国 外 计 算 机 科 学 教 材 系 列

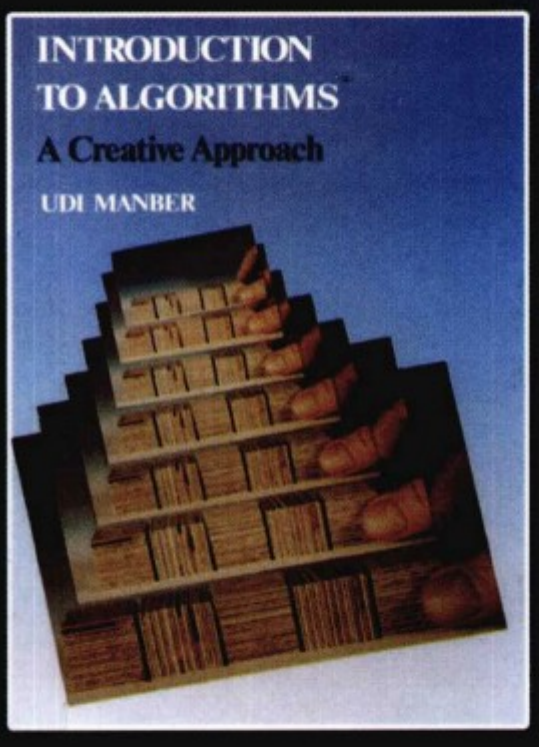
PEARSON

算法引论

——一种创造性方法

Introduction to Algorithms

A Creative Approach



[美] Udi Manber 著

黄林鹏 谢瑾奎 陆首博 等译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

国外计算机科学教材系列

算法引论

—— 一种创造性方法

Introduction to Algorithms

A Creative Approach

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内容简介

本书是国际算法大师乌迪·曼博（Udi Manber）博士撰写的一本享有盛誉的著作。全书共分12章：第1章到第4章为介绍性内容，涉及数学归纳法、算法分析、数据结构等内容；第5章提出了与归纳证明进行类比的算法设计思想；第6章到第9章分别给出了4个领域的算法，如序列和集合的算法、图算法、几何算法、代数和数值算法；第10章涉及归约，也是第11章的序幕，而后者涉及NP完全问题；第12章则介绍了并行算法；最后是部分习题的答案及参考文献。本书的特色有二，旨在提高读者的问题求解能力，使读者能够理解算法设计的过程和思想：一是强调算法设计的创造性过程，注重算法设计背后的创造性思想，而不拘泥于某个具体算法的详细讨论；二是将算法设计类比于定理归纳证明，揭示了算法设计的基本思想和本质。

本书的组织结构清晰且易于理解，强调了创造性，具有浓郁特色，时至今日仍有其巨大的价值，并且适合作为计算机及相关专业算法和高级算法课程的教材。

Authorized translation from the English language edition, entitled INTRODUCTION TO ALGORITHMS: A CREATIVE APPROACH, 9780201120370 by Udi Manber, published by Pearson Education, Inc, publishing as Addision-Wesley, Copyright © 1989 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright © 2010.

本书中文简体字版专有出版权由Pearson Education（培生教育出版集团）授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权贸易合同登记号　图字：01-2005-0327

图书在版编目（CIP）数据

算法引论——一种创造性方法 /（美）曼博（Manber, U.）著；黄林鹏等译.
北京：电子工业出版社，2010.1
（国外计算机科学教材系列）
书名原文：Introduction to Algorithms: A Creative Approach
ISBN 978-7-121-09813-0

Ⅰ.算…　Ⅱ.①曼…　②黄…　Ⅲ.电子计算机-算法理论-教材　Ⅳ.TP301.6

中国版本图书馆CIP数据核字（2009）第201517号

责任编辑：冯小贝
印　　刷：北京市海淀区四季青印刷厂
装　　订：涿州市桃园装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编：100036
开　　本：787 × 1092　1/16　　印张：21.75　字数：571千字
印　　次：2010年1月第1次印刷
定　　价：36.00元

凡所购买电子工业出版社的图书有缺损问题，请向购买书店调换；若书店售缺，请与本社发行部联系。联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

出 版 说 明

21世纪初的5至10年是我国国民经济和社会发展的重要时期，也是信息产业快速发展的关键时期。在我国加入WTO后的今天，培养一支适应国际化竞争的一流IT人才队伍是我国高等教育的重要任务之一。信息科学和技术方面人才的优劣与多寡，是我国面对国际竞争时成败的关键因素。

当前，正值我国高等教育特别是信息科学领域的教育调整、变革的重大时期，为使我国教育体制与国际化接轨，有条件的高等院校正在为某些信息学科和技术课程使用国外优秀教材和优秀原版教材，以使我国在计算机教学上尽快赶上国际先进水平。

电子工业出版社秉承多年来引进国外优秀图书的经验，翻译出版了“国外计算机科学教材系列”丛书，这套教材覆盖学科范围广、领域宽、层次多，既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求，广大师生可自由选择和自由组合使用。这些教材涉及的学科方向包括网络与通信、操作系统、计算机组织与结构、算法与数据结构、数据库与信息处理、编程语言、图形图像与多媒体、软件工程等。同时，我们也适当引进了一些优秀英文原版教材，本着翻译版本和英文原版并重的原则，对重点图书既提供英文原版又提供相应的翻译版本。

在图书选题上，我们大都选择国外著名出版公司出版的高校教材，如Pearson Education培生教育出版集团、麦格劳－希尔教育出版集团、麻省理工学院出版社、剑桥大学出版社等。撰写教材的许多作者都是蜚声世界的教授、学者，如道格拉斯·科默（Douglas E. Comer）、威廉·斯托林斯（William Stallings）、哈维·戴特尔（Harvey M. Deitel）、尤利斯·布莱克（Uyless Black）等。

为确保教材的选题质量和翻译质量，我们约请了清华大学、北京大学、北京航空航天大学、复旦大学、上海交通大学、南京大学、浙江大学、哈尔滨工业大学、华中科技大学、西安交通大学、国防科学技术大学、解放军理工大学等著名高校的教授和骨干教师参与了本系列教材的选题、翻译和审校工作。他们中既有讲授同类教材的骨干教师、博士，也有积累了几十年教学经验的老教授和博士生导师。

在该系列教材的选题、翻译和编辑加工过程中，为提高教材质量，我们做了大量细致的工作，包括对所选教材进行全面论证；选择编辑时力求达到专业对口；对排版、印制质量进行严格把关。对于英文教材中出现的错误，我们通过与作者联络和网上下载勘误表等方式，逐一进行了修订。

此外，我们还将与国外著名出版公司合作，提供一些教材的教学支持资料，希望能为授课老师提供帮助。今后，我们将继续加强与各高校教师的密切联系，为广大师生引进更多的国外优秀教材和参考书，为我国计算机科学教学体系与国际教学体系的接轨做出努力。

电子工业出版社

教材出版委员会

主　任	杨芙清	北京大学教授 中国科学院院士 北京大学信息与工程学部主任 北京大学软件工程研究所所长
委　员	王　珊	中国人民大学信息学院院长、教授
	胡道元	清华大学计算机科学与技术系教授 国际信息处理联合会通信系统中国代表
	钟玉琢	清华大学计算机科学与技术系教授 中国计算机学会多媒体专业委员会主任
	谢希仁	中国人民解放军理工大学教授 全军网络技术研究中心主任、博士生导师
	尤晋元	上海交通大学计算机科学与工程系教授 上海分布计算技术中心主任
	施伯乐	上海国际数据库研究中心主任、复旦大学教授 中国计算机学会常务理事、上海市计算机学会理事长
	邹　鹏	国防科学技术大学计算机学院教授、博士生导师 教育部计算机基础课程教学指导委员会副主任委员
	张昆藏	青岛大学信息工程学院教授

译 者 序

本书作者乌迪·曼博（Udi Manber）博士是美国著名的计算机科学家，国际公认的算法大师，在线信息搜索引擎的先驱。乌迪·曼博曾是美国亚利桑那大学计算机专业的教授，离开学校后在雅虎公司担任执行官，目前是亚马逊（Amazon.com）的副总裁和首席算法师（CAO），也是亚马逊旗下搜索网站A9.com的首席执行官，他提出的UDI测试已经成为衡量搜索引擎质量的评估标准。

本书的特色有二，一是强调算法设计的创造性过程，而不是拘泥于某个具体算法的详细讨论，正如常言道“予之以鱼，不如授之以渔”。在教学过程中常常发现有许多同学（包括译者自己）不是去掌握算法背后的思想，而是钻牛角尖，针对某个算法反复地思考，或者只是掌握了几个具体的算法就沾沾自喜。事实上学习算法不外乎两个目的，一是了解各种算法，在遇到问题时能灵活地应用所掌握的方法技巧；二个研究算法设计技术，当没有现成可用的算法时，能够创造出问题的求解方法。前者非常实际，但后者更加重要，它涉及创新，是推动学科发展的源泉。本书中的习题就很有特色，大致分为两类，一类是操练型的，意在巩固对具体算法的了解，锻炼学生应用书中介绍的算法以及各种变体的设计能力；另一类是创造型的，旨在让学生掌握设计算法的技术而非算法本身，由此锻炼学生的算法设计能力而不是算法应用能力。这种体系结构是一种尝试，虽然目前市面上有许多大部头的算法著作，包括一些算法大师的名著，但本书强调的是创造性，具有浓郁特色，因此时至今日仍有巨大的价值。

本书的特色之二是将算法设计类比于归纳证明。事实上，归纳证明、递归算法、递推关系，它们之间存在着千丝万缕的联系。对事物，人们往往是从简单到复杂，从具体到抽象地进行思考，猜测存在的性质并试图证明，最后得到一般性的规律再加以应用。问题求解也是如此，我们先从简单的情况出发，考虑简单问题的解决方案，然后再推广到一般的情况，在此过程中需要验证的是这种推广的有效性，这无疑就是一种归纳证明。作者将算法设计类比于归纳证明并贯穿全书，将各种归纳方法演绎得淋漓尽致，使通常的算法设计有章可循，更使设计算法成为一种享受。

译者黄林鹏教授与本书的翻译有着不解之缘。2003年初，译者的研究生导师左孝凌教授拟修改他的离散数学教材，准备添置论述归纳法的章节，由此译者为他收集了许多关于归纳法的材料，并粗读了其中一部分，也了解到了一些皮毛。不料2004年暑假左老师因操劳过度，竟不辞西去。译者希望能以这本与归纳法紧密相关的翻译教材的出版向左老师及其未尽事业致敬。此外就是关于并行计算中的脉动阵列算法，15年前译者在读博士的时候，恰好其博士生导师孙永强教授在做关于脉动阵列的研究，译者得以参与，并对此稍有了解。2000年译者在美国哈佛大学进修，遇到脉动阵列体系结构的创始人之一，并有幸旁听了他的课程。而本书最后就是关于脉动阵列算法的，读来译来，会有些如偶遇故知和聆听智者教诲的感觉。虽然脉动阵列是本书成书时的研究热点，现在时过境迁，已经是集群计算、分布对象计算、网格计算的时代，但书中涉及的并行算法设计的思想依然璀璨，富有启迪。事实上，正是因为强调的是算法设计的创造性过程而非算法本身，才使得本书可以永保活力。

本书的翻译工作由上海交通大学的黄林鹏教授负责，并与一些老师和博士生共同完成，其中的第2章、第4章、第6章由陆首博初译，第3章、第5章、第7章由谢谨奎初译，参与本书翻译工作的还有欧佳凡、黄青、杨欢、林海源、毛宏燕、黄晓琴、李东、刘飞、林琳、王佳锃、耿向阳等同志，陆朝俊和任庆生副教授对本书一些章节的翻译提供了大量帮助。在此，对所有为本书出版提供了帮助的人们表示诚挚的感谢！稿件由黄林鹏修改、整理和定稿，并对最终出现的问题负责，请将批评意见寄至lphuang@sjtu.edu.cn，不胜感激。

前　言

编写本书的动机来源于我在教学实践中常常无法为给定算法给出清晰解析的困惑。与许多教师一样，我发现对一些学生来说，要他们亲自动手解决一些简单问题有困难，而让他们理解给定问题的解决方案同样有困难。我相信，事物的两个方面——创造和解释——是相关而不可分离的。为了完全了解一个问题，考察最后的答案远远不够，我们必须了解问题的求解过程。

本书强调了算法设计的创造性方面，其主要目的是要告诉读者如何去设计一个新的算法。本书描述算法的顺序不是“问题 X、算法 A、算法 A'、程序 P、程序 P'”，而是像（但并不总是）“问题 X、直接明了的问题求解算法、缺点、改进这些缺点的困难、（可能包含一些错误的）好的算法、进一步的改进、分析以及其他方法和算法的关系”。本书的目标不是给出一个容易转换为程序代码的算法，而是希望读者理解算法的原理。算法因此被解释为创造过程而不是最终产品。我们讲授算法的目标不仅是说明如何求解特定的问题，还包括传授如何求解未来将产生或遇到的新问题的技术。可以说，讲授算法设计的思维过程与讲授问题求解细节是同样重要的。

为了进一步帮助读者构建算法创造的思维过程，本书在算法设计过程中使用的是一种“老兵新传”式的方法学。该方法学除了涵盖许多已知的算法设计技术外，还是一个优雅的、能深入解释算法设计的直觉的框架。然而，该方法学并不能遍及算法设计所有可能的方法，我们也不打算专用该方法学于算法设计过程。该方法学的核心在于使用归纳法进行数学定理证明的智能过程与组合算法设计过程之间的相似性。尽管这两个过程的目的不同、取得结果的类型有异，但它们的相似程度却比表面上看起来更多。此相似性也已被许多人注意。本书的新颖性在于对该相似性所进行的深度发掘，并将说明该相似性除了涵盖许多已知的算法设计技术外，还可以在相当程度上帮助算法的创建。对于该方法学，本书第 1 章先做了简单的介绍，然后在第 5 章中进行形式化讨论。

考虑下面的类比。你来到一座陌生的城市，租了一辆车，现在想知道旅馆的方向。这时如果别人喋喋不休地告诉你城市的历史、布局和交通状况，你肯定会觉得非常厌烦。你所需要的其实是诸如“向前开、过两个街区、右转、再直着向前开三英里”这样的信息。然而，如果计划在这座城市里生活较长的时间，那么你的看法就会有所改变。你可能希望在另一个方向上行驶一段时间（假定能找到一个人告诉你方向），慢慢地你将对城市有更多的了解。虽然本书也包含了许多问题的求解和解析，但它不是一本告诉你所有具体目标其方位的书籍，主要强调的是一般性的原理和方法。本书的学习具有挑战性，它需要读者的深入思考，而我相信额外的努力是值得的。

在许多不同领域，如数学、统计学、细胞生物学和工程学科，高效非数值算法的设计已变得非常重要，本书可作为算法和非数值计算的导论性书籍。许多专业人士，甚至那些未曾和计算机有过紧密接触的科学家，会认为程序设计是一种低级且无需智能的活动。这种看法将导致简单、低效、价值不高的问题解决方案，而忽视优雅、高效算法的存在。本书的目的就是要向读者说明算法设计是一门既重要又雅致的学问。

本书是自包含的，大部分内容很直观，技术性细节尽量保持到最少或者与讨论的主题内容分离，特别是我们将算法实现细节和算法思想尽可能分离，书中许多算法的设计都强调了这一原则。本书并不是以哪些内容必须掌握和牢记这样的方式来设计的，而是通过思想、例子、反例、修改、

改进等一系列过程加以阐述。在大多数算法描述内容的后面给出了算法的伪代码，每章都包含习题和相关的文献目录。大部分章节后面的习题可分为两类：操练型和创造型。操练型习题的目的在于检查读者对于书中特定例子和算法的理解，创造型习题的目的在于检查读者应用相应章节所介绍的特定算法和技术于新问题求解的能力。本书最后给出了一些精选习题（习题编号加上了下划线）的答案。

本书组织如下。第 1 章到第 4 章的内容是介绍性的。第 2 章涉及数学归纳法，就我看来，数学归纳法对于算法设计是非常重要的，熟悉归纳法对于算法设计来说有莫大的帮助。遗憾的是计算机科学系的学生对于归纳证明不是十分了解，因此对一些学生来说，第 2 章的内容可能比较困难，建议他们在第一次阅读本书的时候跳过那些有难度的例子，待适当时候再回过头来研究。第 3 章涉及算法分析，描述了算法分析的过程，并给出对本书所讨论的算法进行简单分析所需要的工具。第 4 章介绍了数据结构，那些对数据结构熟悉且具备一定数学基础的同学可以直接跳到第 5 章（但阅读一下也是有益的）学习。第 5 章提出了与归纳证明进行类比的算法设计思想，并给出一些简单的例子，描述了算法设计过程。如果你没有太多时间，只打算阅读本书的一章，则请直接阅读第 5 章。

一般来说，有关算法的书籍有两种不同的组织结构，一种是按照领域进行分类的，如分成图算法、几何算法等；另一种是按照算法设计技术进行分类的。虽然本书强调的是算法设计技术，但我选择了前者。本书第 6 章到第 9 章分别给出了下面 4 个领域的算法：序列和集合的算法（如排序、序列比较、匹配等）、图算法（如生或树、最短路径、匹配等）几何算法（如凸包和交集问题等）、代数和数值算法（如矩阵乘法、快速傅里叶变换等）。这种组织结构不但清晰且易于理解。

第 10 章涉及归约或约简。尽管归约的例子也出现在本书前面其他章节之中，但该主题不但独特而且重要，因此有必要作为一章的主题。第 10 章的内容也是第 11 章的序幕，后者涉及 NP 完全问题。复杂度理论已经成为算法的基本组成部分，任何算法设计者都应该了解 NP 完全问题以及证明相关性质的技术。第 12 章介绍了并行算法，本章包含不同并行计算模型上的几个有意义的算法。

本书内容或许不能在一个学期之内讲授完毕，因此对于教师来说就有多种不同的选择。开设的第一门算法设计课程可在某种深度上涉及第 3 章、第 5 章、第 6 章、第 7 章和第 8 章的部分内容，这些章节的高级部分以及第 9 章、第 10 章、第 11 章和 12 章的内容是可选的，它们也可作为高级算法课程的基础。

致谢

首先要感谢我的妻子 Rachel 对于本书的帮助，对本书所基于的方法学她给出了指导性的建议，对于本书的内容她也给出了许多意见和忠告，可以说没有她就没有本书。

特别感谢 Jan Van Leuwen 对本书大部分章节所给出的全面的评论。这些详细的意见、建议和修正方案为本书增色不少。感谢 Eric Bach、Darrah Chavey、Kirk Pruhs 和 Sun Wu，他们在阅读了本书的部分手稿后给出了建设性的意见，感谢 Guy T. Almes（赖斯大学）、Agnes H. Chan（美国东北大学）、Dan Gusfield（加州大学 Davis 分校）、David Harel（以色列魏兹曼学院）、Daniel Hirschberg （加州大学 Irvine 分校）、Jefferey H. Kingston（爱荷华大学）、Victor Klee（华盛顿大学）、Charles Martel（加州大学 Davis 分校）、Michael J. Quinn （新罕布什尔大学）和 Diane M. Spresser（詹姆士·麦迪逊大学）。

感谢 Addison-Wesley 出版公司的编辑没有施加给我通常作者所经历的有关书籍出版的恐怖经历，他们乐于助人，并且充满耐心。特别感谢主管 Bette Aaronson，本书的编辑 Jim DeWolf 和 Lyn Dupre，他们在写作过程中引导我，尽管知道有更好的方法，但他们仍然允许我按照自己的方式行事。在这里，我要感谢美国自然科学基金总统青年研究奖的资助以及 AT&T、数字设备公司、惠普公司和 Tektronix 公司的资助。

然而，必须强调的是，最后的手稿是由排版人员（即作者自己）准备的，他决定了这里所列举的许多人员对本书的评论和建议的取舍，并且将对最后的结果负责。

乌迪·曼博

于美国亚利桑那州图森市

目　录

第 1 章　引论……1

第 2 章　数学归纳法……6

2.1　引言……6
2.2　三个简单的例子……7
2.3　平面内区域的计数……8
2.4　简单的着色问题……10
2.5　复杂一些的加法题……10
2.6　一个简单的不等式……11
2.7　欧拉公式……12
2.8　图论中的一个问题……13
2.9　格雷码……14
2.10　在图上寻找无重边的路……16
2.11　数学平均数和几何平均数定理……17
2.12　循环不变量：将十进制数转换为二进制数……19
2.13　常见的错误……20
2.14　小结……21

第 3 章　算法分析……27

3.1　引言……27
3.2　符号 O……28
3.3　时间与空间复杂度……30
3.4　求和……31
3.5　递推关系……33
3.5.1　巧妙地猜测……34
3.5.2　分治关系……36
3.5.3　涉及全部历史的递推关系……37
3.6　一些有用的证明论据……39
3.7　小结……40

第 4 章　数据结构简介……44

4.1　引言……44
4.2　基本数据结构……44
4.2.1　元素……44
4.2.2　数组……45
4.2.3　记录……45

4.2.4 链表……46
4.3 树……47
4.3.1 树的表示……48
4.3.2 堆……49
4.3.3 二叉搜索树……51
4.3.4 AVL 树……54
4.4 散列……56
4.5 合并–查找问题……58
4.6 图……60
4.7 小结……61
第 5 章 基于归纳的算法设计……66
5.1 引言……66
5.2 多项式求值……66
5.3 最大导出子图……68
5.4 寻找一对一映射……69
5.5 社会名流问题……71
5.6 分治算法：轮廓问题……73
5.7 在二叉树中计算平衡因子……75
5.8 寻找最大连续子序列……76
5.9 增强归纳假设……77
5.10 动态规划：背包问题……77
5.11 常见的错误……80
5.12 小结……80
第 6 章 序列和集合的算法……85
6.1 引言……85
6.2 二叉搜索的几种形式……85
6.2.1 纯二叉搜索……85
6.2.2 循环序列的二叉搜索……86
6.2.3 二叉搜索特殊下标……86
6.2.4 二叉搜索长度未知的序列……88
6.2.5 重叠子序列问题……88
6.2.6 解方程……89
6.3 内插搜索……89
6.4 排序……90
6.4.1 桶排序和基数排序……90
6.4.2 插入排序和选择排序……92
6.4.3 归并排序……93
6.4.4 快速排序……93
6.4.5 堆排序……98

6.4.6 排序问题的下界……100
6.5 顺序统计……102
6.5.1 最大数和最小数……102
6.5.2 查找第 k 小的数……102
6.6 数据压缩……103
6.7 串匹配……105
6.8 序列比较……110
6.9 概率算法……112
6.9.1 随机数……113
6.9.2 着色问题……114
6.9.3 将拉斯维加斯算法变换成确定性算法……114
6.10 查找众数……116
6.11 三个展现有趣证明方法的问题……118
6.11.1 最长递增序列……118
6.11.2 查找集合中两个最大的元素……119
6.11.3 计算多重集合的模……121
6.12 小结……122
第 7 章 图算法……130
7.1 引言……130
7.2 欧拉图……131
7.3 图的遍历……133
7.3.1 深度优先搜索……133
7.3.2 广度优先搜索……139
7.4 拓扑排序……140
7.5 单源最短路径……142
7.6 最小代价生成树……146
7.7 全部最短路径……150
7.8 传递闭包……151
7.9 图的分解……153
7.9.1 双连通分支……153
7.9.2 强连通分支……159
7.9.3 利用图分解的例子……164
7.10 匹配……166
7.10.1 非常稠密图中的完美匹配……166
7.10.2 偶图匹配……167
7.11 网络流量……169
7.12 哈密尔顿旅行……173
7.12.1 反向归纳……173
7.12.2 在非常稠密图中找哈密尔顿回路……173
7.13 小结……174

第 8 章　几何算法……188
8.1　引言……188
8.2　判定点是否在多边形内部……189
8.3　构造简单多边形……191
8.4　凸包……193
8.4.1　直接方法……193
8.4.2　礼品包裹算法……194
8.4.3　Graham 扫描算法……195
8.5　最近点对……197
8.6　水平线段和竖直线段的交点……200
8.7　小结……203
第 9 章　代数和数值算法……207
9.1　引言……207
9.2　求幂运算……207
9.3　欧几里得算法……210
9.4　多项式乘法……211
9.5　矩阵乘法……212
9.5.1　Winograd 算法……212
9.5.2　Strassen 算法……213
9.5.3　布尔矩阵……215
9.6　快速傅里叶变换……218
9.7　小结……224
第 10 章　归约……228
10.1　引言……228
10.2　归约的例子……229
10.2.1　简单字符串匹配问题……229
10.2.2　特殊代表集……229
10.2.3　关于序列比较的归约……230
10.2.4　在无向图中寻找三角形……231
10.3　有关线性规划的归约……232
10.3.1　概述与定义……232
10.3.2　归约到线性规划的例子……233
10.4　下界的归约……235
10.4.1　寻找简单多边形算法复杂度的下界……235
10.4.2　关于矩阵的简单归约……236
10.5　常见的错误……237
10.6　小结……238
第 11 章　NP 完全问题……242
11.1　引言……242

11.2 多项式时间归约 ····242
11.3 非确定性和 Cook 定理 ····244
11.4 NP 完全性的证明例子 ····246
11.4.1 顶点覆盖问题 ····246
11.4.2 支配集问题 ····247
11.4.3 3SAT 问题 ····248
11.4.4 团问题 ····249
11.4.5 3 着色问题 ····250
11.4.6 一般经验 ····251
11.4.7 更多的 NP 完全问题 ····252
11.5 处理 NP 完全问题的技术 ····253
11.5.1 回溯法和分枝限界法 ····254
11.5.2 确保性能的近似算法 ····257
11.6 小结 ····261

第 12 章 并行算法 ····266

12.1 引言 ····266
12.2 并行计算模型 ····267
12.3 共享存储器算法 ····268
12.3.1 并行加 ····268
12.3.2 寻找最大数的算法 ····269
12.3.3 并行前缀问题 ····271
12.3.4 在链表中查寻秩 ····273
12.3.5 欧拉遍历技术 ····274
12.4 互连网络上的算法 ····275
12.4.1 阵列上的排序 ····276
12.4.2 排序网络 ····279
12.4.3 在树中查找第 k 个最小元素 ····280
12.4.4 网孔上的矩阵乘法 ····283
12.4.5 超立方体中的路由 ····284
12.5 脉动计算 ····286
12.5.1 矩阵与向量相乘 ····286
12.5.2 卷积问题 ····287
12.5.3 序列的比较 ····288
12.6 小结 ····290

部分习题答案 ····296

参考文献 ····317

第1章 引 论

“构筑”过程确实蕴涵巨大的重要性，有人宣称由此可以看到科学发展的充分必要条件。但我认为，必要性，是无疑的，但并不是充分的。一个成功而不浪费心智的构筑，由于要作为高层事物的基石，首先应该具备某种形式的协调性，而我们看到的，不应该是一些东西的简单堆砌，而是更多。

——亨利·庞加莱（1902）

在《韦氏大学词典（第九版）》中，算法的解释是“求解数学问题（如寻找最大公约数）的一个过程，该过程步骤有限，通常还涉及重复的操作；广义地说，算法是按部就班解决一个问题或完成某个目标的过程。本书取广义的算法来定义。算法设计是一个古老的研究领域。自古以来，人们总是对发现更好的目标求解方法充满兴趣，不论是取火、建造金字塔还是对邮件进行排序。而计算机算法的研究当然是一个新的领域。一些计算机算法采用的方法早在计算机发明之前就存在，但大多数计算机算法的设计需要新的方法和技术。首先，告诉计算机诸如“察看小山，如果发现敌情就拉响警报”是不够的。一台计算机必须了解“察看”的确切含义，知道如何识别敌情，懂得如何拉响警报（基于技术原因，拉响警报是最容易的）。一台计算机可接受的指令应当是定义明确、长度有限的基本操作序列。将通常的命令转换为计算机可以理解的指令是一个困难的过程，而该过程就是编程，目前有成百万的程序员正在不同层次上进行编程。

计算机上的编程，所需要的不仅仅是将那些为人所理解的命令转换为计算机可以理解的语言。在大多数情况下，程序员必须设计出完全崭新的算法来求解问题。只学习与计算机交谈所用的怪异语言会使编程变得困难，因为只有计算机才知道你说了什么。计算机不仅能以极快的速度执行先前由人完成的操作，它还可以做得更多。计算机能处理数十亿、数万亿比特单位的信息，能在一秒内完成数百万条基本指令。在这个数量级上进行算法设计是一种崭新的实践，有很多方面会与我们的直觉相反，因为我们通常只对自己能感知的事物进行思考。遗憾的是，一些能很好解决小问题的程序在处理大问题时就变得很糟。因此当进行大规模计算时不要忽视算法的复杂度和有效性。

问题的另一个方面是，我们在日常生活中所执行的算法一般不太复杂，执行的次数也不太频繁，因为回报非常低，通常不值得耗费太多精力来开发完美的算法。例如，在日常生活中，从超市采购回家，打开杂货袋放置食品的过程就可能不是最有效的，有效的过程应该考虑杂货袋中的食品以及橱柜的结构，但很少有人会去想这件事，更不会有人为它去专门设计算法。反之，进行大规模商业包装和拆装就必须有一种好的方法。另一个例子是修剪草坪，可以考虑如何使来回次数最少、修剪时间最短，或者到垃圾堆的往返距离最短。除非你非常讨厌修剪草坪，否则就不会花一个小时的时间来思考如何使修剪时间缩短几分钟这样的问题。但另一方面，计算机可以处理非常复杂的任务，而且同一任务往往必须执行多次，因此值得花费时间来设计有效的方法，就算最终得到的方法更加复杂或难以理解也无妨，因为潜在的回报是巨大的。（当然我们也不要太乐观，有时花费数小时的程序设计时间带来的仅仅是节约几秒钟的计算机运算时间。）

对与直觉相悖的大型问题求解算法的需要以及了解这些算法的复杂度是学习算法这门学科的动力所在。首先，必须认识到直观的方法并不总是最好的，寻找更好的方法有时非常重要，为此，程序员必须学习新的方法和技术。本书给出了多种算法的说明。然而，就像要成为一个好棋手仅仅靠记住许多棋局是不够的一样，就算你学习了大量的算法也不足以应付各种情况，因此我们必须了解算法背后的原理，必须知道如何应用它们，当然更重要的是，何时应用它们。

算法的设计和实现类似于房屋的设计和建筑[①]。我们先从最基本的概念出发考虑房子的修建。设计师的工作是给出满足要求的规划；工程师的工作是确认规划是正确且可行的（以保证房子不会在短时间内倒塌）；建设工人的工作是按照规划建筑房子。当然，在所有阶段，都要考虑造价并进行分析。每个阶段的工作是不同的，但它们又有所联系甚至交织在一起。算法的设计也如此，先考虑基本的思想和方法，然后做出规划。我们必须证明规划是可行的并且代价是可接受的。最后的工作是在具体的计算机上实现。简单地说，可将过程分为 4 步：设计、正确性证明、分析和实现。再次，每步都不同但又相关，每步都不能不顾及其他而在真空中完成。当然，我们很少能按照线性次序走过历程。在构造算法的每个阶段都可能出现困难。由此可能需要修改设计，还可能需要再次进行可行性证明、调整代价以及改变算法实现。

本书主要注重算法的第一个阶段，即算法的设计。按照类比，本书合适的名字应该是算法体系结构。然而，体系结构在计算机科学中有它自己的含义，使用这个名字容易造成混淆。当然本书也不会忽略算法的其他方面，书中在对大多数算法描述之后紧接着是算法的正确性证明、算法的复杂度分析以及对算法实现的讨论，有些详细，有些简单，但主要还是围绕算法设计展开的。

要成为一个好的体系结构师并设计出崭新的算法，仅靠学习许多算法是不够的，还必须了解算法设计背后的原理。本书中，我们使用一个与以往不同的方式来解释算法。我们认为学习如何创造事物的最好方式就是尝试去创造它，因此本书首先引导读者给出其自己的算法；其次，更重要的是，我们将遵循有助于培养读者创新思想的方法学。该方法学（由 Manber[1988]首先提出）是一种能对算法进行深入解释的直观的优雅框架。该方法学包括许多不同的方法及变体。在许多不同的方法间做出选择并且应用它们更需要系统的方法学。该方法学并没有完全覆盖算法设计的所有可能方式，但它对于本书中大多数算法的设计很有帮助。

该方法学基于数学归纳法，核心是对证明数学定理的智力过程与设计组合算法的过程进行类比。数学归纳法其原理的主要思想是命题无须从什么都没有开始证明，可以通过先证明对于一个小的基本实例集命题是正确的，然后由“若对相对较小的命题实例成立则可导出较大的命题实例成立”的方式，来证明对所有实例来说命题都是成立的。将该原理转化为算法设计则提示了一种侧重于如何将简单问题的解扩充为较大问题的解的方法。给定一个问题，如果能说明如何应用类似问题（但参数较小）的解决方法进行问题求解，那么任务就完成了。因此它的基本思想是如何扩充一个解决方案而不是从零开始进行构造。正如在本书后面章节中将看到的那样，对于这个问题有许多方法，由此产生了多种不同的算法设计技术。

数学归纳法在这里主要用于高层次算法的设计和解释，我们不对方法本身进行形式化和公理化的研究。但有很多人做出了许多贡献，包括非常杰出的 Dijkstra[1976]、Manna[1980]、Gries[1981]、Dershowitz[1983]和 Paull[1988]等。本书是对以上书籍的补充，主要用于教学。显然，对一个事物解释得更清晰，它就能被理解得更通透。在证明技术上，我们讨论了如何加强归纳假设，如何更精明地选择归纳序列，如何应用双重归纳和反向归纳等。我们的方法的重要性有两个方面：其一，

① 该比喻是受 Tracy Kidder 所著的两本书的启发，这两本书分别是“The Soul of a New Machine”（Little Brown，1981）和“House”（Houghton Mifflin，1985）。

我们收集了许多似乎无关的算法设计技术并归类于一个框架之下；其二，将已知的数学证明技术应用于算法设计。后者特别重要，因为它为已在另一学科中开发多年的强有力技术的使用打开了一扇门。

这种方法的一个缺点在于它不是一种通用的方法，并不是所有程序都可以或者应该按照归纳的思想进行设计。然而，在算法设计中归纳原理非常普遍，使得我们可以侧重对其应用的研究。对于其他原理，本书也不会忽视。对新出现的一种方法学最通常的批评是它可以为已有的东西给出一个有意义的解释，但它无助于事物的创造。这是一个实在的批评，因为只有未来才能告诉你一种方法学的有效性和广泛应用的程度。我坚信归纳法不仅仅是一种算法解释工具，它还是理解算法的必要途径。就个人来说，虽然在设计算法时我也有不遵循该方法学的经历，但碰到了两个例子，我发现它是非常有用的，它能让我快速开发出算法（Manber and McVoy [1988]、Manber and Myers[1989]）。

算法描述记法

除了通过对算法开发的创造性过程进行算法描述外，本书还包含许多算法的伪代码。包含程序代码的目的是加强算法的形象描述。我们并未对程序做太多优化，也不推荐直接使用它们。在许多情况下，我们有意不包括程序的优化版本，因为这可能会引入额外的复杂度，使读者的注意力离开算法的主要思想。有时也没有详细解释如何将算法思想转换为程序，这种转换有时很显然有时却不是。本书的侧重点，正如前面所提到的那样，在于算法设计的原理。

在大多数情况下，我们使用类 Pascal 语言（有时甚至是纯 Pascal 语言）。在许多情况下，基于可读性，Pascal 代码会包含一些高级描述。要注意的一个例外是对标识 Pascal 块的 **begin** 和 **end** 语句的修改，我们仅在程序的开始和结束时用到它们，块之间使用缩排进行标识。这种使用约定在不引起混淆的情况下将大大节省空间。在数据类型清晰的情况下我们通常不准备给出变量的详细说明和数据类型（比如说 G 是一个图，T 是一棵树等）。

习题

对于那些编号包括下划线的习题，本书后面有其解答，标记为星号的习题，则可能会比其他习题稍难。

本章中的习题无须任何与算法相关的先期知识，它们是一些有特定输入的相对简单的问题，这里要求读者手工给出问题的解答。习题的目的是说明处理大规模问题的困难。换言之，这些习题的目标之一是说明直接方法将受挫于此类问题。这里给出的问题将在后面的章节中继续讨论。

1.1　在不同的卡片上分别写下数字 1 到 100，打乱后按顺序进行排列。

1.2　在不同的卡片上分别写下下列 100 个数字并排序。考虑这道习题与上一道习题的不同点。

32918 21192 11923 4233 88231 8312 11 72 971 8234 22238 49283 3295
29347 3102 32883 20938 2930 16 823 9234 9236 29372 2218 9222 21202
83721 9238 8221 30234 93920 81102 1011 18152 2831 29133 9229 10039
9235 48395 2832 37927 73492 8402 48201 38024 2800 32155 2273 82930
2221 3841 311 3022 38099 29920 28349 74212 7011 1823 903 2991 9335
29123 28910 29281 3772 20012 70458 30572 38013 72032 28001 83835
3017 92626 73825 29263 2017 262 8362 77302 8593 3826 9374 2001
83261 48402 4845 79794 27271 39992 22836 444 2937 37201 37322
49472 11329 2253

1.3　考虑由下列数所组成的表。你的工作是删去尽可能少的数使得留下来的数以升序排列。例

如除了前两个数以外删去所有的数就会得到一个升序序列，删去除第 1 个、第 3 个、第 6 个、第 8 个数以外的其他数也将得到一个升序序列（可以比前面少删除一些数）。

9 44 32 12 7 42 34 92 35 37 41 8 20 27 83 64 61 28 39 93 29 17 13 14 55 21 66 72 23 73 99 1 2 88 77 3 65 83 84 62 5 11 74 68 76 78 67 75 69 70 22 71 24 25 26

1.4 重做习题 1.3，要求剩余的数以递减次序排列。

1.5 假定在一个陌生的国度，有 5 种不同的硬币单位 15、23、29、41 和 67（分）。寻找所有组成 18 元 8 分（即 1808 分）的可能组合。假定对于所有面值的硬币你都有足够的硬币。

1.6 给定的输入是由整数对组成的表，其中对 (x, y) 表示 x 等待 y 的应答。当 x 处于等待时，它不做任何事，特别是不能给任何等待它应答的节点发送信息。问题是要寻找一个由序对组成的序列 $(x_1\ x_2), (x_2\ x_3), \cdots, (x_{k-1}\ x_k), (x_k\ x_1)$，其中 $k > 1$（任意 k 都可以）。如果存在这样的序列，则说系统存在死锁，因为没有节点可以继续，每个节点都在等待来自其他节点的应答。

你可以用纸和笔，做任何有关数的计算（如比较、创建一个表），但不能画任何图。（也可以画图来帮助你设计解决该问题的一般方法，但所画的图不能与问题的输入有关。）

1 16, 2 21, 2 25, 2 22, 23 50, 23 47, 24 1, 25 10, 35 7, 36 45, 36 37, 38 42, 39 41, 12 37, 12 23, 12 3, 12 20, 14 25, 41 9, 42 3, 43 5, 43 22, 29 2, 30 48, 31 15, 32 17, 6 45, 6 1, 5 35, 5 20, 5 28, 5 11, 48 4, 48 10, 49 32, 7 31, 7 4, 5 33, 6 29, 6 12, 6 11, 6 3, 6 17, 45 27, 47 34, 48 20, 7 40, 7 34, 8 11, 9 19, 11 30, 11 4, 11 22, 11 25, 20 24, 21 23, 21 46, 22 47, 23 49, 3 39, 3 34, 4 14, 4 37, 5 42, 5 8, 15 2, 15 50, 15 4, 15 37, 16 13, 17 38, 18 28, 19 8, 26 15, 26 42, 27 18, 28 35, 13 36, 13 50, 13 34, 13 22, 29 34, 29 38, 29 30, 29 16, 44 33, 44 36, 44 7, 44 3, 44 32, 44 21, 33 9, 33 21, 33 35, 33 19, 33 41, 26 10, 26 44, 26 16, 26 39, 26 17

1.7 图 1.1 是一张 15×15 的二维表，对任意 i，第 i 行第 i 列表示同一位置。表中条目的数值是相应行、列所表示位置的直接距离。“—”表示两个位置间没有直接的联接。两个位置间的最短路径可能经过一个（或多个）其他位置。例如 1 和 6 之间的最短路径就经过 5 和 12。试寻找 1 和 15、4 和 3、15 和 8 之间的最短路径。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	2	3	-	1	9	6	2	1	7	4	2	8	3	-
2	7	0	2	-	-	-	-	2	1	6	9	1	7	2	8
3	8	-	0	8	9	3	6	8	5	7	-	8	-	3	-
4	-	8	-	0	-	5	4	-	-	1	1	9	-	8	-
5	9	-	8	-	0	3	2	7	5	8	-	1	-	4	2
6	3	2	-	3	6	0	5	3	2	-	8	7	2	-	8
7	2	-	-	2	8	-	0	6	2	-	8	8	2	-	4
8	1	1	-	-	2	3	8	0	-	1	1	-	2	7	-
9	4	-	9	-	2	9	-	2	0	4	9	3	-	-	-
10	-	-	-	-	1	8	-	7	1	0	3	-	-	-	2
11	3	8	7	1	-	-	3	8	-	-	0	2	9	2	1
12	3	-	1	2	8	1	1	-	5	1	9	0	2	-	9
13	7	-	3	1	6	-	-	2	-	3	-	9	0	2	-
14	2	9	6	-	7	-	9	-	3	-	1	1	9	0	-
15	2	9	2	1	-	-	1	-	4	3	6	5	1	-	0

图 1.1 与习题 1.7 和习题 1.8 相关的表

1.8　考虑图 1.1 中的表，寻找数字 5 到所有其他位置的最短路径。

1.9　考虑图 1.2。寻找一条访问每个节点恰好一次的封闭路径。（该图与十二面体的边对应，这个问题首先由爱尔兰的威廉·R·哈密尔顿爵士描述，将在本书后面的 7.12 节中讨论）。

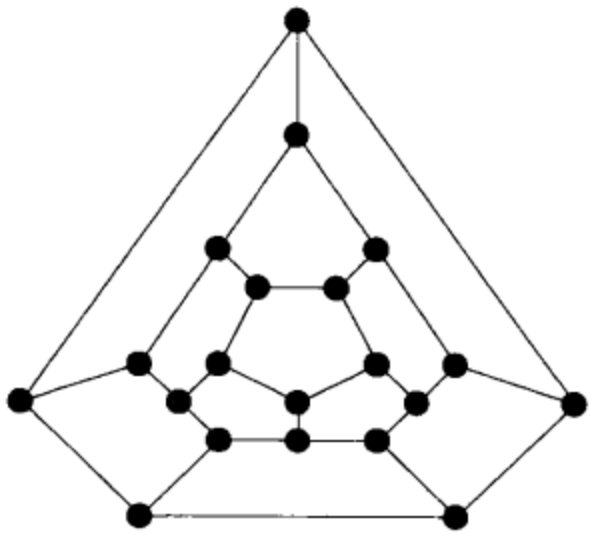


图 1.2　哈密尔顿问题

1.10　下面是一个常规的以数字（而不是图形）表示的迷宫，该迷宫是一个 11 行 11 列的正方形，每格包含 0 到 10 的数值。迷宫的行走方式是沿着行列上下左右地运动。开始点是(0, 0)，目标是(10, 10)。下面的点是不能逾越的障碍：

(3,2) (6,6) (7,0) (2,8) (5,9) (8,4) (2,4) (0,8) (1,3) (6,3) (9,3) (1,9) (3,0) (3,7)
(4,2) (7,8) (2,2) (4,5) (5,6) (10,5) (6,2) (6,10) (4,0) (7,5) (7,9) (8,1) (5,7)
(4,4) (8,7) (9,2) (10,9) (2,6)

（a）寻找一条从出发点到目标的不包含任何障碍的路径；

（b）寻找一条从出发点到目标的不包含任何障碍的最短路径。

1.11　寻找 225277 和 178794 的最大公约数（两个数的最大公约数是能整除这两个数的最大整数）。

1.12　计算 2^{64} 的值。试寻找所需乘法运算最少的计算方式。

1.13　下表是 1988 年美国总统选举中各州的票数（在一个州中得到大多数选票的候选人将获得该州的所有票数）。共计 538 票。用数学方法确定选举是否有可能以平局结束（这个问题被称为分隔问题，是本书 5.10 节中讨论的背包问题的一个特例）。

Alabama	9	Alaska	3	Arizona	7
Arkansas	6	California	47	Colorado	8
Connecticut	8	Delaware	3	Florida	21
Georgia	12	Hawaii	4	Idaho	4
Illinois	24	Indiana	12	Iowa	8
Kansas	7	Kentucky	9	Louisiana	10
Maine	4	Maryland	10	Massachusetts	13
Michigan	20	Minnesota	10	Mississippi	7
Missouri	11	Montana	4	Nebraska	5
Nevada	4	New Hampshire	4	New Jersey	16
New Mexico	5	New York	36	North Carolina	13
North Dakota	3	Ohio	23	Oklahoma	8
Oregon	7	Pennsylvania	25	Rhode Island	4
South Carolina	8	South Dakota	3	Tennessee	11
Texas	29	Utah	5	Vermont	3
Virginia	12	Washington	10	Washington, D.C.	3
West Virginia	6	Wisconsin	11	Wyoming	3

第 2 章　数学归纳法

假说，仅创立者信；实验，唯主事者疑。
——佚名

显而易见与真知灼见水火不容。
——伯特兰·罗素（1872–1970）

2.1　引言

本章将通过一些例子简要地介绍数学归纳法，这些例子有些比较简单，有些则相对较难。对那些从未用过归纳法进行命题证明的读者来说，学习本章的内容可能比较困难。构造证明的过程与设计算法的过程其实是类似的，具备使用归纳法进行命题证明的经验对算法设计来说有所裨益。本书后续章节将逐步介绍数学归纳法在算法设计中的重要作用。

数学归纳法是一种非常有用的定理证明技术，其原理如下：设 T 为欲证的定理，若 T 中含有变量 n，n 取一切自然数（即正整数），那么不必直接证明对于所有 n，T 都成立，只需要证明下面的两个条件成立：

1. 当 $n=1$ 时，T 成立；
2. 对任意 $n>1$，如果 $n-1$ 时 T 成立，那么 n 时 T 也成立。

很容易看出这两个条件是充分的。条件 1 和条件 2 可以直接推得 $n=2$ 时 T 成立。如果 $n=2$ 时 T 成立，那么条件 2 可以导出 $n=3$ 时 T 成立，依次类推。归纳法原理的本身很简单，一般来说无须证明，相反，在定义自然数时它还被用做公理。

一般来说，条件 1 容易得证，证明条件 2 也比直接证明定理要容易。这是因为假设 $n-1$ 时 T 成立（这个假设被称为归纳假设），从某种意义上说，得到归纳假设是毫不费力的；接下来我们不从头开始证明某个定理，而是关注如何通过减小 n 的值来对命题进行归约。下面就通过一个例子进行说明。

□ **定理 2.1**

对任意的自然数 x 和 n，x^n-1 能被 $x-1$ 除尽。

证明：对 n 进行归纳。当 $n=1$ 时，定理显然成立。假设 $n-1$ 时定理仍然成立，即假设对任意自然数 x，$x^{n-1}-1$ 能被 $x-1$ 除尽。接下来要证明的就是 x^n-1 能被 $x-1$ 除尽。证明思路是用 $x^{n-1}-1$ 来表达 x^n-1。根据归纳假设，$x^{n-1}-1$ 能被 $x-1$ 除尽，即 $x^n-1=x(x^{n-1}-1)+(x-1)$ 由归纳假设，等式右边的第一项能被 $x-1$ 除尽，而第二项本身就是 $x-1$。 □

于是可以如下定义归纳法原理：

如果对于带有参数 n 的命题 P，当 $n=1$ 时 P 成立；并且如果对每一个 n（$n>1$），若 $n-1$ 时 P 成立能推出 n 时 P 也成立；那么对任意自然数，P 都成立。

用 n 和 $n+1$ 代替 $n-1$ 和 n，可以得到等价的归纳法原理：

> 如果对于带有参数 n 的命题 P，当 $n=1$ 时 P 成立；并且如果对每一个 n（$n \geqslant 1$），若 n 时 P 成立能推出 $n+1$ 时 P 也成立；那么对任意自然数，P 都成立。

定理 2.1 的证明过程展示了数学归纳法的简单应用。近年来，人们又发展出了归纳法的多个变形。比如，下面就是一种名为**强归纳法**的常见变形：

> 如果对于带有参数 n 的命题 P，当 $n=1$ 时 P 成立；并且如果对每一个 n（$n>1$），若对任意小于 n 的自然数 P 成立能推出对 n 命题 P 也成立；那么对任意自然数，P 都成立。

这个变形与归纳法原理的区别在于它们的归纳假设不同，在证明对 n 命题成立时需要命题对任意小于 n 的自然数都成立。强归纳法在许多情况下都适用。

下面是数学归纳法的另一种简单变形：

> 如果对于带有参数 n 的命题 P，当 $n=1$ 和 $n=2$ 时 P 都成立；并且如果对每一个 n（$n>2$），若 $n-2$ 时 P 成立能推出 n 时 P 也成立；那么对任意自然数，P 都成立。

用这个变形，我们将沿着两条平行路线进行归纳。$n=1$ 作为归纳基础，通过归纳能得到命题在任意奇数时成立；$n=2$ 作为归纳基础，通过归纳能得到命题在任意偶数时成立。

下面来看另一种常见的变形：

> 如果对于带有参数 n 的命题 P，当 $n=1$ 时 P 成立；并且如果对每一个 n（n 大于 1 且为 2 的整数幂），若 $n/2$ 时命题 P 成立能推出对 n 命题 P 也成立；那么对任意一个 2 的整数幂的自然数，P 都成立。

这个变形就是把最初归纳法原理中的参数 n 写为 2^k，然后对参数 k（从 $k=0$ 开始）进行归纳。

归纳法还可用于各种不同的场合，证明的命题不仅可以是数的性质，也可以是结构的性质。在大多数情况下，我们可对衡量问题规模的 n 进行归纳，但有时归纳的量度并不是显而易见的（拿上一个例子来说，如果对 x 而不是对 n 进行归纳，那么证明将会复杂得多）。在没有现成的可以用来衡量问题规模的参数时，为了使用归纳法，我们也必须构造出一个，而常用的思路就是考虑如何把命题从小的结构拓展到大的结构。

2.2　三个简单的例子

考虑求前 n 个自然数之和的表达式 $S(n)=1+2+\cdots+n$，现在就来证明下面的定理。

□ 定理 2.2

> 前 n 个自然数之和为 $n(n+1)/2$。

证明：对 n 进行归纳。当 $n=1$ 时，有 $S(1)=1=1\cdot(1+1)/2$，故命题成立。假设前 n 个自然数之和 $S(n)$为 $n(n+1)/2$，接着证明前 $n+1$ 个自然数之和 $S(n+1)=(n+1)(n+2)/2$。根据 $S(n)$的定义，有 $S(n+1)=S(n)+n+1$，又由假设 $S(n)=n(n+1)/2$，可以推得 $S(n+1)=n(n+1)/2+n+1=(n+2)(n+1)/2$，命题得证。

□

下面再来看一个稍微复杂一些的例子。假设要计算和 $T(n)=8+13+18+23+\cdots+(3+5n)$。根据

上例的结果，已有 $S(n)$等于 $n^2/2+n/2$，而本例式中的每一项都是上例中对应项的 5 倍，因此不妨假设 $T(n)$也能用二次多项式来表示。设 $G(n)=c_1n^2+c_2n+c_3$。这里引入三个待定系数 c_1、c_2 和 c_3，然后通过计算开始的几项来确定这些系数。若 $n=0$，则和为 0，因此 c_3 一定为 0。对 $n=1$ 和 $n=2$，分别有如下两个等式：

（1）$1\cdot c_1+1\cdot c_2=8$

（2）$4\cdot c_1+2\cdot c_2=13+8$

将（2）式减去（1）式乘以 2 的积，有 $2c_1=5$，从而 $c_1=2.5$，$c_2=5.5$。先假设 $G(n)=2.5n^2+5.5n$ 就是所求的表达式，然后用归纳法证明 $G(n)=T(n)$。我们已经验证了归纳基础，现在假设 $G(n)=T(n)$，然后证明 $G(n+1)=T(n+1)$：

$$T(n+1)=T(n)+5(n+1)+3=(\text{根据归纳法})G(n)+5(n+1)+3$$

$$=2.5n^2+5.5n+5n+8=2.5n^2+5n+2.5+5.5n+5.5$$

$$=2.5(n+1)^2+5.5(n+1)=G(n+1)\ 。$$

下述定理得证。

□ **定理 2.3**

级数 $8+13+18+23+\cdots+(3+5n)$ 之和是 $2.5n^2+5.5n$。 □

最后我们来看另一个简单的例子。

□ **定理 2.4**

若 n 是自然数，且 $1+x>0$，则

$$(1+x)^n\geqslant 1+nx。\qquad (2.1)$$

证明：对 n 进行归纳。$n=1$ 时，式（2.1）的两边都等于 $1+x$。假设 $(1+x)^n\geqslant 1+nx$ 对所有满足 $1+x>0$ 的 x 都成立，下面考虑 n+1 时的情况。我们要证明的是对所有满足 $1+x>0$ 的 x，$(1+x)^{n+1}\geqslant 1+(n+1)x$成立：

$$(1+x)^{n+1}=(1+x)(1+x)^n\geqslant(\text{根据归纳法})(1+x)(1+nx)$$

$$=1+(n+1)x+nx^2\geqslant 1+(n+1)x。$$

注意：因为已知 $1+x>0$，所以我们可以在（由归纳法得到的那个）不等式两边同乘$(1+x)$。最后一步成立是因为 nx^2 是非负的。 □

2.3 平面内区域的计数

若平面上的直线，任意二线不平行且任意三线不共点，则称这些直线**居一般位置**（in general position）。下面计算 n 条居一般位置的直线能在平面上构成多少个区域。考虑 n 是较小数时的解答将启发我们对问题的解答进行正确的猜测。$n=1$ 时，只有 2 个区域。两条相交直线构成 4 个区域。3 条不共点的直线构成 7 个区域。因此看来，似乎第 i 条直线会增加 i 个区域，至少在 $i\leqslant 3$ 的时候如此。如果这个猜测对任意 i 成立，那么区域的个数就可以很容易地由上一节中计算过的 $S(n)$

得到。因此，现在要关注的问题就是增加了一条直线以后区域数目是如何增加的。欲证的命题如下所示：

猜测：*在平面上 $n-1$ 条居一般位置的直线上添加一条直线会增加 n 个区域。*

如前所述，当 $n \leqslant 3$ 时命题成立。我们现在就可以把这个猜测作为归纳假设，以求证在平面上 n 条居一般位置的直线上添加一条直线会增加 $n+1$ 个区域。注意：这个假设不能直接计算平面上区域的数目，而是计算添加一条直线后平面上所增加的区域的数目。即使该假设成立，我们还是需要计算平面上区域的数目，只不过计算起来比较容易。

添加一条直线会如何影响区域的数目呢？考虑一下图 2.1。既然所有的直线都居一般位置，因此一条直线不可能恰好处于区域的边界上，或者将一个区域一分为二（这样就增加了一个区域），或者与这个区域不相交。这样，我们只需要证明第$(n+1)$条直线分割了整整 $n+1$ 个已有的区域。现在固然可以直接证明这一点，但是我们想阐明数学归纳法证明的另一种技巧。暂时先把第 n 条直线移去，此时根据归纳假设，由于没有第 n 条直线，因此第$(n+1)$条直线会增加 n 个新的区域。这样，只需要证明第 n 条直线的存在使得第$(n+1)$条直线多增加了一个区域。让我们把第 n 条直线放回去。既然所有直线都居一般位置，第 n 条直线与第$(n+1)$条直线就一定在某个区域 R 中的某点 p 处相交。

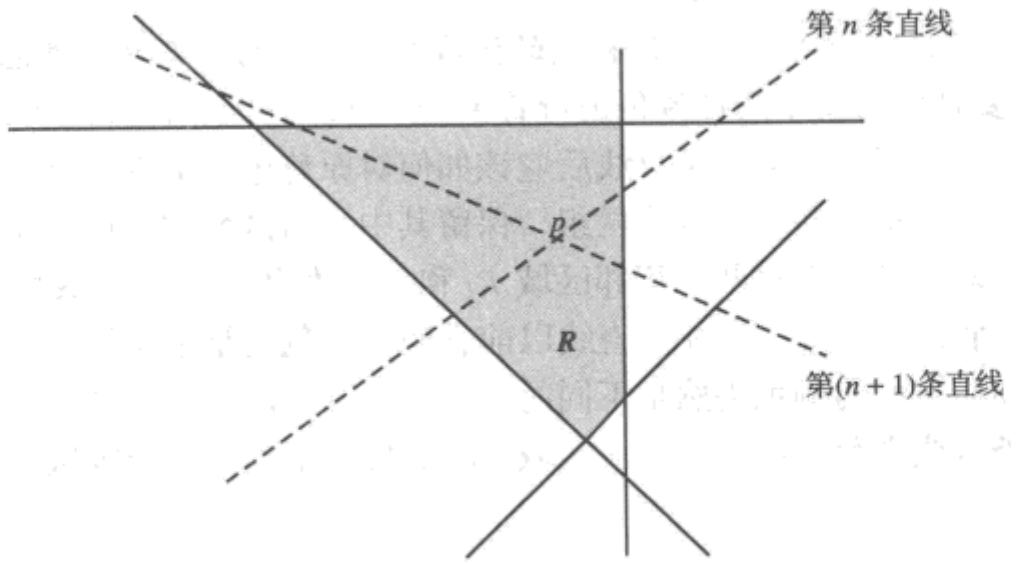


图 2.1　居一般位置的 $n+1$ 条直线

于是这两条直线都分割了区域 R。每条直线单独把 R 分成两部分，但它们也能把 R 分成 4 部分！因此，当没有第 n 条直线的时候，添上第$(n+1)$条直线能把 R 分成两部分。而有第 n 条直线的时候，添上第$(n+1)$条直线就能把 R 多分出两部分而不是只多分出一部分。而且只有 R 才与之有关，因为这两条直线只交于一点。所以在没有第 n 条直线时，第$(n+1)$条直线会增加 n 个区域，而在有第 n 条直线时，第$(n+1)$条直线就会增加 $n+1$ 个区域。证毕。

□ **定理 2.5**

平面上 n 条居一般位置的直线能把平面分割成 $n(n+1)/2+1$ 个区域。

证明：我们已经证明了第 n 条直线会增加 n 个区域。第一条直线会构成两个区域，因此区域的总数（$n>1$ 时）为 $2+2+3+4+5+\cdots+n$。在上一节中已知 $1+2+3+\cdots+n=n(n+1)/2$，所以平面能被分割成 $n(n+1)/2+1$ 个区域。　□

评述　在这段证明中有两个有趣的地方。首先，归纳假设处理的是我们所求函数的增长率，

而不是直接对函数本身进行处理。因此，归纳法证明所针对的就是函数增长率的增长，而不是定义直接证明出定理的归纳假设。可以用两步甚至多步来完成证明。第二，同样的归纳假设用在两种不同的情况下：一次用在第 n 条直线时，另一次用在第$(n+1)$条直线充当第 n 条直线时。两次运用归纳假设并不罕见，它告诉我们的是必须将假设用到极致。

2.4 简单的着色问题

回过头来再考虑一下平面上 n 条不同的直线，这次不一定是居一般位置的。我们感兴趣的是为这些直线构成的区域分配颜色，使得相邻区域有不同的颜色（两个区域被认为相邻当且仅当它们有一条公共边）。如果我们能遵循这条规则，那么就说这些区域“能被着色”，并称其为有效着色。一般来说，任何一张平面图都能用 4 种颜色来着色（其证明让数学家们冥思苦想了大约一个世纪，直到近年来才被解决）。在下面的一个定理中我们会看到由这些（无限条）直线构成的区域有一些特点。

□ **定理 2.6**

平面上任意条直线构成的区域可以仅使用两种颜色进行着色。

证明：使用自然归纳假设来证明。

归纳假设：*平面上小于 n 条直线构成的区域能仅使用两种颜色来着色。*

显然 $n=1$ 时需要且仅需要用两种颜色进行着色。如果归纳假设成立，那么考虑 n 条直线时的情况。同样，我们仅须考虑添加了第 n 条直线后应该如何对原着色方案进行修改。根据区域位于第 n 条直线的哪一侧，可以把这些区域分成两组，保留其中一组区域的颜色，反转另一组区域的颜色，现证该方案为有效着色。考察两个相邻区域 R_1 和 R_2。如果这两个区域都在第 n 条直线的同侧，那么（根据归纳假设）在添上第 n 条直线以前它们的颜色就是不同的，虽然它们有可能需要改变着色，但无论如何它们的颜色仍然是不同的。如果这两个区域的公共边是第 n 条直线的一部分，那么在添上这条直线之前它们属于同一区域。既然反转了其中一个区域的颜色，现在它们的颜色就不同。 □

评述 这个例子告诉我们的基本方法就是如何寻求灵活性，即更大的自由度。解题的思想是要尽可能地拓展并充分利用归纳假设。对本题而言，关键的思想就是在已知有效着色的前提下，反转所有区域的颜色后得到的仍为有效着色，从而解决了添加直线后产生新的区域的着色问题。

2.5 复杂一些的加法题

接下来的问题要复杂一些。考察下面的三角形：

$$
\begin{aligned}
1 &= 1\\
3+5 &= 8\\
7+9+11 &= 27\\
13+15+17+19 &= 64\\
21+23+25+27+29 &= 125
\end{aligned}
$$

这个问题要先计算第 i 行的和的表达式，然后再证明其正确性。

每一行的和似乎有一定的规律，看上去像是立方数的数列。

归纳假设：*上述三角形中第 i 行的和是 i^3。*

这个问题及其归纳假设是用图来定义的。要精确定义这个问题实属不易，更别说是解决它了。要解它的话，主要就是要能从中抽出正确的问题。因此，我们会做一些与上图一致的假设，然后据此解题（当然也可以做其他假设）。第 i 行有 i 个数，都是按序排列的奇数。让我们看一下连续两行的差，要证明第 i 行的和的确是 i^3，只须证明第$(i+1)$行和第 i 行的差是$(i+1)^3-i^3$（已证 $i\leqslant 4$ 时假设成立）。

第$(i+1)$行的头一个数和第 i 行的头一个数的差是多少呢？既然这些数都是按序排列的奇数，且第 i 行有 i 个数，那么这两个数的差就是 $2i$。同样，第$(i+1)$行和第 i 行的第二个数的差也如此。对第三个数、第四个数都是这样。一共有 i 个差，每个差都是 $2i$。在第$(i+1)$行的末尾还有一个数，它不与上一行的任何数匹配。因此，这两行之间的差就是 $2i^2$ 加上第$(i+1)$行的最后一个数。因 $(i+1)^3-i^3=3i^2+3i+1$，我们只须证明第$(i+1)$行的最后一个数的值为 $3i^2+3i+1-2i^2=i^2+3i+1$。这就是以第 i 行的和是 i^3 这个猜测导出的命题。我们把一个求和的问题规约成了求某一项的问题。最后通过归纳法来证明这最后一个命题。

嵌套的归纳假设： 第$(i+1)$行的最后一个数是 i^2+3i+1。

上述命题对 $i=1$ 时成立。现在只要通过归纳法来检验一下两项的差就行了，也就是说，我们要证明第$(i+1)$行和第 i 行的最后一个数的差等于

$$[i^2+3i+1]-[(i-1)^2+3(i-1)+1]=2i+2,$$

而我们已经得到第$(i+1)$行和第 i 行对应项的差是 $2i$。猜测成立。

评述　这段证明同样告诉我们，整个证明过程不必在一步之内完成。只要能取得进展，分步递进不失为一种好办法。这段证明还体现了“逆向而行”的方法。我们可以从最后的问题开始，通过规约逐步来简化问题，而不是从最初的简单问题开始向最后的问题进军。这也是十分常用的方法（不仅是在数学上）。

2.6　一个简单的不等式

在这一节，我们来证明以下的不等式。

□ **定理 2.7**

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{2^n}<1, \tag{2.2}$$

对所有$n\geqslant 1$[①]成立。

证明： 用归纳法证明。显然定理在 $n=1$ 时成立。假设式（2.2）在 n 时成立，下面就来看 $n+1$ 的情况。我们能从归纳假设得到的惟一信息就是前 n 项的和比 1 小。应该如何加入第$(n+1)$项呢？如果在不等式左边加上 $1/2^{n+1}$，则很可能使整个和大于 1。这里的技巧就是对不同的序列用归纳法。已知和

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{2^n}+\frac{1}{2^{n+1}},$$

我们来看最后的 n 项，根据归纳假设有

① 这个不等式一般表示无穷级数的收敛，但这里并不要求读者具有级数方面的知识，此外，这个式子完全是有穷的。

$$\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{2^n}+\frac{1}{2^{n+1}}=\frac{1}{2}\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{2^n}\right)<\frac{1}{2}。$$

现在就可以在不等式两边加上 1/2，于是得到 $n+1$ 时的式（2.2）。 □

评述 在归纳法证明中不必把最后一项看做第$(n+1)$项。有时候考察第 1 项会更容易。有些情况下可以把最后一项认为是满足特殊性质的特殊项。如果遇到这样的问题，则要灵活一些，考察尽可能多的方案。下面的例子更进一步说明了这一观点。

2.7 欧拉公式

下面要证明的是被称为欧拉公式的定理。考虑一个连通的平面图，有 V 个节点、E 条边和 F 个面。（面是一个封闭的区域。图形外部的区域也算做一个面。例如，一个正方形就有 4 个节点、4 条边和 2 个面。）图 2.2 中的图形有 11 个节点、19 条边和 10 个面。如果能通过边从图上的一个节点到另一个节点，则称这两个节点是连通的。如果图上任何两个节点都是连通的，则称图是连通的。直观地说，如果图只有一个部分，那它就是连通的。

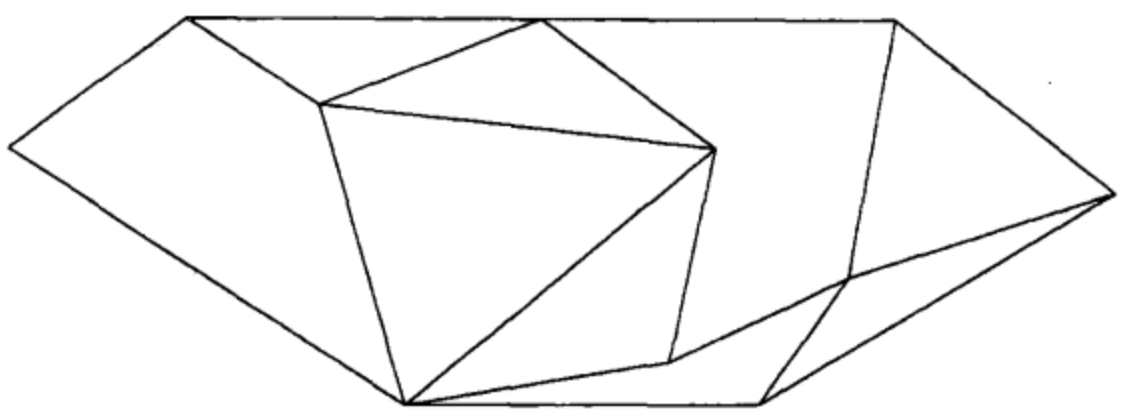


图 2.2 有 11 个节点、19 条边和 10 个面的平面图

□ 定理 2.8

任意一张连通平面图的节点数（V）、边数（E）和面数（F）的关系可由公式 $V+F=E+2$ 表示。

证明： 我们用归纳法的一个变形——双重归纳法——来证明这个定理。首先对节点数归纳，然后再对面数归纳。

首先考察只有一个面的图。这样的图不含回路；否则，回路至少构成一个面，回路以外构成另一个面。连通无回路的图被称为树。我们先证对任意树，$V+1=E+2$ 成立。

第一个归纳假设： *有 n 个节点的树有 $n-1$ 条边。*

显然，归纳基础成立。假设有 n 个节点的树有 $n-1$ 条边，则考察有 $n+1$ 个节点的树。至少存在一个节点 v，该节点只与一条边相连。否则，如果所有节点至少都与 2 条边相连，那么当我们从某个节点开始遍历这棵树时，一定会顺利到达某个曾经访问过的节点。这就说明有回路存在，得到矛盾。现在我们把节点 v 连同与之相连的一条边从树上移走，得到的图仍然是连通的，它还是一棵树，只不过少了一个节点和一条边。命题即得证。

在对面数进行归纳时将上述命题作为归纳基础。

主要的归纳假设： *有 n 个面的平面图如果有 E 条边和 V 个节点，那么 $V+n=E+2$。*

考察有 $n+1$ 个面的图。一定存在某个面 f，使得 f 与最外边的那个面相邻。因为 f 是面，故 f 一定由回路围成。既然去掉这条回路上的一条边，图仍连通，那么我们就去掉 f 与最外边的那个面之间的那条边。这样就少了一个面和一条边，定理得证。 □

评述　本定理有三个参数。证明过程中我们对其中一个参数（面数）进行归纳，而归纳基础的成立需要对另一个参数（节点数）进行归纳。这段证明说明选择归纳顺序的时候要谨慎。有时归纳过程从一个参数转到另一个参数，有时归纳与几个参数同时有关，有时需要同时对两个不同的参数进行归纳。选择不同的顺序进行归纳，证明的难度也大相径庭。在后续章节中，选择不同的顺序进行归纳对算法的效率有着很大的影响。

2.8　图论中的一个问题

首先介绍图论中的几个基本概念（在第 7 章将会详细讨论）。图 $G=(V,E)$是由节点的集合 V 和边的集合 E 构成的序对，每条边对应着一对不同的节点。图分为有向图和无向图。有向图的边是有序偶，即边的两个节点的顺序很重要。这样我们就可以把边画成从一个节点（箭羽）到另一个节点（箭矢）的箭头。无向图的边是无序偶。本节研究的是有向图。节点 v 的度数指的是与节点 v 关联的边数。路径指的是点 $v_1, v_2, \cdots, v_k$ 的序列，它们由边$(v_1, v_2), (v_2, v_3), \cdots, (v_{k-1}, v_k)$相连（这些边一般也被当做是路径的一部分）。如果从节点 u 到 v 有一条路，则称 v 从 u 可达。令 $G=(V, E)$是一个图，U 是节点的集合且 $U \subseteq V$，则把子图 $H=(U, F)$称为由 U 导出的子图，其中 F 是由 E 中所有两个节点都属于 U 的边构成的。图 $G=(V, E)$的独立集 S 是一些节点的集合，其中任意两点都不相邻。

□ **定理 2.9**

令 $G=(V, E)$是一个有向图。G 中存在一个独立集 $S(G)$，使得 G 中的每一个节点都可以从 $S(G)$的某一个节点通过一条长度不超过 2 的路达到。

证明：通过对节点数进行归纳来证明。

归纳假设：对任何节点数小于 n 的有向图，定理成立。

很显然，$n \leqslant 3$ 时定理成立。令 v 是 V 中的任意一个节点，令 $N(v)=\{v\} \cup \{w \in V \mid (v, w) \in E\}$。可见，$N(v)$是 v 相邻节点的集合。由节点集 $V-N(v)$导出的图 H 的节点数比 G 的少，这样，我们就可以对 H 使用归纳假设。令 $S(H)$是归纳假设中 H 的独立集。下面就两种情况分别进行讨论：

1. $S(H) \cup \{v\}$ 是独立的。在这种情况下，可以令 $S(G)$为 $S(H) \cup \{v\}$，因为 $N(v)$中的每一个节点都可以从 v 通过长度为 1 的路达到。由归纳假设可得，不在 $N(v)$中的节点可以从 $S(H)$的某个节点通过长度不超过 2 的路达到。
2. $S(H) \cup \{v\}$ 不是独立的。在这种情况下，一定存在某个节点 $w \in S(H)$，该节点与 v 相邻。这样，$w \in S(H)$ 推得 $w \in V-N(v)$，于是(v, w)就不是 G 的边。然而，由于假设 w 与 v 相邻，所以(w, v)一定是 G 的边。这样，$N(v)$中的每一个节点都可以从 w 经过 v 通过长度不超过 2 的路达到。令 $S(G)$为 $S(H) \cup \{w\}$。命题得证。

□

评述　证明中“归约”的量并不是固定的。也就是说，我们可以根据实际问题将问题的规模

从 n 缩减到比 n 小的数目，而且这个规模小一些的问题并不是任意一个问题，它与原来那个特定的问题关系非常密切。为使证明容易一些，我们拿掉了足够多的节点。在此取得了一个很好的平衡：既没有拿掉太多的节点，这样会使假设太弱；也没有拿掉太少的节点，这样会使假设太强。在许多情况下，找到这个平衡点是证明的关键。注意：这里用了强归纳法，因为我们需要假设对小于某个数的所有数，定理都成立。

2.9 格雷码

已知 n 个物体，需要对它们命名。每个名字都被表示为唯一的位串。一个“好的”命名方案可能有各种不同的目的，本例中，我们只考虑一个目的。用循环队列来排列这些名字，使得每个名字都可以通过只改变其前一个名字中的 1 位（bit）而生成。这样的命名方案被称为格雷码[①]。格雷码在不少场合都适用。例如，某个传感器扫描一些物体，最好能在从一个物体扫描到另一个物体时迅速改变它们的表示。本节就是要探究对任何数量的物体能否构造其格雷码表示。就这个问题而言，物体本身并不重要，我们只关心用以表达它们的数字。

我们可以用图来形象地表示名字之间的关系，名字对应着图上的节点。如果两个名字中只有 1 位不同，那么这两个节点就连上一条边。将格雷码用图来表示就是包含所有节点的回路。

从较小的 n 开始。$n=1$ 和 $n=2$ 的情况是显然的，$n=3$ 的情况又如何呢？不难看出不可能有码字长度为 3 的格雷码。如果从任意一个位串开始，每次改变 1 位，改变两次后得到的位串不是与原来的一样就是跟原来有 2 位的不同，因此改变三次是不能得到原来的位串的。事实上，这就意味着不可能构造码字长度为奇数的格雷码。$n=4$ 时，码字长度为 4 的格雷码如下：00、01、11、10，相应的图是一个正方形。下面我们就可以开始第一个证明。

□ **定理 2.10**

对任意正整数 k，存在码字长度为 $2k$ 的格雷码。

证明： 对 k 进行归纳。$k=1$ 时显然成立。假设有长度为 $2k$ 的格雷码，考虑 $2(k+1)$ 的情况。令 $s_1, s_2, \cdots, s_{2k}$ 是长度为 $2k$ 的格雷码，很明显，如果在任意位串前加上 0 或者 1，得到的仍然是格雷码（见图 2.3）：$0s_1, 1s_1, 1s_2, 0s_2, 0s_3, 0s_4, \cdots, 0s_{2k}$。

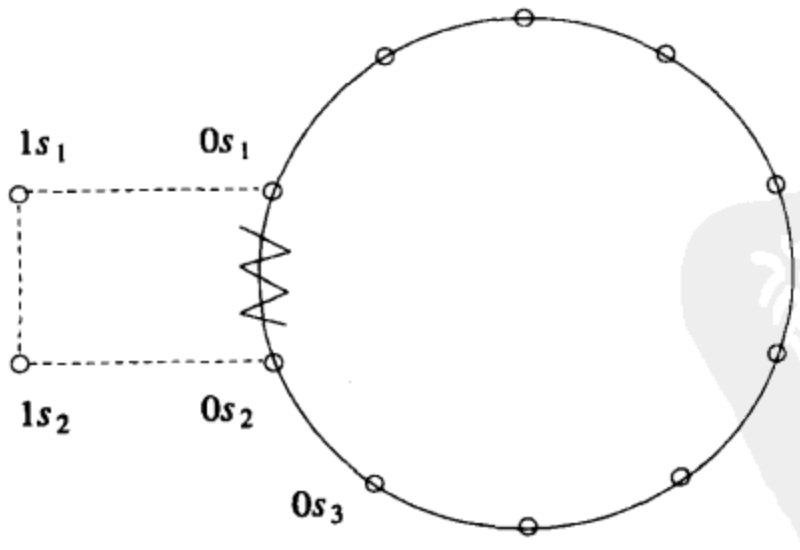


图 2.3 构造长度为 $2k$ 的格雷码

□

① 格雷码一般用来命名数量为 2 的幂的物体。这里对任意 n 都用格雷码。

尽管证明是完整的，但构造的过程并不太令人满意。码字中每一个位串的码字长度至少是物体数量的一半。总的来说，我们可以用⌈lb$n$⌉位来表示 n 个物体。那么，能用少于 $n/2$ 位来构造长度为 n 的格雷码吗？要构造数量为对数个的位，就必须在物体数量增加一倍时相应地增加 1 位。假设我们知道对所有偶数 $2k$ 如何构造格雷码，使得 $k<n$，那么已知 $2n$ 个物体时，我们就从两个长度比 n 小的码字出发，构造所需的码字。

我们马上就遇到了问题。尽管 $2n$ 是偶数，但有可能 n 是奇数，而没有奇数个数目的格雷码。这样，我们就有可能不能用 n 是奇数的归纳假设。现在只考虑这样的 n，n 是 2 的幂。假设我们知道怎样对小于 n 的任何一个 2 的幂构造短的格雷码（后面将会看到短的格雷码的长度），下面来考虑 n 的情况。令 $s_1, s_2, \cdots, s_{n/2}$ 是长度为 $n/2$ 的格雷码，在其前面加上 0 或 1，使得 $0s_1, 0s_2, \cdots, 0s_{n/2}$ 和 $1s_1, 1s_2, \cdots, 1s_{n/2}$ 也成为格雷码。用下述方法把这两个序列合并为一个（见图 2.4）：

$$1s_2, 0s_2, 0s_3, \ldots, 0s_{n/2}, 0s_1, 1s_1, 1s_{n/2}, 1s_{n/2-1}, \ldots, 1s_2。$$

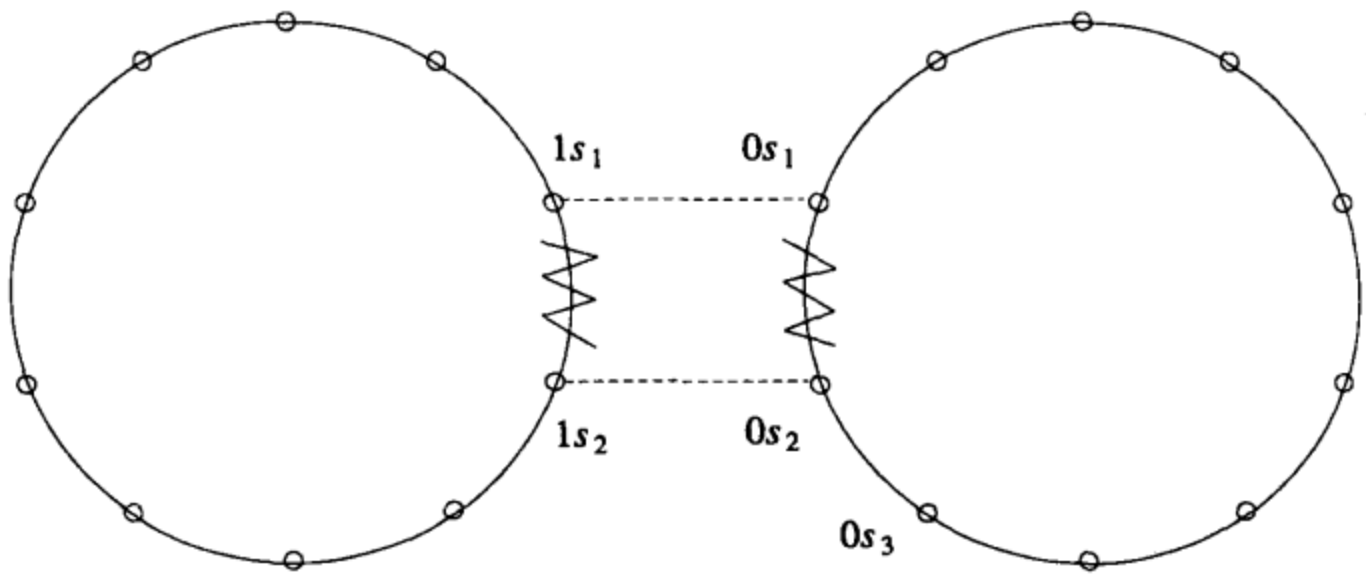


图 2.4　用两个短的格雷码构造长的格雷码

如下所示，可以把 $n=4$ 的格雷码扩展为 $n=8$ 的格雷码。原来的两个序列为 000，001，011，010 和 100，101，111，110。合并以后的序列为 101，001，011，010，000，100，110，111。我们仅使用比长度为 $n/2$ 的格雷码多 1 位就构造了长度为 n 的格雷码。因此，每一个位串的码字长度为 lb$n$。

应该怎样把这种构造方法拓展到任意偶数 n 上呢？回忆一下，构造长度为奇数的格雷码是不可能的，因为无法构成回路。回过头来看一下图 2.4，发现其实不必有两个闭合回路，只要有两个不闭合的序列就可以。如果我们能构造一个长度为奇数的“开的”（换句话说，就是只有两个名字它们之间相差超过 1 位）格雷码，那么就能对一般的情况进行证明的构造。现在分两种情况：

□ **定理 2.11**

对任意正整数 k 存在码字长度为⌈lb$k$⌉的格雷码。长度为偶数的格雷码是闭的，长度为奇数的格雷码是开的。

证明：用一个强一些的归纳假设来证明上述两种情况。

归纳假设：*对任意 $k<n$，存在码字长度为⌈lb$n$⌉的格雷码。长度为偶数的格雷码是闭的，长度为奇数的格雷码是开的。*

归纳基础显然成立。下面分两种情况来构造长度为 n 的格雷码。

1. n 是偶数。在这种情况下的规约与 n 是 2 的幂的情况下进行规约相类似。由归纳假设，存在码字长度为 $n/2$ 的格雷码（可能闭，也可能开）。我们把它复制成两份，一份前面加 0，另一份前面加 1，然后把它们连成一个回路（如图 2.4 所示）。由归纳假设还可以得到最初那个短的格雷码有$\lceil \mathrm{lb}(n/2) \rceil$位。我们可以加 1 位，让物体的数量翻一倍，这样新的格雷码就有$\lceil \mathrm{lb}(n/2) \rceil + 1 = \lceil \mathrm{lb}n \rceil$位。
2. n 是奇数。令 $n = 2k+1$。构造两个长度为 k 的格雷码，然后按先前的方式把它们连起来。如果 $2k$ 不是 2 的幂，那么一定存在码字长度为$\lceil \mathrm{lb}(2k) \rceil$的位串没有被用做名字，这其中的一个位串连接到了已经用过的某个位串。我们把这个位串加到长度为 $2k$ 的闭合回路中，就能得到一个长度为 $2k+1$ 的开回路（见图 2.5）。位数也能满足条件。如果 $2k$ 是 2 的幂，那么就没有剩下的位串了，就需要在码字中加上 1 位，于是位的总长是$\lceil \mathrm{lb}(2k) \rceil + 1$。又因 $2k$ 是 2 的幂，则有$\lceil \mathrm{lb}(2k) \rceil = \mathrm{lb}(2k)$，故 $\mathrm{lb}(2k)+1 = \lceil \mathrm{lb}(2k+1) \rceil$。

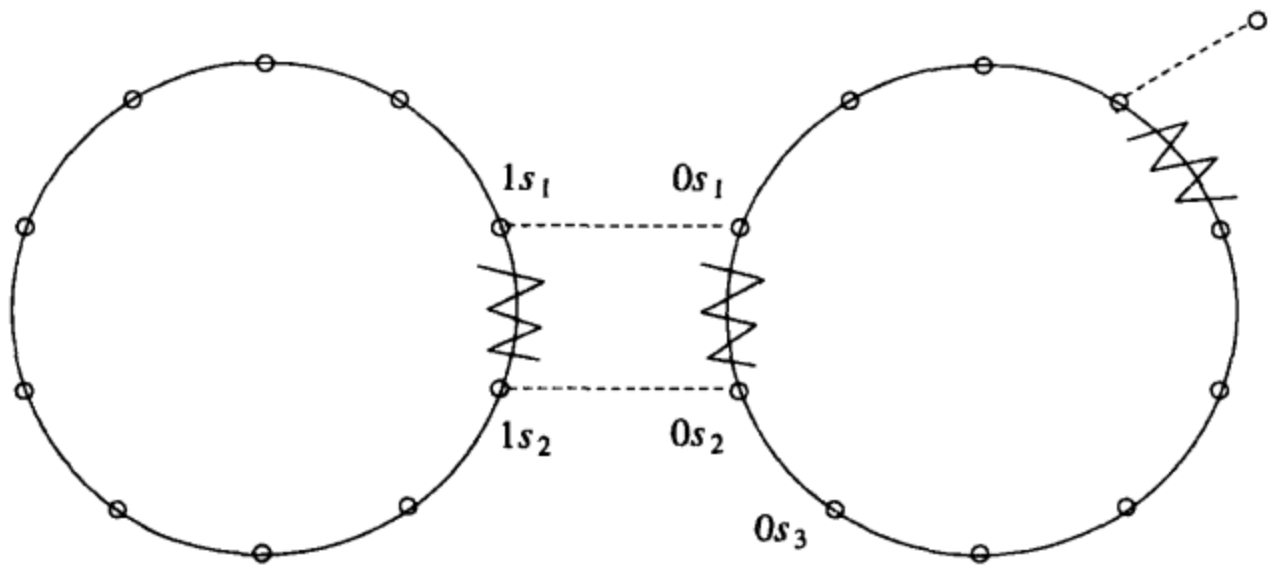


图 2.5 构造“开的”格雷码 □

评述 在本例中，定理分成两种不同情况，显然，要对它们分别进行处理。然而这样做并不总是最好的。尽管这两种情况不同，但同时考虑它们并把它们放在一个归纳假设中会使证明更容易一些。这样的话，证明一种情况时就能从另一种情况的归纳假设中得益。就像用两只脚爬山那样，我们不能分开考虑两只脚的步子，每只脚都离不开另一只脚。有时不妨对一个更一般的问题定义其归纳假设。本例中，这个更一般的问题仅分两种情况。在下一节中，我们的例子将通过扩展原有问题来处理更一般的结构，使得解决原来的问题变得更容易。这样做的好处就是能让归纳假设更强，能被更有效地利用。显然，这是一种权衡。通过假设归纳命题在 n 时正确来证明它在 $n+1$ 时也正确。如果 n 时命题更强，那么在证明过程中用起来也更有力；反过来，需要我们来证明的就更多。在下一节和 5.10 节中，我们将进一步讨论这个问题。注意，在归纳假设中假设的是所有小于 $2n$ 的值，而不仅是 $2n-2$。

2.10 在图上寻找无重边的路

令 $G = (V, E)$是连通的无向图。如果 G 中的两条路不包含相同的边，那么就称这两条路是无重边的。令 O 是 V 中度数为奇数的节点的集合。首先我们断言 O 中有偶数个节点。要证明上述断言，可以把所有节点的度数加起来，这样得到的值恰为边数的两倍（因为每条边都算了两遍）。然而，由于所有度数为奇数的节点都在总和里加了一个奇数，所以一定有偶数个度数为奇数的节点。现在我们来证明下面的定理。

□ **定理 2.12**

令 $G=(V, E)$是连通的无向图，O 是 V 中度数为奇数的节点的集合，我们可以把 O 中的节点分成节点对，对每一对结点都能找到连接它们的与其他路径无重边的路径。

证明：对边数进行归纳。$m=1$ 时定理显然成立。

归纳假设：该定理对所有边数小于 m 的连通的无向图成立。

考察有 m 条边的连通的无向图 G，令 O 是度数为奇数的节点的集合。若 O 是空集，则定理成立；否则，在 O 中任取两点，因 G 是连通的，故存在一条连接这两点的路。从图中去掉这条路，剩余的图的边要比原来的少。下面就用归纳假设在剩下的度数为奇数的节点中找到无重边的路来完成证明。然而问题在于去掉了这条路之后，原来的图有可能就不连通了，而归纳假设仅对连通图成立。所以要谨慎地使用归纳假设。我们可以根据证明的需要改变归纳假设，从而巧妙地避开这个问题。

我们现在碰到的问题就是连通性是否必要。不妨去掉连通性这一限制，这样就得到如下的归纳假设：

修改过的归纳假设：定理对所有边数小于 m 的无向图成立。

显然，这个定理要比原来的那个强。但从另一方面来看，证明却变得简单了。再次考察先前那个有 m 条边的无向图，O 的定义亦如前。图可能不再连通，而是被分成几个连通的部分。我们从同一部分中取出两个度数为奇数的节点。既然每个部分自身都是连通的，那么其中一定有偶数个度数为奇数的节点。因此，如果同一部分中的节点是偶数个，那么就一定能在其中取到两个节点。好，现在已经证得差不多了。由于我们取的两个节点是在同一部分中的，所以这两点间一定可用一条路来连接。下面把这条路从图上去掉，所得到的图的边数一定小于 m，这时就可以用归纳假设，因为已经没有连通性这一限制了。这样，在剩余的图上，就能把节点分成节点对，在每一对中都能找到连接这两个节点的无重边的路。最后，把我们刚才去掉的两个节点间的那条路加上就可以。证毕。

事实上，现在证明的定理要比原来想要证明的那个强，而且我们还证明了连通性是不必要的，证明过程也因此简单了不少。 □

评述　通过加强归纳假设来证明定理，是一种十分有效的方法。在某种程度上，它与上一节的证明方法有异曲同工之妙。主要的技巧就是要根据需要来改变归纳假设，即使定理因此变强了，证明却有可能变得简单。Polya 把这个原则称为“发明者悖论”（inventor paradox）（Polya[1954]）。之所以能够解决这个明显的矛盾，是因为尽管需要证明的东西变多了，然而由于归纳基础变强，因而证明的基础比以前更好。在本书中，我们还会看到更多的例子用了归纳基础加强的方法，该方法非常重要。

2.11　数学平均数和几何平均数定理

下面这个非常漂亮的例子展示的是柯西对数学平均数和几何平均数定理的证明，该证明过程巧妙地使用了数学归纳法的非标准形式。

□ **定理 2.13**

如果 $x_1, x_2, \cdots, x_n$ 都是正整数，则有

$$(x_1 x_2 \cdots x_n)^{\frac{1}{n}} \leqslant \frac{x_1 + x_2 + \cdots + x_n}{n}。 \tag{2.3}$$

证明：对 n 进行归纳。归纳假设同式（2.3）。这个证明的有趣之处在于归纳是倒过来实施的。我们并不是先证明基础的情况，然后再把归纳假设从比较小的 n 扩展到比较大的 n，而是用下面的逆向归纳法原理：

如果命题 P 对某个自然数的无限子集成立，且 P 对 n 成立能推出其对 $n-1$ 成立，那么 P 对任意自然数都成立。

上述原则之所以成立是因为，命题对无限集成立就说明对任意自然数 k，总能在集合中找到比它大的数 m，然后就可以用逆向归纳法从 m 逆向归纳到 k。

我们分两步来证明这个定理。第 1 步，用常规的数学归纳法证明，当 n 是 2 的幂时，定理成立，而 2 的幂构成的集合恰好就是我们所需要的无限集。第 2 步，用逆向归纳法来证明定理对任意 n 都成立。首先考察等于 2 的幂的所有 n，定理对 $n=1$ 显然成立。考察 $n=2$，此时命题化为

$$\sqrt{x_1 x_2} \leqslant \frac{x_1 + x_2}{2},$$

两边平方，可知其成立。现在假设式（2.3）对 $n=2^k$ 成立，考察 $2n=2^{k+1}$ 的情况。把式（2.3）的左边写成

$$(x_1 x_2 \cdots x_{2n})^{\frac{1}{2n}} = \sqrt{(x_1 x_2 \cdots x_n)^{\frac{1}{n}} (x_{n+1} x_{n+2} \cdots x_{2n})^{\frac{1}{n}}}。 \tag{2.4}$$

令 $y_1 = (x_1 x_2 \cdots x_n)^{1/n}$，$y_2 = (x_{n+1} x_{n+2} \cdots x_{2n})^{1/n}$，即可用 $n=2$ 时的定理。于是式（2.4）化为

$$(x_1 x_2 \cdots x_{2n})^{\frac{1}{2n}} = \sqrt{y_1 y_2} \leqslant \frac{y_1 + y_2}{2}。$$

而根据对 n 的归纳假设，我们有

$$\frac{y_1 + y_2}{2} \leqslant \frac{\dfrac{x_1 + x_2 + \cdots + x_n}{n} + \dfrac{x_{n+1} + x_{n+2} + \cdots + x_{2n}}{n}}{2},$$

命题立即得证。

接着就用逆向归纳法证明定理对任意 n 都成立。假设式（2.3）对任意 n 都成立，考察 $n-1$。定义

$$z = \frac{x_1 + x_2 + \cdots + x_{n-1}}{n-1}。$$

由于我们假设定理对任意 n 都成立，那么当然对 $x_1, x_2, \cdots, x_{n-1}$，$z$ 也成立。即

$$(x_1 x_2 \cdots x_{n-1} z)^{\frac{1}{n}} \leqslant \frac{x_1 + x_2 + \cdots + x_{n-1} + z}{n} = \frac{(n-1)z + z}{n} = z。$$

（选择 z 来简化上述表达式的右边）因此，我们有

$$(x_1x_2\cdots x_{n-1}z)^{\frac{1}{n}} \leqslant z\ ,$$

得到

$$x_1x_2\cdots x_{n-1}z \leqslant z^n,$$

和

$$(x_1x_2\cdots x_{n-1})^{\frac{1}{n-1}} \leqslant z = \frac{x_1+x_2+\cdots+x_{n-1}}{n-1},$$

这恰好就是 n–1 时的式(2.3)。 □

2.12　循环不变量：将十进制数转换为二进制数

数学归纳法在证明算法的正确性方面是很有用的。考察一个带有循环的程序，循环用来计算某个数值。要证明的是执行循环后的结果就是我们想要得到的结果。对循环执行的次数进行归纳。归纳假设应该反映出循环执行过程中变量之间的关系。这样的归纳假设被称为循环不变量。下面用图 2.6 中的算法来说明如何使用循环不变量，该算法会把十进制数 n 转换为用数组 b（数组的初始值为 0）表示的二进制数。

算法 *Convert_to_Binary* 中有一个循环，该循环包含三条语句。第一条语句自增 k，k 是数组 b 的下标。第二条语句计算 t *mod* 2，也就是 t 除以 2 所得的余数（换句话说，当 t 为奇数时，t *mod* 2 等于 1；否则等于 0）。第三条语句计算 t 除以 2，这里除法指的是整数除法（即忽略小数部分）。

□ 定理 2.14

当算法 *Convert_to_Binary* 终止时，n 的二进制表示存放在数组 b 中。

```
算法 Convert_to_Binary(n)
输入：n（正整数）
输出：b（位组成的数组，存放 n 的二进制表示）
begin
    t := n ;{用新变量 t 保存 n 的值}
    k := 0 ;
    while t > 0 do
        k := k + 1 ;
        b[k] := t mod 2 ;
        t := t div 2 ;
end
```

图 2.6　算法 *Convert_to_Binary*

证明：对循环执行的次数 k 进行归纳。归纳假设不一定要与命题的陈述相一致，完全可以只针对算法的一部分。在本例中，循环是主要部分，因此我们就用归纳假设来验证循环执行的规律。这里，归纳假设可以被看做是不变量，它只与正确的变量有关，而与循环执行的次数无关。证明中最难的部分就是要找到合适的归纳假设。考察下面的归纳假设：

归纳假设：如果 m 是由二进制数组 $b[1..k]$表示的整数，那么 $n = t\cdot 2^k+m$。

表达式 $t\cdot 2^k+m$ 不仅是循环不变量的核心，同时也是算法的关键。归纳假设说的是，这个表达

式的值与循环执行的次数无关。它抓住了算法背后的思想。在第 k 次循环中，二进制数组表示的是 n 的低 k 位，而 t 移了 k 位以后的值就对应剩余的位。

要证明算法是正确的，就需要证明以下三点成立：（1）在循环开始时假设成立；（2）若假设对第 k 次循环成立，则对第$(k+1)$次循环也成立；（3）假设在循环终止时能推得算法正确。在循环开始时，$k=0$，$m=0$（由定义，数组为空），$n=t$。假设在第 k 次循环开始有 $n=t\cdot 2^k+m$，考察在第 k 次循环结束时相应变量的值。分两种情况讨论。首先，不妨设在第 k 次循环开始时 t 是偶数，这样 $t \bmod 2$ 为 0，对数组没有影响（换句话说，m 不变），t 除以了 2，k 自增。因此，归纳假设仍然成立。其次，假设 t 是奇数，那么 $b[k+1]$被赋予 1，即 m 加上了 2^k，t 的值为$(t-1)/2$，k 自增。因此，在第 k 次循环结束时，相应的表达式为$(t-1)/2\cdot 2^{k+1}+m+2^k=(t-1)\cdot 2^k+m+2^k=t\cdot 2^k+m=n$，这正是我们要证明的。最后，当 $t=0$ 时循环终止，由归纳假设得 $n=0\cdot 2^k+m=m$。 □

2.13 常见的错误

最后，为了给大家一些警示，这里举一些由于草率使用数学归纳法而导致错误的例子。许多错误的证明正是因过分相信而起。如果你对一个定理深信不疑，往往就会把一些看上去可以从该定理推出的“事实”用于证明中。在证明过程中，经常会像下面一样。因为定理是“显然”的，有时我们会在归纳假设中不知不觉地加上几个显然的“事实”，而在从 n 到 $n+1$ 的证明中使用了这些假设。这样一来，归纳假设变强了，而我们要证明的定理却没有相应地变强。举例来说，我们很可能会忽视，定理中的图只被假设成连通的，在证明过程中却忘了在图的边减少后检验图的连通性。像这样的忽略可能很微妙，但一定会导致错误的证明。因此，准确地定义归纳假设是十分重要的。

下面是另一个常见错误。数学归纳法证明中主要的一步就是要证明定理对 n 成立能推得定理对 $n+1$ 也成立。可以先给出 $n+1$ 时的情况，然后证明其可以从 n 时的情况得到；也可以先给出 n 时的情况，然后证明其能推出 $n+1$ 时的情况。这两种方法都可以。然而，$n+1$ 时的情况必须是任意的！如果我们从 n 时的情况推到 $n+1$ 时的情况，而 $n+1$ 时是有一些特殊性质的情况，那么证明就错了。比如说，考察下面有关定理 2.8 的错误证明。从任意一个有 n 个面的图开始，由归纳法假设 $V+n=E+2$。任取一个面，在图上添加一条分割这个面的边，这条边的两个节点是新加上的。而新加上这两个节点意味着有两条原来的边被一分为二。这样一共增加了一个面、三条边和两个节点。由 $V+2+n+1=E+3+2$，故定理在 $n+1$ 个面时也成立。这个证明并不正确，因为添加的这条边是一条特殊边。我们也可以在两个已有的节点，或是一个已有节点和一个新节点之间添一条边。事实上，通过在两个新的节点之间添加一条边所得的图中，节点的度数不超过 3，可见这是一种非常特殊的情况。一般来说，从任意一种情况开始通过归纳假设对定理进行证明都比用其他方法来得稳妥。

另一个危险的陷阱就是在定理中有特殊的情况存在。常见的有 $n\geqslant 3$或“n 不是小于 30 的质数”等特殊情况。数学归纳法原理需要从 $n=1$ 的假设推出 $n=2$ 的情况，从 $n=2$ 的假设推出 $n=3$ 的情况，以次类推。就算其中只有一步不成立，定理也不能成立。我们在关于这点上要举两个例子，第一个例子荒谬得令人捧腹，第二个例子会稍微严肃一些。考察下面的命题：

荒谬的命题：已知平面上的 n 条直线，任意两条线都不平行，则所有的直线一定交于一点。

显然，这个命题是错误的，但我们还是要看一下它的“证明”。一条直线时命题显然成立。

接下来我们仔细考察两条直线的情况，定理依然成立。假设命题在 n 条直线时成立，那么来看 n+1 条直线的情况。由归纳假设，前 n 条直线交于一点，而由归纳假设我们还可以得到后 n 条直线（包括第$(n+1)$条直线）也交于一点。那么前 n 条直线和后 n 条直线的交点对这 n+1 条直线来说就是一个公共点，因为任意有两个交点的两条直线是重合的。这样的话，n+1 条直线就交于一点，命题得证。

证明错在哪里呢？其实只错了一点点。不正确的那一步就在于这个证明无意中（就这个例子而言，绝对是故意的）忽略了 n 至少要大于或等于 3 时才成立。也就是说，命题在 $n=1$、$n=2$ 时成立，接下来如果在 $n=3,4,\cdots$ 时命题成立，那么在 $n+1=4,5,\cdots$ 时命题也成立。问题就出在从 $n=2$ 推到 $n=3$ 的那一步，而这个小小的特殊情况就足以让整个证明过程功亏一篑。作为读者，你可能觉得这个错误太明显了，不太可能犯这样的错误。那么下面就来看另一个没那么明显的例子。

考察下面的命题：

$$n=\sqrt{1+(n-1)\sqrt{1+n\sqrt{1+(n+1)\sqrt{1+(n+2)\cdots}}}}。\tag{2.5}$$

（该表达式是无限的。）下面用数学归纳法来证明式（2.5）。首先我们证明，对任意 n，表达式收敛，这样命题才有意义。这里省略了这一部分的证明（证明下来的确是收敛的）。若 $n=1$，式（2.5）化为 $1=\sqrt{1+0(\cdots)}$，则成立（因括号里的表达式收敛）。假设式（2.5）在 n 时成立，考虑 n+1 时的情况。如果对式（2.5）两边平方，则有

$$n^2=1+(n-1)\sqrt{1+n\sqrt{1+(n+1)\sqrt{1+(n+2)\cdots}}}。$$

整理得

$$\frac{n^2-1}{n-1}=n+1=\sqrt{1+n\sqrt{1+(n+1)\sqrt{1+(n+2)\cdots}}},$$

恰好就是 n+1 时的式（2.5）。证毕。证完了吗？证明过程中有一步出错了，错在除以$(n-1)$时并没有检验它是否为 0。而 $n-1=0$ 即 $n=1$ 时，正好就是归纳法的第一步。我们又一次看到，除了其中的某一步，归纳法对剩下的每一步都成立。这次，这一步是从 $n=1$ 推到 $n=2$，但这足以推翻整个证明。顺便提一下，这个命题是正确的，证明它却没有那么容易。

2.14　小结

数学归纳法是一项内容丰富的技术。前面我们已经见识了归纳法的多种变形，在证明中也用了这几种方法。第 1 步就是定义归纳假设，要决定对哪个参数进行归纳。在许多情况下，只有一个参数，那么选择参数就很简单；而其他情况下就相当灵活，甚至有可能为证明而特别引入一个参数。如前所示，归纳假设不总是直接对定理的命题进行假设。有时归纳要好几步，每一步都逐步接近证明的完成；有时我们会加强归纳假设，使它推出更强的定理。

每次用归纳法证明都分成两步：归纳基础和规约。归纳基础一般都比较容易，偶尔也有不太简单的，但我们往往不太在意。归纳法证明的核心就是规约，有多种规约方法，其中最常用的就是把命题从 n 规约到 $n-1$，从 n+1 规约到 n 也比较常见。强归纳法会把命题从 n 规约到比 n 小（但不一定是 $n-1$）的一个或多个命题。其他变形还有从 $2n$ 规约到 n，以及逆向归纳法，也就是命题在 n 时的情况是由 n+1 时的情况推出的，而归纳基础是个已被证明了的无限集。规约的要点在于要完整保留原命题，不能在规约后的命题中加入任何多余的假设，除非这些假设是特别包含在归

纳假设中的。

归约也可以被看做是扩展。对某个特定的参数，把命题从较小的值扩展到较大的值。我们必须保证这样的扩展覆盖了参数中所有可能的值，且扩展后的命题是定理的一般形式，没有其他多余的假设或限制。在第 5 章中，我们将会看到本章介绍的几种归纳法变形之间的类比，以及一些算法设计方面的技巧。

参考文献和补充读物

数学归纳法原理的发现归功于意大利数学家弗朗西斯·莫罗利卡斯（公元 1494 年）。在 Bussey [1917]（也可参考 Vacca[1909]）中记载了数学归纳法的历史。我们会注意到这样一件有趣的事情，在 12 世纪，人们用一个与数学归纳法非常相似的方法来解释犹太法典（由 J.Gillis 发现）。人们在解释一条法规的时候需要说明“假日前三天”，而犹太法典编写的那个时代，人们在说“假日前三天”时往往要把假日当天包括在内。所以问题就是该不该把假日当天包括在这三天内。人们解释的时候认为三天应该不包括假日当天，否则就会引起歧义。这个观点可以通过对命题进行归纳得到。把一天作为归纳基础。假设我们表达假日当天的时候是不会讲“假日前一天”的，所以“假日前一天”应该不包括假日当天，那么“假日前两天”也不应该包括假日当天，否则就和“假日前一天”指的是同一天了。这样的话，“假日前三天”就不应该包括假日当天。显然，我们通过归纳得到了命题。

2.5 节给出的加法题选自 Polya[1957]。建议读者看一下在 Lakatos[1976]中给出的精彩讨论，是关于将欧拉公式一般化到三维物体上去的。2.8 节中所举的例子选自 lovász[1979]。Gray[1953]介绍了格雷码，在 Hamming[1986]中能找到更多关于编码理论的内容。柯西证明了数学平均数和几何平均数定理（参见 Polya and Szego[1972]，也可参见 Beckenbach and Bellman[1961]）。图论的参考书目请见第 7 章。在 Gries[1981]中可以找到更多关于循环不变量的内容。Darrah Chavey 给出了证明式（2.5）的那个例子。

在 Polya 的经典著作[1954, 1957, 1981]中可以找到进一步研究数学归纳法的内容。其他例子也可以在 Sominskii[1963]、Golovina and Yaglom[1963]，以及本书中找到。

习题

2.1 证明对任意 x、$y(x\neq y)$和 n，x^n-y^n 能被 $x-y$ 整除。

2.2 把 2.2 节中的解答扩展到一般的算术和。也就是说，求 $a_1+a_2+\cdots a_n$ 的和，其中 $a_n=c_1n+c_2$，且 c_1、c_2 是常数。

2.3 求和并证明：$1\cdot 2+2\cdot 3+\cdots+n(n+1)$。

2.4 求和并证明：$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{2^n}$。

2.5 求前 n 个自然数的平方和，并证明之。

2.6 证明：$1^2-2^2+3^2-4^2\cdots+(-1)^{k-1}k^2=(-1)^{k-1}k(k+1)/2$。

2.7 已知一个有 $n+1$ 个数的集合，这些数从前 $2n$ 个自然数 $1, 2, \cdots, 2n$ 中随意选出。证明该集合中一定存在两个数，其中一个数能够整除另一个数。

2.8 令 a、b 和 n 是正整数，证明：$2^{n-1}(a^n+b^n)\geqslant(a+b)^n$。

2.9 用数学归纳法证明：一个以十进制表示的数能被 3 整除当且仅当其每一个数字之和能被 3 整除。

2.10　下面的三角形叫做帕斯卡三角形，求该三角形第 i 行数之和的表达式，并证明之。该三角形的边上都是 1，每一个数字都是上一行中位于正上方的两个数之和。

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
```

2.11　求下列三角形第 i 行数之和的表达式，并证明之。三角形中每一个数字都是上一行中位于正上方的三个数之和（若不存在相应的数，则认为是 0）。

```
                1
              1   1   1
            1   2   3   2   1
          1   3   6   7   6   3   1
        1   4   10  16  19  16  10  4   1
```

2.12　证明对任意 $n>1$，$\frac{1}{n+1}+\frac{1}{n+2}+\cdots+\frac{1}{2n}>\frac{13}{24}$。

*2.13　证明对任意 $n>1$，$1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}=\frac{k}{m}$，其中 k 是奇数，m 是偶数。

2.14　考察数列 1, 2, 3, 4, 5, 10, 20, 40, … 该数列开头是等差数列，第 5 项以后就是等比数列。证明：任意一个正整数表示成这个数可列中的不同数之和。

2.15　考察数列 1, 2, 3, 6, 12, 24, 54, 84, 114, … 该数列开头是等差数列，第 3 项以后就是等比数列，再 3 项以后又回到等差数列。对习题 2.14 的证明方法可用来证明本题吗？如果可以，则找出证明中的错误，因为比如说，81 就不能表示成这个数列中的不同数之和。习题 2.14 的证明有何微妙之处？

2.16　考察平面中 $n\geqslant 3$ 条居一般位置的直线。证明这些直线所构成的区域中至少有一个是三角形的。

2.17　考察平面中 $n\geqslant 3$ 条居一般位置的直线。证明这些直线至少构成 $n-2$ 个三角形区域。

2.18　已知平面上的 n 个点，其中任意 3 点都在单位圆中，求证这 n 个点都在这个单位圆中。

2.19　证明：平面上 n 个圆构成的区域能用两种颜色进行着色，使得任意两个相邻区域着不同的颜色。

2.20　平面上有 n 个圆，每个圆内都有一条弦（见图 2.7）。证明：这 n 个有一条弦的圆构成的区域能够用两种颜色进行着色，使得任意两个相邻区域着不同的颜色。

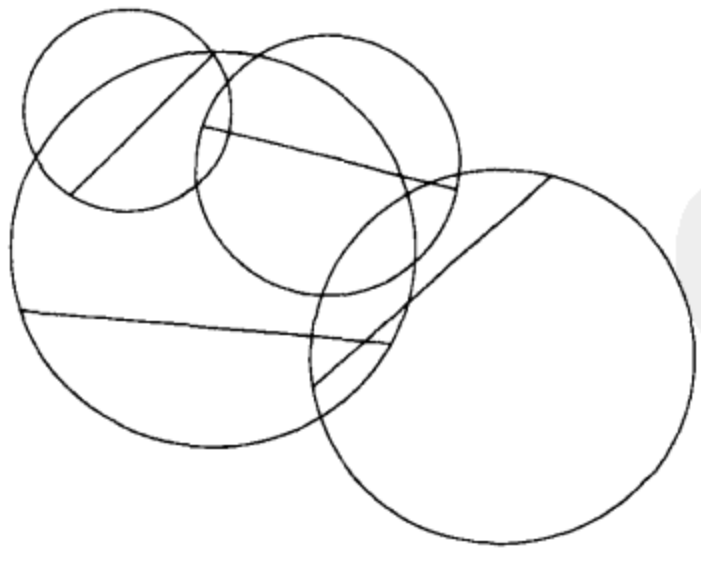


图 2.7　有一条弦的圆

2.21　证明：所有节点的度数都为偶数的平面图所构成的区域能用两种颜色进行着色，使得任意

两个相邻区域着不同的颜色。

2.22 证明：平面图能用三种颜色进行着色，使得任意两个相邻区域着不同的颜色当且仅当每个区域有偶数个相邻区域。如果两个区域有一条公共边，那么我们认为它们是相邻的。

*2.23 平面中的格点指的是具有整数坐标的那些点。令 P 是一个不与自身相交的多边形（这样的多边形被称为简单多边形），其所有节点都是格点（见图 2.8）。令 p 是多边形边界上格点的数目（包括节点），q 是多边形内部格点的数目。证明多边形的面积是 $p/2+q-1$。

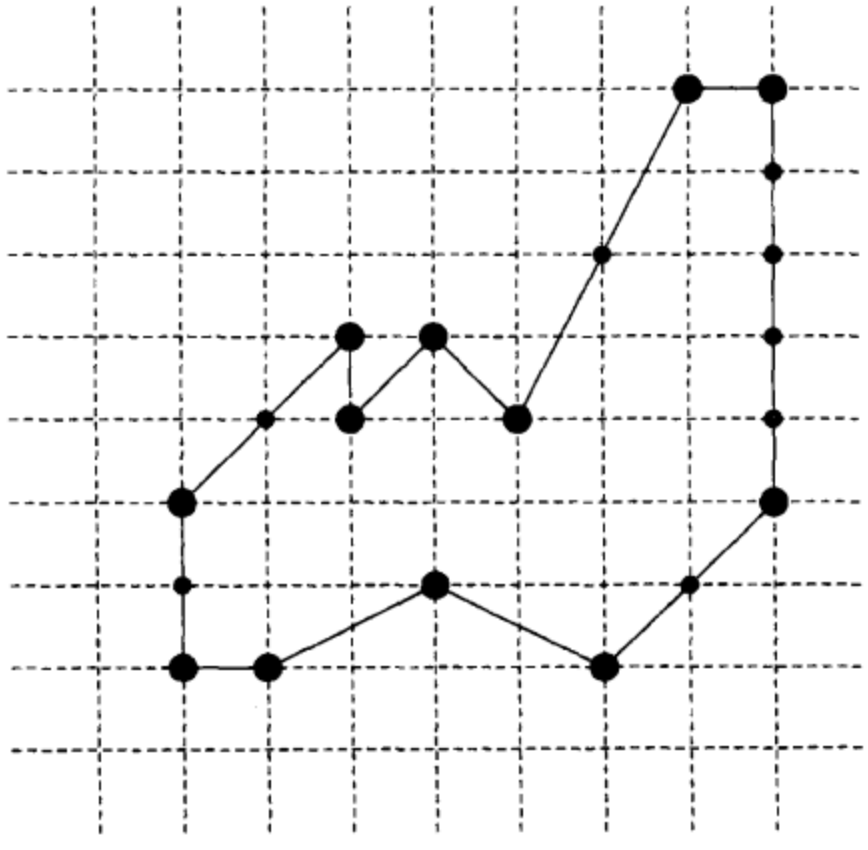


图 2.8 格点上的简单多边形

2.24 我们这样来定义反格雷码。格雷码中连续位串之间的差尽可能小，而反格雷码中的差尽可能大。是不是有可能对任意偶数个物体设计一种编码方式，使得两个连续位串之间相差 k 位（其中 k 是每个位串中的位数）呢？相差 $k-1$ 位行吗（$k-2$、$k-3$、$k-4\cdots$又如何呢）？如果可行，则给出一种高效的构造方法。

2.25 已知树 T 及其下的 k 棵子树，任何一对子树至少有一个公共节点，求证：所有子树至少有一个公共节点。

2.26 令 $d_1, d_2, \cdots, d_n, n \geqslant 2$，是正整数。证明：若 $d_1 + d_2 + \cdots + d_n = 2n - 2$，则存在一棵有 n 个节点的树，其节点的度数分别为 $d_1, d_2, \cdots, d_n$。

*2.27 圆上有 n 个点，每个点和其他所有点之间都有直线相连。假设任意三线均不共点。计算并证明这些直线把圆分割所得的区域数。

*2.28 令 $T=(V, E)$是无向树。令 f 是从节点映射到节点的函数，且满足：若(v, w)是 E 中的边，则$(f(v), f(w))$也是 E 中的边或者 $f(v)=f(w)$。换言之，函数 f 或者把边映射到边，或者把边映射到一个节点。证明在 V 中存在节点 v 使得 $f(v)=v$，或者在 E 中存在边(v, w)，使得 $f(v)=w$、$f(w)=v$（即存在映射到自身的边或节点）。

2.29 用归纳法证明抽屉原理（最简单的形式）：把 $n+1$ 个球（$n \geqslant 1$）放到 n 个盒子中，至少有一个盒子中有不止一个球。

2.30 完全二叉树递归定义如下。高度为 0 的完全二叉树只有一个节点，即根节点。高度为 $h+1$ 的

完全二叉树由两个高度为 h 的完全二叉树连接到某一个新节点组成。令 T 是高度为 h 的完全二叉树。T 中某个节点的高度是 h 减去它到根节点的距离（如根节点的高度为 h，叶子节点的高度为 0）。证明：T 中所有节点高度之和为 $2^{h+1}-h-2$。

2.31 令 $F(n)$是第 n 个斐波那契数，递归定义如下：$F(1)=F(2)=1$；对 $n>2$，$F(n)=F(n-1)+F(n-2)$。证明：$F(n)^2+F(n+1)^2=F(2n+1)$。（提示：加强归纳假设，像介绍格雷码的那一节中那样，同时证明两个看上去没什么关系的定理。）

2.32 令 n 和 m 是整数，且$1 \leqslant m \leqslant n$。用归纳法证明：

$$n^2-m(n+1)+2n+m^2 \leqslant n^2+n$$

（提示：对 m 从两头用归纳法。证明两个归纳基础：$m=1$ 和 $m=n$，然后可以从 $m=1$ 往后证，也可以从 $m=n$ 往前证。）

2.33 无向图中的割边指的是若从图上去掉这条边，图就不连通了。令 $G=(V, E)$是不含割边的连通无向图。证明：G 有如下“耳朵分解”（ear decomposition）性质（见图 2.9）。G 的边能被分成互斥的集合 $E_1, E_2, \cdots, E_k$，使得 E_1 是回路，且对任意 i（$1<i\leqslant k$），E_i 是一条路，其两端的节点在前一条路 E_j（$j<i$）中出现过，剩余的节点没有在前面任意一条 E_j 中出现过（这条路有可能是一条封闭的路，这样的话，它只含有先前的一个节点）。

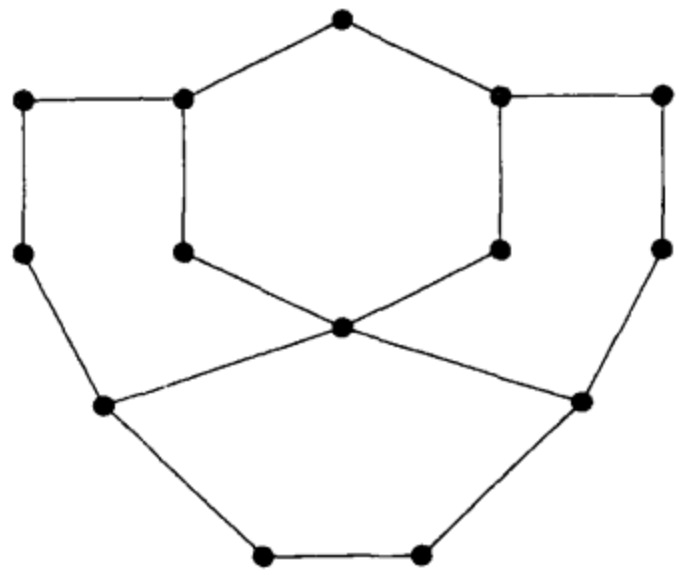


图 2.9　耳朵分解

*2.34 记 K_n 为有 n 个节点的完全无向图（即任意两个节点都有边相连）。令 n 为偶数，证明：K_n 的边可被恰好分成 $n/2$ 棵生成树。（生成树是一个包含所有节点、不含回路的连通子图。）

*2.35 已知无向图 $G=(V, E)$。匹配指的是边的集合，其中任意两条边都没有公共节点。如果每个节点都被匹配，那么称其为完美匹配。构造一个有 $2n$ 个节点和 n^2 条边的图 G，使得 G 有且只有一种完美匹配。

2.36 令 $a_1, a_2, \cdots, a_n$ 是正整数，且 $a_1a_2\cdots a_n=1$。不使用数学平均数和几何平均数的不等式，试证明

$$(1+a_1)(1+a_2)\cdots(1+a_n)\geqslant 2^n。$$

（提示：引入另一个变量代替数列中被特别选出的两个数，对这个变量进行归纳。）

2.37 考察具有递归关系的斐波那契数 $F(n)=F(n-1)+F(n-2)$。不消去递归，比较 $F(n)$和 $G(n)$，其中 $G(n)$由递归式 $G(n)=G(n-1)+G(n-2)+1$ 给出。很明显 $G(n)>F(n)$（因为 $G(n)$比 $F(n)$多 1）。而下面给出的是一个看上去正确的（归纳法）证明 $G(n)=F(n)-1$。归纳假设对所有的 k

($1 \leqslant k \leqslant n$)，$G(k) = F(k)-1$，考察 $G(n+1)$时的情况：

$$G(n+1) = G(n)+G(n-1)+1 = F(n)-1+F(n-1)-1+1 = F(n+1)-1。$$

上述证明错在哪里？

2.38 下面是对数学平均数和几何平均数的不等式的另一种证明。该证明有一个很大的缺陷，从而使得证明不太完整。描述这个缺陷，并对该定理加上一个限制条件，使这个证明变得正确。

令 $S = x_1+x_2+\cdots+x_n$。要找到矛盾之处，我们要寻找 n 个数，其和为 S，其几何平均数大于 S/n。因此就要寻找在所有和为 S 的 n 个数的集合中，其积是最大的那个。也就是要在固定 S 的前提下，求积的最大值。令$\{x_1, x_2, \cdots, x_n\}$就是那个和为 S 的集合中积最大的那个集合。若 $x_1 \neq x_2$，则用其算术平均数$(x_1+x_2)/2$ 来代替 x_1、x_2 两个数。这样就能保证和不变，而积增长了，因为

$$x_1 x_2 \leqslant \left(\frac{x_1+x_2}{2}\right)^2,$$

当 $x_1 = x_2$时，等号成立。若所有的数都相等，那么定理成立；否则就与“和为 S 的集合中积最大的那个集合”这一假设相矛盾。

2.39 设计一个算法把二进制数转换为十进制数，要求该算法是 *Convert_to_Binary* 的逆算法（见图 2.6）。算法的输入是长度为 k 的位数组 b，输出是十进制数 n。用循环不变量证明你的算法是正确的。

2.40 修改算法 *Convert_to_Binary*，使得算法把六进制数转换为二进制数。算法的输入是六进制数字的数组，输出是位数组。用循环不变量证明你的算法是正确的。

第3章　算法分析

若依体型论英雄，母牛也能逮住野兔。
——生活在宾州的德国移民的谚语

仅仅望着树上的果实，而不去测量树的高度，是个愚人。
——鲁弗斯

3.1　引言

算法分析的目的是要预测算法的行为，特别是运行时间，这应与运行算法的特定计算机无关。这样做的好处很显然，相较在具体的计算机上实现算法并对每一特定参数进行测试，该方法简单且便利。此外，一个复杂的程序通常都要包括许多不同的“小”的算法，如果对程序中每一部分的各种不同选择都进行测试，那么无疑工作量就太大了。

遗憾的是，由于存在诸多影响因素，使得预测一个算法的确切行为通常是不可能的，所以我们需要提取算法的主要特征，定义一些在算法分析中重要且必需的参数和方法，而忽略实现的细节。因此，这样的分析是近似的，并不很完美。但另一方面，即使是粗糙的分析也能得到算法的一些重要信息。最重要的是，通过这种分析，可以比较不同的算法，并针对我们的目标选择最好的那一个。这就如同在汽车的里程表上标注“仅用于比较，实际行驶路程可能不同”。

本章将描述一种用于预测算法近似运行时间以及进行算法比较的方法论。这种方法的主要特征是，忽略常数因子并关注输入大小无限增长后算法的行为。例如，输入是一个大小为 n 的数组，如果算法包含 $100n$ 个运行步骤，则忽略常数项 100 并且说运行时间约为 n（下面会引入精确的记法）。如果步骤数是 $2n^2+50$，则忽略常数 2 和 50，并且说运行时间约为 n^2。由于 n^2 比 n 大，所以可说第二个算法慢，即使是 $n=5$（此时第一个算法需要 500 步，第二个算法需要 100 步）也是如此。这是因为，如果 n 足够大，这种近似是有效的，对于所有的 $n \geqslant 50$，第二个算法确实要比第一个慢。另一方面，假设第一个算法的运行时间是 $100n^{1.8}$，则第一个算法看起来仍然要好些，因为 $n^{1.8}$ 比 n^2 小。在这种情况下，n 的值大约到 300 000 000，$100n^{1.8}$ 才比 $2n+50$ 小。幸运的是，大多数算法的运行时间表达式只含较小的常数。因此，即使这种近似方法有时会带来误导，但它在实践过程中很有效。在大多数情况下，作为对效率的初步近似和暗示，仅考察近似的行为已足够。

我们的分析结果将指出：对于特定的输入，一个问题的算法其预期运行会持续多长。然而，除非算法非常简单，否则我们是无法为所有输入列举出精确的运行时间的。可能的不同输入的数目是很庞大的，大多数算法的行为会因输入的不同而发生变化。因此，我们为输入加上一个测度，称之为输入的规模，从而把算法的分析与输入的规模联系起来。算法可能不会对相同规模的输入有着同样的行为，但我们可以期望它们之间的差异是合理的。规模通常被定义为存储输入所需要的空间的大小。试图为所有算法输入引入一个一般的规模定义并不是本书讨论的问题，我们感兴趣的是对相同问题的不同算法进行比较。在大多数情况下，规模的定义是直接的。在后面的例子中，除非特别指明，否则规模将用 n 来表示。

给定一个问题和规模的定义，我们的目标是找到一个与规模有关的算法运行时间的表达式。（关于运行时间的精确定义将在 3.3 节中给出）。前面说过，对所有规模相同的输入并不存在一个通用的值。所以，必须在所有规模相同的输入中进行选择，找到所需要的一个作为指标。最通常的选择也是最坏的输入，这看起来很奇特，为什么不用最佳输入或者平均输入呢？

答案是，因为大多数情况并不具代表性，最佳输入通常会被排除，也就是说，这种情况下的问题通常是无足轻重的。平均输入是一个好的选择，但它有时很难被有效评估。首先，通常我们并不清楚“平均”输入是指什么，当然可以对不同情况下的不同参数进行平均，但如果不小心，平均的结果中将会包括许多实际上不可能出现的情况，从而使得评估变得很离谱。另一个与平均有关的严重问题是进行平均性能分析时数学上的困难。目前还没有全面、易用的平均状况分析技术。因此我们仅可对少数问题进行平均性能分析，绝大部分仍要凭借最坏状况分析。选择最坏输入作为指标是非常有益的。有些时候，从经验上说最坏输入非常接近于平均输入。另一方面，即使最坏输入完全不同于平均输入，那些在最坏输入情况下具有最佳性能的算法对于其他输入情况来说也有很好的性能。除非特别指明，本书使用的都是最坏情况分析技术。

总之，渐近分析和最坏情况分析，仅仅是对特定算法在特定输入作用下运行时间的近似，它们绝不可能给出事情的全貌，但在大多数情况下却是很好的指标。

3.2 符号 O

前面说过，在对特定算法进行运行时间评估时，常数项将被忽略。为了对此进行有效说明，我们需要一些特别的符号。如果存在常数项 c 和 N，使得对于所有的 $n \geqslant N$，有 $g(n) \leqslant cf(n)$，则称函数 $g(n)$ 相对于另一函数 $f(n)$是 $O(f(n))$。换句话说，对于足够大的 n，函数 $g(n)$ 不超过函数 $f(n)$ 的一个固定倍数。函数 $g(n)$ 可能比 $cf(n)$小，甚至是非常小；符号 O 仅是从上限来约束它。例如，$5n^2+15=O(n^2)$，因为当 $n \geqslant 4$时，$5n^2+15 \leqslant 6n^2$。同时也有 $5n^2+15=O(n^3)$，因为当 $n \geqslant 6$时，有 $5n^2+15 \leqslant n^3$。

符号 O 允许我们忽略常数项，因此在符号 O 中包括常数项是毫无意义的。我们总可以写出 $O(n)$，而不是 $O(5n+4)$；同样，可以写出 $O(\log n)$，而不必注明对数的底数，这是因为不同底数的对数间只有常数倍的差别。另外，可用 $O(1)$ 来标明一个常数。这样如果想在表达式的部分中说明常数项，也可以使用符号 O。例如，可以有 $T(n)=3n^2+O(n)$，或者 $S(n)=2n\mathrm{lb}\,n+5n+O(1)$。

通常，确定一个函数 $g(n)$ 是 $O(f(n)$并不那么容易。但本书算法分析中涉及的绝大部分函数相对简单。通过一些简单的规则，就可以处理绝大部分（但绝非全部）情况。最有用的规则如下：如果$n_1 \geqslant n_2$蕴含 $f(n_1) \geqslant f(n_2)$，则说函数 $f(n)$ 是单调递增的。

□ **定理 3.1**

对于所有常数 $c>0$ 和 $a>1$，以及所有单调递增函数 $f(n)$，有

$$(f(n))^c = O(a^{f(n)})。$$

换句话说，一个指数函数要比一个多项式函数增长得快。 □

这条规则可用于许多函数的比较。例如，如果我们用 n 来替换定理 3.1 中的 $f(n)$，则对于所有常数 $c>0$ 和 $a>1$，有

$$n^c = O(a^n)。 \tag{3.1}$$

另一个例子，可以用 $\log_a n$ 来替换 $f(n)$。对于所有常数 $c>0$ 和 $a>1$，有

$$(\log_a n)^c = O(a^{\log_a n}) = O(n)。 \tag{3.2}$$

可以使用下列规则对符号 O 进行加法和乘法操作。

□ **定理 3.2**

1. 如果 $f(n)=O(s(n))$ 并且 $g(n)=O(r(n))$，则 $f(n)+g(n)=O(s(n)+r(n))$。
2. 如果 $f(n)=O(s(n))$ 并且 $g(n)=O(r(n))$，则 $f(n)\cdot g(n)=O(s(n)\cdot r(n))$。

证明： 由定义，存在常数 c_1、N_1、c_2、N_2，使得对于所有 $n\geqslant N_1$ 有 $f(n)\leqslant c_1 s(n)$，并且对于所有 $n\geqslant N_2$ 有 $g(n)\leqslant c_2 r(n)$。取 c_1 和 c_2 中的最大值，以及 N_1 和 N_2 中的最大值，可证明以上两个命题。 □

由于符号 O 对应于关系“$\leqslant$”，所以不可能对其定义减法和除法运算。也就是说，通常 $f(n)=O(s(n))$ 和 $g(n)=O(r(n))$ 并不意味 $f(n)-g(n)=O(s(n)-r(n))$，或者 $f(n)/g(n)=O(s(n)/r(n))$（参见习题 3.15 和习题 3.16）。

关于渐近行为的重要性见表 3.1，它包括一些典型的运行时间，分别对应规模 $n=1000$ 时不同计算速度的算法的行为。从列到列，速度的区别为 2 倍，从每秒 1000 步到每秒 8000 步。可以清楚地看到计算速度（或者算法）的提高可使效率得到常数因子的提高，而使用更快的算法（在表中从下往上）也可得到效率的提高。但一个指数式的算法将需要天文数字（十亿个十亿年）的时间来处理 $n=1000$ 的问题（除非它的底数非常接近 1）。

符号 O 用来表示一个算法运行时间的上限；但仅仅使用上限这一概念是不够的。例如，本书中的所有算法，都有 $O(2^n)$ 的运行时间。即，它们的运行不会超出指数时间。然而，对于绝大多数算法而言，$O(2^n)$ 是一个很粗糙的上限，其运行远远快得多。我们不仅是对上限感兴趣，而且希望尽可能地使表达式接近真实的运行时间。如果找到这样的表达式非常困难，则希望为它找到一个最小上界和最大下界，获得下界要比获得上界更困难。算法运行时间的上界，仅仅意味着存在某些算法，这些算法不需要这么多的时间。下界则意味着，对于这个问题来说没有算法能够有比这更好的运行时间。当然，要逐个考虑所有可能的算法是不可行的。我们需要一个机制来对问题和算法进行建模，从而证明这个下界。有关下界的问题将在 6.4.6 节中进一步讨论。下面引入可忽略常数的下界记法。如果存在常数 c 和 N，使得对于所有的 $n\geqslant N$，在输入规模为 n 时解决问题的步数 $T(n)$ 至少为 $cg(n)$，则我们说 $T(n)=\Omega(g(n))$。例如，$n^2=\Omega(n^2-100)$、$n=\Omega(n^{0.9})$。符号 Ω 对应于关系“$\geqslant$”。

表 3.1 在不同假定（$n=1000$）下的运行时间（s）

运行时间	时间 1 1000 步/秒	时间 2 2000 步/秒	时间 3 4000 步/秒	时间 4 8000 步/秒
lb$n$	0.010	0.005	0.003	0.001
n	1	0.5	0.25	0.125
n lb$n$	10	5	2.5	1.25
$n^{1.5}$	32	16	8	4
n^2	1 000	500	250	125
n^3	1 000 000	500 000	250 000	125 000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

如果一个特定函数 $f(n)$ 同时满足 $f(n)=O(g(n))$ 和 $f(n)=\Omega(g(n))$，则称 $f(n)=\Theta(g(n))$。例如，$5n\,\mathrm{lb}n-10=\Theta(n\log n)$。（在表达式 $\Theta(n\log n)$ 中对数的底数可被忽略，这是因为不同的底数仅仅是为对数带来常数因子的改变。）用来证明 O 部分和 Ω 部分的常数项不需要一致。

符号 O、Ω 和 Θ 分别（大致）对应的是“$\leqslant$”、“$\geqslant$”和“=”。有时我们需要对应于“<”和“>”的符号。可称 $f(n)=o(g(n))$（读做“$f(n)$是 $g(n)$ 的小 o”），如果

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

例如，$n/\mathrm{lb}n=o(n)$，但是 $n/10\neq o(n)$。类似地，如果 $g(n)=o(f(n))$，则称 $f(n)=\omega(g(n))$。

可以通过把 O 替换为 o 来加强定理 3.1：

□ **定理 3.3**

对于所有常数 $c>0$ 和 $a>1$，以及所有单调递增函数 $f(n)$，有 $(f(n))^c=o(a^{f(n)})$。

换句话说，一个指数式函数要比一个多项式函数增长得快。 □

符号 $\mathcal{OO}$

这些年来，符号 O 的使用受到了许多批评，当然，主要的异议在于常数项。使用符号 O 可以很方便地略去常数。要记住的是，符号 O 仅给出了最初的近似，其目的很实际，它的使用在许多实践中促进了算法的发展，它也是复杂度理论的工具之一，可以使算法效率的许多方面得以清楚地展现。

然而，面对各种不同的算法，何时在符号 O 中忽略常数项应该绝对禁止，何时可以实行，在实践中把它们区分开来是很重要的。为了区分它们，本书引入了一个新符号。这并不是一个有精确数学定义的符号——仅用于替代某些算法中乏味的说明，在测量运行时间上它与符号 O 一样，只具有理论价值。我们建议用 $\mathcal{OO}(f(n))$（读做“$f(n)$的 oo”）表示一个 $O(f(n))$的函数，但它在大多数实际应用中具有很大的常数项。（这个符号很容易记住，它就像一个大的 ∞。）

符号 $\mathcal{OO}$ 的使用基于作者的判断，一个特定的常数项是否有“实际用处”并不容易定义。这里不想强化它的定义，主要目的是用简洁的形式向读者表明作者的看法。引入这个符号的另一目的是强调符号 O 并不是万能的。

3.3 时间与空间复杂度

如果不运行一个算法，那么又如何分析算法的运行时间呢？对算法的执行步数进行计数是一种方法。问题是存在不同类型的执行步骤，而且每一类需要的时间都不同。例如，除法要比加法需要更多的计算时间。一种分析算法的方法是分别对不同类别的步骤进行计数。但要把所有的步骤类别列举出来，在绝大多数情况下又是令人厌烦的。再有，不同步骤的执行也依赖于特定的计算机和程序语言，而我们却希望避免这种依赖性。

我们应该关注最主要的计算步骤，而不是对所有的步骤进行计数。例如，如果分析一个排序算法，则会选择比较来作为主要步骤。直觉上，元素的比较是排序的本质；剩余的其他步骤都可看成杂项开销。当然，还必须确定比较操作确实是算法的主要构成部分。由于忽略了常数项因子，因此检查其他全部操作的数目相对于比较操作数目的比例已足够。如果这样可行，并且如果 $O(f(n))$是比较数目的上界，则 $O(f(n))$也是所有步骤数目的上界。这时称一个算法的**时间复杂度**，或者**运行时间**是 $O(f(n))$。只要差别不超出常数倍，这种方法同样也可以解决不同步骤需要

不同计算时间的问题。

一个算法的**空间复杂度**表明的是运行该算法所需要的临时存储空间。在大多数情况下，不把输入和输出需要的空间作为空间复杂度的一部分。这是因为空间复杂度通常用于比较同一问题的不同算法，在这种情况下输入/输出的空间需求都是固定的。另外，没有了输入或者输出，算法将无法工作，我们只想对那些需要保存临时数据的空间进行计数。我们不对程序自身占用的空间计数，因为它是独立于输入规模的。类似于时间复杂度，空间复杂度也归诸于最坏情况，它通常表示为输入规模的渐近表达式。因此，一个 $O(n)$-space 的算法相对于每一输入单位需要常数数量的内存。一个 $O(1)$-space 的算法总共仅需要常数数量的内存，与输入的规模无关。

对主要运算步骤进行计数并不容易。下一节将简要讨论一些用于计算运行时间的数学技巧。与之相反，在大多数情况下，估计一个特定算法的空间复杂度通常比较直接，因此本书将不做讨论。

3.4 求和

如果一个算法由几部分组成，则它的复杂度通常是这些部分的复杂度的总和。但在许多情况下，结果并非如此简单。算法可能包含一个执行许多次的循环，每一次都有不同的复杂度。为了分析这种情况，我们需要一种技术来累计这些表达式的开销。最简单的情况是一个大小为 n 的循环，它的第 i 步（$i \leqslant n$）需要 i 个操作，则总共的操作数为 $1+2+\cdots+n$。我们用符号Σ来表示总和，则前面的式子可以写成 $\sum_{i=1}^{n} i$，它表示“对各项 i 进行相加，i 从 1 递增到 n”。就像在 2.2 节中看到的那样，它的值等于 $n(n+1)/2$。将这个总和与每一步需要 n 个操作的计算相比较，可以看出，通过把第 i 步的运行时间从 n 减至 i，可以节省约二分之一的时间。

□ 例 3.1

考虑执行一个循环，其中第 i 步需要 i^2 个操作。换句话说，寻求下式的总和：

$$S_2(n)=\sum_{i=1}^{n} i^2 \text{。}$$

显然，$S_2(n) \leqslant n^3$，因为 n^3 的和等于在循环的每一步中执行 n^2 个操作。由这个例子做出判断，可以猜测 $S_2(n)$ 与 n^3 之间的差异在常数项范围内。通过归纳，可以证明这个猜测，并找到这个常数项。猜测 $S_2(n)=P(n)=an^3+bn^2+cn+d$，其中 $P(n)$ 必须满足 $P(1)=1$，并且归纳步骤 $P(n+1)=P(n)+(n+1)^2$。此归纳步骤意味着

$$a(n+1)^3+b(n+1)^2+c(n+1)+d-(an^3+bn^2+cn+d)=n^2+2n+1\text{，}$$

该等式又蕴含着（因为相同指数的 n 的系数必须相等），即

$$\begin{aligned} 3a+b-b&=1 && n^2 \text{ 的系数，}\\ 3a+2b+c-c&=2 && n \text{ 的系数，} \end{aligned}$$

和

$$a+b+c+d-d=1 \qquad 1 \text{ 的系数。}$$

由这些等式导出 $a=1/3$、$b=1/2$ 和 $c=1/6$。d 的值可以由初始条件（$P(1)=1$）得到，即 $a+b+c+d=1$。从而 $d=0$。将所有的项组合起来，可得

$$S_2(n)=\frac{n^3}{3}+\frac{n^2}{2}+\frac{n}{6}=\frac{n(n+1)(2n+1)}{6}\text{。} \qquad (3.3)$$

进一步，可以注意到通过把第 i 步的大小从 n^2 降至 i^2，结果将节省约三分之一的时间。 □

另有一个途径可以得到式（3.3）。这是我们将在后面多次使用的技术。如果猜测 $S_2(n)$是一个3次多项式，则可以尝试把 $S_2(n)$表示成这种多项式的组合。通过解这样的等式以及其他多项式，可以得到 $S_2(n)$的解。考虑和式

$$S_3(n)=\sum_{i=1}^{n} i^3 。 \tag{3.4}$$

先把式（3.4）写成另一种形式：

$$S_3(n)=\sum_{i=1}^{n} i^3 = \sum_{i=1}^{n} (i-1+1)^3 = \sum_{i=0}^{n-1}(i+1)^3 = \sum_{i=0}^{n-1} (i^3+3i^2+3i+1) 。 \tag{3.5}$$

换句话说，转换这个和式，使得它从 0 到 $n-1$ 相加，而不是从 1 到 n 相加。这个转换在图 3.1 中做了说明。由式（3.5）的左式等于右式，有

$$\sum_{i=1}^{n} i^3 = \sum_{i=0}^{n-1} (i^3+3i^2+3i+1) 。 \tag{3.6}$$

等式（3.6）左右两边都有从 1 直到 n–1 的 i^3 项，可以被抵销掉。写出包含两边剩余项的等式。

$$n^3 = 0^3 + \sum_{i=0}^{n-1} (3i^2+3i+1)$$

已经知道 $\sum_{i=0}^{n-1} i = n(n-1)/2$，并且显然有 $\sum_{i=0}^{n-1} i^2 = S_2(n)-n^2$（惟一的差别在于第 n 项）。因此，

$$n^3 = 3(S_2(n)-n^2)+3n(n-1)/2+n 。$$

现在可以求解 $S_2(n)$：

$$n^3 - 3n(n-1)/2 - n = 3(S_2(n)-n^2) ,$$

这意味着

$$S_2(n)=\frac{n^3-3n(n-1)/2-n}{3}+n^2=\frac{n^3}{3}+\frac{3n^2}{6}+\frac{n}{6}=\frac{n(n+1)(2n+1)}{6} ,$$

当然，它与表达式（3.3）相同。

		1^3	+	2^3	+	$\cdots$	+	$(n-1)^3$	+	n^3
$(0+1)^3$	+	$(1+1)^3$	+	$(2+1)^3$	+	$\cdots$	+	$(n-1+1)^3$		

图 3.1 和式的转换计算

这个推导的主要技巧是利用了两种不同方式计算一个特定的和式（本例中是$S_3(n)$），使得绝大部分项得以相互抵销。许多其他和式的计算也会出现同样的情况。如果考虑和式 $f_1+f_2+\cdots+f_n$ 及其变换 $f_2+f_3+\cdots+f_{n+1}$，可以看到系数的绝大部分可以互相抵销。

最后只剩下边界项。下面三个例子都用来说明这一技巧。

□ 例 3.2

计算和式

$$F(n)=\sum_{i=0}^{n} 2^i = 1+2+4+\cdots+2^n 。$$

将 $F(n)$ 与另一种包含 $F(n)$ 的表达式对比，从而消去大部分项。$F(n)$ 的连续项之间的差别是因子

2，所以将整个表达式乘以 2（从而得到转换）：

$$2F(n)=2+4+8+\cdots+2^n+2^{n+1}。$$

现在得到一个包含 $F(n)$ 的式子

$$2F(n)-F(n)=2^{n+1}-1。$$

从而有 $F(n)=2^{n+1}-1$。 □

□ 例 3.3

考虑下面这个略微困难的和式：

$$G(n)=\sum_{i=1}^{n} i2^i=1\cdot 2^1+2\cdot 2^2+3\cdot 2^3+\cdots+n\cdot 2^n。$$

应用相同的技巧：

$$2G(n)=1\cdot 2^2+2\cdot 2^3+3\cdot 2^4+\cdots+n\cdot 2^{n+1},$$

这里只是简单地对幂进行了增量。将两个式子相减以消除因子 i 的影响：

$$G(n)=2G(n)-G(n)=n\cdot 2^{n+1}-(1\cdot 2^1+1\cdot 2^2+\cdots+1\cdot 2^n)$$
$$=n\cdot 2^{n+1}-(2^{n+1}-2)=(n-1)2^{n+1}+2。$$ □

□ 例 3.4

最后，考虑下面的和式，该和式还将在 6.4.5 节分析堆排序时出现：

$$G(n)=\sum_{i=1}^{n} i2^{n-i}=1\cdot 2^{n-1}+2\cdot 2^{n-2}+3\cdot 2^{n-3}+\cdots+n\cdot 2^0。$$

利用相同的技巧：

$$2G(n)=1\cdot 2^n+2\cdot 2^{n-1}+3\cdot 2^{n-2}+\cdots+n\cdot 2^1。$$

进一步，两式相减以消除因子 i 的影响：

$$G(n)=2G(n)-G(n)=2^n+1\cdot 2^{n-1}+1\cdot 2^{n-2}+\cdots+1\cdot 2^1-n\cdot 2^0。$$
$$=2^{n+1}-2-n。$$ □

3.5 递推关系

递推关系（recurrence relation，又称递归关系）是一种函数定义方式，它在函数的定义体中包含了函数本身。最著名的递推关系可能是斐波那契数的定义：

$$F(n)=F(n-1)+F(n-2),\quad F(1)=1,\quad F(2)=1。\qquad (3.7)$$

该表达式定义了一个惟一的函数。可以从表达式计算出每一个数 k 所对应的 $F(k)$的值。例如，$F(3)=F(2)+F(1)=2$，$F(4)=F(3)+F(2)=3$，如此等等。然而，如果通过定义来计算函数的值，则需要 $k-2$ 个步骤才能得到 $F(k)$。如果有一个 $F(n)$的显式（或称闭形式的）表达式，那就方便得多了。我们可以快速计算出 $F(n)$并与其他函数相比较，这个过程被称为求解递推关系。有时也把递推关系简称为递推。

递推关系常出现在算法分析中。在这里我们简要讨论一下求解递推关系的技术，并且提供两类递推问题的通解，它们常用于大多数算法的分析中，当然本书后面也将用到。

3.5.1 巧妙地猜测

猜测一个问题的解看起来不是一种科学的方法，但是，先把偏见放在一边，后面我们将看到这种方法可以应用于求解一大类的递推关系。当不是试图得到精确解，而是寻找算法的上界时，它甚至能应用得更好。主要原因在于验证界是有效的要比计算出一个问题的界更容易。考虑以下递推关系，它由 n 的值定义，而 n 为 2 的幂：

$$T(2n)\leqslant 2T(n)+2n-1,\ T(2)=1\ 。 \qquad (3.8)$$

此处递推是一个不等式而非等式。这与问题的目标一致，即仅仅寻找一个上界（使用符号 O 的形式），这样式子右边表现出的就是最坏情况。我们想找到一个函数 $f(n)$，使得 $T(n)=O(f(n))$，但是也希望 $f(n)$ 不要离真实的 $T(n)$太远。

对 $f(n)$ 给出一个猜测，即 $f(n)=n^2$，通过对 n 进行归纳来证明 $T(n)=O(f(n))$。首先，检查归纳基础。在此情况下，$T(2)=1\leqslant f(2)=4$。然后证明 $T(n)\leqslant f(n)$ 蕴含 $T(2n)\leqslant f(2n)$。我们需要证明

$$T(n)\leqslant n^2 \text{ 蕴含着 } T(2n)\leqslant (2n)^2。$$

过程如下：

$$\begin{aligned} T(2n) &\leqslant 2T(n)+2n-1 \text{（由递归定义），} \\ &\leqslant 2n^2+2n-1 \text{（由归纳假设），} \\ &< (2n)^2, \end{aligned}$$

这正是我们想要证明的。这样，$T(n)=O(n^2)$，但 n^2 是 $T(n)$的一个比较好的估计吗？在证明的最后一步，$2n^2+2n-1$ 被更大的 $4n^2$所替代。但是在这两个表达式之间存在一个很大的间隙（大约为 $2n^2$），这给了我们一个提示，n^2 可能是对 $T(n)$过高的估计。

让我们尝试一个较小的估计，即 $f(n)=cn$，其中 c 为常数。显然，cn 的增长要比 $T(n)$更慢，因为 $c2n=2cn$，因此已经没有空间可以容下额外的 $2n-1$。因此，$T(n)$应介于 cn 和 n^2 之间。

让我们尝试 $T(n)\leqslant n\ \mathrm{lb}n$。显然，$T(2)<2\mathrm{lb}2$。假设 $T(n)\leqslant n\ \mathrm{lb}n$，则考虑 $T(2n)$：

$$\begin{aligned} T(2n) &\leqslant 2T(n)+2n-1 \text{（由递推定义），} \\ &\leqslant 2n\ \mathrm{lb}\ n+2n-1 \text{（由归纳假设），} \\ &< 2n(\mathrm{lb}\ 2n), \end{aligned}$$

这正是我们想要证明的。现在证明中的偏差只有 1 了，已经非常接近。后面将证明这确实是一个不超出常数倍偏差的解。

递推关系（3.8）仅被定义为 n 为 2 的幂时的值。通过下面的方式，我们可以定义一个类似的递推关系，它作用在所有 n 上：

$$T(n)\leqslant 2T(\lfloor n/2\rfloor)+n-1,\quad T(2)=1\ 。 \qquad (3.9)$$

（注意，取整符号是必需的，因为 $T(n)$仅仅定义在整数上。）递推关系（3.9）比递推关系（3.8）更通用，因为它定义在所有 n 上，而不仅是 2 的幂上，递推关系（3.9）正好类似于递推关系（3.8）。所以，我们已经知道，当 n 为 2 的幂时，$T(n)=O(n\log n)$。现在我们来说明同样的上界适用于 $T(n)$的所有值。显然，$T(n)$是一个单调增长的函数。如果 n 不是 2 的幂，则 $T(n)$不会大于$T(2^k)$，

其中 2^k 是大于 n 的最小的 2 的幂，即 $2^{k-1} < n < 2^k$；显然有 $T(2^{k-1}) \leqslant T(n) \leqslant T(2^k)$。我们证明了，对于某些常数 c，有 $T(2^k) \leqslant c2^k \text{lb}\, 2^k$。因此，

$$T(n) \leqslant c2^k \text{lb}\, 2^k \leqslant c(2n)lb(2n) \leqslant c_1 n \text{ lb } n。$$

c_1 为另一个常数，这意味着对于所有的 n，有 $T(n) = O(n \log n)$。当我们寻找一个渐近表达式时，假定 n 为 2 的幂就足够了。

总结一下求解递推关系的归纳证明步骤。假设有如下通用递推关系：

$$T(g(n)) = E(T, n), \tag{3.10}$$

其中 $g(n)$是一个 n 的函数（用来定义递推的增长），E 是包含 $T(n)$和 n 的表达式。例如，在式（3.8）中，$g(n) = 2n$，而 $E(T, n) = 2T(n)+2n-1$。进一步，我们猜想对于某些函数 $f(n)$有 $T(n) \leqslant f(n)$。为了证明猜测，需要把 $f(n)$中的 n 替换为 $g(n)$，然后把 E 中每次出现的 $T(n)$替换为 $f(n)$。还需要说明 $f(g(n))$大于或等于 $E(T, n)$替换后的值。换句话说，必须证明

$$f(g(n)) \geqslant E(f, n)。 \tag{3.11}$$

例如，在式（3.8）中猜测 $f(n) = n \text{ lb } n$；从而，必须说明$(2n)(\text{lb}(2n)) \geqslant 2(n \text{ lb } n)+2n-1$。

一个常见的错误是试图证明相反的一面，即把“大于”换成“小于”。一个直觉的，容易记住的解释是：由于试图证明 $f(n)$要比 $T(n)$增长得快，因此，如果把 $f(n)$中的 n 替换成 $g(n)$，其值应比把 $T(n)$中的 n 替换成 $g(n)$所得到的值更大，而 $T(g(n)) = E(T, n)$(递推定义)，因此接着就可以用 $E(f, n)$替换 $T(g(n))$。这个过程可能需要使用不同的函数反复进行猜测，直到不等式的证明足够严密（tight）为止。

另一个错误是当进行猜测时使用符号 O。即猜测解为 $O(f(n))$，并且试图用 $O(f(n))$ 替换 n。然而，符号 O 不可以这样使用。使用符号 O 的问题在于，即使最终不关心常数项，我们也不能在证明过程中忽略它。例如，如果试图通过把 n 替换为$O(n)$来证明式（3.8）的解为 $O(n)$，那么将得到以下结果（基础情况是平凡的）：

$$T(2n) \leqslant 2T(n) + 2n - 1 \text{（由递推定义）},$$

$$\leqslant O(n) + 2n - 1 \text{（由归纳假设）},$$

$$= O(n),$$

由前面的内容，我们知道这个结果是错的。这个错误是由于在“证明”的不同阶段使用（或者忽略了）不同的常数。正确的方法是把常数项显式包括进来。当想猜测解为 $O(f(n))$ 时，我们猜测它是 $cf(n)$，其中 c 是某个常数，然后再确定 c 的值。

现在让我们尝试用猜测法来求解斐波那契递推关系。再一次给出

$$F(n)=F(n-1)+F(n-2),\ F(1)=1,\ F(2)=1。 \tag{3.12}$$

既然 $F(n)$的值是前面两项之和，一个合理的猜测是 $F(n)$每次都被加倍，即它大约为 2^n。让我们试试 $F(n) = c2^n$。在式（3.12）中用 $c2^n$ 进行替换，可得

$$c2^n = c2^{n-1} + c2^{n-2}。$$

这个等式显然不可能，因为 c 被抵销掉后左式总是大于右式。所以我们知道 $c2^n$ 太大了，并且乘法常量 c 在归纳步骤中不起作用。

接下来尝试另一个指数函数，但底数要小一些。用不着猜测底数，我们引入一个参数作为底数，在验证过程中再计算出它的值。我们尝试 $F(n) = a^n$，其中 a 是一个常数。在式（3.12）中用 a^n 代入，可得

$$a^n = a^{n-1} + a^{n-2},$$

这意味着

$$a^2 = a + 1。 \tag{3.13}$$

式（3.13）的两个解是 $a_1 = (1+\sqrt{5})/2$ 和 $a_2 = (1-\sqrt{5})/2$。所以，现在我们知道 $F(n) = O((a_1)^n)$，因为 $(a_1)^n$ 满足递推关系，并且可以容易地找到一个常数 c，使得 $c(a_1)^n$ 比 $n=1$ 和 $n=2$ 时给定的值都要大。

如果想为 $F(n)$ 找到一个确切的值，那么需要更仔细地考虑它的初始值。由于 $(a_1)^n$ 和 $(a_2)^n$ 都满足递推关系，所以它们的任意线性组合也都满足。因此这个递推的通解是

$$c_1(a_1)^n + c_2(a_2)^n。$$

我们需要计算出 c_1 和 c_2 的值，使得表达式满足 $F(1)$ 和 $F(2)$ 的值。可以很容易地验证 $c_1 = 1/\sqrt{5}$ 而 $c_2 = -1/\sqrt{5}$。所以，斐波那契关系的精确解是

$$F(n) = \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^n - \frac{1}{\sqrt{5}}\left[\frac{1-\sqrt{5}}{2}\right]^n。$$

求解递推关系（3.12）时遇到的等式 $a^2 = a+1$ 通常被称为递推关系的**特征等式**，使用的技术是求解下述递推关系的基础：

$$F(n) = b_1F(n-1) + b_2F(n-2) + \cdots + b_kF(n-k),$$

其中 k 为一个常数。

3.5.2 分治关系

在分治法中，问题通常被分割成几个子问题，其中每个子问题也将被递归求解，得到结果后，再使用一个操作来组合子问题的解以得到原始问题的解。假设有 a 个子问题，每个问题的规模是原始问题的 $1/b$，并且用于组合各个子问题解的算法运行时间是 cn^k，其中 a、b、c 和 k 是常数，即

$$T(n) = aT(n/b) + cn^k。 \tag{3.14}$$

为了简化，假定 $n = b^m$，则 n/b 总是一个整数（b 是一个比 1 大的整数）。先尝试展开式（3.14）两次，找一下感觉：

$$T(n) = a(aT(n/b^2)+c(n/b)^k)+cn^k = a(a(aT(n/b^3)+c(n/b^2)^k)+c(n/b)^k)+cn^k。$$

一般来说，如果展开到 $n/b^m = 1$，可得

$$T(n) = a(a(\cdots T(n/b^m) + c(n/b^{m-1})^k) + \cdots) + cn^k。$$

假设 $T(1) = c$（不同的值对最终结果的影响仅仅是常数上的变化）。那么

$$T(n) = ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \cdots + cb^{mk},$$

这意味着

$$T(n) = c\sum_{i=0}^{m} a^{m-i}b^{ik} = ca^m\sum_{i=0}^{m}\left(\frac{b^k}{a}\right)^i。$$

但是，这是一个简单的几何级数，分三种情况，分别取决于(b^k/a)是否小于、大于或等于1。

情况1： $a > b^k$

在这种情况下，几何级数的因子小于 1，所以当 m 趋于无穷时，级数收敛于常数。因此，$T(n) = O(a^m)$。由于 $m = \log_b n$，我们得到 $a^m = a^{\log_b n} = n^{\log_b a}$（最后一个等式可通过对两边都取以 b 为底数的对数得证）。所以有

$$T(n) = O(n^{\log_b a})。$$

情况2： $a = b^k$

在此情况下，几何级数的因子等于 1，从而 $T(n) = O(a^m m)$。注意 $a = b^k$ 意味着 $\log_b a = k$ 且 $m = O(\log n)$。所以有

$$T(n) = O(n^k \log n)。$$

情况3： $a < b^k$

在此情况下，几何级数的因子大于 1。为了计算几何级数的和，可以使用标准表达式。用 F（F 是一个常数）来表示 b^k/a。由于级数的第一个元素是 a^m，我们有

$$T(n) = a^m \frac{F^{m+1} - 1}{F - 1} = O(a^m F^m) = O((b^k)^m) = O((b^m)^k) = O(n^k)。$$

以上三种情况总结在下面的定理中。

□ **定理 3.4**

a 和 b 为整数常数，$a \geqslant 1$、$b \geqslant 2$ 且 c 和 k 是正常数，则递推关系 $T(n) = aT(n/b) + cn^k$ 的解是

$$T(n) = \begin{cases} O(n^{\log_b a}), & \text{如果 } a > b^k \\ O(n^k \log n), & \text{如果 } a = b^k \\ O(n^k), & \text{如果 } a < b^k \end{cases}$$

□

定理 3.4 的结果适用于许多分治算法。这个结论必须牢记。该结论对算法设计阶段也很有帮助，因为它可用于预测运行时间。在习题中，我们将讨论公式的一般化表示。

3.5.3 涉及全部历史的递推关系

一个**涉及全部历史的递推关系**是依赖于先前所有函数值的函数，而并非仅与一个或几个有关。一个最简单的此类函数是

$$T(n) = c + \sum_{i=1}^{n-1} T(i)，\qquad (3.15)$$

其中 c 是一个常数且 $T(1)$给定。我们可以使用与计算和式相同的方法来解这个递归关系。先尝试把这个递归写成大多数项都能被消去的形式（这个方法有时被称为**历史消除法**）。对于递归（3.15），对比 $T(n+1)$和$T(n)$

$$T(n+1) = c + \sum_{i=1}^{n} T(i)。\qquad (3.16)$$

如果把式（3.15）和式（3.16）相减，有 $T(n+1) - T(n) = T(n)$。所以，$T(n+1) = 2T(n)$，这显然蕴含着 $T(n+1) = T(1)2^n$。这个断言对 $T(1)$为真，由归纳法，如果$T(n)$为真，则$T(n+1)$为真，因

为每一次都对前面的值加倍。

这个论证可能是“显然的”，但它并不正确！例如，设 $T(1)=1$ 且 $c=5$，可以看到 $T(2)=6\neq 2T(1)$。这是进行归纳证明时忽略基础情况的又一例子。此错误结果来自于此证明对 $T(2)$ 不成立，因为 $T(1)$ 没有被删除。当一个表达式中某些参数（本例中是 c）没有出现在最终结果中时，这就很值得怀疑。为了正确地解决这个问题，我们明确 $T(2)=T(1)+c$（通过定义）以及以上证明只对所有 $n>2$ 成立。因此，$T(n+1)=(T(1)+c)2^{n-1}$。

上面的递推非常一般，接下来的这个就不简单了，它将在 6.4.4 节中讨论快速排序的平均情况时出现，该递推关系是

$$T(n)=n-1+\frac{2}{n}\sum_{i=1}^{n-1}T(i) \quad （对于 n\geqslant 2）。\ T(1)=0。 \tag{3.17}$$

使用替换和项的消除技术以消除大部分 $T(i)$ 项。让我们看一下 $T(n+1)$ 相应的表达式：

$$T(n+1)=(n+1)-1+\frac{2}{(n+1)}\sum_{i=1}^{n}T(i) \quad (n\geqslant 2)。 \tag{3.18}$$

为了方便，对式（3.17）的两边乘以 n，对式（3.18）的两边乘以 $(n+1)$，有

$$nT(n)=n(n-1)+2\sum_{i=1}^{n-1}T(i) \quad (n\geqslant 2)。 \tag{3.19}$$

$$(n+1)T(n+1)=(n+1)n+2\sum_{i=1}^{n}T(i) \quad (n\geqslant 2)。 \tag{3.20}$$

将式（3.19）减去式（3.20），有

$$(n+1)T(n+1)-nT(n)=(n+1)n-n(n-1)+2T(n)=2n+2T(n) \quad (n\geqslant 2),$$

这意味着

$$T(n+1)=\frac{n+2}{n+1}T(n)+\frac{2n}{n+1} \quad (n\geqslant 2)。$$

这个递推关系很容易求解。首先，我们把 $\frac{2n}{n+1}$ 替换成 2，得到一个近似：

$$T(n+1)\leqslant\frac{n+2}{n+1}T(n)+2 \quad (n\geqslant 2)。 \tag{3.21}$$

如果展开式（3.21），可以得到

$$T(n)\leqslant 2+\frac{n+1}{n}\left[2+\frac{n}{n-1}\left[2+\frac{n-1}{n-2}\left[\cdots\frac{4}{3}\right]\right]\right]$$

$$=2\left[1+\frac{n+1}{n}+\frac{n+1}{n}\frac{n}{n-1}+\frac{n+1}{n}\frac{n}{n-1}\frac{n-1}{n-2}+\cdots+\frac{n+1}{n}\frac{n}{n-1}\frac{n-1}{n-2}\cdots\frac{4}{3}\right]$$

$$=2\left[1+\frac{n+1}{n}+\frac{n+1}{n-1}+\frac{n+1}{n-2}+\cdots+\frac{n+1}{3}\right]$$

$$=2(n+1)\left[\frac{1}{n+1}+\frac{1}{n}+\frac{1}{n-1}+\cdots+\frac{1}{3}\right]$$

$$=2(n+1)(H(n+1)-1.5),$$

其中 $H(n)=1+1/2+1/3+\cdots+1/n$ 是调和级数。调和级数有一个简单的近似，这里我们不做证明，即 $H(n)=\ln n+\gamma+O(1/n)$，其中 $\gamma=0.577..$ 是**欧拉常数**。因此，$T(n)$ 的解是

$$T(n)\leqslant 2(n+1)(\ln n+\gamma-1.5)+O(1)=O(n\log n)。$$

3.6 一些有用的证明论据

在这一节中，我们不进行证明，仅提供一些在算法分析中有用的等式或不等式。

算术级数

$$1+2+3+\cdots+n=\frac{n(n+1)}{2}。\tag{3.22}$$

更一般地，如果 $a_n=a_{n-1}+c$，其中 c 是常数，则

$$a_1+a_2+a_3+\cdots+a_n=\frac{n(a_n+a_1)}{2}。\tag{3.23}$$

几何级数

$$1+2+4+\cdots+2^n=2^{n+1}-1。\tag{3.24}$$

更一般地，如果 $a_n=ca_{n-1}$，其中 $c\neq 1$ 是一个常数，则

$$a_1+a_2+a_3+\cdots+a_n=a_1\frac{c^n-1}{c-1}。\tag{3.25}$$

如果 $0<c<1$，则无穷几何级数的和是

$$\sum_{i=1}^{\infty}a_i=\frac{a_1}{1-c}。\tag{3.26}$$

平方和

$$\sum_{i=1}^{n}i^2=\frac{n(n+1)(2n+1)}{6}。\tag{3.27}$$

调和级数

$$H_n=\sum_{k=1}^{n}\frac{1}{k}=\ln n+\gamma+O(1/n)。\tag{3.28}$$

其中 $\gamma=0.577..$ 是欧拉常数。

对数的基本规则

$$\log_b a=\frac{1}{\log_a b}。\tag{3.29}$$

$$\log_a x=\frac{\log_b x}{\log_b a}。\tag{3.30}$$

$$b^{\log_b x}=x。\tag{3.31}$$

$$b^{\log_a x}=x^{\log_a b}。\tag{3.32}$$

对数和

$$\sum_{i=1}^{n}\lfloor \mathrm{lb}\, i\rfloor=(n+1)\lfloor \mathrm{lb}\, n\rfloor-2^{\lfloor \mathrm{lb}\, n\rfloor+1}+2=\Theta(n\log n)。\tag{3.33}$$

用积分对求和进行约束

如果$f(x)$是一个单调增长的连续函数，则

$$\sum_{i=1}^{n} f(i) \leqslant \int_{x=1}^{x=n+1} f(x)\mathrm{d}x。 \tag{3.34}$$

斯特林近似

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1+O(1/n))。 \tag{3.35}$$

特别地，斯特林近似蕴含着$\mathbf{lb}(n!) = \Theta(n \log n)$。

3.7 小结

波尔曾经说过，“预测是极其困难的，尤其是展望未来。”预测一个算法的行为虽然并不困难，但也绝非易事。这里使用的主要方法是近似，即忽略细节而提取算法最重要的特征。在此，使用符号O是有帮助的，但是必须记住它只是一个初步的近似。另一方面，算法分析的困难并不会阻碍算法的设计，获得算法效率的某些暗示，才是算法分析的本质。

在许多情况下，特别是使用递归算法时，我们可以得到一个递推关系。处理递推关系的首要任务，是考察它前面的若干项，这可以使我们对此关系的行为有些感觉，虽然这有助于对解做出猜测，但还不够。另一个有用的步骤是将递推关系多次展开，就像我们在 3.5.2 节中做的那样。先猜测然后进行验证是求解递推关系的一种较好的技巧，但它也仅仅是初步的技术，有时我们必须小心以免“过度猜测”，这时虽然可得一个正确的上界，但结果太保守了。还有许多其他技巧就不一一提及了。幸运的是，实际算法可能导致的一些递推关系，大部分已在本章进行了描述。在许多情况下，第 1 步通常是假定n具有特定的形式，如n为 2 的幂，这已经足够了。

参考文献和补充读物

渐近分析的思想在 20 世纪 70 年代出现，但遭到了一些人的反对。现在大多数渐近分析方法都是针对算法的效率分析的。一些书籍（主要是离散数学和组合学）覆盖了算法分析中的和式计算、递推关系以及其他一些技巧，如 Brualdi [1977]、Bavel [1982]、Roberts [1984]以及 Graham, Knuth, and Patashnik [1989]。也有一些书籍完全致力于算法分析，如 Knuth [1973a]包含了丰富的资料。其他一些书籍和综述论文可参见 Greene and Knuth [1982]、Lueker [1980]、Purdom and Brown [1985a]、Flajolet and Vitter [1987]以及 Hofri [1987]。

Knuth [1976]讨论了符号O的相关问题。解决递推关系的其他技巧可以在 Lueker [1980]，以及 Bentley, Haken, and Saxe [1980]中找到（后者包括习题 3.23 和习题 3.24 的解答）。Tarjan [1985]讨论了摊销的复杂度，这是一个在更精确的程度上分析特定算法运行时间的优美方法；如果一个算法的某些部分被执行多次，每次有不同的运行时间，则不必每次都采用最坏情况，而是分摊不同的代价。习题 3.19 的递推关系来自于 Manber [1986]。习题 3.21 来自于 Purdom and Brown [1985a]。

操练型习题

3.1 证明，如果$P(n)$是一个n的多项式，则$O(\log(P(n))) = O(\log n)$。

3.2 证明，如果$f(n)=O(g(n))$，则$f(n)=O(g(n))$。反之是否成立？

3.3 利用定理3.1，证明：

$$n(\log_3 n)^5 = O(n^{1.2})。$$

3.4 利用定理3.1，证明，对于所有的常数 $a, b > 0$，有

$$(\mathrm{lb}\, n)^a = O(n^b)。$$

3.5 比较以下各对函数，对每一种情况，说明是否 $f(n)=O(g(n))$ 或 $f(n)=\Omega(g(n))$ 或 $f(n)=\Theta(g(n))$。

	$f(n)$	$g(n)$
a.	$100n+\log n$	$n+(\log n)^2$
b.	$\log n$	$\log(n^2)$
c.	$\frac{n^2}{\log n}$	$n(\log n)^2$
d.	$(\log n)^{\log n}$	$\frac{n}{\log n}$
e.	$n^{1/2}$	$(\log n)^5$
f.	$n2^n$	3^n

3.6 求解下面的递推关系，给出确切解：

$$T(n)=T(n-1)+n/2;\ T(1)=1。$$

3.7 求解下面的递推关系，给出确切解：

$$T(n)=8T(n-1)-15T(n-2);\ T(1)=1;\ T(2)=4。$$

3.8 证明 $T(n)$ 满足 $T(n)=O(n\log^2 n)$，其中 $T(n)$ 为如下递推关系：

$$T(n)=2T(\lfloor n/2\rfloor)+2n\,\mathrm{lb}\, n,\ T(2)=4。$$

3.9 以下递推关系描述了一个矩阵乘法的递归算法的运行时间，参见[Pan 1978]。这个算法的渐近运行时间是什么？

$$T(n)=143640T(n/70)+O(n^2),\ T(1)=1。$$

3.10 找到以下分析的错误。令 A 为一个应用于完全二叉树（即所有叶子具有同样深度的二叉树）上的算法。假设 A 对树上的每片叶子执行 $O(k)$ 步操作，其中 k 是与叶子存储的信息量有关的参数（否则它就独立于此树），并且对每一个内部节点的执行时间为常数 c。则算法的全部运行时间为 $O(k)$。

错误证明：对树的节点数 n 进行归纳。若 $n=1$，则全部的步数显然为 $O(k)$。假定断言对所有节点数小于 n 的完全二叉树为真，则考虑一棵节点数为 n 的完全二叉树。此树包含一个节点和两棵子树，每棵子树的节点数为 $(n-1)/2$。由归纳假设，两棵子树的运行时间为 $O(k)$。因此，此树的运行时间为 $O(k)+O(k)+c$。但是它又等于 $O(k)$，从而得证。

3.11 求解下面涉及全部历史的递推关系：

$$T(n)=\max_i\{T(i)\}，\text{其中}\ T(1)=1。$$

3.12 求解下面涉及全部历史的递推关系：

$$T(n)=n+\sum_{i=1}^{n-1}T(i)，\text{其中}\ T(1)=1。$$

3.13 利用式（3.34），证明对于所有正整数 k，有

$$\sum_{i=1}^{n} i^k = O(n^{k+1})。$$

3.14 利用式(3.34)，证明对于所有正整数 k，有

$$\sum_{i=1}^{n} i^k \operatorname{lb} i = O(n^{k+1} \log n)。$$

创造型习题

3.15 为以下断言寻找一个反例：$f(n)=O(s(n))$ 且 $g(n)=O(r(n))$ 蕴含 $f(n)-g(n)=O(s(n)-r(n))$。

3.16 为以下断言寻找一个反例：$f(n)=O(s(n))$ 且 $g(n)=O(r(n))$ 蕴含 $f(n)/g(n)=O(s(n)/r(n))$。

*3.17 寻找两个单调递增函数 $f(n)$ 和 $g(n)$，使得 $f(n)\neq O(g(n))$ 且 $g(n)\neq O(f(n))$。

3.18 考虑递推关系：

$$T(n)=2T(n/2)+1,\ T(2)=1。$$

证明 $T(n)=O(n)$（可把注意力限制在 2 的指数上）。我们猜测 $T(n)\leqslant cn$，c 是某个未知常数，在表达式中进行 cn 替换（参见 3.5.1 节）。我们必须说明 $cn\geqslant 2c(n/2)+1$。但这显然不成立。为这个递归寻找一个正确的解（可以假定 n 是 2 的指数），并且解释为什么前面的尝试会失败。

3.19 给出满足以下递推关系的 $S(n)$ 的渐近行为：

$$S(mn)\leqslant cm \log_2 m\, S(n)+O(mn),\ S(2)=1,$$

其中 m 和 c 是常数参数（其解应该是 n、m 和 c 的函数）。

3.20 证明以下递推关系的渐近解：

$$T(n)=2T(n-c)+k,$$

其中 c 和 k 是整数常数，问是否对于某个常数 d 有 $T(n)=O(d^n)$？

3.21 以下递推关系出现在分治算法中，其问题被分成规模不等的几部分：

$$T(n)=\sum_{i=1}^{k} a_i T(n/b_i)+cn。$$

所有 a_is 和 b_is 都是常数，且满足

$$1-\sum_{i=1}^{k} a_i/b_i>0。$$

给出该递推关系的渐近行为（通过猜测和验证）。

3.22 求解下面的两个递推关系。找到 $T(n)$ 的渐近行为已足够。

a. $T(n)=4T\left(\left\lceil\sqrt{n}\right\rceil\right)+1;\ T(2)=1$。

b. $T(n)=2T\left(\left\lceil\sqrt{n}\right\rceil\right)+2n;\ T(2)=1$。

（提示：把 n 替换成另一个变量。）

3.23 证明递推关系

$$T(n)=kT(n/2)+f(n),\ T(1)=c$$

的解是

$$T(n)=n^{\log k}(c+g(2)+g(4)+\cdots+g(n)),$$

其中 $g(m)$ 被定义为 $f(m)/m^{\log k}$。可以假定 n 是 2 的幂（这是一个比 3.5.2 节中介绍的解更一般的解，因为它适用于所有的函数 $f(n)$）。

3.24 证明递推关系

$$T(n)=kT(n/d)+f(n),\ T(1)=c$$

的解是

$$T(n)=n^{\log_d k}(c+g(d)+g(d^2)+\cdots+g(n)),$$

其中 $g(m)$ 被定义为 $f(m)/m^{\log k}$。可以假定 n 是 d 的指数。

3.25 寻找以下递推关系定义的函数 $T(n)$ 的渐近行为：

$$T(n)=T(n/2)+T(\lfloor\sqrt{n}\rfloor)+n,\ T(1)=1,\ T(2)=2。$$

可以只考虑 n 为 2 的幂时的值。

3.26 寻找以下递推关系所定义的函数 $T(n)$ 的渐近行为：

$$T(n)=T(n/2)+\sqrt{n},\ T(1)=1。$$

可以只考虑 n 为 2 的幂时的值。

3.27 寻找以下由递推关系定义的函数 $T(n)$ 的渐近行为：

$$T(n)=T(n/2)+T\left(\left\lfloor\frac{n}{\mathrm{lb}\,n}\right\rfloor\right)+n\ (n>2),\ T(1)=1,\ T(2)=2。$$

可以只考虑 n 为 2 的幂时的值。

3.28 求解以下递推关系，给出 $T(n)$ 的渐近行为。另外给出确信的证据，说明你找到的函数 $f(n)$ 满足 $f(n)=\Theta(T(n))$。

$$T(n)=2T\left(\left\lfloor\frac{n}{\mathrm{lb}\,n}\right\rfloor\right)+3n\ (n>2),\ T(1)=1,\ T(2)=2。$$

3.29 虽然在通常情况下计算递归仅考虑 n 是 2 的幂已足够，但有时并不是如此。考虑下面的关系：

$$T(n)=\begin{cases}T(n/2)+1, & \text{如果 } n \text{ 为偶数}\\ 2T((n-1)/2), & \text{如果 } n \text{ 为奇数}\end{cases}$$

其中 $T(1)=1$。

a. 证明对于 2 的幂，递归的解是 $T(2^k)=k+1$（即对于 2 的幂，$T(n)=O(\log n)$）。

b. 证明，当 n 的值趋于无穷时，$T(n)=\Omega(n)$。讨论为什么 2 的幂的假定在此失效。

3.30 利用式（3.34）证明

$$S(n)=\sum_{i=1}^{n}\lceil\mathrm{lb}(n/i)\rceil$$

满足 $S(n)=O(n)$。

3.31 精确计算以下和式：

$$S(n)=\sum_{i=1}^{n}\lceil\mathrm{lb}(n/i)\rceil。$$

可以假定 n 是 2 的幂。

3.32 斐波那契数 $F(n)$ 可以扩展到 n 的值为负数，定义如下：$F(n+2)=F(n+1)+F(n)$，且 $F(1)=1$，$F(0)=0$（例如，$F(-1)=1$，$F(-2)=-1$，如此等等）。令 $G(n)$ 定义为 $F(-n)$。为 $G(n)$ 写一个递推关系，并尝试求解。

3.33 证明 $G(n)=(-1)^{n+1}F(n)$。

第 4 章　数据结构简介

科学不过就是经过训练与组织的常识。
——托马斯·亨利·赫胥黎（1878）

我憎恨理论家，他们居高临下，而我喜欢自下而上求索。
——弗兰克·洛伊德·莱特（1869–1959）

4.1　引言

数据结构是构造计算机算法的基础。算法设计类似于建筑设计，建筑设计师必须采用最有效、最符合建筑物用途的方式将房间整合在一起，因此，仅了解建筑物的功能、效率、形式和美观度是远远不够的，他还必须对建筑物的构建技术有着全面的认识。可以设计达到预期效果的建筑物，但我们无法建造空中楼阁，奇特的构思或许可以实现，但成本可能会高得惊人。同样，在设计算法时我们也必须对数据结构及其使用成本有充分的了解。

本章将以较短的篇幅介绍本书中用到的一些基本的数据结构。我们不打算详尽介绍每一种数据结构，因为这至少需要一整本书的篇幅，而且在这方面已经有许多优秀的教材。我们预计读者对数据结构已经有了一定程度的了解，因此本章只是迅速回顾一下这些知识罢了。

抽象数据类型是数据结构中一个非常有用的思想。一般在编写程序的时候需要说明具体的数据类型（如整型、实型、字符型等），但是在某些情况下，这些数据类型对算法设计而言却并不重要。举例来说，维护一个先进先出（FIFO）的队列需要两种操作，即在队列中插入一项以及删除一项。删除顺序必须与插入顺序相同。在设计这两种操作的算法时用抽象数据类型会显得更方便也更一般，我们只需要说明其操作，即把能进行上述运算的抽象数据类型称为 FIFO 队列。抽象数据类型中最重要的部分就是它能够支持的运算。另一个抽象数据类型的例子是具有优先级的项的队列。删除项的顺序并不是按照插入时的顺序，而是由其优先级决定的。这种抽象数据类型被称为优先队列。同样，对优先队列中的每一项，也无须说明其类型。（这里甚至连优先级的数据类型也不必说明，只须假设优先级是全序的，且我们能确定项的序。）

我们并不关心某个特定问题的具体实现，而是关注某个数据结构上操作的本质。这样就能把算法的设计变得更一般，因而在实现上，比方说一个优先队列的实现，可以基本上与具体的数据类型无关。如果我们认为可以借用某个抽象数据类型来解决问题，那就直接拿过来用好了。引入抽象数据类型可使算法设计的过程更加模块化。

4.2　基本数据结构

4.2.1　元素

本书将非特定类型的数据统称为元素。一个元素可以是一个整数、一个整数集、一个文本文件或者其他数据结构。当讨论的问题与具体类型无关时，我们就用这一术语来表达数据。

比如，我们考察排序算法。如果算法只涉及到元素之间的比较以及位置的移动，那么相同的算法可以用来对整数排序，也可以对名字（字符串）排序。算法的实现（程序）可能有细微的差别，但算法的思想是一致的。我们往往关心算法的思想而不是其实现，因此忽略元素的类型就顺理成章了。

我们对元素有如下假设：

1. 元素之间可以比较是否相等；
2. 元素所在的集合是全序的，因此可以比较一个元素是否“小于”另一个元素。我们一般不关心“小于”关系的确切定义，只要是全序的就行；
3. 元素可被复制。

以上任意一个操作都被认为是可以在一个单位时间内完成的，尽管这个单位时间与实际元素的大小有关，但一般还是把它当做单位时间。在大多数情况下，可以把这些元素看成整数，尽管算法可能对更复杂的数据结构依然是有效的。

4.2.2　数组

数组是一排相同类型的元素，其大小就是数组中元素的个数。数组的大小必须是固定的。由于大小固定，且其中元素类型也都相同，因此事先就可以知道应该为数组分配多少内存。比方说，数组元素是 8 个字符长的字符串，如果每个字符需要用 1 个字节来存储，数组的大小是 100，那么就需要用 800 个字节来存放这个数组。数组是连续存放的。如果数组的第一个字节在内存中的存储位置是 x，那么数组的第 k 个字节的存储位置就是 $x+k-1$。因此，很容易就能算出数组中每个元素存储的起始位置。在前面的例子中，如果数组的起始存储位置是 10000，那么第 55 个字符串就从第 433 个字节开始。如果地址是以字节计算的，那么其存储的起始位置就是 10432。（若地址的计算单位不同，只要稍稍修改一下上面的计算就可以。）

数组是非常高效又十分常见的数据结构。访问数组中的任何一个元素所花费的时间都是常数。用高级语言来设计算法的人基本上不必考虑数组存储位置的计算，这些通常都是由编译器完成的。一般常规的做法就是要尽量使用数组。当然数组也有缺点，也就是它的一些限制，即数组不能存放不同类型（或大小）的元素，数组的大小也不能动态变化。我们会在下面讨论这两点限制。

4.2.3　记录

记录和数组类似，只是记录中的元素类型不必相同。因此记录就是元素类型不同的线性序表，但元素类型确切的顺序是固定的。与数组一样，记录的存储大小也是事先可知的，访问记录中的任何一个元素所花费的时间也是常数。要做到这一点，就需要维护一个与记录中元素个数相同的数组，对记录中的每个元素，在数组中记录其起始位置。这样做就能保证访问记录中的任何一个元素所花费的时间都是常数，因为对每个元素只要去访问其在数组中对应的元素就能得到其起始地址。维护这样一个数组的程序是由编译器自动生成的。

比如说，某个记录如下：先有 2 个整数，然后是 3 个整型数组，每个数组有 20 个整数，再有 4 个整数，接着是两个 12 个字符长的字符串。（注意，记录中这两种类型的数组在这里被看做是记录中的元素。）记录的具体定义见图 4.1。那个与记录一起被维护着的数组中存储了记录中每个元素的相对起始位置。如果每个整数占 4 个字节，那么记录中的第 9 个元素 Int6 就是从第 261 个字节（2×4+3×20×4+3×4+1）开始存放的。既然已知记录中所有元素的大小，我们就可以在常数时间内得到每个元素的存储位置。与数组一样，记录也是连续存放的，同样不能动态地增加元素。

```
record examplel
begin
    lnt1:integer;
    lnt2:integer;
    Ar1:array [1..20] of integer;
    Ar2:array [1..20] of integer;
    Ar3:array[1..20] of integer;
    lnt3: integer;
    lnt4: integer;
    lnt5: integer;
    lnt6: integer;
    Name2: array [1..12] of character;
    Name2: array [1..12] of character
end
```

图 4.1 记录的定义

4.2.4 链表

有许多应用程序，在算法进行过程中，元素的数量也在动态地变化。固然可以定义一个足够大的数组（或记录）保证有足够的存储空间来存放这些元素，但是对最坏情况分配存储空间的做法显然不太高效（而且很多时候最坏情况是未知的）。另外，有时候需要在线性表中间插入和删除元素，如果用数组的话，那么在数组中间插入元素就需要移动其他的元素。如此低的效率显然与数组的连续表示有关，因此对较大的数组而言，在任意位置插入和删除元素的开销很大。这种情况下就可使用动态数据结构。本书中大量使用了动态数据结构，所以有必要熟悉一下它们。

链表是动态数据结构中最简单的形式。假设有一个元素列表，需要在其中高效地插入和删除元素。此时不应使用能连续存放元素的数组，而是单独地表示每个元素，元素之间用指针连接，即用指针来存放另一个元素的地址。链表就是由存储单元对组成的表，每一对由一个元素和指向下一对存储单元地址的指针组成。每对存储单元都是一个记录。链表能够通过指针寻址来扫描它的元素。这样的扫描是线性的，也就是说，不能直接访问链表上的每个元素，只能通过按序遍历链表进行访问。

用链表存储信息有两个主要缺点。首先，使用链表会耗费更大的空间，因为每个元素都有一个指针项。其次，比方说要访问第30个元素，那么就必须从表头开始逐个指针地访问过去，一共要访问 29 次指针项。而对数组来说，只需要简单计算一下，然后直接访问第 30 个元素。从另一方面来看，用链表也有一个很大的好处。假设我们已经访问了第 30 个元素，现在准备插入第 31 个元素[①]，那么只需要两个操作：把第 31 个新元素的指针指向先前的第31个元素（其地址存储在第 30 个元素的指针项），然后把第 30 个元素的指针指向第 31 个新元素（见图 4.2）；而用数组，则要移动第 30 个元素后的所有元素。删除链表中的元素也很简单，如果要删除第 31 个元素，只需要把第 30 个元素的指针指向第 32 个元素就可以，也就是把第 31 个元素的指针项复制给第 30 个元素的指针项（见图 4.3），由此只需要两步。

① 自然数系是连续的，因此说到第 31 个新元素和第 31 个原来的元素，很容易引起混淆。我们一般用“30a”来表示在第 30 个元素后插入的元素，但这种表示会导致许多问题。如果要在第 30 个元素后再插入一个元素，那么这种表示恐怕就无能为力了。（$30a_a$吗？）这个例子说明，对动态数据结构需要有更好的表示方法。

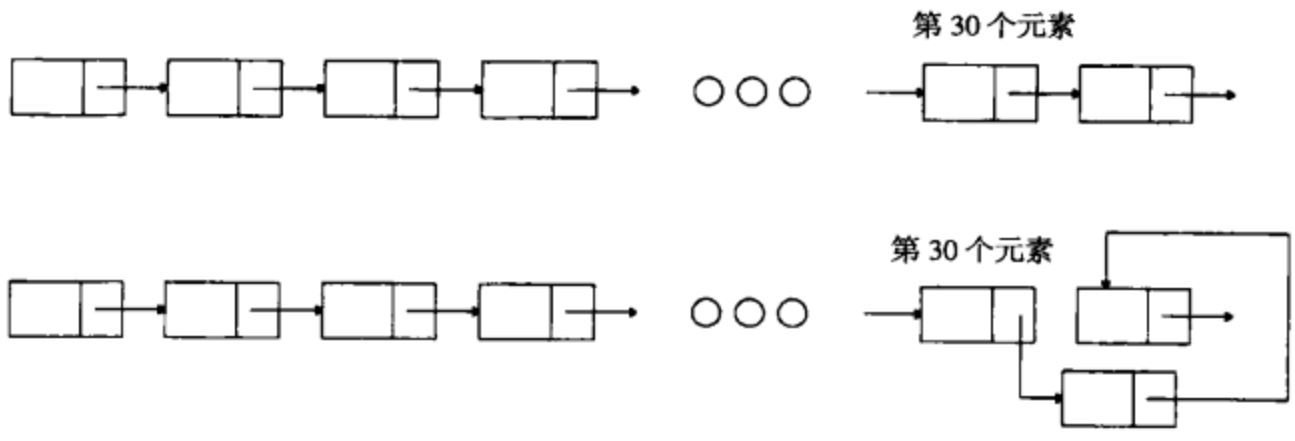


图 4.2　在链表中插入新元素

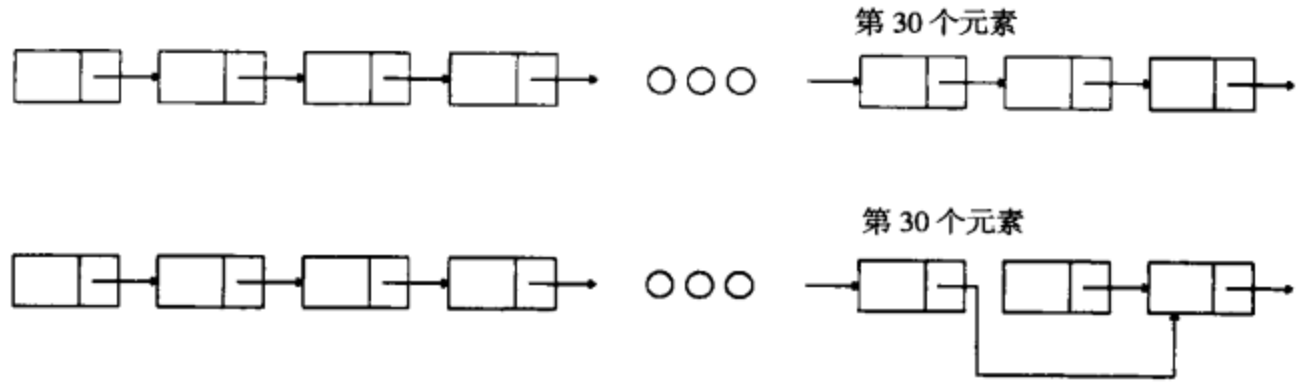


图 4.3　从链表中删除一个元素

在前面关于链表插入和删除元素的讨论中，没有涉及到一些重要的细节，这些细节问题会让链表在实现上更加复杂，比如应该如何判断是否到了链表的结尾。一般可用一个特殊的地址（即空指针，也就是一个不指向任何地址的指针）来表示链表的结尾；也可以用一个常规的记录，其关键字用来保证对链表的搜索在这个记录上停止。这个附加的记录有时也可叫做空记录，因为特殊元素少了，用它能使程序变得简单。对许多数据结构来说，空记录是很有用的。

4.3　树

数组和链表所能表示的结构只是元素的顺序，而许多应用程序需要使用更复杂的数据结构。树是一种能够表示层次结构的数据结构，用它能够更高效地实现对线性结构的某些操作。本节只讨论层次树，它也被称为根树或树结构。根树由元素以及通过某种特殊方式连接这些元素的边的集合（见图4.4）组成，这些元素一般被称为节点。其中（最上层的那个）节点是树的根，连接到根节点的那些节点是第一层节点，同样，连接到第一层节点的就是第二层节点，以次类推。所有这些连接，都是从某个节点连接到它的直接的（惟一的）“监护人”（借用家谱的说法，一般叫做父节点）。仅根节点无父节点。树的主要性质在于它没有环，也就是说，树上两个节点之间只有惟一的一条路。

一般一个节点同时连接到父节点及其下层节点（同样借用家谱的说法，把这些节点称为子节点）。一棵树的度就是子节点最多的节点的子节点数。一般将每个节点的子节点排序，然后根据顺序区分这些节点（如第一个、第二个）。度为 2 的树又称二叉树，子节点叫做左子节点（第一个）和右子节点（第二个）。没有子节点的节点又称叶子节点（这次借用真正的树的术语来命名）。树的高度是其最大的层数，也就是根节点到叶子节点之间的最大距离。每个节点都有一个来自全序集合的关键字（如实数、整数）。只要不引起歧义，既可以用节点的关键字，也可以用节点本身来表示该节点。为方便起见，一般假设节点的关键字各不相同，否则可以把相同关键字的节点

用链表连起来，然后用树上某个节点的指针指向这个链表。每个节点通常都有一个数据域，用来存放这个节点的数据（或者指向其数据的指针）。数据域的表示与具体应用程序有关，这里不做讨论。

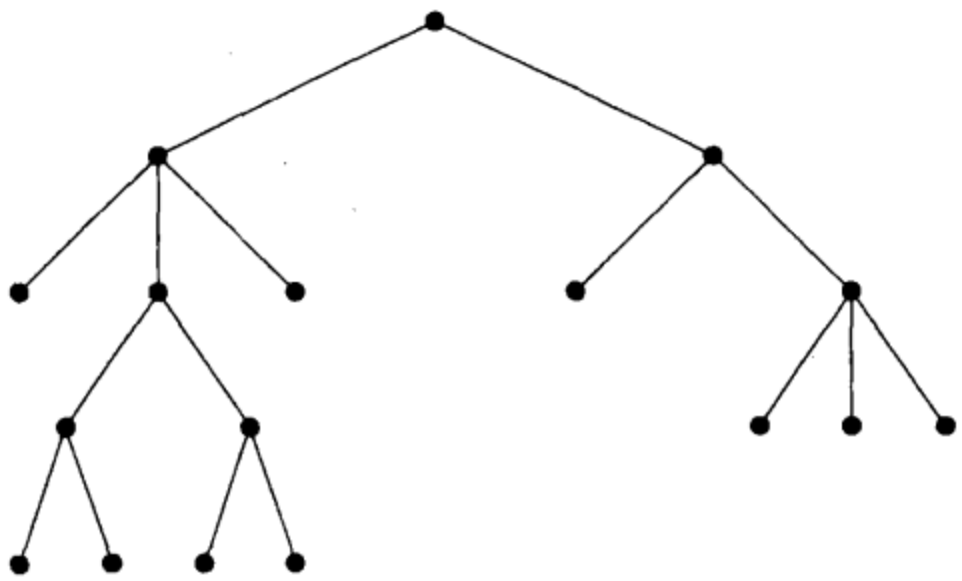


图 4.4 根树

本节将研究树的两种用途：搜索树和堆。对这两种情况我们都使用二叉树。下面就讨论树在内存中的表示。

4.3.1 树的表示

树有两种表示方法：显式表示和隐式表示。如果用显式表示，那么树的节点之间就用指针连接，于是有 k 个子节点的节点就需要一个有 k 个指针的数组（在某些程序中，节点还具有指向父节点的指针）。一般来说，所有节点都有相同的类型，这样处理起来比较方便。因此，任意一个节点都有 m 个指针，其中 m 是树上最大的子节点数。如果每个节点都只有两个指针，那么还可以这样做，即第一个指针指向第一个子节点，第二个指针指向其“兄弟”节点。图 4.5 是同一棵树的两种不同表示。第二种表示方法的主要缺点是，如果要得到某个节点所有的子节点，那么就不得不遍历整个链表。

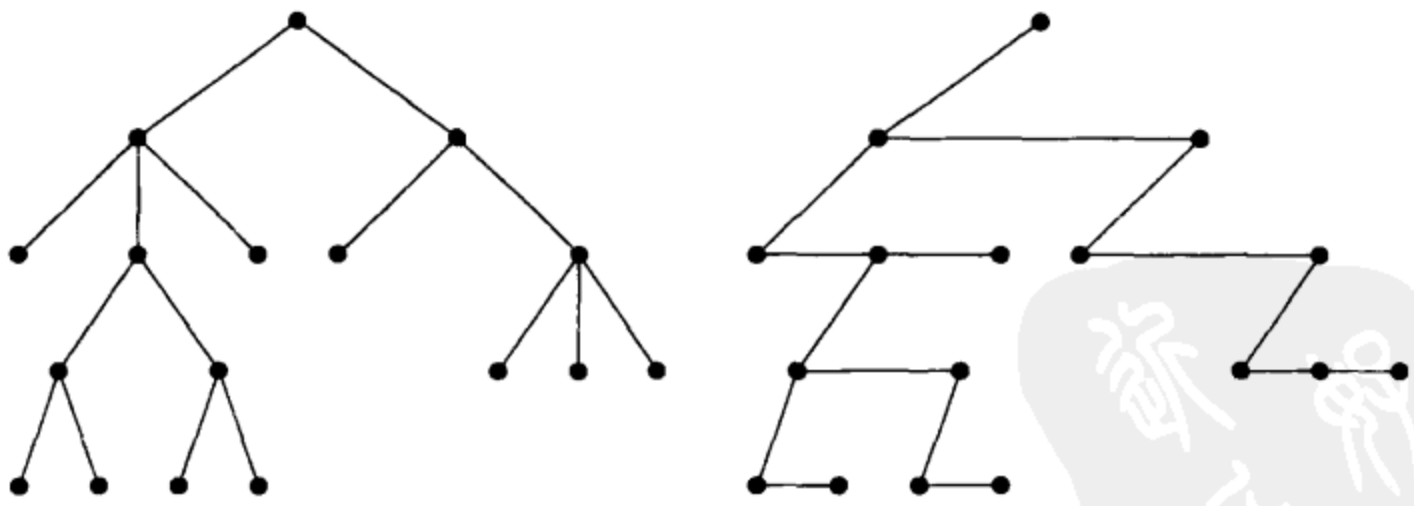


图 4.5 非二叉树的二叉表示

如果用隐式表示，那么就不需要指针。我们用一个数组来存放树上的所有节点，而节点之间的联系可以根据其在数组中的位置得到。下面介绍一种最常用的隐式表示。考察一棵二叉树 T，其根节点在 $A[1]$中。根节点的左、右子节点分别存放在 $A[2]$和 $A[3]$中。左子节点的两个子节点分别存放在 $A[4]$和 $A[5]$中，以次类推，所得到的这个数组就表示了对这棵树从左到右、按层遍历后

的结果。我们用数学归纳法来定义这种表示方法：（1）根节点存放在 $A[1]$中（这是归纳基础）。（2）如果节点 v 存放在 $A[i]$中，那么其左子节点就存放在 $A[2i]$中，其右子节点就存放在 $A[2i+1]$中。这种表示的好处就在于不需要指针，因而节省了存储空间。从另一方面来看，如果树是非平衡的，也就是说，某些叶子节点相对于其他叶子节点来说离根节点要远很多，那么用数组来存放这些节点就不得不表示大量的根本不存在的节点。图 4.6 就是一棵非平衡树，每个节点下的数字代表了它在数组中的位置，要表示这 8 个节点，需要用大小为 30 的数组。因此，隐式表示也未必能节省存储空间，要视树的具体情况而定。而且由于用的是数组，所以在树的中间进行动态运算会带来很大的代价。但是从另一方面来看，如果动态运算仅限于表示数组结尾的节点，那么用数组也未必不好。

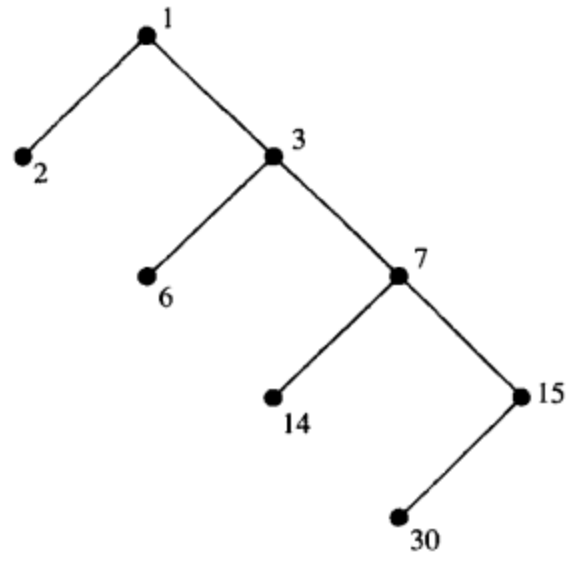


图 4.6　非平衡树的隐式表示

4.3.2　堆

堆是一种二叉树，其所有节点的关键字均满足如下性质：

任意一个节点的关键字都大于或等于其任意一个子节点的关键字。

由传递性原理可知，堆的上述性质说明了堆上任意一个节点的关键字都大于或等于其所有“后代”节点的关键字。堆在实现优先级队列的时候很有用，优先级队列是一种抽象数据类型，通过如下两个操作定义：

Insert(x)：把关键字 x 插入该数据结构。
Remove()：从该数据结构中删除最大的关键字。

堆既可用显式表示来实现，也可以用隐式表示来实现。我们要保证堆的平衡，因此用隐式表示来实现。假设数组 $A[1..k]$是其隐式表示，其中 k 是堆可能包含的元素个数的上限（如果上限不能确定，那么只能用链表）。把堆中目前元素的个数记做 n，也就是说，我们只关心 $A[1..n]$。下面就来描述如何用堆来实现对元素的高效插入和删除。

从 *Remove* 操作入手。根据堆的性质，堆中关键字最大的节点是根节点 $A[1]$，因此 *Remove* 总是删除根节点，现在的问题就是在根节点被删除之后如何调整堆。删除根节点之后的数组是 $A[2..n]$，对应着两个独立的堆。首先取叶子节点 $A[n]$，将它删除，然后把它放到原先根节点的位置。也就是说，令 $A[1] := A[n]$，于是 n 自减 1。新的 $A[1]$的值记为 x，得到的是两个独立的堆以及它们上方的某个值，但是该值未必满足堆的性质。（此时只有从根节点到原先 x 所在节点的路径

上的所有节点的值都是 x，堆的性质才可能满足。）为了满足堆的性质，我们现在把 x 向树的下方移动，直到它到达某棵子树，在这棵子树中 x 是最大值。移动的方式则是通过比较 x 和它的两个子节点（$A[2]$和 $A[3]$）的值，如果 x 不是这三个数中最大的，则把 $A[1]$与它们中间数值最大的那个节点交换。设 $A[2]$就是那个数值最大的节点，那么显然它也就是整个堆中数值最大的，因此以之为根节点。且此时以 $A[3]$为根节点的子树没有变化，仍然满足堆的性质。于是我们只须关注以 $A[2]$为根节点的子树（因为现在其根节点的值是 x）。现在我们可以用相同的方式对这棵子树进行同样的处理，假设目前处理到了第 i 步，$A[j]$中存放着 x。目前只有以 $A[j]$为根的子树不满足堆的性质。用上述方法比较 x 与其当前的两个子节点 $A[2j]$和 $A[2j+1]$（如果存在的话），如果 x 不是这三个数中最大的，则如上进行交换。当 x 成为某棵子树的节点的最大值或者 x 到达叶子节点时，算法终止。删除堆中元素的最大比较次数是 $2\lceil \mathrm{lb}n \rceil$，恰好是树的高度的 2 倍。图 4.7 给出了删除堆中最大元素的算法。

```
算法 Remove_Max_from_Heap(A, n)
输入：A（用来表示堆的大小为 n 的数组）
输出：Top_of_the_Heap（堆中最大的元素）、A（调整后的堆）和 n（调整后
      堆的大小；若 n = 0，则堆为空）
begin
   if n = 0 then print "the heap is empty"
   else
      Top_of_the_Heap := A[1] ;
      A[1] := A[n] ;
      n  = n - 1 ;
      parent := 1 ;
      child := 2 ;
      while chil  ≤ n - 1 do
      if A[child] < A[child+1] then
         child := child + 1 ;
      if A[child] > A[parent] then
         swap(A[parent], A[child]) ;
         parent := child ;
         child := 2*child ;
      else child := n {终止循环}
end
```

图 4.7　算法 *Remove_Max_from_Heap*

Insert 操作相似。首先将 n 自增 1，然后在叶子节点 $A[n]$插入新的值。接着比较这个新的叶子节点及其父节点的值，如果大于父节点的值，则交换这两个节点。这样一来，这个新值就是其子树的节点中值最大的（因为原来的父节点的值最大，而它的值大于父节点的值）。通过归纳，可以假设以 $A[j]$（最初是 $A[n]$）为根节点的树满足堆的性质，且如果去掉这棵树，剩下的部分也满足堆的性质。由此继续执行，将新节点继续往树的上方移动，直到其值不比父节点的大（或者直到它已经到了根节点的位置）。至此，得到的这棵树就完全满足堆的性质。在堆中插入元素的最大比较次数是$\lceil \mathrm{lb}n \rceil$，恰好就是树的高度。图 4.8 给出了在堆中插入一个元素的算法。

总的来说，我们可以以每个操作 $O(\log n)$的开销对堆中的元素进行插入和删除。而另一方面，在堆中进行其他操作则不太高效。比如，我们要在堆中寻找某个值的节点，这样堆的层次就没什么用了。堆很好地实现了一种抽象数据类型，对其中的一些操作十分高效。只要我们需要进行这些操作，就可将堆用于任何一种数据结构。

```
算法 Insert_to_Heap(A, n, x)
输入：  A（用来表示堆的大小为 n 的数组）以及 x（某个数）
输出：  A（调整后的堆）以及 n（调整后堆的大小）
begin
    n := n + 1 ; {假设数组不会越界}
    A [n] := x ;
    child := n ;
    parent := n div 2 ;
    while parent ⩾ 1 do
        if A[parent] < A[child] then
            swap(A[parent], A[child]) ; {参见习题 4.6}
            child := parent ;
            parent := parent div 2 ;
        else parent := 0 {终止循环}
```

图 4.8　算法 *Insert_to_Heap*

4.3.3　二叉搜索树

二叉搜索树在下列操作中的效率很高：

search(*x*)：在数据结构中查找关键字 *x*，或者判定数据结构中无 *x*（为简单起见，可假设每个关键字最多出现一次）

insert(*x*)：在数据结构中插入关键字 *x*（除非 *x* 已存在）

delete(*x*)：如果 *x* 存在，则在数据结构中删除 *x*

将能够进行上述三种操作的抽象数据结构称为字典。用二叉搜索树能够高效实现字典甚至其他更复杂的数据结构。对二叉搜索树而言，动态插入和删除是十分重要的操作，因此在本节中会使用树的显式表示。我们还希望元素的个数没有上限。假设树上的任意一个节点都至少包括三个数据域：*key*、*left* 和 *right*，使得 *key* 中存储节点的关键字（*left* 和 *right*）是指向其他节点（可以是空指针）的指针。二叉搜索树要比堆复杂，因为在堆中仅对叶子进行增加或删除而关键字随之改变，但在二叉搜索树中任何节点都能被删除，其指针也可以有其他用途。为简单起见，假设所有的关键字都不相同。

搜索

搜索树，顾名思义就是用来完成搜索的数据结构。一旦理解了整个搜索过程，那么这个数据结构就显而易见了。假设有一个关键字 *x*，我们想知道 *x* 目前是不是树上某个节点的关键字，如果是，那么请找到这个节点。这样的操作就叫做搜索。首先把 *x* 与树上的根节点的值（记做 *r*）进行比较；若 $x=r$，搜索完成；若 $x<r$，我们就从其左子节点开始搜索，否则就从其右子节点开始。搜索树上的每一个关键字都被用来划分其下关键字的取值范围：左子树中的关键字都比左子节点中的小，而右子树上的都比它大。这就是搜索树的定义。当所有关键字都满足这一条件时，就称这棵树是一致的。图 4.9 是一个对二叉搜索树进行搜索的简单的递归程序。

插入

二叉搜索树的插入操作同样很容易。给定欲插入的关键字 *x*，首先在树上查找 *x*，若树上已有 *x*，则找到 *x* 后，插入操作终止。（因为假设树上任意多个节点的关键字都不相同。）否则，搜索在某个叶子节点终止。这时就可以在该叶子节点下插入这个关键字（插入为左节点还是右节点，视

x 的值而定）。插入以后树仍然是一致的，因为如果搜索 x 的话，我们会到达刚才的叶子节点，然后通过它找到 x。但是要找到这个叶子节点，还要对搜索程序做一些细微的修改[①]。下面给出了一个非递归的搜索程序，如图 4.10 所示。

```
算法 BST_Search (root, x)
输入：root（指向二叉搜索树根节点的指针）以及 x（某个数）
输出：node（指向含有关键字 x 的节点的指针，如果上述节点
        不存在，则指向 nil）
begin
    if root = nil or root^.key = x then node := root
    {root^是 root 的指针所指向的记录}
    else
        if x < root^.key then BST_Search(root^.left, x)
        else BST_Search(root^.right, x)
end
```

图 4.9 算法 *BST_Search*

```
算法 BST_Insert (root, x)
输入：root（指向二叉搜索树根节点的指针）以及 x（某个数）
输出：通过插入由指针 child 指向的、关键字为 x 的节点而被改变了的树。
        如果已有节点关键字为 x，那么 child = nil
begin
    if root = nil then
        create a new node pointed to by child;
        root := child ;
        root^.key := x
    else
        node := root ;
        child := root ; {初始化 child 使其不为 nil }
        while node ≠ nil and child ≠ nil do
            if node^.key = x then child := nil
            else
                parent := node ;
                if x < node^.key then node := node^.left
                else node := node^.right ;
        if child ≠ nil then
            create a new node pointed to by child;
            child^.key := x ;
            child^.left := nil ; child^right := nil ;
            if x < parent^.key then parent^left := child
                else parent^.right := child
end
```

图 4.10 算法 *BST_Insert*

删除

删除操作通常要复杂一些。删除叶子节点比较容易，只需要把指向它的指针置为空。删除只有一个子节点的节点也不难，只需要把指向该节点的指针指向其子节点就可以。然而，如果要删

① 这句话好像是多余的。——译者注

除的节点有两个子节点，那么就要为这两个指针找一个归宿。令 *B* 就是那个要删除的、有两个子节点的节点（见图 4.11）。首先要交换 *B* 的关键字和另一个节点 *X* 的关键字，使得：（1）*X* 最多只有一个节点，（2）（在交换后）删除 *X* 仍能保证树是一致的。其次，删除 *X*，目前 *X* 中的关键字就是我们想要删除的 *B* 的关键字。删除 *X* 很容易是因为它最多只有一个子节点。为了保证树的一致性，*X* 的关键字至少必须与 *B* 的左子树中任何一个节点的关键字一样大，同时必须小于 *B* 的右子树中任何一个节点的关键字。注意，图 4.11 中 *X* 的关键字满足如下条件，即该关键字是 *B* 的左子树中关键字的最大者。在树中，我们称 *X* 是 *B* 的前驱。*X* 不能有右子节点，否则在那棵子树上就没有最大的关键字了。图 4.12 给出了删除操作的算法。

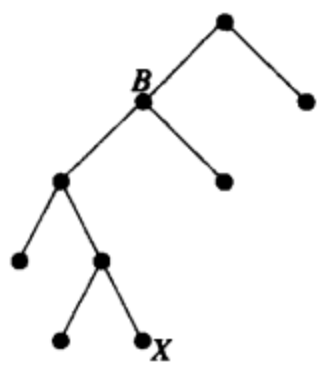


图 4.11　删除有两个子节点的节点

算法 ***BST_Delete*** (*root*, *x*)
输入： *root*（指向二叉搜索树根节点的指针）以及 *x*（某个数）
输出： 如果存在关键字为 *x* 的节点，则将其删除从而改变这棵树

```
{假设永远不会删除根节点，且任意两个节点都不相同。}
begin
    node := root ;
    while node ≠ nil and node^.key ≠ x do
        parent := node ;
        if x < node^.key then node := node^.left
        else node := node^.right ;
    if node = nil then print("x is not in the tree") ; halt ;
    if node ≠ root then
        if node^.left = nil then
            if x⩽parent^.key then
                parent^.left := node^.right
            else parent^.right := node^.right
        else if node^.right = nil then
            if x⩽parent^.key then
                parent^.left := node^.left
            else parent^.right := node^.left
        else {两个子节点的情况}
            node1 := node^.left ;
            parent1 := node ;
            while node1^.right ≠ nil do
                parent1 := node1 ;
                node1 := node1^.right ;
             {下面开始做真正的删除}
            parent1^.right := node1^.left ;
            node^.key := node1^.key
end
```

图 4.12　算法 *BST_Delete*

复杂度 搜索、插入和删除操作的执行次数取决于树的形状以及相应节点的位置。在最坏情况下，搜索路径会一直到达树的底部。算法中其他步骤（如实际插入动作、删除操作中关键字的交换等）可以在常数时间内完成。因此，最坏情况下的运行时间就是从根节点到叶子节点的最长路径，也就是树的高度。如果树是平衡的（下面将定义“平衡”），那么树的高度就近似等于 lb$n$，其中 n 是树上的节点数。在这种情况下，所有操作都是高效的。而如果树不是平衡的，那么这些操作的性能就不那么高了。

如果以随机顺序把关键字插入二叉搜索树，那么树的期望高度就是 $O(\log n)$，更确切地说，应该是 $2\ln n$。这样的话，搜索和插入操作都是高效的。而在最坏情况下，树的高度可能是 n（这时树就成了一个简单的链表）。例如，可以通过插入按照（或基本按照）大小顺序排列的关键字来构造有很长通路的树。另外，即使随机地删除节点，也可能出现问题。这主要是因为我们总是用其前驱节点来替换被删除掉的那个节点，这样做是不对称的。如果我们频繁地删除节点再紧接着插入节点，即使是随机的，也有可能形成一棵高度为$O(\sqrt{n})$的树。当然，这样的不对称是可以避免的，只要我们不总是选择前驱节点，而是交替选择前驱或后继（即其右子树上关键字最小的节点）进行。幸运的是，我们总是有办法避免在二叉搜索树上形成长通路。在下一节中将介绍其中的一种方法。

4.3.4 AVL 树

AVL 树（因 Adel'son-Vel'skii and Landis[1962]而得名）是第一个在最坏情况下保证以 $O(\log n)$ 的时间来进行搜索、插入和删除操作的数据结构（其中 n 是元素的个数）。AVL 树（以及其他可以在对数时间内完成操作的树结构）的主要思想就是在插入和删除的时候花一些时间来保持树的平衡，使得树的高度总是在 $O(\log n)$的范围内。花费在保持树平衡上的时间不能超过 $O(\log n)$，否则插入和删除操作的开销就太大了。我们想用一种比较容易维护的方式来定义树的平衡。

定义：AVL 树是一种二叉搜索树，每一个节点的左、右子树高度之差最多为 1（空树的高度定义为 0）。

如下面的定理所示，上述定义保证了树的最大高度为 $O(\log n)$。

□ **定理 4.1**

有 n 个内部节点且高度为 h 的 AVL 树满足：

$$h < 1.4404\,\mathrm{lb}(n+2) - 0.328。$$

证明：留做练习。 □

从这个定理可知，对 AVL 树的搜索只需要比较 $O(\log n)$次。下面的问题就是该如何进行插入和删除操作使其仍然具有 AVL 树的性质。我们先考虑插入操作，同样，假设所有的关键字都不相同。

令 x 是欲插入 AVL 树中的关键字。首先，如前所述，将 x 插入树的底部。如果插入后仍是一棵 AVL 树，那么就没问题。否则需要对树进行平衡，这将分 4 种情况，图 4.13 表示了其中的两种，剩两种与它们对称（右对称）。

图 4.13（a）中，新节点插入左子树后，B 的高度达到了 h+2，而 C 的高度还是 h。为了使插入后的树平衡，我们旋转这棵树：将 B 转到树顶，然后根据二叉搜索的性质调整树的其他部分（见图 4.14）。于是，以 B 为根节点的子树的高度为 h+2，与插入前的树的高度一样。而对于图 4.13（b）中的情况，旋转一个位置就不够了，需要旋转两个位置（见图 4.15）。同样，如果新子树与原来子树的高度相同，就不必再旋转。AVL 树的一个重要性质就是在插入节点之后旋转一次（一个或

者两个位置）仍能保持其性质。证明略。

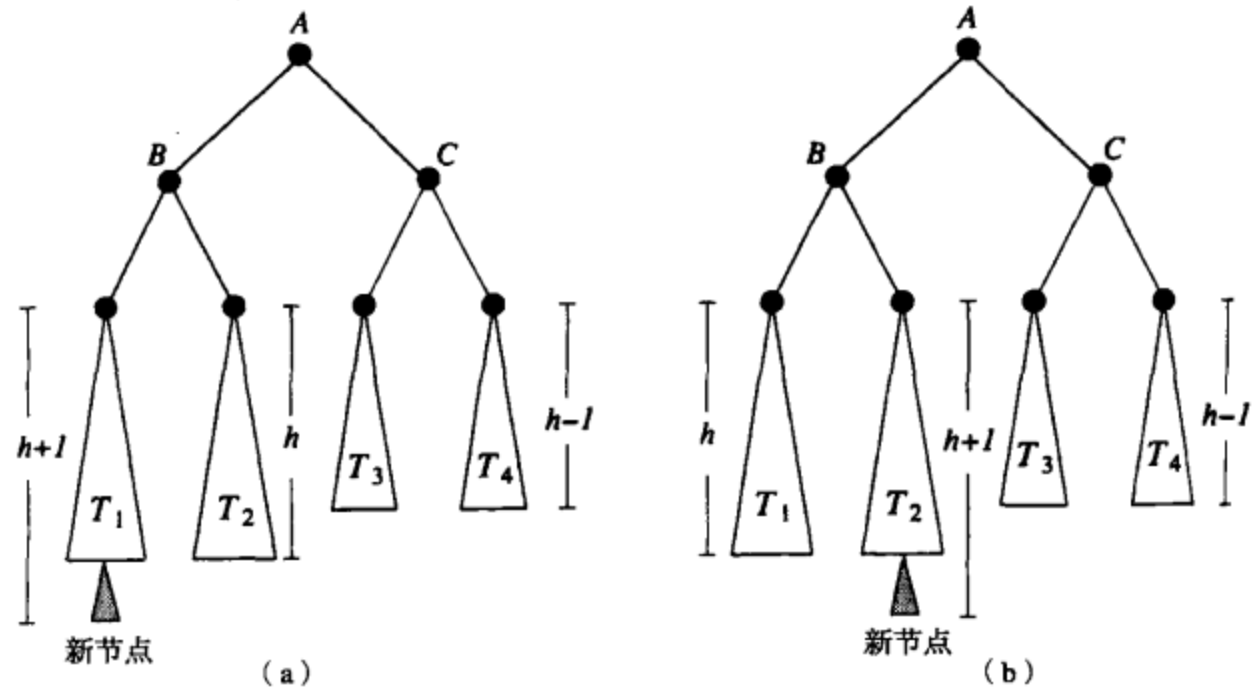


图 4.13　树在插入节点后不满足 AVL 性质

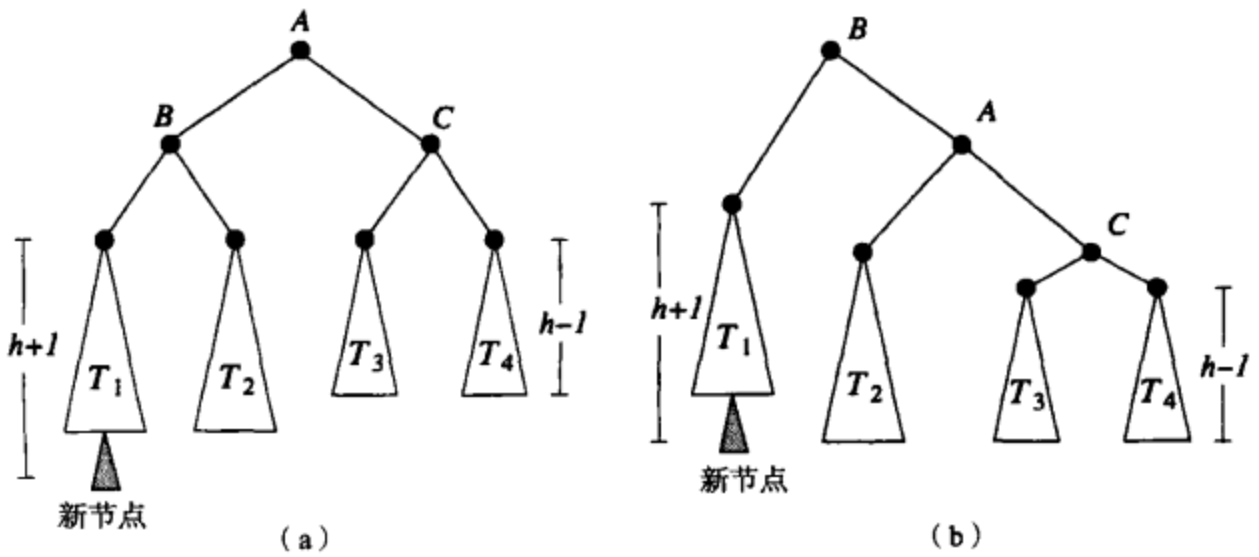


图 4.14　旋转一个位置：（a）旋转前；（b）旋转后

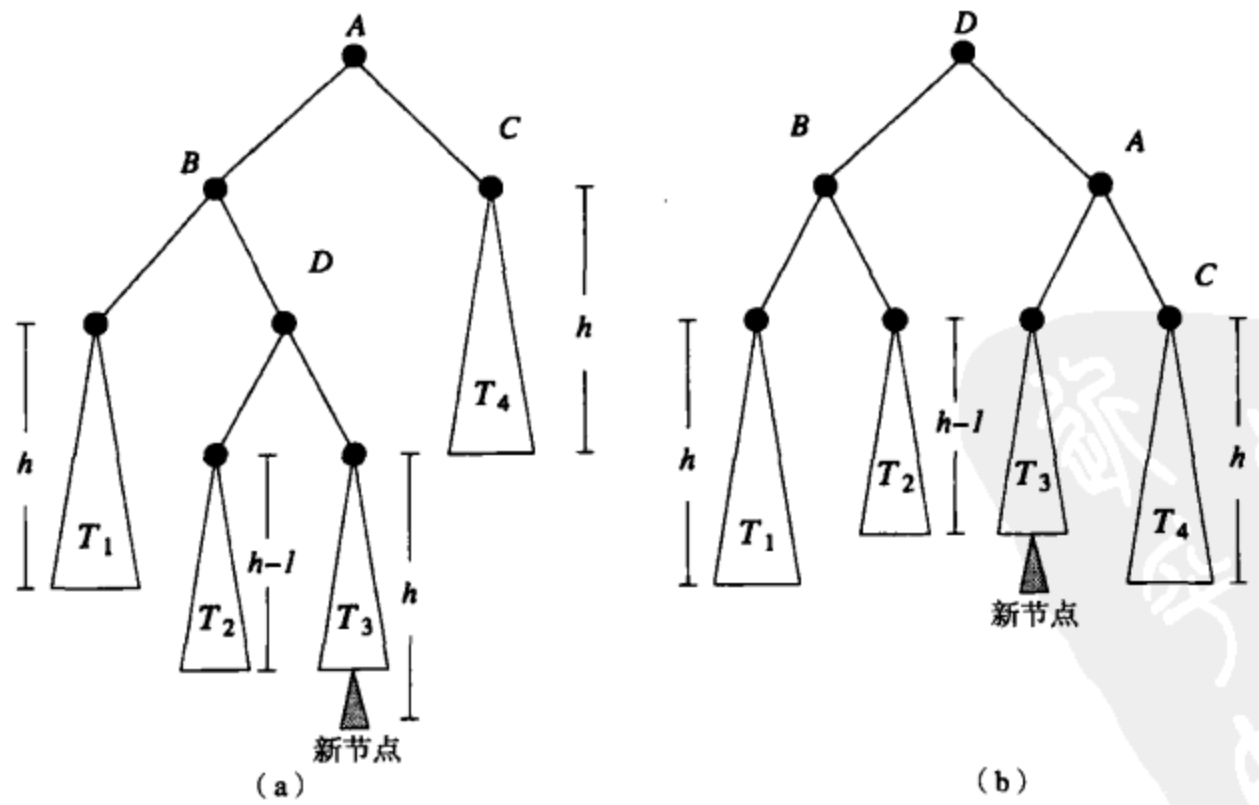


图 4.15　旋转两个位置：（a）旋转前；（b）旋转后

在上面的两个例子中，A 节点被称为关键节点。它是插入操作之后非 AVL 树的最小子树的根节点。因此在插入节点之前，必须找到关键节点，从而判断是哪一种情况。于是我们在每个节点中记录一个平衡因子，即该节点左、右子树的高度之差。对 AVL 树而言，任意一个节点的平衡因子只能是 1、–1 或 0。在子树上插入节点后，如果平衡因子不是 1 或者–1，则说明插入操作错误地增加了另一个方向子树的高度，此时就需要对树重新进行平衡，这就意味着关键节点的平衡因子一定非零。而且如果其下方的节点有非零的平衡因子，那么在树重新平衡之后，该节点的高度就应该与插入之前一样（回忆一下，平衡后关键节点高度不变）。因此，关键节点就是平衡因子非零的新节点的最低祖先节点。在自上而下的过程中，我们查看每一个平衡因子，并记住最后一个非零因子。当到达叶子节点时，就可以很容易地判断我们插入的方向是否正确。然后再执行一遍刚才的步骤（可以自底向上，也可以自顶向下，一般我们自底向上，这样经过的节点较少），重新调整平衡因子，需要的话，再做一次旋转。其中的细节就不一一详述。

删除操作通常比较复杂。删除节点后通过旋转一、两个位置将不再使树保持平衡。有时需要旋转 $O(\log n)$个位置，其中 n 是树的节点数。令人欣慰的是，每次旋转只需要常数步就可以完成，因此最坏情况下删除一个节点的开销仍为 $O(\log n)$。其中的细节也不一一详述。

评述 AVL 树是一种高效的数据结构。在最坏情况下它的开销也不大，比较次数只比最优树多 45%，甚至比平均情况下的还要好。过去的研究发现其平均搜索时间大概是比较 $\mathrm{lb}n+0.25$ 次（见 Knuth [1973], pp. 460）。AVL 树的主要缺点在于它需要额外的空间来存储平衡因子，而实现平衡因子的程序相当复杂。人们还提出了其他平衡搜索树的方案，比如 2-3 树、B 树、权平衡树、红黑树等。

4.4 散列

散列（hashing）是算法中最有用或者说是非常有用的一种数据结构，主要用于插入和搜索，其部分变形也可用于删除。散列的思想很简单，如果关键字从 1 到 n，我们可很容易地设计一个数据结构来存储这些数据，比如使用大小为 n 的数组，使关键字 i 存储在第 i 个位置。于是可以立即访问到任意一个关键字。如果在 1 到 $2n$ 中有 n 个不同的关键字，那么把这些关键字放在大小为 $2n$ 的数组中也不失为一个好办法，尽管存储空间仅有 50%的利用率浪费了一些空间，但我们可以立即访问到任意一个关键字。然而，如果这些关键字是从 1 到 M 的整数，其中 M 是某台计算机可以表示的最大整数，那么我们将无法分配一个大小为 M 的空间。举个例子，如果有 250 个学生，他们各自的社会保险号都不同，当然不会分配一个 10 亿大小的空间来存储他们的信息（假定社会保险号有 10 亿种可能），实际上我们只需要用到社保号的最后三个数字，也就是只需要一个有 1000 个元素的数组。但这不是一个保险的办法，因为有些学生社保号的最后三个数字是相同的（事实上，250 个学生发生冲突的概率是很高的）。下面讨论如何处理这样的冲突。当然可以用社保号的最后四个数字，或者最后三个数字加上头一个数字来减少冲突的概率，但是用更多的数字将意味着需要更大的表，其利用率也会降低。

假设从大小为 M 的很大的集合 U 中取出 n 个关键字，其中 M 远大于 n。我们想把这些关键字存储在大小为 m 的表中，使得 m 和 n 相差不大。这个思想就是用一个函数，一般称其为散列函数，来把从 1 到 M 的关键字映射到从 1 到 m 的新关键字上，从而可以用大小为 m 的数组来存储。对一个很大的整数取最后三位数就是一种这样的函数，它把大小为 10 亿的集合映射到大小为 1000 的集合上。我们会想办法把这些关键字存储到表中指定的位置上。如果这个函数很容易计算，那么我们也就能很容易地去访问这个关键字。然而，由于集合 U 很大，而表很小，所以无论用什么

函数都有可能将许多关键字对应到表中的同一位置。当两个关键字被映射到表上的同一位置时，我们将这种情况称为冲突，于是就会遇到两个问题：（1）如何找到一个冲突概率最小的散列函数，（2）如何处理冲突。

尽管集合 U 比表要大得多，但我们实际上要处理的关键字一般并不多。理想的散列函数对关键字在表中的映射保持均匀。当然，没有散列函数能够映射所有可能的关键字的集合而不发生任何冲突。如果 U 的大小是 M，散列表的大小是 m，那么就至少有 M/m 个关键字映射到了相同的地方。如果散列函数的映射是均匀的，那么每个位置就有大约 M/m 个关键字被映射过来。散列函数应该把关键字的集合均匀地映射到 1 到 m 之间的随机位置。均匀性和随机性是散列函数的本质。举例来说，我们不取社保号的最后三位，而是取学生出生年份的最后三位。显然这个散列函数不太理想，因为同年出生的学生发生冲突的可能性远高于社保号最后三位相同的可能性。

散列函数

设关键字都是整数且散列表的大小为 m，则一个简单且有效的散列函数就是 $h(x) = x \bmod m$，其中 m 是素数。如果散列表的大小调整为素数会比较困难（因为有时大小为 2 的幂更为方便），那么我们不妨用下面的这个函数：$h(x) = (x \bmod p) \bmod m$，其中 p 是素数且 $p > m$（p 应该比 m 大很多才有效，而 p 当然应该比 $|U|$ 小很多）。

前面已经提到，不存在对所有输入都很理想的散列函数。用上述素数方法已经是相当保险了，因为实际应用中大部分数据在结构上都与素数无关。另一方面，也有可能（尽管可能性不大）在某些应用程序中，会存储一系列形如 $r+kp$（r 为常数）的实验数据。当然，如果还用上述 p 的话，那么所有的数就不免映射到同一位置了。把数据映射到散列表的思想方法更进一步，用随机函数来决定使用哪一个散列函数。比如说，可以从某一范围的素数中随机选择 p，可是要找到很多素数并不容易。也可以这样做：随机选择两个数 a 和 b，使得 $a, b < p$ 且 $a \neq 0$，令 $h(x) = [(ax + b) \bmod p] \bmod m$。这个函数要比上面的那个算起来复杂，但其优点在于对所有的输入平均来说会比较理想。当然，对同一张表的访问只能用同一个散列函数，而很多时候我们有很多不相关的表，或者是频繁创建和删除的表。在这种情况下，每次创建不同的表，我们就可以用不一样的散列函数。上述函数还有一些我们想要的其他性质。

解决冲突

解决冲突最简单的方法就是用分离链接法（或链地址法，separate chaining）。散列表的每一个条目作为链表的表头，链表中的每一项都是被散列函数映射到该条目中的关键字。如果要访问某个关键字，只需要在相应的链表上进行线性搜索。插入新的关键字也只相当于在表头插入一个元素（但是必须搜索整个链表以保证关键字不重复）。如果有些链表太长，搜索的性能就不高了。只有散列表的大小与关键字的个数相比非常小，或者散列函数不理想的时候，才会造成链表过长。因此，散列法并不是一个很好的动态数据结构。准确估计关键字的数量是很重要的。分离链接法的主要问题是需要进行动态内存分配（即使关键字不太多，甚至不使用指针，另外，使用指针也会占用额外的空间）。从另一方面来说，如果由于某种原因错误地估计了散列表的大小，分离链接法仍然可行，而其他的静态方法就不起作用了。

另一个简单的方法叫做线性探测法。使用这种方法，表的大小是固定的，而且不用指针，由散列函数来决定关键字在表中的位置。若该位置已有关键字，即发生了冲突，那么就把新关键字存放在其后的第一个空位置。查找关键字也用这样的算法。（我们认为表是循环的，如果到了最后一个位置，且已被占用，则下一个位置就是表头。）因此，如果搜索不成功，则停在表头。当

表相对比较空的时候，这个方法较有效。而表相对比较满的时候，就会发生许多二次聚集，也就是不同散列函数值的关键字发生冲突。我们无法避免相同散列函数值的关键字之间的冲突，因为它们都映射到表的同一位置，但是可以避免二次聚集的发生。让我们看下面这个例子。设第 i 个位置已经满了，而第(i+1)个位置是空的。一个映射到第 i 个位置的新关键字将会与已有的关键字发生冲突，因此就把新关键字存储到第(i+1)个位置。这样做的效率很高，用很少的开销就解决了冲突。而如果现在又有一个映射到第(i+1)个位置的新关键字，就会出现二次聚集，那么就把它放到第(i+2)个位置（如果原来是空的）。可见，把映射到第 i 个位置的新关键字存储到第(i+1)个、第(i+2)个位置不仅会造成二次聚集，同样会让表的这一段空间变得更满，从而给以后带来更多的二次聚集。这种现象被称为集群（或簇，clustering）。当表快要满的时候，用线性探测法会频繁发生二次聚集从而退化为线性查找。

用线性探测法实现删除操作的效率不太高。如果欲插入的关键字“经过”一些位置被存储到一个空槽，而该关键字后来被删除了，那么以后再搜索就找不到了，因为会停留在那个空槽处。因此，如果需要删除，就必须采用基于指针的冲突解决方案。

我们可以用双重散列法来减少集群效应。当发生冲突时，计算另一个散列值 $h_2(x)$。此时就不按照线性顺序（即 i+1、i+2）进行查找，而是查找第($i+h_2(x)$)个、第($i+2h_2(x)$)个位置（以循环顺序）等。当另一个关键字 y 被映射到（比如说）$i+h_2(x)$ 时，下一次就应该查找 $i+h_2(x)+h_2(y)$，而不是 $i+2h_2(x)$。如果 $h_2(x)$ 和 $h_2(y)$ 不相关，那么就消除了集群效应。而我们在确定另一个散列函数的时候必须要小心，使得 $i+h_2(x), i+2h_2(x), \cdots, i+nh_2(x)$ 能覆盖整张表（$h_2(x)$的值与 n 互质就可以了）。

双重散列法的主要缺点是查找时需要更多的计算（即计算第二次散列值）。可以用以下方法避免多余的计算：选择第二次散列值的时候未必要与第一次的散列值完全无关，这样仍然可以减少集群效应。比如说，若 $h_1(x)=0$，则 $h_2(x)=1$；否则 $h_2(x)=m-h_1(x)$（假设 m 是素数且 $h_1(x)=x \bmod m$）。

4.5 合并–查找问题

合并–查找问题（union-find problem，又称等价问题）是用间接的数据结构来改善算法性能。问题如下：有 n 个元素 $x_1, x_2, \cdots, x_n$，分成若干组。初始时每个元素自成一组。对这些元素和组可以按任意顺序进行两种运算：

find (i): 返回包含 x_i 的组名
union(A, B): 把 A 组和 B 组合并成新的组（组名与现有的组名不同）

我们的目标就是要设计一种数据结构能够以尽可能好的性能来支持这两种运算的任意序列。

既然所有的元素事先都知道（且下标为 1 到 n），我们就可以为它们分配数组 $X[1..n]$。最直接的解决方案就是把含有第 i 个元素的组的标识存储在 $X[i]$中，这样一来 find 运算只需要访问一下数组即可，而 union 运算花费的时间长一些。假设将 union(A, B)的返回结果合并起来的某个组，称为 A，那么就要把所有包含 B 的项转换成 A 的。

下面就来介绍这个问题的另一种解法。这次简化 union 运算，而不是 find 运算。采用间接寻址，于是数组里的每一项都变成一个记录，每个记录由元素的标识和指向另一个记录的指针构成。一开始所有的指针都为空。我们通过让记录中指向 B 的指针改指向包含 A 的记录来进行 union(A, B)运算。几次 union 运算之后，数据结构就成为树的集合，如图 4.16 所示。每棵树对应着一个组，每个节点对应着一个元素。树上根节点的元素为组名。如果要查找包含元素 G 的组，只需要循着

G 的指针指向的路径搜索上去直到根节点为止，根节点的指针是空指针。如果要改变地址也是同样的过程，不必通知每一个节点，只要留下指向的地址即可。当然要查找正确的地址就没有先前那么容易了，也就是说，find 运算的效率不如先前那么高。如果 union 操作产生了很高的树，那么 find 运算的效率就特别低。

可见，高效合并–查找算法的思想就是要保持树的平衡，并控制树的高度。从刚才的讨论可知，花一些时间来保持树的平衡是值得的。考察图 4.16 中的 union(A, B)，可以发现有两种可能，既可以把 B 的指针指向 A，也可以把 A 的指针指向 B。显然，前一种方法得到的树更平衡。下面用形式化的方法来说明这个思想。在对应着根节点的每个记录中不仅存储组名，还存储其元素的个数。

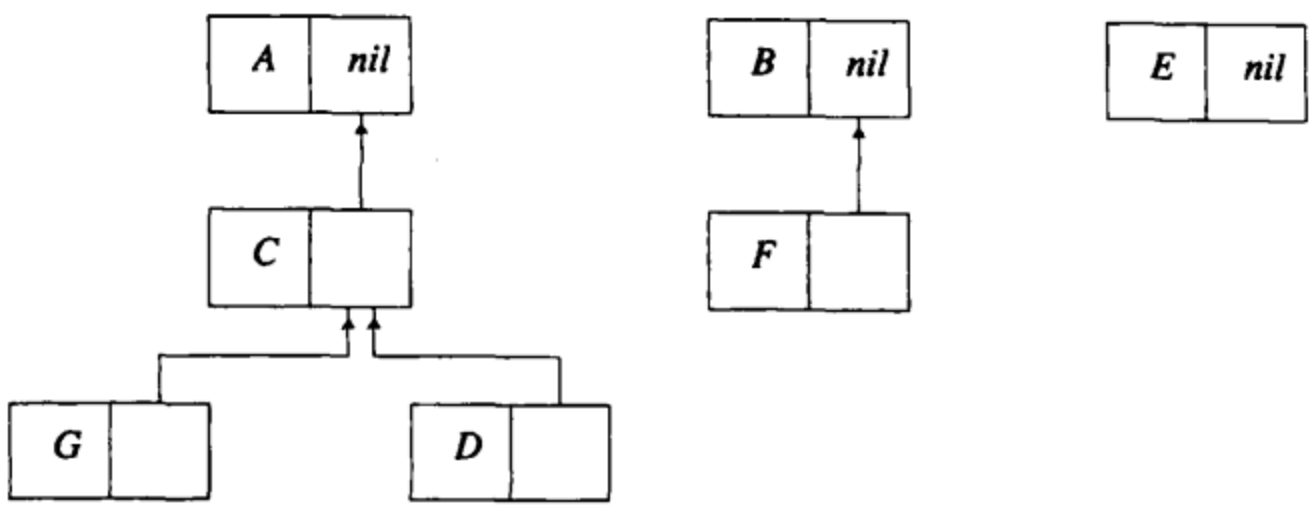


图 4.16　合并 – 查找问题的表示

平衡法：当进行 union 运算时，把较小的组的指针指向较大的组的记录（原来的关系随之中断）。重新计算合并后组的大小，然后存储到根节点的相应域中。

如果 union 运算采用平衡法，那么树的高度就永远不会超过 lb$n$，见下面的定理。

□ **定理 4.2**

如果采用平衡法，则任何一棵高为 h 的树至少包含 2^h 个元素。

证明：对 union 运算的次数进行归纳。显然定理对第一次 union 运算成立，生成有两个元素的高度为 1 的树。考察 union(A, B)，假设 A 在较大的组中，因此 B 会指向 A。分别把与 A 组和 B 组对应的树的高度记为 $h(A)$和 $h(B)$，于是合并后的树的高度就是 $h(A)$和 $h(B)$+1 中较大的那个。如果 $h(A)$较大，则合并后的树与 A 的树一样高，且有更多的节点。因此定理成立。否则，合并后的树中的元素至少是 B 的树中的元素个数的两倍（因假设 B 比 A 小），且其高度比 B 原来的高度大 1。同样，定理也成立。　□

从定理 4.2 可得，find 运算不会查找超过 lb$n$ 个指针，而 union 运算是常数时间的，因此 find 和 union 的任意运算序列 m（$m \geqslant n$）最多在 $O(m \log n)$步内完成。

也可以用以下方法来改进合并–查找数据结构的效率，比如信件转发。如果地址多次改变，那么信件就会从一个地址送到另一个地址，直至其到达最后的目的地。其实如果在半途把最后的目的地通知到所有的转发站，那么信件就会直接送达目的地。类似地，我们从某个记录开始遍历到达其根节点的所有指针后，只需要把路径上的那些指针直接指向根节点就行了（见图 4.17）。这种方法被称为路径压缩。再遍历一次整条路径只是让上述的步数翻一倍而已，因此 find 运算的渐近时间复杂度保持不变。所以每次进行 find 运算时，都可以采用路径压缩法。下面的定理给出了最坏情况下算法的复杂度下界，证明略。

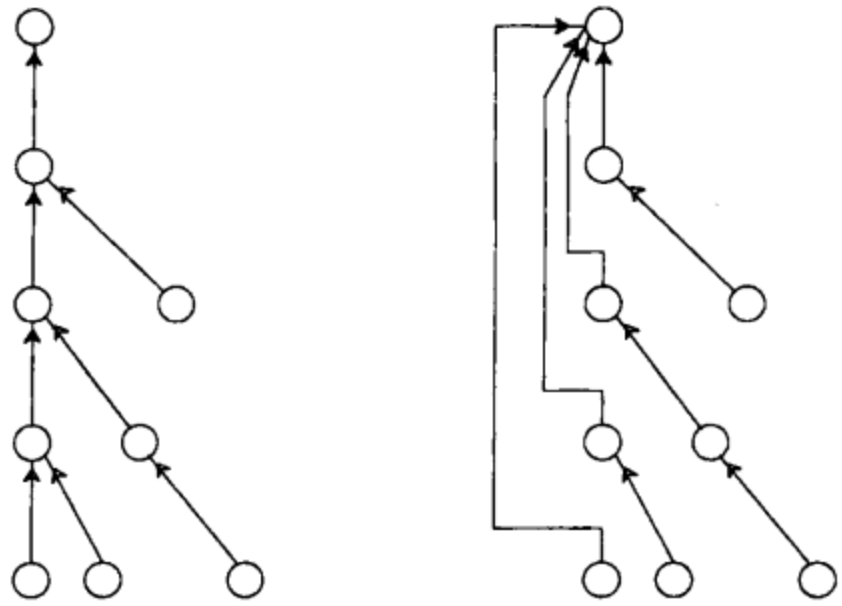


图 4.17 路径压缩：（a）压缩前；（b）压缩后

□ **定理 4.3**

如果同时采用平衡法和路径压缩法，对 $m(m \geqslant n)$的任意（可以是 find 也可以是 union）运算序列在最坏情况下需要的步数是 $O(m \log^* n)$，其中 $\log^* n$ 是迭代对数函数，定义如下。 □

函数 $\log^* n$ 递归定义如下，$\log^* 1 = \log^* 2 = 1$。对任意 $n > 2$，有 $\log^* n = 1 + \log^*(\lceil \mathrm{lb} n \rceil)$。比如 $\log^* 4 = 1 + \log^* 2 = 2$、$\log^* 14 = 1 + \log^* 4 = 3$以及 $\log^* 60000 = 1 + \log^* 16 = 4$。对于任意数 n（其中 $n \leqslant 2^{65536}$这样基本包括了各种目的的应用），我们有 $\log^* n \leqslant 5$。于是，union 和 find 的任意运算序列的复杂度都基本上是线性的（实际应用中就是线性的）。注意，某个 find 运算可能仍然需要 $O(\log n)$ 步，但是总体看来 $O(n)$ 个运算只需要 $O(n \log^* n)$ 步。这个例子很好地说明了什么是平摊分析，也就是把所有的步骤算在一起而不是分开来计算每一步。是否能针对这个问题设计线性时间的算法仍留待解决。

4.6 图

本书将用一整章的内容（参见第 7 章）来介绍图上的算法。本节只讨论用来表示图的数据结构。一个图 $G = (V, E)$由节点的集合 V 和边的集合 E 构成。每条边对应着一对节点，表示节点间的关系。例如，某幅图表示人的集合，其中连接着两个人的边可以表示互相认识的关系。图分为有向图和无向图。有向图的边是有序偶，即边的两个节点的顺序是很重要的。这样我们就可以把边画成从一个节点（箭羽）到另一个节点（箭矢）的箭头。无向图的边是无序偶。树就是一种简单的图。如果要指出树的层次，那么可以让所有的边都从根节点出发，这种树有时叫做有根树，因为只要知道根节点，就可以确定所有边的方向。同样我们也会考察无向图（有时也称自由树），它是没有层次的。

本书中我们会用两种方法来表示图。第一种表示是用图的邻接矩阵。令 $|V| = n$，则图 G 的邻接矩阵就是一个 $n \times n$ 的矩阵 A，使得 $a_{ij} = 1$ 当且仅当 $(v_i, v_j) \in E$。于是，矩阵的第 i 行就是一个大小为 n 的数组，其中若 v_i 到 v_j 有一条边，则数组的第 j 个位置为 1，否则为 0。邻接矩阵最大的缺点就是无论图中有多少条边，都需要 n^2 大小的空间来存放。例如，某棵树有 $n-1$ 条边，这些边

可以用每个节点的一个或两个指针来表示（取决于向上走还是向下走）。使用邻接矩阵，每个节点都对应着大小为 n 的数组，换句话说，如果边数很少，那么邻接矩阵中的大多数项都是空的。

我们无须一一表示出这些值为 0 的项，可以用链表把值为 1 的项（代表的是边）用链表连接起来，每条边都有一个指针。这就是第二种表示，又称邻接表。用邻接表时，每个节点上都关联着一张链表，这张链表包含了所有与这个节点相邻的边，表中边的顺序一般按照边的起始节点的编号排列。于是，整幅图就可以用链表数组来表示。数组中的每一项由节点的编号（或下标）以及指向其边链表的首地址的指针构成。如果图是静态的，也就是说不允许插入或者删除操作，那么可以用下面的方式表示链表。我们分配一个大小为 $|V|+|E|$ 的数组，前 $|V|$ 个项（按顺序）对应着节点。每一项都含有数组的下标，以这个节点为初始节点的边的链表以此为始。例如，有 20 个节点和 50 条边，其中有 4 条边以节点 1 为初始节点，所以第一项是 21（总是 $|V|+1$），第二项就是 25。每一项含有所有其对应边的初始节点。上例中，如果第二个节点的第二条边指向第五个节点，那么第 26 项就等于 5。这些边一般按序存放，尽管未必一定如此。图 4.18 为上述三种表示。邻接矩阵比邻接表用起来要简单，程序也相应比较简洁。而邻接表在当图上的边比较少的时候效率更高。实际情况下，大多数图的边数远远小于最大边数，有向图是 $n(n-1)/2$，无向图是 $n(n-1)$。因此，邻接表更常用。

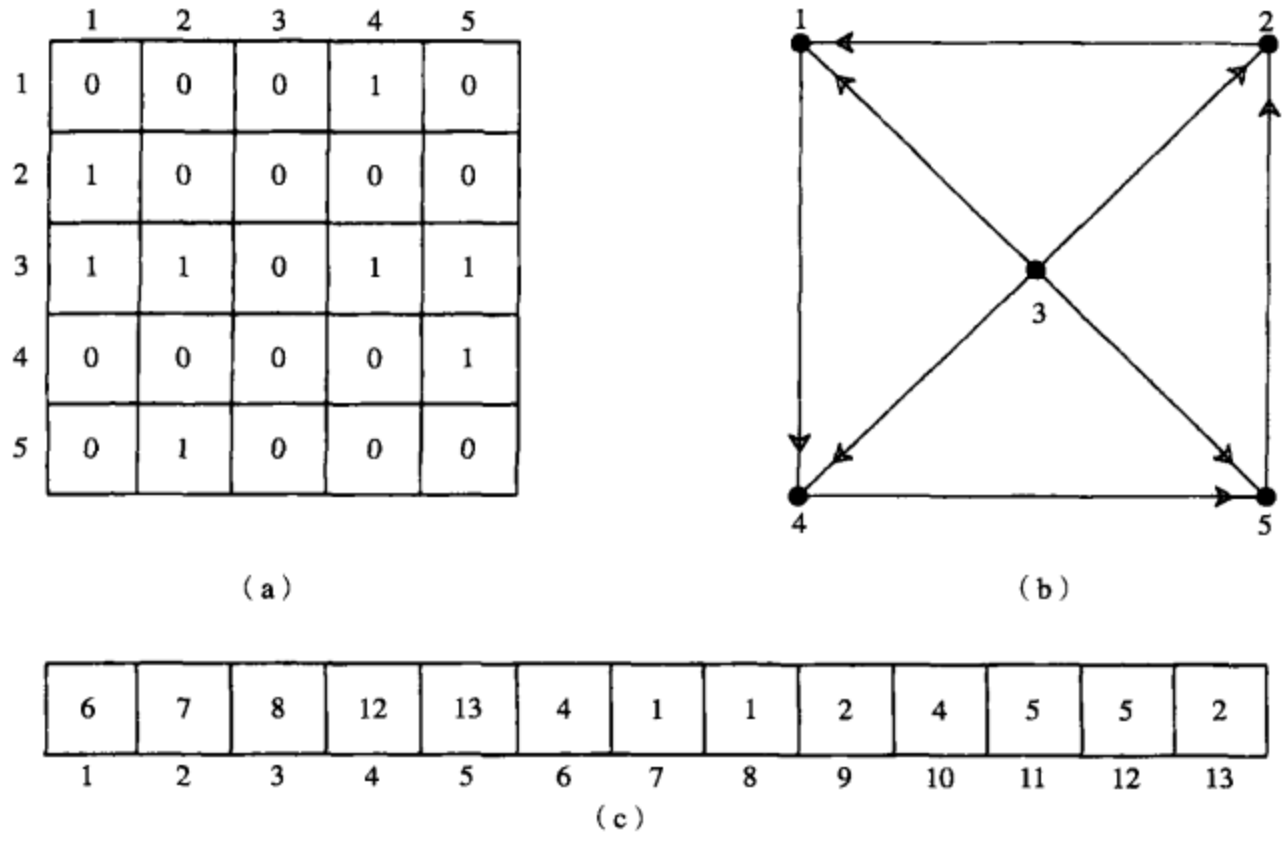


图 4.18　图的三种表示

4.7　小结

数据结构分为静态和动态两种。数组是静态数据结构。在使用数组前要先知道数组的大小（至少有一个上限），而且以后不能扩大，但是访问数组元素的效率却很高。链表是动态的，大小可以随意调整，能够满足任意大小的需要（当然也受当前可用内存的限制）。

数据结构也可分为一维结构和多维结构。数组和链表是一维的，它们只能表示元素之间可能形成的序列。树的结构就不止一维了，还能表达层次关系。图能表示更复杂的结构。当然，也可以构造多维数组和多维链表。

抽象数据类型这个概念很有用，我们只需要关心数据结构所需的运算，把依赖于具体数据类

型的实现细节推到以后再考虑。前面已经描述了字典、优先队列和合并–查找数据结构的实现细节。

如果存储的数据不依赖于特别的结构，那么散列将值得考虑。如果在访问元素时除了关键字外还须考虑其他因素，那么就不能用散列。例如，假如我们想要查找散列表中最小的关键字，此时将不得不扫描整张表。

参考文献和补充读物

数据结构是当前计算机学科教育中公认的基础内容，因此有许多这方面的书籍。Knuth [1973a]和 Knuth[1973b]这两本书详尽介绍了数据结构方面的知识。其他有关这方面的书还有 Standish[1980]、Aho, Hopcroft, and Ullman[1983]、Reingold and Hansen[1983]、Gonnet[1984]以及 Wirth[1986]。Tarjan[1983]是一本关于数据结构和算法的高级专著。

Jones[1986]中对实现优先队列的各种数据结构进行了比较。Jones 的文章中还详尽列出了关于优先队列的参考文献。Hibbard[1962]等文章介绍了二叉树插入和删除算法，它证明了 n 次随机插入之后平均路径长度为 2ln n。欲了解更多的信息，请参见 Knuth[1973b]。Eppinger[1983]完成了对二叉树随机进行插入、删除操作的实证研究，该文猜想平均路径长度有可能达到 $O(\log^3 n)$。Culberson[1985]中证明了在某些条件下随机插入、删除操作产生的平均路径长度为 $O(\sqrt{n})$。Baer and Schwab[1977]给出了不同平衡方案之间的比较。Knuth[1973b]和 Tarjan[1983]也描述了平衡树。Sleator and Tarjan[1985]给出了维护自调整树的几种新方法，其思想就是在每次访问之后把最近访问过的节点移到上层来调整这棵树。尽管树不一定每时每刻都平衡，但是平摊来看，性能就很不错。也就是说，其中的某个操作并不一定快，但是一段时间内每个操作的平均时间很短。

更多关于散列表的信息同样参见 Knuth[1973b]和 Gonnet[1984]。Vitter and Chen[1987]一书十分详尽地介绍了归并散列法。Carter and Wegman[1979]中描述了随机散列函数，又名通用散列函数。关于这个概念的一些有趣应用在 Wegman and Carter[1979]、Karlin and Upfal[1986]和 Kurtz and Manber[1987]中也能找到。还有一些允许表进行动态增长的扩展的散列方案，可参见 Fagin, Nievergelt, Pippenger, and Strong[1979]和 Litwin[1980]。

合并–查找所用的数据结构最初由 Galler and Fischer[1964]、Fischer[1972]和 Hopcroft and Ullman[1973]（得出了定理 4.3 提到的那个结果）等开始研究。Tarjan[1975]把运行时间提高到了 $O(m\alpha(m, n))$，其中 $\alpha(n)$是 Ackerman 函数的倒数，比 $\log^* n$ 增长得更慢。Tarjan and van Leeuwen[1984]研究了路径压缩的几个更简单的变形，同样能够取得相同的运行时间。欲知更多关于图的信息，请参见第 7 章及其参考文献。

操练型习题

4.1 写一个程序，从链表中删除一个元素。

4.2 写一个程序，倒转链表的方向，也就是使指针反向。

4.3 将二叉搜索树的简单递归搜索程序用非递归的方式实现。

4.4 设计一个算法将给定二叉搜索树上的关键字按序排列。

4.5 用数组 $A[1..16]$表示堆（用隐式表示）。若占用大小为 16 的数组，则堆中至少有多少个元素？

4.6 算法 *Insert_to_Heap* 可能在堆中多次交换元素。试修改一下算法使得最多只交换一次元素（仍然允许进行 $O(\log n)$次比较）。

4.7 假设我们用 AVL 树作为优先队列的数据结构，那么所有操作的复杂度是多少？

4.8 依次插入 1 到 20 的数字，会形成怎样的 AVL 树？

4.9　有一棵 AVL 树，删除其上的某个节点就会形成一棵非 AVL 树，而通过一次（一个或两个位置）旋转仍不能得到 AVL 树。画出这棵树，指出那个节点，并解释一下为什么得到的树不能通过一次旋转而平衡。

创造型习题

4.10　设计实现一个抽象数据类型，能支持下列操作：

Insert(x): 即使 x 已在数据结构中，也要进行插入操作。换句话说，数据结构支持重复的元素。
Remove(y): 从数据结构中删除任意一个元素，然后把它赋给 y。同样，任意元素都可以被删除。如果同一个元素有多份副本，那么只需要删除其中一份副本。

这种抽象数据类型又称为池（或着包）。它对存储工作很有用。产生的新工作被插入到池中；而当有工人可用时，将从池中删除一项工作。所有的操作都只需要 $O(1)$的时间。

4.11　用以下方法修改习题 4.10 中的数据类型，即池。假设每个元素在数据结构中只能出现一次，那么插入操作就必须检查是否有重复的元素存在。像上题那样实现这种数据类型，而且要加上重复元素检验的功能。在最坏情况下每个操作的复杂度是多少？对平均情况而言，怎样设计数据结构最好？

4.12　池这种数据类型的另一个变形如下：假设所有的元素都可以用 1 到 n 之间的整数来编号，且 n 足够小以至于我们可以分配大小为 $O(n)$的内存。每个元素最多只能出现一次。设计一个算法，其 *insert* 和 *remove* 操作（见习题 4.10 中定义）的时间复杂度为 $O(1)$。

4.13　设计如下算法：构造一个堆，其元素是另外两个大小分别为 m 和 n 的堆中的所有元素。堆用链表表示（每个节点有指针指向其两个子节点）。在最坏情况下算法的时间复杂度要求为 $O(\log(m+n))$。

4.14　设计如下算法：构造一个堆，包含 k 个堆中所有的元素。算法的复杂度是多少？

4.15　设计实现一个抽象数据类型，能支持下列操作：

Insert(x): 如果 x 不在数据结构中，则插入 x。
Delete(x): 从数据结构中删除 x（如果存在 x）。
Find_Next(x): 在数据结构中搜索比 x 大的最小关键字。

所有这些操作在最坏情况下的复杂度要求为 $O(\log n)$，其中 n 是数据结构中元素的个数。

4.16　设计实现一个抽象数据类型，能支持下列操作：

Insert(x): 如果 x 不在数据结构中，则插入 x。
Delete(x): 从数据结构中删除 x（如果存在 x）。
Find_Smallest(k): 在数据结构中搜索第 k 小的关键字。

所有这些操作在最坏情况下的复杂度要求为 $O(\log n)$，其中 n 是数据结构中元素的个数。

4.17　设计实现一个抽象数据类型，能支持下列操作：

Insert(x): 如果 x 不在数据结构中，则插入 x。
Delete(x): 从数据结构中删除 x（如果存在 x）。
Find_Next(x, k): 在数据结构中把比 x 大的关键字从小到大排序，搜索第 k 个这样的关键字。

所有这些操作在最坏情况下的复杂度要求为 $O(\log n)$，其中 n 是数据结构中元素的个数。

*4.18 回顾 4.3.4 节中的 AVL 算法，其中平衡因子取 1、0 或–1 三个值，需要 2 位。设计一种方法来实现这些算法（需要稍做改动），使得每个节点只多用 1 位。

4.19 有两个集合，其中一个集合中所有的关键字都小于另一个集合中的任意一个关键字。连接操作能把这两个集合合并起来。设计一个算法，将两棵二叉搜索树合并为一棵。最坏情况下要求时间复杂度为 $O(h)$，其中 h 是两棵树中高的那一棵的高度。

4.20 设计如下算法：（根据习题 4.19 的定义）将两棵 AVL 树合并为一棵。最坏情况下要求时间复杂度为 $O(h)$，其中 h 是两棵树中高的那一棵的高度。

4.21 考察一棵由随机插入和删除操作生成的 AVL 树。假设每个平衡因子出现的概率相同（即每个平衡因子的概率均为 1/3），证明从关键节点到插入位置的平均路径长度与树的大小无关。

4.22 按序插入数字 1 至 n 之后生成的 AVL 树的普遍结构是怎样的？这棵树的高度是多少？

4.23 试寻找一棵“最坏情况下的 AVL 树”，即构造一棵节点最少、高为 h 的 AVL 树。用这棵树来证明定理 4.1（见 4.3.4 节），考虑有 n 个节点的 AVL 树的最大高度。（提示：不妨递归地进行构造。）

4.24 令 T_1 和 T_2 是任意两棵有 n 个节点的树。证明：最多用 $2n$ 次旋转就能使 T_1 变成与 T_2 相等。

4.25 两个无向图 $G=(V, E)$ 和 $H=(U, F)$ 的连接指的是一个新的图 $J=(W, D)$，使得 $W=V \cup U$（即新图的节点包含原来两幅图所有的节点），$D=E \cup F \cup V \times U$（即新图的边包含原图所有的边再加上从 V 中的每个节点出发到 U 中每个节点的边）。设计一种理想的图的表示方式，以便能高效地进行连接操作。

4.26 令 $S=\{s_1, s_2, \cdots, s_m\}$ 是一个很大的集合，且 S 被分成 k 块。假设有一个函数 *which_block*，对已知的元素 s_i，*which_block*(s_i)等于包含 s_i 的块数；且具有常数时间复杂度（例如，S 是美国所有的街道地址，而块对应的则是邮政编码）。你需要维护 S 的一个小子集 T，对 T 有如下操作：

Insert(s_i)

Delete(s_i)

Delete_block(j): 删除 T 中所有属于块 j 的元素

一开始 T 是空的。最坏情况下每个操作花费的时间是 $O(\log n)$，其中 n 是目前 T 种元素的个数。*Delete_block* 只是从数据结构中删除元素（的关系），并不是从物理上把它们逐个删除。m 和 k 都很大，因此用大小为 m 或 k 的数组是不合算的。而 n 相对较小，可以用 $O(n)$的空间。

*4.27 令 $A[1..n]$是实数数组。设计一个算法，能够对下列两个运算的任意序列进行运算。

Add(i, y): 对第 i 个数加上 y。

Partial_sum(i): 返回前 i 个数的和 $\sum_{1}^{i} A[i]$。

注意，元素的数量保持不变（没有插入、删除操作），只可能改变元素的值。每个操作需要 $O(\log n)$步，允许多用一个大小为 n 的数组作为工作空间。

*4.28 扩展习题 4.27 中的数据结构，须支持插入和删除操作，其中每个元素有一个关键字和一个值，并可根据关键字访问元素，对值可以进行加法运算（但是访问元素只能通过关键字），而 *Partial_sum* 运算定义为：

Partial_sum(y): 返回集合中值比 y 小的那些元素的和 $\sum_{x_i<y} x_i$

对 $O(n)$个运算的任意序列，最坏情况下的时间复杂度仍须保证 $O(n \log n)$。

4.29 设计一个数据结构来维护一个元素集合，每个元素有一个关键字和一个值。该数据结构需要支持下列操作：

Find_value(x): 查找元素 x 的值（如果 x 不在集合中，则返回空）

Insert(x, y)

Delete(x)

Add(x, y): 为关键字为 x 的元素的值加上 y

Add_all(y): 对集合中所有元素都加上 y

这些操作中的每一个在最坏情况下的时间复杂度均为 $O(\log n)$。

4.30 （这是一个真实的故事。）一个名叫盖伊的程序员曾在使用新的编译器时遇到了一条错误信息，编译器提示在编译程序时内存耗尽了。他对此感到不解，因为他的程序并不需要很大的内存空间。他将问题定位到某条 case 语句，如下所示。没有这段 case 语句，编译就能顺利完成；而加上这段语句，编译器就提示内存耗尽。试判断编译器使用了怎样的数据结构从而引发该问题。（这段 case 语句正确有效；问题出在编译器不能正确地编译 case 语句。）

```
case i of
1:      Statement(1);
2:      Statement(2);
4:      Statement(3);
256:    Statement(4);
65535:  Statement(5);
```

第 5 章　基于归纳的算法设计

没有什么能比见到创造的源泉更为重要，我认为，它比创造本身更有意义。
——G. W. 莱布尼兹（1646–1716）

发明孕育发明。
——R. W. 爱默生（1803–1882）

5.1　引言

本章引入了一种新的算法设计方法，它类似于数学归纳法。我们将给出一些例子，并且讨论与该方法相关的基本原理和技巧。虽然归纳技巧在第 2 章已经讨论过了，但当需要的时候，我们还将重复使本章的内容可以自包含。

数学归纳法是基于**多米诺原理的**。想像有一串倒竖着的多米诺骨牌，我们希望通过仅推倒第一个骨牌就能把全部骨牌推倒。为了确定所有的多米诺骨牌都会倒下，只需要验证确实推倒了第一个，以及每一个骨牌倒下时都会使它的下一个倾倒。当增加一个额外的多米诺骨牌时，也不必倾倒全部的骨牌排列来验证所有多米诺骨牌都会倒下。相同的原理也适用于算法设计。

> 不需要以东拼西凑的方式来设计问题求解过程，只要保证以下两点就够了：（1）解决问题的一个小规模事例是可能的（基础事例），（2）每一个问题的解答都可由更小规模问题的解答构造出来（归纳步骤）。

这个原理首先要把问题简化成一个更小的问题（或者一组更小的问题）。麻烦的是通常并不容易找到一种方法来简化问题。本章将提供几个技巧来处理该过程。本章选择的例子并不是因为它们很重要（有些其应用非常有限），而是因为它们很简单并且能够说明我们想要强调的原理。纵观本书，我们会提供许多关于此方法的其他例子。

5.2　多项式求值

让我们从一个简单的代数问题开始：在给定的点上对一个多项式进行求值。

> **问题**　给定一串实数 $a_n, a_{n-1}, \cdots, a_1, a_0$，以及一个实数 x，计算多项式 $P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ 的值。

看起来归纳方法似乎并不是解决这个问题的自然选择，但是，下面将说明如何使用归纳方法来直接导出一个非常好的问题求解方案。让我们从最简单的（几乎平凡的）方式开始，然后寻找能导致更好结果的方法。

分析一下，这个问题包含 $n+2$ 个数，而归纳方法是根据较小规模问题的解来解决原问题的。换句话说，我们首先应该把问题简化成较小规模的问题，再进行递归求解，或者说，通过归纳的

方法来得到答案。一个自然的想法是通过去掉 a_n 来简化问题，由此得到的多项式计算问题是

$$P_{n-1}(x)=a_{n-1}x^{n-1}+a_{n-2}x^{n-2}+\cdots+a_1x+a_0。$$

除了减少了一个参数，这是一个相同的问题。因此，可以通过归纳求解。

归纳假设：已知如何在给定 $a_{n-1},\cdots,a_1,a_0$ 和点 x 的情况下求解多项式（即已知如何求解 $P_{n-1}(x)$）。

现在通过归纳法，利用假设来求解问题。首先，必须对基本情况进行求解，即计算 a_0，这是平凡的；然后，必须说明如何在较小规模问题答案（就是 $P_{n-1}(x)$的值）的辅助下，求解原始问题（计算 $P_n(x)$）。这一步只要简单地计算 x^n，将它乘以 a_n，再把结果与 $P_{n-1}(x)$相加即可：

$$P_n(x)=P_{n-1}(x)+a_nx^n。$$

在此，将归纳应用于这个问题的求解似乎是琐屑的，因为它仅仅是完成一个非常简单的计算，这是因为前面讨论的算法所表示的只是对多项式从右到左进行计算，稍后我们就会看到这个方法的威力了。

即使这个算法是正确的，它也并不高效，因为它需要 $n+n-1+n-2+\cdots+1=n(n+1)/2$ 次乘法和 n 次加法运算。现在将算法修改一下，以便能更好地使用归纳法进行求解。

一种改进是我们观察到有许多冗余的计算，即 x 的幂被到处计算。若计算 x^n 时可利用 x^{n-1} 的值，则可节省多次乘法运算。由此可把 x^k 的计算包含在归纳假设中得以改进：

更强的归纳假设：已知如何计算多项式 $P_{n-1}(x)$的值，也知道如何计算 x^{n-1}。

这个归纳假设更强大，因为它需要计算 x^{n-1}，但也更容易扩展（因为现在更容易计算 x^n）。计算 x^n 仅需要一次乘法，然后再用一次乘法得到 a_nx^n，最后用一次加法完成计算。（这个归纳假设也并不是太强，因为也需要计算 x^{n-1}。）总共需要 $2n$ 次乘法和 n 次加法。有趣的是，即使这个归纳假设需要更多的计算，它也有更少的工作总量（后面将回过头来讨论它）。这个算法看起来不错，它高效、简单并且易于执行。然而，还有更好的算法存在，下面通过另一种方式使用归纳法。

通过去掉最后的系数 a_n 来简化问题，是一个直接的步骤，但它不是惟一可能的简化方法，也可以去掉第一个系数 a_0，则更小规模的问题变成了计算由系数 $a_n,a_{n-1},\cdots,a_1$ 表达的多项式，即

$$P'_{n-1}(x)=a_nx^{n-1}+a_{n-1}x^{n-2}+\cdots+a_1。$$

（注意，现在 a_n 是第$(n-1)$个系数，a_{n-1} 是第$(n-2)$个系数，如此等等。）所以有新的归纳假设。

归纳假设（翻转了顺序）：已知如何计算由系数 $a_n,a_{n-1},\cdots,a_1$ 表达的多项式在 x 上的值（即我们知道如何计算 $P'_{n-1}(x)$）。

这个假设更适合我们的目的，因为它更容易扩展。显然，$P_n(x)=x\cdot P'_{n-1}(x)+a_0$。所以，从 $P'_{n-1}(x)$ 计算 $P_n(x)$仅需要一次乘法和一次加法。完整的算法可用如下表达式来说明：

$$a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0=((\cdots((a_nx+a_{n-1})x+a_{n-2})\cdots)x+a_1)x+a_0。$$

为了纪念英国数学家 W.G. Horner，这个算法又称 Horner 规则。（它也曾被 Newton 提到过，参见[Knuth 1981], p467。）对多项式求值的程序在图 5.1 中给出。

```
算法 Polynomial_Evaluation (ā, x)
输入: ā = a_0, a_1, a_2, ..., a_n（多项式的系数），x（实数）
输出: P（多项式在 x 上的值）
begin
    P := a_n;
    for i := 1 to n do
        P := x * P + a_{n-i}
end
```

图 5.1 算法 *Polynomial_Evaluation*

复杂度 该算法仅需要 n 次乘法和 n 次加法，以及一个额外的存储空间。因此即使已知的解答方法看起来非常简单且高效，但进一步寻找更好的算法有时也是值得的。这里，新的算法不仅比前面的快，而且它对应的程序也更简单。

评述 归纳法允许我们把精力集中在如何将一个较小规模问题的解扩展到大规模的问题。假设想求解 $P(n)$，即一个依赖于参数 n（通常为规模）的问题 P，我们可从任意的$P(n)$开始，利用已知 $P(n-1)$的解的假设来求解 $P(n)$。定义归纳假设有多种可能的方法，如何利用它也有许多方式。本书将考察其中的一些，并说明它们在设计算法时的威力。

这个简单例子阐明了使用归纳法的灵活性。导出 Horner 规则的窍门是（很少见的）从左到右考虑问题的输入，而不是直觉上的从右到左。另一个常见的可能是对比自上而下与自下而上（当包含一棵树结构时）。当然可能性也并不只这几种。再者说，有时最好的归纳顺序并不是对所有的输入都适用。设计一个算法时，寻找最好的归约方式是值得的。下面将看到所有这些可能性的例子。

5.3 最大导出子图

考虑下面的问题。你正在安排来自不同领域的科学家参加一个会议，并有一份人员名单。假定名单中的每一个人都同意到达并且有充足的机会彼此交换意见，而且每一位科学家都已经写下了他愿意与其进行交流的科学家的名字。现在你尽可能多地邀请名单上的人，但是希望保证每个人都至少有其他 k 个人愿意与他交流（k 是一个固定的数，独立于被邀请者的数目）。不必安排交流的过程；特别是你无须考虑是否有足够的时间让他们碰面。你只是希望把他们吸引到这个会议上来。那么如何确定邀请的人员呢？这个问题与一个图论问题相对应。令 $G=(V, E)$ 是一个无向图。一个 G 的**导出子图**是图 $H=(U, F)$，满足 $U \subseteq V$ 且 U 中两顶点若在 E 中有边则该边也包含在 F 中。一个顶点的**度**是与该顶点相连结的其他顶点的个数。现将图中的顶点对应于科学家，如果两个科学家之间存在交流的可能，则两顶点相连，这样一个导出子图就对应于一个科学家子集。

> **问题** 给定一个无向图 $G=(V, E)$和一个整数 k，试找到 G 的最大规模的导出子图 $H=(U, F)$，其中 H 中所有顶点的度大于或等于 k，或者说明不存在这样的子图。

一种解决问题的直接方法是把度小于 k 的顶点删除。当顶点连同它们连接的边一起被删除后，其他顶点的度也可能会减少。当一个顶点的度变成小于 k 后它也会被删除。但是，删除的次序并不清楚。我们应该首先删除所有度小于 k 的顶点，然后再处理度减少了的顶点，还是应该先删除一个度小于 k 的顶点，然后继续处理剩下受影响的顶点呢？（这两种方法分别对应于广度优先搜索和深度优先搜索，这将在 7.3 节详细讨论。）这两种方法会导致相同的结果吗？最终的图是否就

是最大的导出子图呢？这些问题都很容易回答，下面将要描述的方法甚至会使它们更容易得到解答。

在将算法分解成计算步骤之前，我们先证明一个定理来说明这样的算法确实存在。但这里不打算做一个正式的证明（至少在初期阶段不这样做）。为了获得对问题的深入理解，求解思路是想像定理证明时的步骤。我们的目标是找到满足给定条件的最大导出子图，下面是一个归纳的“证明”。

> **归纳假设**：假设在顶点数目小于 n 时，我们已知如何找到所有顶点的度大于或等于 k 的最大导出子图。

现在需要证明这个定理在基础情况下为真，并且在 n–1 时为真的情况下蕴含 n 情况下为真。一个非平凡的情况出现在 $n = k+1$，因为如果 $n \leqslant k$，则所有的度小于 k。若 $n = k+1$，则可以注意到，所有的度等于 k 的惟一方式是一个**完全图**（即所有的顶点之间都相连）。所以，现在假定 $G = (V, E)$ 是一个顶点数 $n > k+1$ 的图。如果所有的顶点的度大于或等于 k，则整个图满足要求，即为所求；否则，存在一个顶点 v，其度小于 k。显然，在任何 G 的子图中，v 的度都小于 k；因此，v 不属于任何满足该问题条件的子图。所以，我们把 v 及其相连的边删除，不影响定理的条件。当 v 被删除后，图中只有 n–1 个顶点，由归纳假设，可知如何对该问题求解。

现在已经完成了问题的求解任务。问题的算法和答案就很清楚了。任何度小于 k 的顶点都可被删除。删除的次序并不重要。这种删除是必需的，而删除后剩下的图必定是最大的，同样，算法的正确性也很显然，因为我们是在证明算法正确性的过程中设计了算法。

评述　简化问题的最好方法是消除问题的一些组成元素。在本例中，归纳的使用是直接的，主要是因为要删除的顶点以及如何进行删除都很明显，因此简化过程是直接的。然而，通常的消除过程可能并不那么直接。下面将看到把两个元素合并为一个的例子，从而使元素数目减少（参见 6.6 节）；有些例子把消除限制在问题本身上，而不是针对输入（参见 7.7 节）；还有些例子是通过设计特别的算法来寻找能被消除的元素（参见 5.5 节）。另一个对右边的元素进行消除的例子将在下面介绍。有趣的是，如果把问题描述中的“$\geqslant$”替换成“$\leqslant$”（即寻找一个最大导出子图，其所有顶点的度至多为 k），那么这个问题就会变得困难得多（见习题 11.12）。

5.4　寻找一对一映射

令 f 是一个把有限集映射到自身的函数（即 A 中的每一个元素被映射到 A 中的另一个元素）。为了简单起见，用整数 1 到 n 来表示 A 中的元素，假定函数 f 用数组 $f[1..n]$来表示，$f[i]$中占据的是 $f(i)$的值（这是一个介于 1 到 n 的整数）。如果对于每一个元素 j，至多存在一个元素 i 映射到 j，则称 f 是一个一对一的函数。函数 f 可以用图的形式来表示，如图 5.2 所示，两侧都对应同样的集合，中间的边表示映射。图 5.2 中的函数显然不是一个一对一的函数。

> **问题**　给定一个集合 A 和一个从 A 到自身的映射 f，寻找元素个数最多的一个子集 $S \subseteq A$，S 满足：(1) f 把 S 中的每一个元素映射到 S 中的另一元素（即 f 把 S 映射到它自身），(2) S 中没有两个元素映射到相同的元素（即 f 在 S 上是一个一对一函数）。

如果 f 原来就是一对一的，则整个集合 A 满足问题的条件，且 A 也必然是最大的。另一方面，如果对于某些 $i \neq j$ 有 $f(i) = f(j)$，则 S 不可能同时包含 i 和 j。例如，在图 5.2 中满足问题要求的 S 不能同时包含 2 和 3，因为 $f(2) = f(3) = 1$。对于它们之间要消除哪一个，其选择并不是任意的。例如，设想我们决定消除 3。由于 1 映射到 3，所以必须同时消除 1（因为这个映射必须在 S 中，

但现在 3 已经不在 S 里）。然而如果 1 被消除了，则 2 也必须同时被消除（基于相同的原因）。但是这样的子集不是最大的，例如，容易看到可以只单独消除 2（图 5.2 的解是子集{1, 3, 5}）。这个问题是要找一种通用的方法，来决定包含哪些元素。

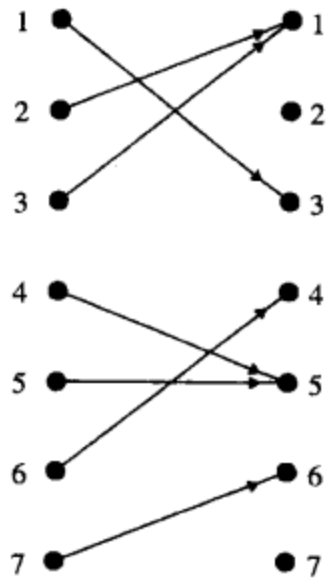


图 5.2 从一个集合到自身的映射（两侧是相同的集合）

幸运的是，在决定如何把问题简化到小问题上有一些灵活性，比如可以通过寻找一个元素是否属于 S 来减小问题的规模。我们将在后面进行这项工作，这里直接使用如下归纳假设。

归纳假设：对于包含 n–1 个元素的集合，如何求解问题是已知的。

基础情况是平凡的：若集合仅含一个元素，则它必定映射到自己，这就是一个一对一的映射。假定有一个包含 n 个元素的集合 A，并且要寻找一个满足问题条件的子集 S。我们断言，任何没有被其他元素映射到的元素 i，不可能属于 S。（换句话说，任何图中右侧的元素 i，如果没有被任何边相连，则它不可能在 S 中。）否则，如果 $i \in S$ 且 S 有 k 个元素，则这 k 个元素映射到至多 k–1 个元素上；从而，这个映射不可能是一对一的。如果存在这样的一个 i，则可简单地把它从集合中删除。现在得到集合 $A' = A-\{i\}$，其元素个数为 n–1，f 映射作用在上面；由归纳假设，我们已知对 A'如何求解。如果不存在这样一个 i，则映射是一对一的，即为所求，结束。

求解的本质是必须消除 i，而我们也证明了 i 不能属于 S。这就是归纳的力量：一旦我们去除了一个元素，减小了问题规模，就可以完成任务。然而，必须小心，简化过的问题（除了规模以外）和初始问题应该是同样的问题。集合 A 与函数 f 的惟一条件是 f 把 A 映射到自身。这个条件一直保留到集合 $A-\{i\}$，因为没有任何元素映射到 i。当没有元素可以被消除时，算法结束。

实现 我们把算法描述成一个递归过程。每一步寻找一个没有被其他元素映射的元素，并把它删除，然后继续递归。但是，在实现时并不一定需要递归，可以用一个计数器$c[i]$来记录在每一个元素 i。直觉上，$c[i]$等于映射到 i 的元素的数目。对于所有的 i，通过在 n 个步骤内扫描数组并递增相应的计数器，可以计算出 $c[i]$。然后把所有计数器为零的元素放进一个队列。在每一步里，把元素 j 从队列（以及集合）中删除，并递减 $c[f(j)]$，同时，如果 $c[f(j)] = 0$，那么就把 $f(j)$放进队列。当队列为空时，算法终止。该算法在图 5.3 中给出。

复杂度 初始化部分需要 $O(n)$个操作。每一个元素可以被放进队列至多一次，关于把元素从队列中删除的步骤需要常数时间。所以总共的步骤数还是 $O(n)$。

评述 这个例子通过从集合里消除元素来减小问题规模。所以，技术是试图找到最容易的方式来消除元素，而不改变问题的本质。惟一的要求是函数把 A 映射到它自身，所以选择那些没有

被其他元素映射的元素是很自然的。

```
算法 Algorithm Mapping ( f, n )
输入：  f（数组，其元素值为 1 到 n 的数值）
输出：  S（1 到 n 的整数集合的一个子集，f 是 S 上的一对一函数）
begin
    S := A; {A 是由 1 到 n 的整数组成的集合}
    for j := 1 to n do  c [j ] := 0;
    for j := 1 to n do  increment c [f [j ]];
    for j := 1 to n do
        if c [j ] = 0 then put j in Queue;
    while Queue is not empty do
        remove i from the top of the queue;
        S := S - {i};
        decrement c [f [i ]];
        if c [f [i ]] = 0 then put f [i ] in Queue
end
```

图 5.3　算法 *Mapping*

5.5　社会名流问题

接下来的这个例子是算法设计中的一个常见习题，这是一个不需要扫描所有数据（或者绝大部分）就能得到解答的很好的例子。在 n 个人中，一个被所有人知道但却不知道别人的人，被定义为社会名流。现在的问题是如果存在，试找出该社会名流。你可以使用的惟一方式是询问："对不起，请问你知道那个人吗？"（假定所有回答都正确，甚至这位社会名流也将回答。）我们的目标是将问题的数目最小化。由于有 $n(n-1)/2$ 对人，如果提问是随意的，那么在最坏情况下可能需要问 $n(n-1)$个问题。在最坏情况下，应该如何做并不那么明显。

可以使用图论来表示该问题，使用一个有向图，每个顶点表示一个人，如果 A 知道 B 则从 A 到 B 有一条边。一个社会名流对应于图中的**汇点**（在此无双关语之意）。一个汇点是一个入度为 $n-1$ 而出度为 0 的顶点。注意一个图中至多有一个汇点。问题的输入对应于一个 $n \times n$ 连结矩阵（如果第 i 个人知道第 j 个人，则 ij 项为 1，否则为 0。）

> **问题**　给定一个 $n \times n$ 连结矩阵，确定是否存在一个 i，其满足在第 i 列所有项（除了第 i_i 项）都为 1，并且第 i 行所有项（除了第 i_i 项）都为 0。

在只有两个人的基础情况，问题是简单的。与以往一样，我们要考察 $n-1$ 个人和 n 个人问题的差别。由归纳法，我们假定能够在 $n-1$ 个人中找到社会名流。由于至多只有一个社会名流，所以有三种可能：（1）社会名流在最初的 $n-1$ 人中，（2）社会名流是第 n 个人，（3）没有社会名流。第一种情况容易处理，我们仅需要检查第 n 个人是否知道这个社会名流，而这个社会名流是否不知道第 n 个人。另外两种情况要困难些，因为为了确定第 n 个人是否是社会名流，我们需要提 $2(n-1)$个问题。如果在第 n 步提了 $2(n-1)$个问题，则总共的问题数目为 $n(n-1)$（这是我们要避免的）。因此需要其他的方法。

这里的技巧是"倒推"考虑问题。确定一个社会名流可能很难，但是确定某人不是社会名流可能会容易些。毕竟，非社会名流要比社会名流多得多。如果我们把某人排除在考虑之外，则问题规模从 n 减小到 $n-1$。再有，我们不需要删除某个特定的人；任何人都可以。设想我们问 Alice 是否知道 Bob。若她知道，则她不可能是社会名流；若她不知道，则 Bob 不可能是社会名流。根据一个问题，我们可以删除他们其中之一。

现在再次考虑开始的三种情况。我们不是随意地把某个人作为第 n 个人。可以使用前面的思想来删除 Alice 或者 Bob，然后求解剩下 $n-1$ 个人的问题。我们保证第三种情况不会发生，因为被删除的人不可能是社会名流。进一步，如果情况 3 出现，即在 $n-1$ 个人中没有社会名流，则在 n 个人中也没有社会名流。最后只剩下情况 1，但这种情况比较简单。如果在 $n-1$ 个人中有社会名流，则只要再多问两个问题就可以验证他是否是整个集合中的社会名流；否则就没有社会名流。

这个算法如下。我们问 A，他是否知道 B，并根据答案删除 A 或者 B。让我们假定删除的是 A。则（由归纳法）在剩下的 $n-1$ 个人中找到一个社会名流。如果没有社会名流，算法就终止；否则，检测 A 是否知道此社会名流，而此社会名流是否不知道 A。

实现 就像上一节的算法那样，不用递归，而是用迭代可以使社会名流算法更高效。算法被分为两个阶段。在第一阶段，通过消除只留下一个候选者，在第二阶段，我们检查这个候选者是否就是社会名流。开始时我们有 n 个候选者，为了方便讨论，假设其存储在一个栈中。对每一对候选者，通过问一个问题就可以消除其中之一，他们中的一个是否知道另一个。开始时从栈中取出最前面的两个候选者，并消除其中之一。然后若栈非空，则从栈中取出另一候选者，又消除了两者之一，则在每一步都只剩下一个候选者。当栈为空时，只有一个候选者留下。我们就检查这个候选者是否就是社会名流。算法如图 5.4 所示（注意，栈的操作是通过索引 i、j 和 *next* 进行的）。

```
算法 Celebrity (Know)
输入： Know（一个 n×n 的布尔数组）
输出： celebrity
begin
    i := 1 ;
    j := 2 ;
    next := 3 ;
    {第一阶段，通过消除只留下一个候选者}
    while next ⩽ n + 1 do
        if Know[i, j] then i := next
        else j := next ;
        next := next + 1 ;
      {i 和 j 的其中之一被消除}
    if i = n + 1 then
        candidate := j
    else
        candidate := i ;
    {现在检查候选者确实是社会名流}
    wrong := false ;
    k := 1 ;
    Know[candidate, candidate] := false ;
      {虚构的变量通过检验}
    while not wrong and k ⩽ n do
        if Know[candidate, k] then wrong := true ;
        if not Know[k, candidate] then
            if candidate ≠ k then wrong := true ;
        k := k + 1 ;
    if not wrong then celebrity := candidate
    else celebrity := 0{没有社会名流}
end
```

图 5.4 算法 *Celebrity*

复杂度　至多要询问 3(n–1)个问题；第一阶段的 n–1 个问题用于消除 n–1 个人，而为了验证侯选者即为社会名流至多需要 2(n–1)个问题。注意，输入的规模不是 n，而是 n(n–1)（矩阵中项的数目）。这个结果表明，只需要查看$O(n)$ 个项就确定一个社会名流是可能的，即使从先验看，解可能和 n(n–1)个项中的每一个都相关。

评述　这个优美解法中的关键思想是，用一种聪明的方式把问题规模从 n 降到 n–1。这个例子表明，有时候花费一些代价来执行约简会更有效率。不要开始时简单地考虑一个规模为 n–1 的任意输入，并试图扩展它。我们将看到更多的例子，在这些例子中我们把真正的时间都花在构造一个正确的归纳次序上，而花费这些时间是值得的。

5.6　分治算法：轮廓问题

到目前为止，我们已经看了一些来自图论和数值计算的问题。现在这个例子处理的是画的轮廓问题。

> **问题**　给定一个精确的位置以及在城市中几座矩形建筑的外形，画出这些建筑的（二维）轮廓，并消去隐藏线。

一个例子的输入在图 5.5（a）中给出；对应的输出在图 5.5（b）中给出。我们只对二维图片感兴趣并假定所有建筑的底部都坐落在一条固定线上（即它们有相同的水平线）。建筑 B_i 通过三元组（L_i, H_i, R_i）来表示。L_i 和 R_i 分别表示建筑的左右 x 坐标，而 H_i 表示建筑的高度。一个轮廓是一列 x 坐标以及与它们相连的高度，按照从左到右排列。例如，图 5.5（a）中的建筑对应于下面的输入：

(1, **11**, 5), (2, **6**, 7), (3, **13**, 9), (12, **7**, 16), (14, **3**, 25), (19, **18**, 22), (23, **13**, 29)和(24, **4**, 28)。

（粗体字的数值表示高度。）在图 5.5（b）中的轮廓如下所示：

(1, **11**, 3, **13**, 9, **0**, 12, **7**, 16, **3**, 19, **18**, 22, **3**, 23, **13**, 29, **0**)。

（再次说明，粗体字的数值表示高度。）

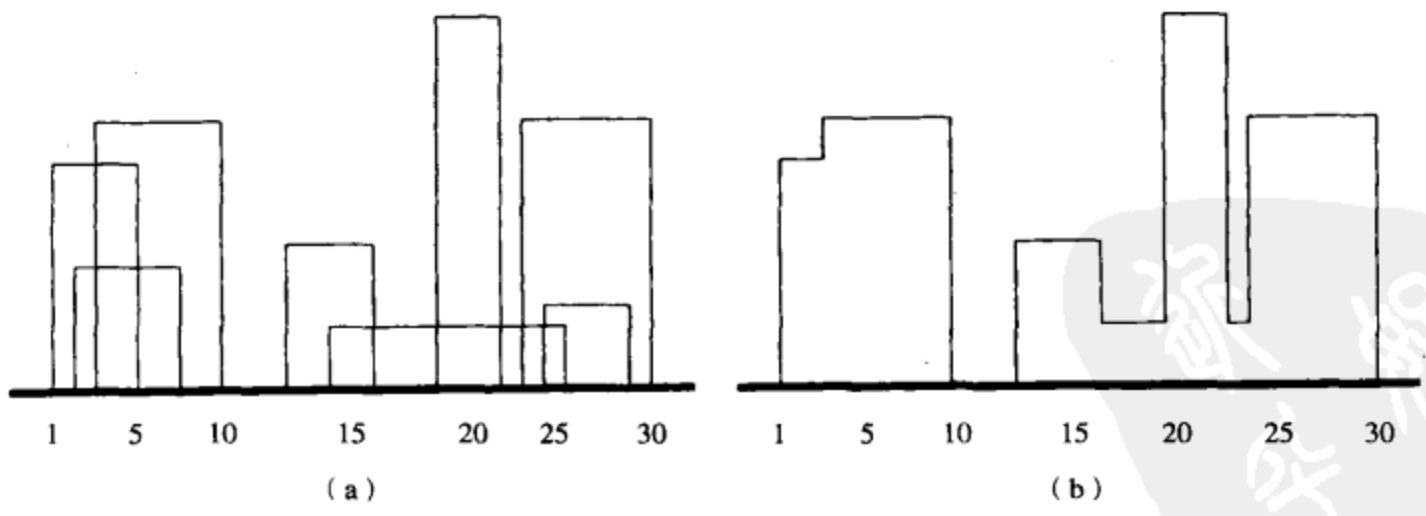


图 5.5　轮廓问题：（a）输入；（b）轮廓

这个问题的直接算法是基于增加一栋建筑到轮廓中。归纳假设很简单。假定已知如何求解 n–1 栋建筑的问题，然后再增加第 n 栋建筑。对于只有一栋建筑，问题是平凡的。为了增加建筑 B_n 到轮廓中，我们需要把它与已有的轮廓贯穿相交（见图 5.6）。令 B_n 是（5, **9**, 26）。我们先从左到

右扫描轮廓，以找到 B_n 左侧安置的点（即寻找适当的 x 坐标，在本例中是 5）。在此情况下，“盖住”5 的水平线是从 3 到 9，并且它的高度是 13。现在扫描轮廓，考虑下一条水平线，并且当 B_n 的高度高于已有高度时进行调整。当到达大于 B_n 右侧的 x 坐标时，就停止。例如，我们不调整从 3 到 9 的高度，但是从 9 到 19 进行调整，然后从 22 到 23 再进行一次调整。新的轮廓表示如下所示：

（1, **11**, 3, **13**, 9, 9, 19, **18**, 22, **9**, 23, **13**, 29, **0**）。

算法显然是正确的，但并不那么高效。在最坏情况下，针对 B_n 所进行的扫描需要 $O(n)$步。因此，总的步数将是 $O(n)+O(n-1)+\cdots+O(1)=O(n^2)$。

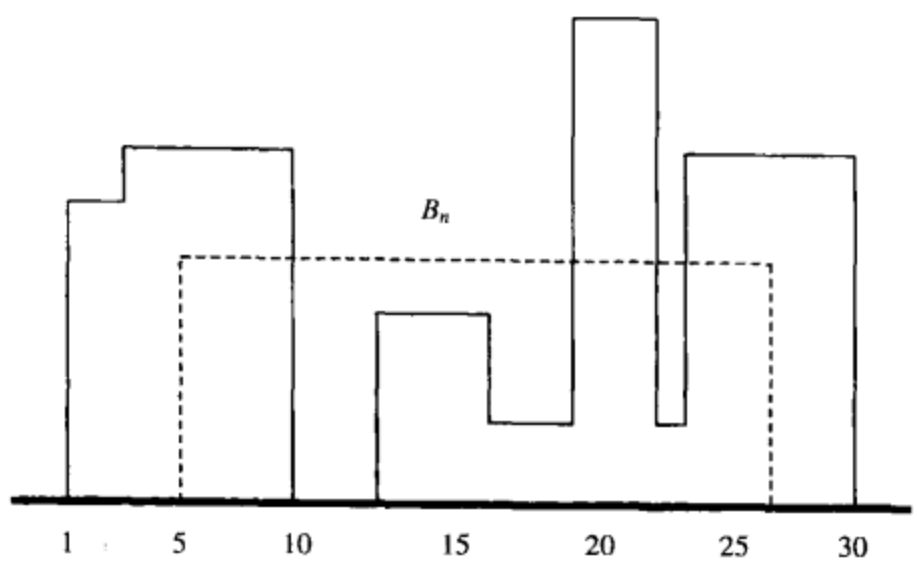


图 5.6 增加一栋建筑（虚线）到图 5.5（b）的轮廓（实线）中

为了改进算法的执行效率，我们使用一个著名的技术，即**分治法**。这里不是用简单的归纳原理把 $n-1$ 的解扩展到 n 的解，而是把 $n/2$ 的解扩展到 n 的解。（再次说明，一栋建筑的基础情况是平凡的。）分治算法把输入分成小的子集，递归求解每一个子集，然后把这些解合并起来。通常，把问题分成几个大约相等规模的子问题会比较有效。正如我们在第 3 章中看到的，递归关系 $T(n)=T(n-1)+O(n)$的解是 $T(n)=O(n^2)$，然而 $T(n)=2T(n/2)+O(n)$的解是 $T(n)=O(n\log n)$。所以，如果我们把问题分成两个相等规模的子问题，则解的组合为线性时间，算法的运行时间为 $O(n\log n)$。分治技巧非常有用，我们将看到更多关于它的例子。

本例中，隐藏在分治算法后的关键思想是，观察到在最坏情况下，把一栋建筑与已有轮廓合并只需要线性时间，并且两个不同轮廓合并也只需要线性时间。大约同时，对于使用后一种方法我们将得到更多的信息。基本上使用类似于把一栋建筑与已有轮廓合并的算法，就能把两个轮廓合并（见图 5.7）。我们从左到右同时扫描两个轮廓，匹配 x 坐标，并在需要时调整高度。这个合并可以在线性时间内完成，因此在最坏情况下完整的算法运行时间是 $O(n\log n)$。这个算法类似于将在 6.4.3 节讨论的归并排序。所以，这里就不给出精确的轮廓算法了。

评述 人总是追求价廉物美，这个原则既不神秘也不抽象。如果算法中包含一个比需求更通用的步骤，那么可以考虑把这个步骤应用到问题的更复杂部分。分治算法如此有用的原因在于，它把合并步骤利用到了最充分。在 3.5.2 节给出的递归关系覆盖了绝大多数常见的分治算法。你必须牢记这些递归关系。

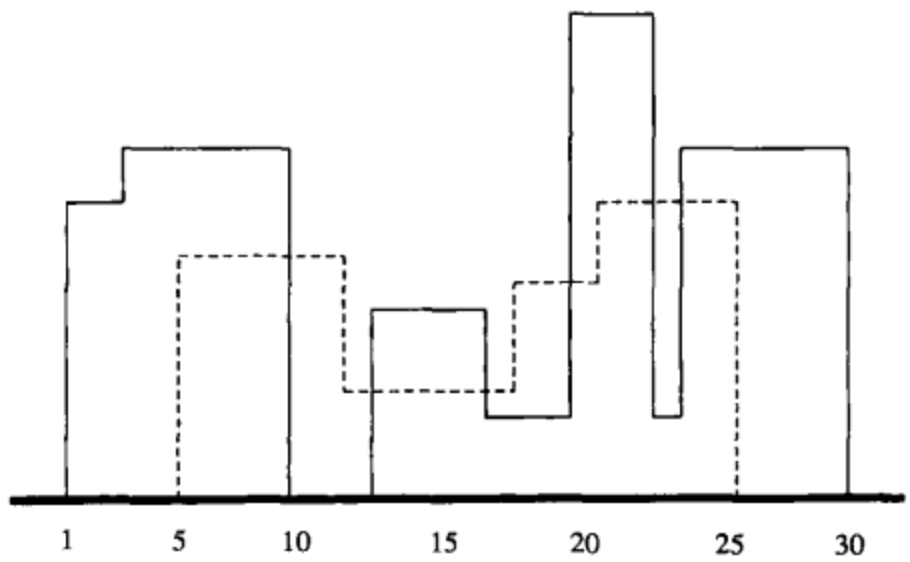


图 5.7　合并两个轮廓

5.7　在二叉树中计算平衡因子

令 T 是一棵根为 r 的二叉树。节点 v 的高度是 v 与树下方最远叶子的距离。节点 v 的**平衡因子**被定义成它的左子树的高度与右子树的高度的差（我们假定节点的儿子被从左到右标记）。在第 4 章中，我们讨论了 AVL 树，其所有节点的平衡因子为–1、0 或 1。在本节，我们考虑任意的二叉树。图 5.8 中显示了一棵树，它的每一个节点被数值 *h/b* 标记，*h* 是节点的高度而 *b* 是节点的平衡因子。

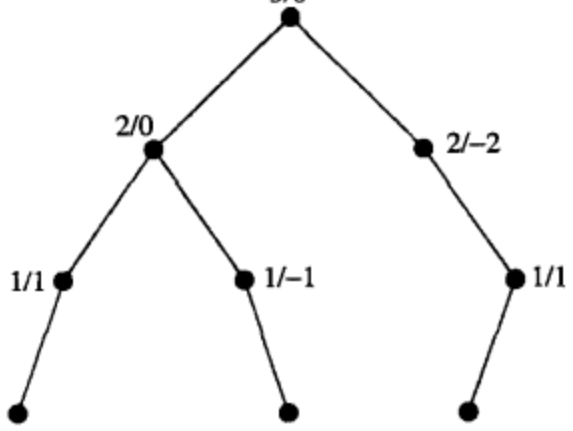


图 5.8　一棵二叉树。在数值 *h/b* 中，*h* 是节点的高度而 *b* 是节点的平衡因子

问题　给定一棵 n 个节点的二叉树 T，计算其所有节点的平衡因子。

下面使用正规的归纳方法和直接的归纳假设。

归纳假设：我们已知如何计算节点数小于 n 的二叉树的全部节点的平衡因子。

$n = 1$ 的基础情况是平凡的。给定一棵节点数 $n>1$ 的数，去掉树的根，则要（归纳）求解剩下的两棵子树。选择去掉根，是因为节点的平衡因子仅依赖于该节点以下的节点。现在我们知道了除根以外所有节点的平衡因子。然而，根的平衡因子，并不依赖于它的儿子的平衡因子，而是依赖于它们的高度。因此，简单的归纳在这里并不适用。我们需要知道根的儿子的高度。这里的想法是在原始问题中把高度查找问题包括进去。

更强的归纳假设：已知如何计算节点数小于 n 的二叉树的全部节点的平衡因子和高度。

再次说明，基础情况是平凡的。现在，当考虑树的根时，通过计算它的儿子的高度的差，就可以容易地确定其平衡因子。进一步，我们可以确定根的高度——它是两个儿子高度的最大值再加上 1。

该算法的关键是对问题做了轻微的扩展。我们并不只计算平衡因子，还计算高度。由于高度容易计算，所以扩展后的问题变成了一个易于解决的问题。许多情况下，求解一个更强的问题会容易些。通过归纳，我们仅需要把一个小规模问题的解扩展到一个大问题的解。如果解是更广泛的（因为问题被扩展了），则归纳步骤可能更容易，这是因为我们有更多的东西可以利用。一个常见的错误是，我们会忘记在这个问题中有两个不同的参数，每一个都必须被分别计算。本书后面将展示几个这类错误的例子。

5.8 寻找最大连续子序列

下面的问题来自于 Bentley [1986]（也出现在 Bates and Constable [1985]中）。

> **问题** 给定一个实数序列 $x_1, x_2, \cdots, x_n$（不必是正数），寻找一个（连续的）子序列 $x_i, x_{i+1}, \cdots, x_j$，使得其数值之和在所有连续子序列数值之和中是最大的。

一般称这个子序列为**最大子序列**。例如，在序列（2, –3, 1.5, –1, 3, –2, –3, 3）中，最大子序列是（1.5, –1, 3），它的和是 3.5。在一个给定序列中可能有几个最大子序列。如果所有的数值为负数，则最大子序列为空（由定义，空的子序列之和为 0）。我们希望有一个解决该问题的算法，并且仅对此序列扫描一次。

直接的归纳假设如下所示：

归纳假设：已知如何找到规模小于 n 的序列的最大子序列。

如果 $n=1$，若该数值非负，则最大子序列只包含一个数值，否则为空子序列。考虑一个规模 $n>1$ 的序列 $S=(x_1, x_2, \cdots, x_n)$。由归纳，已知如何在 $S'=(x_1, x_2, \cdots, x_{n-1})$ 中找到最大子序列。如果其最大子序列为空，则 S'中所有的数值为负数，我们仅需要考察 x_n。假设通过归纳法在 S'中找到的最大子序列是 $S'_M=(x_i, x_{i+1}, \cdots, x_j)$，对于特定的 i 和 j 有 $1 \leqslant i \leqslant j \leqslant n-1$。如果 $j=n-1$（即最大子序列是一个后缀），则容易把这个解扩展到 S 中：若 x_n 是正数，则把它加到 S'_M 中；否则，S'_M 仍是最大子序列。然而如果 $j<n-1$，则有两种可能：或者 S'_M 仍是最大；或者存在另一个子序列，它在 S'中不是最大，但在增加了 x_n 的 S 中是最大者。

这里关键的思想是**增强归纳假设**。我们先利用它来解决最大子序列问题以阐明这种技巧，然后在下一节对它做更一般性的讨论。使用直接归纳假设，我们的问题是可能在 S'中不是最大的子序列经过 x_n 扩展后，从而得出一个新的最大子序列。仅仅知道 S'中的子序列是不够的。x_n 只能扩展一个以 $n-1$ 结束的子序列——即 S'的一个后缀。设想我们把归纳假设增强到包括最大后缀的信息，用 $S'_E=(x_k, x_{k+1}, \cdots, x_{n-1})$表示。

更强的归纳假设：已知如何找到规模小于 n 的序列的最大子序列，以及作为后缀的最大子序列。

如果知道这两个子序列，算法就明确了。我们把 x_n 加到最大后缀中，如果它的和大于原来的最大子序列，则得到一个新的最大子序列（同样也是一个后缀），否则，保留以前的最大子序列。但求解过程还没有完全结束，我们还需要寻找新的最大子序列。我们不能总是简单地把 x_n 加到以前

的最大后缀中。有可能以 x_n 结束的最大后缀的和是负数。在此情况下，最好把空集作为最大后缀（即把随后的 x_{n+1} 单独考虑）。寻找最大子序列之和的算法在图 5.9 中给出。

```
算法 Maximum_Consecutive_Subsequence (X, n)
输入：X（大小为 n 的数组）
输出：Global_Max（最大连续子序列之和）
begin
    Global_Max := 0 ;
    Suffix_Max := 0 ;
    for i := 1 to n do
        if x[i] + Suffix_Max > Global_Max then
            Suffix_Max := Suffix_Max + x[i] ;
            Global_Max := Suffix_Max
        else if x[i] + Suffix_Max > 0 then
                Suffix_Max := x[i] + Suffix_Max
            else Suffix_Max := 0
end
```

图 5.9　算法 *Maximum Consecutive_Subsequence*

5.9　增强归纳假设

增强归纳假设是用归纳法证明数学定理时最重要的技巧之一。当试图用归纳方式证明时，我们经常遇到以下情节。用 P 来指示定理。归纳假设可以用 $P(<n)$ 表示，而证明必须推导出 $P(<n) \Rightarrow P(n)$。在许多情况下，我们可以增加另一个假设，称之为 Q，从而使证明变得容易。即，证明 $[P \text{ and } Q](<n) \Rightarrow P(n)$ 要比证明 $P(<n) \Rightarrow P(n)$ 容易。这个假定似乎正确，但是如何证明它却并不那么明显。这里的窍门是把 Q 包含在归纳假设中。现在必须证明 $[P \text{ and } Q](<n) \Rightarrow [P \text{ and } Q](n)$。$P$ and Q 是比 P 更强的定理，但是通常更强的定理更容易证明。通过正确的增加假定，这个过程可以反复进行，从而使证明变得可操作。最大子序列问题就是一个如何利用该原理对算法进行改进的好例子。

与这个原理类似的是一个众所周知的现象：基于一亿美元的销售来增加一百万美元的利润，要比基于十美元的销售来增加一千美元的利润更容易。

在使用这一技巧时，人们最易犯的错误是忽视了这个事实，增加了一个额外假定，它必须也有相应的证明。换句话说，当他们证明 $[P \text{ and } Q](<n) \Rightarrow P(n)$ 时，忘记了 Q 是被假定的。这个疏忽对应于，在最大子序列例子中忘记了计算新的最大后缀。在平衡因子例子中，它对应于忘记了分别计算高度——遗憾的是，这是个常见的错误。我们再怎么强调这个事实都不过分：

至关重要的是要精确地按照归纳假设进行问题求解。

6.11.3 节、6.13.1 节、7.5 节、8.3 节和 12.3.1 节（以及其他章节）将提供关于增强归纳假设的更复杂例子。

5.10　动态规划：背包问题

假设有许多不同形状和大小的物品，还有一个背包，你惟一的目标是尽可能将物品塞满背包。背包可能对应于一辆卡车、一艘轮船或者一块硅片，问题是要把物品放进去。关于这个问题有许

多不同的版本；这里仅考虑一种简单的情况，它处理一维的物品。背包问题的另一版本参见习题及第 11 章。

> **问题** 给定一个整数 K 和 n 个不同大小的物品，第 i 个物品的大小为整数 k_i，寻找一个物品的子集，它们的大小之和正好为 K，或者确定不存在这样的子集。

我们用 $P(n, K)$表示该问题，其中 n 表示物品的数目而 K 表示背包的大小。我们将隐含地假定 n 个物品是问题给定的输入，并且不把它们的大小包括在问题的符号中。因此，$P(i, k)$表示前 i 个物品和背包大小为 k 的问题。为了简单起见，先只关注判定问题，即确定是否有解存在。我们从直接的归纳假设开始。

归纳假设（最初的设想）：已知如何求解 $P(n-1, K)$。

基础情况是很容易的；如果这个单一物品的大小为 K，则仅有一个解。如果对于 $P(n-1, K)$有一个解，即如果有一种方法把 $n-1$ 个物品中的一些放进背包里，则结束；我们将完全不利用第 n 个物品。然而，假设对于 $P(n-1, K)$不存在解，那么可以用这个否定的结论吗？是的，它意味着第 n 个物品必须被包括在内。在此情况下，剩余的物品必须适合放入大小为 $K-k_n$ 的小一些的背包。这样就将问题归约到较小的子问题 $P(n-1, k)$和 $P(n-1, k-k_n)$。为了完成这个解答，必须增强假设。我们不仅需要对大小为 K 的背包求解，还要为所有不大于 K 的背包求解。

归纳假设（第二次设想）：已知如何求解 $P(n-1, k)$，其中 $0 \leqslant k \leqslant K$。

以前的归约不依赖于 K 的一个特定值；它对所有的 k 都成立。我们能够用这个假设来求解 $P(n, k)$，其中 $0 \leqslant k \leqslant K$。基础情况 $P(1, k)$很容易解决：如果 $k=0$，则总有一个（平凡的）解；否则，只有当第一个物品的大小为 k 时才有一个解。现在我们把 $P(n, k)$归约到两个问题，$P(n-1, k)$和 $P(n-1, k-k_n)$。若 $k-k_n<0$，则忽略第二个问题。这两个问题都可以归纳地解决。这是一个有效的归纳，并且我们得到了算法；然而，这个算法可能是低效率的。我们把一个规模为 n 的问题归约到了两个规模为 $n-1$ 的子问题（也在一个子问题中归约 k 的值）！每一个子问题可能有被归约成另外两个子问题，从而导致一个指数式的算法。

幸运的是，在许多情况下改进这类问题的运行时间是可能的。主要着眼点在于全部可能问题的数目不是很大，也由此引入了 $P(i, k)$来证明对第一个参数有 n 种可能性，对第二个参数有 K 种可能性，那么总共仅有 nK 个可能的问题！指数式的运行时间是因为每经过一次归约后就对问题数目加倍，但是如果仅有 nK 个问题，则我们必须在许多次得到同样的问题。在求解中必须记住所有解答，从而对相同问题不做第二次求解。方法是把增强的归纳假设与强归纳（它不仅利用 $n-1$ 时的解，而且是利用所有较小规模情况时的解）结合起来。现在让我们看看如何实现这个方法。

把所有已知结果保存在一个 $n \times K$ 的矩阵里。矩阵中的第(i, k)项存放的是 $P(i, k)$的解的信息。从第二次设想的归纳假设得到的归约基本上计算出矩阵的第 n 行。第 n 行的每一项由它上面的两项计算出来。若我们的兴趣是找到一个确切的子集，则可以为每一项加一个标记，用于指示对应的项是否在这一步中被选中。标记能够从第(n, K)项回溯，从而子集能够得以复原。算法在图 5.10 中给出。图 5.11 展示了一个给定输入的完整矩阵。

算法 *Knapsack (S, K)*

输入： S（存储物品尺寸的大小为 n 的数组）和 K（背包的大小）

输出： P（一个二维数组，如果对于前 i 个元素和大小为 k 的背包存在解，则 $P[i, k].exist$ = true，同时若该解包括了第 i 个元素，则 $P[i, k].belong$ = true）

{关于算法改进的提示参见习题 5.15}

```
begin
  P [0, 0].exist := true ;
  for k := 1 to K do
      P [0, k ].exist := false ;
      {不需要对 P[i, 0], i≥1 进行初始化，因为其值将由 P[0, 0]计算得到}
  for i := 1 to n do
      for k := 0 to K do
          P [i, k ].exist := false ;{默认值}
          if P [i − 1, k ].exist then
              P [i, k ].exist := true ;
              P [i, k ].belong := false
          else if k − S [i ] ≥ 0 then
                  if P [i − 1, k − S [i ]].exist then
                      P [i, k ].exist := true ;
                      P [i, k ].belong := true
 end
```

图 5.10　算法 *Knapsack*

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
k_4=6	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

图 5.11　背包问题中表的构造例子。输入中包含大小分别为 2、3、5 和 6 的四件物品。表格中的符号如下："I"：一个包含这件物品的解被找到；"O"：一个不包含这个物品的解被找到；"–"：至今没有找到解。如果符号"–"出现在最后一行，则说明对于这个尺寸的背包无解

这里使用的方法是**动态规划**技术。动态规划的本质是把所有前面已知的结果建成一个大表格。这个表格是被迭代构造的。每一项由矩阵中它上面的其他项结合计算得出，或者由它左边的项结合计算得出。主要的问题是用最高效的方式来组织矩阵的构造。动态规划的另一个例子将在 6.8 节中给出。

复杂度　在表格中有 nK 项，每一项由其他两项在常数式时间内计算所得。因此，总的运行时间是 $O(nK)$。如果物品的大小不是太大，则 K 不可能太大，并且 nK 要远好于一个 n 的指数表达式。（如果 K 非常大，或者大小为实数，则这个方法不可行；这个问题将在第 11 章讨论。）如果我们仅对确定是否有解感兴趣，则答案在 $P[n, K]$里。如果感兴趣的是找到一个确切的子集，那么可以从第(n, K)项回溯，例如使用背包程序中的 *belong* 标记，这个复原子集过程需要 $O(n)$的时间。

评述　当问题可以归约到几个小一些但又不是足够小的子问题时，动态规划方法是有效的。所有可能的子问题都可被计算，我们将通过维护一个大的矩阵来进行这种计算。因此，仅当可能的子问题总数不是太大时，动态规划才可行。更有甚者，由于动态规划需要建一个很大的矩阵，

所以它通常需要很大的空间。（在某些情况下，例如图 5.10 中的程序，在任意时刻它仅保存矩阵的一小部分，则可能使用较少的空间。）它的运行时间通常至少是二次方程式。

5.11 常见的错误

本节中，我们简要提及一些在使用归纳进行算法设计时的常见错误。回顾 2.13 节中讨论的归纳证明的常见错误，所有的错误都与这里类似。例如，忘记了基础情况是比较常见的。在递归过程中，基础情况对于终止递归是必需的。另一常见错误是，把对于 n 的解扩展到问题对于 $n+1$ 时的一个特定实例的解，而不是对于任意实例。

无意识地改变假设是另一个常见错误。这里有一个典型的例子。图 $G=(V, E)$被称为**偶图**，如果它的顶点集合可被划分为两个子集，其中同一子集的两个顶点之间没有边相连。如果该图是连通的且为偶图，则此划分惟一（我们忽略这个证明）。

> **问题** 给定一个连通的无向图 $G=(V, E)$，确定它是否为偶图，若是偶图则做出相应的顶点划分。

一个错误解： 由归纳法，删除一个顶点 v，如果可能的话再对剩余的图进行划分。称第一个子集为 *red*，第二个子集为 *blue*。若 v 仅与 *red* 顶点相连，则把它加入 *blue* 子集。若 v 仅与 *blue* 顶点相连，则把它加入 *red* 子集。若 v 与两个子集的顶点都相连，则该图不是偶图（因为划分是惟一的。）

这个尝试中的主要错误是，当删除了一个顶点后，这个图有可能变成不连通的。因此，这个问题的较小实例并不等同于原来的实例，所以不能用归纳法。如果删除了一个顶点后并没有把图断开，则这个解答过程是有效的。这个问题有一个更好的解答方法，它不依赖于图是否连通；我们把它留给读者（见习题 7.32）。一个类似的例子以及常见错误的进一步讨论，参见 7.5 节。有关这个不正确算法的结论，包括在习题 5.24 中。

有时，改变假设是很吸引人的。如果假设是这种形式："已知如何找到什么和什么"，则我们会进一步设想我们能够在相同的努力下找到另一件简单的事物。但是，除非它被包含在归纳假设中，否则不能使用这样的假定。一种避免无意识地改变假设的方法是，把它看做是一个黑盒。不要对黑盒做任何改变，除非你已经把它打开了（即对它明确地做了重新定义）。

5.12 小结

本章介绍的几个设计算法的技术全部是同一种方法的变形。它们决不是所有已知的设计算法的方法，其他技术和大量例子将在后面章节中给出。学习这些技术的最好方法是利用它们来解决问题。本书的剩余部分就是专注于这一目的。本章中提到的原理包括：

- 通过把问题的一个实例归约到一个或多个较小规模的实例，可以使用归纳原理来设计算法。如果归约总能实现，并且基础情况可被解决，则算法可以通过归纳进行设计。因此，主要的思想是如何归约问题，而不是直接对问题求解。
- 把问题规模减小的一种最容易的方法是，去除问题中的某些元素。这个技术应该是处理问题的首要手段。可以有许多不同的方式，除了简单地去除元素外，若它不起作用（如 5.3 节），则把两个元素合并为一个也是可能的，或者找到一个在特定（容易）情况下可以处理的元素，或者引入一个新元素来取代原来的两个或多个元素（见 6.6 节）。

- 可以用多种方式来归约问题的规模。然而，不是所有归约都有同样的效率，因此要考虑所有归约的可能性。特别是，考虑不同的归纳次序是值得一试的。我们看到了先处理最大元素较好的例子，同时，也看到了先处理最小元素较好的例子，还有从中间开始处理的例子（见 6.2 节），也看到对树进行归纳时先删除根的例子（自上而下），以及先删除叶子的例子（自下而上）（见 6.4.4 节）。
- 减小问题规模的一个最有效的方法是把它分成两个（或多个）相等规模的部分。如果问题可以被分割，则分治法非常有效，其中子问题的输出可以很容易生成全问题的输出。分治算法将在 6.4 节、6.5 节、8.2 节、8.4 节、9.4 节以及 9.5 节中给出。
- 由于归约只能改变问题的规模，并不改变问题本身，所以应该寻找尽可能独立的小规模的子问题。例如，在一些项目中寻找某些次序的问题可以归约到按次序寻找（以及删除）项目的问题；剩余项目的相关次序与第一个项目无关（见 6.4 节和 7.5 节）。
- 然而，要有一种方法来克服归约问题必须与原始问题一致的这一局限：改变问题的描述。这是一个经常使用的重要方法，有时，它比削弱假设要好，可获得一个较弱的算法并作为完整算法中的一个步骤来使用（见 6.10 节）。
- 最后，我们可以把这些技术同时使用，或者做不同的组合。例如，可以通过增强归纳假设来使用分治法，从而使不同的子问题变得容易组合（见 8.4 节）。

参考文献和补充读物

本章叙述的方法是作者提出并发展而来的（Manber [1988]），但它决不是新的方法。在一般数学证明和算法领域中，归纳方法的使用都来源于 Goldstine 和 von Neumann（见 von Neumann [1963]），但是在 Floyd [1967]中才第一次得到充分的阐述。Dijkstra [1976]、Manna [1980]、Gries [1981]和 Dershowitz [1983]给出了类似于本书的方法，并与程序正确性证明一起作为程序的开发技术，其中描述程序的方法要比本章更加严格和详尽。2.12 节中循环不变量的使用，在某种意义上可以看成与本章归纳的使用等价。当然，递归技术已经在算法设计中被广泛使用（例如，见 Burge [1975]和 Paull [1988]）。

社会名流问题是由 Aanderaa 首次提出的（见 Rosenberg [1973]）。在验证阶段以及消除阶段的提问，可通过仔细处理来避免重复，通常节省$\lfloor \mathrm{lb}n \rfloor$个提问是可能的（King and Smith Thomas [1982]）。增强归纳假设可能是一个非常旧的窍门了。Polya [1957]把它称为**发明家悖论**（因为更强的事物反而易于发现或者证明。）它有时也被称为**泛化**。动态规划由 Bellman [1957]引入并形式化，它有许多应用和变形。对动态规划更详尽的描述，参见 Dreyfus and Law [1977]，或者 Denardo [1982]。引发习题 5.24 的最初观察是由 Tom Trotter 指出的。

操练型习题

5.1　请为多项式计算设计一个分治算法，它需要多少次加法和乘法？对比 Horner 规则，你能说出这个算法的一个优点吗？

5.2　尝试以下（在 5.3 节中使用过的）归纳推理步骤解决下面的最大导出子图问题：给定一个图 $G=(V, E)$，寻找一个最大导出子图 G'，在 G'中所有顶点的度至多（与 5.3 节问题中的“至少”相反）为 k。这个版本要比原来的困难得多，而且在原来版本中使用的方法在此不起作用。讨论为什么在此不起作用。（关于这个问题 $k=0$ 时简单情况的讨论，参见第 11 章。）

5.3　考虑映射算法（见图 5.3）。在算法结束时，集合 S 是否可能为空集？或者给出一个例子，

或者证明这不可能发生。

5.4 写出社会名流算法（见图 5.4）中第一个 while 循环的相应循环不变量。

5.5 给定一棵二叉树 T。如果 T 的所有节点的平衡因子为 0、1 或–1，则 T 被称为 AVL 树（见 4.3.4 节）。假设节点没有足够的空间来存储平衡因子。设计一个有效的算法来解决以下判定问题。给定一棵树 T，算法应该能确定它是否为 AVL 树。答案只能为“yes”或“no”。

5.6 修改最大连续子序列算法（见图 5.9），使得它能找到确切的子序列，而不仅仅是子序列的和。

5.7 写一个程序解使用 *belong* 标记的背包问题。

5.8 在背包算法中，（通过检查 $P[i-1, j]$）我们先检查第 i 个物品是否是多余的。如果存在一个包含 i–1 个物品的解，则取这个解。也可以做相反的选择，即取包含第 i 个物品的解，如果它存在的话（也就是先检查 $P[i, j-k_i]$）。你认为哪一个版本有更好的表现？重新画图 5.11 来反映这样的选择。

5.9 一个给定的背包问题可能有多个不同解。从背包算法得到的解的特征是什么？是什么把这个解与其他剩余部分区分开来的？如果根据习题 5.7 的策略做选择，要如何改变你的答案？

创造型习题

5.10 解决以下扩展的轮廓问题。设想在轮廓中的建筑物有屋顶。每一个建筑物下面为矩形，上面有一个三角形屋顶。（可以简单假定，所有的屋顶与建筑物成 45 度角。）再有，所有的建筑物在相同的水平线上。设计一个算法画出这种情况下的轮廓。

5.11 设想有两个不同的（有可能会被提议的）轮廓：一个投影到面上全部是蓝色，另一个投影到面上全部是红色。设计一个算法计算变成紫色的形状。换句话说，计算两个轮廓的相交部分。

5.12 令 $x_1, x_2, \cdots, x_n$ 是一串实数（不需要一定为正数）。设计一个 $O(n)$ 的算法，寻找一个（连续的）子序列 $x_i, x_{i+1}, \cdots, x_j$，使得它们的乘积在所有子序列乘积中最大。空子序列的乘积定义为 1。

5.13 假设给定一棵树，它不是 AVL 树。我们称一个节点为 AVL 节点，如果它的平衡因子是 0、1 或–1。设计一个算法对 T 中的非 AVL 节点做出标记，但是这些节点的所有后代必须是 AVL 节点。

5.14 令 $G = (V, E)$ 是有 n 个顶点的二叉树。我们想构造一个 $n \times n$ 矩阵，它的第 ij 项等于 v_i 到 v_j 的距离。（由于树是无向的，所以矩阵应该是对称的。）为一棵用邻接表表示的树，设计一个 $O(n^2)$ 的算法来构造这样的矩阵。

5.15 令 $G = (V, E)$ 是一棵二叉树。G 中两个顶点的距离连接这两个顶点的路径长度（相邻两点之间的距离为 1）。G 的直径是所有顶点之间距离的最大值。设计一个线性时间算法，找到给定一棵树的直径。

5.16 改进背包算法（见 5.10 节）的空间使用率。是否需要一个完整的 $n \times K$ 矩阵呢？改进后算法的空间复杂度是多少？

5.17 求解下面背包问题的一种变形：所有假设与 5.10 节中的相同，但是可以无限制地提供每一件物品。换句话说，问题就是把物品装入一个给定大小的背包里，但是每件物品可以出现许多次。

5.18 下面是背包问题的另一种变形：这里的假设与习题 5.17 的相同（n 个物品，无限制提供，固定背包大小），但是现在每件物品都标出了相应的价格。设计一个算法，找到如何把背包装满，使得其中装入的物品价值总和最大。

5.19 这是背包问题最常见的变形：这里的假设与习题 5.17 的相同（n 个已表明大小和价值的物品，无限制提供，固定背包大小，目标是价值最大），但是现在已不限制要把背包恰好塞满。我们感兴趣的只是使得总价值最大，满足的约束条件是，对于所选的物品背包要有足够的存放空间。

5.20 令 $x_1, x_2, \cdots, x_n$ 是一个整数集合，并且令 $S=\sum_{i=1}^{n} x_i$。设计一个算法，把集合划分成两个总和相等的子集，或者判定不存在这种可能性。算法的运行时间必须在 $O(nS)$ 中。

5.21 假定给你一个如同黑盒的算法，你看不到它是如何设计的，它有如下性质：如果输入任意一串实数以及一个整数 k，算法将会回答“yes”或“no”，以指出是否存在一个子集其总和恰好为 k。说明如何用黑盒找到这个总和为 k 的子集，如果存在的话。允许你使用这个黑盒 $O(n)$ 次（其中 n 为这串实数的规模）。

5.22 汉诺塔问题是一个有简单递归解的非平凡问题的标准例子。在一个桩子上放有 n 个不同大小的盘子，它们按递减顺序自下而上。还有另外两个空的桩子（见图 5.12）。这道难题的目标是移动所有的盘子，一次只能移一个，按照下面的方式从第一个桩子移到另一个桩子。盘子从一个桩子的顶部移到另一个桩子的顶部。只有当盘子比另一个桩子上所有的盘子都小时，它才可以移过去。换句话说，在整个期间，盘子大小自下而上递减的顺序必须保持。目标是，用尽可能少的步骤移动全部盘子。

a.（用归纳法）设计一个算法，为 n 个盘子找到一个最小移动次序以解决汉诺塔问题。

b. 你的算法要移动多少次？为移动的次数构造一个递归关系，并求解。

c. 证明在上小题中的移动次数是最优的，即证明不存在任何其他的算法可以使用更少的移动次数。

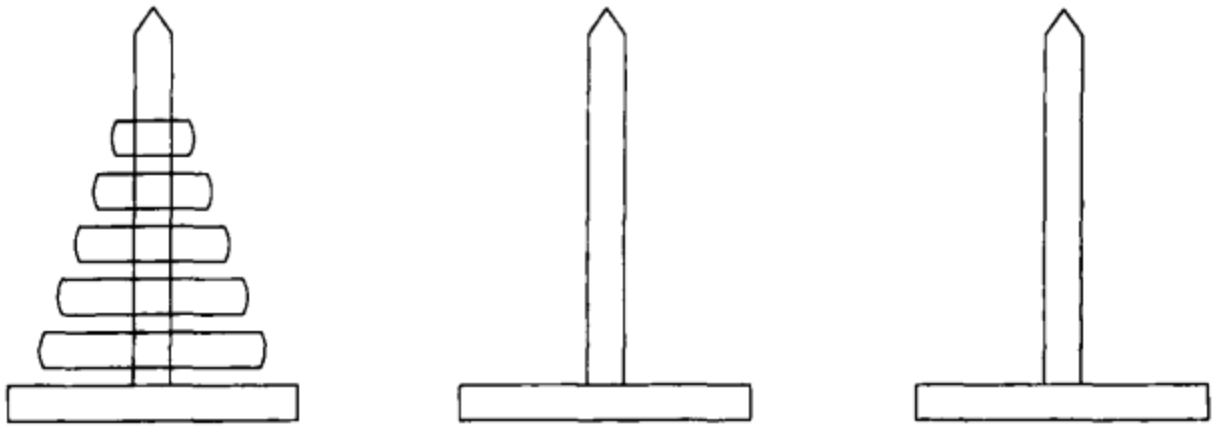


图 5.12　汉诺塔问题

5.23 为（习题 5.22 定义的）汉诺塔问题编写一个非递归的程序。

5.24 以下是汉诺塔问题（见习题 5.22）的一个变形。我们不再假定所有的盘子最初都在一个桩子上。它们可能任意放在三个桩子上，每个桩子上盘子的大小自下而上递减。这道难题的目的仍然是要把所有的盘子移动到一个特定桩子上，其约束与原来的问题相同，并且移动要尽可能少。对于这个版本的汉诺塔问题，在有 n 个盘子时，设计一个算法找到移动的最小次序。

5.25 下面是关于判定一个图是否为偶图的错误算法，这在 5.11 节中已有描述。在某种意义下，本习题表明不仅是算法错误，而且这个简单的方法也不可行。考虑图着色更一般的问题：给定一个无向图 $G=(V, E)$，图 G 的一个有效着色是，为顶点赋色后相邻两点没有相同的颜色。这个问题是找到一个有效的着色，并使用尽可能少的颜色。（通常，这是一个非常难的

问题，它将在第 11 章中讨论。）从而，如果一个图可以用两种颜色着色，则它是偶图。

a. 用归纳法证明树都是偶图。

b. 假定一个图是树（这意味着它是偶图）。我们想找到一个顶点划分，一分为二，使得在每个子集中的顶点之间都没有边相连。再次考虑 5.11 节给出的那个判定一个图是否为偶图的错误算法：任意删除一个顶点，（用归纳法）对剩余部分着色，然后再对这个顶点做最可能的着色。即我们用以前的颜色为此顶点着色，如果这个顶点与所有以前着过色的顶点相连，则增加一个新的颜色。证明，如果不考虑全局的连接状况，一次着色一个顶点，则可能需要大概 $1+\text{lb}n$ 种颜色。你需要设计一种构造方式，使得对于每一种顶点选择次序有最大的颜色数目。这个构造按以下方式依赖于选择次序。算法把一个顶点挑出来作为下一个顶点，并检查它的边。这里，你可以按你的需要增加一些与此顶点相连的边，假设这个图仍然是棵树，使得最后需要的颜色数目最大。在输入过一条边后你不能删除它（这将对算法造成欺骗，因为它已经见过这条边了）。获得这种构造的最好方法是通过归纳。假定已知需要颜色数目小于或等于 k 的一个顶点构造，要构造一个需要 $k+1$ 种颜色的图，但不能增加太多的新顶点。

第 6 章　序列和集合的算法

排序真稀奇，
杂乱无章尽去矣，
简洁有序一般齐。
——安娜·亨普斯特德·布兰奇（1875–1937）

6.1　引言

本章将讨论作为算法输入的有限序列或有限集合的问题。序列与集合的区别在于序列须考虑元素的顺序，而集合不必。还有一个区别是集合中的元素不能重复出现，序列则没有这样的限制。因此，如果算法的输入是有序的，不妨就将其作为序列，否则就作为集合来处理。本章中若非特别说明，算法的输入均用数组表示，且大小固定。序列或集合中的元素取自全序集合（如整数集合、实数集合等），由此任意元素之间可以比较大小。我们还假定问题中所有元素的类型均相同。本章中将研究最大值、排序、一些特殊的子序列、数据压缩和序列相似性等问题。

本章涉及应用广泛的多种不同算法，目的是提供更多的例子以说明第 5 章中介绍的算法设计方法，同时也刻画了一些重要的算法。这些重要算法有些极为常见（如二分查找、排序算法等），有些针对性很强（如文件压缩、序列比较等），还有些算法尽管不那么重要，但还是体现了一些有趣的技巧（如寻找集合中最大的两个元素、重叠子序列问题等）。

本章的第一个例子是二叉搜索问题，这是一个既基本又优雅，且形式多样，十分常见的算法；接着是排序问题（它是最受关注的算法问题之一）、两个与文本操作有关的问题，以及顺序统计和数据压缩等一些概率问题；最后通过几个优雅的算法介绍了一些有趣的设计技巧。

6.2　二叉搜索的几种形式

二叉搜索之于算法犹如轮子之于机械，它简约、优雅且无比重要。人们对它的研究不断有新的发现。二叉搜索的基本思想是通过一次查询把搜索空间（大致）减半。本节将介绍它的几种形式以展示其灵活多变的特点。

6.2.1　纯二叉搜索

> **问题**　给定实数序列 $x_1, x_2, \cdots, x_n$，满足 $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_n$。对某个实数 z，试确定 z 是否在该序列中出现。若 z 出现，试确定下标 i 使得 $x_i = z$。

为简单起见，这里只查找一个满足 $x_i = z$ 的下标 i。当然也可能想知道所有满足条件的 i，也可能对最大的 i 或者最小的 i 感兴趣。算法的思想就是通过检验中间数从而把搜索空间一分为二。为方便处理，不妨设 n 为偶数。若 z 小于 $x_{n/2+1}$，则它显然在序列的前半段；否则就在后半段。接着继续在前半段或后半段中查找 z，这就相当于把问题的规模缩减到 $n/2$ 大小，从而可以用数学归纳法解决。对归纳基础 $n=1$ 的情况，可以直接用 z 与之比较。算法在图 6.1 中给出。

```
算法 Binary_Search (X, n, z)
输入：X（元素下标从 1 至 n 的有序数组）和 z（待搜索的关键字）
输出：Position（使 X[i] = z 的下标 i；若 i 不存在，则输出 0）
begin
    Position := Find(z, 1, n) ;
end

function Find (z, Left, Right) : integer ;
begin
    if Left = Right then
        if X[Left] = z then Find := Left
        else Find := 0
    else
        Middle := ⌈½(Left + Right)⌉ ;
        if z < X[Middle] then
            Find := Find (z, Left, Middle-1)
        else
            Find := Find (z, Middle, Right)
end
```

图 6.1 算法 *Binary_Search*

复杂度 算法每进行一次比较，搜索的范围就减半，因此，在 n 个元素的序列中查找某个数值就需要比较 $O(\log n)$次。上文所述的二叉搜索在算法的末尾才比较 z 与 x_i 是否相等，当然也可以在算法的每一步中都比较这两个数是否相等。这里介绍的版本的缺点在于搜索过程不可能提早结束，其优点则在于每一步只比较一次（而不是每一步既比较两数是否相等，又比较是否不等）。因此一般来说搜索速度较快。尽管这个程序更适合写成递归程序，但要把它转换成非递归程序也非难事。二叉搜索在 n 较小时的性能不如其在 n 较大时的好。如果 n 较小，那么不妨线性地搜索这个序列。

6.2.2 循环序列的二叉搜索

如果在序列 $x_1, x_2, \cdots, x_n$ 中存在某个 i，使 x_i 是序列中的最小者，且序列 $x_i, x_{i+1}, \cdots, x_n, x_1, \cdots, x_{i-1}$ 是递增的，则称序列 $x_1, x_2, \cdots, x_n$ 是循环序列。

> **问题** 对某个已知的循环排序的链表，找出其中最小元素的位置。（为简单起见，假设该位置是惟一的。）

为了找到序列中的最小元素 x_i，这里利用二叉搜索的思想，通过一次比较将序列长度减半。在序列中任取两个元素 x_k 和 x_m，其中 $k < m$。若 $x_k < x_m$，i 就不可能落在 $k < i \leqslant m$ 中，因为 x_i 是序列中的最小数（注意，这里不能排除 x_k）。另一方面，若 $x_k > x_m$，i 就一定落在 $k < i \leqslant m$ 中，因为在这个区间内元素的顺序发生了跳变。这样一来，比较一次元素大小便能排除许多元素。通过选择恰当的 k 和 m，我们能在 $O(\log n)$次比较后找到 i。图 6.2 给出了这个算法。

6.2.3 二叉搜索特殊下标

下面的搜索问题没有待搜索的关键字，要查找的是满足某个特殊性质的下标。

> **问题** 给定一个由不同整数 $a_1, a_2, \cdots, a_n$ 按序排列而成的序列，试确定是否存在某个下标 i 使得 $a_i = i$。

```
算法 Cyclic_Binary_Search (X, n, z)
输入：X（元素各不相同、下标从 1 至 n 的循环排序数组）
输出：Position（x̄ 中最小元素的下标）
begin
    Position := Cyclic_Find(1, n) ;
end

function Cyclic_Find (Left, Right) : integer ;
begin
    if Left = Right then Cyclic_Find := Left
    else
        Middle := ⌊½ (Left + Right)⌋ ;
        if X [Middle ] < X [Right ] then
            Cyclic_Find := Cyclic_Find (Left, Middle)
        else
            Cyclic_Find := Cyclic_Find (Middle+1, Right)
end
```

图 6.2　算法 *Cyclic_Binary_Search*

由于待搜索元素的值未知，故该问题不能用纯二叉搜索来解决，然而该问题仍能借助二叉搜索原理求解。考察 $a_{n/2}$ 的值（这里同样假设 n 是偶数），如果它的值恰好是 $n/2$，则问题得解。否则，若该值小于 $n/2$，则由所有元素各不相同可知，$a_{n/2-1}$ 的值小于 $n/2-1$，依次类推。于是，前半段序列中的元素均不满足性质，可对后半段继续搜索。若该值大于 $n/2$，则同样可以得到相似的结论。图 6.3 给出了这个算法。

```
算法 Special_Binary_Search (A, n)
输入：X（元素各不相同、下标从 1 至 n 的整数数组）
输出：Position（满足 A [Position] = Position 的下标，若所有下标均不满足
        该性质，则返回 0）
begin
    Position := Special_Find(1, n) ;
end

function Special_Find (Left, Right) : integer ;
begin
    if Left = Right then
        if A [Left ] = Left then Special_Find := Left
        else Special_Find := 0 {不成功的搜索}
    else
        Middle := ⌈½(Left + Right)⌉ ;
        if A [Middle ] < Middle then
            Special_Find := Special_Find ( Middle + 1, Right )
        else
            Special_Find := Special_Find ( Left, Middle )
end
```

图 6.3　算法 *Special_Binary_Search*

6.2.4 二叉搜索长度未知的序列

有时可使用类似二叉搜索的过程，但将搜索空间加倍，而不是减半。假设某个一般的搜索问题，序列长度未知。此时我们不能把搜索空间减半，因为序列边界并不清楚。相反，我们要查找的是大于或等于 z 的某个 x_i。如果找到了这个 x_i，就可以在下标为 1 到 i 的范围内进行二叉搜索了。首先将 z 同 x_1 比较，若 $z \leqslant x_1$，则 z 只可能等于 x_1。借助归纳法，假设对某个 $j \geqslant 1$ 有 $z > x_j$，将 z 同 x_{2j} 比较，搜索空间加倍。若 $z \leqslant x_{2j}$，可推得 $x_j < z \leqslant x_{2j}$，然后再通过 $O(\log j)$次比较就能找到 z。总的来看，如果 i 是使 $z \leqslant x_i$ 的最小下标，那么通过 $O(\log i)$次比较就能得到某个 x_j，使得 $z \leqslant x_j$，然后再通过 $O(\log i)$次比较最终得到 i。

相同的算法也可用于序列长度已知的情况，但我们相信 i 是很小的。只有在这种情况下，这个算法才优于一般的二叉搜索算法，因为其花费的时间是 $O(\log i)$而不是 $O(\log n)$。由于它需要进行两次类似二叉搜索的过程，因而还须考虑其运行时间上的二次方因子，故仅当 $i = O(\sqrt{n})$时该算法才更优。

6.2.5 重叠子序列问题

有些问题看上去似乎与搜索无关，也能用二叉搜索的原理解决。令 A 和 B 是有限字母表上的字符序列，$A = a_1 a_2 \cdots a_n$，$B = b_1 b_2 \cdots b_m$，其中 $m \leqslant n$。若存在下标 $i_1 < i_2 < \cdots < i_m$，对所有的 j（$1 \leqslant j \leqslant m$）有 $b_j = a_{i_j}$，则称 B 是 A 的子序列。换句话说，如果我们能把 B 中的字符按序嵌入 A 中，嵌入的位置不必连续，则 B 是 A 的子序列。判断 B 是否为 A 的子序列非常简单。我们从头开始扫描 A 直到 b_1 第一次出现（如果存在的话），继续扫描直到 b_2 出现，以次类推。算法正确性的证明用归纳法很容易就能给出，留做练习。由于算法中包含一次对 A 和 B 的线性扫描，因而其运行时间为 $O(m+n)$。对已知的序列 B，定义 B^i 为 B 的每个字符连续出现 i 次所形成的序列。例如，$B = xyzzx$，则 $B^3 = xxxyyyzzzzzzxxx$。

> **问题** 给定两个序列 A、B，试确定 i 的最大值，使 B^i 是 A 的子序列。

这个问题被称为重叠子序列（stuttering-subsequence）问题。乍一看解决起来可能比较困难，然而用二叉搜索，问题很快就迎刃而解。

对每个已知的 i，构造序列 B^i 是轻而易举的。因此对任意给定的 i，我们能够判定 B^i 是否为 A 的子序列。不仅如此，如果 B^i 是 A 的子序列，那么对 $1 \leqslant i \leqslant j$，$B^i$ 也是 A 的子序列。由此推得 i 的最大值不能超过 n/m，否则序列 B^i 的长度就会超过 A。下面就可以开始二叉搜索了。首先令 $i = \lceil n/m \rceil / 2$，检验 B^i 是否为 A 的子序列。然后根据结果，如果它是 A 的子序列，就删除下界，否则就删除上界。这样共要进行 $\lceil \mathrm{lb}(n/m) \rceil$ 次检验才能确定 i 的最大值。花费的总时间为 $O((n+m)\log(n/m)) = O(n \log(n/m))$。6.8 节还会继续讨论序列比较的问题。

从以上解答可以得到启示：这类问题有一个通用的解法。每当要查找满足某个特定性质的 i，使用该算法就能判断 i 是否满足这一性质。如果 i 有上界，且 i 满足该性质，对 $1 \leqslant j \leqslant i$，$j$ 也满足，那么接下来就可以借助二叉搜索来解决。若 i 的上界未知，可以采取加倍法。即从 $i = 1$ 开始，对 i 的值加倍直至找到合适的范围。这样的搜索方法虽然耗时较长，但在 i 不是特别大的情况下还是足以解决问题的，尽管据此得到的算法并非最优。诸如重叠子序列等不少问题中，还可以删除多余的 $O(\log n)$因子。

6.2.6　解方程

这个问题似乎与本章的主题并无直接关系，但仍然值得一提。假设欲解方程 $f(x)=0$，其中 f 是可计算的连续函数。现已知 x 落在区间[a, b]中（$a \leqslant x \leqslant b$），且 $f(a)\cdot f(b)<0$（即 $f(a)$、$f(b)$一正一负）。我们想求得该方程在指定精度下的解。

由函数的连续性可知区间[a, b]中一定有解。于是就可以利用二叉搜索的一个变形——二分法（又名 Balzano 方法）来求出这个解。求解过程如下：取 $x_1=(a+b)/2$，若 $f(x_1)=0$（在给定精度下），x_1 就是方程的解；否则解一定在[a, b]的半区间[a, x_1]或[x_1, b]中，取区间端点函数值一正一负的那个区间继续进行上述过程，直至求得给定精度下的解。在 k 步之后，含解区间的长度为 $(b-a)/2^k$。

6.3　内插搜索

二叉搜索总是把搜索空间减半，从而保证其性能是对数级的。而当我们在某一步所得的值十分接近 z 时，在该值的邻域内搜索似乎比盲目地减半搜索空间更加合理。特别是当 z 很小时，更应从序列前端开始搜索，而不是机械地将序列一分为二。

比如给定一册书，当欲在其中寻找某页时是如何翻书的呢？假设我们要翻到第 200 页，而书本身看上去有 800 页厚。显然应该瞄准书的四分之一处，将其作为查找的标准。当然第一次不太可能一下子就正好翻到第 200 页，比方说，第一次翻到第 250 页，那么就相当于把搜索空间缩减到了前 250 页中。然后再根据第 200 页的相对位置，把这一次定位在搜索空间的 80%处，即往回翻 20%的页数。继续进行这样的过程直到最后一次只翻一页。这正体现了内插搜索的思想，也就是根据最可能接近目标的量来缩小搜索范围，而不是将搜索空间简单地减半。具体缩小的量由内插法决定，如图 6.4 所示。首次目标定在 $X[8]$，结果比 z 大。第二次插值在 $X[5]$，最后落在 $X[4]$ 上。图 6.5 给出了内插算法及其精确描述。

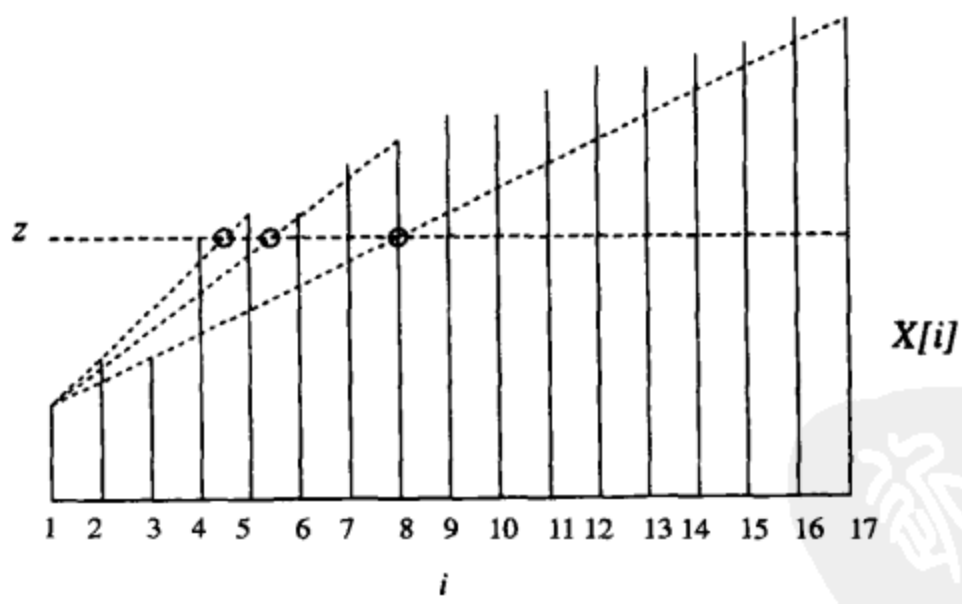


图 6.4　内插搜索

复杂度　内插搜索的性能不仅取决于序列的长度，还与序列自身的特点有关。对于某些序列来说，内插搜索不得不一一检查其中的每个数（见习题 6.4），而对于数据分布相对均匀的序列（毫无疑问书的页码是均匀分布的），内插搜索将十分高效。就所有可能的序列而言，内插搜索的平均比较次数是 $O(\log\log n)$。尽管内插搜索比二叉搜索在性能上有指数阶的进步，但在实际性能上

并无多大改善。原因有二：其一，除非 n 很大，否则 lb$n$ 已经足够小以至于其对数并不比其本身小多少；其二，内叉搜索需要更复杂的数字运算。

```
算法 Interpolation_Search (X, n, z)
输入：X（元素下标从 1 至 n 的有序数组）和 z（待搜索的关键字）
输出：Position（使得 X[i] = z 的下标 i；若 i 不存在，则输出 0）
begin
    if z < X[1] or z > X[n] then Position := 0
        {不成功的搜索}
    else Position := Int_Find(z, 1, n)
end

function Int_Find (z, Left, Right) : integer ;
begin
    if X[Left] = z then Int_Find := Left
    else if Left = Right or X[Left] = X[Right] then
        Int_Find := 0
    else
        Next_Guess := ⌈ Left + (z − X[Left])(Right − Left) / (X[Right] − X[Left]) ⌉ ;
        if z < X[Next_Guess] then
            Int_Find := Int_Find (z, Left, Next_Guess − 1)
        else
            Int_Find := Int_Find (z, Next_Guess, Right)
end
```

图 6.5 算法 *Interpolation_Search*

6.4 排序

排序问题是计算机科学中最受关注的问题之一，它不仅是许多算法的基础，而且在很多典型应用程序中，时间很大一部分就是花在排序上。排序问题种类繁多，因而相应的算法也层出不穷。在这短短的一节中，我们不可能面面俱到，而是介绍了几个常用的排序方法。跟前面一样，我们主要研究算法所基于的、对解决其他问题同样有所裨益的重要原理。当然这一节的内容会比其他章节更深入一些。

> **问题** 对已知的 n 个数 $x_1, x_2, \cdots, x_n$，从小到大排序。也就是要找到由 n 个不同下标 $1 \leqslant i_1, i_2, \cdots, i_n \leqslant n$ 所组成的序列，使得 $x_{i_1} \leqslant x_{i_2} \leqslant \cdots \leqslant x_{i_n}$。

为简单起见，若非特别说明，假设数据均不相同。当然，本节的所有算法在有相同数据时也正确。如果某个排序算法除了存储原始数据的数组之外不再借用额外的工作空间，则称之为原地置换排序算法。

6.4.1 桶排序和基数排序

最简单的排序方法恐怕要数“收发室”排序法了。先分配足够数量的信箱，称之为桶，然后把每个元素放到相应的桶中去，这种方法就叫做桶排序。如果把信件当成元素，然后，比方说根

据不同的州分拣，那么用桶排序算法就很高效。另一方面，如果按（5 位）邮政编码来排序信件，就需要一个非常大的容纳总共 100 000 个信箱的收发室。因此，桶排序仅适合事先预知元素属性范围的情况。下面将详细介绍桶排序算法。

假设有 n 个取值范围为 1 至 m 的元素。我们为它们分配 m 个桶，然后对每个 i，根据 x_i 的值，将 x_i 放到相应的桶中去。最后按序扫描每个桶，并把桶中的元素收集起来。很明显，这个算法很简单，复杂度为 $O(m+n)$。一方面，若 $m=O(n)$，则算法将演化成线性时间排序算法；另一方面，若相对于 n 来说，m 很大（如先前的邮政编码一例），那么 $O(m)$也会很大，而且算法需要 $O(m)$ 的存储空间，这比 m 自身很大更让人头疼。

桶排序思想可以很自然地被拓展为基数排序。不妨再考察一下上述按邮政编码分类的例子。用桶排序进行邮件分拣效率很低主要是因为邮编的范围太大，处理起来不太方便。那么是否可以通过某种措施来缩小范围呢？下面用数学归纳法对邮编的范围进行归纳，分几步进行。首先，用 10 个桶，根据邮编的第一位数字进行分拣，于是每个桶中就有 10 000 个不同的邮编（对应着剩余的 4 位数字）。其时间复杂度为 $O(n)$。第一步结束时，有了 10 个桶，每个桶中对应的邮编范围比分拣前要小。下面就对每个桶用归纳法，从而解决整个问题。由于每一步都能把范围缩小 10 倍，且邮编是 5 位数，因此总共只需 5 步。一旦这些桶被排完序，就能很容易用有序列表将它们组织起来。由于下面还要介绍这个思想的另一种变形，因此本算法的细节留给读者完成（见习题 6.5）。可以注意到上例中邮编范围的划分十分方便，根据邮编的十进制表示，范围逐步缩小十分自然。如果现在关键字改成了字符串，并要求按字典序排列，那么我们可以考虑每次只针对一个字符，从而得到的序列以字典方式排序。以上两个算法十分相似。这里介绍的基数排序（顾名思义，从左到右顺序扫描）也可称为基数调换排序法。

基数调换排序法可以直接用递归实现，只需要一些临时的桶即可（在邮编一例中约需 50 个；亦参见习题 6.5）。基数排序的另一种实现方法是用与先前相反的顺序进行归纳，即自右向左进行排序，从最低位而不是最高位开始。假设元素都是 k 位数的较大整数，每一位的取值范围为 0 到 $d-1$。归纳假设直截了当。

归纳假设： *知道应该怎样对小于 k 位的元素进行排序。*

这种方法和刚才的基数调换排序法都是按照不同的策略来扩展假设的（逆序归纳思想类似于 5.2 节中对 Horner 规则的讨论）。已知元素有 k 位，首先忽略最高位，然后用归纳法按剩余位数对其进行排序，可以得到按低 $k-1$ 位排列的元素序列。接着扫描一遍所有元素，借助 d 个桶，对最高位用桶排序。最后按序收集所有的桶。这个算法被称为直接基数排序法。现在要说明的是算法执行后，元素按 k 位依次排序。

我们断言，在最后一步中放入不同桶中的两个元素，其排列顺序是正确的。证明这个断言还用不到归纳假设，因为以字典序排序的序列，最高位决定了元素排列的顺序，与其他位无关。另一方面，如果两个元素的最高位相同，则由归纳假设，在最后一步之前，它们已经按正确的顺序排列。因此，仅须保证其排列顺序不变，这是算法中惟一一个微妙之处，另外它还提示了应该如何通过归纳法来证明算法的正确性。这里很重要的一点是同一桶内的元素排列顺序不变。要做到这一点，需要为每个桶维护一个队列，在每个阶段的最后把这 d 个队列连接起来从而形成一个包含所有元素的全局队列（按低 i 位排序的队列）。算法的精确描述见图 6.6。

复杂度　我们用 n 步把所有元素放入队列 GQ 中，然后再用 d 步初始化队列 $Q[i]$。算法的主循环执行 k 次，将每个元素从 GQ 中弹出，并压入某个 $Q[i]$中。除此之外，还要把各个 $Q[i]$队列连起来。因此算法的总运行时间为 $O(nk)$。

```
算法 Straight_Radix (X, n, k)
输入：X（元素下标从 1 至 n 的整数数组，每个元素有 k 位）
输出：X（排序后的数组）
begin
    We assume that all elements are initially in a global queue GQ;
    {为简单起见，这里使用 GQ；当然也可以通过 X 本身来实现}
    for i := 1 to d do
    {d 是可能的数制，比如对于十进制，d = 10}
        Initialize queue Q[I] to be empty;
    for i := k downto 1 do
        while GQ is not empty do
            pop x from GQ ;
            d := the ith digit of x ;
            insert x into Q [d] ;
        for t := 1 to d do
            insert Q [t] into GQ ;
    for i := 1 to n do
        pop X [i] from GQ
end
```

图 6.6 算法 *Straight_Radix*

在本节的余下部分中，我们所讨论的排序方法不考虑关键字的“结构”，而是直接在元素间进行比较，因而每次比较都包含整个关键字。除了假定元素之间可以互相比较之外，对元素的类型没有其他假设，故这些算法更为一般。

6.4.2 插入排序和选择排序

下面直接进行归纳。假设知道应该如何对 $n-1$ 个数进行排序。现在给定 n 个数，先对前 $n-1$ 个数排序，然后通过扫描已排完序的 $n-1$ 个数来确定第 n 个数应该插入的位置。显然，这样的过程被称为插入排序。在 n 较小的情况下，插入排序简洁又高效，而当 n 较大时，算法的效率就显得较低。最坏情况下，第 n 个数不得不与前 $n-1$ 个数逐个比较后才能找到合适的位置。因此对 n 个数排序可能就要做高达 $1+2+\cdots+n-1 = \frac{1}{2}(n-1)(n-2) = O(n^2)$次比较。不仅如此，插入第 n 个数时还需要移动其他的数。最坏情况下，第 n 步有可能移动 $n-1$ 个数，因而总的移动次数同样也是 $O(n^2)$。不过我们可以把这些数存储在数组中，然后对 $n-1$ 个已排序的树用二叉搜索来确定插入第 n 个数的正确位置，从而改进算法的性能。这样每插入一个数搜索过程就进行 $O(\log n)$次比较，总体比较次数因而降为 $O(n \log n)$。即便如此，数据移动次数还是没有减少，所得算法仍然是平方级的。

精心挑选第 n 个数还可以进一步改进这个直接进行的归纳。例如，可以把最大数选做第 n 个数。选最大数主要是因为我们知道它应该被放在哪个位置上，这里是数组的末尾。这样得到的算法首先是选出最大的数，然后放在其应处的位置上（只须与该位置上原有的数交换即可），然后递归地对剩余的数如是进行排序。这个算法被称为选择排序。选择排序仅在数据移动次数上优于插入排序，选择排序移动 $n-1$ 个数据（这里指数据交换），而插入排序在最坏情况下的移动次数为 $O(n^2)$。另一方面，由于选出最大数还需要 $n-1$ 次比较（6.5 节中将讨论查找最大数的问题），故总的比较次数仍为 $O(n^2)$，而借助二叉搜索的插入排序仅须比较 $O(n \log n)$次。

我们还可以用平衡树来提高插入或选择排序的性能（参见第 4 章）。比如说如果用 AVL 树，

每次插入花费 $O(\log n)$的时间，扫描 AVL 树从而得到有序数列花费 $O(n)$的时间。若由归纳假设已知如何为 $n-1$ 个数构造 AVL 树，则仅须耗费 $O(\log n)$时间插入最后一个数。算法的总时间包括 $O(n \log n)$的时间将 n 个数插入一棵空的 AVL 树，以及最后 $O(n)$的时间从树上得到有序数列。在 n 很大的情况下，这个算法比插入排序或选择排序更优，只是存储指针需要额外的空间。显然，这不是一个原地置换排序算法，而且算法本身相当复杂，下面我们会介绍比它更好的算法。插入排序和选择排序的程序比较容易，留做练习。

6.4.3　归并排序

为提高插入排序的性能，我们注意到在寻找某个数合适的插入位置时，扫描整个有序序列能为其他许多数同时找到相应的插入位置。在 5.6 节中这个思想已经实践过一次。我们可以只扫描一次就把两个有序序列归并起来。归并过程先依次考察第二个序列中的每个数，然后从小到大分别为其在第一个序列中寻找恰当的位置。把这一过程形式化地表述为：记 $a_1, a_2, \cdots, a_n$ 为第一个集合中的元素，$b_1, b_2, \cdots, b_m$ 为第二个集合中的元素，两个集合中的元素均以升序排列。扫描第一个序列，直至找到插入 b_1 的位置，然后插入 b_1；接着在插入处继续扫描 b_2，直至找到插入 b_2 的位置，如是依次插入余下的数。由于 b 序列是有序的，因此不必回溯查找。最坏情况下的比较次数是两个序列长度之和，那么数据移动的次数如何呢？如果每插入一个数都要移动相应的其他数，那么不少数据将会频繁地移动多次，导致效率十分低。由于归并后的数是按序排列的，我们不妨在归并过程中将其复制到一个临时数组，这样每个数恰好被复制一次。于是归并两个长度分别为 n 和 m 的有序序列共需 $O(n+m)$次比较和移动（前提是有额外的空间可供存储临时数组）。

上述归并过程可作为某个分治排序算法的基础，这个算法就是归并排序算法。具体如下：首先将序列分成相等或基本相等（序列有奇数个元素时）的两部分，然后将每个部分分别递归排序，最后把这两个有序部分按上述过程归并为一个有序序列。详细算法由下页中的图 6.7 给出。图 6.8 是归并排序的例子（复制过程省略）。

复杂度　令 $T(n)$是最坏情况下归并排序需要比较的次数。为简单起见，设 n 是 2 的幂。$T(n)$的值由如下递归关系式给出：

$$T(2n) = 2T(n) + O(n),\quad T(2) = 1。$$

该关系式的解是 $T(n) = O(n \log n)$（参见第 3 章），其渐近时间复杂度优于插入排序或选择排序所需的 $O(n^2)$。数据移动的次数同样是 $O(n \log n)$，比选择排序的次数 $O(n)$多。

尽管在 n 较大时归并排序好于插入排序，但它也有一些缺点。首先，归并排序不易实现；第二，归并过程中需要额外的空间来存储归并后的序列，因此它不是一种原地置换排序算法。（归并排序还有一些变形，需要的额外存储空间是常数；见参考文献部分。）每次归并两个短序列时都要复制所有的数，因而算法效率有所下降。

6.4.4　快速排序

归并排序及其分析体现了分治思想的高效。如果我们能把问题分成两个相同规模的子问题，然后分别解决每个子问题，再把子问题的解合并起来，最后就能得到一个 $O(n \log n)$的算法，当然前提是问题分割和解答合并的步骤是 $O(n)$的。归并排序的症结在于额外存储空间的使用，由于合并过程不确定，因此每个数在序列中的最终位置是不可预知的。那么究竟能不能采取一种分治思想使得每个数的最终位置是确定的呢？快速排序思想正是把较多的时间用在“分”上，而把较少的时间用在“治”上。

```
算法 Mergesort (X, n)
输入：X（下标从 1 至 n 的数组）
输出：X（排序后的数组）
begin
    M_Sort(1, n)
end

procedure M_Sort(Left, Right) ;

begin
    if Right - Left = 1 then
    {检查这种情况下 Left 和 Right 的大小并不必要，因为下面的程序已
    包含对这种情况的处理，然而这样写可使程序更高效}
        if X[Left] > X[Right] then swap (X[Left], X[right] )
    else if Left ≠ Right then
        Middle := ⌈½ (Left + Right)⌉ ;
        M_Sort(Left, Middle-1) ;
        M_Sort(Middle, Right) ;

        {现在我们把上面的两个有序序列归并为一个}

        i := Left ;
        j := Middle ;
        k := 0 ;
        while (i ⩽ Middle − 1) and (j ⩽ Right) do
            k := k + 1 ;
            if X [i] ⩽ X [j] then
                TEMP [k] := X [i] ;
                i := i + 1
            else
                TEMP [k] := X [j] ;
                j := j + 1 ;
        if j > Right then
            {把左边剩下的所有数移到数组的末尾}
            {若 i ⩾ Middle，则右边所有数已在数组的正确位置上}
            for t := 0 to Middle − 1 − i do
                X [Right − t] := X [Middle − 1 − t] ;
        {现在把 TEMP 复制回 X }
        for t := 0 to k − 1 do
            X [Left + t] := TEMP [t]
end
```

图 6.7　算法 *Mergesort*

假设知道某个数 x 满足：序列中一半的数大于或等于 x，另一半小于 x。根据与 x 的大小关系，可将序列一分为二。这样的分割需要 $n-1$ 次比较。由于两部分数据个数相等，因此其中一部分数据放在数组的前半段，另一部分放在后半段。后面将看到这样的分割还不需要额外的存储空间。序列分割完之后，接着将每个子序列递归排序。最后的合并自然而成，因为两个子序列本来就存储在数组中。所以，整个排序过程不需要任何额外空间。

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
(2)	(6)	8	5	10	9	12	1	15	7	3	13	4	11	16	14
2	6	(5)	(8)	10	9	12	1	15	7	3	13	4	11	16	14
(2)	(5)	(6)	(8)	10	9	12	1	15	7	3	13	4	11	16	14
2	5	6	8	(9)	(10)	12	1	15	7	3	13	4	11	16	14
2	5	6	8	9	10	(1)	(12)	15	7	3	13	4	11	16	14
2	5	6	8	(1)	(9)	(10)	(12)	15	7	3	13	4	11	16	14
(1)	(2)	(5)	(6)	(8)	(9)	(10)	(12)	15	7	3	13	4	11	16	14
1	2	5	6	8	9	10	12	(7)	(15)	3	13	4	11	16	14
1	2	5	6	8	9	10	12	7	15	(3)	(13)	4	11	16	14
1	2	5	6	8	9	10	12	(3)	(7)	(13)	(15)	4	11	16	14
1	2	5	6	8	9	10	12	3	7	13	15	(4)	(11)	16	14
1	2	5	6	8	9	10	12	3	7	13	15	4	11	(14)	(16)
1	2	5	6	8	9	10	12	3	7	13	15	(4)	(11)	(14)	(16)
1	2	5	6	8	9	10	12	(3)	(4)	(7)	(11)	(13)	(14)	(15)	(16)
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)

图 6.8　归并排序的一个例子。第一行是元素的原始顺序，后续每一行说明了元素的交换操作或归并运算，每一行中圈起的数据与当前操作有关

上述排序过程基于一个假设，x 的值是预知的。然而一般情况下 x 的值是未知的。不过很容易就能看到算法在 x 未知时依然有效。将分割步骤中的这个数值称为支点。我们的目的就是把序列分为两部分，一部分的数大于支点，另一部分小于或等于支点。下面的算法可以实现这样的分割。首先用两个指针 L 和 R 分别指向数组的左、右两侧，然后指针沿数组项相向而行。下面的归纳假设（即循环不变量）保证了分割的正确性。

归纳假设： 在算法的第 k 步，对任意 i，$i<L$，都有支点 $pivot \geqslant x_i$；对任意 j，$j>R$，都有支点 $pivot < x_j$。

这个归纳假设在算法开始时显然成立（因为 i 和 j 都不满足条件），那么目标就是要在第$(k+1)$步把 L 右移或者将 R 左移，使得归纳假设仍然成立。

当 $L=R$ 时，可能除了 x_L 之外，序列分割基本完成，这种情况会在后文中处理。现在假设 $L<R$，分两种情况讨论。如果 $x_L \leqslant pivot$ 或者 $x_R > pivot$，那么可以继续相向移动指针，假设仍然成立；否则，即 $x_L > pivot$ 同时 $x_R \leqslant pivot$，就交换 x_L 和 x_R，然后再相向移动指针。这两种情况都要至少移动一个指针，因此两个指针最终一定会相遇，算法也必定终止。

剩下的问题就是应该如何选择一个合适的支点以及算法最后一步中两个指针最终相遇的细节。分治算法在各个部分大小相等时性能最好，这就告诉我们支点越靠近序列的中点，算法的运行效率就越高。当然也可以采用序列的中数（下一节将讨论中数查找），但是这样做似乎划不来。在下面的讨论中将会看到，随机选择序列中的某个数不失为一种好方法。如果序列本身是随机排列的，那么不妨把第一个数选为支点。这样处理只是为了简单起见，相应的算法如图 6.9 所示。

```
算法 Partition (X, Left, Right)
输入：X（某个数组）、Left（该数组的左边界），以及 Right（该
      数组的右边界）
输出：X 和 Middle，使得对所有 i ≤ Middle，X [i] ≤ X [Middle]；
      对所有 j > Middle，X [j] > X [Middle]
begin
   pivot := X [Left] ;
   L := Left ; R := Right ;
   while L < R do
      while X [L] ≤ pivot and L ≤ Right do L := L + 1 ;
      while X [R] > pivot and R ≥ Left do R := R - 1 ;
      if L   R then
         exchange X [L] with X [R];
   Middle := R ;
   exchange X [Left] with X [Middle]
end
```

图 6.9 算法 *Partition*

第一个数被选为支点后，在分割的最后一步将其与 x_L 交换。依题意，支点就被置于序列的中点。其余步骤在讨论算法复杂度时介绍。在任何情况下，无论哪个支点都可以与第一个数交换，于是图 6.9 给出的算法仍然适用。

图 6.10 是算法 *Partition* 的一个例子。支点是第一个数（6）。圈起来的数字是上一步交换的。三次交换之后，可以看到 p_h 指向 $X[6]=1$，p_l 指向 $X[7]=12$。最后一次交换在中点（1）与支点（6）之间进行。交换完毕后，支点左边的数均小于或等于支点，右边的数均大于支点。形成的两个子序列（1 到 6 以及 7 到 16 两个）可以再递归排序。因此，快速排序是一种原地置换排序算法。算法在图 6.11 中给出，图 6.12 是快速排序的一个例子。

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
6	2	(4)	5	10	9	12	1	15	7	3	13	(8)	11	16	14
6	2	4	5	(3)	9	12	1	15	7	(10)	13	8	11	16	14
6	2	4	5	3	(1)	12	(9)	15	7	10	13	8	11	16	14
(1)	2	4	5	3	(6)	12	9	15	7	10	13	8	11	16	14

图 6.10 以 6 为支点的序列分割

复杂度 快速排序所花费的时间取决于输入和支点。如果支点永远将序列分成两个相等的部分，那么由递归关系，$T(n)=2T(n/2)+O(n)$，以及 $T(2)=1$ 可得 $T(n)=O(n \log n)$。下面会看到即使条件弱得多，仍能得到 $O(n \log n)$的性能。然而，若支点非常靠近序列的某一端，则运行时间就要长一些。举例来说，如果支点是序列中最小的数，那么首先分割操作需要 $n-1$ 次比较才能把支点放到正确的位置上。若序列中的数按升序排列，且总是把第一个数选为支点，则算法的运行时间增至 $O(n^2)$。当然，我们也有办法消除这种升序或近似升序排列的最坏情况，比较第一个、最后一个以及中间的那个数，然后取三者的中数（即第二大的数）作为支点。更为保险的方法是从序列中随机取数作为支点。这样得到的快速排序算法的最坏情况仍然是$O(n^2)$，因为仍有可能把最小的数选为支点，当然，选中最小数的概率很小。下面就对这种情况进行分析。

```
算法 Quicksort (X, n)
输入：X（下标从 1 至 n 的数组）
输出：X（排序后的数组）
begin
    Q_Sort(1, n)
end

procedure Q_Sort(Left, Right) ;
begin
    if Left < Right then
        Partition(X, Left, Right) ;
        Q_Sort(Left, Middle − 1) ;
        Q_Sort(Middle + 1, Right)
end
```

图 6.11　算法 *Quicksort*

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
1	2	4	5	3	⑥	12	9	15	7	10	13	8	11	16	14
①	2	4	5	3	⑥	12	9	15	7	10	13	8	11	16	14
①	②	4	5	3	⑥	12	9	15	7	10	13	8	11	16	14
①	②	3	④	5	⑥	12	9	15	7	10	13	8	11	16	14
①	②	3	④	5	⑥	8	9	11	7	10	⑫	13	15	16	14
①	②	3	④	5	⑥	7	⑧	11	9	10	⑫	13	15	16	14
①	②	3	④	5	⑥	7	⑧	10	9	⑪	⑫	13	15	16	14
①	②	3	④	5	⑥	7	⑧	9	⑩	⑪	⑫	13	15	16	14
①	②	3	④	5	⑥	7	⑧	9	⑩	⑪	⑫	⑬	15	16	14
①	②	3	④	5	⑥	7	⑧	9	⑩	⑪	⑫	⑬	14	⑮	16

图 6.12　快速排序的例子。第一行是初始输入。每一行都选择一个新的支点，并圈出当前所有的支点。显然，两个支点间的数是已排序的

假设每个 x_i 被选中的概率与支点一样，那么当支点是第 i 小的数时，快速排序的运行时间为

$$T(n) = n-1+ T(i-1) + T(n-i)。$$

（分割序列需要 $n-1$ 次比较，另外还要对两个长度分别为 $i-1$ 和 $n-i$ 的子序列排序。）如果每个数被选中的概率相等，则算法平均运行时间是

$$T(n) = n-1+\frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i))$$

$$= n-1+\frac{1}{n}\sum_{i=1}^{n}T(i-1)+\frac{1}{n}\sum_{i=1}^{n}T(n-i)$$

$$= n-1+\frac{2}{n}\sum_{i=0}^{n-1}T(i)。$$

以上就是递归关系式的完整推导过程。3.5.3 节曾讨论过这个特别的关系式，并求得其解为 $T(n)=O(n \log n)$。所以快速排序在平均时间上速度最快。

在实际应用中，快速排序的确很快，名副其实。主要原因不仅在于优雅的分治算法，还因为有许多数都跟同一个数（支点）进行比较。于是可以把支点存在某个寄存器中，而不必反复从内存中读取。这对大多数计算机而言显然能节省不少时间。

改进快速排序性能的方法之一就是要灵活地选择归纳基础，即不必从 1 开始归纳。上述快速排序一直被递归调用到归纳基础，也就包含了长度为 1 的序列。而对于短序列，像插入排序或选择排序之类的简单排序方法其效果比起适用于长序列的快速排序方法更好。因此可以把快速排序的归纳基础定为大于 1 的数（10 至 20 的范围似乎比较合适，不过还是与具体的应用场合有关），然后对归纳基础再进行插入排序。（换句话说，用“if Left < Right-Threshold”来替换原来对“if Left < Right”的判断，然后再加一个 else 部分进行插入排序。相对于原来的运行时间，修改后的算法节省了一段常数时间。6.11.3 节将介绍如何选择归纳假设来渐近地提高算法的性能。

6.4.5 堆排序

堆排序也是一种速度较快的排序算法。当 n 较大时，尽管一般来说堆排序不如快速排序，但是它并不比快速排序慢多少。另一方面，它不像快速排序那样无法估计性能，其性能是可以预知的。与归并排序类似，堆排序最坏情况下的运行时间是 $O(n \log n)$；与归并排序不同的是，堆排序是一种原地置换排序算法。本节重点介绍堆排序的一个部分——建堆。建堆算法告诉我们算法的设计和分析是相辅相成的。

第 4 章讨论过堆。这里假设用堆的隐式表示，即数组 $A[1..n]$中存储着所有的数，数组与树的对应关系如下。树的根存在 $A[1]$中，任意节点 $A[i]$的子节点存在 $A[2i]$和 $A[2i+1]$中。如果每个节点的值都大于或等于其子节点的值，则数组满足堆的性质。

堆排序的原理是这样的。算法的输入是数组 $A[1..n]$。首先，数组中的元素重新排列成堆，具体建堆过程将在后文介绍。若 A 是堆，则 $A[1]$是数组的最大元素，交换 $A[1]$和 $A[n]$使 $A[n]$中存储正确的那个元素。然后考察数组 $A[1..n-1]$，同样将这个数组中的所有元素排列成堆（其实只须考虑新的 $A[1]$），交换 $A[1]$和 $A[n-1]$，接着继续考察 $A[1..n-2]$。总的来说，堆排序包括一次初始的建堆过程，$n-1$ 次元素交换和重排成堆。元素交换后重排成堆实质上与 4.3.2 节中给出的算法是一致的。建堆过程本身是一件很有趣的事情，下面将详细介绍这个过程。堆排序总的运行时间是 $O(n \log n)$（每次交换花费 $O(\log n)$）加上建堆的时间。很明显，堆排序是一个原地置换排序算法，具体算法由图 6.13 给出。

```
算法 Heapsort (X, n)
输入：X（下标从 1 至 n 的数组）
输出：X（排序后的数组）
begin
    Build_Heap (X) ; {见下文}
    for i := n downto 2 do
        swap (A [1], A [i]) ;
        Rearrange_Heap (i – 1)

        {基本等同于图 4.7 中的程序 Remove_Max_from_Heap}
end
```

图 6.13 算法 *Heapsort*

建堆过程

现在重点介绍如何对任意数组建堆。

> **问题**　已知数组 $A[1..n]$的元素按任意顺序排列，重新排列这些元素使其满足堆的性质。

自顶向下和自底向上是两个很自然的建堆方法，它们分别对应着两种不同的对数组的扫描方式：自左向右或自右向左。图 6.14 对这两种方法都做了说明。首先用归纳法来描述它们，然后再说明它们在性能上有着天壤之别。

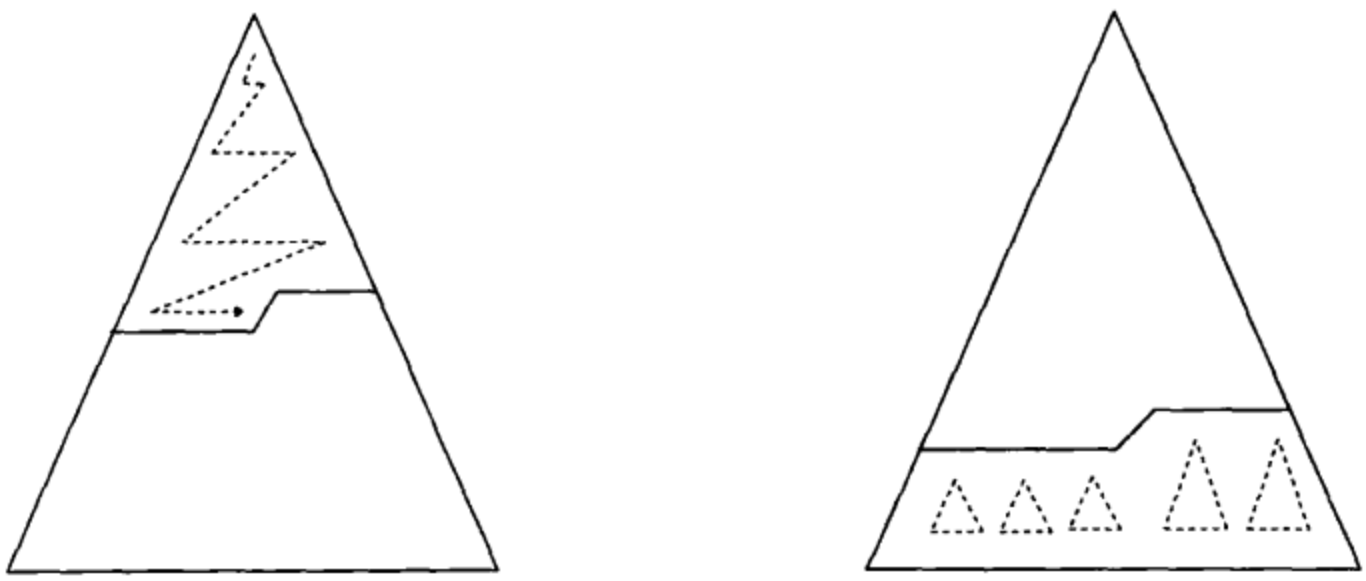


图 6.14　自顶向下和自底向上建堆

先考虑从左到右对数组进行扫描（对应自顶向下建堆法）。

归纳假设（自顶向下）：数组 $A[1..i]$是堆。

归纳假设是显而易见的，因为 $A[1]$本身一定是堆。算法的主要部分在于如何把 $A[i+1]$放到堆 $A[1..i]$中，也就是如何在堆中插入 $A[i+1]$（参见第 4 章）。将 $A[i+1]$与其父节点比较，然后不断交换节点的位置，直到新的父节点大于其自身。最坏情况下的比较次数是$\lfloor \mathbf{lb}(i+1) \rfloor$。

再考虑如何从右到左对数组进行扫描（对应自底向上建堆法）。我们想要说明数组 $A[i+1..n]$ 是堆，然后如何在堆中插入 $A[i]$。可是 $A[i+1..n]$并不代表某个堆，它对应着数个堆。（这里把 $A[i+1..n]$看做是由 $A[i..n]$所表示的树的组成部分，而不仅仅是数组。）因此，归纳假设比刚才要复杂些。

归纳假设（自底向上）：$A[i+1..n]$代表的所有树都满足堆的性质。

$A[n]$本身是堆，于是归纳基础成立。下面的归纳可以更巧妙。整个数组 $A[\lfloor n/2 \rfloor + 1..n]$ 都表示树上的叶子节点，因此对应着 $A[\lfloor n/2 \rfloor + 1..n]$ 的树都是单节点树，于是它们自然满足堆的性质。只须从$\lfloor n/2 \rfloor$开始归纳。这就提示我们自底向上法或许更佳，毕竟能省去一半的归纳过程。（这同样也说明了谨慎选择归纳基础的重要性。）

接下来考察 $A[i]$，它最多只能有两个子节点（$A[2i]$和 $A[2i+1]$），且都是其他某个堆的根节点（由归纳假设推得）。这样，把 $A[i]$插入堆中就很容易了。$A[i]$先与它最大的子节点比较，如果有必要就和那个最大的子节点交换。这个过程类似于在堆中删除元素的操作（参见第4 章）。我们沿着树自上而下不断交换直到 $A[i]$原来的值大于其现在两个子节点的值。自下而上建堆过程由图 6.15 给出。由于 $A[i]$的高度是$\lfloor \mathbf{lb}(n/i) \rfloor$，故最坏情况下的比较次数是$2\lfloor \mathbf{lb}(n/i) \rfloor$。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
2	6	8	5	10	9	12	(14)	15	7	3	13	4	11	16	(1)
2	6	8	5	10	9	(16)	14	15	7	3	13	4	11	(12)	1
2	6	8	5	10	(13)	16	14	15	7	3	(9)	4	11	12	1
2	6	8	5	10	13	16	14	15	7	3	9	4	11	12	1
2	6	8	(15)	10	13	16	14	(5)	7	3	9	4	11	12	1
2	6	(16)	15	10	13	(12)	14	5	7	3	9	4	11	(8)	1
2	(15)	16	(14)	10	13	12	(6)	5	7	3	9	4	11	8	1
(16)	15	(13)	14	10	(9)	12	6	5	7	3	(2)	4	11	8	1

图 6.15 自底向上建堆的例子。上面的数均指下标，每一行中圈起的数是在当前一步被交换的

复杂度（自顶向下） 第 i 步至少需要 $\lfloor \mathbf{lb}i \rfloor \leqslant \lfloor \mathbf{lb}n \rfloor$ 次比较，故运行时间为 $O(n \log n)$，且 $O(n \log n)$并非高估，原因如下：

$$\sum_{i=1}^{n} \lfloor \mathbf{lb}i \rfloor \geqslant \sum_{i=n/2}^{n} \lfloor \mathbf{lb}i \rfloor \geqslant n/2 \lfloor \mathbf{lb}(n/2) \rfloor = \Omega(n \log n)。$$

复杂度（自底向上） 每一步交换次数至多是相应节点高度的两倍（因为每个节点可能与两个子节点都进行比较，然后再顺树而下逐个交换），所以算法复杂度最多是树上所有节点高度和的两倍。下面来计算这个值。首先考察完全树，记 $H(i)$是高度为 i 的完全二叉树所有节点的高度之和，可以导出一个关于 $H(i)$ 的递归关系式。不妨把高为 i 的树看做是两棵高为 i–1 的树再加上一个根节点。于是 $H(i) = 2H(i-1) + i$, $H(0) = 0$。可以（用归纳法）证明，该递归关系式的解是 $H(i) = 2^{i+1} - (i+2)$。因为高为 i 的完全二叉树有 $2^{i+1}-1$ 个子节点，可推得完全二叉树（即有个 2^k-1 节点的堆）自顶向下建堆的算法复杂度是 $O(n)$。为有 n 个节点的二叉树建堆，n 满足 $2^k \leqslant n < 2^{k+1}-1$，其算法复杂度不会超过 $2^{k+1}-1$ 个节点的堆，故仍为 $O(n)$。（更仔细的分析可以发现这个常数不会递增；参见习题 6.32）。自顶向下法比自底向上法快主要是因为树底部的节点多于顶部，所以优化对底部节点的处理其效果好于顶部。

这个例子再一次说明按不同顺序归纳会得到更好的算法。尽管自顶向下法更直接也更易理解，但自底向上法的性能更佳。

评述 很难只用一个段落来总结排序算法。在这一节中主要介绍的是分治法的各种变形。前面已经看到在“分”上多花一些功夫能使“治”更简便。按不同顺序归纳，特别是对某些特殊的子集而不是对任意元素归纳，也能解决问题。还应注意到算法的分析和设计必须同步进行，通过熟悉各种算法，逐渐培养在分析算法前对算法性能的直觉感受，这种直觉对我们探求高效算法大有裨益。真理往往（但并不总是）与直觉近在咫尺。

6.4.6 排序问题的下界

我们一开始介绍的排序算法是 $O(n^2)$的，然后逐渐改进到 $O(n \log n)$。是不是还能进一步改善其性能呢？寻找某个问题下界的过程就是证明不存在更好的算法来解决这个问题。这样的证明非常困难，因为仅仅证明某一种方法是不够的，只有列举出所有可能的算法才可完成证明。我们必

须定义一个能用于任意（非指定的）算法的模型，然后证明符合这个模型的一切算法的运行时间一定大于或等于问题下界。这一节我们会讨论其中一个这样的模型，称为决策树。决策树能模拟以比较为主的计算过程。它不是像图灵机、随机存取机那样的通用计算模型，因此用决策树来证明下界并不十分严格，然而，它在许多方面更简便也更容易应用。决策树有各种变形，不少现有的下界证明方法都用到了它。

我们把决策树定义为由两种类型的节点（即内部节点和叶子节点）构成的二叉树。每个内部节点都是一个查询，查询结果有两种可能，分别由该节点发散的两条边表示。每个叶子节点都是一种可能的输出。现在假设算法的输入是序列 $x_1, x_2, \cdots, x_n$。计算过程从根节点开始，每个节点由输入进行查询，根据查询结果选择左、右分支，直至到达某个叶子节点，把该叶子节点的输出作为整个计算过程的输出。树 T 在最坏情况下的运行时间是 T 的高度，相当于输入最多所需查询的次数。因此决策树事实上对应着算法，尽管它不能模拟所有的算法（比如说，决策树无法计算平方根），但是对于基于比较的算法而言，它的确是一个很合适的模型。由决策树所取得的下界意味着不存在比决策树更好、与它的形式相同的算法。下面就用决策树来证明排序问题的下界。

□ **定理 6.1**

任意一个排序算法的决策树高度为 $\Omega(n \log n)$。

证明：排序算法的输入是序列 $x_1, x_2, \cdots, x_n$，输出是排序后的序列。不妨把输出看做是输入的一个排列，即输出说明了如何重新调整输入的数的排列位置，使调整后的数按序排列。由于输入的顺序可能是任意的，所以每一种排列都是一个可能的输出。若算法对于所有可能的输入都能处理，则这样的算法是正确的。$(1, 2, \cdots, n)$的任意一种排列（重组）都应该被表示为决策树上的可能输出，而决策树上的所有输出都由叶子节点负责。由于不同的排列对应着各自不同的输出，因此它们必须被不同的叶子节点表示。n 个元素的全排列数为 $n!$，且前文中我们假设树是二叉树，故树的高度至少是 $\mathrm{lb}(n!)$。由斯特林公式

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + O(1/n)),$$

得 $\mathrm{lb}(n!) = \Omega(n \log n)$，定理得证。 □

这类下界被称为信息论下界，它与计算过程完全无关（注意刚才甚至没有定义内部节点允许的查询种类），只与输出信息量有关。本例中下界指的是所有排序算法在最坏情况下均需$\Omega(n \log n)$次比较，因为这些算法要区分 $n!$种不同的情况，而每次它们只能分辨出两种可能性。当然我们完全可以把树定义为每个节点有三个子节点（分别对应着“<”、“=”和“>”），这样树的高度至少是 $\log_3(n!)$，不过仍为$\Omega(n \log n)$。换句话说，无论决策树每个节点有几个子节点，下界总是$\Omega(n \log n)$。

问题下界的证明还说明了基于比较的排序算法不可能比$\Omega(n \log n)$快，不过仍可以通过利用关键字的特殊性质或者对关键字进行代数运算来加快排序速度。例如有 n 个取值范围从 1 至 $4n$ 的数，桶排序可以在 $O(n)$时间内将它们排列成有序列表。然而这并不与下界的结论矛盾，因为桶排序不进行比较，而是借助了数值可以用做地址（桶）这一特点。

讨论决策树时，我们一般不考虑树的规模，只关心树的高度。因此，即使最简单的线性时间算法也可能对应到有指数个节点的决策树。然而树的规模并非无关紧要，只不过我们不需要构造它，而仅把它作为证明下界的工具。忽略树的大小使证明更有力，并可用于指数规模的程序上。另一方面，这个方法或许过于强大，对于可用一般大小程序解决的问题的下界束手无策，反倒可用于用指数规模程序解决的问题的下界（比如用表列出所有可能解答的程序）。决策树是一种不

一致的计算模型，取决于 n，即输入的大小。我们可以根据不同的 n 构造出不同树。这不是天方夜谭，恰恰说明了可能花费指数时间完成的问题可以通过构造多项式高度、但却是指数规模的决策树加以解决。这未免有些太乐观了。也就是说，决策树的下界可能远远低于问题的实际复杂度。另一方面，如果问题的下界和某个算法的上界相等——就像这个排序问题——那么这个下界就意味着无论用多大的空间，算法都不可能进一步优化。

有趣的是，任何基于比较的算法的平均运行时间都是Ω (n log n)。证明相当复杂，此处从略（可参见 Aho, Hopcroft, and Ullman [1974]）。

6.5 顺序统计

已知序列 $S = x_1, x_2, \cdots, x_n$。若 x_i 是 S 中第 k 小的数，则称 x_i 在 S 中的秩是 k。通过排序，很容易就能知道每个数在序列中的秩。然而有些关于秩的问题不必排序也能解决。本节中我们就来研究这一类问题。首先从查找最大数和最小数开始，然后逐渐推广到查找第 k 小的数。

6.5.1 最大数和最小数

查找序列中的最大数和最小数是很方便的。如果知道前 n–1 个数构成的序列中的最大数，则只须将这个最大数同第 n 个数做比较，从而得到这 n 个数中的最大数（查找 1 个数构成的序列中的最大数是显然的）。上述过程从第二个数开始，每个数比较一次，因此总共需要比较 n–1 次。

假设现在要同时找到最大数和最小数。

> **问题** 在已知的某个序列中查找最大数和最小数。

最直接的办法就是分别查找最大数和最小数。这样一共要做 $2n$–3 次比较：n–1 次找到最大数，n–2 次找到最小数（因此此时可以排除最大数）。还有更好的方法吗？再一次考虑用归纳的方法。假设我们知道 n–1 个数时问题的解，现在求 n 个数时的解（归纳基础显然成立），那么就必须把第 n 个数与先前找到的最大数和最小数分别进行比较。这样，每个数需要做两次比较，可以推得总共仍需要 $2n$–3 次，因为第一个数不用比，第二个数只比一次。以不同顺序扫描序列中的数也不能改进其性能，这主要是由于序列中数的位置与问题是无关的。

接着我们尝试把解决方案扩展到每次处理多于一个数。先试试看每次两个数，即已知 n–2 个数的解，对 n 个数求解。（为使归纳过程完整，必须用两个归纳基础，$n = 1$ 和 $n = 2$，这样，归纳过程就能包括所有的自然数。）考察 x_{n-1} 和 x_n，同时令 MAX（min）是前 n–2 个数的最大（小）数（由归纳法可以得到）。易见，做三次比较便能得出新的最大数和最小数。首先比较 x_{n-1} 和 x_n，然后将其中大数同 MAX 比较，小数同 min 比较。这样一来，总共只要大约 $3n/2$ 次比较，而不是先前的 $2n$ 次。那么每次三个（或四个）数是不是能更进一步提高算法的性能呢？结果和刚才一样，仍要比较这么多次。可见，无论用什么办法都不能减少比较次数了。有趣的是，用分治法解决这个问题最终也需要约 $3n/2$ 次比较（见习题 6.14）。

6.5.2 查找第 k 小的数

下面来看这个更一般的问题。

> **问题** 已知序列 $S = x_1, x_2, \cdots, x_n$ 以及整数 k，$1 \leqslant k \leqslant n$，试查找 S 中第 k 小的数。

这个问题被称为顺序统计或选择问题。如果 k 接近 1 或 n，那么只要把上述查找最小（大）

数的算法运行 k 次便能找到第 k 小的数。这个方法大概需要做 kn 次比较。排序算法一般都比这个原始的算法好，除非 k 是 $O(\log n)$或者 $n - O(\log n)$。然而，还有另一个算法对任意 k 都能高效地找到第 k 小的数。

除了一个子问题有待解决以外，其余的解题思想大致都与快速排序相同。在快速排序中，序列被支点分割成两个子序列，然后分别对这两个子序列递归排序。现在我们先要确定第 k 小的数在哪个子序列中，然后只要对那个子序列递归排序就行了。不必考虑其余的数。算法由图 6.16 给出。

```
算法 Selection (X, n, k)
输入：X（下标从 1 至 n 的数组）和 k（某个整数）
输出：S（第 k 小的数；数组 X 也做过变动）
begin
    if (k < 1) or (k > n) then print "error"
    else
        S := Select(1, n, k)
end

procedure Select (Left, Right, k) ;
begin
    if Left = Right then
        Select := Left
    else
        Partition (X, Left, Right) ; {见图 6.9}
        Let Middle be the output of Partition ;
        if Middle − Left + 1 ⩾ k then
            Select (Left, Middle, k )
        else
            Select (Middle + 1, Right, k − (Middle − Left + 1))
end
```

图 6.16　算法 *Selection*

复杂度　快速排序中，选择不合适的支点会得到平方级的算法。由于算法只对一个子问题递归调用，其运行时间一定小于快速排序。算法的平均比较次数是 $O(n)$，证明从略。最坏情况下仍能在 $O(n)$步内找到第 k 小的数，而在实际应用中，图 6.16 给出的算法还是相当高效的。

评述　顺序统计在大多数情况下用于查找中数，即第 $n/2$ 小的数。算法 *Selection* 是一个很成功的中数查找算法，而且最为简洁。也就是说，把中数查找算法扩展为查找任意第 k 小的数能使其简化。由于算法对任意的 k 都进行递归，这也说明了加强归纳假设很有必要。

6.6　数据压缩

数据压缩是节省存储空间的重要技术。一般把待压缩的文件看成字符串，压缩后的文件越小就越好，但必须能从中还原出压缩前的文件。数据压缩技术十分有用，比如有些很少访问的文件，其存储代价远远高于压缩和解压的开销。数据压缩技术在通信方面也很重要。有时发送文件的代价比处理文件本身还要大。数据压缩技术应用广泛，发展至今已相当成熟。本节中只介绍数据压缩技术中针对某方面的一个算法。

为简单起见，假设文件是英语字母序列。26 个字母分别用各不相同的位串来表示，这个位串被称为字符编码。如果所有字符的编码长度相等（大多数情况下的确如此），那么文件的字符总长度就只与文件的字符数有关。另一方面，我们也可以将出现频率高的字符（如 *A*）用短位串编码，而对于出现频率低的（如 *Z*）用长位串。例如，在 ASCII（美国信息互换标准代码）中，所有字符都用 7 位位串表示。A 用 1000001 表示，B 用 1000010 表示，以次类推。（一共可以表示 128 种字符，包括小写字母和特殊字符。）单词 “AND”（还有其他三个字母的词）要用 21 位表示。如果我们把 A 的表示改为 1001，那么每出现一次 A 就能节省 3 位。然而，有些编码集是无效的，可能会带来二义性。比方说，我们不能同时将 A 编为 1001，M 编为 1001101，否则遇到 1001 时将无法判断究竟是 A 还是 M 的一部分。当然，我们可以用分隔符把字符分开，不过分隔符本身也需要一个编码。一般来说，某个字符编码的前缀不能是另一个字符的完整编码。我们把这个限制称为前缀限制。缩短某个字符的编码必然意味着增加其他字符的编码。现在的问题就是在假设已知每个字符出现频率的条件下取得一个平衡。

> **问题** 已知某个文本文件（字符串序列），为其设计一种编码方式使之满足前缀限制且使编码后的文本长度最短。

首先，要统计每个字符在文本中出现的次数，这个数被称为字频。（一般使用的是对典型文本进行统计而得的标准字频表，而不是为某个文本进行单独计算所得到的精确的字出现频率。）记字符为 $C_1, C_2, \cdots, C_n$，相应字频为 $f_1, f_2, \cdots, f_n$。若在某种编码方式 E 中，用长度为 s_i 的位串 S_i 表示 C_i，文件 F 用编码方式 E 编码后的长度为

$$L(E, F) = \sum_{i=1}^{n} s_i \cdot f_i。$$

现在的目标是设计一种满足前缀限制的编码方式 E，且 $L(E, F)$最小。

前缀限制使解码不产生二义性，下面就来看一下解码过程。按序扫描序列的每一位直至得到对应于某个字符编码的位串。考察这样一棵二叉树，每个节点不是有一条标着 0、另一条标着 1 的两条发散边，就是没有发散边。树的每个叶子节点对应着一个字符，从根节点到叶子节点的路径就是这个字符的编码（见图 6.17）。当扫描文件的过程中到达叶子节点时，我们就能放心地确定相应的字符。现在的问题就是如何构造这棵树，使 $L(E, F)$最小。树的表示方法对解题并无大碍，用图来表示问题（及其限制）对解题倒是颇有好处。

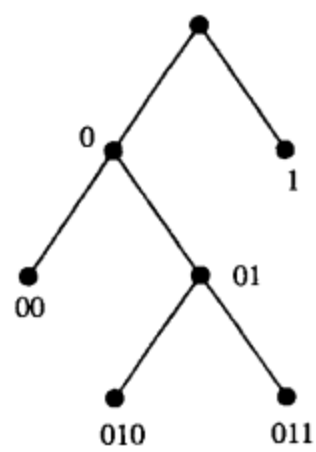


图 6.17 编码的树形表示

算法的基础就是把 n 个字符的问题归约到 $n-1$ 个字符的问题（归纳基础显然成立）。同样，主要的难点还是在于如何定义归纳假设以及按什么顺序逐个删去字符。这次归纳与我们已经介绍

过的不同，并不是简单地把每个考察过的字符删去，而是用一个“人造”字符代替原来的两个已有字符。这样做比较复杂，但目的不变，即缩短输入的长度。令 C_i 和 C_j 是字频最小的两个字符（如果不止两个，则任意断开）。可以证明，存在一棵树使 $L(E, F)$最小，且这些字符都对应着离根节点最远的叶子节点。否则，若存在某个字频更高的字符在树上的位置低于这些字符，那么它就可以与 C_i 或 C_j 交换从而减小 $L(E, F)$。（若字频相等，交换后 $L(E, F)$仍不变。）因为树上每个节点不是有两个子节点就是无子节点（否则可以使树更短），于是断定 C_i 和 C_j 是两个并列的节点。我们用一个新的字符代替 C_i 和 C_j，这个新字符记做 C_{ij}，其字频为 f_i+f_j。

现在，问题有 $n-1$ 个字符（$n-2$ 个原来的和一个新的），这样就可以用归纳假设解决了。通过将归约后问题中的内部节点用两个替换过 C_{ij} 的叶子节点 C_i 和 C_j 还原，就可以得到原来问题的解。最优性证明留做练习。

实现　霍夫曼编码的步骤是：（1）插入某个数据结构，（2）从该数据结构中删除字频最小的两个字符，（3）构造霍夫曼树。前两步用堆作为数据结构比较合适，每一步在最坏情况下都需要 $O(\log n)$。算法由图 6.18 给出。这种压缩技术被称为霍夫曼编码，根据提出这个算法的 D·霍夫曼[1952]命名。

```
算法 Huffman_Encoding (S, f)
输入：S（字符串）和 f（字频数组）
输出：T（S 的霍夫曼树）
begin
    insert all characters into a heap H according to their frequencies ;
    while H is not empty do
        if H contains only one character X then
            make X the root of T
        else
            pick two characters X and Y with lowest frequencies
                and delete them from H ;
            replace X and Y with a new character Z whose frequency is
                the sum of the frequencies of X and Y ;
            insert Z to H ;
            make X and Y children of Z in T {Z 仍没有父节点}
end
```

图 6.18　算法 *Huffman_Encoding*

□ 例 6.1

设一段数据由 6 个字符 A、B、C、D、E、F 组成，其字频分别为 5、2、3、4、10、1。它们的霍夫曼树由图 6.19 给出。内部节点根据生成时间编号。□

复杂度　构造树的过程中，每个节点都用常数时间。插入和删除操作分别用了 $O(\log n)$步。因此，算法的总运行时间是 $O(n \log n)$。

6.7　串匹配

令 $A=a_1 a_2 \cdots a_n$，$B=b_1 b_2 \cdots b_m$（$m \leqslant n$）是两个字符串，其中字符取自一个有限集合。（为方便起见，可以把这些字符看成是英文字母，尽管并不必要。）字符串 A 的子串是取自 A 的连续字符序列 $a_i a_{i+1} \cdots a_j$。这里把特殊子串 $a_1 a_2 \cdots a_i$($b_1 b_2 \cdots b_i$)记为 $A(i)$ ($B(i)$)。

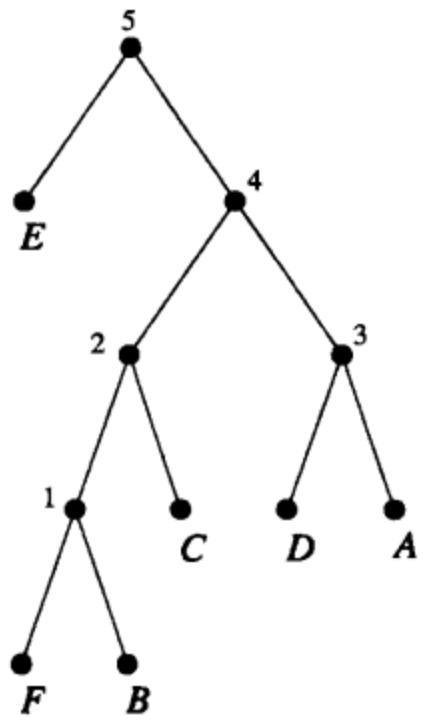


图 6.19 例 6.1 的霍夫曼树

问题 已知字符串 A、B，在 A 中查找 B 出现的第一次（如果有的话）。也就是说，确定最小的 k，使得对任意 i，$1 \leqslant i \leqslant m$，有 $a_{k+i}=b_i$。

这个问题最明显的一例便是在文本中查找某个词或者模式[①]。任何一个文本编辑器都有查找模式的功能。这个问题还有其他应用领域——包括分子生物学中，在 RNA 或 DNA 大分子中查找某些模式。

这个问题看上去很简单。在 A 中匹配 B 只要从 A 的第一个能匹配 b_1 的字符开始，然后依次比较（与 b_2 以及以后的字符比较）直到完全匹配或者中途遇到不匹配的字符。如果遇到了不匹配的字符，就必须开始重新进行匹配。图 6.20 是贯穿本节的说明这个问题的例子。

$A=xyxxyxyxyyxyxyxyyxyxyxx$，　$B=xyxyyxyxyxx$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x
1:	x	y	x	y	·	·	·																
2:		x	·	·	·																		
3:			x	y	·	·	·																
4:				x	y	x	y	y	·	·	·												
5:					x	·	·	·															
6:						x	y	x	y	y	x	y	x	y	x	x							
7:							x	·	·	·													
8:								x	y	x	·	·	·										
9:									x	·	·	·											
10:										x	·	·	·										
11:											x	y	x	y	y	·	·	·					
12:												x	·	·	·								
13:													x	y	x	y	y	x	y	x	y	x	x

图 6.20 字符串直接匹配的例子

① 至少对我来说是最明显的，手头正在编辑的就是文本文件。

例子中 $A=xyxxyxyxyyxyxyxyyxyxyxx$，$B=xyxyyxyxyxx$。第一次因 $a_4 \neq b_4$，在 a_4 处不匹配。那么，重新从 a_2 开始，比较 b_1 和 a_2，立刻发现它们不匹配。接着从 a_3 开始可以匹配，但马上发现 $a_4 \neq b_2$，因此匹配失败。再接下去似乎有些希望，从 a_4 到 a_7 都能匹配，但 a_8 再次匹配失败。现在不得不返回到 a_5，然后比较 b_1 和 a_5（不匹配），再比较 b_1 和 a_6，如此进行下去。最终，从 a_{13} 开始终于找到了一个完全匹配。在这个过程中，我们可能会回溯很多次，然后每次都重新开始匹配，导致在最坏情况下的比较次数可能是 $O(mn)$，请注意，上述过程有许多冗余的工作，如在匹配中重复两次发现了 A 中从 a_{11} 开始的子模式 $xyxy$（第 6 行和第 11 行）。而在文本文件中查找单词时，由于不匹配往往很早就被发现，回溯的发生次数就很少。对这些应用而言，这个算法还是相当不错的。而在其他一些应用中，当字符表很小且模式中有许多重复字符时，回溯的次数就会多很多。上面给出的算法可能在同一位置反复匹配某个子串，因此我们想探索其他的算法尽量避免这种情况。目前有待解决的就是如何利用已经掌握的信息，将其有效地用于后来的匹配过程中。

要改进这个直接的算法，必须首先理解算法的性能瓶颈。刚才讨论的那个例子性能较差的原因主要在于回溯。若模式是 $yyyyyx$，文本是 $yyyyyyyyyyyyx$，就会导致性能变差。我们不得不先把模式中的 5 个 y 与文本一一比较，发现与 x 不匹配，再后移一位，然后继续重复进行 4 次比较。（就这个例子来说，处理起来并不难，但它主要说明了一个普遍的问题。）另一方面，如果模式为 $xyyyyy$，匹配起来只须找到一个 x 后跟着 5 个 y。即使 y 的数目不够也不必回溯，继续再找 x 就行了，查找过的 y 是无用的。这种情况用直接匹配的算法是相当合适的，由于无须回溯，运行时间是线性的。

现在回到最初的那个模式 $B=xyxyyxyxyxx$。假设扫描到 B 的第五个字符时（如图 6.20 第 4 行中与 a_8 的比较）发生了不匹配，那么 A 中的前两个字符一定是 xy（因为它们是已匹配的字符）。而 xy 又是字符串 B 的开始，那么只要把 B 右移一段，然后用 A 当前的字符和 B 中间的某个字符比较（考虑到先前已匹配过的部分）。只要不漏掉可能的匹配，把 B 往右移得越远越好（可以减少比较次数）。这个例子中因为已知 b_1 和 b_2 已经匹配，于是把 B 右移两位，然后继续把 A 中刚才不匹配的当前字符（这里是 a_8）同 b_3 比较。（事实上，这正是后来要做的，在图 6.20 的第 6 行，只不过之前还做了三次多余的比较，第 5 行的 x 和第 6 行开头的 xy。）要注意的是，刚才讨论中并没有涉及到 A 的移动，只知道 A 中的最后几个字符当前与 B 匹配。

后面的讨论其实并不局限于文本中只有两种字符，尽管为简单起见，例子中用到的字符仍只有两种。而只有两种字符的情况下，算法将会更优（习题 6.45 专门讨论了这个问题）。

继续匹配会出现另一种情况。图 6.20 中第 6 行的不匹配发生在 B 的最后一个字符 b_{11}，那么这次可以更往右移。考察 B 的子串 $B(10)=b_1 b_2 \cdots b_{10}$。已知 $B(10)$和 A 当前字符开始的前 10 个字符相同，即 $B(10)=A[6..15]$，因为它们刚被匹配过。我们想要确定的是究竟应该再向右移几位，而又不会错过任何一次可能的匹配。这就得计算 $B(10)$中能和 B 的某个前缀匹配的最大后缀。本例中，满足条件的后缀长度是 3（xyx），见图 6.21。在这个长度以内，如果 B 每次右移一位，就是和自身比较，直到后缀和前缀相匹配。（最后一位 b_{11} 会造成不匹配，因此忽略。）我们现在知道 $B[1..3]=B[8..10]$，接着可以继续比较 a_{16} 和 b_4，直到完全匹配。我们在第 7 行至第 12 行中保存了所有的比较，第 13 行保存了一半。图 6.21 和图 6.20 的惟一区别就是图 6.21 中的信息只与 B 有关。这一点十分重要，因为我们能预先把 B 处理一遍，无论 A 是什么，都能记录下将来匹配时所需的信息。这样我们就能利用图 6.20 第 6 行中的所有匹配，在后续过程中将不必重复。

算法改进后最重要的部分就是对 B 的预处理。我们会仔细检查 B 中每个重复的模式，然后设计一个即使发生不匹配也不必回溯的算法。具体方法如下：对字符串 A 永远向前扫描，不必回溯，尽管 A 中同一个字符可能与 B 的多个字符比较（不匹配时）。而当遇到不匹配时，可以通过查表

决定 B 应该回溯几位，这张表记录着 B 中每个字符在不匹配情况下的回溯情况（即移动的位数）。快速构造这张表的方法稍后将介绍。首先对表进行精确的定义，然后再介绍如何将这张表用于字符串匹配的问题中。

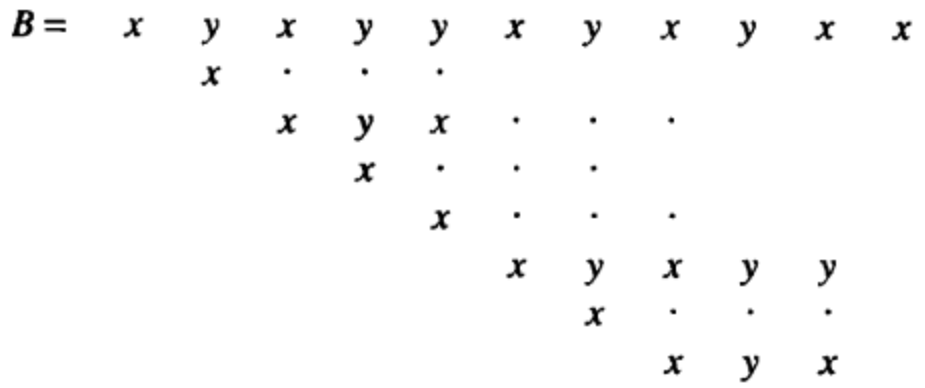


图 6.21 模式自身的匹配过程

下面就来解释构造表的思想。对每个 b_i 都要计算出与 B $(i-1)$的前缀相等的最大后缀。如果这个后缀的长度是 j，则 A 中不匹配的那个字符就可以跳过多余的比较过程直接跟 b_{j+1} 比较，因为之前已经知道 A 中最近匹配的 j 个字符能与 B 的前缀字符串匹配。不仅如此，由于我们得到的后缀是能与前缀匹配的最大后缀，所以 B 再往左一位就不能跟 A 匹配了。这个表被称为 *next*，表中的项精确定义如下：

next (i)等于使 $b_{i-j}b_{i-j+1}\cdots b_{i-1}=B(j)$的最大的 j（$0<j<i-1$），如果不存在这样的 j，则返回 0。

为了区别对待 i 取不同值的情况，可以简单地定义 *next* $(1)=-1$，显然还有 *next*(2)总是等于 0（不存在 j 满足 $0<j<2-1$）。图 6.21 中模式 B 的 *next* 表项的值由图 6.22 给出。我们可以非常轻松地在 $O(m)$时间内计算出所有 *next* 值。现在假设已知这些 *next* 值，下面先来看匹配过程，之后再介绍如何计算 *next* 值。

$i=$	1	2	3	4	5	6	7	8	9	10	11
$B=$	x	y	x	y	y	x	y	x	y	x	x
next $=$	−1	0	0	1	2	0	1	2	3	4	3

图 6.22 *next* 表中的值

匹配过程如下：A 的字符分别与 B 的字符一一比较，直到出现不匹配。假设不匹配发生在 b_i，首先查 *next* 表，然后把 A 的当前字符同 $b_{next(i)+1}$ 比较（前 *next* (i)个字符已经匹配）。如果仍然不匹配，那么再同 $b_{next(next(i)+1)+1}$ 比较，以次类推直到匹配为止。惟一的例外就是如果最后与 b_1 不匹配，那么就只能在 A 中继续了。这个例外情况可以由 *next*(1)这个特殊值−1 决定。字符串匹配的程序由图 6.23 给出。

最后的问题就是如何计算 *next* 表的值。归纳过程如下：已知 *next* $(2)=0$，作为归纳基础。假设已经算出 $1, 2, \cdots, i-1$ 的 *next* 值，下面讨论如何计算 *next* (i)。最佳情况下 *next* (i)等于 *next* $(i-1)+1$，即 $b_{i-1}=b_{next(i-1)+1}$ 时。换句话说，与前缀相同的最大后缀被 b_{i-1} 扩展。这种情况比较简单。如果 $b_{i-1}\neq b_{next(i-1)+1}$，这就相对难一些了。首先要重新确定与前缀相等的新后缀，已知 $B(i-2)$的最大后缀与 $b_1b_2\cdots b_{next(i-1)}$匹配（见图 6.24），而 $b_{i-1}\neq b_{next(i-1)+1}$ 恰恰意味着在 $b_{next(i-1)+1}$ 处发生了不匹配。刚才已经讨论过如何处理这样的情况。如果在下表 j 处不匹配，那么就去查 *next* (j)。这样，在 *next* $(i-1)+1$ 处不匹配自然就去查 *next* (*next* $(i-1)+1$)，即试着匹配 b_{i-1} 和 $b_{next(next(i-1)+1)+1}$。如果它们能够相配，那么 *next* (i) = *next* (*next* $(i-1)+1)+1$，否则继续执行上述查找，直到匹配或者回到开始。

算法 *String_Match* (*A*, *n*, *B*, *m*)
输入： *A*（长度为 *n* 的字符串）和 *B*（长度为 *m* 的字符串）
　　{设 *next* 值已经算好，见图 6.25}
输出： *Start*（使得 *B* 是 *A* 中从 *A* [*Start*]开始的子串的第一个下标）

```
begin
    j := 1 ; i := 1 ;
    Start := 0 ;
    while Start = 0 and i≤n do
        if B [j ]=A [i ] then
            j := j + 1 ;
            i := i + 1
        else
            j := next [j ] + 1 ;
            if j = 0 then
                j := 1 ;
                i := i + 1 ;
        if j = m + 1 then Start := i – m
end
```

图 6.23　算法 *String_Match*

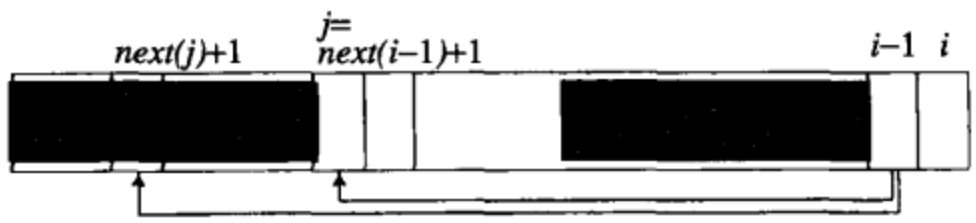


图 6.24　计算 *next*(*i*)

□ **例 6.2**

设 $B = xyxyyxyxyxx$（见图 6.21），试计算 *next* (11)。首先查 *next* (10)得 4，然后比较 b_{10} 和 b_5。如果相等，那么与后缀相等的前缀就是 5，但很遗憾它们不等。所以在 b_5 处不匹配，于是查 *next* (5)得 2。接着再比较 b_{10} 和 b_3，这次它们相等。于是 *next* (11) = 3，这很容易手工验证。 □

计算 *next* 表的算法难以理解，不过其实现倒不算困难。图 6.25 给出了程序。

算法 *Compute_Next* (*B*, *m*)
输入： *B*（长度为 *m* 的字符串）
输出： *next*（大小为 *m* 的数组）

```
begin
    next (1) := –1 ;
    next (2) := 0 ;
    for i := 3 to m do
        j := next (i – 1) + 1 ;
        while b_{i–1} ≠ b_j and j > 0 do
            j := next (j) + 1 ;
        next (i) := j
end
```

图 6.25　算法 *Compute_Next*

复杂度 A 的某个字符可能要与 B 的数个字符比较。如果一次比较结果不匹配，则 A 的这个字符就与 B 的 *next* 表中指向的字符比较。如果结果仍然不匹配，就需要继续查表直到两者匹配或者遇到 B 的首字符。不过即便如此，算法的复杂度仍是 $O(n)$。那么对 A 的某个字符，比方说 a_i，究竟要回溯多少次呢？假设第一次不匹配发生在 b_k 处，由于每次回溯都往回移一位，因此最多回溯 k 次。而要扫描到 b_k，实际上已经在无回溯的情况下向前移了 k 位。那么如果把回溯的开销算在前移上，则最多只要把前移的开销加倍。这个问题中最多前移 n 次，故比较次数是 $O(n)$。

这个算法由 Knuth, Morris, and Pratt [1977]给出，称之为 KMP 算法。解决这个问题的另一个快速算法由 Boyer and Moore [1977]给出。这里简要解释一下。这两个算法的区别在于 Boyer-Moore 算法对字符串是从后往前扫描，而不是从前往后。也就是说，第一次比较 b_m 和 a_m。如果两者匹配，则接下来比较 b_{m-1} 和 a_{m-1}，以次类推。如果不匹配，则与 KMP 算法类似，将整个模式向右移动数位。例如，若 a_m = "Z"，且 Z 不是 B 中的字符，那么将整个模式右移 m 位，然后比较 a_{2m} 和 b_m。若 Z 是 B 中的字符，不妨设 b_i 是 Z，那就移动 $m-i$ 位。如果出现多个部分匹配，则确定移动的位数就会变得相当复杂。一方面我们想要利用已经找到的匹配信息，另一方面，移动的位数越多越好，即使出现重复的比较。这部分细节从略。有趣的是，算法的比较次数有可能小于 n（对一般文本而言），这是因为不匹配会导致算法不做任何比较而直接右移 m 位。

6.8 序列比较

序列比较问题因其在分子生物学上的应用，近年来引起了人们不少的关注。这里只介绍其中的一个问题，把一个序列变成另一个序列的最少修改步数。本节主要用到动态规划方法（见 5.10 节的讨论）。

令 $A = a_1 a_2 \cdots a_n$, $B = b_1 b_2 \cdots b_m$ 是两个字符串，其中字符取自一有限集合（如英语字母表）。我们想一一改变 A 的字符把 A 变成 B。改变过程可以有三类变化（即修改方法），每种变化的代价设为1：（1）插入，把某个字符插入字符串；（2）删除，从字符串中删除某个字符；（3）替换，把某个字符替换成另一个字符。比方说，把 *abbc* 变成 *babb* 就可以首先删除第一个 *a*，得字符串 *bbc*，然后在两个 *b* 中间插入一个 *a*（*babc*），接着把最后一个 *c* 替换成 *b*，一共需要修改三步。然而，我们还可以先在字符串前面插入一个新的 *b*（得 *babbc*），然后再把最后一个 *c* 删除，这样总共只需两步修改。目标就是要尽可能减少单个字符的修改次数。

序列比较问题还可应用于文件比较和版本维护，比如有一个文本文件（或者程序）修改前后的版本，而需要能方便地抽取其中的不同部分。同一程序也可能有多个版本，如果版本之间相差不大，则在存储时只需要存储有差别的部分，而不必存储整个文件。这些情况下可能只用到插入和删除操作，对于其他情况则需要为不同的修改付出不同的代价。

通常情况下把一个字符串变成另一个往往有多种方法，究竟哪个最佳还难以定论。下面就通过归纳法来探索这个问题的解。记前缀子串 $a_1 a_2 \cdots a_i$ ($b_1 b_2 \cdots b_i$) 为 A (i) (B (i))，问题转换为用最少的修改步数将 A (n) 改为 B (m)。假设通过归纳已知将 A ($n-1$) 改为 B (m) 的最佳方法（可能有多种不同的最佳方法，这里假设已知其中一种），那么只要把 a_n 删除就可将 A (n) 改为 B (m)，而这样未必最佳，有可能用 b_m 替换 a_n 更好，甚至还有可能 a_n 本身就与 b_m 相同。

必须考虑在对 A、B 改动最小的条件下构造出修改次数最少的步骤的所有可能性。记 C (i, j) 为 A (i) 变成 B (j) 的最小代价，那么要关注的就是这个最小代价，而不是改动本身。我们感兴趣的是如何建立 C (n, m) 和某个 C (i, j) 之间的关系，这里 i、j 小于 n、m。不难发现，有 4 种可能性分别对应着 3 种操作和 1 种匹配：

删除：如果删除 a_n 就是 A 到 B 的最小改动，那么上述情况完全成立。于是最佳方法就是先把 $A\,(n-1)$ 变成 $B\,(m)$，然后再删除这个字符。换句话说，$C\,(n, m) = C\,(n-1, m) + 1$。

插入：如果 A 到 B 的最小改动需要插入某个字符与 b_m 匹配，那么就有 $C\,(n, m) = C\,(n, m-1) + 1$。也就是（通过归纳）先把 $A(n)$ 以最小代价变成 $B(m-1)$，然后再插入一个与 b_m 相同的字符。

替换：如果用 a_n 替换 b_m，那么首先要找到从 $A\,(n-1)$ 到 $B\,(m-1)$ 的最小改动，接下来若 $a_n \neq b_m$，还要加 1。

匹配：如果 a_n 和 b_m 相同，则 $C\,(n, m) = C\,(n-1, m-1)$。

记

$$c(i, j) = \begin{cases} 0 & \text{如果 } a_i = b_j \\ 1 & \text{如果 } a_i \neq b_j \end{cases}。$$

现在把上面 4 个式子整理为如下递归关系式：

$$C(n, m) = \min \begin{cases} C(n-1, m)+1 & （删除 a_n） \\ C(n, m-1)+1 & （插入与 b_m 匹配的某个字符） \\ C(n-1, m-1)+c(n, m) & （替换或者匹配 a_n） \end{cases}$$

其中，对任意 i，$0 \leqslant i \leqslant n$，有 $C\,(i, 0) = i$；对任意 j，$0 \leqslant j \leqslant m$，有 $C\,(0, j) = j$。

不难证明，这几种情况覆盖了所有的可能性。考察 a_n，a_n 上一定有操作。它不是被删除（由第一种情况处理）就是与 B 中的某个字符匹配。匹配还进一步分为两种情况，一种是 a_n 直接与 b_m 匹配，即第三种或第四种情况，另一种是与 b_m 前的某个字符匹配，即 a_n 后必须插入某个字符。

这种方法的问题在于归纳次数过多！刚才已把一个规模为 (n, m) 的问题归纳到三个规模仅比原问题略小的子问题。如果对这三个子问题分别递归处理，那么每次有常数级的减少，这样会使整体工作量增加两倍，最终得到一个指数级的算法。幸运的事，这里并不需要分别处理每个子问题，因为我们可以看到，总的子问题数并不多。每个可能的子问题都要对某个 i 和 j（$0 \leqslant i \leqslant n, 0 \leqslant j \leqslant m$）计算 $C\,(i, j)$。于是这样的 i 和 j 一共有 nm 种组合，所以总的子问题数不会超过 nm。这和我们在背包问题（见 5.11 节）中所观察到的现象一致。可以用强归纳法来解决上述问题。也就是把所有小于 n 的问题都扩展到 n，而不是仅仅针对 $n-1$ 的问题。对于这个二维问题，相应地把所有规模小于 (n, m) 的子问题扩展到 (n, m)。小于 (n, m) 的意思是“至少有一个值小于其上界且另一个值不大于其上界的所有这样的 (i, j) 组合”。

如果知道所有子问题的解，那么强归纳法将是一个不错的选择。我们用这些解来构造一张表。先看图 6.26。计算 $C\,(i, j)$ 的值需要图中用阴影部分表示的另外三个值。我们希望以某种方式扫描这个矩阵，使得访问任何一块之前，其计算所依赖的其他三块都已被访问。这样一来按行访问（即从左到右逐行扫描）方式正好可以用在这里。以此种方法访问二维数组是一种动态规划方法。

实现　维护数组 $C\,[1..n, 1..m]$，数组的每一项 $C\,[i, j]$ 保存 $C\,(i, j)$ 的值。令 $M\,[i, j]$ 为得到 $C[i, j]$ 的最小值的最后一步（一次改动）。这里只定义最后一步是因为可以从最后一步开始回溯从而找到矩阵中的所有变化。这步改动是 *delete* (i)、*insert* (j) 或 *replace* (i, j) 中的一步。计算 $C[i, j]$ 需要预知 $C\,[i-1, j]$、$C\,[i, j-1]$ 以及 $C\,[i-1, j-1]$。根据得到 $C[i, j]$ 最小值的可能性可以进一步决定最后一步究竟是哪种。算法由图 6.27 给出。

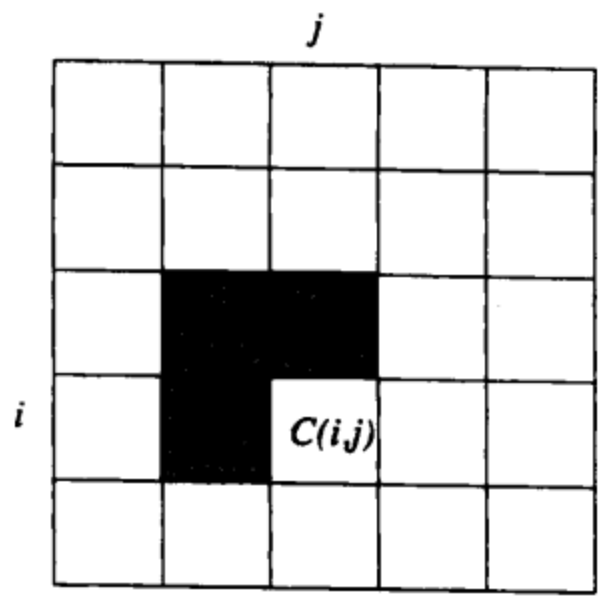


图 6.26 $C(i, j)$的依赖关系

算法 *Minimum_Edit_Distance* (A, n, B, m)
输入： A（长度为 n 的字符串）以及 B（长度为 m 的字符串）
输出： C（代价矩阵）

```
Begin
    for i := 0 to n do C[i, 0] := i ;
    for j := 1 to m do C[0, j] := j ;
    for i := 1 to n do
        for j := 1 to m do
            x := C[i-1, j] + 1 ;
            y := C[i, j-1] + 1 ;
            if a_i = b_j then
                z := C[i-1, j-1]
            else
                z := C[i-1, j-1] + 1 ;
            C[i, j] := min (x, y, z)
            {为 M[i, j]设置合适的值}
end
```

图 6.27 算法 *Minimum_Edit_Distance*

复杂度 显然，图 6.27 算法的复杂度为 $O(nm)$。它的主要缺点在于存储空间也是 $O(nm)$。

评述 当一个问题的解取决于其较小的子问题时，动态规划将十分有用。在动态规划中，往往是用表来存储已得到的结构，并以某种顺序（通常是按行访问）扫描整张表，从而导致平方级的运行时间。所以动态规划的性能一般不如分治法等其他方法。

6.9 概率算法

到目前为止，所讨论的算法都是确定的，每一步都是事先决定好的。如果某个确定的算法有两个相同的输入，则会得到两个相同的执行序列和结果。概率算法和它们不同，运行时不仅取决于输入，还与某些随机时间的结果有关。概率算法有各种变形，这里将介绍两种，先给出一个简单的例子，然后再介绍一种较正规的处理方法。

有数集 $x_1, x_2, \cdots, x_n$，从中选出“前半段”（即大于或等于中数）中的某个数。例如，想要根

据学生成绩选出一名“好”学生。一种方案就是选成绩最好的（当然，肯定是在前半段）。从刚才的讨论中已经知道这个方案需要 $n-1$ 次比较。另一种方案就是采用算法 *maximum-finding*，在到达中点时停止。此时那个数一定在前半段。这样还是要比较 $n/2$ 次。那么有没有更好的方案呢？不难证明，少于 $n/2$ 比较绝不可能得到那个位于前半段的数。这样看来似乎已经得到了最优算法。

如果我们必须保证得到的数位于前半段，那么这个算法的确是最优的。然而，有时候未必需要每次都能得到，以某个概率位于前半段也是可以接受的。例如，我们不能保证在散列过程中不发生冲突，只要发生冲突后能够进行处理就可以。（下面会看到散列法也是一种随机算法。）如果我们无须做这样的保证，那么寻找位于前半段的数就有更好的算法。从数集中随机取出两个数 x_i 和 x_j，$i \neq j$。不妨假设 $x_i \geqslant x_j$。从数集中随机取出的数有至少 1/2 的概率位于前半段（如果有些数与中数相等，则概率会大于 1/2）。因此，x_i 和 x_j 同时不在前半段的概率最多为 1/4。又因 $x_i \geqslant x_j$，于是这个概率就与 x_i 不在前半段的概率相等。因此，x_i 在前半段的概率至少是 3/4。

3/4 的概率通常是不够的，不过刚才介绍的原理可以很容易地扩展。我们随机选择 k 个数，然后取其中最大的数。同理可得，k 个数中的最大数位于前半段的概率至少是 $1-2^{-k}$。例如，若 k=10，则概率是 0.999。若 $k = 20$，则概率是 0.999 999。若 $k = 100$，在实际应用中，所得数不在前半段的概率高于一般编程失误、硬件故障或地震发生的概率，可忽略不计。现在我们就得到了一个算法，它几乎能百发百中地选出位于前半段的数，无论输入有多长，最多 100 次比较就可以得到最终结果。（假设随机选数能用一个操作完成；6.9.1 节将会讨论随机数的生成。）

有时这类算法也叫蒙特卡罗算法。它得出错误结果的概率非常小，而运行时间却比最优的确定性算法好。另一种概率算法虽然不会产生错误，但运行时间却没有保证。它可能很快终止，也可能运行相当长的时间。这类算法又被称为拉斯维加斯算法，在预期运行时间较短时较有用。6.9.2 节将介绍一种拉斯维加斯算法，用于解决着色问题。6.9.3 节将会引入一个优雅的方法将拉斯维加斯算法变换成确定性算法，然后将该方法用于6.9.2 节的着色问题，从而得到一个高效的确定性算法。不过并不是所有高效的拉斯维加斯算法都能用它转换成高效的确定性算法。

概率算法的思想直接类似于数学证明。用概率方法证明组合性质是一种强有力的方法。这个思想从本质上证明一个对象集合中的某个对象具有某种性质的概率大于 0，同时间接证明了存在某个具有该性质的对象。把这种方法用于算法，就是：假设欲查找一个具有某种性质的对象，若生成该对象，则它会以非零概率满足这种性质（这就是存在某个特定对象的概率证明）。借助这一概率证明，我们可以在恰当的时候生成随机事件，从而以某个正概率找到某个对象。不断重复以上过程最后一定能得到这个对象。如果概率方法有用，那么最终将获得高效的拉斯维加斯算法。

6.9.1　随机数

概率算法要求随机选数，因此我们必须找到一个能够高效地随机选数的算法。而任何一个确定性函数都会根据某个固定的模式产生数字，这取决于函数的步骤。如果这个模式完全确定，那么生成的数就不是真正的随机数了。它们之间具有相关性。幸运的是，实际上这并不是一个主要问题。实际应用中，伪随机数就足够了。这些数由确定性函数生成，因此不是真正的随机数，但用它生成的数对大多数程序而言是相互独立的。

对这个问题的深入讨论已超过了本书的范畴。这里仅仅介绍一种高效的方法，被称为线性同余法，用来生成随机数。第 1 步先要选一个整数种子 $r(1)$，常通过某些外部手段随机选取（例如以微秒为单位的当前时间，或者最喜欢的运动队的当前积分）。其余的数可由下式得到：

$$r(i) = (r(i-1) \cdot b + 1) \bmod t,$$

其中 b 和 t 是常数。b 和 t 的选择必须小心谨慎。Knuth [1981]提出了如下指导原则：t 应该相当大，至少是百万数量级，如果方便的话，最好是 2（或者 10）的幂。b 应该比 t 少一位数，且 b 的十进制表示的最后三位应该是 $x21$，其中 x 是偶数。这些（奇怪的）原则能够避免数的重复。由线性同余法生成的随机数的范围是 0 到 t–1。可以把这些随机数乘以一个因子来得到各种不同的范围（t 就应该是该范围中的某个乘积）。

6.9.2 着色问题

令 S 是有 n 个元素的集合，$S_1, S_2, \cdots, S_k$ 是 S 不同的子集，每个子集恰有 r 个元素，使得 $k \leqslant 2^{r-2}$。

> **问题** 用红、蓝两种颜色中的一种为 S 中的每个元素着色，使得每个子集 S_i 至少有一个蓝色元素和一个红色元素。

满足以上条件的着色方案被称为有效着色。研究发现，在以上集合条件下，总是存在一种有效着色。这里将介绍一种简单的概率算法，它是从有效着色的存在性概率证明演变而来的。算法几乎已经简单得不能再简单的。

> 任取 S 中的每个元素，随机着上红色或蓝色（1/2 的概率），每个元素的着色独立于其他元素。

这个算法显然有可能得不到有效着色。现在来计算一下不能有效着色的概率。S_i 所有元素都着红色的概率是 2^{-r}，那么 k 个子集中至少有一个子集元素全着红色的概率不超过 $k2^{r-2} \leqslant 1/4$（因 k 有条件约束）。因此随机染色不能有效着色的概率最多是 1/2（至少有一个子集元素全着蓝色的概率也不超过 1/4）。这就证明了总存在一种有效着色（否则不能有效着色的概率就应该是 1），另外还表明随机染色算法是很不错的。可以很容易验证某种着色方案是否有效，只要检查每个子集中元素的颜色，直到发现两种不同颜色的元素为止。成功率对半。如果失败，再重复一次即可。运行验证有效着色算法的期望次数是 2 次。显然这个算法属于拉斯维加斯算法，因为我们检查每一个元素的着色，在找到有效着色后终止。这是概率方法的一次简单应用，遗憾的是，一般概率算法并没那么简单，接下来我们要改变一下这个算法使它能够确定地找到一种有效着色。

6.9.3 将拉斯维加斯算法变换成确定性算法[①]

下面就来看如何通过归纳法把概率着色算法变换成一个确定性算法。这里介绍的方法并不适用于任一个拉斯维加斯算法。一般认为把每个拉斯维加斯算法都高效地变换为确定性算法并不可能。然而这个方法还是相当有意思的，因为它以一种强有力的方式体现了加强归纳假设的思想。最终得到的算法不仅高效、确定，同时除去了加在原始问题上的限制，可以适用于更一般的问题。

再次令 S 是 n 个元素的集合，$S_1, S_2, \cdots, S_k$ 是 S 不同的子集。概率算法基于这样一个事实，随机对每个元素着色得到有效着色的概率至少是1/2。假设我们能对某个元素着蓝色或红色，使得对其余元素随机着色仍能得到有效着色的概率不为 0。这样就可以对 n 进行归纳，从而得到算法。如果对某个元素着色后，有效着色的概率仍不为 0，那么就可以用归纳法对所有元素着色。

考虑到每次只处理一个元素，我们必须加强归纳假设从而不再强制每个子集大小相同。设 S_i 子集的大小为 s_i。S_i 只着一种颜色的概率是 2^{-s_i+1}，于是失败的概率（即对所有元素随机着色未能得到有效着色的概率）不超过

① 首次阅读的读者不妨跳过本节。

$$F(n)=\sum_{i=1}^{k}2^{-s_i+1}。$$

概率 $F(n)$应该是集合规模的函数，不过这里简写为 n 的函数。只要 $F(n)<1$，论证就是可靠的。归纳假设如下：

归纳假设：*在 $F(n)<1$ 的条件下，我们知道如何为小于 n 个元素的集合 S 着色。*

如果某个子集只有一个元素，那么 $F(n)$是 1，故 $F(n)$不可能小于 1。如果 $n=2$，那么由子集的互异性可知，只有一个子集有两个元素，于是其中一个元素着蓝色，另一个着红色。归纳基础由此建立。下面我们试着把 n 个元素的着色问题规约到 $n-1$ 个。

设 x 是 S 中任意一个元素。x 的着色有两种可能——蓝色或红色。不妨假设 x 着蓝色，那么对集合中其余（$n-1$）元素随机着色后得到有效着色的概率是多少呢？不含 x 的任意集合 S_i 的无效着色概率仍为 2^{-s_i+1}，而含 x 的任意集合 S_j 比不含 x 的少一个元素，而且还必须有一个元素着红色（已有一个蓝色的）。因此，S_j 无效着色的概率是 $2^{-(s_j-1)}$。注意，这个概率与 x 着色前的概率竟然是相同的。这是不是意味着我们已经得到了一个算法？事实并非如此，它仅仅说明第一次选择可以是随机的。选完后问题就改变了。

我们不能再用刚才的归纳假设了，因为第一个元素着色后，部分子集需要用两种颜色着色，部分子集只需要一种。归纳假设必须被加强以反映出这些变化。假定有些元素已被着色，那么此时某个子集必定处于下列 4 种状态之一：（1）子集中既有红色又有蓝色，这种情况无须进一步考虑；（2）子集中至少有一个红色元素，无蓝色元素，此时至少有一个未着色元素应着蓝色；（3）子集中至少有一个蓝色元素，无红色元素，此时至少有一个未着色元素应着红色；（4）子集中所有元素均未着色。子集若处在状态（2），就被称为红子集；处在状态（3）的被称为蓝子集；状态（4）的被称为中性子集。令 u_i 是 S_i 中未着色的元素数。若 S_i 处在状态（1），则它已经成功着色。若 S_i 是红子集或蓝子集，则随机着色无效的概率为 2^{-u_i}。若 S_i 是中性子集，则随机着色无效的概率为 2^{-u_i+1}。再记 f_i 为 S_i 随机着色无效的概率。我们必须保证如下性质成立：

$$F(n)=\sum_{i=1}^{k}f_i<1。\tag{6.1}$$

归纳假设必须反映出所有子集的状态，因此我们准备把问题扩展到任意的红、蓝或中性子集。换言之，当前的输入变成了红、蓝或中性子集，并假定满足条件（6.1）。

> **问题**　用红、蓝两种颜色中的一种为 S 中的每个元素着色，使得每个红子集至少有一个蓝色元素，每个蓝子集至少有一个红色元素，每个中性子集至少有一个红色元素和一个蓝色元素。

下面的归纳假设是问题（非直接）扩展后的直接假设。

归纳假设：*在满足条件（6.1）的前提下，我们知道如何为小于 n 个元素的集合 S 着色以满足问题的条件。*

归纳基础基本同前。给定满足条件（6.1）的有 n 个元素的集合 S，我们将为 S 中的某个元素着色，同时保证 S 仍然满足条件（6.1）。

同样，任取 $x\in S$。x 的着色仍有两种可能，结果导致集合进入两种不同的状态。若 x 着红色，则所有包含 x 的红子集不变（只是少了一个未着色元素），所有包含 x 的蓝子集着色成功（可以不

必进一步考虑），所有包含 x 的中性子集变为红子集。不含 x 的子集不变。若 x 着蓝色，同样有类似的结论。现在可以计算 $F(n-1)$的值了。若 x 着红色，则记为 $F_R(n-1)$；相应地，若 x 着蓝色，记为 $F_B(n-1)$。算法的关键之处就是下面这个引理。

□ 引理 6.2

设 $F(n)$是初始时无效着色的概率，$F_R(n-1)$是 x 着红色后无效着色的概率，$F_B(n-1)$是 x 着蓝色后无效着色的概率，则 $F_R(n-1)+F_B(n-1)\leqslant 2F(n)$。

证明：不含 x 的子集不变，对 $F_R(n-1)$、$F_B(n-1)$ 以及 $F(n)$所起的作用相同，与引理一致。下面考虑对含 x 的子集的影响。根据子集的状态，分三种情况。（1）红子集对 $F_B(n-1)$没有影响，因为此时子集已成功着色；对 $F_R(n-1)$的影响是对 $F(n)$的两倍，因为它比原先少了一个元素。同样，这仍与引理一致。（2）蓝子集的影响类似于红子集。（3）有 u_i 个元素的中性子集使 $F(n)$增加 2^{-u_i+1}，同时使子集减少一个未着色元素，并进入红子集或蓝子集的状态。这分别使 $F_R(n-1)$和 $F_B(n-1)$增加 $2^{-(u_i-1)}$。因此无论哪种情况对 $F(n)$、$F_R(n-1)$和 $F_B(n-1)$都会造成相同的影响。 □

从引理 6.2 可以直接得出算法。单个元素的归纳基础显然成立，因为满足条件（6.1）就说明只能有一个红子集或蓝子集包含这个元素。若只有一个子集，那么就着另一种颜色。如果 $F_R(n-1)+F_B(n-1)\leqslant 2F(n)$，则不是 $F_R(n-1)\leqslant F(n)$ 就是 $F_B(n-1)\leqslant F(n)$（或者都成立）。计算上述这些值，若 $F_B(n-1)$ 小于 $F_R(n-1)$，则 x 着蓝色，否则着红色。由引理 6.2，归纳假设中的条件（6.1）得到满足，故算法成立。算法的实现留给读者完成。

6.10 查找众数

令 E 是整数序列 $x_1, x_2, \cdots, x_n$。E 中 x 的重数（multiplicity）是 x 在 E 中出现的次数。如果某个数 z 的重数大于 $n/2$，则它就是 E 中的众数（majority）。

问题 已知一数列，判断是否存在众数。若存在，则求出众数。

例如用整数表示选举中的投票，那么这个问题就是要判定是否有人赢得了选举。如果候选人数很少，则用桶排序法就能在 $O(n)$时间内快速解决上述问题。而当候选人数很多时（目前选举的趋势）就不能用桶排序了。这里假定对候选人数没有限制，且可以用任意整数表示。选举也可以通过计算机系统进行。

这个问题很好地说明了有些解决方案尽管比较直接，但不太高效；有些方案比较优雅，但还要动一番脑筋，不过性能好，实现起来也更容易。首先我们讨论几种直接解法，然后再介绍这个优雅的算法。

最直接的解法是排序算法。一旦选票被排序后，统计每位候选人的得票数变得轻而易举。而排序算法在最坏情况下需要进行 $O(n \log n)$次比较，后面会看到这还能继续改进。也可以用中数查找法。如果众数存在，它肯定等于中数（中数是第$(n/2)$小的数，而众数的出现次数超过 $n/2$）。所以一旦找到中数，我们就能统计它出现的次数，如果中数不是众数，则众数不存在。由于查找中数比排序更简单，故中数查找法优于排序算法。另一种方法就是采用概率算法。我们从选票中取一小部分作为样本，找出样本中的众数，然后统计该数在全部选票中出现的次数。所以，尽管验证该数是众数将会很容易，但它无法证明众数不存在。这样一个算法的结果可能就是“未定的”。（这正是民意调查所用的方法，有时选情预测的结果实在太接近以至于难分胜负。）而且决定样

本数量也比较困难。

下面就来介绍一个线性时间的算法以查找任意多个候选者中的众数。它比中数查找法更快、更简洁。类似于查找社会名流的算法（见 5.5 节），首先设法从众数的候选者中删除尽可能多的数，最终剩下一个数。找到最后的那个数基于如下观察事实，让我们能够缩小问题的规模：

若 $x_i \neq x_j$，且两个数都被删除，那么原来集合中的众数现在仍为新集合中的众数。

（注意，否命题不成立：1、2、5、5、3 中没有众数，但如果删除 1 和 2，5 就成了新的众数。）

因此，如果我们能找到两张不同的选票，将其删除，然后在剩余选票中查找众数，并检验其是否为原来集合中的众数。要是找不到两张不同的选票怎么办？如果逐个扫描选票发现它们均相同，那就只要盯住一个候选者，直到发现另一张不同的选票，从而利用上述观察事实。若剩余的所有选票均相同，则盯住一个候选者就可以。这是整个思想的本原，下面我们来看它的实现。

选票按出现的顺序依次扫描。我们利用两个变量 C（候选者）和 M（众数）。当考察 x_i 时，C 是 $x_1, x_2, \cdots, x_{i-1}$ 中惟一的候选者，M 等于 C 当前已出现的次数减去被删除的次数。也就是说，选票 $x_1, x_2, \cdots, x_{i-1}$ 被分成大小分别为 $2k$ 和 M 的两组，其中 $2k+M=i-1$。第一组有 k 对不同的选票（由观察事实可以删除），第二组包含已出现 M 次的 C。如果在 $x_1, x_2, \cdots, x_{i-1}$ 中存在众数，那么在不断删除选票后，它必定能胜出，并等于 C。（再次注意，这个否命题还是不成立：C 不是众数，也有可能在删除选票后胜出。）考察 x_i，将它同 C 比较，根据比较结果增加或减少其出现次数。我们还需要考虑候选者不存在的情况（例如，若 $x_2 \neq x_1$，x_3 就属于这种情况）。此时，M 等于 0，且 $C=x_i$，$M=1$。最后我们只剩一个候选者 C，统计其出现次数便能判定它是否为众数。算法由图 6.28 给出。

算法 *Majority* (X, n)

输入： X（有 n 个正数的数组）

输出： *Majority*（X 中的众数；若不存在，则输出−1）

```
begin
    C := X[1];
    M := 1;
    {首次扫描：删去除 C 以外的所有选票}
    for i := 2 to n do
        if M = 0 then
            C := X[i];
            M := 1
        else
            if C = X[i] then M := M + 1
            else M := M − 1;
    {第二次扫描：检验 C 是否为众数}
    if M = 0 then Majority := −1
    else
        Count := 0;
        for i := 1 to n do
            if X[i] = C then Count := Count + 1;
        if Count > n/2 then Majority := C
        else Majority := −1
end
```

图 6.28　算法 *Majority*

复杂度 找到候选者需要$(n-1)$次比较，最坏情况下判定该候选者是否为众数也需要$(n-1)$次比较。因此，总共需要$(2n-2)$次比较。可以进一步把比较次数减少到最优的$(3n/2+1)$次（见 Fischer and Salzberg [1982]）。无论怎样比较，由于每次比较都是常数级的，故总的运行时间是$O(n)$。

6.11 三个展现有趣证明方法的问题[①]

本节中，我们引入三个涉及到序列和多重集合的问题。每个问题的算法都说明了一种独特的证明方法。第一个算法利用加强归纳假设的原则，在求解过程中 4 次加强归纳假设，最终得到一个高效的算法。第二个算法比较明显，通过删除所有不必要的假设来改进“定理”。算法说明了这种方法并非总是那么直接。第三个算法说明如何巧妙选择归纳基础来改进算法。

6.11.1 最长递增序列

令 S 是不同整数 $x_1, x_2, \cdots, x_n$ 的序列。S 的递增序列（IS）是序列 $x_{i_1}, x_{i_2}, \cdots, x_{i_k}$，其中 $i_1 < i_2 < \cdots < i_k$，使得对任意 $1 \leqslant j < k$，有 $x_{i_j} < x_{i_{j+1}}$。S 的最长递增序列（LIS）是具有最大长度的递增序列。

> **问题** 给定一个由不同整数组成的序列，求它的最长递增序列。

这一节设计的算法是一个绝佳的加强归纳假设的例子。归纳假设被多次加强，每次都作为前一次尝试的结果。先考虑第一次直接归纳：

> **归纳假设（首次尝试）**：给定某个长度小于 m 的序列，知道如何求它的某个最长的递增序列。

归纳基础包含长度为 1 的序列，结论显然。给定长度为 m 的序列，我们先对前 $m-1$ 个元素求得其中一个 LIS，然后考察 x_m。如果 x_m 比 LIS 中最后一个归纳的元素大，则 x_m 就能加到 LIS 的末尾，形成的新 LIS 更长。解答完毕。如果 x_m 不大于 LIS 中最后一个归纳的元素，则问题无法继续解决。例如，可能有多个 LIS，而 x_m 也许只能使其中一个更长，而这个 LIS 未必就是由归纳法得到的那个。看来下一步必须得加强归纳假设，加强后的归纳假设如下：

> **归纳假设（第二次尝试）**：给定某个长度小于 m 的序列，知道如何求它的所有最长的递增序列。

归纳基础显而易见。我们用相同的方式进行归纳，只是对所有的 LIS 都考察 x_m，然后判定是否存在更长的 IS。这次尝试解决了刚才归纳假设中的问题，但它却带来了另一个问题，我们必须找到所有的 LIS。如果 x_m 不能使任何 LIS 变长，还有可能存在比最长的短 1 的 IS，且 x_m 可以使它变长，从而得到新的 LIS。这样一来，似乎陷入了僵局，因为现在必须求出所有最长的和第二长的 IS。而若想求出所有第二长的 IS，就必须求出所有第三长和第四长的 IS，如此等等。这充分说明了这个归纳假设过强了。

现在回过头来看一下这个过强的归纳假设。真的需要所有的 LIS 吗？我们只想知道 x_m 是否能让其中某个 LIS 变长，那么是否能找到其中最有可能变长的 LIS 呢？答案是肯定的。最有可能变长的 LIS 就是末尾的数最小的那个。（也许有多个 LIS 的末尾数都最小，它们都最有可能变长。为简单起见，我们用“最后可能的”来表示“某个最有可能的”。下面不妨用一个比先前弱一些的归纳假设。

---

① 首次阅读的读者不妨跳过本节。

归纳假设（第三次尝试）：给定某个长度小于 m 的序列，知道如何求它的某个最长的递增序列，使得其他最长递增序列的末尾的数都不比这个序列末尾的数小。

归纳基础显然成立。对给定的x_m，我们要判定它是否能通过归纳法加到 LIS 的末尾。假设 LIS 的长度为 s。若 x_m 能加到末尾，那么我们就得到一个比原先长的新的 LIS；由于它是惟一的，所以它一定就是“最可能的”。解答完毕。否则，我们仅知道不存在更长的递增序列，而这并不是最终答案。也许 x_m 不能加到最可能的 LIS 中（由于它小于那个 LIS 中的最后一个数），但它可以加到某个长度为 s–1 的 IS 中，使那个 LIS 的末尾的数最小。要证实这种可能性，我们需要求出长为 s–1 的最可能的 IS。同样，如果假设我们已经求出长为 s–1 的最可能的 IS，那么 x_m 也许能加长某个长为 s–2 的 IS 并使之成为长为 s–1 的最可能的 IS。我们必须确定是否存在这种可能性，从而不得不逐个求出长为 s–2、s–3，直到长为 1 的最可能的 IS，当然长为 1 的就是当前序列中的最小数。（即使不用归纳法，也可看出不能随意丢弃短的 IS——也许其中某个 IS 就是最终那个 LIS 的起始序列。）

下面继续考虑加强归纳假设。我们用 BIS (k)表示长度为 k 的最可能的递增序列，即以最小数结尾的序列（如果不止一个，则任取其中某个）。把 BIS (k)中末尾的数记为 BIS (k).*last*。

归纳假设（第四次尝试）：给定某个长度小于 m 的序列，知道如何对任意 $k < m$–1 求出 BIS (k)，如果存在的话。

归纳基础显然成立。给定 x_m，要求出它可以改变的那些 BIS。x_m 能加长某个 BIS(k)当且仅当下列两个条件成立：（1）$x_m >$ BIS (k)，于是 x_m 可加入 BIS (k)；（2）$x_m <$ BIS (k+1).*last*，于是 BIS(k)末尾加上 x_m 后比 BIS(k+1)的可能性更大。我们认为，BIS(1).*last* < BIS(2).*last*< $\cdots$ < BIS (s).*last*，其中 s 是 LIS 的长度。这个命题为真，因为对某个 j，BIS(j).*last* $\leqslant$ BIS(j–1).*last*，则 BIS(j)的前 j–1 个数比 BIS(j–1)更可能。算法如下：给定某个 x_m，依次察看 BIS(i).*last* 的值，其中 $i = s, s$–1, s–2 等，知道找到某个 BIS(j).*last* 小于 x_m。如果这样的 j 不存在，那么 x_m 就是当前序列中最小的数，并将其作为 BIS(1)。若 $j = s$，则把 x_m 加在 BIS(s)后，构造出一个新的 BIS(s+1)。（先前的 BIS(s)不变。）否则，有 BIS(j).*last* < x_m < BIS(j+1).*last*。于是用 BIS(j+1)替换 BIS(j) x_m。

这基本上就是整个算法，而且一旦证明归纳正确，所有的步骤并不复杂。值得注意的是，由于集合是有序的，所以可以用二叉搜索进行查找。这样，每个 x_m 最多增加 O (log m)次比较，整个运行时间为 O (n log n)。算法的细节还要进一步整理，留给读者完成。

6.11.2　查找集合中两个最大的元素

对非实质性的假设或步骤进行详尽的探究是定理证明常见的重要方法之一。而消除非实质性假设有时能够得到更好的算法。不必要的假设有时意味着证明有可能存在错误。以下选自 Polya and Szego[1927]：“我们应该仔细检查证明过程中是否确实用到了所有的假设；应该设法用更少的假设完成同样的证明……另外，除非有反例说明已经到达所有可能性的证明的边界，否则我们应该永不满足。”算法亦是如此。这听起来并不复杂，而在下例中，我们将会看到事实并非如此。

> **问题**　已知集合 S 有 n 个元素 $x_1, x_2, \cdots, x_n$，求其中最大的和第二大的元素。

要寻求的是比较次数最少的算法，其他操作忽略不计，为简单起见，设 n 是 2 的幂。

先试着用分治法将大小为 n 的集合 S 分成大小为 n/2 的两个子集 P 和 Q。如果现在直接进行归纳，则假设已知 P 和 Q 中最大的和第二大的元素，分别记为 p_1, p_2 和 q_1, q_2，然后查找 S 中最大

的和第二大的元素。很明显，两次比较足以找到 S 的这两个元素。第一次比较最大数 p_1 和 q_1，第二次比较在第一次比较的“败”者和“胜”者所在集合的第二大元素间进行（见图 6.29）。用这种方法得出递归关系式 $T(2n)=2T(n)+2$ 及 $T(2)=1$，解得 $T(n)=3n/2-2$。尽管这要优于直接进行 $2n-3$ 次比较，但类似于 6.5.1 节中查找最大数和最小数的问题，算法还可能进一步优化。

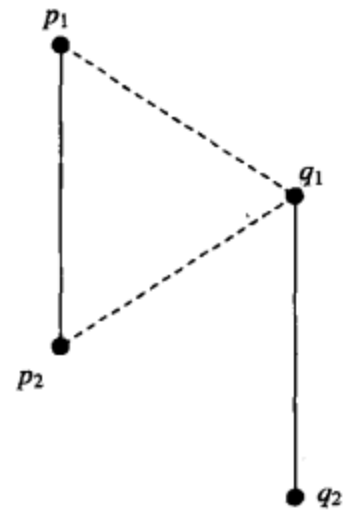


图 6.29 查找最大的和第二大的元素（一条虚线对应一次比较）

若每一步归纳中的两次比较不能减少，那么如何减少总的比较次数呢？仔细观察图 6.29 中的比较，可以看出算法不再使用 q_2，故 q_2 的计算是无用的。如果能精简这样的计算，就可进一步减少比较次数。然而，在 p_1 和 q_1 比较之前，我们并不清楚 p_2 和 q_2 中的哪一个可以不必考虑。如果知道哪个子集会“失败”，那么就可以在这个子集上用常规的查找最大数的算法了，这还能省去不少比较。看来，我们已经意识到有些比较可以避免，只是不清楚究竟哪些可以忽略。

这里的技巧是把第二大元素的计算推迟到算法的最后，只保留第二大元素的候选者列表，假定在下面的归纳假设中第二大元素未知：

归纳假设：给定一个小于 n 的集合，知道如何找到最大的元素以及第二大元素的一个“短短的”候选者列表。

“短短的”到底有多长尚未定义，但在算法设计过程中，我们会找到一个合适的值。

算法执行过程如下：给定大小为 n 的集合 S，把它分为大小为 $n/2$ 的子集 P 和 Q。由归纳假设可知两个集合最大的几个元素是 p_1 和 q_1，再加上第二大元素的候选者集合 C_P 和 C_Q。首先比较 p_1 和 q_1，取其中大者，例如 p_1，作为 S 的最大数。然后舍去 C_Q，因为 C_Q 中的所有元素都小于 q_1，而 q_1 最多是第二大元素，接着再把 q_1 加入 C_P。最终我们得到了一个最大的元素和一个候选者集合，从中可直接选出第二大的元素。查找最大数所需的比较次数满足递归关系式 $T(n)=2T(n/2)+1$ 及 $T(2)=1$，解得 $T(n)=n-1$。很容易看到，扩大一倍集合的大小时，我们能在候选者集合中再加入一个元素，所以 $\mathrm{lb}n$ 足以满足候选者集合大小的需要。于是查找第二大元素需要 $\mathrm{lb}n-1$ 次额外比较，因而总比较次数是 $n-1+\mathrm{lb}n-1$，恰为最可能的次数（见[Knuth 1973b]）。n 为 2 的幂时的归纳假设如下：

归纳假设：给定一个小于 n 的集合，知道如何求出最大的元素和第二大元素的候选者集合，这个集合最多不超过 $\mathrm{lb}n$ 个元素。

评述 一旦算法构造完毕，将有必要检查是否存在对最终结果不起作用的部分。这些部分往往可被删除。即使某些运算不能被删除，也可以用更简单、效率更高的运算代替。

6.11.3　计算多重集合的模

令 $S=(x_1, x_2, \cdots, x_n)$是一个多重集合（元素不必互异），其元素取自一全序集合。多重集合的模（mode）定义为集合中出现最频繁的元素（可能有多个模）。一个元素的出现次数被称为它的重数，所以模是重数最大的元素。

> **问题**　求某个给定的多重集合 S 的模。

希望算法的比较次数最少。求模的方法之一是排序。一旦所有元素排序完毕，就可以对序列进行扫描，统计元素的重数（相同的元素在有序序列中是连续的）。实际上排序并非必不可少，之所以这么说是因为查找中数的算法（见 6.10 节）是线性时间的，而排序算法则是 $O(n \log n)$的。这就有理由猜测，如果模的重数很高，那么求模或许有更快的方法。

先试着直接归纳。假定已知 n–1 个元素的多重集合的模，求 n 个元素的多重集合的模。这有些困难，因为最高重数的元素可能有多个；第 n 个元素会打破已有的局面。再假定归纳假设已知所有最高重数的元素，此时可以判定第 n 个元素是否会打破已有的局面，但是它有可能提高其他元素的重数，使其他元素加入列表中。前面（见 6.11.1 节）已经说明可以同时关注所有不同的“最可能的”的解，不过代价未免太高了。另一方面，第 n 个元素不必任意选择，我们可以指定某个特殊的元素。假设第 n 个元素是最大元素，则问题不变，解答变易。通过删除最大元素的每次出现以缩小问题的规模，然后对缩减后的问题求解，再比较缩小后的多重集合的模和最大元素的重数。

现在得到了一个性能不太好的算法。查找 n 个元素的多重集合的最大元素需要 n–1 次比较。若多重集合有许多不同的元素，那比较次数就太多了。特别是，当多重集合是一个集合时（即所有元素两两互异），这个算法从本质上来说就成了 $O(n^2)$的选择排序。

现在用分治法改善它的性能。这次不是对一个元素或一小部分元素的集合归纳，而是把这个多重集合分成大小相同、元素互异的两部分，使得每部分都形成一个独立的子问题。那么如何把一个多重集合分割成大小基本相同、元素互异的两部分呢？首先，求得这个多重集合的中数，根据元素小于、等于或大于中数的关系把该多重集合分成三部分。前面曾经介绍过在 $O(n)$的期望比较次数下求得中数（见 6.5 节），同样，最坏情况下也可以在 $O(n)$时间内求得，这一点尚未证明。中数查找算法将作为我们这个算法中的一步。给定一个大小为 n 的多重集合，首先求它的中数，然后分割集合，接着分别对两个每个规模不超过 $n/2$ 的子问题求解。由于两个子问题集合是互异的，原问题的模可以很容易从这两个小一些的子问题的模中导出。因中数查找和问题分割均能在线性时间内完成，故得到如下熟悉的递归关系式：

$$T(n) \leqslant 2T(n/2)+O(n),\quad T(2)=1,$$

解得 $T(n)=O(n \log n)$。但是这并不比排序算法更优。事实上，如果不用中数而是任取某个元素来分割问题，则算法实质上等同于快速排序。

下面来介绍算法的核心部分并通过归纳基础来提高算法的性能。假设模的重数是 M，我们断言可以从大小为 M 的多重集合开始归纳。换句话说，不必把多重集合分成小于 M 的部分。由于所有的部分均互异，则只有一个大小为 M 的部分包含着模。这样，其他元素的重数不会超过 M，于是求得模。所以没有必要继续分割多重子集。

直接实现算法比较困难。由于事先不清楚递归应该执行多久，因此不能用递归式。当多重集合的大小最多达到 M 时，递归必须终止，而 M 的值是通过检验所有比当前多重集合更小的多重集合来确定的。每一步检查所有的多重子集，如果没有一个多重子集只含有一个与众不同的元素，

那么它们都应该被继续分割。而如果其中某个多重子集只含有一个独一无二的元素，那么算法可以被终止。算法的实现细节留给读者完成。

复杂度 只须修改递归关系式的归纳基础：

$$T(n)\leqslant 2T(n/2)+O(n),\quad T(M)=O(M),$$

可得次数是 $O(n\log(n/M))$。凭直觉，递归一直执行直至遇到大小为 M 的多重集合才终止，共需 $\log(n/M)$次。每次分割和检查所有子问题都是线性时间的。特别是，若 $M=cn$，其中 c 是某个常数，则算法是线性的。若 M 是常数，则算法是 $O(n\log n)$的。仅当 M 相当大且比较的代价也相当大（子问题的存储开销颇高）时，这个算法才优于排序算法。

6.12 小结

本章中涉及多个主题，包括查找问题、排序问题、顺序统计、数据压缩、字符串处理、概率算法以及其他一些问题。每个主题只介绍了其中的一、两个基本问题。实际应用中的问题不像本章中的那样简洁明了，因此应该设法从一个具体问题中抽象出其中的主要部分。本章所涉及的方法与第 5 章中介绍的十分相似，归纳法仍然起了相当重要的作用。

本章所讨论的问题都有直接解法，如线性查找和选择查找，几乎不用花很大工夫就能解决。若输入很小，这些解法不但有效，而且优于其他更复杂的解法。但是只要输入不是太小（比如超过 100），就有必要尝试寻求更好的解答。例如我们常用线性查找、平方级排序等算法，遗憾的是，当输入很大时，我们往往还使用包括它们在内的一些低效算法[①]。

参考文献和补充读物

排序问题、查找问题及其历史的详细资料可参见 Knuth [1973b]。其他有关序列、集合以及组合数学的主题可查阅 Stanton and White [1986]。二叉搜索模型的正规推导参见 Manna and Waldinger [1987]。6.2 节中的重叠子序列问题摘自 Mirzaian [1987]，它提供了一种线性时间的解法。内插搜索的平均性能由 Perl, Itai, and Avni [1978]研究，van der Nat [1979]给出了平均性能的一些经验性的结果。

归并排序可能由 von Neumann 于 1945 年首创，它是首批被实现的可存储程序之一。归并排序的原地置换算法由 Kronrod [1969]首次提出；亦可参见 Huang and Langston [1988]、Dvorak and Durian [1988]。快速排序由 Hoare[1962]提出。有关快速提序的详细研究参见 Sedgewick[1978]。堆排序首先由 Williams [1964]提出。多伦多大学的计算机图形学小组[1981]制作了一部上佳的电影，其中包括了 9 种主要排序方法的精彩动画演示。尽管排序问题已被广泛研究了许多年，仍有相当数量的问题有待解决。对 n 个数排序所需的确切比较次数仍无定论。习题 6.30 提出的问题由 Ford and Johnson [1959]给出，在相当长的一段时间内，它被认为是所需比较次数最少的算法，不过 Manacher [1979]证明了它并非是最优的。另一个应用广泛的排序算法是由 Shell [1959]发明的希尔排序（Shellsort）。它很简单，实现起来也不难，但是人们对它的复杂度还不了解。希尔排序近期的研究成果和观察结果可参见 Incerpi and Sedgewick [1987]。决策树已被成功用于证明几个基本问题的下界。Moret [1982]给出了一份关于决策树应用的调查。

① 一些事例恰巧揭示了这个问题。1988 年 11 月 2 日，一种计算机（蠕虫）病毒袭击了全美超过 6000 台计算机，明显减慢了其运行速度。其中部分原因归咎于所用的搜索算法都是线性查找（见 Spafford [1988]）。

概率选择算法的分析由 Floyd and Rivest [1975]给出。顺序统计的线性时间确定性算法由 Blum, Blum, Floyd, Pratt, Rivest, and Tarjan [1972]首先提出。可是由于常数因子很大，实际运行时间为 $\mathcal{O}(n)$ 。Schönhage, Paterson, and Pippenger [1976]提出了不超过 $3n$ 次比较的中数查找算法。最著名的查找中数算法（所需比较次数）的下界是 $2n$（Bent and John [1985]），这篇论文包含了顺序统计问题的一般结果；该下界的一般表达式更复杂。

数据压缩技术极其重要，人们对它进行了广泛的研究。6.6 节的算法归功于 Huffman [1952]（亦可参见 Knuth [1973a]）。只扫描一遍的霍夫曼算法的几种变形由 Knuth [1985]和 Vitter [1985]给出。另一个效率高、应用广的算法由 Ziv and Lempel [1978]给出。更多关于数据压缩的常见算法可参见 Lynch [1985]。

6.7 节介绍的序列匹配算法归功于 Knuth, Morris, and Pratt[1977]和 Boyer and Moore[1977]。Galil [1979]对最坏情况下的 Boyer-Moore 算法的运行时间做了改进。更多关于 Boyer-Moore 算法复杂度的介绍可参见 Guibas amd Odlyzko [1980]和 Schaback [1988]。各种字符串匹配的比较经验可查阅 Smit [1982]。Karp and Rabin [1987]提出了一个字符串匹配的概率算法。这个算法借助指纹识别思想，用短字符串表示长字符串，使字符串的匹配效率得到提高。不仅如此，它还适用于两位模式的情况。字符串匹配问题还可以进一步扩展到比字符串更复杂的模式的比较。例如，像十分管用的“百搭”（wild cards）那样，我们也许要查找所有 $B*C$ 模式的字符串，其中 B 和 C 是给定的，*代表任意字符串。更一般的问题是查找任意字符串的正规集合。更多这一类问题可参见 Aho and Corasick [1975]。查找预处理固定文件中的字符串也是一个重要的问题。后缀树（Weiner [1973]、McCreight [1976]）和后缀数组（Manber and Myers [1990]）保证了这类问题的快速查找。

序列比较及其应用在 Sankoff and Kruskal [1983]一书中有所涉及。各类字符串的问题可在 Apostolico and Galil [1985]一书中找到。6.8 节中的算法来自 Wagner and Fischer [1974]。改进这个算法有多种途径，包括存储空间的节省（见 Hirschberg [1975]）、字符表很大（见 Hunt and Szymanski [1977]）或序列很相似（Ukkonen [1985]和 Myers [1986]）时运行时间的缩短。相关结果的调查报告可查阅 Hirschberg [1983]。

在序列前半段查找元素的算法由Yao [1977]提出。Knuth [1981]详细介绍了随机数的生成。6.9.2 节给出的概率着色算法基于一个存在性概率证明，这个证明可参见 Bollobás [1986]。6.9.3 节中引入的把概率算法转换成确定性算法的方法归功于 Raghavan [1986]。K.Pruhs 指出，可以用这种方法解决 6.9.2 节的着色问题。任意大小子集的有效着色问题是 NP 完全的（见 Lovász [1973]）。Erdös and Spencer [1974]提供了不少用概率方法证明组合数学性质的例子。

Misra and Gries [1982]等研究了众数问题。Fischer and Salzberg[1982]利用一个比 6.10 节中更复杂的数据结构，证明了在最坏情况下的比较次数（而不是其他步骤的次数）能降到 $(3n/2 + 1)$ 这个最优下界。

Gries [1981]精彩地描述了最长递增序列的一种解法（本书主要借鉴了这种解法。）Erdös and Szekeres [1935]巧妙利用了抽屉原理，证明了长度为 n^2+1 的不同元素组成的序列一定有一个长度为 $n+1$ 的递增或递减的子序列。Lewis Carroll（参见 Knuth [1973b]）在安排网球巡回赛赛程一文中首次说明了查找最大的和第二大的元素的问题。另一个求模的算法由 Dobkin and Munro [1980]给出（亦可参见 Gonnet [1984]）。

Aho, Hopcroft, and Ullman [1974]讨论了习题 6.27 的解答。习题 6.34 摘自 Karp, Saks, and Wigderson [1986]。Rodeh [1982]给出了习题 6.39 的一种解答。Choueka, Fraenkel, Klein, and Perl [1985]讨论了习题 6.42 的主题。Ryser [1957]提出了可实现序列（realizable sequences）（见习题 6.64）的思想。

操练型习题

6.1 为下面这个著名的游戏设计一种好的策略。游戏的一方先想一个 1 至 n 之间的数。另一方通过不断问“这个数小于（大于）x 吗？”的问题来确定这个数。目标是尽可能地问更少的问题。（假定没有人说谎。）

6.2 假设习题 6.1 中数的取值范围没有限制，可取任意正整数。为这个猜数游戏设计一种策略。

6.3 假定你正在使用一个处理大量文本的程序，如文字处理程序。程序输入为一表示为字符串的文本，然后产生一些输出。有时程序会陷入某个自身无法恢复的错误，而且它甚至无法说明产生了什么错误，哪里发生了错误。换句话说，程序的执行停止在某处，并输出“Error”。假设程序发生的是局部错误，即程序由于某个未知原因无法处理文本中的某个字符串。这个错误与出错字符串的上下文无关。设计一个策略来定位错误发生的原始位置。

6.4 构造一个例子，使得用内插搜索在大小为 n 的表中查找时进行$\Omega(n)$次比较。

6.5 试编写一个完整的实现基数调换排序法的程序。程序的输入是 n 个整数，每个整数有 k 位，每一位的取值范围是从 1 至 m。假设可用存储空间是 $O(m)$。首先把程序写成递归过程，并确定递归过程所需的额外存储空间。然后设计一个非递归程序，尽量减少额外存储空间的使用。

6.6 编写程序，分别完整实现插入排序（包括线性查找和二分查找）和选择排序。

6.7 统计图 6.8（归并排序）和图 6.11（快速排序）中对输入排序所进行比较的次数。再统计对相同输入，插入排序和选择排序所进行比较的次数。

6.8 用循环不变量证明算法 *Mergesort*（见图 6.7）中的一个 if 语句是不必要的。也就是证明，如果去掉这条 if 语句，从“*if Left* ≠ *Right*”这条语句开始执行程序，程序运行结果不变。

6.9 比较归并排序和第5章中水平线问题的解答。试用形式化语言描述它们之间的相似性。是否有可能用一个类似“黑盒”的解答解决另一个问题呢？

6.10 为算法 *Partition*（见图 6.9）的主循环寻找一个合适的循环不变量，并证明算法的正确性。

6.11 构造一个例子，使得当序列的第一个、最后一个和正中间的元素分别作为支点时快速排序进行$\Omega(n^2)$次比较。

6.12 排序算法的输入有时基本上是有序的，即乱序排列的元素很少。描述一下 6.4 节中介绍的各种算法在输入基本有序的情况下是如何执行的？你会选择哪种算法呢？（这是鼓励读者设计自己的算法。）

6.13 构造一个类似图 6.15 中的表，用来自上而下建堆。

6.14 设计一个查找集合中最大和最小元素的分治算法。该算法最多做 $3n/2$ 次比较（这里 $n = 2^k$）。你能指出该算法的比较次数少于直接算法 $2n-30$ 次比较的原因吗？

6.15 为本问题中涉及的字符集合构造霍夫曼树，必须包括所有的字符。用霍夫曼树与用固定长度编码相比，在存储空间上能节省多少位？（译者注：为方便读者统计原文中的字符，保留原文如下：Build the Huffman tree for the set of characters in this question. Include all characters. How many bits are saved in the storage of this question using Huffman trees versus a storage based on a fixed-length encoding?）

6.16 构造字符串 *aabbaababababbaabbaabb* 的 *next* 表（见 6.7 节）。

6.17 用图 6.27 中的算法 *Minimun_Edit_Distance* 比较序列 *aabccbbaabca* 和 *baacbabaccaba*，并构造 C 和 M 两个矩阵。

6.18 为算法 *Majority*（见图 6.28）的第一个循环寻找一个合适的循环不变量，并证明第一阶段算法的正确性。

创造型习题

除非特别说明，以下习题中的序列和集合的大小都是 n，其中元素都是实数。如果算法的运行时间是 $O(n)$，则称之为线性时间的。这里凡是运行时间指的都是最坏情况。

6.19 已知一整数数组 $A[1..n]$，使得对任意的 i，$1 \leqslant i < n$，有 $|A[i]-A[i+1]| \leqslant 1$。令 $A[1]=x$，$A[n]=y$，其中 $x<y$。设计一个高效算法查找 j，使得对任意给定的 z，$x \leqslant z \leqslant y$，有 $A[j]=z$。你设计的算法最多需要跟 z 比较多少次？

6.20 用决策树证明，你为习题 6.19 设计的算法在最坏情况下是最优的（否则请改进算法直到你能证明它是最优的）。

6.21 算法的输入是 n 个实数组成的集合。试设计一个 $O(n)$时间的算法找到某个不在集合中的数。并证明：算法所需步骤的下界是$\Omega(n)$。

6.22 算法的输入是 n 个实数组成的集合以及某个实数 x。

a. 设计一个算法，判定 S 中是否存在两个元素，其和恰为 x。该算法应是 $O(n \log n)$的。

b. 假设集合 S 是有序集合。设计一个能在 $O(n)$时间内解决上述问题的算法。

6.23 给定两个集合 S_1 和 S_2，以及一个实数 x。试判定是否存在 S_1 中的某个元素和 S_2 中的某个元素，它们的和恰为 x。该算法应是 $O(n \log n)$的，其中 n 是两个集合元素个数之和。

6.24 设计一个能够判定两个集合是否互异的算法，并用两个集合的大小 m 和 n 来表示该算法的复杂度。确认你所设计的算法考虑到了 m 比 n 小得多的情况。

6.25 设计一个计算两个大小为$O(n)$的集合的并集的算法。输入集合以数组表示。输出应该是个两个集合并集运算后所得各不相同元素的集合。输出数组的元素不能出现大于一次。最坏情况下该算法应是 $O(n \log n)$的。

6.26 算法的输入是实数序列 $x_1, x_2, \cdots, x_n$，其中 n 是偶数。设计一个算法，能以下列方式把输入序列分割成 $n/2$ 个数对。计算每个数对的和，把这 $n/2$ 个数对记为 $s_1, s_2, \cdots, s_{n/2}$，算法的目标是使最大的和（maximum sum）最小。

*6.27 修改字典序排序算法使之能处理变长字符串。换句话说，所有数都是 k 位的假设不存在了，它们有长有短。当然，可以加上“空”（0）位使它们的长度相同。试找到一个不补空位，且对所有位之和是线性时间的算法。

6.28 算法的输入是以任意顺序排列的整数序列 $x_1, x_2, \cdots, x_n$ 和 1 至 n 这 n 个整数的序列 $a_1, a_2, \cdots, a_n$（可以把 $a_1, a_2, \cdots, a_n$ 看做是 $1, 2, \cdots, n$ 的一个排列）。两个序列均以表示。设计一个$O(n \log n)$的算法，根据第二个序列的顺序重新排列第一个序列。也就是说，对任意的 i，x_i 的位置是 a_i。例如，$x=17, 5, 1, 9$，$a=3, 2, 4, 1$，则输出应是 $x=9, 5, 17, 1$。设计的算法应是原地置换排序算法，故不能使用额外的数组。

6.29 算法的输入是 d 个有序序列，这些序列共有 n 个元素。设计一个 $O(n \log d)$的算法把这 d 个序列合并为一个有序序列。

6.30 下面是一个排序算法不完整的大致描述，它被称为 Ford and Johnson 排序法。

1. 任意组成 $n/2$ 个不同的数对；
2. 比较每个数对中的数；
3. 对前 $n/2$ 个较大的数递归排序；
4. 以某种顺序将剩余 $n/2$ 个数插入已排序的前 $n/2$ 个较大数的序列。

该算法在保证以“好的”顺序完成第 4 步的前提下，比任何其他算法所用的比较次数都少。

考虑 n=5、6 和 8 的情况。为第 4 步寻找一个“好的”顺序，使最终得到的算法在 n 取上述值时，（从比较次数的角度来说）是最优的（事实上在 $n<12$ 的情况下均如此）。

6.31 算法的输入是 n 个有不少重复的数的序列，序列中的不同整数有 $O(\log n)$个。

a. 设计一个排序算法，在最坏情况下最多用 $O(n \log\log n)$次比较将上述序列排序。

b. 为什么这次不能满足$\Omega(n \log n)$的下界呢？

6.32 证明：有 n 个节点的平衡二叉树的所有节点的高度之和不超过 n–1。（有 n 个节点的平衡二叉树用大小为 n 的数组的隐式表示。）试构造一棵高度之和恰为 n–1 的平衡二叉数。

6.33 堆中所有节点的高度（见 6.4.5 节）还可通过节点高度在数组中的对应位置来计算。（大小为 n 的）数组代表的堆中所有节点的高度不超过$\lceil \mathbf{lb}(n-i+1) \rceil$。试用这种方法求高度之和。

6.34 算法的输入是表示为数组的、大小为 n 的堆（最大的元素在堆顶）和一个实数 x。设计一个判定堆中第 k 大的数是否小于或等于 x 的算法。你所设计的算法在最坏情况下的运行时间应是 $O(k)$，且与堆的大小无关。可用空间是 $O(k)$。（注意，不必查找到第 k 个数，只需要判定它与 x 的大小关系。）

6.35 以下是加权选择问题。算法的输入是 n 个不同的整数组成的序列 $x_1, x_2, \cdots, x_n$，其中每个 x_i 都有一个与之相关的正权值 $w(x_i)$。令 W 是所有权重之和。加权选择问题就是要找到某个 x_j，对给定的某个值 X，其中 $0<X\leqslant W$，满足

$$\sum_{x_i > x_j} w(x_i) < X$$

和

$$w(x_j) + \sum_{x_i > x_j} w(x_i) \geqslant X。$$

试设计一个高效的算法解决加权选择问题。（注意，如果所有的权值都是 1，则问题演变为一般的选择问题。）

6.36 假设 A 是通过一系列比较从 n 个元素的集合中找到第 k 大元素的算法。证明：A 收集了足够的信息来判定哪些元素大于第 k 大元素，哪些元素小于第 k 大元素。（换句话说，你可以在不做更多比较的条件下，从第 k 大元素处分割集合。）

6.37 考察查找第 k 大元素的那个问题，这次我们关心的是减小空间。假设每个元素占一个内存单元。算法的输入是一个元素序列，每次只输入一个元素，并将它存入固定单元 C。即在第 i 步把 x_i 放入 C（且覆盖 C 中前一次放入的元素）。在两次输入元素之间可以进行任意计算（当然包括把 C 中元素存入某个临时单元的操作）。算法要求尽可能减少额外所需的单元数。试求出查找第 k 大元素所需单元数的上界和下界。

6.38 类似于习题 6.37，本题将查找第 k 小元素。不过这次不仅要求运行时间最短，还要尽量节省空间（尽管不一定是最小空间）。同样，算法的输入是元素序列 $x_1, x_2, \cdots, x_n$，每次只输入一个元素。设计一个预期时间为 $O(n)$的，仅用 $O(k)$个内存单元查找到第 k 小元素的算法。k 的值是预知的（这样就可以事先分配好足够的内存），但 n 的值在最后一个元素输入之前是未知的。

6.39 设 A 和 B 是两个各有 n 个元素的集合，A、B 分别存储在计算机 P 和 Q 中。P 和 Q 可以通过发送消息进行交互，还可以进行任意一种本机的运算。设计一个查找 A 和 B 并集中第 n 小元素（即中数）的算法。为简单起见，不妨假定所有元素互异。算法的目标是发送的消息数最少。其中每个消息包含一个元素，也可以看做整数。最坏情况下发送的消息数是多少？

6.40 已知整数集合 $S=\{x_1, x_2, \cdots, x_n\}$，求一非空集合 $R \subseteq S$，使得 $\sum_{x_i \in R} O \text{ (modulo } n)$。

6.41 用信息论中边界的思想证明，在序列 $x_1, x_2, \cdots, x_n$ 中查找是否存在某个 i，使得 $x_i = i$ 的算法下界是 $\Omega(\log n)$ 次比较。（这个问题曾在 6.2 节和 6.4.6 节中讨论过。）

6.42 假设你打算用霍夫曼编码，但你所用的程序语言不允许直接对比特位进行操作，只能把位串当做字节（或者由具体机器决定的长为 k 的位段）序列。每个字节（位段）对应一个整数，于是位串的编码就形成了一个整数序列（每个整数均小于 2^8）。设计一个转换整数序列的方法，使得可以用霍夫曼树解码相应的位串。通过构造 $k \times 2^k$ 的表，其中 k 是位段的大小（位段是字节时等于 8），来转换这个序列。这张表由霍夫曼树决定（树是已知的）。你只能用整数乘法、加法和减法运算，不能用位运算。但可以借助这张表来访问序列中某个整数 i 的任意一位。现在若假设表的大小是 2×2^k，请再一次求解这个问题。

6.43 假设某个文本已用霍夫曼编码。已知该霍夫曼树和文本中每个字符的字频。现在假定文本经过了小规模的改动，改动后其中（原已存在的）某个字符 X 的字频增加了 1。你打算重新调整霍夫曼树，使之在文本改动后仍保持最优。一位朋友对调整霍夫曼树的算法提出了如下建议：

首先，他指出霍夫曼树有一个重要的性质，节点的字频往树根方向是非递减的。（换句话说，低字频的节点在树上的位置不能高于高字频的节点。）某个内部节点 v 的字频被定义为 v 的所有后继外部节点的字频之和。他建议，通过检查 X 的上层节点来确认 X 在字频增加后是否仍然满足上述性质。如果上层没有节点的字频小于 X，则 X 不动。否则用上层某个更小字频的字符跟 X 交换位置。这个算法有时也能奏效，但总的来说它是错的。解释一下算法为什么不正确，以及如何改正？你不仅应该说明算法漏了什么，并且重要地要讨论一下算法总的来说错在哪里？也就是说，要么举出反例来说明算法不能构造出最优的霍夫曼树，要么先假设算法正确，然后推出矛盾（或者高度可疑的结论）。仅仅指出算法在某些情况下出错是不够的，也许那些情况可以被忽略。你必须说明算法是完全不正确的。

6.44 算法的输入是两个字符串 $A = a_1a_2\cdots a_n$，$B = b_1b_2\cdots b_n$。设计一个 $O(n)$时间的算法来判定 B 是否为 A 的循环移位后生成的。换句话说，算法应该判定是否存在某个下标 k，$1 \leqslant k \leqslant n$，使得对任意 i，$1 \leqslant i \leqslant n$，有 $a_i = b_{(k+i) \bmod n}$。

6.45 KMP 串匹配算法对二进制位串可以做进一步改进如下：构造 *next* 表时，除了考虑目前已经考察过的字串后缀，还可以考虑不匹配的字符。也就是查找与 B 的某个前缀匹配的 $B(i-1)\overline{b_i}$。$\overline{b_i}$ 是 b_i 的补。这样的话，A 的每个元素都恰与 B 中的某个元素比较一次。

a. 给出修改后的 *next* 表的精确定义，为图 6.21 中的那个例子重新计算新的 *next* 表值。

b. 修改串匹配算法使之更好地利用这一变化。

6.46 **一个在线串匹配算法**：假设每次相对缓慢地输入一个字符（如用键盘输入），输入文本事先给定。我们打算尽可能多地匹配已经输入的字符，而不必等待整个字串全部被输入。换句话说，当输入第 k 个字符时，我们想立刻定位到与前 k–1 个字符匹配的首个位置。修改 KMP 算法以达到这一目的。

6.47 修改 KMP 算法，查找能与 A 的某个子串匹配的 B 的最长前缀。换句话说，你可以不必在 A 中与 B 的所有字符匹配，而只须找到最长的匹配子串（不过必须以 b_1 开头。）

6.48 令 T 和 P 是两个字符串 $t_1, t_2, \cdots, t_n$ 和 $p_1, p_2, \cdots, p_k$，其中 $k \leqslant n$。设计一个 $O(n)$的算法判定 P

是否为 T 的字串。（若存在下标序列 $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$，使得对任意的 j，$1 \leqslant j \leqslant k$，有 $t_{i_j} = p_j$，则 P 是 T 的字串。）

6.49 为习题 6.48 设计一个算法，要求如果 T 中有许多子序列等于 P，则算法要找出序列下标之和最大的那个。也就是要找到下标序列 $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$，使得对任意的 j，$1 \leqslant j \leqslant k$，有 $t_{i_j} = p_j$，且 $\sum_{j=1}^{k} i_j$ 取最大值。

6.50 考虑习题 6.48。假设为 T 的第 i 个字符赋上 $c(i)$的值，求使赋值之和最大的匹配的子序列。也就是找到下标序列 $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$，使得对任意的 j，$1 \leqslant j \leqslant k$，有 $t_{i_j} = p_j$，且 $\sum_{j=1}^{k} c(i_j)$ 取最大值。

6.51 序列 T 和 P 的最长共同子序列（LCS）是某个序列 L，其中 L 是 T 和 P 的共同子序列中最长的那个。序列 T 和 P 的最短共同子序列（SCS）是某个序列 L，其中 L 是 T 和 P 的共同子序列中最短的那个。

a. 设计一个高效的算法求两个已知序列的 LCS 和 SCS。

b. 令 $d(T, P)$是 T 和 P 之间的最小修改距离，不允许进行字符替换（换句话说，只能插入或删除字符）。证明 $d(T, P) = |SCS(T, P)| - |LCS(T, P)|$，其中$|SCS(T, P)|$和$|LCS(T, P)|$是 T 和 P 的最小的 SCS（或 LCS）的长度。

6.52 把 6.8 节中的问题 *Minimal-Edit-Distance* 推广到不计序列首尾插入字符的情况。换句话说，如果 B 在 A 中匹配，那么我们就不必统计扩展 B 所需的插入次数，只要计算能与 A 的子序列匹配的 B 的修改距离。（注意，如果不计在 B 的开始位置插入字符的代价，那就必须计算在 A 的末尾插入字符的次数；反之亦然。）

6.53 序列比较问题可以推广到三个（或多个）序列的情况：允许在每一步从任意序列中插入、删除或替换字符。如果所有序列中的相应字符都相同，那么这步的开销是 0，否则就是 1（即使有两个序列已能匹配，且只需要进行一次插入或删除操作）。例如，假定有三个序列 *aabb*、*bbb* 和 *cbb*。不妨在 *bbb* 和 *cbb* 前插入 *a*（开销是 1），用一个 *a* 在 *bbb* 和 *cbb* 中分别替换一个 *b* 和一个 *c*，剩余序列可以匹配；总的开销是 2。设计一个 $O(n^3)$的算法求三个已知序列的最小修改距离。

6.54 令 $A = a_1 a_2 \cdots a_n$，$B = b_1 b_2 \cdots b_m$是两个字符串。记 $A[i]$为字符串 $a_i a_{i+1} \cdots a_n$（顾名思义，A 的第 i 个后缀）。令 d_i是 B 和 $A[i]$的最小修改距离。设计一个 $O(n^2)$的算法（在所有的 i 中，$1 \leqslant i \leqslant n$）查找 d_i 的最小值。

6.55 算法的输入是数列 $x_1, x_2, \cdots, x_n$。证明，选出的数能位于数列的前半段（大于或等于中数）的确定性算法一定至少需要进行$\lfloor \frac{1}{2} n \rfloor$次比较。

6.56 求 6.9.2 节中概率着色算法所需步数的期望值，用 k 和 r 表示。

6.57 对任意 $k \leqslant n$，假设你有一个能生成 1 至 k 之间随机数的函数。设计一个能生成 n 个数随机排列的算法。每个可能的排列出现的概率应相同。

6.58 民意测验应用了概率算法。假设有两个候选者和 n 个选民。常用的算法是随机征求 k 个选民的意见，取其平均。假定两个候选者各得到恰好一半选民的支持，那么（征求 k 个选民意见的）民意测验的结果在 45%至 55%之间的概率是多少？（答案应该是 n、k 和百分数的表达式。）

6.59 民意测验的结果一般以某个“误差”范围给出。例如，得到的结果可能是：候选者 X 的得票率为 x%，且测验本身有±3%的误差。讨论为什么用绝对误差作为误差率的边界是不准确

的。应该如何精确定义误差边界?

6.60 本题旨在比较蒙特卡罗算法和拉斯维加斯算法。概括地说，蒙特卡罗算法能保证程序的运行时间，但却无法保证算法的正确性；而拉斯维加斯算法保证了正确性，却无法保证算法的运行时间。假设我们考察的问题是一个判定问题，答案只能为“是”或“不是”。再假定蒙特卡罗算法的错误概率不超过 1/4。(这个错误率足以让人接受，因为我们可以多运行几遍算法，然后取结果的众数，从而显著降低出错的可能性。)哪种算法更强大?换句话说，是否可能把其中一个算法转换为另一个?

6.61 设计一个算法，对已知的某个有 n 个元素的列表，找出所有出现超过 $n/4$ 次的元素。算法应该做 $O(n)$次比较。(提示：修改众数算法。)

6.62 要求你安排一个循环制网球巡回赛的赛程。比赛共有 $n=2^k$ 个选手参加，每个选手必须和其余选手一一比赛，且在 $n-1$ 天内每人每天都必须有一场比赛。记选手为 $P_1, P_2, \cdots, P_n$。算法输出每个选手的赛程。(提示：使用分治法，如下所示：首先把选手分成人数相等的两组，然后在 $n/2-1$ 天内每组分别进行比赛。接着在剩下的 $n/2$ 天内进行组间的比赛。

*6.63 为一个循环制网球巡回赛安排赛程(见习题 6.62)。选手数任意。若选手数为奇数，则每一轮有一名选手轮空。

*6.64 令 $r_1, r_2, \cdots, r_n$ 和 $c_1, c_2, \cdots, c_n$ 是两个整数序列，序列中所有元素之和相等，即 $\sum_{i=1}^{n} r_i = \sum_{i=1}^{n} c_i$。如果有一个 $n \times n$ 的矩阵，其元素非 0 即 1，使得对任意的 i，第 i 行的元素之和恰为 r_i，第 i 列之和恰为 c_i，则称这样的序列是可实现的，并非所有的序列都是可实现的。例如，序列 0,2 和序列 0,2 就不是可实现的。这是因为只有第二行的第二个元素非零，而它不应该比 1 大。设计判定两个已知序列是否可实现的算法。如果可实现，则构造一个对应行、列之和的矩阵。提示：首先加强归纳假设，把问题扩展到 $n \times m$ 的矩阵，然后对 n(行数)进行归纳。尽量把 1 放在第一行，这样其余 $n-1$ 行的问题可解当且仅当原问题可解。

第7章 图 算 法

捷径就是两点之间的最长距离。

——佚名

7.1 引言

前一章讨论了集合与序列的算法，研究的关系局限在排序、多重性、重叠等方面。本章将讨论对象之间的关系并用图来为这些关系建模。图可用于非常多的情况，包括从考古学到社会心理学等不同领域。下面是一些重要的对图进行操作并计算图的性质的基本算法。

首先是几个用图进行建模的例子。

1. 在城市里寻找一条好的到达某饭店的路线就是一个图论的问题。城市的街道对应于边（有向边对应于单行街），街道的交叉点对应于顶点。每个顶点和每条边（街道线段）可与一个预计的时间消耗相联系，则问题就是在两个顶点之间寻找“最快”的路径。
2. 一些程序可以划分成几种状态。从每种状态出发，程序会有若干种可能的进展状态。程序中的一些状态并不是期望的状态。寻找哪些状态将导致非期望的状态也是一个图论问题。这里状态对应于顶点，而边则表明可能的从一个状态到另一状态的转移。
3. 在大学中排课问题也可以看做是一个图论问题。顶点对应于课程，如果有学生希望同时选择两门课程，或者这两门课程由同一个教授上课，则它们之间有边相连。问题是安排一个冲突最小的课程表。这是一个难题，因为很难找到好的解决方案。
4. 考虑有多个用户账户的计算机系统。每个用户拥有一个用来访问其账户的安全许可。用户可能想进行协作，并授予其他用户使用自己的账户的许可。然而，如果 A 被许可使用 B 的账户，且 B 被许可使用 C 的账户，则 A 可能同时也可以使用 C 的账户。鉴别哪些用户可以使用哪些账户的问题，可看做一个图论问题。用户对应于顶点，如果用户 A 许可用户 B 使用自己的账户。则从 A 到 B 有一条有向边。

关于图论有相当多的教材（见参考文献），以及许多其他应用。

在 4.6 节中已经讨论过图的表示。本章在大部分情况下将使用邻接表来表示图，这对稀疏图（即图中的边相对较少）比较有效。下面就从引入标准术语开始。一个图 $G=(V, E)$包括一个**顶点**（也称为**节点**）的集合 V 和一个**边的**集合 E。每条边对应一对不相同的顶点。（有时自循环也是允许的，即一个顶点到自身有一条边；但现在假定不允许这种情况。）一个图可以是**有向**的或者**无向**的。有向图中的边是一个有序对；边所连接的两个顶点的次序很重要。在此情况下，把边画成从一个顶点（尾部）到另一顶点（头部）的箭头。无向图中的边是一个无序对；可简单地画成线段。若相同的顶点对之间有若干条边，则对应的图被称为**多重图**（即 E 为多重集）。不是多重图的图有时被称为**简单图**。除非特别指明，本章假定所处理的图都是简单的。一个顶点 v 的**度** $d(v)$ 是与 v 相连接的边的数目。对于有向图，要区分**入度**和**出度**，前者是以 v 为头部的边的数目，出度则是以 v 为尾部的边的数目。

从 v_1 到 v_k 的一条路径是一顶点序列 $v_1, v_2, \cdots, v_k$，它们通过边$(v_1, v_2), (v_2, v_3), \cdots, (v_{k-1}, v_k)$（这些边通常被看做是路径的一部分）相连接。如果每个顶点在路径中至多出现一次，则这条路径被称为**简单**的。如果从顶点 v 到顶点 u 存在一条路径，则称 u 是**可达**的。**回路**是一条起点与终点为同一顶点的路径。若除了第一个和最后一个顶点外，其他顶点至多只出现一次，则此回路被称为**简单**的。一个简单的回路也称为**闭链**。（有时即使回路不是简单的也称为闭链；但本章假定闭链是简单的回路。）一个有向图 $G = (V, E)$的**无向型**（undirected form）是一个与 G 相同的图，但其中的边没有方向性。对于一个图的无向型，若任意两个顶点之间都存在一条路径，则图被称为**连通**的。**森林**是一个不包含闭链的（无向型）图。一棵**树**则是一个连通的森林。**根树**（也叫**树形图**）是一棵有向树，其中一棵特定顶点被称为**根**，所有的边都远离这个根延伸。

图 $G = (V, E)$的一个**子图**是图 $H = (U, F)$，其中 $U \subseteq V$，$F \subseteq E$（译者注：并且 F 所关联的顶点都属于 U）。无向图 G 的一棵**生成树**是 G 的一个子图，它是一棵树并且包含 G 中所有顶点。无向图 G 的一个**生成森林**是 G 的一个子图，它是一个森林并且包含 G 中所有顶点。图 $G = (V, E)$的一个**顶点导出子图**是一个子图 $H = (U, F)$，其中 $U \subseteq V$ 且 F 包含其两个顶点属于 U 的 E 中的所有边。一个顶点导出子图常被简称为**导出子图**。如果图 $G = (V, E)$不是连通的，则可以用惟一的方式把它划分成一些连通子图的集合，这些连通子图被称为 G 的**连通分支**。图 G 的一个连通分支是一个连通子图，它不被任何其他连通子图所包含。换句话说，连同分支就是一个最大连通子图。**偶图**是一个顶点集合可以划分成两个集合的图，满足图中所有的边都只是关联属于不同的集合的顶点，也就是说，同一集合的顶点之间没有边。**加权图**是边附有权重（或者代价及长度）的图。

除了一些明显的差别，对于许多定义，有向图都与无向图相似。例如，有向路径和无向路径都是以同样的方式定义的，但是，在有向路径中边的方向要指明。当讨论某类图时，一般不再使用不同的标记，比如，当在有向图的上下文中讨论路径时，指的就是有向路径。

我们从一个简单例子，即遍历哥尼斯堡的桥开始，该例子通常被认为是第一个图论问题。然后，再讨论如何遍历一个图，如何对图的节点排序，如何在图中寻找最短路径，如何把图划分成满足一定性质的块等。第 10 章将包含图算法与矩阵算法之间关系的讨论，那里也有一些与图相关的算法。

7.2　欧拉图

欧拉图是第一个被解决的图论问题。1736 年，瑞士数学家 Leonhard Euler 遇到了下面这个难题。哥尼斯堡城镇坐落在 Pregel 河的两岸，并且在河中有两座小岛，如图 7.1 所示。这个城镇通过 7 座桥来连接。（众多居民想解决的）问题是，是否存在一种可能，从一处出发经过每座桥恰好一次最后回到出发点。通过对问题进行抽象可以得到解答。在这个问题上，对于图 7.1 的布局，图 7.2 即为等价的抽象。现在这个问题变成了一个图论问题，即是否可能在图中找到一条回路，它仅包含每条边各一次。问题的另一种方式是，是否可能一笔画出图 7.2，并在起笔处结束，而期间不提起笔。欧拉解决了这个问题，他证明了这种遍历是可能的当且仅当图是连通的并且所有顶点的度为偶数。这种图被称为**欧拉图**。由于图 7.2 中的图包含度为奇数的顶点，所以哥尼斯堡桥问题不可能有解。这个定理的证明可以通过归纳法，相应的是一个构造封闭路径的有效算法，这些内容随后将会给出。

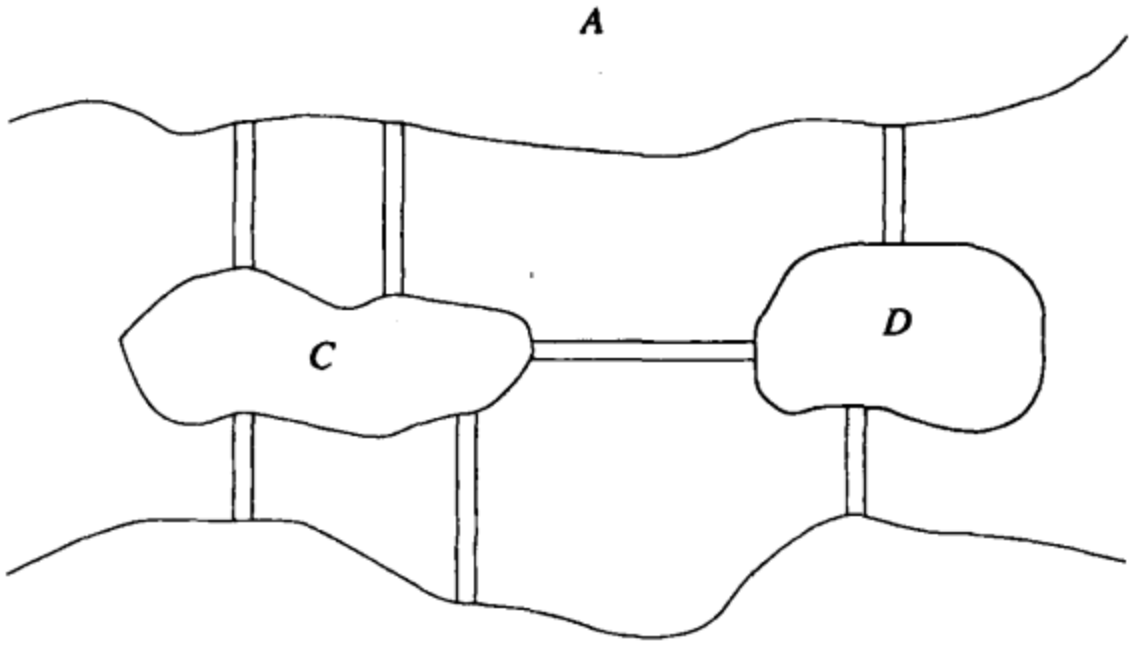


图 7.1 哥尼斯堡桥问题

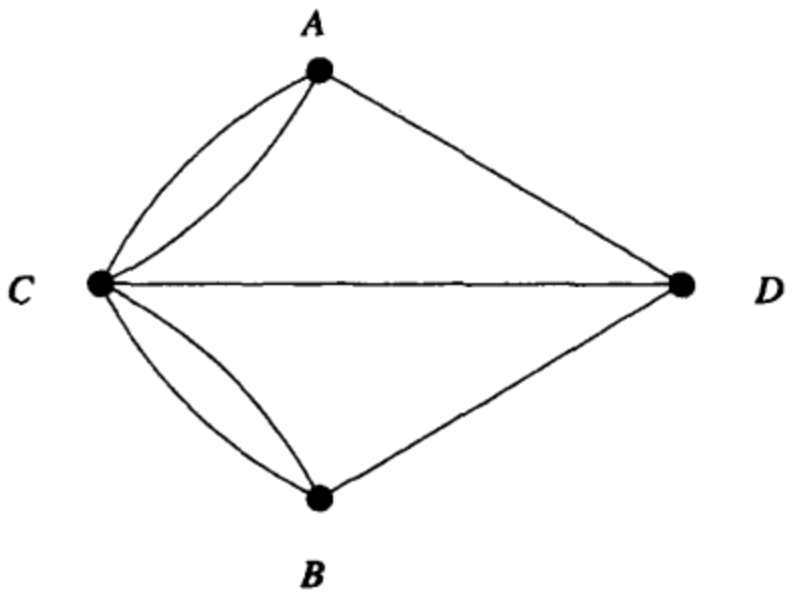


图 7.2 对应哥尼斯堡桥问题的图

问题 给定一个无向连通图 $G=(V,E)$，其所有顶点的度为偶数，寻找一个封闭路径 P，使得 E 中的每一条边在 P 中仅出现一次。

容易证明，若存在这样的封闭路径，则所有顶点的度必须为偶数：当遍历一条封闭路径时，我们进入和离开每一个顶点的次数是相同的。由于每条边仅被使用一次，则与每个顶点相连的边数必定为偶数。要用归纳法证明这个条件是充分的，我们首先必须决定应用到归纳中的参数。首要的考虑是能够简约问题而又不改动问题。若删除一个顶点或一条边，则得到的图可能不满足度为偶数这个性质。我们应该删除一个边的集合 S，使得对于图中的每个顶点 v，来自于 S 中且与 v 相连的边数为偶数（也可能为 0）。任何回路都满足这个要求，所以问题是欧拉图是否总包含一个回路。假设我们开始遍历一个图，从任意顶点 v 开始，按任意次序经过每条边至多一次。我们称这个遍历最终回到 v，因为，当进入另一顶点时，该顶点度的数目就减少 1，变成了奇数，从而总有可能从这个顶点离开。（注意，这个回路可能不包含所有的边。）

现在，准备声明归纳假设并证明这个定理。

归纳假设：对于边数小于 m 的连通图，其所有顶点的度为偶数，则存在一条包含每

条边仅一次的封闭路径，并且我们知道如何找到这条路径。

（虽然归纳是作用在路径上，但根据边的数目要比根据封闭路径的数目来声明归纳假设容易些。）考虑一个有 m 条边的图 $G=(V, E)$。令 P 是 G 中的一条封闭路径。令 G'是一个从 G 中删除所有 P 中的边所得到的图。因为删除的与任意顶点相连的边的数目是偶数，所以 G'中所有顶点的度必定为偶数。但仍然不能简单地应用归纳假设，因为 G'可能是不连通的。记 $G_1', G_2', \cdots, G_k'$ 为图 G' 的连接分量。在每一个分支中，所有顶点的度为偶数。进一步，在每一个分支中边的数目（事实上，它们的总和）小于 m。所以，我们可以把归纳假设应用到每一个分支上。即由归纳假设，每一个分支都有一条包含每条边仅一次的封闭路径，并且知道如何找到它。用 $P_1, P_2, \cdots, P_k$ 表示这 k 条封闭路径。现在需要把这些路径合并成一个覆盖整个图 G 的封闭路径。我们从 P 中任意顶点开始对 P 进行遍历，直至遇到第一个属于某个分支 G_j'的顶点 v_j。在这一点上，我们遍历路径 P_j并返回 v_j。可以继续这个方法，当第一次遇到某个分支时就遍历它的封闭路径，直至回到开始的顶点。回到这一点时，所有的边已被恰好走过一次。这条封闭路径就叫做**欧拉回路**。算法仍然没有结束。我们还需要找到一个有效的方法来鉴别连通分支，以及有效的方法来遍历一个图。这两个问题随后将讨论，另外欧拉回路算法的实现留做习题。

7.3 图的遍历

设计图算法遇到的第一个问题是如何查看输入。在前一章，这个问题是平凡的，因为输入都是一维的，序列和集合等都可以很容易地按线性次序进行扫描。而扫描一个图，或者说**遍历**一个图，并不那么直接。这里提供两个遍历算法，**深度优先搜索**（DFS）和**广度优先搜索**（BFS）。本章中的绝大多数算法都将依赖于其中一种方式。

7.3.1 深度优先搜索

有向图和无向图的深度优先搜索几乎是一样的。然而，由于想探究一些在有向图和无向图之间不同的特性，我们分成两部分来讨论。

无向图

假设无向图 $G=(V, E)$对应于一个艺术画廊，它由一些挂着绘画的走廊组成。G 的边对应于走廊，而顶点对应于走廊的交叉处。我们想走遍这个画廊，从而看到所有的绘画。假定从任意方向经过走廊，都可以看到走廊两侧的画。如果图是欧拉图，则仅路过每条走廊各一次并走遍画廊是可能的。在这里不假定该图为欧拉图，我们允许每条边都可以经过多次（当返回时，这条边经过了两次）。隐藏在深度优先搜索背后的思想如下。当在画廊中行走时，一遇到新的走廊就进去。当第一次遇到交叉口时，在那里放一块石子，然后继续走入另一走廊（除非这里没有出路）。当到达一个曾经留有石子的交叉口时，就从过来的那个走廊返回，再尝试另一个走廊。如果所有从这个交叉口出发的走廊都已经访问过，则我们把这个交叉口的石子拿掉，并返回第一次进入的那条走廊。然后不再游览这条走廊。（把石子拿掉仅仅是为了清理画廊；它在算法中并不是要点。）我们总是试图探究一条新的走廊；只有尝试过所有其他走廊后，才从到达过第一个交叉口的走廊返回。这种方法被称为**深度优先搜索**（DFS），表明首先是要尝试新的边（更深地进入画廊）。DFS 有效性的主要原因是它划分图的方式以及易于使用递归算法。

根据遍历与置放石子，我们给出 DFS 的描述。现在考虑，对于一个用邻接表表示的无向图，DFS 如何实现。遍历是从任意一个顶点 r 开始，它被称为 **DFS 的根**。这个根被**标记**为被访问过。

然后，从与 r 连接的顶点中任意挑一个（未被标记过的）r_1，再从 r_1 开始（递归）执行 DFS。当到达一个顶点 v 时，它的所有连接顶点都已被标记过，此时递归停止。如果当对 r_1 进行 DFS 终止后，所有与 r 连接的顶点都已被标记过，则对 r 进行的 DFS 终止。否则，在与 r 连接的顶点中任选一个未被标记过的顶点 r_2，从 r_2 开始进行 DFS，如此等等。

通常，遍历一个图都有一个目的。为了把 DFS 框架与不同的应用结合起来，可把两类工作与访问顶点或者访问边相联系，即 *preWORK* 与 *postWORK*；*preWORK* 是在顶点被标记的时候执行，而 *postWORK* 是在从一个边原路返回后或者发现这条边导向一个被标记过的顶点后执行。*PreWORK* 和 *postWORK* 都依赖于与 DFS 的运用。通过仅仅定义 *preWORK* 和 *postWORK*，就可以提供若干应用。DFS 程序在图 7.3 中给出。递归调用的起始顶点被称为 v。为简单起见，先假定图是连通的。在图 7.4 中给出一个例子，其中与顶点相关联的数字表示通过 DFS 遍历时顶点被访问的次序。

算法 ***Depth_First_Search*** **(*G*, *v*)**
输入： $G=(V, E)$（一个无向连通图）和 v（G 中的一个节点）
输出： 与应用相关

```
begin
    mark v ;
    perform preWORK on v ; {preWORK 依赖于 DFS 的应用}
    for all edges (v, w) do
        if w is unmarked then Depth_First_Search(G, w) ;
        perform postWORK for (v, w)
        {postWORK 依赖于 DFS 的应用；有时仅对导向
        新标记的节点的边执行该操作}
end
```

图 7.3　深度优先搜索算法

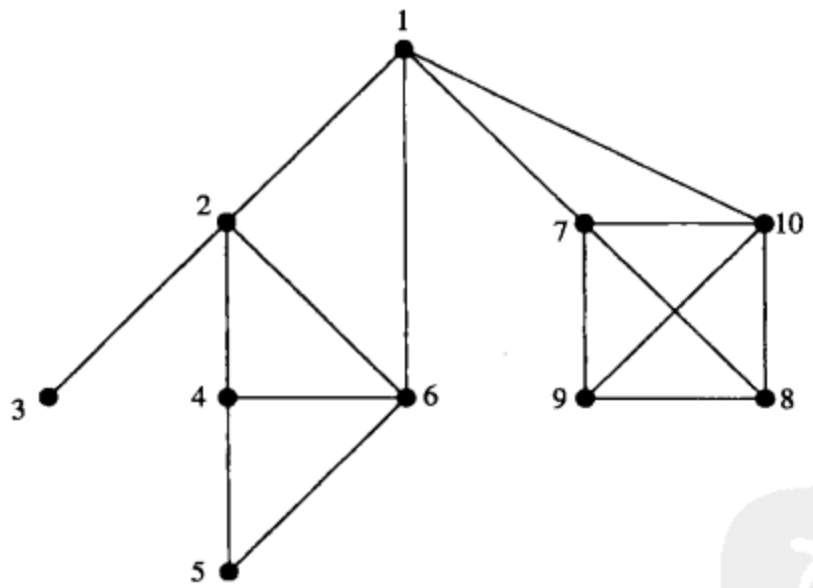


图 7.4　一个无向图的 DFS

□ 引理 7.1

如果 G 是连通的，则通过深度优先搜索可以对它的所有顶点进行标记，并且在算法的执行过程中，它的每条边至少被查看过一次。

证明： 假设结论不成立，令 U 表示算法最终未被标记过的顶点集合。由于 G 是连通的，因此在 U 中至少有一个顶点与一个被标记过的顶点相连。但是这种情况不可能发生，因为一旦一个顶点被访

问过了，则所有与它相连的未被标记过的顶点也会被访问（从而也被标记）。由于所有的顶点都被访问过，并且一旦某个顶点被访问，它相连的边就会被查看，所以每一条边都将被查看过。　□

如果一个图 $G=(V, E)$不是连通的，那么我们必须对 DFS 稍微做些修改。若在第一次尝试后所有的顶点都被标记过，则图是连通的，且任务完成；否则，从任意一个未被标记的顶点开始，执行另一个 DFS，如此等等。所以，我们可以利用 DFS 确定一个图是否连通，并找出它的连通分支。对应的算法在图 7.5 中给出。通常仅考虑连通图，否则要对每个连通分支分别处理。所以，可以如图 7.3 描述的那样使用 DFS，而不是像图 7.5 那样特别注明必须使用多少次。

算法 ***Connected_Components (G)***
输入： $G=(V, E)$（一个无向图）
输出： 对每个节点 v 输出 *v.Component*，它是包含 v 的连通成分的数目

```
begin
    Component_Number := 1 ;
    while there is an unmarked vertex v do
        Depth_First_Search(G, v) ;
        ( using the following preWORK:
            v.Component := Component_Number ; )
        Component_Number := Component_Number + 1
end
```

图 7.5　连通分支算法

复杂度　容易看出，每条边恰好被查看过两次（从每一端点各一次）。所以，运行时间与边的数目成比例。然而，由于图中可能包含许多不与任何东西相连的顶点（这些必须被检测出来），所以在运行时间的表达式中必须包括 $O(|V|)$。因此总的运行时间是 $O(|V|+|E|)$。

构造 DFS 树

接下来是两个 DFS 的简单应用，使用 **DFS 数**对顶点进行计数，以及构造一棵特别的生成树，称之为 **DFS 树**。DFS 数和 DFS 树展现了对许多算法都有用的特殊性质。即使没有显式地把树构造出来，通过考虑它也易于理解许多算法。为了描述这些算法，这里仅需要定义 *preWORK* 或 *postWORK*。对顶点进行 DFS 计数的算法在图 7.6 中给出，而构造 DFS 树的算法在图 7.7 中给出。这两个算法不需要分别执行。

在一棵以 r 为根的树中，如果顶点 v 位于从顶点 w 到 r 的一条惟一路径 T 上，则 v 被称为 w 的**祖先**。若 v 是 w 的一个祖先，则 w 被称为 v 的**后代**。

算法 ***DFS_Numbering (G, v)***
输入： $G=(V, E)$（一个无向图）和 v（G 中的一个节点）
输出： 对每个节点 v 输出 *v.DFS*，即 v 的 DFS 值

```
Initially DFS_Number := 1 ;
Use DFS with the following preWORK:
preWORK:
    v.DFS := DFS_Number ;
    DFS_Number := DFS_Number + 1 ;
```

图 7.6　DFS 计数算法

算法 *Build_DFS_Tree* (*G*, *v*)
输入： $G=(V, E)$（一个无向图）和 v（G 中的一个节点）
输出： T（G 的一棵 DFS 树，初始为空）
Use DFS with the following postWORK:
postWORK：如果 w 未被标记，则将(v, w)加入 T 中
{上面的语句可被包含在算法 *Depth_First_Serach* 第 4 行的 *if* 语句中}

图 7.7 构造 DFS 树算法

□ **引理 7.2 （无向 DFS 树的主要性质）**

令 $G=(V, E)$是一个连通的无向图，且令 $T=(V, F)$是由算法 *Build_DFS_Tree* 构造出的 G 的一棵 DFS 树。E 中的每一条边 $e\in E$，或者属于 T（即 $e\in F$），或者连接 G 中的两个顶点，它们其中一个是 T 中的另一个的祖先。

证明：令(v, u)是 G 中的一个边，假设 v 在 u 之前被 DFS 访问。在 v 被标记过后，我们从所有 v 的未被标记过的邻居开始执行 DFS。由于 u 是 v 的一个邻居，则 DFS 或者从 u 开始，在这种情况下(v, u)将属于 T，或者 DFS 在从 v 回溯之前访问 u，在这种情况下，在 T 中 u 是 v 的后代。 ☒

换句话说，DFS 避免了**交叉边**，这种边连接顶点时横向穿过一棵树。后面将看到，避免交叉边对于在图中执行递归过程是很重要的。

由于 DFS 是一个非常重要的程序，这里也给出它的非递归版本。实现非递归程序的主要工具是栈，它保存“展开”递归调用所需的信息。一个编译器在栈上维护所有与每一个递归事例相关联的局部数据。因此，当一个递归事例终止时，我们可以（根据确切的信息）回到调用过程（它可能是同一递归调用的另一事例）中确切的点。通常，不是所有的局部数据都需要维护在栈中，这也是使用非递归过程更有效率的原因之一。接下来给出的非递归版本是一个把递归程序转为非递归程序的好例子。

在把递归版本转为非递归版本时，面对的一个主要困难是需要显式的记账本（bookkeeping）。我们在一个 for 循环中递归调用 DFS，并且希望程序记住在循环中正确的位置，从而在递归调用之后可以继续执行。在非递归版本中，我们必须显式地维护这些信息。假定每一个顶点 v 有一个连接的列表，按一定次序记录了相关联的边（DFS 将遵照这个次序）。这个列表由 *v.First* 指向。列表中的每一项是包含两个变量的记录：*Vertex* 和 *Next*。*Vertex* 是边的另一端顶点的名字，而 *Next* 则指向下一项。在列表中最后一条边的 *Next* 指向 nil。DFS 的进行就像前面一样，遍历整棵树直至没有新顶点可被找到。在搜索的整个过程中维护着一个栈，这个栈保留了从根到当前顶点的路径上的所有顶点（依照路径的顺序）。在每两个顶点 *Parent* 和 *Child* 之间，栈保留了一个从 *Parent* 到边的指针，当从 *Child* 回溯时这是接着的下一个 DFS 遍历。DFS 的非递归版本在图 7.8 中给出。

有向图

对于有向图，它的 DFS 过程与无向图一样。然而，有向的 DFS 树却有不同的性质。不存在交叉边的性质已不成立，这在图 7.9 中可以看到。现在有 4 种类型的边——**树边**、**后退边**、**前向边**和**交叉边**。前三类边连接两个顶点，其中一个是另一个在树中的后代：树边把树中的 *Parents* 连接到 *Children*，后退边把后代连接到祖先，而前向边把祖先连接到后代。只有交叉边连接树中无“关联”的两个顶点。然而，交叉边必须“从右到左”交叉，这在下面的引理中可以看到。

```
算法 Nonrecursive_Depth_First_Search (G, v)
输入：G=(V, E)（一个无向连通图）和 v（G 中的一个节点）
输出：与应用相关
    {显式使用 Pascal 语言的指针符号^，本章的其他部分并不是如此}
begin
    while there is an unmarked vertex v do
        mark v ;
        perform preWORK on v ;
        Edge := v.First ;
        push v and Edge to the top of the stack ;
        Parent := v ;
        {初始化到此，接下来的是递归的主循环}
        while the stack is not empty do
            remove Edge from the top of the stack ;
            while Edge ≠ nil do
                Child := Edge^.Vertex ;
                if Child is unmarked then
                    mark Child ;
                    perform preWORK on Child ;
                    push Edge^.Next to the top of the stack ;
                    {这样当处理完 Child 后就可以回到下一条边}
                    Edge := Child.First ;
                    Parent := Child ;
                    push Parent to the top of the stack ;
                else{Edge 是一条后退边}
                    perform postWORK for (Parent, Child) ;
                    {如果仅对树的边执行 postWORK，则本步可以略去}
                    Edge := Edge^.Next ;
            remove Child from the top of the stack ;
            if the stack is not empty then
            {当 Child 是根时栈为空}
                let Edge and Parent be at the top of the stack ;
                    {不可将其移去}
                perform postWORK for (Parent, Child)

end
```

图 7.8 非递归的深度优先搜索算法

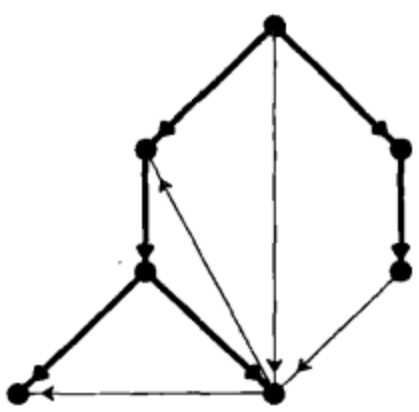


图 7.9 有向图中的一棵 DFS 树

□ **引理 7.3 （有向 DFS 树的主要性质）**

令 $G=(V, E)$是一个有向图，且令 $T=(V, F)$是 G 的一棵 DFS 树。如果(v, w)是 E 中的一条边，满足 $v.DFS_Number < w.DFS_Number$，则 w 是 v 在树 T 中的一个后代。

证明：由于 w 的 DFS 数大于 v 的 DFS 数，所以 w 在 v 之后被访问。既然(v, w)是 E 中的一条边，则(v, w)必须在 v 的 DFS 过程中被查看。如果此时 w 没有被标记过，则把(v, w)加入树中；因此，$(v, w)\in F$，条件满足。否则，从 v 开始进行 DFS 递归调用后 w 被标记。因此，w 必须是 v 在树 T 中的一个后代。 ☒

对于连通无向图的 DFS，从任意顶点开始，即可遍历整个图。但对于有向图则非如此。考虑图 7.10 中的有向图。例如，若从 a 开始进行 DFS，则只有左边一列可以遍历到。仅当从 v 开始进行 DFS，才能遍历图 7.10 中的整个图。如果图中的 v 和与之相连的两条边被删除，则不存在可以通过 DFS 遍历整个图的顶点。我们必须再次从一个未被标记的顶点开始，一直继续下去直到所有的顶点被标记。所以，当讨论有向图的 DFS 时，我们假定它要运行到所有的顶点都被标记过，且所有边都被查看过。

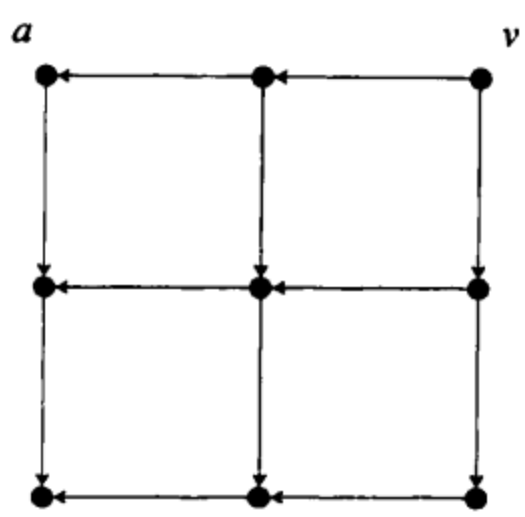


图 7.10　不能遍历整个图的有向图的例子

作为例子，下面利用 DFS 来确定一个图是否是非循环的。

问题　给定一个有向图 $G=(V, E)$，确定它是否含有一个（有向）闭链。

□ **引理 7.4**

令 $G=(V, E)$是一个有向图，且令 T 是 G 的一棵 DFS 树，则当且仅当 G 含有一条（相对于 T 的）后退边时，G 含有一个有向闭链。

证明：若存在一个后退边，则它将导向一个在树中更高层的顶点，所以形成一个闭链。反过来，令 C 是 G 中的一个闭链，v 是 C 中一个具有最小 DFS 数的顶点。下面说明指向 v 的边(w, v)是一个后退边：首先其不可能是一个前向边或者树边，因为它从一个具有较高 DFS 数的顶点指向一个具有较低 DFS 数的顶点。假设 v 不是 w 在树中的祖先，令 u 是 v 和 w 的最小的共同祖先。由于 v 的 DFS 数小于 w 的 DFS 数，所以它在 u 的子树中且在包含 w 的 u 的子树之前被访问。这意味着从 v 到达 w 的惟一途径是经过 u 或者经过 u 的一个祖先（因为它不可能“从左到右”这么走）。但是，C包含一条从 v 到 w 的路径，且 C 不可能含有 v 的祖先，因为 v 在 C 中具有最小的 DFS 数。 □

确定一个有向图是否非循环的算法在图 7.11 中给出。

算法 *Find_a_Cycle (G)*
输入： $G=(V, E)$（一个有向图）
输出： *Find_a_Cycle*（若 G 包含一个闭链，则值为真；否则为假）

Use DFS, starting from an arbitrary vertex, with the following preWORK and postWORK:

preWORK:
　　v.on_the_path := true ;
{如果 x 位于从根到当前节点的路径上，则 *x.on_the_path* 的值为真}
{初始时，对于所有的节点 x，*x.on_the_path* 的值为假，*Find_a_Cycle* 的值也为假}
postWORK:
　　if w.on_the_path then Find_a_Cycle := true ; halt ;
　　if w is the last vertex on v's list then v.on_the_path := false ;

图 7.11　寻找闭链的算法

7.3.2　广度优先搜索

逐层遍历一个图的广度优先搜索（BFS）似乎更有序。如果我们从顶点 v 开始，则首先访问 v 的全部儿子。第二层就是访问全部“孙子”，依次类推（见图 7.12）。这个遍历的实现类似于 DFS 的非递归实现，除了把栈替换成队列。我们可以把与 DFS 数类似的 BFS 数关联到每个顶点。即，如果顶点 w 是通过 BFS 被标记的第 k 个顶点，则其具有的 BFS 数为 k。通过只保留导向新访问顶点的边，可以构造一棵 BFS 树。BFS 算法在图 7.13 中给出。（在 BFS 中没有像 DFS 那样定义 *postWORK*，因为直觉上它的搜索过程并不是“先下后上”进行的，它只有“下向”；所以可以忽略。）

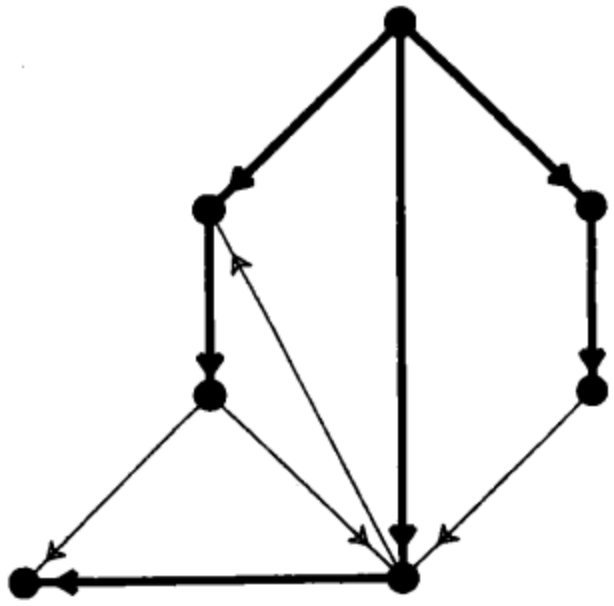


图 7.12　一棵有向图的 BFS 树

□ 引理 7.5

如果边 (u, w) 属于一棵 BFS 树，其中 u 是 w 的父母，则在具有导向 w 的边的顶点中，u 具有最小的 BFS 数。

证明：这个断言来自队列的先进先出特性。　□

算法 ***Breadth_First_Search*** *(G, v)*
输入： $G=(V, E)$（一个无向连通图）和 v（G 中的一个节点）
输出： 与应用相关

```
begin
    mark v ;
    put v in a queue {先进先出}
    while the queue is not empty do
        remove the first vertex w from the queue ;
        perform preWORK on w ;
          {preWORK 依赖于 BFS 的具体应用}
        for all edges (w, x) such that x is unmarked do
            mark x ;
            add (w, x) to the tree T ;
            put x in the queue
end
```

图 7.13 广度优先搜索算法

□ **引理 7.6**

对于每一个顶点 w，在 T 中从根到 w 的路径是在 G 中从根到 w 的最短路径。

证明： 留给读者。 □

顶点 w 的**层数**是在树中从根到 w 的路径长度。BFS 是逐层对图进行遍历。

□ **引理 7.7**

如果(v, w)是 E 中的一条边且不属于 T，则它连接的两个顶点的层数至多相差 1。

证明： 留给读者。 □

现在读者应该了解了如何遍历一个图，下面将提供一些与图有关的算法并再一次尽力展示归纳设计的技巧。

7.4 拓扑排序

假设某时需要执行一组任务，其中一些任务依赖于其他的一些任务，因此前者必须在其他任务完成后才能开始。若所有的依赖性皆为已知，现在想为这些任务安排一个进度表，使得任务的执行次序符合给定的依赖性（即，每个任务只有在其所依赖的所有任务完成后，才能被执行）。我们想设计一个快速算法来产生这样一个进度表。这个问题称为**拓扑排序**。通过下面的方法，我们可以把一个有向图与这些任务及其依赖性关联起来，即每被一个任务与一个顶点相关联，如果只有在任务 x 完成后任务 y 才能开始，则从 x 到 y 画一条有向边。显然，这个图应该是非循环的；否则，一些任务永远都不能开始执行。

> **问题** 给定一个有向非循环图 $G=(V, E)$，它有 n 个顶点，把顶点按照 1 到 n 做标记，满足，若 v 被标记为 k，则通过有向路径从 v 出发可以到达的所有顶点的标记大于 k。

直接归纳假设如下所示。

归纳假设： 已知如何根据以上条件对顶点数小于 n 的有向非循环图做标记。

只有一个顶点的基础情况是平凡的。通常，考虑一个具有 n 个顶点的图，删除一个顶点，应用归纳假设尝试扩展标记。由于可以自由选择任何顶点作为第 n 个顶点，所以，应该选择可以简化工作的顶点。若需要对定点做标记，那么哪一个顶点最容易被标记呢？显然，应该是没有依赖性的顶点（任务），即入度为 0 的顶点。毫无疑问这个顶点可以被标记为 1。我们总是能找到一个入度为 0 的顶点吗？直觉上答案是肯定的，因为我们必须能够从某一处开始。下面的引理确定了这个事实。

□ **引理 7.8**

一个有向非循环图总有一个入度为 0 的顶点。

证明：如果所有的顶点都有一个正数的入度，则我们能够“向后”遍历这个图并永不停止。由于只有有限个顶点，所以必然进入一个循环，但这在一个非循环图中是不可能的。（使用同样的讨论可知，存在一个出度为 0 的顶点。） □

我们将很快看到如何找到一个入度为 0 的顶点。一旦找到它，就把它标记为 1，并删除与它相连的边，再对剩下的图进行标记，当然，它仍然是非循环的，而标记是从 2 到 n。（为了完全精确，归纳假设假定标记从 1 到 n–1，而非 2 到 n，但这并不会造成什么问题。）注意，一旦我们决定选择一个入度为 0 的顶点进行归约，则算法即可毫不费力地进行。

实现 关于实现的惟一问题是如何找到一个入度为 0 的顶点，以及在顶点被删除后如何调整其他顶点的入度。我们把变量 *Indegree* 关联到每一个顶点，初始时 *v.Indegree* 等于 v 的入度。变量 *Indegree* 可以通过任何次序（例如使用 DFS）遍历所有的边来进行初始化，并在遍历过一条边 (v,w) 后对 *w.Indegree* 进行递增。入度为 0 的顶点被放入一个队列（一个栈同样可以）。由引理 7.8，至少有一个入度为 0 的顶点 v。找到 v 很容易，然后把它简单地从队列中删除。然后，对 v 出发的每条边 (v,w)，让 w 的计数器递减 1。当一个计数器变成 0 时，该顶点就被放入队列。删除 v 后剩余的图仍然是非循环的。所以，由引理 7.8，在剩余的图中至少有一个入度为 0 的顶点。当队列为空时，算法终止，此时所有的顶点都已被标记。算法在图 7.14 中给出。

算法 *Topological_Sorting (G)*
输入： $G=(V, E)$（一个有向无回路图）
输出： *Label* 域的值指出图 G 的一个拓扑序

```
begin
    Initialize v.Indegree for all vertices ;{例如通过 DFS 算法}
    G_label := 0 ;
    for i := 1 to n do
        if v_i.Indegree = 0 then put v_i in Queue ;
    repeat
        remove vertex v from Queue ;
        G_label := G_label + 1 ;
        v.label := G_label ;
        for all edges (v, w) do
            w.Indegree := w.Indegree – 1 ;
            if w.Indegree = 0 then put w in Queue ;
    until Queue is empty
end
```

图 7.14 拓扑排序算法

复杂度 对变量 *Indegree* 进行初始化需要 $O(|V|+|E|)$ 时间。找到一个入度为 0 的顶点需要常数式时间（访问一个队列）。（当 v 从队列中删除时）每一条边(v, w)被查看一次。因此，变量需要被更新的次数正好等于图中边的数目。从而算法的运行时间是 $O(|V|+|E|)$，这对于输入规模是线性的。

7.5 单源最短路径

本节处理**加权图**。令 $G=(V, E)$是一个有向图，在每条边上有一个非负的**权重**。本节中权重为长度，因为传统上把这个问题称为最短路径问题（而不是最轻路径问题）。（有时路径的长度表示的是路径中边的数目，因此必须谨慎以免引起混淆。）若图是无向的，则可通过把每一条无向边对应为两条具有同样长度的（方向相反）有向边，由此可得有向图。因此，本节的讨论同样适用于无向图。**路径的长度**是路径中各条边长度的总和。

> **问题** 给定一个有向图 $G=(V, E)$及一个顶点 v，寻找到从 v 出发到达 G 中其他各顶点的最短路径。

为了简化，这里仅讨论如何找到最短路径的长度。算法可以扩展成找到实际的路径。最短路径问题有许多例子。例如，图可以对应于路线图，一个片断的长度可对应于它的实际长度，或者对应于行进所要花费的时间，或者对应于建造它所花的费用，这都取决于各自问题。

非循环情况

先假定图是非循环的。在这种情况下问题比较容易，它的解答可以有助于找到一般情况下的解答。我们尝试对顶点的数目进行归纳。基础情况是平凡的。令$|V|=n$。可以使用上一节讨论过的拓扑排序。如果 v 的标记是 k，则所有标记小于 k 的顶点都无须考虑。从 v 出发没有路线可以到达这些顶点。再有，通过拓扑排序得到的次序是一个好的归纳次序。考虑最后一个顶点，即标记为 n 的顶点 z。（归纳的）假设已知从 v 出发到达除 z 以外的所有顶点的最短路径。用 *w.SP* 表示从 v 到 w 的最短路径长度。为了找到 *z.SP*，我们仅需要检查那些有边导向 z 的顶点 w。由于已知到达所有其他顶点的最短路径，则在全部有边到达 z 的 w 中，计算 *w.SP+length*(w, z)，取 *z.SP* 等于最小值。这样完成了吗？必须谨慎增加 z，使它不会缩短到其他顶点的距离。但是，由于 z 是拓扑排序中最后的顶点，在图中不存在其他从 z 出发可以到达的顶点，所以其他路径都不受影响。因此，通过删除 z，计算它不在时的最短路径，再把它放回去，就解决了整个问题。相应的归纳假设如下。

> **归纳假设**：给定一个拓扑排序，已知如何找到从 v 出发到前面$(n-1)$个顶点的最短路径的长度。

给定一个按照拓扑排序的 n 个顶点的非循环图，删除第 n 个顶点，通过归纳解决精简后的问题，然后在所有 w 中取 *w.SP+length*(w, z)的最小值，其中$(w, z)\in E$。算法在图 7.15 中给出。现在改进算法，使得拓扑顺序可以连同最短路径同时被找到。换句话说，想将两个流程——一个是拓扑排序而另一个是最短路径——合并成一个流程。

考虑（在拓扑顺序找到之后）算法的递归执行方式。为了简单起见，假定 v 的标记在拓扑顺序中是 1。第 1 步是调用递归过程。它将重复调用自己直至 v 已经到达。此时，到 v 的最短路径的长度被置为 0，递归开始展开。接下来将考虑标记为 2 的顶点 u，它的最短路径将置为从 v 到 u 的边的长度，如果存在的话；否则，从 v 到 u 没有路径。下一步将检查标记为 3 的顶点 x。在此情况下，可能从 v 到 x 有边并且（或者）从 u 到 x 有边，则需要比较相应的路径。并不是在某种意义上“回溯的”应用递归，我们现在试图在标记顺序递增时执行同样的步骤。

```
算法 Acyclic_Shortest_Paths (G, v, n)
输入：G=(V, E)（一个加权无回路图）、v（节点）和 n（节点数）
输出：对于每一个节点 w∈V，w.SP 的值为从 v 到 w 的最短路径的长度
{假定已经执行拓扑排序，图 7.16 中给出了同时计算拓扑序的改进算法}
begin
    let z be the vertex labeled n{在拓扑排序中}
    if z ≠ v then
        Acyclic_Shortest_Paths (G − z, v, n − 1) ;
        {G–z 是 G 去掉与 z 相关联的边的结果}
        for all w such that (w, z) ∈ E do
            if w.SP + length(w, z) < z.SP then
                z.SP := w.SP + length(w, z) ;
    else v.SP := 0
end
```

图 7.15　非循环最短路径算法

归纳被用于从 v 开始的标记递增次序中。这个次序将消除对于事先知道标记的需要，从而能够同时运行两个算法。假定已知到达标记为 1 至 m 的顶点的最短路径的长度，考虑标记为$(m+1)$的顶点，不妨称之为 z。为了找到到达 z 的最短路径，我们需要检查所有指向 z 的边。拓扑排序保证了所有这样的边来自于较小标记的顶点。由归纳假设，这些顶点已经被考虑过了；因此，到达它们的最短路径的长度都已经知道。对于每一个边(w, z)，我们知道到达 w 的最短路径长度，$w.SP$，因此从这条边到 z 的最短路径是 $w.SP+length(w, z)$。所以，到达 z 的最短路径长度是：在所有 w 中，取 $w.SP+length(w, z)$的最小值。进一步，与前面一样，我们不需要担心调整到达小标记顶点的最短路径，因为不存在从 z 出发到达它们的路径。改进过的算法在图 7.16 中给出。

```
算法 Improved_Acyclic_Shortest_Paths (G, v)
输入：G=(V, E)（一个加权无回路图）、v（节点）和 n（节点数）
输出：对于每一个节点 w，w.SP 的值为从 v 到 w 的最短路径的长度
{本算法是前面一个算法的非递归版本，另外还包括节点的拓扑排序}
begin
    for all vertices w do
        w.SP := ∞ ;
    Initialize v.indegree for all vertices ; {例如通过 DFS}
    for i := 1 to n do
        if v_i.indegree = 0 then put v_i in Queue ;
    v.SP := 0 ;
    repeat
        remove vertex w from Queue ;
        for all edges (w, z) do
            if w.SP + length(w, z) < z.SP then
                z.SP := w.SP + length(w, z) ;
```

图 7.16　改进过的非循环最短路径算法

```
            z.indegree := z.indegree − 1 ;
            if z.indegree = 0 then put z in Queue ;
        until Queue is empty
    end
```

图 7.16 改进过的非循环最短路径算法（续）

复杂度 在顶点入度初始化和边的尾部从队列中删除时，每条边都被检查一次。队列的访问时间是常数式时间。每一个顶点只被考虑过一次。所以，最坏情况的运行时间是 $O(|V|+|E|)$。

一般情况

当图是循环的时，不存在类似的拓扑排序，我们刚讨论过的算法不能直接使用。然而，对于一般情况使用这些算法的思想，则是可能的。我们提供的算法简易性基于以下拓扑排序的特性：

> 如果 z 是一个标记为 k 的顶点，则（1）不存在从 z 出发到达其他标记小于 k 的顶点的路径（2）不存在从标记大于 k 的顶点到达 z 的路径。

这个性质使得我们可以找到从 v 到 z 的最短路径，而无须考虑那些拓扑排序在 z 之后的顶点。可以用某种方式在一般图的顶点上定义一个次序，从而允许我们做类似的事情吗？

思路是按照从 v 出发的最短路径长度的顺序来考虑图中顶点。当然，初始时并不知道这些长度；我们将在算法的执行过程中找到它们。首先，检查从 v 出发的所有边，令(v, x)是它们之间长度最小的边。由于所有长度都是正数，所以从 v 到达 x 的最短路径就是边(v, x)。其他从 v 出发的边至少也是同样长度。所以，我们知道了到达 x 的最短路径，这可以作为归纳的基础情况。进一步，如何才能找到下一个顶点的最短路径呢？我们选择次靠近 v 的顶点（x 是最靠近它的顶点）。需要考虑的惟一路径是从 v 出发的另一条边或者是包含两条边的路径，第一条边是(v, x)而第二条是从 x 出发的边。我们选择使 $length(v, y)$（$y \neq x$）或者 $length(v, x)+length(x, y)$（$y \neq v$）值最小的 y。再一次，无须考虑任何其他路径，因为这就是从 v 出发的最短路径（除了 x）。下面是一般的归纳假设。

> **归纳假设**：给定一个图和顶点 v，已知与 v 最近的 k 个顶点，以及到达它们的最短路径长度。

注意，这里的归纳假设作用在已计算出最短路径的顶点数上，而不是图的规模上。并且，它假定这些是最靠近 v 的的顶点，可以对它们进行鉴别。我们知道如何找到最靠近的顶点（在 x 之上），所以 $k=1$ 的基础情况是已解决了的。如果 $k=|V|-1$，则整个问题求解完成。

用 V_k 表示包含 v 在内的最靠近 v 的 k 个顶点集合。问题是要在不属于 V_k 的顶点中找到一个最靠近 v 的顶点 w。从 v 到 w 的最短路径只能经过 V_k 中的顶点。它不可能包含不属于 V_k 的顶点，否则会比 w 更靠近 v。所以，为了找到 w，仅考虑从 V_k 顶点连接到非 V_k 顶点的边已足够；现在所有其他的边都可以忽略。令(u, z)是一条边，其 u 在 V_k 中，z 则不在。这样的一条边对应于一个从 v 到 z 的路径，它包含从 v 到 u 的最短路径（已由归纳得知）和边(u, z)。我们仅需要比较所有这样的路径，再从中挑出最短的一条。

由归纳假设蕴含的算法如下。在每一次迭代中，增加一个新的顶点，即顶点 w，其满足

$$\min_{u \in V_k} (u.SP + length(u, w)) , \tag{7.1}$$

是所有不属于 V_k 的 w 中最小的。由以上讨论，w 确实是第$(k+1)$个靠近 v 的顶点；从而，把它加入后扩展了归纳假设。

现在算法已经完成了，但是它的效率还可以提高。算法的主要步骤涉及到寻找下一个最靠近的顶点。它可通过公式（7.1）计算最小路径长度来实现。然而，在每一步中并不需要检查全部的 $u.SP + length(u, w)$值。当增加一个新的顶点时，绝大部分值没有改变；只有那些经过新顶点的相应路径才会改变。可将到所有顶点的已知最短路径的长度保留在 V_k 中，而仅当 V_k 被扩展时才进行更新。w 增加到 V_k 中后，寻找更好最短路径的惟一方法是经过 w。所以，我们需要检查从 w 出发、指向不属于 V_k 顶点的所有边。对每一条这样的边(w, z)，检查 $w.SP + length(w, z)$的长度，并在需要时更新 $z.SP$。从而，每一次迭代都找到一个具有最小 SP 值的顶点，并且更新剩余顶点中部分 SP 的值。该算法就是熟知的 Dijkstra 算法。

实现 我们需要一种方便的方法在一个路径长度的集合中找到最小者，并且经常更新路径长度。堆是一个很好的寻找最小元素及更新元素长度的数据结构。由于需要找到具有最小路径长度的顶点，我们把所有不在 V_k 中的顶点保留在一个堆里，同时用目前已知的从 v 到达它们的最短路径长度作为关键字。初始时，除一个以外所有路径长度都是 ∞，所以堆没有特别的排序（除了 v 位于顶部外）。找到 w 很容易；可以简单地从堆的顶部把它取出。所有的边(w, u)都可以检查到，并且路径长度的更新也无困难。然而，若一条路径长度被更新，例如说 z 被更新，则 z 在堆中的位置可能改变。我们需要能够对堆做相应的修改。为了这个目的，需要知道 z 在堆中的位置。（回忆一下，堆并不是一个搜索结构；它不提供任何对元素定位的功能。）寻找 z 在堆中的位置可以通过与堆相连的另一数据结构来完成。由于所有的顶点标识事先已知，所以可以把它们放在一个数组里，并用指针指向它们在堆中的位置。从而在堆中寻找顶点就仅需要访问一下数组。由于堆中的元素是图中的顶点，空间的需求仅仅是 $O(|V|)$，这是合理的。路径长度只能是递减的。如果堆中的一个元素的路径比它的父母还小，则进行交换往上移，直至找到一个合适的位置。这正是一个正规堆的维护过程（如插入）。最短路径算法在图 7.17 中给出。

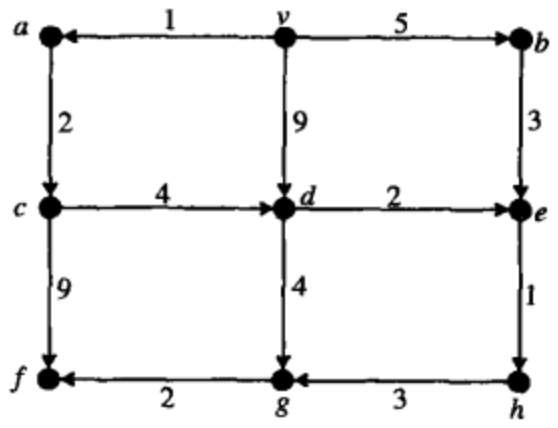
```
算法 Single_Source_Shortest_Paths (G, v)
输入：G = (V, E)（一个加权有向图）和 v（源节点）
输出：对于每一个节点 w，w.SP 的值为从 v 到 w 的最短路径的长度
  {假定所有边的长度都是非负的}
begin
    for all vertices w do
        w.mark := false ;
        w.SP := ∞ ;
    v.SP := 0 ;
    while there exists an unmarked vertex do
        let w be an unmarked vertex such that w.SP is minimal ;
        w.mark := true ;
        for all edges (w, z) such that z is unmarked do
            if w.SP + length(w, z) < z.SP then
                z.SP := w.SP + length(w, z)
end
```

图 7.17 单源最短路径算法

复杂度 更新路径长度需要 $O(\log m)$次比较，其中 m 是堆的规模。有 $|V|$ 次迭代，导致$|V|$次从堆中的删除。同时，至多有 $|E|$ 次更新（因为每条边至多有一次更新），导致在堆中有 $O(|E| \log |V|)$ 次比较。所以，运行时间是 $O((|E| + |V|) \log |V|)$。注意这个算法要比非循环图的同样算法慢，因为后者中的下一顶点是从（任意次序的）队列取出的，并且不需要更新。

□ 例 7.1

单源最短路径算法的例子在图 7.18 中给出。第 1 行仅包括从 v 出发的只含一条边的路径。在这种情况下，选择指向顶点 a 的为最短路径。第 2 行表示路径的更新，现在所有路径或者从 v 出发或者从 a 出发，现在，最短路径指向 c。在每一行选择一个新的顶点，目前已知的从 v 出发的最短路径列在每个顶点上。画了圈的距离就是已知的最短距离。 □



	v	a	b	c	d	e	f	g	h
a	0	1	5	∞	9	∞	∞	∞	∞
c	0	①	5	3	9	∞	∞	∞	∞
b	0	①	5	③	7	∞	12	∞	∞
d	0	①	⑤	③	7	8	12	∞	∞
e	0	①	⑤	③	⑦	8	12	11	∞
h	0	①	⑤	③	⑦	⑧	12	11	9
g	0	①	⑤	③	⑦	⑧	12	11	⑨
f	0	①	⑤	③	⑦	⑧	12	⑪	⑨

图 7.18 单源最短路径算法的例子

评述 这种类型的算法有时被称为**优先搜索**，因为每一个顶点被赋予了一个优先权（例如，目前所知的从源头到此的距离），并且顶点根据优先权进行遍历。当考虑一个顶点时，要检查它的所有相连边。这个检查可能会导致某些优先权的改变。产生这种改变的过程就是把优先搜索与其他搜索区分开的关键。优先搜索要比正规搜索的开销更昂贵。它对涉及到加权图的问题比较有用。

通过每次查找一条路径，可以找到从 v 到达所有其他顶点的最短路径。每一条增加的路径用一条边来标识，它从前面已知的最短路径导向一个新顶点。所有这些边在一起组成了一个以 v 为根的树（见习题 7.6）。此树被称为最短路径树，它在处理各种路径问题时具有重要的作用。

7.6 最小代价生成树

考虑一个由双向链路构成的计算机网络。在每一条链路上发一个消息都有一个为正数的开销。假定在特定链路上发消息的代价与方向性无关。我们想从任一台计算机出发，广播一个消息到所有计算机。假定广播的代价是用于发送消息的各个链路代价的总和。（代价的另一个可能定义是完成整个广播所需要的时间；见习题 7.63。）网络可用一个无向图来表示，并在边上赋予一

个为正数的代价。问题是找到一个固定的连通子图（与广播使用的链路相对应），它包含所有的顶点，并且子图中边的代价总和是最小的。不难看出，这个子图必然是一棵树。若出现任何一个闭链，则我们可以通过删除一条边来把它打断；而图仍然是连通的，但是代价要比原来的小，这是因为所有代价都是正数。这个子图被称为**最小代价生成树**（MCST），除了广播以外它还有许多其他用途。我们的目标是找到一个高效的算法来得到一棵 MCST[1]。为简单起见，假定边的代价是不同的。这个假设意味着 MCST 是惟一的（见习题 7.11），此时问题更易于讨论。若没有这个假定，那么算法和原来一样，当遇到相等代价的边时，可以从中任选其一（即可任意打开一个平局）。在这种情况下，正确性证明将更加复杂。

问题 给定一个无向连通图 $G=(V,E)$，找出 G 的具有最小代价的生成树 T。

（注意，现在把权重称为代价。）直接的归纳假设如下。

归纳假设 1：已知如何为边数小于 m 的连通图找到 MCST。

基础情况是显然的。给定一个 m 条边的 MCST 问题，应该如何把它归约到边数小于 m 的问题。我们称具有最小代价的边必然包含在 MCST 中。若不然，则把它加入 MCST 将产生闭链；从闭链中删去任一其他边，将得到一棵代价更小的树，这与 MCST 的最小性相矛盾。所以，现在已经知道了一条边属于 MCST。我们把这条边从图中删除，将归纳应用到剩下的图上，而此时的图包含较少的边。这是不是一个正确的归纳运用呢？

遗憾的是，这不是一个有效归纳，因为，在删除一条边后，我们面对的问题与原始问题并不一致。首先，对这条边的选择局限在其他边上。其次，在删除了一条边后，图有可能变为不连通的。再怎么强调都不过分，归纳假设必须精确地定义且一直保持。

解决方法是调整归纳假设。已知如何选择第一条边，但我们不能简单地把它删去并遗忘，因为选择的剩余部分也依赖于它。因此，并不是把它删除，而是将它作为被选择对象，并在算法中利用这一（它被选择）事实。算法通过每次在 MCST 中选一条边来进行。从而，归纳并不是作用在图的规模上，而是作用在给定图的已经选择的边数上。

归纳假设 2：给定一个连通图 $G=(V,E)$，已知如何找到 G 的一个 $(k<|V|-1)$ 条边的子图 T，其满足 T 是一棵树，并且是 G 的 MCST 的一个子图。

已经讨论了假设的基础情况，即选择第一条边。假定已经找到满足归纳假设的树 T，并且我们需要通过增加边数来扩展 T。如何才能找到另一条边，并保证它在 MCST 中呢？可以应用查找第一条边中使用的相同方法。已经知道 T 是 MCST 的一部分。因此，在 MCST 中至少有一条边，把 T 与不属于 T 的顶点相连。我们将试图找到这样一条边。令 E_k 是所有把 T 与不属于 T 的顶点相连的边的集合。我们称在 E_k 中具有最小代价的边属于 MCST。用 (u,w) 表示这条边（见图 7.19）。由于 MCST 是一棵生成树，它包含从 u 到 w 的惟一路径（在树中，任意两顶点之间存在一条惟一路径）。若 (u,w) 不属于 MCST，则它不包含在从 u 到 w 的路径中。但是，由于 u 属于 T 但是 w 不属于 T，则在路径中至少有一条边 (x,y)，把 T 连接到一个不属于 T 的顶点。这条边的代价要比 (u,w) 的代价高，因为 (u,w) 在这些边中具有最小的代价值。但是现在能够使用与选择第一条边同样的论点。如果把 (u,w) 增加到 MCST 中并删除边 (x,y)，那么我们将得到另一棵较小代价的生成树，从而矛盾。

---

① 假定整个图是已知的。但通信网络的完整拓扑结构和所有当前代价通常都是未知的，一般所了解的仅仅是局限于本地的信息，这时需要的是一个分布式的算法。

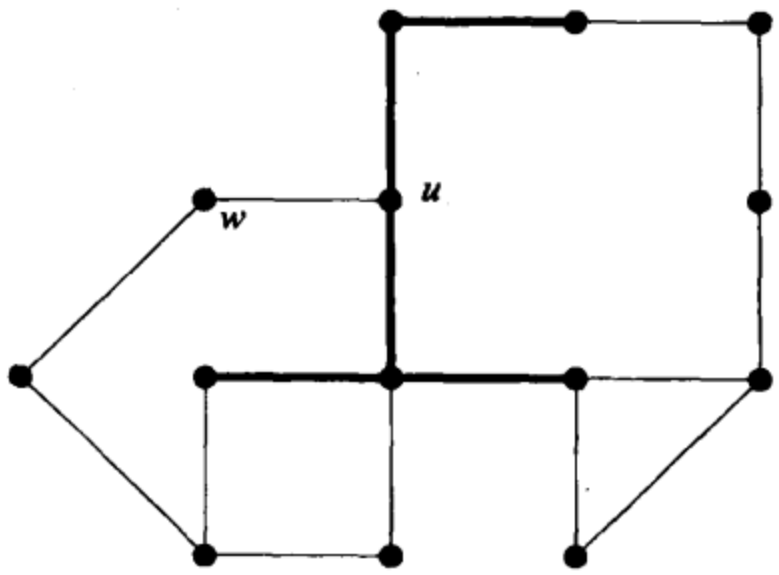


图 7.19 寻找 MCST 的下一条边

实现 这个算法非常像前一节的单源最短路径算法。被选择的第一条边是具有最小代价的边，则 T 被定义为仅有一条边的树。在每一次迭代中，我们需要找到把 T 与它之外顶点相连的最小代价边。在最短路径算法中，我们找到从 T 指向外面的最小长度路径。因此，MCST 算法与最短路径算法之间的惟一区别是，最小值不是取路径的长度而是取边的代价。算法的剩余部分实质上是一样的。对每一个不包含在 T 中的顶点 w，维护从一个 T 中的顶点指向 w 的最小代价边（如果这样的边不存在则为 ∞ ）。每一次迭代选择最小代价边并把相应的顶点 w 连接到 T。然后检查所有与 w 关联的边。如果任何一条边 (w, z)（该 z 不在 T 中）的代价小于当前指向 z 的最佳边的代价，则更新 z 的代价。算法在图 7.20 中给出。

```
算法 MCST (G)
输入： G（一个加权无向图）
输出： T（G 的最小代价生成树）
begin
    Initially T is the empty set ;
    for all vertices w do
        w.Mark := false ; {如果 w 在 T 中，则 w.Mark 为真}
        w.Cost := ∞ ;
    let (x, y) be a minimum cost edge in G ;
    x.Mark := true ; {y 将在主循环中被标记}
    for all edges (x, z) do
        z.Edge := (x, z) ; {从 T 到 z 的具有最小代价的边}
        z.Cost := cost(x, z) ; {z.Edge 的代价}
    while there exists an unmarked vertex do
        let w be an unmarked vertex such that w.Cost is minimal ;
        if w.Cost = ∞ then
            print "G is not connected" ;
            halt
        else
            w.Mark := true ;
            add w.Edge to T ;
            {现在更新与 w 相连接的未被标记的节点的代价}
            for all edges (w, z) do
```

图 7.20 MCST 算法

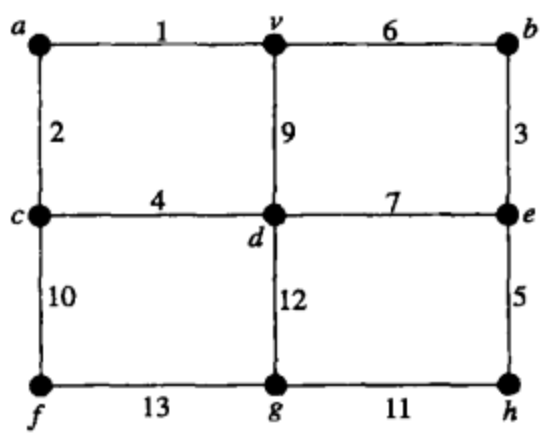
```
        if not z.Mark then
            if cost(w, z) < z.Cost then
                z.Edge := (w, z) ;
                z.Cost := cost(w, z)
    end
```

图 7.20 MCST 算法（续）

复杂度 该算法的复杂度等同于前一节的单源最短路径算法。它的最坏情况运行时间是 $O((|V|+|E|)\log|V|)$。

□ **例 7.2**

图 7.21 说明了一个 MCST 算法的例子。表格中第一列的顶点是在该步骤加入的顶点。第一个顶点是 v，与 v 相连的边以及代价均被列出。在每一行，选择具有最小代价边的顶点。当前指向非标记过顶点的最佳边（以及它们的代价）在每一步中都被更新（只有这些边的尾部被列出）。 □



	v	a	b	c	d	e	f	g	h
v	-	v(1)	v(6)	∞	v(9)	∞	∞	∞	∞
a	-	-	v(6)	a(2)	v(9)	∞	∞	∞	∞
c	-	-	v(6)	-	c(4)	∞	c(10)	∞	∞
d	-	-	v(6)	-	-	d(7)	c(10)	d(12)	∞
b	-	-	-	-	-	b(3)	c(10)	d(12)	∞
e	-	-	-	-	-	-	c(10)	d(12)	e(5)
h	-	-	-	-	-	-	c(10)	h(11)	-
f	-	-	-	-	-	-	-	h(11)	-
g	-	-	-	-	-	-	-	-	-

图 7.21 最小代价生成树算法的例子

评述 寻找 MCST 的算法，是**贪婪法**的一个例子，尽管不是很纯粹。假设正在处理一个元素集合，每一个被赋予一个代价值，我们感兴趣的是要找到一个满足某种约束，并具有最大（或最小）代价的元素集合。在 MCST 问题中，这些元素就是图中的边，而约束就是这些边对应的生成树。贪婪法在每一步都取代价尽可能是最大的元素。在 MCST 算法中，在边的选择上引入了更多约束，特别是我们仅考虑那些与当前树相连的边。所以，MCST 算法不是纯粹的贪婪算法。然而，通过在每一步选择图中某处的最小代价边，也可以找到 MCST，假设这个边不构成闭链（见习题 7.59）。贪婪法并不总能导致一个最优解，通常它只是找到最优解的启发。然而有时候，例如在 MCST 例子中，贪婪法也能导致最优解。

7.7 全部最短路径

现在考虑在图中计算所有顶点对之间的最短路径问题。

> **问题** 给定一个（有向或无向）加权图 $G=(V,E)$，其权值为非负数，要找到所有顶点对之间的最小长度路径。

由于讨论的是最短路径，则再一次把权值定义为长度，这个问题被称为**全部最短路径问题**。为简单起见，这里仅讨论如何找到最短路径的长度，而不是路径本身。假定图是有向的；同样的论点对于无向图也成立。假定本节中所有的权值都是非负的；习题 7.73 要处理的是负数的长度。

如常从直接的归纳开始。这里既可以对边使用归纳，也可以对顶点使用归纳。在把一条新的边(u, w)增加到最短路径中时，会涉及哪些问题呢？首先，这条边必须组成 u 和 w 之间的最短路径。再者，可能有其他的最短路径使用(u,w)。在最坏情况下，需要检查每一对顶点 v_1 和 v_2，看从 v_1 到 u 的最短路径长度加上(u, w)的长度再加上从 w 到 v_2 的最短路径长度，是否要小于已知的从 v_1 到 v_2 的长度。总之，对每一条边，必须做 $O(|V|^2)$次检查，这导致了最坏运行时间 $O(|E||V|^2)$。（由于边的数目可能为 $O(|V|^2)$，所以这是一个 $O(|V|^4)$的算法。）

按照最短路径要求，增加一个新的顶点 u 到图中，将涉及到哪些方面呢？我们首先需要找到从 u 到所有其他顶点的最短路径长度以及从其他顶点到 u 的最短路径长度。由于所有不包含 u 的最短路径已经知道，我们可以按照下面的方式找到从 u 到 w 的最短路径。仅需要确定在此路径中从 u 出发的第一条边。如果这条边是(u,v)，则从 u 到 w 的路径长度是(u,v)的长度加上从 v 到 w 的（已经知道）最短路径长度。所以，我们为所有与 u 相连的顶点比较这些长度，并取其中最小的长度。从 w 到 u 的最短路径也可以类似地找到。但是，这还不够。我们仍须检查每一对顶点，看在两者之间是否存在一个经过新顶点 u 的更短路径。对每一对顶点 v 和 w，检查从 v 到 u 加上从 u 到 w 的长度，并把这个长度与以前知道的最短路径相比较。每增加一个顶点，共需要 $O(|V|^2)$次比较和加法，这将导致一个 $O(|V|^3)$的算法。因此，对顶点进行归纳要好于对边进行归纳，然而，对于这个问题还有一个更好的归纳方法。

窍门是使边和顶点数固定，对允许的路径类型进行约束。这个归纳逐步删除在路径上的约束，直到最后，把所有可能的路径都考虑到。我们从 1 到$|V|$对顶点进行标记。从 u 到 w 的路径被称为 k-path，除了 u 和 w，在路径上最大的顶点被标记为 k。特别地，0-path 是一条边（因为没有其他顶点出现在这条路径上）。

归纳假设： 已知任意一对顶点之间 k-path 的最短路径长度，其中 $k<m$。

归纳的基础情况是 $m=1$，此时仅考虑一条有向边，答案是显然的。假定归纳假设对 m 成立，想把它扩展到 m+1。现在已经考虑过所有的 k-path, $k<m$。所以，只需要考虑的新路径是 m-path。我们必须找到所有顶点对之间最短的 m-path，并检查在 $k<m$ 时是否提升了 k-path。用 v_m 表示标记为 m 的顶点。任何最短 m-path 必须仅包含 v_m 一次。在 u 和 w 之间最短的 m-path 是 u 和 v_m 之间最短的 k-path（对于某些 $k<m$）再悬挂上 v_m 和 w 之间最短的 j-path（对于某些 $j<m$，其中 j 不等于 k）。由归纳，已经知道所有 $k<m$ 时最短 k-path 的长度；因此，仅需要对上面两个长度求和，即可找到 u 到 w 的最短 m-path。这个算法不仅比直接对顶点进行归纳更快（使用常数式因子），而且程序更简单。算法在图 7.22 中给出。

算法中内层的两个循环用于检查所有的顶点对。注意，因为每一次检查都是相对独立的，所以这个检查可以以任意次序用于顶点对。这种灵活性是很重要的，特别是对于算法的并行执行。

```
算法 All_Pairs_Shortest_Paths (Weight)
输入：Weight（表示一个加权图的 n × n 邻接矩阵）
{Weight[x, y]是边(x, y)的权重，如果该边不存在，则其值为∞，另外对于所有 x，
Weight[x, x]的值为 0}
输出：最终，矩阵 Weight 包含所有节点对的最短路径的值
begin
   for m := 1 to n do {归纳序列}
      for x := 1 to n do
         for y := 1 to n do
            if Weight[x, m] + Weight[m, y] < Weight[x, y] then
               Weight[x, y] := Weight[x, m] + Weight[m, y]
end
```

图 7.22　全部最短路径算法

复杂度　对每一个 m，算法仅涉及到每一对顶点间的一次加法和一次比较。归纳序列的长度是 $|V|$，所以加法（和比较）的总数至多为 $|V|^3$。回想一下单源最短路径的运行时间是 $O(|E|\log|V|)$。如果图是稠密的，即它的边数是 $\Omega(n^2)$，则使用该算法要优于为每一个顶点使用单源算法。尽管有可能在 $O(|V|^2)$ 时间内实现单源算法（见习题 7.43），并将为所有顶点对的最短路径产生一个 $O(|V|^3)$ 的算法，但是对于稠密图，本节的算法仍然较好，因为它更易于实现。换句话说，如果图是相对稀疏的，通过使用单源算法 $|V|$ 次，其运行时间是 $O(|E||V|\log|V|)$，则相对较好。

7.8　传递闭包

给定一个有向图 $G=(V, E)$，G 的传递闭包 $C=(V, F)$ 是一个有向图，若满足当且仅当在 G 中存在一条从 v 到 w 的路径，则在 C 中存在一条边 (v, w)。传递闭包是与问题相联系的，例如，对于本章开始时提到的用户账号安全问题。顶点对应于用户，而边则对应于许可。传递闭包为每个用户指明了那些得到他的账户许可的其他用户。传递闭包有许多应用，所以如何高效地找到它就非常重要。

> **问题**　给定一个有向图 $G=(V, E)$，找到它的传递闭包。

下面使用**归约**来求解这个问题。把传递闭包问题实例转化成已知如何求解的另一问题的实例。然后，把后者的解答转化成传递闭包问题的解答。这个归约所用的是全部最短路径问题。

令 $G'=(V, E')$ 是一个完全有向图（即所有顶点之间都有双向连接）。在 E' 中每一条边 e 被赋予长度值 0（如果 $e\in E$），否则赋予长度值 1。现在，为图 G' 求解全部最短路径问题。如果在 G 中从 v 到 w 存在一条路径，则在 G' 中它的长度值为 0，因为所有 G 中的边在 G' 中的长度为 0。所以，在 v 和 w 之间存在一条路径，当且仅当 v 和 w 的最短路径长度在 G' 中为 0。因而，一个全部最短路径问题的解答可以直接转化成传递闭包问题的解答。

关于在两个问题之间使用归约的思想，将在第 10 章做详细的展开说明。在此使用归约主要是为了用一个简单的例子阐明这个技术。可以很容易地把全部最短路径算法直接转化成传递闭包算法，如图 7.23 所示。

把一个问题归约到另一个问题的事实意味着第一个问题的解通常可以嵌入到另一个问题的解中。但是，越通用的解往往代价越高。我们已经见过一个更通用的问题也更易于求解的实例；

然而在许多情况下，你想得到越多则付出的也越多。当使用归约时，我们总是试图通过问题的特征来改进结果。

```
算法 Transitive_Closure (A)
输入： A（表示一个有向图的 n × n 邻接矩阵）
{若图包含边(x, y)，则 A[x, y]为真，否则为假，对于所有 x，A[x, x]为真}
输出：最终，矩阵 A 的值为图的传递闭包
begin
   for m := 1 to n do {归纳序列}
      for x := 1 to n do
         for y := 1 to n do
            if A [x, m ] and A [m, y ] then A [x, y ] := true
            {这一步将在后续算法中得以改进}
end
```

图 7.23 传递闭包算法

考虑该算法的主要步骤：if 语句。它包含两个检查，一个是对 $A[x, m]$，另一个是对 $A[m, y]$。仅当两个检查都满足，才采取动作。对每一对顶点，if 语句要执行 n 次。对这个语句做的任何改进都将导致算法的实质性改进。我们真的需要在所有时候都执行这两个检查吗？第一个检查仅依赖于 x 和 m，而第二个检查依赖于 m 和 y。所以，可以对一个确定的 x 和 m 仅执行第一个检查一次。如果第一个检查失败，则对 y 的任何值都无须执行第二个检查。若第一个检查成功，则也无须再次执行它。这个改变是组合在一起的，改进过的算法如图 7.24 所示。其渐近复杂度仍保持不变，但是算法执行将快两倍。

```
算法 Improved_Transitive_Closure (A)
输入： A（表示一个有向图的 n×n 邻接矩阵）
{若图包含边(x, y)，则 A[x, y]为真，否则为假，对于所有 x，A[x, x]为真}
输出：最终，矩阵 A 的值为图 G 的传递闭包
begin
   for m := 1 to n do {归纳序列}
      for x := 1 to n do
         if A [x, m ] then
            for y := 1 to n do
               if A [m, y ] then A [x, y ] := true
end
```

图 7.24 改进过的传递闭包算法

实现 这个算法的实现很直接。然而，注意最后一行与操作 *or* 作用在矩阵第 x 行有同样的效果。在第 x 行的每一个元素(x, y)被赋予本身的值或者是(m, y)的值。这些操作等价于将 x 行设置为 x 行和 m 行的 *or*。由于许多计算机在同一时间可在多个位上执行 *or* 操作，所以执行行 *or* 操作要比多次逐位执行 *or* 操作更快。因此，在实践中，这个算法的步数是 $O(n^3/w)$，其中 w 是字符的规模（即可以在一步中同时进行 *or* 操作的位数）。这是一个非常简单的并行算法的例子，这个论题也将在 9.5.3 节中讨论。

7.9 图的分解

前面我们已经见到图分解的一种形式，即将图划分成连通分支。通常，对图进行分解就是把图划分成一些子图，其中每个子图满足一个特定要求的性质。然后，当需要设计一个操作图的算法时，就可以分开考虑每个子图，并直接使用合乎需要的性质。例如，大家已经看过几个需要图为连通的算法。通过把图划分成连通分支，就可以分别把这些算法应用到每一个分支，从而避免复杂化。这一节提供另外两种图的分解，即双连通分支和强连通分支。第一个用于无向图，而第二个用于有向图。二者在设计算法时都很有用。特别是，这两个分解都依赖于图中的闭链（分别为无向和有向闭链）。所以，一旦有涉及到这种或另一种形式的闭链（有许多图问题涉及到闭链）的问题，考虑这些分解就是一个很好的想法，虽然有时并不总是有效，但至少值得考虑。本节假定图是连通的。

7.9.1 双连通分支

双向连通性概念扩展了通常形式下连通性的概念。一个无向图是**连通**的，如果从每一个顶点到另一顶点都存在一条路径。一个无向图是双连通的，如果从每一顶点到另一顶点至少存在两条顶点相互分离的路径。双连通图展示了一个更高层次的连通性：如果因为某种原因连接两个顶点的一条路径不再被使用，那么这两个顶点仍然是连通的。这可以得到一个事实，如果一个图不是双连通的，则它可被划分成子图，每一个子图是双连通的。我们主要对这种划分感兴趣。通常，一个无向图被称为 ***k* 连通**的，如果在每两个顶点之间至少存在 k 条不相交的路径（disjoint paths）。下面先来学习几个 k 连通图的性质。

k 连通图的第一个重要性质是由 Menger [1927]得到的定理，它把顶点间不相交的路径的数目与欲把图分离所需删除的顶点数联系起来。

□ **Menger 定理**

令 $G=(V, E)$是一个无向连通图，并令 u 和 v 是 G 中两个非邻接顶点。为了把 u 和 v 分离，所需从 G 中删除的最少顶点数等于从 u 到 v 之间不相交的路径的最大数。

（当一个顶点被删除时，所有与之相连的边也同时被删除。） □

由 Whitney[1932]提出的 Menger 定理的一个简单推论如下：

□ **Whitney 定理**

一个无向图是 k 连通的，当且仅当为了将图分离至少需要删除 k 个顶点。 □

由于 Whitney 定理的条件等价于定义 k 连通性的条件，所以二者可以任意使用一个。这些定理的证明，请见 Chartrand and Lesniak [1986]的例子。（定理的一个方面是显然的：如果删除了 k 个顶点后即可把图分离，则不可能有超过 k 条不相交的路径；另一个方向的推导则比较复杂。）

Menger 定理是图论中最重要的定理之一。根据目的，这两个定理的主要应用是，一个图是非双连通的当且仅当存在一个顶点，把它删除后即可分离该图。这样的一个顶点被称为**关节点**。图 7.25 显示了一个非双连通图的结构。这样的图包含一个或多个关节点。在关节点之间的块，在图中做了标注，块本身内部是双连通的。这些块组成了图的双连通分支。下面对此做更精确的定义。

定义：一个双连通分支是一个边的最大子集，其导出子图是双连通的（即不存在一个更大的把它包含在内的子图，其导出子图是双连通图）。

一个双连通分支可被定义成一个边的集合。一个顶点可以属于几个分支。的确，每个关节点都

属于多个分支（事实上，这个描述提供了关节点的另一特性。）每个图的边集合可以用惟一的方式划分出双连通分支。每一条边仅属于一个分支。以下两个声明证明了划分及其惟一性的存在。

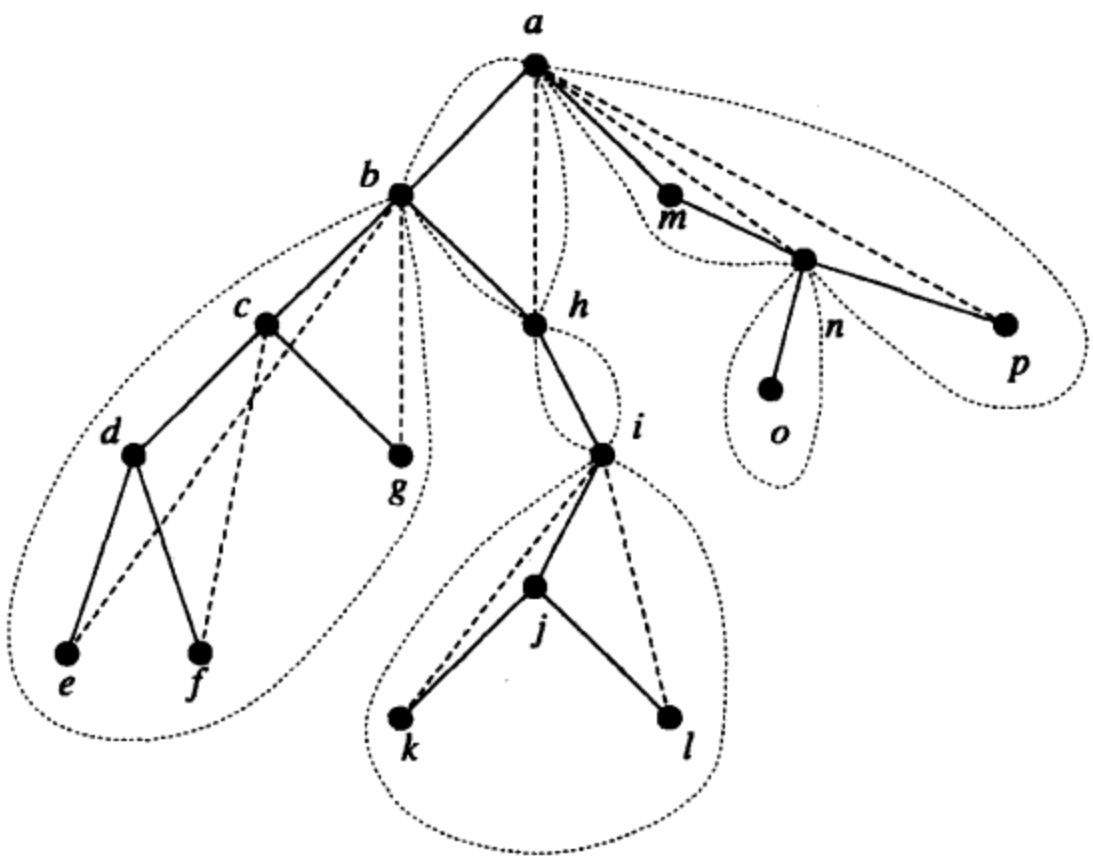


图 7.25 一个非双连通图的结构

□ **引理 7.9**

两条边 e 和 f 属于同一双连通分支，当且仅当有一条包含着两条边的闭链。（注意，一个双连通分支可能仅包含一条边；这个声明只适用于至少包含两条边的双连通分支。）

证明：首先说明在一个双连通分支中总是完整地包含一个闭链。如果一个闭链包含的边来自于多个双连通分支，则可通过增加闭链的其余部分来扩展这些分支。扩展后的子图仍然是双连通的，因为一个闭链不可能仅靠删除一个顶点就分离。这与分支的最大性相矛盾。定理的另一方面，如果两条边属于同一双连通分支，则可以得到一个按以下方式包含它们的闭链。我们增加两个新的（人为的）顶点到 e 和 f 的“中间”。（即如果 $e=(v, w)$，则增加一个新顶点 z 并用两条边(v, z)和(z, w)来取代 e；对 f 同样如此。）作为一个子图，这个分支仍然保持双连通，因为它仍未包含任何关节点。（删除任意一个新的顶点如同删除旧的边，也不能使此分支分离；删除一个旧的顶点其效果和原来一样。）所以，在这两个新顶点之间有两条顶点相互分离的路径，而这些路径正好构成了一个包含 e 和 f 的闭链。 □

□ **引理 7.10**

每一条边仅属于一个双连通分支。

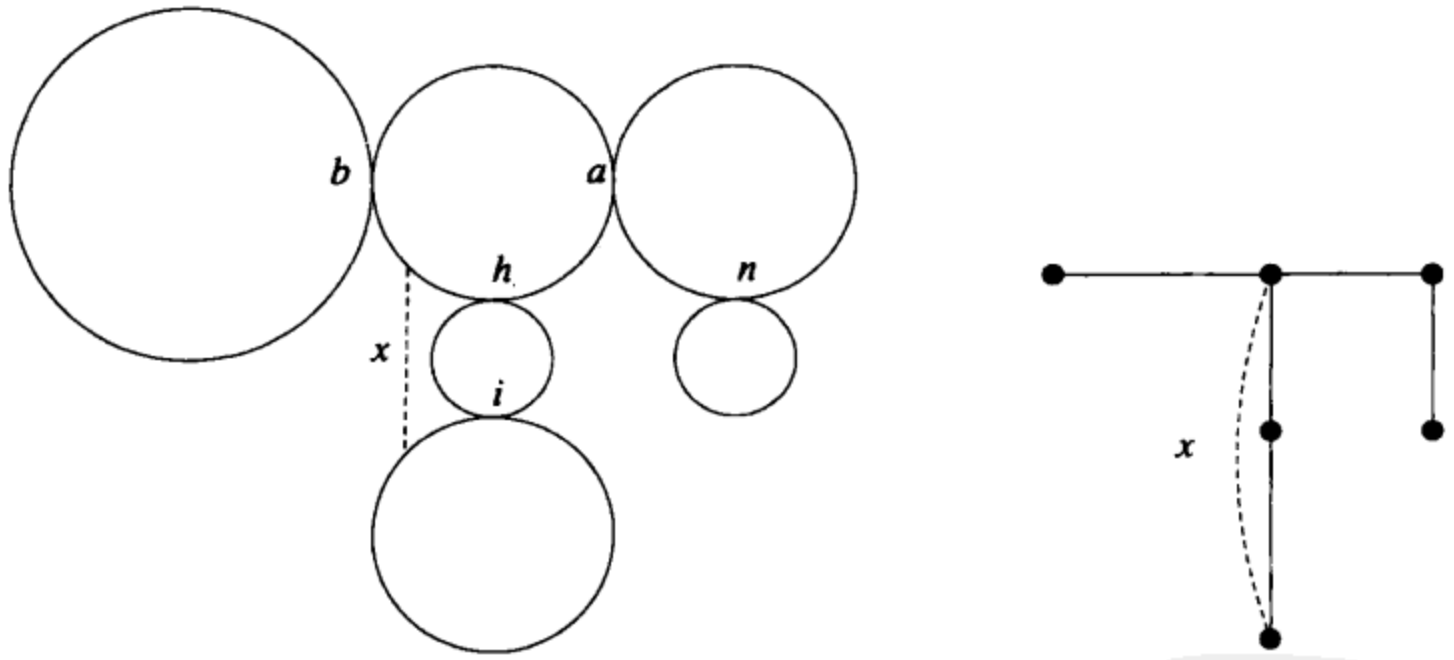
证明：每一条边必定至少属于一个双连通分支（有可能仅包含它自己）。它不可能属于多个双连通分支，因为如果这样的话，在每个分支中都有一个包含这条边在内的闭链。把这两个闭链组合起来，就是一个更大的，包含两个分支边的闭链，而我们已经知道这是不可能的。 □

我们想知道如何划分出双连通分支。与往常一样，从直接归纳假设开始。

归纳假设：已知如何找到其边数小于 m 的连通图的双连通分支。

一个具有一条边的连通图是双连通的。考虑一个有 m 条边的图，并任意挑一条边 x。我们把 x 从图中删除，由归纳假设可以找到双连通分支。现在必须确定增加 x 后对划分产生的影响。最容易的情况是，x 连接同一分支中的两个顶点（例如，图 7.25 中的边(a, n)）。在此情况下，增加 x 对划分没有任何影响（仅使得某个特定分支连通性更强了）。另一个容易的情况是，x 使得图完全分离（例如，图 7.25 中的边(h, i)和(n, o)）。在此情况下，显然 x 的两个端点都是关节点，从而 x 本身就是一个双连通分支。（这样的边被称为**桥**。）显然，其他分支都没有被改变。困难的情况是 x 不能使该图分离，并且连接来自不同分支的顶点。这种边的一个例子是图 7.25 中的边(b, e)，我们也在图 7.26（a）中对此情况做了示意。显然，x 合并了它连接的两个分支，并加上一些“处于中间”的其他分支，变成了一个更大的分支。从而问题是找到所有这些“处于中间”的分支，并有效地合并它们。

回过头再看图 7.25 和图 7.26，可以看到连通分支按以下方式定义了一棵树。将每一个双连通分支看成一个节点（称之为节点是为了与原始的顶点相区别）。我们从任一分支 R（在图 7.25 中，这个分支包含 a、b 和 h）作为树的根开始。R 的儿子是那些与 R 之间具有共同关节点的双连通分支；孙子是那些还没有包含在树中的双连通分支，它们与儿子之间具有共同的关节点，如此等等。换句话说，我们按照广度优先的方式来构造这棵树。不能因为两个双连通分支具有共同的关节点，就简单地说它们是连通的，因为这个关节点可能是多个双连通分支的关节点，我们并不想构成一个闭链。不难证明，按照这种构造总能形成一棵树（见习题 7.17）。这棵树被称为**双连通树**。图 7.26（a）显示了图 7.25 中的双连通分支，而图 7.26（b）显示了相应的双连通树。在图 7.26 中的边 x 表明增加了一条边；作为例子，它可以对应于原始图中连接 a 和 k 的边。



（a）与具有关节点的图 7.25 相对应的分支　　（b）双连通分支树

图 7.26　连接两个不同双连通分支的边

如果现在考察双连通树，那么可以看到一条连接来自不同分支顶点的边在树中产生了一个闭链。在闭链中所有的节点（对应于分支）必然合并成一个大的分支。所以现在我们得到一个算法。在归纳假设中增加一个假设，即已知如何构造这棵树，然后就可以处理前面讨论过的三种情况。由于有了更好的算法，所以在此忽略这些细节。

伴随前面描述的算法的问题，是需要多少时间来找到这个在增加了一条边到双连通树中之后产生的闭链。在一棵树中找到一个闭链可能需要遍历整棵树，在最坏情况下需要查找树中所有的

边。在树中可能的边数为 $O(|V|)$，我们不得不为原始图中的每条边都执行这个步骤。从而，算法可能需要 $O(|V|\cdot|E|)$的时间（这不是一个精确的分析）。我们将避免在每一步都搜索闭链。

改进直接归纳算法的一个常见方法是仔细选择归纳的次序。在前面的讨论中，我们是任意选择一条边。如果按照易于处理双连通树的次序来选择边，则有可能改进算法。一个最初的自然尝试是使用一个较好的图的遍历。在此我们将看到，对于此目的 DFS 的表现是非常优秀的。再次考虑图 7.25。假定 DFS 从顶点 a 开始，考虑关节点 b。令 B 是 DFS 访问过 b 后，在 b "下面" 的分支。（在图 7.25 中，这个分支包含连接顶点 b、c、d、e、f 和 g 的边。）如何才能确定 b 就是一个关节点呢？由定义，如果所有从 B 出发到达图其他部位的路径都经过 b，则 b 是一个关节点。所以，我们想判定是否所有从 B 发出的边都到达图的其余部分。

假定 B 中的顶点接下来通过 DFS 被访问。如果没有从 B 发出的边，则遍历将局限于 B 内部。B 的所有边都将被遍历，并且再次到达 b。进一步，由于 DFS 消除了交叉边，把 B 连接到图其他部位的边只能是后退边。换句话说，b 把 B 分离开，当且仅当没有从 B 发出的到达 b 上方的后退边。（这个规则的惟一例外是在 DFS 树的根部；后面将会讨论这种情况。）现在让我们看看如何判定这些事实。

我们想知道，在 DFS 树中从一棵子树出发能到达的高度。用 DFS 遍历这个图。在每一个顶点 v，先访问 v 下面的整棵子树，然后处理另一顶点，如此等等。令 T_1 是一棵以 v 的一个儿子为根的子树，进行 DFS 访问时先访问这个儿子。设想我们不仅找到了 T_1 中的所有双连通分支，也找到了在树中通过后退边连接到 T_1 的最高顶点。（这正是对归纳假设的增强，一会儿我们就会看到。）用 $High(v)$表示在 DFS 树中由一条后退边连接的最高顶点，它或者连接到 v 或者连接到 v（在 DFS 树中）的后代。假定在 DFS 树中 v 的儿子是 $w_1, w_2, \cdots, w_k$（见图 7.27）。如果知道所有 w_i 的 $High(w_i)$，则可以很容易计算出 $High(v)$：很简单，即所有 $High(w_i)$以及所有从 v 发出的后退边中的最大值。（我们将简短描述如何有效判定一个顶点比另一个更高。）所以，如果执行 DFS，则可以很容易计算出全部的 $High$ 值。例如，在图 7.27 中，$High(w_1)=r$、$High(w_2)=v$，以及 $High(w_3)=w_3$；从 v 出发最高的后退边是指向 q 的，因此 $High(v)=r$。

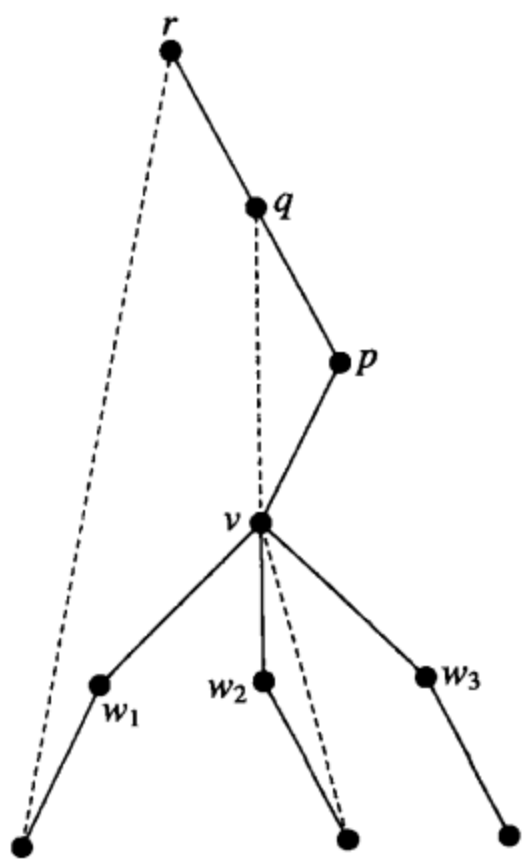


图 7.27　计算 $High$ 值

现在假设已经计算出了全部的 *High* 值。我们说一个顶点 v 是关节点，当且仅当存在一个 v 的儿子 w_i，其满足 $High(w_i)$不高于 v。的确，如果这样的 w_i 存在，则没有从以 w_i 为根的子树出发的边，其指向树中比 v 更高的顶点；因此，v 是一个关节点。（DFS 的优美性在于，它正是按照我们的目的对图进行遍历的。）

根据下面的归纳假设，通过 DFS 一步步地计算 *High* 值。

归纳假设：当用 DFS 方式访问第 k 个顶点时，我们知道如何找到那些已经访问过的，以及位于该顶点之下的顶点的 *High* 值。

归纳的次序遵照 DFS 的次序。当到达一个顶点 v 时，我们对 v 的所有儿子（递归）执行 DFS，（由归纳）找到它们的 *High* 值，并根据定义计算 *High* (v)。同时，我们能够判定一个顶点是否是关节点。

DFS 树的根会呈现一种特殊的情况。显然，没有一个 *High* 值能够超过树的根。容易看到，根是一个关节点，当且仅当它在 DFS 树中有多个儿子。当然，这很容易判定。

对于计算 *High* 值，算法效率的关键是需要执行 DFS 时的所有信息。我们惟一的问题是，如何在 DFS 树中确定一个顶点比另一顶点高。可以使用 DFS 数来进行判断。计算 *High* 值所涉及到的全部顶点在树中都是祖先。所以，它们已经有了 DFS 数。再有，一个祖先位置越高，它的 DFS 数就越低！对于不属于该树的顶点，这个结论不成立；然而幸运的是，我们仅关心后退边。所以，处理 *High* 值的一个实际方式就是利用 DFS 数。如同前面一样定义 *High*(v)，除了它不涉及到最高顶点本身，但仍要涉及到它的 DFS 数。根据 DFS 数来描述算法会使人迷惑，因为较高顶点对应于较低的 DFS 数。所以，我们定义递减的 DFS 数：根具有值为|V|的 DFS 数，而随着每次访问一个新顶点其数值递减。也可以使用负数：赋予根的 DFS 数为–1，而随着每次访问一个新顶点其数值递减。后一方案的优势在于，无须事先知道|V|的值。

剩下的惟一任务是找到实际的双连通分支。可以用暴力法来找到它们，但也有一个优雅的方法。回过头看 7.25。注意，当算法判定 b 是一个关节点时，B 的边是最近被遍历过的。在遍历期间，我们把新顶点放进一个栈里，并增加它遇到的边。当发现一个顶点是关节点时，可以从栈中删除全部顶部的边直至遇到该顶点。而这正是一个双连通分支！现在可以从图中删除这些边，并按同样的方式继续下去。双连通分支的完整算法在图 7.28 中给出。（算法仅用 DFS 中的 *preWORK* 和 *postWORK* 就可以定义，但基于完整性，我们把它完全表示出来。）

算法 *Biconnected_Components* (*G*, *v*, *n*)
输入： $G=(V, E)$（一个无向连通图）、v（作为 DFS 树根的节点）和 n（G 中的节点数）
输出： 对图的双连通分支进行标记同时计算 *High* 的值

```
begin
    for every vertex v of G do
        v.DFS_Number := 0 ;
    {DFS 数同时表示了相应的节点是否已被访问}
    DFS_N := n ;
    {使用渐减 DFS 数的理由将在正文中说明}
    BC(v)
end
```

图 7.28 双连通分支算法

```
procedure BC(v) ;

begin
    v.DFS_Number := DFS_N ;
    DFS_N := DFS_N - 1 ;
    insert v into Stack ;{栈初始化为空}
    v.High := v.DFS_Number ;{初始值}
    for all edges (v, w) do
        insert (v, w) into Stack ;
        {每条边将进栈两次（按边的不同方向）}
        if w is not the parent of v then
            if w.DFS_Number = 0 then
                BC( w ) ;
                if w.High ⩽ v.DFS_Number then
                {v 和 w 在剩下的图中不连通}
                    remove all edges and vertices from Stack until v is
                        reached, and mark the subgraph they form
                        as a biconnected component ;
                    insert v back into Stack ;
                    {v 是 w 分支的一部分，同时也可能是其他分支的一部分}
                v.High := max ( v.High , w.High )
            else {(v, w)是一条后退边或一条前向边}
                v.High := max ( v.High , w.DFS_Number )
end
```

图 7.28　双连通分支算法（续）

复杂度　显然，除了关于 DFS 的操作之外，每个顶点的额外工作量是常数的。因此，算法的运行时间是 $O(|V|+|E|)$。需要的空间也是 $O(|V|+|E|)$，因为这些分支必须在遍历的过程中被记住。

□ **例 7.3**

一个关于双连通分支算法的例子已经在图 7.25 中给出，这里重复一下，并用图 7.29 表示。第 1 行给出了这些顶点，第 2 行给出了它们（递减的）DFS 数。每当开始调用一个新的递归过程，逐次的每行就表示更新过的 *High* 值。当发现一个顶点是关节点时，就对它画个圈。 □

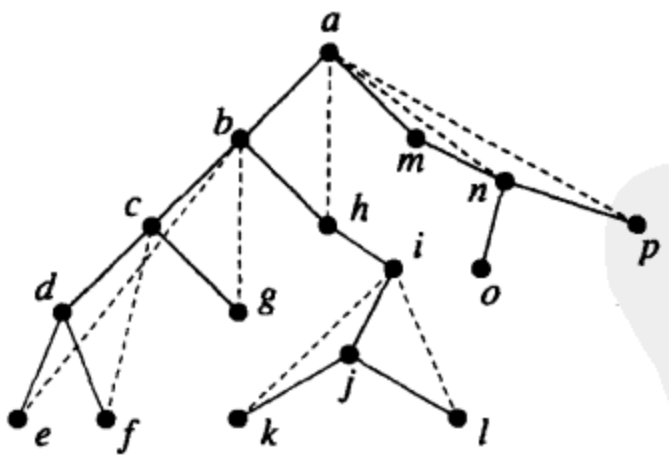


图 7.29　计算 *High* 值与双连通分支的例子

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*p*
	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
a	16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
b	16	15	-	-	-	-	-	-	-	-	-	-	-	-	-	-
c	16	15	14	-	-	-	-	-	-	-	-	-	-	-	-	-
d	16	15	14	13	-	-	-	-	-	-	-	-	-	-	-	-
e	16	15	14	13	15	-	-	-	-	-	-	-	-	-	-	-
d	16	15	14	15	15	-	-	-	-	-	-	-	-	-	-	-
f	16	15	14	15	15	14	-	-	-	-	-	-	-	-	-	-
d	16	15	14	15	15	14	-	-	-	-	-	-	-	-	-	-
c	16	15	15	15	15	14	-	-	-	-	-	-	-	-	-	-
g	16	15	15	15	15	14	15	-	-	-	-	-	-	-	-	-
c	16	15	15	15	15	14	15	-	-	-	-	-	-	-	-	-
(*b*)	16	15	15	15	14	13	15	-	-	-	-	-	-	-	-	-
h	16	15	15	15	15	14	15	16	-	-	-	-	-	-	-	-
i	16	15	15	15	15	14	15	16	8	-	-	-	-	-	-	-
j	16	15	15	15	15	14	15	16	8	7	-	-	-	-	-	-
k	16	15	15	15	15	14	15	16	8	7	8	-	-	-	-	-
j	16	15	15	15	15	14	15	16	8	8	8	-	-	-	-	-
l	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
j	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
(*i*)	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
(*h*)	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
b	16	16	15	15	15	14	15	16	8	8	8	8	-	-	-	-
a	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
m	16	15	15	15	15	14	15	16	8	8	8	8	4	-	-	-
n	16	15	15	15	15	14	15	16	8	8	8	8	4	16	-	-
o	16	15	15	15	15	14	15	16	8	8	8	8	4	16	2	-
(*n*)	16	15	15	15	15	14	15	16	8	8	8	8	4	16	2	-
p	16	15	15	15	15	14	15	16	8	8	8	8	4	16	2	16
n	16	15	15	15	15	14	15	16	8	8	8	8	4	16	2	16
m	16	15	15	15	15	14	15	16	8	8	8	8	16	16	2	16
(*a*)	16	15	15	15	15	14	15	16	8	8	8	8	16	16	2	16

图 7.29 计算 *High* 值与双连通分支的例子（续）

7.9.2 强连通分支

本节仅讨论有向图。一个有向图，如果每一对顶点 v 和 w，从 v 到 w 有一条路径且从 w 到 v 也有一条路径，则称它是强连通的，换句话说，从任一顶点开始可以到达任一其他顶点。

定义：一个强连通分支是一个最大的顶点子集，其导出子图是强连通的（即不存在另一个包含它，并能导出一个强连通图的子图）。

注意，与双连通分支不同，一个强连通分支被定义成顶点的集合。每个图的顶点都可以以惟一的方式划分为强连通分支。每个顶点仅属于一个分支。图中的一条边可以属于一个分支，或者连接两个不同的分支。下面的两个断言证明了划分的存在，它类似于前一节中双连通分支的情况。

□ **引理 7.11**

两个顶点属于同一强连通分支，当且仅当存在一个包含它们的回路。（回忆一下，

回路是一个封闭的有向路径，但并一定是简单的，即它可以包含一个顶点多次。而一个闭链则是一条简单回路。）

证明：一个回路本身就是强连通的。一个强连通分支不可能仅包含回路的一个子集，否则它就不是最大的了（可以把回路中的所有其他顶点都加入这个分支。） 现在，给定同一强连通分支中的任意两个顶点 v 和 w，下面说明它们包含在一个回路中。由强连通性的定义，可知存在一条从 v 到 w 的路径，以及一条从 w 到 v 的路径。把这两条路径组合在一起就构成一个回路（但并不一定是闭链，因为这两条路径的顶点集可能不是不相交的。） □

□ **引理 7.12**

每个顶点仅属于一个强连通分支。

证明：若一个顶点 v 属于多个强连通分支，则存在一条包含 v 以及其他分支顶点的回路。然而，把这些回路组合起来就构成另一个回路，由引理 7.11，它们只能包含在一个强连通分支中。从而矛盾。 □

可以定义类似于双连通树的被称为**强连通分支的图（SCC，或凝聚图，condensation graph。）** SCC 图的节点（称其为节点，是为了与原始顶点相区分。）对应于强连通分支；从节点 a 到节点 b 有一条有向边，当且仅当（在原始图里）从对应于 a 的分支中的任一顶点到对应于 b 的分支中的任一顶点有一条有向边。SCC 图是非循环的，因为一个循环不可能包括多个分支。图 7.30 表示了一个有向图 G 和它的 SCC 图。

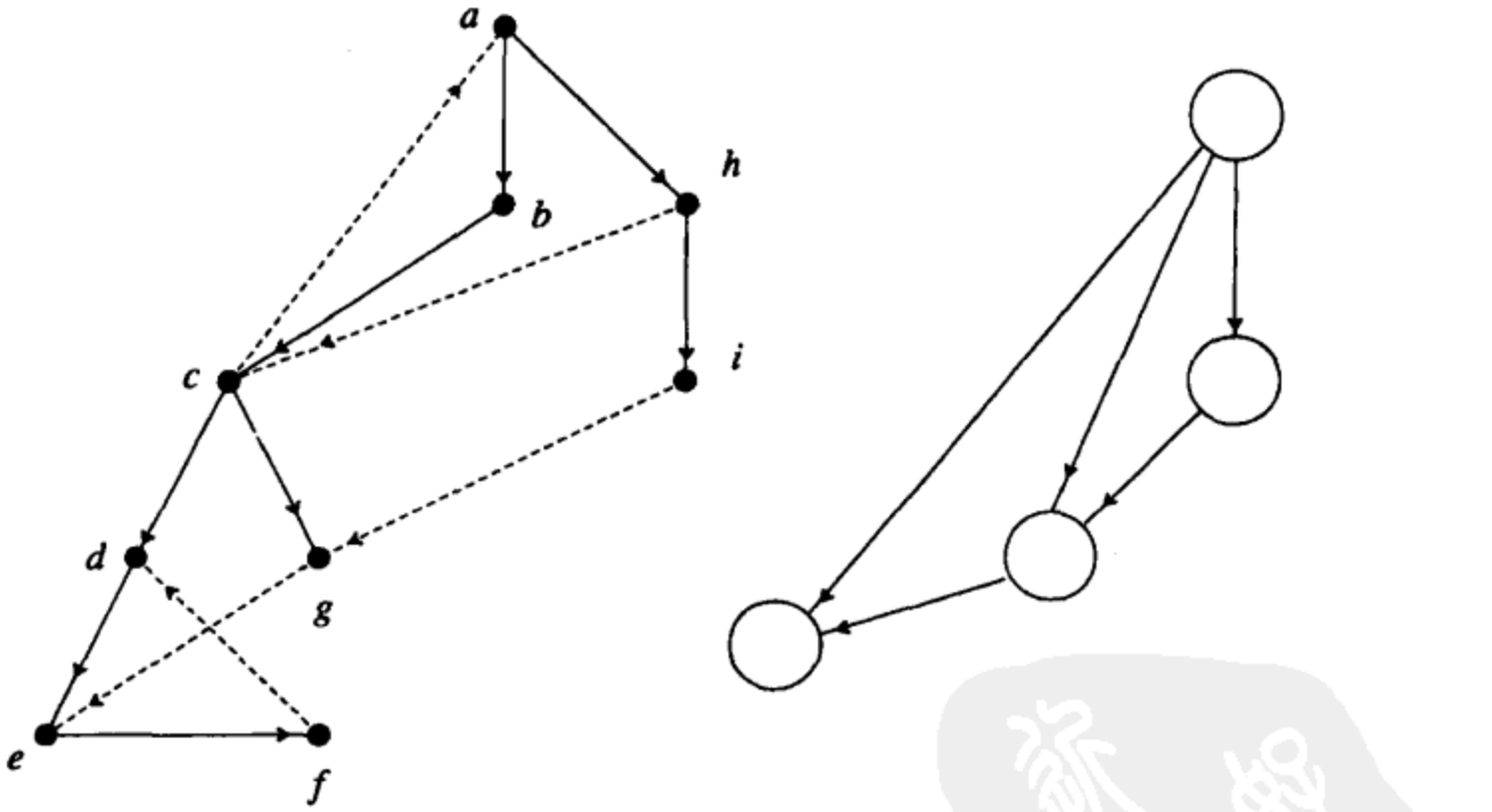


图 7.30 一个有向图和它的强连通分支图

与双连通分支一样，我们用归纳法来设计算法。

归纳假设：已知如何找到边数小于 m 的图的强连通分支，以及如何构造其 SCC 图。

基础情况是平凡的。考虑一个边数为 m 的图，并任意挑选一条边 x。我们把 x 从图中删除，通过归纳，可以找到其强连通分支。现在，必须判断增加 x 后，会对划分造成什么影响。再一次，

当 x 连接来自同一分支的两个顶点时，情况是很容易处理的。在此情况下，增加 x 对划分或者 SCC 图没有任何影响。困难的是，x 连接来自不同分支的两个顶点。这种情况在图 7.31 中进行了说明，边 x 连接图 7.30 中 SCC 图的两个分支。显然，x 把两个分支合并，当且仅当它在 SCC 图中构成一个（有向）循环。在这种情况下，对应于循环中节点的所有分支合并成一个分支，并结束。如果 x 在图 SCC 中不构成一个循环，则对分支不造成改变。如同双连通分支情况，通过以某种特殊的次序考察边，我们还可以改进算法。DFS 再次扮演了主要角色。

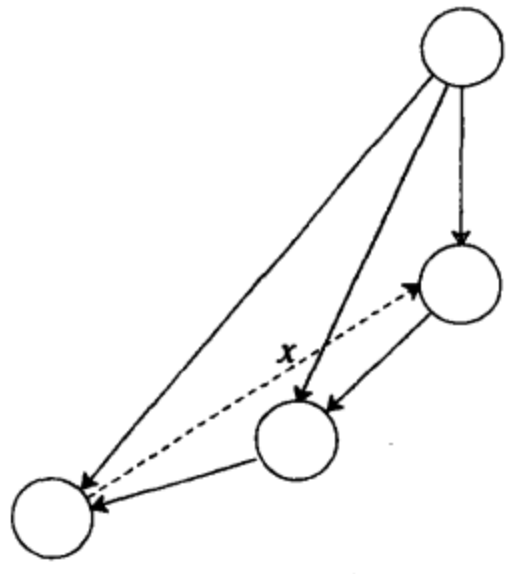


图 7.31 增加一条连接两个不同强连通分支的边

下面尝试使用与双连通分支算法相同的步骤，在需要时进行适当的修改。当使用 DFS 访问一个顶点时，我们想判定它是否与其他顶点同属于某个回路，特别是，那些在 DFS 树中比它高的顶点。*High* 值的概念在此同样被使用。我们正在寻找的是这样的顶点，从它们出发或者从它们的后代出发，不存在到达图的其他部分的路径。我们需要一个类似于关节点的，用于判定“断点”的机制。考虑 DFS 树。强连通分支占领了树中的连接部分（见习题 7.88）。即，强连通分支中的所有顶点必定属于 DFS 树中的一个连接子树。对于一个给定的分支，考察它在树中的最高顶点；我们称这个顶点为**分支的根**。这个根是分支中第一个被 DFS 访问的顶点（例如，在图 7.30 中的根是 a、d、g 和 i）。如果我们可以类似于判定关节点那样来判定这些根，则可以找到划分。下面将看到这些根的确类似于关节点。

算法基于按照 DFS 次序的归纳。令 r 是第一个完全被 DFS 访问的分支的根。通常在 DFS 的图形中，它是最左边的最低的一个分支（在图 7.30 中，$r = d$）。这个分支必然包含 r 在树中的全部后代（不会有任何一个后代可能属于某个更小的分支，因为分支遍历已经被首先完成。）如果在 DFS 期间，可以判定 r 是第一个根，则我们可以判定这个分支并把它从图中删除，然后由归纳继续进行下去。事情可能并不像说的那样简单，但这是算法的主要思想。让我们先看看如何确定 r。

首先，作为某个分支的根的顶点 r，不可能有任何后向边从 r 的后代指向一个比 r 高的顶点。这样的后向边会与这个更高的顶点构成一个循环，这意味着这个更高的顶点与 r 属于同一分支。我们可以像双连通分支那样确定是否存在这样的后向边，即利用 *High* 值。然而，在这里要小心，因为有向图的 DFS 不删除交叉边。考察图 7.32。顶点 g 没有任何后向边，但它有一条到 e 的交叉边，它包含在一个有着更高顶点 b 的循环中。从而，g 的父亲 f 不是一个分支的根，即使不存在任何从它后代出发的后向边。所以，必须同时考虑交叉边。

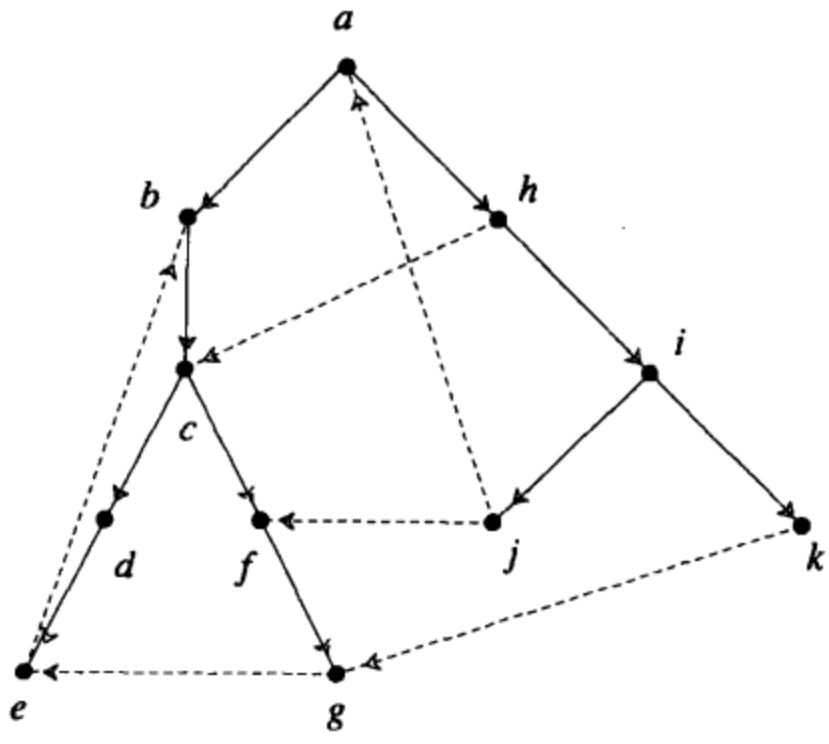


图 7.32 交叉边的影响

交叉边带来的影响是什么？交叉边必须从右到左；换句话说，它们必须指向已被访问过的顶点。回忆一下，我们正在寻找第一个根，如果从 g 到 e 有一条交叉边，并且根还没有找到，则可宣称根不可能是 f，它必定是 f 和 e 共同的祖先。若它不是 f 的祖先，则它必然在我们到达 f 之前就被发现了。特别是，仍然没有找到包含 e 的分支这一事实意味着没有从 e 指向更高顶点的路径。所以，一条从 g 出发到一个先于 f 被访问的顶点的交叉边意味着 f 不是根。但这就像考虑后向边那样容易，仅需要考虑 DFS 数即可！当考察从 g 指向 e 的边的影响时，这条边是否为后向边并不重要。e 的 DFS 数（以及它相对于 f 的值）才是重要的。通过查找一条指向具有最低 DFS 数的顶点的边，我们可以像在双连通分支中那样定义 *High* 值。顶点的 *High* 值是取它的儿子，以及它的后向边或者交叉边中的最大值。一个顶点是第一个根，如果它是第一个其 *High* 值不超过本身的顶点。注意，*High* 值不一定真正指向最高的顶点。g 的 *High* 值是 e 的 DFS 数，即使从 e 可能到达 b（从而是从 g 出发）。我们关心的是能否到达一个比 g（或 f）更高的顶点；而知道这个顶点的身份则并不重要。（当遇到一个后向边时，我们也不想追踪它的指针。）

一旦找到第一个根，就可以找到第一个强连通分支，它包含 DFS 树的根以及全部后代。然后可以从图中删除这个分支。通过删除所有分支的顶点和边，以及所有从其他顶点指向它们的边，这个工作就完成了。因为不存在其他路径可到达分支外部，我们可以忽略来自其他顶点的边。因为现在得到了一个更小的图，剩余的事情就可以通过归纳来完成（读者应该仔细核实所有的假设仍然有效。）注意，*High* 值的定义是动态的。由于删掉了指向刚被发现的分支的边，所以它们在以后的 *High* 值计算中不起作用。（这有别于双连通分支中 *High* 值的“静态”定义，它不依赖于以前的任何分支。）特别是，不需要真正删除顶点或者边，可以简单地标记每一个被发现的分支中的顶点，以后忽略那些指向已标记的顶点的边。强连通分支算法在图 7.33 中给出（为了避免混淆，这里再次使用递减的 DFS 数）。

复杂度 该算法类似于双连通分支算法，所以复杂度相同，时间和空间复杂度都是 $O(|V|+|E|)$。

算法 ***Strongly_Connected_Components (G, v, n)***

输入： $G=(V, E)$（有向图）、v（图的一个节点，作为 DFS 树的根）和 n（图 G 的节点数）

输出： 对图的强连通分支进行标记，同时计算 *High* 的值

{有向 DFS 的一般情况是，该过程可能会被调用几次直至所有节点都被访问为止}

```
begin
    for every vertex v of G do
        v.DFS_Number := 0 ;
        v.Component := 0 ;
    Current_Component := 0 ;
    DFS_N := n ;
    {使用渐减 DFS 数的理由见 7.9.1 节中的解释}
    while there exists a vertex v such that v.DFS_Number = 0 do
        SCC(v)
end

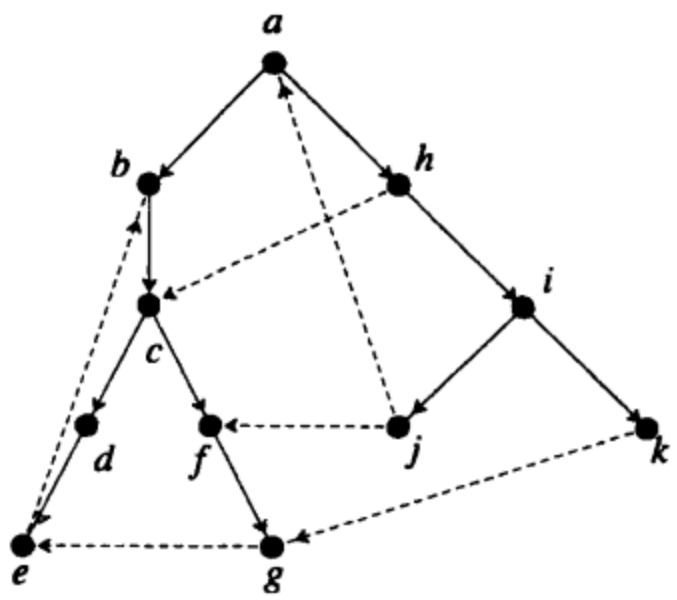
procedure SCC(v) ;

begin
    v.DFS_Number := DFS_N ;
    DFS_N := DFS_N – 1 ;
    insert v into STACK ;
    v.High := v.DFS_Number ;{初始值}
    for all edges (v, w) do
        if w.DFS_Number = 0 then
            SCC(w) ;
            v.High := max ( v.High , w.High )
        else
            if w.DFS_Number > v.DFS_Number and w.Component = 0 then
            {(v, w)是需要考虑的交叉边或后退边}
                v.High := max ( v.High , w.DFS_Number ) ;
    if v.High = v.DFS_Number then{v 是分支的根}
        Current_Component := Current_Component + 1 ;
        repeat {标记新分支的节点}
            remove x from the top of STACK ;
            x.Component := Current_Component ;
        until x = v
end
```

图 7.33 强连通分支算法

□ **例 7.4**

强连通分支算法的一个例子如图 7.32 所示，这里重复一下，在图 7.34 中重新给出。第 1 行给出了顶点，而第 2 行给出了其（递减的）DFS 数。接下来的逐行表示了，当进行新的递归过程调用时，被更新的 *High* 值。当发现一个顶点是某个强连通分支的根时，就对它画个圈。 □



	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*
	11	10	9	8	7	6	5	4	3	2	1
a	11	-	-	-	-	-	-	-	-	-	-
b	11	10	-	-	-	-	-	-	-	-	
c	11	10	9	-	-	-	-	-	-	-	-
d	11	10	9	8	-	-	-	-	-	-	-
e	11	10	9	8	10	-	-	-	-	-	-
d	11	10	9	10	10	-	-	-	-	-	-
c	11	10	10	10	10	-	-	-	-	-	-
f	11	10	10	10	10	6	-	-	-	-	-
g	11	10	10	10	10	6	7	-	-	-	-
f	11	10	10	10	10	7	7	-	-	-	-
c	11	10	10	10	10	7	7	-	-	-	-
(*b*)	11	10	10	10	10	7	7	-	-	-	-
a	11	10	10	10	10	7	7	-	-	-	-
h	11	10	10	10	10	7	7	4	-	-	-
i	11	10	10	10	10	7	7	4	3	-	-
j	11	10	10	10	10	7	7	4	3	11	-
i	11	10	10	10	10	7	7	4	11	11	-
(*k*)	11	10	10	10	10	7	7	4	11	11	1
i	11	10	10	10	10	7	7	4	11	11	1
h	11	10	10	10	10	7	7	11	11	11	1
(*a*)	11	10	10	10	10	7	7	11	11	11	1

图 7.34　计算 *High* 值与强连通分支的例子

7.9.3　利用图分解的例子

本节提供了两个例子，说明利用图的分解可显著地简化问题求解。第一个问题是关于无向图的，而第二个问题是关于有向图的。

问题　给定一个连通无向图 $G=(V, E)$，判定它是否存在一个偶数长度的闭链。

我们已经看到一个闭链必定包含在一个双连通分支中。因此，可以先把图划分成双连通分支，然后分别考察每一个分支。换句话说，现在可以假定图是双连通的！如果一个图是双连通的并且包含不止一条边，则它至少有一个闭链（事实上，每两条边就包含在一个闭链中）。让我们任意找一个闭链 $C_1 = v_1, v_2, \cdots, v_k, v_1$。若 k 是偶数，则工作就已完成。如果没有更多的边，即图正好由一个奇数的闭链构成，则答案显然为否。否则，存在一条不在闭链中的边，但其一个顶点在这个闭链中。令这条边为 (v_i, w)。由于图是双连通的，所以边 (v_i, w) 和 (v_i, v_{i+1}) 被包含在另一个闭链 C_2 中。我们从 w 开始遍历 C_2，一直到再次遇到 C_1，遇到的点被称为 v_j（见图 7.35）。显然，$v_i \neq v_j$。路径 $v_i, w, \cdots, u, v_j$ 定义了两个闭链，如图 7.35 所示。很容易看到，图中三个闭链中的一个必定是偶数。证明以下定理。

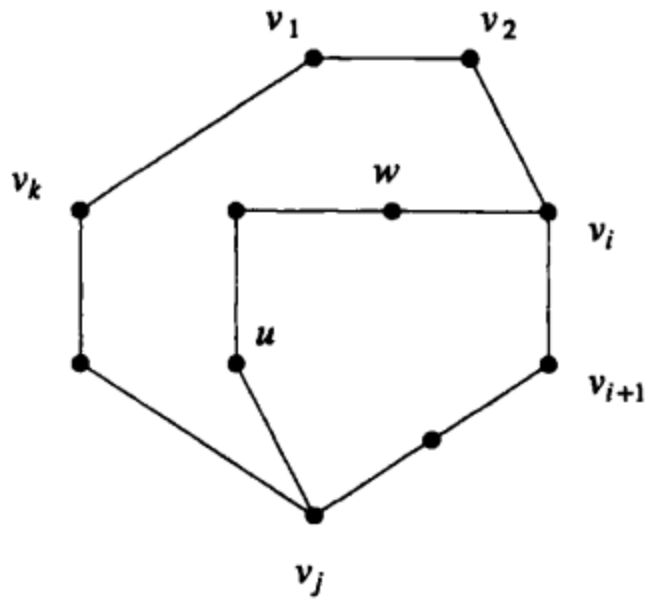


图 7.35　寻找一个偶数长度的闭链

□ 定理 7.13

边数多于 1 的，并且不只有一个奇数长度闭链的双连通图，必然有一个长度为偶数的闭链。 □

第二个问题与前面类似，但对象是有向图。

问题　给定一个有向图 $G = (V, E)$，判定是否包含一个奇数长度的（有向）闭链。

再次说明，我们已经知道一个闭链必定包含在一个强连通分支中，所以可以同样假定图是强连通的。从任一顶点 r 开始执行 DFS，并且对顶点做偶数或者奇数的标记。我们把 r 标记为偶数，然后对于每条边 (v, w)，对 v 和 w 做各自相反的标记。由于 r 可以从任意顶点到达（由强连通性假定），所以称有一个奇数长度闭链，当且仅当我们试图对某个顶点进行标记时发现它已经被做过相反的标记（最显著的例子是，如果再次到达 r 并需要对它做奇数的标记）。我们把这个事实的证明留给读者。它对强连通性假设的依赖非常强。

如果没有使用图分解，这两个问题将会非常难解。由于分解可以在线性时间内有效完成，所以通常在考虑一个给定问题时，假定问题中的图为双连通或者强连通是非常值得的。特别是当问题涉及到闭链时。有趣的是，要注明一下，如何有效判定一个有向图是否包含一个偶数长度闭链仍然是一个公开的问题（见参考文献）。

7.10 匹配

给定一个无向图 $G=(V, E)$，**匹配**是边的一个集合，其中任意两条边没有公共顶点。取这个名字的原因是，一条边可以看做是它关联的两个顶点间的一个匹配。这里要求匹配中的顶点不能属于多条边，所以它是一个一夫一妻的匹配。一个与匹配中任何边都不相关联的顶点被称为**无匹配**的，或称这个顶点不属于该匹配。**完美匹配**是指所有顶点都有匹配的匹配。**最大匹配**是指具有最大边数的匹配。**极大**匹配是指不可能再增加边数的匹配。除了世俗的夫妻匹配，很多情况也会涉及到匹配问题，如工人与工作的匹配、机器与零件的匹配，如此等等。进一步，许多看起来与匹配不相关的问题，事实上与匹配问题也是等价的。

通常，图的匹配是一个难题。本节将讨论局限在两个特殊的匹配问题。第一个问题并不是非常重要，它涉及在非常稠密图（very dense graph）中寻找完美匹配，然而解答这个问题使用的是一种非常有趣的方法，该方法可推广为偶图匹配这一重要问题的解决方法。

7.10.1 非常稠密图中的完美匹配

本例考虑一种非常特殊情况下的完美匹配问题。令 $G=(V, E)$是一个无向图，其$|V|=2n$ 并且每个顶点的度至少为 n。对这样一个图，下面给出一个查找完美匹配的算法。作为一个推论，算法说明了在这种条件下完美匹配总是存在的。

对匹配的规模 m 进行归纳。基础情况，$m=1$，任取一条边作为匹配即可。下面将说明，通过增加另一条边或者用两条新的边来替换已有边，就可以扩展任意非完美匹配。在这两种方式下，匹配的规模递增，其结果如下。

在图 G 中考虑一个有 m 条边的匹配 M，$m<n$。先检查所有不在 M 中的的边，看是否其中的一条可以加入 M 中。如果找到这样一条边，则工作完成。否则，M 就是一个极大匹配。由于 M 不是完美的，则至少有两个非连接顶点 v_1 和 v_2 不属于 M。这两个顶点至少向外伸出 $2n$ 条各不相同的边，它们指向被 M 覆盖的顶点，如若不然，则有一条边可以加入 M 中。由于 M 的边数小于 n 且从 v_1 和 v_2 出发有 $2n$ 条边与它相连，所以至少有一条从 M 发出的边，不妨称之为 (u_1, u_2)，与来自 v_1 和 v_2 的三条边相连。不失一般性，我们假定这三条边是(u_1, v_1)、(u_1, v_2)和(u_2, u_1)，见图 7.36（a）。容易看出，通过从 M 中删除边(u_1, u_2)，并增加两条边(u_1, v_2)和(u_2, v_1)，可得到一个更大的匹配，见图 7.36（b）。

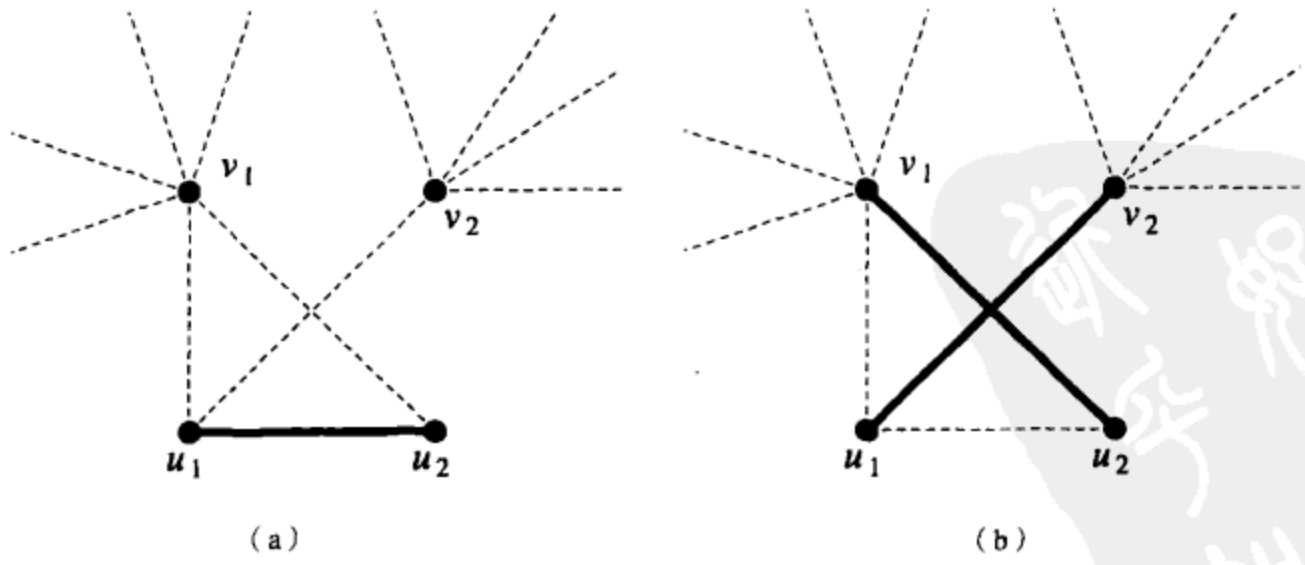


图 7.36 扩展一个匹配

算法的实现留做习题 7.21。这个算法是贪婪法的另一个例子。在把匹配扩展到一个更大匹配

的时候，每一步至多涉及到三条边，在这种情况下本方法足以胜任，但是在一般情况下，找到一个好的匹配是很困难的。一条边的选择可能会影响到图中远处另一条边的选择。接下来要说明的是如何把这个方法推广到另一个匹配问题。

7.10.2 偶图匹配

令 $G=(V, E, U)$是一个偶图，其中 V 和 U 是两个不相交的顶点集合，E 是连接 V 和 U 之间顶点的边集合。

问题 在偶图中找到一个最大匹配。

可以根据实际的匹配来表达这个问题：V 是一个女孩集合，U 是一个男孩集合，而 E 是一个“可能”配对的集合；我们想让男孩和女孩配对，并使配对数目最大。

一个直接的方法是根据某些策略来尝试配对，直到没有任何其他配对为止，我们希望这个策略能保证最优性，或者至少是接近最优。可以尝试不同的策略，例如，我们可以尝试使用贪婪法，先匹配度数小的顶点，希望后面其他的顶点要比它更容易找到未被匹配过的伴侣。（换句话说，先匹配最难找到伴侣的男孩，然后再考虑剩下的成员。）先不对这个策略做分析（这很困难），我们尝试对前一个问题使用该方法。假设从一个极大匹配开始（无须是最大匹配），我们能对它进行改进吗？考察图 7.37（a），其中的匹配用粗线表示。显然，通过用边 1A 和 2B 来替换 2A，可以改进匹配。这类似于在前一问题中的转换。但我们不局限于用两条边来替换一条边。如果找到一个相似的情况，则可用 k+1 条边来替换 k 条边，就得以改进了。例如，用边 3C、4D 和 5E 来替换边 3D 和 4E，可以进一步做出改进，见图 7.37（b）。

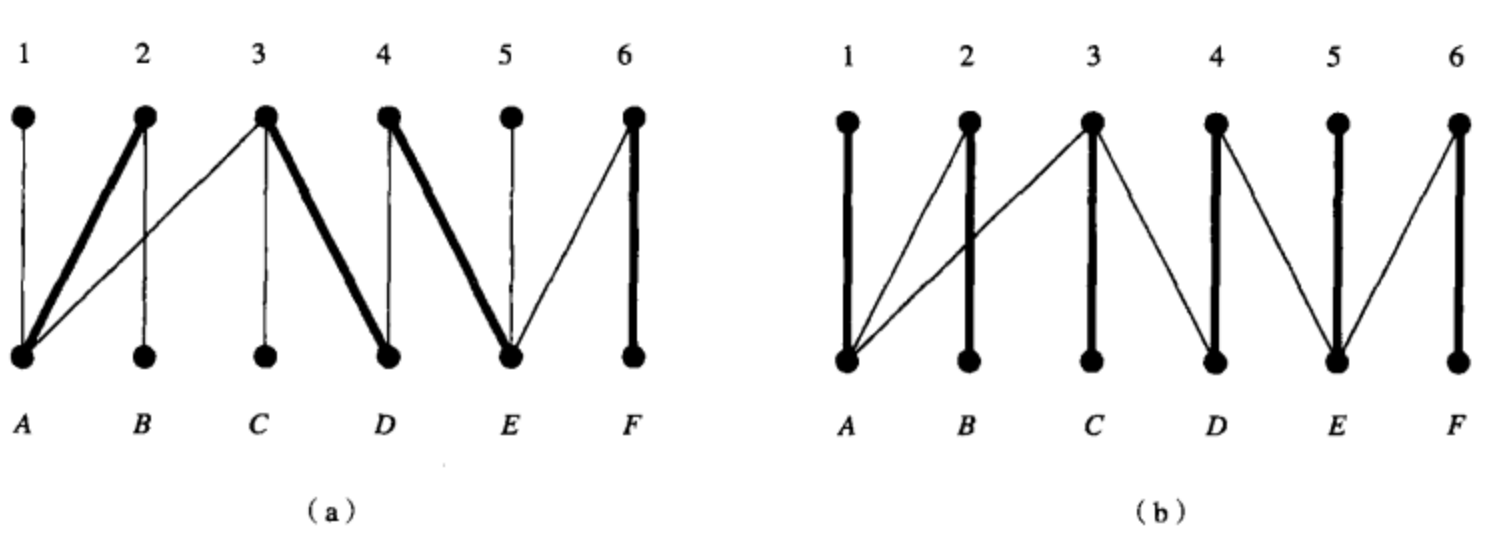


图 7.37 扩展一个偶图匹配

下面研究这些转换。目标是增加更多被匹配的顶点，我们从一个非匹配顶点 v 开始，并尝试为它找到一个配对。如果已经有了一个极大匹配，则 v 的所有邻居都被匹配了，所以必须尝试打破这个匹配。选择另一个顶点 u，它与 v 相连，但已经匹配给 w 了。我们让 v 和 u 进行匹配并打破 u 和 w 的匹配。现在必须为 w 找到一个匹配。如果 w 与某个未被匹配过的顶点相连，则工作完成（这是上面的第一种情况）；如若不然，继续按这种方式打破匹配并尝试再次匹配。把这个思想转换成算法还必须做两件事情。第一，必须能确定过程终止，第二，必须说明，若存在一个改进，则这个过程能找到它。首先，让我们明确地表达这个思想。

对于一个给定匹配，一个**交互路径** P（alternating path）是一个从 V 中顶点 v 出发到达 U 中顶点 u 的路径，而 v 和 u 都没有在 M 中被匹配，且 P 中的边是在 E–M 和 M 中交互进行的。即，P 中的第一条边(v, w)不属于 M（因为 v 不属于 M），第二条边(w, x)属于 M，依次类推，直到 P 的最

后一条边(z, u)，它不属于M。注意，这个交互路径正是我们得到的改进后的匹配。P中的边数必须是奇数，因为P开始于V并终止于U。进一步，P在E–M中的边数恰好比在M中的边数多一条。所以，如果我们用P的不在M里的边来替换M里的边，则得到另一个多一条边数的匹配。例如，在图 7.37（a）中用于改进匹配的第一条交互路径是$(1A, A2, 2B)$，用其中的边$1A$和$2B$可以替换边$A2$；第二条交互路径是$(C3, 3D, D4, 4E, E5)$，用其中的边$C3$、$D4$和$E5$可以替换边$3D$和$4E$。

现在清楚了，如果对于一个给定匹配M存在一条交互路径，则M不是最大的，反之亦然。

□ 交互路径定理

一个匹配是最大的，当且仅当它不存在交互路径。 □

这个断言将在下一节作为更一般的定理证明。

交互路径定理直接提示了一个算法，因为任何非最大匹配都有一个交互路径，而任何一个交互路径都可以扩展一个匹配。从贪婪算法开始，尽可能地把更多的边加入到匹配中，直至得到一个极大匹配为止。然后搜索一条交互路径，并相应地修改匹配，直至找不到交互路径为止，最后得到的匹配就是最大的。由于每一个交互路径按照多增加一条边来扩展一个匹配，而在任何匹配中至多有$n/2$条边（n是顶点数），所以迭代次数至多为$n/2$。剩下来惟一的问题是如何找到交互路径。这个问题的解答如下。通过设定M中边的方向是从U到V，不在M中边的方向是从V到U，可把无向图G转换成有向图G'。图 7.38（a）说明了从图 7.37（a）得到的匹配，而图 7.38（b）说明了相应的有向图G'。一个交互路径恰好对应于从V中非匹配顶点到U中非匹配顶点的有向路径。这样一条有向路径可以通过任意图搜索过程得到，如 DFS。搜索的复杂度是$O(|V|+|E|)$；因此算法的复杂度是$O(|V|(|V|+|E|))$。

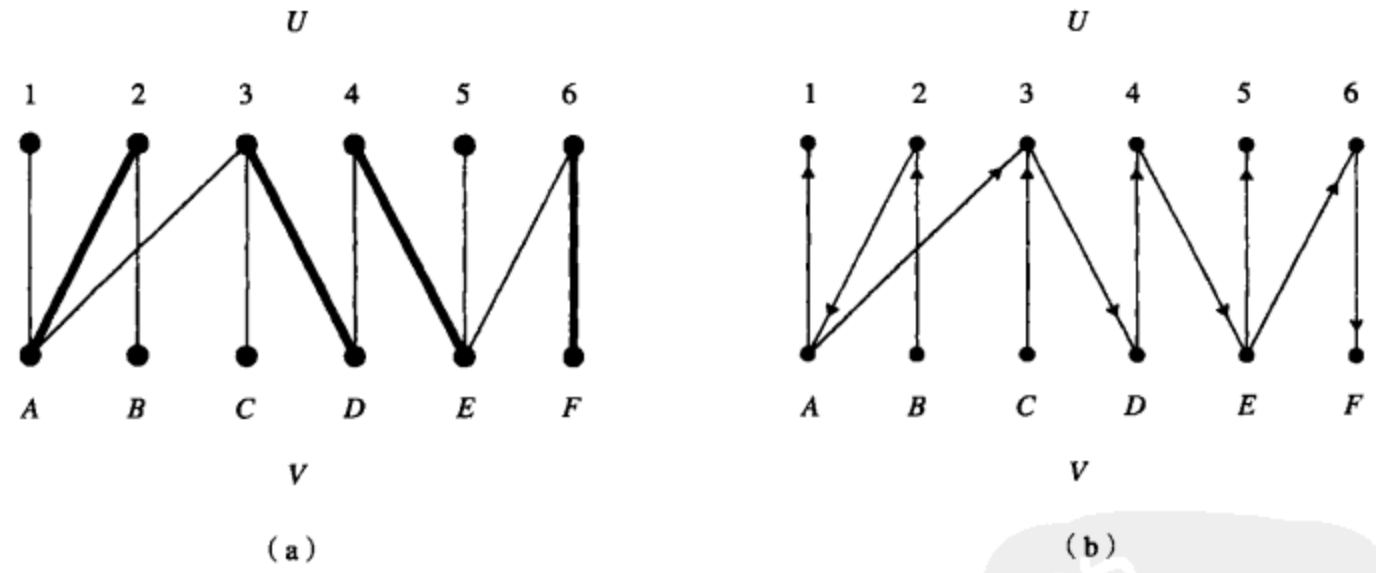


图 7.38 寻找交互路径

一个改进

由于在遍历一条路径的最坏情况下搜索可能会遍历整个图，所以可以尝试在一个搜索中寻找几个交互路径。然而，我们必须确定这些路径不会对其他路径做出改动。保证这些交互路径独立性的一个方法是，限制它们在顶点不相交。如果路径之间的顶点不相交，则它们可以同时进行，改动不同的顶点。改进过的寻找交互路径的新算法如下。首先，在G'中从V中未被匹配的顶点集开始执行 BFS，逐层进行，直至在U中找到未被匹配的顶点。然后，通过 BFS，抽取出G'中顶点不相交路径的极大集合（这就是G中的交互路径）。找到任何一个路径后，删除它的顶点，

然后找另一条路径，再删除顶点，一直下去。（得到的结果不是一个最大集合，但几乎是一个极大集合。）为了在搜索中最大化匹配的边数（每一个顶点不相交的路径为匹配增加一条边），我们选择一个极大集合。最后，利用交互路径集合修改匹配。这个过程一直重复到找不到更多的交互路径（即，新的连通图 G' 把 V 中未被匹配的顶点与 U 中未被匹配的顶点分离开了）。

复杂度 改进过的算法在最坏情况下的迭代次数是 $O(\sqrt{|V|})$。证明略，可以参见 Hopcroft and Karp[1973]。总的最坏情况下的运行时间是 $O((|V|+|E|)\sqrt{|V|})$。

7.11 网络流量

网络流量问题是图论和组合优化中的一个基本问题，它在近几十年里得到了广泛的研究，为此发展出许多算法和数据结构。它不但有许多变形和扩展，而且许多似乎与之无关的问题也可以看成是网络流量问题。网络流量问题的一个基础变形如下。令 $G=(V, E)$是一个有两个特殊顶点的有向图，一个是入度为 0 的 s（源点），一个是出度为 0 的 t（汇点）。E 中的每条边 e 被赋予一个正的权值 $c(e)$，称之为 e 的**容量**。容量用来度量经过这条边的流量。称这样的一个图为**网络**。为方便起见，可为不存在的边赋予容量值 0。**流量**是一个作用在网络的边上的函数 f，它满足以下两个条件：

1. $0 \leqslant f(e) \leqslant c(e)$：经过一条边的流量不能超过这条边的容量。
2. 对于所有的 $v \in V-\{s, t\}$，$\sum_u f(u, v)=\sum_w f(v, w)$：进入一个顶点的流量之和等于离开这个顶点的流量之和（源点和汇点除外）。

这两个条件意味着从 s 发出的全部流量等于进入 t 的全部流量。（若容量是实数，则最大流量的存在并不那么显然；后面我们会说明它们的确存在。）看待这个问题的一种方法是把网络想像成水管网络。目标是尽可能多地往管道中送水。如果太多的水被送到错误的地方，则管道就会爆裂。

首先说明一下，前一节讨论过的偶图匹配问题可以看成是网络流量问题。这看起来似乎无用，因为我们已经知道如何求解匹配问题，但还不知道如何求解网络流量问题（即归约处于一个错误的方向。）。我们在此给出错误方向的归约的原因是，求解网络流量问题的技术类似于求解偶图匹配问题的技术。理解了其中的相似性，将有助于我们理解网络流量算法。

给定一个偶图 $G=(V, E, U)$，想找到一个最大的匹配，在图中增加两个新顶点 s 和 t，让 s 与 V 中的所有顶点相连，而 U 中所有顶点与 t 相连，让 E 中所有的边从 V 指向 U（见图 7.39，其中所有的边从左指向右）。现在我们对所有的边赋予容量值 1，则在一个修改过的图 G'上得到一个有效的网络流量问题。令 M 为 G 中的一个匹配。在 M 与 G' 的流之间有着自然的对应。我们为 M 中的所有边，以及把 s 或 t 连接到 M 中的匹配顶点的边，赋予一个容量为 1 的流量。其他所有的边赋予容量值为 0 的流量。从而总流量等于匹配中的边数。这说明，M 是最大匹配当且仅当对应的流在 G' 中是最大流量。断言的一个方面是显然的：如果流量是最大的并且它对应于一个匹配，则不可能还有更大的匹配，因为这将对应一个更大的流量。断言的另一个方面，我们以某种方式对网络流量修改交互路径的思想，并且说明，如果不存在交互路径，则对应的流量是最大的。下面就来做这些事情。

对应一个给定流的 f，一个**增量路径**（augmenting path）是一个从 s 到 t 的有向路径，它由 G 的边组成，但边的方向不必相同；每条边(v, u)满足以下两个条件之一：

1. 在 G 中(v, u)是相同方向，并且 $f(v, u)<c(v, u)$。在此情况下，边(v, u)被称为**前向边**。一个前向边可以容下多个流量。$c(v, u)-f(v, u)$的差值被称为边的**松弛度**。

2. 在 G 中(v, u)是相反方向（即$(u, v)\in E$），并且$f(u, v) > 0$。在此情况下，边(v, u)被称为**后向边**。从一条后向边有可能借用（即减少）一些流量。

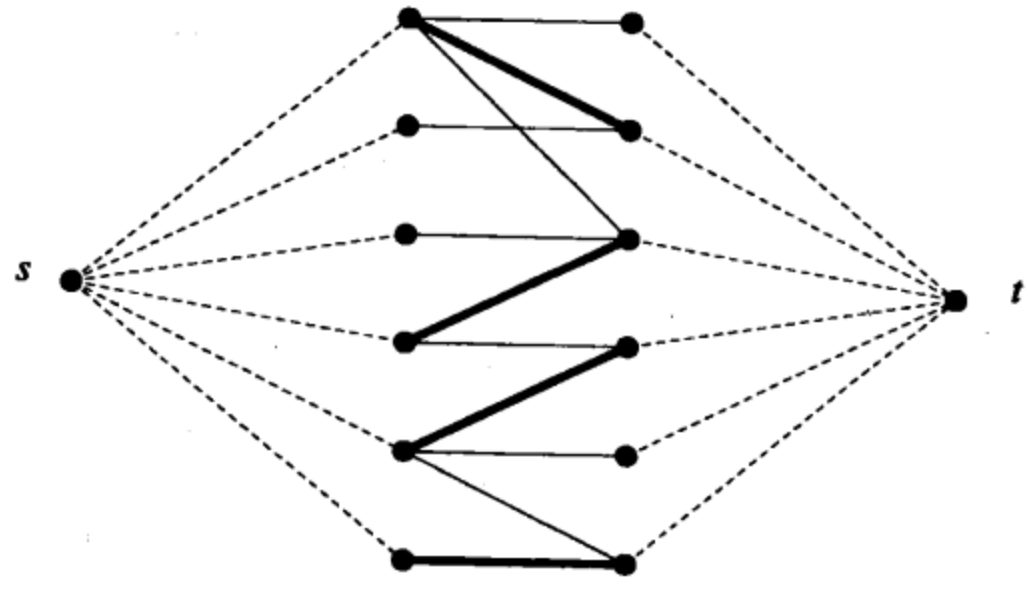


图 7.39　把偶图匹配归约到网络流量（所有边的方向都是从左到右的）

增量路径是交互路径的扩展，与偶图匹配中的交互路径一样，增量路径对网络流量有着同样的效用。如果流量 f 存在一条增量路径（我们说 f **容纳**一条增量路径），则 f 不是最大的。按照以下方式，可以在增量路径上增加流量来修改 f。如果路径中所有的边都是前向边，则可以增加多个单位的流量，而所有约束仍然满足。后向边的情况略微复杂些。考察图 7.40，每一条边被标记了两个数字 a/b，其中 a 是容量，而 b 是当前的流量值。显然，由于从 s 到 t 不存在仅包含前向边的路径，因此不能增加更多的流量。然而，还有一种方法可以扩充流。

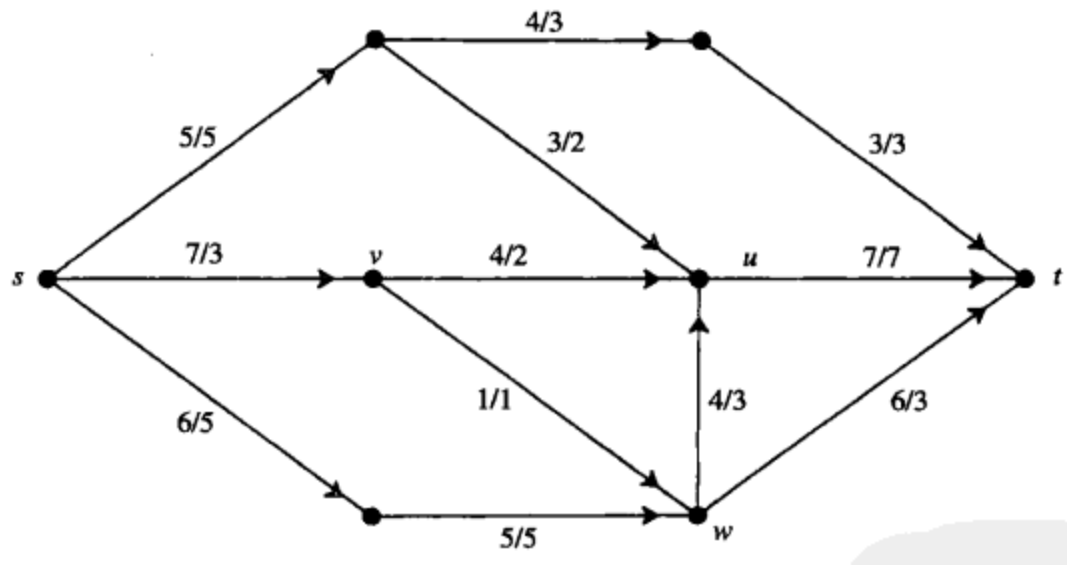


图 7.40　一个具有（非最大）流量的网络

路径 s–v–u–w–t 是一个增量路径。在这条路径上，从 s 到 u 可以增量 2（2 是一直到 u 的所有前向边中最小的松弛度），还可以从 $f(w, u)$减少值为 2 的流量。现在 u 满足守恒约束，因为 u 有一个额外的值为 2 的流量，通过增量路径进入，而从后向边中已经扣除一个值为 2 的流量。我们可以继续用同样的方式把流从 w 输入，推进到一个前向边，并从一个后向边扣除。此时，有一条到达 t 的前向边(w, t)，工作完成。由于仅仅是前向边能够从 s 发出并到达 t，所以总的流增加了。其增加值等于前向边中最小的松弛度或者流过后向边的最小流。图 7.41 显示了修改过流量的网络（该流量是最大的。）

以上分析说明，如果存在一条增量路径，则流量不是最大的，反之亦然。

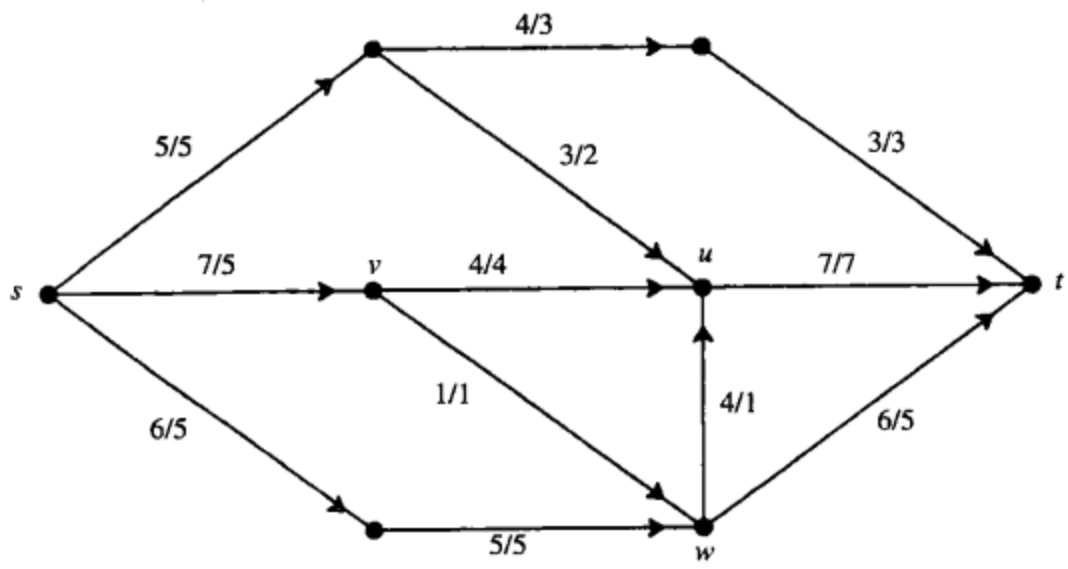


图 7.41　对图 7.40 的流进行扩充后的结果

□ **增量路径定理**

一个流量 f 是最大的，当且仅当它不包含任何增量路径。

证明：前面已经说明定理的一个方向，如果一个流包含一个增量路径，则它不是最大的。现在假定流量 f 不包含任何增量路径。先引入**割**（cut）的概念。直觉上，割是一个把 s 与 t 分开的边集合。更精确地，令 A 是 V 中顶点的一个子集，并且 $s \in A$，$t \notin A$。用 $B = V - A$ 表示剩余的顶点。割是一个边集合（$(v, w) \in E$），其中 $v \in A$，$w \in B$。割的容量定义为其各边容量的总和。显然，没有任何流可以超出任意割的容量。（如果断开管道，则没有水可以从它们流过。）因此，如果找到一个流，其值等于一个（任意）割的容量，则它必定是最大的。接着要证明，如果一个流不容许任何增量路径，则它的值等于一个割的容量，因此它是最大的。

令 f 是一个不容许任何增量路径的流。令 $A \subset V$ 是一个顶点集合，并且根据流 f，每一个顶点 $v \in A$ 都有一条从 s 到 v 的增量路径。显然，$s \in A$ 且 $t \notin A$（因为假定 f 不容许任何增量路径）。所以，A 定义了一个割。可知，对于割中所有的边 (v, w)，$f(v, w) = c(v, w)$。否则 (v, w) 将是一条前向边，并且有一个增量路径到达 w，与最初的假设 $w \notin A$ 相矛盾。根据同样的论证，不可能存在一条边 (w, v)，其中 $w \notin A$ 且 $v \in A$，并且 $f(w, v) > 0$（因为它将是一个后向边，并且能够扩展成增量路径）。因此，流 f 的值等于由 A 定义的割的容量，从而 f 是最大的。 □

这样就证明了以下基本定理。

□ **最大流最小割定理**

在网络中最大流的值，等于一个割的最小容量。 □

增量路径定理也蕴含着以下定理。

□ **整数流量定理**

如果网络中所有边的容量都是整数，则存在一个值为整数的最大流量。

证明：该定理可直接由增量路径导出。事实上，任何仅使用增量路径的算法都将导出一个整数的流量，如果所有容量都是整数的。很显然，因为我们从一个值为 0 的流开始，而每一个增量

路径都增加一个整数到整个流中。 □

现在回到偶图匹配问题。很明显，G 中任何一个交互路径都对应于 G' 中的一个增量路径，反之亦然。增量路径定理蕴含着前一节给出的交互路径定理。如果 M 是一个最大匹配，则对于它而言不存在交互路径，这也意味着在 G' 中不存在增量路径，从而流是最大的。另一方面，存在一个最大整数流量，则它显然对应于一个匹配，因为 V 中的每一个顶点仅通过一条（容量为 1 的）边与 s 连接；因此，V 中的每一个顶点仅能支持值为 1 的流量。对于 U 中的顶点，同样的论证成立。这个匹配必然是最大的，因为如果它可以扩展，则将有个更大的流量。

增量路径定理直接提示了一个算法。我们从一个值为 0 的流量开始，搜索增量路径，并对流量做相应的扩充，直到不存在增量路径。通过增加流量，不断做出改进。对增量路径的搜索可以按照以下方式完成。根据一个网络 $G=(V, E)$和流量 f，定义**余图**，它是一个网络 $R=(V, F)$，与 G 有着相同的顶点、源点和汇点，以及相同的边，但是可能有不同的方向和容量。余图中的边对应于在一个增量路径中可能的边。它们的容量对应于经过这些边可能的流增量的值。更精确地，一条边(v, w)属于 F，如果它或者是个前向边，此时它的容量是 $c(v, w)-f(v, w)$；或者是个后向边，此时它的容量是 $f(v, w)$。从而，一个增量路径是一条在余图中从 s 到 t 的正规有向路径。构造这样的余图需要 $|E|$ 个步骤，因为每条边都恰好被检查一次。

遗憾的是，按照任意方式来选择增量路径可能会导致一个非常慢的算法。这个算法在最坏情况下的运行时间可能不仅仅是关于图规模的函数。考虑在图 7.42 中的网络，其最大流量显然为 $2M$。然而一个可能开始的路径是 s–a–b–t，它只能支持值为 1 的流量。然后，有可能取增量路径 s–b–a–t，它仅对流量增加 1。这个过程会重复 $2M$ 次，而 M 有可能非常大，虽然这个图只有 4 个顶点和 5 条边。（由于 M 的值可以用 $O(\log M)$位表示，所以对于输入规模，在最坏情况下，算法是指数时间复杂度的。）

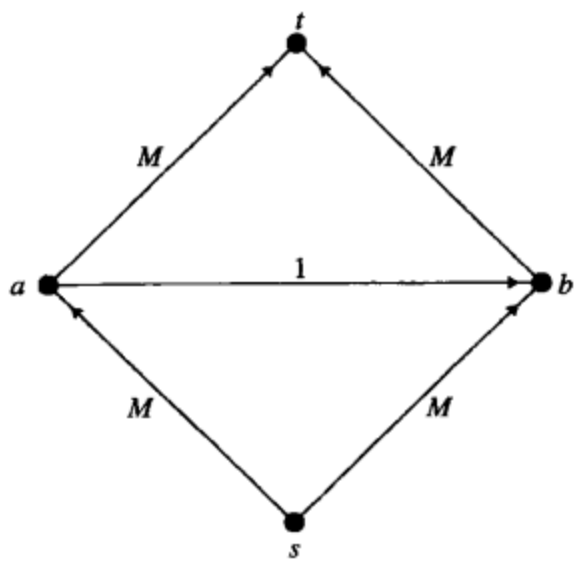


图 7.42 网络流量的糟糕例子

虽然上面的场景未必发生，但必须做出预防。进一步，为了加快算法速度，应使增量的次数最小。例如，Edmonds and Karp [1972]建议通过考虑最小边数来选取下一个增量路径，还证明了如果维持这个策略则最多需要 $(|V|^3-|V|)/4$ 次增量过程。由此可以设计一个在最坏情况下与输入规模呈线性关系的算法。自此以后，人们提出了许多不同的算法，有些非常复杂；有些则相对简单（但没有真正的简单）。由这些算法中的一些，人们得到了网络流量复杂度的上界 $O(|V|^3)$。这里不对这些算法进行描述（见参考文献）。

7.12 哈密尔顿旅行

前面讨论过对图中所有边的遍历。本章结束时，我们将讨论对图中所有顶点的遍历。这也是个著名的问题，并以爱尔兰数学家 William R. Hamilton 命名，他在 1857 年基于此问题设计了一个流行的游戏。

> **问题** 给定一个图 $G=(V,E)$，在 G 中寻找一个简单回路，其包含 V 中每个顶点仅一次。

这个回路又称为**哈密尔顿回路**。包含这样一个回路的图被称为**哈密尔顿图**。该问题分有向和无向两个版本；在此仅考虑无向的情况。

不像欧拉遍历问题那样，寻找哈密尔顿回路（或者特征化哈密尔顿图）的问题非常困难，它属于第 11 章要讨论的 NP 完全类。这里仅提供一个简单的例子，说明我们可以在非常稠密的图中找到哈密尔顿回路。这个例子最有趣的部分是，它使用了一个名为**反向归纳的**技术。

7.12.1 反向归纳

我们已经在 2.11 节中看到过反向归纳，它的思想是利用一个无限集 S（例如，$S=\{2^k\}, k=1, 2, \cdots$）作为归纳的基础。即，证明定理 $P(n)$对于所有属于 S 的 n 值成立。然后，“后退”，证明 $P(n)$有效蕴含着 $P(n-1)$有效。通常在数学里，从 n 走到 $n-1$ 不会比从 $n-1$ 走到 n 更容易，并且证明一个无限的基础情况要比简单情况难得多。另一方面，当设计算法时，从 n 走向 $n-1$ 总要容易些，因为这是求解一个输入规模更小的问题。例如，我们可以引入不影响输出的“虚假”输入。结果在许多情况下，并非设计一个针对所有规模输入的算法，而只是针对来自无限集中规模的输入，这样就足够了。这个原则最通常的使用是，仅为规模是 2 的指数的输入设计算法。这会使设计更加清楚，并清除了许多“染污的”细节。显然，这些细节最终必须被解决。但是先解决主要问题将比较便利。在本书的几个算法中（例如，8.2 节和 9.4 节），我们都假定 n 是 2 的幂。

当可能的元素数目有界时，同样的方法也是有用的。定理的基础情况可以是具有最大元素数目的情况（而非最小数目）。证明可“后退”进行。例如，假设我们想证明一个关于图的定理，并且想对边数进行归纳。可以从完全图开始，对于一个固定的顶点数，它有最大的边数。然后可证明，如果删除一条边后定理依然成立（通常的做法是增加一条边，现在与之相反）。在使用归纳时，这给了我们额外的灵活性。接下来的算法阐明了这个原理。

7.12.2 在非常稠密图中找哈密尔顿回路

令 $G=(V, E)$是一个连通无向图，并令 $d(v)$表示顶点 v 的度。下面涉及到寻找哈密尔顿回路的问题是在非常稠密的图中进行的。我们将说明，问题的条件能保证图具有哈密尔顿回路。下面引入**阐明反向归纳原理的问题**。

> **问题** 给定一个连通无向图 $G=(V, E)$，其顶点数 $n\geqslant 3$，且每一对非邻接顶点 v 和 w 满足 $d(v)+d(w)\geqslant n$，在 G 中寻找一个哈密尔顿回路。

算法基于对边数做反向归纳。基础情况是一个完全图。至少有 3 个顶点的完全图包含一个哈密尔顿回路，并且也很容易找到它（按任意次序取出全部顶点，再把它们连接成回路）。

归纳假设：*已知如何在满足给定条件且边数大于或等于 m 的图中找到一个哈密尔顿回路。*

必须说明如何在一个满足问题条件且边数为$(m-1)$的图中找到一个哈密尔顿回路。令 $G=(V, E)$

是这样的一个图。从 G 中任意一对非邻接顶点 v 和 w，并考察一个图 G'，其中除了 v 和 w 是邻接的之外，其他都与 G 相同。由归纳假设，我们已知如何在 G' 中找到哈密尔顿回路。令 $x_1, x_2, \cdots, x_n$，x_1 是 G' 中的这样的回路（见图 7.43）。如果边 (v, w) 不包含在该回路中；则同样的回路也存在于 G 中，工作完成。否则，不失一般性，假定 $v = x_1$ 且 $w = x_n$。由 G 中给定的条件，$d(v)+d(w) \geqslant n$。现在的阶段是找到一个新的哈密尔顿回路。

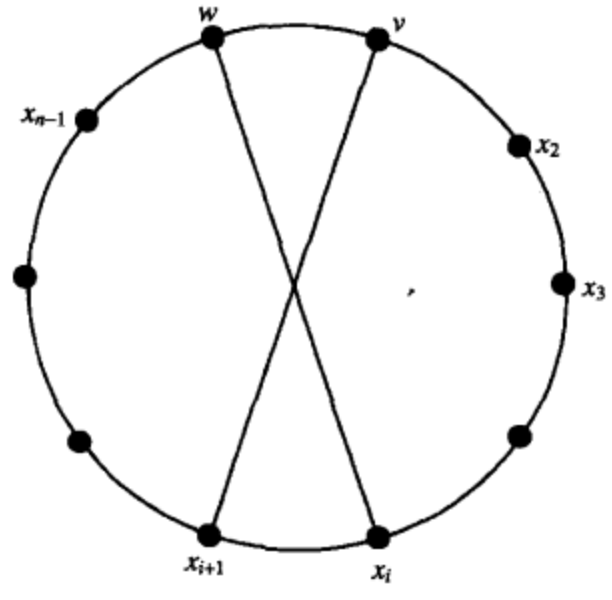


图 7.43 修改哈密尔顿回路

考虑 G 中所有从 v 和 w 发出的边。（由问题的条件）至少有 n 条边。但是 G 包含 $n-2$ 个其他顶点。所以，在回路中存在两个邻接顶点 x_i 和 x_{i+1}，其中 v 与 x_{i+1} 连接而 w 与 x_i 连接。利用边(v, x_{i+1})和(w, x_i)，可以找到一个不经过(v, w)的新的哈密尔顿回路，即回路 $v(=x_1), x_{i+1}, x_{i+2}, \cdots, w\ (=x_n), x_i, x_{i-1}, \cdots, v$（见图 7.43）。

实现 证明的直接实现可以从完全图开始，每一步取代一条边。按下面的方法，从一个较小规模的图开始，还可以做得更好。对于输入图 G，找到一个较大的路径（例如通过 DFS），并增加一些（不在 G 中的）边，从而使路径构成哈密尔顿回路。现在我们有了一个更大的图 G'，其中含有哈密尔顿回路。通常，这仅仅会增加几条边，而在最坏情况下，最多增加 $n-1$ 条边。可以迭代地应用上面的证明，从 G' 开始，直至为 G 找到一个哈密尔顿回路。取代一条边的步骤数是 $O(n)$。有 $O(n)$条边被取代；所以算法的运行时间是 $O(n^2)$。

7.13 小结

图常被用于对象之间关系的建模。由于大多数算法需要对完整的输入进行检查，所以图算法涉及到的第一个问题通常是图的遍历。本章研究了两种类型的图遍历：深度优先搜索（DFS）和广度优先搜索（BFS）。书中给出的几个例子都表明，DFS 比 BFS 更适合，所以我们建议先尝试 DFS（虽然也有许多例子是应用 BFS 更适合）。DFS 特别适合图的递归算法，而 BFS 通常需要更多的空间（但这并不是绝对的，还取决于图的结构）。我们也看到了优先搜索的例子，它常用于计算单源最短路径。优先搜索比正规搜索的开销更大，它对涉及加权图的优化问题比较有用。

回路通常会对图算法造成较大的困难。所以，关于树或者有向非循环图的算法通常比较容易设计，并且执行得也快。要意识到，即使一个边数很少的图也可能有多个不同的回路（见习题 7.54），这一点很重要。在一个图中，若算法需要检查所有的或者大部分回路，则算法会非常慢。

图分解是非常有用的，幸运的是，它的开销通常并不昂贵。书中给出的分解包括连通分支、

双连通分支和强连通分支。基础分解可使我们假定某些性质（诸如连通性）的存在，即使给定的图并没有这些性质。

另一个对图算法有用的技术是归约。由于图可以用矩阵表示，从而在图和矩阵算法之间有着自然的关系。我们将在第 10 章讨论这种关系和归约。网络流量问题和匹配问题是进行归约的好例子。归约也能帮助我们确定一个问题是否是困难的。第 11 章将讨论一类问题，名为 NP 完全问题，它也许不能用最坏情况下运行时间是输入规模多项式的算法来解决。这个类包含许多图问题。有时候，容易的问题与难解问题之间的差别是很微小的。例如，我们已经看到一个有效算法，它能判定有向图是否包含一个奇数长度的闭链；而同样的问题再加上额外的约束，即要求闭链包含指定的顶点（或者边），则此问题是 NP 完全的。理解并拓展对这种差异的直觉很重要。因此，第 11 章的材料对于理解图算法非常重要。

参考文献和补充读物

图论是数学中相对新颖的领域。绝大多数基础成果仅仅是在近几十年中才取得的。然而，图论也是一个成熟且被广泛理解的领域，已有了数千条结论。已经出版许多关于图论的书籍，如 Berge [1962]、Ore [1963]、Harary [1969]、Berge [1973]、Deo [1974]、Bondy and Murty [1976]、Chartrand [1977]、Capobianco and Molluzzo [1978]、Bollobás [1979]、Tutte [1984]，还有 Chartrand and Lesniak [1986]。也有一些书致力于图论算法，包括 Even [1979]、Golumbic [1980]（强调的是完美图和相关类别的问题），Gondran and Minoux [1984]（强调的是优化问题），还有 Gibbons [1985]、Nishizeki and Chiba [1988]（致力于平面图），以及综述文献 van Leeuwen [1986]。

欧拉图的概念来自于 Euler [1736]，它被认为是图论领域第一个有意义的结论。寻找欧拉路径的算法很容易从证明中得到（例如，参见 Even [1979]或 Ebert [1988]）。深度优先搜索最初是在 Lucas [1882]（描述工作由 Trémaux 完成）和 Tarry [1895]中描述，它用于设计一个遍历迷宫的算法。深度优先搜索的重要性由 Tarjan [1972]给出，它也提供了关于双连通分支和强连通分支的算法。

最小代价生成树问题有着广泛的研究基础。7.6 节给出的算法（虽然没有实现）源于 Prim [1957]。另一个算法（见习题 7.59）来源于 Kruskal [1956]。寻找最小生成树的其他算法有：Yao [1975]、Cheriton and Tarjan [1976]、Fredman and Tarjan [1987]，还有 Gabow, Galil, Spencer, and Tarjan [1986]。

7.5 节给出的单源最短路径算法是由 Dijkstra [1959]设计的。而基于堆的实现则来源于 Johnson [1977]（也可见 Tarjan [1983]）。当图是稀疏的时，这是实践中的常见情况，则算法很快：如果边的数目与 $|V|^2$ 成比例，则算法的运行时间是 $O(|V|^2 \log |V|)$。针对稠密图的一个更好的实现可参见习题 7.43，其运行时间是 $O(|V|^2)$。这个问题最著名的渐近运行时间（使用了非常复杂的数据结构）是 $O(|E| + |V| \log |V|)$，来自于 Fredman and Tarjan [1987]。7.7 节给出的寻找全部最短路径的算法来源于 Floyd [1962]。假设不存在权值为负数的闭链，则该算法对于可能有负数权值的加权图也是正确的（见习题 7.73）。在平均情况，更快地寻找全部最短路径是可能的，Spira [1973]提供了一个平均运行时间为 $O(|V|^2 \log^2 |V|)$的算法，而 Moffat and Takaoka [1987]把 Spira 的算法与早期 Dantzig [1960]的算法混合起来，得到了一个平均运行时间为 $O(|V|^2 \log |V|)$的算法。关于最短路径算法的更多信息，参见 Deo and Pang [1984] 的综述（它包含了 222 个参考文献）。7.8 节给出的传递闭包算法来源于 Warshall [1962]。

关于网络流量的增量路径定理和应用，是由 Ford and Fulkerson [1956]提出的。关于网络流量数据结构和组合算法的一个极佳描述，在 Tarjan [1983]中给出。最近，关于网络流量的一个新算

法由 Goldberg and Tarjan [1988]给出。关于网络流量问题的更多信息及其扩展，见 Ford and Fulkerson [1962]、Hu [1969]、Christofides [1975]、Lawler [1976]、Minieka [1978]、Papadimitrious and Steiglitz [1982]，还有 Gondran and Minoux [1984]。Lovász and Plummer [1986]一书覆盖了匹配理论的数学基础和各种匹配问题的算法。Galil [1986]给出了偶图和一般图匹配算法的综述。在 7.12.2 节中，对稠密图寻找哈密尔顿回路的算法是基于 Ore [1960]的一个定理（以及证明）。

在这里，有两个与图算法相关的重要问题没有讨论：平面图与图同构。刻画平面图的特征并把图嵌入到平面，是图论最早的问题。这个问题早期的算法由 Auslander and Parter [1961]、Lempel, Even, and Cederbaum [1966]给出。判定图是否为平面图的线性时间算法由 Hopcroft and Tarjan [1974]给出，它利用一个（基于 DFS 的）线性时间算法把图分解成 3 连通分支（Hopcroft and Tarjan [1973]）。这个算法促进了许多其他算法与数据结构的发展。关于图同构的多项式算法至今仍未找到。图同构是非常少见的，其难易状况（或者是多项式，或者是 NP 难）仍是未知的问题之一（更多内容见第 11 章）。对于这个主题的讨论，可参见 Hoffman [1982]或者 Luks [1982]。

关于 de Bruijn 序列（见习题 7.28）的讨论可以在 Even [1979]中找到。习题 7.46 来源于 Sedgewick and Vitter [1986]。习题 7.55 是受 Bollobás [1986]中一个练习的启发，而习题 7.58 是受 Lovász [1979]中一个练习的启发。Ford [1956]包含一个满足习题 7.75 需求的算法。在习题 7.81 中隐含的传递闭包算法来源于 Warren [1975]。习题 7.97 来自 Lovász and Plummer [1986]。Gabow and Tarjan [1988]为习题 7.100 的瓶颈问题提供了一个有效算法。习题 7.101 和习题 7.102 中的定理被称为 Gomory 定理。习题 7.105 来源于 Lovász [1979]。习题 7.121 与设计高效空间路由表问题有关，它已经在 Manber and McVoy [1988]中得以解决。

操练型习题

7.1 使用 DFS 求解 5.8 节中讨论过的在二叉树中寻找平衡因子的问题。你仅需要定义 *preWORK* 和 *postWORK*。

7.2 令 $G=(V, E)$是一个无向连通图，并令 T 是 G 的以 v 为根的 DFS 树。

a. 令 H 是 G 的任意一个导出子图。说明 T 与 H 的交集并不必然是 H 的生成树。

b. 令 R 是 T 的子树，并令 S 是由 R 中顶点导出的 G 的一个子图。证明，R 可能是 S 的一棵 DFS 树。

7.3 输入一个连通无向图 $G=(V, E)$，以及一棵 G 的生成树 T 和顶点 v。设计一个算法，以判定 T 是否是 G 的一棵以 v 为根的 DFS 生成树。换句话说，判定 T 是否为从 v 开始按照某种次序的 DFS 输出。算法的运行时间应该是 $O(|E|+|V|)$。

7.4 对所有包含某个顶点 v，并且其中存在以 v 为根的 DFS 生成树等同于以 v 为根的 BFS 生成树的无向图，进行特征化。（两棵生成树是等同的，如果它们包含相同的边集合；它们的遍历次序在此不重要。但是，这两棵树必须有相同的根 v。）

7.5 按下列方式修改拓扑排序算法（见图 7.14）。假定你不知道一个图是否为非循环的。显然，若图是循环的，则拓扑排序是不可能的。设计一个算法，若图是非循环的，则输出拓扑排序标记，若图是循环的，则输出一个闭链。算法的运行时间应该是 $O(|E|+|V|)$。

7.6 考虑单源最短路径算法（见图 7.17）。证明在算法执行过程中，所建立的包含所有从 v 出发的最短路径的边的子图，是一棵以 v 为根的树。

7.7　令 $G=(V, E)$是一个无向加权图，并令 T 是以顶点 v 为根的最短路径树（见习题 7.6）。假设 G 中的所有权值增加一个常数值 c，则 T 是否仍然是从 v 发出的最短路径？

7.8　证明或者给出一个反例：若不存在负数权值的闭链，则单源最短路径算法（见图 7.17）对于某些边权值为负数的加权图仍然有效。

7.9　令 $G=(V, E)$是一个无向加权图。证明，若所有的代价都各不相同，则存在惟一的最小代价生成树。

7.10　修改 MCST 算法（见图 7.20），使之可找到最大代价生成树。

7.11　证明或者给出一个反例，说明 MCST 算法（见图 7.20）对于某些边权值为负数的加权图仍然有效。

7.12　a. 给出一个加权连通无向图 $G=(V, E)$和顶点 v，使得 G 的最小代价生成树等同于以 v 为根的最短路径。

b. 给出一个加权连通无向图 $G=(V, E)$和顶点 v，使得 G 的最小代价生成树完全不同于以 v 为根的最短路径。这两棵树有可能完全不相交吗？

7.13　描述图 7.25 中删除顶点 c 后得到的双连通树和双连通分支的区别。

7.14　a. 对图 7.44 运行双连通分支算法。算法必须遵照图中给定的 DFS 数。给出算法在每一步计算中的 *High* 值。

b. 增加一条边(4, 8)到图中，讨论其对算法造成的改变。

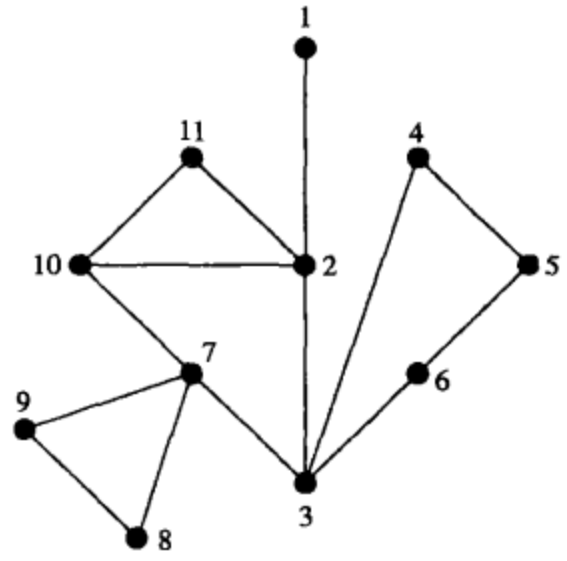


图 7.44　习题 7.14 中具有 DFS 数的无向图

7.15　证明 7.9.1 节中双连通树的定义是正确的。必须说明不存在闭链，并且所有的双连通分支集合是连通的。

7.16　a. 对图 7.45 运行强连通分支算法。算法必须遵照图中给定的 DFS 数。给出算法在每一步计算中的 *High* 值。

b. 增加一条边(4, 1)到图中，讨论其对算法造成的改变。

7.17　令 $G=(V, E)$是一个强连通图，并令 T 是 G 的一棵 DFS 树。证明，如果根据 T 把 G 中所有的前进边都删除，则剩余的图仍是强连通的。

7.18　a. 证明在有向图中寻找奇数长度闭链算法（见 7.9.4 节）的正确性。

b. 给出一个图作为例子，它不是强连通的，因而算法也无法工作。

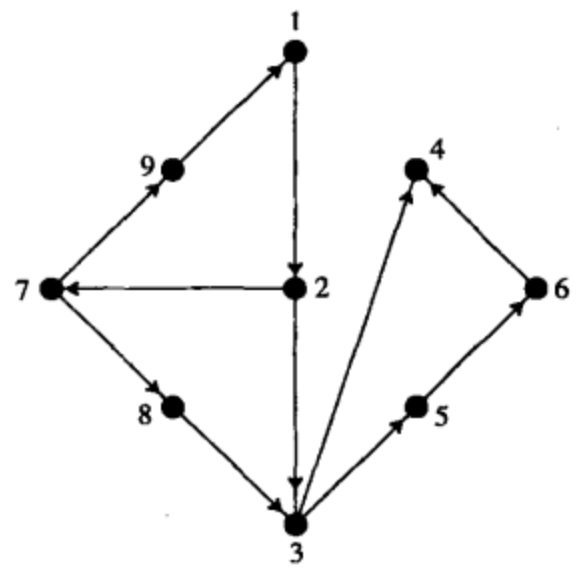


图 7.45 习题 7.16 中具有 DFS 数的有向图

7.19 为 7.10.2 节讨论的算法给出一个实现，在一个具有 $2n$ 个顶点，且每个顶点至少有 n 条边的图中寻找最佳匹配。你的算法在最坏情况下的运行时间是 $O(|V|+|E|)$。

7.20 本题是对在凝聚图中存在最佳匹配的证明进行推广。假设给你一个具有 $2n$ 个顶点的图，但并不是所有顶点都有较高的度，然而，对于任意两个不邻接的顶点，其度之和至少为 $2n$。请问是否仍然总存在最佳匹配？习题 7.19 中得到的算法是否依然有效？

创造型习题

除非特别指明，否则假定给出的图都用邻接表表示。这种表示需要 $O(|V|+|E|)$空间；因此，如果一个算法的运行时间是 $O(|V|+|E|)$，则可称其运行在线性时间内。除非特别指明，否则所有运行时间都是针对最坏情况的。在某些情况下，会给出一个特定的运行时间，并且习题要求要符合这个时间；而在另一些情况下，仅要求一个“有效的”算法。对于后者，读者应该尽量尝试找到可能的最好算法。当然在实际中，会有一些问题，我们并不知道它的最佳运行时间。

7.21 给定一个无向图 $G=(V, E)$和整数 k，寻找 G 的一个最大导出子图 H，使 H 中的每个顶点的度大于 k，或者判定这样的子图不存在。图 $G=(V, E)$的一个导出子图是子图 $H=(U, F)$，其中 $U \subseteq V$ 且 F 包含所有 U 中的顶点在 E 中所对应的边。此算法必须运行在线性时间内。这个问题已在 5.3 节讨论过。

7.22 令 $G=(V, E)$是一个连通无向图。我们想从 G 中挑出一个度为 1 的顶点，把它和与之连接的边从 G 中删除，然后继续这个过程（即，在剩余的图中选定另一个度为 1 的顶点，把它删除，如此继续下去），一直到所有的边都被删除。如果对于特定的图，这个过程总可以进行下去，则用归纳法为这些图设计一个算法会比较容易。对满足这些条件的连通无向图进行特征化。换句话说，为可以进行上述过程的图 G 找到一个充要条件。

7.23 为 7.2 节讨论的欧拉图算法描述一个有效的实现。算法必须运行在线性时间和空间内。

7.24 令 $G=(V, E)$是一个无向图，其中每一个顶点的度都为偶数。设计一个线性算法，给 G 中的边一个方向，使之满足每一个顶点的出度都等于入度。

7.25 有向欧拉回路是一条经过每条边恰好一次的回路。证明，一个有向图包含一个有向欧拉回路，当且仅当每个顶点的入度等于出度，并且其无向图是连通的。对于一个图，如果欧拉回路存在，则设计一个有效算法找出该回路。

7.26 令 $G=(V, E)$是一个无向连通图，其中有 k 个顶点的度为奇数。

a. 证明 k 是偶数。

b. 设计一个算法，找出 $k/2$ 条开放路径，满足 G 中每条边在这些路径中恰好出现一次。

7.27 设计一个算法，在一个连通无向图中找到一个顶点，并满足：删除该顶点后对此图的连通性无影响。该算法应该运行在线性时间内（不使用双连通分支算法）。作为一个结论，证明每一个连通图都包含这样一个顶点。

7.28 二进制 de Bruijn 序列是一个含有 2^n 位 $a_1a_2\cdots a_{2^n}$ 的（闭链）序列，所有规模为 n 的二进制串 s 都在该序列中的某个部位被表示，即存在一个惟一的索引 i，使 $s=a_ia_{i+1}\cdots a_{i+n-1}$（其中的索引号以 2^n 为模）。例如，序列 11010001 是一个 $n=3$ 的二进制 de Bruijn 序列。令 $G_n=(V, E)$ 是一个定义如下的有向图。顶点集合 V 对应于所有规模为$(n-1)$的二进制串组成的集合（$|V|=2^{n-1}$）。对应于串 $a_1a_2\cdots a_{n-1}$ 的顶点有一条边指向对应于串 $b_1b_2\cdots b_{n-1}$ 的顶点，当且仅当 $a_2a_3\cdots a_{n-1}=b_1b_2\cdots b_{n-2}$。证明 G_n 是一个有向欧拉图，并讨论如何实现 de Bruijn 序列。

7.29 为以下问题设计一个有效算法：给定 n 个正整数 $d_1, d_2, \cdots, d_n$，其满足 $d_1+d_2+\cdots+d_n=2n-2$，构造一棵具有 n 个顶点，其度依次为 $d_1, d_2, \cdots, d_n$ 的树。

7.30 令$(i_1, o_1), (i_2, o_2), \cdots, (i_n, o_n)$是一个整数对序列，且满足：

a. $i_1=0$，对于所有的 $1<k\leqslant n$, $i_k=1$

b. $\sum_{j=1}^{n} o_j=n-1$

找到一棵有 n 个顶点的根树，使顶点 k 的入度是 i_k，出度是 o_k。算法运行时间是 $O(n)$。

7.31 令 $G=(V, E)$是一个有向图（不要求是非循环的）。设计一个有效算法，对图中的顶点从 1 至 $|V|$ 进行标记，并且每一个顶点 v 的标记要大于至少某个 v 的前辈的标记（假设 v 有前辈），或者判定这种标记不存在。（w 是 v 的个前辈，如果$(w, v)\in E$。）

7.32 一个无向图 $G=(V, E)$被称为可进行 k 着色的，如果 G 中所有顶点均可使用 k 中不同的颜色进行着色，并且任意邻接的两个顶点没有相同的颜色。设计一个线性时间算法，用两种颜色对一个图进行着色，或者判定仅使用两种颜色无法满足要求。

7.33 令 $G=(V, E)$是一个可以用两种颜色进行着色的无向图。可能存在几种不同的方式用两种颜色对 G 进行着色。利用习题 7.32 中的算法，证明 G 的着色是惟一的（排除两种颜色相互交换，因为这总是可行的）当且仅当 G 是连通的。

7.34 令 T 是一棵根为 r 的无向树（不要求是二叉树）。T 中的每一个顶点与一个固定有限字母表中的某个字符相关联。该树用连接列表来表示。令 P 是一个模式串（用一个字符数组表示）。设计一个算法，寻找该模式是否在从根到叶子的某条路径上至少出现一次。算法在最坏情况下的运行时间必须是 $O(n+m)$，其中 n 是树中的顶点数，而 m 是模式的规模。

7.35 给定一个连通无向图 $G=(V, E)$，其仅恰好包含一个闭链，请对图中的边指定方向，使所有顶点的入度至多为 1。（这个有向图被称为是**单射**的，因为它对应于单射函数。）你的算法复杂度是多少?

7.36 令 $G=(V, E)$是一个无向图。设计一个算法，判定是否可能对 G 的边指明方向，使每个顶点的入度至少为 1。如果有此可能，算法还应说明如何实现。

7.37 给定一个连通无向图 $G=(V, E)$，对其边指定方向，使其满足下面两个条件：

a. 得到的有向图包含一棵根树（即一棵树中所有边都指向远离根的方向）。

b. 任意一条不属于上面树中的边，与树中的边都构成一个有向闭链。

你的算法的复杂度是多少?

7.38 给定一个有向非循环图 $G=(V, E)$，在 G 中找到一个简单（有向）路径，其包含的边数是此类路径中最多的。算法必须运行在线性时间内。

7.39 a. 在加权图情况下求解习题 7.38 的问题。即，要在所有路径中寻找一个权值最大的路径。

b. 你的算法对于权值为负数的边有效吗?

c. 你的算法是否适用于一般的（不要求一定是非循环）图?

7.40 令 $G=(V, E)$是一个有向非循环图，并且 k 是 G 中路径的最大边数。设计一个算法，把顶点分成至多 $k+1$ 组，使得同一组中任意两个顶点 v 和 w 之间，既无从 v 到 w 的路径也无从 w 到 v 的路径。算法必须运行在线性时间内。

7.41 令 $G=(V, E)$是一个有如下性质的有向图。G 由一个非循环子图 H 组成，H 包含 G 的所有顶点，以及额外的后退边，其中 G 的每一个简单路径至多包含一条后退边。设计一个线性时间算法，找到所有从一个固定顶点出发至 G 中其他顶点的最短路径。（注意，H 的特征未知。）

7.42 令 $G=(V, E)$是一个有向图，v 和 w 是 G 中的两个顶点。设计一个线性时间算法，找到 v 和 w 之间不同的最短路径（不要求顶点相互分离，且边上没有权值）的数目。

7.43 给出一个单源最短路径算法（见图 7.15）的实现，其在最坏情况下（对于任意 E 的规模）的运行时间为 $O(|V|^2)$。

7.44 令 $G=(V, E)$是一个加权有向图。设计一个算法在 G 中找到一个具有最小权值的闭链。算法运行时间必须是 $O(|V|^3)$。

7.45 7.5 节中描述的寻找最短路径的算法，是以任意方式打破结扣（tie）的。讨论如何修改算法，使之满足，若存在几个有同样长度的路径，则选择具有最小边数（跳数）的路径。可以使用 $O(|E|)$的额外空间。（具有相同长度或者相同边数的几个路径之间的结扣可以任意打破。）

7.46 **欧几里得图**是一个无向加权图，其中每个顶点对应于平面中的一个点，每条边的权值等于连接的两点间的距离。为了在欧几里得图中找到顶点 s 和 t 之间的最短路径，给出以下启发式建议。除了对单源最短路径使用 Dijkstra 算法，在每一次迭代中，选择之前未选过的顶点 x，使 $dist(s, x)+Euclid_dist(x, t)$的和最小，其中 $dist$ 对应于最短路径而 $Euclid_dist$ 对应于欧几里得路径（它被假定已给出）。当选择到 t 时，找到 s 到 t 的最短路径。

a. 如何实现这个算法？这里只需要关注它与 Dijkstra 算法实现的区别。

b. 解释这个算法为什么不适用于一般的（非欧几里得）图。

c. 给出一个例子，运用启发式于该例子的速度远快于（不仅仅是常数项的）Dijkstra 算法，再给出另一个速度不快的例子。根据顶点数，最坏情况下的运行时间是什么?

7.47 输入一个有向图 $G=(V, E)$，它有一个显著的顶点 v，且每个顶点 w 有一个正代价 $c(w)$。一个有向路径 $v, x_1, x_2, \cdots, x_k, u$ 的代价被定义为 $\sum_{i=1}^{k} c(x_i)$。两个端点 v 和 u 的代价被忽略掉，所以如果$(v, u) \in E$，则从 v 到 u 的代价为 0。设计一个有效算法，找到从 v 出发到其他顶点的所有最小代价路径。

7.48 令 $G=(V, E)$是一个有向加权图，且所有权值为正数。令 v 和 w 是 G 中的两个顶点，k 是一个整数，且 $k \leqslant |V|$。设计一个算法，找到从 v 到 w 且具有 k 条边的最短路径。该路径不要求是简单的。

7.49 有一大类问题，被称为**瓶颈问题**，其形式如下：输入是一个加权图，而我们对该图的某些特性感兴趣（这里指的是最短路径）。定义一个子图的**瓶颈权值**，这是子图中最大权值边的

权值，这与通常定义的权值总和不同。（这个最大权值边就是瓶颈。）在这个问题中，我们考虑瓶颈最短路径（即，路径的代价被定义为在此路径中边的最大代价）。

a. 设计一个算法求解单源最短路径问题，其中路径代价的定义如上。你能说出由该算法得到的最短路径树有何特别吗？

b. 设计一个算法，求解全部最短路径问题，其中路径代价定义如上。

7.50 令 $G=(V, E)$是一个加权非循环有向图，它可能包含负数的权值。设计一个线性时间算法，对给定的源点 v，求解单源最短路径问题。

7.51 令 $d(v)$表示顶点 v 的度。设计一个线性时间算法，对给定的有向图 $G=(V, E)$的全部邻接表按顶点的度递增进行排序。即，如果 $d(u)<d(v)$，则在邻接表中 u 的边在 v 的边的前面。结扣可以任意打破。算法可以使用线性存储空间。

7.52 寻找一个充要条件，使得一个无向图 $G=(V, E)$的边集合 E 可以划分成分离的子集 $E_1, E_2, \cdots, E_k$，且每个 E_i 对应于一个简单回路。设计一个有效算法，在满足该条件的图中找到这种划分。

7.53 给定一个无向连通图 $G=(V, E)$，试找出一个具有最小长度（无权值）的简单闭链。图中最小闭链的长度被称为图的围长。

7.54 证明，具有 n 个顶点和 $O(n)$条边的无向图包含 $2^{\Omega(n)}$个不同的闭链。（这个断言意味着，即使稀疏图也可能含有指数数目的闭链；所以，对于一般的图，检查所有闭链的算法一般是非高效的。）

7.55 设计一个算法求解以下问题：

输入：一个具有$(n+1)$个顶点和 n 条边的有向图 $G=(V, E)$，其基础无向图是一棵树，且每条边(u, w)都被标记上一个值域在$(1, 2, \cdots, n)$上的惟一整数$\lambda(u, w)$。

输出：从顶点到$\{1, 2, \cdots, n\}$的子集上的函数 S，且满足以下两个条件：

1. 如果$(u, w) \in E$，则 $S(w)=S(u) \cup \{\lambda(u, w)\}$
2. 如果 $u \neq w$，则 $S(u) \neq S(w)$

（注意，$S(u)$可以是$\{1, 2, \cdots, n\}$的任意子集，包括空集和全集。）

7.56 再次考虑习题 7.55 中的问题。证明这个问题对于包含一个闭链（不要求有向）的任何图（以及任何标记）都不可解。换句话说，证明将问题限制在树上是必需的。

7.57 有向图 $G=(V, E)$的一个**核**，是一个子集 $V' \subseteq V$，且在 V' 中任意两个顶点之间无边连接，并且对于所有顶点 $w \notin V'$，存在一条边(v, w)，其中 $v \in V'$。输入一个具有 $n+1$ 个顶点和 n 条边的有向图 $G=(V, E)$，其基础无向图是一棵树。设计一个算法在 G 中找到一个核，或者判定核不存在。

7.58 令 $G=(V, E)$是一个有向图，f 是一个定义在 G 的全体边上的函数，若 $e_1, \cdots, e_k$ 是 G 中一个闭链，则 $\sum_{i=1}^{k} f(e_k)=0$。在 G 的顶点上找一个函数 p，使得对于每一条边(v, w)，有 $f(v, w)=p(w)-p(v)$。

7.59 这是另一个 MCST 算法的概略。这里并不维护一棵树并每次扩一条边，维护的是相互分离的多棵树（它们都是 MCST 的一部分）的集合，然后再把它们组合起来，每次增加一条边。初始时，所有顶点都被看成是规模为0 的相互分离的树。在每一步中，算法寻找连接两棵分离树的最小代价边，然后加入这条边把两棵树合并起来。证明这个方法是可行且正确的。

基于这个方法描述一个算法实现。你的算法的复杂度是多少？（提示：这里使用 Union-Find 数据结构将会有帮助。）

7.60 令 $G=(V, E)$是一个无向加权图，令 F 是 G 的一个子图，且 F 是一个森林（即 F 不包含任何闭链）。设计一个有效算法，在 G 中找到一棵生成树，其包含 F 的所有边，并且在所有包含 F 的生成树中具有最小代价。

7.61 令 $G=(V, E)$是一个连通无向加权图，令 T 是 G 的最小代价生成树。假设在 G 中改变一条边 e 的代价。讨论其使得 T 不再是 MCST 的条件。设计一个有效算法，或者找到一个新的 MCST 或者判定 T 仍然是 MCST。（e 可能属于 T，也可能不属于 T。）

7.62 考虑一个通信网络，它可以建模成一个连通无向加权图 $G=(V, E)$。网络中的每个站点表示成一个顶点，通信线路是双向的，并对每条线路赋予一个代价。代价可以对应于线路预期的延迟时间，或者是使用线路的价格。每个站点具有惟一的本地信息，即它仅知道与之相连的边（和顶点）。一个网络的 MCST 可用于把消息广播到所有站点。如果仅使用 MCST 的边广播消息，则总代价最小。假定这样的 MCST 已经通过某种方法计算出来，并且每个站点都知道与之相连的边是否属于此 MCST。现在假设在特定子集 $U \subset V$ 中的站点共享它们自己知道的信息。换句话说，在 U 中的每个顶点不仅知道与之连接的边和顶点，还知道所有与 U 中其他顶点连接的边和顶点。进一步，假定局限在 U 中的 MCST 部分是连通的（也就是一棵树）。考虑一条边 $e \in U$，它属于 MCST，对其代价做了改变。

a. 寻找一个条件，使得改变 e 的代价后并不影响 MCST。仅考虑可被 U 中站点已知信息检查到的条件。换句话说，U 中的站点如何判定，在做过改变后无须修改 MCST？

b. 寻找一个条件，使得修改后的 MCST 和原来 MCST 仅区别于一条属于 U 的边（因此，这个改变可以在本地控制）。仅考虑可被 U 中站点已知信息检查到的条件。

c. 简要描述一个算法，来检查上述 a 和 b 中的条件（再次重申，仅在 U 内部），并相应地修改 MCST。算法不需要处理可能在 U 外部进行 MCST 改动的情况。

7.63 考虑网络中的广播问题，但假定，现在的主要兴趣是信息的快速传播，而非最小价格。换句话说，代价对应于传递消息需要的时间，并且想使广播的时间最小。在不同线路中，消息可以同时发出。假定一个消息从一个固定源点发出，并且被传递到所有的站点，而且每个站点仅收到一个消息副本。你可以假定自己是这个网络拓扑中全部信息的控制者。

a. 设计一个算法来判定最优传递，假设线路上仅赋予延迟信息。

b. 设计一个算法判定最优传递，此时在站点上也赋予延迟信息。对于站点 v，把消息传递到它的邻居需要 $t(v)$时间。如果 v 把消息传递到 k 个邻居，则需要 $kt(v)$时间。（对于所有的 v，都已知 $t(v)$的值。）

7.64 令 $G=(V, E)$是一个连通无向加权图。为简单起见，假设权值是正数且各不相同。令 e 是 G 的一条边。用 $T(e)$表示 G 的生成树，且其在包含 e 的所有生成树中具有最小代价。设计一个算法，为所有的边 $e \in E$ 寻找 $T(e)$。算法运行时间必须是 $O(|V|^2)$。

7.65 设计一个有效算法，在一个连通无向加权图中寻找最小瓶颈权值生成树。（回忆一下，瓶颈权值被定义成子图中边的最大权值。）换句话说，要求你找到一棵生成树，其最大权值是最小的。

7.66 对于有向图，求解习题 7.65 中问题的一个变形：输入一个加权有向图 $G=(V, E)$，其中有一个显著的顶点 v。寻找一个以 v 为根的生成树，树中的最大代价边是所有以 v 为根的生成树中最小的。（回忆一下，在根数中所有的边都遵照远离根的方向。）

7.67 令 $G=(V, E)$是一个无向加权图，T 是 G 的一个 MCST。现在假设 G 中所有权值都增加了一

个常数 c。T 是否仍然是 MCST？若不是，则把 T 修改成 MCST 的难度将如何？

* 7.68 令 $G=(V, E)$是一个连通无向加权图，T 是 G 的一个 MCST。假设现在要增加一个新的顶点 v 到 G 中，并且从 v 到其他顶点由加权边连接。设计一个线性算法找到包含 v 的新的 MCST。

7.69 假设生成树的代价不是各条边代价的总和，而是它们代价（所有边的代价都是正数）的乘积。设计一个有效算法，找到一个符合该假定的最大代价生成树（可以假定所有代价各不相同）。

7.70 令 $G=(V, E)$是一个具有 n 个顶点的连通无向图，顶点从 1 至 n 进行编号。设计一个有效算法，找到一个最小 k 值，使得在连续删除编号为 $1, 2, \cdots, k$（按此顺序）的顶点后得到的图中，所有连通分支至多包含 $n/2$ 个顶点。注意删除一个顶点的同时，也删除了与之连接的边（提示：使用 Union–Find 数据结构）。

7.71 令 $G=(V, E)$是一个无向图。一个边集合 $F \subseteq E$ 被称为**反馈边集合**，如果 G 的每条闭链中至少有一条边在 F 中。设计一个算法，找到一个最小规模的反馈边集合。

7.72 令 $G=(V, E)$是一个权值为正数的无向加权图。设计一个算法，在 G 中找到一个具有最小权值的反馈边集合（定义见习题 7.71）。

7.73 证明图 7.22 给出的全部最短路径算法，如果不存在权值为负数的闭链，则对于可能有负数权值的加权图也是正确的。

7.74 令 $G=(V, E)$是一个有向加权图，其中某些权值可能为负数，但不存在权值为负数的闭链（即不存在边的权值总和为负数的闭链）。令 T 是一棵以 v 为根的 G 的生成树。设计一个线性时间算法，判定 T 是否仅包含从 v 出发到 G 中所有其他顶点的最短路径。仅需要输出“yes”或“no”。

7.75 以下是在具有负数权值但无负数权值闭链的加权图中，计算单源最短路径算法的提示。算法从任意有根的生成树开始，见习题 7.74。然后用前面习题的算法判定该树是否是最短路径树。习题 7.74 中的算法应该提供某些证据，假使这棵树不是所需要的树。这个证据用于对树进行改动，该过程一直进行，直至得到最短路径树。

a. 更详细地描述这个算法。

b. 证明算法在 $O(|V| \cdot |E|)$个步骤后终止。

c. 通过选择一个好的初始树来改进算法。这个改进不需要改进最坏情况，只需要改进“一般”情况。

7.76 令 $G=(V, E)$是一个有向加权图，其中某些权值可能为负数。设计一个有效算法，判定这个图是否包含负数权值的闭链。你仅需要输出“yes”或“no”。

7.77 a. 令 $G=(V, E)$是一个有向图，v 是 V 中的顶点。E 中的每条边都被着色，或者为黑色或为红色。设计一个线性时间算法，判定 G 是否有一个包含 v 的简单闭链，该闭链的颜色是交替出现的，即，在闭链中每一条红色（黑色）边有两个黑色（红色）邻居。如果这样的闭链存在，则算法必须至少找到一个。

b. 去掉闭链必须包含某个特定顶点的限制，再求解该问题。

7.78 给定一个连通无向图 $G=(V, E)$，寻找 G 的一棵具有最小高度的生成树。（一棵树的高度是从根到叶子的最大距离。）

7.79 **哈密尔顿路径**是一个包含图中所有顶点的简单路径。设计一个算法，判定一个给定的非循环有向图 $G=(V, E)$是否包含哈密顿路径。算法必须运行在线性时间内。

7.80 在图 7.24 改进过的传递闭包算法中，有 3 层嵌套的循环。第 1 层（最外层）循环选择一列，第 2 层循环选择 1 行，第 3 层循环则对选择的行进行操作。假设我们交换前两层循环，使

得第 1 层选择行而第 2 层选择列。换句话说，在程序中对前两行进行简单交换，由此可得如图 7.46 所示的错误的传递闭包算法。说明为什么这个改动不可行，给出一个算法无法计算的传递闭包的例子。

```
算法 WRONG_Transitive_Closure (A)
输入： A（表示一个加权图的 n×n 邻接矩阵）
{若图包含边(x, y)，则 A[x, y]为真，否则为假，对于所有 x，A[x, x]为真}
输出： 最终，矩阵 A 的值为图的传递闭包
begin
    for x := 1 to n do
        for m := 1 to n do
            if A [x, m ] then
                for y := 1 to n do
                    if A [m, y ] then A [x, y ] := true
```

图 7.46 错误传递闭包算法

7.81 交换传递闭包算法中扫描矩阵的次序（在习题 7.80 中这个设想是不成功的）出于下面的考虑：如果矩阵非常大以致不能存储在主存里，则需要访问二级存储器。假定矩阵是通过行来进行存储的，每行占据一页，现在想使需要从二级存储器读取的页数最小化。如果第 1 层循环通过列来扫描矩阵，则需要读入所有的行才能看到每一列。另一方面，如果交换前两层循环并找到一个为 false 的元素(x, y)，则无须在下一步读取第 y 行。所以，若矩阵是稀疏的（仅含有少数的 1），则只需要读取较少的页面。图 7.46 中的算法是错误的，但可对它进行修改。

a. 请说明，如果运行算法 $O(\log n)$次，则可正确计算出传递闭包。

b. 请说明，事实上，仅运行算法两次就够了。

7.82 令 $G=(V, E)$是一个**多重图**，即一对顶点之间可以有多条边。在此情况下，E 是一个多重集，令 $|E|$ 是所有边的总数。设计一个 $O(|E|+|V|)$的算法以删除每个度为 2 的顶点，方法是用边(u, w)来替换边(u, v)和(v, w)，并用单独一条边来消除边的多重副本。（注意，消除一条边的多重副本可能会产生新的度为 2 的顶点，而这也必须被删除；同时，删除一个度为 2 的顶点可能会产生多重边，这也必须被消除。）

7.83 一个连通无向图 $G=(V, E)$ 被称为**边双连通**的，如果删除任意一条边后余下的图仍然是连通的。设计一个线性时间算法，判定一个图是否为边双连通的。

7.84 给定一个连通无向图 $G=(V, E)$，以及三条边 a、b 和 c，寻找在 G 中是否存在一个包含 a 与 b，但不含 c 的闭链。算法应该运行在线性时间内。

7.85 令 $G=(V, E)$是一个连通无向图，且令 $T=(V, F)$是一棵 G 的生成树。证明，F 与任意双连通分支中边集合的交集，是构成分支的一棵生成树的边的集合。

7.86 图 $G=(V, E)$的一个**双连通扩展**，是一个双连通图 $G'=(V, E')$，其中 $E \subseteq E'$。给定一个无向图 $G=(V, E)$，寻找它的最小双连通扩展；即寻找一个具有最小边数的双连通扩展。（提示：从考虑非常简单的图开始，逐步考虑一般的图。）

7.87 假设给定一个无向图及其全部关节点列表。说明如果不运行整个双连通分支算法，那么应该如何而找到双连通分支。

7.88 令 $G=(V, E)$是一个有向图，令 T 是 G 的一棵 DFS 树。证明，T 与 G 的任意强连通分支的边的交集构成 T 的一棵子树。

7.89 在强连通分支算法（见图7.33）中计算的 *High* 值，并没有从考察的顶点真正指向“最高的”顶点。它仅用于指出是否找到了一个强连通分支。设计一个线性时间算法，对图中的每个顶点 v，（基于一个具有递减 DFS 数的固定 DFS 树）找到从 v 可到达的具有最大 DFS 数的顶点。

7.90 令 $G=(V, E)$是一个连通无向图。设计一个线性时间算法，判定 G 的边是否可以指明方向变成一个强连通的有向图。若存在这样的定向，则算法必须能把它找出来。

7.91 a. 证明以下定理：一个有向图 $G=(V, E)$是强连通的，当且仅当在 G 中存在一个包含每条边至少一次的回路。（注意，在回路中一条边可以出现多次。）

b. 设计一个有效的算法，在给定的强连通图 $G=(V, E)$中找到这样的回路。

7.92 有向图 $G=(V, E)$的**顶点基**是一个最小规模子集 $B \subseteq V$，它具有以下性质：对于 V 中的每个顶点 v，在 B 中有一个顶点 b，且从 b 到 v 存在一条长度大于或等于 0 的路径。证明下面的两个断言，并利用它们设计一个线性时间算法，在一般的有向图中找到一个顶点基。

a. 一个不在闭链上且入度非 0 的顶点，不可能属于任何顶点基。

b. 一个非循环有向图具有惟一的顶点基，并且容易找到。

7.93 一个有向图 $G=(V, E)$被称做是**单向性**的，如果 G 中任意两个顶点 v 和 w 之间，至少有一个能到达另一个。特别指出，所有强连通图都是单向性的。另一方面，也有许多不是强连通的单向性图。例如，只含两个顶点和一条边的图是单向性的，但它不是强连通的。设计一个线性时间（以及线性空间）算法，判定给定的有向图是否是单向性的。（提示：考虑强连通分支图。）

7.94 一个有向图 $G=(V, E)$被称做是单通的（unipathic），如果顶点 w 从顶点 v 出发可达，则从 v 到 w 仅存在一条简单路径。设计一个有效算法，判定一个给定的图 $G(V, E)$是否是单通的。（提示：先对非循环图进行求解。）

7.95 设计一个线性时间算法在一棵树中找到最大匹配。

7.96 不借助网络流和割，直接证明交互路径定理。（提示：给定两个匹配 M_1 和 M_2，研究它们之间对称差的性质；即那些仅出现在两者之一的边的全体。）

7.97 令 G 是一个无向偶图，令 M 是 G 的一个任意匹配。

a. 证明以下定理：在 G 中存在一个最大匹配，它覆盖了全部 M 所能覆盖的顶点。（一个顶点被一个匹配 M 覆盖，如果它与 M 中的一条边相连。）

b. 把上小题中的证明转换成一个算法，为给定的 G 和 M 寻找一个最大匹配。

*7.98 证明 Hopcroft 和 Karp 偶图匹配算法（改进算法参见 7.10 节）在最坏情况下的运行时间是 $O((m+n)\sqrt{n})$。

7.99 假设想在图中找到一个**非一夫一妻匹配**（nonmonogamous）。换句话说，不是寻找相互分离的边，而是寻找相互分离的**星图**（star graphs），它是一棵树，其中一个顶点（根）与所有其他顶点都有边相连。仅含一条边的图是星图的特殊形式，但不含任何边的单独顶点是一个平凡图，不把它看做是星图。令 $G=(V, E)$是一个连通无向图。目标是设计一个算法，在 G 中找到顶点相互分离的星集合，每个星至少有两个顶点。每个顶点至少属于一个星，但不是所有的边都必须属于某个星。换句话说，星必须覆盖所有的顶点，但不需要覆盖所有边。（不存在最小或者最大的约束。）

a. 寻找以下算法的错误，指出错误论点，并给出反例。

错误算法：使用归纳法。归纳假设是已知如何求解顶点数小于 n 的图。给定一个具有 n 个顶点的图 $G=(V, E)$，首先任意挑选一个顶点 v，并删除 v 和它的邻居。余下的图可能

是不连通的，但可以分别考虑各个连通分支，并且通过归纳应用同样的算法。

b. 对这个问题，设计一个有效（且正确的）算法。

7.100 考虑下面的**瓶颈问题**。输入是一个具有 n 个顶点和 m 条边的加权偶图 $G=(V,E)$。定义匹配 M 的瓶颈权值为 M 中最大权值边的权值。设计一个算法，在所有的最大匹配中找到具有最小瓶颈权值的匹配。算法的运行时间应该是 $O(\sqrt{n}\, m\log n)$。

7.101 考虑一个黑白相隔的 $N\times N$ 的木板（类似于棋盘）。利用交互路径定理证明，若任意删除一个黑块或者白块，则余下的木板可以被多米诺骨牌（大小为 2×1）覆盖。

7.102 证明习题 7.101 的定理，方法是在由木板定义的图中寻找哈密尔顿回路，图的定义如下：顶点表示方块，任意两个挨着的方块都相连。

7.103 令 $G=(V,E)$是一个连通无向图。给定 G 的两棵生成树 T 和 R，寻找一个最短的树序列 $T_0, T_1, \cdots, T_k$，使得 $T_0=T, T_k=R$，且每棵树与前面一棵的区别仅仅是增加了一条边及删除了一条边。

7.104 假设在 n 个选手之间举行循环锦标赛。即，每一个选手都要和另外 $n-1$ 个选手进行比赛。不存在平局，所有的比赛结果放在一个矩阵中。通常，对这些选手进行排位是不大可能的，因为 A 可能打败 B，B 可能打败 C，而 C 可能打败 A（换句话说，结果不一定是传递的）。我们感兴趣的是如下“弱”排位。设计一个算法，对于给定的比赛结果矩阵，为各个选手赋予次序 $P_1, P_2, \cdots, P_n$，使得 P_1 打败 P_2，P_2 打败 P_3，依次类推（最后 P_{n-1} 打败 P_n）。算法在最坏情况下的运行时间应该是 $O(n\log n)$。（矩阵中的任何元素都可以在常数时间内访问到。）

7.105 给定 n 个整数 $0\leqslant d_1\leqslant d_2\leqslant\cdots\leqslant d_n$，且 $d_1+d_2+\cdots+d_n$ 是偶数，并且对于每一个 $2\leqslant i\leqslant n$，有 $d_i\leqslant d_1+d_2+\cdots+d_{i-1}$。构造一个具有 n 个顶点，且顶点的度依次为 $d_1, d_2, \cdots, d_n$ 的无向多重图。证明算法的正确性，这样的多重图总是存在。

*7.106 图的**边着色**是对图中的边赋予一个颜色（每条边一个颜色），且与相同顶点连接的两条边具有不同的颜色。设计一个算法，用 k 种颜色对一个无向偶图进行边着色，该图中所有顶点的度 k 都是 2 的幂。算法的运行时间必须是 $O(|E|\log k)$。

7.107 无向图 $G=(V,E)$的**边覆盖**是一个边集合，图中每个顶点都至少与该集合中的一条边连接。设计一个有效算法，为给定的偶图找到一个最小规模的边覆盖。

7.108 无向图 $G=(V,E)$的**顶点覆盖**是一个顶点集合 U，图中每条边都至少与 U 中的一个顶点连接。设计一个有效算法，为给定的树找到一个最小规模的顶点覆盖。（对于一般图的顶点覆盖将在第 11 章中讨论。）

7.109 令 $G=(V,E)$是一棵树，每个顶点赋予了一个等于顶点的度的权值。设计一个算法，找到 G 的**最小权值顶点覆盖**，即具有最小权值的顶点覆盖。

7.110 设计一个有效算法，为给定的偶图寻找一个最小规模的顶点覆盖。（提示：找到与图中最小割的关系。）

7.111 令 $G=(V,E)$是一个无向图。G 中的**独立集**是一个顶点集合，其中任意两个顶点不相连。设计一个有效算法，在给定的偶图中找到最大规模的独立集。一般图中的独立集将在第 11 章中讨论。（提示：找到与习题 7.110 之间的关系。）

7.112 设计一个算法，在给定的无向图 $G=(V,E)$中找到一个**最大独立集**（见习题 7.111）。这个集合不需要是所有独立集中的规模最大者，仅要求：不可通过加入其他顶点进一步扩充，且扩展后仍然是独立的。

7.113 令 $G=(V,E)$是一棵树，每个顶点 v 有一个权值 $w(v)$。设计一个线性时间算法，找出 G 中具有最大权值的独立集（见习题 7.111）。

7.114 令 $G=(V, E)$是一个连通无向图。设计一个算法以判定 G 是否含有一个至多为 k 个顶点的顶点覆盖（见习题 7.108），且所有顶点都是独立的（即覆盖中的任意两个顶点之间不相连）。

7.115 设计一个算法，判定一个无向图 $G=(V, E)$是否含有一个顶点集合 U，要求 U 是一个最小顶点覆盖和最大独立集。若 U 存在，则算法必须能把它找出来。

7.116 **区间图**是一个无向图，其顶点对应实线轴上的区间，如果顶点对应的区间相交，则两个顶点相连。令 $G=(V, E)$是一个区间图，且对应的区间已知。设计一个算法，找到 G 中的最大独立集。

7.117 无向图 $G=(V, E)$是一个**分裂图**，如果它的顶点集合可被划分成两个不相交的子集 U 和 W，且 U 的导出子图没有边而 W 的导出子图是一个完整的图（即含有所有的边）。设计一个线性时间算法，判定给定的图是否为分裂图。

7.118 a. 设计一个算法，判定给定的无向图 $G=(V, E)$是否含有一个三角形的子图。算法的运行时间应该为 $O(|V|\cdot|E|)$。

b. 你的算法可以找到 G 中的所有三角形子图吗?

7.119 a. 设计一个算法，判定给定的无向图 $G=(V, E)$是否含有一个正方形的子图（即长度为 4 的闭链）。算法的运行时间应该为 $O(|V|^3)$。

b. 把你的算法改进到运行时间为 $O(|V|\cdot|E|)$。

可以使用邻接矩阵表示或者邻接列表表示，哪一个方便就用哪一个。

7.120 证明对于给定的无向图 $G=(V, E)$，不存在最坏运行时间为 $O(|V|\cdot|E|)$的找到所有正方形子图的算法。

*7.121 令 T 是一棵有向根树，不要求是二叉树。对每个顶点赋予一个权值，且每个顶点的权值都大于其父母的权值（换句话说，权值满足最小权值在顶部的堆条件）。每个顶点要么是**正规顶点**（regular vertex），要么是**枢轴顶点**（pivot vertex）。一个枢轴顶点的代价等于它的权值。而正规顶点则要打折，代价是其权值减掉一个最接近的且为枢轴顶点的祖先的权值。从而，选择一个枢轴顶点有可能会增加代价，但同时也会减少它的一些后代的代价。对于枢轴顶点的数目无限制。设计一个有效算法，把各个顶点标记为正规顶点或者枢轴顶点，使所有顶点权值的总和最小。

7.122 令 T 是一棵完全二叉树，其高度为 h，有 $n=2^h-1$ 个顶点。现在想按照以下方法把 T 嵌入一个平面中。每一个顶点对应于惟一的格点（即一个有整数坐标的点），相连顶点之间用直线段连接，并且没有两个直线段是相交的。按这种方法把图嵌入到平面中是芯片设计（特别是 VLSI 设计）中的重要问题。本习题中的目标是，使嵌入的布局面积最小。我们定义这个面积是伴随格点（它不被占据）的最小矩形面积，且要包含这个布局。举个例子，一个有 k 个顶点的直链可以嵌入面积为 $2(k+1)$的矩形中。显然，对于任意 n 个顶点的图，可能的最小面积是$\Omega(n)$。

a. 为 T 描述一个需要 $O(n)$面积的布局。（提示：利用分治法；每一棵完全二叉树包含两棵较小的完全二叉树，它们连接到一个共享的根。假定已经知道如何嵌入高度为 $h-1$ 的树，并找到高度为 h 的树的布局。）

b. 设计一个算法，为 T 中的每个顶点计算它在上小题中布局的坐标。

第8章 几何算法

画乃我思所获，非我见所得。
——帕伯罗·毕加索（1881–1973）

8.1 引言

几何算法在计算机科学的众多领域中起着重要的作用，包括计算机图形学、计算机辅助设计、超大规模集成电路 VLSI 设计、机器人技术以及数据库技术等。在一幅计算机生成的图片中可能包括数以千计，甚或百万计的点、线、正方形和圆形；一个机器人可能要做出上千个动作；计算机芯片的设计可能涉及百万个对象，所有这些问题都涉及到几何对象的处理。由于这些问题的输入规模可能很大，所以为它们设计高效的算法就显得尤为重要。

虽然起源于两个略微不同的领域，但遗憾的是最后都被称为计算几何学，其中之一处理几何对象的连续性；另一个则主要处理几何对象的离散特征。二者的区分并不是很大，有许多相似的问题和技术，而本书将侧重于离散性的讨论。本章将讨论若干基本的几何算法，与其他章节一样，这里所讨论的范围是相当有限的。本章将讨论一些基本算法，它们可作为设计更复杂算法的基本模块，也可以阐释一些有趣的技术，而讨论将限定于二维对象。

这一章所出现的对象包括点、直线、线段以及多边形，算法要处理这些对象并计算它们的某些特征，首先介绍的是一些基本定义，然后将讨论表示不同对象的数据结构。点 p 由坐标(x, y)（假定贯穿整章我们都使用一个固定的坐标系统）表示；直线由点 p 和 q 所组成的点对（可以是直线上任意两个不同的点）表示，记做$-p-q-$；线段同样可以由点 p 和点 q 所组成的点对来代表，不同的是这里假定点 p 和点 q 是线段的端点，记做 $p-q$。路径 P 由点的序列 $p_1, p_2, \cdots, p_n$ 和连接它们的线段 $p_1-p_2, p_2-p_3, \cdots, p_{k-1}-p_k$ 组成，有时将路径中的线段称为边。封闭路径是指终点和起点相同的路径，封闭路径也可称为多边形，定义多边形的点又被称为多边形的顶点。例如，三角形是指含有 3 个顶点的多边形。多边形由顶点的序列而非顶点的集合来表示，原因是顶点的顺序非常重要。即使不改变顶点本身，改变它们的顺序也会生成不同的多边形。简单多边形是指对应的路径不自交的多边形，即除了相邻边在公共顶点相交以外，其余边均不相交。简单多边形围成平面的一个区域，一般称之为多边形的内部。凸多边是指满足以下条件的多边形，即连接多边形内部两点的任何线段，其本身全部位于多边形的内部。凸路径是指由点 $p_1, p_2, \cdots, p_n$ 所组成的路径，连接 p_1 和 p_n 将能生成一个凸多边形。

本书假定读者已经熟悉基本的解析几何概念，例如，计算两线段的交点，判定给定点是否位于给定直线的某一侧以及计算两点间的距离，所有这些操作都可以使用基本的算术操作在常数时间内完成。（此处假定平方根运算可以在常数时间内完成；这一点将在 8.3 节讨论。）

存在着大量的“特例”是许多几何算法所共同面临的麻烦，例如，平面中的两条直线通常有惟一的交点，其例外是两条直线平行或者重合。当计算两条给定直线的交点时，需要考虑所有 3 种可能性。更加复杂的对象可能会有许多其他类型的特例，这一点需要特别注意。通常这些特例大多可以通过直截了当的方式处理，但有时候考虑它们将使几何算法的设计和描述变得冗长乏

味，因此本书偶尔会忽略对于理解算法主要思想并不重要的一些细节。

8.2　判定点是否在多边形内部

先从一个简单的问题开始。

> **问题**　给定简单多边形 P 和点 q，判定该点是在多边形内部还是外部。

这个问题看似浅显，但是考虑到复杂的非凸多边形的情况（见图 8.1），这个问题就没有那么简单了。尝试手工解决问题是个好办法，最初的直观方法是尝试以某种方式从给定点到达外部边界。当尝试这种方法时，我们发现只要统计到达外部边界前与多边形边的交点个数即可。例如，图 8.1 中从给定点向西北方向走（沿着图中的虚线），到达外部边界前将与多边形相交生成 2 个交点。由于从外部边界方向看起的第一个交点将我们带入多边形的内部，而第二个交点又将我们带回多边形的外部，故该点位于多边形的外部。一般说来（此时忽略一些特例），点在多边形的内部当且仅当交点（如前所述）的个数为奇数。现在有了一个算法的概要，并在图 8.2 中给出。

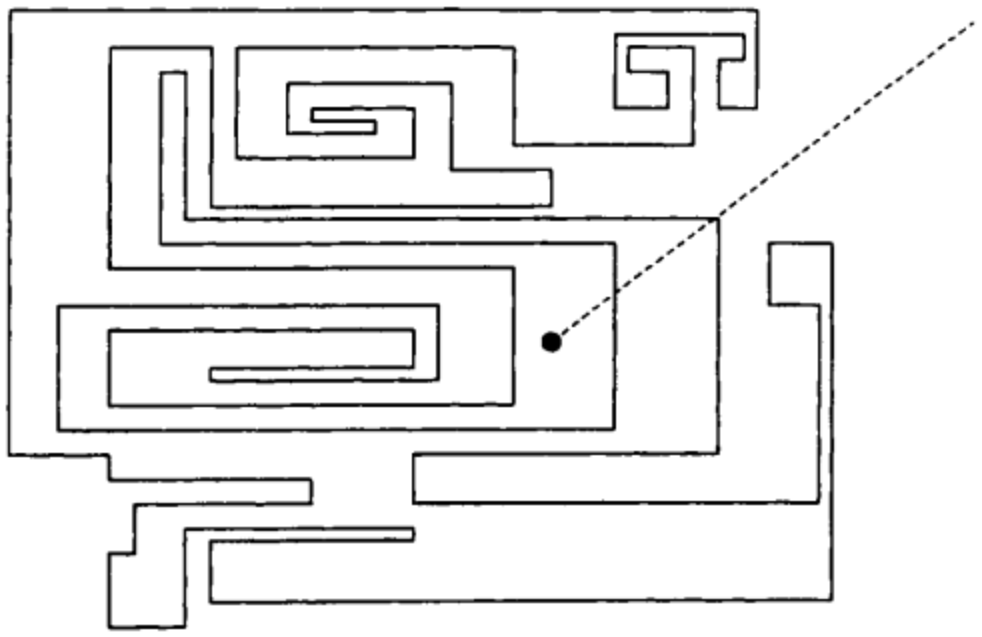


图 8.1　判定点是否在多边形的内部

算法 ***Point_in_Polygon_1 (P, q)***；{初试}
输入： P（一个多边形，顶点为 $p_1, p_2, \cdots, p_n$，边为 $e_1, e_2, \cdots, e_n$）和 q（点）
输出： *Inside*（布尔值，若 q 在 P 中，则值为真，否则为假）

```
begin
    Pick an arbitrary point s outside the polygon ;
    Let L be the line segment q–s ;
    count := 0 ;
    for all edges e_i of the polygon do
        if e_i intersects L then {假定该交点不是多边形的顶点并且
            e_i 和 L 不重合；见下文}
            increment count ;
    if count is odd then Inside := true
    else Inside := false
end
```

图 8.2　算法 *Point_in_Polygon_1*

正如 8.1 节所述，通常有一些特例需要注意。设 s 是多边形外的一点，而 L 是连接 q 和 s 的线段，我们尝试根据 L 和 P 中边的交点个数来判定 q 是否在 P 中。但是，线段 L 可能与 P 中的某些边重合，当一条边与另一条边重合时，称其为一个交点好还是两个交点好呢？在这种情况下，我们显然不想将重合视为相交。另一个特例是 L 与 P 中的一个顶点相交，图 8.3（a）给出了一个例子，其中 L 与 P 中顶点的相交不应计算在内；而图 8.3（b）给出了另一个例子，其中 L 与 P 中顶点的相交应当计算在内。我们将这些特例的刻画以及相应的处理方法留给读者思考（见习题 8.1）。

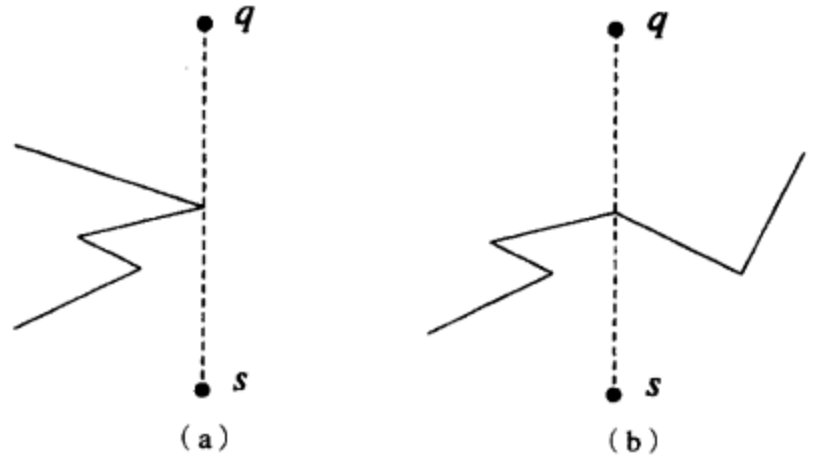


图 8.3　判定点是否在多边形内部的特例

该算法的实现隐式假定我们正在观察图形。在通常的应用场合，输入往往以坐标列表的形式给出，此时的任务就不同了。例如，当手工做这项工作时，我们可以看着这个多边形，因而很容易地找到一条从给定点到外部边界的好的（即与多边形有较少交点）路径。但是，当多边形以坐标序列的形式存储时，这就不是一项容易的任务。以可视化的方式计算交点的个数很容易进行，但当仅仅给出坐标时，这项工作将变得不那么容易。图 8.4 给出了图 8.1 中多边形的坐标列表。（给定点位于中心，坐标|368 308|。）现在如果要求读者仅通过看图 8.4 来解决这个问题，所需的工作显然就是计算所有的交点。如果线段 q–s 平行于坐标轴之一（例如竖直坐标轴），则这项工作可以大大简化。虽然该特定线段上的交点个数可能远大于最优线段上的，但我们并不需要找到最优线段（这个问题实际上要难得多，参见习题 8.3），此外，计算该特定线段上的交点将变得非常简单。修改后的算法如图 8.5 所示。

320.00 368.00 | 320.00 384.00 | 288.00 384.00 | 288.00 380.00 | 308.00 380.00 |
308.00 376.00 | 280.00 376.00 | 280.00 392.00 | 332.00 392.00 | 332.00 364.00 |
364.00 364.00 | 364.00 352.00 | 256.00 352.00 | 256.00 404.00 | 224.00 404.00 |
224.00 332.00 | 352.00 332.00 | 352.00 288.00 | 224.00 288.00 | 224.00 312.00 |
320.00 312.00 | 320.00 300.00 | 256.00 300.00 | 256.00 296.00 | 328.00 296.00 |
328.00 320.00 | 208.00 320.00 | 208.00 280.00 | 384.00 280.00 | 384.00 340.00 |
240.00 340.00 | 240.00 396.00 | 248.00 396.00 | 248.00 348.00 | 416.00 348.00 |
416.00 272.00 | 320.00 272.00 | 320.00 256.00 | 448.00 256.00 | 448.00 320.00 |
432.00 320.00 | 432.00 340.00 | 452.00 340.00 | 452.00 224.00 | 256.00 224.00 |
256.00 244.00 | 320.00 244.00 | 320.00 248.00 | 248.00 248.00 | 248.00 216.00 |
224.00 216.00 | 224.00 240.00 | 232.00 240.00 | 232.00 256.00 | 288.00 256.00 |
288.00 264.00 | 224.00 264.00 | 224.00 272.00 | 192.00 272.00 | 192.00 416.00 |
428.00 416.00 | 428.00 384.00 | 416.00 384.00 | 416.00 400.00 | 424.00 400.00 |
424.00 408.00 | 384.00 408.00 | 384.00 384.00 | 400.00 384.00 | 400.00 396.00 |
388.00 396.00 | 388.00 404.00 | 408.00 404.00 | 408.00 372.00 | 352.00 372.00 |
352.00 404.00 | 264.00 404.00 | 264.00 368.00 |

图 8.4　图 8.1 中的多边形以坐标序列的形式给出

```
算法 Point_in_Polygon_2 (P, q); {再试}
输入: P (一个多边形，顶点为 p1, p2, ⋯, pn，边为 e1, e2, ⋯, en) 以及 q=(x0, y0) (点)
输出: Inside (布尔值，若 q 在 P 中，则值为真，否则为假)
begin
    count := 0 ;
    for all edges ei of the polygon do
        if the line x=x0 intersects ei then
        {假定该交点不是多边形的顶点并且直线 x = x0 与 ei
            不重合}
            Let yi be the y coordinates of the intersection between
                the line x=x0 and ei ;
            if yi < y0 then {交点在 q 的下方}
                increment count ;
    if count is odd then Inside := true
    else Inside := false
end
```

图 8.5　算法 *Point_in_Polygon_2*

例如，对于以坐标(368, 308)表示的点 q，要判定其位于图 8.4 中所给出的多边形的内部还是外部。为了计算从 q 竖直向下的线段与多边形的交点个数，我们需要查看所有的边，对于那些 y 坐标小于 308 的边，检查其 x 坐标是否跨越了 368。以下 4 条边跨越了该线段：

(208, 280)—(384, 280)；

(416, 272)—(320, 272)；

(320, 256)—(448, 256)；

(452, 224)—(256, 224)。

因此，这个点位于多边形的外部。

复杂度　计算平面中两条线段的交点需要花费常数时间，该算法计算 n 次这样的交点（此处 n 指多边形的大小），而执行其他操作需要花费常数时间。因此，该算法总共运行时间是 $O(n)$。

评述　在许多场合下，简单的解决方法往往源自对手工（或者目测）计算过程的分析，然而对于大规模的输入所得的解决方案往往并不高效。但是，在某些场合，也可能不仅简单，而且高效。从“可视化”入手是个好办法，有时通过观察就可以得到算法。在这个例子中，通过观察图形可以发现，注意沿着给定点到外部边界的某条路径，而忽略其他东西，就能解决这个问题，而该观察确实引导我们得到了算法。

8.3　构造简单多边形

一个平面上的点的集合对应许多不同的多边形，它们取决于这些点的顺序。这一节着重寻找由一个点集合所定义的简单多边形。

> **问题**　给定平面中 n 个点所组成的集合，将它们连接起来形成一条简单的封闭路径。

有许多构造简单多边形的方法，这里介绍其中一种，它对应于手工解决这个问题时所采用的方法。考察包含所有点的圆 C，通过始于 C 中心的旋转线来对 C 的区域进行扫描（见图 8.6），不妨暂时假定旋转线每一时刻至多扫描经过一个点。如果按照扫描经过的先后顺序将这些点连接起

来，就会得到一个简单多边形。现在来证明这个论断，根据旋转线经过它们的先后顺序，用 p_1，p_2，…，p_n 表示这些点（第一个点任意选取）。对于所有满足 $1 \leqslant i < n$ 的 i，边 p_i–p_{i+1} 均包含在圆的不同区域中；因而，它也不会与其他边相交。但是，这并不足以证明所构造的多边形是简单的，事实上，它有可能不是。某些 p_i 和 p_{i+1} 之间的角度可能会超过 180°，在这种情况下，边 p_i–p_{i+1} 对应的区域包含了大半个圆。这样，边 p_i–p_{i+1} 会经过其他区域，并且可能和其他边相交。（为了验证这种可能性，可以考虑一个大圆，其中心位于图 8.6 中的圆的外部某处。）这是一个解决几何问题经常遇到的“特例”的典型例子，我们必须非常仔细以确保考虑到所有情况。（当然，对任何一种算法都要这般考虑，只是这类问题在几何算法中更加普遍。）

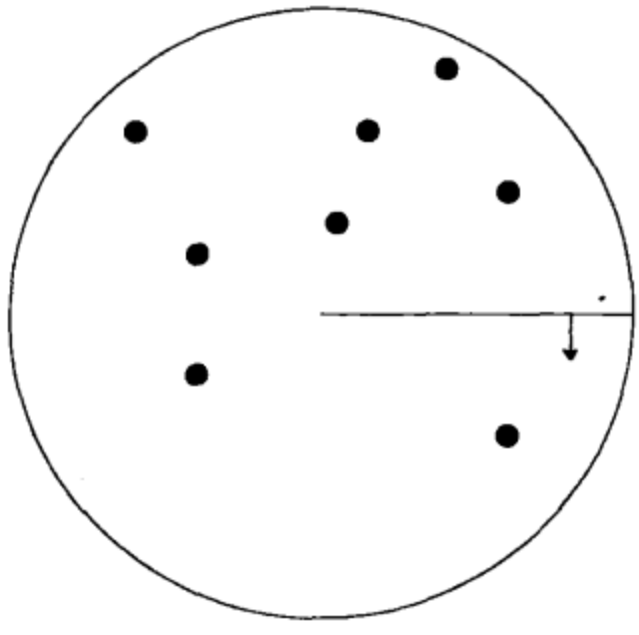


图 8.6　点的扫描

克服这个障碍是很容易的。例如，可以取出集合中任意 3 点，选取这 3 点所组成的三角形的内部一点作为圆心，这样的选择可以确保该圆在 180°的区段内必包含点集合中的点。我们将要使用的是另一种解决方法，选取点集合中的一点作为圆心，即选取有最大 x 坐标的点 z（如果多个点有最大 x 坐标，则选取其中 y 坐标最小的点）。现在，基本上可以使用相同的算法，根据这些点在以 z 为中心的圆中的位置对它们进行排序。这些位置可以通过下列方式计算，将点 z 和另一点的连线与一条固定直线（如 x 轴）之间的角度排序。如果两个（或两个以上）点有相同的角度，那么进一步根据它们和 z 的距离进行排序。接着将 z 分别连接到最小角度的点和最大角度的点，并按顺序连接其他点。由于其他所有点都位于 z 的左边，所以先前提到的坏的情况将不会发生。（但当所有点都位于一条直线上时，仍然存在一种特例；在那种情况下，任何经过这些点的多边形都会有重合的边。）图 8.7 给出了图 8.6 中的点以这种方法所获得的简单多边形。

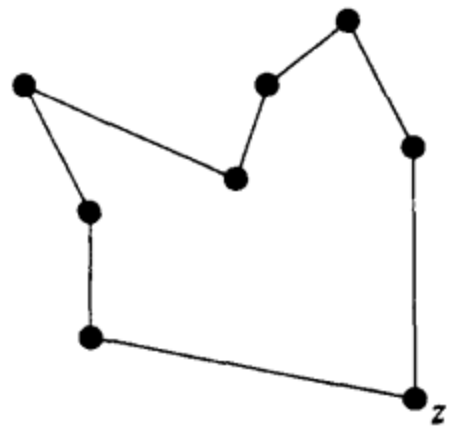


图 8.7　构造一个简单多边形

基于相同的原理，可以在两方面对该方法进行改进。首先，算法中使用角度只是为了发现连接这些点的先后顺序，因此我们不需要计算确切的角度。相同的顺序也可以通过这些线的斜率（即 y 坐标差值和 x 坐标差值的比值）得到，而计算斜率比计算角度更容易（不需要计算反正切的值）。其次，当两点有相同斜率时，同理可以避免计算距离，只要计算距离的平方值就够了！因此，计算平方根是多余的，图 8.8 给出了这个算法。

算法 ***Simple_Polygon*** $(p_1, p_2, \cdots, p_n)$
输入： $p_1, p_2, \cdots, p_n$（平面上的点）
输出： P（一个简单多边形，其顶点是以某种方式排列的）
begin
　for i := 2 to n do
　　compute the angle α_i *between the line* $-p_1-p_i-$ *and the x axis ;*
　　{有时取一个极端的点更有效，比如点集合中具有最大横坐标的点（如果几个点都具有最大横坐标，则取其中纵坐标最小的），而不是 p_1}
　Sort the points according to the angles $\alpha_2, \cdots, \alpha_n$ *;*
　　{如果夹角相同，则按照它们与 p_1 的距离排序}
　P is the polygon defined by the list of points in sorted order
end

图 8.8　算法 *Simple_Polygon*

复杂度　该算法的运行时间主要由排序操作所决定，后者需要的时间为 $O(n\log n)$。

8.4　凸包

点集合的凸包定义为包围所有点的最小凸多边形。可将凸包表示成一个正规多边形，即顶点的循环列表。

问题　计算平面中给定 n 个点的凸包。

处理凸多边形要比处理任意多边形容易，在某种意义上，凸包是包围一个点集合的最小“适宜”区域。凸包的顶点均来自集合中的点，如果某点是凸包的顶点，则称其属于凸包。凸包既可以仅包含 3 个顶点，又可以将所有点均作为其顶点。凸包有多种用途，因而存在大量计算凸包的算法。

8.4.1　直接方法

像往常一样，首先使用直接的归纳方法，很容易发现 3 个点的凸包，假定已经知道如何寻找小于 n 个点的凸包，接着尝试寻找 n 个点的凸包。这第 n 个点如何改变由先前$(n-1)$个点所形成的凸包呢？有两种情况：或者这个额外的点落在凸包内部，在这种情况下，凸包维持不变；或者该点落在凸包外部，在这种情况下，需要将凸包“延拓”以达到该点（如图 8.9 所示）。所以，需要解决两个子问题：判定一个点是否在凸包中，以及当该点在凸包外部时能够延拓这个凸包，这些问题并不容易。这里，需要对直接的归纳方法的使用做若干改进，当然这方面我们已经有了一些经验。第一种改进是，要特别选取第 n 个点，而非任取。尝试选择凸包中的点，这样就不需要做凸包延拓工作了。但是，有一些点必须包括在凸包中，甚至在一些场合下，所有的点均属于该凸

包；另一种可能的选择就是极端点（extreme point），它类似于最大点或者最小点，这种改进对前一个问题的解决也很有效。

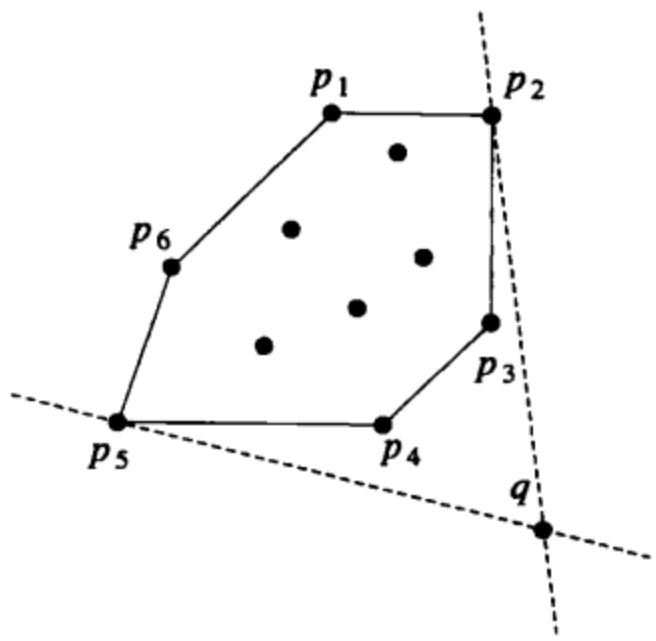


图 8.9 延拓凸多边形

再次选取具有最大 x 坐标值的点（如果多个点有着相同的最大 x 坐标值，则选取其中 y 坐标最小的点），我们将这个点标记为 q。显然，可以确保 q 为凸包的顶点，这样，惟一的问题就是如何修改（延拓）该凸包使之包括 q。首先，需要的是寻找原先属于旧凸包并且现在位于新凸包内部的点（图 8.9 中的 p_3 和 p_4），然后将它们删除；接着，必须在两个现有顶点（图 8.9 中的 p_2 和 p_5）之间将这个新点作为新顶点插入。凸多边形的支撑线（supporting line）是指与多边形恰好只相交于其上一个顶点的直线。这样，多边形就全部位于支撑线的一侧。现在考察支撑线-q-p_2-和支撑线-q-p_5-（见图 8.9），通常情况下，该多边形中只有两个顶点与 q 存在连线，这些连线才是支撑线。（这里忽略掉两个或者两个以上的点与 q 共线的特例。）多边形位于两条支撑线之间，而这恰好就是我们想要的修改方式。在所有多边形中的点与 q 的连线中，两条支撑线和 x 轴之间分别有最大和最小角度。为了寻找这两个顶点，需要考虑从 q 到所有顶点 $p_1, p_2, \cdots, p_n$ 的连线，计算它们的角度，并选取最大值和最小值（见习题 8.4）。一旦识别了这两个极端顶点，就可以构造修改后的凸包。（使用其他一些方法也可以修改凸包，这里介绍的方法不一定是最佳的，选取它是因为它比较简单。）由于不久将给出一个更快的算法，所以这里忽略了该算法的一些相关细节。

复杂度 对于每个点，需要计算其与先前诸点的角度，寻找其中最大和最小角度，并从顶点列表中添加和删除点，所以涉及处理第k个点的工作耗时为$O(k)$。基于递推关系，$T(n) = T(n-1)+O(n)$的解为$O(n^2)$，因而该算法的运行时间是$O(n^2)$。虽然算法同样要求排序操作，但是该算法的运行时间主要由其他操作所决定。

8.4.2 礼品包裹算法

怎样才能改进上述算法呢？当逐点延拓多边形时，我们可能会构造一些含有不在最终凸包上的点的凸多边形，而这会花费许多时间，但应该如何避免呢？如果不考虑点集的子集的凸包，而是从整个集合出发直接构造凸包，即从一个极端点（它必须在这个凸包上），通过寻找支撑线的方法发现它在凸包上的相邻顶点，然后以相同方式从这些相邻顶点出发继续该过程。这样的算法就是著名的礼品包裹算法（gift wrapping），即从“礼品”的一个顶点出发，通过一次次地寻找相邻顶点给礼品裹上包装。该算法如图 8.10 所示，当然也可以对它进行修改以解决高维的问题。

```
算法 Gift_Wrapping (p1, p2, ···, pn)
输入：p1, p2, ···, pn（平面上的点集合）
输出：P（由 p1, p2, ···, pn 组成的凸包）
begin
    set P to be the empty set ;
    Let p be the point in the set with the largest x coordinate
        (and the smallest y coordinate, if there are several points
        with the same largest x coordinate) ;
    Add p to P ;
    Let L be the line containing p which is parallel to the x axis ;
    while P is not complete do
        let q be the point such that the angle between the line –p–q– and L
            (in counterclockwise fashion) is minimal among all points ;
        add q to P ;
        L := line –p–q– ;
        p := q
end
```

图 8.10　算法 *Gift_Wrapping*

礼品包裹算法是以下（对 k）归纳假设的直接应用。

归纳假设：给定平面中 n 个点的集合，可以找到长度 $k<n$ 的凸路径，使它成为该点集合凸包的一部分。

有了这个假设，重点就从延拓凸包变为延伸凸路径，即我们不再寻找更小点集合的凸包，而是寻找最终凸包的一部分。

复杂度　为了将第 k 个点添加进凸包，必须在 n–k 条直线中寻找最大和最小角度。所以，礼品包裹算法的运行时间是 $O(n^2)$，这并不比先前的延拓算法优越。

8.4.3　Graham 扫描算法

现在介绍一个能在 $O(n \log n)$时间内计算凸包的算法。类似于在 8.3 节中所述的构造简单多边形的算法，该算法首先根据角度将这些点排序。设 p_1 是有最大 x 坐标的点（如果多个顶点都有相同的最大 x 坐标值，则选取其中 y 坐标最小的点），对于每个点 p_i，计算直线–p_1–p_i–和 x 轴之间的角度并根据这些角度将这些点排序（见图 8.11）。现在按照它们在多边形中出现的顺序扫描这些点，并且和先前一样尝试发现凸包的顶点。如同礼品包裹算法，我们将维持一条由目前为止扫描过的点的子集所组成的路径。该路径是一条凸路径，其所对应的凸多边形包围了所有目前为止扫描过的点。（将该凸路径的起讫点连接起来形成了所对应的凸多边形。）因此，当所有点均被扫描到时，就找到了这个凸包。这个算法和礼品包裹算法的主要区别是，所维持的路径不一定是最终凸包的一部分，它只是我们目前为止扫描到的点的凸包的一部分。可能包含一些不在最终凸包中的点，但这些点将在以后被除去。例如，图 8.11 中从 p_1 到 q_m 的路径是凸的，但是 q_m 和 q_{m-1} 显然不属于最终凸包。从上述讨论，可以得到一个算法，它主要基于以下归纳假设。

归纳假设：给定平面中 n 个点的集合，其按照（见 8.3 节中）算法 *Simple_Polygon* 排序，我们可以在前 k 个点中找到一条凸路径，它所对应的凸多边形包围了前 k 个点。

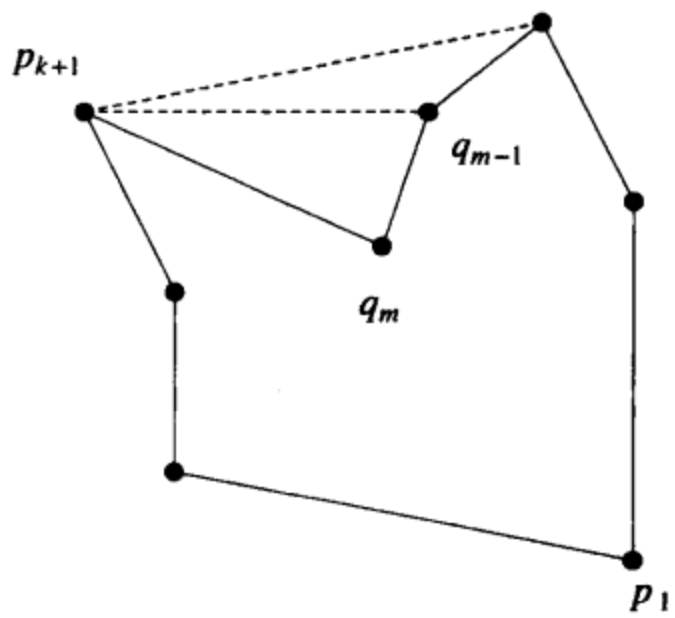


图 8.11 Graham 扫描

$k=1$ 时的情况显然成立。用 $P=q_1, q_2, \cdots, q_m$ 表示从前 k 个点中所（归纳）得到的凸路径，现在要将这个假设推广到$(k+1)$个点。考察直线-q_{m-1}-q_m-和-q_m-p_{k+1}-间的角度（见图 8.11），如果其小于 180°（该角度从多边形的内部测量），那么 p_{k+1} 可以加入到现有路径中（加入它之后的路径仍然是凸的），任务也就完成了。否则，q_m 会位于 P 中去除 q_m 并加入 p_{k+1} 再连接 p_{k+1} 和 p_1 后得到的多边形的内部，因为这些点已经根据其角度进行了排序，直线-p_1-p_{k+1}- 在前 k 个点的左侧，故 q_m 一定位于如上定义的多边形的内部，所以 q_m 可以从 P 中去除而 p_{k+1} 可以添加进 P。任务是否完成了呢？还没有。虽然 q_m 可以被去除，但修改后的路径不一定是凸的。事实上，图 8.11 清楚地表明还要去除其他点。例如，q_{m-1} 现在可能位于修改后路径所定义的多边形的内部。我们必须继续检查路径的后两条边，直至找到两条所成角度小于 180°的边。由于此时路径为凸，所以将假设推广到$(k+1)$个点，任务也就完成了。图 8.12 给出了算法的具体细节。

算法 ***Graham's_scan*** $(p_1, p_2, \cdots, p_n)$
输入：$p_1, p_2, \cdots, p_n$ 平面上的点集合
输出：$q_1, q_2, \cdots, q_m$（由 $p_1, p_2, \cdots, p_n$ 组成的凸包）

```
begin
    Let p1 be the point in the set with the largest x coordinate
        (and smallest y coordinate if there are several points
        with the same largest x coordinate) ;
    Use algorithm Simple_Polygon to arrange the points around p1
        in sorted order ; let the order be p1, p2, ..., pn ;
    q1 := p1 ;
    q2 := p2 ;
    q3 := p3 ;
       {初始时，P 包括 p1、p2 和 p3}   {p3 }
    m := 3 ;
    for k := 4 to n do
        while the angle between –qm–1–qm– and –qm–p   - is ≥ 180 degrees do
            m := m - 1 ;
        m := m + 1 ;
        qm := pk
end
```

图 8.12 Graham 扫描算法

复杂度　该算法的复杂度主要由排序操作所决定，所有其他步骤仅需要 $O(n)$时间。集合中的每一点作为 p_{k+1} 在归纳步骤中恰好被考虑过一次，那时总是将该点加入到凸路径中。相同的点将在以后再次考虑（可能不止一次），以验证其是否包含于当前凸路径，我们将这个阶段称为追溯（backward）测试。一次追溯测试中涉及的点的数目可能很大，但是除了两个点（当前点和被认为仍然在凸路径中的点）之外，其他点均被去除了！因此，添加和删除每一个点都只需要花费常数时间，总的来说，这个阶段需要 $O(n)$步。这样，取决于排序操作，该算法总的运行时间为 $O(n \log n)$。

8.5　最近点对

假定给定 n 个对象的位置，现在希望检查没有两个对象靠得太近。例如，对象可以对应于计算机芯片部件、银河中的星系或者灌溉系统等。本节将讨论该问题的一个变形，它也是一大类相近问题中的一个典型例子。

> **问题**　给定平面中 n 个点所组成的集合，寻找一个相距最近的点对。

其他相近问题包括：为集合中的每一个点寻找与之最接近或者第 k 个接近的点，以及为一个新给定的点寻找与之最接近的点。

直接方法

直接解法之一是检查所有的点对，然后取最小值。该方法需要 $n(n-1)/2$ 次距离计算以及 $n(n-1)/2-1$ 次比较。使用归纳法的直接解法是，先去除一个点，然后解决$(n-1)$个点的问题，最后再考虑这个额外的点。但是，若从$(n-1)$个点的情况中获得的惟一信息是最小距离，那么需要检查该额外点到其他所有$(n-1)$个点的距离。因此，总距离计算量 $T(n)$满足递归关系 $T(n)=T(n-1)+n-1$，$T(2)=1$，由此可得 $T(n)=O(n^2)$。事实上，这两个直接解法完全相同，对于 n 值很大的情况，我们需要找到更高效的算法。

分治算法

将点集合分成相等的两部分，而不是一次只考虑一个点，归纳假设仍然保持不变，因此这里就不是将 n 个点的问题归约到$(n-1)$个点的问题，而是将它归约到两个$n/2$ 个点的问题。为简单起见，这里假定 n 是 2 的幂，以便总可以将集合分成两个相等部分。有许多方法可以将点集合等分，根据目标这里可以选择最佳等分。在解决更小问题过程中应该获得尽可能多的有用信息，这样在考虑整个问题时，那些信息仍然有效。因此将平面划分成两个不相交的部分，每个部分包含集合的一半，这样的划分方法较合理。当在每个部分找到最小距离以后，只要考虑那些靠近集合边界的点之间的距离。划分集合最容易的方法是，将这些点以其 x 坐标排序，然后用平分集合的竖直线划分平面（见图 8.13）。（如果有若干点位于该竖直线上，则可以任意划分它们。）选择该划分方法的目的是使较小问题的解的组合工作最小化，这样排序操作只要进行一次即可。

为简单起见，这里只着重考虑寻找点之间的最小距离，而最近的两点的距离可以直接由算法得出。如果集合仅有两个点，那么可直接取它们之间的距离。设 P 是 n 个点的集合并假定 n 是 2 的幂，如前所述，首先将 P 划分成大小相等的两个子集 P_1 和 P_2。根据归纳，我们寻找每个子集中的最近距离，假设 P_1 和 P_2 中的最小距离分别是 d_1 和 d_2，并且不失一般性，假定 $d_1 \leqslant d_2$。我们需要寻找整个集合的最短距离；也就是说，必须检查是否存在 P_1 中的点和 P_2 中的点，满足二者

的距离小于 d_1。首先，只需要考虑那些位于以两个子集竖直分界线为中轴、宽度为 $2d_1$ 的带状区域（由两条宽度为 d_1 的带组成）中的点（见图 8.13），在那个区域以外不可能存在点与另一子集中的点的距离小于 d_1。利用这一观察结论，通常可以将许多点排除出所考虑的范畴，但是在最坏情况下，所有的点可能全部位于该带状区域，我们不能“承受”为它们使用直接算法所付出的开销。

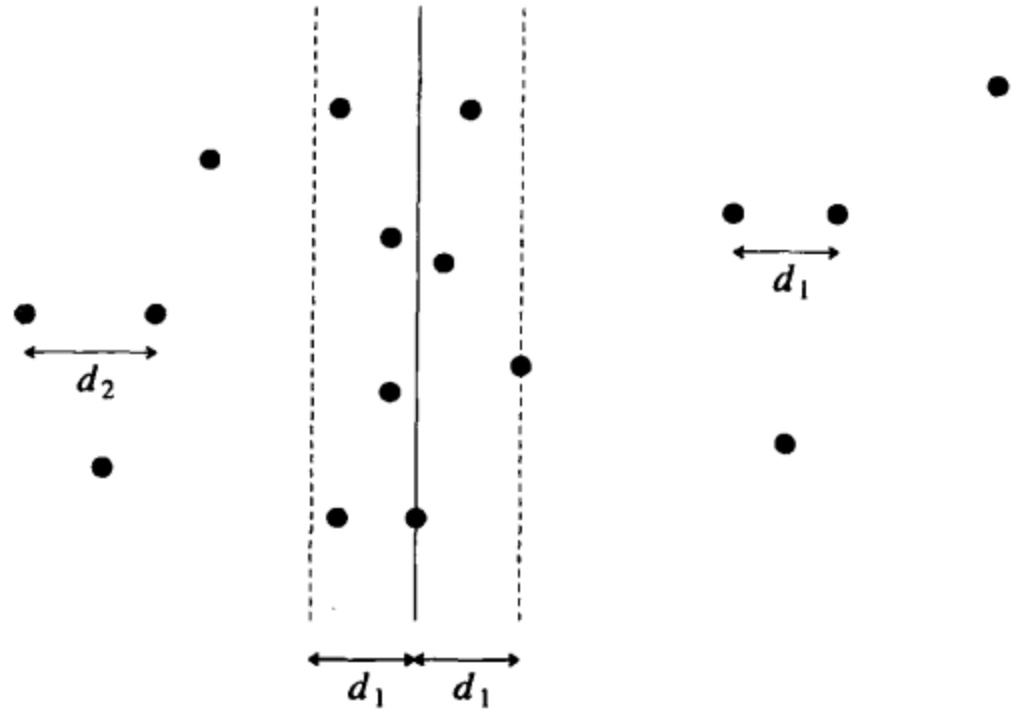


图 8.13　最近点对问题

另一个相对不太明显的观察结论是，对于带状区域中的任意一点 p，在另外一边只有很少数量的点与 p 之间的距离小于 d_1，这是因为在每条带中，所有的点至少相距 d_1。如果带中的点 p 的 y 坐标为 y_p，那么在另外一边只需要考虑 y 坐标 y_q 满足 $|y_p-y_q| < d_1$ 的点，而在另一边带中最多只可能有 6 个这样的点（最坏情况可参考图 8.14）。因此，若将带状区域中的所有点根据其 y 坐标排序并依次扫描这些点，则对于每个点只须按照那个顺序检查其常数多个相邻点（而非所有$(n-1)$个点）。图 8.15 给出了该算法的概要。

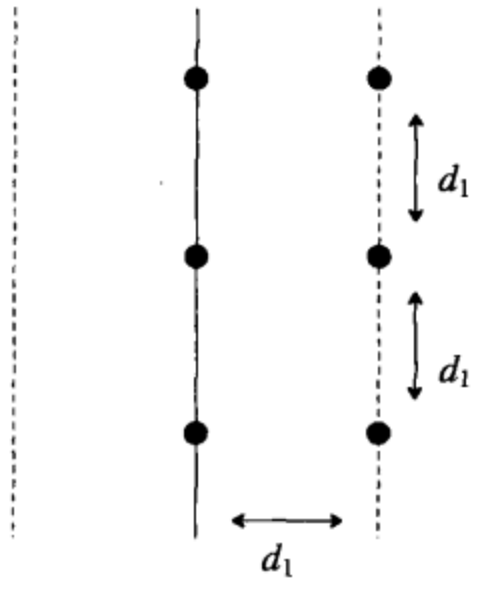


图 8.14　最坏情况下 6 个两两相距 d_1 的点

算法 *Closest_Pair_1* $(p_1, p_2, \cdots, p_n)$；{初试}
输入： $p_1, p_2, \cdots, p_n$（平面上 n 个点组成的集合）
输出： d（集合中两个最近的点的距离）

```
begin
    Sort the points according to their x coordinates ;
        {该排序过程仅在算法开始时进行一次}
    Divide the set into two equal-sized parts ;
    Recursively, compute the minimal distance in each part ;
    Let d be the minimal of the two minimal distances ;
    Eliminate points that lie farther than d apart from the separation line ;
    Sort the remaining points according to their y coordinates ;
    Scan the remaining points in the y order and compute the distances of
        each point to its five neighbors ;
    if any of these distances is less than d then
        update d
end
```

图 8.15　算法 *Closest_Pair_1*

复杂度　根据 x 坐标排序要花费 $O(n \log n)$步，但是这样的排序只须进行一次。我们接着解决两个规模为 $n/2$ 的子问题，消除带状区域外的点可以在 $O(n)$步内完成，根据 y 坐标排序需要花费 $O(n \log n)$步。最后，扫描带状区域中的每个点，并将它依次与其常数多个相邻点比较，该过程需要花费 $O(n)$步。总的说来，解决规模为 n 的一个问题，首先要解决两个规模为 $n/2$ 的子问题再使用 $O(n \log n)$步组合这些解（外加一开始对 x 坐标排序的 $O(n \log n)$步），由此可获得以下递归关系：

$$T(n) = 2T(n/2) + O(n \log n), \quad T(2) = 1。$$

读者可自行验证该递归关系的解为 $T(n) = O(n \log^2 n)$，其渐近复杂度比平方算法好，但还可以做得更好。

$O(n \log n)$的算法

这里的关键思想是加强归纳假设，我们在组合步骤中不得不花 $O(n \log n)$时间的原因在于对 y 坐标的排序。虽然我们知道怎样直接解决排序问题，但这样花费的时间太长了，能否想办法在解决最近点对问题的同时解决排序问题呢？换句话说，我们想要引入排序操作以加强最近点对问题的归纳假设。

归纳假设： 给定平面中小于 n 个点的集合，已知怎样找到最近距离以及怎样输出根据 y 坐标排序后的点集合。

假设这些点在每一步中都根据其 y 坐标排序，已经知道如何找到最小距离。因此，惟一要做的就是推广该假设，使得当包含 n 个点集合的两个（大小为 $n/2$ 的）子集已排序时，它能够对原集合中的 n 个点进行排序，而这种排序恰好就是（6.3.2 节中的）归并排序。该方法的主要优点是，不必在每一次组合解时都重新排序——我们只需要归并即可。由于归并可以在 $O(n)$步内完成，递归关系变成 $T(n) = 2T(n/2) + O(n)$, $T(2) = 1$，从而推得 $T(n) = O(n \log n)$。图 8.16 给出了修改后的算法。

算法 *Closest_Pair_2* ($p_1, p_2, \cdots, p_n$)；{修改版}
输入： $p_1, p_2, \cdots, p_n$（平面上 n 个点组成的集合）
输出： d（集合中两个最近的点的距离）
begin
Sort the points according to their x coordinates ;
{该排序过程仅在算法开始时进行一次}
Divide the set into two equal-sized parts ;
Recursively do the following:
compute the minimal distance in each part ;
sort the points in each part according to their y coordinates ;
Merge the two sorted lists into one sorted list ;
{注意必须在去掉点之前进行归并；还需要为下层递归提供整张有序表}
Let d be the minimal of the minimal distances ;
Eliminate points that lie further than d apart from the separation line ;
Scan the points in the y order and compute the distances of each point to its five neighbors ;
if *any of these distances is less than d* ***then***
update d
end

图 8.16 算法 *Closest_Pair_2*

8.6 水平线段和竖直线段的交点

交点问题在计算几何中相当普遍，它们有许多应用场合。我们有时对计算若干对象的交点感兴趣，有时则仅对检测对象之间是否有交点存在感兴趣。检测问题通常容易些，虽然并不总是容易很多。在这一节，我们给出了一种交点问题，它阐释了计算几何中一项重要的技术。相同的技术也可应用于其他交点问题（以及其他问题），习题中将给出一些这样的问题。

问题 给定平面中 n 条水平线段和 m 条竖直线段，寻找它们之间的所有交点。

这个问题十分重要，一个例子就是 VLSI 电路设计，电路可能包括几十万条“线路”，设计者必须确保不存在预料之外的交点，其在隐线消除（hidden-line elimination）领域内也是非常重要的。（隐线消除问题通常更加复杂，因为线不一定是水平或者竖直的。）为简单起见，在不引起混淆的情况下，本节中将线段简称为线。图 8.17 给出了该问题的一个例子。

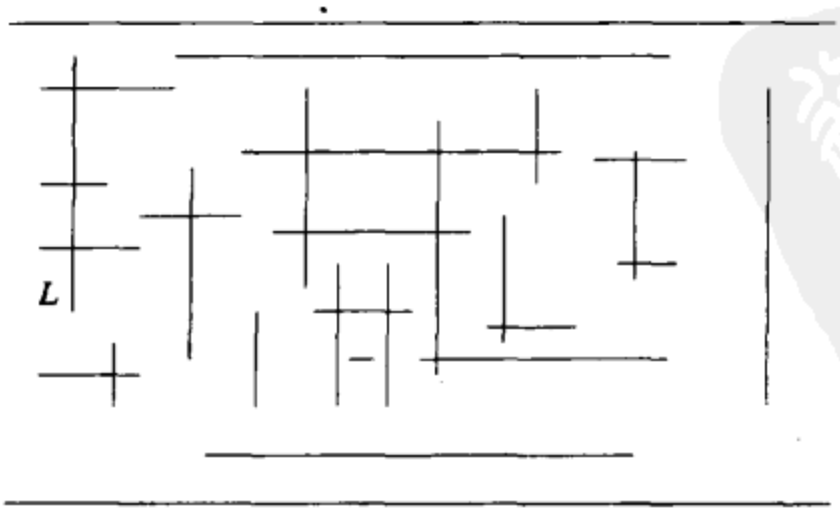


图 8.17 水平线和竖直线的交点

寻找所有竖直线之间或者所有水平线之间的交点是一个简单的问题，我们把它留做练习。为简单起见，假定任何两条竖直线之间和任何两条水平线之间均没有交点。如果通过每次去除一条线的方法来简化问题，那么被去除的线将与所有其他的线进行比较，这样的算法涉及 $O(mn)$次比较。一般来说，可能会多至 mn 个交点，仅仅给出它们就需要 $O(mn)$时间。但是，交点的个数可能远小于mn。所以我们想找到一个算法，当交点个数很少时，该算法的性能很好；而当交点个数很多时，其性能也不会太差。该算法可以通过结合我们最喜欢的两种技术加以实现，即选择特定的归纳顺序以及加强归纳假设。

归纳顺序取决于一条从左向右“横扫”平面的假想线（是无限直线，而非线段），线段按照其与这条假想线的相交顺序排列。除了计算交点以外，还要保存我们目前已经遇到的线段的相关信息。该信息有助于更高效地计算后继交点，这项技术被称为扫描线（line-sweep）技术。

设从左向右横扫平面的假想线是一条竖直线，为了达到这种横扫效果，我们对所有线段的端点根据其 x 坐标进行排序。竖直线的两个端点由于有相同的 x 坐标，只须取其一即可，但是每条水平线的两个端点必须都用上。在所有端点都排序完成以后，我们按照该顺序依次考虑它们。如同通常的归纳方法，假定已经计算了先前线段间的交点并且还维持了一些额外信息，现在尝试处理下一条线段并更新相应的信息。算法的结构如下：从左向右一次考虑一个端点，使用目前为止所收集到的信息（还没有特别指明）处理该端点，寻找由它产生的交点，并为下一个端点更新相应的信息。该算法的主要部分是定义所维持的信息，让我们试着运行该算法并从中发现所需信息。

该归纳假设自然具备的一个特点是，出现在当前扫描线位置左方的所有交点均已知。检测交点应当发生在遇到竖直线时，还是在遇到水平线时？选择前者应该更妥当。当我们遇到竖直线时，与它相交的水平线仍然在考虑范畴之内（由于我们还没有到达其右端点）；而另一方面，当遇到水平线的左端点或右端点时，我们或者还未曾遇到与之相交的竖直线，或者已经将那些竖直线忘了。假定当前扫描线位于竖直线 L 的 x 坐标处（见图 8.17），寻找到所有与 L 相关的交点需要哪一类信息？由于当前扫描线左方的所有交点假定都已知，不需要进一步考虑那些右端点位于扫描线左方的水平线。因此，只要考虑那些左端点位于扫描线左方，而右端点位于扫描线右方的水平线（图 8.17 中有 6 条这样的线），我们需要维持这些水平线的列表。当遇到 L 时，要检查它与这些水平线是否有交点，很重要的一点是，我们不需要通过检查 x 坐标来判断这些水平线是否和 L 相交！由于已经知道所有列表中水平线的 x 坐标均与 L 相符，所以只要检查这些列表中的水平线的 y 坐标是否符合 L 的 y 坐标。现在尝试一种归纳假设。

> **归纳假设**：给定上述 k 个 x 坐标的列表（x_k 为最右的 x 坐标），我们知道如何得到 x_k 左方所有线段间的交点，并去除那些位于 x_k 左方的水平线。

称仍然在考虑范畴内的水平线为候选线。（它们是那些左端点位于当前 x 坐标左方，而右端点位于当前 x 坐标之上或其右方的水平线。）我们需要维持包含这些候选线的数据结构，不久将讨论该数据结构的具体实现。

归纳假设的基础情况很容易，为了推广这个假设，需要处理第$(k+1)$个端点，分为以下 3 种情况：

1. 第$(k+1)$个端点是一条水平线的右端点。在这种情况下，只要简单地将这条线从候选线的集合中删除即可。正如先前所说，只有考虑竖直线时才检测交点，删除这条水平线并不会遗漏交点。由此，该步推广了归纳假设。
2. 第$(k+1)$个端点是一条水平线的左端点。在这种情况下，我们将这条线添加入候选线的集合。因为这条线的右端点还未曾遇到，所以这条线不应被删除。根据上述论证，这是推广

归纳假设的正确方法。

3. 第(k+1)个端点是一条竖直线。这是算法的主要部分，通过检查候选线集合中所有水平线的 y 坐标和该竖直线的 y 坐标，可以找到所有涉及该竖直线的交点。

算法描述完毕，比较操作的数目通常远小于mn。但很遗憾，即使实际的交点个数很少，该算法在最坏情况下仍需要 $O(mn)$次比较操作。例如，如果所有水平线均从左边界延伸至右边界，那么必须检查每条竖直线与所有水平线的相交情况，得到一个 $O(mn)$的算法。即使所有竖直线和这些水平线均不相交，这种坏的情况仍然会发生。

为了改进这个算法，我们需要将竖直线的 y 坐标和那些候选线集合中水平线的 y 坐标之间的比较操作数最小化。设当前所考虑的竖直线的 y 坐标为 y_L 和 y_R，候选线集合中水平线的 y 坐标为 $y_1, y_2, \cdots, y_k$，并且假定候选线集合中水平线已经根据其 y 坐标排序（即 $y_1, y_2, \cdots, y_k$依次递增）。通过执行两次二分搜索（对 y_L 和 y_R 各一次），可以找到所有与该竖直线相交的水平线。假设 $y_i < y_L \leqslant y_{i+1} \leqslant y_j \leqslant y_R < y_{j+1}$，与该竖直线相交的水平线恰好就是 $y_{i+1}, y_{i+2}, \cdots, y_j$。我们也可以只对 y_L进行一次二分搜索，然后扫描这些 y 坐标直至找到 y_j。即使原始问题涉及二维，寻找 $y_{i+1}, \cdots, y_j$也是一个一维问题，在给定一维区间（在这个例子中是指从 y_L到 y_R）内搜索数据被称为一维区间查询（one-dimensional range query）。如果数据经过了排序，那么一维区间查询的运行时间正比于搜索时间外加所查询到的条目个数二者的总和。当然，如果每遇到一条竖直线就对水平线进行排序，那么我们将不能承受这样的开销。

回顾一下这些要求，这里所需要的数据结构允许我们高效地插入新元素、删除元素，以及执行一维区间查询。幸运的是，有若干数据结构（如平衡树）可以以每次 $O(n \log n)$的时间（n 是集合中的元素个数）实现插入、删除和搜索操作，并且以正比于查询到条目个数的线性扫描时间来实现一维区间查询操作。图 8.18 给出了这个算法。

算法 ***Intersection*** $((v_1, v_2, \cdots, v_m), (h_1, h_2, \cdots, h_n))$
输入： $v_1, v_2, \cdots, v_m$（竖直线段的集合）
和 $h_1, h_2, \cdots, h_n$（水平线段的集合）
输出： 所有相交线段对的集合
{$y_B(v_i)$($y_T(v_i)$)分别是线段 v_i 的上端点和下端点的纵坐标}

```
begin
    sort all x coordinates in increasing order and place them in Q ;
    V := ∅ ;
        {V 是当前水平相交线段的候选集，按照水平线段的纵坐标
        的值组织为平衡树}
    while Q is not empty do
        remove the first endpoint p from Q ;
        if p is the right endpoint of h_k then
            remove h_k from V
        else if p is the left endpoint of h_k then
                insert h_k into V
            else if p is the x coordinate of a vertical line v_i then
                perform a one-dimensional range query for the range
                    y_B(v_i) to y_T(v_i) in V
end
```

图 8.18　交点算法

复杂度　根据 x 坐标排序需要时间 $O((m+n)\log(m+n))$。由于每次插入和删除操作需要 $O(\log n)$步，所以处理水平线总共需要的运行时间为 $O(n \log n)$。处理竖直线需要一维区间查询，它可以在 $O(\log n+r)$时间内完成，其中 r 是与这条直线相关的交点个数。这样，该算法的运行时间为 $O((m+n)\log(m+n)+R)$，其中 R 是交点的总数。

8.7　小结

从某种意义上说，由于我们习惯于观察和处理几何对象，几何算法看似比图算法更具体直观，但是表象有时容易误导。处理大量对象与观察小规模的图形是不同的，我们必须小心不要被脑海中的图形所误导。还必须处理许多特例，并确保已经全部将它们考虑在内。判断点是否在多边形内部的算法（参见 8.2 节）就是一个典型的例子，通常不会想到如图 8.1 所示的多边形，此外我们也很容易忽略可能产生的特例。因此，在设计几何算法时，一定要特别小心谨慎。

设计（离散）几何算法的技术与先前章节所学的技术很相似。归纳法在其中起了很重要的作用，基于归纳法的扫描线技术在一些几何算法（8.5节给出了其中的一个）中很常见，分治法同样也很常见。几何算法（除了一些简单的以外）倾向于使用复杂的数据结构，由此发展了许多复杂且具有独创性的数据结构，这里并没有涉及任何这类特别的数据结构。

参考文献和补充读物

本章只涉及一小类几何算法的例子。离散计算几何学是一个相对较新的领域，但也存在着跨越数十年的大量文献著作。有若干关于计算几何的专著，Preparata and Shamos[1985]以及Edelsbrunner[1987]都给出了大量相关技术、例子以及综合性的参考文献，其他书籍还有Mehlhorn[1984]和 Toussaint[1984]等。

Chand and Kapur[1970]给出了凸包的礼品包裹算法，而 Graham[1972]则给出了 Graham 算法，Ronse[1987]刊登了包含 268 篇关于凸包算法和其他凸性问题的论文的参考文献。

Shamos and Hoey[1975]给出了寻找最近点对的算法，Hinrichs, Nievergelt, and Schorn[1988]给出了使用扫描线技术的 $O(n \log n)$算法。关于最近点对问题，Rabin[1976]提出了一个期望运行时间为 $O(n)$的概率算法（另见 Fortune and Hopcroft[1979]）。一项解决毗邻问题的通用技术是构造 Voronoi 图解，给定点集合的 Voronoi 图解是指对平面的区域划分，使得每个区域包含距离集合中一点最近的所有点。Voronoi 图解可以在 $O(n \log n)$ 时间内构造出来（见 Shamos and Hoey[1975]），它对一大类毗邻问题都很有用。

Bentley and Ottmann[1979]给出了求竖直线和水平线间交点的算法（在 8.6 节中给出），该算法 $O(n \log n+R)$的运行时间在最坏情况下是最优的（参见 Preparata and Shamos[1985]）。Shamos and Hoey[1976]提出了判定任意线段之间是否存在交点的算法，它同样使用了扫描线技术。通过增强归纳假设，该问题也可通过分治法来解决（Guting and Wood[1984]）。关于交点问题有许多文献著作，读者可以参考以上列举的任何一本书。习题 8.16 出自 Bentley, Faust, and Preparata[1982]，而习题 8.17 和习题 8.18 则在 Preparata and Shamos[1985]中有讨论。

操练型习题

8.1　给出线 L 与多边形顶点相交或者与多边形边重合时所产生的各种特例，从而完成算法 *Point_in_Polygon*_1（见图 8.2）。

8.2 设计一个算法以判定给定平面上的 n 个点是否在一条直线上。

8.3 设 S 是任意平面中的点集合，是否只存在惟一的简单多边形，使得其顶点集合为集合 S？或者证明该惟一性，或者给出有相同顶点集合的两个相异简单多边形。

8.4 本书给出的第一个凸包算法（见 8.4 节）的思想是：计算从一个极端点到其余点凸包的支撑线。假定对那个额外点的惟一所知就是它在其余点凸包的外部，而它可以位于那个凸包的上方、下方或者任何其他地方。我们仍然可以通过计算额外点 q 到其余点的角度来寻找从 q 到那个凸包的支撑线，由于这些角度可以在任何范围内，所以我们不清楚应该如何选择最大角度和最小角度，试寻找一个方法判别从 q 到其余点的连线中哪两条是支撑线。

8.5 设 $p_1, p_2, \cdots, p_n$ 是一个点集合，根据中心在这些点的凸包内部某处的圆，将这些点以循环顺序排列，试修改 Graham 扫描算法使其工作于（没有额外的排序过程）这样的点集合。

8.6 Graham 扫描算法应用于按照某种顺序排序的点集合，可使用算法 *Simple_Polygon*（见图 8.8）将这些点以如下方式排序：从一个“极端”点 p（可以确保它属于凸包）开始，根据点和 p 的连线与一固定直线（例如 x 轴）之间的角度，将其他所有点进行排序。证明（通过举反例）并非所有的点 p 均可达到那个目的，换句话说，给出一个点集合 S 和不在该集合中的另一点 p，使得相对于 p 将那些点（根据 p 到集合中的点的连线所形成的角度）排序后运用 Graham 扫描算法不能正确求得凸包。

8.7 通过例子说明：对于任何一个 p 值，算法 *Graham's_Scan*（见图 8.12）均可能连续拒绝 p 个点（每次拒绝一个点）。（换句话说，循环可以执行 p 步而不改变 m 的值。）

8.8 平面中有不同 x 坐标的 n 个点，对它们运行算法 *Closest_Pair_2*（见图 8.16）需要 $\Omega(n \log n)$ 步，请给出这样的一个例子。

8.9 给定平面中 n 条水平线段，设计一个算法以寻找它们之间的所有交点，该算法在最坏情况下需要 $O(n \log n)$ 的运行时间。

创造型习题

8.10 输入为平面中 n 个点的集合和一条直线，设计一个线性时间算法以寻找一条平行于给定直线的直线，使得后者将给定的点集合平均分成两个大小相等的子集（如果一点位于这条直线上，那么它可以计算在任何一边）。

8.11 设 P 是包围在给定矩形 R 中的一个简单（不一定是凸的）多边形，而 q 是 R 中任意一点，设计一个高效算法寻找连接 q 和 R 外部一点的线段，使得该线段与 P 相交的边的条数最小（可参见 8.2 节）。

8.12 设 P 是一个以顶点数组（按照循环顺序）的形式给定的凸多边形，设计一个判定给定点 q 是否在 P 内部的算法，其最坏情况下的运行时间应为 $O(\log n)$。

8.13 许多凸包算法都基于或者类似于排序算法，给出一个类似于高效插入排序的凸包算法。在每一次迭代过程中，需要考虑一个以上的点，并且可能将其加入当前凸包，后者是指由目前遇见的点所构成的凸包。这些点应以任意顺序考虑（即起初未经过排序），而算法应当基于某种高效的数据结构，从而可以有效地判定一个给定点是否位于给定凸多边形的内部。指出你所给的算法在最坏情况下的运行时间。（不必对所有的特例均给出其全部细节。）

8.14 考虑计算凸包时利用支撑线逐点延拓凸包的思想（参见 8.4 节），设计一个基于该思想的计算凸包的 $O(n \log n)$ 算法。

8.15 假定你有一个黑盒算法，它可以在时间 $O(|P_1|+|P_2|)$ 内计算两个不相交凸多边形 P_1 和 P_2 并集

的凸包（$|P_i|$代表 P_i 中点的个数）。利用这个黑盒算法，设计一个算法以寻找给定平面中 n 个点所组成集合的凸包，并且其运行时间应为 $O(n \log n)$。

8.16 点集合 P 的 d–近似（d-approximate）凸包是指所有顶点都取自 P 的凸多边形，满足所有 P 中的点要么位于该凸多边形内部要么在其距离 d 之内。（这里将点到多边形的距离定义为所有从该点连接到多边形任何一处线段长度的最小值。）设 P 是一个由 n 个点所组成的集合，满足 P 中任意两点间 x 坐标的最大差值为 X。设计一个计算 P 的 d–近似凸包的算法，其运行时间应为 $O(n+X/d)$。

8.17 设 P 是平面中 n 个点的集合，如下定义 P 中一点 p 的深度：为了使 p 成为凸包的顶点而需要“削去”（去除）的外层凸包数目，设计一个 $O(n^2)$ 算法寻找 P 中所有点的深度。（注意，由于所有凸包均可能只含有常数个顶点，所以寻找凸包然后去除它们的最直接算法所需要的运行时间为 $O(n^2 \log n)$。）

8.18 a. 如果平面中点 p 的 x 坐标和 y 坐标均大于或等于另一点 q，则称 p 支配（dominate）q。在给定的点集合 P 中，如果没有其他点能支配点 p，则称 p 为极大点。设计一个 $O(n \log n)$算法寻找给定 n 个点集合 P 中的所有极大点。

b. 试解决三维情况下的相应问题（支配的定义需要推广以包括所有维数）。

8.19 设 S 是平面中的点集合，对于每一点 $p \in S$，定义$D(p)$为 S 中由 p 所支配的点集合（见习题 8.18）。设计一个算法对所有 $p \in S$ 计算集合 $D(p)$的大小，其在最坏情况下的运行时间应为 $O(n \log n)$。

8.20 给定平面中的 n 个点，寻找连线有最大斜率的点对，该算法在最坏情况下的运行时间应为 $O(n \log n)$。

8.21 输入为平面中n个点的集合，它们按照下列方式以链接表数组的形式给出：数组中的每一个条目有 X（x 坐标）和 *Next*（其指向一个非空链表，包含集合中所有 x 坐标等于 X 的点，根据其 y 坐标排序）两个域，数组本身按照 x 坐标排序。设计一个算法以寻找满足“x 坐标相等或者在数组中相邻”条件的最近点对，该算法在最坏情况下的运行时间应为 $O(n)$。在这个算法中是否有必要计算平方根？你的算法所寻找的是否是最近点对（没有任何约束条件）？

8.22 输入为平面中的线段集合，其中所有的线段或者水平，或者竖直，或者与水平方向成 45°，将寻找水平线和竖直线之间所有交点的算法推广到这种情况，要求是不增加最坏情况下的渐近复杂度。

*8.23 使用分治法设计一个算法，用于计算水平线段和竖直线段之间的所有交点，该算法的运行时间应当与 8.6 节中讨论的算法一样，即所有的交点应在时间 $O((m+n)\log(m+n)+R)$内求得（其中 R 是寻找到的交点个数）。

8.24 输入为平面中任意n 条线段所组成的集合，设计一个算法以判定是否有两条线段相交，该算法只需要输出肯定或者否定的答案，其在最坏情况下的运行时间应为 $O(n \log n)$。（提示：使用类似水平线和竖直线场合所用的扫描线方法，但是需要维持不同的信息。）

8.25 网格多边形（grid polygon）是指每条边平行于 x 轴或 y 轴的简单多边形，设计一个高效的算法以计算给定两个网格多边形的交集（即它们的公共区域），多边形按照其顶点的循环顺序给出。

8.26 输入是由数轴上的区间所组成的集合，这些区间由它们的两个端点表示。设计一个算法来识别所有包含在集合中其他区间的区间，该算法在最坏情况下的运行时间应为 $O(n \log n)$。

8.27 输入是由 n 个其边平行于坐标轴的矩形所组成的集合，推广习题 8.26 中得到的算法，使其能标记所有包含在另一个矩形中的矩形，能否得到一个运行时间为 $O(n \log n)$的算法？

8.28 输入是由 n 个其边平行于坐标轴的矩形所组成的集合，设计一个算法来寻找所有这些矩形的交集。

8.29 输入是由平面中n 个圆所组成的集合，设计一个算法以检测该集合中是否有两个圆的交集不为空，算法不需要计算交集，只要输出肯定或者否定的答案即可，其在最坏情况下的运行时间应为 $O(n \log n)$。

8.30 输入是由 n 个多边形所组成的集合，每个多边形均有 k 个顶点，设计一个算法以检测该集合中是否有两个多边形的交集不为空，算法不需要计算交集，只要输出肯定或者否定的答案即可，其在最坏情况下的运行时间是多少？

8.31 输入是由其顶点列表（按照循环顺序）的形式所给出的两个凸多边形，设计一个线性时间算法来计算这两个多边形的交集，输出的凸多边形由其顶点列表（按照循环顺序）的形式表示。

8.32 输入是由其顶点列表（按照循环顺序）的形式所给出的两个凸多边形，设计一个线性时间算法来计算这两个多边形的并集（即至少被其中一个多边形所包围的区域）。

8.33 输入是由 n 个其边平行于坐标轴的矩形所组成的集合，设计一个算法来计算这些矩形的并集。这个并集显然也是一个多边形，它应当由逆时针顺序排列的顶点列表所表示。（该问题是第 5 章中地平线问题的一个推广。）

8.34 输入是由平面中的 n 个三角形所组成的集合，这些三角形由它们的顶点给出，设计一个 $O(n \log n)$ 算法计算它们的交集（即公共区域）。

8.35 输入是由其顶点列表（按照循环顺序）的形式所给出的一个凸多边形，设计一个线性时间算法以寻找 n 个三角形，使得它们的交集为该给定多边形。

8.36 输入是平面中 n 个点的集合，设计一个 $O(n^2 \log n)$算法以判定在该集合中是否有 4 个点，使得后者为一个正方形的顶点。

8.37 输入是平面中 n 个点的集合，设计一个多项式时间算法以判定在该集合中是否有 k 个点（对于某个$k \leqslant n$），使得后者为一个正多边形的顶点。（正多边形是指所有的边和内角均相等的多边形。）

8.38 输入是 n 个坐标均为整数的点所组成的集合，我们想要寻找一个平行线集合，使得每点至少要包含于该集合中的一条直线，这些直线或者平行于坐标轴或者与坐标轴成 45°。设计一个 $O(n \log n)$ 算法以寻找满足这些条件且规模最小的平行线集合。需要强调的是，这些直线必须是平行的；特别是，如果有一条直线与坐标轴成 45°，那么所有这些直线都与坐标轴成 45°。

8.39 直线将平面划分成两个半平面，任何数目的半平面的交集是一个凸多边形（半平面是凸的，而凸对象的交集总是凸的）。问题是计算 n 个给定半平面的交集，其输出是一个凸多边形，即在多边形中以循环顺序出现的顶点的列表，其输入是定义半平面的线性不等式。设计一个计算该交集的 $O(n \log n)$算法。

8.40 输入是平面中一般位置的 $2n$ 个点（即没有 3 个点位于同一直线上），其中 n 个点着红色，另 n 个点着蓝色。设计一个算法将这些蓝点匹配上红点，并满足以下条件：（1）每一个点均有惟一的匹配点；（2）连接匹配两点的线段不能相交，该算法不需要使用任何图匹配技术。这样的匹配一定存在，虽然这一点不太明显。该算法应当运行在多项式时间内。（提示，使用归纳法：尝试寻找一个红点和一个蓝点，使得二者间的连线不会有问题；如果那样不行，则尝试用一条直线将这个点集合划分，把原始问题划分成两个更小的问题。）

第 9 章　代数和数值算法

1 加 1 等于 2；
2 加 2 等于 4；
4 加 4 等于 8；
8 加 8 大于 10。
——儿歌

9.1　引言

做算术运算实际上就是执行某种算法。由于人类通常对这些运算非常熟悉，所以想当然地会运用相应的算法。但是，当涉及很大的数值或者面临很长的数值序列，而欲进行乘法、除法或者更复杂的算术运算时，直接算法就可能不是最佳的。前面章节中遇到的相同现象也将在这里出现：当输入的规模增长时，一些适用于小输入的算法将变得不再高效。

如同前面的章节，可通过算法所执行的“运算”的数目来度量算法的复杂度。在大多数情况下，可假定基本算术运算（如加法、乘法和除法等）花费一个单位的时间。当操作数可以由一个或者两个计算机字（例如，不太大的整数、单精度或者双精度的实数）表示时，上述假定是合理的。但是，当操作数非常巨大（例如是一个2000 位的整数）时，则必须考虑操作数的规模，或者至少要意识到基本运算已不再那么简单。很有可能由于忽略了操作数的规模而设计出一个“在纸面上”看似高效而实际效率却非常低的算法。

“输入的规模”的含义有时令人迷惑，当想要对给定整数 n 进行算术运算时，我们很自然地将 n 的值视为输入的规模。但是，这却有悖于常规输入规模的定义，即输入所需的存储空间。这样的区别是十分重要的，即使通过手工计算，也可以很快将两个 100 位的数相加；而另一方面，即使是使用最快的计算机，也不能在合理的时间内计数至 100 位数所表示的数值。由于数 n 可以通过$\lceil \mathrm{lb}n \rceil$位来表示，所以它的规模可定义为$\lceil \mathrm{lb}n \rceil$。例如，当输入是 n 时，需要 $O(\log n)$次运算的算法（例如，计算 $2n$ 的算法）被认为是线性的，这是因为 $O(\log n)$是输入规模的线性函数；然而，当输入是 n 时，需要 $O(\sqrt{n})$次运算的算法（例如，通过尝试所有小于或等于 $\sqrt{n}$ 的数值来对 n 进行因数分解）被认为是指数的。

如同往常，本章着重于有趣的算法设计技术。首先讨论如何计算给定数的幂。其次给出可能是已知最古老的非平凡算法：寻找最大公约数的欧几里得算法，令人惊讶的是，现代计算机仍使用这个拥有2200 多年历史的算法。接着讨论多项式乘法以及矩阵乘法的算法。本章最后给出了快速傅里叶变换，它是最重要且最漂亮的算法之一。

9.2　求幂运算

先从一个基本的算术运算开始。

问题　给定两个正整数 n 和 k，计算 n^k。

由于 $n^k = n \cdot n^{k-1}$，可以很容易地将这个问题归约到计算 n^{k-1} 的问题上。因此，该问题可以通过对 k 进行归纳解决，图 9.1 给出了其对应的直接算法。由于所减小的是 k 的值，而不是其规模，所以直接算法需要 k 次迭代。又因为 k 的规模是 lb$k$，所以迭代的次数就是 k 规模的指数函数（$k = 2^{\text{lb}k}$）。虽然该算法对于非常小的 k 值并不差，但是对于较大的 k 值来说难以接受。

```
算法 Power (n, k) ; { 初试 }
输入： n 和 k （两个正整数）
输出： P （n^k 的值）
begin
    P := n ;
    for i := 1 to k − 1 do
        P := n*P
end
```

图 9.1 幂算法

对问题进行另一种归约的方法是利用事实 $n^k = (n^{k/2})^2$，基于这一观察，可将问题归约成输入参数为 n 和 $k/2$ 的问题。将 k 的值减小一半对应于将其规模减小一个常数值，故乘法的运算量将与 k 的规模成正比。由此可得算法的概要，该算法被称为重复平方法。最简单的情况是 $k=2^j$：

$$n^k = n^{2^j} = \left\{\left[\left[n^2\right]^2\right]^{\cdots 2}\right\} j\text{次。}$$

但是，如果 k 不为 2 的幂呢？再考虑一下刚才所使用的归约，首先是将两个参数为 n 和 k 的问题归约到参数为 n 和 $k/2$ 的更小问题。由于 $k/2$ 可能不是整数，所以这步归约并不总那么有效。如果 $k/2$ 不是整数，那么归约后的问题将不满足原始问题的条件。但是，如果 $k/2$ 不是整数，那么 $(k-1)/2$ 就一定是整数，所以适用于以下归约：

$$n^k = n\left[n^{(k-1)/2}\right]^2。$$

现在得到算法：如果 k 是偶数，则简单地对参数为 $k/2$ 的问题的解进行平方；如果 k 是奇数，则将参数为 $(k-1)/2$ 的问题的解平方后再乘以 n。所以，乘法的运算量至多为 2lb$k$。图 9.2 给出了该算法。

```
算法 Power_by_Repeated_Squaring(n, k)
输入： n 和 k （两个正数）
输出： P （n^k 的值）
begin
    if k = 1 then P := n
    else
        z := Power_by_Repeated_Squaring (n, k div 2) ;
        if k mod 2 = 0 then
            P := z*z
        else
            P := n*z*z
end
```

图 9.2 反复平方的幂算法

复杂度 乘法运算数为 $O(\log k)$。但是，数值将算法的进行变得越来越大，因而乘法运算的开

销也将不断变大。读者可以分析在更现实的度量下（即考虑乘法运算的开销增长）该算法的复杂度（见习题 9.12）。下面给出算法的一个应用场合，在那里数值并不会随算法的执行而变大。

密码学中的应用

对密码学的研究超出了本书的范畴，这里只做简要介绍。加密机制通常依赖于完备的保密，任何两个想要交换秘密信息的参与者都必须就加密-解密算法达成一致，同时他们还必须使用仅自身所知的密钥。想要避免每对参与者之间进行密钥交换，于是就有了下述 RSA 公钥加密机制（命名来自该机制的提出者，见 Rivest, Shamir, and Adleman[1978]），一组参与者（例如计算机用户）可以使用该机制实现加密消息的通信。每个参与者仅有两个密钥，其一用于加密，另一个则用于解密（与其他参与者的人数无关）。密钥按照如下方式进行：RSA 机制的参与者 P 选取两个很大的素数 p 和 q 并计算它们的乘积 $n=pq$，然后选择另一个很大的整数 d，使得 d 和 $(p-1)(q-1)$ 没有公因数。（可参见下一节中验证该事实的算法；如果 d 是一个随机数，上述情况很可能发生。）根据 p、q 和 d，可以（虽然并不容易）计算出满足以下条件的数 e：

$$e \cdot d \equiv 1 \pmod{(p-1)(q-1)}。 \tag{9.1}$$

后面将会看到，e 和 d 可以分别作为加密密钥和解密密钥，而 P 可以在中央名录（central directory）公布数值 n 和 e 以便每个人都可以检索。（假定这是一个可信任的名录，在那里其他人无法伪造 P 的密钥。）P 将 d 的值以及 p 和 q 的值予以保密，但 p 和 q 的值将不再需要。

设 M 是一个整数，其对应于 P 想要加密的消息（因为每个消息可以转换成比特序列，而后者又可以进一步转成整数）。可以假定 M 小于 n；如若不然，那么 M 可被分解成若干对应整数小于 n 的短消息。P 所使用的加密函数 E_P 非常简单：

$$E_P(M) = M^e \pmod{n}。$$

由于 n 和 e 都是公开的，每个人都可以加密消息，然后将它们发送给 P。解密函数 D_p 同样简单（但由于 d 的值是保密的，所以只有 P 能够执行解密函数）：

$$D_P(C) = C^d \pmod{n}。$$

可以证明式（9.1）确保了 $D_P(E_P(M)) = M$，所以它们是有效的加密和解密函数。这两个算法仅有一次求幂运算（M^e 或 C^d）与一次除法运算（计算同余），但这些运算涉及很大的数值。模 n 运算可以运用于算法的任何一步，并不一定要在最后进行，这是因为对所有的整数 x、y 和 n，均有

$$x \cdot y \pmod{n} = [x \pmod{n} \cdot y \pmod{n}] \pmod{n},$$

每步计算中的模 n 运算非常重要，操作数的值只有以这种方式才不会增长超过 n。如果使用图 9.2 中的反复平方的幂算法，在计算幂时不但只需要 $O(\log e)$（或 $O(\log d)$）次乘法和除法运算，而且每次乘法和除法运算所涉及的数值都小于 n。对于反复平方的幂算法，我们只需要将其中的每次乘法运算修改为乘法模 n 运算即可。这样，RSA 机制的应用只需要对小于 n 的数进行 $O(\log n)$ 次乘法和除法运算。

由于尚不知是否存在一个算法可以在合理的时间内（例如人的一生）对一个非常大的数（例如 1000 位数）进行因数分解，所以知道 n 的值并不能推得 p 和 q 的值。通常，人们认为（虽然该事实并没有任何已知的证明）在未知 d、p 或 q 的前提下不可能高效地计算出函数 D_P[①]。因此，

① 基于从 n 和 e 计算 d 的算法，人们已经知道能构造出将 n 因数分解的高效概率算法，而这点充分证明了 d 不能被公开（参见 Bach, Miller, and Shallit[1986]）。但是，也可能存在另一种潜在的方法，其可以在未知 d 的情况下计算出 D_P。

只要 P 将 d、p 和 q 保密，就可以在不泄密的前提下从任何人处收到加密的消息。此外，被称为公钥密码系统（public-key cryptosystem）的机制还有其他一些优点。

9.3 欧几里得算法

两个正整数 m 和 n 的最大公约数（GCD，greatest common divisor）是指满足以下两个条件的惟一正整数 k：（1）n 和 m 都能被 k 整除；（2）能同时整除 m 和 n 的其他整数均小于 k，一般用 GCD(m, n)来表示 m 和 n 的最大公约数。

问题 寻找两个给定正整数 m 和 n 的最大公约数。

先尝试将该问题归约到一个规模更小的问题，即能否在不改变该问题的前提下减小 m 或者 n？欧几里得找到了一个十分明显的肯定答案：如果 k 能同时整除 m 和 n，那么它能整除后两者的差！如果 $n > m$，那么 GCD(n, m) = GCD(n–m, m)，由此可得一个更小的问题。但是，再次减小的仍然是问题中数的值，而不是规模。为了得到高效的算法，我们必须尝试减小问题的规模。例如，如果 n 非常大（如 1000 位的数）而 $m = 24$，我们需要重复从 n 中减去 24 的过程大约 n/24 次。该计算过程需要花费 $O(n)$步，而这关于 n 的规模是指数的。

再考察一下算法。从 n 中减去 m，然后将算法应用到 n–m 和 m，如果 n–m 仍然大于 m，则再次减去 m。换句话说，可重复地从 n 中减去 m 直至结果小于 m。但这恰好等价于 n 除以 m 并得到余数的过程，而除法运算可以很快完成。这就直接得到了欧几里得算法，并在图 9.3 中说明。

算法 ***GCD***(m, n)
输入： m 和 n（两个正数）
输出： gcd（m 和 n 的最大公约数）

```
begin
    a := max( n, m ) ;
    b := min( n, m ) ;
    r := 1 ;
    while r > 0 do {r 是余数}
        r := a mod b ;
        a := b ;
        b := r ;
    gcd := a
end
```

图 9.3 最大公约数算法

复杂度 说明欧几里得算法的运行时间关于 m+n 的规模是线性的；更确切地说，它的运行时间（每步运算的花费与操作数的规模无关，均计为一步）是 $O(\log(n+m))$。为了证明这个结论，只需要说明 a 的值经过常数次迭代就会减半。让我们考察算法 GCD 的连续两次迭代：在第一次迭代中，a 和 b（$a > b$）分别转变为 b 和 a mod b；然后在下一次迭代中，它们又分别转变为 a mod b 和 b mod（a mod b）。所以，在两次迭代中第一个数 a 转变为 a mod b。但是由于 $a > b$，可得 a mod b < a/2，从而确立了该结论。

9.4　多项式乘法

设 $P=\sum_{i=0}^{n-1} p_i x^i$ 和 $Q=\sum_{i=0}^{n-1} q_i x^i$ 是两个 $n-1$ 次多项式。此处，多项式由其系数的有序列表来表示。

> **问题**　计算两个给定 $n-1$ 次多项式的乘积。

$$PQ=\left(p_{n-1}x^{n-1}+\cdots+p_0\right)\left(q_{n-1}x^{n-1}+\cdots+q_0\right)= \tag{9.2}$$

$$p_{n-1}q_{n-1}x^{2n-2}+\cdots+\left(p_{n-1}q_{i+1}+p_{n-2}q_{i+2}+\cdots+p_{i+1}q_{n-1}\right)x^{n+i}+\cdots+p_0q_0。$$

可以直接根据式（9.2）计算 PQ 的系数，我们很容易发现，如果按照式（9.2），乘法和加法的运算量将会是 $O(n^2)$。能否做得更好呢？至今已经见过许多对直接平方算法的改进，所以答案自然是肯定的。在 9.6 节中将会讨论一个复杂的 $O(n\log n)$算法，但首先让我们描述一个简单的分治算法。

为简单起见，这里假定 n 是 2 的幂。我们将每个多项式划分成大小相等的两部分，设 $P=P_1+x^{n/2}P_2$ 而 $Q=Q_1+x^{n/2}Q_2$，此处

$$P_1=p_0+p_1x+\cdots+p_{n/2-1}x^{n/2-1},\quad P_2=p_{n/2}+p_{n/2+1}x+\cdots+p_{n-1}x^{n/2-1},$$

而

$$Q_1=q_0+q_1x+\cdots+q_{n/2-1}x^{n/2-1},\quad Q_2=q_{n/2}+q_{n/2+1}x+\cdots+q_{n-1}x^{n/2-1}。$$

现在可得：

$$PQ=(P_1+P_2x^{n/2})(Q_1+Q_2x^{n/2})=P_1Q_1+(P_1Q_2+P_2Q_1)x^{n/2}+P_2Q_2x^n。$$

此时，PQ 的表达式只涉及计算 $n/2$ 次多项式的乘积。我们可以根据归纳法计算更小规模多项式的乘积（例如，P_1Q_1），然后将结果相加来得到完整的解。那么能否直接利用归纳法呢？惟一的约束条件是更小规模的问题要与原始问题相同，以及知道如何计算 1 次多项式的乘积，上述两个条件显然都能满足。该算法总的运算量 $T(n)$由下列递归关系给出：

$$T(n)=4T(n/2)+O(n),\quad T(1)=1。$$

其中，因子 4 是指 4 个更小规模多项式的乘积，而 $O(n)$为更小规模多项式加法运算的开销。该递归关系的解是 $O(n^2)$（参见 3.5.2 节），而这意味着我们未能实现任何改进（见习题 9.4）。

为了改进该平方算法，我们要减少在解决该问题时所涉及到的子问题个数。考察以下乘法运算表（对于如此简单的符号，这里却使用了非常精细的表格，其目的到了下一节就会变得明确）。

×	P_1	P_2
Q_1	A	B
Q_2	C	D

要计算的是 $A+(B+C)x^{n/2}+Dx^n$，通过观察可以发现，不必分别计算 B 和 C；只需要得到它们的总和即可。如果我们计算乘积 $E=(P_1+P_2)(Q_1+Q_2)$，那么 $B+C=E-A-D$。因此，我们只要计算 3 个更小规模多项式的乘积：即 A、D 和 E。所有其余操作都可通过加法和减法运算进行计算，而这只构成递归关系中的 $O(n)$项，所以新的递归关系如下所示：

$$T(n)=3T(n/2)+O(n)。$$

从而推得 $T(n)=O(n^{\lg 3})=O(n^{1.59})$。

注意多项式(P_1+P_2)和(Q_1+Q_2)与原始多项式 P 和 Q 有着看似奇异的联系，即，将后者次数相差 $n/2$ 的系数进行错位求和从而得到前者。虽然这是以非常不直观的方式将多项式相乘的，但该算法对于较大的 n 的确显著地减少了运算量。

□ 例 9.1

设 $P=1-x+2x^2-x^3$ 而 $Q=2+x-x^2+2x^3$，使用分治算法计算它们的乘积，我们仅执行一次递归操作。首先计算 A、D 和 E：

$$A=(1-x)\cdot(2+x)=2-x-x^2,$$

$$D=(2-x)\cdot(-1+2x)=-2+5x-2x^2,$$

$$E=(3-2x)\cdot(1+3x)=3+7x-6x^2。$$

从 A、D 和 E 可以很容易计算 $B+C=E-A-D$：

$$B+C=3+3x-3x^2。$$

现在根据 $P\cdot Q=A+(B+C)x^{n/2}+Dx^n$，可得

$$\begin{aligned}P\cdot Q&=2-x-x^2+3x^2+3x^3-3x^4-2x^4+5x^5-2x^6\\&=2-x+2x^2+3x^3-5x^4+5x^5-2x^6。\end{aligned}$$

注意到相比于直接算法中的 16 次乘法运算，此处只使用了 12 次；而相比于直接算法中的 9 次加减法运算，此处则使用了 12 次。（如果再多进行一次递归操作，就可进一步将乘法运算减至 9 次。）当 n 的值很大时，算法所节省的时间当然会多得多。（加减法的运算量维持在直接算法的常数因子范围内，而乘法运算量大约可以减少 $n^{0.4}$。）□

9.5 矩阵乘法

两个 $n\times n$ 矩阵 A 和 B 的乘积 C 按照以下方式定义：

$$c_{ij}=\sum_{k=1}^{n}a_{ik}\cdot b_{kj}。\tag{9.3}$$

问题 计算两个 $n\times n$ 实数矩阵 A 和 B 的乘积矩阵 $C=A\times B$。

计算矩阵乘积的直接方法（且看似惟一的方法）可以按照式（9.3），这样需要使用 n^3 次乘法运算和$(n-1)n^2$ 次加法运算。注意这里的 n 代表的是矩阵行和列的个数，而不是输入的规模 n^2。为了说明存在改进的可能性，下面给出两种不同的方案。

9.5.1 Winograd 算法

为简单起见，假定 n 是偶数。记

$$A_i=\sum_{k=1}^{n/2}a_{i,2k-1}\cdot a_{i,2k},\quad B_j=\sum_{k=1}^{n/2}b_{2k-1,j}\cdot b_{2k,j}。$$

重新整理各项之后，得到

$$c_{i,j}=\sum_{k=1}^{n/2}(a_{i,2k-1}+b_{2k,j})\cdot(a_{i,2k}+b_{2k-1,j})-A_i-B_j。$$

由于对于每一行或者每一列，A_i 和 B_j 只需要各计算一次，故计算所有的 A_i 和 B_j 只需 n^2 次乘法运算。因此，总共的乘法运算量减少至 $1/2n^3+n^2$，而加法运算量则大约增加了 $1/2n^3$。从而在加法运算快于乘法运算的情况下，该算法要优于直接算法。

评述　该算法说明，重新安排计算的顺序可能有较好的效果，它甚至适用于诸如矩阵乘法等形式简单的表达式，这个思想在下一个算法中将得到进一步的推广。

9.5.2　Strassen 算法

类似于 9.4 节中所介绍的多项式乘法算法，这里也使用分治法。为简单起见，假定 n 是 2 的幂。设

$$A=\begin{bmatrix}A_{1,1} & A_{1,2}\\ A_{2,1} & A_{2,2}\end{bmatrix},\quad B=\begin{bmatrix}B_{1,1} & B_{1,2}\\ B_{2,1} & B_{2,2}\end{bmatrix},\quad C=\begin{bmatrix}C_{1,1} & C_{1,2}\\ C_{2,1} & C_{2,2}\end{bmatrix},$$

这里 $A_{i,j}$、$B_{i,j}$ 和 $C_{i,j}$ 均为 $n/2 \times n/2$ 的矩阵。使用分治法，可以将该问题归约为根据 $A_{i,j}$ 和 $B_{i,j}$ 计算 $C_{i,j}$；也就是说，可以视 $n/2 \times n/2$ 的子矩阵为元素，而将整个问题视为计算两个 2×2 矩阵的乘积。（在将子矩阵替换为元素时，必须非常小心；我们将其作为习题 9.23 的研究主题。）把每一次元素的乘积替换为一次递归调用，由此，可以将计算 2×2 矩阵乘积的算法转换为计算 $n \times n$ 矩阵乘积的算法。计算两个 2×2 矩阵乘积的常规算法要使用 8 次乘法运算，故我们将每次乘法运算替换为一次递归调用，从而可以得到以下递归关系：$T(n) = 8T(n/2)+O(n^2)$，进而推得 $T(n)=O(n^{\mathrm{lb}8})=O(n^3)$。由于正在使用的是常规算法，所以这一点并不令人惊讶。如果能用小于 8 次的乘法运算计算两个 2×2 矩阵的乘积，则可以得到一个渐近复杂度优于立方的算法。

递归中最重要的部分在于计算两个 2×2 矩阵乘积所需的乘法运算次数，这是因为加法运算量总是构成递归关系中的 $O(n^2)$项（这并不是决定渐近复杂度的因素），它不如乘法运算那般重要。（但是，加法运算量确实影响常数因子。）Strassen 发现将两个 2×2 矩阵相乘只需要使用 7 次乘法运算。这里概括给出 Strassen 发现该算法所可能使用的方法，而不是简单地写下 Strassen 算法中的等式。对于类似问题也可使用该方法。

计算乘积

$$\begin{bmatrix}a & b\\ c & d\end{bmatrix}\begin{bmatrix}e & g\\ f & h\end{bmatrix}=\begin{bmatrix}p & s\\ r & t\end{bmatrix}$$

等价于计算乘积

$$\begin{bmatrix}a & b & 0 & 0\\ c & d & 0 & 0\\ 0 & 0 & a & b\\ 0 & 0 & c & d\end{bmatrix}\begin{bmatrix}e\\ f\\ g\\ h\end{bmatrix}=\begin{bmatrix}p\\ r\\ s\\ t\end{bmatrix}。\tag{9.4}$$

将式（9.4）简记为 $A \cdot X = Y$，寻找计算 Y 时所需的最少乘法运算次数。让我们先寻找容易计算的特殊矩阵乘积，事实上，这样的特殊乘积有 4 种类型（后两种非常相似），它们分别如下所示：

类型	乘积	乘法运算次数
α)	$\begin{bmatrix}a & a\\ a & a\end{bmatrix}\begin{bmatrix}e\\ f\end{bmatrix}=\begin{bmatrix}a(e+f)\\ a(e+f)\end{bmatrix}$	1

$$\beta)\quad \begin{bmatrix} a & a \\ -a & -a \end{bmatrix}\begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} a(e+f) \\ -a(e+f) \end{bmatrix} \qquad 1$$

$$\gamma)\quad \begin{bmatrix} a & 0 \\ a-b & b \end{bmatrix}\begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} ae \\ ae+b(f-e) \end{bmatrix} \qquad 2$$

$$\delta)\quad \begin{bmatrix} a & b-a \\ 0 & b \end{bmatrix}\begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} a(e-f)+bf \\ bf \end{bmatrix} \qquad 2$$

现设法将式（9.4）中所给出的一般矩阵乘积分解成上述若干类型。由于这些类型所使用的乘法运算次数少于名义上的，所以最终可以节省一些运算。需要进行反复尝试才能得到正确的组合，该过程并不直接而且十分晦涩，但它也不是匪夷所思的。设

$$B = \begin{bmatrix} b & b & 0 & 0 \\ b & b & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},\quad C = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & c & c \\ 0 & 0 & c & c \end{bmatrix},$$

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ c-b & 0 & 0 & c-b \\ b-c & 0 & 0 & b-c \\ 0 & 0 & 0 & 0 \end{bmatrix},\quad E = \begin{bmatrix} a-b & 0 & 0 & 0 \\ 0 & d-b & 0 & b-c \\ c-b & 0 & a-c & 0 \\ 0 & 0 & 0 & d-c \end{bmatrix}。$$

这样，$A=(B+C+D+E)$，从而 $AX=BX+CX+DX+EX$。除了 EX，上述每一个乘积都可使用类型 α 或 β，从而只用一次乘法运算就可得出结果。现在惟一的问题是计算 EX，而 E 又可再次分解成两个矩阵 $E=F+G$，使得 F 和 G 可分别使用类型 γ 和 δ：

$$F = \begin{bmatrix} a-b & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ c-b & 0 & a-c & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}\quad G = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & d-b & 0 & b-c \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d-c \end{bmatrix}$$

所以，总的来说，$AX=(B+C+D+F+G)X$。类型 α 需要 2 次乘法运算，而类型 β、γ 和 δ 各需要一次，所以总共需要 7 次乘法运算（另见习题 9.10）。

复杂度 对于原始矩阵维数一半的矩阵计算了 7 次矩阵乘积，外加常数次的矩阵加法运算。由于可以在 $O(n^2)$时间（基本上是矩阵规模的线性时间）内完成两个 $n\times n$ 矩阵的加法运算，所以加法运算不如乘法运算那般重要。递归关系 $T(n)=7T(n/2)+O(n^2)$中的 $O(n^2)$项并不是决定因素，因此该递归关系的解为 $T(n)=O(n^{\mathrm{lb}7})$，约等于 $O(n^{2.81})$。如果使用上述推导，则需要 18 次加法运算（见习题 9.10），甚至可以将加法运算的数目减少至 15 次（见 Winograd[1973]），但这样的约简并不影响渐近运行时间。

评述 Strassen 算法有 3 个主要缺点：

1. 根据经验研究发现，n 至少需要为 100 才能使 Strassen 算法快于 $O(n^3)$ 的直接算法（见 Cohen and Roth[1976]）。
2. Strassen 算法不如直接算法稳定，即对于相同的输入误差，Strassen 算法很可能产生更大的输出误差。
3. Strassen 算法显然比直接算法复杂得多并且更难实现，此外，Strassen 算法不像常规算法那样容易并行化。

尽管如此，Strassen 算法仍然是非常重要的，如对于较大的 n，它比常规算法快，而且还可以用来解决其他一些涉及矩阵的问题，如矩阵求逆和行列式计算等。第 10 章中还会遇到一些问题，而它们均等价于矩阵乘法。在实践上，Strassen 算法可以通过下列方式加以改进，即只将它应用于大矩阵，当矩阵的规模小于 100 左右时停止递归操作，这一点类似于 6.4.4 节和 6.11.3 节中所讨论的慎重选取归纳基础的思想。Strassen 算法为其他算法奠定了基础，同时就一些类似看似不可解的问题提出了许多新的突破点。

9.5.3　布尔矩阵

本节将考察计算两个 $n \times n$ 布尔矩阵的乘积，即所有元素均为 0 或 1 而和与积按照下列规则定义（分别对应于“or”和“and”）：

+	0	1
0	0	1
1	1	1

×	0	1
0	0	0
1	0	1

这些和与积的定义当然有别于通常整数的和与积；因此，为整数所设计的算法通常不能用于布尔类型。布尔和定义中产生的一个问题是减法运算将不再是良定义的（0+1 和 1+1 都定义为 1；因此 1–1 既可以是 1，也可以是 0），然而 Strassen 算法却需要减法运算，所以它不适用于布尔矩阵。但是，有一个技巧可以帮助我们使用 Strassen 算法：即将每一位视为一个模$(n+1)$的整数（这里 n 是矩阵的规模），然后使用整数的加法和乘法运算规则。例如，如果 $n = 4$，则有 1+1 = 2、1+1+1 = 3 以及 1+1+1+1+1 = 0。事实上，如果根据这些规则计算矩阵的乘积，并将最终结果中的非 0 项用 1 替换，那么就能得到布尔积。这是因为计算结果不会超过数$(n+1)$而“溢出”（证明略去）。（更确切地说，模 k 的整数全体构成了一个环，后者是指加法和乘法运算满足某些性质的代数结构；Strassen 算法可以应用于任何环；更多细节可参见 Aho, Hopcroft, and Ullman[1974]。）因此，布尔矩阵乘法的复杂度也为 $O(n^{2.81})$。但是，Strassen 算法需要的是整数运算而非布尔运算。下面给出了两个布尔矩阵乘法的算法，它们利用布尔运算的性质改进了运行时间。在大多数计算布尔矩阵乘积的场合下，这两个算法比 Strassen 算法更实用。

由于布尔操作数只需要 1 位的存储空间，所以可以将 k 个操作数存储在一个大小为 k 的计算机字中。特别是，由于假定 n 存储在一个计算机字中，所以当 $k \leqslant \text{lb}n$ 时，可以在一个字中存储 k 位。根据式（9.3），矩阵乘法的常规算法由 n^2 个“行乘以列”的乘积（或**内积**）所组成，其中第 ij 个内积计算 $\sum_{m=1}^{n} a_{im} \cdot b_{mj}$。为简单起见，假定 k 可以整除 n，因而可将每个内积分解成 n/k 个 k 维布尔向量的内积的总和。相比于将两个 k 位的整数相乘，计算两个 k 维布尔向量的内积更容易。假定 k 位整数的乘法运算花费一个单位时间；这样，对于两个 k 维布尔向量的内积运算，假定其花费一个单位时间同样也是合理的。（例如，内积可以分两步进行：首先计算两个向量的与，然后检查结果是否都为 0。）尽管如此，通常不能让算法依赖于任何与计算机原语相关的假定（除了

4 个基本算术运算）。下一步将说明怎样避免使用这些假定，接着将该思想和另外一个思想结合起来，从而进一步改进布尔矩阵的乘法运算，这两个思想都阐释了有趣的算法设计技术。

第一个思想是预先计算所有可能的 k 维布尔内积，由于其中涉及两个 k 维布尔向量，所以共有 2^{2k} 种可能的内积。我们可以在 $O(k2^{2k})$ 时间内全部计算出这些内积（实际上可以做得更好，参见习题 9.24），并将其存储在规模为 $2^k \times 2^k$ 位的二维表中。两个向量 a 和 b 的内积存储在条目 (i_a, i_b) 中，这里 i_a 和 i_b 分别指 a 和 b 中的 k 位所代表的整数。由于 a 与 i_a（或者 b 与 i_b）实际上是以完全相同的方式所表示的，因而此后将不再区分它们。这样，给定两个 k 维布尔向量，可以通过简单检索该表来计算它们的内积。如果可以在 $O(1)$时间内访问规模为 2^{2k} 的表，那么每个 k 维内积就可以在常数时间内计算得出（在表构造之后）。例如，设 $k = \lfloor \text{lb}n/2 \rfloor$，表的规模此时为 $O(n)$，构造该表本身需要 $O(n\log n)$时间。假定可以在常数时间内访问规模为 $O(n)$的表，这一点并非不寻常，先前已经多次（隐含地）使用过这个假定。通常假定：如果 n 是输入的规模，那么可以在一个计算机字（或常数个计算机字）中存储 $\text{lb}n$ 位的数。一旦构造了该表，就可以在时间 $O(n/k) = O(n/\log n)$ 内计算 n 维布尔内积。注意该表所依赖的是 k 的值，而非矩阵。因此，计算两个布尔矩阵的乘积可以在时间 $O(n^3/\log n)$以及额外空间$O(n)$内完成。也可以选取 k 为$\lfloor \text{lb}n \rfloor$，此时表的规模为 $O(n^2)$，而在乘法算法中可以节省额外的因子 2。而如果可以承受大小为 $O(n^2)$的额外空间，还可以找到一个更快的算法。

考虑两个 $n \times n$ 的布尔矩阵 A 和 B，矩阵乘法通常按照式（9.3）定义：进行 n^2 次内积运算，其中每一次运算均涉及 A 的一行与 B 的一列。也可以通过如下“A 列乘以 B 行”的方式计算这两个矩阵的乘积：记 A 的第 r 列为 $A_C[r]$，而 B 的第 r 行为 $B_R[r]$；$A_C[r]$可以视为一个 $n \times 1$ 矩阵，而 $B_R[r]$可以视为一个 $1 \times n$ 矩阵。$A_C[r]$和 $B_R[r]$的乘积是一个 $n \times n$ 矩阵，后者的第 ij 个元素是 $A_C[r]$ 的第 i 个元素与 $B_R[r]$的第 j 个元素的乘积（见图 9.4）。显然有

$$A \cdot B = \sum_{r=1}^{n} A_C[r] \cdot B_R[r]。 \tag{9.5}$$

由于所进行的加法和乘法运算相同，只是这些运算的执行顺序有所不同，因而表达式（9.5）等价于式（9.3）。

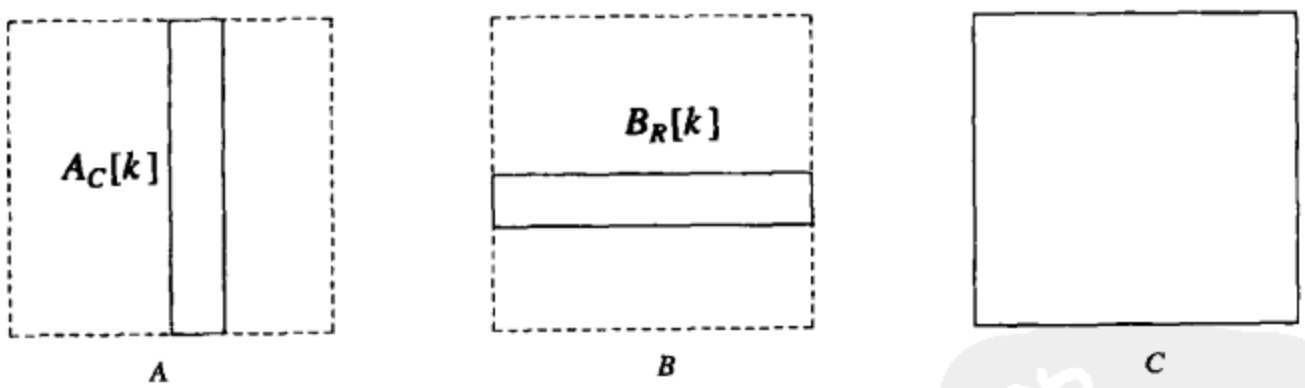


图 9.4 按照“列乘以行”的方式将矩阵相乘

现在将 A 的列、B 的行划分成 n/k 个大小相等的组。（为简单起见，假定 n/k 是整数；否则会有一个额外的规模较小的组。）换句话说，将 A 分解成 $A_1, A_2, \cdots, A_{n/k}$，使得每一个 A_i 构成大小为 $n \times k$ 的矩阵；同理，将 B 分解成 $B_1, B_2, \cdots, B_{n/k}$，使得每一个 B_i 构成大小为 $k \times n$ 的矩阵。显然有

$$A \cdot B = \sum_{i=1}^{n/k} A_i \cdot B_i。 \tag{9.6}$$

现在的问题是如何高效地计算 $C_i = A_i \cdot B_i$，下面通过一个例子来描述这步计算（见图 9.5）。

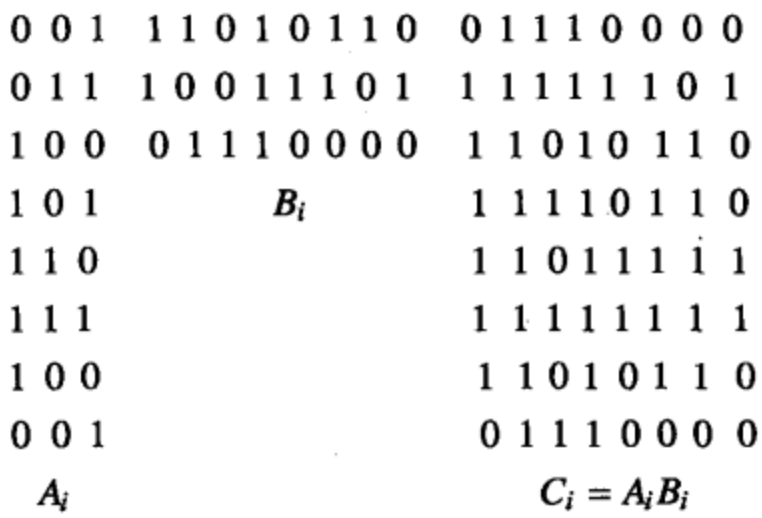


图 9.5　布尔矩阵乘法

由于 A_i 的第 1 行只在第 3 列有一个 1，所以 C_i 的第 1 行恰好与 B_i 的第 3 行完全相同；类似可得，C_i 的第 2 行是 B_i 的第 2 行与第 3 行的布尔和。显然，C_i 的第 j 行是若干 B_i 行的布尔和（根据 A_i 的第 j 行）。这里并不直接计算 C_i 的每一行，而是使用类似于前述算法中所用的方法，即预先计算所有的可能性。由于 A_i 的每行中有 k 个元素，所以 B_i 行的各种可能组合一共有 2^k 种。令 $k = \text{lb}n$ 并再次假定 k 是一个整数，预先计算所有 $2^k = 2^{\text{lb}n} = n$ 种组合，然后将结果存储在一个表中。相比于前面的算法，该表包含 n 行而不是 n 位；因此，存储空间需要 $O(n^2)$。此外，由于该表还依赖于 B_i，所以必须为每一个 B_i 均建立起与之对应的表。为了寻找 C_i 的第 j 行，检索 A_i 的第 j 行并找到需要相加的 B_i 行组合。A_i 的第 j 行的二进制表示所对应的整数可以代表这样的组合（例如，图 9.5 中的 A_i 第 1 行对应于 1，第 2 行对应于 3，第 3 行对应于 4 等），这个整数是 C_i 第 j 行在表中的存储地址。在表中寻找到 C_i 中的一行需要花费 $O(1)$ 时间，而将这一行复制到 C_i 中合适的位置需要花费 $O(n)$ 时间。综上所述，可以在时间 $O(n^2)$ 内计算 C_i。

现在说明可以在时间 $O(n \cdot 2^k)$ 内计算 B_i 行之和的各种可能组合，其中每种行的组合对应于一个 k 位的整数。根据归纳法，假定知道如何计算对应整数小于 i 的行之和组合。首先，计算对应于 0 的行之和是显然可行的。假定 i–1 的二进制表示为 xxxx011111（即最低位的 0 后继 j 个 1），那么对应于 i 的行之和等于对应于 xxxx000000 的行之和加上对应于 0000100000 的行之和。由于 xxxx000000 小于 i，根据归纳假设可以得到其所对应的行之和，现在只需要再添加一行即可，添加一行需要进行 n 次布尔加法运算而我们总共有 2^k 种组合。因此，所有的预先计算均可通过 $O(n \cdot 2^k)$ 次运算完成。若 $k = \text{lb}n$，则运行时间为 $O(n^2)$。该算法以其发明者的国籍和人数命名，叫做 4 个俄国人算法（four-Russians algorithm，Arlazarov et al.[1970]）。图 9.6 给出了该算法。

复杂度　为了计算 $A \cdot B$，必须计算 n/k 次 $A_i \cdot B_i$ 的乘积。由于每一个这样的乘积要花费 $O(n^2)$ 时间外加构造表所花费的 $O(n \cdot 2^k)$，该算法总共的运行时间为 $O(n^3/k + n^2 \cdot 2^k/k)$。如果 $k = \text{lb}n$，那么运行时间即为 $O(n^3/\log n)$。

```
算法 Boolean_Matrix_Multiplication(A, B, n, k)
输入：A、B（两个 n×n 的布尔矩阵）以及 k（整数）
输出：C（A 和 B 的积）
    {为简单起见，假定 k 能够整除 n}
begin
   Initialize the matrix C to 0 ;
   for i := 0 to n/k − 1 do
       Construct Table_i ;
       {Table_i 是一个由 2^k 个大小为 n 的布尔向量组成的数组，
       它包含 B_i 的 k 行之和的所有可能的组合，参看正文}
       m := i * k ;
       for j := 1 to n do
           Let Addr be the k-bit number
               A[j, m+1]A[j, m+2] ··· A[j, m+k] ;
           add Table_i[Addr] to row j in C
end
```

图 9.6 布尔矩阵乘法算法

下面就来说明如何将第一个算法的思想和第二个算法的思想结合在一起，从而改进运行时间，使之进一步减少 $O(\log n)$因子。算法 *Boolean_Matrix_Multiplication*（见图 9.6）的主要步骤是将表中的一行加至 C，通过使用相同的技巧——预先计算所有可能的加法运算，可以在 $O(n/m)$ 时间内完成加法运算。（如果布尔加法可通过快速执行的原语进行运算，则不需要这一步；但是，算法本身并不能依赖于这些假设。）首先构造一个大小为 $2^m \times 2^m$ 的二维表 *Add_Table*，该表包括两个 m 维布尔向量所有可能的加法运算结果。换句话说，表 *Add_Table* 中的条目(i, j)是 i 与 j 的布尔和。（再次强调，i 和 j 可以同时视为整数与布尔向量。）显然，表 *Add_Table* 的构造可以在 $O(m \cdot 2^{2m})$ 时间和空间内完成。注意与算法 *Boolean_Matrix_Multiplication* 中使用的表所不同的是，表 *Add_Table* 独立于 A 和 B；其仅依赖于 m 的值。现在将 B_i 的每一行分成 n/m 组，每组大小为 m（为简单起见，我们再次假定 m 整除 n）。将每一组视为 m 位的整数；这样，B_i 的每一行均可由一个 n/m 元整数元组所代表，故算法的所有步骤都能够以这些元组为单位进行。

为了实现两个 n 维向量的加法运算，使用表 *Add_Table* 在 n/m 步之内将两个n/m 元组相加。其中，每一步包括两部分：读取两个 m 位的数以及从表 *Add_Table*（其包含两数的布尔和）中得到相应的条目。只要计算机字的大小不超过 $2m$，该步骤就可以在常数时间内完成。在常规 4 个俄国人算法中构造表的过程中，以及在整个算法执行时将行相加的过程中都使用这样的技巧。如果选取 m 大约等于$\lfloor \mathrm{lb}\, n/2 \rfloor$，那么 $2^{2m} = O(n)$，并且由于假定 n 可由一个计算机字表示，所以可以用一个字表示一个 $2m$ 位的数。对于这样选取的 m，改进算法的运行时间为 $O(n^3/\log^2 n)$。

评述 我们给出了一个有趣的方法，它计算所有可能性而不像通常那样只计算所需要的。此外，还示范了改变计算顺序可以得到更好的算法。对于那些硬件无法直接处理的位串代数函数，我们可以以相同的方式应用这种计算出所有可能组合的技巧。

9.6 快速傅里叶变换

作为快速傅里叶变换的介绍，这里引用如下出自 John Lipson 所著书籍的一段话：

> 一个有着许多理由值得赏鉴的算法：在技术上，它高效地解决了一个至关重要的实

> 际问题；从美学上，它十分优雅；并且开创了若干崭新且意料之外的应用领域。很可能是因为快速傅里叶变换（fast Fourier transform，FFT）在上述方面都很出色，自从 20 世纪 60 年代中叶发现它以来，它就一直被视为计算机科学中的“超级”算法之一（Lipson[1981, page 293）。

FFT 算法并不简单，它的发展也不是直截了当的。这里仅着重于 FFT 的一个应用场合——多项式乘法。

> **问题**　给定两个多项式 $p(x)$和 $q(x)$，计算它们的乘积 $p(x)\cdot q(x)$。

由于还没有指定多项式的表示方式，上述问题并不是良定义的。通常可将多项式按照次数递减的系数列表方式表示成 $P=a_{n-1}x^{n-1}+a_{n-2}x^{n-2}+\cdots+a_1x+a_0$，这种方式确实可以表示多项式，但它并不是惟一的方式。例如，考察一个 1 次多项式（即线性函数）a_1x+a_0，该线性函数通常由两个系数 a_1 和 a_0 所指定，但由于该函数对应于平面中的一条直线，它可以由那条直线上任何两个（不同的）点所指定。同理，任何 n 次多项式又可由 $n+1$ 个点惟一定义。例如，2 次多项式 $p(x)=x^2+3x+1$ 可以由点(1, 5)、(2, 11)和(3, 19)所定义，它也是惟一包含所有那些点的 2 次多项式。但是，并非只有这 3 点可以定义该多项式，对应曲线上的任何 3 点都可以定义该多项式。

由于将这些点的取值对应于相乘很容易完成，这种表示方法对于多项式乘法非常实用。例如，多项式 $q(x)=2x^2-x+3$ 可以由点(1, 4)、(2, 9)和(3, 18)代表，我们立即可得乘积 $p(x)\cdot q(x)$包含值(1, 20)、(2, 99)和(3, 342)。由于 $p(x)\cdot q(x)$的度为 4，这 3 个点并不足以表示该乘积。如果从每一个较小多项式上各取 5 个点，则可以克服这个问题。例如，对于 $p(x)$添加点(0, 1)和(−1, −1)，而对于 $q(x)$添加点(0, 3)和(−1, 6)，只要通过 5 次标量乘法运算，就可以很容易地得到 5 个属于该乘积的点——(1, 20)、(2, 99)、(3, 342)、(0, 3)和(−1, −6)。使用这一思想，我们只需要用 $O(n)$次乘法运算就能计算两个（以这种方式所表示的）n 次多项式的乘积。

该方法的主要问题是，不能仅为了一个应用去简单地改变表示方式。例如，对于“在任意给定的点必须都能对多项式进行求值”的要求，这种表示方式相比于使用系数的表示方式要困难得多。但是，如果能够高效地从一种表示方式转变成另一种表示方式，则可以得到一个非常好的多项式乘法算法，而这就是 FFT 所能实现的。

通过多项式求值的方法，我们可以从系数的表示转变成点的表示方式。根据 Horner 规则（参见 5.2 节），可以使用 n 次乘法运算来计算多项式（由其系数列表表示）在任意一点的值。由于需要在任意 n 个点对 $p(x)$进行求值，所以总共需要 n^2 次乘法运算。从点的表示方式转变成系数表示方式的过程被称为插值，它通常需要 $O(n^2)$次运算。这里的关键思想（如同本书中许多其他例子）是，不需要针对任意 n 个点，可以自由选取我们想要的任何 n 个不同点所组成的集合。快速傅里叶变换选取了一个非常特别的点集合，使得求值和插值都能很快完成。

正向傅里叶变换

首先考察求值问题。我们需要对两个 $n-1$ 次多项式进行求值，每个多项式各取 $2n-1$ 个点，进而可以对乘积多项式（其度为 $2n-2$）进行插值。但是，我们总可以视 $n-1$ 次多项式为一个 $2n-2$ 次多项式，为此只要将后者的（高）$n-1$ 项系数设置为 0 即可。所以，不失一般性，假定问题是对任意一个 $n-1$ 次多项式 $P=\sum_{i=0}^{n-1}a_ix^i$ 在 n 个不同点进行求值。我们想要找到 n 个点，使得多项式容易对它们进行求值。为简单起见，假定 n 是 2 的幂。

可以使用矩阵术语以简化符号，上述多项式在 n 点 $x_0, x_1, \cdots, x_{n-1}$ 的取值可以表示成如下矩阵

和向量的乘法运算：

$$\begin{bmatrix} 1 & x_0 & (x_0)^2 & \cdots & (x_0)^{n-1} \\ 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & x_{n-1} & (x_{n-1})^2 & \cdots & (x_{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(x_0) \\ P(x_1) \\ \cdot \\ \cdot \\ \cdot \\ P(x_{n-1}) \end{bmatrix}。$$

现在的问题是以何种方式选取 $x_0, x_1, \cdots, x_{n-1}$，从而可以简化这样的乘法运算。考察任意两行——第 i 行和第 j 行，我们想要使其尽可能简单以节省乘法运算。由于这些点两两不等，不能令 $x_i = x_j$；但可以令 $x_j = -x_i$，从而有 $(x_i)^2 = (x_j)^2$。由于 x_i 和 x_j 的所有偶数次幂均相等，所以这是一个好的选择，可以节省一半涉及第 j 行的乘法运算。此外，对于其他每一行对都可使用相同的方法。我们的目标是特别选取 n 行，使得上述过程只需要计算 $n/2$ 次向量积。如果可以做到这一点，就能将问题的规模减小一半，进而得到一个非常高效的算法。让我们尝试以另一种方式表达这个问题，即用一半规模的两个分离子问题提出该问题。

根据上述方案，我们想要将原始问题分解成两个规模为 $n/2$ 的子问题。下列表达式阐释了这个方案：

$$\begin{bmatrix} 1 & x_0 & (x_0)^2 & \cdots & (x_0)^{n-1} \\ 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & x_{n/2-1} & (x_{n/2-1})^2 & \cdots & (x_{n/2-1})^{n-1} \\ & & & & \\ 1 & -x_0 & (-x_0)^2 & \cdots & (-x_0)^{n-1} \\ 1 & -x_1 & (-x_1)^2 & \cdots & (-x_1)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & -x_{n/2-1} & (-x_{n/2-1})^2 & \cdots & (-x_{n/2-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(x_0) \\ P(x_1) \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ P(-x_{n/2-1}) \end{bmatrix} \tag{9.7}$$

式（9.7）中的 $n \times n$ 矩阵分成两个大小为 $n/2 \times n$ 的子矩阵，这两个矩阵非常相似。对于任何满足 $0 \leqslant i < n/2$ 的 i，均有 $x_i = -x_{n/2+i}$。由于两个矩阵中偶次幂的系数恰好完全相等，它们只需要计算一次；而奇次幂的系数虽然不等，但是它们恰好互为相反数。对于 $0 \leqslant i < n/2$，我们将 $P(x_i)$和 $P(-x_i)$的表达式根据其偶次幂系数和奇次幂系数写成如下形式：

$$P(x) = E + O = \sum_{i=0}^{n/2-1} a_{2i} x^{2i} + \sum_{i=0}^{n/2-1} a_{2i+1} x^{2i+1}。$$

“偶”多项式 E 可以写成一个 $n/2-1$ 次常规多项式，后者的系数即为 P 的偶次幂系数：

$$E = \sum_{i=0}^{n/2-1} a_{2i} (x^2)^i = P_e(x^2)。$$

“奇”多项式 O 也可以以相同的方式写成如下形式：

$$O = x \sum_{i=0}^{n/2-1} a_{2i+1} (x^2)^i = x P_o(x^2)。$$

所以，总的来说，可以得到如下表达式：

$$P(x)=P_e(x^2)+xP_o(x^2), \tag{9.8}$$

这里 $P_e(P_o)$是 $n/2-1$ 次多项式，其系数是 P 的偶（奇）次幂系数。当我们把式（9.8）中的 x 用 $-x$ 替换时，可得到 $P(-x)=P_e(x^2)+(-x)P_o(x^2)$。为了对式（9.7）进行求值，我们需要对 $0\leqslant i<n/2$ 计算 $P(x_i)$和 $P(-x_i)$。为了做到这一点，只须对 $P_e(x^2)$和 $P_o(x^2)$各计算 $n/2$ 个值，并执行 $n/2$ 次加法运算、$n/2$ 次减法运算以及 n 次乘法运算。因此，我们有两个规模为 $n/2$ 的子问题以及 $O(n)$次额外计算。

能否以相同的方案继续递归操作呢？如果能，则可以得到熟悉的递归关系 $T(n)=2T(n/2)+O(n)$，从而得到一个 $O(n\log n)$的算法。但这并不那么容易，我们将计算 $P(x)$（其为 $n-1$ 次多项式）在 n 个点取值的问题归约到计算 $P_e(x^2)$和 $P_o(x^2)$（均为 $n/2-1$ 次多项式）在 $n/2$ 个点取值的问题。这是一个有效的归约，仅有一点瑕疵。$P(x)$中 x 的取值可以任意选择，然而 x^2 的值（如出现在 $P_e(x^2)$中的）却只能取正数。由于我们使用负数才能得到这个归约，这就产生了一个问题。让我们从式（9.7）中抽取对应计算 $P_e((x_i)^2)$的矩阵：

$$\begin{bmatrix} 1 & (x_0)^2 & (x_0)^4 & \cdot\cdot & (x_0)^{n-2} \\ 1 & (x_1)^2 & (x_1)^4 & \cdot\cdot & (x_1)^{n-2} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ 1 & (x_{n/2-1})^2 & (x_{n/2-1})^4 & \cdot\cdot & (x_{n/2-1})^{n-2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ \cdot \\ \cdot \\ a_{n-2} \end{bmatrix} = \begin{bmatrix} P_e(x_0) \\ P_e(x_1) \\ \cdot \\ \cdot \\ \cdot \\ P_e(x_{n/2-1}) \end{bmatrix}。$$

如果对这个子问题尝试使用相同的技巧，则需要令$(x_{n/4})^2=-(x_0)^2$，由于平方数总为正数，这看似不可能。但是，如果使用包含 $\sqrt{-1}$ 的复数，那么就能实现上述等式。对于 $0\leqslant j<n/4$，令 $x_{j+n/4}=\sqrt{-1}\,x_j$，这样我们又能再次将问题分解成两部分，同时这次分解具有与第一次分解相同的性质。因此，只要解决两个规模为 $n/4$ 的子问题并进行 $O(n)$次额外计算，就能解决这个规模为 $n/2$ 的问题。

如果想要将该过程进一步推广，则需要一个等于 $\sqrt{\sqrt{-1}}$ 的数；即满足“$z^8=1$ 并且对所有 $0<j<8$ 均有 $z^j\neq 1$”的数 z（从而可以推出 $z^4=-1$ 以及 $z^2=\sqrt{-1}$）。一般说来，对于任意 n 都需要一个满足上述条件的数，这个数被称为 **n 次单位原根**（primitive nth root of unity），用符号ω表示它。（为简单起见，我们并没有在符号中包含 n；而贯穿整节，都将使用相同的 n。）ω满足如下条件：

$$\omega^n=1\text{且}\omega^j\neq 1，\text{对于}0<j<n \tag{9.9}$$

分别选取 $1,\omega,\omega^2,\cdots,\omega^{n-1}$ 作为这 n 个点 $x_0,x_1,\cdots,x_{n-1}$。由此，我们想要计算下列乘积：

$$\begin{bmatrix} 1 & 1 & 1 & \cdot\cdot & 1 \\ 1 & \omega & \omega^2 & \cdot\cdot & \omega^{n-1} \\ \cdot & \omega^2 & \omega^{2\cdot 2} & \cdot\cdot & \omega^{2\cdot(n-1)} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ 1 & \omega^{n-1} & \omega^{(n-1)\cdot 2} & \cdot\cdot & \omega^{(n-1)\cdot(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(1) \\ P(\omega) \\ P(\omega^2) \\ \cdot \\ \cdot \\ P(\omega^{n-1}) \end{bmatrix}。$$

称这样的乘积为$(a_0,a_1,\cdots,a_{n-1})$的**傅里叶变换**。首先，注意到对所有满足 $0\leqslant j<n/2$ 的 j，均有 $x_{j+n/2}=\omega^{n/2}x_j=-x_j$，故我们最先应用于规模为 n 的问题的归约仍然有效。此外，那次归约所生成的子问题

有 $n/2$ 个点，它们分别是 $1, \omega^2, \omega^4, \cdots, \omega^{n-2}$，而这恰好就是用$\omega^2$替换ω之后且规模为 $n/2$ 的子问题。式（9.9）中的条件可以推出ω^2是一个 $n/2$ 次单位原根。因此，可以继续递归操作，从而该算法的复杂度为 $O(n\log n)$。图 9.7 给出了该算法的高层视图。

```
算法 Fast_Fourier_Transform (n, a0, a1, ···, an-1, ω, var V)
输入： n（整数）、a0, a1, ···, an-1（一个元素序列，其类型依赖于实际应用）
       和ω（n 次单位原根）
输出： V（由输出元素构成的数组，范围在[0..n-1]）
       {假定 n 是 2 的幂}
begin
   if n = 1 then
      V[0] := a0 ;
   else
      Fast_Fourier_Transform(n/2, a0, a2, ···, an-2, ω^2, U ) ;
      Fast_Fourier_Transform(n/2, a1, a3, ···, an-1, ω^2, W ) ;
      for j := 0 to n/2 - 1 do { follow (9.8) for x = ω^j }
         V[j] := U[j] + ω^j W[j] ;
         V[j+n/2] := U[j] - ω^j W[j]
end
```

图 9.7　快速傅里叶变换算法

□ **例 9.2**

下面说明了如何计算多项式(0, 1, 2, 3, 4, 5, 6, 7)的傅里叶变换。为了避免混淆，用符号 $P_{j_0 j_1, \cdots, j_k}(x_0, x_1, \cdots, x_k)$表示子问题，此处 $j_0, j_1, \cdots, j_k$ 表示多项式的系数而 $x_0, x_1, \cdots, x_k$ 表示对多项式进行求值之处。所以，这个例子是计算 $P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \cdots, \omega^7)$。（虽然这个符号很不自然，但它却包含了我们所需要的全部信息。）式（9.8）是所使用的主要递归操作。

第 1 步，将 $P_{0,1,2,3,4,5,6,7}(1,\omega, \omega^2, \cdots, \omega^7)$归约到 $P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6)$与 $P_{1,3,5,7}(1, \omega^2, \omega^4, \omega^6)$，然后继续递归操作，将 $P_{0,2,4,6}(1,\omega^2, \omega^4, \omega^6)$归约到 $P_{0,4}(1, \omega^4)$与 $P_{2,6}(1, \omega^4)$，而 $P_{0,4}(1, \omega^4)$又可以进一步归约到 $P_0(1)$与 $P_4(1)$，直接计算可得 $P_0(1)$为 0 而 $P_4(1)$为 4。现在将结果进行组合可得

$$P_{0,4}(1) = P_0(1) + 1 \cdot P_4(1) = 0 + 1 \cdot 4 = 4，\text{以及}$$
$$P_{0,4}(\omega^4) = P_0(\omega^4) + \omega^4 P_4(\omega^4) = 0 + \omega^4 \cdot 4。$$

由于$\omega^4 = -1$，可得 $P_{0,4}(\omega^4) = -4$，组合在一起则有 $P_{0,4}(1, \omega^4) = (4,-4)$。以同样的方式，可以得到 $P_{2,6}(1, \omega^4) = (8,-4)$。

现在将上面的两个向量组合在一起计算 $P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6)$:

$$P_{0,2,4,6}(1) = P_{0,4}(1) + 1 \cdot P_{2,6}(1) = 4 + 8 = 12。$$
$$P_{0,2,4,6}(\omega^2) = P_{0,4}(\omega^4) + \omega^2 \cdot P_{2,6}(\omega^4) = -4 + \omega^2(-4)。$$
$$P_{0,2,4,6}(\omega^4) = P_{0,4}(\omega^8) + \omega^4 \cdot P_{2,6}(\omega^8) = P_{0,4}(1) - 1 \cdot P_{2,6}(1) = 4 - 8 = -4。$$
$$P_{0,2,4,6}(\omega^6) = P_{0,4}(\omega^{12}) + \omega^6 \cdot P_{2,6}(\omega^{12}) = P_{0,4}(\omega^4) - \omega^2 \cdot P_{2,6}(\omega^4) = -4 - \omega^2(-4)。$$

所以，组合在一起有

$$P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6) = (12, -4(1+\omega^2), -4, -4(1-\omega^2))。$$

以同样的方式，可得

$$P_{1,3,5,7}(1, \omega^2, \omega^4, \omega^6) = (16, -4(1+\omega^2), -4, -4(1-\omega^2))。$$

为了计算 $P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \cdots, \omega^7)$，需要计算 8 个值。例如，$P_{0,1,2,3,4,5,6,7}(1) = 12+1\cdot 16 = 28$；同理可得，$P_{0,1,2,3,4,5,6,7}(\omega^4) = 12-1\times 16 = -4$、$P_{0,1,2,3,4,5,6,7}(\omega) = (-4(1+\omega^2))+ \omega\cdot(-4(1+\omega^2))$以及 $P_{0,1,2,3,4,5,6,7}(\omega^5) = (-4(1+\omega^2))\times\omega\cdot(-4(1+\omega^2))$等。其余部分留给读者推导。 □

逆向傅里叶变换

快速傅里叶变换的算法只解决了问题的一半。对两个给定多项式 $p(x)$和 $q(x)$我们都能够迅速在点 $1, \omega, \cdots, \omega^{n-1}$处进行求值，然后将得到的结果相乘，从而得到乘积多项式 $p(x)\cdot q(x)$在那些点的取值。但是，我们还需要根据那些点的取值对乘积多项式的系数进行插值。幸运的是，插值问题事实上非常类似于求值问题，前者可以用几乎完全相同的算法解决。

再次考察矩阵表示法，当给定多项式的系数$(a_0, a_1, \cdots, a_{n-1})$并且想要计算多项式在 n 个点 $1, \omega, \omega^2, \cdots, \omega^{n-1}$ 的取值时，可以按照下列向量乘积对矩阵进行计算：

$$\begin{bmatrix} 1 & 1 & 1 & \cdot\cdot & 1 \\ 1 & \omega & \omega^2 & \cdot\cdot & \omega^{n-1} \\ \cdot & \omega^2 & \omega^{2\cdot 2} & \cdot\cdot & \omega^{2\cdot(n-1)} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ 1 & \omega^{n-1} & \omega^{(n-1)\cdot 2} & \cdot\cdot & \omega^{(n-1)\cdot(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(1) \\ P(\omega) \\ P(\omega^2) \\ \cdot \\ \cdot \\ P(\omega^{n-1}) \end{bmatrix}。$$

另一方面，当给定多项式的取值$(P(1), P(\omega), \cdots, P(\omega^{n-1})) = (v_0, v_1, \cdots, v_{n-1})$而要计算多项式的系数时，我们需要通过求解下列方程组得到 $a_0, a_1, \cdots, a_{n-1}$：

$$\begin{bmatrix} 1 & 1 & 1 & \cdot\cdot & 1 \\ 1 & \omega & \omega^2 & \cdot\cdot & \omega^{n-1} \\ \cdot & \omega^2 & \omega^{2\cdot 2} & \cdot\cdot & \omega^{2\cdot(n-1)} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ 1 & \omega^{n-1} & \omega^{(n-1)\cdot 2} & \cdot\cdot & \omega^{(n-1)\cdot(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \cdot \\ \cdot \\ v_{n-1} \end{bmatrix}。 \quad (9.10)$$

求解方程组通常非常耗时（一般情况下需要 $O(n^3)$时间），但这是一个特殊的方程组。如把这个矩阵等式写成 $V(\omega)\cdot\bar{a} = \bar{v}$，此处 $V(\omega)$指左边的矩阵而 $\bar{a} = (a_0, a_1, \cdots, a_{n-1})$ 以及 $\bar{v} = (v_0, v_1, \cdots, v_{n-1})$。只要 $V(\omega)$可逆，解 $\bar{a}$ 可以写成 $\bar{a} = [V(\omega)]^{-1}\cdot\bar{v}$。事实上，$V(\omega)$总是可逆的；而且它的逆矩阵有一个非常简单的形式（证明略）。

□ **定理 9.1**

$$[V(\omega)]^{-1} = \frac{1}{n} V(\frac{1}{\omega})。$$ □

因此只需要计算一次矩阵与向量的乘积就可以求解方程组（9.10），而下列定理可极大程度地简化这项工作。

□ **定理 9.2**

如果 ω 是 n 次单位原根，那么 $1/\omega$也是 n 次单位原根。 □

因此，只要将ω替换成 $1/\omega$就可以使用快速傅里叶变换的算法来计算乘积 $V(1/\omega)\bar{v}$，该变换被

称为逆向傅里叶变换。

复杂度 总的来说，使用 $O(n\log n)$次运算就可以计算两个多项式的乘积，注意我们需要能够对复数进行加法和乘法运算。

9.7 小结

本章所给出的算法只是已知的代数和数值算法中的一些例子，由此再次说明直接算法并不一定是最佳的。非直观算法可用来解决看似简单的问题，而 Strassen 算法则是其中最显著的例子之一。本章还给出了一些使用归纳法的例子，特别是那些使用分治法的算法的设计。

4 个俄国人算法提出了一项并不基于归纳法的有趣技术，其主要思想是计算某些项的各种可能组合，即使所有组合并非都用得上。在计算全部（或许多）组合的开销远小于分开计算每一种组合的情况下，这项技术非常有用。另一项技术是在问题之间使用归约的方法，它通常可用来解决涉及矩阵的问题，第 10 章将用若干例子来描述这个方法。

参考文献和补充读物

算术和代数算法的最佳参考文献是 Knuth[1981]，其他书籍有 Aho, Hopcroft, and Ullman[1974]、Borodin and Munro[1975]、Winograd[1980]以及 Lipson[1981]等。

重复平方法求幂的算法非常古老：它出现在公元前 200 年左右的印度书籍中（参见 Knuth[1981] page44）。Diffie and Hellman[1976]提出了 RSA 公钥加密机制。欧几里得算法最早出现于欧几里得的《几何原本》第 VII 卷（公元前 300 年左右），但是很有可能在此之前就已经知道这个算法了（参见 Knuth[1981] page318）。Karatsuba and Ofman[1962]发展了两多项式乘积的分治算法（在两大数乘法的上下文中）。

Winograd 算法出现在 Winograd[1968]中（另见 Winograd[1970]），而 Strassen 算法则出现在 Strassen[1969]中。矩阵乘法的渐近运行时间 $O(n^c)$中的常数 c 自从 1969 年以来已经多次减小（首先由 Pan[1978]给出）。当今已知最好的算法（按照渐近运行时间）出自 Coppersmith and Winograd[1987]，其运行时间为$\mathcal{O}(n^{2.376})$。但令人遗憾的是，符号$\mathcal{O}$说明这个算法并不实用。关于矩阵乘法的复杂度及其相关主题的更多信息可以参见 Pan[1984]。Cohen and Roth[1976]中可以找到关于 Strassen 算法的实现。

Arlazarov, Dinic, Kronrod, and Faradzev[1970]给出了 4 个俄国人算法。许多人观察发现使用加法表可以对 4 个俄国人算法进行改进；Rytter[1985]介绍了类似用于上下文无关语言识别的技术，其中简略提及了上述思想。同样的思想还可用于改进序列比较算法（Masek and Paterson[1983]以及 Myers[1988]）。习题 9.26 的解答出现在 Atkinson and Santoro[1988]中。Fischer and Meyer[1971]给出了布尔矩阵乘法和传递闭包问题之间的相互归约。

Cooley and Tuckey[1965]提出了快速傅里叶变换的算法，尽管其原始方法可以追溯至 Runge and König[1924]。关于快速傅里叶变换的更多信息可以参见 Brigham[1974]以及 Elliott and Rao [1982]。

操练型习题

9.1 讨论计算 n^k 的算法 *Power_by_Repeated_Squaring*（见图 9.2）与 k 的二进制表示之间的关系。

9.2 计算 n^k 的算法 *Power_by_Repeated_Squaring*（见图 9.2）并不一定会达到最少乘法运算次数，

给出一个使用更少乘法运算次数计算 n^k（$k>10$）的例子。

9.3　有序对(a, b)可以表示一个正有理数 x（即满足 $x=a/b$），设计一个算法来计算 x 的最小表示；也就是说，x 的表示(a, b)使用的是最小正整数 a 和 b。例如，如果 $x=24/84=6/21=2/7$，那么$(2, 7)$是 x 的最小表示。

9.4　多项式乘法的直接分治算法计算所有 4 个更小规模多项式的乘积，证明其使用的运算量恰好完全等于按照式（9.1）的直接算法，假定此处 n 是 2 的幂。

9.5　使用 9.4 节中所给出的多项式乘法分治算法手工计算乘积 $P(x)\cdot Q(x)$，其中：

$$P(x)=x+2x^2+3x^3+\cdots+15x^{15},$$
$$Q(x)=16+15x+14x^2+\cdots+2x^{14}+1x^{15}。$$

试回答总共需要多少次运算。

9.6　分治技术可用来计算两个二进制数的乘积，请描述这样的算法并讨论它与多项式乘法算法之间的差别。

9.7　使用习题 9.6 中所讨论的算法来计算 10011011 与 10111010 的乘积。

9.8　对于任意基数 b（不仅 $b=2$）所表示的数，分治技术均可用于其乘法运算，使用分治技术计算十进制数乘积 4679×7114，其中的递归操作一直进行到 1 位数。

9.9　设计一个算法来计算两个复数的乘积$(a+bi)(c+di)$，该算法只能使用 3 次乘法运算（i 是–1 的平方根）。

9.10　推导 9.5.2 节中 Strassen 所提出的 2×2 矩阵乘法方案的显式表达式。

9.11　假定你要找到一个使用 k 次乘法计算 4×4 矩阵乘积的算法，基于该算法的一般矩阵乘法算法的复杂度是多少？为了使其渐近复杂度优于 Strassen 算法，最大能取的 k 值是多少？

创造型习题

9.12　考察 9.2 节中计算 n^k 的两个算法（简单迭代法和重复平方法），设 n 是一个 d 位整数，并假定整数乘法可以由常规算法完成（即将两个 d_1 和 d_2 位的整数相乘需要 $d_1\cdot d_2$ 步），这两个算法计算 n^k 各自需要多少步？（可以假定 k 是 2 的幂，并且两个 d_1 和 d_2 位的整数乘积是一个 d_1+d_2 位的整数。）

9.13　设计一个算法寻找 k 个整数的 GCD。

9.14　m 和 n 的最小公倍数（LCM，least common multiple）是指能同时整除 m 和 n 的最小整数，设计一个算法以寻找给定两个整数的最小公倍数。

9.15　设计一个算法以寻找给定 k 个整数的 LCM。（k 个整数的 LCM 是指能同时整除这 k 个数的最小整数。）

9.16　按照下列递归关系定义斐波那契数：

$$F(1)=1,\ F(2)=1,\ F(n)=F(n-1)+F(n-2)\ (n>2)。$$

a. 证明每一个大于 2 的整数 n 都可以写成至多 $\mathrm{lb}n$ 个斐波那契数的和。

b. 设计一个算法对于给定整数 n 寻找这样的表示方式。

9.17　设 $P(x)$和 $Q(x)$是两个多项式。如果存在一个多项式 $S(x)$使得 $P(x)=D(x)\cdot S(x)$，那么可称多项式 $D(x)$整除 $P(x)$。类似地，如果 $R(x)$的度小于 $P(x)$，并且存在一个多项式 $D(x)$使得 $Q(x)=D(x)\cdot P(x)+R(x)$，则可称 $Q(x)=R(x)\mathrm{mod}P(x)$。多项式 $P(x)$和 $Q(x)$的最大公因式是一个能同时整除 $P(x)$和 $Q(x)$且次数最高的多项式 $R(x)$。

a. 证明两个多项式的 GCD 有惟一的定义。

b. 推广欧几里得算法以寻找两个给定多项式的 GCD。

9.18 将 9.4 节所述的多项式乘法算法按照如下方式修改：将每个多项式分成 3 个相等的部分（而不是先前的两部分），然后将较小部分之间的乘法次数最小化。可以假定问题的规模是 3 的幂，该算法的复杂度是多少？

9.19 将 9.4 节所述的多项式乘法算法按照如下方式修改：将每个多项式分成 4 个相等的部分，然后将较小部分之间的乘法次数最小化。可以假定问题的规模是 4 的倍数，该算法的复杂度是多少？

9.20 哈密尔顿四元数（Hamilton's quaternion）是形如$(a+bi+cj+dk)$的向量，这里 a、b、c 和 d 是实数而 i、j 和 k 是特殊符号。四元数的加法和减法定义为各个分量对应的相加和相减，而下列规则定义了四元数的乘法：

$$i^2 = j^2 = k^2 = -1$$
$$ij = -ji = k$$
$$jk = -kj = i$$
$$ki = -ik = j$$

（符号 i、j 和 k 可以与实数及其自身交换）。四元数的乘法运算过程需要多少次实数乘法运算？给出一个算法将乘法运算减少至 12 次。

9.21 说明如何仅用 5 次乘法运算来计算 2×2 矩阵的平方。

9.22 置换矩阵（permutation matrix）是一个 $n \times n$ 矩阵，其满足每一行与每一列恰好只有一个等于 1 的非零元素。置换矩阵可以表示成一个数组 P，如果第 i 行在第 j 列的元素为 1，则 $P[i]=j$。

a. 证明两个置换矩阵的乘积仍然是一个置换矩阵。

b. 设计一个线性时间算法将两个以数组形式表示的置换矩阵相乘，结果也应当以数组形式表示。

9.23 考虑以下对 Strassen 算法的修改建议：根据 Winograd 算法，可以使用大约 $k^3/2$ 次乘法运算来计算两个 $k \times k$ 矩阵的乘积，进一步可将这个乘积作为分治策略的基础，从而代替原来基于 2×2 矩阵的那个策略。如果 k 足够大，则可以得到一个渐近时间优于 Strassen 算法的算法，这个建议中哪里有错误？

9.24 设计一个算法来计算两个 k 维布尔向量全部可能的布尔内积（参见 9.5.3 节），该算法应当创建一个规模为 2^{2k} 的表，两个向量 a 和 b 的内积应当存储在条目 i 处，这里 i 是由 $2k$ 位所表示的整数（其中，a 和 b 的二进制表示分别是最高有效位的 k 和最低有效位的 k），该算法的运行时间应为 $O(2^{2k})$。

9.25 完成布尔矩阵乘法（见图 9.6）的程序，说明如何显式地建立那些表，以及如何在不显著影响效率的前提下处理 n/k 不为整数的情况。

9.26 设计一个布尔矩阵乘法的算法，其将矩阵分成规模为 $k \times k$ 的子矩阵，然后采用预先计算这些子矩阵间所有可能乘积的思想。该算法的运行时间应为 $O(n^3/(\log n)^{1.5})$，并且需要额外存储空间 $O(n\log n)$。可以假定 $\text{lb}n$ 位数的基本运算可以在一步内完成。

9.27 设 A 和 B 是两个 $n \times n$ 的随机布尔矩阵，矩阵中的每一个元素都以 1/2 的概率随机（且独立）地选取为 0 或 1，设计一个期望运算量为 $O(n^2)$的算法来计算 A 和 B 的乘积矩阵。

9.28 设 A 和 B 是两个 $2n \times 2n$ 的布尔矩阵，其按照下列方式用开放格雷码（参见 2.9 节）表示：矩阵的行对应格雷码位串，使得两相邻行恰好相差 1 位（首行和末行可以相差超过 1 位）。设计一个 $O(n^2)$的算法来计算两个矩阵的乘积。

*9.29 设 $M_1, M_2, \cdots, M_n$ 是 n 个实数矩阵，其中 M_i 的维数为 $a_i \times a_{i+1}$，因而乘积 $M_i \cdot M_{i+1}$ 对所有 $1 \leqslant i < n$ 均有定义，我们想要计算乘积 $M_1 \times M_2 \times \cdots \times M_n$。假定将两个 $a_i \times a_{i+1}$ 矩阵和 $a_{i+1} \times a_{i+2}$ 矩阵相乘需要 $a_i a_{i+1} a_{i+2}$ 次运算，问题是寻找执行这些乘法的恰当顺序。例如，设 $n = 3$ 且这些矩阵的维数分别为 10×2、2×5 和 5×3，计算前两个矩阵的乘积并生成一个 10×5 的矩阵需要进行 10×2×5 次运算，将后者再与第三个矩阵相乘需要进行 10×5×3 次运算，故总共需要 250 次运算；另一方面，如果我们首先计算后两个矩阵的乘积，然后再将第一个矩阵与之相乘，最终只需要 90 次运算。设计一个算法寻找执行上述矩阵乘法运算的最佳顺序。



第10章 归　约

知识有两种，一种是对事物的亲身了解，还有一种是知道从何处可找到相关信息。

——塞缪尔·约翰逊（1775）

10.1 引言

下面是一个古老的笑话。有人问一个数学家和她的丈夫如下问题：假设你在地下室，想烧开水，怎么办？数学家说，她会上到厨房并在那里烧水。她丈夫的回答类似。那人继续问：假设你在厨房，想烧开水，怎么办？她丈夫回答说："这就容易的，将水壶灌满然后点燃煤气。"而数学家则回答："我的答案更简单，到地下室去，在那里我已经知道如何解决这个问题了。"

本章将讨论归约技术及其思想，有时候归约不仅有趣，还非常有用。下面是归约的另一个真实的例子：当你用联邦快递把一个包裹从纽约的住宅区送到商业区时，这个包裹将取道孟菲斯。联邦快递所有的包裹都会经过孟菲斯集散点，所以当面对市区间包裹传递的特殊情况时，他们也"已经知道如何解决该问题了"。在这里，这种解决方法就很有意义。识别一种特殊情况并建立一种有效的处理机制可能很难，而统一、平等地处理所有事情总的来说将更简易便捷。这也适用于算法设计，当我们碰到一个问题时，如果发现它是另一个解已知的问题的特殊形式，那么就可以使用已知的解决方法。尽管有时这样的解决方法过于普通或代价昂贵，但在多数情况下，使用一般解是最简单、最方便、最佳的方法，其实我们每天都在使用这样的原理。一些计算问题，例如数据库查询，一般无须写出一个只能解决该问题的特殊程序，而应该充分使用能处理更一般问题的通用软件。通用的解决方案可能不是最有效的，但更易于使用。

假设有一个较复杂的问题 P，而它看起来与一个已知问题 Q 很相似，尽管可能存在其他方法，我们也可以尝试挖掘或借用解决 Q 的一些方法，并应用于 P。可以试着在两个问题间找到一个**归约**（reduction）或者变换（transformation）。非严格地说，归约是使用解决其他问题的"黑盒"来解决另一个问题。取决于问题解决的先后，归约可以达到两个目标之一（也就是说，用哪个黑盒来解决哪个问题）。如果我们已知 Q 的算法，那么就可以把使用了 Q 的黑盒的 P 的解决方法转化成一个 P 的算法。另一方面，如果 P 是一个已知的难题，或者如果知道 P 的下限，那么同样的下限也可能适用于 Q。前一个归约用于获取 P 的信息；而后者则用于获取 Q 的相关信息。

例如，在10.4.2 节中，我们将讨论矩阵乘法与矩阵平方（即自乘）。显然，可以用矩阵相乘的算法来平方一个矩阵，所以，矩阵平方问题可以归约为乘法问题。在 10.4.2 节里还说明可以使用矩阵平方的方法来设计矩阵相乘的算法，因此矩阵相乘也可归约为矩阵平方。后一个归约的目的是为了说明矩阵平方不可能比两个任意的矩阵相乘（在 10.4.2 节中讨论的情况下）完成得更快（多于常数倍数）。

在本章我们将看到归约应用的若干例子。在两个问题间找到一个归约是很有用的，即使其结果并未直接导致新问题的复杂度上下限的变化。归约让我们理解了两个问题间的关系，它是一种可用于寻找解决某问题或其变形的技术，例如，可以用于设计一个问题的并行算法。

使用归约的一种有效方法是定义一个一般问题，使得很多问题都能归约到其上。找到这样的

问题是不容易的。这个问题必须足够一般，以便能覆盖各种广泛的问题变形，但它又必须足够简单以至于能方便地找到它的有效解。10.3 节将讨论一个这样的问题——线性规划问题。

本书中我们已见过若干归约的范例，比如传递闭包问题（transitive-closure problem）、全对最短路径问题（all-pairs shortest-paths problem）的归约（见 7.8 节）。归约是如此重要，值得我们花整个章节的笔墨来描述，何况它还是下一章的基础。

10.2　归约的例子

在这一节中，我们将展示 4 个用归约来获得有效算法的例子。

10.2.1　简单字符串匹配问题

下面是字符串匹配问题的一个简单变形。

> **问题**　设两个长度为 n 的字符串 $A=a_0a_1\cdots a_{n-1}$ 及 $B=b_0b_1\cdots b_{n-1}$，判断 B 是否为 A 的循环平移。

该问题是判断是否存在一个 k，$0\leqslant k\leqslant n-1$，使得对所有的 i，$0\leqslant i\leqslant n-1$：$a_i=b_{(k+i)mod\ n}$。我们把该问题称为 CSM（循环字符串匹配，cyclic string matching），称原始的字符串匹配问题为 SM（见 6.7 节）。我们可以修改 6.7 节中介绍的 Knuth-Morris-Pratt 算法来解决 CSM，但有更好的方法。想法就是把 CSM 作为 SM 的一个规则实例（regular instance）。换句话说，我们应该寻找特定的**文本** T 及特定的**模式** P，使得在 T 中寻找 P 等同于判定 B 是否为 A 的循环平移。如果我们可以做到这样，那么包含 T 和 P 的 SM 问题的解决方法也能应用于包含 A 和 B 的 CSM 问题中。如果往这些方向多想想，答案显而易见：把 T 定义为 AA（也就是 A 再连接它本身），显然，B 是 A 的一个循环平移当且仅当 B 是 AA 的一个子串。既然我们已经知道如何在线性时间内解决 SM 问题，那么同样有 CSM 问题的线性时间算法。

10.2.2　特殊代表集

设 $S_1, S_2, \cdots, S_k$ 是一组集合。**特殊代表集**（system of distinct representative，SDR）是一个集合 $R=\{r_1, r_2, \cdots, r_k\}$，其中 $r_i\in s_i$，对所有的 i，$1\leqslant i\leqslant k$（注意，既然 R 是集合，r_i 就必须是相异的）。换句话说，R 刚好包含每个集合的一个代表元素。给定一组集合未必能找到一个 SDR，例如集合组 $S_1=\{1, 2\}$，$S_2=\{2, 3, 4\}$，$S_3=\{1, 3\}$，$S_4=\{1, 2, 3\}$的 SDR 是{1, 4, 3, 2}，但集合组 $S_1=\{1, 2\}$，$S_2=\{2, 3, 4\}$，$S_3=\{1, 3\}$，$S_4=\{1, 2, 3\}$，$S_5=\{2, 3\}$ 就不存在 SDR。

> **问题**　给定一由有穷个有穷集合组成的组，找出该组的（任意一个）SDR，或判断 SDR 是否存在。

有一个非常优美的定理，就是 P. Hall 给出的 SDR 存在的充要条件。设 $card(S)$ 是 S 的元素个数。

□ Hall 定理

设 $S_1, S_2, \cdots, S_k$ 是一组集合，其存在 SDR 当且仅当下面的条件成立：

$card\left(S_{i_1}\cup S_{i_2}\cup\cdots\cup S_{i_m}\right)\geqslant m$，对 $\{1, 2, 3, \cdots, k\}$ 的每个子集 $\{i_1, i_2, \cdots, i_m\}$

换句话说，任意 m 个集合的并必须至少包含 m 个不同的元素，对所有的 $1 \leqslant m \leqslant k$。 □

显然，这个条件是必要的，因为如果存在某 m 个集合其并起来的（不同）元素个数少于 m，就不可能有 m 个特殊代表。充分性则比较难证，留做习题。

Hall 定理提出了很简单的条件，但是很遗憾，它不能被直接有效地检验——必须检查所有可能的子集，总共 2^k 个。所以我们需要其他方法：想法是把它看成双向匹配问题。设 $G = (V, U, E)$ 是一个双向图，使得对每个集合 S_i 均在 V 中存在一个顶点 v_i，并且对所有可能的元素（即对集合的并中的每个元素）在 U 中都有一个顶点 u_j。每个元素与包含它的所有集合相连，即 $(v_i, u_j) \in E$ 当且仅当 $u_j \in S_i$。这样显而易见，SDR 就是 G 中规模为 k 的匹配，我们就可以应用 7.10 节中讨论的算法。此外，Hall 定理的证明可以从双向匹配和网络流量的性质中获得。

10.2.3 关于序列比较的归约

回忆在 6.8 节中讨论的序列比较（sequence comparison）问题：$A = a_1 a_2 \cdots a_n$ 及 $B = b_1 b_2 \cdots b_m$ 是两个字符串，我们要逐字符地编辑 A 直到它变为 B。允许三种编辑方式，每种涉及一个字符——插入、删除和替换。这些步骤的代价都已给出，我们的目标是使编辑的代价最小。6.8 节给出的解决方法是构造一张 $n \times m$ 的表，其中每一单元对应一个部分编辑（partial edit）。第 ij 项包含把 A 的前 i 个字符编辑成 B 的前 j 个字符的代价。最终目标是照此完成表的右下角项（nm）。我们证明了每一项都只能从对应三种不同编辑步骤的前三项计算出来。

考虑该问题的另一种方法是把表看成一个有向图。表中每一项对应图的一个顶点，这样每个顶点对应一个局部编辑。当 v 对应的局部编辑有一个以上编辑步骤到 w 对应的局部编辑时，存在一条边 (v, w)。此类图的例子见图 10.1，其中 $A = caa$ 及 $B = aba$。水平边对应插入，垂直边对应删除，而斜边对应替换。例如，图 10.1 中阴影路径对应：删除 c，匹配 a，插入 b，再匹配 a。在基本问题中，每条边的代价是 1，除了对应相同字符的斜边（即无须替换）的代价是 0。现在这个问题就变成了一个常规的单起点最短路径问题。每条边都关联一个代价（对应于编辑的代价），而我们正在寻找从顶点[0, 0]到顶点[n, m]的最短路径。我们已经把字符串编辑问题归约成单起点最短路径问题。

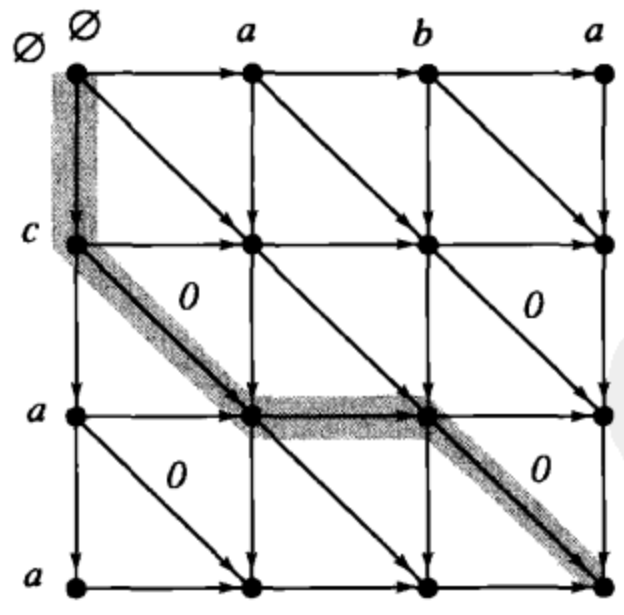


图 10.1 对应序列 $A = caa$ 及 $B = aba$ 的图

一般来说寻找最短路径不比直接解决这个问题简单，但归约还是很有用的。设想有下列各种变形的序列比较问题。编辑的代价不一定是以一个字符进行计算的，在字符串正中插入一块字符的代价与在其他地方逐个插入相同数目字符的代价可能不一样，删除也是如此。换句话说，要对

一块字符的操作赋值，无论其大小，而不是对单步操作赋值。又如，我们想为插入一个 k 字符的块赋予代价，比方说 $I+ck$，其中 I 是“起步”（start-up）价，c 是后来的每个字符的代价。还有许多其他有用的构造法，都可以用最短路径形式对其建模，这远比更改原来的问题来得容易。我们可以在任何地方需要的地方添加边并对它们赋值，而无须修改问题。

10.2.4　在无向图中寻找三角形

图和矩阵存在着强烈的对应关系。一个有n个顶点的图 $G=(V, E)$可以表示成它的**邻接矩阵**（adjacency matrix）A，它是一个 $n\times n$ 的矩阵，其中第 ij 项是 1 当且仅当 $(v_i,v_j)\in E$。如果 G 是无向的，则 A 是**对称**（symmetric）的。如果 G 是一个加权图，那可以定义 A 为一个 $n\times n$ 的矩阵其 ij 项等于边 (v_i, v_j) 的**权重**（weight）或 0，如果图中没这条边。还有其他的方法可以把矩阵和图相关联。例如，一个有 n 个顶点 m 条边的图 $G=(V, E)$的**关联矩阵**（incidence matrix）是一个 $n\times m$ 的矩阵，其 ij 项是 1 当且仅当第 i 个顶点与第 j 条边相邻。

关联不仅停留在表面。图的许多属性可以由观察对应的矩阵来更好地理解，类似地，矩阵的许多属性也可由查看对应的图发现。不奇怪，很多算法问题都是利用这种类推来解决的。这里就有个例子：

> **问题**　设 $G=(V, E)$是一个有 n 个顶点和 m 条边的无向图。设计一个算法判断 G 是否含有两两连接的三个顶点。

最直截了当的方法是查看所有三个顶点的子集，有 $\binom{3}{n}=n(n-1)(n-2)/6$ 个子集，而每个子集都可以在常数时间内检查。因而该算法的时间复杂度是 $O(n^3)$。可以设计出一个时间复杂度为 $O(mn)$ 的算法（见习题 7.118），该算法对稀疏图的效果更好，但还能做得更好吗？我们接着给出了一个不那么直观但逼近更快的算法，这次讨论的主要目的是阐明图算法与矩阵算法间的关系。

设 A 是 G 的邻接矩阵。因为 G 是无向的，所以 A 是对称的。记 A^2 是矩阵 A 的平方，即 $A^2=A\times A$（这里乘法是一般的矩阵乘法）。我们要学习 A^2 中各项与图 G 之间的关系，由矩阵乘法的定义

$$A^2[i, j]=\sum_{k=1}^{n} A[i, k]\cdot A[k, j]。$$

所以，$A^2[i, j]>0$ 当且仅当存在索引 k 使得 $A[i, k]$和 $A[k, j]$均为 1。从图的角度看，$A^2[i, j]>0$，如果当存在一个顶点 k，$k\neq i$ 且 $k\neq j$，并且 i 和 j 都与 k 连接（假设图不包含自回路，这样对所有的 i 都有 $A[i, i]=0$）。无论如何，这意味着存在一个关于 i 和 j 的三角形当且仅当 i 与 j 相连且 $A^2[i, j]>0$。这样，在 G 中存在一个三角形当且仅当存在 i 和 j 使得 $A[i, j]=1$ 且 $A^2[i, j]>0$。

上面的讨论蕴含着一个算法。我们先计算了 A^2 然后对每对 i 和 j 都检查上述条件。检查所有对花费了$O(n^2)$时间，所以算法的时间复杂度由矩阵相乘所决定。我们就这样把在图中寻找三角形的问题归约为布尔矩阵相乘问题（更精确地讲是矩阵平方，但在 10.4.2 节中我们将看到这二者是等价的）。现在可以用矩阵相乘的 Strassen 算法来获取一个寻找三角形的算法，时间复杂度是 $O(n^{2.81})$，还可以用 9.5.3 节中的布尔矩阵相乘算法来获得时间复杂度是 $O(n^3/(\log n)^2)$的更实用的算法。我们把这个图问题归约为布尔矩阵乘法，一般来说，该问题的复杂度是 $O(M)$，其中 M 是布尔矩阵乘法的复杂度。

10.3 有关线性规划的归约

前面的章节中包含了从不同领域进行归约的算法设计实例。我们试图把一个问题映射为另一个，这样就能用一个已知的算法。这一节也介绍归约，但方法略微不同。我们探索一些很多问题可以归约到的“超问题”（super problems），而不是当新的问题出现时寻找归约的一个候选者。有这样一个超问题，或许是最重要的一个，就是**线性规划**（linear programming）。有直效的算法解决线性规划，尽管都不简单。对线性规划彻底的讨论已超出本书范围，所以本节只定义问题的一些类型，并介绍归约到它的若干例子。

10.3.1 概述与定义

许多问题涉及在一定条件下特定函数的最大值和最小值。例如，网络流量问题涉及到流量函数在容量和守恒限制下的最大值。线性规划是此类问题的一般形式，其函数是线性函数，约束也可用下列方法写成线性方程组。设 $\bar{x}=(x_1,x_2,\cdots,x_n)$ 是一个变量向量，**目标函数**（objective function）定义为关于 $\bar{x}$ 变量的线性函数：

$$c(\bar{x})=\sum_{i=1}^{n}c_i x_i,\ \text{其中}\ c_i\ \text{是常量。} \tag{10.1}$$

线性规划的目的是找到满足约束关系（下面列出的）的 $\bar{x}$ 的值，同时使目标函数的值最大化。稍后可以看到，如果有必要，把目标最大化替换为目标最小化是很容易的。首先，定义带有三种约束类型的线性规划的一般形式，当然并非所有问题都需要这些约束；接着就证明问题本身可以简化为带两种约束类型的问题。

设 $\bar{a}_1,\bar{a}_2,\cdots,\bar{a}_k$ 是实数向量，每个长度为 n，并设 $b_1,b_2,\cdots,b_k$ 是实数。**约束不等式**（inequality constraints）如下所示：

$$\begin{aligned}
&\bar{a}_1\cdot\bar{x}\leqslant b_1\\
&\bar{a}_2\cdot\bar{x}\leqslant b_2\\
&\quad\cdots\\
&\quad\cdots\\
&\bar{a}_k\cdot\bar{x}\leqslant b_k。
\end{aligned} \tag{10.2}$$

（除了 $\bar{x}$，所有符号都是常量。）

约束等式（equality constraints）也类似：

$$\begin{aligned}
&\bar{e}_1\cdot\bar{x}=d_1\\
&\bar{e}_2\cdot\bar{x}=d_2\\
&\quad\cdots\\
&\quad\cdots\\
&\bar{e}_m\cdot\bar{x}=d_m,
\end{aligned} \tag{10.3}$$

其中 $\bar{e}_1,\bar{e}_2,\cdots,\bar{e}_m$ 也是长度为 n 的向量，$d_1,d_2,\cdots,d_m$ 是实数。

通常还分开引入下面的**非负约束**（nonnegative constraints），即使它们可以表示成上述约束的特殊形式。

$$x_j \geqslant 0，\text{对所有的} j \in P，\tag{10.4}$$

其中 P 为$\{1, 2, \cdots, n\}$的一给定子集。

现在线性规划问题可形式化地阐述如下：在满足约束不等式、约束等式和非负约束的条件下使函数 $c(\bar{x})$ 最大化。当然，一个具体问题并非一定要包括所有类型的约束。

首先证明，不失一般性，可以舍去约束不等式或等式之一，但不能全舍去。设

$$\bar{e}_i \cdot \bar{x} = d_i \tag{10.5}$$

是任意一个约束等式。我们可以把它替换为下面两个约束不等式：

$$\bar{e}_i \cdot \bar{x} \leqslant d_i \tag{10.6}$$

及

$$-\bar{e}_i \cdot \bar{x} \leqslant -d_i。\tag{10.7}$$

另一方面，可以把约束不等式替换成约束等式。给定一个一般的约束不等式

$$\bar{a}_i \cdot \bar{x} \leqslant b_i，\tag{10.8}$$

引入一个新的变量 y_i，替换成

$$\bar{a}_i \cdot \bar{x} + y_i = b_i，\text{且} y_i \geqslant 0。\tag{10.9}$$

这样的变量被称为**松弛变量**（slack variable）。一个只带约束等式的线性规划被称为**标准式**（standard form）。

所有这些情况下，把约束从一种类型改为另一种类型都可能引起约束数目的增加，所以进行这样的转化未必是好方法。

这里不描述解决线性规划问题的算法，只说明实际上解决线性规划的已知算法非常快，这样，归约到线性规划不是一个技巧练习，而是一个解决问题极好的方法。

10.3.2　归约到线性规划的例子

实际问题很少是以线性规划形式直接给出的，通常必须引入正确的定义使之符合既定的形式，下面是一些例子。

网络流量问题

（该问题在 7.11 节中已讨论过）设变量 $x_1, x_2, \cdots, x_n$ 表示所有边的流出量（这里 n 是边的数目）。目标函数是

$$c(\bar{x}) = \sum_{i \in S} x_i，$$

其中 S 是从源出发的边的集合。对应容量约束的约束不等式是

$$x_i \leqslant c_i，\text{对所有 } i，1 \leqslant i \leqslant n，$$

其中 c_i 是边 i 的容量。对应守恒约束的约束不等式是

$$\sum_{\text{离开 } v \text{ 的 } x_i} x_i - \sum_{\text{进入 } v \text{ 的 } x_j} x_j = 0，\text{对所有 } v \in V - \{s, t\}。$$

最后，对所有变量都应用非负约束（即 10.4 节中定义的集合 P 是全集$\{1, 2, \cdots, n\}$），而如何

求在这些约束下使目标函数最大，即对应流量最大的 x 值，把它留给读者。

静态路由问题

设 $G=(V,E)$是无向图，表示一个通信网络。假设网络中每个节点有一个极限缓存，在一个单位时间内只能接收 B_i 条消息（为简单起见，设所有消息的长度均相同）。再假设可以从任一链路传输的消息数目无限制，每个节点均有无限的消息供应，问题是在单位时间内每条边应该传送多少条消息才能使网络上的总消息数量最多（这是静态路由问题，因为我们假设所有的节点都能传送消息，而一般传送都会动态改变）。用图论表述，问题就是对边赋予权重，连接节点 v_i 的边的权重$\leqslant B_i$，而使所有权重的和最大化。

这个图论问题可以很容易表述成一个线性规划问题。我们可以对每条边 $e_i=(v,w)$关联一个变量 x_i，表示经过 e_i的消息数目。目标函数是$c(\bar{x})=\sum_i x_i$，约束如下：

$\sum_{\text{连接 } v_j \text{ 的 } e_i} x_i \leqslant B_i$，对所有的$v \in V$，以及

$x_i \geqslant 0$，对所有的 i。

慈善家问题

假设有 n 个组织要向 k 个计算机科学部门捐助，每个组织 i 在一年内都有总捐助限度 s_i，对于部门 j 的捐助同样也有限度 a_{ij}（比如对某些部门 a_{ij}可能是 0）。一般来说，s_i要比 $\sum_{j=1}^{k} a_{ij}$小，因而每个组织都要做一些抉择。此外，设每部门也有收受总金额的限制 t_j（虽然这个约束不太现实，但还是很有趣的）。目标是设计一个算法使总捐助金额最大（不论公平与否）。

该问题是 7.10 节中介绍的匹配问题的泛化，可由匹配技术解决，但它有一个简单的线性规划的表述。有 nk 个变量 x_{ij}（$1 \leqslant i \leqslant n$，$1 \leqslant j \leqslant k$），表示组织 i 对部门 j 的捐助数目。目标函数是

$$c(\bar{x})=\sum_{ij} x_{ij}。$$

约束如下：

$x_{ij} \leqslant a_{ij}$，对所有的 i,j，

$\sum_{j=1}^{k} x_{ij} \leqslant s_i$，对所有的 i，

$\sum_{i=1}^{n} x_{ij} \leqslant t_j$，对所有的 j。

当然，所有变量都是非负的。

分配问题

对慈善家问题稍加更改：强调每个组织只能向一个部门捐助，而每个部门也仅接收一个组织的捐助。换句话说，我们使之成为一个加权的标准匹配问题。每个可能的匹配都有附属的数量，而我们不仅要寻找一个完美匹配，还要使总捐助数最大。这个问题是一个双向加权匹配问题，或者被称为**分配问题**。

这个问题的变量与其他问题有所不同。我们要设法抓住匹配的概念，必须强调只有一条边能连接每个节点。我们为每条边(i,j) 赋予一个变量 x_{ij}，当选择该边时为 1，反之为 0。目标函数变为

$$c(\bar{x}) = \sum_{ij} a_{ij} x_{ij}。 \tag{10.10}$$

约束如下：

$\sum_{j=1}^{k} x_{ij} = 1$，对所有的 i，

$\sum_{i=1}^{n} x_{ij} = 1$，对所有的 j。

这些约束保证了每个节点只能选择一条边。另外，所有变量都为非负。

这个设计有一个重大的不足，变量表示了是或非的抉择，但它们最佳值应该是实数。我们只能添加约束来限制变量的值为 0 或 1，这通常很难做到。变量是整数的线性规划被称为**整数线性规划**（integer linear programs），解决它们要涉及**整数规划**（integer programming）。前面章节中讨论的很多问题都可以很自然地表述为整数线性规划问题。尽管线性规划可以有效地解决，可是整数线性规划往往（但不总是）很难。在下一章中我们要讨论这个问题（顺便说一下，分配问题可以由线性规划有效地解决。例如见 Papadimitriou and Steiglitz[1982]）。

10.4　下界的归约

如果解决问题 A 的算法无须增加太多运行时间就可改为解决问题 B 的算法，那么问题 B 的下界也能适用于问题 A。下面给出三个使用归约的例子来说明下界问题的求解，还有一个例子将在下一节出现，它涉及归约应用中普遍出现的错误。

10.4.1　寻找简单多边形算法复杂度的下界

考虑用一个简单闭多边形连接平面上一系列点的问题（参见 8.3 节），我们已经知道如何用排序来解该问题，在某些假设下，这个问题不可能比排序来得更快。所以，不改进排序方法就无法改进简单闭多边形问题的求解速度（改进意味着速度的提高大于常数因子）。

□ **定理 10.1**

给定一个在 $O(T)$ 时间内的简单多边形问题的（黑盒）求解算法，就可以在 $O(T+n)$ 时间内进行排序。 □

证明：考虑圆上的 n 个点（见图 10.2），将这些点连接成一个简单多边形的惟一方法是把每个点与其相邻的点相连，如果将不相邻的两个点相连，连线会把其他的点分隔为两组，从而它们的连线将不可避免地与该线交叉。现在考虑排序问题的一个输入 $x_1, x_2, \cdots, x_n$，如果有一个求解简单多边形问题的黑盒，我们可以用它这样来排序：把输入 $x_1, x_2, \cdots, x_n$ 先转化为 $y_1, y_2, \cdots, y_n$，使 y_i 均是在–180°到 180°的角，而保持与 x_i 中的相对位置相同。接着把这些角转化为分布在单位圆上的点，x_i 对应的点是圆上与某条固定直径成角度 y_i 的点。这些转化可以在线性时间内完成，现在可以用黑盒在 $O(T)$ 时间内由一系列点构造出一个简单多边形。而如我们所说的，该简单多边形必然在圆上连接每个相邻点。那正说明了我们可以按顺序扫描这些点并在 $O(T+n)$ 时间内找出原序列的排序。 □

为了取得简单多边形问题的下界，我们要小心应对设想的计算模型。排序的下界 $\Omega(n \log n)$ 已经在 6.4.6 节中采用决策树（decision-tree）模型证明了，要把该下界用到简单多边形问题中，必须用同样的模型。即先假设解决简单多边形问题的黑盒 $O(T)$ 的比较方法是与决策树模型一致

的，定理必须包含这个假设。然后，必须证明归约也与决策树模型一致。这里归约是合法的，因为排序下界的证明没有对决策树中允许的问题类型做任何限制。这样，涉及点的 x 或 y 坐标对应于角度 y_i 的比较仍然算做决策树的一个比较。决策树求解简单多边形问题可以不显著增加高度而转化成决策树求解排序问题。

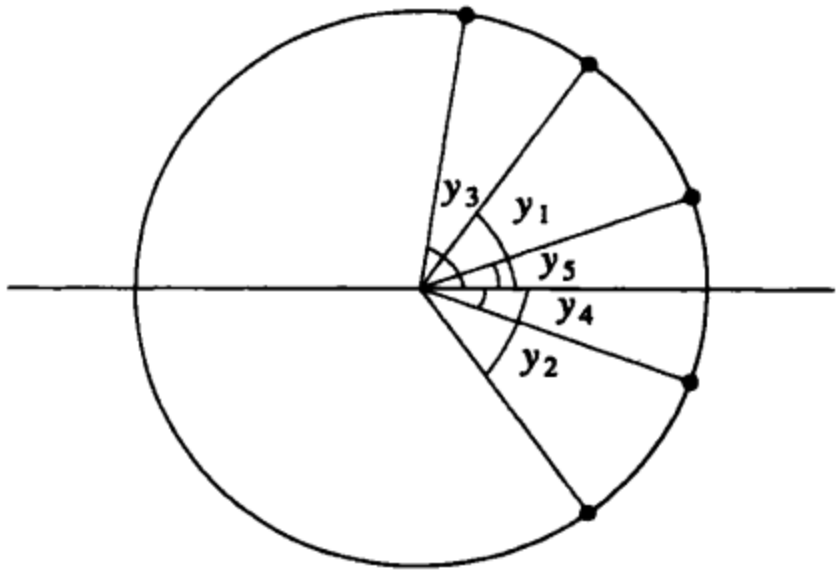


图 10.2 从数字到点的转化

□ **推论 10.2**

在决策树模型下，在平面上寻找连接一系列给定点的简单多边形，在最坏情况下需要 $\Omega(n \log n)$的比较。 □

这个归约证实了排序的确是解决简单多边形问题的核心。

10.4.2 关于矩阵的简单归约

在 9.5 节，我们看到两个矩阵相乘的非常不直接的方法。对称矩阵（symmetric matrices，即矩阵中 ij 项与 ji 项相等）在实践中经常出现，所以很自然会提出这个问题：对称矩阵相乘是否会更简单？对称性完全可能帮助我们找到更好的表达式来相乘，比方说，3×3 的矩阵。这可能通向更好的对称矩阵相乘的逼近算法。现在我们要证明并非如此，两个对称矩阵相乘与两个任意矩阵相乘在常数因子内是同样困难的。

记计算两个任意矩阵相乘问题为 *ArbM*，对称矩阵相乘为 *SymM*。显然，*SymM* 不会比 *ArbM* 困难（因为 *SymM* 不过是 *ArbM* 的一个特例）。现在假设有一个解决 *SymM* 的算法，可证明能用该算法（黑盒）来解决更一般的 *ArbM* 问题。设 A 和 B 是两个任意矩阵，记 A^T 是 A 的**转置**（即从交换 A 的第 ij 项和第 ji 项来获取），利用下面关于两个 $2n \times 2n$ 矩阵的式子，易证明：

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix}。 \tag{10.11}$$

（其中 0 代表所有项均为 0 的 $n \times n$ 矩阵。）归约遵循左边两个矩阵是对称的事实，所以可以使用 *SymM* 问题的算法来得到乘积，而乘积的左上方恰为 AB。这样，可以在两个双倍大小的矩阵上使用 *SymM* 的算法来解决 *ArbM* 问题。有以下的定理：

□ **定理 10.3**

如果有算法在 $O(T(n))$时间内计算两个 $n \times n$ 的对称实矩阵相乘，满足 $T(2n) =$

$O(T(n))$，那么同样有算法在 $O(T(n)+n^2)$时间内计算两个 $n\times n$ 的任意实矩阵相乘。 □

证明：给定任意两个 $n \times n$ 的矩阵，如式(10.11)中所示计算其乘积。需要用 $O(n^2)$步来计算 A^T 和 B^T，并构造两个对称矩阵，$T(2n)$相乘，得证。 □

$T(2n) = O(T(n))$的假设不会过于苛刻，例如任意多项式均可满足。这个归约对确定一个下界刚刚好，但不建议实际用它来做乘法。定理 10.3 告诉我们不可能利用对称矩阵的对称性质使相乘算法更快。这里有另一个类似的归约：

□ **定理 10.4**

如果有算法在 $O(T(n))$时间内计算 $n\times n$ 实矩阵的平方，满足 $T(2n) = O(T(n))$，那么同样有算法在 $O(T(n)+n^2)$时间内计算两个 $n\times n$ 的任意实矩阵相乘。 □

证明：需要在定理 10.3 的证明中寻找一个矩阵，使之平方后足以获取两个任意矩阵的乘积。考虑下面的式子：

$$\begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}^2 = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}。 \tag{10.12}$$

立刻得证。 □

10.5　常见的错误

应该谨慎使用归约，下面就是一些试图使用归约而可能常犯的错误。最普遍的问题是以错误的顺序使用归约，这在应用归约于下界求取时最容易出现。归约必须建立在问题 P 至少与已知复杂度的问题 Q 同样难的情况之上，必须以 Q 的任意一个实例开始，并证明它能以 P 的一个黑盒解决方法来解决。比如说，考虑下面把用霍夫曼编码的数据压缩（见 6.6 节）归约成排序的尝试。目标是证明霍夫曼编码的复杂度下界为$\Omega(n \log n)$。

观察到如果字符的频率间隔甚远，则树将变得不平衡而可用于排序（见图10.3）。那样，字符在树中将以频率的降序出现（树顶端是出现频率最高的字符）。那就意味着霍夫曼编码可用于对频率进行排序。所以建树与排序至少是同等难度，下界$\Omega(n \log n)$似乎可证。

这个论据的错误源于用了排序的一个特例，仅考虑频率间隔甚远的情况。要证明排序的下界，必须以排序的任意一个实例开始。要知道，论据应该表明霍夫曼编码可用于完成任意的排序。应该证明任意一组数可用霍夫曼编码算法来排序，下一章将更深入地讨论该错误。

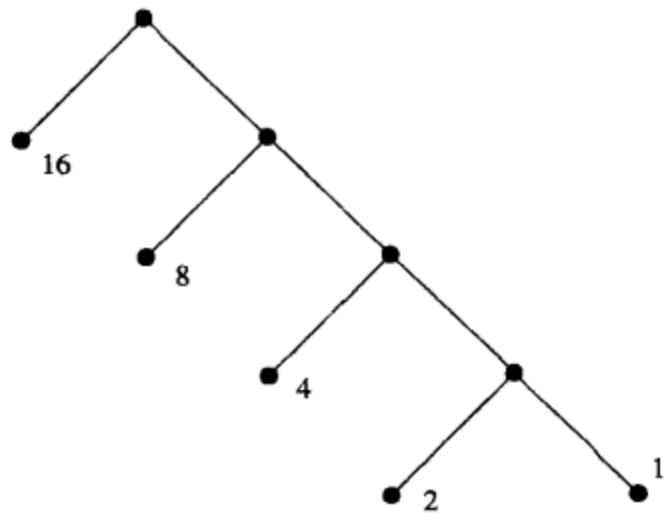


图 10.3　频率间隔甚远的霍夫曼树

问题既出，我们可以修改论据以拯救证明。诀窍是花点时间来修改排序问题的输入（当然是任意的）使之遵从我们的目标。设输入是互不相同的正整数序列 $X = (x_1, x_2, \cdots, x_n)$。可以假设这些数互不相同，因为排序的下界与数相同与否并无关系（事实上，下界也是针对不同的整数而证明的）。对应霍夫曼编码，等于 X 中的某个数的频率可以是任何一般的树，这样上面的论据就无法使用。但是可以把每个 x_i 替换成（比方说）$y_i = 2^{x_i}$，因为对任意正整数 m 有 $2^m > \sum_{i<m} 2^i$，霍夫曼树就可以有图 10.3 所示的形式，所以就能用霍夫曼编码算法来排序 y_i。现在必须确定归约中涉及的额外计算（从 x_i 计算 y_i 之类的）是可行的。幂运算的代价是非常昂贵的，但在这里没什么关系，因为排序下界仅涉及比较，不涉及数的任何其他操作（又见 6.4.6 节）。所以我们在决策树模型下确立了建霍夫曼树在最坏情况下需要Ω (n log n)的比较（不遵照决策树模型的算法来建树也有可能更快）。

我们同样必须小心对待归约过程而不应在其中加入明显低效的操作。考虑 5.11 节中的背包问题（knapsack problem），对它的扩展见 5.17 节（扩展解决每样物品可以无限次地加入背包的问题）。对扩展问题直观的归约为原问题（每样物品至多出现一次）如下：设背包容量为 K，大小为 s_i 的物品不能出现多于 K/s_i 次，所以可以把扩展问题中的每样物品替换成原问题的$\lfloor K/s_i \rfloor$项同样大小的物品。虽然这个归约是正确的，但并不十分有效，因为把问题的规模进行了可观的扩大。还可以更有效地解决这个问题。

10.6 小结

在问题间寻找相似点总是个好主意。通过学习两个问题间的异同点，常能获取对两个问题的内在领悟。给定一个新问题，第一个念头应该是（几乎在所有情况下）：这个问题与已知的某问题相似吗？有时只有经过复杂的归约，两个问题的相似处才变得明了。矩阵与图算法的归约显得特别有趣。本章中我们已经看到一些归约的例子，在下一章将看到更多的例子。

虽然很重要，但线性和整数归约在本章还是介绍得过于简要，有兴趣的读者应该详细而系统地进行学习。

参考文献和补充读物

Hall 定理源于 P. Hall[1935]，Hall 定理的详细讨论几乎可以在任何一本组合学书藉中找到，如 Brualdi[1977]或 M. Hall[1986]。寻找小的回路，如三角形，与矩阵相乘之间的关系在 Itai and Rodeh[1978]中有所讨论。关于矩阵的两个归约（见 10.4.2 节）源于 Munro[1971]。类似但更复杂，归约可以在布尔矩阵相乘、传递闭包问题（见 Fischer and Meyer[1971]）以及矩阵相乘和矩阵倒置（见 Winograd[1970b]）间进行。

线性规划首先由 Dantzig 在 1947 年解决（详细讨论及大量的例子见 Dantzig[1963]），又快又实用的单纯算法（simplex algorithm）从 20 世纪 50 年代就开始广泛应用。然而，Klee and Minty[1972]证明在最差情况下单纯算法的运行时间是指数级的。Khachian[1979]首先展示了一个线性规划算法，其在最差情况下的运行时间也是多项式的。Khachian[1979]的算法，也就是椭圆算法（ellipsoid algorithm），在实践中非常不尽如人意。但椭圆算法有另外一个应用（例如，见 Grötschel, Lovász, and Schrijver[1981]）。线性规划的另一个多项式时间的算法由 Karmarkar[1984]提出，由于它在某些情况下相对单纯算法所存在的优势，该算法得到了普遍关注，并引起了极大的研究热潮。更多关于线性和整数规划材料参见 Papadimitriou and Steiglitz[1982]以及 Schrijver[1986]。

习题 10.7 的答案可以在 van Leeuwen[1986]中找到；习题 10.8 来自 Even[1979]；习题 10.11 来自 Maggs and Plotkin[1988]；习题 10.22 来自 Aho, Hopcroft, and Ullman [1974]。

习题

10.1　使用在网络流量和双向匹配章节中得出的技巧证明 Hall 定理。

10.2　解决下列字符串比较问题的变形。输入是两个序列 A 和 B，目标是编辑 B 使之等于 A。编辑步骤是通常的插入、删除和替换（或匹配）。每一步的代价依赖于对应字符在序列中的位置。在 B 的第 i 位插入一个字符的代价是 ci，c 是一个常数；删除 B 的第 j 个字符的代价是 cj；替换一个字符的代价仍然是 1。算法要找出代价最小的编辑序列。

10.3　在以下两个问题间找到（某个方向的）一个归约：在平面上寻找极大点的问题（见习题 8.18），以及在数轴上区间包含问题（见习题 8.26）。

10.4　X 大学的 D 系进行了一次博士生考试，考试包括 Q 个问题，分成 n 个范围。这样在范围 i 里有 q_i 个问题（$\sum_{i=1}^{n} q_i = Q$），有 k 位教授 $P_1, P_2, \cdots, P_k$（当然不是他们的真名）出题。假设每位教授 P_i 共有 p_i 个问题可用，且 $\sum_{i=1}^{k} p_i \geqslant Q$。有一个委员会负责从教授们提供的题目中为考试选题，设所有的问题都不同，而且都很有价值。进一步假设每位教授强调他（她）的题目不能多于 r 题（r 是与教授无关的常数），这样他（她）可以在今后几年中使用剩余的题目。设计一个有效算法，在这些约束下选择考题，或者判断是否有解。

10.5　考虑下列双向匹配的变形。设有 $2n$ 名学生要入读 n 所大学，考虑这样形成的双向图：由学生和大学组成二组顶点，如果某大学同意接纳某学生，则连这样一条边。寻找一个算法使学生入学数最多，而每所大学不招收多于两名学生（不存在优先选择）。通过探索归约到常规双向匹配问题来求解。

10.6　另一个双向匹配的变形。设有 n 个教程及 n 位教练，如常，考虑把教程和教练作为顶点，以及如果教练适合某教程就连一条边这样一个图。每个教程有两个教练，且每个教练最多可以上两个教程。设计一个算法（归约到一个已知问题）使配对最大化（同样不存在偏好，没有时间安排问题）。

10.7　记 $G=(V, E)$是一个无向图，每个顶点 v 关联一个整数 $b(v) \leqslant degree(v)$。**b–匹配**（b-matching）就是满足所有顶点 v 都有不超过 $b(v)$条边与之相连的边 E 的集合（如果对所有的 v，$b(v)=1$，那正是常规匹配问题）。最大 b–匹配问题是使边数最大的问题，把寻找最大b–匹配问题归约成寻找最大匹配问题。

10.8　记 $G=(V, E)$是一个无环有向图。设计一个算法寻找包含 G 的所有顶点的，无顶点交集的路径的最小数。

10.9　记 $G=(V, E)$是一个源点为 s，汇点为 t 的网络。设 G 是平面的，顾名思义，它可以布在平面上而无相交的边，又设给定这样的布局（以合理的表示），且源点和汇点在布局的外部。不使用最大流量算法，设计一个算法在 G 中寻找最小代价的路径。

10.10　习题 8.40 可以归约为最小权重匹配而证明（这样，归约导致了一个更次的算法，因为最小权重匹配比直接求解该问题更难）。说明该归约并证其正确性（即证明对应的最小权重匹配满足本问题的条件）。

10.11　归约在无向图中寻找 MCST 的问题为**瓶颈最短路径问题**（bottleneck shortest-path problem，

瓶颈问题是一个极小化最大值的问题，而不是值的总和，所以瓶颈最短路径问题关心的是最大代价边的最小路径，而不是整条路径的代价）。作为结果，说明 MCST 问题可以由最短路径方法证明（虽然最短路径算法通常比 MCST 算法昂贵，但该归约可能对并行算法很有帮助）。

10.12 输入是一个带特殊顶点 v 的有向图 $G=(V,E)$，每个顶点 w 均关联一个正代价 $c(w)$，有向路径 $v, x_1, x_2, \cdots, x_k, u$ 的代价定义为 $\sum_{i=1}^{k} c(x_i)$，不计两个端点的代价。所以如果 $(v,u)\in E$，则从 v 到 u 的代价是 0。设计一个有效算法寻找从 v 到所有其他顶点的最小代价路径（本习题与习题 7.47 一样，但这里要求用归约来解决）。

10.13 比10.3节中给出的线性规划更一般的形式允许两种约束不等式的类型：第一种要求用“$\leqslant$”关系，第二种要求用“$\geqslant$”关系（当然系数不同）。证明该形式可以归约为 10.3 节的形式。

10.14 假设一个线性规划算法只能处理非负变量（回忆一下线性规划的定义，不限制所有的变量非负）。试述如何把一般问题归约成这个问题。

10.15 举一个反例，说明类似 $\overline{a}\cdot\overline{x}\neq b$ 的约束不应出现在线性规划公式中。

10.16 假设 n 个人出席一个科学会议，目标是最大化地交流想法。并非每个人都能与其他人交流。把会议表现成一个无向图，每个人与顶点关联，而如果 i 与 j 可以交流想法，则 i 与 j 相连（也可以定义成有向图的版本）。进一步假设发言的时间有限，为简单起见，设有一个共同的 h 小时限制，即每个人最多只能发言 h 小时，不关心时间安排。所以有充分的灵活性来安排任何可能的会议。比如说，有 3 个人，都与其他人“相连”，设 $h=1$。如果其中 2 个相互讨论了 1 个小时，则只算 1 个小时的对话；另一方面，如果 3 个人共同相互讨论了半个小时，那每个人都用光了他的时间，这是一个 1.5 小时的对话。现在要最大化总的对话时间，用线性规划表示这个问题，或者把它归约成讨论过的另一个问题。

10.17 再考虑 10.3.2 节中的慈善家问题。设每个系愿意接收的金额没有限制，请解决该问题。

10.18 考虑安排 n 名选手进行比赛，其顺序与循环竞赛（round-robin competition，见习题 7.104）一致。将该问题归约成排序问题来证明原问题的下界是$\Omega(n\log n)$，说明将该问题归约成排序问题对设计原问题的算法是很有帮助的。

10.19 设 S 是任意 n 凸多边形顶点的集合，点以任何顺序给出，证明需要用$\Omega(n\log n)$的时间把点排列为标准多边形表示法（即相邻顺序）。

10.20 在 9.5.2 节中已知 7 个乘法（而不是看上去的 8 个）就足够计算任意两个 2×2 的矩阵，且导出了更好的矩阵相乘算法，也可以用 5 个乘法计算一个 2×2 的实矩阵的平方（见习题 9.21）。讨论为何该结论不与定理 10.4 相矛盾？

10.21 下三角矩阵是一个方阵 a_{ij} 满足：若 $j>i$，则 $a_{ij}=0$（换句话说，所有非零项在主对角线上以及下方）。上三角矩阵的定义类似，只是非零项在主对角线上以及上方。试证，如果存在一个 $n\times n$ 上三角矩阵和 $n\times n$ 下三角矩阵相乘的算法，时间复杂度是 $O(T(n))$，那么存在一个时间复杂度是 $O(T(n)+n^2)$的任意两个 $n\times n$ 矩阵相乘的算法，可设对任意常数 c，$T(cn)=O(T(n))$。

10.22 试证，如果存在时间复杂度为 $O(T(n))$的两个 $n\times n$ 下三角矩阵相乘的算法，则存在时间复杂度为 $O(T(n)+n^2)$的任意两个 $n\times n$ 矩阵相乘的算法，可设对任意常数 c，$T(cn)=O(T(n))$。

10.23 一个 $n\times n$ 矩阵 A 的传递闭包 A^*定义为：

$A^*=I+A+A^2+\cdots+A^{n-1}$，其中 I 是 $n\times n$ 的单位矩阵。

a. 证明：如果 A 是一个对应图的邻接的布尔矩阵，那么 A^*就对应着图的传递闭包的邻接矩

阵（假设乘法是根据布尔运算法则来进行的）。

b. 证明：如果传递闭包能在$T(n)$时间内算出，$T(n)$是 n 的一个多项式，那么矩阵乘法可以在$O(T(n))$内算出。可假设对任何常量 c，$T(cn) = O(T(n))$。

10.24 设 S 是平面上 n 个点的集合，这些点这样定义一个加权无向图：图是一个完全图（任意两个顶点都相连），边的权重等于两点间的几何距离。试证 MCST 算法在该例中运行的下界是 $\Omega(n \log n)$。

*10.25 设 S 是平面上 n 个点的集合，S 的**直径**（diameter）是 S 中相距最远的两点间的距离，用 *DM* 来表示该问题。设 A 和 B 分别是两个由 n 个实数组成的集合，用 *DJ* 表示判断 A 和 B 是否不存在交集的问题。证明如果存在 $O(T(n))$个算术运算（可以设想任意合理的运算）的 *DM* 算法，那么必然存在 $O(T(n)+n)$个算术运算的 *DJ* 算法。

第 11 章　NP 完全问题

给我一个支点，我就可以撬动地球。
——阿基米德（公元前 287–212）

11.1　引言

本书前几章主要研究了算法技术及其在具体问题上的应用，本章内容有较大的不同。对于任何问题来说，如果应用少数技术就能得到优雅而高效的算法该有多好，但现实并非如此，有许多问题似乎并不能应用目前我们已经掌握的那些技术加以解决。诚然，有可能是我们未尽全力，但也很可能确实存在这样一类问题，它们并没有通用和高效的解决方法。本章就将描述用于识别这类问题的技术。

迄今为止所讨论的大多数算法其运行时间与输入大小的某个多项式函数相关，一般称这样的算法是高效的（efficient），相应的问题是易处理的（tractable）。换句话说，如果算法的运行时间是 $O(P(n))$，这里 $P(n)$是输入大小 n 的多项式函数，那么可称该算法是高效的。我们用 P（指多项式时间）表示所有可被高效算法解决的问题类。这似乎是一个奇怪的定义，毕竟不论是按照何种标准，运行时间为 $O(n^{10})$的算法并不高效（运行时间为 $10^7 n$ 的算法也不是高效的，尽管它是线性的）。尽管如此，鉴于以下两种原因，该定义不仅合理而且有效：首先，从它可发展出一套理论模型；其次（也是最重要的），在实践中它是有效的，事实表明绝大多数易处理问题都存在现实的算法（当然，其中的一些算法要优于另外一些）。换句话说，我们在现实中所遇到过的多项式算法的运行时间通常都是次数较小的多项式（很少有超过 4 次的），相反的情况也总成立：运行时间超过多项式的算法通常对于大的输入并不是实际可用的。

存在着许多问题，迄今为止多项式时间的算法还是个未知数，其中一些问题的高效算法仍有待发现，但我们强烈怀疑许多问题不可能有高效的解决方法。希望能够识别这些问题，从而不必浪费时间去寻找不存在的算法。在这一章，我们将探讨如何处理不被认为是属于 P 的问题，尤其是一类被称为 NP 完全问题的特殊例子。将这些问题归为一类是因为它们在如下意义下是等价的，即某个 NP 完全问题存在着高效算法当且仅当所有的 NP 问题都存在着高效算法。普遍的观点是 NP 完全问题不存在高效算法，但却没有人能够找到该信念的证明。即使存在解决 NP 完全问题的高效算法，但由于研究者寻求了许多年都未曾发现，它们也一定非常复杂。迄今为止，数以百计（甚至可能是数以千计）的问题被证明是 NP 完全的，这也是该主题如此重要的原因。

本章由两部分组成：首先定义 NP 完全问题类，并说明怎样证明一个问题属于该类；接着提出一些近似解决 NP 完全问题的技术及相应例子，这些解决方法可能不是最优的并且有可能会失效，但总比没有解决方法要好。

11.2　多项式时间归约

在这一节中我们将范围限定在判定问题（decision problem），即只考虑那些答案为肯定或者否

定的问题，这样的限定可以简化讨论以及相应的理论。大多数问题都可以转化成判定问题，例如可以将寻找给定图的最大匹配问题转化为判定是否存在一个大于或等于 k 的匹配，如果我们知道怎样解决该判定问题，通常就可以解决原先的问题，如通过二分查找法。

判定问题可以视为语言识别问题，假设 U 是由该判定问题的各种可能的输入所组成的集合，而 $L \subseteq U$ 是回答为肯定的全部输入所组成的集合，一般称 L 为与该问题相关的语言。这里不加区分地使用“问题”和“语言”这两个术语是因为判定问题其实就是识别给定的输入是否属于 L。现在引入语言间多项式时间归约的概念，这是本章用到的主要工具。

定义：设 L_1 和 L_2 分别是输入空间 U_1 和 U_2 的两个语言，如果存在一个多项式时间的算法可以将每个输入 $u_1 \in U_1$ 转化成另一个输入 $u_2 \in U_2$，使得 $u_1 \in U_1$ 当且仅当 $u_2 \in U_2$，则称 L_1 可以多项式归约到 L_2。该算法是 u_1 的大小的多项式函数。我们假定“大小”的概念在输入空间 U_1 和 U_2 中是合适定义的，特别是，u_2 的大小是 u_1 的大小的多项式函数。

上述定义中的算法可以将一个问题转化为另一个问题，如果有了一个针对 L_2 的算法，那么就可以将这两个算法组合起来生成一个针对 L_1 的算法。若将该转化算法表示成 AC，而用 AL_2 表示针对 L_2 的算法，则对于任意输入 $u_1 \in U_1$，可以利用 AC 将其转化成输入 $u_2 \in U_2$，接着就可以使用算法 AL_2 来判定 u_2 是否属于 L_2，而这将告诉我们 u_1 是否属于 L_1。特别是有以下定理：

□ **定理 11.1**

如果 L_1 可以多项式归约到 L_2 并且 L_2 有一个多项式时间的算法，那么 L_1 也存在一个多项式时间的算法。

证明：从以上讨论可以直接得出该结论。　□

归约的概念并不是对称的，事实上，L_1 可以多项式归约到 L_2 并不意味着 L_2 也可以多项式归约到 L_1，不对称性的原因是归约的定义要求任何 L_1 的输入都可以被转化成一个等价的 L_2 的输入，反之并不要求。在许多例子中，与归约相关的 L_2 的输入完全有可能只是 L_2 所有可能输入的一小部分，由此，如果 L_1 可以多项式归约到 L_2，L_2 将被视为更难的问题。

如果两个语言 L_1 和 L_2 可以多项式相互归约到对方，则称它们是多项式等价的，或简言之等价的。特别是，所有非平凡的易处理问题都有多项式时间算法，因此它们都是等价的（这个事实的证明细节留做练习）。正如以下定理所示，“多项式归约”关系具有传递性。

□ **定理 11.2**

如果 L_1 可以多项式归约到 L_2 并且 L_2 可以多项式归约到 L_3，那么 L_1 可以多项式归约到 L_3。

证明：可以将两个转换算法结合在一起，形成一个从 L_1 到 L_3 的转换算法。L_1 中的输入 u_1 将首先转化为 L_2 中的输入 u_2，然后再转化为 L_3 中的输入 u_3。由于我们所使用的是多项式归约并且两个多项式函数的复合函数仍然是一个多项式函数，因而结果仍然是一个多项式时间的转换算法。（这也是我们选择使用多项式函数的一个原因。）　□

本章所给出的方法的精髓在于当一个问题的高效算法无法找到时，可转去寻求是否存在与之等价的问题。当我们无法找到给定问题的高效算法时，可尝试这个问题是否可与一个已知的困难问题等价。NP 完全问题类包括数以百计彼此等价的问题。

11.3 非确定性和 Cook 定理

通过著名的 Cook 定理[1971]创立了 NP 完全性理论。在叙述这个定理之前，我们先解释一些概念，为了使讨论更直观，这里将忽略掉若干技术细节，有兴趣的读者可参阅 Garey and Johnson [1979]。NP 完全性理论是庞大的计算复杂度理论的一个组成部分，后者的大部分内容已经超出了本书的范畴，本书仅讨论那些有助于理论应用的部分。

用精确的数学术语来描述什么是算法需要大量的技术细节，虽然我们至今尚未涉及，但这对于描述具体的算法并不重要，通常只要能合理地使用计算机所提供的单步操作（例如加法、比较操作和存储器访问等）即可，而对于证明下界问题，确切的算法定义是非常重要的（第 6 章中曾使用判定树证明下界问题，但这并不是一个非常严格的模型）。最基本的计算模型是图灵机，另一个通常使用的模型叫做随机存储机。幸运的是，对于本书的目标而言，它们和其他一些合理的模型都是彼此等价的，这是因为我们可以在不超过多项式因子的运行时间内将一个模型上的算法转化成另一个模型上的算法，例如 Cook 定理是通过图灵机证明的，而对于其他模型也是成立的。本章不使用任何特定的模型，因为我们不打算涉及它们的具体细节。

首先需要讨论的是非确定性的概念，由于这个概念非常不直观，因此许多人对 NP 完全性感到很神秘。事实上应当把非确定算法视为一个抽象的概念而非实际目标，非确定性对于计算机理论的发展和解释存在这类问题的重要程度超过其技术的应用。除了具备所有常规确定性算法的操作之外，非确定算法有一个非常强的基本操作（primitive），称之为 nd-选择（nd-choice）。正如其名，nd-choice 用来处理选择，该基本操作与固定数目的一些可选项结合在一起使用，对于每一次选择，算法都会沿着不同的路径继续。不失一般性，我们可以假设备选的数目总是两个。给定输入 x，非确定算法交织执行常规的确定性步骤和 nd-choice 操作，最终决定是否接受 x，确定性和非确定性算法最突出的差异在于它们识别语言的方式上。

如果满足以下条件，则定义一个非确定算法能识别的语言 L：

> 给定一个输入 x，$x \in L$ 当且仅当可以将每一次算法执行过程中的 nd-choice 转变成实际的选择，使得算法能够接受 x。

换句话说，对于属于 L 的输入，算法必须提供至少一条可能的计算路径到达接受状态；而对于那些不属于 L 的输入，算法不可以有任何到达接受状态的计算路径。注意到定义中的不对称性，对于输入 $x \in L$，算法可以有多条到达拒绝状态的路径，对于每个 $x \in L$ 算法只需要确保至少存在一个“好”的选择序列即可；另一方面，对于输入 $x \notin L$ 无论如何进行 nd-choice，算法都必须要到达拒绝状态。显然，nd-choice 操作有时被称为猜测。对于输入 $x \in L$，算法的运行时间是指到达接受状态的最短执行序列，非确定算法的运行时间是指在最坏情况下对于输入 $x \in L$ 的运行时间（不属于 L 的输入被忽略）。

看一个非确定算法的例子，对于给定的图 $G=(V, E)$，判定它是否存在完美匹配。以下是该问题的一个非确定算法，维护一个初始为空的边集 M，检查 G 的所有边，每次检查一条边 e，使用 nd-choice 决定是否将 e 归入 M，当所有的边都检查完之后就可以验证 M 是否为一个完美匹配。由于只要判定 M 是否恰好含有 $|V|/2$ 条边，并且每个顶点是否恰好与 M 中的一条边关联，因此验证可以在线性时间内完成。如果 M 是一个完美匹配，则算法将输出肯定的结果，反之则输出否定的结果。这是完美匹配问题的非确定算法，其正确性在于：（1）如果存在完美匹配，则存在一个选择序列将该完美匹配放入 M 中；（2）若算法输出了肯定结果，则存在完美匹配。下一节将看到更多关于非确定算法的例子。

非确定算法具有非常强的能力，但并非无限，并不是所有的问题都可以被非确定算法高效地解决。例如，假定问题是判定给定图的最大匹配大小是否恰为 k，如果存在大小为 k 的匹配，则可以使用非确定算法找到它，但是我们不能轻易判定（即使是以非确定的方式）不存在更大的匹配。

存在运行时间关于输入大小是多项式函数的非确定算法的这些问题所组成的类被称为 NP 类，认为非确定算法比确定性算法更强大似乎是合理的，但是果真如此吗？证明方法之一是列举一个 NP 问题并证明它不在 P 中，但至今为止没有人能够做到这一点。相反，如果想要证明这两类问题是重合的（即 P = NP），那么我们就不得不证明所有在 NP 中的问题都可以被一个多项式时间的确定性算法解决，同样也没有人能够证明这一点（也很少有人相信这是正确的）。决定 P 和 NP 之间关系的问题被称为 P = NP 问题。

现在定义两类问题，包括许多重要但未知是否属于 P 的（相互等价的）问题和 NP 中最难的问题。

定义：如果所有 NP 中的问题都可以多项式归约到问题 X，则称 X 为 NP 难（NP-hard）问题 。

定义：如果问题 X 满足：（1）X 在 NP 中；（2）X 是 NP 难的，那么称 X 为 NP 完全问题。

由 NP 难的定义可得，任何一个 NP 难问题若被证明属于 P，那么同时也就证明 P = NP。

Cook[1971]不仅证明了 NP 完全问题的存在，还具体给出了这样的一个问题，我们将在稍后描述此具体问题。一旦找到了一个 NP 完全问题，证明其他问题的 NP 完全性也将变得容易，给定一个新问题 Y，只要能够证明 Cook 的那个具体问题或者其他任何 NP 完全问题可以多项式归约到 Y 已足够，主要的依据就是以下引理。

□ 引理 11.3

如果问题 X 满足：（1）X 属于 NP；（2′）对于某个 NP 完全问题 Y，Y 可以多项式归约到 X，那么 X 是 NP 完全问题。

证明：根据 NP 完全性定义的条件（2），每一个 NP 中的问题都可以多项式归约到 Y，又因为 Y 可以多项式归约到 X 以及归约关系具有传递性，所以每一个 NP 中的问题也都可以多项式归约到 X。 □

证明两个问题可以多项式归约要比直接证明前面的条件（2）容易得多。由此可见 Cook 为整个理论奠定了基础。所幸的是，随着越来越多的 NP 完全问题被发现，对于条件（2′）的证明也有了更多的选择。Cook 的结论公布不久，Karp [1972]发现了 24 个被他证明是 NP 完全的重要问题。从那时起，数以百计的问题（甚至可能是数以千计，取决于对同一问题不同变形的计数方式）被发现是 NP 完全的。下一节将介绍 5 个 NP 完全问题及其 NP 完全性的证明，同时也不加证明地列举其他一些 NP 完全问题。这些证明的最难部分通常是（但非总是）验证条件（2）或（2′）。

现在描述被 Cook 证明是 NP 完全的问题，并略述其证明思想，这个问题又称可满足性问题（SAT）。设 S 是一个用合取范式（CNF，conjunctive normal form）表示的布尔表达式，也就是说 S 是一些和（“or”）的积（“and”），例如，$S = (x + y + \overline{z}) \cdot (\overline{x} + y + z) \cdot (\overline{x} + \overline{y} + \overline{z})$，这里加法和乘法分别对应于布尔操作中的“or”和“and”，而每个变量取 0（假）或者取 1（真）（任何一个布尔表达式都可以化成 CNF）。如果存在使整个表达式取值 1 的布尔变量的 0、1 赋值，则称该布尔表达式是可满足的。SAT 问题就是判定所给定的表达式是否可满足（不必找到使之满足的赋值）。

例如上述表达式 S 就是可满足的，因为存在赋值 $x=1$、$y=1$ 和 $z=0$ 使之满足。我们称对布尔表达式变量的 0、1 赋值为真值指派。

由于我们可以猜测一个真值指派并且在多项式时间内验证它是否满足表达式，故 SAT 问题在 NP 中。证明 SAT 问题是 NP 难的思想在于，图灵机（甚至是不确定型的）以及其他在给定输入上的操作可以由一个布尔表达式来描述。在这里“描述”是指表达式可满足当且仅当对于给定的输入，图灵机将停止在接受状态。这可不是能容易做到的，因为这样的表达式非常冗长而且相当复杂，但是它的大小仍然不会超过图灵机运行步骤数的某个多项式函数。由此，任何一个 NP 算法都可以被一个 SAT 问题的实例所描述。

□ **Cook 定理**

SAT 问题是 NP 完全的。 □

11.4 NP 完全性的证明例子

在这一节中我们将证明以下 5 个问题是 NP 完全的：顶点覆盖问题、支配集问题、3SAT 问题、3 着色问题以及团（clique）问题，下面将详细描述这些问题。证明 NP 完全性所使用的技术都是典型的，这将在本节的最后加以总结。要证明一个新问题是 NP 完全的，首先要证明其属于 NP，这通常是（但并非总是）容易的，接着我们要将一个已知的 NP 完全问题在多项式时间内归约到这个新问题。这 5 个问题的归约顺序如图 11.1 所示，为了便于理解，证明并非按照树的顺序而是按照难易顺序进行的。

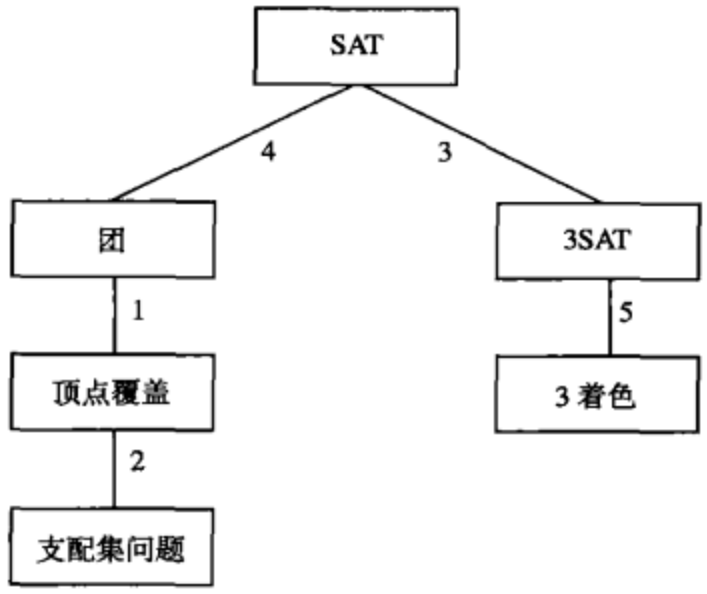


图 11.1 本书中 NP 完全性问题例子的证明顺序

11.4.1 顶点覆盖问题

设 $G=(V, E)$是一个无向图，G 的顶点覆盖是一个顶点集合，满足 G 中所有的边都至少与该集合中的一个顶点相关联。

问题 给定无向图 $G=(V, E)$和一个整数 k，判定 G 是否有包含小于或等于 k 个顶点的顶点覆盖。

□ **定理 11.4**

顶点覆盖问题是 NP 完全的。

证明：由于可以猜想一个小于或等于 k 的顶点集合并且很容易在多项式时间内验证它是否是顶点覆盖，故顶点覆盖问题属于 NP。要证明顶点覆盖问题是 NP 完全的，需要将一个 NP 完全问题归约到它，这里选取团问题并加以描述（团问题的 NP 完全性证明将在 11.4.4 节中给出）。给定无向图 $G=(V, E)$，G 中的团 C 是 G 的一个子图，满足 C 中的任何两个顶点均相邻，换句话说，团即完全子图。团问题是指给定图 G 和整数 k，判定 G 是否包含一个大于或等于 k 的团。我们需要将任意一个团问题的实例转变成顶点覆盖问题的实例，使得团问题的回答是肯定的当且仅当相应顶点覆盖问题的回答也是肯定的。设 $G=(V, E)$和 k 代表任意一个团问题的实例，并设 $\overline{G}=(V, \overline{E})$ 是图 G 的补图；也就是说 $\overline{G}$ 和 G 有相同的顶点集合，并且两个顶点在 $\overline{G}$ 中相邻当且仅当它们在 G 中不相邻。下面说明该团问题可被归约到由图 $\overline{G}$ 和 $n-k$（其中 n 是 G 中的顶点数）所代表的顶点覆盖问题，假定 $C=(U, F)$是 G 中的一个团，由于 $\overline{G}$ 中没有连接 U 中顶点的边（它们都在 G 中），顶点集合 $V-U$ 覆盖了 $\overline{G}$ 中的所有边，这样 $V-U$ 就构成了 $\overline{G}$ 的一个顶点覆盖。所以若 G 有大小为 k 的团，$\overline{G}$ 就有一个大小为 $n-k$ 的顶点覆盖。反之，假设 D 是 $\overline{G}$ 的顶点覆盖，D 覆盖了 $\overline{G}$ 中所有的边，则在 $\overline{G}$ 中没有边连接 $V-D$ 中的顶点，也就是说 $V-D$ 构成了 G 中的一个团。因此如果在 $\overline{G}$ 中有大小为 k 的顶点覆盖，那么在 G 中必有一个大小为 $n-k$ 的团。由于这个归约只须从 G 构造 G'（以及计算 $n-k$），显然它可以在多项式时间内完成。　□

11.4.2　支配集问题

设 $G=(V, E)$是一个无向图，如果顶点集合 D 满足，G 中的所有顶点要么在 D 中，要么与 D 中至少一个顶点相邻，则称 D 为支配集。

> **问题**　给定无向图 $G=(V, E)$和整数 k，判定 G 中是否有一个包含小于或等于 k 个顶点的支配集。

□ **定理 11.5**

支配集问题是 NP 完全的。

证明：由于可以猜想一个小于或等于 k 的顶点集合并很容易在多项式时间内验证它是否是一个支配集，故支配集问题属于NP。下面将顶点覆盖问题归约到支配集问题，给定任意顶点覆盖问题实例 (G, k)，现在的目标是构造一个新图 G' 满足：G' 有一个特定大小的支配集当且仅当 G 有一个小于或等于 k 的顶点覆盖。从 G 出发按以下方式（见图 11.2）添加$|E|$个新顶点和 $2|E|$条新边，对于 G 中的每一条边 (v, w)，添加一个新顶点 vw 以及两条新边(v, vw)和(w, vw)。换句话说，将每条边转化成一个三角形。用 G' 表示新图，可以很容易在多项式时间内构造 G'。

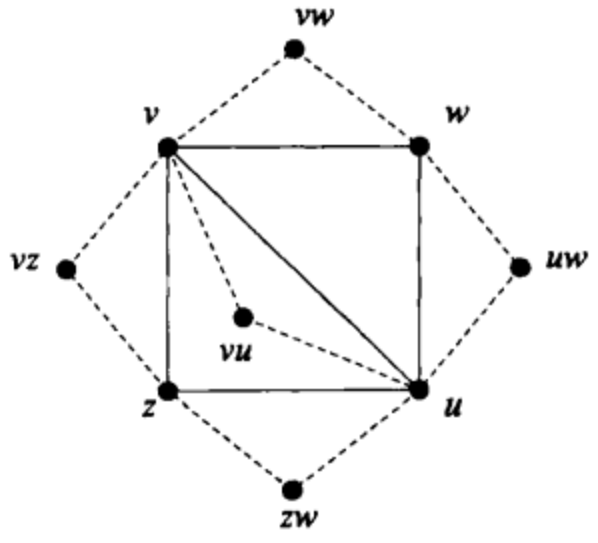


图 11.2　支配集归约

现在说明 G'有一个大小为 m 的支配集当且仅当 G 有一个大小为 m 的顶点覆盖。假设 D 是 G'的一个支配集，如果 D 含有新顶点 vw，则它可以被 v 或 w 替换并且仍为一个支配集（因为顶点 v 和 w 都能覆盖 vw 所覆盖的顶点）。因此不失一般性，可以假定 D 仅包含 G 中的顶点。由于 D 支配了所有的新顶点，它一定至少包含原图中每条边的两端顶点之一，因此它也是 G 的一个顶点覆盖。反之，如果 C 是 G 的一个顶点覆盖，由于所有边都被 C 覆盖，所有新顶点都被 C 所支配，同理可得，所有旧顶点也都被 C 所支配。 □

11.4.3 3SAT 问题

3SAT 问题是一般 SAT 问题的简化，3SAT 的实例是指每个子句中恰好含有 3 个变量的布尔表达式。

> **问题** 给定以 CNF 形式出现并且每个子句恰好含有 3 个变量的布尔表达式，判定其是否可满足。

□ **定理 11.6**

3SAT 问题是 NP 完全的。

证明：由于存在每个子句含有 3 个变量的额外要求，这个问题看上去比一般 SAT 问题更容易，我们将证明解决 3SAT 的方法可用来解决一般的 SAT 问题。首先 3SAT 问题属于 NP，我们可以猜测一个真值指派，并在多项式时间内验证其满足这个表达式。设 E 是任意一个 SAT 问题实例，我们将 E 的每个子句用若干恰好含有 3 个变量的子句替换。设 $C=(x_1+x_2+\cdots+x_k)$是 E 的任意子句（其中 $k\geqslant 4$），为了便于表示，我们用“正出现”形式表示每个变量。现在说明如何用若干只含 3 个变量的子句来替换 C，其思想是在不影响可满足性的前提下引入新变量 $y_1, y_2, \cdots, y_{k-3}$将子句 C 转化成 3SAT 模式，对于每一个子句使用新（并且不同）的变量。转化后的 C' 如下所示：

$$C'=(x_1+x_2+y_1)\cdot(x_3+\overline{y_1}+y_2)\cdot(x_4+\overline{y_2}+y_3)\cdots(x_{k-1}+x_k+\overline{y_{k-3}})。$$

现在说明 C'是可满足的当且仅当 C 也是可满足的。如果 C 是可满足的，那么 x_i之一将被赋值为 1，于是我们就可以指派 C'中 y_i的值使得所有 C'中的子句也都可满足。例如，若 $x_3=1$，则可以指派 $y_1=1$（考虑满足第一个子句），$y_2=0$（由于 $x_3=1$，第二个子句也满足），并将其余 y_i都指派为 0。一般来说，如果 $x_i=1$，则指派 $y_1, y_2, \cdots, y_{i-2}$为 1，其余指派为 0，这样将满足 C'。反之，如果 C'是可满足的，则说明至少有一个 x_i 必是 1。如若不然，x_i 都是 0，那么表达式将变成 $(y_1)\cdot(\overline{y_1}+y_2)\cdot(\overline{y_2}+y_3)\cdots(\overline{y_{k-3}})$，该表达式显然是不可满足的。

使用这个归约策略，可以将任何含有超过 3 个变量的子句用恰好含有 3 个变量的子句替换掉。接着需要转化含有一个或两个变量的子句，如果 C 仅有两个变量，即 $C=(x_1+x_2)$，那么

$$C'=(x_1+x_2+z)\cdot(x_1+x_2+\overline{z}),$$

其中 z 是新变量。最后若 $C=x_1$，那么

$$C'=(x_1+y+z)\cdot(x_1+\overline{y}+z)\cdot(x_1+y+\overline{z})\cdot(x_1+\overline{y}+\overline{z}),$$

其中 y 和 z 都是新变量。

至此，我们已将一般 SAT 问题的实例归约成 3SAT 问题的实例，并且保证其中一方可满足当且仅当另一方也可满足。显然，这样的归约可以在多项式时间内完成。 □

11.4.4　团问题

在 11.4.1 节讨论顶点覆盖问题时已经对团问题进行了定义。

> **问题**　给定无向图 $G=(V,E)$和整数 k，判定 G 是否包含一个大于或等于 k 的团。

□ **定理 11.7**

团问题是 NP 完全的。

证明：由于我们可以猜想一个大于或等于 k 的顶点子集并在多项式时间内验证它构成一个团，故团问题属于 NP。现在将 SAT 问题归约到团问题，设 E 是任意一个 CNF 形式的布尔表达式，$E=E_1\cdot E_2\cdot\cdots\cdot E_m$，考察子句 $E_i=(x+y+z+w)$（这里使用 4 个变量只是为了便于阐述），使用 4 个顶点的“列”来代表 E_i 中的变量，即使这些变量在其他子句中也存在，也就是说对于每个变量的任意一次出现，图 G 均有一个顶点与之对应。问题是怎样将这些顶点连接起来使得 G 含有大于或等于 k 的团当且仅当 E 是可满足的，注意这里对于 k 值的选取没有限制，这是因为我们想要将 SAT 问题归约到团问题，即用解决团问题的方法来解决 SAT 问题，而解决团问题的方法需要对所有 k 值均有效。这是在 NP 完全性证明中常用到的一种重要技巧，我们将 k 选取为子句的个数 m。

按照如下方式定义 G 中的边：同一列中的顶点（即相同子句中变量对应的顶点）没有边相连，不同列中的顶点总有边相连，除非它们对应于以互补形式出现的相同变量，也就是说不连接不同子句中的两个顶点的惟一情况是，一个对应变量 x 而另一个对应变量 $\overline{x}$。$E=(x+y+\overline{z})\cdot(\overline{x}+\overline{y}+z)\cdot(y+\overline{z})$，对应的例子如图 11.3 所示，$G$ 显然可以在多项式时间内构造出来。

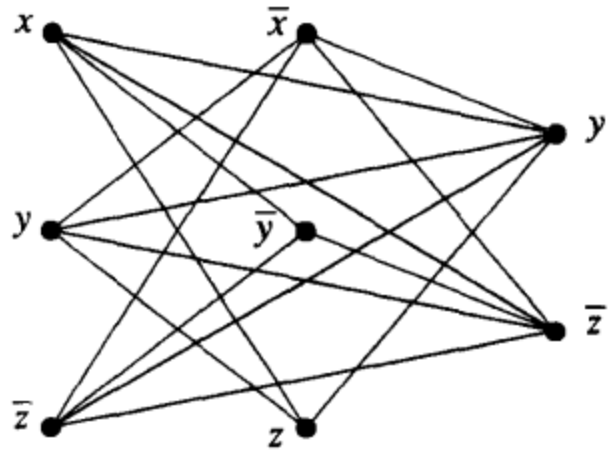


图 11.3　$(x+y+\overline{z})\cdot(\overline{x}+\overline{y}+z)\cdot(y+\overline{z})$ 所对应的团归约的例子

现在说明 G 有一个大于或等于 m 的团当且仅当 E 是可满足的。事实上，在构造中已经确保最大团的大小不可能超过 m，而这与 E 满足与否无关。假定 E 是可满足的，那么存在一个真值指派使得每个子句中至少有一个变量被赋值为 1，我们就选取与这个变量对应的顶点来构造团（如果在子句中有超过一个被赋值为 1，则任选其一）。由于不同列中两个顶点没有边相连的惟一情况是它们对应的变量是互补的，而这当然不可能出现在一致的真值指派中，因此这样构造的结果确实构成一个团。相反，假定 G 包含一个大于或等于 m 的团，该团在每列中一定恰好只含一个顶点（因为相同列中的两个顶点一定不会相邻），我们将对应的变量赋值为 1。如果有变量并未以这种方式被指派，则它可以被任意赋值。由于所有团中的顶点都彼此相邻，可以确保 x 和 $\overline{x}$ 不会相邻，故该指派是一致的。　□

11.4.5 3 着色问题

设 $G=(V,E)$是无向图，G 的有效着色是指对所有顶点的颜色指派，使得每个顶点被指派一种颜色并且相邻顶点不被指派成相同颜色。

问题 给定无向图 $G=(V,E)$，判定 G 是否可以被 3 种颜色着色。

□ **定理 11.8**

3 着色问题是 NP 完全的。

证明：由于可以猜想一个 3 着色并且很容易在多项式时间内验证它是一个有效着色，故 3 着色问题属于 NP。接着，将 3SAT 问题归约到 3 着色问题，基于两方面原因这是一个比较复杂的证明。首先，两个问题所处理的对象不同（布尔表达式对应于图）；其次，不能只将一个对象（如顶点和边）用另一个对象（如子句）来替换，我们必须要处理整个结构。证明思想是构造基本块，然后将它们连接在一起。设 E 是任意一个 3SAT 问题的实例，我们要构造一个图 G，使得 E 可满足当且仅当 G 可以 3 着色。第 1 步，构造主三角形 M，由于 M 是三角形，它至少需要 3 种颜色。我们用“颜色”T（代表真）、F（代表假）和 A（见图 11.4 中底部的三角形）来标注 M，这些颜色仅用来证明，并不是图的一部分，我们稍后将把这些颜色与 E 中变量的真值指派结合在一起。对于每个变量 x 构造另一个三角形 M_x，它的顶点被标注为 x、$\bar{x}$ 和 A，这里的 A 即 M 中的顶点 A。所以，如果有 k 个变量就有 k+1 个共享相同顶点 A 的三角形（见图 11.4）。其构造思想是，如果 x 被着色成颜色 T，那么 $\bar{x}$ 一定要被着色成 F（由于它们都与 A 相邻），反之亦然，而这与 $\bar{x}$ 的含义是一致的。

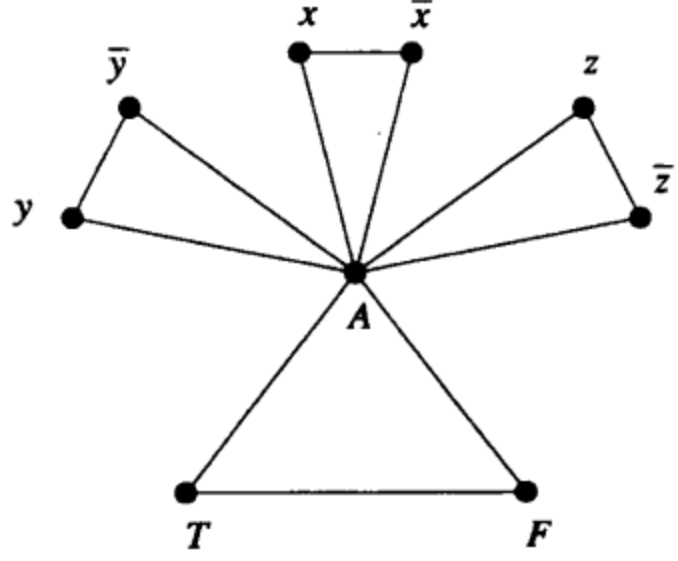


图 11.4 从 3SAT 问题到 3 着色问题的归约构造的第一部分

现在将每个子句中至少有一个变量值为 1 的条件用以下方式添加上。假定子句是 x+y+z，引入 6 个新节点并将它们连接到现有顶点，如图 11.5 所示。（这些标注是一致的，以便在整个图中只有一个顶点被标注为 T，每个 x、y 和 z 也仅标注一个顶点。）将连接到 T 以及 x、y 或 z 的 3 个新顶点称为外部顶点（在图中用 O 标注），而将三角形中的 3 个新顶点称为内部顶点（在图中用 I 标注）。下面说明该构造确保如果所使用的颜色不超过 3 种，那么至少 x、y 和 z 中的一个将被着色为 T。由于它们都与 A 相邻（见图 11.4），所以均不能与 A 着色相同。如果它们都被着色成 F，则与它们相邻的 3 个新顶点必须与 A 着色相同，这样内部的三角形就不能被 3 着色！图 11.6 给出了 $(\bar{x}+y+\bar{z})\cdot(\bar{x}+\bar{y}+z)$ 所对应的完整图。

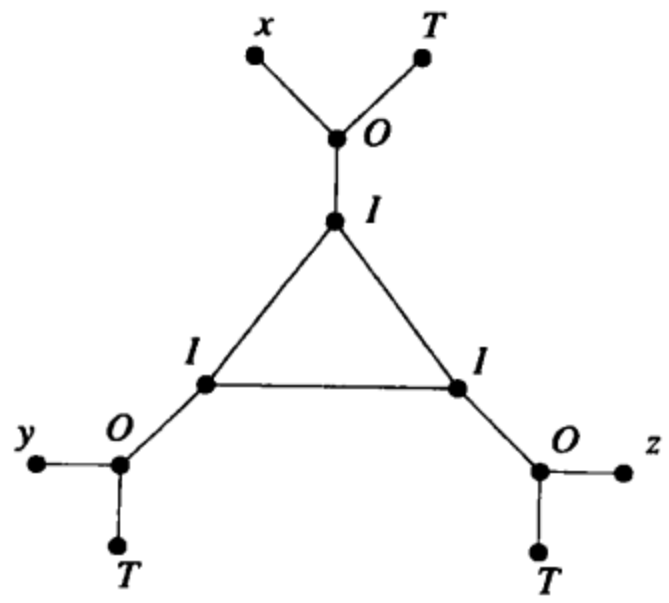


图 11.5　从 3SAT 问题到 3 着色问题归约中与子句 $x+y+z$ 所对应的子图

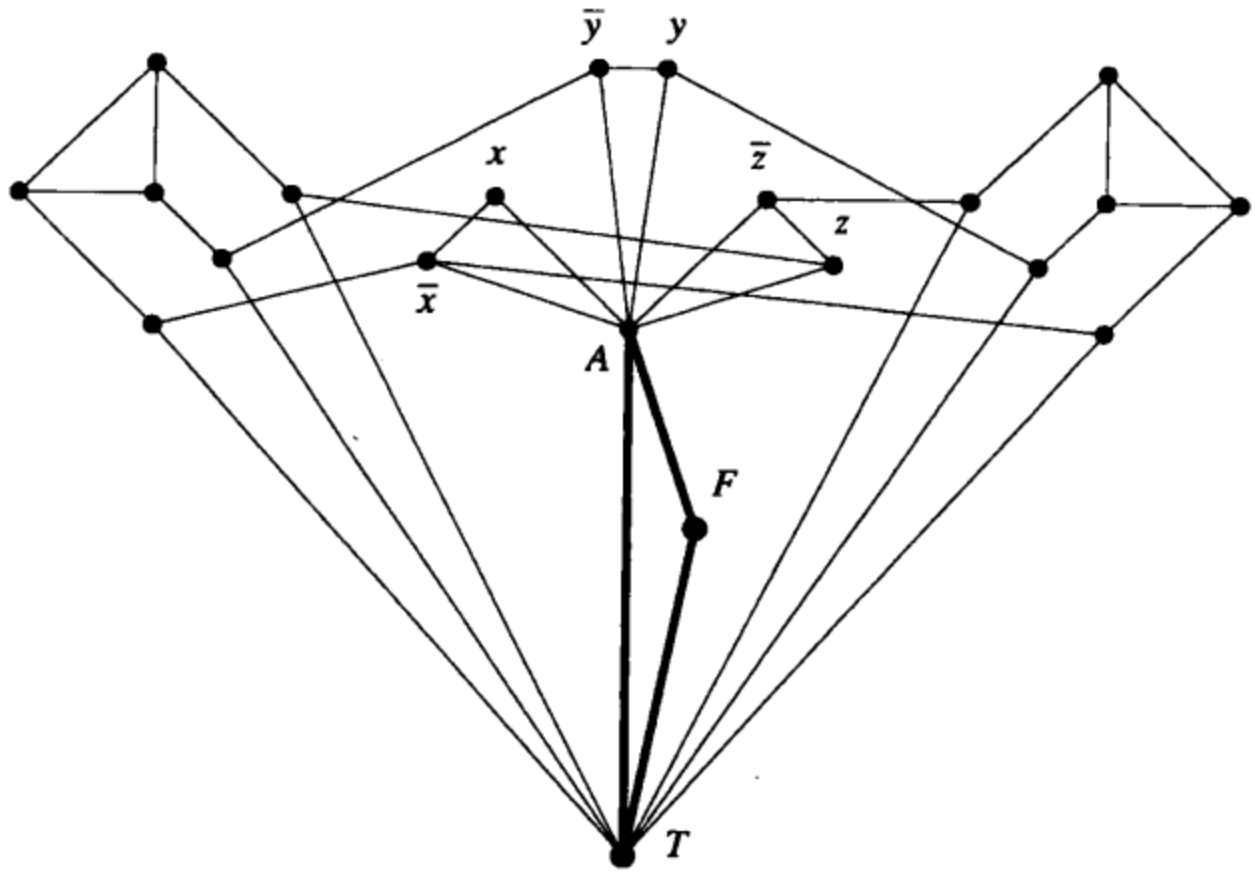


图 11.6　$(\overline{x}+y+\overline{z})\cdot(\overline{x}+\overline{y}+z)$ 所对应的图

在完成整个任务之前还需要证明两点：（1）如果 E 可满足，则 G 可以 3 着色；（2）如果 G 可以 3 着色，则 E 可满足。如果 E 可满足，则存在使其满足的真值指派，可以结合这个真值指派给顶点着色。根据标注对三角形 M 着色为 T（$x=1$ 时）、F 和 A，每个子句至少有一个变量被赋值为 1，我们将相应的外部顶点着色为 F，其余外部顶点着色为 A，并相应地对内部节点进行着色，这样 G 可以 3 着色。相反，若 G 可以 3 着色，则可根据 M 中的顶点命名颜色（M 必可 3 着色）。根据图 11.4 中的三角形，变量的着色对应于一个一致的真值指派，而图 11.6 的构造确保了每个子句中至少有一个变量被着色为 T。最后，G 显然可以在多项式时间内被构造出来，这也就完成了所有证明。　□

11.4.6　一般经验

下面简要讨论一下证明一个问题 Q 的 NP 完全性所使用的一些通用方法。首先，证明 Q 属于 NP，这通常是（但非总是）简单的。接着要选择一些看上去与 Q 相关或者类似的已知 NP 完全问

题，但有时问题看上去差异很大（例如，团问题和 SAT 问题），因此很难选择“类似”的目标。寻找合适的归约起始问题有时也非常困难，这只能通过经验来把握，一般的方法是尝试对几个问题归约，直至发现一个成功的例子为止。

这里要强调的是归约是从任意一个已知的 NP 完全问题实例到问题 Q 的，通常最容易犯的错误是反方向进行归约，记住正确归约顺序的方法是，归约可以确保 NP 完全问题可被问题 Q 的一个黑盒子算法所解决。这确实有点跟我们的直觉相悖，因为给定一个问题 Q 自然是要设法解决它，然而这里却试图证明使用 Q 的算法可以解决另一个问题（NP 完全问题），而并不尝试解决问题 Q 本身！

在归约过程中我们可以有一定的自由度，例如，如果 Q 包含了一个参数，则为了便于证明可以将它设置成任何固定的值（与之对照，归约到 Q 的问题的参数则不能被固定）。毕竟 Q 只是用来解决 NP 完全问题的工具，因此可以根据需要来使用它。除了固定参数外，Q 还可以采用其他方式加以限制，譬如对问题 Q 我们可以仅使用一类特定的输入（例如，正则图和双连通图）。另一个很重要的灵活性是只要归约可以在多项式时间完成，其效率并不重要。不仅可以忽略常数因子（例如，使问题规模增加一倍），还可以使问题规模扩大平方倍！可以引入多项式数目的新变量，可以用一个大的新图替换掉原图的一个顶点等。归约的目的并不在于要使之变成实际算法（至少要等到 P 被证明等于 NP 之后），因此没有必要考虑效率（在保持多项式函数界的前提下）。

在归约的构造过程中存在一些通用技术（Garey and Johnson[1979]提供了许多例子）。其中，证明一个 NP 完全问题是 Q 的特例是最简单的，由于解决了问题 Q 就等于解决了这个 NP 完全问题，即直接可得问题的证明。例如集合覆盖问题，问题的输入是集合 U 的子集 $S_1, S_2, \cdots, S_n$ 所组成的集合和整数 k，问题在于判定是否存在一个元素个数不超过 k 的子集 $W \subseteq U$，满足每个集合 S_i 中至少有一个元素在 W 中。我们发现顶点覆盖问题是集合覆盖问题的特例，此时 U 对应顶点集合，而每个 S_i 对应一条边，即包含了该边所相关的两个顶点。所以，若可以解决任意集合的覆盖问题，就能够解决顶点覆盖问题。

但在使用这个方法时必须非常仔细，一般说来，对一个问题增加更多的要求会使问题变得更加困难的说法是不正确的。例如，对于顶点覆盖问题，假定增加了顶点的覆盖集合不能包含两个相邻顶点这一额外约束，即要寻找一个顶点集合，它既是一个顶点覆盖同时又是一个独立集（独立集是指不包含相邻顶点的顶点集合）。由于必须考虑满足更多的要求，这个问题看似比顶点覆盖问题和独立集问题都难。但是，实际上这个问题是一个比较容易的问题，它可以在多项式时间内解决（见习题 7.115）。其实，额外要求限制了候选的集合，这样将使得最小的覆盖集合能被很容易地找到。

另一个相对容易的技术是局部归约，此时一个问题中的对象被映射到另一个问题中的对象，该映射通常都是局部意义下，即与其他对象保持独立。支配集问题的 NP 完全性证明就是按照这个模式的，我们将一个图中的每条边替换为另一个图中的一个三角形，这样的局部替换足以完成归约。该技术的难点在于如何以最佳方式定义对象。

3 着色问题的 NP 完全性证明了所使用的基本块构造是最复杂的技术。这些基本块通常彼此相互关联，独立设计每一个是不可能的，为了协调设计这些不同的基本块，我们必须考察问题中的所有既定目标。

11.4.7 更多的 NP 完全问题

以下列举一些 NP 完全问题，这些问题作为其他（例如习题中的）归约的基础是非常有用的，

在 Garey and Johnson [1979]中列举了更多这样的问题，有时为归约找到合适的问题往往是成功的一半。

哈密尔顿回路问题　图的哈密尔顿回路是指经过且仅经过图中所有顶点一次的简单回路。该问题就是对于一个给定的图，判定它是否包含哈密尔顿回路。无论对于有向图还是无向图，该问题都是 NP 完全的（可以从顶点覆盖问题归约到该问题）。

旅行商问题　设 $G=(V, E)$是一个加权完全图，旅行商的行程就是一个哈密尔顿回路。该问题就是给定图 G 和数值 W，判定是否存在旅行商的一个行程，满足其经过的边的总代价小于或等于 W（可以直接从哈密尔顿回路问题归约到它）。

哈密尔顿路径问题　图中的哈密尔顿路径是指经过所有顶点且仅经过一次的简单开路。该问题就是给定一个图，判定它是否包含哈密尔顿路径。无论对于有向图还是无向图，该问题都是 NP 完全的（可以从顶点覆盖问题归约到它。）

独立集问题　无向图 $G=(V, E)$中的独立集是指不包含相邻顶点的顶点集。该问题就是给定图 G 和整数 k 判定 G 中是否包含小于或等于 k 的独立集（可以直接从团问题归约到它。）

三维匹配问题　设 X、Y 和 Z 是大小为 k 的不相交集合，M 为满足 $x\in X, y\in Y, z\in Z$ 的三元组（x, y, z）所组成的集合。该问题就是判定是否存在 M 的一个子集满足其包含所有元素且仅包含一次，相应的二维匹配问题即一般偶图（regular bipartite）的匹配问题（可以从 3SAT 问题归约到它）。

划分问题　输入是集合 X，其中每个元素 $x\in X$ 均有对应的大小值 $s(x)$。该问题就是判定是否可以将集合 X 划分成元素大小总和恰好相等的两个子集（可以从三维匹配问题归约到它）。

注意当大小都是小的整数时，这个问题以及下一个问题都可以被背包算法（参见 5.10 节）高效解决。然而，由于输入的大小是指用来表示输入的位的数目，这种被称为伪多项式的算法其复杂度关于输入的大小是指数阶的。

背包问题　输入是集合 X，其中每个元素 $x\in X$ 均有对应的大小 $s(x)$和价值 $v(x)$。该问题就是判定是否存在一个子集 $B\subseteq X$, 其中元素大小总和小于或等于 s 并且元素的总价值大于或等于 v（可以从划分问题归约到它）。

箱柜包装问题　输入为数值序列 $a_1, a_2, \cdots, a_n$ 以及数值 b 和 k。该问题就是判定集合是否可以划分成 k 个子集且满足每个子集的数值总和小于或等于 b（可以从划分问题归约到它）。

11.5　处理 NP 完全问题的技术

NP 完全问题是一个优雅的理论，该理论让我们能识别出那些不太可能存在多项式时间算法的问题。但是证明给定问题是 NP 完全的并不能消除这个问题！我们仍然需要解决它。解决 NP 完全问题的技术与先前所看到的技术有时是不同的。我们（很有可能）不能用多项式时间精确而彻底地解决一个 NP 完全问题，因此必须采取折中的方式。通常折中的考虑因素有最优性、健壮性、高效性以及解的完备性等，每者都会有一些相应的牺牲，相同算法在不同场景下使用将会有不同的折中。

不一定能得到最优解（或者精确解）的算法被称为近似算法。我们特别感兴趣的是可以确保有精确度范围的近似算法，后面将看到这类算法的 3 个例子。

在 6.11 节中我们讨论了可能导致错误的概率算法，这类算法中最著名的是素数测试算法，素数测试问题既没有被认为是在 P 中，又没有被证明是 NP 完全的。（译者注：该问题已于 21 世纪初由 Agrawal, Kayal, and Saxena [2002]证明在 P 中。）由于需要数论的相关知识，我们将不描述素数测试算法。通常认为对所有输入都小概率出错的多项式时间概率算法不能用于解决 NP 完全问

题，因此这样的算法对于解决还未知是否在 P 中且不被认为是 NP 完全的问题可能更有效。这样的问题并不常见，但是概率算法可以作为其他策略方法（例如近似算法）的组成部分加以使用。

另一种折中是放弃在最坏情况下要求算法的运行时间是多项式的，尝试平均情况下能在多项式时间内解决 NP 完全问题。这种方式存在的问题是如何定义平均时间，例如在分析平均情况时，很难将特定问题的平凡输入（如只含孤立顶点的图）情况排除在外，这种平凡输入可以显著减少平均时间。如果实际输入的分布符合原先的假定，那么根据特定随机输入类型所设计的算法是有用的。然而找到合适的分布通常非常困难，设计平均情况下功效好的算法的主要难点是分析它们，这一步通常非常复杂。

最后，我们可以对算法的完备性加以折中，即可以允许算法仅对一些特定的输入是高效的。例如，对于偶图，顶点覆盖问题可以在多项式时间内解决（见习题 7.110）。因此，当从现实生活场景中抽象问题时，应确保在抽象定义中已包含所有与输入相关的额外条件。另一个例子是运行时间是指数的，但对于小的输入其效果较好的算法，因为解决问题这已经足够了。

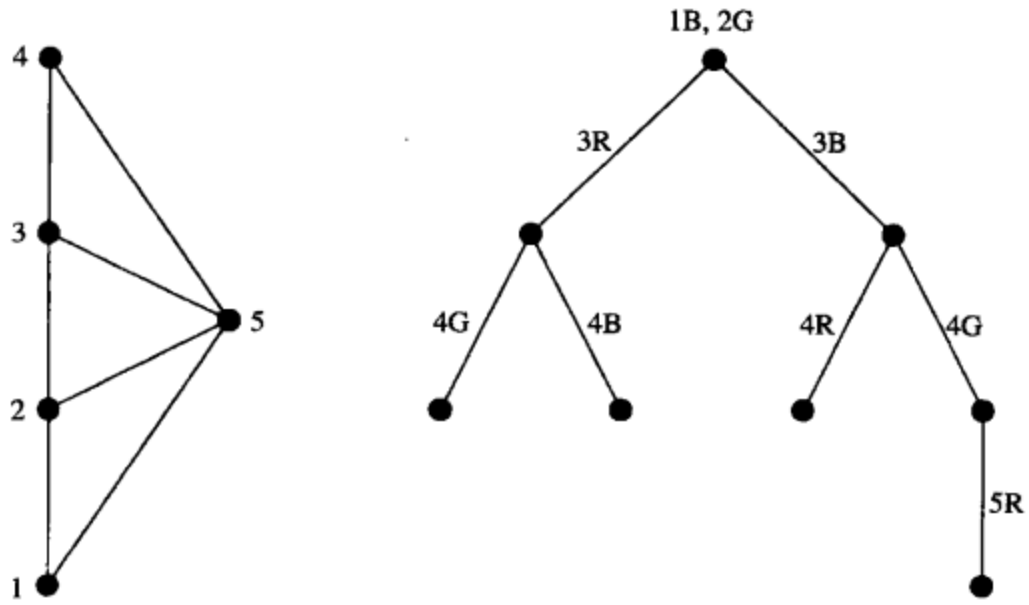
这一节将描述一些这样的技术并通过具体例子加以阐述。首先介绍的是两个一般但有用的技术，即回溯法和分枝限界法，这些技术很相似，它们可以作为近似算法或者对于小的输入是最优的算法的基础，接着将给出一些近似算法的例子。

11.5.1 回溯法和分枝限界法

下面通过一些例子来描述这些技术，考察 3 着色问题，它包含在一定约束下对图中的 n 个顶点进行颜色指派，这是对 n 个参数求最优赋值（在这里指颜色）问题的例子。在这个 3 着色例子中，每个参数对应 3 种颜色，可能有 3 个不同赋值，因此潜在的解的数目为 3^n，即对 n 个顶点进行 3 着色的各种可能。当然，除非图中没有边，否则有效解的数目将远小于 3^n，因为边对着色加以限制。为了探求所有顶点的着色方案，可以首先对其中一个顶点任意着色，并且在满足由边所带来的约束条件（即相邻顶点必须着色不同）的同时，继续给其他顶点着色。当对一个顶点进行着色时，可尝试各种与先前顶点的着色相一致的方案，该过程可以通过树遍历算法来完成，而这正是回溯法和分枝限界法的精髓所在。为了避免混淆图和树的顶点，将树的顶点称为节点。

树根对应问题的初始状态，而每个分枝对应于一个参数的决定过程，我们用 R(ed)、B(lue)和 G(reen)分别代表红、蓝和绿 3 种颜色。首先，选择任意两个相邻顶点 v 和 w 并将它们着色成 B 和 G，由于对任何一种正确的着色方案它们都不能有相同的颜色，而具体颜色的选取并不重要（总可对最终着色进行排列变化），这也是可以从两个顶点而非一个顶点开始着色的原因。这两个顶点的着色对应问题的初始状态，而后者则对应于树根。树在遍历过程中被逐渐构造出来，对于树的每个节点 t，我们选取图的下一个顶点 u 进行着色，可以根据 u 的可行着色数目让 t 增加一个、两个或者三个孩子节点。例如，若 v 和 w 着色后的首选顶点是 u，如果 u 与 w 相邻（w 已经被着色成 G），那么对于 u 有 B 和 R 两种着色方案，我们将对树根添加两个相应的孩子节点并选取这两个孩子节点中的一个继续该过程。在一个顶点被着色后，对剩余顶点着色的限制将增大；因此随着树的深度增加，孩子节点的数目将趋于减少。

如果对图中所有顶点都能够成功着色，那我们就完成了 3 着色任务。但是在更多情况下，我们将遇到不能被着色的顶点（即它已经和 3 种不同颜色顶点相邻了），此时需要回溯——即沿着树往上并探求其他的孩子节点。图 11.7 给出了一个图及其相应 3 着色回溯树的例子，注意在这个例子中一旦顶点 1 和顶点 2 的着色固定下来，图的其余部分只存在一种着色方案（可通过树的最右路径发现）。



（a）图　　（b）回溯树

图 11.7　回溯法解 3 着色问题的例子

可以将树的遍历算法视为基于归纳的算法，这需要增强假设以包括部分顶点已经着色的图。换句话说，归纳假设不仅要处理完全没有着色的图，还要完成已经部分被 3 着色的图。

归纳假设：对于小于 k 个顶点的图，我们知道怎样完成 3 着色或者判定 3 着色无法实现。

给定一个有 k 个顶点未着色的图，选取其中一个顶点并为其寻找各种可能的着色方案。如果所有颜色都已被其相邻顶点所使用，那么 3 着色将无法完成。否则，我们给这个顶点指派上可行的颜色（每次一种）并通过归纳法解决剩余问题（此时有 k–1 个未着色顶点）。该算法如图 11.8 所示。

算法 *3-coloring (G*, var *U)*
输入： $G=(V, E)$（一个无向图）和 U（已被着色的节点的集合）
{U 初始为空}
输出： 使用 3 种颜色对 G 的每个节点进行着色的结果

```
begin
    if U = V then print "coloring is completed"; halt
    else
        pick a vertex v not in U ;
        for C := 1 to 3 do
            if no neighbor of v is colored with color C then
                add v to U with color C ;
                3-coloring(G, U)
end
```

图 11.8　3 着色算法

不难给出一个图及其 3 着色问题的遍历顺序，这样所得到的结果是有指数多个节点的树，见习题 11.34，这在回溯算法中是非常普遍的。我们希望能通过一个“好”的遍历顺序尽可能早地发现解，但目前为止所描述的算法并没有特别说明怎样挑选下一个顶点。由于任何顶点都可作为后继顶点的候选，所以有一定的自由度进而可以利用它来设计启发式策略，稍后将重新回到这个主题。

分枝限界法是回溯法求解涉及寻找某个目标函数最小（或最大）值问题的一种变形方法，考察一般的着色问题，现在感兴趣的是寻找对该图着色所需最少颜色数目而非对 3 着色问题得到肯

定或者否定的答案。我们可以建立一个类似 3 着色问题的回溯树，但是分枝的数目可以很多。每一个新顶点可以用已使用过的颜色（除非它的一个相邻顶点已经使用过那个新颜色）着色或者使用新颜色着色。这样，3 着色算法以两种方式被修改：（1）常数 3 由目前为止所使用过的最多的颜色数目所替代；（2）由于对该图可能存在更好的着色方法，故算法并不在 $V=U$ 时终止。

由于一个新颜色总可以被指派到顶点，该算法的问题在于只有遇到叶子节点（即 $V=U$）时才进行回溯，该算法的性能几乎总是很差的（除非该图非常稠密）。我们可以对这个算法的性能加以改进，主要是基于以下观察，同时这也是分枝限界法的基础。假定一路遍历树到某个叶子节点并且找到一个有效的 k 着色，进一步假设在沿着树向上回溯几步之后，我们遍历另一条路径并到达一个需要第$(k+1)$种颜色的节点，由于已经知道存在一个更好的解，此时可以直接回溯。这样 k 就充当了回溯法的一个界，在每个节点上计算沿着树向下可能找到最优解的下界，如果这个下界超过一个已知解，则进行回溯。使分枝限界法高效的关键在于计算好的下界（或者上界，如果我们的目的在于使目标函数最大化）。另一个关键是寻找好的遍历顺序以便更快发现好的解，这样就可以更早地进行回溯。

下面通过整数线性规划（在 10.3 节中提及）来阐述这个思想，这个问题与线性规划类似，但是包括了变量的值为整数这一额外限制。设 $\bar{x}=(x_1, x_2, \cdots, x_n)$ 是由变量组成的向量；$\bar{a}_1, \bar{a}_2, \cdots, \bar{a}_k$ 是大小为 n 的向量；$b_1, b_2, \cdots, b_k$ 和 $c_1, c_2, \cdots, c_k$ 是实数向量。问题是以下线性目标函数取值的最大化：

$$z=c_1x_1+c_2x_2+\cdots+c_nx_n \tag{11.1}$$

限制条件为 $\bar{x}$ 是整数并且满足以下约束：

$$\begin{aligned}
&\bar{a}_1\cdot\bar{x}\leqslant b_1\\
&\bar{a}_2\cdot\bar{x}\leqslant b_2\\
&\cdots\\
&\bar{a}_k\cdot\bar{x}\leqslant b_k。
\end{aligned}\tag{11.2}$$

（其中 $\bar{a}_i s$ 和 $b_i s$ 都是常数。）许多 NP 完全问题都可很容易使用整数规划问题（后面将给出一个例子）加以表示，故整数规划是 NP 难的。事实上它是 NP 完全的，但证明它属于 NP 非常复杂。

以下是团问题的整数线性规划表示（此时的问题是找到一个最大团而非判定是否存在某一特定大小的团）。存在与顶点对应的 n 个变量 $x_1, x_2, \cdots, x_n$，使得当 v_i 属于这个最大团时 $x_i=1$，否则 $x_i=0$。目标函数是

$$z=x_1+x_2+\cdots+x_n,$$

其含义是希望选取尽可能多的顶点，对每个顶点有以下约束：

$$0\leqslant x_i\leqslant 1，\text{对所有 } 1\leqslant i\leqslant n,$$

并且对于每对不相邻的顶点有以下约束：

对每一对满足 $(v_i, v_j)\notin E$ 的顶点 v_i 和 v_j，有 $x_i+x_j\leqslant 1$。

第一个约束集将变量限制为 0 或 1，第二个约束集则确保两个不相邻顶点不能同时被选择，因此所有被选上的顶点组成了一个团。

整数线性规划可通过分枝限界法来解决，后者利用对应的线性规划（就是没有整数条件约束的相同问题）来计算上、下界。线性规划的解若只包含整数，这样任务就完成了，但是更多情况下解将包含一些非整数值。例如，假定团问题对应的线性规划解为 0.1, 1, …, 0.5 及 $z=7.8$，由于相对整数线性规划，线性规划在更少限制情况下达到目标函数的最大化，它所发现的是整数线性规划最大可能值的上界，因此我们不能期望找到大小超过 7 的团。这类信息可以帮助我们进一步沿着树向下搜索，在一般回溯过程中，当沿着树向下搜索时将进行一些选择，一个更低的节点将对应于原始问题的子问题。例如，子问题可能对应于选取 v 和 w 而将 u 和 x 排除出团，在这种情况下我们将尝试发现包含 v 和 w 但不包含 u 和 x 的最大团，如果在这个时候线性规划的解小于目前已知团的大小，则可以直接回溯。这是分枝限界法的核心思想所在，所找到的上界（或者下界，如果目标函数要求最小化）将使我们尽早进行回溯。

也可以利用线性规划的结果来帮助分枝的选择。例如，根据非整数线性规划的解 $x_2 = 1$，可以猜测在整数线性规划中 $x_2 = 1$ 仍然是解。这可能并不是一个好的猜测，但它的确是所要寻找的启发式策略的一个例子。我们试图增加快速发现最优解的概率（由于该问题是 NP 完全的，一直保持猜测“正确”几乎不可能）。置 x_2 为 1 并更新约束条件（例如，将所有不与 v_2 相邻的顶点置为 0），然后解更新后的线性规划。如果在某一时刻修改后的线性规划有了最大值 $z=a$，而 a 比目前为止所知的最大团要小，则可以进行回溯。

这样线性规划就有两个目的：它给出了上界以便进行回溯，同时也对后继选择给出了提示。我们希望当完成“最有可能成功”的子问题时，能够尽可能多地修剪掉（prune）其他子问题，修剪的数目（以及整个算法的效率）依赖于划分问题并选择探索后继子问题的启发式策略。而这样的启发式策略与具体特定问题相关，这个领域已经存在着大量的研究工作。

当所有子问题都探索了或者已被修剪掉时，分枝限界算法将找到最优解，如果这个过程花费太长的时间，则可以终止这个算法，并且将目前为止所得到的最好结果作为最优解的近似。树的遍历可以采用广度优先搜索、深度优先搜索或者二者的组合。一个很早终止的极端例子是沿着第一条路径（通过某种启发式策略）并找到一个可行解（通常在叶子节点处）作为算法的结果。例如，在着色算法中我们可以根据顶点度的逆序（其思想在于先固定度数较小顶点的颜色会使我们相对不自由）对顶点进行着色，这是一个简单的贪婪算法。

11.5.2　确保性能的近似算法

这一节将讨论 3 个 NP 完全问题的近似算法，它们是顶点覆盖问题、箱柜包装问题以及欧几里得旅行商问题。所有这些近似算法都有确保的性能，即可以证明它们所得的解和最优解的差距并不大。

顶点覆盖问题

首先介绍寻找给定图最小顶点覆盖的一个简单近似算法，该算法可以确保找到覆盖，其包含的顶点个数不超过最小覆盖所包含顶点个数的 2 倍。假设 $G=(V, E)$是一个图，M 是 G 的一个极大匹配，由于 M 是一个匹配，其所包含的边没有共同顶点，又因 M 是极大的，故所有其他边至少和 M 中的一条边有相同顶点。

□ **定理 11.9**

与极大匹配 M 中的边相关联的所有顶点构成一个顶点覆盖，并且其大小不超过最小顶点覆盖大小的 2 倍。

证明：由于 M 是极大匹配，所以属于 M 的顶点集形成一个顶点覆盖。任何一个顶点覆盖必须覆盖所有的边，这当然也包括 M 中的边，但由于 M 是一个匹配，M 中的顶点覆盖 M 中的边的数目不能超过 1 条，因此至少 M 中一半的顶点必须属于每一个顶点覆盖。 □

可以通过以下方法来寻找极大匹配，即简单收集边直至所有边都被覆盖到。由于顶点覆盖包括匹配中所有顶点，所以我们想要寻找到一个较小的极大覆盖，遗憾的是，寻找最小极大匹配（即有最少边个数的极大匹配）问题也是 NP 完全的（Garey and Johnson [1979]的问题[GT10]）。习题 11.35 将讨论另一个确保性能的顶点覆盖问题的近似算法。

一维箱柜包装问题

一维箱柜包装问题是指将不同大小的物体打包到规定大小的箱柜中，并且箱柜使用的数目要尽可能少。例如，在搬家时希望动用的汽车数尽可能少（或尽可能少地让同一汽车往返两地），这可以通过尽可能将汽车塞满的方式实现。搬运是一个三维问题，但这里将集中讨论一维的版本，为简单起见，同时假设所有箱柜的尺寸都是 1。

> **问题** 设 $x_1, x_2, \cdots, x_n$ 构成大小介于 0 和 1 之间的实数集合；将这些数划分成尽可能少的子集合使得每个子集中的数的总和不超过 1。

一维箱柜包装问题是在现实应用中提出的，例如在内存管理问题中会有许多不同大小内存块的需求，这些块将从几个大的可用内存块中分配。箱柜包装问题是一个 NP 完全问题（见习题 11.8）。

该问题的一个启发式策略是将 x_1 放入第一个箱柜，接着对于每个 i，将 x_i 放入能容纳它的第一个箱柜，或者当使用过的箱柜都不能容纳它时使用一个新的箱柜，这个算法被称为首次适应（first fit）算法。正如下一个定理所述，首次适应算法在最坏情况下并不表现得“太差”。

□ **定理 11.10**

首次适应算法至多需要 2OPT 个箱柜，这里 OPT 是最少需要的箱柜数。

证明：首次适应不可能留下两个没有装到半满的箱柜，否则第二个箱柜中的物品可以放入第一个箱柜，因此所使用的箱柜数不可能超过所有物品大小总和（只入不舍）的 2 倍。又因为最优解中的箱柜数目不可能比所有物品总和（在这种情况下，所有的物品被恰好包装）还少，由此可推得这个定理。 □

事实上，定理 11.10 所给出的界非常保守，通过更复杂的分析，定理中的常数 2 可以减为 1.7。对于首次适应算法由于存在解是最优解 1.7 倍的例子，因此常数 1.7 是紧密的（tight）。

用以下简单修改可以改进首次适应，当许多小数目的物品出现在开头将导致最坏情况。因而我们可以不按物品的出现顺序，而是先将它们逆序排序，然后再使用首次适应。这样的修改算法又称为递减首次适应（decreasing first fit），在最坏情况下其解在最优解大约 1.22 的常数因子范围内（此处略去证明）。

□ **定理 11.11**

首次适应算法至多需要 11/9 OPT + 4 个箱柜，这里 OPT 是最少需要的箱柜数。 □

这个常数同样也是紧的。首次适应和递减首次适应都是简单的启发式策略，其他还有一些更好常数因子的方法，但在大多数场合下，分析比较复杂。

我们所描述的策略是启发式算法中的一些典型，它们给出了人们手工处理这些问题的自然方

法。很多场合下，用于处理大输入的直接方法的性能很差，因此分析这些算法的性能是非常重要的。

欧几里得旅行商问题

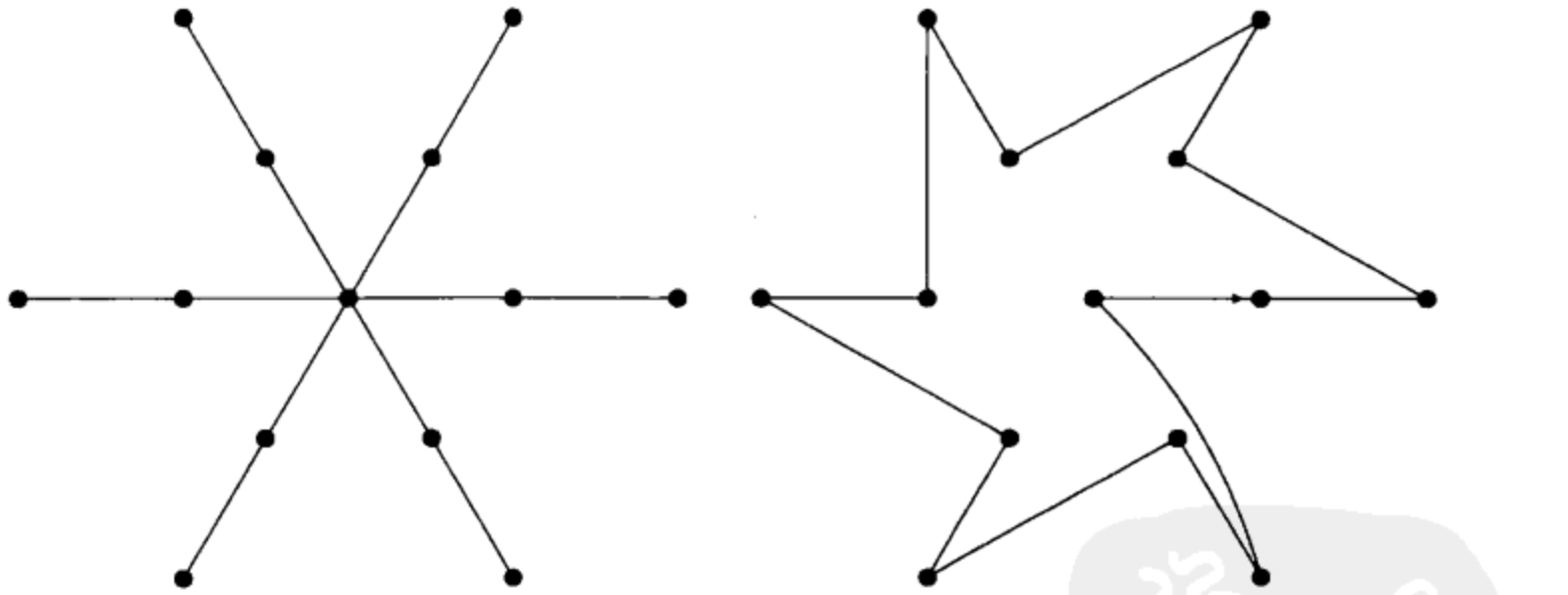
旅行商问题（TSP）是一个有着多种应用背景的重要问题，我们在此讨论的是 TSP 的一个变形，它有一个附加约束，即权值对应欧几里得距离。

> **问题**　$C_1, C_2, \cdots, C_n$ 是平面中点的集合，它对应于 n 个城市的位置；试找到一条长度最短的哈密尔顿回路（旅行商的路线）。

这个问题仍然是 NP 难的，但是会看到欧几里得假设将帮助我们为该问题设计近似算法（在某种意义下，可以放松这个假设，只要假定距离满足三角形不等式，即任意两点间的直线距离比任何通过其他点的路线要短）。

算法首先计算最小代价生成树（这里代价 = 距离），这是一个容易得多的问题（参见 7.6 节）。我们说明树的代价不会超过最佳 TSP 行程，因为 TSP 是包含所有顶点的回路；因此，除去 TSP 行程中的任何一条边将使它成为一棵生成树，故 TSP 行程的代价至少是最小生成树的代价。

但是生成树并非直接对应一个 TSP 行程，因此需要修改。首先考察由深度优先搜索遍历树（从任意顶点出发）所形成的环路，当搜索回溯时其包括了一条反向边（比如说，要遍历每个大厅两边都有展览室的树形画廊，按照每次选择右边的方式进行遍历将形成这样的环路）。每条边恰好被遍历两次，所以这个环路的代价是这棵最小生成树代价的两倍，因而也不会超过最短 TSP 行程的两倍。现在可以通过选取直接路线而非回溯的方法将这个环路转变成一个 TSP 行程（如图 11.9 所示），也就是说我们直接到第一个新顶点而不是通过相同的边进行回溯。关于距离是欧几里得距离的假设是重要的，这是因为它确保了任意两个城市间的直接路线总不比非直接路线差。所以，这样形成的 TSP 行程仍然不会超过最短 TSP 行程长度的两倍，虽然通常情况下其要比后者短。



（a）一棵生成树　　（b）从生成树的中间节点出发首先选择右边的路径所获得的 TSP 行程

图 11.9　将环路转变成 TSP 行程

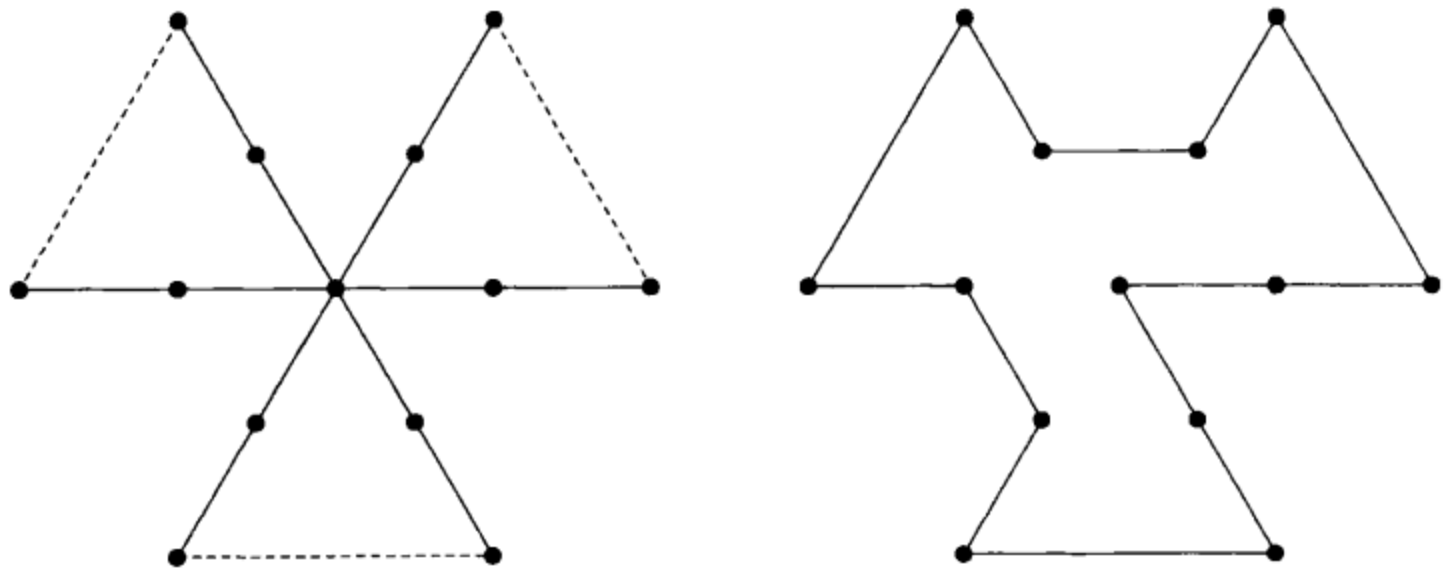
复杂度　该算法的运行时间主要由最小代价生成树算法的运行时间所决定的，对于欧几里得图，后者为 $O(n\log n)$（可参见 Preparata and Shamos [1985]）。

改进

我们刚刚描述的算法可以在以下方面进行改进。该算法中最“马虎”的部分是从树的遍历转变成 TSP 行程，可以从另一个视角看这个转变，即它通过重复每条边两次的方法从树顶建立起一

条欧拉环路，接着通过走捷径从这个欧拉环路中获得 TSP 行程，我们可以更加高效地将这棵树转变成欧拉图。欧拉图中所有节点的度均为偶数，考虑树中的所有度为奇数的节点，它们的个数一定是偶数（否则所有节点度数的总和将会是奇数，由于节点度数总和恰好是边数目的 2 倍，因而这是不可能的）。如果我们对这棵树添加足够多的边，使得所有节点的度均为偶数，这样就得到了一个欧拉图。由于 TSP 行程将由这个欧拉环路（会走一些捷径）构成，我们想要将额外添加的边的长度最小化，所以把这个问题抽象出来。

给定平面中的一棵树，希望添加一些边使之成为一个欧拉图，并且要使添加边的总长最小。对于每个度为奇数的顶点至少要添加一条边，尝试就添加一条。假定有 $2k$ 个度为奇数的顶点，如果添加 k 条边，每条边连接两个度为奇数的顶点，那么所有边的度都为偶数。这样该问题就变成匹配问题，我们想要找到一个最短长度匹配，使其覆盖所有度为奇数的顶点。对于一般图可以在 $O(n^3)$时间内找到权值最小的完美匹配（参见 Gabow [1976]或 Lawler [1976]），最近 Vaidya [1988] 提出了一个针对欧几里得距离特例的算法，它的运行时间为 $O\ (n^{2.5}\ (\log n)^4)$。（在实际中它是否是一个更好的算法还不清楚。）最终的 TSP 行程可以通过走捷径的方法从欧拉图获得，图 11.10 给出了图 11.9 中的树应用该算法所获得的 TSP 行程。



（a）生成树以及匹配　　（b）从欧拉环路中获得的 TSP 行程

图 11.10　最短欧拉环路及其对应的 TSP 行程

□ **定理 11.12**

改进后算法所生成的 TSP 行程，其长度至多为最短 TSP 行程长度的 1.5 倍。

证明：将忽略掉捷径（由于在最坏情况下可能没有任何捷径可走），而着重讨论欧拉环路的长度，环路由树和匹配构成。我们已经证明树的长度至多为最短 TSP 行程长度，因而只需要证明匹配的长度至多为最短 TSP 行程的一半。设 Q 为一个最短 TSP 行程，Q 是一个包含所有顶点的回路，并设 D 是 T 中度为奇数的顶点所组成的集合。可以用下列方式得到 D 的两个不相交的匹配，使得它们的总和不超过 Q 的长度（见图 11.11）。首先从 D 的任意顶点 v 出发，将它与 D 中关于 Q 顺时针方向与之最近的顶点相匹配，然后继续沿着顺时针方向匹配。如果匹配的顶点在 Q 中不相邻，那么它们之间的距离不会超过 Q 中连接它们的路径的长度（根据三角形性质）。该过程给出了一种匹配，第二种匹配可以通过重复相同过程以逆时针的方式进行而获得。如图 11.11 所示，两种匹配长度总和至多是 Q 的长度，但由于 M 是 D 权值最小的匹配，所以它的长度至多是 Q 长度的一半。□

寻找权值最小的完美匹配所花的时间远远大于寻找最小代价生成树所花的时间，但是前者可以得到一个更好的上界，是否能够改进常数 1.5 或者找到一个更快的算法同样达到这个常数仍然是一个未解问题。[译者注：Arora [1998]和 Mitchell [1999]证明了如果这些点在欧几里得平面中，那么存在多项式时间近似方案（polynomial-time approximation schema），即对任意固定的$\varepsilon>0$，可以在多项式时间内（这里强调要把ε视为固定的常数）找到一个 TSP 行程，其长度至多为最短 TSP 行程的$(1+\varepsilon)$倍。]该算法阐明了这类算法一个主要特点：先将一个更容易的问题抽象出来，或者放松原始问题的某些部分，然后设计相应的启发式策略。

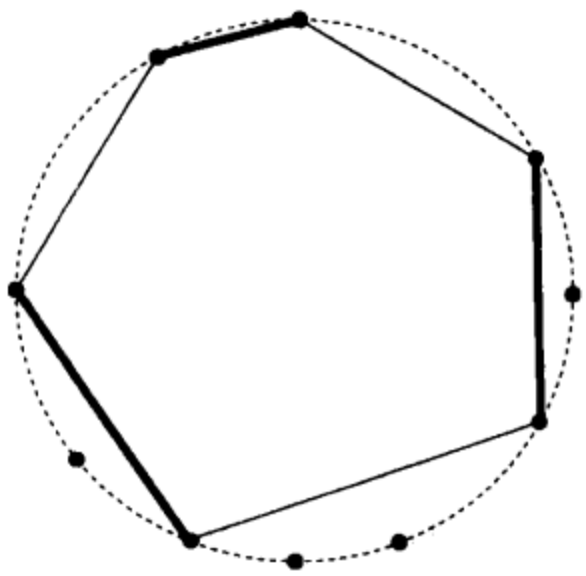


图 11.11　总长至多是 TSP 行程长度的两个匹配

11.6　小结

前面的章节让我们对能够设计出好的算法充满信心，然而本章使我们对现实有了进一步的了解。但很遗憾，存在许多重要问题，它们不能通过优雅、高效的算法解决，我们必须能够识别这些问题并为它们找出次优解。当遇到问题时，有两种攻克它的路线方针，我们可以尝试使用前面章节所介绍的技术来解决它，或者使用本章所介绍的技术来证明这个问题是 NP 完全的。为了在选择正确步骤之前避免走许多弯路，我们需要对问题的困难程度有某种直觉。

参考文献和补充读物

NP 完全性的概念最早由 Cook [1971]的开创性文献引入（前苏联的 Levin [1973]独自发现了类似的结果），Karp[1972]提出了 24 个重要的 NP 完全问题（它们中的一些已经包括在 11.4 节中）。Cook 和 Karp 所获得的图灵奖部分要归功于这项工作，他们的图灵奖报告分别刊登在 Cook[1983]和 Karp [1986]上。人们还研究了其他一些未被认为在 P 中的著名问题类，co-NP 就是其中的一类，它包括了所有 NP 中问题的补问题。例如，判定一个布尔表达式是否恒假就是 co-NP 中的问题。co-NP 和 NP 不同的原因在于，语言被非确定算法所接受的定义是不对称的，即对于接受和拒绝一个输入有着不同的要求。另一个重要的问题类是 PSPACE，它包括所有可以使用多项式空间解决的问题。结果表明，算法限定在多项式空间内，非确定性并不能增加其能力。换句话说，任何可在多项式空间内由非确定算法解决的问题同样可以在多项式空间内由确定性算法解决[Savitch 1970]。推广的 HEX 博弈问题是一个多项式空间完全问题的例子，除了归约可以使用多项式空间外，这里完全性的定义类似于 NP 类的定义方法。介于 P 和 PSPACE（参见 Even and Tarjan[1976]）之间存在一系列类的分层，称之为多项式时间分层（polynomial-time hierarchy），值得注意的是目

前为止并不能证明上述任何一类与 P 不等！我们尚不知道是否有问题在 PSPACE 中却不属于 P，但是确实存在需要指数时间和空间的问题（Meyer and Stockmeyer [1972]），甚至还有需要更多时间和空间的问题（Fischer and Rabin [1974]）。

还有一些问题既没有被认为在 P 中又未被证明是 NP 完全的，其中最著名的是图同构问题和素数测试问题（译者注：如前所述，素数测试问题已于 2002 年被证明在 P 中）。图同构在许多特殊场合可以在多项式时间内解决（参见 Luks [1982]），但是在一般情况下仍然是一个未解问题。Rabin [1976]以及 Solovay and Strassen [1977]为素数测试提供了 Monte Carlo 概率算法。（这些算法以极小的出错概率判定一个数为素数，而当它们判定一个数不是素数时不会出错。）Goldwasser and Killian [1986]提出了测试素数的 Las Vegas 概率算法，它不会出错，并且其期望运行时间是多项式的。另一个看似简单但至今未解的问题是偶环（even-cycle）问题，即给定图判定其是否包含一个偶数长度的（简单）回路（参见 Klee, Ladner, and Manber [1984]）。许多关于 NP 完全性及其相关主题的信息可以在 Garey and Johnson [1979]以及 1981 年起在杂志“Journal of Algorithms”的 NP 完全性专栏中找到。一个很自然的问题被提了出来，即是否所有在 NP 中的问题或者是 NP 完全的或者在 P 中，这个问题由 Ladner [1975]部分地解决了，证明了除非 P = NP，否则在这两个类之间有无穷多个其他问题类。

平均在多项式时间内解决哈密尔顿回路问题的算法在 Angluin and Valiant [1979]中描述，而 Purdom and Brown [1985b]中描述了解决可满足性问题的平均多项式时间算法。解决加权顶点覆盖问题的近似算法在 Bar-Yehuda and Even [1981]中给出，该算法所找到的顶点覆盖其权值至多为最小覆盖的 2 倍，Gusfield and Pitt [1986]给出了该算法的一个更加直观的解释。用 k 种（k 值固定）颜色给图着色的启发式策略由 Brélaz[1979]和 Turner [1988]给出，这些启发式策略被证明可以成功运用在“几乎所有的”图上（精确定义可以参考 Turner [1988]）。事实上，Wilf [1984]证明了图的 k 着色（k 值固定）问题的简单回溯树的平均大小是一个与 n 无关的常数。（对于 $k = 3$，其值小于 200；对于 $k = 5$，其值为 100 万。）但有可能这些算法的良好性能更多是因为平均的定义而非算法本身的能力（可参见 Franco [1986]），图着色问题最著名的且确保上界的近似算法由 Wigderson [1983]给出。回溯技巧在 Golomb and Baumert [1965]中描述（同样可参见 Bitner and Reingold [1975]以及 Horowitz and Sahni [1978]），Knuth [1975]描述了估计回溯算法运行时间的一种技巧，启发式策略的一般讨论由 Pearl [1984]给出。

11.5.2 节中所给出的首次适应和递减首次适应的性能上界由 Johnson et al. [1974]证明，对于与箱柜包装问题的一个最优解非常接近的启发式策略，参见 Karmarkar and Karp [1982]。

根据针对旅行商问题所提出的解决方案来看，它很可能是研究得最多的 NP 完全问题。对一般问题的近似算法由 Lin and Kernighan [1973]给出，对于欧几里得问题 Christofides [1976]所提出的算法达到了上界 1.5，而这是目前为止已知对于多项式算法的最好上界（译者注：如前所述，20 世纪 90 年代后期对它给出了多项式时间近似方案）。Lawler, Lenstra, Rinnooy Kan, and Shmoys [1985]一书中包含 12 篇覆盖该问题大多方面的文章，其中包括启发式策略及其分析、分枝限界算法、一些特例以及相关应用等。

操练型习题

利用本章讨论过的或者本章其他习题中的 NP 完全问题来完成习题。

11.1 根据 11.2 节所给出的定义，证明所有 P 中的问题都是多项式时间相互等价的。

11.2 证明非确定算法识别语言的定义并不允许一个算法识别两种不同的语言。

11.3 考虑以下算法，其判定图中是否有一个大小为 k 的团。首先，生成所有恰好包含 k 个顶点的顶点子集，一共有 $O(n^k)$个这样的子集。接着，检查由这些子集所导出的子图，判断其中是否有完全图。为什么这不是团问题的多项式时间算法，从而可以推出 P = NP 呢?

11.4 对于以下表达式，写出从 SAT 到 3SAT 归约过程中（在 11.4.3 节中给出）所获得的 3SAT 表达式

$$(x+y+\overline{z}+w+u+\overline{v})\cdot(\overline{x}+\overline{y}+z+\overline{w}+u+v)\cdot(x+\overline{y}+\overline{z}+w+u+\overline{v})\cdot(x+\overline{y})。$$

11.5 对于以下表达式，画出从 SAT 到团问题归约过程中（在 11.4.4 节中给出）所获得的图

$$(x+\overline{y}+z)\cdot(\overline{x}+y+\overline{z})\cdot(\overline{x}+y+z)\cdot(x+\overline{y}+\overline{z})。$$

11.6 对于以下表达式，画出从 3SAT 到 3 着色问题归约过程中（在 11.4.5 节中给出）所获得的图

$$(x+\overline{y}+z)\cdot(\overline{x}+y+\overline{z})\cdot(\overline{x}+y+z)。$$

11.7 证明背包问题是 NP 完全的。

11.8 证明箱柜包装问题是 NP 完全的。

11.9 将 3 着色问题以整数线性规划的形式提出来。

创造型习题

利用本章讨论过的或者本章其他习题中的 NP 完全问题来完成习题。

11.10 证明以下问题是 NP 完全的：给定无向图 $G = (V, E)$和整数 k，判定 G 是否包含生成树 T 且满足 T 中所有顶点的度均小于或等于 k。

11.11 证明即使图中所有顶点的度均被限定为偶数，其顶点覆盖问题仍然是 NP 完全的。

11.12 再次考虑第 5 章所讨论的寻找大导出子图问题，假定不是要求每个导出子图顶点的度大于或等于 d，而是要求度小于或等于 d。这里以判定问题的形式将其表示，给定无向图 $G = (V, E)$和两个整数参数 d 和 k，判定 G 是否包含一个至少有 k 个顶点的导出子图 H，满足 H 中所有顶点的度均小于或等于 d。证明这个问题是 NP 完全的。

11.13 证明以下问题是 NP 完全的：给定无向连通图 $G = (V, E)$和整数 k，判定 G 是否同时包含一个大小为 k 的团和一个大小为 k 的独立集。

11.14 证明以下问题是 NP 完全的：给定无向图 $G = (V, E)$和整数 k，判定 G 是否包含一个由 k 个顶点组成的子集，并且后者的导出子图是无环图。

11.15 设 E 是一个 CNF 形式的布尔表达式，满足每个变量 x 恰好以 x 和 $\overline{x}$ 的形式各出现一次。找到一个多项式时间的算法判定这样的表达式是否可满足，或者证明该问题是 NP 完全的。

11.16 证明名为 1-in-3SAT 的 3SAT 问题变形是 NP 完全的。其输入与 3SAT 问题相同，而问题是判定是否存在可满足的指派，使得每个子句的 3 个变量中恰好有一个为真。

11.17 证明 2-in-4SAT 问题是 NP 完全的：输入是每个子句中恰好有 4 个变量的 CNF 布尔表达式，问题是判定是否存在可满足的指派，使得每个子句的 4 个变量中恰好有两个为真。（提示：可以利用习题 11.16 的结果。）

*11.18 输入是一个 CNF 形式的布尔表达式，问题是判定是否存在可满足的指派，使得每个子句包含奇数个值为 1 的变量。例如，如果输入是 3SAT 的输入，则寻找一个指派，使得每个子句中有 1 个或 3 个变量值为 1。（该问题的另一个视角是，每个子句中变量间的或操作被异或操作替代。）这可能看似习题 11.16 和习题 11.17 中问题的另一种变形，但事实上这个问题可

以在多项式时间内解决。请为它找到一个多项式时间的算法。

11.19 输入是无向正则图（即所有顶点的度均相同的图），证明对于正则图，团问题仍然是 NP 完全的。

11.20 3 元集覆盖（X3C, exact cover by 3-sets）问题可陈述如下：输入是含有 $3n$ 个元素的集合 S 以及由 S 的子集 $S_1, S_2, \cdots, S_k$ 所组成的集合，它们中每个都恰好含有 3 个元素，问题是判定是否存在由 $S_{i_1}, S_{i_2}, \cdots, S_{i_n}$ 所组成的子集合，使得 S 的每一个元素恰好包含在某一子集 S_{i_j} 中。证明 X3C 是 NP 完全的。

11.21 证明以下问题是 NP 完全的：给定包含 $3n$ 个顶点的无向图 $G=(V, E)$，判定 G 的顶点是否可以被划分成 n 组，使得每组包含两两相连的 3 个元素。换句话说，问题是判定该图是否可以被划分成 n 个三角形。

11.22 设 $G=(V, U, E)$是一个偶图，其中 V 和 U 是顶点的集合，而 E 是由连接它们的边所组成的集合，V 对应于机器集合而 U 对应于零件集合。如果机器 v_i 可以加工零件 u_j，那么 v_i 将与 u_j 相连。假设房间最多可容纳 K 台机器但不限零件数目（为简单起见，假定所有机器大小均相同），并且假定有足够多的可用房间，但是希望零件从一个房间到另一个房间的搬动尽可能少。每条边(v_i, u_j)的代价为 $c(v_i, u_j)$，即当零件 u_j 和机器 v_i 不在同一房间时，将 u_j 搬运到 v_i 处所需付出的代价，我们定义将机器和零件分配进房间所需花费的代价是连接不同房间零件和机器边的代价总和。证明以下问题是 NP 完全的：给定图 G 以及参数 K 和 C，判定是否可能以代价小于或等于 C 的方式将机器和零件分配进房间。（换句话说，顶点集合应当被划分成若干子集，每个子集最多只能含有 K 个 V 中的顶点，并且满足连接两个不同子集顶点的边所对应的代价总和不超过 C。）

11.23 设 S 是一个集合，而 $C=\{C_1, C_2, \cdots, C_k\}$是 S 的子集所组成的集合，每个这样的子集都含有 4 个元素。如果 S 的两个子集包含公共元素，那么称它们是相连的。如果对于所有 $1 \leqslant i \leqslant k-1$ 均有 C_i 和 C_{i+1} 相连且 C_1 和 C_k 相连，则称集合 C 是一个回路。如果子集 $C' \subseteq C$ 不包含回路，则称 C' 是无环的。证明以下问题是 NP 完全的：给定集合 C，判定其是否包含一个无环子集 C'，满足：（1）任何 C' 中的两个子集最多含有一个公共元素，（2）S 的每一个元素都至少包含在 C' 的一个子集中。

11.24 假设无向图的哈密尔顿路径问题是 NP 完全的，证明无向图的哈密尔顿回路问题也是 NP 完全的。（11.4 节定义了这两个问题。）

11.25 输入是无向图 $G=(V, E)$和 G 中两个不同顶点 v 和 w。证明除非 P = NP，否则不存在判定 G 是否包含起讫于 v 和 w 的哈密尔顿路径的多项式时间算法。

11.26 再次考察判定图 $G=(V, E)$是否包含起讫于顶点 v 和 w 的哈密尔顿路径的问题（参见习题 11.25），设 G'是通过增加两个新顶点 $\overline{v}$ 和 $\overline{w}$ 以及两条新边 $(v, \overline{v})$ 和 $(w, \overline{w})$ 的图。如果 G' 中含有哈密尔顿路径，则它的起讫顶点一定是 $\overline{v}$ 和 $\overline{w}$，因此这样的路径对应于 G 中起讫于 v 和 w 的哈密尔顿路径。试问由刚才的证明可以得出什么结论？

11.27 证明图的 k 着色问题是 NP 完全的。该问题是给定无向图 $G=(V, E)$和整数 k，判定 G 是否能至多用 k 种颜色着色。

11.28 证明如果存在多项式时间的近似算法能够对任何图着色，并且所使用的颜色数目少于 4/3 倍的最少着色所需颜色数目，那么 P = NP。

11.29 证明后向边集合问题是 NP 完全的：给定有向图 $G=(V, E)$和整数参数 k，判定 G 是否包含至多有 k 条边的集合 F，使得任何 G 中的有向回路都至少含 F 中的一条边。

11.30 设 Q 是某个关于无向图的 NP 完全问题，假定你发现了一个多项式时间算法可以解决有特

别限定的图（例如，平面图、包含完美匹配的图、欧拉图）的 Q 问题。这个算法是否可以推得当限定在那类特定图时，所有关于无向图的 NP 完全问题都可在多项式时间内解决？

11.31 设 $G=(V, E)$是无向图，并设$(v_1, w_1), (v_2, w_2), \cdots, (v_k, w_k)$是 G 的 k 对不同顶点。证明以下问题是 NP 完全的：判定 G 中是否存在 k 条路径，使得路径 i 连接 v_i 和 w_i 并且所有路径均不含相同顶点。

11.32 设 $G=(V, E)$是无向图，使得每个顶点都与某项任务相关联。如果对应的任务不能同时执行（例如它们需要相同的资源），那么对应顶点将相连，这是关于并行性的惟一限制。对于任何任务集合，只要它们两两不相连则均可在一步内完成。证明以下问题是 NP 完全的：给定图 $G=(V, E)$和整数参数 k，判定所有相对应的任务是否可以在 k 步内完成。

*11.33 设 $G=(V, E)$是无向图，使得每个顶点所关联的边以循环顺序排列，不同节点间的顺序没有任何关系，进一步假定 G 是欧拉图。问题是找到覆盖 G 的欧拉行程满足以下“非交叉”性质：如果行程在边 e 处进入顶点 v，那么该行程的下一条边就是按照与 v 相关联边的循环顺序与边 e 相邻（可在 e 的任何一面）的边。看待该性质的一种方法是查看一幅地图，目标是旅行经过所有的边（路段），使得只有当从一条边到它的邻边时才穿越交叉口（顶点）。证明对于给定图和循环顺序，判定是否存在欧拉行程的问题是 NP 完全的。（对于平面图，该问题仍然是 NP 完全的，但是证明更加困难。）

11.34 给出一个例子，说明 11.5 节中所描述的，用于解决图的 3 着色问题的简单回溯算法可以导致指数多个节点。

11.35 下面是寻找顶点覆盖的一个简单启发式策略，在这个算法的每个步骤中，度最高的顶点（度相同的顶点可任意选取）被添加到覆盖中，然后将它连同它所相关的边一起从图中删除，当没有任何边留下来时，算法终止。对于任意一条边，由于只有当它所相关的一个顶点被加到覆盖后才被删除，所以该算法确实找到了一个顶点覆盖。因为它总是选取有最高“回报”的顶点，这是一个贪婪算法。最坏情况下该算法的表现并不是太好，请给出一个图以及该算法的执行顺序，结果所找到顶点覆盖的顶点个数超过最小覆盖顶点个数的两倍。

11.36 给出一个箱柜包装问题的例子，使得首次适应算法给出的解所用的箱子个数超过 5/3 倍的最优解所用的箱子个数。

11.37 假定你是一位 20 世纪 80 年代的旅行商，而老板要求你访问 n 座城市，所以你正在计划整个飞行旅程，由于公司会支付所有开销，大可不必在乎旅行成本。你打算最大化的是你乘坐飞机的收益！换句话说，你希望将整个旅程的长度最大化。可以假定航线所计算的里程是实际直线距离，因此这是一个欧几里得问题，并且你不必担心不同航线的选择，只要将总的里程数最大化即可。证明这个问题是 NP 完全的，并且为它提出一个近似算法。

11.38 设 $C_1, C_2, \cdots, C_n$ 是某大学所提供的课程集合，而 $t_1, t_2, \cdots, t_n$ 是这些课程的授课时间间隔（允许时间有重叠）。例如，t_1 可以是从星期四的 10:00 到 11:00，而 t_2 可以是从星期四的 10:30 到 12:00，如此等等。你的工作是为这些课程安排教室，惟一的要求是任何两门课程不能在同一时间使用相同的教室，而目标是在满足上述要求的前提下尽可能少地使用教室。

a. 将这个问题归约到着色问题，并且设计一个高效算法来解决它。

b. 讨论你所给出的解为什么不能蕴含 P = NP（即使着色问题是 NP 完全问题）。

第12章 并行算法

有一个钟表的人知道时间；有两个钟表的人则无所适从。

——佚名

首先，必须注意，尽管共和国可能很小，但为了防止少数人的阴谋，必须有一定数量的议员；另外，尽管共和国可能很大，但为了防止民众混乱，议员人数也必须限定为某一数额。

——詹姆士·麦迪逊（1787）

12.1 引言

近10年来，并行计算已经从计算机科学的边缘成为了计算机科学的主流，与计算机科学的其他领域相比，它的发展是非常迅速的。目前有多种形式的并行计算机，所包括的处理器数也从2到65 536不等。现存的各种并行计算机的差异，就算对没有用过的人来说，也是巨大的。我们再不能假定有一个“通用”的计算模型并希望它适合所有的并行计算机。并行算法的设计、分析和正确性证明比起相应的串行算法来要困难得多。

不能期望短短的一章就可覆盖并行计算的所有或大部分领域，这里只给出了使用不同计算模型和不同技术的例子，并尝试展示并行算法的风格并探究并行算法设计的困难。本章以并行计算共同特性的介绍开始，接着是本章所使用的主要并行计算模型的简单描述，最后是关于算法和技术的例子。

衡量串行算法复杂度的主要标准是运行时间和所占用的空间，这些度量对于并行计算也同样重要，但我们还必须顾及其他的资源开销。一种重要的资源是处理器数，有些问题是固有串行的，它们不能被“并行”化，因此就算有任意多的处理器也无济于事。然而，大多数问题可以在某种程度上并行化，使用的处理器越多（在某个界限之内）算法就越快。研究并行算法的界限，探讨可并行求解的问题的特性，是很重要的。然而，由于处理器数目有限，如何有效地使用这些处理器也很重要。另一个重要的问题是处理器间的通信。通常，交换数据所花费的时间要比对数据进行简单操作所花费的时间长。进一步，一些处理器可能相隔较近，而另一些之间的距离则可能很远。因此，以有效的方式放置处理器以减少通信开销就很重要。还有一个重要的问题是同步，它对于那些运行在由某种通信网络连接在一起的计算机上的并行算法来说是主要的问题。这种形式的算法通常被称为分布式算法。由于篇幅有限，本书不准备讨论分布式算法，而仅讨论那些假定完全同步的模型。

一些并行计算模型要求所有的处理器每步执行完全相同的指令，遵循这一限制的并行计算机被称为SIMD（单指令流多数据流）机器，连接机（connection machine）就是这类机器的一个显著的例子。如果并行计算机内不同的处理器可以执行不同程序，那么该并行计算机就被称为MIMD（多指令流多数据流）机器。除非特别指出，本书假定讨论的计算机都是MIMD的。

12.2　并行计算模型

介绍所有的并行计算模型并不是本书的主题，这里只涉及一些主流的以及那些将在本章中用到的模型。本节包括一些适用于大多数模型的概述性的讨论和定义，后面每一节都将考察一种模型，包括其详细描述和算法例子。

将算法的运行时间记为 $T(n, p)$，其中 n 是输入的大小，p 是处理器的数目。比率 $S(p) = T(n, 1)/T(n, p)$ 被称为算法的加速比。当 $S(p) = p$ 时并行算法是最有效的，因为在这种情况下，算法获得了完美的加速比。$T(n, 1)$的值应取自众所周知的串行算法。处理器利用率的一个重要度量是并行算法的效率，定义为

$$E(n, p) = \frac{S(p)}{p} = \frac{T(n, 1)}{pT(n, p)}。$$

效率是一个处理器（运行串行算法）所花费的时间与 p 个处理器所花费的总时间的比率（总的时间是实际流逝时间乘以处理器的数目）。效率显示了与串行算法相比，未浪费的处理器时间的百分比。如果 $E(n, p) = 1$，则在算法执行过程中所有处理器完成的工作量的总和等于串行算法所需要的工作量，在这种情况下，算法取得了最优的处理器使用效率。获得最优效率的机会是很小的，在大多数情况下，并行算法会引入一些对串行算法来说并不需要的开销。而我们的目标是使效率最大化。

在设计并行算法的时候，可以将 p 固定为已有的处理器数，再尝试使 $T(n, p)$最小，但若如此，当处理器的数目改变时，就必须设计新的算法。设计一个对不同的处理器数均能工作的算法是值得的。下面讨论如何将一个工作在某一 p 值上的算法转换为工作在一个处理器数比 p 小的机器上而不损失效率的方法。一般来说，对任意常数 $k > 1$，我们可以将 $T(n, p) = X$ 的算法修改为 $T(n, p/k) \approx kX$ 的算法。换言之，可以用原来 k 分之一的处理器运行 k 倍时间来完成同样的工作。算法修改的原理是通过一个处理器执行 k 步计算来仿真原有 k 个处理器完成的一步工作。但这个原理不能适用于所有情况。例如，k 或许不能整除 p，算法可能依赖于处理器的互连模式（将在 12.4 节讨论），另外决定哪些处理器进行仿真也将花费一定的时间。然而，该并行折叠原理（parallelism folding principle）不仅普遍而且有用，它表明我们可以在不影响效率的情况下减少处理器的数目。例如，如果原来的（对于一个较大的 p 所设计的）程序有很好的加速比，则对任何小的 p 也可得到同样的加速比。由此，我们应该对最大的 p（只要效率良好，即接近 1）考虑如何得到最好的加速比，以后，就算可用的处理器数较少，也可以继续使用同一算法。另一方面，效率较低的并行算法通常对大量的处理器来说也可能有用。例如，假定有一个算法满足 $T(n, 1) = n$，$T(n, n)=\text{lb}n$，这样加速比 $S(n) = n / \text{lb}n$ 是一个令人印象深刻的加速比，效率 $E(n, n) = 1/\text{lb}n$。假定处理器数 $p = 256$，$n = 1024$。并行算法的运行时间 $T(1024, 256) = 4\text{lb}\ 1024 = 40$（假定并行折叠原理有效），加速比大概是串行算法的 25 倍。另一方面，如果 $p = 16$，则总的运行时间是 640，加速比不好（使用 16 个处理器，却小于 2）。

不同计算模型之间的区别在于处理器间不同的通信和同步方式。这里仅考虑那些假定完全同步的模型，由此我们侧重于不同通信范例的讨论。共享存储器模型假定存在一个随机存取的共享存储器，任何处理器均可在单位时间内访问共享存储器中的任何变量。这种忽略处理器数目和内存大小的单位时间耗费的假设并不现实，但它是一种好的近似。共享存储器模型间的差别在于存取冲突的处理。12.3 节将讨论几种不同的选择方案。

共享存储器模型通常便于对通信的刻画，但硬件实现起来非常困难。其他模型假定处理器间通过互连网络连接。互连网络可表示为图，其中节点表示处理器，如果两个节点间存在一条边则

对应的两个处理器间有一条直接链路。每个处理器通常包括局部存储器，对其访问通常是快捷的。通信是通过消息传递完成的，一个消息在到达目的地前可能要经过多个链路。因此，通信的速度依赖于发生通信的处理器间的距离。计算机学家研究了许多不同的互连拓扑结构。12.4 节将讨论其中的一些。基于互连网络和消息传递的并行计算机有时也被称为多计算机（multicomputer）。

另一种在本书中讨论的模型是脉动计算。脉动体系结构类似于工厂中的一条装配线。数据以和谐的方式在处理器间流动，期间处理器对其施以简单的操作。与访问共享（或非共享）存储器中的数据不同，这里处理器先从它的邻接节点获得数据，然后对其进行操作，最后再传送出去。12.5 节将讨论脉动算法。

本书说明算法的基本理论模型是线路（circuit）。一条线路是一个有向无回路图，其中节点对应于计算，边对应于操作数的流动。例如，布尔线路是所有节点入度最多为 2，所有操作均为布尔操作（*or*、*and* 和 *not*）的线路，该线路包括一些指定的输入节点（入度为 0）和输出节点（出度为 0）。布尔线路的深度定义是从输入到输出的最长路径的长度，其相应于并行执行时间。

12.3 共享存储器算法

共享存储器计算机包含若干处理器和一个共享存储器。本节仅讨论完全同步算法。我们假定一个计算可分为若干步，每个处理器在每一步可以对它持有的数据进行一次操作，也可从共享存储器中读取数据，或将数据写入共享存储器。（实际上，每个处理器可能也有自己的局部处理器，但我们假定所有处理器都是全局的）。共享处理器模型的区别在于内存冲突处理方式的不同。不允许同时读和同时写的 EREW 模型不允许多于一个处理器同时访问同一内存位置；允许同时读但不允许同时写的 CREW 模型允许多个处理器同时读取内存的同一位置，但仅允许一个处理器写入数据；而允许同时读和同时写的 CRCW 模型对内存冲突未提出任何限制。

EREW 和 CREW 都是明确定义的模型，但如果有两个处理器在同一时刻写入同一内存位置，则其结果为何并不清楚。存在若干种处理并发写的处理选择。本书仅讨论最弱的 CRCW 模型，它允许多个处理器在同一时刻将数据写入同一内存位置，前提是这些数据是相同的。如果有两个处理器在同一时刻将不同的数据写入同一内存位置，则算法终止。正如我们将在 12.3.2 节中看到的那样，令人惊讶的是，这个简单特性所包含的表示能力非常强大。另一种选择是假定每个处理器都有一个标记，当有多个处理器想对一个内存位置写入数据时，具有最高标记的处理器胜出。还有一种考虑是选择任意处理器执行写操作。

12.3.1 并行加

先通过归纳法来设计一个简单问题的并行算法，该问题乍一看来似乎是固有串行的。

问题 求两个 n 位二进制数的和。

通常的串行算法是从两个二进制数的最低位开始，将相应位置上的数字以及可能的进位相加。由于不知道是否存在进位，似乎在得到第$(i-1)$步运算结果之前无法确定第 i 步的输出。然而，可以设计另一种算法。

对 n 进行归纳。因为这意味着是一个串行的迭代算法，从 $n-1$ 到 n 进行归纳于事无补。对于并行算法来说，分治法可能是一个更好的潜在选择，因为它可以让我们对子问题并行求解。假定我们将问题分为两个大小为 $n/2$ 的子问题（为了简单起见，假定 n 是 2 的幂）。现在我们能得到两对数的和，但仍然需要面对进位的问题。如果低位的两个二进制串存在进位，则必须修改两个高位二进制串相加的结果。

注意，这里只有两种可能性，有进位或者没有进位。由此，我们可以加强归纳假设以包括这两种情况。修改后的问题是：当存在初始进位或者不存在初始进位时寻找两个二进制数的和。假定我们已经解决了两对子串的问题，并得到了 4 个数，L、L_c、R 和 R_c，它们分别对应于不存在进位时低位两个二进制串的和、存在进位时低位两个二进制串的和、不存在进位时高位两个二进制串的和，以及存在进位时高位两个二进制串的和。对于这些和，我们一样可以知道在计算它们时是否存在进位。没有初始进位的最后的和 S 等于 L 与 R 或者 L 与 R_c 组成的串，依赖于计算 L 时是否存在进位；而 S_c，除了将上面的 L 替换为 L_c 外，与之类似。

这样，通过解决两个大小为 $n/2$ 的问题加上一些常数数目的（分治）计算步骤就得到了大小为 n 的问题的解答。由于两个子问题可以并行求解，故假定处理器可以对一个二进制串的不同数位分别存取，则可得递归关系 $T(n, n) = T(n/2, n/2)+O(1)$，由此有 $T(n, n) = O(\log n)$。进一步，由于两个子问题是完全独立的，该算法仅采取 EREW 模型。对于并行加法，这个算法可能并不是最好的（参见 Ladner and Ficher [1980]），但它是一个算法并行化的优异例子。因此一旦问题变得清晰，就可以非常快地以并行方式求解，当然解决方案也可以进一步得以改进。

12.3.2　寻找最大数的算法

问题　对存放在给定数组中的 n 个不同的数找出其中的最大者。

下面分别对两种不同的共享存储器模型——EREW 和 CRCW——给出问题的解。

设计两个模型上的算法所使用的技术也适用于其他问题。

EREW 模型

寻找最大数的直截了当的串行算法需要有 $n-1$ 次比较。我们可以将数的比较想像成两个数间的游戏，其中大者赢。由此寻找最大者的问题就等价于一场锦标赛，其中胜者是整个数集上的最大者。锦标赛并行运作的一个有效方法是使用树这种数据结构。在第 1 轮，选手分对进行博弈（如果选手数为奇数，则可能有选手轮空），所有胜出的选手再捉对厮杀，直到最后。易见这里的博弈轮数是$\lceil \mathrm{lb}n \rceil$。通过给每一对数间的博弈分配一个处理器（可以将处理器想像为裁判），从锦标赛可以得到一个并行算法。然而，我们必须确认每个处理器知道两个参加博弈的数，这可通过将博弈的胜者置于两个博弈者中具有较大索引的博弈者所在位置来实现。也就是说，如果博弈在 x_i 和 x_j 之间进行，其中 $j > i$，则 x_i 和 x_j 间的最大者将被置于位置 j。在第 1 轮，P_i 比较 x_{2i-1} 和 x_{2i}（$1 \leqslant i \leqslant n/4$）并在必要时进行交换，在第 2 轮，$P_i$ 比较 x_{4i-2} 和 x_{4i}（$1 \leqslant i \leqslant n/4$）并在必要时进行交换，以次类推。因为任何时候每个数仅涉及一个博弈，因此 EREW 可以满足需要。该简单算法的运行时间显然是 $O(\log n)$。下面让我们尝试减少处理器的数目。

上面提及的算法需要的处理器数是$\lfloor n/2 \rfloor$，并且有$T(n, \lfloor n/2 \rfloor) = \lceil \mathrm{lb}n \rceil$。由于对串行算法 $T(n, 1)=n-1$，故并行算法的效率是 $E(n, n/2) \approx 1/\ \mathrm{lb}n$。如果无论如何都有 $n/2$ 个处理器（例如，寻找最大数的算法可能是其他算法的一部分，而后者需要这些处理器），那么这个算法既简单又有效。经过一些修改，还可以得到并行时间 $O(\log n)$而效率为 $O(1)$。

对于本算法所需要的比较次数是 $n-1$，这与串行算法是一样的。算法低效的原因在于许多处理器在后面的博弈中是空闲的。下面通过减少处理器数以及负载平衡来改进效率。假定有大约 $n/\mathrm{lb}n$ 个处理器，我们将输入分为 $n/\mathrm{lb}n$ 组，每组 $\mathrm{lb}n$ 个数，分配一个处理器。在第 1 阶段，每一个处理器使用串行算法找出该处理器所在组的最大数，该开销为 $\mathrm{lb}n$。剩下的问题是在得到的 $n/\mathrm{lb}n$ 个数中找到最大者，由于我们有足够的处理器，所以可以使用上面的锦标赛算法。该算法的运行时间为（假定 n 是 2 的幂）$T(n, \lceil n/\mathrm{lb}n \rceil) \approx 2\mathrm{lb}n$。相应的效率为 $E(n) \approx 1/2$。下面就对节省处理器的思想进行形式化描述。

如果对于一个并行算法中的操作，处理器的分配是预先定义的，则称该算法是静态的（static）。对于静态算法的第 i 步，我们都预先知道与第 j 个处理器相关的操作符和操作数。例如，寻找最大数的算法就是一个静态算法，因为所有的“博弈”都是预先安排的。

□ 引理 12.1

如果存在一个静态的 $T(n, p) = O(t)$的EREW算法，其中所有（处理器上）的计算步数均为 s，则存在一个静态的满足 $T(n, s/t) = O(t)$的EREW算法。

（注意，如果 s 等于问题串行求解的复杂度，则修改后算法的效率为 $O(1)$。）

证明：令 a_i（$i = 1, 2, \cdots, t$）为所有处理器在算法第 i 步完成的计算步数，则 $\sum_{i=1}^{t} a_i = s$。如果 $a_i \leqslant s/t$，则有足够的处理器来执行算法的第 i 步，无须修改；否则，用$\lceil a_i/(s/t) \rceil$步来代替第 i 步，使用可利用的 s/t 个处理器来仿真原算法 p 个处理器完成的工作（依并行折叠原理）。现总的步数为

$$\sum_{i=1}^{t} \left\lceil \frac{a_i}{(s/t)} \right\rceil \leqslant \sum_{i=1}^{t} \left(\frac{a_i \cdot t}{s} + 1 \right) = t + \frac{t}{s} \cdot \sum_{i=1}^{t} a_i = 2t。$$

因此，修改后算法的运行时间仍是 $O(t)$。 □

引理 12.1 又称 Brent 引理（参见 Brent [1973]，该文中包含一个与这里思想相同但适用于更复杂情况定理的证明）。Brent 引理表明，在某些情况下，并行算法的效率主要依赖于所有处理器所完成的计算数与串行算法运行时间之间的比率。

这里局限于静态算法的原因在于我们必须知道哪个处理器执行仿真工作。只要仿真工作能快速地完成，引理 12.1 也适用于非静态算法。在非静态算法第 1 步之后，一些处理机（按照第 1 步所得的结果）决定退出。同样的事情发生于第 1 步之后、第 3 步之后，如此等等。除了我们不知道哪些处理器退出以外，该算法与锦标赛算法类似。如果第 1 步之后想要仿真剩余处理器的工作，那么必须知道哪些处理器是活跃的，但这个过程可能会耗费若干计算时间。

CRCW 模型

乍一看来，似乎如果仅使用比较，并行算法无法在小于 lb$n$ 步内寻找到最大数。但答案并非如此，下面就是一个算法复杂度为 $O(1)$的算法，该算法说明了允许并发写带来的能力。这里采用的并发写版本为：只要写入的数据是一样的，就允许两个或多个处理器向内存的同一位置写入数据。

使用 $n(n-1)/2$ 个处理器，对每一个处理器 P_{ij} 分配一对元素$\{i, j\}$，每个元素 x_i 分配一个共享变量 v_i，其初始值为 1。在算法的第 1 步，处理器比较它的两个元素并将 0 写入较小者对应的共享变量。由于只有一个元素比其他元素大，因此只有它，其相应的共享变量的值仍为 1（参见习题 12.12）；在第 2 步，与胜出的元素相关的处理器可以确定胜者并宣布这一事实。该算法仅需两步，并且独立于 n，然而，由于需要 $O(n^2)$个处理器，算法的效率仍很低。通常将该算法称为两步算法（two-step algorithm）。

可以使用类似于 EREW 模型的方法来改善两步算法的效率。首先将输入分成组，由此可以为每组分配足够的处理器以便使用两步算法来找出其中的最大者。当候选者减少时，每一个候选者可使用的处理器数就增加了，与之相应的是，组的大小也变大了。两步算法表明：如果组的大小为 k，则有 $k(k-1)/2$ 个处理器就可以在常数时间内找到组中元素的最大者。假定有 n 个处理器，这里 n 是 2 的幂。在第 1 轮，每组大小为 2，可一步找到每组的最大者。在第 2 轮，有 $n/2$ 个元素留下，但我们仍有 n 个处理器。如果令每组的大小为 4，则有 $n/8$ 个组，每组可有 8 个处理器，

由于 4×(4–1)/2 = 6，对于两步算法来说，处理器数足够了。在第 3 轮，有 $n/8$ 个数，现在来计算每组可有多少元素。如果每组有 g 个元素，则有 $n/8g$ 个组，每组有 $8g$ 个处理器。为了使用两步算法，需要 $g(g-1)/2$ 个处理器，必须有 $g(g-1)/2 \leqslant 8g$，即 $g \leqslant 17$（简单地令 $g = 16$）。读者可验证组的大小每轮都是前一轮的平方，由此算法需要 $O(\log\log n)$轮的计算。

尽管该算法比两步算法慢一点（即 $O(\log\log n)$与 $O(1)$相比），但它的效率好得多，为 $O(1/\log\log n)$，而两步算法为 $O(1/n)$。该技术又称分压法（divide and crush），因为是先将输入分为足够小的能被多个处理器压服的组。该技术的应用领域不限于 CRCW 模型。

12.3.3　并行前缀问题

并行前缀问题非常重要，它是许多并行算法的基本构件。令•是满足结合律的二元运算，即满足 $x•(y•z)=(x•y)•z$，通常被称为积。例如，•可以表示加法、乘法，或求两个数的最大者。

问题　*考虑数的序列 $x_1, x_2, \cdots, x_n$，对所有 k，$1 \leqslant k \leqslant n$，计算 $x_1 • x_2 • \cdots • x_k$。*

记 $x_i \cdot x_{i+1} \cdot \cdots \cdot x_j$ 为 $PR(i, j)$。算法的目标是对所有 k，$1 \leqslant k \leqslant n$，计算 $PR(1, k)$。前缀问题的串行算法简单而直接，只要按照顺序进行即可。并行前缀问题就没那么简单了。这里使用的方法是分治法，如常，假定 n 是 2 的幂。

归纳假设　*假设已经知道如何对 $n/2$ 个元素求解并行前缀问题。*

只有一个元素的情况不证自明。当 $n > 1$ 时，算法将输入分成两半，再通过归纳分别求解。按归纳假设，对所有 $1 \leqslant k \leqslant n/2$，已经得到 $PR(1, k)$和 $PR(n/2+1, n/2+k)$的值。前面半部分可以直接使用，而 $PR(1, m)$的值，其中 $n/2 \leqslant m \leqslant n$，可以通过计算 $PR(1, n/2) • PR(n/2+1, m)$得到，后面两项按归纳假设是已知的（注意这里的运算满足结合律）。算法见图 12.1。

复杂度　输入在每次递归调用时被分成两个不相交的集合。每个子问题都可以在 EREW 模型下并行求解。如果对于大小为 n 的问题，有 n 个处理器，则对每个子问题可以分配 $n/2$ 个处理器。合并运算需要 $n/2$ 步，当然也可以并行进行，但由于所有的处理器都用到了 $x[Middle]$，因此必须在 CREW 下进行。尽管所有处理器都必须在同一时间内读取 $x[Middle]$的值，但它们写入的内存位置不同，因此不需要 CRCW 模型。大体上，$T(n, n) = O(\log n)$，$E(n, n) = O(1/\log n)$（因为串行算法的运行无疑需要 $O(n)$步）。

遗憾的是，这里不能使用 Brent 引理来改善算法的效率。总的运算步数是 $O(n\log n)$，其浪费来自于第二个递归调用，与之相反的是，串行算法可以计算所有前缀而无须进行与第二个递归调用相应的工作。因此，为了改进性能，必须重新设计算法以减少总的计算步骤。下面就来研究这个问题。另外该问题的（具有相同资源约束）EREW 算法留为练习（见习题 12.18）。

并行前缀算法的改进

技巧仍然是使用归纳法，但对输入的划分方式不同。这里仍然假定 n 是 2 的幂并且有 n 个处理器。令 E 表示所有 x_i 的集合，其中 i 是偶数。如果得到了 E 中所有元素的并行前缀，那么要得到剩余元素（下标为奇数的那些元素）的并行前缀就很容易了：假定对所有 i，$1 \leqslant i \leqslant n/2$，$PR(1, 2i)$已知，要得到所有奇数前缀 $PR(1, 2i+1)$的值，实际上只需要计算 $PR(1, 2i)•x_{2i+1}$。可以分两个阶段计算 E 中元素的前缀。首先，对所有 $1 \leqslant i \leqslant n/2$（并行）计算 $x_{2i-1}•x_{2i}$并将结果存在 x_{2i}。换句话说，我们计算了所有 E 中元素及其左邻居的乘积；然后，（通过归纳法）求解 E 的前缀，这是一个大小为 $n/2$ 的问题。结果，对每个 x_{2i}，算法都将得到一个正确的前缀，这是因为 x_{2i} 已经包含其与 x_{2i-1} 的乘积。另外既然我们知道所有下标为偶数的元素的前缀，那么也就知道了如何并行求取下标为奇数的元素的前缀。读者可以验证该算法仅需要 EREW 模型。算法见图 12.2（见习题 12.17）。

```
算法 Parallel_Prefix_1(x, n)
输入：x(范围为 1 到 n 的数组)
   {假定 n 是 2 的幂}
输出：x(第 i 个元素包括第 i 个前缀)

begin
    PP_1(1, n) ;
end

procedure PP_1 (Left, Right) ;

begin
    if Right - Left = 1 then
        x[Right] := x[Left] • x[Right] { • 是一个满足结合律的二元运算 }
    else
        Middle := (Left + Right - 1)/2 ;
        do in parallel
            PP_1(Left, Middle) ;
            PP_1(Middle + 1, Right) ;
        for i := Middle + 1 to Right do in parallel
            x[i] := x[Middle] • x[i]
end
```

图 12.1 算法 *Parallel_Prefix_1*

```
算法 Parallel_Prefix_2 (x, n)
输入：x(范围为 1 到 n 的数组)
   {假定 n 是 2 的幂}
输出：x(第 i 个元素包括第 i 个前缀)

begin
    PP_2(1) ;
end

procedure PP_2 (Inc) ;

begin
    if Inc = n/2 then
        x[n] := x[n/2] • x[n] { • 是一个满足结合律的二元运算}
    else
        for i := 1 to n/(2 · Inc) do in parallel
            x[2 · i · Inc] := x[2 · i · Inc - Inc] • x[2 · i · Inc] ;
        PP_2(2 · Inc) ;
        for i := 1 to n/(2 · Inc) - 1 do in parallel
            x[2 · i · Inc + Inc] := x[2 · i · Inc] • x[2 · i · Inc + Inc]
end
```

图 12.2 *Parallel_Prefix_2* 算法

复杂度 算法 *Parallel_Prefix_2* 的两个循环使用 $n/2$ 个处理器皆可在 $O(1)$时间内完成。递归调用将算法应用于大小为原来一半的问题，因此算法的运行时间为 $O(\log n)$，总的时间耗费 $S(n)$满足等式 $S(n) = S(n/2)+n-1$，$S(2) = 1$，即 $S(n) = O(n)$。但这表明现在可以使用 Brent 引理了，即可将算法改为一个仅使用 $O(n/\log n)$个处理器并在 $O(\log n)$步内完成的算法，由此得到的效率为 $O(1)$。

改善算法效率的关键是仅使用一次（而不是两次）递归调用并以并行方式执行结果的合并。

12.3.4　在链表中查寻秩

一般说来，在并行环境下，处理链结构要比处理数组困难得多。例如，链表就是一种固有串行的结构。如果对链表的惟一存取方式是通过表头，那么必须每次一个元素以横越整张表，毫无并行性可言。然而，在很多情况下，链表的元素（或指向元素的指针）是顺序存放在一个连续数组中的，元素在表结构中的顺序与数组无关。在这种情况下，对表元素的并行存储是可能的，由此有希望设计出快速的并行算法。

在一个链表中，一个元素的秩（rank）是该元素到表尾的距离（因此，头元素的秩为 n，第二个元素的秩为 n–1，依次类推）。

> **问题**　给定一个包含 n 个元素的链表，所有元素都被存放在数组 $A[1..n]$ 中，计算每个元素在表的秩。

串行求解该问题的方法是简单地横越链表。这里将用一种名为加倍（doubling）的方法来并行求解该问题，首先给每个元素分配一个处理器，一开始，每个处理器只知道右边的毗邻，然后，每个处理器寻找其毗邻的毗邻。第 1 步结束时，每个元素知道与其相隔距离为 2 的元素。如果在第 i 步，每个处理器知道与其相隔距离为 k 的元素，则下一步，每个处理器知道与其相隔距离为 $2k$ 的元素。令 $N[i]$是在给定的某时刻，即 P_i 所知道的离 i 最远的右边元素。一开始，$N[i]$是 i 的右边的毗邻（除了最后一个元素，它的右边毗邻为 nil）。基本说来，每一步，P_i 将 $N[i]$的值修改为 $N[N[i]]$，直至达到最后一个元素。令 $R[i]$为 i 的秩，开始时每个元素的 $R[i]$值设为 0，最后一个元素（可由其指向下一个元素的指针值是 nil 来判定）是个例外，它的 $R[i]$值设为 1。当一个处理器遇到一个秩 R 不为 0 的邻居时，它就可以确定自己的秩了。初始时，只有秩为 1 的元素知道自己的秩，第 1 步之后，秩为 2 的元素知道它右边邻居的秩为 1，因此它知道自己的秩是 2。第 2 步之后，秩为 3 和 4 的元素的秩将被确定，依次类推。如果 P_i 在 d 次加倍之后发现 $N[i]$指向一个已知秩为 R 的元素，则第 i 个元素的秩为 $2^{d-1}+R$。详细算法见图 12.3。可容易地将该算法改成在 EREW 模型下运行（见习题 12.4）。

算法 ***List_Rank*** *(N)*
输入： N（下标范围为 1 到 n 的数组）
输出： N（数组中每个元素的秩）

```
begin
    D := 1 ;
    { 每个处理器有它自己的局部变量 D;
      这里只使用变量 D }
    do in parallel { 处理器 Pi 是活跃的，直到 R[i]非 0 为止 }
        R[i] := 0 ;
        if N[i] = nil then R[i] = 1 ;
        while R[i] = 0 do
            if R[N[i]] ≠ 0 then
                R[i] := D + R[N[i]]
            else
                N[i] := N[N[i]] ;
                D := 2D
end
```

图 12.3　算法 *List_Rank*

复杂度 加倍过程保证了每个处理器最多在⌈lb*n*⌉步之后到达表的末端。因此 $T(n, n) = O(\log n)$。效率 $E(n, n) = O(1/\log n)$。由于总的工作量是 $O(n\log n)$，因此效率的改进需要对算法做出大的改进（见参考文献中最近的讨论）。

有了秩，我们就可在 $O(\log n)$时间内将一个链表转换为数组（尽管效率不是很完美）。在计算出所有的秩之后，可将元素复制到数组的适当位置，剩下的计算就可直接在数组中执行，这比链表上的计算快得多。

12.3.5 欧拉遍历技术

使用对树中位于同一层的节点同时进行处理的技术（例如寻找最大树的锦标赛算法）可以对许多类型的基于树的算法进行并行化。这种算法的运行时间与树的高度成比例。如果树是比较平衡的，则它的高度约为 $O(\log n)$（这里 n 是节点的数目），那么该方法可以说是相当好的。然而，如果树不平衡，树的高度最高可能是 $n-1$，那么此时就需要另外的方法。欧拉回路技术是一种有助于设计树并行处理算法的技术，特别对非平衡树的处理特别有用。

令 T 是一棵树，使用附加有一个额外特性的通常的邻接表来表示。如常，指针 $E(i)$指向由与节点 i 相关联的边所组成的表的第一个元素（若该表为空，则 $E(i)$值为 nil）。每个表元素包含的信息是相应的边(i, j)（由于 i 已知，因此仅存储 j 的值就可以）以及指向表中下一个元素的指针 Next(i, j)。每一条（无向）边(i, j)被表示为两个副本，即两条有向边，一为(i, j)，另一为(j, i)。额外的特性是每条边都有一指向另一副本的指针。有了该特性，当给定边(i, j)时，就可以快速找到边(j, i)。该表示法可由图 12.4 说明（图中未给出连接两个副本的指针）。

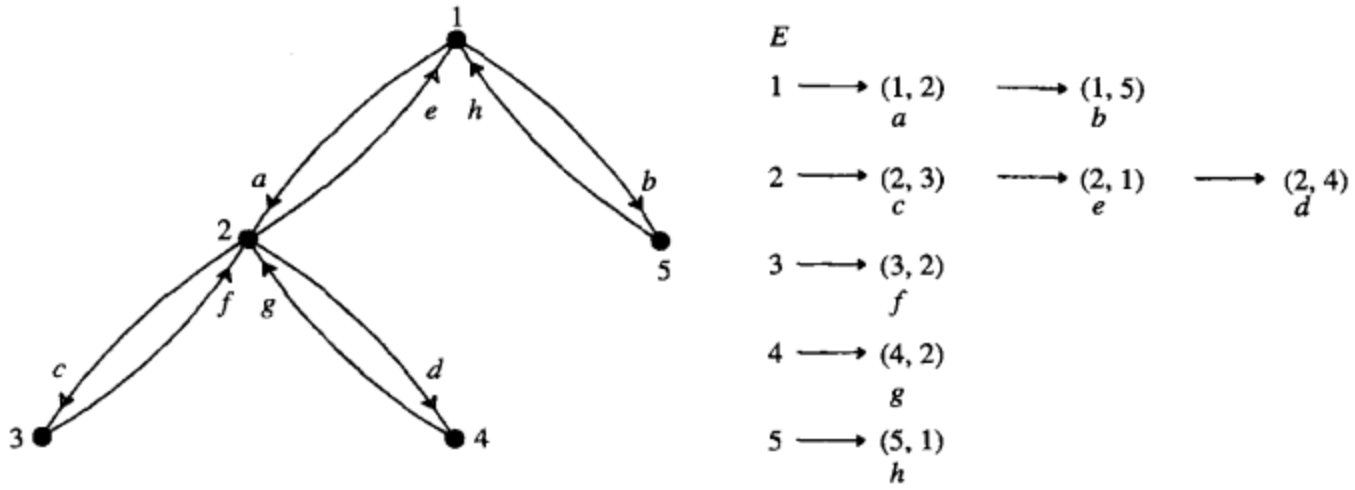


图 12.4 树的表示

欧拉遍历技术的主要思想是构造边的一个序列，该序列是树的有向形式的一个欧拉回路（其中每条边出现两次）。一旦得到了该表，对树的许多操作就可以转成对表的直接操作，而表是线性的。这样，一个串行算法总能横越一棵树且在横越的同时执行运算。这种树的线性化使得我们可以对树进行有效的并行操作。下面将给出两个例子，但首先来讨论一下如何构造欧拉遍历。

使用串行方法得到树 T（其中每条边出现两次）的欧拉遍历是非常容易的。可以使用深度优先搜索算法进行树的遍历，在搜索回溯时取边的另一个方向即可。在并行处理时我们可使用类似的方法。令 *NextTour*(i, j)是对树遍历时在边(i, j)之后被访问的边。下面说明 *NextTour* 可按如下规则定义并且可以方便地进行并行计算：

$$NextTour(i,j)=\begin{cases}Next(j,i)\text{，}& \text{如果 } Next(j,i) \text{ 值不是 nil}\\ E(j)\text{，}& \text{其他}\end{cases}$$

换句话说，与一个节点相关联的边是以循环方式安排的，即如果(j,i)是j节点的邻接表的最后一个元素，则取该表的第一条边，也就是$E(j)$所指向的边。例如，如果从图 12.4 中的边a开始（假定与一个节点关联的边是以顺时针排序的），则边的遍历次序为a、d、g、c、f、e、b、h最后到a。在遍历过程中让(i,j)之后是 Next(j,i)保证了仅在所有与j相关联的边被考虑之后才考虑(j,i)。因此，在回溯到i时以j为根的子树已被完全遍历。该命题的正确性留为练习。

一旦构造出表，算法可以选择从任何一条边(r,t)（任何边皆可）开始并结束于(r,t)之前的边。假定节点r是树的根。现在可以用图 12.3 中的算法 *List_Rank* 对表中的边按照其位置赋予一个序号。令$R(i,j)$是边(i,j)在表中的秩（如$R(r,t)=2(n-1)$，其中n是节点的数目）。下面给出对树进行操作的两个例子，一是按先序给出节点的遍历次序，二是对所有节点计算出它们的后代的数目。

令(i,j)是遍历回路中的一条边，如果其方向是远离根的，则称其为向前边，否则称其为向后边。可以用边的编码来区分该边是向前的还是向后的。虽然，如果$R(i,j)>R(j,i)$则(i,j)是向前的。由于边(i,j)的两个副本通过指针互连，所以我们可以很方便区分出哪一条边是向前的，哪一条边是向后的。进一步，对所有边可以快速并行地进行这种检测。由于只有向前边与节点的先序访问有关，因此我们只须考虑向前边。令(i,j)是一条指向j的向前边（即在树中，i是j的双亲），如果$f(i,j)$是表中(i,j)之后的向前边的数目，则j的先序数为$n-f(i,j)$（由于没有任何向前边指向根r，因此它的先序数为 1）。对每条向前边(i,j)，可以使用加倍算法的一个变形来计算$f(i,j)$的值，该算法的实现留为练习。

第二个算法要完成的任务是，对每一个节点j，树中位于j之下的节点（即j的子孙）的数目。令(i,j)是（惟一）一条指向j的向前边。考虑在(i,j)之后边的数目。虽然，j下面的节点的数目等于树中j下面的向前边的数目。前面已经讨论$f(i,j)$的并行计算，它的值是在表中(i,j)之后向前边的数目。类似地，可以计算出(j,i)后向前边的数目，易见j的子孙的数目就等于$f(i,j)-f(j,i)$，这也可以通过加倍技术得到。给出树中节点先序数和子孙树的算法在 EREW 模型下都可以在$T(n,n)=O(\log n)$时间内完成。

12.4　互连网络上的算法

互连网络可以表示为图（一般是无向图）。处理器对应于节点，如果两个处理器间存在一条直接的链路则对应的节点间存在着一条边。每个处理器都有一个局部存储器，也可以通过网络访问其他处理器的局部存储器。因此，所有存储器都可被共享，但对一个变量的存取代价将依赖于变量和处理器的位置。对共享存储器的访问可能与对局部存储器的访问一样快捷（如果访问的变量恰好位于同一处理器）或者与横越整个网络一样缓慢（当图是一个简单的链时），而通常的代价介于两者之间。处理器间通过消息通信。当一个处理器想对存放在另一个处理器的局部存储器上的共享变量进行访问时，它就给那个处理器发送一条请求信息。该消息通过网络传送。

作为互连网络的图的结构有许多。最简单的有线性阵列、环、二叉树、星形和二维网孔（网格）。一般说来图中的边越多，通信性能越好，但边的增加代价是昂贵的（会使电路板的面积增加，由此使通信的时间递增）。因此必须找到一个好的均衡点。任何一种图结构的性能都严重依赖于算法的通信模式，因此不存在对所有目标都适用的图结构，然而，有若干性质很有用，下面就列出一些并且结合互连网络来进行说明。

图的直径是非常重要的（一个图的直径是图中任意两个节点间最短距离的最大者），它表示了一个消息在网络传输中所需的最大跳跃数。一个 $n \times n$ 的网孔的直径为 $2n$，而一个 n 节点的平衡树的直径为 $2\text{lb}(n+1) - 2$。因此（在最坏情况下），一棵树比一个网格能更好地传递消息。另一方面，树有一个主要的瓶颈，所有从这半边树的节点到另半边树的节点的消息都要经过根。而二维网格就没有这样的瓶颈，而且很对称，这对以对称方式通信的算法来说是非常重要的。

超立方体是一种非常流行的拓扑结构，它具备高对称、小直径、两个节点间存在多条路由并且没有瓶颈等特点。一个 d 维的超立方体包括 $n = 2^d$ 个处理器，处理器以 $0 \sim 2^d - 1$ 编号。因此每个处理器编号为 d 位的二进制数。如果 i 和 j 相差 1 位，则 P_i 和 P_j 相连。由于最多改变 d 位（每次 1 位）就可以从 P_i 走到 P_j，因此任意两个处理器间的距离不会大于 d。图 12.5 是一个 4 维的超立方体。

由于任意两个节点之间存在许多不同的路由（即可以以任何方式改变合适的位），因此超立方体具备丰富的连接特性。我们也可以将超立方体结构和网格组合在一起，比如将网孔嵌入到超立方体的每一个面。其他网络结构有完美洗牌（perfect shuffle）、立方环（cube-connected cycles）、4 叉树、8 叉树、树网（mesh of trees）和蝶形连接等。

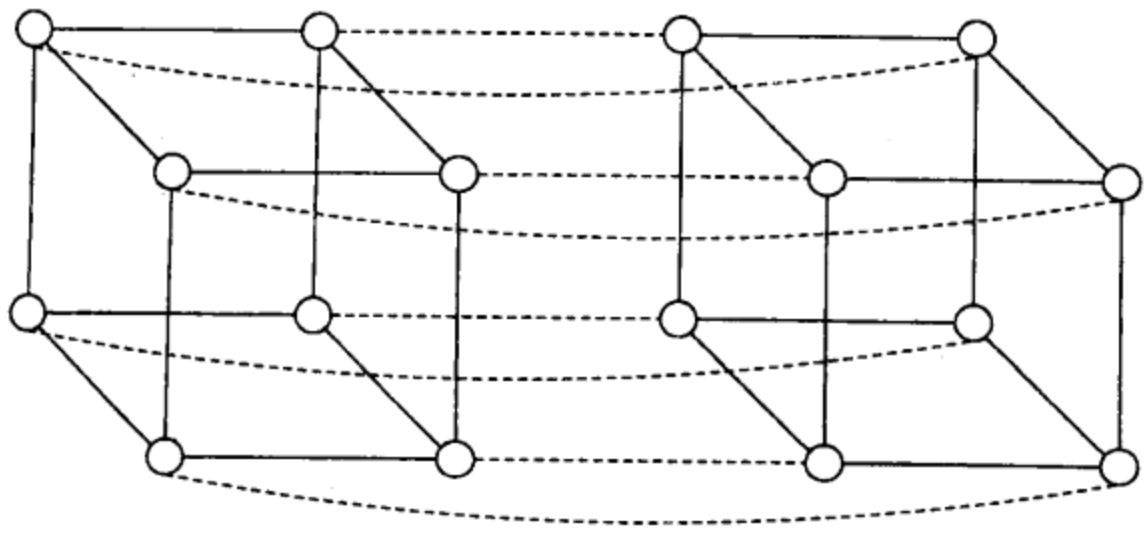


图 12.5　4 维的超立方体

12.4.1　阵列上的排序

下面以一个相对简单的例子，即处理器阵列上的排序开始。假定有 n 个处理器，$P_1, P_2, \cdots, P_n$ 和 n 个输入 $x_1, x_2, \cdots, x_n$。每个处理器保持一个输入值。目标是在处理器上分发输入，使最小的输入保存在 P_1、次小的输入保存在 P_2 中，如此等等。通常可能将多于一个的输入赋予一个处理器。可以看到相同的算法也可用于这种情况。这些处理器是以线性方式连接在一起的，其中处理器 P_i 和处理器 P_{i+1} 相连，这里 $1 \leqslant i < n$。

因为每个处理器只能与它的邻居通信，所以元素间的比较和交换只能发生在阵列上连续的处理器。特别是，在最坏情况下，算法必须允许 $n-1$ 步的操作，这是一个元素从一端移到另一端的时间。算法基本思想如下：每个处理器将它持有的数与它的相邻处理器所持有的数进行比较，如果顺序是错误的，则进行交换，然后再和它的第二个邻居进行相同的工作（必须改变邻居，否则将再次对相同的一对数进行比较）。该过程持续进行直至所有的数以正确的次序排列为止。算法执行的步骤可以区分为奇数步和偶数步。在奇数步，那些标号是奇数的处理器将它持有的数与它的右邻处理器持有的数进行比较；在偶数步，那些标号是偶数的处理器将它持有的数与它的右邻

处理器持有的数进行比较（见图 12.6）。按照这种方式，所有处理器都被同步而且比较总涉及正确的处理器。如果在算法执行的某一步，一个处理器没有相应的邻居（如在算法的第 2 步，第一个处理器），则令该处理器处于空闲状态。一般称该算法为奇–偶互换排序（odd-even transposition sort）。算法见图 12.7，使用该算法对数值进行排序的实例见图 12.8。注意，在例子中，算法在第 6 步之后就完成了。然而，在一个网络中要检查算法是否可以提前结束是很困难的。因此，在许多情况下最好是让算法运行到最坏情况下所需要的步数。

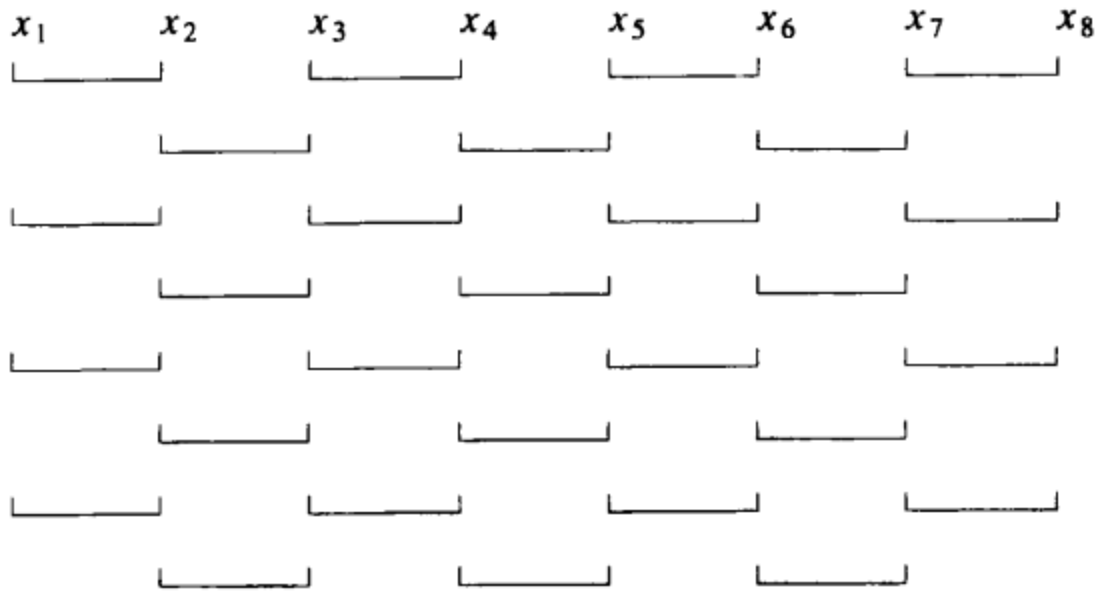


图 12.6　奇–偶互换排序

算法 ***Sorting_on_an_Array*** ***(x, n)***
输入：（下标范围为 1 到 n 的数组，处理器 P_i 持有 x_i）
输出：（排好序的数组，其中第 i 个最小的元素存放在 P_i 中）

```
begin
    do in parallel ⌈n/2⌉ times
        P_{2i-1} and P_{2i} compare their elements and exchange them
            if necessary ; {对所有 i，1≤2i≤n}
        P_{2i} and P_{2i+1} compare their elements and exchange them
            if necessary ; {对所有 i，1≤2i<n}
            {如果 n 是奇数，则该步仅执行⌊n/2⌋次}
end
```

图 12.7　算法 *Sorting_on_an_Array*

算法 *Sorting_on_an_Array* 似乎看起来既自然又清晰，但它的证明并不简单。要考虑的一件事情是，一个元素可能会远离它的最终目的地。例如，在图 12.8 中，数 5 在开始右移之前左移了两次，而数 3 在右移之前被移到最左的位置并呆在那里。基于不同处理器间动作的相互依赖，证明并行算法的正确性会比较困难。在并行计算场景下，一个处理器的行为可能会影响所有其他的处理器，不能只考虑一个处理器并证明它的动作是正确的，而必须将所有的处理器一并考虑进去。

□ **定理 12.2**

当算法 Sorting_on_an_Array 停止时，数被排好序了。

证明：对处理器的数目（或元素的个数）进行归纳。如果只有两个处理器，则一次比较就完成了排序。假定算法对 n 个处理器是正确的，现在考虑 n+1 个处理器的情况。让我们将注意力集

中在最大的元素并假设它是 x_m（如图 12.8 中的数 5）。在第 1 步，依赖于 m 是奇数还是偶数，x_m 和 x_{m-1} 或 x_{m+1} 进行比较。如果 m 是偶数，由于 x_m 大于 x_{m-1}，因此不会有交换发生。但这与一开始 x_m 在 P_{m-1} 中一样（发生了一次交换）。因此，不失一般性，可假定 m 是奇数。在这种情况下，x_m 和 x_{m+1} 进行比较，交换，并逐步（呈对角线）右移（因为它比其他数都大）直至到达 x_{n+1} 为止。这是该元素的正确位置，因此排序工作对最大的元素来说是正确的。

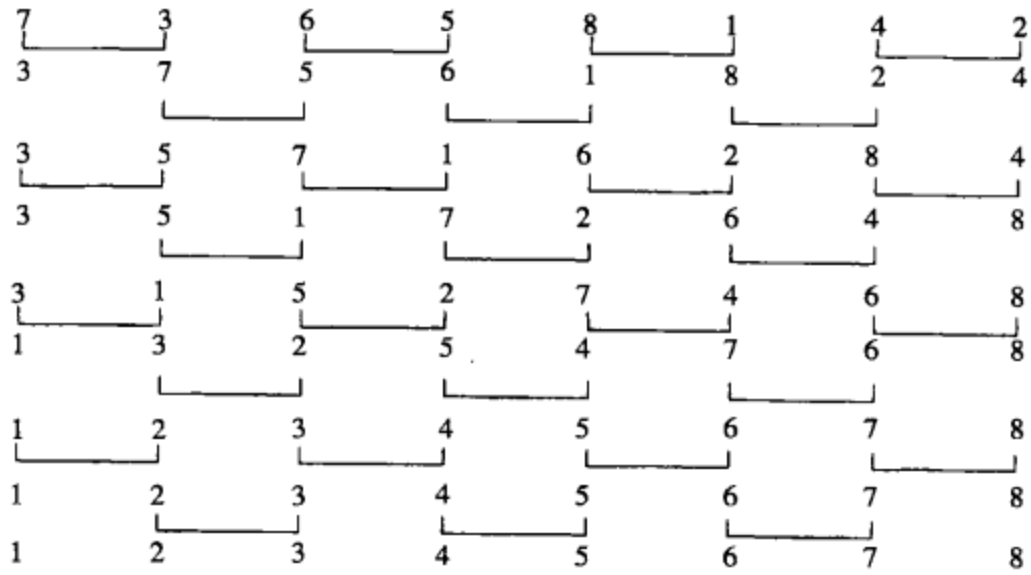


图 12.8　奇–偶互换排序的例子

接着必须证明对剩下的元素所进行的排序也是正确的。剩下的元素有 n 个，因此可使用归纳法。为此，我们必须将在一个大小为 $n+1$ 的阵列上的 n 个处理器的运行映射为 $n-1$ 个处理器在一个大小为 n 的阵列上的可能运行。该映射可如下定义。考虑最大元素移动所形成的对角线（见图 12.9），忽略涉及最大元素的比较（位于对角线的那些比较）。可以将其他的比较分成两组，一组位于对角线的上方，一组位于对角线的下方。我们将对角线上的三角形往下移一步。换句话说，对于对角线上的比较，第 i 步现在被称为第$(i+1)$步。例如，考虑图 12.8 中算法第 1 步所实施的 1 和 8 以及 4 和 2 的比较。第一个比较位于对角线，因此被忽略；第二个比较在对角线之上，我们将它考虑作为第 2 步执行的比较（而不是第 1 步）。因此第 2 步包括 7 和 5、6 和 1 以及 4 和 2 的比较。这是一个仅涉及 n 个元素的有效的偶数步运算。可简单地忽略涉及对角线左边元素的第 1 步运算（即第 1 行）以及所有涉及最大元素的比较（最后一列），剩余的恰好是算法应用于 n 个元素所产生的比较序列。由归纳假设，n 个元素的排序是正确的，因此 $n+1$ 个元素的排序也是正确的，且只需要再进行一步即可。 □

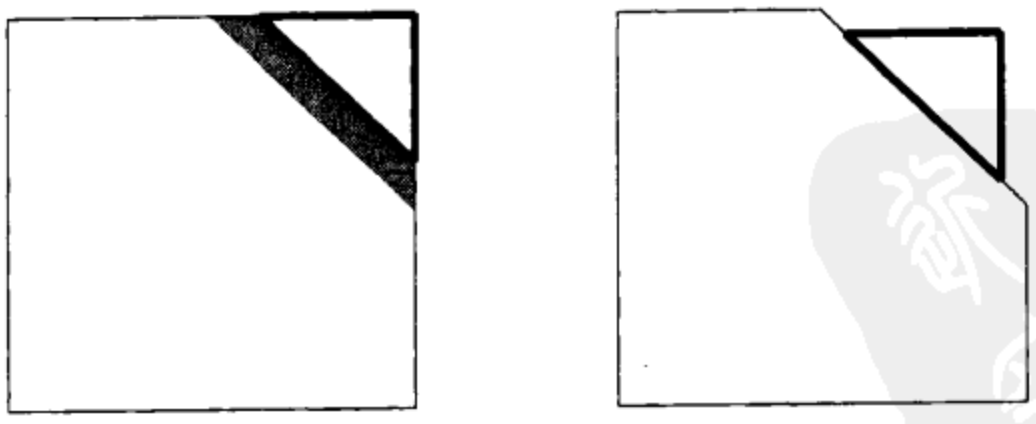


图 12.9　证明定理 12.2 的归纳步骤

到目前为止，我们已经讨论了每个处理器持有一个输入数据的情况。现假定每个处理器持有 k 个数据并且只有两个处理器，目标是对数据重新分配使得 P_1 持有 k 个较小的数据，而 P_2 持有 k 个较大的数据。显然在最坏情况下，所有数据都必须移动，因此工作不会比 $2k$ 个数据移动少。一

种排序方式是重复如下步骤直至排序完成：P_1 将它的最大元素送给 P_2，而 P_2 将它的最小元素送往 P_1。但当 P_1 中的最大元素小于 P_2 中的最小元素时，算法停止。这个过程又称合并–分离（merge-split）。如果将该操作作为奇–偶互换排序的基本步骤，那么我们就可以把算法扩充为一个处理器持有多个数据的排序算法，此时可用合并–分离操作代替毗邻元素的比较以及可能的交换。

尽管这里讨论的排序算法对于阵列是最优的，但算法的效率不高。算法使用了 n 个处理器，每个处理器运行 n 步，因此总运行步数是 n^2 步。高效的算法应该能交换远离的两个数据，由于阵列无法支持这种数据交换，所以奇–偶互换排序算法的低效并不出人意料。下一节将讨论专为高效排序设计的互连网络。

12.4.2　排序网络

高效串行排序算法的设计仅需要注意总的比较次数，而在设计并行算法时，我们还必须使计算尽可能独立。回顾 6.4.3 节的归并排序算法，两个递归调用是完全独立的，因此可以并行执行。然而，算法的合并部分是以完全串行的方式运行的。最终序列中的第 i 个元素只有在它前面的 i–1 个元素被放置好了才能被定位。如果能对合并部分并行化，就能对归并排序进行并行计算。

下面介绍的是由 Batcher[1968]设计的一个不同的归并算法，该算法使用的是分治法。为了方便，假定 n 是 2 的幂。令 $a_1, a_2, \cdots, a_n$ 和 $b_1, b_2, \cdots, b_n$ 是两个待归并的序列，归并的结果是序列 $x_1, x_2, \cdots, x_{2n}$（例如 $x_1 = min(a_1, b_1)$）。希望通过并行合并这些分离序列使最终的合并能变得容易。可以先将两个序列分别分为两部分，即下标为奇数的元素和下标为偶数的元素。每部分分别与另一个序列的对应部分归并。最后再进行合并。令 $o_1, o_2, \cdots, o_n$ 是 $a_1, a_3, \cdots, a_{n-1}$ 和 $b_1, b_3, \cdots, b_{n-1}$ 的归并结果，$e_1, e_2, \cdots, e_n$ 是 $a_2, a_4, \cdots, a_n$ 和 $b_2, b_4, \cdots, b_n$ 的归并结果，显然有 $x_1 = o_1$，$x_{2n} = e_n$。剩下的归并也很方便，可由下面的定理来说明（见图 12.10）。

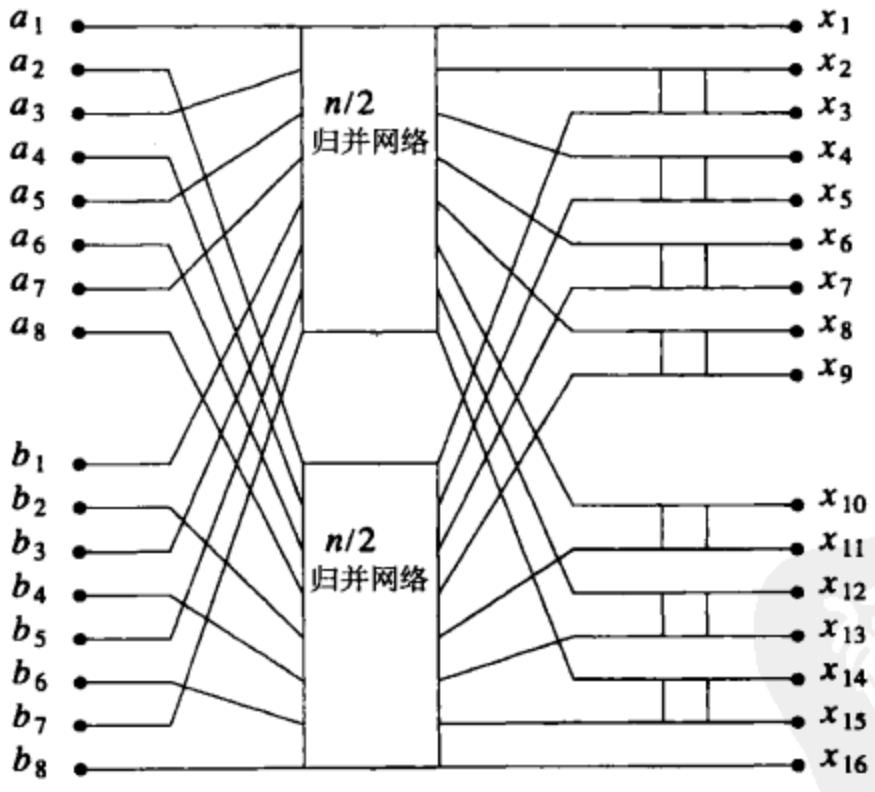


图 12.10　奇–偶归并线路图

□ **定理 12.3**

按照上面的记法，对于所有 $1 \leqslant i \leqslant n-1$，有 $x_{2i} = min(o_{i+1}, e_i)$，$x_{2i+1} = max(o_{i+1}, e_i)$。

证明： 考虑 e_i。由于 e_i 是归并后的偶数序列的第 i 个元素，它一定大于或等于两个偶数序列

中至少 i 个元素。而对于每个偶数元素，则可加入小于 e_i 的一个奇数元素（因为考虑的是两个已经排好序的序列的归并）。因此，e_i 大于或等于两个序列中至少 $2i$ 个元素。换句话说，$e_i \geqslant x_{2i}$。同样，o_{i+1} 大于或等于两个奇数序列中至少 $i+1$ 个元素，因此大于或等于至少 $2i$ 个元素（必须将下标减去 2，因为两个奇数序列的第一个元素前不能插入元素）。因此 $o_{i+1} \geqslant x_{2i}$。现在，由鸽洞原理，e_i 和 o_{i+1} 必须（以某种顺序）等于 x_{2i} 和 x_{2i+1}。因此 e_1 和 o_2 必须等于 x_2 和 x_3，e_2 和 o_3 必须等于 x_4 和 x_5，以次类推。 □

定理 12.3 的重要性在于说明了最后的归并可并行进行。剩下的可由归纳法证明。并行算法可直接依照定理设计。图 12.10 说明了递归归并的构造，图 12.11 则说明完整的排序，该排序又称奇–偶归并排序。图 12.11 左边的（标记为“$n/2$ 排序”）的小方框是整个排序的递归构造。右边的编号是排序之后数的序号。

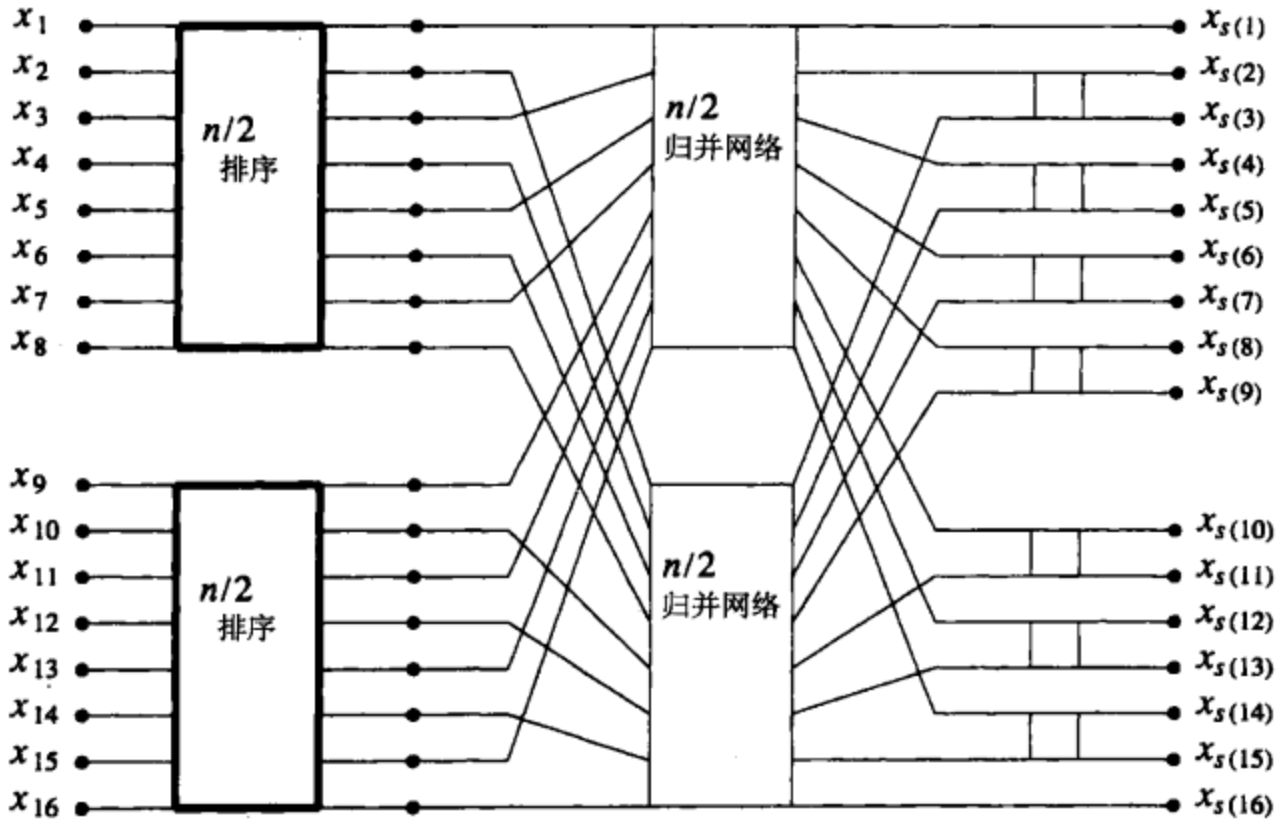


图 12.11 奇–偶归并排序线路图

复杂度 对于归并，总的计算步数 $T_M(n)$ 满足递推关系 $T_M(2n) = 2T_M(n)+n-1$，$T_M(1) = 1$。因此总的比较次数是 $O(n\log n)$，而串行算法仅需要 $O(n)$ 步。递归的深度，对应于并行计算时间来说是 $O(\log n)$。奇–偶归并排序的计算步数 $T_S(n)$ 满足递推关系 $T_S(2n) = 2T_S(n)+O(n\log n)$，$T_S(2) = 1$，即 $T_S(n) = O(n\log^2 n)$。排序电路的每一列包含 n 个处理器，深度为 $O(\log^2 n)$，因此电路中有 $O(n\log^2 n)$ 个处理器（注意，尽管相同处理器可用于所有的列，但它们几乎必须完全互连）。电路中惟一类型的计算就是比较，因此，所需要的处理器是有两个输入和两个输出的比较器。

12.4.3 在树中查找第 k 个最小元素

现在假定互连网络是一棵叶子数 $n = 2^h-1$ 的完全二叉树。有 2^h-1 个处理器，每个处理器对应树中的一片叶子。输入是序列 $x_1, x_2, \cdots, x_n$，叶子 i 有输入 x_i。树形机器主要是为图像处理应用而设计的，其中叶子对应于输入（例如，图像中的像素），对输入数据的处理算法呈层次结构（参见[Uhr 1987]）。在本节中，我们考虑在树中查找第 k 个最小元素的问题，使用的技术是流水线（我们将在 12.5 节详述流水线技术）。

首先，回忆一下在 6.5 节描述的查找第 k 个最小元素的串行算法。为了简单，假定所有元素都是不同的。每一步，算法都随机选择一个元素 x 作为支点（pivot），通过比较 x 与其他元素的大

小来计算 x 的秩，按照秩是小于或大于 k，删除小于或大于 x 的元素。但在支点元素的秩等于 k 时算法停止。期望的迭代次数是 $O(\log n)$，比较次数为 $O(n)$。在算法的每次迭代过程中有三个主要阶段：（1）随机选择一个元素；（2）计算它的秩；（3）删除元素。我们首先描述每个阶段的并行实现，然后考虑进一步的改进。

选择一个随机元素的过程可以通过在树上安排一场锦标赛来实现。每个叶子送给它的双亲一个数，该数与它的兄弟送上去的数通过掷硬币来决定谁将胜出。胜出的数再往上传递，同样的过程在树中持续进行，直到树根选出最后的胜者为止。然后胜出的数往下广播，这样所有叶子都可以将其与自己持有的树进行比较。如果支点是已知的，则所有的叶子可以在一步之内完成比较操作，然后把 1（如果持有的数小于或等于支点）或 0（如果持有的数大于支点）送给它们的双亲。显然，支点的秩就是 1 的数目，在树中统计该数目是很简单的。接着树根将其往下广播，每个叶子决定其持有的数是否要被删除。总之，在每次迭代过程中，有 4 次通信：（1）从叶子往根选出支点；（2）从根往叶子广播支点；（3）从叶子往根计算支点的秩；（4）从根往叶子广播支点的秩。

问题是，当一些元素被删除后，锦标赛就不是公平的了。在极端的情况下，树的一边，所有元素除了一个外都被删除。剩下的那个元素将直接送至树根而无须比较，因此它被选中的概率是 1/2，而位于树的另一边的元素被选中的概率则小得多。欲使每一个元素被选择的概率一样，可按如下方式操作。持有被删除元素的处理器向上送 nil 值，任何与 nil 相比的元素都将胜出，每个参与锦标赛的元素带有一个计数器，初始值为 1，该计数器记录了在锦标赛中与该元素对垒的真正的数的个数（即在一棵子树中未被删除的元素的个数）。当一个元素在树中某个节点赢取了一场比赛，该元素就向上传递，同时失去比赛的元素的计数器的值就被加至该元素的计数器上。现在每场比赛掷投硬币就带有倾向性，它与参加比赛的元素的计算器的值相关。例如，如果 x 赢取了第一场比赛（假设和 y），而 z 轮空，则 x 的计算器值为 2，而 z 的计算器值为 1。如果现在 x 与 z 比赛，则倾向度为 2:1。即 z 有 1/3 的可能性赢取该场比赛，而 x 和 y 有 $1/2\times 2/3 = 1/3$ 的可能性赢取该场比赛。该过程使得最后的选择保证了参选元素有同样被选择的机会。

复杂度　每个阶段所涉及的（并行）计算的步数等于 4 倍的树的高度，由于算法删除元素的方式同串行算法，因此期望的阶段数仍然是 $O(\log n)$，期望的执行时间是 $O(\log^2 n)$。

一个改进的算法

树根是计算的一个主要瓶颈，大多数消息传递都经过树根，但树根只有两个连接，而所有的叶子距离它的距离为 h–1。如果不能改善连接，那么至少应该使根节点尽可能忙。在上面所描述的算法中，树根和叶子活跃一次之后大约有 $2h$ 的时间是空闲的。在上一次迭代完成之前就激活新的迭代可使所有处理器在所有时候都是繁忙的，由此可改善算法，此时所有的迭代都将以流水线的方式在树中上上下下。使用流水线可改善运行时间的理由是：算法用了 $2h$–2 步选择支点（h–1 步到达根，h–1 步广播结果。如果在第 2 步就开始一场新的锦标赛并与第一场锦标赛并行进行（但落后一步），那么我们可以在 $2h$–1 步选择两个支点，而在 $3h$–2 = $O(\log n)$ 步内选取 h 个支点，所有这些支点都可用于删除元素。因此代之以每次使用一个支点删除大约一半的元素，使用 h 个支点可以将搜索区间缩小为原来的 $1/(h+1)$，而后者并未花费太多的额外时间。其他阶段也可交插执行。每步叶子都可以开始一轮新的锦标赛（直至第 k 个最小元素被找到为止）而锦标赛推动了剩余的计算。

□ **例 12.1**

图 12.12 是过程的一个例子（进行方式为从左到右，从上到下）。元素是 1 到 8 的数，算法寻找第 4 个最小的数（在这种情况下，每个元素的秩就等于该元素的值，因此无须同时显示元素的秩和值）。图中的每一步都给出了所有节点的内容。节点内部的数来自该节点下方的节点，而节

点外部的数据是从上面广播下来的数据。第 1 个被选择的支点是 3，见图 12.12（d），第二个被选择的支点是 5，见图 12.12（e），然后是 4、3、1 和 4。在图 12.12（g）中，第一个支点（即 3）到达了叶子，从那时起，开始第二个阶段，也就是计算秩。元素 3 的秩是 3 的事实在图 12.12（j）被发现，而第 4 个最小元素被树根在图 12.12（l）发现。一旦该元素被广播到节点（图中未显示），算法就停止了。在这种情况下，无须继续进行另一轮迭代（或删去元素）；然而，一般说来，该过程应被执行若干次，同时持续删除元素直至第 k 个最小元素被发现为止。 □

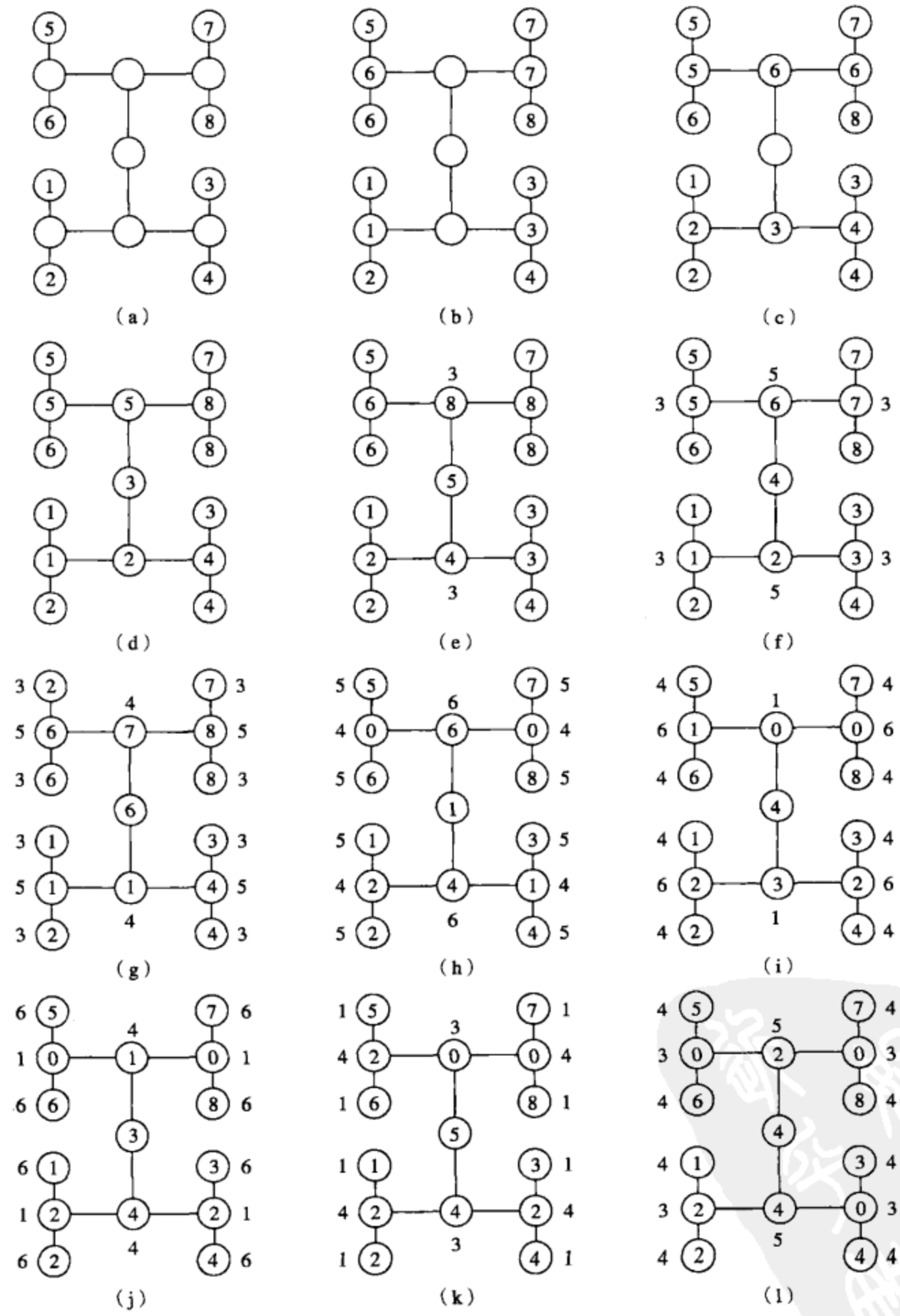


图 12.12　发现第 k 个最小元素流水线算法的例子

复杂度　给定一个支点，通常算法需要执行$O(\log n)$以删除元素。由于可以在同样的时间内产生$O(\log n)$个支点，因此总的来说节省了 $O(\log\log n)$倍。期望的执行时间减少为 $O(\frac{\log^2 n}{\log\log n})$。该事实的证明、算法的细节、模拟结构以及其他一些小的改进，可以在 Greenberg and Manber[1987]中找到。

12.4.4　网孔上的矩阵乘法

现在考虑互连是二维 $n\times n$ 网孔（网格）的情况。处理器 $P[i,j]$位于网孔中的第 i 行第 j 列，与处理器 $P[i-1,j]$、$P[i,j+1]$、$P[i+1,j]$及 $P[i,j-1]$互连。假定网孔的边界是环绕的，例如，$P[0,0]$除了与 $P[0,1]$、$P[1,0]$互连外还与 $P[0,n-1]$及 $P[n-1,0]$互连（见图 12.13）。换句话说，索引上的所有加法和减法都是模 n 的。在这种假设下的算法更加对称和优雅，与没有环绕的算法有着相同的运行时间（在一个常数范围内）。

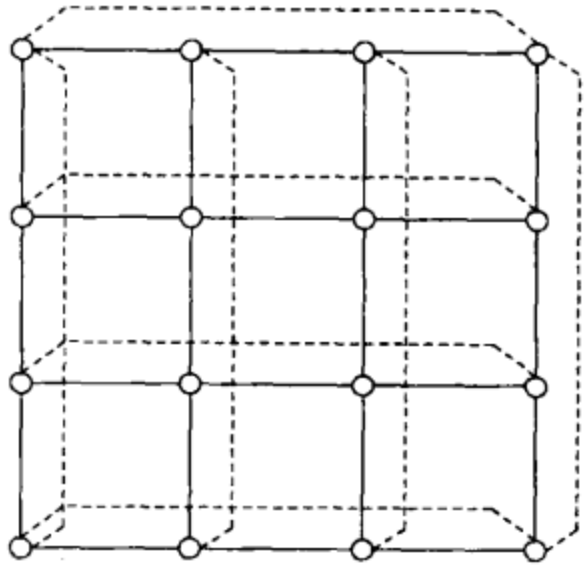


图 12.13　一个二维环绕网孔

> **问题**　给定两个 $n\times n$ 的矩阵 A 和 B，初始时，$P[i,j]$持有 $A[i,j]$和 $B[i,j]$，计算 $C=A\cdot B$，使 $P[i,j]$持有 $C[i,j]$。

使用常规的矩阵乘法运算。问题是移动数据使得正确的数据在正确的时间位于正确的位置上。考虑$C[0,0]=\sum_{k=0}^{n-1}A[0,k]\cdot B[k,0]$，它是 A 的第 1 行元素与 B 的第 1 列元素的内积。希望 $C[0,0]$由 $P[0,0]$计算（最后它也必须持有该值）。这可通过使 A 的第 1 行元素每步左移一个位置，而 B 的第 1 列元素每步上移一个位置实现。在第 1 步，$P[0,0]$持有 $A[0,0]$和 $B[0,0]$，计算它们的乘积，第 2 步，$P[0,0]$得到 $A[0,1]$（从右）和 $B[1,0]$（从下），将它们的乘积加上原来的部分结果，依次类推。在 n 步以后，$C[0,0]$的值就被计算出来了。

问题是需要所有的处理器都做同样的事，而且还必须共享数据。我们必须对数据的移动进行安排，使得不仅 $P[0,0]$能得到它所需要的数据，其他处理器也能得到所需要的数据。技巧是重新安排数据的移动。使得在算法执行过程中，所有处理器总有需要计算乘积的两个数。关键是数据的初始分派。重新安排数据使得 $P[i,j]$持有 $A[i,i+j]$和 $B[i+j,j]$（注意这里加法是模 n 的）。如果可行，每步随着行和列的移动，$P[i,j]$将持有 $A[i,i+j+k]$和 $B[i+j+k,j]$，$1\leqslant k\leqslant n$，这正是 $P[i,j]$所需要的。初始数据的重新分配可通过对所有行和列，第 i 行的数据左移 i 个位置，第 j 列的数据上移 j 个位置完成，算法见图 12.14。图 12.15 说明了初始数据的移动，左边是数据的初始位置，右边是移动后数据的位置。

复杂度　初始行的移动需要 $n/2$ 并行步（当移动的距离大于 $n/2$ 时，可以改为右移），对于列

的移动也一样，不涉及计算。然后，每个处理器需要执行 n 步，包括计算和移动的操作，所有这些都可以并行进行。总的运行时间是 $O(n)$。如果与通常的时间复杂度为 $O(n^3)$的算法相比，该算法的效率为 $O(1)$。如果与渐近较快的算法（如 Strassen 算法）相比，这里的算法效率就是渐近较小的。

```
算法 Matrix_Multiplication (A, B)
输入：A 和 B（n×n 的矩阵）
输出：C（A 和 B 的乘积）
begin
    for all rows do in parallel
        shift row i of A to the left i steps
        {即执行 A[i, j]:= A[i, (j+1) mod n]i 次}
    for all columns do in parallel
        shift column j of B upwards j steps
        {即执行 B[i, j]:= B[(i+1) mod n, j] j 次}
    {现在数据位于合适的开始位置}
    for all i and j do in parallel
        C [i, j ] :=A [i, j ] · B [i, j ] ;
    for k := 1 to n − 1 do
        for all i and j do in parallel
            A [i, j ] :=A [i, (j + 1) mod n ] ;
            B [i, j ] :=B [(i + 1) mod n, j ] ;
            C [i, j ] :=C [i, j ] + A [i, j ] · B [i, j ]
end
```

图 12.14　算法 *Matrix_Multiplication*

a_{11} b_{11}	a_{12} b_{12}	a_{13} b_{13}	a_{14} b_{14}
a_{21} b_{21}	a_{22} b_{22}	a_{23} b_{23}	a_{24} b_{24}
a_{31} b_{31}	a_{32} b_{32}	a_{33} b_{33}	a_{34} b_{34}
a_{41} b_{41}	a_{42} b_{42}	a_{43} b_{43}	a_{44} b_{44}

a_{12} b_{21}	a_{13} b_{32}	a_{14} b_{43}	a_{11} b_{14}
a_{23} b_{31}	a_{24} b_{42}	a_{21} b_{13}	a_{22} b_{24}
a_{34} b_{41}	a_{31} b_{12}	a_{32} b_{23}	a_{33} b_{34}
a_{41} b_{11}	a_{42} b_{22}	a_{43} b_{33}	a_{44} b_{44}

图 12.15　初始数据位置和移动后数据的位置

12.4.5　超立方体中的路由

迄今为止我们所看到的例子充分说明了将一个简单的算法调整成在一个互连网络环境下的算法有多么困难。如果算法严重依赖于体系结构，则程序设计会变得非常困难。一个在互连网络上设计算法的方法是定义一些强有力的基本操作，然后使用这些基本操作设计算法。将使用这些基本操作的所有算法移植到另一种互连结构仅须对这些基本操作给出实现即可。基于这种方法进行并行算法设计的困难在于定义那些基本操作。而另一种方法是设计一个使用一种网络来模拟另一种网络的算法。这种技术实现了两个互连网络间算法的简易转换，但这种转换可能不会产生出高效的算法。本节将简短地讨论一种通用的路由方案，使用该方案在设计互连网络上的算法时，

就像在共享存储器上进行设计一样。

假定网络互连结构是超立方体的（对其他拓扑结构，方案也是类似的）。在一个 EREW 共享存储器算法中，不同的处理器可以在一步内访问任意的变量。假定存储器 P_i 负责变量 x_i。由于 EREW 算法没有任何读和写的冲突，算法中的一步将包括处理器对不同变量的访问。另一种看待算法中的一步的方式是将其视为一个排列σ，处理器 P_i 访问变量 $x_{\sigma(i)}$（并不是所有的处理器在所有的时间内对变量进行访问，但在最坏情况下，可如此假定）。我们将注意力集中于 EREW 算法其中的任意一步，并试图用超立方体的若干步来进行模拟。现在所面临的就是路由问题。假定超立方体上的每个处理器 P_i 想发送一个消息给处理器 $P_{\sigma(i)}$（其持有变量 $x_{\sigma(i)}$）。所有消息都在同一个时刻发出，目标是在超立方体中快速传递它们。假定超立法体的每条边每次只能传递一条消息。因此，现在的问题不仅仅是寻找源和目的地的最短路由，还必须避免在同一个时间尝试使用同一条边带来的冲突。如果有两条消息尝试在同一时间使用同一条边，则其中之一必须被存在缓冲区之中。我们也需要使缓冲区最小。

最好的路由显然与特定排列相关。但通常想通过分析特定的排列来发现最佳路由并不可行，因为排列信息是分布在处理器间的。因此我们想寻找一种方案，平均来说，它可以工作得很好。下面的路由算法是 Valiant[1982]提出的，关键的思想是使用随机选择。路由方案包括两个阶段，在第一个阶段，每个处理器 P_i 将消息发给一个随机的处理器，该处理器是（独立于目的处理器）在所有处理器间均衡选择的（下面将描述如何随机选择处理器以及如何将消息路由给它）。在第二个阶段，消息从随机选择的那个处理器按照最佳路由送至目的处理器。

所有的消息都是按照同一种方式发送的，因此我们可考虑其中之一，如从 i 到 j 的消息。令 i 的二进制表示为 $b_1b_2\cdots b_d$（在开始处可能带有若干用于长度填补的 0），令 j 的二进制表示为 $c_1c_2\cdots c_d$。在第一个阶段，我们需要找到一个随机处理器 r，r 可通过下面的方式找到：逐位考虑 i 的位串表示，每位按可能性 1/2 随机决定是否路由到它的邻居节点。例如，考虑图 12.16，令 $i=000$，$j=110$。首先考虑是否将消息送至 100，假定答案是否定的，然后考虑第 2 位并决定是否修改，换句话说，我们决定是否将消息送至 010。最后考虑第 3 位。在图 12.16 中，随机处理器是 011，当然消息的路由（以粗线标出）将经过它。如果我们决定不发送消息，那么直接就可以进行下一次选择而不是等到第 2 轮；这里假定局部的计算要比消息传递快得多。当对最后一位做出选择时，随机路由就完成了，每个选择都可局部做出。

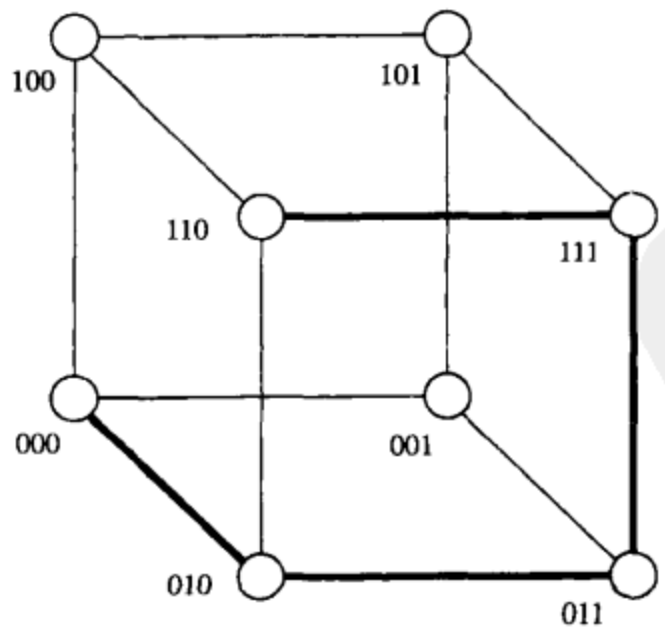


图 12.16　三维立方体中的随机路由例子

由于所有的处理器都在同一时间内发送消息，在同样一条边上一个处理器可能有多于一条的消息等待发送，所以在这种情况下，消息将被（以随机次序）放入队列之中，并当边是可用时再进行发送。

不难看出每个数 k（在一个合适的范围内）有着相同的被选为随机中转的概率。从随机选择的处理器 r 到目的地 j 的路由是确定的。假定 r 和 j 有 t 位不同，它们的下标是 $k_1<k_2<\cdots<k_t$，消息的传递是改变位 k_1，然后是 k_2，如此等等。以图 12.16 为例，消息先从 011 送到 111（因为目标 110 在第 1 位有一个 1），然后送到 110（第 2 位没有改变）。

这个路由方案很容易实现。路由并不一定是最短的路径（在上例中，路由包括 4 条边，而最短路径只需 2 条边）。然而，每条路由的长度并不太长，基本上是 $2d$）。这些路由的主要性质是它们冲突的可能性很小，如此排列期望在 $O(d)$步内完成。而对其他拓扑结构也曾有人提出过类似的路由方案。基于该方案，已经开发出自动将共享存储器上的算法映射为互连网络上算法的高效程序。

12.5 脉动计算

脉动（systolic）结构可以看成装配线、处理器（通常被称为处理单元）以一种很规整的方式排列（通常，但并不总是一维或二维阵列），而数据则以一种和谐的方式经过它们。每个处理器都对它在上一个脉动受到的数据执行着很简单的操作，然后将结果送到下一个站点。每个处理器（如果有的话）包含极其有限的存储单元。大部分的输入是每次“推给”一个变量，而不是被同时装入某段内存区域。脉动阵列结构的特点是高效，不管是从硬件来衡量（既专用又简单），还是从速度方面考虑（内存访问周期是最小的）。如同装配线，关键是避免在工作中寻取工具、材料等带来的开销，在装配机器时，完成一个操作所需要的所有东西都已到位。最大的缺点是这种方案缺乏灵活性，脉动结构仅对一些算法有效，下面将讨论三个脉动算法的例子。

12.5.1 矩阵与向量相乘

先从一个直接简单的例子开始，然后再讨论复杂的算法。

> **问题** 计算 $m\times n$ 的矩阵 A 和一个大小为 n 的列向量 b 的乘积 $x=Ab$。

假定有 n 个处理器，P_i 负责涉及 b_i 的部分和的加法操作。图 12.17 图示了 $n=m=4$ 时数据的移动以及处理器的行为。假定 b 被存放在（或被规整注入到）相应的处理器中。当存放结果的变量 x_i（初始值为 0）从左到右经过处理器时结果被脉动阵列累加起来。在第 1 步，x_1（值为 0）和 a_{11} 进入 P_1，而其他的则靠近脉动阵列一步。P_1 计算 $x_1+a_{11}\cdot b_1$，并将结果移向右边；第 2 步，P_2 从左边收到值 $x_1=a_{11}\cdot b_1$，从上方收到值 a_{12}，计算 $x_1+a_{12}\cdot b_2$，并将结果移向右边。每一步，P_i 从左边接收一个部分结果，等于 $\sum_{k=1}^{i-1}a_{jk}b_k$，从上方接收矩阵的一个适当元素，并从局部存储器（如图所示）或下方接收一个合适的数据，然后计算 $x+a_{ji}b_i$，并将结果右传。当 x_i 离开阵列时，它就有了正确的值。所有的计算在 $m+n$ 步内完成。

设计脉动算法的主要问题在于数据的移动。每个数据都必须在正确的时间位于正确的位置。本例所用的惟一技巧是引入延迟使矩阵第 i 列的元素在第 i 步到达处理器 P_i。由于 A 的每个元素仅被使用一次，因此本例十分简单，当相同的值可被使用多次的时候（也就是在一般的情况下），设计数据的移动就十分复杂了，下例将予以说明。

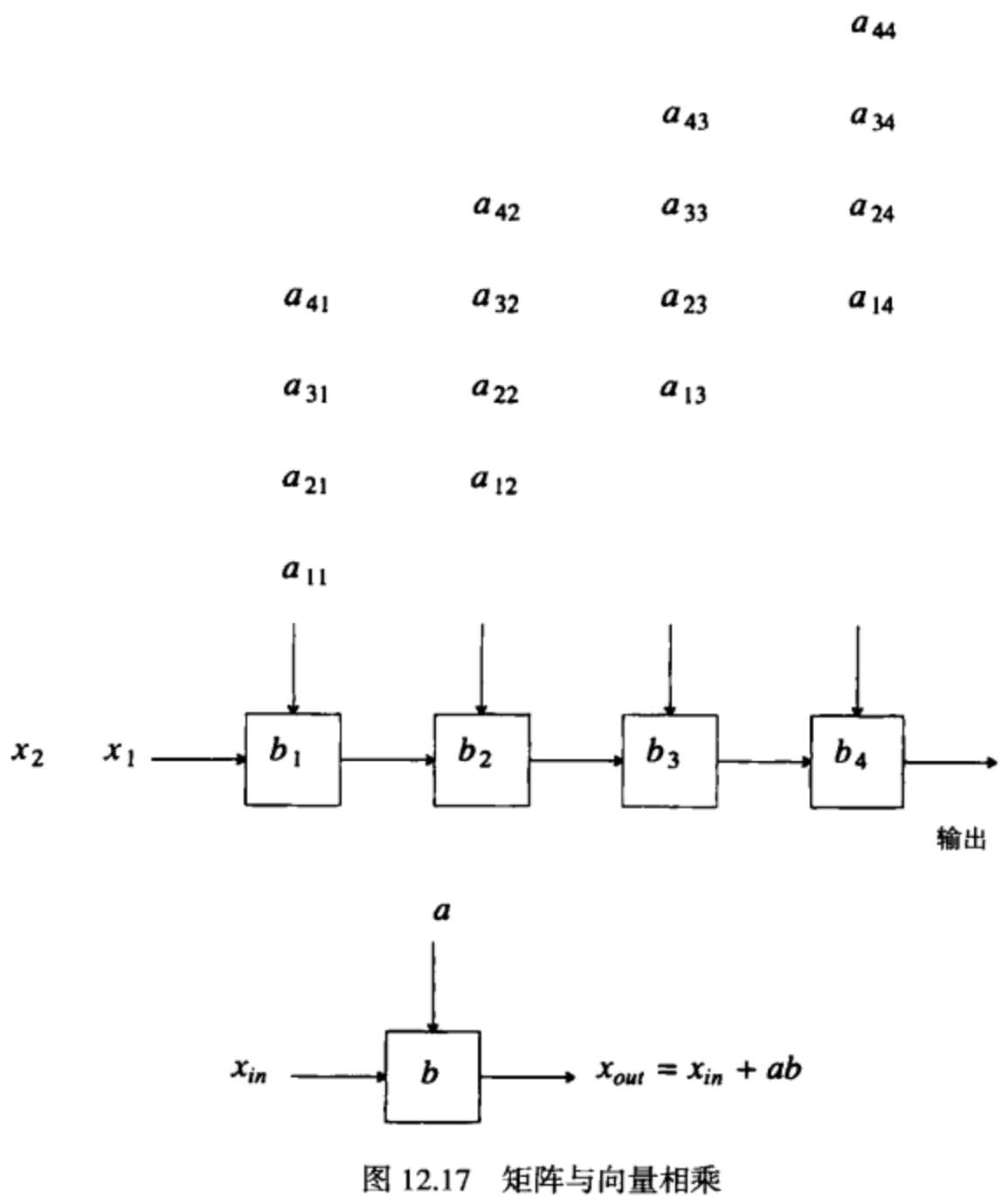


图 12.17　矩阵与向量相乘

12.5.2　卷积问题

令 $x = x_1, x_2, \cdots, x_n$ 和 $w = w_1, w_2, \cdots, w_k$ 是两个实数序列，其中 $k < n$。

> **问题**　计算 $y_1, y_2, \cdots, y_{n+1-k}$，其中 $y_i = w_1 x_i + w_2 x_{i+1} + \cdots + w_k x_{i+k-1}$。

向量 y 被称为 x 和 w 的卷积。可以将卷积问题转化成如下矩阵与向量的乘积:

$$\begin{bmatrix} x_1 & x_2 & x_3 & \cdot\cdot & x_k \\ x_2 & x_3 & x_4 & \cdot\cdot & x_{k+1} \\ x_3 & x_4 & x_5 & \cdot\cdot & x_{k+2} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ x_{n+1-k} & x_{n+2-k} & x_{n+3-k} & \cdot\cdot & x_n \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \cdot \\ w_k \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdot \\ \cdot \\ \cdot \\ y_{n+1-k} \end{bmatrix} \qquad (12.1)$$

简单地将图 12.17 中的矩阵替换为式（12.1）中的矩阵，就可以得到问题的脉动算法，见图 12.18。注意阵列中的每个处理器都在同一时间需要每一个 x_i（除了最初的 $k-1$ 个 x_i，它们不是每个地方都需要）。因此需要一个广播线。下面将给出一个不需要广播的卷积问题的求解方法。

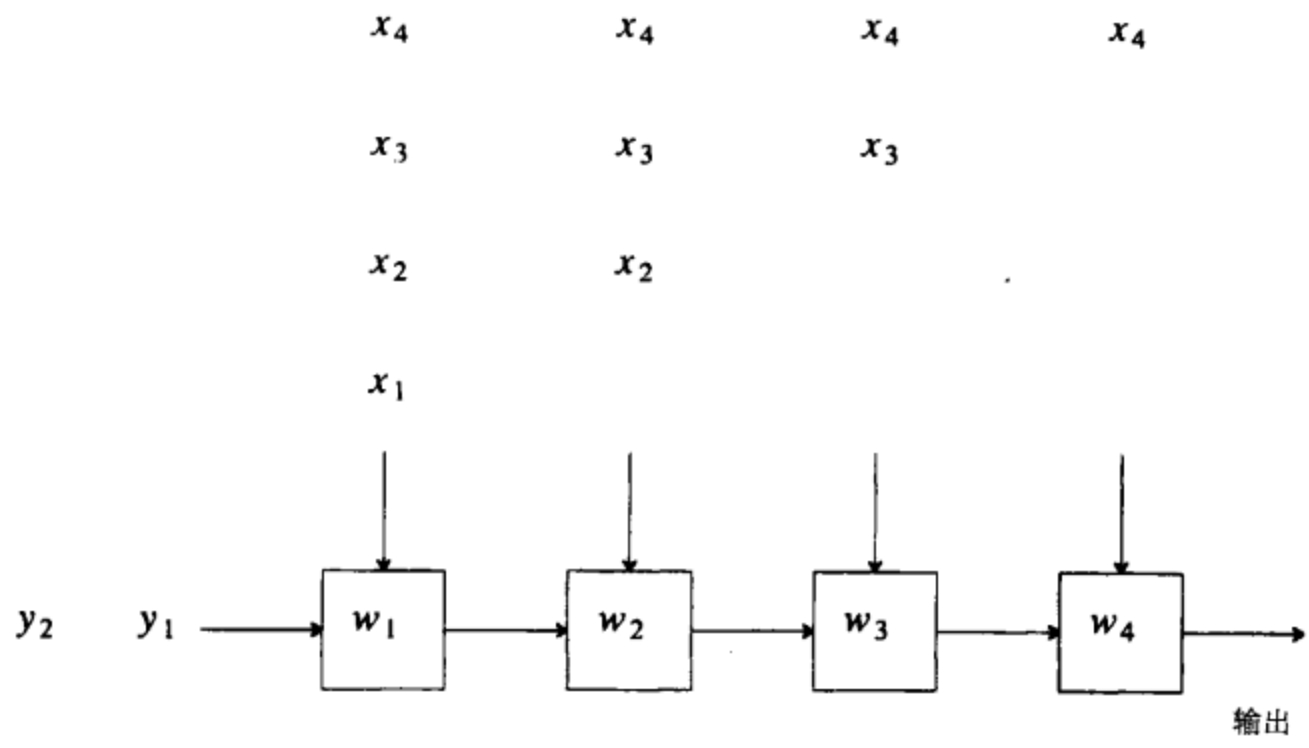


图 12.18 使用广播的卷积问题求解

图 12.18 中的处理器接收多个输入，但仅送出一个结果。现在使用能从两个方向接收输入并从两个方位送出结果的处理单元。基本思想是使向量 x 的数据从左到右移动，向量 w 的数据从右到左移动，而处理器经过累加得到结果 y。此时需要设计合适的移动使 w 的值与 x 的值能正确相遇。在两个不同的方向移动向量的后果是它们将以双倍的速度靠拢，结果是，x 的元素将错过 w 一半的元素，反之亦然。解决方法是让向量以半速移动。左边的输入诸如“x_1, nothing, x_2, nothing”，w 也一样，该解决方案的图示为图 12.19（其中黑点对应于空槽）。

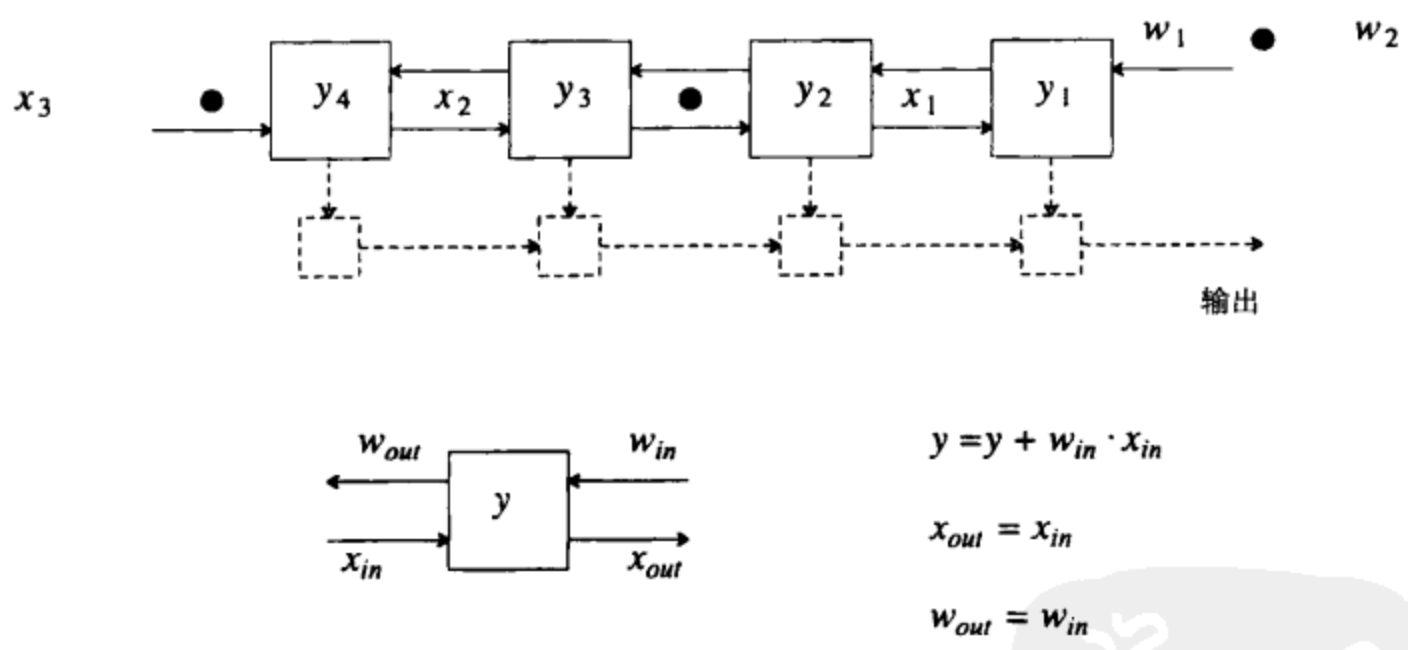


图 12.19 使用双向阵列的卷积问题求解

读者可自行验证 P_i 确实计算了 y_i。当 w_k 离开 P_i 时，P_i 正好计算出了 y_i 的值，它可以通过图 12.19 下方所示的数据通道移去。使数据以半速移动的主要缺点是需要双倍的计算时间。

12.5.3 序列的比较

令 $A = a_1a_2\cdots a_n$ 和 $B = b_1b_2\cdots b_m$ 是两个字符串。目标是找出 A 和 B 的最小编辑距离（参见 6.8 节和 10.2 节）。6.8 节给出了解决该问题的一个动态规划算法。我们先讨论对其进行并行化的一

般方法，然后给出脉动求解方案。

算法逐渐填写表格，最后的答案出现在表格的右下角。表格第 ij 个元素的值依赖于包围它的 3 个元素$(i-1)j$、$(i-1)(j-1)$和 $i(j-1)$的值。首先假定有 nm 个处理器并按照表的结构进行排列（见图 12.20）。每个处理器 P_{ij}从相邻的 3 个处理器，即位于左边、上方以及对角线的处理器，获得输入，类似地，结果也被送往相邻的另外 3 个处理器。B 中的每个字符被逐渐移到表的底部，而 A 中的字符被逐渐移到表的右部。现在观察一下算法的哪些步骤可以并行执行。图 12.20 中的标记给出的是方框的层次，显然同一层次的计算可以并行进行。如果每次完成一个层次的计算，则整张表格可在 $O(n+m)$步内得到。然而，并不需要 nm 个处理器，处理第 i 层的处理器也可以处理 $i+1$、$i+2$ 层的数据。因此 $\min(n, m)$个处理器就足够，问题是如何合理安排数据的移动。

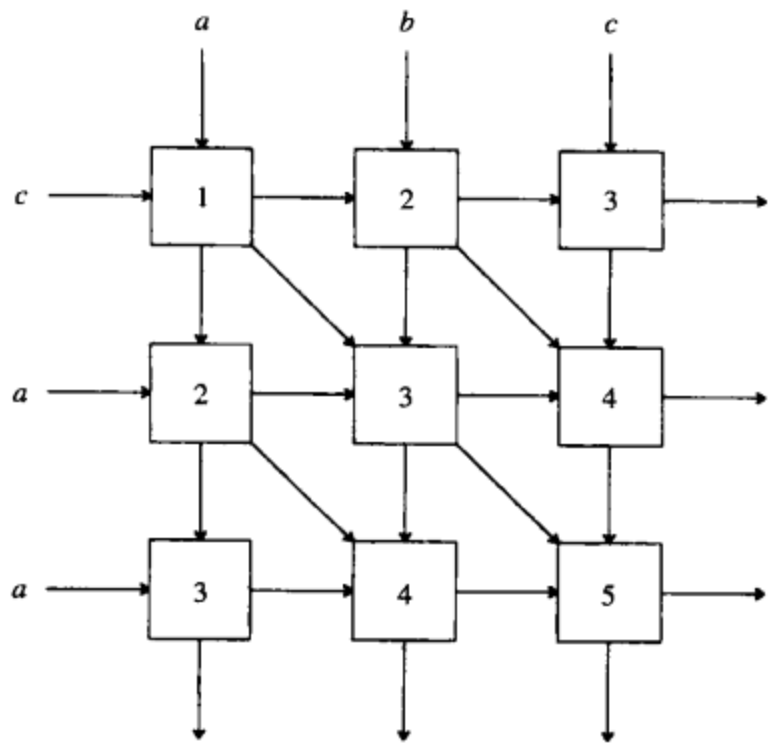


图 12.20　使用二维阵列的字符串的比较

为简单起见，假定 $m=n$，算法使用 $2n$ 个处理器组成一个双向阵列（见图 12.21）。需要保证的是每一对 a_i、b_j在某个时间于某个处理器相遇，此时该处理器知道处理这些字符所需要的 3 个值。这可以通过使 A 从左到右，使 B 从右到左，再以半速移动，另外让每个处理器包含一个用于对角线连接的存储单元来实现。半速移动不仅保证了相应的元素能相遇，而且可让目前所计算出的代价使用两个相邻元素的空槽。例如，令 $A=caa$，$B=aba$（参见 10.2.3 节）。矩阵的第 1 行（和列）是 123（相应于插入和删除这些元素的代价）。右端的输入将是 $a1b2a3$，左端的输入为 $c1a2a3$。所有处理器的内存被初始化为 0。当两个字符 a_i、b_j于某个处理器（如 P_k）相遇时，P_k对它们进行比较，在下一个脉动，a_i、b_j之后的两个字符将到达 P_k，而 P_k知道了计算矩阵中相应入口所需要的信息。$A[1..i]$和 $B[1..j-1]$匹配的最小代价来自于 P_{k-1}，$A[1..i-1]$和 $B[1..j]$匹配的最小代价来自于 P_{k+1}，而 $A[1..i-1]$和 $B[1..j-1]$匹配的最小代价来自于 P_k的内部存储器。在下一步，新计算出的入口的值被同时移到左边和右边，依次类推。图 12.21 给出了不同的 4 步，第 1 行是初始的一步，第 2 行是算法第 3 步的结果，第 3 行是算法第 4 步的结果，最后一行是算法第 7 步的结果。图中显示的内存数值以及代价是执行该步之前的数据。当最后一个字符离开最后一个处理器时，随后的数值就是最小的编辑代价。

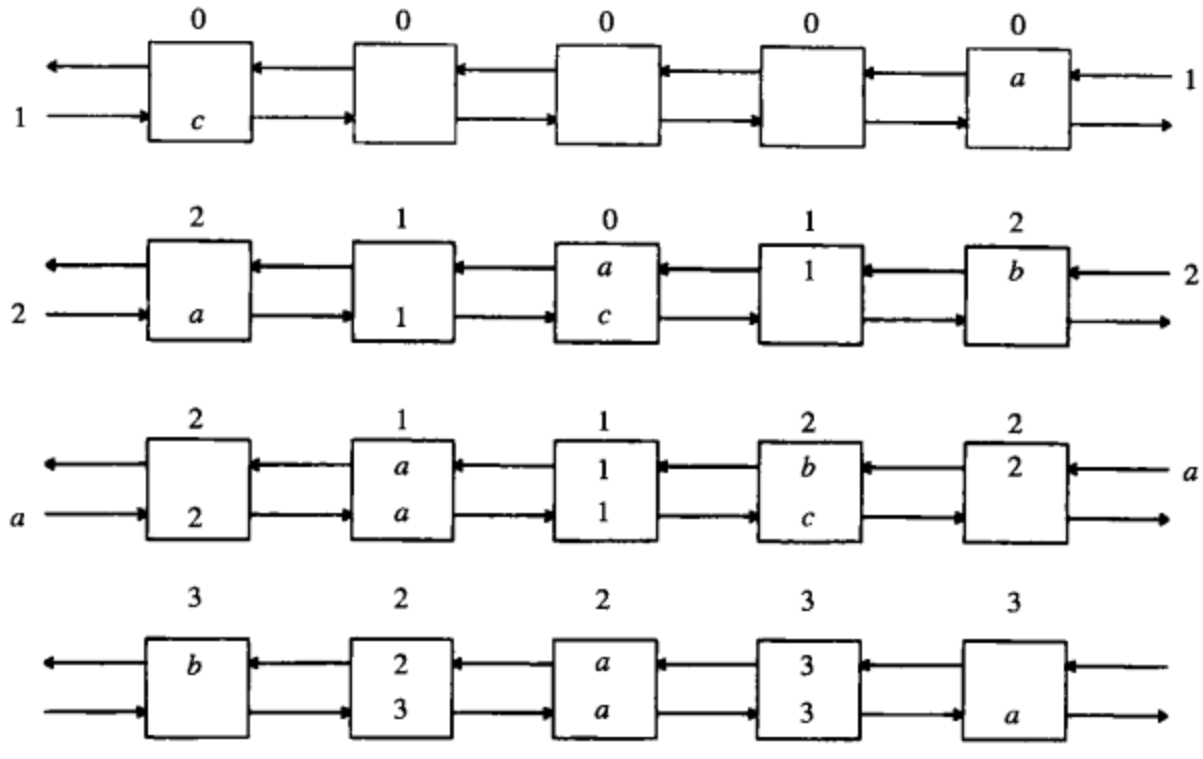


图 12.21 使用一维阵列的字符串的比较

12.6 小结

由于并行算法的设计远比串行算法的设计复杂，因此最好是充分利用积木构件。并行前缀就是一种强有力的积木构件，甚至曾被建议作为一种基本机器指令（如 Blelloch[1987]），同样的构件还有基于排列的路由操作（见 12.4.5 节）。要预料哪一种并行结构在未来将居统治地位还为时过早。然而，有必要给出对许多模型来说都通用的设计技术。本章给出了 4 种技术（当然还存在更多的其他技术）：加倍法（表秩和其他链表上的计算）、并行分治法（并行加、并行前缀和排序）、流水线（在树中查找第 k 个最小元素，一般的脉动阵列）以及欧拉遍历技术（它对许多树和图的并行算法的设计都很有帮助）。归纳法在这里再次起到了重要的作用。

参考文献和补充读物

并行计算领域的发展非常迅速，在最近几年里就出现了许多书籍，描述并行计算的书籍有 Akl[1985]，该书主要讨论并行排序问题；Hong[1986]以及 Parberry[1987]讨论了并行计算的理论模型和模型之间的练习；Quinn[1987]以及 Gibbons and Rytter [1988]是关于并行算法的一般性书籍；Hwang and Briggs [1984]、Lipovski and Malek [1987]，以及 Almasi and Gottlieb [1989]则描述了不同并行计算机体系结构和并行软件；Ullman[1984]主要涉及 VLSI；Gehringer, Siewiorek, and Segall[1987]描述了并行程序设计语言 Cm*；Fox et al.[1987]讨论了超立方体体系结构；Reed and Fujimoto[1987]则包括模型、体系结构和若干详细的应用；Bertsekas and Tsitsiklis[1989]主要涉及数值和优化方法等。Chandy and Misra [1988]提出了一种编写并行（和串行）程序的通用程序设计方法学。图的并行算法的概览可参见 Quinn and Deo[1984]，通用并行算法可参见 Moitra and Iyengar[1986]。Richards[1986]包含大量关于并行排序的文献目录。按照单/多指令流与单/多数据流对并行计算机体系结构进行分类的思想来自 Flynn[1966]。通过线路图来描述算法的例子可见 Borodin[1977]。

Valiant[1975]是较早描述寻找最大数、归并、排序等算法的文章，本书 12.3.1 节中采用了相关的材料。求解这些问题的其他并行算法可见 Shiloach and Vishkin[1981]，以及 Hopcroft[1985]。

求解表秩的另外一些算法可见 Kruskal, Rudolph, and Snir[1985]（使用了并行前缀）、Vishkin[1987]（使用了随机算法），Cole and Vishkin[1986]（提出了将一个随机并行算法转换为确定性算法的通用技术）。Cole [1988]给出了一个可运行于 CREW 和 EREW 下的 $T(n, n) = O(\log n)$的精彩归并排序算法（尽管 CREW 算法更加简单且涉及较少的限制）。Attalah, Cole, and Goodrich[1987]包含了多个 Cole 技术的应用，特别是并行集合算法。

涉及并行前缀及其应用的算法、包括并行加法，可参见 Ladner and Fischer[1980]。Fich[1983]给出了并行前缀线路所使用的门电路数目的上下界。Beame, Cook, and Hoover[1986]描述了除法和相关问题的并行化。欧拉遍历和 12.3.5 节中的算法取自 Tarjan and Vishkin[1985]，也可参见 Kruskal, Rudolph, and Snir[1986]。并行分治计算的讨论见 Horowitz and Zorat[1983]，以及 Stout[1987]。习题 12.24 取自 Kruskal, Rudolph, and Snir[1987]，习题 12.15 取自 Cook and Dwork[1982]。

排序网络，包括奇–偶归并，最早由 Batcher[1968]提出。是否存在一个深度为 $O(\log n)$的排序网络长久以来是一个公开的问题。Ajtai, Komlós, and Szemerédi[1983]证明了这种网络的存在性（尽管它们的工作确切地说是不实用的 $\mathcal{OO}(\log n)$，见 3.2 节）。Reif and Valiant[1987]给出了一个随机 $O(\log n)$ 排序网络。连接机最早由 Hillis[1985]进行描述，并在 Hillis and Steele [1986]给出了许多连接机上的算法。寻找树中第 k 个最小元素的改进算法可见 Greenberg and Manber[1987]（脉动算法见 Shrira, Francez, and Rodeh[1983]）。在最坏情况下的时间开销是 $O(\log^3 n)$的确定算法见 Frederickson[1988]。网孔上的矩阵乘法也可适用于超立方体和其他互连结构（其讨论见 Dekel, Nassimi, and Sahni[1981]）。超立方体上矩阵乘法的试验结果参见 Fox et al.[1988]。12.4.5 节的路由算法取自 Valiant[1982]以及 Valiant and Brebner[1981]（事实上前者更早明确地提出该算法，Valiant [1982]提出以随机次序来选择位，而 Valiant and Brebner[1981]则建议逐位考虑，这也是我们在 12.4.5 节中描述的）。同样的方案可见于 Aleliunas[1982]与 Upfal[1982]。使用网络来模拟共享存储器的方案来自 Karlin and Upfal[1986]、Upfal and Wigderson[1987]以及 Ranade[1987]。若需要阅读更多关于互连网络的文献，可见 Feng[1981]、Siegel[1985]和 Ullman[1984]。

关于脉动阵列的文献可参见 Kung[1979]、Kung and Leiserson[1980]以及 Kung[1982]。使用 Winograd 方法（见 9.5.1 节）的矩阵相乘的脉动阵列算法可见 Jagadish and Kailath[1989]。12.5.3 节中序列比较的脉动阵列算法取自 Lipton and Lopresti[1985]。1987 年 7 月的“Computer”杂志是脉动阵列的专集。

“以并行方式进行思考”是很困难的，设计并行算法通常远比设计相应的串行算法困难。然而，Megiddo[1983]提出了使用并行算法设计一些串行算法的独创性技术。该技术的改进可见 Cole[1984]。

操练型习题

12.1　画出 12.3.1 节所讨论的并行加法相应的线路图。

12.2　设计一个将 k 个 n 位二进制数相加的算法，该算法使用 nk 个处理器，运行时间是 $O(\log n \log k)$。

12.3　画出相应于 12.3.3 节中算法 *Parallel_Prefix_2* 的线路图。

12.4　证明算法 *List_Rank*（见图 12.3）仅需要 EREW 模型。

12.5　证明 12.3.5 节给出的构造欧拉遍历的规则是正确的。

12.6　完成 12.3.5 节计算一棵树所有节点的先序数（preorder numbering）的加倍算法的细节。

12.7　图 12.22 是表示本章所讨论的交互网络的一个图例，试确定该网络并对节点进行适当标识。

12.8　考虑奇–偶互换排序的正确性证明。证明过程中使用了归纳法说明：每当增加一个处理器（和

一个输入），执行数目就增加一步。对于归纳基础 $n=2$，算法显然只需一次比较；因此总的计算步骤似乎是 $n-1$ 而不是 n。举出一个例子说明算法确实需要 n 次比较，并讨论为什么 $n-1$ 步是不够的。

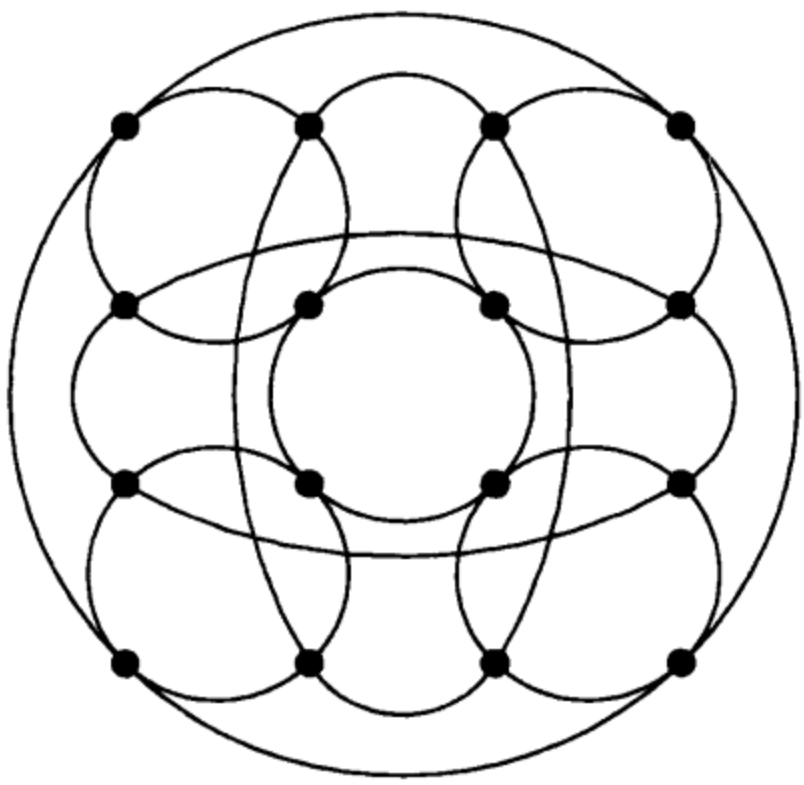


图 12.22　互连网络的例子

12.9　a. 画出归并两个长度为 8 的序列的完整线路图

b. 画出对 16 个输入进行排序的完整线路图

c. 画出对 10 个输入进行排序的完整线路图

12.10 完成图 12.21，给出所有的步骤，直到两个序列被分离为止。

创造型习题

12.11 设计一个 CREW 算法用于两个 $n \times n$ 的矩阵相乘，该算法使用 $O(n^3/\log n)$个处理器，时间开销为 $O(\log n)$。

12.12 修改 12.3.2 节中查找最大数的 CRCW 算法，使其当输入的数不是互相不同时也可以工作，该并行算法的时间开销为 $O(1)$。

12.13 假定 $A[1..n]$是一个数组，现在想将 $A[1]$复制到所有其余的位置（即广播）

a. 设计一个 $T(n, n) = O(\log n)$的 EREW 算法；

b. 设计一个效率为 $O(1)$，运行时间为 $O(\log n)$的 EREW 算法。

12.14 并行化 Horner 规则以计算 n 阶多项式，要求在 EREW 模型下的时间开销为 $O(\log n)$，效率为 $O(1)$。

*12.15 输入是由 n 个布尔变量组成的数组，输出是它们的析取（or）运算的结果。换句话说，如果至少有一个元素为 1，则输出为 1。使用 n 个处理器在 EREW 模型下计算 n 个变量析取的简单算法需要执行$\lceil \text{lb}n \rceil$步。似乎小于$\lceil \text{lb}n \rceil$步是不可能的，但事实并非如此。试设计一个小于$\lceil \text{lb}n \rceil$步的算法。可以使用任意数目的处理器和内存。（这个问题并不实际，但它说明了一些算法的下界并不是简单而显然的。）

12.16 在 CRCW 模型下设计一个时间开销为 $O(1)$的算法来解决习题 12.15（即计算 n 个变量的析取）。

12.17 画出一个深度为 $O(\log n)$，节点数目为 $O(n\log n)$的解决并行前缀问题的线路图，要求每个处理器的输入和输出数不大于 2 且恰好仅涉及一个运算。

12.18 使用习题 12.17 设计的线路图来设计一个 EREW 算法，使得并行前缀问题的时间开销为 $T(n, n) = O(\log n)$。

12.19 设 $x_1, x_2, \cdots, x_n$ 是 n 个不同的数，设计一个 CREW 算法，其在 $O(\log n)$内完成对这些数的排序，可以使用任意数目的处理器和内存。

12.20 设计一个 CRCW 算法对两个有序数组 A 和 B 进行排序，时间开销是 $O(1)$，可以使用任意数目的处理器和内存。

12.21 设 S 是一个包括 n 个元素的集合，其中 $k > 1$ 是一个常数，设计一个 CRCW 算法找出 S 的最大元素，要求 $T(n, n^{1+1/k}) = O(1)$。

12.22 设 $x_1, x_2, \cdots, x_n$ 是 n 个（不一定互不相同的）整数，每个数的取值范围为数 1 至 n，设计一个 CRCW 算法找出最大者，要求 $T(n, n) = O(1)$。

12.23 设 $x_1, x_2, \cdots, x_n$ 是 n 个实数，S 是$\{1, 2, \cdots, n\}$的一个子集，•是实数上的满足结合律的二元运算。S 将序列分成组，其中第 1 组包含元素 $x_1, x_2, \cdots, x_i$，这里 i 是 S 中的最小元素，第 2 组包含元素 $x_{i+1}, x_2, \cdots, x_j$，其中 j 是 S 中的次最小元素，依次类推（当 n 不属于 S 时最后一组包含序列剩下的那些元素）。组上的并行前缀问题需要对所有组分别计算并行前缀。请说明组上的并行前缀问题可以使用与一般的并行前缀问题一样的资源加以解决。

12.24 两个数组 $A[0..n-1]$和 $B[0..m-1]$的叉积是数组 $C[0..nm-1]$，满足如下条件：如果 $i = km+r$，$0 \leqslant k \leqslant n-1$，$0 \leqslant r \leqslant m-1$，则 $C[i] = A[k]B[r]$。（换句话说，C 是一个 $n \times m$ 矩阵的行表示，该矩阵的第 ij 个元素是 $A[i]$和 $B[j]$的乘积。）设计一个使用 p 个处理器计算 C 的 EREW 并行算法，该算法的时间开销为 $O(mn/p+\log p)$。可以使用 $O(mn)$空间（为了简单起见，可以假定 n、m 或 p 是 2 的幂）。

12.25 输入是一个由记录组成的数组。每个记录包含一些数据（对本习题来说这些数据是非实质的）和一个布尔变量 *Mark*，目标是将那些 *Mark* 值为 1 的记录安置到数组的开始处。（例如，为了某种目的而选择了一些记录，对这些记录要进行额外的处理，因此最好是将它们存放在连续的区域。）当移动记录时要求不改变它们的次序。设计一个 $T(n, n) = O(\log n)$的 EREW 算法。

12.26 输入为一个链表，其所有元素（以任意顺序）存放在一个数组之中。令 $F[i]$是一个与元素相关的标记。设计一个 $T(n, n) = O(\log n)$的 EREW 算法，该算法按原先的顺序将那些 $F[i]=1$ 的元素组成一个新的链表（例如，12.3.5 节中先序的计算就需要该算法）。

12.27 输入为一个链表，其所有元素（以任意顺序）存放在一个数组之中。请设计一个构造链表的算法，该链表的元素顺序和原先链表的元素顺序正好相反（即反序）。假定有充足的空间以保存额外的指针。

12.28 设计一个 $T(n, n) = O(\log n)$的 EREW 算法以计算由欧拉遍历方法得到的表中向前边的秩。换句话说，对每一条向前边，给出它之后的向前边的数目（该数在本书中记为 $f(i, j)$，在计算与欧拉遍历相关的先序时需要，见 12.3.5 节）。

12.29 假定树 T 不包含额外的连接边的两个副本的指针，而以通常的邻接表表示，试设计一个快速高效的 EREW 算法以构造这些指针。假定有充足的工作空间。

12.30 假定树 T 使用 12.3.5 节所描述的邻接表表示，设计一个 $T(n, n) = O(\log n)$的 EREW 算法，对每个节点 i，给出在树中 i 的祖先的数目（即从 i 到根节点的路径的长度）。

12.31 构造一计数电路：输入为 n 个二进制数，输出是输入中 1 的数目的二进制表示。

12.32 考虑下面对奇–偶互换排序正确性的证明尝试。假设最大元素已经到了最右边，除了它，还有 n–1 个元素。但是，最大数的存在至多对每个数增加一空闲步。由于有一步可节省，故而算法准时结束。讨论为何该证明是无效的。

12.33 考虑一棵二叉数，假定每片叶子 i 包含数 x_i。在该模型下求解并行前缀问题。即计算停止时，叶子 i 包含 $x_1 \bullet x_2 \bullet \cdots \bullet x_i$。（这里•是满足结合律的可在一步之内完成的二元运算）。该算法的运行应为 $O(\log n)$。

12.34 考虑在 12.4.3 节中所描述的查找树中第 k 个最小元素的并行算法。假定元素并不是互不相同的。描述为了处理这种情况算法应做的修改。

12.35 设计一个 12.4.3 节所描述的完全二叉树互连网络上的排序算法。这里每片叶子包含一个数，目标是将第 k 个最小元素放置在第 k 片叶子。说明算法的运行时间（忽略常数系数）是最优的。

12.36 处理器以环的形式组织。处理器 P_i 包含一个 $n \times n$ 矩阵 A 的第 i 行和向量 $b = (b_1, b_2, \cdots, b_n)$ 的第 i 个元素，设计一个算法来计算矩阵与向量的积 $x = Ab$，要求算法终止时 P_i 包含 x_i（$x_i = \sum_{k=1}^{n} a_{ik} \cdot b_k$），算法的复杂度为 $O(n)$。

12.37 假定初始时，处理器 P_i 包含一个 $n \times n$ 矩阵 A 的第 i 列（和 b_i），重新求解习题 12.36 中的问题。

12.38 一个 $n \times n$ 矩阵 A 的转置矩阵是矩阵 A^T，满足 $a_{ij}{}^T = a_{ji}$。假定开始时 A 存储于一个 $n \times n$ 的网孔中且 $P[i, j]$包含 a_{ij}。设计一个时间复杂度为 $O(n)$的算法以计算 A 的转置矩阵。

12.39 给定一个 $n \times n$ 处理器的网孔，其中每个处理器包含 $n \times n$ 的黑白图像的像素，换句话说，每个处理器包含一个二进制数，1 表示所含的是黑像素而 0 表示所含的是白像素。算法的目标是发现图的连接成分。两个黑像素间如果存在一条由黑像素组成的路径（水平或垂直，对角线不算）就称这两个像素属于一个连接成分。假定在算法中属于同一连接成分的像素用惟一的连接成分标签标注。开始时，第 ij 个元素被标注为 $i \cdot n+j$。在算法中，连接成分的标签可设置为该连接成分中标注值最小的像素的标注值。在算法的每步中，包含黑像素的每个处理器查看相邻的包含黑像素的处理器所持有的黑像素的标注值，如果发现比自己的值小，就将自身修改为相邻处理器持有的值。证明该算法最终将正确地标注图的所有连接成分，但对于某些输入，算法需要 $c \cdot n^2$ 并行步，另外请说明算法的停止条件。

12.40 令 A 是一个 $n \times n$ 的矩阵，以通常的方式存放在一个 $n \times n$ 的网孔之中，再令 σ 和 π 是 $(1, 2, \cdots, n)$ 的两个排列，σ 存放于网孔的第 1 行，π 存放于网孔的第 2 行，设计一个复杂度为 $O(n)$的算法按照 σ 排列 A 的行，按照 π 排列 A 的列（从行开始），也就是说，第 i 行应该被移到第 $\sigma(i)$ 行，第 j 列应该被移到第 $\pi(j)$ 列。

12.41 考虑 n 维超立方体拓扑结构，假定每个处理器包含一个元素，请设计一个 $O(n)$的并行算法以发现 2^n 个元素的最大者。

12.42 在 n 维超立方体上求解并行前缀问题。假定每个处理器包含一个数据。算法可以对处理器赋予任意的索引，当算法终止时，每个处理器都存放着与索引相应的并行前缀。

12.43 设 $G = (V, E)$是一个有向加权图，权为正数。设计一个 CREW 算法来计算图中任意两个节点间最短路径的算法，要求算法时间开销为 $O(\log^2 n)$，这里 n 是节点的数目。可以使用任意数目的处理器和任意的图表示法。

12.44 给定邻接矩阵，对下面的问题设计相应的并行算法，在保持效率为 $O(1)$的条件下，算法可使用任意数目的处理器：

a. 确定给定的无向图是否包含一个三角形作为其子图；

b. 确定给定的无向图是否包含一个四边形作为其子图（即包含长度为 4 的回路）。

12.45 产生随机数的通常方法是使用循环迭代 $x_i = a \cdot x_{i-1} + b (\mathrm{mod}\ p)$，这里 a、b 和 p 是常数（p 是素数），x_1 是种子。设计一个并行算法，其对于给定的种子计算前 p 个随机数。算法运行时间为 $T(p, p) = O(\log p)$。

12.46 a. 证明一棵高度为 $n > 1$ 的完全二叉树（有 $2^n - 1$ 个节点）不能被嵌入 n 维超立方体（有 2^n 个节点）（将图 G 嵌入到图 H 涉及将 G 的每个节点映射为 H 中的惟一节点，同时保证 G 中的邻接节点也映射为 H 中的邻接节点）；

b. 给出一种将完全二叉树近似地嵌入超立方体的方法，也就是一些边可能映射为路径而不是映射为边，考虑嵌入的目的是希望能将完全二叉树上的算法映射为超立方体上的算法（包含合理的效率损失）。

12.47 说明如何将一个 $2^k \times 2^m$ 的网孔嵌入一个 $k+m$ 维超立方体。

12.48 考虑并行闲聊（parallel gossip）问题，假定有 $n = 2^k$ 个人，每个人都知道一些小道消息，每一轮中，每个人都将自己知道的消息（包括在前面步骤中得到的）通过通信手段与他人共享，要求每人每次最多与一个进行通信。设计一种通信模式在 $\mathrm{lb}n$ 步后，所有事被所有人知道。

12.49 另外设计一个求解卷积问题的脉动算法，要求向量 x 和向量 w 以相同的方向移动（如从左到右）。每个处理器的左边有两个输入端而右边有两个输出端，部分值在处理器进行累加。（提示：以不同的速率移动这两个向量。）

12.50 设计一个求解矩阵乘积的二维脉动算法。节点的连接同二维网孔。第一个矩阵的数据移动方向是第 1 行最先到达，第二个矩阵的数据移动方向是第 1 列最先到达，结果数据的移动方向是从最后一列开始从右到左。

12.51 一些生物学应用程序常涉及序列的比较，这些序列通常仅包含 4 个不同的字符（相应于 4 种类型的核苷）。在这种情况下，每个字符可用两位来表示。说明在 12.5.3 节脉动实现中，仅使用两位对字符移动的代价进行编码是可行的（这种短编码非常重要，它可以减少处理器间通信所需要的带宽）。

部分习题答案

既然知道了方法，那就使用它吧。

——柯南道尔

第 1 章

1.13 有不少可能的平局。下面这些州总共有 269 张选票：California, New York, Illinois, New Jersey, Massachusetts, Wisconsin, Maryland, Minnesota, Washington, Iowa, Connecticut, Rhode Island, Washington DC, Ohio, Michigan, North Carolina, Oregon 和 West Virginia。（注意：这是 1988 年选举前的数据。）

第 2 章

2.1 当 $n=1$ 时结论显而易见。归纳假设陈述了对任意自然数 x 和 y（$x \neq y$），$x^{n-1}-y^{n-1}$ 能被 $x-y$ 整除。现在将 x^n-y^n 写成 $x^{n-1}-y^{n-1}$ 的形式：

$$x^n - y^n = x(x^{n-1} - y^{n-1}) + y(x^{n-1} - y^{n-1}) + x \cdot y^{n-1} - y \cdot x^{n-1}$$

$$= x(x^{n-1} - y^{n-1}) + y(x^{n-1} - y^{n-1}) + xy\,(x^{n-2} - y^{n-2})。$$

根据归纳假设，上述表达式中前两项可被 $x-y$ 整除，而第 3 项包含 $x^{n-2}-y^{n-2}$。因此可利用强归纳法原理，将假设改为：对任意自然数 x, y（$x \neq y$），以及自然数 k（$k < n$），都有 x^k-y^k 被 $x-y$ 整除。

2.6 基础情况是 $k=1$ 时，的确 $1^2 = 1(1+1)/2$。假定 $k-1$ 时断言成立，即 $k-1$ 时和为 $(-1)^{(k-1)-1}(k-1)k/2$。k 时的和为：$k-1$ 时的和，再加上第 k 个元素 $(-1)^{(k-1)}k^2$，即

$$(-1)^{k-1}k^2 + (-1)(-1)^{k-1}(k-1)k/2 = (-1)^{k-1}(k^2 - k(k-1)/2) = (-1)^{k-1}k(k+1)/2。$$

2.8 对 n 进行归纳。$n=1$ 时结论明显成立。假定 n 时结论成立，考虑 $n+1$ 的情况。欲证

$$2^n(a^{n+1} + b^{n+1}) \geqslant (a+b)^{n+1}。$$

从右边着手。根据归纳假设，我们有

$$(a+b)^{n+1} = (a+b)(a+b)^n \leqslant (a+b)2^{n-1}(a^n + b^n),$$

将表达式变形，使之尽可能接近期待的结果（$2^n(a^{n+1}+b^{n+1})$）：

$$(a+b)2^{n-1}(a^n + b^n) = 2^{n-1}(a^{n+1} + b^{n+1} + b\,a^n + a\,b^n)$$

$$= 2^n(a^{n+1} + b^{n+1}) + 2^{n-1}(b\,a^n + a\,b^n - a^{n+1} - b^{n+1})。$$

第 1 项是我们需要的，故只须证第 2 项小于或等于 0，即证

$$2^{n-1}(b\,a^n + a\,b^n - a^{n+1} - b^{n+1}) \leqslant 0。$$

因子 2^{n-1} 大于 0 不必考虑，表达式变为

$$b\,a^n + a\,b^n - a^{n+1} - b^{n+1} = a^n(b-a) + b^n(a-b) = (b-a)(a^n - b^n)。$$

当 $a=b$ 时，表达式取最大值 0；当 $a<b$ 时，左边因子为正，右边因子为负，因此结果为负；反之，当 $a>b$ 时亦然。故表达式恒为非正，得证。

2.9 有多种证明该定理的方法。这里给出一个更一般的定理的证明。当 k 是 n 除以 3 的余数时（k 可为 0、1 或 2），我们说 n 与 k 模 3 **同余**，记为 $n \equiv k$ 。现在证明下面一个较强的定理。

□ **定理 A.1**

一个以十进制表示的数与 k 模 3 同余，当且仅当其各位数字之和与 k 模 3 同余。

证明：易用一些较小的数来检验断言。假定断言在 n–1 时成立，考虑 n 的情况。除了 n–1 的个位数字为 9 时，n–1 和 n 用十进制表示的惟一差别在于个位数字。当从 n–1 到 n 时，如果 n–1 的个位数字不为 9，则该数及其对应的各位数字之和同时增加 1，同余性保持不变。如果 n–1 的个位数字为 9，那么其个位数字变为 0，十位数加 1。因为 9 和 0 都可被 3 整除，所以从 9 变为 0 不改变同余性，而在十位数字上加 1 又与在个位数字上加 1 对同余性的影响有着同样的效果（例如，它可能导致更高位上的 9 变为 0）。

2.15 微妙之处在于，当减去数列中的一个元素（记为 a_i）以减小问题 n 的规模时，必须确保 $n-a_i \leqslant a_i$，否则归纳假设将不能用于 $n-a_i$，因为它可能再次用到 a_i，而题目的要求是用到数列中不同的数。

2.26 有两个节点的树的惟一情况是仅有一条边，这种情况下所有节点的度数均为 1。假定对 n–1 个节点定理成立，则考虑 n 个节点的情况。所有的度数均为正并且它们的和为 2n–2，所以至少有一个节点其度数 d_i 为 1，同时至少有一个节点其度数 d_j 大于 1。现在将 d_i 从序列中移除，并将 d_j 减 1，剩余 n–1 个度数，其和为 2n–4；因此，根据归纳假设，存在满足这些度数的树。在这棵树上添加一个节点，并将其用一条边与度为 d_j–1 的节点连接，就一样得到了所需要的树。

2.29 如果将两个球放入一个盒子中，这个盒子将有多于一个的球。现假定原理对 n+1 个球成立，考虑 n+2 个球时的情况。假设结论不成立，即没有盒子有多于一个的球。但这样的话，存在一个盒子中有一个球，将这个盒子和球移除，便得到了与归纳假设间的矛盾。

2.34 断言对 $n = 2$ 时显然成立。假定对 n 时断言成立，考虑 n+2 的情况。任取两个节点 v 和 u。不包含这两个节点的子图为 K_n，由归纳假设可被分为 n/2 棵生成树。我们需要扩展这些生成树包含进 v 和 u，还需要增加一棵生成树。包含在 K_{n+2} 中但不包含在 K_n 中的边是那些连向 v 和 u 的边。我们将 n 个节点分为包含相同节点数的两组 X 和 Y。新的生成树包含边(v，u)、所有连接 v 和 X 中节点的边，以及所有连接 u 和 Y 中节点的边。先前所有生成树包含 K_n 中的所有节点。对每一棵生成树，分别选择 X 和 Y 中一个不同的节点，通过加上一条连接 u 和 X 中此节点的边和一条连接 v 和 Y 中此节点的边，来扩展这棵生成树，使其包含所有 K_{n+2} 中的节点。

2.38 在证明过程中不妨加上一个隐含的微妙假定：可以理所当然地认为最大值始终存在。当所有 x_i 都为正整数时，存在有穷个可能的集合其和为 S，所以最大值始终存在。然而，当有一些 x_i 为实数时，最大值就不一定存在。提示：此证明过程仅对整数成立。

第 3 章

3.6 递归很容易被消除：

$$T(n)=T(n-1)+n/2=T(n-2)+(n-1)/2+n/2=\cdots=$$

$$1+2/2+3/2+\cdots+n/2=n(n+1)/4+1/2\ 。$$

3.8 设 n 是 2 的幂。猜测 $T(n)\leqslant cn(\mathrm{lb}\,n)^2$。只需要验证对某常数 c 有（参见 3.5.1 节）：（1）$c2(\mathrm{lb}2)^2\geqslant 4$；（2）$c(2n)(\mathrm{lb}(2n))^2\geqslant 2cn(\mathrm{lb}\,n)^2+4n\mathrm{lb}(2n)$。（2）可以简化为 $2cn(\mathrm{lb}n)^2+4cn\,\mathrm{lb}n+2cn\geqslant 2cn(\mathrm{lb}\,n)^2+4n\,\mathrm{lb}\,n+4n$。对 $c\geqslant 2$，两个不等式显然成立。

3.14 $$\sum_{i=1}^{n} i^k\,\mathrm{lb}i\leqslant\int_{x=1}^{x=n+1}x^k\,\mathrm{lb}x\mathrm{d}x=\frac{\mathrm{lb}\,e}{k+1}\left(x^{k+1}\log x-x^{k+1}\right)\Big]_{x=1}^{x=n+1}=O(n^{k+1}\log n)$$

3.15 该断言的一个反例为：$f(n)=3n,\ g(n)=2n,\ s(n)=r(n)=n$

3.17 可以构造两个函数，对于所有偶数，其中一个增长较快；而对于所有奇数，另外一个增长较快。令 $f(1)=g(1)=1$，可如下递归定义这两个函数：

$$f(n)=\begin{cases}f(n-1)+1, & \text{当 } n \text{ 为奇数时}\\ f(n-1)+(g(n-1))^2, & \text{当 } n \text{ 为偶数时}\end{cases}$$

$$g(n)=\begin{cases}g(n-1)+1, & \text{当 } n \text{ 为偶数时}\\ g(n-1)+(f(n-1))^2, & \text{当 } n \text{ 为奇数时}\end{cases}$$

对于所有偶数，$f(n)$大约为 $g(n)$的平方，而对于所有奇数，$g(n)$大约为 $f(n)$的平方。因此，$f(n)\neq O(g(n))$并且 $g(n)\neq O(f(n))$。

3.19 可以利用定理 3.4，因为 m 和 c 是常数，结果为

$$S(n)=O\left(n^{\log_m(cm\log m)}\right)。$$

3.21 答案是 $T(n)=O(n)$。为了证明此结论，我们猜测对某常数 d 有 $T(n)\leqslant dn$。因此只须证明

$$dn\geqslant\sum_{i=1}^{k}a_i(dn/b_i)+cn=dn\sum_{i=1}^{k}a_i/b_i+cn,$$

因为 $f=1-\sum_{i=1}^{k}a_i/b_i>0$，因此对 $d\geqslant c/f$ 不等式成立。

3.25 如果将项 $T\left\lfloor\sqrt{n}\right\rfloor$ 替换为 $T(n/2)$，则得到了一个典型的分治关系，根据定理 3.4，其解为 $O(n\log n)$。对 $n\geqslant 2$，$\sqrt{n}$ 不大于 $n/2$，故 $T(n)=O(n\log n)$。然而，有可能 $T(n)$比此更小。试着使 $T(n)=cn$，显然 $c\geqslant 1$。考虑 $2n$，需要证明 $2cn\geqslant cn+\sqrt{cn}+n$，对 $c=2$ 和 $n\geqslant 2$ 此结果明显成立。

3.29 a. 如果 n 是 2 的幂，递归的上面一部分将始终被用到。在这种情况下，用 k 代替 n，可得下面的递推关系：

$$T'(k)=T(2^k)。$$

$$T'(k)=T'(k-1)+1,\ T'(0)=1。$$

很明显，此递归的解为 $T'(k)=k+1$。

b. 现在考虑 $n=2^k-1$。如果使用递归的下面一部分，那么 $T(n)$被 $T((n-1)/2)$替换，$(n-1)/2=2^{k-1}-1$。换句话说，如果从 $n=2^k-1$ 开始，那么我们总是使用递归关系的下面一部分，因为被包含的数将总是 2^j-1（对某个 j）的形式。现在可以定义 $T'(k)=T(2^k-1)$，并获得递归关系：

$$T'(k) = 2T'(k-1),\ T'(1) = 1。$$

这个递归关系的解显然是 $T'(k) = 2^{k-1}$。对 2 的幂和非 2 的幂，递归关系的表现会大不相同，因为它的定义在偶数和奇数上有很大的区别。如果限制为 2 的幂，则递归关系的解是一个表现“良好”的函数。特别是，大多数算法的运行时间表达式是单调递增函数。如果 $T(n)$和 $T(2n)$的值的差别只是一个常数且 $T(n)$是单调递增的，那么对 $n < m < 2n$，$T(m)$只是一个常数时间 $T(n)$。因此，在那种情况下对 2 的幂的分析是足够的。

第 4 章

4.5 因为 16 被占用，所以 16 的父节点 8 被占用，同理，4、2、1 都必须被占用。因此，用大小为 16 的数组表示的堆中至少包含 5 个元素。

4.12 可以利用大小为 n 的数组，并将元素 i 放在位置 i。这样，插入操作较容易，而删除则复杂些，因为 i 没有被逐一登记，我们不知道哪些位置被占用。可以用一个链表连接所有被占用的位置。插入 i 的工作包括标记第 i 个位置，并将 i 插入到链表中，删除操作则将第一个元素从链表中删除，并去掉元素的标记。

4.13 先从堆中任意删除一个元素（例如它的根），接着重新调整该堆。然后将被删除的元素作为新的根，两个堆的根节点连向新的根作为其子节点。新的根可能处于不合适的位置，但可以用 $O(\log(m+n))$步将其沿着组合成的堆向下调整到正确位置。

4.15 利用 AVL 树，插入和删除操作可在 $O(\log n)$步内完成。为了完成 *Find_Next* (x)，可对 x 采用常规的搜索算法，但当查找到 x 并且 x 不是叶子节点时可以沿右分支向下继续查找。如果 x 不是叶子节点，那么所查找的节点正是查找过程中的最后节点。如果 x 是叶子节点，那么所查找的节点是从根到 x 的路径上最后一次在此转向左分支的节点（如果始终沿着右分支搜寻，那么 x 是树中的最大节点）。在搜索过程中，我们可以记录最后一次在此转向左分支节点的指针，当 x 是叶子节点时可以利用此指针。

4.17 利用 AVL 树，插入和删除操作可以在 $O(\log n)$步内完成。对每个节点 v，添加一个新的域，表示为 $v.D$，这个域记录 v 的所有子孙（包含 v）数。可以利用这个域来计算在搜索过程中节点 v 的“等级”（即树中小于或等于 $v.Key$ 的节点的个数）。计算过程可利用下面的归纳法。假定我们知道从根到某节点 w 的路径上所有节点的等级，用如下方法可以找到 w 的子节点的等级。记节点 v 的等级为 $v.Rank$。

$w.Left.Rank = w.Rank - w.Left.Right.D - 1$。

$w.Right.Rank = w.Rank + w.Right.Left.D + 1$。

（如果某个子节点不存在，则 D 为 0。）为了完成 *Find_Next*(x, k)，我们首先找到节点 v 使得 $v.Key = x$（如果不存在此节点，则搜索失败）。然后搜索等级为 $v.Rank+k$ 的节点。根据等级搜索与根据关键字搜索一样简单。然而，到此工作并没有完成。因为引入了一个新的域，我们必须考虑在插入或删除过程中如何维护该域。这个问题可以这样来解决：当插入（删除）一个元素时，增加（减少）搜索路径上所有节点的子孙数。（注意，当插入或删除不成功时，应该恢复这些变化。）

这里有一个问题：为什么添加的是子孙数的域，而非等级数的域？哪种方案可使搜索更简单？

4.20 设左树（即有着较小元素的树）的高度为 h_1，右树高度为 h_2。假定 $h_1 \geqslant h_2$（其他情况相似）。首先，从左树中删除最大的元素，记这个元素为 r。然后，将 r 作为右树的新的根，并把这

个根插入到左树右侧合适的位置。更精确些，只取右分支用 h_1–h_2 遍历左树。设在这个位置的节点为 v，它的父亲为 p。新连接成的树将使 r 作为 p 的右儿子于 v 的位置上，v 作为 r 的左儿子，右树的根作为 r 的右儿子（右树保持它的根在 r 的右侧）。容易验证这是一棵二叉查找树。插入操作可能会破坏 AVL 性质，这种情况可以用通常的旋转操作加以恢复。

4.23 设 T_h 是最坏情况下高度为 h 的 AVL 树（即有着最少节点的树）。显然，T_1 是有着一个根和一个儿子的树。T_2 有着一个根和两个儿子，其中一个儿子又有它自己的一个儿子，令一个儿子是叶子节点。一般说来，可假定已经知道如何构造 T_h，再考虑 T_{h+1} 的构造。T_{h+1} 有一个儿子，此儿子同时是高为 h 的子树的根。因为这棵子树必须满足 AVL 树的性质，并且希望其有着最少个数的节点，我们可以利用 T_h。此外，对根来说 AVL 树的性质也必须满足，根节点的另一个儿子必须是深度至少为 h–1 的子树的根。有着最少节点的子树为 T_{h-1}。因此，T_{h+1} 有一个根连接 T_h 和 T_{h-1}。T_{h+1} 的节点数 $N(h+1)$满足

$$N(h+1)=N(h)+N(h-1)+1。$$

这是一个斐波那契递推关系。它的解是 AVL 数节点个数的下限。这个下限可被用来证明有着 n 个节点的 AVL 树高度的上限。

4.27 使用一个记做 S 的额外的数组，它以一种特殊方式来保存数的和。首先假定对某个 k 有 $n=2^k-1$，将从 1 到 2^{k-1} 所有数的和保存在 $S[2^{k-1}]$。这样，就将问题划分为规模减半的子问题，即维护数组的左半部分（从 1 到 $2^{k-1}-1$）和右半部分（从 $2^{k-1}+1$ 到 2^k-1）即可。每一个子问题都可通过归纳法求解。对第一个子问题，除了每次加法操作时需要适当地改变 $S[2^{k-1}]$外，能够独立解决。对第二个子问题，除了在 *Partial_sum* 的返回值上加上 $S[2^{k-1}]$外，也能独立解决。通过递归可以发现，$S[i]$保存的是从 $A[j]$到 $A[i]$中所有数的和，其中 j 是最接近 i（比 i 小的最大的 2 的幂的值加 1）的下标。例如，$S[12]$保存 $A[9]+A[10]+A[11]+A[12]$，$S[6]$保存 $A[5]+A[6]$。问题的详细解答略。

4.29 利用 AVL 树我们能够直接实现除 *Add_all* 外的所有操作。为了支持 *Add_all* 操作，可以在 AVL 树上绑定一个全局变量 *Scale*。现在 *Add_all*(*y*)只须简单地在 *Scale* 上加上 *y*，*Find_value*(*x*)在 *x* 的值上再加上 *Scale*，*Insert*(*x*, *Value*)将 *x* 的值设为 *Value*–*Scale* 即可。

第 5 章

5.4 循环不变量如下：

只有前面 *next*–1 人中的一位有可能是候选者，并且如果 *next* = *j*，那么它就是 *i*，如果 *next* = *i*，那么它就是 *j*。

5.12 问题的解与最大连续子序列相似，但需要增强归纳假设。可假定我们知道：（1）乘积最大的子序列；（2）乘积最大的后缀；（3）有着最小负乘积的子序列；（4）有着最小负乘积的后缀。如何处理这些情况略。

5.14 对有着一个或两个节点的树，解是平凡的。假定我们知道对有着 n 个节点的树如何求解此问题，考虑有 n+1 个节点的树 T。设 v 是树 T 上的任意一个叶子节点，再设 w 是连接 v 的惟一节点。如果删除 v，则将得到一个小一些的树 T'。从 v 到树 T'上任意一个节点 u 的距离，等于 w 到 u 的距离加 1。

5.15 设 v 是 G 上任意一个节点，我们将其指定为树的根。如果从树上删除 v，那么将得到几个规模小一些的树，它们的根是与 v 相邻的节点。重复这一过程，每次将得到一些新的根和小一些的树，直到所有的边被全部删除。对该过程所需重复的次数应用归纳法。如果对应于树

的直径的路径不包含根，那么树的直径同时也是其中一个小一些的树的直径，而这可以用归纳法得到。如果根包含在对应于直径的路径中，那么此路径是连接两个小一些的树中距离根最远的节点。观察结果提示可使用如下归纳假设（这要比直接归纳强一些）：

归纳假设：*我们知道如何找到有着小于 n 个节点的子树的直径，并且知道怎样找到距离根有最大距离的节点*

基础情况是平凡的。给定具有 n 个节点的树，设 v 是它的根，对所有以 v 的儿子为根的子树用归纳法。注意我们找到的距离是到 v 的儿子的距离，而非到 v 的距离。但是，为了找到到 v 的距离，只须在左右这些距离上加 1 即可。然后，比较所有子树中的最大直径，以及两个与 v 的最大距离之和，其中较大的就是直径。

5.21 假定存在总和为 k 的子集。将输入集中去掉一个元素作为黑盒的输入。如果答案是“yes”，那么解中没有此元素；如果答案是“no”，那么此元素是必要的。无论是哪种情况，问题的规模总是减小了，因为其中一个元素的状态已经被确定。

第 6 章

6.3 可以将文本先分成两半再利用折半查找，即对其中的一半运行算法。如果有错，那么可以确定错误在这一半中，可继续对这一半采用同样的方法；如果没有错误，那么错误在另外一半中（假定错误的文本没有被切割）。

6.11 修改算法 *Partition*（见图 6.9）使其满足：如果支点不等于 *X*[*Left*]，则先将支点同 *X*[*Left*]交换。设 $X[1]=1$，$X[n]=2$，其他所有元素均大于 2。在这种情况下，支点显然为 *X*[*n*]。根据上面提到的，算法 *Partition* 将首先交换 $X[1]$和 $X[n]$，然后交换 $X[n]$和 $X[2]$（因为 $X[n]<2$，$X[2]>2$）。由于所有其他元素都比 2 大，所以没有发生其他的交换。此次分割的结果是将 1 和 2 放到了正确的位置（数组的开头）。快速排序将递归地对数组的第三个到最后一个元素进行排序。继续这种模式，我们需要把第三小的元素放在第三个位置，因为分割后最后位置上的元素最初位于第二位，所以需要把第四小的元素放在第二位。例如：对 $n=8$，一个坏的次序为：1、4、3、6、5、8、7、2。

6.14 将这个集合划分为两个大小相同的子集（如果原集合大小为奇数，则划分为大小相差 1 的两个集合）。递归求解每个子问题，并合并结果。每一个递归的解是两个值：最大元素和最小元素。合并结果只需两次比较：一次比较最大元素，一次比较最小元素。由此可得下面的递归关系：

$$T(n)=2T(n/2)+2,\quad T(2)=1。$$

通过归纳容易验证 $T(n)=3n/2-2$：$T(2)=3\times2/2-2=1$，$T(2n)=2T(n)+2=2(3n/2-2)+2=3n-2$。比较次数的节省出自基础情况。找到两个数中最小和最大的只需一次比较。对 $n/2$ 个元素对进行如此操作（在最终展开递归关系时），所以可节省 $n/2$ 次比较。有趣的是，当 n 不是 2 的幂时将需要多于 $3n/2-2$ 次的比较。例如，由 $T(3)=3$，可得 $T(6)=8$，$T(12)=18$，如此等等。在这种情况下，6.5.1 节建议的算法要好一些。

6.19 考虑 $w=A[n/2]$（不失一般性，假定 n 为偶数）。如果 $w\geqslant z$，那么 z 必然出现在数组 $A[1]$和 $A[n/2]$之间，因为数组中两个连续元素的差距至多为 1，因此没有数字能够被跳过。如果 $w<z$，同理，z 必然出现在 $A[n/2+1]$和 $A[n]$之间。任何一种情况下，我们都可以用一次比较将搜索空间减半。因此最大比较次数为$\lceil \mathrm{lb}n \rceil$。

6.20 可以利用简单的信息论下界原理。习题 6.19 中问题的可能答案有 n 个（z 可以等于 $A[1]$、$A[2]$，如此等等）。因此，每一棵解决此问题的决策树必须有至少 n 个叶子（其中第 i 个叶子对应于“z 和 $A[i]$相等”）。这棵决策树的高度至少为$\lceil \mathrm{lb}\, n \rceil$。

6.22 a. 对集合 S 排序，然后对 S 中每一个元素 z，折半查找 $x-z$；

b. 利用归纳法。

归纳假设：对排好序的小于 n 个元素的集合知道如何求解问题。

假定 S 为增序。考虑 $S[1]$和 $S[n]$，设 $y = S[1]+S[n]$。如果 $y = x$，问题得解。如果 $y > x$，那么显然 $S[n]$不可能是解的一部分，因为对任意 i，有 $S[n]+S[i] > x$。因此，能排除 $S[n]$并用归纳法求解剩下的问题。如果 $y < x$，那么同理，可以排除 $S[1]$。

6.26 对集合排序，将最小的元素同最大的元素配对，第二小的元素同第二大的元素配对，依次类推。基于下面的理由，上述方法是正确的。假定有另外一种分割方法比此更优，其中没有将 x_1 和 x_n 配对，设 x_1 和 x_i 配对，x_n 和 x_j 配对。这样的话，我们可以调整分割方法，使得 x_1 和 x_n 配对，x_i 和 x_j 配对，而不会增加最大和，矛盾！

6.28 排列 $a_1, a_2, \cdots, a_n$ 定义了 $x_1, x_2, \cdots, x_n$ 上的一个全序列。也就是说，能够通过比较 a_i 和 a_j 来“比较”任意一对元素 x_i 和 x_j。现在想对 x_i 重新排列，使得 a_i 按顺序排列。因此，可使用任意一种排序算法，根据 a_i 的值排序并一起移动 x_i 和 a_i，由此可得期待的结果。因为我们需要就地排序，所以可以利用（例如）堆排序。

6.29 最初，将所有 d 个序列中的最小元素放入堆中。然后，在每一步中，去掉堆中最小的元素，并插入对应序列中的下一个元素。

6.31 a. 将每一个元素插入到一棵平衡二叉搜索树（AVL 树）中。树中的节点包含一个键值和指针，其中指针指向具有相同键值的所有元素组成的链表。由于不同的键值数有 $O(\log n)$，树中的节点数有 $O(\log n)$，因此树的高度为 $O(\log \log n)$。构建这棵树后，可通过中序遍历树，将这些不同的链表连接为一个。

b. 排序的下界不适用于这个问题，因为其要求所有排列都为有效输入。而这个问题限制输入有 $O(\log n)$个不同元素。

6.33 高度之和至多为

$$T(n) = 2\sum_{i=1}^{n} \lceil \mathrm{lb}((n+1)/i) \rceil 。$$

为了计算 $T(n)$，先看一个小的具体例子。考虑 $T(7)$:

$$\begin{aligned} T(7) &= \mathrm{lb}8 + \mathrm{lb}4 + \lceil \mathrm{lb}8/3 \rceil + \mathrm{lb}2 + \lceil \mathrm{lb}8/5 \rceil + \lceil \mathrm{lb}8/6 \rceil + \lceil \mathrm{lb}8/7 \rceil \\ &= 3+2+2+1+1+1+1。 \end{aligned}$$

其中有 4（$=(n+1)/2$）项大小为 1，两项为 2，一项为 3。假定 $n+1 = 2^k$，不难发现 $T(n)$有$(n+1)/2$ 项大小为 1，$(n+1)/4$ 项大小为 2，……，1 项大小为 k。（这正好是完全二叉树中所有节点的高度。）所以，总的来说，

$$T(n) = \sum_{i=1}^{k} i2^{k-i} 。$$

在 3.4 节中这个和被证明是 $O(n)$。如果 $n+1$ 不是 2 的幂，那么对每一项至多可能有 1 的额外贡献，其加起来不超过 n。

6.36 假设算法 A 通过一系列比较找到了 x_i 为第 k 大的元素，但根据其结果不能确定一个特定的

元素 x_j 是比 x_i 大还是小。这意味着，x_j 等于一个大于 x_i 的值 y，或者等于一个小于 x_i 的值 z。而每种情况都将导致第 k 大的元素有所不同，矛盾！

6.38 方法是一次考虑 $2k$ 个元素。从最初的 $2k$ 个元素开始，找出它们的中间数。所有比中间数大的元素可以被排除。然后看下 k 个元素，执行同样的操作。这个过程大约需要 n/k 个阶段，每个阶段包含计算 $2k$ 个元素的中间数。

6.41 对这个问题正好有 $n+1$ 种可能的答案，所以每一个求解此问题的决策树必须至少有 $n+1$ 个叶子。依此很快就可以证明原断言。

6.44 这个问题将在 10.2 节中讨论。

6.46 基本思路是对已知的部分模式构建 *next* 表，并随更多模式中的字符被识别而扩展。这并不难做到，因为 *next* 表的构造与不同的模式匹配问题类似（参见 Takaoka[1986]）。

6.51 b 部分对 a 部分的求解起到了一个很好的暗示作用。在不允许进行字符串替换的假设下，我们找到 T 与 P 间的最小修改距离（最小修改序列可以使用同一算法，除了将一次替换的代价设为 2）。两个序列的 LCS 是在最小修改序列中匹配的字符的集合（即，那些没有被包含在插入和删除操作中的字符）。SCS 是 LCS 加上所有插入和删除操作。证明略。

6.54 最简单的求解此问题的方法是利用常规的序列比较算法，除了 $C(i,0)$（参见算法 *Minimal_Edit_Distance*）的初始值为 0。换句话说，插入 A 的开始部分并不花费任何代价，这正是我们所需要的。

6.60 正确的答案比快速得到的解答要好（这个答案并不让人意外）。假定一个拉斯维加斯算法，预期的运行时间是$T(n)$，并且每次都产生正确的结果。我们可以将其转换成一个蒙特卡罗算法：如果算法在 $4T(n)$步后仍未中止，则强行中止它。因为它预期的运行时间是 $T(n)$步，故它在$4T(n)$步后仍未中止的概率不超过 1/4。当被强行中止时，算法可输出任意一个值作为答案。算法现在运行在 $O(T(n))$的时间级别，并且错误概率不超过 1/4。

6.61 利用算法 *majority*，但保持多达三个不同的候选者。每一个元素与候选者比较，如果它等于其中的一个，则将相应的众数增加 1；否则，如果新的元素和所有存在的三个候选者都不一致，那么将所有候选者的众数减 1（与算法 *majority* 相同，在考虑时我们可以排除四个不同的值）。如果有少于三个的候选者，则将当前元素作为新的候选者。最后，所有留下的候选者与所有的元素进行比较。

第 7 章

7.4 设 T 是一个满足此问题条件的树，考虑图的其他部分。设 e 是不在 T 中的边。因为 e 不在 T 中，所以 e 必然是后向边，也就是说，e 必然连向其祖先的一个节点。但是，由于 T 同时是 BFS 树，所以 e 必然连接处于同一层或相邻层的两个节点。这是矛盾的。因此，满足问题约束条件的图是树。

7.8 断言是错误的。

7.24 此问题的条件暗示着 G 是欧拉图。考虑 G 的一个欧拉回路 C 以及它的遍历。

7.27 进行深度优先搜索（广度优先搜索同样可行），并构建一棵 DFS 树。证明树中的任意一片叶子满足问题的条件。

7.29 用归纳法求解。基础情况很简单。现在缩小问题为 $n-1$ 个数，而约束相同。因为所有 d_i 为正，并且它们的和为 $2n-2$，因此至少存在一个 k 使得 $d_k=1$，同时也存在 j 使得 $d_j>1$。因此，可以从列表中将 d_k 移除，并从 d_j 中减去 1。这是一个大小为 $n-1$ 的有效问题（所有数为正，其和是 $2n-4$），可通过归纳法求解之。找出一棵有着对应度数的树 T，现在仅需要在 T 中对

应于 d_j 的节点上加上一片叶子，便可得到求解原问题的树。

7.34 可利用 6.7 节中的 KMP 模式匹配算法，并结合一种类似于 DFS 的树的（先序）遍历进行问题求解。在遍历过程，对于每一个顶点，算法需要维护关于当前潜在匹配的状态。可定义 *v.match* 等于模式中与从根到 v 的路径的后缀相匹配的最大前缀的大小。（这正好与 KMP 算法中维护的信息相同。）对 v 来说 *preWORK* 可定义为 KMP 算法中的下一步包含 v 的字母，并计算 *v.match*。*postWORK* 定义为重新载入 v 的状态，即继续利用 *v.match* 从 v 进行匹配。算法的运行时间对树中节点数目是线性的。

7.35 没有环的话，图将是一棵树，这个问题很容易回答（任取一个节点为根，并令所有边的指向都是离开根的）。利用深度优先搜索，当找到第一条（也是惟一一条）指向先前访问过的节点的边的时候，就找到闭链了。这个闭链可以被标向，使得它成为一个有向闭链，并且上面所有顶点入度为 1。所有其他边的指向应该是离开该环的。这可以按如下方式进行：从闭链中任意顶点开始另一个深度优先搜索，并使所有边指向搜索过程中其被访问的方向。惟一的回边将指向搜索的根，其入度为 1（所有其他顶点入度也为 1）。

7.37 用深度优先搜索此图，构建一棵 DFS 树 T。因为图中所有其他边是相对于 T 的后向边，所以可以使所有 T 中的边指向远离根的方向，而所有其他的边指向靠近树根的顶点（即祖先）。

7.40 修改拓扑排序算法，使得所有入度为 0 的顶点一起从队列中移除并放入一组。算法如常处理这些顶点（即删除所有它发出的边，并减小相应顶点的入度），然后移除所有入度为 0 的顶点。可证明所有这些组都满足问题的条件，并且总共有不超过 k+1 个组。

7.42 从 v 开始进行广度优先搜索。在搜索过程中，遇到的每一条边(x, y)（y 是一个新顶点）都是从 v 到 y 的最短路径中的一部分。一个常见的错误是当遇到边(x, y)时增加到 y 的最短路径的计数值。然而，可能已经有许多条已知的到 x 的最短路径，而其中任何一条都是到 y 的最短路径（以边(x, y)结束）的一部分。因此，正确的解法是将到 x 的最短路径的计数值（已知）加到 y 的最短路径的计数值上。所有计数值初始化为 0。

7.43 主要的思路是不使用堆。可通过观察所有顶点来寻找一条新的最短路径。这将花费 O (|V|) 的时间，但现在更新每一条边仅花费常数级的时间（因为不必维护堆）。

7.45 给每一条边增加一个权值 1u，其中 u 代表单位（unit）。利用同一算法，但此时路径的长度是路径的常规长度加上所有单位的总和。这些单位用来比较有着相同常规权值的两条路径。除了利用这些人工添加的单位外，也可以给每一条边的权值加上一个非常小的数 x。x 的值必须足够小，使得它不影响有着不同常规权值的路径。例如，x 可以取 m/(|V|+1)，其中 m 是权值的精度（例如，当权值为整数时，$m = 1$）。

7.48 对 k 使用归纳法，现在不仅要寻找到达 w 的路径，而且要寻找到达所有其他顶点的路径。为简单起见，这里仅讨论寻找路径长度而非路径本身。$k = 1$ 的情况很明显。记 $w.SP_k$ 为从 v 到 w 并恰好包含 k 条边的最短路径长度。假定知道从 v 到其他所有顶点并包含 k–1 条边的最短路径，即对所有 w 我们知道 $w.SP_{k-1}$ 的值。现在需要考虑所有边，对每一条边(x, y)，如果 $x.SP_{k-1}$+*length*(x, y)比当前 $y.SP_k$ 小，则设置 $y.SP_k$ 为 $x.SP_{k-1}$+*length*(x, y)。运行时间因此是 O(|E|•|V|)。

7.50 算法 *Improved_Acyclic_Shortest_Paths*（见图 7.16）对有着负代价的边也能正常工作。（提示：无论何时考虑顶点，所有导向它的路径都已经被比较过了。）

7.52 这个图必然是欧拉图，因为每一条回路都对其上的每个顶点增加度 2。从另一个角度来说，欧拉图中的边总能被分割为回路的集合。注意欧拉图中存在至少一个回路，从中删除一个

回路时将得到一组（可能是一个）连通的欧拉图。

7.55 这是一个利用归纳法求解的很好的例子。对顶点数使用归纳法。基础情况很简单。设 v 是树中的任意一片叶子。删除 v，求解余留树所构成的问题，并找到满足要求的 $S(v)$ 的一个值。如果连接 v 的边指向 v，那么归纳法很简单：设边为(w, v)，那么 $S(v)$能够被设置为 $S(w)\cup\{\lambda(w, v)\}$，容易看到条件满足。难的情况出现在边为$(v, w)$时。这种情况下，设置 $S(v)$等于 $S(w)$，并改变所有其他子集，每个子集加上 $\lambda(v, w)$。容易验证所有条件都满足。

7.60 将 F 作为当前最小生成树并从 F 开始，根据习题 7.59 中的算法在 F 上添加边。

7.64 首先找到最小代价生成树，记做 T。（对 T 中所有边的答案显然是 T 本身。）对每一对顶点 v 和 w，在到 T 中连接 v 和 w 的（不同）路径中找到最大代价边。这是预处理的步骤。可能并不需要对所有顶点对获取此信息，但对所有顶点对进行计算会更简单。包含边(x, y)的最小代价生成树可以这样获得：增加一条边(x, y)到 T，然后从其形成的惟一回路中删除代价最大的边（如果(x, y)最大，则删除代价次大的边）。这条边是 T 中从 x 到 y 的路径中代价最大的边，其标识在处理的步骤中获得。处理的步骤可以对 T 深度优先遍历$|V|$次完成，每次从另一个顶点开始。

7.67 在所有代价上增加一个常数并不影响 MCST，因为对算法重要的仅仅是代价的相对顺序。无论在算法过程中什么时候用到，代价都是被用做简单的相互比较（“<”或“>”），而不以任何其他方式使用（或操作）。因此，在所有代价上增加一个常数并不影响代价的次序，从而不影响结果。

7.68 （这只是对一个相当复杂的算法的简要描述。）首先，注意到我们可以将精力集中在 T 上，并忽略图的其他部分。因为 T 已经是 MCST，G 中的任意一条其他边都与 T 中的边一起构成一个回路，并在回路中其是最大代价边。增加另一个顶点不影响结果。为简单起见，我们同时假定 v 和 T 中的所有顶点相连（例如，可以添加虚设的边，其代价为无穷）。对 T 中顶点数用归纳法。把 T 看做（可取任意一个节点为根）根树，观察连接多片叶子的内部顶点 w。每一对与 w 连接的叶子 x 和 y，同 v 和 w 构成一个回路（包含边(w, x)、(x, v)、(v, y)，(y, w)）。此回路中的最大代价边应该被删除掉，这将在新的图中形成一片新叶子。此叶子肯定在新的 MCST 中，这将减小问题的规模。但还没有成功，因为 w 可能只含有一片叶子 z。这种情况下，回路是一个由(w, z)、(z, v)和(v, w)组成的三角形。当(v, w)是回路中的最大边时，此时需要被删除的情况较复杂，原因在于没有找到新的叶子。然而，我们仍然可以通过“压缩”w 来减小问题的规模。可用边(p, z)来替代两条边(w, z)和(w, p)（其中 p 是根树中 w 的父亲），并设置这条新的边的代价为两条旧的边中较大的代价。将上述讨论转变为算法的任务留给读者，其中需要证明压缩是有效的，并且设计出合适的数据结构来维护算法运行过程中的必要信息。

7.70 主要的思路是反向解决此问题。不是按照问题中所陈述的来删除顶点，相反这里将添加顶点。我们利用合并–查找数据结构（见 4.5 节）来维护当前在图中的分支的集合。从顶点 n 开始，并将其作为分支添加到数据结构中。然后考虑顶点 n–1、n–2 等。对每一个顶点 i，首先将其作为分支添加到数据结构中，然后检查所有 $j > i$ 的边(i, j)。对每一条这样的边(i, j)，检查 i 和 j 是否属于相同的分支（利用 find 操作），如果不是，则合并对应的两个分支（利用 union 操作）。当一个分支包含多于 $n/2$ 个顶点时停止。

7.71 因为 G 中的每一个回路必须包含反馈边集合 F 中的一条边，所以集合 E–F 中不可能包含回路。要想最小化 F 的大小，就是最大化 E–F 的大小。图中没有回路的边的最大集合对应于

一棵生成树（如果图不是连通的，则对应于生成森林）。因此，无向图中最小的反馈边集合是图的生成树的补集，其大小为|*E*|–|*V*|+1（假定图是连通的）。对应于有向图此问题是 NP 完全的（参见第 11 章）。

7.74 检查所有不在树中的边。如果有一条边(*v*, *w*)满足 *v*.SP+cost(*v*, *w*) < *w*.SP（其中 *x*.SP 是通过树到达 *x* 的最短路径的长度），那么这棵树显然不是最短路径树。接下来仍须证明，如果不存在这样的边，那么这棵树确实是最短路径树。

7.79 找到一个拓扑顺序，然后对所有有着连续标号的顶点，检查它们在图中是否相连。如果是，那么此拓扑顺序就是一个哈密尔顿路径。换句话说，如果存在哈密尔顿路径，那么任意拓扑顺序将把所有节点根据其在哈密尔顿路径中的位置排序。

7.84 在 7.9 节中证明了，两条边被包含在一个闭链中，当且仅当它们属于同一个双连通分支。因此，这里只须从图中删除 *c*，并运行双连通算法，确定 *a* 和 *b* 是否同属于去掉 *c* 后的图的一个双连通分支。

7.85 证明一个稍微强一些的定理（读者可自行证明此定理蕴涵习题的断言）。

□ **定理 A.2**

如果 v 和 w 是同一个双连通分支 B 中的两个顶点，那么 v 和 w 间的任意简单路径被包含在 B 中。

证明：很明显至少有一个 *v* 和 *w* 间的简单路径在 *B* 中。如果有一条路径不被包含在 *B* 中，那么它必须先离开 *B* 后又回到 *B*。它必须从一个分支中离开 *B*，并通过另一个分支回到 *B*（因为它是简单路径，并且两个分支能够仅通过一个关节点被连接）。这蕴含着存在一个回路其包含多于一个的双连通分支，而这是不可能的。

7.87 选取任意一个关节点 *v*。从图中删除 *v* 及其关联的边，并从生成的图中找到不同的连通分支。然后将删除的边放回合适的连通分支中。再分别考虑各个分支，选取其他关节点进行同样的处理。

7.89 第一种方法可以是尝试修改强连通分支算法，即修改 *High* 值使之更准确。例如，不用 DFS 数，而用当前已知的 *High* 值。换句话说，当一条边(*v*, *w*)被考虑用做计算 *v.High* 时，则在计算中使用 *w.High* 而不是 *w.DFS*。然而，*w.High* 可能晚一些会被修改（例如，它的一个孩子可能有一个到达更高顶点的路径）。但更新所有使用先前 *w.High* 值的节点耗费太大，因此可考虑另一种方法。

记从 *v* 可到达的最大的 DFS 数为 *v.Real_High*。同一个强连通分支中的所有顶点有着相同的可到达特性。也就是说，如果其中的一个顶点可以到达顶点 *w*，那么它们中的所有顶点都可以到达 *w*。因此，对每一个强连通分量计算一个代表 *v.Real_High* 已足够。我们首先用强连通分量算法构建 SCC 图 *SG*。*SG* 中的每一个节点都被标记为其对应的强连通分支的 DFS 数（这是分支中有着最高 DFS 数的顶点）。*SG* 必然是非循环的，所以可将问题简化为寻找无环图中的 *Real_High* 值。简化的优点在于可通过反复更新已获得的值来求解问题。因为 *SG* 无环，所以没有后向边。因此，当边(*v*, *w*)被用来更新 *v.Real_High* 时，*w.Real_High* 的值是正确的最终值。读者可自行证明这个结论，例如，通过归纳（但这个证明过程不是平凡的）。

7.92 a. 假定 *v* 属于一个顶点基（vertex basis）*B*。如果 *v* 不在一个回路上，并且 *v* 入度不为 0，那么存在一个顶点 *w*，使得(*w*, *v*)是一条边并且从 *v* 不可达 *w*。因为 *B* 是顶点基，所以或

者 w 属于 B，或者存在一个从 B 中某顶点到 w 的路径。对每一种情况来说，都有一个从 B 中顶点到 v 的路径，故 v 能够从 B 中删除。这与 B 的最小性矛盾。

b. 由于非循环图中没有顶点在闭链上，从 a 部分可得仅有那些入度为 0 的顶点才能在顶点基上。但是，每一个入度为 0 的顶点必须在一个顶点基上同样成立，因为没有路径通向它。因此，非循环图的惟一顶点基是入度为 0 的顶点的集合（7.3 节曾证明过这个集合非空）。

给定一个一般图，首先找到 SCC 图（见 7.9 节）。正如已经说明的，SCC 图是非循环的。强连通分支上任意一个顶点覆盖此分支上所有其他顶点，故而在每一个分支上选择一个顶点已足够。因此，从每一个对应于 SCC 图上入度为 0 的节点的分支中任取一个顶点，顶点基便由这些顶点组成。

7.95 将这棵树看成是以任意节点为根的根树。任取一片叶子 v，将它同其父亲 w 匹配，并从树中删除 v 和 w（v 的所有兄弟将不被连接）。可以通过归纳解决余下的问题。我们需要证明边(v, w)确实属于一个最大的匹配。设 M 为最大匹配。如果 M 不包含(v, w)，那么 v 没有被匹配（它仅和 w 连接）。如果(u, w)在最大匹配中，那么可简单地用(v, w)替换它。这仍然是一个匹配，并且有着相同的势。

7.99 a. 为了得到这样的覆盖，此图必须不包含度为 0 的顶点。当我们删除任意一个顶点时，最终可能获得仅由一个顶点组成的分支。

b. 现在假定这个图仅含度大于 0 的顶点，现在要确保简化后的图也满足这一条件。只有那些度为 1 的顶点会造成一些问题。所以，如果图包含度为 1 的顶点，则删除该顶点和它的边，并把这个边（它本身也是一个星形图）添加到覆盖中。容易看到只是一个其他顶点的度减少了 1，因此这是一个有效的归纳。

7.100 用折半查找结合 7.10 节中描述的最大匹配算法。让我们首先解决这个问题：验证是否存在一个最大的匹配，使得它的所有边有着小于或等于 x 的权值（对于某个给定的 x）。这个问题可以这样解决：从图中移除所有权值大于 x 的边，应用最大匹配算法看余下的图的最大匹配是否与原图的最大匹配有相同的边数。这个验证的代价取决于最大匹配算法的代价，为 $O(\sqrt{n}m)$。G 中有 m 条边，所以权值至多有 m 种不同的可能性。现在用折半查找寻找最小的 x，使得 x 是某条边的权值，且有一个最大匹配使得它的所有边的权值小于或等于 x。（有着更低时间复杂度的算法参见 Gabow and Tarjan[1988]。）

7.104 利用分治法。先将选手划分为相同大小的两组，对每一组构建一条所需的链，然后完全按常规的合并方法合并这两条链。这个过程的正确性留给读者验证。这个算法与归并排序的相似性不是偶然的，参见习题 10.18。

7.106 显然，至少 k 种颜色是必要的，因为所有附加于同一顶点的边必须被着成不同的颜色。我们将证明 k 种颜色同时也是足够的。利用分治法，但首先必须对问题进行划分，使得子问题与原问题相同。如果将其划分为两个小一些的子图，就必须确保每个子图中所有顶点的度相等并且为2 的幂。这可以通过寻找图的欧拉回路获得（这样的回路存在，因为所有度数为偶数）。然后，遍历此回路并把边依次分成两组（也就是说，第一条边放入第一组，第二条边放入第二组，第三条边放入第一组，如此等等）。每组都使用所有原有顶点和所有原有边的一半来定义一个子图。每个子图中的每一个顶点的度恰好为 $k/2$，因为无论何时，当进入一个顶点的边放入一个组时，下一条边（离开这个顶点）将被放到第二组。余下的可以通过归纳法完成。现在有一些图，它们有较少的边，更重要的是，有着较小的度数。分别对两个子图的所有边着色，每一组使用不同的颜色。因此，我们能够将这些色彩放在一起。这

个算法的复杂度取决于寻找欧拉回路的算法，其将花费线性时间，但必须对每一个子图执行。这个算法总共需 lb$k$ 个阶段（因为每个阶段 k 减半）；因此，总的运行时间是 O ($|E|\log k$)。（注意，对每一个最大度数为 k 的图，能够用至多 k+1 中颜色对边着色；参见 Chartand and Lesniak[1986]。）

7.108 一个解决此问题的普通方法是考虑树的根（任意一个顶点都能被当做根），对此根下所有的子树找到最小顶点覆盖。然后，如果并非它的所有儿子已经包含在子树的顶点覆盖中，则将根也包含进来。这似乎是一种好的方法，因为根必须确实在顶点覆盖中，当且仅当它的儿子不在顶点覆盖中。然而，可能有多个不同的最小顶点覆盖（见图 A.1）。对一棵给定的子树，可能同时存在一个包含根的最小顶点覆盖和一个不包含根的顶点覆盖。因此，这种方法不一定必然获得最小顶点覆盖（除非特别小心）。从叶子开始考虑可获得一个不同的归纳法。从一片叶子到它的父亲的边必须被覆盖，且能够被叶子或其父亲覆盖。选择叶子没有任何优势，因为它不能覆盖其他任何边。更精确些，存在一个包含父亲的最小顶点覆盖。因此，如果选择父亲（或任何叶子），将它和与其依附的边删除，然后通过归纳求解余下的问题，那么我们能够保证获得最小顶点覆盖。这个算法不难在线性时间内实现。

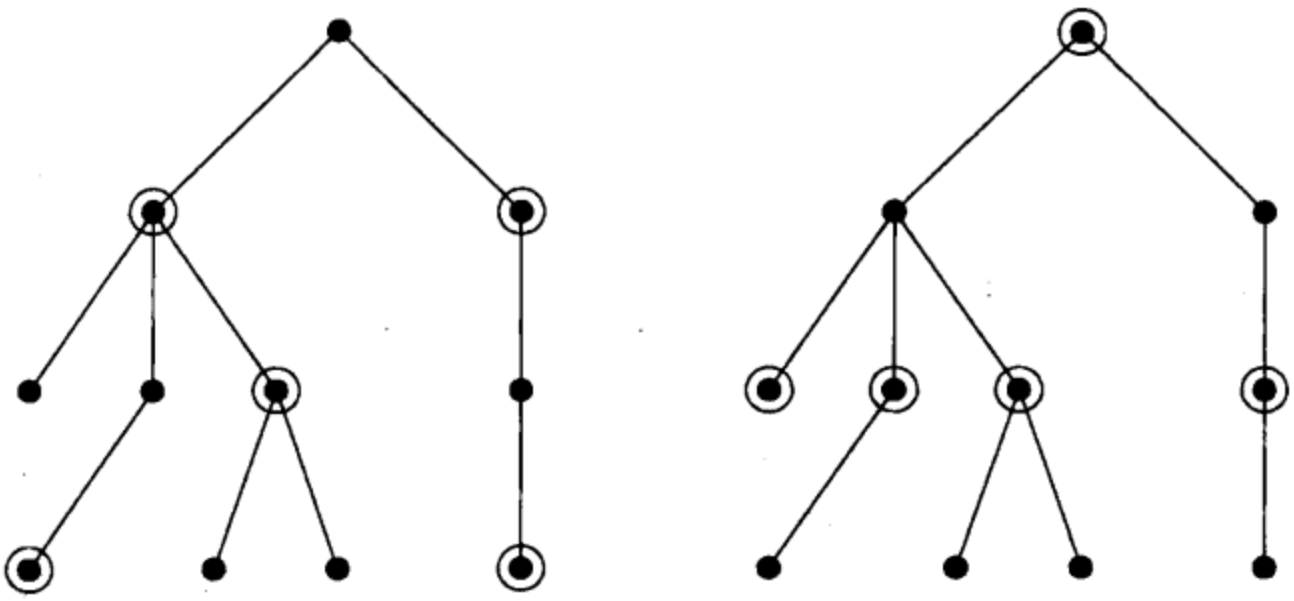


图 A.1　有着两个最小顶点覆盖的图

7.109 容易看到，在这种情况下一个顶点覆盖的权值必须至少等于边的数目（因为所有的边必须被覆盖，每条边至少对覆盖贡献 1 的代价）。一个对每一条边恰好包含一个顶点的覆盖是最小权值覆盖。下面就来证明这样的覆盖总能被找到。

方法 1：把任一顶点 v 当做根，进行广度优先搜索，并为每一个顶点根据它到 v 的距离赋予一个层数。现在把所有层数为奇的顶点增加到顶点覆盖中（所有层数为偶的顶点也同样可行）。结果是一个顶点覆盖，因为每一条边连接两邻近层中的顶点，所以其中一个顶点层数为奇。此外，每一条边都恰好被一个顶点覆盖，所以这个覆盖是最小的。

方法 2：归纳假设陈述了我们知道如何解决所有小于 n 个顶点的树的问题。现在选取任意一片叶子 v，并将它（及其依附的边）从树中移除。设惟一与 v 邻接的顶点为 w，并设余下的树为 G'，通过归纳法求解 G' 问题。再次考虑原来的树 G。如果 w 在最小权值顶点覆盖中，那么这个覆盖仍然有效。否则，在 G 的顶点覆盖中包含进 v。我们需要证明在每种情况下 G 都是最小的。但是，G 的最小权值顶点覆盖的大小至少比 G' 多 1，因为这条额外边必须被 v 或 w 覆盖（并且在每种情况下花费比 1 更多，因为 w 的度增加了）。我们得到了下限，所以结果顶点覆盖是最小的。

7.114 设 v 为 G 中的任意一个顶点。记与 v 邻接的顶点集合为 $N(v)$。如果 v 属于顶点覆盖，那么 $N(v)$中的所有顶点都不能属于顶点覆盖（因为覆盖中的顶点必须是相互独立的）。此外，所有与 $N(v)$中顶点邻接的顶点必须属于覆盖，因为这是覆盖那些边的惟一途径。特别是，如果 $N(v)$中的任意两个顶点是邻接的，那么它们间的边不会被覆盖，程序失败。这个程序运行到或者它失败（这种情况下，v 不能属于任何顶点覆盖），或者找到一个顶点覆盖为止。程序中的所有步骤取决于 v 的选择；因此，找到的顶点覆盖是惟一包含 v 且满足问题条件的顶点覆盖。如果这个顶点覆盖不是很大，那么任务就完成了；否则，v 不属于顶点覆盖。但是，如果确定 v 不属于顶点覆盖，那么它的所有邻接顶点必然属于覆盖，因此可以应用相同的程序。

7.116 每一个区间 I_j 表示为两个数(l_j, r_j)（左端点和右端点）。根据 r_i 对区间排序。记区间按所排的顺序为 $I_1, I_2, \cdots, I_n$（即 r_1 是所有右端点中最小的）。我们断言有一个包含 I_1 的最大独立集。为了证明这个断言，取任意一个最大独立集，考虑其中的一个区间 I_j，其中 r_j 是所有右端点中最小的。由于 r_1 最小，故 $r_j \geqslant r_1$。这暗含着 I_1 不与集合中任何其他区间相交（除了可能同 I_j 相交）。因此，I_1 能够替换 I_j，并且，修改后的集合仍然是最大独立集。因为知道如何处理 I_1，所以算法可直接通过归纳获得。

7.119 a. 设 v_1、v_2、v_3、v_4 为长度为 4 的闭链（按该顺序）。考虑图的邻接矩阵，并设主对角线为 0（即一个顶点不和它自身相连）。v_1 和 v_3 对应的行，在对应 v_2 和 v_4 的位置都包含 1。这是在某种意义上对长度为 4 的闭链的描述。（注意：由于图是无向的，所以矩阵是对称矩阵。）观察任意两行，对每一对检查其交集。换句话说，对每一对顶点，检查所有同它们二者都邻接的顶点。当且仅当其中任意一个交集包含至少两个顶点时，正方形的子图存在。任意行的交（主要是行的“and”操作）可以在线性时间内完成。由于总共有 $O(|V|^2)$个顶点对，所以总的运行时间为 $O(|V|^3)$。

b. 利用邻接列表表示。首先整理列表中的所有边。然后说明 a 部分描述的两行间的交，可以在与依附于对应两顶点的边数成比例的时间内完成。

7.120 计算有$|V|$个顶点的图所可能包含的最多的正方形数目，说明其可能多于 $O(|V|\cdot|E|)$。因此，仅列举出所有的正方形就将花费多于 $O(|V|\cdot|E|)$的时间。

第 8 章

8.6 图 A.2 给出了一个反例。尽管圈出的点在凸包中，但还是会被 Graham 扫描算法删除。

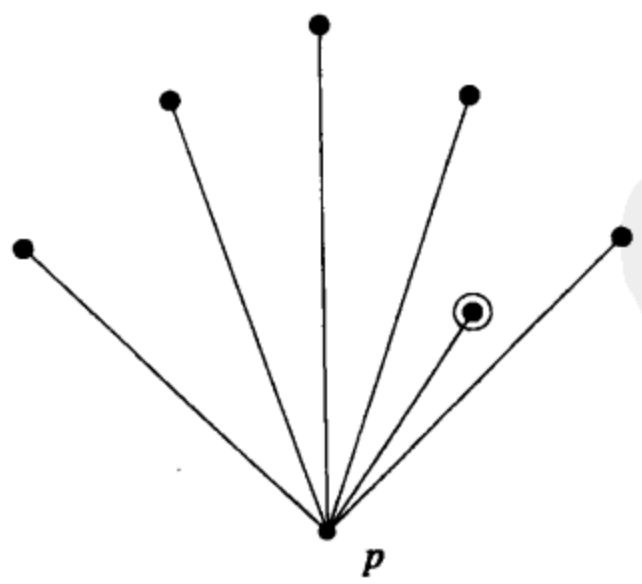


图 A.2 习题 8.6 的反例

8.11 将多边形的顶点根据其与点 q 的连接线段（与一固定直线）所成的角度进行排序。依此顺序扫描这些顶点。从任一顶点 v 开始，利用 8.2 节中讲述的算法来确定多边形中与从 q 到 v 的射线相交的边的条数。然后，当顶点 w 被访问时，可以在常数时间内确定经过 q 和 w 的直线与多边形相交的边的条数相比是多了一个还是少了一个，或者保持不变。

8.16 将平面分割为宽为 d 的垂直栏，从所有点中具最小 x 坐标的点开始，到具最大的 x 坐标的点结束。在每一栏中找到有着最小和最大的 y 坐标的点。这可以在 $O(n+X/d)$时间内完成，因为包含给定点的栏能够在常数时间内找到。先对所有最小的点再对所有最大的点应用类似于 Graham 的扫描算法。这将产生两条凸路径，再将它们的两端连接起来，形成一个凸多边形 C。C 不一定是这些点的凸包，但容易说明集合中位于凸包外的任意点到凸包的距离至多为 d。

8.17 礼品包裹算法可用于这个问题而不增加复杂度。这里必须注意每一点都要与所有余下的点进行比较。因此，算法为了找到闭包的第一个点需要 $n-1$ 次比较，找到第二个点需要 $n-2$ 次比较，依次类推。在闭包找到之后，闭包上的所有顶点都会被标记上一合适的深度，然后移去。继续同样的算法。第一个点需要 $n-k$ 次比较（其中 $k-1$ 是目前已经移除的顶点数），依次类推。总共需要 $O(n^2)$步。

8.22 惟一困难的情况是寻找竖直线段和 45 度角的线段的交点（另外的情况或者是对称的，或者可用 8.6 节中的算法来处理）。可利用与 8.6 节中描述的交点算法类似的方法进行处理。对线段根据其端点的 x 坐标排序。用同样的方法执行扫描线算法。对 45 度角的线段，当其左端点碰到扫描线时插入它，当其右端点碰到扫描线时删除它。现在必须找到一条新的竖直线和多个候选的 45 度角线段的交点。可以这样做：对每一条 45 度角的线段，计算其所在的直线与 x 轴的交点。后面当进行区间查询时会用到这个值。区间设置如下：区间的左端点是过竖直线段上端点且与水平方向成 45 度角的直线与 x 轴的交点，区间的右端点是过竖直线下端点且与水平方向成 45 度角的直线与 x 轴的交点。由此可将此问题转化为一维区间查询问题，而复杂度保持不变。

8.26 稍稍加强归纳假设：

> **归纳假设**：对于小于 n 个区间的集合，知道如何被标识包含在其他区间中的区间，并且知道如何找到它们中最大的右端点。

首先根据它们的左端点对这些区间排序。假定对前（最左边的）$n-1$ 个区间解决了此问题，考虑第 n 个区间。如果它的右端点比目前最大的右端点还大，那么它不被包含在任何区间；修改最大的右端点。否则，它将被包含在其他区间中；标识它。

8.28 边平行于坐标轴的矩形的交集可以在常数时间内计算。并且，交集或者为空，或者为矩形。因此，可以按任意顺序接连求取交集，在线性时间内求解该问题。

8.31 这里对问题的解决方案仅给出一个粗略的描述。基本思路是将每个多边形划分为厚片（slab），并分别求交集。首先，按照它们的 x 坐标排序。这个排序可以在线性时间内完成，因为每个多边形顶点的循环顺序已知。再将每个顶点与一条竖线关联，并用这些竖线将这两个多边形划分为厚片，如图 A.3 所示。因为知道竖直线的顺序，所以仅需要分别计算各对厚片的交。两个厚片的交可以在常数时间内计算完成，因为每一个厚片至多有 4 条边。然后就可以在线性时间内将对应的交集组装为多边形。

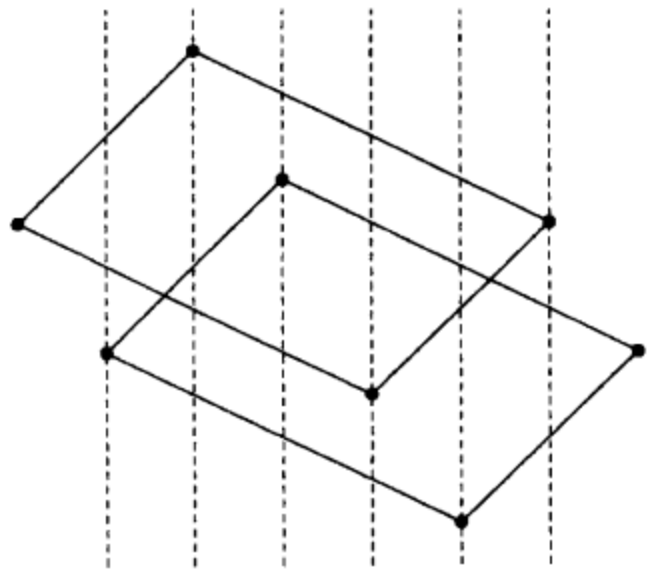


图 A.3　经厚片方法处理的两个凸多边形的交集

8.34　三角形的交集为一个凸多边形。可以在常数时间内求两个三角形的交，在线性时间内求两个凸多边形的交（见习题 8.31）。因此，利用分治算法将三角形的集合划分为两个集合，对每一个集合递归地求交集，然后求两个作为结果的凸多边形的交集，运行时间为 $O(n \log n)$。

第 9 章

9.2　一个例子是 $n = 15$。9.2 节中提出的方法需要 6 次乘法（$x^{15} = (((x^2 \cdot x)^2) \cdot x)^2 \cdot x$）。但仅用 5 次乘法计算 x^{15} 是可能的（$x^{15} = ((x \cdot x)^2 \cdot x)^3$）。对此更详细的讨论见[Knuth 1981, p.443]。

9.10　使用 9.5.2 节中的符号。需要计算的是$(B+C+D+F+G)X$。先如下定义（直接对应于 9.5.2 节中介绍的 4 种乘积类型中的 7 次乘法）：$z_1 = b(e+f)$, $z_2 = c(g+h)$, $z_3 = (c-b)(e+h)$, $z_4 = (a-b)e$, $z_5 = (a-c)(g-e)$, $z_6 = (d-c)h$, $z_7 = (d-b)(f-h)$。现在考虑各个矩阵的贡献。B 贡献$[z_1, z_1, 0, 0]$（为了方便这里写成行矩阵而非列矩阵），C 贡献$[0, 0, z_2, z_2]$，D 贡献$[0, z_3, -z_3, 0]$，F 贡献$[z_4, 0, z_4 + z_5, 0]$，G 贡献$[0, z_6 + z_7, 0, z_6]$。总之有：$p = C_{1,1} = z_1+z_4$，$r = C_{2,1} = z_1+z_3+z_6+z_7$，$s = C_{1,2} = z_2+z_3+z_4+z_5$，$t = C_{2,2} = z_2+z_6$，即有 18 次加法和 7 次乘法。

9.16　问题困难的部分在于证明这样的一种表示始终存在。现在用归纳法证明它确实存在。基本情况容易验证。设 n 是大于 2 的整数，并设 $F(k)$是不大于 n 的最大的斐波那契数（也就是说，$F(k) \leqslant n$ 且 $F(k+1) > n$）。可以断言 $F(k) > n/2$，因为否则 $F(k+1) = F(k)+F(k-1) < n$。通过归纳假设，$n-F(k)$能被表示为至多 lb$(n-F(k))$个斐波那契数的和。但是，因为 $n-F(k) < F(k)$，增加 $F(k)$到 $n-F(k)$的表示仍然保持这些数互不相同。此外，容易看到 lb $(n-F(k))+1 <$ lb$n$。为了找到所需的表示，现在计算所有的斐波那契数，直至到达一个大于或等于 n 的数，然后就可以遵从这里的证明过程。

9.19　记第一个多项式的 4 个部分为 a_1、b_1、c_1 和 d_1，第二个多项式的 4 个部分为 a_2、b_2、c_2 和 d_2。可以用图 A.4 中的 4×4 的表来揭示此问题。这个表与 9.4 节中给出的 2×2 的表相似。表中每一个条目对应于多项式中两部分的乘积。没必要计算每一个条目的值，而只须计算图中每条对角线上值的和。每条对角线对应于多项式乘积中特定度的系数。下面的 9 个乘积足以计算所有对角线的值：（1）$a_1 \cdot a_2$；（2）$b_1 \cdot b_2$；（3）$(a_1+b_1) \cdot (a_2+b_2)$；（4）$c_1 \cdot c_2$；（5）$d_1 \cdot d_2$；（6）$(c_1+d_1) \cdot (c_2+d_2)$；（7）$(a_1+c_1) \cdot (a_2+c_2)$；（8）$(b_1+d_1) \cdot (b_2+d_2)$；（9）$(a_1+b_1+c_1+d_1) \cdot (a_2+b_2+c_2+d_2)$。读者可自行验证所有对角线都能从这 9 个乘积中计算。对应的递归关系是 $T(n) = 9T(n/4)+O(n)$，这蕴涵着 $T(n) = O(n^{\log 9}) = O(n^{\log 3})$，其和将每个多项式划分为两部分的

算法运行时间相等。这是巧合，还是有理由的？换句话说，这两个算法的相似性是什么？

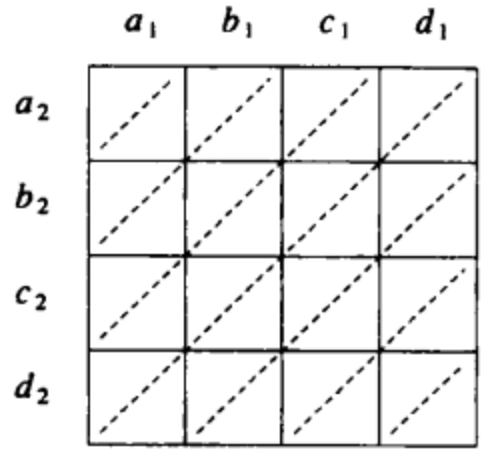


图 A.4 通过将 2 个多项式分成 4 个部分求取其积

9.23 Winograd 算法假定乘法的交换性（也就是假定 $x \cdot y = y \cdot x$）。如果使用 Winograd 算法作为递归的基础，就必须在假定中将元素替换为矩阵，但事实上这是不允许的，因为矩阵乘法不是可交换的。

9.29 通过对 n 归纳求解此问题。$n=1$ 时的情况是平凡的。假定对 $n>1$，我们知道如何求解，考虑 $n+1$。记 $M[i..j]$为 $M_i \times \cdots \times M_j$ 的积。计算 $M[1..n+1]$的最佳方法是首先对某些 i（目前未知），计算 $M[1..i]$和 $M[i+1..n+1]$，然后将这两个乘积相乘。可以通过尝试所有可能的情况来找到使代价最小的 i 的值。通过归纳就能知道计算 $M[1..i]$的最佳方法。然而，还需要知道如何计算 $M[i+1..n+1]$。为此，加强归纳假设：

强一些的归纳假设： 对所有 $1 \leqslant i < j \leqslant n$，知道计算 $M[i..j]$的最好方法。

为了扩展这个假设到 $n+1$，需要对所有 $1 \leqslant i \leqslant n$ 计算 $M[i..n+1]$。这里可通过另一个（嵌套的）归纳求解此问题，这次基于 i，但是使用相反的顺序。对 $i=n$，仅有一种方法计算乘积 $M[n..n+1]$；因此，这个问题是平凡的。假定我们知道计算 $M[i..n+1]$的最佳方法，考虑 $i-1$。现在可以进行归纳。对任意 j：$i<j<n+1$，检查计算 $M[i-1..j]$的代价（通过原始归纳知道）、$M[j+1..n+1]$的代价（通过嵌套归纳知道），以及将两个乘积相乘的代价。然后选择使代价最小的 j。总的来说，这两个归纳过程对应于两个循环，第一个大小为 n，第二个大小为 $n-i$，内部循环由 $n-i$ 步组成。因此总的步数为 $\sum_{k=1}^{n}\sum_{i=k}^{n}\sum_{j=i}^{n}O(1)$，这是 $O(n^3)$级别的。

第 10 章

10.3 这里说明区间包含问题是如何归约为极大点算法的。对于每个区间 $I_j=(L_j, R_j)$，定义平面上的一个点使其 x 坐标为 $-L_j$，y 坐标为 R_j，读者可自行证明一个区间包含于另一个区间当且仅当其对应的点不是极大点。

10.5 把一所大学对应的一个顶点分割为两个顶点，把这两点都与进入该校的所有学生连接。这便是一个常规的双向匹配问题。

10.12 解决这个问题最简单的方法就是归约。按以下方法构造图 H：由 G 的每个顶点 w 分出两个顶点 w_1 和 w_2，称 w_1 是顶点 w 的尾，w_2 是 w 的头，对 G 中的每条边(w, u)，添加代价为 0 的边(w_2, u_1)到 H，换句话说，使所有边从头顶点走到尾顶点；另外，对 G 的每个顶点 u，添加代价为 $c(u)$的边(u_1, u_2)到 H。现在问题就变成常规的单起点最短路径问题，即从 v_2 到所有

尾顶点的最短路径问题。

10.15 最简单的反例只需一个变量 x，而欲最大化的目标函数就是 x，仅有的两个约束是 $x \leqslant 0$ 和 $x \neq 0$。该问题无解，因为没有比 0 小的最大化数。

10.17 可以简单地更改一下线性规划，但易观察最佳的解决方法是每个组织都捐助其最大值。

10.20 计算一个矩阵的平方，的确只需要它本身一半大小的5 个矩阵相乘，但是这样就变成了两个任意矩阵的乘法，而非其一半大小矩阵的平方。

10.24 这里说明如何用这样一个算法来排序。给定一个序列 $x_1, x_2, \cdots, x_n$ 互不相同，要对其进行排序。把每个点 p_i 关联到 x_i，使所有点都在一条固定直线上，从一个固定点到点 p_i 的距离是 x_i。易见最小生成树必须在线上连接每个点与它的邻接点，即树是一个链。树有两片叶子，是序列的最大和最小元素，如果知道最小生成树，则可以这样在线性时间内找到点的排序：首先，在点中找到最小值，如 x_i，由树的链给出排好的顺序，可以在线性时间内得到。这样在决策树模型（证明排序问题下界的模型）下就有该时间的下界 $\Omega(n\log n)$。

第 11 章

11.3 $O(n^k)$对 n 是一个多项式函数，但对 k 它是一个指数函数。因为 k 是输入的一部分（可能和 n 一样大），故这不是一个多项式时间算法。

11.11 从常规的顶点覆盖问题进行归约。设 $G=(V, E)$为一个任意无向图，并设 U 为 G 中度为奇的顶点的集合。修改 G，在 G 上添加三个新顶点 x、y 和 z，它们相互连接（在一个三角形上）。现在同时连接 x 和 U 中的所有顶点，易证修改后的图有着大小为 K 的顶点覆盖当且仅当 G 有大小为 K–2 的顶点覆盖。

11.14 这个问题明显属于 NP，因为可以猜测它的子集，并在多项式时间内检查它所诱导的子图。从 3SAT 进行归约。设 $E=C_1 \cdot C_2 \cdots C_n$ 为 3SAT 的任意一个实例。按如下方式构建一个有 $4n+1$ 个顶点的图 $G=(V, E)$。对每一个子句 C_i，其包含四个顶点，一个与子句本身关联，其他三个与对应的变量关联。这四个与一个子句关联的顶点完全互连。对任意变量 x，同时连接图中关联到 x 和 $\bar{x}$ 的顶点。最后，添加一个额外的顶点 r，它与所有关联到变量的顶点相连（但不和关联到子句本身的顶点相连）。可以断言：这个图有大小为 $2n+1$ 的可归约到无环图的子集，当且仅当表达式可满足。

1. 如果表达式可满足，那么可以找到一个满足每个子句的恒真指派。子集将由顶点 r、对应于子句的 n 个顶点、每个子句中对应于使子句为真的变量的那个顶点组成。这个子集可诱导出一个无环图，因为 r 仅连接变量，并且当两个变量属于一个恒真指派时它们不连接。
2. 如果存在一个有着 $2n+1$ 个顶点的可归约到无环图的子集 S，那么可以断言 S 包含 r 和每一个子句中的恰好两个顶点。的确，S 不可能包含一个子句中多于两个的顶点，因为这样的顶点是相互连接的（因此形成一个三角形）。同时，由于 S 包含 r，所以对某变量 x，S 不可能包含对应 x 和 $\bar{x}$ 的两个顶点。因此，获得一个 E 的真值指派是可能的。

11.16 从 3SAT 进行归约。设 $C=(x+y+z)$是某任意 3SAT 问题中的一个子句。用下面三个子句替换 C（a_i 是所有新的变量）：$(x+a_1+a_2)$、$(y+a_3+a_4)$、$(z+a_5+a_6)$。在这个 1-in-3SAT 问题中，上述三个子句中的每一个恰好有一个变量必然被满足。要想保证 x、y 和 z 中至少一个被满足，可以进行如下操作：多增加一些子句以保证 a_1、a_3 和 a_5 中不多于一个被满足，同时，a_2、a_4 和 a_6 中不多于一个被满足。这些子句是：$(a_1+a_3+a_7)$、$(a_3+a_5+a_8)$、$(a_5+a_1+a_9)$、$(a_2+a_4+a_{10})$、

$(a_4+a_6+a_{11})$和$(a_6+a_2+a_{12})$。验证略。

11.19 将团问题归约到这个问题。设 $G=(V,E)$和 k 为团问题的任意一个实例。我们需要将 G 转换成一个常规的图 R，使得对 G 的团问题能够通过求解对 R 的团问题得到解决。不能简单地向 G 中添加边直到 G 变成正规的，因为这将潜在地增加 G 的团的大小。我们必须添加顶点和边到 G 中，使得没有新的团形成。如果 G 中最大的度为偶数，则设 d 为 G 中最大的度，否则设 d 为 G 中最大的度加 1，又设 n 为 G 中的顶点数。对 G 中每一个度 $d(v)$小于 d 的顶点 v，增加 $d-d(v)$个顶点，并将它们每一个与 v 连接。额外顶点的总数为 $dn-\sum_{i=1}^{n}d(v_i)=dn-2|E|$。因为所选择的 d 为偶数，所以这个值是偶数。注意现在所有原有的顶点有着相同的度 d，并且未增加新的团（因为每一个新顶点仅连向一个顶点）。惟一的问题是新顶点的度为 1。可以通过如下方法在它们之间添加边，使其度变为 d 而不引入新的团。接着将新顶点划分为两个相等的集合。然后连接一个集合中的每个顶点到另一个集合中的 $d-1$ 个顶点。新顶点因此归约到一个双向图，这个双向图不包含大于 2 的团。按期望的方法构造这个双向图是可能的，留给读者验证。

11.25 如果能够确定是否存在一个有着特定端点的哈密尔顿路径，那么可以通过尝试所有顶点对来确定是否存在一条哈密尔顿路径。因此，这个问题的一个多项式时间算法导致哈密尔顿路径问题的多项式时间算法，从而导致对所有 NP 问题的一个多项式时间算法。注意：这不是 11.2 节定义的纯粹归约，但它足够了。

11.29 从顶点覆盖问题归约。设 $G=(V,E)$和 K 是顶点覆盖问题的任意一个实例。通过替换 G 中的每个顶点为两个顶点 v_1 和 v_2（v_1 和 v_2 通过有向边(v_1,v_2)连接），构造出一个有向图 G'。替换 G 中的每条边(v,w)为两条有向边(w_2,v_1)和(v_2,w_1)。现在可断言 G' 包含一个大小为 K 的反馈边集当且仅当 G 包含大小为 K 的顶点覆盖。

11.33 这里简要描述从 SAT 进行的一个归约。设 v 是 G 中的一个顶点，并设邻接 v 的边（按循环序）为 $e_1, e_2, \cdots, e_k$。欧拉行程定义了 v 的边的一个配对，使行程中连续进入和离开 v 的边被配对。根据问题的条件，边 e_i 同 e_{i-1} 或 e_{i+1} 配对（加法和减法均模 k）。可以观察到，如果 e_i 和 e_{i+1} 配对，那么 v 的所有其他边的配对也确定（e_{i+2} 必须同 e_{i+3} 配对，如此等等）。e_i 和 e_{i-1} 配对的情况也一样。换句话说，对 v 的边只有两种配对方法。将 SAT 表达式中的每一个变量关联上一个顶点，使得在真值指派中这个变量的值对应于在行程中这个顶点配对的方法。必须确保真值指派是相容的（也就是说，x 和 $\bar{x}$ 的值互补），并且每个子句被满足。现在对每个变量 x 有一个顶点 v_x，对 $\bar{x}$ 有顶点 $\overline{v_x}$，连接它们，使得从 v_x 的配对可强制得到 $\overline{v_x}$ 的配对，并依此赋予真值。然后，对每一个子句$(x+y+z)$，连接相应的三个顶点，使得至少有一个和取值为真的变量关联的配对必须被取到。详情（并不直接）留给读者完成。平面的情况更复杂，证明参见 Bent and Manber[1987]。

第 12 章

12.13 采用加倍法。第 1 步只有一个处理器 P_1 参与：P_1 简单地将 $A[1]$复制到 $A[2]$。在第 i 步，有 2^{i-1} 个参与的处理器，它们将 $A[1..2^{i-1}]$复制到 $A[2^{i-1}+1..2^i]$。为了提高效率，可使用 Brent 引理。

12.16 这个问题可以用与查找最大值完全相同的方法解决。

12.17 图 A.5 中给出了 $n=8$ 的线路图。

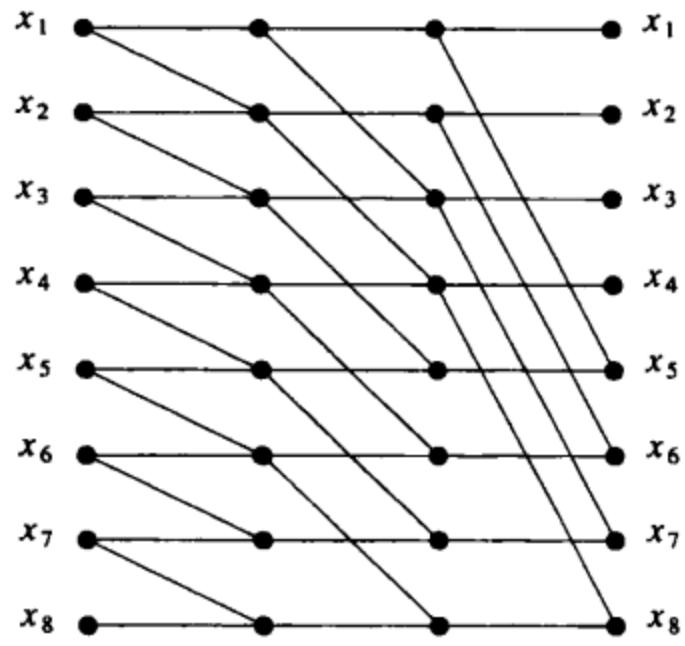


图 A.5　并行前缀电路

12.20 为每一对 $A[i]$和 $B[j]$分配一个处理器 P_{ij}，P_{ij}首先比较 $A[i]$和 $B[j]$，然后比较 $A[i]$和 $B[j+1]$。如果 $A[i]$在 $B[k]$和 $B[k+1]$间，那么 P_{ik}会发现这个，并据此推断 $A[i]$在最终数组的第$(i+k)$个位置。同样的程序可以应用于 B。

12.25 这里简单陈述一种利用并行前缀问题的较好的解决方案（这个问题也可被直接解决）。首先用“+”操作计算 *Mark* 数组的并行前缀。每一个 *Mark* 值为 1 的记录的前缀值将等于它在压缩后数组中的位置。然后使 *Mark* 数组完整以对其他记录完成同样的操作。一旦所有的索引都被计算到，实际的移动可以在一（并行）步之内完成，因为不存在冲突。

12.27 如果对每一条记录关联一个处理器，那么每一个处理器可以通过一步在该记录后继的适当域中写入其索引。现在每一个处理器可知它在链表中的前一个记录。

12.32 不可以独立考虑每一个数。确实每一个数最多增加一空闲步，但是，如果数不能足够快地到达，一个处理器的延时可能将导致其他处理器更多的延时。

12.33 使用归纳法。如果树的高度为 2，那么左叶子将它包含的数 x_1 传递给根，根再将其传递下来，最后被加到 x_2 上。假定存在对高为 h 的数的算法，考虑高为 $h+1$ 的树。（这里仅考虑完全二叉树的情况，但容易修改算法为对一般情况的处理。）设 R 为根，R_L 和 R_R 为根的左儿子和右儿子。我们称一特定子树上所有叶子的和为子树的和。一个直接的解决方案是对左子树和右子树分别求解此问题。然后将左子树的和广播到右子树的所有叶子上。右子树上的每一片叶子简单地将左子树的和加到它目前已计算出的前缀上。这种方法的问题在于它的运行时间是 $O(h^2)$，因为递归关系是 $T(h+1) = T(h)+h$。注意实际上没有必要等待左子树完成计算，由此可以改进解决方案。右子树需要获得左子树的和，这个和可以在 $h+1$ 步从根获得。因此，对 R_L 的需求是接收到所有它的后代的和并将其向上传送到根。对 R_R 的需求是从根接收和并向下传送给它的所有后代。这将直接导致下面的规则：（1）每一片叶子从将其值向上传送开始（必须稍微修改高为 2 的简单的解决方案，因为根需要知道和）；（2）当中间节点接收到从下面传来的值时，加上这些值并向上传送；（3）当中间节点接收到从上面传来的值时，将这些值向下传递给所有的儿子；（4）中间节点同时也在它自己的子树中起着根的作用，并将它从左儿子处接收到的值传给右儿子。读者可自行验证这个算法的正确性，算法的运行时间为 $2h$。

12.37 假定我们首先只对计算 x_1 感兴趣。处理器 P_2 计算它对 x_1 的值的贡献 $a_{12} \cdot b_2$，并将其传递给

P_3；处理器 P_3 在接收到的值上加上 $a_{12} \cdot b_3$，并将其向前传给 P_4，如此等等。在 $n-1$ 步后，P_1 将从 P_n 接收值 $x_1-a_{11} \cdot b_1$，并完成计算。现在，为了计算所有的 x_i，需要将这个过程管道化。在第 1 步，每个处理器 P_i 计算 $a_{(i-1)i} \cdot b_i$（所有索引运算都是模 n 的），并且将其传递至 P_{i+1}。在第 j 步，P_i 接受值 $\sum_{k=1}^{i-1} a_{(i-j)(i-k)} \cdot b_{i-k}$，并在其上加 $a_{(i-j)i} \cdot b_i$，并将结果传递至 P_{i+1}（同样，所有索引运算都是模 n 的）。

12.40 这个问题可以直接求解，但对矩阵乘法使用归纳法会更简单。首先使用 σ 构造一个排列矩阵 S，使得对任意列 i，仅位于 $\sigma(i)$行条目的值为 1，该列的所有其他条目的值为 0。这个矩阵容易构造，并可在 n 步内将其分配给合适的处理器。按照 σ 排列所有的行与计算 $S \cdot A$ 的积是一样的，而后者已知可在 $O(n)$步内完成。对列的排列可以使用与排列矩阵 Q 相同的方法进行，但此时积是 $A \cdot Q$。

12.42 直接的归纳方法是并行地对每一个半立方体计算并行前缀，然后将其中的一个半立方体（不妨指定为较小者）中最大的前缀广播到另一个中（对一维的基本情况是平凡的）。在一个 d 维立方体中广播需要 d 步。因此，这个算法的运行时间满足下面的递推关系：$T(d+1) = T(d)+1$，这蕴涵着 $T(d) = O(d_2)$。我们可以通过加强归纳假设来改进算法。假定每一个处理器不仅计算它对应的前缀，还要计算立方体中所有数的和。对一维的基础情况仍然是简单的：两个处理器简单地交换它们的值。给定一个 $d+1$ 维立方体，我们将其划分为两个 d 维立方体，并对它们用归纳法求解。但是现在，我们不需要广播左边立方体的和。因为左边立方体的每一个顶点都知道和，因此一步就能将和传递给右边立方体中的邻居。所有的前缀因此能够多用一步而被计算。此外，如果右边子树上的节点将它们（全局的）的和传送给它们左边子树上的邻居，那么所有的和就能多用一步而被计算。因此递推关系是：$T(d+1) = T(d)+2$，这蕴涵着 $T(d) = O(d)$。通过以上描述，将索引分配给处理器的过程应该就很清楚。

12.48 因为它涉及到计算机网络，对于闲聊（gossip）这个主题，有着丰富的文献，可参见 Hedetniemi, Hedetniemi, and Liestman[1988]。本习题陈述了一个相对简单的闲聊问题。它可以通过图 A.6 所描述的名为蝴蝶的并行计算模型来解决，当然该模型还有许多其他用途（最有名的是用于 FFT 的并行计算）。图 A.6 给出了 $n = 8$ 时的解。

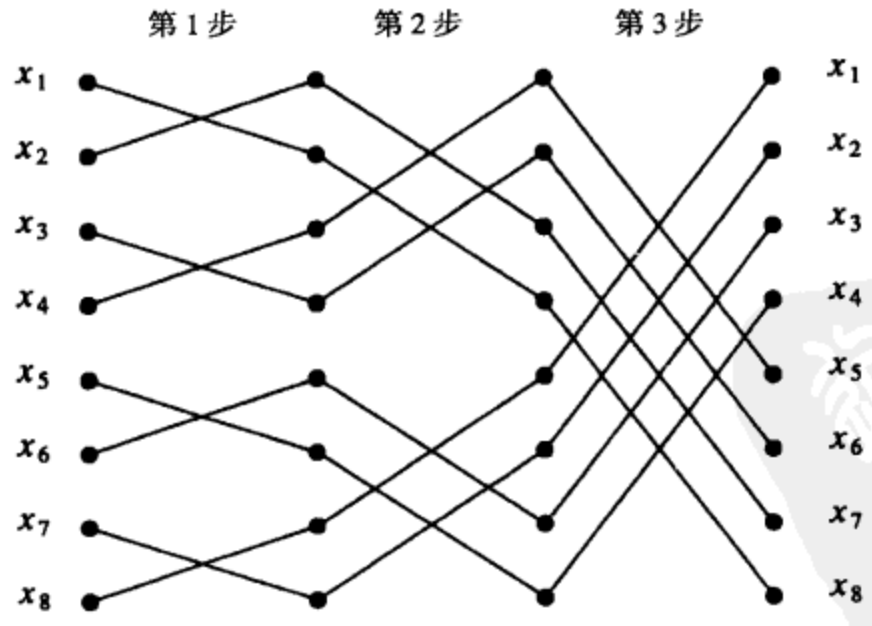


图 A.6　$n = 8$ 时的闲聊问题的解

参 考 文 献

Adel'son-Vel'skii G. M., and Y. M. Landis, "An algorithm for the organization of information," *Soviet Math. Dokl.*, **3** (1962), pp. 1259–1262.

Aho A. V., and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, **18** (June 1975), pp. 333–340.

Aho A. V., J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

Aho A. V., J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

Ajtai M., J. Komlós, and E. Szemerédi, "An $O(n \log n)$ sorting network," *15th Annual ACM Symposium on Theory of Computing*, Boston (April 1983), pp. 1–9.

Akl S. G., *Parallel Sorting Algorithms*, Academic Press, New York, 1985.

Aleliunas R., "Randomized parallel communication," *First ACM Symposium on Principles of Distributed Computing*, Ottawa (August 1982), pp. 60–72.

Almasi G. S., and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, CA, 1989.

Angluin D., and L. G. Valiant, "Fast probabilistic algorithms for Hamiltonian circuits and matchings," *Journal of Computer and System Sciences*, **18** (April 1979), pp. 155–193.

Apostolico A., and Z. Galil, *Combinatorial Algorithms on Words*, Springer-Verlag, New York, 1985.

Arlazarov V. L., E. A. Dinic, M. A. Kronrod, and I. A. Faradzev, "On economical construction of the transitive closure of a directed graph," *Soviet Math Dokl*, **11** (May 1970), pp. 1209–1210.

Atkinson M. D., and N. Santoro, "A practical algorithm for Boolean matrix multiplication," *Information Processing Letters*, **29** (September 1988), pp. 37–38.

Attalah M. J., R. Cole, and M. T. Goodrich, "Cascading divide–and–conquer: A technique for designing parallel algorithms," *28th Annual Symposium on Foundations of Computer Science*, Los Angeles (October 1987), pp. 151–160.

Auslander L., and S. V. Parter, "On imbedding graphs in the plane," *J. Math. and Mech.*, **10** (May 1961), pp. 517–523.

Bach E., G. Miller, and J. Shallit, "Sums of divisors, perfect numbers and factoring," *SIAM Journal on Computing*, **15** (November 1986), pp. 1143–1154.

Baer J.-L., and B. Schwab, "A comparison of tree–balancing algorithms," *Communications of the ACM*, **20** (May 1977), pp. 322–330.

Bar-Yehuda R., and S. Even, "A linear time approximation algorithm for the weighted vertex cover problem," *Journal of Algorithms*, **2** (1981), pp. 198–203.

Batcher K. E., "Sorting networks and their applications," in *Proceedings AFIPS 32nd Spring Joint Computer Conference*, (1968), pp. 307–314.

Bates J. L., and R. L. Constable, "Proofs as programs," *ACM Transactions on Programming Languages and Systems*, **7** (January 1985), pp. 113–136.

Bavel Z., *Math Companion for Computer Science*, Reston Publishing Company, Reston, Virginia, 1982.

Beame P. W., S. A. Cook, and H. J. Hoover, "Log depth circuits for division and related problems," *SIAM Journal on Computing*, **15** (February 1986), pp. 994–1003.

Beckenbach E., and R. Bellman, *An Introduction to Inequalities*, New Mathematical Library, Random House, New York, 1961.

Bellman R. E., *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

Bent S. W., and J. John, "Finding the median requires 2*n* comparisons," *17th Annual ACM Symposium on Theory of Computing*, Providence, RI (May 1985), pp. 213–216.

Bent S. W., and U. Manber, "On non-intersecting Eulerian circuits," *Discrete Applied Mathematics*, **18** (1987), pp. 87–94.

Bentley J. L., *Programming Pearls*, Addison-Wesley, Reading, MA, 1986.

Bentley J. L., M. G. Faust, and F. P. Preparata, "Approximation algorithms for convex hulls," *Communication of the ACM*, **25** (January 1982), pp. 64–68.

Bentley J. L., D. Haken, and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *SIGACT News*, (Fall 1980), pp. 36–44.

Bentley J. L., and B. W. Kernighan, "Tools for printing indexes," *Electronic Publishing*, **1** (1988), pp. 3–17.

Bentley J. L., and T. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Transactions on Computers*, **C-28** (Sept. 1979), pp. 643–647.

Berge C., *The Theory of Graphs and Its Applications*, John Wiley and Sons, New York, 1962.

Berge C., *Graphs and Hypergraphs*, North Holland, London, 1973.

Bertsekas D. P., and J. N. Tsitsiklis, *Parallel and Distributed Computation, Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

Bitner J. R., and E. M. Reingold, "Backtrack programming techniques," *Communications of the ACM*, **18** (November 1975), pp. 651–656.

Blelloch G., "Scans as primitive parallel operations," *1987 International Conference on Parallel Processing*, (August 1987), pp. 355–362.

Blum M., R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *Journal of Computer and System Sciences*, **7** (1972), pp. 448–461.

Bollobás B., *Graph Theory: An Introductory Course*, Springer Verlag, New York, 1979.

Bollobás B., *Combinatorics*, Cambridge University Press, Cambridge, 1986.

Bondy J. A., and U. S. R. Murty, *Graph Theory with Applications*, Elsevier, New York, 1976.

Borodin A., "On relating time and space to size and depth," *SIAM Journal on Computing*, **6** (December 1977), pp. 733–744.

Borodin A., and J. E. Hopcroft, "Routing, merging and sorting on parallel models of computation," *Journal of Computer and System Sciences*, **30** (1985), pp. 130–145.

Borodin A., and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems*, Elsevier Computer Science Library, New York, 1975.

Boyer R. S., and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, **20** (October 1977), pp. 762–772.

Brélaz D., "New methods to color the vertices of a graph," *Communications of the ACM*, **22** (April 1979), pp. 251–256.

Brigham E. O., *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

Brualdi R. A., *Introductory Combinatorics*, North Holland, New York, 1977.

Burge W. H., *Recursive Programming Techniques*, Addison-Wesley, Reading, MA, 1975.

Bussey W. H., "Origin of mathematical induction," *American Mathematical Monthly*, **24** (1917), pp. 199–207.

Capobianco M., and J. C. Molluzzo, *Examples and Counterexamples in Graph Theory*, North-Holland, New York, 1978.

Carter J. L., and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, **18** (April 1979), pp. 143–154.

Chand D. R., and S. S. Kapur, "An algorithm for convex polytopes," *Journal of the ACM*, **17** (January 1970), pp. 78–86.

Chandy K. M., and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, MA, 1988.

Chartrand G., *Graphs as Mathematical Models*, Wadsworth International Group, Belmont, CA, 1977.

Chartrand G., and L. Lesniak, *Graphs & Digraphs*, Second Edition, Wadsworth & Brooks/Cole, Monterey, CA, 1986.

Cheriton D., and R. E. Tarjan, "Finding minimum spanning trees," *SIAM Journal on Computing*, **5** (December 1976), pp. 724–742.

Choueka Y., A.S. Fraenkel, S.T. Klein, and Y. Perl, "Huffman coding without bit-manipulation," *Proceedings of the Eighth Annual ACM-SIGIR Conference*, Montreal, Canada (1985), pp. 122–130.

Christofides N., *Graph Theory: An Algorithmic Approach*, Academic Press, London, 1975.

Christofides N., "Worst-case analysis of a new heuristic for the traveling salesman

problem,'' Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.

Cohen J., and M. Roth, ''On the implementation of Strassen's fast multiplication algorithm,'' *Acta Informatica*, **6** (1976), pp. 341–355.

Cole R., ''Slowing down sorting networks to obtain faster sorting algorithms,'' *25th Annual Symposium on Foundations of Computer Science*, Singer Island (October 1984), pp. 255–259.

Cole R., ''Parallel merge sort,'' *SIAM Journal on Computing*, **17** (August 1988), pp. 770–785.

Cole R., and U. Vishkin, ''Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms,'' *18th Annual ACM Symposium on Theory of Computing*, Berkeley (May 1986), pp. 206–219.

Cook S. A., ''The complexity of theorem proving proceudres,'' *Third Annual ACM Symposium on Theory of Computing*, New York (1971), pp. 151–158.

Cook S. A., ''An overview of computational complexity,'' *Communications of the ACM*, **26** (June 1983), pp. 400–408.

Cook S. A., and C. Dwork, ''Bounds on the time of parallel RAMs to compute simple functions,'' *14th Annual ACM Symposium on Theory of Computing*, San Francisco (May 1982), pp. 231–233.

Cooley J. M., and J. W. Tuckey, ''An algorithm for the machine calculation of complex Fourier series,'' *Math. Comp.*, **19** (1965), pp. 297–301.

Coppersmith D., and S. Winograd, ''Matrix multiplication via arithmetic progressions,'' *19th Annual ACM Symposium on Theory of Computing*, New York (May 1987), pp. 1–6.

Culberson J., ''The effects of updates in binary search trees,'' *17th Annual ACM Symposium on Theory of Computing*, Providence, RI (May 1985), pp. 205–212.

Dantzig G. B., ''On the shortest route through a newtwork,'' *Management Science*, **6** (1960), pp. 187-190.

Dantzig G. B., *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

Dekel E., D. Nassimi, and S. Sahni, ''Parallel matrix and graph algorithms,'' *SIAM Journal on Computing*, **10** (November 1981), pp. 657–675.

Denardo E. V., *Dynamic Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

Deo N., *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

Deo N., and C. Pang, ''Shortest-path algorithms: Taxonomy and annotation,'' *Networks*, **14** (1984), pp. 275–323.

Dershowitz N., *The Evolution of Programs*, Birkhauser, Boston, 1983.

Diffie W., and M. E. Hellman, ''New directions in cryptography,'' *IEEE Transactions on Information Theory*, **IT-22** (June 1976), pp. 644–651.

Dijkstra E. W., "A note on two problems in connexion with graphs," *Numerische Mathematik*, **1** (1959), pp. 269–271.

Dijkstra E. W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

Dobkin D. P., and J. I. Munro, "Determining the mode," *Theoretical Computer Science*, **12** (1980), pp. 255–263.

Dreyfus S. E., and A. M. Law, *The Art and Theory of Dynamic Programming*, Academic Press, New York, 1977.

Dvorak, S., and B. Durian, "Unstable linear time $O(1)$ space merging," *The Computer Journal*, **31** (1988), pp. 279–283.

Ebert J., "Computing Eulerian trails," *Information Processing Letters*, **28** (June 1988), pp. 93–97.

Edelsbrunner H., *Algorithms in Combinatorial Geometry*, Springer-verlag, Berlin, 1987.

Edmonds J., and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the ACM*, **19** (1972), pp. 248–264.

Elliott D. F., and K. R. Rao, *Fast Transforms: Algorithms Analyses, Applications*, Academic Press, New York, 1982.

Eppinger J. L., "An empirical study of insertion and deletion in binary search trees," *Communications of the ACM*, **26** (September 1983), pp. 663–669.

Erdös P., and A. Szekers, "A combinatorial problem in geometry," *Compositio Mathematica*, **2** (1935), pp. 463–470.

Erdös P., and J. Spencer, *Probabilistic Methods in Combinatorics*. Academic Press, New York, 1974.

Euler L., "Solutio problematis ad geometriam situs pertinentis," *Commentarii Academiae Scientiarum Petropolitanae*, **8** (1736), pp. 128–140.

Even S., *Graph Algorithms*, Computer Science Press, Rockville, MD, 1979.

Even S., and R. E. Tarjan, "A combinatorial problem which is complete in polynomial space," *Journal of the ACM*, **23** (1976), pp. 710–719.

Fagin R., J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing — a fast access method for dynamic files," *ACM Transaction on Database Systems*, **4** (September 1979), pp. 315–355.

Feng T., "A survey of interconnection networks," *Computer*, ***14*** (December 1981), pp. 12–27.

Fich F. E., "New bounds for parallel prefix circuits," *15th Annual ACM Symposium on Theory of Computing*, Boston (April 1983), pp. 100–109.

Fischer M. J., "Efficiency of equivalence algorithms," in *Complexity and Computations*, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 153–168.

Fischer M. J., and A. R. Meyer, "Boolean matrix multiplication and transitive closure," *IEEE 12th Annual Symposium on Switching and Automata Theory*, East Lansing, MI (October 1971), pp. 129–131.

Fischer M. J., and M. O. Rabin, "Super-exponential complexity of Presburger

arithmetic,'' in *Complexity of Computation*, R. M. Karp Ed., SIAM–AMS, 1974.

Fischer M. J., and S. L. Salzberg, ''Finding a majority among *n* votes,'' *Journal of Algorithms*, **3** (1982), pp. 375–379.

Flajolet P., and J. S. Vitter, ''Average-case analysis of algorithms and data structures,'' Technical Report 718, INRIA, France, August 1987.

Floyd R. W., ''Algorithm 97: Shortest paths,'' *Communications of the ACM*, **5** (June 1962), pp. 345.

Floyd R. W., ''Assigning meanings to programs,'' *Symposium on Appllied Mathematics*, American Mathematical Society (1967), pp. 19–32.

Floyd R. W., and R. L. Rivest, ''Expected time bounds for selection,'' *Communication of the ACM*, **18** (March 1975), pp. 165–172.

Flynn M. J., ''Very high-speed computing systems,'' *Proceedings of the IEEE*, **54** (1966), pp. 1901–1909.

Ford L. R., ''Network flow theory,'' The Rand Corporation P-293, Santa Monica, CA (1956).

Ford L. R., and D. R. Fulkerson, ''Maximal flow through a network,'' *Canadian Journal of Mathematics*, **8** (1956), pp. 399–404.

Ford L. R., and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

Ford L. R., and S. M. Johnson, ''A tournament problem,'' *American Mathematical Monthly*, **66** (1959), pp. 387–389.

Fortune S., and J. Hopcroft, ''A note on Rabin's nearest-neighbor algorithm,'' *Information Processing Letters*, **8** (1979), pp. 20–23.

Fox G. C., M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors, Volume 1: General Techniques and Regular Problems*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

Franco J., ''On the probabilistic performance of algorithms for the satisfiability problem,'' *Information Processing Letters*, **23** (August 1986), pp. 103–106.

Frederickson G., ''Distributed algorithms for selection in sets,'' *Journal of Computer and System Sciences*, **37** (December 1988), pp. 337–348.

Fredman M. L., and R. E. Tarjan, ''Fibonacci heaps and and their uses in network optimization,'' *Journal of the ACM*, **34** (July 1987), pp. 596–615.

Gabow H. N., ''An efficient implementation of Edmonds's algorithm for maximum matching on graphs,'' *Journal of the ACM*, **23** (1976), pp. 221–234.

Gabow H. N., Z. Galil, T. H. Spencer, and R. E. Tarjan, ''Efficient algorithms for finding minimum spanning trees in undirected and directed graphs,'' *Combinatorica*, **6** (1986), pp. 109–122.

Gabow H. N., and R. E. Tarjan, ''Algorithms for two bottleneck optimization problems,'' *Journal of Algorithms*, **9** (September 1988), pp. 411–417.

Galil Z., ''On improving the worst case running time of the Boyer-Moore string searching algorithm,'' *Communications of the ACM*, **22** (September 1979), pp.

505–508.

Galil Z., "Efficient algorithms for finding maximum matching in graphs," *Computing Surveys*, **18** (March 1986), pp. 23–38.

Galler B. A., and M. J. Fishcer, "An improved equivalence algorithm," *Communications of the ACM*, **7** (1964), pp. 301–303.

Garey M. R., and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco, CA, 1979.

Gehringer E. F., D. P. Siewiorek, and Z. Segall, *Parallel Processing: The Cm* Experience*, Digital Press, Bedford, MA, 1987.

Gibbons A., *Algorithmic Graph Theory*, Cambridge University Press, Cambridge, 1985.

Gibbons A., and W. Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, 1988.

Goldberg A. V., and R. E. Tarjan, "A new approach to the maximal-flow problem," *Journal of the ACM*, **35** (October 1988), pp. 921–940.

Goldwasser S., and J. Killian, "Almost all primes can be quickly certified," *18th Annual ACM Symposium on Theory of Computing*, Berkeley (May 1986), pp. 316–329.

Golomb S., and L. Baumert, "Backtrack programming," *Journal of the ACM*, **12** (1965), pp. 516–524.

Golovina L. I., and I. M. Yaglom, *Induction in Geometry* (translated from Russian), D. C. Heath, Boston, 1963.

Golumbic M., *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

Gondran M., and M. Minoux, *Graphs and Algorithms*, John Wiley & Sons, New York, 1984.

Gonnet G. H., *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, MA, 1984.

Graham R. L., "An efficient algorithm for determining the convex hull of a planar set," *Information Processign Letters*, **1** (1972), pp. 132–133.

Graham R. L., D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, MA, 1989.

Gray F., *Pulse Code Communication*, US Patent 2632058 (March 1953).

Greenberg A., and U. Manber, "A probabilistic pipeline algorithm for k-selection on the tree machine," *IEEE Transactions on Computers*, **C-36** (March 1987), pp. 359-362.

Greene D. H., and D. E. Knuth, *Mathematics for the Analysis of Algorithms*, Birkhauser, Boston, 1982.

Gries D., *The Science of Programming*, Springer-Verlag, New York, 1981.

Grötschel M., L. Lovász, and A. Schrijver, "The ellipsoid method and its consequences in combinatorial optimization," *Combinatorica*, **1** (1981) pp. 169–197.

Guibas L. J., and A. M. Odlyzko, "A new proof of the linearity of the Boyer-Moore string searching algorithm," *SIAM Journal on Computing*, **9** (1980), pp. 672–682.

Gusfield D., and L. Pitt, "Equivalent approximation algorithms for node cover," *Information Processing Letters*, **22** (May 1986), pp. 291–294.

Guting R. H., and D. Wood, "Finding rectangles intersections by divide-and-conquer," *IEEE Transactions on Computers*, **C-33** (July 1984), pp. 771–775.

Hall M., *Combinatorial Theory*, Second Edition, John Wiley and Sons, New York, 1986.

Hall P., "On representatives of subsets," *Journal of the London Mathematical Society*, **10** (1935), pp. 26–30.

Hamming R. W., *Coding and Information Theory*, Second Edition, Prentice-Hall, Englewood Cliffs, NJ, 1986.

Harary F., *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

Hedetniemi S. T., S. M. Hedetniemi, and A. L. Liestman, "A survey of broadcasting and gossiping in communication networks," *Networks*, to appear (1989).

Hibbard T. N., "Some combinatorial properties of certain trees with applications to searching and sorting," *Journal of the ACM*, **9** (January 1962), pp. 13–28.

Hillis W. D., *The Connection Machine*, MIT Press, Cambridge, MA, 1985.

Hillis W. D., and G. L. Steele, "Data parallel algorithms," *Communications of the ACM*, **29** (December 1986), pp. 1170–1183.

Hinrichs K., J. Nievergelt, and P. Schorn, "Plane-sweep solves the closest pair problem elegantly," *Information Processing Letters*, **26** (1988), pp. 255–261.

Hirschberg D. S., "A linear-space algorithm for computing maximal common subsequences," *Communications of the ACM*, **18** (June 1975), pp. 341–343.

Hirschberg D. S., "Recent results on the complexity of common-subsequence problems," in *Time Wraps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, D. Sankoff and J. B. Kruskal, eds., Addison-Wesley, Reading, MA, 1983.

Hoare C. A. R., "Quicksort," *The Computer Journal*, (1962), pp. 10–15.

Hoffman C. M., *Group Theoretic Algorithms and Graph Isomorphism*, Lecture Notes in Computer Science, **136**, Springer-Verlag, New York, 1982.

Hofri M., *Probabilistic Analysis of Algorithms*, Springer-Verlag, New York, 1987.

Hong, J.-W., *Computation: Computability, Similarity and Duality*, Pitman, London, 1986.

Hopcroft J. E., and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing*, **2** (December 1973), pp. 225–231.

Hopcroft J. E., and R. E. Tarjan, "Dividing a graph into triconnected components," *SIAM Journal on Computing*, **2** (September 1973), pp. 135–158.

Hopcroft J. E., and R. E. Tarjan, "Efficient planarity testing," *Journal of the ACM*, **21** (October 1974), pp. 549–568.

Hopcroft J. E., and J. D. Ullman, "Set-merging algorithms," *SIAM Journal on Computing*, **2** (December 1973), pp. 294–303.

Horowitz E., and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD (1978).

Horowitz E., and A. Zorat "Divide and conquer for parallel processing," *IEEE Transactions on Computers*, **C-32** (June 1983), pp. 582–585.

Hu T. C., *Integer Programming and Network Flows*, Addison-Wesley, Reading, MA, 1969.

Huang B. C., and M. A. Langston, "Practical in-place merging," *Communications of the ACM*, **31** (March 1988), pp. 348–352.

Huffman D. A., "A method for the construction of minimum redundancy codes," *Proceedings of the IRE*, **40** (September 1952), pp. 1098–1101.

Hunt J. W., and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, **20** (May 1977), pp. 350–353.

Hwang K., and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

Incerpi J., and R. Sedgewick, "Practical variations of shellsort," *Information Processing Letters*, **26** (1987) pp. 37–43.

Itai A., and M. Rodeh, "Finding a minimum circuit in a graph," *SIAM Journal on Computing*, **7** (November 1978), pp. 413–423.

Jagadish H. V., and T. Kailath, "A family of new efficient arrays for matrix multiplication," *IEEE Transactions on Computers*, **C-38** (January 1989), pp. 149–155.

Johnson D. S., A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on Computing*, **3** (1974), pp. 299–325.

Johnson D. B., "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM*, **24** (January 1977), pp. 1–13.

Jones D. W., "An empirical comparison of priority–queue and event–set implementations," *Communications of the ACM*, **29** (April 1986), pp. 300–311.

Karatsuba A., and Yu. Ofman, "Multiplication of multidigit numbers on automata," translated in *Sov. Phys. Dokl.*, **7** (1963), pp. 595–596. Originally appeared in *Dokl. Akad. Nauk SSSR*, **145** (1962), pp. 293–294.

Karlin A. R., and E. Upfal, "Parallel hashing — An efficient implementation of shared memory," *18th Annual ACM Symposium on Theory of Computing*, Berkeley (May 1986), pp. 160–168.

Karmarkar N., "A new polynomial time algorithm for linear programming," *Combinatorica*, **4** (1984), pp. 373–395.

Karmarkar N., and R. M. Karp, "An efficient approximation scheme for the one-dimensional bin packing problem," *23th Annual Symposium on Foundations of Computer Science*, (November 1982), pp. 312–320.

Karp R. M., "Reducibilities among combinatorial problems," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York (1972), pp. 85–103.

Karp R. M., M. Saks, and A. Wigderson, "On a search problem related to branch-and-bound procedures," *27th Annual Symposium on Foundations of Computer Science*, Toronto (October 1986), pp. 19–28.

Karp R. M., "Combinatorics, complexity, and randomness," *Communications of the ACM*, **29** (February 1986), pp. 98–109.

Karp R. M., and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, **31** (March 1987), pp. 249–260.

Khachian L. G., "A polynomial algorithm in linear programming," *Soviet Math. Dokl.*, **20** (1979), pp. 191–194.

King K. N., and B. Smith-Thomas, "An optimal algorithm for sink-finding," *Information Processing Letters*, **14** (May 1982), pp. 109–111.

Klee V., and G. L. Minty, "How good is the simplex algorithm?" in *Inequalities III*, O. Shisha, ed., Academic Press, New York (1972), pp. 159-175.

Klee V., R. E. Ladner, and R. Manber, "Signsolvability revisited," *Linear Algebra and its Applications*, **59** (1984), pp. 131–157.

Knuth D. E., *The Art of Computer Programming, Volume 1/ Fundamental Algorithms*, Second edition, Addison-Wesley, Reading, MA, 1973a.

Knuth D. E., *The Art of Computer Programming, Volume 3/ Sorting and Searching*, Addison-Wesley, Reading, MA, 1973b.

Knuth D. E., "Estimating the efficiency of backtrack programs," *Mathematics of Computation*, **29** (1975), pp. 121–136.

Knuth D. E., "Big omicron and big omega and big theta," *SIGACT News*, (April–June 1976), pp. 18–24.

Knuth D. E., *The Art of Computer Programming, Volume 2/ Seminumerical Algorithms*, Second edition, Addison-Wesley, Reading, MA, 1981.

Knuth D. E., "Dynamic Huffman coding," *Journal of Algorithms*, **6** (1985), pp. 163–180.

Knuth D. E., J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, **6** (June 1977), pp. 323–350.

Kronrod, M. A., "An optimal ordering algorithm without a field of operation," (in Russian), *Dok. Akad. Nauk. SSSR*, **186** (1969), pp. 1256–1258.

Kruskal J. B., "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, **71** (1956), pp. 48–50.

Kruskal C. P., L. Rudolph, and M. Snir, "The power of parallel prefix," *IEEE Transactions on Computers*, **C-34** (November 1985), pp. 965–968.

Kruskal C. P., L. Rudolph, and M. Snir, "Efficient parallel algorithms for graph

problems,'' *1986 International Conference on Parallel Processing*, (August 1986), pp. 869–876.

Kruskal C. P., L. Rudolph, and M. Snir, ''Techniques for parallel manipulation of sparse matrices,'' IBM Research report RC-13364, (December 1987).

Kung H. T., ''Let's design algorithms for VLSI systems,'' *Proceedings of the Caltech Conference on VLSI*, (1979), pp. 65–90.

Kung H. T., ''Why systlic architecture,'' *Computer*, **15** (January 1982), pp. 37–46.

Kung H. T., and C. E. Leiserson, ''Algorithms for VLSI processor arrays,'' in *Introduction to VLSI Systems*, C. Mead and L. Conway, eds., Addison-Wesley, Reading, MA, (1980), pp. 271–292.

Kurtz T., and U. Manber, ''A probabilistic distributed algorithm for set intersection and its analysis,'' *Theoretical Computer Science*, **49** (1987), pp. 267–282.

Ladner R. E., ''On the structure of polynomial time reducibility,'' *Journal of the ACM*, **22** (January 1975), pp. 155–171.

Ladner R. E., and M. J. Fischer, ''Parallel prefix computation,'' *Journal of the ACM*, **27** (October 1980), pp. 831–838.

Lakatos I., *Proofs and Refutations: The Logic of Mathematical Discovery*, Cambridge University Press, Cambridge, 1976.

Lawler E. L., *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart & Winston, New York, 1976.

Lawler E. L., J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, *The Traveling Salesman Problem*, John Wiley & Sons, New York, 1985.

Lempel A., S. Even, and I. Cederbaum, ''An algorithm for planarity testing of graphs,'' *Theory of Graphs: An International Symposium*, Rome (July 1966), pp. 215–232.

Levin L. A., ''Universal sorting problems,'' *Problemy Peredaci Informacii*, **9** (1973), pp. 115–116. English translation in *Problems of Information Transmission*, **9** (1973), pp. 265–266.

Lin S., and B. W. Kernighan, ''An effective heuristic for the traveling salesman problem,'' *Operations Research*, **21** (1973), pp. 498–516.

Lipovski G. J., and M. Malek, *Parallel Computing: Theory and Comparisons*, John Wiley & Sons, New York, 1987.

Lipson J. D., *Elements of Algebra and Algebraic Computing*, Bejamin Cummings, Menlo Park, CA, 1981.

Lipton R. J., and D. Lopresti, ''A systolic array for rapid string comparison,'' *Proceedings of the 1985 Chapel Hill Conference on VLSI*, (1985), pp. 363–376.

Litwin W., ''Linear hashing: A new tool for file and table addressing,'' *Proceedings of the Sixth Conference on Very Large Databases*, Montreal, Canada (1980) pp. 212–223.

Lovász L., ''Coverings and colorings of hypergraphs,'' *Proceedings of the Fourth Southeastern Conference on Combinatorics, Graph Theory, and Computing*, Utilitas

Mathematica, Winnipeg (1973), pp. 3–12.

Lovász L., *Combinatorial Problems and Exercises*, North Holland, Amsterdam, 1979.

Lovász L., and M. D. Plummer, *Matching Theory*, North Holland, Amsterdam, 1986.

Lucas E., *Recreations Mathematiques*, Paris, 1882.

Lueker G. S., "Some techniques for solving recurrences," *Computing Surveys*, **12** (December 1980), pp. 419–436.

Luks E. M., "Isomorphism of graphs of bounded valence can be tested in polynomial time," *Journal of Computer and System Sciences*, **25** (1982), pp. 42–65.

Lynch T. J., *Data Compression Techniques and Applications*, Van Nostrand Reinhold, New York, 1985.

Maggs B. M., and S. A. Plotkin, "Minimum-cost spanning tree as a path-finding problem," *Information Processing Letters*, **26** (January 1988), pp. 291–293.

Manacher G. K., "The Ford–Johnson sorting algorithm is not optimal," *Journal of the ACM*, **26** (July 1979), pp. 441–456.

Manber U., "On maintaining dynamic information in a concurrent environment," *SIAM Journal on Computing*, **15** (November 1986), pp. 1130–1142.

Manber U., "Using induction to design algorithms," *Communications of the ACM*, **31** (November 1988), pp. 1300–1313.

Manber U., and L. McVoy, "Efficient storage of nonadaptive routing tables," *Networks*, **18** (1988), pp. 263–272.

Manber U., and E. Myers, "Suffix arrays: A new method for on-line string searches," *first Annual ACM-SIAM Symposium on Discrete Algorithms* (January 1990).

Manna Z., *Lectures on the Logic of Computer Programming*, CBMS–NSF Regional Conference series in Applied Mathematics, SIAM, Philadelphia, PA, 1980.

Manna Z., and R. Waldinger, "The origin of a binary search paradigm," *Science of Computer Programming*, **9** (1987) pp. 37–83.

Masek W. J., and M. S. Paterson, "How to compute string-edit distances quickly," in *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, D. Sankoff and J. B. Kruskal, eds., Addison-Wesley, Reading, MA (1983), pp. 337–349.

McCreight, E. M., "A space-economical suffix tree construction algorithm," *Journal of the ACM* **23** (1976), pp. 262–272.

Megiddo N., "Applying parallel computation algorithms in the design of serial algorithms," *Journal of the ACM*, **30** (October 1983) , pp. 852–865.

Mehlhorn K., *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

Menger K., "Zur allgeminen Kurventheorie," *Fund. Math.*, **10** (1927), pp. 95–115.

Meyer A. R., and L. Stockmeyer, "The equivalence problem for regular expression with squaring requires exponential space," *13th Annual Symposium on Switching and*

Automata Theory, (1972), pp. 125–129.

Minieka E., *Optimization Algorithms for Networks and Graphs*, Marcel Dekker, New York, 1978.

Mirzaian A., "A halving technique for the longest stuttering subsequence problem," *Information Processing Letters*, **26** (1987), pp. 71–75.

Misra J., and D. Gries, "Finding repeated elements," *Science of Computer Programming*, **2** (1982), pp. 143–152.

Moffat A., and T. Takaoka, "An all pair shortest path algorithm with expected time $O(n^2 \log n)$," *SIAM Journal on Computing*, **16** (December 1987), pp. 1023–1031.

Moitra A., and S. S. Iyengar, "Discussion of parallel algorithms," Technical Report TR-86-759, Department of Computer Science, Cornell University (June 1986).

Moret B. M. E., "Decision trees and diagrams," *Computing Surveys*, **14** (December 1982), pp. 593–623.

Munro I., "Problems related to matrix multiplication," in *Computational Complexity*, R. Rustin, ed., Algorithmics Press, New York, 1971.

Myers E. W., "An *O(ND)* difference algorithm and its variations," *Algorithmica*, **1** (1986), pp. 251–266.

Myers E. W., "A four-Russians algorithm for regular expression pattern matching," Technical Report #88-34, Department of Computer Science, University of Arizona, October 1988.

Nishizeki T., and N. Chiba, *Planar Graphs, Theory and Algorithms*, Annals of Discrete Mathematics, **32**, North Holland, Netherlands, 1988.

Ore O., "Note on Hamiltonian circuit," *American Mathematical Monthly*, **67** (1960), p. 55.

Ore O., *Graphs and Their Uses*, Random House, New York, 1963.

Pan V., "Strassen's algorithm is not optimal," *19th Annual Symposium on Foundations of Computer Science*, Ann Arbor, MI (October 1978), pp. 166–176.

Pan V., *How to Multiply Matrices Faster*, Lecture Notes in Computer Science, Volume 129, Springer-Verlag, Berlin, 1984.

Papadimitriou C. H., and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

Parberry I., *Parallel Complexity Theory*, John Wiley & Sons, New York, 1987.

Paull M. C., *Algorithm Design: A Recursion Transformation Framework*, John Wiley & Sons, New York, 1988.

Pearl J., *Heuristics — Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA, 1984.

Perl Y., A. Itai, and H. Avni, "Interpolation search — A $\log \log N$ search," *Communications of the ACM*, **21** (July 1978), pp. 550–553.

Polya G., *Induction and Analogy in Mathematics*, Princeton University Press, Princeton,

NJ, 1954.

Polya G., *How to Solve It*, Second Edition, Princeton University Press, Princeton, NJ, 1957.

Polya G., *Mathematical Discovery*, Combined Edition, John Wiley & Sons, New York, 1981.

Polya G., and G. Szego, *Aufgaben und Lehrsatze aus der Analysis*, Volume I, Berlin, Springer, 1927, p. 7.

Polya G., and G. Szego, *Problems and Theorems in Analysis I*, Springer-Verlag, Berlin, 1972.

Preparata F. P., and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

Prim R. C., "Shortest connection networks and some generalizations," *Bell System Technical Journal*, **36** (1957), p. 1389.

Purdom P. W., and C. A. Brown, *The Analysis of Algorithms*, Holt, Rinehart & Winston, New York, 1985a.

Purdom P. W., and C. A. Brown, "The pure literal rule and polynomial average time," *SIAM Journal on Computing*, **14** (November 1985b), pp. 943–953.

Quinn M. J., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, 1987.

Quinn M. J., and N. Deo, "Parallel graph algorithms," *Computing Surveys*, **16** (September 1984), pp. 319–348.

Rabin M. O., "Probabilistic algorithms," in *Algorithms and Complexity, Recent Results and New Directions*, J. F. Traub, ed., Academic Press, New York, 1976, pp. 21–39.

Raghavan P., "Probabilistic construction of deterministic algorithms: Approximating packing integer programs," *27th Annual Symposium on Foundations of Computer Science*, Toronto (October 1986), pp. 10–18.

Ranade A. G., "How to emulate shared memory," *28th Annual Symposium on Foundations of Computer Science*, Los Angeles (October 1987), pp. 185–194.

Reed D. A., and R. M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press, Cambridge, MA, 1987.

Reif J. H., and L. G. Valiant, "A logarithmic time sort for linear size networks," *Journal of the ACM*, **34** (January 1987), pp. 60–76.

Reingold E. M., and W. J. Hansen, *Data Structures*, Little, Brown, Boston, MA, 1983.

Richards D., "Parallel sorting — A bibliography," *SIGACT News*, **18** (Summer 1986).

Rivest R. L., A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, **21** (February 1978), pp. 120–126.

Roberts F. S., *Applied Combinatorics*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

Rodeh M., "Finding the median distributively," *Journal of Computer and System*

Sciences, **24** (1982), pp. 162–166.

Ronse C., "A bibliography on digital and computational convexity," Manuscript M185, Philips Research Laboratory, Brussels, (February 1987).

Rosenberg A. L., "On the time required to recognize properties of graphs," *SIGACT News*, (1973), pp. 15–16.

Runge C., and H. König, *Die Grundlehrn der mathematischen Wissenschften*, Springer, Berlin, 1924.

Ryser H. J., "Combinatorial properties of matrices of zeros and ones," *Canadian Journal of Mathematics*, **9** (1957), pp. 371–377.

Rytter W., "Fast recognition of pushdown automaton and context-free languages," *Information and Control*, **67** (1985) pp. 12–22.

Sankoff D., and J. B. Kruskal, *Time Wraps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, 1983.

Savitch W. J., "Relationships between nondeterministic and deterministic tape complexities," *Journal of Computer and System Sciences*, **4** (1970), pp. 177–192.

Schaback R., "On the expected sublinearity of the Boyer-Moore algorithm," *SIAM Journal on Computing*, **17** (August 1988), pp. 648–658.

Schönhage A., M. S. Paterson, and N. Pippenger, "Finding the median," *Journal of Computer and System Science*, **13** (October 1976), pp. 184–199.

Schrijver A., *Theory of Linear and Integer Programming*, John Wiley & Sons, Chichester, 1986.

Sedgewick R., *Quicksort*, Garland, New York, 1978.

Sedgewick R., *Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1988.

Sedgewick R., and J. S. Vitter, "Shortest paths in Euclidean graphs," *Algorithmica*, **1** (1986), pp. 31–48.

Shamos M. I., and D. Hoey, "Closest-point problems," *16th Annual Symposium on Foundations of Computer Science*, Berkeley (October 1975), pp. 151–162.

Shamos M. I., and D. Hoey, "Geometric intersection problems," *17th Annual Symposium on Foundations of Computer Science*, Houston (October 1976), pp. 208–215.

Shell D. L., "A high-speed sorting procedure" *Communications of the ACM*, **2** (July 1959), pp. 30–32.

Shiloach Y, and U. Vishkin, "Finding the maximum, merging, and sorting in a parallel computation model," *Journal of Algorithms*, **3** (March 1981), pp. 88–102.

Shrira L., N. Francez, and M. Rodeh, "Distributed k selection: From a sequential to a distributed algorithm," *Second ACM Symposium on Principles of Distributed Computing*, Ottawa (August 1983), pp. 143–153.

Siegel H. J., *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*, Lexington Books, Lexington, MA, 1985.

Sleator D. D., and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, **32** (July 1985), pp. 652–686.

Smit G. V., "A comparison of three string matching algorithms," *Software — Practice and Experience*, **12** (1982), pp. 57–66.

Solovay R., and V. Strassen, "A fast Monte-Carlo test for primality," *SIAM Journal on Computing*, **6** (March 1977), pp. 84–85; erratum (February 1978), p. 118.

Sominskii I. S., *The Method of Mathematical Induction*, (translated from Russian), D. C. Heath, Boston, 1963.

Spafford E. H., "The Internet worm program: An analysis," Technical Report CSD-TR-823, Department of Computer Science, Purdue University (November 1988).

Spira P. M., "A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$," *SIAM Journal on Computing*, **2** (1973), pp. 28-32.

Standish T. A., *Data Structure Techniques*, Addison-Wesley, Reading, MA., 1980.

Stanton D., and D. White, *Constructive Combinatorics*, Springer-Verlag, New York, 1986.

Stout Q. F., "Supporting divide-and-conquer algorithms for image processing," *Journal of Parallel and Distributed Computing*, **4** (1987), pp. 95–115.

Strassen V., "Gaussian elimination is not optimal," *Numerische Mathematik*, **13** (1969), pp. 354–356.

Takaoka T., "An on-line pattern matching algorithm," *Information Processing Letters*, **22** (May 1986), pp. 329–330.

Tarjan R. E., "Depth first search and linear graph algorithms," *SIAM Journal on Computing*, **1** (June 1972), pp. 146–160.

Tarjan R. E., "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, **22** (April 1975), pp. 215–225.

Tarjan R. E., *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.

Tarjan R. E., "Amortized computational complexity," *SIAM Journal on Applied and Discrete Mathematics*, **6** (1985), pp. 306–318.

Tarjan R. E., and J. van Leeuwen, "Worst case analysis of set union algorithms," *Journal of the ACM*, **31** (April 1984), pp. 245–281.

Tarjan R. E., and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM Journal on Computing*, **14** (November 1985), pp. 862–874.

Tarry G., "Le probleme des labyrinths," *Nouvelles Ann. de Math*, (1895), p. 187.

Toussaint G., *Computational Geometry*, North Holland, Amsterdam, 1984.

Turner J. S., "Almost all k-colorable graphs are easy to color," *Journal of Algorithms*, **9** (1988), pp. 63–82.

Tutte W. T., *Graph Theory*, Encyclopedia of Mathematics, Volume 21, Addison-Wesley, Reading, MA, 1984.

Uhr L., *Parallel Computer Vision*, Academic Press, New York, 1987.

Ukkonen E., "Algorithms for approximate string matching," *Information and Control*, **64** (1985), pp. 100–118.

Ullman J. D., *Computational Aspects of VLSI*, Computer Science Press, Rockville, MD, 1984.

Upfal E., "Efficient schemes for parallel communication," *First ACM Symposium on Principles of Distributed Computing*, Ottawa (August 1982), pp. 55–59.

Upfal E., and A. Wigderson, "How to share memory in a distributed system," *Journal of the ACM*, **34** (January 1987), pp. 116–127.

Vacca G., "Maurolycus, the first discoverer of the principle of mathematical induction," *Bulletin of the American Mathematical Society*, **16** (1909), p. 70.

Vaidya P. M., "Geometry helps in matching," *20th Annual ACM Symposium on Theory of Computing*, Chicago (May 1988), pp. 422–425.

Valiant L. G., "Parallelism in comparisons problems," *SIAM Journal on Computing*, **4** (September 1975), pp. 348–355.

Valiant L. G., "A scheme for fast parallel communication," *SIAM Journal on Computing*, **11** (May 1982), pp. 350–361.

Valiant L. G., and G. J. Brebner, "Universal schemes for parallel communication," *13th Annual ACM Symposium on Theory of Computing*, Milwaukee (May 1981), pp. 263–277.

van der Nat M., "On interpolation search," *Communications of the ACM*, **22** (December 1979), p. 681.

van Leeuwen J., "Graph algorithms," Technical Report RUU-CS-86-17, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, (October 1986).

von Neumann J., *Collected Works*, **5**, Macmillan, New York, 1963, pp. 91–99.

Vishkin U., "Randomized parallel speedups for list ranking," *Journal of Parallel and Distributed Computing*, **4** (1987), pp. 319–333.

Vitter J. S., "Design and analysis of dynamic Huffman coding," *26th Annual Symposium on Foundations of Computer Science*, Portland, OR (October 1985), pp. 293–302.

Vitter J. S., and W. C. Chen, *Design & Analysis of Coalesced Hashing*, Oxford University Press, New York, 1987.

von Neumann J., *Collected Works*, **5**, Macmillan, New York, 1963.

Wagner R. A., and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, **21** (January 1974), pp. 168–173.

Warren H. S., "A modification of Warshall's algorithm for transitive closure of binary relations," *Communications of the ACM*, **18** (April 1975), pp. 218–220.

Warshall S., "A theorem on Boolean matrices," *Journal of the ACM*, **9** (January 1962), pp. 11–12.

Wegman M. N., and J. L. Carter, "New classes and applications of hash functions," *20th*

Annual Symposium on Foundations of Computer Science, (October 1979), pp. 175–182.

Weiner, P., "Linear pattern matching algorithm," *14th IEEE Symposium on Switching and Automata Theory*, Iowa City, Iowa (October 1973), pp. 1–11.

Whitney H., "Congruent graphs and the connectivity of graphs," *American Journal of Mathematics*, **54** (1932), pp. 150–68.

Wigderson A., "Improving the performance guarantee for approximate graph coloring," *Journal of the ACM*, **30** (October 1983), pp. 729–735.

Wilf H. S., "Backtrack: An $O(1)$ expected time algorithm for the graph coloring problem," *Information Processing Letters*, **18** (1984), pp. 119–121.

Williams J. W. J., "Algorithm 232: Heapsort," *Communications of the ACM*, **7** (June 1964), p. 701.

Winograd S., "A new algorithm for inner product," *IEEE Transactions on Computers*, **C-17** (1968), pp. 693–694.

Winograd S., "On the number of multiplications necessary to compute certain functions," *Communications on Pure and Applied Mathematics*, **23** (1970a), pp. 165–170.

Winograd S., "The algebraic complexity of functions," *Proceedings of the International Congress of Mathematicians*, **3** (1970b), pp. 283–288.

Winograd S., "Some remarks on fast multiplication of polynomials," in *Complexity of Sequential and Parallel Numerical Algorithms*, J. F. Traub, ed., Academic Press, New York (1973), pp. 181–196.

Winograd S., *Arithmetic Complexity of Computations*, SIAM, Philadelphia, PA, 1980.

Wirth N., *Algorithms & Data Structures*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

Yao A., "An $O(|E|\log\log|V|)$ algorithm for finding minimum spanning trees," *Information Processing Letters*, **4** (1975), pp. 21–23.

Yao A., "Probabilistic computations — toward a unified measure of complexity," *18th Annual Symposium on Foundations of Computer Science*, Providence, RI (October 1977), pp. 222–227.

Ziv J., and A. Lempel, "Compression of indiviual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, **IT-24** (September 1978), pp. 530–536.

权威经典

算法引论

——一种创造性方法

Introduction to Algorithms
A Creative Approach

本书是国际算法大师乌迪·曼博（Udi Manber）博士撰写的一本享有盛誉的著作，强调了算法设计的创造性方面，通过算法开发步骤来描述算法设计过程。此外，本书创造性地将算法设计过程同定理归纳证明过程进行类比，揭示了算法设计的基本思想和本质，旨在提高读者的问题求解以及理解算法设计的过程和思想的能力。

本书特点

- 包括经典算法以及流行算法
- 算法设计技巧及其综合应用
- 并行算法设计
- 大多数算法的伪代码表示
- 500多道习题，其中四分之一给出了答案
- 将算法实现细节和算法思想尽可能分离

Udi Manber

美国著名的计算机科学家，国际公认的算法大师，在线信息搜索引擎的先驱。1982年于华盛顿大学获得计算机科学博士学位，曾是美国亚利桑那大学计算机专业教授。离开学校后在雅虎公司担任执行官，目前是亚马逊（Amazon.com）的副总裁和首席算法师（CAO），也是亚马逊旗下搜索网站A9.com的首席执行官。他提出的UDI测试已经成为衡量搜索引擎质量的评估标准。

PEARSON Education

PEARSON

www.PearsonEd.com

ISBN 978-7-121-09813-0

9 787121 098130 >

定价：36.00 元

责任编辑：冯小贝

责任美编：喻 晓

华信 欢迎登录 免费 获取优质教学资源 http://www.hxedu.com.cn

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。