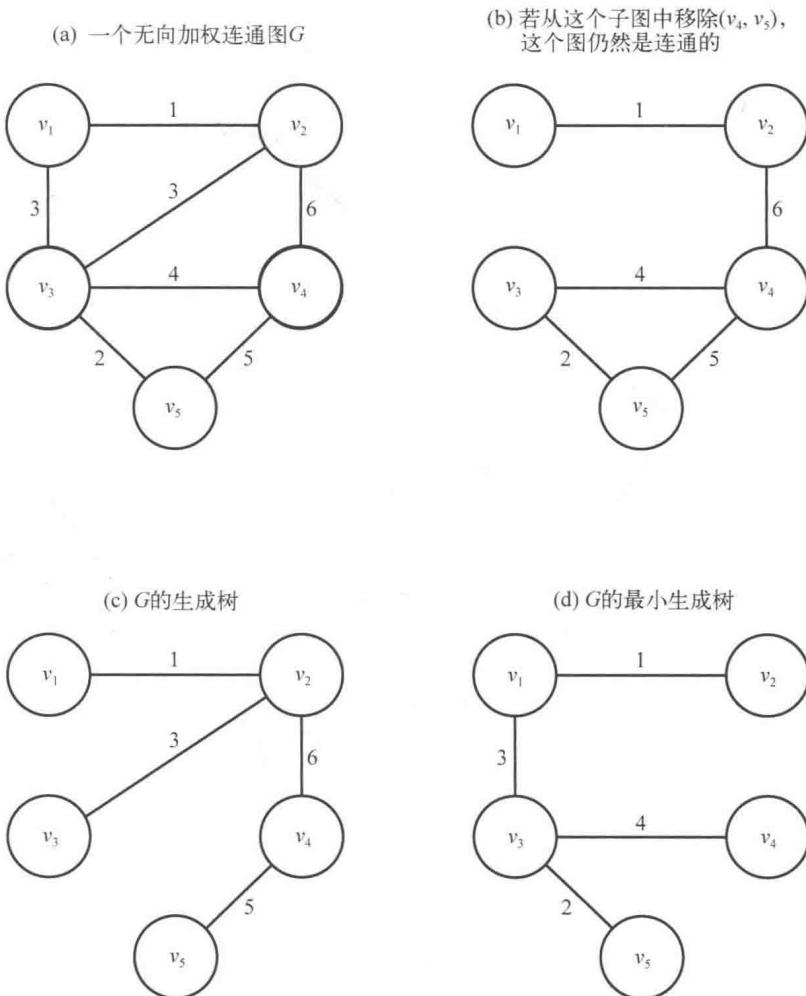


图 4-3 一个加权图和三个子图

在无向图中，从一个顶点到其自身的路径（至少包含三个顶点，而且其中所有中间顶点都互不相同）称为简单环（simple cycle）。一个没有简单环的无向图称为无环的（acyclic）。图 4-3c 和图 4-3d 是无环的，而图 4-3a 和图 4-3b 则不是无环的。树（严格来说，是自由树，free tree）是无环的、连通的无向图。图 4-3c 和图 4-3d 中的图是树。在这一定义中，没有选出哪个顶点作为根节点，而有根树（rooted tree）的定义则是一种指定某个顶点作为根节点的树。因此，有根树就是人们经常说的树（3.5 节即是如此）。

考虑这样一个问题，从一个连通加权无向图  $G$  中删除一些边，构成一个子图，其中的所有顶点仍然是连通的，而且剩余各边上的权重尽可能小。这种问题有大量应用。前面说到，在道路建设中，可能希望用最少量的道路连通一组城市。同样，在长途通信中，可能希望使用最短长度的电缆，而在探测中，可能希望使用最少量的管子。具有最小权重的子图必然是一棵树，因为如果一个子图不是树，那它就包含简单环，删除环上的任意边，都会得到一个具有更小权重的连通图。为说明这一点，请看图 4-3。图 4-3b 是图 4-3a 的一个子图，它不可能拥有最小权重，因为如果删除简单环 $[v_3, v_4, v_5, v_3]$  上的任意边，该子图仍然是连通的。例如，可以删除连接  $v_4$  与  $v_5$  的边，得到一个权重更小的连通图。

$G$  的生成树（spanning tree）是一个连通子图，其中包含了  $G$  中的所有顶点，而且是一棵树。图 4-3c 和图 4-3d 中的树都是  $G$  的生成树。具有最小权重的连通子图必然是生成树，但并非所有生成树都具有最小权重。例

如，图 4-3c 中的生成树就没有最小权重，因为图 4-3d 中的生成树具有更小的权重。上述问题的算法必须获得具有最小权重的生成树。这种树称为最小生成树（minimum spanning tree）。图 4-3d 中的树是  $G$  的一个最小生成树。一个图可能会有多个最小生成树。图  $G$  就还有另外一个最小生成树，读者可能愿意自行找出。

如果希望通过考虑所有生成树的暴力方法来找出最小生成树，其最差情况要差于指数函数。我们将使用贪婪方法以更高效率解决该问题。首先需要正式定义无向图。

**定义** 无向图  $G$  由一个有限集合  $V$  和一个集合  $E$  组成， $V$  的成员称为  $G$  中的顶点， $E$  由  $V$  中的顶点对组成。这些顶点对称为  $G$  的边。我们将  $G$  表示为：

$$G = (V, E)$$

$V$  的成员用  $v_i$  表示， $v_i$  和  $v_j$  之间的边用  $(v_i, v_j)$  表示。

**例 4.1** 对于图 4-3a 中的图，

$$\begin{aligned} V &= \{v_1, v_2, v_3, v_4, v_5\} \\ E &= \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_3, v_5), (v_4, v_5)\} \end{aligned}$$

在无向图中，为表示一条边而列出的顶点顺序是无所谓的。例如， $(v_1, v_2)$  与  $(v_2, v_1)$  表示的是同一条边。

前面列出顶点时，将索引较小的顶点放在前面。

$G$  的生成树  $T$  拥有与  $G$  一样的顶点  $V$ ，但  $T$  的边的集合是  $E$  的子集  $F$ 。我们将生成树表示为  $T = (V, F)$ 。我们的问题是，找出  $E$  的一个子集  $F$ ，使得  $T = (V, F)$  是  $G$  的一棵最小生成树。该问题的一个高级贪婪算法可按如下进行：

```

F=∅ // 将边的集合初始化为空集
while (实例尚未解决){
    根据某一局部最优因素选择一条边; // 选择过程
    if (向 F 中添加该边时没有生成坏
        添加它; // 可行性检查
    if (T=(V, F)是一个生成树) // 答案检查
        实例得到解决;
}

```

此算法只是简单地说道“根据某一局部最优因素选择一条边”。对于一个给定问题，并不存在唯一的局部最优性质。我们将研究这一问题的两种贪婪算法——Prim 算法和 Kruskal 算法。它们分别使用了不同的局部最优性质。回想一下，某一给定贪婪算法并不一定总能给出最优解。到底能否给出，必须加以证明。我们将证明，Prim 算法和 Kruskal 算法总能生成最小生成树。

### 4.1.1 Prim 算法

Prim 算法首先取边的一个空子集  $F$  和顶点的一个子集  $Y$ ，其中包含一个任意顶点。我们将  $Y$  初始化为  $\{v_1\}$ 。与  $Y$  最近的顶点是  $V - Y$  中的一个顶点，它由一条具有最小权重的边与  $Y$  中的顶点连接在一起。（回想第 3 章的内容，加权图中的权重和距离术语是可以互换使用的。）在图 4-3a 中，当  $Y = \{v_1\}$  时， $v_2$  与  $Y$  最近。这个与  $Y$  最接近的顶点被加入  $Y$  中，连接它的边被加到  $F$  中。当多个顶点的距离一致时，可任意选择。在本例中， $v_2$  被加至  $Y$  中， $(v_1, v_2)$  被加至  $F$  中。重复这一添加最近顶点的过程，直到  $Y = V$ 。下面是这一过程的高级算法。

```

F=∅; // 将边的集合初始化为空集
Y={v1}; // 将顶点的集合初始化为仅包含第一个顶点
while (实例尚未解决){
    从 V-Y 中选择一个与 Y 最接近的顶点; // 选择过程和可行性检查
    将该顶点加入 Y 中;
    将该边加入 F 中;
    if (Y == V) // 答案检查
}

```

实例解决;  
}

选择过程和可行性检查是一起完成的,因为从  $V - Y$  中取新的顶点可以保证不会生成环。图 4-4 演示了 Prim 算法。在图中的每一步,  $Y$  中包含了加阴影的顶点,  $F$  包含了加阴影的边。

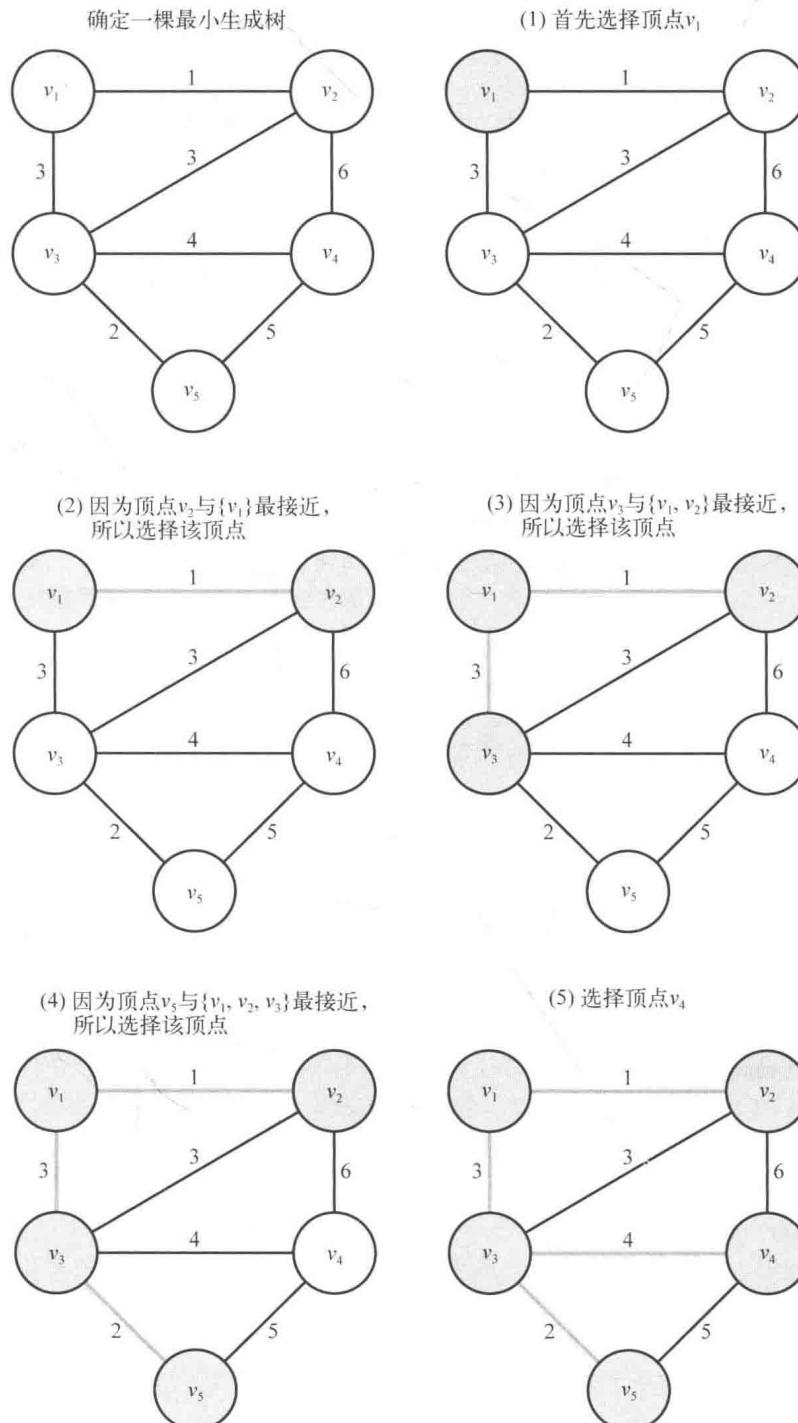


图 4-4 一个加权图(左上角)及该图 Prim 算法的步骤。每一步中,  $Y$  中的顶点和  $F$  中的边以阴影显示

利用这一高级算法，人们可以很轻松地根据一幅小型图的图示为其创建一个最小生成树。通过观察就能找出与  $Y$  最接近的顶点。但是，要编写一个可以用计算机语言实现的算法，就需要描述一个逐步过程，为此，我们用邻接矩阵来表示一个加权图。也就是说，用一个  $n \times n$  的数字数组来表示它，其中：

$$W[i][j] = \begin{cases} \text{边上的权重 (如果 } v_i \text{ 和 } v_j \text{ 之间存在边)} \\ \infty \quad \quad \quad (\text{如果 } v_i \text{ 和 } v_j \text{ 之间没有边}) \\ 0 \quad \quad \quad (\text{如果 } i=j) \end{cases}$$

图 4-3a 中的图以这种方式表示于图 4-5 中。我们维护两个数组，nearest 和 distance，其中，对于  $i=2, \dots, n$ ，  
 $\text{nearest}[i]=Y$  中与  $v_i$  最接近的顶点的索引

$\text{distance}[i]$ =一条边的权重，该边位于  $v_i$  与索引为  $\text{nearest}[i]$  的顶点之间

因为在开始时  $Y=\{v_1\}$ ，所以  $\text{nearest}[i]$  被初始化为 1， $\text{distance}[i]$  被初始化为  $v_1$  与  $v_i$  之间的边上的权重。在向  $Y$  中添加顶点时，会更新这两个数组，使之引用  $Y$  中的一个新顶点，这个新顶点与  $Y$  之外的每个顶点最为接近。为了判断要将哪个顶点加入  $Y$  中，在每次迭代中，都会计算出其  $\text{distance}[i]$  最小的索引。这个索引称为  $\text{vnear}$ 。将  $\text{distance}[\text{vnear}]$  设为 -1，以将  $\text{vnear}$  为索引的顶点加入  $Y$  中。下面的算法实现了这一过程。

	1	2	3	4	5
1	0	1	3	$\infty$	$\infty$
2	1	0	3	6	$\infty$
3	3	3	0	4	2
4	$\infty$	6	4	0	5
5	$\infty$	$\infty$	2	5	0

图 4-5 图 4-3a 所示图的数组表示  $W$

#### 算法 4.1 Prim 算法

问题：确定一棵最小生成树。

输入：整数  $n \geq 2$ ；一个包含  $n$  个顶点的连通加权无向图。该图由一个二维数组  $W$  表示，它的行列索引都是由 1 到  $n$ ，其中  $W[i][j]$  是第  $i$  个顶点与第  $j$  个顶点之间边上的权重。

输出：该图最小生成树中的边的集合  $F$ 。

```

void prim (int n,
           const number W[][],
           set_of_edges& F)
{
    index i, vnear;
    number min;
    edge e;
    index nearest[2..n];
    number distance[2..n];

    F = ∅;
    for (i = 2; i <= n; i++) {
        nearest[i] = 1; // 对于所有顶点，将  $v_1$  初始化为  $Y$  中的最近顶点
        distance[i] = W[1][i]; // 将与  $Y$  的距离初始化为到  $v_1$  的边上的权重
    }

    } repeat (n-1)次{
        min = ∞;
        for (i = 2; i <= n; i++) // 检查每个顶点是否与  $Y$  最近
            if (i <= distance[i] < min){
                vnear = i;
                min = distance[i];
            }
        F.add (edge(vnear, nearest[vnear]));
        nearest[vnear] = vnear;
        for (i = 2; i <= n; i++)
            if (W[vnear][i] < distance[i])
                distance[i] = W[vnear][i];
}

```

```

    min = distance[i];
    vnear = i;
}
e=连接两个顶点的边，这两个顶点以 vnear 和 nearest[vnear]为索引；
将 e 加入 F 中;
distance[vnear] = -1;           // 将以 vnear 为索引的顶点加入 Y 中
for (i = 2; i <= n; i++){
    if (W[i][vnear]<distance[i]{           // 对于每个不在 Y 中的顶点，更新它与 Y 的距离
        distance[i]=W[i][vnear];
        nearest[i]=vnear;
    }
}
}
}

```

#### 算法 4.1 的分析 所有情况时间复杂度 (Prim 算法)

基本运算: repeat 循环内部有两个循环, 各有  $n-1$  次迭代。可以将执行其中每个循环中的指令看作执行了一次基本运算。

输入规模:  $n$ , 顶点数。

因为 repeat 循环有  $n-1$  次迭代, 所以时间复杂度为:

$$T(n)=2(n-1)(n-1) \in \Theta(n^2)$$

显然, Prim 算法得到了一棵生成树。但是, 它一定是最小的吗? 因为每一步都选择了与  $Y$  最接近的顶点, 所以从直觉上来看, 这棵树应当是最小的。但是, 到底是否如此, 需要加以证明。尽管贪婪算法的开发通常要比动态规划算法更容易一些, 但要判断一种贪婪算法是否总能给出最优解, 通常要更难一些。回想一下, 对于一种动态规划算法, 只需要知道适用最优化原理即可。而对于贪婪算法, 通常需要进行正式证明。接下来给出 Prim 算法的这一证明。

设给定一个无向图  $G=(V, E)$ 。如果可以向  $E$  的一个子集  $F$  添加边, 使其构成一棵最小生成树, 就说子集  $F$  是有希望的 (promising)。图 4-3a 中的子集  $\{(v_1, v_2), (v_1, v_3)\}$  是有希望的, 而子集  $\{(v_2, v_4)\}$  不是有希望的。

**◆引理 4.1** 设  $G=(V, E)$  是一个连通加权无向图,  $F$  是  $E$  的有希望子集,  $Y$  是由  $F$  中的边连接的顶点集合。如果  $e$  是连接  $Y$  中一个顶点和  $V-Y$  中一个顶点的边, 而且它的权重最小, 则  $F \cup \{e\}$  是有希望的。

**证明:** 因为  $F$  是有希望的, 所以必然存在边的某个集合  $F'$ , 使得

$$F \subseteq F'$$

且  $(V, F')$  是一棵最小生成树。如果  $e \in F'$ , 则

$$F \cup \{e\} \subseteq F'$$

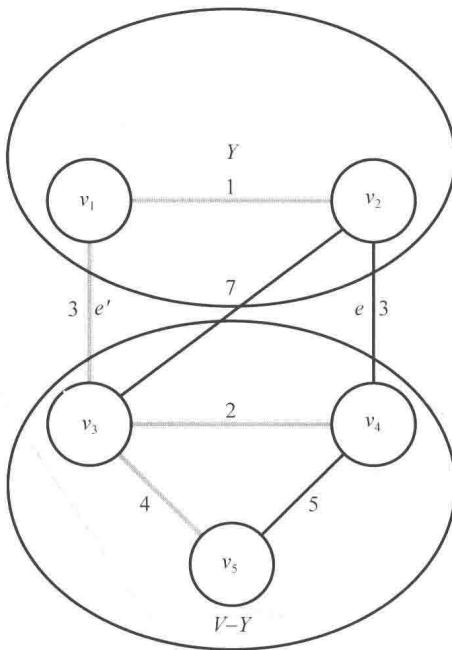
它意味着  $F \cup \{e\}$  是有希望的, 证毕。否则, 因为  $(V, F')$  是生成树, 所以  $F' \cup \{e\}$  必然恰好包含一个简单环, 而  $e$  必然在环内。图 4-6 演示了这一点, 其中的简单环是  $[v_1, v_2, v_4, v_3]$ 。在图 4-6 中可以看出, 这个简单环中必然还有另外一条边  $e' \in F'$ , 也将  $Y$  中的一个顶点连接到  $V-Y$  中的一个顶点。如果将  $e'$  从  $F' \cup \{e\}$  中移出, 则简单环消失, 也就是得到一棵生成树。因为  $e$  是将  $Y$  中的一个顶点连接到  $V-Y$  中一个顶点且具有最小权重的边, 所以  $e$  的权重必然小于或等于  $e'$  的权重 (事实上, 它们必然是相等的)。因此,

$$F' \cup \{e\} - \{e'\}$$

是一棵最小生成树。现在, 因为  $e'$  不可能在  $F$  中 (回想一下,  $F$  中的边只连接  $Y$  中的顶点), 所以:

$$F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$$

因此,  $F \cup \{e\}$  是有希望的。证毕。

图 4-6 一个辅助证明引理 4.1 的图。 $F'$  中的边以虚线显示

**定理 4.1** Prim 算法总能得出一棵最小生成树。

证明：我们用归纳法证明，在 repeat 循环的每次迭代之后，集合  $F$  是有希望的。

归纳基础：空集  $\emptyset$  显然是有希望的。

归纳假设：假定在 repeat 循环的一次给定迭代之后，当前选定的边的集合，也就是  $F$ ，是有希望的。

归纳步骤：需要证明集合  $F \cup \{e\}$  是有希望的，其中  $e$  是在下一次迭代中选定的边。因为在下一次迭代中选定的边  $e$  是连接  $Y$  中一个顶点和  $V-Y$  中一个顶点且具有最小权重的边，所以根据引理 4.1， $F \cup \{e\}$  是有希望的。归纳证毕。

根据此归纳证明，这个最终的边集是有希望的。因为这个集合包含了生成树中的边，所以这棵树必然是最小生成树。

## 4.1.2 Kruskal 算法

最小生成树问题的 Kruskal 算法首先创建  $V$  的不相交子集，每个顶点都有一个子集，且每个子集中仅包含该顶点。然后根据非递减权重检查各边（当有权重相同时，任意选择）。如果一条边连接不相交子集中的两个顶点，则添加这条边，并把这些子集合并到一个集合。一直重复这一过程，直到所有子集都合并到一个子集中。下面是这一过程的高级算法。

```

F=∅; // 将边的集合初始化为空集
创建V的不相交子集，每个顶点都有一个子集，每个子集中仅包含该顶点；

按非递减顺序对E中的边排序；

while(实例尚未解决){
    选择下一条边; // 选择过程
    if (该边连接了不相交子集中的两个顶点){
        合并子集;
        将该边添加到F中;
    }
    if (所有子集均已合并) // 答案检查
}

```

实例已解决；  
}

图 4-7 演示了 Kruskal 算法。

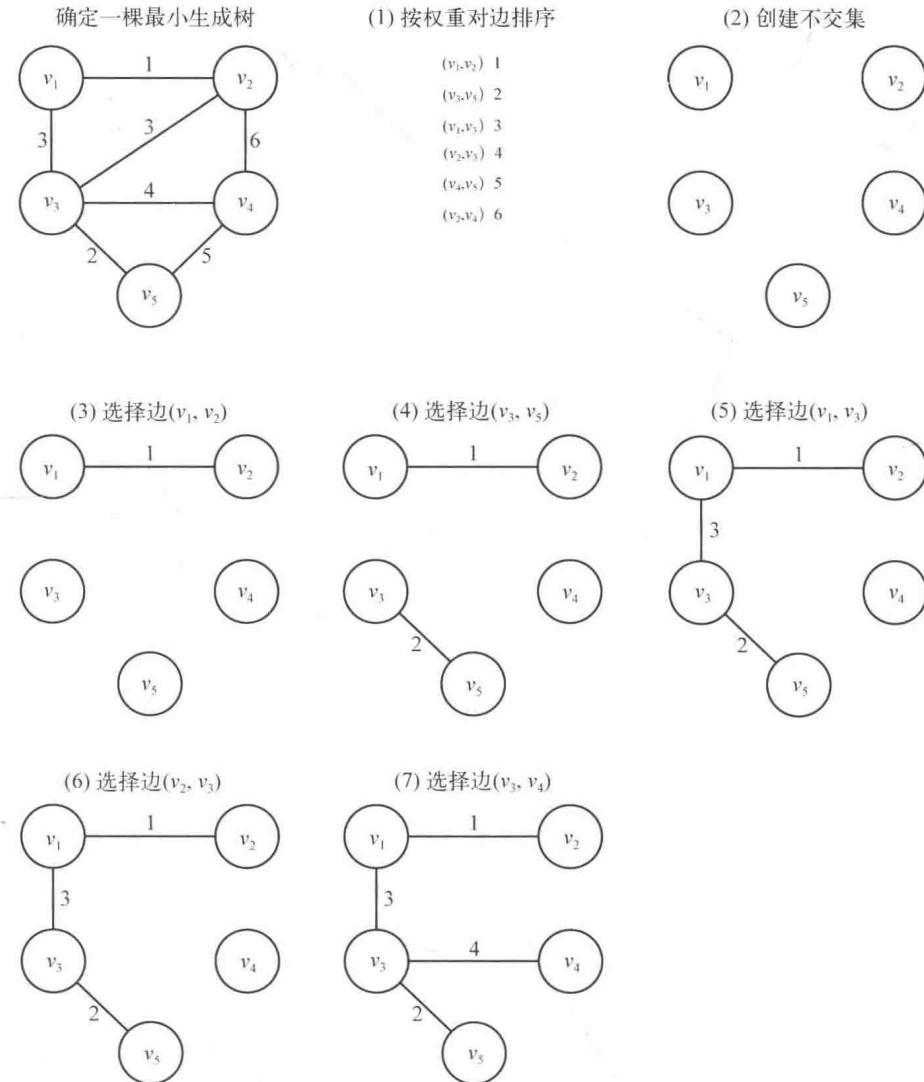


图 4-7 一个加权图（左上角）和该图 Kruskal 算法的步骤

要编写 Kruskal 算法的正式版本，需要一个不交集的抽象数据类型。附录 C 中实现了这样一种数据类型。因为该实现是为索引的不相交子集准备的，所以只需要以索引来引用这些顶点，即可使用该实现。这个不交集抽象数据类型包括数据类型 `index` 和 `set_pointer`，还有例程 `initial`、`find`、`merge` 和 `equal`。若按如下声明变量：

```
index i;
set_pointer p, q;
```

则例程的作用分别为：

- `initial(n)` 初始化  $n$  个不相交子集，每个子集中恰好包含 1 至  $n$  中的一个索引；
- `p=find(i)` 使  $p$  指向包含索引  $i$  的集合；
- `merge(p, q)` 将  $p$  和  $q$  指向的两个集合合并到集合中；
- `equal(p, q)` 在  $p$  和  $q$  指向同一集合时返回 `true`。

算法如下。

### 算法 4.2 Kruskal 算法

问题：确定一棵最小生成树。

输入：整数  $n \geq 2$ ；正整数  $m$ ；一个包含  $n$  个顶点和  $m$  条边的连通加权无向图。该图由一个集合  $E$  表示，其中包含了图中的边及其权重。

输出： $F$ ，一棵最小生成树中的边的集合。

```
void kruskal (int n, int m,
              set_of_edges E,
              set_of_edges& F)
{
    index i, j;
    set_pointer p, q;
    edge e;
    按权重的非递减顺序对 E 中的 m 条边进行排序;
    F = ∅;
    initial(n); // 初始化 n 个不相交子集。
    while (F 中的边数小于 n-1){
        e = 尚未考虑的具有最小权重的边;
        i, j = 由 e 连接的顶点的索引;
        p = find(i);
        q = find(j);
        if (! equal(p, q)){
            merge(p, q);
            将 e 添加到 F 中;
        }
    }
}
```

当  $F$  中有  $n-1$  条边时，退出 `while` 循环，这是因为生成树中有  $n-1$  条边。

### 算法 4.2 的分析 最差情况时间复杂度 (Kruskal 算法)

基本运算：一个比较指令。

输入规模：顶点个数  $n$  和边的个数  $m$ 。

此算法共有三个考虑事项。

(1) 边的排序时间。在第 2 章得到了一种排序算法（合并排序），其最差情况下的复杂度为  $\Theta(m \lg m)$ 。第 7 章将会证明，对于通过键的比较来进行排序的算法，不可能进一步提升此性能。因此，对边进行排序的时间复杂度由下式给出：

$$W(m) \in \Theta(m \lg m)$$

(2) `while` 循环中的时间。处理不交集所需要的时间在这一循环中占主导地位（因为其他一切都是常量）。在最差情况下，在退出 `while` 循环之前会考虑每一条边，这意味着该循环执行  $m$  遍。一个循环中包含对例程 `find`、`equal` 和 `merge` 的常数次调用，利用附录 C 中对“不交集数据结构 II”的实现，可以知道对这种循环执行  $m$  遍的时间复杂度为：

$$W(m) \in \Theta(m \lg m)$$

其中，基本运算是一条比较指令。

(3) 初始化  $n$  个不交集的时间。利用前面提到的不交集数据结构实现，此初始化的时间复杂度为：

$$T(n) \in \Theta(n)$$

因为  $m \geq n-1$ ，排序和不交集的处理占据初始化时间的主要部分，这意味着：

$$W(m, n) \in \Theta(m \lg m)$$

看起来，最差情况似乎与  $n$  没有关系。但在最差情况下，每个顶点都可以连接到其他每个顶点，这意味着：

$$m = \frac{n(n-1)}{2} \in \Theta(n^2)$$

因此，可以将最差情况下的时间复杂度写为：

$$w(m, n) \in \Theta(n^2 \lg n^2) = \Theta(n^2 \lg n) = \Theta(n^2 \lg n)$$

在将 Kruskal 算法与 Prim 算法进行对比时，使用这两个最差情况表达式是有帮助的。

我们需要用下面的引理来证明 Kruskal 算法总是得出一个最优答案。

**◆引理 4.2** 设  $G=(V, E)$  是一个连通加权无向图， $F$  是  $E$  的一个有希望子集， $e$  是  $E-F$  中具有最小权重且使  $F \cup \{e\}$  没有简单环的一条边，则  $F \cup \{e\}$  是有希望的。

证明：此证明与引理 4.1 中的证明类似。因为  $F$  是有希望的，所以必然存在边的某个集合  $F'$ ，使得：

$$F \subseteq F'$$

且  $(V, F')$  是一棵最小生成树。如果  $e \in F'$ ，则

$$F \cup \{e\} \subseteq F'$$

这意味着  $F \cup \{e\}$  是有希望的，证毕。否则，因为  $(V, F')$  不是生成树，所以  $F' \cup \{e\}$  必然恰好包含一个简单环， $e$  必然在此环中。因为  $F \cup \{e\}$  不包含简单环，所以必然存在某个边  $e' \in F'$ ，它在环内，但不在  $F$  中。也就是说， $e' \in E - F$ 。集合  $F \cup \{e'\}$  不是  $F'$  的子集，所以没有简单环。因此， $e$  的权重不大于  $e'$  的权重。（回想一下，我们假定  $e$  是  $E-F$  中具有最小权重且使  $F \cup \{e\}$  没有环的一条边。）如果从  $F' \cup \{e\}$  中移除  $e'$ ，这个集合中的简单环将消失，这就意味着得到了一棵生成树。事实上，

$$F' \cup \{e\} - \{e'\}$$

是一棵最小生成树，这是因为，如下所述， $e$  的权重不大于  $e'$  的权重。因为  $e'$  不在  $F$  中，所以

$$F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$$

因此， $F \cup \{e\}$  是有希望的，证毕。

**定理 4.2** Kruskal 的算法总是得出最小生成树。

证明：此证明从边的空集入手，以归纳法完成。在习题中将要求你应用引理 4.2 来完成此证明。

### 4.1.3 Prim 算法与 Kruskal 算法的比较

我们得到下面的时间复杂度。

Prim 算法： $T(n) \in \Theta(n^2)$

Kruskal 算法： $W(m, n) \in \Theta(m \lg m)$ ,  $W(m, n) \in \Theta(n^2 \lg n)$

我们还证明了，在连通图中：

$$n-1 \leq m \leq \frac{n(n-1)}{2}$$

如果一个图的边数  $m$  接近上述下限（这个图是非常稀疏的），则 Kruskal 算法为  $\Theta(n \lg n)$ ，这意味着 Kruskal 算法应当更快速一些。但是，如果一个图的边数接近上限（这个图是高度连通的），则 Kruskal 的算法为  $\Theta(n^2 \lg n)$ ，这意味着 Prim 算法应当更快一些。

### 4.1.4 最终讨论

如前所述，一个算法的时间复杂度有时会依赖于实现该算法时使用的数据结构。Johnson（1977 年）利用堆，为 Prim 算法创建了一种  $\Theta(m \lg n)$  实现。对于一个稀疏图，其时间复杂度为  $\Theta(n \lg n)$ ，相对于我们的实现有所提升。但对于稠密图，其时间复杂度为  $\Theta(n^2 \lg n)$ ，要比我们的实现更慢一些。Fredman 和 Tarjan（1987 年）利用斐波那契堆设计了 Prim 算法的最快速实现。它们的实现是  $\Theta(m + n \lg n)$ 。对于稀疏图，它是  $\Theta(n \lg n)$ ，对于稠密图，为  $\Theta(n^2)$ 。

Prim 算法最初出现在 Jarnik（1930 年）的文献中，后来在 Prim（1957 年）的文献中以同名发表。Kruskal

的算法源于 Kruskal (1956 年)。最小生成树问题的历史在 Graham 和 Hell (1985 年) 的文献中讨论。该问题的其他算法可在 Yao (1975 年) 和 Tarjan (1983 年) 的文献中找到。

## 4.2 单源最短路径的 Dijkstra 算法

3.2 节设计了一种  $\Theta(n^3)$  算法, 用于在一个加权有向图中确定从每个顶点到所有其他顶点的最短路径。如果只是希望知道从一个特定顶点到所有其他顶点的最短路径, 那这种算法就有些大材小用了。接下来, 我们将使用贪婪方法为这一问题(称为单源最短路径问题)设计一种  $\Theta(n^2)$  算法。此算法是由 Dijkstra (1959 年) 提出的。我们假定从所关注的顶点到其他每个顶点都有一条路径, 在此基础上给出该算法。对于其他情况, 只需稍做修改即可。

此算法类似于最小生成树问题的 Prim 算法。先对集合  $Y$  进行初始化, 使其仅包含一个顶点, 也就是要为其确定最短路径的顶点。假定该顶点为  $v_1$ 。将边的集合  $F$  初始化为空集。首先选择一个与  $v_1$  最接近的顶点  $v$ , 将它加到  $Y$  中, 并将边  $\langle v_1, v \rangle$  加到  $F$  中 ( $\langle v_1, v \rangle$  是指从  $v_1$  到  $v$  的有向边)。这条边显然是从  $v_1$  到  $v$  的最短路径。接下来查看从  $v_1$  到  $V - Y$  中的顶点, 且仅以  $Y$  中顶点为中间顶点的路径。这些路径中的最短者是一条最短路径(需要证明)。将这样一条路径一端的顶点添加到  $Y$  中, 将该路径上与该顶点相接触的边添加到  $F$  中。一直持续此过程, 直到  $Y$  等于  $V$ , 即所有顶点的集合。这时,  $F$  中包含了最短路径中的边。下面是这一方法的高级算法。

```

 $Y = \{v_1\};$ 
 $F = \emptyset;$ 

while(实例尚未解决){
    从  $V - Y$  中选择一个顶点  $v$ , 当仅以  $Y$  中顶点为中间顶点时,           // 选择过程和可行性检查
    从  $v_1$  到顶点  $V$  的路径最短;

    将新顶点  $v$  添加到  $Y$  中;
    将触及  $v$  的边(最短路径)添加到  $F$  中;

    if ( $Y == V$ )
        实例已解决;                                // 答案检查
    }
}

```

图 4-8 演示了 Dijkstra 算法。和 Prim 算法中的情况一样, 仅当人们在小型图上通过观察来求解问题实例时, 这一高级算法才是有效的。接下来给出详细算法。对于这一算法, 加权图用一个二维数组表示, 其做法与 3.2 节完全一样。这一算法非常类似于算法 4.1 (Prim 算法)。区别在于这里的数组不是 nearest 和 distance, 而是数组 touch 和 length, 其中, 对于  $i=2, \dots, n$ ,

$touch[i]=Y$  中顶点  $v$  的索引, 在仅以  $Y$  中顶点为中间顶点时, 边  $\langle v, v_i \rangle$  是从  $v_1$  到  $v_i$  的当前最短路径上的最后一条边。

$length[i]=$  仅以  $Y$  中顶点为中间节点时, 从  $v_1$  到  $v_i$  的当前最短路径的长度。

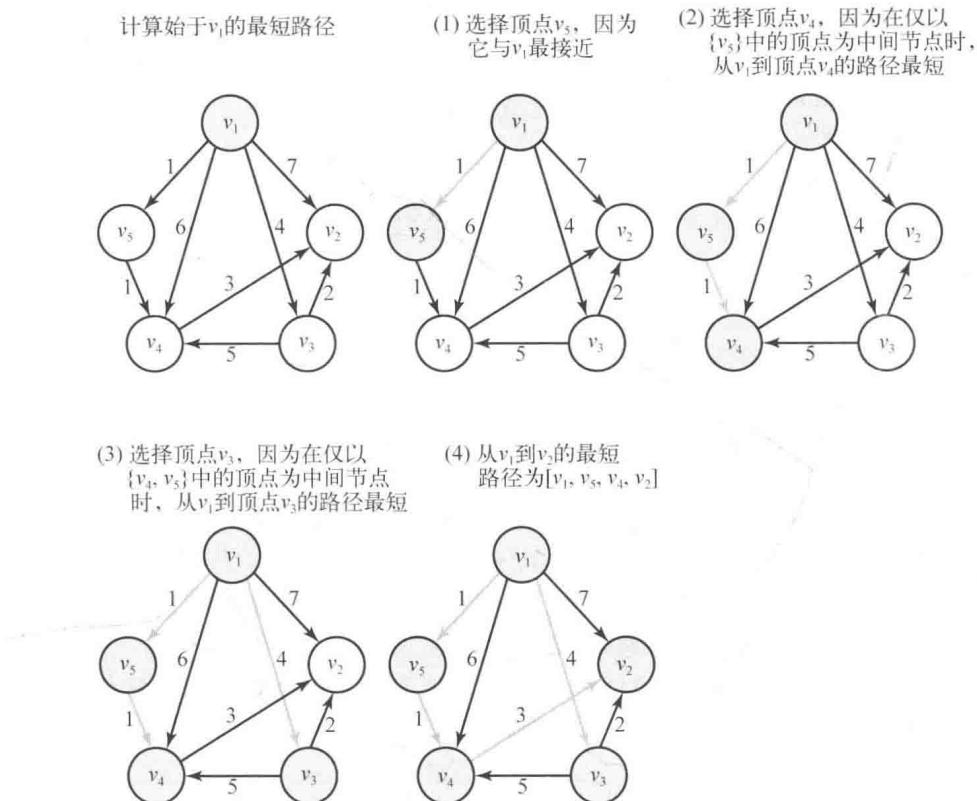


图 4-8 一个加权有向图(左上角)和该图 Dijkstra 算法中的步骤。在每一步中,  $Y$  中的顶点和  $F$  中的边以阴影显示。

算法如下。

### 算法 4.3 Dijkstra 算法

问题: 在一个加权有向图中, 确定从  $v_1$  到所有其他顶点的最短路径。

输入: 整数  $n \geq 2$ ; 一个包含  $n$  个顶点的连通加权有向图。该图用一个二维数组  $W$  表示, 它的行列索引范围都是 1 至  $n$ , 其中  $W[i][j]$  是从第  $i$  个顶点到第  $j$  个顶点的边上的权重。

输出: 边的集合  $F$ , 其中包含最短路径上的边。

```

void dijkstra(int n, const number W[][], set_of_edges& F)
{
    index i, vnear;
    edge e;
    index touch[2..n];
    number length[2..n];

    F = ∅;
    for (i = 2; i <= n; i++) {
        touch[i] = 1;
        length[i] = W[1][i];
    }

    repeat (n-1 次) {
        min = ∞;
        for (i = 2; i <= n; i++)
            if (0 <= length[i] < min) {
                min = length[i];
                vnear = i;
            }
        // 向 Y 中增加所有 n-1 个顶点。
        // 检查每个拥有最短路径的顶点。
        for (i = 2; i <= n; i++)
            if (length[vnear] + W[vnear][i] < length[i])
                length[i] = length[vnear] + W[vnear][i];
    }
}

```

```

e = 一条边，从以 touch[vnear] 为索引的顶点，到以 vnear 为索引的顶点；  

将 e 加到 F 中；  

for (i = 2; i <= n; i++)  

    if (length[vnear] + W[vnear][i] < length[i]) {  

        length[i] = length[vnear] + W[vnear][i];  

        touch[i] = vnear; // 对于不在 Y 中的每个顶点，更新其最短路径。  

    }  

    length[vnear] = -1; // 将以 vnear 为索引的顶点增加到 Y 中。  

}
}

```

因为我们假定从  $v_1$  到任意其他顶点都存在一条路径，所以在 repeat 的每次迭代中，变量 vnear 都会有一个新值。如果不是这样，根据所写的算法，最终会一遍又一遍地增加最后一条边，直到完成 repeat 循环的  $n-1$  次迭代。

算法 4.3 仅确定最短路径中的边。它并没有给出这些路径的长度。这些长度可以由这些边获得。或者，可以对算法稍做修改，使它能够计算边长，也将它们存储在一个数组中。

算法 4.3 中的控制与算法 4.1 相同。因此，由算法 4.1 的分析可知，对于算法 4.3 来说，有

$$T(n) = 2(n-1)^2 \in \Theta(n^2)$$

尽管这里并没有证明算法 4.3 总是给出最短路径，但这是可以实现的。此证明使用的归纳论证类似于证明 Prim 算法（算法 4.1）总能给出一个最小生成树时的方法。

和 Prim 算法的情景一样，Dijkstra 算法也可以使用堆或斐波那契堆实现。堆实现为  $\Theta(m\lg n)$ ，斐波那契堆实现为  $\Theta(m+n\lg n)$ ，其中  $m$  是边的个数。关于后一种实现，请参见 Fredman 和 Tarjan (1987 年) 的文献。

## 4.3 调度计划

假设一位发型师有几位客户正在等待接受不同的服务（例如，剪发、剪发加洗发、烫发、染发）。这些服务需要的时间并不完全相同，但发型师知道每种服务需要的时间。一个很自然的目标就是能合理安排客户的进程安排，使他们的等待时间和接受服务的时间之和最短。这样一种调度计划被认为是最优的。消耗在等待和接受服务上的总时间称为系统内时间（time in the system）。使系统内时间最短的问题有许多应用。例如，我们可能希望合理调度用户对磁盘驱动器的访问，使他们花在等待和接受服务上的总时间最短。

还有另一种调度问题：每项任务（每位客户）需要的完成时间相等，但每项任务都有一个最后期限，该任务必须在此期限之前启动，才能获得相关利益。我们的目的就是合理调度这些任务，使总利益最大化。我们将首先讨论使系统内总时间最短化的简单调度问题，然后再讨论带有最后期限的调度问题（scheduling with deadlines）。

### 4.3.1 使系统内总时间最短

要使系统内总时间最短，一种很简单的解决方案就是考虑所有可能存在的调度安排，并取最小值。下面的例子演示了这一方法。

**例 4.2** 假定有三项任务，这些任务的服务时间为：

$$t_1=5, t_2=10, t_3=4$$

实际的时间单位与问题无关。如果按 1、2、3 的顺序对其进行调度，则为这三项任务耗费的系统内时间如下：

任 务	系统内时间
1	5 (服务时间)
2	5 (等待任务 1 的时间) + 10 (服务时间)
3	5 (等待任务 1 的时间) + 10 (等待任务 2 的时间) + 4 (服务时间)

这一调度安排的系统内总时间为：

$$\underbrace{5}_{\text{任务1的时间}} + \underbrace{(5+10)}_{\text{任务2的时间}} + \underbrace{(5+10+4)}_{\text{任务3的时间}} = 39$$

利用这一计算方法可以给出所有调度安排及其系统内总时间，如下表：

调度安排	系统内总时间
[1, 2, 3]	$5+(5+10)+(5+10+4)=39$
[1, 3, 2]	$5+(5+4)+(5+4+10)=33$
[2, 1, 3]	$10+(10+5)+(10+5+4)=44$
[2, 3, 1]	$10+(10+4)+(10+4+5)=43$
[3, 1, 2]	$4+(4+5)+(4+5+10)=32$
[3, 2, 1]	$4+(4+10)+(4+10+5)=37$

调度安排[3, 1, 2]是最优调度，总时间为 32。

显然，一种考虑所有可能调度的算法具有阶乘时间复杂度。注意，在前面的例子中，出现最优调度的情景是：首先安排具有最短服务时间的任务（任务 3，服务时间为 4），然后是具有第二短服务时间的任务（任务 1，服务时间为 5），最后是最长服务时间的任务（任务 2，服务时间为 10）。直观来看，这种调度应当是最优的，因为它首先完成了耗费最短的任务。这一方法的高级贪婪算法如下所示：

根据服务时间的非递减顺序对任务进行排序；

```

while (实例尚未解决){
    调度安排下一项任务;           // 选择过程和可行性检查
    if (没有更多任务)           // 答案检查
        实例已解决;
}

```

我们采用贪婪算法的一般形式来书写此算法，以证明它实际上就是一种贪婪算法。但显然，该算法所做的全部工作就是根据服务时间对任务进行排序。因此，它的时间复杂度就是：

$$W(n) \in \Theta(n \lg n)$$

尽管从直观上看起来这种算法生成的调度安排是最优的，但这一假设需要证明。下面的定理证明了一个更强大的结果——这一调度安排是唯一的最优调度安排。

**定理 4.3** 可使系统内总时间最小化的唯一调度安排是根据服务时间以非递减顺序调度任务的安排。

**证明：**对于  $1 \leq i \leq n-1$ ，设  $t_i$  是在某一特定最优调度中第  $i$  项任务的服务时间（该最优调度使系统内总时间最小化）。我们需要证明，该调度是根据服务时间的非递减顺序安排任务的。我们以反证法进行证明。如果它们不是按非递减顺序安排的，则至少有一个  $i$  值 ( $1 \leq i \leq n-1$ )，满足：

$$t_i > t_{i+1}$$

可以重新调整原来的安排，将第  $i$  项和第  $i+1$  项任务交换。这样，就从第  $i+1$  项任务（指原调度安排中的序号）花费的系统内时间中移去  $t_i$  个单位。原因是，它无须等待第  $i$  项任务（指原调度安排中的序列）接受服务。同理，需要向第  $i$  项任务（指原调度安排中的序列）花费的系统内时间中增加  $t_{i+1}$  个单位。显然，这并没有改变其他任何任务花费的系统内时间。因此，如果原调度安排中的系统内总时间为  $T$ ，调整后的调度安排的总时间为  $T'$ ，有

$$T' = T + t_{i+1} - t_i$$

因为  $t_i > t_{i+1}$ ，所以

$$T' < T$$

它与原调度安排为最优的假设矛盾。

可以很轻松地推广我们的算法，以处理多服务器调度问题。假定有  $m$  个服务器。按任意方式对这些服务器进行排序。仍然按服务时间的非递减顺序对任务排序。设第一台服务器为第一项任务提供服务，第二台服

器为第二项任务提供服务……第  $m$  台服务器为第  $m$  项任务提供服务。第一台服务器将首先完成，因为该服务器执行的任务需要的服务时间最短。因此，第一台服务器将为第  $m+1$  项任务提供服务。同理，第二台服务器将为第  $m+2$  项任务提供服务，以此类推。其安排方案如下。

由服务器 1 提供服务的任务编号为：1, (1+m), (1+2m), (1+3m), …

由服务器 2 提供服务的任务编号为：2, (2+m), (2+2m), (2+3m), …

⋮

由服务器  $i$  提供服务的任务编号为： $i, (i+m), (i+2m), (i+3m), \dots$

⋮

由服务器  $m$  提供服务的任务编号为： $m, (m+m), (m+2m), (m+3m), \dots$

显然，这些任务接受处理的任务如下：

1, 2, …,  $m$ ,  $1+m$ ,  $2+m$ , …,  $m+m$ ,  $1+2m$ , …

也就是说，这些任务是根据服务时间的非递减顺序接受处理的。

### 4.3.2 带有最终期限的调度安排

在这一调度安排问题中，每项任务的完成都需要一个时间单位，而且有一个最终期限和一个收益值。如果该任务的启动时间不晚于其最终期限，则获得该收益值。并非所有任务都必须进行安排。如果有任何一种调度安排将某项任务安排在其最终期限之后，那就不必再考虑该项安排，因为它的收益值与根本不安排该任务的收益值相同。我们将这种调度安排称为不可能的。下面的例子演示了这种问题。

**例 4.3** 假定有以下任务、最终期限和收益值：

任务	最终期限	收益值
1	2	30
2	1	35
3	2	25
4	1	40

当我们说任务 1 的最终期限为 2 时，是说任务 1 可以在时刻 1 或时刻 2 开始。不存在时刻 0。因为任务 2 的最终期限为 1，所以该任务只能在时刻 1 启动。可能出现的调度安排和总收益值如下所示：

调度安排	总收益值
[1, 3]	$30+25=55$
[2, 1]	$35+30=65$
[2, 3]	$35+25=60$
[3, 1]	$25+30=55$
[4, 1]	$40+30=70$
[4, 3]	$40+25=55$

不可能的调度安排没有列出。例如，调度安排 [1, 2] 是不可能的，因此没有列出，因为任务 1 将首先在时刻 1 启动，并需要 1 个时间单位来完成，使任务 2 在时刻 2 启动。但是，任务 2 的最终期限为时刻 1。又例如，调度安排 [1, 3] 是可能的，因为任务 1 在其最终期限之前启动，而且任务 3 也在其最终期限之前启动。可以看出，调度安排 [4, 1] 是最优的，其总收益值为 70。

和例 4.3 中一样，考虑所有调度需要阶乘时间。在这个例子中要注意，具有最高收益值的任务（任务 4）包含在该最优调度安排中，而具有第二大收益值的任务（任务 2）则没有包含在内。因为这两项任务的最终期限都等于 1，所以不可能同时安排它们。当然，具有最高收益值的任务将被安排在内。另一项被安排的任务是

任务 1，因为它的收益值大于任务 3。这就让人们想到解决这一问题的一种合理贪婪方法：首先根据收益值的非递减顺序来排列任务，接下来依次查看每项任务，如果可能的话，将它添加到调度表。我们需要先有一些定义，然后才能为这一方法创建算法，哪怕是一个高级算法。

如果一个序列中的所有任务都在其最终期限之前启动，则将这个序列称为可行序列（feasible sequence）。例如，例 4.3 中的[4,1]是一个可行序列，但[1,4]不是可行序列。如果在一个任务集合中至少存在一个可行序列，则将这个集合称为可行集合（feasible set）。在例 4.3 中，{1, 4}是一种可行集合，因为安排调度序列[4, 1]是可行的，而{2, 4}则不是可行集合，因为没有一种安排调度序列允许两个任务都能在其最终期限之前启动。我们的目的是找出一个具有最大总收益值的可行序列。这样一个序列称为最优序列（optimal sequence），序列中的任务集合称为最优任务集（optimal set of jobs）。现在可以为“带有最终期限的调度”问题展示一个高级贪婪算法了。

```
根据收益值的非递减顺序对任务排序；  
S=∅  
while (实例尚未解决){  
    选择下一任务；           // 选择过程  
    if (在添加这一任务后，S 是可行集合)   // 可行性检查  
        将此任务添加到 S 中；  
    if (没有更多任务了)           // 答案检查  
        实例已解决；  
}
```

下例演示了这一算法。

**例 4.4** 假定有以下任务、最终期限和收益值：

任务	最终期限	收益值
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

在标记这些任务之前，已经对其进行了排序。前面的贪婪算法执行以下操作。

- (1)  $S$  被设定为  $\emptyset$ 。
- (2)  $S$  被设定为 {1}，因为序列 {1} 是可行的。
- (3)  $S$  被设定为 {1, 2}，因为序列 [2, 1] 是可行的。
- (4) {1, 2, 3} 被拒绝，因为对于这一集合没有可行序列。
- (5)  $S$  被设定为 {1, 2, 4}，因为序列 [2, 1, 4] 是可行的。
- (6) {1, 2, 4, 5} 被拒绝，因为对于这一集合没有可行序列。
- (7) {1, 2, 4, 6} 被拒绝，因为对于这一集合没有可行序列。
- (8) {1, 2, 4, 7} 被拒绝，因为对于这一集合没有可行序列。

$S$  的最终值为 {1, 2, 4}，这个集合的可行序列是 [2, 1, 4]。因为任务 1 和 4 的最终期限都是 3，所以也可以使用可行序列 [2, 4, 1]。

在证明这一算法总是给出最优序列之前，先来编写它的一个正式版本。为此，需要有一种高效方式来判断一个集合是否可行。要考虑所有可能序列是不可接受的，因为这样会需要阶乘时间。下面的引理使我们能够高效地检查一个集合是否可行。

**引理 4.3** 设  $S$  是一个任务集合，则  $S$  为可行集合的充要条件是：根据非递减顺序最终期限对  $S$  中的任务进行排序，所得到的序列是可行的。

**证明：**假定  $S$  是可行的。那么  $S$  中的任务至少存在一个可行序列。在这个序列中，假定任务  $x$  被安排在任务  $y$  之前，而且任务  $y$  的最终期间小于（早于）任务  $x$ 。如果将序列中的这两项任务相互交换，任务  $y$  将仍然在其最终期限之前启动，因为它启动得更早了些。而且，因为任务  $x$  的最终期限大于任务  $y$  的最终期限，且为任务  $x$  提供的新时间间隔足以执行任务  $y$ ，所以任务  $x$  也能在其最终期限之前启动。因此，新序列仍然是可行的。在对原可行序列执行交换排序（算法 1.3）时重复应用这一事实，就可以证明排序后的序列是可行的。当然，从另一个方向来说，如果排序后序列是可行的，那  $S$  也是可行的。

**例 4.5** 假定有例 4.4 中的任务。由引理 4.3 可知，要判断  $\{1, 2, 4, 7\}$  是否可行，只需要检查以下序列的可行性：

$$\begin{matrix} [2, & 7, & 1, & 4] \\ \uparrow & \uparrow & \uparrow & \uparrow \\ 1 & 2 & 3 & 3 \end{matrix}$$

每项任务的最终期限已经列在该任务的下方。因为任务 4 没有安排在其最终期限之前，所以这个序列不是可行的。根据引理 4.3，这个集合是不可行的。

算法如下。假定在将任务传送给算法之前，已经根据收益值的非递减顺序对任务进行了排序。因为只有在对任务进行排序时才需要这些收益值，所以没有将它们列为算法的参数。

#### 算法 4.4 带有最终期限的调度安排

**问题：**假定每项任务都有一个收益值，而且只有当该项任务的调度安排不晚于其最终期限时，才会获得该收益值，试确定具有最大总收益值的调度安排。

**输入：** $n$ ，任务数；整数数组  $\text{deadline}$ ，范围范围为 1 至  $n$ ，其中  $\text{deadline}[i]$  是第  $i$  项任务的最终期限。该数组已经根据各任务相关收益值的非递减顺序进行了排序。

**输出：**任务的一个最优序列  $J$ 。

```
void schedule (int n,
               const int deadline[],
               sequence_of_intenger& J)
{
    index i;
    sequence_of_intenger K;

    J=[1];
    for (i = 2; i <= n; i++){
        K = J, 其中根据 deadline[i] 的非递减顺序值添加了 i;
        if (K 是可行的)
            J = K;
    }
}
```

在分析这一算法之前，先来运用它。

**例 4.6** 假定有例 4.4 中的任务。回想一下，它们有以下最终期限：

任务	最终期限
1	3
2	1
3	1
4	3
5	1
6	3
7	2

算法 4.4 执行以下操作。

(1)  $J$  被设定为  $[1]$ 。

- (2)  $K$  被设定为  $[2, 1]$ , 且被判定为可行的。  
 $J$  被设定为  $[2, 1]$ , 因为  $K$  是可行的。
- (3)  $K$  被设定为  $[2, 3, 1]$  且被拒绝, 因为它不是可行的。
- (4)  $K$  被设定为  $[2, 1, 4]$ , 且被判定可行的。  
 $J$  被设定为  $[2, 1, 4]$ , 因为  $K$  是可行的。
- (5)  $K$  被设定为  $[2, 5, 1, 4]$  且被拒绝, 因为它不是可行的。
- (6)  $K$  被设定为  $[2, 1, 6, 4]$  且被拒绝, 因为它不是可行的。
- (7)  $K$  被设定为  $[2, 7, 1, 4]$  且被拒绝, 因为它不是可行的。
- $J$  的最终值为  $[2, 1, 4]$ 。

#### 算法 4.4 的分析 最差情况时间复杂度 (带有最终期限的调度安排)

基本运算: 这些任务的排序需要比较操作, 在将  $K$  设定为等于添加了任务  $i$  之后的  $J$  时, 需要进行更多的比较操作, 此外还需要执行比较操作来检查  $K$  是可行的。因此, 基本运算就是一条比较操作。

输入规模:  $n$ , 任务数。

在将任务传送给过程之前, 需要花费  $\Theta(n \lg n)$  的时间先对它们进行排序。在  $\text{for-}i$  循环的每次迭代中, 最多需要  $i-1$  次比较, 以将第  $i$  个任务添加到  $K$  中, 最多需要  $i$  次比较, 以检查  $K$  是否可行。因此, 最差情况是:

$$\sum_{i=2}^n [(i-1) + i] = n^2 - 1 \in \Theta(n^2)$$

第一个等式是在附录 A 的例 A.1 中获得的。因为这一时间占去排序时间的主要部分, 所以有

$$W(n) \in \Theta(n^2)$$

最后, 我们证明这一算法总会给出一种最优解。

◆ 定理 4.4 算法 4.4 总是给出一个最优任务集。

证明: 此证明利用对任务数  $n$  的归纳完成。

归纳基础: 显然, 如果只有一项任务, 此定理成立。

归纳假设: 假定该算法从前  $n$  项任务获得的任务集对于前  $n$  项任务来说是最优的。

归纳步骤: 需要证明, 该算法从前  $n+1$  次任务中获得的任务集对于前  $n+1$  项任务来说是最优的。为此, 设  $A$  是该算法从前  $n+1$  次任务获得的任务集, 设  $B$  是从前  $n+1$  项任务获得的最优任务集。此外, 设  $\text{job}(k)$  是有序任务列表中的第  $k$  项任务。

共有两种情况。

情景 1:  $B$  中不包括任务  $(n+1)$ 。

在这种情景中,  $B$  是从前  $n$  项任务获得的任务集。但是, 根据归纳假设,  $A$  中包含了从前  $n$  项任务获得的最优任务集。因此,  $B$  中任务的总收益值不可能大于  $A$  中任务的总收益值,  $A$  必然是最优的。

情景 2:  $B$  中包含任务  $(n+1)$ 。

假定  $A$  中包含任务  $(n+1)$ 。于是,

$$B=B' \cup \{\text{任务}(n+1)\} \text{ 和 } A=A' \cup \{\text{任务}(n+1)\}$$

其中  $B'$  是由前  $n$  项任务获得的一个集合,  $A'$  是该算法由前  $n$  项任务获得的集合。根据归纳假设,  $A'$  对于前  $n$  项任务是最优的。因此,

$$\begin{aligned} \text{profit}(B) &= \text{profit}(B') + \text{profit}(n+1) \\ &\leq \text{profit}(A') + \text{profit}(n+1) = \text{profit}(A) \end{aligned}$$

其中  $\text{profit}(n+1)$  是任务  $(n+1)$  的收益值,  $\text{profit}(A)$  是  $A$  中任务的总收益值。由于  $B$  对于前  $n+1$  项任务来说是最优的, 所以可以得出结论,  $A$  也是如此。

假定  $A$  并不包括任务  $(n+1)$ 。考虑  $B$  中由任务  $(n+1)$  占据的时隙。如果在算法考虑任务  $(n+1)$  时，该时隙可用，就安排该任务。因此，该时隙必然被分配给  $A$  中的某一任务，比如任务  $i_1$ 。若任务  $i_1$  在  $B$  中，则它在  $B$  中占据的任何位置都必然被  $A$  中的某一任务占据，比如任务  $i_2$ ，因为如果不是这样，算法就会将任务  $i_1$  放在该位置，将任务  $(n+1)$  放到任务  $i_1$  的位置。显然，任务  $i_2$  不等于任务  $i_1$  或任务  $(n+1)$ 。若任务  $i_2$  在  $B$  中，那它在  $B$  中占用的任何时隙必然被  $A$  中的某一任务占用，比如任务  $i_3$ ，否则，算法就会将任务  $i_2$  放在该时隙，将任务  $i_1$  放到任务  $i_2$  的时隙，将任务  $(n+1)$  放到任务  $i_1$  的时隙。显然，任务  $i_3$  不等于任务  $i_2$ 、任务  $i_1$  或任务  $(n+1)$ 。我们可以无限重复这一论证过程。由于这些调度安排是有限的，所以最后必然会找到一项任务，比如任务  $i_k$ ，它是在  $A$  中，但不在  $B$  中。否则， $A \subseteq B$ ，这意味着我们的算法本来可以用  $A$  中的任务来安排任务  $(n+1)$ 。可以修改  $B$ ，将任务  $i_1$  放到任务  $(n+1)$  的时隙，将任务  $i_2$  放到任务  $i_1$  的时隙……将任务  $i_k$  放到任务  $i_{k-1}$  的时隙。我们可以高效地用  $B$  中的任务  $i_k$  来代替任务  $(n+1)$ 。因为这些任务是按非递减顺序排列的，所以任务  $i_k$  的收益值至少与任务  $(n+1)$  的收益值一样大。因此，这样就得到了一个任务集，它的总收益值至少与  $B$  中任务的总收益值一样大。但是，这个集合是由前  $n$  项任务得到的。因此，利用归纳假设，它的总收益值不可能大于  $A$  中任务的总收益值，这就意味着  $A$  是最优的。

利用附录 C 中给出的不交集数据结构 III，有可能创建过程 `schedule`（算法 4.4 中）的一个  $\Theta(nlgm)$  版本，其中  $m$  是两个数值中的最小者——一个数是  $n$ ，另一个数是这  $n$  项任务中的最大最终期限。因为排序时间仍然是  $\Theta(nlgn)$ ，所以整个算法为  $\Theta(nlgn)$ 。这一修改将在习题中讨论。

## 4.4 霍夫曼编码

尽管辅助存储设备的容量越来越大，它们的成本也越来越低，但由于存储需求的不断增加，这些设备仍然在持续不断地被填满。给定一个数据文件，当然希望能够找到一种方式，尽可能高效地存储该文件。数据压缩（data compression）的问题就是找出一种对数据文件进行编码的高效方法。接下来将讨论一种称为霍夫曼码（Huffman code）的编码方法，还有一种为给定文件找出霍夫曼编码方法的贪婪算法。

表示文件的一种常见方法是使用二进制代码（binary code）。在这种代码中，每个字符都是由一个独一无二的二进制字符串表示，称为码字（codeword）。定长二进制代码（fixed-length binary code）用相同数量的比特表示每个字符。例如，假定有字符集  $\{a, b, c\}$ 。将使用 2 比特对每个字符编码，因为两个比特可以提供四种码字，而我们只需要三个。可以编码如下：

a:00 b:01 c:11

有了这一代码，如果文件为：

ababcbccbc (4.1)

则编码为：

000100011101010111

利用变长二进制代码（variable-length binary code）可以获得一种更高效的编码方式。这种代码可以使用不同数量的比特来表示不同字符。在当前例子中，可以将某个字符表示为 0。由于  $b$  出现得最为频繁，所以用这个码字为  $b$  编码是最为高效的。但这样的话，就不能将  $a$  编码为 00 了，因为这样就无法将一个  $a$  和两个  $b$  区分开来。此外，也不能将  $a$  编码为 01，因为在遇到 0 时，如果不查看当前数位之后的内容，就无法判断它是表示一个  $b$ ，还是一个  $a$  的起始比特。于是，可编码如下：

a: 10 b: 0 c:11 (4.2)

有了这一编码，文件 4.1 就可以编码为：

1001001100011

利用这种编码方式，只需要 13 比特来表示这个文件，而前面的编码方式需要 18 比特。不难看出，这种编码方式是最优的，因为在用二进制字符代码表示该文件时，它需要的比特数最少。

给定一个文件，最优二进制代码问题就是为文件中的字符找出一种二进制字符代码，以最少量的比特数来表示该文件。首先讨论前缀码，然后开发一种解决这一问题的霍夫曼算法。

#### 4.4.1 前缀码

一种特定类型的变长代码是前缀码（prefix code）。在前缀码中，任何一个字符的码字都不会构成另一字符码字的起始部分。例如，如果 01 是 a 的码字，则 011 不会是 b 的码字。代码 4.2 是前缀码的一个例子，定长代码也是一种前缀码。每个前缀码都可以由一个二叉树表示，它的叶子就是要编码的字符。与代码 4.2 对应的二叉树在图 4-9 中给出。前缀码的好处在于在分析文件时不需要查看当前比特之后的内容。这一点很容易从该代码的树形表示看出来。在进行解析时，首先从文件左边的第一个比特开始，也就是从树的根节点开始。依次查看这些比特，并根据遇到的比特是 0 还是 1，决定向树的左下方或右下方移动。当到达叶节点时，就得到该叶节点处的字符，然后返回到根节点，从序列中的下一比特开始重复该过程。

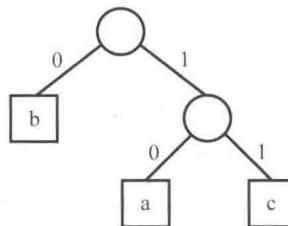


图 4-9 与代码 4.2 对应的二叉树

**例 4.7** 假定字符集为  $\{a, b, c, d, e, f\}$ ，每个字符在文件中的出现次数如表 4-1 所示。这个表中还给出三个可用于对文件进行编码的不同代码。下面计算每种编码方式的比特数：

$$\text{bits}(C1) = 16(3) + 5(3) + 12(3) + 17(3) + 10(3) + 25(3) = 255$$

$$\text{bits}(C2) = 16(2) + 5(5) + 12(4) + 17(3) + 10(5) + 25(1) = 231$$

$$\text{bits}(C3) = 16(2) + 5(4) + 12(3) + 17(2) + 10(4) + 25(2) = 212$$

可以看出，代码 C2 是对定长代码 C1 的提升，而 C3（霍夫曼代码）甚至还要优于 C2。在下一节将会看到，C3 是最优的。与代码 C2 和 C3 对应的二叉树分别在图 4-10a 和图 4-10b 中给出。树中还在每个字符旁边给出了其出现频率。

表 4-1 同一文件的三种代码。C3 是最优的

字 符	频 率	C1 (定长)	C2	C3 (霍夫曼)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

从上面的例子可以看出，给定与某一代码相对应的二叉树  $T$  后，对一个文件进行编码时所需要的比特数为：

$$\text{bits}(T) = \sum_{i=1}^n \text{frequency}(v_i) \text{depth}(v_i) \quad (4.3)$$

其中  $\{v_1, v_2, \dots, v_n\}$  是文件中的字符集，而  $\text{frequency}(v_i)$  是  $v_i$  在文件中的出现次数， $\text{depth}(v_i)$  是  $v_i$  在  $T$  中的深度。

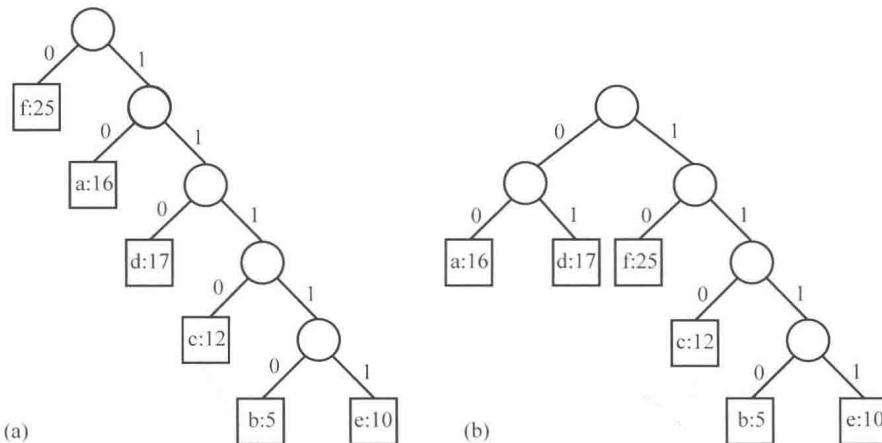


图 4-10 (a)为例 4.7 代码 C2 的二进制字符代码, (b)为代码 C3 (霍夫曼) 的二进制字符代码

#### 4.4.2 霍夫曼算法

霍夫曼开发了一种贪婪算法, 通过构建与最优代码对应的二叉树来生成最优二进制字符代码。由这种算法生成的代码称为霍夫曼码。我们将给出这一算法的高级版本。但是, 由于这一算法涉及树的构造, 所以这个算法要比其他高级算法更详细一些。具体来说, 首先声明以下类型:

```

struct nodetype
{
    char symbol; // 字符的值。
    int frequency; // 字符在文件中的出现次数。

    nodetype* left;
    nodetype* right;
};
  
```

此外, 还需要使用优先级队列。在优先级队列 (priority queue) 中, 接下来被移除的总是具有最高优先级的元素。在本例中, 具有最高优先级的元素是文件中出现频率最低的字符。优先级队列可实现为链表, 但更高效的做法是实现为堆 (关于堆的讨论, 请参阅 7.6 节)。霍夫曼算法给出如下。

$n$  = 文件中的字符数;

在优先级队列  $PQ$  中安排  $n$  个指向 **nodetype** 记录的指针, 如下所示。

对于  $PQ$  中的每个指针  $p$ ,

$p->\text{symbol}$ =文件中独一无二的一个字符;  
 $p->\text{frequency}$ =该字符在文件中的出现频率;  
 $p->\text{left} = p->\text{right} = \text{NULL}$ ;

优先级的依据是频率值, 较低的频率具有较高的优先级。

```

for ( $i = 1$ ;  $i <= n-1$ ;  $i++$ ) { // 没有答案检查;
    remove ( $PQ$ ,  $p$ ); // 而是在  $i=n-1$  时得到解。
    remove ( $PQ$ ,  $q$ ); // 选择过程。
     $r = \text{new nodetype};$  // 没有可行性检查。
     $r->\text{left} = p;$ 
     $r->\text{right} = q;$ 
     $r->\text{frequency} = p->\text{frequency} + q->\text{frequency};$ 
    insert ( $PQ$ ,  $r$ );
}
remove ( $PQ$ ,  $r$ );
return  $r$ ;
  
```

如果一个优先级队列实现为一个堆，可以在  $\theta(n)$  时间内完成其初始化。此外，每个堆操作需要  $\theta(\lg n)$  时间。由于  $\text{for-}i$  循环将执行  $n-1$  遍，所以该算法的运行时间为  $\theta(n \lg n)$ 。

接下来，将给出应用上述算法的一个例子。

**例 4.8** 假定字符集是  $\{a, b, c, d, e, f\}$ ，每个字符在文件中的出现次数如表 4-1 所示。图 4-11 给出了在每次执行  $\text{for-}i$  循环后，该算法构建的树的集合。每个节点中存储的值就是节点中 frequency 字段中的值。注意，最终的树如图 4-10b 所示。

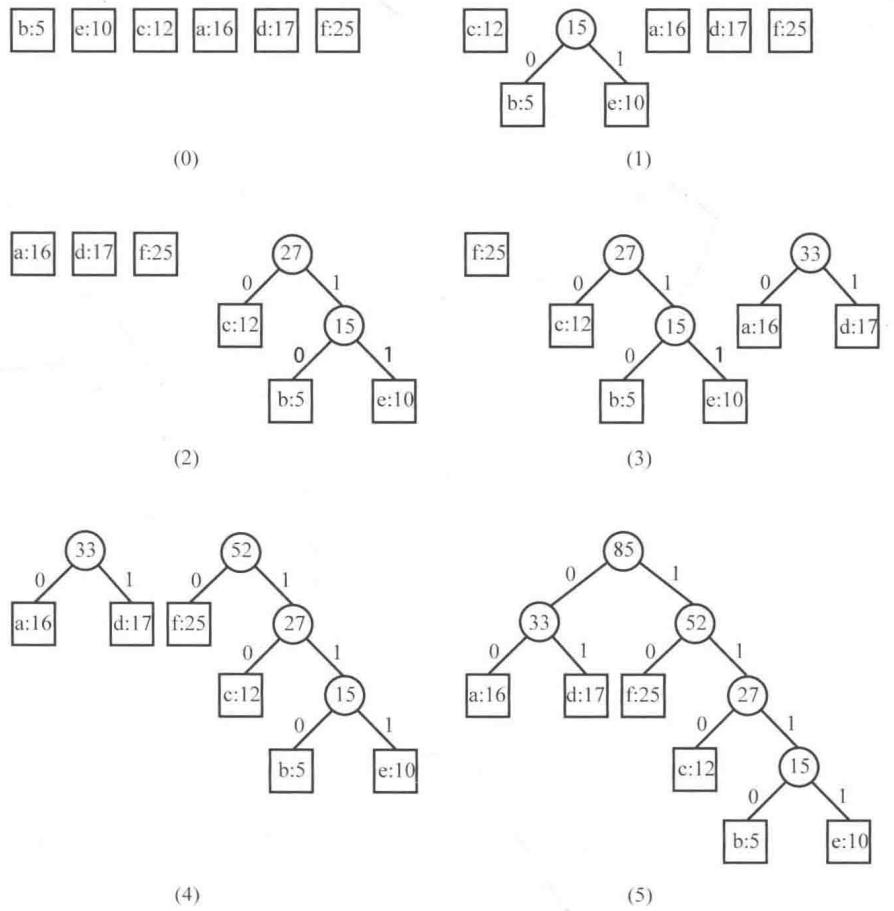


图 4-11 给定一个文件，其频率如表 4-1 所示，本图是在每次执行  $\text{for-}i$  循环后，由霍夫曼算法构建的子树状态。第一棵树是进入循环之前的状态

接下来证明此算法总是给出一种最优二进制字符代码。首先是一个引理。

**引理 4.4** 与最优二进制前缀码对应的二叉树是满的。也就是说，每个非叶节点都有两个子节点。

证明：此证明留作习题。

在继续讨论之前，需要知道一些术语。如果一棵树中的两个节点拥有同一父节点，则将这两个节点称为同胞节点（*sibling*）。树  $T$  中根节点  $v$  的一个分支（*branch*）是以  $v$  为根节点的子树。现在证明霍夫曼算法的最优化。

**定理 4.5** 霍夫曼算法生成一种最优二进制代码。

证明：此证明采用归纳法。假定第  $i$  步获得的树集是一棵与最优代码相对应的二叉树的分支，现在证明，在第  $i+1$  步获得的树集也是一棵与最优代码相对应的二叉树的分支。随后可以得出结论，在第  $n-1$  步生成的二叉树对应于一种最优代码。

归纳基础：显然，在第 0 步获得的单节点集合是一棵与最优代码相对应的二叉树的分支。

归纳假设：假定在第  $i$  步获得的树集是一棵与最优代码相对应的二叉树的分支。设这棵二叉树为  $T$ 。

归纳步骤：设  $u$  和  $v$  是霍夫曼算法第  $i+1$  步所合并树的根节点。若  $u$  和  $v$  是  $T$  中的同胞节点，那在霍夫曼算法第  $i+1$  步获得的树的集合不是  $T$  中的分支，得证。

否则，不失一般性的，假设  $u$  在  $T$  中的级别不高于  $v$ 。与前面一样，设  $\text{frequency}(v)$  是一些字符的频率之和，这些字符存储在以  $v$  为根节点的分支的叶节点处。根据引理 4.4， $u$  在  $T$  中有某一同胞节点  $w$ 。设  $S$  是在霍夫曼算法第  $i$  步之后存在的树集。显然， $T$  中以  $w$  为根节点的分支或者是  $S$  中的树之一，或者包含其中之一作为子树。无论是哪种情况，由于以  $v$  为根节点的树是由霍夫曼算法在这一步骤选择的，所以：

$$\text{frequency}(w) \geq \text{frequency}(v)$$

此外，在  $T$  中

$$\text{depth}(w) \geq \text{depth}(v)$$

交换  $T$  中分别以  $v$  和  $w$  为根节点的分支位置，可以创建一个新二叉树  $T'$ ，如图 4-12 所示。由式 4.3 和前面两个不等式不难看出：

$$\text{bits}(T') = \text{bits}(T) + [\text{depth}(w) - \text{depth}(v)][\text{frequency}(v) - \text{frequency}(w)] \leq \text{bits}(T)$$

这意味着与  $T'$  对应的代码是最优的。显然，在霍夫曼算法第  $i+1$  步获得的树的集合是  $T'$  中的分支。

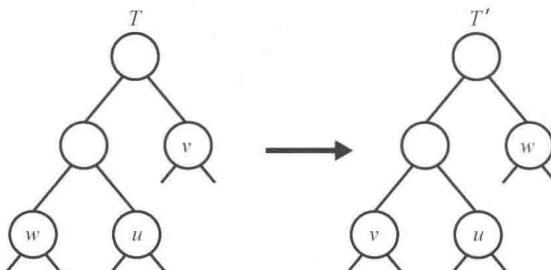


图 4-12 交换分别以  $v$  和  $w$  为根节点的分支

## 4.5 贪婪方法与动态规划的比较：背包问题

贪婪方法和动态规划是解决优化问题的两种方法。一个问题往往可以通过两种方法加以解决。例如，算法 3.3 中使用动态规划解决了单源最短路径问题，而算法 4.3 中则使用贪婪方法解决该问题。但是，动态规划在这里有些大材小用，因为它生成了始于所有来源的最短路径。我们无法修改该算法，使其更高效地仅给出单个来源的最短路径，因为无论如何都需要整个数组  $D$ 。因此，动态规划方法为该问题生成的算法是  $\Theta(n^3)$ ，而贪婪方法生成一种  $\Theta(n^2)$  算法。在用贪婪方法解决一个问题时，通常可以得到一种更简单、更高效的算法。

而另一方面，通常很难确定一种贪婪算法总能给出最优解。如“找零问题”所示，并非所有贪婪算法都能给出最优解。要表明一种具体贪婪算法总能生成最优解，需要正式证明，而证明其逆命题，只需要给出一个反例即可。回想在动态规划中，我们只需要确定最优性原理是否适用即可。

为进一步演示这两种方法之间的区别，我们将给出两种非常类似的问题——0-1 背包问题和部分背包问题。我们将开发一种贪婪算法，它能成功解决部分背包问题，但对于 0-1 背包问题则无效。然后再使用动态规划成功地解决 0-1 背包问题。

### 4.5.1 0-1 背包问题的一种贪婪方法

这种问题的一个例子是一个带着背包的贼闯入一家珠宝店。如果他偷的东西超过了某一最大重量  $W$ ，背包就会破裂。每一件东西都有特定的价值和重量。这个贼的两难之处在于，既要使所偷东西的总价值最大，又

不能使总重量超过  $W$ 。这个问题称为 0-1 背包问题。可以用公式表示如下。

假设有  $n$  件物品。设

$$S = \{\text{item}_1, \text{item}_2, \dots, \text{item}_n\}$$

$w_i$ =item<sub>i</sub>的重量

$p_i$ =item<sub>i</sub>的价值

$W$ =背包能承受的最大重量

其中  $w_i$ 、 $p_i$  和  $W$  是正整数。试确定  $S$  的一个子集  $A$ ，使得

$$\sum_{\text{item}_i \in A} p_i \text{ 最大, 且 } \sum_{\text{item}_i \in A} w_i \leq W$$

暴力解决方案是考虑  $n$  件物品的所有集合，放弃总重量超过  $W$  的子集，然后在剩下的子集中，取决总价值最高的一个。附录 A 中的例 A.10 表明，一个包含  $n$  项的集合共有  $2^n$  个子集。因此，暴力算法具有指数时间复杂度。

一种容易想到的贪婪策略就是先偷最贵重的物品；也就是说，根据价值的非递减顺序偷窃物品。但如果最贵重物品的重量与价值比太大，这一策略就不是那么有效了。例如，假定有三件物品，第一次的重量是 25 磅，价值为 10 美元，第二件、第三件物品的重量为 10 磅，价值为 9 美元。如果背包的容量  $W$  为 30 磅，这种贪婪策略只会给出 10 美元的价值，而最优解为 18 美元。

另一种容易想到的贪婪策略是先偷最轻的物品。当轻物品的价值重量比很小时，这种策略的效果极差。

为避免前两种贪婪方法的缺陷，一种更周密的贪婪策略是首先盗窃单位重量下价值最大的物品。也就是说，根据单位重量的价值对物品进行非递减顺序排列，然后依次选择。如果一件物品的重量不会使总重量超过  $W$ ，就将其放在背包里。这种方法在图 4-13 中显示。在该图中，每件物品的重量和价值列在该物品的旁边， $W$  的值为 30，列在背包中。有以下单位重量的价值。

$$\text{item}_1: \frac{50 \text{ 美元}}{5} = 10 \text{ 美元} \quad \text{item}_2: \frac{60 \text{ 美元}}{10} = 6 \text{ 美元} \quad \text{item}_3: \frac{140 \text{ 美元}}{20} = 7 \text{ 美元}$$

根据单位重量的价值对这些物品排序，得：

item<sub>1</sub>, item<sub>3</sub>, item<sub>2</sub>

在图中可以看出，这种贪婪方法选择 item<sub>1</sub> 和 item<sub>3</sub>，得到的总价值为 190 美元，而最优解是选择 item<sub>2</sub> 和 item<sub>3</sub>，得到的总价值为 200 美元。问题在于，在选择 item<sub>1</sub> 和 item<sub>3</sub> 之后，还剩下 5 磅的容量，但因为 item<sub>2</sub> 的重量为 10 磅，所以这 5 磅的剩余容量即被浪费。即使是这一更为周密的贪婪算法也没有解决 0-1 背包问题。

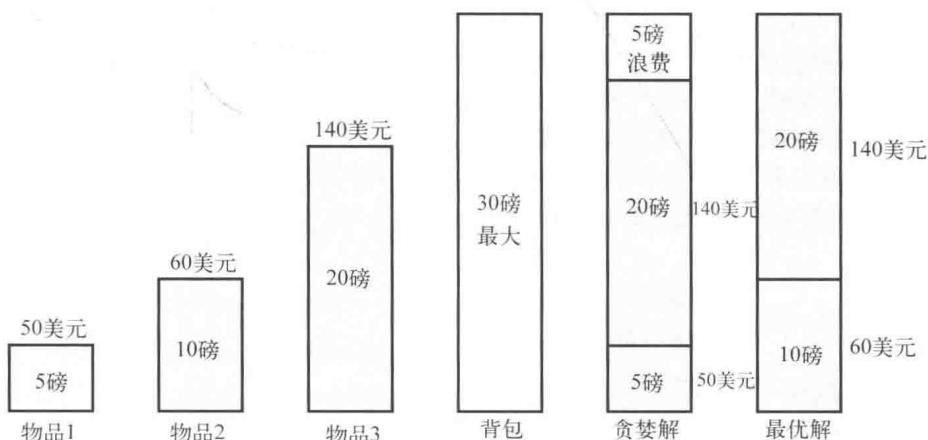


图 4-13 0-1 背包问题的贪婪解和最优解

### 4.5.2 部分背包问题的贪婪方法

在部分背包问题中，窃贼不必偷走整个一件物品，而是可以偷走该物品的任意部分。我们可以将0-1背包问题中的物品想象为金锭或银锭，而部分背包问题中的物品可以是金粉或银粉袋。假定有图4-13中的物品。如果我们的贪婪策略仍然是首先选择单位重量下价值最大的物品，则和前面一样，仍然会取走item<sub>1</sub>和item<sub>3</sub>的全部。但是，我们可以利用剩下的5磅容量取走item<sub>2</sub>的5/10。得到的总价值为：

$$50\text{美元} + 140\text{美元} + \frac{5}{10}(60\text{美元}) = 220\text{美元}$$

在部分背包问题中，我们的贪婪算法从来不会像在0-1背包问题中那样浪费任何容量。因此，它总是给出最优解。习题中将要求读者证明这一点。

### 4.5.3 0-1背包问题的动态规划方法

如果可以证明最优性原理适用，就可以使用动态规划解决0-1背包问题。为此，设A是n件物品的一个最优子集。有两种情况：A中或者包括item<sub>n</sub>，或者不包括。如果A中不包括item<sub>n</sub>，A就等于前n-1件物品的最优子集。如果A中包含item<sub>n</sub>，则A中物品的总价值就等于p<sub>n</sub>加上从前n-1件物品所选物品的最优价值，当然，这一选择要满足约束条件：总重量不超过W-w<sub>n</sub>。因此，最优性原理适用。

刚刚获得的结果可推广如下。如果对于i>0和w>0，令P[i][w]表示在总重量不超过w的约束条件下，仅从前i项物品中进行选择时获得的最优价值，则

$$P[i][w] = \begin{cases} \max(P[i-1][w], p_i + P[i-1][w - w_i]) & (w_i \leq w) \\ P[i-1][w] & (w_i > w) \end{cases}$$

最大价值等于P[n][W]。可以利用一个二维数组P确定这一价值，该数组的行索引范围为0至n，列索引范围为0至W。我们利用上面P[i][w]的表达式依次计算该数组各行中的值。P[0][w]和P[i][0]的值被设定为0。习题中会要求实际编写此算法。容易计算得出，该数组中的项数为：

$$nW \in \Theta(nW)$$

### 4.5.4 0-1背包问题动态规划算法的改进

由上面的表达式可以看出，数组项目个数的计算复杂度是n的线性函数，这一事实可能会让人们误认为这一算法对于所有包含n件物品的实例都是高效的。事实并非如此。表达式中的另一项为W，而且n和W之间没有关系。因此，对于给定n值，如果取任意大的W值，就能得到一些实例，其运行时间为任意大。例如，如果W等于n!，所计算的项数就是 $\Theta(n \times n!)$ 。若n=20，W=20!，这一算法在当今计算机上就会需要数千年的运行时间。当W相对于n而言极大时，这一算法要劣于直接考虑所有子集的暴力算法。

可以改进这一算法，使最差情况下计算的项目数属于 $\Theta(2^n)$ 。进行这一改进后，其性能永远不会差于暴力算法，而且通常还要好得多。其改进基于这样一个事实：我们并不需要对于1至W之间的每个w都计算第i行中的项目。此外，在第n行中，也只需要确定P[n][W]。因此，第n-1行只需要在计算P[n][W]时用到的项目。因为

$$P[n][W] = \begin{cases} \max(P[n-1][W], p_n + P[n-1][W - w_n]) & (w_n \leq W) \\ P[n-1][W] & (w_n > W) \end{cases}$$

所以第n-1行所需要的项目只有：

$$P[n-1][W] \text{ 和 } P[n-1][W - w_n]$$

我们从n持续逆向工作，以判断需要哪些项目。也就是说，在确定第i行需要哪些项目之后，我们用以下事实来确定第i-1行需要哪些项目：

$P[i][w]$ 由  $P[i-1][w]$  和  $P[i-1][w-w_i]$  计算得出。

当  $n=0$  或  $w \leq 0$  时停止计算。在确定所需要的项目之后，从第一行开始计算。下面的例子演示这一方法。

例 4.9 假定有图 4-13 中的项目，且  $W=30$ 。首先确定每一行中需要哪些项目。

确定第 3 行需要的项目：

需要

$$P[3][W] = P[3][30]$$

确定第 2 行需要的项目：

为计算  $P[3][30]$ ，需要

$$P[3-1][30] = P[2][30], \text{ 且 } P[3-1][30-w_3] = P[2][10]$$

确定第 1 行需要的项目：

为计算  $P[2][30]$ ，需要

$$P[2-1][30] = P[1][30], \text{ 且 } P[2-1][30-w_2] = P[1][20]$$

为计算  $P[2][10]$ ，需要

$$P[2-1][10] = P[1][10], \text{ 且 } P[2-1][10-w_1] = P[1][0]$$

接下来进行以下计算。

计算第 1 行：

$$\begin{aligned} P[1][W] &= \begin{cases} \max(P[0][w], 50 \text{ 美元} + P[0][w-5]) & (w_1 = 5 \leq w) \\ P[0][w] & (w_1 = 5 > w) \end{cases} \\ &= \begin{cases} 50 \text{ 美元} & (w_1 = 5 \leq w) \\ 50 \text{ 美元} & (w_1 = 5 > w) \end{cases} \end{aligned}$$

因此，

$$P[1][0] = 0 \text{ 美元}$$

$$P[1][10] = 50 \text{ 美元}$$

$$P[1][20] = 50 \text{ 美元}$$

$$P[1][30] = 50 \text{ 美元}$$

计算第 2 行：

$$\begin{aligned} P[2][10] &= \begin{cases} \max(P[1][10], 60 \text{ 美元} + P[1][0]) & (w_2 = 10 \leq 10) \\ P[1][10] & (w_2 = 10 > 10) \end{cases} \\ &= 60 \text{ 美元} \\ P[2][30] &= \begin{cases} \max(P[1][30], 60 \text{ 美元} + P[1][20]) & (w_2 = 10 \leq 30) \\ P[1][30] & (w_2 = 10 > 30) \end{cases} \\ &= 60 \text{ 美元} + 50 \text{ 美元} = 110 \text{ 美元} \end{aligned}$$

计算第 3 行：

$$\begin{aligned} P[3][30] &= \begin{cases} \max(P[2][30], 140 \text{ 美元} + P[2][10]) & (w_3 = 20 \leq 30) \\ P[2][30] & (w_3 = 20 > 30) \end{cases} \\ &= 140 \text{ 美元} + 60 \text{ 美元} = 200 \text{ 美元} \end{aligned}$$

该算法的这一版本仅计算 7 项，而原版本将计算  $3 \times 3 = 90$  项。

现在来确定这一版本在最差情况下的效率如何。注意，在第  $n-i$  行最多计算  $2^i$  项。因此，最多计算的项目数为：

$$1 + 2 + 2^2 + \dots + 2^{n-1} = 2^n - 1$$

此等式是在附录 A 的例 A.3 中获得的。以下实例大约需要计算  $2^n$  个项目（其价值可以为任意值），其证明留作练习：

$$w_i = 2^{i-1} \quad (1 \leq i \leq n), \quad W = 2^n - 2$$

合并这两个结果，可以得出结论，在最差情况下计算的项数为：

$$\Theta(2^n)$$

上述限度仅与  $n$  有关。现在来获得一个结合了  $n$  和  $W$  的限度。我们知道所计算的项数属于  $O(nW)$ ，但也许这一版本可以避免达到这一限制。事实并非如此。在习题中将会证明，如果  $n=W+1$ ，且对于所有  $i$  值，都有  $w_i=1$ ，则计算的总项数大约为：

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2} = \frac{(W+1)(n+1)}{2}$$

第一个等式在附录 A 中的例 A.1 获得，第二个等式源于一个事实：在此实例中， $n=W+1$ 。因此，对于任意大的  $n$  和  $W$  值，是可能达到这一界限的，这意味着，在最差情况下计算的项数属于：

$$\Theta(nW)$$

结合前面的两个结果，可知最差情况下计算的项数属于：

$$O(\min(2^n, nW))$$

这个算法的实现并不需要创建整个数组，而是可以仅存储会用到的项目。整个数组是隐式存在的。如果以这种方式实现算法，最差情况下的内存使用也有相同界限。

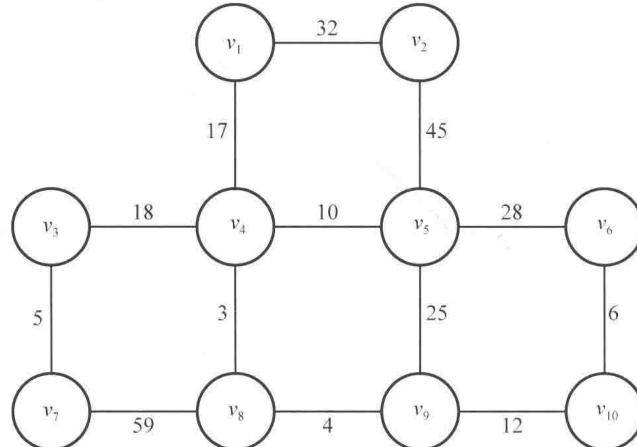
利用在开发动态规划算法时使用的  $P[i][w]$  表达式，还可以编写分而治之算法。对于这种算法，最差情况下计算的项数也属于  $\Theta(2^n)$ 。动态规划算法的主要好处在于增加了用  $nW$  表示的界限，分而治之算法没有这一界限。实际上，这一界限的获得就是因为动态规划与分而治之方法之间的根本性区别。也就是说，动态规划不会多次处理同一实例。当  $W$  与  $n$  相比不是很大时，这个用  $nW$  表示的界限是非常重要的。

和旅行推销员问题一样，也从来没有人能为 0-1 背包问题找出一种算法，使其最差时间复杂度优于指数函数，但也还没人能证明这种算法是不存在的。第 9 章集中讨论了此类问题。

## 4.6 习题

### 4.1 节

- 对于找零问题，当硬币面额为  $D^0, D^1, D^2, \dots, D^i$  ( $D$  和  $i$  为大于 0 的整数) 时，证明：贪婪方法总能找到一种最优解。
- 利用 Prim 算法（算法 4.1）为下图找出一棵最小生成树。给出操作步骤。



3. 考虑以下数组：

	1	2	3	4	5	6
1	0	$\infty$	72	50	90	35
2	$\infty$	0	71	70	73	75
3	72	71	0	$\infty$	77	90
4	50	70	$\infty$	0	60	40
5	90	73	77	60	0	80
6	35	75	90	40	80	0

(a) 从顶点  $v_4$  开始，执行 Prim 算法，为此数组表示的图找出最小生成树。

(b) 给出构成最小生成树的边的集合。

(c) 最小生成树的代价是多少？

4. 绘制一个拥有多个最小生成树的图。

5. 在你的系统上实现 Prim 算法（算法 4.1），并使用不同的图研究其性能。

6. 修改 Prim 算法（算法 4.1），以检查一个无向加权图是否是连通图。分析你的算法，并用阶的符号给出分析结果。

7. 使用 Kruskal 算法（算法 4.2）为第 2 题中的图找出最小生成树。给出逐步操作步骤。

8. 在你的系统上实现 Kruskal 算法（算法 4.1），并使用不同的图研究其性能。

9. 你认为一棵最小生成树是否可能存在环？说明理由。

10. 假定在一个计算机网络中，任意两台计算机都可以互相链接。给定每种可能链接的代价估计值，应当使用算法 4.1（Prim 算法）还是算法 4.2（Kruskal 算法）？说明理由。

11. 应用引理 4.2 完成定理 4.2 的证明。

#### 4.2 节

12. 对于由第 3 题中数组表示的图，使用 Dijkstra 算法（算法 4.3）找出从顶点  $v_5$  到所有其他顶点的最短路径。给出逐步操作步骤。

13. 使用 Dijkstra 算法（算法 4.3）找出第 2 题所示图中从顶点  $v_4$  到所有其他顶点的最短路径。给出逐步操作步骤。假定每个无方向边表示两个具有相同权值的有方向边。

14. 在你的系统上实现 Kruskal 算法（算法 4.3），并使用不同的图研究其性能。

15. 修改 Dijkstra 算法（算法 4.3），使它计算最短路径的长度。分析修改后的算法，并用阶的符号给出分析结果。

16. 修改 Dijkstra 算法（算法 4.3），检查一个有向图是否存在环。分析你的算法，并用阶的符号给出分析结果。

17. 在具有某些负数权重的图中，能否使用 Dijkstra 算法（算法 4.3）找出最短路径？说明理由。

18. 使用归纳法证明 Dijkstra 算法的正确性（算法 4.3）。

#### 4.3 节

19. 考虑以下任务和服务时间。使用 4.3.1 节的算法使系统内花费的总时间降至最低。

任务	服务时间
1	7
2	3
3	10
4	5

20. 在你的系统上实现 4.3.1 节的算法，并针对第 17 题的实例运行它。

21. 4.3.1 节将单服务器调度安排问题推广为多服务器调度安排问题，试为其编写一个算法。分析你的算法，并用阶的符号给出分析结果。

22. 考虑以下任务、最终期限和收益值。使用带有最终期限的调度安排算法（算法 4.4），使总收益值最大化。

任务	最终期限	收益值
1	2	40
2	4	15
3	3	60
4	2	20
5	3	10
6	1	45
7	1	55

23. 考虑带有最终期限的调度安排算法（算法 4.4）中的过程 `schedule`。设  $d$  是  $n$  项任务中最终期限的最大值。

修改此过程，使它尽可能晚地将一项任务添加到正在创建的调度安排表中，但不能晚于其最终期限。为此，初始化  $d+1$  个不交集，其中分别包含整数  $0, 1, \dots, d$ 。设  $\text{small}(S)$  是集合  $S$  的最小成员。在调度安排一项任务时，找出一个集合  $S$ ，其中包含该项任务的最终期限与  $n$  的最小值。如果  $\text{small}(S)=0$ ，则拒绝该任务。否则，将其安排在时刻  $\text{small}(S)$ ，并将  $S$  与包含  $\text{small}(S)-1$  的集合合并。假定我们使用附录 C 中的不交集数据结构 III，证明这一版本的复杂度为  $\theta(nlgm)$ ，其中  $m$  是  $d$  和  $n$  的最小值。

24. 实现在第 21 题开发的算法。

25. 有  $n$  个长度分别为  $l_1, l_2, \dots, l_n$  的文件，假定我们要将这些文件在磁带上的平均存储时间缩至最短。如果请求文件  $k$  的概率为  $p_k$ ，则依照  $k_1, k_2, \dots, k_n$  顺序加载这  $n$  个文件的预期访问时间由下式给出：

$$T_{\text{average}} = C \sum_{j=1}^n \left( p_{k_j} \sum_{i=1}^j l_{k_i} \right)$$

常数  $C$  表示诸如磁盘速度和记录密度等参数。

- (a) 一种贪婪方法应当按照什么顺序存储这些文件，才能保证平均访问时间最短？
- (b) 编写存储这些文件的算法，分析你的算法，并用阶的符号给出分析结果。

#### 4.4 节

26. 用霍夫曼算法为下表中的字母构建一个最小二进制前缀码。

字母： A B I M S X Z

频率： 12 7 18 10 8 5 2

27. 用霍夫曼算法为下表中的字母构建一个最小二进制前缀码。

字母： c e i r s t x

频率： 0.11 0.22 0.16 0.12 0.15 0.10 0.14

28. 用第 26 题的二进制代码对以下每个比特串解码。

- (a) 01100010101010
- (b) 1000100001010
- (c) 11100100111101
- (d) 1000010011100

29. 用第 27 题的二进制代码为每个单词编码。

- (a) rise
- (b) exit
- (c) text
- (d) exercise

30.  $a, b, c, d, e$  的代码由  $a:00, b:01, c:101, d:x10, e:yz1$  给出，其中  $x, y, z$  取 0 或 1。判断  $x, y$  和  $z$ ，使给定代码为一种前缀码。

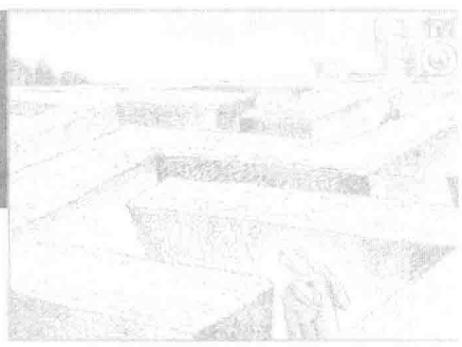
31. 实现霍夫曼算法，并对第 24 题和第 25 题的问题实例运行它。
32. 证明：与最优二进制前缀代码对应的二叉树必然是满的。满二叉树中的每个节点要么是叶节点，要么有两个子节点。
33. 证明：对于一个最优二叉前缀码，如果对字符进行排序，使其频率为非递减顺序，则它们的码字长度也是非递减的。
34. 给定与一种二进制前缀码对应的二叉树，编写一个算法，确定所有字符的码字。确定算法的时间复杂度。

#### 4.5 节

35. 为 0-1 背包问题编写动态规划算法。
36. 使用一种贪婪方法构造一种最优二叉查找树：为根节点考虑最可能的键  $\text{Key}_k$ ，以相同方式，为  $\text{Key}_1, \text{Key}_2, \dots, \text{Key}_{k-1}$  和  $\text{Key}_{k+1}, \text{Key}_{k+2}, \dots, \text{Key}_n$  递归构建左右子树。
  - (a) 假定这些键已经是有序的，这一方法的最差时间复杂度为多少？说明理由。
  - (b) 用一个例子来说明，这一贪婪方法并不总是给出一种最优二叉查找树。
37. 假定为  $n$  项任务分配  $n$  个人。设  $C_{ij}$  是将第  $i$  个人指定给第  $j$  项任务的代价。使用一种贪婪方法编写一个算法，找出一种分配方式，使得将所有  $n$  个人分配给所有  $n$  项任务的总代价最小化。分析你的算法，并以阶的符号给出分析结果。
38. 使用动态规划方法为第 26 题的问题编写一种算法。分析你的算法，并以阶的符号给出分析结果。
39. 使用一种贪婪方法编写一种算法，使得在合并  $n$  个文件的问题中，所移动的记录数最少。采用一种双向合并方式（在每个合并步骤中合并两个文件）。分析你的算法，并以阶的符号给出分析结果。
40. 使用动态规划方法为第 28 题编写一个算法。分析你的算法，并以阶的符号给出分析结果。
41. 证明部分背包问题的贪婪方法可给出最优解。
42. 证明：0-1 背包问题的改进动态规划算法在最差情况下计算的项目数属于  $\Omega(2^n)$ 。为进行这一证明：考虑一个实例，其中  $W=2^n-2$  且  $w_i=2^{i-1}$  ( $1 \leq i \leq n$ )。
43. 证明：在 0-1 背包问题的改进动态规划算法中，当  $n=W+1$  及对于所有  $i$  值都有  $w_i=1$  时，所计算的总项目数大约为  $(W+1) \times (n+1)/2$ 。

#### 补充习题

44. 请用一个反例证明：对于找零问题，当硬币为美国硬币，而且无法保证每种硬币都至少有一枚时，贪婪方法无法总给出最优解。
45. 证明：一个完全图（每对顶点之间都有一条边的图）有  $n^{n-2}$  棵生成树。 $n$  是图中的顶点个数。
46. 使用一种贪婪方法为旅行推销员问题编写一个算法。证明你的算法并不总会找出最短旅程。
47. 证明：第 21 题的多服务者调度安排问题的算法总能找出一种最优调度安排。
48. 在不构造霍夫曼树的情况下，为给定字符集生成霍夫曼码。
49. 将霍夫曼算法推广到三进制码字，并证明它能生成最优三进制码。
50. 证明：如果根据出现频率对字符进行排序，则可以在线性时间内构建出霍夫曼树。



# 回溯

如果你正在英国汉普敦宫旁边的著名树篱迷宫里寻找出路，那你可能没有别的选择，只能沿着一条毫无希望的小径一直走到死胡同。然后，你会退回到分岔口，探索另一条路径。所有曾经尝试过解决迷宫难题的人都曾体验过走进死胡同的挫折。想象一下，如果有一个标志，放在一条路径的不远处，告诉你这条路径只会通向死胡同。如果这个标志放在接近路径起点的位置，会节省很多时间，因为这个标志之后的所有分岔都会不予考虑。这意味着将会避免的不是一个死胡同，而是很多个。在这个著名的树篱迷宫以及大多数迷宫题中，都没有这种标志。但是，后面将会看到，它们在回溯算法中是的确存在的。

回溯对于某些问题是非常有用的，比如 0-1 背包问题。尽管在 4.5.3 节中为这一问题找到了一种动态规划算法，当背包的容量  $W$  不是很大时，它的效率还是很高的，但在最差情况下，这一算法仍然具有指数时间复杂度。0-1 背包问题属于第 9 章讨论的一类问题。从来没人能为其中任何一个问题找到一种算法，使其最差情况时间复杂度优于指数函数，但也从来没人能证明这些算法是不可能存在的。一种尝试解决 0-1 背包问题的方法是实际生成所有子集，但这就像是尝试迷宫中的每个路径，直到碰到死胡同为止。回想一下 4.5.1 节的内容，共有  $2^n$  个子集，这意味着暴力方法只有在  $n$  值很小时才是可行的。但是，如果在生成子集时能够找到一些标志，告诉我们其中有许多子集是不需要生成的，那通常会避免大量不必要的工作。这就是回溯算法要做的事情。对于诸如 0-1 背包问题之类的问题，其回溯算法在最差情况下仍然是指数时间复杂度（甚至还会更差）。它们之所以有用，是因为它们对许多大型实例都很高效，而不是因为它们对所有大型实例都很高效。5.7 节将再回来讨论 0-1 背包问题。在此之前，5.1 节将以一个简单的例子来介绍回溯，在其他几节中解决几个其他问题。

## 5.1 回溯方法

回溯方法用于解决这样一类问题：从一个指定集合中选择一个对象序列，使该序列满足某一标准 (criterion)。回溯方法的经典用例是在  $n$  皇后问题中。这种问题的目的是将  $n$  个皇后放在一个  $n \times n$  的棋盘上，使任意两个皇后都不会相互威胁到对方；也就是说，任意两个皇后都不会在同一行、同一列或同一对角线上。在这个问题中，前面所说的序列就是这些皇后的  $n$  个放置位置，可供每次选择的集合就是棋盘上  $n^2$  个可能位置，标准就是任何两个皇后都不会相互威胁到对方。 $n$  皇后问题的一个特例是当  $n=8$  时，这个实例使用的是标准棋盘。为简单起见，我们使用  $n=4$  时的实例来说明回溯方法。

回溯是一种经过修改的深度优先查找方法，但它是在树中查找（这里的“树”是指有根树）。所以在继续讨论之前，先来回顾一下深度优先查找。尽管深度优先查找一般是为图定义的，但我们将仅讨论树的查找，因为回溯仅涉及树查找。先序 (preorder) 树遍历是树的一种深度优先查找 (depth-first search)。也就是说，先查看根节点，查看一个节点后，立即查看该节点的所有后代。尽管深度优先查找并不要求以任何特定顺序查看子节点，但在本章的应用中，将会从左向右查看一个节点的子节点。图 5-1 给出一种以这种方式对树进行的深度优先查找。图中节点根据它们的查看顺序编号。注意，在深度优先查找中，会尽可能深地沿一条路径下行，直到遇到死胡同为止。在死胡同处回退，直到遇到一个节点，它有一个我们尚未查看过的子节点，然后再次尽可能深地下行。

有一种简单的递归算法可以完成深度优先查找。因为我们现在只关心树的先序遍历，所以下面给出一个专门完成这一任务的版本。调用此过程时，将顶级根节点传送给它。

```

void depth_first_tree_search (node v)
{
    node u;
    visit v;
    for (v 的每个子节点 u)
        depth_first_tree_search(u);
}

```

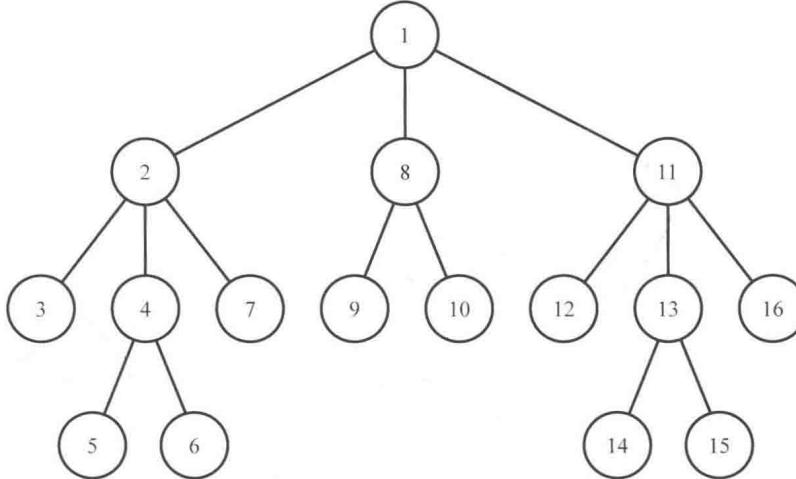


图 5-1 一棵树，其中的节点根据深度优先查找编号

这个通用算法没有要求必须以任意特定顺序访问子节点。但是，如前所述，我们是从左向右访问它们的。

现在用  $n=4$  时的  $n$  皇后问题实例来演示回溯方法。我们的任务是在一个  $4 \times 4$  棋盘上放置四个皇后，使任何两个皇后都不会相互威胁。意识到任何两个皇后不能放在同一行上，马上就可以简化问题。为解决这一实例，可以先将每个皇后安排到一个不同的行，然后检查哪些列组合方式可以给出答案。因为每个皇后可以都放在四列之一，所以共有  $4 \times 4 \times 4 \times 4 = 256$  种候选答案。

为了构造这些候选答案，我们可以构建一棵树，其中，可供第一个皇后（第 1 行中的皇后）选择的列存储在树中的第 1 级节点（别忘了，根节点在第 0 级），可供第二个皇后（第 2 行中的皇后）选择的列存储在第 2 级节点，以此类推。从根节点到一个叶节点的路径就是候选答案（回想一下，树中的叶节点（leaf）就是一个没有子节点的节点）。这棵树称为状态空间树（state space tree）。图 5-2 给出了它的一小部分。整个树共有 256 个叶节点，每个候选答案对应一种。注意，每个节点中存储了一个有序对  $\langle i, j \rangle$ 。这个有序对表示第  $i$  行的皇后放在第  $j$  列。

为确定这些答案，从最左侧的路径开始，依次检查每个候选答案（每条从根节点到一个叶节点的路径）。最先检查的少数几条路径如下：

```

[<1, 1>, <2, 1>, <3, 1>, <4, 1>]
[<1, 1>, <2, 1>, <3, 1>, <4, 2>]
[<1, 1>, <2, 1>, <3, 1>, <4, 3>]
[<1, 1>, <2, 1>, <3, 1>, <4, 4>]
[<1, 1>, <2, 1>, <3, 2>, <4, 1>]

```

注意，这些节点是依据一种深度优先查找顺序进行访问的，根据这种顺序，从左向右查看一个节点的子节点。对一棵状态空间树执行的简单深度优先查找类似于沿着迷宫中的每条路径前进，直到碰上死胡同为止。它并没有利用沿途的任何标志。我们可以通过查看这些标志来提高查找效率。例如，如图 5-3a 所示，任何两个皇后都不能在同一列中。因此，对于从图 5-2 中  $\langle 2, 1 \rangle$  所在节点发出的整个分支，构造和查看其中的任何路径都没有意义（因为我们已经将第 1 个皇后放在了第 1 列，所以不可能再在那里放置第 2 个皇后了）。这个标志告诉我们，这个节点只会引向死胡同。同理，如图 5-3b 所示，任何两个皇后都不能在同一对角线上，因此，对

于从图 5-2 中 $<2, 2>$ 所在节点发生的整个分支，都没有任何构造和查看的意义。

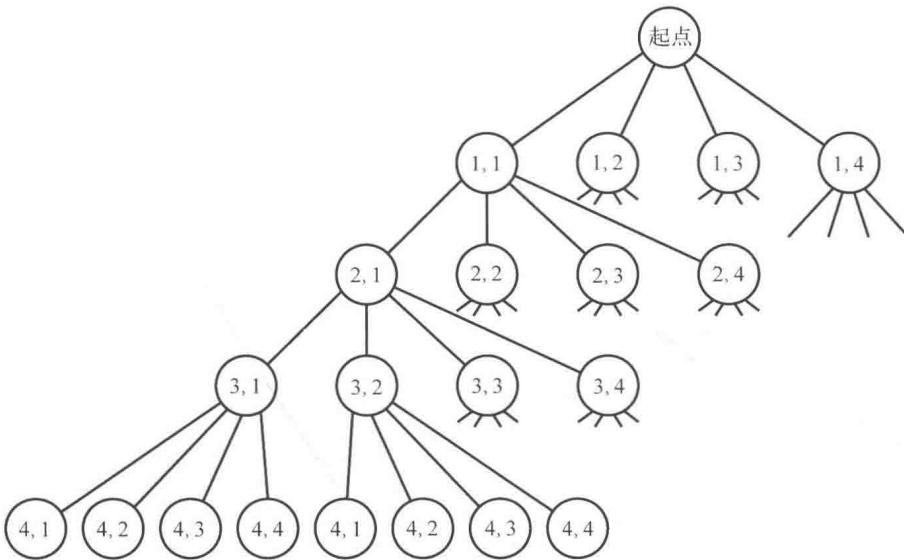


图 5-2  $n$  皇后问题实例 ( $n=4$ ) 的部分状态空间树。每个节点处的有序对 $\langle i, j \rangle$ 表示第  $i$  行的皇后放在第  $j$  列。每条从根节点到一个叶节点的路径都是一个候选答案

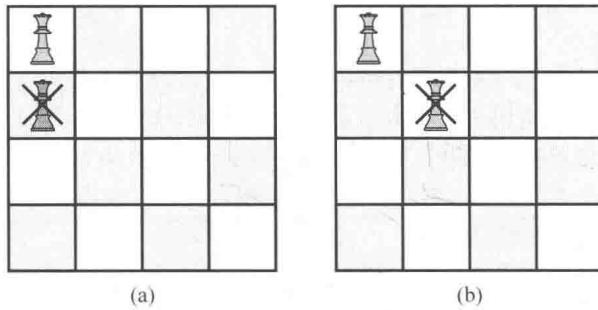


图 5-3 图形显示，如果第一个皇后被放在第 1 列，第二个皇后就不能被放在第 1 列(a)或第 2 列(b)

回溯 (backtracking) 过程是指，在确定一个节点只会引向死胡同时，我们回退（“回溯”）到该节点的父节点，在下一个子节点上继续查找。如果在查找一个节点时发现，它不可能给出答案，那就说它是没希望的 (nonpromising)。否则，就说它是有希望的 (promising)。总结一下，回溯过程包括：对一个状态空间树进行深度优先查找，检查每个节点有没有希望，如果没有希望，则回溯到该节点的父节点。这称为状态空间树的修剪 (pruning)，由已访问节点组成的子树称为修剪后的状态空间树。回溯方法的一般算法如下：

```
void checknode (node v)
{
    node u;

    if (promising (v))
        if (在 v 处有一个答案)
            写出该答案;
        else
            for (v 中的每个子节点 u)
                checknode(u);
}
```

状态空间树最高级别的根节点被传送给 `checknode`。查看一个根节点就是检查它有没有希望。如果有希望且该节点处有一个答案，就输出该答案。如果在一个有希望的节点处没有答案，就访问该节点的子节点。函数

promising 在回溯方法的每个应用中都是不一样的。我们称之为算法的 promising 函数。回溯算法基本上与深度优先查找相同，区别就是它仅在一个节点有希望，而且该节点处没有答案时才会访问它的子节点。（某些回溯算法不同于  $n$  皇后问题的算法，可能在到达状态空间树的叶节点之前就找到了答案。）我们将上述回溯过程称为 checknode 而不是 backtrack，是因为在调用此过程时还没有发生回溯，而是在发现一个节点没有希望并查找其父节点的另一个子节点时，才会回溯。在递归算法的计算机实现中，是将一个无希望节点的活动记录从活动记录栈中弹出，以此来完成回溯。

下面用回溯方法解决  $n=4$  时的  $n$  皇后问题实例。

**例 5.1** 回想一下，函数 promising 对于回溯方法的每个应用都是不同的。对于  $n$  皇后问题，当一个节点及该节点的任意祖先将皇后放在同一列或同一对角线时，该函数都必须返回 false。图 5-4 给出了在用回溯方法解决  $n=4$  的实例时，所生成的一部分修剪后的状态空间树。图中只给出了为找出第一个答案而查看的节点。图 5-5 显示了实际棋盘。在图 5-4 中，如果一个节点没有希望，则为其标记一个“ $\times$ ”。同样，图 5-5 中的无希望位置也有一个叉。图 5-4 中带有阴影的节点就是找到第一个解的地方。回溯过程走过的一次往返移动如下。我们用节点中存储的有序对来指称该节点。一些节点中包含的有序对相同，但在遍历图 5-4 中的树时，可以判断出我们说的是哪个节点。

(a)  $<1, 1>$  是有希望的。(因为第 1 个皇后是最早被放置的。)

(b)  $<2, 1>$  是没希望的。(因为第 1 个皇后在第 1 列。)

$<2, 2>$  是没希望的。(因为第 1 个皇后在左对角线。)

$<2, 3>$  是有希望的。

(c)  $<3, 1>$  是没希望的。(因为第 1 个皇后在第 1 列。)

$<3, 2>$  是没希望的。(因为第 2 个皇后在右对角线。)

$<3, 3>$  是没希望的。(因为第 2 个皇后在第 3 列。)

$<3, 4>$  是没希望的。(因为第 2 个皇后在左对角线。)

(d) 回退到  $<1, 1>$ 。

$<2, 4>$  是有希望的。

(e)  $<3, 1>$  是没希望的。(因为第 1 个皇后在第 1 列。)

$<3, 2>$  是有希望的。(这是第二次尝试  $<3, 2>$ 。)

(f)  $<4, 1>$  是没希望的。(因为第 1 个皇后在第 1 列。)

$<4, 2>$  是没希望的。(因为第 3 个皇后在第 2 列。)

$<4, 3>$  是没希望的。(因为第 3 个皇后在左对角线。)

$<4, 4>$  是没希望的。(因为第 2 个皇后在第 4 列。)

(g) 回溯到  $<2, 4>$

$<3, 3>$  是没希望的。(因为第 2 个皇后在右对角线。)

$<3, 4>$  是没希望的。(因为第 2 个皇后在第 4 列。)

(h) 回溯到根节点。

$<1, 2>$  是有希望的。

(i)  $<2, 1>$  是没希望的。(因为第 1 个皇后在右对角线。)

$<2, 2>$  是没希望的。(因为第 1 个皇后在第 2 列。)

$<2, 3>$  是没希望的。(因为第 1 个皇后在左对角线。)

$<2, 4>$  是有希望的。

(j)  $<3, 1>$  是有希望的。(这是我们第三次尝试  $<3, 1>$ 。)

(k)  $<4, 1>$  是没希望的。(因为第 3 个皇后在第 1 列。)

$<4, 2>$  是没希望的。(因为第 1 个皇后在第 2 列。)

$<4, 3>$  是有希望的。

这时找到了第一个解。它出现在图 5-5k 中，在图 5-4 中，在找到该解的节点处添加了阴影。

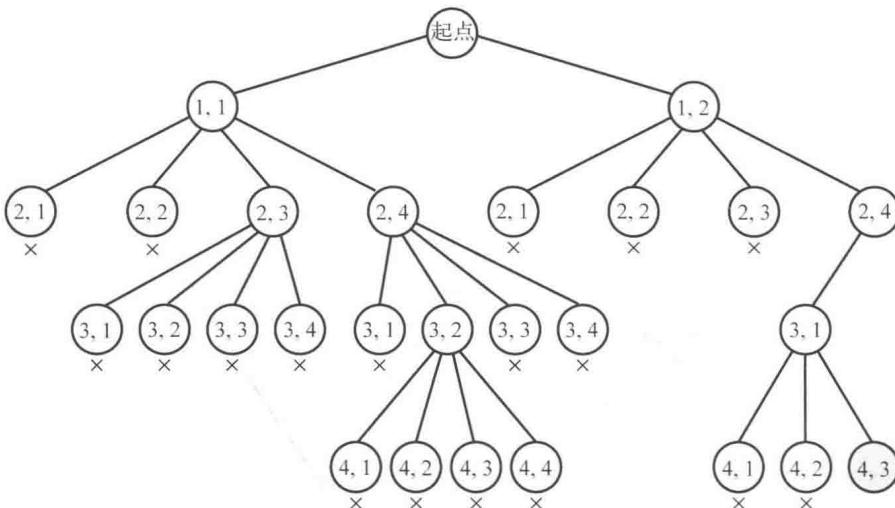


图 5-4 在用回溯方法解决  $n$  皇后问题的  $n=4$  实例时，所生成的一部分修剪后的状态空间树。图中只给出了寻找第一个解时查看的节点。这个解是在带有阴影的节点处找到的。每个无希望节点都标有“ $\times$ ”

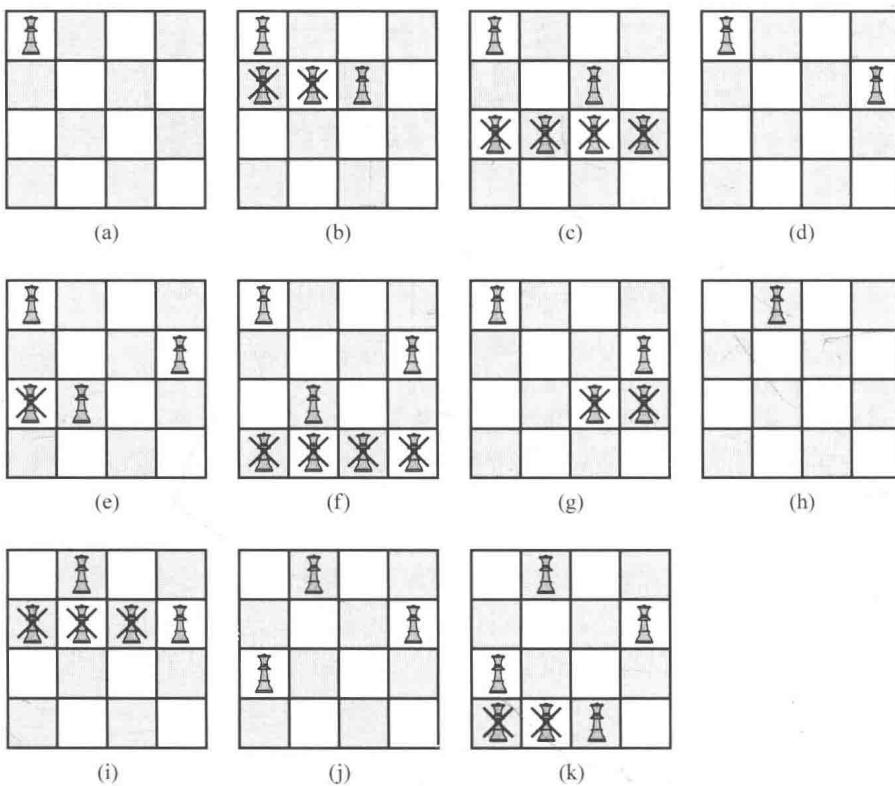


图 5-5 在使用回溯法解决  $n=4$  的  $n$  皇后问题实例时，尝试的实际模型位置。每个无希望位置都标有一个叉

注意，回溯算法不需要实际创建树。只需要跟踪当前研究分支中的值。我们就是这样实现回溯算法的。我们说，状态空间树在这个算法中是隐式存在的，因为并没有实际构建它。

图 5-4 中的节点数表明，此回溯算法在找到答案之前检查了 27 个节点。在习题中将要证明，没有回溯的

话，状态空间树的深度优先查找要检查 155 个节点才能找到同一个解。

读者可能已经观察到，在回溯的一般算法（过程 `checknode`）中存在一些低效特性。也就是说，先将一个节点传送给该过程之后才查看它是否有希望。这意味着无希望节点的活动记录也会被放到活动记录栈中，这是不必要的。在传送一个节点之前先检查它是否有希望，就可以避免上述情况。进行上述操作的一般回溯算法如下：

```
void expand(node v)
{
    node u;

    for (v 中的每个子节点 u)
        if (promising(u))
            if (在 u 处有一个答案)
                写出该答案;
            else
                expand(u);
}
```

在顶级传送给该过程的节点仍然是树的根节点。我们将这一过程称为 `expand` 是因为展开了一个有希望的节点。这一算法的计算机实现在从一个无希望节点回溯时，不会将该节点的活动记录压到活动记录栈中。

本章在解释算法时，首先使用算法的第一个版本（过程 `checknode`），因为我们发现这一版本通常可以给出易于理解的算法。原因在于，`checknode` 的一次执行包含了在访问单个节点时完成的步骤。也就是说，它包括以下步骤：判断该节点是否有希望；如果有希望，且该节点处有一个答案，就输出该答案，否则访问其子节点。而 `expand` 的一次执行会对一个节点的所有子节点执行相同步骤。在查看算法的第一个版本后，不难写出其第二个版本。

下面为几个问题开发回溯算法，首先从  $n$  皇后问题开始。在所有这些问题中，状态空间树中的节点都是指指数级的，甚至还要大。回溯方法用于避免对节点进行不必要的检查。给定两个具有相同  $n$  值的实例，一种回溯算法对于其中一个实例可能只需检查非常少的几个节点，而对另一个实例却需要查看整个状态空间树。这就是说，我们并没有像前面各章的算法那样，为回溯算法获得高效的时间复杂度。因此，本章进行的分析不同于前几章，而是使用蒙特卡洛方法来分析回溯算法。这一方法可以用来判断，是否可以预期一种给定回溯算法能够高效地解决某个特定实例。5.3 节讨论蒙特卡洛方法。

## 5.2 $n$ 皇后问题

我们已经讨论了  $n$  皇后问题的目标。`promising` 函数必须检查两个皇后是否在同一列或同一对角线中。如果令  $\text{col}(i)$  表示第  $i$  行皇后所在的列，那么，为了检查第  $k$  行中的皇后是否在同一列，需要检查下式是否成立：

$$\text{col}(i) = \text{col}(k)$$

接下来看看如何检查对角线。图 5-6 演示了  $n=8$  的实例。在这个图中，第 6 行的皇后在其左对角线上受到第 3 行皇后的威胁，在其右对角线受到第 2 行皇后的威胁。注意：

$$\text{col}(6) - \text{col}(3) = 4 - 1 = 3 = 6 - 3$$

也就是说，对于从左侧威胁的皇后，列数之差与行数之差相同。此外，

$$\text{col}(6) - \text{col}(2) = 4 - 8 = -4 = 2 - 6$$

也就是说，对于从右侧威胁的皇后，列数之差等于行数之差的相反数。这是下面这个一般性结果的例子：如果第  $k$  行中的皇后沿其一个对角线威胁第  $i$  行的皇后，则

$$\text{col}(i) - \text{col}(k) = i - k \text{ 或 } \text{col}(i) - \text{col}(k) = k - i$$

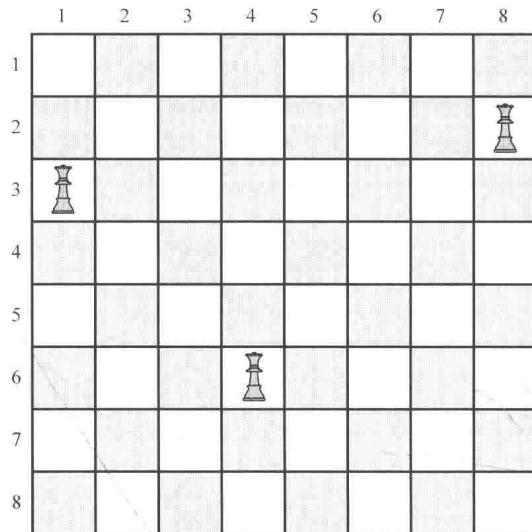


图 5-6 第 6 行的皇后在其左对角线受到第 3 行皇后的威胁，在其右对角线受到第 2 行皇后的威胁

下面给出算法。

### 算法 5.1 $n$ 皇后问题的回溯算法

问题：将  $n$  个皇后放在一个棋盘上，使任何两个都不在同一行、同一列或同一对角线上。

输入：正整数  $n$ 。

输出：将  $n$  个皇后放在  $n \times n$  棋盘上，使任意两个皇后都不会相互威胁的所有可能方式。每个输出包括一个整数数组  $\text{col}$ ，其索引范围为 1 至  $n$ ，其中  $\text{col}[i]$  是第  $i$  行皇后所在的列。

```

void queens (index i)
{
    index j;

    if (promising(i))
        if (i == n)
            cout << col[1] 至 col[n];
        else
            for (j = 1; j <= n; j++) {           // 查看第(i+1)行的皇后是否可以放在 n 列中的第一列中。
                col[i+1]=j;
                queens(i+1);
            }
    }

bool promising (index i)
{
    index k;
    bool switch;

    k=1;
    switch = true;                         // 检查是否有任何皇后威胁第 i 行的皇后。
    while (k < i && switch) {
        if (col[i]==col[k] || abs(col[i]-col[k]==i-k)
            switch = false;
        k++;
    }
    return switch;
}

```

当一个算法中包含多个例程时，我们并没有根据任何特定程序设计语言的规则对例程排序，而是首先给出主例程。在算法 5.1 中，主例程为 `queens`。根据 2.1 节讨论的约定， $n$  和  $\text{col}$  不是递归例程 `queens` 的输入。如果在实现该算法时将  $n$  和  $\text{col}$  定义为全局变量，则对 `queens` 的顶级调用为：

```
queens(0);
```

算法 5.1 给出了 *n* 皇后问题的所有解，因为我们就是这样表述问题的。根据我们对问题的表述方式，不需要在找到一个循环时退出实现。这样可以降低算法的混乱程度。一般情况下，本章中的问题可表述为需要一个、多个或全部解。在实践中，具体需要几个解，取决于应用需要。我们的大多数算法都编写为生成所有解。只需稍作修改就可以使这些算法在找到一个解后停止。

算法 5.1 的理论分析非常困难。为此，必须将要检查的节点数表示为皇后数目 *n* 的函数。通过计算整个状态空间树中的节点数，可以获得已修剪状态空间树中的节点数上限。在完整的状态空间树中，第 0 级包含 1 个节点，第 1 级包含 *n* 个节点，第 2 级包含  $n^2$  个节点……第 *n* 级包含  $n^n$  个节点。总节点数为：

$$1+n+n^2+n^3+\cdots+n^n = \frac{n^{n+1}-1}{n-1}$$

这个等式在附录 A 的例 A.4 中得出。对于 *n*=8 的实例，状态空间树中包含的节点数为：

$$\frac{8^{8+1}-1}{8-1} = 19\,173\,961$$

这一分析的价值非常有限，因为回溯的整个目的就是避免检查其中大量节点。

我们可以尝试的另一分析是获得有希望节点的上限个数。为计算这一界限，可以利用如下事实：任何两个皇后都不能放在同一列。例如，考虑 *n*=8 的实例。第一个皇后可以放在 8 列中的任何一列。第一个皇后放定之后，第二个皇后最多只能放在 7 列中的一列；第二个放定之后，第三个皇后最多只能放在 6 列中的一列；以此类推。因此，最多有

$$1+8+8\times7+8\times7\times6+8\times7\times6\times5+\cdots+8!=109\,601 \text{ 个有希望节点。}$$

将这一结果推广到任意 *n* 值，则最多有

$$1+n+n(n-1)+n(n-1)(n-2)+\cdots+n! \text{ 个有希望节点。}$$

这一分析并没有让我们很好地了解该算法的效率，原因如下：第一，它没有考虑函数 *promising* 的对角线检查。因此，真正的有希望节点可能远小于这一上限；第二，所检查的总节点数同时包含了有希望的节点和没希望的节点。后面将会看到，无希望节点的数目可能远大于有希望节点的数目。

确定该算法效率的一种直接方式是在一台计算机上实际运行该算法，并计算它检查了多少个节点。表 5-1 给出了 *n* 取几个不同值时的结果，并将回溯算法与 *n* 皇后问题的其他两种算法进行了对比。算法 1 是对状态空间树进行的一种无回溯深度优先查找。它检查的节点数就是状态空间树中的数目。算法 2 仅利用了“任何两个皇后都不能在同一行或同一列”的事实。算法 2 生成了 *n*! 种候选解：尝试将第 row-1 个皇后放到 *n* 列中的每一列，将第 row-2 个皇后放在 *n*-1 个未被第一个皇后占用的列中，将第 row-3 个皇后放在未被前两个皇后占用的 *n*-2 列中，以此类推。在生成一个候选解后，检查两个皇后是否在一个对角线上相互威胁。注意，回溯算法的优势会随着 *n* 的增大而急剧增大。当 *n*=4 时，算法 1 检查的节点数不到回溯算法的 6 倍，回溯算法似乎稍劣于算法 2。但当 *n*=14 时，算法 1 检查的节点数几乎是回溯算法的 3200 万倍，而算法 2 生成的候选解数大约是回溯算法所检查节点数的 230 倍。表 5-1 中列出了有希望的节点数，表明在所检查的节点中，有许多都是无希望的。这意味着，我们对回溯方法的第二种实现方式（5.1 节讨论的过程 *expand*）可以节省大量时间。

表 5-1 关于在 *n* 皇后问题中借助回溯可以节省多少次检查的说明\*

<i>n</i>	算法 1 检查的节点数 <sup>†</sup>	算法 2 检查的节点数 <sup>‡</sup>	回溯检查的节点数	回溯发现的有希望节点数
4	341	24	61	17
8	19 173 961	40 320	15 721	2057
12	$9.73 \times 10^{12}$	$4.79 \times 10^8$	$1.01 \times 10^7$	$8.56 \times 10^5$
14	$1.20 \times 10^{16}$	$8.72 \times 10^{10}$	$3.78 \times 10^8$	$2.74 \times 10^7$

\*项目表示为找出所有解而需要的检查数。

† 算法 1 对状态空间树执行无回溯深度优先查找。

‡ 算法 2 生成 *n*! 种候选答案，将每个皇后放在不同行列。

通过实际运行一个算法来确定其效率（就像为创建表 5-1 所做的那样），并不是真正的分析。我们这样做，只是为了说明回溯可以节省多少时间。分析的目的应当是事先判断一个算法是否高效。下一节将说明如何利用蒙特卡洛方法来估计一个回溯算法的效率。

回想一下，在  $n$  皇后问题的状态空间树中，我们利用了“任何两个皇后都不能在同一行”的事实。或者，我们可以创建一个状态空间树，在  $n^2$  个棋盘位置的每个位置中尝试放置每个皇后。只要一个皇后与之前放置的皇后位于同一行、同一列或同一对角线，就会在此树中回溯。这个状态空间树中的每个节点会有  $n^2$  个子节点，每个棋盘位置各有一个。树中会有  $(n^2)^n$  个叶节点，分别表示一个不同的候选答案。对于在状态空间树中进行回溯的算法，其所找到的有希望节点数不会超过我们的算法，但它仍然要比我们的算法更慢一些，这是因为在函数 `promising` 中检查行需要额外的时间，而且会检查更多的无希望节点（这些节点尝试将皇后放在已被占用的行中）。一般情况下，最高效的做法是在状态空间树中放入尽可能多的信息。

在许多回溯算法中都需要考虑在 `promising` 函数中的花费的时间。也就是说，我们的目的并非一定要削减要检查的节点数，而是提高整体效率。一个耗费大量时间的 `promising` 函数会抵消检查较少节点带来的好处。在算法 5.1 中，可以通过多种方式来改进 `promising` 函数：跟踪由已放置皇后控制的列集合、左对角线集合和右对角线集合。这样，就不用检查已放置的皇后是否会威胁当前皇后，只需要检查当前皇后是否被放在一个受控制的列或对角线中。这一改进将在习题中研究。

### 5.3 用蒙特卡洛算法估计回溯算法的效率

前面曾经提到，在以下各节给出的每种算法的状态空间树中，其节点数都是指数级的，甚至更多。给定具有相同  $n$  值的两个实例，其中一个需要检查的节点非常少，而另一个可能需要检查整个状态空间树。如果能够估计一种给定回溯算法处理一个特定实例的效率，就能判断在该实例上应用该算法是否合理。利用蒙特卡洛算法可以获得这样一个估计。

蒙特卡洛算法是概率性算法。所谓概率性算法（probabilistic algorithm），是指下一条要执行的指令有时是根据某一概率分布随机确定的（如无特殊说明，假定该概率分布是均匀分布）。而确定性算法（deterministic algorithm）则是指不可能出现这种情况的算法。到目前为止，我们讨论的所有算法都是确定性算法。假定在一个样本空间上定义了一个随机变量，蒙特卡洛算法（Monte Carlo algorithm）会根据该随机变量在该样本空间一个随机样本上的平均值，来估计该随机变量的期望值。（关于样本空间、随机样本、随机变量和期望值的讨论，请参阅附录 A 的 A.8.1 节。）在这里并不能保证这个估计结果与真正的期望值非常接近，但当该算法的可用次数增加时，它们互相接近的概率值也会增大。

蒙特卡洛算法可用于估计一种回溯算法用于特定实例的效率，如下所示。我们在一棵树中生成“典型路径”（这个棵中包含了在给定实例中要检查的节点），然后由这条路径估计树中的节点数。它估计的是为了获得所有答案而可能检查的节点总数。也就是说，估计的是经过修剪的状态空间树的节点数。为适用此方法，算法必须满足以下条件：

- (1) 针对状态空间树中同一级别的所有节点必须使用同一 `promising` 函数；
- (2) 状态空间树中同一级别的节点必须拥有相同数目的子节点。

注意，算法 5.1 ( $n$  皇后问题的回溯算法) 满足这些条件。

蒙特卡洛方法要求根据均匀分布生成一个节点的有希望子节点。也就是说，使用随机过程来生成有希望的子节点。（关于随机过程的讨论，请参阅附录 A 的 A.8.1 节。）在计算机上实现该方法时，只能生成伪随机的有希望子节点。方法如下。

- 设  $m_0$  是根节点的有希望子节点数。
- 在第 1 级随机生成一个有希望的节点。设  $m_1$  是这一节点的有希望子节点数。
- 为上一步获得的节点随机生成一个有希望的子节点。设  $m_2$  是这一节点的有希望子节点数。

⋮

□ 为上一步获得的节点随机生成一个有希望的子节点。设  $m_i$  是这一节点的有希望子节点数。

⋮

一直持续这一过程，直到找不出有希望的子节点为止。因为我们假定状态空间树中相同级别的节点都具有相同的子节点数，所以  $m_i$  就是第  $i$  级节点的有希望子节点的平均数。设

$t_i =$  第  $i$  级一个节点的子节点总数

因为一个节点的所有  $t_i$  个子节点都会检查，而且只有  $m_i$  个有希望的子节点才拥有被检查的子节点，所以，回溯算法为找出所有答案而检查的总节点数为：

$$1 + t_0 + m_0 t_1 + m_0 m_1 t_2 + \cdots + m_0 m_1 \cdots m_{i-1} t_i + \cdots$$

计算这一估计值的一般算法如下。在此算法中，用变量  $mprod$  表示每个级别的乘积  $m_0 m_1 \cdots m_{i-1}$ 。

### 算法 5.2 蒙特卡洛估计

问题：使用蒙特卡洛算法估计回溯算法的效率。

输入：回溯算法所解决问题的一个实例。

输出：对该算法生成的已修剪状态空间树中节点数的估计值，也就是该算法为了找出该实现的所有解而检查的节点数。

```
int estimate()
{
    node v;
    int m, mprod, t, numnodes;

    v = 状态空间树的根;
    numnodes=1;
    m=1;
    mprod=1;
    while (m !=0){
        t = v 的子节点数;
        mprod = mprod * m;
        numnodes = numnode + mprod * t;
        m = v 的有希望子节点数;
        if (m != 0)
            v=v 的随机选定的有希望子节点;
    }
    return numnodes;
}
```

下面给出算法 5.2 的一个特定版本，用于处理算法 5.1 ( $n$  皇后问题的回溯算法)。我们向算法 5.2 传送  $n$ ，因为它是算法 5.1 的参数。

### 算法 5.3 算法 5.1 的蒙特卡洛估计 ( $n$ 皇后问题的回溯算法)

问题：估计算法 5.1 的效率。

输入：正整数  $n$ 。

输出：对算法 5.1 生成的已修剪状态空间树中节点数的估计值，也就是该算法要检查这么多节点之后，才会找出在一个  $n \times n$  棋盘上放置  $n$  个皇后，而且使任何两个皇后不会相互威胁的所有方法。

```
int estimate_n_quenns (int n)
{
    index i, j, col[1..n];
    int m, mprod, numnodes;
    set_of_index prom_children;

    i = 0;
    numnodes = 1;
```

```

m = 1;
mprod = 1;
while (m != 0 && i != n){
    mprod = mprod * m;
    numnodes = numnodes + mprod * n;      // 子节点数为 n
    i++;
    m = 0;
    prom_children = ∅;                    // 将有希望子节点的集合初始化为空集
    for (j = 1; j <= n; j++){
        col[i]=j;
        if (promising(j)){               // 确定有希望的子节点。函数 promising 是
            m++;                         // 算法 5.1 中的一个。
            prom_children = prom_children ∪ {j};
        }
    }
    if (m != 0){                        // 从 prom_children 中随机选择;
        j = 从 prom_children 中随机选择;
        col[i]=j;
    }
}
return numnodes;
}

```

在使用蒙特卡洛算法时，应当进行多次估计，并以结果的平均值作为实际估计值。利用统计学的标准方法，可以根据实验结果为实际检查的节点数确定一个置信区间。作为一条经验规则，大约 20 次实验通常就足够了。这里要提醒，尽管在将蒙特卡洛算法运行多次之后，获得良好估计值的概率会很高，但从来不能保证它一定是一个好的估计值。

$n$  皇后问题对于每个  $n$  值只有一个实例。对于大多数用回溯算法解决的问题来说，都不是这种情况。任何一次运用蒙特卡洛方法生成的估计值都是针对一种特定实例的。前文曾经讨论过，给定两个具有相同  $n$  值的实例，一种可能只需要检查非常少的几个节点，而另一种可能需要检查整个状态空间树。

使用蒙特卡洛估计方法得到的估计值不一定能很好地表明要检查多少个节点才会找到第一个解。如果只需要获得一个解，该算法检查的节点数可能只是找出全部解时的一小部分。例如，图 5-4 表明，如果只是为了寻找一个解，分别将第一个皇后放在第三列和第四列的两个分支都不需要遍历。

## 5.4 “子集之和” 问题

回想 4.5.1 节的小偷与 0-1 背包问题。在这个问题中，小偷可以偷走一组物品（一个集合），每件物品有自己的重量和价值。如果背包中的物品总重量超过了  $W$ ，背包就会破损。因此，目标是在总重量不超过  $W$  的情况下，使所偷物品的总价值最高。这里我们假定所有物品拥有相同的单位重量价值。那么，小偷的最优解就是一组物品（一个集合），在总重量不超过  $W$  的约束条件下，使总价值最大化。这个小偷可能会首先尝试判断，是否有一组物品的总重量恰好等于  $W$ ，因为这是最佳情况。判断这种集合的问题称为子集之和（sum-of-subsets）问题。

具体来说，在“子集之和”问题中，有  $n$  个正整数（重量） $w_i$  和一个正整数  $W$ ，目标是找出这些整数中所有总和等于  $W$  的子集。如前所述，我们对问题的表述通常是要求出所有解，但对于这个小偷的应用，只需要找出一种解。

**例 5.2** 假定  $n=5$ ,  $W=21$ , 且

$$w_1=5 \quad w_2=6 \quad w_3=10 \quad w_4=11 \quad w_5=16$$

因为

$$\begin{aligned} w_1+w_2+w_3 &= 5+6+10=21 \\ w_1+w_5 &= 5+16=21 \end{aligned}$$

$$w_3 + w_4 = 10 + 11 = 21$$

所以解为：

$$\{w_1, w_2, w_3\}, \quad \{w_1, w_5\}, \quad \{w_3, w_4\}$$

这个实例可通过观察来解决。对于较大的  $n$  值，则需要一种系统方法。一种方法是创建一个状态空间树。图 5-7 中给出了一种构建这种树的可能方法。为简单起见，图中的这个树只是针对三种重量的。从根节点向左移动将包含  $w_1$ ，向右移动将排除  $w_1$ 。同理，在第 1 级的一个节点向左移动，将包含  $w_2$ ，向右移动将排除  $w_2$ ，等等。每个子集都由根节点到一个叶节点的路径表示。当包含  $w_i$  时，就将  $w_i$  写在包含它的边上；当没有包含  $w_i$  时，就记下 0。

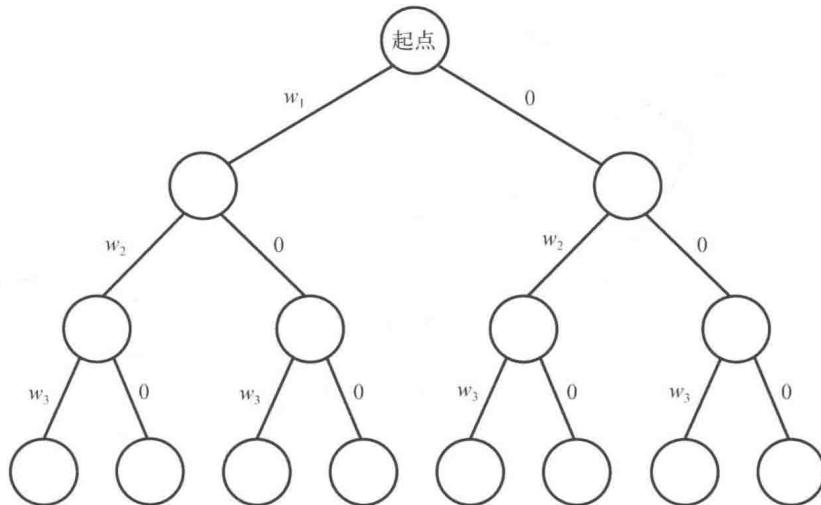


图 5-7 “子集之和”问题的一棵状态空间树，其中  $n=3$

例 5.3 在图 5-8 给出的状态空间树中,  $n=3$ ,  $W=6$ , 且

$$w_1=2 \quad w_2=4 \quad w_3=5$$

在每个节点，都会将已经包含在内的重量之和写在该点。因此，每个叶节点处都包含了引向该叶节点的子集中的重量之和。左起第二个叶节点是唯一包含 6 的，到达这一叶节点的路径表示子集  $\{w_1, w_2\}$ ，所以这个子集是唯一解。

如果在查找之前，先按照非递减顺序对重量排序，那就有一个明显的标志，告诉我们一个节点是没希望的。如果按照这种方式对重量排序，那当我们处于第  $i$  级时， $w_{i+1}$  就是最轻重量。设 weight 是已经被包含在第  $i$  行一个节点处的重量之和。如果  $w_{i+1}$  使 weight 值大于  $W$ ，那它后面的所有其他重量也会如此。因此，除非 weight 等于  $W$ （这意味着在该节点处存在一个解），否则，当

**weight+ $w_{i+1}$ >W**

时，位于第  $i$  级的一个节点是没有希望的。

还有另外一种不太明显的标记，可以告诉我们一个节点是没有希望的。如果在一个给定节点，将剩余物品的全部重量加到 weight，都不会使 weight 至少等于  $W$ ，那在该节点处再行展开，也绝对不会使 weight 等于  $W$ 。这就意味着，如果 total 是剩余物品的总重量，当

weight+total <  $W$

时，一个节点是没希望的。

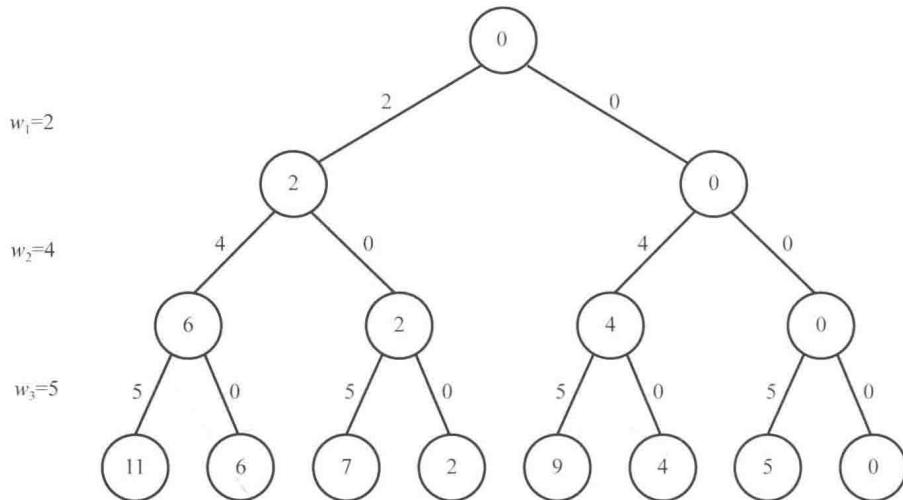


图 5-8 例 5.3 所示“子集之和”问题实例的状态空间树。存储在每个节点中的内容是到该节点为止所包含的总重量

下面的例子演示了这些回溯策略。

例 5.4 图 5-9 给出了在以下条件下使用回溯方法时的已修剪状态空间树:  $n=4$ ,  $W=13$ , 且

$$w_1=3 \quad w_2=4 \quad w_3=5 \quad w_4=6$$

唯一解是在带有阴影的节点处找到的。这个解是  $\{w_1, w_2, w_4\}$ 。没有希望的节点标有“ $\times$ ”。包含 12、8、9 的节点是没有希望的，因为加上下一个重量(6)后，会使 weight 超过  $W$ 。包含 7, 3, 4, 0 的节点是没有希望的，因为剩下的总重量不足以使 weight 值达到  $W$ 。注意，状态空间树中不包含解的那些叶节点可自动判断为没希望的，因为没有剩余重量可以使 weight 达到  $W$ 。包含 7 的叶节点说明了这一点。在这个已修剪状态空间树中只有 15 个节点，而整个状态空间树中则包含 31 个节点。

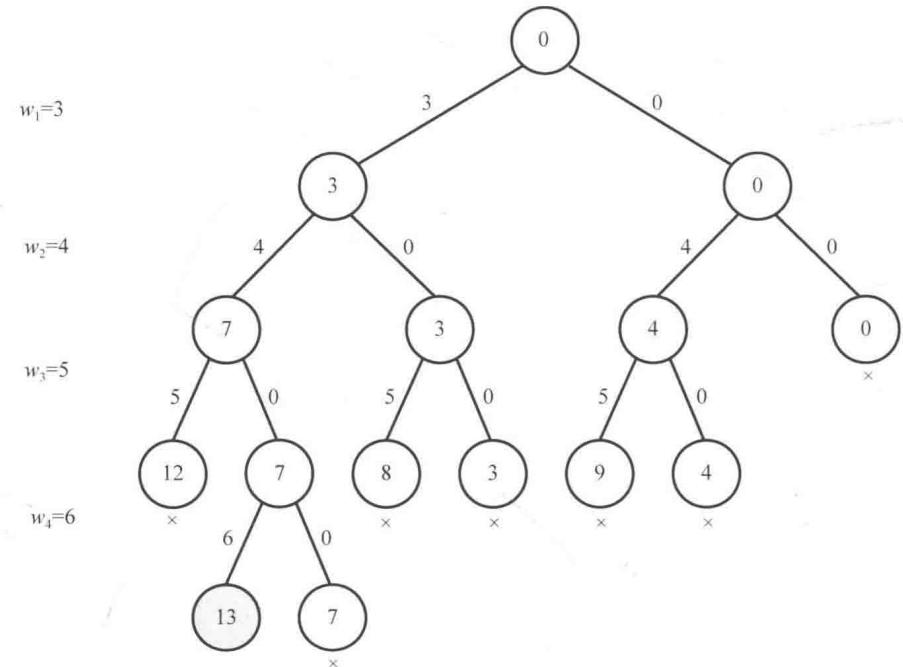


图 5-9 例 5.4 中使用回溯法生成的已修剪状态空间树。每个节点中存储的是到达该节点处所包含的总重量。在阴影节点中找到了唯一解。每个无希望节点都标有“ $\times$ ”

当到一节点所包含的重量之和等于  $W$  时，该节点处存在一个解。因此，我们不能再通过包含更多物品来获得另一个解。这就是说，如果

$$W = \text{weight}$$

就应输出该解，并回溯。这一回溯是由我们的通用过程 `checknode` 自动提供的，因为它从来不会展开一个已经找到解的有希望节点。回想一下，在讨论 `checknode` 时曾经提到，一些回溯算法有时会在到达状态空间树的叶节点之前找到一个解。上面就是这样一种算法。

接下来给出一种采用这些策略的算法。该算法使用数组 `include`。若包含了  $w[i]$ ，则将 `include[i]` 设定为 `yes`，否则，设定为 `no`。

#### 算法 5.4 “子集之和”问题的回溯算法

问题：给定  $n$  个正整数（重量）和一个正整数  $W$ ，确定这些整数中所有总和为  $W$  的组合。

输入：正整数  $n$ ；正整数的有序数组（非递减顺序） $w$ ，其索引范围为 1 至  $n$ ；正整数  $W$ 。

输出：这些整数中所有总和为  $W$  的组合。

```
void sum_of_subsets(index i,
                     int weight, int total)
{
    if (promising(i))
        if (weight == W)
            cout << include[1] 至 include[i];
        else {
            include[i+1] = "yes";           // 包含 w[i+1]
            sum_of_subsets(i+1, weight+w[i+1], total-w[i+1]);
            include[i+1] = "no";           // 不包含 w[i+1]
            sum_of_subsets(i+1, weight, total-w[i+1]);
        }
    }

bool promising (index i);
{
    return (weight+total >= W) && (weight == W || weight+w[i+1] <= W);
}
```

根据我们的一贯约定， $n$ 、 $w$ 、 $W$  和 `include` 不是例程的输入。如果这些变量定义为全局变量，则对 `sum_of_subsets` 的顶级调用如下：

```
sum_of_subsets(0, 0, total);
```

最初，

$$\text{total} = \sum_{j=1}^n w[j]$$

回想一下，状态空间树中不包含解的叶节点是没有希望的，因为没有留下可以使 `weight` 值等于  $W$  的重量。这意味着该算法应当不需要检查终止条件  $i=n$ 。现在验证该算法正确地实现了这一点。当  $i=n$  时，`total` 值为 0（因为没有剩余重量）。因此，此时

$$\text{weight} + \text{total} = \text{weight} + 0 = \text{weight}$$

这意味着，仅当  $\text{weight} \geq W$  时，

$$\text{weight} + \text{total} \geq W$$

为真。因为我们总是保持  $\text{weight} \leq W$ ，所以必须使  $\text{weight} = W$ 。因此，如果  $i=n$ ，则仅当  $\text{weight} = W$  时，函数 `promising` 返回 `true`。但在这种情况下，不存在递归调用，因为我们找到了一个解。因此，不需要检查终止条件  $i=n$ 。注意，在函数 `promising` 中从来不会引用不存在的数组项  $w[n+1]$ ，因为我们假定，在一个 `or` 表达式中，当第一个条件为真时，不再评估第二个条件。

算法 5.4 在状态空间树中查找的节点数等于

$$1 + 2 + 2^2 + \dots + 2^n = 2^{n+1} - 1$$

这一等式在附录A的例A.3中获得。在仅给出这一结果时存在一种可能性：最差情况可能远优于这一数值。也就是说，对于每个实例，都有可能仅查找很少一部分状态空间树。事实并非如此。对于每个n值，都可能构造出一种实例，使该算法访问的节点数达到指数级别。即使我们只想要一个解时，这一点也是成立的。为此，如果取

$$\sum_{i=1}^{n-1} w_i < W \quad (w_n = W)$$

则仅有一个解 $\{W_n\}$ ，但必须查看指数个节点后才能找到它。前面曾经强调过，尽管最差情况是指数级别的，但该算法对于许多大型实例的效率仍然可能很高。习题中会要求你使用蒙特卡洛方法编写程序，以估计算法5.4针对各种实例的效率。即使我们在表述问题时仅要求一个解，但诸如0-1背包问题之类的“子集之和”问题也属于第9章讨论的一类问题。

## 5.5 图的着色

“m着色问题”关心的是如何找出所有符合以下条件的着色方式：最多使用m种颜色为一个无向图着色，使任意两个相邻顶点的颜色都不相同。我们通常针对每个m值，为m着色问题起一个具体的名字。

**例5.5** 考虑图5-10中的图。2着色问题对此图无解，这是因为，如果最多能使用两种颜色，那就无法找出一种使任意两相邻顶点都不同色的着色方式。对于本图的3着色问题，它的一个解是：

顶点	颜色
$v_1$	颜色1
$v_2$	颜色2
$v_3$	颜色3
$v_4$	颜色2

针对此图的3着色问题，总共有6个解。但是，这六种解的唯一不同就是这些颜色的排列方式。例如，另一个解是为 $v_1$ 着颜色2，为 $v_2$ 和 $v_4$ 着颜色1，为 $v_3$ 着颜色3。

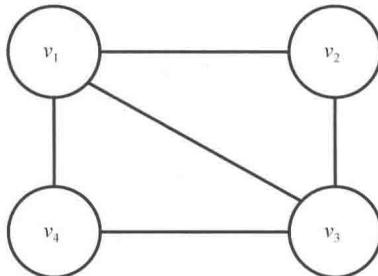


图5-10 针对此图的2着色问题无解。例5.5中给出了3着色问题针对此图的一个解

图着色的一种重要应用就是地图的着色。如果可以在一个平面上绘制一幅图，使任何两条边都不会交叉，则称该图是平面的（planar）。图5-11底部的图是平面的。但是，如果添加边 $(v_1, v_5)$ 和 $(v_2, v_4)$ ，它就不再是平面的。对于每幅地图，都有一个对应的平面图（planar graph）。地图中的每个区域用一个顶点表示。如果一个区域与另一区域相邻，则用一条边将它们的相应顶点连起来。图5-11在上方给出一幅地图，在下方给出其平面图表示。平面图的m着色问题就是求出：在最多使用m种颜色时，为使任何两个相邻区域都不同色，共有多少种方式可对该地图进行着色。

m着色问题的一个直接状态空间树就是为级别1的顶点 $v_1$ 尝试每种可能的颜色，为级别2的顶点尝试每种可能的颜色，以此类推，直到为级别n中的顶点 $v_n$ 尝试每种可能的颜色。从根节点到叶节点的每条路径都是一种候选解。我们检查是否有任何两个相邻顶点具有相同颜色，以判断一个候选解是不是真正的解。为避免混淆，请记住，下面讨论中的“节点”是指状态空间树中的一个节点，“顶点”是指正在着色的图中的顶点。

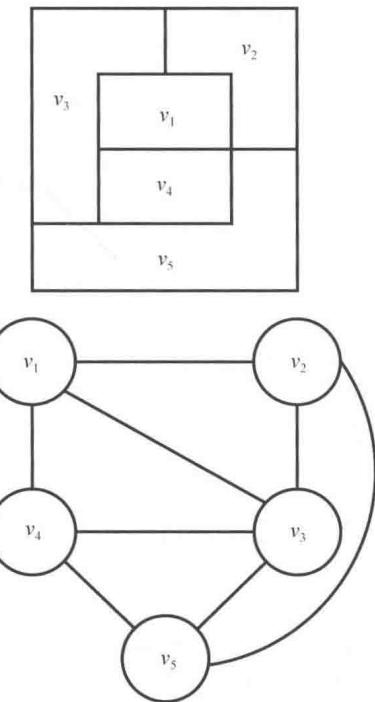
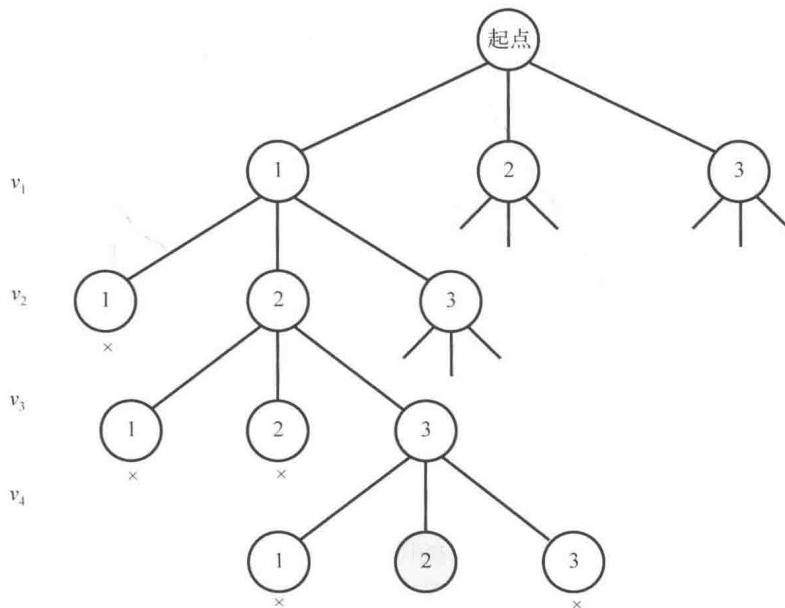


图 5-11 地图（上）和它的平面图表示（下）

假定我们正在为一个节点处的顶点着色，如果与该顶点相邻的一个顶点已经被设定为要为该节点使用的颜色，那这个节点就是没有希望的，于是可以在此问题中回溯。在应用回溯策略解决图 5-10 所示图的 3 着色问题时，会得到图 5-12 所示的一部分已修剪状态空间树。节点中的数字是指该节点正在被着色的顶点使用了哪种颜色。第一个解是在阴影节点处找到的。没有希望的节点标有“ $\times$ ”。在将 \$v\_1\$ 标为颜色 1 后，为 \$v\_2\$ 选择颜色 1 是没有希望的，因为 \$v\_1\$ 与 \$v\_2\$ 相邻。同理，在分别将 \$v\_1\$、\$v\_2\$ 和 \$v\_3\$ 着色为颜色 1、2、3 后，为 \$v\_4\$ 选择颜色 1 也是没有希望的，因为 \$v\_1\$ 与 \$v\_4\$ 相邻。

图 5-12 对图 5-10 所示图的 3 着色问题应用回溯法时生成的一部分已修剪状态空间树。第一个解是在阴影节点处找到的。每个没有希望的节点都标有一个“ $\times$ ”

接下来给出一种算法，用来解决对于所有  $m$  值的  $m$  着色问题。在此算法中，图用一个邻接矩阵表示，如 4.1 节一样。但是，因为这个图是没有加权的，所以矩阵中的每一项要么是真，要么是假，具体取决于两个顶点之间是否存在边。

### 算法 5.5 $m$ 着色问题的回溯算法

**问题：**判断所有满足以下条件的着色方式，仅使用  $m$  种颜色，对一个无向图中的顶点进行着色，使相邻顶点不同色。

**输入：**整数数  $n$  和  $m$ ；一个包含  $n$  个顶点的无向图，这个图用一个二维数组  $W$  表示，它的行、列索引都是 1 至  $n$ ，其中，如果第  $i$  个顶点与第  $j$  个顶点之间存在一条边，则  $W[i][j]$  为真，否则为假。

**输出：**最多使用  $m$  种颜色对该图进行的着色方案，使任何两个相邻顶点都不同色。每种颜色的输出都是一个数组  $vcolor$ ，其索引为 1 至  $n$ ，其中  $vcolor[i]$  是分配给第  $i$  个顶点的颜色（一个介于 1 到  $m$  之间的整数）。

```
void m_coloring(index i)
{
    int color;

    if (promising(i))
        if (i == n)
            cout << vcolor[1] 至 vcolor[n];
        else
            for (color = 1; color <= m; color++){
                vcolor[i+1]=color;           // 为下个顶点尝试每种颜色。
                m_coloring(i+1);
            }
    }

bool promising (index i)
{
    index j;
    bool switch;

    switch = true;
    j=1;
    while (j < i && switch){           // 检查一个相邻顶点是否已经是该颜色。
        if (W[i][j] && vcolor[i]==vcolor[j])
            switch = false;
        j++;
    }
    return switch;
}
```

根据我们的一贯约定， $n$ 、 $w$ 、 $W$  和  $vcolor$  不是任何一个例程的输入。在此算法的实现中，这些例程将被局部定义在一个简单过程内， $n$ 、 $m$  和  $W$  是该过程的输入，而  $vcolor$  则定义为全局变量。对  $m\_coloring$  的顶级调用将是：

```
m_coloring(0);
```

此算法状态空间树中的节点数等于：

$$1 + m + m^2 + \dots + m^n = \frac{m^{n+1} - 1}{m - 1}$$

这一等式在附录 A 的例 A.4 中获得。对于一个给定的  $m$  和  $n$ ，有可能创建一种实例，至少要检查指数级的节点（ $n$  的指数）。例如，如果  $m$  仅为 2，并取一幅图，其中  $v_n$  拥有到达所有其他节点的一条边，除此之外，只有  $v_{n-2}$  和  $v_{n-1}$  之间的一条边了，在此情况下，不存在解，但为了确认这一结果，几乎会查看状态空间树中的每一个节点。和任何回溯算法一样，该算法对于一种特定的大实例可能非常高效。5.3 节介绍的蒙特卡洛方法适用于这一算法，也就是说，可以用它来估计对一种特定实例的效率。

在习题中，将会要求你使用一种算法来求解 2 着色问题，该算法的最差时间复杂度不是  $n$  的指数。当  $m \geq 3$  时，还从来没人能开发出一种在最差情况下非常高效的算法。与“子集之和”问题、0-1 背包问题一样， $m \geq 3$  时的  $m$  着色问题也属于第 9 章讨论的一类问题。即使仅寻求该图的一种  $m$  着色方式，也是如此。

## 5.6 哈密顿回路问题

回想例 3.12，Nancy 和 Ralph 正在竞争同一个销售职位。能在最短时间内跑完销售区域内所有 20 个城市的一位将赢得该工作。利用一种动态规划算法，其时间复杂度为：

$$T(n)=(n-1)(n-2)2^{n-3}$$

Nancy 在 45 秒内找到一条最短旅程。Ralph 尝试计算所有  $19!$  条旅程。因为他的算法需要 3800 多年的时间，所以它仍在运行中。当然，Nancy 赢得了工作。假定老板看到她的工作如此出色，将她的销售区域扩大，变成 40 个城市。但在这一区域内，并非每个城市都有道路连接到所有其他城市。回想一下，Nancy 的动态规划需要 1 微秒的时间处理其基本运算。快速计算后可知，这一算法需要

$$(40-1)(40-2)2^{40-3} \text{ 微秒} = 6.46 \text{ 年}$$

的时间为这个包含 40 个城市的区域确定一条最短旅程。因为这样一个时间量是不可接受的，所以 Nancy 必须寻找另外一种算法。她推断，也许要找出一条最优旅程太过困难了，因此她满足于能够找出任意一条旅程。如果每座城市到所有其他城市之间都有道路，任意排列这些城市都会得到一条旅程。但别忘了，Nancy 的新区域中并不是这样的。因此，她的问题是找出图中的任意旅程。这个问题称为哈密顿回路问题，以提出该问题的威廉·哈密顿爵士的名字命名。该问题可以针对有向图表述（旅行推销员问题的表述方式），也可以针对无向图表述。因为它通常是针对无向图表述的，所以我们这里也选用这种表述方式。对于 Nancy 困境来说，这就意味着，当两个城市相互连接在一起时，存在着一条独一无二的双向道路将它们连接在一起。

具体来说，给定一个连通无向图，哈密顿回路（Hamiltonian Circuit，也称为旅程）是指一条路径，它始于一个给定顶点，对图中的每个顶点都恰好访问一次，最后结束于起始顶点。图 5-13a 中的图包含哈密顿回路  $[v_1, v_2, v_8, v_7, v_6, v_5, v_4, v_3, v_1]$ ，但图 5-13b 中的图没有包含哈密顿回路。哈密顿回路问题就是确定一个连通无向图中的哈密顿回路。

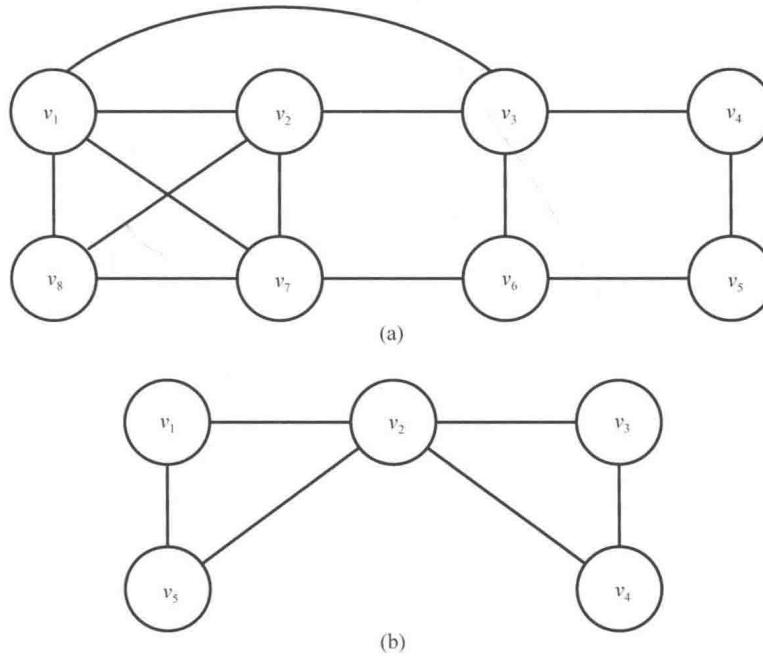


图 5-13 (a)中的图包含哈密顿回路  $[v_1, v_2, v_8, v_7, v_6, v_5, v_4, v_3, v_1]$ ；(b)中的图不包含哈密顿回路

这个问题的状态空间树如下。将起始顶点设在树中的第0级，将它称为路径中的第0个顶点。在第1级，考虑将除起始顶点之外的每个顶点作为起始顶点之后的第一个顶点。在第2级，考虑将同一批顶点中的每个顶点作为第二个顶点，以此类推。最后，在第n-1级，考虑同一批顶点中的每个顶点作为第n-1个顶点。

以下考虑事项使我们能够在这一状态空间树中回溯：

- (1) 路径上的第*i*个顶点必然与路径上的第*i*-1个顶点相邻；
- (2) 第*n*-1个顶点必然与第0个顶点（起始顶点）相邻；
- (3) 第*i*个顶点不能是前*i*-1个顶点之一。

下面的算法利用这些考虑事项进行回溯。这一算法中以硬编码方式规定v<sub>1</sub>作为起始顶点。

### 算法 5.6 哈密顿回路问题的回溯算法

问题：确定一个连通无向的所有哈密顿回路。

输入：正整数*n*和包含*n*个顶点的无向图。该图由一个二维数组*W*表示，它的行列索引范围均为1至*n*，其中，如果在第*i*个顶点与第*j*个顶点之间存在一边条，则*W[i][j]*为真，否则为假。

输出：所有起始于一个给定顶点，将图中所有其他顶点都恰好访问一次，最后终止于起始节点的路径。每条路径的输出是一个索引数组*vindex*，其索引范围为0至*n*-1，其中*vindex[i]*是路径中第*i*个顶点的索引。起始顶点的索引为*vindex[0]*。

```
void hamiltonian (index i)
{
    index j;

    if (promising(i))
        if (i == n - 1)
            cout << vindex[0] 至 vindex[n-1];
        else
            for (j = 2; j<=n; j++) {           // 为下一个顶点尝试所有顶点。
                vindex[i+1]=j;
                hamiltonian(i+1);
            }
    }

bool promising (index i)
{
    index j;
    bool switch;

    if (i==n-1 && !W[vindex[n-1]][vindex[0]])
        switch = false;                  // 第一个顶点必须与最后一个顶点相邻。
    else if (i>0 && !W[vindex[i-1]][vindex[i]])      // 第i个顶点必须与第i-1个顶点相邻。
    else{
        switch = true;
        j = 1;
        while (j<i && switch){          // 检查顶点是否已被选定。
            if (vindex[i]==vindex[j])
                switch = false;
            j++;
        }
    }
    return switch;
}
```

根据约定，*n*、*W*和*vindex*不是任何一个例程的输入。如果这些变量定义为全局变量，则对*hamiltonian*的顶级调用如下：

```
vindex[0]=1;                      // 使v1为起始顶点。
hamiltonian(0);
```

此算法状态空间中的节点数为：

$$1 + (n-1) + (n-1)^2 + \cdots + (n-1)^{n-1} = \frac{(n-1)^n - 1}{n-2}$$

它要比指数复杂度差得多。这一等式在附录 A 的例 A.4 中获得。尽管以下实例并没有检查整个状态空间树，但它检查的节点数仍然要多于指级别。设到达  $v_1$  的唯一边是来自  $v_2$ ，且设除  $v_1$  之外的所有其他顶点之间均有边。这个图中没有哈密顿回路，但为了获知这一信息，该算法需要检查的节点数多于指级别。

回到 Nancy 的难题，为了解决她的 40 城市实例，回溯算法（哈密顿回路问题）花费的时间甚至可能长于动态规划算法（旅行推销员问题）。因为这一问题满足使用蒙特卡洛方法的条件，所以 Nancy 可以使用这一方法来估计该算法解决其实例的效率。但是，蒙特卡洛方法估计的是检查所有回路的时间。因为 Nancy 只需要一个回路，所以她可以在找到第一个回路后（如果存在回路的话），让算法停止。建议读者设计一个  $n=40$  的实例，估计该算法能够多么快地找出这一实例中的所有回路，并运行算法，找出一个回路。这样，你就可以自己为故事设计一个结尾。

即使我们只想要一个旅程，哈密顿回路问题也属于第 9 章讨论的一类问题。

## 5.7 0-1 背包问题

4.5 节用动态规划解决了这一问题，现在使用回溯来解决它。然后我们将对比回溯算法与动态规划算法。

### 5.7.1 0-1 背包问题的回溯算法

回想一下，在这个问题中有一组物品，每件物品都有自己的重量和价值，这些重量和价值是正整数。一个小偷计划将偷来的物品放到一个背包里，如果放入背包中的物品总重量超过某一正整数  $W$ ，背包将会损坏。这个小偷的目的是确定一组物品，在总重量不能超过  $W$  的约束条件下，使总价值最大化。

我们可以使用“子集之和”问题（见 5.4 节）中的同一状态空间树来解决这一问题。也就是说，从根节点向左则包含第一项，向右则排除它。同理，从第 1 级的一个节点向左移则包含第二项，向右移则排除它，以此类推。从根节点到一个叶节点的每条路径都是一个候选解。

这一问题不同于本章讨论的其他问题，因为它是一个最优化问题。这就是说，在查找完成之前，无法知道一个节点是否包含一个解。因此，我们的回溯将稍有不同。在到达一个节点时，如果所包含项目的总价值大于到目前为止的最优解，我们将修改最佳解的当前值。但是，在这个节点的后代节点中还可能找出更好的解（通过偷盗更多物品）。因此，对于最优化问题，我们总会查看一个有希望节点的子节点。下面是对于最优化问题的一般回溯算法。

```
void checknode (node v)
{
    node u;
    if (value(v) 优于 best)
        best = value(v);
    if (promising(v))
        for (v 的每个子节点 u)
            checknode(u);
}
```

变量  $best$  是当前最优解的值， $value(v)$  是该节点处解的值。首先将  $best$  初始化为一个劣于任意候选解的值，然后在顶级传递根节点。注意，只有当我们应当展开一个节点的子节点时，这个节点才是有希望的。回想一下，在其他算法中，如果一个节点处存在一个解，就说它是有希望的。

下面将这一方法用于 0-1 背包问题。首先寻找一些标志，可以告诉我们一个节点是没有希望的。有一个很明显的标志可以表明一个节点是没有希望的——背包中没有容纳更多物品的空间了。因此，如果  $weight$  是到某

一节点时所包含项目的重量之和，则当

$$\text{weight} \geq W$$

时，该节点是没有希望的。即使 weight 等于  $W$ ，它也是没有希望的，因为在最优化问题中，“有希望”表示我们应当展开到子节点。

我们可以使用贪婪算法的考虑事项找到一个不太明显的标志。回想一下，在 4.5 节中，这一方法并没有给出该问题的一个最优解。在这里我们仅使用贪婪考虑事项来限制查找范围，并不会开发一种贪婪算法。为此，首先根据  $p_i/w_i$  的值对项目进行非递减排序，其中  $w_i$  和  $p_i$  分别为第  $i$  件物品的重量和价值。假定我们正在试图确定一个特定节点有没有希望。从这一节点开始，无论如何选择剩余物品，所能获得的价值都不可能高于在部分背包问题条件下的结果。（回想一下，在部分背包问题中，小偷可以盗取一件物品的任意部分。）因此，通过如下展开节点，可以得到所能实现的价值上限。设 profit 是到该节点为止，所包含物品的价值总和。回想一下，weight 是这些项目的重量之和。分别将变量 bound 和 totweight 初始化为 profit 和 weight。接下来，贪婪地获取物品，将它们的价值添加到 bound，将它们的重量添加到 totweight，直到碰到一件物品，如果选择该物品，就会使 totweight 高于  $W$ 。根据剩余重量的允许范围，取得该物品的一部分，并将该部分的价值添加到 bound 中。如果能够仅获得这一最终重量的一部分，那这个节点就不会使 value 等于 bound，但 bound 仍然是通过展开该节点所能实现的价值上限。假定该节点位于第  $i$  级，而且第  $k$  级的节点会使重量总和大于  $W$ 。则

$$\begin{aligned}\text{totweight} &= \text{weight} + \sum_{j=i+1}^{k-1} w_j \\ \text{bound} &= \underbrace{\left( \text{profit} + \sum_{j=i+1}^{k-1} p_j \right)}_{\text{前 } k-1 \text{ 件所取物品的价值}} + \underbrace{(W - \text{totweight}) \times}_{\text{可用于第 } k \text{ 件物品的容量}} \underbrace{\frac{p_k}{w_k}}_{\text{第 } k \text{ 件物品的单位重量价值}}\end{aligned}$$

如果 maxprofit 是当前所找到的最佳解的价值，则当

$$\text{bound} \leq \text{maxprofit}$$

时，位于第  $i$  级的一个节点是没有希望的。我们使用贪婪考虑事项，只是为了获得一个界限，告诉我们是否应当展开一个节点。我们没有用它来贪婪地获取一些不可能在将来再次考虑的物品（就像贪婪方法中所做的那样）。

在给出算法之前，先给出一个例子。

**例 5.6** 假定  $n=4$ ,  $W=16$ , 则有以下结果:

$i$	$p_i$	$w_i$	$\frac{p_i}{w_i}$
1	40 美元	2	20 美元
2	30 美元	5	6 美元
3	50 美元	10	5 美元
4	10 美元	5	2 美元

我们已经根据  $p_i/w_i$  对物品进行了排序。为简单起见，在选择  $p_i$  和  $w_i$  时，使  $p_i/w_i$  为整数。一般情况下，并不要求这样。图 5-14 给出了利用前面讨论的回溯考虑事项而得到的已修剪状态空间树。在每一个节点，自上而下指定了总价值、总重量和界限。也就是前面讨论中提到的变量 profit、weight 和 bound 的取值。最大价值在带有阴影的节点处找到。每个节点都标有它在树中的级别和位置（从左算起）。例如，带有阴影的节点标有(3, 3)，这是因为它处于第 3 级，而且它是这一级从左算起的第三个节点。接下来，给出生成已修剪树的步骤。在这些步骤中，用节点的标记来指代节点。

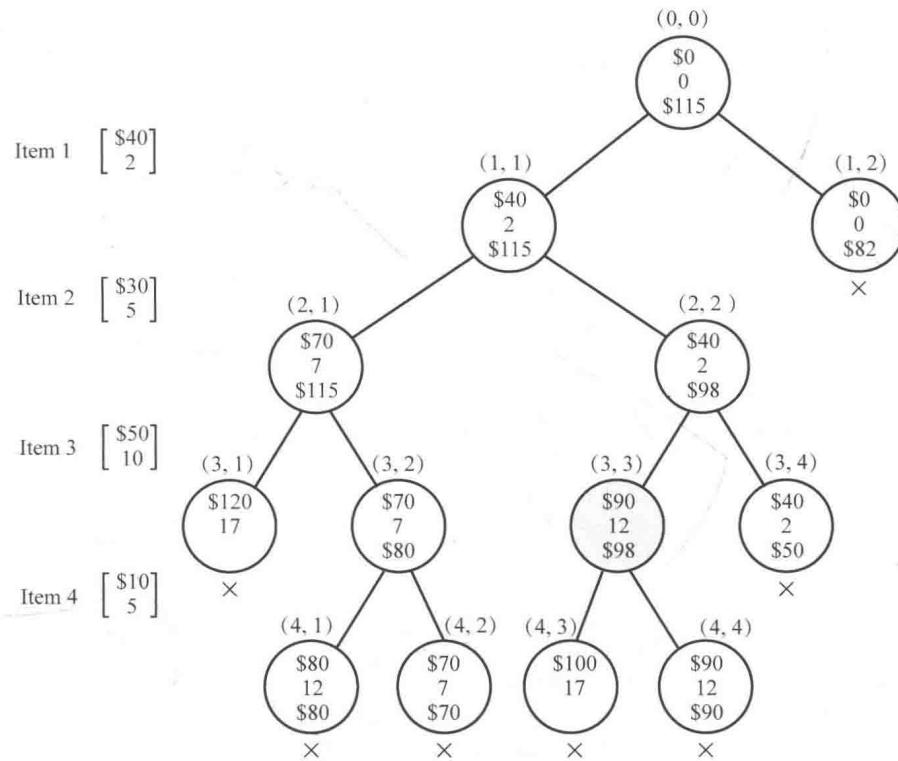


图 5-14 在例 5.6 中使用回溯方法生成的已修剪状态空间树。自上而下，每个节点中存储的内容分别是到该节点为止所偷物品的总价值、总重量和通过展开该节点所能获得的总价值上限。最优解在阴影节点处获得。每个无希望节点都标有一个“ $\times$ ”

- (1) 设定  $\text{maxprofit}$  为 0 美元。
- (2) 查看节点  $(0, 0)$  (根节点)。

(a) 计算其价值和重量。

$$\begin{aligned}\text{profit} &= 0 \text{ 美元} \\ \text{weight} &= 0\end{aligned}$$

(b) 计算其上限。因为  $2+5+10=17$ ，且  $17>16$ ，也就是  $W$  的值，所以第三件物品会使重量之和超过  $W$ 。因此， $k=3$ ，且有

$$\begin{aligned}\text{totweight} &= \text{weight} + \sum_{j=0+1}^{3-1} w_j = 0 + 2 + 5 = 7 \\ \text{bound} &= \text{profit} + \sum_{j=0+1}^{3-1} p_j + (W - \text{totweight}) \times \frac{p_3}{w_3} \\ &= 0 + 40 + 30 + (16 - 7) \times \frac{50}{10} = 115 \text{ 美元}\end{aligned}$$

(c) 判定该节点是有希望的，因为它的重量 0 小于 16，也就是  $W$  的值，且它的上限 115 美元大于 0 美元，也就是  $\text{maxprofit}$  的值。

- (3) 访问节点  $(1, 1)$ 。
- (a) 计算其价值和重量。

$$\begin{aligned}\text{profit} &= 0 + 40 = 40 \text{ 美元} \\ \text{weight} &= 0 + 2 = 2\end{aligned}$$

(b) 因为它的重量 2 小于或等于 16, 也就是  $W$  的值, 而且它的价值 40 美元大于 0 美元, 也就是  $\text{maxprofit}$  的值, 所以设定  $\text{maxprofit}$  的值为 40 美元。

(c) 计算其上限。因为  $2+5+10=17$ , 且  $17>16$ , 也就是  $W$  的值, 所以第三件物品会使重量之和超过  $W$ 。因此,  $k=3$ , 且有

$$\begin{aligned}\text{totweight} &= \text{weight} + \sum_{j=1+1}^{3-1} w_j = 2 + 5 = 7 \\ \text{bound} &= \text{profit} + \sum_{j=1+1}^{3-1} p_j + (W - \text{totweight}) \times \frac{p_3}{w_3} \\ &= 40 + 30 + (16 - 7) \times \frac{50}{10} = 115 \text{ 美元}\end{aligned}$$

(d) 判定该节点是有希望的, 因为它的重量 2 小于 16, 也就是  $W$  的值, 且它的上限 115 美元大于 0 美元, 也就是  $\text{maxprofit}$  的值。

(4) 访问节点(2, 1)。

(a) 计算其价值和重量。

$$\text{profit} = 40 + 30 = 70 \text{ 美元}$$

$$\text{weight} = 2 + 5 = 7$$

(b) 因为它的重量 7 小于或等于 16, 也就是  $W$  的值, 而且它的价值 70 美元大于 40 美元, 也就是  $\text{maxprofit}$  的值, 所以设定  $\text{maxprofit}$  的值为 70 美元。

(c) 计算其上限。因为  $2+5+10=17$ , 且  $17>16$ , 也就是  $W$  的值, 所以第三件物品会使重量之和超过  $W$ 。因此,  $k=3$ , 且有

$$\begin{aligned}\text{totweight} &= \text{weight} + \sum_{j=2+1}^{3-1} w_j = 7 \\ \text{bound} &= 70 + (16 - 7) \times \frac{50}{10} = 115 \text{ 美元}\end{aligned}$$

(d) 判定该节点是有希望的, 因为它的重量 7 小于 16, 也就是  $W$  的值, 且它的上限 115 美元大于 70 美元, 也就是  $\text{maxprofit}$  的值。

(5) 访问节点(3, 1)。

(a) 计算其价值和重量。

$$\text{profit} = 70 + 50 = 120 \text{ 美元}$$

$$\text{weight} = 7 + 10 = 17$$

(b) 因为它的重量 17 大于 16, 也就是  $W$  的值, 所以  $\text{maxprofit}$  的值不变。

(c) 判定该节点是没有希望的, 因为它的重量 17 大于或等于 16, 也就是  $W$  的值。

(d) 不再为这个节点计算界限, 因为它的重量已经决定了它是没有希望的。

(6) 回溯到节点(2, 1)。

(7) 访问节点(3, 2)。

(a) 计算其价值和重量。因为没有包含物品 3, 所以

$$\begin{aligned}\text{profit} &= 70 \text{ 美元} \\ \text{weight} &= 7\end{aligned}$$

(b) 因为它的价值 70 美元小于或等于 70 美元, 也就是  $\text{maxprofit}$  的值, 所以  $\text{maxprofit}$  的值不变。

(c) 计算其上限。第四件物品的重量不会使这些物品的重量之和超过  $W$ , 而且一共只有四件物品。因此,  $k=5$ , 且有

$$\text{bound} = \text{profit} + \sum_{j=3+1}^{5-1} p_j = 70 + 10 = 80 \text{ 美元}$$

(d) 判定该节点是有希望的, 因为它的重量 7 小于 16, 也就是  $W$  的值, 且它的上限 80 美元大于 70 美元, 也就是  $\text{maxprofit}$  的值。

(从现在开始, 我们将价值、重量和界限的计算留作练习。此外, 当  $\text{maxprofit}$  不改变时, 将不再提及。)

(8) 访问节点(4, 1)。

(a) 计算其价值和重量为 80 美元和 12。

(b) 因为它的重量 12 小于或等于 16, 也就是  $W$  的值, 而且它的价值 80 美元大于 70 美元, 也就是  $\text{maxprofit}$  的值, 将  $\text{maxprofit}$  设定为 80 美元。

(c) 计算其上限为 80 美元。

(d) 判定该节点是没有希望的, 因为它的上限 80 美元小于或等于 80 美元, 也就是  $\text{maxprofit}$  的值。  
该状态空间树中的叶节点自动是没有希望的, 因为它们的界限总是小于或等于  $\text{maxprofit}$ 。

(9) 回溯到节点(3, 2)。

(10) 访问节点(4, 2)。

(a) 计算其价值和重量为 70 美元和 7。

(b) 计算其上限为 70 美元。

(c) 判定该节点是没有希望的, 因为它的上限 70 美元小于或等于 80 美元, 也就是  $\text{maxprofit}$  的值。

(11) 回溯到节点(1, 1)。

(12) 访问节点(2, 2)。

(a) 计算其价值和重量为 40 美元和 2。

(b) 计算其上限为 98 美元。

(c) 判定该节点是有希望的, 因为它的重量 2 小于 16, 也就是  $W$  的值, 而且它的上限 98 美元大于 80 美元, 也就是  $\text{maxprofit}$  的值。

(13) 访问节点(3, 3)。

(a) 计算其价值和重量为 90 美元和 12。

(b) 因为它的重量 12 小于或等于 16, 也就是  $W$  的值, 而且它的价值 90 美元大于 80 美元, 也就是  $\text{maxprofit}$  的值, 将  $\text{maxprofit}$  设定为 90 美元。

(c) 计算其上限为 98 美元。

(d) 判定该节点是有希望的, 因为它的重量 12 小于 16, 也就是  $W$  的值, 而且它的上限 98 美元大于 90 美元, 也就是  $\text{maxprofit}$  的值。

(14) 访问节点(4, 3)。

(a) 计算其价值和重量为 100 美元和 17。

(b) 判定该节点是没有希望的, 因为它的重量 17 大于或等于 16, 也就是  $W$  的值。

(c) 不计算这一节点的界限, 因为它的重量已经决定了它是没有希望的。

(15) 回溯至节点(3, 3)。

(16) 访问节点(4, 4)。

(a) 计算其价值和重量为 90 美元和 12。

(b) 计算其界限为 90 美元。

(c) 判定该节点是没有希望的, 因为它的界限 90 美元小于或等于 90 美元, 也就是  $\text{maxprofit}$  的值。

(17) 回溯至节点(2, 2)。

(18) 访问节点(3, 4)。

(a) 计算其价值和重量为 40 美元和 2。

(b) 计算其界限为 50 美元。

(c) 判定该节点是没有希望的，因为它的界限 50 美元小于或等于 90 美元，也就是 maxprofit 的值。

(19) 回溯至根节点。

(20) 访问节点(1, 2)。

(a) 计算其价值和重量为 0 美元和 0。

(b) 计算其界限为 82 美元。

(c) 判定该节点是没有希望的，因为它的界限 82 美元小于或等于 90 美元，也就是 maxprofit 的值。

(21) 回溯至根节点。

(a) 根节点没有更多子节点。大功告成。

在这个已修剪状态空间树中仅有 13 个节点，而整个状态空间树中有 31 个节点。

接下来给出算法。因为这是一个最优化问题，所以要增加一项任务：跟踪当前最佳物品集合及其价值总和。我们用数组 bestset 和变量 maxprofit 完成这一任务。与本章的基本问题不同，这一问题表述为仅需要一个最优解。

### 算法 5.7 0-1 背包问题的回溯算法

**问题：**设给定  $n$  件物品，每件物品都有自己的重量和价值，这些重量和价值是正整数。此外，设给定一个正整数  $W$ 。试确定一组物品，在其重量总和不超过  $W$  的约束条件下，总价值达到最大。

**输入：**正整数  $n$  和  $W$ ；数组  $w$  和  $p$ ，索引范围均为 1 至  $n$ ，均包含有正整数，这些正整数分别根据  $p[i]/w[i]$  值的非递减顺序排列。

**输出：**数组 bestset，其索引范围为 1 至  $n$ ，若最优集中包含第  $i$  件物品，则  $\text{bestset}[i]$  的值为 yes，否则为 no；一个正整数 maxprofit，也就是最大价值。

```

void knapsack (index i,
                int profit, int weight)
{
    if (weight <= W && profit > maxprofit){           // 这个集合是到目前最好的。
        maxprofit = profit;
        numbest = i;                                     // 设定 numbest 为所考虑物品的数目。
        bestset = include;                                // 设定 bestset 为这个解。
    }

    if (promising(i)){
        include[i+1] = "yes";                            // 包含 w[i+1]
        knapsack(i+1, profit + p[i+1], weight + w[i+1]);
        include[i+1] = "no";                             // 不包含 w[i+1]。
        knapsack(i+1, profit, weight);
    }
}

bool promising (index i)
{
    index j, k;
    int totweight;
    float bound;

    if (weight >= W)
        return false;                                 // 只有在应当展开一个节点的子节点时，该节点才是有希望的。
    else {
        j = i + 1;                                  // 必须为子节点留出一些空间。
        bound = profit;
        totweight = weight;
        while (j <= n && totweight + w[j] <= W){
            totweight = totweight + w[j];             // 获取尽可能多的物品。
        }
    }
}

```

```

        bound = bound + p[j];
        j++;
    }
    k=j;
    if (k <= n)           // 使用 k, 以与正文中的公式保持一致。
        bound = bound + (W - totweight)*p[k]/w[k];
                           // 获取第 k 件物品的一部分。
    return bound > maxprofit;
}
}

```

根据我们的一般惯例,  $n$ 、 $w$ 、 $p$ 、 $W$ 、 $\text{maxprofit}$ 、 $\text{include}$ 、 $\text{bestset}$  和  $\text{numbest}$  不是任何一个例程的输入。如果这些变量定义为全局变量, 以下代码将生成最大价值和一个拥有该价值的集合:

```

numbest = 0;
maxprofit = 0;
knapsack(0, 0, 0);
cout << maxprofit;           // 输出最大价值。
for (j = 1; j <= numbest; j++) // 给出物品的一个最优集合。
    cout << bestset[i];

```

回想一下, 该状态空间树中的叶节点自动是没有希望的, 因为它们的界限不能大于  $\text{maxprofit}$ 。因此, 应当不需要在函数  $\text{promising}$  中检查  $i=n$  的终止条件。现在来确认, 我们的算法不需要进行这一检查。如果  $i=n$ ,  $\text{bound}$  不会改变其初始值  $\text{profit}$ 。因为  $\text{profit}$  小于或等于  $\text{maxprofit}$ , 所以表达式  $\text{bound} > \text{maxprofit}$  为真, 这意味着函数  $\text{promising}$  返回  $\text{false}$ 。

当我们重复向状态空间树中的左侧前进, 直到第  $k$  级的节点时, 我们的上限不会改变数值。(再次查看例 5.6 中的前几个步骤即可看出这一点。) 因此, 每次确定一个  $k$  值时, 可以保存它的值, 并继续向左前进且不调用函数  $\text{promising}$ , 直到第  $k-1$  级的一个节点。我们知道, 这个节点的左子节点是没有希望的, 因为包含第  $k$  件物品会使  $\text{weight}$  值大于  $W$ 。因此, 我们仅由此节点向右前进。只有在向右移动之后, 才需要调用函数  $\text{promising}$ , 并确定  $k$  的一个新值。在习题中将要求你给出这一改进。

0-1 背包问题中的状态空间树与“子集之和”问题中的一样。如 5.4 节所示, 树中的节点数为:

$$2^{n+1}-1$$

算法 5.7 为以下实例检查状态空间树中的所有节点。对于一个给定  $n$  值, 令  $W=n$ , 且

$$\begin{aligned} p_i &= 1 & w_i &= 1 & (1 \leq i \leq n-1) \\ p_n &= n & w_n &= n \end{aligned}$$

这个最优解是仅取第  $n$  项, 为找到这个解, 必须遍历所有以下路径: 向右到达  $n-1$  的深度, 然后再向左移动。但在找到最优解之后, 会发现每个非叶节点都是有希望的, 这意味着状态空间树中的所有节点都会被检查。因为蒙特卡洛方法适用于这一问题, 所以可以用它来估计该算法针对一个特定实例的效率。

### 5.7.2 比较 0-1 背包问题的动态规划算法与回溯算法

回想 4.5 节, 用动态规划算法求解 0-1 背包问题时, 在最差情况下计算的项目数属于  $O(\min(2^n, nW))$ 。在最差情况下, 回溯算法检查  $\Theta(2^n)$  个节点。由于增加了界限  $nW$ , 所以动态规划算法看起来更好一些。但在回溯算法中, 通过最差情况很难看出回溯操作通常会节省多少次检查。因为有如此之多的考虑因素, 所以很难从理论上分析这两种算法的相对效率。在诸如此类的情景中, 为比较两种算法, 可以针对许多样本实例运行它们, 并了解哪种算法的性能通常更好一点。Horowitz 和 Sahni (1978 年) 做了这一工作, 并发现, 回溯算法的效率通常要高于动态规划算法。

Horowitz 和 Sahni (1974 年) 将分而治之方法与动态规划方法结合在一起, 为 0-1 背包问题设计了一种算法, 它在最差情况下的复杂度为  $O(2^{n/2})$ 。他们证明了, 这一算法的效率通常高于回溯算法。

## 5.8 习题

### 5.1节和5.2节

1. 使用回溯算法给出  $n=6$  和  $n=7$  时的两个解（每个实例两个解）。
2. 运用  $n$  皇后问题的回溯算法（算法 5.1）求解  $n=8$  时的问题实例，并给出操作步骤。绘制出在找到第一个解时，这一算法产生的已修改状态空间树。
3. 使用过程 `expand` 的版本，而不是过程 `checknode` 的版本，为  $n$  皇后问题编写回溯算法。
4. 编写一个算法，以整数  $n$  为输入，确定  $n$  皇后问题的解的个数。
5. 证明：在不进行回溯时，要为  $n=4$  的  $n$  皇后问题实例找出第一个解，必须检查 155 个节点（与图 5-4 中的 27 个顶点相对照）。
6. 在你的系统上实现  $n$  皇后问题的回溯算法（算法 5.1），并针对  $n=4, 8, 10$  和  $12$  的问题实例运行它。
7. 改进  $n$  皇后问题的回溯算法（算法 5.1），使 `promising` 函数跟踪受已放置皇后控制的列集、左对角线集和右对角线集。
8. 修改  $n$  皇后问题的回溯算法（算法 5.1），使之仅给出一个解，而不是生成全部可能解。
9. 假定有一个  $n$  皇后实例（ $n=4$ ）的解。能否扩展这个解，为  $n=5$  时的问题实例找出一个解？随后，能否使用  $n=4$  和  $n=5$  的解，为  $n=6$  的实例构造一个解？并继续这一动态规划方法，为任何  $n > 4$  的实例找出一个解？说明理由。
10. 找出至少两个没有解的  $n$  皇后问题实例。

### 5.3节

11. 在你的系统上实现算法 5.3（ $n$  皇后问题回溯算法的蒙特卡洛估计），针对  $n=8$  的问题实例运行 20 次，找出 20 个估计的平均值。
12. 修改  $n$  皇后问题的回溯算法（算法 5.1），使它找出为一个问题的一个实例所检查的节点数，针对  $n=8$  的问题实例运行它，并将此结果与第 9 题的平均值进行比较。

### 5.4节

13. 使用“子集之和”问题的回溯算法（算法 5.4）找出以下数字中所有总和等于  $W=52$  的组合方式：

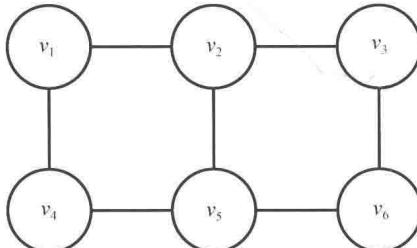
$$w_1=2 \quad w_2=10 \quad w_3=13 \quad w_4=17 \quad w_5=22 \quad w_6=42$$

给出操作步骤。

14. 在你的系统上实现“子集之和”问题的回溯算法（算法 5.4），并针对第 13 题的问题实例运行它。
15. 为事先未对重量进行排序的“子集之和”问题编写一种回溯算法。对比这一算法与算法 5.4 的性能。
16. 修改“子集之和”问题的回溯算法（算法 5.4），使之仅给出一个解，而不是生成所有可能解。与算法 5.4 相比，这一算法的性能如何？
17. 使用蒙特卡洛方法估计“子集之和”问题的回溯算法（算法 5.4）的效率。

### 5.5节

18. 使用  $m$  着色问题的回溯算法（算法 5.5）找出使用红绿白三种颜色为下图进行着色的所有可能方式。给出操作步骤。



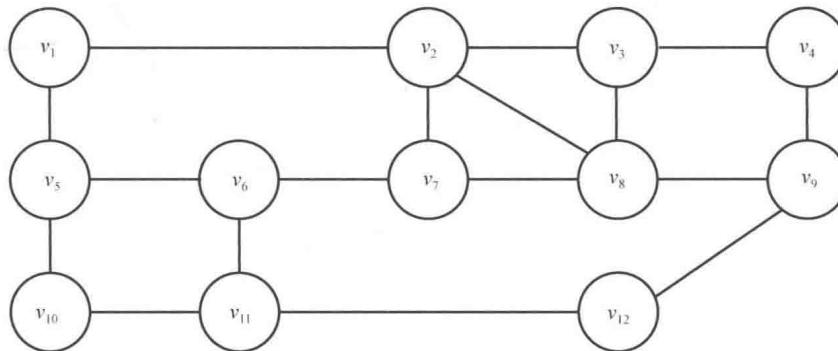
19. 假设为了正确地为一个图着色，选择一个起始顶点和一种颜色，为尽可能多的顶点着色。然后选择一种新

颜色和一个新的未着色顶点，为尽可能多的顶点着色。重复这一过程，直到完成图中所有顶点的着色，或者用完了所有颜色。为这一贪婪算法编写一种算法，为一个  $n$  顶点图进行着色。分析这一算法，并用阶的符号给出分析结果。

20. 使用第 19 题的算法为第 18 题的图着色。
21. 假定我们关心的是如何使用最少量的颜色为一幅图进行着色。第 19 题的贪婪方法能否保证给出最优解？说明理由。
22. 试构建一个包含  $n$  个顶点的连通图，对于该图，3 着色回溯算法需要花费指数时间后才能发现无法对该图实现 3 着色。
23. 比较  $m$  着色问题的回溯算法（算法 5.5）和第 19 题贪婪算法的性能。考虑对比结果及你对第 21 题的答案，说明为什么人们可能更乐意使用基于贪婪方法的算法？
24. 为 2 着色问题编写一种算法，其最差情况下的时间复杂度不是  $n$  的指数。
25. 列出一些可以用  $m$  着色问题表示的实际应用。

### 5.6 节

26. 使用哈密顿回路问题的回溯算法（算法 5.6），找出下图的所有可能的哈密顿回路。给出操作步骤。



27. 在你的系统上实现哈密顿回路问题的回溯方法（算法 5.6），并针对第 26 题的问题实例运行它。
28. 改变哈密顿回路问题的回溯算法（算法 5.6）在第 27 题中的起始顶点，并将其性能与算法 5.6 的性能进行对比。
29. 修改哈密顿电路问题的回溯算法（算法 5.6），使它仅给出一个解，而不是生成所有可能解。这一算法的性能相对于算法 5.6 如何？
30. 分析哈密顿回路问题的回溯算法（算法 5.6），并使用阶的符号给出最差情况复杂度。
31. 使用蒙特卡洛方法评估哈密顿回路问题的回溯算法（算法 5.6）的效率。

### 5.7 节

32. 计算例 5.6（5.7.1 节）中访问节点(4, 1)之后的剩余值和界限。
33. 使用 0-1 背包问题回溯算法（算法 5.7），使以下问题实例的价值最大化。给出操作步骤。

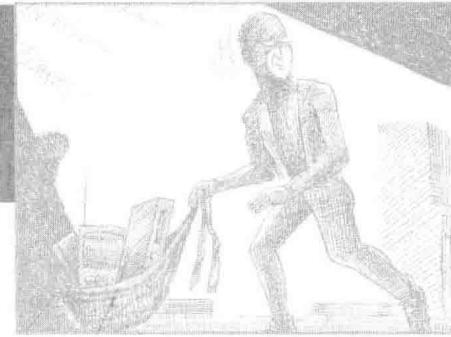
$i$	$p_i$	$w_i$	$\frac{p_i}{w_i}$	$W=9$
1	20 美元	2	10	
2	30 美元	5	6	
3	35 美元	7	5	
4	12 美元	3	4	
5	3 美元	1	3	

34. 在你的系统上实现 0-1 背包问题的回溯算法（算法 5.7），并针对第 33 题的问题实例运行它。

- 
- 35. 实现 0-1 背包问题的动态规划算法（见 4.5.3 节），并使用此问题的大型实例，将这一算法的性能与 0-1 背包问题的回溯算法（算法 5.7）相对比。
  - 36. 改进 0-1 背包问题的回溯算法（算法 5.7），使之仅在向右移动时才调用 promising 函数。
  - 37. 使用蒙特卡洛方法估计 0-1 背包问题的回溯算法（算法 5.7）的效率。

#### 补充习题

- 38. 列出回溯的三种应用。
- 39. 修改  $n$  皇后问题的回溯算法（算法 5.1），使它仅给出在镜像和旋转时保持不变的解。
- 40. 给定一个包含  $n^3$  个单元格的  $n \times n \times n$  立方体，我们要将  $n$  个皇后放在这个立方体内，使任何两个皇后都不会相互挑战（也就是，任何两个皇后不会在同一行、同一列或同一对角线中）。能否扩展  $n$  皇后算法（算法 5.1）以解决这一问题？如果可以，写出该算法，并在你的系统上实现它，以求解  $n=4$  和  $n=8$  的问题实例。
- 41. 修改“子集之和”问题的回溯算法（算法 5.4），在一个变长列表中生成解。
- 42. 试解释如何利用  $m$  着色问题的回溯算法（算法 5.5），使用同样的三种颜色为第 18 题所示图的边进行着色，使拥有公共端点的边染上不同颜色。
- 43. 修改哈密顿回路问题的回溯算法（算法 5.6），为一个加权图找出具有最低代价的哈密顿回路。这个算法的性能如何？
- 44. 修改 0-1 背包问题的回溯算法（算法 5.7），在一个变长列表中生成解。



## 分支定界

我们已经为小偷提供了 0-1 背包问题的两种算法：4.5 节的动态规划算法和 5.7 节的回溯算法。因为这两种算法在最差情况下均为指数时间复杂度，所以它们在解决小偷的特定实例时，都要花费许多年的时间。本章将为我们的小偷再提供一种方法，称为分支定界。后面将会看到，这里开发的分支定界算法（branch-and-bound algorithm）是回溯算法的一种改进。因此，即使其他两种算法都不能高效地解决小偷的实例，分支定界算法也许能够解决。

就求解问题所用的状态空间树来说，分支定界设计策略与回溯方法非常类似。区别在于分支定界方法没有限制任何一种特定的树遍历方式，且仅用于最优化问题。分支定界算法在一个节点处计算一个数值（界限），以判断该节点有没有希望。这个数值是通过展开该节点所能获得的解的价值界限。如果这个界限不比当前最佳解的价值更好，那这个节点就是没有希望的。否则，它就是有希望的。因为最优值在某些问题中是最小值，而在另一些问题中则是最大值，这里说的“更好”可能是更小，也可能是更大，取决于具体问题。和回溯算法的情景一样，分支定界算法在最差情况下通常为指数时间复杂度（甚至更差）。但是，它们对于许多大型实例的效率可能很高。

5.7 节 0-1 背包问题的背包算法实际上是一种分支定界算法。在此算法中，如果 bound 值不大于 maxprofit 的当前值，promising 函数返回 false。但是，回溯算法没有充分发挥分支定界的真正好处。除了使用界限来判断一个节点是否有希望之外，还可以比较有希望节点的界限，并选出具有最佳界限的一个节点，访问其子节点。采用这一方式获得最优解的速度，往往要快于根据某一预定顺序访问各节点的速度（比如尝试优先查找）。这种方法称为采用分支定界修剪的最佳优先查找。此方法的实现是对另外一种称为采用分支定界修剪的宽度优先查找方法进行了简单修改。因此，尽管后一方法相对于深度优先查找并没有什么优势，但 6.1 节仍将首先使用一种宽度优先查找来求解 0-1 背包问题。这样可以更轻松地解释最佳优先查找，并利用它来解决 0-1 背包问题。6.2 节和 6.3 节将最佳优先查找方法应用于另外两个问题。

在继续讨论之前，先来回顾宽度优先查找。对于树来说，宽度优先查找（breadth-first search）包括：首先查看根节点，接下来是第 1 级的所有节点，然后是第 2 级的所有节点，以此类推。图 6-1 给出一棵树的宽度优先查找，在这一查找过程中，是从左向右进行的。这些节点根据其访问顺序编号。

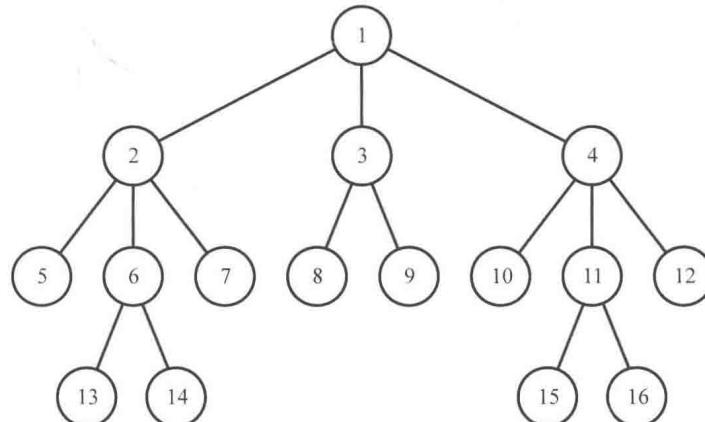


图 6-1 树的宽度优先查找。这些节点根据其访问顺序编号。一个节点的子节点是从左向右访问的

与深度优先查找不同的是，对于宽度优先查找不存在简单的递归算法。但是，可以使用一个队列来实现它。下面的算法就是这样做的。此算法是专门为树编写的，因为目前我们只对树感兴趣。我们用一个 enqueue 过程在队列的末尾插入一项，用一个 dequeue 过程从其前端删除一项。

```

void breadth_first_tree_search (tree T);
{
    queue_of_node Q;
    node u, v,
    initialize(Q);           // 将 Q 初始化为空
    v=T 的根节点;
    访问 v;
    enqueue(Q, v);
    while (!empty(Q)){
        dequeue(Q, v);
        for (v 的每个子节点 u) {
            访问 u;
            enqueue(Q, u);
        }
    }
}

```

如果你不确认这一过程产生了一个宽度优先查找，应当针对图 6-1 中的树实际运行一遍此算法。在该树中，如前所述，一个节点的子节点是从左向右查看的。

## 6.1 用 0-1 背包问题说明分支定界

现有通过对 0-1 背包问题的应用来说明如何使用分支定界设计策略。首先讨论一种简单版本，名为“带有分支定界修剪的宽度优先查找”。然后，再给出此简单版本的一种改进：带有分支定界修剪的最佳优先查找。

### 6.1.1 带有分支定界修剪的宽度优先查找

让我们用一个例子来说明这一方法。

**例 6.1** 假定有例 5.6 中给出的 0-1 背包问题实例。即， $n=4$ ,  $W=16$ ，并有：

$i$	$p_i$	$w_i$	$\frac{p_i}{w_i}$
1	40 美元	2	20 美元
2	30 美元	5	6 美元
3	50 美元	10	5 美元
4	10 美元	5	2 美元

和例 5.6 中一样，这些物品已经根据  $p_i/w_i$  进行了排序。在使用带有分支定界修剪的宽度优先查找时，与例 5.6 中使用回溯方法时的做法完全一致，只是我们使用了宽度优先查找，而不是深度优先查找。即，设 weight 和 profit 是到一个节点为止，所含物品的总重量和总价值。为判断该节点是否有希望，分别将 totweight 和 bound 初始化为 weight 和 profit，然后贪婪地获取物品，将它们的重量和价值加到 totweight 和 bound，直到某一件物品的重量使 totweight 超过  $W$ 。我们在不超出容纳重量的前提下，获取该物品的一部分，并这一部分的价值加到 bound。这样，bound 就变为通过展开该节点所能获得的价值总和上限。如果此节点位于第  $i$  级，且位于第  $k$  级的节点是使重量大于  $W$  的一个，则

$$\text{totweight} = \text{weight} + \sum_{j=i+1}^{k-1} w_j$$

及

$$\text{bound} = \left( \text{profit} + \sum_{j=i+1}^{k-1} p_j \right) + (w - \text{totweight}) \times \frac{p_k}{w_k}$$

如果这个界限小于或等于  $\text{maxprofit}$ , 那这个节点就是没有希望的 ( $\text{maxprofit}$  是到该点为止, 所发现的最佳解的价值)。回想一下, 如果

$$\text{weight} \geq W$$

则一个节点也是没有希望的。图 6-2 中给出一棵经过修剪的状态空间树, 它是对本例中的实例应用宽度优先查找, 并使用上述界限修剪分支后的结果。在每个节点处, 自上而下给出 profit、weight 和 bound 的取值。阴影节点是得到最大价值的节点。这些节点根据其在树中的级别和从左侧算起的位置进行标记。

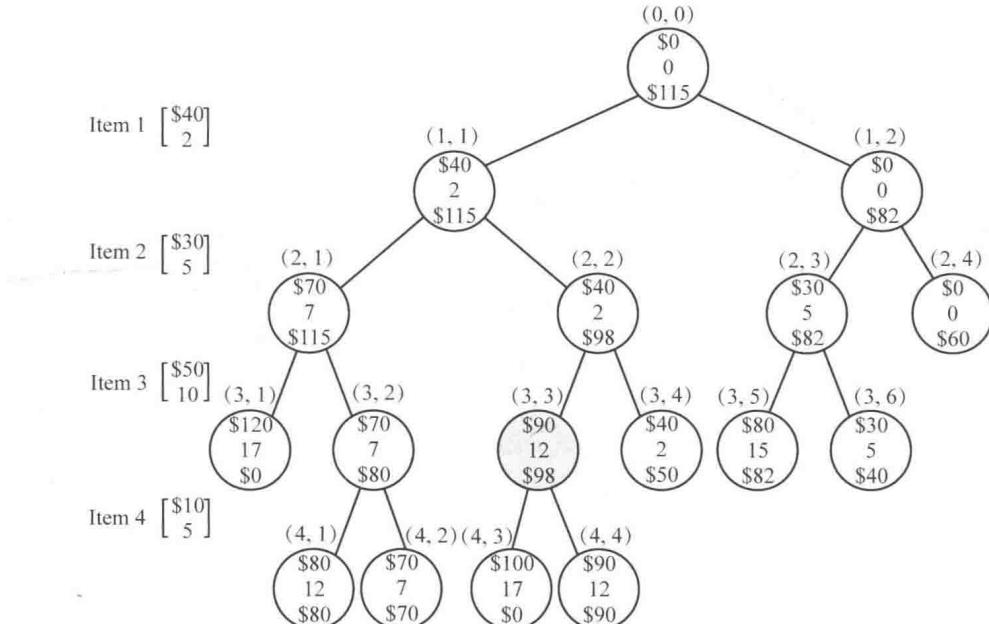


图 6-2 在例 6.1 中使用带有分支定界修剪的宽度优先查找方法, 生成的经过修剪的状态空间树。在每个节点中自上而下存储的是到该节点为止所盗物品的总价值、总重量以及展开该节点所能获得的总价值。阴影节点是找出最优解的位置

因为这些步骤与例 5.6 中非常类似, 所以我们不再逐步完成它们, 只提几个重要的点。我们使用节点在树中的级别和从左算起的位置来指代各个节点。首先, 注意节点(3, 1)和(4, 3)的界限为 0 美元。分支定界算法检查一个节点的界限是否优于当前最佳解的值, 以此来判断是否要展开该节点。然后, 当一个节点因为其重量不小于  $W$  而没有希望时, 将其界限设定为 0 美元。这样, 就能确保它的界限不会优于当前最佳解的值。其次, 回想在对这一实例应用回溯方法 (深度优先查找) 时, 发现节点(1, 2)是没有希望的, 没有展开这一节点。

但在宽度优先查找中, 这一节点是第三个被访问的节点。在访问它时,  $\text{maxprofit}$  的值仅为 40 美元。因为它的界限 82 美元超过了此时的  $\text{maxprofit}$ , 所以我们展开该节点。最后, 在带有分支定界修剪的简单宽度优先查找中, 到底要不要访问一个节点的子节点, 是在访问该节点时决定的。也就是说, 如果到子节点的分支被修剪, 就是在访问该节点时修剪它们。因此, 在访问节点(2, 3)时, 我们决定访问它的子节点, 因为此时的  $\text{maxprofit}$  值仅为 70 美元, 而这个节点的界限为 82 美元。与深度优先查找不同, 在宽度优先查找中,  $\text{maxprofit}$  的值可以在实际访问子节点时修改。在本例中, 当我们访问节点(2, 3)的子节点时,  $\text{maxprofit}$  的值为 90 美元。因此, 我们浪费了检查这些子节点的时间。在下一节介绍的最佳优先查找中会避免这一情况。

前面已经演示了这一方法，现在给出带有分支定界修剪的宽度优先查找的一般算法。尽管我们说状态空间树  $T$  是这一通用算法的输入，但在实际应用中，此状态空间树只是隐式存在的。问题参数是算法的真正输入，并决定了状态空间树  $T$ 。

```
void breadth_first_branch_and_bound(state_space_tree T,
                                     number& best)
{
    queue_of_node Q;
    node u, v;

    initialize(Q);           // 将 Q 初始化为空。
    v = T 的根节点;          // 访问根节点。
    enqueue(Q, v);
    best = value(v);
    while(!empty(Q)){
        dequeue(Q, v);
        for (v 的每个子节点 u){ // 访问每个子节点。
            if (value(u) 优于 best)
                best = value(u);
            if (bound(u) 优于 best)
                enqueue(Q, u);
        }
    }
}
```

此算法是对本章开头所介绍的宽度优先查找算法的一种修改。但在这一算法中，只有当一个节点的界限优于当前最佳解时，才会展开该节点（也就是访问其子节点）。当前最佳解（变量  $best$ ）的值被初始化为根节点处解的值。在某些应用中，在根节点处不存在解，这是因为必须在状态空间树的叶节点处才能获得解。在这种情况下，我们将  $best$  初始化为一个劣于任意解的值。在 `breadth_first_branch_and_bound` 的每次应用中，函数 `bound` 和 `value` 各不相同。后面将会看到，我们通常不会实际编写函数 `value`，而是直接计算该值。

接下来给出一个专门解决 0-1 背包问题的算法。因为没有递归可资利用（这就是说，没有每次递归调用时创建的新变量可供使用），所以必须将属于一个节点的信息存储到该节点处。因此，此算法中的节点将具有如下类型：

```
struct node
{
    int level;           // 该节点在树中的级别。
    int profit;
    int weight;
};
```

#### 算法 6.1 用于 0-1 背包问题的带有分支定界修剪算法的宽度优先查找

**问题：**给定  $n$  件物品，每件物品都有自己的重量和价值。这些重量和价值是正整数。此外，设给定一个正整数  $W$ 。试确定一组物品，在其重量之和不超过  $W$  的约束条件下，使其总价值最大。

**输入：**正整数  $n$  和  $W$ ；正整数数组  $w$  和  $p$ ，索引范围均为 1 至  $n$ ，分别根据  $p[i]/w[i]$  的值进行非递减顺序排列。

**输出：**一个正整数  $maxprofit$ ，即最优集合中的价值之和。

```
void knapsack2 (int n,
                 const int p[], const int w[],
                 int W,
                 int& maxprofit)
{
    queue_of_node Q;
    node u, v;

    initialize(Q);           // 将 Q 初始化为空。
    v.level = 0; v.profit = 0; v.weight = 0; // 将 v 初始化为根节点。
```

```

maxprofit = 0;
enqueue(Q, v);
while (!empty(Q)){
    dequeue(Q, v);
    u.level = v.level + 1;
    u.weight = v.weight + w[u.level];
    u.profit = v.profit + p[u.level];

    if (u.weight <= W && u.profit > maxprofit)
        maxprofit = u.profit;
    if (bound(u)>maxprofit)
        enqueue(Q, u);
    u.weight = v.weight;
    u.profit = v.profit;
    if (bound(u)>maxprofit)
        enqueue(Q, u);
}
}

float bound (node u)
{
    index j, k;
    int totweight;
    float result;
    if (u.weight >= W)
        return 0;
    else{
        result = u.profit;
        j=u.level+1;
        totweight = u.weight;
        while (j <= n && totweight+w[j]<=W){
            totweight = totweight + w[j];           // 获取尽可能多的物品。
            result = result + p[j];
            j++;
        }
        k = j;                                // 使用 k, 以与正文中保持一致。
        if (k <= n)
            result = result +(W-totweight)*p[k]/w[k]; // 获取第 k 件物品的一部分。
        return result;
    }
}

```

在未包含当前物品时，不需要检查  $u.profit$  是否超过  $maxprofit$ ，因为在此情况下， $u.profit$  是与  $u$  的父节点相关联的价值，这就是说，它不可能超过  $maxprofit$ 。不需要将此界限存储在一个节点中（如图 6-2 中所示），因为在将这个界限与  $maxprofit$  进行对比之后，不再需要引用该界限。

函数  $bound$  与算法 5.7 中的函数  $promising$  基本相同。区别在于，我们在编写  $bound$  时，依据的正是创建分支定界算法的原则，因此，函数  $bound$  返回一个整数。而函数  $promising$  返回一个布尔值，因为它是根据回溯原则编写的。在我们的分支定界算法中，与  $maxprofit$  的对比是在发出调用的过程中进行的。在函数  $bound$  中不需要检查条件  $i=n$ ，因为在这种情况下， $bound$  返回的值小于或等于  $maxprofit$ ，这意味着不会将此节点放到队列中。

算法 6.1 不会生成一个最优物品集，它只是判断最优集合中的价值总和。可以对此算法做如下修改，使其生成一个最优集合。每个节点中还存储着一个变量  $items$ ，它是到当前节点为止所包含物品的集合，我们还维护了一个变量  $bestitems$ ，它是当前的最佳物品集合。当  $maxprofit$  被设定为等于  $u.profit$  时，还设定  $bestitems$  等于  $u.items$ 。

### 6.1.2 带有分支定界修剪的最佳优先查找

一般情况下，宽度优先查找策略并不优于深度优先查找（回溯）。但是，我们也可以对查找过程进行改进，使界限的用途不仅限于判断一个节点是否有希望。在访问一个给定节点的所有子节点之后，可以看到所有未展开的有希望节点，并展开其中具有最佳界限的一个。回忆一下，当一个节点的界限优于当前最佳解的值时，这个节点是有希望的。采用这种方法获取最优解的速度，往往要快于按照预定顺序盲目处理的速度。下面的例子演示了这一方法。

**例 6.2** 假定有例 6.1 中的 0-1 背包问题实例。最佳优先查找会生成图 6-3 中经过修剪的状态空间树。同样在树中每个节点处自上而下给出 profit、weight 和 bound 的值。阴影节点是获取最大价值的位置。下面给出生成这棵树的步骤。我们再次用节点在树中的级别及从左算起的位置来指代该节点。数值与界限的计算方式与例 5.6 和例 6.1 中完全相同。在逐步执行这些步骤时没有给出计算过程。此外，只会提到何时发现一个节点是没有希望的，而不会提到何时发现它是有希望的。

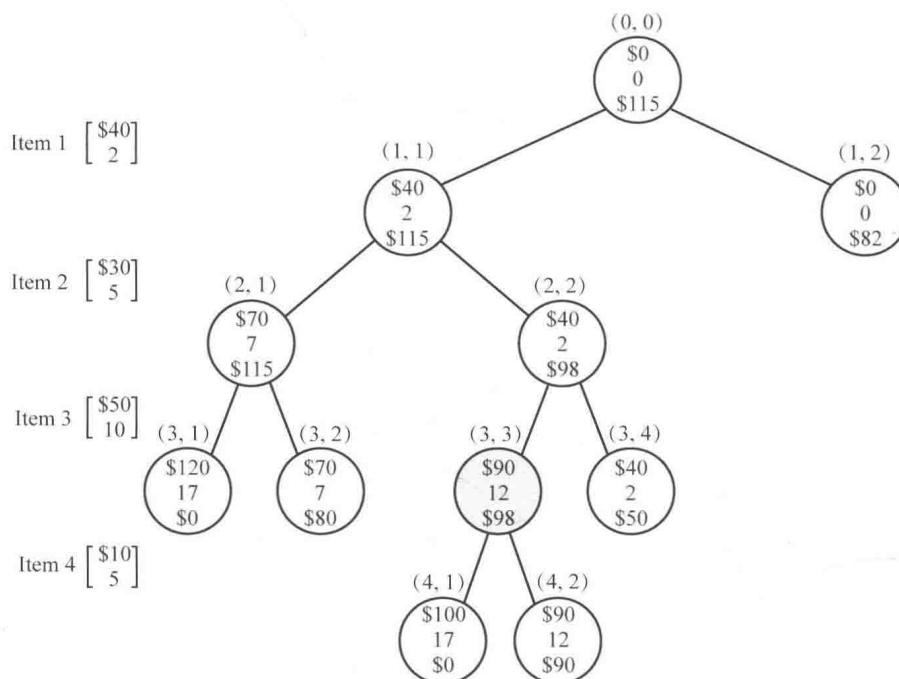


图 6-3 在例 6.2 中使用带有分支定界修剪的最佳优先查找生成的已修剪状态状态树。在每个节点中自上而下存储的是到该节点为止所偷物品的总价值、通过展开该节点所能获得的总价值上限。带有阴影的节点是发现最优解的位置

这些步骤如下。

(1) 访问节点(0, 0)（根节点）。

(a) 设定其价值和重量为 0 美元和 0。

(b) 计算其界限为 115 美元。（具体计算见例 5.6。）

(c) 设定 maxprofit 为 0。

(2) 访问节点(1, 1)。

(a) 计算其价值和重量为 40 美元和 2。

(b) 因为它的重量 2 小于或等于 16，也就是  $W$  的值，而且它的价值 40 美元大于 0 美元，也就是 maxprofit 的值，所以设定 maxprofit 为 40 美元。

(c) 计算其界限为 115 美元。

- (3) 访问节点(1, 2)。
- 计算其价值和重量为 0 美元和 0。
  - 计算其界限为 82 美元。
- (4) 确定具有最大界限的有希望、未展开节点。
- 因为节点(1, 1)的界限为 115, 节点(1, 2)的界限为 82 美元, 所以节点(1, 1)是具有最大界限的有希望、未展开节点。接下来访问其子节点。
- (5) 访问节点(2, 1)。
- 计算其价值和重量为 70 美元和 7。
  - 因为它的重量 7 小于或等于 16, 也就是  $W$  的值, 而且它的价值 70 美元大于 40 美元, 也就是 maxprofit 的值, 所以设定 maxprofit 为 70 美元。
  - 计算其界限为 115 美元。
- (6) 访问节点(2, 2)。
- 计算其价值和重量为 40 美元和 2。
  - 计算其界限为 98 美元。
- (7) 确定具有最大界限的有希望、未展开节点。
- 该节点为节点(2, 1)。接下来访问其子节点。
- (8) 访问节点(3, 1)。
- 计算其价值和重量为 120 美元和 17。
  - 因为它的重量 17 大于或等于 16, 也就是  $W$  的值, 所以确定它是没有希望的。我们将它的界限设定为 0 美元, 使它成为没有希望的。
- (9) 访问节点(3, 2)。
- 计算其价值和重量为 70 美元和 7。
  - 计算其界限为 80 美元。
- (10) 确定具有最大界限的有希望、未展开节点。
- 该节点为节点(2, 2)。接下来访问其子节点。
- (11) 访问节点(3, 3)。
- 计算其价值和重量为 90 美元和 12。
  - 因为它的重量 12 小于或等于 16, 也就是  $W$  的值, 而且它的价值 90 美元大于 70 美元, 也就是 maxprofit 的值, 所以设定 maxprofit 为 90 美元。
  - 此时, 节点(1, 2)和(3, 2)变为没有希望的, 因为它们的各自界限 82 美元和 80 美元都小于或等于 90 美元, 也就是 maxprofit 的新值。
  - 计算其界限为 98 美元。
- (12) 访问节点(3, 4)。
- 计算其价值和重量为 40 美元和 2。
  - 计算其界限为 50 美元。
  - 确定它是没有希望的, 因为它的界限 50 美元小于或等于 90 美元, 也就是 maxprofit 的值。
- (13) 确定具有最大界限的有希望、未展开节点。
- 唯一未展开的有希望节点是节点(3, 3)。接下来访问其子节点。
- (14) 访问节点(4, 1)。
- 计算其价值和重量为 100 美元和 17。
  - 因为它的重量 17 大于或等于 16, 也就是  $W$  的值, 所以判断它是没有希望的。将其界限设定为 0 美元。

(15) 访问节点(4, 2)。

(a) 计算其价值和重量为 90 美元和 12。

(b) 计算其界限为 90 美元。

(c) 因为它的界限 90 美元小于或等于 90 美元，也就是 maxprofit 的值，所以判断该节点是没有希望的。状态空间树中的叶节点自动是没有希望的，因为它们的界限不可能超过 maxprofit。

因为现在没有未展开的有希望节点了，所以工作完成。

使用最佳优先查找时，仅检查了 11 个节点，比使用宽度优先查找中检查的节点数少 6 个（图 6-2），比使用深度优先查找中检查的节点数少 2 个（见图 5-14）。减少 2 个节点并不会留下什么深刻印象，但是，在大型状态空间树中，当使用最佳优先查找方法查找最优解时，这一节点数量可能是非常巨大的。然而必须强调的是，看起来是最佳的节点并不一定能实际给出最优解。在例 6.2 中，节点(2, 1)看起来要优于节点(2, 2)，但节点(2, 2)会给出最优解。一般情况下，最佳优先查找最终仍然能为某些实例生成大多数或全部状态空间树。

最佳优先查找的实现是对宽度优先查找的一种简单修改。我们不使用队列，而使用优先级队列。回忆 4.4.2 节讨论的优先级队列。最佳优先查找算法的一般算法如下。再次强调，树  $T$  只是隐式存在的。在算法中，过程  $\text{insert}(PQ, v)$  将  $v$  添加到优先级队列  $PQ$ ，而过程  $\text{remove}(PQ, v)$  删除具有最佳界限的节点，并将其值指定为  $v$ 。

```
void best_first_branch_and_bound (state_space_tree T,
                                  number& best)
{
    priority_queue_of_node PQ;
    node u, v;

    initialize(PQ); // 将 PQ 初始化为空。
    v = T 的根节点;
    best = value(v);
    insert(PQ, v);

    while(!empty(PQ)){ // 删除具有最佳界限的节点。
        remove(PQ, v);
        if (bound(v) 优于 best) // 检查节点是否仍然是有希望的。
            for (v 的每个子节点 u){
                if (value(u) 优于 best)
                    best = value(u);
                if (bound(u) 优于 best)
                    insert(PQ, u);
            }
    }
}
```

除了使用优先级队列代替队列之外，在从优先级队列中删除一个节点之后，还增加了一次检查。此检查判断该节点的界限是否仍然优于  $best$ 。我们就是这样在访问一个节点之后判断它是否变为没有希望的。例如，图 6-3 中的节点(1, 2)在访问它时是有希望的。在我们的实现中，就是在将其插入  $PQ$  的时刻。但是，当  $maxprofit$  的值变为 90 美元时，这个节点就变成没有希望的了。在我们的实现中，这一过程是在从  $PQ$  中删除该节点之前。我们能够知道这一点，是因为在将它从  $PQ$  删除之后，将它的界限和  $maxprofit$  进行了对比。这样，就避免在访问一个变成没有希望的节点之后，再访问它的子节点。

0-1 背包问题的这一特定算法如下。因为需要一个节点在插入、删除时的界限，并对优先级队列中的节点排序，所以将界限存储在节点中。类型声明如下：

```
struct node
{
    int level; // 节点在树中的级别
    int profit;
    int weight;
    float bound;
};
```

### 算法 6.2 0-1 背包问题的带有分支定界修剪算法的最佳优先查找

问题：给定  $n$  件物品，每件物品都有自己的重量和价值。这些重量和价值是正整数。此外，设给定一个正整数  $W$ 。试确定一组物品，在其重量之和不超过  $W$  的约束条件下，使其总价值最大。

**输入:** 正整数  $n$  和  $W$ ; 正整数数组  $w$  和  $p$ , 索引范围均为 1 至  $n$ , 分别根据  $p[i]/w[i]$  的值排列为非递减顺序。

输出：一个正整数 maxprofit，即最优集合中的价值之和。

```

void knapsack3 (int n,
                 const int p[], const int w[],
                 int W,
                 int& maxprofit)
{
    priority_queue<node> PQ;
    node u, v;

    initialize(PQ);                                // 将 PQ 初始化为空。
    v.level = 0; v.profit = 0; v.weight = 0;          // 将 v 初始化为根节点。
    maxprofit = 0;
    v.bound = bound(v);
    insert(PQ, v);

    while (!empty(PQ)){                            // 删除具有最佳 bound 的节点。
        remove(PQ, v);
        if (v.bound > maxprofit){                  // 检查节点是否仍然是有希望的。
            u.level = v.level + 1;
            u.weight = v.weight + w[u.level];
            u.profit = v.profit + p[u.level];

            if (u.weight <= W && u.profit > maxprofit)
                maxprofit = u.profit;
            u.bound = bound(u);
            if (u.bound > maxprofit)
                insert(PQ, u);
            u.weight = v.weight;
            u.profit = v.profit;
            u.bound = bound(u);
            if (u.bound > maxprofit)
                insert(PQ, u);
        }
    }
}

```

函数 bound 与算法 6.1 中相同。

## 6.2 旅行推销员问题

在例 3.12 中, Nancy 战胜 Ralph, 赢得了销售职位, 因为她使用一种  $\Theta(n^2 2^n)$  动态规划算法来解决旅行推销员问题, 在 45 秒内为一个包含 20 座城市的区域找出了一条最优旅程。Ralph 使用暴力算法生成所有 19! 条旅程。因为暴力算法需要超过 3800 年的时间, 所以它还在运行之中。我们上一次见到 Nancy 是在 5.6 节, 她当时的销售区域扩展到 40 座城市。因为她的动态规划算法需要六年多的时间才能为这一区域找出一条最优旅程, 所以她退而求其次, 只需找出任意一条旅程即可。她使用哈密顿回路问题的回溯算法来完成这一任务。即使这一算法的确高效地找出了一条旅程, 但它与“最优”可能相距甚远。例如, 如果在两座相距 2 英里的城市之间存在一条长达 100 英里的迂回路线, 上述算法很可能会生成一条包含此迂回路线的旅程, 而实际上可能存在一座城市, 距离上述两城市分别只有 1 英里, 用它就能将这两座城市连在一起。这就是说, 利用这一回溯算法生成的旅程, Nancy 周游其区域的效率可能会非常低下。鉴于上述因素, 她可能希望自己最好回过头来, 查找一条最优旅程。如果 40 座城市是高度连通的, 那让回溯算法生成所有旅程可能是行不通的, 因为在最坏情况下的旅

程数可能会达到指数级。假设 Nancy 的老师没有在她的算法课中讲授分支界定方法（这就是为什么 Nancy 在 5.6 节满足于有任意一条旅程即可）。她回去翻看算法教科书，发现分支定界方法是专门为最优化问题设计的，于是决定用它来解决旅行推销员问题。她可能像下面这样操作。

回忆一下，这个问题的目的是在有向图中找出一条最短路径，始于一个给定顶点，将图中的每个顶点恰好访问一次，最后再回到起始顶点。这样一条路径称为最优旅程（optimal tour）。因为从哪个顶点开始是无关紧要的，所以起始顶点可以直接设为第一个顶点。图 6-4 给出了一个图的邻接矩阵表示以及该图的一个最优旅程，这个图包含五个顶点，从每个顶点到其他每个顶点都存在一条边。

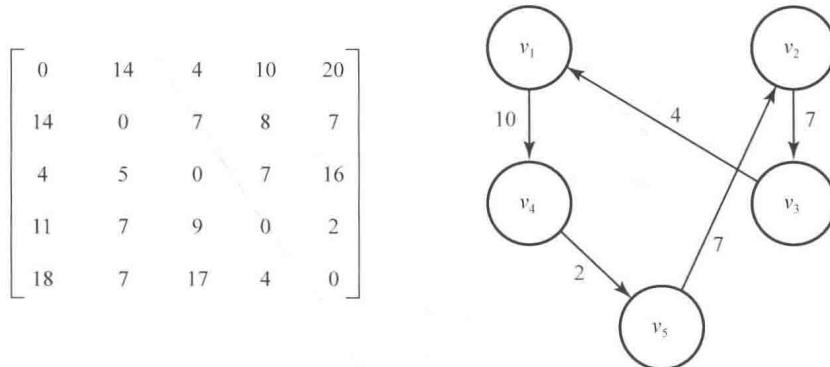


图 6-4 一幅图的邻接矩阵表示（左）（在此图中，每个顶点到其他每个顶点之间都有一条边）以及图中的节点和最优旅程中的边（右）

这一问题有一棵显而易见的状态空间树，其中，除起始顶点之外的每个顶点都在第 1 级被尝试作为（起始顶点之后的）第一个顶点。除起始顶点及第 1 级选定顶点之外的每个顶点都在第 2 级被尝试作为第二个顶点，以此类推。图 6-5 中给出这一状态空间树的一部分，其中有五个顶点，每个顶点到其他每个顶点之间都存在一条边。在以下讨论中，“节点”是指状态空间树中的节点，“顶点”是指图中的顶点。在图 6-5 中的每个顶点处，已经包含了在该节点处选定的路径。为简单起见，我们直接用一个顶点在图中的索引来表示该顶点。一个不是叶节点的节点可以表示所有以该节点所存路径作为起始的旅程。例如，包含[1, 2, 3]的节点表示所有以路径[1, 2, 3]作为起始的旅程。也就是说，它表示旅程[1, 2, 3, 4, 5, 1]和[1, 2, 3, 5, 4, 1]。每个叶节点表示一条旅程。我们需要找出一个包含最优旅程的叶节点。当一个节点存储的路线中有四个顶点时，我们将停止展开该树，因为此时，第 5 个顶点是唯一确定的。例如，最左边的叶节点表示旅程[1, 2, 3, 4, 5, 1]，因为一旦指定了路线[1, 2, 3, 4]，下一个顶点必然就是第 5 个。

要使用最佳优先查找，需要能为每个节点确定一个界限。考虑到 0-1 背包问题的目标（在保持总重量不超过  $W$  的情况下使价值最大），我们计算一个界限，也就是通过展开一个给定节点所能得到的价值上限，如果此界限大于当前最大价值，就说该节点是有希望的。在这个问题中，也需要确定一个界限，但它是通过展开一个给定节点所能获得的最短旅程长度（是一个下限），只有当这个下限小于当前最短旅程长度时，才说这个节点是有希望的。可以像下面这样获得一个界限。在任意旅程中，要离开一个顶点，必然要选择一条边，而这条边的长度必然不短于以该顶点为始点的最短边。因此，离开顶点  $v_i$  的代价（所选边的长度）的下限就是邻接矩阵中第  $i$  行各项的最小值（等于 0 的项目除外），离开顶点  $v_2$  的代价下限，就是第 2 行所有非零项的最小值，以此类推。对于图 6-4 所示图中的五个顶点，离开这些顶点的代价下限分别为：

$$\begin{aligned}
 v_1 & \text{ minimum}(14, 4, 10, 20) = 4 \\
 v_2 & \text{ minimum}(14, 7, 8, 7) = 7 \\
 v_3 & \text{ minimum}(4, 5, 7, 16) = 4 \\
 v_4 & \text{ minimum}(11, 7, 9, 2) = 2 \\
 v_5 & \text{ minimum}(18, 7, 17, 4) = 4
 \end{aligned}$$

因为一条旅程必须离开每个顶点一次，所以一条旅程的长度下限就是这些最小值之和。因此，一条周游路

径的长度下限为：

$$4+7+4+2+4=21$$

这并不是说存在这样一个长度的旅程，而是说不可能存在长度更短的旅程。

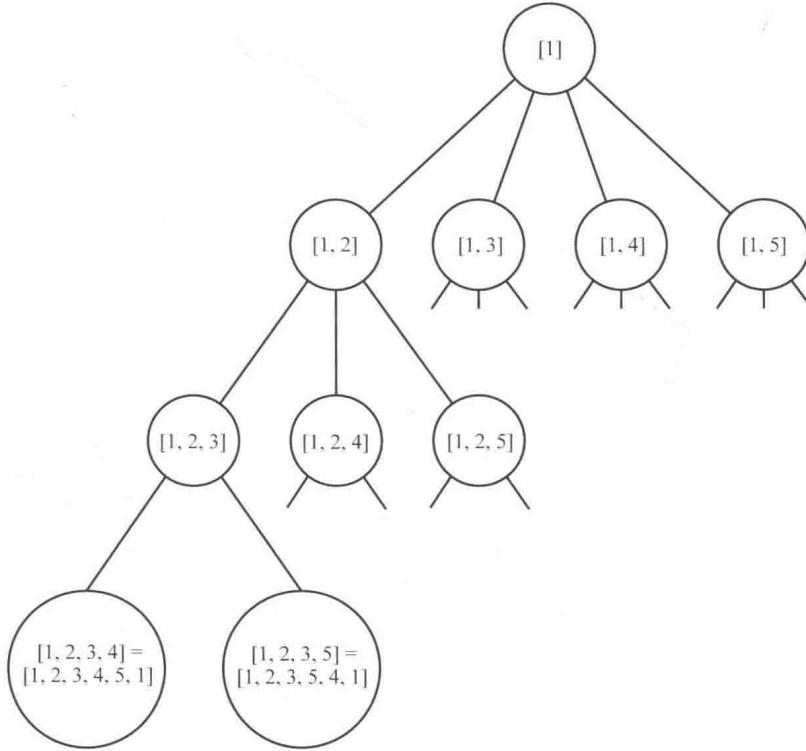


图 6-5 旅行推销员问题的一棵状态空间树，其中有五个顶点，每个节点处存储了部分旅程中的顶点索引

假定已经访问了图 6-5 中包含了  $[1, 2]$  的节点。在此情况下，我们已经决定将  $v_2$  作为旅程上的第二个顶点，到达  $v_2$  的代价就是从  $v_1$  到  $v_2$  的边上的权重，即 14。因此，通过展开这一节点获得的任意周游路径，其离开这些顶点的代价下限为：

$v_1$	14
$v_2$	$\text{minimum}(7, 8, 7) = 7$
$v_3$	$\text{minimum}(4, 7, 16) = 4$
$v_4$	$\text{minimum}(11, 9, 2) = 2$
$v_5$	$\text{minimum}(18, 17, 4) = 4$

为获得  $v_2$  的最小值，我们没有包含指向  $v_1$  的边，因为  $v_2$  不能返回  $v_1$ 。为获得其他顶点的最小值，我们没有包含指向  $v_2$  的边，因为我们已经位于  $v_2$  了。通过展开包含  $[1, 2]$  的节点，所得到的任意旅程的长度下限就是这些最小值之和，即

$$14+7+4+2+4=31$$

为进一步说明用于确定界限的方法，假定已经访问了图 6-5 中包含  $[1, 2, 3]$  的节点。我们已经确定  $v_2$  为第二个顶点， $v_3$  为第三个顶点。通过展开此节点所获得的任意旅程，其离开这些顶点的代价下限为：

$v_1$	14
$v_2$	7
$v_3$	$\text{minimum}(7, 16) = 7$
$v_4$	$\text{minimum}(11, 2) = 2$

$$v_5 \text{ minimum}(18, 4) = 4$$

为获得  $v_4$  和  $v_5$  的最小值，我们没有考虑指向  $v_2$  和  $v_3$  的边，因为我们已经到过这些顶点。通过展开包含  $[1, 2, 3]$  的节点，所能获得的任意旅程的长度下限为：

$$14+7+7+2+4=34$$

同样，对于展开状态空间树中任意节点所能获得的旅程，都可以获得其长度下限，并在最佳优先查找中使用这些下限。下面的例子说明了这一方法。在例子中我们不会进行任何实际计算。其完成过程如上所述。

**例 6.3** 给定图 6-4 中的图，并利用前面概述的界限考虑事项，带有分支定界修剪的最佳优先查找方法生成图 6-6 中的树。界限存储在一个非叶节点中，而旅程的长度存储在叶节点中。我们给出生成这棵树的步骤。我们将最佳解的值初始化为  $\infty$ （无限大），因为根节点处没有候选解。（候选解仅存在于状态空间树的叶节点处。）我们没有为此状态空间树中的叶子计算界限，因为编写此算法就是为了不展开叶节点。当步骤中说到一个节点时，是指存储在该节点的部分旅程。这一点与 0-1 背包问题中提及节点的目的不同。

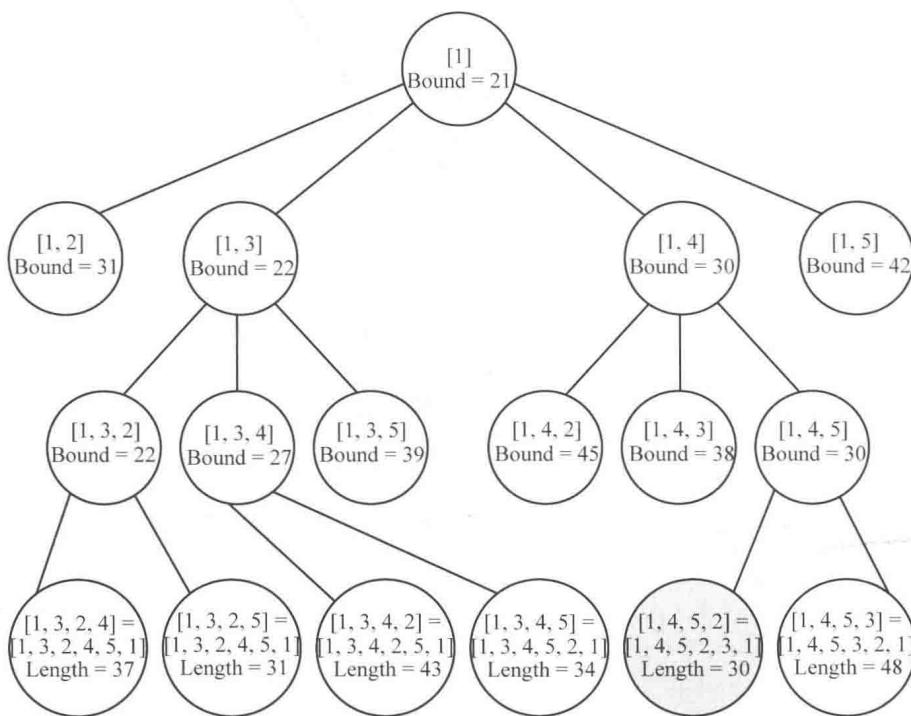


图 6-6 例 6.3 中使用带有分支定界修剪的最佳优先查找方法生成的已修剪状态空间树。在状态空间树的每个非叶节点处，部分旅程位于上方，通过展开该节点所能获得的旅程的长度界限在下方给出。在状态空间树的每个叶节点，旅程位于上方，其长度位于下方。阴影节点是找到最优旅程的节点

这些步骤如下。

- (1) 访问包含 [1] 的节点（根节点）。
  - (a) 计算界限为 21。（这是旅程的长度下限。）
  - (b) 设定 minlength 为  $\infty$ 。
- (2) 访问包含 [1, 2] 的节点。
  - (a) 计算界限为 31。
- (3) 访问包含 [1, 3] 的节点。

- (a) 计算界限为 22。
- (4) 访问包含[1,4]的节点。  
 (a) 计算界限为 30。
- (5) 访问包含[1,5]的节点。  
 (a) 计算界限为 42。
- (6) 确定具有最小界限的有希望、未展开节点。  
 (a) 此节点是包含[1,3]的节点。接下来访问其子节点。
- (7) 访问包含[1,3,2]的节点。  
 (a) 计算界限为 22。
- (8) 访问包含[1,3,4]的节点。  
 (a) 计算界限为 27。
- (9) 访问包含[1,3,5]的节点。  
 (a) 计算界限为 39。
- (10) 确定具有最小界限的有希望、未展开节点。  
 (a) 此节点是包含[1,3,2]的节点。接下来访问其子节点。
- (11) 访问包含[1,3,2,4]的节点。  
 (a) 因为这个节点是叶节点，计算出旅程长度为 37。  
 (b) 因为它的长度 37 小于  $\infty$ ，也就是 minlength 的值，所以将 minlength 设定为 37。  
 (c) 包含[1,5]和[1,3,5]的节点变成无希望的，因为它们的界限 42 和 39 大于或等于 37，也就是 minlength 的新值。
- (12) 访问包含[1,3,2,5]的节点。  
 (a) 因为这个节点是叶节点，计算出旅程长度为 31。  
 (b) 因为它的长度 31 小于 37，也就是 minlength 的值，所以设定 minlength 为 31。  
 (c) 包含[1,2]的节点变成没希望的，这是因为它的界限 31 大于或等于 31，也就是 minlength 的新值。
- (13) 确定具有最小界限的有希望、未展开节点。  
 (a) 此节点是包含[1,3,4]的节点。接下来访问其子节点。
- (14) 访问包含[1,3,4,2]的节点。  
 (a) 因为这个节点是叶节点，计算出旅程长度为 43。
- (15) 访问包含[1,3,4,5]的节点。  
 (a) 因为这个节点是叶节点，计算出旅程长度为 34。
- (16) 确定具有最小界限的有希望、未展开节点。  
 (a) 唯一有希望的未展开节点是包含[1,4]的节点。接下来访问其子节点。
- (17) 访问包含[1,4,2]的节点。  
 (a) 计算其界限为 45。  
 (b) 判定该节点是没有希望的，因为它的界限 45 大于或等于 31，也就是 minlength 的值。
- (18) 访问包含[1,4,3]的节点。  
 (a) 计算其界限为 38。  
 (b) 判定该节点是没有希望的，因为它的界限 38 大于或等于 31，也就是 minlength 的值。
- (19) 访问包含[1,4,5]的节点。  
 (a) 计算其界限为 30。
- (20) 确定具有最小界限的有希望、未展开节点。  
 (a) 唯一有希望的未展开节点是包含[1,4,5]的节点。接下来访问其子节点。

(21) 访问包含[1,4,5,2]的节点。

(a) 因为这个节点是叶节点，计算出旅程长度为30。

(b) 因为它的长度30小于31，也就是minlength的值，所以设定其minlength为30。

(22) 访问包含[1,4,5,3]的节点。

(a) 因为这个节点是叶节点，计算出旅程长度为48。

(23) 确定具有最小界限的有希望、未展开节点。

(a) 没有其他有希望的未展开节点了。任务结束。

我们已经判定包含[1,4,5,2]的节点包含了一个最优旅程（此节点表示旅程[1,4,5,2,3,1]，最优旅程的长度为30）。

图6-6的树中有17个节点，而整个状态空间树中的节点数为 $1+4+4\times3+4\times3\times2=41$ 。

在实现上例所用策略的算法中，将使用以下数据类型：

```
struct node
{
    int level;           // node 在树中的级别
    ordered_set path;
    number bound;
};
```

字段path包含该节点中存储的部分旅程。例如，在图6-6中，根节点最左侧子节点的path值为[1,2]。算法如下。

### 算法6.3 旅行推销员问题的带有分支定界修剪的最佳优先查找算法

问题：确定一个加权有向图中的最优旅程。权重为非负整数。

输入：一个加权有向图，以及图中的顶点数n。该图用一个二维数组W表示，它的行列索引范围均为1至n，其中 $W[i][j]$ 是从第i个顶点到第j个顶点的边上的权重。

输出：变量minlength，它的值是一条最优旅程的长度；变量opttour，它的值是一条最优旅程。

```
void travel2(int n,
            const number W[][],          // 表示加权有向图
            ordered_set& opttour,         // 最优旅程
            number& minlength)           // 最短旅程长度
{
    priority_queue<node> PQ;      // 建立一个优先队列
    node u, v;                    // 定义两个顶点
    initialize(PQ);              // 将PQ初始化为empty。
    v.level = 0;                  // 第一个顶点成为起始顶点
    v.path = [1];                 // 使第一个顶点成为起始顶点
    v.bound = bound(v);
    minlength = infinity;
    insert(PQ, v);
    while (!empty(PQ)){
        remove(PQ, v);           // 从PQ中删除具有最佳界限的节点
        if (v.bound < minlength){
            u.level = v.level + 1; // 设定u为v的一个子节点
            for (int i = 2; i <= n && i != v.path.back(); i++) {
                u.path = v.path;
                push(i, u.path);    // 将i放在u.path的末端
                if (u.level == n - 2){ // 检查下一个顶点是否完成一条旅程
                    minlength = length(u); // 将不在u.path中的唯一顶点的索引放在u.path的末尾
                    if (length(u) < minlength){ // 使第一个顶点放在最后一个
                        minlength = length(u);
                        opttour = u.path;
                    }
                }
            }
        }
    }
}
```

在习题中将要求你编写函数 `length` 和 `bound`。函数 `length` 返回旅程 `u.path` 的长度，函数 `bound` 基于前面讨论的因素返回一个节点的界限。

一个问题不一定拥有独一无二的定界函数。例如，在旅行推销员问题中可以看出，每个顶点都必须恰好访问一次，然后使用邻接矩阵中各列取值的最小值，而不是各行取值的最小值。或者，注意到每个顶点都必须恰好进入一次、退出一次，所以可以同时利用行与列。对于一条给定边，可以将它的一半权重与它离开的顶点相关联，将另一半与它进入的顶点相关联。于是，访问一个顶点的成本就是进入它、退出它的相关权重之和。例如，假定我们要为一条旅程的长度确定初始界限。取第二列各值中最小值的  $1/2$ ，可得到进入  $v_2$  的最小代价。取第二行各值中最小值的  $1/2$ ，可获得退出  $v_2$  的最小代价。于是，访问  $v_2$  的最小代价为：

$$\frac{\text{minimum}(14, 5, 7, 7) + \text{minimum}(14, 7, 8, 7)}{2} = 6$$

利用这一定界函数，分支定界算法在例 6.3 的实例中仅检查 15 个顶点。

当有两个或多个定界函数可用时，一个定界函数可能在一个节点处给出一个更好的界限，而另一个函数在另一个节点处给出更好的界限。事实上，旅行推销员问题中的定界函数就是这样，习题中将要求你验证这一点。当确实如此时，算法可以使用所有可用定界函数计算这些界限，然后使用最佳界限。但是，第 5 章曾经讨论过，我们的目的不是尽可能少访问节点，而是要使算法的整体效率最高。使用多个定界函数时增加的计算量，可能无法抵消通过减少访问节点所节省的计算量。

回想一下，分支定界算法可能会高效地解决一个大型实例，但对另一个大型实例来说，检查的节点数可能是指数级的（甚至更差）。回到 Nancy 的难题，如果连分支定界算法也无法高效地解决她的 40 城市实例，她该怎么做呢？处理诸如旅行推销员问题还有另外一种方法，就是开发近似算法。近似算法（approximation algorithm）不一定能给出最优解，而是给出相当接近最优的解。9.5 节将讨论这些算法，届时会再次讨论旅行推销员问题。

### ⊕6.3 漩因推理（诊断）

本节需要离散概率论和贝叶斯定理方面的知识。

人工智能和专家系统中的一个重要问题是为某些调查结果给出最可能的解释。例如，在医学上，我们希望在给定一组症状时确定最可能的疾病。在电子电路中，希望为电路中某一点发生的故障找出最可能的解释。另一个例子是为汽车故障找出最可能的原因。这种为一组调查结果确定最可能解释的过程称为溯因推理(abductive inference)。

为专注起见，这里的讨论采用医学术语。假定有  $n$  种疾病  $d_1, d_2, \dots, d_n$ ，每种疾病可能出现在一位病人身上。已知这位病人有个特定的症状集合  $S$ 。我们的目标是找出最可能出现的疾病集合。严格来说，可能会存在两个或更多个集合。但在讨论问题时，经常假定最可能存在一个独一无二的集合。

贝叶斯网络 (Bayesian network) 已经成为表示诸如疾病与症状之间概率关系的标准。信念网络 (belief network) 的讨论超出本书范围。Neapolitan (1990 年, 2003 年) 和 Pearl (1988 年) 的文献中对它们进行了详细讨论。对于许多贝叶斯网络应用, 都存在一些高效算法, 用于 (在发现任何症状之前) 确定病人仅患某一组特定疾病的先验概率。这些算法也在 Neapolitan (1990 年, 2003 年) 和 Pearl (1988 年) 的文献中进行了讨论。这里直接假定这些算法结果是可供我们使用的。例如, 这些算法可以给出病人仅患有  $d_1$ 、 $d_3$  和  $d_6$  三种疾病的先

验概率。我们将这一概率表示为：

$$p(d_1, d_3, d_6) \quad \text{和} \quad p(D)$$

其中，

$$D = \{d_1, d_3, d_6\}$$

这些算法还可以计算出，当已知出现  $S$  中的症状时，仅患有  $d_1$ 、 $d_3$  和  $d_6$  的概率。将这一条件概率称为：

$$p(d_1, d_3, d_6|S) \quad \text{和} \quad p(D|S)$$

假设我们可以计算这些概率（使用前面提到的算法），就可以利用类似于前面 0-1 背包问题的状态空间树，（在出现某些症状的情况下）解决确定最可能疾病集的问题。我们向根节点的左侧移动以包含  $d_1$ ，向右侧移动以排除它。同理，在第 1 级的一个节点向左移动，以包含  $d_2$ ，向右侧移动以排除它。以此类推。状态空间树中的每个叶节点表示一个可能解（也就是到该叶节点为止，已经包含的疾病集合）。为解决这一问题，我们在每个叶节点计算出现该疾病集合的条件概率，并判断哪个疾病集的条件概率最大。

为使用最佳优先查找进行修剪，需要找到一个定界函数。以下定义针对一大类实例给出了这一函数。

**定理 6.1** 如果  $D$  和  $D'$  是两个疾病集，满足

$$p(D') \leq p(D)$$

则

$$p(D'|S) \leq \frac{p(D)}{p(S)}$$

证明：根据贝叶斯定理，

$$\begin{aligned} p(D'|S) &= \frac{p(S|D')p(D')}{p(S)} \\ &\leq \frac{p(S|D)p(D)}{p(S)} \\ &\leq \frac{p(D)}{p(S)} \end{aligned}$$

第一个不等式是根据本定理中的假设得出的，第二个不等式是基于一个事实：任何概率都小于或等于 1。此定理得证。

对于一个给定节点，设  $D$  是到该节点已经包含在内的疾病集，对于该节点的某一后代，设  $D'$  是到该后代节点已经包含的疾病集，则  $D \subseteq D'$ 。做出以下假设通常是合理的：

$$\text{当 } D \subseteq D' \text{ 时, } p(D') \leq p(D)$$

其理由是，一个病人得一组病的概率不低于他得了这一组病再加上更多疾病的概率。（回想一下，它们就是在观察到任意症状之前的先验概率。）如果做出这一假设，则根据定理 6.1，

$$p(D'|S) \leq \frac{p(D)}{p(S)}$$

因此，在该节点的任意后代中，该疾病集的条件概率上限为  $p(D)/p(S)$ 。下例说明如何利用这一界限修剪分支。

**例 6.4** 假定有四种可能的疾病  $d_1, d_2, d_3, d_4$  和一组症状  $S$ 。本例的输入还包含一个贝叶斯网络，其中包含了疾病与症状之间的概率关系。本例中使用的概率是利用前述方法由这一贝叶斯网络计算得出的。在本书其他地方不再计算这些概率。我们指定任意概率，用以演示最佳优先查找算法。当在一个算法中（在本例中，是指最佳优先查找算法）使用来自另一算法（在本例中，是指信念网络中的推理算法）的结果时，可以直接在第一个算法中做出假设，说明第二种方法在什么位置提供其结果。认识到这一点是非常重要的。

图 6-7 是由最佳优先查找生成的已修剪状态空间树。树中已经为概率指定了任意值。在每个节点中，条件概率在上，界限在下。阴影节点是找出最佳解的位置。和在 6.1 节中一样，我们根据节点在树中的深度和从左侧算起的位置对节点进行标记。生成该树的步骤如下。变量 best 是当前最佳解，而  $p(\text{best}|S)$  是条件概率。我们的目标是计算出使这一条件概率取最大值的 best 值。任意假定：

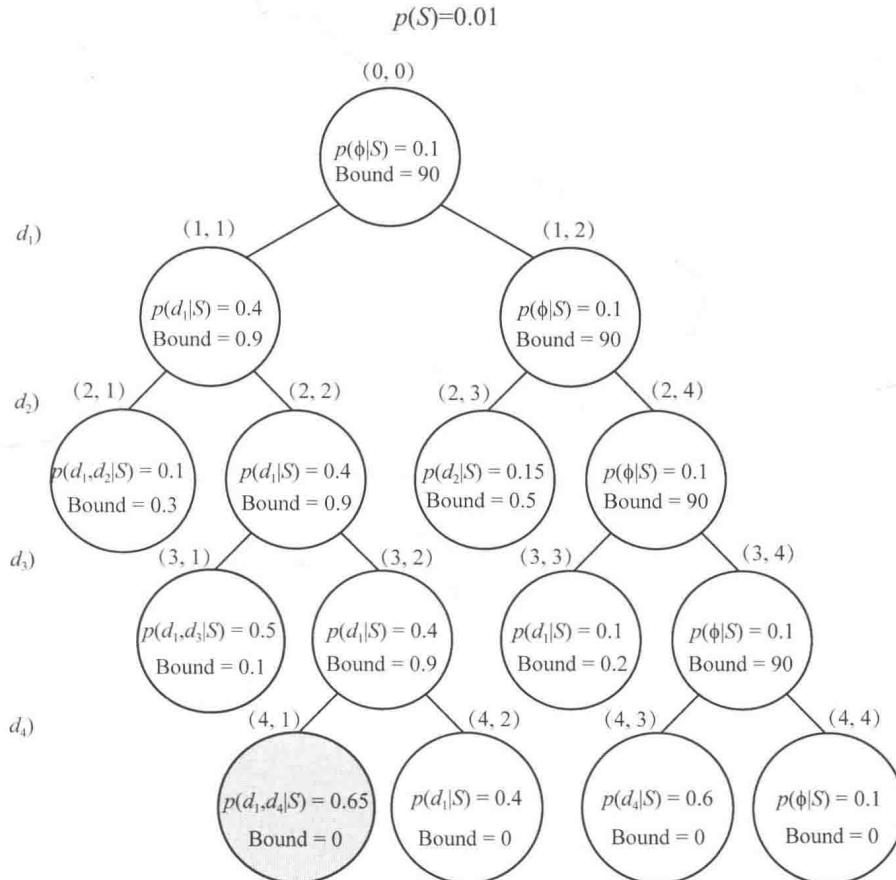


图 6-7 在例 6.4 中使用带有分支界限修剪的最佳优先查找生成的已修剪状态空间树。在每一节点中，上方是到该节点为止所包含疾病的条件概率，下方是通过展开该节点所能获得的条件概率界限。阴影节点是找到最优集合的节点

(1) 访问节点  $(0,0)$  (根节点)。

(a) 计算其条件概率。 $(\emptyset \text{ 为空集, 表示没有疾病。})$

$P(\emptyset|S)=0.1$  (这些计算由另一算法完成。我们分配任意值。)

(b) 设定

$$\text{best} = \emptyset \quad p(\text{best}|S) = p(\emptyset|S) = 0.1$$

(c) 计算其先验概率和界限。

$$p(\emptyset) = 0.9$$

$$\text{bound} = \frac{p(\emptyset)}{p(S)} = \frac{0.9}{0.01} = 90$$

(2) 访问节点  $(1,1)$ 。

(a) 计算其条件概率。

$$p(d_1|S)=0.4$$

(b) 因为  $0.4 > p(\text{best}|S)$ , 所以设定

$$\text{best} = \{d_1\} \quad \text{和} \quad p(\text{best}|S)=0.4$$

(c) 计算其先验概率和界限。

$$p(d_1)=0.009$$

$$\text{bound} = \frac{p(d_1)}{p(S)} = \frac{0.009}{0.01} = 0.9$$

(3) 访问节点(1,2)。

(a) 它的条件概率就是其父节点的条件概率, 即 0.1。

(b) 它的先验概率和界限就是其父节点的先验概率和界限, 即 0.9 和 90。

(4) 确定具有最大界限的有希望、未展开节点。

(a) 该节点为节点(1,2)。接下来访问其子节点。

(5) 访问节点(2,3)。

(a) 计算其条件概率。

$$p(d_2|S)=0.15$$

(b) 计算其先验概率和界限。

$$p(d_2)=0.005$$

$$\text{bound} = \frac{p(d_2)}{p(S)} = \frac{0.005}{0.01} = 0.5$$

(6) 访问节点(2,4)。

(a) 它的条件概率就是其父节点的条件概率, 即 0.1。

(b) 它的先验概率和界限就是其父节点的先验概率和界限, 即 0.9 和 90。

(7) 确定具有最大界限的有希望、未展开节点。

(a) 该节点为节点(4,4)。接下来访问其子节点。

(8) 访问节点(3,3)。

(a) 计算其条件概率。

$$p(d_3|S)=0.1$$

(b) 计算其先验概率和界限。

$$p(d_3)=0.002$$

$$\text{bound} = \frac{p(d_3)}{p(S)} = \frac{0.002}{0.01} = 0.2$$

(c) 判定它是没有希望的, 因为它的界限 0.2 小于或等于 0.4, 也就是  $p(\text{best}|S)$  的值。

(9) 访问节点(3,4)。

(a) 它的条件概率就是其父节点的条件概率, 即 0.1。

(b) 它的先验概率和界限就是其父节点的先验概率和界限, 即 0.9 和 90。

(10) 确定具有最大界限的有希望、未展开节点。

(a) 该节点为节点(3,4)。接下来访问其子节点。

(11) 访问节点(4,3)。

(a) 计算其条件概率。

$$p(d_4|S)=0.6$$

(b) 因为  $0.6 > p(\text{best}|S)$ , 设定

$$\text{best} = \{d_4\} \quad p(\text{best}|S)=0.6$$

- (c) 设定其界限为 0，因为它是状态空间树中的一个叶节点。
- (d) 此时，节点(2,3)变成无希望的，因为它的界限 0.5 小于或等于 0.6，也就是  $p(\text{best}|S)$  的新值。
- (12) 访问节点(4,4)。
- (a) 它的条件概率就是其父节点的条件概率，即 0.1。
- (b) 设定其界限为 0，因为它是状态空间树中的一个叶节点。
- (13) 确定具有最大界限的有希望、未展开节点。
- (a) 该节点为节点(1,1)。接下来访问其子节点。
- (14) 访问节点(2,1)。
- (a) 计算其条件概率。
- $$p(d_1, d_2|S)=0.1$$
- (b) 计算其先验概率和界限。
- $$p(d_1, d_2)=0.003$$
- $$\text{bound} = \frac{p(d_1, d_2)}{p(S)} = \frac{0.003}{0.01} = 0.3$$
- (c) 确定它是没有希望的，因为它的界限 0.3 小于或等于 0.6，也就是  $p(\text{best}|S)$  的值。
- (15) 访问节点(2,2)。
- (a) 它的条件概率就是其父节点的条件概率，即 0.4。
- (b) 它的先验概率和界限就是其父节点的先验概率和界限，即 0.009 和 0.9。
- (16) 确定具有最大界限的未展开、有希望节点。
- (a) 唯一的未展开、有希望节点是节点(2,2)。接下来访问其子节点。
- (17) 访问节点(3,1)。
- (a) 计算其条件概率。
- $$p(d_1, d_3|S)=0.05$$
- (b) 计算其先验概率和界限。
- $$p(d_1, d_3)=0.001$$
- $$\text{bound} = \frac{p(d_1, d_3)}{p(S)} = \frac{0.001}{0.01} = 0.1$$
- (c) 判断它是没有希望的，因为它的界限 0.1 小于或等于 0.6，也就是  $p(\text{best}|S)$  的值。
- (18) 访问节点(3,2)。
- (a) 它的条件概率就是其父节点的条件概率，即 0.4。
- (b) 它的先验概率和界限就是其父节点的先验概率和界限，即 0.009 和 0.9。
- (19) 确定具有最大界限的有希望、未展开节点。
- (a) 唯一的有希望、未展开节点是节点(3,2)。接下来访问其子节点。
- (20) 访问节点(4,1)。
- (a) 计算其条件概率。
- $$p(d_1, d_4|S)=0.65$$
- (b) 因为  $0.65 > p(\text{best}|S)$ ，所以设定
- $$\text{best} = \{d_1, d_4\} \quad p(\text{best}|S)=0.65$$
- (c) 设定其界限为 0，因为它是状态空间树中的一个叶节点。
- (21) 访问节点(4,2)。
- (a) 它的条件概率就是其父节点的条件概率，即 0.4。
- (b) 设定其界限为 0，因为它是状态空间树中的一个叶节点。
- (22) 确定具有最大界限的有希望、未展开节点。

(a) 不存在未展开的有希望节点。任务完成。

我们已经确定，最可能出现的疾病集为 $\{d_1, d_4\}$ ，且 $p(d_1, d_4|S)=0.65$ 。

此问题中的一种合理策略是：在开始时根据这些疾病的条件概率，将它们排列为非递减顺序。但这一策略并不保证使查找时间最短。我们没有在例 6.5 中执行这一操作，因而共检查了 15 个节点。在习题中，将要求你确认，如果这些疾病是有序的，将会检查 23 个节点。

接下来给出算法，它使用以下声明：

```
struct node
{
    int level;           // 节点在树中的级别。
    set_of_indices D;
    float bound;
};
```

字段 $D$ 中包含了到该节点为止所包含疾病的索引。这一算法的一个输入是贝叶斯网络 $BN$ 。前文曾经提到，贝叶斯网络表示疾病和症状之间的概率关系。本节开头提到的算法可以由这样一个网络计算出所需概率。

下面的算法由 Cooper 开发（1984 年）。

#### 算法 6.4 Cooper 为溯因推理开发的带有分支定界修剪的最佳优先查找算法

问题：给定一组症状，确定最可能出现的疾病集合（解释）。假定如果疾病集合 $D$ 是疾病集合 $D'$ 的一个子集，则 $p(D') \leq p(D)$ 。

输入：正整数 $n$ ；一个贝叶斯网络 $BN$ ，表示 $n$ 种疾病与其症状之间的概率关系；一个症状集合 $S$ 。

输出：一个集合 $best$ ，其中包含了（在 $S$ 条件下）最可能集合中疾病的索引；一个变量 $pbest$ ，它是在给定 $S$ 情况下， $best$ 的概率。

```
void cooper (int n,
             Bayesian_network_of_n_diseases BN,
             set_of_symptoms S,
             set_of_indices& best, float& pbest)
{
    priority_queue_of_node PQ;
    node u, v;

    v.level = 0;           // 设定 v 为根节点。
    v.D = ∅;              // 在根节点处存储空集。
    best = ∅;
    pbest = p(∅ | S);
    v.bound = bound(v);
    insert(PQ, v);

    while(!empty(PQ)){
        remove(PQ, v);      // 删除具有最佳界限的节点。
        if (v.bound > pbest){
            u.level = v.level + 1; // 设定 u 为 v 的子节点。
            u.D = v.D;          // 设定 u 为包含下一疾病的子节点。
            将 u.level 放入 u.D 中;
            if (p(u.D | S) > pbest){
                best = u.D;
                pbest = p(u.D | S);
            }
            u.bound = bound(u);
            if (u.bound > pbest)
                insert(PQ, u);
            u.D = v.D;          // 设定 u 为不包含下一疾病的子节点。
            u.bound = bound(u);
            if (u.bound > pbest)
                insert(PQ, u);
        }
    }
}
```

```

    }
}

int bound (node u)
{
    if(u.level == n)      // 叶节点是没有希望的。
        return 0;
    else
        return p(u.D|S);
}

```

符号  $p(D)$  表示  $D$  的先验概率,  $p(S)$  表示  $S$  的先验概率,  $p(D|S)$  表示给定  $S$  的情况下,  $D$  的条件概率。这些值可利用本节开始提到的算法, 由贝叶斯网络  $BN$  计算得出。

我们已经严格按照编写最佳优先查找算法的规则编写了此算法。还可能进行一处改进, 对于一个节点的右子节点, 不需要调用函数 `bound`。原因在于, 右子节点包含的疾病集与原节点相同, 也就是说, 它的界限相同。因此, 只有在左子节点处将 `pbest` 值改为大于或等于这一界限的值时, 才会修改右子节点。我们可以修改算法; 当发生上述情况时, 修剪右子节点; 在未发生上述情况时, 则展开至右子节点。

与本章介绍的其他问题一样, 溯因推理问题也属于第 9 章讨论的一类问题。

如果有多个解, 上述算法只给出其中一个。我们可以很轻松地修改算法, 使其给出所有最佳解。我们还可对其进行修改, 给出  $m$  个最可能的解释, 其中  $m$  是任意正整数。这一修改由 Neapolitan (1990 年) 进行讨论。Neapolitan (1990 年) 还详细讨论了此算法。

## 6.4 习题

### 6.1 节

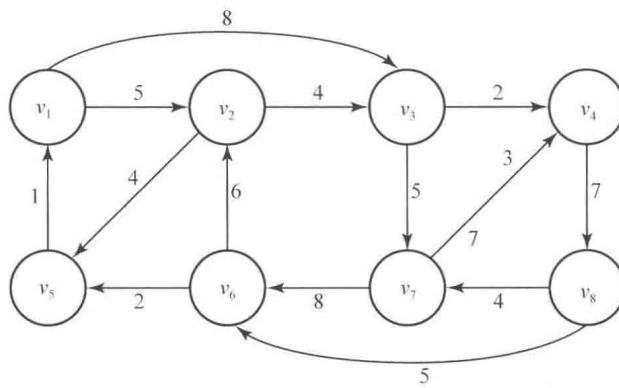
- 利用算法 6.1 (0-1 背包问题的带有分支定界修剪的宽度优先查找算法) 使以下问题实例的价值最大。给出逐步操作步骤。

$i$	$p_i$	$w_i$	$\frac{p_i}{w_i}$	
1	20 美元	2	10	
2	30 美元	5	6	
3	35 美元	7	5	
4	12 美元	3	4	
5	3 美元	1	3	$W=13$

- 在你的系统上实现算法 6.1, 并针对第 1 题的问题实例运行它。
- 修改算法 6.1, 以生成一组最优物品。对比此算法与算法 6.1 的性能。
- 利用算法 6.2 (0-1 背包问题的带有分支定界修剪的最佳查找算法), 使第 1 题问题实例的代价最大。给出逐步操作步骤。
- 在你的系统上实现算法 6.2, 并针对第 1 题的问题实例运行它。
- 对于该问题的大型实例, 比较算法 6.1 与算法 6.2 的性能。

### 6.2 节

- 使用算法 6.3 (旅行推销员问题的带有分支定界修剪的最佳优先查找算法), 针对下图找出一种最优旅程, 并给出该最优旅程的长度。给出逐步操作步骤。



8. 一个图的邻接矩阵由以下数组给出，使用算法 6.3 为该图找出一个最优旅程。给出逐步操作步骤。

	1	2	3	4	5
1	0	6	6	10	8
2	3	0	12	7	6
3	8	7	0	14	20
4	5	13	9	0	8
5	9	8	10	6	0

9. 编写算法 6.3 中使用的函数 length 和 bound。
10. 考虑旅行推销员问题。
- (a) 为这个问题编写一个考虑所有可能旅程的暴力算法。
  - (b) 实现该算法，并用它解决规模为 6、7、8、9、10、15 和 20 的实例。
  - (c) 使用(b)中开发的实例，比较这一算法与算法 6.3 的性能。
11. 在你的系统上实现算法 6.3，并针对第 7 题的问题实例运行它。使用不同定界函数，并研究结果。
12. 使用旅行推销员问题的大型实例，比较动态规划算法（见 3.6 节，习题 27）与算法 6.3 的性能。
- ### 6.3 节
13. 修改算法 6.4 (Cooper 为溯因推理开发的带有分支定界修剪的最佳优先查找算法)，生成  $m$  种最可能的解释，其中  $m$  为任意正整数。
14. 证明：如果例 6.4 中的疾病根据它们的条件概率进行非递减顺序排列，所检查的节点数将是 23，而不是 15。  
假设  $p(d_4)=0.08$ ,  $p(d_4, d_1)=0.007$ 。
15. 如果一组解释的概率大于或等于一个合适的度量  $p$ ，则称这组解释满足该度量。修改算法 6.4，给出一组满足  $p$  的解释，其中  $0 \leq p \leq 1$ 。用尽可能少的解释做到这一点。
16. 在你的系统上实现算法 6.4。用户应当能够像习题 13 一样输入一个整数  $m$ ，或者像习题 15 一样输入一个合适的度量  $p$ 。
17. 分支定界设计策竟能否用于解释第 3 章习题 34 讨论的问题？给出理由。
18. 为 4.3.2 节讨论的带有最终期限的调度安排问题编写分支定界算法。
19. 分支定界设计策竟能否用于解释第 4 章习题 26 讨论的问题？给出理由。
20. 分支定界设计策竟能否用于求解 3.4 节讨论的链式矩阵乘法？给出理由。
21. 再给出分支定界设计策略的三种应用。

除非我找到一个 $\theta(n)$ 排序算法，否则我不会放弃。

# 第 7 章

## 计算复杂度介绍：排序问题



1.1 节给出了一种二次时间的排序算法（交换排序）。如果计算机科学家满足于这种排序算法，那当今许多应用的运行速度要慢得多，而另外一些应用根本就不会出现。回忆一下表 1-4，使用二次时间算法对 10 亿个键进行排序需要花费数年的时间。现在已经开发了更为高效的排序算法。具体来说，2.2 节介绍了合并排序，最差情况为  $\Theta(n \lg n)$ 。尽管这一算法的速度并不是非常快，还不能在瞬间完成 10 亿项的排序，但表 1-4 表明，在离线应用中，它还是可以在可忍受的时间内完成这些项目的排序。假定有人希望在联机应用中几乎瞬时完成 10 亿个项目的排序，他可能花费数小时甚至许多年的时间来开发一种线性时间或更好的排序算法。会不会有这样一种可能：他在穷尽毕生之力之后，才知道这种算法根本就不可能存在？解决一个问题有两种方法。一种方法是尝试为这一问题开发一种更高效的算法，另一种方法是尝试证明不可能存在一种更高效的算法。一旦证明了这一点，就知道不应当再追求获得更快速的算法了。后面将会看到，对于一大类排序算法，已经证明了不可能存在优于  $\Theta(n \lg n)$  的算法。

### 7.1 计算复杂度

前几章关心的是为各种问题设计和分析算法。我们经常使用不同的方法求解同一问题，希望为这种问题找出更高效的算法。在分析一种特定算法时，我们计算它的时间（或内存）复杂度，或其时间复杂度（或内存复杂度）的阶，并没有分析这一算法所解决的问题。例如，在分析算法 1.4（矩阵乘法）时，发现它的时间复杂度为  $n^3$ 。但是，这并不是说这个矩阵乘法问题需要一个  $\Theta(n^3)$  算法。函数  $n^3$  是算法的一个性质，但它不一定是矩阵乘法问题的性质。2.5 节设计了时间复杂度为  $\Theta(n^{2.38})$  的 Strassen 矩阵乘法算法。此外还提到，已经为该算法设计了一种  $\Theta(n^{2.38})$  的变体。有一个重要的问题是，是否可能找到一种更为高效的算法。

计算复杂度（computational complexity）是与算法设计分析并存的一个研究领域，研究可以解决一种给定问题的所有可能算法。计算复杂度分析尝试为一个给定问题的所有算法确定一个效率下限。在 2.5 节的最后曾经提到，已经证明了矩阵乘法问题需要一种时间复杂度为  $\Omega(n^2)$  的算法。这一结论就是通过计算复杂度分析得到的。这一结果可以表述为：矩阵乘法问题的下限（lower bound）为  $\Omega(n^2)$ 。这并不是说，一定可以为矩阵乘法开发出一种  $\Theta(n^2)$  算法。它只是说，不可能为它开发出一种优于  $\Theta(n^2)$  的算法。因为目前的最好算法是  $\Theta(n^{2.38})$ ，而下限为  $\Omega(n^2)$ ，所以值得继续研究该算法。这一研究可以沿两个方向进行。一个方向是利用算法设计方法，找出一种更高效的算法；另一个方向是利用计算复杂度分析，获得一个更大的下限。也许有一天会设计出一种优于  $\Theta(n^{2.38})$  的算法，也许有一天我们能够证明，存在一个大于  $\Omega(n^2)$  的下限。一般来说，对于一个给定问题，我们的目标是确定其下限  $\Omega(f(n))$ ，并为该问题开发一种  $\Theta(f(n))$  算法。一旦做到了这一点，我们就知道，除了提高常量之外，不可能再对该算法进行其他任何改进了。

一些作者使用“计算复杂度分析”时，同时包含了算法分析和问题分析。本书在提到计算复杂度分析时，仅表示问题分析。

我们通过研究排序问题来介绍计算复杂度分析。选择这一问题有两个理由。第一，已经为解决这一问题设计了相当多的算法。通过研究和对比这些算法，可以深入理解如何在同一问题的几种算法中做出选择，以及如何改进某一给定算法。第二，现在只有几种问题，人们为其开发的算法的时间复杂度几乎达到了其下限，而排序问题就是其中之一。也就是说，对于一大类排序算法，人们已经确定了其下限  $\Omega(n \lg n)$ ，而且还设计了  $\Omega(n \lg n)$  算法。因此可以说，就这一类算法来说，我们已经解决了排序问题。

前面所说的这类排序算法，也就是其性能几乎达到下限的这类算法，包含了所有仅通过键的比较来进行排序的算法。在第1章的开头已经讨论过，使用“键”一词，是因为每条记录都得到一个独一无二的标识符，称为键（key），它是一个有序集中的一个元素。如果这些记录排列为任意序列，排序任务（sorting task）就是调整这些记录的顺序，使它们根据键的取值排序。在我们的算法中，这些键存储在一个数组中，而且不谈及非键字段，但假定这些字段会随这些键一起重新排列。仅通过键的比较进行排序的算法可以对比两个键，判断哪个更大一些，可以复制键，但不能对它们进行其他操作。到目前为止，我们遇到的排序算法（算法1.3、算法2.4和算法2.6）都属于这一类。

7.2节至7.8节讨论了仅通过键的比较进行排序的算法。具体来说，7.2节讨论了插入排序和选择排序，这是两种最高效的二次时间排序算法。7.3节证明了，只要局限于插入排序和选择排序，就不可能在二次时间的基础上再做改进。7.4节和7.5节回顾了 $\Theta(n\lg n)$ 排序算法——合并排序和快速排序。7.6节给出了另一种 $\Theta(n\lg n)$ 排序算法——堆排序。7.7节比较三种 $\Theta(n\lg n)$ 排序算法。7.8节证明了，对于通过键的比较来进行排序的算法，其下限就是 $\Omega(n\lg n)$ 。7.9节讨论了基排序，这种排序算法不是通过键的比较来完成排序的。

我们从键的比较次数和记录的赋值次数两个方面来分析算法。例如，在算法1.3（交换排序）中， $S[i]$ 和 $S[j]$ 的交换可实现如下：

```
temp = S[i];
S[i] = S[j];
S[j] = temp;
```

这意味着完成一次交换要进行三次记录赋值。我们要分析记录的赋值次数，是因为当记录很大时，为记录赋值所花费的时间可能会很长。我们还会分析除了存储输入内容的空间之外，这些算法还需要多少额外空间。当额外空间为常量时（也就是说，在待排序的键数 $n$ 增大时，此空间不变），就说此算法为原地排序（in-place sort）。最后，我们假定是在进行非递减顺序排列。

## 7.2 插入排序和选择排序

插入排序（insertion sort）算法是将记录插入已有的有序数组来完成排序。下面是简单插入排序的一个例子。假定前 $i-1$ 个数组位置的键是有序的。设 $x$ 是第 $i$ 个位置中的键的取值。依次将 $x$ 与第 $i-1$ 、 $i-2$ 等位置的键进行对比，直到找出一个小于 $x$ 的键。设 $j$ 是这个键所在的位置。将第 $j+1$ 至 $i-1$ 位置的键移至第 $j+2$ 至 $i$ 个位置，并将 $x$ 插入第 $j+1$ 个位置。针对 $i=2$ 至 $i=n$ 重复这一过程。图7-1演示了这一排序过程。插入排序的一个算法如下：

### 算法7.1 插入排序

**问题：**将 $n$ 个键排列为非递减顺序。

**输入：**正整数 $n$ ；键的数组 $S$ ，其索引范围为1至 $n$ 。

**输出：**数组 $S$ ，其中包含了非递减顺序的键。

```
void insertionsort (int n, keytype S[])
{
    index i, j;
    keytype x;
    for (i = 2; i <= n; i++){
        x = S[i];
        j = i - 1;
        while (j > 0 && S[j]>x){
            S[j + 1] = S[j];
            j--;
        }
        S[j + 1] = x;
    }
}
```

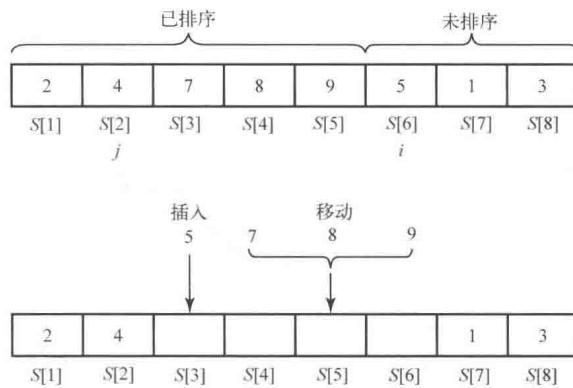


图 7-1 一个示例，说明当  $i=6$  和  $j=2$  时插入排序所做的工作。  
上图为插入之前的数组，下图为插入步骤

### 算法 7.1 的分析 对键的比较次数进行最差情况时间复杂度分析（插入排序）

基本运算： $S[j]$  与  $x$  的比较。

输入规模： $n$ ，待排序的键的个数。

对于一个给定  $i$  值， $S[j]$  与  $x$  的比较多在退出 `while` 循环时进行，因为  $j$  变为等于 0。假定在一个`&&`表达式中，如果第一个条件为假，就不再评估第二个条件，那么当  $j$  为 0 时，不会再将  $S[j]$  与  $x$  进行比较。因此，对于一个给定  $i$  值，最多进行  $i-1$  次此种比较。因为  $i$  的取值范围为 2 至  $n$ ，所以比较总数最多为：

$$\sum_{i=2}^n (i-1) = \frac{n(n-1)}{2}$$

以下证明留作练习：如果数组中的键最初是非递减顺序，则可以实现这一下限。因此，

$$W(n) = \frac{n(n-1)}{2}$$

### 算法 7.1 的分析 对键的比较次数进行的平均情况时间复杂度分析（插入排序）

对于一个给定  $i$  值，共有  $i$  个位置可以插入  $x$ 。也就是说， $x$  可以留在第  $i$  个位置，进入第  $i-1$  个位置，进入第  $i-2$  个位置，等等。因为之前没有在这一算法中研究或使用过  $x$ ，所以没有理由认为它出现在某个位置的可能性要高于其他位置。因此，我们为前  $i$  个位置指定相同的概率。也就是说，每个位置的概率均为  $1/i$ 。下表列出了在将  $x$  插入每个位置时所进行的比较次数。

位置	比较次数
$i$	1
$i-1$	2
$\vdots$	
2	$i-1$
1	$i-1$

将  $x$  插入第一个位置时的比较次数是  $i-1$ ，而不是  $i$ ，原因在于当  $j=0$  时，`while` 循环控制表达式中的第一个条件为假，也就是说不再需要评估第二个条件。对于一个给定  $i$  值，为插入  $x$  而进行的平均比较次数为：

$$\begin{aligned} 1\left(\frac{1}{i}\right) + 2\left(\frac{1}{i}\right) + \cdots + (i-1)\left(\frac{1}{i}\right) + (i-1)\left(\frac{1}{i}\right) &= \frac{1}{i} \sum_{k=1}^{i-1} k + \frac{i-1}{i} \\ &= \frac{(i-1)i}{2i} + \frac{i-1}{i} \\ &= \frac{i+1}{2} - \frac{1}{i} \end{aligned}$$

因此，对这一数组进行排序所需要的平均比较次数为：

$$\sum_{i=2}^n \left( \frac{i+1}{2} - \frac{1}{i} \right) = \sum_{i=2}^n \frac{i+1}{2} - \sum_{i=2}^n \frac{1}{i} \approx \frac{(n+4)(n-1)}{4} - \ln n$$

最后一个等式是利用附录 A 中例 A.1 和例 A.9 的结果，并进行了一点代数运算得到的。我们已经证明了：

$$A(n) \approx \frac{(n+4)(n-1)}{4} - \ln n \approx \frac{n^2}{4}$$

接下来分析额外空间的使用。

#### 算法 7.1 的分析 额外空间使用量的分析（插入排序）

唯一随  $n$  的增大而增大的空间使用量就是输入数组  $S$  的大小。因此，此算法是一种原地排序算法，额外空间属于  $\Theta(1)$ 。

在习题中会要求你证明：插入排序完成的记录赋值数的最差情况与平均情况时间复杂度为：

$$W(n) = \frac{(n+4)(n-1)}{2} \approx \frac{n^2}{2} \quad A(n) = \frac{n(n+7)}{4} - 1 \approx \frac{n^2}{4}$$

接下来将插入排序与本书中讨论的另一种二次时间算法进行比较，也就是交换排序（算法 1.3）。回想一下，在交换排序中，键的比较次数的所有情况时间复杂度为：

$$T(n) = \frac{n(n-1)}{2}$$

在习题中会要求你证明：交换排序完成的记录赋值数的最差情况与平均情况时间复杂度为：

$$W(n) = \frac{3n(n-1)}{2} \quad A(n) = \frac{3n(n-1)}{4}$$

显然，交换排序是一种原地排序。

表 7-1 总结了有关交换排序和插入排序的结果。由这个表格可以看出，就键的比较次数而言，插入排序的性能总是不低于交换排序，而在平均情况下，是要更好一些。就记录的赋值而言，插入排序的性能在最差情况下和平均情况下都要更好一些。因为它们都是原地排序，所以插入排序算法要更好一些。注意，表 7-1 中还包含了另一种算法——选择排序。这种算法对交换排序稍做了一点修改，消除了交换排序的一个缺点。下面就来介绍它。

表 7-1 交换排序、插入排序和选择排序\*的分析小结

算 法	键的比较	记录的赋值	额外空间使用量
交换排序	$T(n) = \frac{n^2}{2}$	$W(n) = \frac{3n^2}{2}$ $A(n) = \frac{3n^2}{4}$	原地
插入排序	$W(n) = \frac{n^2}{2}$ $A(n) = \frac{n^2}{4}$	$W(n) = \frac{n^2}{2}$ $A(n) = \frac{n^2}{4}$	原地
选择排序	$T(n) = \frac{n^2}{2}$	$T(n) = 3n$	原地

\*各项是近似的。

### 算法 7.2 选择排序

问题：将  $n$  个键排列为非递减顺序。

输入：正整数  $n$ ；键的数组  $S$ ，索引范围为 1 至  $n$ 。

输出：数组  $S$ ，包含非递减顺序的键。

```
void selectionsort (int n, keytype S[])
{
    index i, j, smallest;
    for (i = 1; i <= n - 1; i++){
        smallest = i;
        for (j = i + 1; j <= n; j++)
            if (S[j] < S[smallest])
                smallest=j;
        交换 S[i]与 S[smallest];
    }
}
```

显然，就键的比较次数而言，此算法的时间复杂度与交换排序一样。但是，记录的赋值就有很大不同了。在交换排序中，每次发现  $S[j]$  小于  $S[i]$  时都会交换  $S[i]$  和  $S[j]$ （见算法 1.3），而选择排序只是跟踪第  $i$  至  $n$  个位置中当前最小键的索引。在确定该记录之后，将它与第  $i$  个位置中的记录交换。这样，在第一遍执行 `for-i` 循环后，最小的键被放在第一个位置，在第二遍之后，第二小的键被放在第二个位置，以此类推。结果与交换排序相同。但是，由于在 `for-i` 循环的底部仅进行一次交换，从而使交换次数恰好为  $n-1$ 。因为这一交换需要三次赋值，所以选择排序所执行的记录赋值数的所有情况时间复杂度为：

$$T(n)=3(n-1)$$

回想一下，交换排序在平均情况下的记录赋值数大约为  $3n^2/4$ 。因此，在平均情况下，已经用线性时间代替了二次时间。交换排序的性能有时会优于选择排序。例如，如果这些记录已经是有序的，交换排序不会进行记录赋值。

选择排序与插入排序相比，又是如何呢？再来看表 7-1。就键的比较次数来说，插入排序的性能总是不差于选择排序，在平均情况下，其性能要更好一些。但是，就记录的赋值次数来说，选择排序的时间复杂度是线性的，而插入排序则是二次时间。回想一下，当  $n$  很大时，线性时间要远快于二次时间。因此，如果  $n$  很大，而且记录很大（所以记录赋值的时间很长），那选择排序的性能应当更好一些。

任何一种依次选择记录并将它们放在正确位置的排序算法都称为选择排序（selection sort）。这意味着交换排序也是一种选择排序。7.6 节给出了另外一种选择排序——堆排序，但算法 7.2 已经占有了“选择排序”的名字。

对比交换排序、插入排序和选择排序的目的是为了尽可能简单地引入对排序算法的全面对比。在实践中，这些算法对于极大型实例都是不可行的，因为它们在平均情况下和最差情况下都是二次时间的。接下来将证明，只要我们选择的算法与这三种算法属于同一类别，那就不可能通过改进使其键的比较次数优于二次时间。

## 7.3 每次比较最多减少一个倒置的算法的下限

在每次比较之后，插入排序要么什么也不做，要么就是将第  $j$  个位置的键移到第  $j+1$  个位置。通过将第  $j$  个位置的键上升一个位置，我们对“ $x$  应当出现在该键之前”这一事实做出了补救。但我们已经完成的全部工作也就是这些了。我们将证明，所有仅通过键的比较进行排序，而且在每次比较中仅执行有限次重排的排序算法都至少需要二次时间。这些结果的获取基于如下假设：要排序的键是互不相同的。显然，即使撤销这一限制，最差情况下的下限仍然成立，这是因为，对于一部分输入获得的最差性能下限，也是考虑所有输入时的最差性能下限。

一般情况下，我们考虑对  $n$  个互不相同、来自任意有序集合的键进行排序。但是，不失一般性，可以假定

要排序的键就是正整数  $1, 2, \dots, n$ , 因为可以用 1 替代最小键, 用 2 替代第二小的键, 以此类推。例如, 假定有字母输入[Ralph, Clyde, Dave], 可以将 1 与 Clyde 相关联, 将 2 与 Dave 相关联, 将 3 与 Ralph 相关联, 从而获得等价输入[3, 1, 2]。任何仅通过键的比较来对这些整数进行排序的算法, 其执行的比较次数必然与对这三个名字进行排序的比较次数相同。

前  $n$  个正整数的排列 (permutation) 可以看作这些整数的一种排序。因为前  $n$  个正整数共有  $n!$  种排列 (见 A.7 节), 所以这些整数有  $n!$  种不同排序。例如, 下面六种排列是前三个正整数的所有排序:

$$[1, 2, 3] \quad [1, 3, 2] \quad [2, 1, 3] \quad [2, 3, 1] \quad [3, 1, 2] \quad [3, 2, 1]$$

这就是说, 在考虑  $n$  个互不相同的键时, (一种排序算法) 会有  $n!$  种不同输入。这六个排序就是规模为 3 时的不同输入。

一个排列可表示为  $[k_1, k_2, \dots, k_n]$ 。也就是说,  $k_i$  是第  $i$  个位置的整数。例如, 对于排列 [3, 1, 2],

$$k_1 = 3, k_2 = 1, k_3 = 2$$

排列中的一个倒置 (inversion) 是指下面这样一对整数:

$$(k_i, k_j), \text{ 满足 } i < j, \text{ 且 } k_i > k_j$$

例如, 排列 [3, 2, 4, 1, 6, 5] 包含了以下倒置: (3, 2), (3, 1), (2, 1), (4, 1) 和 (6, 5)。显然, 当且仅当一个排列的顺序为  $[1, 2, \dots, n]$  时, 其中不包含倒置。这就是说, 要对  $n$  个互不相同的键进行排序, 就是要删除一个排列中的所有倒置。下面给出本节的主要结果。

**定理 7.1** 任何一种算法, 如果仅通过键的比较来对  $n$  个互不相同的键进行排序, 而且每次比较之后最多仅删除一个倒置, 那它在最差情况下执行的键比较次数最少为:

$$\frac{n(n-1)}{2}$$

平均情况下执行的键比较次数最少为:

$$\frac{n(n-1)}{4}$$

**证明:** 要导出最差情况下的结果, 只需要证明存在一个具有  $n(n-1)/2$  个倒置的排序, 因为当该排列为输入时, 该算法必须删除这么多倒置, 从而至少进行这么多比较。 $[n, n-1, \dots, 2, 1]$  就是这样一个排列, 其证明留作练习。

为导出平均情况下的结果, 我们将排列  $[k_n, k_{n-1}, \dots, k_1]$  与排列  $[k_1, k_2, \dots, k_n]$  配对。这一排列称为原排序的转置 (transpose)。例如, [3, 2, 4, 1, 5] 是 [5, 1, 4, 2, 3] 的转置。不难看如, 如果  $n > 1$ , 则每个排列都有唯一不同于原排列的转置。设

$$r \text{ 和 } s \text{ 是介于 } 1 \text{ 到 } n \text{ 之间的整数, 且 } s > r$$

给定一个排列, 整数对  $(s, r)$  或者是该排列中的一个倒置, 或者是其转置中的一个倒置, 但不可能同时是这两者中的倒置。对于 1 到  $n$  之间的整数, 共存在  $n(n-1)/2$  个这样的整数对, 其证明留作习题。这就是说, 一个排列与其转置之间恰好有  $n(n-1)/2$  个倒置。因此, 一个排列与其转置中的平均倒置个数是:

$$\frac{1}{2} \times \frac{n(n-1)}{2} = \frac{n(n-1)}{4}$$

因此, 如果作为输入的所有排列都是等概率的, 则输入中的平均倒置数也是  $n(n-1)/4$ 。因为我们假设该算法在每次比较之后最多只删除一个倒置, 所以平均来说, 它至少要执行这么多次比较, 才能删除所有倒置, 从而完成输入排序。

插入排序在每次比较之后最多删除由  $S[j]$  和  $x$  组成的倒置, 因此这一算法也属于定理 7.1 讨论的一类算法。交换排序和选择排序也属于这类算法, 要看出这一点略微困难一些。为说明它们确属此类算法, 下面给出一个使用交换排序的例子。首先回想一下, 交换排序的算法如下:

```

void exchangesort (int n, keytype S[])
{
    index i, j;

    for (i = 1; i <= n - 1; i++)
        for (j = i + 1; j <= n; j++)
            if (S[j] < S[i])
                交换 S[i]与 S[j];
}

```

假定数组  $S$  中当前包含排列  $[2, 4, 3, 1]$ ，而且我们正在将 2 与 1 进行比较。在此次比较之后，2 和 1 将被交换，从而删除倒置  $(2, 1)$ 、 $(4, 1)$  和  $(3, 1)$ 。但是，也添加了倒置  $(4, 2)$  和  $(3, 2)$ ，倒置数目的净减少量仅为 1。这个例子说明了一个一般结果：在每次比较之后，交换排序算法使倒置的净减少量最多为 1。

由于插入排序在最差情况下的时间复杂度为  $n(n-1)/2$ ，且平均情况下的时间复杂度大约为  $n^2/4$ ，它的性能（就键的比较次数而言）与仅通过键的对比排序且每次比较之后最多只删除一个倒置的算法大体相当。回想一下，合并排序（算法 2.2 和算法 2.4）与快速排序（算法 2.6）的时间复杂度要优于这一结果。现在让我们重新研究这些算法，看看它们如何不同于诸如插入排序之类的算法。

## 7.4 再谈合并排序

合并排序已经在 2.2 节介绍过。现在将证明，它有时会在一次比较之后消除多个倒置。然后将说明可以如何对其进行改进。

在定理 7.1 的证明中曾经提到，当输入为逆序时，在每次比较之后最多仅消除一个倒置的算法至少要执行  $n(n-1)/2$  次比较。图 7-2 演示了合并排序 2（算法 2.4）是如何处理这种输入的。在合并子数组  $[3, 4]$  和  $[1, 2]$  时，这些比较将消除多个倒置。在比较 3 和 1 之后，1 被放在第一个数组位置，从而消除了倒置  $(3, 1)$  和  $(4, 1)$ 。

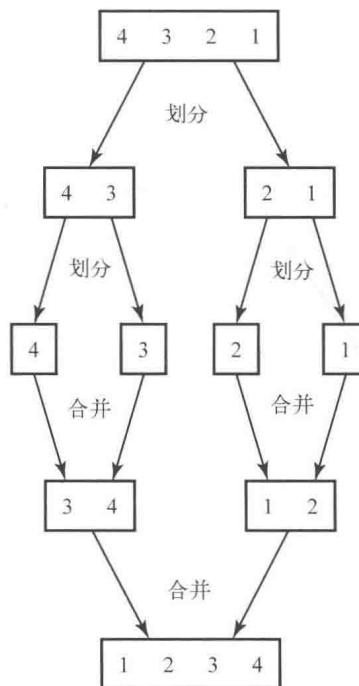


图 7-2 合并排序算法对逆序输入进行排序

在比较了 3 和 2 之后，2 被放在第二个数组，从而消除了倒置  $(3, 2)$  和  $(4, 2)$ 。

回想一下，就键比较次数而言，合并排序执行的最差情况时间复杂度为：

$$W(n) = n \lg n - (n-1)$$

其中， $n$  是 2 的幂，一般情况下，它属于  $\Theta(n \lg n)$ 。

我们已经颇有收益：设计的这种排序算法，有时可以在一次比较后消除多个倒置。回想一下 1.4.1 节的内容， $\Theta(n \lg n)$  算法可以处理非常大的输入，而二次时间算法则不能。

利用“生成函数”方法来求解递归，可以证明，当  $n$  为 2 的幂时，就键比较次数而言，合并排序执行的平均情况时间复杂度为：

$$A(n) = n \lg n - 2n \sum_{i=1}^{\lg n} \frac{1}{2^i + 2} \approx n \lg n - 1.26n$$

尽管此方法未在附录 B 中讨论，但可以在 Sahni (1988 年) 的文献中找到。平均情况比最差情况好不了多少。

就记录的赋值次数而言，合并排序的所有情况时间复杂度近似为：

$$T(n) \approx 2n \lg n$$

其证明留作习题。

接下来分析合并排序的空间使用。

### 算法 7.2 的分析 额外空间使用量的分析（合并排序 2）

2.2 节曾经讨论过，即使是算法 2.4 中给出的经过改进的合并排序版本，也另外需要一个大小为  $n$  的数组。此外，当此算法对第一个子数组进行排序时，mid、mid+1、low 和 high 的值都需要存储在活动记录栈中。因为总是从中部将该数组划分开，所以这个栈的深度将增大为  $\lceil \lg n \rceil$ 。附加记录数组占去大部分空间，也就是说，所有情况下额外空间使用量属于  $\Theta(n)$  条记录。“属于  $\Theta(n)$  条记录”的意思是其记录数属于  $\Theta(n)$ 。

### 合并排序的改进

我们可以通过三种途径来改进基本合并排序算法。一种是合并排序的动态规划版本，一种是链接版本，一种是更复杂的合并算法。

对于第一种改进，再看图 2-2 中应用合并排序的例子。如果你正在手工完成合并排序，并不需要将数组划分到只有单独一项，而是直接从单项开始，再将这些单项合并为两个一组，然后再合并为四个一组，以此类推，直到数组变为有序为止。我们可以编写一个迭代版本的合并排序来模拟这一方法，从而避免了实现递归所需要的栈操作开销。注意，这是合并排序的一种动态规划方法。这一版本的算法如下。该算法中的循环将数组大小当作 2 的幂进行处理。当  $n$  值不是 2 的幂时，仍然会将该循环执行  $2^{\lceil \lg n \rceil}$  次，只是不再合并超过  $n$  的部分。

### 算法 7.3 合并排序 3（动态规划版本）

问题：将  $n$  个键排列为非递减顺序。

输入：正整数  $n$ ；键的数组  $S$ ，其索引范围为 1 至  $n$ 。

输出：数组  $S$ ，其中包含了非递减顺序的键。

```
void mergesort3 (int n, keytype S[])
{
    int m;
    index low, mid, high, size;

    m = 2lgn; // 将数组大小当作 2 的幂。
    size = 1; // size 是所合并子数组的大小。
    repeat (lg m 次){
        for (low = 1; low <= m-2*size + 1; low = low + 2*size){
            mid = low + size - 1;
            high = minimum(low + 2*size - 1, n); // 不合并超过 n 的部分。
            merge3(low, mid, high, S);
        }
    }
}
```

```

size = 2 * size; // 使子数组的大小加倍。
}

```

通过这一改进，还可以使记录赋值数递减。数组  $U$  在进程 `merge2`（算法 2.5）中定义为局部变量，在 `mergesort3` 中可以定义为局部数组，索引范围为 1 至  $n$ 。在第一遍执行 `repeat` 循环后， $U$  将包含  $S$  中的项目，其中合并了各对单个数据项。这里不再需要像 `merge2` 中最后所做的那样，将这些项目复制回  $S$  中，而是在第二遍执行 `repeat` 循环时，直接将  $U$  中的项目合并到  $S$  中。也就是，我们颠倒了两个数组的角色。在后续每次循环中，都会颠倒其角色。关于如何编写执行这一操作的 `mergesort3` 和 `merge3` 版本，留作习题。这样，就可以将记录赋值次数由大约  $2n\lg n$  减少至大约  $n\lg n$ 。我们已经推导出：就记录的赋值次数而言，算法 7.3 的所有情况时间复杂度大约为：

$$T(n) \approx n\lg n$$

合并排序的第二种改进是该算法的一种链接版本。如 7.1 节中的讨论，排序问题通常是根据键的取值对记录进行排序。如果记录很大，合并排序使用的额外空间量可能就相当可观了。如果向每个记录增加一个链接字段，就可以减少额外空间。然后，通过调整这些链接就可以将记录排列为一个有序链表，而无需移动这些记录。这意味着不再需要创建一个额外的记录数组。因为一个链接占用的空间要远小于大型记录所占的空间，所以节点的空间量是相当大的。此外，由于调整链接的时间要短于移动大型记录所需的时间，所以还可以节省时间。图 7-3 演示了如何使用链接来完成这一合并过程。下面的算法包含了上述修改。因为不再需要进一步分析 `Mergesort` 和 `Merge`，所以将它们作为一个算法给出。出于可读性考虑，合并排序写为递归形式。当然，前面提到的迭代改进也可以与这一改进一同实现。如果使用了迭代和链接，那重复颠倒  $U$ 、 $S$  角色的改进将不会包含在内，因为在合并链接时不需要额外空间。在这一算法中，数组  $S$  中各项目的数据类型如下：

```

struct node
{
    keytype key;
    index link;
};

```

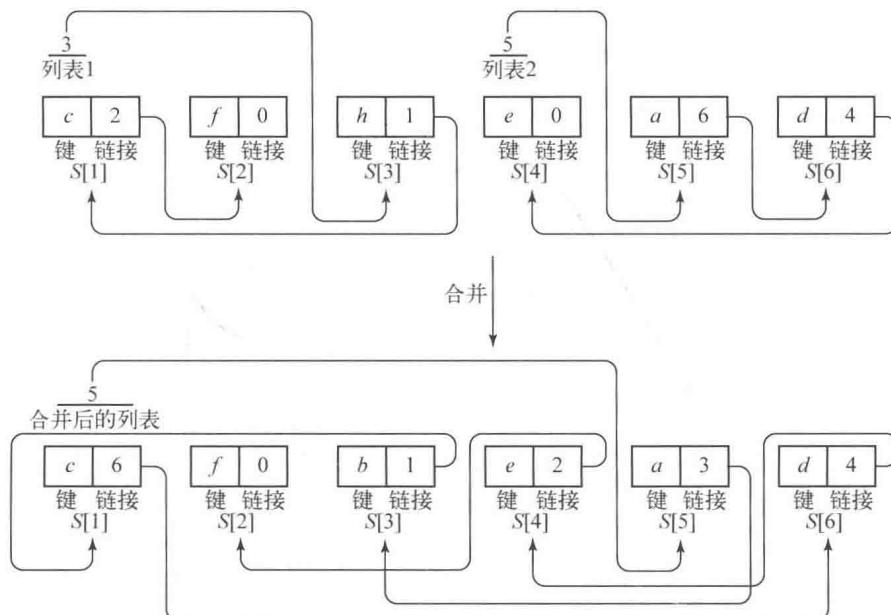


图 7-3 使用链接的合并。箭头用于表示链接是如何工作的。

这里的键是字母，以避免与索引相混淆

#### 算法 7.4 合并排序 4 (链接版本)

问题：将  $n$  个键排列为非递减顺序。

输入：正整数  $n$ ；键的数组  $S$ ，其索引范围为 1 至  $n$ 。

输出：数组  $S$ ，其中包含了非递减顺序的键。

```

void mergesort4(index low, index high, index& mergedlist)
{
    index mid, list1, list2;
    if (low == high){
        mergedlist = low;
        S[mergedlist].link=0;
    }
    else{
        mid = [(low + high)/2];
        mergesort4(low, mid, list1);
        mergesort4(mid + 1, high, list2);
        merge4(list1, list2, mergedlist);
    }
}

void merge4(index list1, index list2, index& mergedlist)
{
    index lastsorted;
    if (S[list1].key < S[list2].key){ // 找出合并列表的起始。
        mergedlist = list1;
        list1=S[list1].link;
    }
    else {
        mergedlist = list2;
        list2 = S[list2].link;
    }
    lastsorted = mergedlist;
    while (list1 != 0 && list2 !=0)
        if S[list1].key < S[list2].key{ // 将较小键附加到合并后的列表。
            S[lastsorted].link = list1;
            lastsorted = list1;
            list1=S[list1].link;
        }
        else{
            S[lastsorted].link = list2;
            lastsorted = list2;
            list2=S[list2].link;
        }
    if (list1 == 0)           // 在一个列表结束后，附接另一个列表的剩余部分。
        S[lastsorted].link=list2;
    else
        S[lastsorted].link=list1;
}

```

并不需要在 `merge4` 的入口处检查 `list1` 或 `list2` 是否为 0，因为 `mergesort4` 从来不会向 `merge4` 传递空列表。和算法 2.4（合并排序 2）中的情景一样， $n$  和  $S$  不是 `mergesort4` 的输入。顶级调用是：

```
mergesort4(1, n, listfront);
```

执行后，`listfront` 将包含有序列表中第一条记录的索引。

在执行后，通常希望有序序列中的记录存储在连续的数组位置中（也就是说，通常意义上的有序），以便使用二分查找（算法 2.1），通过键字段快速访问它们。一旦根据链接完成记录排序后，就有可能采用一种原地  $\Theta(n)$  算法对其重新排列，在连续数组位置变为有序序列。习题中将要求你编写这样一个算法。

合并排序的这一改进完成了两件事。第一，它由原来需要  $n$  个附加记录转化为仅需要  $n$  个链接。也就是说，

有如下分析。

#### 算法 7.4 的分析 额外空间使用量的分析（合并排序 4）

在所有情况下，额外空间使用量都属于  $\Theta(n)$  个链接。“属于  $\Theta(n)$  个链接”是指链接数属于  $\Theta(n)$ 。

第二，如果不需要使连续数组位置中的记录保持有序，那就可以将记录赋值数的时间复杂度缩减至 0，如果需要使其在连续数组位置中保持有序，则缩减至  $\Theta(n)$ 。

对合并排序的第三种改进是 Huang 和 Langston (1988 年) 给出的一种更复杂的合并算法。这一合并算法也是  $\Theta(n)$  的，但其附加空间为很小的常量。

## 7.5 再谈快速排序

先来复习一下快速排序的算法，将其重复如下：

```
void quicksort (index low, index high)
{
    index pivotpoint;

    if (high > low){
        partition(low, high, pivotpoint);
        quicksort(low, pivotpoint - 1);
        quicksort(pivotpoint + 1, high);
    }
}
```

尽管它的最差情况时间复杂度为二次函数，但在 2.4 节已经看到，就键的比较次数而言，快速排序算法的平均情况时间复杂度为：

$$A(n) \approx 1.38(n+1)\lg n$$

它与合并排序相差得不是太多。快速排序相对于合并排序的一个优势就是它不需要额外数组。但是，它仍然不是原地排序，因为当该算法对第一个子数组进行排序时，需要将另一个子数组的第一个索引和最后一个索引存储在活动记录栈中。与合并排序不同的是，并不能保证总会在中间划分数组。在最差情况下，partition 可能会一次又一次地将数组划分为右侧的一个空子数组和左侧一个仅减少一项的子数组。这样，最终会将  $n-1$  对索引放入栈中，也就是说，最差情况下的额外空间使用量属于  $\Theta(n)$ 。可以对快速排序进行修改，使额外空间使用量最多为大约  $\lg n$ 。在给出快速排序的这一改进及其他改进之前，先来讨论快速排序所做记录赋值数的时间复杂度。

习题中将要求你推导：快速排序执行的平均交换次数大约为  $0.69(n+1)\lg n$ 。假定一次交换需要进行三次赋值，则就记录赋值数目而言，快速排序的平均情况时间复杂度为：

$$A(n) \approx 2.07(n+1)\lg n$$

#### 对基本快速排序算法的改进

我们首先以五种不同方式减少快速排序的额外空间使用量。第一，在过程 quicksort 中，我们确定哪个子数组较大，且总是将这个子数组放入栈中，而对另一个进行排序。下面是这一 quicksort 版本所用空间的分析。

#### 额外空间使用量的分析（改进后的快速排序）

在这一版本中，当 partition 恰好每次将数组分为两半时发生最差情况空间使用量，得到的栈深度大约等于  $\lg n$ 。因此，最差情况空间使用量属于  $\Theta(\lg n)$  个索引。

第二，如习题中的讨论，存在 partition 的一个版本，可以大幅减少记录赋值的平均数目。对于该版本，就记录赋值数目而言，快速排序的平均情况时间复杂度为：

$$A(n) \approx 0.69(n+1)\lg n$$

第三，过程 quicksort 中的每个递归调用都会导致 low、high 和 pivotpoint 被放入栈中。大量压入与弹出操

作都是不必要的。在处理对 quicksort 的第一次递归调用时，只有 pivotpoint 和 high 的值需要保存在栈中。而在处理第二次递归调用时，不需要保存任何内容。将 quicksort 写为迭代形式，并在过程中对栈进行排序，可以避免不必要的栈操作。也就是说，我们执行显式栈操作，而不是进行递归栈操作。习题中会要求你完成这一任务。

第四，在2.7节曾经讨论过，可以通过计算一个阈值来改进诸如快速排序之类的递归算法，当达到这一阈值时，该算法不再细分实例，而是调用迭代算法。

最后，在算法2.5（快速排序）的最差情况分析中曾经提到，当输入数组已经有序时，这一算法的效率较低。输入数组越接近有序状态，其性能就越接近最差性能。因此，如果有理由相信数组可能是近乎有序的，那就不要总是选择第一项作为枢纽项，以提高其性能。在这种情况下可以使用的一个好策略是：选择  $S[\text{low}]$ 、 $S[\lfloor(\text{low}+\text{high})/2\rfloor]$  和  $S[\text{high}]$  的中值作为枢纽点。当然，如果没理由认为输入数组中存在任何特定结构，那一般来说，选择任一项目作为枢纽项的性能不会有明显差别。在本例中，取中值作为枢纽项的所有好处就是保证子数组之一不为空（只要三个值互不相同即可）。

## 7.6 堆排序

堆排序不同于合并排序与快速排序，它是一种原地  $\Theta(n \lg n)$  算法。我们首先复习堆的知识，并介绍使用堆进行排序时所需要的基本堆例程，然后介绍如何实现这些例程。

### 7.6.1 堆和基本堆例程

回忆一下，树中一个节点的深度就是从根节点到该节点处唯一路径中的边数，树的深度就是树中所有节点的最大深度，树中的叶节点就是任何没有子节点的节点（见3.5节）。树中的内部节点（internal node）就是至少有一个子节点的节点。也就是说，它是任何不是叶节点的节点。完全二叉树（complete binary tree）是满足以下条件的二叉树：

- 所有内部节点有两个子节点；
- 所有叶节点的深度为  $d$ 。

准完全二叉树（essentially complete binary tree）是满足以下条件的二叉树：

- 在深度为  $d-1$  一级为完全二叉树；
- 具有深度  $d$  的节点都尽可能排列在左侧。

尽管准完全二叉树的定义比较麻烦，但很容易通过一幅图来掌握其性质。图7-4给出了一棵准完全二叉树。

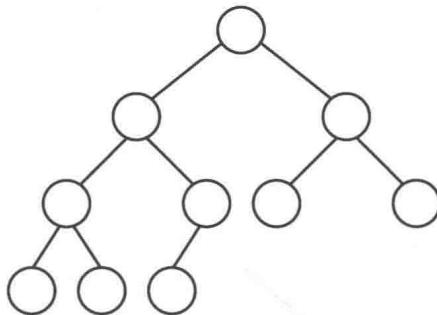


图 7-4 一个准完全二叉树

现在可以定义堆了。堆（heap）就是满足以下条件的准完全二叉树：

- 节点中存储的值来自一个有序集合；
- 每个节点中存储的值大于或等于其子节点中存储的值。这一条称为堆性质（heap property）。

图7-5显示了一个堆。因为我们现在感兴趣的是排序，所以将堆中存储的项称为键。

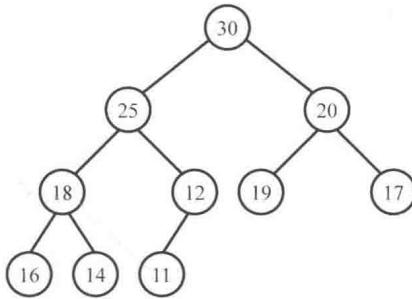


图 7-5 堆

假定已经通过某种方式将要排序的键安排在堆中。如果我们在保持堆性质的同时重复删除存储在根节点的键，那这些键将是按照非递减顺序删除的。如果我们在删除这些键时，把它们放在一个数组中，从第  $n$  个位置开始，一直下移到第 1 个位置，那它们就会按照非递减顺序排列在数组中。在删除了根节点处的键之后，我们可以通过以下操作来恢复堆性质：用存储在底部节点（这里所说的“底部节点”是指最右边的叶节点）的键代替存储在根节点处的键，删除底部节点，调用过程 siftdown，将现在位于根节点的键沿堆下移，直到恢复堆性质。为完成筛选过程，首先将位于根节点的键与其子节点的较大键进行比较。如果根节点处的键较小，则交换这两个键。沿着树向下重复这一过程，直到一个节点处的键不小于其子节点的较大键。图 7-6 说明了这一过程。它的高级伪代码如下：

```

void siftdown (heat& H)
{
    node parent, largerchild;
    parent = H 的根;
    largerchild = parent 拥有较大键的子节点;
    while (parent 的键小于 largerchild 的键){
        交换 parent 与 largerchild 的键;
        parent = largechild;
        largerchild = parent 拥有较大键的子节点;
    }
}
// 在开始时，除根节点外，H 中的所有节点都满足堆性质。
// H 最终是一个堆。

```

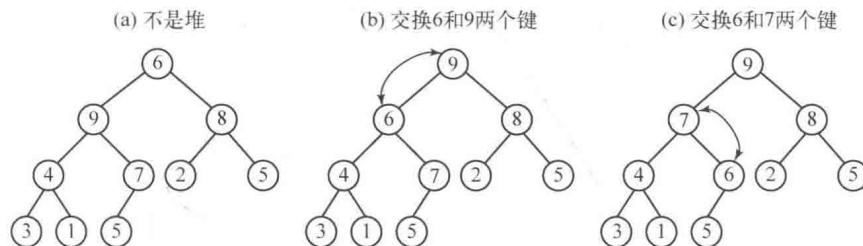


图 7-6 过程 siftdown 将 6 向下移动，直到恢复堆性质

下面给出一个函数的高级伪代码，它删除根节点处的键，并恢复堆性质：

```

keytype root (heap& H)
{
    keytype keyout;

    keyout = 根节点处的键;
    将底部节点处的键移到根节点处; // 底部节点为最右侧的叶节点。
    删除底部节点;
    siftdown(H);
    return keyout;
}
// 恢复堆性质。

```

给定一个包含  $n$  个键的堆，下面是一个过程的高级伪代码，它将有序序列中的键放到数组  $S$  中。

```
void removekeys (int n,
                 heap H,
                 keytype S[])
{
    index i;

    for (i = n; i >= 1; i--)
        S[i] = root(H);
}
```

现在只剩下一项任务：首先将这些键安排在堆中。假定它们已经被安排到一个准完全二叉树中，它不一定具有堆性质（下一小节将会看到如何进行这一操作）。我们只要重复调用 siftdown，以执行以下操作，就可以将一个树转换为一个堆：首先，将根节点深度为  $d-1$  的所有子树转换为堆；其次，将根节点深度为  $d-2$  的所有子树转换为堆；……最后，将整棵树（唯一一棵根节点深度为 0 的子树）转换为一个堆。

这一过程在图 7-7 中说明，由以下高级伪代码概括的过程实现。

(a) 初始结构

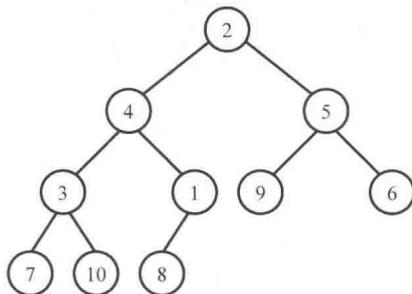
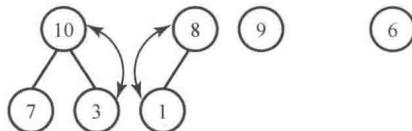
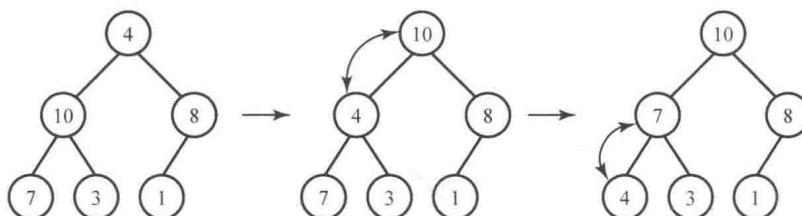
(b) 根节点深度为  $d-1$  的子树被转换为堆(c) 根节点深度为  $d-2$  的左子树被转换为一个堆

图 7-7 使用 siftdown，由一棵准完全二叉树生成一个堆。在执行所示步骤之后，根节点 / 深度为  $d-2$  的右子树必然转换为一个堆，整棵树最后必然转换为一个堆

```
void makeheap (int n, heap& H) // H 最终为一个堆。
{
    index i;
    heap Hsub; // Hsub 最终为一个堆。
```

```

for (i = d - 1; i>=0; i--)
    // 树的深度为 d。
    for (根节点深度为 i 的所有子树 Hsub)
        siftdown(Hsub);
}

```

最后给出堆排序的高级伪代码（假设这些键已经安排在  $H$  的一个准完全二叉树中）：

```

void heapsort (int n,
               heap H,
               keytype S[])
{
    makeheap(n, h);
    removekeys(n, H, S);
}

```

似乎我们没有提前说出实情，因为这个堆排序算法看起来不是一种原地排序。也就是说，需要为堆准备额外空间。但是，接下来将使用一个数组来实现堆。我们将会表明，用来存储输入（待排序的键）的同一数组也可用于实现堆，而且从来不会同时将同一数组位置用于多种用途。

## 7.6.2 堆排序的一种实现

我们可以在一个数组中表示一个准完全二叉树：将根节点存储在第一个数组位置，根节点的左右子节点分别存储在第二、第三位置；根节点左子节点的左右子节点分别存储在第四、第五位置，以此类推。图 7-8 给出了图 7-5 所示堆的数组表示。注意，一个节点的左子节点的索引是原节点索引的两倍，右子节点的索引比原节点的两倍大 1。回想一下，在堆排序的高级伪代码中，需要这些键最初是在一棵准完全二叉树中。如果我们将这些键以任意顺序放在一个数组中，那么根据刚刚讨论的表示形式，将它们设定为某个准完全二叉树。下面的低级伪代码使用了这种表示法。

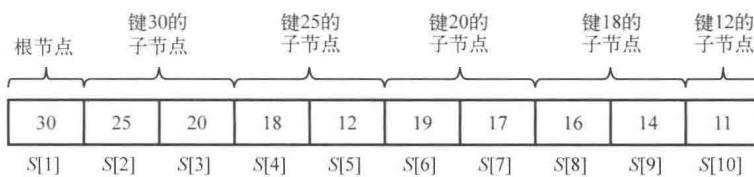


图 7-8 图 7-5 中堆的数组表示

### 堆数据结构

```

struct heap
{
    keytype S[1..n];
    int heapsize;
}

void siftdown(heap& H, index i)
{
    index parent, largerchild;
    keytype siftkey;
    bool spotfound;

    siftkey=H.S[i];
    parent = i;
    spotfound = false;
    while (2*parent <= H.heapsize && ! spotfound){
        if (2*parent < H.heapsize && H.S[2*parent] < H.S.[2*parent + 1])
            largerchild = 2*parent + 1; // 左子节点的索引是 parent 的两倍加 1。
        else
            largerchild= 2*parent; // 左子节点的索引是 parent 的两倍。
        if (siftkey < H.S[largerchild]){

```

```

H.S[parent] = H.S[largerchild];
parent = largerchild;
}
else
spotfound = true;
}
H.S[parent]=siftkey;
}

keytype root (heap& H)
{
keytype keyout;

keyout = H.S[1];           // 获取位于根节点处的键。
H.S[1]=H.S[heapsize];     // 将底部键移至根节点处。
H.heapsize = H.heapsize - 1; // 删除底部节点。
Siftdown(H, 1);           // 恢复堆性质。
return keyout;
}

void removekeys (int n,          // H是按地址传递的，以节省内容。
                 heap& H,
                 keytype S[])
{
index i;
for (i = n; i >= 1; i--)
    S[i] = root(H);
}

void makeheap (int n,          // H最终为一个堆。
               heap& H)
{
index i;                   // 假定n个键都在数组H.S中。

H.heapsize = n;
for (i=[n/2]; i>=1; i--)   // 上一个深度为d-1且具有子节点的节点
    siftdown(H, i);         // 位于数组的第[n/2]个位置。
}

```

现在可以给出堆排序的一种算法了。此算法假设待排序的键已经在  $H.S$  中。这样就自动根据图 7-8 中的表示方式，将它们放在一个准完全二叉树中。在将这个准完全二叉树变为堆后，从第  $n$  个数组位置开始，一直向下到达第一个数组位置，从堆中删除这些键。因为它们在输出数组被安排为同一顺序的有序序列，所以可以将  $H.S$  用作输出数组，而不用担心会覆盖堆中的键。这一策略给出了下面的原位算法。

### 算法 7.5 堆排序

问题：将  $n$  个键排列为非递减顺序。

输入：正整数  $n$ ；键的数组  $S$ ，其索引范围为 1 至  $n$ 。

输出：数组  $S$ ，其中包含了非递减顺序的键。

```

void heapsort (int n, heap& H)
{
makeheap(n, H);
removekeys(n, H, H.S);
}

```

### 算法 7.5 的分析 键比较次数的最差情况时间复杂度（堆排序）

基本指令：过程 `siftdown` 中的键比较。

输入规模： $n$ ，待排序的键的数目。

过程 `makeheap` 和 `removekeys` 都会调用 `siftdown`。我们将分别分析这些过程。首先分析  $n$  为 2 的幂时的情景，然后利用附录 B 中的定理 B.4，将结果扩散到一般  $n$  值。

### 1. makeheap 的分析

设  $d$  是作为输入的准完全二叉树的深度。图 7-9 演示了，当  $n$  为 2 的幂时，这棵树的深度  $d$  为  $\lg n$ ，恰有一个节点具有这一深度，且该节点有  $d$  个祖先。在构建这个堆时，会将树中各个键筛选通过一定的节点数，对于这个  $d$  级节点所有祖先节点中的键来说，它们可能要比该节点不存在时多通过一个节点（也就是这个级别为  $d$  的节点）。而对于所有其他键，筛选通过的节点数与该节点不存在时的数目相同。我们首先给出一个上限，说明当这个深度为  $d$  的节点不存在时，所有键被筛选通过的总节点数。因为该节点有  $d$  个祖先，每个祖先节点的键都可能会多筛选通过一个节点，所以我们可以向这个上限增加一个  $d$ ，得到所有键都被筛选通过的总节点数的实际上限。为此，如果那个深度为  $d$  的节点不存在，那在构建堆时，原来在深度  $d-1$  节点处的每个键将被筛选通过 0 个节点，原来在深度  $d-2$  节点处的每个键最多被筛选通过 1 个节点，以此类推，直到最后，位于根节点处的键最多被筛选通过  $d-1$  个节点。以下证明留作练习：当  $n$  为 2 的幂时，共有  $2^j$  个节点的深度为  $j$  ( $0 \leq j < d$ )。当  $n$  为 2 的幂时，下表给出了每一深度的节点数，以及该深度的一个键最多会被筛选通过多少个节点(前提是不存在深度为  $d$  的那个节点)：

深度	具有这一深度的节点数	一个键会被筛选移动的最大节点数
0	$2^0$	$d-1$
1	$2^1$	$d-2$
2	$2^2$	$d-3$
$\vdots$	$\vdots$	$\vdots$
$j$	$2^j$	$d-j-1$
$\vdots$	$\vdots$	$\vdots$
$d-1$	$2^{d-1}$	0

因此，如果不存在深度为  $d$  的节点，则所有键将被筛选通过的节点数最多为：

$$\sum_{j=0}^{d-1} 2^j(d-j-1) = (d-1)\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{d-1} j(2^j) = 2^d - d - 1$$

最后一个等式是利用附录 A 例 A.3 和 A.5 中的结果，并进行一些代数运算后得到的。回想一下，我们需要向这一上限加上一个  $d$ ，以得到所有键被筛选通过的总节点数的实际上限。因此，该实际上限为：

$$2^d - d - 1 + d = 2^d - 1 = n - 1$$

第二个等式源于以下事实：当  $n$  是 2 的幂时， $d = \lg n$ 。每当一个键被筛选通过一个节点时，过程 siftdown 中的 while 循环都会执行一遍。因为每执行一遍该循环，会进行两次键的比较，所以 makeheap 对键的比较次数最多为：

$$2(n-1)$$

这是一个多少让人感到惊讶的结果：堆可以在线性时间内构建完毕。如果可以在线性时间删除这些键，就可以得到一种线性时间排序算法。但后面将会看到，事实并非如此。

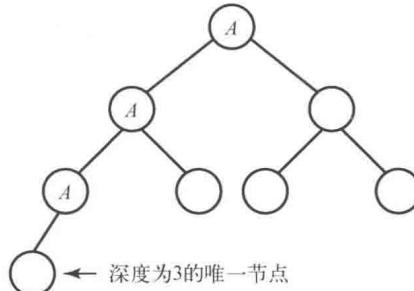


图 7-9  $n=8$  的一个例示，说明：如果一个准完全二叉树有  $n$  个节点，且  $n$  是 2 的幂，则该树的深度  $d$  为  $\lg n$ ，存在一个深度为  $d$  的节点，该节点有  $d$  个祖先。这个节点的三个祖先标有“ $A$ ”

## 2. removekeys 的分析

图 7-10 演示了当  $n=8$  和  $d=\lg 8=3$  时的情景。如图 7-10a 和图 7-10b 所示，在分别删除第一个键和第四个键时，移到根节点的键最多筛选通过  $d-1=2$  个节点。显然，介于第一、第四个键之间的两个键也是如此。因此，当在删除前四个键时，被移到根节点处的键最多筛选通过两个节点。如图 7-10c 和图 7-10d 所示，在分别删除接下来的两个键时，移到根节点的键最多筛选通过  $d-2=1$  个节点。最后，图 7-10e 显示，在删除下一个键时，移到根节点的键被筛选通过 0 个节点。显然，在删除最后一个键时，也不存在筛选操作。所有键被筛选通过的总节点数最多为：

$$1(2) + 2(4) = \sum_{j=1}^{3-1} j2^j$$

不难看出，这一结果可以扩展到  $n$  为 2 的任意幂时的情景。但每将一个键筛选通过一个节点时，都会将过程 siftdown 中的 while 循环执行一遍，因为每执行该循环一次，都会进行两次键的比较，removekeys 完成的键比较次数最多为：

$$2 \sum_{j=1}^{d-1} j2^j = 2(d2^d - 2^{d+1} + 2) = 2n\lg n - 4n + 4$$

第一个等式是运用附录 A 例 A.5 中的结果，并进行一些代数运算后得到的，而第二个等式则源于如下事实：当  $n$  是 2 的一个幂时， $d=\lg n$ 。

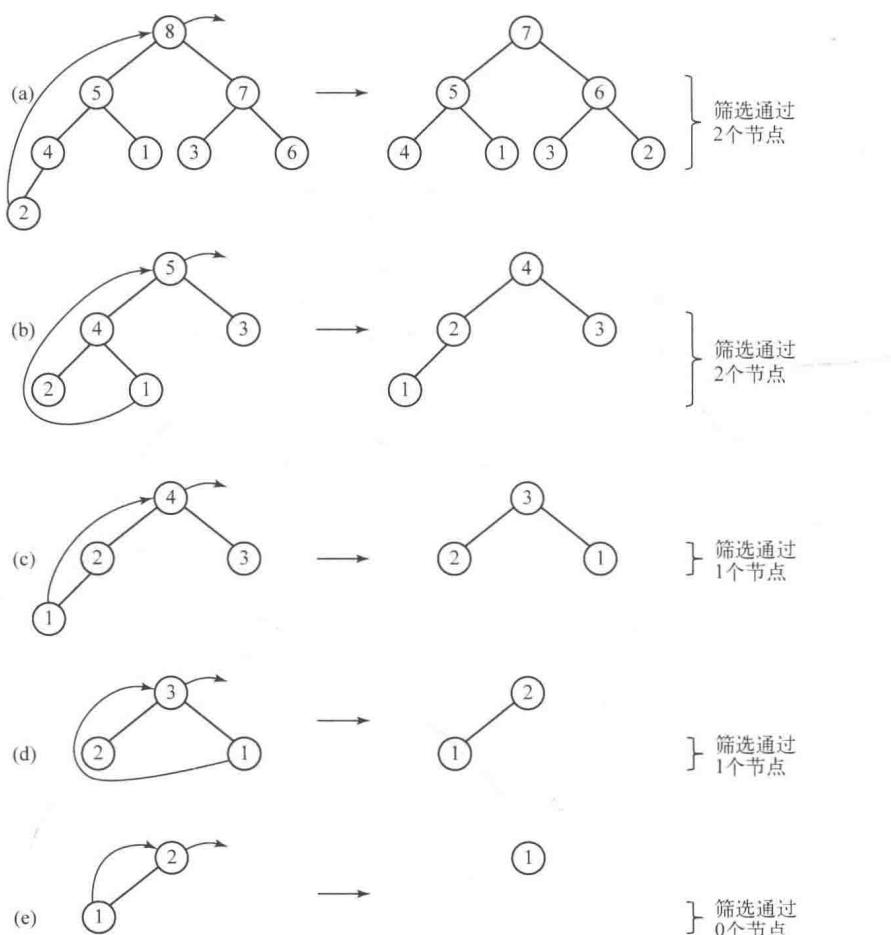


图 7-10 从一个 8 节点堆中删除键。(a)中描述第 1 个键的删除过程；(b)中是第 4 个；(c)中是第 5 个；(d)中是第 6 个；(e)中是第 7 个。移到节点处的键被筛选通过的节点数在右侧给出

### 3. 结合两个分析

结合对 makeheap 和 removekeys 的分析可知，当  $n$  为 2 的幂时，堆排序中执行的键比较次数最多为：

$$2(n-1)+2n\lg n-4n+4=2(n\lg n-n+1) \approx 2n\lg n$$

习题中将要求你证明：存在一种需要这一比较次数的情景。因此，当  $n$  为 2 的幂时，

$$W(n) \approx 2n\lg n \in \Theta(n\lg n)$$

有可能证明  $W(n)$  是最终非递减的。因此，由附录 B 中的定理 B.4 可以推出，对于一般的  $n$  值，有

$$W(n) \in \Theta(n\lg n)$$

似乎很难以解析方式来分析堆排序的平均情况时间复杂度。但是，试验研究已经表示，这一平均情况并不比其最差情况好太多。这就是说，就键比较次数而言，堆排序的平均情况时间复杂度大约为：

$$A(n) \approx 2n\lg n$$

习题中将要求你证明：堆排序完成的记录赋值次数的最差情况时间复杂度近似为：

$$W(n) \approx n\lg n$$

和键的比较次数一样，堆排序在平均情况下的性能并未胜出上述结果太多。

最后，根据之前的讨论结果，有以下空间利用量。

#### 算法 7.5 的分析 额外空间使用量的分析（堆排序）

堆排序是一种原地排序，也就是说，其额外空间属于  $\Theta(1)$ 。

7.2 节曾经提到，堆排序是选择排序的一个例子，因为它的排序过程是依次选择记录，再将它们放在正确位置。将记录放在其正确位置的工作是通过调用 removekeys 完成的。

## 7.7 合并排序、快速排序和堆排序的比较

表 7-2 总结了这三种算法的结果。因为平均来说，堆排序在键的比较和记录赋值两个方面都要劣于快速排序，而且因为快速排序的额外空间使用量最小，所以快速排序通常胜于堆排序。因为我们对合并排序的最初实现（算法 2.2 和算法 2.4）另外使用了一个完整的记录数组，而且在平均情况下，合并排序执行的记录赋值数总是快速排序的大约三倍，所以快速排序通常胜于合并排序，尽管在平均情况下，快速排序执行的键比较次数要稍多一些。但是，合并排序的链表实现（算法 7.4）消除了合并排序的几乎所有缺点。唯一剩下的缺点就是  $\Theta(n)$  个额外链接所使用的附加空间。

表 7-2  $\Theta(n\lg n)$  排序算法的分析小结\*

算 法	键的比较	记录的赋值	额外空间使用量
合并排序 (算法 2.4)	$W(n)=n\lg n$ $A(n)=n\lg n$	$T(n)=2n\lg n$	$\Theta(n)$ 条记录
合并排序 (算法 7.4)	$W(n)=n\lg n$ $A(n)=n\lg n$	$T(n)=0^\dagger$	$\Theta(n)$ 条链接
快速排序 (进行改进后)	$W(n)=n^2/2$ $A(n)=1.38n\lg n$	$A(n)=0.69n\lg n$	$\Theta(\lg n)$ 个索引
堆排序	$W(n)=2n\lg n$ $A(n)=2n\lg n$	$W(n)=n\lg n$ $A(n)=n\lg n$	原地

\*这些项目是近似结果；合并排序与堆排序的平均情况稍优于最差情况。

†如果要求连续数组位置中的记录保持有序序列，则最差情况属于  $\Theta(n)$ 。

## 7.8 仅通过键的比较进行排序的下限

我们已经开发了  $\Theta(n \lg n)$  排序算法，相对于二次时间算法来言，这些算法的提升是非常显著的。有一个很好的问题是：能否开发出一些排序算法，使其时间复杂度的阶数更好一些？我们将证明，只要是通过键的对比来进行排序，那就不可能找出这样的算法。

尽管在考虑概率排序算法时，我们的结果仍然成立，但这些结果是针对确定性排序算法获得的。（关于概率性算法和确定性算法的讨论，请见 5.3 节。）正如 7.3 节的做法一样，这些分析结果是在  $n$  个键互不相同的假设下得到的。此外，在该节还讨论过，可以假设这  $n$  个键就是正整数  $1, 2, \dots, n$ ，因为可以用 1 代替最小键，用 2 代替第二小的键，以此类推。

### 7.8.1 排序算法的决策树

考虑下面对三个键进行排序的算法。

```
void sortthree (keytype S[])
{
    keytype a, b, c;
    a = S[1]; b = S[2]; c = S[3];
    if (a < b)
        if (b < c)
            S = a, b, c;           // 其含义是 S[1]=a; S[2]=b; S[3]=c;
        else if (a < c)
            S = a, c, b;
        else
            S = c, a, b;
    else if (b < c)
        if (a < c)
            S = b, a, c;
        else
            S = b, c, a;
    else
        S = c, b, a;
}
```

我们为过程 `sortthree` 关联一个二叉树，如下所示。将  $a$  和  $b$  的比较放在根节点处。根节点的左子节点比较  $a$  与  $b$ ，判断是否  $a < b$ ，而其右子节点判断是否  $a \geq b$ 。我们将向下进行，创建树中的各个节点，直到为该算法可能执行的所有比较均分配了节点为止。排序后的键存储在叶节点处。图 7-11 给出了整棵树。这棵树称为决策树，因为在每个节点都需要判断接下来要访问哪个节点。过程 `sortthree` 对特定输入的操作对应于从根节点到叶节点的一条唯一路径，该路径由输入决定。对于这三个键的每一种排序，树中都有一个叶节点与之对应，这是因为该算法可以对规模为 3 的所有可能输入进行排序。

如果对于  $n$  个键的每种排列，都存在从根节点到一个叶节点的一条路径，用来对该排列排序，则说这个决策树对于这  $n$  个键的排序是有效的。也就是说，它可以对规模为  $n$  的所有输入进行排序。例如，图 7-11 中的决策树对于 3 个键的排序是有效的，但如果从树中删除任意分支，它就不再有效。对于每个对  $n$  个键进行排序的确定性算法，都至少有一棵有效决策树。图 7-11 中的决策树对应于过程 `sortthree`，图 7-12 中的决策树对应于对三个键进行排序的交换排序算法（建议读者验证这一点）。在此树中， $a$ 、 $b$  和  $c$  仍然是  $S[1]$ 、 $S[2]$  和  $S[3]$  中的初始值。比如，当一个节点包含比较  $c < b$  时，这并不是说交换排序会在这一点对比  $S[3]$  和  $S[2]$ ，而是说，交换排序会将当前值为  $c$  的数组项与当前值为  $b$  的数组项进行对比。在图 7-12 的树中，注意，包含比较  $b < a$  的第 2 级节点没有右子节点。其原因是如果对这一比较的回答为“否”，则与在通向该节点的路径上获得的回答矛盾，这就是说，如果从根节点开始做出的一系列决策都是一致的，那就不可能到达它的右子节点。交换排序会在这里进行一次不必要的比较，因为交换排序不“知道”对此问题回答必须为“是”。在次优排序算法中经常发生这种情况。在一棵决策树中，如果通过一系列前后一致的决策，可以由根节点到达每个叶节点，就说这棵

决策树是经过修剪的。图 7-12 中的决策树是经过修剪的，但如果向刚刚讨论的节点添加一个右子节点，这个树就不再是经过修剪的，尽管它仍然是有效的，仍然与交换排序相对应。显然，对于每个对  $n$  个键进行排序的确定性算法来说，都有一棵经过修剪的有效决策树与之相对应。于是有以下引理。

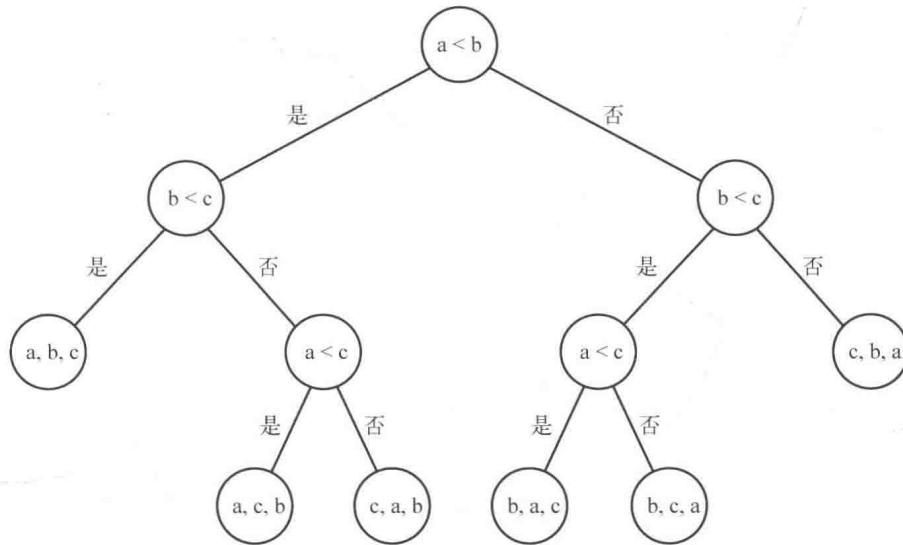


图 7-11 与过程 sortthree 对应的决策树

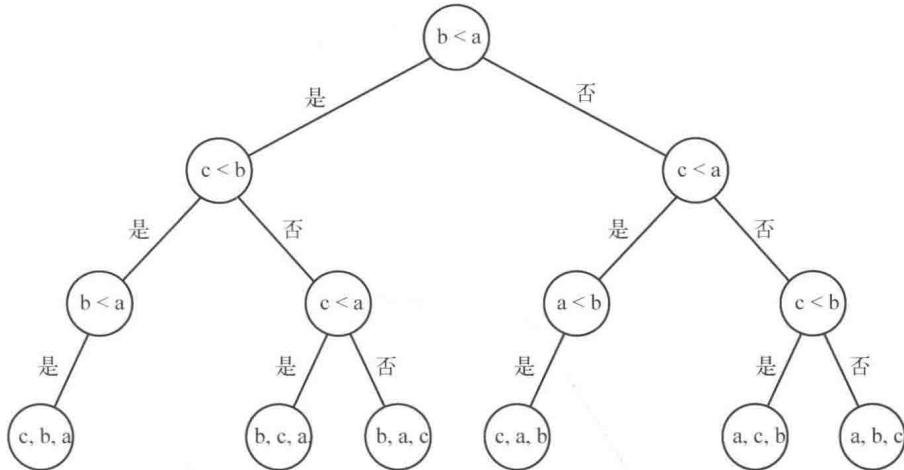


图 7-12 与交换排序算法对三个键进行排序所对应的决策树

**引理 7.1** 对于每个对  $n$  个互不相同键进行排序的确定性算法，都有一棵恰好包含  $n!$  个叶节点、经过修剪的有效二叉树与之相对应。

**证明：**如前所述，对于任何对  $n$  个键进行排序的算法都有一个经过修剪的有效决策树与之相对应。当所有这些键互不相同时，比较的结果总是“ $<$ ”或“ $>$ ”。因此，树中每个节点最多有两个子节点，这就意味着它是一棵二叉树。接下来证明它有  $n!$  个叶子。因为共有  $n!$  种不同输入中包含  $n$  个互不相同的键，而且只有当一棵决策树对于每个输入均有一个叶节点时，它对于  $n$  个不同键的排序才是有效的，所以这棵树至少有  $n!$  个叶节点。因为对于  $n!$  个不同输入中的每一个，树中都有一条独一无二的路径，而且在一棵经过修剪的决策树中，每个叶节点都必定可以到达，所以树中的叶节点数不会超过  $n!$  个。因此，树中恰好有  $n!$  个叶节点。

利用引理 7.1，通过研究具有  $n!$  个叶节点的二叉树，可以为  $n$  个不同键的排序计算出下限。下面就来进行计算。

## 7.8.2 最差情况下的下限

为了获得在最差情况下键比较次数的界限，需要以下引理。

**引理 7.2** 一棵决策树在最差情况下完成的比较次数等于其深度。

**证明：**给定某一输入，一棵决策树完成的比较次数就是为该输入所走路径上的内部节点数。这一内部节点数与路径的长度相同。因此，决策树在最差情况下完成的比较次数就是到一个叶节点的最长路径的长度，也就是决策树的深度。

根据引理 7.1 和引理 7.2，针对一棵包含  $n!$  个叶节点的二叉树，只需要求出其深度下限即可获得最差表现时的下限。利用以下引理和定理即可求出所需要的深度下限。

**引理 7.3**

**证明：**首先证明

$$2^d \geq m \quad (7.1)$$

显然，如果一棵二叉树是不完全的，就可以通过添加叶节点由原树生成一棵新的二叉树，使新树的叶节点多于原树，但深度与原树相同。因此，为完全二叉树获得式 7.1 就足够了。实际上，我们将针对完全二叉树获得该等式。证明过程采用归纳法。

**归纳基础：**深度为 0 的（完全）二叉树只有一个节点，它既是根节点，又是唯一的叶节点。因此，对于这样一棵树，叶子数  $m$  等于 1，且

$$2^0 = 1$$

**归纳假设：**假定对于深度为  $d$  的完全二叉树，有

$$2^d = m$$

其中  $m$  是叶子数。

**归纳步骤：**需要证明，对于深度为  $d+1$  的完全二叉树，有

$$2^{d+1} = m'$$

其中  $m'$  是叶子数。我们可以由深度为  $d$  的完全二叉树获得一个深度为  $d+1$  的完全二叉树：向原二叉树中的每个叶子恰好增加两个子节点，这些子节点就成为新树中仅有的叶子。因此，

$$2m = m'$$

根据此等式及归纳假设，有

$$2^{d+1} = 2 \times 2^d = 2m = m'$$

完成归纳证明。

对不等式 7.1 的两侧取  $\lg$ ，得：

$$\lg d \geq \lg m$$

因为  $d$  是一个整数，所以可导出：

$$d \geq \lceil \lg m \rceil$$

**定理 7.2** 任何仅通过键的比较对  $n$  个不同键进行排序的确定性算法，在最差情况下至少要进行  $\lceil \lg(n!) \rceil$  次键的比较。

**证明：**根据引理 7.1，对于任何此类算法，都有一棵包含  $n!$  个叶子、经过修剪的有效二叉决策树与之相对应。根据引理 7.3，该树的深度大于或等于  $\lceil \lg(n!) \rceil$ 。现在可以得出此定理了，因为引理 7.2 表明，任何决策树在最差情况下的比较次数由其深度给出。

这一下限与合并排序在最差情况下的性能（也就是  $n \lg n - (n-1)$ ）相比如何呢？利用引理 7.4 可以比较两者。

④引理 7.4 对于任意正整数  $n$ ,

$$\lg(n!) \geq n \lg n - 1.45n$$

证明: 此证明需要积分知识。我们有

$$\begin{aligned} \lg(n!) &= \lg[n(n-1)(n-2)\cdots(2)1] \\ &= \sum_{i=2}^n \lg i \quad \{ \text{因为 } \lg 1 = 0 \} \\ &\geq \int_1^n \lg x dx = \frac{1}{\ln 2} (n \ln n - n + 1) \geq n \lg n - 1.45n \end{aligned}$$

定理 7.3 任何仅通过键的比较对  $n$  个不同键进行排序的确定性算法, 在最差情况下至少要进行  $[n \lg n - 1.45n]$  次键的比较。

证明: 由定理 7.2 和引理 7.4 可以得证。

我们看到, 合并排序的最差性能  $n \lg n - (n-1)$  已经接近最优了。接下来将证明, 对于平均性能也是如此。

### 7.8.3 平均情况下的下限

以下这些结果是在假设“输入内容取任意可能排列的概率相同”的前提下得出的。

对于一个对  $n$  个不同键进行排序的确定性排序算法, 如果在与其对应的已修剪的有效二叉决策树中, 存在只有一个子节点的比较节点(图 7-12 中的树即是如此), 那就可以将每个此种节点用其子节点代替, 然后修剪掉该子节点, 获得一个新的决策树, 此决策树进行排序时使用的比较次数不会超过原树。新树中的每个非叶节点将恰好包含两个子节点。如果一棵二叉树中的每个非叶节点都恰好包含两个子节点, 则称之为 2 树。我们用以下引理来总结这一结果。

引理 7.5 对于每棵对  $n$  个不同键进行排序的已修剪有效二叉决策树, 都有一棵已修剪的有效决策 2 树之前对应, 其效率至少与原树一样高。

证明: 由以下讨论可得出证明。

一棵树的外部路径长度(EPL, External Path Length)是指从根节点到叶节点的所有路径总长。例如, 对于图 7-11 中的树,

$$EPL = 2+3+3+3+3+2=16$$

回想一下, 一棵决策树为到达一个叶节点所执行的比较次数就是到该叶节点的路径长度。因此, 一棵决策树的 EPL 就是对所有可能输入进行排序时所完成的总比较次数。因为规模为  $n$  的不同输入共有  $n!$  种(当所有键互不相同时), 而且我们假设所有输入的概率相同, 所以一棵决策树对  $n$  个不同键进行排序所执行的平均比较数为:

$$\frac{EPL}{n!}$$

利用这一结果可以证明一个重要的引理。首先, 将  $\min EPL(m)$  定义为包含  $m$  个叶节点的 2 树的最小 EPL。下面给出引理。

引理 7.6 任何仅通过键的比较对  $n$  个不同键进行排序的确定性算法, 平均来说都至少要进行  $\frac{\min EPL(n!)}{n!}$  次键的比较。

证明: 由引理 7.1 可知, 对于每个对  $n$  个不同键进行排序的确定性算法, 都有一棵包含  $n$  个叶子的已修剪的有效二叉树与之对应。由引理 7.5 可知, 可以将此决策树转换为一个 2 树, 其效率至少与原树一样。因为原树有  $n!$  个叶子, 所以由其获得的 2 树也必然如此。由之前的讨论可得出此引理。

根据引理 7.6, 要获得平均情况下的下限, 只需要找出  $\min EPL(m)$  的下限。利用下面 4 个引理可完成这一

任务。

**引理 7.7** 任何一个拥有  $m$  个叶节点且 EPL 等于  $\text{minEPL}(m)$  的 2 树，其所有叶子最多分布在底部两层。

证明：假定有些 2 树的叶子并非都分布在底部两层。设  $d$  是树的深度，设  $A$  是树中不在底部两层的一个叶节点。因为底层节点的深度为  $d$ ，所以

$$k \leq d-2$$

我们将证明，在叶节点个数相同的树中，这个树的 EPL 不可能是最小的，具体做法就是设计一个具有相同叶节点数的 2 树，但其 EPL 要更小一些。为此，从原树的  $d-1$  级选择一个非叶节点  $B$ ，删除它的两个子节点，并为  $A$  提供两个叶节点，如图 7-13 所示。显然，新树中的叶节点数与原树相同。在新树中， $B$  的子节点和  $A$  都不是叶节点，但它们是老树中的叶节点。因此，EPL 将会减去指向  $A$  的路径的长度，还会减去另外两条路径的长度，也就是指向  $B$  的两个子节点的路径。因此，EPL 共缩减了

$$k+d+d=k+2d$$

在新树中， $A$  的两个新子节点和  $B$  是叶节点，但它们不是老树中的叶节点。因此，EPL 将减去指向  $B$  的路径的长度，还会减去另外两条路径的长度，也就是指向  $B$  的两个新子节点的路径。因此，EPL 共缩减了

$$d-1+k+1+k+1=d+2k+1$$

EPL 的净变化为：

$$(d+2k+1)-(k+2d)=k-d+1 \leq d-2-d+1=-1$$

这个不等式的出现是因为  $k \leq d-2$ 。因为 EPL 的净变化为负值，所以新树的 EPL 较小。这就证明了：在叶节点个数相同的树中，老树的 EPL 不可能是最小的。

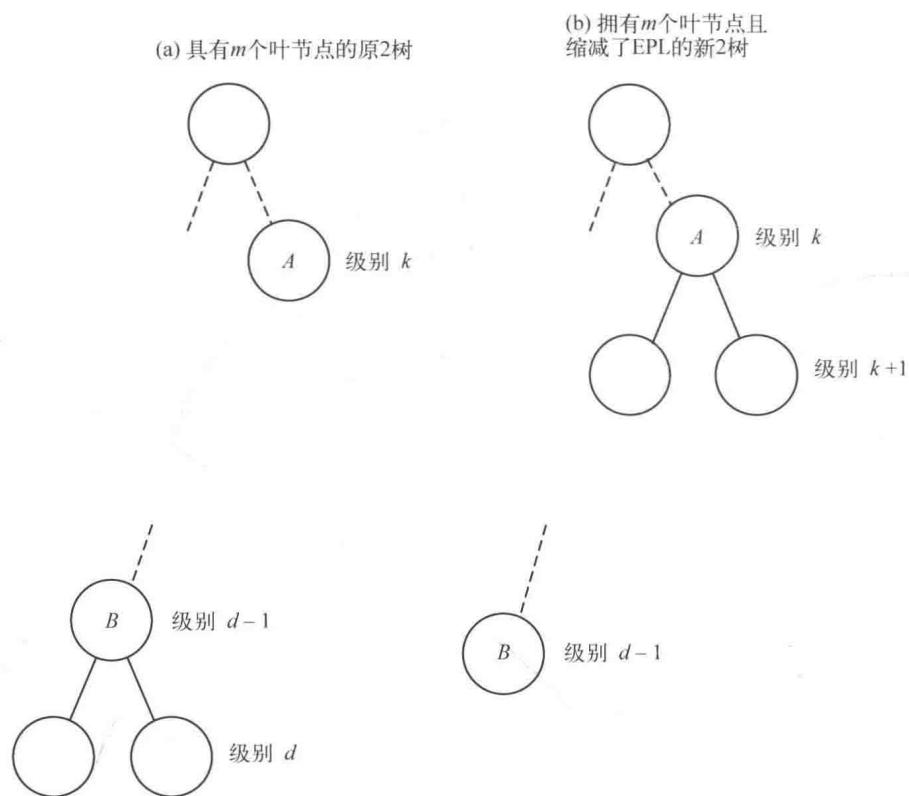


图 7-13 (a)和(b)中的树具有相同数目的叶节点，但(b)中树的 EPL 较小

**引理 7.8** 任何一个拥有  $m$  个叶节点，且其 EPL 等于  $\text{minEPL}(m)$  的 2 树都必然：在  $d-1$  级拥有  $2^{d-1}-m$  个叶节点，在  $d$  级拥有  $2m-2^d$  个叶节点，而且没有任何其他叶节点，其中  $d$  是树的深度。

**证明：**因为由引理 7.7 可知，所有叶节点都在底部两级，而且因为 2 树中的非叶节点都必然有两个子节点，所以不难看出，在  $d-1$  级必然有  $2^{d-1}$  个节点。因此，如果  $r$  是  $d-1$  级的叶节点个数，则这一级的非叶节点个数为  $2^{d-1}-r$ 。因为 2 树中的非叶节点恰有两个子节点，所以对于  $d-1$  级的每个非叶节点，都在  $d$  级有两个叶节点。因为它们是  $d$  级仅有的叶节点，所以  $d$  级的叶节点数等于  $2(2^{d-1}-r)$ 。因为根据引理 7.7 知道，所有叶节点都在  $d$  级或  $d-1$  级，所以

$$r+2(2^{d-1}-r)=m$$

化简后得到：

$$r=2^d-m$$

因此，在  $d$  级的叶节点数为：

$$m-(2^d-m)=2m-2^d$$

**引理 7.9** 对于任何拥有  $m$  个叶节点且 EPL 等于  $\text{minEPL}(m)$  的 2 树，其深度  $d$  由下式给出：

$$d=\lceil \lg m \rceil$$

**证明：**我们证明当  $m$  是 2 的幂时的情景。一般情况下的证明留作练习。如果  $m$  是 2 的幂，则对于某个整数  $k$ ，有

$$m=2^k$$

设  $d$  是最小树的深度。和在引理 7.8 中一样，设  $r$  是在  $d-1$  级的叶节点个数。根据该引理，有

$$r=2^d-m=2^d-2^k$$

因为  $r \geq 0$ ，所以必然有  $d \geq k$ 。我们将证明，假设  $d > k$  将导致矛盾。若  $d > k$ ，则

$$r=2^d-2^k \geq 2^{k+1}-2^k=2^k=m$$

因为  $r \leq m$ ，所以这意味着  $r=m$ ，所有叶节点都在  $d-1$  级。但  $d$  级必然有一些叶节点。这一矛盾可推出  $d=k$ ，它意味着  $r=0$ 。因为  $r=0$ ，所以

$$2^d-m=0$$

也就是说， $d=\lceil \lg m \rceil$ 。因为当  $m$  是 2 的幂时， $\lceil \lg m \rceil = \lceil \lg m \rceil$ ，证毕。

**引理 7.10** 对于所有整数  $m \geq 1$ ，有

$$\text{minEPL}(m) \geq m \lceil \lg m \rceil$$

**证明：**根据引理 7.8，任何一个使这一 EPL 取最小值的 2 树都必然在  $d-1$  级拥有  $2^{d-1}-m$  个叶节点，在  $d$  级拥有  $2m-2^d$  个叶节点，而且没有任何其他叶节点。于是有

$$\text{minEPL}(m)=(2^{d-1}-m)(d-1)+(2m-2^d)d=md+m-2^d$$

因此，根据引理 7.9，

$$\text{minEPL}(m)=m(\lceil \lg m \rceil)+m-2^{\lceil \lg m \rceil}$$

如果  $m$  是 2 的幂，这个表达式显然等于  $m \lceil \lg m \rceil$ ，在本例中，后者等于  $m \lceil \lg m \rceil$ 。如果  $m$  不是 2 的幂，则  $\lceil \lg m \rceil = \lfloor \lg m \rfloor + 1$ 。所以在本例中，

$$\begin{aligned} \text{minEPL}(m) &= m(\lfloor \lg m \rfloor + 1) + m - 2^{\lceil \lg m \rceil} \\ &= m\lfloor \lg m \rfloor + 2m - 2^{\lceil \lg m \rceil} > m\lfloor \lg m \rfloor \end{aligned}$$

这个不等式成立，是因为在一般情况下， $2m > 2^{\lceil \lg m \rceil}$ 。证毕。

现在已经获得了  $\text{minEPL}(m)$  的下限，可以证明主要结果了。

**定理 7.4** 任何一个仅通过键的比较对  $n$  个不同键进行排序的确定性算法，其在平均情况下至少要进行  $\lceil n \lg n - 1.45n \rceil$  次键的比较。

证明：根据引理 7.6，任何此种算法在平均情况下至少要进行  $\frac{\min EPL(n!)}{n!}$  次键的比较。

根据引理 7.10，这一表达式大于或等于

$$\frac{n! \lfloor \lg(n!) \rfloor}{n!} = \lfloor \lg(n!) \rfloor$$

由引理 7.4 可得到证明。

我们看到，合并排序的平均性能大约为  $n \lg n - 1.26n$ ，作为仅通过键的比较进行排序的算法来说，这已经是接近最优的了。

## 7.9 分配排序（基数排序）

上一节证明了：任何仅通过键的比较来进行排序的算法，不可能优于  $\Theta(n \lg n)$ 。如果我们对键的全部了解就是它们来自一个有序集合，那除了比较键之外，别无选择。但是，如果还掌握更多信息，就可以考虑其他排序算法。下面我们将利用键的更多信息，开发这样一种算法。

假定已知这些键都是以 10 为基数表示的非负整数。假定它们的位数都相同，可以根据最左边一位的数值将它们分配到不同的堆中。也就是说，最左边数位相同的键被放在同一堆中。然后再根据左数第二位将每个堆分到不同堆中。然后再根据左数第三位，将新堆分到不同堆中，以此类推。在检查了所有数位之后，这些键就变成有序的了。图 7-14 演示了这一过程，它称为分配排序（sort by distribution），因为这些键被分配到了不同堆中。

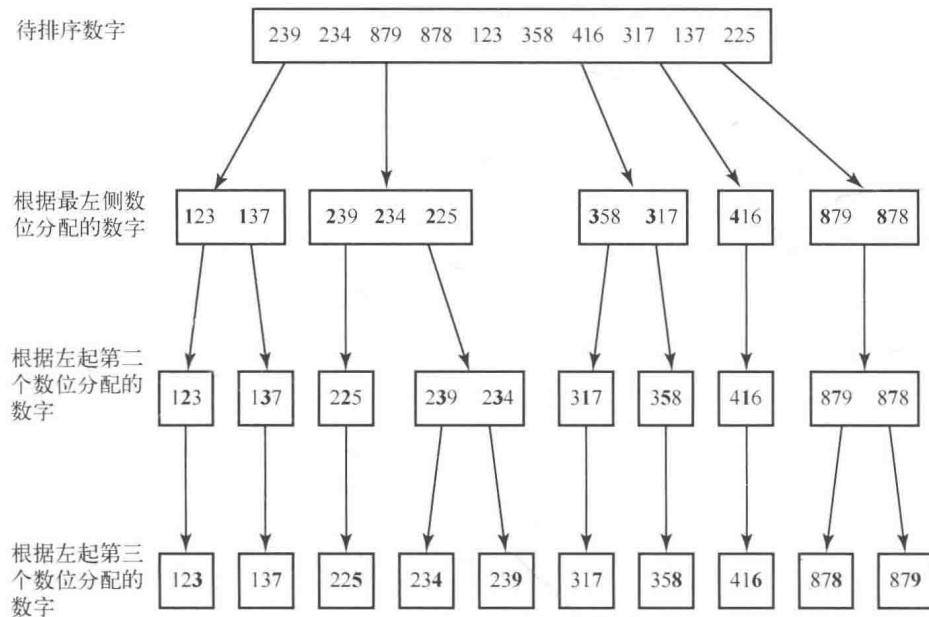


图 7-14 分配排序，自左向右查看各个数位

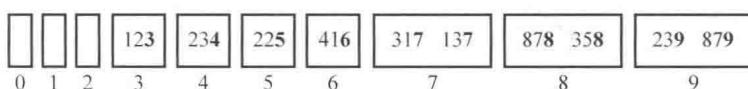
这一过程的一个难点在于，需要不同数目的堆。假定我们恰好分配 10 堆（每个十进制数位各占一个），从右向左查看数位，总是将一个键放到与当前所看数位对应的堆中。如果这样做的话，只要遵从以下规则，这些键最终仍然是有序的：在每次扫描中，如果两个键被放在同一堆中，（在上一次扫描中）来自最左堆的键放

在其他键的左边。这个过程在图 7-15 中说明。我们通过一个例子来说明如何放置这些键，注意在第一遍扫描后，键 416 是在键 317 左边的一个堆中。因此，当它们在第二遍扫描中都被放在第一堆时，键 416 放在键 317 的左边。但在第三遍扫描中，键 416 最终是在键 317 的右边，因为它被放在第四堆中，而键 317 被放在第三堆中。这样，在第一遍扫描之后，这些键将根据它们最右边的数位处于正确顺序，在第二遍扫描之后，将根据它们最右边的两个数位处于正确顺序，在第三遍扫描之后，将根据所有三个数位处于正确顺序，这就是说，它们变成有序的了。

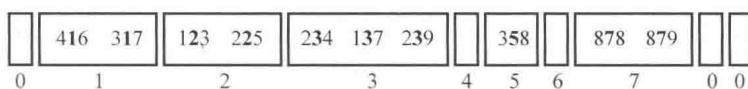
待排序数字

239 234 879 879 123 358 416 317 137 137 225

根据最右侧数位分配的数字



根据右起第二个数位分配的数字



根据右起第三个数位分配的数字

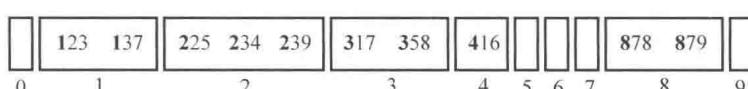


图 7-15 分配排序，自右向左查看各个数位

这种排序方法的出现要早于计算机，是老式卡片排序机中使用的一种方法。它被称为基数排序(radix sort)，这是因为，用于对这些键进行排序的信息是一个具体的基数。这个基数既可以是任意数字，也可以是字母表中的字符。堆的个数与基数相同。例如，在对十六进制数字进行排序时，堆数为 16；在对英文字母表表示的字符键进行排序时，因为该字母表中有 26 个字母，所以堆数将是 26。

因为一个特定堆中的键数会在每遍扫描中发生变化，所以实现该算法的一种好办法是使用链表。每个堆用一个链表表示。在每遍扫描之后，将这些键从列表（堆）中删除，接合到一个主链表中。它们在这个列表中的排序依据是：它们是从哪个列表（堆）中被删除的。在下一遍扫描中，从头遍历该主列表，每个键被放在它在该遍扫描中所属列表（堆）的最后。这样就遵守了前面给出的规则。为便于理解，我们在给出此算法时，假定这些键是用十进制表示的非负整数。稍做修改，就可以用来对其他基数表示的键进行排序，而且不会影响时间复杂度的阶。此算法需要以下变量声明：

```
struct nodetype
{
    keytype key;
    nodetype* link;
};

typedef nodetype* node_pointer;
```

#### 算法 7.6 基数排序

问题：将  $n$  个以基数 10 表示的非负整数表示为非递减顺序。

输入： $n$  个非负数组成的链表 masterlist；一个整数 numdigits，它是每个整数中的最大十进制位数。

输出：以非递减顺序包含这些整数的链表 masterlist。

```

void radixsort (node_pointer& masterlist,
                int numdigits)
{
    index i;
    node_pointer list[0..9];

    for (i = 1; i <= numdigits; i++){
        distribute(masterlist, i);
        coalesce(masterlist);
    }
}

void distribute (node_pointer& masterlist, index i);
                    // i 是当前所查看数位的索引。
{
    index j;
    node_pointer p;

    for (j = 0; j<=9; j++)           // 清空当前堆。
        list[j]=NULL;
    p = masterlist;                  // 遍历 masterlist。
    while (p != NULL){
        j = p-> key 中第 i 位的值 (右起) ;
        将 p 链接到 list[j] 的末尾;
        p = p -> link;
    }
}

void coalesce(node_pointer& masterlist);
{
    index j;

    masterlist = NULL;             // 清空 masterlist。
    for (j = 0; j <= 9; j++)
        将 list[j] 中的节点链接到 masterlist 的末尾;
}

```

下面来分析该算法。

#### 算法 7.6 的分析 所有情况时间复杂度（基数排序）

**基本运算：**因为这一算法中没有键的比较，所以需要找出一种不同的基本运算。在 coalesce 的一种高效实现中，包含这些堆的列表中会同时拥有指向其开头与结尾的指针，这样可以很轻松地将每个链接添加到前一列表的末尾，而无需遍历该列表。因此，在每次执行该过程中的 for 循环时，只需要为一个指针变量分配一个地址，就能将一个列表添加到 masterlist 的最后。我们可以将这一赋值看作基本运算。在过程 distribute 中，可以将 while 循环中的部分或全部指令看作基本运算。因此，为与 coalesce 保持一致，我们选择的基本运算是：为一个指针变量分配一个地址，以向列表末尾添加一个键。

**输入规模：** $n$ ，masterlist 中的整数个数； $\text{numdigits}$ ，每个整数中的最大十进制位数。

遍历整个 masterlist 时，总是需要将 distribute 中的 while 循环执行  $n$  遍。将所有列表添加到 masterlist 时，总是需要将 coalesce 中的 for 循环执行 10 遍。radixsort 会将两个过程分别调用  $\text{numdigits}$  次。因此，

$$T(\text{numdigits}, n) = \text{numdigits}(n+10) \in \Theta(\text{numdigits} \times n)$$

这不是一个  $\Theta(n)$  算法，因为它的界限同时与  $\text{numdigits}$  和  $n$  有关。只要使  $\text{numdigits}$  任意大，就能创建一个关于  $n$  的任意大时间复杂度。例如，因为 1 000 000 000 共有 10 位，所有如果 10 个数中的最大数为 1 000 000 000，那对这 10 个数字进行排序将需要  $\Theta(n^2)$  的时间。在实践中，数位的个数通常远小于数字的个数。例如，如果正

在对 1 000 000 个社会保障号码排序，则  $n$  为 1 000 000，而 numdigits 仅为 9。不难看出，当键互不相同时，基数排序的最佳时间复杂度属于  $\Theta(n \lg n)$ ，通常我们可以达到这一最佳情景。

接下来分析基数排序使用的额外空间。

#### 算法 7.6 的分析 额外空间使用量的分析（基数排序）

在此算法中，从来不会分配新的节点，因为从来不会在 masterlist 和代表一个堆的列表中同时用到一个键。这意味着，唯一的额外空间就是最初在链表中表示键时所需要的空间。因此，其额外空间属于  $\Theta(n)$  个链接。这里所说的“属于  $\Theta(n)$  个链接”是指链接数属于  $\Theta(n)$ 。

## 7.10 习题

### 7.1 节和 7.2 节

- 实现插入排序算法（算法 7.1），在你的系统上运行它，并使用几个问题实例研究它的最佳情况、平均情况和最差情况时间复杂度。
- 证明：在以非递减顺序输入各个键时，插入排序算法（算法 7.1）执行的比较次数最多。
- 证明：插入排序算法（算法 7.1）执行的记录赋值数的最差和平均情况时间复杂度分别为：

$$W(n) = \frac{(n+4)(n-1)}{2} \approx \frac{n^2}{2} \quad A(n) = \frac{n(n+7)}{4} - 1 \approx \frac{n^2}{4}$$

- 证明：交换排序算法（算法 1.3）执行的记录赋值数的最差和平均情况时间复杂度分别为：

$$W(n) = \frac{3n(n-1)}{2} \quad A(n) = \frac{3n(n-7)}{4}$$

- 比较交换排序（算法 1.3）和插入排序（算法 7.1）的最佳情况时间复杂度。
- 如果需要在一个包含  $n$  个键的列表中按照非递减顺序找出第  $k$  大的键（或按照非递减顺序找出第  $k$  小的键），交换排序（算法 1.3）或插入排序（算法 7.1）是否更为合适？
- 将插入排序算法（算法 7.1）改写如下。包含一个额外的数组位置  $S[0]$ ，其取值小于所有键。这样就不需要再在 while 循环的顶端将  $j$  与 0 进行比较。计算该算法这一版本的准确时间复杂度。与算法 7.1 的时间复杂度相比，是较好还是较差？哪种版本的效率更高？说明理由。
- 插入排序能够利用列表中的元素顺序，受此启发，出现了一种称为 Shell 排序的算法。在 Shell 排序中，整个列表被分为不连续的子列表，它们的元素之间存在距离  $h$  ( $h$  为某一数字)。然后使用插入排序法对每个子列表排序。在下一轮中，减少  $h$  值，增大每个子列表的大小。在选择每个  $h$  值时，通常会使它与前一个值互质。在对列表进行最后一轮排序时，取  $h$  值为 1。写出 Shell 排序的算法，研究其性能，并将结果与插入排序的性能进行对比。

### 7.3 节

- 证明：排列  $[n, n-1, \dots, 2, 1]$  有  $n(n-1)$  个倒置。
- 给出排列  $[2, 5, 1, 6, 3, 4]$  的转置，并求出两个排列的倒置个数。倒置的总数为多少？
- 证明：在一个包含  $n$  个不同有序元素的排序中，相对于其转置共有  $n(n-1)/2$  个倒置。
- 证明：一个排列及其转置中的倒置总数为  $n(n-1)/2$ 。利用此结果求出第 10 题中的排列及其转置的倒置总数。

### 7.4 节（也见 2.2 节的习题）

- 实现 2.2 节和 7.4 节讨论的不同合并排序算法，在你的系统上运行它们，并使用几个问题实例研究它们的最佳、平均和最差性能。
- 证明：合并算法（算法 2.2 和算法 2.4）的记录赋值数的时间复杂度近似为  $T(n)=2n\lg n$ 。
- 编写一个原地线性时间算法，它以合并排序 4 算法（算法 7.4）构建的链表作为输入，并根据记录的键值，以非递减顺序将记录存储在连续的数组位置中。

16. 使用分而治之方法编写非递归合并排序算法。分析你的算法，并用阶的符号给出分析结果。注意，有必要在算法中显式维护一个栈。
17. 实现第 16 题的非递归合并排序算法，使用第 13 题的问题实例在你的系统上运行它，并将其结果与第 13 题合并排序的递归版本的结果进行比较。
18. 编写 mergesort3（算法 7.3）的一个版本，和 merge3 的相应版本，在每次执行 repeat 循环时，颠倒两个数组  $S$  和  $U$  的角色。

### 7.5 节（也见 2.4 节的习题）

19. 实现 2.4 节讨论的快速排序算法（算法 2.6），在你的系统上运行它，并使用几个问题实例研究它的最佳、平均和最差性能。
20. 证明：快速排序算法执行的平均交换次数的时间复杂度近似为  $0.69(n+1)\lg n$ 。
21. 编写一个非递归快速排序算法。分析你的算法，并用阶的符号给出分析结果。注意，有必要在算法中显式维护一个栈。
22. 实现第 21 题的非递归快速排序算法，使用第 20 题用到的相同问题实例，在你的系统上运行它，并将其结果与第 20 题快速排序递归版本的结果进行比较。
23. 下面是过程 partition 的一个更快速版本，由过程 quicksort 调用。

```

void partition (index low, index high, index& pivotpoint)
{
    index i, j; keytype pivotitem;

    pivotitem = S[low]; i = low; j = high + 1;
    do
        i++;
    while (i < high && S[i] <= pivotitem);
    do
        j--;
    while (S[j] > pivotitem);
    while (i < j){
        交换 S[i] 和 S[j];
        do
            i++;
        while (S[i] <= pivotitem);
        do
            j--;
        while (S[j] > pivotitem);
    }
    pivotpoint=j;
    交换 S[low] 和 S[pivotpoint];           // 将 pivotitem 放在 pivotpoint.
}

```

证明：利用这一 partition 过程，就记录赋值个数而言，快速排序的时间复杂度为：

$$A(n) \approx 0.69(n+1)\lg n$$

进一步证明，键比较次数的平均情况时间复杂度与上式大体相等。

24. 给出两个最适于采用快速排序算法的实例。
25. 另一种通过交换乱序键来对列表进行排序的方法称为冒泡排序。冒泡排序扫描相邻的一对记录，如果它们的键序不对，则交换它们。在第一轮扫描列表之后，具有最大键（或最小键）的记录将被移到其正确位置。对列表中的剩余无序部分重复这一过程，直到列表变为完全有序为止。编写此冒泡排序算法。分析你的算法，并使用阶的符号给出分析结果。将冒泡排序算法的性能与插入排序、交换排序和选择排序的性能进行比较。

### 7.6 节

26. 编写一个算法，检查一个准完全二叉树是不是堆。分析你的算法，并用阶的符号给出分析结果。
27. 证明：在一个拥有  $n$  个节点（ $n$  是 2 的幂）的堆中，当  $j < d$  时（ $d$  是堆的深度），共有  $2^j$  个节点的深度为  $j$ 。
28. 证明：一个拥有  $n$  个节点的堆中有  $\lceil n/2 \rceil$  个叶节点。
29. 实现堆排序算法（算法 7.5），在你的系统上运行它，并使用几个问题实例研究它的最佳、平均和最差性能。
30. 证明：存在一种堆排序情景，可以得到最差时间复杂度  $W(n) \approx 2n\lg n \in \Theta(n\lg n)$ 。
31. 证明：就记录的赋值次数而言，堆排序的最差时间复杂度大约为  $W(n) \approx n\lg n$ 。
32. 修改堆排序，在根据非递减顺序找到第  $k$  大的键后停止。分析你的算法，并使用阶的符号给出这些结果。

### 7.7 节

33. 根据键的比较次数和记录赋值次数，列出本章讨论的所有排序算法的优缺点。
34. 使用几个问题实例，在你的系统上运行本章讨论的所有排序算法的实现。利用这些结果和第 33 题提供的信息，给出这些排序算法的详尽比较。
35. 针对下面的各个列表排序情景，你会从选择排序、插入排序、合并排序、快速排序和堆排序中选择哪一种算法？说明理由。
  - (a) 列表有数百条记录。这些记录很长，但键很短。
  - (b) 列表有大约 45 000 条记录。需要在所有情况下都能快速完成排序。内存容量刚好能够容纳 45 000 条记录。
  - (c) 列表中有大约 45 000 条记录，但在开始时只有少许乱序。
  - (d) 列表中有大约 25 000 条记录。希望在平均情况下能够尽可能快速地完成排序，但并不强制要求在每个具体情况下能够快速完成排序。
36. 为本章讨论的每种排序算法给出至少两个问题实例，在这些问题实例中，该算法不是最恰当的排序选择（就键的比较次数而言）。

### 7.8 节

37. 编写一个线性排序算法，对整数 1 至  $n$ （含）的一个排列进行排序。（提示：使用一个  $n$  元素数组。）
38. 第 37 题中的算法具有线性时间特性，它是否违背了仅通过键的比较进行排序的下限？说明理由。
39. 证明引理 7.9 的一般情景，即叶节点数  $m$  不是 2 的幂时的情景。

### 7.9 节

40. 实现基数排序算法（算法 7.6），在你的系统上运行它，并使用几个问题实例研究它的最佳、平均和最差性能。
41. 证明：当所有键各不相同时，基数排序（算法 7.6）的最佳时间复杂度属于  $\Theta(n\lg n)$ 。
42. 在重建主列表的过程中，当堆数（基数）很大时，基数排序算法（算法 7.6）在检查空子列表方面浪费了大量的时间。是否有可能仅检查不为空的子列表？

### 补充习题

43. 编写一个将  $n$  元素列表排列为非递减顺序的算法，具体做法是：找出最大和最小元素，并将它们与第一个位置和最后一个位置处的元素交换。然后将列表大小减 2，排除已经处于正确位置的两个元素，并对剩下的列表部分重复该过程，直到整个列表都变为有序。分析你的算法，并用阶的符号给出分析结果。
44. 实现快速排序算法，使用不同算法选择枢纽项，在你的系统上运行该算法，并使用几个问题实例研究不同策略下的最佳、平均和最差性能。
45. 研究基于一个三叉堆设计排序算法的思路。三叉堆与普通的堆类似，只是每个内部节点有三个子节点。
46. 假定我们要找出一个  $n$  元素列表中的  $k$  个最小元素，而对它们的相对顺序不感兴趣。当  $k$  为常数时，能否找出一种线性时间算法？说明理由。
47. 假定有一个非常大的列表需要排序，它存储在外面存储器中。假定这个列表太大，无法放入内部存储器中，

在设计一种外部排序算法时，应当考虑哪些主要因素？

48. 根据算法背后的思想，对本章讨论的排序算法进行分类。例如，堆排序和选择排序是找出最大（或最小）键，并将它与符合所需顺序的最后元素（或第一个）元素相交换。
49. 稳定排序算法是指使相同键保持原顺序的算法。本章讨论的哪种排序算法是稳定的？哪种是不稳定的？说明理由。
50. 在第 49 题中被认定为不稳定的排序算法中，哪一种可轻松修改为稳定排序算法？

# 再谈计算复杂度：查找问题

我想知道有没有一种更快的方法可以找到Colleen的电话号码



回想在第 1 章的开头，Barney Beagle 可以使用一种经过修改的二分查找法，快速找出 Colleen Collie 的电话号码。Barney 现在可能想知道，能不能设计一种更快的方法来查找 Colleen 的号码。我们接下来将分析查找问题，以确定是否有此可能。

和排序一样，查找也是计算机科学领域最有用的应用之一。待解决的问题通常是根据某一键字段的值提取整条记录。例如，一条记录可能包括个人信息，而键字段可能是社会保障号。我们这里的目的与前一章类似，希望分析查找问题，并证明已经获得了时间复杂度几乎接近下限的查找算法。此外，还希望讨论这些算法使用的数据结构，以及一种数据结构何时满足一种特定应用的需要。

8.1 节将给出仅通过键的比较在一个数组中查找键时的复杂度下限（就像上一章针对排序所做的工作一样），并证明二分查找（算法 1.5 和算法 2.1）的时间复杂度达到了这些下限。在查找电话号码时，Barney Beagle 实际使用了二叉查找的一种修改形式，称为“插值查找”，它所做的工作不仅限于键的对比。也就是说，在查找 Colleen Collie 的号码时，Barney 不是从电话簿的中间开始查找，因为他知道 C 打头的名字靠近前端。他“进行插值”，从电话簿的开头附近开始查找。8.2 节将介绍插值查找。8.3 节将展示数组不满足特定应用（除查找之外）的某些其他需要。因此，尽管二分查找是最优的，但在某些应用中并不能使用这一算法，因为它是以数组实现为基础的。我们会表明树满足这些需求，并讨论树的查找。有些情况下，并不要求按照某一特定顺序来提取数据，8.4 节关注的就是这种情况下的查找。我们将在 8.4 节讨论散列。8.5 节讨论一种不同的查找问题——选择问题。这个问题是在一个包含  $n$  个键的列表中找出第  $k$  小（或第  $k$  大）的键。8.5 节将介绍“对手论证”（adversary argument），利用这种方式，也可以获得解决同一问题的所有算法的性能下限。

## 8.1 仅通过键的比较进行查找的下限

查找键的问题可以描述如下：给定一个包含  $n$  个键的数组  $S$  和一个键  $x$ ，如果  $x$  等于这些键中的一个，则找出一个索引  $i$ ，使得  $x=S[i]$ ；如果  $x$  与所有这些键均不相等，则报告失败。

当数组有序时，二分查找（算法 1.5 和算法 2.1）可以非常高效地解决这一问题。回想一下，它的最差时间复杂度为  $\lfloor \lg n \rfloor + 1$ 。能否改进这一性能呢？后面将会看到，只要是仅通过比较键来进行查找的算法，这种改进就是不可能的。仅通过键的比较在数组中查找一个键  $x$  的算法可以将这些键相互对比，也可以将它与要查找的键  $x$  进行对比，还可以复制键，但不能对它们执行其他操作。然而，为了帮助查找，可以利用“数组有序”这一信息（就像二分查找中的做法一样）。和第 7 章的做法一样，我们将获取确定性算法的下限。我们得到的结果对于概率算法同样成立。此外，和第 7 章中一样，假定数组中的键互不相同。

和前面对确定性排序算法的做法一样，对于每个在  $n$  键数组中查找键  $x$  的确定性算法，都可以为其关联一棵决策树。图 8-1 给出了一棵决策树，对应于查找七个键的二分查找法，而图 8-2 给出的则是与顺序查找（算法 1.1）对应的决策树。在这些树中，每个大节点都表示将一个数组项与一个查找键  $x$  进行比较，每个小节点（叶节点）都包含了一个报告结果。当  $x$  在数组中时，会报告与它相等的项目的索引；当  $x$  不在数组时，则报告一个“F”，表示失败。在图 8-1 和图 8-2 中， $s_1$  至  $s_7$  是满足以下各式的值：

$$S[1]=s_1, S[2]=s_2, \dots, S[7]=s_7$$

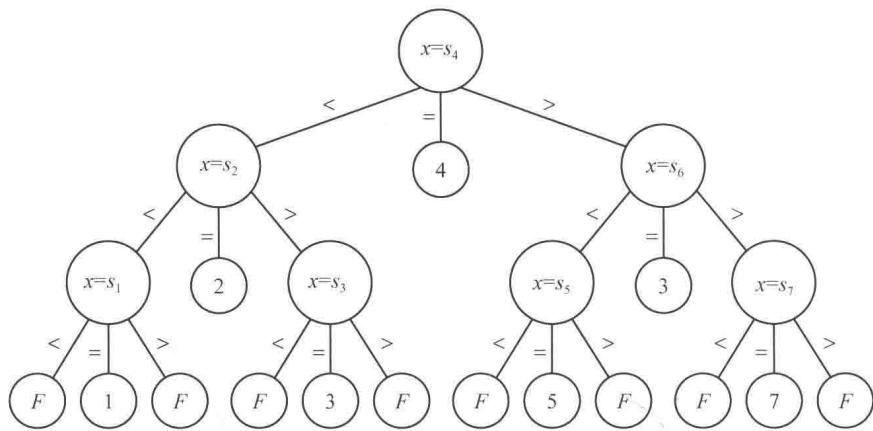


图 8-1 与查找七个键的二分查找法相对应的决策树

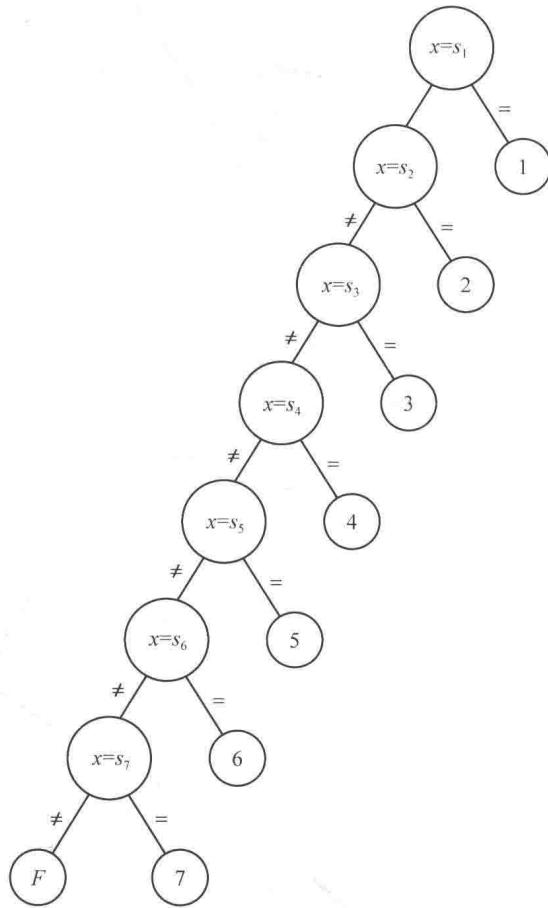


图 8-2 与查找七个键的顺序查找法相对应的决策树

我们假定查找算法绝对不会改变任何数组值，所以在完成查找之后，仍然是这些值。

对于在  $n$  个键中查找一个键  $x$  的决策树，其中的每个叶节点都表示一个时刻，算法在此时刻停下来，报告一个满足  $x=s_i$  的索引  $i$ ，或者报告失败。每个内部节点都表示一次比较。如果对于每个可能出现的输出结果，决策树中都存在一个从根节点到一个叶节点的路径，用来报告该输出结果，就说该决策树对于在  $n$  个键中查找一个键  $x$  是有效的（valid）。也就是说，必然存在满足  $x=s_i$  ( $1 \leq i \leq n$ ) 的路径，还有一条引向失败的路径。如

果决策树中的每片叶子都是可达到的，就称该树是已修剪的（pruned）。任何一个在  $n$  键数组中查找键  $x$  的算法，都有一个对应的已修剪有效决策树。一般情况下，查找算法并不需要总是将  $x$  与数组项进行对比。也就是说，它可以比较两个数组项。但是，因为我们假定所有键都互不相同，所以只有在与  $x$  进行比较时，输出结果才可能为相等。在二分查找和顺序查找中，当算法确定  $x$  等于一个数组项时，它会停止，并返回该数组项的索引。而一种低效算法可能会在确定  $x$  等于一个数组项之后继续进行比较，之后才会返回该索引。我们可以用一种即时停止并返回索引的算法来代替这种低效算法，新算法的效率至少不低于原算法。因此，对于在  $n$  个不同键中查找一个键  $x$  的过程，我们只需要考虑已修剪有效决策树，在这些树中，如果比较结果是相等，将会引向一个返回索引值的叶节点。因为一个比较只有三种可能结果，所以确定性算法在每次比较之后最多取三种不同路径。这意味着相应决策树中的每个比较节点最多有三个子节点。因为比较结果为相等时，必然引向一个返回索引的叶节点，所以最多有两个子节点是比较节点。因此，树中的比较节点集合构成一个二叉树。具体例子，可参见图 8-1 和图 8-2 中的大节点集合。

### 8.1.1 最差表现的下限

因为在已修剪的有效决策树中，每个叶节点都必须是可到达的，所以这样一棵树在最差情况下完成的比较次数，就是在由这些比较节点组成的二叉树中，从根节点到叶节点的最长路径中的节点数。这个数字等于二叉树的深度加 1。因此，要确定最差情况下的比较次数下限，只需要计算一下，在由比较节点构成的二叉树中，其深度下限为多少。通过下面的引理和定理可确定这样一个下限。

**引理 8.1** 如果  $n$  是二叉树中的节点数， $d$  是其深度，则

$$d \geq \lfloor \lg(n) \rfloor$$

**证明：**因为根节点只有一个，最多有两个节点的深度为 1， $2^2$  个节点的深度为 2， $\dots$ ， $2^d$  个节点的深度为  $d$ ，所以有

$$n \leq 1 + 2 + 2^2 + 2^3 + \dots + 2^d$$

应用附录 A 中例 A.3 的结果可得：

$$n \leq 2^{d+1} - 1$$

这意味着：

$$\begin{aligned} n &< 2^{d+1} \\ \lg n &< d + 1 \\ \lfloor \lg n \rfloor &\leq d \end{aligned}$$

下面的定理尽管看起来显而易见，但其严格证明不容易。

**引理 8.2** 一个由比较节点组成的二叉树，要想成为一个已修剪的有效决策树，用于在  $n$  个不同键中查找一个键  $x$ ，必须至少包含  $n$  个节点。

**证明：**设  $s_i$  ( $1 \leq i \leq n$ ) 是  $n$  个键的值。首先证明每个  $s_i$  都必须在至少一个比较节点中（也就是说，它必须至少参与一次比较）。假定对于某个  $i$  值不是如此。取两个输入，除第  $i$  个键不同外，对于所有其他键都是相同的。设  $x$  的取值是其中一个输入中的  $s_i$  值。因为  $s_i$  不参与任何比较，而且两个输入中的所有其他键都相同，所以决策树对于两个输入的表现必然相同。但是，它必须对其中一个输入报告  $i$ ，而对于另一个输入不报告  $i$ 。这一矛盾表明，每个  $s_i$  都必然至少在一个比较节点中。

因为每个  $s_i$  都必然至少在一个比较节点中，所以要想使比较节点数少于  $n$  个，唯一的方法就是将至少一个键  $s_i$  仅参与与其他键的比较，也就是说，有一个  $s_i$  从来不会与  $x$  进行比较。假定的确有这样一个键。取两个输入，除  $s_i$  外，这两个输入完全相等， $s_i$  是两个输入中的最小键。设  $x$  是其中一个输入的第  $i$  个键。从包含  $s_i$  的比较节点引出一条路径，它的走向对于两个输入来说必然相同，而且两个输入中的所有其他键都是相同的。因此，决策树对于这两个输入的表现也必然相同。但是，它必须对其中一个输入报告  $i$ ，而对另一个输入不报告  $i$ 。这

一矛盾证明了该引理。

**定理 8.1** 任何一个仅通过键的比较，在一个包含  $n$  个不同键的数组中查找一个键  $x$  的确定性算法，其在最差情况下至少要进行  $\lfloor \lg n \rfloor + 1$  次键的比较。

证明：与该算法相对应，存在一个经过修剪的有效决策树，用于在  $n$  个不同键中查找一个键  $x$ 。由此决策树中的比较节点组成一棵二叉树，由其根节点到一个叶节点的最长路径中的节点数，就是最差情况下的比较次数。这个数字就是该二叉树的深度加 1。引理 8.2 表明这个二叉树至少有  $n$  个节点。因此，根据引理 8.1，它的深度大于或等于  $\lfloor \lg n \rfloor$ 。此定理得证。

回想 2.1 节中的内容，二叉查找完成的最差比较次数为  $\lfloor \lg n \rfloor + 1$ 。因此，就最差性能而言，二叉查找算法是最优的。

### 8.1.2 平均情况下的下限

在讨论平均情况下的下限之前，先对二叉查找法进行平均情况分析。之所以一直等到现在才进行这一分析，是因为决策树的使用为这一分析提供了方便。首先，我们需要一个定义和一个定理。

如果一个二叉树直到  $d-1$  的深度都是完全的，则称它为近乎完全二叉树（nearly complete binary tree）。每个准完全二叉树都是近乎完全的，但并不是所有近乎完全二叉树都是准完全的，如图 8-3 所示。（关于完全、准完全二叉树的定义，请参见 7.6 节。）

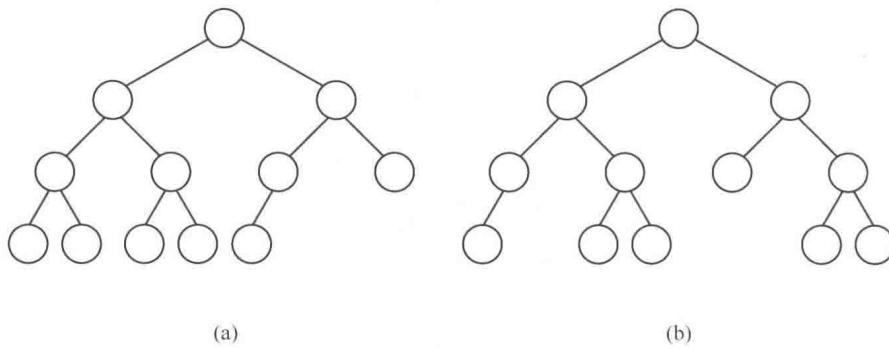


图 8-3 (a) 准完全二叉树；(b) 近乎完全但不是准完全的二叉树

与引理 8.2 类似，下面的引理看起来非常明显，但其严格证明并不容易。

**引理 8.3** 一棵与二分查找对应的已修剪有效决策树，由其中比较节点组成的树是一棵近乎完全二叉树。

证明：此证明通过对键数  $n$  的归纳完成。显然，由这些比较节点组成的树是一棵二叉树，其中包含  $n$  个节点，每个键对应一个。因此，我们针对这个二叉树中的节点数进行归纳。

归纳基础：只有一个节点的二叉树是近乎完全的。

归纳假设：假定对于所有  $k < n$ ，包含  $k$  个节点的二叉树是近乎完全的。

归纳步骤：我们需要证明包含  $n$  个节点的二叉树是近乎完全的。分别对于奇数和偶数  $n$  值进行证明。

如果  $n$  为奇数，二分查找中的第一次划分将数组分为两个大小各为  $(n-1)/2$  的子数组。因此，左右子树都是与二分查找对应的二叉树，只不过是在查找  $(n-1)/2$  个键，这就是说，就结构而言，这些树是相同的。根据归纳假设，它们是近乎完全的。因为它们是相同的近乎完全树，所以以它们作为左右子树的树也是近乎完全的。

如果  $n$  为偶数，二分查找中的第一次划分将数组划分为两个子数组，右侧的大小为  $n/2$ ，左侧为  $(n/2)-1$ 。为了能说得具体一些，我们将讨论左侧有奇数个键的情景。当右侧有奇数个键时，其证明是类似的。当左侧有奇数个键时，左子树的左右子树是相同树（如前文的讨论）。右子树的一个子树也是相同的（你应当验证这一点）。因为右子树是近乎完全的（根据归纳假设），而且因为它的子树之一与左子树的左右子树相同，所以整棵树必然是近乎完全的。图 8-4 给出一个图形表示。

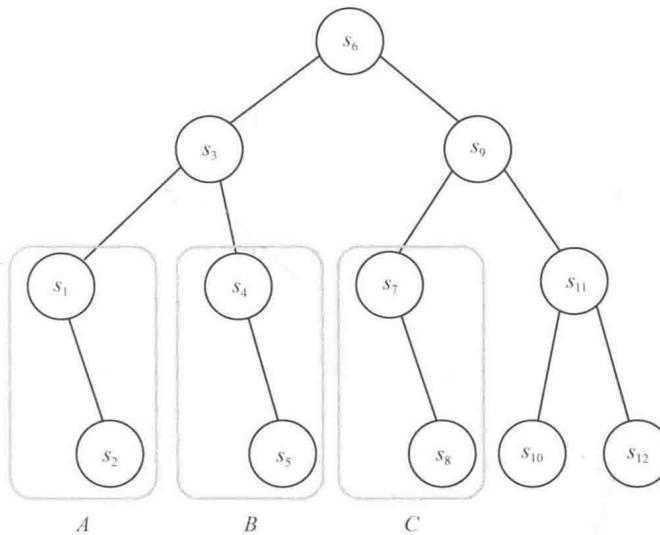


图 8-4 由一棵决策树中的比较节点组成的二叉树，此决策树对应于  $n=12$  时的二叉查找法。  
节点中仅给出了键的值。子树  $A$ 、 $B$  和  $C$  均具有相同的结构

现在可以对二分查找进行平均情况分析了。

**算法 8.1 的分析 平均情况时间复杂度（二分查找，递归）**

基本运算： $x$  与  $S[mid]$  的比较。

输入规模： $n$ ，数组中的项数。

首先分析  $x$  在数组中的情景。进行这一分析时，假定  $x$  位于每个数组位置的概率相等。从根节点到某一节点的路径上有若干节点，这一节点数称为该节点的节点距离（node distance），所有节点的节点距离之和称为树的总节点距离（TND，Total Node Distance）。注意，到一个节点的距离等于根节点到该节点的路径长度加 1。对于由图 8-1 中比较节点组成的二叉树，

$$TND = 1 + 2 + 2 + 3 + 3 + 3 + 3 = 17$$

不难看出，为了从所有可能数组位置处找出  $x$ ，二分查找的决策树完成的总比较次数等于一个二叉树中的 TND，此二叉树由该决策树中的比较节点组成。假定所有数组位置的可能性相等，则找出  $x$  所需要的平均比较数为  $TND/n$ 。为简单起见，在开始时假定  $n=2^k-1$ ， $k$  为某一整数。对于与二分查找法对应的决策树，由其中的比较节点组成一棵二叉树，根据引理 8.3，此二叉树是近乎完全的。不难看出，如果一棵近乎完全二叉树中包含  $2^k-1$  个节点，它就是一棵完全二叉树。由图 8-1 中比较节点组成的二叉树演示了  $k=3$  时的情景。在一棵完全二叉树中，TND 由下式给出：

$$\begin{aligned} TND &= 1 + 2(2) + 3(2^2) + \cdots + k(2^{k-1}) \\ &= \frac{1}{2}[(k-1)2^{k+1} + 2] = (k-1)2^k + 1 \end{aligned}$$

倒数第二个等式是应用附录 A 中例 A.5 的结果得出的。因为  $2^k=n+1$ ，所以平均比较次数为：

$$A(n) = \frac{TND}{n} = \frac{(k-1)(n+1)+1}{n} \approx k-1 = \lfloor \lg n \rfloor$$

对于一般的  $n$  值，平均比较次数的上下限近似为：

$$\lfloor \lg n \rfloor - 1 \leq A(n) \leq \lceil \lg n \rceil$$

如果  $n$  是 2 的幂，或者稍大于 2 的一个幂，则此平均值接近下限；如果  $n$  稍小于 2 的一个幂，则此平均值

接近上限。习题中将向你说明如何推导这一结果。图 8-1 直观地展示了为什么是这样。如果仅增加一个节点，使总节点数为 9，那  $\lfloor \lg n \rfloor$  就由 2 跳到 3，但平均比较数几乎不变。

接下来分析  $x$  可能不在数组中的情景。共有  $2n+1$  种可能： $x$  可能小于所有项目，介于任意两个项目之间，或者大于所有项目。也就是说，可有以下各种情景：

$$\begin{aligned}x &= s_i \quad (1 \leq i \leq n) \\x &< s_1 \\s_i &< x < s_{i+1} \quad (1 \leq i \leq n-1) \\x &> s_n\end{aligned}$$

我们分析这些概率相等的情景。为简单起见，再次假设  $n=2^k-1$  ( $k$  为某一整数)。查找成功时的总比较次数就是由比较节点组成的二叉树的 TND。回想一下，这个数字等于  $(k-1)2^k+1$ 。对于  $n+1$  个失败查找中的每次查找，共进行  $k$  次比较（见图 8-1）。因此，平均比较次数为：

$$A(n) = \frac{\text{TND} + k(n+1)}{2n+1} = \frac{(k-1)2^k + 1 + k(n+1)}{2n+1}$$

因为  $2^k=n+1$ ，所以有

$$\begin{aligned}A(n) &= \frac{(k-1)(n+1) + 1 + k(n+1)}{2n+1} \\&= \frac{2k(n+1) + 1 - (n+1)}{2n+1} \\&\approx k - \frac{1}{2} = \lfloor \lg n \rfloor + 1 - \frac{1}{2} = \lfloor \lg n \rfloor + \frac{1}{2}\end{aligned}$$

对于一般的  $n$  值，平均比较次数的上下限近似为：

$$\lfloor \lg n \rfloor - \frac{1}{2} \leq A(n) \leq \lfloor \lg n \rfloor + \frac{1}{2}$$

如果  $n$  是 2 的幂，或者稍大于 2 的一个幂，则此平均值接近下限；如果  $n$  稍小于 2 的一个幂，则此平均值接近上限。习题中将要求你推导这一结果。

二分查找的平均性能比其最差性能好不了太多。再次查看图 8-1 就可看出其中缘由。在该图中，树底部的结果节点（在最差情况下发生）要多于树中其余部分的结果节点。即使在没有考虑失败查找时，这一点也是成立的。（注意，所有未成功的查找都在树的底部。）

接下来证明，在给定前一分析的假设条件时，二分查找在平均情况下是最优的。首先，将  $\text{minTND}(n)$  定义为  $n$  节点二叉树的最小 TND。

**引理 8.4** 当且仅当一个  $n$  节点二叉树为近乎完全时，它的 TND 等于  $\text{minTND}(n)$ 。

**证明：**首先证明，若一棵树的  $\text{TND}=\text{minTND}(n)$ ，则这棵树是近乎完全的。为此，假定某个二叉树不是近乎完全的。那必然存在某一节点，它不在底部两级，且最多有一个子节点。我们可以从最底层删除任意节点  $A$ ，使它成为前述节点的一个子节点。这样得到的树仍然是包含  $n$  个节点的二叉树。这棵树中通向  $A$  的路径中的节点数，至少要比原树中通向  $A$  的路径中的节点数小 1。指向所有其他节点的路径中的节点数都是相同的。这样，我们就生成了一个包含  $n$  个节点的二叉树，它的 TND 要小于原来的数，也就是说，原来的树不具备最小 TND。

不难看出，对于所有包含  $n$  个节点的近乎完全二叉树，这个 TND 都是相同的。因此，每个此种树都必然具有最小 TND。

**引理 8.5** 假定我们正在查找  $n$  个键，查找键  $x$  在数组中，而且所有数组位置的概率相同。因此，二分查找的平均情况时间复杂度为：

$$\frac{\min TND(n)}{n}$$

证明：如二分查找平均情况分析中的讨论过程，由相应决策树中的比较节点组成一个二叉树，将其 TND 除以  $n$ ，即可得到平均情况时间复杂度。由引理 8.3 和引理 8.4 可得出此证明。

**引理 8.6** 如果假定  $x$  在数组中，并且出现在所有数组位置的概率相同，要在一个由  $n$  个不同键组成的数组中查找一个键  $x$ ，任何此种确定性算法的平均情况时间复杂度具有如下下限：

$$\frac{\min TND(n)}{n}$$

证明：如引理 8.2 所示，在与该算法对应的决策树中，每个数组项  $s_i$  都必须与  $x$  至少进行一次比较。有一个节点中包含了  $s_i$  与  $x$  的比较，设  $c_i$  是指向该节点的最短路径中的节点数。因为每个键等于搜索键  $x$  的概率相同，均为  $1/n$ ，所以平均情况时间复杂度的下限为：

$$c_1\left(\frac{1}{n}\right) + c_2\left(\frac{1}{2}\right) + \dots + c_n\left(\frac{1}{n}\right) = \frac{\sum_{i=1}^n c_i}{n}$$

最后一个表达式中的分子大于或等于  $\min TND(n)$ ，其证明留作练习。

**定理 8.2** 要在一个由  $n$  个不同键组成的数组中查找键  $x$ ，如果假定  $x$  在数组中，且所有数组位置的概率相等，则在此类确定性算法中，二分查找方法是最优的。因此，在这些假设条件下，任何此类算法平均来说，至少要进行大约  $\lfloor \lg n \rfloor - 1$  次键的比较。

证明：由引理 8.5 和引理 8.6 及二分查找的平均情况时间复杂度分析可以得证。

还可能证明，如果假定所有  $2n+1$  种可能结果（如在对二分查找进行平均情况分析时的讨论）是等概率的，则二分查找的平均性能也是最优的。

我们是基于特定概率分布假设，证明了二分查找的平均性能是最优的。对于其他概率分布，它可能不是最优的。例如，如果  $x$  等于  $S[1]$  的概率为 0.9999，那首先将  $x$  与  $S[1]$  进行对比将是最快的。这个例子的人为色彩有些重了。一个更现实的例子是查找一个从美国人中随机选择的名字。如 3.5 节中的讨论，我们不会认为名字“Tom”和“Ursula”是等概率的。那么，这里所做的分析就不再适用，而是需要考虑其他一些因素。3.5 节讨论了其中一些因素。

## 8.2 插值查找

前面获得的界限适用于那些仅通过键的比较进行查找的算法。如果有某些其他信息可用于帮助查找，我们就可以改进这些界限。回想一下，Barney Beagle 在电话簿中查找 Colleen Collie 的号码时，并不只是比较了几个键。他没有从电话簿的中间开始查找，因为他知道 C 打头的姓名靠近前部。他在前端附近“插入”并开始查找。下面将为这一策略开发一种算法。

假定正在查找 10 个整数，并且知道第一个整数的范围为 0 到 9，第二个为 10 到 19，第三个为 20 到 29……第 10 个为 90 到 99。如果查找键  $x$  小于 0 或大于 99，则立即报告查找失败；如果不是这两种情况，则只需要将  $x$  与  $S[1+\lfloor x/10 \rfloor]$  进行比较。例如，我们将  $x=25$  与  $S[1+\lfloor 25/10 \rfloor]$  进行比较。如果它们不相等，则报告查找失败。

我们通常没有这么多信息。但是，在某些应用中，可以合理地假定这些键近似平均分布在第一个键和最后一个键之间。在这种情况下，我们不去检查  $x$  是否等于中间键，而是先预测会在哪里找到  $x$ ，然后检查该位置的键是否与  $x$  相等。例如，如果有 10 个键近似平均分布在 0 到 99 之间，我们预期会在第 3 个位置找到  $x=25$ ，那就首先将  $x$  与  $S[3]$  进行比较，而不是与  $S[5]$  比较。实现这一策略的算法称为插值查找（Interpolation Search）。和二分查找一样， $low$  在开始时被设置为 1， $high$  被设置为  $n$ 。然后使用线性插值来确定大约会在哪里找到  $x$ 。

也就是说，计算

$$\text{mid} = \text{low} + \left\lfloor \frac{x - S[\text{low}]}{S[\text{high}] - S[\text{low}]} \times (\text{high} - \text{low}) \right\rfloor$$

例如，如果  $S[1]=4$ ,  $S[10]=97$ , 而且正在查找  $x=25$ , 则

$$\text{mid} = 1 + \left\lfloor \frac{25 - 4}{97 - 4} \times (10 - 1) \right\rfloor = 1 + \lfloor 2.032 \rfloor = 3$$

除了计算 mid 的方式不同，以及还有其他一些额外的记账工作之外，插值查找算法的过程与二分查找（算法 1.5）类似。

### 算法 8.1 插值查找

问题：确定  $x$  是否在一个大小为  $n$  的有序数组  $S$  中。

输入：正整数  $n$ ; 有序（非递减顺序）数字数组  $S$ , 其索引范围为 1 至  $n$ 。

输出： $x$  在  $S$  中的位置  $i$ , 如果  $x$  不在  $S$  中，则为 0。

```
void interpsrch (int n,
                  const number S[],
                  number x, index& i)
{
    index low, high, mid;
    number denominator;

    low = 1; high = n; i = 0;
    if (S[low] <= x <= S[high])
        while (low <= high && i == 0){
            denominator = S[high]-S[low];
            if (denominator == 0)
                mid = low;
            else
                mid = low + [(x-S[low])*(high-low)]/denominator;
            if (x == S[mid])
                i = mid;
            else if (x < S[mid])
                high = mid - 1;
            else
                low = mid + 1;
        }
}
```

如果这些键之间的间隔近似相等，插值查找可以比二分查找更快速地找出  $x$  的可能位置。例如，在上面的例子中，如果  $x=25$  小于  $S[3]$ ，插值查找会将规模为 10 的实例编写为规模为 2 的实例，而二分查找会将它缩减为规模为 4 的实例。

假定这些键均匀分布在  $S[1]$  和  $S[n]$  之间。也就是说，一个随机选定的键处于一个特定范围的概率，等于它处于另一个等长范围的概率。如果确实如此，那可以预期在插值查找确定的大体位置能找到  $x$ ，从而可以预期插值查找的平均性能优于二分查找。事实上，如果假设键为均匀分布，而且查找键  $x$  出现在每个数组位置的概率相等，那就可能证明，插值查找的平均情况时间复杂度为：

$$A(n) \approx \lg(\lg n)$$

如果  $n$  等于 10 亿，则  $\lg(\lg n)$  大约为 5，而  $\lg n$  大约为 30。

插值查找的一个缺点是它的最差性能。再次假设有 10 个键，它们的值为 1, 2, 3, 4, 5, 6, 7, 8, 9, 100。如果  $x$  为 10,  $mid$  将被重复设置为  $low$ ,  $x$  将与每个键进行比较。在最差情况下，插值查找退化为顺序查找。注意，当  $mid$  被重复设置为  $low$  时会发生最差情况。插值查找有一种称为强壮插值查找（Robust Interpolation Search）的变体，它确定了一个变量  $gap$ ，使得  $mid-low$  和  $high-mid$  总是大于  $gap$ ，从而弥补了上述情景。

最初设定：

$$\text{gap} = \lfloor (\text{high} - \text{low} + 1)^{1/2} \rfloor$$

并利用前面的线性插值公式计算 mid。在计算之后，可以通过以下计算修改 mid 的值：

$$\text{mid} = \min(\text{high} - \text{gap}, \max(\text{mid}, \text{low} + \text{gap}))$$

在前面  $x=10$  且 10 个键分别为 1, 2, 3, 4, 5, 6, 7, 8, 9 和 100 的例子中，gap 最初被设置为  $\lfloor (10-1+1)^{1/2} \rfloor = 3$ ，mid 最初为 1，我们将得到：

$$\text{mid} = \min(10-3, \max(1, 1+3)) = 4$$

这样就可以保证，用于比较的索引至少与 low 和 high 相距 gap 个位置。当继续在包含较数组元素的子数组中查找  $x$  时，我们将 gap 值加倍，但它从来不会大于该子数组中元素个数的一半。例如，在前面的例子中，将继续在较大子数组（从  $S[5]$  至  $S[10]$ ）中查找  $x$ 。因此，我们将使 gap 加倍，只是在这种情况下，该子数组仅包含 6 个数组元素，gap 加倍后，会使它超过子数组中元素个数的一半。我们将 gap 加倍是为了快速退出较大的元素聚集部分。如果发现  $x$  位于仅有较少元素的子数组中，则将 gap 重置为其初始值。在假设这些键为均匀分布，而且查找键  $x$  出现在各数组位置的概率相等时，强壮插值查找的平均情况时间复杂度为属于  $\Theta(\lg(\lg n))$ 。它的最差时间复杂度属于  $\Theta((\lg n)^2)$ ，要劣于二分查找，但远优于插值查找。

强壮插值查找相对于插值查找，插值查找相对于二分查找，都会增加不少计算量。在实践中，应当具体分析一下，是否值得以计算量的增加来换取比较次数的减少。

这里介绍的查找方法也适用于单词，因为可以很轻松地将单词编码为数字。因此，可以应用经过修改的二分查找方法来翻查电话簿。

## 8.3 树中的查找

接下来将会讨论，尽管二分查找及其变体、插值查找和强壮插值查找都是非常高效的，但在许多应用情景中都无法使用，因为在这些应用情景中，不适合采用数组作为数据存储结构。后面将会介绍，这些应用情景中适合采用树。此外，我们还会说明，可以使用一些  $\Theta(\lg n)$  算法来查找树。

**静态查找** (static searching) 是指将所有记录一次性加到文件中，之后不再需要添加可删除记录。适合采用静态查找的一种情景是操作系统命令执行的查找。但许多应用需要**动态查找** (dynamic searching)，也就是说要频繁添加和删除记录。飞机订票系统就是一种需要动态查找的应用程序，因为客户经常会打电话要求预订和取消预订。

数组结构不适于动态查找，因为在向一个有序数组中按顺序添加记录时，必须移动所加记录之后的所有记录。二分查找要求数组中的键必须具有某种结构，因为必须有一种可以快速找出中间项的高效方法。这意味着不会将二分查找法用于动态查找。尽管利用链接可以轻松地添加和删除记录，却没有一种查找链表的高效方法。因此，链接不能满足动态查找应用的查找需要。如果需要从有序序列中快速提取键，直接访问存储（散列）也无能为力（散列将在下一节讨论）。但是，利用树结构可以高效实现动态查找。我们首先讨论二叉查找树，然后讨论 B 树，它是对二叉查找树的一种改进。B 树会一直保持平衡。

我们现在是要进一步分析查找问题，因此，只是简单地讨论与二叉查找树和 B 树有关的算法。这些算法在许多数据结构教科书都有详细讨论，比如 Kruse (1994 年) 的著作。

### 8.3.1 二叉查找树

二叉查找树曾在 3.5 节介绍过，但当时的目的是为了讨论一种静态查找应用。也就是说，希望根据查找键的概率来创建一棵最优树。构建该树的算法（算法 3.9）要求一次性加入所有键，也就是说，这种应用需要静态查找。二叉查找树也适用于动态查找。利用一个二叉查找树，通常可以使平均查找时间保持较低水平，而且能够快速添加和删除键。此外，通过对树的先序遍历，可以快速提取有序序列中的键。回想一下，在对二叉树

进行先序遍历（in-order traversal）时，首先使用先序遍历访问左子树中的所有节点，然后访问根节点，最后使用先序遍历访问右子树中的所有节点。

图 8-5 给出了一棵包含前七个整数的二叉查找树。查找算法（算法 3.8）在查找树时，将查找键  $x$  与根节点处的值进行比较。如果相等，则完成查找；如果  $x$  较小，则向左子节点应用该查找策略；如果  $x$  较大，则向右子节点应用该查找策略。以这种方式向树的下方进行，直到找出  $x$ ，或判定  $x$  不在树中。你可能已经注意到，在将这一算法应用于图 8-5 的树时，所执行的比较序列与二分查找决策树（见图 8-1）执行的比较序列相同。这表明了二分查找和二叉查找树之间的一种基本关系。也就是说，如果一棵决策树对应于查找  $n$  个键的二分查找算法，则其中的比较节点也表示一棵二叉查找树，算法 3.8 在此树中执行的比较与二分查找完全相同。因此，在将算法 3.8 应用于此树时，它和二分查找一样，对于查找  $n$  个键也是最优的。

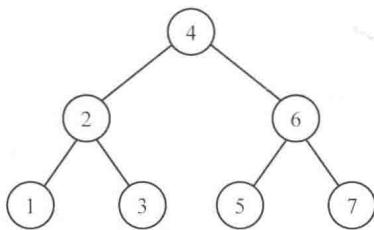


图 8-5 包含前七个整数的二叉查找树

在图 8-5 的树中，可以非常高效地添加键和删除键。例如，要添加键 5.5，只需要沿树向下移动，如果 5.5 大于一个给定节点处的键，就向右移动，否则向左移动，直到找出包含 5 的叶节点为止。然后加入 5.5，将其作为这个叶节点的右子节点。前面曾经提到，在 Kruse (1994 年) 的文献中可以找到用于完成添加和删除的实际算法。

二叉查找树的缺点是在动态添加和删除键时，不能保证所得到的树是平衡的。例如，如果这些键都是按照递增序列添加的，就会得到图 8-6 中的树。这种树称为倾斜树（skewed tree），就是一个链表。将算法 3.8 应用于这种树时，产生的效果就是一种顺序查找。在这种情况下，用二叉查找树代替链表不会有任何好处。

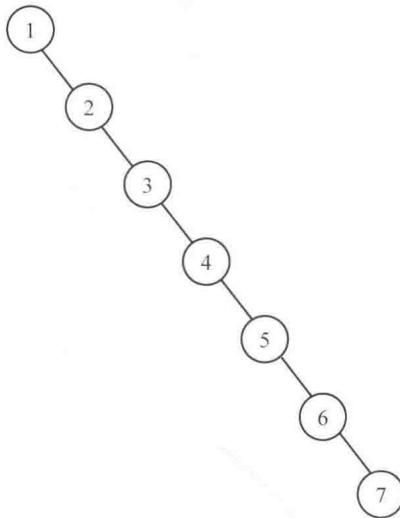


图 8-6 包含前七个整数的倾斜二叉查找树

如果这些键是随机添加的，在直觉上，最终得到的树接近平衡树的可能性应当远高于接近链表的可能性（请参阅附录 A 中 A.8.1 节对随机性的讨论）。因此，在一般情况下，可以预期其查找性能是很高效的。事实上，关于这一效果有一个定理。首先来解释这个定理的结果。这一定理在“所有输入为等概率”的假设前提下，得出了当输入为  $n$  个键时的平均查找时间。所谓的“所有输入为等概率”，是指这  $n$  个键的每种排列顺序作为算

法输入的概率相等（这里所说的算法是指构建树的算法）。例如，如果  $n=3$ ，且  $s_1 < s_2 < s_3$  是三个键，则这些输入全是等概率的：

$$\begin{array}{lll} [s_1, s_2, s_3] & [s_1, s_3, s_2] & [s_2, s_1, s_3] \\ [s_2, s_3, s_1] & [s_3, s_1, s_2] & [s_3, s_2, s_1] \end{array}$$

注意，两个不同输入可能会得到同一棵树。例如， $[s_2, s_3, s_1]$  和  $[s_3, s_1, s_2]$  会得到同一棵树，即以  $s_2$  为根节点， $s_1$  在左子节点， $s_3$  在右子节点。只有这两个输入会生成这棵树。有时，一棵树只能由一种输入生成。例如，由输入  $[s_1, s_2, s_3]$  生成的树是无法由其他输入生成的。我们假设的是输入为等概率的，而不是树为等概率的。因此，所列出的每个输入都具有  $\frac{1}{6}$  概率，而由  $[s_2, s_3, s_1]$  和  $[s_3, s_1, s_2]$  生成的树拥有  $\frac{1}{3}$  概率，由  $[s_1, s_2, s_3]$  生成的树拥有  $\frac{1}{6}$  概率。我们还会假定搜索键  $x$  以相同概率取  $n$  个键之一。下面给出定理。

**定理 8.3** 假设所有输入的概率相等，并且搜索键  $x$  以相同概率取  $n$  个键之一，则对于所有包含  $n$  个不同键的输入，使用二叉树查找方法的平均查找时间近似为：

$$A(n) \approx 1.38 \lg n$$

证明：这里的证明假设查找键  $x$  在树中。在习题中将会证明，即使去除这一限制，只要  $2n+1$  种可能输出结果的概率相同，上述结果仍然成立。（8.1.2 节对二分查找的平均情况分析中讨论了这些输出结果。）

考虑所有包含  $n$  个键的二叉查找树，设根节点处的键为第  $k$  小的键。在所有这些树中，左子树中有  $k-1$  个节点，右子树中有  $n-k$  个节点。对于生成这些树的输入内容，其平均查找时间等于以下三个数量之和：

- 这些树中左子树的平均查找时间乘以  $x$  在左子树出现的概率；
- 这些树中右子树的平均查找时间乘以  $x$  在右子树出现的概率；
- 根节点处的一次比较。

这些树中左子树的平均查找时间为  $A(k-1)$ ，右子树的平均查找时间为  $A(n-k)$ 。因为我们已经假设查找键  $x$  以相同概率取  $n$  个键之一，所以  $x$  处于左右子树的概率分别为：

$$\frac{k-1}{n} \text{ 和 } \frac{n-k}{n}$$

有一些规模为  $n$  的输入，在其生成的二叉查找树中，根节点处的键为第  $k$  小的键，如果用  $A(n|k)$  表示所有这些输入的平均查找时间，根据前述讨论可知：

$$A(n|k) = A(k-1) \frac{k-1}{n} + A(n-k) \frac{n-k}{n} + 1$$

因为所有输入的概率相等，所以每个键作为输入中第一个键的概率相同，从而位于根节点的概率也相同。因此，对于所有这些大小为  $n$  的输入，其平均查找时间就是令  $k$  由 1 变到  $n$  时， $A(n|k)$  的平均值。于是有

$$A(n) = \frac{1}{n} \sum_{k=1}^n \left[ \frac{k-1}{n} A(k-1) + \frac{n-k}{n} A(n-k) + 1 \right]$$

如果设  $C(n)=nA(n)$ ，并将  $C(n)/n$  代入  $A(n)$  的表达式，可得：

$$\frac{C(n)}{n} = \frac{1}{n} \left[ \sum_{k=1}^n \frac{k-1}{n} \frac{C(k-1)}{k-1} + \frac{n-k}{n} \frac{C(n-k)}{n-k} + 1 \right]$$

化简后可得：

$$\begin{aligned} C(n) &= \sum_{k=1}^n \left[ \frac{C(k-1)}{n} + \frac{C(n-k)}{n} + 1 \right] \\ &= \sum_{k=1}^n \frac{1}{n} [C(k-1) + C(n-k)] + n \end{aligned}$$

初始条件为：

$$C(1)=1 \quad A(1)=1$$

这一递推关系几乎与快速排序（算法 2.6）平均情况的递推关系一模一样。模拟快速排序的平均情况分析，可得：

$$C(n) \approx 1.38(n+1)\lg n$$

也就是说：

$$A(n) \approx 1.38\lg n$$

一定要当心，不要错误解读定理 8.3 的结果。这个定理并不是说，一个包含  $n$  个键的特定输入的平均查找时间大约为  $1.38\lg n$ 。一个特定输入生成的树可能会降级为图 8-6 中的树，也就是说，平均查找时间属于  $\Theta(n)$ 。定理 8.3 是针对所有包含  $n$  个键的输入给出的平均查找时间。因此，对于任意包含  $n$  个键的给定输入，平均查找时间可能属于  $\Theta(\lg n)$ ，也可能属于  $\Theta(n)$ 。在这里并不能保证查找非常高效。

### 8.3.2 B 树

在许多应用中，其性能可能会因为线性查找时间而大幅下降。例如，大型数据库中各条记录的键通常无法一次性全部放到计算机的高速存储器（RAM，Random Access Memory）中。因此，需要多次磁盘访问才能完成查找。（这种查找称为外部查找，而当所有键都同时放在内存中时，称为内部查找。）因为磁盘访问涉及读写头的机械移动，而 RAM 访问只涉及电子数据转移，所以磁盘访问的速度要比 RAM 访问慢上若干个量级。而一个线性时间的外部查找可能是无法接受的，我们可不希望出现这种可能性。

这一问题的一种解决方案是编写一个平衡程序，它以现有的二叉查找树为输入，生成一个包含相同键的平衡二叉查找树。然后定期运行这一程序。算法 3.9 就是这样一种程序的算法。此算法绝不仅仅是一种简单的平衡算法，因为它能够考虑每个键作为查找键的概率。

在一种非常动态化的环境中，如果这种树根本就不会变成非平衡的，那可能会更好一些。1962 年，两位俄国数学家 G. M. Adel'son-Vel'skii 和 E. M. Landis 设计了在添加和删除节点的同时保持平衡二叉树的算法（因此，平衡二叉树也经常被称为 AVL 树。）这些算法可以在 Kruse（1994 年）的文献中找到。这些算法的添加和删除时间可以保证属于  $\Theta(\lg n)$ ，查找时间也是如此。

1972 年，R. Bayer 和 E. M. McCreight 对二叉查找树进行了改进，设计了一种 B 树。在 B 树中添加或删除键时，可以保证所有叶节点都在同一级别，这甚至比保持平衡还要好一些。B 树的实际操作算法可以在 Kruse（1994 年）的文献中找到。这里只是演示如何在添加键时使所有叶节点都保持在同一级别。

B 树实际上代表着一类树，其中最简单的就是 3-2 树。我们将演示向这种树中添加节点的过程。3-2 树是具有以下性质的树：

- 每个节点都包含一个或两个键；
- 如果一个非叶节点包含一个键，它就有两个子节点，如果它包含两个键，就有三个子节点；
- 一个给定节点左子树中的键要小于或等于该节点中存储的键；
- 一个给定节点右子树中的键要大于或等于该节点中存储的键；
- 如果一个节点包含两个键，则该节点中间子树中的键大于或等于左键，小于或等于右键；
- 所有叶节点都在同一级别；

图 8-7a 给出了一棵 3-2 树，图中其余部分表明如何向这棵树中添加一个新键。注意，这棵树会保持平衡，因为它增加深度的位置是在根节点处，而不是叶节点。同理，当需要删除一个节点时，这棵树也是在根节点处缩减深度的。这样，所有叶节点都保持在同一级别，可以保证查找、添加和删除时间都属于  $\Theta(\lg n)$ 。显然，对这棵树进行先序遍历可以按有序序列提取键。由于这些原因，大多数现代数据库管理系统中都使用 B 树。

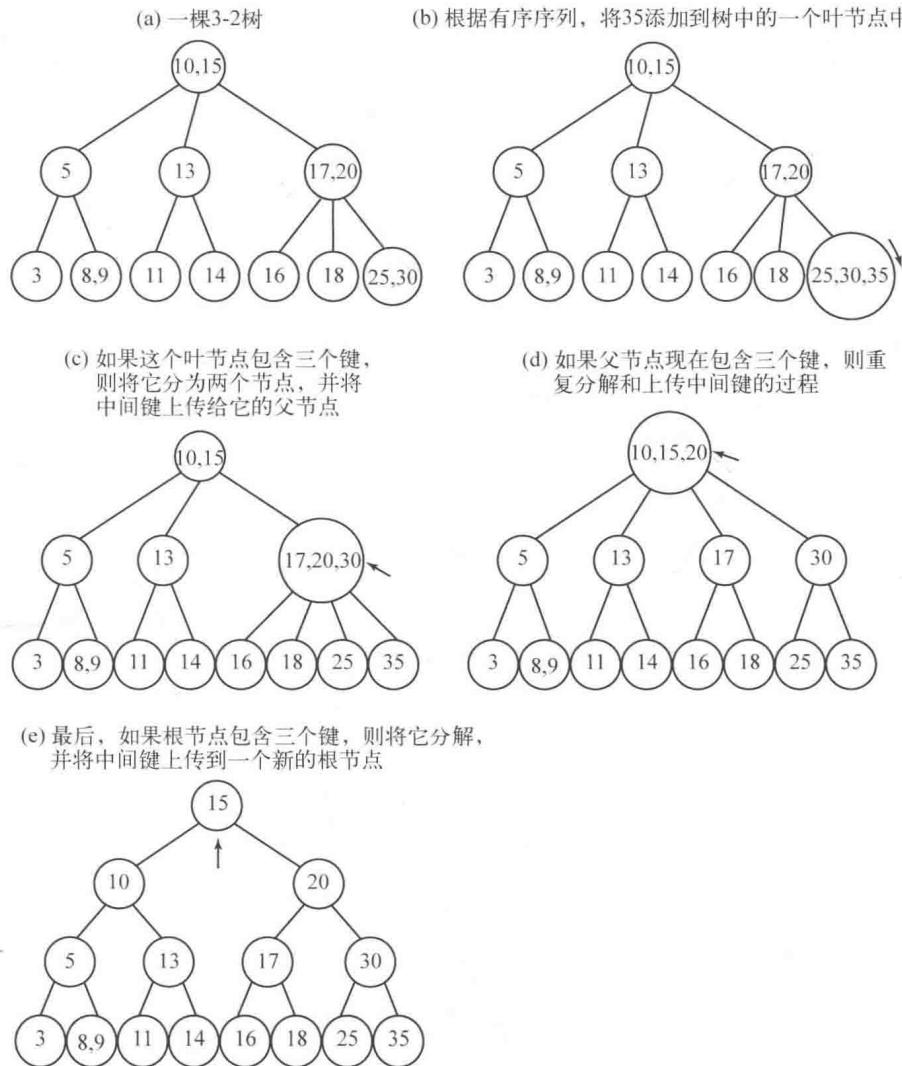


图 8-7 向 3-2 树中添加一个新键的方式

## 8.4 散列

假设键是从 1 到 100 的整数, 大约有 100 条记录。存储这些记录的一种高效方法是创建一个包含 100 项的数组  $S$ , 并将键等于  $i$  的记录存储在  $S[i]$  中。提取任务可即时完成, 不需要进行键的比较。如果大约有 100 条记录, 而且键是长为 9 位的社会保障号, 也可以使用同一策略。但在这种情况下, 此策略在内存方面的效率就非常低下了, 因为仅仅为了存储 100 条记录而使用了一个存储 10 亿个项目的数组。或者, 我们可以创建一个仅有 100 条记录的数组, 其索引范围为 0 到 99, 希望能将每个键“散列”到介于 0 到 99 之间的一个值。散列函数 (hash function) 就是一种将键转换为索引的函数, 将散列函数应用于一个特定键的过程, 称为“散列该键”。对于社会保障号的例子, 一种可能使用的散列函数是

$$h(\text{key}) = \text{key} \% 100$$

(%返回 key 除以 100 所得的余数。) 这个函数只是返回该键的后两位。如果一个特定键被散列为  $i$ , 就将这个键及其记录存储在  $S[i]$  处。这一策略没有将键存储为有序序列, 也就是说, 只有在不需要高效提取有序序列中的记录时, 这一策略才是适用的。如果需要高效提取, 那就应当使用之前讨论过的某种方法。

如果任何两个键都不会散列到同一索引，这种方法就能很好地工作。但当键数很大时，很少会是如此。例如，如果有 100 个键，每个键以同等概率被散列为 100 个索引之一，那任何两个键都不被散列为同一索引的概率为：

$$\frac{100!}{100^{100}} \approx 9.3 \times 10^{-43}$$

我们几乎可以肯定，至少会有两个键被散列为同一索引。这种情况称为冲突 (collision) 或散列冲突 (hash clash)。有多种方法可以解决这一冲突。最好的方法之一就是使用开放散列 (open hashing，也称为开放定址，open addressing)。利用开放散列，我们为每个可能的散列值创建一个桶，并将所有散列到一个值的键放到与该值相关联的桶中。开放散列通常用链表实现。例如，如果散列到一个数字的后两位，可以创建一个指针数组 Bucket，其范围范围为 0 到 99。所有被散列到  $i$  的键都被放在一个以  $\text{Bucket}[i]$  开头的链表中。如图 8-8 所示。

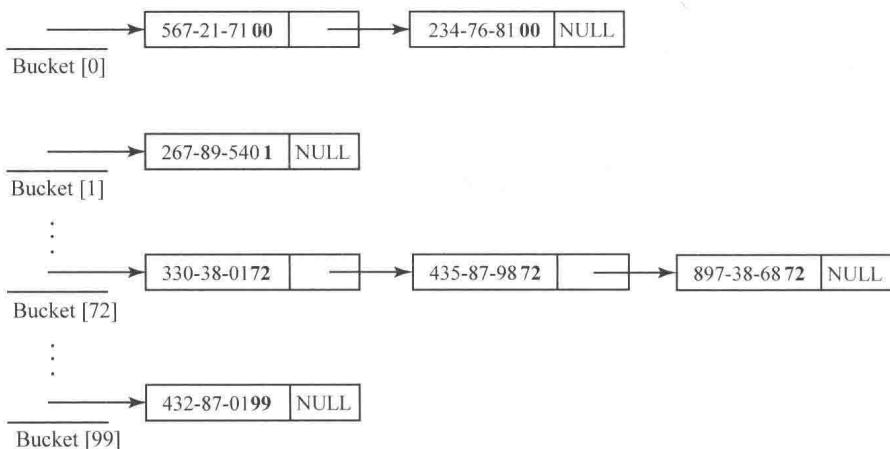


图 8-8 开放散列的示意图。后两位数字相同的键被放在同一个桶内

桶的数量不一定与键的数目相同。例如，如果散列到一个数字的后两位，那桶的数目必然为 100。但我们可以存储 100 个、200 个、1000 个或任意数目的键。当然，存储的键越多，越可能出现冲突。如果键的数目大于桶的数目，那肯定会出现冲突。因为一个桶中仅存储一个指针，所以分配一个桶并不会浪费太多空间。因此，所分配的桶数至少与键数相同，这样做通常是合理的。

在查找一个键时，需要对包含该键的桶（链表）进行顺序查找。如果所有键被散列到同一个桶中，那查找过程就退化为顺序查找。发生这种可能性的概率如何呢？如果有 100 个键和 100 个桶，而且一个键被散列为每个桶的概率相同，则所有键最终都出现在同一个桶中的概率为：

$$100 \times \left( \frac{1}{100} \right)^{100} = 10^{-198}$$

因此，几乎不可能出现所有键都在同一个桶中的情况。那 90 个、80 个、70 个或任意其他较大量键出现在同一桶中的概率呢？我们真正关心的是，散列后的平均查找性能优于二分查找的概率有多大。后面将会证明，如果文件相当大，那散列后的平均查找性能几乎一定会优于二分查找。但首先来看散列能好到什么程度。

从直觉上来看，应当很清楚，当这些键均匀分布在桶中的时候，可以达到最佳效果。也就是说，如果有  $n$  个键和  $m$  个桶，每个桶中应当包含  $n/m$  个键。实际上，只有当  $n$  是  $m$  的整数倍时，每个桶中才可能恰好包含  $n/m$  个键。如果不是它的倍数，那只能得到近似的均匀分布。下面的定理说明了在这些键为均匀分布时会发生什么情况。为简单起见，这些定理的表述针对的是严格均匀分布（也就是说， $n$  为  $m$  的整数倍）。

**定理 8.4** 如果  $n$  个键均匀分布在  $m$  个桶内，则一次失败查找中的比较次数为  $n/m$ 。

**证明：**因为这些键是均匀分布的，所以每个桶中的键数为  $n/m$ ，这就意味着，每次失败查找需要  $n/m$  次比较。

**定理 8.5** 如果  $n$  个键均匀分布在  $m$  个桶内，每个键作为搜索键的概率相同，则成功查找中的平均比较次数为：

$$\frac{n}{2m} + \frac{1}{2}$$

**证明：**每个桶中的平均查找时间等于对  $n/m$  个键进行顺序查找时的平均查找时间。1.3 节对算法 1.1 的平均情况分析表明，这一平均值等于该定理中给出的表达式。

下面的例子应用了定理 8.5。

**例 8.1** 如果键为均匀分布，而且  $n=2m$ ，则每个失败查找仅需要  $2m/m=2$  次比较，一次成功查找的平均比较次数为：

$$\frac{2m}{2m} + \frac{1}{2} = \frac{3}{2}$$

当这些键是均匀分布时，查找时间非常短。尽管散列有可能给出这样出色的结果，但有人可能还是会说，仍然应当使用二分查找法，以保证查找过程不会退化到接近顺序查找。下面的定理表明，如果文件相当大，散列过程与二分查找一样差的概率是非常小的。此定理假设一个键被散列到每个桶中的概率相同。当社会保障号被散列到它的后两位时，这一条件应当是满足的。但是，并不是所有散列函数都满足这一条件。例如，如果将姓名散列到它们的最后字母，那散列到“th”的概率要远高于散列为“qz”的概率，因为以“th”结尾的名字要多很多。一些数据结构教科书中，比如 Kruse (1994 年) 的文献，讨论了选择优秀散列函数的方法。我们这里的目的是分析一下，散列可以多么出色地解决查找问题。

**定理 8.6** 如果有  $n$  个键和  $m$  个桶，假设一个键被散列到任意桶中的概率相等，则至少一个桶中包含至少  $k$  个键的概率小于或等于

$$\binom{n}{k} \left(\frac{1}{m}\right)^{k-1}$$

**证明：**对于一个给定的桶， $k$  个键的任一特定组合最终在该桶内出现的概率为  $(1/m)^k$ ，这就是说，这个桶中如果至少包含了这种组合中的各个键，其概率为  $(1/m)^k$ 。一般来说，对于两个事件  $S$  和  $T$ ，

$$p(S \text{ 或 } T) \leq p(S) + p(T) \quad (8.1)$$

因此，一个给定桶中至少包含  $k$  个键的概率，小于或等于若干个概率之和，这若干个概率就是这个桶中至少包含每一  $k$  键组合的概率。因为由  $n$  个键可以获得  $k$  个键的  $\binom{n}{k}$  种不同组合形式（见附录 A 的 A.7 节），所以任意给定桶中至少包含  $k$  个键的概率小于或等于

$$\binom{n}{k} \left(\frac{1}{m}\right)^k$$

由式 8.1 及共有  $m$  个桶的事实可证得此定理。

回想一下，二分查找的平均查找时间为  $\lg n$ 。表 8-1 给出了当  $n$  取不同值时，至少一个桶中包含至少  $\lg n$  个键和  $2\lg n$  个键的概率上限。在此表中假设  $n=m$ 。即使当  $n$  仅为 128 时，查找时间超过  $2\lg n$  的概率也小于 10 亿分之一。当  $n=1024$  时，查找时间超过  $2\lg n$  的机会小于 1 亿亿分之 3。当  $n=65536$  时，查找时间超过  $\lg n$  的机会大约为 10 亿分之 3。而乘坐喷气式飞机在一次旅行中丧生的概率大约为 100 万分之 6，平均驾驶汽车一年，在车祸中丧生的概率大约为 100 万分之 270。可许多人都并没有刻意采取什么措施来避免这些行为。我们要说的是，为了做出合理的决策，人们经常会忽略一些特别小的概率，就像它们根本不会发生一样。因为当  $n$  值很大时，我们几乎可以确信使用散列的性能要优于二分查找法，所以使用散列就是合理的。20 世纪 70 年代

有一件很有趣的事情，一家计算机制造商定期游说数据处理管理员购买非常昂贵的新硬件，它的策略就是向这些管理员描述在特定应用情景下可能发生的灾难。但他们的论述中有一个缺点，也是数据处理管理员经常忽略的一点，那就是出现这种应用情景的概率可以忽略。对风险的厌恶程度与人们的性格偏爱有关。因此，一个极端厌恶风险的人可能选择不去冒十亿分之一甚至一亿亿分之三的风险。但是，一个人在做出这种选择之前，应当认真考虑一下，这样一种决策是否真正体现了他对待风险的态度。Clemen (1991 年) 的文献中讨论了进行这种分析的一些方法。

表 8-1 至少一个桶中包含至少  $k$  个键的概率上限\*

$n$	当 $k=\lg n$ 时的上限	当 $k=2\lg n$ 时的上限
128	0.021	$7.02 \times 10^{-10}$
1024	0.000 27	$3.49 \times 10^{-16}$
8192	0.000 001 3	$1.95 \times 10^{-23}$
65 536	$3.1 \times 10^{-9}$	$2.47 \times 10^{-31}$

\*假定键的个数  $n$  等于桶的个数。

## 8.5 选择问题：对手论证

到现在，我们已经讨论了一个  $n$  键列表中查找一个键  $x$  的过程。接下来讨论一种名为选择问题 (selection problem) 的不同的查找问题。这个问题是找出一个  $n$  键列表中的第  $k$  大 (或第  $k$  小) 键。假定这些键存储在一个无序数组中 (对于有序数组来说，这个问题就不值一提了)。首先讨论  $k=1$  时的问题，也就是说，我们要找的是最大键 (或最小键)。接下来将表明，在同时找出最小键和最大键时，所需要的比较次数可以小于分别找出它们时的比较次数。然后我们将讨论当  $k=2$  时的问题，也就是说找出第二大 (或第二小) 的键。最后讨论一般情景。

### 8.5.1 找出最大键

下面是一种找出最大键的简单算法。

#### 算法 8.2 找出最大键

问题：在大小为  $n$  的数组  $S$  中找出最大键。

输入：正整数  $n$ ；键的数组  $S$ ，其索引范围为 1 至  $n$ 。

输出：变量  $\text{large}$ ，它的值是  $S$  中的最大键。

```
void find_largest (int n,
                   const keytype S[],
                   keytype& large)
{
    index i;

    large = S[1];
    for (i=2; i<=n; i++)
        if (S[i]>large)
            large=S[i];
}
```

显然，该算法完成的键比较次数为：

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

从直觉上来看，似乎不可能改进这一性能。下面的定理表明确实如此。我们可以把确定最大键的算法看作键之间的一场锦标赛。每次比较都是锦标赛中的一场比赛，较大的键是比较的胜者，较小的键是败者。锦标赛的胜者是最大键。本节将一直使用这一术语。

**定理 8.7** 任何一种确定性算法，如果仅通过键的比较，在每个可能输入中找出  $n$  个键中的最大者，至少要进行  $n-1$  次键的比较。

证明：此证明采用反证法。也就是说，我们将证明，如果对于大小为  $n$  的某个输入，该算法执行的比较次数少于  $n-1$ ，那该算法必然会为其他某一输入给出错误答案。如果对于某一输入，该算法最多进行  $n-2$  次比较就找出最大键，那输入中至少有两个键从来没有进行比较。这两个键中至少有一个不能被报告为最大键。我们可以创建一种新的输入：（在必要时，）将这个键用一个大于原输入中所有键的键代替。因为所有比较的结果都与原输入的比较相同，所以新键也不会被报告为最大键，这就是说，这个算法对于新的输入会给出错误答案。这一矛盾证明，对于每个大小为  $n$  的输入，此算法必须至少进行  $n-1$  次比较。

在解读定理 8.7 时，一定要非常小心，不要误读。它并不是说，每一种仅通过键的比较进行查找的算法都至少需要  $n-1$  次比较才能找出最大键。例如，如果数组是有序的，那根本就不需要任何比较，只需返回数组中的最后一项就能给出最大键。但是，只有当最大键为最后一项时，这种返回数组最后一项的算法才能找出最大键。它无法在每个可能输入中都找出最大键。定理 8.7 讨论的是在所有可能输入中都能找出最大键的算法。

当然，我们可以使用算法 8.2 的一个类似版本在  $n-1$  次比较中找出最小键，使用定理 8.7 的一个类似版本来证明： $n-1$  是找出最小键的下限。

### 8.5.2 同时找出最大键和最小键

同时找出最大、最小键的一种简单方法是将算法 8.2 修改如下。

#### 算法 8.3 找出最小键和最大键

问题：在大小为  $n$  的数组  $S$  中找出最小键和最大键。

输入：正整数  $n$ ；键的数组  $S$ ，其索引范围为 1 至  $n$ 。

输出：变量  $small$  和  $large$ ，它们的数值是  $S$  中的最小键和最大键。

```
void find_both (int n,
                const keytype S[],
                keytype& small,
                keytype& large)
{
    index i;

    small = S[1];
    large = S[1];
    for (i = 2; i<=n; i++)
        if (S[i] < small)
            small = S[i];
        else if (S[i] > large)
            large = S[i];
}
```

使用算法 8.3 要好于分别查找最小键和最大键，因为对于某一输入来说，不会针对每个  $i$  值都将  $S[i]$  与  $large$  进行比较。因此，我们已经针对所有情景的性能进行了改进。但只要  $S[1]$  是最小键，就会对所有  $i$  进行比较。因此，最差情况下的键比较次数为：

$$W(n)=2(n-1)$$

这恰好就是在分别查找最小键和最大键时完成的比较次数。看起来似乎不能再提升这一性能，但实际上是可以的。技巧在于对这些键进行配对，找出每一对中的较小键。利用大约  $n/2$  次比较可以做到这一点。通过大约  $n/2$  次比较可以找出所有较小键的最小键，通过大约  $n/2$  次比较找出所有较大的最大键。这样，只需要大约  $3n/2$  次比较就能同时找出最小键和最大键。这一方法的算法如下。此算法假设  $n$  为偶数。

#### 算法 8.4 通过键的配对找出最小键和最大键

问题：在大小为  $n$  的数组  $S$  中找出最小键和最大键。

输入：正的偶整数  $n$ ；键的数组  $S$ ，其索引范围为 1 至  $n$ 。

输出：变量  $\text{small}$  和  $\text{large}$ ，它们的值就是  $S$  中的最小键和最大键。

```

void find_both2 (int n,                                // 假设 n 为偶数。
                 const keytype S[], 
                 keytype& small,
                 keytype& large)
{
    index i;

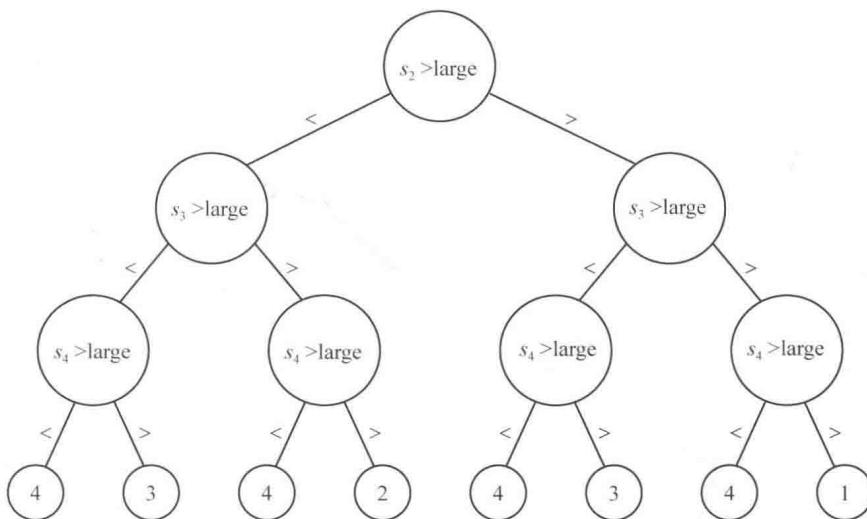
    if (S[1] < S[2]){
        small = S[1];
        large = S[2];
    }
    else {
        small = S[2];
        large = S[1];
    }
    for (i = 3; i <= n - 1; i = i + 2){           // 将 i 递增 2。
        if (S[i] < S[i + 1]) {
            if (S[i] < small)
                small = S[i];
            if (S[i + 1] > large)
                large = S[i + 1];
        }
        else {
            if (S[i + 1] < small)
                small = S[i + 1];
            if (S[i] > large)
                large = S[i];
        }
    }
}

```

以下工作留作练习：修改该算法，使其在  $n$  为奇数时也能正常工作，并证明其执行的键比较次数为

$$T(n) = \begin{cases} \frac{3n}{2} - 2 & (\text{若 } n \text{ 为偶数}) \\ \frac{3n}{2} - \frac{3}{2} & (\text{若 } n \text{ 为奇数}) \end{cases}$$

能不能再提升这一性能呢？我们将证明，答案是“不能”。我们不会使用决策树来证明，因为决策树对于选择问题的效果不是很好。原因如下。我们知道选择问题的决策树必然包含至少  $n$  个叶节点，这是因为有  $n$  种可能输出。根据引理 7.3，如果二叉树有  $n$  个叶节点，它的深度将大于或等于  $\lceil \lg n \rceil$ 。因此，叶节点数目的下限可以给出最差情况下的比较次数下限为  $\lceil \lg n \rceil$ 。这并不是一个非常好的下限，因为我们已经知道，仅仅因为要找出最大键，就至少需要  $n-1$  次比较（定理 8.7）。引理 8.1 不再有用，因为我们只能轻松地证明，决策树中至少有  $n-1$  个比较节点。决策树对于选择问题的效果不是特别好，因为一个结果可能出现在多个叶节点中。图 8-9 给出了当  $n=4$  时算法 8.2（找出最大键）的决策树。共有四个叶节点报告 4，两个叶节点报告 3。该算法完成的比较次数是 3，而不是  $\lceil \lg 4 \rceil = 2$ 。可以看出， $\lceil \lg n \rceil$  是一个很弱的下限。

图 8-9 与  $n=4$  时的算法 8.2 相对应的决策树

我们使用另外一种名为对手论证 (adversary argument) 的方法来确定下限。所谓对手，就是一位反对者。假定你是在本地一家单身酒吧里，一位好奇的陌生人向你问了那个古老的问题：“你是什么座的？”也就是说，他想知道你是哪个星座的。一共有 12 个星座，分别对应着大约 30 天的时间。如果你是 8 月 25 日出生的，那你的星座就是处女座。为了让这样一次没有任何新意的相遇变得更让人兴奋一些，你决定增加一点趣味，自己来扮演一位对手。你告诉这位陌生人，可以通过询问“是/否”问题来猜测你的星座。作为一位对手，你不想暴露自己的星座——你只是想让陌生人提出尽可能多的问题。因此，你的回答总是尽量避免缩小搜索范围。例如，假设陌生人问道：“你是出生在夏天吗？”如果你回答“否”，那就会将搜索范围缩小为 9 个月，而回答“是”会将其缩小到 3 个月，所以你回答“否”。如果陌生人接着问：“你出生的月份有 31 天吗？”你回答“是”，因为在剩下的 9 个可能月份中，一半以上有 31 天。对你的回答只有一条要求，那就是它们与前面已经给出的回答一致。例如，如果陌生人忘了你已经说过自己不是出生在夏天，后来又问你是否出生在七月份，你不能回答“是”，因为这样就与之前的回答不一致。答案不一致时，就无法找出与之匹配的星座（生日）来。因为你回答每个问题时，都让剩余可能答案的数目最多，而且因为你的答案是一致的，所以就会迫使陌生人提出尽可能多的问题，以得出一个合乎逻辑的结论。

假定某位对手的目标是让一个算法尽可能努力地工作（就像你让那位陌生人提出尽可能多的问题一样）。每当算法必须做出决策时（比如，在对键进行一次比较后），这些对手尝试选择的决策结果会尽可能让算法走得长一些。唯一的约束条件就是，这位对手选择的结果必须与之前给出的结果保持一致。只要这些结果保持一致，就一定存在一个导致这一系列结果的输入。如果对手强制将基本指令执行  $f(n)$  次，那  $f(n)$  就是该算法最差时间复杂度的下限。

要同时找出最大键和最小键，在最差情况下需要多少次比较？我们使用对手论证来获得这一数值的下限。这一论证过程最早出现在 Pohl (1972 年) 的文献中。为了确定此限值，假定这些键互不相同。之所以能这样做，是因为一个输入子集（键值互不相同的输入）的最差时间复杂度的下限就是考虑所有子集时最差时间复杂度的下限。在给出这一定理之前，先来介绍对手的策略。假定有一种算法，它仅通过键的比较来解决找出最小键和最大键的问题。如果所有键互不相同，在该算法执行期间的任意给定时刻，一个给定键必为以下状态之一。

状态	状态描述
X	该键没有参与比较。
L	该键在至少一次比较中失败，而且从来没有胜过。
W	该键在至少一次比较中获胜，而且从来没有输过。
WL	该键在至少一次比较中获胜，而且在至少一次比较中失败过。

我们可以把这些状态看作包含了若干个信息单位。如果键的状态为 X，则有零个信息单位。如果键的状态为 L 或 WL，则有一个信息单位，因为我们知道这个键在一次比较中胜过或败过。如果一个键的状态为 WL，则有两个信息单位，因为我们知道它既在比较中胜过，也在比较中失败过。该算法要确认一个键 small 是最小键，另一个键 large 是最大键，必须知道除 small 之外的每个键都在一次比较获胜过，而除了 large 之外的每个键都在一次比较中失败过。也就是说，此算法必须掌握  $(n-1)+(n-1)=2n-2$  个信息单位。

因为对手的目的就是让算法尽可能努力地工作，所以他希望在每次比较中提供尽可能少的信息。例如，如果算法首先比较  $s_2$  和  $s_1$ ，那对手回答什么都是无关紧要的，因为无论回答什么，都会提供两个信息单位。假定对手回答  $s_2$  较大， $s_2$  的状态于是由 X 变为 W， $s_1$  的状态由 X 变为 L。假定算法接下来比较  $s_3$  和  $s_1$ 。如果对手回答  $s_1$  较大， $s_1$  的状态由 L 变为 WL， $s_3$  的状态由 X 变为 L。这意味着将揭示两个信息单位。因为回答  $s_3$  较大时只会揭示一个信息单位，所以我们让对手给出这一答案。表 8-2 给出了一种对手策略，它总是揭示最少量的信息。如果选择哪个键都无关紧要，那直接选择比较中的第一个键。这一策略就是证明该定理所需要的全部信息。但首先给出一个例子，说明我们的对手会如何实际应用这一策略。

表 8-2 阻止一个算法找出最小键和最大键的对手策略\*

比较前		对手声明的较大键	比较后		算法了解到信息单位
$s_i$	$s_j$		$s_i$	$s_j$	
X	X	$s_i$	W	L	2
X	L	$s_i$	W	L	1
X	W	$s_j$	L	W	1
X	WL	$s_i$	W	WL	1
L	L	$s_i$	WL	L	1
L	W	$s_j$	L	W	0
L	WL	$s_j$	L	WL	0
W	W	$s_i$	W	WL	1
W	WL	$s_i$	W	WL	0
WL	WL	与之前的答案一致	WL	WL	0

\*比较  $s_i$  和  $s_j$ 。

例 8.2 表 8-3 给出了我们的对手针对一个大小为 5 的输入，阻止算法 8.3 的策略。我们为这些键指定了一些与回答一致的值。并不是一定要指定这些值，但对手必须跟踪这些回答为这些键设定的顺序，以便在两个键的状态均为 WL 时，可以给出一致回答。为此，最轻松的方法就是为它们指定值。此外，数值的指定还表明，一组相互一致的答案必然有一个与其相关联的输入。除了由对手策略确定的顺序之外，这些答案是任意的。例如，当  $s_3$  被声明为大于  $s_1$  时，我们为  $s_3$  指定数值 15。也可以为其指定任何大于 10 的值。

表 8-3 对于大小为 5 的输入，我们的对手为尝试阻止算法 8.3 而给出的回答

比 较	对手声明的较大键	状态/指定值					算法了解到的信息单位
		$s_1$	$s_2$	$s_3$	$s_4$	$s_5$	
$s_2 < s_1$	$s_2$	L/10	W/20	X	X	X	2
$s_2 > s_1$	$s_2$	L/10	W/20	X	X	X	0
$s_3 < s_1$	$s_3$	L/10	W/20	W/15	X	X	1
$s_3 > s_2$	$s_4$	L/10	WL/20	W/30	X	X	1
$s_4 < s_1$	$s_4$	L/10	WL/20	W/30	W/15	X	1
$s_4 > s_3$	$s_4$	L/10	WL/20	WL/30	W/40	X	1
$s_5 < s_1$	$s_6$	L/10	WL/20	WL/30	W/40	W/15	1
$s_5 > s_4$	$s_5$	L/10	WL/20	WL/30	WL/40	W/50	1

注意，在将  $s_3$  与  $s_2$  对比之后，我们将  $s_3$  的值由 15 改为 30。请记住，在向对手展示此算法时，他手里

并没有实际输入。他所给出的答案（也就是输入值）是在获知算法当前的决策时，动态构造出来的。将  $s_3$  的值改为 15 是有必要的，因为  $s_2$  大于 15，对手给出的答案是  $s_3$  大于  $s_2$ 。

在算法完成后， $s_1$  已经输给了所有其他键， $s_5$  已经赢得所有其他键。因此， $s_1$  最小， $s_5$  最大。在对手构造的输入中， $s_1$  为最小键，因为这样一个输入会使算法 8.3 工作得最为吃力。注意，表 8-3 中完成了八次比较，当输入大小为 5 时，算法 8.3 在最差情况下执行的比较次数为：

$$W(5)=2(5-1)=8$$

这就是说，当输入大小为 5 时，我们的对手成功地使算法 8.3 的工作难度达到极致。

当我们的对手面对另外一种查找最大键和最小键的算法时，他提供的回答会让该算法尽可能努力地工作。希望读者能够判断他在面对算法 8.4（通过键的配对找出最小键和最大键）和某一输入大小时会给出什么样的答案。

对于解决某一问题的算法，我们在设计对手策略时的目的是让算法尽可能努力地工作。拙劣的对手可能无法真正实现这一目标。但无论能否实现该目标，都可以利用这一策略来获取一个下限，表明这些算法必须多么努力地工作。接下来就用刚刚描述的对手策略来确定这一下限。

**定理 8.8** 如果一种确定性算法，仅通过键的比较就能在每种可能输入的  $n$  个键中找出最小值和最大值，它在最差情况下至少要完成的键比较次数如下：

$$\frac{3n}{2}-2 \quad (\text{若 } n \text{ 为偶数})$$

$$\frac{3n}{2}-\frac{3}{2} \quad (\text{若 } n \text{ 为奇数})$$

证明：我们将证明当这些键互不相同时，该算法至少要完成这么多次的键比较，以此来证明这是最差情况下的下限。如前文所述，要找出最小键和最大键，该算法必须知道  $2n-2$  个信息单位。假定我们向对手展示了这一算法。表 8-2 表明，只有当两个键都未参与过之前的比较时，我们的对手才会在一次比较中提供两个信息单位。如果  $n$  为偶数，最多在  $\frac{n}{2}$  次比较中属于这种情况，这就是说，采用这一方式，该算法最多可以获得  $2(n/2)=n$  个信息单位。因为在其他比较中，我们的对手最多提供一个信息单位，所以该算法至少要执行  $2n-2-n=n-2$  次额外比较，才能获得所需要的全部信息。因此，我们的对手强制该算法至少执行  $\frac{n}{2}+n-2=\frac{3n}{2}-2$  次键的比较。

当  $n$  为奇数时的情景分析，留作练习。

因为算法 8.4 执行的比较次数在定理 8.8 给出的界限之内，所以该算法的最差性能是最优的。我们选择了一位最有价值的对手，因为我们找到了一种算法，其性能可以达到它能提供的界限。因此可以知道，没有其他对手可以提供更大的下限了。

例 8.2 说明了算法 8.3 为什么是次优的。在这个例子中，算法 8.3 用了 8 次比较来掌握八个信息单位，而一个最优算法只需要 6 次比较。表 8-3 表明，第二次比较是没有用的，因为这次比较中没有掌握到信息。

在使用对手论证时，对手有时被称为 oracle。在古希腊和古罗马，oracle 是指拥有渊博的知识、可以回答人们问题的个体或团体。

### 8.5.3 找出第二大的键

为找出第二大的键，可以使用算法 8.2（求最大键），用  $n-1$  次比较找出最大键，然后删除该键，再次使用算法 8.2，用  $n-2$  次比较找出剩余键中的最大键。因此，可以用  $2n-3$  次比较找出第二大的键。我们应当可以改进这一结果，因为在查找最大键时完成的比较中，有许多可以用来将一些键排除在外，它们不可能作为第二大键的竞争者。也就是说，对于任何一个键，如果它输给了最大键之外的键，那就不可能是第二大的。下面介绍的锦标赛方法利用了这一事实。

锦标赛方法的得名是因为它模仿了在淘汰锦标赛中使用的方法。例如，为确定美国最佳大学篮球队，64

支队伍在美国大学生篮球联赛中竞赛。这些队伍捉对比赛，第一轮共有 32 场比赛。32 支获胜队伍在第二轮中捉对比赛，共举行 16 场比赛，一直持续这一过程，直到只剩下最后一轮中的两支队队伍。这一场比赛的胜者就是冠军。共需要  $\lg 64=6$  轮比赛才能决出冠军。

为简单起见，假定各个数字互不相同，而且  $n$  是 2 的幂。和美国大学生篮球联赛中的做法一样，我们将这些键配对，并在各轮中比较各对键，直到只剩下一轮。如果有 8 个键，则第一轮进行 4 次比较，第二轮 2 次，最后一轮 1 次。最后一轮的胜者就是最大键。图 8-10 演示了这一方法。只有当  $n$  是 2 的幂时才能直接应用锦标赛方法。如果  $n$  不是 2 的幂，那就在数组的最后添加足够多的项目，使数组大小变为 2 的幂。例如，如果该数组中有 53 个整数，那就在数组的最后增加 11 个元素，其取值均为  $-\infty$ ，使数组包含 64 个元素。在下面的讨论中，直接假定  $n$  是 2 的幂。

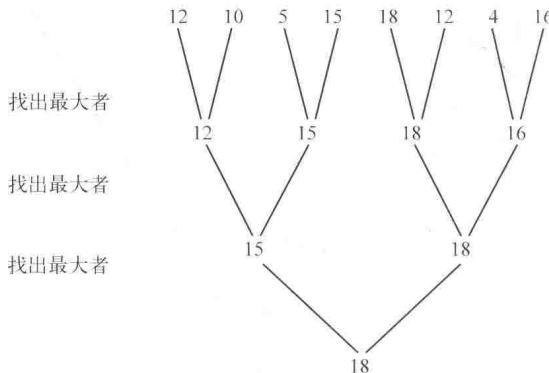


图 8-10 锦标赛方法

尽管最后一轮的胜者为最大键，但该轮的负者不一定是第二大的。在图 8-10 中，第二大的键（16）在第二轮败给了最大键（18）。这是许多实际锦标赛中的难题，因为两个最佳球队不一定总是在冠亚军决赛中相遇。所有熟悉美国足球超级碗的人都了解这一点。为找出第二大的键，可以跟踪所有败给最大键的键，然后用算法 8.2 找出这些键中的最大者。但在未能事先知道哪个键是最大键时，如何跟踪这些键呢？为此，可以分别为每个键维护一个列表。在某个键输掉比赛时，就将它加到获胜键的链表中。如何为这一方法编写一种算法，留作练习。如果  $n$  是 2 的幂，则第一轮有  $n/2$  次比较，第二轮有  $n/2^2$  次……最后一轮有  $n/2^{\lg n}=1$  次。所有各轮的总比较次数为：

$$T(n) = n \sum_{i=1}^{\lg n} \left(\frac{1}{2}\right)^i = n \left[ \frac{(1/2)^{\lg(n)+1} - 1}{1/2 - 1} \right] = n - 1$$

倒数第二个等式是应用附录 A 中例 A.4 的结果得到的。这就是完成整个锦标赛所需要的比较次数。（注意，我们通过最优次数的比较找出了最大键。）最大键将参与  $\lg n$  场比赛，也就是说，在它的链表中会有  $\lg n$  个键。如果利用算法 8.2，将需要  $\lg n - 1$  次比较在这个链表中找出最大键。这个键就是第二大的键。因此，找出第二大键所需要的总比较次数为：

$$T(n) = n - 1 + \lg n - 1 = n + \lg n - 2$$

对于一般的  $n$  值，

$$T(n) = n + \lceil \lg n \rceil - 2$$

其证明留作练习。

相对于两次运用算法 8.2 来查找第二大的键，这一性能的提升非常显著。回忆一下，两次运用算法 8.2 时需要  $2n-3$  次比较。能不能获得一种比较次数更少的算法呢？我们用对方论证来证明这是不可能的。

**定理 8.9** 如果一种确定性算法，仅通过键的比较就能在每种可能输入的  $n$  个键中找出最小值和最大值，它在最差情况下至少要进行  $n + \lceil \lg n \rceil - 2$  次键的比较。

**证明** 一种算法要确定一个键是第二大的，就必须判定在除最大键之外的  $n-1$  个键中，这个键是最大的。

设  $m$  是最大键获胜的比较次数。在确定第二大键是剩余  $n-1$  个键中的最大键时，这些比较都没有用。根据定理 8.7，这一确定至少需要  $n-2$  次比较。因此，总比较次数至少为  $m+n-2$ 。这就是说，要证明此定理，一个对手只需强制该算法让最大键至少参与  $\lceil \lg n \rceil$  次比较。我们对手的策略是为每个键关联树中的一个节点。开始时我们创建了  $n$  个单节点树，每个键各有一个。我们的对手利用这些树给出  $s_i$  与  $s_j$  的比较结果，方法如下。

- 如果  $s_i$  和  $s_j$  是两棵树的根节点，而且这两棵树中包含的节点数相同，则回答是任意的。然后将这两棵树合并在一起，让那个被声明为较小的键作为另一个键的子节点。
- 如果  $s_i$  和  $s_j$  是树的根节点，而且一棵树的节点多于另一棵树，较小树的根节点被声明为较小的，在合并这两棵树时，使这个根节点成为另一根节点的子节点。
- 如果  $s_i$  为根节点，而  $s_j$  不同，则将  $s_j$  声明为较小者，两棵树不做修改。
- 如果  $s_i$  和  $s_j$  都不是根节点，则答案应当与之前的指定值一致，而且两棵树不做修改。

图 8-11 显示了在向对手展示锦标赛方法和一个大小为 8 的实例时，如何合并两棵树。图中只给出了到锦标赛完成时的比较。在此之后，还要完成更多的比较，以在输给最大键的键中找出最大者，而树不发生变化。

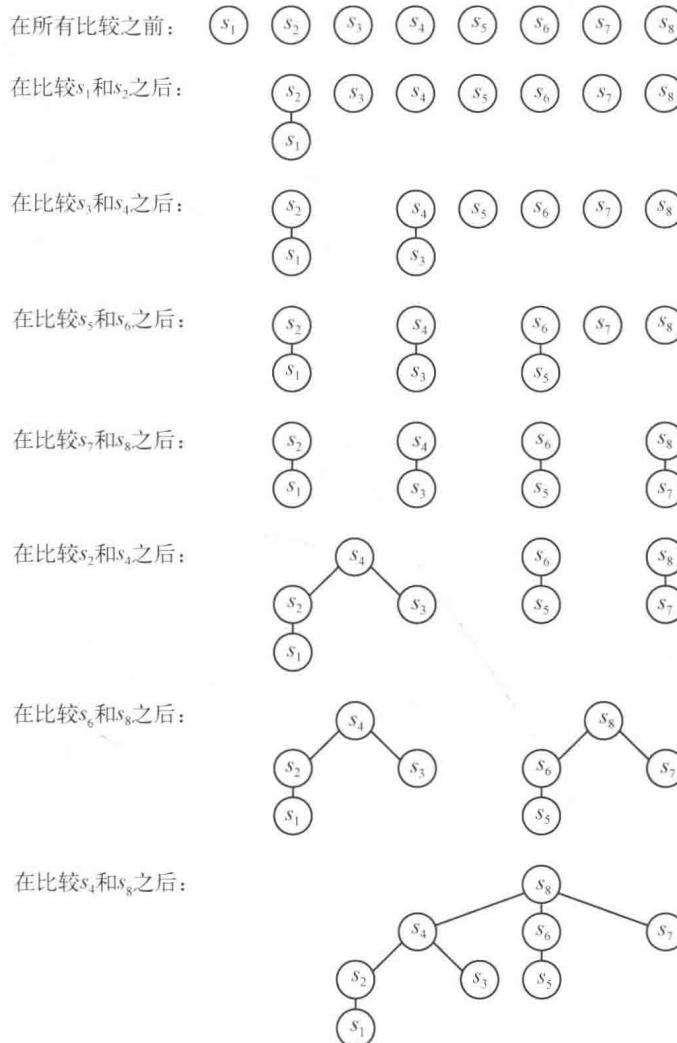


图 8-11 在向定理 8.9 中的对手展示锦标赛方法且输入大小为 8 时，它生成的树

设某个键在赢取了第  $k$  次比较之后恰为最大键， $\text{size}_k$  是以此键为根节点的树中的节点数。于是，

$$\text{size}_k \leq 2\text{size}_{k-1}$$

这是因为，在败者键的树中，其节点数不可能大于胜者树中的节点数。初始条件为  $\text{size}_0=1$ 。可以利用附录B中的方法求解此递推关系，得出结论：

$$\text{size}_k \leq 2^k$$

如果一个键在算法停止时为根节点，那它从来没有输过任何比较。因此，如果当算法停止时还存在两棵树，那有两个键从来没有输过任何比较。这两个键中至少有一个未被报告为第二大的键。我们可以创建一个新的输入，所有其他值都相同，只修改两个根节点的值，使未被报告为第二大的键真正成为第二大的键。对于这一输入，此算法会给出错误答案。这样，在算法停止时，所有  $n$  个键都会在一棵树中。显然，这个树的根节点必然是最大键。因此，如果  $m$  是最大键获胜的总比较次数，则

$$\begin{aligned} n = \text{size}_m &\leq 2^m \\ \lg n &\leq m \\ \lceil \lg n \rceil &\leq m \end{aligned}$$

最后一个式子源于“ $m$  是一个整数”这一事实。定理得证。

根据锦标赛方法的性能达到了下限，所以它是最优的。我们又找到了一个有价值的对手，没有其他对手可以生成更大的下限了。

在最差情况下，至少需要  $n-1$  次比较找出最大键，需要至少  $n+\lceil \lg n \rceil - 2$  次比较找出第二大的键。任何找出第二大键的算法也必须找出最大键，因为要知道一个键是第二大键，必须知道它输掉了一次比较，而且必定是输给了最大键。因此，第二大键的查找要更难一些，这就不足为奇了。

#### 8.5.4 查找第 $k$ 小的键

一般情况下，选择算法需要找出第  $k$  大的键或第  $k$  小的键。到目前为止，我们已经讨论了最大键的查找，因为它与我们使用的术语好像更为吻合。也就是说，将最大键称为胜者更合适一些。现在将讨论找出第  $k$  小的键，因为它使我们的算法变得更为明晰。为简单起见，假定这些键是互不相同的。

一种在  $\Theta(n\lg n)$  时间内找出第  $k$  小的键的方法是对键进行排序，并返回第  $k$  个键。我们设计了一种需要较少比较的方法。

回想算法 2.7 中的过程 `partition`，快速排序（算法 2.6）中用到了这一过程，该过程对一个数组进行划分，使数组中所有小于某一枢纽项的键都在该枢纽项之前，所有大于该枢纽项的键都在它的后面。枢纽项所在的位置称为枢纽点。一直重复划分过程，直到枢纽项位于第  $k$  个位置，从而就能解决选择问题。为此，当  $k$  小于枢纽点时，就对左子树（小于枢纽项的键）进行递归划分；当  $k$  大于枢纽点时，就在右子树进行递归划分。当  $k=\text{pivotpoint}$  时，任务完成。这种分而治之算法采用这一方法解决了问题。

#### 算法 8.5 选择

**问题：**在一个包含  $n$  个不同键的数组  $S$  中，找出第  $k$  小的键。

**输入：**正整数  $n$  和  $k$ ，其中  $k \leq n$ ；由不同键组成的数组  $S$ ，其索引范围为 1 至  $n$ 。

**输出：** $S$  中第  $k$  小的键，它是作为函数 `selection` 的值返回的。

```
keytype selection (index low, index high, index k)
{
    index pivotpoint;

    if (low == high)
        return S[low];
    else {
        partition (low, high, pivotpoint);
        if (k == pivotpoint)
            return S[pivotpoint];
```

```

    else if (k<pivotpoint)
        return selection(low, pivotpoint-1, k);
    else
        return selection (pivotpoint+1, high, k);
}
}

void partition (index low, index high, // 这是出现在算法 2.7 中的同一例程。
               index& pivotpoint)
{
    index i, j;
    keytype pivotitem;

    pivotitem=S[low];           // 为 pivotitem 选择第一项。
    j=low;
    for (i = low+1; i<=high; i++)
        if (S[i]<pivotitem){
            j++;
            交换 S[i] 与 S[j];
        }
    pivotpoint = j;
    交换 S[low] 与 S[pivotpoint]; // 将 pivotitem 放在 pivotpoint。
}

```

和前几章的递归函数一样， $n$  和  $S$  不是函数 selection 的输入。对该函数的顶级调用为：

```
kthsmallest = selection(1, n, k)
```

和在快速排序（算法 2.6）中一样，当每次递归调用的输入中仅减少一项时，发生最差情况。例如，当数组为递增顺序且  $k=n$  时就会发生这一情况。于是，算法 8.5 拥有与算法 2.6 一样的最差时间复杂度，这意味着，就键的比较次数而言，算法 8.5 的最差情况时间复杂度为：

$$W(n) = \frac{n(n-1)}{2}$$

尽管最差情况与快速排序相同，但接下来将会证明，算法 8.5 的平均性能要好得多。

#### ◆ 算法 8.5 的分析 平均情况时间复杂度（选择）

基本运算：partition 中将  $S[i]$  与 pivotitem 的比较。

输入规模： $n$ ，数组中的项数。

假定所有输入的出现概率相同。这就是说，假设所有  $k$  值的输入频率相同，而且所有 pivotpoint 值的返回频率相同。用  $p$  表示 pivotpoint。对于  $n$  个比较结果不存在递归调用（即， $p=k$ ,  $k=1, 2, \dots, n$ ）。对于两种比较结果，其第一次递归调用的输入规模为 1（即， $k=1$  且  $p=2$ ;  $k=n$  且  $p=n-1$ ）。对于 2(2)=4 种比较结果，其第一次递归调用的输入规模为 2（即， $k=1$  或 2 且  $p=3$ ;  $k=n-1$  或  $n$  且  $p=n-2$ ）。下面列出的是对于所有输入规模的输出个数：

第一次递归调用中的输入规模	产生给出规模的输出个数
0	$n$
1	2
2	2(2)
3	2(3)
:	:
$i$	2( $i$ )
:	:
$n-1$	2( $n-1$ )

不难看出，对于其中每一个输入规模， $k$  的所有允许值都以相同频率出现。回想一下，根据算法 2.7 中的所有情况分析，过程 partition 中的比较次数为  $n-1$ 。因此，平均复杂度由以下递推式给出：

$$A(n) = \frac{nA(0) + 2[A(1) + 2A(2) + \dots + iA(i) + \dots + (n-1)A(n-1)]}{n + 2(1 + 2 + \dots + i + \dots + n-1)} + n - 1$$

利用附录 A 例 A.1 中的结果及  $A(0)=0$  的事实，进行化简后可得：

$$A(n) = \frac{2[A(1) + 2A(2) + \dots + iA(i) + \dots + (n-1)A(n-1)]}{n^2} + n - 1$$

接下来将模仿在分析算法 2.6（快速排序）平均情况时间复杂度时使用的方法。也就是说，我们将  $A(n)$  的表达式乘以  $n^2$ ，将该表达式应用于  $n-1$ ，并从  $n-1$  的表达式中减去  $n$  的表达式，化简后可得：

$$\begin{aligned} A(n) &= \frac{n^2 - 1}{n^2} A(n-1) + \frac{(n-1)(3n-2)}{n^2} \\ &< A(n-1) + 3 \end{aligned}$$

因为  $A(0)=0$ ，有以下递推关系：

$A(n) < A(n-1) + 3 \quad (n > 0)$
$A(0) = 0$

如附录 B 中的 B.1 节所述，这一递推关系可使用归纳法求解。它的解是

$$A(n) < 3n$$

同样，可以使用此递推关系来证明， $A(n)$  的界限不超过一个线性函数。因此，

$$A(n) \in \Theta(n)$$

可以很轻松地使用  $A(n)$  的递推关系来证明：对于大的  $n$  值，有

$$A(n) \approx 3n$$

平均来说，算法 8.5（选择）只进行线性次数的比较。当然，此算法的平均性能优于算法 2.6（快速排序）的原因是快速排序两次调用了 partition，而此算法只调用一次。但是，当递归调用的输入为  $n-1$  时，它们都退化为同一复杂度。（在本例中，快速排序在一端输入一个空子数组。）这个时间复杂度是二次时间的。如果可以防止在算法 8.5 中发生这种情况，就可以对最差情况的二次时间复杂度加以改进。接下来说明如何进行这一改进。

最好的情况是 pivotpoint 正好在中间划分数组，因为每次递归调用都会将其折半。回忆一下， $n$  个不同键的中值(median)就是比一半键大、比一半键小的那个键(只有当  $n$  为奇数时才会恰好如此)。如果总能为 pivotitem 选择中值，那就能获得最优性能。但如何确定中值呢？在进程 partition 中，可以尝试调用函数 selection，为其提供一个输入，其中包含原数组和一个大约等于该数组一半大小的  $k$  值。但这样并没有帮助，因为最后还会退入选择，且实例的规模与原实例相同。但是，下面的小技巧是有效的。暂时假定  $n$  是 5 的倍数（我们不需要使用 5，本节最后将讨论这一点）。将  $n$  个键分为  $n/5$  组，每组包含 5 个键。直接找到每一组的中值。习题中将要求你证明，用六次比较可完成这一任务。然后调用函数 selection 来确定这  $n/5$  个中值的中值。中值的中值不一定是  $n$  个键的中值，但如图 8-12 所示，它们是相当接近的。在该图中，最小中值左侧的键（键 2 和键 3）必然小于中值的中值，而最大中值右侧的键（键 18 和键 22）必然大于中值的中值。一般情况下，最小中值右侧的键（键 8 和键 12）和最大中值左侧的键（键 6 和键 14）可位于中值的中值的任一侧。注意，在中值的中值的任一侧，可以有  $2\left(\frac{15}{5}-1\right)$  个键。不难看出，只要  $n$  是 5 的奇数倍，则在中值的中值的任一侧，可以有  $2\left(\frac{n}{5}-1\right)$  个键。因此，在中值的中值的一侧，最多有

$$\underbrace{\frac{1}{2} \left[ n - 1 - 2 \left( \frac{n}{5} - 1 \right) \right]}_{\text{我们知道在一侧的键数}} + 2 \left( \frac{n}{5} - 1 \right) = \frac{7n}{10} - \frac{3}{2}$$

个键。在分析使用这一策略的算法时，会再来看这一结果。首先给出算法。

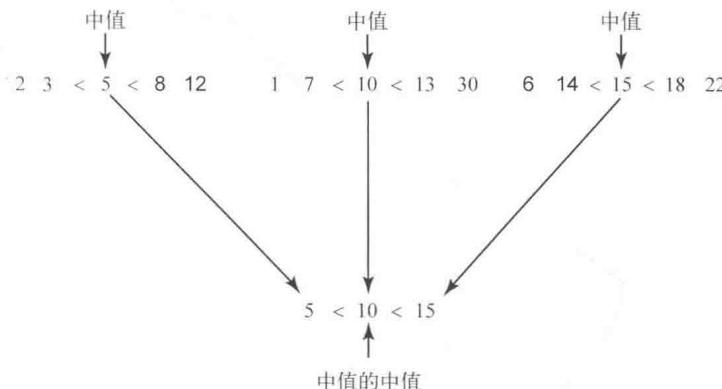


图 8-12 每条竖线表示一个键。我们不知道黑体键是小于还是大于中值的中值

### 算法 8.6 使用中值的选择

问题：在由  $n$  个不同键组成的数组  $S$  中找出第  $k$  小的键。

输入：正整数  $n$  和  $k$ ，其中  $k \leq n$ ；不同键组成的数组  $S$ ，其索引范围为 1 至  $n$ 。

输出： $S$  中第  $k$  小的键，它作为函数 select 的值返回。

```

keytype select (int n,
                keytype S[],
                index k)
{
    return selection2(S, 1, n, k);
}

keytype selection2 (keytype S[],
                     index low, index high, index k)
{
    if (high == low)
        return S[low];
    else{
        partition2(S, low, high, pivotpoint);
        if (k==pivotpoint)
            return S[pivotpoint];
        else if (k < pivotpoint)
            return selection2(S, low, pivotpoint - 1, k);
        else
            return selection2(S, pivotpoint+1, high, k);
    }
}

void partition2 (keytype S[],
                 index low, index high,
                 index& pivotpoint)
{
    const arraysize = high - low + 1;
    const r = [arraysize / 5];
    index i, j, mark, first, last;
    keytype pivotitem, T[1..r];

    for (i = 1; i<=r, i++){

```

```

first = low + 5*i -5;
last = minimum(low + 5*i -1, arraysize);
T[i]= S[first]至 S[last]的中值;
}
pivotitem = select(r, T, [(r+1)/2]); // 中值的近似值。
j = low;
for (i = low; i<= high; i++)
  if (S[i]==pivotitem){
    交换 S[i]与 S[j];
    mark = j;           // 标出 pivotitem 的所在位置。
    j++;
  }
  else if (S[i]<pivotitem){
    交换 S[i]与 S[j];
    j++;
  }
pivotpoint = j-1;
交换 S[mark]与 S[pivotpoint]; // 将 pivotitem 放在 pivotpoint 处。
}

```

与我们的其他递归算法不同，算法 8.6 中给出了一个调用递归函数的简单函数。原因是，需要在两个位置以不同输入调用这一简单函数。也就是说，在 partition2 中以  $T$  为输入调用它，以及在全局范围内调用如下：

```
kthsmallest = select(n, S, k)
```

我们还让这个函数作为递推函数 selection2 的一个输入，因为会调用这一函数来处理全局数组  $S$  和局部数组  $T$ 。

接下来分析该算法。

#### ◆ 算法 8.6 的分析 最差情况时间复杂度（使用中值的选择）

基本运算：partition2 中  $S[i]$  与枢纽项的比较。

输入规模： $n$ ，数组中的项数。

为简单起见，在推导递归方程时假设  $n$  为 5 的奇数倍。此递推式对于一般的  $n$  值也是近似成立的。递推中的各项如下。

- 在从函数 selection2 中调用时，函数 selection2 中的时间。前面已经讨论过，如果  $n$  是 5 的奇数倍，则 pivotpoint 的一侧最终最多有  $\frac{7n}{10} - \frac{3}{2}$  个键。也就是说，这是对 selection2 的这一调用中，最差情况下的输入键数。
- 在从过程 partition2 调用时，函数 selection2 中的时间。在对 selection2 的这一调用中，其输入的键数为  $n/5$ 。
- 求出中值所需要的比较数。如前所述，进行 6 次比较可以找出 5 个数的中值。当  $n$  是 5 的倍数时，算法首先求出  $n/5$  组数的中值（每组恰好 5 个数）。因此，为找出中值，所需要的比较次数为  $6n/5$ 。
- 划分数组所需要的比较数。这一数字为  $n$ （假定比较的实现非常高效）。

我们已经推出以下递推关系：

$$\begin{aligned}
W(n) &= W\left(\frac{7n}{10} - \frac{3}{2}\right) + W\left(\frac{n}{5}\right) + \frac{6n}{5} + n \\
&\approx W\left(\frac{7n}{10}\right) + W\left(\frac{n}{5}\right) + \frac{11n}{5}
\end{aligned}$$

有可能证明，即使  $n$  不是 5 的奇数倍，近似式也是成立的。当然， $W(n)$  并非真的拥有非整数输入。但是，对这些输入的考虑可以简化我们的分析。这一递推式没有给出任何可以在归纳论证中使用的明显答案。此外，它也无法利用附录 B 中的任何其他方法加以求解。但是，可以使用一种称为建设性归纳（structive induction）的方法，获得一种可以在归纳论证中使用的候选解。也就是说，因为我们怀疑  $W(n)$  是线性的，所以就假定对

于所有  $m < n$  及某一常数  $c$ ,  $W(m) \leq cm$ 。由此递归式可得：

$$W(n) \approx W\left(\frac{7n}{10}\right) + W\left(\frac{n}{5}\right) + \frac{11n}{5} \leq c\frac{7n}{10} + c\frac{n}{5} + \frac{11n}{5}$$

因为我们希望能得出  $W(n) \leq cn$  的结论, 所以需要求解

$$c\frac{7n}{10} + c\frac{n}{5} + \frac{11n}{5} \leq cn$$

以确定一个  $c$  值, 可以在归纳论证中使用。它的解是

$$22 \leq c$$

于是选择满足该不等式的最小  $c$  值, 接下来以正式的归纳论证来证明, 当  $n$  不是很小时, 最差情况时间复杂度的界限近似为:

$$W(n) \leq 22n$$

显然, 此不等式对于  $n \leq 5$  成立。完整的归纳论证过程留作习题。定理 8.7 表明,  $k=1$  时的求解, 只需要线性时间。可以得出结论:

$$W(n) \in \Theta(n)$$

我们已经采用一种在最差情况下为线性的算法, 成功地解决了选择问题。前面曾经提到, 并不一定要把数组分为大小为 5 的小组才能解决这一问题。如果  $m$  为小组大小, 任何  $m \geq 5$  的奇数值都会得到一个线性时间复杂度。现在给出原因。至于如何推导出我们描述的结果, 留作习题。对于任意  $m$  值, 最差时间复杂度的递归方程式为:

$$W(n) \approx W\left(\frac{(3m-1)n}{4m}\right) + W\left(\frac{n}{m}\right) + an \quad (8.2)$$

其中  $a$  是一个正常数。右侧表达式中  $n$  的系数之和为:

$$\frac{3m-1}{4m} + \frac{1}{m} = \frac{3m+3}{4m}$$

不难看出, 当且仅当  $m > 3$  时, 右侧的表达式小于 1。可以证明, 若  $p+q < 1$ , 则以下递归式

$$W(n) = W(pn) + W(qn) + an$$

描述了一个线性方程。因此, 对于所有  $m \geq 5$ , 递归式 8.2 描述了一个线性方程。

当  $m=3$  时, 递归式 8.2 如下:

$$W(n) \approx W\left(\frac{2n}{3}\right) + W\left(\frac{n}{3}\right) + \frac{5n}{3}$$

利用归纳法, 有可能证明: 对于这一递归式, 有

$$W(n) \in \Omega(n \lg n)$$

因此, 5 是给出线性特性的最小奇数  $n$  值。

当  $m=7, 9$  或  $11$  时, 时间复杂度中  $c$  的上限稍大于  $m=5$  时的结果。当  $m$  增大到超过 11 时,  $c$  的值增长得非常缓慢。当  $m$  不是很小时, 有可能证明  $c$  近似为  $4 \lg m$ 。例如, 若  $m=100$ , 则此常数大约为  $4 \lg 100 = 26.6$ 。

前面介绍的这种解决选择问题的线性时间算法由 Blum、Floyd、Pratt、Rivest 和 Tarjan 给出 (1973 年)。原来的版本更复杂一些, 但它的键比较次数仅为大约  $5.5n$ 。

Hyafil (1976 年) 已经证明了, 当  $k > 1$  时, 在一组  $n$  个键中找出第  $k$  小的键的下限为:

$$n + (k-1) \left\lceil \lg \left( \frac{n}{k-1} \right) \right\rceil - k$$

其证明也可以在 Horowitz 和 Sahni (1978年) 的文献中找到。注意，定理 8.9 是这一结果的一个特例。

其他选择算法和下限可以在 Schonhage、Paterson 和 Pippenger (1976年) 及 Fussenegger 和 Gabow (1976年) 的文献中找到。

### 8.5.5 选择问题的一种概率算法

在为算法计算下限时，假定了这些算法是确定性的，但我们也曾提到，这些界限对于概率算法也是成立的。下面给出选择问题的一种概率算法，用以说明这种算法在什么情况下是有用的，以此结束对这一主题的讨论。

5.3节给出了一种概率算法，即用于近似计算回溯算法效率的蒙特卡洛算法。回想一下，蒙特卡洛算法不一定会给出正确答案，而是给出答案的一个估计值，随着算法可用次数的增加，此估计值接近正确答案的概率也会增加。下面将给出一种不同的概率算法——Sherwood 算法。

**Sherwood 算法**总是给出正确答案。能让这样一种算法大展身手的情景是：当某一确定算法的平均运行速度远快于它在最差情况下的运行速度时。回想一下，算法 8.5(选择算法)就是如此。当一个特定输入的 pivotpoint 在递归调用中反复接近于 low 或 high 时，该算法的复杂度为最差情况下的二次时间。例如，当数组为升序排列，且  $k=n$  时即是如此。因为对于大多数输入来说都不是这样，所以该算法的平均性能是线性的。假定，对于一个特定输入，根据均匀分布随机选择枢纽项，那在针对此输入运行此算法时，pivotpoint 远离端点的概率要高于接近端点的概率。(附录 A 中的 A.8 节回顾了有关随机性的内容。)因此，获得线性性能的概率更高一些。对所有输入进行平均后，比较次数是线性的，因此，依靠直觉来看，当根据均匀分布随机选择枢纽项时，对于一个特定输入完成的键比较次数的期望值应当是线性的。我们后面将对此进行证明，但先来强调一下这个期望值与前面为算法 8.5 获得的平均值之间的区别。假定所有可能输入的出现次数相同，则算法 8.5 完成的平均比较次数是线性的。对于任意给定输入，算法 8.5 执行的比较次数总是相同的（对于某些输入来说，这一次数为二次函数）。而下面将要介绍的 Sherwood 算法，它在每次针对同一给定输入运行时，所执行的比较次数并不一定相同。对于任意给定输入，有时完成的比较次数是线性，有时则是平方次数的。但是，如果使用同一输入将该算法运行多次，则可以预期运行时间的均值为线性的。

你可能会问，既然算法 8.6(使用中值的选择)能够保证线性性能，那为什么还希望使用这样一种算法呢。原因在于，在近似求解中值时需要一定的开销，所以算法 8.6 中有一个很高的常数。对于一个给定输入，Sherwood 算法的平均运行时间要快于算法 8.6。到底是应当使用算法 8.6 还是使用 Sherwood 算法，取决于具体应用的需求。如果最重要的是要平均性能更好一些，那应当使用 Sherwood 算法。如果无法忍受平方性能，那应当使用算法 8.6。我们再次强调，Sherwood 算法相对于算法 8.5(选择)的优势。只要输入是均匀分布的，算法 8.5 的平均性能也要优于算法 8.6。但是，在一些具体应用中，输入总是接近算法 8.5 的最差情况。在这种应用中，算法 8.5 总是呈现二次时间性能。Sherwood 算法可以在任意应用中避免这一难题，其对策就是根据均匀分布随机选择枢纽项。

下面介绍概率 (Sherwood) 算法。

#### 算法 8.7 概率选择

**问题：**在由  $n$  个不同键组成的数组  $S$  中找出第  $k$  小的键。

**输入：**正整数  $n$  和  $k$ ，其中  $k \leq n$ ；不同键组成的数组  $S$ ，其索引范围为 1 至  $n$ 。

**输出：** $S$  中第  $k$  小的键，它作为函数 selection3 的值返回。

```
keytype selection3 (index low, index high, index k)
{
    if (low == high)
        return S[low];
    else {
        partition3(low, high, pivotpoint);
        if (k == pivotpoint)
            return S[pivotpoint];
        else if (k < pivotpoint)
```

```

    return selection3(low, pivotpoint-1,k);
else
    return selection3(pivotpoint+1, high, k);
}

void partition3 (index low, index high,
                 index& pivotpoint)
{
    index i, j, randspot;
    keytype pivotitem;
    randspot = 根据均匀分布在 low 和 high(含)之间随机选择的 index;
    pivotitem = S[randspot];           // 随机选择 pivotitem。
    j = low;
    for (i = low+1; i <= high; i++)
        if (S[i]<pivotitem){
            j++;
            交换 S[i]和 S[j];
        }
    pivotpoint = j;
    交换 S[low]和 S[pivotpoint];    // 将 pivotitem 放在 pivotpoint。
}

```

算法 8.7 与算法 8.5 的唯一区别就是它对枢纽项的随机选择。接下来将证明，比较次数的期望值对于任意输入都是线性的。这一分析必然不同于对算法 8.5（选择）的平均情况分析，因为在该分析中，假定以相同频率输入  $k$  的所有值。我们希望为每个输入都获得一个线性时间期望值。因为每个输入都有一个特定的  $k$  值，所以不能假定所有  $k$  值的频率都是相等的。我们将证明，无论  $k$  取何值，此期望值都是线性的。

#### ◆算法 8.7 的分析 期望值时间复杂度（概率选择）

基本运算：partition 中  $S[i]$  与 pivotitem 的比较。

输入规模： $n$ ，数组中的项数。

假设我们正在一个规模为  $n$  的输入中查找第  $k$  小的键，下表给出了输入的规模，以及在为每个 pivotpoint 值首次递归调用时的新  $k$  值。

pivotpoint	输入规模	$k$ 的新值
1	$n-1$	$k-1$
2	$n-2$	$k-2$
$\vdots$	$\vdots$	$\vdots$
$k-1$	$n-(k-1)$	1
$k$	0	
$k+1$	$k$	$k$
$k+2$	$k+1$	$k$
$\vdots$	$\vdots$	$\vdots$
$n$	$n-1$	$k$

所有 pivotpoint 值的概率都相等，所以关于该期望值有以下递推关系：

$$E(n, k) = \frac{1}{n} \left[ \sum_{p=1}^{k-1} E(n-p, k-p) + \sum_{p=k}^{n-1} E(p, k) \right] + n - 1$$

可以像算法 8.6 的最差情况分析中一样，使用建设性归纳法分析这一递推关系。也就是说，因为我们怀疑此递推关系是线性的，所以寻找一个  $c$  值，通过归纳论证，证明  $E(n, k) \leq cn$ 。这一次给出归纳论证过程，但留下一个练习：证明  $c=4$  是使该式成立的最小常数。

归纳基础：因为当  $n=1$  时不进行比较，所以

$$E(1, k) = 0 \leq 4(1)$$

归纳假设：假定对于所有  $m < n$  及所有  $k \leq m$ , 有

$$E(mk, k) \leq 4m$$

归纳步骤：我们需要证明，对于所有  $k \leq n$ , 有

$$E(n, k) \leq 4n$$

根据递推关系和归纳假设，有

$$E(n, k) \leq \frac{1}{n} \left[ \sum_{p=1}^{k-1} 4(n-p) + \sum_{p=k}^{n-1} 4p \right] + n - 1 \quad (8.3)$$

有

$$\begin{aligned} \sum_{p=1}^{k-1} 4(n-p) + \sum_{p=k}^{n-1} 4p &= 4 \left[ \sum_{p=1}^{k-1} n - \sum_{p=1}^{k-1} p + \sum_{p=k}^{n-1} p \right] \\ &= 4 \left[ (k-1)n - 2 \sum_{p=1}^{k-1} p + \sum_{p=k}^{n-1} p \right] \\ &= 4 \left[ (k-1)n - (k-1)k + \frac{(n-1)n}{2} \right] \\ &= 4 \left[ (k-1)(n-k) + \frac{(n-1)n}{2} \right] \\ &< 4 \left[ k(n-k) + \frac{n^2}{2} \right] \leq 4 \left[ \frac{n^2}{4} + \frac{n^2}{2} \right] = 3n^2 \end{aligned}$$

第三个等式是通过两次应用附录 A 中例 A.1 的结果而得到的。最后一个不等式来自如下事实：一般情况下， $k(n-k) \leq n^2/4$ 。将刚刚获得的结果代入不等式 8.3, 得：

$$E(n, k) < \frac{1}{n} (3n^2) + n - 1 = 3n + n - 1 < 4n$$

我们已经证明，无论  $k$  值如何，都有

$$E(n, k) \leq 4n \in \Theta(n)$$

## 8.6 习题

### 8.1 节

- 假定在使用顺序查找算法（算法 1.1）时，不是从列表的开头开始查找，而是从上一次查找结束时所在的列表索引处开始。进一步假设，我们正在查找的项目是随机选择的，与之前查找的目标无关。在这些假设条件下，平均比较次数为多少？
- 设  $S$  和  $T$  是两个分别拥有  $m$  个和  $n$  个元素的数组。编写一个算法，查找它们的所有公共元素，并存储在数组  $U$  中。证明可以在  $\Theta(n+m)$  时间内完成。
- 在成功查找的假设下，改进二分查找算法（算法 1.5）。分析算法，并用阶的符号给出分析结果。
- 证明：如果  $x$  在数组中，并且位于每个数组位置的概率相等，则二分查找（算法 1.6）平均情况时间复杂度的界限近似为：

$$\lfloor \lg n \rfloor - 1 \leq A(n) \leq \lceil \lg n \rceil$$

提示：根据引理 8.4，对于某个  $k$  值， $n-(2^k-1)$  是最底层一级的节点数。对这些节点 TND 的贡献等于

$(n-2^k-1)(k+1)$ 。将这一表达式加到 $(k-1)2^k+1$ （就是在对二分查找平均情况分析中建立的公式），可以得到该决策树的 TND。

5. 假定所有以下 $2n+1$ 种可能情况的概率相同：

$$\begin{array}{ll} x=s_i & (1 \leq i \leq n) \\ x < s_1 \\ s_i < x \leq s_{i+1} & (1 \leq i \leq n-1) \\ x > s_n \end{array}$$

证明：二分查找算法（算法 1.5）的平均情况时间复杂度的上下限近似为：

$$\lfloor \lg n \rfloor - \frac{1}{2} \leq A(n) \leq \lfloor \lg n \rfloor + \frac{1}{2}$$

提示：参见第 4 题的提示。

6. 完成引理 8.6 的证明。

## 8.2 节

7. 在你的系统上实现二分查找、插值查找和强壮插值查找算法，并使用若干问题实例研究它们的最佳、平均和最差性能。  
 8. 假定各个键为均匀分布，且查找键 $x$ 位于每个数组位置的可能性相同，证明：插值查找的平均情况时间复杂度属于 $\Theta(\lg(\lg n))$ 。  
 9. 假定各个键为均匀分布，且查找键 $x$ 位于每个数组位置的可能性相同，证明：插值查找的最差情况时间复杂度属于 $\Theta((\lg n)^2)$ 。

## 8.3 节

10. 编写一个算法，找出一个二叉查找树中的最大键。分析你的算法，并用阶的符号给出分析结果。  
 11. 定理 8.3 表明，对于一次成功查找，在使用二叉查找树时，对于所有 $n$ 键输入的平均查找时间属于 $\Theta(\lg n)$ 。  
 证明：对于失败查找，这一结果仍然成立。  
 12. 考虑到所有可能情景，编写一个从二叉查找树中删除节点的算法。分析你的算法，并用阶的符号给出分析结果。  
 13. 编写一个算法，由一个键的列表生成一个 3-2 树。分析你的算法，并用阶的符号给出分析结果。  
 14. 编写一个算法，按自然顺序列了一个 3-2 树中的所有键。分析你的算法，并用阶的符号给出分析结果。

## 8.4 节

15. 另一种冲突解决策略是线性探查。根据这一策略，所有元素都存储在由桶（散列表）组成的数组中。说明线性探查方法如何解决在图 8-8 的问题实例中发生的冲突。（线性探查也称为闭合散列。）  
 16. 讨论两种碰撞解决策略——开放散列和线性探查（见第 15 题）的优缺点。  
 17. 编写一个算法，以线性探查作为冲突解决策略，从一个散列表中删除元素。分析你的算法，并用阶的符号给出分析结果。  
 18. 一种称为双重散列的再散列方案，在发生冲突时使用第二个散列函数。设第一个散列函数为 $h$ ，第二个散列函数为 $s$ ，对于一个可用桶，将为其检查散列表的各个位置，这些位置的整个序列由以下等式给出（其中 $p_i$ 是该序列的第 $i$ 个位置）：

$$p_i(\text{key}) = [(k(\text{key}) + i \times s(\text{key}) - 1) \% \text{table\_size}] + 1$$

（%返回用第一个操作数除以第二个操作数得到的余数。）为图 8-8 中的问题实例定义一个第二散列函数，并给出将所有键插入此散列表后的表。

## 8.5 节

19. 修改算法 8.4（通过键的配对找出最小键和最大键），使其在 $n$ （给定数组中的键数）为奇数时可以正常工作，并证明其时间复杂度由下式给出：

$$\frac{3n}{2} - 2 \quad (\text{当 } n \text{ 为偶数时})$$

$$\frac{3n}{2} - \frac{3}{2} \quad (\text{当 } n \text{ 为奇数时})$$

20. 完成定理 8.8 的证明。即证明：若  $n$  为奇数，如果一个确定性算法仅通过键的比较来查找  $n$  个键中的最小键和最大键，则它在最差情况下至少要进行  $(3n-3)/2$  次比较。
21. 为 8.5.3 节讨论的方法编写一种算法，找出一个给定数组中的第二大键。
22. 证明：对于一般的  $n$  值，为找出一个给定数组中的第二大键，8.5.3 节讨论的方法需要的总比较次数为：

$$T(n) = n + \lceil \lg n \rceil - 2$$

23. 证明：进行六次比较可以找出五个数的中值。
24. 使用归纳法证明，算法 8.6（使用中值的选择）的最差时间复杂度由下式给出近似界限：

$$W(n) \leq 22n$$

25. 证明：对于任意  $m$ （组大小），算法 8.6（使用中值的选择）的最差时间复杂度的递推关系由下式给出：

$$W(n) \approx W\left(\frac{(3m-1)n}{4}\right) + W\left(\frac{n}{m}\right) + an$$

- 其中  $a$  是常数。这是 8.5.4 节的递推关系 8.2。
26. 使用归纳法证明，对于以下递推关系，有  $W(n) \in \Omega(n \lg n)$ 。这是 8.5.4 节的递推式 8.2，其中  $m$ （组大小）为 3。

$$W(n) = W\left(\frac{2n}{3}\right) + W\left(\frac{n}{3}\right) + \frac{5n}{3}$$

27. 证明：在算法 8.7（概率选择）的期望值时间复杂度分析中，以下不等式中的常数  $c$  不能小于 4。

$$E(n, k) \leq cn$$

28. 在你的系统上实现算法 8.5、算法 8.6 和算法 8.7（在一个数组中查找第  $k$  小键的选择算法），并使用若干问题实例研究其最佳、平均和最差性能。
29. 编写一个概率算法，判断一个  $n$  元素数组中是否有一个多元素（出现次数最多的元素）。分析你的算法，并用阶的符号给出分析结果。

#### 补充习题

30. 假定一个非常大的有序列表存放在外部存储中。假定不能将此列表放入内部存储中，设计一种查找算法，在此列表中查找一个键。在设计外部查找算法时，应当考虑哪个（些）主要因素？确定这个（些）主要因素，分析你的算法，并用阶的符号给出分析结果。
31. 讨论分别使用以下结构时的优缺点。
- (a) 带有平衡机制的二叉查找树。
  - (b) 3-2 树。
32. 至少给出两种不适用散列的情景实例。
33. 设  $S$  和  $T$  是两个分别包含  $n$  个数的数组，它们已经处于非递减顺序。编写一个在所有  $2n$  个数字中查找中值的算法，其时间复杂度属于  $\Theta(\lg n)$ 。
34. 编写一种概率算法，使用函数 prime 和 factor 对任意整数进行分解因数。函数 prime 是一个布尔函数，如果给定整数是一个质数，则返回 true，如果不是质数则返回 false。函数 factor 返回一个给定合数的非平凡因数。分析你的算法，并用阶的符号给出分析结果。
35. 列出本章讨论的所有查找算法的优缺点。
36. 对于本章讨论的每种查找算法，给出至少两种最适合该算法的情景实例。

# 第 9 章

## 计算复杂度和难解性： NP 理论简介



Garey 和 Johnson (1979 年) 讲述了一个故事，下面是其中的一个场景。假定你在一家公司工作，老板给你分配了一项任务：为某个对公司至关重要的问题找出一种高效解决算法。在经过一个多月的努力之后，你仍然毫无进展。你最终放弃了，到老板面前羞愧地承认自己找不出一种高效算法。老板说要解雇你，找一位更聪明的算法设计师来代替你。你回答说，也可能并不是因为你太笨了，可能是这种高效算法压根就不存在。老板很不情愿地又给了你一个月的时间，让你证明自己的论点。你又熬了一个月的夜，想证实自己的说法，但最终还是失败了。这时，你既没能给出一种高效算法，也没能证明这种算法是不可能存在的。你又濒临被解雇的边缘了，突然，你想起一些伟大的计算机科学家一直在努力为旅行推销员问题寻求高效算法，但从来没有人能找出一种其最差时间复杂度能优于指数复杂度的算法。而且，也从来没有人能证明这种算法是不可能实现的。你看到了最后一线希望，如果你能够证明，若能为自己公司的这个问题找到一种高效算法，就能自动为旅行推销员问题得出一种高效算法，那就意味着，你的老板正在要求你完成一项即便是伟大的计算机科学家也要头疼的任务。你请求老板给你一个证明这一想法的机会，老板勉强同意了。仅仅用了一周的时间，你真的证明了：公司这个问题的高效算法会自动给出旅行推销员问题的高效算法。你不但没有被解雇，而且还升了职，因为你为公司节省了大量资金。你的老板意识到，如果继续投注大量精力来寻求问题的精确解决算法，将是不明智的做法，而应当探索其他途径，比如寻找一种近似解。

我们刚刚描述的这种情景就是计算机科学家在过去 25 年里已经成功完成的任务。我们已经证明，旅行推销员问题和其他数以千计的问题都是同等难度的，这里所说的同等难度是说，如果为其中任何一个问题找到了高效算法，就能为所有这些问题找到高效算法。还从来没有找到这样一种算法，但也从来没有证明这种算法是不可能存在的。这些有趣的问题称为 *NP 完全 (NP-complete)* 问题，也就是本章的焦点。我们说一个不存在高效算法的问题是“难解的”。9.1 节将会更具体地解释“一个问题是难解的”是什么意思。9.2 节表明，当希望判断一个问题是否难解时，必须考虑算法中的输入规模。9.3 节讨论在考虑难解性时可以对问题进行的三种一般分类。9.4 节讨论了 *NP* 理论和 *NP* 完全问题。9.5 节介绍了一些处理 *NP* 完全问题的方法。

### 9.1 难解性

英文字典中对“*intractable*”(难解)的定义是“*difficult to treat or work*”(难以处理)。也就是说，如果一个计算机科学问题难以用计算机解决，那就说该问题是难解的。这一解释太过含糊，很难使用。为使概念更具体一些，现在引入“多项式时间算法”的概念。

**定义** 多项式时间算法 (polynomial-time algorithm) 是指其最差情况时间复杂度的上限是其输入规模的一个多项式函数。也就是说，如果  $n$  为输入规模，则存在一个多项式  $p(n)$ ，使得：

$$W(n) \in O(p(n))$$

例 9.1 具有以下最差时间复杂度的算法都是多项式时间算法：

$$2n \quad 3n^3 + 4n \quad 5n + n^{10} \quad n \lg n$$

具有以下最差情况时间复杂度的算法不是多项式时间算法：

$$-2^n \quad 2^{0.01n} \quad 2^{\sqrt{n}} \quad n!$$

注意， $n\lg n$  不是  $n$  的多项式。但是，因为  $n\lg n < n^2$ ，所以它也以  $n$  的一个多项式为限，这意味着具有这一时间复杂度的算法也满足被称为“多项式时间算法”的标准。

在计算机科学中，如果一个问题不可能用一种多项式时间算法解决，就称该问题难解。我们要强调，难解性是一个问题的一个性质，它不是该问题的任意算法的性质。一个问题要是难解的，必然不存在可以解决它的多项式时间算法。为一个问题找到一种非多项式时间算法，并不能说明这个问题是难解的。例如，链式矩阵乘法问题的暴力算法（见 3.4 节）是非多项式时间算法。利用 3.4 节递归性质的分而治之算法也是如此。但是，该节设计的动态规划算法（算法 3.6）是  $\Theta(n^3)$ 。这个问题不是难解的，因为利用算法 3.5，可以在多项式时间内解决它。

由第 1 章知道，多项式时间算法通常要远优于非多项式时间的算法。再来看表 1-4，可以看出，如果它需要 1 纳秒的时间来处理基本指令，那时间复杂度为  $n^3$  的算法将在 1 毫秒内处理大小为 100 的实例，而时间复杂度为  $2^n$  的算法将需要 10 亿年。

我们可以创建一些极端的例子，对于实际中会出现的输入规模，使非多项式时间算法优于多项式时间算法。例如，如果  $n=1\ 000\ 000$ ，则  $2^{(0.00001)n}=1024$ ，而  $n^{10}=10^{60}$ 。

此外，有许多最差情况时间复杂度不是多项式的算法，它们在许多实际实例中的性能非常高效。许多回溯和分支定界算法就是这样。因此，我们对“难解”的定义是真正“难解性”的唯一一个良好指标。在任何特定情况下，对于我们所考虑的实际输入规模，一个已经找到多项式时间算法的问题可能要比还未找到此种算法的问题更难以解决。

从难解性的角度考虑，可以将问题分为三个一般类别：

- (1) 已经为其找到多项式时间算法的问题；
- (2) 已经证明是难解的问题；
- (3) 未能证明是难解的，但也没有为其找到多项式时间算法的问题。

一个非常让人惊讶的现象是计算机科学中的大多数问题似乎都属于第一类或第三类。

在判断一个算法是否为多项式时间时，要特别留意一个被我们称为“输入规模”的因素。因此，在处理之前，先来更深入地讨论一下输入规模（关于输入规模的最初讨论，请参见 1.3 节）。

## 9.2 再谈输入规模

到目前为止，将  $n$  称为算法的输入规模通常就足够了，因为  $n$  合理地度量了输入中的数据量。例如，在排序算法中，待排序的键数  $n$  很好地度量了输入中的数据量。所以我们将  $n$  称为输入规模。但是，绝对不能漫不经心地就将  $n$  称为一个算法的输入规模。考虑下面的算法，它用来判断一个正整数  $n$  是否为质数。

```
bool prime (int n)
{
    int i; bool switch;
    switch = true;
    i = 2;
    while (switch && i<=[n1/2])
        if (n % i == 0)           // %返回当 n 除以 i 时的余数。
            switch = false;
        else
            i++;
    return switch;
}
```

在这个判断质数的算法中, `while` 循环的执行次数显然属于  $\Theta(n^{\frac{1}{2}})$ 。但是, 它是不是多项式时间算法呢? 参数  $n$  是算法的输入, 它不是输入的规模。也就是说, 每个  $n$  值都构成问题的一个实例。排序算法的情景就不同于此, 其中的  $n$  是键数, 实例是  $n$  个键。如果  $n$  的值是函数 `prime` 的输入, 而不是其输入规模, 那输入的规模又是多少呢? 在回答这个问题之前, 先来给出输入规模的另一个定义, 它要比 1.3 节中的定义更具体一些。

**定义** 对于一个给定算法, 输入规模 (input size) 的定义是为书写其输入所需要的字符数。

这个定义与 1.3 节给出的定义并没有什么不同, 它只不过更具体地说明了如何测量输入的规模。为了计算书写输入所需要的字符数, 需要知道如何对输入进行编码。假定采用计算机内部的二进制编码, 那编码所用字符数就是二进制位数(比特), 对正整数  $x$  进行编码所需要的字符数就是  $\lfloor \lg x \rfloor + 1$ 。例如,  $31 = 1111_2$ ,  $\lfloor \lg 31 \rfloor + 1 = 5$ 。我们直接说, 采用二进制对正整数  $x$  进行编码时, 大约需要  $\lg x$  位。假定采用二进制编码, 有一个对  $n$  个正整数进行排序的算法, 我们希望为其确定输入规模。待排序的整数是此算法的输入。因此, 输入规模就是对其编码所需要的比特数。如果最大整数为  $L$ , 并且对每个整数进行编码时, 所用比特数都是对最大整数编码所需要的比特数, 那对每个整数进行编码将需要大约  $\lg L$  个比特。于是, 当有  $n$  个整数时, 输入规模大约为  $n \lg L$ 。假定现在采用基数 10 对整数进行编码, 用于编码的字符就是十进制数位, 需要大约  $\log_{10} L$  个字符对最大整数进行编码, 当有  $n$  个整数时, 输入规模大约为  $n \log_{10} L$ 。因为  $n \log_{10} L = (\log 2)(n \lg L)$ , 所以, 如果一个算法对于其中一个输入规模为多项式时间, 那对于另一个输入规模也必然为多项式时间。

如果我们仅采用“合理的”编码机制, 那具体使用的编码机制不会影响对一个算法是否为多项式时间的判断。但什么是“合理的”, 似乎并没有一个令人满意的正式定义。但是, 对大多数算法来说, 关于“合理”的含义还是能达成一致的。例如, 对于本书中的任何一个算法, 只要对整数进行编码的基数不是 1, 就不会影响对于一个算法是否为多项式时间的判断。因此可以认为任意此类编码系统都是合理的, 以基数 1 进行的编码(这种编码称为“一元形式”)则不被认为是合理的。

在前面几章, 我们直接将待排序的键数  $n$  称为排序算法的输入规模。在以  $n$  为输入规模时, 我们证明了这些算法是多项式时间的。如果采用输入规模的准确定义, 它们是否仍然是多项式时间的呢? 下面将说明, 答案是肯定的。要对输入规模更精确一些, 那对最差时间复杂度的定义也要更精确一些(相对于 1.3.1 节而言)。它的精确定义如下:

**定义** 对于一个给定算法,  $W(s)$  定义为该算法对输入规模  $s$  所执行的最大步骤数。 $W(s)$  称为该算法的最差时间复杂度。

步骤 (step) 可看作一次机器比较或赋值, 或者为了使分析过程与机器无关, 可以看作一次比特比较或赋值。这一定义也与 1.3 节中的定义没有什么不同。它只是对基本运算表达得更具体一些。也就是说, 根据这一定义, 每个步骤就是基本运算的一次执行。这里输入规模用  $s$  表示, 而不是  $n$ , 是因为(1)算法的参数  $n$  并非总是输入规模的度量(例如, 在本节开头给出的质数判断算法中就是如此); (2)当  $n$  是输入规模的度量时, 它通常也不是一个准确度量。根据刚刚给出的定义, 我们必须计算该算法执行的所有步骤数。现在通过分析算法 1.3(交换排序)来说明如何计算此数目, 同时要避免讨论实现细节。为简单起见, 假定这些键都是正整数, 而且记录中没有其他字段。再次讨论算法 1.3。为递归循环和进行分支所执行的步骤上限为常数  $c$  乘以  $n^2$ 。如果这些整数足够大, 计算机将无法在一个步骤中完成对它们的比较和赋值。在 2.6 节讨论大整数算术运算时曾经见过类似情景。因此, 不应当将一次键比较或一次键赋值看作一个步骤。为使分析与机器无关, 我们将一个步骤看作一次比特比较或一次比特赋值。因此, 如果  $L$  是最大整数, 那至少需要  $\lg L$  个步骤来比较一个整数, 或者赋值一个整数。在 1.3 节和 7.2 节分析算法 1.3 时曾经看到, 为了对  $n$  个正整数进行排序, 最差交换排序要执行  $n(n-1)/2$  次键比较和  $3n(n-1)/2$  次赋值。因此, 交换排序执行的最大步骤数不大于

$$cn^2 + \underbrace{\frac{n(n-1)}{2} \lg L}_{\text{比特比较}} + \underbrace{\frac{3n(n-1)}{2} \lg L}_{\text{比特赋值}}$$

我们以  $s=n \lg L$  作为输入规模，则

$$\begin{aligned} W(s) &= W(n \lg L) \quad \{ \text{用 } n \lg L \text{ 替换 } s \} \\ &\leq cn^2 + \frac{n(n-1)}{2} \lg L + \frac{3n(n-1)}{2} \lg L \\ &< cn^2 (\lg L)^2 + n^2 (\lg L)^2 + 3n^2 (\lg L)^2 \\ &< (c+4)(n \lg L)^2 = (c+4)s^2 \end{aligned}$$

我们已经证明了，在使用输入规模的准确定义时，交换排序仍然是多项式时间的。对于任何一种算法，如果已经使用输入规模的非精确定义证明了它是多项式时间的，那就可以为该算法获得类似结果。此外，可以证明，对于已经证明为非多项式时间的算法（比如算法 3.11），在采用输入规模的精确定义时，仍然是非多项式时间的。可以看出，当  $n$  可以用来度量输入中的数据量时，直接以  $n$  作为输入规模，可以正确地判断出一个算法是否为多项式时间。因此，如果  $n$  可以度量输入数据量，我们就继续以  $n$  作为输入规模。

现在再回头来看质数判断算法。因为此算法的输入是  $n$  的值，所以输入规模就是对  $n$  进行编码所需要的字符数。回想一下，如果使用以 10 为基数的编码，那  $n$  的编码将需要  $\lfloor \log n \rfloor + 1$  个字符。例如，如果该数字为 340，那它的编码将需要 3 个十进制数位，而不是 340 个。一般情况下，如果使用以 10 为基数的编码方式，而且设  $s=\log n$ ，那  $s$  就近似为输入的规模。在最差情况下，函数 prime 中将会执行  $\lfloor n^{1/2} \rfloor - 1$  遍循环。因为  $n=10^s$ ，所以最差情况下的循环执行次数大约为  $10^{s/2}$ 。因为总步骤数至少等于该循环的执行遍数，所以时间复杂度为非多项式的。如果使用二进制编码，那  $n$  的编码将需要大约  $\lg n$  个字符。因此，如果使用二进制编码， $r=\lg n$  将约等于输入规模，循环的执行遍数大约等于  $2^{r/2}$ 。时间复杂度仍然是非多项式的。只要我们使用“合理的”编码方案，这一结果就不会改变。前面曾经提到，我们不认为一元编码是“合理的”。如果使用这种编码方案，那对数字  $n$  的编码就需要  $n$  个字符。例如，数字 7 将被编码为 1111111。利用这一编码方案，质数检查算法将具有多项式时间复杂度。由此看出，如果偏离了“合理的”编码方案，结果就会发生改变。

在像质数检查这样的算法中，将  $n$  称为输入的大小（magnitude）。我们已经看到其他一些算法，它们的时间复杂度就大小而言是多项式的，但就规模而言则不是多项式的。用于计算第  $n$  个斐波那契项的算法 1.7，其时间复杂度属于  $\Theta(n)$ 。因为  $n$  是输入的大小，而且  $\lg n$  衡量了输入的规模，所以算法 1.7 的时间复杂度就大小而言是线性的，但就规模而言是指数的。用于计算二项式系数的算法 3.2，其时间复杂度属于  $\Theta(n^2)$ 。因为  $n$  是输入的大小，而  $\lg n$  衡量的是输入规模，所以算法 3.2 的时间复杂度就大小而言是二次的，而就规模而言则是指数的。4.5.4 节讨论的用于解决 0-1 背包问题的动态规模算法，其时间复杂度属于  $\Theta(nW)$ 。在这一算法中， $n$  是规模的度量，因为它就是输入中的项数。但  $W$  是一个大小，因为它是背包的最大容量， $\lg W$  衡量的是  $W$  的规模。这个算法的时间复杂度就大小来说是多项式的，但就规模来说，它是指数的。

如果一个算法的最差时间复杂度可以由其规模和大小的多项式函数划定上限，则称之为伪多项式时间的（pseudopolynomial-time）。这种算法经常是非常有用的，因为只有当实例中包含非常大的数字时，它的效率才会显得低下。这些实例可能不属于我们关注的应用。例如，在 0-1 背包问题中，我们通常只关注  $W$  不是特别大的情景。

## 9.3 三类一般问题

下面讨论可根据难解性对问题进行划分的三个一般类别。

### 9.3.1 已经找到多项式时间算法的问题

任何已经找到多项式时间算法的问题都属于这个类别。我们已经为排序找到了  $\Theta(n \lg n)$  算法，为查找有序

数组找到了一个  $\Theta(\lg n)$  算法，为矩阵乘法找到了一个  $\Theta(n^{2.38})$  算法，为链式矩阵乘法找到了一个  $\Theta(n^3)$  算法，等等。这些问题中有许多都有一些不是多项式时间的算法。前面已经提到，链式矩阵乘法算法就是如此。还有其他一些问题，我们已经为其找到了多项式时间算法，但其容易想到的暴力算法是非多项式的，这些问题包括最短路径问题、最优二叉查找树问题和最小生成树问题。

### 9.3.2 已经证明难解的问题

这一类别中有两种问题。第一种问题需要的输出数为非多项式的。回想一下 5.6 节判断所有哈密顿回路的问题。如果从每个节点到每个其他节点都有一条边，那就会有  $(n-1)!$  个此种回路。为求解此问题，算法必须输出所有这些回路，这意味着我们的要求是不合理的。我们在第 5 章注意到，当时在表述问题时要求给出所有解，因为这样可以给出不是那么混乱的算法，但这些算法在经过很简单的修改后，都可用于求解仅要求一个解的问题。仅要求一个回路的哈密顿回路问题显然不是这种问题。尽管了解这种难解性也是很重要的，但诸如此类的问题通常并没有什么难度。一般情况下，很容易就能意识到所要求的输出数不是多项式函数，而一旦意识到这一点，就能想到，这只是因为我们要求提供的信息过多了，超出了真正的需要。也就是说，这个问题的定义不是那么现实。

如果我们的请求是合理的（也就是说，没有要求提供非多项式数量的输出），而且可以证明无法在多项式时间内解决某个问题，那就出现了第二种难解性。但奇怪的是，目前只发现了极少数的此类问题。第一种问题称为不可判定问题。之所以将这些问题称为“不可判定的”，是因为可以证明不存在能够解决这些问题的算法。这些问题中最著名的就是停机问题。在这个问题中，我们以任意算法及该算法的任意输入作为问题的输入，判断在将该算法应用于其输入时，是否会停止运行。1936 年，阿兰·图灵证明了这一问题是不可判定的。1953 年，A. Grzegorczyk 设计了一种难解的可判定问题。Hartmanis 和 Stearns (1965 年) 讨论了类似结果。但是，这些问题都是为了具备某些特定性质而“人为”构造的。在 20 世纪 70 年代早期，一些现在存在的可判定决策问题被证明是难解的。一个决策问题的输出就是一个简单的回答——“是”或“否”。因此，所需要的输出量当然是合理的。最著名的此类问题之一就是“Presburger 算术运算”，Fischer 和 Rabin 在 1974 年证明了它是难解的。这一问题及其难解性的证明，都可以在 Hopcroft 和 Ullman (1979 年) 的文献中找到。

目前被证明为难解的所有问题都不属于 9.4 节讨论的  $NP$  集合，这一点也已经得到证实。但是，大多数看起来难解的问题都属于  $NP$  集合。接下来就讨论这些问题。前面曾经提到，一个多少令人惊讶的现象是，已经被证实为难解的问题是相对较少的，直到 20 世纪 70 年代早期，才有一个现实决策问题被证实是难解的。

### 9.3.3 未被证明是难解的，但也从来没有找到多项式时间算法的问题

对于这类问题，从来没有为其找到过一种多项式时间算法，但也从来没有人证明这种算法是不可能存在的。前面已经讨论过，这种问题有许多。例如，如果我们将问题表述为仅要求一个解，那 0-1 背包问题、旅行推销员问题、子集求和问题、 $m \geq 3$  时的  $m$  着色问题、哈密顿回路问题、贝叶斯网络中的溯回推理问题，都属于此类问题。我们已经为这些问题找到了分而治之算法、回溯算法和其他一些对于许多大型实现都非常高效的算法。也就是说，如果问题实例取自一些受限集合，那就存在  $n$  的一个表达式，作为所完成基本运算次数的上限。但是，对于所有实例组成的集合，并不存在这样一个多项式。为证明这一点，只需找出实例的某个无穷序列，对于这个序列，不存在  $n$  的一个多项式，可以作为所完成基本运算次数的上限。回想一下，第 5 章对回溯算法就是这样做的。

这一类别中的许多问题之间都存在一种紧密而有趣的关系。下一节将找出这一关系。

---

## 9.4 NP 理论

---

如果在开始时仅限于判定问题，那这一理论的提出要更方便一些。回想一下，一个判定问题的输出就是简单的“是”或“否”。但在介绍前面提到的一些问题时（在第 3、4、5、6 章），我们将它们表述为最优化问题，

也就是说，其输出是一个最优解。但每个最优化问题都有一个相应的判定问题，如下面的例子所示。

### 例 9.2 旅行推销员问题

假设给定一个加权有向图。回想一下，这样一个图中的一条**旅程**就是一条路径，它始于一个顶点，终止于该顶点，并将图中所有顶点都恰好访问一次。**旅行推销员最优化问题**就是确定一条旅程，使其各边上的总权重最小。

**旅行推销员判定问题**就是对于一个给定的正整数  $d$ ，判断是否存在一条旅程，使其总权重大于  $d$ 。这个问题的参数与旅行推销员最优化问题相同，只是增加了一个参数  $d$ 。

### 例 9.3 0-1 背包问题

回忆一下，0-1**背包最优化问题**就是假设每件物品都有自己的重量和价值，并且背包的最大可容纳重量为  $W$ ，然后计算背包中所能容纳物品的最大总价值。

0-1**背包判定问题**就是对于一个给定的价值  $P$ ，判断是否可以找出一种装包方式，使总重量不大于  $W$ ，而总价值至少等于  $P$ 。这个问题的参数与0-1背包最优化问题相同，只是增加了一个参数  $P$ 。

### 例 9.4 图的着色问题

**图的着色最优化问题**是确定为一幅图进行着色的最少颜色数量，使任何两个相邻顶点的颜色都不相同。这个数量称为这幅图的**色彩数**。

**图的着色判定问题**就是对于一个整数  $m$ ，判断是否存在一种着色方案，最多使用  $m$  种颜色，且任何两个相邻顶点的颜色都不相同。这个问题的参数与图的着色最优化问题相同，只是增加了一个参数  $m$ 。

### 例 9.5 分团（clique）问题

无向图  $G=(V,E)$  中的一个**分团**是  $V$  的一个子集  $W$ ，使得  $W$  中的每个顶点都与  $W$  中的所有其他顶点相邻。对于图 9-1 中的图， $\{v_2, v_3, v_4\}$  是一个分团，而  $\{v_1, v_2, v_3\}$  则不是分团，因为  $v_1$  不与  $v_3$  相邻。**最大分团**是规模最大的分团。图 9-1 所示图中的唯一最大分团是  $\{v_1, v_2, v_4, v_5\}$ 。

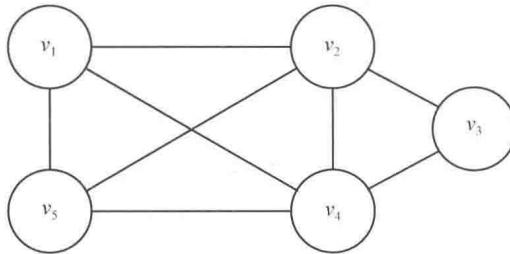


图 9-1 最大分团为  $\{v_1, v_2, v_4, v_5\}$

**分团最优化问题**就是为一个给定图确定最大分团的大小。

**分团判定问题**就是对于一个正整数  $k$ ，确定是否存在一个至少包含  $k$  个顶点的分团。这个问题的参数与分团最优化问题相同，只是增加了一个参数  $k$ 。

对于上述任意例子的判定问题和最优化问题，都未能找到多项式时间算法。但是，如果可以为其中任何一个例子的最优化问题找到多项式时间算法，那也就能够为相应的判定问题找到多项式时间算法。之所以如此，是因为一个最优化问题的解可以为相应的判定问题生成一个解。例如，如果知道旅游推销员最优化问题一个具体实例的最优旅程总权重为 120，那对于  $d \geq 120$  来说，相应判定问题的答案就是“是”，否则为“否”。同样，如果知道 0-1 背包最优化问题一个实例的最优价值为 230 美元，那对于  $p \leq 230$  美元来说，相应判定问题的答案就是“是”，否则为“否”。

因为一个最优化问题的多项式时间算法会自动为相应的判定问题生成一个多项式时间算法，所以在刚开始提出理论时，可以仅考虑判定问题。我们接下来就是这样做的，之后会再来讨论最优化问题。届时将会看到，通常可以证明：一个最优化问题甚至与其相应的判定问题之间存在更紧密的关系。也就是说，对于许多判定问题（包括前面例子中的问题），可以证明：判定问题的多项式时间算法也会给出相应最优量的多项式时间算法。

### 9.4.1 集合 $P$ 和 $NP$

首先考虑可以用多项式时间算法解决的判定问题集。有以下定义。

**定义**  $P$  是所有可以由多项式时间算法解决的判定问题之集。

$P$  中都有哪些问题呢？所有已经为其找到多项式时间算法的判定问题当然属于  $P$ 。例如，判定一个键是否在一个数组中的问题、判定一个键是否在一个有序数组中的问题都属于  $P$ 。第 3 章和第 4 章介绍了一些已经为其找到了多项式时间算法的最优化问题，这些问题的相应判定问题也属于  $P$ 。但是，一些尚未为其找到多项式算法的判定问题是否也属于  $P$  呢？例如，旅行推销员判定问题是否属于  $P$  呢？尽管还没有人找到一种可以解决此问题的多项式时间算法，但也没有人能证明不可能用多项式时间算法来解决它。因此，它可能属于  $P$ 。要知道一个判定问题不属于  $P$ ，必须证明不可能为其设计出多项式时间算法。人们并没有为旅行推销员判定问题进行这一证明。这些考虑事项对于例 9.2 至例 9.5 中的其他判定问题同样成立。

哪些判定问题不属于  $P$  呢？因为我们不知道例 9.2 至例 9.5 中的判定问题是否属于  $P$ ，所以它们中的每一个都可能不属于  $P$ ，只是我们并不知道罢了。此外，同一类别中还有数以千计的判定问题。也就是说，我们不知道它们是否属于  $P$ 。Garey 和 Johnson (1979 年) 讨论了其中许多问题。其实，我们可以确信不属于  $P$  的判定问题是相对较少的。这些问题就是那些我们已经证明不可能存在多项式时间算法的判定问题。9.3.2 节讨论了这种问题。前面已经说过，Presburger 算术运算问题是最著名的一个。

接下来会定义一个可能更为广泛的判定问题集，其中包含了例 9.2 到例 9.5 的问题。为引出这一定义，先来更深入地讨论一下旅行推销员判定问题。假定某人声称自己知道该问题某一实例的答案为“是”。也就是说，这个人说，对于某个图和数字  $d$ ，存在一条旅程，其总权重不大于  $d$ 。如果我们要求这个人实际给出一条总权重不大于  $d$  的旅程，以此来“证实”他的说法，应当也是合乎情理的。如果这个人给出了什么结果，那我们就能编写下面的算法，验证他们给出的结果是不是一个权重不大于  $d$  的旅程。此算法的输入就是图  $G$ 、距离  $d$  和字符串  $S$ ，后者就是声称其权重不大于  $d$  的旅程。

```
bool verify (weighted_digraph G,
             number d,
             claimed_tour S)
{
    if (S 是一条旅程 && S 中各边的总权重<=d)
        return true;
    else
        return false;
}
```

这个算法首先检查  $S$  是否真的是一条旅程。如果是，该算法随后将这条旅程上的权重相加。如果这些权重之和不大于  $d$ ，则返回 “true”。这意味着它已经验证了那个“是”，的确有一条总权重不大于  $d$  的旅程，我们知道这个判定问题的答案为“是”。如果  $S$  不是旅程，或者权重之和超过了  $d$ ，那算法就会返回 “false”。返回  $false$  只是意味着所声称的旅程不是总权重不大于  $d$  的旅程。并不能表明不存在这样的旅程，因为可能会有其他一条不同旅程的总权重不大于  $d$ 。

在习题中将要求你更具体地实现这一算法，并证明它是多项式时间的。这就是说，给定一条候选旅程，可以在多项式时间内对这一候选旅程进行验证，看它能否证明相关判定问题的答案为“是”。如果最后发现所谓的旅程根本就不是旅程，或者其总长度大于  $d$ （那个人也许在撒谎），并不能证明判定问题的答案一定为“否”。因此，我们讨论的并不是说能在多项式时间内验证判定问题的答案为“否”。

这种可以在多项式时间内进行验证的特性是集合  $NP$  中的问题所拥有的，集合  $NP$  的定义将在下面给出。这并不是说这些问题一定可以在多项式时间内得以解决。在验证一条候选旅程的总权重不大于  $d$  时，并没有包含找出该旅程所花费的时间。我们只是说，验证部分耗用的是多项式时间。为了更具体地表述“多项式时间可

验证性”的概念，我们引入非确定性算法（nondeterministic algorithm）的概念。可以将这样一种算法看作由以下两个步骤组成。

(1) 猜测（非确定性）阶段：给定一个问题的一个实例，这一阶段只是生成某个字符串  $S$ 。这个字符串可以看作对该实例某一答案的猜测。但是，它也可能是一个毫无意义的字符串。

(2) 验证（确定性）阶段：上述实例和字符串  $S$  是这一阶段的输入内容。这一阶段以通常的确定形式进行，或者(a)在最终停止时输出“true”，意思是它已经验证了这一实例的答案为“是”；或者(b)在停止时给出输出“false”；或者(c)根本就不停止（也就是说，进入一个无限循环）。在后两种情况中，没有验证这个实例的回答为“是”。后面将会看到，就我们的目的而言，这两种情况是无法区分的。

函数 verify 完成旅行推销员判定问题的验证阶段。注意，它是一个普通的确定性算法。不确定的是猜测阶段。这一阶段称为非确定性的，是因为没有为其指定不可变更的逐步操作指令。相反，在这一阶段，计算机可以以任意方式生成任意字符串。“非确定性阶段”的提出只是为了给出多项式时间可验证性的概念。它并不是求解判定问题的现实方法。

尽管我们从来没有实际使用一种非确定性算法来解决某个问题，但如果满足以下条件，就说一个非确定性算法“解决”了一个判定问题。

(1) 对于任何一个实例，如果它的回答为“是”，那就存在某个字符串  $S$ ，使验证阶段返回“true”。

(2) 对于任何一个实例，如果它的回答为“否”，那就不存在使验证阶段返回“true”的字符串。

当实例为图 9-2 中的图， $d$  为 15，并为函数 verify 提供一些输入字符串  $S$  时，其结果将如下表所示。

$S$	输出	原因
$[v_1, v_2, v_3, v_4, v_1]$	False	总权重大于 15
$[v_1, v_4, v_2, v_3, v_1]$	False	$S$ 不是旅程
$\#@12*&%a\ $	False	$S$ 不是旅程
$[v_1, v_3, v_2, v_4, v_1]$	True	$S$ 是一条总权重不大于 15 的旅程

第三个输入表明， $S$  可以是一个完全没有意义的字符串（前文曾对此进行了讨论）。

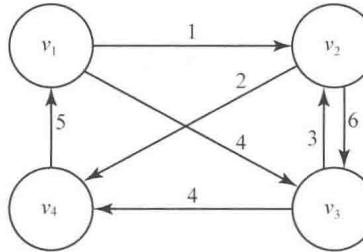


图 9-2 旅程 $[v_1, v_3, v_4, v_1]$ 的总权重不大于 15

一般来说，如果对一个具体实例的回答为“是”，且函数 verify 的输入是一个总权重不大于  $d$  的旅程，该函数会返回“true”。因此，非确定性算法的判据(1)满足。另一方面，只有当输入的总权重不大于  $d$  时，函数 verify 才会返回“true”。因此，如果对一个实例的回答为“否”，那函数 verify 对于任意  $S$  值都不会返回“true”，也就是说，判据(2)得到满足。这样，一个只是在猜测阶段生成字符串，并在验证阶段调用函数 verify 的非确定性算法“解决”了旅行推销员判定问题。接下来将确定“多项式时间非确定性算法”的含义。

**定义 多项式时间非确定性算法** (polynomial-time nondeterministic algorithm) 是一种非确定性算法，其验证阶段是一个多项式时间算法。

现在可以定义 NP 集合了。

**定义**  $NP$  是指可以用多项式时间非确定性算法解决的所有判定问题组成的集合。

注意,  $NP$  的含义是“Nondeterministic Polynomial”(非确定性多项式)。对于  $NP$  中的一个判定问题, 必然存在一种可以在多项式时间内完成验证的算法。因为对于旅游推销员判定问题就是如此, 所以该问题属于  $NP$ 。必须强调, 这并不是说我们必然有一个可以解决该问题的多项式时间算法。实际上, 到目前为止, 都没有为旅行推销员判定问题找到这样一个算法。如果对于该问题一个特定实例的回答为“是”, 那可能要在非确定性阶段尝试所有旅程, 然后才能找到一条会让 verify 返回“true”的路径。如果从每个顶点到所有其他顶点之间都存在一条边, 那就会有 $(n-1)!$ 条旅程。因此, 如果尝试所有旅程, 那就不可能在多项式时间内找到答案。此外, 如果一个实例的答案为“否”, 那使用这一方法来求解该问题时, 就绝对需要尝试所有旅程。之所以要引入非确定性算法与  $NP$  的概念, 目的是为了对算法进行分类。相对于生成和验证字符串的算法, 通常会有更好的算法来实际解决一下问题。例如, 旅行推销员问题的分而治之算法(算法 6.3)的确可以生成旅程, 但它使用定界函数避免了生成太多旅程。因此, 它比盲目生成旅程的算法要好得多。

$NP$  中还有其他什么样的判定问题呢? 在习题中将要求你来证明, 例 9.2 至例 9.5 中的其他判定问题都属于  $NP$ 。

此外, 还有数以千计的其他问题, 没人能用多项式时间算法来解决, 但已经证明它们属于  $NP$ , 因为已经为它们设计了多项式时间非确定性算法。(Garey 和 Johnson 1979 年的文献中有许多此类问题。)最后, 还有大量问题是明显属于  $NP$  的。也就是说,  $P$  中的每个问题也都属于  $NP$ 。这一点明显成立, 因为  $P$  中的任何一个问题都可以用一种多项式时间算法来解决。这样, 在非确定性阶段可以随便生成任何没有意义的内容, 然后在确定性阶段运行那个多项式时间算法。因为此算法解决这个问题的方法就是回答“是”或“否”, 所以对于任意给定的输入字符串  $S$ , (对于一个回答就为“是”的实例,) 验证回答的确为“是”。

哪些判定问题不属于  $NP$  呢? 让人奇怪的是, 已经被证实不属于  $NP$  的判定问题, 就是那些已被证实为难解的问题, 别无其他。也就是说, 停机问题、Presburger 算术运算问题和 9.3.2 节讨论的其他问题, 都已经被证实不属于  $NP$ 。再说一次, 我们找到的此类问题是较少的。

图 9-3 给出了所有判定问题的集合。注意, 在这个图中,  $P$  包含于  $NP$  中, 是它的一个真子集。但是, 实际情况可能并非如此。也就是说, 还从来没有人能够证明  $NP$  中有不属于  $P$  的问题。因此,  $NP-P$  可能为空。事实上, 关于  $P$  是否等于  $NP$  的问题, 是计算机科学中最能引起兴趣、最为重要的问题之一。这个问题之所以重要, 正如前面讨论过的一样, 是因为我们已经给出的大多数判定问题都属于  $NP$ 。因此, 如果  $P=NP$ , 那大多数已知判定问题就找到多项式时间算法了。

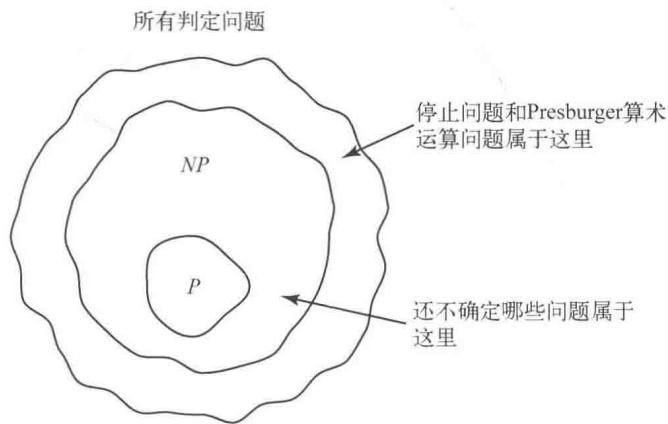


图 9-3 所有判定问题的集合

要证明  $P \neq NP$ , 必然在  $NP$  中找出一个不属于  $P$  的问题, 而要证明  $P=NP$ , 就必须为  $NP$  中的每个问题都找出一个多项式时间算法。接下来将会看到, 后一任务可以大幅简化。也就是说, 我们将会证明, 只需要为一

大类问题中的一个问题找出一个多项式时间算法。尽管有这样大幅的一个简化，但许多研究人员仍然怀疑  $P$  等于  $NP$ 。

### 9.4.2 NP完全问题

例 9.2 至例 9.5 中的问题看似难度各异。例如，关于旅行推销员问题的动态规划算法（算法 3.11）在最差情况下为  $\Theta(n^2 2^n)$ 。而 0-1 背包问题的动态规划算法（在 4.4 节）在最差情况下则为  $\Theta(2^n)$ 。此外，旅行推销员问题的分支定界算法（算法 6.3）的状态空间树有  $(n-1)!$  个叶子，而 0-1 背包问题的分支定界算法（算法 6.2）的状态空间树则只有大约  $2^{n+1}$  个节点。最后，0-1 背包问题的动态规划算法为  $\Theta(nW)$ ，也就是说，只要背包的容量  $W$  不是非常非常大，那它的效率还是很高的。了解到这一点，似乎 0-1 背包问题在本质上可能就要比旅行推销员问题更容易一些。我们将会证明，尽管有上述结论，但这两个问题、例 9.2 至例 9.5 中的其他问题，以及数以千计的其他问题在某种意义上都是等价的，也就是说，如果其中任何一个属于  $P$ ，那所有这些问题都必然属于  $P$ 。所有这些问题都被称为是  $NP$  完全的。为推导出这一结果，首先描述  $NP$  完全性理论的一个基本问题——CNF 可满足性问题。

#### 例 9.6 CNF 可满足性问题

逻辑（布尔）变量（logical/Boolean variable）可以取两个值之一：真或假。如果  $x$  是一个变量， $\bar{x}$  是  $x$  的“非”。也就是说，当且仅当  $\bar{x}$  为假时， $x$  为真。文字（literal）是一个逻辑变量或一个逻辑变量的非。子句（clause）是一个由逻辑或（or）运算符（ $\vee$ ）分隔的文字序列。合取范式（CNF, Conjunctive Normal Form）是一个用逻辑与（and）运算符（ $\wedge$ ）分隔的子句序列。下面是 CNF 中一个逻辑表达式的例子：

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee x_3 \vee x_4)$$

CNF 可满足性判定问题（CNF-Satisfiability Decision Problem）就是对于一个给定的 CNF 逻辑表达式，判断是否存在某种真值赋值（将真和假赋值给变量），使该表达式为真。

#### 例 9.7 对于实例

$$(x_1 \vee x_2) \wedge (x_2 \vee \bar{x}_3) \wedge \bar{x}_2$$

CNF 可满足性的答案为“是”，这是因为为  $x_1$  赋值“真”，为  $x_2$  赋值“假”，为  $x_3$  赋值“假”，即可使此表达式为真。对于实例

$$(x_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2$$

CNF 可满足性的答案为“否”，因为没有任何真值赋值方式可以使此表达式为真。

很容易就能编写一个多项式时间算法，它以一个 CNF 逻辑表达式和对变量的一组真值赋值为输入，然后验证此表达式对于此赋值是否为真。因此，这个问题属于  $NP$  集合。还没有人为这一问题找到多项式时间算法，也没有人证明不能在多项式时间内解决它。所以我们不知道它是否属于  $P$ 。关于这一问题，最值得注意的事情发生在 1971 年，Stephen Cook 发表了一篇论文，证明了：如果 CNF 可满足性问题属于  $P$ ，则  $P=NP$ 。（L. A. Levin 在 1973 年独立发表了这一定理的一个变化形式。）为了能够严格地表述这一定理，需要先给出一个新概念——多项式时间可约简（polynomial-time reducibility）。

假定我们希望解决判定问题  $A$ ，而且已经有了一个用于解决判定问题  $B$  的算法。再假定我们可以编写一个算法，由问题  $A$  的每个实例  $x$  都生成问题  $B$  的一个相应实例  $y$ ，使得当且仅当问题  $A$  对实例  $x$  的回答为“是”时，问题  $B$  的一种算法也对实例  $y$  回答“是”。这样一种由  $x$  生成  $y$  的算法称为变换算法（transformation algorithm），实际上就是一个函数，将问题  $A$  的每个实例映射为问题  $B$  的一个实例。可以将其表示如下：

$$y=\text{tran}(x)$$

这个变换算法与问题  $B$  的算法相结合，可以得出算法  $A$  的一个算法。图 9-4 展示了这一点。

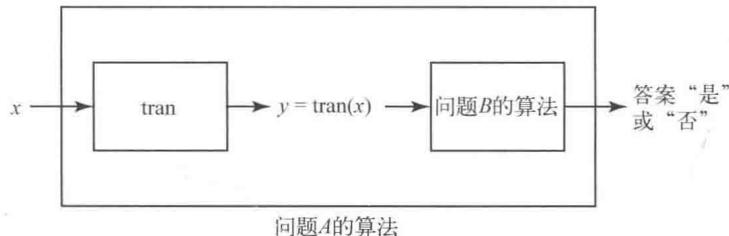


图 9.4 算法  $\text{tran}$  是一种变换算法，它将判定问题  $A$  的每个实例  $x$  映射为判定问题  $B$  的一个实例  $y$ 。它与判定问题  $B$  的算法一起，为判定问题  $A$  生成一个算法

下面的例子与  $NP$  完全性理论没有什么关系。给出这个例子只是因为它是变换算法的一个简单例子。

#### 例 9.8 一个变换算法

假设第一个判定问题是：给定  $n$  个逻辑变量，它们中是否至少有一个变量的取值为 “true”？假设第二判定问题是：给定  $n$  个逻辑变量，它们的最大者是否为正数？设我们的变换函数为：

$$k_1, k_2, \dots, k_n = \text{tran}(x_1, x_2, \dots, x_n)$$

其中， $x_i$  为真时  $k_i$  为 1； $x_i$  为假时， $k_i$  为 0。当且仅当至少一个  $k_i$  为 1 时，第二个问题的算法返回 “yes”，而当且仅当至少一个  $x_i$  为真时至少一个  $k_i$  为 1。因此，当且仅当至少一个  $x_i$  为真时，第二个问题的算法返回 “yes”，这就意味着我们的变换是成功的，可以使用第二个问题的算法来解决第一个问题。

关于刚刚介绍的概念，有以下定义。

**定义** 如果存在一个由判决问题  $A$  到判决问题  $B$  的多项式时间变换算法，那算法  $A$  就是算法  $B$  的多项式时间多对一约简 (polynomial-time many-one reducible，通常就说是问题  $A$  约简为问题  $B$ )。用符号表示就是：

$$A \propto B$$

我们说“多对一”，是因为变换算法是一个函数，可以将问题  $A$  的多个实例映射到问题  $B$  的一个实例。也就是说，它是多对一函数。

如果这个变换算法是多项式时间的，而且问题  $B$  有一个多项式时间算法，那凭直觉判断，将这个变换算法与问题  $B$  的算法结合在一起时为问题  $A$  生成的算法，必然也是多项式时间算法。例如，例 9.8 中的变换算法显然是多项式时间的。因此，如果运行该算法之后再运行第二个问题的某个多项式时间算法，那就可以在多项式时间内解决第一个问题。以下定理证明了确实如此。

**定理 9.1** 如果判定问题  $B$  属于  $P$ ，而且  $A \propto B$ ，那判定问题  $A$  就属于  $P$ 。

**证明：**假设有一个由算法  $A$  到算法  $B$  的多项式时间变换算法， $p$  是该算法时间复杂度的界限。再设  $q$  是一个多项式，它是  $B$  的一个多项式算法的时间复杂度的界限。假定有问题  $A$  的一个实例，其规模为  $n$ 。因为该变换算法中最多有  $p(n)$  个步骤，在最差情况下，该算法会在每个步骤输出一个符号，所以由该变换算法生成的问题  $B$  的实例，其规模最大为  $p(n)$ 。当该实例是问题  $B$  算法的输入时，这意味着最多有  $q[p(n)]$  个步骤。因此，将问题  $A$  的实例转换为问题  $B$  的一个实例，然后再求解  $B$ ，以获得问题  $B$  的正确答案，所需要的工作量最多为：

$$p(n) + q[p(n)]$$

它是  $n$  的一个多项式。

下面定义  $NP$  完全。

**定义** 如果以下两个条件均为真，则说问题  $B$  是 NP 完全的。

(1)  $B$  属于  $NP$ 。

(2) 对于  $NP$  中的每个其他问题  $A$ ，有

$$A \propto B$$

根据定理 9.1，如果可以证明任何  $NP$  完全问题属于  $P$ ，那就可以得出  $P=NP$  的结论。1971 年，Stephen Cook 设法找到一个  $NP$  完全的问题。下面的理论表述了这一结果。

**定理 9.2** (Cook 的定理) CNF 可满足性问题是  $NP$  完全的。

**证明：**此证明可在 Cook (1971 年) 和 Garey、Johnson (1979 年) 的文献中找到。

尽管这里没有证明 Cook 的定理，但我们要说一下，其证明并不是将  $NP$  中的每个问题都分别约简为 CNF 可满足性。如果真是这样做的，只要发现一个属于  $NP$  的新问题，就得将其约简添加到证明过程中。实际上，其证明过程利用了  $NP$  中各个问题的共有性质，证明这个集合中的任何问题都必然可以约简为 CNF 可满足性问题。

一旦证明了这一奠基性定理，许多其他问题都被证明是  $NP$  完全的。这些证明依赖于以下定理。

**定理 9.3** 如果以下两个条件同时为真，则问题  $C$  是  $NP$  完全的。

(1)  $C$  属于  $NP$ 。

(2) 对于其他某个  $NP$  完全问题  $B$ ， $B \propto C$ 。

**证明：**因为  $N$  是  $NP$  完全的，所以对于  $NP$  中的任意问题  $A$ ，有  $A \propto B$ 。不难看出这一约简是可传递的。于是， $A \propto C$ 。因为  $C$  属于  $NP$ ，所以可以得出结论： $C$  是  $NP$  完全的。

根据 Cook 的定理和定理 9.3，要证明一个问题  $B$  是  $NP$  完全的，可以通过证明它属于  $NP$ ，并且 CNF 可满足性问题可以约简为该问题。这些约简通常要比例 9.7 中给出的约简复杂得多。现在给出这样一个约简。

**例 9.9** 证明分团判定问题是  $NP$  完全的。为这个问题编写一个多项式验证算法就可以证明它属于  $NP$ ，将这一工作留作练习。因此，现在只需要证明

CNF 可满足性问题  $\propto$  分团判定问题

就能得出结论：该问题为  $NP$  完全的。首先回想一下，一个无向图中的分团就是一个顶点的子集，使该子集中的每个顶点与该子集中的所有其他顶点都是相邻的，分团判定问题就是对于一个图和一个正整数  $k$ ，判断该图中是否存在一个至少包含  $k$  个顶点的分团。设

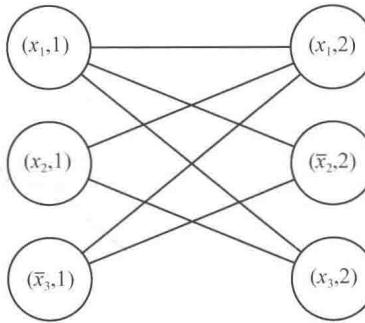
$$B = C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

是一个 CNF 逻辑表达式，其中每个  $C_i$  是  $B$  的一个子句，并设  $x_1, x_2, \dots, x_n$  是  $B$  中的变量。如下所示，将  $B$  转换为一个图  $G=(V, E)$ ：

$$V = \{(y, i), y \text{ 是子句 } C_i \text{ 中的一个文字}\}$$

$$E = \{((y, i), (z, j)), i \neq j \text{ 且 } \bar{z} \neq y\}$$

图 9-5 中给出了  $G$  的一个示例构造。这一转换是多项式时间的，其证明留作练习。因此，我们只需要证明当且仅当  $G$  中有一个大小至少为  $k$  的分团时， $B$  是 CNF 可满足的。接下来就来证明。

图 9-5 当  $B=(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$  时，例 9.9 中的图  $G$ 

(1) 证明：如果  $B$  是 CNF 可满足的， $C$  中就有一个大小至少为  $k$  的分团。

如果  $B$  是 CNF 可满足的，那就存在  $x_1, x_2, \dots, x_n$  的真值赋值，使得每个子句在此赋值下都为真。这就意味着，采用这些赋值时，每个  $C_i$  中至少有一个文字为真。从每个  $C_i$  中选择这样一个文字。然后令

$$V' = \{(y, i) | y \text{ 是选自 } C_i \text{ 且为真的文字}\}$$

显然， $V'$  构成了  $G$  中大小为  $k$  的一个分团。

(2) 证明：如果  $G$  有一个大小至少为  $k$  的分团，则  $B$  是 CNF 可满足的。

因为不可能存在一条从顶点  $(y, i)$  到顶点  $(z, i)$  的边，所以一个分团中所有顶点的索引必然都是互不相同的。因为只有  $k$  个不同顶点，所以这意味着一个分团最多可以有  $k$  个顶点。因此，如果  $G$  中有一个大小至少为  $k$  的分团  $(V', E')$ ，则  $V'$  中的顶点数必然恰好为  $k$ 。因此，如果设

$$S = \{y | (y, i) \in V'\}$$

则  $S$  中包含  $k$  个文字。此外， $S$  包含来自  $k$  个子句中每个子句的一个文字，这是因为，对于任意文字  $y$  和  $z$ ，以及索引  $i$ ，都不存在连接  $(y, i)$  和  $(z, i)$  的边。最后， $S$  不能同时包含一个文字  $y$  和它的补  $\bar{y}$ ，这是因为对于任意  $i$  和  $j$ ，不存在连接  $(y, i)$  和  $(\bar{y}, j)$  的边。因此，如果设

$$x_i = \begin{cases} \text{true} & (\text{若 } x_i \in S) \\ \text{false} & (\text{若 } \bar{x}_i \in S) \end{cases}$$

并且为不在  $S$  中的变量指定任意真值，则  $B$  中的所有子句均为真。因此， $B$  是 CNF 可满足的。

回忆 5.6 节的哈密顿回路问题。哈密顿回路判定问题是判断一个连通无向图是否至少有一个旅程（也就是一条路径，始于一个顶点，将图中的每个顶点访问一次，最后终止于起始顶点）。有可能证明：

#### CNF 可满足性问题 $\propto$ 哈密顿回路判定问题

这一约简甚至比前面例子中给出的还要乏味，可以在 Horowitz 和 Sahni (1978 年) 的文献中找到。在习题中会要求你为这一问题编写一个多项式时间验证算法。因此，可以得出结论：哈密顿回路判定问题是 NP 完全的。

既然已经知道分团判定问题和哈密顿回路判定问题是 NP 完全的，要证明 NP 中的其他某个问题是 NP 完全的，只需要证明分团判定问题或者哈密顿回路判定问题可以约简为该问题即可（根据定理 9.3）。也就是说，在证明各个问题的 NP 完全性时，不需要再重新使用 CNF 可满足性问题。下面给出更多的示例约简。

**例 9.10** 考虑旅行推销员判定问题的一种变化形式，其中的图是没有方向的。也就是说，给定一个加权无向图和一个正整数  $d$ ，判断是否存在一个总权重不大于  $d$  的无向旅程。显然，前面对于普通旅行推销员问题给出的多项式时间验证算法对于这个问题也是有效的。因此，该问题属于 NP，只需要证明某个 NP 完全问题可以化简为它，就能得出它是 NP 完全的结论了。我们将证明：

#### 哈密顿回路判定问题 $\propto$ 旅行推销员（无向）判定问题

将哈密顿回路判定问题的一个实例  $(V, E)$  变换为旅行推销员（无向）判定问题的一个实例  $(V, E')$ ，该实例具有相同的顶点集合  $V$ ，在每对顶点之间有一条边，且具有以下权重：

$$(u, v) \text{ 的权重等于 } \begin{cases} 1 & (\text{若 } (u, v) \in E) \\ 2 & (\text{若 } (u, v) \notin E) \end{cases}$$

图 9-6 中给出这一变换的一个例子。显然，当且仅当  $(V, E')$  拥有一个总权重不大于  $n$  的旅程时， $(V, E)$  拥有一个哈密顿回路，其中  $n$  是  $V$  中的顶点数。要完成此示例，需要证明此变换是多项式时间的，这一工作留作习题。

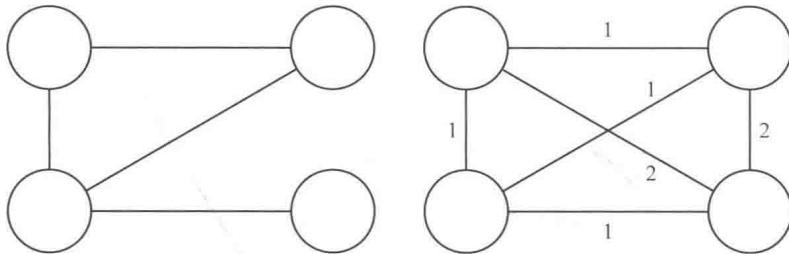


图 9-6 例 9.10 中的变换算法将左边的无向图映射为右边的加权有向图

**例 9.11** 我们已经为普通的旅行推销员判定问题编写了一个多项式时间验证算法。因此，这个问题是属于 NP 的，通过证明

旅行推销员（无向）判定问题  $\propto$  旅行推销员判定问题

就可以证明这个问题是 NP 完全的。

将旅行推销员（无向）判定问题的一个实例  $(V, E)$  变换为旅行推销员问题的实例  $(V, E')$ ，它具有相同的顶点集合  $V$ ，只要  $(u, v) \in E$ ， $E'$  中还拥有边  $\langle u, v \rangle$  和  $\langle v, u \rangle$ 。 $\langle u, v \rangle$  和  $\langle v, u \rangle$  的有向权重都与  $(u, v)$  的无向权重相同。显然，当且仅当  $(V, E')$  中有一个总权重不大于  $d$  的有向旅程时， $(V, E)$  中有一个总权重不大于  $d$  的无向旅程。证明该变换是多项式时间的，即可完成这一示例，此工作留作习题。

前面曾经提到，已经使用类似于前面刚刚给出的约简，证明了包含例 9.2 至例 9.5 在内的数以千计的问题都是 NP 完全的。Garey 和 Johnson (1979 年) 的文献中包含了许多示例约简，并列出了 300 多个 NP 完全问题。

### NP 的状态

图 9-3 将  $P$  显示为 NP 的一个真子集，但前面曾经提到，它们也可能是同一集合。NP 完全问题的集合应当怎样放到这个图中呢？首先，根据定义，它是 NP 的一个子集。因此，Presburger 算术运算、停机问题和其他不属于 NP 的判定问题都不是 NP 完全的。

属于 NP 但不是 NP 完全的判定问题明显是对于所有实例都回答 “yes”（或者是对所有实例都回答 “no”）的平凡判定问题。这个问题不是 NP 完全的，因为不可能将一个非平凡判定问题转换为它。

如果  $P=NP$ ，其情景将如图 9-7 中的左图所示。如果  $P \neq NP$ ，其情景将如其中的右图所示。也就是说，如果  $P \neq NP$ ，则

$$P \cap NP \text{ 完全} = \emptyset$$

其中 NP 完全表示所有 NP 完全问题组成的集合。该式之所以成立，是因为如果  $P$  中的某一问题是 NP 完全的，则由定理 9.1 可知，我们能够在多项式时间内解决 NP 中的任意问题。

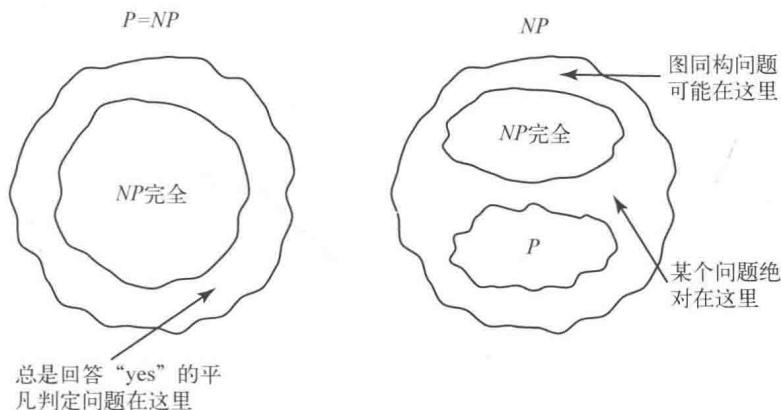


图 9-7 集合 NP 或者如左图所示，或者如右图所示

注意，图 9-7（左）表明“图同构”问题可能属于：

$$NP - (P \cup NP \text{ 完全})$$

这个问题涉及图的同构，其定义如下。有两个图（无方向的或有方向的） $G=(V, E)$ 和 $G'=(V', E')$ ，如果存在一个从 $V$ 到 $V'$ 的一对一函数 $f$ ，使得对于 $V$ 中的每个 $v_1$ 和 $v_2$ ，当且仅当边 $(f(u), f(v))$ 属于 $E'$ 时，边 $(u, v)$ 属于 $E$ ，则说这两个图是同构的（isomorphic）。图同构问题如下。

#### 例 9.12 图同构问题

给定两个图  $G=(V, E)$  和  $G'=(V', E')$ ，它们是否同构？

图 9-8 中的有向图是同构的，因为存在如下函数：

$$u_5=f(v_1) \quad u_3=f(v_2) \quad u_1=f(v_3) \quad u_4=f(v_4) \quad u_2=f(v_5)$$

以下任务留作习题：为图同构问题编写一个多项式时间验证算法，从而证明它属于 NP。这个问题的一种直接算法是检查所有  $n!$  个一对一映射，其中  $n$  是顶点的个数。该算法的时间复杂度要差于指数复杂度。还没有人为这一问题找到一种多项式时间算法，但也没有人证明它是 NP 完全的。因此，我们不知道它是否属于  $P$ ，也不知道它是不是 NP 完全的。

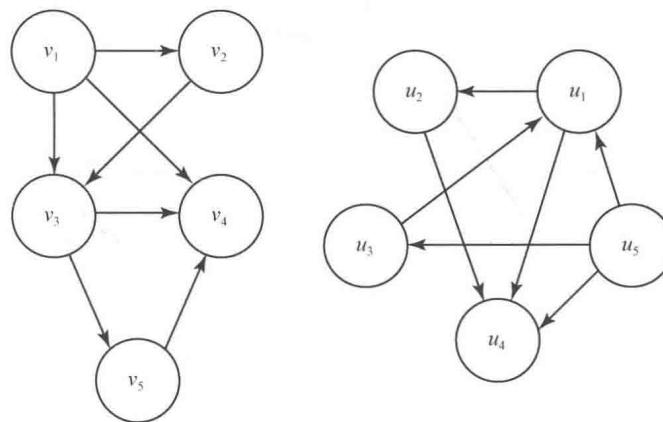


图 9-8 这些图是同构的

还没有人能够证明，NP 中存在既不属于  $P$  也不是 NP 完全的问题（这样一个证明将会自动证明  $P \neq NP$ ）。但是，人们已经证明：如果  $P \neq NP$ ，则必然存在这样一个问题。这一结果在图 9-7 的右图中表示，其正式表述就是以下定理。

**定理 9.4** 如果  $P \neq NP$ ，则集合

$NP - (P \cup NP\text{ 完全})$

不为空。

证明：在 Ladner（1975 年）的文献中可以找到一个更一般的结果，由此可以证明此定理。

#### 互补问题

注意以下两个问题之间的相似性。

#### 例 9.13 质数问题

给定一个正整数  $n$ ,  $n$  是否为质数？

#### 例 9.14 合数问题

给定一个正整数  $n$ , 是否存在整数  $m > 1$  和  $k > 1$ , 使得  $n = mk$ ？

质数问题就是用 9.2 节开头的算法解决的那个问题。它是合数问题的互补问题。一般来说，一个判定问题的互补问题（complementary problem）就是在原问题回答“no”时，它就回答“yes”；在原问题回答“yes”时，它回答“no”。下面是互补问题的另一个例子。

#### 例 9.15 互补旅行推销员判定问题

给定一个加权图和一个正整数  $d$ , 是否不存在总权重不大于  $d$  的旅程？

显然，如果为一个问题找到了一个普通的确定性多项式时间算法，那它的互补问题就有一个确定性的多项式时间算法。例如，如果可以在多项式时间内判定一个数字是否为合数，那也就判定了它是否为质数。但是，为一个问题找到一个多项式时间的非确定性算法，并不一定自动为其互补问题生成一种多项式的非确定性问题。也就是说，证明了一个问题属于  $NP$  后，并不能自动证明其互补问题也在  $NP$  中。至于互补旅行推销员问题，该算法必须能在多项式时间内验证不存在权重不大于  $d$  的旅程。还从来没有人为互补旅行推销员判定问题找到一个多项式时间验证算法。事实上，从来没有人能证明任何已知  $NP$  完全问题的互补问题是属于  $NP$  的。另一方面，也没有人已经证明某个问题属于  $NP$ ，而其互补问题不属于  $NP$ 。但已经有人得到了以下结果。

定理 9.5 如果任意  $NP$  完全问题的互补问题属于  $NP$ ，则  $NP$  中每个问题的互补问题都属于  $NP$ 。

证明：此证明可以在 Garey 和 Johnson（1979 年）的文献中找到。

现在让我们更深入地讨论图同构问题和质数问题。上一小节已经提到，还没有人为图同构问题找到多项式时间算法，也还没有人能证明它是  $NP$  完全的。一直以来，质数问题也是如此。但 2002 年，Agrawal 等人为质数问题设计了一种多项式时间算法。10.6 节将给出这一算法。在此之前，Pratt 已经在 1975 年证明了该质数问题属于  $NP$ ，而且其互补问题（合数问题）属于  $NP$  的证明要更简单一些。同样，线性规划问题及其互补问题也都被证明属于  $NP$ ，之后，Chachian（1979 年）为它设计了一种多项式时间算法。另一方面，还没人能证明图同构问题的互补问题属于  $NP$ 。根据这些结果，证明图“同构问题是  $NP$  完全”的可能性似乎要高于为其找出一种多项式算法的可能性。另一方面，它也可能属于集合  $NP - (P \cup NP\text{ 完全})$ 。

### 9.4.3 $NP$ 困难、 $NP$ 容易和 $NP$ 等价问题

到目前为止，我们只讨论了判定问题。接下来将结果推广到一般问题。回想一下，由定理 9.1 可知，如果判定问题  $A$  可以在多项式时间内多对一约简为问题  $B$ ，则就可以使用问题  $B$  的一个多项式时间算法，在多项式时间中解决问题  $A$ 。我们将这一概念推广到具有以下定义的非判定问题。

**定义** 如果利用问题  $B$  的一个假设多项式时间算法，可以在多项式时间内解决问题  $A$ ，那就说问题  $A$  可以在多项式时间内图灵约简（polynomial-time Turing reducible）为问题  $B$ （通常，我们就说  $A$  图灵约简为  $B$ ）。可以用符号写为：

$$A \leq_T B$$

这个定义并不要求问题  $B$  存在一种多项式算法。它只是说，如果的确存在这样一个算法，那问题  $A$  也可以在多项式时间中解决。显然，如果  $A$  和  $B$  都是判定问题，那

$A \propto B$  就可导出  $A \propto_T B$

接下来将  $NP$  完全的概念推广到非判定问题。

**定义** 如果对于某个  $NP$  完全问题  $A$ , 有  $A \propto_T B$ , 就说问题  $B$  是  $NP$  困难 ( $NP\text{-hard}$ ) 的。

不难看出, 图灵约简是可传递的。因此,  $NP$  中的所有问题可约简为任意  $NP$  困难问题。这就是说, 如果对于任意  $NP$  困难问题存在一个多项式时间算法, 则  $P=NP$ 。

哪些问题是  $NP$  困难的呢? 显然, 每个  $NP$  完全问题是  $NP$  困难的。因此, 我们转而询问, 哪些非判定问题是  $NP$  困难的。之前我们曾经提到, 如果可以为一个最优化问题找到一种多项式时间算法, 那就自动为相应的判定问题找到了多项式时间算法。因此, 与任何  $NP$  完全问题相对应的最优化问题都是  $NP$  困难的。下面的例子正式应用图灵约简的定义, 证明了旅行推销员问题的这一结果。

#### 例 9.16 旅行推销员最优化问题是 $NP$ 困难的

假定旅行推销员最优化问题有一个假设的多项式时间算法。假设给出了旅行推销员问题的一个实例, 其中包含了图  $G$  和正整数  $d$ 。将此假设算法应用到图  $G$ , 获得最优解  $\text{midist}$ 。如果  $d \leq \text{midist}$ , 则对此判定问题实例的回答为 “no”, 否则为 “yes”。显然, 最优化问题的假设多项式时间算法, 再加上这一附加步骤, 就可以在多项式时间内给定此判定问题的答案。因此,

旅行推销员判定问题  $\propto_T$  旅行推销员最优化问题

哪些问题不是  $NP$  困难的呢? 我们不知道是否存在这样的问题。事实上, 如果要证明某个问题不是  $NP$  困难的, 那就是在证明  $P \neq NP$ 。原因是, 如果  $P=NP$ , 那  $NP$  中的每个问题都可以由一种多项式时间算法加以解决。因此, 利用任意问题  $B$  的一个假设多项式时间算法, 可以解决  $NP$  中的每个问题, 只需要为每个问题调用该算法即可。我们甚至不需要问题  $B$  的这个假设算法。因此, 所有问题都会是  $NP$  困难的。

另一方面, 已经为其找到多项式时间算法的问题可能不是  $NP$  困难的。事实上, 如果要证明某个已经为其找到了多项式时间算法的问题是  $NP$  困难的, 那就是在证明  $P=NP$ 。原因是, 这样的话, 我们将拥有某个  $NP$  困难问题的实际多项式时间算法, 而不再是假设的。因此, 利用由该问题到该  $NP$  困难问题的图灵约简, 就可以在多项式时间内解决  $NP$  中的每个问题。

图 9-9 说明  $NP$  困难问题的集合如何放入所有问题组成的集合中。

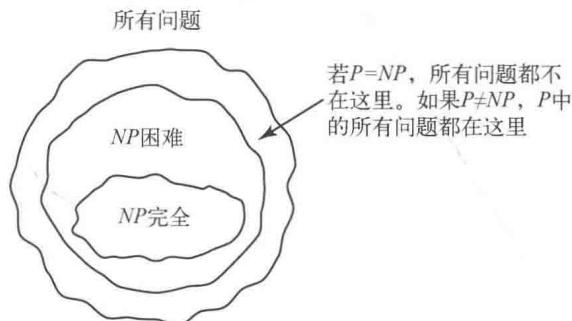


图 9-9 所有问题组成的集合

如果一个问题  $A$  是  $NP$  困难的, 那它至少和  $NP$  完全问题一样难 (是指为其找出多项式时间算法的希望而言)。例如, 旅行推销员最优化问题至少与  $NP$  完全问题一样难。但是, 反过来是否一定成立呢? 也就是说,  $NP$  完全问题是否至少与旅行推销员最优化问题一样难呢?  $NP$  困难性并不难导出这一结论。我们需要另一个定义。

**定义** 如果对于 NP 中的某个问题  $B$ , 有

$$A \propto_T B$$

就说问题  $A$  是 NP 容易 (NP-easy) 的。

显然, 如果  $P=NP$ , 则对于所有 NP 容易问题都存在一种多项式时间算法。注意, 我们对 NP 容易的定义并不完全与 NP 困难的定义对称。一个问题是 NP 容易的, 等价于它可以约简为一个 NP 完全问题, 其证明留作习题。

哪些问题是 NP 容易的? 显然,  $P$  中的问题、NP 中的问题、已经为其找到多项式时间算法的非判定问题都是 NP 容易。与 NP 完全判定问题对应的最优化问题, 通常可以证明是 NP 容易的。但是, 其证明过程并非显而易见, 如下面的例子所示。

**例 9.17** 旅行推销员最优化问题是 NP 容易的。为导出这一结果, 我们引入以下问题。

#### 旅行推销员延伸判定问题

设给定了旅行推销员判定问题的一个实例, 其中, 图中的顶点数为  $n$ , 整数为  $d$ 。此外, 假设给定一个部分旅程  $T$ , 它包括  $m$  个不同顶点。这个问题就是判断是否可以延伸  $T$ , 以得到一个总权重不大于  $d$  的完整旅程。这个问题的参数与旅行推销员判决问题的参数相同, 只是增加了一个部分旅程  $T$ 。

不难证明, 旅行推销员延伸判定问题属于 NP。因此, 要获得我们想要的结果, 只需要证明:

$$\text{旅行推销员最优化问题} \propto_T \text{旅行推销员延伸判定问题}$$

为此, 设 polyalg 是旅行推销员延伸判定问题的一个假设多项式时间算法。polyalg 的输入是一个图、一个部分旅程和一个距离。设给定旅行推销员最优化问题的一个实例  $G$ , 其规模为  $n$ 。设此实例中的顶点为:

$$\{v_1, v_2, \dots, v_n\}$$

并设

$$\begin{aligned} d_{\min} &= n \\ d_{\max} &= n \times \text{maximum (从 } v_i \text{ 到 } v_j \text{ 的边上的权重)} \quad (1 \leq i, j \leq n) \end{aligned}$$

如果 mindist 是一条最优旅程上各边的总权重, 则

$$d_{\min} \leq \text{mindist} \leq d_{\max}$$

因为任何顶点都可能是旅程的第一个顶点, 所以可以使  $v_1$  作为第一顶点。考虑以下调用:

$$\text{polyalg}(G, [v_1], d)$$

作为这一调用输入的部分旅程就是  $[v_1]$ , 它是离开  $v_1$  之前的部分旅程。对于这一调用, 可以返回 “true”的最小  $d$  值为  $d=d_{\min}$ , 如果存在一条旅程, 那若  $d=d_{\max}$ , 该调用一定会返回 “true”。如果当  $d=d_{\max}$  时返回 “false”, 那  $\text{mindist}=\infty$ 。否则, 使用一种二分查找, 就可以找出  $d$  的一个最小值, 使得在  $G$  和  $[v_1]$  为输入时, polyalg 会为其返回 “true”。这个值就是 mindist。这意味着, 最多只需要大约调用  $\lg(d_{\max})$  次 polyalg, 就可以计算出 mindist, 也就是说, 可以在多项式时间内计算 mindist。

一旦知道了 mindist 的值, 就可以使用 polyalg, 在多项式时间内构造一个最优旅程, 如下所示。如果  $\text{mindist}=\infty$ , 那就没有旅程, 任务完成。否则, 如果一个部分旅程可以延伸为一条总权重等于 mindist 的旅程, 那就说这条部分旅程是可延伸的 (extendible)。显然,  $[v_1]$  是可延伸的。因为  $[v_1]$  是可延伸的, 那必然至少存在一个  $v_i$ , 使得  $[v_1, v_i]$  是可延伸的。将 polyalg 最多调用  $n-2$  次, 就可以确定这样一个  $v_i$ , 如下所示:

$$\text{polyalg}(G, [v_1, v_i], \text{mindist})$$

其中  $2 \leq i \leq n-1$ 。在找到一个可延伸旅程时, 或者当  $i$  达到  $n-1$  时, 就停下来。我们不需要检查最后一个顶点, 因为, 如果其他顶点都失败了, 那最后一个顶点必然是可延伸的。

一般来说, 给定一个包含  $m$  个顶点的可延伸部分旅程, 最多只需要将 polyalg 调用  $n-m-1$  次, 就可

以找到一个可延伸的部分旅程。那么，最多只需要对 polyalg 调用以下次数，就可以构建一个最优旅程：

$$(n-2) + \dots + (n-m-1) + \dots + 1 = \frac{(n-2)(n-1)}{2}$$

这就是说，我们也可以在多项式时间内构造一个最优旅程，而且有一个多项式时间图灵约简。

针对例 9.2 至例 9.5 中的其他问题，以及与大多数 NP 完全判断问题相对应的最优化问题，已经给出了类似证明。关于此类问题，有以下定义。

**定义** 如果一个问题既是 NP 困难的，又是 NP 容易的，那就说它是 NP 等价的 (NP-equivalent)。

显然，当且仅当所有 NP 等价问题都存在多项式时间算法时， $P=NP$ 。

可以看到，最初将我们的理论仅局限于判定问题，并没有损失一般性，因为通常可以证明，与 NP 完全判定问题相对应的最优化问题是 NP 等价的。这就意味着，为最优化问题找出一个多项式时间算法，就等价于为判定问题找出这样一个算法。

我们的目标是为 NP 理论提供一种浅显的介绍。如需更详尽的介绍，请参考我们已经多次引用的教科书，即 Garey 和 Johnson (1979 年) 的文献。尽管这本教科书非常老了，但它仍然是对 NP 理论进行了全面介绍的佳作之一。另外一本很好的介绍性教科书是 Papadimitriou (1994 年) 的文献。

## 9.5 处理 NP 困难问题

对于已知为 NP 困难的问题没有多项式时间算法，那我们又能做些什么来解决此类问题呢？第 5 章和第 6 章给出了一种方式。这些问题的回溯算法和分而治之算法在最差情况下都是非多项式时间的。但是，它们对于许多大型实例经常是非常高效的。因此，对于某个真正关心的大实例，回溯算法或分而治之算法可能已经足够了。回想 5.3 节曾经说过，可以使用蒙特卡洛方法估计一种给定算法对于一个特定实例是不是高效的。如果估计结果显示它可能是高效的，那就可以尝试使用该算法来解决该实例。

另一种方法是为一个 NP 困难问题的一个实例子类找出一种高效算法。例如，6.3 节讨论了贝叶斯网络中的概率推断 (probabilistic inference) 问题，它是 NP 困难的。一般情况下，贝叶斯网络由一个有向无环图和一个概率分布组成。在一个实例子类中，图是单连通的，人们已经为这个子类中的实例找到了多项式时间算法。对于一个有向无环图，如果从任意顶点到任意其他顶点的路径不超过一条，就说该图是单连通的 (single connected)。Pearl (1988 年) 和 Neapolitan (1990 年, 2003 年) 的文献讨论了这些算法。

这里要研究的第三种方法是开发近似算法。一个 NP 困难最优化问题的近似算法 (approximation algorithm) 是一个不能保证给定最优解的算法，但它们生成的解是相当接近最优的。通常可以获得一个界限，保证一个解与最优解的接近程度。例如，我们推导出一种近似算法，它为旅行推销员最优化问题的一个变型给出了一个解，我们称之为 minapprox。我们证明了

$$\text{minapprox} \leq 2 \times \text{mindist}$$

其中 mindist 是最优解。这并不意味着 minapprox 总是 mindist 的差不多两倍。对于许多实例来说，它可能更接近于甚至等于 mindist。它真正的含义是，可以保证 minapprox 永远不会大到 mindist 的两倍。我们接下来开发这一算法，并对其进行改进。然后进一步演示近似算法：为另一个问题推导一种近似算法。

### 9.5.1 旅行推销员问题的近似算法

我们的算法针对的是该问题的以下变化形式。

### 例 9.18 带有三角形不等式的旅行推销员问题

设给定一个加权无向图  $G=(V, E)$ , 使得:

- (1) 每两个不同节点之间都存在一条边;
- (2) 如果  $W(u, v)$  表示将顶点  $u$  连接到顶点  $v$  的边上的权重, 那么, 对于其他每个顶点  $y$ , 有

$$W(u, v) \leq W(u, y) + W(y, v)$$

第二个条件称为**三角形不等式** (triangular inequality), 图 9-10 中对其进行了演示。如果该权重表示城市之间的实际距离 (直线距离), 那这一条件就可以满足。回想一下, 对于加权图来说, 权重和距离这两个术语是互换使用的。由第一个条件可知, 每个城市到其他每个城市之间都存在一条双向道路。

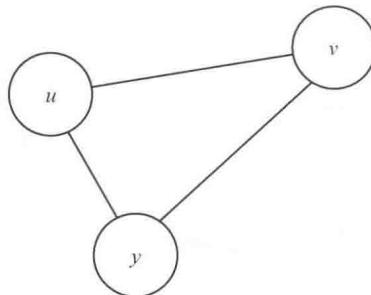


图 9-10 由三角形不等式可知, 从  $u$  到  $v$  的“距离”不大于从  $u$  到  $y$  的“距离”再加上从  $y$  到  $v$  的“距离”

问题是找出一条最短路径 (最优旅程), 它始于和终于同一顶点, 而其他每个顶点都恰好访问一次。可以证明, 该问题的这一变化形式也是  $NP$  困难的。

注意, 在该问题的这一变化形式中, 这个图是无方向的。如果从这样一幅图的最优旅程中删除任意一条边, 就会得到这幅图的一棵生成树。因此, 最小生成树的总权重必然小于最小旅程的总权重。利用算法 4.1 或算法 4.2, 可以在多项式时间中获得最小生成树。两次游走于该生成树, 就能将它转换为一条访问每个城市的路径。图 9-11 中演示了这一点。图 9-11a 中是一幅图, 图 9-11b 是这个图的最小生成树, 图 9-11c 是两次游走于该树而生成的路径。如图中所示, 所得到的路径可能会多次访问某些顶点。我们可以通过“取捷径”的方式修改该路径, 使其不再多次访问某些顶点。也就是说, 我们从某一任意顶点开始遍历该路径, 并依次访问每个未访问的顶点。如果树中有多个未访问的顶点与当前顶点相邻, 那直接访问最近的一个。如果与当前顶点相邻的所有顶点都已经被访问过了, 那就通过走捷径的方式绕过它们, 走到下一个未访问的顶点。三角形不等式保证捷径不会延长该路径。图 9-11d 给出了使用这一方法得到的旅程。在该图中, 我们从左下方的顶点开始。注意, 所得到的旅程不是最优的。但是, 如果从左上方的顶点开始, 就能得到一个最优旅程。

刚刚概括的方法可概括为以下步骤:

- (1) 确定一个最小生成树;
- (2) 通过两次游走于这棵树, 创建一条访问每个城市的路径;
- (3) 通过走捷径的方式, 创建一条从来不会多次访问任意顶点的路径 (也就是说, 一条旅程)。

一般情况下, 无论选哪个起始顶点, 使用这个方法获得的旅程都不一定是最优的。但是, 下面的定理可以保证这条旅程与最优路线的接近程度。

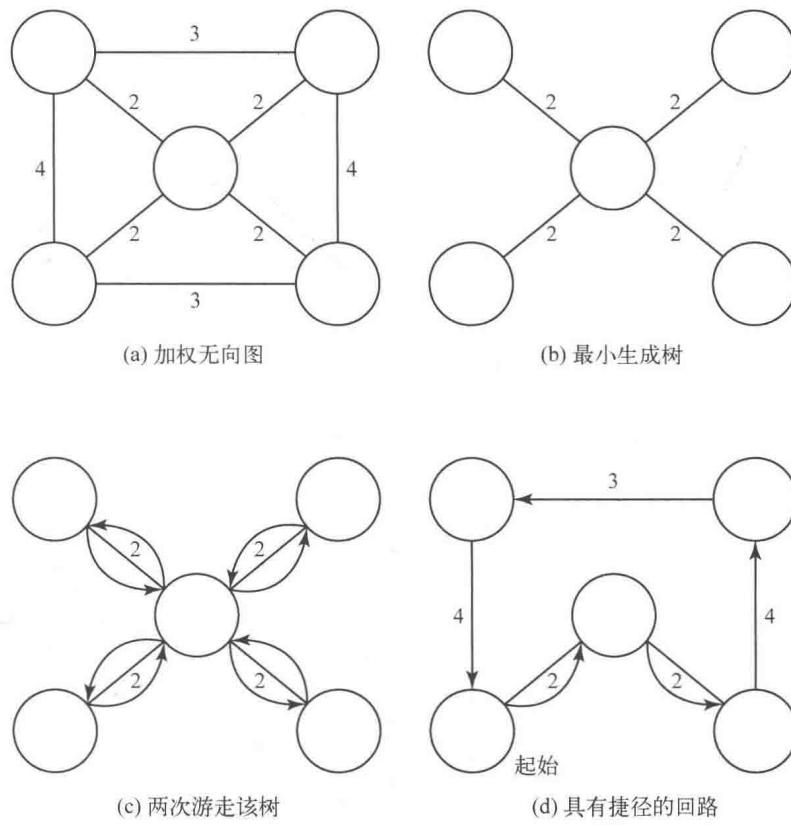


图 9-11 由最小生成树获得一个最优旅程的近似路径

**定理 9.6** 设  $\text{mindist}$  是一条最优旅程的总权重,  $\text{minapprox}$  是使用上述方法获得的旅程的总权重。则

minapprox<2×mindist

证明：前面已经讨论过，一棵最小生成树的总权重小于  $\text{mindist}$ 。因为  $\text{minapprox}$  的总权重不大于两棵最小生成树的总权重，所以该定理成立。

有可能创建一些实例来证明：可以使  $\text{minapprox}$  任意接近  $2 \times \text{mindist}$ 。因此，一般来说，定理 9.6 中获得的界限是我们能找到的最好界限。

我们甚至可以为这一问题找到一种更好的近似算法，如下所示。首先像上面一样获得一棵最小生成树。然后考虑一个顶点集合  $V'$ ，其中包含了所有接触奇数条边的顶点。不难证明，必然存在偶数个此种顶点。将  $V'$  中的顶点配对，使每个顶点都与另外一个顶点准确配对。这一创建顶点对的过程称为  $V'$  的匹配（matching）。将连接每对顶点的边添加到生成树中。因为每个顶点都有偶数条边与其接触，所以最终得到的路径会访问每个城市。此外，这条路径上的总边数不大于直接通过两次遍历该树所得路径的总边数（通常是小于）。图 9-12a 给出了一棵最小生成树，图 9-12b 给出了用一种可能匹配由该树获得的路径，图 9-12c 给出了取捷径之后获得的旅程。

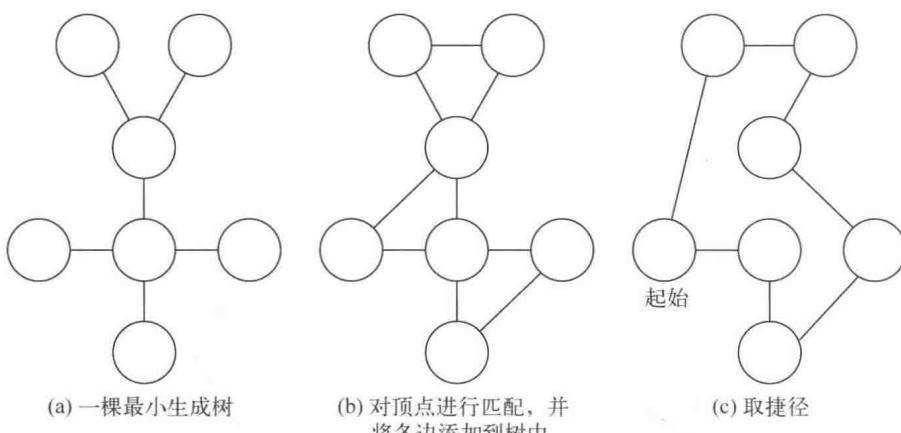


图 9-12 通过匹配获得的旅程

$V'$ 的一个最小权重匹配 (minimal weight matching) 是指由该匹配获得的边的总权重最小。Lawler (1976年) 表明了如何在多项式时间内获得最小权重。通过以下步骤，可以在多项式时间内近似解决例 9.17 中旅行推销员问题的变化形式：

- (1) 获得一棵最小生成树；
- (2) 获得  $V'$  中各顶点的一个最小权重匹配，其中  $V'$  是该生成树中接触奇数条边的顶点集合；
- (3) 将连接匹配顶点的各边添加到树中，创建一个访问所有顶点的路径；
- (4) 通过取捷径，获得一条不会对同一顶点访问两次的路径（即旅程）。

图 9-11 演示了这些步骤，而没有给出任意实际权重。以下定理表明，这一方法给出的界限要好于本节第一种方法的界限。

**定理 9.7** 设  $\text{mindist}$  是一条最优旅程的总权重， $\text{minapprox2}$  是使用最小权重匹配获得的旅程的总权重。则

$$\text{minapprox2} < 1.5 \times \text{mindist}$$

**证明** 设  $V'$  是所有接触奇数条边的顶点组成的集合。绕过旅程中不属于  $V'$  的顶点，将其转换为一条路径，使其仅连接属于  $V'$  的顶点。根据三角形不等式，这条路径的总权重可以不大于  $\text{mindist}$ 。此外，这条路径为  $V'$  中的顶点提供了两个匹配，如下所示。选择  $V'$  中的一个任意顶点，并将它与路径中的同侧顶点匹配。然后继续在同一方向上将相邻顶点配对。这是一个匹配。接下来将原顶点与路径上另一侧的顶点匹配，并继续在另一方向上将相邻顶点配对。这是第二个匹配。因为两个匹配中的边包含了路径中的所有边，所以这两个匹配的总权重之和等于该路径的权重。因此，这两个匹配中至少有一个匹配的总权重不大于路径权重的一半。这条路径的权重不大于  $\text{mindist}$ ，而且任意匹配的权重至少等于最小权重匹配的权重，所以有

$$\text{minmatch} \leq 0.5 \times \text{mindist}$$

其中  $\text{minmatch}$  是最小权重匹配的权重。回忆一下，一棵最小生成树的权重小于  $\text{mindist}$ 。因为在最小权重匹配方法第 3 步中获得的边中包含了生成树的边和由最小匹配获得的边，所以这些边的总权重小于  $1.5 \times \text{mindist}$ 。因为在第 4 步得到的最终旅程中各边的总权重不大于第 3 步所得路线的权重，所以定理得证。

有可能创建一些实例，使近似结果任意接近于  $1.5 \times \text{mindist}$ 。因此，一般情况下，这一算法的界限不会优于定理 9.7 中获得的界限。

回忆一下，我们的销售员 Nancy 一直在尝试使用旅行推销员方法的分支定界算法（算法 6.3），为其 40 个城市的销售区域找出一条最优旅程。因为这个算法在最差情况下是非多项式时间的，所以可能需要花费许多年的时间才能解决它的特定实例。如果 Nancy 的各个城市之间的距离满足“带有三角形不等式的旅行推销员问题”中的假设条件，那她最终会找到一种肯定有效的替代方法。也就是说，可以使用最小权重匹配方法，在多项式时间内获得最优旅程的一个良好近似。

## 9.5.2 装箱问题的近似算法

我们引入下面的新问题。

### 例 9.19 装箱问题

给定  $n$  件物品，其尺寸为：

$$s_1, s_2, \dots, s_n, \text{ 其中 } 0 < s_i \leq 1$$

假定我们要将这些物品装入箱中，其中每个箱子的容量为 1。问题是确定最少需要多少个箱子才能装入所有物品。

这个问题已经被证明是 NP 困难的。这一问题有一个非常简单的近似算法，称为“first fit”（首次适应）。首次适应策略是将物品放在第一个能放下它的箱子里。如果它不能放入一个箱子中，则启用一个新箱子。另一个很好的想法是按非递增顺序对物品进行装箱。因此，我们的策略称为非递增顺序首次适应。可以用以下高级算法来描述它。

```

按非递增顺序排列物品;
while (仍然有未装箱的物品){
    获取下一件物品;
    while(物品尚未装箱, 而且还有更多已启用的箱子){
        获取下一个箱子;
        if(该物品可放在此箱中)
            将它装在此箱中;
    }
    if (该物品尚未装箱){
        启用一个新箱;
        将该物品放在新箱中;
    }
}
}

```

图 9-13 给出了应用这一算法的结果。注意，它在一个贪婪算法中包含了一个贪婪算法。也就是说，我们贪婪地攫取物品，对于每件物品，我们又贪婪地攫取箱子。以下任务留作习题：为这个算法编写一个详尽版本，并证明它是  $\Theta(n^2)$  的。注意，图 9-12 中的解不是最优的。如果将大小为 0.5 的物品、大小为 0.3 的物品、一个大小为 0.2 的物品放在第二个箱子中，将两个大小为 0.4 的物品和另一个大小为 0.2 的物品放在第三个箱子里，那只需要三个箱子就可以装下这些物品。

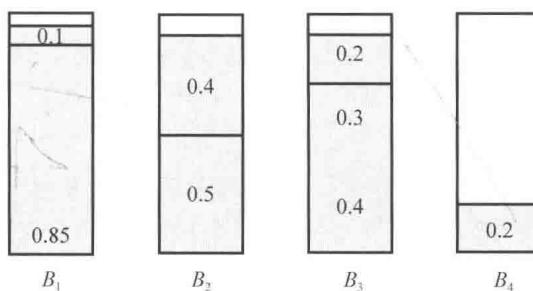


图 9-13 应用非递增顺序首次适应的结果

关于近似解可以在多大程度上接近最优解，我们接下来给出一个界限。这个界限在定理 9.8 中获得。该定理的证明需要以下两条引理。

**引理 9.1** 设  $opt$  是装箱问题某一给定实例的最优箱数。任何根据非递增顺序首次适应策略放在额外箱子（也就是说索引值大于  $opt$  的箱子）中的物品，其大小最多等于  $1/3$ 。

◆证明：设  $i$  是放在第  $opt+1$  号箱中第一件物品的索引。因为这些物品是按非递增顺序排列的，所以足以证明：

$$s_i \leq \frac{1}{3}$$

采用反证法。假设  $s_i > 1/3$ ，则

$$s_1, s_2, \dots, s_{i-1} \text{ 都大于 } \frac{1}{3}$$

这就是说，在索引不大于  $\text{opt}$  的所有这些箱子中，分别最多包含两件物品。如果这些箱子中的每一个都包含两件物品，那在一个最优解中，每个箱子中必然要有前  $i-1$  件物品中的两件。但是，因为它们的大小均大于  $\frac{1}{3}$ ，所以用其中一个箱子将无法容纳  $s_i$ 。因此，在索引不大于  $\text{opt}$  的至少一个箱子中，只包含一件物品。如果索引不大于  $\text{opt}$  的每个箱子中都仅包含一件物品，那就不会有两件物品装在同一个箱子中，第  $i$  件物品不能与其中任何一件物品装在一起（否则，我们的算法就会将它与其中之一装在一起）。但一个最优解将需要超过  $\text{opt}$  个箱子。因此，在索引不大于  $\text{opt}$  的箱子中，至少有一个包含两件物品。

我们证明，存在某个满足  $0 < j < \text{opt}$  的  $j$  值，使得前  $j$  个箱子各包含一件物品，剩下的  $\text{opt}-j$  个箱子各包含两件物品。如果不是这样，那就会存在箱子  $B_k$  和  $B_m$  ( $k < m$ )，使  $B_k$  包含两件物品， $B_m$  包含一件物品。但是，因为这些物品是按非递增顺序装箱的，所以  $B_m$  中的物品不会大于  $B_k$  中的第一件物品， $s_i$  不大于  $B_k$  中的第二件物品。因此， $B_m$  中的物品大小加上  $s_i$  的大小，其总和不大于  $B_k$  中两件物品的大小之和，也就是说， $s_i$  可以放在  $B_m$  中。因此，上面（关于  $j$  的）猜测必然为真，这些箱子将如图 9-14 所示。

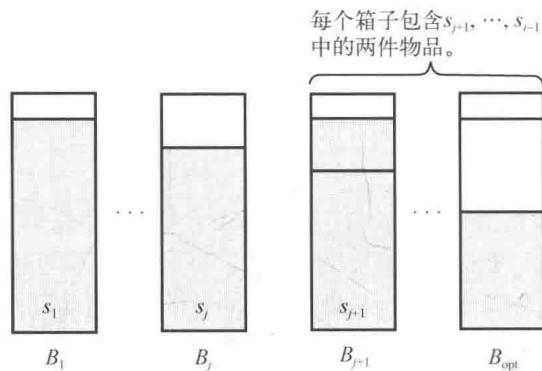


图 9-14 如果  $s_i > 1/3$ ，我们的算法会如图中所示进行装箱

假设给定一个最优解。在这样一个解中，前  $j$  件物品在  $j$  个不同的箱子中，因为如果其中任何两个可以放在一个箱子中，我们的方法肯定已经将它们放在一起了。此外，以下物品

$$s_{j+1}, s_{j+2}, \dots, s_{i-1}$$

都在剩下的  $\text{opt}-j$  个箱子中，因为它们中的任何一个都不能与前  $j$  件物品放在一起。因为我们的算法在  $\text{opt}-j$  个箱子中分别放入上述物品的两个，所以必然有  $2 \times (\text{opt}-j)$  件此种物品。因为我们假定这些物品的大小都大于  $1/3$ ，所以在剩下的  $\text{opt}-j$  个箱子中，任何一个都不可能装有其中的三件物品，这就是说，这些箱子中，每个箱子中必然恰好有两件物品。因为我们假设  $s_i$  的大小也大于  $1/3$ ，所以不能将它放在各包含两件物品的  $\text{opt}-j$  个箱子中。此外，它也不能放到包含有前  $j$  件物品的某个箱子中，因为如果它能与这些物品放在一起，那我们的算法就会已经将它与它们放在一起了。因为我们假定这是一种最优解，所以  $s_i$  必然在这些箱子之一中，这就是说，产生了矛盾，所以该引理得证。

**引理 9.2** 设  $\text{opt}$  是装箱问题某一实例的最优箱数。根据非递增顺序首次适应策略放在额外箱子中的物品数最多为  $\text{opt}-1$ 。

**证明：**因为所有物品都放在  $\text{opt}$  个箱子中，所以

$$\sum_{i=1}^n s_i \leq \text{opt}$$

采用反证法，假定我们的近似算法的确将  $\text{opt}$  件物品放在额外箱子里。设  $z_1, z_2, \dots, z_{\text{opt}}$  是这些物品的大小，对于  $1 \leq i \leq \text{opt}$ ，设  $\text{tot}_i$  是我们的算法在箱子  $B_i$  中放入的总大小。则下式必然为真：

$$\text{tot}_i + z_i \geq 1$$

这是因为，如果不是这样，那大小为  $z_i$  的物品也可以被放在  $B_i$  中。于是有

$$\sum_{i=1}^n s_i \geq \sum_{i=1}^{\text{opt}} \text{tot}_i + \sum_{i=1}^{\text{opt}} z_i = \sum_{i=1}^{\text{opt}} (\text{tot}_i + z_i) > \sum_{i=1}^{\text{opt}} 1 = \text{opt}$$

这与我们在证明开头给出的结果矛盾。这一矛盾证明了引理。

**定理 9.8** 设  $\text{opt}$  是装箱问题某一实例的最优箱数，并设  $\text{approx}$  是非递增顺序首次适应算法使用的箱数。则

$$\text{approx} \leq 1.5 \times \text{opt}$$

**证明：**根据引理 9.1 和引理 9.2，非递增顺序首次适应算法最多在额外箱中放入  $\text{opt}-1$  件物品，每个物品的大小最多为  $1/3$ 。因此，额外的箱数最多为：

$$\left\lceil (\text{opt}-1) \times \frac{1}{3} \right\rceil = \left\lceil \frac{\text{opt}-1}{3} \right\rceil = \frac{\text{opt}-1+k}{3}$$

其中  $k=0, 1$  或  $2$ 。取  $k$  的最大可能值，可以得出结论：额外箱数小于或等于  $(\text{opt}+1)/2$ 。这意味着：

$$\text{approx} \leq \text{opt} + \frac{\text{opt}+1}{3}$$

且因此，

$$\frac{\text{approx}}{\text{opt}} \leq \frac{\text{opt} + (\text{opt}+1)/3}{\text{opt}} = \frac{4}{3} + \frac{1}{3\text{opt}}$$

当  $\text{opt}=1$  时，这一比值取最大值。但是，当  $\text{opt}=1$  时，我们的近似算法仅使用一个箱子，因此是最优的。这就是说，可以取  $\text{opt}=2$ ，使该比值取最大值，并得出结论：

$$\frac{\text{approx}}{\text{opt}} \leq \frac{4}{3} + \frac{1}{3 \times 2} = \frac{3}{2}$$

有可能创建任意大小的实例，使此比值恰为  $3/2$ 。因此，一般情况下，我们不能再改进在定理 9.8 中获得的界限。

要深入了解一种近似算法的品质，一种方法是运行实验测试，将近似算法获得的解与最优解进行对比。装箱问题的近似算法已经针对很大的  $n$  值进行了广泛测试。你可能会感到奇怪，在没有一个能保证给出最优解的多项式时间算法时，怎么可能进行这种测试呢？因为我们不可能在可容忍的时间内为大的  $n$  值获得最优解。事实上，如果有了这种多项式时间算法，那就不再需要再费心考虑近似算法了。这个悖论的解答在于，对于装箱问题，我们并不需要实际计算最优解，才能了解近似算法的品质，而是可以计算近似算法所用箱中的未使用空间。这种算法使用的额外箱数不可能多于未使用空间的数量。这是因为，我们可以重新排列近似解中的物品，使它们放在最优数目个箱子中，空出额外的箱子。这个最优解中的未使用空间量再加上额外箱的总空间，就等于近似解中的未用空间量。因此，因为额外箱中的总空间等于额外箱数，所以额外箱数可以不大于近似解中的未用空间量。

在一个实证研究中，输入大小为 128 000，物品大小在 0、1 之间均匀分布，我们的近似算法平均使用大约 64 000 个箱子，平均拥有大约 100 个单位的未用空间。这意味着，平均来说，在这一研究的实例中，额外箱数的上限为 100。由定理 9.8 可以导出，当近似解为 64 000 时，

$$64\,000 \leq 1.5 \times \text{opt}$$

这意味着  $\text{opt} \geq 42\,666$ ，额外箱数不大于 21 334。可以看出，实证研究表明：该算法的执行性能平均来说远优于上限。

对于所关心的任意特定实例，可以计算出在近似算法生成的解中有多少未用空间。这样，就能判断该算法针对该实例的性能好坏。

关于近似算法的更多例子，同样可参考 Garey 和 Johnson（1970 年）的文献。

## 9.6 习题

### 9.1~9.3 节

1. 列出拥有多项式时间算法的问题。说明理由。
2. 给出一个问题，及其输入的两种编码方案。用编码方案表示它的性能。
3. 证明：一个以顶点数作为实例规模度量的图问题，与一个以边数作为实例规模试题的图问题是多项式等价的。
4. 对于 9.3 节讨论的三个一般类别，计算第  $n$  个斐波那契项的问题属于其中哪一类？说明理由。
5. 当且仅当(a)一个图是连通的，(b)每个顶点的度都是偶数，则该图拥有欧拉回路。对于所有用来判断一个图中是否存在欧拉回路的算法，为其时间复杂度计算下限。对于 9.3 节讨论的三个一般类别，此问题属于其中哪一类？说明理由。
6. 对于 9.3 节讨论的三个一般类别，为其中每一类至少列出两个问题。

### 9.4 节

7. 在你的系统上实现 9.4.1 节讨论的旅行推销员判定问题的验证算法，并研究它的多项式时间性能。
8. 试确认例 9.2 至例 9.5 中的问题属于  $NP$ 。
9. 为分团判定问题编写一个多项式时间验证算法。
10. 证明：CNF 可满足性问题向分团判定问题的约简可以在多项式时间内完成。
11. 为哈密顿回路判定问题编写一个多项式验证算法。
12. 证明：哈密顿回路判定问题向旅行推销员（无方向）判定问题的约简可以在多项式时间内完成。
13. 证明：旅行推销员（无方向）判定问题向旅行推销员判定问题的约简可以在多项式时间内完成。
14. 证明：当且仅当一个问题可约简为  $NP$  完全问题时，该问题是  $NP$  容易的。
15. 假定问题  $A$  和问题  $B$  是两种不同的判定问题。此外，假定问题  $A$  可以在多项式时间向以多对一约简为问题  $B$ 。如果问题  $A$  是  $NP$  完全的，那  $B$  是否为  $NP$  完全的？说明理由。
16. 当 CNF 可满足性问题的所有实例在每个子句中恰有三个文字，则将其称为“3-可满足性问题”。知道了“3-可满足性问题”是  $NP$  完全的，证明图的“3 着色问题”也是  $NP$  完全的。
17. 证明：如果一个问题不属于  $NP$ ，那它就不是  $NP$  容易的。因此，Presburger 算术运算和停机问题都不是  $NP$  容易的。

### 9.5 节

18. 在你的系统上实现旅行推销员问题的近似算法，并使用若干问题实例研究它们的性能。
19. 为 9.5.2 节给出的装箱问题的近似算法编写出详尽算法，并证明它的时间复杂度属于  $\Theta(n^2)$ 。
20. 对于第 5 章讨论的“子集求和”问题，能否开发一种在多项式时间内运行的近似算法？
21. 能否将一个 CNF 可满足性问题表述为一个最优化问题，以为其开发一种近似算法？也就是说，能否为表达式中的文字找出一种真值赋值，使最大可能数量的子句为真？

### 补充习题

22. 一个问题的一种算法，有没有可能在使用一种编码方案时为多项式时间算法，而在使用另一种编码方案时成为指数时间算法？说明理由。
23. 为 9.4.2 节给定的函数 `verify_composite` 编写一个更具体的算法，对其进行分析，证明它是一种多项式时间

算法。

24. 编写一个多项式时间算法，检查一个无向图中是否存在哈密顿回路，假定该图中所有顶点的度都不超过 2。
25. 汉诺塔问题是不是一个  $NP$  完全问题？它是不是一个  $NP$  容易问题？它是不是一个  $NP$  困难问题？它是不是一个  $NP$  等价问题？说明理由。这个问题在第 2 章的第 17 题中给出。
26. 给定一个包含  $n$  个正整数的列表 ( $n$  为偶数)，将这个列表分为两个子列表，使两个子列表中的整数之和的差值最小。这个问题是不是  $NP$  完全问题？这个问题是不是  $NP$  困难问题？

## 遗传算法和遗传编程

进化是种群遗传构成的变化过程。自然选择是拥有能够更好地适应环境压力的特性的有机体以远超其他类似有机体的数量生存和繁殖，进而增加这些有利特性在未来世代中存在的过程。

进化计算（evolutionary computation）是以自然选择中的进化机制为范例，努力为诸如最优化之类的问题获得近似解。进化计算的四个领域是遗传算法、遗传编程、进化编程和进化策略。本章将详细讨论前两个领域。首先简单复习一些遗传知识，以便为这些算法提供一个合适的上下文。

### 10.1 遗传知识复习

这部分复习内容非常简要，假定读者之前看过相关材料。关于遗传学的介绍，请参阅 *An Introduction to Genetic Analysis* (Griffiths 等, 2007 年) 或 *Essential Genetics* (Hartl 和 Jones, 2006 年)。

有机体是生命的个体形式，比如一株植物或一个动物。细胞是有机体的基本结构与功能单元。染色体是生物学遗传特性的载体。染色体组是一个有机体中的完整染色体集合。人类的染色体组包含 23 条染色体。单倍体包含一个染色体组；也就是说，它包含一组染色体。所以人类的一个单倍体包含 23 条染色体。二倍体包含两个染色体组；也就是说，它包含两组染色体。一组染色体中的每条染色体与另一组中的一条染色体配对。这样的染色体对称为同系对。每一对中的每个染色体称为一个同源染色体。所以一个人的二倍体中包含  $2 \times 23 = 46$  条染色体。一个同源染色体来自双亲。

体细胞是有机体身体中的一种细胞。单倍体有机体的体细胞为单倍体。二倍体有机体的体细胞为二倍体。人类是二倍体有机体。

配子是一个成熟的性繁殖细胞，它与另一个配子结合在一起，变成一个合子，最终生长成一个新的有机体。配子总是单倍体。由雄性生成的配子称为精子，由雌性生成的配子则称为卵子。生殖细胞是配子的前体。它们是二倍体。

在二倍体有机体中，每个成体产生一个配子，两个配子结合在一起构成一个合子，合子生长，变成一个新的成体。这一过程称为有性生殖。单细胞单倍体有机体通常通过一种二分裂的过程进行无性生殖，有机体直接分裂为两个新的有机体。因此，每个新的有机体都与原有机体拥有完全相同的遗传内容。一些单细胞单倍体有机体通过一种称为融合（fusion）的过程进行有性生殖。两个成体细胞首先合并，构成一个所谓的暂时二倍体性母细胞。这个暂时二倍体性母细胞类似一个同源染色体对，双亲各提供一个。一个孩子可以从双亲处各获得一个同源染色体，所以孩子并不是双亲的遗传副本。例如，如果染色体组的大小为 3，那一个孩子可以拥有  $2^3 = 8$  种不同的染色体组合。

染色体由复合脱氧核糖核酸（DNA）组成。DNA 由四个称为核苷酸的基本分子组成。每个核苷酸包含一个戊糖（脱氧核糖）、一个磷酸基团和一个嘌呤或嘧啶基。嘌呤——腺嘌呤（A）和鸟嘌呤（G）的结构相似，嘧啶——胞嘧啶（C）和胸腺嘧啶（T）也是如此。DNA 是由两链互补基因组成的大分子，每个链包含一个核苷酸序列。这两个链由核苷酸对之间的氢键结合在一起。腺嘌呤总是与胸腺嘧啶配对，鸟嘌呤总是与胞嘧啶配对。这样的每一对都称为一个正规碱基对（bp），A、G、C 和 T 称为碱基。

图 10-1 描绘了一个 DNA 片断。可以回想生物课上学的内容，这两个链相互绞在一起，构成一个右旋双螺

① 原材料来源：*Contemporary Artificial Intelligence*, Tarlor & Francis Books, Inc. 授权使用。

旋。但是，从计算的目的来看，只需要把它们看作字符串即可，如图 10-1 所示。



图 10-1 DNA 片断

基因是一个染色体片断，经常包含数千个碱基对，其大小差异很大。基因既负责有机体的结构，也负责其过程。一个有机体的基因型就是它的遗传构成，而一个有机体的表现型就是它的外在表现，它是基因型与环境相互作用的结果。

**等位基因**是一个基因几种形式中的任意一种，通常通过突变产生。遗传变异就是由等位基因造成的。

**例 10.1** 人类第 15 条染色体的 *bey2* 基因决定眼睛的颜色。蓝眼睛有一个等位基因，称为 BLUE，棕眼睛有一个等位基因，称为 BROWN。和所有基因一样，一个个体从双亲那里各获得一个等位基因。BLUE 等位基因是**隐性的**。这就是说，如果一个人收到一个 BLUE 等位基因、一个 BROWN 等位基因，那这个人就会拥有棕色眼睛。这个人拥有蓝眼睛的唯一途径就是拥有两个 BLUE 等位基因。我们也可以说明色等位基因是**显性的**。

因为一个人类配子有 23 条染色体，这些染色体的每一个都来自任一染色体组，所以一个父亲/母亲可以向其后代传递  $2^{23}=8\,388\,608$  种不同的基因组合。实际数目要远多过这个数字。在减数分裂（生成配子的细胞分裂）期间，每个染色体复制自己，并与它的对应染色体对准。这些复制品称为染色单体。通常，相对同源染色单体之间的对应基本片断会发生交换。这种交换称为染色体交叉，在图 10-2 中说明。

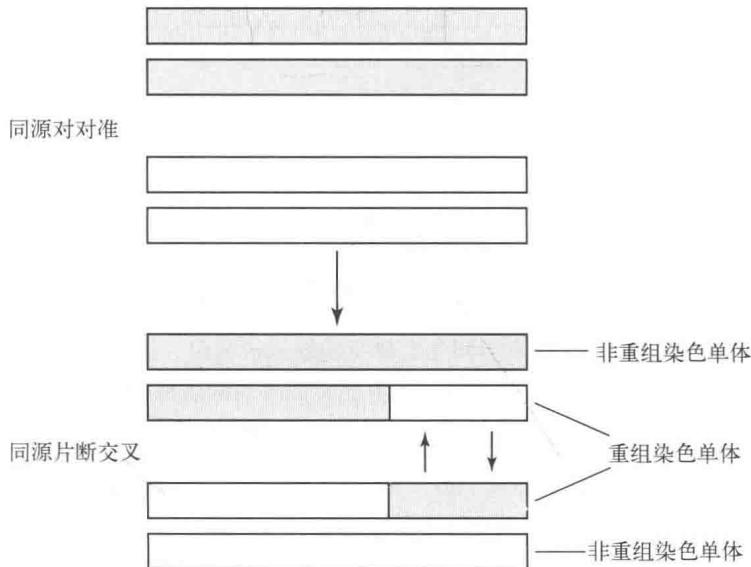


图 10-2 染色体交叉

有时，在细胞分裂期间，DNA 复制过程中会出现错误。这些错误称为突变。突变既可以在体细胞中发生，也可能在生殖细胞中发生。据信，生殖细胞中的突变是进化过程中所有变异的来源。另一方面，体细胞中的突变可能会影响一个有机体（例如，导致癌症），但对其后代没有影响。

在**置换突变**中，一个核苷酸直接由另一个核苷酸代替。当染色体中插入一个 DNA 片断时，就会发生**插入突变**；当从一个染色体中移去一个 DNA 片断时，会发生**缺失突变**。

进化是种群遗传构成发生变化的过程。据信，遗传构成中的变化是由突变造成的。前面已经提到，自然选择是一种过程，有机体通过这一过程获得一些特性，使它们能更好地适应环境压力，比如捕食性天敌、气候变

化、食物或配偶的竞争，从而能生存下来，使繁殖数量超过其他类似有机体，增加这些有利特性在未来世代中的存在。因此，这些有利特性是由一些等位基因赋予个体的，自然选择会导致这些等位基因相对频率的增加。等位基因相对频率这种偶然发生的变化过程称为基因漂变。至于进化变化的主因是自然选择还是基因漂变，科学界存在一些不同意见（Li, 1997年）。

## 10.2 遗传算法

我们首先介绍基本遗传算法，然后提供两种应用。

### 10.2.1 算法

遗传算法以单倍体有机体中的融合为模型。一个问题的候选解由种群中的单倍体个体表示。每个个体有一条染色体。染色体的字母表不再是真实有机体中的A、G、C和T，而是用于表示解的字母。在每个世代中，将允许繁殖特定数量的适应个体。代表更佳解的个体要更适应一些。来自两个适应个体的染色体将排在一起，通过交叉交换基因材料（问题解的子字符串）。此外，也可能发生突变，这样会导致生成下一代个体。一直重复这一过程，直到满足某一终止条件。下面是一般遗传算法的高级伪代码。

```

void generate_populations()
{
    t=0;
    初始化种群 P0;
    repeat
        评估种群 Pt 中每一个体的适应度;
        基于适应度选择要繁殖的个体;
        对选定的个体执行交叉或突变;
        t++;
    until 满足终止条件;
}

```

在根据适应度选择个体时，不一定直接选择最适应的个体，而是可以同时应用开发（exploitation）和探索（exploration）。一般来说，在评估一个搜索空间中要研究的候选区域时，开发是指仅关注表现很好的区域，开发利用已经获得的知识；而探索是寻找新的区域，不考虑它们的当前表现。在选择个体时，我们将以概率 $\epsilon$ 进行探索，选择一个随机个体，以概率 $1-\epsilon$ 进行开发，选择一个适应个体。

### 10.2.2 说明范例

我们的目标是找到一个 $x$ 值，使 $f(x) = \sin\left(\frac{x\pi}{256}\right)$ 在区间 $0 \leq x \leq 255$ 上取最大值，其中 $x$ 仅取一个整数。当然，正弦函数在 $\pi/2$ 处取最大值1，也就是说，当 $x=128$ 时该函数取最大值。因此，从实践的角度来看，没有必要为解决这一问题而开发一种算法。但是，我们这样做是为了说明此类算法的各个方面。该算法的开发步骤如下。

- (1) 选择一个字母表，用来表示问题的解。因为候选解就是从0到255的整数，所以可以使用8个比特表示每个个体（候选解）。例如，整数189可表示为10111101。
- (2) 确定多少个个体构成一个种群。一般情况下，可以有数千个个体。在这个简单的例子中，将使用8个个体。
- (3) 确定如何初始化该种群。这一工作经常是随机完成的。我们将随机生成8个介于0到255之间的数字。

表 10-1 中给出了可能采用的初始值。

表 10-1 个体的初始种群及其适应度

个 体	$x$	$f(x)$	归一化 $f(x)$	累积归一化 $f(x)$
1 0 1 1 1 1 0 1	189	0.733	0.144	0.144
1 1 0 1 1 0 0 0	216	0.471	0.093	0.237
0 1 1 0 0 0 1 1	99	0.937	0.184	0.421
1 1 1 0 1 1 0 0	236	0.243	0.048	0.469
1 0 1 0 1 1 1 0	174	0.845	0.166	0.635
0 0 1 0 0 0 1 1	74	0.788	0.155	0.790
0 0 1 0 0 0 1 1	35	0.416	0.082	0.872
0 0 1 1 0 1 0 1	53	0.650	0.128	1.000

(4) 确定如何评估适应度。因为我们的目标是使  $f(x)=\sin(x\pi/256)$  取最大值，所以个体  $x$  的适应度就是这个函数的值。

(5) 确定选择哪些个体进行繁殖。这里将综合利用探索和开发。将每个适应度除以所有适应度之和 ( 5.083 )，生成归一化适应度。这样，归一化适应度之和将等于 1。随后使用这些归一化适应度确定累积适应度值，它根据每个个体的适应度，为每个个体在轮盘机上提供一个个楔子。如表 10-1 所示。例如，第二个个体的归一化适应度为 0.093，该个体被指定一个与区间(0.144, 0.237]相对应的楔子，该区间的宽度为 0.093。然后由区间(0, 1]选择一个随机数。该数字将落在恰好为一个个体指定的范围内，而这就是选择的那个个体。这一过程将执行 8 次。

假定被选择进行繁殖的个体是表 10-2 中的个体。注意，一个个体可以出现多次，它出现的频率取决于其适应度。

表 10-2 被选择进行繁殖的个体

个 体
0 1 1 0 0 0 1 1
0 0 1 1 0 1 0 1
1 1 0 1 1 0 0 0
1 0 1 0 1 1 1 0
0 1 0 0 1 0 1 0
1 0 1 0 1 1 1 0
0 1 1 0 0 0 1 1
1 0 1 1 1 1 0 1

(6) 确定如何进行交叉和突变。首先，使个体随机配对，得到 4 对。对于每一对，沿个体随机选择两个点。交换这两个交叉点之间的基因材料。表 10-3 给出了可能的结果。注意，如果在个体中，第二个点出现在第一个点之前，那以回绕方式执行交叉。表 10-3 中的第三对个体显示了这种情景。根据表 10-1 和表 10-3 中的值，交叉之前的平均适应度为 0.635，而交叉之后则为 0.792。此外，在交叉之后，两个个体的适应度高于 0.99。

表 10-3 由交叉得到的双亲和孩子

双 亲	孩 子	$x$	$f(x)$
0 1 1 1   0 0 0   2 1 1	0 1 1 1   1 0 1   2 1 1	119	0.994
0 0 1 1   0 1   0 1	0 0 1 0   0 0   0 1	33	0.394
1 1   0 1 0 1   2 0 0 0	1 0   1 0 1 0   2 0 0 0	168	0.882
1 0   1 0 1   1 1 0	1 1   0 1 1   1 1 0	222	0.405
0 1   2 0 0 1 0 1   0	1 0   2 0 0 1 0 1   0	138	0.992
1 0   0 0 1 1 1   0	1 0   1 0 1 1 1   0	110	0.976
0 1 1 0 0   0 1 1   2	0 1 1 0 0   1 0 1   2	101	0.946
1 0 1 1 1   1 0 1   1	1 0 1 1 1   0 1 1   1	187	0.749

接下来确定如何执行突变。对于每个个体中的每个比特，随机决定是否翻转该比特（所谓翻转就是将 0 改为 1，或者将 1 改为 0）。突变概率通常在 0.01 到 0.001 范围内。

(7) 确定如何结束。我们可以在达到某一最大世代个数时、在超过某个指定的时间时、在大多数适应个体的适应度达到特定水平时，或者在满足这些条件之一时终止。在这个例子中，可以在生成 10 000 个世代时，或者在适应度超过 0.999 时终止。

注意，步骤(2)、(3)、(5)和(7)是通用的，也就是说，可以将这些策略运用到大多数问题上。

### 10.2.3 旅行推销员问题

旅行推销员问题 (TSP) 是一个著名的 *NP* 困难问题。*NP* 困难问题是一类特定问题，从来没有人为它们开发出多项式时间算法，但也从来没有人证明这种算法不可能存在。

假定一位销售员计划到  $n$  个城市进行一次销售旅行。每个城市都由一条道路与其他某些城市相连。为使旅行时间最短，我们希望找出一条最短路线（称为旅程），它开始于销售员所在的城市，其他每个城市都恰好访问一次，然后结束于起始城市。确定最短旅程的问题是 TSP。注意，起始城市与最短旅程无关。

TSP 问题用一个加权有向图表示，其中的顶点表示城市，边上的权重表示道路长度。一般情况下，一个 TSP 实例中的图不需要是完全的，所谓完全图就是指每两个顶点之间都存在一条边。此外，如果边  $v_i \rightarrow v_j$  和  $v_j \rightarrow v_i$  都在图中，那它们的权重也不需要相等。除了运用运输调度之外，TSP 还被应用于其他一些问题，比如调度一台机器在电路板上打孔和 DNA 排序。

接下来将给出 TSP 的三个遗传算法。

#### 1. 顺序交叉

首先介绍顺序交叉。（注意：只给出与 10.2.2 节所示内容不同的步骤。）

(1) 选择一个字母表，用来表示该问题的解。要表示 TSP 的一个解，一种很直接的方法就是将顶点标记为 1 至  $n$ ，并按照访问顺序列出这些顶点。例如，如果共有 9 个顶点，[2 1 3 9 5 4 8 7 6] 表示在访问顶点  $v_2$  之后访问  $v_1$ ，在访问  $v_1$  之后访问  $v_3$ ……在访问  $v_6$  之后访问  $v_2$ 。再次重复，起始顶点是无关紧要的。

(4) 适应度是旅程的长度，其中，长度越短的旅程越为适应。

(6) 确定如何执行交叉和突变。和前面一样，对个体随机配对，并对于每一对个体，沿个体随机选择两点。这些点之间的片断称为选取部分。我们必须确定一次交叉的结果是合法的旅程，也就是说，每个城市都必须仅被列出一次。因此，不能简单地交换选取部分。在顺序交叉中，孩子中的选取部分与一个亲本中的选取部分具有相同值，而非选取部分的区域则填充有来自另一亲本的值，省略了还没有显示的值，填充顺序就是这些值在另一亲本中的出现顺序，其起点是另一亲本的选取部分。这些值称为它的模板。表 10-4 显示了上述内容。注意，孩子  $c_1$  选取部分的值为 [9 5 4]，与亲本  $p_1$  相同。亲本  $p_2$  的选取部分为 [6 8 9]。这个亲本的模板通过以下方式构建：从位置 6 开始，依次列出所有不在 [9 5 4] 中的城市。这一过程是环绕进行的。因此，模板为 [6 8 7 1 3 2]。这些值被依次填充到孩子  $c_2$  的非选取部分区域。

表 10-4 顺序交叉的例子

双亲	另一模板	孩子
$p_1: 2 \ 1 \ 3   9 \ 5 \ 4   8 \ 7 \ 6$	来自 $p_2$ 的 6 8 7 1 3 2	$c_1: 6 \ 8 \ 7   9 \ 5 \ 4   2 \ 1 \ 3 \ 2$
$p_2: 5 \ 3 \ 2   6 \ 8 \ 9   7 \ 1 \ 4$	来自 $p_1$ 的 5 4 7 2 1 3	$c_2: 5 \ 4 \ 7   6 \ 8 \ 9   2 \ 1 \ 3$

如果这个图是不完全的，那就需要检查新孩子是否表示一个实际旅程，如果不是，则拒绝该交叉。

至于突变，不能直接将一个给定节点与另一节点进行交换来完成一个位置的突变，因为一个顶点可能会因此而出现两次。但是，我们可以通过交换两个顶点或者颠倒一个顶点子集的顺序来进行突变。然而，如果图是不完全的，那就必须确认最终结果表示一个实际旅程。

## 2. 最近邻交叉

TSP 的最近邻算法 (NNA, Nearest Neighbor Algorithm) 是一种贪婪算法。NNA 从任意顶点开始一个旅程，然后反复将最邻近的未访问顶点添加到部分旅程中，直到完成一个旅程。NNA 假定图是完全的，否则它可能无法得出一个旅程。该算法如下。

### 算法 10.1 旅行推销员问题的最近邻算法 (NNA)

问题：确定一个加权无向图中的最优旅程。图中权重是非负数。

输入：一个加权无向图  $G$ ;  $G$  中的顶点数  $n$ 。

输出：一个变量 tour，其中包含了  $G$  中顶点的一个有序列表。

```
void generate_tour (int n, graph G, orderedlist& tour)
{
    tour = [vi] 其中  $v_i$  是  $G$  中随机选择的一个顶点;

    repeat
        在未访问的顶点中找出与 tour 当前最末顶点距离最近的顶点，将其添加到 tour 中;
    until tour 中有  $n$  个顶点;
}
```

图 10-3a 给出了 TSP 的一个实例，其中的无向边表示在两个方向上都有一条具有给定权重的边，图 10-3b 给出了最短旅程，图 10-3c 给出了在从  $v_4$  开始应用 NNA 时获得的旅程，图 10-3d 给出了在从  $v_1$  开始应用 NNA 时获得的旅程。注意，后面这个旅程是最优的。

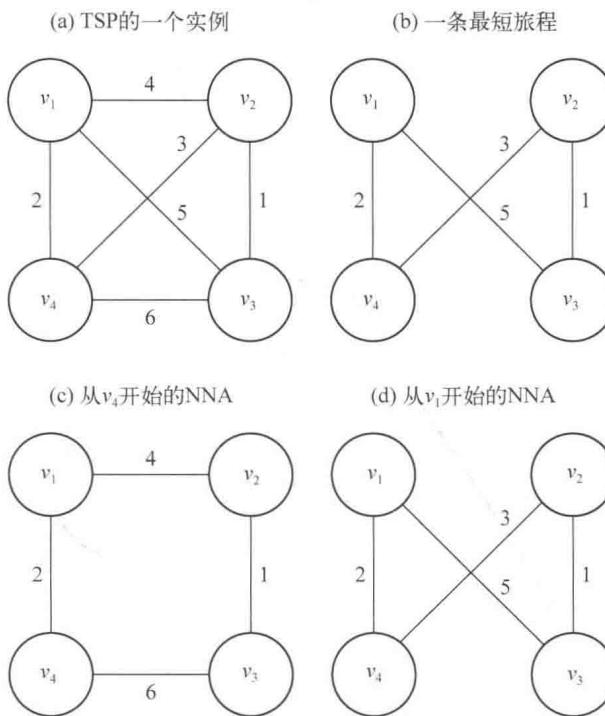


图 10-3 说明 NNA 的一个 TSP 实例

接下来给出最近邻交叉 (NNX, Nearest Neighbor Crossover)，它是 2010 年由 Süral 等人设计的。以下是这些研究者在评估该算法时使用的步骤。

- (1) 选择一个字母表，表示问题的解。其表示与顺序交叉中相同，在 10.2.3 节的开头给出。
- (2) 判断有多少个个体组成一个种群。使用的种群大小为 50 和 100。
- (3) 所尝试的第一种方法是随机初始化整个初始种群。第二种方法是随机初始化种群的一半，另一半则使

用涉及 NNX 和贪婪边算法（下面讨论）的混合方法进行初始化。

(4) 确定如何评估适应度。适应度与顺序交叉中相同。

(5) 判断要选择哪些个体进行繁殖。对于按适应度排序的个体，允许繁殖其中的前 50%。将这些个体中每个个体的四个副本放入池中，然后从池中随机选择双亲对，选择后不再放回。

(6) 确定如何执行交叉和突变。在 NNX 中，双亲仅生一个孩子。因此，这一过程实际上并不是交叉，但我们仍然这样称呼它。双亲首先合并，构成一个联合图，其中包含了两位双亲中的边。这一过程如图 10-4 中所示。接下来，向这个联合图应用 NNA。这一过程也在图 10-4 中给出。在这个图中，从顶点  $v_6$  开始应用 NNA，并得到一个比其双亲都要更适应的孩子。现在还必须证明，如果从  $v_1$  开始，那就不会出现这一情景。如果从  $v_3$  开始，那就会在  $v_1$  处碰到死胡同，不会得到一条旅程。如果选择的顶点会得到一条旅程，可以尝试其他顶点，直到其中一个可以得到旅程为止。如果所有起始顶点都不能得到旅程，可以使完整图中的其他边适用于我们的旅程。

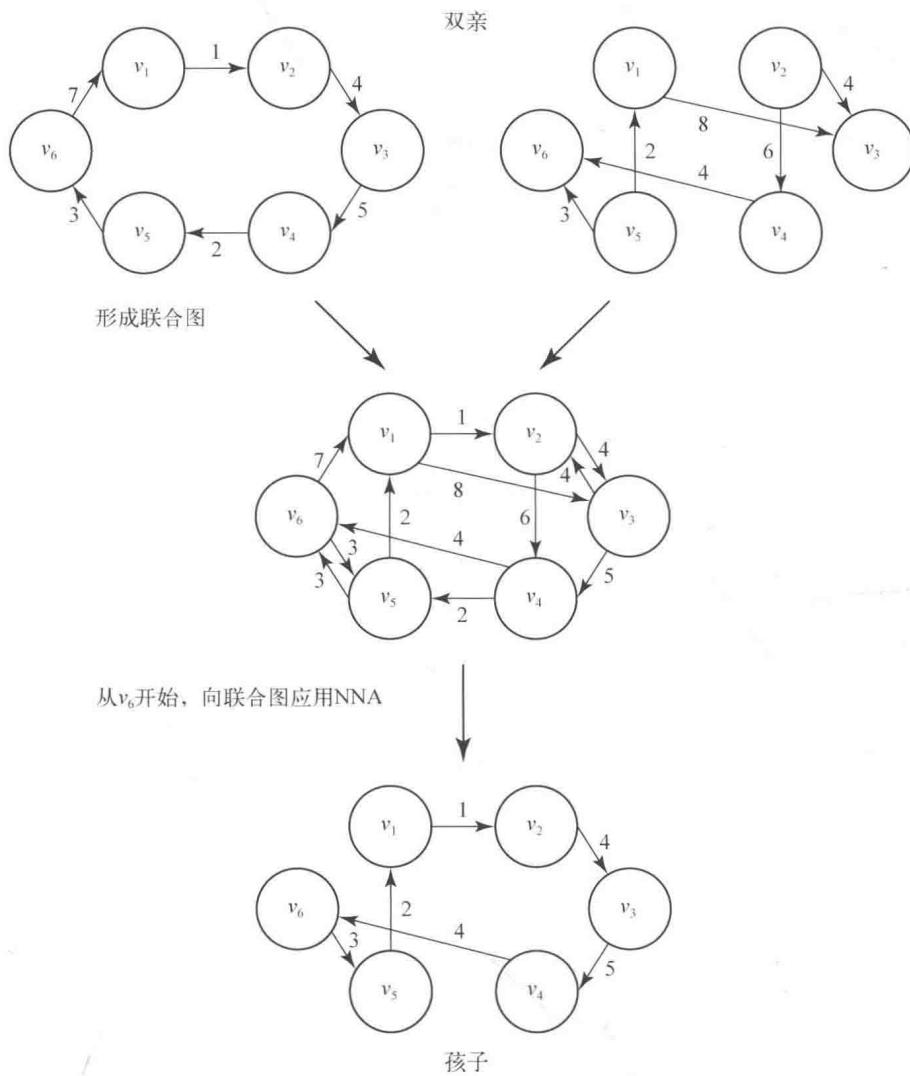


图 10-4 形成联合图，然后从顶点  $v_6$  开始，向该图应用 NNA

注意，因为使用了每位双亲的 4 个副本，而且每对双亲都生成 1 个后代，所以孩子一代的大小是允许繁殖的双亲数量的两倍。但是，因为只允许一半双亲这样做，所以每一代的种群大小保持不变。

突变的执行过程如下。随机选择两个顶点，并颠倒连接这两个顶点的子路径。这一点如图 10-5 所示，其

中选择的顶点是  $v_1$  和  $v_7$ 。共有两个突变版本。在版本 M1 中，仅向每一代中的最佳后代应用突变。在版本 M2 中，向所有后代应用它。

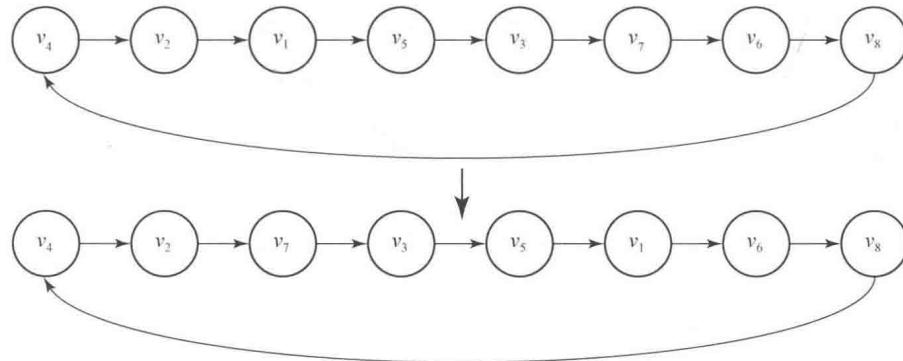


图 10-5 一种突变，其中连接顶点  $v_1$  与  $v_7$  的子路径会被颠倒

作为一种变化形式，NNX 的一种随机版本将随机地选择下一条与当前顶点连接的边。被选定的机率与该边的长度成反比。这种做法可能会提高种群多样性，从而增加所探讨的研究空间部分。但是，Süral 等人在 2010 年发现，这种随机方法的执行性能明显差于确定性版本，它们在最终测试中没有包含这一版本，我们稍后进行讨论。

(7) 确定何时结束。当两个连续世代的平均适应度相同，或者在生成 500 个世代时，此算法结束。

NNA 和 NNX 与最短路径问题的 Dijkstra 算法非常类似，和该算法一样，需要花费  $\theta(n^2)$  的时间，其中  $n$  是顶点的个数。

### 3. 贪婪边交叉

在贪婪边算法 (GEA, Greedy Edge Algorithm) 中，首先按非递减顺序对各边排序。然后从第一条边开始，贪婪地将各条边添加到旅程中，同时确定任何顶点都不会有超过两条边与其接触，也不会创建小于  $n$  的环。GEA 假定图是完全的，否则，它可能不会得出旅程。算法如下。

#### 算法 10.2 旅行推销员问题的贪婪边算法 (GEA)

问题：确定一个加权无向图中的最优旅程。图中权重是非负数。

输入：一个加权无向图  $G$ ， $G$  中的顶点数  $n$ 。

输出：一个变量 tour，其中包含了  $G$  中顶点的一组边。

```
void generate_tour (int n, graph G, setofedges& tour)
{
    按非递减顺序对各边排序;
    tour = ∅;           //旅程由一组边表示。
    repeat
        if 向旅程中添加下一条边不会导致一个顶点有两条与其接触的边
            and 不会生成一个小于 n 的环
                将该边添加到 tour 中;
    until 旅程中有 n-1 条边;
}
```

图 10-6 使用与图 10-3 相同的实例来说明 GEA。

贪婪边交叉算法 (GEX，也见于 Süral 等人在 2010 年的文献) 的步骤均与 NNX 相同，只有第 6 步除外，给出如下。

(6) 确定如何执行交叉和突变。和在 NNX 中一样，在 GEX 中，双亲也是结合起来构成一个联合图。然后向这个联合图中的边应用 GEA。如果这一过程不会得到旅程，则向完全图中应用 GEA，以获得旅程的剩余边。但是，这一过程会导致探索很少，从而过多地保留边，可能会过早地收敛到一个低质量的解。为增加探索，新

旅程的前一半可以从联合图中获取，后一半可以从完全图中获取。在开始研究时，这一版本的性能要远优于尽量多地从联合图中获取边的版本。下面的评估中就是采用这一版本。

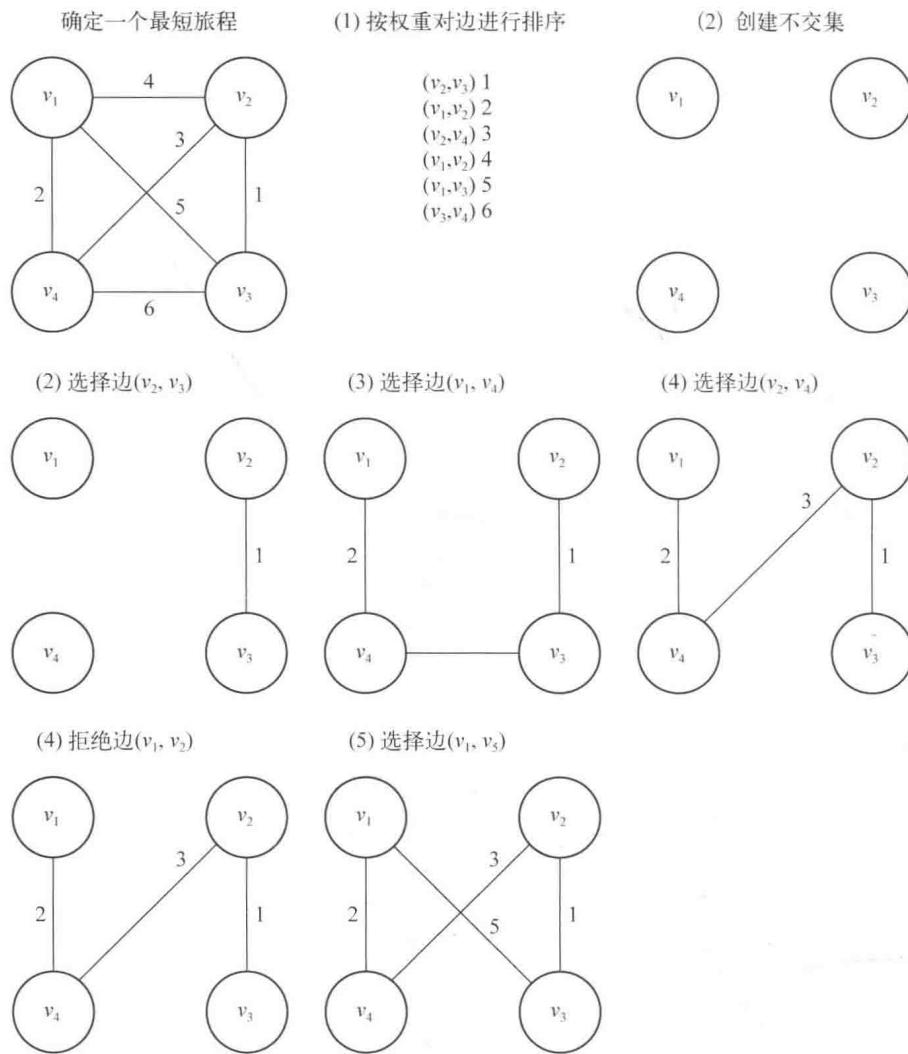


图 10-6 TSP 的一个实例，说明贪婪边算法

NNA 和 NNX 与最小生成树问题的 Kruskal 算法非常类似，和该算法一样花费  $\theta(n^2 \log n)$  和  $\theta(m \log m)$  时间，其中  $n$  是顶点数， $m$  是边数。

#### 4. 评估

和许多探索性算法一样，遗传算法也没有可证实的正确性能。因此，我们通过研究它们针对大量问题实例的性能来评估它们。Süral 等人（2010 年）对 NNX 和 GEX 就是这样做的，如下所示。

首先，我们从 TSPLIB 中获得 TSP 的 10 个实例，表示真实城市之间的距离，这些距离是对称的（在两个方向上相同）。最大实例的  $n$ （城市个数）=226。他们研究了 NNX 和 GEX 的各个版本，没有突变的、具有突变 M1 的（在前文讨论过）和具有突变 M2 的（在前文讨论过）。此外，他们还尝试随机（R）初始化第一种群；根据最近邻和贪婪边启发式算法，使一种混合方法（H）初始化第一种群。对于突变类型和初始化类型的每种组合形式，他们针对 10 个实例中的每个实例，将每个算法运行 30 次，共运行 300 次。该算法以 ANSI C 编码，运行机器为奔腾 IV 1600 MHz 计算机，内存为 256 MB RAM，操作系统为 RedHat Linux 8.0。

表 10-5 给出了对所有 300 次运行求取的平均值。表中的标题表示以下内容。

- 算法：所用算法。
- 突变：所用变异的类型。“无”表示没有突变。
- 种群初始化：种群是随机初始化（R），还是使用混合初始化（H）。
- 偏差：最终最佳解背离最优解的百分比值偏差。
- 世代数：直到收敛的世代数。
- 时间：直到收敛时的时间数（秒）。

表 10-5 对 10 个小问题实例重复 30 次时得到的平均结果，种群规模为 50

算 法	突 变	种群初始化	偏 差	世 代 数	时 间
NNX	无	R	3.10	45.39	0.38
		H	4.82	33.52	2.95
	M1	R	1.67	40.21	0.62
		H	1.57	36.09	3.55
	M2	R	0.55	53.37	5.52
		H	0.55	43.53	8.11
GEX	无	R	12.54	17.35	48.23
		H	7.19	16.37	54.27
	M1	R	4.36	60.44	208.70
		H	3.67	48.44	178.65
	M2	R	3.30	26.30	82.79
		H	3.01	25.83	90.58
50%NNX	无	R	8.15	42.50	73.25
50%GEX	无	H	5.53	38.47	75.67
		R	1.92	66.04	113.81
	M1	H	1.68	61.21	112.77
		R	1.76	19.25	26.40
	M2	H	1.61	20.68	34.19
90%NNX	无	R	7.23	41.16	13.39
10%GEX	无	H	5.19	34.93	14.95
		R	1.84	55.60	19.14
	M1	H	1.67	46.93	20.16
		R	0.51	37.13	19.26
	M2	H	0.48	37.24	21.95
95%NNX	无	R	6.69	41.23	6.74
5%GEX	无	H	5.06	33.04	8.93
		R	1.77	52.62	10.03
	M1	H	1.41	44.33	11.30
		R	0.49	37.15	11.58
	M2	H	0.44	36.19	14.88

从表 10-5 中可以看出，NNX 的执行性能要远优于 GEX，突变 M2 执行的突变最好，混合初始化的性能并没有远优于随机初始化。研究人员随后研究了在使用 NNX 时，采用不同比例的 GEX 能否提高性能。表 10-5 也显示了这些结果。例如，当表中说 50% 的 NNX 和 50% 的 GEX 时，是说在 50% 的时间内，下一代是使用 NNX 生成的，另外 50% 的时间里，是使用 GEX 生成的。可以观察到，在 NNX 应用百分比较高时采用突变 M2，其性能相对于纯粹的 NNX 略有改进。

研究人员根据这些结果得出了结论：考虑到计算时间的增加，混合使用两种算法是不划算的。他们还另外执行了一些仅使用 NNX 的试验。

他们仅使用 NNX、采用随机初始化，但种群数量为 100，于是得到了表 10-6 中的结果。注意，对于突变

M2，平均百分比偏差为 0.35，平均时间为 26.2。再次研究表 10-5，可以看出，使用同一组合形式，但种群数量为 50，则平均百分比偏差为 0.55，平均时间为 5.52。这一精度的提高应当值得多花的时间了。

表 10-6 对 10 个小问题实例重复 30 次时得到的平均结果，其中种群大小为 100，初始种群随机生成

算 法	突 变	偏 差	时 间
NNX	无	5.40	18.4
	M1	1.44	26.4
	M2	0.35	26.2

接下来，研究人员研究了来自 TSPLIB 的更大型实例，其中  $318 \leq n \leq 1748$ 。他们前面已经得出结论，为混合初始化增加的时间也是不值得的，所以他们采用随机初始化，种群大小为 100，分别对这些实例将 NNX 运行了 10 次。表 10-7 给出了结果。让人奇怪的是，对于大型问题实例，突变 M2 并没有远优于突变 M1，但确实需要更多的计算时间。再次查看表 10-6，可以看出，对于较小实例，M2 的性能远优于 M1，而计算成本的增加很小。

表 10-7 对 15 个大问题实例重复 10 次时得到的平均结果，其中种群大小为 100，初始种群随机生成

算 法	突 变	偏 差	时 间
NNX	无	7.61	25.3
	M1	4.94	65.0
	M2	4.70	1063.0

只是对大量实例测试一种启发式算法，并不能表明它是否算得上一种进步。我们还需要看它相对于之前已经存在的启发式算法有所进步。Süral 等人（2010 年）使用 10 种基准 TSP 实例，将 NNX 与启发式 TSP 算法 Meta-RaPS（DePuy 等，2005 年）和 ESOM（Leung 等，2004 年）进行了对比。表 10-8 给出了比较结果。对于每个问题实例，NNX-M1（突变 M1）或 NNX-M2（突变 M2）的性能都是最好的。

表 10-8 对于 10 个问题实例，比较 NNX 与 TSP 的其他两种启发式算法

问 题	NNX-M1		NNX-M2		Meta-RaPS		ESOM3	
	偏差	时间 <sup>1</sup>	偏差	时间 <sup>1</sup>	偏差	时间 <sup>2</sup>	偏差	时间 <sup>3</sup>
ei101	0.93	8.7	0.82	14.3	NA	NA	3.43	NA
bier127	0.62	15.3	0.28	12.0	0.90	48	1.70	NA
pr136	2.87	18.4	0.37	35.2	0.39	73	4.31	NA
kroa200	1.78	87.3	0.32	98.6	1.07	190	2.91	NA
pr226	0.79	93.0	0.01	21.6	0.23	357	NA	NA
lin318	1.87	8.0	2.01	105	NA	NA	2.89	NA
pr439	3.44	10.0	1.48	240	3.30	2265	NA	NA
pcb442	4.75	15.0	3.18	270	NA	NA	7.43	NA
pcb1173	3.00	97.0	8.01	1230	NA	NA	9.87	200
vm1748	7.05	203	7.09	4215	NA	NA	7.27	475

1. 奔腾 4, 16 GHz

2. AMD Athlon 900 MHz

3. SUN Ultra 5/270

### 10.3 遗传编程

在遗传算法中，“染色体”或“个体”表示的一个问题的解，而在遗传编程中，个体表示的是解决问题的程序。个体的适应度函数通过某种方式来衡量该程序解决该问题的完美程度。我们首先从一个初始的程序种群入手，允许更适应的程序通过交叉繁殖，对孩子的种群执行突变，然后重复这一过程，直到满足某一终止条件。

这一过程的高级算法与遗传算法完全相同。但是，出于完整性考虑，以下还是给出了这一算法。

```

void generate_populations()
{
    t=0;
    初始化种群 P0;

    repeat
        评估种群 Pt 中每一个体的适应度;
        基于适应度选择要繁殖的个体;
        对选定的个体执行交叉或突变;
        t++;
    until 满足终止条件;
}

```

遗传程序中的个体（程序）用树表示；在每个节点中，或者有一个终止符号（terminal symbol），或者有一个函数符号（function symbol）。如果一个节点是一个函数符号，它的参数就是它的孩子。举个例子，假定有以下数学表达式（程序）：

$$\frac{x+2}{5-3x} \quad (10.1)$$

它的树结构表示如图 10-7 所示。

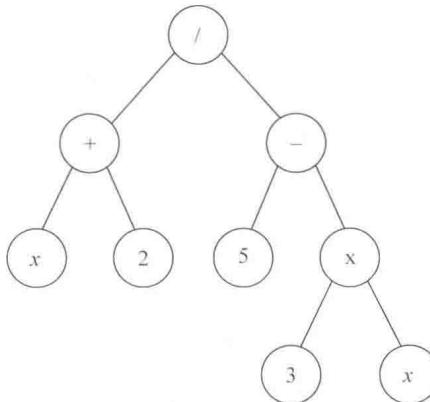


图 10-7 表达式 10.1 的树

### 10.3.1 说明范例

演示遗传编程的一种简单方法是给出一种应用，已知点对  $(x_i, y_i)$  满足函数  $y=f(x)$ ，该应用就从这些点对来了解该函数。例如，假设有表 10-9 中给出的点对。

表 10-9 我们希望基于以下 10 个点来了解一个描述  $x$  与  $y$  之间关系的函数

$x$	$y$
0	0
0.1	0.005
0.2	0.020
0.3	0.045
0.4	0.080
0.5	0.125
0.6	0.180
0.7	0.245
0.8	0.320
0.9	0.405

这些点实际上都是由下面的函数生成的：

$$y=x^2/2$$

但是，假定我们并不知道这一点，而且要试图找出这个函数。为这一发现问题开发一个遗传程序的步骤如下。

(1) 确定终止集合 T。设此终止集合 T 中包含符号  $x$  和介于 -5 到 5 之间的整数。

(2) 确定函数集 F。设该函数集为符号 +、-、× 和 /。注意，在必需时，也可以包含其他函数，比如正弦、余弦等。

(3) 确定由多少个个体组成一个种群。我们的种群大小将为 600。

(4) 确定如何初始化种群。每个初始个体都是由一个称为培育树的过程创建。首先，由  $T \cup F$  随机生成一个符号。如果它是一个终止符号，则停止工作，我们的树由唯一的节点组成，其中包含了这个符号。如果它是一个函数符号，则为该符号随机生成孩子。沿树继续向下前进，随机生成符号，并在终止符号处停止。例如，图 10-7 中的树是通过以下随机生成符号的过程获得的。首先生成符号 /，然后是其孩子的值 + 和 -。为 + 生成的孩子是  $x$  和 2。我们在这些孩子处停止。为 - 生成的孩子为 5 和 ×。我们在 5 处停止。最后为 × 生成孩子 3 和  $x$ 。

(5) 确定一个适应度函数。这里使用的适应度函数为平方差。也就是说，函数  $f(x)$  的适应度如下：

$$\sum_{i=1}^{10} (f(x_i) - y_i)^2$$

其中每个  $(x_i, y_i)$  都是表 10-9 中的一个点，错误越小的函数，越为适应。

(6) 判断选择哪些个体进行繁殖。使用一个包含 4 个个体的锦标赛选择过程。在锦标赛选择过程中，随机从种群中选择  $n$  个个体。在本例中， $n=4$ 。我们说，这  $n$  个个体进入一场锦标赛，其中  $n/2$  个最适应的获胜，另外  $n/2$  个失败。 $n/2$  个胜者获准生成  $n/2$  个孩子。这些孩子在下一代中将代替种群中的  $n/2$  个失败者。在本例中，2 个获胜者繁殖两次，生成 2 个孩子来代替 2 个失败者。

小规模锦标赛将导致低选择过程，而大规模锦标赛将导致高繁殖压力。

(7) 判断如何执行交叉和突变。两个个体之间的交叉将通过交换随机选择的子树来实现。图 10-8 中显示了这样一种交叉。突变的执行过程是随机选择一个节点，然后培育一棵新子树，以代替选定节点处的子树。对 5% 的后代随机执行突变。

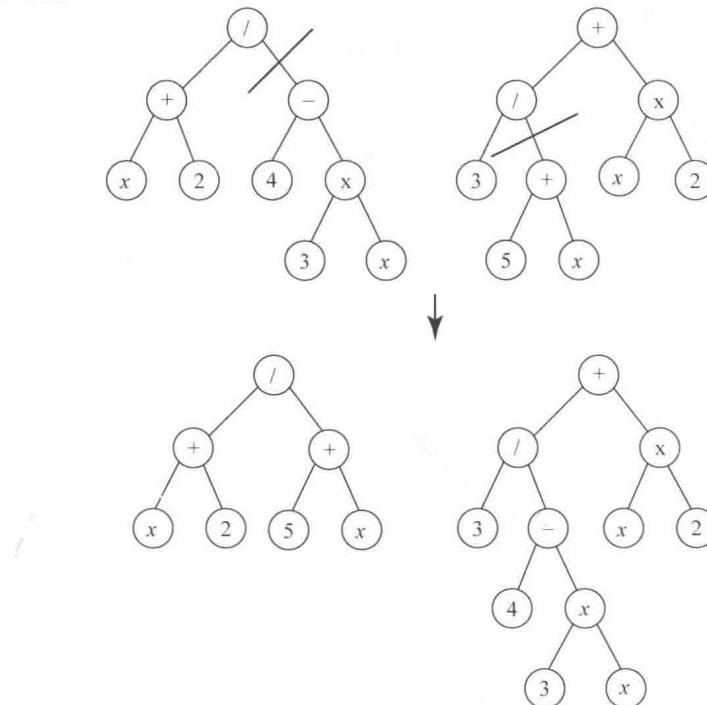


图 10-8 通过交换子树进行交叉

(8) 判断何时终止。当最适应个体的平方差为 0 时, 或者在生成了 100 个世代时, 终止。

Banzhaf 等人(1998 年)将前面刚刚介绍的技术应用于表 10-9 中的数据。下面给出前 4 代中最适应的个体:

世 代	最适应的个体
1	$\frac{x}{3}$
2	$\frac{x}{6-3x}$
3	$\frac{x}{x(x-4)-1+\frac{4}{x}-\frac{\frac{9(x+1)}{5x}+x}{6-3x}}$
4	$\frac{x^2}{2}$

最适应的个体在第 3 代扩展为一棵非常大的树, 但在第 4 代又收缩为正确的解(生成该数据的函数)。

### 10.3.2 人造蚂蚁

考虑这样一个问题: 设计一个沿食物踪迹行走的机器蚂蚁。图 10-9 给出了这样一条踪迹, 称为“Santa Fe 踪迹”。每个黑方块表示一粒食物, 共有 89 个这样的方块。这个蚂蚁从标有“起点”的方块开始, 面向右前进, 它的目标是用尽可能少的步骤, 访问所有 89 个黑方块(从而吃尽踪迹上的所有食物), 然后到达标有“89”的方块。这一问题是规划问题 (planning problem) 的一个例子。

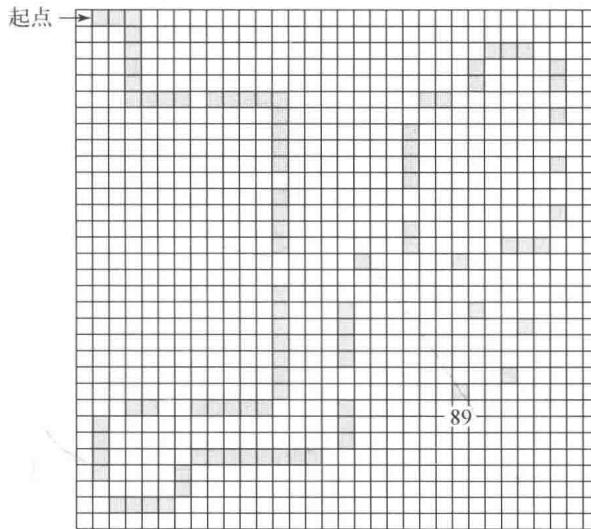


图 10-9 Santa Fe 踪迹。每个黑方块代表一粒食物

这个蚂蚁有这样一个传感器:

food\_ahead: 当蚂蚁面对的方块中有食物时, 其取值为真; 否则取值为假。  
在每个时隙中可以执行三种不同操作。

right: 蚂蚁右转 90°(不移动)。

left: 蚂蚁左转 90°(不移动)。

move: 蚂蚁向前移动到它面对的方块中。如果这个方块中有食物, 蚂蚁将吃掉食物, 从而从方块中消去食物, 从踪迹中删除它。

Koza(1992 年)为这一问题开发了下面的遗传编程算法。

(1) 确定终止集合 T。终止集合 T 包含了蚂蚁可以采取的动作。即：

$$T = \{\text{right}, \text{left}, \text{move}\}$$

(2) 确定函数集 F。此函数集如下：

(a) if\_food\_ahead(instruction1, instruction2);

(b) do2(instruction1, instruction2);

(c) do3(instruction1, instruction2, instruction3);

第一个函数在 food\_ahead 为真时执行 instruction1，否则执行 instruction2。第二个函数无条件地执行 instruction1 和 instruction2。第三个函数无条件地执行 instruction1、instruction2 和 instruction3。例如，

`do2(right, move)`

会将蚂蚁右转，然后向前移动。

(3) 确定多少个个体组成一个种群。种群大小为 500。

(4) 确定如何初始化种群。每个初始个体都通过培育树来创建（见 10.3.1 节）。

(5) 确定适应度函数。假定每个操作 (right, left, move) 执行一个时间步，每个个体允许 400 个时间步。每个个体在左上角面向东开始。适应度定义为在所分配时间内消耗的食物粒数。最大适应度为 89。

(6) 确定选择哪些个体进行繁殖。

(7) 确定如何执行交叉和突变。两个个体之间的交叉通过交换随机选择的子树来完成。图 10-8 演示了这样一个交叉。突变的执行过程是随机选择一个节点，并培育一棵新子树，替代选定节点处的子树。对 5% 的后代随机执行突变。

(8) 确定何时终止。当一个个体的适应度为 89 时，或者当达到某一最大迭代数时，终止。

在每次迭代中，种群中的每个个体（程序）将重复运行，直到执行了 400 个时间步为止。然后评价每个个体的适应度。在一次特定运行中，Koza (1992 年) 在第 22 代得到一个个体，其适应度为 89。该个体显示在图 10-10 中。第 0 代各个体的平均适应度为 3.5，而第 0 代的最适应个体的适应度为 32。

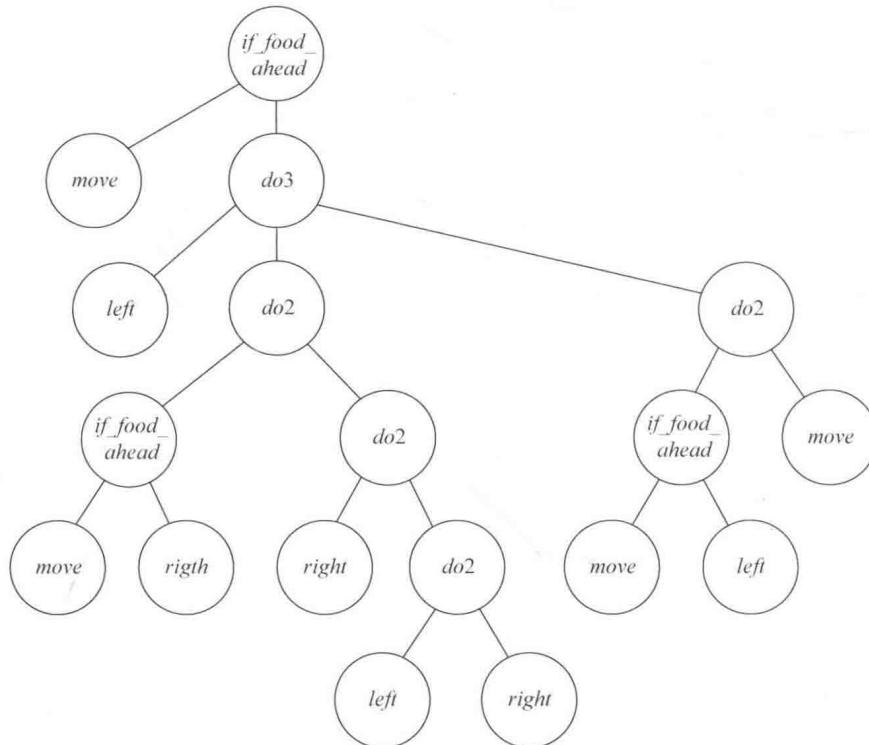


图 10-10 第 22 代一个适应度为 89 的个体

### 10.3.3 在金融贸易中的应用

在股票和其他金融市场进行投资的每个人都要面对一项重要决策：在给定的某一天里，应当买入、卖出，还是继续持有。人们已经投入了大量精力来开发出能为我们做出这一决策的自动化系统。Farnsworth 等人（2004 年）使用遗传编程开发了这样一个系统。他们的系统基于给定日期的市场指标值做出投资决策。下面来讨论这一系统。

#### 1. 开发交易系统

首先给出创建该系统的八个步骤。

(1) 确定终止集 T。终止符号是市场指标。研究人员尝试并测试了各种指标之后，确定了一组包含在最终系统中的指标。我们只给出最终指标，如下所示。

(a) S&P 500 是一个以美国经济领先行业中 500 家领先企业为基础的市场指标。对于某一给定的日期，基于 S&P 500 的指标如下：

$$\frac{S\&P 500_{\text{today}} - S\&P 500_{\text{avg}}}{S\&P 500_{\sigma}}$$

其中， $S\&P 500_{\text{today}}$  是当天 S&P 500 的值， $S\&P 500_{\text{avg}}$  是过去 200 天 S&P 500 的平均值， $S\&P 500_{\sigma}$  是过去 200 天的标准偏差。这个指标简单地表示为 SP 500。

(b) 一种有价证券的 k 天指数平均数 (EMA, k-day Exponential Moving Average) 是对过去 k 天中该有价证券进行加权平均。有价证券 x 的平滑异同移动平均线 (MACD, Moving Average Convergence/Divergence) 如下：

$$\text{MACD}(x)=12 \text{ 天 EMA}(x)-26 \text{ 天 EMA}(x)$$

MACD 被看作一种动量指标：当该指标为正时，交易员就说上升动量在增加；当它为负时，就说下行动量在增加。

这一系统中使用的第二个指标为 MACD (S&P 500)，将其简单地记为 MACD。

(c) 指标 MACD9 是 S&P 500 的 MACD 的 9 天 EMA。

(d) 设 Diff 是上升、下降有价证券的数目之差。麦克连指标 (MCCL) 如下：

$$\text{MCCL}=19 \text{ 天 EMA}(\text{Diff})-39 \text{ 天 EMA}(\text{Diff})$$

当 MCCL>100 时，交易员认为市场买入过多；当 MCCL<-100 时，卖出过多。

这个指标简单地记为 MCCL。

(e) 指标 SP500lag、MACDlag、MACD9lag、MCCLlag 是这些指标在前一天的取值。

(f) 其他终止符号包括区间 [-1, 1] 中的常数。所有指标值都被归一化到区间 [-1, 1]。

(2) 确定函数集 F。函数符号包括 +、- 和 ×，控制结构如下。

(a) if  $x > 0$  then  $y$  else  $z$ 。这一结构在树中用符号 IF 表示，它有孩子  $x$ 、 $y$  和  $z$ 。

(b) if  $x > w$  then  $y$  else  $z$ 。这一结构在树中用符号 IFGT 表示，它有孩子  $x$ 、 $w$ 、 $y$  和  $z$ 。

(3) 确定多少个个体构成一个种群。尝试各种种群大小后发现大小低于 500 时是低效的，而 2500 左右的大小可以得到好的结果。

(4) 确定如何初始化种群。每个初始个体都是通过培育树而创建的，如 10.3.1 节的讨论。最大允许的级数为 4，最大允许的总节点数为 24。在未来种群中也执行这一约束条件。添加这些限制的目的是避免过度适应，当树与训练数据非常匹配，但对不可见数据的预测价值有限时，就会发生过度适应。

(5) 确定适应度函数。这些数据是根据 1983 年 4 月 4 日到 2004 年 6 月 18 日的 S&P 收盘价格获得的。一棵给定树对其中前 4750 天的数据进行了分析。这棵树从第一天的 1 美元开始。如果该树在某一给定日子返回的数值大于 0，则生成买入信号；否则生成卖出信号。在买入或卖出时，总是投入或收回当前的所有货币。不存在部分投资的方式。如果树还在前一天的市场内部，而且生成了买入信号，那就不采取措施。同样，如果树超出

了前一天的市场，而且生成了卖出信号，也不采取措施。当完成 4750 天的交易之后，减去最初的美元数，使最终数值能够表示收益。为进一步避免过度适应，设定一个与总交易数成正比的适应度罚金。

(6) 确定选择哪些个体进行繁殖。根据适应度对种群进行排序。排在下面的 25% 被“处死”，用更适应的个体代替。

(7) 确定如何执行交叉和突变。通过交换随机选定的子树来执行两个个体之间的交叉。节点突变 (node mutation) 的完成过程是随机选择一些节点，并将它们的值随机改为取相同参数的节点。对函数节点的突变会改变操作，对终止节点的突变会改变指标或常数值。树突变 (tree mutation) 的完成过程是随机选择一棵子树，并用一棵新的随机子树代替它。

(8) 确定何时终止。最大世代数在 300~500 范围内。

图 10-11 给出了一棵树，它在一次具体运行中作为最适应的树存活下来。

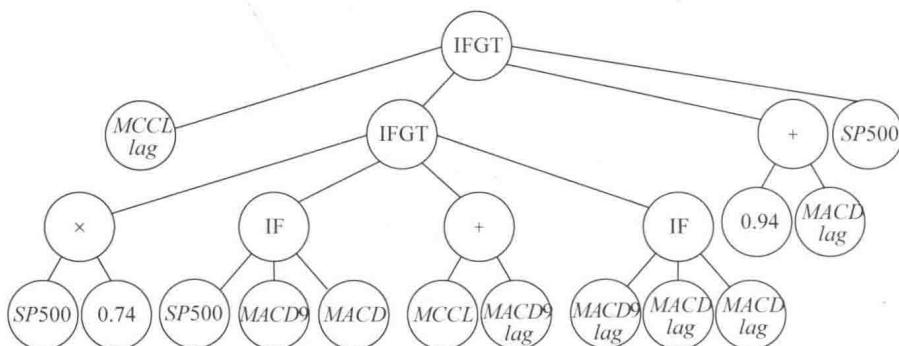


图 10-11 一棵树，它在一次运行中作为最适应树存活下来，并在评估过程中表现良好

## 2. 评估

回忆一下，这些数据是根据 1983 年 4 月 4 日到 2004 年 6 月 18 日的 S&P 收盘价获取的，利用其中前 4750 天的数据来确定适应度，以了解一个系统，剩下的 500 天则用来对系统进行评估，也就是根据这些数据计算其适应度。图 10-11 中的树表示一个系统，为该系统评估的适应度为 0.397。投资的买入并持有策略就是买入一种有价证券，并持有它。根据评估数据，这种买入并持有策略的适应度仅为 0.109 8。

## 10.4 讨论及扩展阅读

进化计算的另两个领域是进化编程和进化策略。进化编程 (evolutionary programming) 类似于遗传算法，它也是使用候选解的种群，演化出对一个特定问题的答案。进化编程的区别在于，它关心的是为可观察系统开发出与环境进行互动的行为模型。由 Fogel (1994 年) 提出了这一方法。进化策略以物种作为问题解的模型。Rechenberg (1994 年) 说道，进化策略的领域是以进化的进化为基础的。关于进化计算所有四个领域的完全介绍，请参见 Kennedy 和 Eberhart (2001 年) 的文献。

## 10.5 习题

### 10.1 节

1. 描述二倍体有机体的有性繁殖、单倍体有机体的二分裂繁殖和单倍体有机体的融合之间的区别。
2. 假定一种二倍体有机体的一个染色体组中有 10 条染色体。
  - (a) 它的每个体细胞中有多少条染色体？
  - (b) 它的每个配子中有多少条染色体？
3. 假定两个成年二倍体有机体以融合方式繁殖。

(a) 将生成多少个孩子?

(b) 这些孩子的遗传内容是否全都相同?

4. 考虑人类的眼睛颜色是由 *bey2* 基因决定的。回想一下，棕色眼睛的等位基因是显性的。对于以下每种双亲等位基因组合，确定个体的眼睛颜色。

父 亲	母 亲
BLUE	BLUE
BLUE	BROWN
BROWN	BLUE
BROWN	BROWN

## 10.2 节

5. 考虑表 10-1。假定 8 个个体的适应度为 0.61、0.23、0.85、0.11、0.27、0.36、0.55 和 0.44。计算归一化适应度和累积归一化适应度。
6. 假定我们执行如表 10-3 所示的基本交叉，双亲为 01101110 和 11010101，交叉的起始点和终止点为 3 和 7。给出所生成的两个孩子。
7. 实现一种遗传算法，用来找出使  $f(x)=\sin(x\pi/256)$  取最大值的  $x$  值，10.2.2 节对此进行了讨论。
8. 考虑图 10-12 中 TSP 的实例。假定一条边上两个方向的权重都相同。找出最短旅程。
9. 假定我们执行顺序交叉，双亲为 3 5 2 1 4 6 8 7 9 和 5 3 2 6 9 1 8 7 4，起始点和终止点为 4 和 7。给出所生成的两个孩子。
10. 考虑图 10-12 中 TSP 的实例。从每个顶点入手，应用最近邻算法。是否有其中一个可以生成最短旅程？

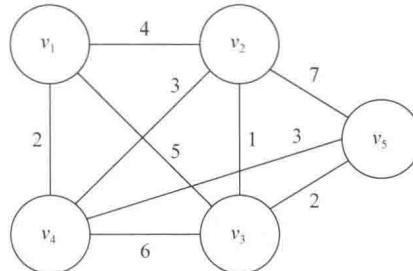


图 10-12 TSP 的一个实例

11. 给出图 10-13 中两个旅程的联合图，并从顶点  $v_5$  入手，向生成的联合图应用最近邻算法。

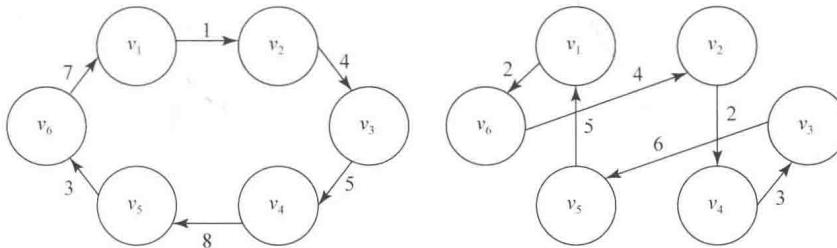


图 10-13 两条旅程

12. 向图 10-12 中 TSP 的实例应用贪婪边算法。是否会生成一条最短旅程？

## 10.3 节

13. 考虑图 10-8 中的两棵树。如果将左树中以 4 为起点的子树与右树中以 “+” 为起点的子树进行交换，试给出将会得到的新树。
14. 考虑图 10-10 的个体（程序）。给出在沿 Santa Fe 跟踪移动前 10 个时间步时，该程序所生成的移动步骤。
15. 为 10.3.2 节讨论的 Santa Fe 跟踪实现遗传编程算法。

# 数论算法



假定 Bob 希望通过互联网向 Alice 发送一封秘密情书，但他担心朋友们可能会拦截阅读该信息。如果他能对消息进行编码，使它显示为杂乱信息，而只有 Alice 可以解码这条杂乱信息，变回原信息，他可能就不需要担心朋友们截获这条消息了。数论算法可以帮助 Bob 设计这样一个系统。接下来就来讨论这些算法。

数论 (number theory) 是研究整数特性的数学分支。数论算法是解决有关整数问题的算法。例如，一种数论算法可能是找出两个整数的最大公约数。在 11.1 节复习了基本数论之后，11.2 节给出求最大公约数的欧氏算法。接下来，11.3 节复习求模运算，11.4 节给出了一种求解模线性方程的算法，11.5 节开发一种用于计算模幂的算法。11.6 节介绍用于判断一个数字是否为质数的算法。数论算法的一个重要应用是密码学 (cryptography)，这一学科是将由一方向另一方发送的信息加密，使截获该消息的人不能进行解码。11.7 节介绍了 Rivest-Shamir-Adelman (RSA) 公钥加密系统，它就是完成这一工作的系统。

在继续后续讨论之前先说一下，本章会再次像第 2 章至第 6 章一样设计算法。但与前面几章给出的方法不同，数论算法考虑的是求解特定类型的问题（也就是涉及整数的问题），这些算法并没有共用同一种方法。

## 11.1 数论回顾

现在复习数论中的一些基本要素。

### 11.1.1 合数与质数

下面的集合称为整数集。

$$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

对于任意两个整数  $n, h \in \mathbb{Z}$ ，如果存在某一整数  $k$ ，满足  $n=kh$ ，就说  $h$  能整除 (divide)  $n$ ，记作  $h|n$ 。如果  $h|n$ ，也说  $n$  可被  $h$  整除， $n$  是  $h$  的倍数 (multiple)， $h$  是  $n$  的约数 (divisor) 或因数 (factor)。

**例 11.1** 因为  $20=(5)(4)$ ，所以有  $4|20$ 。因为找不到一个整数  $k$ ，使得  $20=(k)(3)$ ，所以整数 3 不能整除 20。

**例 11.2** 12 的约数为：

$$1, 2, 3, 4, 6 \text{ 和 } 12$$

如果一个大于 1 的整数  $n$ ，只有 1 和  $n$  两个约数，就说整数  $n$  是质数 (prime)。质数没有因数。大于 1 且不是质数的整数称为合数 (composite number)。合数至少有一个因数。

**例 11.3** 前 10 个质数为：

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29$$

**例 11.4** 整数 12 是合数，因为它拥有约数 2、3、4 和 6。整数 4 是合数，因为它有约数 2。

### 11.1.2 最大公约数

如果  $h|n$  和  $h|m$ ，则将  $h$  称为  $n$  和  $m$  的公约数 (common divisor)。如果  $n$  和  $m$  都不是 0，则  $n$  和  $m$  的最大公约数 (greatest common divisor) 就是能同时整除这两个数的最大整数，记作  $\gcd(n, m)$ 。

**例 11.5** 24 的正约数为：

$$1, 2, 3, 4, 6, 8, 12 \text{ 和 } 24$$

30 的正约数为：

1, 2, 3, 5, 6, 10, 15 和 30

所以 24 和 30 的正公约数为：

1, 2, 3 和 6

$\gcd(24, 30)$  的值为 6。

关于公约数有以下定理。

**定理 11.1** 如果  $h|n$  和  $h|m$ , 则对于任意整数  $i$  和  $j$ , 有

$$h|(in+jm)$$

证明：由于  $h|n$  和  $h|m$ , 所以存在整数  $k$  和  $l$ , 使得  $n=kh$  和  $m=lh$ 。因此,

$$in+jm=ikh+jlh=(ik+jl)h$$

这就意味着  $h|(in+jm)$ 。

在继续之前, 需要再给出一些定义。对于任意两个整数  $n$  和  $m$ , 其中  $m \neq 0$ ,  $n$  除以  $m$  的商 (quotient)  $q$  等于:

$$q=\lfloor n/m \rfloor$$

而  $n$  除以  $m$  的余数 (remainder)  $r$  则是

$$r=n-qm$$

这个余数表示为  $n \bmod m$ 。不难看出, 如果  $m > 0$ , 则  $0 \leq r < m$ ; 如果  $m < 0$ , 则  $m < r \leq 0$ 。所以有

$$n=qm+r \text{ 及 } \begin{cases} 0 \leq r < m & (m > 0) \\ m < r \leq 0 & (m < 0) \end{cases} \quad (11.1)$$

除法算则表明, 式 11.1 中的整数  $q$  和  $r$  不仅存在, 而且是唯一存在的。

**例 11.6** 下表给出了一些商和余数。

$n$	$m$	$q=\lfloor n/m \rfloor$	$r=n-qm$
23	5	$\lfloor 23/5 \rfloor = 4$	$23-(4)(5)=3$
-23	5	$\lfloor -23/5 \rfloor = -5$	$-23-(-5)(5)=2$
23	-5	$\lfloor 23/-5 \rfloor = -5$	$23-(-5)(-5)=2$
-23	-5	$\lfloor -23/-5 \rfloor = 4$	$-23-(4)(-5)=-3$
20	5	$\lfloor 20/5 \rfloor = 4$	$20-(4)(5)=0$

**定理 11.2** 设  $n$  和  $m$  是整数, 且都不等于 0, 并令

$$d=\min\{in+jm|i, j \in \mathbb{Z}, in+jm>0\}$$

即,  $d$  是  $n$  和  $m$  的最小正线性组合。然后有

$$d=\gcd(n, m)$$

证明: 设  $i$  和  $j$  是生成最小值  $d$  的整数。即  $d=in+jm$ 。此外, 设  $q$  和  $r$  分别是将  $n$  除以  $d$  时的商和余数。根据式 11.1 及  $d$  为整数这一事实,

$$n=qd+r \text{ 及 } 0 \leq r < d$$

于是有

$$\begin{aligned} r &= n - qd \\ &= n - q(in + jm) \\ &= n(1 - qi) + m(-qj) \end{aligned}$$

也就是说,  $r$  是  $n$  和  $m$  的线性组合。由于  $d$  是  $n$  和  $m$  的最小正线性组合, 且  $r < d$ , 所以可得出结论  $r=0$ , 即  $d|n$ 。同理,  $d|m$ 。因此,  $d$  是  $m$  和  $n$  的一个公约数, 也就是说:

$$d \leq \gcd(n, m)$$

由于  $\gcd(n, m)$  可同时整除  $n$  和  $m$ , 且  $d=ln+jm$ , 所以由定理 11.1 可推出  $\gcd(n, m)|d$ 。得出结论:

$$d \geq \gcd(n, m)$$

综合最后两个不等式, 得  $d=\gcd(n, m)$ 。

例 11.7 有  $\gcd(12, 8)=4$ , 且

$$4=3(12)+(-4)8$$

推论 11.1 假定  $n$  和  $m$  是整数, 且都不为 0, 则  $n$  和  $m$  的每个公约数都是  $\gcd(n, m)$  的约数。即, 如果  $h|n$  和  $h|m$ , 则

$$h|\gcd(n, m)$$

证明: 根据前面的定理,  $\gcd(n, m)$  是  $n$  和  $m$  的一个线性组合。现在由定义 11.1 即可得证。

例 11.8 如例 11.5 所示, 24 和 30 的正公约数为:

$$1, 2, 3 \text{ 和 } 6$$

$\gcd(24, 30)$  的值为 6。由前面的推论可知, 1、2、3 和 6 都可以整除 6。

定理 11.3 假定有整数  $n \geq 0$  和  $m > 0$ 。如果令  $r=n \bmod m$ , 则

$$\gcd(n, m)=\gcd(m, r)$$

如果  $n=0$ ,  $r=m$ , 此不等式显然成立。否则, 将证明  $\gcd(n, m)$  和  $\gcd(m, r)$  能相互整除。当且仅当两个正整数相等时, 它们才能相互整除, 其证明留作习题, 由此也就完成了定理的证明。首先证明  $\gcd(n, m)|\gcd(m, r)$ 。如果令  $d'=\gcd(n, m)$ , 则  $d'|n$  和  $d'|m$ 。

此外,

$$r=n-qm$$

其中  $q$  是  $n$  除以  $m$  的商, 这就是说,  $r$  是  $n$  和  $m$  的线性组合。于是, 可由定理 11.1 推出  $d'|r$ 。由于  $d'|m$  和  $d'|r$ , 由推论 11.1 可知:

$$d'|\gcd(m, r)$$

于是完成了这一方向的证明。接下来证明  $\gcd(m, r)|\gcd(n, m)$ 。如果令  $d''=\gcd(m, r)$ , 则有  $d''|m$  和  $d''|r$ 。另外,

$$n=qm+r$$

其中  $q$  是  $n$  除以  $m$  的商, 这就是说,  $n$  是  $m$  和  $r$  的线性组合。于是, 可由定理 11.1 推出  $d''|n$ 。由于  $d''|m$  和  $d''|n$ , 由推论 11.1 可知:

$$d''|\gcd(n, m)$$

于是完成了这一方向的证明。

例 11.9 根据前面的定理,

$$\gcd(64, 24)=\gcd(24, 16)$$

这是因为  $16=64 \bmod 24$ 。可以继续以这种方式来计算  $\gcd(64, 24)$ 。即,

$$\begin{aligned} \gcd(64, 24) &= \gcd(24, 16) \\ &= \gcd(16, 8) \\ &= \gcd(8, 0) \\ &= 8 \end{aligned}$$

### 11.1.3 质因数分解

每个大于 1 的整数都可以写为质数的乘积, 而且这一乘积是唯一的。下面提出能够证明这一断言的理论。

有两个均不为 0 的整数  $n$  和  $h$ , 若  $\gcd(n, h)=1$ , 就说它们互质。

例 11.10 因为  $\gcd(12, 25)=1$ , 所以整数 12 和 25 互质。因为  $\gcd(12, 15)=3$ , 所以整数 12 和 15 不是互质的。

定理 11.4 如果  $h$  和  $m$  互质, 且  $h$  能整除  $nm$ , 则  $h$  能整除  $n$ 。也就是说,  $\gcd(n, m)=1$  且  $h|nm$  可推出  $h|n$ 。

证明: 由定理 11.2 可以推出, 存在整数  $i$  和  $j$ , 满足:

$$ih+jm=1$$

将这个不等式乘以  $n$  可得:

$$(ni)h+j(nm)=n$$

显然,  $h$  可以整除这个不等式左侧第一项, 而且由于  $h|nm$ , 所以  $h$  也可以整除第二项。因此,  $h$  可以整除左侧, 也就是说,  $h$  可以整除  $n$ 。

**例 11.11** 整数 9 和 4 互质,  $9|72$ , 且  $72=18\times 4$ 。由前面的定理可以推出  $9|18$ 。

**推论 11.2** 给定整数  $n$ 、 $m$  和质数  $p$ , 若  $p|nm$ , 则  $p|n$  或  $p|m$  (或同时满足)。

证明: 由定理 11.4 容易得证。

下面的定理称为唯一因式分解定理或算术基本定理。

**定理 11.5** 每个整数  $n>1$  都可以唯一地分解为质数的乘积。也就是说,

$$n = p_1^{k_1} p_2^{k_2} \cdots p_j^{k_j}$$

其中  $p_1 < p_2 < \cdots < p_j$  为质数, 而且  $n$  的这种表示方式是唯一的。整数  $k_i$  称为  $p_i$  在  $n$  中的阶 (order)。

证明: 我们使用归纳法来证明这种表示法的存在。

归纳基础: 有  $2=2^1$ 。

归纳假设: 假定所有满足  $2 \leq m < n$  的整数  $m$  都可以写为质数的乘积。

归纳步骤: 如果  $n$  是质数, 则  $n=n^1$  就是我们需要的表示方式。否则,  $n$  就是合数, 这就意味着存在着整数  $m$ ,  $h>1$ , 使

$$n=mh$$

显然,  $m, h < n$ 。因此, 根据归纳假设,  $m$  和  $h$  都可以分别写为质数的乘积。即

$$\begin{aligned} m &= p_1^{k_1} p_2^{k_2} \cdots p_j^{k_j} \\ h &= q_1^{l_1} q_2^{l_2} \cdots q_i^{l_i} \end{aligned}$$

由于  $n=mh$ , 有

$$n = p_1^{k_1} p_2^{k_2} \cdots p_j^{k_j} q_1^{l_1} q_2^{l_2} \cdots q_i^{l_i}$$

将相等的质数分组, 并根据质数值的递增顺序排列各项, 就可以获得想要的表达方式。这就完成了归纳证明。

根据推论 11.2 可以证明这个乘积是唯一的, 其证明留作习题。

**例 11.12** 有

$$22\ 275=3^4 5^2 11$$

前面的定理表明, 这个不等式右侧的表示是唯一的。

**定理 11.6**  $\gcd(n,m)$  是  $n$  和  $m$  共有质数的乘积, 其中, 乘积中每个质数的幂就是它在  $n$  和  $m$  中较小的阶数。

证明: 此证明留作习题。

**例 11.13** 有  $300=2^2\times 3^1\times 5^2$ ,  $1125=3^2\times 5^3$ 。所以  $\gcd(300, 1125)=2^0\times 3^1\times 5^2=75$ 。

#### 11.1.4 最小公倍数

与最大公约数类似的一个概念是最小公倍数。如果  $n$  和  $m$  都不等于 0, 则  $n$  和  $m$  的最小公倍数 (least common multiple) 就是它们都能整除的最小正整数, 记为  $\text{lcm}(n, m)$ 。

**例 11.14** 因为  $6|18$ ,  $9|18$ , 而且没有它们能够整除的更小正整数, 所以  $\text{lcm}(6, 9)=18$ 。

**定理 11.7**  $\text{lcm}(n, m)$  是  $n$  和  $m$  共有质数的乘积, 其中乘积中每个质数的幂就是它在  $n$  和  $m$  中的较大阶数。

证明: 此证明留作习题。

**例 11.15** 有  $12=2^2\times 3^1$ ,  $45=3^2\times 5^1$ 。所以  $\text{lcm}(12, 45)=2^2\times 3^2\times 5^1=180$ 。

## 11.2 计算最大公约数

定理 11.6 给出了一种计算两个整数最大公约数的直接方法。我们直接求出两个整数的唯一因式分解方式，确定哪些质数是它们共有的，其最大公约数就是这些公共质数的一个乘积，乘积中每个质数的幂就是它在两个整数中的较小阶数。例 11.13 演示了这一计算过程。下面是另一个例子。

例 11.16 有

$$\begin{aligned} 3\ 185\ 325 &= 3^4 5^2 11^2 13^1 \\ 7\ 276\ 500 &= 2^2 3^3 5^3 7^2 11^1 \end{aligned}$$

所以

$$\gcd(3\ 185\ 325, 7\ 276\ 500) = 3^3 5^2 11^1 = 7425$$

上述方法有一个问题，就是无法轻松找到一个整数的唯一因式分解。在上例中，如果我们没有给出因式分解结果，你在分解时可能会遇到一些问题。想象一下，当这些整数的位数是 25 而不是 7 时的难度。事实上，还从来没有人找到一种算法，能够在多项式时间内确定整数的因式分解。下面给出一种计算最大公约数的更高效方法。

### 11.2.1 欧氏算法

定理 11.1 提供了一种为两个整数计算最大公约数的直接方法。例 11.9 演示了这一方法。也就是说，为了找出  $\gcd(n, m)$ ，我们递归应用定理中的等式，直到  $m=0$  为止，随后返回  $n$ 。这个方法称为欧氏算法，因为它是欧基里德在公元前 300 年左右发明的。该算法如下。

**算法 11.1 欧氏算法**

**问题：**计算一个正整数和一个非负整数的最大公约数。

**输入：**一个正整数  $n$  和一个非负整数  $m$ 。

**输出：** $n$  和  $m$  的最大公约数。

```
int gcd(int n, int m)
{
    if (m==0)
        return n;
    else
        return gcd(m, n % m); // C++ 代码为 n % m。
}
```

例 11.17 使用定理 11.1 计算例 11.16 中两个数字的最大公约数。

$$\begin{aligned} \gcd(7\ 276\ 500, 3\ 185\ 325) &= \gcd(3\ 185\ 325, 905\ 850) \\
&= \gcd(905\ 850, 467\ 775) \\
&= \gcd(467\ 775, 438\ 075) \\
&= \gcd(438\ 075, 29\ 700) \\
&= \gcd(29\ 700, 22\ 275) \\
&= \gcd(22\ 275, 7425) \\
&= \gcd(7425, 0) \\
&= 7425 \end{aligned}$$

用欧氏算法计算最大公约数，看起来要容易得多。现在来分析算法 11.1，看它到底有多么容易。我们首先需要一个引理和一个定理。

**引理 11.1** 如果  $n > m \geq 1$ ，而且调用  $\gcd(n, m)$ （在定理 11.1 中）会导致  $k$  次递归调用，其中  $k \geq 1$ ，则

$$n \geq f_{k+2}, \quad m \geq f_{k+1}$$

其中  $f_k$  是斐波那契序列的第  $k$  个数。

**证明：**采用归纳法证明。

**归纳基础：**假定调用  $\gcd(n, m)$  导致 1 次递归调用。由于  $m \geq 1$ ，所以

$$m \geq f_{1+1} = f_2 = 1$$

由于  $n > m$ ，有  $n \geq 2$ ，这意味着：

$$n \geq f_{1+2} = f_3 = 2$$

**归纳假设：**假设在进行  $k-1$  次递归调用时此引理为真。

**归纳步骤：**证明在进行  $k \geq 2$  次递归调用时该引理为真。我们需要证明：

$$n \geq f_{k+2} \quad m \geq f_{k+1}$$

第一次递归调用为  $\gcd(m, n \bmod m)$ 。由于共有  $k$  次递归调用，所以这一调用必然需要  $k-1$  次递归调用。由于  $k \geq 2$ ，所以至少还有一次递归调用，也就是说  $n \bmod m \geq 1$ 。因此，由于  $m > n \bmod m$ ，所以满足归纳假设的条件，这意味着：

$$m \geq f_{k+1} \quad n \bmod m \geq f_k \quad (11.2)$$

这就得到了一个要证明的不等式。为证明另一个不等式，我们有

$$n = qm + n \bmod m \quad (11.3)$$

其中  $q$  是  $n$  除以  $m$  的商。由于  $n > m$ ,  $q \geq 1$ ，所以不等式 11.2 和等式 11.3 可以导出：

$$n \geq m + n \bmod m \geq f_{k+1} + f_k = f_{k+2}$$

证明完成。

**定理 11.8** (Lamé) 对于每个整数  $k \geq 1$ ，若  $n > m \geq 1$  且  $m \leq f_k$  (斐波那契序列中的第  $k$  个数)，则 (定理 11.1 中) 调用  $\gcd(n, m)$  会导致少于  $k$  次递归调用。

**证明：**由前面的引理可以马上得出证明。

接下来分析定理 11.1。回想 9.2 节，在数值算法中，作为输入的数字并不是输入规模。输入规模应当是书写该输入所需要的字符数。如果使用二进制编码，输入规模是对这些数字进行编码所需要的比特数，大约等于该数字的  $\lg$ 。

#### ◆算法 11.1 的分析 最差情况时间复杂度 (欧氏算法)

**基本运算：**计算余数时的一位运算。

**输入规模：**对  $n$  进行编码需要的比特数  $s$ ，对  $m$  进行编码需要的比特数  $t$ 。也就是说，

$$s = \lfloor \lg m \rfloor + 1 \quad t = \lfloor \lg n \rfloor + 1$$

不失一般性，我们将分析  $1 \leq m < n$  的情景。也就是说，如果  $m=n$ ，那就没有递归调用，而当  $m > n$  时，第一次递归调用将是  $\gcd(m, n)$ ，这意味着第一个参数较大。

我们不会计算准确的时间复杂度。而是首先证明递归调用的次数  $\text{Calls}(s, t)$  为  $\Theta(t)$ ，然后再为最差时间复杂度找出一个界限值。

首先证明  $\text{Calls}(s, t)$  属于  $O(t)$ 。假定  $m \geq 2$ ，设  $f_k$  是斐波那契列中的数字，且满足

$$f_{k-1} < m \leq f_k \quad (11.4)$$

在 B.2.1 节的例 B.9 中，我们证明了

$$f_k = \frac{\left[ (1 + \sqrt{5})/2 \right]^k - \left[ (1 - \sqrt{5})/2 \right]^k}{\sqrt{5}} \quad (11.5)$$

$k-1$  和  $k-2$  中有一个数为奇数。不失一般性，假定  $k-1$  为奇数。根据式 11.5 和不等式 11.4，有

$$\frac{[(1+\sqrt{5})/2]^{k-1}}{\sqrt{5}} < m < \frac{[(1+\sqrt{5})/2]^k}{\sqrt{5}} \quad (11.6)$$

根据式 11.4, 可由定理 11.8 推出:

$$\text{Calls}(s, t) < k \quad (11.7)$$

由不等式 11.6 和式 11.7 可以推出

$$\text{Calls}(s, t) \in O(t) \quad (11.8)$$

调用  $\text{gcd}(f_{k+1}, f_k)$  恰好需要  $k-1$  次递归调用的证明留作习题。我们得出结论:

$$W \text{ calls}(s, t) \in \Theta(t)$$

其中  $W \text{ calls}(s, t)$  是在输入规模为  $s, t$  时, 最差情况下的递归调用次数。

对于每个递归调用, 我们计算一个余数。如果使用标准的长除算法 (如 2.6 节中的讨论) 来计算余数, 那计算  $r=n \bmod m$  ( $m < n$ ) 所需要的最差位运算次数的上限为:

$$c[(1+\lg q)\lg m] \quad (11.9)$$

其中  $q$  是  $n$  除以  $m$  的商,  $c$  是一个常量。这一结果的证明留作习题。我们将证明, 如果  $r>0$ , 则对于足够大的输入规模, 计算  $r$  所需要的最差位运算次数的上限为:

$$c[(1+\lg n)\lg m - \lg m \lg r] \quad (11.10)$$

为此, 由于  $q=(n-4)/m$  且  $1 \leq r < m$ , 所以

$$\begin{aligned} 1 + \lg q &= 1 + \lg \left( \frac{n-r}{m} \right) \\ &\leq 1 + \lg \left( \frac{n-r}{r} \right) \\ &\leq 1 + \lg \left( \frac{n}{r} \right) \\ &\leq 1 + \lg n - \lg r \end{aligned}$$

结合最后一个不等式和关系 11.9, 即可确定界限 11.10。根据这一界限, 在所有递归调用中计算所有余数时需要的位运算总数上限为:

$$\begin{aligned} &c[(1+\lg n)\lg m - \lg m \lg r] \\ &+ (1+\lg m)\lg r - \lg r \lg(m \bmod r) \\ &+ (1+\lg r)\lg(m \bmod r) - \cdots \\ &= c[\lg n \lg m + \lg m + \lg r + \lg(m \bmod r) + \cdots] \end{aligned} \quad (11.11)$$

关系 11.8 意味着递归调用次数的上限为  $dt$ , 其中  $d$  为大于 0 的常量。这就意味着界限 11.11 中的项数上限为  $dt$ 。因此, 由于  $n > m > r > m \bmod r > \cdots$  (其中的省略号表示界限 11.11 的其他各项), 所以由界限 11.11 可得出结论:

$$W(s, t) \in O(st)$$

## 11.2.2 欧氏算法的扩展

定理 11.1 要求存在整数  $i$  和  $j$ , 以满足

$$\text{gcd}(n, m) = in + jm$$

了解这些整数对于下一节求解模线性方程的算法非常重要。接下来将修改定理 11.1, 以便也能生成这些整数。在这一版本中, 将  $\text{gcd}$  作为一个变量, 因为这样可以让算法正确性的证明变得更易懂。

### 算法 11.2 欧氏算法 2

问题：计算一个正整数和一个非负整数的最大公约数。

输入：一个正整数  $n$  和一个非负整数  $m$ 。

输出： $n$  和  $m$  的最大公约数  $\text{gcd}$ ，以及整数  $i$  和  $j$ ，它们满足  $\text{gcd}=in+jm$ 。

```
void Euclid (int n, int m, int& gcd, int& i, int& j)
{
    if (m==0){
        gcd=n; i=1; j=0;
    }
    else {
        int i', j';
        Euclid (m, n mod m, gcd', i', j');           // C++代码为 n%m
        gcd=gcd';
        i=j';
        j=i'-[n/m] j';
    }
}
```

算法 11.2 的时间复杂度显然与算法 11.1 相同，所以我们只需要证明它的正确性。在此之前，先来证明一个应用该算法的例子。

**例 11.18** 表 11-1 演示了算法 11.2 在顶级调用如下时的流程：

Euclid(42, 30, gcd, i, j);

顶级调用返回的值为  $\text{gcd}=6$ ,  $i=-2$ ,  $j=3$ 。

表 11-1 当  $n=42$  和  $m=30$  时由算法 11.2 确定的值。顶点调用标为 0，三个递归调用标为 1~3。其中的箭头显示了确定这些值的顺序

调 用	$n$	$m$	$\text{gcd}$	$i$	$j$
	↓	↓			
0	42	30	6	-2	3
1	30	12	6	1	-2
2	12	6	6	0	1
3	6	0	6	1	0
	→	→	↑	↑	↑

接下来证明算法 11.2 是正确的。

**定理 11.9** 算法 11.2 返回的  $i$  和  $j$  值，是满足下式的整数：

$$\text{gcd}(n, m)=in+jm$$

证明：采用归纳法证明。

归纳基础：在上一次归纳调用中  $m=0$ ，这意味着  $\text{gcd}(n, m)=n$ 。由于在该调用中，分别为  $i$  和  $j$  的值指定了 1 和 0，我们有

$$i \times n + j \times m = 1 \times n + 0 \times m = n = \text{gcd}(n, m)$$

归纳假设：假定在第  $k$  次递归调用中，为  $i$  和  $j$  确定的值满足

$$\text{gcd}(n, m)=in+jm$$

然后，针对  $i'$  及  $j'$  由该调用返回的值[第  $k-1$  次递归调用]满足以下条件。

$$\text{gcd}(m, n \bmod m)=i'm+j'n \bmod m$$

归纳步骤：对于第  $k-1$  次调用，有

$$\begin{aligned} in + mj &= j'n + (i' - \lfloor n/m \rfloor j')m \\ &= i'm + j'(n - \lfloor n/m \rfloor m) \\ &= i'm + j'n \bmod m \\ &= \gcd(m, n \bmod m) \\ &= \gcd(n, m) \end{aligned}$$

倒数第二个等式是由归纳假设得出，最后一个等式是由定理 11.3 得出。

注意，算法 11.2 在最后一次递归调用中为  $j$  返回的值可以是任意整数。为简单起见，我们选择 0。选择不同整数会给出一个不同的  $(i, j)$  对，并满足  $\gcd(n, m) = in + jm$ 。

## 11.3 模运算回顾

我们在群论的上下文中推导模运算，所以首先来回顾群论。

### 11.3.1 群论

集合  $S$  上的闭合二元运算 (closed binary operation)  $*$  是一种运算规则，它合并  $S$  中的两个元素，生成  $S$  的另一个元素。我们有以下定义。

**定义 群 (group)**  $G=(S, *)$  包括一个集合  $S$  及该集合上定义的一个闭合二元运算  $*$ ，并满足以下条件。

(1)  $*$  满足结合律。也就是说，对于所有  $a, b, c \in S$ ，有

$$a * (b * c) = (a * b) * c$$

(2)  $S$  中有一个单位元  $e$ 。即，对于每个  $a \in S$ ，有

$$e * a = a * e = a$$

(3) 对于每个  $a \in S$ ，存在一个逆元  $a'$ ，满足

$$a * a' = a' * a = e$$

**例 11.19** 整数  $Z$  和加法运算构成一个群。单位元为 0， $a$  的逆为  $-a$ 。

**例 11.20** 非零实数和乘法运算构成一个群。单位元为 1， $a$  的逆为  $1/a$ 。

**例 11.21** 设  $S=(a, b, e)$ ，并指定

$$\begin{aligned} a * b &= b * a = e \\ a * a &= b \\ b * b &= a \\ a * e &= e * a = a \\ b * e &= e * b = b \\ e * e &= e \end{aligned}$$

$(S, *)$  是一个群，其证明留作习题。

群中的一个元素能否有多个逆？下面的定理表明，其答案是“不能”。

**定理 11.10** 一个群中每个元素的逆是唯一的。

证明：假定  $a'$  和  $a''$  都是  $a$  的逆。则

$$a' * (a * a'') = a' * e = a'$$

但是，

$$a' * (a * a'') = (a' * a) * a'' = e * a'' = a''$$

由上面两个等式可以得出结论  $a'=a''$ 。

**定理 11.11** 如果群中存在元素  $a$  和  $b$ , 使得  $a*b=a$  或者  $b*a=a$ , 则  $b$  是群中的单位元  $e$ 。

证明: 如果  $a*b=a$ , 则

$$a'*(a*b)=a'*a=e$$

其中  $a'$  是  $a$  的逆。但是,

$$a'*(a*b)=(a'*a)*b=e*b=b$$

结合这两个等式可证明这一情景。另一情景同理可证。

上一个定理推导出: 一个群中的单位元是独一无二的。

有一个群  $G=(S, *)$ , 如果对于所有  $a, b \in S$ , 都有

$$a*b=b*a$$

就说这个群是可交换群(或者说阿贝尔群)。如果  $S$  中包含有限个元素, 则说群  $G=(S, *)$  是有限群。例 11.19、例 11.20 和例 11.21 中的群都是可交换群, 但只有例 11.21 中的群是有限群。

### 11.3.2 关于 $n$ 同余

先给出一个定义。

**定义** 设  $m$  和  $k$  为整数,  $n$  是一个正整数。若  $n|(m-k)$ , 则说  $m$  关于模  $n$  与  $k$  同余, 并记作:

$$m \equiv k \pmod{n}$$

**例 11.22** 由于  $5|(33-18)$ , 所以有

$$33 \equiv 18 \pmod{5}$$

回想一下,  $m \pmod{n}$  给出当  $m$  除以  $n$  时的余数。下面的定理表明可以使用 mod 函数发现同余。

**定理 11.12**  $m \equiv k \pmod{n}$  等价于

$$m \pmod{n} = k \pmod{n}$$

证明: 此证明留作习题。

**例 11.23** 已经知道  $33 \equiv 18 \pmod{5}$ 。由前面的定理可知,  $33 \pmod{5} = 18 \pmod{5} = 3$ 。

在 11.7 节开发 RSA 密码系统时会用到下面这个有关同余的定理。

**定理 11.13** 假定  $n_1, n_2, \dots, n_j$  两两互质, 且

$$n = n_1 n_2 \cdots n_j$$

则对于所有整数  $m$  和  $k$ ,

$$m \equiv k \pmod{n}$$

等价于对于  $1 \leq i \leq j$ , 有

$$m \equiv k \pmod{n_i}$$

证明: 假设, 对于  $1 \leq i \leq j$ , 有

$$m \equiv k \pmod{n_i}$$

则存在整数  $h_1, h_2, \dots, h_j$ , 满足

$$m - k = h_1 n_1 = h_2 n_2 = \cdots = h_j n_j$$

以下任务留作习题。使用定理 11.4 证明这一等式, 以及由  $n_1, n_2, \dots, n_j$  两两互质这一事实可推出:

$$h_1 = c n_2 n_3 \cdots n_j$$

其中  $c$  是一个整数。于是有

$$m-k = h_1 n_1 = (c n_2 n_3 \cdots n_j) n_1 = c n$$

由于  $c$  是整数，所以这意味着  $m \equiv k \pmod{n}$ 。

在另一个方向上，若  $m \equiv k \pmod{n}$ ，则存在一个整数  $h$ ，使得

$$m - k = hn$$

因此，

$$m - k = hn_1 n_2 \cdots n_j$$

这意味着，对于每个  $i$ ,  $m - k$  都是  $n_i$  的倍数。证毕。

**例 11.24** 整数 2、5、9 两两互质，且

$$184 \equiv 4 \pmod{2}$$

$$184 \equiv 4 \pmod{5}$$

$$184 \equiv 4 \pmod{9}$$

由于  $2 \times 5 \times 9 = 90$ ，则由前面的定义可知：

$$184 \equiv 4 \pmod{90}$$

接下来说明，关于模  $n$  同余是一个等价关系。

**定理 11.14** 对于任意正整数  $n$ ，关于模  $n$  同余是所有整数集上的一个等价关系。即

(1) ( 反身性 )

$$m \equiv m \pmod{n}$$

(2) ( 对称性 )

$$m \equiv k \pmod{n} \Rightarrow k \equiv m \pmod{n}$$

(3) ( 传递性 )

$$m \equiv k \pmod{n} \quad k \equiv j \pmod{n} \Rightarrow m \equiv j \pmod{n}$$

**证明：**此证明留作习题。

关于模  $n$  与  $m$  同余的所有整数组成的集合称为关于模  $n$  包含  $m$  在内的同余类。由于同余是一种等价关系，所以一个给定类中的任意整数就确定了这个类。对于一个给定整数  $m$ ，它就是如下集合：

$$\{m + in | i \in \mathbb{Z}\}$$

**例 11.25** 关于模 5 包含 13 在内的同余类为：

$$\{\dots, -7, -2, 3, 8, 13, 18, 23, 28, 33, \dots\}$$

我们用  $[m]_n$  表示关于模  $n$  包含  $m$  在内的同余类。所以上面例子中的同余类可表示为  $[13]_5$ 。它也可以表示为  $[3]_5$ ,  $[8]_5$ , 等等。通常，我们使用一个同余类中的最小非负整数来表示这个类。所以，上面例子中的同余类通常表示为  $[3]_5$ 。关于模  $n$  的所有同余类的集合表示为  $\mathbf{Z}_n$ 。也就是说，

$$\mathbf{Z}_n = \{[0]_n, [1]_n, \dots, [n-1]_n\}$$

**例 11.26** 有

$$\mathbf{Z}_5 = \{[0]_5, [1]_5, [2]_5, [3]_5, [4]_5\}$$

对于  $\mathbf{Z}_n$  的两个元素，定义加法运算如下：

$$[m]_n + [k]_n = [m+k]_n$$

为使这一定义有意义，其结果当然不能取决于我们选择  $[m]_n$  和  $[k]_n$  的哪些成员。下面就来证明。我们需要证明，如果

$$s \in [m]_n, t \in [k]_n, \text{ 则 } s+t \in [m+k]_n$$

为此，存在满足以下关系式的整数  $i$  和  $j$ ：

$$s = m + in, \quad t = k + jn$$

所以

$$s+t = m+k+(i+j)n$$

这意味着  $s+t \in [m+k]_n$ 。

例 11.27 有

$$[2]_5 + [4]_5 = [6]_5 = [1]_5$$

定理 11.15 对于每个正整数  $n$ ,  $(\mathbf{Z}_n, +)$  是一个有限可交换群。

证明：由整数+的结合律和交换律可以轻松推导出+的结合律和交换律。单位元 (identity element) 为  $[0]_n$ 。  
 $[m]_n$  的加法逆为  $[n-m]_n = [-m]_n$ , 这是因为

$$[m]_n + [-m]_n = [m + (-m)]_n = [0]_n$$

例 11.28 考虑群  $(\mathbf{Z}_5, +)$ 。回想一下,

$$\mathbf{Z}_5 = \{[0]_5, [1]_5, [2]_5, [3]_5, [4]_5\}$$

$[1]_5$  的加性逆是  $[5-1]_5 = [4]_5$ 。注意：

$$[1]_5 + [4]_5 = [5]_5 = [0]_5$$

同理,  $[2]_5$  的加性逆是  $[3]_5$ 。注意：

$$[2]_5 + [3]_5 = [5]_5 = [0]_5$$

对于  $\mathbf{Z}_n$  的两个元素, 定义乘法如下:

$$[m]_n \times [k]_n = [m \times k]_n$$

为使这一定义有意义, 其结果当然不能取决于我们选择  $[m]_n$  和  $[k]_n$  的哪些成员。我们需要证明, 如果

$$s \in [m]_n, \quad t \in [k]_n, \quad \text{则 } s \times t \in [m \times k]_n$$

其证明留作习题。

例 11.29 有

$$[2]_5 \times [4]_5 = [8]_5 = [3]_5$$

现在  $(\mathbf{Z}_n, \times)$  不再总是一个群了, 因为存在  $n$  (即非质数), 使得并非  $\mathbf{Z}_n$  中的每个元素都有乘法逆。也就是说, 如果设  $[1]_n$  是单位元, 则存在  $[m]_n$ , 使得没有一个  $k$  能满足  $[m]_n \times [k]_n = [1]_n$ 。

例 11.30 考虑

$$\mathbf{Z}_9 = \{[0]_9, [1]_9, [2]_9, [3]_9, [4]_9, [5]_9, [6]_9, [7]_9, [8]_9\}$$

假设  $[6]_9$  有一个乘法逆  $[k]_9$ 。则

$$[6]_9 \times [k]_9 = [6 \times k]_9 = [1]_9$$

这意味着存在一个整数  $i$ , 使得

$$1 = 6k + 9i$$

因此, 定理 11.2 可推出  $\gcd(6, 9)$ , 实际情况显然不是如此。

前面例子中的问题在于 6 和 9 不是互质的。看起来, 如果我们只包含与  $n$  互质的数字, 那就会拥有一个群。下面的定理证明了这一点。首先给出这个集合的表示符号。设

$$\mathbf{Z}_n^* = \{\text{所有 } [m]_n \in \mathbf{Z}_n | [m]_n \text{ 中的每个成员都与 } n \text{ 互质}\}$$

显然, 如果  $n$  是质数, 则  $\mathbf{Z}_n^* = \mathbf{Z}_n - \{[0]_n\}$ 。不难看出,  $[m]_n$  的一个成员与  $n$  互质就等价于其所有成员都与  $n$  互质。只需要查看前  $n-1$  个整数, 就能确定  $\mathbf{Z}_n^*$  的成员。

例 11.31 有

$$\mathbf{Z}_9^* = \{[1]_9, [2]_9, [4]_9, [5]_9, [7]_9, [8]_9, \}$$

定理 11.16 对于每个正整数  $n$ ,  $(\mathbf{Z}_n^*, \times)$  是一个有限可交换群。

证明: 根据推论 11.2,  $m \times k$  的任意约数都或是  $m$  的约数, 或是  $k$  的约数。因此, 如果  $m$  和  $k$  分别与  $n$  互质, 则  $m \times k$  也与  $n$  互质, 这就意味着  $\times$  是  $\mathbf{Z}_n^*$  上的闭合二元运算。由整数  $\times$  的结合律和交换律很容易就可以推出  $\times$  的结合律和交换律。单位元是  $[1]_n$ 。如果  $m$  与  $n$  互质, 则由定理 11.2 可知, 存在整数  $i$  和  $j$ , 使得

$$1 = in + jm$$

这意味着:

$$jm \equiv 1 \pmod{n}$$

所以  $[j]_n$  是  $[m]_n$  的逆。

例 11.32 对于群  $(\mathbf{Z}_9^*, \times)$ , 有以下乘法逆:

$$\begin{aligned} [1]_9 \times [1]_9 &= [1]_9 \\ [2]_9 \times [5]_9 &= [10]_9 = [1]_9 \\ [4]_9 \times [7]_9 &= [28]_9 = [1]_9 \\ [8]_9 \times [8]_9 &= [64]_9 = [1]_9 \end{aligned}$$

定理 11.17  $\mathbf{Z}_n^*$  中的元素个数由欧拉  $\varphi$  函数给出, 该函数为:

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

其中, 该乘积是所有可整除  $n$  的质数之积, 如果  $n$  为质数, 则包含  $n$  在内。

证明: 在 Graham 等人 (1989 年) 的文献中可找到这一证明。

注意, 如果  $p$  为质数, 则

$$\varphi(p) = p \left(1 - \frac{1}{p}\right) = p - 1$$

例 11.33  $\mathbf{Z}_9^*$  中的元素数为:

$$\varphi(9) = 9 \prod_{p|9} \left(1 - \frac{1}{p}\right) = 9 \left(1 - \frac{1}{3}\right) = 6$$

在例 11.31 中计数就可以验证这一结果。

例 11.34  $\mathbf{Z}_{60}^*$  中的元素数为:

$$\varphi(60) = 60 \prod_{p|60} \left(1 - \frac{1}{p}\right) = 60 \left(1 - \frac{1}{2}\right) \left(1 - \frac{1}{3}\right) \left(1 - \frac{1}{5}\right) = 16$$

例 11.35 由于 7 为质数，所以  $\mathbf{Z}_7^*$  中的元素数为：

$$\varphi(7) = 7 - 1 = 6$$

### 11.3.3 子群

如果  $G=(S, *)$  是一个群， $S' \subseteq S$ ，且  $G'=(S', *)$  是一个群，就说  $G'$  是  $G$  的一个子群 (subgroup)。如果  $S' \neq S$ ，就说它是一个真子群 (proper subgroup)。

例 11.36 对于偶整数集合  $E$  和整数集合  $Z$ ,  $(E, +)$  是  $(Z, +)$  的一个真子群。

在继续讨论之前，先给出一些有用的符号。给出一个群  $G=(S, *)$ ，且  $a \in S$  具有逆元素  $a'$ ，则对于  $k > 0$ ，有如下定义：

$$\begin{aligned} a^k &= a * a * \cdots * a \quad (k \text{ 次}) \\ a^0 &= 1 \\ a^{-k} &= (a')^k \end{aligned}$$

假设有一个群  $G=(S, *)$  和一个子集  $S' \subseteq S$ ，使得对于所有  $a, b \in S'$ ，有  $a * b \in S'$ 。就说  $S'$  关于  $*$  是闭合的。

定理 11.18 假定我们有一个有限群  $G=(S, *)$ ，一个非空子集  $S' \subseteq S$ ，它关于  $*$  闭合的。则  $(S', *)$  是  $G$  的子群。

证明：显然，结合律成立。设  $a \in S'$ 。下面的证明留作习题：由于  $G$  是有限的，则存在整数  $k, m \geq 1$ ，使得：

$$a^k = a^m a^m$$

在此等式的两边乘以  $a^k$  的逆，得：

$$e = a^m$$

其中  $e$  是单位元。由于  $S'$  是闭合的，所以这意味着  $e$  属于  $S'$ 。接下来将证明，每个  $a \in S'$  的逆都属于  $S'$ 。上面已经给出，对于所有这种  $a$ ，则存在一个  $m$ ，使得  $e = a^m$ 。如果  $m=1$ ，则  $a=e$ ，这意味着  $a$  的逆是  $e$ ，我们已经证明  $e$  属于  $S'$ 。否则，

$$e = a^m = a * a^{m-1}$$

这意味着  $a^{m-1}$  是  $a$  的唯一逆。但是，由于  $S'$  是闭合的，所以  $a^{m-1}$  属于  $S'$ 。证毕。

定理 11.19 (拉格朗日) 假定有一个有限群  $G=(S, *)$  和  $G$  的一个子群  $G'=(S', *)$ 。则

$$|S'| \mid |S|$$

其中， $|S|$  表示  $S$  中的元素个数。

证明：在 Jacobson (1951 年) 的文献中可以找到其证明。

例 11.37 考虑群  $(\mathbf{Z}_{12}, +)$ 。下面的证明留作习题：若

$$S' = \{[0]_{12}, [3]_{12}, [6]_{12}, [9]_{12}\}$$

则  $(S', +)$  是  $(\mathbf{Z}_{12}, +)$  的子群。由前面的定理可知， $|S'|=4$ ， $|\mathbf{Z}_{12}|=12$  及  $4 \mid 12$ 。

推论 11.3 如果  $G'=(S', *)$  是  $G=(S, *)$  的真子群，则

$$|S'| \leq |S|/2$$

证明：由前面的定理可以直接得出证明。

假定有一个有限群  $G=(S, *)$ ，且  $a \in S$ 。设

$$\langle a \rangle = \{a^k, \text{ 其中 } k \text{ 为一个正整数}\}$$

显然， $\langle a \rangle$  关于 $*$ 是闭合的。所以，由定理 11.18 可知，( $\langle a \rangle, *$ ) 是  $G$  的一个子群。这个群称为由  $a$  生成的子群。如果由  $a$  生成的子群为  $G$ ，就称  $a$  是  $G$  的一个生成元 (generator)。

例 11.38 考虑群  $(\mathbf{Z}_6, +)$ 。我们有

$$\langle [2]_6 \rangle = \{[2]_6, [2]_6+[2]_6, [2]_6+[2]_6+[2]_6, [2]_6+[2]_6+[2]_6+[2]_6, \dots\} = \{[2]_6, [4]_6, [0]_6, [2]_6, \dots\}$$

上面的例子表明，一旦在生成子群里获得了单位元，就可以停止了，因为它只是重复已经生成的项目。下一个定理将获得这一结果。首先我们需要一个定义。给定一个群，群元素  $a$  的阶就是满足  $a^t=e$  的最小正整数  $t$ ，记为  $\text{ord}(a)$ ，其中  $e$  是单位元。

定理 11.20 假定我们有一个有限群  $G=(S, *)$ ，且  $a \in S$ 。设  $t=\text{ord}(a)$ 。则由  $a$  生成的子群  $\langle a \rangle$  包含以下  $t$  个不同元素：

$$a^1, a^2, \dots, a^{t-1}$$

证明：首先证明这些元素是不同的。假定，对于  $1 \leq k < j \leq t$ ，则有

$$a^j=a^k$$

由于  $j>k$ ，所以

$$a^j=a^k a^{j-k}$$

其中  $i \geq 1$ 。根据后面这两个等式和定理 11.11，得到  $a^i=e$ 。由于  $i<t$ ，所以得到了一个矛盾。接下来证明，在这个子群中没有其他元素。如果  $k>t$ ，则存在正整数  $q$  和  $r$ ，使得

$$k=qt+r \quad (0 \leq r < t)$$

于是有

$$a^k=a^{(qt+r)}=(a^q)^r a^r=e^q a^r=a^r$$

若  $r=0$ ，则  $a^r=e=a^t$ ；否则， $1 \leq r < t$ 。证毕。

由于前面的定理，可以得到：

$$\langle a \rangle = \{a^0, a^1, \dots, a^{t-1}\}$$

其中  $t=\text{ord}(a)$ 。注意， $\langle a \rangle$  中的元素数就是  $\text{ord}(a)$ 。

例 11.39 考虑群  $(\mathbf{Z}_6, +)$ 。有

$$\langle [3]_6 \rangle = \{[3]_6, [3]_6+[3]_6\} = \{[3]_6, [0]_6\}$$

且

$$\langle [2]_6 \rangle = \{[2]_6, [2]_6+[2]_6, [2]_6+[2]_6+[2]_6\} = \{[2]_6, [4]_6, [0]_6\}$$

显然，

$$\langle [1]_6 \rangle = \mathbf{Z}_6$$

所以  $[1]_6$  是  $\mathbf{Z}_6$  的生成元。

例 11.40 考虑群  $(\mathbf{Z}_9^*, \times)$ 。回想一下，

$$\mathbf{Z}_9^* = \{[1]_9, [2]_9, [4]_9, [5]_9, [7]_9, [8]_9\}$$

我们有

$$\langle [4]_9 \rangle = \{[4]_9, [4]_9 \times [4]_9, [4]_9 \times [4]_9 \times [4]_9\} = \{[4]_9, [7]_9, [1]_9\}$$

下面结果的证明留作习题：

$$\langle [2]_9 \rangle = \mathbf{Z}_9^*$$

所以  $[2]_9$  是  $\mathbf{Z}_9^*$  的一个生成元。

**例 11.41** 考虑群  $(\mathbf{Z}_7^*, \times)$ 。回想一下，

$$\mathbf{Z}_7^* = \{[1]_7, [2]_7, [3]_7, [4]_7, [5]_7, [6]_7\}$$

我们有

$$\langle [4]_7 \rangle = \{[4]_7, [4]_7 \times [4]_7, [4]_7 \times [4]_7 \times [4]_7\} = \{[4]_7, [2]_7, [1]_7\}$$

**推论 11.4** 假定有一个有限群  $G=(S, *)$  及  $a \in S$ 。设  $t=\text{ord}(a)$ 。考虑以下序列

$$a^0, a^1, \dots$$

当且仅当  $i \equiv j \pmod t$  时，这一序列中的两个元素  $a^i$  和  $a^j$  相等。

这就是说，此序列是循环的，周期为  $t$ ，且  $t$  是该序列的最小周期。也就是说，对于  $0 \leq k < t$  及所有  $i \geq 0$ ，

$$a^{k+it} = a^k$$

且  $t$  是具有这一性质的最小数。

**证明：**由前面的定义即可给出证明，具体过程留作习题。

**例 11.42** 考虑群  $(\mathbf{Z}_5^*, \times)$ 。下面的表给出  $[4]_5$  的幂：

$i$	0	1	2	3	4	5	6	7	8	...
$([4]_5)^i$	$[1]_5$	$[4]_5$	$[1]_5$	$[4]_5$	$[1]_5$	$[4]_5$	$[1]_5$	$[4]_5$	$[1]_5$	...

而下表给出  $[3]_5$  的幂：

$i$	0	1	2	3	4	5	6	7	8	...
$([3]_5)^i$	$[1]_5$	$[3]_5$	$[4]_5$	$[2]_5$	$[1]_5$	$[3]_5$	$[4]_5$	$[2]_5$	$[1]_5$	...

在这个群中， $\text{ord}([4]_5)=2$ ， $\text{ord}([3]_5)=4$ 。注意，根据推论 11.4 可以知道这些幂相对于这些阶的循环特性。进一步注意到  $[3]_5$  是  $\mathbf{Z}_5^*$  的生成元。

**推论 11.5** 假定有一个有限群  $G=(S, *)$ ，其单位元为  $e$ 。则对于所有  $a \in S$ ，有

$$a^{|S|}=e$$

其中  $|S|$  表示  $S$  中的元素个数。

**证明：**由定理 11.20 可知，由  $a$  生成的子群  $\langle a \rangle$  中的元素个数为  $\text{ord}(a)$ 。因此，由定理 11.19 可知，对于某一整数  $k$ ， $|S|=\text{ord}(a) \times k$ ，因此，

$$a^{|S|}=a^{\text{ord}(a) \times k}=e^k=e$$

证毕。

**定理 11.21** (欧拉) 对于某个整数  $n>1$ ，对于所有  $[m]_n \in \mathbf{Z}_n^*$ ，有

$$([m]_n)^{\varphi(n)}=[1]_n$$

**证明：**由定理 11.17 和推论 11.5 马上可以得证。

**例 11.43** 考虑  $(\mathbf{Z}_{20}^*, \times)$ 。我们有

$$\varphi(20)=20 \prod_{p:p|20} \left(1 - \frac{1}{p}\right) = 20 \left(1 - \frac{1}{5}\right) \left(1 - \frac{1}{2}\right) = 8$$

及

$$([3]_{20})^8 = [6561]_{20} = [1]_{20}$$

由前一定理可推出这一结果。

**定理 11.22** (费马) 若  $p$  为质数, 则对于所有  $[m]_p \in \mathbf{Z}_p^*$ , 有

$$([m]_p)^{p-1} = [1]_p$$

证明: 根据前面的定理, 再结合事实若  $p$  为质数, 则  $\varphi(p)=p-1$ , 即可证明上述结果。

**例 11.44** 考虑  $(\mathbf{Z}_7^*, \times)$ 。由前面的定理可得:

$$([2]_7)^{7-1} = [64]_7 = [1]_7$$

## ◆ 11.4 模线性方程的求解

接下来讨论如何求解下面有关  $x$  的模方程:

$$[m]_n x = [k]_n \quad (11.12)$$

其中,  $x$  是关于模  $n$  的等价类, 且  $m, n > 0$ 。在 11.7 节开发 RSA 密码系统时将应用这一结果。

设  $\langle [m]_n \rangle$  是由  $[m]_n$  生成的关于群  $(\mathbf{Z}_n^+, +)$  的子群。由于  $\langle [m]_n \rangle = \{[0]_n, [m]_n, [2m]_n, \dots\}$ , 所以当且仅当  $[k]_n \in \langle [m]_n \rangle$  时, 式 11.12 有一个解。

**例 11.45** 考虑群  $(\mathbf{Z}_8, +)$ 。由于

$$\langle [6]_8 \rangle = \{[0]_8, [6]_8, [4]_8, [2]_8\}$$

所以当且仅当  $[k]_8$  为  $[0]_8, [6]_8, [4]_8$  或  $[2]_8$  时, 以下方程有解:

$$[6]_8 x = [k]_8$$

例如,

$$[6]_8 x = [4]_8$$

的解是  $x = [2]_8$ ,  $x = [6]_8$ 。

下面的定理将集合  $\langle [m]_n \rangle$  的元素准确地告诉我们。

**定理 11.23** 考虑群  $(\mathbf{Z}_n, +)$ 。对于任意  $[m]_n \in \mathbf{Z}_n$ , 我们有

$$\langle [m]_n \rangle = \langle [d]_n \rangle = \{[0]_n, [d]_n, [2d]_n, \dots, [(n/d-1)d]_n\}$$

其中  $d = \gcd(n, m)$ 。这就意味着:

$$\text{ord}([m]_n) = |\langle [m]_n \rangle| = \frac{n}{d}$$

证明: 首先证明  $\langle [d]_n \rangle \subseteq \langle [m]_n \rangle$ 。由定理 11.2 可知, 必然存在整数  $i$  和  $j$ , 满足

$$d = im + jn$$

根据前面的等式可知, 对于任意整数  $k$ , 有

$$kd \equiv k(im + jn) \pmod{n}$$

其中  $[kd]_n = [kim]_n$ , 因此,  $[kd]_n \in \langle [m]_n \rangle$ , 从而得出结论:  $\langle [d]_n \rangle \subseteq \langle [m]_n \rangle$ 。

接下来证明:  $\langle [m]_n \rangle \subseteq \langle [d]_n \rangle$ 。由于  $d \mid m$ , 所以存在一个整数  $i$ , 使得  $m = id$ 。因此, 对于任意整数  $k$ , 有

$$[km]_n = [kid]_n$$

这意味着  $[km]_n \in \langle [d]_n \rangle$ 。我们得出结论:  $\langle [m]_n \rangle \subseteq \langle [d]_n \rangle$ 。

根据刚刚推出的结果及定理 11.20, 可以证明最终结果。

**推论 11.6** 当且仅当  $d \mid k$  时 (其中  $d = \gcd(n, m)$ ), 方程  $[m]_n x = [k]_n$  有解。

此外，如果这个方程有一个解，那它就有  $d$  个解。

证明：在本节开头曾经提过，当且仅当  $[k]_n \in \langle [m]_n \rangle$  时，此方程有一个解。由于根据定理 11.23 可以推出

$$\langle [m]_n \rangle = \langle [d]_n \rangle = \{[0]_n, [d]_n, [2d]_n, \dots, [(n/d-1)d]_n\}$$

这意味着，当且仅当对于某个  $k' \in [k]_n$  满足  $d|k'$  时，该方程有一个解。由于  $[k]_n$  是关于模  $n$  的同余类，且  $d|n$ ，显然，若对于  $k' \in [k]_n$  有  $d|k'$ ，则等价于对于  $[k]_n$  中的所有成员都是如此。这就证明了推论的第一部分。

至于第二部分，根据定理 11.23， $\text{ord}([m]_n) = n/d$ 。因此，根据推论 11.4，序列

$$[0]_n, [m]_n, [2m]_n, \dots$$

的周期为  $n/d$ ，且前  $n/d$  项互不相同。这就意味着，如果  $[k]_n \in \langle [m]_n \rangle$ ，则  $[k]_n$  在以下集合中恰好出现  $d$  次：

$$\{[0]_n, [m]_n, [2m]_n, \dots, [(n-1)m]_n\}$$

显然，每一次出现都是因为  $\mathbf{Z}_n$  的一个不同成员所导致的。证毕。

**推论 11.7** 已知当且仅当  $\gcd(n, m)=1$  时，方程  $[m]_n x = [k]_n$  对于每个同余类  $[k]_n$  有一个解。此外，如果确实  $\gcd(n, m)=1$ ，则每个  $[k]_n$  都有一个唯一解。

证明：由推论 11.6 马上可以得出证明。

**推论 11.8** 当且仅当  $\gcd(n, m)=1$  时，同余类  $[m]_n$  拥有一个以  $n$  为模的乘法逆。也就是说，当且仅当  $\gcd(n, m)=1$  时，方程  $[m]_n x = [1]_n$  有一个解。此外，如果有多个解，则这个解是唯一的。

证明：由推论 11.6 马上可以得到证明。唯一性实际上也可以由定理 11.16 推出。

**例 11.46** 考虑群  $(\mathbf{Z}_8, +)$ 。由于  $\gcd(8, 6)=2$ ，根据定理 11.23，有

$$\langle [6]_8 \rangle = \{[0]_8, [2]_8, [4]_8, [6]_8\}$$

这与例 11.45 中的结果一致。根据推论 11.6，当  $[k]_8$  是  $\langle [6]_8 \rangle$  的任意成员时，方程  $[6]_8 x = [k]_8$  恰有两个解。在例 11.45 中，我们注意到当  $[k]_8 = [4]_8$  时，两个解为  $x = [2]_8, x = [6]_8$ 。

**例 11.47** 再次考虑群  $(\mathbf{Z}_8, +)$ 。由于  $\gcd(8, 5)=1$ ，根据定理 11.23，有

$$\langle [5]_8 \rangle = \{[0]_8, [1]_8, [2]_8, [3]_8, [4]_8, [5]_8, [6]_8, [7]_8\}$$

根据推论 11.7，当  $[k]_8$  为  $\langle [5]_8 \rangle$  的任意成员时，方程  $[5]_8 x = [k]_8$  恰有一个解。例如，当  $[k]_8 = [3]_8$ ，这个解为  $x = [7]_8$ 。

**例 11.48** 考虑  $\mathbf{Z}_9$ 。由于  $\gcd(9, 6)=3$ ，由推论 11.8 可知， $[6]_9$  不存在以 9 为模的乘法逆。由于  $\gcd(9, 5)=1$ ，由推论 11.8 可知， $[5]_9$  没有以 9 为模的乘法逆。该逆为  $[2]_9$ 。

**定理 11.24** 设  $d=\gcd(n, m)$ ，并设  $i$  和  $j$  为满足以下条件的整数：

$$d=ni+jm \quad (11.13)$$

(由定理 11.2 可知，这种  $i$  和  $j$  是存在的。) 进一步假定  $d|k$ 。于是，方程  $[m]_n x = [k]_n$  有以下解： $x = \left[ \frac{jk}{d} \right]_n$ 。

证明：根据式 11.13，有

$$[jm]_n = [d]_n$$

由于  $\frac{k}{d}$  是一个整数，可以将这一等式的两侧乘以  $\left[ \frac{k}{d} \right]_n$ ，得出：

$$\left[ m \frac{jk}{d} \right]_n = \left[ dk \right]_n$$

这就意味着：

$$[m]_n \left[ j \frac{k}{d} \right]_n = [k]_n$$

这就证明了本定理。

#### 例 11.49 考虑方程

$$[6]_8x=[4]_8$$

我们有  $\gcd(8, 6)=2$ ,  $2=(1)8+(-1)6$  及  $2|4$ 。因此, 前面的定理可以推出, 方程  $[6]_8x=[4]_8$  有以下解:

$$x=\left[\frac{(-1)4}{2}\right]_8=[-2]_8=[6]_8$$

**定理 11.25** 假定方程  $[m]_nx=[k]_n$  是可解的,  $x=[j]_n$  是一个解, 且  $d=\gcd(n, m)$ 。因此, 这个方程的  $d$  个不同解为:

$$\left[j+\frac{\ln}{d}\right]_n \quad (l=0, 1, \dots, d-1) \quad (11.14)$$

证明: 根据推论 11.6 可知, 该方程恰有  $d$  个解。显然,

$$0 \leq \frac{\ln}{d} < n \quad (l=0, 1, \dots, d-1)$$

因此, 式 11.14 中的  $d$  个同余类都是互不相同的。我们将证明其中每个类都是方程的一个解, 以此来完成此定理。由于  $[j]_n$  是  $[m]_nx=[k]_n$  的一个解,

$$[mj]_n=[k]_n$$

因此, 对于  $l=0, 1, \dots, d-1$ ,

$$\left[m\left(j+\frac{\ln}{d}\right)\right]_n = \left[mj + \left(\frac{ml}{d}\right)n\right]_n = [mj]_n + [0]_n = [k]_n$$

这意味着  $\left[j+\frac{\ln}{d}\right]_n$  也是该方程的一个解。

**例 11.50** 在例 11.49 中, 证明了  $[6]_8x=[4]_8$  的一个解是  $[6]_8$ 。由于  $\gcd(8, 6)=2$ , 所以由上一定理可知, 其他解为:

$$\left[6+\frac{1(8)}{2}\right]_8=[10]_8=[2]_8$$

利用推论 11.6、定理 11.24 和定理 11.25, 可以编写一个简单的算法来求解模线性方程。也就是说, 首先利用推论 11.6 来研究是否存在解。如果存在, 则利用定理 11.24 来求出一个解, 利用定理 11.25 来求出其他解。算法如下。

#### 算法 11.3 求解模线性方程

问题: 求出一个模线性方程的所有解。

输入: 正整数  $m$  和  $n$ ; 整数  $k$ 。

输出: 如果方程  $[m]_nx=[k]_n$  是可解的, 则给出它的所有解。

```
void solve_linear (int n, int m, int k)
{
    index l;
    int i, j, d;
    Euclid(n, m, d, i, j);           // 调用算法 11.2
    if (d|k)
        for (l=0; l<=d-1; l++)
            cout << left < [jk + ln] / d < right;
}
```

算法 11.3 的输入规模是对输入进行编码所需要的比特数，可由下面的式子给出：

$$s = \lfloor \lg n \rfloor + 1 \quad t = \lfloor \lg m \rfloor + 1 \quad u = \lfloor \lg k \rfloor + 1$$

时间复杂度包含了算法 11.2( 欧氏算法 2 )的时间复杂度再加上 `for-l` 循环的时间复杂度，已知前者为  $O(st)$ 。由于  $d$  可能与  $m$  或  $n$  一样大，所以这一时间复杂度在最差情况下可能是输入规模的指数函数。但是，我们对此无能为力，因为这一问题在最差情况下需要计算和显示指数个数的值。

## ◆ 11.5 计算模的幂

对于一个元素  $[m]_n \in \mathbf{Z}_n$  及非负整数  $k$ ，将计算  $([m]_n)^k$  的问题称为计算模的幂的问题。例 11.43 和例 11.44 展示了一些模的幂的计算。下面给出另一个例子。

**例 11.51** 考虑  $\mathbf{Z}_{20}$ 。我们有

$$([7]_{20})^{11} = [1\ 977\ 326\ 743]_{20} = [3]_{20}$$

在例 11.51 中，是通过计算 7 的第 11 次幂来确定模的幂。显然，此方法的时间复杂度是输入规模的指数函数（输入规模近似为数字的对数）。接下来将开发一种更高效的算法。

此算法要求将  $k$  表示为一个二进制数。设

$$\{b_j, b_{j-1}, \dots, b_1, b_0\}$$

是该表示法中二进制数位的有序集合。例如，由于 13 的二进制表示为 1101，所以如果  $k=13$ ，

$$\{b_3, b_2, b_1, b_0\} = \{1, 1, 0, 1\}$$

现在给出算法。我们说，这一算法使用了重复平方（repeated squaring）的方法。原因是显而易见的。

### 算法 11.4 计算模的幂

**问题：**对于一个元素  $[m]_n \in \mathbf{Z}_n$  及非负整数  $k$ ，计算  $([m]_n)^k$ 。

**输入：**正整数  $n$ ，非负整数  $m$  和  $k$ 。

**输出：**  $([m]_n)^k$ 。

```
void compute_power(int n, int m, int k, Znmember& a)
{
    index i;

    a=[1]_n;
    {b_j, b_{j-1}, ..., b_1, b_0}=k 的二进制表示中数位的有序集合;

    for (i = j; i >= 0; i--)
        a=a^2;
        if (b_i==1)
            a=a*x[m]_n;
}
```

下面给出一个应用算法 11.4 的例子。

**例 11.52** 假设

$$n=257 \quad m=5 \quad k=45$$

由于 45 的二进制表示为 101101，所以

$$\{b_5, b_4, b_3, b_2, b_1, b_0\} = \{1, 0, 1, 1, 0, 1\}$$

表 11-2 是在给定这一输入后，对算法 11.4 中的 `for-i` 循环进行每次迭代之后， $a$  的状态。 $a$  的最终值，即  $[147]_{257}$ ，是  $([5]_{257})^{45}$  的值。

表 11-2 当  $n=257$ ,  $m=5$ ,  $k=45$  时, 对算法 11.4 中的 for- $i$  循环进行每次迭代之后,  $a$  的状态。  
第三行给出了由  $k$  的二进制表示中的前  $j-i+1$  位确定的值

$i$	5	4	3	2	1	0
$b_i$	1	0	1	1	0	1
$k_i$	1	2	5	11	22	45
$a$	$[5]_{257}$	$[25]_{257}$	$[41]_{257}$	$[181]_{257}$	$[122]_{257}$	$[147]_{257}$

在表 11-2 中,  $k_i$  是由  $k$  的二进制表示中前  $j-i+1$  位确定的值, 也就是说

$$k_i \text{ 为 } [b_j b_{j-1} \cdots b_i]_2$$

例如,

$$k_3 = [b_5 b_4 b_3]_2 = 101_2 = 5_{10}$$

显然,  $k_0=k$ 。在下面证明这一算法的正确性时, 我们将使用这些变量。

**定理 11.26** 在每次迭代算法 11.4 的 for- $i$  循环之后,

$$a = ([m]_n)^{k_i}$$

由于  $k_0=k$ , 所以这意味着  $a$  的最终值为  $([m]_n)^k$ 。

**证明:** 采用归纳法证明。

**归纳基础:** 假定采用最小二进制表示, 所以高位  $b_j$  等于 1。于是,

$$k_j = 1$$

在进入 for- $i$  循环之前,  $a=[1]_n$ 。由于  $b_j$  等于 1, 所以第一次迭代中的 if 条件为真, 这意味着要执行指令  $a=a \times [m]_n$ 。于是, 在第一次迭代之后, 有

$$a = ([1]_n)^2 \times [m]_n = ([m]_n)^1 = ([m]_n)^{k_j}$$

**归纳假设:** 假定在执行完索引值为  $i$  的迭代后, 有

$$a = ([m]_n)^{k_i}$$

**归纳步骤:** 需要证明, 在执行完索引值为  $i-1$  的迭代后, 有

$$a = ([m]_n)^{k_{i-1}}$$

有两种情况: 若  $b_{i-1}=0$ , 显然,

$$k_{i-1} = 2k_i$$

由于 if 语句中的条件为假, 所以只有指令  $a=a^2$  改变了  $a$  的值。因为根据归纳假设,  $a$  的前一个值为  $([m]_n)^{k_i}$ , 所以在进行这一迭代后, 有

$$a = ([m]_n)^{2k_i} = ([m]_n)^{k_{i-1}}$$

若  $b_{i-1}=1$ , 显然,

$$k_{i-1} = 2k_i + 1$$

由于 if 语句中的条件为真, 所以还会执行  $a=a \times [m]_n$ 。所以在这一情况下, 执行这一迭代之后, 有

$$a = ([m]_n)^{2k_i} \times [m]_n = ([m]_n)^{2k_i + 1} = ([m]_n)^{k_{i-1}}$$

如果设  $s$  是对输入进行编码所需要的位数, 容易看出, 算法 11.4 的时间复杂度属于  $O(s^3)$ 。

## 11.6 寻找大质数

寻找大的质数，对于 RSA 公钥密码系统（将在 11.7 节讨论）的成功是不可或缺的。在讨论了大质数的查找之后，我们会给出一种判断一个数字是否为质数的算法。

### 11.6.1 寻找大质数

为找出一个大质数，我们随机选择适当大小的整数，然后判断每个选定整数是否为质数，如此重复，直到找出一个质数。在使用这一方法时，一个重要的考虑因素就是在随机选择一个整数时，能够找到质数的可能性。

下面将要给出的质数分布定理可以让我们近似估算这一概率。

质数分布函数  $\pi(n)$  是小于或等于  $n$  的质数个数。例如， $\pi(12)=5$ ，这是因为有五个质数小于或等于 12，即 2, 3, 5, 7, 11。定理 11.27 给出的质数定理给出了  $\pi(n)$  的近似值。

**定理 11.27** 有

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n / \ln n} = 1$$

证明：此证明可在 Hardy 和 Wright (1960 年) 的文献中找到。

根据上面的定理，对于大的  $n$  值，可以将  $n / \ln n$  作为小于或等于  $n$  的质数数目的估计值。因此，如果根据均匀分布随机选择介于 1 到  $n$  之间的一个整数，它是质数的概率大约为：

$$\frac{n / \ln n}{n} = \frac{1}{\ln n}$$

**例 11.53** 如果根据均匀分布随机选择介于 1 到  $n=10^{16}$  之间的一个整数，则它是质数的概率大约为：

$$\frac{1}{\ln 10^{16}} = 0.027143$$

假定我们随机选择 200 个这样的数字。于是，它们都不是质数的概率为：

$$(1 - 0.027143)^{200} = 0.004$$

**例 11.54** 如果根据均匀分布随机选择介于 1 到  $n=10^{100}$  之间的一个整数，则它是质数的概率大约为：

$$\frac{1}{\ln 10^{100}} = 0.0043429$$

假定我们随机选择 200 个这样的数字。于是，它们都不是质数的概率为：

$$(1 - 0.0043429)^{200} = 0.04$$

在本节开头曾经指出，为找到一个大的质数，我们在适当位置中随机选择数字，然后检查它是不是质数，重复此过程，直到发现其中一个是质数为止。从前面的例子可以知道，找出一个质数应当不会花费太长的时间（相对于该范围内的数字大小而言）。接下来讨论如何检查一个数字是否为质数。

### ◆ 11.6.2 检查一个数字是否为质数

在 9.2 节的开头，我们已经给出了一种用于判断一个数字是否为质数的简单算法。但是，该节也讨论到了，这个算法在最差情况下具有指数时间复杂度，所以它不能用于检查大的数字。注意，这个低效算法会在数字为合数时指出它的因数，所以，可以（重复）使用该算法来完成一个数字的因数分解。

长期以来，一直没人能为质数检查问题找到一种多项式时间算法，许多人认为它是 NP 完全的。进行质数检查的标准高效算法是一种蒙特卡洛算法，称为 Miller-Rabin 随机化质数检测，它出现在 Rabin (1980 年) 的文献中。如果一个数字为质数，这个算法会说该数字一直是质数。但是，在非常罕见的情况下，这个算法会将一个合数说成质数。这种情况非常罕见，所以这种算法的准确性基本上还是可信赖的。Miller-Rabin 算法在判

定一个数字为合数时，并不会给出该数字的因数，所以不能用于因数分解。

最后，在 2002 年，Agrawal 等人成功地为质数检查问题开发了一种多项式时间算法。现在我们就来介绍这种算法。

### 1. 多项式时间算法

首先，我们需要数论中的其他一些结果。

**定义** 设  $f(x)$  和  $g(x)$  是具有整数系数的多项式。如果  $x$  的每个幂的系数都关于模  $n$  同余，我们就说  $f(x)$  和  $g(x)$  关于模  $n$  同余，并记作：

$$f(x) \equiv g(x) \pmod{n}$$

例 11.55 因为

$$\begin{aligned} 6 &\equiv 2 \pmod{4} \\ 9 &\equiv 1 \pmod{4} \\ 1 &\equiv -3 \pmod{4} \end{aligned}$$

所以有

$$(6x^2+9x+1) \equiv (2x^2+x-3) \pmod{4}$$

例 11.56 因为

$$8 \not\equiv 1 \pmod{4}$$

所以有

$$(6x^2+8x+1) \not\equiv (2x^2+x-3) \pmod{4}$$

对于缺失的幂，认为其系数为 0。

例 11.57 因为

$$\begin{aligned} 10 &\equiv 0 \pmod{5} \\ 7 &\equiv 2 \pmod{5} \\ 21 &\equiv 6 \pmod{5} \end{aligned}$$

所以有

$$(10x^3+7x^2+21) \equiv (2x^2+6) \pmod{5}$$

例 11.58 因为

$$1 \not\equiv 0 \pmod{5}$$

所以有

$$x^5 \not\equiv x \pmod{5}$$

最后这个例子说明了一点非常有趣的事情。根据定理 11.22，对于所有整数  $x$ ，有

$$x^5 \equiv x \pmod{5}$$

但是，多项式  $x^5$  和  $x$  关于模 5 并不同余。

接下来给出一个定理，它提供了一种判定一个数字是否为质数的简单方法。但首先需要一个引理。

**引理 11.2** 如果  $n$  是质数，则对于所有整数  $m$ ，有

$$(x-m)^n \equiv (x^n - m) \pmod{n}$$

证明：我们有

$$(x-m)^n = \sum_{i=0}^n \binom{n}{i} x^i (-m)^{n-i}$$

因此,

$$\begin{aligned}(x-m)^n - (x^n - m) &= \sum_{i=0}^n \binom{n}{i} (-m)^{n-i} x^i - (x^n - m) \\ &= \sum_{i=1}^{n-1} \binom{n}{i} (-m)^{n-i} x^i + (-m)^n + m\end{aligned}\quad (11.15)$$

由于  $n$  是一个质数, 显然, 对于  $1 < i \leq n-1$ , 有

$$\binom{n}{i} \equiv 0 \pmod{n}$$

因此, 对于每个  $i$  值, 式 11.15 中  $x^i$  的系数关于模  $n$  与 0 同余。所以我们只需要证明:

$$(-m)^n + m \equiv 0 \pmod{n}$$

若  $n=2$ , 则

$$\begin{aligned}[(-m)^2 + m] &\equiv m(m+1) \pmod{2} \\ &\equiv 0 \pmod{2}\end{aligned}$$

最后一步是基于如下事实:  $m$  和  $m+1$  中必然有一个为偶数。若  $n>2$ , 则  $n$  为奇数, 而且

$$\begin{aligned}[(-m)^n + m] &\equiv (-m^n + m) \pmod{n} \\ &\equiv m(-m^{n-1} + 1) \pmod{n}\end{aligned}$$

如果  $m$  是  $n$  的倍数, 则  $m \equiv 0 \pmod{n}$ 。否则, 根据定理 11.22,  $(-m^{n-1}+1) \equiv 0 \pmod{n}$ 。证毕。

**定理 11.28** 假定  $m$  和  $n$  是互质的。因此, 当且仅当

$$(x-m)^n \equiv (x^n - m) \pmod{n} \quad (11.16)$$

时,  $n$  为质数。

**证明:** 根据引理 11.2, 当  $n$  为质数, 满足上面的同余式。

在另一个方向上, 假定  $n$  为合数。设  $q$  是  $n$  的一个因数, 并设  $k$  是  $n$  中  $q$  的次数。 $q^k \not\equiv \binom{n}{q}$  的证明留作习题。由于  $n$  与  $m$  互质, 显然,  $q^k$  与  $(-m)^{n-q}$  互质。因此,  $\binom{n}{q} (-m)^{n-1}$  的因数中不包括  $q^k$ , 这就是说, 它不是关于模  $n$  与 0 同余的。但是, 这个表达式是式 11.15 中的系数之一。证毕。

**例 11.59** 数字 9 和 2 是互质的。根据 11.28 可知:

$$(x-9)^2 = x^2 - 18x + 81 \equiv (x^2 - 9) \pmod{2}$$

这是因为

$$\begin{aligned}1 &\equiv 1 \pmod{2} \\ -18 &\equiv 0 \pmod{2} \\ 81 &\equiv -9 \pmod{2}\end{aligned}$$

**例 11.60** 数字 9 和 4 互质, 且 4 不是质数。根据定理 11.28,

$$(x-9)^4 \not\equiv (x^4 - 9) \pmod{4}$$

以下任务留作习题: 展开此指数形式, 并说明此结果。

**定理 11.28** 给出了一种用于判断一个数字是否为质数的简单算法。即, 给定一个整数  $n$ , 选择一个与  $n$  互质的整数  $m$ , 并判断是否满足同余式 11.16。但是, 由于需要计算同余式 11.16 左侧的  $n+1$  个系数, 所以这个算法具有指数时间复杂度。为改进这一点, 我们玩一点“花招”。但是, 首先需要再给一些定义。

和整数的情景一样,  $\text{mod}$  函数返回一个多项式除以另一个多项式时的余式。也就是说,

$$f(x) \bmod g(x)$$

是  $f(x)$  除以  $g(x)$  时的余式。下面的例子说明了这一思想。

例 11.61 由于

$$\frac{12x^3 + 23x^2 + 7x + 9}{4x^2 + x} = 3x + 5 + \frac{2x + 9}{4x^2 + x}$$

所以

$$(12x^3 + 23x^2 + 7x + 9) \bmod (4x^2 + x) = 2x + 9$$

现在假设  $f(x)$ 、 $g(x)$  和  $h(x)$  是具有整数系数的多项式。如果

$$[f(x) \bmod h(x)] \equiv [g(x) \bmod h(x)] \bmod n$$

我们写为：

$$f(x) \equiv g(x) \bmod [h(x), n]$$

例 11.62 有

$$\begin{aligned}(x-10)^3 \bmod (x^2-1) &= 301x - 1030 \\ (x^3 - 10) \bmod (x^2 - 1) &= x - 10\end{aligned}$$

及

$$(301x - 1030) \equiv (x - 10) \bmod 3$$

所以

$$(x-10)^3 \equiv (x^3 - 10) \bmod (x^2 - 1, 3)$$

例 11.63 有

$$\begin{aligned}(x-14)^7 \bmod (x^3-1) &= -11290188x^2 + 52610713x - 104069042 \\ (x^7 - 14) \bmod (x^3 - 1) &= x - 14\end{aligned}$$

及

$$(-11290188x^2 + 52610713x - 104069042) \equiv (x - 14) \bmod 7$$

所以

$$(x-14)^7 \equiv (x^7 - 14) \bmod (x^3 - 1, 7)$$

例 11.64 有

$$\begin{aligned}(x-20)^7 \bmod (x^4-1) &= 5600001x^3 - 67200140x^2 + 448008400x - 1280280000 \\ (x^7 - 20) \bmod (x^4 - 1) &= x^3 - 20\end{aligned}$$

及

$$-1280280000 \not\equiv -20 \bmod 7$$

所以

$$(x-20)^7 \not\equiv (x^7 - 20) \bmod (x^4 - 1, 7)$$

前面的例子说明了下面定理的正确性。

定理 11.29 假定  $n$  和  $r$  为质数。则对于所有整数  $m$ ，有

$$(x-m)^n \equiv (x^n - m) \bmod (x^r - 1, n) \quad (11.17)$$

证明：由引理 11.2 可得到证明，具体过程留作习题。

如果前面的定理说，当且仅当满足同余式 11.17 时， $n$  为质数，那马上就能使用同余式 11.16 进行改进，判断一个数字是否为质数。也就是说， $(x^n - m) \bmod (x^r - 1)$  的计算是很简单的，Knuth (1998 年) 设计了一种用于计算  $(x-m)^n \bmod (x^r - 1)$  的算法，这种算法使用了快速傅里叶乘法，其时间复杂度属于  $O[r(\lg n)^2]$ 。因此，如果

定理 11.29 中的关系是“当且仅当”，而且  $r$  很小，那就可以相当快速地将同余式 11.16 的判定转为一个系数要少得多的同余式判定。下面的例子说明了这一点。

**例 11.65** 假定我们希望判定 11 是否为质数，而且定理 11.29 是“当且仅当”。选择  $m=4$  和  $r=3$ 。则有

$$(x-4)^{11} \bmod (x^3-1) = -12536127x^2 + 6212052x + 6146928$$

及

$$(x^{11}-4) \bmod (x^3-1) = x^2-4$$

现在要来判定下式是否成立：

$$(-12536127x^2 + 6212052x + 6146928) \equiv (x^2-4) \bmod 11$$

而不再判定下式是否成立：

$$(x-4)^{11} \equiv (x^{11}-4) \bmod 11$$

所以我们将判定 12 个同余式的问题简化为判定 3 个。

这一过程中的问题在于，即使我们选择与  $n$  互质的  $m$  值，定理 11.29 也不是“当且仅当”的关系。也就是说，对于特定的  $m$  和  $r$  值（其中  $m$  与  $n$  互质），一些合数会满足同余式 11.17。所以，如果我们只是检查此同余式是否仅对于特定值满足，那一个合数也可能通过测试。为解决这一问题，我们选择一个适当的  $r$  值，并对几个  $m$  值进行检查。这样，任何合数都不会再被认为是质数，而且仍然会得到一个多项式时间算法。在给出这个算法之后，我们将会得到这些结果。在此之前，需要增加一些符号表示。

回想一下，群元素  $a$  的阶，记为  $\text{ord}(a)$ ，就是满足  $a^t=e$  的最小正整数  $t$ ，其中  $e$  是单位元。给定群  $(\mathbf{Z}_r^*, \times_r)$ ， $\text{ord}([n]_r)$  是满足  $([n]_r)^t=[1]_r$  的最小正整数。也就是说，它是满足下式的最小正整数：

$$n^t \equiv 1 \pmod r$$

我们将它称为  $n$  关于模  $r$  的阶，将其表示为  $\text{ord}_r(n)$ 。

### 算法 11.5 在多项式时间内判定质数

问题：确定一个整数是否为质数。

输入：一个整数  $n > 1$ 。

输出：如果  $n$  为质数，则为真；如果  $n$  为合数，则为假。

```
bool prime (int n)
{
    int q, r, m; bool switch;

    if (对于 k, j>1, n == kj)
        return false;
    r=2; switch = false;
    while(r<n && ! switch){
        if (gcd(n, r)≠1)
            return false;
        if (r 为质数){
            q=r-1 的最大质因数;
            if (q >= 4 √r lg n && n(r-1)/q ≢ 1 mod r)
                switch = true;
        }
        r = r+1;
    }
    if (switch) {
        m = 1;
        while (m <= 2 √r lg n){
            if (((x-m)n ≢ (xn-m) mod (xr-1, n))
                return false;
            m = m + 1;
        }
    }
}
```

```

    }
}
return true;
}

```

## 2. 此算法的正确性

接下来证明此算法的正确性。首先证明这个算法总是将质数判定为质数。

**定理 11.30** 如果算法 11.5 的输入是一个质数，该算法返回真。

证明：如果  $n$  为质数，则对于所有  $r < n$ ，有

$$\gcd(n, r) = 1$$

这意味着在第一个 `while` 循环中不会返回 `false`。由于  $n$  和  $r$  互质，所以根据定理 11.29，在第二个 `while` 循环中也不会返回 `false`。我们得出结论：该算法在最后一条指令处退出，该指令返回 `true`。

要证明这一算法总是将合数判定为合数，会更难一些。下面就来证明。首先需要下面两个引理，它们是引理 11.2 和定理 11.29 的推广。

**引理 11.3** 假定  $g(x)$  是一个系数为整数的多项式，且  $n$  为质数。则

$$[g(x)]^n \equiv g(x^n) \pmod{n}$$

证明：设

$$g(x) = a_0 + a_1x + \cdots + a_dx^d$$

则

$$g(x^n) = a_0 + a_1x^n + \cdots + a_dx^{dn} \quad (11.18)$$

此外， $[g(x)]^n$  中  $x^i$  的系数为：

$$b_i = \sum_{\substack{i_0 + \cdots + i_d = n \\ i_1 + 2i_2 + \cdots + di_d = i}} a_0^{i_0} \cdots a_d^{i_d} \frac{n!}{i_0! \cdots i_d!} \quad (11.19)$$

**情景 1：**对于任意  $j$ ，有  $i \neq jn$ 。则由式 11.18 可以看出， $g(x^n)$  中  $x^i$  的系数为 0。此外，由式 11.19 可以看出，在所有包含  $b_i$  的项中，对于所有  $j$ ，都有  $i_j \neq n$ 。但是， $n$  能整除每一项，这意味着  $n | b_i$ ，且  $b_i \equiv 0 \pmod{n}$ 。这就证明了这一情景。

**情景 2：**当  $0 \leq j \leq d$  时，对于某个  $j$ ，有  $i = jn$ 。于是，由式 11.18 可以看出， $g(x^n)$  中  $x^i$  的系数为  $a_j$ 。此外，如果取  $i_j = n$ ，则有  $ji_j = jn = i$ 。因此，由式 11.19 可以看出， $b_i$  中的某一项为：

$$a_j^{i_j} = a_j^n$$

对于  $b_i$  中的其他每一项，显然，对于每个  $k$  值，都有  $i_k \neq n$ ，这意味着  $n$  可以整除该项，该项关于模  $n$  与 0 同余。根据定理 11.22 有

$$a_j^n \equiv a_j \pmod{n}$$

由这一事实可证得此情景。证毕。

**引理 11.4** 假设  $g(x)$  是一个整系数的多项式，且  $n$  和  $r$  互质。则

$$[g(x)]^n \equiv g(x^n) \pmod{(r-1, n)}$$

证明：由前一引理可得出证明，具体过程留作习题。

我们还需要以下引理。

**引理 11.5** 如果  $r$  和  $q$  为质数， $q$  能整除  $r-1$ ，且  $q \geq r \sqrt{r} \lg n$ ，则当且仅当

$$n^{\frac{r-1}{q}} \not\equiv 1 \pmod{r}$$

时,  $q|\text{ord}_r(n)$ 。

证明: 若令  $t=\text{ord}_r(n)$ , 则根据  $r$  为质数这一事实, 以及定理 11.17 和定理 11.19, 必存在一个满足  $r-1=tk$  的整数  $k$ 。由于  $q$  能整数  $r-1$ , 且  $q$  为质数, 所以由推论 11.2 可以推出  $q|t$  或  $q|k$ 。现在证明“当且仅当”:

假定  $n^{\frac{r-1}{q}} \not\equiv 1 \pmod{r}$ , 且采用反证法, 假设  $q \nmid t$ 。由于  $q|t$  或  $q|k$ , 这就是说  $q|k$ 。因此, 由于  $r-1=tk$ , 所以存在一个满足  $r-1=tjq$  的整数  $j$ , 这意味着  $(r-1)/q=tj$ 。但是, 这样就会得到  $n^{\frac{r-1}{q}}=n^j$ , 这意味着

$$n^{\frac{r-1}{q}} \equiv 1 \pmod{r}$$

这一矛盾证明了  $q|t$ 。

在另一方向上, 假设  $q$  能整除  $t$ , 由于  $q \geq r \sqrt{r} \lg n$ , 所以

$$\frac{r-1}{q} < q \leq t$$

由于  $t=\text{ord}_r(n)$ , 所以这意味着:

$$n^{\frac{r-1}{q}} \not\equiv 1 \pmod{r}$$

证毕。

**引理 11.6** 如果  $n$  是一个合数,  $q$  是一个质数, 且  $q$  能整除  $\text{ord}_r(n)$ , 则存在  $n$  的一个质因数, 满足

$$q|\text{ord}_r(p)$$

证明: 设  $p_1, p_2, \dots, p_k$  是  $n$  的质因数。以下证明留作习题:

$$\text{ord}_r(n) | \text{lcm}(\text{ord}_r(p_1), \text{ord}_r(p_2), \dots, \text{ord}_r(p_k))$$

其中 lcm 是最小公倍数 (least common multiple)。由于  $q|\text{ord}_r(n)$ , 且  $q$  为质数, 所以由定理 11.7 可知, 存在某一满足  $q|\text{ord}_r(p_i)$  的  $p_i$ 。证毕。

下一条定理的证明严重依赖于以下引理。这一引理的证明需要的代数知识超出了本书的范围。对于熟悉抽象代数的读者, 我们在本节末尾给出了证明这一引理的结果。现在给出该引理。

**引理 11.7** 假定算法 11.5 中的第二个 while 循环因为 switch 为真而退出。如果  $p$  与引理 11.6 中相同, 并设  $l = \lfloor 2\sqrt{r} \lg n \rfloor$ , 则存在一个多项式

$$g(x) = (x-1)^{k_1} (x-2)^{k_2} \cdots (x-l)^{k_l}$$

具有以下性质: 如果令

$$J_{g(x)} = \{m, \text{ 满足 } g(x)^m \equiv g(x^m) \pmod{x^r - 1, p}\}$$

则

(1)  $J_{g(x)}$  关于乘法是闭合的。

(2) 存在一个整数  $a > n^{2\sqrt{r}} / 2$ , 使得对于  $m, k \in J_{g(x)}$ , 若

$$m \equiv k \pmod{r}$$

则

$$m \equiv k \pmod{a}$$

证明: 见本节末尾。

现在可以证明我们的主要结果了。

**定理 11.31** 如果算法 11.5 的输入是一个合数，该算法将会返回 false。

**证明：**如果该数字是一个合数，第一个 while 循环将因为其中的 return 语句而直接结束，于是返回 false，得证。如果第一个 while 循环因为  $r=n-1$  而退出，则  $n$  必须为质数。所以我们将假定输入了一个合数，而且第一个 while 循环因为 switch 为真而退出。因此，根据引理 11.5， $q$  能整除  $\text{ord}_r(n)$ 。采用反证法，假定该算法返回 true。则根据第二个 while 循环，对于  $1 \leq m \leq 1 = \lfloor 2\sqrt{r} \lg n \rfloor$

$$(x-m)^n \equiv (x^n - m) \pmod{(x^r - 1, n)}$$

这意味着：

$$(x-m)^n \equiv (x^n - m) \pmod{(x^r - 1, p)} \quad (11.20)$$

其中  $p$  和引理 11.6 中一样。因此，引理 11.7 中  $g(x)$  的每一项  $(x-m)$  都分别满足同余式 11.20，这意味着：

$$[g(x)]^n \equiv g(x^n) \pmod{(x^r - 1, p)}$$

因此， $n \in J_{g(x)}$ ，其中  $J_{g(x)}$  与引理 11.7 中的定义相同。此外，根据引理 11.4 有  $p \in J_{g(x)}$ ，而且显然有  $1 \in J_{g(x)}$ 。现在考虑如下集合：

$$E = \{n^i p^j, \text{ 满足 } 0 \leq i, j \leq \lfloor \sqrt{r} \rfloor\}$$

根据引理 11.7 的第一部分， $E \subseteq J_{g(x)}$ 。此外，

$$|E| = \left(1 + \lfloor \sqrt{r} \rfloor\right)^2 > r$$

因此，根据鸽笼原理， $E$  中有两个元素  $n^i p^j$  和  $n^h p^k$  ( $i \neq h$  或  $j \neq k$ )，使得

$$n^i p^j \equiv n^h p^k \pmod{r}$$

因此，根据引理 11.7 的第二部分，有

$$n^i p^j \equiv n^h p^k \pmod{a} \quad (11.21)$$

其中  $a$  的定义与该引理中一样。由于  $p|n$ ，所以  $n$  为合数， $i, j \leq \lfloor \sqrt{r} \rfloor$ ，

$$n^i p^j \leq n^{\sqrt{r}} \left(\frac{n}{2}\right)^{\sqrt{r}} = \frac{n^{2\sqrt{r}}}{2^{\sqrt{r}}}$$

同理，由于  $h, k \leq \lfloor \sqrt{r} \rfloor$

$$n^h p^k \leq \frac{n^{2\sqrt{r}}}{2^{\sqrt{r}}}$$

但是，由于  $a > n^{2\sqrt{r}} / 2$ ，于是可由同余式 11.21 推出：

$$n^i p^j = n^h p^k$$

由于  $p|n$ ，且有  $i \neq h$  或  $j \neq k$ ，所以由最后一个等式可推出，对于某一整数  $s \geq 1$ ，有

$$n = p^s$$

但是，在该算法的第一个步骤中，我们检查  $n$  是否为  $p^s$  形式 ( $s \geq 2$ )。因此， $s=1$ ，且  $n$  为质数。这一矛盾证明了本定理。

### 3. 此算法的时间复杂度

接下来讨论算法 11.5 的时间复杂度。首先需要一些引理和一个定理。

**引理 11.8** 设  $q_m$  是  $m$  的最大质因数。则存在一个正常数  $c$  和整数  $N$ ，使得对于  $n > N$ ，有

$$\left| \left\{ \text{质数 } p \text{ 满足 } p \leq n, \text{ 且 } q_{p-1} > n^{2/3} \right\} \right| \geq c \frac{n}{\lg n}$$

证明：在 Baker 和 Harman (1996 年) 的文献中可找到此证明。

引理 11.9 设  $\pi(m)$  是小于或等于  $m$  的质数的个数。则对于  $m \geq 1$ , 有

$$\frac{m}{6\lg m} \leq \pi(m) \leq \frac{8m}{\lg m}$$

证明：可在 Apostol (1997 年) 的文献中可找到此证明。

引理 11.10 给定正整数  $m$  和  $n$ , 下面的乘积最多有  $m^2 \lg n$  个质因数。

$$(n-1)(n^2-1)\cdots(n^m-1)$$

证明：每一项最多有  $m \lg n$  个质数，共有  $m$  项。

定理 11.32 存在正常数  $c_1, c_2$  和整数  $N$ , 使得对于每个  $n > N$ , 在区间

$$(c_1(\lg n)^6, c_2(\lg n)^6)$$

中存在一个质数  $r$ , 使  $r-1$  的最大质因数  $q$  满足

$$q \geq 4r^{1/2} \lg n, \quad n^{\frac{r-1}{q}} \not\equiv 1 \pmod{r}$$

证明：暂时设  $c_1$  和  $c_2$  是任意正整数。根据引理 11.8, 存在一个正常数  $c$  和整数  $N$ , 使得对于  $c_2(\lg n)^6 > N$ , 有

$$\left| \left\{ \text{质数 } p, \text{ 满足 } p \leq c_2(\lg n)^6 \text{ 及 } q_{p-1} > \left[ c_2(\lg n)^6 \right]^{2/3} \right\} \right| \geq c \frac{c_2(\lg n)^6}{\lg(c_2(\lg n)^6)} \quad (11.22)$$

由式 11.22 可以推出：

$$\left| \left\{ \text{质数 } p, \text{ 满足 } p \leq c_2(\lg n)^6 \text{ 及 } q_{p-1} > \left[ c_2(\lg n)^6 \right]^{2/3} \right\} \right| \geq \frac{cc_2}{7} \frac{(\lg n)^6}{\lg(\lg n)}$$

其证明留作习题。如果设引理 11.9 中  $m = c_1(\lg n)^6$ , 在经过一些运算之后, 可以得到小于或等于  $c_1(\lg n)^6$  的质数的个数不大于

$$\frac{8c_1}{6} \frac{(\lg n)^6}{\lg(\lg n)} \quad (11.23)$$

结合不等式 11.22 和式 11.23 可以得到, 在区间

$$\left[ c_1(\lg n)^6, c_2(\lg n)^6 \right] \quad (11.24)$$

中满足

$$q_{p-1} > \left[ c_2(\lg n)^6 \right]^{2/3} \quad (11.25)$$

的质数  $p$  的个数大于或等于

$$\left( \frac{cc_2}{7} - \frac{8c_1}{6} \right) \frac{(\lg n)^6}{\lg(\lg n)}$$

将这些质数称为特殊质数。回想一下,  $c_1$  和  $c_2$  是任意正整数。注意, 我们选择  $c_1 \geq 4^6$ 。于是, 对于任意特殊质数  $p$ , 有

$$\begin{aligned} q_{p-1} &> p^{2/3} = p^{1/2} p^{1/6} \\ &\geq p^{1/2} \left[ c_1(\lg n)^6 \right]^{1/6} \\ &\geq p^{1/2} \left[ 4^6 (\lg n)^6 \right]^{1/6} \\ &\geq p^{1/2} 4 \lg n \end{aligned} \quad (11.26)$$

第一个不等式是通过不等式 11.24 和不等式 11.25 得出的，第二个是通过不等式 11.25 得出的。

现在选择  $c_2$ ，使得  $c_3 = \left( \frac{cc_2}{7} - \frac{8c_1}{6} \right) > c_2^{2/3}$ 。于是可得，对得  $n > N$ ，特殊质数的个数大于或等于

$$c_3 \frac{(\lg n)^6}{\lg(\lg n)} \quad (11.27)$$

现在令  $x = c_2(\lg n)^6$ 。则

$$x^{2/3} \lg n = \left[ c_2(\lg n)^6 \right]^{2/3} \lg n = c_2^{2/3} (\lg n)^5 < c_3 \frac{(\lg n)^6}{\lg(\lg n)} \quad (11.28)$$

根据引理 11.10，乘积

$$\Pi = (n-1)(n^2-1)\cdots(n^{\lfloor x^{1/3} \rfloor}-1)$$

最多有  $\lfloor x^{2/3} \rfloor \lg n$  个质因数。因此，根据式 11.27 和式 11.28，至少存在一个不能整除  $\Pi$  的特殊质数  $r$ 。

我们将证明， $r$  满足定理的条件。显然，根据不等式 11.24 和不等式 11.26， $r$  满足前两个条件。因此，如果令  $q = q_{r-1}$ ，只需要证明：

$$n^{\frac{r-1}{q}} \not\equiv 1 \pmod{r}$$

为此，假定

$$n^{\frac{r-1}{q}} \equiv 1 \pmod{r}$$

则  $r \mid \left( n^{\frac{r-1}{q}} - 1 \right)$ 。于是，由于  $r$  不能整除  $\Pi$ ，所以只需要证明：

$$\frac{r-1}{q} \leq \lfloor x^{1/3} \rfloor = \lfloor c_2^{1/3} (\lg n)^2 \rfloor$$

来得到矛盾。由于不等式 11.25 表明  $q > [c_2(\lg n)^6]^{2/3}$ ，所以只需要证明：

$$\begin{aligned} r-1 &\leq \lfloor c_2^{1/3} (\lg n)^2 \rfloor \lceil c_2(\lg n)^6 \rceil^{2/3} \\ &\leq c_2(\lg n)^6 \end{aligned}$$

但是，根据不等式 11.24，最后一个不等式的确成立。这一矛盾完成了证明：

**算法 11.5 的分析** 最差时间复杂度（在多项式时间内确定质数）

**基本运算：**一位运算。

**输入规模：**用于对  $n$  进行编码所需要的位数  $s$ ，给出如下。

$$s = \lceil \lg n \rceil + 1$$

为确定  $n$  是否为  $k^j$  形式，要检查的根数属于  $O(s)$ 。也就是说，需要检查  $n^{1/2}, n^{1/3}, \dots, n^{1/m}$ ，其中， $m = \lfloor \lg n \rfloor$ 。利用 2.6 节讨论的暴力方法，检查每个根的时间复杂度属于  $O(s^2)$ 。因此，这一确定过程的总时间复杂度属于

$$O(ss^2) = O(s^3)$$

根据定理 11.32，第一个 while 循环的执行次数属于  $O(s^6)$ 。现在来讨论在每次循环中完成的工作。由于  $r < n$ ，如果使用算法 11.1，则  $\gcd(n, r)$  计算的时间复杂度属于  $O(s^2)$ 。假定我们使用 9.2 节开头给出的算法来判定  $r$  是否为质数，并求出  $r-1$  的最大质因数。算法中 while 循环最多执行  $r^{1/2}$  遍，利用 2.6 节讨论的暴力方法，在每次执行中余数的计算复杂度属于  $O(s^2)$ 。因此，总时间复杂度属于  $O(r^{1/2}s^2)$ ，这就意味着，根据定理 11.32，该时

间复杂度属于  $O(s^3 s^2) = O(s^5)$ 。再次使用 2.6 节中的方法，在第一个 `while` 循环中递增  $r$  并检查退出条件的时间复杂度属于  $O(s^2)$ 。因此，第一个 `while` 循环的总时间复杂度属于

$$O(s^6 s^5) = O(s^{11})$$

由于  $r < c(\lg n)^6$ ，所以有可能获得一个比上面更严格的界限。但是，后面将会看到，无论如何，最终界限主要由第二个 `while` 循环决定。

第二个 `while` 循环的执行次数属于  $O(r^{1/2}s)$ ，这就意味着，根据定理 11.32，执行的次数属于  $O(s^3 s) = O(s^4)$ 。在定理 11.29 之后我们曾经提到，如果使用快速傅里叶乘法，在该循环中判定同余性的时间复杂度属于  $O(rs^2)$ ，也就是说，再次根据定理 11.32 可得，该时间复杂度属于  $O(s^6 s^2) = O(s^8)$ 。因此，第二个 `while` 循环的总时间复杂度属于

$$O(s^8 s^4) = O(s^{12})$$

得出结论：

$$W(S) \in O(s^{12})$$

Agrawal 等人（2002 年）注意到，在实践中，算法 11.5 的速度可能远快于前面得出的界限。事实上，他们证明了，如果他们的推测正确，那该算法的启发式时间复杂度属于  $O(s^6)$ 。

#### ⊕4. 证明引理 11.7 的引理

接下来将给出证明引理 11.7 的引理。本节需要抽象代数的知识。首先解释我们使用的符号。给定一个环  $R$ ，用  $R[x]$  表示相应的多项式环。例如，与  $\mathbf{Z}_n$  对应的多项式环用  $\mathbf{Z}_n[x]$  表示。给定一个环  $R$  和  $R$  中的一个理想  $L$ ，由开始  $\{r+l : l \in L\}$  组成的商环表示为  $R/L$ 。此外，如果  $r \in R$ ，而且  $L$  是由  $r$  的所有倍数组成的理想，则将  $R/L$  记为  $R/r$ 。例如， $\mathbf{Z}_n$  可以表示为  $\mathbf{Z}/n$ 。再给一个例子，给定一个多项式环  $R(x)$  和多项式  $f(x) \in R(x)$ ，由  $f(x)$  的所有倍数组成的理想所确定的商环表示为  $R(x)/f(x)$ 。给定一个质数  $p$ ，我们感兴趣的是环（实际上是域） $\mathbf{Z}_p$ 、多项式环  $\mathbf{Z}_p[x]$  和商环  $\mathbf{Z}_p[x]/h(x)$ 。

我们不加证明地给出以下引理。这些证明使用标准代数运算，可依次完成，在 Agrawal 等人（2002 年）的文献中可找到这些证明。

**引理 11.11** 设  $p$  和  $r$  是质数，且  $p \neq r$ 。

(1) 若  $h(x)$  是  $x^r - 1$  的一个因式，且  $m \equiv k \pmod{r}$ ，则

$$x^m \equiv x^k \pmod{h(x)}$$

(2) 若令  $\text{ord}_r(p)$  是  $p$  关于模  $r$  的阶，则  $(x^r - 1)/(x - 1)$  因式分解为多项式，它与  $\mathbf{Z}_p$  是不可约的，度为  $\text{ord}_r(p)$ 。

假定算法 11.5 中的第二个 `while` 循环因为 `switch` 为真而退出， $p$  与引理 11.6 中相同，令  $l = \lfloor 2\sqrt{r} \lg n \rfloor$ 。根据前面引理的第二部分，存在一个多项式  $h(x)$ ，它是  $x^r - 1$  的一个因式，且与  $\mathbf{Z}_p$  不可约，度为  $\text{ord}_r(p)$ 。下面的引理全都假定算法 11.5 中的第二个 `while` 循环是因为 `switch` 为真而退出， $p$  与引理 11.6 中相同， $h(x)$  是上面刚刚描述的多项式， $l = \lfloor 2\sqrt{r} \lg n \rfloor$ 。

**引理 11.12** 在  $\mathbf{Z}_p[x]/h(x)$  中，设  $G$  是由  $l$  个二项式生成的群

$$(x-1), (x-2), \dots, (x-l)$$

即

$$G = \left\{ \prod_{1 \leq m \leq l} (x-m)^{k_m} \mid \text{满足对于 } 1 \leq m \leq l \text{ 有 } k_m \geq 0 \right\}$$

则  $G$  是周期性的，且

$$|G| > \left\lceil \frac{\text{ord}_r(p)}{l} \right\rceil$$

其中  $|G|$  是  $G$  中的元素个数。

引理 11.13 设  $G$  是前面引理中的群,  $g(x)$  是  $G$  的生成元, 且  $a_g$  是  $g(x)$  在  $\mathbb{Z}_p[x]/h(x)$  中的阶。则

$$a_g > \frac{n^{2\sqrt{r}}}{2}$$

证明: 由于  $l = \lfloor 2\sqrt{r} \lg n \rfloor$  及  $q \geq 4\sqrt{r} \lg n$ , 所以有  $q \geq 2l$ 。由于引理 11.6 表明  $q \mid \text{ord}_r(p)$ , 所以这意味着  $\text{ord}_r(p) \geq 2l$ 。根据前一引理, 可得:

$$|G| > \left( \frac{\text{ord}_r(p)}{l} \right)^l \geq \left( \frac{2l}{l} \right)^l = 2^l = 2^{\lfloor 2\sqrt{r} \lg n \rfloor} \geq 2^{2\sqrt{r} \lg n - 1} = \frac{n^{2\sqrt{r}}}{2}$$

现在, 根据  $g(x)$  是  $G$  的生成元这一事实即可得证。

引理 11.14 设  $g(x)$  与前一定理中相同。于是, 集合

$$J_{g(x)} = \{m, \text{ 满足 } g(x)^m \equiv g(x^m) \pmod{x^r - 1, p}\}$$

关于乘法是闭合的。

引理 11.15 设  $g(x), a_g$  和  $J_{g(x)}$  和前面的引理中一样。于是对于所有  $m, k \in J_{g(x)}$ , 如果

$$m \equiv k \pmod{r}$$

则

$$m \equiv k \pmod{a_g}$$

如果令  $a = a_g$ , 由前一定理可以直接得出引理 11.7。

## 11.7 RSA 公钥密码系统

回想本章开头讨论的一种情景, 当时说到 Bob 希望通过互联网向 Alice 发送一封秘密情书。我们注意到, 如果他可以对消息进行编码, 使其显示为杂乱信息, 且只有 Alice 可以将此杂乱信息解码回原消息, 那他就不需要担心他的朋友们截获该消息。公钥加密系统可以让 Bob 做到这一点。在描述此类系统后, 我们将给出 RSA 公钥加密系统。

### 11.7.1 公钥加密系统

公钥加密系统 (public-key cryptosystem) 包括一组许可消息, 一组参与者, 其中每位参与者都有一个公钥 (public key) 和一个私钥 (secret key), 还包括一个在参与者之间发送消息的网络。这组许可消息中可能包含了某一给定长度的所有字符序列, 也可能较少一些。如果令

$$M = \text{许可消息的集合}$$

则每个参与者  $x$  的公钥  $\text{pkey}_x$  和私钥  $\text{skey}_x$  分别决定了由  $M$  到  $M$  的函数  $\text{pub}_x$  和  $\text{sec}_x$ , 它们是互逆的。也就是说, 对于每个  $b \in M$

$$\begin{aligned} b &= \text{pub}_x(\text{sec}_x(b)) \\ b &= \text{sec}_x(\text{pub}_x(b)) \end{aligned}$$

现在, 每位参与者的公钥都为所有参与者所了解, 但  $x$  的私钥只有他自己知道。例如, 如果 Bob 希望向 Alice 发送一封秘密情书  $b$ , 他和 Alice 可执行以下步骤。

- (1) Bob 使用 Alice 的公钥  $\text{pkey}_{\text{Alice}}$  计算  $c = \text{pub}_{\text{Alice}}(b)$ 。消息  $c$  被称为密文 (ciphertext)。它是不可读的。
- (2) Bob 将密文  $c$  发送给 Alice。
- (3) Alice 使用她的私钥  $\text{skey}_{\text{Alice}}$  计算  $b = \text{sec}_{\text{Alice}}(c)$ 。

例 11.66 假定 Bob 希望向 Alice 发送消息 “I love you”。其步骤如下。

(1) Bob 计算

$$\text{pub}_{\text{Alice}}(\text{"I love you"})$$

假定结果为 “@!##%\*(!”。

(2) Bob 将这一消息发送给 Alice。Bob 的朋友看到 “@!##%\*(!”。

(3) Alice 计算

$$\text{sec}_{\text{Alice}}(\text{"@!##%*(!"}) = \text{I love you.}$$

第(1)步应用  $\text{pub}_{\text{Alice}}$  的过程称为加密 ( encryption )，而第(3)步应用  $\text{sec}_{\text{Alice}}$  的过程称为解密 ( decryption )。这些步骤在图 11-1 中演示。注意，由于只有 Alice 知道  $\text{sec}_{\text{Alice}}$ ，所以只有她可以将密文  $c$  解码为原消息  $b$ 。所以 Bob 的朋友们只能看到密文。

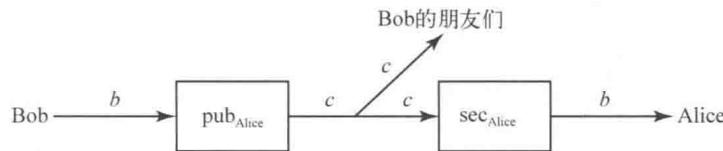


图 11-1 Bob 使用 Alice 的公钥对消息  $b$  加密，Alice 使用她的私钥对消息解密。Bob 的朋友们只看到密文  $c$

只要不能由  $\text{pkey}_x$  确定  $\text{skey}_x$  ( 或者至少它非常困难 )，那这一方法就是有效的。接下来给出一个系统，它让这一过程变得非常困难。

### 11.7.2 RSA 加密系统

RSA 加密系统依赖于这样一些事实：我们可以相当轻松地找到大质数，却没有高效方法用来完成对大数的因式分解。在给出此方法之后，将进一步讨论这些事实。

#### 1. 系统

在 RSA 公钥加密系统中，每位参与者都根据以下步骤生成他的公钥和私钥。

- (1) 选择两个非常大的质数  $p$  和  $q$ 。表示  $p$  和  $q$  所需要的比特数可能为 1024。
- (2) 计算

$$\begin{aligned} n &= pq \\ \varphi(n) &= (p-1)(q-1) \end{aligned}$$

$\varphi(n)$  的公式是根据定理 11.17 给出的。

- (3) 选择一个与  $\varphi(n)$  互质的小质数  $g$ 。

- (4) 利用算法 11.3，计算  $[g]_{\varphi(n)}$  的乘法逆  $[h]_{\varphi(n)}$ 。即，

$$[g]_{\varphi(n)} [h]_{\varphi(n)} = [1]_{\varphi(n)}$$

根据推论 11.8，这个逆存在且唯一。

- (5) 设  $\text{pkey}=(n, g)$  为公钥， $\text{skey}=(n, h)$  为私钥。

许可消息的集合为  $\mathbf{Z}_n$ 。与公钥  $\text{pkey}=(n, g)$  对应的函数为：

$$\text{pub}(b) = b^g \quad (11.29)$$

其中  $b \in \mathbf{Z}_n$ ，与私钥  $\text{skey}=(n, h)$  对应的函数为：

$$\text{sec}(b) = b^h \quad (11.30)$$

这些函数的值可使用算法 11.5 计算。

为使此系统正确运行，与这些公钥和私钥对应的函数必须互逆。接下来进行证明。

◆定理 11.33 式 11.29 和式 11.30 中的函数互逆。

证明：根据等式 11.29 和式 11.30，对于任意  $b \in \mathbf{Z}_n$ ，有

$$\text{pub}(\sec(b)) = \sec(\text{pub}(b)) = b^{gh}$$

所以只需要证明：

$$b^{gh} = b$$

为此，设  $m \in b$ 。（回想  $b \in \mathbf{Z}_n$ 。）于是  $m^{gh} \in b^{gh}$ 。我们将证明：

$$[m^{gh}]_p = [m]_p \quad (11.31)$$

由于  $g$  和  $h$  是以  $\varphi(n) = (p-1)(q-1)$  为模的乘法逆，

$$[gh]_{(p-1)(q-1)} = [1]_{(p-1)(q-1)}$$

这意味着存在一个整数  $k$ ，满足

$$gh = 1 + k(p-1)(q-1)$$

存在两种情景。

情景 1：假定  $[m]_p \neq [0]_p$ 。于是有

$$\begin{aligned} [m^{gh}]_p &= [m^{1+k(p-1)(q-1)}]_p \\ &= [m]_p ([m]_p^{(p-1)})^{k(q-1)} \\ &= [m]_p [1]_p^{k(q-1)} \\ &= [m]_p \end{aligned}$$

上面第三个等式是根据定理 11.22 得到的。

情景 2：如果  $[m]_p = [0]_p$ ，则

$$[m^{gh}]_p = [m]_p^{gh} = [0]_p^{gh} = [0]_p = [m]_p$$

所以我们已经推导出式 11.31。同理

$$[m^{gh}]_q = [m]_q \quad (11.32)$$

根据式 11.31 和式 11.32，

$$m^{gh} \equiv m \pmod{p}$$

及

$$m^{gh} \equiv m \pmod{q}$$

由于定理 11.13，于是有

$$m^{gh} \equiv m \pmod{n}$$

这意味着：

$$b^{gh} = b$$

证毕。

## ◆2. 讨论

前面曾经提到，RSA 加密系统的成功依赖于如下事实：我们可以相当轻松地找到大质数，却没有高效方法可以完成大整数的因数分解。也就是说，可以像下面这样找到大的质数。首先，随机选择适当大小的质数。

如 11.6.1 节开头的讨论，应当不需要太长时间就能找到一个相当大的质数。对于选定的每个整数，可以使用算法 11.5 来查看该数字是否为质数（此算法是多项式时间的）。重复这一过程，直到找出两个大质数。如 11.6.2 节中的讨论，即使是在开发出算法 11.5 之前，使用 Miller-Rabin 随机化质数检测方法就可以非常高效地判断一个数字是否为质数，而且错误率极低。实际上，由于 Miller-Rabin 随机化质数检测算法的时间复杂度属于  $O(es^3)$ ，所以在选择判断质数的算法时，它仍然可以作为一个选项（其中  $s$  是对输入进行编码所需要的比特数， $e$  是选定的一个整数，使该算法出错的概率不大于  $2^{-e}$ ）。因此，如果选择的  $e$  仅为 40，则一个错误的概率不大于  $2^{-40}=9.0949\times10^{-13}$ 。

另一方面，还从来没有人发现一种实现因数分解的多项式时间算法。有这样一种可能：有人可能无需对  $n$  进行因数分解就能找出密钥  $\text{skey}=(n, h)$  中的  $h$  值。但也还没人能找到一种高效方法来完成这一任务。目前，如果使用的整数中包含大约 1024 或更多个比特，RAS 加密系统就能保障加密的安全性。

## 11.8 习题

### 11.1 节

1. 求以下整数的正约数。

- (a) 72
- (b) 31
- (c) 123

2. 证明：若  $h|m$  且  $m|n$ ，则  $h|n$ 。

3. 证明：当且仅当两个整数相等时，它们能相互整除。

4. 设  $p$  和  $q$  是两个质数。若  $p=q+2$ ，则称  $p$  和  $q$  为“孪生质数”。找出两队孪生质数。

5. 证明： $\gcd(n, m)=\gcd(m, n)$ 。

6. 证明：若  $m$  和  $n$  都是偶数，则  $\gcd(m, n)=2 \gcd(m/2, n/2)$ 。

7. 证明：若  $n \geq m > 0$ ，则  $\gcd(m, n)=\gcd(m, n-m)$ 。

8. 证明：若  $p$  为质数且  $0 < h < p$ ，则  $\gcd(p, h)=1$ 。

9. 利用推论 11.2 证明：如定理 11.5 中的讨论，一个整数的质因数分解是唯一的。

10. 将下面每个整数写为质数的乘积。

- (a) 123
- (b) 375
- (c) 927

11. 证明定理 11.6。

12. 证明定理 11.7。

13. 证明：对于正整数  $m$  和  $n$ ，当且仅当  $m=n$  时， $\gcd(m, n)=\text{lcm}(m, n)$ 。

### 11.2 节

14. 阐明算法 11.1 在顶级调用为  $\gcd(68, 40)$  时的流程。

15. 编写算法 11.1 的一个迭代版本。你的算法应当仅使用恒定数量的内存（即，空间复杂度函数属于  $\theta(1)$ ）。

16. 编写一种算法，使用算法 11.1 将一个有理数表示为它的最低项。可以假定这个有理数以分数  $m/n$  的形式给出，其中  $m$  和  $n$  为整数。

17. 阐明算法 11.2 在顶级调用为  $\text{Euclid}(64, 40, \gcd, i, j)$  时的流程。

18. 编写一个算法，使用减法来计算最大公约数（见第 7 题）。分析你的算法。

### 11.3 节

19. 证明：例 11.21 中  $(S, *)$  是一个群。

20. 证明定理 11.12。
21. 以下工作是在证明定理 11.13 时留作习题的部分。证明存在一个整数  $c$ , 使得  $h_1 = cn_2n_3 \cdots n_j$ 。
22. 证明定理 11.14。
23. 证明: 若  $s \in [m]_n$ , 且  $t \in [k]_n$ , 则  $s \times t \in [m \times k]_n$ 。
24. 证明: 若  $G=(S, *)$  是一个有限群, 且  $a \in S$ , 则存在整数  $k, m \geq 1$ , 使得  $a^k = a^m$ 。
25. 证明: 若  $S=\{[0]_{12}, [3]_{12}, [6]_{12}, [9]_{12}\}$ , 则  $(S, +)$  是  $(\mathbf{Z}_{12}, +)$  的一个子群。
26. 利用定理 11.19 定理推论 11.3。
27. 考虑群  $(\mathbf{Z}_9^*, \times)$ 。证明  $\langle [2]_9 \rangle = \mathbf{Z}_9^*$ 。

#### 11.4 节

28. 求解以下模方程。
- $[8]_{10}x = [4]_{10}$
  - $[4]_{17}x \equiv [5]_{17}$
29. 实现算法 11.3, 并针对各种问题实例运行它。
30. 求出方程  $[1]_7x = [3]_7$  和  $[12]_9x = [6]_9$  的所有解。

#### 11.5 节

31. 通过求 3 的 12 次幂, 计算  $([3]_{73})^{12}$ 。
32. 通过求 7 的 15 次幂, 计算  $([7]_{73})^{15}$ 。
33. 使用定理 11.4 计算  $([3]_{73})^{12}$ 。
34. 使用定理 11.4 计算  $([7]_{73})^{15}$ 。
35. 实现算法 11.4, 并针对各种问题实例运行它。

#### 11.6 节

36. 计算小于或等于 100 的质数的个数。
37. 如果根据均匀分布随机选择一个介于 1 到 10 000 的整数, 它为质数的近似概率近似为多少?
38. 假定我们根据均匀分布随机选择了介于 1 到 10 000 之间的 100 个数字, 所有它们都不是质数的概率近似为多少。
39. 证明: 若  $n$  是一个质数, 且  $1 < k \leq n-1$ , 则  $B(n, k) \equiv 0 \pmod{n}$ , 其中  $B(n, k)$  表示二项式系数。
40. 证明: 若  $q$  是  $n$  的一个因数,  $k$  是  $q$  在  $n$  中的阶, 则  $q^k | B(n, q)$ , 其中  $B(n, q)$  表示二项式系数。
41.  $9x^3+2x$  和  $x^2-4$  是否关于模 2 同余?
42. 证明:  $(x-9)^4$  与  $(x^4-9)$  关于模 4 不同余。
43. 证明:  $(x-5)^3$  与  $(x^3-5)$  关于模 3 同余。
44. 证明定理 11.29。
45. 实现算法 11.5, 并针对不同问题实例运行它。
46. 使用引理 11.3 证明引理 11.4。
47. 下面是在证明引理 11.6 时留作习题的问题。证明:  $\text{ord}_r(n) | \text{lcm}(\text{ord}_r(p_1), \text{ord}_r(p_2), \dots, \text{ord}_r(p_k))$ 。
48. 使用不等式 11.22 获得它后面的不等式。

#### 11.7 节

49. 公钥和私钥之间有什么区别?
50. 考虑一个  $p=7, q=11$  和  $g=13$  的 RSA 加密系统。
- 计算  $n$ 。
  - 计算  $\varphi$ 。
  - 找出  $h$ 。
51. 考虑一个  $p=23, q=41$  和  $g=3$  的 RSA 加密系统。对消息  $[847]_{943}$  进行加密。

52. 使用第 51 题的 RSA 加密系统，对该题中的加密消息进行解密。
53. 在一个 RAS 加密系统中，证明：如果可以发现  $\varphi(n)$ ，那就可能危及该加密系统的安全。
- 补充习题
54. 证明：存在无穷多个质数。
55. 证明：gcd 运算符合结合律。即，对于所有整数  $m, n$  和  $h$ ，都有  $\text{gcd}(m, \text{gcd}(n, h)) = \text{gcd}(\text{gcd}(m, n), h)$ 。
56. 证明：如果  $m$  为奇数， $n$  为偶数，则  $\text{gcd}(m, n) = \text{gcd}(m, n/2)$ 。
57. 证明：如果  $m$  和  $n$  都为奇数，则  $\text{gcd}(m, n) = \text{gcd}((m-n)/2, n)$ 。
58. 给出可以由等式  $mx \equiv my \pmod{n}$  推出  $x \equiv y \pmod{n}$  的必要条件。
59. 假定  $p$  是一个质数，求解方程  $x^2 \equiv [1]_p$ 。
60. 在一个 RSA 加密系统中，设  $p$  和  $q$  是大质数，设  $n=pq$ ，且 pub 是公钥。证明： $\text{pub}(a)\text{pub}(b)$  与  $\text{pub}(ab)$  关于模  $n$  同余。



## 并行算法简介

假设你想在后院建一道篱笆，需要为 10 个篱笆桩各挖一个深洞。考虑到依次挖 10 个洞会非常累人，你决定寻找一种替代方法。你想起马克·吐温笔下的著名人物汤姆·索亚，他假装粉刷篱笆是件很有趣的事，从而哄骗他的朋友们来帮他完成任务。你决定利用同样的小花招，但要稍做更新。你向自己那些注重健康的邻居们分发了一些传单，宣布要在你的后院举办一场挖坑比赛，身体最好的人应当可以在最短时间内挖好一个坑，从而赢得比赛。你提供了一点微不足道的冠军奖品，比如六瓶装的啤酒，你知道奖品对于你的邻居们来说并不重要。他们只是希望自己有多健康。在比赛日，10 位强壮的邻居同时挖了 10 个坑。这叫作并行挖坑。你自己省了很大的力气，而且完成速度要远快于你自己逐一挖掘的速度。

让朋友们同时挖坑，可以更快速地完成任务。同样，如果一台计算机能够有许多处理器并行执行命令，那它也能更快速地完成任务（计算机中的处理器是一种用于处理指令和数据的硬件）。到目前为止，我们只讨论了顺序处理。也就是说，我们现在给出的所有算法，在设计时就是准备在一种传统的顺序计算机上实现的。这种计算机有一个依次执行指令的处理器，类似于你自己依次挖 10 个坑。这些计算机基于约翰·冯诺依曼引入的模型。如图 12-1 所示，这一模型包括单个处理器（称为中央处理器，CPU）和内存。这个模型获得单个指令序列，同时对单个数据序列进行处理。这种计算机称为单指令流、单数据流（SISD）计算机，通常称为串行计算机（serial computer）。



图 12-1 传统串行计算机

如果一台计算机拥有多个可以同时（并行）执行指令的处理器，那许多问题的解决速度都会快得多。这好像是让你的 10 位邻居同时挖 10 个坑。例如，考虑 6.3 节引入的贝叶斯网络。图 12-2 给出了这样一个网络。网络中的每个顶点表示病人的一种可能状态。如果拥有第一个顶点处的状态，可能会导致拥有第二个顶点处的状态，那就存在一条从第一个顶点到第二顶点的边。例如，右上方的顶点表示其状态是吸烟者，由该顶点发出的线表示抽烟可能导致肺癌。某种给定原因并非总会导致其潜在的结果。因此，每种原因导致每种结果的概率也需要存储在网络中。例如，作为吸烟者的概率（0.5）存储在包含“吸烟者”的顶点处。在包含“肺癌”的顶点处存储了两个概率，一个是已知其为吸烟者时得肺癌的概率（0.1），一个是已知其不是吸烟者时得肺癌的概率（0.01）。在包含“胸部 x 射线阳性”的顶点处存储了四个概率，一个是已知病人同时患有肺结核和肺癌时胸部 x 射线为阳性的概率（0.99），另外三个概率是其他三种不同组合原因导致胸部 x 射线阳性的概率。一个贝叶斯网络中的基本推理问题是：在已知特定顶点存在某些状态时，计算在所有其他顶点处拥有这些状态的概率。例如，如果已知一位病人是吸烟者，而且胸部 x 射线为阳性，那可能希望知道这位病人患肺癌、肺结核、喘不上气和最近访问过亚洲的概率。Pearl（1986 年）为解决这一问题开发了一种推理算法。在这种算法中，每个顶点都向其父顶点和子顶点发送消息。例如，在知道病人的胸部 x 射线为阳性时，“胸部 x 射线阳性”的顶点会向其父顶点“肺结核”和“肺癌”发送消息。当这些顶点分别接到它的消息后，就会计算在该顶点处出现该状态的概率，再将这消息发送给它的父顶点和其他子顶点。当这些顶点接收到消息后，再计算这些顶点处拥有这些状态的新概率，之后这些顶点再发送消息。这一消息传递机制在根顶点和叶顶点处终止。当了解到该病人还是一位吸烟者时，在该顶点处启动另一个

消息流。传统串行计算机只能同时计算一个消息或一个概率值。可以首先计算传送给“肺结核”的消息值，然后是肺结核的新概率，然后是发送给“肺癌”的消息值，然后是它的概率，以此类推。

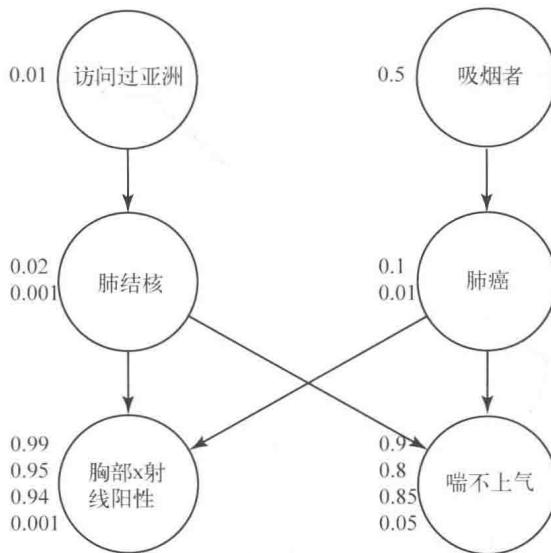


图 12-2 一个贝叶斯网络

如果每个顶点都它自己的处理器，能够向其他顶点的处理器发送消息，那当了解到病人的胸部 x 射线为阳性时，可以首先计算并向“肺结核”和“肺癌”发送消息。当这些顶点接收到它的消息时，可以独立计算并向其父顶点和其他子顶点发送消息。此外，如果还知道这位病人是一位吸烟者，那包含“吸烟者”的顶点可以同时计算并向其子节点发送消息。显然，如果所有这些都同时发生，那推理过程的完成要快得多。在实际应用中使用的贝叶斯网络经常会包含数百个顶点，而且马上就会用到推理出来的概率。这意味着，节省下来的时间可能是非常可观的。

我们刚刚描述的是一种称为并行计算机 (parallel computer) 的体系结构。这种计算机称为“并行”，是因为每个处理器都与所有其他处理器同时（并行）执行指令。在过去三十年里，处理器的成本已经大幅下降。当前，商用微处理器的速度与最快速的串行计算机在同一量级。但微处理器的成本要低许多个量级。因此，如果像上一段描述的那样，将微处理器连接在一起，所得到的计算能力要比最快速的串行计算机还要快，而成本却要低得多。有许多应用都可以从这种并行计算中获得很大的好处。人工智能中的应用包含前面介绍的贝叶斯网络问题、神经网络中的推理、自然语言的理解、语音识别和机器视觉。其他一些应用包含数据库查询处理、天气预报、污染监测、蛋白质结构分析和其他许多应用。

有许多方法可以设计并行计算机。12.1 节讨论了在并行设计中需要注意的一些事项，还有最流行的一些并行体系结构。12.2 节介绍了如何为一种特定类别的并行计算机编写算法，这种计算机称为并行随机访问计算机 (PRAM, Parallel Random Access Machine)。后面将会看到，这种计算机并不是特别实用。但是，PRAM 模型是串行计算模型的直接推广。此外，RPAM 算法可以转化为许多实用计算机的算法。所以 PRAM 算法非常适合用来介绍并行算法。

## 12.1 并行体系结构

并行计算机的构造可以在以下三个方面有所不同：

- (1) 控制机制；
- (2) 地址空间的组织；
- (3) 互联网络。

### 12.1.1 控制机制

并行计算机中的每个处理器既可以在一个中心控制单元的控制下工作，也可以在自己控制单元的控制下独立工作。第一种体系结构称为单指令流、多数据流（SIMD）。图 12-3a 展示了一种 SIMD 体系结构。图中介绍的互联网络表示能使处理器相互通信的硬件。互联网络在 12.1.3 节中讨论。在 SIMD 体系结构中，所有处理单元可以在中心控制单元的控制下同步执行相同指定。并非所有片器都必须在每个周期中执行一条指令，任何给定处理器都可以在任意给定期限内被关闭。

如果并行计算机的每个控制器都有自己的控制单元，那就称之为多指令流、多数据流（MIMD）计算机。图 12-3b 展示了一种 MIMD 体系结构。MIMD 体系结构在每个处理器上都存储着操作系统和程序。

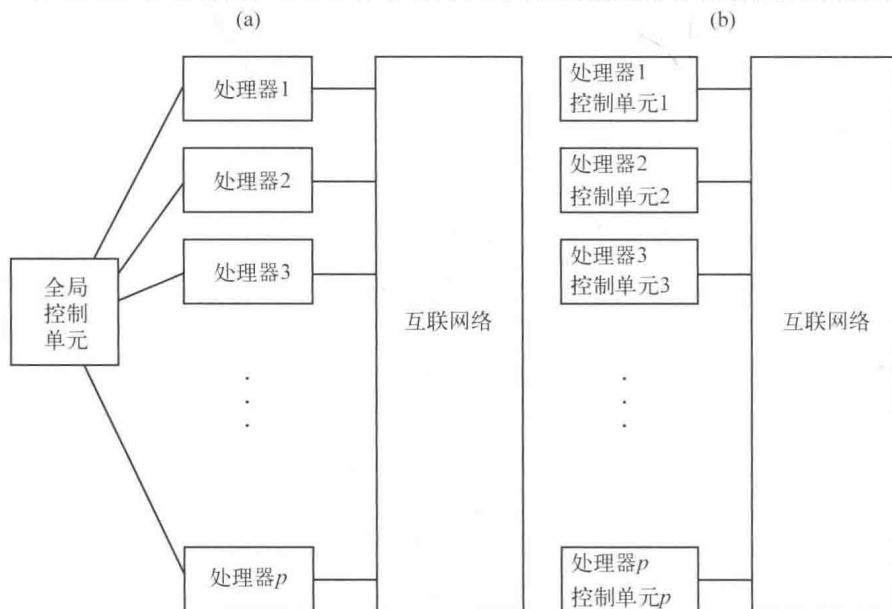


图 12-3 (a) 单指令流多数据流 (SIMD) 体系结构。(b) 多指令流多数据流 (MIMD) 体系结构

SIMD 处理器适合那些对一个数据集中的不同元素执行同一指令集的程序。这种程序称为 **数据并行程序**。 SIMD 计算机的一个缺点就是它们不能在同一周期内执行不同指令。例如，假定执行下面的条件语句：

```
if (x==y)
    执行指令 A;
else
    执行指令 B;
```

任何发现  $x \neq y$  的处理器（别忘了，这些处理器是在处理不同的数据元素）都不能做任何事情，而发现  $x=y$  的处理器正在执行指令 A。随后那些发现  $x=y$  的处理器必须空闲，而其他处理器则执行指令 B。

一般情况下，SIMD 计算机最适合那些需要同步的并行算法。许多 MIMD 计算机都有另外的硬件提供同步，这意味着它们可以模拟 SIMD 计算机。

### 12.1.2 地址空间的组织

处理器之间可以通过修改一个公共地址空间的数据或者通过传送消息来通信。根据所使用的通信方法，地址空间可采用不同的组织方式。

#### 1. 共享地址空间体系结构

在共享地址空间体系结构 (shared-address-space architecture) 中，由硬件负责所有处理器对一个共享地址空间的读写访问。处理器通过修改共享地址空间中的数据进行通信。图 12-4a 给出了一种共享地址空间体系结

构，其中每个处理器访问内存中任意字的时间都是相同的。这种计算机称为统一内存访问（UMA，Uniform Memory Access）计算机。在 UMA 计算机中，每个处理器都拥有自己的专用内存，如图 12-4a 所示。这一专用内存仅用于存储该处理器执行运算时所需要的局部变量，算法的实际输入都不会存储在专用区域。UMA 计算机的一个缺点就是互联网络必须同时为所有处理器提供对共享内存的访问，这样可能会使性能大幅降低。另一种方案是为每个处理器都提供共享内存的一部分。图 12-4b 中显示了这种方案。这个内存不是专用的，与图 12-4a 中的本地内存不同。也就是说，每个处理器都可以访问另一个处理器的内存。但是，它访问自己内存的速度要远快于访问另一个处理器的内存。如果一个处理器的大多数访问都是对自己内存的访问，那性能应当会很好。这种计算机称为非统一内存访问（NUMA，Non-Uniform Memory Access）计算机。

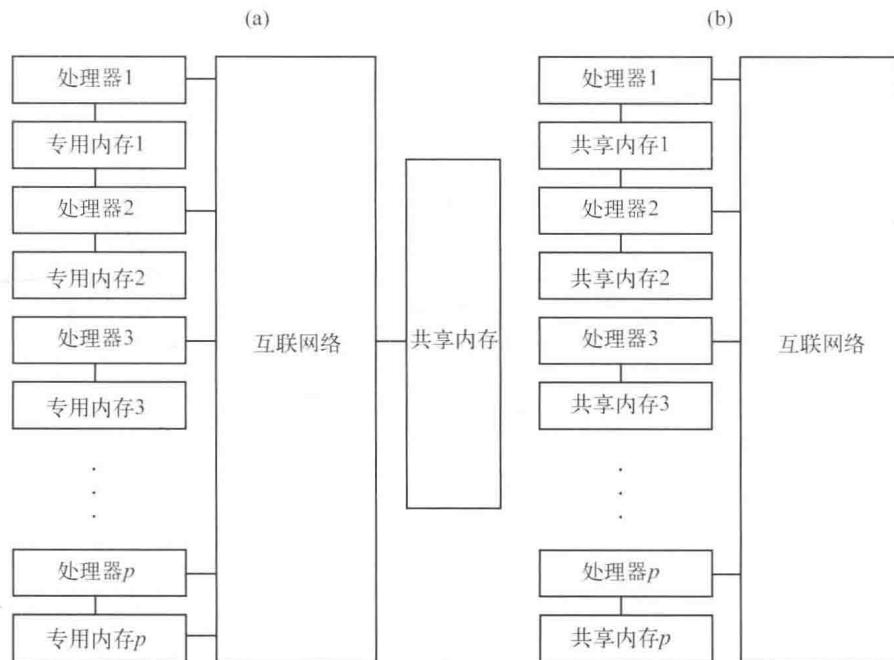


图 12-4 (a) 统一内存访问 (UMA) 计算机；(b) 非统一内存访问 (UMA) 计算机

## 2. 消息传送体系结构

在消息传送体系结构 (message-passing architecture) 中，每个处理器都有自己的专用内存，而且只能供该处理器访问。处理器通过向其他处理器发送消息来进行通信，而不是通过修改数据元素。图 12-5 给出了一种消息传送体系结构。注意，图 12-5 看起来与图 12-4b 非常类似，区别在于互联网络的连线方式。在 NUMA 计算机中，互联网络的连线使每个处理器都能访问其他处理器的内存，而在消息传送计算机中，它的连线允许每个处理器直接向其他每个处理器发送消息。

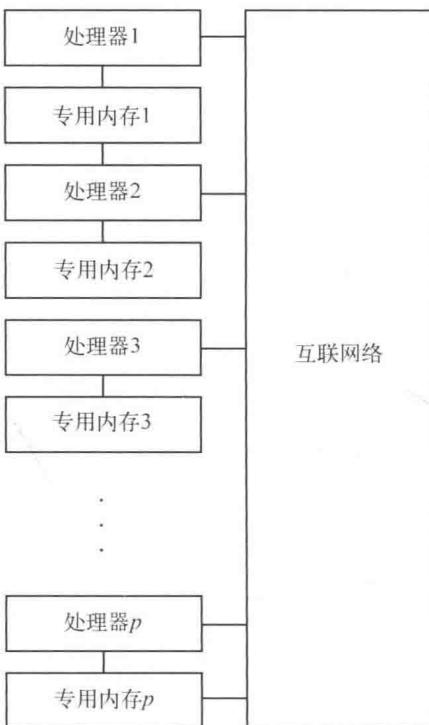


图 12-5 一种消息传送体系结构。每个处理器的内存只能供该处理器访问。处理器通过互联网络相互发送消息，以此实现通信

### 12.1.3 互联网络

互联网络有两大类别：静态的和动态的。静态网络通常用于构建消息传送体系结构，而动态网络通常用于构建共享地址空间体系结构。下面依次来讨论这两种网络类别。

#### 1. 静态互联网络

静态互联网络包含了处理器之间的直接连接，有时称为直接网络。有几种不同类型的静态互联网络。我们讨论最常见的一些。最高效但也最昂贵的是完全连通网络，如图 12-6a 所示。在这种网络中，每个处理器都直接连接到所有其他处理器。因此，一个处理器可以通过该处理器上的连接直接向另一个处理器发送消息。因为连接的个数是处理器数量的平方，所以这种网络的成本非常昂贵。

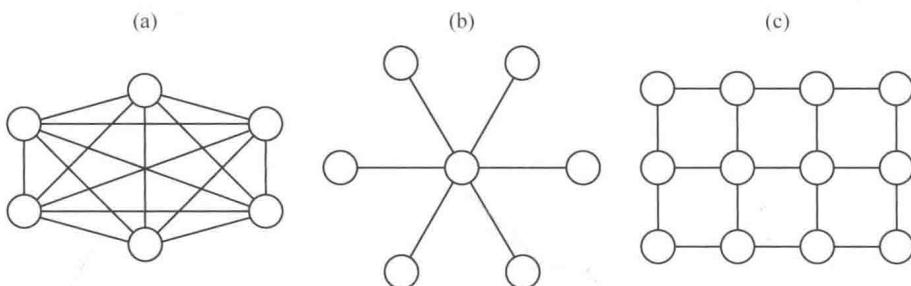


图 12-6 (a) 完全连通网络；(b) 星型网络；(c) 一个度数为 4 的度数有限网络

在星型网络中，一个处理器充当中心处理器。这就是说，其他每个处理器都只有连向该处理器的一条连接。图 12-6a 描述了一个星型网络。在星型网络中，处理器要向另一处理器发送消息，它先向中心处理器发送消息，再由中心处理器将该消息转发给要接收的处理器。

在一个度数为  $d$  的度数有限网络中，每个处理器最多连到  $d$  个其他处理器。图 12-6c 给出了一个度数为 4

的度数有限网络。在一个度数为 4 的度数有限网络中，要传送一条消息，可以首先沿一个方向传送它，然后再沿另一个方向发送，直到到达自己的目的地为止。

一种稍为复杂但更流行的静态网络是超立方体。零维超立方体由单个处理器组成。一维超立方体通过连接两个维超立方体中的处理器构成。二维超立方体的构建方法是将一个一维超立方体中的每个处理器连接到另一个一维超立方体中的一个处理器。利用这种递归方式， $d+1$  维超立方体的构建方法就是将一个  $d$  维超立方体中的每个处理器连接到另一个  $d$  维超立方体中的一个处理器。第一个超立方体中的给定处理器连接到第二个超立方体中占据对应位置的处理器。图 12-7 展示了超立方体网络。

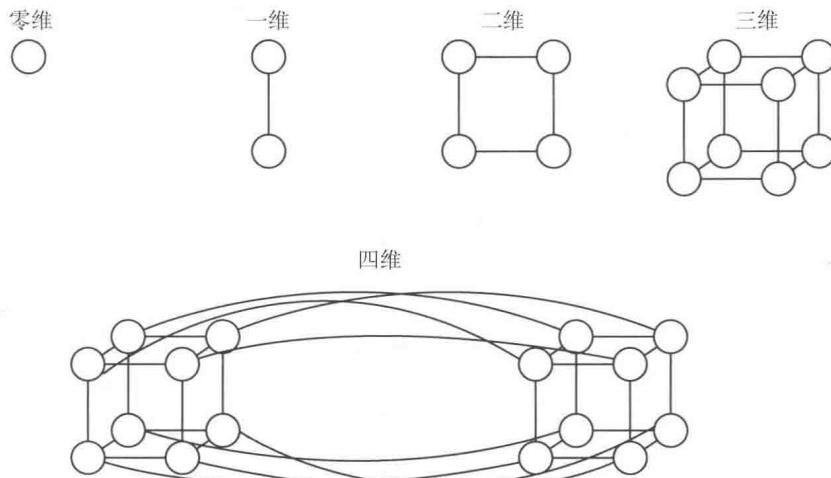


图 12-7 超立方体网络

应当清楚，静态网络通常用于实现消息传送体系结构的原因在于，这种网络中的处理器是直接连接，支持消息的流动。

## 2. 动态互联网络

在动态互联网络中，处理器通过一组交换元件连接到内存。这样做的最直接方法之一就是使用交叉开关网络。在这种网络中， $p$  个处理器通过一个交换元件栅格连接到  $m$  个内存组，如图 12-8 所示。例如，如果处理器 3 当前要访问内存组 2，图 12-8 中加圈栅格位置处的开关被合上（合上开关使电路变得完整，从而允许电流流动）。这个网络称为“动态的”，是因为处理器与内存组之间的连接是在合上一个开关时动态建立的。交叉开关网络是非阻塞的。也就是说，一个处理器到一个给定内存组的连接不会阻塞另一个处理器到另一个开关的连接。理想情况下，在交叉开关网络中，对于内存中的每个字都应当有一个组。但是，这显然是不现实的。通常情况下，内存组数至少与处理器个数一样多，这样，在给定时刻，每个处理器都至少能访问一个内存组。交叉开关网络中的开关数等于  $pm$ 。也就是说，如果需要  $m$  大于或等于  $p$ ，开关数就会大于或等于  $p^2$ 。因此，当处理器的数量很大时，交叉开关网络可能会变得非常昂贵。

其他一些动态互联网络（本书不做讨论）包含基于总线的网络和多级互联网络。

通常使用动态互联网络来实现共享地址空间体系结构，其原因应当很明显了。也就是说，在这样一个网络中，每个处理器都可以访问内存中的每个字，但不能向任意其他处理器发送直接消息。

这里对并行硬件的介绍主要基于 Kumar、Grama、Gupta 和 Karypis (1994 年) 的讨论。更详尽的介绍，特别是对基于总线和多项互联网络的介绍，请参阅该书。

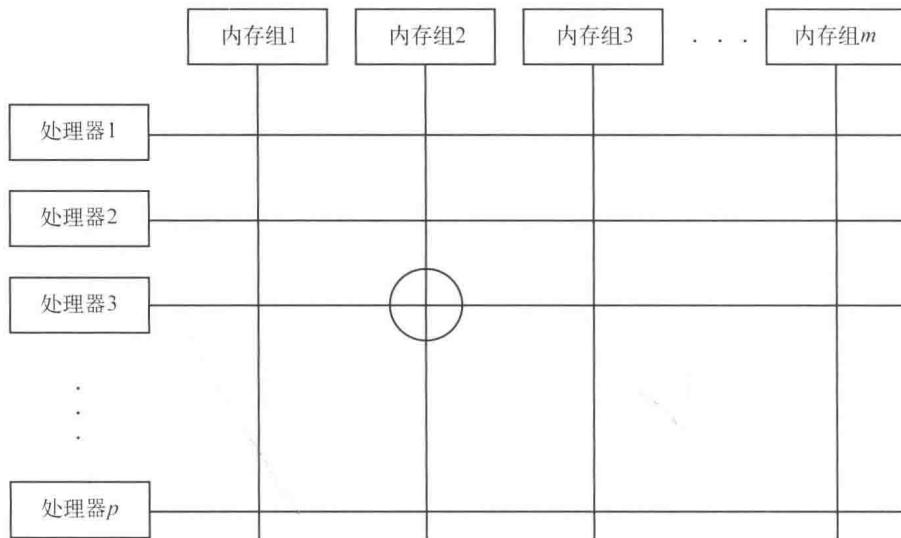


图 12-8 交叉开关网络。在网格上的每个位置都有一个开关。带圆圈的开关是闭合的，当前允许第三个处理器访问第二个内存组，使信息可以在处理器 3 和内存组 2 之间流动

## 12.2 PRAM 模型

如前一节的讨论，可能存在相当多不同的并行体系结构，实际上，已经制造了具有许多此类体系结构的计算机。而另一方面，所有串行计算机都具有如图 12-1 所示的体系结构，这就是说，冯·诺依曼模型是所有串行计算机的通用模型。在设计前面各章的算法时，唯一的假设就是它们将在一个符合冯·诺依曼模型的计算机上运行。因此，对于其中每个算法，无论使用什么样的程序设计语言或计算机来实现，其时间复杂度都是一样的。串行计算机的应用之所以增长迅速，这是一个关键因素。

为并行计算找到一种通用模型会是很有用的。任何此种模型都必须足够通用，以包含一大类并行体系结构的关键特性。再者，根据这一模型设计的算法必须能够在实际的并行计算机上高效运行。但目前还没人知道这种模型，似乎也不大可能会找到一种。

尽管目前没人知道通用模型，但并行随机访问机器（PRAM, Parallel Random Access Machine）计算机已经被广泛用作并行机器的理论模型了。PRAM 计算机包括  $p$  个处理器，所有这些处理器都可以统一访问一个大型共享内存。处理器共享一个公共时钟，但在每个周期内可以执行不同指令。因此，PRAM 计算机是一个同步的 MIMD、UMA 计算机。这意味着图 12-3b 和 12-4a 描述了一个 PRAM 计算机的体系结构，而图 12-8 则给出了这样一个机器的互联网络。前面已经提到，真正构建这样一台计算机可能是非常昂贵的。但是，PRAM 模型是串行计算模型的自然延伸。这样，在设计算法时，PRAM 模型在概念上非常容易理解。此外，为 PRAM 模型开发的算法可以转换成为许多更实用计算机使用的算法。例如，在一个度数有限网络中，可以用  $\Theta(\lg p)$  条指令模拟一条 PRAM 指令，其中  $p$  是处理器数量。此外，对于一大类问题，PRAM 算法的速度与超立方的算法渐近相同。基于这些原因，PRAM 模型非常适合用于介绍并行算法。

在一台诸如 PRAM 之的共享内存计算机中，可以有多个处理器同时读取或写入同一个内存位置。根据处理并发内存访问的方式，PRAM 模型存在四种版本。

(1) 独占读，独占写 (EREW, Exclusive-Read Exclusive-Write)。在该版本中，不允许并发读取或写入。也就是说，在给定时刻，只允许一个处理器访问一个给定内存位置。这是最弱版本的 PRAM 计算机，因为它允许的并发最少。

(2) 独占读，并发写 (ERCW, Exclusive-Read Concurrent-Write)。在该版本中，允许同时写入操作，但不允许同时读取操作。

(3) 并发读，独占写（CREW，Concurrent-Read Exclusive-Read）。在该版本中，允许同时读取操作，但不允许同时写入操作。

(4) 并发读，并发写（CRCW，Concurrent-Read Concurrent-Write）。在该版本中，同时读取操作和同时写入操作都是允许的。

我们讨论 CREW PRAM 模型和 CRCW PRAM 模型的算法设计。首先讨论 CREW 模型，然后说明使用 CRCW 模型开发的算法有时是多么高效。在继续深入之前，先来讨论如何表示并行算法。尽管存在用于并行算法的程序设计语言，但我们还是会使用具有某些附加特性的标准伪代码，后面将会对它们进行讨论。

只需要编写算法的一个版本，在编译之后，就能由所有处理器同时执行。因此，每个处理器在执行算法时都需要知道自己的索引。我们假定处理器的索引编号为  $P_1, P_2, P_3$ ，等等，而下面的指令

$p = \text{本处理器的索引};$

返回一个处理器的索引。在算法中声明的变量可以是共享内存中的一个变量，这意味着它可以供所有处理器访问；当然也可以在专用内存中（见图 12-4a）。在后一种情况下，每个处理器都有属于自己的变量副本。在声明这样一个变量时，使用关键字 `local`。

我们的所有算法都将是数据并行算法，如 12.1.1 节中的讨论。也就是说，这些处理器会对一个数据集的不同元素执行相同指令集。这个数据集将存储在共享内存中。如果一条指令将这一数据集中一个元素的值指定给一个局部变量，就将这一过程称为读取共享内存，如果它将一个局部变量的值指定给这一数据集的一个元素，就将这一过程称为写入共享内存。我们用于操作这一数据集元素的指令只有读取和写入共享内存。例如，我们从来不会直接比较数据集中两个元素的值，而是会将它们的值读入本地内存中的变量，然后比较这些变量的值。我们将允许直接比较一些变量，比如  $n$ ——数据集的大小。数据并行算法包含一系列步骤，每个处理器同时启动每个步骤，也同时终止每个步骤。此外，所有在一个给定步骤进行读取的处理器会同时读取，所有在一个给定步骤进行写入的处理器会同时写入。

最后，假定我们想用多少个处理器，就可以用多少个。前面已经提到，在实践中，这通常是一个不切实际的假设。

下面的算法演示了这些约定。假定在共享内存中有一个整数数组  $S$ ，索引范围为 1 至  $n$ ；假定有  $n$  个处理器，索引范围为 1 至  $n$ ，它们并行执行该算法。

```
void example (int n,
              int S[])
{
    local index p;
    local int temp;

    p = 本处理器的索引;
    将 S[p] 读入 temp;           // 这是对共享内存的读取。
    if (p < n)
        将 temp 写入 S[p+1];     // 这是对共享内存的写入。
    else
        将 temp 写入 S[1];
}
```

数组  $S$  中的所有值都被同时读入  $n$  个不同的局部变量  $temp$  中。然后，再将  $n$  个  $temp$  中的值全部同时写回  $S$  中。 $S$  中的每个元素都被指定了它前面一个元素的值（采用回绕方式）。图 12-9 演示了该算法的操作。注意，每个处理器总是可以访问整个数组  $S$ ，因为  $S$  是在共享内存中。所以，第  $p$  个处理器可以写入第  $p+1$  个数组位置。

这个简单的算法中只有一个步骤。在有多个步骤时，我们编写下面的循环：

```
for (step = 1; step <= numsteps; step++) {
    // 在每个步骤中都要执行的代码放在这里。每个处理器同时启动每个步骤，同时结束每个步骤。
    // 所有在一个给定步骤进行读取的处理器会同时读取，
    // 所有在一个给定步骤进行写入的处理器会同时写入。
}
```

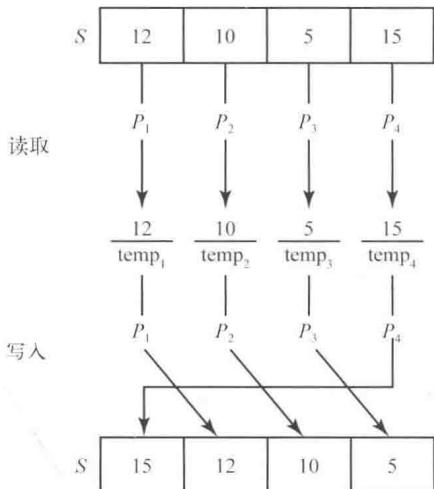


图 12-9 过程应用举例

这个循环可以有多种实现方式。一种方式是让一个独立控制单元来进行递归和判断。它会发出指令，告诉其他处理器何时读取、何时对局部变量执行指令、何时写入。在循环内部，有时会对一个在所有处理器中都取相同值的变量进行计算。例如，算法 12.1 和算法 12.3 都执行以下指令：

```
size=2*size;
```

其中  $size$  对于所有处理器都取相同值。对于这种变量，并不需要每个处理器都拥有自己的副本，为此，我们将这种变量声明为共享内存中的一个变量。作为一种实现方式，可以让一个独立控制单元来执行它。我们将不再深入讨论实现，而是开始编写算法。

### 12.2.1 为 CREW PRAM 模型设计算法

我们用下面的示范问题来演示 CREW PRAM 算法。

#### 1. 找出数组中的最大键

定理 8.7 证明：仅通过键的比较，至少需要  $n-1$  次比较才能找出最大键，这意味着该问题的任意算法，只要它是设计用来在串行计算机上运行的，那必然为  $\Theta(n)$ 。利用并行计算，可以缩短这一运行时间。并行算法仍然必须进行至少  $n-1$  次比较。但通过并行完成其中许多比较，就能在更短的时间内完成。下面就来开发这一算法。

回想一下，算法 8.2（查找最大键）在最优时间内找到了最大键，如下所示：

```
void find_largest(int n,
                  const keytype S[],
                  keytype& large)
{
    index i;

    large = S[1];
    for (i = 2; i<=n; i++)
        if (S[i]>large)
            large = S[i];
}
```

这个算法不能通过使用更多处理器而获益，因为每次循环迭代的结果都是下一次迭代所需要的。回想一下 8.5.3 节中查找最大键的锦标赛方法。这一方法将数字两两配对，找出每一对的最大值（获胜者）。然后它将获胜者配对，再找出每一对的最大者。一直持续这一过程，直到只剩下一个键为止。图 8-10 演示了这一方法。锦标赛方法的串行算法与算法 8.2 具有相同的时间复杂度。但是，这一方法可以通过使用多个处理器而获益。例如，假定你希望使用这一方法找出 8 个键中的最大者。必须首先在第一轮中依次确定 4 个获胜者，然后才能

进入第二轮。如果有三个朋友可以帮你，你们每个人可以同时确定第一轮中的一位获胜者。这意味着第一轮的完成速度提高四倍。在此轮之后，你们中的两个可以休息，而另外两个人进行第二轮的计算。在最后一轮中，你们当中只需要一个人进行比较。

图 12-10 演示了可以使此方法进行下去的并行算法。所需要的处理器个数只有数组元素的一半。每个处理器将两个数组元素读入局部变量 first 和 second 中。然后，将 first 和 second 中的较大者写入前面读取的第一个数组位置中。在经过这样三轮之后，最大键将放在  $S[1]$  中。每一轮都是算法中的一个步骤。在图 12-10 所示的例子中， $n=8$ ，共有  $\lg 8=3$  个步骤。算法 12.1 是图 12-10 所示操作的算法。注意，这个算法被写为一个函数。在将一个并行算法写为函数时，至少一个处理器需要返回一个值，所有要返回值的处理器都需要返回相同值。

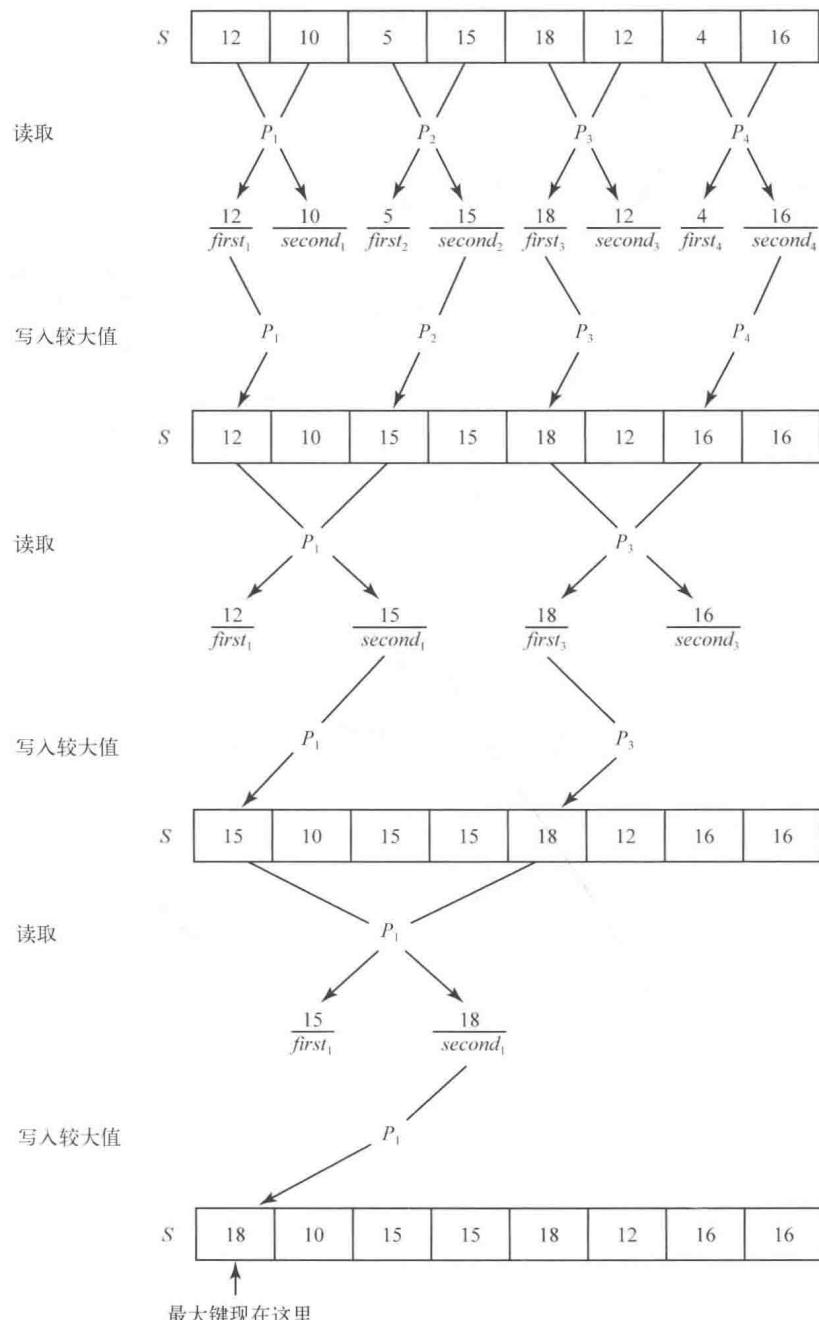


图 12-10 使用并行处理器实现查找最大键的锦标赛方法

### 算法 12.1 并行查找最大键

问题：在大小为  $n$  的数组  $S$  中找出最大键。

输入：正整数  $n$ ；键的数组  $S$ ，索引范围为 1 至  $n$ 。

输出： $S$  中最大键的值。

备注：假定  $n$  是 2 的幂，我们有  $n/2$  个并行执行该算法的处理器。这些处理器的索引范围为 1 至  $n/2$ ，命令“本处理器的索引”返回一个处理器的索引。

```
keytype parlargest (int n, keytype S[])
{
    index step, size;
    local index p;
    local keytype first, second;

    p = 本处理器的索引;
    size = 1; // size 是子数组的大小。
    for (step = 1; step <= lgn; step++)
        if (这个处理器需要在这个 step 中执行){
            将 S[2*p-1] 读入 first;
            将 S[2*p-1+size] 读入 second;
            将 maximum(first, second) 写入 S[2*p-1];
            size = 2 * size;
        }
    return S[1];
}
```

我们使用顶级伪代码“`if (这个处理器需要在这个 step 中执行)`”，使这个算法尽可能保持清晰。在图 12-10 中，我们看到在一个给定步骤使用的处理器满足以下条件：对于某一整数  $k$ ，

$$p = 1 + \text{size} * k$$

(注意，在每一步中，size 值都会加倍)。因此，真正要对处理器进行的检查是：

```
if ((p-1)%size == 0) // % 返回用 p-1 除以 size 时的余数。
```

或者，可以直接允许所有处理器在每一步中执行。不需要执行的处理器就做一些没有任何用途的比较。例如，在第二轮中，处理器  $P_2$  将  $S[3]$  的值与  $S[5]$  的值进行比较，并将较大值写入  $S[3]$ 。尽管并非必要，但  $P_2$  也可以这样做，因为让  $P_2$  保持空闲不会有任何收益。最重要的事情是，应当运行的处理器正在运行，而且，这些处理器所需要的内存位置的值，也没有被其他处理器改变。当允许不必要的处理器执行指令时，唯一的问题就是它们有时会引用  $S$  范围之外的数组元素。例如，在前面的算法中， $P_4$  在第二轮中引用了  $S[9]$ 。只需要为  $S$  补充一些位置就可以解决这一问题。这样会浪费一些空间，但节省了用于检查处理器是否应当运行的时间。

在分析并行算法时，不是分析该算法完成的总工作量，而是分析任意一个处理器完成的总工作量，因为这样可以让我们很好地了解计算机在处理输入时有多快。因为每个处理器都会将算法 12.1 中的 for-step 循环执行大约  $\lg n$  遍，所以有

$$T(n) \Theta(\lg n)$$

相对于串行算法，这是一个很大的改进。

## 2. 动态规划的应用

许多动态规划应用都可修改为并行设计，因为一个给定行或对角线中的项目通常是可以同时计算的。为演示这一方法，我们将二项式系数的算法（算法 3.2）改写为并行算法。在这一算法中，帕斯卡三角（见图 3-1）中一个给定行中的项目是并行计算的。

### 算法 12.2 并行二项式系数

问题：计算二项式系数。

输入：非负整数  $n$  和  $k$ ，其中  $k \leq n$ 。

输出：二项式系数  $\binom{n}{k}$ 。

备注：假定有  $k+1$  个处理器并行执行算法。这些处理器的索引为 0 至  $k$ ，命令“本处理器的索引”返回一个处理器的索引。

```
int parbin(int n, int k)
{
    index i;                      // 使用 i 替代 step 来控制步骤，以与算法 3.2 保持一致。
    int B[0..n][0..k];
    local index j;                // 使用 j 替代 p 来获得处理器的索引，以与算法 3.2 保持一致。
    local int first, second;

    j = 本处理器的索引;
    for (i = 0; i <= n; i++)
        if (j <= 0 = minimum(i, k))
            if (j == 0 || j == i)
                将 1 写至 B[i][j];
            else{
                将 B[i-1][j-1] 写入 first;
                将 B[i-1][j] 写入 second;
                将 first + second 写入 B[i][j];
            }
    return B[n][k];
}
```

算法 3.2 中的控制语句

```
for (j=0; j<=minimum(i, k); j++);
```

在这个算法中用下面的控制语句代替：

```
if (j <= minimum(i, k));
```

因为所有  $k$  个处理器在每遍执行 for- $i$  循环时都会运行。并行算法不是依次计算  $B[i][j]$  的值 ( $j$  的变化范围为 0 至  $\min(i, k)$ )，而是让索引范围为 0 至  $\min(i, k)$  为处理器同时计算这些值。

显然，我们的并行算法中将一个循环执行  $n+1$  遍。回想一下，在串行算法（算法 3.2）中，一个循环要执行  $\Theta(nk)$  遍。

回想一下习题 3.4，有可能仅使用一个索引范围为 0 至  $k$  的一维数组  $B$  来实现算法 3.2。在并行算法中，这一修改非常简单，因为在进入 for- $i$  循环的第  $i$  遍执行时，帕斯卡三角的整个第  $i-1$  行都可以从  $B$  中读入到  $k$  个局部变量对  $first$  和  $second$  中。然后，在退出时，可以将整个第  $i$  行写入  $B$  中。伪代码如下：

```
for (i = 0; i <= n; i++)
    if (j <= minimum(i, k))
        if (j == 0 || j == i)
            将 1 写至 B[j];
        else {
            将 B[j-1] 读入 first;
            将 B[j] 读入 second;
            将 first+second 写入 B[j] 中;
        }
    return B[k];
```

### 3. 并行排序

回想 Mergesort3 的动态规划版本（算法 7.3）。这个算法在开始时将各个键看作单个的，然后将键对合并到包含两个键的有序列表中，再将这些列表对合并到包含四个键的有序列表中，以此类推。也就是说，它执行图 2-2 所示的合并过程。这一过程非常类似于使用锦标赛方法查找最大值。也就是说，可以在每一步并行执行合并。下面的算法实现了这一方法。为简单起见，再次假定  $n$  是 2 的幂。如果它不是 2 的幂，可以将数组大小看作 2 的幂，但对于超过  $n$  的部分不进行合并。Mergesort3 的动态规划版本（算法 7.3）显示了如何进行这一过程。

### 算法 12.3 并行合并排序

问题：将  $n$  个键排列为非递减顺序。

输入：正整数  $n$ ；键的数组  $S$ ，其索引范围为 1 至  $n$ 。

输出：数组  $S$ ，其中的键按非递减顺序排列。

备注：假定  $n$  是 2 的幂，且有  $n/2$  个处理器并行执行该算法。处理器的索引范围为 1 至  $n/2$ ，命令“本处理器的索引”返回一个处理器的索引。

```

void parmergesort (int n, keytype S[])
{
    index step, size;
    local index p, low, mid, high;
    p=本处理器的索引;
    size =1;                                // 所合并子数组的大小。
    for (step = 1; step <= lgn; step++){
        if (本处理器需要在这一步骤中执行){
            low = 2*p -1;
            mid = low + size -1;
            high = low + 2*size -1;
            parmerge(low, mid, high, S);
            size = 2*size;
        }
    }

    void parmerge(local index low, local index mid,
                 local index high, keytype S[])
    {
        local index i, j, k;
        local keytype first, second, U[low..high];
        i = low; j=mid+1; k=low;
        while (i<=mid && j<=high){
            将 S[i]读入 first;
            将 S[j]读入 second;
            if (first < second){
                U[k]=first;
                i++;
            }
            else{
                U[k]=second;
                j++;
            }
            k++;
        }
        if (i>mid)
            将 S[j]至 S[high]读入 U[k]至 U[high];
        else
            将 S[i]至 S[mid]读入 U[k]读入 U[high];
            将 U[low]至 U[high]写入 S[low]至 S[high];
    }
}

```

关于一个处理器是否应当在一个给定步骤执行的检查，与算法 12.1 中的相同。也就是说，需要进行以下检查：

```
if((p-1)% size == 0) // %返回 p-1 除以 size 时的余数
```

回想一下，在合并排序的单处理器迭代版本（算法 7.3 中）中，在每遍执行 **for** 循环时颠倒  $U$  和  $S$  的角色，减少了记录赋值的数目。这里可以进行同样的改进。如果进行了这一改进， $U$  会是共享内存中的一个数组，索引范围为 1 至  $n$ 。出于简单考虑，我们给出 **parmerge** 的基本版本。

这个算法的时间复杂度不是那么明显。所以下面对其正式分析。

### 算法 12.3 的分析 最差时间复杂度

基本运算：parmerge 中发生的比较。

输入规模： $n$ ，数组中的键数。

这一算法执行的比较次数与普通串行合并排序相同，区别在于这些比较有许多是并行完成的。在第一遍执行 for-step 循环时，同时合并  $n/2$  对各包含一个键的数组。所以，任意处理器在最差情况下执行的比较次数为  $2-1=1$ （见 2.2 节算法 2.3 中的分析）。在第二遍执行中，同时合并  $n/4$  对各包含两个键的数组。所以最差情况下的比较次数为  $4-1=3$ 。在第三遍执行中，同时合并  $n/8$  对各包含四个键的数组。所以最差情况下的比较次数为  $8-1=7$ 。一般来说，在第  $i$  遍执行中，同时合并  $n/2^i$  对各包含  $2^{i-1}$  个键的数组，最差情况下的比较次数为  $2^i-1$ 。最后，在最后一遍执行时，将合并两个各包含  $n/2$  个键的数组，这意味着这一遍在最差情况下的比较次数为  $n-1$ 。每个处理器在最差情况下执行的总比较次数为：

$$\begin{aligned} W(n) &= 1 + 3 + 7 + \dots + 2^i - 1 + \dots + n - 1 \\ &= \sum_{i=1}^{\lg n} (2^i - 1) = 2n - 2 - \lg n \in \Theta(n) \end{aligned}$$

最后一个等式是根据附录 A 中例 A.3 的结果并进行一些代数运算得到的。

我们已经在线性时间内，仅通过键的比较成功地完成了并行排序，相对于串行排序所需要的  $\Theta(n \lg n)$ ，这是一个重大改进。有可能改进我们的并行合并算法，使并行合并排序在  $\Theta((\lg n)^2)$  时间内完成。这一改进在习题中讨论。但这不是最优的，因为并行排序可以在  $\Theta(\lg n)$  时间内完成。关于并行排序的详尽讨论，请参阅 Kumar、Grama、Gupta 和 Karypis (1994 年) 或 Akl (1985 年) 的文献。

## 12.2.2 为 CRCW PRAM 模型设计算法

回想一下，CRCW 表示并发读取并发写入。与并发读不同，当两个处理器尝试在同一步骤中写入同一内存位置时，必须解决并发写入问题。解决这种冲突的最常用协议如下。

- **普通。**这一协议仅在所有处理器尝试写入相同值时才允许并发写入。
- **任意。**这一协议任意选择一个处理器，允许它写入内存位置。
- **优先级。**在这一协议中，所有处理器都放在一个预先定义的优先级列表中，只允许具有最高优先级的一个写入。
- **求和。**这一协议对各处理器正要写入的数量求和，并写入该和值。（可以扩展这一协议，使用任意针对要写入数量定义的结合运算符。）

我们编写了一个在数组中查找最大键的算法，它使用普通写入、任意写入和优先级写入协议，而且要快于前面针对 CREW 模型给出的算法（算法 12.1）。此算法的过程如下。设共享内存的数组  $S$  中有  $n$  个键。共享内存中还维护着一个包含  $n$  个整数的第二数组  $T$ ，并将  $T$  中的所有元素都初始化为 1。接下来，假定有  $n(n-1)/2$  个处理器，其索引如下：

$$P_{ij} \quad (1 \leq i < j \leq n)$$

让所有处理器并行比较  $S[i]$  和  $S[j]$ 。这样， $S$  中的每个元素都与  $S$  中的任意其他元素相比较。如果  $S[i]$  输掉比较，每个处理器将 0 写入  $T[i]$ ；如果  $S[j]$  输掉，则将 0 写入  $T[j]$ 。只有最大键从来不会输掉比较。因此， $T$  中唯一等于 1 的元素就是索引为  $k$  ( $S[k]$  中包含最大键) 的元素。所以该算法只需要返回使  $T[k]=1$  的  $S[k]$  的值。图 12-11 演示了这些步骤，算法如下。在此算法中注意到，当多个处理器写入同一内存位置时，它们都写入相同值。这意味着该算法使用普通写入、任意写入、优先级写入协议。

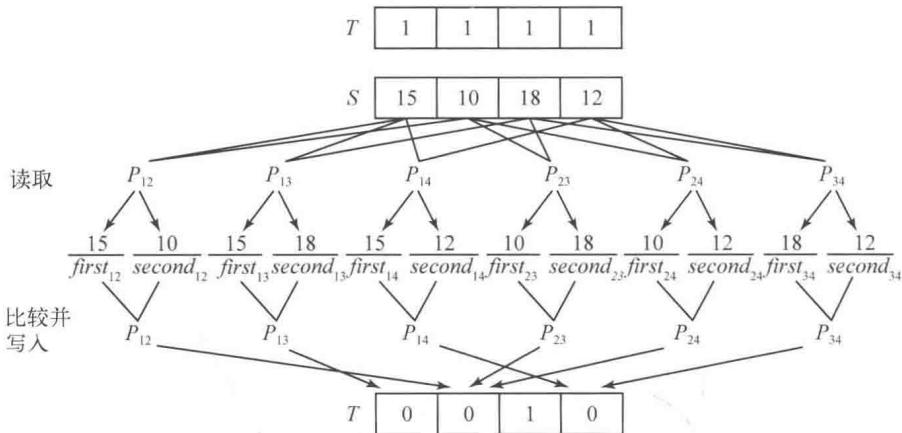


图 12-11 算法 12.4 的应用。只有  $T[3]$  最终等于 1，因为  $S[3]$  是最大键，它是唯一从未输过一次比较的键。

#### 算法 12.4 并行 CRCW 查找最大键

问题：在一个包含  $n$  个键的数组  $S$  中找出最大键。

输入：正整数  $n$ ；键的数组  $S$ ，索引范围为 1 至  $n$ 。

输出： $S$  中最大键的值。

备注：假定  $n$  是 2 的幂，我们有  $(n-1)/2$  个并行执行该算法的处理器。这些处理器的索引范围为

$$P_{ij} \quad (1 \leq i < j \leq n)$$

命令“本处理器的第一个索引”返回  $i$  的值，而命令“本处理器的第二个索引”返回  $j$  的索引。

```

keytype parlargest2(int n, const keytype S[])
{
    int T[1..n];
    local index i, j;
    local keytype first, second;
    local int chkfrst, chksnd;

    i=本处理器的第一个索引;
    j=本处理器的第二个索引;
    将 1 写入 T[i];           // 因为 1 ≤ i ≤ n-1 且 2 ≤ j ≤ n，这些写入指令将 T 的每个数组元素初始化为 1。
    将 1 写入 T[j];
    将 S[i] 读入 first;
    将 S[j] 读入 second;
    if (first < second)
        将 0 写入 T[i];           // 当且仅当 S[k] 至少输掉一次比较时，T[k] 最终为 0。
    else
        将 0 写入 T[j];
    将 T[i] 读入 chkfrst;
    将 T[j] 读入 chksnd;
    if (chkfrst == 1)          // 当且仅当 S[k] 中包含最大键时，T[k] 仍然等于 1。
        return S[i];
    else if (chksnd = 1)        // 当最大键为 S[n] 时，需要检查 T[j]。
        return S[j];             // 回想一下，i 的取值范围仅为 1 至 n-1。
}

```

此算法中没有循环，这就是说，它在常量时间内找出最大键。这一点让人印象深刻，因为这意味着它在查找 1 000 000 个键的最大值时，所需要的时间与仅查找 10 个键的最大值一样。但是，这一最优时间复杂度是以平方时间的处理器复杂度为代价的。我们需要大约  $1 000 000^2/2$  个处理器来查找 1 000 000 个键的最大值。

本章只对并行算法进行了简单介绍。如果进行详尽介绍，单是这一主题就需要整整一本书。Kumar、Gramma、

Gupta 和 Karypis (1994 年) 的著作就是这样一本书。

## 12.3 习题

### 12.1 节

- 如果假定一个人可以在  $t_a$  时间内完成两个数字的相加，如果将加法运算看作基本运算，那这个人将两个  $n \times n$  矩阵相加，将需要多长时间？说明理由。
- 如果有两个人对数字相加，一个人需要  $t_a$  时间完成两个数字的相加，那么如果将加法运算看作基本运算，这两个人将两个  $n \times n$  矩阵相加，将需要多长时间？说明理由。
- 考虑两个  $n \times n$  矩阵相加的问题。如果一个人完成两个数字相加需要  $t_a$  时间，那需要多少个人才能使获得最终答案的总时间最短？如果假定人手足够，求出答案的最短时间为多少？说明理由。
- 假定一个人可以在  $t_a$  时间内完成两个数字的相加，如果将加法运算看作基本运算，那这个人要将一个列表中的所有  $n$  个数字加起来，共需要多长时间？说明理由。
- 如果两个人对一个列表中的  $n$  个数字相加，一个人完成两个数字的相加需要  $t_a$  时间，如果将加法运算看作基本运算，而且将一个人的加法结果传送给另一个人需要  $t_p$  时间，那这两个人对列表中的  $n$  个数字相加，共需要多长时间？说明理由。
- 考虑对一个列表中的  $n$  个数字进行相加的问题。如果一个人需要  $t_a$  时间完成两个数字的相加，将一个人的加法结果传送给另一个人不需要时间，那需要多少个人才能使获得最终答案的总时间最短？如果假定人手足够，求出答案的最短时间为多少？说明理由。

### 12.2 节

- 编写一个 CREW PRAM 算法，在  $\Theta(\lg n)$  时间内完成对一个列表中所有  $n$  个数字的相加。
- 编写一个 CREW PRAM 算法，使用  $n^2$  个处理器对两个  $n \times n$  矩阵相乘。此算法的性能应当优于标准的  $\Theta(n^3)$  时间串行算法。
- 为快速排序编写一个 PRAM 算法，使用  $n$  个处理器对一个  $n$  元素列表排序。
- 编写一个实现锦标赛方法的串行算法，在一个包含  $n$  个键的数组中找出最大键。证明：这个效率不高于标准串行算法。
- 编写一个 PRAM 算法，使用  $n^3$  个处理器对两个  $n \times n$  矩阵相乘。你的算法应当在  $\Theta(\lg n)$  时间内完成。
- 为 3.2 节的最短路径问题编写一个 PRAM 算法。对比该算法与 Floyd 算法（算法 3.3）的性能。
- 为 3.4 节的链式矩阵乘法问题编写一个 PRAM 算法。对比该算法与最小相乘算法（算法 3.6）的性能。
- 为 3.5 节的最优二叉查找树问题编写一个 PRAM 算法。对比该算法与最优二叉查找树算法（算法 3.9）的性能。

### 补充习题

- 考虑  $n$  个数字的相加问题。如果一个人需要  $t_a(n-1)$  时间完成所有  $n$  个数字的相加，那  $m$  个人是否可能在短于  $[t_a(n-1)]/m$  时间内完成总和的计算？说明理由。
- 为合并排序问题编写一个 PRAM 算法，其运行时间属于  $\Theta((\lg n)^2)$ 。（提示：使用  $n$  个处理器，为每个处理器指定一个键，并利用二叉查找确定该键在最终列表中的位置。）
- 为 3.6 节的旅行推销员问题编写一个 PRAM 算法。对比此算法与旅行推销员算法（算法 3.11）的性能。

# 附录 A

## 必备数学知识回顾

除了标有◆或◆的内容之外，本书并不要求读者具有很强的数学背景知识。具体来说，我们并没有假定读者学习过微积分。但是，为了分析算法，还是需要一定的数学知识的。本附录就回顾了这些必备的数学知识。你应当已经非常熟悉其中大部分甚至全部内容了。

### A.1 符号

有时需要提到大于或等于一个实数  $x$  的最小整数。将这个整数表示为  $\lceil x \rceil$ 。例如，

$$\lceil 3.3 \rceil = 4 \quad \left\lceil \frac{9}{2} \right\rceil = 5 \quad \lceil 6 \rceil = 6 \quad \lceil -3.3 \rceil = -3 \quad \lceil -3.7 \rceil = -3 \quad \lceil -6 \rceil = -6$$

将  $\lceil x \rceil$  称为  $x$  的上取整（ceiling）。对于任意整数  $n$ ,  $\lceil n \rceil = n$ 。有时还会提到小于或等于一个实数  $x$  的最大整数。将这个整数表示为  $\lfloor x \rfloor$ 。例如，

$$\lfloor 3.3 \rfloor = 3 \quad \left\lfloor \frac{9}{2} \right\rfloor = 4 \quad \lfloor 6 \rfloor = 6 \quad \lfloor -3.3 \rfloor = -4 \quad \lfloor -3.7 \rfloor = -4 \quad \lfloor -6 \rfloor = -6$$

将  $\lfloor x \rfloor$  称为  $x$  的下取整（floor）。对于任意整数  $n$ ,  $\lfloor n \rfloor = n$ 。

只能确定所需结果的近似值时，使用符号  $\approx$ ，表示“约等于”。例如，你应当熟悉数字  $\pi$ ，在计算圆的面积和周长时会用到它。 $\pi$  的值不能用有限个十进制数字给出，因为永远都可以再生成更多的数位。（事实上，它甚至不存在像  $\frac{1}{3} = 0.333333\cdots$  这样的形式。）因为  $\pi$  的前六位为 3.14159，所以我们写为：

$$\pi \approx 3.14159$$

我们使用符号  $\neq$  表示“不等于”。例如，如果要表示变量  $x$  和  $y$  的值不相等，则写为：

$$x \neq y$$

我们还经常需要引用一些类似项目之和。如果项目不是太多，那还是很简单的。例如，如果需要引用前七个正整数之和，可以直接写为：

$$1+2+3+4+5+6+7$$

如果需要引用前七个正整数的平方和，只需写为：

$$1^2+2^2+3^2+4^2+5^2+6^2+7^2$$

当项目不多时，这一方法非常有效。但是，如果我们要引用前 100 个正整数之和，那就不能令人满意了。一种做法是写出前几项、一般项和最后一项。也就是，记作：

$$1+2+\cdots+i+\cdots+100$$

如果需要引用前 100 个正整数的平方和，可以写为：

$$1^2+2^2+\cdots+i^2+\cdots+100^2$$

有些时候，如果从前几项可以轻松看出一般项，那就不用再费心写出该项了。例如，对于前 100 个正整数之和，可以直接写为：

$$1+2+\cdots+100$$

如果写出其中一些项目是有意义的，那就写出其中一些。但是，一种更简洁的方法是使用希腊字母  $\Sigma$ ，其

发音为 sigma。例如，利用  $\Sigma$ ，可以将前 100 个正整数之和表示如下：

$$\sum_{i=1}^{100} i$$

这个符号表示对变量  $i$  的值求和，而  $i$  依次取 1 到 100 的值。同理，前 100 个正整数的平方和可以表示为：

$$\sum_{i=1}^{100} i^2$$

我们经常需要表示的一种情景是，和式的最后一个整数为任意整数  $n$ 。利用刚刚介绍的方法，可以将前  $n$  个正整数之和表示为：

$$1+2+\cdots+i+\cdots+n, \text{ 或者 } \sum_{i=1}^n i$$

同理，前  $n$  个正整数的平方和可以表示为：

$$1^2+2^2+\cdots+i^2+\cdots+n^2, \text{ 或者 } \sum_{i=1}^n i^2$$

有时需要对和式求和。例如，

$$\begin{aligned} \sum_{i=1}^4 \sum_{j=1}^i j &= \sum_{j=1}^1 j + \sum_{j=1}^2 j + \sum_{j=1}^3 j + \sum_{j=1}^4 j \\ &= (1) + (1+2) + (1+2+3) + (1+2+3+4) = 20 \end{aligned}$$

同理，可以对和式之和再求和，以此类推。

最后，有时需要引用一个大于任意实数的实体。我们将这个实体称为无穷大，表示为  $\infty$ 。对于任意实数  $x$ ，有

$$x < \infty$$

## A.2 函数

非常简单，一个变量的函数（function）就是一种规则，将变量  $x$  关联到一个独一无二的值  $f(x)$ 。例如，将一个实数的平方与给定实数  $x$  关联在一起的函数  $f$  为：

$$f(x)=x^2$$

一个函数确定一个有序对的集合。例如，函数  $f(x)=x^2$  确定所有有序对  $(x, x^2)$ 。一个函数的曲线（graph）是该函数所确定的所有有序对组成的集合。函数  $f(x)=x^2$  的曲线在图 A-1 中给出。

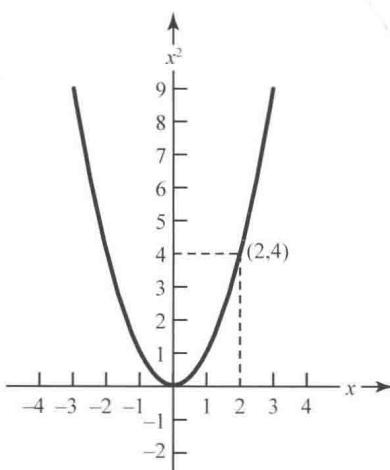


图 A-1 函数  $f(x)=x^2$  的曲线。图中给出了有序对  $(2, 4)$

## 函数

$$f(x) = \frac{1}{x}$$

仅在  $x \neq 0$  时有定义。一个函数的定义域 (domain) 是一个取值集合，这个函数在此集合上有定义。例如， $f(x)=1/x$  的定义域是不等于 0 的所有实数，而  $f(x)=x^2$  的定义域是所有实数。

注意，函数

$$f(x)=x^2$$

只能取非负值。所谓“非负值”是指大于或等于 0 的值，而“正值”表示严格大于 0 的值。一个函数的值域 (range) 是该函数所能取的值集。 $f(x)=x^2$  的值域是非负实数， $f(x)=1/x$  的值域是所有不等于 0 的实数，而  $f(x)=(1/x)^2$  的值域是所有正实数。我们说，一个函数是从其定义域到其值域。例如，函数  $f(x)=x^2$  是从实数到非负实数。

## A.3 数学归纳

有些和式存在闭合形式的表达式。例如，

$$1+2+\dots+n=\frac{n(n+1)}{2}$$

可以通过验证一些  $n$  值来说明这一等式，如下所示：

$$1+2+3+4=10=\frac{4(4+1)}{2}$$

$$1+2+3+4+5=15=\frac{5(5+1)}{2}$$

但是，因为有无穷多个正整数，所以仅仅验证个别值无法确定这个等式对于所有正整数  $n$  都成立。验证个别情况，只会告诉我们这个等式看起来是正确的。要为所有正整数  $n$  获得一个结果，一种强有力的方法是数学归纳法。

数学归纳法 (mathematical induction) 的工作方法与多米诺原理相同。图 A-2 显示，如果两个多米诺骨牌之间的距离总是小于多米诺骨牌的高度，那只需推倒第一个多米诺骨牌就能推倒所有骨牌。之所以能这样做，是因为

(1) 我们推倒了第一张多米诺；

(2) 多米诺骨牌的放置使其中任何两个之间的距离都小于其高度，这样就能保证，如果第  $n$  张骨牌倒下，第  $n+1$  张也会倒下。

如果推倒了第一张骨牌，它会推倒第二张，第二张会推倒第三张，以此类推。理论上，可以有任意多的骨牌，而它们都会倒下。

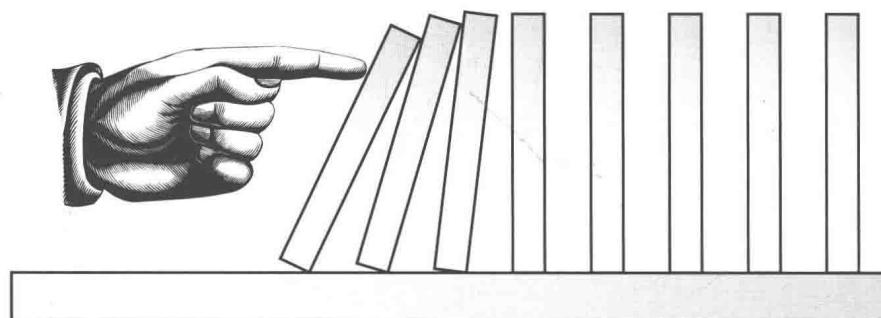


图 A-2 如果推倒了第一个多米诺骨牌，那所有骨牌都会倒下

归纳证明的工作方式与此相同。我们首先证明：对于  $n=1$  来说，要证明的结果是成立的。接下来证明，如果它对于任意正整数  $n$  成立，那对于  $n+1$  也必然成立。一旦证明了这一点，就能知道：因为它对于  $n=1$  成立，所以必然对于  $n=2$  成立；因为它对  $n=2$  成立，那必然对于  $n=3$  成立；以此类推，直到无穷大。从而可以得出结论：它对所有正整数  $n$  都是成立的。在使用归纳法证明某个有关正整数的命题为真时，我们使用以下术语。

**归纳基础**是证明该命题对于  $n=1$ （或其他某个初始值）成立。

**归纳假设**是假定该命题对于任意  $n \geq 1$ （或其他某个初始值）成立。

**归纳步骤**是证明如果该命题对  $n$  成立，则它也必然对  $n+1$  成立。

归纳基础相当于推倒第一张多米诺骨牌，而归纳步骤证明如果第  $n$  张多米诺骨牌倒下，第  $n+1$  张也会倒下。

**例 A.1** 证明，对于所有正整数  $n$ ，有

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

**归纳基础：**对于  $n=1$ ，有

$$1=\frac{1(1+1)}{2}$$

**归纳假设：**假设，对于任意正整数  $n$ ，有

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

**归纳步骤：**需要证明

$$1+2+\cdots+(n+1)=\frac{(n+1)[(n+1)+1]}{2}$$

为此，

$$\begin{aligned} 1+2+\cdots+(n+1) &= 1+2+\cdots+n+n+1 \\ &= \frac{n(n+1)}{2} + n+1 \\ &= \frac{n(n+1) + 2(n+1)}{2} \\ &= \frac{(n+1)(n+2)}{2} \\ &= \frac{(n+1)[(n+1)+1]}{2} \end{aligned}$$

在归纳步骤中，我们突出显示了与归纳假设相等的项目。我们经常这样做，以表明在什么地方应用了归纳假设。注意在归纳步骤所完成的工作。我们做出如下归纳假设：

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

并进行一些代数运算，就可以得出结论：

$$1+2+\cdots+(n+1)=\frac{(n+1)[(n+1)+1]}{2}$$

因此，如果此假设对于  $n$  成立，则它必然对于  $n+1$  也成立。因为在归纳基础中证明了它对于  $n=1$  成立，所以就可以利用多米诺原理得出结论：它对所有正整数  $n$  都是成立的。

你可能感到奇怪，为什么我们最早会认为例 A.1 中的等式可能成立。这是非常重要的一点。我们通常会研究一些情景，做出有依据的猜测，以此推导出一个可能成立的语句。例 A.1 中的等式最初就是这样想出来的，然后利用归纳法来验证这一命题为真。当然，如果不成立，那归纳证明就会失败。从来不可能使用归纳法来推

导一个真命题，能意识到这一点是非常重要的。只有在已经推导出可能为真的命题之后，归纳法才能上场。在 8.5.4 节讨论了建设性归纳，这种方法可以帮助我们发现真命题。

还有一点非常重要，那就是初始值不一定为  $n=1$ 。也就是说，也许只有当  $n \geq 10$  时命题才会成立。在这种情况下，归纳基础就是  $n=10$ 。在某些情况下，归纳基础为  $n=0$ 。这意味着我们是在证明该命题对于所有非负整数成立。

下面再给出一些归纳证明实例。

**例 A.2 证明：**对于所有正整数  $n$ ，有

$$1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

归纳基础：对于  $n=1$ ，有

$$1^2 = 1 = \frac{1(1+1)[(2 \times 1)+1]}{6}$$

归纳假设：假设对于任意正整数  $n$ ，有

$$1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

归纳步骤：我们需要证明

$$1^2 + 2^2 + \cdots + (n+1)^2 = \frac{(n+1)[(n+1)+1][2(n+1)+1]}{6}$$

为此，

$$\begin{aligned} 1^2 + 2^2 + \cdots + (n+1)^2 &= 1^2 + 2^2 + \cdots + n^2 + (n+1)^2 \\ &= \frac{n(n+1)(2n+1)}{6} + (n+1)^2 \\ &= \frac{n(n+1)(2n+1) + 6(n+1)^2}{6} \\ &= \frac{(n+1)(2n^2 + n + 6n + 6)}{6} \\ &= \frac{(n+1)(2n^2 + 7n + 6)}{6} \\ &= \frac{(n+1)(n+2)(2n+3)}{6} \\ &= \frac{(n+1)[(n+1)+1][2(n+1)+1]}{6} \end{aligned}$$

在下一个例子中，归纳基础为  $n=0$ 。

**例 A.3 证明：**对于所有非负整数  $n$ ，有

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$$

采用求和符号，这个等式为：

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

归纳基础：对于  $n=0$ ，有

$$2^0 = 1 = 2^{0+1} - 1$$

归纳假设：假设对于任意一个非负整数  $n$ ，有

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$$

归纳步骤：我们需要证明

$$2^0 + 2^1 + 2^2 + \cdots + 2^{n+1} = 2^{(n+1)+1} - 1$$

为此，

$$\begin{aligned} 2^0 + 2^1 + 2^2 + \cdots + 2^{n+1} &= 2^0 + 2^1 + 2^2 + \cdots + 2^n + 2^{n+1} \\ &= 2^{n+1} - 1 + 2^{n+1} \\ &= 2(2^{n+1}) - 1 \\ &= 2^{(n+1)+1} - 1 \end{aligned}$$

例 A.3 是下例结果的一种特殊情况。

例 A.4 证明对于所有非负整数  $n$  和实数  $r \neq 1$ ，有

$$\sum_{i=0}^n r^i = \frac{r^{n+1} - 1}{r - 1}$$

和式中的这些项目称为等比级数 (geometric progression)。

归纳基础：对于  $n=0$ ，有

$$r^0 = 1 = \frac{r^{0+1} - 1}{r - 1}$$

归纳假设：假定对于一个任意非负整数  $n$ ，有

$$\sum_{i=0}^n r^i = \frac{r^{(n+1)+1} - 1}{r - 1}$$

归纳步骤：我们需要证明

$$\sum_{i=0}^{n+1} r^i = \frac{r^{(n+1)+1} - 1}{r - 1}$$

为此，

$$\begin{aligned} \sum_{i=0}^{n+1} r^i &= r^{n+1} + \sum_{i=0}^n r^i \\ &= r^{n+1} + \frac{r^{n+1} - 1}{r - 1} \\ &= \frac{r^{(n+1)}(r - 1) + r^{n+1} - 1}{r - 1} \\ &= \frac{r^{n+2} - 1}{r - 1} = \frac{r^{(n+1)+1} - 1}{r - 1} \end{aligned}$$

有些时候，通过其他方法可能更容易获得使用归纳法得到的结果。例如，前面的例子证明了

$$\sum_{i=0}^n r^i = \frac{r^{n+1} - 1}{r - 1}$$

也可以不使用归纳法，将上述等式左侧的表达式乘以右侧的分母，并化简后如下：

$$\begin{aligned} (r - 1) \sum_{i=0}^n r^i &= r \sum_{i=0}^n r^i - \sum_{i=0}^n r^i \\ &= (r + r^2 + \cdots + r^{n+1}) - (1 + r + r^2 + \cdots + r^n) \\ &= r^{n+1} - 1 \end{aligned}$$

等式两端除以  $r - 1$ ，即可得到所需结果。

下面再给出一个归纳证明的例子。

**例 A.5** 证明对于所有正整数  $n$ , 有

$$\sum_{i=1}^n i2^i = (n-1)2^{n+1} + 2$$

归纳基础：对于  $n=1$ , 有

$$1 \times 2^1 = 2 = (1-1)2^{1+1} + 2$$

归纳假设：假定对于一个任意正整数  $n$ , 有

$$\sum_{i=1}^n i2^i = (n-1)2^{n+1} + 2$$

归纳步骤：我们需要证明

$$\sum_{i=1}^{n+1} i2^i = [(n+1)-1]2^{(n+1)+1} + 2$$

为此，

$$\begin{aligned} \sum_{i=1}^{n+1} i2^i &= \sum_{i=1}^n i2^i + (n+1)2^{n+1} \\ &= (n-1)2^{n+1} + 2 + (n+1)2^{n+1} \\ &= 2n2^{n+1} + 2 \\ &= [(n+1)-1]2^{(n+1)+1} + 2 \end{aligned}$$

另一种进行归纳假设的方式是假定命题对于所有大于或等于初始值且小于  $n$  的  $k$  值成立, 然后在归纳步骤中, 证明它对于  $n$  也成立。1.2.2 节证明定理 1.1 时就是这样做的。

尽管上面给出的例子都涉及闭合和式的推导, 实际上还有许多其他归纳应用。本书中就遇到了其中一些。

## A.4 定理和引理

词典中将定理定义为一种命题, 提出一些要证明的事情。它在数学中的含义是一样的。上一节的所有例子都可以表述为定理, 而归纳证明就构成了所述定理的证明。例如, 例 A.1 可表述如下。

**定理 A.1** 对于所有整数  $n > 0$ , 都有

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

**证明：**证明过程应当在这里给出。具体到这一定理, 就是例 A.1 中使用的归纳证明。

一般来说, 表述和证明一个定理的目的是为了获得一个可应用于许多具体情景的一般性结果。例如, 对于任意正整数  $n$ , 可以利用定理 A.1 快速计算前  $n$  个整数之和。

有些时候, 学生们可能难以理解定理中“若”(if)陈述和“当且仅当”(if and only if)陈述的区别。下面的两个定理说明了这一区别。

**定理 A.2** 对于任意实数  $x$ , 若  $x > 0$ , 则  $x^2 > 0$ 。

**证明：**两个正整数的乘积为正数, 由这一事实即可证明本定理。

定理 A.2 的逆命题是不成立的。也就是说, “若  $x^2 > 0$ , 则  $x > 0$ ”是不成立的。例如,  $(-3)^2 = 9 > 0$  而  $-3$  不大于 0。事实上, 任意负数的平方都大于 0。因此, 定理 A.2 是“若”陈述的一个例子。当逆命题为真时, 这个定理就是“当且仅当”陈述, 对其有必要证明正逆两个命题。下面的定理就是“当且仅当”陈述的一个例子。

**定理 A.3** 对于任意实数  $x$ , 当且仅当  $1/x > 0$  时,  $x > 0$ 。

**证明：**证明正命题。假定  $x > 0$ , 则

$$\frac{1}{x} > 0$$

因为两个正整数的商大于 0。

证明逆命题。假设  $1/x > 0$ , 则

$$x = \frac{1}{1/x} > 0$$

同样是因为两个正数之商大于 0。

词典中对引理的定义为用于证明另一命题所采用的辅助命题。和定理一样，引理也是一个命题，提出某个要证明的东西。但是，我们通常不关心引理本身的命题，而是当一个定理的证明需要一个或多个辅助命题都正确时，才经常会表述和证明与这些命题有关的引理，然后再用这些引理来证明定理。

## A.5 对数

对数是大多数算法分析中都会用到的数学工具之一。我们简单复习一下它们的性质。

### A.5.1 对数的定义和性质

一个数的常用对数是指为了得到这个数，必须对 10 求多少次幂。如果  $x$  为一个给定数字，则将其常用对数表示为  $\log x$ 。

**例 A.6** 一些常用对数如下：

$$\log 10 = 1, \text{ 因为 } 10^1 = 10$$

$$\log 10000 = 4, \text{ 因为 } 10^4 = 10000$$

$$\log 0.001 = -3, \text{ 因为 } 10^{-3} = \left(\frac{1}{10}\right)^3 = 0.001$$

$$\log 1 = 0, \text{ 因为 } 10^0 = 1$$

回忆一下，任何非零数字的 0 次幂都为 1。

一般情况下，数字  $x$  的对数是指另一个数字  $a$ （称为底数）必须求多少次幂才能得出  $x$ 。数字  $a$  可以是任意大于 1 的正数，而  $x$  必须是正数。也就是说，负数和 0 没有对数这一说。用符号表示，就是  $\log_a x$ 。

**例 A.7**  $\log_a x$  的一些例子如下：

$$\log_2 8 = 3, \text{ 因为 } 2^3 = 8$$

$$\log_3 81 = 4, \text{ 因为 } 3^4 = 81$$

$$\log_2 \frac{1}{16} = -4, \text{ 因为 } 2^{-4} = \left(\frac{1}{2}\right)^4 = \frac{1}{16}$$

$$\log_2 7 \approx 2.807, \text{ 因为 } 2^{2.807} \approx 7$$

注意例 A.7 中的最后一个结果，它是一个数字的对数，而这个数字并不是底数的整数次幂。对数对于所有正数都存在，而不只是对底数的整数次幂存在。当一个数字不是底数的整数次幂时，它的对数是什么含义超出了本附录的范围。我们这里只需注意对数函数是一个增函数。也就是说，

$$\text{若 } x < y, \text{ 则 } \log_a x < \log_a y$$

于是，

$$2 = \log_2 4 < \log_2 7 < \log_2 8 = 3$$

在例 A.7 中看到,  $\log_2 7$  大约是 2.807, 介于 2 与 3 之间。

下面列出一些会在算法分析中用到的重要对数性质。

对数的一些性质 (在所有情况下,  $a > 1$ ,  $b > 1$ ,  $x > 0$ ,  $y > 0$ ) :

$$(1) \log_a 1 = 0$$

$$(2) a^{\log_a x} = x$$

$$(3) \log_a(xy) = \log_a x + \log_a y$$

$$(4) \log_a \frac{x}{y} = \log_a x - \log_a y$$

$$(5) \log_a x^y = y \log_a x$$

$$(6) x^{\log_a y} = y^{\log_a x}$$

$$(7) \log_a x = \frac{\log_b x}{\log_b a}$$

例 A.8 一些应用上述性质的例子如下:

$$2^{\log_2 8} = 8 \quad \{ \text{根据性质(2)} \}$$

$$\log_2(4 \times 8) = \log_2 4 + \log_2 8 = 2 + 3 = 5 \quad \{ \text{根据性质(3)} \}$$

$$\log_3 \frac{27}{9} = \log_3 27 - \log_3 9 = 3 - 2 = 1 \quad \{ \text{根据性质(4)} \}$$

$$\log_2 4^3 = 3 \log_2 4 = 3 \times 2 = 6 \quad \{ \text{根据性质(5)} \}$$

$$8^{\log_2 4} = 4^{\log_2 8} = 4^3 = 64 \quad \{ \text{根据性质(6)} \}$$

$$\log_4 16 = \frac{\log_2 16}{\log_2 4} = \frac{4}{2} = 2 \quad \{ \text{根据性质(7)} \}$$

$$\log_2 128 = \frac{\log 128}{\log 2} \approx \frac{2.10721}{0.30103} = 7 \quad \{ \text{根据性质(7)} \}$$

$$\log_3 67 = \frac{\log 67}{\log 3} \approx \frac{1.82607}{0.47712} \approx 3.82728 \quad \{ \text{根据性质(7)} \}$$

因为许多计算器都有一个  $\log$  函数 (回想一下,  $\log$  的意思是  $\log_{10}$ ), 所以例 A.8 中的最后两个结果说明了可以如何使用计算器针对任意底数计算对数。(我们就是这样计算的。)

在算法分析中经常遇到以 2 为底的对数, 因此我们为它指定了一个简单符号, 就是将  $\log_2 x$  表示为  $\lg x$ 。从现在开始, 我们就使用这一符号。

### A.5.2 自然对数

你可能会想起数字  $e$ , 它的值近似为 2.718281828459。和  $\pi$  一样, 数字  $e$  也不能用有限个十进制数位准确表示, 事实上, 它的十进制展开数中甚至不存在重复模式。我们将  $\log_e x$  表示为  $\ln x$ , 将其称为  $x$  的自然对数(natural logarithm)。例如,

$$\ln 10 \approx 2.3025851$$

你可能想知道, 我们是如何得到这一答案的。其实就是一个带有  $\ln$  函数的计算器。在没有学习微积分的情况下, 是不可能理解如何计算自然对数的, 也不能理解它为什么会被称为“自然”。事实上, 如果只是观察  $e$ , 自然对数会显得最不自然。尽管对微积分的讨论超出了本书的范围(除了标有❷的内容), 但我还是希望能够探讨自然对数的一条性质, 它对于算法分析非常重要。利用微积分, 可以证明  $\ln x$  就是曲线  $1/x$  之下介于 1 到  $x$  之间的面积。图 A-3 的上图中给出了  $x=5$  的情景。在该图的下图中, 说明了如何对每个宽度为单

位 1 的矩形面积求和，以此计算该面积的近似值。这个图形表明， $\ln 5$  的近似值为：

$$(1 \times 1) + \left(1 \times \frac{1}{2}\right) + \left(1 \times \frac{1}{3}\right) + \left(1 \times \frac{1}{4}\right) \approx 2.0833$$

注意，这一面积总是大于真实面积。利用计算器可以算出：

$$\ln 5 \approx 1.60944$$

这些矩形的面积并没有很好地近似表示  $\ln 5$  的值。但是，当我们到达最后一个矩形时（介于  $x$  值 4 和 5 之间的矩形），这一面积与曲线下方  $x=4$  与  $x=5$  之间的面积并没有太大差别。第一个矩形的情况并非如此，而所有后续矩形的近似水平都好于它前面的矩形。因此，当数字不是很小时，这些矩形的面积之和就接近于自然对数值。下面的例子展示了这一结果的用途。

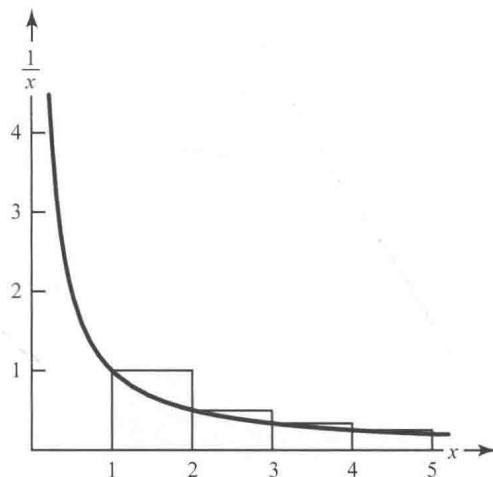
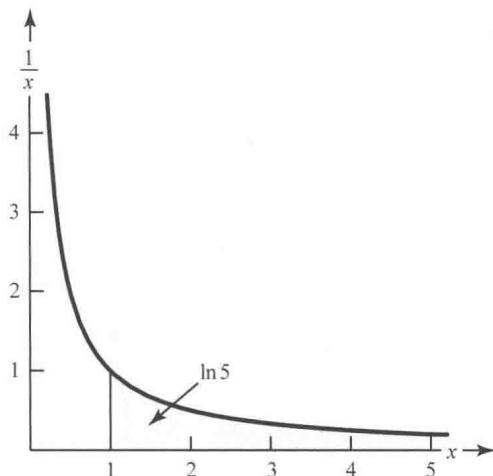


图 A-3 上图中的阴影面积为  $\ln 5$ 。下图中阴影矩形的面积之和是  $\ln 5$  的近似值

**例 A.9** 假定我们希望计算

$$1 + \frac{1}{2} + \cdots + \frac{1}{n}$$

这个和值没有闭合形式的表达式。但是，根据前面的讨论，如果  $n$  不是特别小，则

$$(1 \times 1) + \left(1 \times \frac{1}{2}\right) + \cdots + \left(1 \times \frac{1}{n-1}\right) \approx \ln n$$

当  $n$  不是很小时,  $1/n$  的值与总和相比可以忽略。因此, 可以加上该项, 得到结果

$$1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n$$

在一些分析中会用到这一结果。一般情况下, 可以证明:

$$\frac{1}{2} + \cdots + \frac{1}{n} < \ln n < 1 + \frac{1}{2} + \cdots + \frac{1}{n-1}$$

## A.6 集合

通俗地说, 集合就是一组对象。我们用大写字母(比如  $S$ )来表示集合, 如果枚举一个集合中的所有对象, 就将它们放在大括号中。例如,

$$S = \{1, 2, 3, 4\}$$

是一个包含前四个正整数的集合。对象的列表顺序无关紧要。这就是说,

$$\{1, 2, 3, 4\} \text{ 和 } \{3, 1, 4, 2\}$$

是同一集合, 即前四个正整数的集合。另一个集合的例子是

$$S = \{\text{Wed, Sat, Tues, Sun, Thurs, Mon, Fri}\}$$

这是由一周内每一天的英文名字组成的集合。当一个集合为无限时, 可以通过描述集合中的对象来表示集合。例如, 如果希望表示由 3 的整数倍组成的正整数集合, 可以写为:

$$S = \{\text{对于某一正整数 } i, \text{ 满足 } n=3i \text{ 的 } n\}$$

或者, 可以给出一些项目和一般项, 如下所示:

$$S = \{3, 6, 9, \dots, 3i, \dots\}$$

集合中的对象称为集合的元素(element)或成员(member)。如果  $x$  是集合  $S$  的一个元素, 则记作  $x \in S$ 。若  $x$  不是  $S$  的元素, 则写为  $x \notin S$ 。例如,

$$\text{若 } S = \{1, 2, 3, 4\}, \text{ 则 } 2 \in S, 5 \notin S$$

若集合  $S$  和  $T$  具有相同元素, 即它们是相等的, 则记作  $S=T$ 。如果它们不相等, 则记作  $S \neq T$ 。例如,

$$\text{若 } S = \{1, 2, 3, 4\}, T = \{2, 1, 4, 3\}, \text{ 则 } S = T$$

若  $S$  和  $T$  是两个集合, 且  $S$  中的每个元素也都在  $T$  中, 就说  $S$  是  $T$  的一个子集, 记作  $S \subseteq T$ 。例如,

$$\text{若 } S = \{1, 3, 5\}, \text{ 且 } T = \{1, 2, 3, 5, 6\}, \text{ 则 } S \subseteq T$$

每个集合都是它自己的一个子集。也就是说, 对于任意集合  $S$ , 都有  $S \subseteq S$ 。若  $S$  是  $T$  的一个子集, 且不等于  $T$ , 就说  $S$  是  $T$  的一个真子集, 记作  $S \subset T$ 。例如,

$$\text{若 } S = \{1, 3, 5\}, T = \{1, 2, 3, 5, 6\}, \text{ 则 } S \subset T$$

对于两个集合  $S$  和  $T$ ,  $S$  和  $T$  的交集定义为由同时存在于  $S$  和  $T$  中的所有元素组成的集合, 记作  $S \cap T$ 。例如,

$$\text{若 } S = \{1, 4, 5, 6\}, T = \{1, 3, 5\}, \text{ 则 } S \cap T = \{1, 5\}$$

对于两个集合  $S$  和  $T$ ,  $S$  和  $T$  的并集定义为由或在  $S$  或在  $T$  中的所有元素组成的集合, 记作  $S \cup T$ 。例如,

$$\text{若 } S = \{1, 4, 5, 6\}, T = \{1, 3, 5\}, \text{ 则 } S \cup T = \{1, 3, 4, 5, 6\}$$

对于两个集合  $S$  和  $T$ ,  $S$  和  $T$  的差集定义为由所有在  $S$  中但不在  $T$  中的元素组成的集合, 记作  $S-T$ 。例如,

$$\text{若 } S = \{1, 4, 5, 6\}, T = \{1, 3, 5\}, \text{ 则 } S-T = \{4, 6\}, T-S = \{3\}$$

空集定义为不包含元素的集合。空集表示为  $\emptyset$ 。

若  $S=\{1, 4, 6\}$ ,  $T=\{2, 3, 5\}$ , 则  $S \cap T = \emptyset$

全集  $U$  定义为包含了所关心元素的全部集合。这意味着, 如果  $S$  是我们考虑的任意集合, 则  $S \subseteq U$ 。例如, 如果考虑正整数的集合, 则

$$U=\{1, 2, 3, \dots, i, \dots\}$$

## A.7 排列与组合

假定在一个壶中或其他容器中有四个分别标有 A、B、C、D 的球, 并从壶中拿出两个球。要赢得一次抽奖, 我们必须按照球的绘制顺序取出球来。为了深入理解我们获胜的可能性, 应当计算会有多少种可能结果。可能结果包括:

AB	AC	AD
BA	BC	BD
CA	CB	CD
DA	DB	DC

例如, 结果 AB 与 BA 是不一样的, 因为我们必须按正确顺序取出球。我们已经列出了 12 种不同结果。但是, 能否确定这就是全部结果呢? 注意, 我们将这些结果排列为 4 行 3 列。每一行对应于第一个球的不同选择, 共有四个此种选择。一旦做出选择之后, 同一行中各项目的第二个字符就对应于第二个球的剩余不同选择, 共有三个此种选择。因此, 结果总数为:

$$(4)(3)=12$$

这一结果可以进行推广。例如, 如果有四个球, 并取出三个, 第一个球可以是四个球中的任意一个; 一旦取出了第一个球, 第二个球可以是三个中的任意一个; 一旦拿出了第二个球, 第三个球可以是两个球中的任意一个。于是, 输出结果的数目为:

$$(4)(3)(2)=24$$

一般情况下, 如果有  $n$  个球, 取其中的  $k$  个, 则可能结果的数目为:

$$(n)(n-1)(n-k+1)$$

这叫作一次从  $n$  个对象中取出  $k$  个的排列数。如果  $n=4$ ,  $k=3$ , 则由这一公式得出:

$$(4)(3)\cdots(4-3+1)=(4)(3)(2)=24$$

我们之前已经得到了这一结果。如果  $n=10$ ,  $k=3$ , 则由这一公式得出:

$$(10)(9)\cdots(10-5+1)=(10)(9)(8)(7)(6)=30\,240$$

若  $k=n$ , 就是取出所有球。我们将其称为  $n$  个对象的排列数。前面的公式表明, 这个数目为:

$$n(n-1)(n-n+1)=n!$$

回想一下, 对于一个正整数  $n$ ,  $n!$  定义为该整数乘以小于它的所有正整数,  $0!$  的值定义为 1,  $n!$  关于负整数没有意义。

下面考虑仅仅通过取出正确球就能获胜的抽奖。也就是说, 不需要顺序也正确。我们再次假定有四个分别标有 A、B、C、D 的球, 从中取出两个。这一抽奖中的每个结果对应于前一抽奖中的两个结果。例如, 前一抽奖中的结果 AB 和 BA 在这一抽奖中是相同的。我们将这一结果称为:

A 和 B

因为前一抽奖中的两个结果对应于这一抽奖中的一个结果, 所以将前一抽奖中的结果数除以 2, 就能计算出这一抽奖有多少种结果。这就是说, 这一抽奖中共有

$$\frac{(4)(3)}{2}=6$$

个结果。这六个不同结果为：

A 和 B      A 和 C      A 和 D      B 和 C      B 和 D      C 和 D

现在壶中有标有 A、B、C、D 的四个球，要从中取出三个球，并不要求顺序正确，对于这一抽奖来说，下面的结果都是相同的：

ABC    ACB    BAC    BCA    CAB    CBA

这些输出就是三个对象的排列。回想一下，这些排列的数目为  $3! = 6$ 。为计算不考虑顺序的抽奖中有多少个不同结果，需要将上面彩票的不同结果数除以  $3!$ 。也就是说，这一抽奖中共有

$$\frac{(4)(3)(2)}{3!} = 4$$

种结果。它们是

A 和 B 和 C    A 和 B 和 D    A 和 C 和 D    A 和 C 和 D

一般来说，如果共有  $n$  个球，从中取出  $k$  个球，而且顺序并不重要，则不同结果的数目为：

$$\frac{(n)(n-1)\cdots(n-k+1)}{k!}$$

这是一次从  $n$  个对象中取出  $k$  个的组合数。因为

$$\begin{aligned}(n)(n-1)\cdots(n-k+1) &= (n)(n-1)\cdots(n-k+1) \times \frac{(n-k)!}{(n-k)!} \\ &= \frac{n!}{(n-k)!}\end{aligned}$$

所以，一次从  $n$  个对象取出  $k$  个的组合数公式通常写为：

$$\frac{n!}{k!(n-k)!}$$

利用这一公式，一次从 8 个对象中取出 3 个对象的组合数为：

$$\frac{8!}{3!(8-3)!} = 56$$

在代数教科书中证明的二项式定理表明，对于任意非负整数  $n$  和实数  $a, b$ ，有

$$(a+b)^n = \sum_{k=0}^n \frac{n!}{k!(n-k)!} a^k b^{n-k}$$

因为一次从  $n$  个对象中取  $k$  个对象的组合数就是这个表达式中  $a^k b^{n-k}$  的系数，所以这个数字称为二项式系数。将其表示为  $\binom{n}{k}$ 。

**例 A.10** 我们将证明，对于一个包含  $n$  个项目的集合，其子集数目为  $2^n$ （其中包括空集）。对于  $0 \leq k \leq n$ ，大

小为  $k$  的子集的个数就是一次从  $n$  个对象中取  $k$  个对象的组合数，即  $\binom{n}{k}$ 。这就是说，子集的总数为：

$$\sum_{k=0}^n \binom{n}{k} = \sum_{k=0}^n \binom{n}{k} 1^k 1^{n-k} = (1+1)^n = 2^n$$

倒数第二个等式是根据二项式定理推出的。

## A.8 概率

你可能会回想起在某些情况下对概率论的应用，比如从一个壶中取球时，从一副扑克牌中取最上方的牌时，抛掷一个硬币时。我们将取一个球、取最上方的牌或者抛掷一个硬币的动作称为一次试验。一般情况下，当一个试验拥有可以描述的不同结果集时，可以适用概率论。所有可能结果的集合称为样本空间（sample space）或群体（population）。数学上通常称为“样本空间”，而社会学家通常会说“群体”（因为他们研究的是人）。我们互换使用这些术语。一个样本空间的任意子集称为一个事件（event）。仅包含一个元素的子集称为基本事件（elementary event）。

**例 A.11** 在从一副普通扑克牌中取出最上方扑克的试验中，样本空间包含 52 张不同纸牌。集合

$$S = \{\text{红心 K, 梅花 K, 黑桃 K, 方块 K}\}$$

是一个事件，而集合

$$E = \{\text{红桃 K}\}$$

是一个基本事件。这个样本空间中共有 52 个基本事件。

一件事件（子集）的含义是说，该子集中的元素之一是试验的结果。在例 A.11 中，事件  $S$  的含义是，所取的纸牌是四张 K 中的任意一个，基本事件  $E$  的含义是取出的纸牌是红桃 K。

我们用一个称为事件概率的实数来衡量一个事件包含试验结果的确定性。下面是当样本空间为有限时，概率的一般定义。

**定义** 假定有一个样本空间，其中包含  $n$  个不同结果：

$$\text{样本空间} = \{e_1, e_2, \dots, e_n\}$$

存在一个函数为每个事件  $S$  指定一个实数  $p(S)$ ，如果此函数满足以下条件，则称之为概率函数（probability function）。

$$(1) 0 \leq p(e_i) \leq 1 (1 \leq i \leq n)$$

$$(2) p(e_1) + p(e_2) + \dots + p(e_n) = 1$$

(3) 对于每个不是基本事件的事件  $S$ ， $p(S)$  是一些基本事件的概率之和，这些基本事件的输出都在  $S$  中。例如，如果

$$S = \{e_1, e_2, e_7\}$$

则

$$p(S) = p(e_1) + p(e_2) + p(e_7)$$

样本空间和函数  $p$  一起称为概率空间（probability space）。

因为我们将概率定义为一个集合的函数，所以在提到一个基本元素的概率时，应当写为  $p(\{e_i\})$ ，而不是  $p(e_i)$ 。但是，为避免混乱，我们不会这么做。同样，在引用一个非基本事件的概率时，也不会使用括号。例如，我们将事件  $\{e_1, e_2, e_7\}$  的概率记为  $p(e_1, e_2, e_7)$ 。

我们将一个结果与包含该结果的基本事件关联在一起，然后就可以谈论一个结果的概率了。显然，这是指包含该结果的基本事件的概率。

指定概率的最简单方法是使用中立原则。这一原则是说，如果没有理由偏向某一个结果，就认为这些结果是等概率的。根据这一原则，如果有  $n$  个不同结果，每一个结果的概率是  $1/n$ 。

**例 A.12** 假定壶中有四个分别标有 A、B、C 和 D 的球，而试验是取出一个球。样本空间为 {A, B, C, D}，根据中立原则，

$$p(A) = p(B) = p(C) = p(D) = \frac{1}{4}$$

事件 {A, B} 是指取出球 A 或球 B。它的概率为：

$$p(A, B) = p(A) + p(B) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$$

**例 A.13** 假定一个试验是从一副普通扑克牌中取出最上方的一张纸牌。因为共有 52 张纸牌，所以根据中立原则，每张纸牌的概率是  $\frac{1}{52}$ 。例如，

$$p(\text{红心 K}) = \frac{1}{52}$$

事件

$$S = \{\text{红心 K}, \text{梅花 K}, \text{黑桃 K}, \text{方块 K}\}$$

是指取出的纸牌是一张老 K。它的概率为：

$$\begin{aligned} p(S) &= p(\text{红心 K}) + p(\text{梅花 K}) + p(\text{黑桃 K}) + p(\text{方块 K}) \\ &= \frac{1}{52} + \frac{1}{52} + \frac{1}{52} + \frac{1}{52} = \frac{1}{13} \end{aligned}$$

有时，可以使用上一节给出的排列与组合公式来计算概率。下面的例子说明如何这样做。

**例 A.14** 假定盒中有五个球，标有 A、B、C、D、E，试验是从中取出三个球，顺序不论。我们将计算  $p(A \text{ 和 } B \text{ 和 } C)$ 。回想一下，“A 和 B 和 C”是说输出结果是按任意顺序取出 A、B 和 C。为使用中立原则计算概率，我们需要计算不同结果的个数。也就是说，我们需要计算一次从五个对象中取出三个的组合数。利用上一节的公式可知，该数字为：

$$\frac{5!}{3!(5-3)!} = 10$$

于是，根据中立原则，

$$p(A \text{ 和 } B \text{ 和 } C) = \frac{1}{10}$$

它与其他 9 个结果的概率相同。

学生们经常没有机会深入研究概率论，他们会有一种印象：概率论就是关于比率的。留下这种印象是不公平的，即使是在这种粗略的概述性内容中也是如此。事实上，概率的大多数重要应用都与比率没有任何关系。为说明这一点，我们给出两个简单例子。

教科书上给出的经典概率例子是抛掷硬币。因为硬币是对称的，所以通常使用中立原则来指定概率。于是，我们指定

$$p(\text{正面}) = p(\text{反面}) = \frac{1}{2}$$

另一方面，也可以抛掷一个图钉。和硬币一样，图钉可以以两种方式着地。它可以平端着地，也可以以平端的边缘（及尖头）着地。我们假定它无法仅尖头着地。图 A-4 中展示了这两种着地方式。利用硬币术语，我们可以将平端称为“正面”，把另一结果称为“反面”。因为图钉缺少对称性，所以没有理由使用中立原则，为正面和反面指定相同概率。那应当如何指定概率呢？在硬币的例子中，当我们说  $p(\text{正面}) = \frac{1}{2}$  时，是在隐含地假定，如果将硬币抛掷 1000 次，应当会有大约 500 次正面着地。事实上，如果它正面着地仅有 100 次，就会怀疑它是否重量不匀，概率不是  $\frac{1}{2}$ 。这种重复执行同一试验的想法提供了一种实际计算概率的方法。也就是说，如果将一个试验重复许多次，就可以相当肯定地知道，一个结果的概率大约等于该结果实际出现的次数比例。（一些哲学家实际上将概率定义这一比例在试验次数趋向于无穷时的极限。）例如，一名学生将一个图钉抛掷了 10 000 次，它的平端（正面）着地 3761 次。因此，对于这个大头钉来说，

$$p(\text{正面}) \approx \frac{3761}{10\,000} = 0.3761 \quad p(\text{反面}) \approx \frac{6239}{10\,000} = 0.6239$$

我们看到两个事件的概率不一定相同，但概率总和仍然为 1。这种确定概率的方法称为概率的相对频率方法。在由相对频率计算概率时，我们使用  $\approx$  符号，因为无论执行多少次试验，都不能确定相对频率就准确等于概率。例如，假定壶中有两个标有 A 和 B 的球，将取球试验重复 10 000 次。我们不能确定标有 A 的球将被恰好取出 5000 次，它可能仅被取出 4967 次。根据中立原则可得：

$$p(A)=0.5$$

而使用相对频率方法会得到：

$$p(A) \approx 0.4967$$

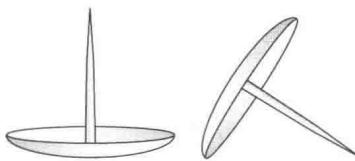


图 A-4 图钉的两种着地方式。因为图钉缺少对称性，所以这两种方式的概率不一定相同

相对频率方法并不仅限于只有两种可能结果的试验。例如，如果有一个六面骰子，它不是一个完美的立方体，六个基本事件的概率可能是不同的。但是，它们的总和仍然是 1。下面的例子说明了这一情景。

**例 A.15** 假定有一个非对称的六面骰子，在 1000 次抛掷中，确定六个面的出现次数如下：

面	次数
1	200
2	150
3	100
4	250
5	120
6	180

则

$$\begin{aligned} p(1) &\approx 0.2 \\ p(2) &\approx 0.15 \\ p(3) &\approx 0.1 \\ p(4) &\approx 0.25 \\ p(5) &\approx 0.12 \\ p(6) &\approx 0.18 \end{aligned}$$

根据概率空间定义中的条件 3，

$$p(2, 3) = p(2) + p(3) \approx 0.15 + 0.1 = 0.25$$

这是抛掷一次骰子时出现 2 或 3 的概率。

还有其他一些计算概率的方法，将概率理解为一次结果中的置信度，就是其中一种方法，而且决不是应用最少的一种。例如，假定芝加哥熊队要与达拉斯牛仔队举行一场足球比赛。在编写本书时，我没有什么理由相信熊队会战胜，因此，我不会为每个队指定相同的获胜概率。因为这个比赛不可能重复许多次，所以不能使用相对频率方法获得概率。但是，如果要对这场比赛打赌，那我希望得到熊队获胜的概率。可以用概率的主观方法来获取，方法如下：如果买熊队获胜的彩票需要 1 美元，投注人就要判断如果熊队真的胜了，他是否感到这场彩票是值得的。我感到它必须得值 5 美元。也就是说，只有当这张彩票在熊队获胜时至少值 5 美元时，我才

会为这张彩票支付 1 美元。对我来说，熊队获胜的概率为：

$$p(\text{熊队获胜}) = \frac{1\text{美元}}{5\text{美元}} = 0.2$$

也就是说，这个概率的计算依据是这个投注人认定的公平赌局。这种方法称为是“主观的”，因为别人可能会说这张彩票只值 4 美元。对那个人来说， $p(\text{熊队获胜})=0.25$ 。我们两个人在逻辑上可能都没错。当一个概率就是表示某个个体的信心时，就那不存在一个独一无二的正确概率。概率是个体信心的函数，这意味着它是主观的。如果有人认为获胜量应当等于投注量（也就是说，这张彩票应当值 2 美元），那对于这个人来说，

$$p(\text{熊队获胜}) = \frac{1\text{美元}}{2\text{美元}} = 0.5$$

我们看到，概率的内涵远不止比率那么简单。关于概率含义及其哲学的详尽介绍，可阅读 Fine (1973 年) 的文献。概率的相对频率方法在 Neapolitan (1992 年) 的文献中讨论。“中立原则”这一表达方式最早出现在 Keynes (1948 年) 的文献中（最初于 1921 年发表）。Neapolitan (1990 年) 讨论了因为使用中立原则而导致的悖论。

### A.8.1 随机性

尽管人们在交流中经常使用“随机”一词，但要严格定义它却是非常困难的。随机性涉及一个过程。从直观上来说，随机过程 (random process) 是指以下含义。第一，该过程必须能够生成一个任意长的结果序列。例如，重复抛掷同一个硬币的过程可以生成一个或为正面或为反面的任意长结果序列。第二，结果必须是不可预测的。但什么是“不可预测的”，却多少有些含糊。我们似乎又回到了起点；我们只是用“不可预测”代替了“随机”。

在 20 世纪早期，Richard von Mises 使随机性的概率变得更为具体。他说，一个“不可预测的”过程应当不允许存在一个成功的赌博策略。也就是说，如果选择对这样一个过程的结果打赌，那对其中一个结果子序列投注的获胜机会并不会高于对每个结果投注的机会。例如，假定我们决定对重复抛掷硬币中出现正面投注。我们大多数人都认为，对相隔一次的抛掷结果进行投注，其获胜机会并不会高于对每次抛掷结果进行投注。另外，我们大多数人都感觉没有其他“特殊”子序列可以提高获胜机会。如果真的无法通过对某一子序列投注来提高获胜机会，那重复抛掷硬币的过程就是一个随机过程。再举一个例子，假定在一个群体中包含有癌症和无癌症的个体，我们重复从这一群体中进行个体采样，而且在进行下一次采样之前，会将所采样的个体放回群体中。（这种方式称为有放回的采样。）假定我们选择对癌症投注。如果采样时从来不会偏向任意特定个体，我们大多数人都会认为，仅对某个子序列投注时的获胜机会并不会高于每次都进行投注的机会。如果对某一子序列进行投注并不会真的提高获胜机会，那这一采样过程就是随机的。但是，如果我们有时偏向于选择吸烟者，每四次中有一次仅抽取吸烟者，其他各次都是从整个群体中采样，那这一过程就不再是随机的，因为用每四次投注一次代替每次投注可以提高获胜机会。

从直观上来说，当我们说“采样时从来不会偏向任意特定个体”时，是指我们完成采样的方式中不存在模式。例如，如果有放回地对一个壶中的球进行采样，并且在每次采样之前大力摇晃这个壶，使里面的球充分混合，那就不会有任何偏向。你可能已经注意到，在抽取国家彩票时，那些球混合得是多么充分。在对人类群体进行采样时，要保证没有任何偏向性是很不容易的。采样方法的讨论超出了本附录的范围。

Von Mises 关于“不允许存在成功赌博策略”的要求，让我们更好地理解了有机性的含义。一个可预测的或者说非随机过程是允许存在某种成功的赌博策略的。非随机过程的一个例子就是上面提到的——每四次抽样中有一次只抽取吸烟者。一个不太明显的例子是作者的锻炼模式。我倾向于在周二、周四和周日在健身俱乐部锻炼，但如果错过了某一天，会在其他各天中补回一天。如果要对我是否会在某一具体日子锻炼而打赌，那针对每个周二、周四和周日投注的机会，要远高于对每一天投注的机会。这个过程不是随机的。

尽管 Von Mises 能让我们更好地理解随机性，但他没能给出一个严格的数学定义。Andrei Kolmogorov 最终用可压缩序列的概念给出了这一定义。简单来说，对于一个有限序列，如果能够采用某种编码方式，使所需

比特数少于对序列中每一项进行编码时的比特数，那这一序列就是可压缩的。例如，序列

就是将“10”重复16次，可以将其表示为

1610

因为对这种表示方式进行编码的比特数要少于对序列中每个项目进行编码所需的比特数，所以这个序列是可压缩的。一个不可压缩的序列称为随机序列（random sequence）。例如，序列

1001101000101101

是随机的，因为它没有更为高效的表示方式。从直观上来说，随机序列就是没有规则或模式的序列。

根据 Kolmogorov 理论，随机过程是指这样一种进程，它在持续足够长的时间后，会生成一个随机序列。例如，假定重复抛掷一个硬币，并将 1 与正面相关联，将 0 与负面相关联。在六次抛掷之后，可以看到序列

101010

但根据 Kolmogorov 理论，整个序列最终将不会有这种规律性。将随机过程定义为一定会生成随机序列的过程，存在某种哲学难题。许多概率学家认为，这只是说明该序列为随机的可能性比较高，但该序列不随机的可能性还是存在的。例如，在重复抛掷一个硬币时，他们相信，尽管可能性很少，但这个硬币的确可能永远正面向上。前面已经说过，随机性是一个很难的概念，即使在今天，关于它的性质也存在一些争议。

我们现在来讨论随机性与概率的关系。一个随机过程确定了一个概率空间（见本节开头的定义），每次执行该空间的试验时，这个过程就会生成一个结果。下面的例子对此进行了说明。

**例 A.16** 假定有一个壶，其中一个黑球和一个白球，我们重复抽取一个球，并放回它。这一随机过程确定了一个概率空间，其中

$$p(\text{黑球})=p(\text{白球})=0.5$$

在每次执行此空间的试验时，都会抽取一个球。

**例 A.17** 例 A.15 中重复抛掷一个非对称的六面骰子，就是一个随机过程，它确定了一个概率空间，其概率如下：

$$\begin{aligned} p(1) &\approx 0.2 \\ p(2) &\approx 0.15 \\ p(3) &\approx 0.1 \\ p(4) &\approx 0.25 \\ p(5) &\approx 0.12 \\ p(6) &\approx 0.18 \end{aligned}$$

每次执行这个空间的试验时，都抛掷该骰子。

**例 A.18** 假定一个群体中包括  $n$  个人，其中一些患有癌症，采用有放回的方式对人们进行采样，采样时从来不会偏向任意特定个体。这一随机过程确定了一个概率空间，其中群体是一个样本空间（回想一下，“样本空间”和“群体”是可以互换使用的），每个人被抽中（基本事件）的概率为  $1/n$ 。一个患有癌症的人被抽中的概率为：

$$\frac{\text{患有癌症的人数}}{n}$$

每次执行该试验时，就说从群体中随机抽取了一个人。重复执行该试验得到的全部结果称为对该群体的一个随机样本 (random sample)。利用统计方法可以证明，如果一个随机样本很大，那这个样本可以用来代替该群体的可能性就很大。例如，如果随机样本很大，在所采样的人中有 $\frac{1}{3}$ 患有癌症，

那群体中患有癌症的比例就非常可能接近于 $\frac{1}{3}$ 。

**例 A.19** 假定有一副普通扑克牌，依次翻转纸牌。这一过程不是随机的，纸牌的选择也不是随机的。在每次生成一个结果时，这个非随机过程会确定一个不同的概率空间。在第一次试验中，每张纸牌的概率为  $\frac{1}{52}$ 。在第二次试验中，在第一次试验中已经被翻过来的纸牌拥有概率 0，其他每张纸牌的概率为  $\frac{1}{51}$ ，以此类推。

假定重复抽取最上方的纸牌，放回它，再次洗牌。这是不是一个随机过程呢？纸牌的选取是否随机呢？答案是否定的。魔术师和统计学家 Persi Diaconis 已经证明，必须将纸牌洗七次，充分混合它们，这一过程才会是随机的（见 Aldous 和 Diaconis, 1986 年）。

尽管在今天看来，von Mises 的随机性概念在直观上很有吸引力，但在他提出自己理论的时代（20 世纪早期），他的观点并没有被普遍接受。他最强大的对手是哲学家 K. Marbe。Marbe 认为大自然是有记忆的。根据他的理论，如果在重复抛掷一个均匀硬币时（也就是说，对于这个硬币，正面的相对频率是 0.5），反面连续出现 15 次，那在下一次抛掷中出现正面的概率将增加，因为大自然会为之前的所有反面做出补偿。如果这一理论正确，那在出现很长的反面序列之后对正面进行投注，就会提高获胜机会。Iverson 等人（1974 年）执行了一些试验，来验证 von Mises 和 kolmogorov 的观点。具体来说，他们的试验表明，硬币抛掷和骰子的抛掷的确生成随机序列。今天很少有科学家赞同 Marbe 的理论了，但相当多的赌徒还是赞同的。

Von Mises 的最初理论出现在 von Mises (1919 年) 的文献中，在 von Mises (1957 年) 的文献中进行了更容易理解的讨论。关于可压缩序列和随机序列的详尽讨论，可以在 Van Lambalgen (1987 年) 的文献中找到。前面曾经说过，将随机过程定义为一定能生成随机序列的过程时，存在一些困难，Neapolitan (1992 年) 和 Van Lambalgen (1987 年) 都解决了这些难题。

## A.8.2 期望值

我们用一个例子来介绍期望值（均值）。

**例 A.20** 假定有四位学生，身高分别为 68、72、67 和 74 英寸。他们的平均身高为：

$$\text{平均身高} = \frac{68 + 72 + 67 + 74}{4} \text{ 英寸} = 70.25 \text{ 英寸}$$

现在假定有 1000 位学生，他们的身高分布符合以下百分比：

学生百分比	身高（英寸）
20	66
25	68
30	71
10	72
15	74

为计算平均身高，我们首先确定每位学生的身高，然后像前面一样进行计算。但是，如果直接像下面这样来计算均值，效率要高得多：

$$\text{平均身高} = 66(0.2) + 68(0.25) + 71(0.3) + 72(0.1) + 74(0.15) \text{ 英寸} = 69.8 \text{ 英寸}$$

注意，这个例子中的百分比就是使用中立原则获得的概率。也就是说，20% 的学生身高为 66 英寸，其意思是说如果随机选择 1000 位学生，有 200 位学生的身高为 66 英寸，则

$$p(\text{高 66 英寸}) = \frac{200}{1000} = 0.2$$

一般地，期望值定义如下。

**定义** 假定有一个概率空间，其样本空间为：

$$\{e_1, e_2, \dots, e_n\}$$

每个结果  $e_i$  有一个实数  $f(e_i)$  与其相关联，则  $f(e_i)$  称为样本空间上的随机变量（random variable）。 $f(e_i)$  的期望值（expected value），或说均值给出如下：

$$f(e_1)p(e_1)+f(e_2)p(e_2)+\dots+f(e_n)p(e_n)$$

随机变量被称为“随机”，是因为随机过程可以确定随机变量的值。有时也会使用“chance variable”和“stochastic variable”这样的表达方式。

**例 A.21** 假定有例 A.15 中的非对称六面骰子。也就是说，

$$\begin{aligned} p(1) &\approx 0.2 & p(4) &\approx 0.25 \\ p(2) &\approx 0.15 & p(5) &\approx 0.12 \\ p(3) &\approx 0.1 & p(6) &\approx 0.18 \end{aligned}$$

样本空间由可能出现的六个不同面组成，这个样本空间上的随机变量就是写在面上的数字，这一随机变量的期望值为：

$$\begin{aligned} &1p(1)+2p(2)+3p(3)+4p(4)+5p(5)+6p(6) \\ &\approx 1(0.2)+2(0.15)+3(0.1)+4(0.25)+5(0.12)+6(0.18)=3.48 \end{aligned}$$

如果将骰子抛掷许多次，就可以预期，所出现数字的均值约等于 3.48。

一个样本空间上并不一定只定义一个独有的随机变量。这一样本空间上的另一个随机变量可能是一个函数，出现奇数时的取值为 0，出现偶数时的取值为 1，则这个随机变量的期望值为：

$$\begin{aligned} &0p(1)+1p(2)+0p(3)+1p(4)+0p(5)+1p(6) \\ &\approx 0(0.2)+1(0.15)+0(0.1)+1(0.25)+0(0.12)+1(0.18)=0.58 \end{aligned}$$

**例 A.22** 假定例 A.20 中的 1000 位学生是我们的样本空间。例 A.20 中计算的身高是这个样本空间上的一个随机变量。另一个随机变量为体重，如果体重的分布符合以下百分比：

学生百分比	体重（磅）
15	130
35	145
30	160
10	170
10	185

则这个随机变量的期望值为：

$$130(0.15)+145(0.35)+160(0.30)+170(0.10)+185(0.10)=153.75 \text{ 磅}$$

## A.9 习题

### A.1 节

1. 计算以下各值。

- (a)  $\lfloor 2.8 \rfloor$       (b)  $\lfloor -10.42 \rfloor$
- (c)  $\lceil 4.2 \rceil$       (d)  $\lceil -34.92 \rceil$
- (e)  $\lfloor 5.2-4.7 \rfloor$       (f)  $\lceil 2\pi \rceil$

2. 证明： $\lceil n \rceil = -\lfloor -n \rfloor$ 。

3. 证明：对于任意实数  $x$ ,

$$\lfloor 2x \rfloor = \lfloor x \rfloor + \left\lfloor x + \frac{1}{2} \right\rfloor$$

4. 证明：对于任意整数  $a>0$ ,  $b>0$  及  $n$ ,

$$(a) \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2} \right\rfloor = n$$

$$(b) \left\lfloor \frac{\lfloor n/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{n}{ab} \right\rfloor$$

5. 用求和符号 (sigma) 写出以下各式。

$$(a) 2+4+6+\cdots+2(99)+2(100)$$

$$(b) 2+4+6+\cdots+2(n-1)+2n$$

$$(c) 3+12+27+\cdots+1200$$

6. 计算以下各个和式。

$$(a) \sum_{i=1}^5 (2i + 4)$$

$$(b) \sum_{i=1}^{10} (i^2 - 4i)$$

$$(c) \sum_{i=1}^{200} \left( \frac{i}{i+1} - \frac{i-1}{i} \right)$$

(提示：如果不加化简地写出前面两项或三项，应当可以看出其中的模式。)

$$(d) \sum_{i=0}^5 (2^i n^{5-i}) \quad (\text{当 } n=4 \text{ 时})$$

$$(e) \sum_{i=1}^4 \sum_{j=1}^i (j+5)$$

## A.2 节

7. 给出函数  $f(x)=\sqrt{x-4}$  的曲线。这个函数的定义值和值域是什么？

8. 给出函数  $f(x)=(x-2)/(x+5)$  的曲线。这个函数的定义值和值域是什么？

9. 给出函数  $f(x)=\lfloor x \rfloor$  的曲线。这个函数的定义值和值域是什么？

10. 给出函数  $f(x)=\lceil x \rceil$  的曲线。这个函数的定义值和值域是什么？

## A.3 节

11. 利用数学归纳法证明，对于所有整数  $n>0$ ，有

$$\sum_{k=1}^n k(k!) = (n+1)! - 1$$

12. 利用数学归纳法证明，对于每个可能的整数  $n$ ,  $n^2-n$  为偶数。

13. 使用数学归纳法证明，对于所有整数  $n>4$ ，有

$$2^n > n^2$$

14. 利用数学归纳法证明，对于所有整数  $n>0$ ，有

$$\left( \sum_{i=1}^n i \right)^2 = \sum_{i=1}^n i^3$$

## A.4 节

15. 证明：若  $a$  和  $b$  都是奇整数，则  $a+b$  是一个偶整数。其逆命题是否成立？

16. 证明：当且仅当  $a$  和  $b$  奇偶性不同时， $a+b$  是一个奇整数。

**A.5 节**

17. 计算以下各值。

- |                        |                           |                   |
|------------------------|---------------------------|-------------------|
| (a) $\log 1000$        | (b) $\log 100\,000$       | (c) $\log_4 64$   |
| (d) $\lg \frac{1}{16}$ | (e) $\log_5 125$          | (f) $\log 23$     |
| (g) $\lg(16 \times 8)$ | (h) $\log(1000/100\,000)$ | (i) $2^{\lg 125}$ |

18. 在同一坐标系中绘出  $f(x)=2^x$  和  $g(x)=\lg x$  的曲线。

19. 给出使  $x^2+6x+12>8x+20$  成立的  $x$  值。

20. 给出使  $x>500\lg x$  成立的  $x$  值。

21. 证明:  $f(x)=2^{3\lg x}$  不是一个指数函数。

22. 证明: 对于任意正整数  $n$ , 有

$$\lfloor \lg n \rfloor + 1 = \lceil \lg(n+1) \rceil$$

23. 对于大的  $n$  值, 利用  $n!$  的 Stirling 近似式

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

找出  $\lg(n!)$  的公式。

**A.6 节**

24. 设  $U=\{2, 4, 5, 6, 8, 10, 12\}$ ,  $S=\{2, 4, 5, 10\}$ ,  $T=\{2, 6, 8, 10\}$ 。( $U$  是全集。)计算以下各式。

- (a)  $S \cup T$
- (b)  $S \cap T$
- (c)  $S - T$
- (d)  $T - S$
- (e)  $((S \cap T) \cup S)$
- (f)  $U - S$  (称为  $S$  的补集)

25. 设集合  $S$  中包含  $n$  个元素, 证明  $S$  有  $2^n$  个子集。

26. 设  $|S|$  表示  $S$  中的元素个数。证明下式的有效性。

$$|S \cup T| = |S| + |T| - |S \cap T|$$

27. 证明以下各式是等价的。

- (a)  $S \subset T$
- (b)  $S \cap T = S$
- (c)  $S \cup T = T$

**A.7 节**

28. 计算一次从 10 个对象中取出 6 个对象时的排列数。

29. 计算一次从 10 个对象中取出 6 个对象时的组合数。也就是说, 计算

$$\binom{10}{6}$$

30. 假定有一个抽奖, 是要从一个包含 10 个球的壶中取出 4 个球。一个获胜的彩票必须按照它们的取出顺序给出所取的球。存在多少种可以区分的彩票?

31. 假定有一个抽奖, 是要从一个包含 10 个球的壶中取出 4 个球。一个获胜的彩票只需各球正确, 无须考虑取出顺序。存在多少种可以区分的彩票?

32. 利用数学归纳法证明 A.7 节给出的二项式定理。

33. 证明下面恒等式的有效性。

$$\binom{2n}{2} = 2 \binom{n}{2} + n^2$$

34. 假定第一种对象有  $k_1$  个，第二种对象有  $k_2$  个……第  $m$  种对象有  $k_m$  个，其中  $k_1+k_2+\cdots+k_m=n$ 。证明：这  $n$  个对象的不同排列数等于

$$\frac{n!}{(k_1!)(k_2!)\cdots(k_m!)}$$

35. 设  $f(n, m)$  是将  $n$  个相同对象分布到  $m$  个集合中，这些集合的顺序是有关系的。例如，如果  $n=4, m=2$ ，对象集合为  $\{A, A, A, A\}$ ，可能分布如下：

- (1)  $\{A, A, A, A\}, \emptyset$
- (2)  $\{A, A, A\}, \{A\}$
- (3)  $\{A, A\}, \{A, A\}$
- (4)  $\{A\}, \{A, A, A\}$
- (5)  $\emptyset, \{A, A, A, A\}$

我们看到  $f(4, 2)=5$ 。证明：一般情况下，

$$f(n, m) = \binom{n+m-1}{m-1}$$

提示：不是所有这些分布的集合包含了将  $n$  个  $A$  放在第一个位置的所有分布、将  $n-1$  个  $A$  放在第一个位置的所有分布……将 0 个  $A$  放到第一个位置的所有分布。对  $m$  使用归纳法。

36. 证明以下恒等式的有效性。

$$\binom{n}{k+1} = \frac{n-k}{k+1} \binom{n}{k}$$

## A.8 节

37. 假定有第 30 题中的彩票。假定所有可能彩票都被打印了出来，而且所有彩票都是互不相同的。

- (a) 计算购买一张彩票获胜的概率。
- (b) 计算购买七张彩票获胜的概率。

38. 假定从一副普通扑克（52 张牌）中取出一手牌（5 张）。

- (a) 计算这一手牌中包含 4 张 A 的概率。
- (b) 计算这一手牌中包含一色 4 张牌的概率。

39. 假定要滚动一个均匀的六面骰子（也就是说，每一面向上的概率为  $\frac{1}{6}$ ）。玩家获得的美元数与向上的点数一样，但 5 点和 6 点向上时例外，在这些情况下，玩家将分别损失 5 美元或 6 美元。

- (a) 计算玩家将获取或输掉的钱数的期望值。
- (b) 如果这个游戏重复 100 次，计算玩家最多输掉的钱数，最多赢得的钱数，预期赢得或输掉的钱数。

40. 假定我们正在一个包含  $n$  个不同元素的列表中查找一个元素。在使用顺序查找算法（线性查找）时，需要的平均（期望）比较次数为多少？

41. 在对一个  $n$  元素数组执行一次删除操作时，元素移动的期望数为多少？

# 附录 B

## 求解递归方程：在递归算法分析中的应用

递归算法的分析不像迭代算法那样简单。但一般情况下，用递归方程来表示一个递归算法的时间复杂度也并不是特别困难。要计算其时间复杂度，必须求解递归方程。本附录讨论求解此类方程的技巧，并在递归算法分析中应用求解结果。

### B.1 用归纳法求解递归方程

附录 A 中复习了数学归纳法，下面说明如何用它来分析一些递归算法。首先考虑一个计算  $n!$  的递归算法。

#### 算法 B.1 阶乘

问题：计算  $n!=n(n-1)(n-2)\cdots(3)(2)(1)$  ( $n \geq 1$ )

$$0! = 1.$$

输入：非负整数  $n$ 。

输出： $n!$ 。

```
int fact (int n)
{
    if (n==0)
        return 1;
    else
        return n*fact(n-1);
}
```

为了解这一算法的效率，我们计算一下这个函数为每个  $n$  值执行多少次乘法。对于一个给定的  $n$  值，所完成的乘法次数就是在执行  $\text{fact}(n-1)$  时完成的次数再加上将  $n$  乘以  $\text{fact}(n-1)$  时的一次乘法。如果用  $t_n$  表示对于一个给定  $n$  值完成的乘法次数，则可以推导得出：

$$t_n = \underbrace{t_{n-1}}_{\text{递归调用中的乘法}} + 1 \quad \underbrace{\quad}_{\text{顶级调用中的乘法}}$$

这样一个方程称为递归方程 (recurrence equation)，因为  $n$  的函数值是由一个较小  $n$  值处的函数值给出的。递归本身并没有给出一个独一无二的函数。必须还要有一个起点，称为初始条件。在这个算法中，当  $n=0$  时不执行乘法。因此，初始条件为：

$$t_0=0$$

对于更大的  $n$  值，可以计算  $t_n$  如下：

$$\begin{aligned} t_1 &= t_{1-1}+1=t_0+1=0+1=1 \\ t_2 &= t_{2-1}+1=t_1+1=1+1=2 \\ t_3 &= t_{3-1}+1=t_2+1=2+1=3 \end{aligned}$$

这样继续下去，可以给出越来越多的  $t_n$  值，但如果不是从 0 开始，就不能计算出任意  $n$  值的  $t_n$ 。我们需要有  $t_n$  的显式表达式。这种表达式称为递归方程的一个解 (solution)。回想一下，使用归纳法是不可能找到解的。归纳法只是验证一个候选解是正确的。(8.5.2 节讨论的建设性归纳可以帮助找出一个解。) 检查前面少数几个值，就可以找到这个递归方程的一个候选解。查看刚刚计算得出的几个解，发现：

$$t_n=n$$

是一个解。既然有了候选解，就可以用归纳法尝试证明它是正确的。

**归纳基础：**对于  $n=0$ ，有

$$t_0 = 0$$

**归纳假设：**假设对于任意正整数  $n$ ，有

$$t_n = n$$

**归纳步骤：**我们需要证明

$$t_{n+1} = n + 1$$

如果在递推式中插入  $n+1$ ，则得到：

$$t_{n+1} = t_{(n+1)-1} + 1 = t_n + 1 = n + 1$$

这就完成了我们的归纳证明：候选解  $t_n$  是正确的。注意，式中突出显示了根据归纳假设为相等的各项。我们在归纳证明中经常这样做，以表明归纳假设的应用位置。

在递归算法分析中有两个步骤。第一个步骤是确定递归方程，第二步是解决它。现在的目的是说明如何求解该递归方程。为本书中递归算法确定递归方程的工作，我们已经在讨论算法时完成了。因此，本附录的剩余部分不再讨论算法，只是直接将已给出的递归方程拿来用。下面给出更多使用归纳法求解递归方程的例子。

**例 B.1** 考虑以下递归方程：

$t_n = t_{n/2} + 1 \quad (n > 1, n \text{ 是 } 2 \text{ 的幂})$
$t_1 = 1$

前几个值为：

$$\begin{aligned} t_2 &= t_{2/2} + 1 = t_1 + 1 = 1 + 1 = 2 \\ t_4 &= t_{4/2} + 1 = t_2 + 1 = 2 + 1 = 3 \\ t_8 &= t_{8/2} + 1 = t_4 + 1 = 3 + 1 = 4 \\ t_{16} &= t_{16/2} + 1 = t_8 + 1 = 4 + 1 = 5 \end{aligned}$$

似乎有

$$t_n = \lg n + 1$$

我们使用归纳法来证明它是正确的。

**归纳基础：**对于  $n=1$ ，有

$$t_1 = 1 = \lg 1 + 1$$

**归纳假设：**假定对于任意  $n > 0$ ，且  $n$  为 2 的幂，有

$$t_n = \lg n + 1$$

**归纳步骤：**因为递归方程仅对 2 的幂成立，则在  $n$  之后要考虑的下一个值是  $2n$ 。因此，需要证明

$$t_{2n} = \lg(2n) + 1$$

如果在递归方程中插入  $2n$ ，则得到：

$$\begin{aligned} t_{2n} &= t_{(2n)/2} + 1 = t_n + 1 = \lg n + 1 + 1 \\ &= \lg n + \lg 2 + 1 \\ &= \lg(2n) + 1 \end{aligned}$$

**例 B.2** 考虑以下递归方程

$t_n = 7t_{n/2} \quad (n > 1, n \text{ 为 } 2 \text{ 的幂})$
$t_1 = 1$

前几个值为

$$t_2 = 7t_{2/2} = 7t_1 = 7$$

$$\begin{aligned}t_4 &= 7t_{4/2} = 7t_2 = 7^2 \\t_8 &= 7t_{8/2} = 7t_4 = 7^3 \\t_{16} &= 7t_{16/2} = 7t_8 = 7^4\end{aligned}$$

似乎有

$$t_n = 7^{\lg n}$$

我们使用归纳法来证明它是正确的。

**归纳基础：**对于  $n=1$ ，有

$$t_1 = 1 = 7^0 = 7^{\lg 1}$$

**归纳假设：**假定，对于任意  $n > 0$ ，且  $n$  为 2 的幂，有

$$t_n = 7^{\lg n}$$

**归纳步骤：**我们需要证明

$$t_{2n} = 7^{\lg(2n)}$$

如果在递归方程中插入  $2n$ ，则得到：

$$t_{2n} = 7t_{(2n)/2} = 7t_n = 7 \times 7^{\lg n} = 7^{1+\lg n} = 7^{\lg 2 + \lg n} = 7^{\lg(2n)}$$

这就完成了归纳证明。最后，因为

$$7^{\lg n} = n^{\lg 7}$$

所以，这个递归方程的解通常给出如下：

$$t_n = n^{\lg 7} \approx n^{2.81}$$

**例 B.3** 考虑以下递归方程

$$\begin{aligned}t_n &= 2t_{n/2} + n - 1 \quad (n > 1, n \text{ 为 } 2 \text{ 的幂}) \\t_1 &= 0\end{aligned}$$

前几个值为：

$$\begin{aligned}t_2 &= 2t_{2/2} + 2 - 1 = 2t_1 + 1 = 1 \\t_4 &= 2t_{4/2} + 4 - 1 = 2t_2 + 3 = 5 \\t_8 &= 2t_{8/2} + 8 - 1 = 2t_4 + 7 = 17 \\t_{16} &= 2t_{16/2} + 16 - 1 = 2t_8 + 15 = 49\end{aligned}$$

由这些值看出不什么明显的候选解。前面曾经提到，归纳只能证明一个解是正确的。因为我们没有候选解，所以不能使用归纳法来求解这个递归方程。但是，可以使用下一节讨论的方法来解决它。

## B.2 用特征方程求解递归方程

我们开发了求解一大类递归方程的方法。

### B.2.1 齐次线性递归

**定义** 有以下形式的递归方程：

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

其中  $k$  和  $a_i$  都是常数，则此递归方程称为常系数齐次线性递归方程。

这种递归方程称为“线性的”，是因为每一项  $t_i$  都只出现在一次幂中。也就是说，不存在诸如  $t_{n-i}^2$ ， $t_{n-i} t_{n-j}$  这样的项。但是，还有另外一条要求，就是不存在  $t_{c(n-i)}$  这样的项，其中  $c$  是不等于 1 的正常数。例如，没有诸如  $t_{n/2}$ ， $t_{3(n-4)}$  这样的项。这种递归方程被称为“齐次的”，是因为这些项的线性组合等于 0。

**例 B.4** 下面这些都是常系数齐次线性递归方程。

$$\begin{aligned} 7t_n - 3t_{n-1} &= 0 \\ 6t_n - 5t_{n-1} + 8t_{n-2} &= 0 \\ 8t_n - 4t_{n-3} &= 0 \end{aligned}$$

**例 B.5** 在 1.2.1 小节讨论的斐波那契序列定义如下：

$t_n = t_{n-1} + t_{n-2}$
$t_0 = 0$
$t_1 = 1$

两边都减去  $t_{n-1}$  和  $t_{n-2}$ ，则有

$$t_n - t_{n-1} - t_{n-2} = 0$$

这表明斐波那契序列是由一个齐次线性递归方程定义的。

接下来看看如何求解齐次线性递归方程。

**例 B.6** 假定有以下递归方程

$t_n - 5t_{n-1} + 6t_{n-2} = 0 \quad (n > 1)$
$t_0 = 0$
$t_1 = 1$

注意，如果设定

$$t_n = r^n$$

则

$$t_n - 5t_{n-1} + 6t_{n-2} = r^n - 5r^{n-1} + 6r^{n-2}$$

因此，如果  $r$  是以下方程的一个解，则  $t_n = r^n$  是该递归方程的一个解。

$$r^n - 5r^{n-1} + 6r^{n-2} = 0$$

因为

$$r^n - 5r^{n-1} + 6r^{n-2} = r^{n-2}(r^2 - 5r + 6)$$

它的根是  $r=0$  及以下方程的根：

$$r^2 - 5r + 6 = 0 \quad (\text{B.1})$$

这些根可通过因式分解求得：

$$r^2 - 5r + 6 = (r-3)(r-2) = 0$$

这些根是  $r=3$  和  $r=2$ 。因此，

$$t_n = 0, \quad t_n = 3^n, \quad t_n = 2^n$$

都是递归方程的解。我们验证  $3^n$ ，将其代入递归方程的左侧，如下所示：

$$\begin{array}{ccc} 3^n & 3^{n-1} & 3^{n-2} \\ \downarrow & \downarrow & \downarrow \\ t_n & = & 5t_{n-1} + 6t_{n-2} \end{array}$$

代入后，左侧变为：

$$\begin{aligned} 3^n - 5(3^{n-1}) + 6(3^{n-2}) &= 3^n - 5(3^{n-1}) + 2(3^{n-1}) \\ &= 3^n - 3(3^{n-1}) = 3^n - 3^n = 0 \end{aligned}$$

这意味着  $3^n$  是该递归方程的一个解。

我们已经求出这个递归方程的三个解，但还有更多个解，这是因为，如果  $3^n$  和  $2^n$  是解，那下面的式子也是它的解：

$$t_n = c_1 3^n + c_2 2^n$$

其中  $c_1$  和  $c_2$  是任意常数。这一结果在习题中获得。尽管这里没有证明，但可以证明，它们是仅有的解。这个表达式是该递归方程的通解。（取  $c_1=c_2=0$ ，平凡解  $t_n=0$  就包含在这个通解中。）我们得到了无限个解，但哪个是这个问题的答案呢？这是由初始条件决定的。回想一下，我们有初始条件

$$t_0=0 \text{ 和 } t_1=1$$

这两个条件决定了  $c_1$  和  $c_2$  的唯一解，如下所示。如果将这个通解应用于两个条件，则得到下面两个方程，其中有两个未知数：

$$\begin{aligned} t_0 &= c_1 3^0 + c_2 2^0 = 0 \\ t_1 &= c_1 3^1 + c_2 2^1 = 1 \end{aligned}$$

这两个等式简化为：

$$\begin{aligned} c_1 + c_2 &= 0 \\ 3c_1 + 2c_2 &= 1 \end{aligned}$$

这个方程组的解为  $c_1=1$  和  $c_2=-1$ 。因此，递归方程的解为：

$$t_n = 1(3^n) - 1(2^n) = 3^n - 2^n$$

如果上面例子中的初始条件不同，则可以获得一个不同解。递归方程实际上表示了一类函数，每个函数对应于一个不同的初始条件赋值。现在来看看，对于例 B.6 中给出的递归方程，在使用以下初始条件时会得到什么方程：

$$t_0=1 \text{ 和 } t_1=2$$

将例 B.6 中的通解应用于每个条件，得到：

$$\begin{aligned} t_0 &= c_1 3^0 + c_2 2^0 = 1 \\ t_1 &= c_1 3^1 + c_2 2^1 = 2 \end{aligned}$$

这两个方程简化为：

$$\begin{aligned} c_1 + c_2 &= 1 \\ 3c_1 + 2c_2 &= 2 \end{aligned}$$

这个方程组的解是  $c_1=0$ ， $c_2=1$ 。因此，以此为初始条件的递归方程的解为：

$$t_n = 0(3^n) + 1(2^n) = 2^n$$

例 B.6 中的方程 B.1 称为该递归方程的特征方程。一般来说，这一方程定义如下。

**定义 常系数齐次线性递归方程**

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

的特征方程 (characteristic equation) 定义为：

$$a_0 r^k + a_1 r^{k-1} + \cdots + a_k r^0 = 0$$

$r^0$  的值就是 1。我们将该项写为  $r^0$  是为了表明特征方程与递归方程之间的关系。

**例 B.7** 下面给出一个递归方程及其特征方程。

$$\begin{array}{r} 5t_n - 7t_{n-1} + 6t_{n-2} = 0 \\ \hline 5r^2 - 7r + 6 = 0 \end{array}$$

我们用箭头表示特征方程的次数为  $k$ （在本例中为 2）。

例 B.6 中用于求解的步骤可扩展为一条定理。为求解一个常系数齐次线性递归方程，只需要引用该定理即可。该定理如下，其证明在本附录的末尾处给出。

**定理 B.1** 设给出一个常系数齐次线性递归方程如下：

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

如果它的特征方程

$$a_0 r^k + a_1 r^{k-1} + \cdots + a_k r^0 = 0$$

有  $k$  个不同解  $r_1, r_2, \dots, r_k$ , 则该递归方程仅有的解是

$$t_n = c_1 r_1^n + c_2 r_2^n + \cdots + c_k r_k^n$$

其中  $c_i$  项为任意常数。

$k$  个常量  $c_i$  的值是由初始条件确定的。我们需要  $k$  个初始值来唯一确定  $k$  个常数。用于确定常数值的方法在下面的例子中演示。

例 B.8 求解以下递归方程

$t_n - 3t_{n-1} - 4t_{n-2} = 0 \quad (n > 1)$
$t_0 = 0$
$t_1 = 1$

(1) 获得特征方程：

$$\begin{array}{c} t_n - 3t_{n-1} - 4t_{n-2} \\ \hline r^2 - 3r - 4 = 0 \end{array}$$

(2) 求解特征方程：

$$r^2 - 3r - 4 = (r-4)(r+1) = 0$$

它的根为  $r=4$  和  $r=-1$ 。

(3) 应用定理 B.1, 得到递归方程的通解：

$$t_n = c_1 4^n + c_2 (-1)^n$$

(4) 将通解应用于初始条件，确定常数值：

$$\begin{array}{l} t_0 = 0 = c_1 4^0 + c_2 (-1)^0 \\ t_1 = 1 = c_1 4^1 + c_2 (-1)^1 \end{array}$$

这些值化简为：

$$\begin{array}{l} c_1 + c_2 = 0 \\ 4c_1 - c_2 = 1 \end{array}$$

这个方程组的解是  $c_1 = 1/5$ ,  $c_2 = -1/5$ 。

(5) 将这些常数代入通解，得到特解：

$$t_n = \frac{1}{5} 4^n - \frac{1}{5} (-1)^n$$

例 B.9 求解生成斐波那契序列的递归方程

$t_n - t_{n-1} - t_{n-2} = 0 \quad (n > 1)$
$t_0 = 0$
$t_1 = 1$

(1) 获得特征方程：

$$\begin{array}{c} t_n - t_{n-1} - t_{n-2} = 0 \\ \hline r^2 - r - 1 = 0 \end{array}$$

(2) 求解该特征方程：

由二次方程的求根公式，可得这个特征方程的根为：

$$r = \frac{1+\sqrt{5}}{2} \text{ 和 } r = \frac{1-\sqrt{5}}{2}$$

(3) 应用定理 B.1, 得到这个递归方程的通解：

$$t_n = c_1 \left( \frac{1+\sqrt{5}}{2} \right)^n + c_2 \left( \frac{1-\sqrt{5}}{2} \right)^n$$

(4) 将此通解应用于初始条件, 确定常数值：

$$\begin{aligned} t_0 &= c_1 \left( \frac{1+\sqrt{5}}{2} \right)^0 + c_2 \left( \frac{1-\sqrt{5}}{2} \right)^0 = 0 \\ t_1 &= c_1 \left( \frac{1+\sqrt{5}}{2} \right)^1 + c_2 \left( \frac{1-\sqrt{5}}{2} \right)^1 = 1 \end{aligned}$$

这些方程化简为：

$$\begin{aligned} c_1 + c_2 &= 0 \\ \left( \frac{1+\sqrt{5}}{2} \right) c_1 + \left( \frac{1-\sqrt{5}}{2} \right) c_2 &= 1 \end{aligned}$$

解这一方程组, 得到  $c_1=1/\sqrt{5}$  和  $c_2=-1/\sqrt{5}$ 。

(5) 将常数代入通解, 得到特解：

$$t_n = \frac{\left[ (1+\sqrt{5})/2 \right]^n - \left[ (1-\sqrt{5})/2 \right]^n}{\sqrt{5}}$$

尽管例 B.9 为第  $n$  个斐波那契项提供了一个显式公式, 却没什么实用价值, 因为表示  $\sqrt{5}$  所需要的精度随着  $n$  的提高而提高。

定理 B.1 要求特征方程的所有  $k$  个根都互不相同。该定理不允许出现如下形式的特征方程：

$$\begin{array}{c} \text{重数} \\ \downarrow \\ (r-1)(r-2)^3 = 0 \end{array}$$

因为  $r-2$  项被升到三次幂, 所以 2 被称为该方程的 3 重根。下面的定理允许一个根是多重的。这个定理的证明在本附录末尾处给出。

**定理 B.2** 设  $r$  是一个常系数齐次线性递归方程的特征方程的  $m$  重根。则

$$t_n = r^n, \quad t_n = nr^n, \quad t_n = n^2r^n, \quad t_n = n^3r^n, \dots, \quad t_n = n^{m-1}r^n$$

都是该递归方程的解。因此, 在这个递归方程的通解中 (定理 B.1 中给出) 会为这些解中的每一个都包含一项。

下面给出这一定理的示例应用。

**例 B.10** 求解以下递归方程

$t_n - 7t_{n-1} + 15t_{n-2} - 9t_{n-3} = 0 \quad (n > 2)$ $t_0 = 0$ $t_1 = 1 \qquad t_2 = 2$
--

(1) 获得特征方程：

$$\begin{array}{c} t_n - 7t_{n-1} + 15t_{n-2} - 9t_{n-3} = 0 \\ \downarrow \\ r^3 - 7r^2 + 15r - 9 = 0 \end{array}$$

(2) 求解该特征方程：

$$\begin{array}{c}
 \text{重数} \\
 \downarrow \\
 r^3 - 7r^2 + 15r - 9 = (r-1)(r-3)^2 = 0
 \end{array}$$

它的根为  $r=1$  和  $r=3$ , 且  $r=3$  为 2 重根。

(3) 应用定理 B.2 获得该递归方程的通解：

$$t_n = c_1 1^n + c_2 3^n + c_3 n 3^n$$

因为 3 是一个两重根, 所以包含了  $3^n$  和  $n 3^n$  的项目。

(4) 将通解应用于该初始条件, 确定常数值:

$$\begin{aligned}
 t_0 &= 0 = c_1 1^0 + c_2 3^0 + c_3 (0)(3^0) \\
 t_1 &= 1 = c_1 1^1 + c_2 3^1 + c_3 (1)(3^1) \\
 t_2 &= 2 = c_1 1^2 + c_2 3^2 + c_3 (2)(3^2)
 \end{aligned}$$

这些值化简为:

$$\begin{aligned}
 c_1 + c_2 &= 0 \\
 c_1 + 3c_2 + 3c_3 &= 2 \\
 c_1 + 9c_2 + 18c_3 &= 3
 \end{aligned}$$

解这一方程组, 得到  $c_1 = -1$ ,  $c_2 = 1$  和  $c_3 = \frac{1}{3}$ 。

(5) 将这些常数代入通解, 以得到特解:

$$\begin{aligned}
 t_n &= (-1)(1^n) + (1)(3^n) + \left(-\frac{1}{3}\right)(n 3^n) \\
 &= -1 + 3^n - n 3^{n-1}
 \end{aligned}$$

**例 B.11** 求解下面的递归方程

$t_n - 5t_{n-1} + 7t_{n-2} - 3t_{n-3} = 0 \quad (n \geq 2)$ $t_0 = 1$ $t_1 = 2 \quad t_2 = 3$
---

(1) 获得特征方程:

$$\begin{array}{l}
 t_n - 5t_{n-1} + 7t_{n-2} - 3t_{n-3} = 0 \\
 \downarrow \\
 r^3 - 5r^2 + 7r - 3 = 0
 \end{array}$$

(2) 求解该特征方程:

$$\begin{array}{c}
 \text{重数} \\
 \downarrow \\
 r^3 - 5r^2 + 7r - 3 = (r-3)(r-1)^2 = 0
 \end{array}$$

它的根为  $r=3$  和  $r=1$ , 且根 1 为 2 重根。

(3) 应用定理 B.2 获得该递归方程的通解:

$$t_n = c_1 3^n + c_2 1^n + c_3 n 1^n$$

(4) 将通解应用于该初始条件, 确定常数值:

$$\begin{aligned}
 t_0 &= 1 = c_1 3^0 + c_2 1^0 + c_3 (0)(1^0) \\
 t_1 &= 2 = c_1 3^1 + c_2 1^1 + c_3 (1)(1^1) \\
 t_2 &= 3 = c_1 3^2 + c_2 1^2 + c_3 (2)(1^2)
 \end{aligned}$$

这些值化简为:

$$\begin{aligned}
 c_1 + c_2 &= 1 \\
 3c_1 + c_2 + c_3 &= 2 \\
 9c_1 + c_2 + 2c_3 &= 3
 \end{aligned}$$

解这一方程组，得到  $c_1=0$ ,  $c_2=1$  和  $c_3=1$ 。

(5) 将这些常数代入通解，以得到特解：

$$\begin{aligned} t_n &= 0(3^n) + 1(1^n) + 1(n1^n) \\ &= 1+n \end{aligned}$$

## B.2.2 非齐次线性递归方程

**定义** 有如下形式的递归方程：

$$a_0t_n + a_1t_{n-1} + \cdots + a_kt_{n-k} = f(n)$$

其中  $k$  和  $a_i$  项为常数， $f(n)$  是一个不为零函数的函数，则这种递归方程称为常系数非齐次线性递归方程。

这里所说的零函数，是指函数  $f(n)=0$ 。如果使用零函数，就会得到一个齐次线性递归方程。对于求解非齐次线性递归方程，还没有已知的通用方法。我们为求解下面的常见特例推导了一种方法。

$$a_0t_n + a_1t_{n-1} + \cdots + a_kt_{n-k} = b^n p(n) \quad (\text{B.2})$$

其中  $b$  为常数， $p(n)$  为  $n$  的多项式。

**例 B.12** 递归方程

$$t_n - 3t_{n-1} = 4^n$$

是递归方程 B.2 的一个例子，其中  $k=1$ ,  $b=4$ ,  $p(n)=1$ 。

**例 B.13** 递归方程

$$t_n - 3t_{n-1} = 4^n(8n+7)$$

是递归方程 B.2 的一个例子，其中  $k=1$ ,  $b=4$ ,  $p(n)=8n+7$ 。

递归方程 B.2 所示的特例可以通过将其转换为一个齐次线性递推方式来求解。下面的例子说明如何实现。

**例 B.14** 求解如下递归方程：

$t_n - 3t_{n-1} = 4^n \quad (n > 1)$
$t_0 = 0$
$t_1 = 4$

因为右边有  $4^n$  项，所以这个递归方程不是齐次的。可以去掉该项，方法如下。

(1) 将原递归方程中的  $n$  用  $n-1$  替代，将该递归方程表示为  $4^{n-1}$  在右侧：

$$t_{n-1} - 3t_{n-2} = 4^{n-1}$$

(2) 将原递归方程除以 4，使该递归方程变为另外一种以  $4^{n-1}$  在右侧的形式：

$$\frac{t_n}{4} - \frac{3t_{n-1}}{4} = 4^{n-1}$$

(3) 原递归方程必然与它的这些版本具有相同解。因此，它还必然与它们的差有些相同解。这意味着可以用第 2 步获得的递归方程减去第 1 步获得的递归方程，消去  $4^{n-1}$  项。结果为：

$$\frac{t_n}{4} - \frac{7t_{n-1}}{4} + 3t_{n-2} = 0$$

可以在方程两端乘以 4，消去分数：

$$t_n - 7t_{n-1} + 12t_{n-2} = 0$$

这是一个齐次线性递归方程，这意味着它可以通过应用定理 B.1 求解。也就是说，可以求解特征方程

$$r^2 - 7r + 12 = (r-3)(r-4) = 0$$

获得通解

$$t_n = c_1 3^n + c_2 4^n$$

并使用初始值  $t_0=0$  和  $t_1=4$ ，以确定特解：

$$t_n = 4^{n+1} - 4(3^n)$$

在例 B.14 中，通解具有以下项：

$$c_1 3^n \text{ 和 } c_2 4^n$$

第一项源自如果该递归方程为齐次时获得的特征方程，而第二项源自该递归方程的非齐次部分，也就是  $b$ 。这个例子中的多项式  $p(n)$  等于 1。如果不是 1，那将这个递归方程转换为齐次方程所需要的操作要更复杂一些。但是，其结果就是在所得齐次线性递归方程的特征方程中为  $b$  增加一个重数。这一结果在下面的定理中给出。这一定理仅作表述，未加证明。其证明步骤类似于例 B.14。

**定理 B.3** 一个如下形式的非齐次线性递归方程

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n)$$

可以转换为具有以下特征方程的齐次线性递归方程

$$(a_0 r^k + a_1 r^{k-1} + \cdots + a_k)(r-b)^{d+1} = 0$$

其中  $d$  是  $p(n)$  的次数。注意，特征方程包括两个部分：

(1) 相应齐次递归方程的特征方程；

(2) 由递归方程非齐次部分得到的项目。

如果右侧有多个类似于  $b^n p(n)$  的项，则每一项都会向特征方程中贡献一项。

在应用这一定理之前，先回想一下，多项式  $p(n)$  的次数就是  $n$  的最高次幂。例如，

多项式	次数
$p(n)=3n^2+4n-2$	2
$p(n)=5n+7$	1
$p(n)=8$	0

现在应用定理 B.3。

**例 B.15** 求解以下递归方程

$t_n - 3t_{n-1} = 4^n (2n+1) \quad (n > 1)$ $t_0 = 0$ $t_1 = 12$
--

(1) 为相应的齐次递归方程获取特征方程：

$$\begin{aligned} t_n - 3t_{n-1} &= 0 \\ r^1 - 3 &= 0 \end{aligned}$$

(2) 从递归方程的非齐次部分获得一项：

$$\begin{array}{c} d \\ \downarrow \\ 4^n (2n^1 + 1) \\ \uparrow \\ b \end{array}$$

由非齐次部分获得的项为：

$$(r-b)^{d+1} = (r-4)^{1+1}$$

(3) 应用定理 B.3，由步骤 1 和 2 获得的项目获取特征方程。该特征方程为：

$$(r-3)(r-4)^2$$

在获得特征方程之后，完全按照线性齐次情景的做法执行。

(4) 求解特征方程：

$$(r-3)(r-4)^2=0$$

它的根是  $r=3$  和  $r=4$ ，根  $r=4$  为 2 重根。

(5) 应用定理 B.2，获取递归方程的通解：

$$t_n = c_1 3^n + c_2 4^n + c_3 n 4^n$$

有三个未知数，但只有两个初始条件。在这种情况下，必须计算在  $n$  取第二大值时，该递归方程的取值，以找出另外一个初始条件。在这种情况下，取值为 2。因为

$$t_2 - 3t_1 = 4^2(2 \times 2 + 1)$$

且  $t_1 = 12$ ，

$$t_2 = 3 \times 12 + 80 = 116$$

在习题中要求完成这两个步骤：(6)确定常数值，(7)将这些常数代入通解，得到

$$t_n = 20(3^n) - 20(4^n) + 8n4^n$$

例 B.16 求解以下递归方程

$t_n - t_{n-1} = n - 1$	$(n > 0)$
$t_0 = 0$	

(1) 为相应的齐次递归方程获取特征方程：

$$\begin{array}{l} t_n - t_{n-1} = 0 \\ r^1 - 1 = 0 \end{array}$$

(2) 从递归方程的非齐次部分获得一项：

$$n - 1 = \overbrace{1^n}^d (n^1 - 1) \quad \begin{matrix} \downarrow \\ b \end{matrix}$$

该项为：

$$(r-1)^{1+1}$$

(3) 应用定理 B.3，由步骤 1 和 2 获得的项目获取特征方程。该特征方程为：

$$(r-1)(r-1)^2$$

(4) 求解特征方程：

$$(r-1)^3 = 0$$

它的根是  $r=1$ ，它是 3 重根。

(5) 应用定理 B.2，获取递归方程的通解：

$$\begin{aligned} t_n &= c_1 1^n + c_2 n 1^n + c_3 n^2 1^n \\ &= c_1 + c_2 n + c_3 n^2 \end{aligned}$$

我们还需要另外两个初始条件：

$$t_1 = t_0 + 1 - 1 = 0 + 0 = 0$$

$$t_2 = t_1 + 2 - 1 = 0 + 1 = 1$$

在习题中要求完成这两个步骤：(6)确定常数值，(7)将这些常数代入通解，得到

$$t_n = \frac{n(n-1)}{2}$$

例 B.17 求解以下递归方程

$$\boxed{\begin{array}{l} t_n - 2t_{n-1} = n + 2^n \quad (n > 1) \\ t_1 = 0 \end{array}}$$

(1) 为相应的齐次递归方程获取特征方程：

$$\begin{array}{c} t_n - 2t_{n-1} = 0 \\ \downarrow \\ r^1 - 2 = 0 \end{array}$$

(2) 在这一情景中，右侧有两项。根据定理 B.3 所述，每一项都对特征方程有贡献，如下所示：

$$\begin{array}{ccc} d & & d \\ \downarrow & & \downarrow \\ n = (1^n)n^1 & & 2n = (2^n)n^0 \\ \uparrow & & \uparrow \\ b & & b \end{array}$$

这两项为：

$$(r-1)^{1+1} \text{ 和 } (r-2)^{0+1}$$

(3) 应用定理 B.3，由所有项目获取特征方程：

$$(r-2)(r-1)^2(r-2) = (r-2)^2(r-1)^2$$

习题中将要求完成这一问题。

### B.2.3 变量变换（域变换）

有些递归方程的形式无法用定理 B.3 解决，但对变量进行转换后，可以将方程转换为一种能用该定理求解的新递归方程。下面的例子说明了这一方法。在这些例子中，原递归方程表示为  $T(n)$ ，这是因为  $t_k$  被用于表示新递归方程。符号  $T(n)$  与  $t_n$  的含义相同，即每个  $n$  值都关联着一个独一无二的数字。

例 B.18 求解以下递归方程

$$\boxed{\begin{array}{l} T(n) = T\left(\frac{n}{2}\right) + 1 \quad (n > 1, \text{ 且 } n \text{ 为 } 2 \text{ 的幂}) \\ T(1) = 1 \end{array}}$$

回想一下，在 B.1 节已经使用归纳法解决了这一递归方程。现在再次来解这一方程，说明变量变换方法。该递归方程因为具有  $n/2$  项，所以其形式无法通过应用定理 B.3 求解。可以将它转换为能用定理 B.3 求解的递归方程。首先设

$$n = 2^k, \text{ 这意味着 } k = \lg n$$

第二，用  $2^k$  代替递归方程中的  $n$ ，得到：

$$\begin{aligned} T(2^k) &= T\left(\frac{2^k}{2}\right) + 1 \\ &= T(2^{k-1}) + 1 \end{aligned} \tag{B.3}$$

接下来，在递归方程 B.3 中设

$$t_k = T(2^k)$$

以得到新的递归方程：

$$t_k = t_{k-1} + 1$$

这个新递归方程的形式可以运用定理 B.3 来解决。因此，通过应用该定理，可以确定其通解为：

$$t_k = c_1 + c_2 k$$

现在可以通过以下两个步骤获得原递归方程的通解。

(1) 用  $T(2^k)$  代替通解中的  $t_k$ , 得到新的递归方程:

$$T(2^k)=c_1+c_2k$$

(2) 将第(1)步所得方程中的  $2^k$  用  $n$  代替,  $k$  用  $\lg n$  代替:

$$T(n)=c_1+c_2\lg n$$

有了原递归方程的通解, 我们就可以正常进行后续处理。也就是说, 我们使用初始条件  $T(1)=1$  来确定第二初始条件, 然后再计算常数值, 以得到:

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

**例 B.19** 求解以下递归方程

$T(n)=2T\left(\frac{n}{2}\right)+n-1 \quad (n>1, n \text{ 为 } 2 \text{ 的幂})$ $T(1)=0$
---

回想一下, 在例子 B.3 中无法用归纳法求解这个递归方程。这里通过变量的变换来求解它。第一, 用  $2^k$  代替  $n$ , 得出:

$$\begin{aligned} T(2^k) &= 2T\left(\frac{2^k}{2}\right)+2^k-1 \\ &= 2T(2^{k-1})+2^k-1 \end{aligned}$$

接下来, 在此方程中设

$$t_k=T(2^k)$$

得到:

$$t_k=2t_{k-1}+2^k-1$$

向这个新递归方程应用定理 B.3, 得到:

$$t_k=c_1+c_22^k+c_3k2^k$$

执行以下这些步骤, 给出原递归方程的通解。

(1) 用  $T(2^k)$  代替新递归方程通解中的  $t_k$ :

$$T(2^k)=c_1+c_22^k+c_3k2^k$$

(2) 将第(1)步所得方程中的  $2^k$  用  $n$  代替,  $k$  用  $\lg n$  代替:

$$T(n)=c_1+c_2n+c_3n\lg n$$

下面正常进行。也就是说, 使用初始条件  $T(1)=0$ , 再确定两个初始条件, 然后计算这些常数的值。得到的解为:

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

**例 B.20** 求解以下递归方程:

$T(n)=7T\left(\frac{n}{2}\right)+18\left(\frac{n}{2}\right)^2 \quad (n>1, \text{ 且 } n \text{ 为 } 2 \text{ 的幂})$ $T(1)=0$
---

用  $2^k$  代替递归方程中的  $n$ , 得到:

$$T(2^k)=7T(2^{k-1})+18(2^{k-1})^2 \tag{B.4}$$

在递归方程 (B.4) 中设

$$t_k=T(2^k)$$

得到:

$$t_k = 7t_{k-1} + 18(2^{k-1})^2$$

这个递归方程看起来与定理 B.3 所需要的类型不完全一样，但可以让它变成那种类型，如下所示：

$$\begin{aligned} t_k &= 7t_{k-1} + 18(2^{k-1})^2 \\ &= 7t_{k-1} + 18(4^{k-1}) \\ &= 7t_{k-1} + 4^k \left(\frac{18}{4}\right) \end{aligned}$$

向这个新递归方程应用定理 B.3，得到：

$$t_k = c_1 7^k + c_2 4^k$$

执行以下这些步骤，给出原递归方程的通解。

(1) 用  $T(2^k)$  代替通解中的  $t_k$ ：

$$T(2^k) = c_1 7^k + c_2 4^k$$

(2) 将第(1)步所得方程中的  $2^k$  用  $n$  代替， $k$  用  $\lg n$  代替：

$$\begin{aligned} T(n) &= c_1 7^{\lg n} + c_2 4^{\lg n} \\ &= c_1 n^{\lg 7} + c_2 n^2 \end{aligned}$$

使用初始条件  $T(1)=0$ ，确定第二个初始条件，然后计算这些常数的值。解为：

$$T(n) = 6n^{\lg 7} - 6n^2 \approx 6n^{2.81} - 6n^2$$

### B.3 用代入法求解递归方程

有的递归方程可以用一种代入法求解。如果无法使用前两节中的方法得到解，可以尝试这一方法。下面的例子演示了代入法。

**例 B.21** 求解以下递归方程：

$t_n = t_{n-1} + n \quad (n > 1)$
$t_1 = 1$

从某种意义上来说，代入法与归纳法相反。也就是说，我们从  $n$  开始，然后反向处理：

$$\begin{aligned} t_n &= t_{n-1} + n \\ t_{n-1} &= t_{n-2} + n - 1 \\ t_{n-2} &= t_{n-3} + n - 2 \\ &\vdots \\ t_2 &= t_1 + 2 \\ t_1 &= 1 \end{aligned}$$

然后将每个等式代入前一个等式，如下所示：

$$\begin{aligned} t_n &= t_{n-1} + n \\ &= t_{n-2} + n - 1 + n \\ &= t_{n-3} + n - 2 + n - 1 + n \\ &\vdots \\ &= t_1 + 2 + \dots + n - 2 + n - 1 + n \\ &= 1 + 2 + \dots + n - 2 + n - 1 + n \\ &= \sum_{i=1}^n i = \frac{n(n+1)}{2} \end{aligned}$$

最后一个等式是附录 A 中例 A.1 的结果。

例 B.21 中的递归方程可以使用特征方程求解。下例中的递归方程则不能。

例 B.22 求解以下递归方程：

$$\boxed{\begin{aligned} t_n &= t_{n-1} + \frac{2}{n} & (n > 1) \\ t_1 &= 0 \end{aligned}}$$

首先，从  $n$  开始反向处理：

$$\begin{aligned} t_n &= t_{n-1} + \frac{2}{n} \\ t_{n-1} &= t_{n-2} + \frac{2}{n-1} \\ t_{n-2} &= t_{n-3} + \frac{2}{n-2} \\ &\vdots \\ t_2 &= t_1 + \frac{2}{2} \\ t_1 &= 0 \end{aligned}$$

将每个方程代入前一个方程，当  $n$  不太小时：

$$\begin{aligned} t_n &= t_{n-1} + \frac{2}{n} \\ &= t_{n-2} + \frac{2}{n-1} + \frac{2}{n} \\ &= t_{n-3} + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} \\ &\vdots \\ &= t_1 + \frac{2}{2} + \cdots + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} \\ &= 0 + \frac{2}{2} + \cdots + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} \\ &= 2 \sum_{i=2}^n \frac{1}{i} \approx 2 \ln n \end{aligned}$$

近似等式由附录 A 中的例 A.9 得到的。

## B.4 将 $n$ 是正常数 $b$ 的幂时获得的结果推广到一般 $n$ 值

介绍下面的内容时，假定读者熟悉第 1 章中的内容。

在一些递归算法中，只有当  $n$  是某一底数  $b$  ( $b$  是一个正常数) 的幂时，才能轻松确定算法的精确时间复杂度。底数  $b$  经常是 2。对于许多分而治之算法（见第 2 章），这一点尤其成立。从直觉上来说，如果一个结果在  $n$  为  $b$  的幂时成立，那对于一般的  $n$  值也应当近似成立。例如，如果对于某一算法，已知在  $n$  为 2 的幂时有

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

那对于一般的  $n$  值，似乎也应当能够得出如下结论：

$$T(n) \in \Theta(n\lg n)$$

事实表明，通常是可以得出这种结论的。接下来讨论能做如此推论的情景。首先需要一些定义。对于其定

义域和值域为实数任意子集的任意函数，它们都适用，但这里仅针对复杂度函数给出这些定义，这是因为我们现在关心的就是这些函数（所谓复杂度函数就是将正整数映射到正实数的函数）。

**定义** 如果一个复杂度函数  $f(n)$  在  $n$  变大时也总是变大，那就说它是严格递增的。也就是说，若  $n_1 > n_2$ ，则

$$f(n_1) > f(n_2)$$

图 B-1a 中的函数是严格递增的。（为清楚起见，图 B-1 中各函数的定义域都是非负实数。）我们在算法分析时遇到的许多函数，对于  $n$  的非负值都是严格递增的。例如，只要  $n$  是非负数， $\lg n$ 、 $n$ 、 $n \lg n$ 、 $n^2$  和  $2^n$  都是严格递增的。

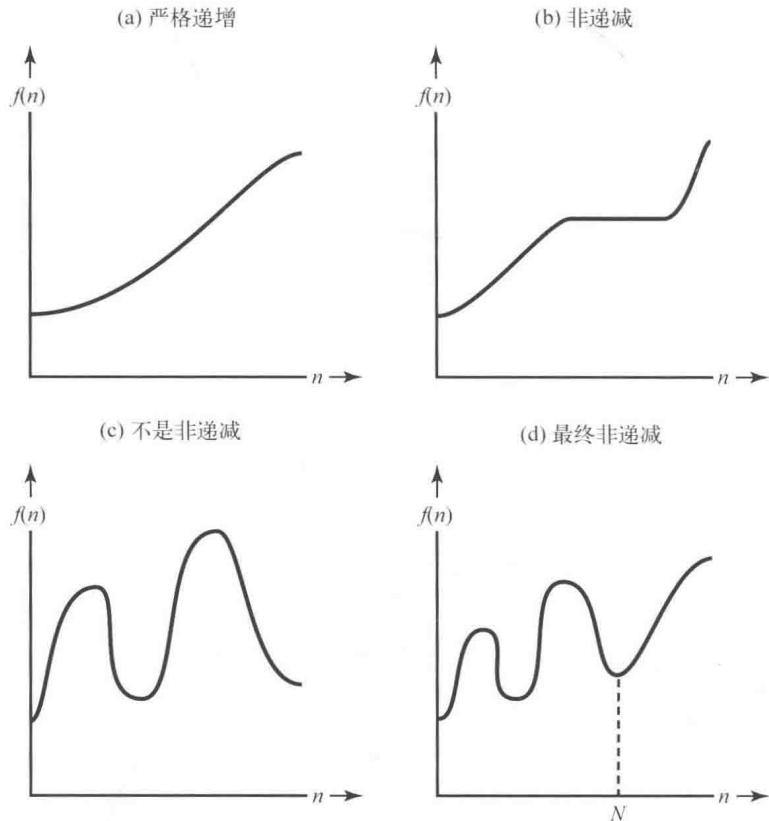


图 B-1 四个函数

**定义** 如果一个复杂度函数  $f(n)$  在  $n$  增大时从来不会变小，则说该函数是**非递减的**。也就是说，若  $n_1 > n_2$ ，则

$$f(n_1) \geq f(n_2)$$

任何严格递增的函数都是非递减的，而一个可能变为水平的函数是非递减的，但不是严格递增的。图 B-1b 中的函数是这种函数的一个例子。图 B-1c 中的函数不是非递减的。

大多数算法的时间（或内存）复杂度通常都是非递减的，因为处理输入时所需要的时间通常不会因为输入规模的增大而下降。研究图 B-1，只要函数为非递减的，似乎就能将  $n$  为  $b$  的幂时获得的分析结果扩展为一般的  $n$  值。例如，假定当  $n$  为 2 的幂时，已经确定了  $f(n)$  的值。对于图 B-1c 的函数，在比如说  $2^3=9$  和  $2^4=16$  之间可以发生任何事情。因此，根据该函数在 8 和 16 处的值，无法从该函数在 8 到 16 之间的行为得出任何结论。但是，对于非递减函数  $f(n)$ ，若  $8 \leq n \leq 16$ ，则

$$f(8) \leq f(n) \leq f(16)$$

看起来，当  $n$  为 2 的幂时，似乎能够根据  $f(n)$  的值确定  $f(n)$  的次数。事实上，从直观上看起来正确的内容，可以针对一大类函数得到证明。在给出表述上述内容的定理之前，先回忆一下，次数仅与长期特性有关。因为一个函数的初始值并不重要，所以该定理只要求函数最终是非递减的。我们有以下定义：

**定义** 如果对于某一点之后的所有  $n$  值，复杂度函数  $f(n)$  从来不会在  $n$  变大时变小，就说这个函数  $f(n)$  是最终非递减的。也就是说，存在一个  $N$  值，使得当  $n_1 > n_2 > N$  时，有

$$f(n_1) \geq f(n_2)$$

任何非递减函数都是最终非递减的。图 B-1d 所示函数是一个最终非递减函数的例子，它不是非递减的。我们先给出下面的定义，然后再给出定理，用来扩展当  $n$  为  $b$  的幂时的结果：

**定义** 如果一个复杂度函数  $f(n)$  是最终非递减的，而且如果

$$f(2n) \in \Theta(f(n))$$

就说此函数是平滑的。

**例 B.23** 当  $k \geq 0$  时，函数  $\lg n$ 、 $n$ 、 $n \lg n$  和  $n^k$  都是平滑的。我们将对  $\lg n$  进行证明。在习题中将要求针对其他函数进行证明。我们已经注意到  $\lg n$  是最终非递减的。关于第二个条件，有

$$\lg(2n) = \lg 2 + \lg n \in \Theta(\lg n)$$

**例 B.24** 函数  $2^n$  是不平滑的，因为 1.4.2 节“阶的性质”可推出

$$2^n = o(4^n)$$

因此

$$2^{2n} = 4^n \text{ 不属于 } \Theta(2^n)$$

现在给出一个定理，可以用来推广在  $n$  为  $b$  的幂时获得的结果。其证明在本附录末尾处给出。

**定理 B.4** 设  $b \geq 2$  是一个整数， $f(n)$  是一个平滑复杂度函数， $T(n)$  是最终非递减复杂度函数。若

$$T(n) \in \Theta(f(n)) \quad (n \text{ 是 } b \text{ 的幂})$$

则

$$T(n) \in \Theta(f(n))$$

此外，如果用“大  $O$ ”、“ $\Omega$ ”或“小  $o$ ”代替  $\Theta$ ，上述推导同样成立。

“当  $n$  是  $b$  的幂时， $T(n) \in \Theta(f(n))$ ”是指，已知  $\Theta$  的一般条件在  $n$  限制为  $b$  的一个幂时成立。注意，在定理 B.4 中的附加条件是  $f(n)$  为平滑的。

接下来应用定理 B.4。

**◆例 B.25** 假定对于某个复杂度函数，已经确定了

$$\boxed{\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 && (n > 1) \\ T(1) &= 1 \end{aligned}}$$

当  $n$  为 2 的幂时，有例 B.18 中的递归方程。因此，根据该范例，有

$$T(n) = \lg n + 1 \in \Theta(\lg n) \quad (n \text{ 为 } 2 \text{ 的幂})$$

因为  $\lg n$  是平滑的，所以只需要证明  $T(n)$  是最终非递减的，就可以应用定理 B.4 得出如下结论：

$$T(n) \in \Theta(\lg n)$$

有人可能会想，根据“ $\lg n + 1$  是最终非递减的”这一事实得出“ $T(n)$  是最终非递减的”。但我们并不能这样做，因为我们只知道当  $n$  为 2 的幂时， $T(n) = \lg n + 1$ 。在仅给定这一事实， $T(n)$  可以在 2 的幂之间呈现任意可能特性。

我们采用归纳法来证明：对于  $n \geq 2$ ，若  $1 \leq k < n$ ，则

$$T(k) \leq T(n)$$

以此证明  $T(n)$  是最终非递减的。

归纳基础：对于  $n=2$ ，

$$\begin{aligned} T(1) &= 1 \\ T(2) &= T\left(\left\lfloor \frac{2}{2} \right\rfloor\right) + 1 = T(1) + 1 = 1 + 1 = 2 \end{aligned}$$

于是，

$$T(1) \leq T(2)$$

归纳假设：做出归纳假设的一种方式是，假定该命题对于所有  $m \leq n$  成立。然后，和通常一样，证明它对于  $n+1$  是成立的。我们需要的表述方式如下。设  $n$  是大于或等于 2 的任意整数。假设对于所有  $m \leq n$ ，如果  $k < m$ ，则

$$T(k) \leq T(m)$$

归纳步骤：因为在归纳假设中已经假定，对于  $k < n$ ，有

$$T(k) \leq T(n)$$

只需要证明：

$$T(n) \leq T(n+1)$$

为此，不难看出，如果  $n \geq 1$ ，则

$$\left\lfloor \frac{n}{2} \right\rfloor \leq \left\lfloor \frac{n+1}{2} \right\rfloor \leq n$$

因此，根据归纳假设，有

$$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \leq T\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right)$$

利用该递归方程，可得：

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \leq T\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + 1 = T(n+1)$$

证毕。

最后设计一种一般方法，用于确定某些常见递归方程的次数。

**定理 B.5** 假设一个复杂度函数  $T(n)$  是最终非递减的，且满足

$T(n) = aT\left(\frac{n}{b}\right) + cn^k \quad (n \geq 1, \text{ 且 } n \text{ 为 } b \text{ 的幂})$
$T(1) = d$

其中  $b \geq 2$ ,  $k \geq 0$  是常整数， $a, c, d$  为常数，且满足  $a > 0, c > 0, d \geq 0$ 。于是，

$$T(n) \in \begin{cases} \Theta(n^k) & (a < b^k) \\ \Theta(n^k \lg n) & (a = b^k) \\ \Theta(n^{\log_b a}) & (a > b^k) \end{cases} \quad (\text{B.5})$$

此外，若在该递归式的表述中，将

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k$$

代以

$$T(n) \leq aT\left(\frac{n}{b}\right) + cn^k \text{ 或 } T(n) \geq aT\left(\frac{n}{b}\right) + cn^k$$

则在分别以“大 O”或  $\Omega$  代替  $\Theta$  时，结果 B.5 成立。

利用特征方程，然后再应用定理 B.4，解出该一般递归方程，即可证明这一定理。定理 B.5 的示例应用如下。

**例 B.26** 假定  $T(n)$  是最终非递减的，而且满足

$a$	$b$	$k$	
↓	↓	↓	( $n > 1$ , $n$ 为 4 的幂 )
$T(n) = 8T(n/4) + 5n^2$			
$T(1) = 3$			

根据定理 B.5，因为  $8 < 4^2$ ，所以

$$T(n) \in \Theta(n^2)$$

**例 B.27** 假定  $T(n)$  是最终非递减的，而且满足

$a$	$b$	$k$	
↓	↓	↓	( $n > 1$ , $n$ 为 3 的幂 )
$T(n) = 9T(n/3) + 5n^1$			
$T(1) = 7$			

根据定理 B.5，因为  $9 > 3^1$ ，所以

$$T(n) \in \Theta(n^{\log_3 9}) = \Theta(n^2)$$

定理 B.5 的表述是为了尽可能简单地引入一个重要定理。它实际上是如下定理在常数  $s$  等于 1 时的一个特例。

**定理 B.6** 假定复杂度函数  $T(n)$  是最终非递减的，而且满足

$T(n) = aT\left(\frac{n}{b}\right) + cn^k$	( $n > 2$ , $n$ 为 $b$ 的幂 )
$T(s) = d$	

其中  $s$  是一个常数，且是  $b$  的一个幂， $b \geq 2$  和  $k \geq 0$  是常整数， $a, c$  和  $d$  是常数，满足  $a > 0$ ， $c > 0$  和  $d \geq 0$ 。则定理 B.5 中的结果仍然成立。

**例 B.28** 假定  $T(n)$  是最终非递减的，而且满足

$a$	$b$	$k$	
↓	↓	↓	( $n > 64$ , $n$ 为 2 的幂 )
$T(n) = 8T(n/2) + 5n^3$			
$T(64) = 200$			

根据定理 B.6，因为  $8=2^3$ ，所以

$$T(n) \in \Theta(n^3 \lg n)$$

我们对递归方程求解的讨论到此结束。还有一种求解递归方程的方法，是使用“生成函数”。这一方法在 Sahni (1988 年) 的文献中讨论。Bentley、Haken 和 Sax (1980 年) 提供了一种通用方法，用于求解在分析分而治之算法时 (见第 2 章) 得到的递归方程。

## B.5 定理的证明

在证明定理 B.1 时会用到下面的引理。

**引理 B.1** 假定有齐次线性递归方程

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

它的特征方程为：

$$a_0 r_1^n + a_1 r_1^{n-1} + \cdots + a_k = 0$$

如果  $r_1$  是特征方程的一个根，则

$$t_n = r_1^n$$

是递归方程的一个解。

**证明：**若对于  $i=n-k, \dots, n$ , 用  $r_1^i$  代替递归方程中的  $t_i$ , 则得到：

$$\begin{aligned} a_0 r_1^n + a_1 r_1^{n-1} + \cdots + a_k r_1^{n-k} &= r_1^{n-k} (a_0 r_1^k + a_1 r_1^{k-1} + \cdots + a_k) \\ &= r_1^{n-k} (0) \\ &= 0 \quad \uparrow \\ &\text{因为 } r_1 \text{ 是特征方程的一个根} \end{aligned}$$

因此,  $r_1^n$  是递归方程的一个解。

**定理 B.1 的证明** 不难看出, 对于一个线性齐次递归方程, 一个常数乘以任意解、任意两个解之和, 都是该递归方程的解。因此, 可以应用引理 B.1 得出结论: 若

$$r_1, r_2, \dots, r_k$$

是特征方程的  $k$  个不同根, 则

$$c_1 r_1^n + c_2 r_2^n + \cdots + c_k r_k^n$$

是该递归方程的一个解(其中的  $c_i$  项是任意常数)。尽管直接在此处给出, 但可以证明它们是仅有的解。

**◆定理 B.2 的证明** 我们证明当重数  $m$  等于 2 的情景。 $m$  更大时的情景就是一种简单的推广。设  $r_1$  是一个两重根。设

$$\begin{aligned} p(r) &= a_0 r^k + a_1 r^{k-1} + \cdots + a_k \quad \{ \text{特征方程} \} \\ q(r) &= r^{n-k} p(r) = a_0 r^n + a_1 r^{n-1} + \cdots + a_k r^{n-k} \\ u(r) &= r q'(r) = a_0 n r^n + a_1 (n-1) r^{n-1} + \cdots + a_k (n-k) r^{n-k} \end{aligned}$$

其中  $q'(r)$  表示一阶导数。如果用  $r_1^i$  代替递归方程中的  $t_i$ , 则得到  $u(r_1)$ 。因此, 如果可以证明  $u(r_1)=0$ , 则可以得出结论:  $t_n = n r_1^n$  是递归方程的一个解, 且完成证明。

为此, 有

$$\begin{aligned} u(r) &= r q'(r) \\ &= r \left[ \left( r^{n-k} \right) p(r) \right]' \\ &= r \left[ r^{n-k} p'(r) + p(r)(n-k) r^{n-k-1} \right] \end{aligned}$$

因此, 为证明  $u(r_1)=0$ , 只需要证明  $p(r_1)$  和  $p'(r_1)$  都等于 0。证明如下: 因为  $r_1$  是特征方程  $p(r)$  的一个两重解, 所以存在一个  $v(r)$ , 满足

$$p(r) = (r - r_1)^2 v(r)$$

因此,

$$p'(r) = (r - r_1)^2 v'(r) + 2v(r)(r - r_1)$$

且  $p(r_1)$  和  $p'(r_1)$  都等于 0。证毕。

◆定理 B.4 的证明 我们给出针对“大 O”的证明。针对  $\Omega$  和  $\Theta$  的证明可以通过类似方式得出。对于所有  $n$ , 只要它是  $b$  的一个幂, 就有  $T(n) \in O(f(n))$ , 所以存在一个正数  $c_1$  和一个非负整数  $N_1$ , 使得对于  $n > N_1$ , 且  $n$  为  $b$  的一个幂, 有

$$T(n) \leq c_1 \times f(n) \quad (\text{B.6})$$

对于任意给定整数  $n$ , 存在唯一的  $k$  值满足

$$b^k \leq n < b^{k+1} \quad (\text{B.7})$$

有可能证明, 对于一个平滑函数, 如果  $b \geq 2$ , 则有

$$f(bn) \in \Theta(f(n))$$

也就是说, 如果此条件对于 2 成立, 则对于任意  $b > 2$  也成立。因为, 存在一个正整数  $c_2$  和一个非负整数  $N_2$ , 使得对于  $n > N_2$ , 有

$$f(bn) \leq c_2 f(n)$$

因此, 如果  $b^k \geq N_2$ , 有

$$f(b^{k+1}) = f(b \times b^k) \leq c_2 f(b^k) \quad (\text{B.8})$$

因为  $T(n)$  和  $f(n)$  都是最终非递减的, 所以存在一个  $N_3$ , 使得对于  $m > n > N_3$ , 有

$$T(n) \leq T(m), f(n) \leq f(m) \quad (\text{B.9})$$

设  $r$  很大, 满足

$$b^r > \max(N_1, N_2, N_3)$$

如果  $n > b^r$  和  $k$  是与不等式 B.7 中的  $n$  相对应的值, 则

$$b^k \geq b^r$$

因此, 根据不等式 B.6、B.7、B.8 和 B.9, 对于  $n > b^r$ , 有

$$T(n) \leq T(b^{k+1}) \leq c_1 f(b^{k+1}) \leq c_1 c_2 f(b^k) \leq c_1 c_2 f(n)$$

这意味着:

$$T(n) \in O(f(n))$$

## B.6 习题

### B.1 节

1. 用归纳法验证以下每个递归方程的候选解。

$$(a) t_n = 4t_{n-1} \quad (n > 1)$$

$$t_1 = 3$$

候选解为  $t_n = 3(4^{n-1})$ 。

$$(b) t_n = t_{n-1} + 5 \quad (n > 1)$$

$$t_1 = 2$$

候选解为  $t_n = 5n - 3$ 。

$$(c) t_n = t_{n-1} + n \quad (n > 1)$$

$$t_1 = 1$$

候选解为  $t_n = \frac{n(n+1)}{2}$ 。

$$(d) t_n = t_{n-1} + n^2 \quad (n > 1)$$

$$t_1 = 1$$

$$\text{候选解为 } t_n = \frac{n(n+1)(2n+1)}{6}.$$

$$(e) t_n = t_{n-1} + \frac{1}{n(n+1)} \quad (n > 1)$$

$$t_1 = \frac{1}{2}$$

$$\text{候选解为 } t_n = \frac{n}{(n+1)}.$$

$$(f) t_n = 3t_{n-1} + 2^n \quad (n > 1)$$

$$t_1 = 1$$

$$\text{候选解为 } t_n = 5(3^{n-1}) - 2^{n+1}.$$

$$(g) t_n = 3t_{n/2} + n \quad (n \geq 1, n \text{ 是 } 2 \text{ 的幂})$$

$$t_1 = \frac{1}{2}$$

$$\text{候选解为 } t_n = \frac{5}{2}n^{\lg 3} - 2n.$$

$$(h) t_n = nt_{n-1} \quad (n > 0)$$

$$t_0 = 1$$

$$\text{候选解为 } t_n = n!.$$

2. 为序列 2, 6, 18, 54, … 的第  $n$  项编写一个递归方程，并使用归纳法证明候选解  $s_n = 2(3^{n-1})$ 。

3. 汉诺塔问题（见第 2 章的第 17 题）所需要的移动次数（当有  $n$  个环时，用  $m_n$  表示）由下面的递归方程给出：

$$\begin{aligned} m_n &= 2m_{n-1} + 1 \quad (n > 1) \\ m_1 &= 1 \end{aligned}$$

使用归纳法证明，这一递归方程的解为  $m_n = 2^n - 1$ 。

4. 下面的算法返回数组  $S$  中最大元素的位置。编写一个递归方程，表示在找出该最大元素时所需要的比较次数  $t_n$ 。使用归纳法证明该方程的解为  $t_n = n - 1$ 。

```
index max_position(index low, index high)
{
    index position;

    if (low == high)
        return low;
    else{
        position = max_position(low+1, high);
        if (S[low] > S[position])
            position = low;
        return position;
    }
}
```

顶级调用为：

```
max_position(1, n);
```

5. 古希腊人对由几何图形得到的序列非常感兴趣，比如下面的三角数：

$$\begin{array}{c} \text{O} \\ \text{O} \quad \text{OO} \\ \text{O}, \text{ O O}, \text{ O O O}, \dots \end{array} \rightarrow (1, 3, 6, \dots)$$

为这一序列的第  $n$  项编写一个递归方程，猜测一个解，并使用归纳法验证你的解。

6. 有  $n$  条直线，其中每一对直线都相交，但不会有超过两条直线交于同一条，它们可以将一个平面划分为多少个区域？为  $n$  条线划分的区域数编写一个递归方程，为你的方程猜测一个解，并使用归纳法验证你的解。
7. 证明： $B(n, k) = \binom{n+k}{n}$  是以下递归方程的解。

$$B(n, k) = B(n-1, k) + B(n, k-1) \quad (n > 0, k > 0)$$

$$B(n, 0) = 1$$

$$B(0, k) = 1$$

8. 编写并实现一个算法，用于计算以下递推式的值，并使用不同问题实例运行该算法。利用所得结果为这一递归方程猜测一个解，使用归纳法验证你的解。

$$t_n = t_{n-1} + 2n - 1 \quad (n > 1)$$

$$t_1 = 1$$

## B.2 节

9. 指出 B.1 节各题中的哪些递归方程属于以下各类别。

(a) 线性方程

(b) 齐次方程

(c) 常系数方程

10. 为 B.1 节的所有常系数线性递归方程找出特征方程。

11. 证明：如果  $f(n)$  和  $g(n)$  都是一个常系数线性齐次递归方程的解，则  $c \times f(n) + d \times g(n)$  也是它的一个解（其中  $c$  和  $d$  是常数）。

12. 使用特征方程求解以下递归方程。

$$(a) t_n = 4t_{n-1} - 3t_{n-2} \quad (n > 1)$$

$$t_0 = 0$$

$$t_1 = 1$$

$$(b) t_n = 3t_{n-1} - 2t_{n-2} + n^2 \quad (n > 1)$$

$$t_0 = 0$$

$$t_1 = 1$$

$$(c) t_n = 5t_{n-1} - 6t_{n-2} + 5^n \quad (n > 1)$$

$$t_0 = 0$$

$$t_1 = 1$$

$$(d) t_n = 5t_{n-1} - 6t_{n-2} + n^2 - 5n + 7^n \quad (n > 1)$$

$$t_0 = 0$$

$$t_1 = 1$$

13. 完成例 B.15 所给递归方程的解。

14. 完成例 B.16 所给递归方程的解。

15. 使用特征方程求解下面的递归方程。

$$(a) t_n = 6t_{n-1} - 9t_{n-2} \quad (n > 1)$$

$$t_0 = 0$$

$$t_1 = 1$$

(b)  $t_n=5t_{n-1}-8t_{n-2}+4t_{n-3}$  ( $n \geq 2$ )

$$t_0=0$$

$$t_1=1$$

$$t_2=1$$

(c)  $t_n=2t_{n-1}-t_{n-2}+n^2+5^n$  ( $n \geq 1$ )

$$t_0=0$$

$$t_1=1$$

(d)  $t_n=6t_{n-1}-9t_{n-2}+(n^2-5n)7^n$  ( $n \geq 1$ )

$$t_0=0$$

$$t_1=1$$

16. 完成例 B.17 所给递归方程的解。

17. 证明以下递归方程

$t_n=(n-1)t_{n-1}+(n-1)t_{n-2}$  ( $n \geq 2$ )

$$t_1=0$$

$$t_2=1$$

可写为

$t_n=nt_{n-1}+(-1)^n$  ( $n \geq 1$ )

$$t_1=0$$

18. 求解第 17 题的递归方程。这个解给出了  $n$  个对象的错位排列（所有数字都不在正确位置）的个数。

19. 用特征方程求解以下递归方程。

(a)  $T(n)=2T\left(\frac{n}{3}\right)+\log_3 n$  ( $n \geq 1$ , 且  $n$  为 3 的幂)

$$T(1)=0$$

(b)  $T(n)=10T\left(\frac{n}{5}\right)+n^2$  ( $n \geq 1$ , 且  $n$  为 5 的幂)

$$T(1)=0$$

(c)  $nT(n)=(n-1)T(n-1)+3$  ( $n \geq 1$ )

$$T(1)=1$$

(d)  $nT(n)=3(n-1)T(n-1)-2(n-2)T(n-2)+4^n$  ( $n \geq 1$ )

$$T(0)=0$$

$$T(1)=0$$

(e)  $nT^2(n)=5(n-1)T^2(n-1)+n^2$  ( $n \geq 0$ )

$$T(0)=6$$

### B.3 节

20. 用代入法求解第 1 题中的递归方程。

### B.4 节

21. 证明：

(a)  $f(n)=n^3$  是严格递增函数。

(b)  $g(n)=2n^3-6n^2$  是最终非递减函数。

22. 如果一个函数  $f(n)$  对于所有  $n$  值都是非递减的、非递增的，可以对该函数得出什么结论？

23. 证明以下函数是平滑的。

(a)  $f(n)=n\lg n$

(b)  $g(n)=n^k$  (对于所有  $k \geq 0$ )

24. 假定在每种情况下  $T(n)$  都为最终非递减的，使用定理 B.5 确定以下递归方程的次数。

$$(a) T(n) = 2T\left(\frac{n}{5}\right) + 6n^3 \quad (n \geq 1, n \text{ 为 } 5 \text{ 的幂})$$

$$T(1) = 6$$

$$(b) T(n) = 40T\left(\frac{n}{3}\right) + 2n^3 \quad (n \geq 1, n \text{ 为 } 3 \text{ 的幂})$$

$$T(1) = 5$$

$$(c) nT(n) = 16T\left(\frac{n}{2}\right) + 7n^4 \quad (n \geq 1, n \text{ 为 } 2 \text{ 的幂})$$

$$T(1) = 1$$

25. 假定在每种情况下  $T(n)$  都为最终非递减的，使用定理 B.6 确定以下递归方程的次数。

$$(a) T(n) = 14T\left(\frac{n}{5}\right) + 6n \quad (n \geq 25, n \text{ 为 } 5 \text{ 的幂})$$

$$T(25) = 60$$

$$(b) T(n) = 4T\left(\frac{n}{4}\right) + 2n^2 \quad (n \geq 16, n \text{ 为 } 4 \text{ 的幂})$$

$$T(16) = 50$$

26. 给定  $g(n) \in \Theta(n)$ ，已知在  $a > c$  时，以下递归方程

$$T(n) = aT\left(\frac{n}{c}\right) + g(n) \quad (n \geq 1, n \text{ 为 } c \text{ 的幂})$$

$$T(1) = d$$

有以下解：

$$T(n) \in \Theta(n^{\log_c a})$$

证明：如果假定

$$g(n) \in O(n^t), \text{ 其中 } t < \log_c a$$

则该递归方程具有相同解。

# 附录 C

## 不交集的数据结构

Kruskal 的算法（4.1.2 节的算法 4.2）要求我们创建不相交子集，每个子集中包含图中的一个不同顶点，然后再重复这些子集，直到所有顶点都在同一集合中。为实现这一算法，需要为不交集准备数据结构。不交集有许多其他非常有用的应用。例如，它们在 4.3 节可用于改进算法 4.4（带有最终期限的调度安排）的时间复杂度。

回想一下，抽象数据类型包括数据对象，以及对这些对象的许可操作。在实现不交集抽象数据类型之前，需要指定所需要的对象和操作。首先从元素的一个全集  $U$  开始。例如，可以有

$$U = \{A, B, C, D, E\}$$

然后希望有一个过程 `makeset`，它由  $U$  的一个元素构建一个集合。图 C-1a 中的不交集应当由以下调用创建：

```
for (每个 x ∈ U)
    makeset(x);
```

我们需要一个类型 `set_pointer` 和一个函数 `find`，利用在  $p$  和  $q$  为 `set_pointer` 类型时有以下调用：

```
p=find('B');
q=find('C');
```

则  $p$  应当指向包含 B 的集合， $q$  应当指向包含 C 的集合。这一点在图 C-1a 中说明。我们还需要一个过程 `merge`，将两个集合合并为一个。例如，如果进行以下操作：

```
p=find('B');
q=find('C');
merge(p, q);
```

则图 C-1a 中的集合应当变成图 C-1b 中的集合。给定图 C-1b 中的不交集，如果有以下调用：

```
p=find('B');
```

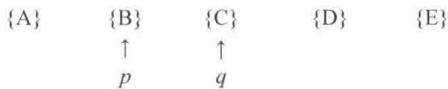
应当得到如图 C-1c 所示的结果。最后，我们需要一个例程 `equal`，用于检查两个集合是否为同一集合。例如，如果有图 C-1b 中的集合，且有调用：

```
p=find('B');
q=find('C');
r=find('A');
```

则  $\text{equal}(p, q)$  应当返回真， $\text{equal}(p, r)$  应当返回假。

我们已经指定了一个抽象数据类型，它的对象包含这些元素的一个全集和不交集中的元素，其操作为 `makeset`、`find`、`merge` 和 `equal`。

(a) 有五个不交集。已经执行了  $p=\text{find}(B)$  和  $q=\text{find}(C)$



(b) 在合并{B}和{C}之后有四个不交集



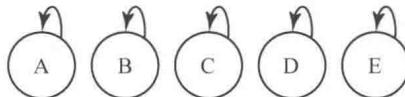
(c) 已经执行了  $p=\text{find}(B)$



图 C-1 一个不交集数据结构的例子

一种表示不交集的方法是使用带有倒置指针的树。在这些树中，每个非根节点指向其父节点，而每个根节点指向它自己。图 C-2a 给出了与图 C-1a 不交集相对应的树，图 C-2b 给出了与图 C-1b 中不交集相对应的树。为了尽可能简单地实现这些树，假定我们的全集仅包含索引（整数）。要将这一实现扩展到另一个有限全集，只需要索引该全集中的元素。可以使用数组  $U$  来实现这些树，其中  $U$  的每个索引表示全集中的一个索引。如果一个索引  $i$  表示一个非根节点，则  $U[i]$  的值就是表示其父节点的索引。如果一个索引  $i$  表示一个根节点，则  $U[i]$  的值是  $i$ 。例如，如果全集中有 10 个索引，则将它们存储到一个索引范围为 1 到 10 的索引数组  $U$  中。

(a) 倒置树表示的五个不交集



(b) [B]和[C]合并后的倒置树

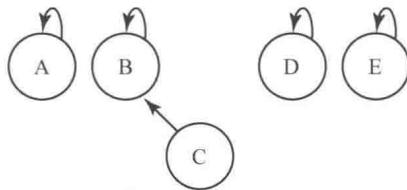


图 C-2 一个不交集数据结构的倒置树表示

为将所有索引初始放在不交集中，对于所有  $i$  值，设定

$U[i]=i;$

图 C-3a 中给出了 10 个不交集的树表示及相应的数组实现。图 C-3b 中给出了一个合并的例子。在合并集合 {4} 和 {10} 时，使包含 10 的顶点变成 4 所在节点的子节点。设定  $U[10]=4$  即可做到这一点。一般情况下，在合并两个集合时，首先确定哪个树的根节点中存储有较大索引，随后使它的根节点成为另一个树的根节点的子节点。图 C-3c 给出了在进行几次合并之后的树表示和相应的数组实现。这时只有三个不交集。下面给出这些例程的一个实现。为简化符号，在此处及 Kruskal 算法的讨论中（见 4.1.2 节），我们没有将全集  $U$  列为这些例程的参数。

### 不交集数据结构 I

```

const int n=全集中的元素个数;

typedef int index;
typedef index set_pointer;
typedef index universe[1..n]; // 全集的索引范围为 1 到 n

universe U;
  
```

```

void makeset(index i)
{
    U[i]=i;
}

set_pointer find(index i)
{
    index j;

    j=i;
    while (U[j]!=j)
        j=U[j];
    return j;
}

void merge(set_pointer p, set_pointer q)
{
    if (p<q)           // p 指向合并后的集合;
        U[q]=p;          // q 不再指向一个集合。
    else
        U[p]=q;          // q 指向合并后的集合;
    }                     // p 不再指向一个集合

bool equal (set_pointer p, set_pointer q)
{
    if (p==q)
        return true;
    else
        return false;
}

void initial (int n)
{
    index i;

    for (i=1; i<=n; i++)
        makeset(i);
}

```

调用 `find(i)` 时返回的值，是  $i$  所在树的根节点中存储的索引。我们已经包含了一个用于初始化  $n$  个不交集的例程 `initial`，因为在使用不交集的算法中经常会用到这样一个例程。

在许多使用不交集的算法中，会先初始化  $n$  个不交集，然后将一个循环执行  $m$  次（ $n$  和  $m$  的值不一定相等）。在循环内部，存在对例程 `equal`、`find`、`merge` 的常数个调用。在分析算法时，需要知道该初始化过程及循环关于  $n$  和  $m$  的时间复杂度。显然，例程 `initial` 的时间复杂度属于  $\Theta(n)$ 。

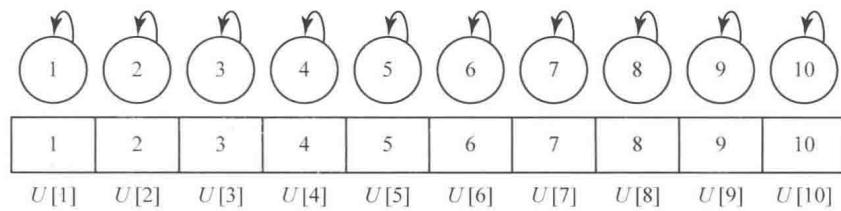
因为阶数不受相乘常数的影响，所以可以假定，在对该循环的  $m$  遍重复执行中，每一遍中仅将例程 `equal`、`find` 和 `merge` 各调用一次。显然，`equal` 和 `merge` 的运行时间均为常数。只有函数 `find` 中包含一个循环。因此，所有调用的时间复杂度的阶数主要由函数 `find` 决定。现在来计算在最差情况下，`find` 中执行的比较次数。例如，假定  $m=5$ 。在执行以下合并序列时会发生最差情况：

```

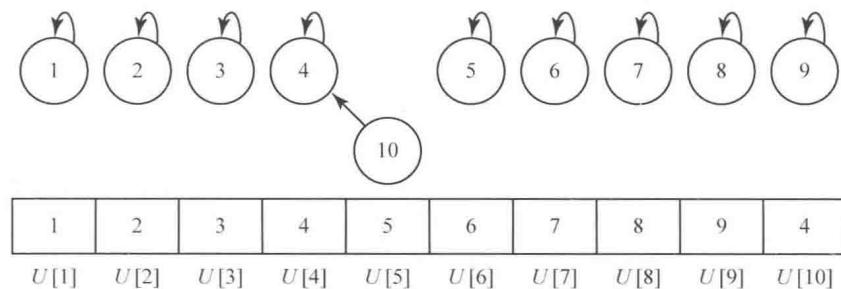
merge({5}, {6});
merge({4}, {5, 6});
merge({3}, {4, 5, 6});
merge({2}, {3, 4, 5, 6});
merge({1}, {2, 3, 4, 5, 6});

```

(a) 10个不交集的倒置树和数组表示



(b) (a)部分中的集合{4}和{10}已经被合并。第10个数组位置现在的值为4



(c) 几次合并之后的倒置树和数组表示。合并的顺序决定了树的结构

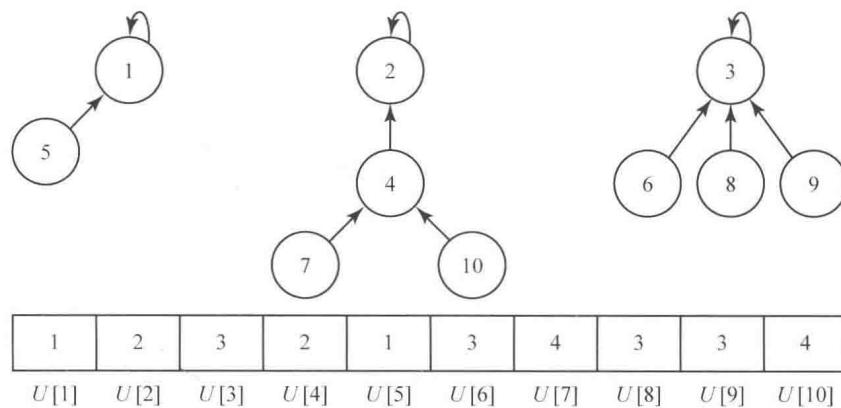


图 C-3 一个不交集数据结构倒置树表示的数组表示

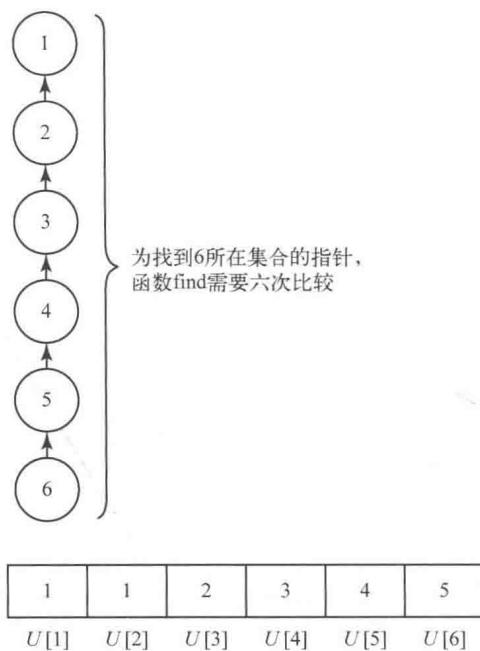
在每次 merge 之后调用 find, 查找索引 6。(实际集合写为 merge 的输入, 以作说明。) 最终树和数组实现在图 C-4 中给出。find 中完成的总比较次数为:

$$2+3+4+5+6$$

将这一结果推广到任意  $m$ , 得到最差情况下的比较次数等于:

$$2+3+\cdots+(m+1) \in \Theta(m^2)$$

我们没有考虑函数 equal, 因为该函数对于函数 find 中完成的比较次数没有影响。

图 C-4 当  $m=5$  时，不交集数据结构最差情况的一个例子

如果我们的合并顺序得到一棵树，其深度比树中的节点数小 1，则发生最差情况。如果修改进程 `merge`，使这种情况不会发生，那应当能够提高效率。为此，可以跟踪每棵树的深度，在合并时，总让深度较小的树成为子节点。图 C-5 将旧的合并方式与这一新方式进行了对比。注意，新合并方式得到的树较浅。为实现这一方法，需要在每个根节点处存储该树的深度。下面的实现进行了这一存储。

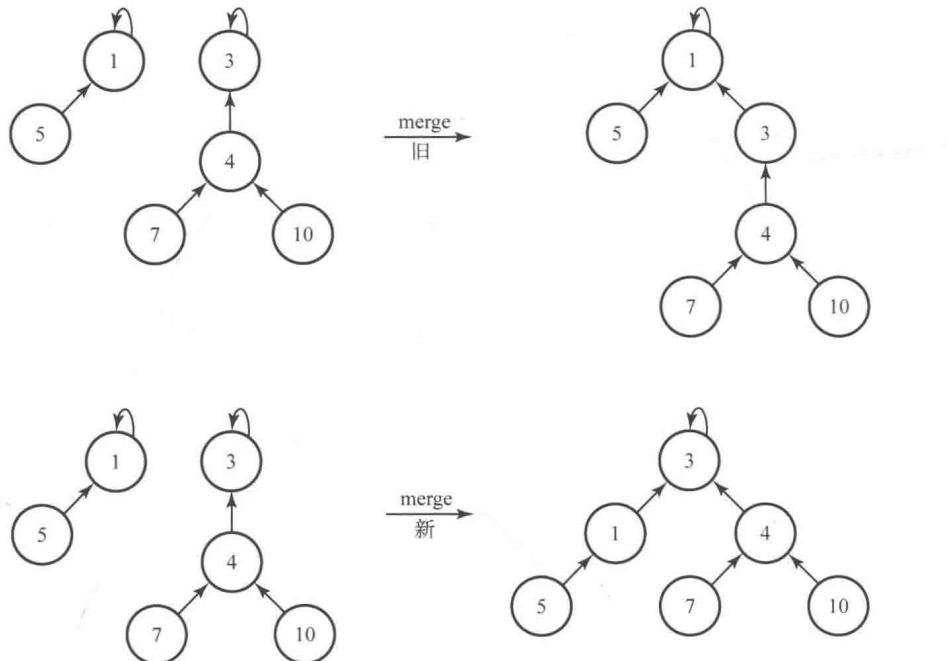


图 C-5 在新的合并方式中，将深度较浅的树的根节点，作为另一个树的根节点的子节点

## 不交集数组结构 II

```
const int n=全集中的元素数;
```

```

typedef int index;
typedef index set_pointer;

struct nodetype
{
    index parent;
    int depth;
}
typedef nodetype universe[1..n];           // 索引范围为 1 到 n 的全集。

universe U;

void makeset (index i);
{
    U[i].parent = i;
    U[i].depth = 0;
}

set_pointer find(index i)
{
    index j;

    j=i;
    while (U[i].parent != j)
        j=U[j].parent;
    return j;
}

void merge(set_pointer p, set_pointer q)
{
    if (U[p].depth==U[q].depth{
        U[p].depth = U[p].depth + 1;           // 树的深度必须增加。
        U[q].parent = p;
    }
    else if (U[p].depth < U[q].depth)       // 使深度较浅的树成为子树。
        U[p].parent = q;
    else
        U[q].parent = p;
}
bool equal (set_pointer p, set_pointer q)
{
    if (p==q)
        return true;
    else
        return false;
}

void initial (int n)
{
    index i;

    for (i=1; i<=n; i++)
        makeset(i);
}

```

可以证明，对于一个对例程 equal、find、merge 进行常数次调用的循环，将其执行  $m$  遍所执行的最差比较次数属于  $\Theta(m \lg m)$ 。

在某些应用中，需要高效地找出一个集合中的最小成员。利用第一个实现，这一点是很容易做到的，因为最小成员总是在树的根节点处。但在我们的第二个实现中，就不一定是这样了。我们可以轻松修改该实现，在

每个树的根节点处存储变量 `smallest`，使其高效地返回最小成员。这个变量包含了树中的最小索引。下面的实现就是这样做的。

### 不交集数据结构 III

```

const int n=全集中的元素数;

typedef int index;
typedef index set_pointer;

struct nodetype
{
    index parent;
    int depth;
    int smallest;
}
typedef nodetype universe[1..n];           // 索引范围为 1 到 n 的全集。

universe U;

void makeset (index i);
{
    U[i].parent = i;
    U[i].depth = 0;                         // 唯一的索引 i 是最小的。
    U[i].smallest = i;
}

void merge(set_pointer p, set_pointer q)
{
    if (U[p].depth==U[q].depth{
        U[p].depth = U[p].depth + 1;          // 树的深度必须增加。
        U[q].parent = p;
        if (U[q].smallest < U[p].smallest)   // q 的树包含 smallest 索引。
            U[p].smallest = U[q].smallest;
    }
    else if (U[p].depth < U[q].depth){
        U[p].parent = q;                     // 使深度较浅的树成为子树。
        if (U[p].smallest < U[q].smallest)
            U[q].smallest = U[p].smallest;
    }
    else{
        U[q].parent = p;
        if (U[q].smallest < U[p].smallest)   // q 的树中包含最小索引。
            U[p].smallest = U[q].smallest;
    }
}

int small(set_pointer p)
{
    return U[p].smallest;
}

```

这里仅给出了与“不交集数据结构 II”中不一样的例程。函数 `small` 返回一个集合中的最小成员。因为函数 `small` 的运算时间为常数，所以对于一个对例程 `equal`、`find`、`merge` 进行常次数调用的循环，将其执行  $m$  遍所执行的最差比较次数与“不交集数据结构 II”中相同。即，属于

$$\Theta(m \lg m)$$

利用一种称为路径压缩（path compression）的方法，有可能开发出一种实现，使其在将一个循环执行  $m$  遍时，最差比较次数几乎与  $m$  成线性关系。这一实现在 Brassard 和 Bratley（1988 年）的文献中进行了讨论。

# 参考文献

- Adel'son-Vel'skii, G. M., and E. M. Landis. 1962. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR* 146:263-266.
- Agrawal, A., N. Kayal, and N. Saxena. 2002. Not yet published. Available at <http://www.cse.iitk.ac.in/news/primality.html>.
- Akl, S. 1985. *Parallel sorting*. Orlando, Fl.: Academic Press.
- Aldous, D., and P. Diaconis. 1986. Shuffling cards and stopping times. *The American Mathematical Monthly* 93:333-347.
- Apostol, T. M. 1997. *Introduction to analytic number theory*. New York: Springer-Verlag.
- Baker, R. C., and G. Harman. 1996. The Brun-Titchmarsh Theorem on average. Proceedings of a conference in honor of Heini Halberstam 1:39-103.
- Banzhaf, W. P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming An Introduction*, Morgan Kaufmann, 1998.
- Bayer, R., and C. McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica* 1, no. 3:173-189.
- Bedell, J., I. Korf and M. Yandell. 2003. *BLAST*, O'Reilly & Associates, Inc.
- Bentley, J. L., D. Haken, and J. B. Saxe. 1980. A general method for solving divide-and-conquer recurrences. *SIGACT News* 12, no. 3:36-44.
- Blum, M., R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. 1973. Time bounds for selection. *Journal of Computer and System Sciences* 7, no. 4:448-461.
- Borodin, A. B., and J. I. Munro. 1975. *The computational complexity of algebraic and numeric problems*. New York: American Elsevier.
- Brassard, G. 1985. Crusade for better notation. *SIGACT News* 17, no. 1: 60-64.
- Brassard, G., and P. Bratley. 1988. *Algorithmics: Theory and practice*. Englewood Cliffs, N.J.: Prentice Hall.
- Brassard, G., S. Monet, and D. Zuffellato. 1986. L'arithmétique des très grands entiers. *TSI: Technique et Science Informatiques* 5, no. 2:89-102.
- Chacian, L. G. 1979. A polynomial algorithm for linear programming. *Doklady Adad. Nauk U.S.S.R.* 224, no. 5: 1093-1096.
- Clemen, R. T. 1991. *Making hard decisions*. Boston: PWS-Kent.
- Cook, S. A. 1971. The complexity of theorem proving procedures. *Proceedings of 3rd annual ACM symposium on the theory of computing*, 151-158. New York: ACM.
- Cooper, G. F. 1984. "NESTOR": A computer-based medical diagnostic that integrates causal and probabilistic knowledge. *Technical Report HPP-84-48*, Stanford, Cal.: Stanford University.

- Coppersmith, D., and S. Winograd. 1987. Matrix multiplication via arithmetic progressions. *Proceedings of 19th annual ACM symposium on the theory of computing*, 1-6. New York: ACM.
- DePuy, G. W., R. J. Moraga, and G. E. Whitehouse, "Meta-RaPS: A Simple and Effective Approach for Solving the Traveling Salesman Problem," *Transportation Research Part E*, Vol. 41, No. 2, 2005.
- Dijkstra, E. W. 1959, A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269-271.
- . 1976. *A discipline of programming*. Englewood Cliffs, N.J.: Prentice-Hall.
- Farnsworth, G. V., J. A. Kelly, A. S. Othling, and R. J. Pryor, "Successful Technical Trading Agents Using Genetic Programming," Technical Report # SAND2004-4774, Sandia National Laboratories, Albuquerque, NM, 2004.
- Fine, T. L. 1973. *Theories of probability*. New York: Academic Press.
- Fischer, M. J., and M. O. Rabin. 1974. "Super-exponential complexity of Presburger Arithmetic." In *Complexity of computation*, R. M. Karp, ed., 27-41. Providence, R.I.: American Mathematical Society.
- Floyd, R. W. 1962. Algorithm 97: Shortest path. *Communications of the ACM* 5, no. 6:345.
- Fogel, D. B., "Evolutionary Programming in Perspective: The Top-Down View," in Zurada, J. M., R. J. Marks II, and C. J. Robinson (Eds.): *Computational Intelligence: Imitating Life*, IEEE Press, 1994.
- Fredman, M. L., and R. E. Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization problems. *Journal of the ACM* 34, no. 3: 596-615.
- Fussenegger, F., and H. Gabow. 1976. Using comparison trees to derive lower bounds for selection problems. *Proceedings of 17th annual IEEE symposium on the foundations of computer science*, 178-182. Long Beach, Cal.: IEEE Computer Society.
- Gardner-Stephen, P. and G. Knowles. 2004. "DASH: Localizing Dynamic Programming for Order of Magnitude Faster, Accurate Sequence Alignment." In *Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference*.
- Garey, M. R., and D. S. Johnson. 1979. *Computers and intractability*. New York: W. H. Freeman.
- Gilbert, E. N., and E. F. Moore. 1959. Variable length encodings. *Bell System Technical Journal* 38, no. 4:933-968.
- Godbole, S. 1973. On efficient computation of matrix chain products. *IEEE Transactions on Computers* C-22, no. 9:864-866.
- Graham, R. L., D. E. Knuth, and O. Patashnik. 1989. *Concrete mathematics*. Reading, Mass.: Addison-Wesley.
- Graham, R. L., and P. Hell. 1985. On the history of the minimum spanning tree problem. *Annals of the History of Computing* 7, no. 1:43-57.
- Gries, D. 1981. *The science of programming*. New York: Springer-Verlag.
- Griffiths, J. F., S. R. Wessler, R. C. Lewontin, and S. B. Carroll, *An Introduction to Genetic Analysis*, W. H. Freeman and Company, 2007.
- Grzegorczyk, A. 1953. Some classes of recursive functions. *Rosprawy Matematyczne* 4. Mathematical Institute of the Polish Academy of Sciences.
- Hardy, G. H., and E. M. Wright. 1960. *The theory of numbers*. New York: Oxford University Press.
- Hartl, D. L., and E. W. Jones, *Essential Genetics*, Jones and Bartlett, 2006.
- Hartmanis, J., and R. E. Stearns. 1965. On the computational complexity of algorithms. *Transactions of the*

- American Mathematical Society 117: 285-306.
- Hoare, C. A. R. 1962. Quicksort. *Computer Journal* 5, no. 1:10-15.
- Hopcroft, J. E., and J. D. Ullman. 1979. *Introduction to automata theory, languages, and computation*. Reading, Mass.: Addison-Wesley.
- Horowitz, E., and S. Sahni. 1974. Computing partitions with applications to the knapsack problem. *Journal of the ACM* 21:277-292.
- . 1978. *Fundamentals of computer algorithms*. Woodland Hills, Cal.: Computer Science Press.
- Hu, T. C., and M. R. Shing. 1982. Computations of matrix chain products, Part 1. *SIAM Journal on Computing* 11, no. 2:362-373.
- . 1984. Computations of matrix chain products, Part 2. *SIAM Journal on Computing* 13, no. 2:228-251.
- Huang, B. C., and M. A. Langston. 1988. Practical in-place merging. *Communications of the ACM* 31:348-352.
- Hyafil, L. 1976. Bounds for selection. *SIAM Journal on Computing* 5, no. 1: 109-114.
- Iverson, G. R., W. H. Longcor, F. Mosteller, J. P. Gilbert, and C. Youtz. 1971. Bias and runs in dice throwing and recording: A few million throws. *Psychometrika* 36:1-19.
- Jacobson, N. 1951. *Lectures in abstract algebra*. New York: D. Van Nostrand Company.
- Jarník, V. 1930. O jistém problému minimálním. *Práce Moravské Průrakové Společnosti* 6:57-63.
- Johnson, D. B. 1977. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM* 24, no. 1:1-13.
- Kennedy, J., and R. C. Eberhart, *Swarm Intelligence*, Morgan Kaufmann, 2001.
- Keynes, J. M. 1948. *A treatise on probability*. London: Macmillan. (Originally published in 1921.)
- Kingston, J. H. 1990. *Algorithms and data structures: Design, correctness, and analysis*. Reading, Mass.: Addison-Wesley.
- Knuth, D. E. 1998. *The art of computer programming, vol. II: Seminumerical algorithms*. Reading, Mass.: Addison-Wesley.
- Koza, J., *Genetic Programming*, MIT Press, 1992.
- . 1973. *The art of programming, Volume III: Sorting and searching*. Reading, Mass.: Addison-Wesley.
- . 1976. Big omicron and big omega and big theta. *SIGACT News* 8, no. 2:18-24.
- Kruse, R. L. 1994. *Data structures and program design*. Englewood Cliffs, N.J.: Prentice Hall.
- Kruskal, J. B., Jr. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society* 7, no. 1:48-50.
- Kumar, V., A. Grama, A. Gupta, and G. Karypis. 1994. *Introduction to parallel computing*. Redwood City, Cal.: Benjamin Cummings.
- Ladner, R. E. 1975. On the structure of polynomial time reducibility. *Journal of the ACM* 22:155-171.
- van Lambalgen, M. 1987. Random sequences. Ph.D. diss., University of Amsterdam.
- Lawler, E. L. 1976. *Combinatorial optimization: Networks and matroids*. New York: Holt, Rinehart and Winston.
- Leung, K. S., H. D. Jin, and Z. B. Xu, "An Expanding Self-Organizing Neural Network for the Traveling Salesman Problem," *Neurocomputing*, Vol. 62, 2004.
- Levin, L. A. 1973. Universal sorting problems. *Problemy Peredaci, Informacii* 9:115-116 (in Russian). English

- translation in *Problems of Information Transmission* 9:265-266.
- Li, W. 1997. *Molecular Evolution*, Sinauer Associates.
- von Mises, R. 1919. Grundlagen der Wahrscheinlichkeitsrechnung. *Mathematische Zeitschrift* 5:52-99.
- . 1957. *Probability, statistics, and truth*. London: George, Allen & Unwin. (Originally published in Vienna in 1928.)
- Neapolitan, R. E. 1990. *Probabilistic reasoning in expert systems*. New York: Wiley.
- . 1992. A limiting frequency approach to probability based on the weak law of large numbers. *Philosophy of Science* 59, no. 3:389-407.
- . 2003. *Learning Bayesian Networks*. Prentice Hall.
- . 2009. *Probabilistic Method for Bioinformatics*. Burlington, Mass.: Morgan Kaufmann.
- Papadimitriou, C. H. 1994. *Computational complexity*. Reading, Mass.: Addison-Wesley.
- Pearl, J. 1986. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence* 29, no. 3:241-288.
- . 1988. *Probabilistic reasoning in intelligent systems*. San Mateo, Cal.: Morgan Kaufmann.
- Pratt, V. 1975. Every prime number has a succinct certificate. *SIAM Journal on Computing* 4, no. 3:214-220.
- Prim, R. C. 1957. Shortest connection networks and some generalizations. *Bell System Technical Journal* 36:1389-1401.
- Rabin, MO. 1980. Probabilistic algorithms for primality testing. *Journal of Number Theory* 12: 128-138.
- Rechenberg, I., Evolution Strategies, in Zurada, J. M., R. J. Marks II, and C. J. Robinson (Eds.): *Computational Intelligence: Imitating Life*, IEEE Press, 1994.
- Sahni, S. 1988. *Concepts in discrete mathematics*. North Oaks, Minn.: The Camelot Publishing Company.
- Schonhage, A., M. Paterson, and N. Pippenger. 1976. Finding the median. *Journal of Computer and System Sciences* 13, no. 2:184-199.
- Strassen, V. 1969. Gaussian elimination is not optimal. *Numerische Mathematik* 13:354-356.
- Süral, H., N. E. Özdemirel, I. Önder, and M. S. Turan, "An Evolutionary Approach for the TSP and the TSP with Backhauls," in Tenne, Y., and C. K. Goh (Eds.): *Computational Intelligence in Expensive Optimization Problems*, Springer-Verlag, 2010.
- Tarjan, R. E. 1983. *Data structures and network algorithms*, Philadelphia: SIAM.
- Turing, A. 1936. On computable numbers, with an application to the Entscheidungsproblem. *Proceeding of the London Mathematical Society* 2, no. 42: 230-265.
- . 1937. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 2, no. 43: 544-546.
- Waterman, M. S. 1984. "General Methods of Sequence Comparisons." *Mathematical Biology*, 46.
- Yao, A. C. 1975. An  $O(|E|\log \log|V|)$  algorithm for finding minimum spanning trees. *Information Processing Letters* 4, no. 1:21-23.
- Yao, F. 1982. Speed-up in dynamic programming. *SIAM Journal on Algebraic and Discrete Methods* 3, no. 4:532-540.



《算法基础》自1997年出版以来深受读者喜爱，已经被翻译成多种语言出版，并成为世界上许多高校广泛采用的算法教材之一。书中对算法设计、算法的复杂度分析和计算复杂度进行了恰如其分的介绍。作者用平实的语言和简单的符号介绍了各种抽象的数学概念，既浅显易懂，又不失严谨。为了便于读者理解和记忆，作者还提供了大量的示例，并在附录中介绍了基本的数学概念。

第5版新增了一章，介绍遗传算法和遗传编程，其中提供了理论和实践两方面的应用。此外，这一版还对练习和示例进行了全面更新，并且改进了教师资源。本书可作为本科生和研究生算法课程的教材，也可供程序员及算法分析和设计人员阅读。

#### 本书特点：

- 使用C++和Java伪代码，帮助读者理解复杂算法
- 不需要微积分背景知识
- 提供了大量示例，帮助读者理解和掌握理论概念

图灵社区：[iTuring.cn](http://iTuring.cn)  
热线：(010)51095186转600

分类建议 计算机/计算机科学/算法

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)



“这本书条理清晰，讲解透彻。虽然书中的一些概念很深奥，但作者深入浅出，读起来毫无艰涩之感。如果你想挑选算法教材或者想自学算法，我强烈推荐这本书！”

——亚马逊读者评论

ISBN 978-7-115-41657-5



9 787115 416575 >

ISBN 978-7-115-41657-5

定价：99.00元

[General Information]

书名=算法基础 第5版

作者=(美)RICHARD E.NEAPOLITAN著

页数=398

SS号=13933336

DX号=

出版日期=2016.03

出版社=北京人民邮电出版社