HZ BOOKS
华章教育

计 算 机 科 学 丛 书

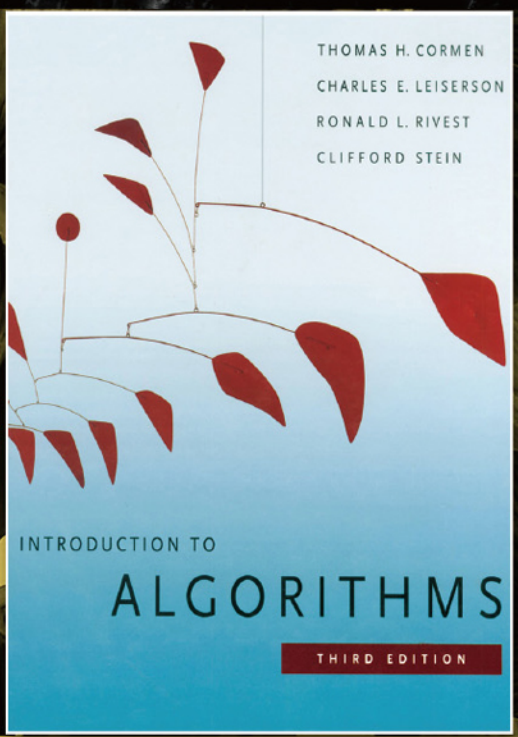
原书第3版

算法导论

(美) Thomas H. Cormen Charles E. Leiserson
Ronald L. Rivest Clifford Stein 著

殷建平 徐 云 王 刚 刘晓光 苏 明 邹恒明 王宏志 译

Introduction to Algorithms
Third Edition



机械工业出版社
China Machine Press

计算机科学丛书

算法导论

（原书第3版）

Introduction to Algorithms，Third Edition

（美）Thomas H. Cormen，Charles E. Leiserson，
Ronald L. Rivest，Clifford Stein 著

殷建平　徐云　王刚　刘晓光　苏明　邹恒明　王宏志 译

机械工业出版社
China Machine Press

本书提供了对当代计算机算法研究的一个全面、综合性的介绍。全书共八部分，内容涵盖基础知识、排序和顺序统计量、数据结构、高级设计和分析技术、高级数据结构、图算法、算法问题选编，以及数学基础知识。书中深入浅出地介绍了大量的算法及相关的数据结构，以及用于解决一些复杂计算问题的高级策略（如动态规划、贪心算法、摊还分析等），重点在于算法的分析与设计。对于每一个专题，作者都试图提供目前最新的研究成果及样例解答，并通过清晰的图示来说明算法的执行过程。此外，全书包含957道练习和158道思考题，并且作者在网站上给出了部分题的答案。

本书内容丰富，叙述深入浅出，适合作为计算机及相关专业本科生数据结构课程和研究生算法课程的教材，同时也适合专业技术人员参考使用。

本书版权登记号：图字：01-2009-5641

图书在版编目（CIP）数据

算法导论（原书第3版）/（美）科尔曼（Cormen，T. H.）等著；殷建平等译. —北京：机械工业出版社，2013.1
（计算机科学丛书）
书名原文：Introduction to Algorithms，Third Edition
ISBN 978-7-111-40701-0

Ⅰ. 算…　Ⅱ. ①科…　②殷…　Ⅲ. 电子计算机－算法理论　Ⅳ. TP301.6

中国版本图书馆CIP数据核字（2012）第290499号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：王春华
印刷
2013年1月第1版第1次印刷
185mm×260mm·49.75印张
标准书号：ISBN 978-7-111-40701-0
定价：128.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www. hzbook. com
电子邮件：hzjsj@hzbook. com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037

华章教育
华章科技图书出版中心

译者序

Introduction to Algorithms，Third Edition

我从1994年开始每年都为本科生讲授“算法设计与分析”课程，粗略地统计一下，发现至今已有5 000余名各类学生听过该课。算法的重要性不言而喻，因为不管新概念、新方法、新理论如何引人注目，信息的表示与处理总是计算技术（含软件、硬件、应用、网络、安全、智能等）永恒的主题。信息处理的核心是算法。在大数据时代，设计高效的算法显得格外重要。

当初，为了教好这门基础必修课，提高教学质量，我觉得应该从教学内容的改革入手，具体来说，采用的教材应该与国际一流大学接轨。1997年访美期间，在Stanford大学了解到他们采用的教材是Thomas H. Cormen等人著的《Introduction to Algorithms》，于是从Stanford书店买了一本带回来，从第二年开始便改用该书作为教材。至今，15年过去了，我们一直追随其变迁，从第2版到第3版。教学实践证明它确实是一本好教材，难怪世界范围内包括MIT、CMU、Stanford、UCB、Cornell、UIUC等国际、国内名校在内的1 000余所大学都一直用它作为教材或教学参考书，也难怪它印数巨大且在《高引用计算机科学文献》（《Most Cited Computer Science Citations》）一览表中名列前茅。

这本书的原版有1 200多页，内容非常丰富，不但涵盖了典型算法、算法分析、算法设计方法和NP完全等内容，而且还包括数据结构，甚至高级数据结构的介绍。后者可以作为国内“数据结构”课程的教材或教学参考资料。在学时有限的情况下，要在本科阶段教完前者的所有内容也是困难的，故要做取舍。好在该书的各个章节相对独立且难度由浅入深，我们的做法是将相对容易的、一般的入门性内容留在本科阶段，而将相对难的、专门的、较深入的内容并入研究生课程“算法及复杂性”或“计算复杂性”。除本校外，本人就曾多次应邀在兰州大学、湖南大学和浙江师范大学等院校为研究生讲授过这些内容。其实该书也适合希望增强自身程序设计能力和程序评判能力的广大应用计算技术的社会公众，特别是参加信息学奥林匹克竞赛和ACM程序设计竞赛的选手及其教练员。

在教学过程中，我们发现该书具有以下特点：1）选材与时俱进，具有实用性且能引起读者的兴趣。该书中研究的许多问题都是当前现实应用中的关键技术问题。2）采用伪代码描述算法，既简洁易懂又便于抓住本质，再配上丰富的插图来描述和解释算法的执行过程，使得教学内容更加通俗，便于自学。3）对算法正确性和复杂度的分析比较全面，既有严密的论证，又有直观的解释。4）既有结论性知识的介绍，也有逐步导出结论的研究过程的展示。5）丰富的练习题和思考题使得及时检验所学知识掌握情况和进一步拓展学习内容成为可能。

在第3版的《Introduction to Algorithms》出版后，我们应机械工业出版社编辑的邀请，启动了长久的翻译工程，先后参加翻译工作的老师有：国防科学技术大学的殷建平教授（翻译第1～3章）、中国科学技术大学的徐云教授（翻译第10～14章、第18～21章和第27章）、南开大学的王刚教授（翻译第4章和第15～17章）、南开大学的刘晓光教授（翻译第6～9章）、南开大学的苏明副研究员（翻译第5章和第28～30章）、上海交通大学的邹恒明教授（翻译第22～26章）、哈尔滨工业大学的王宏志副教授（翻译第31～35章和附录部分）。由于水平有限且工作量巨大，译文中一定存在许多不足，在此敬请各位同行专家学者和广大读者批评指正，欢迎大家将发现的错误或提出的意见与建议发送到邮箱：algorithms@hzbook.com。在整个工程即将完成之际，我们要特别感谢潘金贵、顾铁成、李成法和叶懋等参与本书第2版翻译的老师，是他们使得这本重要教材在国内有了广泛读者。同时也要感谢机械工业出版社的温莉芳编辑和王春华编辑，没有你们的信任、耐心和支持，整个翻译工作不可能完成。

殷建平

2012年11月于长沙

在计算机出现之前，就有了算法。现在有了计算机，就需要更多的算法，算法是计算的核心。

本书提供了对当代计算机算法研究的一个全面、综合的介绍。书中给出了多个算法，并对它们进行了较为深入的分析，使得这些算法的设计和分析易于被各个层次的读者所理解。我们力求在不牺牲分析的深度和数学严密性的前提下，给出深入浅出的说明。

书中每一章都给出了一个算法、一种算法设计技术、一个应用领域或一个相关的主题。算法是用英语和一种“伪代码”来描述的，任何有一点程序设计经验的人都能看得懂。书中给出了244幅图，说明各个算法的工作过程。我们强调将算法的效率作为一种设计标准，对书中的所有算法，都给出了关于其运行时间的详细分析。

本书主要供本科生和研究生的算法或数据结构课程使用。因为书中讨论了算法设计中的工程问题及其数学性质，所以，本书也可以供专业技术人员自学之用。

本书是第3版。在这个版本里，我们对全书进行了更新，包括新增了若干章、修订了伪代码等。

致使用本书的教师

本书的设计目标是全面、适用于多种用途。它可用于若干课程，从本科生的数据结构课程到研究生的算法课程。由于书中给出的内容比较多，只讲一学期一般讲不完，因此，教师们应该将本书看成是一种“缓存区”或“瑞典式自助餐”，从中挑选出能最好地支持自己希望教授的课程的内容。

教师们会发现，要围绕自己所需的各个章节来组织课程是比较容易的。书中的各章都是相对独立的，因此，你不必担心意想不到的或不必要的各章之间的依赖关系。每一章都是以节为单位，内容由易到难。如果将本书用于本科生的课程，可以选用每一章的前面几节内容；用于研究生的课程中，则可以完整地讲授每一章。

全书包含957道练习和158道思考题。每一节结束时给出练习，每一章结束时给出思考题。练习一般比较短，用于检查学生对书中内容的基本掌握情况。有一些是简单的自查性练习，有一些则要更充实，可以作为家庭作业布置给学生。每一章后的思考题都是一些叙述较为详细的实例研究，它们常常会介绍一些新的知识。一般来说，这些思考题都会包含几个小问题，引导学生逐步得到问题的解。

根据本书前几版的读者反馈，我们在本书配套网站上公布了其中一些练习和思考题的答案（但不是全部），网址为 http://mitpress.mit.edu/algorithms/。我们会定期更新这些答案，因此需要教师每次授课前都到这个网站上来查看。

在那些不太适合本科生、更适合研究生的章节和练习前面，都加上了星号（*）。带星号的章节也不一定就比不带星号的更难，但可能要求了解更多的数学知识。类似地，带星号的练习可能要求有更好的数学背景或创造力。

致使用本书的学生

希望本教材能为学生提供关于算法这一领域的有趣介绍。我们力求使书中给出的每一个算法都易于理解和有趣。为了在学生遇到不熟悉或比较困难的算法时提供帮助，我们逐个步骤地描述每一个算法。此外，为了便于大家理解书中对算法的分析，对于其中所需的数学知识，我们

给出了详细的解释。如果对某一主题已经有所了解，会发现根据书中各章的编排顺序，可以跳过一些介绍性的小节，直接阅读更高级的内容。

本书是一本大部头著作，学生所修的课程可能只讲授其中的一部分。我们试图使它能成为一本现在对学生有用的教材，并在其将来的职业生涯中，也能成为一本案头的数学参考书或工程实践手册。

阅读本书需要哪些预备知识呢？

- 需要有一些程序设计方面的经验，尤其需要理解递归过程和简单的数据结构，如数组和链表。
- 应该能较为熟练地利用数学归纳法进行证明。书中有一些内容要求学生具备初等微积分方面的知识。除此之外，本书的第一部分和第八部分将介绍需要用到的所有数学技巧。

我们收到学生的反馈，他们强烈希望提供练习和思考题的答案，为此，我们在 http://mitpress.mit.edu/algorithms/这个网站上给出了少数练习和思考题的答案，学生可以根据我们的答案来检验自己的解答。

致使用本书的专业技术人员

本书涉及的主题非常广泛，因而是一本很好的算法参考手册。因为每一章都是相对独立的，所以读者可以重点查阅自己感兴趣的主题。

在我们所讨论的算法中，多数都有着极大的实用价值。因此，我们在书中涉及了算法实现方面的考虑和其他工程方面的问题。对于那些为数不多的、主要具有理论研究价值的算法，通常还给出其实用的替代算法。

如果希望实现这些算法中的任何一个，你会发现将书中的伪代码翻译成你熟悉的某种程序设计语言是一件相当直接的事。伪代码被设计成能够清晰、简明地描述每一个算法。因此，我们不考虑错误处理和其他需要对读者所用编程环境有特定假设的软件工程问题。我们力求简单而直接地给出每一个算法，而不会让某种特定程序设计语言的特殊性掩盖算法的本质内容。

如果你是在课堂外使用本书，那么可能无法从教师那里得到答案来验证自己的解答，因此，我们在 http://mitpress.mit.edu/algorithms/这个网站上给出了部分练习和思考题的答案，读者可以免费下载参考。

致我们的同事

我们在本书中给出了详尽的参考文献。每一章在结束时都给出了“本章注记”，介绍一些历史性的细节和参考文献。但是，各章的注记并没有提供整个算法领域的全部参考文献。有一点可能是让人难以置信的，即使是在本书这样一本大部头中，由于篇幅的原因，很多有趣的算法都没能包括进来。

尽管学生们发来了大量的请求，希望我们提供思考题和练习的解答，但我们还是决定基本上不提供思考题和练习的参考答案（少数除外），以打消学生们试图查阅答案，而不是自己动手得出答案的念头。

第 3 版中所做的修改

在本书的第 2 版和第 3 版之间有哪些变化呢？这两版之间的变化量和第 2 版与第 1 版之间的变化量相当，正如在第 2 版的前言中所说，这些版本之间的变化可以说不太大，也可以说很大，具体要看读者怎么看待这些变化了。

快速地浏览一遍目录，你就会发现，第 2 版中的多数章节在第 3 版中都出现了。在第 3 版

中，去掉了两章和一节的内容，新增加了三章以及两节的内容。如果单从目录来判断第 3 版中改动的范围，得出的结论很可能是改动不大。

我们依然保持前两版的组织结构，既按照问题领域又根据技术来组织章节内容。书中既包含基于技术的章，如分治法、动态规划、贪心算法、摊还分析、NP 完全性和近似算法，也包含关于排序、动态集的数据结构和图问题算法的完整部分。我们发现虽然读者需要了解如何应用这些技术来设计和分析算法，但是思考题中很少提示应用哪个技术来解决这些问题。

下面总结了第 3 版的主要变化：

- 新增了讨论 van Emde Boas 树和多线程算法的章节，并且将矩阵基础移至附录。
- 修订了递归式那一章的内容，更广泛地覆盖分治法，并且前两节介绍了应用分治法解决两个问题。4.2 节介绍了用于矩阵乘法的 Strassen 算法，关于矩阵运算的内容已从本章移除。
- 移除两章很少讲授的内容：二项堆和排序网络。排序网络中的关键思想——0-1 原理，在本版的思考题 8-7 中作为比较交换算法的 0-1 排序引理进行介绍。斐波那契堆的处理不再依赖二项堆。
- 修订了动态规划和贪心算法相关内容。与第 2 版中的装配线调度问题相比，本版用一个更有趣的问题——钢条切割来引入动态规划。而且，我们比在第 2 版中更强调助记性，并且引入子问题图这一概念来阐释动态规划算法的运行时间。在我们给出的贪心算法例子（活动选择问题）中，我们以更直接的方式给出贪心算法。
- 我们从二叉搜索树（包括红黑树）删除一个结点的方式，现在保证实际所删除的结点就是请求删除的结点（在前两版中，有些情况下某个其他结点可能被删除）。用这种新的方式删除结点，如果程序的其他部分保持指针指向树中的结点，那么终止时就不会错误地将指针指向已删去的结点。
- 流网络相关材料现在基于边上的全部流。这种方法比前两版中使用的净流更直观。
- 由于关于矩阵基础和 Strassen 算法的材料移到了其他章，矩阵运算这一章的内容比第 2 版中所占的篇幅更小。
- 修改了对 Knuth-Morris-Pratt 字符串匹配算法的讨论。
- 修正了上一版中的一些错误。在网站上，这些错误大多数都已在第 2 版的勘误中给出，但是有些没有给出。
- 根据许多读者的要求，我们改变了书中伪代码的语法，现在用“=”表示赋值，用“==”表示检验相等，正如 C、C++、Java 和 Python 所用的。同样，我们不再使用关键字 **do** 和 **then** 而是使用“//”作为程序行末尾的注释符号。我们现在还使用点标记法表明对象属性。书中的伪代码仍是过程化的，而不是面向对象的。换句话说，我们只是简单地调用过程，将对象作为参数传递，而不是关于对象的运行方法。
- 新增 100 道练习和 28 道思考题，还更新并补充了参考文献。
- 最后，我们对书中的语句、段落和小节进行了一些调整，以使本书条理更清晰。

网站

读者可以通过 http://mitpress.mit.edu/algorithms/这个网站来获取补充资料，以及与我们联系。这个网站上给出了已知错误的清单、部分练习和思考题的答案等。此外，网站上还告诉读者如何报告错误或者提出建议。

第 3 版致谢

我们已经与 MIT Press 合作 20 多年，建立了很好的合作关系！感谢 Ellen Faran、Bob Prior、

Ada Brunstein 和 Mary Reilly 的帮助和支持。

在出版第 3 版时，我们在达特茅斯学院计算机科学系、MIT 计算机科学与人工智能实验室、哥伦比亚大学工业工程与运筹学系从事教学和科研工作。感谢这些学校和同事为我们提供的支持和实验环境。

Julie Sussman，P. P. A 担当本书第 3 版的技术编辑，再次拯救了我们。每次审阅，我们都觉得已经消除了错误，但是 Julie 还是发现了许多错误。她还帮我们改进了几处文字表述。如果有技术编辑名人堂，Julie 一定第一轮就可以入选。Julie 是非凡的，我们怎么感谢都是不够的。Priya Natarajan 也发现了一些错误，使得我们可以在将本书交给出版社前修正这些错误。书中的任何错误（毫无疑问，一定存在一些错误）都由作者负责（或许这些错误有些是 Julie 审阅材料后引入的）。

对于 van Emde Boas 树的处理出自于 Erik Demaine 的笔记，转而也受到 Michael Bender 的影响。此外，我还将 Javed Aslam、Bradley Kuszmaul 和 Hui Zha 的思想也整合到这一版。

多线程算法这一章是基于与 Harald Prokop 一起撰写的笔记，其他在 MIT 从事 Cilk 项目的同事也对本部分内容有所贡献，包括 Bradley Kuszmaul 和 Matteo Frigo。多线程伪代码的设计灵感来自 MIT Cilk 扩展到 C，以及由 Cilk Arts 的 Cilk++扩展到 C++。

我们还要感谢许多第 1 版和第 2 版的读者，他们报告了所发现的错误，或者提出了改进本书的建议。我们修正了全部报告来的真实错误，并且尽可能多地采纳了读者的建议。我们很高兴有这么多的人为本书做出贡献，但是很遗憾我们无法全部列出这些贡献者。

最后，非常感谢我们各自的妻子 Nicole Cormen、Wendy Leiserson、Gail Rivest 和 Rebecca Ivry，还有我们的孩子 Ricky、Will、Debby，Katie Leiserson、Alex，Christopher Rivest，以及 Molly、Noah 和 Benjamin Stein。感谢他们在我们写作本书过程中给予的爱和支持。正是由于有了来自家庭的耐心和鼓励，本书的写作工作才得以完成。谨将此书献给他们。

Thomas H. Cormen，新罕布什尔州黎巴嫩市
Charles E. Leiserson，马萨诸塞州剑桥市
Ronald L. Rivest，马萨诸塞州剑桥市
Clifford Stein，纽约州纽约市

第三部分　数据结构

第四部分　高级设计和分析技术

第五部分　高级数据结构

基础知识

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

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

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

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

第 4 章更深入地讨论了第 2 章引入的分治法，给出了更多分治法的例子，包括用于两方阵相乘的 Strassen 方法。第 4 章包含了求解递归式的方法。递归式用于描述递归算法的运行时间。“主方法”是一种功能很强的技术，通常用于解决分治算法中出现的递归式。虽然第 4 章中的相当一部分内容都是在证明主方法的正确性，但是如果跳过这一部分证明内容，也没有什么太大的影响。

第5章介绍概率分析和随机化算法。概率分析一般用于确定一些算法的运行时间，在这些算法中，由于同一规模的不同输入可能有着内在的概率分布，因而在这些不同输入之下，算法的运行时间可能有所不同。在有些情况下，我们假定算法的输入服从某种已知的概率分布，于是，算法的运行时间就是在所有可能的输入之下，运行时间的平均值。在其他情况下，概率分布不是来自于输入，而是来自于算法执行过程中所做出的随机选择。如果一个算法的行为不仅由其输入决定，还要由一个随机数生成器生成的值来决定，那么它就是一个随机化算法。我们可以利用随机化算法强行使算法的输入服从某种概率分布，从而确保不会有某一输入会始终导致算法的性能变坏；或者，对于那些允许产生不正确结果的算法，甚至能够将其错误率限制在某个范围之内。

附录A～D包含了一些数学知识，它们对读者阅读本书可能会有所帮助。在阅读本书之前，读者很有可能已经知道了附录中给出的大部分知识(我们采用的某些符号约定与读者过去见过的可能会有所不同)，因而可以将附录视为参考材料。另外，你很可能从未见过第一部分中给出的
4 内容。第一部分中的所有各章和附录都是以一种入门指南的风格来编写的。

算法在计算中的作用

什么是算法？为什么算法值得研究？相对于计算机中使用的其他技术来说算法的作用是什么？本章我们将回答这些问题。

1.1 算法

非形式地说，**算法**(algorithm)就是任何良定义的计算过程，该过程取某个值或值的集合作为**输入**并产生某个值或值的集合作为**输出**。这样算法就是把输入转换成输出的计算步骤的一个序列。

我们也可以把算法看成是用于求解良说明的**计算问题**的工具。一般来说，问题陈述说明了期望的输入/输出关系。算法则描述一个特定的计算过程来实现该输入/输出关系。

例如，我们可能需要把一个数列排成非递减序。实际上，这个问题经常出现，并且为引入许多标准的设计技术和分析工具提供了足够的理由。下面是我们关于**排序问题**的形式定义。

输入：n 个数的一个序列$\langle a_1, a_2, \cdots, a_n\rangle$。

输出：输入序列的一个排列$\langle a_1', a_2', \cdots, a_n'\rangle$，满足 $a_1' \leqslant a_2' \leqslant \cdots \leqslant a_n'$。

例如，给定输入序列〈31，41，59，26，41，58〉，排序算法将返回序列〈26，31，41，41，58，59〉作为输出。这样的输入序列称为排序问题的一个**实例**(instance)。一般来说，**问题实例**由计算该问题解所必需的(满足问题陈述中强加的各种约束的)输入组成。

因为许多程序使用排序作为一个中间步，所以排序是计算机科学中的一个基本操作。因此，已有许多好的排序算法供我们任意使用。对于给定应用，哪个算法最好依赖于以下因素：将被排序的项数、这些项已被稍微排序的程度、关于项值的可能限制、计算机的体系结构，以及将使用的存储设备的种类(主存、磁盘或者磁带)。

若对每个输入实例算法都以正确的输出停机，则称该算法是**正确的**，并称正确的算法**解决了**给定的计算问题。不正确的算法对某些输入实例可能根本不停机，也可能以不正确的回答停机。与人们期望的相反，不正确的算法只要其错误率可控有时可能是有用的。在第 31 章，当我们研究求大素数算法时，将看到一个具有可控错误率的算法例子。但是通常我们只关心正确的算法。

算法可以用英语说明，也可以说明成计算机程序，甚至说明成硬件设计。唯一的要求是这个说明必须精确描述所要遵循的计算过程。

算法解决哪种问题

排序绝不是已开发算法的唯一计算问题(当看到本书的厚度时，你可能觉得算法也同样多)。算法的实际应用无处不在，包括以下例子：

- 人类基因工程已经取得重大进展，其目标是识别人类 DNA 中的所有 10 万个基因，确定构成人类 DNA 的 30 亿个化学基对的序列，在数据库中存储这类信息并为数据分析开发工具。这些工作都需要复杂的算法。虽然对涉及的各种问题的求解超出了本书的范围，但是求解这些生物问题的许多方法采用了本书多章内容的思想，从而使得科学家能够有效地使用资源以完成任务。因为可以从实验技术中提取更多的信息，所以既能节省人和机器的时间又能节省金钱。
- 互联网使得全世界的人都能快速地访问与检索大量信息。借助于一些聪明的算法，互联网上的网站能够管理和处理这些海量数据。必须使用算法的问题示例包括为数据传输寻找好的路由(求解这些问题的技术在第 24 章给出)，使用一个搜索引擎来快速地找到特定

信息所在的网页(有关技术在第 11 章和第 32 章中)。

- 电子商务使得货物与服务能够以电子方式洽谈与交换，并且它依赖于像信用卡号、密码和银行结单这类个人信息的保密性。电子商务中使用的核心技术包括(第 31 章中包含的)公钥密码与数字签名，它们以数值算法和数论为基础。
- 制造业和其他商务企业常常需要按最有益的方式来分配稀有资源。一家石油公司也许希望知道在什么地方设置其油井，以便最大化其预期的利润。一位政治候选人也许想确定在什么地方花钱购买竞选广告，以便最大化赢得竞选的机会。一家航空公司也许希望按尽可能最廉价的方式把乘务员分配到班机上，以确保每个航班被覆盖并且满足政府有关乘务员调度的法规。一个互联网服务提供商也许希望确定在什么地方放置附加的资源，以便更有效地服务其顾客。所有这些都是可以用线性规划来求解的问题的例子，我们将在第 29 章学习这种技术。

虽然这些例子的一些细节已超出本书的范围，但是我们确实说明了一些适用于这些问题和问题领域的基本技术。我们还说明如何求解许多具体问题，包括以下问题：

- 给定一张交通图，上面标记了每对相邻十字路口之间的距离，我们希望确定从一个十字路口到另一个十字路口的最短道路。即使不允许穿过自身的道路，可能路线的数量也会很大。在所有可能路线中，我们如何选择哪一条是最短的？这里首先把交通图(它本身就是实际道路的一个模型)建模为一个图(第六部分和附录 B 将涉及这个概念)，然后寻找图中从一个顶点到另一个顶点的最短路径。第 24 章将介绍如何有效地求解这个问题。
- 给定两个有序的符号序列 $X=\langle x_1, x_2, \cdots, x_m\rangle$ 和 $Y=\langle y_1, y_2, \cdots, y_n\rangle$，求出 X 和 Y 的最长公共子序列。X 的子序列就是去掉一些元素(可能是所有，也可能一个没有)后的 X。例如，$\langle A, B, C, D, E, F, G\rangle$ 的一个子序列是 $\langle B, C, E, G\rangle$。$X$ 和 Y 的最长公共子序列的长度度量了这两个序列的相似程序。例如，若两个序列是 DNA 链中的基对，则当它们具有长的公共子序列时我们认为它们是相似的。若 X 有 m 个符号且 Y 有 n 个符号，则 X 和 Y 分别有 2^m 和 2^n 个可能的子序列。除非 m 和 n 很小，否则选择 X 和 Y 的所有可能子序列做匹配将花费使人望而却步多的时间。第 15 章将介绍如何使用一种称为动态规划的一般技术来有效地求解这个问题。

7

- 给定一个依据部件库的机械设计，其中每个部件可能包含其他部件的实例，我们需要依次列出这些部件，以使每个部件出现在使用它的任何部件之前。若该设计由 n 个部件组成，则存在 $n!$ 种可能的顺序，其中 $n!$ 表示阶乘函数。因为阶乘函数甚至比指数函数增长还快，(除非我们只有几个部件，否则)先生成每种可能的顺序再验证按该顺序每个部件出现在使用它的部件之前，是不可行的。这个问题是拓扑排序的一个实例，第 22 章将介绍如何有效地求解这个问题。
- 给定平面上的 n 个点，我们希望寻找这些点的凸壳。凸壳就是包含这些点的最小的凸多边形。直观上，我们可以把每个点看成由从一块木板钉出的一颗钉子来表示。凸壳则由一根拉紧的环绕所有钉子的橡皮筋来表示。如果橡皮筋因绕过某颗钉子而转弯，那么这颗钉子就是凸壳的一个顶点(例子参见图 33-6)。n 个点的 2^n 个子集中的任何一个都可能是凸壳的顶点集。仅知道哪些点是凸壳的顶点还很不够，因为我们还必须知道它们出现的顺序。所以为求凸壳的顶点，存在许多选择。第 33 章将给出两种用于求凸壳的好方法。

虽然这些问题的列表还远未穷尽(也许你已经再次从本书的重量推测到这一点)，但是它们却展示了许多有趣的算法问题所共有的两个特征：

1. 存在许多候选解，但绝大多数候选解都没有解决手头的问题。寻找一个真正的解或一个最好的解可能是一个很大的挑战。

2. 存在实际应用。在上面所列的问题中，最短路径问题提供了最易懂的例子。一家运输公司(如公路运输或铁路运输公司)对如何在公路或铁路网中找出最短路径，有着经济方面的利益，因为采用的路径越短，其人力和燃料的开销就越低。互联网上的一个路由结点为了快速地发送一条消息可能需要寻找通过网络的最短路径。希望从纽约开车去波士顿的人可能想从一个恰当的网站寻找开车方向，或者开车时她可能使用其 GPS。 8

算法解决的每个问题并不都有一个容易识别的候选解集。例如，假设给定一组表示信号样本的数值，我们想计算这些样本的离散傅里叶变换。离散傅里叶变换把时域转变为频域，产生一组数值系数，使得我们能够判定被采样信号中各种频率的强度。除了处于信号处理的中心之外，离散傅里叶变换还应用于数据压缩和大多项式与整数相乘。第 30 章为该问题给出了一个有效的算法——快速傅里叶变换(通常称为 FFT)，并且这章还概述了计算 FFT 的硬件电路的设计。

数据结构

本书也包含几种数据结构。**数据结构**是一种存储和组织数据的方式，旨在便于访问和修改。没有一种单一的数据结构对所有用途均有效，所以重要的是知道几种数据结构的优势和局限。

技术

虽然可以把本书当做一本有关算法的“菜谱”来使用，但是也许在某一天你会遇到一个问题，一时无法很快找到一个已有的算法来解决它(例如本书中的许多练习和思考题就是这样的情况)。本书将教你一些算法设计与分析的技术，以便你能自行设计算法、证明其正确性和理解其效率。不同的章介绍算法问题求解的不同方面。有些章处理特定的问题，例如，第 9 章的求中位数和顺序统计量，第 23 章的计算最小生成树，第 26 章的确定网络中的最大流。其他章介绍一些技术，例如第 4 章的分治策略，第 15 章的动态规划，第 17 章的摊还分析。

难题

本书大部分讨论有效算法。我们关于效率的一般量度是速度，即一个算法花多长时间产生结果。然而有一些问题，目前还不知道有效的解法。第 34 章研究这些问题的一个有趣的子集，其中的问题被称为 NP 完全的。

为什么 NP 完全问题有趣呢？第一，虽然迄今为止不曾找到对一个 NP 完全问题的有效算法，但是也没有人能证明 NP 完全问题确实不存在有效算法。换句话说，对于 NP 完全问题，是否存在有效算法是未知的。第二，NP 完全问题集具有一个非凡的性质：如果任何一个 NP 完全问题存在有效算法，那么所有 NP 完全问题都存在有效算法。NP 完全问题之间的这种关系使得有效解的缺乏更加诱人。第三，有几个 NP 完全问题类似于(但又不完全同于)一些有着已知有效算法的问题。计算机科学家迷恋于如何通过对问题陈述的一个小小的改变来很大地改变其已知最佳算法的效率。 9

你应该了解 NP 完全问题，因为有些 NP 完全问题会时不时地在实际应用中冒出来。如果要求你找出某一 NP 完全问题的有效算法，那么你可能花费许多时间在毫无结果的探寻中。如果你能证明这个问题是 NP 完全的，那么你可以把时间花在开发一个有效的算法，该算法给出一个好的解，但不一定是最好的可能解。

作为一个具体的例子，考虑一家具有一个中心仓库的投递公司。每天在中心仓库为每辆投递车装货并发送出去，以将货物投递到几个地址。每天结束时每辆货车必须最终回到仓库，以便准备好为第二天装货。为了减少成本，公司希望选择投递站的一个序，按此序产生每辆货车行驶的最短总距离。这个问题就是著名的“旅行商问题”，并且它是 NP 完全的。它没有已知的有效算法。然而，在某些假设条件下，我们知道一些有效算法，它们给出一个离最小可能解不太远的总

距离。第 35 章将讨论这样的“近似算法”。

并行性

我们或许可以指望处理器时钟速度能以某个持续的比率增加多年。然而物理的限制对不断提高的时钟速度给出了一个基本的路障：因为功率密度随时钟速度超线性地增加，一旦时钟速度变得足够快，芯片将有熔化的危险。所以，为了每秒执行更多计算，芯片被设计成包含不止一个而是几个处理“核”。我们可以把这些多核计算机比拟为在单一芯片上的几台顺序计算机；换句话说，它们是一类“并行计算机”。为了从多核计算机获得最佳的性能，设计算法时必须考虑并行性。第 27 章给出了充分利用多核的“多线程”算法的一个模型。从理论的角度来看，该模型具有
10 一些优点，它形成了几个成功的计算机程序的基础，包括一个国际象棋博弈程序。

练习

1.1-1 给出现实生活中需要排序的一个例子或者现实生活中需要计算凸壳的一个例子。

1.1-2 除速度外，在真实环境中还可能使用哪些其他有关效率的量度？

1.1-3 选择一种你以前已知的数据结构，并讨论其优势和局限。

1.1-4 前面给出的最短路径与旅行商问题有哪些相似之处？又有哪些不同？

1.1-5 提供一个现实生活的问题，其中只有最佳解才行。然后提供一个问题，其中近似最佳的一个解也足够好。

1.2 作为一种技术的算法

假设计算机是无限快的并且计算机存储器是免费的，你还有什么理由来研究算法吗？即使只是因为你还想证明你的解法会终止并以正确的答案终止，那么回答也是肯定的。

如果计算机无限快，那么用于求解某个问题的任何正确的方法都行。也许你希望你的实现在好的软件工程实践的范围内(例如，你的实现应该具有良好的设计与文档)，但是你最常使用的是最容易实现的方法。

当然，计算机也许是快的，但它们不是无限快。存储器也许是廉价的，但不是免费的。所以计算时间是一种有限资源，存储器中的空间也一样。你应该明智地使用这些资源，在时间或空间
11 方面有效的算法将帮助你这样使用资源。

效率

为求解相同问题而设计的不同算法在效率方面常常具有显著的差别。这些差别可能比由于硬件和软件造成的差别要重要得多。

作为一个例子，第 2 章将介绍两个用于排序的算法。第一个称为**插入排序**，为了排序 n 个项，该算法所花时间大致等于 $c_1 n^2$，其中 c_1 是一个不依赖于 n 的常数。也就是说，该算法所花时间大致与 n^2 成正比。第二个称为**归并排序**，为了排序 n 个项，该算法所花时间大致等于 $c_2 n \lg n$，其中 $\lg n$ 代表 $\log_2 n$ 且 c_2 是另一个不依赖于 n 的常数。与归并排序相比，插入排序通常具有一个较小的常数因子，所以 $c_1 < c_2$。我们将看到就运行时间来说，常数因子可能远没有对输入规模 n 的依赖性重要。把插入排序的运行时间写成 $c_1 n \cdot n$ 并把归并排序的运行时间写成 $c_2 n \cdot \lg n$。这时就运行时间来说，插入排序有一个因子 n 的地方归并排序有一个因子 $\lg n$，后者要小得多。(例如，当 $n=1\,000$ 时，$\lg n$ 大致为 10，当 n 等于 100 万时，$\lg n$ 大致仅为 20。)虽然对于小的输入规模，插入排序通常比归并排序要快，但是一旦输入规模 n 变得足够大，归并排序 $\lg n$ 对 n 的优点将足以补偿常数因子的差别。不管 c_1 比 c_2 小多少，总会存在一个交叉点，超出这个点，归并排序更快。

作为一个具体的例子，我们让运行插入排序的一台较快的计算机(计算机 A)与运行归并排序的一台较慢的计算机(计算机 B)竞争。每台计算机必须排序一个具有 1 000 万个数的数组。(虽然

1 000 万个数似乎很多，但是，如果这些数是 8 字节的整数，那么输入将占用大致 80MB，即使一台便宜的便携式计算机的存储器也能多次装入这么多数。)假设计算机 A 每秒执行百亿条指令(快于写本书时的任何单台串行计算机)，而计算机 B 每秒仅执行 1 000 万条指令，结果计算机 A 就纯计算能力来说比计算机 B 快 1 000 倍。为使差别更具戏剧性，假设世上最巧妙的程序员为计算机 A 用机器语言编码插入排序，并且为了排序 n 个数，结果代码需要 $2n^2$ 条指令。进一步假设仅由一位水平一般的程序员使用某种带有一个低效编译器的高级语言来实现归并排序，结果代码需要 $50n\lg n$ 条指令。为了排序 1 000 万个数，计算机 A 需要

$$\frac{2 \cdot (10^7)^2 \text{ 条指令}}{10^{10} \text{ 条指令 / 秒}} = 20\,000 \text{ 秒(多于 5.5 小时)}$$

而计算机 B 需要

12

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ 条指令}}{10^7 \text{ 条指令 / 秒}} \approx 1\,163 \text{ 秒(少于 20 分钟)}$$

通过使用一个运行时间增长较慢的算法，即使采用一个较差的编译器，计算机 B 比计算机 A 还快 17 倍！当我们排序 1 亿个数时，归并排序的优势甚至更明显：这时插入排序需要 23 天多，而归并排序不超过 4 小时。一般来说，随着问题规模的增大，归并排序的相对优势也会增大。

算法与其他技术

上面的例子表明我们应该像计算机硬件一样把算法看成是一种**技术**。整个系统的性能不但依赖于选择快速的硬件而且还依赖于选择有效的算法。正如其他计算机技术正在快速推进一样，算法也在快速发展。

你也许想知道相对其他先进的计算机技术(如以下列出的)，算法对于当代计算机是否真的那么重要：

- 先进的计算机体系结构与制造技术
- 易于使用、直观的图形用户界面(GUI)
- 面向对象的系统
- 集成的万维网技术
- 有线与无线网络的快速组网

回答是肯定的。虽然某些应用在应用层不明确需要算法内容(如某些简单的基于万维网的应用)，但是许多应用确实需要算法内容。例如，考虑一种基于万维网的服务，它确定如何从一个位置旅行到另一个位置。其实现依赖于快速的硬件、一个图形用户界面、广域网，还可能依赖于面向对象技术。然而，对某些操作，如寻找路线(可能使用最短路径算法)、描绘地图、插入地址，它还是需要算法。

而且，即使是那些在应用层不需要算法内容的应用也高度依赖于算法。该应用依赖于快速的硬件吗？硬件设计用到算法。该应用依赖于图形用户界面吗？任何图形用户界面的设计都依赖于算法。该应用依赖于网络吗？网络中的路由高度依赖于算法。该应用采用一种不同于机器代码的语言来书写吗？那么它被某个编译器、解释器或汇编器处理过，所有这些都广泛地使用算法。

13

算法是当代计算机中使用的大多数技术的核心。

进一步，随着计算机能力的不断增强，我们使用计算机来求解比以前更大的问题。正如我们在上面对插入排序与归并排序的比较中所看到的，正是在较大问题规模时，算法之间效率的差别才变得特别显著。

是否具有算法知识与技术的坚实基础是区分真正熟练的程序员与初学者的一个特征。使用现代计算技术，如果你对算法懂得不多，你也可以完成一些任务，但是，如果有一个好的算法背景，那么你可以做的事情就多得多。

练习

1.2-1　给出在应用层需要算法内容的应用的一个例子，并讨论涉及的算法的功能。

1.2-2 假设我们正比较插入排序与归并排序在相同机器上的实现。对规模为 n 的输入，插入排序运行 $8n^2$ 步，而归并排序运行 $64n\lg n$ 步。问对哪些 n 值，插入排序优于归并排序？

1.2-3 n 的最小值为何值时，运行时间为 $100n^2$ 的一个算法在相同机器上快于运行时间为 2^n 的另一个算法？

思考题

1-1 （运行时间的比较） 假设求解问题的算法需要 $f(n)$ 毫秒，对下表中的每个函数 $f(n)$ 和时间 t，确定可以在时间 t 内求解的问题的最大规模 n。

14

	1秒钟	1分钟	1小时	1天	1月	1年	1世纪
$\lg n$							
$\sqrt{n}$							
n							
$n\lg n$							
n^2							
n^3							
2^n							
$n!$							

本章注记

关于算法的一般主题存在许多优秀的教科书，包括由以下作者编写的那些：Aho、Hopcroft 和 Ullman[5，6]，Baase 和 Van Gelder[28]，Brassard 和 Bratley[54]，Dasgupta、Papadimitriou 和 Vazirani[82]，Goodrich 和 Tamassia[148]，Hofri[175]，Horowitz、Sahni 和 Rajasekaran[181]，Johnsonbaugh 和 Schaefer[193]，Kingston[205]，Kleinberg 和 Tardos[208]，Knuth[209，210，211]，Kozen[220]，Levitin[235]，Manber[242]，Mehlhorn[249，250，251]，Purdom 和 Brown[287]，Reingold、Nievergelt 和 Deo[293]，Sedgewick[306]，Sedgewick 和 Flajolet[307]，Skiena[318]，以及 Wilf[356]。Bentley[42，43]和 Gonnet[145]讨论了算法设计的一些更实际的方面。算法领域的全面评述也可以在《Handbook of Theoretical Computer Science，Volume A》[342]以及 CRC 出版的《Algorithms and Theory of Computation Handbook》[25]中找到。计算生物学中使用的算法的概述可以在由 Gusfield[156]、Pevzner[275]、Setubal 和 Meidanis[310]以及 Waterman[350]编写的教材中找到。

15

分治策略

在2.3.1节中，我们介绍了归并排序，它利用了分治策略。回忆一下，在分治策略中，我们递归地求解一个问题，在每层递归中应用如下三个步骤：

分解(Divide)步骤将问题划分为一些子问题，子问题的形式与原问题一样，只是规模更小。

解决(Conquer)步骤递归地求解出子问题。如果子问题的规模足够小，则停止递归，直接求解。

合并(Combine)步骤将子问题的解组合成原问题的解。

当子问题足够大，需要递归求解时，我们称之为**递归情况**(recursive case)。当子问题变得足够小，不再需要递归时，我们说递归已经“触底”，进入了**基本情况**(base case)。有时，除了与原问题形式完全一样的规模更小的子问题外，还需要求解与原问题不完全一样的子问题。我们将这些子问题的求解看做合并步骤的一部分。

在本章中，我们将看到更多基于分治策略的算法。第一个算法求解最大子数组问题，其输入是一个数值数组，算法需要确定具有最大和的连续子数组。然后我们将看到两个求解 $n \times n$ 矩阵乘法问题的分治算法。其中一个的运行时间为 $\Theta(n^3)$，并不优于平凡算法。但另一算法(Strassen算法)的运行时间为 $O(n^{2.81})$，渐近时间复杂性击败了平凡算法。

递归式

递归式与分治方法是紧密相关的，因为使用递归式可以很自然地刻画分治算法的运行时间。一个**递归式**(recurrence)就是一个等式或不等式，它通过更小的输入上的函数值来描述一个函数。例如，在2.3.2节中，我们用递归式描述了MERGE-SORT过程的最坏情况运行时间 $T(n)$：

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

求解可得 $T(n) = \Theta(n \lg n)$。

递归式可以有很多形式。例如，一个递归算法可能将问题划分为规模不等的子问题，如2/3对1/3的划分。如果分解和合并步骤都是线性时间的，这样的算法会产生递归式 $T(n) = T(2n/3) + T(n/3) + \Theta(n)$。

子问题的规模不必是原问题规模的一个固定比例。例如，线性查找的递归版本(练习2.1-3)仅生成一个子问题，其规模仅比原问题的规模少一个元素。每次递归调用将花费常量时间再加上下一层递归调用的时间，因此递归式为 $T(n) = T(n-1) + \Theta(1)$。

本章介绍三种求解递归式的方法，即得出算法的“Θ”或“O”渐近界的方法：

- **代入法** 我们猜测一个界，然后用数学归纳法证明这个界是正确的。
- **递归树法** 将递归式转换为一棵树，其结点表示不同层次的递归调用产生的代价。然后采用边界和技术来求解递归式。
- **主方法** 可求解形如下面公式的递归式的界：

  $$T(n) = aT(n/b) + f(n) \tag{4.2}$$

  其中 $a \geqslant 1$，$b > 1$，$f(n)$是一个给定的函数。这种形式的递归式很常见，它刻画了这样一个分治算法：生成 a 个子问题，每个子问题的规模是原问题规模的 $1/b$，分解和合并步骤总共花费时间为 $f(n)$。

为了使用主方法，必须要熟记三种情况，但是一旦你掌握了这种方法，确定很多简单递归式的渐近界就变得很容易。在本章中，我们将使用主方法来确定最大子数组问题和矩阵相乘问题的分治算法的运行时间，本书中其他使用分治策略的算法也将用主方法进行分析。

我们偶尔会遇到不是等式而是不等式的递归式，例如 $T(n) \leqslant 2T(n/2)+\Theta(n)$。因为这样一种递归式仅描述了 $T(n)$的一个上界，因此可以用大 O 符号而不是 Θ 符号来描述其解。类似地，如果不等式为 $T(n) \geqslant 2T(n/2)+\Theta(n)$，则由于递归式只给出了 $T(n)$的一个下界，我们应使用 Ω 符号来描述其解。

递归式技术细节

在实际应用中，我们会忽略递归式声明和求解的一些技术细节。例如，如果对 n 个元素调用 MERGE-SORT，当 n 为奇数时，两个子问题的规模分别为$\lfloor n/2 \rfloor$和$\lceil n/2 \rceil$，准确来说都不是 $n/2$，因为当 n 是奇数时，$n/2$ 不是一个整数。技术上，描述 MERGE-SORT 最坏情况运行时间的准确的递归式为

$$T(n) = \begin{cases} \Theta(1) & \text{若 } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{若 } n > 1 \end{cases} \tag{4.3}$$

边界条件是另一类我们通常忽略的细节。由于对于一个常量规模的输入，算法的运行时间为常量，因此对于足够小的 n，表示算法运行时间的递归式一般为 $T(n)=\Theta(1)$。因此，出于方便，我们一般忽略递归式的边界条件，假设对很小的 n，$T(n)$为常量。例如，递归式(4.1)常被表示为

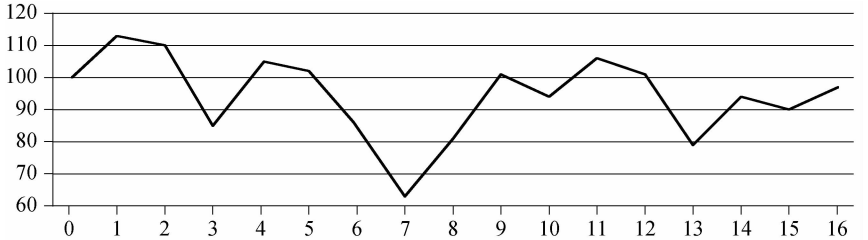
$$T(n) = 2T(n/2) + \Theta(n) \tag{4.4}$$

去掉了 n 很小时函数值的显式描述。原因在于，虽然改变 $T(1)$的值会改变递归式的精确解，但改变幅度不会超过一个常数因子，因而函数的增长阶不会变化。

当声明、求解递归式时，我们常常忽略向下取整、向上取整及边界条件。我们先忽略这些细节，稍后再确定这些细节对结果是否有较大影响。通常影响不大，但你需要知道什么时候会影响不大。这一方面可以依靠经验来判断，另一方面，一些定理也表明，对于很多刻画分治算法的递归式，这些细节不会影响其渐近界(参见定理 4.1)。但是，在本章中，我们会讨论某些细节，展
67 示递归式求解方法的要点。

4.1 最大子数组问题

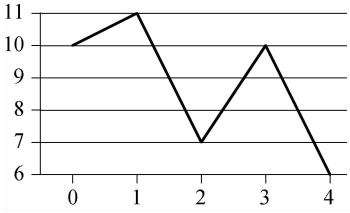
假定你获得了投资挥发性化学公司的机会。与其生产的化学制品一样，这家公司的股票价格也是不稳定的。你被准许可以在某个时刻买进一股该公司的股票，并在之后某个日期将其卖出，买进卖出都是在当天交易结束后进行。为了补偿这一限制，你可以了解股票将来的价格。你的目标是最大化收益。图 4-1 给出了 17 天内的股票价格。第 0 天的股票价格是每股 100 美元，你可以在此之后任何时间买进股票。你当然希望“低价买进，高价卖出”——在最低价格时买进股票，之后在最高价格时卖出，这样可以最大化收益。但遗憾的是，在一段给定时期内，可能无法做到在最低价格时买进股票，然后在最高价格时卖出。例如，在图 4-1 中，最低价格发生在第 7 天，而最高价格发生在第 1 天——最高价在前，最低价在后。



天	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
价格	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
变化		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

图 4-1 17 天内，每天交易结束后，挥发性化学公司的股票价格信息。横轴表示日期，纵轴表示股票价格。表格的最后一行给出了股票价格相对于前一天的变化

你可能认为可以在最低价格时买进，或在最高价格时卖出，即可最大化收益。例如，在图 4-1 中，我们可以在第 7 天股票价格最低时买入，即可最大化收益。如果这种策略总是有效的，则确定最大化收益是非常简单的：寻找最高和最低价格，然后从最高价格开始向左寻找之前的最低价格，从最低价格开始向右寻找之后的最高价格，取两对价格中差值最大者。但图 4-2 给出了一个简单的反例，显示有时最大收益既不是在最低价格时买进，也不是在最高价格时卖出。 68



天	0	1	2	3	4
价格	10	11	7	10	6
变化		1	−4	3	−4

图 4-2　本例说明最大收益并不一定从最低价格开始或者到最高价格结束。与图 4-1 一样，横轴表示日期，纵轴表示价格。在本例中，最大收益为每股 3 美元，第 2 天买进，第 3 天卖出可获得此最大收益。第 2 天的价格 7 美元并非最低价格，而第 3 天的价格 10 美元也并非最高价格

暴力求解方法

我们可以很容易地设计出一个暴力方法来求解本问题：简单地尝试每对可能的买进和卖出日期组合，只要卖出日期在买入日期之后即可。n 天中共有 $\binom{n}{2}$ 种日期组合。因为 $\binom{n}{2}=\Theta(n^2)$，而处理每对日期所花费的时间至少也是常量，因此，这种方法的运行时间为 $\Omega(n^2)$。有更好的方法吗？

问题变换

为了设计出一个运行时间为 $o(n^2)$ 的算法，我们将从一个稍微不同的角度来看待输入数据。我们的目的是寻找一段日期，使得从第一天到最后一天的股票价格净变值最大。因此，我们不再从每日价格的角度去看待输入数据，而是考察每日价格变化，第 i 天的价格变化定义为第 i 天和第 $i-1$ 天的价格差。图 4-1 中的表格的最后一行给出了每日价格变化。如果将这一行看做一个数组 A，如图 4-3 所示，那么问题就转化为寻找 A 的和最大的非空连续子数组。我们称这样的连续子数组为**最大子数组**(maximum subarray)。例如，对图 4-3 中的数组，$A[1..16]$的最大子数组为 $A[8..11]$，其和为 43。因此，你可以在第 8 天(7 天之后)买入股票，并在第 11 天后卖出，获得每股收益 43 美元。

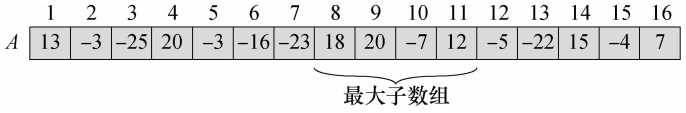


图 4-3　股票价格变化值的最大子数组问题。本例中，子数组 $A[8..11]$的和是 43，是 A 的所有连续子数组中和最大的

乍一看，这种变换对问题求解并没有什么帮助。对于一段 n 天的日期，我们仍然需要检查 $\binom{n-1}{2}=\Theta(n^2)$ 个子数组。练习 4.1-2 要求证明，虽然计算一个子数组之和所需的时间是线性 69 的，但当计算所有 $\Theta(n^2)$ 个子数组和时，我们可以重新组织计算方式，利用之前计算出的子数组和来计算当前子数组的和，使得每个子数组和的计算时间为 $O(1)$，从而暴力求解方法所花费的时间仍为 $\Theta(n^2)$。

接下来，我们寻找最大子数组问题的更高效的求解方法。在此过程中，我们通常说“一个最

大子数组”而不是“最大子数组”，因为可能有多个子数组达到最大和。

只有当数组中包含负数时，最大子数组问题才有意义。如果所有数组元素都是非负的，最大子数组问题没有任何难度，因为整个数组的和肯定是最大的。

使用分治策略的求解方法

我们来思考如何用分治技术来求解最大子数组问题。假定我们要寻找子数组 $A[low..high]$ 的最大子数组。使用分治技术意味着我们要将子数组划分为两个规模尽量相等的子数组。也就是说，找到子数组的中央位置，比如 mid，然后考虑求解两个子数组 $A[low..mid]$ 和 $A[mid+1..high]$。如图 4-4(a)所示，$A[low..high]$ 的任何连续子数组 $A[i..j]$ 所处的位置必然是以下三种情况之一：

- 完全位于子数组 $A[low..mid]$ 中，因此 $low \leqslant i \leqslant j \leqslant mid$。
- 完全位于子数组 $A[mid+1..high]$ 中，因此 $mid < i \leqslant j \leqslant high$。
- 跨越了中点，因此 $low \leqslant i \leqslant mid < j \leqslant high$。

因此，$A[low..high]$ 的一个最大子数组所处的位置必然是这三种情况之一。实际上，$A[low..high]$ 的一个最大子数组必然是完全位于 $A[low..mid]$ 中、完全位于 $A[mid+1..high]$ 中或者跨越中点的所有子数组中和最大者。我们可以递归地求解 $A[low..mid]$ 和 $A[mid+1..high]$ 的最大子数组，因为这两个子问题仍是最大子数组问题，只是规模更小。因此，剩下的全部工作就是寻找跨越中点的最大子数组，然后在三种情况中选取和最大者。

70

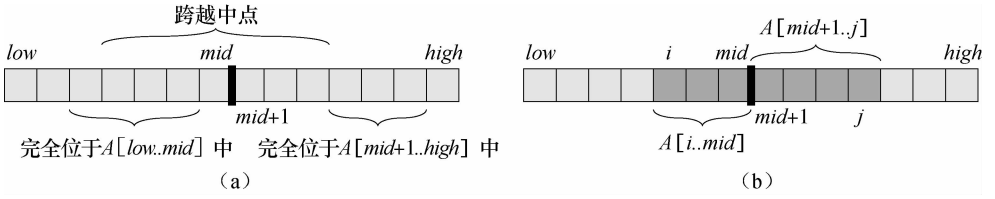


图 4-4 (a)$A[low..high]$ 的子数组的可能位置：完全位于 $A[low..mid]$ 中，完全位于 $A[mid+1..high]$ 中，或者跨越中点 mid。(b)$A[low..high]$ 的任何跨越中点的子数组由两个子数组 $A[i..mid]$ 和 $A[mid+1..j]$ 组成，其中 $low \leqslant i \leqslant mid$ 且 $mid < j \leqslant high$

我们可以很容易地在线性时间(相对于子数组 $A[low..high]$ 的规模)内求出跨越中点的最大子数组。此问题并非原问题规模更小的实例，因为它加入了限制——求出的子数组必须跨越中点。如图 4-4(b)所示，任何跨越中点的子数组都由两个子数组 $A[i..mid]$ 和 $A[mid+1..j]$ 组成，其中 $low \leqslant i \leqslant mid$ 且 $mid < j \leqslant high$。因此，我们只需找出形如 $A[i..mid]$ 和 $A[mid+1..j]$ 的最大子数组，然后将其合并即可。过程 FIND-MAX-CORSSING-SUBARRAY 接收数组 A 和下标 low、mid 和 $high$ 为输入，返回一个下标元组划定跨越中点的最大子数组的边界，并返回最大子数组中值的和。

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4      sum=sum+A[i]
 5      if sum > left-sum
 6          left-sum = sum
 7          max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
```

```
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

71

此过程的工作方式如下所述。第 1～7 行求出左半部 $A[low..mid]$的最大子数组。由于此子数组必须包含 $A[mid]$，第 3～7 行的 **for** 循环的循环变量 i 是从 mid 开始，递减直至达到 low，因此，它所考察的每个子数组都具有 $A[i..mid]$的形式。第 1～2 行初始化变量 *left-sum* 和 *sum*，前者保存目前为止找到的最大和，后者保存 $A[i..mid]$中所有值的和。每当第 5 行找到一个子数组 $A[i..mid]$的和大于 *left-sum* 时，我们在第 6 行将 *left-sum* 更新为这个子数组的和，并在第 7 行更新变量 *max-left* 来记录当前下标 i。第 8～14 求右半部 $A[mid+1..high]$的最大子数组，过程与左半部类似。此处，第 10～14 行的 **for** 循环的循环变量 j 是从 $mid+1$ 开始，递增直至达到 $high$，因此，它所考察的每个子数组都具有 $A[mid+1..j]$的形式。最后，第 15 行返回下标 *max-left* 和 *max-right*，划定跨越中点的最大子数组的边界，并返回子数组 $A[max\text{-}left..max\text{-}right]$的和 *left-sum*+*right-sum*。

如果子数组 $A[low..high]$包含 n 个元素(即 $n = high - low + 1$)，则调用 FIND-MAX-CROSSING-SUBARRAY(A, low, mid, $high$)花费 $\Theta(n)$时间。由于两个 **for** 循环的每次迭代花费 $\Theta(1)$时间，我们只需统计一共执行了多少次迭代。第 3～7 行的 **for** 循环执行了 $mid-low+1$ 次迭代，第 10～14 行的 **for** 循环执行了 $high-mid$ 次迭代，因此总循环迭代次数为

$$(mid - low + 1) + (high - mid) = high - low + 1 = n$$

有了一个线性时间的 FIND-MAX-CROSSING-SUBARRAY 在手，我们就可以设计求解最大子数组问题的分治算法的伪代码了：

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2      return (low, high, A[low])                  // base case: only one element
 3  else mid = ⌊(low+high)/2⌋
 4      (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5      (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY(A, mid+1, high)
 6      (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8          return (left-low, left-high, left-sum)
 9      elseif rightr-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

72

初始调用 FIND-MAXIMUM-SUBARRAY(A, 1, $A.length$)会求出 $A[1..n]$的最大子数组。

与 FIND-MAX-CROSSING-SUBARRAY 相似，递归过程 FIND-MAXIMUM-SUBARRAY 返回一个下标元组，划定了最大子数组的边界，同时返回最大子数组中的值之和。第 1 行测试基本情况，即子数组只有一个元素的情况。在此情况下，子数组只有一个子数组——它自身，因此第 2 行返回一个下标元组，开始和结束下标均指向唯一的元素，并返回此元素的值作为最大和。第 3～11 行处理递归情况。第 3 行划分子数组，计算中点下标 mid。我们称子数组 $A[low..mid]$为**左子数组**，$A[mid+1..high]$为**右子数组**。因为我们知道子数组 $A[low..high]$至少包含两个元

素，则左、右两个子数组各至少包含一个元素。第 4 行和第 5 行分别递归地求解左右子数组中的最大子数组。第 6～11 行完成合并工作。第 6 行求跨越中点的最大子数组(回忆一下，第 6 行求解的子问题并非原问题的规模更小的实例，因为我们将其看做合并部分)。第 7 行检测最大和子数组是否在左子数组中，若是，第 8 行返回此子数组。否则，第 9 行检测最大和子数组是否在右子数组中，若是，第 10 行返回此子数组。如果左、右子数组均不包含最大子数组，则最大子数组必然跨越中点，第 11 行将其返回。

分治算法的分析

接下来，我们建立一个递归式来描述递归过程 FIND-MAXIMUM-SUBARRAY 的运行时间。如 2.3.2 节中分析归并排序那样，对问题进行简化，假设原问题的规模为 2 的幂，这样所有子问题的规模均为整数。我们用 $T(n)$表示 FIND-MAXIMUM-SUBARRAY 求解 n 个元素的最大子数组的运行时间。首先，第 1 行花费常量时间。对于 $n=1$ 的基本情况，也很简单：第 2 行花费常量时间，因此，

$$T(1) = \Theta(1) \tag{4.5}$$

当 $n>1$ 时为递归情况。第 1 行和第 3 行花费常量时间。第 4 行和第 5 行求解的子问题均为 $n/2$ 个元素的子数组(假定原问题规模为 2 的幂，保证了 $n/2$ 为整数)，因此每个子问题的求解时间为 $T(n/2)$。因为我们需要求解两个子问题——左子数组和右子数组，因此第 4 行和第 5 行给
73 总运行时间增加了 $2T(n/2)$。而我们前面已经看到，第 6 行调用 FIND-MAX-CROSSING-SUBARRAY 花费 $\Theta(n)$时间。第 7～11 行仅花费 $\Theta(1)$时间。因此，对于递归情况，我们有

$$T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) = 2T(n/2) + \Theta(n) \tag{4.6}$$

组合式(4.5)和式(4.6)，我们得到 FIND-MAXIMUM-SUBARRAY 运行时间 $T(n)$的递归式：

$$T(n) = \begin{cases} \Theta(1) & \text{若 } n = 1 \\ 2T(n/2) + \Theta(n) & \text{若 } n > 1 \end{cases} \tag{4.7}$$

此递归式与式(4.1)归并排序的递归式一样。我们在 4.5 节将看到用主方法求解此递归式，其解为 $T(n)=\Theta(n\lg n)$。你也可以重新回顾一下图 2-5 中的递归树，来理解为什么解是 $T(n)=\Theta(n\lg n)$。

因此，我们看到利用分治方法得到了一个渐近复杂性优于暴力求解方法的算法。通过归并排序和本节的最大子数组问题，我们开始对分治方法的强大能力有了一些了解。有时，对某个问题，分治方法能给出渐近最快的算法，而其他时候，我们(不用分治方法)甚至能做得更好。如练习 4.1-5 所示，最大子数组问题实际上存在一个线性时间的算法，并未使用分治方法。

练习

4.1-1 当 A 的所有元素均为负数时，FIND-MAXIMUM-SUBARRAY 返回什么？

4.1-2 对最大子数组问题，编写暴力求解方法的伪代码，其运行时间应该为 $\Theta(n^2)$。

4.1-3 在你的计算机上实现最大子数组问题的暴力算法和递归算法。请指出多大的问题规模 n_0 是性能交叉点——从此之后递归算法将击败暴力算法？然后，修改递归算法的基本情况——当问题规模小于 n_0 时采用暴力算法。修改后，性能交叉点会改变吗？

74 **4.1-4** 假定修改最大子数组问题的定义，允许结果为空子数组，其和为 0。你应该如何修改现有算法，使它们能允许空子数组为最终结果？

4.1-5 使用如下思想为最大子数组问题设计一个非递归的、线性时间的算法。从数组的左边界开始，由左至右处理，记录到目前为止已经处理过的最大子数组。若已知 $A[1..j]$的最大子数组，基于如下性质将解扩展为 $A[1..j+1]$的最大子数组：$A[1..j+1]$的最大子数组要么是 $A[1..j]$的最大子数组，要么是某个子数组 $A[i..j+1](1\leqslant i\leqslant j+1)$。在已知 $A[1..j]$的最大子数组的情况下，可以在线性时间内找出形如 $A[i..j+1]$的最大子数组。

4.2 矩阵乘法的 Strassen 算法

如果你以前曾经接触过矩阵，可能了解如何进行矩阵乘法(否则，请阅读 D.1 节)。若 $A=(a_{ij})$和 $B=(b_{ij})$是 $n\times n$ 的方阵，则对 i，$j=1$，2，…，n，定义乘积 $C=A\cdot B$ 中的元素 c_{ij} 为：

$$c_{ij}=\sum_{k=1}^{n}a_{ik}\cdot b_{kj} \tag{4.8}$$

我们需要计算 n^2 个矩阵元素，每个元素是 n 个值的和。下面过程接收 $n\times n$ 矩阵 A 和 B，返回它们的乘积——$n\times n$ 矩阵 C。假设每个矩阵都有一个属性 *rows*，给出矩阵的行数。

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n×n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

过程 SQUARE-MATRIX-MULTIPLY 工作过程如下。第 3～7 行的 **for** 循环计算每行中的元素，在第 i 行中，第 4～7 行的 **for** 循环计算每列中的每个元素 c_{ij}。第 5 行将 c_{ij} 初始化为 0，开始公式(4.8)中的求和计算，第 6～7 行的 **for** 循环的每步迭代将公式(4.8)中的一项累加进来。 75

由于三重 **for** 循环的每一重都恰好执行 n 步，而第 7 行每次执行都花费常量时间，因此过程 SQUARE-MATRIX-MULTIPLY 花费 $\Theta(n^3)$时间。

你最初可能认为任何矩阵乘法都要花费 $\Omega(n^3)$时间，因为矩阵乘法的自然定义就需要进行这么多次的标量乘法。但这是错误的：我们有方法在 $o(n^3)$时间内完成矩阵乘法。在本节中，我们将看到 Strassen 的著名 $n\times n$ 矩阵相乘的递归算法。我们将在 4.5 节证明其运行时间为 $\Theta(n^{\lg 7})$。由于 lg7 在 2.80 和 2.81 之间，因此，Strassen 算法的运行时间为 $O(n^{2.81})$，渐近复杂性优于简单的 SQUARE-MATRIX-MULTIPLY 过程。

一个简单的分治算法

为简单起见，当使用分治算法计算矩阵积 $C=A\cdot B$ 时，假定三个矩阵均为 $n\times n$ 矩阵，其中 n 为 2 的幂。我们做出这个假设是因为在每个分解步骤中，$n\times n$ 矩阵都被划分为 4 个 $n/2\times n/2$ 的子矩阵，如果假定 n 是 2 的幂，则只要 $n\geqslant 2$ 即可保证子矩阵规模 $n/2$ 为整数。

假定将 A、B 和 C 均分解为 4 个 $n/2\times n/2$ 的子矩阵：

$$A=\begin{bmatrix}A_{11}&A_{12}\\A_{21}&A_{22}\end{bmatrix},\quad B=\begin{bmatrix}B_{11}&B_{12}\\B_{21}&B_{22}\end{bmatrix},\quad C=\begin{bmatrix}C_{11}&C_{12}\\C_{21}&C_{22}\end{bmatrix} \tag{4.9}$$

因此可以将公式 $C=A\cdot B$ 改写为：

$$\begin{bmatrix}C_{11}&C_{12}\\C_{21}&C_{22}\end{bmatrix}=\begin{bmatrix}A_{11}&A_{12}\\A_{21}&A_{22}\end{bmatrix}\cdot\begin{bmatrix}B_{11}&B_{12}\\B_{21}&B_{22}\end{bmatrix} \tag{4.10}$$

公式(4.10)等价于如下 4 个公式：

$$C_{11}=A_{11}\cdot B_{11}+A_{12}\cdot B_{21} \tag{4.11}$$

$$C_{12}=A_{11}\cdot B_{12}+A_{12}\cdot B_{22} \tag{4.12}$$

$$C_{21}=A_{21}\cdot B_{11}+A_{22}\cdot B_{21} \tag{4.13}$$

$$C_{22}=A_{21}\cdot B_{12}+A_{22}\cdot B_{22} \tag{4.14}$$

每个公式对应两对 $n/2\times n/2$ 矩阵的乘法及 $n/2\times n/2$ 积的加法。我们可以利用这些公式设计一个

76 直接的递归分治算法：

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
 1  n = A.rows
 2  let C be a new n×n matrix
 3  if n==1
 4      c11 = a11 · b11
 5  else partition A, B, and C as in equations (4.9)
 6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
 7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
 8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
 9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

这段伪代码掩盖了一个微妙但重要的实现细节。在第 5 行应该如何分解矩阵？如果我们真的创建 12 个新的 $n/2\times n/2$ 矩阵，将会花费 $\Theta(n^2)$时间复制矩阵元素。实际上，我们可以不必复制元素就能完成矩阵分解，其中的诀窍是使用下标计算。我们可以通过原矩阵的一组行下标和一组列下标来指明一个子矩阵。最终表示子矩阵的方法与表示原矩阵的方法略有不同，这就是我们省略的细节。这种表示方法的好处是，通过下标计算指明子矩阵，执行第 5 行只需 $\Theta(1)$的时间(虽然我们将看到是否通过复制元素来分解矩阵对总渐近运行时间并无影响)。

现在，我们推导出一个递归式来刻画 SQUARE-MATRIX-MULTIPLY-RECURSIVE 的运行时间。令 $T(n)$表示用此过程计算两个 $n\times n$ 矩阵乘积的时间。对 $n=1$ 的基本情况，我们只需进行一次标量乘法(第 4 行)，因此

$$T(1) = \Theta(1) \tag{4.15}$$

当 $n>1$ 时是递归情况。如前文所讨论，在第 5 行使用下标计算来分解矩阵花费 $\Theta(1)$时间。第 6～9 行，我们共 8 次递归调用 SQUARE-MATRIX-MULTIPLY-RECURSIVE。由于每次递归调用完成两个 $n/2\times n/2$ 矩阵的乘法，因此花费时间为 $T(n/2)$，8 次递归调用总时间为 $8T(n/2)$。我们还需要计算第 6～9 行的 4 次矩阵加法。每个矩阵包含 $n^2/4$ 个元素，因此，每次矩阵加法花
77 费 $\Theta(n^2)$时间。由于矩阵加法的次数是常数，第 6～9 行进行矩阵加法的总时间为 $\Theta(n^2)$(这里我们仍然使用下标计算方法将矩阵加法的结果放置于矩阵 C 的正确位置，由此带来的额外开销为每个元素 $\Theta(1)$时间)。因此，递归情况的总时间为分解时间、递归调用时间及矩阵加法时间之和：

$$T(n) = \Theta(1) + 8T(n/2) + \Theta(n^2) = 8T(n/2) + \Theta(n^2) \tag{4.16}$$

注意，如果通过复制元素来实现矩阵分解，额外开销为 $\Theta(n^2)$，递归式不会发生改变，只是总运行时间将会提高常数倍。

组合公式(4.15)和公式(4.16)，我们得到 SQUARE-MATRIX-MULTIPLY-RECURSIVE 运行时间的递归式：

$$T(n) = \begin{cases} \Theta(1) & \text{若 } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{若 } n > 1 \end{cases} \tag{4.17}$$

我们在 4.5 节将会看到利用主方法求解递归式(4.17)，得到的解为 $T(n)=\Theta(n^3)$。因此，简单的分治算法并不优于直接的 SQUARE-MATRIX-MULTIPLY 过程。

在继续介绍 Strassen 算法之前，让我们先回顾一下公式(4.16)的几个组成部分都是从何而来

的。用下标计算方法分解每个 $n\times n$ 矩阵花费 $\Theta(1)$时间，但有两个矩阵需要分解。虽然你可能认为分解两个矩阵需要 $\Theta(2)$时间，但实际上 Θ 符号中已经包含常数 2 在内了。假定每个矩阵包含 k 个元素，则两个矩阵相加需花费 $\Theta(k)$时间。由于每个矩阵包含 $n^2/4$ 个元素，每次加法花费 $\Theta(n^2/4)$时间。但是同样，Θ 符号已经包含常数因子 1/4，因此，两个 $n/2\times n/2$ 矩阵相加花费 $\Theta(n^2)$时间。我们需要进行 4 次矩阵加法，再次，我们并不说花费了 $\Theta(4n^2)$时间，而是 $\Theta(n^2)$时间。(当然，你可能发现我们可以说 4 次矩阵加法花费了 $\Theta(4n^2/4)$时间，而 $4n^2/4=n^2$，但此处的要点是 Θ 符号已经包含了常数因子，无论怎样的常数因子均可省略。)因此，我们最终得到两项 $\Theta(n^2)$，可以将它们合二为一。

但是，当分析 8 次递归调用时，就不能简单省略常数因子 8 了。换句话说，我们必须说递归调用共花费 $8T(n/2)$时间，而不是 $T(n/2)$时间。至于这是为什么，你可以回顾一下图 2-5 中的递归树，它对应递归式(2.1)(与递归式(4.7)相同)，其递归情况为 $T(n)=2T(n/2)+\Theta(n)$。因子 2 决定了树中每个结点有几个孩子结点，进而决定了树的每一层为总和贡献了多少项。如果省略 78
公式(4.16)中的因子 8 或递归式(4.1)中的因子 2，递归树就变为线性结构，而不是“茂盛的”了，树的每一层只为总和贡献了一项。

因此，切记，虽然渐近符号包含了常数因子，但递归符号(如 $T(n/2)$)并不包含。

Strassen 方法

Strassen 算法的核心思想是令递归树稍微不那么茂盛一点儿，即只递归进行 7 次而不是 8 次 $n/2\times n/2$ 矩阵的乘法。减少一次矩阵乘法带来的代价可能是额外几次 $n/2\times n/2$ 矩阵的加法，但只是常数次。与前文一样，当建立递归式刻画运行时间时，常数次矩阵加法被 Θ 符号包含在内。

Strassen 算法不是那么直观(这可能是本书陈述最不充分的地方了)。它包含 4 个步骤：

1. 按公式(4.9)将输入矩阵 A、B 和输出矩阵 C 分解为 $n/2\times n/2$ 的子矩阵。采用下标计算方法，此步骤花费 $\Theta(1)$时间，与 SQUARE-MATRIX-MULTIPLY-RECURSIVE 相同。

2. 创建 10 个 $n/2\times n/2$ 的矩阵 S_1，S_2，…，S_{10}，每个矩阵保存步骤 1 中创建的两个子矩阵的和或差。花费时间为 $\Theta(n^2)$。

3. 用步骤 1 中创建的子矩阵和步骤 2 中创建的 10 个矩阵，递归地计算 7 个矩阵积 P_1，P_2，…，P_7。每个矩阵 P_i 都是 $n/2\times n/2$ 的。

4. 通过 P_i 矩阵的不同组合进行加减运算，计算出结果矩阵 C 的子矩阵 C_{11}，C_{12}，C_{21}，C_{22}。花费时间 $\Theta(n^2)$。

我们稍后会看到步骤 2～4 的细节，但现在可以建立 Strassen 算法的运行时间递归式。假定一旦矩阵规模从 n 变为 1，就进行简单的标量乘法计算，正如 SQUARE-MATRIX-MULTIPLY-RECURSIVE 的第 4 行那样。当 $n>1$ 时，步骤 1、2 和 4 共花费 $\Theta(n^2)$时间，步骤 3 要求进行 7 次 $n/2\times n/2$ 矩阵的乘法。因此，我们得到如下描述 Strassen 算法运行时间 $T(n)$的递归式：

$$T(n)=\begin{cases}\Theta(1) & \text{若 } n=1\\ 7T(n/2)+\Theta(n^2) & \text{若 } n>1\end{cases} \tag{4.18}$$ 79

我们用常数次矩阵乘法的代价减少了一次矩阵乘法。一旦我们理解了递归式及其解，就会看到这种交换确实能带来更低的渐近运行时间。利用 4.5 节的主方法，可以求出递归式(4.18)的解为 $T(n)=\Theta(n^{\lg 7})$。

我们现在来介绍 Strassen 算法的细节。在步骤 2 中，创建如下 10 个矩阵：

$$S_1=B_{12}-B_{22}$$

$$S_2=A_{11}+A_{12}$$

$$
\begin{aligned}
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

由于必须进行 10 次 $n/2 \times n/2$ 矩阵的加减法，因此，该步骤花费 $\Theta(n^2)$ 时间。

在步骤 3 中，递归地计算 7 次 $n/2 \times n/2$ 矩阵的乘法，如下所示：

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{aligned}
$$

注意，上述公式中，只有中间一列的乘法是真正需要计算的。右边这列只是用来说明这些乘积与步骤 1 创建的原始子矩阵之间的关系。

步骤 4 对步骤 3 创建的 P_i 矩阵进行加减法运算，计算出 C 的 4 个 $n/2 \times n/2$ 的子矩阵，首先，

80

$$
C_{11} = P_5 + P_4 - P_2 + P_6
$$

利用每个 P_i 的展开式展开等式右部，每个 P_i 的展开式位于单独一行，并将可以消去的项垂直对齐，我们可以看到 C_{11} 等于

$$
\begin{array}{llll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} \\
 & - A_{22} \cdot B_{11} & & + A_{22} \cdot B_{21} \\
- A_{11} \cdot B_{22} & & & - A_{12} \cdot B_{22} \\
 & & - A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} + A_{12} \cdot B_{22} + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} & & & + A_{12} \cdot B_{21}
\end{array}
$$

与公式(4.11)相同。类似地，令

$$
C_{12} = P_1 + P_2
$$

则 C_{12} 等于

$$
\begin{array}{ll}
A_{11} \cdot B_{12} & - A_{11} \cdot B_{22} \\
 & + A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
\hline
A_{11} \cdot B_{12} & + A_{12} \cdot B_{22}
\end{array}
$$

与公式(4.12)相同。令

$$
C_{21} = P_3 + P_4
$$

使 C_{21} 等于

$$
\begin{array}{ll}
A_{21} \cdot B_{11} & + A_{22} \cdot B_{11} \\
 & - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \\
\hline
A_{21} \cdot B_{11} & + A_{22} \cdot B_{21}
\end{array}
$$

与公式(4.13)相同。最后，令

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

则 C_{22} 等于

$$\begin{aligned} & A_{11}\cdot B_{11} + A_{11}\cdot B_{22} + A_{22}\cdot B_{11} + A_{22}\cdot B_{22} \\ & \quad - A_{11}\cdot B_{22} \qquad\qquad + A_{11}\cdot B_{12} \\ & \qquad - A_{22}\cdot B_{11} \qquad\qquad - A_{21}\cdot B_{11} \\ & - A_{11}\cdot B_{11} \qquad\qquad - A_{11}\cdot B_{12} + A_{21}\cdot B_{11} + A_{21}\cdot B_{12} \\ \hline & \qquad\qquad A_{22}\cdot B_{22} \qquad\qquad + A_{21}\cdot B_{12} \end{aligned}$$

81

与公式(4.14)相同。在步骤4中，共进行了8次 $n/2\times n/2$ 矩阵的加减法，因此花费 $\Theta(n^2)$ 时间。

因此，我们看到由4个步骤构成的Strassen算法，确实生成了正确的矩阵乘积，递归式(4.18)刻画了它的运行时间。由于我们将在4.5节看到此递归式的解为 $T(n)=\Theta(n^{\lg 7})$，Strassen方法的渐近复杂性低于直接的SQUARE-MATRIX-MULTIPLY过程。本章注记会讨论Strassen算法实际应用方面的一些问题。

练习

注意：虽然练习4.2-3、4.2-4和4.2-5是关于Strassen算法的变形的，但你应该先阅读4.5节，然后再尝试求解这几个问题。

4.2-1 使用Strassen算法计算如下矩阵乘法：

$$\begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix}\begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix}$$

给出计算过程。

4.2-2 为Strassen算法编写伪代码。

4.2-3 如何修改Strassen算法，使之适应矩阵规模 n 不是2的幂的情况？证明：算法的运行时间为 $\Theta(n^{\lg 7})$。

4.2-4 如果可以用 k 次乘法操作(假定乘法的交换律不成立)完成两个 3×3 矩阵相乘，那么你可以在 $o(n^{\lg 7})$ 时间内完成 $n\times n$ 矩阵相乘，满足这一条件的最大的 k 是多少？此算法的运行时间是怎样的？

4.2-5 V. Pan发现一种方法，可以用132 464次乘法操作完成 68×68 的矩阵相乘，发现另一种方法，可以用143 640次乘法操作完成 70×70 的矩阵相乘，还发现一种方法，可以用155 424次乘法操作完成 72×72 的矩阵相乘。当用于矩阵相乘的分治算法时，上述哪种方法会得到最佳的渐近运行时间？与Strassen算法相比，性能如何？

82

4.2-6 用Strassen算法作为子进程来进行一个 $kn\times n$ 矩阵和一个 $n\times kn$ 矩阵相乘，最快需要花费多长时间？对两个输入矩阵规模互换的情况，回答相同的问题。

4.2-7 设计算法，仅使用三次实数乘法即可完成复数 $a+bi$ 和 $c+di$ 相乘。算法需接收 a、b、c 和 d 为输入，分别生成实部 $ac-bd$ 和虚部 $ad+bc$。

4.3 用代入法求解递归式

我们已经看到如何用递归式刻画分治算法的运行时间，下面将学习如何求解递归式。我们从“代入”法开始。

代入法求解递归式分为两步：

1. 猜测解的形式。

2. 用数学归纳法求出解中的常数，并证明解是正确的。

当将归纳假设应用于较小的值时，我们将猜测的解代入函数，因此得名“代入法”。这种方法很强大，但我们必须能猜出解的形式，以便将其代入。

我们可以用代入法为递归式建立上界或下界。例如，我们确定下面递归式的上界：

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{4.19}$$

该递归式与递归式(4.3)和(4.4)相似。我们猜测其解为 $T(n)=O(n\lg n)$。代入法要求证明，恰当选择常数 $c>0$，可有 $T(n)\leqslant cn\lg n$。首先假定此上界对所有正数 $m<n$ 都成立，特别是对于 $m=\lfloor n/2 \rfloor$，有 $T(\lfloor n/2 \rfloor)\leqslant c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$。将其代入递归式，得到

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

83

其中，只要 $c\geqslant 1$，最后一步都会成立。

数学归纳法要求我们证明解在边界条件下也成立。为证明这一点，我们通常证明对于归纳证明，边界条件适合作为基本情况。对递归式(4.19)，我们必须证明，通过选择足够大的常数 c，可以使得上界 $T(n)\leqslant cn\lg n$ 对边界条件也成立。这一要求有时可能引起问题。例如，为了方便讨论，假设 $T(1)=1$ 是递归式唯一的边界条件。对 $n=1$，边界条件 $T(n)\leqslant cn\lg n$ 推导出 $T(1)\leqslant c1\lg 1=0$，与 $T(1)=1$ 矛盾。因此，我们的归纳证明的基本情况不成立。

我们稍微多付出一点努力，就可以克服这个障碍，对特定的边界条件证明归纳假设成立。例如，在递归式(4.19)中，渐近符号仅要求我们对 $n\geqslant n_0$ 证明 $T(n)\leqslant cn\lg n$，其中 n_0 是我们可以自己选择的常数，我们可以充分利用这一点。我们保留麻烦的边界条件 $T(1)=1$，但将其从归纳证明中移除。为了做到这一点，首先观察到对于 $n>3$，递归式并不直接依赖 $T(1)$。因此，将归纳证明中的基本情况 $T(1)$ 替换为 $T(2)$ 和 $T(3)$，并令 $n_0=2$。注意，我们将递归式的基本情况($n=1$)和归纳证明的基本情况($n=2$ 和 $n=3$)区分开来了。由 $T(1)=1$，从递归式推导出 $T(2)=4$ 和 $T(3)=5$。现在可以完成归纳证明：对某个常数 $c\geqslant 1$，$T(n)\leqslant cn\lg n$，方法是选择足够大的 c，满足 $T(2)\leqslant c2\lg 2$ 和 $T(3)\leqslant c3\lg 3$。事实上，任何 $c\geqslant 2$ 都能保证 $n=2$ 和 $n=3$ 的基本情况成立。对于我们所要讨论的大多数递归式来说，扩展边界条件使归纳假设对较小的 n 成立，是一种简单直接的方法，我们将不再总是显式说明这方面的细节。

做出好的猜测

遗憾的是，并不存在通用的方法来猜测递归式的正确解。猜测解要靠经验，偶尔还需要创造力。幸运的是，你可以使用一些启发式方法帮助你成为一个好的猜测者。你也可以使用递归树来做出好的猜测，我们将在 4.4 节看到这一方法。

如果要求解的递归式与你曾见过的递归式相似，那么猜测一个类似的解是合理的。例如，考虑如下递归式：

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

84

看起来很困难，因为在等式右边 T 的参数中增加了“17”。但直观上，增加的这一项不会显著影响递归式的解。当 n 较大时，$\lfloor n/2 \rfloor$和$\lfloor n/2 \rfloor+17$ 的差距不大：都是接近 n 的一半。因此，我们猜测 $T(n)=O(n\lg n)$，你可以使用代入法验证这个猜测是正确的(见练习 4.3-6)。

另一种做出好的猜测的方法是先证明递归式较松的上界和下界，然后缩小不确定的范围。例如，对递归式(4.19)，我们可以从下界 $T(n)=\Omega(n)$ 开始，因为递归式中包含 n 这一项，还可以证明一个初始上界 $T(n)=O(n^2)$。然后，我们可以逐渐降低上界，提升下界，直至收敛到渐近紧确界 $T(n)=\Theta(n\lg n)$。

微妙的细节

有时你可能正确猜出了递归式解的渐近界，但莫名其妙地在归纳证明时失败了。问题常常出在归纳假设不够强，无法证出准确的界。当遇到这种障碍时，如果修改猜测，将它减去一个低阶的项，数学证明常常能顺利进行。

考虑如下递归式：

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

我们猜测解为 $T(n)=O(n)$，并尝试证明对某个恰当选出的常数 c，$T(n) \leqslant cn$ 成立。将我们的猜测代入递归式，得到

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

这并不意味着对任意 c 都有 $T(n) \leqslant cn$。我们可能忍不住尝试猜测一个更大的界，比如 $T(n)=O(n^2)$。虽然从这个猜测也能推出结果，但原来的猜测 $T(n)=O(n)$ 是正确的。然而为了证明它是正确的，我们必须做出更强的归纳假设。

直觉上，我们的猜测是接近正确的：只差一个常数 1，一个低阶项。但是，除非我们证明与归纳假设严格一致的形式，否则数学归纳法还是会失败。克服这个困难的方法是从先前的猜测中减去一个低阶项。新的猜测为 $T(n) \leqslant cn-d$，d 是大于等于 0 的一个常数。我们现在有

$$\begin{aligned} T(n) &\leqslant (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \leqslant cn - d \end{aligned}$$

85

只要 $d \geqslant 1$，此式就成立。与以前一样，我们必须选择足够大的 c 来处理边界条件。

你可能发现减去一个低阶项的想法与直觉是相悖的。毕竟，如果证明上界失败了，就应该将猜测增加而不是减少，更松的界难道不是更容易证明吗？不一定！当利用归纳法证明一个上界时，实际上证明一个更弱的上界可能会更困难一些，因为为了证明一个更弱的上界，我们在归纳证明中也必须使用同样更弱的界。在当前的例子中，当递归式包含超过一个递归项时，将猜测的界减去一个低阶项意味着每次对每个递归项都减去一个低阶项。在上例中，我们减去常数 d 两次，一次是对 $T(\lfloor n/2 \rfloor)$项，另一次是对 $T(\lceil n/2 \rceil)$项。我们以不等式 $T(n) \leqslant cn-2d+1$ 结束，可以很容易地找到一个 d 值，使得 $cn-2d+1$ 小于等于 $cn-d$。

避免陷阱

使用渐近符号很容易出错。例如，在递归式(4.19)中，我们可能错误地“证明”$T(n)=O(n)$：猜测 $T(n) \leqslant cn$，并论证

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

因为 c 是常数。错误在于我们并未证出与归纳假设*严格一致的形式*，即 $T(n) \leqslant cn$。因此，当要证明 $T(n)=O(n)$时，需要显式地证出 $T(n) \leqslant cn$。

改变变量

有时，一个小的代数运算可以将一个未知的递归式变成你所熟悉的形式。例如，考虑如下递归式：

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

它看起来很困难。但我们可以通过改变变量来简化它。为方便起见，我们不必担心值的舍入误差问题，只考虑$\sqrt{n}$是整数的情形即可。令 $m=\lg n$，得到

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

现在重命名 $S(m)=T(2^m)$，得到新的递归式：

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

86

它与递归式(4.19)非常像。这个新的递归式确实与(4.19)具有相同的解：$S(m)=O(m\lg m)$。再从 $S(m)$转换回 $T(n)$，我们得到 $T(n)=T(2^m)=S(m)=O(m\lg m)=O(\lg n \lg\lg n)$。

练习

4.3-1 证明：$T(n)=T(n-1)+n$ 的解为 $O(n^2)$。

4.3-2 证明：$T(n)=T(\lceil n/2\rceil)+1$ 的解为 $O(\lg n)$。

4.3-3 我们看到 $T(n)=2T(\lfloor n/2\rfloor)+n$ 的解为 $O(n\lg n)$。证明 $\Omega(n\lg n)$也是这个递归式的解。从而得出结论：解为 $\Theta(n\lg n)$。

4.3-4 证明：通过做出不同的归纳假设，我们不必调整归纳证明中的边界条件，即可克服递归式(4.19)中边界条件 $T(1)=1$ 带来的困难。

4.3-5 证明：归并排序的“严格”递归式(4.3)的解为 $\Theta(n\lg n)$。

4.3-6 证明：$T(n)=2T(\lfloor n/2\rfloor+17)+n$ 的解为 $O(n\lg n)$。

4.3-7 使用 4.5 节中的主方法，可以证明 $T(n)=4T(n/3)+n$ 的解为 $T(n)=\Theta(n^{\log_3 4})$。说明基于假设 $T(n)\leqslant cn^{\log_3 4}$ 的代入法不能证明这一结论。然后说明如何通过减去一个低阶项完成代入法证明。

4.3-8 使用 4.5 节中的主方法，可以证明 $T(n)=4T(n/2)+n$ 的解为 $T(n)=\Theta(n^2)$。说明基于假设 $T(n)\leqslant cn^2$ 的代入法不能证明这一结论。然后说明如何通过减去一个低阶项完成代入法证明。

87

4.3-9 利用改变变量的方法求解递归式 $T(n)=3T(\sqrt{n})+\log n$。你的解应该是渐近紧确的。不必担心数值是否是整数。

4.4 用递归树方法求解递归式

虽然你可以用代入法简洁地证明一个解确是递归式的正确解，但想出一个好的猜测可能会很困难。画出递归树，如我们在 2.3.2 节分析归并排序的递归式时所做的那样，是设计好的猜测的一种简单而直接的方法。在**递归树**中，每个结点表示一个单一子问题的代价，子问题对应某次递归函数调用。我们将树中每层中的代价求和，得到每层代价，然后将所有层的代价求和，得到所有层次的递归调用的总代价。

递归树最适合用来生成好的猜测，然后即可用代入法来验证猜测是否正确。当使用递归树来生成好的猜测时，常常需要忍受一点儿“不精确”，因为稍后才会验证猜测是否正确。但如果在画递归树和代价求和时非常仔细，就可以用递归树直接证明解是否正确。在本节中，我们将使用递归树生成好的猜测，并且在 4.6 节中，我们将使用递归树直接证明主方法的基础定理。

我们以递归式 $T(n)=3T(\lfloor n/4\rfloor)+\Theta(n^2)$ 为例来看一下如何用递归树生成一个好的猜测。首先关注如何寻找解的一个上界。因为我们知道舍入对求解递归式通常没有影响(此处即是我们需要忍受不精确的一个例子)，因此可以为递归式 $T(n)=3T(\lfloor n/4\rfloor)+cn^2$ 创建一棵递归树，其中已将渐近符号改写为隐含的常数系数 $c>0$。

图 4-5 显示了如何从递归式 $T(n)=3T(\lfloor n/4\rfloor)+cn^2$ 构造出递归树。为方便起见，我们假定 n 是 4 的幂(忍受不精确的另一个例子)，这样所有子问题的规模均为正数。图 4-5(a)显示了 $T(n)$，它在图 4-5(b)中扩展为一棵等价的递归树。根结点中的 cn^2 项表示递归调用顶层的代价，根的三棵子树表示规模为 $n/4$ 的子问题所产生的代价。图 4-5(c)显示了进一步构造递归树的过程，将图 4-5(b)中代价为 $T(n/4)$的结点逐一扩展。我们继续扩展树中每个结点，根据递归式确定的关系将其分解为几个组成部分(孩子结点)。

88

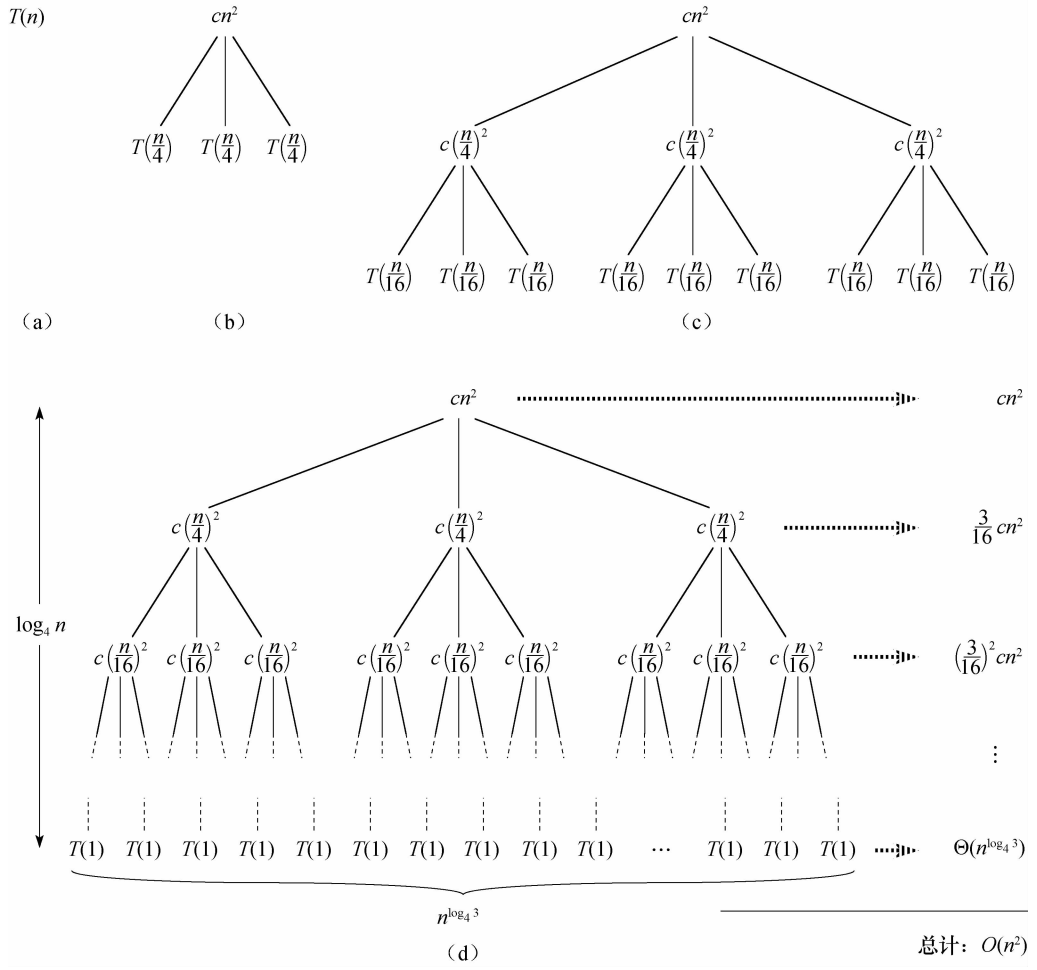


图 4-5　为递归式 $T(n)=3T(\lfloor n/4\rfloor)+cn^2$ 构造递归树。(a)显示了 $T(n)$，在(b)～(d)中逐步扩展为递归树的形式。(d)中显示了扩展完毕的递归树，其高度为 $\log_4 n$(有 $\log_4 n+1$ 层)

89

因为子问题的规模每一步减少为上一步的 1/4，所以最终必然会达到边界条件。那么根结点与距离为 1 的子问题距离多远呢？深度为 i 的结点对应规模为 $n/4^i$ 的子问题。因此，当 $n/4^i=1$，或等价地 $i=\log_4 n$ 时，子问题规模变为 1。因此，递归树有 $\log_4 n+1$ 层(深度为 0，1，2，…，$\log_4 n$)。

接下来确定树的每一层的代价。每层的结点数都是上一层的 3 倍，因此深度为 i 的结点数为 3^i。因为每一层子问题规模都是上一层的 1/4，所以对 $i=0,1,2,\cdots,\log_4 n-1$，深度为 i 的每个结点的代价为 $c(n/4^i)^2$。做一下乘法可得，对 $i=0,1,2,\cdots,\log_4 n-1$，深度为 i 的所有结点的总代价为 $3^i c(n/4^i)^2=(3/16)^i cn^2$。树的最底层深度为 $\log_4 n$，有 $3^{\log_4 n}=n^{\log_4 3}$ 个结点，每个结点的代价为 $T(1)$，总代价为 $n^{\log_4 3}T(1)$，即 $\Theta(n^{\log_4 3})$，因为假定 $T(1)$ 是常量。

现在我们求所有层次的代价之和，确定整棵树的代价：

$$\begin{aligned}T(n) &= cn^2+\frac{3}{16}cn^2+\left(\frac{3}{16}\right)^2 cn^2+\cdots+\left(\frac{3}{16}\right)^{\log_4 n-1} cn^2+\Theta(n^{\log_4 3})\\ &= \sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2+\Theta(n^{\log_4 3})\\ &= \frac{(3/16)^{\log_4 n}-1}{(3/16)-1}cn^2+\Theta(n^{\log_4 3}) \quad \text{(根据公式(A.5))}\end{aligned}$$

最后的这个公式看起来有些凌乱，但我们可以再次充分利用一定程度的不精确，并利用无限递减几何级数作为上界。回退一步，应用公式(A.6)，我们得到

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

这样，对原始的递归式 $T(n)=3T(\lfloor n/4 \rfloor)+\Theta(n^2)$，我们推导出了一个猜测 $T(n)=O(n^2)$。在本
90 例中，cn^2 的系数形成了一个递减几何级数，利用公式(A.6)，得出这些系数的和的一个上界——常数 16/13。由于根结点对总代价的贡献为 cn^2，所以根结点的代价占总代价的一个常数比例。换句话说，根结点的代价支配了整棵树的总代价。

实际上，如果 $O(n^2)$ 确实是递归式的上界(稍后就会证明这一点)，那么它必然是一个紧确界。为什么？因为第一次递归调用的代价为 $\Theta(n^2)$，因此 $\Omega(n^2)$ 必然是递归式的一个下界。

现在用代入法验证猜测是正确的，即 $T(n)=O(n^2)$ 是递归式 $T(n)=3T(\lfloor n/4 \rfloor)+\Theta(n^2)$ 的一个上界。我们希望证明 $T(n) \leqslant dn^2$ 对某个常数 $d>0$ 成立。与之前一样，使用常数 $c>0$，我们有

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

当 $d \geqslant (16/13)c$ 时，最后一步推导成立。

在另一个更复杂的例子中，图 4-6 显示了如下递归式的递归树：

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

(为简单起见，再次忽略了舍入问题。)与之前一样，令 c 表示 $O(n)$ 项中的常数因子。对图中显示出的递归树的每个层次，当求代价之和
91 时，我们发现每层的代价均为 cn。从根到叶的最长简单路径是 $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$。由于当 $k=\log_{3/2} n$ 时，$(2/3)^k n = 1$，因此树高为 $\log_{3/2} n$。

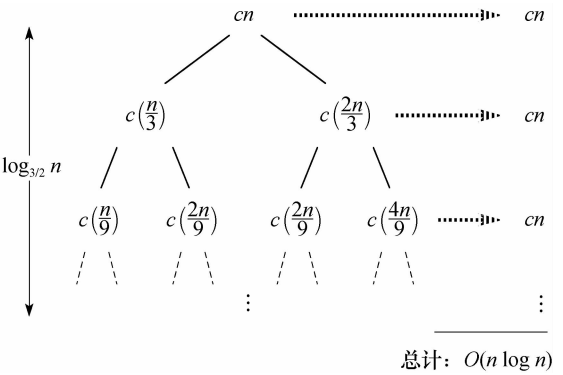


图 4-6 递归式 $T(n)=T(n/3)+T(2n/3)+cn$

直觉上，我们期望递归式的解最多是层数乘以每层的代价，即 $O(cn \log_{3/2} n)=O(n \lg n)$。但图 4-6 仅显示了递归树的顶部几层，并不是递归树中每个层次的代价都是 cn。考虑叶结点的代价。如果递归树是一棵高度为 $\log_{3/2} n$ 的完全二叉树，则叶结点的数量应为 $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$。由于每个叶结点的代价为常数，因此所有叶结点的总代价为 $\Theta(n^{\log_{3/2} 2})$，由于 $\log_{3/2} n$ 是严格大于 1 的常数，因此叶结点代价总和为 $\Omega(n \lg n)$。但递归树并不是完全二叉树，因此叶结点数量小于 $n^{\log_{3/2} 2}$。而且，当从根结点逐步向下走时，越来越多的内结点是缺失的。因此，递归树中靠下的层次对总代价的贡献小于 cn。我们可以计算出所有代价的准确值，但记住我们只是希望得到一个猜测，用于代入法。我们还是忍受一些不精确，尝试证明猜测的上界 $O(n \lg n)$ 是正确的。

我们确实可以用代入法验证 $O(n \lg n)$ 是递归式解的一个上界。我们来证明 $T(n) \leqslant dn \lg n$，其中 d 是一个适当的正常数。我们有

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

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

只要 $d \geqslant c/(\lg 3-(2/3))$。因此，无需对递归树的代价进行更精确的计算。

练习

4.4-1 对递归式 $T(n)=3T(\lfloor n/2 \rfloor)+n$，利用递归树确定一个好的渐近上界，用代入法进行验证。

4.4-2 对递归式 $T(n)=T(n/2)+n^2$，利用递归树确定一个好的渐近上界，用代入法进行验证。 92

4.4-3 对递归式 $T(n)=4T(n/2+2)+n$，利用递归树确定一个好的渐近上界，用代入法进行验证。

4.4-4 对递归式 $T(n)=T(n-1)+1$，利用递归树确定一个好的渐近上界，用代入法进行验证。

4.4-5 对递归式 $T(n)=T(n-1)+T(n/2)+n$，利用递归树确定一个好的渐近上界，用代入法进行验证。

4.4-6 对递归式 $T(n)=T(n/3)+T(2n/3)+cn$，利用递归树论证其解为 $\Omega(n\lg n)$，其中 c 为常数。

4.4-7 对递归式 $T(n)=4T(\lfloor n/2 \rfloor)+cn$（$c$ 为常数），画出递归树，并给出其解的一个渐近紧确界。用代入法进行验证。

4.4-8 对递归式 $T(n)=T(n-a)+T(a)+cn$，利用递归树给出一个渐近紧确解，其中 $a \geqslant 1$ 和 $c>0$是常数。

4.4-9 对递归式 $T(n)=T(\alpha n)+T((1-\alpha)n)+cn$，利用递归树给出一个渐近紧确解，其中 $0<\alpha<1$ 和 $c>0$ 是常数。

4.5 用主方法求解递归式

主方法为如下形式的递归式提供了一种“菜谱”式的求解方法

$$T(n) = aT(n/b) + f(n) \tag{4.20}$$

其中 $a \geqslant 1$ 和 $b>1$ 是常数，$f(n)$是渐近正函数。为了使用主方法，需要牢记三种情况，但随后你就可以很容易地求解很多递归式，通常不需要纸和笔的帮助。 93

递归式(4.20)描述的是这样一种算法的运行时间：它将规模为 n 的问题分解为 a 个子问题，每个子问题规模为 n/b，其中 a 和 b 都是正常数。a 个子问题递归地进行求解，每个花费时间 $T(n/b)$。函数 $f(n)$包含了问题分解和子问题解合并的代价。例如，描述 Strassen 算法的递归式中，$a=7$，$b=2$，$f(n)=\Theta(n^2)$。

从技术的正确性方面看，此递归式实际上并不是良好定义的，因为 n/b 可能不是整数。但将 a 项 $T(n/b)$都替换为 $T(\lfloor n/b \rfloor)$或 $T(\lceil n/b \rceil)$并不会影响递归式的渐近性质(我们将在下一节证明这个断言)。因此，我们通常发现当写下这种形式的分治算法的递归式时，忽略舍入问题是很方便的。

主定理

主方法依赖于下面的定理。

定理 4.1(主定理) 令 $a \geqslant 1$ 和 $b>1$ 是常数，$f(n)$是一个函数，$T(n)$是定义在非负整数上的递归式：

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

其中我们将 n/b 解释为 $\lfloor n/b \rfloor$ 或 $\lceil n/b \rceil$。那么 $T(n)$ 有如下渐近界：

1. 若对某个常数 $\varepsilon > 0$ 有 $f(n) = O(n^{\log_b a - \varepsilon})$，则 $T(n) = \Theta(n^{\log_b a})$。

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

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

在使用主定理之前，我们花一点儿时间尝试理解一下它的含义。对于三种情况的每一种，我们将函数 $f(n)$ 与函数 $n^{\log_b a}$ 进行比较。直觉上，两个函数较大者决定了递归式的解。若函数 $n^{\log_b a}$ 更大，如情况 1，则解为 $T(n) = \Theta(n^{\log_b a})$。若函数 $f(n)$ 更大，如情况 3，则解为 $T(n) = \Theta(f(n))$。若两个函数大小相当，如情况 2，则乘上一个对数因子，解为 $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$。

在此直觉之外，我们需要了解一些技术细节。在第一种情况中，不是 $f(n)$ 小于 $n^{\log_b a}$ 就够了，
94 而是要多项式意义上的小于。也就是说，$f(n)$ 必须渐近小于 $n^{\log_b a}$，要相差一个因子 n^{ε}，其中 ε 是大于 0 的常数。在第三种情况中，不是 $f(n)$ 大于 $n^{\log_b a}$ 就够了，而是要多项式意义上的大于，而且还要满足“正则”条件 $af(n/b) \leqslant cf(n)$。我们将会遇到的多项式界的函数中，多数都满足此条件。

注意，这三种情况并未覆盖 $f(n)$ 的所有可能性。情况 1 和情况 2 之间有一定间隙，$f(n)$ 可能小于 $n^{\log_b a}$ 但不是多项式意义上的小于。类似地，情况 2 和情况 3 之间也有一定间隙，$f(n)$ 可能大于 $n^{\log_b a}$ 但不是多项式意义上的大于。如果函数 $f(n)$ 落在这两个间隙中，或者情况 3 中要求的正则条件不成立，就不能使用主方法来求解递归式。

使用主方法

使用主方法很简单，我们只需确定主定理的哪种情况成立，即可得到解。

我们先看下面这个例子

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

对于这个递归式，我们有 $a = 9$，$b = 3$，$f(n) = n$，因此 $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$。由于 $f(n) = O(n^{\log_3 9 - \varepsilon})$，其中 $\varepsilon = 1$，因此可以应用主定理的情况 1，从而得到解 $T(n) = \Theta(n^2)$。

现在考虑

$$T(n) = T(2n/3) + 1$$

其中 $a = 1$，$b = 3/2$，$f(n) = 1$，因此 $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$。由于 $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$，因此应用情况 2，从而得到解 $T(n) = \Theta(\lg n)$。

对于递归式

$$T(n) = 3T(n/4) + n \lg n$$

我们有 $a = 3$，$b = 4$，$f(n) = n \lg n$，因此 $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$。由于 $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$，其中 $\varepsilon \approx 0.2$，因此，如果可以证明正则条件成立，即可应用情况 3。当 n 足够大时，对于 $c = 3/4$，$af(n/b) = 3(n/4)\lg(n/4) \leqslant (3/4)n \lg n = cf(n)$。因此，由情况 3，递归式的解为 $T(n) = \Theta(n \lg n)$。

主方法不能用于如下递归式：

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

虽然这个递归式看起来有恰当的形式：$a = 2$，$b = 2$，$f(n) = n \lg n$，以及 $n^{\log_b a} = n$。你可能错误地
95 认为应该应用情况 3，因为 $f(n) = n \lg n$ 渐近大于 $n^{\log_b a} = n$。问题出在它并不是多项式意义上的大于。对任意正常数 ε，比值 $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ 都渐近小于 n^{ε}。因此，递归式落入了情况 2 和情况 3 之间的间隙（此递归式的解参见练习 4.6-2）。

我们利用主方法求解在 4.1 节和 4.2 节中曾见过的递归式(4.7)，

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

它刻画了最大子数组问题和归并排序的分治算法的运行时间（按照通常的做法，我们忽略了递归

式中基本情况的描述)。这里，我们有 $a=2$，$b=2$，$f(n)=\Theta(n)$，因此 $n^{\log_b a}=n^{\log_2 2}=n$。由于 $f(n)=\Theta(n)$，应用情况2，于是得到解 $T(n)=\Theta(n\lg n)$。

递归式(4.17)，

$$T(n)=8T(n/2)+\Theta(n^2)$$

它描述了矩阵乘法问题第一个分治算法的运行时间。我们有 $a=8$，$b=2$，$f(n)=\Theta(n^2)$，因此 $n^{\log_b a}=n^{\log_2 8}=n^3$。由于 n^3 多项式意义上大于 $f(n)$(即对 $\varepsilon=1$，$f(n)=O(n^{3-\varepsilon})$)，应用情况1，解为 $T(n)=\Theta(n^3)$。

最后，我们考虑递归式(4.18)，

$$T(n)=7T(n/2)+\Theta(n^2)$$

它描述了Strassen算法的运行时间。这里，我们有 $a=7$，$b=2$，$f(n)=\Theta(n^2)$，因此 $n^{\log_b a}=n^{\log_2 7}$。将 $\log_2 7$ 改写为 $\lg 7$，由于 $2.80<\lg 7<2.81$，我们知道对 $\varepsilon=0.8$，有 $f(n)=O(n^{\lg 7-\varepsilon})$。再次应用情况1，我们得到解 $T(n)=\Theta(n^{\lg 7})$。

练习

4.5-1 对下列递归式，使用主方法求出渐近紧确界。

a. $T(n)=2T(n/4)+1$

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

c. $T(n)=2T(n/4)+n$

d. $T(n)=2T(n/4)+n^2$

96

4.5-2 Caesar教授想设计一个渐近快于Strassen算法的矩阵相乘算法。他的算法使用分治方法，将每个矩阵分解为 $n/4\times n/4$ 的子矩阵，分解和合并步骤共花费 $\Theta(n^2)$ 时间。他需要确定，他的算法需要创建多少个子问题，才能击败Strassen算法。如果他的算法创建 a 个子问题，则描述运行时间 $T(n)$ 的递归式为 $T(n)=aT(n/4)+\Theta(n^2)$。Caesar教授的算法如果要渐近快于Strassen算法，a 的最大整数值应是多少?

4.5-3 使用主方法证明：二分查找递归式 $T(n)=T(n/2)+\Theta(1)$ 的解是 $T(n)=\Theta(\lg n)$。(二分查找的描述见练习2.3-5)。

4.5-4 主方法能应用于递归式 $T(n)=4T(n/2)+n^2\lg n$ 吗? 请说明为什么可以或者为什么不可以。给出这个递归式的一个渐近上界。

***4.5-5** 考虑主定理情况3的一部分：对某个常数 $c<1$，正则条件 $af(n/b)\leqslant cf(n)$ 是否成立。给出一个例子，其中常数 $a\geqslant 1$，$b>1$ 且函数 $f(n)$ 满足主定理情况3中除正则条件外的所有条件。

* 4.6 证明主定理

本节给出主定理(定理4.1)的证明。但如果只是为了使用主定理，你不必理解这个证明。

证明分为两部分。第一部分分析主递归式(4.20)，为简单起见，假定 $T(n)$ 仅定义在 $b(b>1)$ 的幂上，即仅对 $n=1$，b，b^2，…定义。这一部分给出了为理解主定理是正确的所需的所有直觉知识。第二部分显示了如何将分析扩展到所有正整数 n；这一部分应用了处理向下和向上取整问题的数学技巧。

在本节中，我们有时会稍微滥用渐近符号，用来描述仅仅定义在 b 的幂上的函数的行为。回忆一下，渐近符号的定义要求对所有足够大的数都证明函数的界，而不是仅仅对 b 的幂。因为可以定义出仅仅应用于集合 $\{b^i: i=0, 1, 2, \cdots\}$ 上而不是所有非负数上新的渐近符号，所以这种滥用问题不大。

97

然而，当我们在一个局限的值域上使用渐近符号时，必须时刻小心，避免得到错误的结论。例如，对 n 是 2 的幂的情况证明 $T(n)=O(n)$并不保证 $T(n)=O(n)$。函数 $T(n)$可能是这样定义的：

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

此例中适用于所有 n 值的最佳上界为 $T(n)=O(n^2)$。由于可能导致这种严重后果，在并不绝对清楚应用环境的情况下，永远也不要在一个有限的值域上使用渐近符号。

4.6.1 对 b 的幂证明主定理

主定理证明的第一部分分析主定理的递归式(4.20)：

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

假定 n 是$b(b>1)$的幂，b 不一定是一个整数。我们将分析过程分解为三个引理。第一个引理将求解主递归式的问题归约为一个求和表达式的求值问题。第二个引理确定这个和式的界。第三个引理将前两个引理合二为一，证明 n 为 b 的幂的情况下的主定理。

引理 4.2 令 $a\geqslant 1$ 和 $b>1$ 是常数，$f(n)$是一个定义在 b 的幂上的非负函数。$T(n)$是定义在 b 的幂上的递归式：

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

其中 i 是正整数。那么

$$T(n)=\Theta(n^{\log_b a})+\sum_{j=0}^{\log_b n-1}a^j f(n/b^j) \tag{4.21}$$

证明 使用图 4-7 中的递归树。树的根结点的代价为 $f(n)$，它有 a 个孩子结点，每个的代价为 $f(n/b)$。(将 a 看做一个整数非常方便，当可视化递归树时尤其如此，但从数学角度并不要求这一点)。每个孩子结点又有 a 个孩子，使得在深度为 2 的层次上有 a^2 个结点，每个的代价为 $f(n/b^2)$。一般地，深度为 j 的层次上有a^j 个结点，每个的代价为 $f(n/b^j)$。每个叶结点的代价为 $T(1)=\Theta(1)$，深度为 $\log_b n$，因为 $n/b^{\log_b n}=1$。树中共有 $a^{\log_b n}=n^{\log_b a}$个叶结点。

98

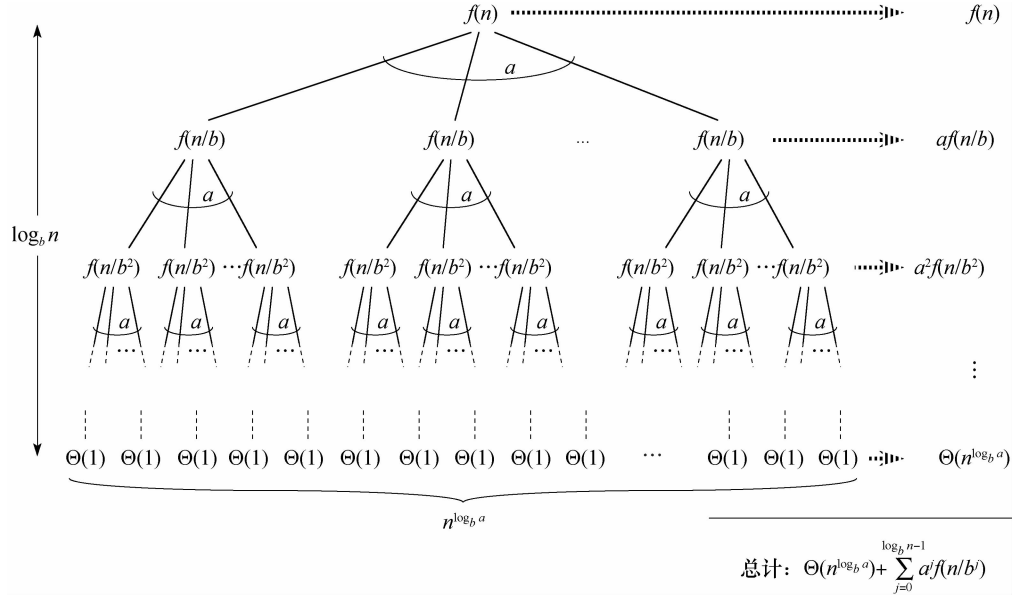


图 4-7 $T(n)=aT(n/b)+f(n)$的递归树。该树是一棵完全 a 叉树，高度为 $\log_b n$，共有 $n^{\log_b a}$ 个叶结点。每层结点的代价显示在右侧，代价和如公式(4.21)所示

我们将图 4-7 所示的递归树中的每层结点的代价求和，得到公式(4.21)。深度为 j 的所有内部结点的代价为 $a^j f(n/b^j)$，所以内部结点的总代价为：

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

在分治算法中，这个和表示分解子问题与合并子问题解的代价。所有叶结点的代价(表示完成所有 $n^{\log_b a}$ 个规模为 1 的子问题的代价)为 $\Theta(n^{\log_b a})$。 ■ 99

从递归树看，主定理的三种情况分别对应以下三种情况：(1)树的总代价由叶结点的代价决定；(2)树的总代价均匀分布在树的所有层次上；(3)树的总代价由根结点的代价决定。

公式(4.21)中的和式描述了分治算法中分解与合并步骤的代价。下一个定理则给出了这个和式增长速度的渐近界。

引理 4.3　令 $a \geqslant 1$ 和 $b > 1$ 是常数，$f(n)$是一个定义在 b 的幂上的非负函数。$g(n)$是定义在 b 的幂上的函数：

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \tag{4.22}$$

对 b 的幂，$g(n)$有如下渐近界：

1. 若对某个常数 $\varepsilon > 0$ 有 $f(n) = O(n^{\log_b a-\varepsilon})$，则 $g(n) = O(n^{\log_b a})$。
2. 若 $f(n) = \Theta(n^{\log_b a})$，则 $g(n) = \Theta(n^{\log_b a} \lg n)$。
3. 若对某个常数 $c < 1$ 和所有足够大的 n 有 $af(n/b) \leqslant cf(n)$，则 $g(n) = \Theta(f(n))$。

证明　对情况 1，我们有 $f(n) = O(n^{\log_b a-\varepsilon})$，这意味着 $f(n/b^j) = O((n/b^j)^{\log_b a-\varepsilon})$。代入公式(4.22)得

$$g(n) = O\Big(\sum_{j=0}^{\log_b n-1} a^j \Big(\frac{n}{b^j}\Big)^{\log_b a-\varepsilon}\Big) \tag{4.23}$$

对于 O 符号内的和式，通过提取因子并化简来求它的界，得到一个递增的几何级数：

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

100

由于 b 和 ε 是常数，因此可以将最后一个表达式重写为 $n^{\log_b a-\varepsilon}O(n^\varepsilon) = O(n^{\log_b a})$。用这个表达式代换公式(4.23)中的和式，得到 $g(n) = O(n^{\log_b a})$，因此情况 1 得证。

由于情况 2 假定 $f(n) = \Theta(n^{\log_b a})$，因此有 $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$。代入公式(4.22)得

$$g(n) = \Theta\Big(\sum_{j=0}^{\log_b n-1} a^j \Big(\frac{n}{b^j}\Big)^{\log_b a}\Big) \tag{4.24}$$

采用与情况 1 相同的方式，求出 Θ 符号内和式的界，但这次并未得到一个几何级数，而是发现和式的每一项都是相同的：

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

用这个表达式替换公式(4.24)中的和式，我们得到

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

情况 2 得证。

情况 3 的证明类似。由于 $f(n)$出现在 $g(n)$的定义(4.22)中，且 $g(n)$的所有项都是非负的，因此可以得出结论：对 b 的幂，$g(n) = \Omega(f(n))$。假定在这个引理中，对某个常数 $c < 1$ 和所有足够大的 n 有 $af(n/b) \leqslant cf(n)$。将这个假设改写为 $f(n/b) \leqslant (c/a)f(n)$并迭代 j 次，得到 $f(n/b^j) \leqslant (c/a)^j f(n)$，或等价地，$a^j f(n/b^j) \leqslant c^j f(n)$，其中假设进行迭代的值足够大。由于最后一个，也

就是最小的值为 n/b^{j-1}，因此假定 n/b^{j-1} 足够大就够了。

代入公式(4.22)并化简，我们得到一个几何级数，但与情况 1 证明中的几何级数不同，这次
101 得到的是递减的几何级数。使用一个 $O(1)$ 项来表示 n 足够大这个假设未覆盖的项：

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

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

因为 c 是一个常数。因此可以得到结论：对 b 的幂，$g(n)=\Theta(f(n))$。情况 3 得证，引理证毕。 ■

现在我们来证明 n 为 b 的幂的情况下的主定理。

引理 4.4 令 $a \geqslant 1$ 和 $b > 1$ 是常数，$f(n)$ 是一个定义在 b 的幂上的非负函数。$T(n)$ 是定义在 b 的幂上的递归式：

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

其中 i 是正整数。那么对 b 的幂，$T(n)$ 有如下渐近界：

1. 若对某个常数 $\varepsilon > 0$ 有 $f(n)=O(n^{\log_b a-\varepsilon})$，则 $T(n)=\Theta(n^{\log_b a})$。

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

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

证明 利用引理 4.3 中的界对引理 4.2 中的和式(4.21)进行求值。对情况 1，我们有
102

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

对于情况 2，

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

对于情况 3，

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

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

4.6.2 向下取整和向上取整

为了完成主定理的证明，我们必须将上述分析扩展到主递归式中使用向下取整和向上取整的情况，这样递归式就定义在所有整数上，而非仅仅针对 b 的幂。很容易获得如下递归式的下界：

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.25}$$

以及如下递归式的上界：

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.26}$$

因为我们可以对第一种情况应用下界 $\lceil n/b \rceil \geqslant n/b$ 来得到所需结果，对第二种情况应用上界 $\lfloor n/b \rfloor \leqslant n/b$。可以使用几乎一样的技术来处理递归式(4.26)的下界和递归式(4.25)的上界，因此我们只给出后一个界的证明。

对图 4-7 中的递归树进行修改，得到图 4-8 中的递归树。当沿着递归树向下时，我们得到如下递归调用的参数序列：

$$n$$
$$\lceil n/b \rceil$$
$$\lceil \lceil n/b \rceil / b \rceil$$
$$\lceil \lceil \lceil n/b \rceil / b \rceil / b \rceil$$
$$\vdots$$

用 n_j 表示序列中第 j 个元素，其中

$$n_j = \begin{cases} n & \text{若 } j = 0 \\ \lceil n_{j-1}/b \rceil & \text{若 } j > 0 \end{cases} \tag{4.27}$$

103

我们的第一个目标是确定 n_k 是常数时的深度 k。利用不等式 $\lceil x \rceil \leqslant x+1$，可得

$$\begin{aligned} n_0 &\leqslant n \\ n_1 &\leqslant \frac{n}{b} + 1 \\ n_2 &\leqslant \frac{n}{b^2} + \frac{1}{b} + 1 \\ n_3 &\leqslant \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1 \\ &\vdots \end{aligned}$$

一般地，我们有

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

令 $j = \lfloor \log_b n \rfloor$，可得

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

因此我们可以看到在深度 $\lfloor \log_b n \rfloor$，问题规模至多是常数。

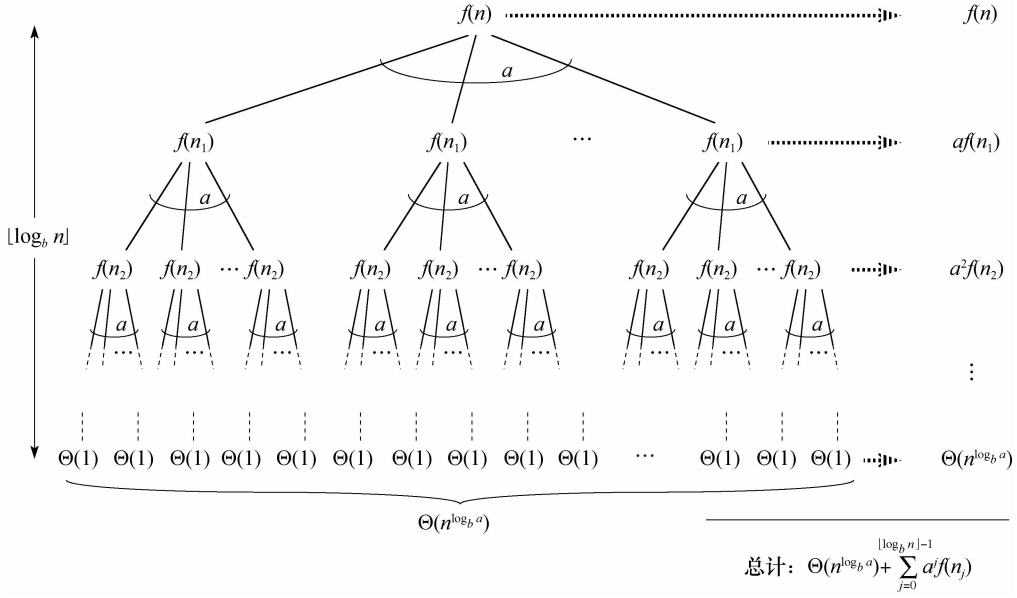


图 4-8　$T(n) = aT(\lceil n/b \rceil) + f(n)$ 的递归树。递归参数 n_j 的定义见公式(4.27)

从图 4-8 可以看出，

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.28}$$

除了 n 为任意整数，未局限为 b 的幂之外，这个公式与公式(4.21)几乎一样。

我们现在可以对公式(4.28)中的和式进行求值

$$g(n)=\sum_{j=0}^{\lfloor \log_b n \rfloor -1} a^j f(n_j) \tag{4.29}$$

方法与引理 4.3 的证明类似。我们从情况 3 开始，如果对 $n>b+b/(b-1)$，$af(\lceil n/b \rceil)\leqslant cf(n)$成立，其中 $c<1$ 是常数，则有 $a^j f(n_j)\leqslant c^j f(n)$。因此，我们可以像引理 4.3 的证明一样来对公式(4.29)的和式进行求值。对于情况 2，我们有 $f(n)=\Theta(n^{\log_b a})$。如果能证明 $f(n_j)=O(n^{\log_b a}/a^j)=O((n/b^j)^{\log_b a})$，则情况 2 的证明直接使用引理 4.3 证明的方法即可。观察到 $j\leqslant \lfloor \log_b n \rfloor$意味着 $b^j/n\leqslant 1$。界 $f(n)=O(n^{\log_b a})$意味着存在常数 $c>0$，使得对所有足够大的 n_j，

104 ~ 105

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

因为 $c(1+b/(b-1))^{\log_b a}$是常量。因此，情况 2 得证。情况 1 的证明几乎是一样的。关键是证明界 $f(n_j)=O((n/b^j)^{\log_b a-\varepsilon})$，这部分与情况 2 证明中的对应部分相似，尽管使用的代数方法更复杂些。

现在我们已经对所有整数 n 证明了主定理的上界。下界的证明类似。

练习

***4.6-1** 对 b 是正整数而非任意实数的情况，给出公式(4.27)中 n_j 的简单而准确的表达式。

***4.6-2** 证明：如果 $f(n)=\Theta(n^{\log_b a}\lg^k n)$，其中 $k\geqslant 0$，那么主递归式的解为 $T(n)=\Theta(n^{\log_b a}\lg^{k+1} n)$。为简单起见，假定 n 是 b 的幂。

***4.6-3** 证明：主定理中的情况 3 被过分强调了，从某种意义上来说，对某个常数 $c<1$，正则条件 $af(n/b)\leqslant cf(n)$成立本身就意味着存在常数 $\varepsilon>0$，使得 $f(n)=\Omega(n^{\log_b a+\varepsilon})$。

106

思考题

4-1 (*递归式例子*) 对下列每个递归式，给出 $T(n)$的渐近上界和下界。假定 $n\leqslant 2$ 时 $T(n)$是常数。给出尽量紧确的界，并验证其正确性。

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

b. $T(n)=T(7n/10)+n$

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

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

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

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

g. $T(n)=T(n-2)+n^2$

4-2 (*参数传递代价*) 我们有一个贯穿本书的假设——过程调用中的参数传递花费常量时间，即使传递一个 N 个元素的数组也是如此。在大多数系统中，这个假设是成立的，因为传递的是指向数组的指针，而非数组本身。本题讨论三种参数传递策略：

1. 数组通过指针来传递。时间$=\Theta(1)$。
2. 数组通过元素复制来传递。时间$=\Theta(N)$，其中 N 是数组的规模。
3. 传递数组时，只复制过程可能访问的子区域。若子数组 $A[p..q]$被传递，则时间$=\Theta(q-p+1)$。

a. 考虑在有序数组中查找元素的递归二分查找算法(参见练习 2.3-5)。分别给出上述三种参数传递策略下，二分查找最坏情况运行时间的递归式，并给出递归式解的好的上界。

令 N 为原问题的规模，n 为子问题的规模。

b. 对 2.3.1 节的 MERGE-SORT 算法重做(a)。 107

4-3 （更多的递归式例子） 对下列每个递归式，给出 $T(n)$的渐近上界和下界。假定对足够小的 n，$T(n)$是常数。给出尽量紧确的界，并验证其正确性。

a. $T(n)=4T(n/3)+n\lg n$

b. $T(n)=3T(n/3)+n/\lg n$

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

d. $T(n)=3T(n/3-2)+n/2$

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

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

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

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

i. $T(n)=T(n-2)+1/\lg n$

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

4-4 （斐波那契数） 本题讨论递归式(3.22)定义的斐波那契数的性质。我们将使用生成函数技术来求解斐波那契递归式。**生成函数**(又称为**形式幂级数**)$\mathcal{F}$定义为

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

其中 F_i 为第 i 个斐波那契数。

a. 证明：$\mathcal{F}(z)=z+z\mathcal{F}(z)+z^2\mathcal{F}(z)$。 108

b. 证明：

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

其中

$$\phi=\frac{1+\sqrt{5}}{2}=1.618\,03\cdots$$

$$\hat{\phi}=\frac{1-\sqrt{5}}{2}=-0.618\,03\cdots$$

c. 证明：

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

d. 利用(c)的结果证明：对 $i>0$，$F_i=\phi^i/\sqrt{5}$，结果舍入到最接近的整数。（提示：观察到 $|\hat{\phi}|<1$。）

4-5 （芯片检测） Diogenes 教授有 n 片可能完全一样的集成电路芯片，原理上可以用来相互检测。教授的测试夹具同时只能容纳两块芯片。当夹具装载上时，每块芯片都检测另一块，并报告它是好是坏。一块好的芯片总能准确报告另一块芯片的好坏，但教授不能信任坏芯片报告的结果。因此，4 种可能的测试结果如下：

芯片 A 的结果	芯片 B 的结果	结　论
B 是好的	A 是好的	两片都是好的，或都是坏的
B 是好的	A 是坏的	至少一块是坏的
B 是坏的	A 是好的	至少一块是坏的
B 是坏的	A 是坏的	至少一块是坏的

a. 证明：如果超过 $n/2$ 块芯片是坏的，使用任何基于这种逐对检测操作的策略，教授都不能确定哪些芯片是好的。假定坏芯片可以合谋欺骗教授。

109

b. 考虑从 n 块芯片中寻找一块好芯片的问题，假定超过 $n/2$ 块芯片是好的。证明：进行 $\lfloor n/2 \rfloor$ 次逐对检测足以将问题规模减半。

c. 假定超过 $n/2$ 块芯片是好的，证明：可以用 $\Theta(n)$ 次逐对检测找出好的芯片。给出描述检测次数的递归式，并求解它。

4-6 （Monge 阵列） 对一个 $m \times n$ 的实数阵列 A，若对所有满足 $1 \leqslant i < k \leqslant m$ 和 $1 \leqslant j < l \leqslant n$ 的 i，j，k 和 l 有

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

则称 A 是 **Monge 阵列**（Monge array）。换句话说，无论何时选出 Monge 阵列的两行和两列，对于交叉点上的 4 个元素，左上和右下两个元素之和总是小于等于左下和右上元素之和。例如，下面就是一个 Monge 阵列：

$$\begin{array}{rrrrr} 10 & 17 & 13 & 28 & 23 \\ 17 & 22 & 16 & 29 & 23 \\ 24 & 28 & 22 & 34 & 24 \\ 11 & 13 & 6 & 17 & 7 \\ 45 & 44 & 32 & 37 & 23 \\ 36 & 33 & 19 & 21 & 6 \\ 75 & 66 & 51 & 53 & 34 \end{array}$$

a. 证明：一个数组是 Monge 阵列当且仅当对所有 $i=1, 2, \cdots, m-1$ 和 $j=1, 2, \cdots, n-1$，有

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

（提示：对于“当”的部分，分别对行和列使用归纳法。）

b. 下面数组不是 Monge 阵列。改变一个元素使其变成 Monge 阵列。（提示：利用(a)的结果。）

$$\begin{array}{rrrr} 37 & 23 & 22 & 32 \\ 21 & 6 & 7 & 10 \\ 53 & 34 & 30 & 31 \\ 32 & 13 & 9 & 6 \\ 43 & 21 & 15 & 8 \end{array}$$

110

c. 令 $f(i)$ 表示第 i 行的最左最小元素的列下标。证明：对任意 $m \times n$ 的 Monge 阵列，$f(1) \leqslant f(2) \leqslant \cdots \leqslant f(m)$。

d. 下面是一个计算 $m \times n$ 的 Monge 阵列 A 每一行最左最小元素的分治算法的描述：

提取 A 的偶数行构造其子矩阵 A'。递归地确定 A' 每行的最左最小元素。然后计算 A 的奇数行的最左最小元素。

解释如何在 $O(m+n)$ 时间内计算 A 的奇数行的最左最小元素（在偶数行的最左最小元素已知的情况下）。

e. 给出(d)中描述的算法的运行时间的递归式。证明其解为 $O(m + n \log m)$。

本章注记

分治作为一种算法设计技术至少可以追溯到 1962 年 Karatsuba 和 Ofman 的一篇文章[194]。但是在这之前，分治技术已经有很好的应用，根据 Heideman、Johnson 和 Burrus 的论文[163]，卡尔·弗雷德里希·高斯在 1805 年设计了第一个快速傅里叶变换算法，而高斯的算法就是将问

题分解为更小的子问题，求解完子问题后将它们的解组合起来。

4.1 节中讨论的最大子数组问题是 Bently[43，第 7 章]研究的问题的一个简单变形。

Strassen 算法[325]发表于 1969 年，它的出现引起了很大的轰动。在此之前，很少人敢设想一个算法能渐近快于平凡算法 SQUARE-MATRIX-MULTIPLY。矩阵乘法的渐近上界自此被改进了。到目前为止，$n\times n$ 矩阵相乘的渐近复杂性最优的算法是 Coppersmith 和 Winograd[78]提出的，运行时间为 $O(n^{2.376})$。已知的最好的下界显然是 $\Omega(n^2)$(这是显然的下界，因为我们必须填写结果矩阵的 n^2 个元素)。

从实用的角度看，Strassen 算法通常并不是解决矩阵乘法的最好选择，原因有 4 个：

1. 隐藏在 Strassen 算法运行时间 $\Theta(n^{\lg 7})$ 中的常数因子比过程 SQUARE-MATRIX-MULTIPLY 的 $\Theta(n^3)$时间的常数因子大。

2. 对于稀疏矩阵，专用算法更快。 111

3. Strassen 算法的数值稳定性不如 SQUARE-MATRIX-MULTIPLY 那么好。换句话说，由于计算机计算非整数值时有限的精度，Strassen 算法累积的误差比 SQUARE-MATRIX-MULTIPLY 大。

4. 递归过程中生成的子矩阵消耗存储空间。

后两个原因在 1990 年左右得到了缓解。Higham[167]显示了数值稳定性上的差异被过分强调了；虽然 Strassen 算法对某些应用来说数值稳定性太差，但对其他应用来说，它所产生的数值误差还在可接受的范围内。Bailey、Lee 和 Simon[32]讨论了降低 Strassen 算法内存需求的技术。

在实际应用中，稠密矩阵的快速乘法程序在矩阵规模超过一个“交叉点”时使用 Strassen 算法，一旦子问题规模降低到交叉点之下，就切换到一个更简单的方法。交叉点的确切值高度依赖于具体系统。有一些分析统计操作次数，但忽略 CPU 缓存和流水线的影响，得出的交叉点低至 $n=8$(Higham[167])或 $n=12$(Huss-Lederman 等人[186])。D'Alberto 和 Nicolau[81]设计了一个自适应方法，在软件包安装完毕后通过基准测试确定交叉点。他们发现，在不同的系统上，交叉点的值从 $n=400$ 到 $n=2\,150$ 变化，而在几个系统中无法找到交叉点。

递归式的研究最早可追溯到 1202 年李奥纳多·斐波那契的工作，斐波那契数就是以他命名的。A. De Moivre 提出了用生成函数(参见思考题 4-4)求解递归式的方法。主方法改自 Bentley、Haken 和 Saxe[44]的方法，这篇文章提供了一种扩展方法，在练习 4.6-2 中已经得到验证。Knuth[209]和 Liu[237]展示了如何使用生成函数的方法求解线性递归式。Purdom 和 Brown 的论文[287]及 Graham、Knuth 和 Patashnik 的论文[152]包含了递归式求解的进一步讨论。

相对于主方法可求解的分治算法递归式，多名研究者，包括 Akra 和 Bazzi[13]、Roura[299]、Verma[346]及 Yap[360]，都给出过更一般的递归式的求解方法。我们介绍一下 Akra 和 Bazzi 的结果，这里给出的是 Leighton[228]修改后的版本。Akra-Bazzi 方法求解如下形式的递归式：

$$T(x)=\begin{cases}\Theta(1) & \text{若 } 1\leqslant x\leqslant x_0\\ \displaystyle\sum_{i=1}^{k}a_iT(b_ix)+f(x) & \text{若 } x>x_0\end{cases}\tag{4.30}$$

其中

- $x\geqslant 1$ 是一个实数，
- x_0 是一个常数，满足对 $i=1,2,\cdots,k$，$x_0\geqslant 1/b_i$ 且 $x_0\geqslant 1/(1-b_i)$，
- 对 $i=1,2,\cdots,k$，a_i 是一个正常数， 112
- 对 $i=1,2,\cdots,k$，b_i 是一个常数，范围在 $0<b_i<1$，
- $k\geqslant 1$ 是一个整数常数，且

- $f(x)$是一个非负函数，满足**多项式增长条件**：存在正常数 c_1 和 c_2，使得对所有 $x \geqslant 1$，$i=1, 2, \cdots, k$ 以及所有满足 $b_i x \leqslant u \leqslant x$ 的 u，有 $c_1 f(x) \leqslant f(u) \leqslant c_2 f(x)$。（若 $|f'(x)|$ 的上界是 x 的某个多项式，则 $f(x)$满足多项式增长条件。例如，对任意实常数 α 和 β，$f(x)=x^{\alpha} \lg^{\beta} x$ 满足此条件。）

虽然主定理不能应用于 $T(n)=T(\lfloor n/3 \rfloor)+T(\lfloor 2n/3 \rfloor)+O(n)$这样的递归式，但 Akra-Bazzi 方法可以。为了求解递归式(4.30)，我们首先寻找满足 $\sum_{i=1}^{k} a_i b_i^p = 1$ 的实数 p（这样的 p 总是存在的）。那么递归式的解为

$$T(n) = \Theta\left(x^p\left(1+\int_1^x \frac{f(u)}{u^{p+1}}\mathrm{d}u\right)\right)$$

Akra-Bazzi 方法可能有点儿难用，但它可以求解那些子问题划分不均衡的算法的递归式。主方法
113 很容易使用，但只能用于子问题规模相等的情况。

概率分析和随机算法

本章介绍**概率分析**和**随机算法**。如果你不熟悉概率论的基本知识，应先阅读附录C，复习这部分材料。我们将在本书中多次提到概率分析和随机算法。

5.1 雇用问题

假如你要雇用一名新的办公助理。你先前的雇用尝试都失败了，于是你决定找一个雇用代理。雇用代理每天给你推荐一个应聘者。你面试这个人，然后决定是否雇用他。你必须付给雇用代理一小笔费用，以便面试应聘者。然而要真的雇用一个应聘者需要花更多的钱，因为你必须辞掉目前的办公助理，还要付一大笔中介费给雇用代理。你承诺在任何时候，都要找最适合的人来担任这项职务。因此，你决定在面试完每个应聘者后，如果该应聘者比目前的办公助理更合适，就会辞掉当前的办公助理，然后聘用新的。你愿意为该策略付费，但希望能够估算该费用会是多少。

下面给出的HIRE-ASSISTANT过程以伪代码表示该雇用策略。假设应聘办公助理的候选人编号为1到n。该过程中假设你能在面试完应聘者i后，决定应聘者i是否是你目前见过的最佳人选。初始化时，该过程创建一个虚拟的应聘者，编号为0，他比其他所有应聘者都差。


```
HIRE-ASSISTANT(n)
1  best = 0            // candidate 0 is a least-qualified dummy candidate
2  for i = 1 to n
3      interview candidate i
4      if candidate i is better than candidate best
5          best = i
6          hire candidate i
```

这个问题的费用模型与第2章中描述的模型不同。我们关注的不是HIRE-ASSISTANT的执行时间，而是面试和雇用所产生的费用。表面上看起来，分析这个算法的费用与分析归并排序等的运行时间有很大不同。然而，我们在分析费用或者分析运行时间时，所采用的分析技术却是相同的。在任何情形中，我们都是在计算特定基本操作的执行次数。

面试的费用较低，比如为c_i，然而雇用的费用较高，设为c_h。假设m是雇用的人数，那么该算法的总费用就是$O(c_in+c_hm)$。不管雇用多少人，我们总会面试n个应聘者，于是面试产生的费用总是c_in。因此，我们只关注于分析c_hm，即雇用的费用。这个量在该算法的每次执行中都不同。

这个场景用来作为一般计算范式的模型。我们通常通过检查序列中的每个成员，并且维护一个当前的“获胜者”，来找出序列中的最大值或最小值。这个雇用问题对当前获胜成员的更新频率建立模型。

最坏情形分析

在最坏情况下，我们实际上雇用了每个面试的应聘者。当应聘者质量按出现的次序严格递增时，这种情况就会出现，此时雇用了n次，总的费用是$O(c_hn)$。

当然，应聘者并非总以质量递增的次序出现。事实上，我们既不知道他们出现的次序，也不能控制这个次序。因此，很自然地会问在一种典型或者平均的情形下，会有什么发生。

概率分析

概率分析是在问题分析中应用概率的理念。大多数情况下，我们采用概率分析来分析一个
115 算法的运行时间，有时也用它来分析其他的量，例如，过程 HIRE-ASSISTANT 中的雇用费用。为了进行概率分析，我们必须使用或者假设关于输入的分布。然后分析该算法，计算出一个平均情形下的运行时间，其中我们对所有可能的输入分布取平均值。因此，实际上，我们对所有可能输入产生的运行时间取平均。当报告此种类型的运行时间时，我们称其为**平均情况运行时间**。

我们在确定输入分布时必须非常小心。对于某些问题，我们可以对所有可能的输入集合做某种假定，然后采用概率分析来设计一个高效算法，并加深对问题的认识。对于其他一些问题，我们不能描述一个合理的输入分布，此时就不能采用概率分析。

在雇用问题中，我们可以假设应聘者以随机顺序出现。这一假设意味着什么？假定可以对任何两个应聘者进行比较，并决定哪一个更有资格；也就是说，所有应聘者存在一个全序关系（全序的定义可参见附录 B）。因此，可以使用从 1 到 n 的唯一号码对应聘者排列名次，用 $rank(i)$ 表示应聘者 i 的名次，并照常约定一个较高名次对应一个更好的应聘者。有序序列 $\langle rank(1), rank(2), \cdots, rank(n)\rangle$ 是序列 $\langle 1, 2, \cdots, n\rangle$ 的一个排列。称应聘者以随机顺序出现，等价于称这个排名列表是数字 1 到 n 的 $n!$ 种排列表中的任一个。或者，我们也称这些排名构成一个**均匀随机排列**；也就是说，在 $n!$ 种可能的排列中，每一种以等概率出现。

5.2 节包含这个雇用问题的一个概率分析。

随机算法

为了利用概率分析，我们需要了解关于输入分布的一些信息。在许多情况下，我们对输入分布了解很少。即使知道输入分布的某些信息，也可能无法从计算上对该分布知识建立模型。然而，我们通过使一个算法中某部分的行为随机化，常可以利用概率和随机性作为算法设计和分析的工具。

在雇用问题中，看起来应聘者好像以随机顺序出现，但我们无法知道是否的确如此。因此，为了设计雇用问题的一个随机算法，我们必须对面试应聘者的次序有更大的控制。所以，稍稍改变这个模型。假设雇用代理有 n 个应聘者，而且他们事先给我们一份应聘者名单。每天随机选择
116 某个应聘者来面试。尽管除了应聘者的名字外对其他信息一无所知，但我们已经做了一个显著的改变。不是像以前依赖于猜测应聘者以随机次序出现，取而代之，我们获得了对流程的控制并且加强了随机次序。

更一般地，如果一个算法的行为不仅由输入决定，而且也由**随机数生成器**（random-number generator）产生的数值决定，则称这个算法是**随机的**（randomized）。我们将假设有一个可以自由使用的随机数生成器 RANDOM。调用 RANDOM(a，b)将返回一个介于 a 和 b 之间的整数，并且每个整数以等概率出现。例如，RANDOM(0，1)产生 0 的概率是 1/2，产生 1 的概率也是1/2。调用 RANDOM(3，7)将返回 3、4、5、6 或 7，每个出现的概率都是 1/5。每次 RANDOM 返回的整数独立于前面调用的返回值。可以将 RANDOM 想象成掷一个 $(b-a+1)$ 面的骰子，获得出现的点数。（在实践中，大多数编程环境会提供一个**伪随机数生成器**，它是一个确定性算法，返回值在统计上看起来是随机的。）

当分析一个随机算法的运行时间时，我们以运行时间的期望值衡量，其中输入值由随机数生成器产生。我们将一个随机算法的运行时间称为**期望运行时间**，以此来区分这类算法和那些输入是随机的算法。一般而言，当概率分布是在算法的输入上时，我们讨论的是平均情况运行时间；当算法本身做出随机选择时，我们讨论其期望运行时间。

练习

5.1-1 证明：假设在过程 HIRE-ASSISTANT 的第 4 行中，我们总能决定哪一个应聘者最佳，

则意味着我们知道应聘者排名的全部次序。

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

*5.1-3　假设你希望以 1/2 的概率输出 0 与 1。你可以自由使用一个输出 0 或 1 的过程 BIASED-RANDOM。它以某概率 p 输出 1，概率 $1-p$ 输出 0，其中 $0<p<1$，但是 p 的值未知。请给出一个利用 BIASED-RANDOM 作为子程序的算法，返回一个无偏的结果，能以概率 1/2 返回 0，以概率 1/2 返回 1。作为 p 的函数，你的算法的期望运行时间是多少？ 117

5.2　指示器随机变量

为了分析雇用问题在内的许多算法，我们采用指示器随机变量(indicator random variable)。它为概率与期望之间的转换提供了一个便利的方法。给定一个样本空间 S 和一个事件 A，那么事件 A 对应的**指示器随机变量** $\mathrm{I}\{A\}$ 定义为：

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

举一个简单的例子，我们来确定抛掷一枚标准硬币时正面朝上的期望次数。样本空间为 $S=\{H, T\}$，其中 $\Pr\{H\}=\Pr\{T\}=1/2$。接下来定义一个指示器随机变量 X_H，对应于硬币正面朝上的事件 H。这个变量计数抛硬币时正面朝上的次数，如果正面朝上则值为 1，否则为 0。我们记成：

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

在一次抛掷硬币时，正面朝上的期望次数就是指示器变量 X_H 的期望值：

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

因此抛掷一枚标准硬币时，正面朝上的期望次数是 1/2。如下面引理所示，一个事件 A 对应的指示器随机变量的期望值等于事件 A 发生的概率。

引理 5.1　给定一个样本空间 S 和 S 中的一个事件 A，设 $X_A=\mathrm{I}\{A\}$，那么 $\mathrm{E}[X_A]=\Pr\{A\}$。 118

证明　由等式(5.1)指示器随机变量的定义，以及期望值的定义，我们有

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

其中 $\overline{A}$ 表示 $S-A$，即 A 的补。 ■

虽然指示器随机变量看起来很麻烦，比如在计算单枚硬币一次投掷的正面次数期望时，但是它在分析重复随机试验时是有用的。例如，指示器随机变量为我们求等式(C.37)的结果提供了一个简单方法。在这个等式中，我们分别考虑出现 0 个、1 个、2 个…正面朝上的概率，以计算抛 n 次硬币时正面朝上的次数。等式(C.38)中给出了简单方法，隐含使用了指示器随机变量。为使讨论更清楚，我们设指示器随机变量 X_i 对应第 i 次抛硬币时正面朝上的事件：$X_i=\mathrm{I}\{$第 i 次抛掷时出现事件 $H\}$。设随机变量 X 表示 n 次抛硬币中出现正面的总次数，于是

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

我们希望计算正面朝上次数的期望，所以对上面等式两边取期望，得到

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

上面等式给出了 n 个指示器随机变量总和的期望值。由引理 5.1，我们容易计算出每个随机变量的期望值。根据反映期望线性性质的等式(C.21)，容易计算出总和的期望值：它等于 n 个随机变量期望值的总和。期望的线性性质利用指示器随机变量作为一种强大的分析技术；当随机变量

119 之间存在依赖关系时也成立。现在我们可以轻松地计算正面出现次数的期望：

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

因此，和等式(C.37)中用到的方法相比，指示器随机变量极大地简化了计算过程。我们将在本书中一直采用指示器随机变量。

用指示器随机变量分析雇用问题

返回到雇用问题上来。我们希望计算雇用一个新的办公助理的期望次数。为了利用概率分析，假设应聘者以随机顺序出现，如前一节所述。(我们将看到在5.3节如何去除这个假设。)设X是一个随机变量，其值等于我们雇用一个新办公助理的次数。然后，应用等式(C.20)中期望值的定义，得到

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

但是这种计算会很麻烦。取而代之，我们将采用指示器随机变量来大大简化计算。

为了利用指示器随机变量，我们不是通过定义与雇用一个新办公助理所需次数对应的一个变量来计算$\mathrm{E}[X]$，而是定义n个变量，与每个应聘者是否被雇用对应。特别地，假设X_i对应于第i个应聘者被雇用该事件的指示器随机变量。因而，

$$X_i = \mathrm{I}\{\text{应聘者 } i \text{ 被雇用}\} = \begin{cases} 1 & \text{如果应聘者 } i \text{ 被雇用} \\ 0 & \text{如果应聘者 } i \text{ 不被雇用} \end{cases}$$

以及

120
$$X = X_1 + X_2 + \cdots + X_n \tag{5.2}$$

根据引理5.1，我们有

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

因此必须计算HIRE-ASSISTANT中第5～6行被执行的概率。

在第6行中，应聘者i被雇用，正好应聘者i比从1到$i-1$的每一个应聘者优秀。因为我们已经假设应聘者以随机顺序出现，所以前i个应聘者也以随机次序出现。这些前i个应聘者中的任意一个都等可能地是目前最有资格的。应聘者i比应聘者1到$i-1$更有资格的概率是$1/i$，因而也以$1/i$的概率被雇用。由引理5.1，可得

$$\mathrm{E}[X_i] = 1/i \tag{5.3}$$

现在可以计算$\mathrm{E}[X]$：

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(根据等式(5.2))} \qquad (5.4) \\ &= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(根据期望的线性性质)} \\ &= \sum_{i=1}^{n} 1/i && \text{(根据等式(5.3))} \\ &= \ln n + O(1) && \text{(根据等式(A.7))} \qquad (5.5) \end{aligned}$$

尽管我们面试了n个人，但平均起来，实际上大约只雇用他们之中的$\ln n$个人。我们用下面的引理来总结这个结果。

引理5.2 *假设应聘者以随机次序出现，算法HIRE-ASSISTANT总的雇用费用平均情形下为$O(c_h \ln n)$。*

证明 根据雇用费用的定义和等式(5.5)，可以立即推出这个界，说明雇用的人数期望值大约是$\lg n$。 ■

121 平均情形下的雇用费用比最坏情况下的雇用费用$O(c_h n)$有了很大的改进。

练习

5.2-1　在 HIRE-ASSISTANT 中，假设应聘者以随机顺序出现，你正好雇用一次的概率是多少？正好雇用 n 次的概率是多少？

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

5.2-3　利用指示器随机变量来计算掷 n 个骰子之和的期望值。

5.2-4　利用指示器随机变量来解如下的**帽子核对问题**(hat-heck problem)：n 位顾客，他们每个人给餐厅核对帽子的服务生一顶帽子。服务生以随机顺序将帽子归还给顾客。请问拿到自己帽子的客户的期望数是多少？

5.2-5　设 $A[1..n]$是由 n 个不同数构成的数列。如果 $i<j$ 且 $A[i]>A[j]$，则称(i, j)对为 A 的一个**逆序对**(inversion)。(参看思考题 2-4 中更多关于逆序对的例子。)假设 A 的元素构成$\langle 1, 2, \cdots, n\rangle$上的一个均匀随机排列。请用指示器随机变量来计算其中逆序对的数目期望。

5.3　随机算法

在前面一节中，我们已说明了输入的分布是如何有助于分析一个算法的平均情况行为。许多时候，我们无法得知输入分布的信息，因而阻碍了平均情况分析。如 5.1 节中所提及，我们也许可以采用一个随机算法。

对于诸如雇用问题之类的问题，其中假设输入的所有排列等可能出现往往有益，通过概率分析可以指导设计一个随机算法。我们不是假设输入的一个分布，而是设定一个分布。特别地，在算法运行前，先随机地排列应聘者，以加强所有排列都是等可能出现的性质。尽管已经修改了这个算法，我们仍希望雇用一个新的办公助理大约需要 $\ln n$ 次期望值。但是现在我们期望对于所 122
有的输入它都是这种情况，而不是对于一个具有特别分布的输入。

我们来进一步探索概率分析和随机算法之间的区别。在 5.2 节中，我们断言：如果应聘者以随机顺序出现，则聘用一个新办公助理的期望次数大约是 $\ln n$。注意，这个算法是确定性的；对于任何特定输入，雇用一个新办公助理的次数始终相同。此外，我们雇用一个新办公助理的次数将因输入的不同而不同，而且依赖于各个应聘者的排名。既然次数仅依赖于应聘者的排名，我们可以使用应聘者的有序排名列表来代表一个特定的输入，例如$\langle rank(1), rank(2), \cdots, rank(n)\rangle$。给定排名列表 $A_1=\langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle$，一个新的办公助理会雇用 10 次，因为每一个后来应聘者都优于前一个，在算法的每次迭代中，第 5～6 行都要被执行。给定排名列表 $A_2=\langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1\rangle$，一个新的办公助理只雇用一次，在第一次迭代中。给定排名序列 $A_3=\langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6\rangle$，一个新的办公助理会雇用三次，即面试排名为 5、8 和 10 的 3 位应聘者。回顾一下，算法的费用依赖于雇用一个新办公助理的次数。我们可以看到，有昂贵的输入(如 A_1)、不贵的输入(如 A_2)，以及适中贵的输入(如 A_3)。

另外，考虑先对应聘者进行排列，然后确定最佳应聘者的随机算法。此时，我们让随机发生在算法上，而不是在输入分布上。给定一个输入，如上面的 A_3，我们无法说出最大值会被更新多少次，因为此变量在每次运行该算法时都不同。第一次在 A_3 上运行这个算法时，可能会产生排列 A_1 并执行 10 次更新；但第二次运行算法时，可能会产生排列 A_2 并只执行 1 次更新。第三次执行时，可能会产生其他次数的更新。每次运行这个算法时，执行依赖于随机选择，而且很可能和前一次算法的执行不同。对于该算法及许多其他的随机算法，*没有特别的输入会引出它的最坏情况行为*。即使你最坏的敌人也无法产生最坏的输入数组，因为随机排列使得输入次序不再相关。只有在随机数生成器产生一个“不走运”的排列时，随机算法才会运行得很差。

123 对于雇用问题，代码中唯一需要改变的是随机地变换应聘者序列。

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

通过这个简单改变，我们已经建立了一个随机算法，其性能和假设应聘者以随机次序出现所得结果是匹配的。

引理 5.3 过程 RANDOMIZED-HIRE-ASSISTANT 的雇用费用期望是 $O(c_h \ln n)$。

证明 对输入数组进行变换后，我们已经达到了和 HIRE-ASSISTANT 概率分析时相同的情况。 ■

比较引理 5.2 和引理 5.3 突出了概率分析和随机算法的差别。在引理 5.2 中，我们在输入上做了一个假设。在引理 5.3 中，我们没有做这种假设，尽管随机化输入会花费一些额外时间。为了保持术语的一致性，我们用平均情形下的雇用费用来表达引理 5.2，而用期望雇用费用来表达引理 5.3。在本节余下部分里，我们讨论关于随机排列输入的一些议题。

随机排列数组

很多随机算法通过对给定的输入变换排列以使输入随机化。(还有其他使用随机化的方法。)这里，我们将讨论两种随机化方法。不失一般性，假设给定一个数组 A，包含元素 1 到 n。我们的目标是构造这个数组的一个随机排列。

一个通常的方法是为数组的每个元素 $A[i]$赋一个随机的优先级 $P[i]$，然后依据优先级对数组 A 中的元素进行排序。例如，如果初始数组 $A=\langle 1, 2, 3, 4\rangle$，随机选择的优先级 $P=\langle 36, 3, 62, 19\rangle$，则将产生一个数组 $B=\langle 2, 4, 1, 3\rangle$，因为第 2 个优先级最小，接下来是第 4 个，
124 然后第 1 个，最后第 3 个。我们称这个过程为 PERMUTE-BY-SORTING：

```
PERMUTE-BY-SORTING(A)
1  n = A.length
2  let P[1..n] be a new array
3  for i = 1 to n
4      P[i] = RANDOM(1, n^3)
5  sort A, using P as sort keys
```

第 4 行选取一个在 $1\sim n^3$ 之间的随机数。我们使用范围 $1\sim n^3$ 是为了让 P 中所有优先级尽可能唯一。(练习 5.3-5 要求读者证明所有元素都唯一的概率至少是 $1-1/n$，练习 5.3-6 问如何在两个或更多优先级相同的情况下，实现这个算法。)我们假设所有的优先级都唯一。

这个过程中耗时的步骤是第 5 行的排序。正如我们将在第 8 章看到的那样，如果使用比较排序，排序将花费 $\Omega(n \lg n)$时间。我们可以达到这个下界，因为我们已经看到归并排序时间代价为 $\Theta(n \lg n)$。(我们将在第二部分看到，其他的比较排序花费时间代价为 $\Theta(n \lg n)$。练习 8.3-4 要求读者解决一个非常类似的问题，在 $O(n)$时间内对 $0\sim n^3-1$ 范围之内的整数排序。)排序以后，如果 $P[i]$是第 j 个最小的优先级，那么 $A[i]$将出现在输出位置 j 上。用这种方式，我们得到了一个排列。还需要证明这个过程能产生一个**均匀随机排列**，即该过程等可能地产生数字 $1\sim n$ 的每一种排列。

引理 5.4 假设所有优先级都不同，则过程 PERMUTE-BY-SORTING 产生输入的均匀随机

排列。

证明　我们从考虑每个元素 $A[i]$分配到第 i 个最小优先级的特殊排列开始，并说明这个排列正好发生的概率是 $1/n!$。对 $i=1, 2, \cdots, n$，设 E_i 代表元素 $A[i]$分配到第 i 个最小优先级的事件。然后我们想计算对所有的 i，事件 E_i 发生的概率，即

$$\Pr\{E_1 \cap E_2 \cap E_3 \cap \cdots \cap E_{n-1} \cap E_n\}$$

运用练习 C.2-5，这个概率等于

$$\Pr\{E_1\} \cdot \Pr\{E_2 \mid E_1\} \cdot \Pr\{E_3 \mid E_2 \cap E_1\} \cdot \Pr\{E_4 \mid E_3 \cap E_2 \cap E_1\}$$
$$\cdots \Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap \cdots \cap E_1\} \cdots \Pr\{E_n \mid E_{n-1} \cap \cdots \cap E_1\}$$

因为 $\Pr\{E_1\}$是从一个 n 元素的集合中随机选取的优先级最小的概率，所以有 $\Pr\{E_1\}=1/n$。接下来，我们观察到 $\Pr\{E_2 \mid E_1\}=1/(n-1)$，因为假定元素 $A[1]$有最小的优先级，余下来的 $n-1$ 个 125
元素都有相等的可能成为第二小的优先级别。一般地，对 $i=2, 3, \cdots, n$，我们有 $\Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap \cdots \cap E_1\}=1/(n-i+1)$。因为给定元素 $A[1]$到 $A[i-1]$(按顺序)有前 $i-1$ 小的优先级，剩下的 $n-(i-1)$个元素中，每一个都等可能具有第 i 小优先级。所以有

$$\Pr\{E_1 \cap E_2 \cap E_3 \cap \cdots \cap E_{n-1} \cap E_n\} = \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\cdots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right) = \frac{1}{n!}$$

并且我们已说明，获得等同排列的概率是 $1/n!$。

我们可以扩展这个证明，使其对任何优先级的排列都有效。考虑集合$\{1, 2, \cdots, n\}$的任意一个确定排列 $\sigma=\langle\sigma(1), \sigma(2), \cdots, \sigma(n)\rangle$。我们用 r_i 表示赋予元素 $A[i]$优先级的排名，其中优先级第 j 小的元素名次为 j。如果定义 E_i 为元素 $A[i]$分配到优先级第 $\sigma(i)$小的事件，或者 $r_i=\sigma(i)$，则同样的证明仍适用。因此，如果要计算得到任何特定排列的概率，该计算与前面的计算完全相同，于是得到此排列的概率也是 $1/n!$。 ■

你可能会这样想，要证明一个排列是均匀随机排列，只要证明对于每个元素 $A[i]$，它排在位置 j 的概率是 $1/n$。练习 5.3-4 证明这个弱条件实际上并不充分。

产生随机排列的一个更好方法是原址排列给定数组。过程 RANDOMIZE-IN-PLACE 在 $O(n)$ 时间内完成。在进行第 i 次迭代时，元素 $A[i]$是从元素 $A[i]$到 $A[n]$中随机选取的。第 i 次迭代以后，$A[i]$不再改变。

```
RANDOMIZE-IN-PLACE(A)
1  n = A.length
2  for i = 1 to n
3      swap A[i] with A[RANDOM(i, n)]
```

我们将使用循环不变式来证明过程 RANDOMIZE-IN-PLACE 能产生一个均匀随机排列。一个具有 n 个元素的***k* 排列**(k-permutation)是包含这 n 个元素中的 k 个元素的序列，并且不重复(参见附录 C)。一共有 $n!/(n-k)!$ 种可能的 k 排列。 126

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

证明　我们使用下面的循环不变式：

在第 2～3 行 **for** 循环的第 i 次迭代以前，对每个可能的$(i-1)$排列，子数组 $A[1..i-1]$包含这个$(i-1)$排列的概率是$(n-i+1)!/n!$。

我们需要说明这个不变式在第 1 次循环迭代以前为真，循环的每次迭代能够维持此不变式，并且当循环终止时，这个不变式提供一个有用的性质来说明正确性。

初始化：考虑正好在第 1 次循环迭代以前的情况，此时 $i=1$。由循环不变式可知，对每个可能的 0 排列，子数组 $A[1..0]$包含这个 0 排列的概率是$(n-i+1)!/n! = n!/n! = 1$。子数组 $A[1..0]$是一个空的子数组，并且 0 排列也没有元素。因而，$A[1..0]$包含任何 0 排列的概率是

1，在第 1 次循环迭代以前循环不变式成立。

保持：我们假设在第 i 次迭代之前，每种可能的$(i-1)$排列出现在子数组 $A[1..i-1]$中的概率是$(n-i+1)!/n!$，我们要说明在第 i 次迭代以后，每种可能的 i 排列出现在子数组 $A[1..i]$中的概率是$(n-i)!/n!$。下一次迭代 i 累加后，还将保持这个循环不变式。

我们来检查第 i 次迭代。考虑一个特殊的 i 排列，并以$\langle x_1, x_2, \cdots, x_i\rangle$来表示其中的元素。这个排列中包含一个$(i-1)$排列$\langle x_1, x_2, \cdots, x_{i-1}\rangle$，后面接着算法在 $A[i]$里放置的值 x_i。设 E_1 表示前 $i-1$ 次迭代已经在 $A[1..i-1]$中构造了特殊$(i-1)$排列的事件。根据循环不变式，$\Pr\{E_1\}=(n-i+1)!/n!$。设 E_2 表示第 i 次迭代在位置$A[i]$放置 x_i 的事件。当 E_1 和 E_2 恰好都发生时，i 排列$\langle x_1, \cdots, x_i\rangle$出现在 $A[1..i]$中，因此，我们希望计算 $\Pr\{E_2 \cap E_1\}$。利用等式(C.14)，我们有

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

概率 $\Pr\{E_2 \mid E_1\}$等于 $1/(n-i+1)$，因为在算法第 3 行，从 $A[i..n]$的 $n-i+1$ 个值中随机选取
127 x_i。因此，我们有

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

终止：终止时，$i=n+1$，子数组 $A[1..n]$是一个给定 n 排列的概率为$(n-(n+1)+1)/n! = 0!/n! = 1/n!$。

因此，RANDOMIZE-IN-PLACE 产生一个均匀随机排列。 ■

一个随机算法通常是解决一个问题最简单、最有效的方法。我们将在本书中偶尔用到随机算法。

练习

5.3-1 Marceau 教授不同意引理 5.5 证明中使用的循环不变式。他对第 1 次迭代之前循环不变式是否为真提出质疑。他的理由是，我们可以很容易宣称一个空数组不包含 0 排列。因此，一个空的子数组包含一个 0 排列的概率应是 0，从而第 1 次迭代之前循环不变式无效。请重写过程 RANDOMIZE-IN-PLACE，使得相关循环不变式适用于第 1 次迭代之前的非空子数组，并为你的过程修改引理 5.5 的证明。

5.3-2 Kelp 教授决定写一个过程来随机产生除恒等排列(identity permutation)外的任意排列。他提出了如下过程：

```
PERMUTE-WITHOUT-IDENTITY(A)
1  n = A.length
2  for i = 1 to n - 1
3      swap A[i] with A[RANDOM(i + 1, n)]
```

这段代码实现了 Kelp 教授的意图吗？

5.3-3 假设我们不是将元素 $A[i]$与子数组 $A[i..n]$中的一个随机元素交换，而是将它与数组任
128 何位置上的随机元素交换：

```
PERMUTE-WITH-ALL(A)
1  n = A.length
2  for i = 1 to n
3      swap A[i] with A[RANDOM(1, n)]
```

这段代码会产生一个均匀随机排列吗？为什么会或为什么不会？

5.3-4 Armstrong 教授建议用下面的过程来产生一个均匀随机排列：

```
PERMUTE-BY-CYCLIC(A)
1  n = A.length
2  let B[1..n] be a new array
3  offset = RANDOM(1, n)
4  for i = 1 to n
5      dest = i + offset
6      if dest > n
7          dest = dest − n
8      B[dest] = A[i]
9  return B
```

请说明每个元素 $A[i]$出现在 B 中任何特定位置的概率是 $1/n$。然后通过说明排列结果不是均匀随机排列，表明 Armstrong 教授错了。

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

5.3-6 请解释如何实现算法 PERMUTE-BY-SORTING，以处理两个或更多优先级相同的情形。也就是说，即使有两个或更多优先级相同，你的算法也应该产生一个均匀随机排列。

5.3-7 假设我们希望创建集合$\{1, 2, 3, \cdots, n\}$的一个**随机样本**，即一个具有 m 个元素的集合 S，其中 $0 \leqslant m \leqslant n$，使得每个 m 集合能够等可能地创建。一种方法是对 $i=1, 2, \cdots, n$ 设$A[i]=i$，调用 RANDOMIZE-IN-PLACE(A)，然后取最前面的 m 个数组元素。这种方法会对 RANDOM 过程调用 n 次。如果 n 比 m 大很多，我们能够创建一个随机样本，只对 RANDOM 调用更少的次数。请说明下面的递归过程返回$\{1, 2, 3, \cdots, n\}$的一个随机 m 子集 S，其中每个 m 子集是等可能的，然而只对 RANDOM 调用 m 次。 129

```
RANDOM-SAMPLE(m, n)
1  if m == 0
2      return ∅
3  else S = RANDOM-SAMPLE(m − 1, n − 1)
4      i = RANDOM(1, n)
5      if i ∈ S
6          S = S ∪ {n}
7      else S = S ∪ {i}
8      return S
```

*5.4　概率分析和指示器随机变量的进一步使用

本节通过 4 个例子进一步阐释概率分析。第 1 个例子确定在一个有 k 个人的屋子中，某两个人生日相同的概率。第 2 个例子讨论把球随机投入箱子的问题。第 3 个例子探究抛硬币时连续出现正面的情况。最后一个例子分析雇用问题的一个变形，其中你必须在没有面试所有的应聘者时做出决定。

5.4.1　生日悖论

我们的第一个例子是**生日悖论**。一个屋子里人数必须要达到多少人，才能使其中两人生日相同的机会达到 50%？这个问题的答案是一个很小的数值，让人吃惊。下面我们将看到，所出现的悖论在于，这个数目实际上远小于一年中的天数，甚至不足一年天数的一半。

为了回答这个问题，我们用整数 1，2，…，k 对屋子里的人编号，其中 k 是屋子里的总人数。另外，我们不考虑闰年的情况，并且假设所有年份都有 $n=365$ 天。对于 $i=1, 2, \cdots, k$，设 b_i 表示编号为 i 的人的生日，其中 $1 \leqslant b_i \leqslant n$。还假设生日均匀分布在一年的 n 天中，因此对

$i=1,2,\cdots,k$ 和 $r=1,2,\cdots,n$，$\Pr\{b_i=r\}=1/n$。

两个人 i 和 j 的生日正好相同的概率依赖于生日的随机选择是否独立。从现在开始，假设生
130 日是独立的，于是 i 和 j 的生日都落在同一日 r 上的概率是

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

这样，他们的生日落在同一天的概率是

$$\Pr\{b_i=b_j\}=\sum_{r=1}^{n}\Pr\{b_i=r \text{ 且 } b_j=r\}=\sum_{r=1}^{n}(1/n^2)=1/n \tag{5.6}$$

更直观地说，一旦选定 b_i，b_j 被选在同一天的概率是 $1/n$。因此，i 和 j 有相同生日的概率与他们其中一个的生日落在给定一天的概率相同。然而需要注意，这个巧合依赖于各人的生日是独立的这个假设。

我们可以通过考察一个事件补的方法，来分析 k 个人中至少有两人生日相同的概率。至少有两个人生日相同的概率等于 1 减去所有人生日都不相同的概率。k 个人生日互不相同的事件为

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

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

$$\Pr\{B_k\}=\Pr\{B_{k-1}\}\Pr\{A_k\mid B_{k-1}\} \tag{5.7}$$

其中取 $\Pr\{B_1\}=\Pr\{A_1\}=1$ 作为初始条件。换句话说，对 $i=1,2,\cdots,k-1$，假设 $b_1,b_2,\cdots,b_{k-1}$ 两两不同，那么 $b_1,b_2,\cdots,b_k$ 两两不同的概率等于 $b_1,b_2,\cdots,b_{k-1}$ 两两不同的概率乘以 $i=1,2,\cdots,k-1$ 时 $b_k\neq b_i$ 的概率。

如果 $b_1,b_2,\cdots,b_{k-1}$ 两两不同，对于 $i=1,2,\cdots,k-1$，$b_k\neq b_i$ 的条件概率是 $\Pr\{A_k\mid B_{k-1}\}=$
131 $(n-k+1)/n$，这是因为 n 天中有 $n-(k-1)$ 天没被占用。我们反复应用递归式(5.7)得到

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

由不等式(3.12)，$1+x\leqslant e^x$，我们得出

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

当 $-k(k-1)/2n\leqslant\ln(1/2)$ 时成立。当 $k(k-1)\geqslant 2n\ln 2$，或者，解二次方程，当 $k\geqslant(1+\sqrt{1+(8\ln 2)n})/2$ 时，所有 k 个生日两两不同的概率至多是 1/2。当 $n=365$ 时，必有 $k\geqslant 23$。因而，如果至少有 23 个人在一间屋子里，那么至少有两个人生日相同的概率至少是 1/2。在火星上，一年有 669 个火星日，所以达到相同效果须有 31 个火星人。

采用指示器随机变量的一个分析

我们可以利用指示器随机变量给出生日悖论的一个简单而近似的分析。对屋子里 k 个人中的每一对 (i,j)，对 $1\leqslant i<j\leqslant k$，定义指示器随机变量 X_{ij} 如下：

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

根据等式(5.6)，两个人生日相同的概率是 $1/n$，因此据引理 5.1，我们有

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

设 X 表示计数生日相同两人对数目的随机变量，我们有 132

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

两边取期望，并应用期望的线性性质，我们得到

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

因此，当 $k(k-1) \geqslant 2n$ 时，生日相同的两人对的期望数至少是 1。因此，若屋子里至少有 $\sqrt{2n}+1$ 个人，我们可以期望至少有两人生日相同。对于 $n=365$，若 $k=28$，生日相同人对数目的期望值为 $(28 \cdot 27)/(2 \cdot 365) \approx 1.035\,6$。因此，如果至少有 28 人，我们可以期望找到至少一对人生日相同。在火星上，一年有 669 个火星日，我们至少需要 38 个火星人。

第一种分析仅用了概率，确定了为使存在至少一对人生日相同概率大于 1/2 所需的人数；第二种分析使用了指示器随机变量，给出了相同生日期望数为 1 时的人数。虽然两种情形下人的准确数目不同，但它们在渐近阶数上是相等的，都为 $\Theta(\sqrt{n})$。

5.4.2　球与箱子

现在我们来考虑这样一个过程，即把相同的球随机投到 b 个箱子里，箱子编号为 1，2，…，b。每次投球都是独立的，每一次投球，球等可能落在每一个箱子中。球落在任一个箱子中的概率为 $1/b$。因此，投球的过程是一组伯努利试验(参见附录 C.4)，每次成功的概率是 $1/b$，其中成功是指球落入指定的箱子中。这个模型对分析散列(参见第 11 章)特别有用，而且我们可以回答关于该投球过程的各种有趣问题。(思考题 C-1 提出了另外一些关于球和箱子的问题。) 133

有多少球落在给定的箱子里？落在给定箱子里的球数服从二项分布 $b(k; n, 1/b)$。如果投 n 个球，公式(C.37)告诉我们，落在给定箱子里的球数期望值是 n/b。

*在平均意义下，我们必须要投多少个球，才能在给定的箱子里投中一个球？*直至给定箱子收到一个球的投球次数服从几何分布，概率为 $1/b$，根据等式(C.32)，成功的投球次数期望是 $1/(1/b)=b$。

我们需要投多少次球，才能使每个箱子里至少有一个球？一次投球落在空箱子里称为一次“命中”。我们想知道为了获得 b 次命中，所需的投球次数期望 n。

采用命中次数，可以把 n 次投球分为几个阶段。第 i 个阶段包括从第 $i-1$ 次命中到第 i 次命中之间的投球。第 1 阶段包含第 1 次投球，因为我们可以保证一次命中，此时所有的箱子都是空的。对第 i 阶段的每一次投球，有 $i-1$ 个箱子有球，$b-i+1$ 个箱子是空的。因而，对第 i 阶段的每次投球，得到一次命中的概率是 $(b-i+1)/b$。

设 n_i 表示第 i 阶段的投球次数。因而，为得到 b 次命中所需的投球次数为 $n = \sum_{i=1}^{b} n_i$。每个随机变量 n_i 服从几何分布，成功的概率是 $(b-i+1)/b$，根据公式(C.32)，于是有

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

根据期望的线性性质，我们有

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

$$= b\sum_{i=1}^{b} \frac{1}{i} = b(\ln b + O(1)) \quad \text{(根据等式(A.7))}$$

所以，在我们期望每个箱子里都有一个球之前，大约要投 $b \ln b$ 次。这个问题也称为**礼券收集者问题**，意思是一个人如果想要收集齐 b 种不同礼券中的每一种，大约需要 $b \ln b$ 张随机得到的礼

134 券才能成功。

5.4.3 特征序列

假设抛投一枚标准的硬币 n 次，最长连续正面的序列的期望长度有多长？答案是 $\Theta(\lg n)$，如以下分析所示。

首先证明最长的连续正面的特征序列的长度期望是 $O(\lg n)$。每次抛硬币时是一次正面的概率为 $1/2$。设 A_{ik} 为这样的事件：长度至少为 k 的正面特征序列开始于第 i 次抛掷，或更准确地说，k 次连续硬币抛掷 i，$i+1$，…，$i+k-1$ 得到的都是正面，其中 $1\leqslant k\leqslant n$，$1\leqslant i\leqslant n-k+1$。因为每次抛硬币是互相独立的，对任何给定事件 A_{ik}，所有 k 次抛掷都是正面的概率是

$$\Pr\{A_{ik}\}=1/2^k \tag{5.8}$$

对于 $k=2\lceil\lg n\rceil$，

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

因而，长度至少为 $2\lceil\lg n\rceil$、起始于位置 i 的一个正面特征序列的概率是很小的。这种序列起始位置至多有 $n-2\lceil\lg n\rceil+1$ 个。所以长度至少为 $2\lceil\lg n\rceil$的正面特征序列开始于任一位置的概率是

$$\Pr\left\{\bigcup_{i=1}^{n-2\lceil\lg n\rceil+1}A_{i,2\lceil\lg n\rceil}\right\}\leqslant\sum_{i=1}^{n-2\lceil\lg n\rceil+1}1/n^2<\sum_{i=1}^{n}1/n^2=1/n \tag{5.9}$$

因为根据布尔不等式(C.19)，一组事件并集的概率至多是各个事件的概率之和。(注意，即使这些事件不独立，布尔不等式依然成立。)

我们现在利用不等式(5.9)来给出最长特征序列的长度界。对于 $j=0$，1，2，…，n，令 L_j 表示最长连续正面的特征序列长度正好是 j 的事件，并设最长特征序列的长度是 L。由期望值的定义，我们有

135
$$\mathrm{E}[L]=\sum_{j=0}^{n}j\Pr\{L_j\} \tag{5.10}$$

我们可以尝试用每个 $\Pr\{L_j\}$的上界来估计这个和，就像不等式(5.9)所计算的那样。遗憾的是，这种方法将导致弱的界。不过，我们可以用从上面分析得到的一些直观知识来得到一个好的界。然而非正式地说，我们观察到在等式(5.10)的总和中，没有任何一项同时让 j 和 $\Pr\{L_j\}$因子都是大的。为什么呢？当 $j\geqslant 2\lceil\lg n\rceil$时，$\Pr\{L_j\}$很小；当 $j<2\lceil\lg n\rceil$时，j 相当小。更正式地说，我们注意到对于 $j=0$，1，…，n，事件 L_j 是不相交的，因此长度至少为 $2\lceil\lg n\rceil$的连续正面特征序列起始于任一位置的概率为 $\sum_{j=2\lceil\lg n\rceil}^{n}\Pr\{L_j\}$。根据不等式(5.9)，我们有 $\sum_{j=2\lceil\lg n\rceil}^{n}\Pr\{L_j\}<1/n$。另外，注意到 $\sum_{j=0}^{n}\Pr\{L_j\}=1$，我们有 $\sum_{j=0}^{2\lceil\lg n\rceil-1}\Pr\{L_j\}\leqslant 1$。因此，我们得到

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

正面特征序列长度超过 $r\lceil\lg n\rceil$次抛掷的概率随着 r 变小而很快减少。对 $r\geqslant 1$，正面特征序列长度至少为 $r\lceil\lg n\rceil$，起始于位置 i 的概率是

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

因此，最长特征序列长度至少为 $r\lceil\lg n\rceil$的概率至多是 $n/n^r=1/n^{r-1}$，或等价地，最长特征序列长

度小于 $r\lceil \lg n\rceil$的概率至少是 $1-1/n^{r-1}$。

看一个例子，抛掷 $n=1\,000$ 次硬币，最少出现 $2\lceil \lg n\rceil=20$ 次连续正面的几率至多是$1/n=1/1\,000$。长度超过 $3\lceil \lg n\rceil=30$ 次连续正面特征序列的几率至多是 $1/n^2=1/1\,000\,000$。

现在我们证明一个补充的下界：在 n 次硬币抛掷中，最长的正面特征序列的长度期望为 $\Omega(\lg n)$。为证明这个界，我们通过把 n 次抛掷划分成大约 n/s 个组，每组 s 次抛掷，来看长度为 136
s 的特征序列。如果选择 $s=\lfloor(\lg n)/2\rfloor$，可以说明这些组中至少有一组可能全是正面，因而可能最长特征序列的长度至少是 $s=\Omega(\lg n)$。然后将表明最长特征序列的长度期望是 $\Omega(\lg n)$。

我们把 n 次硬币抛掷划分成至少$\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor$个组，每组$\lfloor(\lg n)/2\rfloor$次连续抛掷，然后对没有组全是正面的概率求界。根据等式(5.8)，从位置 i 开始都是正面的组的概率是

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

所以长度至少为$\lfloor(\lg n)/2\rfloor$的正面特征序列不从位置 i 开始的概率至多是 $1-1/\sqrt{n}$。既然$\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor$个组是由彼此互斥、独立的抛掷硬币构成，其中每个组都不是长度为$\lfloor(\lg n)/2\rfloor$的特征序列的概率至多是

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

关于此论证，我们用到了不等式(3.12)，即 $1+x\leqslant \mathrm{e}^x$，还用到了你可能想验证的一个事实：对足够大的 n，有$(2n/\lg n-1)/\sqrt{n}\geqslant\lg n$。

因此，最长特征序列超过$\lfloor(\lg n)/2\rfloor$的概率为

$$\sum_{j=\lfloor(\lg n)/2\rfloor}^{n}\Pr\{L_j\}\geqslant 1-O(1/n)\tag{5.11}$$

现在我们可以计算最长特征序列的长度期望的一个下界，从等式(5.10)开始，采用类似于我们上界分析的方式： 137

$$\begin{aligned}\mathrm{E}[L]&=\sum_{j=0}^{n}j\Pr\{L_j\}=\sum_{j=0}^{\lfloor(\lg n)/2\rfloor-1}j\Pr\{L_j\}+\sum_{j=\lfloor(\lg n)/2\rfloor}^{n}j\Pr\{L_j\}\\&\geqslant\sum_{j=0}^{\lfloor(\lg n)/2\rfloor-1}0\cdot\Pr\{L_j\}+\sum_{j=\lfloor(\lg n)/2\rfloor}^{n}\lfloor(\lg n)/2\rfloor\Pr\{L_j\}\\&=0\cdot\sum_{j=0}^{\lfloor(\lg n)/2\rfloor-1}\Pr\{L_j\}+\lfloor(\lg n)/2\rfloor\sum_{j=\lfloor(\lg n)/2\rfloor}^{n}\Pr\{L_j\}\\&\geqslant 0+\lfloor(\lg n)/2\rfloor(1-O(1/n))\qquad(\text{根据不等式(5.11)})\\&=\Omega(\lg n)\end{aligned}$$

和生日悖论一样，可以采用指示器随机变量来得到一个简单而近似的分析。设 $X_{ik}=\mathrm{I}\{A_{ik}\}$ 表示对应于特征序列长度至少为 k、开始于第 i 次抛掷硬币的指示器随机变量。为了计数这些特征序列的总数，定义

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

两边取期望并利用期望的线性性质，我们有

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

通过代入不同的 k 值，可以计算出长度为 k 的特征序列的数目期望。如果这个数大(远大于1)，那么我们期望很多长度为 k 的特征序列会出现，而且出现一个的概率很高。如果这个数小 138
(远小于 1)，那么我们期望很少的长度为 k 的特征序列会出现，而且出现一个的概率很低。如果对某个正常数 c，有 $k=c\lg n$，那么可以得到

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

如果 c 较大，长度为 $c\lg n$ 的特征序列的数目期望将很小，并且我们的结论是它们不大可能发生。另外，如果 $c=1/2$，那么 $\mathrm{E}[X]=\Theta(1/n^{1/2-1})=\Theta(n^{1/2})$，并且我们期望会有大量长度为 $(1/2)\lg n$ 的特征序列。所以，这种长度的特征序列很可能发生。仅通过这些粗略估计，我们可得出结论：最长特征序列的长度期望是 $\Theta(\lg n)$。

5.4.4 在线雇用问题

作为最后一个例子，我们考虑雇用问题的一个变形。假设现在我们不希望面试所有的应聘者以找到最好的一个。我们也不希望因为有更好的申请者出现，不停地雇用新人解雇旧人。取而代之，我们愿意雇用接近最好的应聘者，只雇用一次。我们必须遵守公司的一个要求：每次面试后，或者我们必须马上提供职位给应聘者，或者马上拒绝该应聘者。如何在最小化面试次数和最大化所雇用应聘者的质量两方面取得平衡？

我们可以通过如下方式对该问题建模。在面试一个应聘者之后，我们能够给每人一个分数；令 $score(i)$ 表示给第 i 个应聘者的分数，并且假设没有两个应聘者得到同样分数。在已看过 j 个应聘者后，我们知道这 j 人中哪一个分数最高，但是不知道在剩余的 $n-j$ 个应聘者中会不会有更高分数的应聘者。我们决定采用这样一个策略：选择一个正整数 $k<n$，面试然后拒绝前 k 个应聘者，再雇用其后比前面的应聘者有更高分数的第一个应聘者。如果最好的应聘者在前 k 个面试之中，那么将雇用第 n 个应聘者。我们形式化地表达该策略在过程 ON-LINE-MAXIMUM(k，n) 中，它返回的是我们希望雇用的应聘者下标。

139

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

对每个可能的 k，我们希望确定能雇用最好应聘者的概率。然后选择最佳的 k 值，并用该值来实现这个策略。暂时先假设 k 是固定的。设 $M(j)=\max_{1\leqslant i\leqslant j}\{score(i)\}$ 表示应聘者 $1\sim j$ 中的最高分数。设 S 表示成功选择最好应聘者的事件，S_i 表示最好的应聘者是第 i 个面试者时成功的事件。既然不同的 S_i 不相交，我们有 $\Pr\{S\}=\sum_{i=1}^{n}\Pr\{S_i\}$。注意到，当最好应聘者是前 k 个应聘者中的一个时，我们不会成功，于是对 $i=1$，2，…，k，有 $\Pr\{S_i\}=0$。因而得到

$$\Pr\{S\}=\sum_{i=k+1}^{n}\Pr\{S_i\} \tag{5.12}$$

现在来计算 $\Pr\{S_i\}$。为了当第 i 个应聘者是最好时成功，两件事情必须发生。第一，最好的应聘者必须在位置 i 上，用事件 B_i 表示。第二，算法不能选择从位置 $k+1\sim i-1$ 中任何一个应聘者，而这个选择当且仅当满足 $k+1\leqslant j\leqslant i-1$ 时发生，在程序第 6 行有 $score(j)<bestcore$。（因为分数是唯一的，所以可以忽略 $score(j)=bestcore$ 的可能性。）换句话说，所有 $score(k+1)$ 到 $score(i-1)$ 的值都必须小于 $M(k)$；如果其中有大于 $M(k)$ 的数，则将返回第一个大于 $M(k)$ 的数的下标。我们用 O_i 表示从位置 $k+1$ 到 $i-1$ 中没有任何应聘者入选的事件。幸运的是，两个事件 B_i 和 O_i 是独立的。事件 O_i 仅依赖于位置 1 到 $i-1$ 中值的相对次序，而 B_i 仅依赖于位置 i 的值是否大于所有其他位置的值。从位置 1 到 $i-1$ 的排序并不应影响位置 i 的值是否大于上述所有

值，并且位置 i 的值也不会影响从位置 1 到 $i-1$ 值的次序。因而应用等式(C.15)得到 140

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\}\Pr\{O_i\}$$

$\Pr\{B_i\}$的概率显然是 $1/n$，因为最大值等可能地是 n 个位置中的任一个。若事件 O_i 要发生，从位置 1 到 $i-1$ 的最大值必须在前 k 个位置的一个，并且最大值等可能地在这 $i-1$ 个位置中的任一个。于是，$\Pr\{O_i\}=k/(i-1)$，$\Pr\{S_i\}=k/(n(i-1))$。利用公式(5.12)，我们有

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

我们利用积分来近似约束这个和数的上界和下界。根据不等式(A.12)，我们有

$$\int_k^n \frac{1}{x}\mathrm{d}x \leqslant \sum_{i=k}^{n-1} \frac{1}{i} \leqslant \int_{k-1}^{n-1} \frac{1}{x}\mathrm{d}x$$

求解这些定积分可以得到下面的界：

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

这提供了 $\Pr\{S\}$的一个相当紧确的界。因为我们希望最大化成功的概率，所以关注如何选取 k 值使 $\Pr\{S\}$的下界最大化。(此外，下界表达式比上界表达式更容易最大化。)以 k 为变量对表达式 $(k/n)(\ln n-\ln k)$求导，得到

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

令此导数为 0，我们看到当 $\ln k=\ln n-1=\ln(n/\mathrm{e})$或等价地，$k=n/\mathrm{e}$ 时，概率下界最大化。因而，如果用 $k=n/\mathrm{e}$ 来实现我们的策略，那么将以至少 $1/\mathrm{e}$ 的概率成功雇用到最好的应聘者。 141

练习

5.4-1　一个屋子里必须要有多少人，才能让某人和你生日相同的概率至少为 1/2？必须要有多少人，才能让至少两个人生日为 7 月 4 日的概率大于 1/2？

5.4-2　假设我们将球投入到 b 个箱子里，直到某个箱子中有两个球。每一次投掷都是独立的，并且每个球落入任何箱子的机会均等。请问投球次数期望是多少？

***5.4-3**　在生日悖论的分析中，要求各人生日彼此独立是否很重要？或者，是否只要两两成对独立就足够了？证明你的答案。

***5.4-4**　一次聚会需要邀请多少人，才能让其中 3 人的生日很可能相同？

***5.4-5**　在大小为 n 的集合中，一个 k 字符串构成一个 k 排列的概率是多少？这个问题和生日悖论有什么关系？

***5.4-6**　假设将 n 个球投入 n 个箱子里，其中每次投球独立，并且每个球等可能落入任何箱子。空箱子的数目期望是多少？正好有一个球的箱子的数目期望是多少？

***5.4-7**　为使特征序列长度的下界变得更精确，请说明在 n 次硬币的公平抛掷中，不出现比 $\lg n-2\lg\lg n$ 更长的连续正面特征序列的概率小于 $1/n$。 142

思考题

5-1　(概率计数)　利用一个 b 位的计数器，我们一般只能计数到 2^b-1。而用 R. Morris 的**概率计数法**，我们可以计数到一个大得多的值，代价是精度有所损失。

对 $i=0$，1，…，2^b-1，令计数器值 i 表示 n_i 的计数，其中 n_i 构成了一个非负的递增序列。假设计数器初值为 0，表示计数 $n_0=0$。INCREMENT 运算单元工作在一个计数器上，它以概率的方式包含值 i。如果 $i=2^b-1$，则该运算单元报告溢出错误；否则，INCREMENT 运算单元以概率 $1/(n_{i+1}-n_i)$把计数器增加 1，以概率 $1-1/(n_{i+1}-n_i)$保持计

数器不变。

对所有的 $i \geqslant 0$，若选择 $n_i = i$，此计数器就是一个普通的计数器。若选择 $n_i = 2^{i-1}$（$i > 0$），或者 $n_i = F_i$（第 i 个斐波那契数，参见 3.2 节），则会出现更多有趣的情形。

对于这个问题，假设 n_{2^b-1} 已足够大，发生一个溢出错误的概率可以忽略。

a. 请说明在执行 n 次 INCREMENT 操作后，计数器所表示的数期望值正好是 n。

b. 分析计数器表示的计数的方差依赖于 n_i 序列。我们来看一个简单情形：对所有 $i \geqslant 0$，$n_i = 100i$。在执行了 n 次 INCREMENT 操作后，请估计计数器所表示数的方差。

5-2 （查找一个无序数组） 本题将分析三个算法，它们在一个包含 n 个元素的无序数组 A 中查找一个值 x。

考虑如下的随机策略：随机挑选 A 中的一个下标 i。如果 $A[i] = x$，则终止；否则，继续挑选 A 中一个新的随机下标。重复随机挑选下标，直到找到一个下标 j，使 $A[j] = x$，或者直到我们已检查过 A 中的每一个元素。注意，我们每次都是从全部下标的集合中挑选，于是可能会不止一次地检查某个元素。

a. 请写出过程 RANDOM-SEARCH 的伪代码来实现上述策略。确保当 A 中所有下标都被
143 挑选过时，你的算法应停止。

b. 假定恰好有一个下标 i 使得 $A[i] = x$。在我们找到 x 和 RANDOM-SEARCH 结束之前，必须挑选 A 下标的数目期望是多少？

c. 假设有 $k \geqslant 1$ 个下标 i 使得 $A[i] = x$，推广你对(b)部分的解答。在找到 x 或 RANDOM-SEARCH 结束之前，必须挑选 A 的下标的数目期望是多少？你的答案应该是 n 和 k 的函数。

d. 假设没有下标 i 使得 $A[i] = x$。在检查完 A 的所有元素或 RANDOM-SEARCH 结束之前，我们必须挑选 A 的下标的数目期望是多少？

现在考虑一个确定性的线性查找算法，我们称之为 DETERMINISTIC-SEARCH。具体地说，这个算法在 A 中顺序查找 x，考虑 $A[1]$，$A[2]$，$A[3]$，…，$A[n]$，直到找到 $A[i] = x$，或者到达数组的末尾。假设输入数组的所有排列都是等可能的。

e. 假设恰好有一个下标 i 使得 $A[i] = x$。DETERMINISTIC-SEARCH 平均情形的运行时间是多少？DETERMINISTIC-SEARCH 最坏情形的运行时间又是多少？

f. 假设有 $k \geqslant 1$ 个下标 i 使得 $A[i] = x$，推广你对(e)部分的解答。DETERMINISTIC-SEARCH 平均情形的运行时间是多少？DETERMINISTIC-SEARCH 最坏情形的运行时间又是多少？你的答案应是 n 与 k 的函数。

g. 假设没有下标 i 使得 $A[i] = x$。DETERMINISTIC-SEARCH 平均情形的运行时间是多少？DETERMINISTIC-SEARCH 最坏情形的运行时间又是多少？

最后，考虑一个随机算法 SCRAMBLE-SEARCH，它先将输入数组随机变换排列，然后在排列变换后的数组上，运行上面的确定性线性查找算法。

h. 设 k 是满足 $A[i] = x$ 的下标的数目，请给出在 $k = 0$ 和 $k = 1$ 情况下，算法 SCRAMBLE-SEARCH 最坏情形的运行时间和运行时间期望。推广你的解答以处理 $k \geqslant 1$ 的情况。

144 **i.** 你将会使用 3 种查找算法中的哪一个？解释你的答案。

本章注记

Bollobás[53]、Hofri[174]和 Spencer[321]介绍了大量高等概率技术。随机算法的优点在 Karp[200]和 Rabin[288]中有讨论和综述。Motwani 和 Raghavan[262]的教材中大量论述了随机算法。

雇用问题的很多变形已经得到广泛研究。这些问题通常被称为“秘书问题”。Ajtai、Meggido
145 和 Waarts[11]中给出了该领域中的一个例子。

算法导论（原书第3版）

Introduction to Algorithms Third Edition

“本书是算法领域的一部经典著作，书中系统、全面地介绍了现代算法：从最快算法和数据结构到用于看似难以解决问题的多项式时间算法；从图论中的经典算法到用于字符串匹配、计算几何学和数论的特殊算法。本书第3版尤其增加了两章专门讨论van Emde Boas树（最有用的数据结构之一）和多线程算法（日益重要的一个主题）。”

—— Daniel Spielman，耶鲁大学计算机科学系教授

“作为一个在算法领域有着近30年教育和研究经验的教育者和研究人员，我可以清楚明白地说这本书是我所见到的该领域最好的教材。它对算法给出了清晰透彻、百科全书式的阐述。我们将继续使用这本书的新版作为研究生和本科生的教材及参考书。”

—— Gabriel Robins，弗吉尼亚大学计算机科学系教授

在有关算法的书中，有一些叙述非常严谨，但不够全面；另一些涉及了大量的题材，但又缺乏严谨性。本书将严谨性和全面性融为一体，深入讨论各类算法，并着力使这些算法的设计和分析能为各个层次的读者接受。全书各章自成体系，可以作为独立的学习单元；算法以英语和伪代码的形式描述，具备初步程序设计经验的人就能看懂；说明和解释力求浅显易懂，不失深度和数学严谨性。

第3版的主要变化：

- 新增了van Emde Boas树和多线程算法，并且将矩阵基础移至附录。
- 修订了递归式（现在称为“分治策略”）那一章的内容，更广泛地覆盖分治法。
- 移除两章很少讲授的内容：二项堆和排序网络。
- 修订了动态规划和贪心算法相关内容。
- 由于关于矩阵基础和Strassen算法的材料移到了其他章，矩阵运算这一章的内容所占篇幅更小。
- 修改了对Knuth–Morris–Pratt字符串匹配算法的讨论。
- 新增100道练习和28道思考题，还更新并补充了参考文献。

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

封面设计 锡 杉

上架指导：计算机科学/算法
ISBN 978-7-111-40701-0

9 787111 407010

定价：128.00元