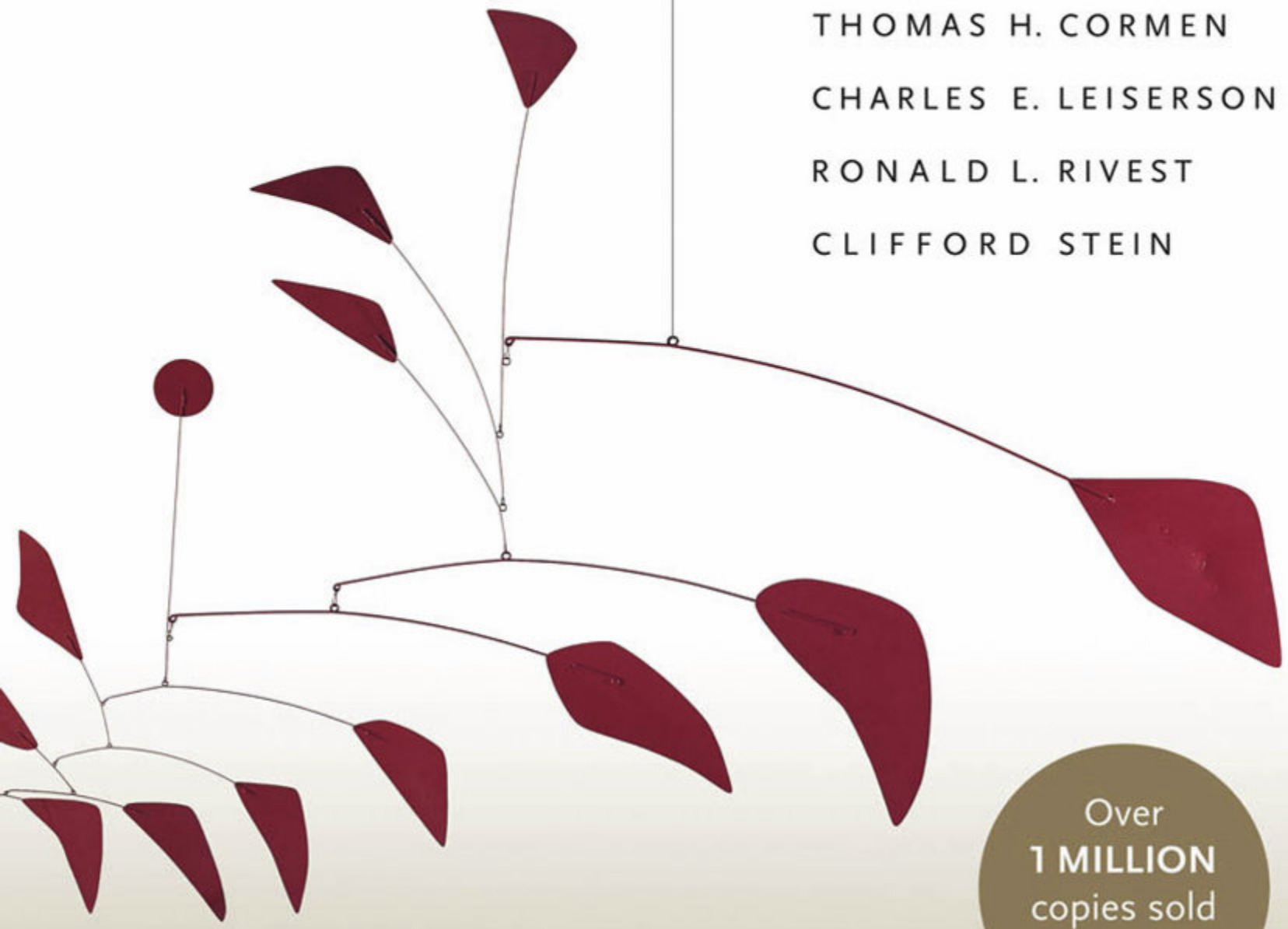
THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN



Over
1 MILLION
copies sold
worldwide

INTRODUCTION TO ALGORITHMS

FOURTH EDITION

Introduction to Algorithms

Fourth Edition

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

Introduction to Algorithms

Fourth Edition

The MIT Press
Cambridge, Massachusetts London, England

c 2022 麻省理工学院

保留所有权利。未经出版商书面许可，不得以任何形式或任何电子或机械手段（包括影印、录制或信息存储和检索）复制本书的任何部分。

麻省理工学院出版社感谢对本书草稿提供评论的匿名同行评审员。学术专家的慷慨工作对于建立我们出版物的权威性和质量至关重要。我们衷心感谢这些未署名读者的贡献。

本书的作者采用了 Times Roman 和 MathTime Professional II 字体。

名称：Cormen，Thomas H.，作者。j Leiserson，Charles Eric，作者。j Rivest，Ronald L.，作者。j Stein，Clifford，作者。标题：算法简介 / Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest、Clifford Stein。说明：第四版。j 马萨诸塞州剑桥：麻省理工学院出版社，[2022] j 包括参考书目和索引。标识符：LCCN 2021037260 j ISBN 9780262046305 主题：LCSH：计算机编程。j 计算机算法。分类：LCC QA76.6 .C662 2022 j DDC 005.13--dc23 LC 记录可在 http://lccn.loc.gov/2021037260 上找到

10 9 8 7 6 5 4 3 2 1

Contents

Preface

不久前，任何听到过“ 算法 ”这个词的人几乎肯定都是计算机科学家或数学家。然而，随着计算机在我们现代生活中的普及，这个术语已不再深奥。如果你环顾你的家，你会发现算法在最普通的地方运行：你的微波炉、洗衣机，当然还有你的电脑。你要求算法为你提供建议：你可能会喜欢什么音乐，或者开车时要走哪条路线。我们的社会，无论好坏，都要求算法为被定罪的罪犯提供量刑建议。你甚至依靠算法来让你活下去，或者至少不让你死去：你汽车或医疗设备中的控制系统。[1]“ 算法 ”这个词似乎每天都会出现在新闻的某个地方。

因此，您不仅应该以计算机科学学生或从业者的身份了解算法，还应该以世界公民的身份了解算法。一旦您了解了算法，您就可以教育其他人算法是什么、算法如何运作以及算法的局限性。

本书全面介绍了现代计算机算法研究。它介绍了许多算法，并对其进行了相当深入的介绍，但让各个层次的读者都能理解它们的设计。所有的分析都进行了布局，有些简单，有些比较复杂。我们试图在不牺牲覆盖深度或数学严谨性的情况下保持解释的清晰。

每章介绍一种算法、一种设计技术、一个应用领域或一个相关主题。算法以英语和伪代码描述，旨在让任何做过一点编程的人都能读懂。本书包含 231 个图（其中许多图由多个部分组成）说明算法的工作原理。由于我们强调 *efficiency* 作为设计标准，因此我们对算法的运行时间进行了仔细的分析。

---

[1] To understand many of the ways in which algorithms influence our daily lives, see the book by Fry [162].

本书主要面向本科生或研究生的算法或数据结构课程。由于本书不仅讨论了数学方面的问题，还讨论了算法设计中的工程问题，因此也非常适合技术专业人员自学。

在这第四版中，我们再次更新了整本书。这些变化涵盖了广泛的领域，包括新的章节和部分、彩色插图，以及我们希望您会发现的更具吸引力的写作风格。

致老师

我们设计这本书的目的是使其既通用又完整。您会发现它对各种课程都很有用，从本科的数据结构课程到研究生的算法课程。由于我们提供的材料比典型的一学期课程要多得多，因此您可以选择最适合您想要教授的课程的材料。

您应该会发现，围绕您需要的章节来组织课程非常容易。我们已将章节设计得相对独立，这样您就不必担心章节之间出现意外且不必要的依赖。在本科课程中，您可能只使用章节中的部分内容，而在研究生课程中，您可能涵盖整个章节。

我们提供了 931 个练习和 162 个问题。每个部分都以练习结束，每个章节都以问题结束。练习通常是测试对材料基本掌握程度的简短问题。有些是简单的自我检查思维练习，但许多练习内容丰富，适合作为指定的家庭作业。问题包括更详细的案例研究，通常会介绍新材料。它们通常由几个部分组成，引导学生完成解决问题所需的步骤。

与本书第三版一样，我们公开了部分（但绝不是全部）问题和练习的答案。您可以在我们的网站 http://mitpress.mit.edu/algorithms/ 上找到这些答案。您需要查看此网站，看看它是否包含您计划布置的练习或问题的解决方案。由于我们发布的答案集可能会随着时间的推移而增加，因此我们建议您每次教授课程时都查看该网站。

我们对更适合研究生而非本科生的章节和练习标上了星号。标有星号的章节并不一定比没有星号的章节更难，但它可能需要对更高级的数学有所了解。同样，标有星号的练习可能需要高级背景或超过平均水平的创造力。

对学生来说

我们希望这本教科书能为您提供算法领域的有趣介绍。我们试图让每个算法都易于理解和有趣。为了帮助您在遇到不熟悉或困难的算法时，我们会逐步描述每个算法。我们还提供了理解算法分析所需的数学知识的详细解释，并提供了支持性图表，以帮助您直观地了解正在发生的事情。

由于本书篇幅较大，您的课堂可能只会涉及其中的一部分内容。尽管我们希望您发现本书作为课程教科书对您有所帮助，但我们也试图使其足够全面，以保证您未来专业书架上有足够的空间。

阅读这本书的先决条件是什么？

您需要一些编程经验。特别是，您应该了解递归过程和简单的数据结构，例如数组和链接列表（尽管第 10.2 节介绍了链接列表和您可能发现的新变体）。• 您应该具备一些数学证明能力，尤其是数学归纳法证明。本书的一些部分依赖于一些初等微积分知识。虽然本书通篇都使用数学，但第 I 部分和附录 A-D 会教您所需的所有数学技巧。

我们的网站 http://mitpress.mit.edu/algorithms/ 提供了一些问题和练习的答案链接。您可以随意对照我们的答案检查您的答案。不过，我们要求您不要将答案发送给我们。

对于专业人士

本书涵盖的主题非常广泛，是一本出色的算法手册。由于每章都相对独立，因此您可以专注于与您最相关的主题。

由于我们讨论的大多数算法都具有很强的实用性，因此我们讨论了实现问题和其他工程问题。我们经常为少数主要具有理论意义的算法提供实用的替代方案。

如果您希望实现任何算法，您应该会发现将我们的伪代码翻译成您喜欢的编程语言是一项相当简单的任务。我们设计了伪代码来清晰简洁地呈现每个算法。因此，我们不处理错误处理和其他需要对您的编程环境做出特定假设的软件工程问题。我们试图简单直接地呈现每个算法，而不会让特定编程语言的特性掩盖其本质。如果您习惯于 0 源数组，您可能会发现我们经常使用

从 1 开始索引数组是一个小障碍。您始终可以从索引中减去 1，或者只是过度分配数组并将位置 0 保留未使用。

我们理解，如果您在课程之外使用本书，则可能无法根据讲师提供的解决方案检查问题和练习的解决方案。我们的网站 http://mitpress.mit.edu/algorithms/ 链接到部分问题和练习的解决方案，以便您可以检查您的工作。请不要将您的解决方案发送给我们。

致我们的同事

我们提供了详尽的参考书目和当前文献的链接。每章结尾都有一组章节注释，提供历史细节和参考文献。然而，章节注释并没有提供整个算法领域的完整参考。虽然对于这样一本厚的书来说可能很难相信，但篇幅限制使我们无法包含许多有趣的算法。

尽管学生们多次要求提供问题和练习的解决方案，但我们采取了不为他们引用参考文献的政策，以消除学生查找解决方案而不是自己发现解决方案的诱惑。

第四版的变化

正如我们所说的第二版和第三版的变化，这本书的变化要么不大，要么很大，这取决于你怎么看。快速浏览一下目录就会发现，第三版的大多数章节和部分都出现在了第四版中。我们删除了三章和几个部分，但除了这些新章节之外，我们还添加了三章和几个新部分。

我们保留了前三版的混合结构。本书不是仅按问题领域或仅按技术来组织章节，而是融合了两者的元素。它包含基于技术的章节，包括分而治之、动态规划、贪婪算法、摊销分析、增强数据结构、NP 完全性和近似算法。但它也有关于排序、动态集的数据结构和图问题算法的完整部分。我们发现，尽管您需要知道如何应用技术来设计和分析算法，但问题很少会告诉您哪种技术最适合解决它们。

第四版中的一些变化普遍适用于整本书，而一些变化则只针对特定的章节或部分。以下是最重要的一般变化的摘要：

我们添加了 140 个新练习和 22 个新问题。我们还改进了许多旧练习和问题，这通常是读者反馈的结果。（感谢所有提出建议的读者。）

我们有颜色！在麻省理工学院出版社的设计师的帮助下，我们选择了有限的调色板，旨在传达信息并令人赏心悦目。（我们很高兴以红色和黑色显示红黑树！）为了增强可读性，索引中的定义术语、伪代码注释和页码都是彩色的。• 伪代码程序出现在棕褐色背景上，以便于识别，并且它们不一定出现在第一次引用的页面上。如果没有，文本会将您引导到相关页面。同样，对编号方程、定理、引理和推论的非局部引用也包括页码。• 我们删除了很少教授的主题。我们完全删除了关于斐波那契堆、van Emde Boas 树和计算几何的章节。此外，还删除了以下材料：最大子数组问题、实现指针和对象、完美散列、随机构建的二叉搜索树、拟阵、最大流的推送重标记算法、迭代快速傅立叶变换方法、线性规划的单纯形算法细节和整数分解。您可以在我们的网站 http://mitpress.mit.edu/algorithms/ 上找到所有被删除的材料。• 我们审查了整本书，并重写了句子、段落和章节，使写作更清晰、更个性化、性别中立。例如，以前版本中的 “ 旅行商问题” 现在称为 “ 旅行推销员问题 。” 我们认为，对于工程和科学（包括我们自己的计算机科学领域）来说，欢迎所有人至关重要。 （让我们感到困惑的地方是第 13 章，该章需要一个表示父母兄弟姐妹的术语。由于英语中没有这样的性别中立术语，我们遗憾地坚持使用 “uncle”。）• 章节注释、参考书目和索引均已更新，反映了自第三版以来算法领域的急剧增长。

我们修正了错误，将大多数修正发布在了第三版勘误表的网站上。在我们全力准备这一版时报告的错误没有发布，但在这一版中得到了修正。（再次感谢所有帮助我们发现问题的读者。）

第四版的具体变化包括以下内容：

我们重新命名了第 3 章，并在深入研究正式定义之前添加了一个概述渐近符号的部分。• 第 4 章进行了重大修改，以改进其数学基础并使其更加健壮和直观。引入了算法递归的概念，并讨论了在递归中忽略下限和上限的主题 。

rences 得到了更严格的处理。主定理的第二种情况包含多对数因子，现在提供了主定理的 “连续” 版本的严格证明。我们还介绍了强大而通用的 Akra-Bazzi 方法（无需证明）。

第 9 章中的确定性顺序统计算法略有不同，并且对随机化和确定性顺序统计算法的分析都进行了修改。

除了堆栈和队列之外，第 10.1 节还讨论了存储数组和矩阵的方法。

第 11 章关于哈希表的内容包括对哈希函数的现代处理。它还强调，当底层硬件实现缓存以支持本地搜索时，线性探测是一种解决冲突的有效方法。

为了替换第 15 章中关于拟阵的章节，我们将第三版中关于离线缓存的问题转换为完整的章节。• 第 16.4 节现在包含对分析表加倍和减半的潜在函数的更直观的解释。• 关于扩充数据结构的第 17 章从第三部分移至第五部分，反映了我们认为该技术超越了基本材料的观点。• 第 25 章是有关二分图中匹配的新章节。它介绍了寻找最大基数匹配、解决稳定婚姻问题以及寻找最大权重匹配（称为“分配问题”）的算法。• 关于任务并行计算的第 26 章已使用现代术语更新，包括章节名称。• 第 27 章介绍在线算法，是另一个新章节。在在线算法中，输入会随时间到达，而不是在算法开始时全部可用。本章介绍了几个在线算法的示例，包括确定在走楼梯之前等待电梯的时间、通过移至前端启发式方法维护链接列表以及评估缓存的替换策略。

在第 29 章中，我们删除了单纯形算法的详细介绍，因为它涉及大量数学知识，但并没有真正传达出许多算法思想。本章现在重点介绍如何将问题建模为线性规划的关键方面，以及线性规划的基本对偶性质。

第 32.5 节在字符串匹配章节中增加了简单但功能强大的后缀数组结构。

第 33 章是有关机器学习的第三章，介绍了机器学习中使用的几种基本方法：将相似项归为一组的聚类、加权多数算法和用于求函数极小值的梯度下降法。

第 34.5.6 节总结了多项式时间约简的策略，表明问题是 NP 难题。

修改了第 35.3 节中集合覆盖问题的近似算法的证明。

网站

您可以使用我们的网站 http://mitpress.mit.edu/algorithms/ 获取补充信息并与我们交流。该网站链接到已知错误列表、未包含在第四版中的第三版材料、选定练习和问题的答案、本书中许多算法的 Python 实现、解释老套教授笑话的列表（当然），以及我们可能添加的其他内容。该网站还告诉您如何报告错误或提出建议。

我们如何制作这本书

与前三版一样，第四版采用 LaTeX 2" 格式。我们使用 Times 字体，数学排版采用 MathTime Professional II 字体。与所有前几版一样，我们使用 Windex（一种 C 语言专业版）编制索引。我们编写的 gram，并使用 BIBTEX 制作了参考书目。本书的 PDF 文件是在运行 macOS 10.14 的 MacBook Pro 上创建的。

在第三版的前言中，我们恳求苹果更新 MacDraw Pro 以适应 macOS 10，但是没有得到回应，因此我们继续在运行 MacDraw Pro 的英特尔之前的 Mac 上在旧版本 macOS 10 的 Classic 环境下绘制插图。插图中出现的许多数学表达式都是使用 psfrag 包中的 LaTeX 2" 编写的。

第四版致谢

自 1987 年开始编写第一版以来，我们一直与麻省理工学院出版社合作，与多位导演、编辑和制作人员合作。在我们与麻省理工学院出版社合作期间，他们的支持一直非常出色。特别感谢我们的编辑 Marie Lee，她忍受了我们太久，还有 Elizabeth Swayze，她推动我们超越了 ûnish 的界限。还要感谢导演 Amy Brand 和 Alex Hoopes。

与第三版一样，在编写第四版时，我们的工作地点也是分散的，分别在达特茅斯学院计算机科学系、麻省理工学院计算机科学与人工智能实验室和麻省理工学院电气工程与计算机科学系以及哥伦比亚大学工业工程与运筹学系、计算机科学系和数据科学研究所工作。在 COVID-19 疫情期间，我们大部分时间都在家工作。我们感谢各自的大学和同事为我们提供了如此支持和激励的环境。在完成这本书时，我们这些尚未退休的人都渴望回到各自的大学，因为疫情似乎正在减弱。

朱莉·萨斯曼（Julie Sussman）在巨大的时间压力下再次用她的技术性文字编辑拯救了我们。如果没有朱莉，这本书会充满错误（或者说，错误比现在多得多），可读性也会大大降低。朱莉，我们将永远感激你。剩下的错误是作者的责任（可能是在朱莉阅读材料后插入的）。

在编写此版本的过程中，我们更正了之前版本中的数十处错误。我们感谢多年来报告错误并提出改进建议的读者（数量太多，无法一一列出）。

在准备本版中的一些新材料时，我们得到了很多帮助。麻省理工学院的 Neville Campbell（无党派人士）、Bill Kuszmaul 和纽约大学的 Chee Yap 在第 4 章中就复发的处理提供了宝贵的建议。加州大学河滨分校的 Yan Gu 在第 26 章中就并行算法提供了反馈。微软研究院的 Rob Shapire 在第 33 章中提供了详细评论，改变了我们对机器学习材料的处理方式。麻省理工学院的 Qi Qi 帮助分析了蒙提霍尔问题（问题 C-1）。

麻省理工学院出版社的 Molly Seaman 和 Mary Reilly 帮助我们选择了插图的调色板，达特茅斯学院的 Wojciech Jarosz 建议改进我们新上色的插图的设计。Yichen (Annie) Ke 和 Linda Xiao 现已从达特茅斯学院毕业，她们协助为插图上色，Linda 还制作了本书网站上提供的许多 Python 实现。

最后，我们要感谢我们的妻子4温迪·莱瑟森、盖尔·里维斯特、丽贝卡·艾弗里和已故的妮可·科曼4以及我们的家人。爱我们的人的耐心和鼓励使得这个项目成为可能。我们深情地把这本书献给他们。

托马斯·H科尔曼 — *Lebanon, New Hampshire*
查尔斯·E·莱瑟森 — *Cambridge,Massachusetts*
罗纳德·李维斯特 — *Cambridge,Massachusetts*
克利福德·斯坦 — *New York, New York*

June,2021

Part I Foundations

介绍

当你设计和分析算法时，你需要能够描述它们的运作方式和设计方法。你还需要一些数学工具来证明你的算法能正确高效地完成任务。本部分将帮助你入门。本书的后续部分将以此为基础。

第 1 章概述了算法及其在现代计算系统中的地位。本章定义了什么是算法并列举了一些示例。它还提出了将算法视为一种技术的理由，与快速硬件、图形用户界面、面向对象系统和网络等技术并列。

在第 2 章中，我们介绍了第一种算法，它解决了对 n 个数字序列进行排序的问题。这些算法是用伪代码编写的，虽然不能直接转换成任何传统的编程语言，但它足够清楚地传达了算法的结构，您应该能够用您选择的语言来实现它。我们研究的排序算法是使用增量方法的插入排序和使用递归技术的归并排序，称为 “分而治之”。虽然每个算法所需的时间都会随着 n 值的增加而增加，但两种算法增加的速度不同。我们在第 2 章中确定了这些运行时间，并开发了一个有用的 “渐近” 符号来表示它们。

第 3 章精确定义了渐近符号。我们将使用渐近符号来限制函数的增长，这些函数最常用于描述上面和下面的算法的运行时间。本章首先非正式地定义最常用的渐近符号，并给出如何应用它们的示例。然后，它正式定义了我的渐近符号，并介绍了如何将它们组合在一起的惯例。第 3 章的其余部分主要是数学符号的介绍，更多的是为了确保您对符号的使用与本书中的符号相匹配，而不是教您新的数学概念。

第 4 章进一步探讨了第 2 章中介绍的分治法。它提供了另外两个用于乘以方阵的分治算法示例，包括施特拉森的令人惊讶的方法。第 4 章包含解决递归的方法，这些方法对于描述递归算法的运行时间很有用。在替代方法中，你猜测一个答案并证明它是正确的。递归树提供了一种生成猜测的方法。第 4 章还介绍了“主方法”的强大技术，你通常可以使用该方法来解决分治算法中出现的递归。虽然本章提供了主定理所依赖的基础定理的证明，但是你可以随意使用主方法而不必深入研究证明。第 4 章以一些高级主题作为结尾。

第 5 章介绍概率分析和随机算法。通常，在由于存在固有概率分布的情况下，算法的运行时间可能因相同大小的不同输入而不同，这时可以使用概率分析来确定算法的运行时间。在某些情况下，您可能假设输入符合已知的概率分布，这样就可以对所有可能的输入平均运行时间。在其他情况下，概率分布不是来自输入，而是来自算法过程中的随机选择。如果算法的行为不仅由其输入决定，还由随机数生成器生成的值决定，那么该算法就是随机算法。您可以使用随机算法对输入强制实施概率分布4，从而确保没有特定的输入总是导致性能不佳4，甚至可以限制在有限范围内产生错误结果的算法的错误率。

附录 A – D 包含其他数学材料，在阅读本书时会对您有所帮助。您可能在阅读本书之前已经看过附录章节中的大部分材料（尽管我们使用的具体定义和符号约定在某些情况下可能与您过去所见的不同），因此您应该将附录视为参考资料。另一方面，您可能还没有看过第一部分中的大部分材料。第一部分的所有章节和附录都是以教程风格编写的。

1 The Role of Algorithms in Computing

什么是算法？为什么算法的研究是值得的？相对于计算机中使用的其他技术，算法的作用是什么？本章将回答这些问题。

1.1 算法

通俗地说，***algorithm*** 是任何定义明确的计算过程，它接受某个值或一组值作为 ***input***，并在有限的时间内产生某个值或一组值作为 ***output***。因此，算法是将输入转换为输出的一系列计算步骤。

您还可以将算法视为解决明确指定的 ***computa- tional problem*** 的工具。问题陈述一般指定了问题实例所需的输入/输出关系，通常大小任意大。算法描述了实现所有问题实例的输入/输出关系的特定计算过程。

例如，假设您需要将数字序列按单调递增顺序排序。这个问题在实践中经常出现，为引入许多标准设计技术和分析工具提供了沃土。以下是我们对 ***sorting problem*** 的正式定义：

输入：一个由 n 个数字组成的序列 ha $_1$; a $_2$; : : : ; a $_n$ i。

输出：输入序列的排列（重新排序）ha $'_1$; a $'_2$; : : :; a $'_n$ i，使得 a $'_1$ a $'_2$ a $'_n$。

因此，给定输入序列 h31; 41; 59; 26; 41; 58i，正确的排序算法将返回序列 h26; 31; 41; 41; 58; 59i 作为输出。这样的输入序列是

称为排序问题的 ***instance***。一般来说，***instance of a problem*** [1] 由计算问题解决方案所需的输入（满足问题陈述中施加的任何约束）组成。

由于许多程序都将其用作中间步骤，因此排序是计算机科学中的一项基本操作。因此，您可以使用大量优秀的排序算法。对于给定的应用程序，哪种算法最适合取决于4其他因素，包括要排序的项目数量、项目已经排序的程度、项目值的可能限制、计算机的体系结构以及要使用的存储设备类型：主内存、磁盘，甚至4古老的4磁带。

如果对于每个作为输入的问题实例，算法能够在有限时间内完成计算并输出该问题实例的正确解，则该算法为 ***correct***。给定的计算问题对应的正确算法是 ***solves***。不正确的算法可能在某些输入实例上根本不会停止，或者可能会因答案错误而停止。与你可能预料的相反，如果你能控制不正确算法的错误率，那么它有时会很有用。在第 31 章学习寻找大素数的算法时，我们将看到一个错误率可控的算法的例子。然而，通常我们只关注正确的算法。

算法可以用英语、计算机程序甚至硬件设计来描述。唯一的要求是，该规范必须提供要遵循的计算过程的精确描述。

算法可以解决哪些类型的问题？

排序绝不是唯一一个开发了算法的计算问题。（当你看到这本书的篇幅时，你可能会怀疑这一点。）算法的实际应用无处不在，包括以下示例：

人类基因组计划在识别人类 DNA 中大约 30,000 个基因、确定构成人类 DNA 的大约 30 亿个化学碱基对的序列、将这些信息存储在数据库中以及开发数据分析工具方面取得了巨大进展。这些步骤中的每一个都需要复杂的算法。虽然所涉及的各种问题的解决方案超出了本书的范围，但解决这些生物学问题的许多方法都使用了这里介绍的思想，使科学家能够在高效利用资源的同时完成任务。动态规划，正如

[1] Sometimes, when the problem context is known, problem instances are themselves simply called "problems."

在第 14 章中，这是解决这些生物学问题的重要技术，特别是涉及确定 DNA 序列相似性的问题。节省了人力和机器的时间，也节省了金钱，因为可以通过实验室技术提取更多信息。

互联网使世界各地的人们能够快速访问和检索大量信息。借助巧妙的算法，互联网上的网站能够管理和操纵如此大量的数据。算法必不可少的问题示例包括找到数据传输的良好路线（解决此类问题的技术出现在第 22 章中），以及使用搜索引擎快速查找特定信息所在的页面（相关技术在第 11 章和第 32 章中）。

电子商务使商品和服务能够以电子方式进行协商和交换，它依赖于个人信息（如信用卡号、密码和银行对账单）的隐私。电子商务使用的核心技术包括公钥加密和数字签名（第 31 章介绍），它们基于数值算法和数论。

制造业和其他商业企业通常需要以最有利的方式配置稀缺资源。石油公司可能希望知道在哪里布置油井，以使预期利润最大化。政治候选人可能想确定在哪里花钱购买竞选广告，以最大限度地提高赢得选举的机会。航空公司可能希望以尽可能最低的成本为航班分配机组人员，确保覆盖每个航班并符合政府关于机组人员排班的规定。互联网服务提供商可能希望确定在何处放置额外资源，以便更有效地为其客户提供服务。所有这些都是可以通过将它们建模为线性规划来解决的问题的例子，第 29 章将对此进行探讨。

虽然这些示例的一些细节超出了本书的范围，但我们确实给出了适用于这些问题和问题领域的基础技术。我们还展示了如何解决许多具体问题，包括以下内容：

你有一张公路地图，上面标明了每对相邻交叉口之间的距离，你想确定从一个交叉口到另一个交叉口的最短路线。即使你禁止交叉路线，可能的路线数量也可能非常多。你怎样才能从所有可能的路线中选出最短的路线呢？你可以先将公路地图（它本身就是实际道路的模型）建模为一个图形（我们将在第 VI 部分和附录 B 中看到）。在这个图形中，你想找到从一个顶点到另一个顶点的最短路径。第 22 章将介绍如何有效地解决这个问题。

给定一个机械设计，即零件库，其中每个零件可能包含其他零件的实例，按顺序列出零件，以便每个零件出现在使用它的任何零件之前。如果设计包含 n 个零件，则有 nŠ 个可能的顺序，其中 nŠ 表示阶乘函数。由于阶乘函数的增长速度甚至比指数函数还快，因此您无法生成每个可能的顺序，然后验证在该顺序中，每个零件是否出现在使用它的零件之前（除非您只有几个零件）。这个问题是拓扑排序的一个实例，第 20 章将介绍如何有效地解决这个问题。

医生需要确定一幅图像是癌性肿瘤还是良性肿瘤。医生有许多其他肿瘤的图像，其中一些已知是癌性的，一些已知是良性的。癌性肿瘤与其他癌性肿瘤的相似性可能大于与良性肿瘤的相似性，而良性肿瘤与其他良性肿瘤的相似性则更高。通过使用聚类算法，如第 33 章中所述，医生可以确定哪种结果更有可能出现。

您需要压缩包含文本的大型文件，以使其占用更少的空间。目前已知有许多方法可以做到这一点，包括“LZW 压缩，” 查找重复的字符序列。第 15 章研究了一种不同的方法，即“霍夫曼编码，” 它使用不同长度的位序列对字符进行编码，出现频率较高的字符用较短的位序列进行编码。

这些列表远非详尽无遗（正如你可能从本书的篇幅中推测的那样），但它们表现出许多有趣的算法问题所共有的两个特征：

1. 它们有许多候选解决方案，但绝大多数都不能解决当前的问题。要在不明确检查每个可能的解决方案的情况下找到一个能解决当前问题或“最佳”的解决方案，可能会非常困难。 2. 它们有实际应用。在上面列出的问题中，寻找最短路径是最简单的例子。运输公司（例如卡车运输或铁路公司）对寻找公路或铁路网络的最短路径有经济利益，因为走较短的路径可以降低劳动力和燃料成本。或者，互联网上的路由节点可能需要找到通过网络的最短路径才能快速路由消息。又或者，想要从纽约开车到波士顿的人可能想使用导航应用查找行车路线。

并非每个用算法解决的问题都有一组容易识别的候选解决方案。例如，给定一组表示以固定时间间隔采集的信号样本的数值，离散傅里叶变换将

将时间域转换为频率域。也就是说，它将信号近似为正弦波的加权和，产生各种频率的强度，这些频率相加后近似于采样信号。离散傅里叶变换除了是信号处理的核心之外，还可用于数据压缩和大多项式与整数的乘法。第 30 章针对此问题提供了一种有效的算法，即快速傅里叶变换（通常称为 FFT）。本章还概述了硬件 FFT 电路的设计。

数据结构

本书还介绍了几种数据结构。***data structure*** 是一种存储和组织数据的方法，以便于访问和修改。使用适当的数据结构是算法设计的重要组成部分。没有一种数据结构可以适用于所有目的，因此您应该了解其中几种数据结构的优点和局限性。

技术

尽管你可以将本书作为算法的“烹饪书”，但你可能某一天会遇到一个无法轻易找到已发表算法的问题（例如，本书中的许多练习和问题）。本书将教你算法设计和分析的技术，以便你可以自己开发算法，证明它们给出正确答案，并分析它们的效率。不同的章节涉及算法问题解决的不同方面。某些章节解决特定问题，例如第 9 章中的寻找中位数和顺序统计量、第 21 章中的计算最小生成树以及第 24 章中的确定网络中的最大流。其他章节介绍了一些技术，例如第 2 章和第 4 章中的分治法、第 14 章中的动态规划和第 16 章中的摊销分析。

难题

本书的大部分内容都是关于高效算法的。我们通常衡量效率的标准是速度：一个算法需要多长时间才能产生结果？然而，有些问题我们不知道有哪个算法能在合理的时间内运行。第 34 章研究了这些问题中一个有趣的子集，即 NP 完全问题。

为什么 NP 完全问题如此有趣？首先，尽管从未发现过解决 NP 完全问题的有效算法，但从未有人证明不存在解决 NP 完全问题的有效算法。换句话说，没有人知道是否存在解决 NP 完全问题的有效算法。其次，

NP 完全问题具有一个显著的特性：如果其中任何一个问题都存在有效算法，那么所有问题都存在有效算法。NP 完全问题之间的这种关系使得缺乏有效解决方案变得更加诱人。第三，几个 NP 完全问题与我们已知的有效算法问题相似，但不完全相同。计算机科学家对问题陈述的微小变化如何导致最著名算法的效率发生巨大变化感到好奇。

您应该了解 NP 完全问题，因为其中一些问题在实际应用中经常出现。如果您被要求为 NP 完全问题开发一种有效的算法，您很可能会花费大量时间进行无果而终的搜索。相反，如果您可以证明该问题是 NP 完全的，那么您可以花时间开发一种有效的近似算法，即一种可以给出良好但不一定是最好的解决方案的算法。

举一个具体的例子，假设有一家拥有中央仓库的配送公司。每天，配送公司都会在仓库装货，然后派出配送卡车将货物运送到多个地址。一天结束时，每辆卡车都必须回到仓库，以便为第二天装货做好准备。为了降低成本，该公司希望选择一个配送站点顺序，使每辆卡车的总行驶距离最短。这个问题就是众所周知的“旅行商问题”，它是 NP 完全的。[2] 它没有已知的有效算法。然而，在某些假设下，我们知道一些有效的算法，它们可以计算出接近最小可能的总距离。第 35 章讨论了此类“近似算法。”

替代计算模型

多年来，我们一直指望处理器时钟速度以稳定的速度增长。然而，物理限制是不断提高时钟速度的一个根本障碍：由于功率密度随时钟速度超线性增加，一旦时钟速度足够高，芯片就有熔化的风险。因此，为了每秒执行更多的计算，芯片被设计为不仅包含一个，而且包含多个处理“核心。”我们可以将这些多核计算机比作单个芯片上的几台顺序计算机。换句话说，它们是一种“并行计算机。”为了从多核计算机中获得最佳性能，我们需要在设计算法时考虑并行性。第 26 章介绍了一种利用多个处理核心的”任务并行”算法模型。该模型具有理论和实践方面的优势

[2] To be precise, only decision problems—those with a “yes/no” answer—can be NP-complete. The decision version of the traveling salesperson problem asks whether there exists an order of stops whose distance totals at most a given amount.

从实际角度来看，许多现代并行编程平台都采用了类似这种并行模型。

本书中的大多数示例都假设算法开始运行时所有输入数据都可用。算法设计中的许多工作都做出了相同的假设。然而，对于许多重要的现实世界示例，输入实际上是随着时间的推移而到达的，算法必须在不知道未来会到达哪些数据的情况下决定如何进行。在数据中心，作业不断地到达和离开，调度算法必须决定何时何地运行作业，而不知道未来会到达哪些作业。流量必须根据当前状态在互联网上路由，而不知道流量将来会到达何处。医院急诊室对首先治疗哪些患者进行分类决定，而不知道其他患者何时会到达以及他们需要什么治疗。随着时间的推移接收输入而不是在开始时就拥有所有输入的算法是***online algorithms***，第 27 章将对此进行研究。

练习

1.1-1
描述你自己现实世界中需要排序的例子。描述一个需要找到两点之间最短距离的例子。

1.1-2
除了速度之外，在现实环境中您可能还需要考虑哪些其他效率衡量标准？

1.1-3
选择您见过的数据结构，并讨论其优点和局限性。

1.1-4上面给出的最短路径问题和旅行商问题有何相似之处？它们有何不同？

1.1-5
提出一个只有最佳解决方案才能解决的现实问题。然后想出一个“近似”最佳解决方案就足够好的问题。

1.1-6
描述一个现实世界的问题，其中有时在您需要解决问题之前整个输入都是可用的，但有时输入并不完全提前可用并且会随着时间的推移而到达。

1.2 算法作为一种技术

如果计算机的速度无限快，并且计算机内存可用，您还有理由学习算法吗？答案是肯定的，如果您没有其他原因，那么您仍然希望确保您的解决方法能够终止并且得到正确的答案。

如果计算机的速度无限快，那么任何正确的解决问题的方法都可以。您可能希望您的实现符合良好的软件工程实践（例如，您的实现应该经过精心设计和记录），但您通常会使用最容易实现的方法。

当然，计算机可能很快，但并不是无限快。因此，计算时间是一种有限的资源，这使得它非常宝贵。虽然俗话说，“时间就是金钱”，但时间比金钱更宝贵：金钱花掉后可以收回，但时间一旦花掉，就再也收不回来了。内存可能很便宜，但它既不是无限的也不是免费的。你应该选择那些能有效利用时间和空间资源的算法。

效率

为解决同一问题而设计的不同算法在效率上往往存在巨大差异。这些差异可能比硬件和软件之间的差异更为显著。

作为示例，第 2 章介绍了两种排序算法。第一个算法为 ***insertion sort***，对 n 个项目进行排序所需的时间大致等于 $c_1 n^2$，其中 c_1 是一个与 n 无关的常数。也就是说，它所需的时间大致与 n^2 成比例。第二个算法为 ***merge sort***，所需时间大致等于 $c_2 n \lg n$，其中 $\lg n$ 代表 $\log_2 n$，c_2 是另一个与 n 无关的常数。插入排序的常数因子通常比归并排序小，因此 $c_1 < c_2$。我们将看到，常数因子对运行时间的影响远小于对输入大小 n 的依赖性。我们将插入排序的运行时间写为 $c_1 n - n$，将归并排序的运行时间写为 $c_2 n - \lg n$。然后我们看到，插入排序的运行时间为 n 倍，而归并排序的运行时间为 $\lg n$ 倍，两者相差甚远。例如，当 n 为 1000 时，$\lg n$ 约为 10，而当 n 为 1,000,000 时，$\lg n$ 约为 20。虽然对于较小的输入，插入排序通常比归并排序运行得更快，但是一旦输入大小 n 足够大，归并排序相对于 n 的 $\lg n$ 优势就会绰绰有余。无论 c_1 比 c_2 小多少，总会有一个交叉点，超过该点，归并排序就会更快。

举一个具体的例子，让我们对比一下运行插入排序的较快计算机（计算机 A）和运行合并排序的较慢计算机（计算机 B）。它们各自都必须对 1000 万个数字的数组进行排序。（尽管 1000 万个数字似乎很多，但如果这些数字是 8 字节整数，则输入占用大约 80MB，这甚至是廉价笔记本电脑内存的很多倍。）假设计算机 A 每秒执行 100 亿条指令（在撰写本文时比任何单台顺序计算机都快），而计算机 B 每秒仅执行 1000 万条指令（比大多数现代计算机慢得多），因此计算机 A 的原始计算能力比计算机 B 快 1000 倍。为了让差异更加明显，假设世界上最狡猾的程序员用机器语言为计算机 A 编写插入排序代码，生成的代码需要 $2n^2$ 条指令来对 n 个数字进行排序。进一步假设，一个普通程序员使用高级语言和低效编译器实现合并排序，生成的代码需要 50 n lg n 条指令。要对 1000 万个数字进行排序，计算机 A 需要

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ seconds (more than 5.5 hours)},$$

而计算机 B 则

$5010^7 \lg 10^7$ 条指令10^7 条指令/秒1163秒（20分钟内）：

通过使用运行时间增长较慢的算法，即使编译器较差，计算机 B 的运行速度也比计算机 A 快 17 倍以上！在对 1 亿个数字进行排序时，归并排序的优势更加明显：插入排序需要 23 天以上，而归并排序只需不到 4 个小时。虽然 1 亿个数字似乎很大，但每半小时就有超过 1 亿次网络搜索，每分钟有超过 1 亿封电子邮件发送，一些最小的星系（称为超紧凑矮星系）包含大约 1 亿颗恒星。一般来说，随着问题规模的增加，归并排序的相对优势也会增加。

算法和其他技术

上面的例子表明，你应该将算法视为 ***technology***，就像计算机硬件一样。整个系统的性能既取决于选择高效的算法，也取决于选择快速的硬件。正如其他计算机技术正在快速发展一样，算法也在快速发展。

你可能会想，考虑到其他先进技术，算法在当代计算机上是否真的那么重要，例如

先进的计算机架构和制造技术、• 易于使用、直观的图形用户界面（GUI）、• 面向对象的系统、• 集成 Web 技术、• 快速网络（有线和无线）、• 机器学习、• 和移动设备。

答案是肯定的。虽然有些应用程序没有明确要求在应用程序级别使用算法内容（例如一些简单的基于 Web 的应用程序），但许多应用程序确实需要。例如，考虑一个基于 Web 的服务，该服务确定如何从一个位置到另一个位置。它的实现将依赖于快速硬件、图形用户界面、广域网，也可能依赖于面向对象。它还需要算法来执行诸如查找路线（可能使用最短路径算法）、渲染地图和插入地址等操作。

此外，即使是在应用程序级别不需要算法内容的应用程序也严重依赖算法。应用程序是否依赖于快速硬件？硬件设计使用了算法。应用程序是否依赖于图形用户界面？任何 GUI 的设计都依赖于算法。应用程序是否依赖于网络？网络中的路由严重依赖于算法。应用程序是否使用机器码以外的语言编写？然后它由编译器、解释器或汇编器处理，所有这些都大量使用算法。算法是当代计算机中使用的大多数技术的核心。

机器学习可以被认为是一种执行算法任务的方法，无需明确设计算法，而是从数据中推断模式，从而自动学习解决方案。乍一看，机器学习可以自动化算法设计过程，似乎使算法学习变得过时。然而事实并非如此。机器学习本身就是算法的集合，只是名称不同。此外，目前看来，机器学习的成功似乎主要针对那些我们人类并不真正了解正确算法的问题。突出的例子包括计算机视觉和自动语言翻译。对于人类很好理解的算法问题，比如本书中的大多数问题，为解决特定问题而设计的高效算法通常比机器学习方法更为成功。

数据科学是一门跨学科领域，其目标是从结构化和非结构化数据中提取知识和见解。数据科学使用方法

来自统计学、计算机科学和优化。算法的设计和分析是该领域的基础。数据科学的核心技术与机器学习的核心技术有很大的重叠，包括本书中的许多算法。

此外，随着计算机能力的不断提升，我们用它们来解决的问题比以往任何时候都要大。正如我们在上文对插入排序和归并排序的比较中所看到的，在问题规模较大时，算法之间的效率差异尤为突出。

拥有扎实的算法知识和技术基础是真正熟练的程序员的特征之一。借助现代计算技术，您无需了解太多算法即可完成某些任务，但如果拥有良好的算法背景，您可以做更多的事情。

练习

1.2-1
给出一个在应用程序级需要算法内容的应用程序的示例，并讨论所涉及算法的功能。

1.2-2
假设在特定计算机上，对于大小为 n 的输入，插入排序需要 $8n^2$ 步，而归并排序需要 64 n lg n 步。对于哪些 n 值，插入排序优于归并排序？

1.2-3
在同一台机器上，使得运行时间为 $100n^2$ 的算法比运行时间为 2^n 的算法运行得更快的 n 的最小值是多少？

问题

1-1Comparison of running times
对于下表中的每个函数 f.n/ 和时间 t，确定可以在时间 t 内解决的问题的最大规模 n，假设解决该问题的算法需要 f.n/ 微秒。

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

章节注释

关于算法这一一般性话题，有许多优秀的文本，包括 Aho、Hopcroft 和 Ullman [5, 6]、Dasgupta、Papadimitriou 和 Vazirani [107]、Edmonds [133]、Erickson [135]、Goodrich 和 Tamassia [195, 196]、Kleinberg 和 Tardos [257]、Knuth [259, 260, 261, 262, 263]、Levitin [298]、Louridas [305]、Mehlhorn 和 Sanders [325]、Mitzenmacher 和 Upfal [331]、Neapolitan [342]、Roughgarden [385, 386, 387, 388]、Sanders、Mehlhorn、Dietzfelbinger 和 Dementiev [393]、Sedgewick 和 Wayne [402]、Skiena [414]、Soltys-Kulinicz [419]、Wilf [455]以及Williamson 和Shmoys [459]。Bentley [49, 50, 51]、Bhargava [54]、Kochenderfer 和 Wheeler [268]以及McGeoch [321]讨论了算法设计中一些更实际的方面。Atallah 和Blanton [27, 28]以及Mehta 和Sahhi [326]的书中也对算法领域进行了概述。有关非技术性材料，请参阅Christian 和Griffiths [92]、Cormen [104]、Erwig [136]、MacCormick [307]以及V ¨ocking 等人[448]的书籍。关于计算生物学中使用的算法的概述，可以在 Jones 和 Pevzner [240]、Elloumi 和 Zomaya [134] 以及 Marchisio [315] 的书籍中找到。

2 Getting Started

本章将帮助您熟悉我们将在整本书中用来思考算法设计和分析的框架。它是独立的，但它确实包含对第 3 章和第 4 章中介绍的材料的几个引用。（它还包含几个求和题，附录 A 显示了如何求解。）

我们将首先研究插入排序算法，以解决第 1 章中介绍的排序问题。我们将使用伪代码来指定算法，如果您做过计算机编程，那么您应该能够理解这些伪代码。我们将了解插入排序为何能正确排序，并分析其运行时间。分析引入了一种符号，描述了运行时间如何随着要排序的项目数量而增加。在讨论插入排序之后，我们将使用一种称为分而治之的方法来开发一种称为合并排序的排序算法。我们将以对合并排序的运行时间的分析结束。

2.1 插入排序

我们的第一个算法，插入排序，解决了第 1 章中介绍的 ***sorting problem***：

输入：n 个数字的序列 ha $_1$; a $_2$; : : : ; a $_n$ 我。

输出：输入序列的排列（重新排序）ha $'_1$; a $'_2$; : : : ; a $'_n$ i，使得 a $'_1$ a $'_2$ a $'_n$。

要排序的数字也称为 ***keys***。虽然问题在概念上是关于对序列进行排序，但输入是以包含 n 个元素的数组的形式出现的。当我们想要对数字进行排序时，通常是因为它们是与其他数据相关联的键，我们将其称为 ***satellite data***。键和卫星数据一起形成 ***record***。例如，考虑一个包含学生记录的电子表格，其中包含许多相关数据，例如年龄、平均绩点和所修课程数。这些数量中的任何一个都可以是键，但是当

电子表格排序时，它会将相关记录（卫星数据）与键一起移动。在描述排序算法时，我们关注的是键，但重要的是要记住，通常还有相关的卫星数据。

在本书中，我们通常将算法描述为用 ***pseu- docode*** 编写的程序，这种程序在很多方面类似于 C、C++、Java、Python、[1] 或 JavaScript。（如果我们遗漏了您最喜欢的编程语言，请原谅，因为我们无法全部列出。）如果您接触过这些语言中的任何一种，那么理解用伪代码 “ 编码的” 算法应该不难。伪代码与真实代码的区别在于，在伪代码中，我们采用最清晰、最简洁的表达方法来指定给定的算法。有时最清晰的方法是英语，所以如果您在更像真实代码的部分中遇到嵌入的英文短语或句子，请不要感到惊讶。伪代码和真实代码之间的另一个区别是，伪代码通常会忽略软件工程的某些方面，例如数据抽象、模块化和错误处理4，以便更简洁地传达算法的本质。

我们从 ***insertion sort*** 开始，这是一种对少量元素进行排序的有效算法。插入排序的工作方式与对一手扑克牌进行排序的方式类似。从空的左手开始，桌子上有一堆牌。拿起堆中的第一张牌并用左手握住它。然后，用右手从堆中一次取出一张牌，并将其插入左手的正确位置。如图 2.1 所示，通过将牌与左手中已有的每张牌进行比较，从右侧开始向左移动，可以找到牌的正确位置。只要您看到左手中的牌的值小于或等于右手中的牌，就将右手中的牌插入左手中这张牌的右侧。如果左手中所有牌的值都大于右手中的牌，则将这张牌作为左手最左边的牌。在任何时候，你左手所握的牌都是经过排序的，而且这些牌原本是桌子上那堆牌最上面的牌。

插入排序的伪代码在上页中以过程 I NSERTION-SORT 的形式给出。它需要两个参数：一个包含要排序的值的数组 A 和排序值的数量 n。这些值占据数组的位置 AŒ1 到 AŒn ，我们将其表示为 AŒ1 W n 。当 I NSERTION-SORT 过程完成后，数组 AŒ1 W n 包含原始值，但按排序顺序排列。

---

[1] If you’re familiar with only Python, you can think of arrays as similar to Python lists.

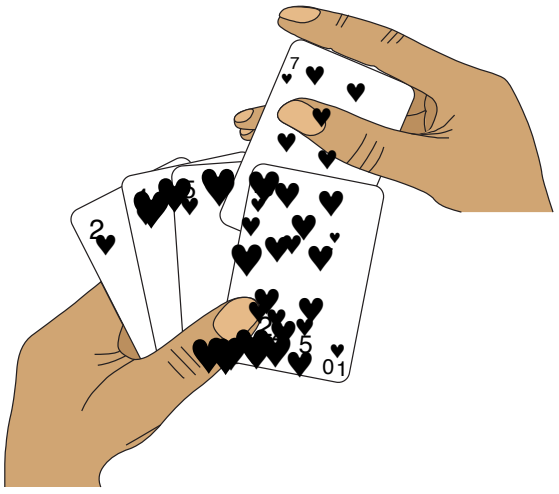


图 2.1 使用插入排序对一手牌进行排序。

插入排序 .A; n/

1 for i D 2 to n 2 *key* D AŒi 3 // 将 AŒi 插入到
已排序的子数组 AŒ1 W i 1 中。 4 j D i 1 5 wh
ile j > 0 and AŒj > *key* 6 AŒj C 1 D AŒj 7 j D
j 1 8 AŒj C 1 D *key*

大号 oop 不变量和插入的正确性 种类

图 2.2 显示了该算法如何对以序列 h5；2；4；6；1；3i 开头的数组 A 执行操作。索引 i 表示插入手中的 “ 当前牌 ”。在以 i 为索引的 for 循环每次迭代开始时，由元素 AŒ1 W i 1 （即 AŒ1 到 AŒi 1 ）组成的 ***subarray***（数组的连续部分）构成当前已排序的手牌，剩余的子数组 AŒi C 1 W n （元素 AŒi C 1 到 AŒn ）对应于仍在桌子上的那堆牌。实际上，元素 AŒ1 W i 1 是位置 1 到 i 1 的元素 *originally*，但现在是按排序顺序排列的。我们将 AŒ1 W i 的这些性质正式表述为 ***loop invariant***：

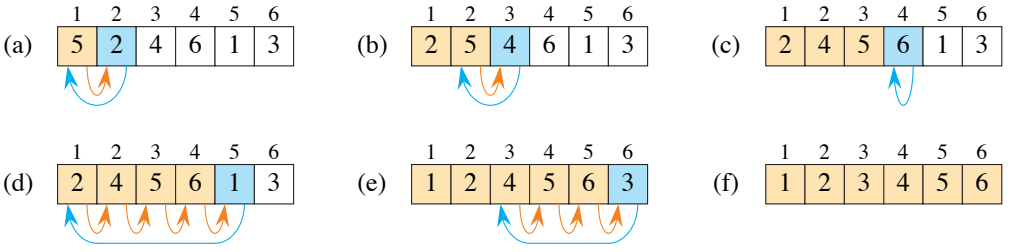


图 2.2 INSERTION-SORT.A；n/ 的操作，其中 A 最初包含序列 h5；2；4；6；1；3i 和 n D 6 。数组索引出现在矩形上方，存储在数组位置的值出现在矩形内。（a）–（e）第 138 行的 for 循环迭代。在每次迭代中，蓝色矩形保存从 AŒi 中获取的键，该键在第 5 行的测试中与其左侧的棕褐色矩形中的值进行比较。橙色箭头表示第 6 行中数组值向右移动一个位置，蓝色箭头表示第 8 行键移动到的位置。（f）最终排序后的数组。

在第 138 行的 for 循环每次迭代开始时，子数组 AŒ1 W i 1 由 AŒ 1 W i 1 中的原始元素组成，但按排序顺序排列。

循环不变量帮助我们理解算法为什么是正确的。使用循环不变量时，需要说明三点：

初始化：在循环第一次迭代之前为真。

维护：如果在循环迭代之前为真，则在下一次迭代之前它仍然为真。

终止：循环终止，当它终止时，不变量4通常与循环终止的原因一起给我们一个有用的属性，有助于证明算法是正确的。

当前两个属性成立时，循环不变量在循环的每次迭代之前都为真。（当然，您可以自由使用除循环不变量本身之外的既定事实来证明循环不变量在每次迭代之前保持为真。）循环不变量证明是一种数学归纳法，其中要证明属性成立，您需要证明一个基本情况和一个归纳步骤。在这里，证明不变量在第一次迭代之前成立对应于基本情况，而证明不变量在每次迭代中都成立对应于归纳步骤。

第三个属性可能是最重要的，因为您使用循环不变量来显示正确性。通常，您将循环不变量与导致循环终止的条件一起使用。数学归纳法通常无限地应用归纳步骤，但在循环不变量中，“归纳”在循环终止时停止。

让我们看看这些属性如何适用于插入排序。

初始化：我们首先证明循环不变量在第一次循环迭代之前成立，此时 i D 2.[2] 子数组 AŒ1 W i 1 仅包含单个元素 AŒ1 ，该元素实际上是 AŒ1 中的原始元素。此外，这个子数组是排序的（毕竟，只有一个值的子数组怎么可能不排序呢？），这表明循环不变量在循环的第一次迭代之前成立。

维护：接下来，我们处理第二个属性：显示每次迭代都保持循环不变量。非正式地，for 循环体的工作方式是将 AŒi 1 、AŒi 2 、AŒi 3 等中的值向右移动一个位置，直到找到 AŒi 的正确位置（第 437 行），此时它插入 AŒi 的值（第 8 行）。然后，子数组 AŒ1 W i 由 AŒ1 W i 中的原始元素组成，但按排序顺序排列。然后，for 循环的下一次迭代将 ***Incrementing*** i（将其值增加 1）保留循环不变量。

> 对第二个属性的更正式处理需要我们陈述并展示第 537 行 while 循环的循环不变量。我们暂时不要陷入这种形式主义。相反，我们将依靠非正式分析来证明第二个属性适用于外循环。

终止：最后，我们检查循环终止。循环变量 i 从 2 开始，每次迭代增加 1。一旦 i 的值在第 1 行超过 n，循环就会终止。也就是说，一旦 i 等于 n C 1，循环就会终止。在循环不变量的措辞中，用 n C 1 代替 i 可得出子数组 AŒ1 W n 由 AŒ1 W n 中的原始元素组成，但按排序顺序排列。因此，该算法是正确的。

本书各处均使用了循环不变量方法来证明正确性。

伪代码约定

我们在伪代码中使用以下约定。

缩进表示块结构。例如，从第 1 行开始的 for 循环体由第 238 行组成，而从第 248 行开始的 while 循环体由第 358 行组成。

---

[2] When the loop is a **for** loop, the loop-invariant check just prior to the first iteration occurs immediately after the initial assignment to the loop-counter variable and just before the first test in the loop header. In the case of INSERTION-SORT, this time is after assigning 2 to the variable i but before the first test of whether $i \leq n$.

从第 5 行开始包含第 637 行但不包含第 8 行。我们的缩进样式也适用于 if-else 语句[3]。使用缩进代替块结构的文本指示符（例如开始和结束语句或花括号）可减少混乱，同时保持甚至增强清晰度。[4]

循环结构 while、for 和 repeat-until 以及 if-else 条件结构的解释与 C、C++、Java、Python 和 JavaScript 中的解释类似。[5] 在本书中，循环计数器在退出循环后仍保留其值，这与 C++ 和 Java 中出现的某些情况不同。因此，在 for 循环之后，循环计数器的值是首次超出 for 循环界限的值。[6] 我们在插入排序的正确性论证中使用了这个属性。第 1 行的 for 循环头是 for i D 2 to n，因此当此循环终止时，i 等于 nC1。当 for 循环在每次迭代中增加其循环计数器时，我们使用关键字 to；当 for 循环在每次迭代中减少其循环计数器（将其值减少 1）时，我们使用关键字 downto。当循环计数器的变化量大于 1 时，变化量遵循可选关键字 by。

符号“//”表示该行的其余部分是注释。

变量（例如 i 、 j 和 *key*）是给定过程的局部变量。如果没有明确指示，我们不会使用全局变量。

我们通过指定数组名称和方括号中的索引来访问数组元素。例如，AŒi 表示数组 A 的第 i 个元素。

尽管许多编程语言都强制数组从 0 开始索引（0 是最小的有效索引），但我们选择最清晰易懂的索引方案。由于人们通常从 1 开始计数，而不是从 0 开始，因此本书中的大多数（但不是全部）数组都使用从 1 开始的索引。

---

[3] In an **if**-**else** statement, we indent **else** at the same level as its matching **if**. The first executable line of an **else** clause appears on the same line as the keyword **else**. For multiway tests, we use **elseif** for tests after the first one. When it is the first line in an **else** clause, an **if** statement appears on the line following **else** so that you do not misconstrue it as **elseif**.

[4] Each pseudocode procedure in this book appears on one page so that you do not need to discern levels of indentation in pseudocode that is split across pages.

[5] Most block-structured languages have equivalent constructs, though the exact syntax may differ. Python lacks **repeat**-**until** loops, and its **for** loops operate differently from the **for** loops in this book. Think of the pseudocode line “**for** $i = 1$ **to** n” as equivalent to “for i in range(1, n+1)” in Python.

[6] In Python, the loop counter retains its value after the loop is exited, but the value it retains is the value it had during the final iteration of the **for** loop, rather than the value that exceeded the loop bound. That is because a Python **for** loop iterates through a list, which may contain nonnumeric values.

为明确某个算法是否假设 0 源或 1 源索引，我们将明确指定数组的边界。如果您正在实现我们指定使用 1 源索引的算法，但您正在使用强制执行 0 源索引的编程语言（例如 C、C++、Java、Python 或 JavaScript）编写代码，那么您应该相信自己能够进行调整。您可以始终从每个索引中减去 1，也可以为每个数组分配一个额外的位置并忽略位置 0。

符号 “W” 表示子数组。因此，AŒi W j 表示由元素 AŒi ; AŒi C 1 ; : : : ; AŒj 组成的 A 子数组。[7] 我们还使用此符号来指示数组的边界，就像我们之前讨论数组 AŒ1 W n 时所做的那样。

我们通常将复合数据组织成***objects***，而***objects***由***attributes***组成。我们使用许多面向对象编程语言中的语法来访问特定属性：对象名称，后跟一个点，后跟属性名称。例如，如果对象 x 具有属性 f，我们将该属性表示为 x：f。

我们将表示数组或对象的变量视为指向表示数组或对象的数据的指针（在某些编程语言中称为引用）。对于对象 x 的所有属性 f，设置 y D x 会导致 y:f 等于 x:f。此外，如果我们现在设置 x:f D 3，那么之后不仅 x:f 等于 3，而且 y:f 也等于 3。换句话说，在赋值 y D x 之后，x 和 y 指向同一个对象。这种处理数组和对象的方式与大多数当代编程语言一致。

我们的属性符号可以“级联。”例如，假设属性 f 本身是指向具有属性 g 的某种类型的对象的指针。那么符号 x:f:g 被隐式地括起来为 .x:f/:g。换句话说，如果我们已将 y 赋值给 D x:f，那么 x:f:g 与 y:g 相同。

有时指针根本不指向任何对象。在这种情况下，我们赋予它特殊值 NIL。

我们将参数传递给过程 ***by value***：被调用过程接收其自己的参数副本，并且如果它为参数赋值，则调用过程会看到更改 *not*。传递对象时，指向表示对象的数据的指针会被复制，但对象的属性不会被复制。例如，如果 x 是被调用过程的参数，则在 x D y 中的赋值

---

[7] If you’re used to programming in Python, bear in mind that in this book, the subarray $A[i : j]$ includes the element $A[j]$. In Python, the last element of $A[i : j]$ is $A[j - 1]$. Python allows negative indices, which count from the back end of the list. This book does not use negative array indices.

被调用过程对调用过程不可见。但是，如果调用过程具有指向与 x 相同的对象的指针，则赋值 x:ƒ D 3 是可见的。类似地，数组是通过指针传递的，因此传递的是指向数组的指针，而不是整个数组，并且对单个数组元素的更改对调用过程是可见的。同样，大多数当代编程语言都是这样工作的。

返回语句立即将控制权转回调用过程中的调用点。大多数返回语句还会将一个值传回给调用者。我们的伪代码与许多编程语言不同，因为我们允许在单个返回语句中返回多个值，而无需创建对象将它们打包在一起。[8]

布尔运算符 “and” 和 “or” 是 ***short circuiting***。也就是说，通过首先计算 x 来计算表达式 “x and y ”。如果 x 计算结果为 FALSE，则整个表达式不能计算为 TRUE，因此不计算 y。另一方面，如果 x 计算结果为 TRUE，则必须计算 y 才能确定整个表达式的值。类似地，在表达式 “x or y ” 中，只有 x 计算结果为 FALSE 时才计算表达式 y。短路运算符允许我们编写布尔表达式，例如 “x ¤ NIL 和 x:ƒ D y ”，而不必担心在 x 为 NIL 时计算 x:ƒ 会发生什么。

关键字 error 表示由于调用过程的条件错误而发生错误，过程立即终止。调用过程负责处理错误，因此我们不指定要采取什么操作。

练习

2.1-1

以图 2.2 为模型，说明对初始包含序列 h31；41；59；26；41；58i 的数组执行 I NSERTION-SORT 操作。

2.1-2

考虑对页上的 SUM-ARRAY 过程。它计算数组 AŒ1 W n 中 n 个数字的总和。为此过程写出一个循环不变量，并使用其初始化、维护和终止属性来说明 SUM-ARRAY 过程返回 AŒ1 W n 中数字的总和。

---

[8] Python’s tuple notation allows **return** statements to return multiple values without creating objects from a programmer-defined class.

```
SUM-数组.A；n/
1 sum D 0 2 for i D 1 至 n 3
sum D sum C AŒi    4 返回
sum
```

2.1-3
重写 INSERTION-SORT 过程，按单调递减顺序而不是单调递增顺序排序。

2.1-4
考虑 ***searching problem***：

输入：存储在数组 AŒ1 W n 中的 n 个数字序列 ha $_1$; a $_2$; : : : ; a $_n$ i 和一个值 x 。
输出：索引 i，使得 x 等于 AŒi ，或者如果 x 不出现在 A 中，则为特殊值 NIL。

为 ***linear search*** 编写伪代码，从头到尾扫描数组，查找 x 。使用循环不变量证明你的算法是正确的。确保你的循环不变量满足三个必要的属性。

2.1-5
考虑两个 n 位二进制整数 a 和 b 的加法问题，这两个整数存储在两个 n 元素数组 AŒ0 W n 1 和 BŒ0 W n 1 中，其中每个元素均为 0 或 1，a D P $_{i=0}^{n-1}$ AŒi 2 ^{i}，b D P $_{i=0}^{n-1}$ BŒi 2 ^{i}。两个整数的和 c D a C b 应以二进制形式存储在 .n C 1/元素数组 C Œ0 W n 中，其中 c D P $_{i=0}^{n}$ C Œi 2 ^{i}。编写一个过程 ADD-BINARY-I NTEGERS，将数组 A 和 B 以及长度 n 作为输入，并返回保存和的数组 C。

2.2 分析算法

Analyzing算法已经成为预测算法所需的资源。您可能会考虑诸如内存、通信带宽或能耗等资源。然而，最常见的是，您需要测量计算时间。如果您分析了某个问题的几种候选算法，

你可以找出最有效的算法。可能存在不止一个可行的候选算法，但你通常可以在此过程中排除几种较差的算法。

在分析算法之前，您需要一个算法所基于的技术模型，包括该技术的资源以及表达其成本的方法。本书的大部分内容假设通用的单处理器 ***random-access ma- chine (RAM)*** 计算模型作为实现技术，并理解算法是以计算机程序的形式实现的。在 RAM 模型中，指令一个接一个地执行，没有并发操作。RAM 模型假设每条指令所花费的时间与任何其他指令相同，并且每次数据访问4（使用变量的值或将数据存储在变量中4）所花费的时间与任何其他数据访问相同。换句话说，在 RAM 模型中，每条指令或数据访问都需要恒定的时间4，即使索引到数组中也是如此。[9]

严格来说，我们应该精确定义 RAM 模型的指令及其成本。然而，这样做会很乏味，而且对算法设计和分析没有多大帮助。然而，我们必须小心，不要滥用 RAM 模型。例如，如果 RAM 中有一条排序指令，会怎样？那么只用一个步骤就可以排序。这样的 RAM 是不现实的，因为真实的计算机中不会出现这样的指令。因此，我们的指导原则是真实的计算机是如何设计的。RAM 模型包含真实的计算机中常见的指令：算术（如加、减、乘、除、求余、上下限、上限）、数据移动（加载、存储、复制）和控制（条件和无条件分支、子程序调用和返回）。

RAM 模型中的数据类型为整数、浮点数（用于存储实数近似值）和字符。实际计算机通常没有单独的布尔值 TRUE 和 FALSE 数据类型。相反，它们经常测试整数值是 0（FALSE）还是非零（TRUE），就像在 C 中一样。虽然我们通常不关心本书中浮点值的精度（许多数字不能用浮点数精确表示），但精度对于大多数应用程序来说至关重要。我们还假设每个数据字的位数都有限制。例如，当处理大小为 n 的输入时，我们通常

---

[9] We assume that each element of a given array occupies the same number of bytes and that the elements of a given array are stored in contiguous memory locations. For example, if array $A[1:n]$ starts at memory address 1000 and each element occupies four bytes, then element $A[i]$ is at address $1000 + 4(i - 1)$. In general, computing the address in memory of a particular array element requires at most one subtraction (no subtraction for a 0-origin array), one multiplication (often implemented as a shift operation if the element size is an exact power of 2), and one addition. Furthermore, for code that iterates through the elements of an array in order, an optimizing compiler can generate the address of each element using just one addition, by adding the element size to the address of the preceding element.

假设整数用 $c \log_2 n$ 位表示，其中 c 为常数 1。我们要求 c 为 1，这样每个字都可以保存 n 的值，使我们能够索引各个输入元素，并且我们将 c 限制为常数，这样字长就不会任意增长。（如果字长可以任意增长，我们可以在一个字中存储大量数据，并在常数时间内对其进行操作4——一种不切实际的情况。）

实际计算机包含上面未列出的指令，这类指令代表 RAM 模型中的灰色区域。例如，幂运算是恒定时间指令吗？一般情况下不是：当 x 和 n 是一般整数时，计算 x^n 通常需要 n 的对数时间（参见第 934 页的公式 (31.34)），并且您必须担心结果是否适合计算机字。但是，如果 n 是 2 的精确幂，则幂运算通常可看作是恒定时间运算。许多计算机都有 “左移” 指令，该指令在恒定时间内将整数的位向左移动 n 位。在大多数计算机中，将整数的位向左移动 1 位等效于乘以 2，因此将位向左移动 n 位等效于乘以 2^n。因此，此类计算机可以通过将整数 1 向左移动 n 个位置，在 1 条常量时间指令中计算 2^n，只要 n 不超过计算机字中的位数。我们将尝试在 RAM 模型中避免此类灰色区域，并将计算 2^n 和乘以 2^n 视为常量时间操作，前提是结果足够小，无法容纳在计算机字中。

RAM 模型不考虑当代计算机中常见的内存层次结构。它既不建模缓存也不建模虚拟内存。其他几个计算模型试图考虑内存层次结构的影响，这种影响有时在真实机器上的实际程序中很重要。本书的第 11.5 节和一些问题研究了内存层次结构的影响，但大多数情况下，本书中的分析并不考虑它们。包含内存层次结构的模型比 RAM 模型复杂得多，因此很难使用。此外，RAM 模型分析通常是实际机器性能的极好预测指标。

虽然在 RAM 模型中分析算法通常很简单，但有时却可能非常具有挑战性。您可能需要使用数学工具，例如组合学、概率论、代数技巧以及识别公式中最重要的项的能力。由于算法对于每个可能的输入可能有不同的行为，因此我们需要一种用简单、易于理解的公式总结该行为的方法。

插入排序分析

I NSERTION-SORT 过程需要多长时间？一种方法是，在计算机上运行它，并计算运行时间。当然，你会

首先，你必须用真正的编程语言来实现它，因为你不能直接运行我们的伪代码。这样的时间测试会告诉你什么？你会发现插入排序在你的特定计算机上、在特定输入下、在你创建的特定实现下、在你运行的特定编译器或解释器下、在你链接的特定库下以及在你的计算机上与时间测试同时运行的特定后台任务下（例如检查通过网络传入的信息）需要多长时间。如果你在计算机上使用相同的输入再次运行插入排序，你甚至可能会得到不同的时间结果。通过在一台计算机上仅运行一个插入排序实现和一个输入，如果你给它一个不同的输入，如果你在另一台计算机上运行它，或者如果你用不同的编程语言实现它，你能确定插入排序的运行时间吗？不多。我们需要一种方法来预测，给定一个新的输入，插入排序将需要多长时间。

我们可以通过分析算法本身来确定插入排序的运行时间，而不是计算一次甚至多次运行的时间。我们将检查它执行每行伪代码的次数以及每行伪代码的运行时间。我们首先会得出一个精确但复杂的运行时间公式。然后，我们将使用方便的符号提取公式的重要部分，这可以帮助我们比较同一问题的不同算法的运行时间。

我们如何分析插入排序？首先，让我们承认运行时间取决于输入。排序一千个数字比排序三个数字花费的时间更长，你不应该感到非常惊讶。此外，插入排序对两个大小相同的输入数组进行排序所需的时间也不同，这取决于它们已经排序的程度。尽管运行时间可能取决于输入的许多特征，但我们将重点关注已被证明具有最大影响的特征，即输入的大小，并将程序的运行时间描述为其输入大小的函数。为此，我们需要更仔细地定义术语“运行时间”和“输入大小”。我们还需要明确我们讨论的是导致最坏情况行为、最佳情况行为还是其他情况的输入的运行时间。

input size 的最佳表示法取决于所研究的问题。对于许多问题，例如排序或计算离散傅里叶变换，最自然的度量是 *number of items in the input*4（例如，排序项的数量 n）。对于许多其他问题，例如将两个整数相乘，输入大小的最佳度量是用普通二进制表示法表示输入所需的 *total number of bits*。有时用多个数字来描述输入的大小更为合适。例如，如果算法的输入是图形，我们通常用数字来表征输入的大小

图中顶点数和边数。我们将指出我们研究的每个问题所使用的输入大小度量

。

算法对特定输入的 ***running time*** 是执行的指令和数据访问的数量。我们如何计算这些成本应该独立于任何特定计算机，但在 RAM 模型的框架内。目前，让我们采用以下观点。执行每行伪代码都需要一个恒定的时间。一行可能比另一行花费更多或更少的时间，但我们假设第 k 行的每次执行都需要 c $_k$ 时间，其中 c $_k$ 是一个常数。这个观点与 RAM 模型保持一致，它也反映了伪代码在大多数实际计算机上的实现方式。[10]

让我们分析一下 INSERTION-SORT 过程。正如承诺的那样，我们将首先设计一个精确的公式，该公式使用输入大小和所有语句成本 c $_k$。然而，这个公式结果很乱。然后我们将切换到更简单的符号，它更简洁，更易于使用。这个更简单的符号清楚地说明了如何比较算法的运行时间，尤其是当输入的大小增加时。

为了分析 INSERTION-SORT 过程，让我们在下一页查看它，其中包含每个语句的时间成本和每个语句的执行次数。对于每个 i D 2; 3; : : : ; n，让 t $_i$ 表示针对该 i 值执行第 5 行中的 while 循环测试的次数。当 for 或 while 循环以通常的方式退出时4由于循环头中的测试结果为 FALSE4测试比循环体多执行一次。由于注释不是可执行语句，因此假设它们不占用任何时间。

算法的运行时间是每个执行语句的运行时间之和。执行需要 c $_k$ 步并执行 m 次的语句对总运行时间贡献 c $_k$ m。[11] 我们通常用 T.n/ 表示算法在大小为 n 的输入上的运行时间。要计算 T.n/（INSERTION-SORT 在 n 个值输入上的运行时间），我们将 *cost* 和 *times* 列的乘积相加，得到

---

[10] There are some subtleties here. Computational steps that we specify in English are often variants of a procedure that requires more than just a constant amount of time. For example, in the RADIX-SORT procedure on page 213, one line reads "use a stable sort to sort array A on digit i," which, as we shall see, takes more than a constant amount of time. Also, although a statement that calls a subroutine takes only constant time, the subroutine itself, once invoked, may take more. That is, we separate the process of ***calling*** the subroutine—passing parameters to it, etc.—from the process of ***executing*** the subroutine.

[11] This characteristic does not necessarily hold for a resource such as memory. A statement that references m words of memory and is executed n times does not necessarily reference mn distinct words of memory.

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

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i - 1) + c_7 \sum_{i=2}^{n}(t_i - 1) + c_8(n-1) .$$

即使对于给定大小的输入，算法的运行时间也可能取决于给定该大小的 *which* 输入。例如，在 INSERTION-SORT 中，最佳情况发生在数组已经排序时。在这种情况下，每次执行第 5 行时，*key*4（AŒi 中的原始值）的值已经大于或等于 AŒ1 W i 1 中的所有值，因此第 537 行的 while 循环总是在第 5 行的第一次测试后退出。因此，我们有 t $_i$ D 1 for i D 2; 3; : : : ; n，最佳情况的运行时间为

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned} \tag{2.1}$$

我们可以将此运行时间表示为 C b，其中 *constants* a 和 b 取决于语句成本 c $_k$ (，其中 a D c $_1$ Cc $_2$ Cc $_4$ Cc $_5$ Cc $_8$ 和 b D c $_2$ Cc $_4$ Cc $_5$ Cc $_8$)。因此，运行时间为 n 的 ***linear function***。

最坏的情况是数组按逆序排序4，即数组从降序开始。该过程必须将每个元素 AŒi 与整个排序子数组 AŒ1 W i 1 中的每个元素进行比较，因此 t $_i$ D i for i D 2; 3; : : : ; n。（该过程发现 AŒj 每次在第 5 行都 $> key$，并且只有当 j 达到 0 时才退出 while 循环。）注意

$$\begin{aligned} \sum_{i=2}^{n} i &= \left(\sum_{i=1}^{n} i\right) - 1 \\ &= \frac{n(n+1)}{2} - 1 \quad \text{(by equation (A.2) on page 1141)} \end{aligned}$$

和

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i$$
$$= \frac{n(n-1)}{2} \quad \text{(again, by equation (A.2))} ,$$

我们发现，在最坏的情况下，INSERTION-SORT 的运行时间为

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned} \tag{2.2}$$

我们可以将这个最坏情况的运行时间表示为 ^{2} C b n C c，其中常数 a、b 和 c 又取决于语句成本 c $_k$ (now、a D c $_5$/2 C c $_6$/2 C c $_7$/2、b D c $_1$C c $_2$C c $_4$C c $_5$/2 c $_6$/2 c $_7$/2 C c $_8$ 和 c D .c $_2$C c $_4$C c $_5$C c $_8$/)。因此，运行时间为 n 的 ***quadratic function***。

通常，与插入排序一样，算法的运行时间对于给定的输入是固定的，尽管我们也会看到一些有趣的“随机化”算法，即使对于固定的输入，其行为也会发生变化。

最坏情况和平均情况分析

我们对插入排序的分析考虑了最佳情况（输入数组已经排序）和最坏情况（输入数组反向排序）。不过，在本书的其余部分，我们通常（但并非总是）专注于仅找到 ***worst-case running time***，即大小为 n 的 *any* 输入的最长运行时间。为什么？原因如下：

算法的最坏情况运行时间为 *any* 输入的运行时间提供了一个上限。如果你知道这个上限，那么你就可以保证算法永远不会再花费更多时间。你不需要对运行时间做出一些有根据的猜测，并希望它永远不会变得更糟。这个特性对于实时计算尤其重要，因为实时计算中的操作必须在截止期限前完成。

对于某些算法来说，最坏情况经常发生。例如，在数据库中搜索特定信息时，搜索算法的最坏情况通常发生在数据库中不存在该信息时。在某些应用中，对缺失信息的搜索可能很频繁。

“平均情况”通常与最坏情况大致一样糟糕。假设你对一个由 n 个随机选择的数字组成的数组运行插入排序。需要多长时间才能确定在子数组 AŒ1 W i 1 中的哪个位置插入元素 AŒi ？平均而言，AŒ1 W i 1 中有一半元素小于 AŒi ，而另一半元素大于 AŒi 。因此，平均而言，AŒi 仅与子数组 AŒ1 W i 1 的一半进行比较，因此 t_i 约为 i/2。结果得出的平均情况运行时间与最坏情况运行时间一样，是输入大小的二次函数。

在某些特殊情况下，我们会对算法的 ***average-case*** 运行时间感兴趣。我们将在本书中看到 ***probabilistic analysis*** 技术应用于各种算法。平均情况分析的范围有限，因为对于特定问题，什么构成“平均”输入可能并不明显。通常，我们会假设给定大小的所有输入都具有相同的可能性。在实践中，这个假设可能会被违反，但我们有时可以使用 ***randomized algorithm***（它会进行随机选择）来进行概率分析并得出 ***expected*** 运行时间。我们将在第 5 章和其他几个后续章节中进一步探讨随机算法。

成长顺序

为了便于分析 INSERTION-SORT 过程，我们使用了一些简化的抽象。首先，我们忽略每个语句的实际成本，使用常量 c_k 来表示这些成本。但是，等式 (2.1) 和 (2.2) 中的最佳情况和最坏情况运行时间相当笨拙。这些表达式中的常量为我们提供的细节比我们真正需要的要多。这就是为什么我们还将最佳情况运行时间表示为 C b，其中常量 a 和 b 取决于语句成本 c_k，以及将最坏情况运行时间表示为 ² C b n C c，其中常量 a、b 和 c 取决于语句成本。因此，我们不仅忽略了实际语句成本，还忽略了抽象成本 c_k。

现在让我们再做一个简化的抽象：我们真正感兴趣的是运行时间的 ***rate of growth*** 或 ***order of growth***。因此，我们只考虑公式的首项（例如，²），因为对于较大的 n 值，低阶项相对不重要。我们还忽略首项的常数系数，因为在确定大输入的计算效率方面，常数因子不如增长率重要。对于插入排序的最坏情况运行时间，当我们忽略低阶项和首项的常数系数时，只剩下首项的因子 n ²。该因子 n ² 是运行时间中最重要的部分。例如，假设在特定机器上实现的算法在大小为 n 的输入上花费 n ²/100 C 100n C 17 微秒。尽管 n ² 项的 1/100 系数和 n 项的 100 系数相差四个数量级，但 n ²/100 项占主导地位

一旦 n 超过 10,000，就会取消 100n 项。虽然 10,000 看起来很大，但它比一个普通城镇的人口还要少。许多现实世界的问题都有更大的输入规模。

为了突出运行时间的增长顺序，我们有一个特殊的符号，使用希腊字母 ，（theta）。我们写插入排序的最坏情况运行时间为 ，.n 2 /（发音为 “theta of n-squared” 或简称为 “theta n-squared”）。我们还写插入排序的最佳情况运行时间为 ，.n/（“theta of n” 或 “theta n”）。现在，将 ，符号想象为 “当 n 很大时大致成比例”，因此 ，.n 2 / 表示 “当 n 很大时大致与 n 2 成比例”，而 ，.n/ 表示 “当 n 很大时大致与 n 成比例”。我们将在本章中非正式地使用 ，符号，并在第 3 章中对其进行精确定义。

如果一种算法在最坏情况下的运行时间增长阶较低，我们通常认为该算法比另一种算法更高效。由于常数因子和低阶项的原因，对于较小的输入，运行时间增长阶较高的算法可能比运行时间增长阶较低的算法花费的时间更少。但是对于足够大的输入，例如，最坏情况运行时间为 ，.n 2 / 的算法在最坏情况下比最坏情况运行时间为 ，.n 3 / 的算法花费的时间更少。不管 ，符号隐藏的常数是什么，总有某个数，比如 n 0，使得对于所有的输入大小 n ≈ n 0， ，.n 2 / 算法在最坏情况下胜过 ，.n 3 / 算法。

练习

2.2-1用，符号表示函数 $n^3/1000$ C $100n^2$ 100n C 3。

2.2-2

考虑对数组 AŒ1 W n 中存储的 n 个数字进行排序，首先找到 AŒ1 W n 中的最小元素并将其与 AŒ1 中的元素交换。然后找到 AŒ2 W n 中的最小元素，并将其与 AŒ2 交换。然后找到 AŒ3 W n 中的最小元素，并将其与 AŒ3 交换。继续以此方式对 A 的前 n 1 个元素进行排序。编写该算法的伪代码，称为 ***selection sort***。该算法保持什么循环不变量？为什么它只需要对前 n 1 个元素运行，而不是对所有 n 个元素运行？给出 ，表示法中选择排序的最坏情况运行时间。最佳情况的运行时间会更好吗？

2.2-3

再次考虑线性搜索（参见练习 2.1-4）。假设被搜索的元素是数组中任何元素的可能性都相等，那么平均需要检查输入数组中的多少个元素？最坏的情况又如何呢？

使用，符号，给出线性搜索的平均情况和最坏情况的运行时间。证明你的答案的合理性。

2.2-4如何修改任何排序算法以获得良好的最佳运行时间？

2.3 设计算法

您可以从多种算法设计技术中进行选择。插入排序使用 ***incremental*** 方法：对于每个元素 AŒi ，将其插入子数组 AŒ1 W i 中的适当位置，此时已经对子数组 AŒ1 W i 1 进行排序。

本节将介绍另一种设计方法，即“分治法”，我们将在第 4 章中详细探讨该方法。我们将使用分治法设计一种排序算法，其最坏情况运行时间比插入排序少得多。使用遵循分治法的算法的一个优点是，使用我们将在第 4 章中探讨的技术，分析其运行时间通常很简单。

2.3.1 分治法

许多有用的算法在结构上都是 ***recursive***：为了解决给定的问题，它们会一次或多次 ***recurse***（调用自身）来处理密切相关的子问题。这些算法通常遵循 ***divide-and-conquer*** 方法：它们将问题分解为几个与原始问题类似但规模较小的子问题，递归解决子问题，然后组合这些解决方案以创建原始问题的解决方案。

在分治法中，如果问题足够小4，那么***base case***4你只需直接解决它而无需递归。否则4，那么***recursive case*** 4你执行三个典型步骤：

将问题划分为一个或多个子问题，这些子问题都是同一问题的较小实例。

通过递归解决子问题来解决它们。

结合 ubproblem 解决方案形成原始解决方案 最终问题。

merge sort 算法严格遵循分治法。在每一步中，它对子数组 AŒp W r 进行排序，从整个数组 AŒ1 W n 开始，然后递归到越来越小的子数组。合并排序的工作原理如下：

将要排序的子数组 AŒp W r 分成两个相邻的子数组，每个子数组的大小为原来的一半。为此，计算 AŒp W r 的中点 q（取 p 和 r 的平均值），然后将 AŒp W r 分成子数组 AŒp W q 和 AŒq C 1 W r 。
通过使用归并排序对两个子数组 AŒp W q 和 AŒq C 1 W r 进行递归排序来征服。
将两个已排序的子数组 AŒp W q 和 AŒq C 1 W r 合并回 AŒp W r ，得到已排序的答案。
递归 “ 触底”4当要排序的子数组 AŒp W r 仅有 1 个元素时（即 p 等于 r 时），它达到基本情况4。正如我们在 INSERTION-SORT 循环不变量的初始化参数中指出的那样，仅包含一个元素的子数组始终是已排序的。

合并排序算法的关键操作发生在 “combine” 步骤中，该步骤将两个相邻的已排序子数组合并。合并操作由下页的辅助过程 MERGE.A; p; q; r / 执行，其中 A 是一个数组，p、q 和 r 是数组的索引，并且 p q < r 。该过程假设相邻子数组 AŒp W q 和 AŒq C 1 W r 已经递归排序。它 ***merges*** 两个已排序子数组形成一个已排序子数组，以替换当前子数组 AŒp W r 。

要理解 MERGE 过程的工作原理，让我们回到打牌的主题。假设桌子上有两堆牌，面朝上。每堆都经过排序，最小面值的牌放在最上面。您希望将这两堆合并为一个排序的输出堆，面朝下放在桌子上。基本步骤包括选择面朝上的两堆牌中较小的一张，将其从其所在的堆中移除，露出一张新的顶牌，然后将这张牌面朝下放在输出堆上。重复此步骤，直到一个输入堆为空，此时您可以拿起剩余的输入堆，将其翻过整个堆，面朝下放在输出堆上。

让我们考虑一下合并两堆已排序的牌需要多长时间。每个基本步骤都需要常数时间，因为你只比较最上面的两张牌。如果你开始的两个已排序的牌堆各有 n/2 张牌，那么基本步骤数至少为 n/2（因为无论清空哪堆，每张牌都比另一堆中的某张牌小），最多为 n（实际上，最多为 n 1，因为在 n 1 个基本步骤之后，其中一个堆必定是空的）。由于每个基本步骤都需要常数时间，并且基本步骤总数介于 n/2 和 n 之间，我们可以说合并所需的时间大致与 n 成正比。也就是说，合并需要 ，.n/ 时间。

具体来说，MERGE 过程如下。它将两个子数组 AŒp W q 和 AŒq C 1 W r 复制到临时数组 L 和 R（“left” 和 “right”）中，然后将 L 和 R 中的值合并回 AŒp W r 。第 1 行和第 2 行分别计算子数组 AŒp W q 和 AŒq C 1 W r 的长度 n_L 和 n_R。然后

```
合并。A；p；q；r/

 1 n_L D q  p C 1 // AŒp W q   的长度 2 n_R D r  q // AŒq C 1 W r
的长度 3 令 LŒ0 W n_L  1   和 RŒ0 W n_R  1   为新数组 4 对于 i D 0
至 n_L  1 // 将 AŒp W q   复制到 LŒ0 W n_L  1   5 LŒi   D AŒp C i
   6 对于 j D 0 至 n_R  1 // 将 AŒq C 1 W r   复制到 RŒ0 W n_R  1   7
RŒj   D AŒq C j C 1   8 i D 0 // i 索引 L 中剩余的最小元素 9 j D 0 //
j 索引 R 中剩余的最小元素 10 k D p // k 索引 A 中到 ûll 11 的位置 // 只
要数组 L 和 R 中的每一个包含未合并元素，// 就会将最小的未合并元
素复制回 AŒp W r   中。 12 while i < n_L and j < n_R 13 if LŒi       R
Œj   14 AŒk   D LŒi   15 i D i C 1 16 else AŒk   D RŒj   17 j D j
C 1 18 k D k C 1 19 // 完整遍历 L 和 R 中的一个后，将另一个的 // 余数
复制到 AŒp W r   的末尾。 20 while i < n_L 21 AŒk   D LŒi   22 i
D i C 1 23 k D k C 1 24 while j < n_R 25 AŒk   D RŒj   26 j D j C 1 27
k D k C 1
```

第 3 行创建数组 LŒ0 W n_L 1   和 RŒ0 W n_R 1   ，长度分别为 n_L 和 n_R 。[12] 第 435 行的 for 循环将子数组 AŒp W q   复制到 L 中，第 637 行的 for 循环将子数组 AŒq C 1 W r   复制到 R 中。

图 2.3 中所示的 8318 行执行基本步骤。12318 行的 while 循环重复识别 L 和 R 中尚未被计算的最小值。

---

[12] This procedure is the rare case that uses both 1-origin indexing (for array A) and 0-origin indexing (for arrays L and R). Using 0-origin indexing for L and R makes for a simpler loop invariant in Exercise 2.3-3.

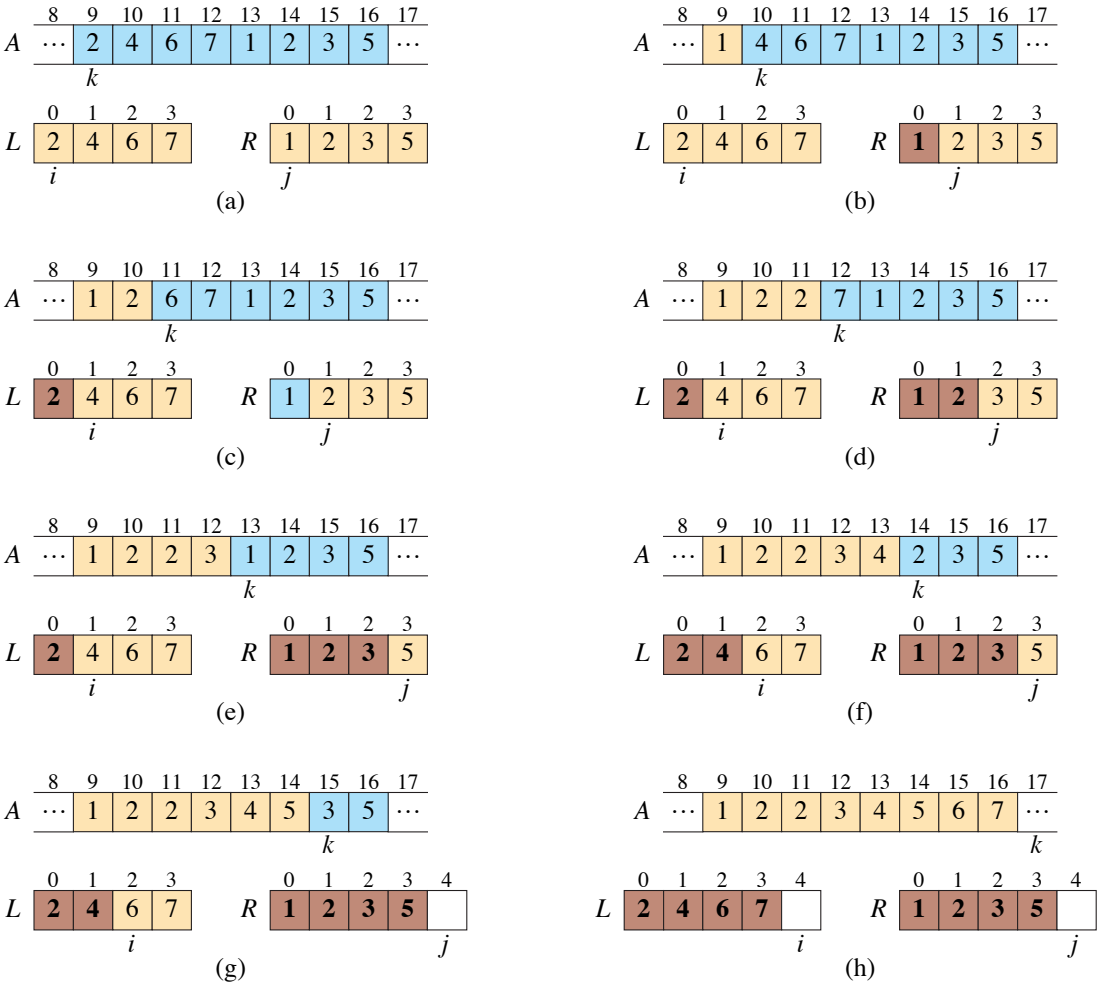


图 2.3 调用 M ERGE.A; 9; 12; 16/ 中第 83 行至第 18 行 while 循环的运行情况，此时子数组 A ŒŒ9 W 16 包含值 h2; 4; 6; 7; 1; 2; 3; 5i。分配并复制到数组 L 和 R 后，数组 L 包含 h2; 4; 6; 7i，数组 R 包含 h1; 2; 3; 5i。A 中的 tan 位置包含它们的最终值，L 和 R 中的 tan 位置包含尚未复制回 A 的值。综合起来，tan 位置始终包含 AŒŒ9 W 16 中的原始值。A 中的蓝色位置包含将要复制的值，而 L 和 R 中的深色位置包含已经复制回 A 的值。（a）–（g）第 12318 行循环每次迭代之前的数组 A、L 和 R，以及它们各自的索引 k、i 和 j。在部分 (g) 中的点，R 中的所有值都已复制回 A（j 等于 R 的长度表示），因此第 12318 行的 while 循环终止。（h）终止时的数组和索引。第 20323 行和第 24327 行的 while 循环将 L 和 R 中剩余的值复制回 A，这些值是 AŒŒ9 W 16 中最初的最大值。这里，第 20323 行将 LŒŒ2 W 3 复制到 A ŒŒ15 W 16 中，由于 R 中的所有值都已复制回 A，因此第 24327 行的 while 循环迭代了 0 次。此时，AŒŒ9 W 16 中的子数组已排序。

被复制回 AŒp W r 并将其复制回去。如注释所示，索引 k 给出被填充的 A 的位置，索引 i 和 j 分别给出剩余最小值在 L 和 R 中的位置。最终，将 L 的全部或 R 的全部复制回 AŒp W r ，此循环终止。如果循环终止是因为 R 的全部都已被复制回来，即因为 j 等于 n_R，则 i 仍然小于 n_L，因此 L 的某些值尚未复制回来，并且这些值在 L 和 R 中都是最大的。在这种情况下，第 20323 行的 while 循环将 L 的这些剩余值复制到 AŒp W r 的最后几个位置中。因为 j 等于 n_R，所以第 24327 行的 while 循环迭代 0 次。如果第 12318 行的 while 循环因为 i 等于 n_L 而终止，那么所有 L 都已被复制回 AŒp W r 中，而第 24327 行的 while 循环会将 R 的剩余值复制回 AŒp W r 的末尾。

要查看 MERGE 过程在 ，.n/ 时间内运行，其中 n D r p C 1,[13] ，请观察第 133 行和第 8310 行中的每一行都花费恒定时间，第 437 行的 for 循环花费 ，.n_L C n_R/ D ，.n/ 时间。[14] 要解释第 12318 行、第 20323 行和第 24327 行的三个 while 循环，请观察这些循环的每次迭代都会将 L 或 R 中的一个值复制回 A，并且每个值都会恰好复制回 A 一次。因此，这三个循环总共需要 n 次迭代。由于三个循环中的每次迭代都需要恒定时间，因此这三个循环所花费的总时间为 ，.n/。

现在，我们可以将 MERGE 过程用作合并排序算法的子程序。对页上的过程 MERGE-SORT.A; p; r / 对子数组 AŒp W r 中的元素进行排序。如果 p 等于 r，则子数组仅有 1 个元素，因此已经排好序。否则，我们必须有 p < r，然后 MERGE-SORT 运行分治和合并步骤。除法步骤只是计算一个索引 q，将 AŒp W r 分成两个相邻的子数组：AŒp W q ，包含 dn/2e 个元素，以及 AŒq C 1 W r ，包含 bn/2c 个元素。[15] 初始调用 MERGE-SORT .A; 1; n/ 对整个数组 AŒ1 W n 进行排序。

图 2.4 说明了该算法对 n D 8 的操作，同时还显示了划分和合并步骤的顺序。该算法以递归方式将数组划分为 1 个元素的子数组。合并步骤将 1 个元素的子数组对合并为 1 个元素的子数组。

---

[13] If you're wondering where the "+1" comes from, imagine that $r = p + 1$. Then the subarray $A[p : r]$ consists of two elements, and $r - p + 1 = 2$.

[14] Chapter 3 shows how to formally interpret equations containing Θ-notation.

[15] The expression $\lceil x \rceil$ denotes the least integer greater than or equal to x, and $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x. These notations are defined in Section 3.3. The easiest way to verify that setting q to $\lfloor (p + r)/2 \rfloor$ yields subarrays $A[p : q]$ and $A[q + 1 : r]$ of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively, is to examine the four cases that arise depending on whether each of p and r is odd or even.

```
MERGE-SORT(A, p, r)
1  if p ≥ r                                  // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                           // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)                       // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)                   // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

射线形成长度为 2 的排序子数组，将这些子数组合并形成长度为 4 的排序子数组，然后将这些子数组合并形成长度为 8 的最后一个排序子数组。如果 n 不是 2 的精确幂，则某些除法步骤会创建长度相差 1 的子数组。（例如，在除一个长度为 7 的子数组时，一个子数组的长度为 4，另一个子数组的长度为 3。）无论合并的两个子数组的长度是多少，合并总共 n 个项目的时间都是“ .n/ ”。

2.3.2 分析分治算法

当算法包含递归调用时，您通常可以用 ***recurrence equation*** 或 ***recurrence*** 来描述其运行时间，这描述了规模为 n 的问题的总体运行时间，即同一算法在较小输入下的运行时间。然后，您可以使用数学工具来解决递归问题，并为算法的性能提供界限。

分治算法运行时间的递归公式源自基本方法的三个步骤。与插入排序一样，设 T .n/ 为大小为 n 的问题的最坏情况运行时间。如果问题规模足够小，比如 n < n 0，其中 n 0 > 为某个常数，则直接求解所需的时间为常数，我们将其写为 ，.1/。[16] 假设问题的划分产生一个子问题，每个子问题的大小为 n /b，即原始子问题的大小为 1/b。对于合并排序，a 和 b 都是 2，但我们会看到其他分治算法，其中 a ¤ b。解决一个大小为 n/b 的子问题需要 T .n/b/ 时间，因此解决所有子问题需要 aT .n/b/ 时间。如果需要 D.n/ 时间将问题划分为子问题，并需要 C.n/ 时间将子问题的解组合成原问题的解，则我们得到递归

[16] If you're wondering where $\Theta(1)$ comes from, think of it this way. When we say that $n^2/100$ is $\Theta(n^2)$, we are ignoring the coefficient $1/100$ of the factor n^2. Likewise, when we say that a constant c is $\Theta(1)$, we are ignoring the coefficient c of the factor 1 (which you can also think of as n^0).

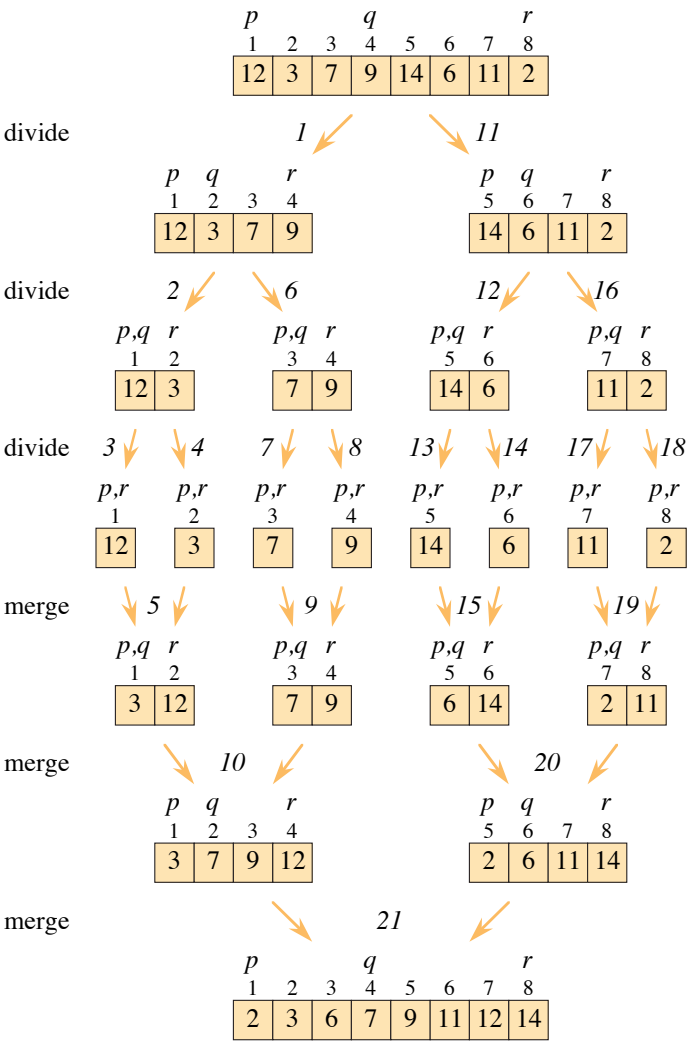


图 2.4 对长度为 8 的数组 A 执行归并排序操作，该数组最初包含序列 h12；3；7；9；14；6；11；2i。每个子数组的索引 p、q 和 r 出现在它们的值上方。斜体数字表示在首次调用 M E RGE-SORT.A；1；8/ 之后调用 M ERGE-SORT 和 MERGE 过程的顺序。

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \text{ ,} \\ D(n) + aT(n/b) + C(n) & \text{otherwise .} \end{cases}$$

第 4 章展示如何解决这种形式的常见递归。

有时，划分步骤的大小 n/b 不是整数。例如，MERGE-SORT 过程将大小为 n 的问题划分为大小为 dn/2e 和 bn/2c 的子问题。由于 dn/2e 和 bn/2c 之间的差最多为 1，

对于较大的 n，这比将 n 除以 2 的效果要小得多，我们稍微眯起眼睛，把它们都称为大小 n/2。正如第 4 章将要讨论的那样，这种忽略下限和上限的简化通常不会影响分治递归解的增长顺序。

我们将采用的另一个惯例是省略递归基本情况的陈述，我们也将在第 4 章中更详细地讨论这一点。原因是基本情况几乎总是 T .n/ D ，.1/，如果 n < n $_0$ 为某个常数 n $_0$ > 0。这是因为算法在常数大小的输入上的运行时间是常数。通过采用这一惯例，我们可以节省大量额外的写作时间。

归并排序分析

下面说明了如何设置 T.n/ 的递归，即对 n 个数字进行合并排序的最坏情况运行时间。

除法：除法步骤仅计算子数组的中间值，这需要常数时间。因此，D.n/ D ，.1/。

征服：递归地解决两个子问题，每个子问题的大小为 n/2，为运行时间贡献 2T .n/2/（忽略我们所讨论的最低限度和最高限度）。

合并：由于对 n 个元素子数组的 MERGE 过程花费 ，.n/ 时间，因此我们得到 C.n/ D ，.n/。

当我们将函数 D.n/ 和 C.n/ 相加用于归并排序分析时，我们实际上添加了一个函数 ，.n/ 和一个函数 ，.1/。这个和是 n 的线性函数。也就是说，当 n 很大时，它大致与 n 成正比，因此归并排序的划分和合并时间加起来为 ，.n/。将 ，.n/ 添加到征服步骤中的 2T .n/2/ 项中，可得出归并排序最坏情况运行时间 T .n/ 的递归式：

$$T(n) = 2T(n/2) + \Theta(n) . \tag{2.3}$$

第 4 章介绍了主定理 “， ”，该定理表明 T .n/ D ，.n lg n/。[17] 与最坏情况运行时间为 ，.n ^{2} / 的插入排序相比，归并排序用 n 的倍数换取了 lg n 的倍数。由于对数函数的增长速度比任何线性函数都慢，因此这是一笔不错的交易。对于足够大的输入，归并排序的最坏情况运行时间为 ，.n lg n/，优于最坏情况运行时间为 ，.n ^{2} / 的插入排序。

---

[17] The notation lg n stands for $\log_2 n$, although the base of the logarithm doesn't matter here, but as computer scientists, we like logarithms base 2. Section 3.3 discusses other standard notation.

然而，我们不需要主定理就能直观地理解为什么递归 (2.3) 的解是 T .n/ D ，.n lg n/。为简单起见，假设 n 是 2 的精确幂，隐含的基例是 n D 1。那么递归 (2.3) 本质上就是

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 , \\ 2T(n/2) + c_2 n & \text{if } n > 1 , \end{cases} \tag{2.4}$$

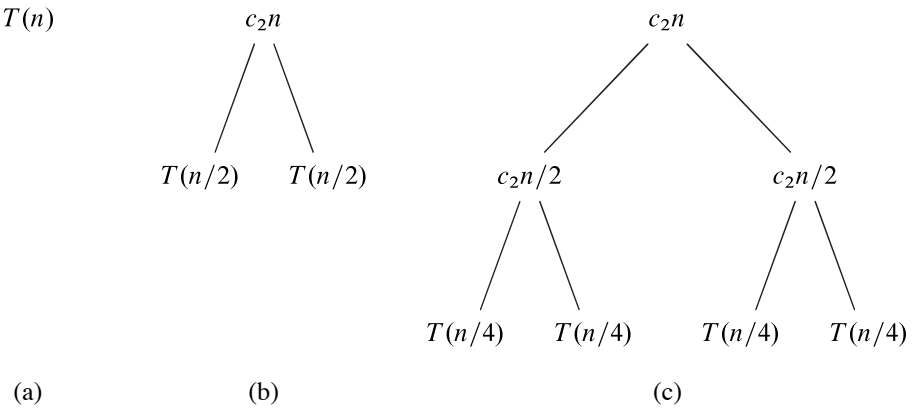
其中常数 c 1 > 0 表示解决大小为 1 的问题所需的时间，c 2 > 0 是划分和合并步骤中每个数组元素所需的时间。[18]

图 2.5 说明了求解递归 (2.4) 的一种方式。图的 (a) 部分显示 T .n/，而 (b) 部分将其扩展为表示递归的等效树。c 2 n 项表示在递归顶层进行划分和组合的成本，根的两个子树是两个较小的递归 T .n/2/。图 (c) 部分通过扩展 T .n/ 2/ 显示了此过程的进一步发展。在第二层递归的两个节点上进行划分和组合的成本为 c 2 n/2。继续扩展树中的每个节点，将其分解为由递归确定的组成部分，直到问题规模降至 1，每个节点的成本为 c 1。图 (d) 部分显示了结果 ***recursion tree***。

接下来，将树中每一层的成本相加。顶层的总成本为 c 2 n，下一层的总成本为 c 2 .n/2/ C c 2 .n/2/ D c 2 n，再下一层的总成本为 c 2 .n/4/ C c 2 .n/4/ C c 2 .n/4/ C c 2 .n/4/ D c 2 n，以此类推。每一层的节点数都是上一层的两倍，但每个节点的成本仅为上一层节点成本的一半。从一层到下一层，加倍和减半相互抵消，因此每一层的成本相同：c 2 n。一般而言，顶层以下 i 级有 2^i 个节点，每个节点贡献的成本为 c 2 .n/2^i/，因此顶层以下第 i 级的总成本为 2^i c 2 .n/2^i/ D c 2 n。底层有 n 个节点，每个节点贡献的成本为 c 1，总成本为 c 1 n。

图 2.5 中的递归树的总层数为 lg n C 1，其中 n 是叶子节点的数量，与输入大小相对应。一个非正式的归纳论证支持这一说法。当 n D 1 时，会发生基本情况，在这种情况下树只有 1 个层。由于 lg 1 D 0，我们可以得出 lg n C 1 给出了正确的层数。现在假设一个归纳假设，具有 2^i 个叶子节点的递归树的层数为 lg 2^i C 1 D i C 1（因为对于任何 i 值，我们都有 lg 2^i D i）。因为我们假设输入大小是 2 的幂，所以下一个要考虑的输入大小是 2^{i+1}。具有 n D 2^{i+1} 个叶子节点的树还有 1 个

[18] It is unlikely that c_1 is exactly the time to solve problems of size 1 and that $c_2 n$ is exactly the time of the divide and combine steps. We'll look more closely at bounding recurrences in Chapter 4, where we'll be more careful about this kind of detail.



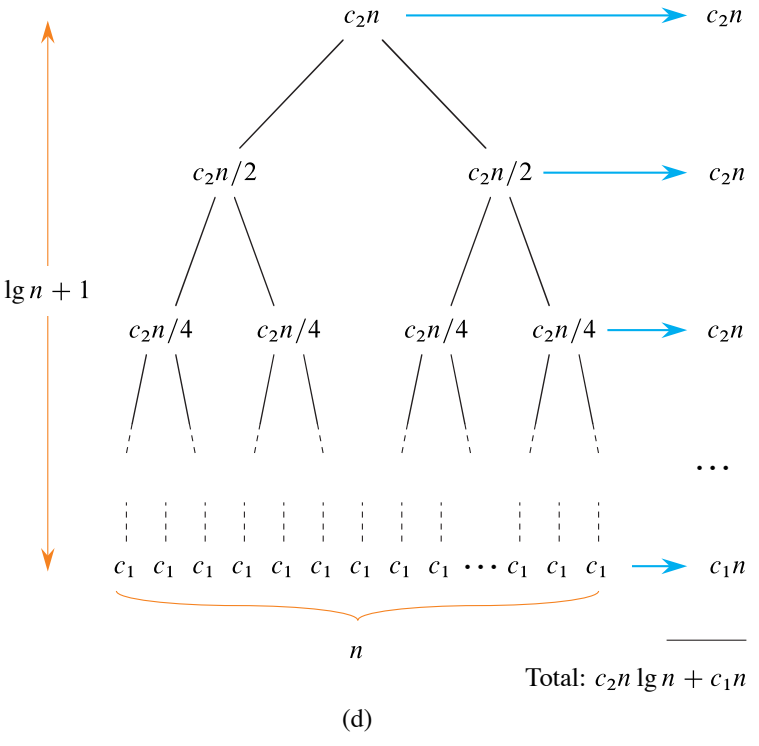


图 2.5 如何为递归 (2.4) 构造递归树。部分 (a) 显示 T .n/，它在 (b) – (d) 中逐步扩展以形成递归树。部分 (d) 中完全展开的树有 lg n C 1 层。叶子层以上的每一层总成本为 c 2 n，叶子层总成本为 c 1 n。因此，总成本为 c 2 n lg n C c 1 n D ，.n lg n/。

级别比具有 2^i 个叶子节点的树高，因此总级别数为 .i C 1/ C 1 D lg 2^{i+1} C 1 。

要计算递归 (2.4) 所表示的总成本，只需将所有级别的成本相加即可。递归树有 lg n C 1 个级别。叶子上方的每个级别成本为 c_2n，叶子级别成本为 c_1n，总成本为 c_2n lg n Cc_1n D ，.n lg n/。

练习

2.3-1
以图 2.4 为模型，说明对初始包含序列 h3；41；52；26；38；57；9；49i 的数组进行归并排序的操作。

2.3-2
MERGE-SORT 过程第 1 行的测试为 “if p ? r ”，而不是 “if p ¤ r .”。如果使用 p > r 调用 MERGE-SORT，则子数组 AŒp W r 为空。论证只要 MERGE-SORT.A; 1; n/ 的初始调用有 n ? 1，测试 “if p ¤ r ” 就足以确保没有递归调用有 p > r。

2.3-3
为 MERGE 过程第 12318 行的 while 循环陈述一个循环不变量。说明如何使用它以及第 20323 行和第 24327 行的 while 循环来证明 MERGE 过程是正确的。

2.3-4
用数学归纳法证明，当n2为2的精确幂时，递归式的解

$$T(n) = \begin{cases} 2 & \text{if } n = 2\,, \\ 2T(n/2) + n & \text{if } n > 2 \end{cases}$$

是 T .n/ D n lg n。

2.3-5
你也可以将插入排序视为递归算法。为了对 AŒ1 W n 进行排序，需要对子数组 AŒ1 W n 1 进行递归排序，然后将 AŒn 插入到已排序的子数组 AŒ1 W n 1 中。为这种递归版本的插入排序编写伪代码。给出其最坏情况运行时间的递归式。

2.3-6
回顾搜索问题（参见练习 2.1-4），观察如果被搜索的子数组已经排序，则搜索算法可以根据 v 检查子数组的中点，并从进一步的搜索中消除一半的子数组。

考虑。***binary search*** 算法重复此过程，每次将子数组剩余部分的大小减半。为二分查找编写迭代或递归的伪代码。论证二分查找的最坏情况运行时间为 ，.lg n/。

2.3-7
2.1 节中的 INSERTION-SORT 过程的第 537 行的 while 循环使用线性搜索来（向后）扫描已排序的子数组 AŒ1 W j 1 。如果插入排序使用二分搜索（参见练习 2.3-6）而不是线性搜索，结果会怎样？这是否会将插入排序的总体最坏情况运行时间改善为 ，.n lg n/ ' ？

2.3-8
描述一个算法，给定一个由 n 个整数组成的集合 S 和另一个整数 x ，判断 S 是否包含两个元素，它们的和恰好等于 x 。在最坏的情况下，你的算法应该花费 ，.n lg n/ 时间。

问题

2-1Insertion sort on small arrays in merge sort
尽管归并排序在最坏情况下的运行时间为 ，.n lg n/，而插入排序在最坏情况下的运行时间为 ，.n^2/，但插入排序中的常数因子在实践中可以使它在很多机器上对于小规模问题更快。因此，当子问题变得足够小时，在归并排序中使用插入排序对递归的叶子进行 ***coarsen*** 是有意义的。考虑对归并排序的一种修改，其中使用插入排序对长度为 k 的 n/k 个子列表进行排序，然后使用标准合并机制进行合并，其中 k 是一个待确定的值。

a.说明插入排序可以在最坏情况下对 n/k 个子列表（每个子列表长度为 k）进行排序，时间为 ，.nk/。***b.*** 说明如何在最坏情况下合并子列表，时间为 ，.n lg.n/k//。***c.*** 假设修改后的算法在最坏情况下运行时间为 ，.nk C n lg.n/k// ，那么，以 ，符号表示，使修改后的算法与标准合并排序的运行时间相同的 k 关于 n 的函数的最大值是多少？***d.*** 在实践中应如何选择 k？

2-2Correctness of bubblesort

冒泡排序是一种流行但效率低下的排序算法。它的工作原理是反复交换无序的相邻元素。程序 B UBBLESORT 对数组 AŒ1 W n 进行排序。

```
冒泡排序 .A; n/
1 对于 i D 1 到 n  1 2 对于 j D n 向下到 i C 1
3 如果 AŒj  < AŒj  1 4 用 AŒj  1 交换
AŒj  1
```

a.让 A′ 表示执行 BUBBLESORT .A; n/ 后的数组 A。要证明 BUBBLESORT 是正确的，您需要证明它终止并且

$$A'[1] \le A'[2] \le \cdots \le A'[n] . \tag{2.5}$$

为了证明 BUBBLESORT 确实可以排序，您还需要证明什么？

接下来的两部分证明不等式（2.5）。

b.精确陈述 234 行 for 循环的循环不变量，并证明该循环不变量成立。你的证明应使用本章中介绍的循环不变量证明的结构。

c.使用 (b) 部分中证明的循环不变量的终止条件，为 134 行中的 for 循环声明一个循环不变量，以便证明不等式 (2.5)。你的证明应该使用本章中介绍的循环不变量证明的结构。

d.BUBBLESORT 的最坏情况运行时间是多少？它与 INSERTION-SORT 的运行时间相比如何？

2-3Correctness of Horner's rule

给定多项式的系数 $a_0; a_1; a_2; : : : ; a_n$

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n , \end{aligned}$$

并且你想要针对给定的 x 值计算该多项式的值。 ***Horner's rule*** 表示根据以下括号计算多项式的值：

$$P(x) = a_0 + x\left(a_1 + x\left(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots\right)\right).$$

程序 HORNER 实现霍纳规则，给定数组 AŒ0 W n 中的系数 a 0 ；a 1 ；a 2 ；:::；a n 和 x 的值，计算 P .x/。

```
HORNER.A; n; x/
1 p D 0 2 for i D n downto
0 3 p D AŒi C x p 4 r
eturn p
```

a.用，符号表示，该程序的运行时间是多少？

b.编写伪代码来实现从头开始计算多项式每个项的朴素多项式求值算法。该算法的运行时间是多少？与 HORNER 相比如何？

c.考虑过程 HORNER 的以下循环不变量：

在第 233 行的 for 循环每次迭代开始时，

$$p = \sum_{k=0}^{n-(i+1)} A[k+i+1]\,x^k$$

将没有项的总和解释为等于 0。按照本章中介绍的循环不变式证明的结构，使用此循环不变式来证明在终止时，$p = \sum_{k=0}^{n} A[k]\,x^k$。

2-4Inversions

假设 AŒ1 W n 为 n 个不同数字的数组。如果 i < j 且 AŒi > AŒj ，则对 .i; j / 称为 A 的 ***inversion***。

a.列出数组 h2；3；8；6；1i 的五个反转。

b.包含集合 f1; 2; : : : ; ng 中的元素的哪个数组具有最多的反转？它有多少个？

c.插入排序的运行时间和输入数组中的反转次数有什么关系？证明你的答案。

d.给出一个算法，在最坏情况下，在 ，.n lg n/ 的时间内确定 n 个元素上任意排列中的反转次数。（*Hint:* 修改合并排序。）

章节注释

1968 年，Knuth 出版了三卷本的第一卷，总标题为 *The Art of Computer Programming* [259, 260, 261]。第一卷开启了现代计算机算法研究，重点是运行时间分析。对于这里介绍的许多主题，整个系列仍然是引人入胜且值得参考的。根据 Knuth 的说法，“algorithm” 一词源自 9 世纪波斯数学家的名字 “al-Khowˆarizmˆı”。

Aho、Hopcroft 和 Ullman [5] 提倡使用第 3 章介绍的符号（包括 ，符号）对算法进行渐近分析，以此作为比较相对性能的方法。他们还推广使用递归关系来描述递归算法的运行时间。

Knuth [261] 对许多排序算法进行了百科全书式的论述。他对排序算法的比较（第 381 页）包括精确的步数计算分析，就像我们在这里对插入排序进行的分析一样。Knuth 对插入排序的讨论涵盖了该算法的几种变体。其中最重要的是 D. L. Shell 提出的 Shell 排序，它对输入的周期性子数组使用插入排序来生成更快的排序算法。

Knuth 也描述了归并排序。他提到，1938 年发明了一种能够一次性合并两副打孔卡的机械整理器。计算机科学的先驱之一 J. von Neumann 显然在 1945 年在 EDVAC 计算机上编写了一个归并排序程序。

Gries [200] 描述了证明程序正确性的早期历史，他认为 P. Naur 是该领域的第一篇文章的作者。Gries 将循环不变量归功于 R. W. Floyd。Mitchell [329] 的教科书是有关如何证明程序正确性的很好的参考资料。

3 Characterizing Running Times

第 2 章定义了算法运行时间的增长顺序，它提供了一种简单的方法来描述算法的效率，也使我们能够将其与其他算法进行比较。一旦输入大小 n 变得足够大，归并排序（其最坏情况运行时间为 '.n lg n'）将击败插入排序（其最坏情况运行时间为 '.n ^{2} '）。虽然我们有时可以确定算法的确切运行时间，就像我们在第 2 章中对插入排序所做的那样，但额外的精度很少值得花费精力去计算它。对于足够大的输入，确切运行时间的乘法常数和低阶项受输入大小本身的影响。

当我们查看足够大的输入大小，使运行时间的增长顺序变得相关时，我们正在研究算法的***asymptotic***效率。也就是说，我们关心的是算法的运行时间如何随着输入*in the limit*的大小而增加，因为输入的大小无限增加。通常，除了非常小的输入外，渐近更高效的算法是最佳选择。

本章给出了几种简化算法渐近分析的标准方法。下一节非正式地介绍了三种最常用的“渐近符号”，其中我们已经在 ，符号中看到了一个例子。它还展示了一种使用这些渐近符号来推断插入排序的最坏情况运行时间的方法。然后我们更正式地研究渐近符号，并介绍本书中使用的几种符号约定。最后一节回顾了分析算法时经常出现的函数行为。

3.1 O-符号、-符号和 ，-符号

当我们在第 2 章分析插入排序的最坏情况运行时间时，我们从复杂的表达式开始

$$\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n - (c_2+c_4+c_5+c_8)\ .$$

然后我们舍弃低阶项 .c 1 C c 2 C c 4 C c 5/2 c 6/2 c 7/2 C c 8/n 和 c 2 C c 4 C c 5 C c 8，并且我们还忽略了 n 2 的系数 c 5/2 C c 6/2 C c 7/2。这样就只剩下因子 n 2，我们将其放入 ' 符号中，即 '.n 2/。我们使用这种风格来表征算法的运行时间：舍弃低阶项和首项的系数，并使用关注运行时间增长率的符号。

，符号并不是唯一这样的 “ 渐近符号。” 在本节中，我们还将看到其他形式的渐近符号。我们从直观地了解这些符号开始，重新审视插入排序，看看我们如何应用它们。在下一节中，我们将看到渐近符号的正式定义，以及使用它们的惯例。

在讨论具体内容之前，请记住，我们将看到的渐近符号是为了描述一般函数而设计的。我们最感兴趣的函数恰好表示算法的运行时间。但渐近符号可以应用于描述算法其他方面（例如，它们使用的空间量）的函数，甚至可以应用于与算法毫无关系的函数。

O 符号

O 符号表示函数渐近行为上的 *upper bound*。换句话说，它表示函数基于最高阶项以高于某个速率增长 *no faster* 。例如，考虑函数 7n 3 C 100n 2 20n C 6。其最高阶项是 7n 3，因此我们称该函数的增长率为 n 3。由于该函数的增长速度不快于 n 3，因此我们可以将其写为 O.n 3/。您可能会感到惊讶，我们还可以将其写为函数 7n 3 C 100n 2 20n C 6 为 O.n 4/。为什么？因为该函数的增长速度比 n 4 慢，所以我们说它增长速度不快是正确的。您可能已经猜到了，这个函数也是 O.n 5/、O.n 6/ 等等。更一般地，对于任何常数 c 3，它都是 O.n c/。

符号

符号根据函数的渐近行为来表征 *lower bound*。换句话说，它表示函数以一定的速率增长 *at least as fast* ，基于 4O 符号 4中的最高阶项。因为函数 7n ³ C 100n ² 20n C 6 中的最高阶项至少与 n ³ 一样快地增长，所以该函数为 .n ³/。该函数也是 .n ²/ 和 .n/。更一般地，对于任何常数 c = 3，它都是 .n *c* /。

，-符号

，符号表示函数的渐近行为上的 *tight bound*。它表示函数以一定的速率增长 *precisely*，同样基于最高阶项。换句话说， ，符号表示函数的增长率在上方一个常数因子内和下方一个常数因子内。这两个常数因子不必相等。

如果你能证明，对于某个函数 f .n/，某个函数既是 O.f .n// 又是 .f .n//，那么你就证明了该函数是 ，.f .n//。（下一节将以定理的形式陈述这一事实。）例如，由于函数 7n ³C100n ² 20nC6 既是 O.n ³/ 又是 .n ³/，因此它也是 ，.n ³/。

例子：插入排序

让我们重新回顾插入排序，并看看如何使用渐近符号来描述它的“ .n² ”最坏情况运行时间，而不需要像第 2 章那样求和。这里再次给出插入排序过程：

插入排序 .A; n/

1 for i D 2 to n 2 *key* D AŒi 3 // 将 AŒi 插入到已排序的子数组 AŒ1 W i 1 中。 4 j D i 1 5 while j > 0 and AŒj > *key* 6 AŒj C 1 D AŒj 7 j D j 1 8 AŒj C 1 D *key*

我们可以观察到伪代码是如何运行的？该过程有嵌套循环。外循环是一个 for 循环，无论排序的值是什么，它都会运行 n 1 次。内循环是一个 while 循环，但它进行的迭代次数取决于排序的值。循环变量 j 从 i 1 开始

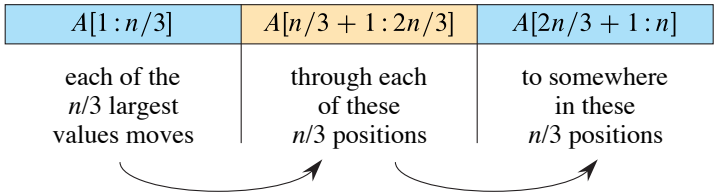


图 3.1 插入排序的 .n 2 / 下限。如果前 n/3 个位置包含 n/3 个最大值，则这些值中的每一个都必须逐个移动中间的 n/3 个位置，每次一个位置，最终到达最后 n/3 个位置的某个位置。由于 n/3 个值中的每一个至少会移动 n/3 个位置中的每一个，因此在这种情况下所花费的时间至少与 .n/3/.n/3/ D n 2 /9 或 .n 2 / 成比例。

每次迭代都会减少 1，直到达到 0 或 AŒj *key*。对于给定的 i 值，while 循环可能会迭代 0 次、i 1 次或介于两者之间的任何次数。while 循环体（第 637 行）每次迭代都需要常数时间。

这些观察足以推断出任何 INSERTION-SORT 情况的运行时间为 O.n 2 /，这为我们提供了涵盖所有输入的概括语句。运行时间由内循环决定。因为外循环的 n 1 次迭代中的每一次都导致内循环最多迭代 i 1 次，并且 i 最多为 n ，所以内循环的总迭代次数最多为 -n 1 / -n 1 /，小于 n 2 。由于内循环的每次迭代都需要常数时间，所以内循环所花费的总时间最多为常数乘以 n 2 ，或 O.n 2 /。

稍微发挥一下创造力，我们还可以看到 I NSERTION-SORT 的最坏情况运行时间为 .n 2 /。说一个算法的最坏情况运行时间为 .n 2 /，我们的意思是，对于每个大于某个阈值的输入大小 n，对于某个正常数 c，至少有一个大小为 n 的输入，该算法至少需要 cn 2 时间。这并不一定意味着该算法对所有输入都至少需要 cn 2 时间。

现在让我们看看为什么 INSERTION-SORT 的最坏情况运行时间为 .n 2 / 。如果一个值要最终位于其起始位置的右侧，它必须在第 6 行被移动。事实上，如果一个值要最终位于其起始位置的右侧 k 个位置，第 6 行必须执行 k 次。如图 3.1 所示，我们假设 n 是 3 的倍数，这样我们就可以将数组 A 分成 n/3 个位置的组。假设在 INSERTION-SORT 的输入中，最大的 n/3 个值占据了数组 AŒ1 W n/3 的前 n/3 个位置。（它们在前 n/3 个位置中的相对顺序无关紧要。）一旦对数组进行排序，这 n/3 个值中的每一个都会出现在最后 n/3 个位置 AŒ2n/3 C 1 W n 的某个位置。要做到这一点，这 n/3 个值中的每一个都必须经过中间 n/3 个位置 AŒn/3 C 1 W 2n/3 。这 n/3 个值中的每一个都会经过这些中间

n/3 每次定位一个位置，至少执行 n/3 次第 6 行。因为至少 n/3 个值必须经过至少 n/3 个位置，所以在最坏情况下 INSERTION-SORT 所花费的时间至少与 .n/3/.n/3/ D n ²/9 成比例，即 .n ²/。

因为我们已经证明了 I NSERTION-SORT 在所有情况下的运行时间为 O.n ²/ 并且有一个输入使其需要 .n ²/ 时间，所以我们可以得出结论，I NSERT ION-SORT 的最坏情况运行时间为 .n ²/。上限和下限的常数因子可能不同并不重要。重要的是，我们已经将最坏情况的运行时间限定在常数因子之内（不考虑低阶项）。这个论点并没有表明 I NSERTION-SORT 在 *all* 情况下的运行时间为 .n ²/ 时间。事实上，我们在第 2 章中看到，最佳情况的运行时间为 .n/。

练习

3.1-1
修改插入排序的下限参数来处理不一定是 3 的倍数的输入大小。

3.1-2
使用与插入排序类似的推理，分析练习 2.2-2 中的选择排序算法的运行时间。

3.1-3
假设 α 是 $0 < \alpha < 1$ 范围内的一个分数。说明如何推广插入排序的下限参数，以考虑输入，其中 αn 个最大值从前 αn 个位置开始。您需要对 α 施加什么额外限制？α 的什么值使 αn 个最大值必须通过中间 .1 2α/n 数组位置的次数最大？

3.2 渐近符号：形式定义

非正式地了解了渐近符号后，让我们来正式了解一下。我们用来描述算法渐近运行时间的符号是根据函数定义的，这些函数的定义域通常是自然数集 N 或实数集 R。这些符号对于描述运行时间函数 T.n/ 很方便。本节定义了基本的渐近符号，并介绍了一些常见的“正确”符号滥用。

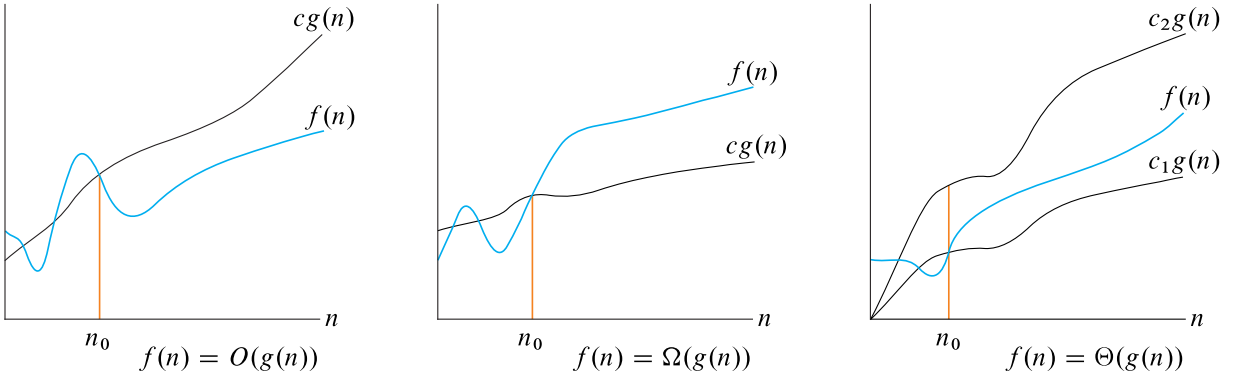


图 3.2 O、 和 ，符号的图形示例。在每一部分中，所示的 n 0 的值都是可能的最小值，但是任何更大的值也可以。（a）O 符号给出函数在常数因子内的上限。如果存在正常数 n 0 和 c，使得在 n 0 右侧，f .n/ 的值始终位于 cg.n/ 上或之下，则我们记为 f .n/ D O.g.n//。（b）符号给出函数在常数因子内的下限。如果存在正常数 n 0 和 c，使得在 n 0 右侧，f .n/ 的值始终位于 cg.n/ 上或之上，则我们记为 f .n/ D .g.n//。（c），符号将函数限制在常数因子内。如果存在正常数 n 0、c 1 和 c 2，使得在 n 0 的右边，f .n/ 的值始终位于 c 1 g.n/ 和 c 2 g.n/ 之间（含两个端点），则记为 f .n/ D ，.g.n//。

O 符号

正如我们在第 3.1 节中看到的，O 符号描述了 ***asymptotic upper bound***。我们使用 O 符号给出函数的上限，在常数因子内。

这是 O 符号的正式定义。对于给定函数 g.n/，我们用 O.g.n//（发音为 “big -oh of g of n” 或有时只是 “oh of g of n”）表示 *set of functions*

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$[1]

如果存在一个正常数 c，使得当 n 足够大时，f.n/ = cg.n/，则函数 f.n/ 属于集合 O.g.n//。图 3.2(a) 显示了 O 符号背后的直觉。对于 n 0 处及右侧的所有 n 值，函数 f.n/ 的值都在 cg.n/ 上或 cg.n/ 之下。

O.g.n// 的定义要求集合 O.g.n// 中的每个函数 f.n/ 都是 ***asymptotically nonnegative***：只要 n 足够大，f.n/ 就必须是非负的。（***asymptotically positive*** 函数是对所有

[1] Within set notation, a colon means “such that.”

足够大的 n。）因此，函数 g.n/ 本身必须是渐近非负的，否则集合 O.g.n// 为空。因此，我们假设 O 符号中使用的每个函数都是渐近非负的。此假设也适用于本章中定义的其他渐近符号。

你可能会惊讶于我们用集合来定义 O 符号。事实上，你可能会认为我们会写成 “f .n/ 2 O.g.n//” 来表示 f .n/ 属于集合 O.g.n//。相反，我们通常写成 “f .n/ D O.g.n//” 并说 “f .n/ 是 g.n/” 的大 O，以表达相同的概念。虽然乍一看以这种方式滥用平等似乎令人困惑，但我们将在本节后面看到这样做有其优势。

让我们探索一个例子，说明如何使用 O 符号的形式定义来证明我们丢弃低阶项并忽略最高阶项的常数系数的做法是合理的。我们将证明 $4n^2 C100n C500 D O.n^2/$，即使低阶项的系数比首项大得多。我们需要找到正常数 c 和 n_0，使得对于所有 $n \approx n_0$，$4n^2 C 100n C 500 \#cn^2$。两边除以 n^2 可得出 $4 C 100/n C 500/n^2 \#c$。对于许多 c 和 n_0 的选择，这个不等式都是成立的。例如，如果我们选择 $n_0 D 1$，则该不等式对 c D 604 成立。如果我们选择 n_0 D 10，则 c D 19 有效，而选择 n_0 D 100 则允许我们使用 c D 5:05。

我们还可以使用 O 符号的形式定义来表明函数 $n^3 \ 100n^2$ 不属于集合 $O.n^2/$，即使 n^2 的系数是一个很大的负数。如果有 $n^3 \ 100n^2 D O.n^2/$，那么就会有正常数 c 和 n_0，使得对于所有 $n \ n_0$，$n^3 \ 100n^2 = cn^2$。同样，我们将两边除以 n^2，得出 $n \ 100 = c$。不管我们为常数 c 选择什么值，这个不等式对任何 n > c C 100 的值都不成立。

符号

正如 O 符号为函数提供渐近 *upper* 界限一样， 符号提供 ***asymptotic lower bound***。对于给定函数 g.n/，我们用 .g.n//（读作 “big-omega of g of n” 或有时读作 “omega of g of n”）表示函数集

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$

图 3.2(b) 显示了 符号背后的直觉。对于 n_0 处或右侧的所有 n 值，f.n/ 的值都在 cg.n/ 上或上方。

我们已经证明了 $4n^2 C 100n C 500 D O.n^2/$。现在让我们证明 $4n^2 C 100n C 500 D \ .n^2/$。我们需要找到正常数 c 和 n_0，使得 $4n^2 C 100n C 500$ 及其 cn^2 对所有 n 及其 n_0。和前面一样，我们将两边除以 n^2，

给出 4 C 100/n C 500/n ² c 。当 n ₀ 为任意正整数且 c D 4 时，此不等式成立。

如果我们从 4n ² 项中减去低阶项而不是将它们相加，结果会怎样？如果 n ² 项的系数很小，结果会怎样？函数仍为 .n ²/。例如，假设 n ²/100 100n 500 D .n ²/。除以 n ² 可得出 1/100 100/n 500/n ² c 。我们可以为 n ₀ 选取至少为 10,005 的任意值，并为 c 找到一个正值。例如，当 n ₀ D 10,005 时，我们可以选取 c D 2.49 10 ⁻⁹。是的，对于 c 来说，这是一个很小的值，但它是正值。如果我们为 n ₀ 选择更大的值，我们也可以增加 c 。例如，如果 n ₀ D 100,000，那么我们可以选择 c D 0:0089。n ₀ 的值越高，我们可以选择的 c 越接近系数 1/100。

，-符号

我们使用 ，符号表示 ***asymptotically tight bounds***。对于给定函数 g.n/，我们用 ，.g.n// (“n 的 g 的 θ ”) 表示函数集

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$

图 3.2(c) 显示了 ，符号背后的直觉。对于 n ₀ 及其右侧的所有 n 值，f.n/ 的值位于 c ₁ g.n/ 或以上，以及 c ₂ g.n/ 或以下。换句话说，对于所有 n ≥ n ₀，函数 f.n/ 等于 g.n/，且在常数因子范围内。

O-、 - 和 ，- 符号的定义导致了下面的定理，其证明我们留在练习 3.2-4 中。

Theorem3.1
对于任意两个函数 f .n/ 和 g.n/，当且仅当 f .n/ D O.g.n// 和 f .n/ D .g.n// 时，才有 f .n/ D ，.g.n//。 ■

我们通常应用定理 3.1 来从渐近上限和下限证明渐近紧限。

渐近符号和运行时间

当您使用渐近符号来描述算法的运行时间时，请确保使用的渐近符号尽可能精确，但不要夸大其适用的运行时间。以下是一些正确和不正确使用渐近符号来描述运行时间的示例。

让我们从插入排序开始。我们可以正确地说，插入排序的最坏情况运行时间为 O.n ²/， .n ²/, and4，这归因于定理 3.14， .n ²/。虽然

如果描述最坏情况运行时间的三种方法都是正确的，则 ，.n ² / 界限是最精确的，因此也是最优选的。我们还可以正确地说，插入排序的最佳运行时间为 O.n/、 .n/ 和 ，.n/，同样， ，.n/ 是最精确的，因此也是最优选的。

这是我们 *cannot* 正确的表述：插入排序的运行时间为 ‘ .n ² /。这是一种夸大其词，因为通过从语句中省略 “最坏情况”，我们只剩下一个涵盖所有情况的笼统语句。这里的错误在于插入排序并非在所有情况下都运行于 ‘ .n ² / 的时间内，因为正如我们所见，它在最好情况下运行于 ‘ .n/ 的时间内。但是，我们可以正确地说插入排序的运行时间为 O.n ² /，因为在所有情况下，它的运行时间增长速度都不会比 n ² 快。当我们说 O.n ² / 而不是 ‘ .n ² / 时，运行时间增长速度比 n ² 慢的情况也没有问题。同样，我们不能正确地说插入排序的运行时间为 ‘ .n/，但我们可以说它的运行时间为 ‘ .n/。

那么归并排序怎么样？由于归并排序在所有情况下的运行时间为 ，.n lg n/ ，我们可以说它的运行时间为 ，.n lg n/ ，而无需指定最坏情况、最佳情况或任何其他情况。

人们有时会将 O 符号与 ，符号混淆，错误地使用 O 符号来表示渐近紧边界。他们会说，“O.n lg n/ 时间算法比 O.n ² / 时间算法运行得更快。” 可能是，也可能不是。由于 O 符号仅表示渐近上限，因此所谓的 O.n ² / 时间算法实际上可能运行 ，.n/ 时间。您应该小心选择适当的渐近符号。如果要表示渐近紧边界，请使用 ，符号。

我们通常使用渐近符号来提供最简单、最精确的界限。例如，如果一个算法在所有情况下的运行时间都是 3n ² C 20n，我们使用渐近符号来表示它的运行时间为 ，.n ² /。严格地说，我们也可以正确地写出运行时间为 O.n ³ / 或 ，.3n ² C 20n/。但是，在这种情况下，这两种表达方式都没有 ，.n ² / 有用：如果运行时间为 3n ² C 20n，则 O.n ³ / 不如 ，.n ² / 精确，而 ，.3n ² C 20n / 引入的复杂性会掩盖增长顺序。通过写出最简单、最精确的界限，例如 ，.n ² /，我们可以对不同的算法进行分类和比较。在整本书中，你会看到几乎总是基于多项式和对数的渐近运行时间：诸如 n、n lg ² n、n ² lg n 或 n ¹/² 之类的函数。你还会看到一些其他函数，例如指数、lg lg n 和 lg * n（参见第 3.3 节）。通常比较这些函数的增长率相当容易。问题 3-3 可以为你提供很好的练习。

方程和不等式中的渐近符号

尽管我们正式用集合来定义渐近符号，但在公式中使用等号 (D) 代替集合成员符号 (2)。例如，我们写成 4n ² C 100n C 500 D O.n ²/。我们也可以写成 2 n ² C 3n C 1 D 2n ² C ，.n/。我们如何解释这样的公式？

当渐近符号单独存在（即不在较大的公式中）于等式（或不等式）的右边时，如 4n ² C 100n C 500 D O.n ²/，等号表示集合成员：4n ² C 100n C 500 2 O.n ²/。但一般来说，当渐近符号出现在公式中时，我们将其解释为代表某个我们不想命名的匿名函数。例如，公式 2n ²C 3n C 1 D 2n ²C ，.n/ 表示 2n ²C 3n C 1 D 2n ²C f .n/，其中 f .n/ 2 ，.n/。在这种情况下，我们让 f .n/ D 3n C 1，它确实属于 ，.n/。

以这种方式使用渐近符号可以帮助消除方程中不必要的细节和混乱。例如，在第 2 章中，我们将归并排序的最坏情况运行时间表示为递归

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

如果我们只对 T .n/ 的渐近行为感兴趣，那么准确指定所有低阶项是没有意义的，因为它们都被理解为包含在用项 ‘ .n/ 表示的匿名函数中。

表达式中匿名函数的数量被理解为等于渐近符号出现的次数。例如，在表达式中

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

只有一个匿名函数（i 的函数）。因此，此表达式 *not* 与 O.1/ C O.2/ C C O .n/ 相同，后者实际上没有明确的解释。

在某些情况下，渐近符号出现在等式的左边，例如

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

使用以下规则解释此类方程：*No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid*。因此，我们的例子意味着，对于 *any* 函数 f.n/ 2 ，.n/，存在 *some* 函数 g.n/ 2 ，.n ²/，使得对于所有 n，2n ²Cf.n/ D g.n/。换句话说，等式的右侧提供的细节级别比左侧更粗糙。

我们可以将多个这样的关系串联起来，例如

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) . \end{aligned}$$

根据上述规则，分别解释每个方程。第一个方程表示存在 *some* 函数 f .n/ 2 ，.n/ 使得对于所有 n，2n ²C3n C1 D 2n ² C f .n/。第二个方程表示，对于 *any* 函数 g.n/ 2 ，.n/（例如刚刚提到的 f .n/），存在 *some* 函数 h.n/ 2 ，.n ²/ 使得对于所有 n，2n ² C g.n/ D h.n/。此解释意味着 2n ² C 3n C 1 D ，.n ²/，这正是方程链直观地表达的意思。

渐近符号的正确滥用

除了滥用等式来表示集合成员（我们现在看到它具有精确的数学解释）之外，当必须从上下文推断出趋向于 1 的变量时，还会滥用渐近符号。例如，当我们说 O.g.n// 时，我们可以假设我们感兴趣的是 g.n/ 随着 n 的增长而增长，如果我们说 O.g.m//，我们谈论的是 g.m/ 随着 m 的增长而增长。表达式中的自由变量表示哪个变量将趋向于 1。

最常见的情况是，当渐近符号中的函数为常数时，需要上下文知识来确定哪个变量趋向于 1，如表达式 O.1/ 中所示。我们无法从表达式中推断出哪个变量趋向于 1，因为那里没有变量。上下文必须消除歧义。例如，如果使用渐近符号的方程为 f .n/ D O.1/，则显然我们感兴趣的变量是 n。但是，从上下文知道感兴趣的变量是 n，我们便可以使用 O 符号的形式定义完全理解该表达式：表达式 f .n/ D O.1/ 表示函数 f .n/ 在 n 趋向于 1 时由常数从上方限定。从技术上讲，如果我们在渐近符号中明确指出趋向于 1 的变量，则歧义可能会减少，但这会使符号变得混乱。相反，我们只是确保上下文清楚地表明哪个变量（或哪个变量）趋向于 1。

当渐近符号中的函数以正常数为界时，如 T .n/ D O.1/，我们常常以另一种方式滥用渐近符号，尤其是在陈述递归时。我们可以写出类似这样的句子：对于 n < 3，T .n/ D O.1/。根据 O 符号的形式定义，这个陈述毫无意义，因为定义只说 T .n/ 以正常数 c 为界，对于 $n \geq n_0$，其中 $n_0 >$ 为 0。对于 $n < n_0$，T .n/ 的值不必如此有界。因此，在 n < 3 的示例中，在 T .n/ D O.1/ 中，当 n < 3 时，我们无法推断出对 T .n/ 的任何约束，因为可能是 $n_0 > 3$。

当我们说 T .n/ D O.1/ for n < 3 时，通常的意思是存在一个正常数 c 使得 T .n/ c for n < 3。此约定可省去

我们免去了命名边界常数的麻烦，允许它在我们关注分析中更重要的变量时保持匿名。其他渐近符号也存在类似的滥用。例如，对于 n < 3，T .n/ D ，. 1/ 表示当 n < 3 时，T .n/ 的上下边界均为正常数。

偶尔，描述算法运行时间的函数可能无法针对某些输入大小进行定义，例如，当算法假设输入大小是 2 的精确幂时。我们仍然使用渐近符号来描述运行时间的增长，同时要理解任何约束都只在函数定义时才适用。例如，假设 f.n/ 仅在自然数或非负实数的子集上定义。那么 f.n/ D O.g.n// 意味着 O 符号定义中的界限 0 → T.n/ → cg.n/ 对 f.n/ 定义域上的所有 n ∈ n $_0$ 都成立，也就是说，其中 f.n/ 是定义的。这种滥用很少被指出，因为从上下文中通常可以清楚地看出它的意思。

在数学中，只要我们不误用符号，滥用符号是可以的，而且往往是可取的。如果我们准确理解滥用的含义，并且不得出错误的结论，它可以简化我们的数学语言，有助于我们更高层次的理解，并帮助我们专注于真正重要的事情。

o 符号

O 符号提供的渐近上界可能是也可能不是渐近紧的。界限 2n ^{2} D O.n ^{2} / 是渐近紧的，但界限 2n D O.n ^{2} / 不是。我们使用 o 符号来表示不是渐近紧的上界。我们正式定义 o.g.n// (“ g of n” 的小 oh) 为集合

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} .$$

例如，2n D o.n ^{2} /，但 2n ^{2} ¤ o.n ^{2} /。

O 符号和 o 符号的定义类似。主要区别在于，在 f .n/ D O.g.n// 中，边界 0 = f .n/ = cg.n/ 适用于 *some* 常数 c > 0，但在 f .n/ D o.g.n// 中，边界 0 = f .n/ < cg.n/ 适用于 *all* 常数 c > 0。直观地看，在 o 符号中，随着 n 变大，函数 f .n/ 相对于 g.n/ 变得不重要：

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

有些作者使用这个极限作为 o 符号的定义，但本书中的定义还限制匿名函数为渐近非负的。

！-符号

类比起来，!-符号与 -符号的关系相当于 o-符号与 O-符号的关系。我们使用 !-符号来表示非渐近紧的下界。定义它的一种方法是

f .n/ 2 !.g.n// 当且仅当 g.n/ 2 o.f .n// ：

然而，正式地，我们将 !.g.n// (“n” 的 g 的小欧米茄) 定义为集合

!.g.n// D ff .n/ W 对于任何正常数 c> 0，存在一个常数 $n_0 > 0$ 使得 0
cg.n/< f .n/ 对所有 n 及其 n_0 g：

o 符号的定义是 f .n/ < cg.n/ ，而 ! 符号的定义则相反：cg.n/ < f .n/ 。对于 ! 符号的例子，我们有 $n^2/2$ D !.n/ ，但 $n^2/2$ ¤ !.n^2/ 。关系 f .n/ D !.g.n// 意味着

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

如果极限存在。也就是说，当 n 变大时，f.n/ 相对于 g.n/ 变得任意大。

比较函数

实数的许多关系性质也适用于渐近比较。以下假设 f.n/ 和 g.n/ 渐近正。

及物性：

$$\begin{aligned}
&f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)) ,\\
&f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)) ,\\
&f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)) ,\\
&f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)) ,\\
&f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)) .
\end{aligned}$$

实证性：

f .n/ D , .f .n// ; f .n
/ D O.f .n// ; f .n/ D
.f .n// :

对称：

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

转置对称性：

f .n/ D O.g.n// 当且仅当 g.n/ D .f .n// ; f .n/ D o.g.n
// 当且仅当 g.n/ D !.f .n// :

由于这些性质对于渐近符号成立，我们可以将两个函数 f 和 g 的渐近比较与两个实数 a 和 b 的比较进行类比：

f .n/ D O.g.n// 类似于 a b ; f .n/ D .g.n// 类似于 a b ; f .n/ D ，.g .n// 类似于 a D b ; f .n/ D o.g.n// 类似于 a < b ; f .n/ D !.g.n// 类似于 a > b :

如果 f .n/ D o.g.n//，则我们说 f .n/ 比 g.n/ 更具有 ***asymptotically smaller*** 性质；如果 f .n/ D !.g.n//，则我们说 f .n/ 比 g.n/ 更具有 ***asymptotically larger*** 性质。然而，实数的一个性质并不适用于渐近符号：

三分法：对于任意两个实数 a 和 b，以下情况中恰好有一个成立：a < b，a D b，或 a > b。

虽然任何两个实数都可以比较，但并非所有函数都是渐近可比的。也就是说，对于两个函数 f .n/ 和 g.n/，可能 f .n/ D O.g.n// 和 f .n/ D .g.n// 都不成立。例如，我们不能使用渐近符号比较函数 n 和 $n^{1+\sin n}$，因为 $n^{1+\sin n}$ 中的指数值在 0 和 2 之间振荡，取其间的所有值。

练习

3.2-1

令 f .n/ 和 g.n/ 为渐近非负函数。利用 ′ 符号的基本定义，证明 max ff .n/；g.n/g D ′ .f .n/ C g.n//。

3.2-2

解释为什么语句“算法 A 的运行时间至少为 $O.n^2/$，” 没有意义。

3.2-3 2^{n+1} D $O.2^n/$ 是否正确？2^{2n} D $O.2^n/$ 是否正确？

3.2-4

证明定理3.1。

3.2-5
证明：一种算法的运行时间为，.g.n//当且仅当其最坏情况运行时间为 O.g.n// ，最佳情况运行时间为 .g.n//。

3.2-6证明o.g.n// \ !.g.n//是空集。

3.2-7
我们可以将符号扩展到两个参数 n 和 m 的情况，它们可以以不同的速率独立地变为 1。对于给定函数 g.n; m/，我们用 O.g.n; m// 表示函数集

O.g.n; m// D ff .n; m/ W 存在正常数 c 、 n_0 和 m_0 ，使得对于所有 n 及其 n_0 或 m 及其 n_0g ，0 = f .n; m/ ; cg.n; m/ ：

请给出 .g.n; m// 和 ，.g.n; m// 的相应定义。

3.3 标准符号和常用函数

本节回顾一些标准数学函数和符号，探讨它们之间的关系，并说明渐近符号的用法。

单调性

如果 m # n 意味着 f .m/ # f .n/，则函数 f .n/ 为 ***monotonically increasing***。类似地，如果 m # n 意味着 f .m/ 和 f .n/，则函数 f .n/ 为 ***monotonically decreasing***。如果 m < n 意味着 f .m/ < f .n/，则函数 f .n/ 为 ***strictly increasing*** ；如果 m < n 意味着 f .m/ > f .n/，则函数 f .n/ 为 ***strictly de- creasing***。

地板和天花板

对于任何实数 x ，我们用 bx c 表示小于或等于 x 的最大整数（读作 “x 的 üo or ”），用 dx e 表示大于或等于 x 的最小整数（读作 “x 的 ceiling ”）。üoor 函数是单调递增的，ceiling 函数也是如此。

地板和地板 满足以下性质。对于任何整数 rn，我们有

$$\lfloor n \rfloor = n = \lceil n \rceil \ . \tag{3.1}$$

对于所有实数 x ，我们有

$$x - 1 \;<\; \lfloor x \rfloor \;\le\; x \;\le\; \lceil x \rceil \;<\; x + 1\,. \tag{3.2}$$

我们还有

$$-\lfloor x \rfloor = \lceil -x \rceil\,, \tag{3.3}$$

或者等价地,

$$-\lceil x \rceil = \lfloor -x \rfloor\,. \tag{3.4}$$

对于任何实数 x 0 和整数 a; b > 0，我们有

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil\,, \tag{3.5}$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor\,, \tag{3.6}$$

$$\left\lceil \frac{a}{b} \right\rceil \le \frac{a + (b-1)}{b}\,, \tag{3.7}$$

$$\left\lfloor \frac{a}{b} \right\rfloor \ge \frac{a - (b-1)}{b}\,. \tag{3.8}$$

对于任何整数 n 和实数 x，我们有

$$\lfloor n + x \rfloor = n + \lfloor x \rfloor\,, \tag{3.9}$$

$$\lceil n + x \rceil = n + \lceil x \rceil\,. \tag{3.10}$$

模数运算

对于任何整数 a 和任何正整数 n，a mod n 的值是商 a/n 的 ***remainder***（或 ***residue***）：

$$a \bmod n = a - n\,\lfloor a/n \rfloor\,. \tag{3.11}$$

因此

$$0 \le a \bmod n < n\,, \tag{3.12}$$

即使 a 为负数。

给定一个整数除以另一个整数后的余数的明确概念，提供特殊符号来表示余数相等是很方便的。如果 .a mod n/ D .b mod n/，我们写为 a D b .mod n/，并称 a 等于 b 的 ***equivalent***，模 n。换句话说，如果 a 和 b 除以 n 后余数相同，则 a D b .mod n/。等价地，当且仅当 n 是 b 的除数时，a D b .mod n/。如果 a 不等同于 b 的模 n，我们写为 a ¤ b .mod n/。

多项式

给定一个非负整数 d，***polynomial in*** n ***of degree*** d 是形式为 p.n/ 的函数

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

其中常数 a 0; a 1; : : : ; a d 为多项式的 ***coefficients*** 和 a d ¤ 0。多项式渐近正当且仅当 a d > ≥ 0。对于 d 次渐近正多项式 p.n/，有 p.n/ D ，.n d /。对于任何实数常数 a ¬ 0，函数 n a 都是单调递增的，对于任何实数常数 a ‹ 0，函数 n a 都是单调递减的。如果对于某个常数 k，f.n/ D O.n k /，则我们说函数 f.n/ 为 ***polynomially bounded***。

指数

对于所有实数 a > 0、m 和 n，我们有以下恒等式：

$$\begin{aligned} a^0 &= 1 \ , \\ a^1 &= a \ , \\ a^{-1} &= 1/a, \\ (a^m)^n &= a^{mn} \ , \\ (a^m)^n &= (a^n)^m \ , \\ a^m a^n &= a^{m+n} \ . \end{aligned}$$

对于所有的 n 和 a ≥ 1，函数 a^n 关于 n 单调递增。方便时，我们假设 $0^0 \geq 1$。

我们可以通过以下事实将多项式和指数的增长率联系起来。对于所有实数常数 a > 1 和 b，我们有

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

由此我们可以得出以下结论

$$n^b = o(a^n) \ . \tag{3.13}$$

因此，任何底数严格大于 1 的指数函数增长速度都比任何多项式函数快。

使用 e 表示自然对数函数的底数 2:71828:::，对于所有实数 x,

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

其中“Š”表示本节后面定义的阶乘函数。对于所有实数 x，我们有不等式

$$1 + x \le e^x , \tag{3.14}$$

其中，只有当 x D 0 时，等式才成立。当 jx j 1 时，我们有近似值

$$1 + x \le e^x \le 1 + x + x^2 . \tag{3.15}$$

当 x ! 0 时，e ^{x} 通过 1Cx 的近似值非常好：

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

(在这个方程中，渐近符号用来描述极限行为，即 x ! 0，而不是 x ! 1。) 对于所有 x，我们有，

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x . \tag{3.16}$$

对数

我们使用以下符号：

lg n D log $_2$ n（二进制对数），ln n D log $_e$ n（自然对数），lg ^{k} n D .lg n/ ^{k} (指数)，lg lg n D lg.lg n/（组合）。

我们采用以下符号约定：如果没有括号，则为 *a logarithm function applies only to the next term in the formula*，因此 lg n C 1 表示 .lg n/ C 1 而不是 lg.n C 1/。对于任何常数 b > 1，如果 n = 0，则函数 log $_b$ n 为未定函数；如果 n > 0，则函数 log $_b$ n 为严格递增函数；如果 0 < n < 1，则函数 log $_b$ n 为负函数；如果 n > 1，则函数 log $_b$ n 为正函数；如果 n D 1，则函数 log 0。对于所有实数 a > 0、b > 0、c > 0 和 n，我们有

$$a = b^{\log_b a} \tag{3.17}$$

$$\log_c(ab) = \log_c a + \log_c b , \tag{3.18}$$

$$\log_b a^n = n \log_b a ,$$

$$\log_b a = \frac{\log_c a}{\log_c b} , \tag{3.19}$$

$$\log_b(1/a) = -\log_b a , \tag{3.20}$$

$$\log_b a = \frac{1}{\log_a b} ,$$

$$a^{\log_b c} = c^{\log_b a} , \tag{3.21}$$

其中，在上面的每个等式中，对数底数都不为 1。

根据公式 (3.19)，将对数的底数从一个常数更改为另一个常数只会使对数的值改变一个常数因子。因此，当我们不关心常数因子时，我们经常使用符号 “lg n”，例如在 O 符号中。计算机科学家发现 2 是对数最自然的底数，因为许多算法和数据结构都涉及将问题分为两部分。

当 jx j < 1 时，ln.1 C x/ 有一个简单的级数展开式：

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots . \tag{3.22}$$

对于 x > ，我们还有以下不等式 1：

$$\frac{x}{1+x} \le \ln(1+x) \le x , \tag{3.23}$$

其中等式仅对 x D 0 成立。

如果函数 f .n/ D O.lg ^{k} n/ 中存在某个常数 k，则称该函数为 ***polylogarithmically bounded***。我们可以在方程 (3.13) 中用 lg n 代替 n，用 2 ^{a} 代替 a，从而关联多项式和多对数的增长。对于所有实数常数 a > 0 和 b，我们有

$$\lg^b n = o(n^a) . \tag{3.24}$$

因此，任何正多项式函数的增长速度都比任何多对数函数快。

阶乘

对于整数 n ≥ 0，符号 nŠ（读作 “n 阶乘”）定义为

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

因此，nŠ D123n。

阶乘函数的弱上界为 nŠ n ^{n}，因为阶乘积中的 n 项中的每一项最多为 n。***Stirling's approximation***，

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) , \tag{3.25}$$

其中 e 是自然对数的底数，它给出了更严格的上界和下界。练习 3.3-4 要求你证明以下三个事实

$$n! = o(n^n) , \tag{3.26}$$

$$n! = \omega(2^n) , \tag{3.27}$$

$$\lg(n!) = \Theta(n \lg n) , \tag{3.28}$$

其中，斯特林近似有助于证明公式 (3.28)。以下公式对所有 n ≥ 1 也成立：

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{3.29}$$

在哪里

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n} .$$

功能迭代

我们使用符号 f (i) .n/ 来表示函数 f .n/ 迭代应用 i 次到初始值 n。正式地，让 f .n/ 成为实数函数。对于非负整数 i ，我们递归地定义

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 , \\ f(f^{(i-1)}(n)) & \text{if } i > 0 . \end{cases} \tag{3.30}$$

例如，如果 f .n/ D 2n，则 f (i) .n/ D 2 i n。

重对数函数

我们使用符号 lg * n（读作 “log star of n”）来表示迭代对数，定义如下。令 l g (i) n 如上定义，其中 f .n/D lg n。由于非正数的对数是未定义的，因此仅当 lg (i−1) n > 0 时，lg (i) n 才定义。务必区分 lg (i) n（从参数 n 开始连续应用 i 次的对数函数）和 lg i n（n 的 i 次方对数）。然后我们将迭代对数函数定义为

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

重对数是一个 *very* 缓慢增长的函数：

lg * 2 D 1；
lg * 4 D 2；lg * 16
D 3；lg * 65536
D 4；lg * .2 65536 /
D 5：

由于可观测宇宙中的原子数量估计约为 10^{80}，远小于 2^{65536} D$10^{65536/\lg 10}$ 10 ^{19,728}，我们很少遇到输入大小 n 使得 lg* n>5。

斐波那契数

对于 i ≈ 0，我们定义 ***Fibonacci numbers*** F $_i$ 如下：

$$F_i = \begin{cases} 0 & \text{if } i = 0 , \\ 1 & \text{if } i = 1 , \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2 . \end{cases} \tag{3.31}$$

因此，在前两个斐波那契数列之后，每个斐波那契数列都是前两个数列之和，从而得出以下数列

0; 1; 1; 2； 3； 5； 8； 13； 21； 34； 55； :::::

斐波那契数与 ***golden ratio*** 及其共轭 y 相关，它们是方程的两个根

$$x^2 = x + 1 .$$

练习 3.3-7 要求你证明，黄金比例由下式给出

$$\begin{aligned} \phi &= \frac{1 + \sqrt{5}}{2} \\ &= 1.61803 \ldots, \end{aligned} \tag{3.32}$$

及其共轭

$$\begin{aligned} \widehat{\phi} &= \frac{1 - \sqrt{5}}{2} \\ &= -.61803 \ldots . \end{aligned} \tag{3.33}$$

具体来说，我们有

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

可以通过归纳法证明（练习 3.3-8）。由于 $|$|y $|$|< 1，我们有

$$\begin{aligned} \frac{|\widehat{\phi}^i|}{\sqrt{5}} &< \frac{1}{\sqrt{5}} \\ &< \frac{1}{2} , \end{aligned}$$

这意味着

$$F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor , \tag{3.34}$$

也就是说，第 i 个斐波那契数 F $_i$ 等于 ^{i}/ p 5 四舍五入到最接近的整数。因此，斐波那契数呈指数增长。

练习

3.3-1
证明：如果 f.n/ 和 g.n/ 是单调递增函数，则函数 f.n/ C g.n/ 和 f.g.n// 也都是单调递增函数；且如果 f.n/ 和 g.n/ 另外都是非负的，则 f.n/g.n/ 是单调递增的。

3.3-2
证明：对于任何整数 n 和实数 ，在范围 0 1 内，b ncC d.1 ne D n。

3.3-3
利用公式（3.14）或其他方法证明，对于任何实数常数 k，.n C o.n// ^{k} D ，.n ^{k}/。得出 dne ^{k} D ，.n ^{k}/ 和 bnc ^{k} D ，.n ^{k}/。

3.3-4
证明下列命题：

a.方程(3.21)。***b.*** 方程(3.26
)3(3.28)。

c.lg. ，.n// D ，.lg n/.

3.3-5
函数 dlg ne Š 是否多项式有界? 函数 dlg lg ne Š 是否多项式有界?

?***3.3-6***

Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

3.3-7证明黄金比率 及其共轭 y 均满足方程 x ^{2} D x C 1。

3.3-8
通过归纳法证明第 i 个斐波那契数满足以下方程

$$F_i = (\phi^i - \hat{\phi}^i)/\sqrt{5},$$

其中 是黄金比率，y 是其共轭。

3.3-9
证明 k lg k D ，.n/ 蕴涵 k D ，.n/ lg n/。

问题

3-1Asymptotic behavior of polynomials

让

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

其中 a $_d$ > 0 是 n 的 d 次多项式，令 k 为常数。利用渐近符号的定义来证明以下性质。

a.如果 k ≥ d，则 p.n/ D O.n ^{k} /.

b. 如果 k ≥ d，则 p.n/ D .n ^{k} /

. ***c.*** 如果 k D d，则 p.n/ D ，.n ^{k}

/. ***d.*** 如果 k > d，则 p.n/ D o.n ^{k} /

. ***e.*** 如果 k < d，则 p.n/ D !.n ^{k} /.

3-2Relative asymptotic growths

对于下表中的每对表达式 .A; B/，指出 A 是否是 B 的 O、o、 、！或 ，。假设 k 1、 > 0 和 c > 1 为常数。以表格形式写出您的答案，并在每个框中写上“是”或“否”。

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

3-3Ordering by asymptotic growth rates

a.按增长顺序排列以下函数。即，找到满足 g $_1$ D .g $_2$ /, g $_2$ D .g $_3$ /, . . . , g $_{29}$ D .g $_{30}$ / 的函数的排列 g $_1$; g $_2$; : : : ; g $_{30}$ 。将列表划分为等价类，当且仅当 f .n/ D .g.n// 时，函数 f .n/ 和 g.n/ 属于同一类。

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

b.给出单个非负函数 f.n/ 的例子，使得对于部分 (a) 中的所有函数 g_i .n/，f.n/ 既不是 O.g_i .n// 也不是 .g_i .n//。

3-4Asymptotic notation properties
令 f.n/ 和 g.n/ 为渐近正函数。证明或反驳下列每个猜想。

a.f .n/ D O.g.n// 意味着 g.n/ D O.f .n//。

b.f .n/ C g.n/ D ，.min ff .n/; g.n/g/。***c.*** f .n/ D O.g.n// 意味着 lg f .n/ D O.lg g.n// ，其中对于所有足够大的 n，lg g.n/ ≥ 1 且 f .n/ ≥ 1。***d.*** f .n/ D O.g.n// 意味着 $2^{f(n)}$ D O ã $2^{g(n)}$ ä。***e.*** f .n/ D O ..f .n// ²/。***f.*** f .n/ D O.g.n// 意味着 g.n/ D .f .n//。***g.*** f .n/ D ，.f .n/2//。***h.*** f .n/ C o.f .n// D ，.f .n//.

3-5Manipulating asymptotic notation
令 f.n/ 和 g.n/ 为渐近正函数，证明下列恒等式：

a. ‘ . ’.f .n/// D ‘ .f .n//。 ***b.*** ‘ .f .n// C O.f . n// D ‘ .f .n//。 ***c.*** ‘ .f .n// C ‘ .g.n// D ‘ .f . n/ C g.n//。 ***d.*** ‘ .f .n// ‘ .g.n// D ‘ .f .n/ g. n//。

e.论证对于任何实数常数 a 1; b 1 > 0 和整数常数 k 1; k 2，以下渐近界成立：

$$(a_1 n)^{k_1} \lg^{k_2}(a_2 n) = \Theta(n^{k_1} \lg^{k_2} n) .$$

f. Prove that for $S \subseteq \mathbb{Z}$, we have

$$\sum_{k \in S} \Theta(f(k)) = \Theta\left(\sum_{k \in S} f(k) \right) ,$$

假设两个和都收敛。

?***g.*** Show that for $S \subseteq \mathbb{Z}$, the following asymptotic bound does not necessarily hold, even assuming that both products converge, by giving a counterexample:

$$\prod_{k \in S} \Theta(f(k)) = \Theta\left(\prod_{k \in S} f(k) \right) .$$

3-6Variations on 否 ***and Ω***

有些作者定义　符号的方式与本教科书略有不同。我们将使用命名法 ∞（读作 “omega inûnity”）来表示这种替代定义。如果存在一个正常数 c，使得对于无穷多个整数 n，f .n/ ¹ cg.n/ ¹ 0，则我们称 f .n/ D ∞ .g.n//。

a.证明对于任意两个渐近非负函数 f .n/ 和 g.n/，都有 f .n/ D O.g.n// 或 f .n/ D ∞ .g.n//（或两者）。***b.*** 证明存在两个渐近非负函数 f .n/ 和 g.n/，对于它们，f .n/ D O.g.n// 和 f .n/ D .g.n// 都不成立。***c.*** 描述使用 ∞ 符号代替 符号来表征程序运行时间的潜在优点和缺点。

有些作者也以略有不同的方式定义 O。我们将使用 O′ 作为替代定义：当且仅当 jf .n/j D O.g.n// 时，f .n/ D O′.g.n//。

d.如果我们用 O′ 代替 O 但仍使用 ，那么第 56 页定理 3.1 中的 “当且仅当 ” 的每个方向会发生什么情况？

有些作者将 e O (读作 “soft-oh”) 定义为忽略对数因子的 O：

$$\widetilde{O}(g(n)) = \{f(n) : \text{ there exist positive constants } c, k, \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n)\lg^k(n) \text{ for all } n \ge n_0\} .$$

e.类似地，证明 e 和 e ，。证明定理 3.1 的相应类比。

3-7Iterated functions

我们可以将 lg * 函数中使用的迭代运算符 * 应用于实数上的任何单调递增函数 f.n/。对于给定常数 c 2 R，我们通过以下公式定义迭代函数 f_c^*

$$f_c^*(n) = \min\{i \ge 0 : f^{(i)}(n) \le c\} ,$$

在所有情况下，都不需要明确定义。换句话说，量 f_c^* .n/ 是将函数 f 的自变量减小到 c 或更小所需的迭代应用的最小次数。

对于下表中的每个函数 f .n/ 和常数 c，给出 f_c^* .n/ 尽可能严格的界限。如果不存在满足 $f^{(i)}$ .n/ c 的 i，则将 “unde- ûned” 写为答案。

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

章节注释

Knuth [259] 将 O 符号的起源追溯到 P. Bachmann 于 1892 年发表的一本数论教材。o 符号是 E. Landau 于 1909 年为讨论素数分布而发明的。Knuth [265] 提倡使用 和 ，符号，以纠正文献中流行但技术上不太严谨的用 O 符号表示上界和下界的做法。如本章前面所述，许多人在 ，符号更精确的地方继续使用 O 符号。问题 3-6 中引入了软 oh 符号 e O

由 Babai、Luks 和 Seress [31] 定义，尽管它最初写成 O。现在有些作者将 e O .g.n// 定义为忽略 g.n/ 中对数的因子，而不是 n 中的对数因子。根据这个定义，我们可以说 n2 ^{n} D e O.2 ^{n} /，但根据问题 3-6 中的定义，这个说法不正确。关于渐近符号的历史和发展的进一步讨论出现在 Knuth [259, 265] 和 Brassard 和 Bratley [70] 的著作中。

虽然各种定义在大多数情况下是一致的，但并非所有作者都以相同的方式定义渐近符号。一些替代定义涵盖非渐近非负函数，只要它们的绝对值有适当的界限。

方程(3.29)是Robbins [381]提出的。初等数学函数的其他性质可以在任何优秀的数学参考书中找到，例如Abramowitz和Stegun [1]或Zwillinger [468]，也可以在微积分书籍中找到，例如Apostol [19]或Thomas等人[433]。Knuth [259]和Graham、Knuth和Patashnik [199]包含大量有关计算机科学中使用的离散数学的材料。

4 Divide-and-Conquer

分治法是设计渐近有效算法的强大策略。我们在学习归并排序时在第 2.3.1 节中看到了分治法的一个例子。在本章中，我们将探索分治法的应用，并获得宝贵的数学工具，您可以使用这些工具来解决分析分治算法时出现的递归。

回想一下，对于分治法，您要递归地解决给定的问题（实例）。如果问题足够小4，那么 ***base case***4，您直接解决它而不进行递归。否则4，那么 ***recursive case***4，您将执行三个典型步骤：

将问题划分为一个或多个子问题，这些子问题都是同一问题的较小实例。

通过递归解决子问题来解决它们。

结合苏 b问题解决方案形成原始解决方案 的问题。

分而治之算法将大问题分解为更小的子问题，而这些子问题本身又可以分解为更小的子问题，依此类推。当递归达到基准情况且子问题足够小而可以直接解决而无需进一步递归时，递归 ***bottoms out***。

复发

要分析递归分治算法，我们需要一些数学工具。***recurrence*** 是一个方程，它用函数在其他参数（通常较小）上的值来描述函数。递归与分治法密切相关，因为它们为我们提供了一种自然的方式来用数学方式描述递归算法的运行时间。当我们分析归并排序的最坏情况运行时间时，您在第 2.3.2 节中看到了一个递归示例。

对于第 4.1 和 4.2 节中介绍的分治矩阵乘法算法，我们将推导出描述其最坏情况运行时间的递归。要理解这两个分治算法为何如此执行，您需要学习如何解决描述其运行时间的递归。第 4.3 节至第 4.7 节讲授了几种解决递归的方法。这些部分还探讨了递归背后的数学原理，这可以让您更直观地设计自己的分治算法。

我们希望尽快了解算法。因此，我们现在只介绍一些递归基础知识，然后在查看矩阵乘法示例后，我们将更深入地研究递归，尤其是如何解决它们。

递归的一般形式是使用函数本身描述整数或实数上的函数的方程或不等式。它包含两个或更多个案例，具体取决于参数。如果案例涉及对不同（通常较小）输入的函数的递归调用，则为 ***recursive case***。如果案例不涉及递归调用，则为 ***base case***。可能有零个、一个或多个函数满足递归语句。如果至少有一个函数满足递归，则递归为 ***well defined***，否则为 ***ill defined***。

算法递归

我们对描述分治算法运行时间的递归尤其感兴趣。如果对于每个足够大的 ***threshold*** 常数 $n_0 > 0$，以下两个属性成立，则递归 T .n/ 为 ***algorithmic***：

1. 对于所有 $n < n_0$，我们有 T .n/ D ，.1/。2. 对于所有 $n \neg n_0$，每条递归路径都会在有限次数的递归调用中终止于一个已定义的基准案例。

与我们有时滥用渐近符号（参见第 60 页）类似，当一个函数没有针对所有自变量进行定义时，我们认为这个定义受限于 T.n/ 定义的 n 的值。

为什么表示（正确的）分治算法的最坏情况运行时间的递归 T .n/ 会对所有足够大的阈值常数满足这些性质？第一个性质表明存在常数 $c_1; c_2$ 使得对于 $n < n_0$，$0 < c_1 = T .n/ = c_2$。对于每个合法输入，算法必须在有限时间内输出它所求解问题的解（参见第 1.1 节）。因此，我们可以让 c_1 表示调用和返回过程所需的最短时间，该时间必须为正，因为调用过程需要执行机器指令。如果没有合法的输入，则算法的运行时间可能无法针对某些 n 值进行定义，但必须针对至少一个 n 值进行定义，否则 “ 算法 ” 无法解决任何问题。因此，我们可以让 c_2 成为算法在任何大小为 $n < n_0$ 的输入上的最大运行时间，其中 n_0 是

足够大，以至于算法至少能解决一个小于 n_0 的问题。最大值定义得很好，因为至多有有限个小于 n_0 的输入，如果 n_0 足够大，则至少有一个。因此，T .n/ 满足第一个属性。如果第二个属性对 T .n/ 不成立，则算法不正确，因为它最终会陷入无限递归循环或无法计算解决方案。因此，正确的分治算法的最坏情况运行时间的递归是算法的，这是有道理的。

重复惯例

我们采用以下惯例：

> *Whenevera recurrence is stated without an explicit base case, we assume that the recurrence is algorithmic*.

这意味着您可以自由选择任何足够大的阈值常数 n_0，用于 T .n/ D ，.1/ 的基例范围。有趣的是，在分析算法时，您可能看到的大多数算法递归的渐近解并不依赖于阈值常数的选择，只要它足够大以使递归得到很好的定义即可。

当我们放弃整数定义的递归中的任何上限或下限并将其转换为实数定义的递归时，算法分治递归的渐近解也不会改变。第 4.7 节给出了忽略上限和下限的充分条件，该条件适用于您可能看到的大多数分治递归。因此，我们经常会陈述没有上限和下限的算法递归。这样做通常会简化递归的陈述，以及我们对它们进行的任何数学运算。

有时，你可能会看到一些不是方程而是不等式的递归，例如 T .n/ 2T .n /2/ C ，.n/。由于这样的递归仅陈述了 T .n/ 的上限，因此我们使用 O 符号而不是 ，符号来表示它的解。同样，如果不等式反转为 T .n/ ‹ 2T .n/2/ C ，.n/，那么，由于递归仅给出了 T .n/ 的下限，因此我们在其解中使用 符号。

分而治之和递归

本章通过介绍和使用递归来分析两个用于乘以 n n 矩阵的分治算法，从而说明分治法。第 4.1 节介绍了一种简单的分治算法，该算法通过将大小为 n 的矩阵乘法问题分解为四个大小为 n/2 的子问题来解决该问题，然后以递归方式求解。该算法的运行时间可以用递归来表征

$$T(n) = 8T(n/2) + \Theta(1) ,$$

最终得到解 T .n/ D ，.n ^{3} /。虽然这种分治算法并不比使用三重嵌套循环的直接方法快，但它可以得到 V. Strassen 提出的渐近更快的分治算法，我们将在 4.2 节中探讨这一点。Strassen 的出色算法将一个大小为 n 的问题划分为七个大小为 n/2 的子问题，并以递归方式求解。Strassen 算法的运行时间可以用递归公式来描述

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

其解为 T .n/ D ，.n $^{\lg 7}$ / D O.n $^{2.81}$ /。Strassen 算法渐进地击败了直接循环方法。

这两种分治算法都将一个大小为 n 的问题分解为几个大小为 n/2 的子问题。虽然使用分治算法时，所有子问题的大小通常都相同，但情况并非总是如此。有时将大小为 n 的问题分解为不同大小的子问题会很有成效，然后描述运行时间的递归会反映出这种不规则性。例如，考虑一种分治算法，它将一个大小为 n 的问题分解为一个大小为 n/3 的子问题和另一个大小为 2n/3 的子问题，花费 ，.n/ 的时间来分解问题并合并子问题的解。然后，该算法的运行时间可以通过递归来描述

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) ,$$

结果是解 T .n/ D ，.n lg n/。我们甚至会在第 9 章中看到一种算法，该算法通过递归解决一个大小为 n/5 的子问题和另一个大小为 7n/10 的子问题来解决大小为 n 的问题，其中除法和合并步骤花费 ，.n/ 的时间。其性能满足递归

$$T(n) = T(n/5) + T(7n/10) + \Theta(n) ,$$

其解为 T .n/ D ，.n/。

虽然分治算法通常会产生大小为原始问题大小的常数部分的子问题，但情况并非总是如此。例如，线性搜索的递归版本（参见练习 2.1-4）只创建一个子问题，其元素比原始问题少一个。每次递归调用都需要常数时间加上递归解决元素少一个的子问题的时间，从而导致递归

$$T(n) = T(n-1) + \Theta(1) ,$$

其解为 T .n/ D .n/。然而，绝大多数有效的分治算法所解决的子问题都是原始问题规模的常数部分，这也是我们将重点关注的地方。

解决递归问题

在学习了 4.1 和 4.2 节中矩阵乘法的分治算法之后，我们将探索几种用于解决递归问题的数学工具，即获得其解的渐近 β 界、O 界或 β 界。我们需要简单易用的工具来处理最常见的情况。但我们也需要通用工具，这些工具可能只需多花一点功夫，就能解决不太常见的情况。本章提供了四种解决递归问题的方法：

在***substitution method***（第 4.3 节）中，你猜测一个边界的形式，然后使用数学归纳法来证明你的猜测是正确的，并求解常数。这种方法可能是解决递归问题最可靠的方法，但它也要求你做出正确的猜测并给出归纳证明。

recursion-tree method（第 4.4 节）将递归建模为一棵树，其节点表示递归各个级别产生的成本。要解决递归问题，您需要确定每个级别的成本并将其相加，也许可以使用第 A.2 节中的边界求和技术。即使您不使用此方法来正式证明边界，它也有助于猜测用于替代方法的边界形式。

master method（第 4.5 和 4.6 节）是最简单的方法。它为以下形式的递归提供了界限

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

其中 a > 0 和 b > 1 是常数，f .n/ 是给定的 “ 驱动 ” 函数。这种类型的递归在算法研究中出现的频率比其他任何类型都要高。它描述了一种分治算法，该算法创建了子问题，每个子问题的大小都是原始问题的 1/b 倍，使用 f .n/ 时间进行分治和合并步骤。要应用主方法，您需要记住三种情况，但一旦记住，您就可以轻松确定许多分治算法的运行时间的渐近界限。

Akra-Bazzi method（第 4.7 节）是解决分治递归的通用方法。尽管它涉及微积分，但它可以用来解决比主方法更复杂的递归。

4.1 方阵相乘

我们可以使用分治法来乘以方阵。如果你以前见过矩阵，那么你可能知道如何乘以它们。（否则，

您应该阅读第 D.1 节。）设 A D .a $_{ik}$ / 和 B D .b $_{jk}$ / 为方阵 n n 矩阵。矩阵乘积 C D A B 也是一个 n n 矩阵，其中对于 i; j D 1; 2; : : : ; n，C 的 .i; j / 项由下式给出

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \ . \tag{4.1}$$

一般来说，我们假设矩阵为 ***dense***，这意味着 n 个 ² 元素中的大多数不为 0，与 ***sparse*** 相反，其中 n 个 ² 元素中的大多数为 0，而非零元素可以比 n n 数组更紧凑地存储。

计算矩阵 C 需要计算 n ² 个矩阵项，每个项都是来自 A 和 B 的输入元素的 n 个成对乘积之和。MATRIX-MULTIPLY 过程以一种直接的方式实现了这一策略，并且稍微概括了该问题。它将三个 n n 矩阵 A、B 和 C 作为输入，并将矩阵乘积 A B 加到 C 上，将结果存储在 C 中。因此，它计算的是 C D C C A B，而不仅仅是 C D A B。如果只需要乘积 A B，则只需在调用该过程之前将 C 的所有 n ² 项初始化为 0，这会额外花费 ，.n ²/ 时间。我们将看到，矩阵乘法的成本渐近地大于这个初始化成本。

矩阵乘法 .A; B; C; n/

1 对于 i D 1 到 n // 计算 n 行中的每一行的条目 2 对于 j D 1 到 n // 计算第 i 行中的 n 个条目 3 对于 k D 1 到 n 4 c $_{ij}$ D c $_{ij}$ C a $_{ik}$ b $_{kj}$ // 添加方程（4.1）的另一项

MATRIX-MULTIPLY 的伪代码工作原理如下。第 134 行的 for 循环计算每行 i 的条目，在给定的行 i 内，第 234 行的 for 循环计算每列 j 的每个条目 c $_{ij}$。第 334 行的 for 循环的每次迭代都会在公式 (4.1) 中添加一项。

因为每个三重嵌套的 for 循环都恰好运行 n 次迭代，并且第 4 行的每次执行都需要常数时间，所以 M ATRIX-MULTIPLY 过程的运行时间为 ，.n ³/。即使我们加上将 C 初始化为 0 的 ，.n ²/ 时间，运行时间仍为 ，.n ³/。

简单的分治算法

让我们看看如何使用分治法计算矩阵乘积 A B。对于 n > 1，除法步骤将 n n 矩阵划分为四个 n/2 n/2 子矩阵。我们假设 n 是 2 的精确幂，这样当算法递归时，我们就能保证子矩阵维度是整数。（练习 4.1-1 要求你

放宽这一假设。）与 MATRIX-MULTIPLY 一样，我们实际上会计算 C D C C A B 。但为了简化算法背后的数学，我们假设 C 已初始化为零矩阵，这样我们确实在计算 C D A B 。

划分步骤将 n n 矩阵 A、B 和 C 分别视为四个 n/2 n/2 子矩阵：

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}. \tag{4.2}$$

然后我们可以将矩阵乘积写成

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{4.3}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}, \tag{4.4}$$

对应于方程

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.5}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.6}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.7}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.8}$$

方程 (4.5)3(4.8) 涉及八次 n/2 n/2 乘法和四次 n/2 n/2 子矩阵的加法。

当我们寻求将这些方程转换为可以用伪代码描述的算法，甚至真正实现的算法时，有两种常见的方法来实现矩阵分割。

一种策略是分配临时存储空间来保存 A 的四个子矩阵 A_{11}、A_{12}、A_{21} 和 A_{22} 以及 B 的四个子矩阵 B_{11}、B_{12}、B_{21} 和 B_{22}。然后将 A 和 B 中的每个元素复制到相应子矩阵中的相应位置。在递归征服步骤之后，将 C 的四个子矩阵 C_{11}、C_{12}、C_{21} 和 C_{22} 中的元素复制到 C 中的相应位置。此方法需要 ，.n^2/ 次，因为要复制 $3n^2$ 个元素。

第二种方法使用索引计算，速度更快，更实用。可以通过指示子矩阵在矩阵中的位置来指定子矩阵，而无需触及任何矩阵元素。对矩阵（或递归地，子矩阵）进行分区仅涉及对此位置信息的算术，该信息的大小与矩阵的大小无关，为常数。对子矩阵元素的更改会更新原始矩阵，因为它们占用相同的存储空间。

接下来，我们假设使用索引计算，并且分区可以在 ，.1/ 时间内完成。练习 4.1-3 要求你证明，无论矩阵分区使用第一种方法复制还是第二种方法，对矩阵乘法的整体渐近运行时间都没有影响

这是指数计算的第二种方法。但是对于其他分治矩阵计算，例如矩阵加法，它可能会产生影响，正如练习 4.1-4 要求您展示的那样。

过程 MATRIX-MULTIPLY-RECURSIVE 使用方程 (4.5)3(4.8) 来实现方阵乘法的分治策略。与 MATRIX-MULTIPLY 一样，过程 MATRIX-MULTIPLY-RECURSIVE 计算 C D C C A B，因为如果需要，可以在调用过程之前将 C 初始化为 0，以便只计算 C D A B。

矩阵乘法递归 .A; B; C; n/

1 if n ==1 2 // 基本情况。 3 c 11 D c 11 C a 11 b 11 4 返回 5 // 除法。 6 将 A、B 和 C 分成 n/2 n/2 个子矩阵 A 11 ; A 12 ; A 21 ; A 22 ; B 11 ; B 12 ; B 21 ; B 22 ; 和 C 11 ; C 12 ; C 21 ; C 22 ; 7 // 征服。 8 矩阵乘法递归。A 11 ; B 11 ; C 11 ; n/2/ 9 矩阵乘法递归 .A 11 ; B 12 ; C 12 ; n/2/ 10 矩阵乘法递归 .A 21 ; B 11 ; C 21 ; n/2/ 11 矩阵乘法递归 . A 21 ; B 12 ; C 22 ; n/2/ 12 矩阵乘法递归 .A 12 ; B 21 ; C 11 ; n/2/ 13 矩阵乘法递归 .A 12 ; B 22 ; C 12 ; n/2/ 14 矩阵乘法递归 .A 22 ; B 21 ; C 21 ; n/2/ 15 矩阵乘法递归 .A 22 ; B 22 ; C 22 ; n/2/

在介绍伪代码时，我们将推导出一个递归式来描述其运行时间。令 T .n/ 为使用此过程将两个 n n 矩阵相乘的最坏时间。

在基本情况中，当 n D 1 时，第 3 行仅执行一次标量乘法和一次加法，这意味着 T .1/ D ，.1/。按照我们对常量基本情况的惯例，我们可以在递归语句中省略此基本情况。

递归情况发生在 n > 1 时。如前所述，我们将使用索引计算对第 6 行中的矩阵进行分区，耗时 ，.1/ 次。第 83 至第 15 行共递归调用 MATRIX-MULTIPLY-RECURSIVE 八次。前四个递归调用计算方程 (4.5)3(4.8) 的第一项，随后的四个递归调用计算并添加第二项。每个递归调用将 A 的子矩阵和 B 的子矩阵的乘积添加到相应的子矩阵

由于索引计算，C 可以原地更新。由于每次递归调用都会将两个 n/2 n/2 矩阵相乘，从而为总运行时间贡献 T .n/2/，因此所有八次递归调用所花费的时间是 8T .n/2/。没有合并步骤，因为矩阵 C 是原地更新的。因此，递归情况的总时间是分区时间和所有递归调用时间的总和，即 ，.1/ C 8T .n/2/。

因此，省略基本情况的陈述，我们对 MATRIX-MULTIPLY-RECURSIVE 运行时间的递归是

$$T(n) = 8T(n/2) + \Theta(1) . \tag{4.9}$$

从4.5节的主方法中我们可以看出，递归(4.9)有解T.n/D ′ .n³/，这意味着它具有与直接的MATRIX-MULTIPLY过程相同的渐近运行时间。

为什么这个递归式的 ，.n ³/ 解比第 41 页的归并排序递归式 (2.3) 的 ，.n l g n/ 解大这么多呢？毕竟，归并排序的递归式包含一个 ，.n/ 项，而递归矩阵乘法的递归式只包含一个 ，.1/ 项。

让我们想象一下递归 (4.9) 的递归树与归并排序的递归树（如第 43 页的图 2.5 所示）相比会是什么样子。归并排序递归中的因子 2 决定了每个树节点有多少个子节点，这反过来又决定了树的每一级中有多少项对和做出贡献。相比之下，对于矩阵乘法递归的递归 (4.9)，递归树中的每个内部节点都有 8 个子节点，而不是 2 个，这导致 “ 更加茂密” 的递归树具有更多的叶子，尽管每个内部节点都小得多。因此，递归 (4.9) 的解的增长速度比递归 (2.3) 的解快得多，这在实际解中得到了证实： ，.n ³/ 对 ，.n lg n/。

练习

Note: 在尝试这些练习之前，您可能希望阅读第 4.5 节。

4.1-1

将 MATRIX-MULTIPLY-RECURSIVE 推广到 n n 个矩阵的乘法，其中 n 不一定是 2 的幂。给出一个描述其运行时间的递归。论证它在最坏情况下的运行时间为 ，.n ³/。

4.1-2

使用 MATRIX-MULTIPLY-RECURSIVE 作为子程序，您能多快地将一个 k n n 矩阵（k n 行和 n 列）乘以一个 n k n 矩阵（其中 k 1）？回答将一个 n k n 矩阵乘以一个 k n n 矩阵的相同问题。哪个速度渐近更快，快多少？

4.1-3
假设不是在 MATRIX-MULTIPLY-RECURSIVE 中通过索引计算来划分矩阵，而是将 A、B 和 C 的相应元素分别复制到单独的 n/2n/2 子矩阵 A_{11}、A_{12}、A_{21}、A_{22}；B_{11}、B_{12}、B_{21}、B_{22}；和 C_{11}、C_{12}、C_{21}、C_{22} 中。递归调用之后，将 C_{11}、C_{12}、C_{21} 和 C_{22} 的结果复制回 C 中的适当位置。递归（4.9）如何变化，它的解决方案是什么？

4.1-4
编写分治算法 MATRIX-ADD-RECURSIVE 的伪代码，该算法将两个 n n 矩阵 A 和 B 划分为四个 n/2 n/2 子矩阵，然后递归地对相应的子矩阵对求和，从而对两个 n n 矩阵 A 和 B 求和。假设矩阵划分使用 ，.1/ 时间索引计算。编写 MATRIX-ADD-RECURSIVE 最坏情况运行时间的递归式，并求解递归式。如果使用 ，.n^2/ 时间复制来实现划分而不是索引计算，会发生什么情况？

4.2 Strassen 矩阵乘法算法

你可能很难想象任何矩阵乘法算法可以在 o.n^3/ 时间内完成，因为矩阵乘法的自然定义需要 n^3 次标量乘法。事实上，许多数学家都认为不可能在 o.n^3/ 时间内完成矩阵乘法，直到 1969 年，V. Strassen [424] 发表了一个出色的递归算法，用于乘以 n 乘 n 个矩阵。Strassen 的算法运行时间为 o.$n^{\lg 7}$/。由于 lg 7 D 2:8073549 : : :，Strassen 算法运行时间为 O.$n^{2.81}$/，渐近地优于‘ .n^3/ MATRIX-MULTIPLY 和 MATRIX-MULTIPLY-RECURSIVE 过程。

斯特拉森方法的关键是利用矩阵乘法递归过程中的分而治之思想，但使递归树不那么繁琐。实际上，我们将使每个除法和合并步骤的工作量增加一个常数倍，但繁琐程度的减少是值得的。我们不会将繁琐程度从八路递归分支（4.9）一直降低到两路递归分支（2.3），但我们会对其进行一点改进，这会带来很大的不同。斯特拉森算法不需要对 n/2 n/2 矩阵进行八次递归乘法，而只需执行七次。消除一次矩阵乘法的代价是需要对 n/2 n/2 矩阵进行几次新的加法和减法，但仍然只是一个常数。我们不再到处说“加法和减法”，而是采用调用的通用术语——

将它们都设置为 “加法”，因为减法在结构上与加法的计算相同，只是符号有所不同。

为了了解如何减少乘法次数，以及为什么减少乘法次数对于矩阵计算是可取的，假设有两个数字 x 和 y，并且要计算 $x^2 \quad y^2$。简单的计算需要两次乘法来计算 x 和 y 的平方，然后进行一次减法（你可以将其视为 “负加”）。但让我们回想一下古老的代数技巧 $x^2 \quad y^2 \mathrel{D} x^2 \quad xy \mathrel{C} xy \quad y^2 \mathrel{D} x.x \quad y/ \mathrel{C} y.x \quad y/ \mathrel{D} .x \mathrel{C} y/.x \quad y/$。使用所需数量的这个公式，你可以计算和 x C y 以及差 x y，然后将它们相乘，只需要一次乘法和两次加法。以额外的加法为代价，只需要一次乘法就可以计算出一个看起来需要两次的表达式。如果 x 和 y 是标量，则没有太大区别：两种方法都需要三次标量运算。但是，如果 x 和 y 是大型矩阵，则乘法的成本大于加法的成本，在这种情况下，第二种方法优于第一种方法，尽管不是渐近的。

施特拉森以增加矩阵加法为代价来减少矩阵乘法次数的策略并不明显——这也许是本书中最大的低估！与矩阵乘法递归一样，施特拉森算法使用分而治之的方法来计算 C D C C A B，其中 A、B 和 C 都是 n n 矩阵，n 是 2 的精确幂。施特拉森算法分四步根据第 82 页的方程 (4.5)3(4.8) 计算 C 的四个子矩阵 C_{11}、C_{12}、C_{21} 和 C_{22}。我们将在开发总运行时间的递归 T .n/ 的过程中分析成本。让我们看看它是如何工作的：

1. 如果 n D 1，则每个矩阵都包含一个元素。执行一个标量乘法和一个标量加法，如 MATRIX-MULTIPLY-RECURSIVE 的第 3 行所示，耗时 ，.1/ 次，然后返回。否则，将输入矩阵 A 和 B 以及输出矩阵 C 划分为 n/2 n/2 个子矩阵，如公式 (4.2) 所示。此步骤通过索引计算耗时 ，.1/ 次，与 MATRIX-MULTIPLY-RECURSIVE 相同。

2. 创建 n/2 n/2 矩阵 $S_1; S_2; : : :; S_{10}$，每个矩阵都是步骤 1 中两个子矩阵的和或差。创建并将七个 n/2 n/2 矩阵 $P_1; P_2; : : :; P_7$ 的元素清零，以保存七个 n/2 n/2 矩阵乘积。可以在 ，$.n^2/$ 时间内创建所有 17 个矩阵，并初始化 P_i。

3. 使用步骤 1 中的子矩阵和步骤 2 中创建的矩阵 $S_1; S_2; : : :; S_{10}$，递归计算七个矩阵乘积 $P_1; P_2; : : :; P_7$ 中的每一个，耗时 7T .n/2/ 次。

4. 通过加减各个 P_i 矩阵来更新结果矩阵 C 的四个子矩阵 C_{11}；C_{12}；C_{21}；C_{22}，这需要 ，$.n^2/$次。

我们稍后会看到步骤 234 的细节，但我们已经有足够的信息来为施特拉森方法的运行时间建立递归。与通常情况一样，步骤 1 中的基本情况需要 ，.1 / 时间，在陈述递归时我们将省略这段时间。当 n > 1 时，步骤 1、2 和 4 总共需要 ，.n 2 / 时间，步骤 3 需要 n/2 n/2 矩阵的七次乘法。因此，我们得到施特拉森算法的运行时间的以下递归：

$$T(n) = 7T(n/2) + \Theta(n^2) . \tag{4.10}$$

与矩阵乘法递归相比，我们用一个递归子矩阵乘法换取了常数个子矩阵加法。一旦你理解了递归及其解决方案，你就能明白为什么这种权衡实际上会导致更短的渐近运行时间。根据第 4.5 节中的主方法，递归 (4.10) 的解为 T .n/ D ，.n lg 7 / D O.n 2.81 /，击败了 ，.n 3 / 时间算法。

现在，让我们深入研究细节。第 2 步创建以下 10 个矩阵：

$$\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}$$

此步骤对 n/2 n/2 矩阵进行 10 次加或减，耗时 ，.n 2 / 次。

步骤 3 递归地将 n/2 n/2 矩阵乘以 7 次，以计算以下 n/2 n/2 矩阵，每个矩阵都是 A 和 B 子矩阵乘积的和或差：

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

算法执行的唯一乘法是这些方程式中间一列中的乘法。右侧一列仅显示这些乘积相对于在步骤 1 中创建的原始子矩阵的等式，但算法从未明确计算过这些项。

步骤 4 对步骤 3 中创建的各个 P_i 矩阵乘积 C 的四个 n/2 n/2 子矩阵进行加减运算。我们首先

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

展开右侧的计算，将每个 P_i 展开到自己的行上，并垂直对齐抵消的项，我们看到对 C_{11} 的更新等于

$$\begin{array}{l} A_{11}\cdot B_{11} + A_{11}\cdot B_{22} + A_{22}\cdot B_{11} + A_{22}\cdot B_{22} \\ \qquad\qquad - A_{22}\cdot B_{11} \qquad\qquad + A_{22}\cdot B_{21} \\ \qquad - A_{11}\cdot B_{22} \qquad\qquad\qquad\qquad - A_{12}\cdot B_{22} \\ \qquad\qquad\qquad\qquad - A_{22}\cdot B_{22} - A_{22}\cdot B_{21} + A_{12}\cdot B_{22} + A_{12}\cdot B_{21} \\ \hline A_{11}\cdot B_{11} \qquad\qquad\qquad\qquad\qquad\qquad \mathrm{C\,A_{12}\,B_{21}} \ ; \end{array}$$

对应于公式（4.5）。同样地，设

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

意味着对 C_{12} 的更新等于

$$\begin{array}{l} A_{11}\cdot B_{12} - A_{11}\cdot B_{22} \\ \qquad + A_{11}\cdot B_{22} + A_{12}\cdot B_{22} \\ \hline A_{11}\cdot B_{12} \qquad\qquad + A_{12}\cdot B_{22} \ , \end{array}$$

对应公式（4.6）。设

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

意味着对 C_{21} 的更新等于

$$\begin{array}{l} A_{21}\cdot B_{11} + A_{22}\cdot B_{11} \\ \qquad - A_{22}\cdot B_{11} + A_{22}\cdot B_{21} \\ \hline A_{21}\cdot B_{11} \qquad\qquad + A_{22}\cdot B_{21} \ , \end{array}$$

对应方程（4.7）。最后设

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

意味着对 C_{22} 的更新等于

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

对应于公式 (4.8)。总而言之，由于我们在步骤 4 中对 n/2n/2 矩阵进行了 12 次加减运算，因此此步骤确实需要 ，.n 2 / 次。

我们可以看到，Strassen 的卓越算法包含 134 步，使用 7 次子矩阵乘法和 18 次子矩阵加法生成正确的矩阵乘积。我们还可以看到递归 (4.10) 表征了它的运行时间。由于第 4.5 节表明此递归具有解 T .n/ D ，.n lg 7 / D o.n 3 /，因此 Strassen 的方法渐进地击败了 ，.n 3 / MATRIX-MULTIPLY 和 MATRIX-MULTIPLY-RECURSIVE 程序。

练习

Note: 在尝试这些练习之前，您可能希望阅读第 4.5 节。

4.2-1
使用 Strassen 算法计算矩阵乘积

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

展示你的作品。

4.2-2
为斯特拉森算法编写伪代码。

4.2-3
最大的 k 是多少，使得如果你能用 k 次乘法（不假设乘法交换性）乘以 3 3 个矩阵，那么你就能在 o.n lg 7 / 时间内乘以 n n 个矩阵？这个算法的运行时间是多少？

4.2-4
V. Pan 发现了一种使用 132,464 次乘法来乘以 68 68 个矩阵的方法，一种使用 143,640 次乘法来乘以 70 70 个矩阵的方法，以及一种使用 155,424 次乘法来乘以 72 72 个矩阵的方法。在分治矩阵乘法算法中使用哪种方法可获得最佳渐近运行时间？它与 Strassen 算法相比如何？

4.2-5
说明如何仅使用三次实数乘法将复数 a C bi 和 c C d i 相乘。该算法应以 a、b、c 和 d 作为输入，并分别生成实部 ac bd 和虚部 ad C bc。

4.2-6
假设你有一个 ，.n ˛/ 时间算法来计算 n n 个矩阵的平方，其中 ˛ 2. 说明如何使用该算法在 ，.n ˛/ 时间内将两个不同的 n n 矩阵相乘。

4.3 求解递归式的代换法

现在您已经了解了递归如何表征分治算法的运行时间，让我们学习如何解决它们。本节我们从 ***substitution method*** 开始，这是本章四种方法中最通用的。替换方法包括两个步骤：

1. 使用符号常数猜测解决方案的形式。 2. 使用数学归纳法证明解决方案有效，并找到常数。

要应用归纳假设，你需要用猜测的解代替较小值的函数，因此得名“替代法。”这种方法很有效，但你必须猜测答案的形式。虽然产生一个好的猜测似乎很难，但稍加练习就能迅速提高你的直觉。

您可以使用替换法建立递归的上限或下限。通常最好不要同时尝试两者。也就是说，与其尝试直接证明 ' 界，不如先证明 O 界，然后证明 ' 界。两者结合起来，您就会得到 ' 界（第 56 页上的定理 3.1）。

作为替代方法的一个例子，让我们确定递归的渐近上限：

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n) . \tag{4.11}$$

此递归与第 41 页中归并排序的递归 (2.3) 类似，但 üoor 函数除外，它确保 T .n/ 在整数上定义。我们假设渐近上界相同 4T .n/ D O.n lg n/4，并使用替换法来证明这一点。

我们将采用归纳假设，即对所有的 n ∈ n $_0$ ，T .n/ cn lg n，其中，我们将在稍后看到什么之后，选择特定的常数 c > 0 和 n $_0$ > 0。

它们必须遵守的约束条件。如果我们能建立这个归纳假设，我们就能得出 T .n/ D O.n lg n/ 的结论。使用 T .n/ D O.n lg n/ 作为归纳假设是危险的，因为常数很重要，正如我们稍后在讨论陷阱时会看到的那样。

通过归纳推理假设该界限对所有至少等于 n_0 且小于 n 的数字都成立。因此，特别地，如果 $n \geq 2n_0$，则它对 bn/2c 成立，得到 T .bn/2c/ ∠c bn/2c lg.bn/2c/。代入递归 (4.11)4，因此得名 “替代” 方法4得到

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

其中，如果我们限制常数 n_0 和 c 足够大，使得对于 $n \geq 2n_0$，数量 cn 支配着被 “ .n/ ” 项隐藏的匿名函数，则最后一步成立。

我们已经证明了归纳假设对于归纳情况成立，但我们还需要证明归纳假设对于归纳的基本情况也成立，即当 $n_0 \leq n < 2n_0$ 时，T .n/ ≤ cn lg n。只要 $n_0 > 1$（n_0 上的新约束），我们就有 lg n > 0，这意味着 n lg n > 0。因此，我们取 n_0 D 2。由于递归的基本情况（4.11）没有明确说明，按照我们的惯例，T .n/ 是算法的，这意味着 T .2/ 和 T .3/ 是常数（如果它们描述了任何实际程序在大小为 2 或 3 的输入下的最坏情况运行时间，它们就应该是常数）。选择 c D max fT .2/; T .3/g 得出 T .2/ c < .2 lg 2/c 和 T .3/ c < .3 lg 3 /c，为基础案例建立了归纳假设。

因此，对于所有的 $n \geq 2$，我们有 T .n/ + cn lg n，这意味着递归（4.11）的解是 T .n/ + D O.n lg n/。

在算法文献中，人们很少将代换证明做到如此详细，特别是在处理基准情况时。原因是对于大多数算法分而治之递归，基准情况的处理方式都差不多。你将归纳建立在从方便的正常数 n_0 到某个常数 $n_0' > n_0$ 的一个值域上，使得对于 $n \approx n_0'$，递归总是在 n_0 和 n_0' 之间的一个常数大小的基准情况中触底。（此例使用了 n_0' D $2n_0$。）然后，通常很明显，无需详细说明，只要选择适当大的首项常数（例如本例中的 c），就可以使归纳假设对从 n_0 到 n_0' 范围内的所有值都成立。

做出正确的猜测

不幸的是，没有通用的方法可以正确地猜测任意递归的最紧密渐近解。做出正确的猜测需要经验，有时还需要创造力。幸运的是，学习一些解决递归问题的启发式方法，以及尝试递归以获得经验，可以帮助您成为一名优秀的猜测者。您还可以使用递归树（我们将在第 4.4 节中看到）来帮助生成正确的猜测。

如果某个递归式与你之前见过的递归式相似，那么猜测类似的解决方案是合理的。例如，考虑以下递归式

$T(n) = 2T(n/2 + 17) + \Theta(n)$,

定义在实数上。这个递归式看起来有点像归并排序递归式 (2.3)，但是由于在右边的 T 参数中添加了 “17”，所以它更复杂。但是从直觉上讲，这个附加项不会对递归式的解产生很大影响。当 n 很大时，n/2 和 n/2C17 之间的相对差异并不大：两者都将 n 减少了近一半。因此，猜测 T.n/DO.nlgn/ 是有意义的，您可以使用代换法验证它是否正确（参见练习 4.3-1）。

做出正确猜测的另一种方法是确定递归的宽松上限和下限，然后减少不确定性范围。例如，对于递归 (4.11)，您可以从 T .n/ D .n/ 的下限开始，因为递归包含项 ，.n/，并且您可以证明初始上限为 T .n/ D O.n ²/。然后将时间分成两部分，一部分尝试降低上限，另一部分尝试提高下限，直到您收敛到正确的、渐近严格的解决方案，在本例中为 T .n/ D ，.n lg n/。

行业秘诀：减去低阶项

有时，您可能会正确猜出递归解的严格渐近界限，但不知何故，数学在归纳证明中无法解决。问题通常是归纳假设不够强。解决这个问题的诀窍是，当您遇到这样的障碍时，用 *subtracting* 一个低阶项来修改您的猜测。然后数学通常就会通过。

考虑复发

$$T(n) = 2T(n/2) + \Theta(1) \tag{4.12}$$

定义在实数上。我们假设解为 T .n/ D O.n/，并尝试证明 T .n/ cn 为 n ? n $_0$ ，其中我们适当地选择常数 c; n $_0$ > 0。将我们的猜测代入递归式，我们得到

$$\begin{aligned} T(n) &\le 2(c(n/2)) + \Theta(1) \\ &= cn + \Theta(1) , \end{aligned}$$

不幸的是，这并不意味着对于 c 的 *any* 选择，T .n/ cn。我们可能会尝试更大的猜测，比如 T .n/ D O.n 2 /。虽然这个更大的猜测有效，但它只提供了一个宽松的上限。事实证明，我们最初对 T .n/ D O.n/ 的猜测是正确的，而且很严格。然而，为了证明它是正确的，我们必须加强我们的归纳假设。

直观地看，我们的猜测几乎是正确的：我们只偏离了 1/，一个低阶项。然而，数学归纳法要求我们证明归纳假设的 *exact* 形式。让我们试试从之前的猜测中减去一个低阶项的技巧：T .n/ cn d，其中 d 0 是一个常数。我们现在有

$$\begin{aligned} T(n) &\le 2(c(n/2) - d) + \Theta(1) \\ &= cn - 2d + \Theta(1) \\ &\le cn - d - (d - \Theta(1)) \\ &\le cn - d \end{aligned}$$

只要我们选择 d 大于 ，符号隐藏的匿名上限常数。减去一个下阶项就可以了！当然，我们不能忘记处理基本情况，即选择足够大的常数 c，使得 cn d 支配隐式基本情况。

你可能会觉得减去低阶项的想法有悖常理。毕竟，如果数学运算不行，难道不应该增加猜测值吗？不一定！当递归包含多个递归调用（递归 (4.12) 包含两个）时，如果在猜测值中添加一个低阶项，那么最终会为每个递归调用添加一次低阶项。这样做会让你离归纳假设更远。另一方面，如果从猜测值中减去一个低阶项，那么每次递归调用都要减去它一次。在上面的例子中，我们减去了常数 d 两次，因为 T .n/2/ 的系数是 2。我们得到不等式 T .n/ cn d .d .1//，并且我们很容易就找到了 d 的合适值。

避免陷阱

避免在替代法的归纳假设中使用渐近符号，因为它容易出错。例如，对于递归 (4.11)，如果我们不明智地采用 T .n/ D O.n/ 作为我们的归纳假设，我们可能会错误地“证明” T .n/ D O.n/：

$$\begin{aligned} T(n) &\le 2 \cdot O(\lfloor n/2 \rfloor) + \Theta(n) \\ &= 2 \cdot O(n) + \Theta(n) \\ &= O(n) . \qquad \Longleftarrow \textit{wrong!} \end{aligned}$$

这种推理的问题在于，O 符号隐藏的常数发生了变化。我们可以通过使用显式常数重复“证明”来揭示谬误。对于归纳假设，假设对所有 n ∈ n 0，T .n / cn，其中 c; n 0 > 0 为常数。重复不等式链中的前两个步骤可得出

$$\begin{aligned} T(n) &\le 2(c\,\lfloor n/2 \rfloor) + \Theta(n) \\ &\le cn + \Theta(n)\,. \end{aligned}$$

现在，确实 cnC，.n/ D O.n/，但 O 符号隐藏的常数必须大于 c，因为 ，.n/ 隐藏的匿名函数是渐近正的。我们不能采取第三步得出 cn C ，.n/ cn 的结论，从而暴露了谬误。

在使用代换法或更一般的数学归纳法时，必须注意，任何渐近符号隐藏的常数在整个证明过程中都是相同的常数。因此，最好在归纳假设中避免使用渐近符号，并明确命名常数。

这是替代方法的另一个错误用法，它表明递归 (4.11) 的解是 T .n/ D O.n/。我们猜测 T .n/ cn，然后论证

$$\begin{aligned} T(n) &\le 2(c\,\lfloor n/2 \rfloor) + \Theta(n) \\ &\le cn + \Theta(n) \\ &= O(n)\,, \qquad \Longleftarrow \textit{wrong!} \end{aligned}$$

因为 c 是正常数。错误源于我们的目标4证明 T .n/ D O.n/4与我们的归纳假设4证明 T .n/ cn 之间的差异。使用替换方法或在任何归纳证明中，必须证明归纳假设的 *exact* 陈述。在这种情况下，我们必须明确证明 T .n/ cn 以表明 T .n/ D O.n/。

练习

4.3-1

使用代换法证明下列每个在实数上定义的递归都有指定的渐近解：

a.T .n/ D T .n 1/ C n 有解 T .n/ D O.n 2 /. ***b.*** T .n/ D T .n/2/ C ，.1/ 有解 T .n/ D O.lg n/. ***c.*** T .n/ D 2T .n/2/ C n 有解 T .n/ D ，.n lg n/. ***d.*** T .n/ D 2T .n/2 C 17/ C n 有解 T .n/ D O.n lg n/. ***e.*** T .n/ D 2T .n/3/ C ，.n/ 有解 T .n/ D ，.n/. ***f.*** T .n/ D 4T .n/ 2/ C ，.n/ 有解 T .n/ D ，.n 2 /。

4.3-2递归式 T .n/ D 4T .n/2/ C n 的解为 T .n/ D ，.n ²/。说明假设 T .n/ cn ² 的代换证明不成立。然后说明如何减去低阶项以使代换证明成立。

4.3-3
递归式 T .n/ D 2T .n 1/ C 1 的解为 T .n/ D O.2 ⁿ/。说明代换证明在假设 T .n/ c2 ⁿ 下不成立，其中 c > 0 为常数。然后说明如何减去低阶项以使代换证明成立。

4.4 解决递归问题的递归树方法

虽然你可以使用替换法来证明递归的解是正确的，但你可能很难得出一个好的猜测。画出递归树（就像我们在第 2.3.2 节中对归并排序递归的分析中所做的那样）会有所帮助。在 ***recursion tree*** 中，每个节点代表递归函数调用集合中某个子问题的成本。你通常将树中每一层的成本相加以获得每层的成本，然后将所有每层的成本相加以确定所有递归层的总成本。然而，有时将总成本相加需要更多的创造力。

递归树最适合用于生成良好猜测的直觉，然后您可以通过替换法进行验证。但是，如果您在绘制递归树和计算成本时非常细致，则可以将递归树用作递归解决方案的直接证明。但是，如果您仅使用它来生成良好的猜测，则通常可以容忍少量的“草率，”这可以简化数学运算。当您稍后使用替换法验证您的猜测时，您的数学运算应该是精确的。本节演示如何使用递归树来解决递归、生成良好的猜测并获得递归的直觉。

一个例子

让我们看看递归树如何为递归的上限解提供良好的猜测

$$T(n) = 3T(n/4) + \Theta(n^2) . \tag{4.13}$$

图 4.1 显示了如何推导 T .n/D 3T .n/4/C cn ² 的递归树，其中常数 c > 0 是 ，.n ²/ 项中的上限常数。图 (a) 部分显示了 T .n/，图 (b) 部分扩展为表示递归的等效树。根处的 cn ² 项表示递归顶层的成本，根的三个子树表示由递归引起的成本

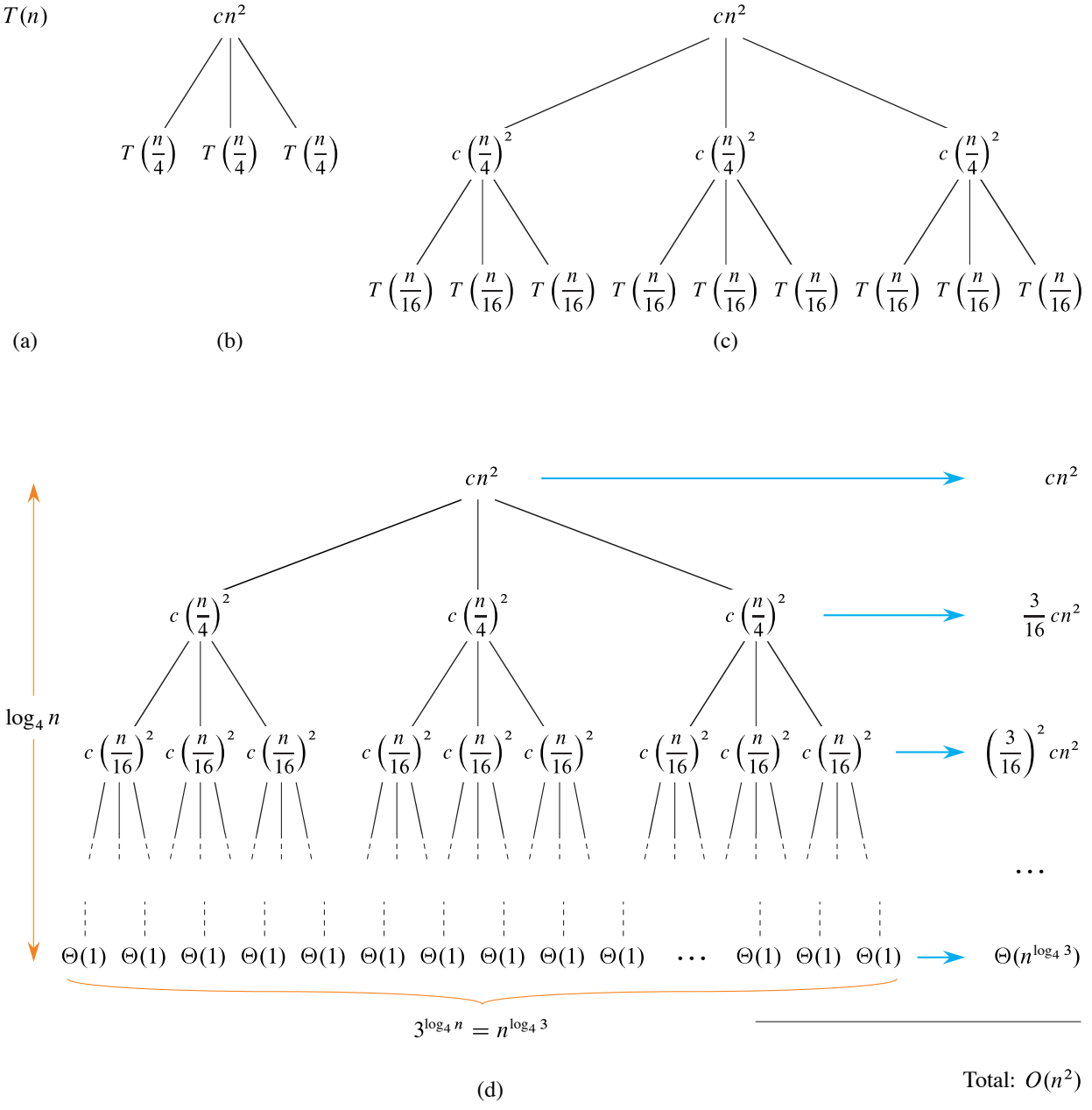


图 4.1 为递归 T .n/ D 3T .n/4/ C cn 2 构建递归树。图 (a) 显示了 T .n/，它在 (b) – (d) 中逐步扩展以形成递归树。图 (d) 中完全展开的树的高度为 log 4 n。

大小为 n/4 的子问题。第 (c) 部分显示此过程更进一步，即扩展第 (b) 部分中成本为 T .n/4/ 的每个节点。根的三个子节点的成本均为 c.n/4/ 2 。我们继续扩展树中的每个节点，将其分解为由递归确定的组成部分。

因为每次我们下降一级，子问题的大小都会减少 4 倍，所以递归最终必须在 n < n 0 的基准情况下触底。按照惯例，基准情况是 n < n 0 的 T .n/ D ，.1/，其中 n 0 > 0 是任何足够大的阈值常数，以便可以很好地定义递归。然而，为了直观起见，让我们稍微简化一下数学。我们假设 n 是 4 的精确幂，基准情况是 T .1/ D ，.1/。事实证明，这些假设不会影响渐近解。

递归树的高度是多少？深度为 i 的节点的子问题大小为 n/4^i。当我们从根节点沿树向下移动时，当 n/4^i D1 或 i D log 4 n 时，子问题大小达到 n D 1。因此，树在深度 0；1；2；:::;log 4 n 处有内部节点，在深度 log 4 n 处有叶子节点。

图 4.1 中的 (d) 部分显示了树的每一层的成本。每一层的节点数都是上一层的三倍，所以深度 i 处的节点数为 3^i。由于距离根节点每增加一层，子问题的大小就会减小 4 倍，所以深度 i 处的每个内部节点 D 0; 1; 2; : : : ; log 4 n 1 的成本为 c.n/4^i /2。相乘后，我们发现给定深度 i 处所有节点的总成本为 3 ^{i} c.n/4^i /2 D.3/16/^{i} cn^2。最底层的深度为 log 4 n，包含 3 $^{\log_4 n}$ D n $^{\log_4 3}$ 个叶子节点（使用第 66 页的公式 (3.21)）。每个叶子节点贡献 ，.1/，因此总的叶子节点成本为 ，.n $^{\log_4 3}$ /。

现在我们将各层级的成本加起来以确定整棵树的成本：

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)}\, cn^2 + \Theta(n^{\log_4 3}) \qquad \text{(by equation (A.7) on page 1142)} \\
&= \frac{16}{13}\, cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) \qquad (\Theta(n^{\log_4 3}) = O(n^{0.8}) = O(n^2)) \,.
\end{aligned}
$$

我们已经推导出原始递归式 T .n/ D O.n 2 / 的猜测。在这个例子中，cn 2 的系数形成一个递减的几何级数。根据方程 (A.7)，这些系数的总和由常数 16/13 限定。因为

根对总成本的贡献是 cn^2，根的成本决定了树的总成本。

事实上，如果 O.n ² / 确实是递归的上限（我们稍后会验证），那么它一定是紧限。为什么？第一个递归调用的成本为 .n ²/，因此 .n ²/ 一定是递归的下限。

现在让我们使用代换法来验证我们的猜测是否正确，即 T .n/ D O.n ²/ 是递归 T .n/ D 3T .n/4/C，.n ²/ 的上限。我们想证明 T .n/ d n ² 中存在某个常数 d > 0。使用与前面相同的常数 c > 0，我们得到

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

如果我们选择 d，则最后一步成立。.16/13/c 。

对于归纳法的起始条件，设 n ₀ > 0 为一个足够大的阈值常数，当 n < n ₀ 中 T .n/ D ，.1/ 时，递归式定义良好。我们可以选取足够大的 d，使 d 支配 ，所隐藏的常数，在这种情况下，d n ² ≈ d ≈ T .n/，1 = n < n ₀，完成起始条件的证明。

我们刚刚看到的代换证明涉及两个命名常数 c 和 d。我们命名了 c 并用它来表示由 ，表示法隐藏并保证存在的上界常数。我们不能任意选择 c（它是给定的），尽管对于任何这样的 c，任何常数 c′ 也足够了。我们还命名了 d，但我们可以自由地为其选择任何符合我们需要的值。在这个例子中，d 的值恰好取决于 c 的值，即 ûne，因为如果 c 是常数，则 d 也是常数。

不规则示例

让我们为另一个更不规则的例子找到渐近上界。图 4.2 显示了递归的递归树

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) . \tag{4.14}$$

此递归树是不平衡的，不同的根到叶路径具有不同的长度。在任何节点向左移动都会产生三分之一大小的子问题，向右移动会产生三分之二大小的子问题。令 n ₀ > 0 为隐式阈值常数，使得对于 0 < n < n ₀，T .n/ D ，.1/，令 c 表示被 n ¬ n ₀ 的，.n/ 项隐藏的上限常数。这里实际上有两个 n ₀ 常数4，一个用于递归中的阈值，另一个用于 ，表示法中的阈值，因此我们令 n ₀ 为两个常数中较大的一个。

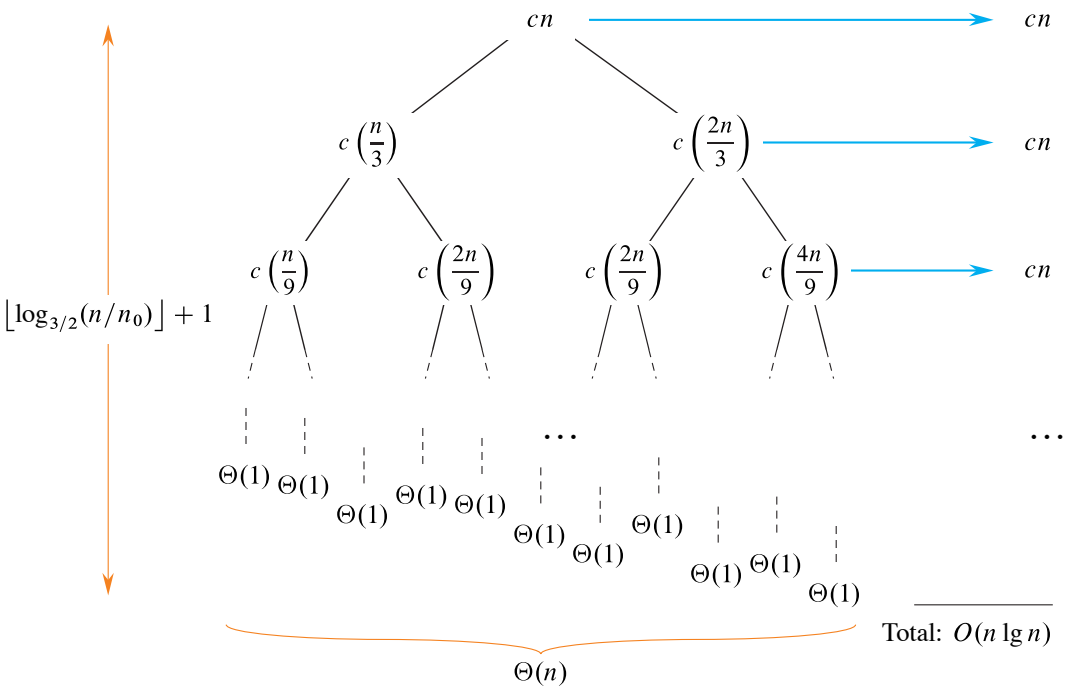


图 4.2 递归 T .n/ D T .n/3/ C T .2n/3/ C cn 的递归树。

树的高度沿着树的右边缘延伸，分别对应于大小为 n；.2/3/n；.4/9/n；: : : ；，.1/ 的子问题，其成本受 cn；c.2n/3/；c.4n/9/；: : : ；，.1/ 的限制。当 .2/3/ ^{h} n < n $_{0}$ .2/3/ $^{h-1}$ n 时，我们命中了最右边的叶子，当 h D blog $_{3/2}$ .n/n $_{0}$/c C 1 时就会发生这种情况，因为应用第 64 页公式 (3.2) 中的 üoor 界限和 x D log $_{3/2}$ .n/n $_{0}$/ 时，我们得到 .2/3/ ^{h} n D .2/3/ $^{\lfloor x \rfloor+1}$ n < .2/3/ ^{x} n D .n $_{0}$/n/n D n $_{0}$ 和 .2/3/ $^{h-1}$ n D .2/3/ $^{\lfloor x \rfloor}$ n > .2/3/ ^{x} n D .n $_{0}$/n/n D n $_{0}$。因此，树的高度为 h D 现在我们可以理解上限了。让我们暂时不处理叶子节点。将每层内部节点的成本相加，我们最多有每层 cn 乘以 ，.lg n/ 树高，所有内部节点的总成本为 O.n lg n/。

剩下要处理的是递归树的叶子，它们代表基本情况，每个成本为 ，.1/。有多少片叶子？由于递归树包含在这样一个完全二叉树中，因此很容易将它们的数量上限设为高度为 h D blog $_{3/2}$ .n/n $_{0}$/c C 1 的完全二叉树中的叶子数量。但这种方法结果给我们的界限很差。完全二叉树在根部有 1 个节点，在深度 1 处有 2 个节点，在深度 k 处通常有 2 ^{k} 个节点。由于高度为 h D blog $_{3/2}$ nc C 1，因此有

2 h D 2 $^{\lfloor \log_{3/2} n \rfloor + 1}$ # 2n $^{\log_{3/2} 2}$ 叶子在完全二叉树中，这是递归树中叶子数量的上限。由于每片叶子的成本为 ，.1/，因此分析表明，递归树中所有叶子的总成本为 O.n $^{\log_{3/2} 2}$ / D O.n $^{1.71}$ /，这是比所有内部节点的 O.n lg n/ 成本渐近更大的界限。事实上，正如我们即将看到的，这个界限并不紧。递归树中所有叶子的成本是 O.n/4，渐近于 O.n lg n/。换句话说，内部节点的成本决定了叶子的成本，反之亦然。

与其分析叶子节点，我们现在可以放弃，通过代换证明 T .n/ D ，.n lg n/。这种方法有效（参见练习 4.4-3），但了解这棵递归树有多少个叶子节点是有益的。您可能会看到叶子节点的成本大于内部节点成本的递归，如果您有分析叶子节点数量的经验，那么您的情况会更好。

为了计算出实际有多少个叶子，我们来写一个递归式 L.n/，表示 T.n/ 的递归树中的叶子数量。由于 T.n/ 中的所有叶子要么属于根的左子树，要么属于根的右子树，因此我们有

$$L(n) = \begin{cases} 1 & \text{if } n < n_0 , \\ L(n/3) + L(2n/3) & \text{if } n \geq n_0 . \end{cases} \tag{4.15}$$

这个递归式与递归式 (4.14) 类似，但它缺少 ，.n/ 项，并且包含一个显式基本情况。由于这个递归式省略了 ，.n/ 项，因此求解起来要容易得多。让我们应用代换法来证明它有解 L.n/ D O.n/。对某个常数 d > 0 使用归纳假设 L.n / d n，并假设归纳假设对所有小于 n 的值都成立，我们有

$$\begin{aligned} L(n) &= L(n/3) + L(2n/3) \\ &\leq dn/3 + 2(dn)/3 \\ &\leq dn , \end{aligned}$$

对于任何 d > 0，该定理都成立。现在我们可以选择足够大的 d 来处理 0 < n < n $_0$ 的基本情况 L.n/ D 1，对于该情况 d D 1 就足够了，从而完成了叶子上界的替代方法。（练习 4.4-2 要求您证明 L.n/ D ，.n/。）

回到 T .n/ 的递归 (4.14)，现在很明显，所有级别的叶子节点的总成本必须是 L.n/ ，.1/ D ，.n/。由于我们已经推导出内部节点成本的界限 O.n lg n/，因此递归 (4.14) 的解为 T .n/ D O.n lg n/ C ，.n/ D O.n lg n/。（练习 4.4-3 要求您证明 T .n/ D ，.n lg n/。）

使用替代法来验证用递归树获得的任何界限是明智的，特别是如果你已经做出了简化假设。但另一个

策略的共同点是使用更强大的数学，通常是以下一节中的主方法（不幸的是，它不适用于递归（4.14））或 Akra-Bazzi 方法（它适用，但需要微积分）的形式。即使你使用强大的方法，递归树也可以提高你对复杂数学背后发生的事情的直觉。

练习

4.4-1
对于以下每个递归，画出它的递归树，并猜测其解的一个好的渐近上界。然后使用替换法来验证你的答案。

a.T .n/ D T .n/2/ C n ^{3} . ***b.*** T .n/ D 4T .n/3/ C n。 ***c.*** T .n / D 4T .n/2/ C n。 ***d.*** T .n/ D 3T .n 1/ C 1。

4.4-2
利用代换法证明递归式（4.15）有渐近下界L.n/ D .n/。得出L.n/ D ，.n/。

4.4-3
利用代换法证明递归式（4.14）有解T .n/ D .n lg n/。得出T .n/ D ，.n lg n/ 。

4.4-4
使用递归树来证明对递归 T .n/ D T .˛n/CT ..1 ˛/n/C ，.n/ 的解的合理猜测，其中˛是 $0 < \alpha < 1$ 范围内的常数。

4.5 解决递归问题的主方法

主方法提供了一种“食谱”方法来解决以下形式的算法递归

$$T(n) = aT(n/b) + f(n)\,, \tag{4.16}$$

其中 a > 0 和 b > 1 是常数。我们称 f .n/ a ***driving function***，并将此一般形式的递归称为 a ***master recurrence***。要使用主方法，您需要记住三种情况，但之后您将能够非常轻松地解决许多主递归。

主递归描述了分治算法的运行时间，该算法将大小为 n 的问题划分为 a 个子问题，每个子问题的大小为 n/b < n 。该算法以递归方式解决 a 个子问题，每个子问题都需要 T .n/b/ 时间。驱动函数 f .n/ 包含递归前划分问题的成本，以及合并子问题递归解结果的成本。例如，由 Strassen 算法产生的递归是一个主递归，其中 a D 7、b D 2，驱动函数 f .n/ D ，.n 2 /。

正如我们提到的，在解决描述算法运行时间的递归时，我们经常宁愿忽略的一个技术细节是输入大小 n 必须是整数的要求。例如，我们看到归并排序的运行时间可以用递归 (2.3) 来描述，T .n/ D 2T .n/2/ C ，.n/，第 41 页。但如果 n 是奇数，我们实际上并没有两个大小恰好是一半的问题。相反，为了确保问题大小是整数，我们将一个子问题向下舍入为 bn/2c，将另一个子问题向上舍入为 dn/2e，因此真正的递归是 T .n/ D T .dn/2e C T .bn/2c/ C ，.n/ 。但是，与基于实数定义的递归 (2.3) 相比，这种上下限递归写起来更长，处理起来也更麻烦。如果没有必要，我们宁愿不担心上下限，尤其是因为这两个递归具有相同的 “ .n lg n ” 解。

主方法允许你陈述一个没有上限和下限的主递归，并隐式地推断它们。无论参数如何向上或向下舍入到最接近的整数，它提供的渐近界限都保持不变。此外，正如我们将在 4.6 节中看到的那样，如果你在实数上定义主递归，而没有隐式的上限和下限，渐近界限仍然不会改变。因此，你可以忽略主递归的上限和下限。4.7 节给出了在更一般的分治递归中忽略上限和下限的充分条件。

主定理

主方法依赖于以下定理。

Theorem4.1 (Master theorem)
令 a > 0 和 b > 1 为常数，令 f .n/ 为驱动函数，该函数在所有足够大的实数上定义且为非负。定义 n 2 N 上的递归 T .n/ 为

$$T(n) = aT(n/b) + f(n) , \tag{4.17}$$

其中 aT .n/b/ 实际上意味着 a′T .bn/bc/ C a″T .dn/be/，其中某些常数 a′ 0 和 a″ 0 满足 a D a′ C a″。那么 T .n/ 的渐近行为可以描述如下：

1. 如果存在一个常数 > 0 使得 $f(n) = O(n^{\log_b a - \epsilon})$，则 $T(n) = \Theta(n^{\log_b a})$。

2. 若存在常数 $k \geq 0$ 使得 $f(n) = \Theta(n^{\log_b a} \lg^k n)$，则 $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$。

3. 如果存在一个常数 > 0 使得 $f(n) = \Omega(n^{\log_b a + \epsilon})$，且如果 $f(n)$ 另外满足 ***regularity condition*** $af(n/b) \leq cf(n)$ (其中 $c < 1$ 为常数，且 n 足够大)，则 $T(n) = \Theta(f(n))$。 ■

在将主定理应用到一些例子之前，让我们花几分钟来大致了解一下它的含义。函数 $n^{\log_b a}$ 称为 ***water- shed function***。在这三种情况下，我们都将驱动函数 $f(n)$ 与分水岭函数 $n^{\log_b a}$ 进行比较。直观地说，如果分水岭函数的渐近增长速度快于驱动函数，则适用情况 1。如果两个函数的渐近速率几乎相同，则适用情况 2。情况 3 是情况 1 的 " 对立面"，其中驱动函数的渐近增长速度快于分水岭函数。但技术细节很重要。

在情况 1 中，分水岭函数不仅必须比驱动函数渐近增长得更快，而且它必须以更快的速度增长 *polynomially*。也就是说，分水岭函数 $n^{\log_b a}$ 必须渐近地比驱动函数 $f(n)$ 大至少 $\Omega(n^{\epsilon})$ 的倍数，其中某个常数 > 0。主定理则指出，解决方案是 $T(n) = \Theta(n^{\log_b a})$。在这种情况下，如果我们查看递归树的递归，则每级成本从根到叶至少呈几何级数增长，并且叶的总成本决定了内部节点的总成本。

在情况 2 中，分水岭函数和驱动函数以几乎相同的渐近速率增长。但更具体地说，驱动函数的增长速度比分水岭函数快 $\Omega(\lg^k n)$，其中 $k \geq 0$。主定理指出，我们在 $f(n)$ 上增加一个额外的 $\lg n$ 因子，得到解 $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$。在这种情况下，递归树的每一级成本大约相同 $\Theta(n^{\log_b a} \lg^k n/4)$，并且有 $\Omega(\lg n)$ 级。实际上，情况 2 最常见的情况是 $k \geq 0$，在这种情况下分水岭函数和驱动函数具有相同的渐近增长，解为 $T(n) = \Theta(n^{\log_b a} \lg n)$。

情况 3 与情况 1 类似。驱动函数不仅必须比分水岭函数渐近增长得更快，而且必须以更快的速度增长 *polynomially*。也就是说，驱动函数 $f(n)$ 必须渐近地比分水岭函数 $n^{\log_b a}$ 大至少 1 倍，其中某个常数 > 0。此外，驱动函数必须满足规律性条件，即 $af(n/b) \leq cf(n)$。在应用情况 3 时，您可能遇到的大多数多项式有界函数都满足此条件。规律性条件可能不满足

如果驱动函数在局部区域增长缓慢，但总体增长相对较快。（练习 4.5-5 给出了此类函数的示例。）对于情况 3，主定理指出解决方案是 T .n/ D ，.f .n/ /。如果我们查看递归树，则每级成本至少从根到叶呈几何级下降，并且根成本主导所有其他节点的成本。

值得再次考虑的是，无论是案例 1 还是案例 3，分水岭函数和驱动函数之间都必须存在多项式分离。分离不需要很大，但必须存在，并且必须呈多项式增长。例如，对于递归 T .n/ D 4T .n/2/ C n $^{1.99}$（（当然，这不是您在分析算法时可能看到的递归））， 分水岭函数为 n $^{\log_b a}$ D n ^{2}。因此，驱动函数 f . n/ D n $^{1.99}$ 多项式地小了 n $^{0.01}$ 的倍数。因此，案例 1 适用， D 0:01。

使用主方法

要使用主方法，您需要确定主定理的哪种情况（如果有）适用并写下答案。

作为第一个例子，考虑递归式 T .n/ D 9T .n/3/ C n。对于此递归式，我们有 a D 9 和 b D 3，这意味着 n $^{\log_b a}$ D n $^{\log_3 9}$ D ′ .n ^{2})。由于对于任何常数 1，f .n/ D n D O.n $^{2-\epsilon}$/，我们可以应用主定理的第 1 种情况来得出结论，解为 T .n/ D ′ .n ^{2}/。

现在考虑递归式 T .n/ D T .2n/3/ C 1，其中 a D 1 和 b D 3/2，这意味着分水岭函数为 n $^{\log_b a}$ D n $^{\log_{3/2} 1}$ D n ^{0} D 1。情况 2 适用，因为 f .n/ D 1 D ，.n $^{\log_b a}$ lg ^{0} n/ D ，.1/。递归式的解为 T .n/ D ，.lg n/。

对于递归 T .n/ D 3T .n/4/ C n lg n，我们有 a D 3 和 b D 4，这意味着 n $^{\log_b a}$ D n $^{\log_4 3}$ D O.n $^{0.793}$/。由于 f .n/ D n lg n D .n $^{\log_4 3+\epsilon}$/，其中 可以大到大约 0:2，只要规则性条件对 f .n/ 成立，情况 3 就适用。确实如此，因为对于足够大的 n，我们有 f .n/b/ D 3.n/4/ lg.n/4/ .3/4/n lg n D cf .n/，其中 c D 3 /4。根据情况 3，递归的解是 T .n/ D ，.n lg n/。

接下来，我们看一下递归式 T .n/ D 2T .n/2/ C n lg n，其中有 a D 2、b D 2 和 n $^{\log_b a}$ D n $^{\log_2 2}$ D n。情况 2 适用，因为 f .n/ D n lg n D ，.n $^{\log_b a}$ lg ^{1} n/。我们得出结论，解决方案是 T .n/ D ，.n lg ^{2} n/。

我们可以使用主方法来解决第 2.3.2、4.1 和 4.2 节中看到的递归。

递归（2.3），T .n/ D 2T .n/2/ C ，.n/，第 41 页，表征了归并排序的运行时间。由于 a D 2 和 b D 2，分水岭函数为 n $^{\log_b a}$ D n $^{\log_2 2}$ D n。情况 2 适用，因为 f .n/ D ，.n/，解为 T .n/ D ，.n lg n/。

递归 (4.9)，T .n/ D 8T .n/2/ C ，.1/，在第 84 页，描述了矩阵乘法的简单递归算法的运行时间。我们有 a D 8 和 b D 2，这意味着分水岭函数为 $n^{\log_b a}$ D $n^{\log_2 8}$ D n^3。由于 n^3 多项式大于驱动函数 f .n/ D ，.1/4（实际上，对于任何正 < 34，我们有 f .n/ D O.$n^{3-\epsilon}$/（情况 1 适用）。我们得出结论 T .n/ D ，.n^3/。

最后，第 87 页上的递归 (4.10)，T .n/ D 7T .n/2/ C ，.n^2/，来自对 Strassen 矩阵乘法算法的分析。对于这个递归，我们有 a D 7 和 b D 2，分水岭函数为 $n^{\log_b a}$ D $n^{\lg 7}$。观察到 lg 7 D 2:807355 : : :，我们可以让 D 0:8 并限制驱动函数 f .n/ D ，.n^2/ D O.$n^{\lg 7-\epsilon}$/。情况 1 适用于解 T .n/ D ，.$n^{\lg 7}$/。

当主方法不适用时

在某些情况下，您无法使用主定理。例如，分水岭函数和驱动函数可能无法进行渐近比较。对于无限数量的 n 值，我们可能有 f .n/ $n^{\log_b a}$，但对于无限数量的不同 n 值，我们也可能有 f .n/ $n^{\log_b a}$。然而，实际上，在算法研究中出现的大多数驱动函数都可以与分水岭函数进行有意义的比较。如果您遇到主递归，而情况并非如此，则必须求助于替代或其他方法。

即使可以比较驱动函数和分水岭函数的相对增长，主定理也不能涵盖所有可能性。当 f .n/ D o.$n^{\log_b a}$/ 时，情况 1 和情况 2 之间存在差距，但分水岭函数的增长速度并不比驱动函数快多项式。同样，当 f .n/ D !.$n^{\log_b a}$/ 时，情况 2 和情况 3 之间存在差距，驱动函数的增长速度比分水岭函数快多对数倍，但并不比分水岭函数快多项式倍。如果驱动函数落入其中一个差距，或者情况 3 中的规律性条件不成立，则需要使用主方法以外的其他方法来解决递归问题。

作为驱动函数陷入间隙的一个例子，考虑递归 T .n/ D 2T .n/2/ C n/ lg n。由于 a D 2 和 b D 2，分水岭函数为 $n^{\log_b a}$ D $n^{\log_2 2}$ D n^1 D n。驱动函数为 n / lg n D o.n/，这意味着它比分水岭函数 n 的渐近增长速度更慢。但 n/ lg n 的增长速度仅比 n 慢 *logarithmically*，而不是 *polynomially*。更准确地说，第 67 页的公式 (3.24) 表明，对于任何常数 > 0，lg n D o.n^{ϵ}/，这意味着 1/ lg n D !.$n^{-\epsilon}$/ 和 n/ lg n D !.$n^{1-\epsilon}$/ D !.$n^{\log_b a-\epsilon}$/。因此，不存在常数 > 0 使得 n/ lg n D O.$n^{\log_b a-\epsilon}$/，这是情况 1 适用的必要条件。情况 2 也不适用，因为 n/ lg n D ，.$n^{\log_b a} \lg^k$ n/，其中 k D 1，但 k 必须为非负数，情况 2 才适用。

要解决这种递归问题，必须使用其他方法，如代换法（第 4.3 节）或 Akra -Bazzi 方法（第 4.7 节）。（练习 4.6-3 要求你证明答案是“ .n lg lg n/ ”。）虽然主定理不处理这种特定的递归问题，但它确实处理了实践中出现的绝大多数递归问题。

练习

4.5-1
使用主方法为以下递归提供严格的渐近界限。

a.T .n/ D 2T .n/4/ C 1. ***b.*** T .n/ D 2T .n/4/ C p n。 ***c.*** T .n/ D 2T .n/ 4/ C p n lg ^{2} n。 ***d.*** T .n/ D 2T .n/ 4/ C n。 ***e.*** T .n/ D 2T .n/4/ C n ^{2} 。

4.5-2
Caesar 教授想要开发一种比 Strassen 算法渐进更快的矩阵乘法算法。他的算法将使用分治法，将每个矩阵划分为 n/4 n/4 个子矩阵，划分和合并步骤总共需要 ，.n ^{2} / 时间。假设教授的算法创建了一个大小为 n/4 的递归子问题。对于 a 的最大整数值，他的算法可能比 Strassen 算法渐进更快地运行，那么这个最大的整数值是多少？

4.5-3
使用主方法证明二分查找递归式 T .n/ D T .n/2/ C ，.1/ 的解是 T .n/ D ，.lg n/ 。（有关二分查找的描述，请参阅练习 2.3-6。）

4.5-4
考虑函数 f .n/ D lg n。论证虽然 f .n/2/ < f .n/ ，但正则性条件 af .n/b/ cf .n / (其中 a D 1 和 b D 2) 对任何常数 c < 1 均不成立。进一步论证对于任何 > 0 ，情况 3 中的条件 f .n/ D .n $^{\log_b a+\epsilon}$ / 均不成立。

4.5-5证明对于适当的常数 a、b 和 ，函数 f .n/ D 2 $^{\lceil \lg n \rceil}$ 满足主定理案例 3 中的所有条件（除了正则性条件）。

?4.6 连续主定理的证明

证明主定理（定理 4.1）的全面性，尤其是处理棘手的上下限和上限技术问题，超出了本书的范围。然而，本节陈述并证明了主定理的一个变体，称为***continuous master theorem*** [1]，其中主递归（4.17）在足够大的正实数上定义。此版本的证明不受上下限和上限的影响，包含理解主递归行为所需的主要思想。第 4.7 节更详细地讨论了分治递归中的上下限和上限，并给出了它们不影响渐近解的充分条件。

当然，由于您无需理解主定理的证明即可应用主方法，因此您可以选择跳过本节。但是，如果您希望学习超出本教科书范围的更高级算法，您可能会希望更好地理解底层数学，这是连续主定理的证明所提供的。

尽管我们通常假设递归是算法性的，不需要明确陈述基本情况，但对于证明实践合理性的证明，我们必须更加谨慎。本节中的引理和定理明确陈述了基本情况，因为归纳证明需要数学基础。在数学世界中，非常小心地证明定理是很常见的，这些定理可以证明实践中更随意的行为是合理的。

连续主定理的证明涉及两个引理。引理 4.2 使用稍微简化的主递归，其阈值常数为 n_0 D 1，而不是未说明的基本情况所暗示的更一般的 $n_0 > 0$ 阈值常数。引理使用递归树将简化主递归的解简化为求和的解。引理 4.3 随后为求和提供了渐近界限，反映了主定理的三个情况。最后，连续主定理本身（定理 4.4）给出了主递归的渐近界限，同时推广到未说明的基本情况所暗示的任意阈值常数 $n_0 > 0$。

[1] This terminology does not mean that either $T(n)$ or $f(n)$ need be continuous, only that the domain of $T(n)$ is the real numbers, as opposed to integers.

一些证明使用第 723-73 页问题 3-5 中描述的属性来组合和简化复杂的渐近表达式。尽管问题 3-5 仅涉及 ，符号，但其中列举的属性也可以扩展到 O 符号和 符号。

这是第一个引理。

Lemma4.2
令 a>0 和 b>1 为常数，令 f.n/ 为定义在实数 n 1 上的函数。则递归

$$T(n) = \begin{cases} \Theta(1) & \text{if } 0 \le n < 1 , \\ aT(n/b) + f(n) & \text{if } n \ge 1 \end{cases}$$

有解决方案

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) . \tag{4.18}$$

Proof考虑图 4.3 中的递归树。我们先看看它的内部节点。树的根节点的成本为 f.n/，它有 a 个子节点，每个子节点的成本为 f.n/b/。（将 a 视为整数很方便，尤其是在可视化递归树时，但数学并不要求这样做。）这些子节点中的每一个都有 a 个子节点，使得深度为 2 的节点有 ^{2} 个，并且每个 a 个子节点的成本为 f.n/b ^{2}/。通常，深度为 j 的节点有 a ^{j} 个，每个节点的成本为 f.n/b ^{j}/。

现在，让我们继续了解叶子节点。树向下生长，直到 n/b^j 小于 1。因此，树的高度为 blog $_b$nc C 1，因为 n/b$^{\lfloor \log_b n \rfloor}$ ≥ n/b$^{\log_b n}$ D 1 且 n/b$^{\lfloor \log_b n \rfloor + 1}$ < ≥ n/b$^{\log_b n}$ D 1。正如我们所观察到的，深度 j 处的节点数为 a ^{j}，所有叶子节点的深度均为 blog $_b$ nc C 1，因此树包含 a $^{\lfloor \log_b n \rfloor + 1}$ 个叶子节点。使用第 66 页上的恒等式 (3.21)，我们有 a $^{\lfloor \log_b n \rfloor + 1}$ = a $^{\log_b n + 1}$ D an $^{\log_b a}$ D O.n $^{\log_b a}$/，因为 a 是常数，并且 a $^{\lfloor \log_b n \rfloor + 1}$ ≈ a $^{\log_b n}$ D n $^{\log_b a}$ D .n $^{\log_b a}$/。因此，叶子总数渐近为 ，.n $^{\log_b a}$/4，即分水岭函数。

现在我们可以通过求和树中每个深度节点的成本来推导出方程 (4.18)，如图所示。方程中的第一项是叶子节点的总成本。由于每个叶子节点的深度为 blog $_b$nc C 1 和 n/b $^{\lfloor \log_b n \rfloor + 1}$ < 1，递归的基本情况给出叶子节点的成本：T . n/b $^{\lfloor \log_b n \rfloor + 1}$/ D .1/。因此，根据问题 3-5(d)，所有 .n $^{\log_b a}$/ 叶子节点的成本为 .n $^{\log_b a}$/ - .1/ D .n $^{\log_b a}$/。方程 (4.18) 中的第二项是内部节点的成本，在底层的分治算法中，它表示将问题划分为子问题的成本和

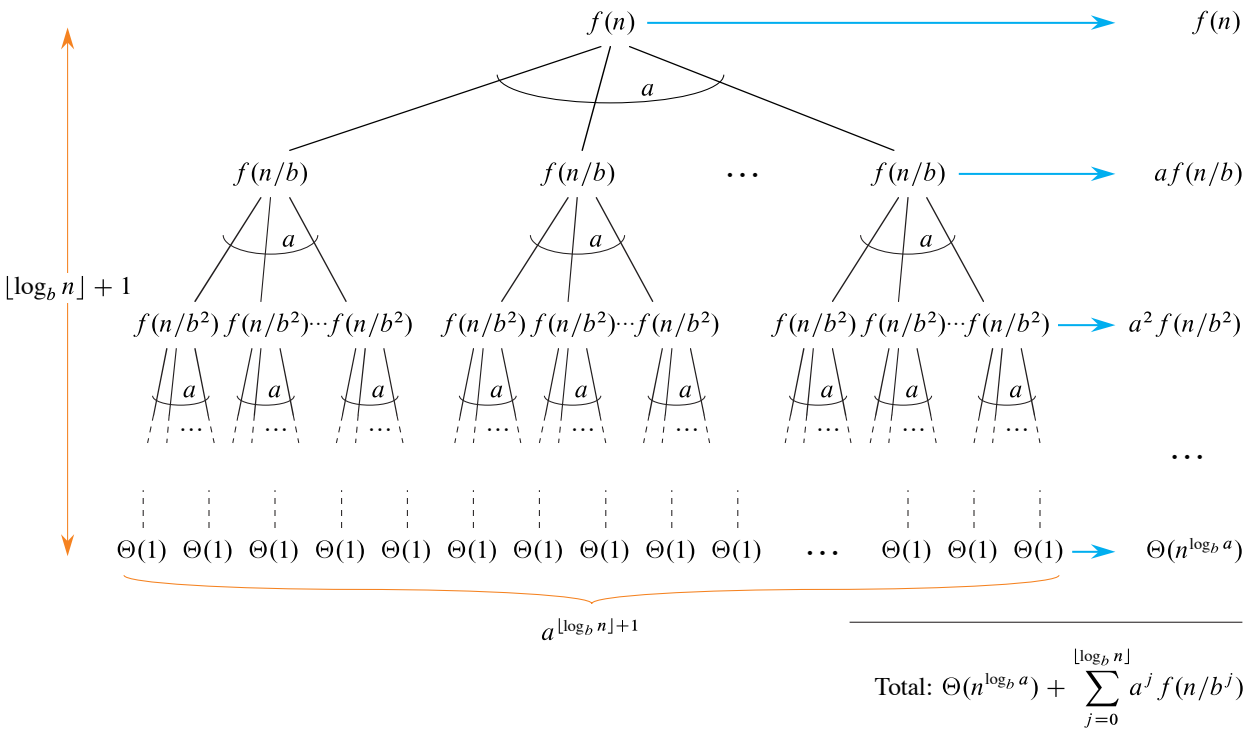


图 4.3 由 T .n/ D aT .n/b/ C f .n/ 生成的递归树。该树是一个完全 a 元树，有 a $^{\lfloor \log_b n \rfloor + 1}$ 个叶子节点，高度为 blog $_b$ nc C 1。每个深度节点的成本显示在右侧，其总和在公式 (4.18) 中给出。

然后重新组合子问题。由于深度 j 处所有内部节点的成本为 a ^{j}f .n/b ^{j}/，因此所有内部节点的总成本为

$$\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) \,.$$

■

我们将看到，主定理的三种情况取决于递归树各层间总成本的分布：

情况 1：成本从根到叶呈几何级数增加，每一级都以一个常数因子增长。

情况 2：成本取决于定理中的 k 值。当 k D 0 时，每层的成本相等；当 k D 1 时，成本从根到叶呈线性增长；当 k D 2 时，增长呈二次函数；一般而言，成本随 k 呈多项式增长。

情况 3：成本从根到叶呈几何级数减少，每一级都按一个常数因子减少。

等式 (4.18) 中的求和描述了底层分治算法中划分和合并步骤的成本。下一个引理提供了求和增长的渐近界限。

Lemma4.3
令 a>0 和 b>1 为常数，令 f (n) 为定义在实数 n 1 上的函数。则该函数的渐近行为

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) , \tag{4.19}$$

定义为n≥1，可具有如下特征：

1. 如果存在常数 $\epsilon > 0$ 使得 f .n/ D O.n $^{\log_b a-\epsilon}$/，则 g.n/ D O.n $^{\log_b a}$/。

2. 若存在常数k ≈ 0，使得f .n/ D ，.n $^{\log_b a}$ lg ^{k} n/，则g.n/ D ，.n $^{\log_b a}$ lg $^{k+1}$ n/。

3. 如果存在一个常数 c 在 0<c<1 范围内，使得对所有 n≥1，0<af.n/b/···cf.n/，则 g.n/D，.f.n//。

Proof对于情况 1，我们有 f .n/ D O.n $^{\log_b a-\epsilon}$/，这意味着 f .n/b ^{j}/ D O..n/b ^{j}/ $^{\log_b a-\epsilon}$/。代入方程 (4.19) 可得

$$\begin{aligned}
g(n) &= \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j O\left(\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) \\
&= O\left(\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) && \text{(by Problem 3-5(c), repeatedly)} \\
&= O\left(n^{\log_b a-\epsilon} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j\right) \\
&= O\left(n^{\log_b a-\epsilon} \sum_{j=0}^{\lfloor \log_b n \rfloor} (b^\epsilon)^j\right) && \text{(by equation (3.17) on page 66)} \\
&= O\left(n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon(\lfloor \log_b n \rfloor+1)}-1}{b^\epsilon-1}\right)\right) && \text{(by equation (A.6) on page 1142)} ,
\end{aligned}$$

最后一个级数是几何级数。由于 b 和 ϵ 是常数，因此 b $^{\epsilon}$ 1 分母不影响 g.n/ 的渐近增长，并且 1 中的

分子。由于 b $^{\epsilon(\lfloor \log_b n \rfloor + 1)}$ = .b $^{\log_b n + 1}$ / $^{\epsilon}$ D b $^{\epsilon}$ n $^{\epsilon}$ D O.n $^{\epsilon}$/，我们得到 g.n/ D O.n $^{\log_b a - \epsilon}$ + O.n $^{\epsilon}$// D O.n $^{\log_b a}$/，从而证明了情况 1。

案例 2 假设 f .n/ D ，.n $^{\log_b a}$ lg ^{k} n/，由此我们可以得出结论 f .n/b ^{j}/ D ，..n /b ^{j}/ $^{\log_b a}$ lg ^{k} .n/b ^{j}//。代入方程 (4.19) 并反复应用问题 3-5(c) 可得出

$$
\begin{aligned}
g(n) &= \Theta\left(\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j \left(\frac{n}{b^j}\right)^{\log_b a} \lg^k\left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \frac{a^j}{b^{j \log_b a}} \lg^k\left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \lg^k\left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{\log_b (n/b^j)}{\log_b 2}\right)^k\right) && \text{(by equation (3.19) on page 66)} \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{\log_b n - j}{\log_b 2}\right)^k\right) && \text{(by equations (3.17), (3.18), and (3.20))} \\
&= \Theta\left(\frac{n^{\log_b a}}{\log_b^k 2} \sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k\right) && (b > 1 \text{ and } k \text{ are constants}) .
\end{aligned}
$$

' -符号中的和可以从上面限定如下：

$$
\begin{aligned}
\sum_{j=0}^{\text{blog}_b n\rfloor} (\log_b n - j)^k &\le \sum_{j=0}^{\lfloor \log_b n \rfloor} (\lfloor \log_b n \rfloor + 1 - j)^k \\
&= \sum_{j=1}^{\lfloor \log_b n \rfloor + 1} j^k && \text{(reindexing—pages 1143–1144)} \\
&= O((\lfloor \log_b n \rfloor + 1)^{k+1}) && \text{(by Exercise A.1-5 on page 1144)} \\
&= O(\log_b^{k+1} n) && \text{(by Exercise 3.3-3 on page 70) .}
\end{aligned}
$$

练习 4.6-1 要求你证明，同样，求和结果可以从下方被 .log $^{k+1}_{b}$ n/ 所限制。由于我们有严格的上限和下限，所以求和结果为 .log $^{k+1}_{b}$ n/，由此我们可以得出结论：g.n/ D ã n $^{\log_b a}$ log $^{k+1}_{b}$ n ä，从而完成案例 2 的证明。

对于情况 3，观察到 f.n/ 出现在 g.n/ 的定义 (4.19) 中（当 j D 0 时），并且 g.n/ 的所有项均为正。因此，我们必定有 g.n/ D .f.n//，只需证明 g.n/ D O.f .n//。对不等式 af.n/b/ cf.n/ 进行 j 次迭代可得出 a ^{j} f.n/b ^{j} / c ^{j} f.n/。代入公式 (4.19)，我们得到

$$
\begin{aligned}
g(n) &= \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) \\
&\le \sum_{j=0}^{\lfloor \log_b n \rfloor} c^j f(n) \\
&\le f(n) \sum_{j=0}^{\infty} c^j \\
&= f(n) \left(\frac{1}{1-c} \right) \qquad \text{(by equation (A.7) on page 1142 since } |c| < 1) \\
&= O(f(n)) .
\end{aligned}
$$

因此，我们可以得出结论：g.n/ D ，.f .n//。通过证明情况 3，引理的整个证明就完成了。 ■

现在我们可以陈述并证明连续主定理。

Theorem4.4 (Continuous master theorem)
令 a > 0 和 b > 1 为常数，令 f .n/ 为驱动函数，该函数在所有足够大的实数上定义且为非负。在正实数上定义算法递归 T .n/ 如下

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

然后，T .n/ 的渐近行为可以表征如下：
1. 如果存在一个常数 > 0 使得 f .n/ D O.n $^{\log_b a - \epsilon}$ /，则 T .n/ D ，.n $^{\log_b a}$ /。

2. 若存在常数k ≈ 0 使得f .n/ D ，.n $^{\log_b a}$ lg ^{k} n/，则T .n/ D ，.n $^{\log_b a}$ lg $^{k+1}$ n/ 。
3. 如果存在一个常数 > 0 使得 f .n/ D .n $^{\log_b a + \epsilon}$ /，且如果 f .n/ 另外满足正则性条件 af .n/b/ cf .n/ (其中 c 为常数 c < 1 且 n 足够大)，则 T .n/ D ，.f .n//。

Proof这个想法是应用引理 4.3 来限制引理 4.2 中的和 (4.18)。但我们必须使用 0 < n < 1 的基准情况来解释引理 4.2,

而该定理使用 0 < n < n 0 的隐式基例，其中 n 0 > 0 是任意阈值常数。由于递归是算法性的，我们可以假设 f .n/ 是针对 n 定义的

对于 n > 0，我们定义两个辅助函数 T 0.n/ D T .n 0n/ 和 f 0.n/ D f .n 0n/。我们有

$$\begin{aligned} T'(n) &= T(n_0 n) \\ &= \begin{cases} \Theta(1) & \text{if } n_0 n < n_0 , \\ aT(n_0 n/b) + f(n_0 n) & \text{if } n_0 n \ge n_0 \end{cases} \\ &= \begin{cases} \Theta(1) & \text{if } n < 1 , \\ aT'(n/b) + f'(n) & \text{if } n \ge 1 . \end{cases} \end{aligned} \tag{4.20}$$

我们得到了 T 0.n/ 的递归，它满足引理 4.2 的条件，根据该引理，解为

$$T'(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f'(n/b^j) . \tag{4.21}$$

为了解决 T 0.n/，我们首先需要限制 f 0.n/。让我们检查一下定理中的具体情况。

情况 1 的条件是 f .n/ D O.n log b a /，其中某个常数 > 0。我们有

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= O((n_0 n)^{\log_b a - \epsilon}) \\ &= O(n^{\log_b a - \epsilon}) , \end{aligned}$$

因为 a、b、n 0 和 都是常数。函数 f 0.n/ 满足引理 4.3 中情况 1 的条件，引理 4.2 中方程 (4.18) 中的求和结果为 O.n log b a /。因为 a、b 和 n 0 都是常数，所以我们有

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + O((n/n_0)^{\log_b a}) \\ &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}) \qquad \text{(by Problem 3-5(b))} , \end{aligned}$$

从而完成了定理的案例1。

情况 2 的条件是 f .n/ D ，.n log b a lg k n/，其中 k 为常数 ¬ 0。我们有

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= \Theta((n_0 n)^{\log_b a} \lg^k (n_0 n)) \\ &= \Theta(n^{\log_b a} \lg^k n) \qquad \text{(by eliminating the constant terms)} . \end{aligned}$$

与情况 1 的证明类似，函数 f ′.n/ 满足引理 4.3 中情况 2 的条件。因此，引理 4.2 中方程 (4.18) 中的求和为 ¸.n $^{\log_b a}$ lg $^{k+1}$ n/，这意味着

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + \Theta((n/n_0)^{\log_b a} \lg^{k+1}(n/n_0)) \\ &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg^{k+1} n) \\ &= \Theta(n^{\log_b a} \lg^{k+1} n) \qquad \text{(by Problem 3-5(c))} , \end{aligned}$$

这证明了定理的第二个情况。

最后，案例 3 的条件为，对于某个常数 > 0，f .n/ D .n $^{\log_b a+\epsilon}$/，并且 f .n/ 另外满足正则性条件 af .n/b/ #cf .n/，对于所有 n ¬ n $_0$ 和某些常数 c < 1 和 n $_0$ > 1。案例 3 的第一部分与案例 1 类似：

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= \Omega((n_0 n)^{\log_b a+\epsilon}) \\ &= \Omega(n^{\log_b a+\epsilon}) . \end{aligned}$$

利用 f′·n的定义，以及对所有 n ? 1 ， n_0 n≥n_0 的事实，对于 n ? 1 ，我们有

$$\begin{aligned} af'(n/b) &= af(n_0 n/b) \\ &\le cf(n_0 n) \\ &= cf'(n) . \end{aligned}$$

因此 f ′.n/ 满足引理 4.3 中情况 3 的要求，而引理 4.2 中方程 (4.18) 中的求和结果为 ，.f ′.n//，得出

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + \Theta(f'(n/n_0)) \\ &= \Theta(f'(n/n_0)) \\ &= \Theta(f(n)) , \end{aligned}$$

这完成了定理第三个案例的证明，从而完成了整个定理。 ■

练习

4.6-1证明 P $_{j=0}^{\lfloor \log_b n \rfloor}$ .log $_b$n j / ^{k} D .log $_b^{k+1}$ n/

。

? ***4.6-2***

证明主定理的案例 3 被夸大了（这也是为什么引理 4.3 的案例 3 不要求 f .n/ D .n $^{\log_b a+\epsilon}$/），因为

规律性条件 af .n/b/ cf .n/ 对于某个常数 c < 1，意味着存在一个常数 > 0 使得 f .n/ D .n $^{\log_b a+\epsilon}$/。

?**4.6-3**

对于 f .n/ D ，.n $^{\log_b a}$/ lg n/，证明方程 (4.19) 中的求和有解 g.n/ D ，.n $^{\log_b a}$ lg lg n/。得出结论：以 f .n/ 为驱动函数的主递归 T .n/ 有解 T .n/ D ，.n $^{\log_b a}$ lg lg n/。

? 4.7 Akra-Bazzi 递归

本节概述了与分治递归相关的两个高级主题。第一部分涉及使用下限和上限时出现的技术问题，第二部分讨论了 Akra-Bazzi 方法，该方法涉及一些微积分，用于解决复杂的分治递归。

具体来说，我们将研究 M. Akra 和 L. Bazzi [13] 最初研究的算法分治递归类。这些 ***Akra-Bazzi*** 递归采用以下形式

$$T(n) = f(n) + \sum_{i=1}^{k} a_i T(n/b_i) , \tag{4.22}$$

其中 k 为正整数；所有常数 a $_1$ ；a $_2$ ；: : : ；a $_k$ 2 R 都严格为正数；所有常数 b $_1$ ；b $_2$ ；: : : ；b $_k$ 2 R 都严格大于 1；驱动函数 f.n/ 在足够大的非负实数上定义，其本身也是非负的。

Akra-Bazzi 递归概括了主定理所针对的递归类。主递归表征了将问题分解为大小相等的子问题（模下限和上限）的分治算法的运行时间，而 Akra-Bazzi 递归可以描述将问题分解为不同大小的子问题的分治算法的运行时间。然而，主定理允许您忽略下限和上限，但用于解决 Akra-Bazzi 递归的 Akra-Bazzi 方法需要额外的要求来处理下限和上限。

但在深入研究 Akra-Bazzi 方法本身之前，让我们先了解一下在 Akra-Bazzi 递归中忽略上限和下限的局限性。如您所知，算法通常处理整数大小的输入。然而，对于实数，递归的数学运算通常比整数更容易，因为在整数中，我们必须处理上限和下限以确保术语定义正确。两者之间的差异似乎并不大4尤其是因为递归通常都是如此4但为了在数学上正确，我们必须小心谨慎地使用我们的

假设。由于我们的最终目标是理解算法，而不是数学极端情况的变化，我们希望既随意又严谨。我们如何才能随意对待最低限度和最高限度，同时又能确保严谨性？

从数学角度来看，处理上限和下限的困难在于，有些驱动函数可能非常非常奇怪。因此，在 Akra-Bazzi 递归中，通常不能忽略上限和下限。幸运的是，我们在算法研究中遇到的大多数驱动函数都表现良好，上限和下限不会造成影响。

多项式增长条件

如果方程 (4.22) 中的驱动函数 f .n/ 在以下意义上表现良好，那么可以放弃地板和天花板。

> 如果存在一个常数 y n > 0 满足以下条件，则定义在所有足够大的正实数上的函数 f .n/ 满足 ***polynomial-growth condition***：对于每个常数 1，都存在一个常数 d > 1（取决于 ），使得对于所有 1 和 n y n，f .n//d f . n/ df .n/。

这个定义可能是本教科书中最难理解的定义之一。首先，它表示 f .n/ 满足属性 f ..n// D .f .n//，尽管多项式增长条件实际上更强一些（参见练习 4.7-4）。该定义还意味着 f .n/ 是渐近正的（参见练习 4.7-3）。

满足多项式增长条件的函数示例包括任何形式为 f .n/ D ，.n $^{\alpha}$ lg $^{\beta}$ n lg lg $^{\gamma}$ n/ 的函数，其中 α β和 为常数。本书中使用的大多数多项式有界函数都满足该条件。指数函数和超指数函数不满足该条件（例如，参见练习 4.7-2），也存在不满足该条件的多项式有界函数。

“nice” 循环中的地板和天花板

当 Akra-Bazzi 递归中的驱动函数满足多项式增长条件时，下限和上限不会改变解的渐近行为。以下定理（未提供证明）形式化了这一概念。

Theorem4.5

令 T.n/ 为一个定义在非负实数上的函数，满足递归性 (4.22)，其中 f.n/ 满足多项式增长条件。令 T′ .n/ 为另一个定义在自然数上的函数，也满足递归性 (4.22),

除了每个 T .n/b $_i$/ 被替换为 T .dn/b $_i$e/ 或 T .bn/b $_i$c/ 外，还有 T ′.n/ D ，.T . n//。 ■

下限和上限表示对递归中的参数的较小扰动。根据第 64 页的不等式 (3.2) ，它们最多会扰动一个参数 1。但可以容忍更大的扰动。只要递归 (4.22) 中的驱动函数 f .n/ 满足多项式增长条件，就可以证明，将任何项 T .n/b $_i$/ 替换为 T .n/b $_i$ C h $_i$.n//，其中 jh $_i$.n/j D O.n/ lg $^{1+\epsilon}$ n/，其中 jh $_i$.n/j D O.n/ lg $^{1+\epsilon}$ n/，对于某个常数 > 0 和足够大的 n，渐近解不受影响。因此，分治算法中的除法步骤可以适度粗糙，而不会影响其运行时间递归的解。

Akra-Bazzi 方法

毫不奇怪，Akra-Bazzi 方法是为解决 Akra-Bazzi 递归 (4.22) 而开发的，根据定理 4.5，该方法适用于存在最低限度和最高限度甚至更大扰动的情况，正如刚才讨论的那样。该方法首先涉及确定唯一实数 p，使得 P $_{i=1}^{k}$ a $_i$/b $_i^{p}$ D 1 。这样的 p 总是存在的，因为当 p ! 1 时，总和趋向于 1；随着 p 的增加，总和会减少；而当 p ! 1 时，总和会趋向于 0。然后，Akra-Bazzi 方法给出递归的解，如下所示

$$T(n) = \Theta\left(n^p\left(1 + \int_1^n \frac{f(x)}{x^{p+1}}\,dx\right)\right) . \tag{4.23}$$

举个例子，考虑一下递归

$$T(n) = T(n/5) + T(7n/10) + n . \tag{4.24}$$

当我们研究从一组 n 个数字中选择第 i 个最小元素的算法时，我们将在第 240 页看到类似的递归 (9.1)。此递归具有方程 (4.22) 的形式，其中 a $_1$ D a $_2$ D 1，b $_1$ D 5，b $_2$ D 10/7，且 f .n/ D n。为了解决它，Akra-Bazzi 方法表示我们应该确定满足的唯一 p

$$\left(\frac{1}{5}\right)^p + \left(\frac{7}{10}\right)^p = 1 .$$

求解 p 有点麻烦，因为 p D 0:83978 : : :4但我们无需知道 p 的确切值即可求解递归。观察到 .1/5/ ^{0} C .7/10/ ^{0} D 2 和 .1/5/ ^{1} C .7/10/ ^{1} D 9/10，因此 p 位于 0 < p < 1 范围内。事实证明，这足以让 Akra-Bazzi 方法给出解决方案。我们将利用微积分中的事实，即如果 k ¤ 1，则 R x ^{k} dx D x $^{k+1}$/.k C 1/，我们将用 k D p ¤ 1 来应用它。阿克拉巴齐

解 (4.23) 给出

$$\begin{aligned}
T(n) &= \Theta\left(n^p\left(1+\int_1^n \frac{f(x)}{x^{p+1}}\,dx\right)\right) \\
&= \Theta\left(n^p\left(1+\int_1^n x^{-p}\,dx\right)\right) \\
&= \Theta\left(n^p\left(1+\left[\frac{x^{1-p}}{1-p}\right]_1^n\right)\right) \\
&= \Theta\left(n^p\left(1+\left(\frac{n^{1-p}}{1-p}-\frac{1}{1-p}\right)\right)\right) \\
&= \Theta\left(n^p\cdot\Theta(n^{1-p})\right) && \text{(because } 1-p \text{ is a positive constant)} \\
&= \Theta(n) && \text{(by Problem 3-5(d)) .}
\end{aligned}$$

尽管 Akra-Bazzi 方法比主定理更通用，但它需要微积分，有时还需要更多的推理。如果您想忽略下限和上限，您还必须确保您的驱动函数满足多项式增长条件，尽管这很少是个问题。当它适用时，主方法使用起来要简单得多，但仅限于子问题大小大致相等时。它们都是您的算法工具包中的好工具。

练习

★ ***4.7-1***

考虑实数上的 Akra-Bazzi 递归 T .n/，如递归 (4.22) 中所示，并且 deûne T ′.n / 为

$$T'(n) = cf(n) + \sum_{i=1}^{k} a_i T'(n/b_i)\ ,$$

其中 c > 0 为常数。证明无论 T .n/ 的隐式初始条件是什么，都存在 T ′.n/ 的初始条件，使得对于所有 n > 0，T ′.n/ D cT .n/。得出结论，我们可以在任何 Akra-Bazzi 递归中放弃驱动函数的渐近性，而不会影响其渐近解。

4.7-2

证明 f .n/ D n ² 满足多项式增长条件，但 f .n/ D 2 ₙ 不满足。

4.7-3

设 f .n/ 为满足多项式增长条件的函数。证明 f .n/ 是渐近正函数，即存在一个常数 n ₀ ¬ 0，使得对于所有 n ¬ n ₀，f .n/ ¬ 0。

4.7-4

给出一个函数 f .n/ 的例子，它不满足多项式增长条件，但 f . ，.n// D ，.f .n// 。

4.7-5

使用 Akra-Bazzi 方法解决以下递归问题。

a.T .n/ D T .n/2/ C T .n/3/ C T .n/6/ C n lg n。 ***b.*** T .n/ D 3T .n/3/ C 8T .n/4/ C n ²/ lg n。 ***c.*** T .n/ D .2/3/T .n/3/ C .1/3/T .2n/3/ C lg n。 ***d.*** T .n/ D .1/3/T .n/3/ C 1/n。 ***e.*** T .n/ D 3T .n/3/ C 3T .2n/ 3/ C n ²。

？ ***4.7-6***

使用Akra-Bazzi方法证明连续主定理。

问题

4-1Recurrence examples

给出下列每个算法递归中 T.n/ 的渐近严格的上限和下限。证明你的答案。

a.T .n/ D 2T .n/2/ C n ³. ***b.*** T .n / D T .8n/11/ C n。 ***c.*** T .n/ D 1 6T .n/4/ C n ². ***d.*** T .n/ D 4T .n /2/ C n ² lg n。 ***e.*** T .n/ D 8T .n /3/ C n ². ***f.*** T .n/ D 7T .n/2/ C n ² lg n。 ***g.*** T .n/ D 2T .n/4/ C p n。 ***h.*** T .n/ D T .n 2/ C n ² 。

–

4-2Parameter-passing costs

在本书中，我们假设过程调用期间的参数传递需要常数时间，即使传递的是N 个元素的数组也是如此。此假设在大多数系统中都是有效的，因为传递的是指向数组的指针，而不是数组本身。此问题检查了三种参数传递策略的含义：

1. 数组通过指针传递。时间 D ，.1/。

2.数组通过复制传递。时间D ' .N/，其中N是数组的大小。

3. 数组的传递只复制被调用过程可能访问的子范围。如果子数组包含 n 个元素，则时间 D ，.n/。

考虑以下三种算法：

a.在排序数组中查找数字的递归二分查找算法（参见练习 2.3-6）。

b.来自第 2.3.1 节的 MERGE-SORT 程序。

c.第 4.1 节中的 MATRIX-MULTIPLY-RECURSIVE 过程。

给出九个递归式 T $_{a1}$ .N; n/; T $_{a2}$ .N; n/; : : : ; T $_{c3}$ .N; n/，当使用上述三种参数传递策略传递数组和矩阵时，上述三种算法的最坏情况运行时间。求解您的递归式，给出严格的渐近界限。

4-3Solving recurrences with a change of variables

有时，一些代数运算可以使未知递归与你之前见过的递归相似。让我们解决递归

$$T(n) = 2T\left(\sqrt{n}\right) + \Theta(\lg n) \tag{4.25}$$

采用变量变换法。

a.Deûne m D lg n 和 S.m/ D T .2 ^{m} /. 用 m 和 S.m/ 重写递归 (4.25)。

b.解决 S.m/ 的递归问题。

c.使用 S.m/ 的解决方案来得出结论：T .n/ D ，.lg n lg lg n/。

d.画出递归（4.25）的递归树，并用它直观地解释为什么解是T.n/D，.lgnlglg n/。

通过改变变量来解决以下递归：

e.T .n/ D 2T . p n/ C ，.1/.

f.T .n/D 3T . ^{3}p n/ C ，.n/.

4-4More recurrence examples
给出下列每个递归中 T .n/ 的渐近严格的上限和下限。证明你的答案。

a.T .n/ D 5T .n/3/ C n lg n。 ***b.***
T .n/ D 3T .n/3/ C n/ lg n。

c.T .n/ D 8T .n/2/ C n $_{3}$p n.

d.T.n/D2T.n/22/Cn/2.

e.T .n/ D 2T .n/2/ C n/ lg n.

f.T .n/ D T .n/2/ C T .n/4/ C T .n/8/ C n.

g.T.n/DT.n 1/C1/n.

h.T.n/DT.n 1/Clgn.

i.T .n/ D T .n 2/ C 1/ lg n. ***j.*** T .
n/ D p n T . p n/ C n.

4-5Fibonacci numbers
本题开发了斐波那契数列的性质，这些性质由第 69 页的递归 (3.31) 定义。我们将探索生成函数的技术来解决斐波那契递归。定义 ***generating function*** (或 ***formal power series***) F 为

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

其中 F $_{i}$ 是第 i 个斐波那契数。

a.证明 F .z/D z C F .z/C z^{2}F .z/。

b.表明

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

其中 是黄金比率，y 是其共轭（见第 69 页）。

c.表明

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

您可以使用第 1142 页的方程 (A.7) 的生成函数版本，P $\sum_{k=0}^{\infty}$ x ^{k} D 1/.1 x/，无需证明。由于此方程涉及生成函数，x 是形式变量，而不是实值变量，因此您不必担心求和的收敛性，也不必担心方程 (A.7) 中的要求 jx j < 1，这在这里没有意义。

d.使用部分 (c) 证明 F $_i$ D ^{i} / p 5 其中 i > 0，四舍五入到最接近的整数。（*Hint:* 观察到 $_{||}$y $_{||}$< 1。）

e.证明：当 i 为 0 时，F $_{i+2}$ ≈ ^{i}。

4-6Chip testing

Diogenes 教授有 n 个据称完全相同的集成电路芯片，从原则上讲，它们可以相互测试。教授的测试夹具一次可容纳两个芯片。当夹具装载完毕后，每个芯片都会测试另一个芯片并报告其好坏。好芯片总是能准确地报告另一个芯片的好坏，但教授不能相信坏芯片的答案。因此，测试的四种可能结果如下：

Chip *A* says	Chip *B* says	Conclusion
B is good	*A* is good	both are good, or both are bad
B is good	*A* is bad	at least one is bad
B is bad	*A* is good	at least one is bad
B is bad	*A* is bad	at least one is bad

a.证明如果至少有 n/2 个芯片是坏的，教授就无法使用基于这种成对测试的任何策略确定哪些芯片是好的。假设坏芯片可以合谋愚弄教授。

现在，您将设计一个算法来识别哪些芯片是好的，哪些是坏的，假设超过 n /2 个芯片是好的。首先，您将确定如何识别一个好的芯片。

b.说明 bn/2c 成对测试足以将问题规模缩小到近一半。也就是说，说明如何使用 bn/2c 成对测试获得最多 dn/2e 个芯片的集合，并且仍然具有一半以上的芯片是好芯片的特性。

c.说明如何递归地将 (b) 部分的解决方案应用于识别一个好芯片。给出并求解描述识别一个好芯片所需测试次数的递归式。

是o你现在已经确定如何识别一个好的c 时髦的。

d.说明如何利用附加的“ .n/ ”成对检验来识别所有优质芯片。

4-7Monge arrays

如果对于所有的 i 、 j 、 k 和 l 都有 1 = i < k = m 和 1 = j < l = n，则 m = n 实数数组 A 为 ***Monge array***，有

$A[i, j] + A[k, l] \le A[i, l] + A[k, j]$.

换句话说，每当我们选取 Monge 数组的两行和两列，并考虑行和列交叉处的四个元素时，左上角和右下角元素的总和小于或等于左下角和右上角元素的总和。例如，以下数组是 Monge：

10 17 13 28 23 1
7 22 16 29 23 24
28 22 34 24 11 1
3 6 17 7 45 44 32
37 23 36 33 19 2
1 6 75 66 51 53 3
4

a.证明一个数组是 Monge 当且仅当对于所有的 i D 1; 2; :::; m 1 和 j D 1; 2; :::; n ，我们有

$A[i, j] + A[i + 1, j + 1] \le A[i, j + 1] + A[i + 1, j]$.

(*Hint:* 对于 “if” 部分，分别对行和列使用归纳法。)

b.以下数组不是 Monge。更改一个元素以使其成为 Monge。（*Hint:* 使用部分 (a)。）

37 23 22 32 2
1 6 7 10 53 3
4 30 31 32 13
9 6 43 21 15
8

c.令 f .i / 为包含第 i 行最左边最小元素的列的索引。证明对于任何 m n Monge 数组，f .1/ = f .2/ = = f .m/。

d.这里描述了一种分治算法，该算法计算 m n Monge 数组 A 中每一行最左边的最小元素：

构造由 A 的偶数行组成的 A 的子矩阵 A′。递归地确定 A′ 中每行的最左最小值。然后计算 A 的奇数行中最左最小值。

解释如何在 O.m C n/ 时间内计算 A 奇数行的最左最小值（假设偶数行的最左最小值已知）。

e.写出部分(d)中算法运行时间的递归式。证明其解为O.m C n log m/。

章节注释

分而治之作为一种设计算法的技术至少可以追溯到 1962 年 Karatsuba 和 Ofman 的一篇文章 [242]，但它可能在那之前就被使用了。根据 Heideman、Johnson 和 Burrus [211] 的说法，C. F. Gauss 于 1805 年设计了第一个快速傅里叶变换算法，而 Gauss 的公式将问题分解为更小的子问题，然后合并这些子问题的解决方案。

Strassen 算法 [424] 在 1969 年问世时引起了极大的轰动。在此之前，很少有人想象到有一种算法可以比基本的矩阵乘法程序渐进地更快。此后不久，S. Winograd 将子矩阵加法的次数从 18 次减少到 15 次，同时仍然使用 7 次子矩阵乘法。这一改进（Winograd 显然从未发表过，并且在文献中经常被错误引用）可能增强了该方法的实用性，但不会影响其渐进性能。Probert [368] 描述了 Winograd 算法，并表明在 7 次乘法的情况下，15 次加法是可能的最小值。

Strassen 的矩阵乘法界限一直保持不变，直到 1987 年，Coppersmith 和 Winograd [103] 取得了重大进展，改进了

使用基于张量积的复杂但极不实用的算法，将时间限制在 $O.n^{2.376}/$。大约 25 年后，渐近上限才再次得到改善。2012 年，Vassilevska Williams [445] 将其改进为 $O.n^{2.37287}/$，两年后，Le Gall [278] 实现了 $O.n^{2.37286}/$，他们两人都使用了数学上令人着迷但不切实际的算法。迄今为止的最佳下限显然是 $n^2/$ 界限（显而易见，因为任何矩阵乘法算法都必须填写乘积矩阵的 n^2 个元素）。

在实践中，可以通过粗化递归叶子节点来提高 MATRIX-MULTIPLY-RECURSIVE 的性能。它还表现出比 MATRIX-MULTIPLY 更好的缓存行为，尽管 MATRIX-MULTIPLY 可以通过“平铺” 来改进。Leiserson 等人 [293] 进行了一项矩阵乘法性能工程研究，其中并行和矢量化的分治算法取得了最高性能。Strassen 算法适用于大型密集矩阵，尽管大型矩阵往往是稀疏的，而稀疏方法可以快得多。当使用有限精度的浮点值时，Strassen 算法产生的数值误差比 ，$.n^3/$ 算法更大，尽管 Higham [215] 证明 Strassen 算法对某些应用已经足够准确。

早在 1202 年，列奥纳多·波纳契 [66]（又名斐波那契）就研究了递归，斐波那契数以他的名字命名，尽管印度数学家早在几个世纪前就发现了斐波那契数。法国数学家 De Moivre [108] 引入了生成函数的方法，并用它来研究斐波那契数（见问题 4-5）。Knuth [259] 和 Liu [302] 是学习生成函数方法的良好资源。

Aho、Hopcroft 和 Ullman [5, 6] 提出了最早的解决分治算法分析中出现的递归问题的通用方法之一。主要方法改编自 Bentley、Haken 和 Saxe [52]。Akra-Bazzi 方法 (不出所料) 归功于 Akra 和 Bazzi [13]。许多研究人员都对分治递归进行了研究，包括 Campbell [79]、Graham、Knuth 和 Patashnik [199]、Kuszmaul 和 Leiserson [274]、Leighton [287]、Purdom 和 Brown [371]、Roura [389]、Verma [447] 和 Yap [462]。

Leighton [287] 研究了分治递归中的下限和上限问题，包括与定理 4.5 类似的定理。Leighton 提出了一个多项式增长条件的版本。Campbell [79] 消除了 Leighton 表述中的几个限制，并证明了存在不满足 Leighton 条件的多项式有界函数。Campbell 还仔细研究了许多其他技术问题，包括分治递归的明确性。Kuszmaul 和 Leiserson [274] 提供了定理 4.5 的证明，其中不涉及微积分或其他高等数学。Campbell 和 Leighton 都探索了简单下限和上限之外的论证扰动。

5 概率分析和随机算法

本章介绍了概率分析和随机算法。如果您不熟悉概率论的基础知识，则应阅读附录 C 的 C.13C.4 节，其中回顾了这些材料。我们将在本书中多次回顾概率分析和随机算法。

5.1 招聘问题

假设您需要雇用一名新的办公室助理。您之前的招聘尝试均未成功，因此您决定使用职业介绍所。职业介绍所每天向您发送一名候选人。您面试该候选人，然后决定是否雇用该候选人。您必须向职业介绍所支付少量费用才能面试申请人。但是，实际雇用申请人的成本更高，因为您必须担任当前的办公室助理，并向职业介绍所支付大量雇用费用。您致力于始终找到最适合这份工作的人。因此，您决定，在面试每位申请人之后，如果该申请人比当前的办公室助理更有资格，您将担任当前的办公室助理并雇用新申请人。您愿意支付此策略的最终成本，但您希望估计该成本是多少。

对页上的 HIRE-ASSISTANT 程序以伪代码形式表达了这种招聘策略。办公室助理职位的候选人编号为 1 到 n，并按该顺序进行面试。该程序假设在面试候选人 i 之后，您可以确定候选人 i 是否是您迄今为止见过的最佳候选人。它首先创建一个虚拟候选人，编号为 0，该候选人的资格低于其他每个候选人。

这个问题的成本模型与第 2 章中描述的模型不同。我们关注的不是 HIRE-ASSISTANT 的运行时间，而是面试和招聘所支付的费用。从表面上看，分析该算法的成本

```
僱傭助理 .n/
1 best  D 0 // 候选 0 是最低资格虚拟候选
2 for i D 1 to n 3 面试候选人 i 4 如果候选人 i 比
候选人 best  更好 5 best  D i 6 聘用候选人 i
```

这可能看起来与分析合并排序等的运行时间有很大不同。然而，无论我们分析成本还是运行时间，所使用的分析技术都是相同的。无论哪种情况，我们都在计算某些基本操作的执行次数。

面试的成本很低，例如 c_i，而招聘成本很高，为 c_h。设 m 为招聘人数，则与该算法相关的总成本为 O.c_i n C c_h m/。无论你招聘多少人，你总是会面试 n 名候选人，因此面试的成本总是 c_i n。因此，我们专注于分析招聘成本 c_h m。这个数量取决于你面试候选人的顺序。

此场景可作为常见计算范式的模型。算法通常需要通过检查序列中的每个元素并维护当前“获胜者”来找到序列中的最大值或最小值。招聘问题模拟了程序更新其当前获胜元素概念的频率。

最坏情况分析

最糟糕的情况是，你实际上会聘用面试过的每一位候选人。这种情况发生在候选人的质量严格按递增顺序排列的情况下，在这种情况下你会聘用 n 次，总聘用成本为 O.c_h n/。

当然，候选者并不总是按照质量递增的顺序出现。事实上，你不知道他们出现的顺序，也无法控制这个顺序。因此，很自然地，我们会问，在典型或平均情况下，我们期望会发生什么。

概率分析

Probabilisticanalysis 是在问题分析中使用概率。最常见的是，我们使用概率分析来分析算法的运行时间。有时我们用它来分析其他数量，例如

程序 HIRE-ASSISTANT。为了进行概率分析，我们必须使用输入分布的知识或做出假设。然后我们分析我们的算法，计算平均情况的运行时间，其中我们取可能输入分布的平均值或期望值。当报告这样的运行时间时，我们将其称为 ***average-case running time***。

您必须谨慎决定输入的分布。对于某些问题，您可以合理地假设所有可能输入的集合，然后可以使用概率分析作为设计有效算法的技术，并作为洞察问题的一种手段。对于其他问题，您无法描述合理的输入分布，在这些情况下您不能使用概率分析。

对于招聘问题，我们可以假设申请人是按随机顺序排列的。这对于这个问题意味着什么呢？我们假设你可以比较任意两个候选人并决定哪一个更有资格，也就是说，候选人是全序的。（有关全序的定义，请参见第 B.2 节。）因此，你可以用从 1 到 n 的唯一数字对每个候选人进行排名，使用 *rank*.i / 表示申请人 i 的排名，并采用较高排名对应于更有资格的申请人的惯例。有序列表 h*rank*.1 /; *rank*.2 /; : : : ; *rank*.n /i 是列表 h1; 2; : : : ; ni 的排列。说申请人是按随机顺序排列的，相当于说这个排名列表同样可能是数字 1 到 n 的 n Š 个排列中的任一个。或者，我们说等级形成 ***uniform random permutation***，也就是说，每个可能的 n Š 排列出现的概率均相同。

5.2 节包含招聘问题的概率分析。

随机算法

为了使用概率分析，您需要了解输入的分布。在许多情况下，您对输入分布知之甚少。即使您对分布有所了解，您也可能无法通过计算来建模这些知识。然而，概率和随机性通常作为算法设计和分析的工具，使算法的一部分随机运行。

在招聘问题中，候选人似乎是以随机顺序呈现给你的，但你无法知道他们是否真的如此。因此，为了开发一种用于招聘问题的随机算法，你需要更好地控制面试候选人的顺序。因此，我们将稍微改变模型。就业机构会提前给你发一份 n 名候选人的名单。每天，你随机选择面试哪位候选人。虽然你对这些候选人一无所知（除了他们的名字），但我们做了一个重大改变。我们不再接受给定的顺序，而是选择面试候选人。

就业机构给你提供就业机会并希望它是随机的，你反而控制了整个过程并执行了随机命令。

更一般地，如果算法的行为不仅由其输入决定，还由 ***random-number generator*** 产生的值决定，我们称该算法为 ***randomized***。假设我们有一个随机数生成器 RANDOM。调用 RANDOM.a;b/ 将返回一个介于 a 和 b 之间的整数（包括 a 和 b），每个整数的出现概率均相同。例如，RANDOM.0;1/ 以概率 1/2 生成 0，以概率 1/2 生成 1。调用 RANDOM.3;7/ 将返回 3、4、5、6 或 7 中的任意一个，每个结果的概率均为 1/5。RANDOM 返回的每个整数都与以前调用返回的整数无关。您可以将 RANDOM 想象为掷一个 .b C 1/ 面骰子来获得其输出。（实际上，大多数编程环境都提供 ***pseudorandom-number generator***：一种确定性算法，返回 “看起来” 统计上随机的数字。）

在分析随机算法的运行时间时，我们取随机数生成器返回值分布的运行时间期望。我们将随机算法的运行时间称为 ***ex- pected running time***，以此将这些算法与输入随机的算法区分开来。一般来说，我们讨论概率分布在算法输入上时的平均运行时间，并讨论算法本身进行随机选择时的预期运行时间。

练习

5.1-1

说明在程序 HIRE-ASSISTANT 第 4 行中，您始终能够确定哪个候选人是最佳的假设意味着您知道候选人排名的总体顺序。

5.1-2

描述过程 R ANDOM.a; b/ 的实现，该实现仅调用 RANDOM.0; 1/。作为 a 和 b 的函数，该过程的预期运行时间是多少？

5.1-3

您希望实现一个程序，该程序以概率 1/2 输出 0，以概率 1/2 输出 1。您可以使用一个程序 B IASED-RANDOM，它输出 0 或 1，但它以某个概率 p 输出 1，以概率 1 输出 0，其中 $0 < p < 1$。您不知道 p 是什么。给出一个使用 BIASED-RANDOM 作为子程序的算法，并返回一个无偏答案，以概率 1/2 返回 0，以概率 1/2 返回 1。作为 p 的函数，您的算法的预期运行时间是多少？

5.2 指示随机变量

为了分析许多算法，包括招聘问题，我们使用指示随机变量。指示随机变量提供了一种在概率和期望之间转换的便捷方法。给定样本空间 S 和事件 A，与事件 A 相关的 ***indicator random variable*** I fAg 定义为

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs}, \\ 0 & \text{if } A \text{ does not occur}. \end{cases} \tag{5.1}$$

举一个简单的例子，让我们确定抛出一枚公平硬币时掷出正面的预期次数。单个硬币抛出的样本空间为 S D fH; T g，其中 Pr fH g D Pr fT g D 1/2。然后，我们可以定义一个指示随机变量 X $_H$，与硬币掷出正面相关，即事件 H。此变量计算此抛出的正面的次数，如果硬币掷出正面，则为 1，否则为 0。我们写为

$$\begin{aligned} X_H &= \mathrm{I}\{H\} \\ &= \begin{cases} 1 & \text{if } H \text{ occurs}, \\ 0 & \text{if } T \text{ occurs}. \end{cases} \end{aligned}$$

一次投币得到正面的期望次数就是我们的指示变量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，一枚公平硬币掷出正面的期望次数为 1/2。如下面的引理所示，与事件 A 相关的指示随机变量的期望值等于 A 发生的概率。

Lemma5.1
给定样本空间 S 和样本空间 S 中的一个事件 A，令 X $_A$ D I fAg。则 E ŒX $_A$ D Pr fAg。

Proof根据方程（5.1）中指示随机变量的定义和期望值的定义，我们得到

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \\ &= \Pr\{A\} , \end{aligned}$$

其中A表示SA，即A的补集。 ■

尽管指示随机变量对于诸如计算一枚硬币正面朝上的预期次数这样的应用来说似乎很麻烦，但它们对于分析进行重复随机试验的情况很有用。例如，在附录 C 中，指示随机变量提供了一种确定 n 枚硬币正面朝上的预期次数的简单方法。一种方法是分别考虑得到 0 个正面、1 个正面、2 个正面等的概率，以得出第 1199 页方程 (C.41) 的结果。或者，我们可以采用方程 (C.42) 中提出的更简单的方法，该方法隐式地使用了指示随机变量。为了更明确地说明这个论点，让 X $_i$ 成为与第 i 个硬币出现正面的事件相关的指示随机变量：X $_i$ D I 如果第 i 个硬币导致事件 H g。令 X 为表示 n 枚硬币正面朝上的总数的随机变量，则

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

为了计算预期的正面次数，对上式两边取期望值，可得到

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \ . \tag{5.2}$$

根据引理 5.1，每个随机变量的期望为 E ŒX $_i$ D 1/2，其中 i D 1; 2; : : :; n 。然后，我们可以计算期望之和：P $_{i=1}^{n}$ E ŒX $_i$ D n/2。但公式 (5.2) 要求的是和的期望，而不是期望之和。如何解决这个难题？期望的线性，第 1192 页的公式 (C.24) 可以解决此问题：*the expectation of the sum always equals the sum of the expectations*。即使随机变量之间存在依赖关系，期望的线性也适用。将指示随机变量与期望的线性相结合，为我们提供了一种强大的技术来计算多个事件发生时的预期值。我们现在可以计算预期的正面次数：

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

因此，与公式（C.41）中的方法相比，指示性随机变量大大简化了计算。我们在本书中通篇使用指示性随机变量。

使用指示随机变量分析招聘问题

回到招聘问题，我们现在想计算你雇佣新办公室助理的预期次数。为了使用概率分析，我们假设候选人以随机顺序到达，如第 5.1 节所述。（我们将在第 5.3 节中看到如何消除这一假设。）让 X 成为其值等于你雇佣新办公室助理的次数的随机变量。然后我们可以应用第 1192 页方程 (C.23) 中预期值的定义来获得

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

但这种计算会很麻烦。相反，我们可以使用指示随机变量来简化计算。

要使用指示随机变量，不要只定义一个表示您雇用新办公室助理的次数的变量来计算 E ŒX ，而是将招聘过程视为重复的随机试验，并定义 n 个变量来表示每个特定候选人是否被录用。具体来说，让 X $_i$ 成为与第 i 个候选人被录用的事件相关的指示随机变量。因此，

$$\begin{aligned} X_i &= \mathrm{I}\{\text{candidate } i \text{ is hired}\} \\ &= \begin{cases} 1 & \text{if candidate } i \text{ is hired} , \\ 0 & \text{if candidate } i \text{ is not hired} , \end{cases} \end{aligned}$$

和

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

引理 5.1 给出

$$\mathrm{E}[X_i] = \Pr\{\text{candidate } i \text{ is hired}\} ,$$

因此我们必须计算 HIRE-ASSISTANT 的第 536 行被执行的概率。

在第 6 行中，当候选人 i 比候选人 1 到 i 1 中的每一个候选人都优秀时，候选人 i 才会被录用。因为我们假设候选人以随机顺序出现，所以前 i 个候选人的出现顺序也是随机的。这些前 i 个候选人中的任何一个都有可能成为迄今为止最有资格的候选人。候选人 i 有 1/ i 个概率比候选人 1 到 i 1 更有资格，因此有 1/ i 个概率被录用。根据引理 5.1，我们得出结论

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

现在我们可以计算 E ŒX ：

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(by equation (5.3))} && (5.5)\\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(by equation (C.24), linearity of expectation)} \\
&= \sum_{i=1}^{n} \frac{1}{i} && \text{(by equation (5.4))} \\
&= \ln n + O(1) && \text{(by equation (A.9), the harmonic series) .} && (5.6)
\end{aligned}$$

即使你面试了 n 个人，但平均下来你实际上只会聘用其中大约 ln n 个人。我们将此结果总结为以下引理。

Lemma5.2
假设候选人以随机顺序出现，算法 HIREASSISTANT 的平均总招聘成本为 O.c$_h$ ln n/。

Proof这个界限可以直接从我们对雇佣成本的定义和方程（5.6）得出，这表明预期雇佣人数大约为 ln n。

■

平均情况下的雇用成本较最坏情况下的雇用成本 O.c$_h$ n/ 有显著改善。

练习

5.2-1
在 HIRE-ASSISTANT 中，假设候选人以随机顺序出现，您恰好雇用一次的概率是多少？您恰好雇用 n 次的概率是多少？

5.2-2
在 HIRE-ASSISTANT 中，假设候选人以随机顺序出现，您恰好雇用两次的概率是多少？

5.2-3
使用指示随机变量计算 n 个骰子总和的期望值。

5.2-4

本练习要求您（部分）验证即使随机变量不独立，期望的线性是否成立。考虑两个独立掷出的 6 面骰子。总和的期望值是多少？现在考虑第一个骰子正常掷出，然后第二个骰子设置为等于第一个骰子上显示的值的情况。总和的期望值是多少？现在考虑第一个骰子正常掷出，然后第二个骰子设置为等于 7 减去第一个骰子的值的情况。总和的期望值是多少？

5.2-5

使用指示随机变量解决以下问题，即 ***hat-check problem***。餐厅的 n 位顾客每人将一顶帽子交给帽子保管员。帽子保管员以随机顺序将帽子还给顾客。取回自己帽子的顾客预期数量是多少？

5.2-6

假设 AŒ1 W n 为 n 个不同数字的数组。如果 i < j 且 AŒi > AŒj ，则对 .i; j / 称为 A 的 ***inversion***。（有关反转的更多信息，请参阅第 47 页的问题 2-4。）假设 A 的元素形成 h1; 2; : : : ; ni 的均匀随机排列。使用指示随机变量计算反转的预期次数。

5.3 随机化算法

在上一节中，我们展示了了解输入的分布如何帮助我们分析算法的平均行为。如果你不知道分布怎么办？那么你就无法执行平均分析。但是，如第 5.1 节所述，您可能能够使用随机算法。

对于诸如招聘问题这样的问题，假设输入的所有排列都具有同等可能性是有帮助的，概率分析可以指导我们开发随机算法。我们不是使用 *assuming* 输入分布，而是使用 *impose* 分布。具体来说，在运行算法之前，让我们随机排列候选人，以强制执行每个排列都具有同等可能性的属性。虽然我们已经修改了算法，但我们仍然希望大约 ln n 次雇用一名新的办公室助理。但现在我们预计这是 *any* 输入的情况，而不是从特定分布中抽取的输入的情况。

让我们进一步探讨概率分析和随机算法之间的区别。在第 5.2 节中，我们声称，假设候选

以随机顺序到达，您雇用新办公室助理的预期次数约为 ln n。该算法是确定性的：对于任何特定输入，雇用新办公室助理的次数始终相同。此外，雇用新办公室助理的次数因输入的不同而不同，并且取决于各个候选人的排名。由于这个数字仅取决于候选人的排名，因此为了表示特定输入，我们可以按顺序列出候选人的排名 h*rank*.1/; *rank*.2/; : : : ; *rank*.n/i。给定排名列表 A_1 D h 1; 2; 3; 4; 5; 6; 7; 8; 9; 10i，新办公室助理总是被雇用 10 次，因为每个后续候选人都比前一个候选人更好，并且每次迭代都会执行 HIRE-ASSISTANT 的第 536 行。给定排名列表 A_2 D h10；9；8；7；6；5；4；3；2；1i，新办公室助理只在第一次迭代中被雇用一次。给定排名列表 A_3 D h5；2；1；8；4；7；10；9；3； 6i，在面试排名 5、8 和 10 的候选人后，新办公室助理被雇用了三次。回想一下，我们的算法的成本取决于您雇用新办公室助理的次数，我们发现有昂贵的输入，例如 A_1，便宜的输入，例如 A_2，以及中等昂贵的输入，例如 A_3。

另一方面，考虑一下随机算法，该算法首先对候选列表进行排列，然后确定最佳候选。在这种情况下，我们在算法中进行随机化，而不是在输入分布中进行随机化。给定一个特定的输入，比如上面的 A_3，我们无法说出最大值更新了多少次，因为这个数量随着算法的每次运行而不同。第一次对 A_3 运行算法时，它可能会产生排列 A_1 并执行 10 次更新。但第二次运行算法时，它可能会产生排列 A_2 并只执行一次更新。第三次运行算法时，它可能会执行其他次数的更新。每次运行算法时，其执行都取决于所做的随机选择，并且可能与算法的上次执行不同。对于此算法和许多其他随机算法，*no particular input elicits its worst-case behavior*。即使你最坏的敌人也无法产生糟糕的输入数组，因为随机排列使得输入顺序变得无关紧要。只有当随机数生成器产生 “不幸” 排列时，随机算法才会表现不佳。

对于招聘问题，代码中唯一需要做的改变是随机排列数组，就像在 RANDOMIZED-HIRE-ASSISTANT 程序中所做的那样。这个简单的改变创建了一个随机算法，其性能与假设候选人以随机顺序出现时的性能相匹配。

随机僱傭助理 .n/

1 随机排列候选人名单 2 HIRE-ASSISTA
NT .n/

Lemma5.3
程序 R ANDOMIZED-HIRE-ASSISTANT 的预期雇用成本为 O.c $_h$ ln n/。

Proof对输入数组进行排列可实现与第 5.2 节中 HIRE-ASSISTANT 的概率分析相同的情况。 ■

通过仔细比较引理 5.2 和引理 5.3，你可以看到概率分析和随机算法之间的区别。引理 5.2 对输入做出了假设。引理 5.3 没有做出这样的假设，尽管随机化输入需要一些额外的时间。为了与我们的术语保持一致，我们用平均招聘成本来表达引理 5.2，用预期招聘成本来表达引理 5.3。在本节的其余部分，我们将讨论随机排列输入所涉及的一些问题。

随机排列数组

许多随机化算法通过对给定的输入数组进行排列来随机化输入。我们将在本书的其他地方看到其他随机化算法的方法，但现在，让我们看看如何随机排列一个包含 n 个元素的数组。目标是产生一个 ***uniform random permutation***，即一个与任何其他排列一样可能的排列。由于有 nŠ 种可能的排列，我们希望产生任何特定排列的概率为 1/nŠ。

你可能认为，要证明一个排列是均匀随机排列，只要证明对于每个元素 AŒi ，该元素最终出现在位置 j 的概率为 1/n 就足够了。练习 5.3-4 表明，这个较弱的条件实际上是不够的。

我们生成随机排列的方法是对数组 ***in place*** 进行排列：输入数组中最多有常数个元素存储在数组之外。程序 RANDOMLY-PERMUTE 在 ，.n/ 时间内对数组 AŒ1 W n 进行原地排列。在第 i 次迭代中，它从元素 AŒi 到 AŒn 中随机选择元素 AŒi 。在第 i 次迭代之后，AŒi 永远不会改变。

随机排列 .A; n/

1 for i D 1 to n 2 将 AŒi 与 AŒRANDOM.i; n/ 交换

我们使用循环不变量来表明过程 RANDOMLY-PERMUTE 产生均匀随机排列。一组 n 个元素上的 k-***permutation*** 是 se-

包含 n 个元素中的 k 个的序列，没有重复。（参见附录 C 第 1180 页。）有 n Š /.n k/ Š 个这样的可能的 k-排列。

Lemma5.4
程序 RANDOMLY-PERMUTE 计算均匀随机排列。

Proof我们使用以下循环不变量：

> 在第 132 行的 for 循环第 i 次迭代之前，对于 n 个元素的每个可能的 .i 1/-排列，子数组 AŒ1 W i 1 包含该 .i 1/-排列，概率为 .n C 1/ Š /n Š。

我们需要证明这个不变量在第一次循环迭代之前为真，循环的每次迭代都保持不变变量，循环终止，并且不变量提供了一个有用的属性来显示循环终止时的正确性。

初始化：考虑第一次循环迭代之前的情况，即 i D 1。循环不变量表示，对于每个可能的 0 排列，子数组 AŒ1 W 0 包含此 0 排列的概率为 .n i C 1/ Š /n Š D n Š /n Š D 1。子数组 AŒ1 W 0 为空子数组，0 排列没有元素。因此，AŒ1 W 0 包含任何 0 排列的概率为 1，循环不变量在第一次迭代之前成立。

维护：通过循环不变量，我们假设在第 i 次迭代之前，每个可能的 .i 1/-排列以概率 .n i C 1/ Š /n Š 出现在子数组 AŒ1W i 中。我们将证明在第 i 次迭代之后，每个可能的 i -排列以概率 .n i / Š /n Š 出现在子数组 AŒ1W i 中。然后在下一次迭代中增加 i 以保持循环不变量。

> 让我们检查第 i 次迭代。考虑一个特定的 i -排列，并将其中的元素表示为 hx $_1$; x $_2$; : : : ; x $_i$ i。此排列由一个 .i 1/-排列 hx $_1$; : : : ; x $_{i-1}$ i 组成，后跟算法放置在 AŒi 中的值 x $_i$ 。令 E $_1$ 表示前 i 1 次迭代在 AŒ1 W i 1 中创建特定 .i 1/-排列 hx $_1$; : : : ; x $_{i-1}$ i 的事件。根据循环不变量，Pr fE $_1$ g D .n 1 C 1/ Š /n Š。假设 E $_2$ 表示第 i 次迭代将 x $_i$ 置于位置 AŒi 的事件。当 E $_1$ 和 E $_2$ 同时出现时，i 置换 hx $_1$; : : : ; x $_i$ i 恰好出现在 AŒ1 W i 中，因此我们希望计算 Pr fE $_2$ \ E $_1$ g。使用第 1187 页上的公式 (C. 16)，我们得到
>
> $$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\}\Pr\{E_1\} \ .$$
>
> 概率 Pr fE $_2$ j E $_1$ g 等于 1/.nC1/，因为在第 2 行中，算法从位置 AŒi W n 中的 n C1 值中随机选择 x $_i$ 。因此，我们有

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

终止：循环终止，因为它是一个迭代 n 次的 for 循环。终止时，i D n C 1，并且我们有子数组 AŒ1 W n 是给定的 n 次排列，概率为 .n .n C 1/ C 1/Š /nŠ D 0Š/nŠ D 1/nŠ。

因此，RANDOMLY-PERMUTE 产生均匀随机排列。 ■

随机算法通常是解决问题最简单、最有效的方法。

练习

5.3-1

马索教授反对引理 5.4 证明中使用的循环不变量。他质疑它在第一次迭代之前是否成立。他推断，我们可以很容易地声明一个空子数组不包含 0 排列。因此，空子数组包含 0 排列的概率应该为 0，从而使第一次迭代之前的循环不变量无效。重写过程 RANDOMLY-PERMUTE，使其相关的循环不变量适用于第一次迭代之前的非空子数组，并修改引理 5.4 的证明以适合您的过程。

5.3-2

凯尔普教授决定编写一个程序，随机生成除 ***identity permutation*** 之外的任何排列，其中每个元素都回到其开始的位置。他提出了 PERMUTE-WITHOUT-I DENTITY 程序。这个程序能达到凯尔普教授的预期吗？

```
无 I 身份排列 .A; n/

1 for i D 1 to n  1 2 将 AŒi     与 AŒRAND
OM.i C 1; n/     交换
```

5.3-3

考虑一下本页上的 PERMUTE-WITH-ALL 过程，它不是将元素 AŒi 与子数组 AŒi W n 中的随机元素交换，而是将其与数组中任意位置的随机元素交换。PERMUTE-WITH-ALL 会产生均匀随机排列吗？为什么会产生或为什么不会产生？

```
排列所有 .A；n/

1 for i D 1 to n 2 将 AŒi      与 AŒRAN
DOM.1; n/     交换
```

5.3-4

Knievel 教授建议使用 P ERMUTE-BY-CYCLE 程序来生成均匀随机排列。证明每个元素 AŒi 都有 1/n 的概率出现在 B 中的任意特定位置。然后通过证明生成的排列不是均匀随机的来表明 Knievel 教授是错误的。

```
按周期排列.A; n/

1 让 BŒ1 W n     成为新数
组 2 offset  D RANDOM.1; n/
3 for i D 1 到 n 4 dest  D i C
offset  5 if dest > n 6 dest  D
dest  n 7 BŒdest      D AŒi
    8 返回 B
```

5.3-5

Gallup 教授想要创建集合 f1; 2; 3; : : : ; ng 的 ***random sample***，即一个包含 m 个元素的子集 S，其中 0 = m = n，并且每个 m 子集被创建的概率相等。一种方法是设定 AŒi = Di，对于 i D 1; 2; 3; : : : ; n，调用 RANDOMLY-PERMUTE .A/，然后仅取前 m 个数组元素。此方法会调用 n 次 RANDOM 过程。在 Gallup 教授的应用程序中，n 比 m 大得多，因此教授想要通过更少的 RANDOM 调用来创建随机样本。

```
随机样本 .m; n/

1 S D ; 2 for k D n  m C 1 to n // 迭代 m 次 3 i D RA
NDOM.1; k/ 4 if i 2 S 5 S D S [ fkg 6 else S D S [ fig
7 return S
```

说明上一页中的过程 RANDOM-SAMPLE 返回 f1; 2; 3; : : : ; ng 的随机 m 子集 S，其中每个 m 子集的可能性均等，同时仅对 RANDOM 进行 m 次调用。

?5.4 指示随机变量的概率分析及进一步应用

本高级部分通过四个示例进一步说明了概率分析。第一个示例确定在一个有 k 个人的房间里，其中两个人生日相同的概率。第二个示例检查将球随机扔进箱子时会发生什么。第三个示例调查投掷硬币时连续出现正面的“条纹”。最后一个示例分析了招聘问题的一个变体，在该问题中，您必须在不实际面试所有候选人的情况下做出决策。

5.4.1 生日悖论

我们的第一个例子是 ***birthday paradox***。一间屋子里必须有多少人，其中两个人出生在同一天的概率才会达到 50%？答案是出人意料的少。矛盾的是，事实上，这个数字远远少于一年的天数，甚至少于一年天数的一半，我们将会看到这一点。

为了回答这个问题，我们用整数 1; 2; : : : ; k 来索引房间里的人，其中 k 是房间里的人数。我们忽略闰年的问题，假设所有年份都有 n = 365 天。对于 i = 1; 2; : : : ; k，让 b $_i$ 表示人 i 的生日所在的日期，其中 1 b $_i$ n。我们还假设生日在一年中的 n 天中均匀分布，因此对于 i = 1; 2; : : : ; k 和 r = 1; 2; : : : ; n，有 Pr fb $_i$ D r g D 1/n。

两个给定的人（例如 i 和 j）生日相同的概率取决于生日的随机选择是否独立。从现在开始，我们假设生日是独立的，因此 i 的生日和 j 的生日都落在第 r 天的概率为

$$\begin{aligned}\Pr\{b_i = r \text{ and } b_j = r\} &= \Pr\{b_i = r\}\Pr\{b_j = r\} \\ &= \frac{1}{n^2}\,.\end{aligned}$$

因此，它们都在同一天下跌的概率是

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

$$= \sum_{r=1}^{n} \frac{1}{n^2}$$
$$= \frac{1}{n} . \tag{5.7}$$

更直观地讲，一旦选择了 b $_i$，则选择 b $_j$ 为同一天的概率为 1/n。只要生日是独立的，i 和 j 有同一天生日的概率与其中一个人的生日落在某一天的概率相同。

我们可以通过观察互补事件来分析 k 个人中至少有 2 个人生日相同的概率。至少有两个生日相同的概率是 1 减去所有生日都不同的概率。k 个人生日不同的事件 B $_k$ 是

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

其中 A $_i$ 表示对于所有 j < i，人 i 的生日与人 j 的生日都不同的事件。由于我们可以写成 B $_k$ D A $_k$ \ B $_{k-1}$，因此我们从第 1189 页的方程 (C.18) 中得到递归

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

其中，我们以 Pr fB $_1$ g D Pr fA $_1$ g D 1 作为初始条件。换句话说，假设 b $_1$ ；b $_2$ ；: : : ；b $_k$ 是不同生日的概率，等于 b $_1$ ；b $_2$ ；: : : ；b $_{k-1}$ 是不同生日的概率乘以 b $_k$ ¤ b $_i$ (其中 i D 1; 2; : : :; k 1) 的概率。

如果 b $_1$; b $_2$; : : : ; b $_{k-1}$ 不同，则对于 i D 1; 2; : : : ; k 1，b $_k$ ¤ b $_i$ 的条件概率为 Pr fA $_k$ j B $_{k-1}$ g D .n k C 1//n，因为在 n 天中，有 n .k 1/ 天未取。我们迭代地应用递归公式 (5.8) 得到

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

第 66 页的不等式 (3.14)，1 C x e ^{x}，得出

$$\begin{aligned}\Pr\{B_k\} &\le e^{-1/n}e^{-2/n}\cdots e^{-(k-1)/n}\\ &= e^{-\sum_{i=1}^{k-1} i/n}\\ &= e^{-k(k-1)/2n}\\ &\le \frac{1}{2}\end{aligned}$$

当 .k 1//2n ln.1/2/ 时。当 k.k 1/ 2n ln 2 时，或当 k .1 C p 1 C .8 ln 2/n//2 时，所有 k 个生日不同的概率最多为 1/2。对于 n ≥ 365，我们必须有 k 23 。因此，如果一个房间里至少有 23 个人，那么至少有两个人生日相同的概率至少为 1/2。由于火星上的一年是 669 个火星日，因此需要 31 个火星人才能达到同样的效果。

使用指示随机变量的分析

指示随机变量可以对生日悖论进行更简单但近似的分析。对于房间中 k 个人中的每一对 .i; j，定义指示随机变量 X_{ij}，其中 1 = i < j = k，通过

$$\begin{aligned}X_{ij} &= \mathrm{I}\{\text{person } i \text{ and person } j \text{ have the same birthday}\}\\ &= \begin{cases}1 & \text{if person } i \text{ and person } j \text{ have the same birthday},\\ 0 & \text{otherwise}.\end{cases}\end{aligned}$$

根据公式（5.7），两个人生日相同的概率为 1/n，因此根据第 130 页的引理 5.1，我们有

$$\begin{aligned}\mathrm{E}[X_{ij}] &= \Pr\{\text{person } i \text{ and person } j \text{ have the same birthday}\}\\ &= 1/n\ .\end{aligned}$$

假设 X 为随机变量，用于统计生日相同的个体对的数量，则有

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

取双方的期望值并应用期望的线性，我们得到

$$\begin{aligned}\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k-1}\sum_{j=i+1}^{k} X_{ij}\right]\\ &= \sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\mathrm{E}[X_{ij}]\end{aligned}$$

$$= \binom{k}{2}\frac{1}{n}$$
$$= \frac{k(k-1)}{2n}.$$

因此，当 k≥1/2n 时，生日相同的人的预期对数至少为 1。因此，如果一个房间里至少有 p2nC1 个人，我们可以预期至少有两个人的生日相同。对于 n≥365，如果 k≥28，生日相同的人的预期对数为 .28（27//.2）（365/1:356）。因此，如果至少有 28 个人，我们预计至少会找到一对生日相同的人。在火星上，每年有 669 天，我们需要至少 38 个火星人。

第一个分析只使用了概率，确定了使生日配对的概率超过 1/2 所需的人数；第二个分析使用了指示随机变量，确定了使生日配对的预期数量为 1 所需的人数。虽然两种情况下的具体人数不同，但它们渐近相同： , . p n/。

–

5.4.2 球和箱子

考虑这样一个过程：你随机地将相同的球扔进 b 个箱子里，编号为 1；2；：：：；b。每次抛球是独立的，每次抛球都有同等的可能性落入任意一个箱子里。抛出的球落入任意一个箱子的概率为 1/b。如果我们将抛球过程视为一系列伯努利试验（参见附录 C.4），其中成功意味着球落入给定的箱子中，那么每次试验的成功概率为 1/b。这个模型对于分析散列（参见第 11 章）特别有用，我们可以回答有关抛球过程的各种有趣问题。（问题 C-2 提出了有关球和箱子的更多问题。）

How many balls fall in a given bin? 落入给定箱子的球数服从二项分布 b.kI n; 1/b/。如果抛 n 个球，第 1199 页的公式 (C.41) 告诉我们，落入给定箱子的球的预期数量为 n/b。 • *How many balls must you toss, on the average, until a given bin contains a ball?* 给定箱子收到球之前的投掷次数遵循几何分布，概率为 1/b，根据第 1197 页的公式 (C.36)，成功之前的预期投掷次数为 1/.1/b / D b。 • *How many balls must you toss until every bin contains at least one ball?* 我们将一次投掷中将球落入空箱称为一次“命中。”我们想知道获得 b 次命中所需的预期投掷次数 n。

利用命中数，我们可以将 n 次投掷划分为几个阶段。第 i 个阶段包括第 .i 1 次命中之后到第 i 次命中（包括第 i 次命中）的投掷。第一个阶段包括第一次投掷，因为当所有箱子都为空时，您一定会命中。对于第 i 个阶段的每次投掷，有 i 1 个箱子装有球，而 b 1 C 1 个箱子为空。因此，对于第 i 个阶段的每次投掷，命中的概率为 .b 1 C 1//b。

令 n_i 表示第 i 阶段的投掷次数。获得 b 次命中所需的投掷次数为 n D P $_{i=1}^{b} n_i$。每个随机变量 n_i 都具有成功概率为 .b i C 1//b 的几何分布，因此，根据方程 (C.36)，我们有

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

根据期望的线性，我们有

$$\begin{aligned}
\mathrm{E}[n] &= \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] \\
&= \sum_{i=1}^{b} \mathrm{E}[n_i] \\
&= \sum_{i=1}^{b} \frac{b}{b-i+1} \\
&= b\sum_{i=1}^{b} \frac{1}{i} && \text{(by equation (A.14) on page 1144)} \\
&= b(\ln b + O(1)) && \text{(by equation (A.9) on page 1142) .}
\end{aligned}$$

因此，大约需要 b ln b 次投掷，我们才能预期每个箱子里都有一个球。这个问题也称为 ***coupon collector's problem***，它表示如果您尝试收集 b 张不同的优惠券，那么您应该预期获得大约 b ln b 张随机获得的优惠券才能成功。

5.4.3 条纹

假设你投掷一枚公平硬币 n 次。你期望看到的连续正面的最长连续次数是多少？我们将分别证明上限和下限，以表明答案是“ .lg n ”。

我们首先证明，最长的连续正面的期望长度为 O.lg n/。每枚硬币正面的概率为 1/2。设 A_{ik} 表示至少有 k 个正面的连续事件从第 i 枚硬币开始，或者更准确地说，表示连续 k 枚硬币 i；i C 1；：：：；i C k 1 只出现正面，其中 1 = k = n 和 1 = i = n k C1。由于硬币相互独立，对于任何给定事件 A_{ik}，所有 k 枚硬币都是正面的概率为

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

对于 k D 2 dlg ne,

$$\begin{aligned}\Pr\{A_{i,2\lceil \lg n\rceil}\} &= \frac{1}{2^{2\lceil \lg n\rceil}} \\ &\le \frac{1}{2^{2\lg n}} \\ &= \frac{1}{n^2} ,\end{aligned}$$

因此，长度至少为 2 dlg ne 的一连串头球从位置 i 开始的概率非常小。最多有 n 2 dlg ne C 1 个位置可以开始这样的一连串头球。因此，长度至少为 2 dl g ne 的一连串头球从任何地方开始的概率为

$$\begin{aligned}&\Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n\rceil+1} A_{i,2\lceil \lg n\rceil}\right\} \\ &\quad\le \sum_{i=1}^{n-2\lceil \lg n\rceil+1} \Pr\{A_{i,2\lceil \lg n\rceil}\} \quad \text{(by Boole's inequality (C.21) on page 1190)} \\ &\quad\le \sum_{i=1}^{n-2\lceil \lg n\rceil+1} \frac{1}{n^2} \\ &\quad< \sum_{i=1}^{n} \frac{1}{n^2} \\ &\quad= \frac{1}{n} .\end{aligned} \tag{5.10}$$

我们可以使用不等式 (5.10) 来限制最长条纹的长度。对于 j D 0; 1; 2; : : : ; n，令 L_j 为最长正面条纹的长度恰好为 j 的事件，令 L 为最长条纹的长度。根据期望值的定义，我们有

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

我们可以尝试使用类似于不等式 (5.10) 中计算的每个 Pr fL $_j$g 的上限来评估这个和。不幸的是，这种方法产生的界限很弱。然而，我们可以使用从上述分析中获得的一些直觉来获得一个好的界限。因为等式 (5.11) 中的和中没有一个单独的项是因子 j 和 Pr fL $_j$g 同时很大的。为什么？当 j $\geq$ 2 dlg ne 时，Pr fL $_j$g 非常小，而当 j < $\geq$ 2 dlg ne 时，j 相当小。更准确地说，由于 j D 0; 1; : : : ; n 的事件 L $_j$ 是不相交的，所以长度至少为 2 dlg ne 的连续头部从任何地方开始的概率为 P $^n_{j=2\lceil \lg n\rceil}$ Pr fL $_j$g。不等式 (5.10) 告诉我们，长度至少为 2 dlg ne 的一连串头部从任意位置开始的概率小于 1/n，这意味着 P $^n_{j=2\lceil \lg n\rceil}$ Pr fL $_j$g < 1/n。此外，注意到 P $^n_{j=0}$ Pr fL $_j$g D 1，我们有 P $^{2\lceil \lg n\rceil -1}_{j=0}$ Pr fL $_j$g = 1。因此，我们得到

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

连续掷出超过 r 次正面的概率会随着 r 的增加而迅速减小。让我们粗略地计算一下连续掷出至少 r 次正面的概率，其中 r 为 1。连续掷出至少 r 次正面的概率从位置 i 开始为

$$
\begin{aligned}
\Pr\{A_{i,r\lceil \lg n\rceil}\} &= \frac{1}{2^{r\lceil \lg n\rceil}} \\
&\leq \frac{1}{n^r} .
\end{aligned}
$$

至少 r 次正面连续投掷不可能在最后 n 个硬币中开始，但我们可以高估这种连续投掷的概率，允许它在 n 个硬币中的任意位置开始。那么至少 r 次正面连续投掷的概率

最多发生

$$\begin{aligned}\Pr\left\{\bigcup_{i=1}^{n} A_{i,r\lceil \lg n\rceil}\right\} &\le \sum_{i=1}^{n} \Pr\{A_{i,r\lceil \lg n\rceil}\} \qquad \text{(by Boole's inequality (C.21))}\\ &\le \sum_{i=1}^{n} \frac{1}{n^r}\\ &= \frac{1}{n^{r-1}}\,.\end{aligned}$$

等效地，最长条纹的长度小于 r dlg ne 的概率至少为 1 1/n $^{r-1}$。

举例来说，在 n D 1000 次投币过程中，出现至少 2 次正面朝上的概率最多为 1/n D 1/1000。出现至少 3 次正面朝上的概率最多为 1/n ^{2} D 1/1,000,000。

现在让我们证明一个补充的下限：n 个硬币中，正面最长的连串长度预期为 .lg n/。为了证明这个界限，我们通过将 n 个硬币分成大约 n/s 个组（每个组 s 个硬币）来寻找长度为 s 的连串。如果我们选择 s D b.lg n//2c，我们将发现这些组中至少有一个组出现正面的可能性很大，这意味着最长连串的长度至少为 s D .lg n/。然后我们将证明最长连串的预期长度为 .lg n/。

让我们将 n 个硬币分成至少 bn/ b.lg n//2cc 个连续硬币组，每个组有 b.lg n//2c 个硬币，并限制没有一组硬币全是正面的概率。根据公式 (5.9)，从位置 i 开始的一组硬币全是正面的概率为

$$\begin{aligned}\Pr\{A_{i,\lfloor(\lg n)/2\rfloor}\} &= \frac{1}{2^{\lfloor(\lg n)/2\rfloor}}\\ &\ge \frac{1}{\sqrt{n}}\,.\end{aligned}$$

因此，长度至少为 b.lg n//2c 的一连串正面不从位置 i 开始的概率至多为 1 1 / p n。由于 bn/ b.lg n//2cc 组是由相互排斥的独立硬币组成的，因此这些组 *fails* 中的每一组都是长度为 b.lg n//2c 的一连串正面的概率至多为

$$\begin{aligned}\left(1-1/\sqrt{n}\right)^{\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor} &\le \left(1-1/\sqrt{n}\right)^{n/\lfloor(\lg n)/2\rfloor-1}\\ &\le \left(1-1/\sqrt{n}\right)^{2n/\lg n-1}\\ &\le e^{-(2n/\lg n-1)/\sqrt{n}}\\ &= O(e^{-\ln n})\\ &= O(1/n)\,. \qquad (5.12)\end{aligned}$$

为了进行这一论证，我们使用了第 66 页的不等式 (3.14)，1Cx∈E^x，以及您可以验证的事实，即当 n 足够大时，.2n/lgn1//pnlnn。

我们要限制最长连续掷硬币次数等于或超过 b.lg n//2c 的概率。为此，令 L 为最长连续掷硬币次数等于或超过 s D b.lg n//2c 的事件。令 L 为互补事件，即最长连续掷硬币次数严格小于 s，即 Pr fLg C Pr L D 1。令 F 为每组掷硬币次数都不能成为 s 次掷硬币的事件。根据不等式 (5.12)，我们有 Pr fF g D O.1/n/。如果最长连续掷硬币次数小于 s，则每组掷硬币次数肯定不能成为 s 次掷硬币的事件，这意味着事件 L 蕴涵事件 F 。当然，即使事件 L 不发生，事件 F 也可能发生（例如，如果连续 s 次或更多次掷出正面的事件跨越了两组之间的边界），因此我们有 Pr L Pr fF g D O.1/n/。由于 Pr fLg C Pr L D 1，我们有

$$\begin{aligned}
\Pr\{L\} &= 1 - \Pr\{\overline{L}\} \\
&\geq 1 - \Pr\{F\} \\
&= 1 - O(1/n) .
\end{aligned}$$

也就是说，最长连续线等于或超过 b.lg n//2c 的概率是

$$\sum_{j=\lfloor(\lg n)/2\rfloor}^{n} \Pr\{L_j\} \geq 1 - O(1/n) . \tag{5.13}$$

现在，我们可以计算最长条纹预期长度的下限，从公式 (5.11) 开始，按照与分析上限类似的方式进行：

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{\lfloor(\lg n)/2\rfloor-1} j \Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor}^{n} j \Pr\{L_j\} \\
&\geq \sum_{j=0}^{\lfloor(\lg n)/2\rfloor-1} 0 \cdot \Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor}^{n} \lfloor(\lg n)/2\rfloor \Pr\{L_j\} \\
&= 0 \cdot \sum_{j=0}^{\lfloor(\lg n)/2\rfloor-1} \Pr\{L_j\} + \lfloor(\lg n)/2\rfloor \sum_{j=\lfloor(\lg n)/2\rfloor}^{n} \Pr\{L_j\} \\
&\geq 0 + \lfloor(\lg n)/2\rfloor (1 - O(1/n)) \qquad \text{(by inequality (5.13))} \\
&= \Omega(\lg n) .
\end{aligned}$$

与生日悖论一样，我们可以使用指示随机变量进行更简单但近似的分析。我们不会确定最长条纹的预期长度，而是找到至少具有给定长度的条纹的预期数量。让 X $_{ik}$ D I fA $_{ik}$ g 成为与从第 i 个硬币开始的长度至少为 k 的正面条纹相关的指示随机变量。要计算此类条纹的总数，需要

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

取期望值并利用期望的线性，我们有

$$\begin{aligned} \mathrm{E}[X_k] &= \mathrm{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\ &= \sum_{i=1}^{n-k+1} \mathrm{E}[X_{ik}] \\ &= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\ &= \sum_{i=1}^{n-k+1} \frac{1}{2^k} \\ &= \frac{n-k+1}{2^k} . \end{aligned}$$

通过代入各种 k 值，我们可以计算长度至少为 k 的条纹的预期数量。如果这个预期数量很大（远大于 1），那么我们预计会出现许多长度为 k 的条纹，而且出现一条条纹的概率很高。如果这个预期数量很小（远小于 1），那么我们预计会看到很少长度为 k 的条纹，而且出现一条条纹的概率很低。如果 k D c lg n，对于某个正常数 c，我们得到

$$\begin{aligned} \mathrm{E}[X_{c \lg n}] &= \frac{n - c \lg n + 1}{2^{c \lg n}} \\ &= \frac{n - c \lg n + 1}{n^c} \\ &= \frac{1}{n^{c-1}} - \frac{(c \lg n - 1)/n}{n^{c-1}} \\ &= \Theta(1/n^{c-1}) . \end{aligned}$$

如果 c 很大，长度为 c lg n 的条纹的预期数量就很小，我们得出结论，它们不太可能发生。另一方面，如果 c D 1/2，那么我们

得到 E ŒX $_{(1/2)\lg n}$ D ，.1/n $^{1/2-1}$ / D ，.n $^{1/2}$ /，并且我们预期会存在许多长度为 .1/2/ lg n 的条纹。因此，这种长度的条纹很可能会出现一次。我们可以得出结论，最长条纹的预期长度为 ，.lg n/。

5.4.4 在线招聘问题

作为最后一个示例，让我们考虑招聘问题的一个变体。现在假设您不想面试所有候选人以找到最佳候选人。您还希望避免在找到越来越好的申请人时雇用和打电话。相反，您愿意接受接近最佳的候选人，以换取只雇用一次。您必须遵守公司的一项要求：每次面试后，您必须立即向申请人提供职位或立即拒绝申请人。最小化面试次数和最大化雇用候选人的质量之间的权衡是什么？

我们可以按以下方式对这个问题进行建模。在会见一位申请人之后，你可以给每个人打分。令 *score*.i / 表示你给第 i 位申请人的分数，并假设没有两个申请人会得到相同的分数。在见过 j 位申请人之后，你知道 j 位申请人中哪一位的分数最高，但不知道剩余的 n j 位申请人中是否有人会得到更高的分数。你决定采用以下策略：选择一个正整数 k < n，面试然后拒绝前 k 位申请人，然后聘用得分高于所有前面申请人的第一个申请人。如果结果证明最有资格的申请人是前 k 位面试的申请人，那么你就会聘用第 n 位申请人（即最后面试的申请人）。我们在过程 ONLINE-MAXIMUM.k; n / 中正式化了这一策略，该过程返回你希望聘用的候选人的索引。

```
在线最大值.k; n/

1 best-score D 1 2 for i D 1 到 k 3 if
score.i / > best-score 4 best-score D
score.i / 5 for i D k C 1 到 n 6 if score.i /
> best-score 7 return i 8 return n
```

如果我们确定了 k 的每个可能值，你聘用最合格应聘者的概率，那么你就可以选择最佳的 k 并使用该值实施策略。目前，假设 k 是固定的。让

M.j / D max f*score*.i / W 1 i j g 表示申请人 1 到 j 中的最高分数。令 S 表示您成功选择最有资格的申请人的事件，令 S_i 表示当最有资格的申请人是第 i 位面试者时您成功的事件。由于各个 S_i 是不相交的，因此我们有 Pr fS g D P $_{i=1}^{n}$ Pr fS $_i$ g。注意到当最有资格的申请人是前 k 位之一时您永远不会成功，我们有 Pr fS $_i$ g D 0，其中 i D 1; 2; : : : ; k。因此，我们得到

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} \ . \tag{5.14}$$

现在我们计算 Pr fS $_i$ g。当最有资格的应聘者是第 i 位时，为了成功，必须满足两件事。首先，最有资格的应聘者必须处于位置 i，我们将此事件表示为 B_i。其次，算法不得选择位置 k C 1 至 i 1 中的任何应聘者，这仅当对于每个满足 k C 1 j i 1 的 j，第 6 行发现 *score*.j / < *best-score* 时才会发生。（因为分数是唯一的，我们可以忽略 *score*.j / D *best-score* 的可能性。）换句话说，所有 *score*.k C 1/ 至 *score*.i 1/ 的值都必须小于 M.k/。如果有任何大于 M.k/ 的事件，算法就返回第一个较大的事件的索引。我们用 O_i 来表示位置 k(1 至 i 1) 中的申请者均未被选中的事件。幸运的是，两个事件 B_i 和 O_i 是独立的。事件 O_i 只取决于位置 1 至 i 1 中的值的相对顺序，而 B_i 只取决于位置 i 中的值是否大于所有其他位置中的值。位置 1 至 i 1 中的值的顺序不影响位置 i 中的值是否大于所有值，位置 i 中的值也不影响位置 1 至 i 1 中的值的顺序。因此，我们可以应用第 1188 页的公式 (C.17) 得到

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

由于最大值在 n 个位置中的任意一个位置出现的可能性相同，因此我们有 Pr fB $_i$ g D 1/n。要使事件 O $_i$ 发生，位置 1 至 i 1 中的最大值（在 i 1 个位置中的任意一个位置出现的可能性相同）必须位于前 k 个位置之一。因此，Pr fO $_i$ g D k/.i 1/ 和 Pr fS $_i$ g D k/.n.i 1//。利用公式 (5.14)，我们有

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

$$= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1}$$
$$= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i} .$$

我们用积分来近似地限制这个总和。根据第 1150 页的不等式 (A.19)，我们得到

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

评估这些有限积分可以得到界限

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

这为 Pr fS g 提供了一个相当严格的界限。因为您希望最大限度地提高成功概率，所以我们将重点放在选择最大化 Pr fS g 下限的 k 值上。（此外，下限表达式比上限表达式更容易最大化。）对表达式 .k/n/.ln n ln k/ 进行 k 微分，我们得到

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

将此导数设置为 0，我们可以看到，当 ln k D ln n 1 D ln.n/e 时，或者当 k D n/e 时，您可以最大化概率的下限。因此，如果您以 k D n/e 实施我们的策略，您将至少有 1/e 的概率成功聘用最合格的申请人。

练习

5.4-1

房间里必须有多少人，才会使某人的生日和您同一天的概率至少达到 1/2？房间里必须有多少人，才会使至少两个人的生日都是 7 月 4 日的概率大于 1/2？

5.4-2

房间里必须有多少人才能使两个人生日相同的概率至少达到 0:99？对于这么多人来说，生日相同的人的预期数量是多少？

5.4-3
你把球扔进 b 个箱子里，直到某个箱子里有两个球。每次抛球都是独立的，每个球落入任意箱子的概率都相同。抛球的预期次数是多少？

5.4-4
对于生日悖论的分析，生日相互独立是否重要，还是两两独立就足够了？请证明你的答案。

5.4-5
为了使得*three* 人们的生日有可能是同一天，应该邀请多少人参加聚会？

5.4-6
一个 k 字符串（定义见第 1179 页）在大小为 n 的集合中形成 k 排列的概率是多少？这个问题与生日悖论有何关联？

5.4-7
你将 n 个球扔进 n 个箱子里，每次扔球都是独立的，球落入任意箱子的概率都相同。空箱子的预期数量是多少？只有一个球的箱子的预期数量是多少？

5.4-8
通过证明在 n 枚公平硬币中，出现长度为 lg n 2 lg lg n 的连续正面的概率至少为 1 1/n，进一步明确条纹长度的下限。

问题

5-1Probabilistic counting
使用 b 位计数器，我们通常只能计数到 2^b 1。使用 R. Morris 的 ***probabilistic counting***，我们可以计数到更大的值，但代价是损失一些精度。

我们让计数器值 i 表示 i D 0; 1; : : :; 2^b 1 的计数 n_i，其中 n_i 形成一个递增的非负值序列。我们假设计数器的初始值为 0，表示计数 n_0 D 0。INCREMENT 操作以概率方式对包含值 i 的计数器进行操作。如果 i D 2^b 1，则该操作报告溢出错误。否则，INCREMENT 操作以概率 $1/(n_{i+1} - n_i)$ 将计数器增加 1，并以概率 $1 - 1/(n_{i+1} - n_i)$ 保持不变。

如果我们对所有的 i 0 选择 n $_i$ D i，那么计数器就是一个普通的计数器。如果我们对 i > 0 选择 n $_i$ D 2 $^{i-1}$ ，或者对第 i 个斐波那契数选择 n $_i$ D F $_i$ （，就会出现更有趣的情况4参见公式（第 69 页的 3.31））。

对于这个问题，假设 n $_{2^b-1}$ 足够大，以至于发生溢出错误的概率可以忽略不计。

a.证明执行 n 次 I NCREMENT 操作后计数器所表示的期望值恰好是 n。

b.计数器所表示计数的方差分析取决于 n_i 的序列。让我们考虑一个简单的情况：n_i D 100i，对于所有的 i ≈ 0。估计在执行 n 次 I NCREMENT 操作后寄存器所表示的值的方差。

5-2Searching an unsorted array

本问题考察在由 n 个元素组成的未排序数组 A 中查找值 x 的三种算法。

考虑以下随机化策略：在 A 中随机选取一个索引 i。如果 AŒi ≠ D x，则终止；否则，继续搜索，在 A 中随机选取一个新的索引。继续在 A 中随机选取索引，直到找到一个索引 j，使得 AŒj ≠ D x 或直到 A 中的每个元素都已检查。此策略可能会多次检查给定元素，因为它每次都从整个索引集中选取。

a.编写 RANDOM-SEARCH 程序的伪代码来实现上述策略。确保当 A 中的所有索引都被选中时，算法终止。

b.假设只有一个索引 i 使得 AŒi = D x 。在找到 x 并且 RANDOM-SEARCH 终止之前，必须选取 A 中预期多少个索引？***c.*** 将你的解决方案推广到部分 (b)，假设有 k 1 个索引 i 使得 AŒi = D x 。在找到 x 并且 RANDOM-SEARCH 终止之前，必须选取 A 中预期多少个索引？你的答案应该是 n 和 k 的函数。***d.*** 假设没有索引 i 使得 AŒi = D x 。在检查完 A 的所有元素并且 RANDOM-SEARCH 终止之前，必须选取 A 中预期多少个索引？

现在考虑一个确定性线性搜索算法。我们称该算法为确定性搜索算法，它按顺序在 A 中搜索 x，考虑 AŒ1 ; AŒ2 ;

AŒ3 ; : : : ; AŒn 直到找到 AŒi D x 或到达数组末尾。假设输入数组的所有可能排列都是等概率的。

e.假设只有一个指标 i 使得 AŒi = D x 。DETERMINISTIC-SEARCH 的平均运行时间是多少？DETERMINISTIC-SEARCH 的最坏情况运行时间是多少？***f.*** 将你的解决方案推广到部分 (e)，假设有 k 1 个指标 i 使得 AŒi = D x 。DETERMINISTIC-SEARCH 的平均运行时间是多少？DETERMINISTIC-SEARCH 的最坏情况运行时间是多少？你的答案应该是 n 和 k 的函数。***g.*** 假设没有指标 i 使得 AŒi = D x 。DETERMINISTIC-SEARCH 的平均运行时间是多少？DETERMINISTIC-SEARCH 的最坏情况运行时间是多少？

最后，考虑一个随机算法 SCRAMBLE-SEARCH，该算法首先随机排列输入数组，然后对结果排列数组运行上面给出的确定性线性搜索。

h.令 k 为满足 AŒi D x 的索引 i 的数量，给出 k D 0 和 k D 1 情况下 SCRAMBLE-SEARCH 的最坏情况和预期运行时间。推广您的解决方案以处理 k ¹ 1 的情况。

i.您会使用这三种搜索算法中的哪一种？解释您的答案。

章节注释

Bollob´as [65]、Hofri [223] 和 Spencer [420] 包含大量高级概率技术。Karp [249] 和 Rabin [372] 讨论并概述了随机算法的优势。Motwani 和 Raghavan [336] 的教科书对随机算法进行了广泛的论述。

RANDOMLY-PERMUTE 程序由 Durstenfeld [128] 提出，基于 Fisher 和 Yates 早期的程序 [143，第 34 页]。

招聘问题的几种变体已得到广泛研究。这些问题通常被称为“秘书问题”。该领域的研究范例包括 Ajtai、Meggido 和 Waarts 的论文 [11] 以及 Kleinberg 的另一篇论文 [258]，后者将秘书问题与在线广告拍卖联系起来。

Part II Sorting and Order Statistics

介绍

本部分介绍了解决以下***sorting problem***的几种算法：

输入：一个由 n 个数字组成的序列 ha $_1$; a $_2$; : : : ; a $_n$ i。

输出：输入序列的排列（重新排序）ha $'_1$; a $'_2$; : : : ; a $'_n$ i，使得 a $'_1$ a $'_2$ a $'_n$。

输入序列通常是一个 n 元素数组，尽管它也可以以其他方式表示，例如链表。

数据结构

实际上，要排序的数字很少是孤立的值。每个数字通常都是称为 ***record*** 的数据集合的一部分。每条记录都包含一个 ***key***，即要排序的值。记录的其余部分由 ***satellite data*** 组成，通常与键一起携带。实际上，当排序算法对键进行排列时，它也必须对附属数据进行排列。如果每条记录都包含大量附属数据，则为了最大限度地减少数据移动，通常最好排列指向记录的指针数组，而不是记录本身。

从某种意义上说，这些实现细节将算法与成熟的程序区分开来。排序算法描述了确定排序顺序的 *method*，无论排序的是单个数字还是包含许多字节卫星数据的大型记录。因此，当关注排序问题时，我们通常假设输入仅由数字组成。将对数字进行排序的算法转换为对记录进行排序的程序在概念上很简单，尽管在给定的工程情况下，其他细微之处可能会使实际的编程任务成为一项挑战。

为什么要排序？

许多计算机科学家认为排序是算法研究中最基本的问题。原因如下：

有时应用程序本身需要对信息进行排序。例如，为了准备客户报表，银行需要按支票号对支票进行排序。• 算法通常将排序用作关键子程序。例如，渲染相互叠加的图形对象的程序可能必须根据 “above” 关系对对象进行排序，以便它可以从下到上绘制这些对象。我们将在本文中看到许多使用排序作为子程序的算法。

我们可以从各种各样的排序算法中汲取经验，它们采用了丰富的技术。事实上，算法设计过程中使用的许多重要技术都出现在多年来开发的排序算法中。这样一来，排序也是一个具有历史意义的问题。

我们可以证明排序的一个非平凡下界（我们将在第 8 章中这样做）。由于最佳上界与下界渐近匹配，我们可以得出结论，我们的某些排序算法是渐近最优的。此外，我们可以使用排序的下界来证明各种其他问题的下界。

在实施排序算法时，许多工程问题浮出水面。特定情况下最快的排序程序可能取决于许多因素，例如关于密钥和卫星数据的先验知识、主机的内存层次结构（缓存和虚拟内存）以及软件环境。许多这些问题最好在算法层面处理，而不是通过“调整”代码。

排序算法

我们在第 2 章中介绍了两种对 n 个实数进行排序的算法。插入排序在最坏情况下需要 '.n ^{2} / 时间。但是，由于其内部循环紧凑，因此对于较小的输入量，它是一种快速排序算法。此外，与归并排序不同，它对 ***in place*** 进行排序，这意味着输入数组中最多有常数个元素存储在数组外部，这对于空间效率是有利的。归并排序具有更好的渐近运行时间，'.n lg n/，但它使用的 MERGE 过程不是原地运行的。（我们将在第 26.3 节中看到归并排序的并行化版本。）

本部分介绍了另外两种对任意实数进行排序的算法。第 6 章介绍的堆排序在 O.n lg n/ 时间内对 n 个数字进行排序。它使用一种称为堆的重要数据结构，该结构也可以实现优先级队列。

第 7 章中的快速排序也对 n 个数字进行就地排序，但其最坏情况运行时间为 ，.n ²/。然而，其预期运行时间为 ，.n lg n/，而且在实践中，它通常比堆排序表现更好。与插入排序一样，快速排序的代码紧凑，因此其运行时间中的隐藏常数因子很小。它是一种流行的大型数组排序算法。

插入排序、归并排序、堆排序和快速排序都是比较排序：它们通过比较元素来确定输入数组的排序顺序。第 8 章首先介绍决策树模型，以研究比较排序的性能限制。利用该模型，我们证明了任何比较排序在 n 个输入上最坏情况运行时间的下限为 .n lg n/，从而表明堆排序和归并排序是渐近最优的比较排序。

第 8 章接着说明，如果一种算法可以通过比较元素以外的其他方式收集有关输入排序顺序的信息，我们也许能够突破 .n lg n/ 这个下限。例如，计数排序算法假设输入数字属于集合 f0; 1; : : : ; kg。通过使用数组索引作为确定相对顺序的工具，计数排序可以在 .k C n/ 时间内对 n 个数字进行排序。因此，当 k D O.n/ 时，计数排序的运行时间与输入数组的大小成线性关系。相关算法基数排序可用于扩展计数排序的范围。如果有 n 个整数需要排序，每个整数有 d 位数字，并且每个数字最多可以取 k 个可能值，则基数排序可以在 .d.n C k// 时间内对数字进行排序。当 d 为常数且 k 为 O.n/ 时，基数排序的运行时间为线性时间。第三种算法是桶排序，它需要知道输入数组中数字的概率分布。它可以对半开区间 Œ0; 1/ 中均匀分布的 n 个实数进行排序，平均时间为 O.n/ 次。

下页的表格总结了第 2 章和第 6 章中的排序算法的运行时间。与往常一样，n 表示要排序的项目数。对于计数排序，要排序的项目是集合 f0; 1; : : :; k g 中的整数。对于基数排序，每个项目都是一个 d 位数字，其中每位数字有 k 个可能值。对于桶排序，我们假设键是半开区间 Œ0; 1/ 内均匀分布的实数。最右边的一列给出平均情况或预期运行时间，表明当它与最坏情况运行时间不同时，给出的是哪一个。我们省略了堆排序的平均运行时间，因为我们在本书中不对其进行分析。

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

订单统计

一组 n 个数字的第 i 阶统计量是该集合中第 i 个最小的数字。当然，您可以通过对输入进行排序并对输出的第 i 个元素进行索引来选择第 i 阶统计量。由于不对输入分布进行任何假设，此方法的运行时间为 .n lg n/ 时间，如第 8 章中证明的下限所示。

第 9 章展示了如何在 O.n/ 时间内找到第 i 个最小元素，即使元素是任意实数。我们提出了一种随机算法，该算法具有严格的伪代码，在最坏情况下运行时间为 ，.n 2 / 时间，但预期运行时间为 O.n/。我们还给出了一种更复杂的算法，在最坏情况下运行时间为 O.n/。

背景

虽然本部分的大部分内容并不依赖复杂的数学知识，但有些章节确实需要数学知识。特别是，快速排序、桶排序和顺序统计算法的分析使用了概率（附录 C 中对此进行了回顾），以及第 5 章中关于概率分析和随机算法的材料。

6 堆排序

本章介绍了另一种排序算法：堆排序。与归并排序类似，但与插入排序不同，堆排序的运行时间为 O.n lg n/。与插入排序类似，但与归并排序不同，堆排序是就地排序：任何时候，只有常数个数组元素存储在输入数组之外。因此，堆排序结合了我们已经讨论过的两种排序算法的更好属性。

堆排序还引入了另一种算法设计技术：使用数据结构（在本例中我们称之为“堆，”）来管理信息。堆数据结构不仅对堆排序有用，而且还可以构成一个高效的优先级队列。堆数据结构将在后面的章节中的算法中再次出现。

术语“heap”最初是在堆排序的上下文中创造的，但后来它开始指代“垃圾收集存储，”例如编程语言 Java 和 Python 提供的。请不要混淆。堆数据结构是 *not* 垃圾收集存储。本书始终使用术语“heap”来指代数据结构，而不是存储类。

6.1 堆

(binary) heap 数据结构是一个数组对象，我们可以将其视为近乎完全的二叉树（参见 B.5.3 节），如图 6.1 所示。树的每个节点都与数组的一个元素相对应。树的所有层级都完全填充，除了最低层之外，最低层从左向上填充到某个点。表示堆的数组 AŒ1 W n 是一个具有属性 A: *heap-size* 的对象，该属性表示堆中有多少元素存储在数组 A 中。也就是说，尽管 AŒ1 W n 可能包含数字，但只有 AŒ1 W A: *heap-size* 中的元素（其中 0 A: *heap-size* n）才是堆的有效元素。如果 A: *heap-size* D 0，则堆为空。树的根是 AŒ1 ，给定节点的索引 i,

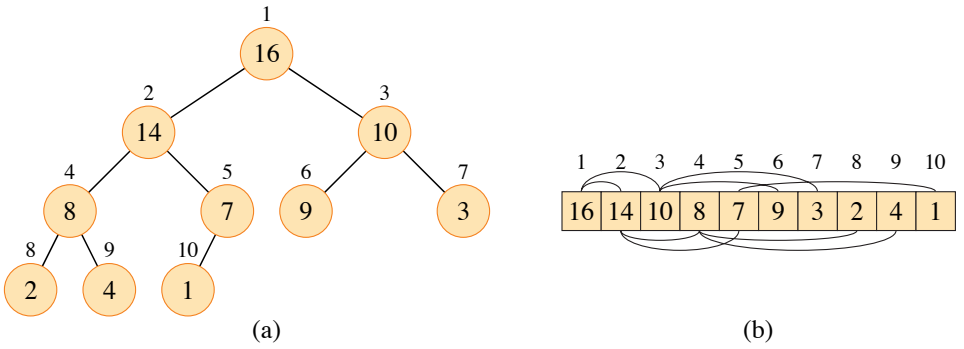


图 6.1 最大堆可视为 (a) 二叉树和 (b) 数组。树中每个节点圆圈内的数字是该节点存储的值。节点上方的数字是数组中相应的索引。数组上方和下方是显示父子关系的线，父节点始终位于子节点的左侧。树的高度为 3，索引 4 处的节点（值为 8）的高度为 1。

有一个简单的方法可以使用单行程序 PARENT、LEFT 和 RIGHT 来计算其父节点、左孩子节点和右孩子的索引。

PARENT.i /

1 返回 bi/2c

左.i /

1 返回 2i

右.i /

1 返回 2i C 1

在大多数计算机上，LEFT 过程只需将 i 的二进制表示左移一位，即可在一条指令中计算出 2i。同样，RIGHT 过程只需将 i 的二进制表示左移一位，然后加 1，即可快速计算出 2i C 1。PARENT 过程只需将 i 右移一位，即可计算出 bi/2c。良好的堆排序实现通常将这些过程实现为宏或内联过程。

二叉堆有两种类型：最大堆和最小堆。在这两种类型中，节点中的值都满足 ***heap property***，具体情况取决于堆的类型。在 ***max-heap*** 中，***max-heap property*** 表示对于除根之外的每个节点 i，

AŒPARENT.i / AŒi ;

也就是说，一个节点的值最多是其父节点的值。因此，最大堆中的最大元素存储在根节点，并且以节点为根的子树包含的值不大于节点本身包含的值。***min-heap*** 以相反的方式组织：***min-heap property*** 是对于除根节点以外的每个节点 i,

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

最小堆中的最小元素位于根。

堆排序算法使用最大堆。最小堆通常实现优先级队列，我们将在 6.5 节中讨论。我们将精确指定对于任何特定应用是否需要最大堆或最小堆，并且当属性适用于最大堆或最小堆时，我们只使用术语 “ 堆。”

将堆视为一棵树，我们将堆中节点的 ***height*** 定义为从节点到叶子的最长简单向下路径上的边数，并将堆的高度定义为其根的高度。由于 n 个元素的堆基于完全二叉树，因此其高度为 '.lg n/'（参见练习 6.1-2）。我们将看到，堆上的基本操作最多与树的高度成比例，因此需要 0.lg n/' 时间。本章的其余部分将介绍一些基本过程，并展示如何在排序算法和优先级队列数据结构中使用它们。

MAX-HEAPIFY 过程运行时间为 O.lg n/ s，是维持最大堆特性的关键。• BUILD-MAX-HEAP 过程运行时间为线性时间，它从无序输入数组生成最大堆。• HEAPSORT 过程运行时间为 O.n lg n/ s，它对数组进行就地排序。• 过程 MAX-HEAP-INSERT、MAX-HEAP-EXTRACT-MAX、MAX-HEAP-INCREASE-KEY 和 MAX-HEAP-MAXIMUM 允许堆数据结构实现优先级队列。它们运行时间为 O.lg n/ s，加上插入优先级队列的对象与堆中索引之间的映射时间。

练习

6.1-1
高度为 h 的堆中元素的最小数量和最大数量是多少？

6.1-2证明一个 n 元素堆的高度为 blg nc。

6.1-3
证明在最大堆的任何子树中，子树的根包含该子树中任何地方出现的最大值。

6.1-4
假设所有元素都是不同的，那么最大堆中的最小元素可能位于哪里？

6.1-5对于 2 k bn/2c，假设所有元素都是不同的，那么第 k 大元素可能位于最大堆的哪一层？

6.1-6
按排序顺序排列的数组是最小堆吗？

6.1-7
值为 h33；19；20；15；13；10；2；13；16；12i 的数组是最大堆吗？

6.1-8
证明，使用用于存储 n 元素堆的数组表示，叶子节点是通过 bn/2c C 1; bn/2 c C 2; : : : ; n 索引的节点。

6.2 维护堆属性

上页中的 MAX-HEAPIFY 过程维护最大堆属性。其输入是具有 *heap-size* 属性的数组 A 和数组中的索引 i。调用它时，MAX-HEAPIFY 假定以 LEFT.i / 和 RIGHT.i / 为根的二叉树是最大堆，但 AŒi 可能小于其子节点，从而违反最大堆属性。MAX-HEAPIFY 让 AŒi “ 处的值在最大堆中向下浮动”，以便以索引 i 为根的子树遵循最大堆属性。

图 6.2 说明了 MAX-HEAPIFY 的操作。每一步都确定元素 AŒi 、AŒLEFT.i / 和 AŒRIGHT.i / 中最大的一个，并将最大元素的索引存储在 *largest* 中。如果 AŒi 最大，则以节点 i 为根的子树已经是最大堆，不需要执行任何其他操作。否则，两个子节点之一包含最大元素。位置 i 和 *largest* 交换其内容，这导致节点 i 及其子节点满足最大堆属性。但是，以 *largest* 为索引的节点的值刚刚减少，因此以 *largest* 为根的子树可能违反最大堆属性。因此，MAX-HEAPIFY 在该子树上递归调用自身。

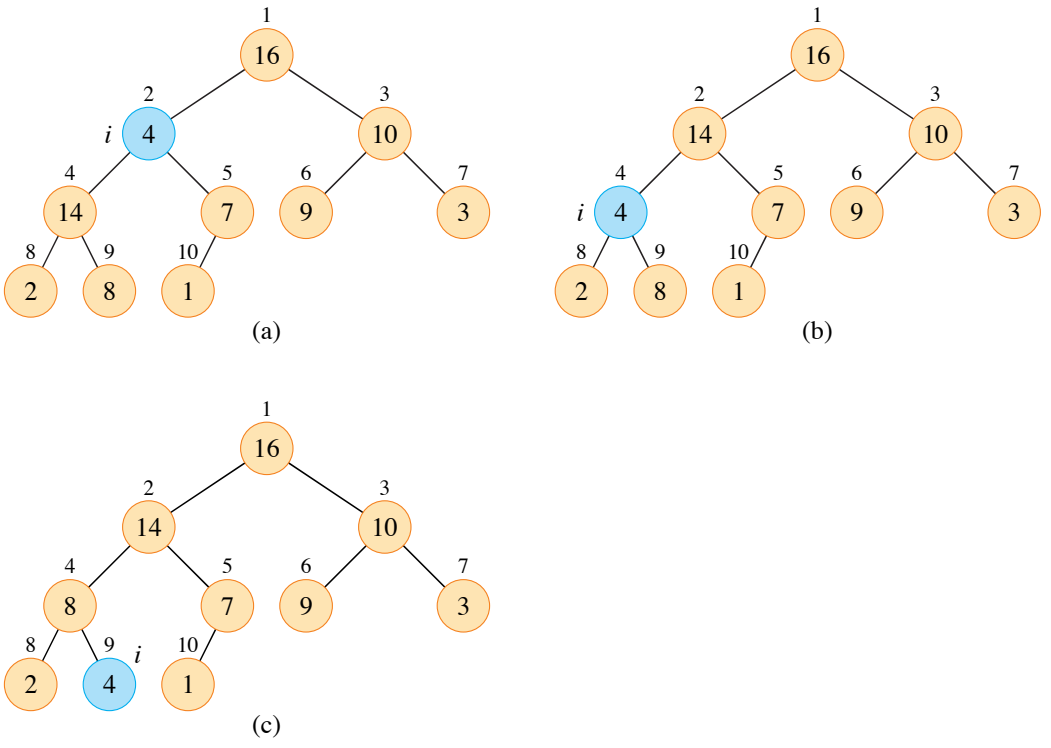


图 6.2 MAX-HEAPIFY.A; 2/ 的操作，其中 A: *heap-size* D 10。可能违反最大堆性质的节点以蓝色显示。（a）初始配置，节点 i D 2 处的 ACŒ2 违反最大堆性质，因为它不大于两个子节点。在 (b) 中，通过将 ACŒ2 与 ACŒ4 交换，节点 2 的最大堆性质得以恢复，这会破坏节点 4 的最大堆性质。递归调用 MAX-HEAPIFY.A; 4/ 现在具有 i D 4。交换 ACŒ4 和 ACŒ9 之后，如 (c) 所示，节点 4 得到修复，递归调用 MAX-HEAPIFY.A; 9/ 不会对数据结构产生进一步的改变。

MAX-HEAPIFY .A; i /

1 l D LEFT.i / 2 r D RIGHT.i / 3 如果 l A: *heap-size* 且 ACŒl > ACŒi 4 *largest* D l 5 else *largest* D i 6 如果 r A: *heap-size* 且 ACŒr > ACŒ*largest* 7 *largest* D r 8 如果 *largest* ¤ i 9 将 ACŒi 与 ACŒ*largest* 交换

10 MAX-HEAPIFY .A; *largest* /

为了分析 MAX-HEAPIFY，让 T.n/ 表示该过程在最大大小为 n 的子树上所花费的最坏情况运行时间。对于以给定节点 i 为根的树，运行时间是“ .1/ ”的时间来修正元素 AŒi 、AŒLEFT.i / 和 AŒRIGHT.i / 之间的关系，加上在以节点 i 的一个子节点为根的子树上运行 MAX-HEAPIFY 的时间（假设发生递归调用）。每个子树的大小最多为 2n/3（参见练习 6.2-2），因此我们可以用递归公式来描述 MAX-HEAPIFY 的运行时间

$$T(n) \le T(2n/3) + \Theta(1) . \tag{6.1}$$

根据主定理（第 102 页的定理 4.1）的案例 2，此递归的解为 T .n/ D O.lg n/ 。或者，我们可以将 MAX-HEAPIFY 在高度为 h 的节点上的运行时间表征为 O.h/。

练习

6.2-1
以图 6.2 为模型，说明 MAX-HEAPIFY .A; 3/ 对数组 A D h27; 17; 3; 16; 13; 1 0; 1; 5; 7; 12; 4; 8; 9; 0i 的操作。

6.2-2
证明 n 节点堆的根的每个子节点都是包含最多 2n/3 个节点的子树的根。最小常数 α是多少，使得每个子树最多有 αn 个节点？这对递归 (6.1) 及其解决方案有何影响？

6.2-3
从过程 MAX-HEAPIFY 开始，编写过程 MIN-HEAPIFY .A; i / 的伪代码，该过程对最小堆执行相应的操作。MIN-HEAPIFY 的运行时间与 MAX-HEAPIF Y 的运行时间相比如何？

6.2-4
当元素 AŒi 大于其子元素时，调用 MAX-HEAPIFY .A; i / 会产生什么效果？

6.2-5
调用 MAX-HEAPIFY .A; i / for i > A: *heap-size*/2 的效果是什么？

6.2-6
MAX-HEAPIFY 的代码在常数因子方面非常高效，除了第 10 行中的递归调用，某些编译器可能会生成低效的代码。编写一个高效的 MAX-HEAPIFY，使用迭代控制结构（循环）而不是递归。

6.2-7
证明 MAX-HEAPIFY 在大小为 n 的堆上的最坏情况运行时间为 -1.lg n/。（*Hint:* 对于具有 n 个节点的堆，给出节点值，使得从根到叶的简单路径上的每个节点都递归调用 MAX-HEAPIFY。）

6.3 构建堆

过程 BUILD-MAX-HEAP 通过自下而上调用 MAX-HEAPIFY 将数组 AŒ1 W n 转换为最大堆。练习 6.1-8 说子数组 AŒbn/2c C 1 W n 中的元素都是树的叶子，因此每个元素都是一个元素堆。BUILD-MAX-HEAP 遍历树的剩余节点并对每个节点运行 MAX-HEAPIFY。图 6.3 显示了 BUILD-MAX-HEAP 操作的一个示例。

构建-最大-堆 .A; n/

1 A: *heap-size* D n 2 对于 i D
bn/2c 向下到 1 3 MAX-HEA
PIFY .A; i /

为了说明 BUILD-MAX-HEAP 为何能正常工作，我们使用以下循环不变量：

> 在第 233 行的 for 循环每次迭代开始时，每个节点 i C 1; i C 2; : : : ; n 都是最大堆的根。

我们需要证明这个不变量在第一次循环迭代之前为真，循环的每次迭代都保持不变变量，循环终止，并且不变量提供了一个有用的属性来显示循环终止时的正确性。
初始化：在循环的第一次迭代之前，i D bn/2c。每个节点 bn/2c C 1; bn/2c C 2; : : : ; n 都是叶子，因此是平凡最大堆的根。
维护：要确保每次迭代都保持循环不变量，请观察节点 i 的子节点的编号高于 i 。因此，根据循环不变量，它们都是最大堆的根。这正是调用 MAX-HEAPIFY .A; i / 使节点 i 成为最大堆根所需的条件。此外，MAX-HEAPIFY 调用保留了节点 i C 1; i C 2; : : : ; n 都是最大堆根的属性。在 for 循环更新中减少 i 会为下一次迭代重新建立循环不变量。

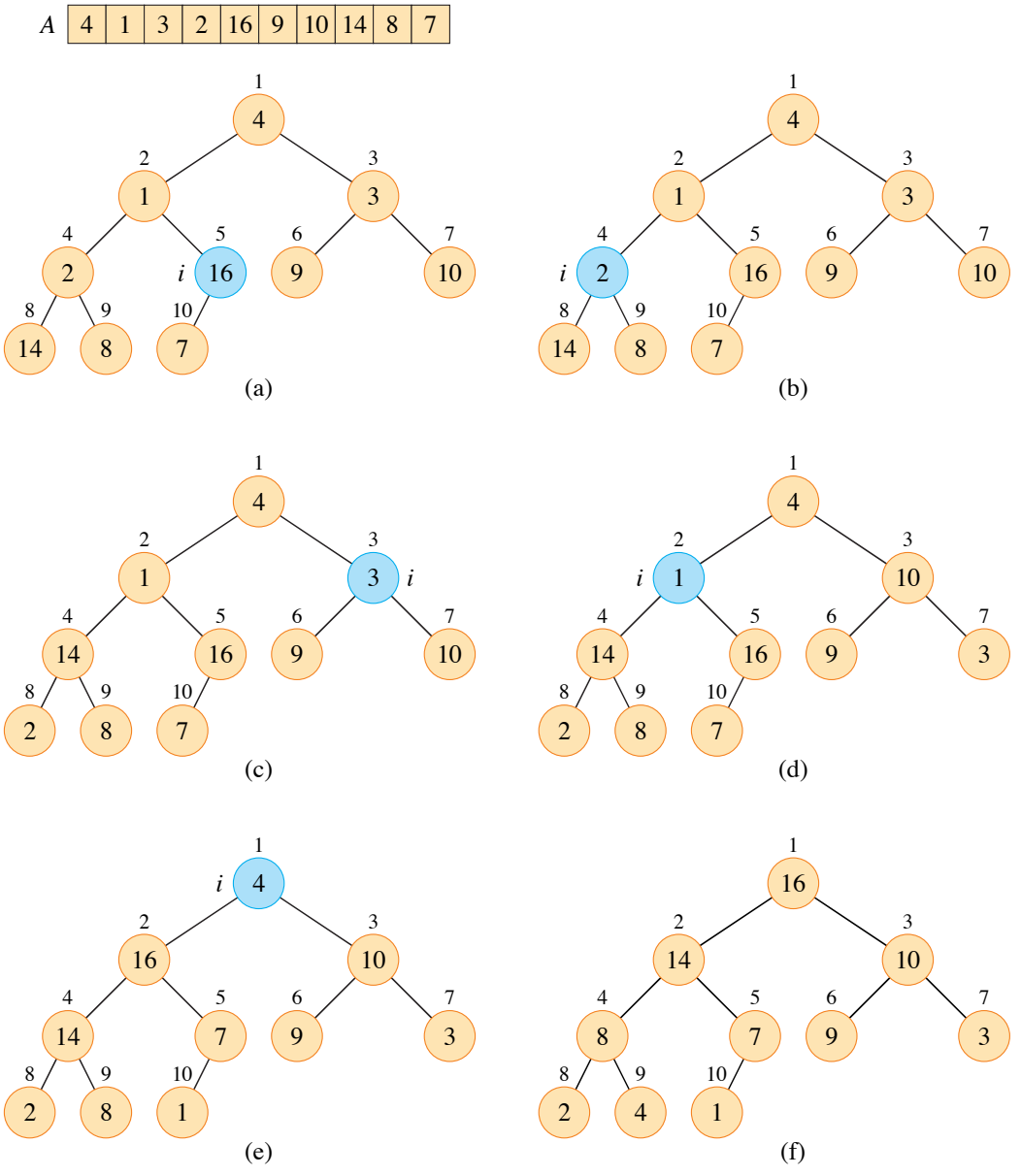


图 6.3 BUILD-MAX-HEAP 的操作，显示了在 BUILD-MAX-HEAP 第 3 行调用 MAX-HEAPIFY 之前的数据结构。每次迭代中索引为 i 的节点以蓝色显示。（a）10 个元素的输入数组 A 及其表示的二叉树。在调用 MAX-HEAPIFY.A; i/ 之前，循环索引 i 指的是节点 5。（b）产生的数据结构。下一次迭代的循环索引 i 指的是节点 4。（c）–（e）BUILD-MAX-HEAP 中 for 循环的后续迭代。观察到每当在节点上调用 MAX-HEAPIFY 时，该节点的两个子树都是最大堆。（f）BUILD-MAX-HEAP 完成后的最大堆。

终止：循环恰好进行 bn/2c 次迭代，因此终止。终止时，i D 0。根据循环不变量，每个节点 1；2；：：：；n 都是最大堆的根。具体来说，节点 1 是。

我们可以按如下方式计算 B UILD-MAX-HEAP 运行时间的简单上限。每次调用 MAX-HEAPIFY 需要花费 O.lg n/ 时间，而 BUILD-MAX-HEAP 进行了 O.n/ 次这样的调用。因此，运行时间为 O.n lg n/。这个上限虽然正确，但并不尽如人意。

我们可以通过观察 M AX-HEAPIFY 在节点上运行的时间随树中节点的高度而变化，并且大多数节点的高度都很小，从而得出更严格的渐近界限。我们更严格的分析依赖于以下属性：n 元素堆具有高度 blg nc（参见练习 6.1-2）和最多 Û n/2 $_{h+1}$ 个节点，高度为任意 h（参见练习 6.3-4）。

在高度为 h 的节点上调用 MAX-HEAPIFY 所需的时间为 O.h/。令 c 为渐近符号中隐含的常数，我们可以将 BUILD-MAX-HEAP 的总成本表示为 P $_{h=0}^{\lfloor \lg n \rfloor}$ Û n/2 $_{h+1}$ ch 的上限。如练习 6.3-2 所示，对于 0 h blg nc，我们有 Û n/2 $_{h+1}$ 1/2。由于对于任何 x ¹ 1/2，dx e 2x，我们有 Û n/2 $_{h+1}$ n/2 ^{h}。因此我们得到

$$
\begin{aligned}
&\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \\
&\qquad \le \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \\
&\qquad = cn \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \\
&\qquad \le cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\
&\qquad \le cn \cdot \frac{1/2}{(1-1/2)^2} \quad \text{(by equation (A.11) on page 1142 with } x = 1/2) \\
&\qquad = O(n) \, .
\end{aligned}
$$

因此，我们可以在线性时间内从无序数组构建最大堆。

要构建最小堆，请使用过程 BUILD-MIN-HEAP，它与 BUILD-MAX-HEAP 相同，但第 3 行对 MAX-HEAPIFY 的调用被对 MIN-HEAPIFY 的调用取代（参见练习 6.2-3）。BUILD-MIN-HEAP 在线性时间内从无序线性数组生成最小堆。

练习

6.3-1
以图 6.3 为模型，说明 BUILD-MAX-HEAP 对数组 A D h5; 3; 17; 10; 84; 19; 6; 22; 9i 的操作。

6.3-2证明对于 0 h blg nc，Û n/2 $_{h+1}$ ¹ 1 /2。

6.3-3为什么BUILD-MAX-HEAP第2行的循环索引i从bn/2c减少到1，而不是从1增加到bn/2c？

6.3-4
证明任意 n 元素堆中至多有 Û n/2$_{h+1}$ 个高度为 h 的节点。

6.4 堆排序算法

堆排序算法由过程 H EAPSORT 给出，它首先调用 BUILD-MAX-HEAP 过程在输入数组 AŒ1 W n 上构建最大堆。由于数组的最大元素存储在根 AŒ1 中，因此 HEAPSORT 可以通过将其与 AŒn 交换将其放置在正确的位置。如果该过程随后从堆4 中丢弃节点 n（它可以通过简单地减少 A: *heap-size*4 来实现），根的子节点仍为最大堆，但新的根元素可能违反最大堆性质。要恢复最大堆性质，该过程只需调用 MAX-HEAPIFY .A; 1/，这将在 AŒ1 W n 1 中留下一个最大堆。然后，HEAPSORT 过程对大小为 n 的最大堆重复此过程，直到大小为 2 的堆。（有关精确的循环不变量，请参见练习 6.4-2。）

```
堆排序 .A; n/

1 BUILD-MAX-HEAP .A; n/ 2 for i D n
downto 2 3 将 AŒ1 与 AŒi 交换
4 A: heap-size D A: heap-size 1 5 MAX-
HEAPIFY .A; 1/
```

图 6.4 显示了第 1 行构建初始最大堆之后 HEAPSORT 操作的示例。该图显示了第 235 行 for 循环第一次迭代之前以及每次迭代之后的最大堆。

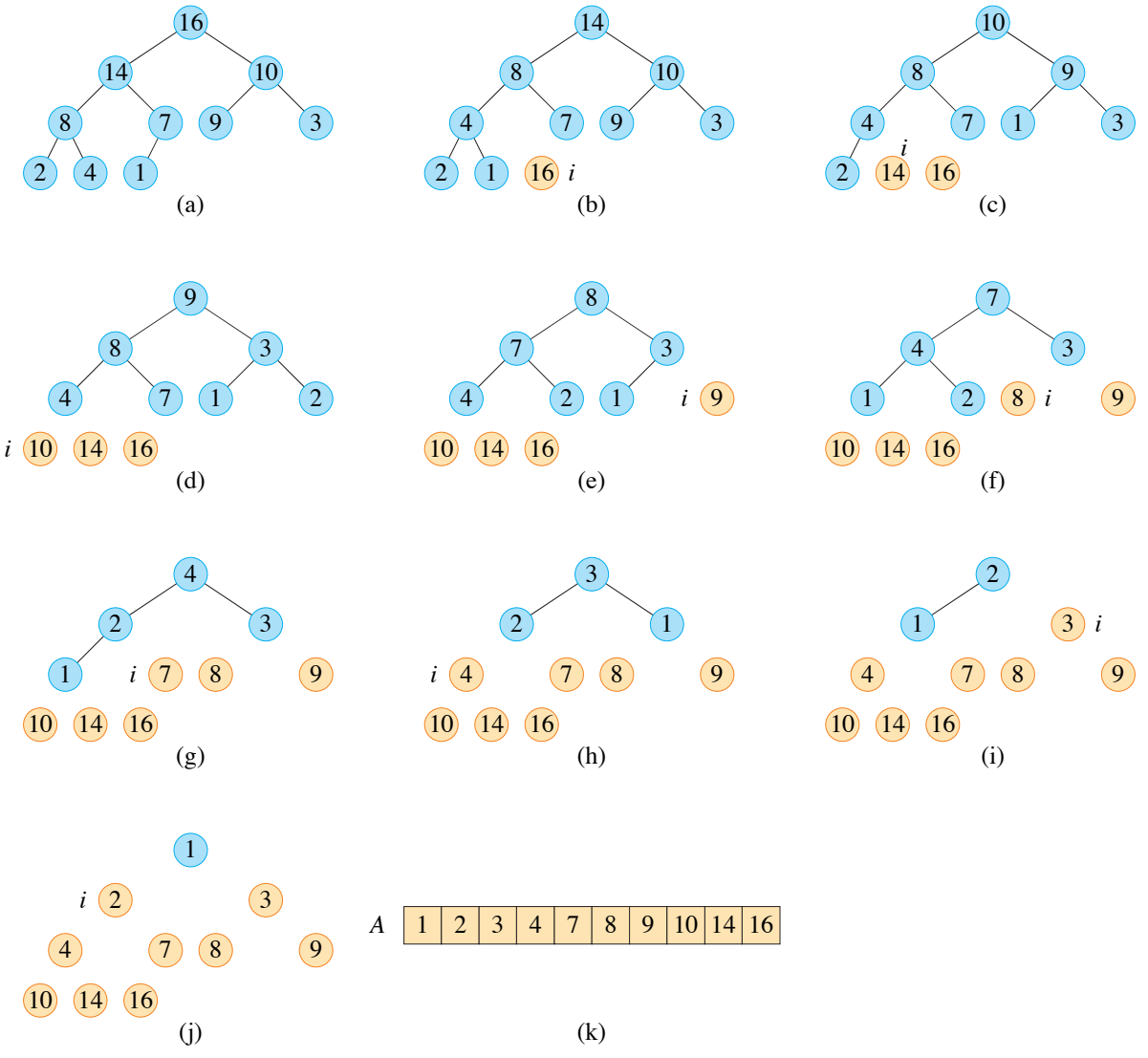


图 6.4 HEAPSORT 的操作。（a）第 1 行中 BUILD-MAX-HEAP 构建后的最大堆数据结构。（b）–（j）第 5 行中每次调用 MAX-HEAPIFY 后的最大堆，显示当时 i 的值。堆中只剩下蓝色节点。棕褐色节点按排序顺序包含数组中的最大值。（k）生成的排序数组 A。

HEAPSORT 过程花费 O.n lg n/ 时间，因为调用 BUILD-MAX-HEAP 花费 O.n/ 时间，并且对 MAX-HEAPIFY 的 n 1 次调用中的每一个都花费 O.lg n/ 时间。

练习

6.4-1
以图 6.4 为模型，说明 HEAPSORT 对数组 A D h5; 13; 2; 25; 7; 17; 20; 8; 4i 的操作。

6.4-2
使用以下循环不变量论证 HEAPSORT 的正确性：

> 在第 235 行的 for 循环每次迭代开始时，子数组 AŒ1Wi 是一个最大堆，包含 AŒ1Wn 中最小的 i 个元素，而子数组 AŒiC1Wn 包含 AŒ1Wn 中已排序的 ni 个最大元素。

6.4-3
对于长度为 n 且已经按升序排序的数组 A，HEAPSORT 的运行时间是多少？如果数组已经按降序排序，那会怎样？

6.4-4
证明HEAPSORT的最坏情况运行时间为 .n lg n/。

?***6.4-5***
证明当 A 的所有元素都不同时，HEAPSORT 的最佳运行时间为 .n lg n/。

6.5 优先级队列

在第 8 章中，我们将看到任何基于比较的排序算法都需要 .n lg n/ 次比较，因此需要 .n lg n/ 次时间。因此，在基于比较的排序算法中，堆排序是渐近最优的。然而，第 7 章中介绍的快速排序的良好实现通常在实践中胜过它。尽管如此，堆数据结构本身有很多用途。在本节中，我们介绍堆最流行的应用之一：作为高效的优先级队列。与堆一样，优先级队列也有两种形式：最大优先级队列和最小优先级队列。在这里我们将重点介绍如何实现最大优先级队列，而最大优先级队列又基于最大堆。练习 6.5-3 要求您编写最小优先级队列的程序。

priority queue 是一种数据结构，用于维护元素集 S，每个元素都有一个关联值，称为 ***key***。***max-priority queue*** 支持以下操作：

I NSERT.S; x; k/ 将键为 k 的元素 x 插入到集合 S 中，相当于操作 S D S [fx g.

MAXIMUM.S / 返回 S 中具有最大键的元素。

EXTRACT-MAX.S / 删除并返回 S 中具有最大键的元素。

I NCREASE-KEY .S; x; k/ 将元素 x 的键的值增加为新值 k，假定该值至少与 x 的当前键值一样大。

在其其他应用中，您可以使用最大优先级队列在多个用户共享的计算机上调度作业。最大优先级队列跟踪要执行的作业及其相对优先级。当作业完成或中断时，调度程序通过调用 EXTRACT-MAX 从待处理作业中选择最高优先级的作业。调度程序可以随时通过调用 INSERT 将新作业添加到队列中。

或者，***min-priority queue*** 支持操作 I NSERT、MINIMUM、EXTRACT-MI N 和 DECREASE-KEY。最小优先级队列可用于事件驱动的模拟器。队列中的项目是需要模拟的事件，每个事件都有一个关联的发生时间作为其键。必须按照事件发生时间的顺序模拟事件，因为一个事件的模拟可能会导致将来模拟其他事件。模拟程序在每个步骤中调用 EXTRACT-MIN 来选择下一个要模拟的事件。当产生新事件时，模拟器通过调用 I NSERT 将它们插入到最小优先级队列中。我们将在第 21 章和第 22 章中看到最小优先级队列的其他用途，重点介绍 DECREASE-KEY 操作。

当您使用堆在给定应用程序中实现优先级队列时，优先级队列的元素与应用程序中的对象相对应。每个对象都包含一个键。如果优先级队列由堆实现，则需要确定哪个应用程序对象对应于给定的堆元素，反之亦然。由于堆元素存储在数组中，因此您需要一种将应用程序对象映射到数组索引的方法。

在应用程序对象和堆元素之间进行映射的一种方法是使用 ***handles***，它是存储在对象和堆元素中的附加信息，可提供足够的信息来执行映射。句柄通常被实现为对周围代码不透明，从而在应用程序和优先级队列之间保持抽象屏障。例如，应用程序对象中的句柄可能包含堆数组中的相应索引。但由于只有优先级队列的代码访问此索引，因此索引对应用程序代码完全隐藏。由于堆元素在

在堆操作期间更新数组时，优先级队列的实际实现在重新定位堆元素时还必须更新相应句柄中的数组索引。相反，堆中的每个元素可能包含指向相应应用程序对象的指针，但堆元素只知道此指针是一个不透明的句柄，应用程序将此句柄映射到应用程序对象。通常，维护句柄的最坏情况开销是每次访问 O.1。

作为将句柄合并到应用程序对象中的替代方法，您可以在优先级队列中存储从应用程序对象到堆中的数组索引的映射。这样做的优点是映射完全包含在优先级队列中，因此应用程序对象无需进一步修饰。缺点在于建立和维护映射的额外成本。映射的一个选项是哈希表（参见第 11 章）。[1] 哈希表将对象映射到数组索引的额外预期时间仅为 O.1/，尽管最坏情况的时间可能高达 ，.n/。

让我们看看如何使用最大堆来实现最大优先级队列的操作。在前面的部分中，我们将数组元素视为要排序的键，隐式假设任何卫星数据都会随相应键一起移动。当堆实现优先级队列时，我们将每个数组元素视为指向优先级队列中对象的指针，以便排序时该对象类似于卫星数据。我们进一步假设每个这样的对象都有一个属性 *key*，它决定了对象在堆中的位置。对于由数组 A 实现的堆，我们引用 AŒi : *key*。

对页上的 MAX-HEAP-MAXIMUM 过程在 ，.1/ 时间内实现 MAXIMUM 操作，而 MAX-HEAP-EXTRACT-MAX 实现 EXTRACT-MAX 操作。MAX-HEAP-EXTRACT-MAX 与 HEAPSORT 过程的 for 循环体（335 行）类似。我们隐式假设 MAX-HEAPIFY 根据优先级队列对象的 *key* 属性进行比较。我们还假设当 MAX-HEAPIFY 交换数组中的元素时，它会交换指针，并且它会更新对象和数组索引之间的映射。MAX-HEAP-EXTRACT-MAX 的运行时间为 O.lg n/，因为它仅在 MAX-HEAPIFY 的 O.lg n/ 时间之上执行了常量的工作，再加上 MAX-HEAPIFY 中将优先级队列对象映射到数组索引所产生的任何开销。

第 176 页的过程 MAX-HEAP-IN NCREASE-KEY 实现了 IN NCREASE-KEY 操作。它首先验证新键 k 不会导致对象 x 中的键减少，如果没有问题，它将新键值赋予 x。然后，该过程在与对象 x 对应的数组中找到索引 i,

---

[1] In Python, dictionaries are implemented with hash tables.

最大堆最大.A/

1 if A: *heap-size* < 1 2 错误 “he
ap underüow” 3 返回 AŒ1

最大堆提取最大.A/

1 *max* D MAX-HEAP-MAXIMUM.A/ 2 AŒ1
D AŒA: *heap-size* 3 A: *heap-size* D A:
heap-size 1 4 MAX-HEAPIFY .A; 1/ 5 返回
max

这样 AŒi 为 x 。由于增加 AŒi 的键可能会违反最大堆性质，因此该过程接下来以类似于第 19 页 INSERTION-SORT 的插入循环（第 537 行）的方式，从此节点向根节点遍历一条简单路径，为新增加的键 找到合适的位置。当 MAX-HEAP-INCREASE-KEY 遍历这条路径时，它会反复将元素的键与其父元素的键进行比较，如果元素的键较大，则交换指针并继续，如果元素的键较小，则终止，因为现在最大堆性质成立。（有关精确的循环不变量，请参见练习 6.5-7。）与在优先级队列中使用时的 MAX-HEAPIFY 一样，在交换数组元素时，MAX-HEAP-INCREASE-KEY 会更新将对象映射到数组索引的信息。图 6.5 显示了 MAX-HEAP-I NCREASE-KEY 操作的示例。除了将优先级队列对象映射到数组索引的开销之外，在 n 元素堆上执行 MAX-H EAP-I NCREASE-KEY 的运行时间为 O.lg n/，因为从第 3 行更新的节点到根的路径长度为 O.lg n/。

下一页的过程 MAX-HEAP-INSERT 实现了 INSERT 操作。它将实现最大堆的数组 A、要插入到最大堆中的新对象 x 以及数组 A 的大小 n 作为输入。该过程首先验证数组是否有空间容纳新元素。然后，它通过向树中添加一个键为 1 的新叶子来扩展最大堆。然后，它调用 MAX-HEAP-IN NCREASE-K EY 将此新元素的键设置为其正确值并维护最大堆属性。在 n 元素堆上执行 MAX-HEAP-INSERT 的运行时间为 O.lg n/ 加上将优先级队列对象映射到索引的开销。

总之，堆可以在 O.lg n/ 时间内支持对大小为 n 的集合执行任何优先级队列操作，再加上将优先级队列对象映射到数组索引的开销。

MAX-HEAP-I NCREASE-KEY .A; x; k/

1 if k < x: *key* 2 error “新密钥小于当前密钥” 3 x: *key* D k 4 找到数组 A 中对象 x 出现的索引 i 5 while i > 1 and AŒPARENT.i / : *key* < AŒi : *key* 6 将 AŒi 与 AŒPARENT.i / 交换，更新将优先级队列对象映射到数组索引的信息 7 i D PARENT.i / MAX-HEAP-I NSERT .A; x; n/

1 if A: *heap-size* == n 2 error “heap overüo w” 3 A: *heap-size* D A: *heap-size* C 1 4 k D x: *key* 5 x: *key* D 1 6 AŒA: *heap-size* D x 7 将 x 映射到数组中的索引 *heap-size* 8 MAX-HEAP-I NCREASE-KEY .A; x; k/

练习

6.5-1
假设最大优先级队列中的对象只是键。说明 MAX-HEAP-EXTRACT-MAX 在堆 A D h15; 13; 9; 5; 12; 8; 7; 4; 0; 6; 2; 1i 上的操作。

6.5-2
假设最大优先级队列中的对象只是键。说明 MAX-HEAP-I NSERT .A; 10/ 在堆 A D h15; 13; 9; 5; 12; 8; 7; 4; 0; 6; 2; 1i 上的操作。

6.5-3
通过编写过程 MIN-HEAP-MINIMUM、MIN-HEAP-EXTRACT-MIN、MIN-HEAP-DECREASE-KEY 和 MIN-HEAP-I NSERT，编写伪代码来实现具有最小堆的最小优先级队列。

6.5-4
为最大堆中的 MAX-HEAP-DECREASE-KEY .A; x; k/ 过程编写伪代码。该过程的运行时间是多少？

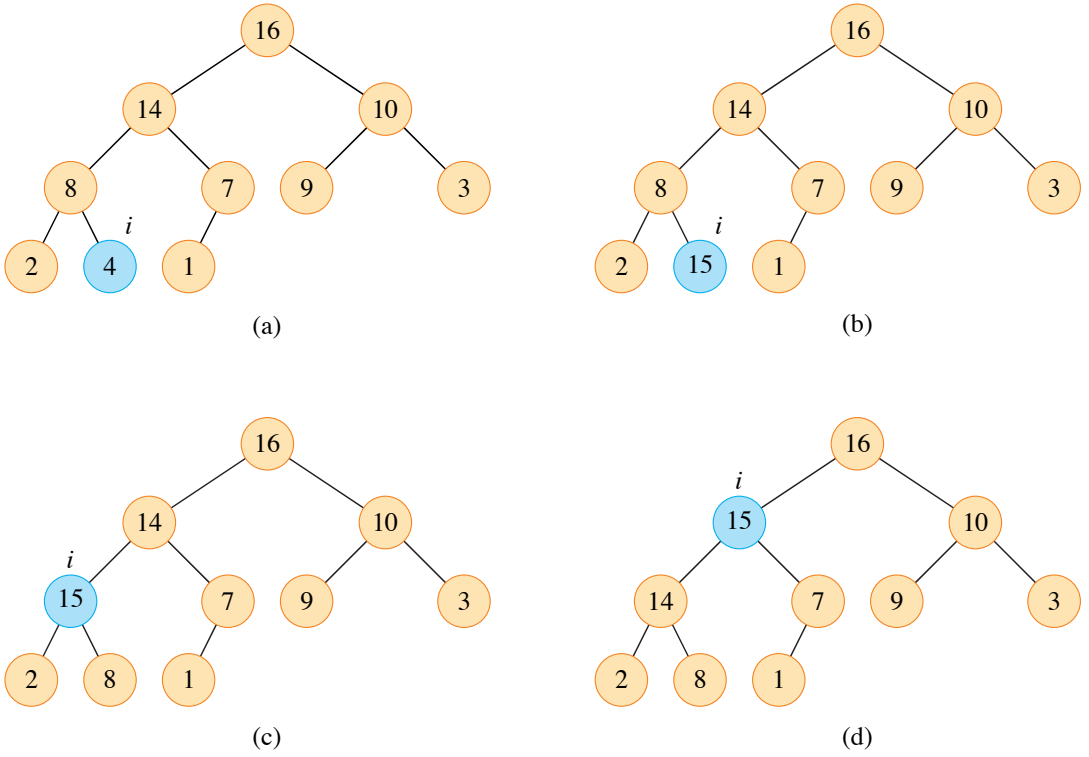


图 6.5 MAX-HEAP-I NCREASE-KEY 的操作。图中只显示了优先级队列中每个元素的键。每次迭代中索引为 i 的节点以蓝色显示。（a）图 6.4（a）中的最大堆，其中 i 索引即将增加键的节点。（b）此节点的键增加到 15。（c）经过 537 行的 while 循环一次迭代后，节点与其父节点交换了键，索引 i 上移到父节点。（d）经过一次 while 循环迭代后的最大堆。此时，AŒPARENT.i/ ¹AŒi 。最大堆属性现在成立，过程终止。

6.5-5

既然第 8 行会将对象的键设置为所需的值，为什么 MAX-HEAP-I NSERT 会在第 5 行费心将插入对象的键设置为1？

6.5-6

Uriah 教授建议用 MAX-HEAP-I NCREASE-KEY 中 537 行的 while 循环替换为对 MAX-HEAPIFY 的调用。解释教授想法中的问题。

6.5-7

使用以下循环不变量论证 MAX-HEAP-I NCREASE-KEY 的正确性：

在第 537 行的 while 循环每次迭代开始时：a. 如果节点 PARENT.i / 和 LEFT.i / 都存在，则 AŒPARENT.i / : *key* ¹ AŒLEFT.i / : *key*。b. 如果节点 PARENT.i / 和 RIGHT.i / 都存在，则 AŒPARENT.i / : *key* ¹ AŒRIGHT.i / : *key*。c. 子数组 AŒ1 W A: *heap-size* 满足最大堆属性，不过可能存在一处违规，即 AŒi : *key* 可能大于 AŒPARENT.i / : *key*。

你可以假设子数组 AŒ1 W A: *heap-size* 在调用 MAX-HEAP-INCREASE-KEY 时满足最大堆属性。

6.5-8
MAX-HEAP-IN NCREASE-KEY 第 6 行中的每个交换操作通常需要三次赋值，不包括从对象到数组索引的映射更新。说明如何使用 INSERTION-SORT 内循环的思想将三次赋值减少为一次赋值。

6.5-9
说明如何使用优先级队列实现先入先出队列。说明如何使用优先级队列实现堆栈。（队列和堆栈的定义见第 10.1.3 节。）

6.5-10
操作 MAX-HEAP-DELETE .A; x/ 从最大堆 A 中删除对象 x。给出 n 元素最大堆的 MAX-HEAP-DELETE 实现，运行时间为 O.lg n/ 加上将优先级队列对象映射到数组索引的开销。

6.5-11
给出一个 O.n lg k/-time 算法将 k 个排序列表合并为一个排序列表，其中 n 是所有输入列表中元素的总数。（*Hint:* 使用最小堆进行 k 路合并。）

问题

6-1Building a heap using insertion
构建堆的一种方法是重复调用 MAX-HEAP-INSERT 将元素插入堆中。请考虑上页中的 BUILD-MAX-HEAP ′ 过程。它假设插入的对象只是堆元素。

```
BUILD-MAX-HEAP 0 .A; n/
1 A: heap-size D 1 2 对于 i D 2 到 n 3 M
AX-HEAP-I NSERT .A; AŒi    ; n/
```

a.程序 BUILD-MAX-HEAP 和 BUILD-MAX-HEAP′在相同的输入数组上运行时是否总是创建相同的堆？请证明它们确实如此，或者提供反例。

b.证明在最坏情况下，BUILD-MAX-HEAP′需要，.n lg n/ 时间来构建一个 n 元素堆。

6-2Analysis of d ***-ary heaps***

d ***-ary heap*** 类似于二叉堆，但（可能有一个例外）非叶节点有 d 个子节点，而不是两个子节点。在此问题的所有部分中，假设维护对象和堆元素之间的映射的时间是每次操作的 O.1/。

a.描述如何在数组中表示 d 元堆。

b.使用 ，表示法，用 n 和 d 表示 n 个元素的 d 元堆的高度。***c.*** 给出 d 元最大堆中 EXTRACT-MAX 的有效实现。用 d 和 n 分析其运行时间。***d.*** 给出 d 元最大堆中 I NCREASE-KEY 的有效实现。用 d 和 n 分析其运行时间。***e.*** 给出 d 元最大堆中 I NSERT 的有效实现。用 d 和 n 分析其运行时间。

6-3Young tableaus

m n ***Young tableau*** 是一个 m n 矩阵，其中每行的条目按从左到右的顺序排列，每列的条目按从上到下的顺序排列。Young 表的一些条目可能为 1，我们将其视为不存在的元素。因此，Young 表可用于保存 r mn 个 ûnite 数字。

a.绘制一个 4 4 Young 表，其中包含元素 f9；16；3；2；4；8；5；14；12g。

b.如果 Y ≥ 1; 1 D 1，则论证 m n Young 表 Y 为空。如果 Y ≥ m; n ，则论证 Y 为满的（包含 mn 个元素）。< 1。***c.*** 给出一个算法，在非空 m n Young 表上实现 EXTRACT-MIN，运行时间为 O.m C n/min。你的算法应该使用递归子程序，通过递归求解 .m 1/ n 或 m .n 1/min 子问题来解决 m n 问题。（*Hint:* 思考 MAX-HEAPIFY。）解释为什么你的 EXTRACT-MIN 实现运行时间为 O.m C n/min。***d.*** 说明如何在 O.m C n/min 时间内将新元素插入非满的 m n Young 表。 ***e.*** 不使用其他排序方法作为子程序，说明如何使用 n n Young 表在 O.n^3/ 时间内对 n^2 个数字进行排序。***f.*** 给出一个 O.m C n/-time 算法来确定给定数字是否存储在给定的 m n Young 表中。

章节注释

堆排序算法是由 Williams [456] 发明的，他还描述了如何使用堆来实现优先级队列。BUILD-MAX-HEAP 程序由 Floyd [145] 提出。Schaffer 和 Sedgewick [395] 表明，在最佳情况下，堆排序过程中元素在堆中移动的次数约为 .n/2 /lg n，平均移动次数约为 nlg n。

我们在第 15、21 和 22 章中使用最小堆实现最小优先级队列。其他更复杂的数据结构为某些最小优先级队列操作提供了更好的时间界限。Fredman 和 Tarjan [156] 设计了斐波那契堆，它支持在 O.1/ 摊销时间内执行 I NSERT 和 DECREASE-KEY（参见第 16 章）。也就是说，这些操作的平均最坏情况运行时间为 O.1/。随后，Brodal、Lagogiannis 和 Tarjan [73] 设计了严格的斐波那契堆，使这些时间界限成为实际运行时间。如果键是唯一的且从集合 f0; 1 ; : : : ; 中抽取n 1g 个非负整数，van Emde Boas 树[440, 441] 在 O.lg lg n/ 时间内支持 INSERT、DELETE、SEARCH、MINIMUM、MAXIMUM、PREDECESSOR 和 SUCCESSOR 操作。

如果数据是 b 位整数，并且计算机内存由可寻址的 b 位字组成，Fredman 和 Willard [157] 展示了如何在 O.1/ 时间内实现 MINIMUM，并在 O. p lg n/ 时间内实现 INSERT 和 EXTRACT-MIN。Thorup [436] 有

通过使用随机散列，将 O. p lg n/ 绑定改进为 O.lg lg n/ 时间，只需要线性空间。

优先级队列的一个重要特例是 EXTRACT-MIN 操作的序列为 ***monotone***，即连续的 EXTRACT-MIN 操作返回的值随时间单调递增。这种情况出现在几个重要的应用中，例如我们在第 22 章中讨论的 Dijkstra 的单源最短路径算法，以及离散事件模拟。对于 Dijkstra 算法，高效实现 DECREASE-KEY 操作尤为重要。对于单调情况，如果数据是 1; 2; : : : 范围内的整数； C、Ahuja、Mehlhorn、Orlin 和 Tarjan [8] 描述了如何在 O.lg C/ 摊销时间内实现 EXTRACT-MIN 和 INSERT（第 16 章介绍摊销分析）以及在 O.1/ 时间内实现 DECREASE-KEY，其中使用的数据结构称为基数堆。使用斐波那契堆和基数堆可以将 O.lg C/ 界限改进为 O.p lg C/。Cherkassky、Goldberg 和 Silverstein [90] 通过将 Denardo 和 Fox [112] 的多层存储结构与前面提到的 Thorup 的堆相结合，进一步将界限改进为 O.lg $^{1/3+\epsilon}$ C/ 预期时间。Raman [375] 进一步改进这些结果，得到 O ã min $\{$lg $^{1/4+\epsilon}$C 的界限； lg $^{1/3+\epsilon}$n ä，对于任何固定的 $\epsilon > 0$。

人们提出了许多其他的堆变体。Brodal [72] 概述了其中一些发展。

7 Quicksort

对于包含 n 个数字的输入数组，快速排序算法的最坏情况运行时间为 '.n ^{2} / 。尽管最坏情况运行时间很慢，但快速排序通常是排序的最佳实际选择，因为它的平均效率非常高：当所有数字都不同且 '.n lg n/ 符号中隐藏的常数因子很小时，它的预期运行时间为 '.n lg n/。与归并排序不同，它还具有就地排序的优势（参见第 158 页），即使在虚拟内存环境中也能很好地工作。

我们对快速排序的研究分为四个部分。第 7.1 节描述了该算法以及快速排序用于分区的一个重要子程序。由于快速排序的行为很复杂，我们将在第 7.2 节中从直观的角度讨论其性能，并在本章末尾对其进行精确分析。第 7.3 节介绍了一种随机版本的快速排序。当所有元素都不同时，[1] 此随机算法具有良好的预期运行时间，并且没有特定输入会导致其最坏情况行为。（有关元素可能相等的情况，请参见问题 7-2。）第 7.4 节分析了随机算法，表明它在最坏情况下运行时间为 ，.n ^{2} / 时间，假设元素不同，预期运行时间为 O .n lg n/ 时间。

---

[1] You can enforce the assumption that the values in an array A are distinct at the cost of $\Theta(n)$ additional space and only constant overhead in running time by converting each input value $A[i]$ to an ordered pair $(A[i], i)$ with $(A[i], i) < (A[j], j)$ if $A[i] < A[j]$ or if $A[i] = A[j]$ and $i < j$. There are also more practical variants of quicksort that work well when elements are not distinct.

7.1 快速排序的描述

与归并排序类似，快速排序也应用了 2.3.1 节中介绍的分治法。下面是对子数组 AŒp W r 进行排序的三步分治过程：

通过将数组 AŒp W r 分割（重新排列）为两个（可能为空）子数组 AŒp W q 1 （***low side***）和 AŒq C 1 W r （***high side***）来进行划分，使得分割低端的每个元素都小于或等于 ***pivot*** AŒq ，而后者又小于或等于高端的每个元素。计算主元的索引 q 作为此分割过程的一部分。

通过递归调用快速排序对每个子数组 AŒp W q 1 和 AŒq C 1 W r 进行排序来征服。

无需执行任何操作即可合并：由于两个子数组已经排序，因此无需执行任何操作即可合并它们。AŒp W q 1 中的所有元素都已排序且小于或等于 AŒq ，AŒq C 1 W r 中的所有元素都已排序且大于或等于主元 AŒq 。整个子数组 AŒp W r 必然是已排序的！

QUICKSORT 过程实现快速排序。要对整个 n 元素数组 AŒ1 W n 进行排序，初始调用为 QUICKSORT.A; 1; n/。

```
QUICKSORT.A; p; r /
1 if p < r 2 // 围绕主元对子数组进行分区，最终得到 AŒq 。 3 q
D PARTITION.A; p; r / 4 QUICKSORT.A; p; q 1/ // 以递归方式对低
端进行排序 5 QUICKSORT.A; q C 1; r / // 以递归方式对高端进行排
序
```

对数组进行分区

该算法的关键是下一页的 PARTITION 过程，该过程重新排列子数组 AŒp W r ，并返回分区两侧分界点的索引。

图 7.1 显示了 PARTITION 如何对 8 元素数组进行操作。PARTITION 始终选择元素 x D AŒr 作为基准。在程序运行时，每个元素都会落入四个区域中的其中一个，其中一些区域可能为空。在第 336 行的 for 循环每次迭代开始时，这些区域都满足某些属性，如图 7.2 所示。我们将这些属性表示为循环不变量：

```
分区.A；p；r /

1 x D AŒr    // 主元 2 i D p  1 // 进入低侧的最高索引 3 对于 j D p 到 r  1
// 处理除主元之外的每个元素 4 如果 AŒj        x // 此元素是否属于低侧
？ 5 i D i C 1 // 低侧新位置的索引 6 将 AŒi    与 AŒj    // 交换，将此
元素放在那里 7 将 AŒi C 1    与 AŒr    // 交换，主元刚好位于低侧的右
侧 8 返回 i C 1 // 主元的新索引
```

在第 336 行循环的每次迭代开始时，对于任何数组索引 k，以下条件成立：

1. 如果 p = k = i，则 AŒk = x （图 7.2 中的棕褐色区域）；2. 如果 i C 1 = k = j 1，则 AŒk > x （蓝色区域）；3. 如果 k D r，则 AŒk D x （黄色区域）。

我们需要证明这个循环不变量在第一次迭代之前是正确的，循环的每次迭代都保持不变变量，循环终止，并且循环终止时正确性由不变量决定。

初始化：在循环第一次迭代之前，我们有 i D p 1 和 j D p。由于 p 和 i 之间没有值，i C 1 和 j 1 之间也没有值，因此循环不变量的前两个条件很容易满足。第 1 行中的赋值满足第三个条件。

维护：如图 7.3 所示，我们根据第 4 行测试的结果考虑两种情况。图 7.3(a) 显示了当 AŒj = > x 时发生的情况：循环中的唯一操作是增加 j 。增加 j 之后，第二个条件对 AŒj 1 成立，而其他所有项保持不变。图 7.3(b) 显示了当 AŒj = x 时发生的情况：循环增加 i ，交换 AŒi = x ，然后增加 j 。由于交换，我们现在得到 AŒi = x ，并且条件 1 得到满足。类似地，我们还得到 AŒj 1 = > x ，因为根据循环不变量，交换到 AŒj 1 中的项大于 x 。

终止：由于循环恰好进行了 r p 次迭代，因此它终止，此时 j D r 。此时，未检查的子数组 AŒj W r 1 为空，并且数组中的每个条目都属于不变量描述的另外三个集合之一。因此，数组中的值被划分为三个集合：小于或等于 x 的集合（低端）、大于 x 的集合（高端）和包含 x 的单例集合（主元）。

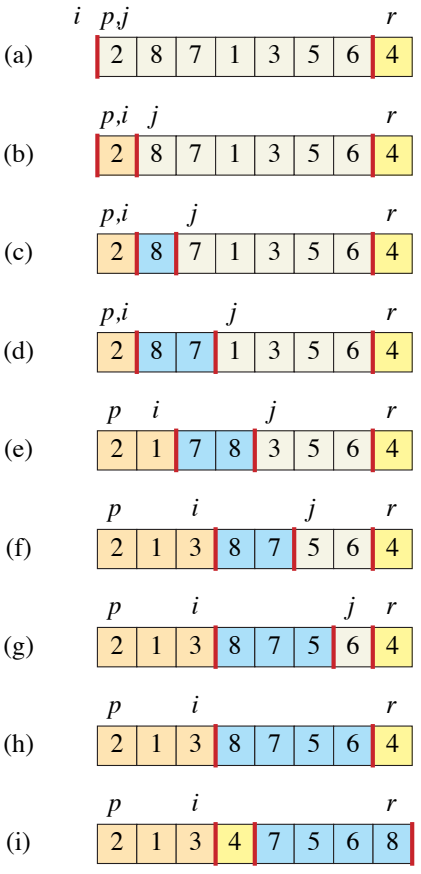


图 7.1 PARTITION 对示例数组的操作。数组条目 AŒr 成为枢轴元素 x。棕褐色数组元素全部属于分区的低端，其值最多为 x。蓝色元素属于高端，其值大于 x。白色元素尚未放入分区的两侧，黄色元素为枢轴 x。（a）初始数组和变量设置。没有任何元素被放入分区的两侧。（b）值 2 “与自身”交换并放入低端。（c）–（d）值 8 和 7 被放入高端。（e）值 1 和 8 被交换，低端增长。（f）值 3 和 7 被交换，低端增长。（g）–（h）分区的高端增长到包括 5 和 6，循环终止。（i）第 7 行交换枢轴元素，使其位于分区的两侧之间，第 8 行返回枢轴的新索引。

PARTITION 的最后两行通过将基准与最左边大于 x 的元素交换而结束，从而将基准移动到分区数组中的正确位置，然后返回基准的新索引。 PARTITION 的输出现在满足除法步骤的规范。事实上，它满足一个稍微强一点的条件：在 QUICKSORT 的第 3 行之后，AŒq 严格小于 AŒq C 1 W r 中的每个元素。

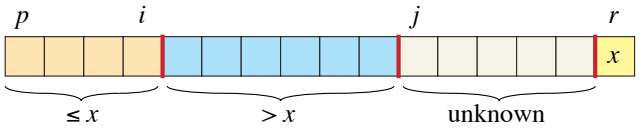


图 7.2 过程 PARTITION 在子数组 AŒp W r 上维护的四个区域。AŒp W i 中的 tan 值全部小于或等于 x，AŒi C 1 W j 1 中的蓝色值全部大于 x，AŒj W r 1 中的白色值与 x 的关系未知，AŒr D x。

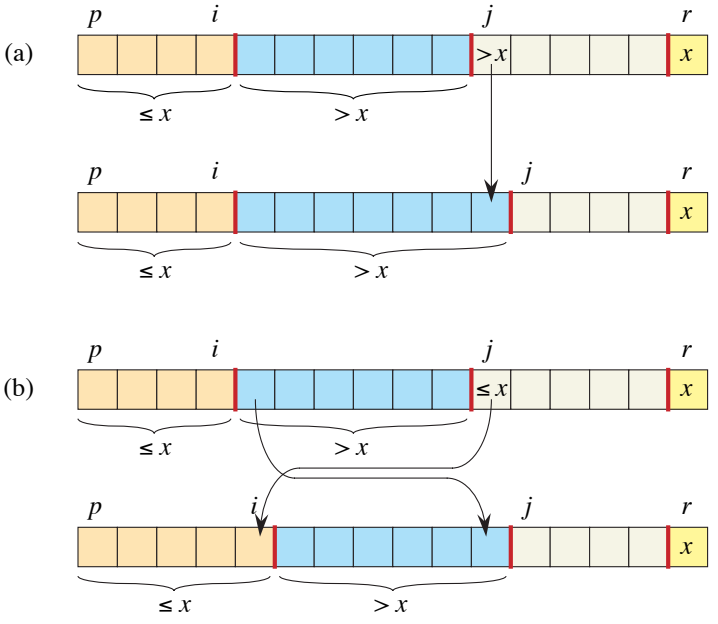


图 7.3 PARTITION 过程一次迭代的两种情况。 (a) 如果 AŒj ∠ > x，则唯一的操作是增加 j，这保持了循环不变式。 (b) 如果 AŒj ∠ x，则索引 i 增加，AŒi ∠ 和 AŒj ∠ 交换，然后增加 j。同样，循环不变式得到保持。

练习 7.1-3 要求你证明，对有 n D r p C 1 个元素的子数组 AŒp W r 进行 PARTITION 的运行时间为 ，.n/。

练习

7.1-1

以图7.1为模型，说明对数组A D h13; 19; 9; 5; 12; 8; 7; 4; 21; 2; 6; 11i 进行PARTITION的操作。

7.1-2
当子数组 AŒp W r 中的所有元素都具有相同的值时，PARTITION 返回 q 的什么值？修改 PARTITION，使得当子数组 AŒp W r 中的所有元素都具有相同的值时，q D b.p C r //2c。

7.1-3
简要论证一下 PARTITION 在大小为 n 的子数组上的运行时间为‘ .n/ ’。

7.1-4修改 QUICKSORT 以按单调递减顺序排序。

7.2 快速排序的性能

快速排序的运行时间取决于每个分区的平衡程度，而平衡程度又取决于使用哪些元素作为枢轴。如果分区的两侧大小大致相同4分区是平衡的4那么该算法的运行速度与合并排序一样快。但是，如果分区不平衡，它的运行速度可能与插入排序一样慢。为了让您在深入进行正式分析之前获得一些直觉，本节非正式地研究了在平衡分区和不平衡分区假设下快速排序的性能。

但是首先，让我们简要地看一下快速排序所需的最大内存量。尽管根据第 158 页的定义，快速排序是就地排序的，但它使用的内存量（除排序的数组外）不是恒定的。由于每个递归调用都需要运行时堆栈上除排序的数组之外的恒定空间量，因此快速排序所需的空间与递归的最大深度成比例。正如我们现在所看到的，在最坏的情况下，这可能和“ .n ”一样糟糕。

最坏情况划分

当划分产生一个包含 n 个 1 个元素的子问题和一个包含 0 个元素的子问题时，快速排序将出现最坏情况。（参见第 7.4.1 节。）我们假设这种不平衡划分出现在每次递归调用中。划分花费 ，.n/ 时间。由于对大小为 0 的数组的递归调用只是返回而不执行任何操作，因此 T .0/ D ，.1/，运行时间的递归式为

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n) .$$

通过对递归中每一级产生的成本求和，我们得到一个算术级数（第 1141 页的方程 (A.3)），其值为 ，.n 2 /。实际上，代换法可用于证明递归 T .n/ D T . n 1/ C ，.n/ 有解 T .n/ D ，.n 2 /。（见练习 7.2-1。）

因此，如果算法中每个递归级别的分区都最大程度地不平衡，则运行时间为 ，.n 2 /。因此，快速排序的最坏情况运行时间并不比插入排序好。此外，当输入数组已经完全排序时，运行时间为 ，.n 2 /，而插入排序的运行时间为 O.n/。

最佳情况划分

在最均匀的分割中，PARTITION 会产生两个子问题，每个子问题的大小不超过 n/2，因为一个子问题的大小为 b.n 1//2c n/2，另一个子问题的大小为 d.n 1//2e 1 n/2。在这种情况下，快速排序的运行速度要快得多。然后可以通过递归来描述运行时间的上限
$T(n) = 2T(n/2) + \Theta(n)$.

根据主定理的案例 2（第 102 页的定理 4.1），此递归具有解 T .n/ D ，.n lg n /。因此，如果在递归的每个级别上分区都同样平衡，则会产生一个渐近更快的算法。

平衡分区

正如第 7.4 节的分析所示，快速排序的平均运行时间更接近最佳情况，而不是最坏情况。通过了解分区平衡如何影响描述运行时间的递归，我们可以了解原因。

例如，假设分割算法总是产生 9 比 1 的比例分割，乍一看似乎很不平衡。然后我们得到递归

$$T(n) = T(9n/10) + T(n/10) + \Theta(n) ,$$

在快速排序的运行时间上。图 7.4 显示了此递归的递归树，其中为简单起见，驱动函数 “ .n/ ” 已被 n 替换，这不会影响递归的渐近解（如第 118 页练习 4.7-1 所述）。树的每一层都花费 n，直到递归在深度 $\log_{10} n$ D “ .lg n/ ” 的基线中触底，然后各层花费

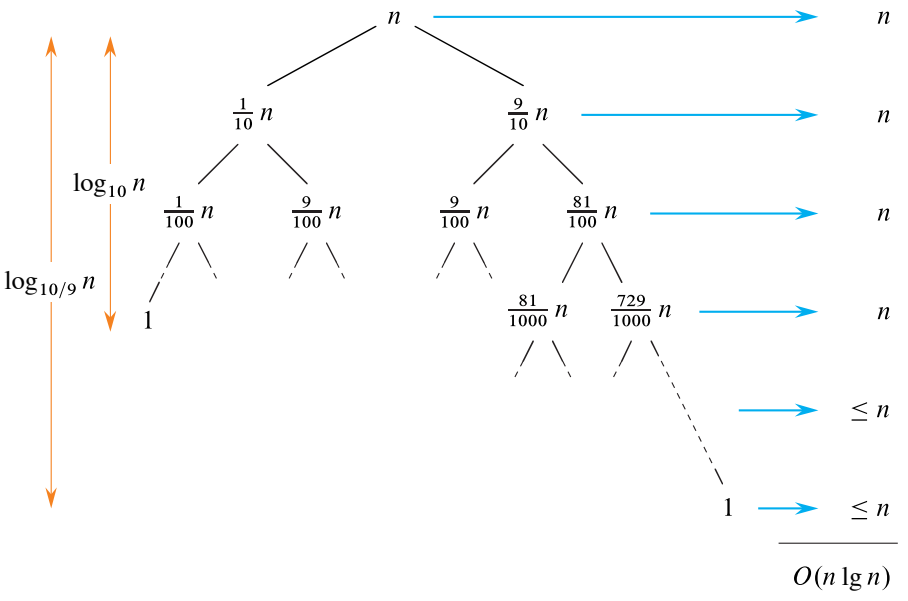


图 7.4 QUICKSORT 的递归树，其中 PARTITION 总是产生 9 比 1 的分割，运行时间为 O.n lg n/。节点显示子问题的大小，右侧显示每级成本。

最多为 n。递归终止于深度 log $_{10/9}$ n D ，.lg n/。因此，如果在每一级递归中都按 9 比 1 的比例分割，这在直观上似乎非常不平衡，则快速排序的运行时间为 O.n lg n/ 时间 4，与从中间分割时的时间渐近相同。事实上，即使是 99 比 1 的分割，运行时间也为 O.n lg n/。实际上，任何 *constant* 比例的分割都会产生深度为 ，.lg n/ 的递归树，其中每一级的成本为 O.n/。因此，只要分割具有恒定的比例，运行时间就是 O.n lg n/。分割比例仅影响隐藏在 O 符号中的常数。

普通情况下的直觉

为了清楚地了解快速排序的预期行为，我们必须假设其输入如何分布。由于快速排序仅使用输入元素之间的比较来确定排序顺序，因此其行为取决于作为输入给出的数组元素中值的相对顺序，而不是数组中的特定值。与第 5.2 节中招聘问题的概率分析一样，假设输入数字的所有排列都是等概率的，并且元素是不同的。

当快速排序在随机输入数组上运行时，分区不太可能像我们的非正式分析所假设的那样在每个级别上以相同的方式发生。

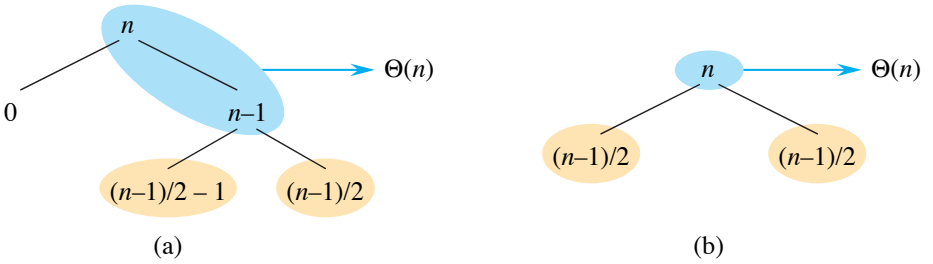


图 7.5 (a) 快速排序的两层递归树。在根节点处进行划分的成本为 n，并产生 “ 坏 ” 分割：两个大小分别为 0 和 n 1 的子数组。对大小为 n 1 的子数组进行划分的成本为 n 1，并产生 “ 好 ” 分割：大小为 .n 1//2 1 和 .n 1//2 的子数组。 (b) 平衡良好的单层递归树。在这两部分中，用蓝色阴影表示的子问题的划分成本均为 ，.n/。然而，(a) 中剩余待解的子问题（用棕褐色阴影表示）并不比 (b) 中剩余待解的相应子问题大。

我们期望一些分割会相当平衡，而一些会相当不平衡。例如，练习 7.2-6 要求您证明 PARTITION 产生的分割在大约 80% 的情况下至少与 9 比 1 一样平衡，而大约 20% 的情况下产生的分割比 9 比 1 更不平衡。

在平均情况下，PARTITION 会产生 “好” 和 “坏” 分割。在平均情况下执行 PARTITION 的递归树中，好分割和坏分割随机分布在整个树中。为了直观起见，假设好分割和坏分割在树中交替出现，并且好分割是最佳情况分割，坏分割是最坏情况分割。图 7.5(a) 显示了递归树中两个连续级别的分割。在树的根部，分割成本为 n，生成的子数组的大小为 n 1 和 0：最坏情况。在下一级，大小为 n 1 的子数组将经过最佳情况分割，划分为大小为 .n 1//2 1 和 .n 1//2 的子数组。我们假设大小为 0 的子数组的基本成本为 1。

坏分割和好分割的组合会产生三个子数组，大小分别为 0、.n 1//2 1 和 .n 1//2，总分割成本为 ，.n/C ，.n 1/D ，.n/。这种情况最多比图 7.5（b）中的情况差一个常数倍，即单级分割产生两个大小为 .n 1//2 的子数组，成本为 ，.n/。但后一种情况是平衡的！直观地说，图 7.5（a）中坏分割的 ，.n 1/ 成本可以吸收到好分割的 ，.n/ 成本中，因此最终的分割是好的。因此，当级别在好分割和坏分割之间交替时，快速排序的运行时间与单独使用好分割的运行时间一样：仍然是 O.n lg n/，但 O 符号隐藏了一个稍大的常数。我们将在 7.4.2 节中严格分析随机版本的快速排序的预期运行时间。

练习

7.2-1
利用代换法证明，如第 7.2 节开头所述，递归 T .n/ D T .n 1/ C ，.n/ 有解 T .n/ D ，.n ²/。

7.2-2
当数组 A 的所有元素具有相同的值时，QUICKSORT 的运行时间是多少？

7.2-3
证明当数组 A 包含不同元素且按降序排序时，QUICKSORT 的运行时间为，.n²/。

7.2-4
银行通常按交易时间顺序记录账户中的交易，但许多人喜欢收到按支票号码顺序列出的银行对账单。人们通常按支票号码顺序开具支票，而商家通常会以合理的速度兑现支票。因此，将交易时间排序转换为支票号码排序的问题就是对几乎排序的输入进行排序的问题。请令人信服地解释为什么 INSERTION-SORT 过程可能在这个问题上胜过 QUICKSORT 过程。

7.2-5
假设快速排序每一层的分割都是按常数比例 α 到 β 其中 $\alpha + \beta = 1$ 和 $0 < \alpha \leq \beta < 1$。证明递归树中叶子的最小深度约为 $\log_{1/\alpha} n$，最大深度约为 $\log_{1/\beta} n$。（不要担心整数舍入。）

7.2-6
考虑一个具有不同元素的数组，其中元素的所有排列都具有相同的可能性。论证对于任何常数 $0 < \alpha \leq 1/2$，PARTITION 产生至少与 $1 - \alpha$ 到 α 一样平衡的分割的概率约为 $1 - 2\alpha$。

7.3 快速排序的随机版本

在探索快速排序的平均行为时，我们假设输入数字的所有排列都是等概率的。然而，这一假设并不总是成立，例如，在

练习 7.2-4。第 5.3 节表明，有时可以将合理的随机化添加到算法中，以获得对所有输入的良好预期性能。对于快速排序，随机化产生了一种快速实用的算法。许多软件库都提供随机版本的快速排序作为对大型数据集进行排序的首选算法。

在 5.3 节中，RANDOMIZED-HIRE-ASSISTANT 程序明确地排列其输入，然后运行确定性的 HIRE-ASSISTANT 程序。我们也可以对快速排序做同样的事情，但不同的随机化技术可以产生更简单的分析。随机化版本不是总是使用 AŒr 作为主元，而是从子数组 AŒp W r 中随机选择主元，其中 AŒp W r 中的每个元素被选中的概率相等。然后在分割之前将该元素与 AŒr 交换。由于主元是随机选择的，我们预计输入数组的分割平均而言是相当平衡的。

PARTITION 和 QUICKSORT 的改动很小。新的分区程序 RANDOMIZED-PARTITION 只是在执行分区之前进行交换。新的快速排序程序 RANDOMIZED-QUICKSORT 调用 RANDOMIZED-PARTITION 而不是 PARTITION。我们将在下一节中分析此算法。

```
随机分区 .A; p; r /

1 i D RANDOM.p; r / 2 将 AŒ
r 与 AŒi 交换 3 返回 P
ARTITION.A; p; r /
随机快速排序 .A; p; r /

1 如果 p < r 2 q D 随机分区 .A; p; r / 3 随机快
速排序 .A; p; q  1/ 4 随机快速排序 .A; q C 1; r /
```

练习

7.3-1

为什么我们要分析随机算法的预期运行时间而不是最坏情况运行时间？

7.3-2
当 RANDOMIZED-QUICKSORT 运行时，在最坏情况下对随机数生成器 RANDOM 进行了多少次调用？在最好情况下又如何？请用 ，符号给出你的答案。

7.4 快速排序的分析

第 7.2 节对快速排序的最坏情况行为以及我们期望该算法快速运行的原因给出了一些直观的说明。本节将更严格地分析快速排序的行为。我们从最坏情况分析开始，该分析适用于 QUICKSORT 或 RANDOMIZED-QUICKSORT ，最后分析 RANDOMIZED-QUICKSORT 的预期运行时间。

7.4.1 最坏情况分析

我们在 7.2 节中看到，快速排序中每一级递归的最坏情况拆分都会产生 '.n ² / 的运行时间，直观地看，这是算法的最坏情况运行时间。现在我们来证明这一论断。

我们将使用替换法（参见第 4.3 节）来证明快速排序的运行时间为 O.n² / 。令 T.n/ 为 QUICKSORT 程序在大小为 n 的输入上最坏情况的时间。由于 PARTITION 程序产生两个总大小为 n 的子问题，我们得到递归

$$T(n) = \max\{T(q) + T(n-1-q) : 0 \le q \le n-1\} + \Theta(n)\,, \tag{7.1}$$

我们猜测，对于某个常数 c > 0，T .n/ cn ² 。将此猜测代入递归 (7.1) 可得

$$\begin{aligned} T(n) &\le \max\left\{cq^2 + c(n-1-q)^2 : 0 \le q \le n-1\right\} + \Theta(n) \\ &= c \cdot \max\left\{q^2 + (n-1-q)^2 : 0 \le q \le n-1\right\} + \Theta(n)\,. \end{aligned}$$

让我们把注意力集中在最大化上。对于 q D 0; 1; : : : ; n 1，我们有

$$\begin{aligned} q^2 + (n-1-q)^2 &= q^2 + (n-1)^2 - 2q(n-1) + q^2 \\ &= (n-1)^2 + 2q(q-(n-1)) \\ &\le (n-1)^2 \end{aligned}$$

因为 q n 1 意味着 2q.q .n 1// 0。因此最大化中的每个项都受 .n 1/² 的限制。

继续分析 T.n/，我们得到

$$\begin{aligned} T(n) &\le c(n-1)^2 + \Theta(n) \\ &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

通过选取足够大的常数 c，使得 c.2n 1/ 项大于 ，.n/ 项。因此 T .n/ D O.n ² / 。第 7.2 节展示了快速排序需要 ，.n ² / 时间的一个特殊情况：当分区极不平衡时。因此，快速排序的最坏情况运行时间为 ，.n ² /。

7.4.2 预计运行时间

我们已经看到了 RANDOMIZED-QUICKSORT 的预期运行时间为 O.n lg n/ 的直观原因：如果在每一级递归中，RANDOMIZED-PARTITION 引起的分割将任意常数部分元素放在分割的一侧，则递归树的深度为 ，.lg n/ ，且每一级执行 O.n/ 次工作。即使我们添加几个新级，并以最不平衡的方式在这些级之间进行分割，总时间仍然是 O.n lg n/。我们可以精确分析 RANDOMIZED-QUICKSORT 的预期运行时间，方法是首先理解分割过程的运行方式，然后利用这种理解推导出预期运行时间的 O.n lg n/ 界限。预期运行时间的上限与我们在第 7.2 节中看到的 ，.n lg n/ 最佳情况界限相结合，得出 ，.n lg n/ 的预期运行时间。我们始终假设被排序元素的值是不同的。

运行时间和比较

QUICKSORT 和 RANDOMIZED-QUICKSORT 过程仅在选择枢轴元素的方式上有所不同。它们在所有其他方面都是相同的。因此，我们可以通过考虑 QUICKSORT 和 PARTITION 过程来分析 RANDOMIZED-QUICKSORT，但假设枢轴元素是从传递给 RANDOMIZED-PARTITION 的子数组中随机选择的。让我们首先将 QUICKSORT 的渐近运行时间与元素比较的次数（都在 PARTITION 的第 4 行中）联系起来，并理解此分析也适用于 RANDOMIZED-QUICKSORT。请注意，我们计算的是 *array elements* 的比较次数，而不是索引的比较次数。

Lemma7.1
QUICKSORT 对 n 个元素数组的运行时间为 O.n C X /，其中 X 是执行的元素比较次数。

Proof QUICKSORT 的运行时间主要由 PARTITION 过程所花费的时间决定。每次调用 PARTITION 时，它都会选择一个枢轴元素，该元素永远不会包含在对 QUICKSORT 和 PARTITION 的任何未来递归调用中。因此，在整个快速排序算法的执行过程中，最多可以调用 n 次 PARTITION。每次 QUICKSORT 调用 PARTITION 时，它也会递归调用自身两次，因此最多可以调用 2n 次 QUICKSORT 过程本身。

一次 PARTITION 调用需要花费 O.1/ 时间，再加上与 336 行中 for 循环的迭代次数成比例的时间。此 for 循环的每次迭代都会执行第 4 行中的一次比较，将枢轴元素与数组 A 的另一个元素进行比较。因此，在所有执行过程中，for 循环所花费的总时间与 X 成比例。由于最多有 n 次调用 PARTITION，并且每次调用在 for 循环之外所花费的时间是 O.1/，因此在 for 循环之外的 PARTITION 所花费的总时间为 O.n/。因此，快速排序的总时间为 O.n C X /。 ■

因此，我们分析 RANDOMIZED-QUICKSORT 的目标是计算随机变量 X 的期望值 E ŒX ，该值表示在所有 PARTITION 调用中执行的总比较次数。为此，我们必须了解快速排序算法何时比较数组的两个元素，何时不比较。为便于分析，让我们根据数组 A 中元素在排序输出中的位置而不是在输入中的位置对其进行索引。也就是说，尽管 A 中的元素可能以任何顺序开始，但我们将通过 $z_1; z_2; : : : ; z_n$ 来引用它们，其中 $z_1 < z_2 < \rightarrow < z_n$，严格不等式，因为我们假设所有元素都是不同的。我们将集合 $fz_i; z_{i+1}; : : : ; z_j g$ 由 Z_{ij}。下一个引理描述两个元素的比较情况。

Lemma7.2

在对 n 个不同元素的数组 $z_1 < z_2 < + [] + [] < z_n$ 执行 RANDOMIZED-QUICKSORT 时，元素 z_i 与元素 z_j（其中 i < j ）进行比较，当且仅当它们中的一个在集合 Z_{ij} 中的任何其他元素之前被选为枢轴。此外，不会有两个元素被比较两次。

Proof 让我们看一下在算法执行过程中第一次选择元素 x 2 Z_{ij} 作为枢轴。有三种情况需要考虑。如果 x 既不是 z_i 也不是 z_j4，即 $z_i < x < z_j$4，那么 z_i 和 z_j 都不会在后续时间进行比较，因为它们位于 x 周围分区的不同侧。如果 x D z_i，则 PARTITION 会将 z_i 与 Z_{ij} 中的每个其他项进行比较。同样，如果 x D z_j，则 PARTITION 会将 z_j 与 Z_{ij} 中的每个其他项进行比较。因此，当且仅当从 Z_{ij} 中选出的第一个元素是 z_i 或 z_j 时，才会比较 z_i 和 z_j。在后两种情况下，会选择 z_i 和 z_j 之一

作为枢轴，由于枢轴已从未来的比较中移除，因此它永远不会再与其他元素进行比较。 ■

作为该引理的一个例子，考虑以任意顺序输入 1 到 10 的数字进行快速排序。假设第一个枢轴元素为 7。然后对 PARTITION 的第一次调用将数字分成两个集合：f1；2；3；4；5；6g 和 f8；9；10g。在此过程中，枢轴元素 7 与所有其他元素进行比较，但第一个集合中的任何数字（例如 2）都不会与第二个集合中的任何数字（例如 9）进行比较。比较值 7 和 9 是因为 7 是从 $Z_{7,9}$ 中选为枢轴的第一个项。相反，2 和 9 永远不会被比较，因为从 $Z_{2,9}$ 中选择的第一个枢轴元素是 7。

下一个引理给出了两个元素被比较的概率。

Lemma7.3

考虑对 n 个不同元素的数组 $z_1 < z_2 < + < z_n$ 执行 RANDOMIZED-QUICKSORT 过程。给定两个任意元素 z_i 和 z_j，其中 i < j，它们被比较的概率为 2/.j i C 1/。

Proof让我们看一下 RANDOMIZED-QUICKSORT 进行的递归调用树，并考虑作为每次调用的输入提供的元素集。最初，根集包含 Z_{ij} 的所有元素，因为根集包含 A 中的每个元素。属于 Z_{ij} 的元素在 RANDOMIZED-QUICKSORT 的每次递归调用中都保持在一起，直到 PARTITION 选择某个元素 x 2 Z_{ij} 作为主元。从那一刻起，主元 x 不会出现在后续输入集中。RANDOMIZED-SELECT 第一次从包含 Z_{ij} 所有元素的集合中选择主元 x 2 Z_{ij} 时，Z_{ij} 中的每个元素都有同等可能是 x，因为主元是随机均匀选择的。由于 $|Z_{ij}| = j - i + 1$，因此 Z_{ij} 中的任何给定元素是从 Z_{ij} 中选择的第一个主元的概率为 1/.j i C 1/。因此，根据引理 7.2，我们有

$$\begin{aligned}\Pr\{z_i \text{ is compared with } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\ &= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} \\ &\quad + \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\} \\ &= \frac{2}{j-i+1},\end{aligned}$$

第二行继承自第一行，因为这两个事件是互相排斥的。

■

现在我们可以完成随机快速排序的分析了。

Theorem7.4
对于 n 个不同元素的输入，RANDOMIZED-QUICKSORT 的预期运行时间为 O.n lg n/。

Proof分析中使用指示随机变量（见第 5.2 节）。设 n 个不同元素为 $z_1 < z_2 < + < z_n$，对于 1 = i < j = n，定义指示随机变量 X $_{ij}$ D I fz_i 与 z_jg 进行比较。根据引理 7.2，每对最多比较一次，因此我们可以将 X 表示如下：

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

通过对两边取期望值并利用期望的线性（第 1192 页的方程 (C.24)）和第 130 页的引理 5.1，我们得到

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared with } z_j\} && \text{(by Lemma 5.1)} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} && \text{(by Lemma 7.3) .}
\end{aligned}$$

我们可以使用变量变换 (k D j i) 和第 1142 页公式 (A.9) 中的调和级数的界限来计算这个和：

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) .
\end{aligned}$$

该界限和引理 7.1 使我们得出结论，RANDOMIZED-QUICKSORT 的预期运行时间为 O.n lg n/（假设元素值不同）。

■

练习

7.4-1
表明复发

$$T(n) = \max\{T(q) + T(n-q-1) : 0 \le q \le n-1\} + \Theta(n)$$

下限为 T .n/ D .n 2 /。

7.4-2
证明快速排序的最佳运行时间为 .n lg n/。

7.4-3
证明当 q D 0 或 q D n 1 时，表达式 q 2 C .n q 1/2 在 q D 0; 1; : : : ; n 1 上达到最大值。

7.4-4
证明 RANDOMIZED-QUICKSORT 的预期运行时间为 .n lg n/。

7.4-5
粗化递归（如我们在问题 2-1 中对合并排序所做的那样）是实践中改善快速排序运行时间的常用方法。我们修改了递归的基本情况，以便如果数组的元素少于 k 个，则按插入排序对子数组进行排序，而不是通过继续递归调用快速排序。论证此排序算法的随机版本在 O.nk C n lg.n/k// 的预期时间内运行。您应该如何选择 k，无论是在理论上还是在实践中？

7.4-6
考虑修改 PARTITION 过程，从子数组 AŒp W r 中随机选取三个元素，并根据它们的中位数（三个元素的中间值）进行分割。估计比从 α 到 .1 的 α 分割更差的概率，作为 α 的函数，范围为 $0 < \alpha < 1/2$。

问题

7-1Hoare partition correctness

本章给出的 PARTITION 版本不是原始的分区算法。这是原始的分区算法，由 C. A. R. Hoare 提出。

```
HOARE-分区 .A; p; r /

 1 x D AŒp    2 i D p  1 3 j D r C 1 4
while TRUE 5 repeat 6 j D j  1 7 till A
Œj          x 8 repeat 9 i D i C 1 10 till
AŒi       x 11 if i < j 12 将 AŒi     与
AŒj     交换 13 else return j
```

a.演示 HOARE-PARTITION 对数组 A D h13; 19; 9; 5; 12; 8; 7; 4; 11; 2; 6; 21i 的操作，显示第 4313 行 while 循环每次迭代之后数组的值以及索引 i 和 j 。

b.描述当 AŒp W r　中的所有元素都相等时，第 7.1 节中的 PARTITION 程序与 HOARE-PARTITION 有何不同。描述 HOARE-PARTITION 在快速排序中使用时相对于 PARTITION 的实际优势。

接下来的三个问题要求你仔细论证 HOARE-PARTITION 程序的正确性。假设子数组 AŒp W r　至少包含两个元素，请证明以下内容：

c.索引 i 和 j 使得过程永远不会访问子数组 AŒp W r　之外的 A 元素。

d.当 HOARE-PARTITION 终止时，它返回一个值 j，使得 p　 j < r。

e.当 HOARE-PARTITION 终止时，AŒp W j　的每个元素都小于或等于 AŒj C 1 W r　的每个元素。

7.1 节中的 PARTITION 过程将主元值（原先位于 AŒr 中）从其形成的两个分区中分离出来。另一方面，HOARE-PARTITION 过程始终将主元值（原先位于 AŒp 中）放入两个分区 AŒp W j 和 AŒj C 1 W r 中的一个。由于 p j < r，因此两个分区都不为空。

f.重写 QUICKSORT 过程以使用 HOARE-PARTITION。

7-2Quicksort with equal element values

7.4.2 节中对随机快速排序的预期运行时间的分析假设所有元素值都是不同的。本题考察当所有元素值相同时会发生什么情况。

a.假设所有元素值都相等。在这种情况下，随机快速排序的运行时间是多少？

b.PARTITION 过程返回一个索引 q，使得 AŒp W q 1 中的每个元素都小于或等于 AŒq ，并且 AŒq C 1 W r 中的每个元素都大于 AŒq 。修改 PARTITION 过程以生成过程 PARTITION′.A; p; r /，该过程对 AŒp W r 的元素进行置换并返回两个索引 q 和 t，其中 p q t r，使得

AŒq W t 中的所有元素都相等，• AŒp W q 1 中的每个元素都小于 AŒq ，并且• AŒt C 1 W r 中的每个元素都大于 AŒq 。

与 PARTITION 类似，您的 PARTITION′ 过程应该花费 ，.r p/ 时间。

c.修改 RANDOMIZED-PARTITION 过程以调用 PARTITION′，并将新过程命名为 RANDOMIZED-PARTITION′。然后修改 QUICKSORT 过程以生成一个过程 QUICKSORT′.A; p; r /，该过程调用 RANDOMIZED-PARTITION′ 并仅在元素彼此未知相等的分区上进行递归。

d.使用 QUICKSORT′，调整第 7.4.2 节中的分析，以避免假设所有元素都是不同的。

7-3Alternative quicksort analysis

随机快速排序的运行时间的另一种分析方法是关注随机快速排序每次递归调用的预期运行时间，而不是执行的比较次数。与第 7.4.2 节的分析一样，假设元素的值不同。

a.假设给定一个大小为 n 的数组，任何特定元素被选为主元的概率为 1/n。使用此概率定义指示随机变量 X $_i$ D I 第五个最小元素被选为主元。E ŒX $_i$ 是多少？

b.假设 T.n/ 是一个随机变量，表示快速排序在大小为 n 的数组上的运行时间。

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right] . \tag{7.2}$$

c.说明如何将方程（7.2）重写为

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=1}^{n-1}\mathrm{E}[T(q)] + \Theta(n) . \tag{7.3}$$

d.表明

$$\sum_{q=1}^{n-1} q \lg q \le \frac{n^2}{2}\lg n - \frac{n^2}{8} \tag{7.4}$$

对于 n 2.（*Hint:* 将求和拆分为两部分，一部分对于 q D 1; 2; : : : ; dn/2e 1 进行求和，另一部分对于 q D dn/2e ; : : : ; n 1 进行求和。）

e.利用方程 (7.4) 中的边界，证明方程 (7.3) 中的递归式有解 E ŒT .n/ D O. n lg n/。（*Hint:* 通过代入证明，当 n 足够大且某个正常数 a 时，E ŒT .n/ 是 lg n。）

7-4Stooge sort

Howard、Fine 和 Howard 教授提出了一种看似简单的排序算法，为了纪念他们，他们将其命名为 “ stooge 排序 ”，如以下页面所示。

a.论证调用 STOOGE-SORT.A；1；n/ 正确地对数组 AŒ1 W n 进行排序。

b. 给出 STOOGE-SORT 最坏情况运行时间的递归式以及最坏情况运行时间的严格渐近（ , -符号）界限。

c.将 STOOGE-SORT 的最坏情况运行时间与插入排序、归并排序、堆排序和快速排序进行比较。这些教授值得终身任职吗？

```
STOOGE-SORT.A; p; r /
1 如果 AŒp  > AŒr  2 将 AŒp  与 AŒr  交换 3 如
果 p C 1 < r 4 k D b.r  p C 1//3c // 向下舍入 5 STOOGE-SO
RT.A; p; r  k/ // 前三分之二 6 STOOGE-SORT.A; p C k; r /
// 后三分之二 7 STOOGE-SORT.A; p; r  k/ // 再次前三分之
二
```

7-5Stack depth for quicksort

第 7.1 节中的 QUICKSORT 过程对其自身进行了两次递归调用。在 QUICKSORT 调用 PARTITION 之后，它会递归地对分区的低端进行排序，然后递归地对分区的高端进行排序。QUICKSORT 中的第二次递归调用实际上并不是必要的，因为该过程可以使用迭代控制结构。这种称为 ***tail-recursion elimination*** 的转换技术由优秀的编译器自动提供。应用尾递归消除将 QUICKSORT 转换为 TRE-QUICKSORT 过程。

```
TRE-QUICKSORT .A; p; r /
1 while p < r 2 // 分区然后对低端进行排序
。 3 q D PARTITION.A; p; r / 4 TRE-QUICK
SORT.A; p; q 1/ 5 p D q C 1
```

a.论证 TRE-QUICKSORT .A; 1; n/ 正确对数组 AŒ1 W n 进行排序。

编译器通常使用包含每次递归调用相关信息（包括参数值）的 ***stack*** 来执行递归过程。最近一次调用的信息位于堆栈顶部，初始调用的信息位于堆栈底部。调用过程时，其信息是 ***pushed*** 进入堆栈，终止时，其信息是 ***popped***。由于我们假设数组参数由指针表示，因此堆栈上每个过程调用的信息需要 O.1/ 堆栈空间。***stack depth*** 是计算过程中任何时候使用的最大堆栈空间量。

b.描述一个场景，其中 TRE-QUICKSORT 的堆栈深度在 n 元素输入数组上为 “ .n/ ”。

c.修改TRE-QUICKSORT，使最坏情况堆栈深度为‘ .lg n/ ’。保持算法的预期运行时间为O.n lg n/。

7-6Median-of-3 partition

改进 RANDOMIZED-QUICKSORT 过程的一种方法是围绕一个更仔细选择的主元进行分割，而不是从子数组中随机挑选一个元素。一种常用的方法是 ***median-of-3*** 方法：选择从子数组中随机选择的 3 个元素的中位数（中间元素）作为主元。（参见练习 7.4-6。）对于这个问题，假设输入子数组 AŒp W r 中的 n 个元素不同，且 n ≈ 3。将 AŒp W r 的排序版本表示为 z_1 ；z_2 ；: : : ；z_n。使用 3 中位数法选择主元元素 x，得出 p_i D Pr fx D z_ig。

a.给出 p_i 作为 n 和 i 的函数的精确公式，其中 i D 2; 3; : : :; n 1。（观察到 p_1 D p_n D 0。）

b.与普通实现相比，3 中位数法将选择 x D $z_{\lfloor(n+1)/2\rfloor}$（AŒp W r 的中位数）作为主元的可能性提高了多少？假设 n ! 1，并给出这些概率的极限比。

c.假设我们定义一个 “好的” 分割意味着选择主元为 xDz_i，其中 n/3 = i = 2n/3。与普通实现相比，3 中位数方法将获得良好分割的可能性提高了多少？（*Hint:* 用积分近似总和。）

d.论证说，在快速排序的 .n lg n/ 运行时间中，3中位数方法仅影响常数因子。

7-7Fuzzy sorting of intervals

考虑一个排序问题，你不知道其中的确切数字。相反，对于每个数字，你知道它所属的实数线上的区间。也就是说，给定 n 个形式为 Œa_i; b_i 的闭区间，其中 a_i b_i。目标是 ***fuzzy-sort*** 这些区间：产生区间的排列 hi_1; i_2; : : :; i_ni，使得对于 j D 1; 2; : : :; n，存在 c_j 2 Œa_{i_j}; b_{i_j} 满足 c_1 c_2 c_n。

a.设计一个随机算法，对 n 个区间进行模糊排序。你的算法应该具有快速排序左端点（a_i 值）的算法的一般结构，但它应该利用重叠区间来提高运行时间。（随着区间重叠越来越多，概率

模糊排序间隔的过程变得越来越容易。您的算法应该利用这种重叠，只要它存在。）

b.假设你的算法通常运行时间为 ，.n lg n/ 预期时间，但当所有间隔重叠时（即，当存在一个值 x 使得 x 2 Œa $_i$; b $_i$ 对于所有 i ），则运行时间为 ，.n / 预期时间。你的算法不应该明确检查这种情况，而是其性能应该随着重叠量的增加而自然提高。

章节注释

快速排序由 Hoare [219] 发明，他的 PARTITION 版本出现在问题 7-1 中。Bentley [51，第 117 页] 将第 7.1 节中给出的 PARTITION 过程归功于 N. Lomuto。第 7.4 节中的分析基于 Motwani 和 Raghavan [336] 的分析。Sedgewick [401] 和 Bentley [51] 提供了有关实现细节及其重要性的良好参考。

McIlroy [323] 展示了如何设计一个“杀手对手”，该对手会生成一个数组，而几乎任何快速排序实现都需要，.n ^{2} / 的时间。

8 Sorting in Linear Time

我们已经看到一些可以在 O.n lg n/ 时间内对 n 个数字进行排序的算法。归并排序和堆排序在最坏情况下达到这个上限，而快速排序平均可以达到这个上限。此外，对于这些算法中的每一种，我们都可以生成一个由 n 个输入数字组成的序列，使算法在 .n lg n/ 时间内运行。

这些算法有一个有趣的特性：*the sorted order they determine is based only on comparisons between the input elements*。我们将此类排序算法称为 ***comparison sorts***。到目前为止介绍的所有排序算法都是比较排序。

在 8.1 节中，我们将证明，任何比较排序在最坏情况下都必须进行 .n lg n/ 次比较才能对 n 个元素进行排序。因此，归并排序和堆排序是渐近最优的，并且不存在比其快超过常数倍的比较排序。

8.2、8.3 和 8.4 节研究了三种排序算法4：计数排序、基数排序和桶排序4，它们在某些类型的输入上以线性时间运行。当然，这些算法使用除比较之外的其他操作来确定排序顺序。因此， .n lg n/ 下限不适用于它们。

8.1 排序的下限

比较排序仅使用元素之间的比较来获取有关输入序列 ha $_1$; a $_2$; : : : ; a $_n$ i 的顺序信息。也就是说，给定两个元素 a $_i$ 和 a $_j$，它执行以下测试之一：a $_i$ <、a $_j$、a $_i$、a $_j$、a $_i$、a $_j$、a $_i$、a $_j$ 或 a $_i$ >、a $_j$ 来确定它们的相对顺序。它不能检查元素的值或以任何其他方式获取有关它们的顺序信息。

由于我们要证明下限，因此在本节中，我们假设所有输入元素都是不同的，且不失一般性。毕竟，当元素可能不同或可能不不同时，不同元素的下限适用。因此，

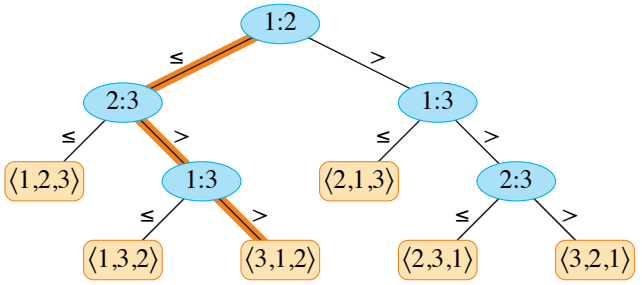


图 8.1 对三个元素进行插入排序的决策树。标有 i:j 的内部节点（显示为蓝色）表示 a $_i$ 和 a $_j$ 之间的比较。标有排列 h .1/; .2/; : : :; .n/ i 的叶子节点表示排序 a $_{\pi(1)}$ a $_{\pi(2)}$ a $_{\pi(n)}$。突出显示的路径表示对输入序列 ha $_1$ D 6; a $_2$ D 8; a $_3$ D 5i 进行排序时做出的决策。从标有 1:2 的根节点向左走表示 a $_1$ a $_2$。从标有 2:3 的节点向右走表示 a $_2$ > a $_3$。从标记为 1:3 的节点向右走表示 a $_1$ > a $_3$。因此，我们有顺序 a $_3$ a $_1$ a $_2$，如标记为 h3; 1; 2i 的叶子所示。由于三个输入元素有 3Š D 6 种可能的排列，因此决策树必须至少有 6 个叶子。

形式为 a $_i$ D a $_j$ 的比较是无用的，这意味着我们可以假设没有发生精确相等的比较。此外，比较 a $_i$ a $_j$、a $_i$ 和 a $_j$、a $_i$ > a $_j$ 和 a $_i$ < a $_j$ 都是等价的，因为它们会给出关于 a $_i$ 和 a $_j$ 相对顺序的相同信息。因此，我们假设所有比较都具有形式 a $_i$ a $_j$。

决策树模型

我们可以从决策树的角度抽象地看待比较排序。***decision tree*** 是一棵完整的二叉树（每个节点要么是叶子节点，要么有两个子节点），它表示由特定排序算法对给定大小的输入执行的元素之间的比较。控制、数据移动和算法的所有其他方面都被忽略。图 8.1 显示了对应于第 2.1 节中对三个元素的输入序列进行操作的插入排序算法的决策树。

决策树的每个内部节点都用 i :j 标注，其中 i 和 j 介于 1 # i; j # n 之间，其中 n 是输入序列中元素的数量。我们还用排列 h .1/; .2/; : : :; .n/ i 标注每个叶子节点。（有关排列的背景知识，请参见第 C.1 节。）内部节点和叶子节点中的索引始终引用排序算法开始时数组元素的原始位置。比较排序算法的执行对应于从决策树的根到叶子节点追踪一条简单路径。每个内部节点表示一个比较 a $_i$ # a $_j$。然后，左子树指示子

一旦我们知道 a $_i$ # a $_j$，我们就会进行后续比较，而当 a $_i$ > # a $_j$ 时，右子树会指示后续比较。到达叶子节点时，排序算法已经建立了顺序 a $_{\pi(1)}$ # a $_{\pi(2)}$ # # a $_{\pi(n)}$。因为任何正确的排序算法都必须能够生成其输入的每个排列，所以为使比较排序正确，n 个元素上的 nŠ 个排列中的每一个都必须至少作为决策树的一个叶子节点出现。此外，这些叶子节点中的每一个都必须可以从根节点通过与实际执行比较排序相对应的向下路径到达。（我们称这样的叶子节点为 “可达。”）因此，我们只考虑每个排列都作为可达叶子节点出现的决策树。

最坏情况的下限

从决策树的根到其任何可达叶子的最长简单路径的长度表示相应排序算法执行的最坏情况比较次数。因此，给定比较排序算法的最坏情况比较次数等于其决策树的高度。因此，所有决策树的高度下限（其中每个排列都作为可达叶子出现）是任何比较排序算法运行时间的下限。以下定理建立了这样的下限。

Theorem8.1
任何比较排序算法在最坏的情况下都需要 .n lg n/ 次比较。

Proof从前面的讨论中，我们足以确定决策树的高度，其中每个排列都显示为一个可达叶子。考虑一个高度为 h 的决策树，它有 l 个可达叶子，对应于 n 个元素的比较排序。由于输入的 nŠ 个排列中的每一个都显示为一个或多个叶子，因此我们有 nŠ = l。由于高度为 h 的二叉树最多有 2 个 ^{h} 个叶子，因此我们有

$n! \le l \le 2^h$,

通过取对数，可以得出

h ¹ lg.nŠ/ （因为 lg 函数是单调递增的） D .n lg n/ （根据第 67 页的公式 (3.28)）。 ■

Corollary8.2
堆排序和 合并排序是渐近最优的比较 排序。

Proof堆排序和归并排序的运行时间的 O.n lg n/ 上限与定理 8.1 中的 .n lg n/ 最坏情况的下限相匹配。 ■

练习

8.1-1对于比较排序，决策树中叶子的最小可能深度是多少？

8.1-2
不使用斯特林近似，获得 lg.nŠ/ 的渐近紧边界。相反，使用 A.2 节中的技术来计算和 P $_{k=1}^{n}$ lg k。

8.1-3
证明不存在对长度为 n 的 nŠ 输入的至少一半而言运行时间呈线性的比较排序。长度为 n 的输入中，1/n 的一小部分又如何？1/2 ^{n} 的一小部分又如何？

8.1-4
给定一个 n 元素输入序列，并且您事先知道该序列部分排序如下。初始位置为 i 的每个元素（i mod 4 D 0）要么已经位于其正确位置，要么距离其正确位置只有一个位置。例如，您知道排序后，初始位置为 12 的元素属于位置 11、12 或 13。您事先不知道位置为 i 且 i mod 4 ¤ 0 的其他元素。请证明在这种情况下，基于比较的排序的 .n lg n/ 下限仍然成立。

8.2 计数排序

Countingsort 假设 n 个输入元素中的每一个都是 0 到 k 范围内的整数，其中 k 是某个整数。它运行时间为 ，.n C k/ 时间，因此当 k D O.n/ 时，计数排序运行时间为 ，n/ 时间。

计数排序首先确定每个输入元素 x 中小于或等于 x 的元素数量。然后，它使用此信息将元素 x 直接放入输出数组中的位置。例如，如果有 17 个元素小于或等于 x，则 x 属于输出位置 17。我们必须稍微修改此方案以处理多个元素具有相同值的情况，因为我们不希望它们都位于同一位置。

对页上的 COUNTING-SORT 过程以数组 AŒ1 W n 、该数组的大小 n 以及 A 中非负整数值的极限 k 作为输入。它在数组 BŒ1 W n 中返回其排序后的输出，并使用数组 C Œ0 W k 作为临时工作存储。

```
计数排序.A; n; k/
 1 让 BŒ1 W n  和 C Œ0 W k  为新数组 2 对于 i D 0 到 k 3 C Œi
   D 0 4 对于 j D 1 到 n 5 C ŒAŒj      D C ŒAŒj      C 1 6 // C
Œi  现在包含等于 i 的元素数量。 7 对于 i D 1 到 k 8 C Œi  D C
Œi  C C Œi  1  9 // C Œi  现在包含小于或等于 i 的元素数
量。 10 // 将 A 复制到 B，从 A 的末尾开始。 11 for j D n downto 1
12 BŒC ŒAŒj      D AŒj  13 C ŒAŒj      D C ŒAŒj
      1 // 处理重复值 14 return B
```

图 8.2 说明了计数排序。在第 233 行的 for 循环将数组 C 初始化为全零之后，第 435 行的 for 循环遍历数组 A 以检查每个输入元素。每当它找到一个值为 i 的输入元素时，它就将 C Œi 加 1。因此，在第 5 行之后，C Œi 为每个整数 i D 0; 1; : : : ; k 保存等于 i 的输入元素数量。第 738 行通过保持数组 C 的运行总和来确定每个 i D 0; 1; : : : ; k 中有多少输入元素小于或等于 i。

最后，第 113 行和第 123 行的 for 循环再次遍历 A，但顺序相反，将每个元素 AŒj 放入输出数组 B 中的正确排序位置。如果所有 n 个元素都不相同，那么当第一次输入第 11 行时，对于每个 AŒj ，值 C ŒAŒj 就是 AŒj 在输出数组中的正确最终位置，因为有 C ŒAŒj 个元素小于或等于 AŒj 。由于元素可能不不同，因此每次将值 AŒj 放入 B 时，循环就会减少 C ŒAŒj 。减少 C ŒAŒj 会导致 A 中值等于 AŒj 的前一个元素（如果存在）移动到输出数组 B 中 AŒj 之前的位置。

计数排序需要多少时间？第 233 行的 for 循环需要 ，.k/ 时间，第 435 行的 for 循环需要 ，.n/ 时间，第 738 行的 for 循环需要 ，.k/ 时间，第 11313 行的 for 循环需要 ，.n/ 时间。因此，总时间为 ，.k C n/。在实践中，我们通常在有 k D O.n/ 时使用计数排序，这种情况下运行时间为 ，.n/。

计数排序可以超越 8.1 节中证明的下限 .n lg n/ ，因为它不是比较排序。事实上，代码中任何地方都不会发生输入元素之间的比较。相反，计数排序使用

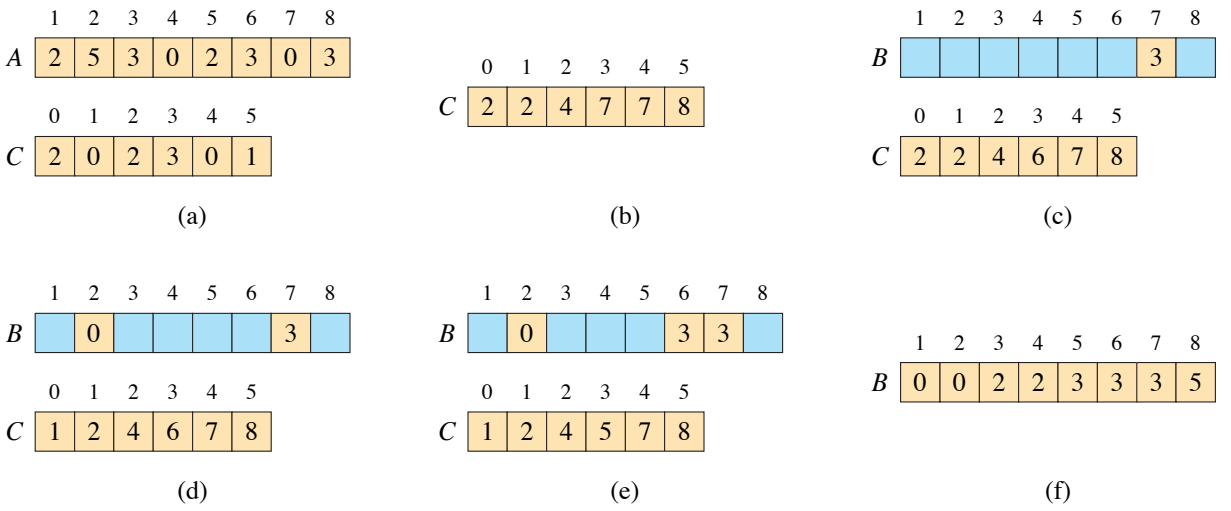


图 8.2 COUNTING-SORT 对输入数组 AŒ1 W 8 的运算，其中 A 的每个元素都是不大于 k D 5 的非负整数。（a）第 5 行之后的数组 A 和辅助数组 C。（b）第 8 行之后的数组 C。（c）–（e）第 113 行中循环一次、两次和三次迭代后的输出数组 B 和辅助数组 C。仅填充了数组 B 的 tan 元素。（f）最终排序后的输出数组 B。

元素索引到数组中。当我们脱离比较排序模型时， .n lg n/ 排序的下限不适用。

计数排序的一个重要特性是它是 ***stable***：具有相同值的元素在输出数组中的出现顺序与在输入数组中的出现顺序相同。也就是说，它按照以下规则打破两个元素之间的联系：在输入数组中首先出现的元素在输出数组中首先出现。通常，稳定性特性仅在卫星数据与正在排序的元素一起携带时才重要。计数排序的稳定性之所以重要还有另一个原因：计数排序通常用作基数排序的子程序。正如我们将在下一节中看到的那样，为了使基数排序正常工作，计数排序必须是稳定的。

练习

8.2-1

以图 8.2 为模型，说明对数组 A D h6; 0; 2; 0; 1; 3; 4; 6; 1; 3; 2i 进行 COUNTING-SORT 的操作。

8.2-2

证明COUNTING-SORT是稳定的。

8.2-3
假设我们将 COUNTING-SORT 的第 11 行的 for 循环头重写为

11 为 j D 1 至 n

证明该算法仍然正常工作，但不稳定。然后重写计数排序的伪代码，以便将具有相同值的元素按索引递增的顺序写入输出数组，并且算法是稳定的。

8.2-4
证明 COUNTING-SORT 的以下循环不变量：

> 在第 11313 行的 for 循环每次迭代开始时，A 中值为 i 且尚未复制到 B 中的最后一个元素属于 BŒC Œi 。

8.2-5
假设要排序的数组仅包含 0 到 k 范围内的整数，并且没有卫星数据可以使用这些键移动。修改计数排序以仅使用数组 A 和 C ，将排序结果放回到数组 A 中，而不是放入新数组 B 中。

8.2-6
描述一种算法，给定 0 到 k 范围内的 n 个整数，预处理其输入，然后在 O.1/ 时间内回答关于 n 个整数中有多少个落在范围 Œa W b 中的任何查询。您的算法应使用 ，.n C k/ 预处理时间。

8.2-7
如果输入值有小数部分，但小数部分的位数很少，计数排序也能有效地工作。假设你得到 n 个介于 0 到 k 之间的数字，每个数字的小数点右侧最多有 d 位十进制数（以 10 为基数）。修改计数排序，使其运行时间为 ，.n C 10 ^{d} k / 次。

8.3 基数排序

Radixsort 是卡片分类机使用的算法，现在只能在计算机博物馆找到。卡片有 80 列，机器可以在每列的 12 个位置之一打孔。分类机可以机械地“编程”检查一副牌中每张牌的给定列，并将卡片分配到一个

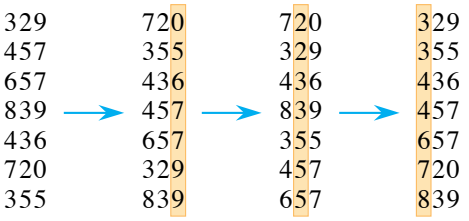


图 8.3 对七个 3 位数字进行基数排序的运算。最左边的列是输入。其余列显示按越来越重要的数字位置连续排序后的数字。棕褐色阴影表示按数字位置排序以根据前一个列表生成每个列表。

12 个箱子，具体取决于打孔的位置。然后操作员可以逐箱收集卡片，这样打孔第一个位置的卡片位于打孔第二个位置的卡片之上，依此类推。

对于十进制数字，每列仅使用 10 个位置。（其他两个位置保留用于编码非数字字符。）d 位数字占据 d 列的字段。由于卡片分类器一次只能查看一列，因此按 d 位数字对 n 张卡片进行分类的问题需要一种排序算法。

直观地讲，您可以按数字的 *most significant*（最左边）数字对数字进行排序，然后以递归方式对每个结果箱进行排序，然后按顺序组合卡片组。不幸的是，由于必须将 10 个箱中的 9 个中的卡片放在一边才能对每个箱进行排序，因此此过程会产生许多中间卡片堆，您必须跟踪这些卡片。（参见练习 8.3-6）。

基数排序通过首先按 *least significant* 位进行排序，解决了卡片分类问题4（与直觉相反4）。然后，该算法将卡片组合成一副牌，其中 0 号箱子中的卡片在 1 号箱子中的卡片之前，在 2 号箱子中的卡片之前，依此类推。然后，它再次按第二低有效数字对整副牌进行排序，并以类似方式重新组合整副牌。该过程持续进行，直到卡片按所有 d 位数字排序完毕。值得注意的是，此时卡片已按 d 位数字完全排序。因此，只需对整副牌进行 d 次排序即可进行排序。图 8.3 显示了基数排序如何对由七个 3 位数字组成的“牌组”进行操作。

为了使基数排序正确工作，数字排序必须稳定。卡片分类器执行的排序是稳定的，但操作员必须小心不要改变卡片从箱子里出来的顺序，即使箱子里的所有卡片在所选列中都有相同的数字。

在典型的计算机（一种顺序随机存取机器）中，我们有时使用基数排序对由多个字段键控的信息记录进行排序。例如，我们可能希望按三个键对日期进行排序：年、月和日。我们可以运行一个排序算法，该算法带有一个比较函数，给定两个日期，

比较年份，如果出现平局，则比较月份，如果再次出现平局，则比较天数。或者，我们可以对信息进行三次稳定排序：第一次按天（“最不重要”部分），第二次按月，最后一次按年。

基数排序的代码很简单。RADIX-SORT 程序假设数组 AŒ1 W n 中的每个元素都有 d 位数字，其中数字 1 是最低位数字，数字 d 是最高位数字。

```
RADIX-SORT.A; n; d /
1 for i D 1 to d 2 使用稳定排序对数组 AŒ1 W n
按数字 i 进行排序
```

尽管 RADIX-SORT 的伪代码没有指定使用哪种稳定排序，但 COUNTING-SORT 是常用的排序方式。如果使用 COUNTING-SORT 作为稳定排序，则可以修改 COUNTING-SORT 以将指向输出数组的指针作为参数，让 RADIX-SORT 预分配此数组，并在 RADIX-SORT 中 for 循环的连续迭代中在两个数组之间交替输入和输出，从而使 RADIX-SORT 更加高效。

Lemma8.3
给定 n 个 d 位数字，其中每个数字最多可以取 k 个可能的值，如果 RADIX-SORT 使用的稳定排序需要 ，.n C k/ 时间，那么它可以在 ，.d.n C k// 时间内正确地对这些数字进行排序。

Proof基数排序的正确性可以通过对排序列进行归纳得出（参见练习 8.3-3）。运行时间的分析取决于用作中间排序算法的稳定排序。当每个数字都在 0 到 k 1 的范围内（这样它可以取 k 个可能的值），并且 k 不是太大时，计数排序是显而易见的选择。然后，对 n 个 d 位数字的每次遍历需要 ，.n C k/ 时间。共有 d 次遍历，因此基数排序的总时间为 ，.d.n C k//。 ■

当 d 为常数且 k D O.n/ 时，我们可以使基数排序以线性时间运行。更一般地，我们在如何将每个键分解为数字方面具有一定的灵活性。

Lemma8.4
给定 n 个 b 位数和任意正整数 r # b，如果 RADIX-SORT 使用的稳定排序对于 0 到 k 范围内的输入需要 ，.n C k/ 时间，则 RADIX-SORT 能够在 ，..b/r /.n C 2^r // 时间内正确地对这些数字进行排序。

Proof 对于值 r = b，将每个键视为具有 d D db/r 个数字，每个数字为 r 位。每个数字都是 0 到 2^r 1 范围内的整数，因此我们可以使用计数排序，其中 k D 2^r 1。（例如，我们可以将 32 位字视为具有四个 8 位数字，因此 b D 32，r D 8，k D 2^r 1 D 255，d D b/r D 4。）计数排序的每次传递需要 ，.n C k/ D ，.n C 2^r/ 时间，共有 d 次传递，总运行时间为 ，.d.n C 2^r// D ，..b/r /.n C 2^r//。■

给定 n 和 b，r 和 b 的哪个值可使表达式 .b/r /.n C 2^r/ 最小化？随着 r 减小，因子 b/r 增大，但随着 r 增大，因子 2^r 也增大。答案取决于 b < blg nc 是否成立。如果 b < blg nc，则 r 和 b 意味着 .nC2^r/ D ，.n/。因此，选择 r D b 可得到 .b/b/.n C 2^b/ D ，.n/ 的运行时间，这是渐近最优的。如果 b 和 blg nc，则选择 r D blg nc 可得到常数因子内的最佳运行时间，如下所示。[1] 选择 r D blg nc 可得到运行时间为 ，.b n/ lg n/。当 r 增加到 blg nc 以上时，分子中的 2^r 项比分母中的 r 项增加得更快，因此将 r 增加到 blg nc 以上可得到运行时间为 .b n/ lg n/。如果 r 减少到 blg nc 以下，则 b/r 项会增加，而 n C 2^r 项仍为 ，.n/。

基数排序是否优于基于比较的排序算法，如快速排序？如果 b≤O.lg n/（通常情况如此），且 r≤lg n，则基数排序的运行时间为 ′ .n/，这似乎比快速排序的预期运行时间 ′ .nlg n/ 要好。然而，隐藏在 ′ 符号中的常数因子有所不同。尽管基数排序在 n 个键上进行的遍历次数可能比快速排序少，但基数排序的每次遍历所花的时间可能要长得多。选择哪种排序算法取决于实现、底层机器（例如，快速排序通常比基数排序更有效地使用硬件缓存）和输入数据的特性。此外，使用计数排序作为中间稳定排序的基数排序版本不是原地排序，而许多 ′ .nlg n/ 次比较排序都是原地排序。因此，当主存储器存储非常宝贵时，快速排序等就地算法可能是更好的选择。

练习

8.3-1

以图 8.3 为模型，说明 RADIX-SORT 对以下英文单词列表的操作：COW、DOG、SEA、RUG、ROW、MOB、BOX、TAB、BAR、EAR、TAR、DIG、BIG、TEA、NOW、FOX。

---

[1] The choice of $r = \lfloor \lg n \rfloor$ assumes that $n > 1$. If $n \leq 1$, there is nothing to sort.

8.3-2
以下哪种排序算法是稳定的：插入排序、归并排序、堆排序和快速排序？给出一种使任何比较排序稳定的简单方案。你的方案需要多少额外的时间和空间？

8.3-3
使用归纳法证明基数排序有效。你的证明在哪里需要假设中间排序是稳定的？

8.3-4
假设 COUNTING-SORT 用作 RADIX-SORT 中的稳定排序。如果 RADIX-SORT 调用 COUNTING-SORT d 次，那么由于每次调用 COUNTING-SORT 都会对数据进行两次传递（第 435 行和第 11313 行），因此总共对数据进行了 2d 次传递。描述如何将总传递次数减少到 d C 1。

8.3-5说明如何在 O.n/ 时间内对 0 到 n^3 1 范围内的 n 个整数进行排序。

8.3-6
在本节中的第一个卡片分类算法中，首先按最高有效数字进行分类，在最坏的情况下，需要多少次分类才能对 d 位十进制数进行分类？在最坏的情况下，操作员需要跟踪多少堆卡片？

8.4 桶排序

Bucketsort 假设输入服从均匀分布，平均运行时间为 O.n/。与计数排序一样，桶排序速度很快，因为它对输入做了一些假设。计数排序假设输入由小范围内的整数组成，而桶排序则假设输入由随机过程生成，该过程将元素均匀且独立地分布在区间 Œ0; 1/ 上。（有关均匀分布的定义，请参阅第 C.2 节。）

桶排序将区间 Œ0; 1/ 划分为 n 个大小相等的子区间，即 ***buckets***，然后将 n 个输入数字分配到桶中。由于输入在 Œ0; 1/ 上均匀且独立地分布，因此我们预计每个桶中不会有很多数字。要生成输出，我们只需对每个桶中的数字进行排序，然后按顺序遍历桶，列出每个桶中的元素。

下一页的 BUCKET-SORT 过程假设输入是一个数组 AŒ1 W n ，并且数组中的每个元素 AŒi 都满足 0 AŒi < 1。代码需要一个由链接列表（桶）组成的辅助数组 BŒ0 W n 1 ，并假设

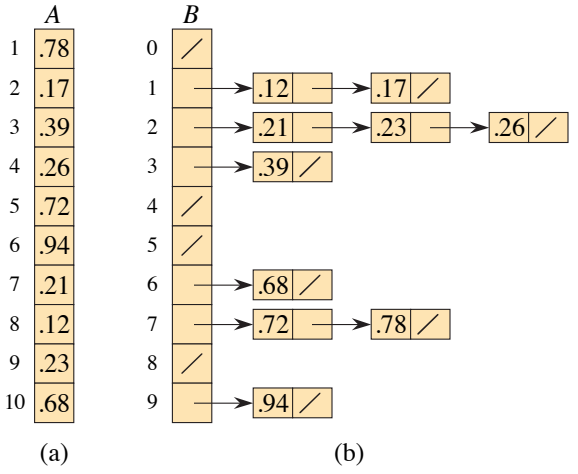


图 8.4 BUCKET-SORT 对 n D 10 的操作。（a）输入数组 AŒ1 W 10 。（b）算法第 7 行之后的排序列表（桶）数组 BŒ0 W 9 ，斜线表示每个桶的末尾。桶 i 保存半开区间 Œi/10; .i C 1//10/ 中的值。排序后的输出由按顺序连接的列表 BŒ0 ; BŒ1 ; : : : ; BŒ9 组成。

有一种机制可以维护这样的列表。（第 10.2 节介绍如何实现链表的基本操作。）图 8.4 显示了对包含 10 个数字的输入数组进行桶排序的操作。

桶排序.A; n/

1 令 BŒ0 W n 1 为一个新数组 2 对于 i D 0 到 n 1 3 令 BŒi 为空列表 4 对于 i D 1 到 n 5 将 AŒi 插入列表 BŒbn A Œi c 6 对于 i D 0 到 n 1 7 使用插入排序对列表 BŒi 进行排序 8 按顺序连接列表 BŒ0 ; BŒ1 ; : : : ; BŒn 1 9 返回连接后的列表

要验证此算法是否有效，请考虑两个元素 AŒi 和 AŒj 。不失一般性地假设 AŒi AŒj 。由于 bn AŒi c bn AŒj c，元素 AŒi 要么与 AŒj 进入同一个桶，要么进入索引较低的桶。如果 AŒi 和 AŒj 进入同一个桶，则第 637 行的 for 循环会将它们放入正确的顺序。如果 AŒi 和 AŒj 进入不同的桶，则第 8 行会将它们放入正确的顺序。因此，桶排序可以正常工作。

为了分析运行时间，请注意，除了第 7 行之外，所有行在最坏情况下总共需要 O.n/ 时间。我们需要分析第 7 行中 n 次插入排序调用所花费的总时间

° 为了分析插入排序的调用成本，设 n $_i$ 为随机变量，表示放入桶 BŒi 中的元素数量。由于插入排序以二次时间运行（参见第 2.2 节），因此桶排序的运行时间为

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \; . \tag{8.1}$$

现在，我们通过计算运行时间的期望值来分析桶排序的平均运行时间，其中我们取输入分布的期望值。取两边的期望值并使用期望的线性（第 1192 页的方程 (C.24)），我们得到

$$\begin{aligned} \mathrm{E}\left[T(n)\right] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \quad \text{(by linearity of expectation)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \quad \text{(by equation (C.25) on page 1193) 。} \end{aligned} \tag{8.2}$$

我们主张

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n \tag{8.3}$$

对于 i D 0; 1; : : : ; n 1. 每个桶 i 都有相同的 E Œn$_i^2$ 值，这并不奇怪，因为输入数组 A 中的每个值都有同等的可能性落入任何一个桶中。

为了证明公式 (8.3)，将每个随机变量 n $_i$ 视为 n 次伯努利试验中的成功次数（见 C.4 节）。当一个元素以 p D 1/n 的概率和 q D 1 1/n 的概率进入桶 BŒi 时，试验成功。二项分布计数 n 次试验中的成功次数 n $_i$。根据第 1199 31200 页上的公式 (C.41) 和 (C.44)，我们有 E Œn $_i$ D np D n.1/n/ D 1 和 V ar Œn $_i$ D npq D 1 1/n。第 1194 页上的公式 (C.31) 给出

$$\begin{aligned} \mathrm{E}\left[n_i^2\right] &= \mathrm{Var}\left[n_i\right] + \mathrm{E}^2\left[n_i\right] \\ &= (1 - 1/n) + 1^2 \\ &= 2 - 1/n \; , \end{aligned}$$

证明了公式（8.3）。利用公式（8.2）中的期望值，我们得到桶排序的平均运行时间为 ，.n/ Cn O.2 1/n/ D ，.n/。

即使输入不是来自均匀分布，桶排序仍可以在线性时间内运行。只要输入具有桶大小平方和与元素总数成线性关系的属性，公式 (8.1) 告诉我们桶排序可以在线性时间内运行。

练习

8.4-1

以图 8.4 为模型，说明 BUCKET-SORT 对数组 A D h:79; :13; :16; :64; :39; :20; :89; :53; :71; :42 i 的操作。

8.4-2

解释为什么桶排序的最坏情况运行时间是 ，.n ²/。对算法进行什么简单的改变可以保持其线性平均运行时间，并使其最坏情况运行时间为 O.n lg n/？

8.4-3

假设 X 是一个随机变量，等于一枚公平硬币中两枚正面朝上的次数。E ŒX ² 是多少？E ² ŒX 是多少？

8.4-4

大小为 n > 10 的数组 A 按以下方式填充。对于每个元素 AŒi ，均匀且独立于 Œ0; 1/ 地选择两个随机变量 x $_i$ 和 y $_i$。然后设置

$$A[i] = \frac{\lfloor 10x_i \rfloor}{10} + \frac{y_i}{n} .$$

修改桶排序，以便它在 O.n/ 预期时间内对数组 A 进行排序。

?***8.4-5***

给定单位圆盘中的 n 个点 p $_i$ D .x $_i$; y $_i$，其中 i = 1; 2; : : :; n，有 0 < x $_i^2$ C y $_i^2$ = 1。假设这些点是均匀分布的，也就是说，在圆盘的任何区域找到一个点的概率与该区域的面积成正比。设计一个平均运行时间为 ，.n/ 的算法，根据 n 个点与原点的距离 d $_i$ D p x $_i^2$ C y $_i^2$ ~~对它们~~进行排序。（*Hint:* 设计 BUCKET-SORT 中的桶大小以反映单位圆盘中点的均匀分布。）

8.4-6

随机变量 X 的 ***probability distribution function*** P .x/ 定义为 P .x/ D Pr fX x g 。假设你绘制一个包含 n 个随机变量的列表

$X_1; X_2; : : : ; X_n$ 来自于可在 O.1/ 时间内计算的连续概率分布函数 P（给定 y，你可以在 O.1/ 时间内找到 x，使得 P .x/ D y）。给出一个在线性平均时间内对这些数字进行排序的算法。

问题

8-1Probabilistic lower bounds on comparison sorting

在本题中，您将证明对 n 个不同输入元素进行任何确定性或随机比较排序的运行时间的概率下限为 .n lg n/。您将首先检查具有决策树 T_A 的确定性比较排序 A。假设 A 的输入的每个排列都具有同等可能性。

a.假设 T_A 的每个叶子都标有给定随机输入时到达的概率。证明恰好 n Š 个叶子标记为 1/n Š，其余叶子标记为 0。

b.令 D.T / 表示决策树 T 的外部路径长度（即 T 所有叶子节点的深度之和）。令 T 为具有 k > 1 个叶子节点的决策树，令 *LT* 和 *RT* 分别为 T 的左子树和右子树。证明 D.T / D D.*LT* / C D.*RT* / C k。

c.令 d.k/ 为所有具有 k > 1 个叶子节点的决策树 T 中 D.T / 的最小值。证明 d.k/ D min fd.i / C d.k i / C k W 1 i k 1g。（*Hint:* 考虑一棵具有 k 个叶子节点的决策树 T，该决策树达到最小值。令 i_0 为 *LT* 中的叶子节点数，令 k i_0 为 *RT* 中的叶子节点数。）

d.证明：对于给定的 k > 1 值，且 i 在 1 i k 1 范围内，函数 i lg i C .k i / lg.k i / 在 i D k/2 处最小化。得出 d.k/ D .k lg k/ 的结论。

e.证明 D.T_A / D .n Š lg.n Š //，并得出结论，对 n 个元素进行排序的平均时间为 .n lg n/。

现在考虑一个 *randomized* 比较排序 B 。我们可以通过合并两种类型的节点来扩展决策树模型以处理随机化：普通比较节点和 “随机化” 节点。随机化节点模拟算法 B 做出的随机选择，形式为 RANDOM.1; r / 。该节点有 r 个子节点，每个子节点在算法执行期间被选中的概率相同。

f.证明对于任何随机比较排序 B ，存在一个确定性比较排序 A ，其预期比较次数不超过 B 的比较次数。

8-2Sorting in place in linear time

您需要对一个包含 n 条数据记录的数组进行排序，每条记录的键均为 0 或 1 。对此类记录集进行排序的算法可能具有以下三个理想特征的某个子集：

1. 该算法在 O.n/ 时间内运行。2. 该算法是稳定的。3. 该算法在原地排序，除了原始数组之外，仅使用常量大小的存储空间。***a.*** 给出一个满足上述标准 1 和 2 的算法。***b.*** 给出一个满足上述标准 1 和 3 的算法。***c.*** 给出一个满足上述标准 2 和 3 的算法。***d.*** 你能否使用 (a)-(c) 部分中的任何排序算法作为 RADIX-SORT 第 2 行中使用的排序方法，以便 RADIX-SORT 在 O.b n/ 时间内对具有 b 位键的 n 条记录进行排序？解释如何做到或为什么不行。***e.*** 假设 n 条记录的键在 1 到 k 范围内。说明如何修改计数排序，以便它在 O.n C k/ 时间内对记录进行原地排序。您可以在输入数组之外使用 O.k/ 存储。您的算法稳定吗？

8-3Sorting variable-length items

a.给定一个整数数组，其中不同的整数可能具有不同的位数，但数组中整数的位数之和为 n。说明如何在 O.n/ 时间内对数组进行排序。

b.给定一个字符串数组，其中不同的字符串可能具有不同数量的字符，但所有字符串的字符总数为 n。说明如何在 O.n/ 时间内对字符串进行排序。（所需顺序是标准字母顺序：例如，a < ab < b。）

8-4Water jugs

给你 n 个红色和 n 个蓝色水壶，它们的形状和大小各不相同。所有红色水壶的容量都不同，所有蓝色水壶的容量也不同，你无法根据水壶的大小判断它能装多少水。此外，对于每个一种颜色的水壶，都有一个另一种颜色的水壶可以装相同量的水。

你的任务是将水壶分成两对，红水壶和蓝水壶，每对水壶的容量相同。为此，你可以执行以下操作：挑选一对

两个水壶，一个是红色的，一个是蓝色的，将水倒入红色水壶，然后将水倒入蓝色水壶。此操作会告诉您红色水壶或蓝色水壶是否能容纳更多的水，或者它们的容量是否相同。假设这种比较需要一个时间单位。您的目标是找到一种算法，该算法进行最少的比较来确定分组。请记住，您不能直接比较两个红色水壶或两个蓝色水壶。

a.描述一种确定性算法，该算法使用 .n ² / 次比较将水壶分组为两对。***b.*** 证明解决此问题的算法必须进行的比较次数的下限为 .n lg n/。***c.*** 给出一个随机算法，其预期比较次数为 O.n lg n/，并证明该界限是正确的。您的算法的最坏情况比较次数是多少？

8-5Average sorting

假设我们不对数组进行排序，而只是要求元素平均增加。更准确地说，如果对于所有 i D 1; 2; : : : ; n k 满足以下条件，则称 n 元素数组为 A k-***sorted***：

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

a.数组按 1 排序是什么意思？

b.给出数字 1；2；:::；10 的排列，该排列是 2 排序的，但无序的。

c.证明：一个 n 元素数组是 k 排序的当且仅当对于所有的 i D 1; 2; : : :; n k, AŒi AŒi C k 。

d.给出一个算法，在 O.n lg.n/k// 时间内对 n 元素数组进行 k 排序。

当 k 为常数时，我们还可以显示生成 k 排序数组的时间下限。

e.说明如何在 O.n lg k/ 时间内对长度为 n 的 k 排序数组进行排序。（*Hint:* 使用练习 6.5-11 的解决方案。）

f.证明当 k 为常数时，对 n 个元素的数组进行 k 排序需要 .n lg n/ 时间。（*Hint:* 使用部分 (e) 的解以及比较排序的下限。）

8-6Lower bound on merging sorted lists

合并两个排序列表的问题经常出现。我们在 2.3.1 节中看到了一个合并过程，即子程序 MERGE。在本问题中，你将证明合并两个排序列表（每个列表包含 n 个项目）所需的最坏情况比较次数的下限为 2n 1。首先，你将使用决策树显示 2n 1 的下限。

a.给定 2n 个数字，计算将它们分成两个排序列表的可能方法数，每个列表有 n 个数字。

b.使用决策树和你对部分 (a) 的答案，表明任何正确合并两个排序列表的算法都必须执行至少 2n o.n/ 次比较。

现在你将展示一个稍微更紧密的 2n 1 界限。

c.证明如果两个元素按排序顺序连续并且来自不同的列表，则必须对它们进行比较。***d.*** 使用部分 (c) 的答案来证明合并两个排序列表的下限为 2n 1 次比较。

8-7The 0-1 sorting lemma and columnsort

对两个数组元素 AŒi 和 AŒj （其中 i < j ）进行 ***compare-exchange*** 运算，其形式为

比较交换 .A; i; j /

1 如果 AŒi > AŒj 2 交换 AŒi 与 AŒj

经过比较交换操作，我们知道 AŒi AŒj 。

oblivious compare-exchange algorithm 仅通过一系列预先指定的比较交换操作进行操作。序列中比较位置的索引必须事先确定，尽管它们可以取决于要排序的元素数量，但不能取决于要排序的值，也不能取决于任何先前的比较交换操作的结果。例如，上页的 COMPARE-EXCHANGE-INSERTION-SORT 过程显示了插入排序的一种变体，即无意识的比较交换算法。（与第 19 页的 INSERTION-SORT 过程不同，无意识版本在所有情况下的运行时间为 ，.n ^{2} / 秒。）

0-1 sorting lemma 提供了一种强有力的方法来证明无意识比较交换算法可以产生排序结果。它指出，如果无意识比较交换算法正确地对所有仅由 0 和 1 组成的输入序列进行排序，那么它就可以正确地对所有包含任意值的输入进行排序。

```
比较-交换-插入-排序.A; n/
1 为 i D 2 至 n 2 为 j D i  1 至 1 3 比较交换 .A; j
; j C 1/
```

您将通过证明其逆否命题来证明 0-1 排序引理：如果一个无意识的比较交换算法无法对包含任意值的输入进行排序，那么它就无法对某些 0-1 输入进行排序。假设无意识的比较交换算法 X 无法正确对数组 AŒ1 W n 进行排序。令 AŒp 为算法 X 放入错误位置的 A 中的最小值，令 AŒq 为算法 X 移动到 AŒp 应该进入的位置的值。按如下方式设计一个由 0 和 1 组成的数组 BŒ1 W n ：

$$B[i] = \begin{cases} 0 & \text{if } A[i] \le A[p] , \\ 1 & \text{if } A[i] > A[p] . \end{cases}$$

a.论证 AŒq > AŒp ，使得 BŒp D 0 且 BŒq D 1。

b.为了完成 0-1 排序引理的证明，证明算法 X 无法对数组 B 正确排序。

现在，你将使用 0-1 排序引理来证明特定排序算法正确运行。算法 ***columnsort*** 适用于包含 n 个元素的矩形数组。该数组有 r 行和 s 列（因此 n D r s），但受三个限制：

r 必须为偶数，•s 必须是 r 的
因数，且 •$r \ge 2s^2$。

当列排序完成时，数组按 ***column-major order*** 排序：依次向下读取每一列，从左到右，元素单调增加。

列排序分 8 步进行，与 n 的值无关。奇数步都一样：对每列单独排序。每个偶数步都是一个固定排列。步骤如下：

1. 对各列进行排序。
2. 转置数组，但将其重新整形为 r 行 s 列。换句话说，按顺序将最左边的列转换为顶部 r/s 行；按顺序将下一列转换为接下来的 r/s 行；依此类推。

10 14 5	4 1 2	4 8 10	1 3 6	1 4 11
8 7 17	8 3 5	12 16 18	2 5 7	3 8 14
12 1 6	10 7 6	1 3 7	4 8 10	6 10 17
16 9 11	12 9 11	9 14 15	9 13 15	2 9 12
4 15 2	16 14 13	2 5 6	11 14 17	5 13 16
18 3 13	18 15 17	11 13 17	12 16 18	7 15 18
(a)	(b)	(c)	(d)	(e)

1 4 11	5 10 16	4 10 16	1 7 13
2 8 12	6 13 17	5 11 17	2 8 14
3 9 14	7 15 18	6 12 18	3 9 15
5 10 16	1 4 11	1 7 13	4 10 16
6 13 17	2 8 12	2 8 14	5 11 17
7 15 18	3 9 14	3 9 15	6 12 18
(f)	(g)	(h)	(i)

图 8.5 列排序的步骤。（a）输入数组有 6 行 3 列。（此示例不遵循 $r \geq 2s^2$ 要求，但它可以工作。）（b）在步骤 1 中对每一列进行排序之后。（c）在步骤 2 中转置和重塑之后。（d）在步骤 3 中对每一列进行排序之后。（e）执行步骤 4 之后，该步骤反转了步骤 2 中的排列。（f）在步骤 5 中对每一列进行排序之后。（g）在步骤 6 中移动半列之后。（h）在步骤 7 中对每一列进行排序之后。（i）执行步骤 8 之后，该步骤反转了步骤 6 中的排列。步骤 6-8 将每一列的下半部分与下一列的上半部分进行排序。步骤 8 之后，数组按列主序排序。

3. 对每列进行排序。

4. 执行步骤 2 中排列的逆操作。

5. 对每列进行排序。

6. 将每列的上半部分移入同一列的下半部分，并将每列的下半部分移入右侧下一列的上半部分。将最左侧列的上半部分留空。将最后一列的下半部分移入新的最右侧列的上半部分，并将该新列的下半部分留空。

7. 对每列进行排序。

8. 执行步骤 6 中排列的逆操作。

可以将步骤 638 视为对每列的下半部分和下一列的上半部分进行排序的单个步骤。图 8.5 显示了 r D 6 和 s D 3 的列排序步骤示例。（尽管此示例违反了 $r \geq 2s^2$ 的要求，但它恰好可以工作。）

c.认为我们可以将列排序视为一种无意识的比较交换算法，即使我们不知道奇数步骤使用了什么排序方法。

尽管似乎很难相信列排序确实可以排序，但您可以使用 0-1 排序引理来证明它确实可以排序。0-1 排序引理适用，因为我们可以将列排序视为一种无意识的比较交换算法。一些定义将帮助您应用 0-1 排序引理。如果我们知道数组的区域全为 0 或全为 1，或者为空，则称该数组的区域为 ***clean***。否则，该区域可能包含混合的 0 和 1，并且它是 ***dirty***。从这里开始，假设输入数组仅包含 0 和 1，并且我们可以将其视为具有 r 行和 s 列的数组。

d.证明在步骤 133 之后，数组由顶部的干净行 0、底部的干净行 1 以及它们之间最多 s 个脏行组成。（其中一个干净行可能为空。）***e.*** 证明在步骤 4 之后，按列主序读取的数组以 0 的干净区域开始，以 1 的干净区域结束，中间具有最多 s^2 个元素的脏区域。（同样，其中一个干净区域可能为空。）***f.*** 证明步骤 538 产生完全排序的 0-1 输出。得出结论，列排序可以正确对包含任意值的所有输入进行排序。***g.*** 现在假设 s 不能整除 r。证明在步骤 133 之后，数组由顶部的干净行 0、底部的干净行 1 以及它们之间最多 2s 1 个脏行组成。（再次，其中一个干净区域可以是空的。）当 s 不能整除 r 时，r 与 s 相比必须有多大，才能使列排序正确排序？***h.*** 建议对步骤 1 进行简单更改，使我们即使当 s 不能整除 r 时也能保持要求 $r \approx 2s^2$，并证明使用您的更改后，列排序可以正确排序。

章节注释

用于研究比较排序的决策树模型是由 Ford 和 Johnson [150] 提出的。Knuth 关于排序的综合论文 [261] 涵盖了排序问题的许多变体，包括本文给出的排序复杂度的信息论下限。Ben-Or [46] 使用决策树模型的泛化研究了排序的下限。

Knuth 认为 H. H. Seward 在 1954 年发明了计数排序，并提出了将计数排序与基数排序相结合的想法。从最低有效数字开始的基数排序似乎是广泛用于以下运算符的民间算法：

机械卡片分类机。根据 Knuth 的说法，最早发表的关于该方法的参考文献是 L. J. Comrie 于 1929 年发表的一篇描述穿孔卡片设备的论文。桶式分类自 1956 年起开始使用，当时 Isaac 和 Singleton 提出了其基本思想 [235]。

Munro 和 Raman [338] 给出了一种稳定的排序算法，在最坏情况下执行 $O(n^{1+\epsilon})$ 次比较，其中 $0 < \epsilon \leq 1$ 为任意固定常数。尽管任何 $O(n \lg n)$ 次算法进行的比较次数都较少，但 Munro 和 Raman 的算法仅移动数据 $O(n)$ 次，并且原地操作。

许多研究人员都考虑过在 $o(n \lg n)$ 时间内对 n 个 b 位整数进行排序的情况。他们已经获得了一些积极的结果，每个结果都基于对计算模型和算法限制的略有不同的假设。所有结果都假设计算机内存被分成可寻址的 b 位字。Fredman 和 Willard [157] 引入了融合树数据结构，并用它在 $O(n \lg n / \lg \lg n)$ 时间内对 n 个整数进行排序。Andersson [17] 后来将此界限改进为 $O(n \lg n)$ 时间。这些算法需要使用乘法和几个预先计算的常量。Andersson、Hagerup、Nilsson 和 Raman [18] 展示了如何在不使用乘法的情况下在 $O(n \lg \lg n)$ 时间内对 n 个整数进行排序，但是他们的方法需要的存储空间在 n 方面是无界的。使用乘法散列，我们可以将所需的存储空间减少到 $O(n)$，但 $O(n \lg \lg n)$ 的最坏情况运行时间界限变成了预期时间界限。通过推广 Andersson [17] 的指数搜索树，Thorup [434] 给出了一种 $O(n \lg \lg n)^2$ 时间排序算法，该算法不使用乘法或随机化，而是使用线性空间。将这些技术与一些新想法相结合，Han [207] 将排序界限改进为 $O(n \lg \lg n \lg \lg \lg n)$ 时间。虽然这些算法是重要的理论突破，但它们都相当复杂，目前似乎不太可能在实践中与现有的排序算法相媲美。

问题 8-7 中的列排序算法是由 Leighton [286] 提出的。

9 Medians and Order Statistics

一组 n 个元素的第 i 个 ***order statistic*** 是第 i 个最小元素。例如，一组元素的 ***minimum*** 是一阶统计量 (i D 1)，***maximum*** 是 n 阶统计量 (i D n)。非正式地说，***median*** 是该集合的 “中点”。当 n 为奇数时，中位数是唯一的，出现在 i D .n C1//2 处。当 n 为偶数时，有两个中位数，***lower median*** 出现在 i D n/2 处，***upper median*** 出现在 i D n/2 C 1 处。因此，无论 n 是否奇偶，中位数都出现在 i D b.n C 1//2c 和 i D d.n C 1//2e 处。不过，为了本文的简单起见，我们始终使用短语 “中位数” 来指代下中位数。

本章讨论从一组 n 个不同数字中选择第 i 个阶统计量的问题。为方便起见，我们假设该集合包含不同的数字，尽管我们所做的几乎所有事情都扩展到集合包含重复值的情况。我们正式指定 ***selection problem*** 如下：

输入：一个由 n 个不同数字 [1] 和一个整数 i 组成的集合 A，其中 1 = i = n。
输出：比 A 中恰好 i 1 个其他元素大的元素 x 2 A。

我们只需使用堆排序或归并排序对数字进行排序，然后输出排序数组中的第 i 个元素，即可在 O.n lg n/ 时间内解决选择问题。本章介绍了渐近更快的算法。

9.1 节研究了选择一组元素中的最小值和最大值的问题。更有趣的是一般的选择问题，我们将在接下来的两节中对其进行研究。9.2 节分析了一种实用的随机算法，该算法实现了 O.n/ 的预期运行时间，假设离散

[1] As in the footnote on page 182, you can enforce the assumption that the numbers are distinct by converting each input value $A[i]$ to an ordered pair $(A[i], i)$ with $(A[i], i) < (A[j], j)$ if either $A[i] < A[j]$ or $A[i] = A[j]$ and $i < j$.

tinct 元素。第 9.3 节包含一个更具理论意义的算法，该算法在最坏情况下可实现 O.n/ 的运行时间。

9.1 最小值和最大值

需要多少次比较才能确定 n 个元素集合中的最小值？要获得 n 1 次比较的上限，只需依次检查集合中的每个元素并跟踪迄今为止看到的最小元素。MIN IMUM 过程假设集合位于数组 AŒ1 W n 中。

```
最小值.A; n/
1 min D AŒ1 2 for i D 2 t
o n 3 if min > AŒi 4
min D AŒi 5 return
min
```

通过 n 1 次比较找到最大值不再是困难的。

这个求最小值的算法是我们能做的最好的吗？是的，因为事实证明，确定最小值的问题存在 n 1 次比较的下限。将任何确定最小值的算法视为元素之间的竞赛。每次比较都是竞赛中的一场比赛，其中两个元素中较小的一个获胜。由于除获胜者之外的每个元素都必须输掉至少一场比赛，因此我们可以得出结论，确定最小值需要 n 1 次比较。因此，就执行的比较次数而言，算法 MINIMUM 是最佳的。

同时达到最小值和最大值

某些应用程序需要找到一组 n 个元素的最小值和最大值。例如，图形程序可能需要将一组 .x; y/ 数据缩放到矩形显示屏或其他图形输出设备上。为此，程序必须首先确定每个坐标的最小值和最大值。

当然，我们可以使用 2n 2 D 2n 次比较来确定 n 个元素的最小值和最大值。我们只需独立地查找最小值和最大值，对每个元素使用 n 1 次比较，总共进行 2n 2 D 2n 次比较。

尽管 2n 2 次比较是渐近最优的，但可以改进首项常数。我们最多使用 3bn /2c 次比较就可以找到最小值和最大值。诀窍是保留迄今为止看到的最小值和最大值。不是通过将输入中的每个元素与当前最小值和最大值进行比较来处理输入的每个元素（每个元素需要进行 2 次比较），而是成对处理元素。首先比较输入中的元素对 *with each other*，然后将较小的元素与当前最小值进行比较，将较大的元素与当前最大值进行比较，每 2 个元素需要进行 3 次比较。

如何设置当前最小值和最大值的初始值取决于 n 是奇数还是偶数。如果 n 为奇数，则将最小值和最大值都设置为第一个元素的值，然后成对处理其余元素。如果 n 为偶数，则对前 2 个元素执行 1 次比较以确定最小值和最大值的初始值，然后成对处理其余元素，就像奇数 n 的情况一样。

让我们计算一下比较的总数。如果 n 为奇数，则会发生 3 bn/2c 次比较。如果 n 为偶数，则会发生 1 次初始比较，随后再进行 3.n 2//2 次比较，总共 3n/2 2 次比较。因此，无论哪种情况，比较的总数最多为 3 bn/2c。

练习

9.1-1
证明在最坏情况下，通过 n C dlg ne 2 次比较即可找到 n 个元素中的第二小元素。（*Hint:* 也找到最小元素。）

9.1-2
给定 n > 2 个不同的数字，你想找到一个既不是最小值也不是最大值的数字。你需要执行的最少比较次数是多少？

9.1-3
赛马场可以同时举办五匹马的比赛，以确定它们的相对速度。假设具有传递性，则需要六场比赛才能确定最快的马（参见第 1159 页）。至少需要多少场比赛才能确定 25 匹马中最快的三匹？

9.1-4
证明在最坏情况下 d3n/2e 2 次比较的下限，以找到 n 个数字中的最大值和最小值。（*Hint:* 考虑有多少个数字可能是最大值或最小值，并研究比较如何影响这些计数。）

9.2 预期线性时间的选择

一般选择问题 4——对任何 i 值 4 找到第 i 阶统计量似乎比找到最小值这个简单的问题更困难。然而，令人惊讶的是，这两个问题的渐近运行时间是一样的：＇.n/＇。本节介绍一种选择问题的分而治之算法。RANDOMIZED-SELECT 算法仿照第 7 章的快速排序算法建模。像快速排序一样，它以递归方式对输入数组进行分区。但与快速排序递归处理分区的两侧不同，RANDOMIZED-SELECT 仅对分区的一侧进行处理。这种差异在分析中显现出来：快速排序的预期运行时间为＇.n lg n/，而 RANDOMIZED-SELECT 的预期运行时间为＇.n/（假设元素不同）。

RANDOMIZED-SELECT 使用 7.3 节中介绍的 RANDOMIZED-PARTITION 过程。与 RANDOMIZED-QUICKSORT 一样，它是一种随机算法，因为其行为部分由随机数生成器的输出决定。RANDOMIZED-SELECT 过程返回数组 AŒp W r 的第 i 个最小元素，其中 1 = i = r p C 1。

随机选择 .A; p; r; i /

1 如果 p == r 2 返回 AŒp // 1 i r p C 1 当 p == r 表示 i D 1 3 q D RANDOMIZED-PARTITION .A; p; r / 4 k D q p C 1 5 如果 i == k 6 返回 AŒq // 枢轴值是答案 7 否则如果 i < k 8 返回 RANDOMIZED-SELECT .A; p; q 1; i / 9 否则返回 RANDOMIZED-SELECT .A; q C 1; r; i k/

图 9.1 说明了 RANDOMIZED-SELECT 过程的工作原理。第 1 行检查递归的基本情况，其中子数组 AŒp W r 仅由一个元素组成。在这种情况下，i 必须等于 1，第 2 行仅返回 AŒp 作为第 i 个最小元素。否则，第 3 行对 RANDOMIZED-PARTITION 的调用将数组 AŒp W r 划分为两个（可能为空）子数组 AŒp W q 1 和 AŒq C 1 W r ，使得 AŒp W q 1 的每个元素都小于或等于 AŒq ，而后者又小于 AŒq C 1 W r 的每个元素。（尽管我们的分析假设元素是不同的，但即使存在相等的元素，该过程仍会得出正确的结果。）与快速排序一样，我们将 AŒq 称为 ***pivot*** 元素。第 4 行计算子数组 AŒp W q 中的元素数量 k，即

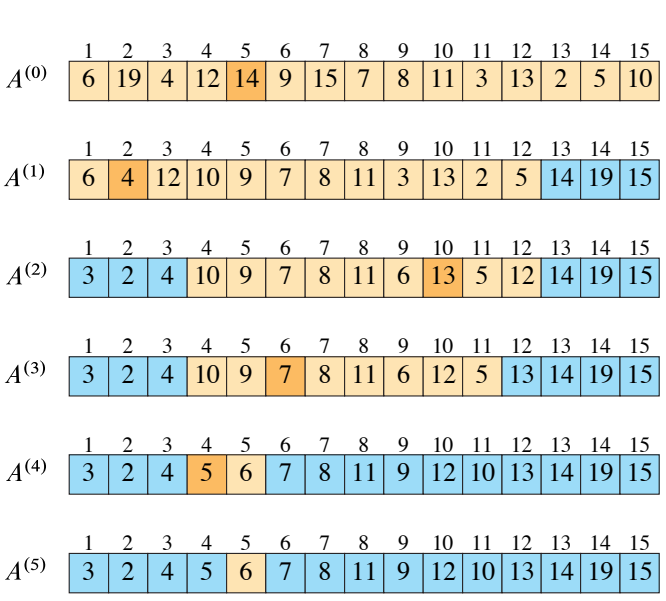


图 9.1 RANDOMIZED-SELECT 的操作，连续分区缩小了子数组 AŒp W r ，显示每次递归调用时参数 p、r 和 i 的值。每个递归步骤中的子数组 AŒp W r 都以棕褐色显示，深棕褐色元素被选为下一次分区的枢轴。蓝色元素在 AŒp W r 之外。答案是底部数组中的棕褐色元素，其中 p D r D 5 和 i D 1。数组名称 A (0) ；A (1) ；: : : ；A (5) 、分区数以及分区是否有用将在下一页中解释。

分区低端元素的数量，加上 1 作为枢轴元素。然后，第 5 行检查 AŒq 是否是第 i 个最小元素。如果是，则第 6 行返回 AŒq 。否则，算法确定第 i 个最小元素位于两个子数组 AŒp W q 1 和 AŒq C 1 W r 中的哪一个中。如果 i < k，则所需元素位于分区的低端，第 8 行从子数组中递归选择它。但是，如果 i > k，则所需元素位于分区的高端。由于我们已经知道小于 AŒp W r 的第 i 个最小元素的 k 个值（即 AŒp W q 的元素），因此所需元素是 AŒq C 1 W r 的第 .i k/小元素，该元素递归执行 9 遍。代码似乎允许对具有 0 个元素的子数组进行递归调用，但练习 9.2-1 要求您证明这种情况不会发生。

RANDOMIZED-SELECT 的最坏情况运行时间为 ，.n 2 /，即使找到最小值也是如此，因为它可能非常不走运，在只剩下一个元素时，总是先对剩余的最大元素进行划分，然后才确定第 i 个最小元素。在这种最坏情况下，每个递归步骤仅从考虑中移除枢轴。因为划分 n 个元素需要 ，.n/ 时间，所以最坏情况运行时间的递归与 QUICKSORT 相同：

T .n/ D T .n 1/ C ′ .n/，其解为 T .n/ D ′ .n ²/。我们将看到，该算法具有线性预期运行时间，但是，由于它是随机的，因此没有特定的输入会引起最坏情况的行为。

为了直观地了解线性预期运行时间背后的原理，假设每次算法随机选择一个主元时，主元都位于第二和第三四分位数4（按排序顺序排列的剩余元素的中间一半“4）的某个位置。如果第 i 个最小元素小于主元，则在所有未来的递归调用中忽略所有大于主元的元素。这些被忽略的元素至少包括最上四分位数，甚至可能更多。同样，如果第 i 个最小元素大于主元，则在所有未来的递归调用中忽略所有小于主元4（至少是第一四分位数4）的元素。因此，无论哪种方式，在所有未来的递归调用中都会忽略剩余元素中的至少 1/4，从而最多剩下 3/4 个剩余元素 ***in play***：驻留在子数组 AŒp W r 中。由于 RANDOMIZED-PARTITION 在 n 个元素的子数组上花费 ，.n/ 时间，最坏情况运行时间的递归式为 T .n/ D T .3n/4/ C ，.n/。根据主方法的第 3 种情况（第 102 页的定理 4.1），该递归式的解为 T .n/ D ，.n/。

当然，主元不一定每次都落在中间一半。由于主元是随机选择的，因此它每次落入中间一半的概率约为 1/2。我们可以将选择主元的过程视为伯努利试验（参见 C.4 节），成功等同于主元位于中间一半。因此，成功所需的预期试验次数由几何分布给出：平均只需两次试验（第 1197 页的方程 (C.36)）。换句话说，我们预计一半的分割会使仍在起作用的元素数量至少减少 3/4，而另一半的分割不会有太大帮助。因此，预期分割次数最多是主元始终落入中间一半的情况的两倍。每次额外分割的成本都小于之前的分割成本，因此预期运行时间仍为 ，.n/。

为了使上述论证严谨，我们首先将随机变量 $A^{(j)}$ 定义为 A 中在 j 次分区之后仍在起作用的元素集合（即，在调用 j 次 RANDOMIZED-SELECT 之后的子数组 AŒp W r 内），以便 $A^{(0)}$ 由 A 中的所有元素组成。由于每次分区都会从起作用的元素4（即主元4）中移除至少一个元素，因此序列 $jA^{(0)}j; jA^{(1)}j; jA^{(2)}j; : : :$ 严格减少。集合 $A^{(j-1)}$ 在第 j 次分区之前起作用，集合 $A^{(j)}$ 之后仍然起作用。为方便起见，假设初始集合 $A^{(0)}$ 是第 0 次“虚拟”分区的结果。

如果 $jA^{(j)}j \quad .3/4/jA^{(j-1)}j$，我们将第 j 个分区称为 ***helpful***。图 9.1 显示了集合 $A^{(j)}$ 以及分区对示例数组是否有用。有用的分区对应于成功的伯努利试验。以下引理表明分区至少有同样的可能性是有用的。

Lemma9.1
分区有帮助的概率至少为 1/2。

Proof分割是否有用取决于随机选择的枢轴。我们在上面的非正式论证中讨论了“ 中间一半 ”。让我们更精确地将 n 元素子数组的中间一半定义为除最小 dn/ 4e 1 和最大 dn/ 4e 1 元素之外的所有元素（即，如果子数组已排序，则除第一个 dn/ 4e 1 和最后一个 dn/ 4e 1 元素之外的所有元素）。我们将证明，如果枢轴落入中间一半，则枢轴会导致有用的分割，我们还将证明枢轴落入中间一半的概率至少为 1/2。

无论主元落在哪里，所有大于它的元素或所有小于它的元素以及主元本身在分割后都将不再起作用。因此，如果主元落入中间一半，则至少有 dn/4e 1 个小于主元的元素或 dn/4e 1 个大于主元加上主元的元素在分割后将不再起作用。也就是说，至少有 dn/4e 个元素将不再起作用。剩余起作用的元素数最多为 n dn/4e，根据第 70 页的练习 3.3-2 等于 b3n/4c。由于 b3n/4c = 3n /4，因此分割是有帮助的。

为了确定随机选择的枢轴点落入中间一半的概率的下限，我们确定了它不落入中间一半的概率的上限。这个概率是

$$\begin{aligned}\frac{2(\lceil n/4\rceil - 1)}{n} &\le \frac{2((n/4+1)-1)}{n} \quad \text{(by inequality (3.2) on page 64)}\\ &= \frac{n/2}{n}\\ &= 1/2\,.\end{aligned}$$

因此，主元落入中间一半的概率至少为 1/2，因此分区有帮助的概率至少为 1/2。 ■

我们现在可以限制 R ANDOMIZED-SELECT 的预期运行时间。

Theorem9.2
对包含 n 个不同元素的输入数组执行 RANDOMIZED-SELECT 过程的预期运行时间为 ，.n/。

Proof由于并非每种划分都一定有用，我们给每种划分一个从 0 开始的索引，并用 hh $_0$; h $_1$; h $_2$; : : :; h $_m$ i 表示有用的划分序列，以便第 h $_k$ 个划分对 k D 0; 1; 2; : : :; m 有帮助。虽然有用划分的数量 m 是随机变量，但

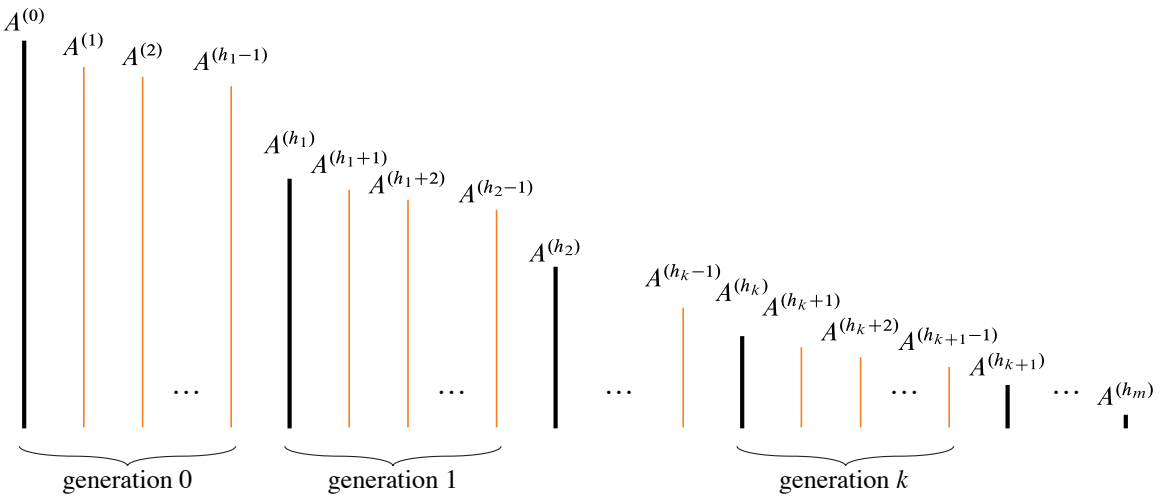


图 9.2 定理 9.2 证明中每一代中的集合。垂直线表示集合，每条线的高度表示集合的大小，等于发挥作用的元素数。每一代都以集合 A $^{(h_k)}$ 开始，这是有用的分区的结果。这些集合用黑色绘制，其大小最多是其左侧紧邻集合的 3/4。用橙色绘制的集合不是一代中的第一个集合。一代可能只包含一个集合。第 k 代中的集合是 A $^{(h_k)}$ ；A $^{(h_k+1)}$ ；:::；A $^{(h_{k+1}-1)}$ 。集合 A $^{(h_k)}$ 定义为 jA $^{(h_k)}$ j = .3/4/jA $^{(h_{k-1})}$ j。如果分区一直进行到第 h $_m$ 代，则集合 A $^{(h_m)}$ 最多有一个发挥作用的元素。

可以，我们可以限制它，因为在最多 dlog $_{4/3}$ ne 次有用的分区之后，只有一个元素仍在发挥作用。将虚拟的第 0 次分区视为有用，因此 h $_0$ D 0。用 n $_k$ 表示 jA $^{(h_k)}$ j，其中 n $_0$ D jA $^{(0)}$ j 是原始问题大小。由于第 h $_k$ 次分区是有用的，并且集合 A $^{(j)}$ 的大小严格减少，因此我们有 n $_k$ D jA $^{(h_k)}$ j .3/4/jA $^{(h_k-1)}$ j D .3/4/ n $_{k-1}$ ，其中 k D 1; 2; : : : ; m。通过迭代 n $_k$ .3/4/ n $_{k-1}$ ，我们可以得到 n $_k$ .3/4^k n $_0$ 。

如图 9.2 所示，我们将集合 A $^{(j)}$ 对于 j D 0; 1; 2; : : : 序列拆分为 m ***genera- tions*** 个连续划分的集合，从有用划分的结果 A $^{(h_k)}$ 开始，到下一个有用划分之前的最后一个集合 A $^{(h_{k+1}-1)}$ 结束，这样第 k 代的集合为 A $^{(h_k)}$ ；A $^{(h_k+1)}$ ；:::；A $^{(h_{k+1}-1)}$ 。然后，对于第 k 代中的每个元素集合 A $^{(j)}$ ，我们有 jA $^{(j)}$ j # jA $^{(h_k)}$ j D n $_k$ # .3/4/^{k} n $_0$ 。

接下来，我们定义随机变量

$$X_k = h_{k+1} - h_k$$

其中，k≤D≤0；1；2；:::;m≥1。即X_k为第k代集合的数量，因此第k代集合为A$^{(h_k)}$ ；A$^{(h_k+1)}$ ；:::；A$^{(h_k+X_k-1)}$ 。

根据引理 9.1，分区有用的概率至少为 1/2。实际上，概率甚至更高，因为即使枢轴

不落入中间的一半，但第 i 个最小元素恰好位于分割的较小一侧。不过，我们只使用 1/2 的下限，然后公式 (C.36) 给出 E ŒX $_k$ 2，其中 k D 0; 1; 2; : : : ; m 1。

由于运行时间主要由比较决定，因此我们来推导一下分割过程中总共进行了多少次比较的上限。由于我们正在计算上限，因此假设递归一直进行到只剩下一个元素在起作用。第 j 次分割采用起作用的元素集合 A $^{(j-1)}$，并将随机选择的枢轴与所有其他 jA $^{(j-1)}$j 元素进行比较，因此第 j 次分割进行的比较少于 jA $^{(j-1)}$j 次。第 k 代集合的大小为 jA $^{(h_k)}$j; jA $^{(h_k+1)}$j; : : :; jA $^{(h_k+X_k-1)}$j。因此，分割过程中的总比较次数小于

$$\begin{aligned}\sum_{k=0}^{m-1}\sum_{j=h_k}^{h_k+X_k-1}|A^{(j)}| &\le \sum_{k=0}^{m-1}\sum_{j=h_k}^{h_k+X_k-1}|A^{(h_k)}| \\ &= \sum_{k=0}^{m-1}X_k\,|A^{(h_k)}| \\ &\le \sum_{k=0}^{m-1}X_k\left(\frac{3}{4}\right)^k n_0\ .\end{aligned}$$

由于 E ŒX $_k$ #2，我们有分区期间预期的总比较次数小于

$$\begin{aligned}\mathrm{E}\left[\sum_{k=0}^{m-1}X_k\left(\frac{3}{4}\right)^k n_0\right] &= \sum_{k=0}^{m-1}\mathrm{E}\left[X_k\left(\frac{3}{4}\right)^k n_0\right] \quad \text{(by linearity of expectation)} \\ &= n_0\sum_{k=0}^{m-1}\left(\frac{3}{4}\right)^k \mathrm{E}\,[X_k] \\ &\le 2n_0\sum_{k=0}^{m-1}\left(\frac{3}{4}\right)^k \\ &< 2n_0\sum_{k=0}^{\infty}\left(\frac{3}{4}\right)^k \\ &= 8n_0 \qquad \text{(by equation (A.7) on page 1142)}\ 。\end{aligned}$$

由于 n_0 是原始数组 A 的大小，我们得出结论，RANDOMIZED-SELECT 的预期比较次数（因此预期运行时间为 O.n/）。在第一次调用 RANDOMIZED -SELECT 时，将检查所有 n 个元素

PARTITION，给出下限为 .n/。因此预期运行时间为 .n/。

■

练习

9.2-1表明 RANDOMIZED-SELECT 永远不会对 0 长度数组进行递归调用。

9.2-2编写 RANDOMIZED-SELECT 的迭代版本。

9.2-3
假设使用 RANDOMIZED-SELECT 选择数组 A D h2; 3; 0; 5; 7; 9; 1; 8; 6; 4i 的最小元素。描述导致 RANDOMIZED-SELECT 最差性能的分区序列。

9.2-4
论证 RANDOMIZED-SELECT 的预期运行时间不依赖于其输入数组 AŒp W r 中元素的顺序。也就是说，对于输入数组 AŒp W r 的任何排列，预期运行时间都是相同的。（*Hint:* 通过对输入数组的长度 n 进行归纳论证。）

9.3 最坏情况线性时间的选择

现在，我们将研究一种引人注目且理论上有趣的选择算法，其运行时间在最坏情况下为 .n/。尽管第 9.2 节中的 RANDOMIZED-SELECT 算法实现了线性预期时间，但我们发现其在最坏情况下的运行时间是二次的。本节介绍的选择算法在最坏情况下实现了线性时间，但它并不像 RANDOMIZED-SELECT 那样实用。它主要具有理论意义。

与预期的线性时间 RANDOMIZED-SELECT 一样，最坏情况线性时间算法 SELECT 通过递归划分输入数组来找到所需元素。然而，与 RANDOMIZED-SELECT 不同，SELECT *guarantees* 通过在划分数组时选择可证明的良好枢轴来获得良好的分割。该算法的巧妙之处在于它以递归方式找到枢轴。因此，SELECT 有两次调用：一次用于找到一个好的枢轴，第二次用于递归地找到所需的顺序统计。

SELECT 使用的分区算法类似于快速排序中的确定性分区算法 PARTITION（参见第 7.1 节），但经过修改，将要分区的元素作为附加输入参数。与 PARTITION 一样,

PARTITION-AROUND 算法返回枢轴的索引。由于它与 PARTITION 非常相似，因此省略了 PARTITION-AROUND 的伪代码。

SELECT 过程以包含 n D r p C 1 个元素的子数组 AŒp W r] 和介于 1 至 i 至 n 之间的整数 i 作为输入。它返回 A 中第 i 个最小元素。伪代码实际上比乍一看更容易理解。

```
SELECT(A, p, r, i)
 1  while (r − p + 1) mod 5 ≠ 0
 2      for j = p + 1 to r                    // put the minimum into A[p]
 3          if A[p] > A[j]
 4              exchange A[p] with A[j]
 5      // If we want the minimum of A[p : r], we're done.
 6      if i == 1
 7          return A[p]
 8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
 9      p = p + 1
10      i = i − 1
11  g = (r − p + 1)/5                         // number of 5-element groups
12  for j = p to p + g − 1                    // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)          // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                           // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

伪代码首先执行 1310 行的 while 循环，以减少子数组中元素的数量 r p C 1 ，直到它可以被 5 整除。while 循环执行 0 到 4 次，每次重新排列 AŒp W r 的元素，以使 AŒp 包含最小元素。如果 i D 1，这意味着我们实际上想要最小元素，那么该过程只会在第 7 行返回它。否则，SELECT 会从子数组 AŒp W r 中消除最小值并迭代以找到 AŒp C 1 W r 中的第 .i 个元素。第 9310 行通过增加 p 和减少 i 来实现这一点。如果 while 循环完成所有迭代而没有返回

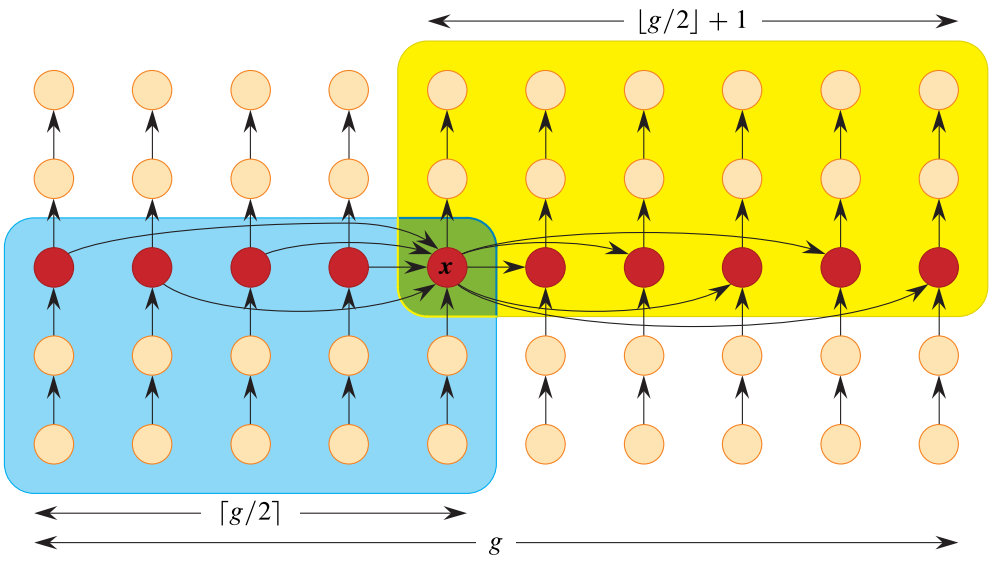


图 9.3 选择算法 SELECT 的第 17 行之后元素之间的关系（以圆圈表示）。共有 g D .r p C 1/ /5 组，每组 5 个元素，每组占据一列。例如，最左边的列包含元素 AŒp 、AŒp C g 、AŒp C 2g 、AŒp C 3g 、AŒp C 4g ，下一列包含 AŒp C 1 、AŒp C g C 1 、AŒp C 2g C 1 、AŒp C 3g C 1 、AŒp C 4g C 1 。各组的中位数为红色，标有枢轴 x。箭头从较小元素指向较大元素。蓝色背景上的元素均已知小于或等于 x，并且不能落入围绕 x 的分区的高端。黄色背景上的元素已知大于或等于 x，并且不能落入 x 周围分区的低侧。枢轴 x 属于蓝色和黄色区域，显示在绿色背景上。白色背景上的元素可以位于分区的任一侧。

结果，程序执行了第 11324 行中的算法核心，确保 AŒp W r 中元素的数量 r p C 1 可以被 5 整除。

算法的下一部分实现了以下想法，如图 9.3 所示。将 AŒp W r 中的元素分成 g D .r p C1//5 组，每组 5 个元素。第一个 5 元素组是

$$\langle A[p], A[p+g], A[p+2g], A[p+3g], A[p+4g] \rangle ,$$

第二个是

$$\langle A[p+1], A[p+g+1], A[p+2g+1], A[p+3g+1], A[p+4g+1] \rangle ,$$

等等，直到最后，也就是

$$\langle A[p+g-1], A[p+2g-1], A[p+3g-1], A[p+4g-1], A[r] \rangle .$$

(请注意，r D p C 5g 1.) 第 13 行使用插入排序（第 2.1 节）等方法对每个组进行排序，因此对于 j D p; p C 1; : : : ; p C g 1，我们有

$A[j] \le A[j+g] \le A[j+2g] \le A[j+3g] \le A[j+4g]$.

图 9.3 中的每一列都代表一个排序好的 5 元素组。每个 5 元素组的中位数为 AŒj C 2g ，因此所有 5 元素中位数（以红色显示）都位于 AŒp C 2g W p C 3g 1 范围内。

接下来，第 16 行通过递归调用 SELECT 来查找 g 组中位数的中位数（具体来说，即第 dg/2 个最小值），从而确定主元 x。第 17 行使用改进的 PARTITION-AROUND 算法围绕 x 分割 AŒp W r 中的元素，返回 x 的索引 q ，这样 AŒq D x ，AŒp W q 中的元素最多为 x ，AŒq W r 中的元素大于或等于 x。

代码的其余部分与 RANDOMIZED-SELECT 的代码相同。如果主元 x 是第 i 个大数，则该过程返回它。否则，该过程在 AŒp W q 1 或 AŒq C 1 W r 上递归调用自身，具体取决于 i 的值。

让我们分析一下 SELECT 的运行时间，看看枢轴 x 的明智选择如何保证其最坏情况的运行时间。

Theorem9.3
奔跑 对 n 个元素的输入执行 SELECT 操作的时间为，.n/.

Proof将 T.n/ 定义为对任意输入子数组 AŒp W r （其大小最多为 n）运行 SELECT 的最坏情况时间，即 r p C 1 = n。根据此定义，T.n/ 单调递增。

我们首先确定第 16、23 和 24 行中递归调用之外所花费时间的上限。第 1 3-10 行中的 while 循环执行 0 到 4 次，即 O.1/ 次。由于循环内的主要时间是第 2-3-4 行中计算最小值，这花费了 1/2.n/ 次时间，因此第 13-10 行的执行时间为 O.1/2 = 1/2.n/D O.n/ 次。第 12-3-13 行中 5 元素组的排序花费了 1/2.n/ 次，因为每个 5 元素组都需要 1/2.1/ 次进行排序（即使使用插入排序等渐近低效的排序算法），并且有 g 个元素需要排序，其中 n/5 1 < g = n/5。最后，第 17 行的分割时间为 ，.n/，如第 187 页练习 7.1-3 要求您显示的那样。由于剩余的簿记工作仅花费 ，.1/ 时间，因此递归调用之外的总花费时间为 O.n/ C ，.n/ C ，.n/ C ，.1/ D ，.n/。

现在让我们确定递归调用的运行时间。第 16 行中查找枢轴的递归调用耗时 T .g/ T .n/5/，因为 g n/5 和 T .n/ 单调递增。第 23 行和第 24 行中的两个递归调用最多执行一个。但我们会看到，无论实际执行这两个 SELECT 递归调用中的哪一个，递归调用中的元素数量最多为 7n/10，因此第 23 行和第 24 行的最坏情况成本最多为 T .7n/10/。现在让我们展示使用组中位数的机制和选择枢轴 x 作为组中位数的中位数可确保此属性。

图 9.3 有助于直观地了解正在发生的事情。共有 g # n/5 个组，每个组包含 5 个元素，每组显示为从下到上排序的列。箭头显示列内元素的顺序。列从左到右排序，x 的组左侧的组的组中位数小于 x ，而 x 的组右侧的组的组中位数大于 x 。虽然每个组内的相对顺序很重要，但是 x 列左侧组之间的相对顺序并不重要，x 列右侧组之间的相对顺序也不重要。重要的是，左侧组的组中位数小于 x（由进入 x 的水平箭头表示），而右侧组的组中位数大于 x（由离开 x 的水平箭头表示）。因此，黄色区域包含我们知道大于或等于 x 的元素，而蓝色区域包含我们知道小于或等于 x 的元素。

这两个区域各自包含至少 3g/2 个元素。黄色区域中的组中位数数量为 bg /2c C 1，对于每个组中位数，有两个额外元素大于它，总共为 3.bg/2c C 1/ 3g/2 个元素。同样，蓝色区域中的组中位数数量为 dg/2e，对于每个组中位数，有两个额外元素小于它，总共为 3 dg/2e 3g/2。

黄色区域中的元素不能落入以 x 为中心的分区的低端，蓝色区域中的元素不能落入高端。两个区域内的元素4（即位于白色背景上的元素4）都不能落入分区的任何一侧。但由于分区低端不包括黄色区域中的元素，总共有 5g 个元素，因此我们知道分区低端最多可包含 5g 3g/2D 7g/2 7n/10 个元素。同样，分区高端不包括蓝色区域中的元素，类似的计算表明它也最多包含 7 n/10 个元素。

所有这些都会导致 SELECT 的最坏情况运行时间重复出现：

$$T(n) \le T(n/5) + T(7n/10) + \Theta(n) \; . \tag{9.1}$$

我们可以通过代换证明 T .n/ D O.n/。[2] 更具体地说，我们将证明，对于某个适当大的常数 c > 0 和所有 n > 0， T .n/ = cn。将这个归纳假设代入递归（9. 1）的右边，并假设 n ≈ 5，得到

[2] We could also use the Akra-Bazzi method from Section 4.7, which involves calculus, to solve this recurrence. Indeed, a similar recurrence (4.24) on page 117 was used to illustrate that method.

$$\begin{aligned} T(n) &\le c(n/5) + c(7n/10) + \Theta(n) \\ &\le 9cn/10 + \Theta(n) \\ &= cn - cn/10 + \Theta(n) \\ &\le cn \end{aligned}$$

如果 c 足够大，使得 c/10 支配被 ，.n/ 隐藏的上限常数。除了这个约束之外，我们可以选择足够大的 c，使得对于所有 n 4，T .n/ cn，这是 SELECT 中递归的基本情况。因此，SELECT 的运行时间在最坏情况下为 O.n/，并且由于仅第 13 行就花费了 ，.n/ 时间，因此总时间为 ，.n/。 ■

与比较排序（参见第 8.1 节）一样，SELECT 和 RANDOMIZED-SELECT 仅通过比较元素即可确定有关元素相对顺序的信息。回想一下第 8 章，在比较模型中，排序平均需要 .n lg n/ 时间（参见问题 8-1）。第 8 章中的线性时间排序算法对输入的类型做出了假设。相反，本章中的线性时间选择算法不需要对输入的类型做出任何假设，只需要假设元素是不同的并且可以按照线性顺序成对比较。本章中的算法不受 .n lg n/ 下限的限制，因为它们无需对所有元素进行排序即可解决选择问题。因此，通过排序和索引解决选择问题（如本章简介中所述）在比较模型中是渐近低效的。

练习

9.3-1

在 SELECT 算法中，输入元素被分成 5 个组。说明如果将输入元素分成 7 个组而不是 5 个组，该算法可以在线性时间内完成。

9.3-2

假设 S ELECT 中 1310 行的预处理被 n 的基例 n_0 所取代，其中 n_0 是一个合适的常数；g 被选为 br p C 1//5c；AŒ5g W n 中的元素不属于任何群。证明虽然运行时间的递归变得更混乱，但它仍然可以解为 ，.n/。

9.3-3

说明如何使用 SELECT 作为子程序，使快速排序在最坏情况下在 O.n lg n/ 时间内运行，假设所有元素都是不同的。

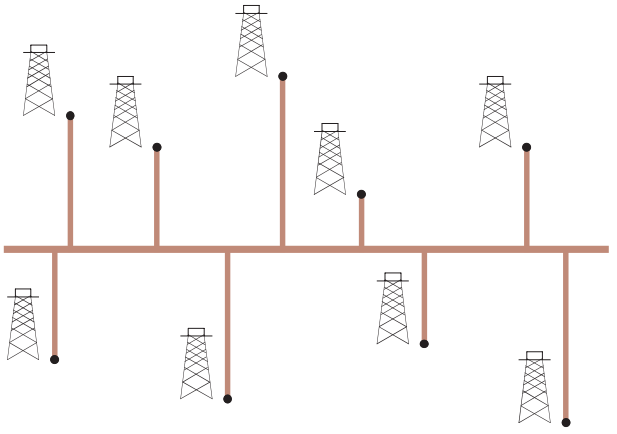


图 9.4 Olay 教授需要确定东西向输油管道的位置，使南北支线的总长度最小。

9.3-4

假设一个算法只使用比较来找出一组 n 个元素中的第 i 个最小元素。证明它也能找出第 i 1 个较小的元素和第 n 1 个较大的元素，而无需进行任何额外的比较。

9.3-5

说明如何仅使用 6 次比较来确定 5 元素集的中位数。

9.3-6

你有一个“黑盒”最坏情况线性时间中值子程序。给出一个简单的线性时间算法来解决任意顺序统计量的选择问题。

9.3-7

Olay 教授正在为一家石油公司提供咨询服务，该公司计划修建一条贯穿 n 个油井的油田的大型管道，从东向西。该公司希望将每口油井的支线管道沿最短路线（向北或向南）直接连接到主管道，如图 9.4 所示。给定油井的 x 和 y 坐标，教授应如何选择主管道的最佳位置以最小化支线的总长度？说明如何在线性时间内确定最佳位置。

9.3-8

一个 n 元素集合的第 k 个 ***quantiles*** 是将有序集合分成 k 个大小相等的集合（精度在 1 以内）的 k 1 阶统计量。给出一个 O.n lg k/-time 算法来列出集合的第 k 个分位数。

9.3-9
描述一个 O.n/-time 算法，给定一个由 n 个不同数字组成的集合 S 和一个正整数 k = n，确定 S 中与 S 中位数最接近的 k 个数字。

9.3-10
假设 X Œ1 W n 和 Y Œ1 W n 是两个数组，每个数组包含 n 个已经排序的数字。给出一个 O.lg n/-time 算法来找出数组 X 和 Y 中所有 2n 个元素的中位数。假设所有 2n 个数字都是不同的。

问题

9-1Largest 我 numbers in sorted order
给定一组 n 个数字，您希望使用基于比较的算法按排序顺序找到最大的 i 个数字。描述实现以下每种方法的算法，并具有最佳渐近最坏情况运行时间，并根据 n 和 i 分析算法的运行时间。

a.对数字进行排序，并列出最大的 i 个数字。***b.*** 从数字中构建一个最大优先级队列，并调用 EXTRACT-MAX i 次。***c.*** 使用顺序统计算法来找到第 i 个最大数字，围绕该数字进行分区，并对第 i 个最大数字进行排序。

9-2Variant of randomized selection
孟德尔教授建议简化随机选取法，取消对 i 和 k 是否相等的检查。简化后的程序为更简单的随机选取法。

更简单的随机选择 .A; p; r; i /

1 如果 p == r 2 返回 AŒp // 1 i r p C 1 表示 i D 1 3 q D RANDOMIZED-PARTITION .A; p; r / 4 k D q p C 1 5 如果 i k 6 返回 SIMPLER-RANDOMIZED-SELECT .A; p; q; i / 7 否则返回 SIMPLER-RANDOMIZED-SELECT .A; q C 1; r; i k/

a.争论说在最坏的情况下，SIMPLER-RANDOMIZED-SELECT 永远不会终止。

b.证明SIMPLER-RANDOMIZED-SELECT的期望运行时间仍然为O.n/。

9-3Weighted median

考虑 n 个元素 x 1; x 2; : : : ; x n，其权重为正 w 1; w 2; : : : ; w n，使得 P $_{i=1}^{n}$ w i D 1。***weighted (lower) median*** 是满足以下条件的元素 x k

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

和

$$\sum_{x_i > x_k} w_i \le \frac{1}{2} .$$

例如，考虑以下元素 x i 和权重 w i：

i	1	2	3	4	5	6	7
x_i	3	8	2	5	4	1	6
w_i	0.12	0.35	0.025	0.08	0.15	0.075	0.2

对于这些元素，中位数为 x 5 D 4，但加权中位数为 x 7 D 6。要了解加权中位数为何为 x 7，请观察小于 x 7 的元素为 x 1、x 3、x 4、x 5 和 x 6，以及和 w 1 C w 3 C w 4 C w 5 C w 6 D 0:45，小于 1/2。此外，只有元素 x 2 大于 x 7，并且 w 2 D 0:35 不大于 1/2。

a.论证 x 1; x 2; : : : ; x n 的中位数是 x i 的加权中位数，其中权重为 w i D 1/n，其中 i D 1; 2; : : : ; n。***b.*** 说明如何使用排序在 O.n lg n/ 最坏情况下计算 n 个元素的加权中位数。***c.*** 说明如何使用线性时间中位数算法（如第 9.3 节中的 SELECT）在，.n/ 最坏情况下计算加权中位数。

post-office location problem 定义如下。输入为 n 个点 p 1; p 2; : : : ; p n 及其相关权重 w 1; w 2; : : : ; w n。解决方案是点 p（不一定是输入点之一），它最小化总和 P $_{i=1}^{n}$ w i d.p; p i /，其中 d.a; b/ 是点 a 和 b 之间的距离。

d.认为加权中值是一维邮局选址问题的最佳解，其中点只是实数，点 a 和 b 之间的距离为 d.a; b/D ja ‖ bj。

e.寻找二维邮局选址问题的最佳解，其中，点为 .x; y/ 坐标对，点 a D .x_1; y $_1$/ 和 b D .x_2; y $_2$/ 之间的距离是通过 d.a; b/ D jx $_1$ x $_2$j C jy $_1$ y $_2$j 给出的 ***Manhattan distance***。

9-4Small order statistics

我们用 S.n/ 表示 S ELECT 从 n 个数字中选择第 i 个统计量时最坏情况下的比较次数。虽然 S.n/ D ，.n/，但 ，符号隐藏的常数相当大。当 i 相对于 n 较小时，有一种算法使用 SELECT 作为子程序，但在最坏情况下进行的比较较少。

a.描述一种使用 U $_i$.n/ 比较来查找 n 个元素中第 i 个最小值的算法，其中

$$U_i(n) = \begin{cases} S(n) & \text{if } i \geq n/2\,, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + S(2i) & \text{otherwise}\,. \end{cases}$$

(*Hint:* 从 bn/2c 不相交的成对比较开始，并对包含每对中较小元素的集合进行递归。)

b.证明，如果 i < n/2，则 U $_i$.n/ D n C O.S.2i / lg.n/ i //。

c.证明如果 i 是小于 n/2 的常数，则 U $_i$.n/ D n C O.lg n/。

d.证明若 i D n/k 其中 k ≈ 2，则 U $_i$.n/ D n C O.S.2n/k/ lg k/。

9-5Alternative analysis of randomized selection

在这个问题中，你将使用指示随机变量来分析 RANDOMIZED-SELECT 过程，其方式类似于我们在第 7.4.2 节中对 RANDOMIZED-QUICKSORT 的分析。

与快速排序分析一样，我们假设所有元素都是不同的，并将输入数组 A 的元素重命名为 $z_1; z_2; : : : ; z_n$，其中 z_i 是第 i 个最小元素。因此调用 RANDOMIZED-SELECT .A; 1; n; i / 返回 z_i。

For $1 \leq j < k \leq n$, let

在算法执行过程中的某个时间，将 X ij k D I f z_j 与 z_k 进行比较，以找到 z_ig ：

a.给出 E ŒX $_{ijk}$ 的精确表达式。（*Hint:* 您的表达式可能具有不同的值，具体取决于 i 、 j 和 k 的值。）

b.令 X $_{i}$ 表示在找到 z_i 时数组 A 元素之间的比较总数。证明

$$\mathrm{E}[X_i] \le 2\left(\sum_{j=1}^{i}\sum_{k=i}^{n}\frac{1}{k-j+1} + \sum_{k=i+1}^{n}\frac{k-i-1}{k-i+1} + \sum_{j=1}^{i-2}\frac{i-j-1}{i-j+1}\right) .$$

c.证明 E ŒX $_{i}$ 4n。

d.得出结论，假设数组 A 的所有元素都是不同的，RANDOMIZED-SELECT 运行时间为 O.n/ 预期时间。

9-6Select with groups of 3

练习 9.3-1 要求你证明，如果将元素分成 7 个组，SELECT 算法仍能在线性时间内运行。此问题询问分成 3 个组的情况。

a.证明如果将元素分成大小为任何大于 3 的奇数常数的组，则 SELECT 将在线性时间内运行。***b.*** 证明如果将元素分成大小为 3 的组，则 SELECT 将在 O .n lg n/ 时间内运行。

由于 (b) 部分中的界限只是一个上限，我们不知道 3 组策略是否真的可以在 O.n/ 时间内运行。但通过在中间一组中位数上重复 3 组策略，我们可以选择一个保证 O.n/ 时间的枢轴。下一页的 SELECT3 算法确定 n > 1 个不同元素的输入数组中的第 i 个最小值。

c.用英语描述SELECT3算法的工作原理。在描述中包括一个或多个合适的图表。

d.说明 SELECT3 在最坏情况下运行时间为 O.n/ 分钟。

章节注释

最坏情况线性时间中值查找算法由 Blum、Floyd、Pratt、Rivest 和 Tarjan [62] 设计。快速随机化版本由 Hoare [218] 设计。Floyd 和 Rivest [147] 开发了一种改进的随机化版本，该版本围绕从小样本元素中递归选择的元素进行划分。

```
SELECT3 .A; p; r; i / 1 while .r  p C 1/ mod 9 ¤ 0 2 for j D p C 1 to r // 将最小值放入 AŒp  3 i
f AŒp  > AŒj  4 将 AŒp  与 AŒj  交换 5 // 如果我们想要 AŒp W r  的最小值,
那就完成了。 6 if i ==1 7 return AŒp  8 // 否则，我们想要 AŒp C 1 W r  的第 .i 个 1/st
元素。 9 p D p C 1 10 i D i  1 11 g D .r  p C 1/=3 // 3 元素组的数量 12 对于 j D p 到 p C g  1 //
遍历组 13 对 hAŒj ; AŒj C g; AŒj C 2gi 进行排序 14 // 所有组中值现在位于 AŒp W r 的
中间三分之一。 15 g 0 D g=3 // 3 元素子组的数量 16 对于 j D p C g 到 p C g C g 0  1 // 对子
组进行排序 17 对 hAŒj ; AŒj C g 0 ; AŒj C 2g 0  i 就位 18 // 现在所有子组中位数都位于
AŒp W r  的中间九分之一处。 19 // 以递归方式查找枢轴 x 作为子组中位数的中位数。 2
0 x D SELECT3 .A; p C 4g 0 ; p C 5g 0  1; dg 0 =2e/ 21 q D PARTITION-AROUND.A; p; r; x/ //
围绕枢轴进行分区 22 // 其余部分与 SELECT 的第 19324 行相同。 23 k D q  p C 1 24 if i ==
k 25 return AŒq  // 枢轴值为答案 26 elseif i < k 27 return SELECT3 .A; p; q 1; i / 28 否则返
回 SELECT3 .A; q C 1; r; i k/
```

目前仍不清楚究竟需要多少次比较才能确定中位数。Bent 和 John [48] 给出了中位数确定的下限，即 2n 次比较，而 Sch¨onhage、Paterson 和 Pippenger [397] 给出了中位数的上限，即 3n。Dor 和 Zwick 对这两个界限都进行了改进。他们的上限 [123] 略小于 2.95n，对于一个小的正常数 ，他们的下限 [124] 为 .2 C /n，因此比 Dor 等人 [122] 的相关工作略有改进。Paterson [354] 描述了其中一些结果以及其他相关工作。

问题 9-6 受到 Chen 和 Dumitrescu [84] 的论文的启发。

Part III Data Structures

介绍

集合对于计算机科学和数学一样重要。数学集合是不变的，而算法操纵的集合可以随时间增长、缩小或以其他方式改变。我们将此类集合称为 ***dynamic*** 。接下来的四章介绍了一些表示无限动态集合并在计算机上操纵它们的基本技术。

算法可能需要对集合执行几种类型的操作。例如，许多算法只需要能够向集合中插入元素、从集合中删除元素以及测试集合中的成员资格。我们将支持这些操作的动态集合称为 ***dictionary***。其他算法需要更复杂的操作。例如，第 6 章在堆数据结构上下文中介绍的最小优先级队列支持向集合中插入元素和从集合中提取最小元素的操作。实现动态集合的最佳方法取决于您需要支持的操作。

动态集的元素

在动态集的典型实现中，每个元素都由一个对象表示，只要给出指向该对象的指针，就可以检查和操作该对象的属性。某些类型的动态集假设对象的属性之一是标识 ***key***。如果所有键都不同，我们可以将动态集视为一组键值。对象可能包含 ***satellite data***，这些值在其他对象属性中传递，但在集合实现中未使用。它还可能具有由集合操作操纵的属性。这些属性可能包含数据或指向集合中其他对象的指针。

一些动态集合假设键是从一个完全有序的集合中提取的，例如实数，或通常字母顺序下的所有单词的集合

排序。例如，全排序允许我们定义集合的最小元素，或者说出集合中大于给定元素的下一个元素。

动态集上的操作

动态集合上的操作可分为两类：***queries***，仅返回有关集合的信息；***modifying operations***，更改集合。以下是典型操作的列表。任何特定应用程序通常只需要实现其中的几个。

搜索.S; k/
: 给定一个集合 S 和一个键值 k，查询返回一个指向 S 中某个元素的指针 x，使得 x: *key* D k，如果没有这样的元素属于 S，则返回 NIL。

我插入.S; x/
: 修改操作将 x 指向的元素添加到集合 S 中。我们通常假设集合实现所需的元素 x 中的任何属性都已初始化。

删除 .S; x/
: 修改操作，给定指向集合 S 中元素的指针 x，从 S 中删除 x。（请注意，此操作接受指向元素 x 的指针，而不是键值。）

MINIMUM.S / 和 MAXIMUM.S /
: 对全序集 S 进行查询，返回指向 S 中具有最小（对于 MINIMUM）或最大（对于 MAXIMUM）键的元素的指针。

继任者；x/
: 给定一个元素 x （其键来自完全有序集合 S ），查询返回指向 S 中下一个较大元素的指针，如果 x 是最大元素，则返回 NIL。

前身 .S; x/
: 给定一个元素 x （其键来自完全有序集合 S ），查询返回指向 S 中下一个较小元素的指针，如果 x 是最小元素，则返回 NIL。

在某些情况下，我们可以扩展查询 SUCCESSOR 和 PREDECESSOR，使它们适用于具有非唯一键的集合。对于具有 n 个键的集合，通常的假设是，调用 MINIMUM 后再调用 n 1 个 SUCCESSOR，即可按排序顺序枚举集合中的元素。

我们通常根据集合的大小来衡量执行集合操作所需的时间。例如，第 13 章描述了一种数据结构，它可以在 O.lg n/ 时间内支持对大小为 n 的集合执行上述任何操作。

当然，你总是可以选择用数组来实现动态集。这样做的优点是动态集操作的算法很简单。然而缺点是这些操作中的许多操作在最坏情况下的运行时间为 ′.n/′。如果数组未排序，则 I NSERT 和 DELETE 可能需要 ′.1/′ 时间，但其余操作需要 ′.n/′ 时间。如果数组保持排序顺序，则 MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 需要 ′.1/′ 时间；如果使用二分搜索实现，SEARCH 需要 0.lg n/′ 时间；但在最坏情况下 I NSERT 和 DELETE 需要 ′.n/′ 时间。本部分研究的数据结构改进了许多动态集操作的数组实现。

第三部分概述

第 103 至 13 章描述了几种可用于实现动态集的数据结构。稍后我们将使用这些数据结构中的许多结构来为各种问题构建有效的算法。我们已经在第 6 章中看到了另一个重要的数据结构——堆。

第 10 章介绍了使用简单数据结构（如数组、矩阵、堆栈、队列、链接列表和根树）的基本知识。如果您已经学习过入门编程课程，那么您应该熟悉其中的大部分内容。

第 11 章介绍了哈希表，这是一种广泛使用的数据结构，支持字典操作 IN SERT、DELETE 和 SEARCH。在最坏的情况下，哈希表需要 ½.n/ 时间来执行 SEARCH 操作，但哈希表操作的预期时间为 0.1/。我们依靠概率来分析哈希表操作，但即使没有概率，您也可以理解操作的工作原理。

第 12 章将介绍二叉搜索树，它支持上面列出的所有动态集操作。在最坏的情况下，对具有 n 个元素的树执行每个操作需要 1/20 ...

第 13 章介绍了红黑树，它是二叉搜索树的一种变体。与普通的二叉搜索树不同，红黑树保证性能良好：在最坏的情况下，操作需要 O.lg n/ 时间。红黑树是一种平衡搜索树。第五部分的第 18 章介绍了另一种平衡搜索树，称为 B 树。虽然红黑树的机制有些复杂，但您可以从本章中了解其大部分属性，而无需详细研究机制。尽管如此，您可能会发现浏览代码很有启发性。

10 Elementary Data Structures

在本章中，我们将通过使用指针的简单数据结构来研究动态集合的表示。虽然你可以使用指针构建许多复杂的数据结构，但我们只介绍最基本的数据结构：数组、矩阵、堆栈、队列、链接列表和有根树。

10.1 简单的基于数组的数据结构：数组、矩阵、堆栈、队列

10.1.1 数组

我们假设数组和大多数编程语言一样，在内存中以连续的字节序列存储。如果数组的第一个元素的索引为 s（例如，在以 1 为起始索引的数组中，s D 1），则该数组从内存地址 a 开始，每个数组元素占用 b 个字节，则第 i 个元素占用从 a C b.i s/ 到 a C b.i s C 1/ 1 的字节。由于本书中大多数数组的索引从 1 开始，少数数组从 0 开始，因此我们可以稍微简化这些公式。当 s D 1 时，第 i 个元素占用字节 a C b.i 1/ 到 a C bi 1，当 s D 0 时，第 i 个元素占用字节 a C bi 到 a C b.i C 1/ 1。假设计算机可以在相同的时间内访问所有内存位置（如 2.2 节中描述的 RAM 模型），则访问任何数组元素都需要常数时间，无论索引如何。

大多数编程语言都要求特定数组的每个元素具有相同的大小。如果给定数组的元素可能占用不同的字节数，则上述公式不适用，因为元素大小 b 不是常量。在这种情况下，数组元素通常是大小不同的对象，并且每个数组元素中实际出现的是指向该对象的指针。无论指针引用什么，指针占用的字节数通常相同，因此要访问数组中的对象，上述公式给出指向该对象的指针的地址，然后必须按照指针来访问对象本身。

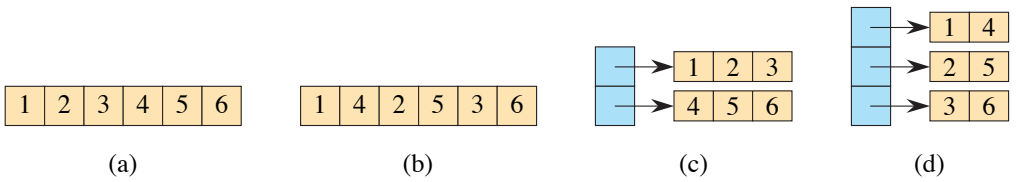


图 10.1 根据公式 (10.1) 存储 23 矩阵 M 的四种方法。 (a) 按行主序，在一个数组中。 (b) 按列主序，在一个数组中。 (c) 按行主序，每行一个数组（棕褐色），一个指向行数组的指针数组（蓝色）。 (d) 按列主序，每列一个数组（棕褐色），一个指向列数组的指针数组（蓝色）。

10.1.2 矩阵

我们通常用一个或多个一维数组来表示矩阵或二维数组。存储矩阵的两种最常见方式是行主序和列主序。让我们考虑一个 mn 矩阵4a，即有 m 行和 n 列的矩阵。在 ***row-major order*** 中，矩阵按行存储，在 ***column- major order*** 中，矩阵按列存储。例如，考虑 23 矩阵

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}. \tag{10.1}$$

按行主序存储两行 1 2 3 和 4 5 6，而按列主序存储三列 1 4、2 5 和 3 6。

图 10.1 的 (a) 和 (b) 部分显示了如何使用单个一维数组存储此矩阵。在 (a) 部分中它按行主顺序存储，在 (b) 部分中按列主顺序存储。如果行、列和单个数组都从 s 开始索引，则 M Œi; j 第 i 行和第 j 列的元素位于数组索引 s C .n.i s// C .j s/ （按行主顺序）和 s C .m.j s// C .i s/ （按列主顺序）。当 s D 1 时，单个数组索引为 n.i 1/ C j （按行主顺序）和 i C m.j 1/ （按列主顺序）。当 s D 0 时，单数组索引更简单：ni C j 采用行主序，i C mj 采用列主序。对于以 1 为原点索引的示例矩阵 M，元素 M Œ2; 1 采用行主序存储在单数组中的索引 3.2 1/ C1 D 4 处，采用列主序存储在索引 2 C 2.1 1/ D 2 处。

图 10.1 的 (c) 和 (d) 部分显示了存储示例矩阵的多数组策略。在 (c) 部分中，每行都存储在其自己的长度为 n 的数组中，如棕褐色所示。另一个具有 m 个元素的数组（以蓝色显示）指向 m 行数组。如果我们将蓝色数组称为 A，则 AŒi 指向存储 M 的第 i 行条目的数组，数组元素 AŒi Œj 存储矩阵元素 M Œi; j 。 (d) 部分显示了多数组表示的列主版本，有 n 个数组，每个数组

长度m，代表n列。矩阵元素MŒi;j 存储在数组元素AŒj Œi 中。

在现代机器上，单数组表示通常比多数组表示更高效。但多数组表示有时可能更灵活，例如，允许使用 “不规则数组，” 其中行主版本中的行可能具有不同的长度，或者对于列主版本对称，其中列可能具有不同的长度。

偶尔，也会使用其他方案来存储矩阵。在 ***block representa- tion*** 中，矩阵被分成块，每个块连续存储。例如，一个 4 4 矩阵被分成 2 2 个块，例如

$$\left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right)$$

可能以 h1; 2; 5; 6; 3; 4; 7; 8; 9; 10; 13; 14; 11; 12; 15; 16i 的顺序存储在单个数组中。

10.1.3 堆栈和队列

堆栈和队列是动态集合，其中 DELETE 操作从集合中删除的元素是预先指定的。在 ***stack*** 中，从集合中删除的元素是最近插入的元素：堆栈实现 ***last-in, first-out*** 或 ***LIFO*** 策略。同样，在 ***queue*** 中，删除的元素始终是集合中存在时间最长的元素：队列实现 ***first-in, first-out*** 或 ***FIFO*** 策略。有几种在计算机上实现堆栈和队列的有效方法。在这里，您将看到如何使用带有属性的数组来存储它们。

堆栈

堆栈上的 I NSERT 操作通常称为 PUSH，而 DELETE 操作（不接受元素参数）通常称为 P OP。这些名称暗指物理堆栈，例如自助餐厅使用的弹簧式盘子堆。盘子从堆栈中弹出的顺序与将它们推入堆栈的顺序相反，因为只有顶部的盘子是可访问的。

图 10.2 显示了如何使用数组 S Œ1 W n 实现最多包含 n 个元素的堆栈。堆栈具有属性 S: *top*（索引最近插入的元素）和 S: *size*（等于数组的大小 n）。堆栈由元素 S Œ1 W S: *top* 组成，其中 S Œ1 是堆栈底部的元素，S Œ S: *top* 是堆栈顶部的元素。

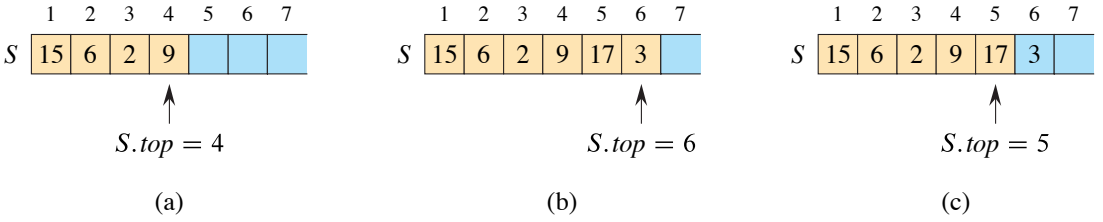


图 10.2 堆栈 S 的数组实现。堆栈元素仅出现在棕褐色位置。（a）堆栈 S 有 4 个元素。顶部元素为 9。（b）调用 PUSH.S; 17/ 和 PUSH.S; 3/ 之后的堆栈 S。（c）调用 POP.S/ 之后的堆栈 S 返回元素 3，这是最近推送的元素。虽然元素 3 仍出现在数组中，但它不再位于堆栈中。顶部是元素 17。

当 S: *top* D 0 时，堆栈不包含任何元素，为 ***empty***。我们可以使用查询操作 STAC K-EMPTY 来测试堆栈是否为空。尝试弹出空堆栈时，堆栈为 ***underflows***，这通常是错误。如果 S: *top* 超过 S: *size*，堆栈为 ***overflows***。

STAC K-EMPTY、PUSH 和 POP 过程仅用几行代码就实现了每个堆栈操作。图 10.2 显示了修改操作 PUSH 和 POP 的效果。这三个堆栈操作中的每一个都需要 O.1/ 时间。

STAC K-空.S /

1 如果 S: *top* ==0 2
返回 TRUE 3 否则返
回 FALSE

推.S; x/

1 if S: *top* ==S: *size* 2 error
“overüow” 3 else S: *top* D
S: *top* C 1 4 S ŒS: *top* D
x

POP.S /

1 如果 STAC K-EMPTY .S /
2 错误 “underüow” 3 否则 S
: *top* D S: *top* 1 4 返回 S Œ
S: *top* C 1

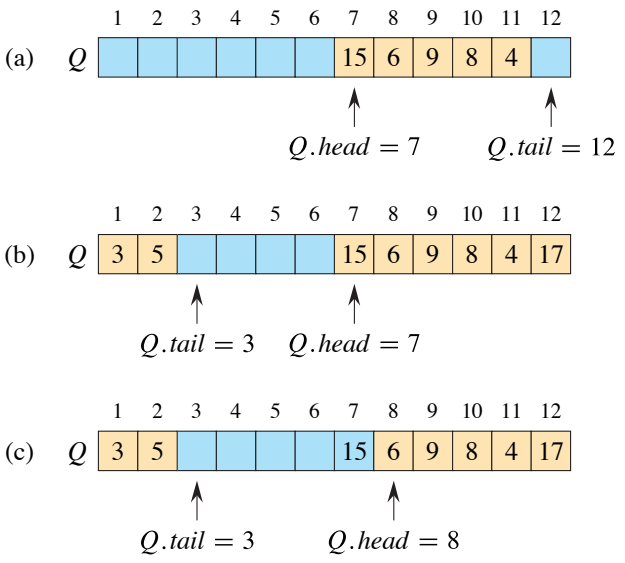


图 10.3 使用数组 QŒ1 W 12 实现的队列。队列元素仅出现在棕褐色位置。（a）队列有 5 个元素，位于位置 QŒ7 W 11 。（b）调用 ENQUEUE.Q；17/、ENQUEUE.Q；3/ 和 ENQ UEUE.Q；5/ 之后的队列配置。（c）调用 DEQUEUE.Q/ 之后的队列配置返回原来位于队列头部的键值 15。新头部的键为 6。

队列

我们将队列上的 I NSERT 操作称为 ENQUEUE，将 DELETE 操作称为 DEQ UEUE。与堆栈操作 POP 一样，DEQUEUE 不接受元素参数。队列的 FIFO 属性使其操作起来就像一排等待服务的顾客。队列具有 ***head*** 和 ***tail***。当一个元素入队时，它会占据队尾的位置，就像新到达的顾客占据队尾的位置一样。出队的元素始终是队头的元素，就像队头等待时间最长的顾客一样。

图 10.3 显示了使用数组 QŒ1 W n 实现最多有 n 1 个元素的队列的一种方法，其中属性 Q: *size* 等于数组的大小 n。该队列具有属性 Q: *head* ，用于索引或指向其头部。属性 Q: *tail* 索引新到达元素将插入队列的下一个位置。队列中的元素位于位置 Q: *head* ；Q: *head* C 1；: : : ；Q: *tail* 1，其中我们“环绕”，即位置 1 以循环顺序紧跟在位置 n 之后。当 Q: *head* D Q: *tail* 时，队列为空。最初，我们有 Q: *head* D Q: *tail* D 1。尝试从空队列中出队元素会导致队列下流。当 Q: *head* D Q: *tail* C1 或两者同时发生时

Q：*head* D 1 和 Q：*tail* D Q：*size*，队列已满，尝试将元素入队会导致队列溢出。

在过程 ENQUEUE 和 DEQUEUE 中，我们省略了对下流和上流的错误检查。（练习 10.1-5 要求您提供这些检查。）图 10.3 显示了 ENQUEUE 和 DEQUEUE 操作的效果。每个操作花费 O.1/ 时间。

```
入队.Q; x/
1 QŒQ：tail    D x 2 if Q：
tail ==Q：size 3 Q：tail  D
1 4 else Q：tail  D Q：tail C
1
出队.Q/
1 x D QŒQ: head     2 if Q:
head ==Q: size 3 Q: head  D 1 4
else Q: head  D Q: head C 1 5 re
turn x
```

练习

10.1-1

考虑一个按行主序排列的 m n 矩阵，其中 m 和 n 都是 2 的幂，行和列的索引都从 0 开始。我们可以用 lg m 位 hi $_{\lg m-1}$; i $_{\lg m-2}$; : : : ; i $_0$ i 以二进制表示行索引 i，用 lg n 位 hj $_{\lg n-1}$; j $_{\lg n-2}$; : : : ; j $_0$ i 以二进制表示列索引 j。假设此矩阵是 2 2 分块矩阵，其中每个块有 m/2 行和 n/2 列，并且它将由具有从 0 开始索引的单个数组表示。说明如何根据 i 和 j 的二进制表示形式，将 .lg m C lg n/ 位索引构造到单个数组的二进制表示形式。

10.1-2

以图 10.2 为例，说明在数组 SŒ1W6 中存储的初始为空的堆栈 S 上执行序列 PUSH.S；4/、PUSH.S；1/、PUSH.S；3/、POP.S/、PUSH.S；8/ 和 POP.S/ 中每个操作的结果。

10.1-3
解释如何在一个数组 AŒ1 W n 中实现两个堆栈，除非两个堆栈中的元素总数加起来为 n，否则两个堆栈都不会溢出。PUSH 和 POP 操作应在 O.1/ 时间内运行。

10.1-4
以图 10.3 为模型，说明对存储在数组 QŒ1 W 6 中的初始为空的队列 Q 执行序列 ENQUEUE.Q；4/、ENQUEUE.Q；1/、ENQUEUE.Q；3/、DEQUEUE.Q/、ENQUEUE.Q；8/ 和 DEQUEUE.Q/ 中每个操作的结果。

10.1-5
重写 ENQUEUE 和 DEQUEUE 来检测队列的欠流和溢出。

10.1-6
堆栈仅允许在一端插入和删除元素，队列允许在一端插入而在另一端删除元素，而 ***deque***（双端队列，发音为 “deck”）允许在两端插入和删除元素。编写四个 O.1/-time 过程，以在由数组实现的双端队列的两端插入元素和删除元素。

10.1-7
说明如何使用两个堆栈实现队列。分析队列操作的运行时间。

10.1-8
说明如何使用两个队列实现堆栈。分析堆栈操作的运行时间。

10.2 链表

linked list 是一种数据结构，其中的对象按线性顺序排列。然而，与数组不同，数组中的线性顺序由数组索引决定，而链表中的顺序由每个对象中的指针决定。由于链表的元素通常包含可以搜索的键，因此链表有时被称为 ***search lists***。链表为动态集提供了一种简单、灵活的表示，支持（但不一定有效）第 250 页列出的所有操作。

如图 10.4 所示，***doubly linked list*** L 中的每个元素都是一个对象，具有一个属性 *key* 和两个指针属性：*next* 和 *prev*。该对象可以

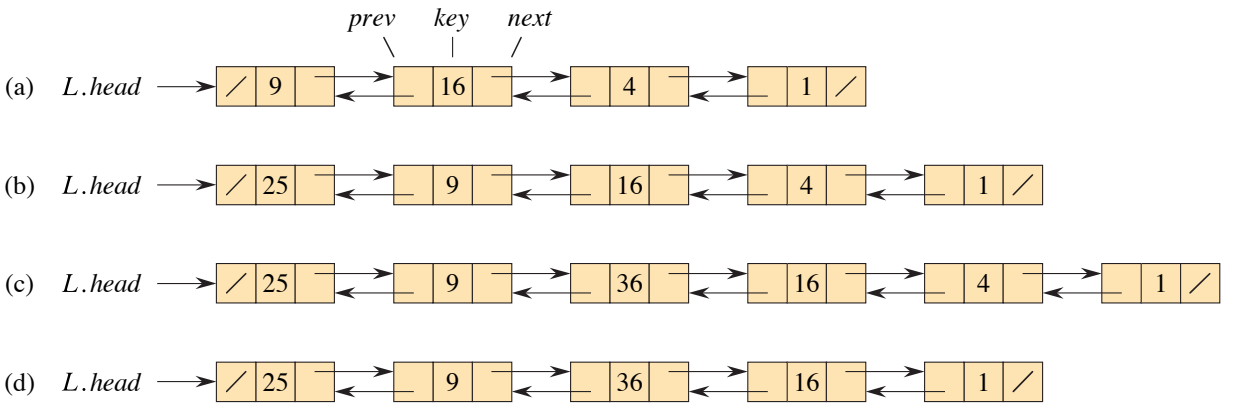


图 10.4 (a) 双向链表 L 表示动态集合 f1; 4; 9; 16g。列表中的每个元素都是一个对象，该对象具有键的属性以及指向下一个和上一个对象的指针（如箭头所示）。尾部的 *next* 属性和头部的 *prev* 属性为 NIL，用斜杠表示。属性 L: *head* 指向头部。 (b) 执行 LIST-PREPEND.L; x/, where x: *key* D 25 之后，链表有一个键为 25 的对象作为新的头部。这个新对象指向键为 9 的旧头部。 (c) 调用 LIST-I NSERT.x; y/, where x: *key* D 36 和 y 的结果指向键为 9 的对象。 (d) 后续调用 LIST-DELETE.L; 的结果x/，其中 x 指向键为 4 的对象。

还包含其他卫星数据。给定列表中的元素 x，x：*next* 指向链接列表中的后继，x：*prev* 指向其前任。如果 x：*prev* D NIL，则元素 x 没有前任，因此是列表的第一个元素或 ***head***。如果 x：*next* D NIL，则元素 x 没有后继，因此是列表的最后一个元素或 ***tail***。属性 L：*head* 指向列表的第一个元素。如果 L：*head* D NIL，则列表为空。

列表可能有多种形式。它可以是单链表或双链表，可以是排序的也可以是非排序的，可以是循环的也可以是非循环的。如果列表是 ***singly linked***，则每个元素都有一个 *next* 指针，但没有 *prev* 指针。如果列表是 ***sorted***，则列表的线性顺序对应于存储在列表元素中的键的线性顺序。最小元素是列表的头部，最大元素是列表的尾部。如果列表是 ***unsorted***，则元素可以按任何顺序出现。在 ***circular list*** 中，列表头部的 *prev* 指针指向尾部，列表尾部的 *next* 指针指向头部。您可以将循环列表视为元素环。在本节的其余部分中，我们假设我们正在使用的列表是未排序的和双链表。

搜索链接列表

过程 LIST-SEARCH .L; k/ 通过简单的线性搜索在列表 L 中找到第一个带有键 k 的元素，并返回指向该元素的指针。如果列表中没有出现带有键 k 的对象，则该过程返回 NIL。对于图 10.4(a) 中的链表，调用 LIST-SEARCH .L; 4 / 返回指向第三个元素的指针，调用 LIST-SEARCH .L; 7/ 返回 NIL。要搜索包含 n 个对象的列表，LIST-SEARCH 过程在最坏情况下需要 ，.n/ 时间，因为它可能必须搜索整个列表。

```
列表搜索 .L; k/
1 x D L: head  2 当 x ¤ NIL 且 x:
key ¤ k 3 x D x: next  4 返回 x
```

插入链接列表

给定一个元素 x，其 *key* 属性已经设置，LIST-PREPEND 过程将 x 添加到链接列表的前面，如图 10.4(b) 所示。（回想一下，我们的属性表示法可以级联，因此 L:*head*:*prev* 表示 L:*head* 指向的对象的 *prev* 属性。）对于包含 n 个元素的列表，LIST-PREPEND 的运行时间为 O.1/。

```
LIST-PREPEND .L; x/
1 x: next  D L: head  2 x:
prev  D NIL 3 if L: head ¤
NIL 4 L: head: prev  D x 5
L: head  D x
```

您可以在链接列表的任何位置插入。如图 10.4(c) 所示，如果您有一个指向列表中某个对象的指针 y，则对面页面 “ 上的 LIST-I NSERT 过程会将” 新元素 x 拼接到列表中，紧接着 y ，耗时 O.1/。由于 LIST-I NSERT 从不引用列表对象 L，因此不会将其作为参数提供。

列表插入 .x; y/

1 x: *next* D y: *next* 2 x:
prev D y 3 如果 y: *next* ¤
NIL 4 y: *next*: *prev* D x 5
y: *next* D x

从链接列表中删除

过程 LIST-DELETE 从链接列表 L 中删除元素 x。必须为它提供一个指向 x 的指针，然后它通过更新指针将 x 从列表中拼接出来。要删除具有给定键的元素，首先调用 LIST-SEARCH 来检索指向该元素的指针。图 10.4(d) 显示了如何从链接列表中删除元素。LIST-DELETE 的运行时间为 O.1/，但要删除具有给定键的元素，调用 LIST-SEARCH 会使最坏情况运行时间为 '.n/。

列表-删除 .L; x/

1 如果 x：*prev* ¤ NIL 2 x：
prev：*next* D x：*next* 3 否则
L：*head* D x：*next* 4 如果 x
：*next* ¤ NIL 5 x：*next*：*prev*
D x：*prev*

插入和删除在双向链表中比在数组中更快。如果要在数组中插入新的第一个元素或删除数组中的第一个元素，同时保持所有现有元素的相对顺序，则每个现有元素都需要移动一个位置。因此，在最坏的情况下，在数组中插入和删除需要 ，.n/ 时间，而在双向链表中则需要 ，.1/ 时间。（练习 10.2-1 要求您证明在最坏的情况下从单链表中删除元素需要 ，.n/ 时间。）但是，如果您想按线性顺序找到第 k 个元素，则在数组中无论 k 是多少都只需要 ，.1 / 时间，但在链表中，您必须遍历 k 个元素，需要 ，.k/ 时间。

哨兵

如果忽略列表头部和尾部的边界条件，LIST-DELETE 的代码会更简单：

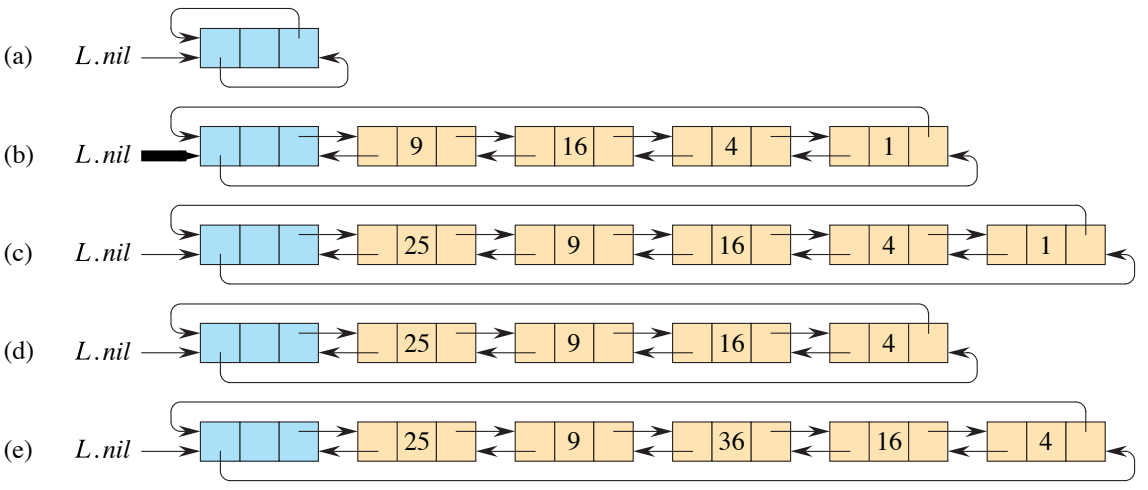


图 10.5 带有标记的循环双向链表。标记 L: *nil*（蓝色）出现在表头和表尾之间。属性 L: *head* 不再需要，因为表头是 L: *nil*: *next*。（a）空表。（b）图 10.4（a）中的链表，键 9 位于表头，键 1 位于表尾。（c）执行 LIST-I NSERT′.x; L: *nil*/（其中 x: *key* D 25）后的链表。新对象成为链表的表头。（d）删除键 1 的对象后的链表。新的表尾是键 4 的对象。（e）执行 LIST -I NSERT′.x; y/（其中 x: *key* D 36 且 y 指向键 9 的对象）后的链表。

列表-删除 0 .x/

1 x: *prev*: *next* D x: *next*
2 x: *next* : *prev* D x: *prev*

sentinel 是一个虚拟对象，可以让我们简化边界条件。在链表 L 中，哨兵是一个对象 L: *nil* ，它表示 NIL，但具有列表中其他对象的所有属性。对 NI L 的引用被对哨兵 L: *nil* 的引用所取代。如图 10.5 所示，此更改将常规双向链表变为 ***circular, doubly linked list with a sentinel***，其中哨兵 L: *nil* 位于头部和尾部之间。属性 L: *nil*: *next* 指向列表的头部，而 L: *nil*: *prev* 指向尾部。类似地，尾部的 *next* 属性和头部的 *prev* 属性都指向 L: *nil*。由于 L: *nil*: *next* 指向头部，因此属性 L: *head* 被完全消除，对它的引用被对 L: *nil*: *next* 的引用取代。图 10.5(a) 显示一个空列表仅由标记组成，并且 L: *nil*: *next* 和 L: *nil*: *prev* 都指向 L: *nil*。

要从列表中删除元素，只需使用前面的两行过程 LIST-DELETE′。正如 L IST-I NSERT 从不引用列表对象 L 一样，

LIST-DELETE 0 。你永远不应该删除标记 L: *nil* ，除非你要删除整个列表！

LIST-I NSERT′ 过程将元素 x 插入到对象 y 后面的列表中。不需要单独的前置过程：要插入到列表的头部，让 y 为 L: *nil*；要插入到列表的尾部，让 y 为 L: *nil*: *prev*。图 10.5 显示了 LIST-I NSERT′ 和 LIST-DELETE′ 对示例列表的影响。

列表插入 0 .x; y/

1 x：*next* D y：
next 2 x：*prev* D y
3 y：*next*：*prev* D x
4 y：*next* D x

使用标记搜索循环双向链表与不使用标记搜索具有相同的渐近运行时间，但可以减少常数因子。LIST-SEARCH 第 2 行的测试进行了两次比较：一次检查搜索是否已超出列表末尾，如果没有，则检查键是否位于当前元素 x 中。假设您 *know* 键位于列表中的某个位置。那么您无需检查搜索是否超出列表末尾，从而消除了 while 循环每次迭代中的一次比较。

标记提供了一个在开始搜索之前放置密钥的位置。搜索从列表 L 的头部 L: *nil*: *next* 开始，如果在列表中的某个位置找到密钥，则搜索停止。现在，搜索可以保证找到密钥，无论是在标记中还是在到达标记之前。如果在到达标记之前找到密钥，则它确实位于搜索停止的元素中。但是，如果搜索遍历列表中的所有元素并仅在标记中找到密钥，则密钥实际上不在列表中，搜索将返回 NIL。过程 LIST-SEARCH′ 体现了这一想法。（如果您的标记要求其 *key* 属性为 NIL，那么您可能希望在第 5 行之前分配 L: *nil*: *key* D NIL。）

列表搜索 0 .L; k/

1 L: *nil*: *key* D k // 将密钥存储在标记中以保证它在列表中 2 x D L: *nil*: *next* // 从列表头部开始 3 while x: *key* ¤ k 4 x D x: *next* 5 if x == L: *nil* // 在标记中发现 k 6 return NIL // k 实际上不在列表中 7 else return x // 在元素 x 中找到 k

哨兵通常可以简化代码，就像在搜索链接列表时一样，它们可能会将代码速度提高一个常数倍，但它们通常不会改善渐近运行时间。请谨慎使用它们。当有许多小列表时，它们的哨兵使用的额外存储空间可能会浪费大量内存。在本书中，我们仅在哨兵可以显著简化代码时才使用它们。

练习

10.2-1
解释为什么单链表上的动态设置操作 I NSERT 可以在 O.1/ 时间内实现，但 DELETE 的最坏情况时间却是 O.n/。

10.2-2
使用单链表实现堆栈。操作 P USH 和 POP 仍应花费 O.1/ 时间。您需要向列表添加任何属性吗？

10.2-3
使用单链表实现队列。ENQUEUE 和 DEQUEUE 操作仍需花费 O.1/ 时间。您需要向列表添加任何属性吗？

10.2-4
动态集合运算 UNION 以两个不相交集合 S_1 和 S_2 作为输入，返回由 S_1 和 S_2 的所有元素组成的集合 S D S_1 [S_2。集合 S_1 和 S_2 通常会被该运算破坏。说明如何使用合适的列表数据结构在 O.1/ 时间内支持 U NION。

10.2-5
给出一个 n 次非递归过程，用于反转一个包含 n 个元素的单链表。该过程应使用除列表本身所需存储空间之外的常量存储空间。

10.2-6
解释如何实现双向链表，每个项目仅使用一个指针值 x: *np* ，而不是通常的两个（*next* 和 *prev*）。假设所有指针值都可以解释为 k 位整数，并且 deûne x: *np* D x: *next* XOR x: *prev*，即 x: *next* 和 x: *prev* 的 k 位 “ 异或 ”。值 NIL 用 0 表示。请务必描述访问列表头部所需的信息。说明如何在此类列表上实现 SEARCH、INSERT 和 DELETE 操作。还说明如何在 O.1/ 时间内反转此类列表。

10.3 表示有根树

链接列表非常适合表示线性关系，但并非所有关系都是线性的。在本节中，我们将专门研究通过链接数据结构表示根树的问题。我们首先研究二叉树，然后介绍一种根树的方法，其中节点可以有任意数量的子节点。

我们用一个对象表示树的每个节点。与链表一样，我们假设每个节点包含一个 *key* 属性。其余感兴趣的属性是指向其他节点的指针，它们根据树的类型而有所不同。

二叉树

图 10.6 显示了如何使用属性 p、*left* 和 *right* 存储指向二叉树 T 中每个节点的父节点、左子节点和右子节点的指针。如果 x：*p* D NIL，则 x 为根。如果节点 x 没有左子节点，则 x：*left* D NIL，右子节点也是如此。整个树 T 的根由属性 T：*root* 指向。如果 T：*root* D NIL，则树为空。

有根且分支无限的树

将表示二叉树的方案扩展为任何树类（其中每个节点的子节点数最多为某个常数 k）很简单：将 *left* 和 *right* 属性替换为 $child_1$；$child_2$；：：：：；$child_k$。但是，当节点的子节点数无界时，此方案不再有效，因为我们不知道要提前分配多少个属性。此外，如果子节点数 k 受一个大常数限制，但大多数节点的子节点数较少，我们可能会浪费大量内存。

幸运的是，有一种巧妙的方案可以表示具有任意数量子节点的树。它的优点是对于任何 n 节点根树仅使用 O.n/ 空间。图 10.7 中出现了 ***left-child, right-sibling representation***。与前面一样，每个节点包含一个父指针 p，并且 T: *root* 指向树 T 的根。但是，每个节点 x 不是指向其每个子节点的指针，而是只有两个指针：

1. x: *left-child* 指向节点 x 的最左边的孩子，并且
2. x: *right-sibling* 指向 x 紧邻其右边的兄弟节点。

如果节点 x 没有子节点，则 x：*left-child* D NIL，并且如果节点 x 是其父节点的最右边子节点，则 x：*right-sibling* D NIL。

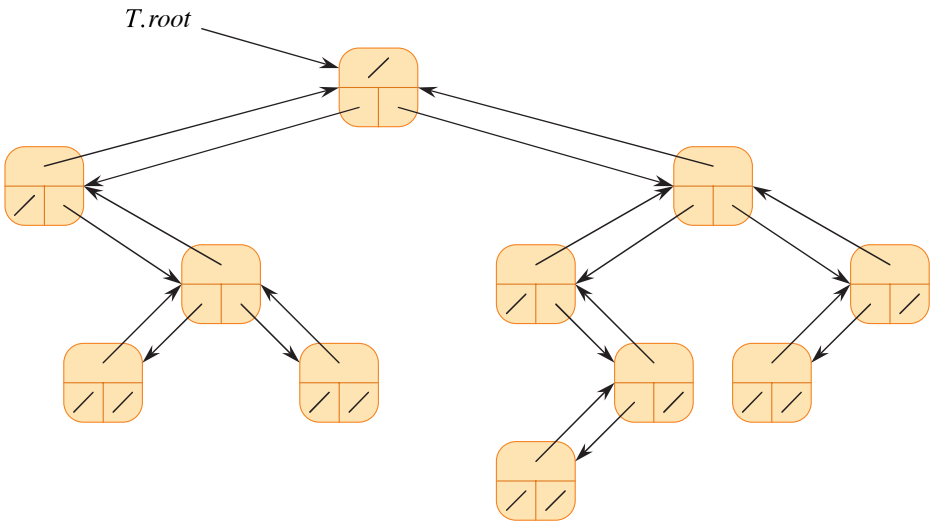


图 10.6 二叉树 T 的表示。每个节点 x 都具有属性 x：*p* (top)、x：*left* (lower left) 和 x：*right* (lower right)。*key* 属性未显示。

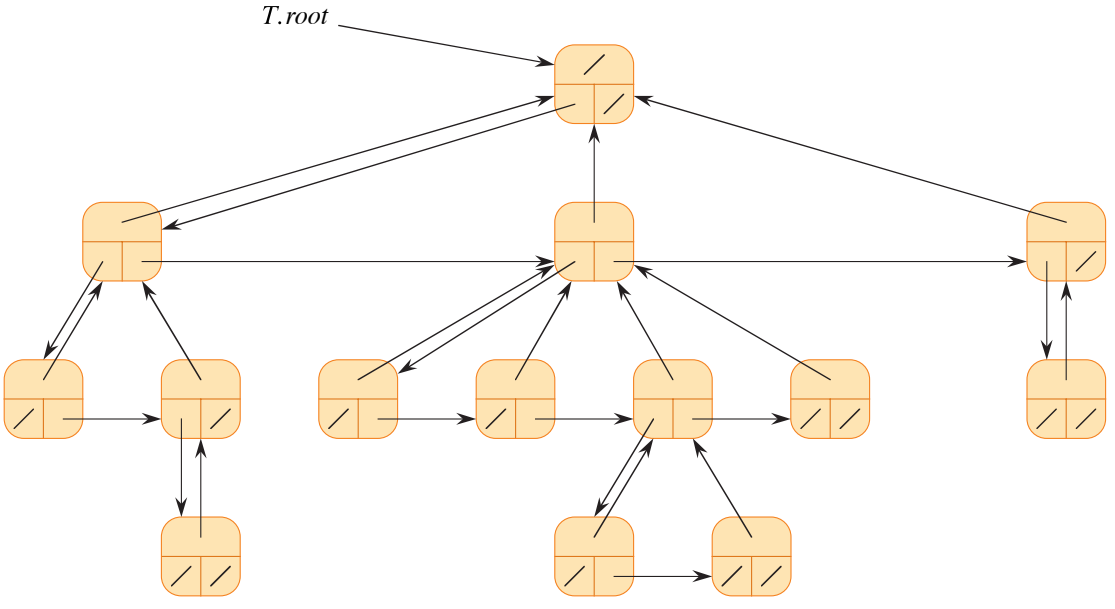


图 10.7 树 T 的左子节点、右兄弟节点表示。每个节点 x 都具有属性 x: *p* (top)、x: *left-child*（左下）和 x: *right-sibling* (lower right)。*key* 属性未显示。

其他树表示

我们有时用其他方式表示有根树。例如，在第 6 章中，我们用一个数组和一个属性来表示基于完全二叉树的堆，该属性给出堆中最后一个节点的索引。第 19 章中出现的树只向根方向遍历，因此只有父指针存在：没有指向子节点的指针。许多其他方案都是可能的。哪种方案最好取决于应用程序。

练习

10.3-1
绘制以索引 6 为根的二叉树，其表示属性如下：

index	*key*	*left*	*right*
1	17	8	9
2	14	NIL	NIL
3	12	NIL	NIL
4	20	10	NIL
5	33	2	NIL
6	15	1	4
7	28	NIL	NIL
8	22	NIL	NIL
9	13	3	7
10	25	NIL	5

10.3-2
编写一个 O.n/-time 递归程序，给定一个 n 节点二叉树，打印出树中每个节点的键。

10.3-3
编写一个 O.n/-time 非递归程序，给定一个 n 节点二叉树，打印出树中每个节点的键。使用堆栈作为辅助数据结构。

10.3-4
编写一个 O.n/-time 过程，打印出任意有 n 个节点的有根树的所有键，其中树使用左孩子、右兄弟表示法存储。

10.3-5
编写一个 O.n/-time 非递归程序，给定一个 n 节点二叉树，打印出每个节点的键。在外部使用不超过常数的额外空间

树本身，并且在过程中不要修改树，即使是暂时的。

? ***10.3-6***
任意有根树的左子节点、右兄弟节点表示法在每个节点中使用三个指针：*left-child*、*right-sibling* 和 *parent* 。从任何节点，都可以在常数时间内访问其父节点，并且可以在与子节点数量成线性关系的时间内访问其所有子节点。说明如何在每个节点 x 中仅使用两个指针和一个布尔值，以便可以在与 x 子节点数量成线性关系的时间内访问 x 的父节点或 x 的所有子节点。

问题

10-1Comparisons among lists
对于下表中的四种类型的列表，列出的每个动态集操作的渐近最坏情况运行时间是多少？

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH				
INSERT				
DELETE				
SUCCESSOR				
PREDECESSOR				
MINIMUM				
MAXIMUM				

10-2Mergeable heaps using linked lists
mergeable heap 支持以下操作：MAKE-HEAP（创建一个空的可合并堆）、INSERT、MINIMUM、EXTRACT-MIN 和 UNION。[1]

[1] Because we have defined a mergeable heap to support MINIMUM and EXTRACT-MIN, we can also refer to it as a ***mergeable min-heap***. Alternatively, if it supports MAXIMUM and EXTRACT-MAX, it is a ***mergeable max-heap***.

说明如何在以下每种情况下使用链表实现可合并堆。尽量使每个操作尽可能高效。根据所操作的动态集的大小分析每个操作的运行时间。

a.列表已排序。

b.列表未排序。

c.列表 s 是未排序的，要合并的动态集是 d 是联合的。

10-3Searching a sorted compact list

我们可以用两个数组 *key* 和 *next* 表示单链表。给定一个元素的索引 i，其值存储在 *key*Œi 中，其后继元素的索引由 *next* Œi 给出，其中 *next* Œi D NIL 表示最后一个元素。我们还需要列表中第一个元素的索引 *head* 。如果以这种方式存储的 n 元素列表仅存储在 *key* 和 *next* 数组的 1 到 n 位置，则它为 ***compact***。假设所有键都是不同的，并且紧凑列表也是有序的，即，对于所有 i D 1; 2; : : : ; n，*key*Œi < *key*Œ*next* Œi ，使得 *next* Œi ¤ NIL。在这些假设下，您将证明随机算法 C OMPAC T -LIST-SEARCH 在 O. p n/ 预期时间内在列表中搜索键 k。

```
紧凑列表搜索 .key; next ; head ; n; k/ 1 i D head 2 当
i ¤ NIL 且 keyŒi < k 3 j D RANDOM.1; n/ 4 如果
keyŒi < keyŒj 且 keyŒj k 5 i D j 6 如果
keyŒi ==k 7 返回 i 8 i D next Œi 9 如果 i ==NIL
或 keyŒi > k 10 返回 NIL 11 否则返回 i
```

如果忽略程序的第 337 行，您会发现这是一个用于搜索排序链表的普通算法，其中索引 i 依次指向列表的每个位置。一旦索引 i “ 超出列表末尾，或者 *key*Œi ≠ k，搜索就会终止。在后一种情况下，如果 *key*Œi ≠ D k，则程序已找到值为 k 的键。但是，如果 *key*Œi ≠ > k，则搜索将永远不会找到值为 k 的键，因此终止搜索是正确的操作。

337 行尝试跳转到随机选择的位置 j 。如果 *key*Œj 大于 *key*Œi 且不大于 k，则这种跳过会有所帮助。在这种情况下，j 标记列表中 i 在普通列表搜索期间将到达的位置。由于列表是紧凑的，我们知道在 1 和 n 之间选择任何 j 都会索引列表中的某个元素。

您无需直接分析 COMPAC T -LIST-SEARCH 的性能，而是分析一个相关算法，COMPAC T -LIST-SEARCH ′，该算法执行两个单独的循环。此算法采用一个附加参数 t，它指定了第一个循环的迭代次数的上限。

紧凑列表搜索 0 .*key*; *next* ; *head* ; n; k; t /

1 i D *head* 2 for q D 1 to t 3 j D RANDO M.1; n/ 4 如果 *key*Œi < *key*Œj 且 *key*Œj k 5 i D j 6 如果 *key*Œi == k 7 返回 i 8 当 i ¤ NIL 且 *key*Œi < k 9 i D *next* Œi 10 如果 i == NIL 或 *key*Œi > k 11 返回 NIL 12 否则返回 i

为了比较两种算法的执行情况，假设 RANDOM.1；n/ 的调用序列为两种算法产生相同的整数序列。

a.论证：对于任何 t 值，COMPAC T -LIST-SEARCH .*key*; *next* ; *head*; n; k/ 和 COMPAC T -LIST-SEARCH ′.*key*; *next* ; *head*; n; k; t / 均返回相同结果，并且 COMPAC T - LIST-SEARCH 中第 238 行的 while 循环的迭代次数最多为 COMPAC T -LIST-SEARCH ′ 中 for 和 while 循环的总迭代次数。

在调用 COMPAC T -LIST-SEARCH ′.*key*; *next* ; *head*; n; k; t / 中，让 X_t 为描述在 237 行的 for 循环进行 t 次迭代之后，链表（即通过 *next* 指针链）中从位置 i 到所需键 k 的距离的随机变量。

b.论证 COMPAC T -LIST-SEARCH ′.*key*; *next* ; *head* ; n; k; t / 的预期运行时间为 O.t C E ŒX $_t$ /。

c.证明 E ŒX $_t$ D P $_{r=1}^{n}$ .1r/n/ ^{t} 。（*Hint:* 使用第 1193 页的公式 (C.28)。）

d.证明 P $\sum_{r=0}^{n-1}$ r ^{t} n $^{t+1}$ /.t C1/。（*Hint:* 使用第 1150 页的不等式 (A.18)。）

e.证明 E ŒX $_{t}$ n/.t C 1/.

f.证明 COMPAC T -LIST-SEARCH ′ .*key*; *next* ; *head* ; n; k; t / 的预期运行时间为 O.t C n/t /。

g.得出结论：COMPAC T-LIST-SEARCH 在预期时间內运行 O. p n/。

h.为什么我们假设 C OMPAC T-LIST-SEARCH 中的所有键都是不同的？当列表包含重复的键值时，随机跳过不一定能渐近地提供帮助。

章节注释

Aho、Hopcroft 和 Ullman [6] 以及 Knuth [259] 是基本数据结构的优秀参考书。许多其他教科书既涵盖基本数据结构，也涵盖其在特定编程语言中的实现。这类教科书的例子包括 Goodrich 和 Tamassia [196]、Main [311]、Shaffer [406] 和 Weiss [452, 453, 454]。Gonnet 和 Baeza-Yates [193] 的书提供了许多数据结构操作性能的实验数据。

堆栈和队列作为计算机科学中的数据结构，其起源尚不清楚，因为在数字计算机出现之前，数学和纸质商业实践中就已经存在相应的概念。Knuth [259] 引用了 A. M. Turing 在 1947 年开发用于子程序链接的堆栈。

基于指针的数据结构似乎也是民间发明。根据 Knuth 的说法，指针显然用于早期带有磁鼓存储器的计算机。G. M. Hopper 于 1951 年开发的 A-1 语言将代数公式表示为二叉树。Knuth 认为，1956 年由 A. Newell、J. C. Shaw 和 H. A. Simon 开发的 IPL-II 语言认识到了指针的重要性并推动了指针的使用。他们于 1957 年开发的 IPL-III 语言包括显式堆栈操作。

11 哈希表

许多应用程序需要仅支持字典操作 INSERT、SEARCH 和 DELETE 的动态集。例如，翻译编程语言的编译器维护一个符号表，其中元素的键是与语言中的标识符相对应的任意字符串。哈希表是实现字典的有效数据结构。尽管在实践中，在哈希表中搜索元素所花的时间在最坏情况下与在链接列表中搜索元素所花的时间一样长，但哈希的性能非常好。在合理的假设下，在哈希表中搜索元素的平均时间为 O.1/。事实上，Python 的内置字典就是用哈希表实现的。

哈希表概括了普通数组的简单概念。直接寻址普通数组可以利用任何数组元素的 O.1/ 访问时间。第 11.1 节更详细地讨论了直接寻址。要使用直接寻址，您必须能够分配一个包含每个可能键的位置的数组。

当实际存储的键的数量相对于可能的键的总数较小时，哈希表成为直接寻址数组的有效替代方法，因为哈希表通常使用大小与实际存储的键的数量成比例的数组。我们并不直接将键用作数组索引，而是从键中获得数组索引*compute*。11.2 节介绍了主要思想，重点介绍了“ 链接 ”以处理“ 冲突 ”，其中多个键映射到同一个数组索引。11.3 节介绍如何使用哈希函数根据键计算数组索引。我们介绍并分析了基本主题的几种变体。11.4 节介绍“ 开放寻址 ”，这是另一种处理冲突的方法。总之，哈希是一种非常有效和实用的技术：基本字典操作平均只需要 O.1/ 时间。第 11.5 节讨论了现代计算机系统的分层内存系统，并说明了如何设计在这种系统中运行良好的哈希表。

11.1 直接地址表

直接寻址是一种简单的技术，当密钥域 U 相当小时，它非常有效。假设某个应用程序需要一个动态集合，其中每个元素都有一个不同的密钥，该密钥取自域 U D f0; 1; : : : ; m 1g，其中 m 不是太大。

为表示动态集合，可以使用数组或 ***direct-address table***，记为 T Œ0 W m 1 ，其中每个位置或 ***slot*** 对应于域 U 中的一个键。图 11.1 说明了这种方法。位置 k 指向集合中键为 k 的元素。如果集合中不包含键为 k 的元素，则 T Œ k 下一页的字典操作 DIRECT-ADDRESS-SEARCH、DIRECT-ADDRESS-INSERT 和 DIRECT-ADDRESS-DELETE 实现起来很简单。每个操作仅需 O.1/ 时间。

对于某些应用程序，直接地址表本身可以保存动态集中的元素。也就是说，不是将元素的键和卫星数据存储在直接地址表外部的对象中，而是将表中某个槽的指针指向该对象，而是将对象直接存储在槽中以节省空间。要指示空槽，请使用特殊键。再说一遍，为什么要存储对象的键呢？对象 *is* 的索引就是它的键！当然，那么您需要某种方法来判断槽是否为空。

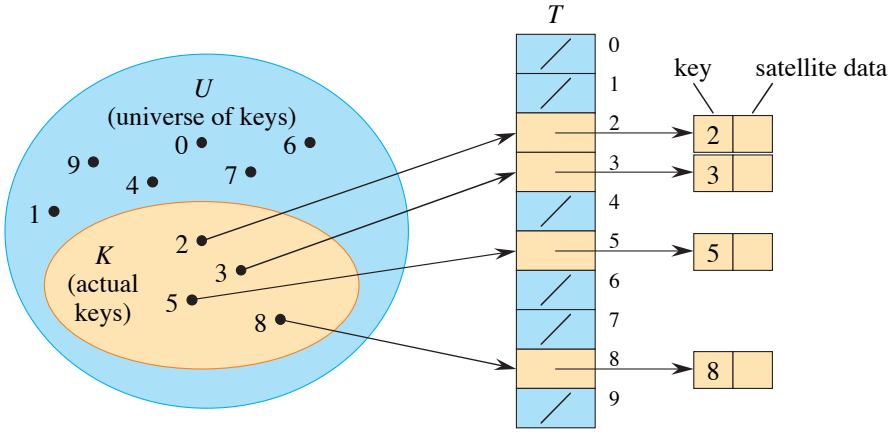


图 11.1 如何通过直接地址表 T 实现动态集合。域 U D f0; 1; : : : ; 9g 中的每个键都对应于表中的一个索引。实际键的集合 K D f2; 3; 5; 8g 决定了表中包含指向元素的指针的槽。其他槽（蓝色）包含 NIL。

直接地址搜索.T; k/

1 返回 T Œk

直接地址插入.T; x/

1 $T[x.key] = x$

直接地址删除.T; x/

1 T Œx: *key* D 无

练习

11.1-1

动态集合 S 由长度为 m 的直接地址表 T 表示。描述一个查找 S 中最大元素的过程。你的程序的最坏情况性能是多少？

11.1-2

bit vector 只是一个位数组（每个位为 0 或 1）。长度为 m 的位向量比 m 个指针的数组占用的空间少得多。描述如何使用位向量表示从集合 f0; 1; : : : ; m 1g 中提取的动态不同元素集，并且没有卫星数据。字典操作应在 O.1/ 时间内运行。

11.1-3

建议如何实现一个直接地址表，其中存储元素的键不需要不同，并且元素可以具有卫星数据。所有三个字典操作（INSERT、DELETE 和 SEARCH）都应该在 O.1/ 时间内运行。（不要忘记 DELETE 将指向要删除的对象的指针作为参数，而不是键。）

11.1-4

假设您想通过在 *huge* 数组上使用直接寻址来实现字典。也就是说，如果数组大小为 m，并且字典在任何时候最多包含 n 个元素，则 m n。开始时，数组条目可能包含垃圾，并且由于其大小，初始化整个数组是不切实际的。描述在巨型数组上实现直接地址字典的方案。每个存储的对象应使用 O.1/ 空间；操作 SEARCH、INSERT 和 DELETE 应分别花费 O.1/ 时间；初始化数据结构应花费 O.1/ 时间。（*Hint:* 使用一个额外的数组，有点像堆栈，其大小是字典中实际存储的键的数量，以帮助确定巨型数组中的给定条目是否有效。）

11.2 哈希表

直接寻址的缺点显而易见：如果域 U 很大或无限大，那么在典型计算机上的可用内存下，存储大小为 jU j 的表 T 可能不切实际，甚至不可能。此外，密钥集 K *actually stored* 相对于 U 可能太小，以至于分配给 T 的大部分空间将被浪费。

当字典中存储的键集合 K 比所有可能键的集合 U 小得多时，哈希表所需的存储空间比直接寻址表少得多。具体来说，存储需求减少到“ .jKj/ ”，同时保持了在哈希表中搜索元素仍然只需要 O.1/ 时间的优点。问题在于，这个界限适用于 *average-case time*、[1]，而对于直接寻址，它适用于 *worst-case time*。

使用直接寻址时，密钥为 k 的元素存储在槽 k 中，但使用哈希函数时，我们使用 ***hash function*** h 根据密钥 k 计算槽号，这样元素就进入槽 h.k/。哈希函数 h 将密钥域 U 映射到 ***hash table*** T Œ0 W m 1 的槽中：

$$h : U \to \{0, 1, \ldots, m-1\} ,$$

其中哈希表的大小 m 通常远小于 jU j。我们称具有密钥 k 的元素 ***hashes*** 到槽 h.k/，并且我们还称 h.k/ 是密钥 k 的 ***hash value***。图 11.2 说明了基本思想。哈希函数减少了数组索引的范围，从而减少了数组的大小。数组的大小可以不是 jU j，而是 m。一个简单但不是特别好的哈希函数示例是 h.k/ D k mod m。

有一个问题，即两个密钥可能散列到同一个槽。我们将这种情况称为 ***collision***。幸运的是，有有效的技术可以解决碰撞造成的冲突。

当然，理想的解决方案是完全避免冲突。我们可以通过选择合适的哈希函数 h 来实现这一目标。一种想法是使 h 看起来是随机的，从而避免冲突或至少将冲突次数降至最低。哈希一词本身就让人联想到随机混合和切碎，体现了这种方法的精神。（当然，哈希函数 h 必须是确定性的，即给定的输入 k 必须始终产生相同的输出 h.k/。）但是，由于 jU j > m，必须至少有两个具有相同哈希值的密钥，

[1] The definition of “average-case” requires care—are we assuming an input distribution over the keys, or are we randomizing the choice of hash function itself? We’ll consider both approaches, but with an emphasis on the use of a randomly chosen hash function.

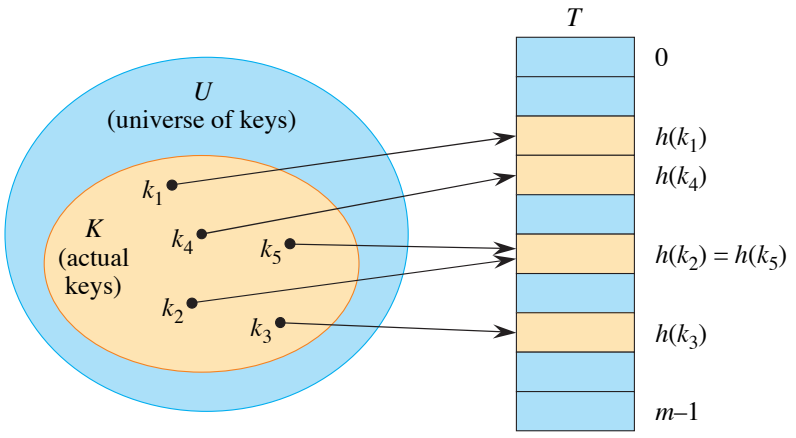


图 11.2 使用哈希函数 h 将键映射到哈希表槽位。由于键 k_2 和 k_5 映射到同一个槽位，因此它们发生冲突。

完全避免冲突是不可能的。因此，尽管设计良好、看似随机的哈希函数可以减少冲突的数量，但我们仍然需要一种方法来解决确实发生的冲突。

本节的剩余部分首先给出了独立均匀散列法的定义，该定义捕捉了散列函数随机性最简单的概念。然后介绍并分析了最简单的冲突解决技术，即链式连接。第 11.4 节介绍了一种解决冲突的替代方法，即开放寻址法。

独立统一哈希

对于域 U 中的每个可能输入 k，“ 理想” 哈希函数 h 都会有一个输出 h.k/，该输出是从范围 f0；1；：：：；m 1g 中随机且独立地均匀选择的元素。一旦随机选择了值 h.k/，则使用相同输入 k 对 h 的每次后续调用都会产生相同的输出 h.k/。

我们将这种理想的哈希函数称为 ***independent uniform hash function***。这种函数也经常被称为 ***random oracle*** [43]。当哈希表使用独立的统一哈希函数实现时，我们说使用的是 ***independent uniform hashing***。

独立均匀散列是一种理想的理论抽象，但它在实践中无法合理实现。尽管如此，我们将在独立均匀散列假设下分析散列的效率，然后介绍实现此理想的实用近似值的方法。

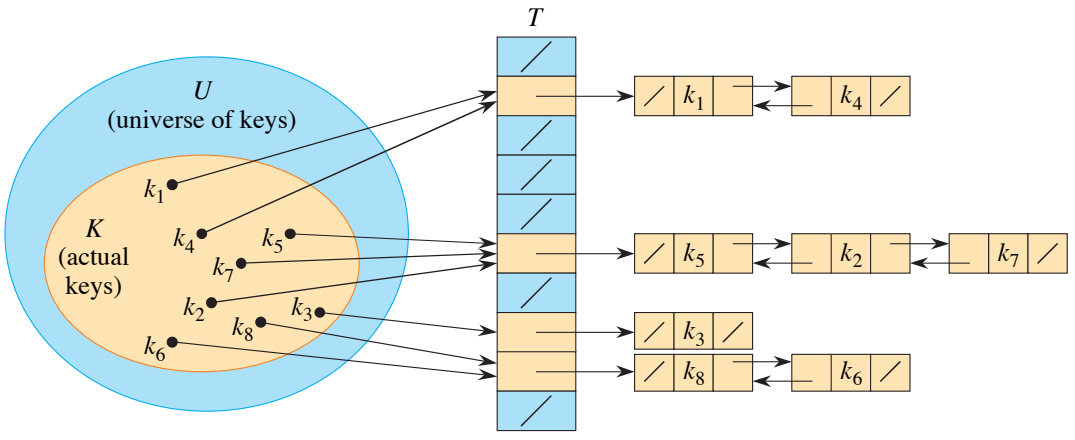


图 11.3 通过链接解决冲突。每个非空哈希表槽 T Œj 指向哈希值为 j 的所有键的链接列表。例如，h.k 1 / D h.k 4 / 和 h.k 5 / D h.k 2 / D h.k 7 /。列表可以是单链接的，也可以是双链接的。我们将其显示为双链接的，因为当删除过程知道要删除哪个列表元素（而不仅仅是哪个键）时，删除速度可能会更快。

通过链接解决碰撞

从高层次来看，你可以将哈希链式算法看作是分而治之的非递归形式：n 个元素的输入集被随机分成 m 个子集，每个子集的大小约为 n/m。哈希函数确定元素属于哪个子集。每个子集都作为列表独立管理。

图 11.3 展示了 ***chaining*** 背后的思想：每个非空槽都指向一个链表，所有哈希值指向同一个槽的元素都进入该槽的链表。槽 j 包含一个指向所有存储的哈希值为 j 的元素的列表头部的指针。如果没有这样的元素，则槽 j 包含 NIL。

当通过链接解决冲突时，字典操作很容易实现。它们出现在下一页并使用第 10.2 节中的链接列表过程。插入的最坏情况运行时间为 O.1/。插入过程很快，部分原因是它假设插入的元素 x 尚未存在于表中。为了强化这一假设，你可以在插入之前搜索（额外花费）键为 x: *key* 的元素。对于搜索，最坏情况运行时间与列表的长度成正比。（我们将在下面更仔细地分析此操作。）如果列表是双向链接的，则删除需要 O.1/ 时间，如图 11.3 所示。（由于 CHAINED-HASH-DELETE 将元素 x 而不是其键 k 作为输入，因此不需要搜索。如果哈希表支持删除，则其链接列表应该是双向链接的，以便快速删除项目。如果列表仅是单链接的，那么根据练习 10.2-1，删除

```
CHAINED-HASH-I NSERT .T; x/ 1 LIST-PRE
PEND .T Œh.x: key/   ; x/ CHAINED-HASH-
SEARCH .T; k/ 1 返回 LIST-SEARCH .T Œh.
k/   ; k/ CHAINED-HASH-DELETE .T; x/ 1 LI
ST-DELETE .T Œh.x: key/   ; x/
```

所花的时间与列表的长度成正比。对于单链表，删除和搜索的渐近运行时间相同。）

哈希链分析

链式哈希算法的性能如何？具体来说，搜索给定键的元素需要多长时间？

给定一个哈希表 T，该表有 m 个槽，存储 n 个元素，我们将 T 的 ***load factor*** α定义为 n/m，即链中存储元素的平均数量。我们的分析将以 α为依据，它可以小于、等于或大于 1。

链式哈希算法的最坏情况是糟糕的：所有 n 个键都哈希到同一个槽，创建一个长度为 n 的列表。因此，最坏情况下的搜索时间是 1.n/ 加上计算哈希函数的时间 4，并不比对所有元素使用一个链表好。我们显然不会使用哈希表，因为它们的最坏情况性能不佳。

散列的平均性能取决于散列函数 h 在平均情况下如何将要存储的键集分配到 m 个槽中（这意味着相对于要散列的键的分布以及散列函数的选择，如果此选择是随机的）。第 11.3 节讨论了这些问题，但现在我们假设任何给定元素都有同等的可能性散列到 m 个槽中的任何一个。也就是说，散列函数为 ***uniform***。我们进一步假设给定元素散列到的位置是其他任何元素散列到的位置的 *independent* 。换句话说，我们假设使用的是 ***independent uniform hashing***。

因为不同密钥的哈希值被认为是独立的，所以独立均匀哈希值为 ***universal***：任何两个不同密钥 k_1 和 k_2 发生碰撞的概率最多为 1/m。通用性在我们的分析中很重要，在通用哈希函数系列的规范中也很重要，我们将在 11.3.2 节中看到这一点。

对于 j D 0; 1; : : : ; m 1，用 n_j 表示列表 T Œj 的长度，以便

$$n = n_0 + n_1 + \cdots + n_{m-1} , \tag{11.1}$$

并且 n_j 的期望值为 E Œn_j D ¤D n/m。

我们假设 O.1/ 时间足以计算哈希值 h.k/，因此搜索具有密钥 k 的元素所需的时间与列表 T Œh.k/ 的长度 $n_{h(k)}$ 线性相关。除了计算哈希函数和访问槽 h.k/ 所需的 O.1/ 时间之外，我们将考虑搜索算法检查的元素的预期数量，即算法检查列表 T Œh.k/ 中是否有任何元素具有等于 k 的密钥的元素数量。我们考虑两种情况。在第一种情况下，搜索不成功：表中没有元素具有密钥 k。在第二种情况下，搜索成功找到了具有密钥 k 的元素。

Theorem11.1
在通过链接解决冲突的哈希表中，在独立均匀哈希的假设下，一次不成功的搜索平均需要 ，.1 C ¤/次。

Proof 在独立均匀散列的假设下，任何尚未存储在表中的密钥 k 都有同等可能散列到 m 个位置中的任何一个。搜索密钥 k 失败的预期时间是搜索到列表 T Œh.k/ 末尾的预期时间，该列表的预期长度为 E Œ$n_{h(k)}$ D ¤ 因此，在一次不成功的搜索中检查的元素的预期数量是 ¤，所需的总时间（包括计算 h.k/ 的时间）是 ，.1 C ¤/。 ■

成功搜索的情况略有不同。不成功的搜索同样有可能进入哈希表的任何位置。但是，成功的搜索不能进入空位置，因为它是针对链表中存在的元素。我们假设搜索的元素同样有可能是表中的任何一个元素，因此列表越长，搜索其中某个元素的可能性就越大。即便如此，预期搜索时间仍然是 ，.1 C ¤/。

Theorem11.2
在通过链接解决冲突的哈希表中，在独立均匀哈希的假设下，成功搜索平均需要 ，.1 C ¤/次。

Proof 我们假设被搜索的元素是表中存储的 n 个元素中的任何一个，这种可能性是相等的。在成功搜索元素 x 时检查的元素数量比 x 的列表中出现在 x 之前的元素数量多 1。由于新元素被放置在列表的前面，

列表中 x 之前的元素均在插入 x 之后插入。令 x_i 表示插入表中的第 i 个元素，其中 $i = 1, 2, \ldots, n$，令 $k_i = x_i.key$。

我们的分析广泛使用指示随机变量。对于表中的每个槽 q 以及每对不同的密钥 k_i 和 k_j，我们定义指示随机变量

$$X_{ijq} = \mathrm{I}\{\text{the search is for } x_i, h(k_i) = q, \text{and } h(k_j) = q\} .$$

也就是说，当键 k_i 和 k_j 在槽 q 处发生冲突并且搜索元素 x_i 时，$X_{ijq} = 1$。由于 $\Pr\{\text{搜索针对 } x_i\} = 1/n$、$\Pr\{h(k_i) = q\} = 1/m$、$\Pr\{h(k_j) = q\} = 1/m$，并且这些事件都是独立的，因此我们有 $\Pr\{X_{ijq} = 1\} = 1/nm^2$。第 130 页的引理 5.1 给出 $\mathrm{E}[X_{ijq}] = 1/nm^2$。

接下来，我们为每个元素 x_j 定义指示随机变量

$$\begin{aligned} Y_j &= \mathrm{I}\{x_j \text{ appears in a list prior to the element being searched for}\} \\ &= \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq} , \end{aligned}$$

因为 X_{ijq} 中最多有一个等于 1，即当所搜索的元素 x_i 与槽 q) 指向的 x_j (属于同一列表，并且 $i < j$ （因此 x_i 出现在列表中的 x_j 之后）。

我们的最终随机变量是 Z，它计算在被搜索元素之前列表中出现的元素数：

$$Z = \sum_{j=1}^{n} Y_j .$$

因为我们必须计算被搜索的元素以及列表中所有在它之前的元素，所以我们想计算 $\mathrm{E}[Z + 1]$。利用期望的线性（第 1192 页的方程 (C.24)），我们有

$$\begin{aligned} \mathrm{E}[Z + 1] &= \mathrm{E}\left[1 + \sum_{j=1}^{n} Y_j\right] \\ &= 1 + \mathrm{E}\left[\sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}\right] \\ &= 1 + \mathrm{E}\left[\sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} X_{ijq}\right] \\ &= 1 + \sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} \mathrm{E}[X_{ijq}] \qquad \text{(by linearity of expectation)} \end{aligned}$$

$$
\begin{aligned}
&= 1 + \sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} \frac{1}{nm^2} \\
&= 1 + m \cdot \frac{n(n-1)}{2} \cdot \frac{1}{nm^2} \qquad \text{(by equation (A.2) on page 1141)} \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{n}{2m} - \frac{1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} .
\end{aligned}
$$

因此，成功搜索所需的总时间（包括计算哈希函数的时间）为 $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$。 ■

这个分析意味着什么？如果表中元素的数量最多与哈希表槽的数量成正比，则我们有 $n = O(m)$，因此，$\alpha = n/m = O(m)/m = O(1)$。因此，搜索平均需要常数时间。由于当列表是双向链接时（假设要删除的列表元素是已知的，而不仅仅是它的键），插入需要 $O(1)$ 最坏情况时间，删除需要 $O(1)$ 最坏情况时间，因此我们可以平均在 $O(1)$ 时间内支持所有字典操作。

上述两个定理中的分析仅取决于独立均匀散列的两个基本属性：均匀性（每个密钥都有同等的可能性散列到 m 个槽中的任何一个）和独立性（因此任何两个不同的密钥以 $1/m$ 的概率发生冲突）。

练习

11.2-1

使用哈希函数 h 将 n 个不同的键哈希到长度为 m 的数组 T 中。假设采用独立均匀哈希，预期的冲突次数是多少？更准确地说，$\{k_1, k_2\}$ 且 $k_1 \neq k_2$ 和 $h(k_1) = h(k_2)$ 的预期基数是多少？

11.2-2

考虑一个有 9 个槽的哈希表和哈希函数 $h(k) = k \bmod 9$。演示插入键 5；28；19；15；20；33；12；17；10 后会发生什么，并通过链接解决冲突。

11.2-3
Marley 教授假设，通过修改链接方案以保持每个列表按排序顺序排列，他可以获得显著的性能提升。教授的修改如何影响成功搜索、不成功搜索、插入和删除的运行时间？

11.2-4
建议如何通过创建 “空闲列表”（所有未使用插槽的链接列表）来分配和释放哈希表本身中的元素存储空间。假设一个插槽可以存储一个 üag 和一个元素加一个指针或两个指针。所有字典和空闲列表操作应在预期时间内运行 O.1/。空闲列表是否需要双向链接，还是单向链接的空闲列表就足够了？

11.2-5
你需要将一组包含 n 个键的密钥存储在一个大小为 m 的哈希表中。说明如果这些密钥来自具有 jU j > .n 1/m 的域 U，则 U 有一个大小为 n 的子集，该子集由所有散列到同一槽的密钥组成，因此使用链接进行散列的最坏情况搜索时间为 ，.n/。

11.2-6
你已经将 n 个密钥存储在一个大小为 m 的哈希表中，并通过链接解决冲突，并且你知道每个链的长度，包括最长链的长度 L。描述一个过程，该过程从哈希表中的密钥中均匀随机地选择一个密钥，并在预期时间内返回该密钥 O.L .1 C 1/。

11.3 哈希函数

要使哈希算法有效运行，就需要一个好的哈希函数。除了计算效率高之外，好的哈希函数还具有哪些属性？如何设计好的哈希函数？

本节首先尝试基于两种创建哈希函数的临时方法来回答这些问题：除法哈希和乘法哈希。虽然这些方法对于某些输入键集很有效，但它们有局限性，因为它们试图提供一个适用于任何称为 ***static hashing*** 的 data4an 方法的单一固定哈希函数。

然后，我们看到，通过设计合适的 *family* 哈希函数并在运行时从该系列中随机选择一个哈希函数（与要哈希的数据无关），可以获得 *any* 数据的可证明的良好平均性能。我们研究的方法称为随机哈希。一种特殊的随机

哈希算法，通用哈希算法，效果很好。正如我们在第 7 章中看到的快速排序一样，随机化是一种强大的算法设计工具。

什么才是好的哈希函数？

好的哈希函数满足（近似地）独立均匀哈希假设：每个密钥都同样可能被哈希到 m 个位置中的任何一个，而与其他密钥被哈希到的位置无关。这里的“相等可能”是什么意思？如果哈希函数是固定的，则任何概率都必须基于输入密钥的概率分布。

不幸的是，你通常没有办法检查这种情况，除非你恰好知道钥匙抽取的概率分布。而且，钥匙可能不是独立抽取的。

有时你可能知道分布。例如，如果你知道密钥是独立且均匀分布在 0 至 k < 1 范围内的随机实数 k，那么哈希函数

$$h(k) = \lfloor km \rfloor$$

满足独立均匀散列的条件。

良好的静态哈希方法会以您期望的方式导出哈希值，使其与数据中可能存在的任何模式无关。例如，“除法”（在 11.3.1 节中讨论）将密钥除以指定素数后的余数作为哈希值。如果您（以某种方式）选择与密钥分布中的任何模式无关的素数，则此方法可能会产生良好的结果。

随机散列法（如第 11.3.2 节中所述）从合适的散列函数系列中随机挑选要使用的散列函数。此方法无需了解输入密钥的概率分布，因为良好平均行为所需的随机化来自用于从散列函数系列中挑选散列函数的（已知）随机过程，而不是来自用于创建输入密钥的（未知）过程。我们建议您使用随机散列法。

键是整数、向量或字符串

实际上，哈希函数被设计用于处理以下两种类型之一的键：

一个短的非负整数，位于 w 位机器字中。w 的典型值为 32 或 64。

一个由非负整数组成的短向量，每个向量的大小都是有限的。例如，每个元素可能是一个 8 位字节，在这种情况下，向量通常称为（字节）字符串。向量的长度可能是可变的。
首先，我们假设密钥是短的非负整数。处理向量密钥更加复杂，将在 11.3.5 和 11.5.2 节中讨论。

11.3.1 静态散列

静态散列使用单个固定的散列函数。唯一可用的随机化是通过输入密钥的（通常未知的）分布。本节讨论了静态散列的两种标准方法：除法和乘法。虽然不再推荐使用静态散列，但乘法也为“非静态”散列4（更好地称为随机散列4）提供了良好的基础，其中散列函数是从合适的散列函数系列中随机选择的。

分割法

用于创建哈希函数的 ***division method*** 通过将 k 除以 m 取余数，将密钥 k 映射到 m 个槽位之一。也就是说，哈希函数是

$h(k) = k \bmod m$.

例如，如果哈希表的大小为 m D 12，密钥为 k D 100，则 h.k/ D 4。由于只需要一个除法运算，因此通过除法进行哈希处理的速度非常快。

当 m 是一个不太接近 2 的精确幂的素数时，除法可能会很好地工作。但是，并不能保证此方法提供良好的平均情况性能，并且它可能会使应用程序复杂化，因为它将哈希表的大小限制为素数。

乘法

创建哈希函数的一般方法分为两个步骤。首先，将密钥 k 乘以 $0 < A < 1$ 范围内的常数 A，并提取 kA 的小数部分。然后，将此值乘以 m，并对结果取üoor。也就是说，哈希函数是

$h(k) = \lfloor m\,(kA \bmod 1) \rfloor$,

其中“kA mod 1”表示kA的小数部分，即kAbkAc。一般乘法的优点是m的值不是至关重要的，你可以独立于乘法常数A来选择m的值。

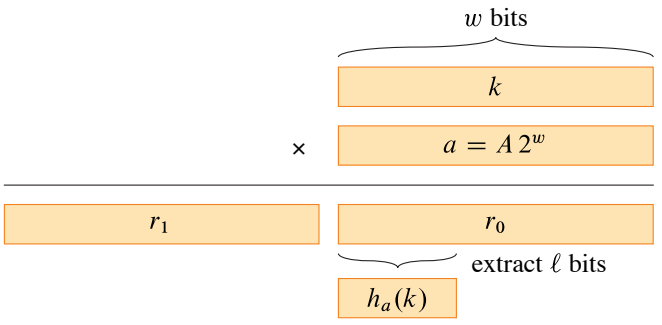


图 11.4 用乘法移位法计算哈希函数。密钥 k 的 w 位表示形式与 w 位值 a D A 2 ^{w} 相乘。乘积低 w 位一半的 ℓ个最高位构成所需的哈希值 h $_{a}$ .k/。

多次移位法

实际上，乘法方法最适合于特殊情况，即哈希表槽的数量 m 是 2 的精确幂，因此对于某个整数 ℓ m D 2 $^{\ell}$，其中 ℓ= w，w 是机器字中的位数。如果您选择一个固定的 w 位正整数 a D A 2 ^{w}，其中 0 < A < 1，如乘法方法中所示，这样 a 的范围为 0 < a < 2 ^{w}，您可以在大多数计算机上按如下方式实现该函数。我们假设密钥 k 被分解为单个 w 位字。

参考图 11.4，首先将 k 乘以 w 位整数 a。结果为 2w 位值 r $_{1}$ 2 ^{w}C r $_{0}$，其中 r $_{1}$ 是乘积的高位 w 位字，r $_{0}$ 是乘积的低位 w 位字。所需的 ℓ位哈希值由 r $_{0}$ 的 ℓ个最高有效位组成。（由于忽略了 r $_{1}$，因此哈希函数可以在给定两个 w 位输入的情况下仅产生 w 位乘积的计算机上实现，即乘法运算计算模 2 ^{w} 。）

换句话说，你定义哈希函数 h D h $_{a}$，其中

$$h_a(k) = (ka \bmod 2^w) \ggg (w - \ell) \tag{11.2}$$

对于固定非零 w 位值 a。由于两个 w 位字的乘积 ka 占用 2w 位，因此将此乘积模 2^w 会将高位 w 位（r_1）清零，只留下低位 w 位（r_0）。o 运算符执行逻辑右移 w ℓ位，将零移到左侧空出的位置，这样 r_0 的 ℓ个最高有效位移到 ℓ最右边的位置。（这与除以 $2^{w-\ell}$ 并取结果的下限相同。）结果值等于 r_0 的 ℓ个最高有效位。哈希函数 h_a 可以用三条机器指令实现：乘法、减法和逻辑右移。

举例来说，假设 k D 123456、ℓD 14、m D 2 ^{14} D 16384 和 w D 32。进一步假设我们选择一个 D 2654435769（按照建议

Knuth [261] 的证明）。则 ka D 327706022297664 D .76300 − 2 ^{32} /C 17612864，因此 r $_1$ D 76300 和 r $_0$ D 17612864。r $_0$ 的 14 个最高有效位得出值 h $_a$ .k/D 67。

尽管乘法移位法速度很快，但它并不能保证平均情况下的性能良好。下一节介绍的通用哈希方法提供了这样的保证。当程序开始时选择 a 作为随机选择的奇数整数时，乘法移位法的简单随机变体在平均情况下效果很好。

11.3.2 随机散列

假设恶意攻击者选择要通过某个固定哈希函数进行哈希处理的密钥。那么攻击者可以选择 n 个密钥，这些密钥都哈希到同一个槽中，平均检索时间为 ，.n/。任何静态哈希函数都容易受到这种可怕的最坏情况的影响。改善这种情况的唯一有效方法是选择哈希函数 *randomly*，使其等于实际要存储的密钥的 *independent* 。这种方法称为 ***random hashing***。这种方法的一个特例称为 ***universal hashing***，当通过链接处理冲突时，无论攻击者选择哪个密钥，都可以平均获得可证明的良好性能。

要使用随机散列，请在程序执行开始时从合适的函数系列中随机选择散列函数。与快速排序的情况一样，随机化可确保没有任何单个输入总是会引起最坏情况的行为。由于您随机选择散列函数，因此即使对于同一组要散列的键，算法在每次执行时都可以有不同的表现，从而保证良好的平均性能。

令 H 为一个哈希函数有限族，将给定的密钥域 U 映射到范围 f0；1；：：：；m 1g 中。如果对于每对不同的密钥 k $_1$ ；k $_2$ 2 U，满足 h.k $_1$ /D h.k $_2$ / 的哈希函数数量 h 2 H 最多为 jH j /m，则称这样的哈希函数族为 ***universal***。换句话说，对于从 H 中随机选择的哈希函数，如果 h.k $_1$ / 和 h.k $_2$ / 是从集合 f0；1；：：：；m 1g 中随机独立地选择，则不同密钥 k $_1$ 和 k $_2$ 之间发生碰撞的概率不超过发生碰撞的概率 1/m。

独立均匀散列与从 m 个 ^{n} 散列函数系列中随机均匀地选取一个散列函数相同，该散列函数系列的每个成员都以不同的方式将 n 个键映射到 m 个散列值。

每个独立的均匀随机哈希函数族都是通用的，但反之不一定成立：考虑 U D f0; 1; : : : ; m 1g 的情况，并且该族中唯一的哈希函数是恒等函数。两个不同密钥发生冲突的概率为零，即使每个密钥都被哈希为一个固定值。

第 279 页定理 11.2 的以下推论表明，通用散列提供了期望的回报：对手不可能选择强制最坏情况运行时间的操作序列。

Corollary11.3
使用通用散列和冲突解决，通过链接到具有 m 个槽的最初为空的表，需要花费，.s/ 预期时间来处理包含 n D O.m/ 个 I NSERT 操作的任何 s I NSERT 、SEARCH 和 DELETE 操作序列。

Proof I NSERT 和 DELETE 操作花费的时间是常数。由于插入次数 n 为 O.m/ ，因此有 *α*D O.1/。此外，每个 SEARCH 操作的预期时间为 O.1/，这可以通过检查定理 11.2 的证明看出。该分析仅取决于碰撞概率，通过选择该定理中的独立均匀散列函数，对于任何键对 $k_1; k_2$，碰撞概率为 1/m。在这里使用通用散列函数系列而不是使用独立均匀散列会将碰撞概率从 1/m 更改为最多 1/m。因此，根据期望的线性，整个 s 个操作序列的预期时间为 O.s/ 。由于每个操作花费 .1/ 时间，因此 .s/ 界限成立。

■

11.3.3 随机散列的可实现属性

关于 H 族哈希函数的属性以及它们与哈希效率的关系，有大量文献。我们在此总结了一些最有趣的内容。

假设 H 为一组哈希函数，每个函数的定义域为 U，取值范围为 f0; 1; : : : ; m 1g，假设 h 是从 H 中均匀随机选取的任意哈希函数。所提到的概率都是 h 选取的概率。

如果对于 U 中的任意密钥 k 以及范围 f0；1；：：：；m 1g 中的任意槽 q，h.k/D q 的概率为 1/m，则哈希函数族 H 为 ***uniform***。• 如果对于 U 中的任意不同密钥 k_1 和 k_2，$h.k_1/D h.k_2/$ 的概率至多为 1/m，则哈希函数族 H 为 ***universal***。• 如果对于 U 中的任意不同密钥 k_1 和 k_2，$h.k_1/D h.k_2/$ 的概率至多为 ，则哈希函数族 H 为 ***-universal***。因此，通用哈希函数族也是 1/m 通用的。[2]

---

[2] In the literature, a (c/m)-universal hash function is sometimes called c-universal or c-approximately universal. We'll stick with the notation (c/m)-universal.

如果对于 U 中任何不同的键 k 1 , k 2 , ... , k d 以及 f0; 1; : : :; m 中的任何槽 q 1 , q 2 , ... , q d （不一定不同），则家族 H 为 d **-independent**，如果对于 f0; 1; : : :; m 1g，h.k i / D q i 的概率为 i D 1; 2; : : :; d 为 1/m d 。
通用哈希函数系列尤其令人感兴趣，因为它们是最简单的类型，支持对任何输入数据集进行可证明有效的哈希表操作。许多其他有趣且理想的属性（例如上述属性）也是可能的，并允许进行有效的专用哈希表操作。

11.3.4 设计通用的哈希函数系列

本节介绍两种设计通用（或 通用）哈希函数系列的方法：一种基于数论，另一种基于第 11.3.1 节中介绍的乘法移位方法的随机变体。第一种方法更容易证明通用性，但第二种方法在实践中更新、更快。

基于数论的通用哈希函数系列

我们可以用一点数论知识来设计一个通用的哈希函数系列。如果你不熟悉数论中的基本概念，你可以参考第 31 章。

首先选择一个足够大的素数 p，使得每个可能的密钥 k 都在 0 到 p 1 的范围内（包括 0 和 p 1）。这里我们假设 p 具有“合理的” 长度。（有关处理长输入密钥（如变长字符串）的方法的讨论，请参见第 11.3.5 节。）令 Z p 表示集合 f0；1；：：：；p 1g，令 Z_p^* 表示集合 f1；2；：：：；p 1g。由于 p 是素数，我们可以使用第 31 章中给出的方法解模 p 方程。由于密钥范围的大小大于哈希表中的槽数（否则，只需使用直接寻址），我们有 p > m。

给定任意 a 2 Z_p^* 和任意 b 2 Z p，将哈希函数 h ab 定义为线性变换，然后对 p 进行模约简，然后对 m 进行模约简：

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m . \tag{11.3}$$

例如，对于 p D 17 和 m D 6，我们有

$$\begin{aligned} h_{3,4}(8) &= ((3 \cdot 8 + 4) \bmod 17) \bmod 6 \\ &= (28 \bmod 17) \bmod 6 \\ &= 11 \bmod 6 \\ &= 5 . \end{aligned}$$

给定 p 和 m，所有此类哈希函数系列为

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\} . \tag{11.4}$$

每个哈希函数 h $_{ab}$ 将 Z $_{p}$ 映射到 Z $_{m}$。该哈希函数系列具有良好的特性，即输出范围的大小 m（即哈希表的大小）是任意的 4 它不必是素数。由于您可以从 a 的 p 1 个值和 b 的 p 个值中进行选择，因此系列 H $_{pm}$ 包含 p.p 1/ 个哈希函数。

Theorem11.4
由公式 (11.3) 和公式 (11.4) 定义的哈希函数族 H_{pm} 是通用的。

Proof考虑来自 Z $_{p}$ 的两个不同密钥 k $_{1}$ 和 k $_{2}$，因此 k $_{1}$ ¤ k $_{2}$。对于给定的哈希函数 h $_{ab}$，让

$$r_1 = (ak_1 + b) \bmod p ,$$
$$r_2 = (ak_2 + b) \bmod p .$$

我们首先注意到 r $_{1}$ ¤ r $_{2}$。为什么？因为我们有 r $_{1}$ r $_{2}$ D a.k $_{1}$ k $_{2}$/ .mod p/，因此可知 r $_{1}$ ¤ r $_{2}$，因为 p 为素数，并且 a 和 .k $_{1}$ k $_{2}$/ 模 p 非零。根据第 908 页的定理 31.6，它们的乘积也必须模 p 非零。因此，在计算任何 h $_{ab}$ 2 H $_{pm}$ 时，不同的输入 k $_{1}$ 和 k $_{2}$ 映射到不同的值 r $_{1}$ 和 r $_{2}$ 模 p，并且在 “mod p 级别尚未发生碰撞。” 此外，对于 a ¤ 0 的对 .a; b/，每个可能的 p.p 1/ 选择都会产生一个 *different* 结果对 .r $_{1}$; r $_{2}$/ 和 r $_{1}$ ¤ r $_{2}$，因为我们可以根据 r $_{1}$ 和 r $_{2}$ 解出 a 和 b：

$$a = \big((r_1 - r_2)((k_1 - k_2)^{-1} \bmod p)\big) \bmod p ,$$
$$b = (r_1 - ak_1) \bmod p ,$$

其中 ..k $_{1}$ k $_{2}$/ $^{-1}$ mod p/ 表示 k $_{1}$ k $_{2}$ 的唯一乘法逆元，模 p。对于 r $_{1}$ 的 p 个可能值中的每一个，r $_{2}$ 仅有 p 1 个可能值不等于 r $_{1}$，从而仅存在 p.p 1/ 个可能的对 .r $_{1}$; r $_{2}$/ 和 r $_{1}$ ¤ r $_{2}$。因此，对 .a; b/ 和 a ¤ 0 以及对 .r $_{1}$; r $_{2}$/ 和 r $_{1}$ ¤ r $_{2}$ 之间存在一一对应关系。因此，对于任何给定的不同输入对 k $_{1}$ 和 k $_{2}$，如果我们从 Z $_{p}^{*}$ Z $_{p}$ 中均匀随机地选取 .a; b/，则得到的对 .r $_{1}$; r $_{2}$/ 同样可能是模 p 的任何一对不同值。

因此，当 r $_{1}$ 和 r $_{2}$ 被随机选为模 p 的不同值时，不同密钥 k $_{1}$ 和 k $_{2}$ 发生冲突的概率等于 r $_{1}$ D r $_{2}$ .mod m/ 的概率。对于给定的 r $_{1}$ 值，在 r $_{2}$ 的 p 1 个可能剩余值中，满足 r $_{2}$ ¤ r $_{1}$ 和 r $_{2}$ D r $_{1}$ .mod m/ 的值 r $_{2}$ 的数量最多为

$$\left\lceil \frac{p}{m} \right\rceil - 1 \le \frac{p + m - 1}{m} - 1 \quad \text{(by inequality (3.7) on page 64)}$$
$$= \frac{p - 1}{m} .$$

当模 m 减小时，r_2 与 r_1 发生冲突的概率最多为 ..p 1//m//.p 1/D 1/m，因为 r_2 同样可能是 Z_p 中不同于 r_1 的 p 1 个值中的任意一个，但是这些值中最多有 .p 1//m 等同于 r_1 模 m。

因此，对于任何一对不同的值 $k_1; k_2 \in Z_p$，

$$\Pr\{h_{ab}(k_1) = h_{ab}(k_2)\} \leq 1/m ,$$

所以 H_{pm} 确实是通用的。 ■

基于乘法移位方法的 2 /m-通用哈希函数系列

我们建议您在实践中使用以下基于乘法移位方法的哈希函数系列。它非常高效，并且（尽管我们省略了证明）可证明是 2/m 通用的。假设 H 是具有奇数常数 a 的乘法移位哈希函数系列：

$$\mathcal{H} = \{h_a : a \text{ is odd}, 1 \leq a < m, \text{and } h_a \text{ is defined by equation (11.2)}\} . \qquad (11.5)$$

Theorem11.5
由公式（11.5）给出的哈希函数族H是2/m-通用的。 ■

也就是说，任何两个不同的密钥发生冲突的概率最多为 2/m。在许多实际情况下，与通用哈希函数相比，计算哈希函数的速度足以弥补两个不同密钥发生冲突的概率的较高上限。

11.3.5 对长输入（如向量或 st）进行哈希处理 戒指

有时哈希函数的输入太长，以至于无法轻易地对一个合理大小的素数 p 进行模数编码，也无法将其编码在一个字（例如 64 位）内。例如，考虑向量类，例如 8 位字节的向量（这是许多编程语言中字符串的存储方式）。向量可能具有任意非负长度，在这种情况下，哈希函数的输入长度可能因输入而异。

数论方法

设计适合可变长度输入的良好哈希函数的一种方法是扩展第 11.3.4 节中使用的思想来设计通用哈希函数。练习 11.3-6 探讨了一种方法。

加密哈希

为可变长度输入设计良好哈希函数的另一种方法是使用为加密应用程序设计的哈希函数。***Cryptographic hash func- tions*** 是复杂的伪随机函数，专为需要超出此处所需属性的应用程序而设计，但它们具有鲁棒性、广泛实现，并且可以用作哈希表的哈希函数。

加密哈希函数以任意字节字符串作为输入，并返回固定长度的输出。例如，NIST 标准确定性加密哈希函数 SHA-256 [346] 可为任何输入生成 256 位（32 字节）的输出。

一些芯片制造商在其 CPU 架构中包含指令，以提供某些加密函数的快速实现。特别令人感兴趣的是有效实现高级加密标准 (AES) 轮次的指令，即 “AES-NI” 指令。这些指令的执行时间只有几十纳秒，通常足以用于哈希表。基于 AES 的消息认证代码（例如 CBC-MAC）和 AES-NI 指令的使用可能是一种有用且高效的哈希函数。我们在此不进一步探讨专用指令集的潜在用途。

加密哈希函数很有用，因为它们提供了一种实现随机预言近似版本的方法。如前所述，随机预言相当于一个独立的统一哈希函数系列。从理论的角度来看，随机预言是一个无法实现的理想：一个确定性函数，为每个输入提供随机选择的输出。因为它是确定性的，所以如果再次查询相同的输入，它会提供相同的输出。从实践的角度来看，基于加密哈希函数构建哈希函数系列是随机预言的合理替代品。

有很多方法可以将加密哈希函数用作哈希函数。例如，我们可以定义

$$h(k) = \text{SHA-256}(k) \bmod m \ .$$

要定义此类哈希函数系列，可以在对输入进行哈希处理之前，先在输入前面添加一个 “salt” 字符串 a，如下所示

$$h_a(k) = \text{SHA-256}(a \parallel k) \bmod m \ ,$$

其中 a k k 表示由字符串 a 和 k 连接而成的字符串。消息认证码 (MAC) 方面的文献提供了其他方法。

随着计算机将内存排列成不同容量和速度的层次结构，哈希函数设计的加密方法变得越来越实用。第 11.5 节讨论了一种基于 RC6 加密方法的哈希函数设计。

练习

11.3-1

您希望搜索长度为 n 的链接列表，其中每个元素包含一个键 k 以及一个哈希值 h.k/。每个键都是一个长字符串。在列表中搜索具有给定键的元素时，如何利用哈希值？

11.3-2

通过将包含 r 个字符的字符串视为基数为 128 的数字，然后使用除法，可以将该字符串哈希处理为 m 个位置。您可以将数字 m 表示为 32 位计算机字，但将包含 r 个字符的字符串视为基数为 128 的数字需要很多字。如何应用除法来计算字符串的哈希值，而无需在字符串本身之外使用超过常数个存储字数？

11.3-3

考虑除法的一个版本，其中 h.k/ D k mod m，其中 m D 2 ^{p} 1，k 是以基数 2 ^{p} 解释的字符串。说明如果可以通过排列字符串 x 的字符将其转换为字符串 y，则 x 和 y 的哈希值相同。给出一个在哈希函数中不需要此属性的应用示例。

11.3-4

考虑一个大小为 m D 1000 的哈希表和相应的哈希函数 h.k/ D bm .kA mod 1/c（其中 A D . p 5 1//2）。计算键 61、62、63、64 和 65 映射到的位置。

11.3-5

证明从有限集 U 到有限集 Q 的任何 -通用哈希函数族 H 都有 1 1/ jQj 1/ jU j。

11.3-6

假设 U 是从 Z $_{p}$ 中提取的 d 元组值的集合，假设 Q D Z $_{p}$，其中 p 为素数。对来自 U 的输入 d 元组 ha $_{0}$; a $_{1}$; : : : ; a $_{d-1}$ i 定义 b 2 Z $_{p}$ 的哈希函数 h $_{b}$ W U ! Q，如下所示

$$h_b(\langle a_0, a_1, \ldots, a_{d-1}\rangle) = \left(\sum_{j=0}^{d-1} a_j b^j\right) \bmod p ,$$

令 H D fh $_{b}$ W b 2 Z $_{p}$ g。论证 H 对于 D .d 1//p 是 -通用的。（*Hint:* 参见练习 31.4-4。）

11.4 开放寻址

本节介绍开放寻址，这是一种冲突解决方法，与链接不同，它不使用哈希表本身之外的存储空间。在 ***open addressing*** 中，所有元素都占据哈希表本身。也就是说，每个表条目都包含动态集的元素或 NIL。与链接不同，没有列表或元素存储在表之外。因此，在开放寻址中，哈希表可以 “ 填满 ”，这样就无法再进行插入。一个结果是负载因子 α 永远不会超过 1。

冲突处理如下：当要将新元素插入表中时，如果可能，则将其放置在其 “ ûrst-choice” 位置。如果该位置已被占用，则将新元素放置在其 “second-choice” 位置。该过程继续，直到找到一个空位置来放置新元素。不同的元素对这些位置的偏好顺序不同。

要搜索某个元素，请按优先顺序系统地检查该元素的首选表槽，直到找到所需的元素或找到一个空槽，从而验证该元素不在表中。

当然，您可以使用链接并将链接列表存储在哈希表中未使用的哈希表槽中（参见练习 11.2-4），但开放寻址的优点是它完全避免了指针。您无需遵循指针，而是计算要检查的槽序列。通过不存储指针而释放的内存为哈希表提供了相同内存量中的更多槽，从而可能减少冲突并加快检索速度。

要使用开放寻址执行插入操作，请连续检查哈希表（或 ***probe***），直到找到一个可以放置密钥的空槽。探测位置的顺序不是固定为 0; 1; : : : ; m 1（这意味着搜索时间为 1.n），而是取决于插入的密钥。为了确定要探测哪些槽，哈希函数将探测编号（从 0 开始）作为第二个输入。因此，哈希函数变为

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\} \ .$$

开放寻址要求对于每个键 k，***probe sequence*** hh.k; 0/; h.k; 1/; : : : ; h.k; m 1/i 是 h0; 1; : : : ; m 1i 的排列，因此当哈希表填满时，每个哈希表位置最终都被视为新键的插槽。下一页的 HASH-INSERT 过程假设哈希表 T 中的元素是没有卫星信息的键：键 k 与包含键 k 的元素相同。每个插槽包含一个键或 NIL（如果插槽为空）。HASH-INSERT 过程将哈希表 T 和键 k 作为输入

假设哈希表中还不存在该键。它要么返回存储键 k 的槽号，要么发出错误，因为哈希表已满。

```
HASH-I NSERT .T; k/

1 i D 0 2 重复 3 q D h.k; i / 4
如果 T Œq    ==NIL 5 T Œq
    D k 6 返回 q 7 否则 i D i C
1 8 直到 i ==m 9 错误 “哈希
表溢出”



HASH-SEARCH.T; k/

1 i D 0 2 重复 3 q D h.k; i / 4
如果 T Œq    ==k 5 返回 q 6 i
D i C 1 7 直到 T Œq    ==NIL
或 i ==m 8 返回 NIL
```

搜索密钥 k 的算法探测的槽序列与插入密钥 k 时插入算法检查的槽序列相同。因此，当找到一个空槽时，搜索可能会终止（失败），因为 k 会插入到那里，而不是稍后插入到其探测序列中。过程 H ASH-SEARCH 将哈希表 T 和密钥 k 作为输入，如果发现槽 q 包含密钥 k，则返回 q，如果表 T 中不存在密钥 k，则返回 NIL。

从开放地址哈希表中删除密钥比较棘手。当您从插槽 q 中删除密钥时，如果只是在插槽 q 中存储 NIL 就将该插槽标记为空，那就错了。如果这样做，您可能无法检索任何密钥 k，因为在插入 k 时，插槽 q 被探测到并被占用。解决此问题的一种方法是标记插槽，在其中存储特殊值 DELETED 而不是 NIL。然后，HASH-INSERT 过程必须将此类插槽视为空，以便可以在那里插入新密钥。HASH-SEARCH 过程在搜索时会忽略 DELETED 值，因为在插入要搜索的密钥时，包含 DELETED 的插槽已被填充。但是，使用特殊值 DELETED 意味着搜索时间不再取决于加载因子 α 因此链接是

当必须删除键时，经常选择将其作为冲突解决技术。开放寻址有一个简单的特殊情况，即线性探测，它避免了用 DELETED 标记槽位的需要。第 11.5.1 节介绍了使用线性探测时如何从哈希表中删除。

在我们的分析中，我们假设 ***independent uniform permutation hashing***（在文献中也称为 ***uniform hashing***，容易引起混淆）：每个键的探测序列同样可能是 h0; 1; : : : ; m 1i 的 mŠ 排列中的任何一个。独立均匀排列散列将先前定义的独立均匀排列概念推广为一个散列函数，该散列函数不仅产生单个槽号，而且产生整个探测序列。然而，真正的独立均匀排列散列很难实现，在实践中使用合适的近似值（例如双重散列，如下所述）。

我们将研究双重散列及其特殊情况——线性探测。这些技术保证对于每个键 k，hh.k；0/；h.k；1/；：：：：；h.k；m 1/i 是 h0；1；：：：：；m 1i 的排列。（回想一下，散列函数 h 的第二个参数是探测号。）但是，双重散列和线性探测均不符合独立均匀排列散列的假设。双重散列不能生成超过 m^2 个不同的探测序列（而不是独立均匀排列散列所要求的 mŠ）。尽管如此，双重散列具有大量可能的探测序列，并且如您所料，它似乎能给出良好的结果。线性探测受到的限制更多，只能生成 m 个不同的探测序列。

双重哈希

双重散列是开放寻址的最佳方法之一，因为产生的排列具有许多随机选择排列的特征。***Double hashing*** 使用以下形式的散列函数

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m ,$$

其中 h_1 和 h_2 均为 ***auxiliary hash functions***。初始探测到位置 T Œh $_1$.k/ ，后续探测位置与先前位置的偏移量为 h_2.k/，模 m。因此，此处的探测序列以两种方式依赖于密钥 k，因为初始探测位置 h_1.k/、步长 h_2.k/ 或两者可能会发生变化。图 11.5 给出了通过双重散列进行插入的示例。

为了搜索整个哈希表，值 h_2.k/ 必须与哈希表大小 m 互质。（参见练习 11.4-5。）确保此条件的一个方便方法是让 m 为 2 的精确幂，并设计 h_2 使其始终产生奇数。另一种方法是让 m 为素数，并设计 h_2 使其始终返回小于 m 的正整数。例如，

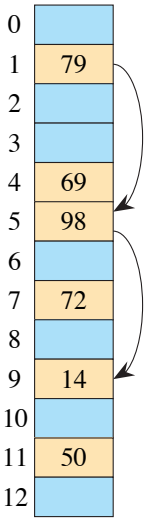


图 11.5 通过双重散列进行插入。哈希表的大小为 13，其中 $h_1.k/ D k \bmod 13$ 和 $h_2.k/ D 1 C . k \bmod 11/$。由于 14 D 1 .mod 13/ 和 14 D 3 .mod 11/，因此在检查槽 1 和 5 并发现它们已被占用后，密钥 14 进入空槽 9。

可以选择 m 素数并让

$$h_1(k) = k \bmod m ,$$
$$h_2(k) = 1 + (k \bmod m') ,$$

其中 m′ 被选择为略小于 m（例如，m 1）。例如，如果 k D 123456、m D 701 和 m′ D 700，则 $h_1.k/ D 80$ 和 $h_2.k/ D 257$，因此第一次探测进入位置 80，后续探测检查每 257 个槽（模 m），直到找到密钥或检查完每个槽。

尽管在原则上，除素数或 2 的精确幂之外的 m 值都可以用于双重散列，但在实践中，要想有效地生成 $h_2.k/$（除了选择 $h_2.k/ D 1$，这可以进行线性探测），并确保它与 m 互质，变得更加困难，部分原因是对于一般的 m，这类数字的相对密度 .m//m 可能很小（参见第 921 页的公式 (31.25)）。

当 m 为素数或 2 的精确幂时，双重散列会产生 ，$.m^2/$ 探测序列，因为每个可能的 $.h_1.k/; h_2.k//$ 对都会产生不同的探测序列。因此，对于这样的 m 值，双重散列似乎表现得接近独立均匀置换散列的 “ideal” 方案。

线性探测

Linearprobing 是双重散列的一个特例，是解决冲突的最简单的开放寻址方法。与双重散列一样，辅助散列函数 h 1 确定插入元素的第一个探测位置 h 1 .k /。如果槽 T Œh 1 .k/ 已被占用，则探测下一个位置 T Œh 1 .k/ C 1 。根据需要继续探测，直到槽 T Œm 1 ，然后绕回到槽 T Œ0 、T Œ1 等等，但永远不会超过槽 T Œh 1 .k/ 1 。要将线性探测视为双重散列的一个特例，只需将双重散列步长函数 h 2 固定在 1：对于所有 k，h 2 .k/ D 1。也就是说，散列函数是

$$h(k,i) = (h_1(k) + i) \bmod m \tag{11.6}$$

对于 i D 0; 1; : : : ; m 1. h 1 .k/ 的值决定了整个探测序列，因此假设 h 1 .k/ 可以取 f0; 1; : : : ; m 1g 中的任意值，线性探测只允许 m 个不同的探测序列。

我们将在第 11.5.1 节中重新讨论线性探测。

开放地址哈希分析

和 11.2 节中对链式引用的分析一样，我们根据哈希表的负载因子 ˛D n/m 来分析开放寻址。使用开放寻址时，每个槽最多只有一个元素，因此 n = m，这意味着 ˛ 1。下面的分析要求 ˛严格小于 1，因此我们假设至少有一个槽是空的。因为从开放地址哈希表中删除并不会真正释放槽，所以我们也假设没有发生删除。

对于哈希函数，我们假设独立的均匀排列哈希。在这个理想化的方案中，用于插入或搜索每个键 k 的探测序列 hh.k；0/；h.k；1/；：：：：；h.k；m 1/i 等有可能是 h0；1；：：：；m 1i 的任意排列。当然，任何给定的键都有一个与之关联的唯一固定探测序列。我们在这里指的是，考虑到键空间上的概率分布和哈希函数对键的操作，每个可能的探测序列等有可能。

我们现在在独立均匀置换散列的假设下，分析开放寻址散列的预期探测次数，从不成功搜索的预期探测次数开始（假设，如上所述，$\alpha < 1$）。

已证明的界限为 1/.1 ˛D 1 C ˛C ˛2 C ˛3 C ，具有直观的解释。第一次探测总是会发生。大约有 ˛的概率，第一次探测会找到一个被占用的槽，因此会发生第二次探测。大约有 ˛2 的概率，前两个槽被占用，因此会进行第三次探测，依此类推。

Theorem11.6
给定一个负载因子为$\alpha = n/m < 1$的开放地址哈希表，假设独立均匀排列哈希且没有删除，则一次不成功搜索的预期探测次数最多为$1/(1-\alpha)$。

Proof在一次不成功的搜索中，除了最后一次探测之外，每次探测都会访问一个不包含所需密钥的已占用槽位，并且最后探测的槽位是空的。设随机变量 X 表示一次不成功搜索中所进行的探测次数，并定义事件 A_i（对于 i D 1；2；：：：）为第 i 次探测发生并且到达已占用槽位的事件。那么事件 fX i g 就是事件 $A_1 \backslash A_2 \backslash \ \backslash A_{i-1}$ 的交集。我们通过限制 Pr $fA_1 \backslash A_2 \backslash \ \backslash A_{i-1}$ g 来限制 Pr fX i g。根据第 1190 页的练习 C.2-5,

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ &\qquad \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} \ .\end{aligned}$$

由于有 n 个元素和 m 个槽位，因此 Pr fA_1 g D n/m。对于 j > 1，假设前 j 1 个探测指向已占用的槽位，则存在第 j 次探测并且它指向已占用的槽位的概率为 .n j C 1//.m j C 1/。这个概率的原因是，第 j 次探测将在 .m .j 1// 个未检查的槽位之一中找到剩余的 .n .j 1// 个元素之一，并且根据独立均匀排列散列的假设，概率是这些数量的比率。由于 n < m 意味着对于所有在 0 至 j < m 范围内的 j，.n j //.m j / n/m，因此对于所有在 1 至 i 至 m 范围内的 i，我们有

$$\begin{aligned}\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} \ .\end{aligned}$$

第一行中的乘积有 i 1 个因子。当 i D 1 时，乘积为 1，即乘法恒等式，我们得到 Pr fX ? 1g D 1，这是有道理的，因为必须始终至少有 1 个探测。如果前 n 次探测都针对已占用的槽位，则所有已占用的槽位都已探测过。然后，第 n C 1/st 次探测必须针对一个空槽位，这为 i > n C 1 得出 Pr fX ? i g D 0。现在，我们使用第 1193 页上的公式 (C.28) 来限制预期的探测次数：

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &= \sum_{i=1}^{n+1} \Pr\{X \geq i\} + \sum_{i>n+1} \Pr\{X \geq i\}\end{aligned}$$

$$\leq \sum_{i=1}^{\infty} \alpha^{i-1} + 0$$
$$= \sum_{i=0}^{\infty} \alpha^{i}$$
$$= \frac{1}{1-\alpha} \qquad \text{(by equation (A.7) on page 1142 because } 0 \leq \alpha < 1) \text{ .} \qquad \blacksquare$$

如果 α为常数，则定理 11.6 预测一次不成功的搜索将花费 O.1/ 次的时间。例如，如果哈希表半满，则一次不成功搜索的平均探测次数最多为 1/.1 :5/ D 2。如果哈希表 90% 满，则平均探测次数最多为 1/.1 :9/ D 10。

定理 11.6 几乎可以立即得出 H ASH-I NSERT 过程的执行情况。

Corollary11.7
将元素插入到负载因子为 α(其中 $\alpha <$ 为 1）的开放地址哈希表中，平均最多需要 1/.1 α次探测，假设独立的均匀排列哈希并且没有删除。

Proof仅当表中有空间时才会插入元素，因此为 $\alpha < 1$。插入键需要搜索失败，然后将键放入找到的第一个空槽中。因此，预期探测次数最多为 1/.1 α。 ■

计算成功搜索的预期探测次数还需要做更多的工作。

Theorem11.8
给定一个负载因子为 $\alpha < 1$ 的开放地址哈希表，成功搜索的预期探测次数最多为

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} ,$$

假设独立的均匀排列散列没有删除，并且假设表中的每个键被搜索的可能性都是相同的。

Proof搜索键 k 会重现与插入键 k 的元素时相同的探测序列。如果 k 是插入哈希表中的第 .i C 1/st 个键，则插入时的负载因子为 i/m，因此根据推论 11.7，搜索 k 时进行的探测次数预期最多为 1/.1 i/m/ D m/.m i /。对哈希表中所有 n 个键取平均值可得出

成功搜索的预期探测次数：

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}\,dx \qquad \text{(by inequality (A.19) on page 1150)} \\
&= \frac{1}{\alpha}\left(\ln m - \ln(m-n)\right) \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\,.
\end{aligned}$$

■

如果哈希表半满，则成功搜索的预期探测次数小于 1:387。如果哈希表 90 % 满，则预期探测次数小于 2:559。如果 $\alpha = 1$，则在不成功的搜索中，必须探测所有 m 个槽。练习 11.4-4 要求您分析当 $\alpha = 1$ 时的成功搜索。

练习

11.4-1
考虑使用开放寻址将键 10; 22; 31; 4; 15; 28; 17; 88; 59 插入到长度为 m D 11 的哈希表中。使用 h.k; i / D .k C i / mod m 的线性探测和使用 h_1.k/ D k 和 h_2 .k/ D 1 C .k mod .m 的双重哈希来说明插入这些键的结果。

11.4-2
编写 HASH-DELETE 的伪代码，用特殊值 DELETED 填充已删除键的槽，并根据需要修改 HASH-SEARCH 和 HASH-INSERT 来处理 DELETED。

11.4-3
考虑一个具有独立均匀置换散列且无删除的开放地址哈希表。当负载因子为 3/4 和 7/8 时，给出不成功搜索的预期探测次数的上限以及成功搜索的预期探测次数的上限。

11.4-4
证明当 $d = 1$（即 $n = m$）时，成功搜索所需的预期探测次数为 H_m，即第 m 个谐波数。

11.4-5
证明：使用双重哈希，如果 m 和 $h_2(k)$ 对于某个密钥 k 具有最大公约数 $d \geq 1$，则对密钥 k 的不成功搜索将检查哈希表的第 $(1/d)$ 个，然后返回到槽 $h_1(k)$。因此，当 $d = 1$ 时，m 和 $h_2(k)$ 互质，搜索可能会检查整个哈希表。（*Hint:* 参见第 31 章。）

11.4-6
考虑一个负载因子为 α 的开放地址哈希表。近似非零值 α 使得不成功搜索的预期探测次数等于成功搜索的预期探测次数的两倍。对于这些预期探测次数，请使用定理 11.6 和 11.8 给出的上限。

11.5 实际考虑

高效的哈希表算法不仅具有理论意义，而且具有巨大的实际意义。常数因素可能很重要。因此，本节讨论了现代 CPU 的两个方面，它们未包含在第 2.2 节中介绍的标准 RAM 模型中：

内存层次结构：现代 CPU 的内存有许多级别，从快速寄存器到一个或多个 ***cache memory*** 级别，再到主内存级别。每个后续级别存储的数据都比前一个级别多，但访问速度较慢。因此，完全在快速寄存器内进行的复杂计算（例如复杂的哈希函数）所花的时间可能比从主内存中读取一次操作所花的时间要少。此外，高速缓存以 ***cache blocks*** 的形式组织，每个缓存（例如）64 字节，它们总是一起从主内存中提取。确保内存使用是本地的，这有很大的好处：重用同一个缓存块比从主内存中提取不同的缓存块要高效得多。

标准 RAM 模型通过计算探测的哈希表槽的数量来衡量哈希表操作的效率。实际上，这个指标只是对事实的粗略近似，因为一旦缓存块进入缓存，对该缓存块的连续探测比必须访问主内存的探测要快得多。

高级指令集：现代 CPU 可能具有复杂的指令集，可实现对加密或其他形式的密码术有用的高级原语。这些指令可能有助于设计极其高效的哈希函数。

第 11.5.1 节讨论了线性探测，在内存层次结构存在的情况下，它成为首选的冲突解决方法。第 11.5.2 节建议如何基于加密原语构建 “ 高级 ” 哈希函数，适合在具有层次化内存模型的计算机上使用。

11.5.1 线性探测

线性探测经常被贬低，因为它在标准 RAM 模型中性能不佳。但线性探测在分层内存模型中表现优异，因为连续探测通常针对的是同一缓存内存块。

线性探测删除

线性探测在实践中经常不使用的另一个原因是，如果不使用特殊的 DELETED 值，删除操作似乎很复杂或不可能。然而，我们现在将看到，即使没有 DELETED 标记，从基于线性探测的哈希表中删除也不那么困难。删除过程适用于线性探测，但不适用于一般的开放地址探测，因为使用线性探测时，所有键都遵循相同的简单循环探测序列（尽管起点不同）。

删除过程依赖于线性探测哈希函数 h.k; i / D .h $_1$ .k/ C i / mod m 的 “ 逆” 函数，该函数将密钥 k 和探测号 i 映射到哈希表中的槽号。逆函数 g 将密钥 k 和槽号 q（其中 0 # q < m）映射到到达槽 q 的探测号：

$$g(k, q) = (q - h_1(k)) \bmod m \ .$$

如果 h.k; i / D q，则 g.k; q/ D i，因此 h.k; g.k; q// D q。

对页上的过程 LINEAR-PROBING-HASH-DELETE 从哈希表 T 中删除存储在位置 q 的键。图 11.6 显示了其工作原理。过程 ûrst 通过在第 2 行中将 T Œq 设置为 NIL 来删除位置 q 中的键。然后，它搜索槽 q′（（如果有）），其中包含应移动到键 k 刚空出的槽 q 中的键。第 9 行提出了一个关键问题：是否需要将槽 q′ 中的键 k′ 移动到空出的槽 q，以保持 k′ 的可访问性？如果 g.k′; q/ < g.k′; q′/，则在将 k′ 插入表期间，检查了槽 q，但发现该槽已被占用。但现在搜索 k′ 的槽位 q 为空。在这种情况下，键 k′ 移至第 10 行的槽位 q，并且

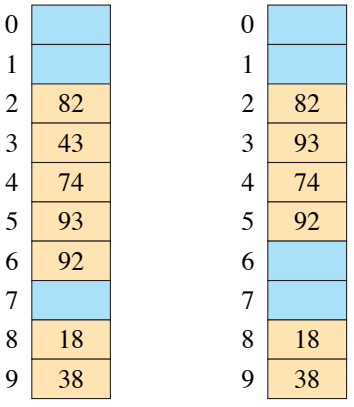


图 11.6 使用线性探测的哈希表的删除操作。哈希表的大小为 10，其中 h 1 .k/D k mod 10。（a）按顺序插入键 74、43、93、18、82、38、92 后的哈希表。（b）从槽 3 中删除键 43 后的哈希表。键 93 向上移动到槽 3 以保持可访问，然后键 92 向上移动到键 93 刚腾出的槽 5。无需移动其他键。

搜索继续，查看是否任何后续的键也需要移动到 k ′ 移动时刚释放的插槽 q ′ 中。

线性探测哈希删除 .T; q/

```
1  while TRUE
2      T[q] = NIL                      // make slot q empty
3      q' = q                          // starting point for search
4      repeat
5          q' = (q' + 1) mod m         // next slot number with linear probing
6          k' = T[q']                  // next key to try to move
7          if k' == NIL
8              return                  // return when an empty slot is found
9      until g(k', q) < g(k', q')      // was empty slot q probed before q'?
10     T[q] = k'                       // move k' into slot q
11     q = q'                          // free up slot q'
```

线性探测分析

线性探测是一种常见的实现方式，但它表现出一种称为 ***primary clustering*** 的现象。长时间占用的时隙会累积起来，从而增加平均

搜索时间。集群的出现是因为前面有 i 个满槽的空槽以 .i C 1//m 的概率被填满。占用槽的长期运行往往会变得更长，平均搜索时间也会增加。

在标准 RAM 模型中，主聚类是一个问题，而一般的双重散列通常比线性探测效果更好。相比之下，在分层内存模型中，主聚类是一个有益的特性，因为元素通常一起存储在同一个缓存块中。搜索先遍历一个缓存块，然后再继续搜索下一个缓存块。使用线性探测，HASH-INSERT、HASH-SEARCH 或 LINEAR-PROBING-HASH-DELETE 的键 k 的运行时间最多与 $h_1.k/$ 到下一个空槽的距离成正比。

以下定理由 Pagh 等人提出 [351]。Thorup [438] 给出了更近的证明。我们在此省略证明。5-独立性的必要性并不明显；请参阅引用的证明。

Theorem11.9

如果 h_1 是 5 独立的并且 $\alpha \# 2/3$，那么使用线性探测在哈希表中搜索、插入或删除键需要预期的常数时间。 ■

（实际上，对于 $\alpha D 1$ ，预期操作时间为 $O.1/\ ^2/$。）

?11.5.2 分层内存模型的哈希函数

本节介绍在具有内存层次结构的现代计算机系统中设计高效哈希表的方法。

由于内存层次结构的存在，线性探测是解决冲突的一个很好的选择，因为探测序列是连续的，并且倾向于停留在缓存块内。但是当哈希函数很复杂时（例如，定理 11.9 中的 5 独立函数），线性探测是最有效的。幸运的是，拥有内存层次结构意味着可以有效地实现复杂的哈希函数。

如第 11.3.5 节所述，一种方法是使用加密哈希函数，例如 SHA-256。此类函数对于哈希表应用程序来说很复杂且足够随机。在具有专门指令的机器上，加密函数可能非常高效。

相反，我们在此介绍一个仅基于加法、乘法和交换字的两半的简单哈希函数。此函数可以完全在快速寄存器中实现，并且在具有内存层次结构的机器上，其延迟与访问哈希表的随机槽所花费的时间相比很小。它与 RC 6 加密算法相关，并且可以在实际应用中被视为“随机预言机。”

wee 哈希函数

令 w 表示机器的字长（例如，w D 64），假设为偶数，令 a 和 b 为 w 位无符号（非负）整数，且 a 为奇数。令 swap.x/ 表示交换 w 位输入 x 的两个 w/2 位一半的 w 位结果。即，

$$\text{swap}(x) = (x \ggg (w/2)) + (x \lll (w/2))$$

其中 “o” 为 “逻辑右移” （如公式 (11.2) 所示），“n 为 “左移。” Deûne

$$f_a(k) = \text{swap}((2k^2 + ak) \bmod 2^w) .$$

因此，要计算 f $_a$ .k/，需求二次函数 2k ^{2} C ak modulo 2 ^{w}，然后交换结果的左半部分和右半部分。

令 r 表示计算哈希函数所需的“轮数”。我们将使用 r≥4，但对于任何非负 r，哈希函数都有很好的定义。用 $f_a^{(r)}$.k 表示从输入值 k 开始对 f_a 迭代 r 次（即 r 轮）的结果。对于任何奇数 a 和任何 r≥0，函数 $f_a^{(r)}$ 虽然很复杂，但却是一一对应的（参见练习 11.5-1）。密码学家会将 $f_a^{(r)}$ 视为对 w 位输入块进行操作的简单分组密码，具有 r 轮和密钥 a。

我们首先为短输入定义小哈希函数 h，其中 “短” 是指 “ 其长度 t 最多为 w 位，” 以便输入位于一个计算机字内。我们希望不同长度的输入以不同的方式进行哈希处理。对于 t 位输入 k，参数 a、b 和 r 的 ***wee hash function*** h $_{a,b,t,r}$.k/ 定义为

$$h_{a,b,t,r}(k) = \left(f_{a+2t}^{(r)}(k + b)\right) \bmod m . \tag{11.7}$$

也就是说，通过将 $f_{a+2t}^{(r)}$ 应用于 k C b，然后取最终结果模 m，可获得 t 位输入 k 的哈希值。添加值 b 可提供依赖于哈希的输入随机化，从而确保对于可变长度输入，0 长度输入没有固定的哈希值。将值 2t 添加到 a 可确保哈希函数对不同长度的输入采取不同的行为。（我们使用 2t 而不是 t 来确保如果 a 为奇数，则密钥 a C 2t 为奇数。）我们将此哈希函数称为 “wee”，因为它仅使用极少量的内存4更准确地说，它可以仅使用计算机的快速寄存器高效实现。（此哈希函数在文献中没有名称；它是我们为这本教科书开发的一个变体。）

wee 哈希函数的速度

令人惊讶的是，局部性可以带来如此高的效率。实验（作者未发表）表明，评估小哈希函数所需的时间更少

比探测哈希表中随机选择的 *single* 槽位更省时。这些实验是在一台笔记本电脑（2019 MacBook Pro）上运行的，其 w D 为 64，d 为 123。对于大型哈希表，评估小哈希函数比对哈希表执行单次探测快 2 到 10 倍。

可变长度输入的 wee 哈希函数 s

有时输入很长（长度超过一个 w 位字）或长度可变，如第 11.3.5 节所述。我们可以扩展上面定义的小哈希函数，用于长度最多为一个 w 位字的输入，以处理长输入或长度可变的输入。下面是实现此目的的一种方法。

假设输入 k 的长度为 t（以位为单位）。将 k 拆分为 w 位字序列 hk $_1$; k $_2$; : : : ; k $_u$ i，其中 u D dt /we，k $_1$ 包含 k 的最低有效位 w 位，k $_u$ 包含最高有效位。如果 t 不是 w 的倍数，则 k $_u$ 包含的位少于 w 位，在这种情况下，用 0 位填充 k $_u$ 中未使用的高位。定义函数 chop 以返回 k 中的 w 位字序列：

$$\text{chop}(k) = \langle k_1, k_2, \ldots, k_u \rangle .$$

chop 操作最重要的特性是，给定 t 时，它是一对一的：对于任何两个 t 位密钥 k 和 k ′，如果 k ¤ k ′ 则 chop.k/ ¤ chop.k ′/，并且 k 可以从 chop.k/ 和 t 中得出。 chop 操作还有一个有用的特性，即单字输入密钥会产生单字输出序列：chop.k/ D hki。

有了 chop 函数之后，我们可以为长度为 t 位的输入 k 指定 wee 哈希函数 h $_{a,b,t,r}$ .k/，如下所示：

$$h_{a,b,t,r}(k) = \textsc{Wee}(k, a, b, t, r, m) ,$$

其中，上页定义的程序 WEE 遍历 chop.k/ 返回的 w 位字的元素，将 f $_a^r$ 应用于当前字 k $_i$ 与之前计算的哈希值之和，最后返回模 m 得到的结果。这个针对可变长度和长（多字）输入的定义是方程 (11.7) 中针对短（单字）输入的定义的一个一致扩展。在实际使用中，我们建议 a 是随机选择的奇数 w 位字，b 是随机选择的 w 位字，r D 4。

请注意，wee 哈希函数实际上是一个哈希函数族，其中各个哈希函数由参数 a、b、t、r 和 m 决定。wee 哈希函数族对于可变长度输入的 (近似) 5 独立性可以通过假设 1 字 wee 哈希函数是随机预言机以及密码分组链接消息认证码 (CBC-MAC) 的安全性来论证，如 Bellare 等人所研究的 [42]。这里的情况实际上比文献中研究的要简单，因为如果两个消息的长度分别为 t 和 t ′，则它们的 “ 密钥 ” 也不同：a C 2t ¤ a C 2t ′。我们省略细节。

WEE.k; a; b; t; r; m/

1 $u = \lceil t/w \rceil$
2 $\langle k_1, k_2, \ldots, k_u \rangle = \text{chop}(k)$
3 $q = b$
4 **for** $i = 1$ **to** u
5 $\quad q = f^{(r)}_{a+2t}(k_i + q)$
6 **return** $q \bmod m$

这个受密码启发的哈希函数系列的定义旨在切合实际，但仅供参考，还有许多变化和改进的可能性。请参阅章节注释以获取建议。

总之，我们看到，当内存系统是分层的时，使用线性探测（双重哈希的特殊情况）会变得有利，因为连续探测往往停留在同一个缓存块中。此外，仅使用计算机的快速寄存器即可实现的哈希函数非常高效，因此它们可以非常复杂，甚至受到密码学的启发，从而提供线性探测最高效工作所需的高度独立性。

练习

11.5-1

完成论证：对于任何奇数正整数 a 和任何整数 r 及其和 0，函数 $f_a^{(r)}$ 是一一对应的。使用反证法证明，并利用函数 f $_a$ 模 2 ^{w} 成立的事实。

11.5-2

论证说随机预言是 5 独立的。

11.5-3

考虑一下，当我们将输入值 k 的单个位 k $_i$ 转换为不同的 r 值时，值 $f_a^{(r)}$ .k/ 会发生什么情况。设 k D P $_{i=0}^{w-1}$ k $_i$ 2 ^{i} 和 g $_a$ .k/ D P $_{j=0}^{w-1}$ b $_j$ 2 $_j$ 定义输入中的位值 k $_i$ （k $_0$ 为最低有效位）和 g $_a$ .k/ D .2k ^{2} C ak/ mod 2 ^{w} (中的位值 b $_j$ ，其中 g $_a$ .k/ 是当其两半交换后变为 f $_a$ .k/) 的值。假设对 j ∈ i 来说，除以输入 k 的单个位 k $_i$ 可能导致 g $_a$ .k/ 的任意位 b $_j$ 为 üip 。除以任意单个位 k $_i$ 的值可能导致输出 $f_a^{(r)}$ .k/ 的 *any* 位为 üip 的 r 的最小值是多少？解释一下。

问题

11-1Longest-probe bound for hashing
假设您正在使用大小为 m 的开放地址哈希表来存储 n m/2 个项目。

a.假设独立均匀排列散列，表明对于 i D 1; 2; : : : ; n，第 i 次插入严格需要超过 p 次探测的概率最多为 2^{-p}。

b.证明对于 i D 1; 2; : : : ; n，第 i 次插入需要超过 2 lg n 次探测的概率为 O.1/ n^2/。

让随机变量 X_i 表示第 i 次插入所需的探针数量。您已在部分 (b) 中表明 Pr f X_i > 2 lg ng D O.1/n^2/。让随机变量 X D max fX $_i$ W 1 i ng 表示 n 次插入中所需的最大探针数量。

c.证明 Pr fX > 2 lg ng D O.1/n/。

d.证明最长探测序列的预期长度 E ŒX 为 O.lg n/。

11-2Searching a static set
要求您实现一个可搜索的 n 个元素集合，其中的键是数字。该集合是静态的（没有 INSERT 或 DELETE 操作），唯一需要的操作是 SEARCH。您可以随意预处理这 n 个元素，以便快速运行 SEARCH 操作。

a.说明如何在最坏情况下不使用超出存储集合元素所需的额外存储空间，仅使用 O.lg n/ 的存储空间来实现搜索。

b.考虑通过 m 个槽上的开放地址散列实现集合，并假设独立的均匀排列散列。要使不成功的 SEARCH 操作的平均性能至少与部分 (a) 中的界限一样好，所需的最小额外存储量 m n 是多少？您的答案应该是关于 n 的 m 的渐近界限。

11-3Slot-size bound for chaining
给定一个有 n 个槽的哈希表，通过链接解决冲突，假设有 n 个键插入到表中。每个键被哈希到每个槽的可能性相等。让 M 表示所有键都已哈希到任何槽中的最大键数

已插入。您的任务是证明 E ŒM （M 的期望值）的 O.lg n/ lg lg n/ 上限。

a.假设恰好有 k 个密钥散列到特定槽的概率 Q $_k$ 为

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} .$$

b.令 P $_k$ 为 M D k 的概率，也就是包含最多密钥的槽包含 k 个密钥的概率。证明 P $_k$ nQ $_k$。

c.证明 Q $_k$ < e ^{k}/k ^{k}. *Hint:* 使用第 67 页的斯特林近似值，公式 (3.25)。

d.证明存在常数 c > 1，使得对于 k $_0$ D c lg n/ lg lg n，Q $_{k_0}$ < 1/n ^{3}。得出对于 k k $_0$ D c lg n/ lg lg n，P $_k$ < 1/n ^{2}。

e.认为

$$\mathrm{E}[M] \le \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \le \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n} .$$

Conclude that $\mathrm{E}[M] = O(\lg n / \lg \lg n)$.

11-4Hashing and authentication

令 H 为一组哈希函数，其中每个哈希函数 h 2 H 将密钥集合 U 映射到 f0; 1; : : : ; m 1g。

a.证明如果哈希函数族 H 是 2 独立的，那么它是通用的。

b.假设宇宙 U 是从 Z $_p$ D f0; 1; : : : ; p 1g 中抽取的 n 元组集合，其中 p 为素数。考虑元素 x D hx $_0$; x $_1$; : : : ; x $_{n-1}$ i 2 U 。对于任何 n 元组 a D ha $_0$; a $_1$; : : : ; a $_{n-1}$ i 2 U，定义哈希函数 h $_a$ 为

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j\right) \bmod p .$$

令 H D fh $_a$ W a 2 U g。证明 H 是通用的，但不是 2 独立的。（*Hint:* 找到一个密钥，使 H 中的所有哈希函数都产生相同的值。）

c.假设我们对 (b) 部分中的 H 进行稍微修改：对于任何 a 2 U 和任何 b 2 Z $_p$ ，deûne

$$h'_{ab}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

且 H ′ D fh $'_{ab}$ W a 2 U 和 b 2 Z $_p$ g。论证 H ′ 是 2 独立的。（*Hint:* 考虑固定 n 元组 x 2 U 和 y 2 U ，其中 x $_i$ ¤ y $_i$ 为某个 i 。h $'_{ab}$ .x/ 和 h $'_{ab}$ .y/ 作为 a $_i$ 和 b 在 Z $_p$ 上的范围会发生什么情况？）

d.Alice 和 Bob 秘密地商定了一个来自 2 独立的哈希函数系列 H 的哈希函数 h。每个 h 2 H 都从密钥宇宙 U 映射到 Z $_p$，其中 p 为素数。后来，Alice 通过互联网向 Bob 发送一条消息 m，其中 m 2 U。她通过发送一个身份验证标签 t D h.m/ 来向 Bob 验证这条消息的真实性，而 Bob 检查他收到的对 .m;t/ 确实满足 t D h.m/。假设一个对手在途中截获了 .m;t/，并试图通过将对 .m;t/ 替换为不同的对 .m ′;t ′/ 来欺骗 Bob。论证对手成功欺骗 Bob 接受 .m ′; t′/ 最多为 1/p，无论对手拥有多强的计算能力，即使对手知道所使用的哈希函数系列 H。

章节注释

Knuth [261] 和 Gonnet 和 Baeza-Yates [193] 的书籍是分析哈希算法的绝佳参考。Knuth 认为 H. P. Luhn (1953) 发明了哈希表，以及解决冲突的链接方法。大约在同一时间，G. M. Amdahl 提出了开放寻址的想法。随机预言的概念由 Bellare 等人提出。[43]。Carter 和 Wegman [80] 于 1979 年引入了哈希函数通用系列的概念。

Dietzfelbinger 等人 [113] 发明了乘法移位哈希函数，并给出了定理 11.5 的证明。Thorup [437] 提供了扩展和附加分析。Thorup [438] 给出了一个简单的证明，即使用 5 独立哈希的线性探测每次操作的预期时间为常数。Thorup 还描述了使用线性探测在哈希表中删除的方法。

Fredman、Koml ´os 和 Szemer´edi [154] 开发了一种针对静态集合的完美哈希方案4“，非常完美”，因为可以避免所有冲突。Dietzfelbinger 等人 [114] 给出了他们的方法对动态集合的扩展，以 O.1/ 的摊销预期时间处理插入和删除。wee 哈希函数基于 RC6 加密算法 [379]。Leiserson 等人 [292] 提出了一个 “RC 6MIX” 函数，该函数与

wee 哈希函数。他们给出了实验证据，证明该函数具有良好的随机性，并且还给出了一个 “DOTMIX” 函数来处理可变长度的输入。Bellare 等人 [42] 对密码块链接消息认证码的安全性进行了分析。该分析表明 wee 哈希函数具有所需的伪随机性。

12 Binary Search Trees

搜索树数据结构支持第 250 页列出的每一种动态集操作：搜索、最小值、最大值、前导、后续、插入和删除。因此，您既可以将搜索树用作字典，也可以将其用作优先级队列。

二叉搜索树的基本操作所花的时间与树的高度成正比。对于具有 n 个节点的完全二叉树，此类操作最坏情况下的运行时间为 ，.lg n/。但是，如果树是 n 个节点的线性链，则相同操作最坏情况下的运行时间为 ，.n/。在第 13 章中，我们将看到二叉搜索树的一种变体，即红黑树，其操作保证高度为 O.lg n/。我们不会在这里证明这一点，但是如果你在一组随机的 n 个键上构建二叉搜索树，即使你不尝试限制其高度，其预期高度也是 O.lg n/。

在介绍二叉搜索树的基本属性之后，以下各节将介绍如何遍历二叉搜索树以按排序顺序打印其值、如何在二叉搜索树中搜索值、如何查找最小或最大元素、如何查找元素的前驱或后继以及如何在二叉搜索树中插入或删除元素。树的基本数学属性见附录 B。

12.1 什么是二叉搜索树？

二叉搜索树，顾名思义，是按二叉树的形式组织的，如图 12.1 所示。你可以用链接的数据结构来表示这样的树，如第 10.3 节中所示。除了 *key* 和卫星数据之外，每个节点对象还包含属性 *left* 、*right* 和 p，分别指向其左子节点、右子节点和父节点对应的节点。如果缺少子节点或父节点，则相应的属性包含值 NIL。树本身具有属性 *root*

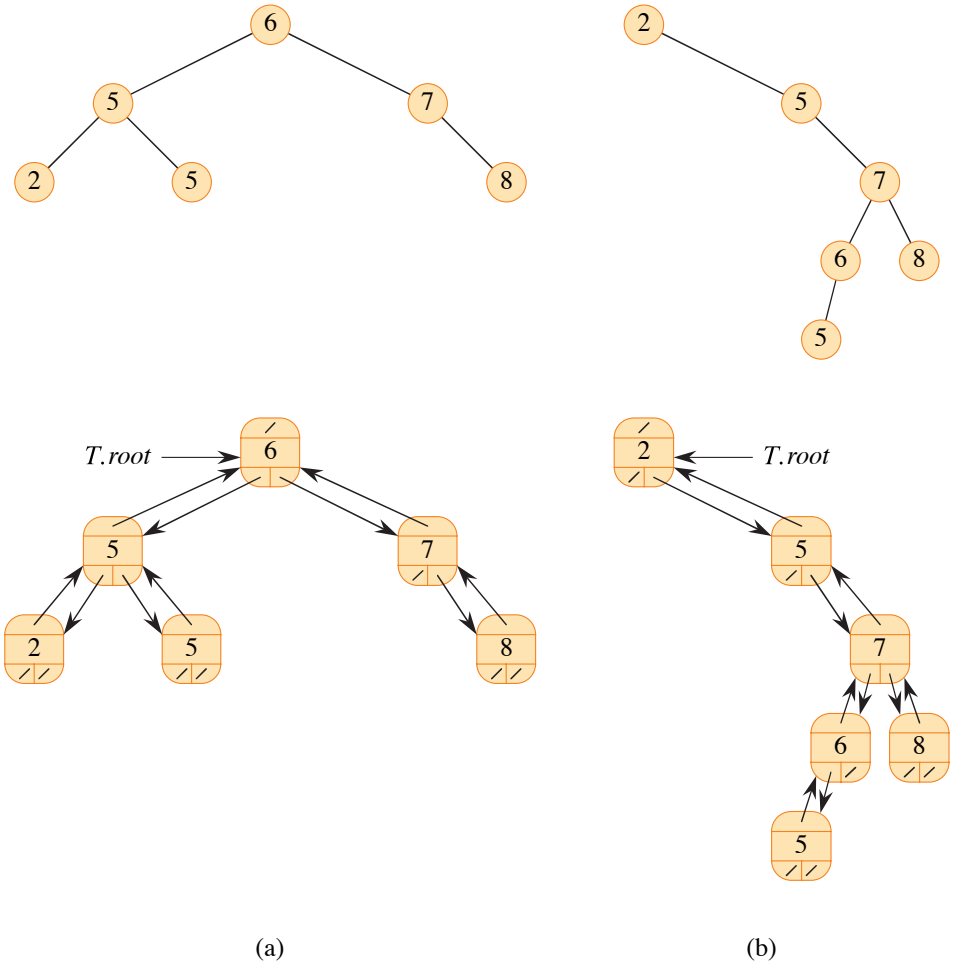


图 12.1 二叉搜索树。对于任何节点 x，x 的左子树中的键最多为 x：*key*，而 x 的右子树中的键至少为 x：*key*。不同的二叉搜索树可以表示同一组值。大多数搜索树操作的最坏情况运行时间与树的高度成正比。（a）一棵有 6 个节点、高度为 2 的二叉搜索树。上图显示了如何从概念上查看树，下图显示了每个节点中的 *left*、*right* 和 p 属性，其样式与第 266 页的图 10.6 相同。（b）一棵效率较低的二叉搜索树，高度为 4，包含相同的键。

指向根节点，如果树为空，则指向 NIL。根节点 T: *root* 是树 T 中父节点为 NIL 的唯一节点。

二叉搜索树中的键总是以满足***binary-search-tree property***的方式存储：

设 x 是二叉搜索树中的一个节点。如果 y 是 x 左子树中的一个节点，则 y：*key* # x：*key*。如果 y 是 x 右子树中的一个节点，则 y：*key* x：*key*。

因此，在图 12.1(a) 中，根节点的键为 6，其左子树中的键 2、5 和 5 不大于 6，其右子树中的键 7 和 8 不小于 6。树中的每个节点都具有同样的属性。例如，将根节点的左子节点视为子树的根，该子树根具有键 5，其左子树中的键 2 不大于 5，其右子树中的键 5 不小于 5。

由于二叉搜索树的特性，你可以用一个简单的递归算法按排序顺序打印出二叉搜索树中的所有键，该算法称为 ***inorder tree walk***，由过程 I NORDER-TREE-WALK 给出。该算法之所以如此命名，是因为它在打印左子树的值和打印右子树的值之间打印子树根的键。（类似地，***preorder tree walk*** 在任一子树的值之前打印根，而 ***postorder tree walk*** 在其子树的值之后打印根。）要打印二叉搜索树 T 中的所有元素，请调用 I NORDER-TREE-WALK .T: *root* /。例如，中序树遍历按 2; 5; 5; 6; 7; 的顺序打印图 12.1 中的两个二叉搜索树中每一个中的键。 8. 算法的正确性可以通过二叉搜索树性质直接归纳得出。

```
我 NORDER-TREE-WALK .x/

1 如果 x ¤ NIL 2 我 NORDER-TREE-W
ALK .x: left/ 3 打印 x: key 4 我 NORDE
R-TREE-WALK .x: right /
```

遍历一棵 n 节点二叉搜索树需要 ，.n/ 时间，因为在初始调用之后，该过程对树中的每个节点递归调用自身两次4，一次针对其左子节点，一次针对其右子节点。以下定理给出了执行中序树遍历需要线性时间的正式证明。

Theorem12.1
如果 x 是 n 节点子树的根，则调用 I NORDER-TREE-WALK .x/ 需要 ，.n/ 时间。

Proof令 T .n/ 表示在 n 节点子树的根上调用 I NORDER-TREE-WALK 所花费的时间。由于 I NORDER-TREE-WALK 访问子树的所有 n 个节点，因此我们有 T .n/ D .n/。剩下要证明的是 T .n/ D O.n/。

由于 I NORDER-TREE-WALK 在空子树上（对于测试 x ¤ NIL）花费的时间很短，为常数，因此我们有 T .0/D c，其中 c > 0 为常数。

对于 n > 0，假设在节点 x 上调用 I NORDER-TREE-WALK，该节点的左子树有 k 个节点，右子树有 n k 1 个节点。执行 I NORDER-TREE-WALK .x / 的时间限制为 T .n/ T .k/CT .n 1/Cd，其中常数 d > 0 反映了执行 I NORDER-TREE-WALK .x/ 主体的时间上限，不包括递归调用所花费的时间。

我们使用代换法证明 T .n/ D O.n/，即 T .n/ = .c C d /n C c 。对于 n D 0，我们有 .c C d / 0 C c D c D T .0/。对于 n > 0，我们有

$$\begin{aligned} T(n) &\le T(k) + T(n-k-1) + d \\ &\le ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c \ , \end{aligned}$$

从而完成证明。 ■

练习

12.1-1

对于键集合 f1；4；5；10；16；17；21g，绘制高度为 2、3、4、5 和 6 的二叉搜索树。

12.1-2

第 163 页上的二叉搜索树性质和最小堆性质有什么区别？最小堆性质能否在 O.n/ 时间内按排序顺序打印出 n 节点树的键？说明如何打印，或者解释为什么不可以打印。

12.1-3

给出一个执行中序树遍历的非递归算法。（*Hint:* 一个简单的解决方案使用堆栈作为辅助数据结构。一个更复杂但更优雅的解决方案不使用堆栈，但假设你可以测试两个指针是否相等。）

12.1-4

给出在 n 个节点的树上以 ，n/ 时间执行前序和后序树遍历的递归算法。

12.1-5

论证说，由于在比较模型中，对 n 个元素进行排序在最坏情况下需要 .n lg n/ 时间，因此，任何基于比较的算法，用于从任意 n 个元素的列表构造二叉搜索树，在最坏情况下都需要 .n lg n/ 时间。

12.2 查询二叉搜索树

二叉搜索树可以支持查询 MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR，以及 SEARCH。本节将介绍这些操作，并展示如何在高度为 h 的任意二叉搜索树上以 0.h/ 的时间为单位支持每一项操作。

搜索

要在二叉搜索树中搜索具有给定键的节点，请调用 TREE-SEARCH 过程。给定指向子树根的指针 x 和键 k，如果子树中存在具有键 k 的节点，则 TREE-SEARCH .x; k/ 返回指向该节点的指针；否则，返回 NIL。要在整个二叉搜索树 T 中搜索键 k，请调用 TREE-SEARCH .T: *root* ; k/。

```
树搜索 .x; k/

1 如果 x == NIL 或 k == x: key 2 返回 x 3
如果 k < x: key 4 返回 TREE-SEARCH.x:
left; k/ 5 否则返回 TREE-SEARCH.x:
right ; k/

我迭代树搜索.x; k/

1 当 x ¤ NIL 且 k ¤ x: key 2 如果
k < x: key 3 x D x: left  4 否则 x
D x: right  5 返回 x
```

TREE-SEARCH 过程从根节点开始搜索，然后在树中向下追踪一条简单路径，如图 12.2(a) 所示。对于遇到的每个节点 x，它将键 k 与 x: *key* 进行比较。如果两个键相等，则搜索终止。如果 k 小于 x: *key*，则继续在 x 的左子树中搜索，因为二叉搜索树属性意味着 k 不能位于右子树中。对称地，如果 k 大于 x: *key*，则继续在右子树中搜索。递归期间遇到的节点从树的根节点向下形成一条简单路径，因此 TREE-SEARCH 的运行时间为 O.h/，其中 h 是树的高度。

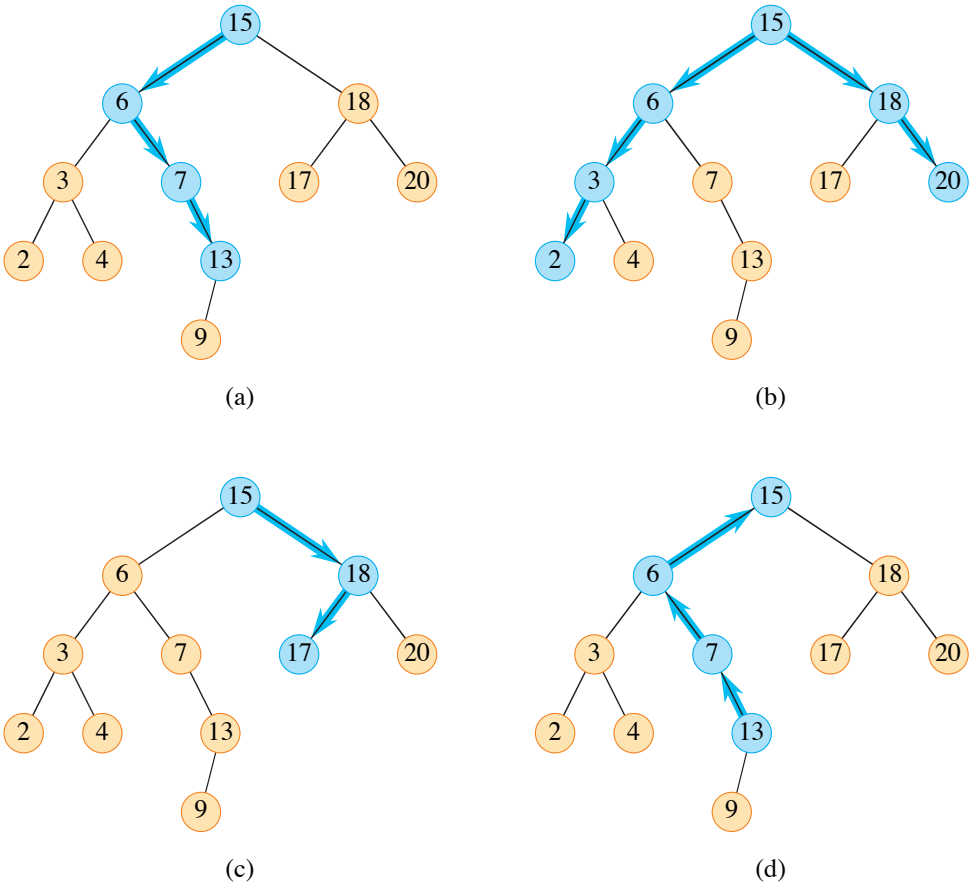


图 12.2 二叉搜索树上的查询。每个查询中遵循的节点和路径都标为蓝色。（a）在树中搜索键 13 时，将从根开始沿着路径 15 ! 6 ! 7 ! 13 进行搜索。（b）树中的最小键为 2，可通过从根开始的 *left* 指针找到。最大键 20 可通过从根开始的 *right* 指针找到。（c）键 15 节点的后继是键 17 节点，因为它是 15 的右子树中的最小键。（d）键 13 节点没有右子树，因此其后继是其最低祖先，而该祖先的左子节点也是祖先。在这种情况下，键 15 节点是其后继。

由于 TREE-SEARCH 过程要么在左子树上递归，要么在右子树上递归，但不能同时在两个子树上递归，因此我们可以重写该算法，将递归展开为一个 while 循环。在大多数计算机上，上页中的 I TERATIVE-TREE-SEARCH 过程效率更高。

最小值和最大值

要在二叉搜索树中找到键最小的元素，只需从根节点开始跟踪 *left* 子指针，直到遇到 NIL，如图 12.2(b) 所示。

TREE-MINIMUM 过程返回指向以给定节点 x 为根的子树中的最小元素的指针，我们假设该节点非 NIL。

树-最小值.x/

1 当 x: *left* ¤ NIL 2 x
D x: *left* 3 返回 x

树-最大值.x/

1 当 x: *right* ¤ NIL 2 x
D x: *right* 3 返回 x

二叉搜索树属性保证 T REE-MINIMUM 是正确的。如果节点 x 没有左子树，那么由于 x 的右子树中的每个键至少与 x: *key* 一样大，因此以 x 为根的子树中的最小键为 x: *key*。如果节点 x 有左子树，那么由于右子树中没有小于 x: *key* 的键，并且左子树中的每个键都不大于 x: *key*，因此以 x 为根的子树中的最小键位于以 x: *left* 为根的子树中。

TREE-MAXIMUM 的伪代码是对称的。TREE-MINIMUM 和 TREE-MAXIMUM 都在高度为 h 的树上运行 O.h/ 时间，因为与 TREE-SEARCH 一样，遇到的节点序列从根向下形成一条简单路径。

继任者和前任

给定二叉搜索树中的一个节点，如何按照中序树遍历确定的排序顺序找到它的后继？如果所有键都是不同的，则节点 x 的后继是具有大于 x 的最小键的节点：*key*。无论键是否不同，我们都将节点的 ***successor*** 定义为中序树遍历中访问的下一个节点。二叉搜索树的结构允许您在不比较键的情况下确定节点的后继。上页中的 T REE-SUCCESSOR 过程返回二叉搜索树中节点 x 的后继（如果存在）或 NIL（如果 x 是在中序遍历期间访问的最后一个节点）。

TREE-SUCCESSOR 的代码有两种情况。如果节点 x 的右子树非空，则 x 的后继就是 x 的右子树中最左边的节点，第 2 行通过调用 TREE-MINIMUM. x: *right* / 找到该节点。例如，图 12.2(c) 中键为 15 的节点的后继就是键为 17 的节点。另一方面，正如练习 12.2-6 要求你证明的那样，如果节点 x 的右子树为空，且 x 有一个后继节点 y，那么 y 就是 x 的最低祖先，其

树继任者.x/

1 if x: *right* ¤ NIL 2 return TREE-MINIMUM.x: *right* / **//** 右子树中最左节点 3 else **//** 找到 x 的最低祖先，其左孩子是 x 的祖先 4 y D x: *p* 5 while y ¤ NIL and x ==y: *right* 6 x D y 7 y D y: *p* 8 return y

左子节点也是 x 的祖先。在图 12.2(d) 中，键为 13 的节点的后继节点是键为 15 的节点。要找到 y，从 x 开始沿树向上走，直到遇到根节点或其父节点的左子节点。TREE-SUCCESSOR 的第 438 行处理这种情况。

在高度为 h 的树上，TREE-SUCCESSOR 的运行时间为 O.h/，因为它要么沿着树上的简单路径向上，要么沿着树上的简单路径向下。与 TREE-SUCCESSOR 对称的程序 TREE-PREDECESSOR 的运行时间也是 O.h/。

总而言之，我们证明了以下定理。

Theorem12.2
可以实现动态集操作 SEARCH、MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR，使得每个操作在高度为 h 的二叉搜索树上以 O.h/ 的时间运行。 ■

练习

12.2-1
您正在包含 1 到 1000 之间的数字的二叉搜索树中搜索数字 363。以下哪个序列 *cannot* 是检查的节点序列？

a.2、252、401、398、330、344、397、363。

b.924、220、911、244、898、258、362、363。***c.*** 925、202、911、240、912、245、363。***d.***2、399、387、219、266、382、381、278、363。***e.***935、278、347、621、299、392、358、363。

12.2-2编写TREE-MINIMUM 和TREE-MAXIMUM 的递归版本。

12.2-3
编写TREE-PREDECESSOR过程。

12.2-4
基尔默教授声称发现了二叉搜索树的一个显著特性。假设在二叉搜索树中搜索键 k 最终到达一个叶子节点。考虑三个集合：A，搜索路径左侧的键；B，搜索路径上的键；C，搜索路径右侧的键。基尔默教授声称，任何三个键 a 2 A、b 2 B 和 c 2 C 都必须满足 a b c 。请给出一个最小的反例来反驳教授的说法。

12.2-5
证明如果二叉搜索树中的一个节点有两个子节点，那么它的后继节点没有左子节点，而它的前任节点没有右子节点。

12.2-6
考虑一个二叉搜索树 T，它的键是不同的。说明如果 T 中节点 x 的右子树为空，且 x 有一个后继节点 y，则 y 是 x 的最低祖先，其左子节点也是 x 的祖先。（回想一下，每个节点都是它自己的祖先。）

12.2-7
执行 n 个节点二叉搜索树的中序树遍历的另一种方法是，通过调用 TREE-MINIMUM，然后对 TREE-SUCCESSOR 进行 n 1 次调用，找到树中的最小元素。证明此算法在 ，.n/ 时间内运行。

12.2-8
证明无论从高度为 h 的二叉搜索树的哪个节点开始，k 次连续调用 TREE-SUCCESSOR 都需要 O.k C h/ 次。

12.2-9
假设 T 是一棵二叉搜索树，其键值不同，x 是叶节点，y 是其父节点。证明 y: *key* 是 T 中大于 x: *key* 的最小键值，或者是 T 中小于 x: *key* 的最大键值。

12.3 插入和删除

插入和删除操作会导致二叉搜索树所表示的动态集发生变化。必须修改数据结构以反映这种变化，但必须以二叉搜索树属性继续保持的方式进行。我们将看到，修改树以插入新元素相对简单，但从二叉搜索树中删除节点则更为复杂。

插入

TREE-INSERT 过程将新节点插入二叉搜索树。该过程采用二叉搜索树 T 和节点 z，其中 $z.key$ 已填充，$z.left$ = NIL，以及 $z.right$ = NIL。它修改 T 和 z 的一些属性，以便将 z 插入树中的适当位置。

```
树插入 .T; z/
 1 x = T.root                    // 节点与 z 进行比较
 2 y = NIL                       // y 将成为 z 的父节点
 3 while x ≠ NIL                 // 下降直到到达叶子节点
 4     y = x
 5     if z.key < x.key
 6         x = x.left
 7     else x = x.right
 8 z.p = y                       // 找到带有父节点 y 的位置
 4 插入 z
 9 if y == NIL
10     T.root = z                // 树 T 为空
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z               类型 z
```

图 12.3 显示了 TREE-INSERT 的工作原理。与过程 TREE-SEARCH 和 ITERATIVE-TREE-SEARCH 一样，TREE-INSERT 从树的根开始，指针 x 沿着一条简单的路径向下寻找一个 NIL 来替换输入节点 z。该过程将 ***trailing pointer*** y 维护为 x 的父节点。初始化后，第 337 行中的 while 循环使这两个指针沿树向下移动，根据 $z.key$ 与 x.key 的比较向左或向右移动，直到 x 变为 NIL。这个 NIL 占据了节点 z 将要去的位置。更准确地说，这个 NIL 是将成为 z 父节点的节点的 $left$ 或 $right$ 属性，或者如果树 T 当前为空，则它是 T.$root$。该过程需要

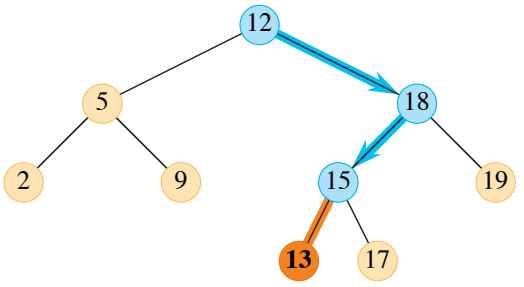


图 12.3 将键为 13 的节点插入二叉搜索树。从根节点到节点插入位置的简单路径以蓝色显示。新节点及其父节点的链接以橙色突出显示。

尾随指针 y，因为当它找到 z所属的 NIL 时，搜索已经超出需要更改的节点一步。第 8313 行设置了导致插入 z的指针。

与搜索树上的其他原始操作一样，过程 T REE-INSERT 在高度为 h 的树上运行时间为 O.h/。

删除

从二叉搜索树 T 中删除节点 z的总体策略有三种基本情况，正如我们将看到的，其中一种情况有点棘手。

如果 z没有子节点，则只需修改其父节点，用 NIL 替换 z作为其子节点，即可将其删除。• 如果 z只有一个子节点，则修改 z的父节点，用 z的子节点替换 z，从而提升该子节点，使其取代 z在树中的位置。• 如果 z有两个子节点，则找到 z的后继 y 4（它必须属于 z的右子树4），并将 y 移动到 z在树中的位置。z原始右子树的其余部分将成为 y 的新右子树，而 z的左子树将成为 y 的新左子树。因为 y 是 z的后继，所以它不能有左孩子，而 y 的原始右孩子会移到 y 的原始位置，y 的原始右子树的其余部分也会自动跟随。这种情况比较棘手，因为我们将会看到，y 是否是 z的右孩子很重要。

从二叉搜索树 T 中删除给定节点 z的过程以指向 T 和 z的指针作为参数。它通过考虑图 12.4 中所示的四种情况来组织其情况，这与之前概述的三种情况略有不同。

如果 z没有左孩子，那么如图 (a) 部分所示，用其右孩子替换 z，该右孩子可能是也可能不是 NIL。当 z的右孩子为 NIL 时，这种情况处理

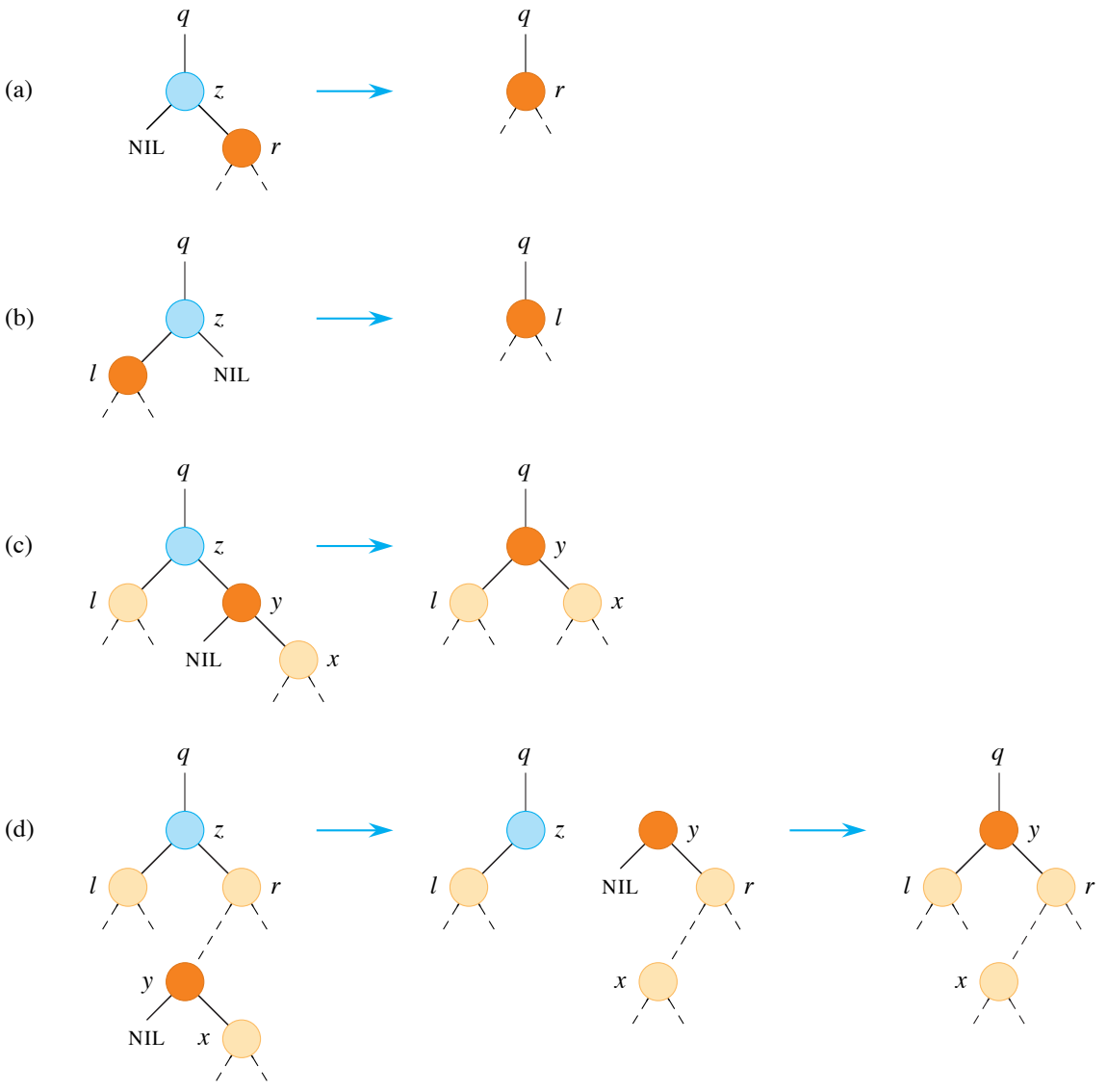


图 12.4 从二叉搜索树中删除节点 z（蓝色）。节点 z 可能是根、节点 q 的左孩子或 q 的右孩子。将在树中取代节点 z 位置的节点被标记为橙色。（a）节点 z 没有左孩子。用它的右孩子 r 替换 z，后者可能是也可能不是 NIL。（b）节点 z 有一个左孩子 l 但没有右孩子。用 l 替换 z。（c）节点 z 有两个孩子。它的左孩子是节点 l，它的右孩子是其后继 y（没有左孩子），y 的右孩子是节点 x。用 y 替换 z，将 y 的左孩子更新为 l，但保留 x 作为 y 的右孩子。（d）节点 z 有两个子节点（左子节点 l 和右子节点 r），其后继节点 y ¤ r 位于以 r 为根的子树中。首先用其自己的右子节点 x 替换 y，并将 y 设置为 r 的父亲。然后将 y 设置为 q 的孩子和 l 的父亲。

z没有子节点的情况。当 z的右子节点非 NIL 时，此情况处理 z只有一个子节点（即其右子节点）的情况。

否则，如果 z只有一个孩子，那么该孩子就是左孩子。如图 (b) 部分所示，用其左孩子替换 z。• 否则，z既有左孩子也有右孩子。找到 z的后继 y，它位于 z的右子树中，并且没有左孩子（参见练习 12.2-5）。将节点 y 从其当前位置拼接出来，并在树中用 y 替换 z。如何执行此操作取决于 y 是否是 z的右孩子：

B 如果 y 是 z的右子节点，则如图 (c) 部分所示，用 y 替换 z，保留 y 的右子节点不变。◦ 否则，y 位于 z的右子树中，但不是 z的右子节点。在这种情况下，如图 (d) 部分所示，首先用其自己的右子节点替换 y，然后用 y 替换 z。

作为删除节点过程的一部分，子树需要在二叉搜索树内移动。子程序 TRANSPLANT 将一棵子树替换为其父节点的子树。当 TRANSPLANT 将以节点 u 为根的子树替换为以节点 v 为根的子树时，节点 u 的父节点将成为节点 v 的父节点，而 u 的父节点最终将 v 作为其适当的子节点。TRANSPLANT 允许 v 为 NIL，而不是指向节点的指针。

```
移植 .T; u; v/

1 if u: p == NIL 2 T:
root  D v 3 elseif u ==
u: p: left  4 u: p: left  D
v 5 else u: p: right  D v
6 if v ¤ NIL 7 v: p  D u
: p
```

TRANSPLANT 的工作方式如下。第 132 行处理 u 是 T 的根的情况。否则，u 是其父级的左子级或右子级。第 334 行负责更新 u：*p*：*left*（如果 u 是左子级），第 5 行负责更新 u：*p*：*right*（如果 u 是右子级）。因为 v 可能为 NIL，所以第 637 行仅当 v 非 NIL 时才更新 v：*p*。过程 TRANSPLANT 不会尝试更新 v：*left* 和 v：*right*。是否更新是 TRANSPLANT 调用者的责任。上页中的 TREE-DELETE 过程使用 TRANSPLANT 从二叉搜索树 T 中删除节点 z。它执行以下四种情况。第 132 行处理节点 z没有左子节点的情况（图 12.4(a)），第 334 行

处理 z有左孩子但没有右孩子的情况（图 12.4(b)）。第 5312 行处理余下的两种情况，即 z有两个孩子。第 5 行找到节点 y，它是 z的后继。因为 z有一个非空的右子树，所以它的后继必须是该子树中具有最小键的节点；因此调用 TREE-MINIMUM.z*right* /。如前所述，y 没有左孩子。该过程需要将 y 从其当前位置拼接出来，并在树中用 y 替换 z. 如果 y 是 z的右孩子（图 12.4(c)），则第 10312 行用 y 替换 z（作为其父节点的孩子），并用 z的左孩子替换 y 的左孩子。节点 y 保留其右子节点（图 12.4(c) 中的 x），因此 y: *right* 无需发生任何变化。如果 y 不是 z的右子节点（图 12.4(d)），则必须移动两个节点。第 739 行用 y 的右子节点（图 12.4(c) 中的 x）替换 y 作为其父节点的子节点，并使 z的右子节点（图中的 r）成为 y 的右子节点。最后，第 10312 行用 y 替换 z作为其父节点的子节点，并用 z的左子节点替换 y 的左子节点。

树删除 .T; z

```
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)          // replace z by its right child
3  elseif z.right == NIL
4      TRANSPLANT(T, z, z.left)           // replace z by its left child
5  else y = TREE-MINIMUM(z.right)         // y is z's successor
6      if y ≠ z.right                     // is y farther down the tree?
7          TRANSPLANT(T, y, y.right)      // replace y by its right child
8          y.right = z.right              // z's right child becomes
9          y.right.p = y                  //       y's right child
10     TRANSPLANT(T, z, y)                // replace z by its successor y
11     y.left = z.left                    // and give z's left child to y,
12     y.left.p = y                       //       which had no left child
```

TREE-DELETE 的每一行，包括对 TRANSPLANT 的调用，都需要恒定的时间，除了第 5 行对 TREE-MINIMUM 的调用。因此，TREE-DELETE 在高度为 h 的树上运行时间为 O.h/。

总而言之，我们证明了以下定理。

Theorem12.3

可以实现动态集操作 I NSERT 和 DELETE，使得每个操作在高度为 h 的二叉搜索树上以 O.h/ 时间运行。 ■

练习

12.3-1给出 TREE-I NSERT 过程的递归版本。

12.3-2
假设你通过反复向树中插入不同的值来构建二叉搜索树。假设在树中搜索某个值时检查的节点数是 1 加上首次将该值插入树时检查的节点数。

12.3-3
你可以对给定的一组 n 个数字进行排序，方法是首先构建一个包含这些数字的二叉搜索树（重复使用 TREE-INSERT 逐个插入数字），然后通过中序树遍历打印数字。此排序算法的最坏情况和最佳情况的运行时间是多少？

12.3-4
当TREE-DELETE调用TRANSPLANT时，什么情况下TRANSPLANT的参数v可以为NIL？

12.3-5
删除操作 “ 可交换” 吗？即从二叉搜索树中删除 x 然后删除 y 和先删除 y 然后删除 x 得到的树是一样的。请说明原因或给出反例。

12.3-6
假设每个节点 x 不保留属性 x: *p*（指向 x 的父节点），而是保留 x: *succ*（指向 x 的后继节点）。给出使用此表示法在二叉搜索树 T 上执行 TREE-SEARCH、TREE-INSERT 和 TREE-DELETE 的伪代码。这些过程应在 O.h/ 时间内运行，其中 h 是树 T 的高度。您可以假设二叉搜索树中的所有键都是不同的。（*Hint:* 您可能希望实现一个返回节点父节点的子例程。）

12.3-7
当 TREE-DELETE 中的节点 *z*有两个子节点时，您可以选择节点 y 作为其前任节点，而不是其后继节点。如果这样做，TREE-DELETE 还需要做哪些其他更改？有人认为，公平策略赋予前任节点和后继节点同等的优先级，可以产生更好的经验性能。如何对 TREE-DELETE 进行最小程度的更改以实现这种公平策略？

问题

12-1Binary search trees with equal keys
相等的键对二叉搜索树的实现提出了一个问题。

a.当将 n 个具有相同键的项目插入到最初为空的二叉搜索树中时，TREE-I N SERT 的渐近性能如何？

考虑更改 TREE-I NSERT 以在第 5 行之前测试是否 *z*:*key* D x: *key*，并在第 11 行之前测试是否 *z*:*key* D y: *key*。如果相等，则实施以下策略之一。对于每个策略，找到将 n 个具有相同键的项目插入最初为空的二叉搜索树的渐近性能。（第 5 行描述了这些策略，该行比较了 *z* 和 x 的键。用 y 代替 x 以得出第 11 行的策略。）

b.在节点 x 处保留一个布尔值标记 x: *b* ，并根据 x: *b* 的值将 x 设置为 x: *left* 或 x: *right* ，每次 TREE-INSERT 访问 x 并插入与 x 具有相同键的节点时，该值在 FALSE 和 TRUE 之间交替。***c.*** 在节点 x 处保留一个布尔值标记 x: *b* ，并根据 x: *b* 的值将 x 设置为 x: *left* 或 x: *right* 。每次 TREE-INSERT 访问 x 并插入与 x 具有相同键的节点时，该值在 FALSE 和 TRUE 之间交替。***d.*** 将 x 在 x 处具有相等键的节点的 st，并将 *z* 插入到第 e 列表。随机设置为 x: *left* 或 x: *right*。（给出最坏情况的性能并非正式地得出预期运行时间。）

12-2Radix trees
给定两个字符串 a D a $_0$a $_1$: : : a $_p$ 和 b D b $_0$b $_1$: : : b $_q$，其中每个 a $_i$ 和每个 b $_j$ 都属于某个有序字符集，如果满足以下任一条件，我们称字符串 a 为 ***lexicographically less than*** 字符串 b
1. 存在一个整数j，其中0≤j≤minfp;qg，使得对于所有的iD0;1;:::;j1，有a$_i$ Db $_i$和a$_j$ <b$_j$，或者
2. 对所有的 i D 0; 1; : : : ; p，p < q 和 a $_i$ D b $_i$。

例如，如果 a 和 b 是位串，则根据规则 1 为 10100 < 10110（设 j D 3），根据规则 2 为 10100 < 101000。此排序与英语词典中使用的排序类似。

图 12.5 中所示的 ***radix tree*** 数据结构（也称为 a ***trie***）存储位串 1011、10、011、100 和 0。在搜索键 a D a $_0$a $_1$: : : a $_p$ 时，如果 a $_i$ D 为 0，则从深度为 i 的节点向左移动，如果 a $_i$ D 为 1，则从深度为 i 的节点向右移动。设 S 为一组长度和为 n 的不同位串。说明如何使用基数树在 ，.n/ 时间内按字典顺序对 S 进行排序。对于图 12.5 中的示例，排序的输出应为序列 0、011、10、100、1011。

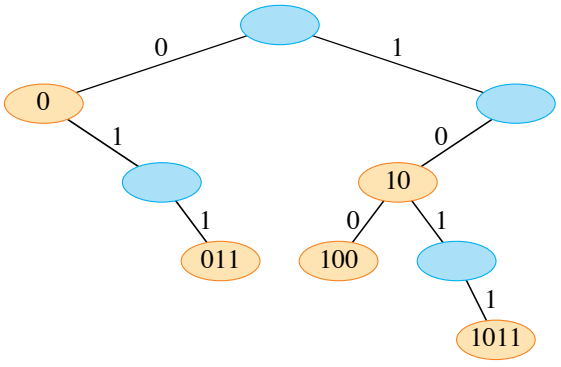


图 12.5 存储位串 1011、10、011、100 和 0 的基数树。要确定每个节点的密钥，请遍历从根到该节点的简单路径。因此，无需将密钥存储在节点中。此处出现的密钥仅用于说明目的。对应于蓝色节点的密钥不在树中。此类节点的存在只是为了建立到其他节点的路径。

12-3Average node depth in a randomly built binary search tree

n 个键上的 ***randomly built binary search tree*** 是一棵二叉搜索树，它从一棵空树开始，以随机顺序插入键，其中键的每个 nŠ 排列都具有相同的可能性。在本问题中，您将证明随机构建的具有 n 个节点的二叉搜索树中节点的平均深度为 O.lg n/。该技术揭示了二叉搜索树的构建与第 7.3 节中 RANDOMIZED-QUICKSORT 的执行之间惊人的相似性。

用 d.x; T / 表示树 T 中任意节点 x 的深度。那么树 T 的 ***total path length*** P . T / 就是 T 中所有节点 x 的 d.x; T / 之和。

a.假设 T 中节点的平均深度为

$$\frac{1}{n}\sum_{x \in T} d(x,T) = \frac{1}{n}P(T) .$$

因此，您需要证明 P .T / 的预期值是 O.n lg n/。

b.令 T_L 和 T_R 分别表示树 T 的左子树和右子树。假设 T 有 n 个节点，则

$$P(T) = P(T_L) + P(T_R) + n - 1 .$$

c.令 P.n/ 表示随机构建的具有 n 个节点的二叉搜索树的平均总路径长度。证明

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1) .$$

d.演示如何将 P .n/ 重写为

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) .$$

e.回顾问题 7-3 中给出的快速排序随机版本的替代分析，得出结论：P.n/DO. nlgn/。

每次递归调用随机快速排序都会选择一个随机枢轴元素来划分要排序的元素集。二叉搜索树的每个节点都会划分落入以该节点为根的子树的元素集。

f.描述快速排序的一种实现，其中对一组元素进行排序的比较与将元素插入二叉搜索树的比较完全相同。（进行比较的顺序可能不同，但必须进行相同的比较。）

12-4Number of different binary trees

令 b $_n$ 表示具有 n 个节点的不同二叉树的数量。在此问题中，您将找到 b $_n$ 的公式以及渐近估计。

a.证明 b $_0$ D 1 并且对于 n ‹ 1,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

b.请参阅第 121 页的问题 4-5，了解生成函数的定义，设 B.x/ 为生成函数

$$B(x) = \sum_{n=0}^{\infty} b_n x^n .$$

证明 B.x/ D xB.x/ ^{2} C 1，因此以封闭形式表达 B.x/ 的一种方法是

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1-4x}\right) .$$

f .x/ 在点 x D a 附近的 ***Taylor expansion*** 由下式给出

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k ,$$

其中 f (k) .x/ 是 f 的 k 阶导数 —t余。

c.表明

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(通过使用 p 1 1 4x 围绕 x D 0 的泰勒展开式，可以得到第 n 个 ***Catalan number***) 式。（如果愿意，也可以不使用泰勒展开式，而使用二项式定理的推广，即第 1181 页的公式 (C.4)，将其应用于非整数指数 n，其中，对于任何实数 n 和任何整数 k，如果 k ≥ 0，则可以将 ã $\binom{n}{k}$ ä 解释为 n.n 1/ .n k C 1//k Š，否则为 0。）

d.表明

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}}(1 + O(1/n)).$$

章节注释

Knuth [261] 很好地讨论了简单的二叉搜索树以及许多变体。二叉搜索树似乎是在 20 世纪 50 年代末由许多人独立发现的。基数树通常称为 “tries，”，源自单词 *retrieval* 中的中间字母。Knuth [261] 也讨论了它们。

许多文本，包括本书的前两个版本，都描述了一种从二叉搜索树中删除节点的稍微简单的方法，当该节点的两个子节点都存在时。不是用其后继 y 替换节点 *z*，而是删除节点 y，但将其键和卫星数据复制到节点 *z* 中。这种方法的缺点是实际删除的节点可能不是传递给删除过程的节点。如果程序的其他组件维护指向树中节点的指针，则它们可能会错误地得到指向已删除节点的 “过时的” 指针。虽然本书的这个版本中介绍的删除方法稍微复杂一些，但它保证调用删除节点 *z* 会删除节点 *z* 并且只删除节点 *z*。

14.5 节将展示如何在构建二叉搜索树之前知道搜索频率的情况下构建最佳二叉搜索树。也就是说，给定搜索每个键的频率以及搜索树中键之间的值的频率，在构建的二叉搜索树中进行一组搜索将检查最少数量的节点。

13 Red-Black Trees

第 12 章表明，高度为 h 的二叉搜索树可以在 O.h/ 时间内支持任何基本动态集操作4，例如搜索、前导、后继、最小值、最大值、插入和删除4。因此，如果搜索树的高度较小，集合操作会很快。但是，如果其高度很大，集合操作的运行速度可能不会比使用链接列表快。红黑树是许多搜索树方案之一，这些方案是“平衡的”，以保证基本动态集操作在最坏情况下需要 O.lg n/ 时间。

13.1 红黑树的性质

red-black tree 是二叉搜索树，每个节点有一个额外的存储空间：即其 ***color***，可以是红色或黑色。通过限制从根到叶的任何简单路径上的节点颜色，红黑树可确保任何一条路径的长度都不会超过其他路径的两倍，因此该树大约为 ***balanced***。事实上，正如我们即将看到的，具有 n 个键的红黑树的高度最多为 2 lg.n C 1/，即 O.lg n/。

现在，树的每个节点都包含属性 *color* 、 *key* 、 *left* 、 *right* 和 p。如果节点的子节点或父节点不存在，则该节点的相应指针属性包含值 NIL。将这些 NIL 视为指向二叉搜索树的叶子（外部节点）的指针，将普通的、包含键的节点视为树的内部节点。

红黑树是一棵满足以下***red-black properties***的二叉搜索树：

1. 每个节点要么是红色要么是黑色。2. 根节点是黑色。3. 每个叶子节点 (NIL) 都是黑色。

4. 如果一个节点是红色的，那么它的两个子节点都是黑色的。5. 对于每个节点，从该节点到后代叶子的所有简单路径都包含相同数量的黑色节点。

图 13.1(a) 显示了红黑树的一个例子。

为了方便处理红黑树代码中的边界条件，我们使用单个哨兵来表示 NIL（参见第 262 页）。对于红黑树 T，哨兵 T：*nil* 是一个具有与树中普通节点相同属性的对象。其 *color* 属性为 BLACK，其其他属性4p、*left* 、*right* 和 *key*4 可以取任意值。如图 13.1(b) 所示，所有指向 NIL 的指针都被替换为指向哨兵 T：*nil* 的指针。

为什么要使用哨兵？哨兵可以将节点 x 的 NIL 子节点视为父节点为 x 的普通节点。另一种设计是为树中的每个 NIL 使用不同的哨兵节点，这样每个 NIL 的父节点都有很好的定义。但是，这种方法会浪费空间。相反，只有一个哨兵 T：*nil* 代表所有 NIL（所有叶子和根的父节点）。哨兵的属性 p、*left* 、*right* 和 *key* 的值并不重要。红黑树程序可以将任何值放入哨兵中，以产生更简单的代码。

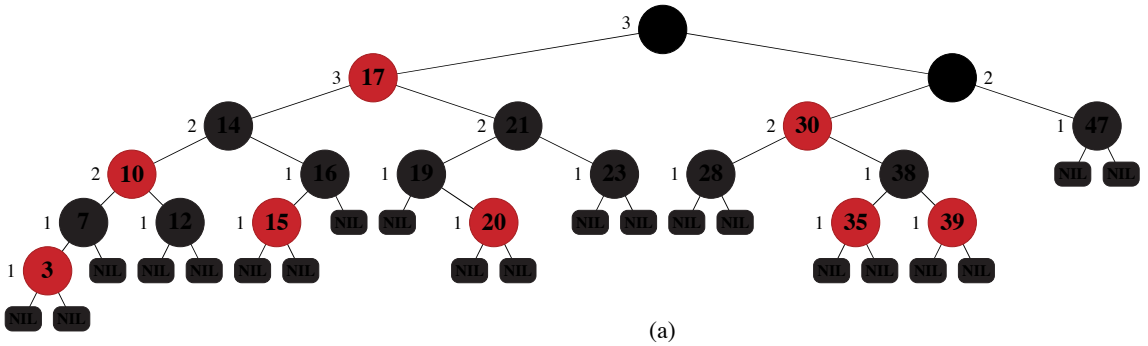
我们通常将兴趣集中在红黑树的内部节点上，因为它们保存着键值。本章的其余部分将省略红黑树图中的叶子，如图 13.1(c) 所示。

我们将从节点 x 向下到叶子的任何简单路径（但不包括该路径）上的黑色节点数称为节点的 ***black-height***，记为 bh.x/。根据属性 5，黑色高度的概念定义得很好，因为从节点向下的所有简单路径都具有相同数量的黑色节点。红黑树的黑色高度是其根的黑色高度。

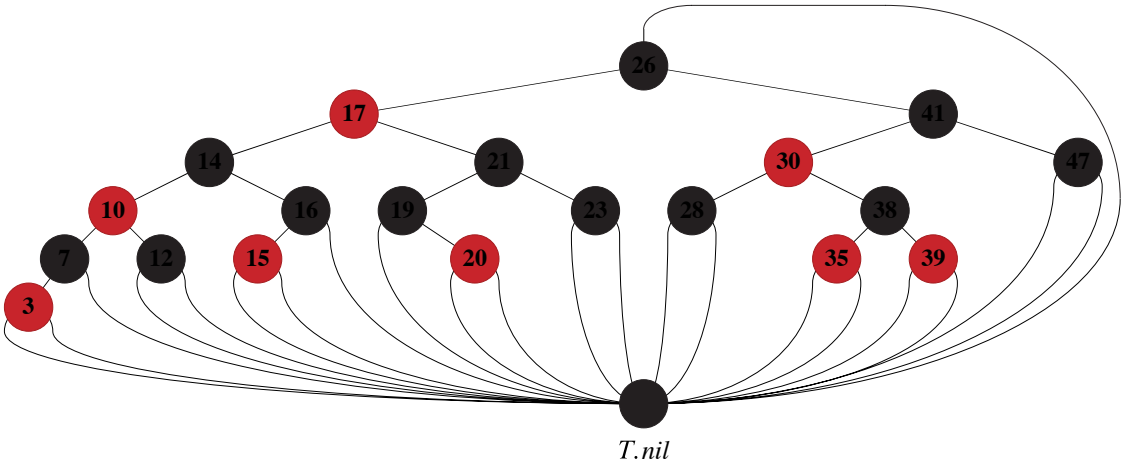
以下引理说明了为什么红黑树可以成为优秀的搜索树。

Lemma13.1
具有 n 个内部节点的红黑树的高度最多为 2 lg.n C 1/。

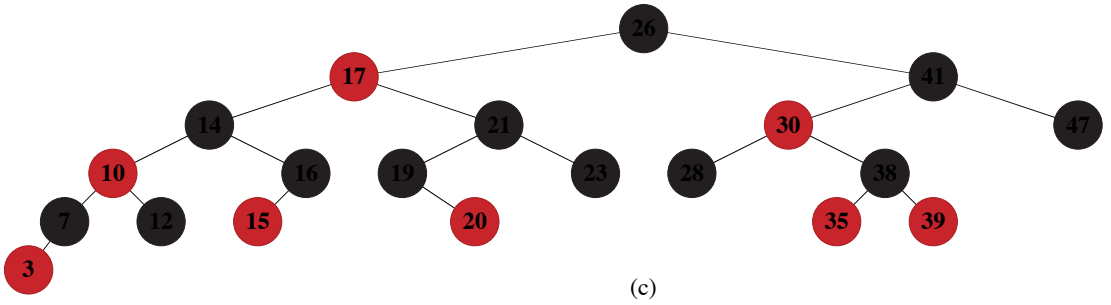
Proof我们首先证明以任何节点 x 为根的子树至少包含 $2^{bh(x)}$ 1 个内部节点。我们通过对 x 的高度进行归纳来证明这一说法。如果 x 的高度为 0，则 x 必定是一片叶子（T：*nil*），且以 x 为根的子树确实包含至少 $2^{bh(x)}$ 1 D 2^0 1 D 0 个内部节点。对于归纳步骤，考虑一个节点 x，它具有正高度并且是内部节点。则节点 x 有两个子节点，其中一个或两个都是叶子节点。如果子节点是黑色，则它对 x 的黑色高度贡献 1，但不贡献其自身的黑色高度。如果子节点是红色，则它既不贡献 x 的黑色高度，也不贡献其自身的黑色高度。因此，每个子节点的黑色高度要么是 bh.x/ 1 （如果它是黑色），要么是 bh.x/ （如果它是红色）。由于 x 的孩子的身高小于 x 本身的身高，我们可以应用归纳法



(a)



(b)



(c)

图 13.1 红黑树。红黑树中的每个节点要么是红色要么是黑色，红色节点的子节点都是黑色，从节点到后代叶子的每条简单路径都包含相同数量的黑色节点。（a）每个叶子（显示为 NIL）都是黑色的。每个非 NIL 节点都标有其黑色高度，其中 NIL 的黑色高度为 0。（b）相同的红黑树，但每个 NIL 都被单个哨兵 T：*nil* 替换，该哨兵始终为黑色，并且省略了黑色高度。根的父节点也是哨兵。（c）相同的红黑树，但叶子和根的父节点被完全省略。本章的其余部分使用这种绘制风格。

假设得出每个孩子至少有 $2^{bh(x)-1}$ 1 个内部节点的结论。因此，以 x 为根的子树至少包含 .2 $^{bh(x)-1}$ 1/ C.2 $^{bh(x)-1}$ 1/ C1 D 2 $^{bh(x)}$ 1 个内部节点，证明了该断言。

为了完成引理的证明，设 h 为树的高度。根据性质 4，从根到叶的任何简单路径（不包括根）上至少有一半节点必须是黑色的。因此，根的黑色高度必须至少为 h/2，因此，

$$n \geq 2^{h/2} - 1 .$$

将 1 移到左边并对两边取对数，得出 lg.n C 1/ h/2，或 h 2 lg.n C 1/。

■

作为该引理的直接推论，每个动态集合操作 SEARCH、MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 都在红黑树上运行 O.lg n/ 时间，因为每个操作都可以在高度为 h 的二叉搜索树上运行 O.h/ 时间（如第 12 章所示），并且任何 n 个节点上的红黑树都是高度为 O.lg n/ 的二叉搜索树。（当然，第 12 章算法中对 NIL 的引用必须替换为 T: *nil*。）虽然第 12 章中的过程 TREE-INSERT 和 TREE-DELETE 在给定红黑树作为输入时运行时间为 O.lg n/ 时间，但您不能仅使用它们来实现动态集合操作 INSERT 和 DELETE。它们不一定保持红黑属性，因此您可能不会得到合法的红黑树。本章的其余部分将展示如何在 O.lg n/ 时间内向红黑树中插入和删除数据。

练习

13.1-1

按照图 13.1(a) 的样式，在键 f1；2；:::；15g 上绘制高度为 3 的完全二叉搜索树。添加 NIL 叶子并以三种不同的方式对节点着色，使得生成的红黑树的黑色高度分别为 2、3 和 4。

13.1-2

绘制在图 13.1 中的树上用键 36 调用 TREE-I NSERT 后得到的红黑树。如果插入的节点为红色，则生成的树是红黑树吗？如果是黑色的呢？

13.1-3

定义一棵 ***relaxed red-black tree*** 为满足红黑性质 1、3、4 和 5 的二叉搜索树，但其根可以是红色或黑色。考虑一棵根为红色的宽松红黑树 T。如果将 T 的根更改为黑色但没有发生其他变化，则生成的树是红黑树吗？

13.1-4
假设红黑树 “ 中的每个黑色节点吸收了其所有红色子节点”，因此任何红色节点的子节点都将成为黑色父节点的子节点。（忽略键发生的变化。）吸收了所有红色子节点后，黑色节点的可能度数是多少？您能说出结果树的叶子的深度是多少吗？

13.1-5
证明从红黑树中的节点 x 到后代叶子的最长简单路径的长度最多是从节点 x 到后代叶子的最短简单路径的两倍。

13.1-6
红黑树中黑色高度为 k 的树的最大内部节点数是多少？最小内部节点数是多少？

13.1-7
描述一个具有 n 个键的红黑树，该树实现红色内部节点与黑色内部节点的最大可能比率。这个比率是多少？哪棵树的比率最小，这个比率是多少？

13.1-8
认为在 红黑树，红色节点不能有且只有一个 n on-NIL 子项。

13.2 旋转

搜索树操作 TREE-INSERT 和 TREE-DELETE 在具有 n 个键的红黑树上运行时，需要 O.lg n/ 时间。由于它们修改了树，因此结果可能违反第 13.1 节中列举的红黑属性。要恢复这些属性，需要更改节点内的颜色和指针。

指针结构通过 ***rotation*** 发生变化，这是搜索树中的局部操作，它保留了二叉搜索树的属性。图 13.2 显示了两种旋转：左旋转和右旋转。让我们看一下对节点 x 的左旋转，它将图右侧的结构转换为左侧的结构。节点 x 有一个右孩子 y ，它一定不是 T：*nil*。左旋转通过将 x 和 y 之间的链接向左扭转“来更改最初以 x 为根的子树。子树的新根是节点 y ， x 作为 y 的左孩子，y 的原始左孩子（图中 β 表示的子树）作为 x 的右孩子。

下页显示的 LEFT-ROTATE 伪代码假设 x: *right* ¤ T: *nil* 且根节点的父节点是 T: *nil*。图 13.3 显示了

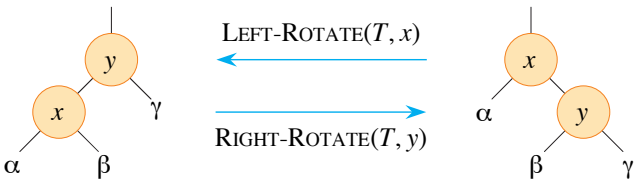


图 13.2 二叉搜索树的旋转操作。操作 L EFT-ROTATE.T; x/ 通过改变常数个指针，将右侧两个节点的配置转换为左侧的配置。逆操作 R IGHT-ROTATE.T; y/ 将左侧的配置转换为右侧的配置。字母 α、β和 表示任意子树。旋转操作保留了二叉搜索树的属性：α中的键位于 x: *key* 之前，而 x: *key* 位于 β中的键之前，而 y: *key* 位于 中的键之前。

LEFT-ROTATE 如何修改二叉搜索树的示例。RIGHT-ROTATE 的代码是对称的。LEFT-ROTATE 和 RIGHT-ROTATE 的运行时间均为 O.1/。旋转只会改变指针，节点中的所有其他属性保持不变。

左旋转 .T; x/

1 y D x: *right* 2 x: *right* D y: *left* // 将 y 的左子树变为 x 的右子树 3 if y: *left* ¤ T: *nil* // 如果 y 的左子树不为空 . . . 4 y: *left*: *p* D x // . . . 则 x 成为子树根的父节点 5 y: *p* D x: *p* // x 的父节点成为 y 的父节点 6 if x: *p* == T: *nil* // 如果 x 是根 . . . 7 T: *root* D y // . . . 则 y 成为根 8 elseif x == x: *p*: *left* // 否则，如果 x 是左孩子 . . . 9 x: *p*: *left* D y // . . . 那么 y 就成为左孩子 10 else x: *p*: *right* D y // 否则，x 是右孩子，现在 y 是 1 1 y: *left* D x // 使 x 成为 y 的左孩子 12 x: *p* D y

练习

13.2-1为 RIGHT-ROTATE 编写伪代码。

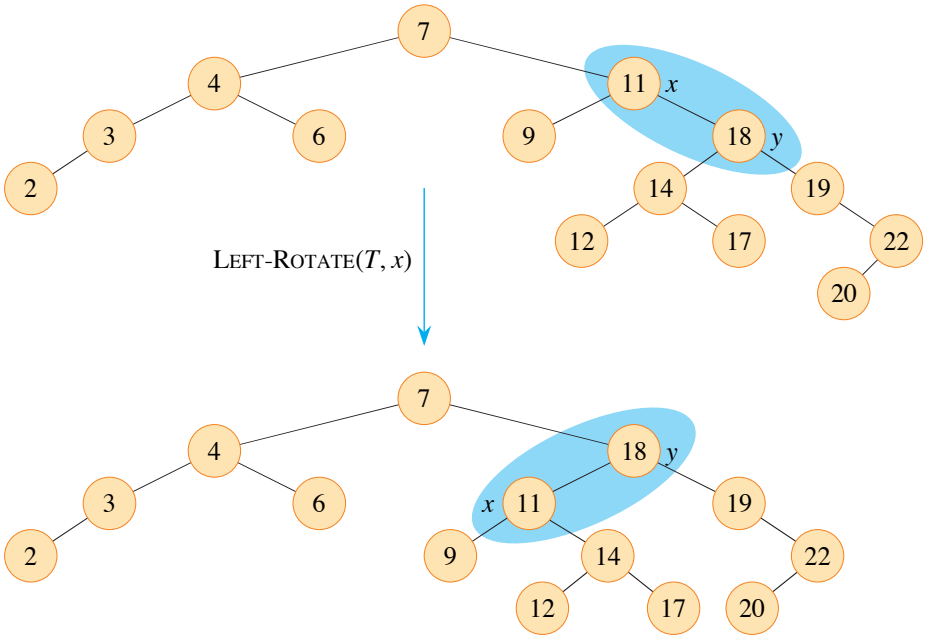


图 13.3 程序 L EFT-ROTATE.T; x/ 如何修改二叉搜索树的示例。输入树和修改后的树的中序树遍历产生相同的键值列表。

13.2-2

论证在每一棵 n 节点二叉搜索树中，恰好有 n 1 种可能的旋转。

13.2-3

假设 a、b 和 c 分别是图 13.2 右树中子树 α、β和 中的任意节点。当对图中节点 x 执行左旋转时，a、b 和 c 的深度如何变化？

13.2-4

证明任意 n 节点二叉搜索树都可以通过 O.n/ 次旋转变换为任意其他 n 节点二叉搜索树。（*Hint:* 首先证明最多 n 1 次右旋转足以将树变换为右行链。）

13.2-5

如果可以通过一系列 RIGHT-ROTATE 调用从 T_1 获得 T_2，则我们称二叉搜索树 T_1 可以从 ***right-converted*** 转换为二叉搜索树 T_2。给出两个树 T_1 和 T_2 的示例，使得 T_1 不能右转换为 T_2。然后，说明如果树 T_1 可以右转换为 T_2，则可以使用 $O(n^2)$ 次 RIGHT-ROTATE 调用将其右转换。

13.3 插入

为了在 O.lg n/ 时间内将一个节点插入到具有 n 个内部节点的红黑树中并保持红黑属性，我们需要稍微修改第 321 页的 TREE-INSERT 过程。过程 RB-INSERT 首先将节点 z 插入到树 T 中，就好像它是一棵普通的二叉搜索树，然后将 z 涂成红色。（练习 13.3-1 要求您解释为什么要将节点 z 设为红色而不是黑色。）为了保证保留红黑属性，对页上的辅助过程 RB-INSERT-FIXUP 会重新着色节点并执行旋转。调用 RB-INSERT .T; z/将节点 z（假设其 *key* 已经填充）插入到红黑树 T 中。

```
RB-INSERT .T; z/
 1 x D T: root                    // 节点与 z 进行比较 2 y D T:
nil                    // y 将成为 z 的父节点 3 while x ¤ T: nil                // 下降
直到到达哨兵节点 4 y D x 5 if zkey < x: key 6 x D x: left  7 else x D x: right
8 zp  D y // 找到带有父节点 y 的位置 4 插入 z 9 if y == T: nil  10 T: root  D z
// 树 T 为空 11 elseif zkey < y: key 12 y: left  D z 13 else y : right  D z 14 z: left
D T : nil                    // z 的两个孩子都是哨兵 15 z: right  D T : nil  16 z:
color D RED // 新节点从红色开始 17 RB-INSERT-FIXUP .T; z/ // 纠正任
何违反红黑性质的行为
```

TREE-INSERT 和 RB-INSERT 过程有四个不同之处。首先，TREE-INSERT 中所有 NIL 实例均被替换为 T: *nil*。其次，RB-INSERT 的第 14315 行将 z*left* 和 z*right* 设置为 T: *nil*，以保持正确的树结构。（TREE-INSERT 假定 z 的子节点已经为 NIL。）第三，第 16 行将 z 染成红色。第四，由于将 z 染成红色可能导致违反红黑属性之一，因此 RB-INSERT 的第 17 行调用 RB-INSERT-FIXUP .T; z/以恢复红黑属性。

RB-I NSERT-FIXUP .T; / 1 while : p: color == RED 2 if : p == : p: p: left // z的父母是否是左孩子？ 3 y D : p: p: right // y 是 z的叔叔 4 if y: color == RED // z的父母和叔叔是否都是红色？ 5 : p: color D 黑色 * 6 y.color = BLACK

7 z.p.p.color = RED 8 z = z.p.p

9 else

11 z = z.p 10 if z == z.p.right

12 LEFT-ROTATE(T, z)

13 z.p.color = BLACK

14 z.p.p.color = RED

15 RIGHT-ROTATE(T, z.p.p)

16 else // same as lines 3–15, but with “right” and “left” exchanged

17 y = z.p.p.left

18 if y.color == RED

19 z.p.color = BLACK

20 y.color = BLACK 21 z.p.p.color = RED

22 z = z.p.p

23 else

24 if z.p.left

27 z.p.color = BLACK

25 z = z.p

26 RIGHT-ROTATE(T, z)

28 z.p.p.color = RED

29 LEFT-ROTATE(T, z.p.p) 30 T.root.color = BLACK

要了解 RB-I NSERT-FIXUP 的工作原理，让我们分三个主要步骤检查代码。首先，我们将确定在插入节点 z 并将其涂成红色时，RB-I NSERT 中可能出现哪些违反红黑属性的情况。其次，我们将考虑 1329 行中 while 循环的总体目标。最后，我们将探索 while 循环主体中的三种情况（情况 2 会落入情况 3，因此这两种情况并不相互排斥），并查看它们如何实现目标。

在描述红黑树的结构时，我们经常需要引用节点父节点的兄弟节点。我们使用术语 ***uncle*** 来表示这样的节点。[1] 图 13.4 显示了 RB-I NSERT-FIXUP 如何对示例红黑树进行操作，其中的情况部分取决于节点、其父节点和其叔节点的颜色。

调用 RB-I NSERT-FIXUP 时，可能会发生哪些违反红黑性质的情况？性质 1 当然继续成立（每个节点要么是红色要么是黑色），性质 3 也一样（每个叶子都是黑色），因为新插入的红色节点的两个子节点都是哨兵节点 T：*nil* 。性质 5 表示从给定节点出发的每条简单路径上的黑色节点数相同，该性质也得到满足，因为节点 z 替换了（黑色）哨兵节点，而节点 z 是红色且有哨兵子节点。因此，唯一可能违反的性质是性质 2（要求根节点是黑色）和性质 4（表示红色节点不能有红色子节点）。由于 z 是红色，因此可能出现这两种违反情况。如果 z 是根节点，则违反性质 2；如果 z 的父节点是红色，则违反性质 4。图 13.4（a）显示了插入节点 z 后违反属性 4 的情况。

1329 行的 while 循环有两种对称可能性：第 3315 行处理节点 z 的父节点 $z.p$ 是 z 祖父节点 $z.p.p$ 的左子节点的情况，第 17329 行适用于 z 的父节点是右子节点的情况。我们的证明将仅关注第 3315 行，并依赖第 17329 行中的对称性。

我们将证明 while 循环在循环的每次迭代开始时保持以下三部分不变量：

a. 节点 z 为红色。
b. 如果 $z.p$ 是根，则 $z.p$ 为黑色。
c. 如果树违反了红黑属性中的任何一个，那么它最多违反其中的一个，并且违反的是属性 2 或属性 4，但不是同时违反两者。如果树违反了属性 2，那是因为 z 是根并且是红色的。如果树违反了属性 4，那是因为 z 和 $z.p$ 都是红色的。

部分 (c) 处理违反红黑属性的情况，它比部分 (a) 和 (b) 更能说明 RB-I NSERT-FIXUP 恢复了红黑属性，我们将在后续过程中使用这两个部分来理解代码中的情况。因为我们将重点关注节点 z 及其附近的节点，所以从部分 (a) 中知道 z 是红色会有所帮助。部分 (b) 将有助于说明 z 的祖父节点 $z.p.p$ 在第 2、3、7、8、14 和 15 行被引用时存在（回想一下，我们只关注第 33 至第 15 行）。

---

[1] Although we try to avoid gendered language in this book, the English language lacks a gender-neutral word for a parent's sibling.

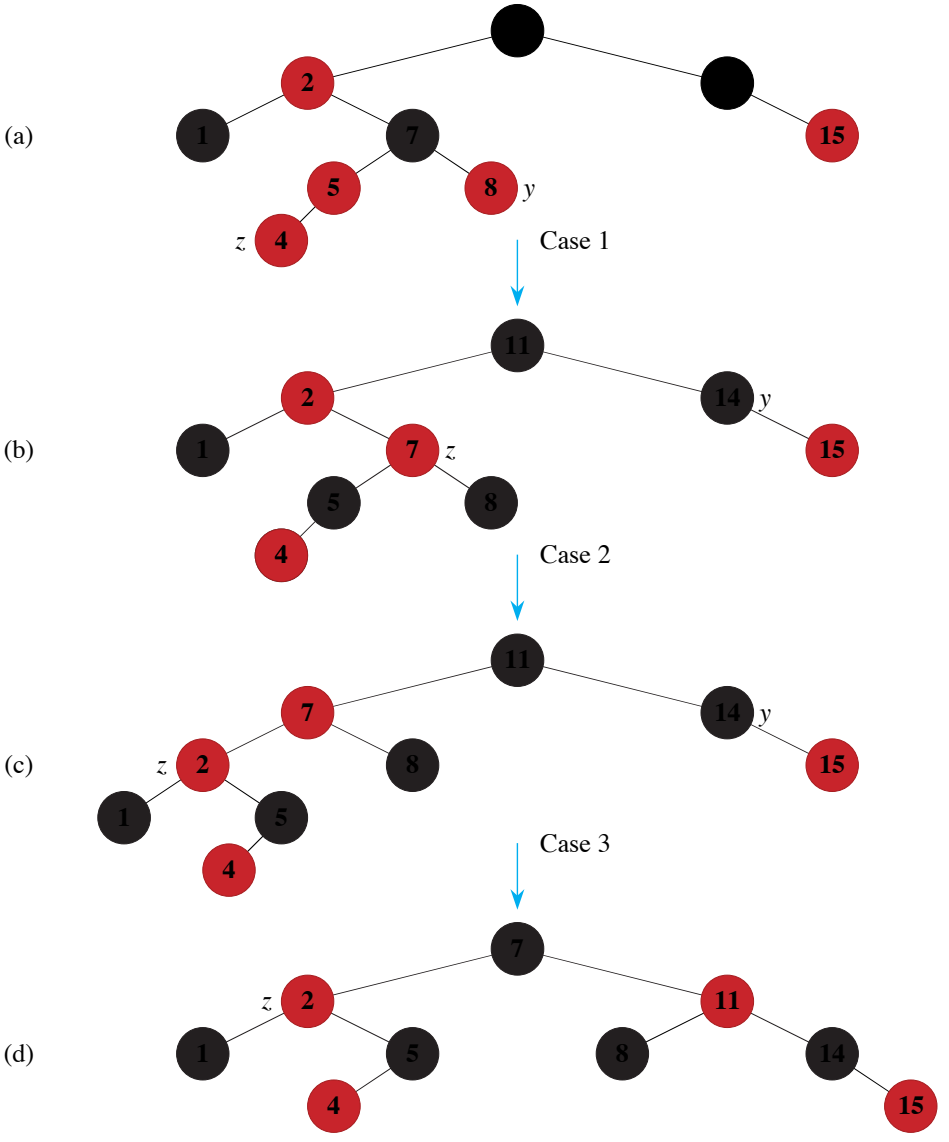


图 13.4 RB-I NSERT-FIXUP 操作。（a）插入后的节点 z。由于 z及其父节点 $z.p$ 均为红色，因此违反了属性 4。由于 z的叔叔 y 为红色，因此代码中的情况 1 适用。节点 z的祖父节点 $z.p.p$ 必定为黑色，并且其黑色程度会向下一级转移到 z的父节点和叔叔节点。一旦指针 z在树中向上移动两级，就会得到 (b) 中所示的树。再次，z及其父节点均为红色，但这次 z的叔叔 y 为黑色。由于 z是 $z.p$ 的右子节点，因此情况 2 适用。执行左旋转会得到 (c) 中的树。现在 z是其父节点的左子节点，情况 3 适用。重新着色和右旋转产生 (d) 中的树，这是一棵合法的红黑树。

回想一下，要使用循环不变量，我们需要证明在进入循环的第一次迭代时不变量为真，每次迭代都保持不变量不变，循环终止，并且循环终止时循环不变量为我们提供了一个有用的属性。我们将看到，循环的每次迭代都有两种可能的结果：指针 z 沿树向上移动，或者发生一些旋转然后循环终止。

初始化：在调用 RB-I NSERT 之前，红黑树没有任何违规行为。RB-I NSERT 添加一个红色节点 z 并调用 RB-I NSERT-FIXUP。我们将证明在调用 RB-I NSERT-FIXUP 时，不变量的每个部分都成立：

a. 调用 RB-I NSERT-FIXUP 时，z 是添加的红色节点。b. 如果 $z.p$ 是根节点，则 $z.p$ 一开始是黑色，并且在调用 RB-I NSERT-FIXUP 之前没有变化。c. 我们已经看到，当调用 RB-I NSERT-FIXUP 时，属性 1、3 和 5 成立。如果树违反属性 2（根节点必须是黑色），则红色根节点必须是新添加的节点 z。它是树中唯一的内部节点。由于 z 的父节点和两个子节点都是哨兵节点，是黑色的，因此树也不会违反属性 4（红色节点的两个子节点都是黑色）。因此，对属性 2 的这种违反是整个树中唯一违反红黑属性的情况。

如果树违反了属性 4，那么，由于节点 z 的子节点是黑色标记，并且树在添加 z 之前没有其他违规行为，违规行为一定是因为 z 和 $z.p$ 都是红色。此外，树不违反其他红黑属性。

维护：while 循环中有六种情况，但我们只检查第 3315 行中的三种情况，即节点 z 的父节点 $z.p$ 是 z 的祖节点 $z.p.p$ 的左子节点。第 17329 行的证明是对称的。节点 $z.p.p$ 存在，因为根据循环不变量的部分 (b)，如果 $z.p$ 是根，则 $z.p$ 为黑色。由于 RB-I NSERT-FIXUP 仅在 $z.p$ 为红色时才进入循环迭代，因此我们知道 $z.p$ 不能是根。因此，$z.p.p$ 存在。

案例 1 与案例 2 和 3 的区别在于 z 的叔叔 y 的颜色。第 3 行让 y 指向 z 的叔叔 $z.p.p.right$，第 4 行测试 y 的颜色。如果 y 为红色，则执行案例 1。否则，控制权将传递给案例 2 和 3。在这三种情况下，z 的祖父 $z.p.p$ 都是黑色，因为其父级 $z.p$ 是红色，并且仅在 z 和 $z.p$ 之间违反了属性 4。

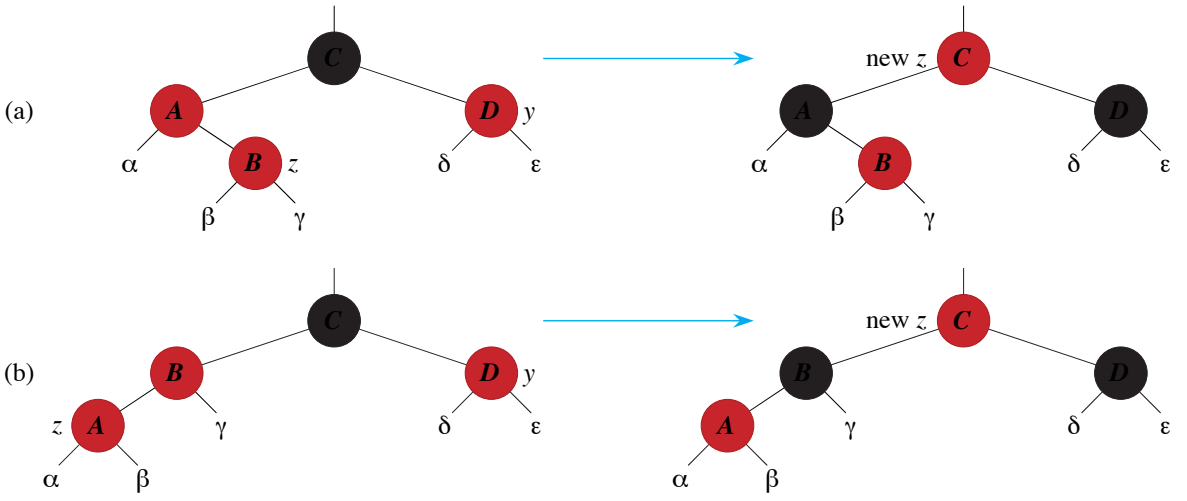


图 13.5 过程 RB-I NSERT-FIXUP 的案例 1。z及其父节点 zp 均为红色，违反了属性 4。在案例 1 中，z的叔叔 y 为红色。无论 (a) z是右孩子还是 (b) z是左孩子，都会发生相同的操作。子树 α β 、 ı 和 " 中的每一个都有一个黑色根4（可能是哨兵4），并且每个子树都具有相同的黑色高度。案例 1 的代码将 z祖父的黑色向下移动到 z的父节点和叔叔节点，从而保留属性 5：从节点到叶子的所有向下简单路径具有相同数量的黑色。while 循环继续以节点 z的祖父节点 zp: p 作为新的 z 如果案例 1 的操作导致发生新的违反属性 4 的情况，那么它必须只发生在新的 z（红色）与其父节点（如果它也是红色）之间。

Case1: z's uncle* 和 *is red

图 13.5 显示了第 1 种情况（第 538 行），即 zp 和 y 均为红色时的情况。由于 z的祖父节点 zp: p 为黑色，因此其黑色度可以向下传递一级至 zp 和 y，从而解决了 z和 zp 均为红色的问题。由于其黑色度向下传递了一级，z的祖父节点变为红色，从而保持了属性 5。while 循环以 zp: p 作为新节点 z重复，因此指针 z在树中向上移动了两级。

现在，我们展示情况 1 在下一次迭代开始时保持循环不变。我们使用 z表示当前迭代中的节点 z 并使用 z D zp: p 表示在下一次迭代中将在第 1 行的测试中称为节点 z的节点。

a. 因为本次迭代将 zp: p 染成红色，所以节点 z 在下一次迭代开始时为红色。 b. 节点 z: p 在本次迭代中为 zp: p: p，并且此节点的颜色不变。如果此节点是根节点，则在本次迭代之前它是黑色的，并且在下一次迭代开始时它仍为黑色。

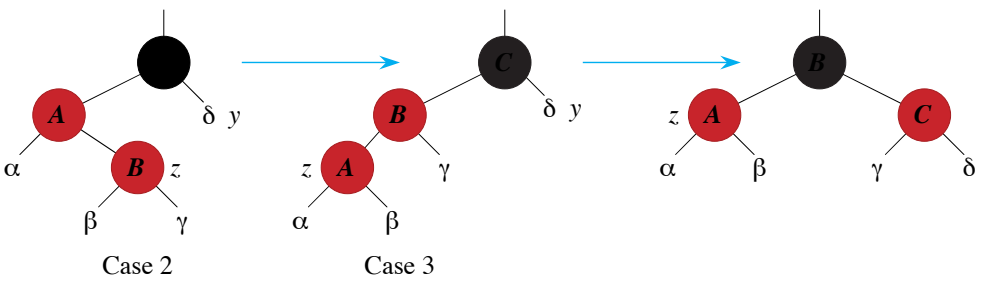


图 13.6 过程 RB-I NSERT-FIXUP 的案例 2 和案例 3。与案例 1 一样，在案例 2 或案例 3 中，属性 4 都被违反，因为 z 及其父节点 $z.p$ 都是红色的。每个子树 α、β、γ 和 δ 都有一个黑色根（α、β 和 γ 来自属性 4，以及 δ，因为否则将适用案例 1），并且每个子树都具有相同的黑色高度。案例 2 通过左旋转转换为案例 3，这保留了属性 5：从节点到叶子的所有向下简单路径具有相同数量的黑色。案例 3 导致一些颜色变化和右旋转，这也保留了属性 5。然后 while 循环终止，因为属性 4 得到满足：一行中不再有两个红色节点。

c. 我们已经论证过，案例 1 维持了性质 5，并且没有违反性质 1 或性质 3。

如果节点 z 在下一次迭代开始时是根节点，则案例 1 纠正了此迭代中唯一违反属性 4 的情况。由于 z 为红色且是根节点，因此属性 2 成为唯一违反的属性，而此违反是由于 z 造成的。

如果节点 z 在下一次迭代开始时不是根节点，则案例 1 未违反属性 2。案例 1 纠正了此迭代开始时存在的唯一违反属性 4 的情况。然后，它将 z 变为红色，而将 $z.p$ 单独保留。如果 $z.p$ 为黑色，则不违反属性 4。如果 $z.p$ 为红色，则将 z 染成红色会在 z 和 $z.p$ 之间造成一次违反属性 4 的情况。

Case2: z's uncle* 是 *is black and z is a right child
Case 3: z's uncle* 是 *is black and z is a left child
在案例 2 和案例 3 中，z 的叔叔 y 的颜色为黑色。我们根据 z 是 $z.p$ 的右子节点还是左子节点来区分这两种情况，这两种情况都假设 z 的父节点 $z.p$ 是红色且是左子节点。第 11312 行构成案例 2，与案例 3 一起显示在图 13.6 中。在案例 2 中，节点 z 是其父节点的右子节点。左旋转会立即将情况转换为案例 3（第 13315 行），其中节点 z 是左子节点。由于 z 和 $z.p$ 都是红色，因此旋转既不影响节点的黑色高度，也不影响属性 5。无论案例 3 是直接执行还是通过案例 2 执行，z 的叔叔 y 都是黑色，否则案例 1 就会运行。此外，节点 $z.p.p$ 存在，因为我们已经论证了这一点

节点在执行第 2 行和第 3 行时存在，并且在第 11 行将 z上移一级，然后在第 12 行下移一级之后，$z.p = p$ 的身份保持不变。案例 3 执行了一些颜色更改和右旋转，从而保留了属性 5。此时，一行中不再有两个红色节点。while 循环在第 1 行的下一个测试时终止，因为 $z.p$ 现在是黑色。

我们现在证明情况 2 和 3 保持了循环不变式。（正如我们刚才所论证的，$z.p$ 在第 1 行的下一个测试中将为黑色，并且循环体将不会再次执行。）

a. 情况 2 使 z指向 $z.p$，而 $z.p$ 为红色。在情况 2 和情况 3 中，z及其颜色均未发生进一步改变。b. 情况 3 使 $z.p$ 为黑色，因此，如果 $z.p$ 在下一次迭代开始时是根，则它为黑色。c. 与情况 1 一样，在情况 2 和情况 3 中仍保留属性 1、3 和 5。由于节点 z不是情况 2 和情况 3 中的根，因此我们知道没有违反属性 2。情况 2 和情况 3 并未引入对属性 2 的违反，因为在情况 3 中，唯一变为红色的节点通过旋转成为黑色节点的子节点。情况 2 和情况 3 纠正了对属性 4 的唯一违反，并且未引入其他违反。

终止：要查看循环是否终止，请观察如果仅发生情况 1，则节点指针 z会在每次迭代中向根移动，因此最终 $z.p$ 为黑色。（如果 z是根，则 $z.p$ 是哨兵 T: *nil*，它是黑色。）如果发生情况 2 或情况 3，则我们已经看到循环终止。由于循环因 $z.p$ 为黑色而终止，因此树在循环终止时不违反属性 4。根据循环不变量，唯一可能不成立的属性是属性 2。第 30 行通过将根涂成黑色来恢复此属性，因此当 RB-I NSERT-FIXUP 终止时，所有红黑属性都成立。

因此，我们证明了 RB-I NSERT-FIXUP 正确恢复了红黑属性。

分析

RB-I NSERT 的运行时间是多少？由于 n 个节点上的红黑树的高度为 O.lg n/，因此 RB-I NSERT 的 1316 行需要 O.lg n/ 时间。在 RB-I NSERT-FIXUP 中，只有当 case 1 发生时，while 循环才会重复，然后指针 z会沿树向上移动两层。因此，while 循环可以执行的总次数为 O.lg n/。因此，RB-I NSERT 总共需要 O.lg n/ 时间。此外，它

永远不会执行超过两次旋转，因为当执行了情况 2 或情况 3 时，while 循环就会终止。

练习

13.3-1
RB-I NSERT 的第 16 行将新插入的节点 *z*的颜色设置为红色。如果将 *z*的颜色设置为黑色，则不会违反红黑树的属性 4。为什么不将 *z*的颜色设置为黑色？

13.3-2
展示将键 41；38；31；12；19；8 依次插入到最初为空的红黑树后得到的红黑树。

13.3-3
假设图 13.5 和图 13.6 中的每棵子树 *α β* ；ı；" 的黑高均为 k。用黑高标记每幅图中每个节点，以验证所示变换是否保留属性 5。

13.3-4
Teach 教授担心 RB-I NSERT-FIXUP 可能会将 T: *nil*: *color* 设置为 RED，在这种情况下，当 *z*为根时，第 1 行中的测试不会导致循环终止。通过论证 RB -I NSERT-FIXUP 永远不会将 T: *nil*: *color* 设置为 RED，可以证明教授的担心是没有根据的。

13.3-5
考虑使用 RB-I NSERT 插入 n 个节点形成的红黑树。论证如果 n > 1，则该树至少有一个红色节点。

13.3-6
如果红黑树的表示不包含父指针的存储，请建议如何有效地实现 RB-I NSER T。

13.4 删除

与 n 节点红黑树上的其他基本操作一样，删除一个节点需要 O.lg n/ 时间。从红黑树中删除一个节点比插入一个节点更复杂。

从红黑树中删除节点的过程基于第 325 页的 TREE-DELETE 过程。首先，我们需要定制 T RANSPLANT

第 324 页的子例程由 TREE-DELETE 调用，以便它适用于红黑树。与 TRANSPLANT 一样，新过程 RB-TRANSPLANT 用以节点 v 为根的子树替换以节点 u 为根的子树。RB-TRANSPLANT 过程与 TRANSPLANT 有两点不同。首先，第 1 行引用标记 T: *nil* 而不是 NIL。其次，第 6 行中对 v: p 的赋值是无条件发生的：即使 v 指向标记，过程也可以赋值给 v: p 。我们将利用当 v D T: *nil* 时赋值给 v: p 的能力。

```
RB-移植 .T; u; v/
1 if u：p ==T：nil  2
T：root  D v 3 elseif u
==u：p：left  4 u：p
：left  D v 5 else u：p
：right  D v 6 v：p  D
u：p
```

下一页的过程 RB-DELETE 与 TREE-DELETE 过程类似，但增加了伪代码行。增加的代码行处理可能涉及违反红黑性质的节点 x 和 y。被删除的节点 z 最多有一个子节点，则 y 将为 z。当 z 有两个子节点时，与 TREE-DELETE 一样，y 将成为 z 的后继，它没有左子节点并移至树中 z 的位置。此外，y 将采用 z 的颜色。在任一情况下，节点 y 最多有一个子节点：节点 x，它将取代 y 在树中的位置。（如果 y 没有子节点，则节点 x 将成为标记 T：*nil* 。）由于节点 y 要么从树中移除，要么在树内移动，因此该过程需要跟踪 y 的原始颜色。如果删除节点 z 后红黑性质可能被违反，RB-DELETE 会调用辅助过程 RB-DELETE-FIXUP，该过程会改变颜色并执行旋转以恢复红黑性质。

尽管 RB-DELETE 包含的伪代码行数几乎是 TREE-DELETE 的两倍，但这两个过程具有相同的基本结构。您可以在 RB-DELETE 中找到 TREE-DELETE 的每一行（将 NIL 替换为 T: *nil* ，并将对 TRANSPLANT 的调用替换为对 RB-TRANSPLANT 的调用），这些过程在相同的条件下执行。

具体来说，这两个程序之间的其他差异如下：

第 1 行和第 9 行按上述方式设置节点 y：第 1 行表示节点 z 最多有一个子节点，第 9 行表示 z 有两个子节点。• 因为节点 y 的颜色可能会改变，所以变量 *y-original-color* 会在发生任何变化之前存储 y 的颜色。第 2 行和第 10 行在对 y 赋值后立即设置此变量。当节点 z 有两个子节点时，节点 y 和 z 是

RB-删除 .T; z

```
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)          // replace z by its right child
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)           // replace z by its left child
 9  else y = TREE-MINIMUM(z.right)            // y is z's successor
10      y-original-color = y.color
11      x = y.right
12      if y ≠ z.right                        // is y farther down the tree?
13          RB-TRANSPLANT(T, y, y.right)      // replace y by its right child
14          y.right = z.right                 // z's right child becomes
15          y.right.p = y                     //       y's right child
16      else x.p = y                          // in case x is T.nil
17      RB-TRANSPLANT(T, z, y)                // replace z by its successor y
18      y.left = z.left                       // and give z's left child to y,
19      y.left.p = y                          //       which had no left child
20      y.color = z.color
21  if y-original-color == BLACK          // if any red-black violations occurred,
22      RB-DELETE-FIXUP(T, x)             //       correct them
```

不同。在本例中，第 17 行将 y 移动到树中 z 的原始位置（即调用 RB-DELETE 时 z 在树中的位置），第 20 行将 y 赋予与 z 相同的颜色。当节点 y 最初为黑色时，删除或移动它可能会导致违反红黑属性，这可以通过第 22 行中调用 RB-DELETE-FIXUP 来纠正。

如上所述，该过程跟踪在调用时移动到节点 y 的原始位置的节点 x。第 4、7 和 11 行中的赋值将 x 设置为指向 y 的唯一子节点，或者，如果 y 没有子节点，则指向标记 T: *nil*。

由于节点 x 移动到节点 y 的原始位置，因此必须正确设置属性 x: p 。如果节点 z 有两个孩子，而 y 是 z 的右孩子，则 y 只需移动到 z 的位置，x 仍是 y 的孩子。第 12 行检查这种情况。尽管您可能认为在第 16 行将 x: p 设置为 y 是不必要的，因为 x 是 y 的孩子，但 RB-DELETE-FIXUP 的调用依赖于 x: p 为 y，即使 x 是 T: *nil*。因此，当 z 有两个孩子，而 y 是 z 的右孩子时，执行

如果 y 的右孩子是 T: *nil*，则第 16 行是必需的，否则它不会改变任何内容。
否则，节点 z 要么与节点 y 相同，要么是 y 原始父节点的正确祖先。在这些情况下，第 5、8 和 13 行中的 RB-TRANSPLANT 调用在 RB-TRANSPLANT 的第 6 行中正确设置了 x: p。（在这些 RB-TRANSPLANT 调用中，传递的第三个参数与 x 相同。）

最后，如果节点 y 为黑色，则可能出现一个或多个违反红黑属性的情况。第 22 行中的 RB-DELETE-FIXUP 调用恢复了红黑属性。如果 y 为红色，则当 y 被移除或移动时，红黑属性仍然成立，原因如下：

1. 树中的黑色高度没有改变。（参见练习 13.4-1。）
2. 没有红色节点变得相邻。如果 z 最多有一个子节点，那么 y 和 z 是同一个节点。该节点被移除，并由一个子节点替代它。如果被移除的节点是红色，那么它的父节点和子节点都不能是红色，因此移动子节点来取代它不会导致两个红色节点变得相邻。另一方面，如果 z 有两个子节点，那么 y 将取代 z 在树中的位置，同时取代 z 的颜色，因此 y 在树中的新位置不能有两个相邻的红色节点。此外，如果 y 不是 z 的右子节点，那么 y 的原始右子节点 x 将在树中替换 y。由于 y 是红色，所以 x 一定是黑色，因此用 x 替换 y 不会导致两个红色节点变得相邻。
3. 因为如果 y 是红色的话它就不可能是根，所以根仍然是黑色。

如果节点 y 为黑色，则可能出现三个问题，调用 RB-D ELETE-FIXUP 可以解决这些问题。首先，如果 y 是根，而 y 的一个红色子节点成为新的根，则违反了性质 2。其次，如果 x 和它的新父节点都是红色，就会违反性质 4。第三，在树内移动 y 会导致任何以前包含 y 的简单路径少一个黑色节点。因此，树中 y 的任何祖先现在都会违反性质 5。我们可以通过以下方式纠正性质 5 的违反：当黑色节点 y 被移除或移动时，它的黑色状态会转移到移动到 y 的原始位置的节点 x，从而使 x 得到一个 “额外的” 黑色。也就是说，如果我们将任何包含 x 的简单路径上的黑色节点数加 1，那么按照这种解释，性质 5 成立。但现在出现了另一个问题：节点 x 既不是红色也不是黑色，因此违反了属性 1。相反，节点 x 要么是 “双黑”，要么是 “红黑”，并且它分别对包含 x 的简单路径上的黑色节点数贡献 2 或 1。x 的 *color* 属性仍将是红色（如果 x 是红黑）或黑色（如果 x 是双黑）。换句话说，节点上的额外黑色反映在 x 指向该节点上，而不是 *color* 属性上。

下一页的过程 RB-DELETE-FIXUP 恢复了属性 1、2 和 4。练习 13.4-2 和 13.4-3 要求你证明该过程恢复了属性 2 和 4，因此在本节的剩余部分，我们将重点关注属性 1。1343 行中的 while 循环的目标是将额外的黑色沿树向上移动，直到

1. x 指向一个红黑节点，这种情况下第 44 行将 x （单独）涂成黑色；
2. x 指向根节点，这种情况下多余的黑色就会消失；或者
3. 执行适当的旋转和重新着色后，退出循环。

和 RB-I NSERT-FIXUP 一样，RB-DELETE-FIXUP 过程处理两种对称情况：第 3322 行处理节点 x 为左子节点的情况，第 24343 行处理节点 x 为右子节点的情况。我们的证明重点关注第 3322 行中所示的四种情况。

在 while 循环中，x 始终指向非根双黑节点。第 2 行确定 x 是其父节点 x: *p* 的左子节点还是右子节点，因此在给定的迭代中将执行第 3322 行或第 24343 行。x 的兄弟节点始终由指针 w 表示。由于节点 x 是双黑节点，因此节点 w 不能是 T: *nil*，因为否则，从 x: *p* 到（单黑）叶子 w 的简单路径上的黑色节点数量将小于从 x: *p* 到 x 的简单路径上的黑色节点数量。

回想一下，RB-DELETE 过程总是在调用 RB-DELETE-FIXUP 之前赋值给 x: *p* （在第 13 行调用 RB-TRANSPLANT 或在第 16 行赋值），即使节点 x 是标记 T: *nil*。这是因为 RB-DELETE-FIXUP 在几个地方引用了 x 的父节点 x: *p* ，即使 x 是 T: *nil*，这个属性也必须指向在 RB-DELETE4 中成为 x 父节点的节点。

图 13.7 演示了节点 x 为左孩子时代码中的四种情况。（与 RB-I NSERT-FIXUP 中一样，RB-DELETE-FIXUP 中的情况并不互相排斥。）在详细研究每种情况之前，让我们更概括地看一下如何验证每种情况下的变换是否保留了性质 5。关键思想是，在每种情况下，应用的变换都保留了从所示子树的根到每个子树 $\alpha\beta$: : : ; 的根之间的黑节点数（包括 x 的额外黑节点）。因此，如果性质 5 在变换之前成立，则在变换之后也继续成立。例如，在图 13.7 (a) 中说明了情况 1，在变换之前和之后，从根到子树 α 或 β 的根之间的黑节点数都是 3。（再次记住，节点 x 添加了一个额外的黑色。）类似地，无论是在变换之前还是之后，从根到 、 ı 、 " 和 中任何一个的根的黑色节点数都是 2。[2] 在图 13.7(b) 中，计数必须涉及所示子树根的 *color* 属性的值 c，它可以是红色或黑色。

---

[2] If property 5 holds, we can assume that paths from the roots of $\gamma, \delta, \varepsilon$, and ζ down to leaves contain one more black than do paths from the roots of α and β down to leaves.

RB-删除-修复 .T; x/

1 当 x ¤ T: *root* and x: *color* == 黑色 2 如果 x == x: *p*: *left* // x 是左孩子吗？ 3 w D x: *p*: *right* // w 是 x 的兄弟 4 如果 w: *color* == 红色 5 w: *color* D 黑色

```
*
        6           x.p.color = RED
        7           LEFT-ROTATE(T, x.p)
case 1
8       w = x.p.right
9       if w.left.color == BLACK and w.right.color == BLACK
10          w.color = RED                                  } case 2
11          x = x.p
12      else
13          if w.right.color == BLACK                      *
14              w.left.color = BLACK
                                                           case 3
15          w.color = RED
16          RIGHT-ROTATE(T, w)
17          w = x.p.right                                  • • •
18          w.color = x.p.color
19          x.p.color = BLACK                              case 4
20          w.right.color = BLACK
21          LEFT-ROTATE(T, x.p)
22          x = T.root
23  else // same as lines 3–22, but with “right” and “left” exchanged
24      w = x.p.left
25      if w.color == RED
26          w.color = BLACK
29          w = x.p.left
27          x.p.color = RED
30      if w.right.color == BLACK and w.left.color == BLACK
28          RIGHT-ROTATE(T, x.p)
31          w.color = RED
32          x = x.p
33      else
35          w.right.color = BLACK
34          if w.left.color == BLACK
36          w.color = RED
```

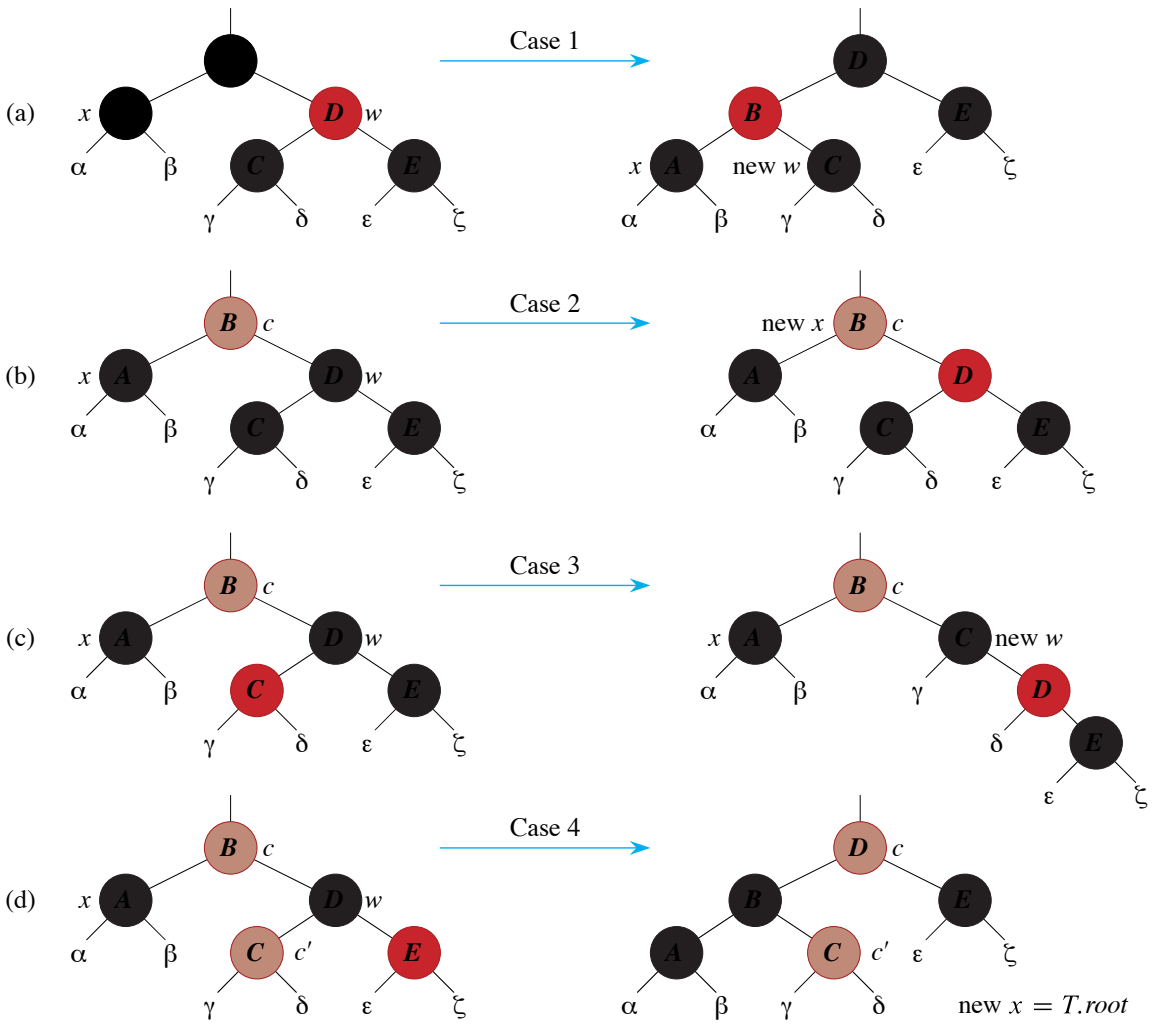


图 13.7 过程 RB-D ELETE-FIXUP 第 33 至 22 行中的案例。棕色节点具有由 c 和 c′ 表示的 *color* 属性，可以是红色或黑色。字母 α, β ::: ； 表示任意子树。每个案例通过更改某些颜色和/或执行旋转将左侧的配置转换为右侧的配置。x 指向的任何节点都有额外的黑色，并且是双黑或红黑。只有案例 2 会导致循环重复。（a）通过交换节点 B 和 D 的颜色并执行左旋转，案例 1 转换为案例 2、3 或 4。 (b) 在情况 2 中，指针 x 所表示的额外黑色通过将节点 D 染成红色并将 x 设置为指向节点 B 而沿树向上移动。如果通过情况 1 进入情况 2，则 while 循环终止，因为新节点 x 是红黑相间的，因此其 *color* 属性的值 c 为红色。 (c) 通过交换节点 C 和 D 的颜色并执行右旋转，将情况 3 转换为情况 4。 (d) 情况 4 通过更改某些颜色并执行左旋转（不违反红黑属性）删除 x 所表示的额外黑色，然后循环终止。

如果我们定义 count.RED/ D 0 和 count.BLACK/ D 1，那么从根到 d 的黑色节点数为 2 C count.c/，无论是在变换之前还是之后。在这种情况下，变换之后，新节点 x 具有 *color* 属性 c，但该节点实际上要么是红黑相间（如果 c D RED），要么是双黑（如果 c D BLACK）。您可以类似地验证其他情况（参见练习 13.4-6）。

Case1: x***'s sibling*** 宽 ***is red***
情况 1（第 538 行和图 13.7(a)）发生在节点 x 的兄弟节点 w 为红色时。由于 w 为红色，因此它必定有黑色子节点。此情况交换了 w 和 x 的颜色：p，然后对 x 执行左旋转：p，而不会违反任何红黑性质。x 的新兄弟节点（即旋转前 w 的子节点之一）现在是黑色，因此情况 1 转换为情况 2、3 或 4 之一。

当节点 w 为黑色并且通过 w 的子节点的颜色进行区分时，会发生情况 2、3 和 4。

Case2: x***'s sibling*** 含 ***is black, and both of*** 含***'s children are black***
在情况 2 中（第 10311 行和图 13.7(b)），w 的两个子节点都是黑色。由于 w 也是黑色，这种情况下会从 x 和 w 中各删去一个黑色，使得 x 只剩下一个黑色，而 w 为红色。为了补偿 x 和 w 各自丢失一个黑色，x 的父节点 x: p 可以多加一个黑色。第 11 行通过将 x 上移一级来实现这一点，这样 while 循环以 x: p 作为新节点 x 重复进行。如果情况 2 通过情况 1 进入，则新节点 x 为红黑相间，因为原始 x: p 为红色。因此，新节点 x 的 *color* 属性的值为红色，循环在测试循环条件时终止。然后第 44 行将新节点 x（单独）染成黑色。

Case3: x***'s sibling*** w ***is black,*** w***'s left child is red, and*** w***'s right child is black*** 情况 3（第 143-17 行和图 13.7(c)）发生在 w 为黑色、其左子节点为红色、其右子节点为黑色时。此情况交换了 w 及其左子节点 w 的颜色：*left*，然后对 w 执行右旋转，但不违反任何红黑性质。x 的新兄弟 w 现在是一个黑色节点，其右子节点为红色，因此情况 3 落入情况 4。

Case4: 横***'s sibling*** 横***is black, and*** 横***'s right child is red***
情况 4（第 18322 行和图 13.7(d)）发生在节点 x 的兄弟节点 w 为黑色，而 w 的右子节点为红色时。对 x 进行一些颜色更改和左旋转：p 可使 x 上的额外黑色消失，使其成为单黑色，而不会违反任何红黑性质。第 22 行将 x 设置为根，当下一次测试循环条件时，while 循环终止。

分析

RB-DELETE 的运行时间是多少？由于 n 个节点的红黑树的高度为 O.lg n/，因此不调用 RB-DELETE-FIXUP 的情况下，整个过程的总成本为 O.lg n/ 时间。在 RB-DELETE-FIXUP 中，情况 1、3 和 4 均在执行恒定次数的颜色变化和最多三次旋转后导致终止。情况 2 是唯一可以重复 while 循环的情况，然后指针 x 最多沿树向上移动 O.lg n/ 次，不执行任何旋转。因此，过程 RB-DELETE-FIXUP 需要 O.lg n/ 时间并最多执行三次旋转，因此 RB-DELETE 的总时间也是 O.lg n/。

练习

13.4-1
证明如果 RB-DELETE 中的节点 y 为红色，则黑色高度不会发生变化。

13.4-2
论证说RB-DELETE-FIXUP执行后，树的根必定是黑色的。

13.4-3
争论说如果在 RB-DELETE 中 x 和 x: p 都是红色，那么通过调用 RB-DELETE-FIXUP .T; x/ 可以恢复属性 4。

13.4-4
在第 346 页的练习 13.3-2 中，你找到了将键 41、38、31、12、19、8 连续插入到一棵最初为空的树中后得到的红黑树。现在请展示按 8、12、19、31、38、41 的顺序连续删除键后得到的红黑树。

13.4-5
RB-DELETE-FIXUP 代码的哪些行可能会检查或修改标记 T：*nil*？

13.4-6
在图 13.7 的每种情况下，给出从所示子树的根到每个子树 $\alpha\beta$: : : ;　的根的黑色节点数，并验证每个计数在转换后是否保持不变。当节点具有 *color* 属性 c 或 c′ 时，请在计数中使用符号 count.c/ 或 count.c′/。

13.4-7
Skelton 和 Baron 教授担心，在 RB-DELETE-FIXUP 的第 1 个案例开始时，节点 x: p 可能不是黑色。如果 x: p 不是黑色，那么第 536 行

错误。说明 x: *p* 在案例 1 的开头必须是黑色，这样教授们就不必担心了。

13.4-8
使用 RB-I NSERT 将节点 x 插入红黑树，然后立即使用 RB-DELETE 将其删除。生成的红黑树是否始终与初始红黑树相同？证明你的答案。

13.4-9
考虑操作 RB-ENUMERATE .T; r; a; b/，该操作输出 n 节点红黑树 T 中以节点 r 为根的子树中的所有键 k，使得 a # k # b。描述如何在 ，.m C lg n/ 时间内实现 RB-ENUMERATE，其中 m 是输出的键数。假设 T 中的键是唯一的，并且值 a 和 b 作为键出现在 T 中。如果 a 和 b 可能不出现在 T 中，您的解决方案会如何变化？

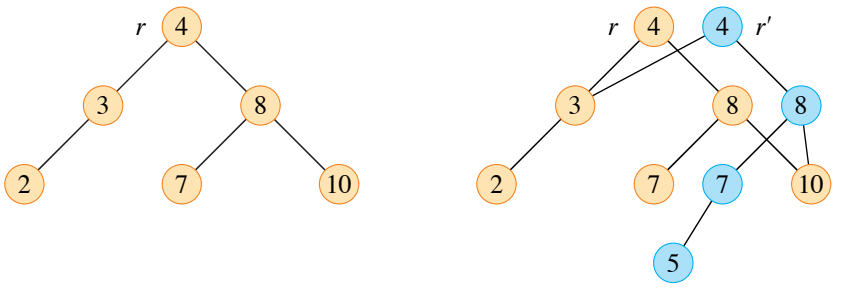
问题

13-1Persistent dynamic sets
在算法过程中，您有时会发现需要在更新动态集时维护其过去的版本。我们将此类集合称为 ***persistent***。实现持久集的一种方法是每次修改时复制整个集合，但这种方法会降低程序速度并消耗大量空间。有时，您可以做得更好。

考虑一个持久集合 S，它具有插入、删除和搜索操作，可以使用二叉搜索树实现这些操作，如图 13.8(a) 所示。为集合的每个版本维护一个单独的根。为了将键 5 插入到集合中，创建一个键为 5 的新节点。由于不能修改键为 7 的现有节点，因此该节点将成为键为 7 的新节点的左子节点。类似地，键为 7 的新节点将成为键为 8 的新节点的左子节点，而该新节点的右子节点是键为 10 的现有节点。键为 8 的新节点又将成为键为 4 的新根 r ′ 的右子节点，该新根的左子节点是键为 3 的现有节点。因此，只复制树的一部分并与原始树共享一些节点，如图 13.8(b) 所示。

假设每个树节点都有属性 *key*、*left* 和 *right* ，但没有父节点。（另请参见第 346 页的练习 13.3-6。）

a.对于持久二叉搜索树（不是红黑树，只是二叉搜索树），确定插入或删除节点时需要更改的节点。



(a) (b)

图 13.8 (a) 包含键 2；3；4；7；8；10 的二叉搜索树。(b) 插入键 5 后生成的持久二叉搜索树。集合的最新版本由可从根 r′ 到达的节点组成，而先前版本由可从 r 到达的节点组成。插入键 5 时会添加蓝色节点。

b.编写一个过程 PERSISTENT-TREE-I NSERT .T; *k*, 给定一个持久二叉搜索树 T 和一个要插入的节点 *z*, 返回一个新的持久树 T′，它是将 *z* 插入 T 的结果。假设您有一个过程 COPY-NODE.x/，它复制节点 x，包括其所有属性。

c.如果持久二叉搜索树 T 的高度为 h，你实现 P ERSISTENT-TREE-INSERT 的时间和空间要求是多少？（空间要求与复制的节点数成正比。）

d.假设您在每个节点中都包含父属性。在这种情况下，PERSISTENT-TREE-I NSERT 过程需要执行额外的复制。证明 PERSISTENT-TREE-INSERT 需要 .n/ 时间和空间，其中 n 是树中的节点数。

e.说明如何使用红黑树来保证每次插入或删除的最坏情况运行时间和空间为 O.lg n/。您可以假设所有键都是不同的。

13-2Join operation on red-black trees

join 运算采用两个动态集合 S_1 和 S_2 以及一个元素 x，使得对于任何 x_1 2 S_1 和 x_2 2 S_2，我们都有 x_1: *key* x: *key* x_2: *key*。它返回一个集合 S D S_1 [fx g [S_2。在本题中，我们研究如何在红黑树上实现连接运算。

a.假设你将红黑树 T 的黑色高度存储为新属性 T: *bh*。争论 RB-I NSERT 和 R B-DELETE 可以维护 *bh*

属性，而无需在树的节点中额外存储，也不会增加渐近运行时间。说明如何确定在 T 中下降时访问的每个节点的黑色高度，每个访问节点的时间是 O.1/。

假设 T_1 和 T_2 为红黑树，x 为键值，使得对于 T_1 中的任何节点 x_1 和 T_2 中的任何节点 x_2，我们都有 x_1: *key* x: *key* x_2: *key*。您将展示如何实现操作 RB-J OIN.T_1; x; T_2/，该操作会销毁 T_1 和 T_2 并返回红黑树 T D T_1 [fx g [T_2。假设 n 为 T_1 和 T_2 中的节点总数。

b.假设 T_1: *bh* ≈ T_2: *bh*。描述一个 O.lg n/ 次算法，该算法在 T_1 中从黑色高度为 T_2: *bh* 的节点中找到具有最大键值的黑色节点 y。***c.*** 令 T_y 为以 y 为根的子树。描述如何在 O.1/ 次内替换 T_y [fx g [T_2 而不破坏二叉搜索树属性。***d.*** 应将 x 设为什么颜色，以保持红黑属性 1、3 和 5？描述如何在 O.lg n / 次内强制执行属性 2 和 4。***e.*** 论证做出第 (b) 部分中的假设不会丧失一般性。描述当 T_1: *bh* T_2: *bh* 时出现的对称情况。

f.论证RB-J OIN的运行时间为O.lg n/。

13-3AVL trees

AVL tree 是一棵二叉搜索树，其满足 ***height balanced***：对于每个节点 x，x 的左子树和右子树的高度最多相差 1。要实现 AVL 树，需要在每个节点中维护一个额外的属性 h，使得 x: *h* 是节点 x 的高度。对于任何其他二叉搜索树 T，假设 T: *root* 指向根节点。

a.证明具有 n 个节点的 AVL 树高度为 O.lg n/。（*Hint:* 证明高度为 h 的 AVL 树至少具有 F_h 个节点，其中 F_h 是第 h 个斐波那契数。）

b.要插入 AVL 树，首先将节点放入二叉搜索树顺序中的适当位置。之后，树可能不再是高度平衡的。具体来说，某个节点的左子节点和右子节点的高度可能相差 2。描述一个过程 BALANCE.x/，它取一个以 x 为根的子树，其左子节点和右子节点高度平衡，并且高度不同

最多 2，因此 jx: *right* : *h* x: *left*: *h*j # 2，并修改以 x 为根的子树，使其高度平衡。该过程应返回一个指向发生更改后子树根节点的指针。（*Hint:* 使用旋转。）

c.使用部分 (b)，描述递归过程 AVL-I NSERT .T; *k*，该过程采用 AVL 树 T 和新创建的节点 *z*（其键已填入），并将 *z*添加到 T 中，保持 T 是 AVL 树的属性。与第 12.3 节中的 TREE-I NSERT 一样，假设 *zkey* 已填入，并且 *zleft* D NIL 和 *zright* D NIL。还假设 *zh* D 0。

d.说明 AVL-I NSERT 在 n 节点 AVL 树上运行，需要 O.lg n/ 时间并执行 O.lg n/ 次旋转。

章节注释

平衡搜索树的想法是由 Adel ’ son-Vel ’ skiˇ ı 和 Landis [2] 提出的，他们在 1962 年引入了一类称为 “AVL 树” 的平衡搜索树，如问题 13-3 中所述。另一类搜索树称为 “2-3 树”，由 J. E. Hopcroft（未发表）于 1970 年引入。2-3 树通过操纵树中节点的度来保持平衡，其中每个节点都有两个或三个子节点。第 18 章介绍了 Bayer 和 McCreight [39] 引入的 2-3 树的泛化，称为 “B 树。”

红黑树是由 Bayer [38] 发明的，当时的名称是 “对称二叉 B 树。” Guibas 和 Sedgewick [202] 深入研究了红黑树的性质，并引入了红/黑颜色约定。Andersson [16] 给出了一种更易于编码的红黑树变体。Weiss [451] 将此变体称为 AA 树。AA 树类似于红黑树，只是左子树永远不能是红色。

Sedgewick 和 Wayne [402] 提出红黑树是 2-3 树的改良版本，其中具有三个子节点的节点被拆分为两个节点，每个节点具有两个子节点。其中一个节点成为另一个节点的左子节点，并且只有左子节点可以是红色的。他们将这种结构称为 “ 左倾红黑二叉搜索树。” 虽然左倾红黑二叉搜索树的代码比本章中的红黑树伪代码更简洁，但左倾红黑二叉搜索树上的操作不会将每次操作的旋转次数限制为常数。这一区别将在第 17 章中发挥作用。

Treap 是二叉搜索树和堆的混合体，由 Seidel 和 Aragon [404] 提出。它们是 LEDA [324] 中字典的默认实现，LEDA 是一个实现良好的数据结构和算法集合。

平衡二叉树还有许多其他变体，包括权重平衡树[344]、k 邻居树[318]和替罪羊树[174]。也许

最有趣的是 Sleator 和 Tarjan [418] 引入的 “伸展树”，它们具有“自我调整的功能。”（有关伸展树的详细描述，请参阅 Tarjan [429]。）伸展树不需要任何明确的平衡条件（如颜色）就能保持平衡。相反，每次访问时都会在树内执行“伸展操作”（涉及旋转）。对一棵 n 节点树执行每个操作的摊销成本（见第 16 章）为 O.lg n/。据推测，伸展树的性能与最佳的基于旋转的树相差无几。已知最好的基于旋转的树的竞争比（见第 27 章）是 Demaine 等人的 Tango 树 [109]。

跳跃表 [369] 为平衡二叉树提供了一种替代方案。跳跃表是一个链表，其中增加了许多附加指针。在包含 n 个项目的跳跃表上，每个字典操作的运行时间为 O.lg n/ 预期时间。

Part IV Advanced Design and Analysis Techniques

介绍

本部分涵盖了设计和分析高效算法的三种重要技术：动态规划（第 14 章）、贪婪算法（第 15 章）和摊销分析（第 16 章）。前面的部分介绍了其他广泛适用的技术，例如分而治之、随机化以及如何解决递归。本部分中的技术有些复杂，但您将能够使用它们解决许多计算问题。本部分介绍的主题将在本书后面再次出现。

动态规划通常适用于优化问题，在此类问题中，您需要做出一系列选择才能获得最佳解决方案，每个选择都会生成与原始问题形式相同的子问题，并且相同的子问题会重复出现。关键策略是存储每个此类子问题的解，而不是重新计算它。第 14 章展示了这个简单的想法如何有时将指数时间算法转变为多项式时间算法。

与动态规划算法一样，贪婪算法通常适用于优化问题，即您需要做出一系列选择才能获得最佳解决方案。贪婪算法的理念是以局部最优的方式做出每个选择，从而产生比动态规划更快的算法。第 15 章将帮助您确定贪婪方法何时有效。

摊销分析技术适用于执行一系列类似操作的某些算法。摊销分析不是通过分别限制每个操作的实际成本来限制操作序列的成本，而是对整个序列的实际成本提供最坏情况的约束。这种方法的一个优点是，尽管某些操作可能很昂贵，但许多其他操作可能很便宜。您可以在设计算法时使用摊销分析，因为算法的设计和其运行时间的分析通常紧密相关。第 16 章介绍了对算法进行摊销分析的三种方法。

14 Dynamic Programming

动态规划与分治法类似，通过组合子问题的解来解决问题。（此处的“编程”是指表格方法，而不是编写计算机代码。）正如我们在第 2 章和第 4 章中看到的那样，分治算法将问题划分为不相交的子问题，以递归方式解决子问题，然后组合它们的解来解决原始问题。相反，动态规划适用于子问题重叠的情况4，即子问题共享子子问题。在这种情况下，分治算法会做比必要更多的工作，反复解决共同的子子问题。动态规划算法只解决每个子子问题一次，然后将其答案保存在表中，从而避免了每次解决每个子子问题时重新计算答案的工作。

动态规划通常适用于 ***optimization problems***。此类问题可能有许多可能的解决方案。每个解决方案都有一个值，您希望找到一个具有最优（最小或最大）值的解决方案。我们将这样的解决方案称为问题的 *an* 最优解决方案，而不是 *the* 最优解决方案，因为可能有多个解决方案可以实现最优值。

要开发动态规划算法，请遵循以下四个步骤：

1. 描述最优解的结构。 2. 递归地定义最优解的值。

3. 计算最优解的值，通常采用自下而上的方式。

4. 根据计算信息构建最佳解决方案。

步骤 133 构成了问题动态规划解决方案的基础。如果您只需要最优解的值，而不是解本身，则可以省略步骤 4。当您执行步骤 4 时，在步骤 3 期间保留其他信息通常是有益的，这样您就可以轻松构建最优解。

接下来的章节使用动态规划方法解决一些优化问题。第 14.1 节探讨了将一根杆切割成

以最大化其总价值的方式将长度较小的杆分成多个小杆。第 14.2 节展示了如何在执行最少的总标量乘法的同时对矩阵链进行乘法。根据这些动态规划示例，第 14.3 节讨论了问题必须具备的两个关键特征，以使动态规划成为可行的解决方案。然后，第 14.4 节展示了如何通过动态规划找到两个序列的最长公共子序列。最后，第 14.5 节使用动态规划构造二叉搜索树，这些树在已知要查找的键的分布的情况下是最优的。

14.1 棒材切割

我们的第一个示例使用动态规划来解决一个简单的问题，即决定在哪里切割钢棒。Serling Enterprises 购买长钢棒，然后将其切成较短的钢棒，然后出售。每次切割都是免费的。Serling Enterprises 的管理层想知道切割钢棒的最佳方法。

Serling Enterprises 有一张表格，其中给出了 i D 1; 2; : : : 长度为 i 英寸的杆的价格 p_i（以美元计）。每根杆的长度（以英寸为单位）始终为整数。图 14.1 给出了一个示例价格表。

rod-cutting problem 如下。给定一根长度为 n 英寸的杆和一个价格表 p_i，其中 i D 1; 2; : : : ; n，确定通过切割杆并出售碎片可获得的最大收入 r_n。如果长度为 n 的杆的价格 p_n 足够大，则最佳解决方案可能根本不需要切割。

考虑 n D 4 的情况。图 14.2 显示了切割 4 英寸长杆的所有方法，包括完全不切割的方法。将 4 英寸长的杆切成两段 2 英寸长的杆可产生收益 $p_2 C p_2$ D 5 C 5 D 10，这是最优的。

Serling Enterprises 可以将一根长度为 n 的棒切成 2^{n-1} 种不同的方式，因为它们可以独立选择在距左端 i 英寸处切割或不切割，其中 i D 1; 2; : : : ; n 1 .[1] 我们使用普通的加法符号表示分解成碎片，因此 7 D 2 C 2 C 3 表示将长度为 7 的棒切成三段4两段长度为 2 一段长度为 3 . 如果最优解将棒切成 k 段，对于某个 1 k n，则最优分解

$$n = i_1 + i_2 + \cdots + i_k$$

[1] If pieces are required to be cut in order of monotonically increasing size, there are fewer ways to consider. For $n = 4$, only 5 such ways are possible: parts (a), (b), (c), (e), and (h) in Figure 14.2. The number of ways is called the ***partition function***, which is approximately equal to $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$. This quantity is less than 2^{n-1}, but still much greater than any polynomial in n. We won't pursue this line of inquiry further, however.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

图 14.1 棒材价格表样本。每根长度为 i 英寸的棒材为公司带来 p $_i$ 美元的收益。

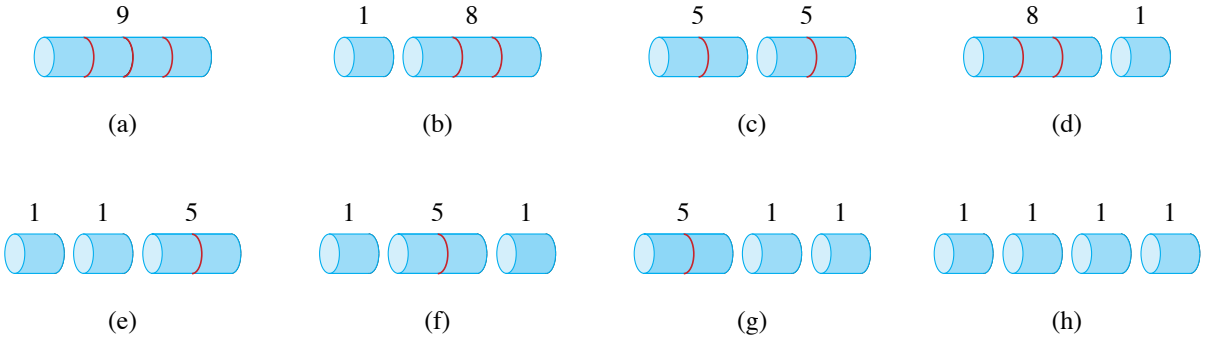


图 14.2 切割长度为 4 的杆的 8 种可能方法。根据图 14.1 的示例价格表，每段上方是该段的价值。最佳策略是 (c)4 部分将杆切成两段，长度为 24，总价值为 10。

将杆切成长度为 i $_1$, i $_2$, . . . , i $_k$ 的段，可获得最大相应收入

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} .$$

对于图 14.1 中的示例问题，您可以通过检查确定 i D 1; 2; : : : ; 10 的最佳收入数字 r $_i$ ，以及相应的最佳分解

r 1 D 1 来自解决方案 1 D 1 (无切割) ；r $_2$ D 5 来自解决方案 2 D 2 (无切割) ；r $_3$ D 8 来自解决方案 3 D 3 (无切割) ；r $_4$ D 10 来自解决方案 4 D 2 C 2 ；r $_5$ D 13 来自解决方案 5 D 2 C 3 ；r $_6$ D 17 来自解决方案 6 D 6 (无切割) ；r $_7$ D 18 来自解决方案 7 D 1 C 6 或 7 D 2 C 2 C 3 ；r $_8$ D 22 来自解决方案 8 D 2 C 6 ；r $_9$ D 25 来自解决方案 9 D 3 C 6 ； r $_{10}$ D 30 来自解决方案 10 D 1 0 （无切割） ：

更一般地，我们可以用较短杆的最佳收益来表示 n 1 的值 r_n ：

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\} . \tag{14.1}$$

第一个参数 p_n 对应于不进行任何切割并按原样出售长度为 n 的杆。max 的另外 n 1 个参数对应于将杆初次切割成大小为 i 和 n 的两段，每段 i = 1; 2; : : :; n 1，然后以最优方式进一步切割这些段，从这两段获得收入 r_i 和 r_{n-i}。由于你事先不知道哪个 i 值可以优化收入，因此你必须考虑 i 的所有可能值并选取使收入最大化的那个。如果最大收入来自出售未切割的杆，你也可以选择不选取 i。

要解决原始的 n 级问题，您需要解决相同类型的较小问题。一旦您进行了第一次切割，得到的两个部分就构成了杆切割问题的独立实例。整体最优解决方案结合了两个子问题的最优解决方案，从而最大限度地提高了这两个部分的收益。我们称杆切割问题表现出 ***optimal sub- structure***：问题的最优解决方案结合了相关子问题的最优解决方案，您可以独立解决这些子问题。

为解决杆切割问题，我们采用一种相关但更简单的递归结构，将杆的分解看作由从左端切下长度为 i 的第一段，然后切下长度为 n 的右端余段组成。只有余段可以进一步分割，而第一段不能。将长度为 n 的杆的每次分解视为：先分解第一段，然后分解余段。然后，我们可以表示完全没有切割的解，即第一段的大小为 i D n，收入为 p_n，余段的大小为 0，对应的收入为 r_0 D 0。因此，我们得到方程 (14.1) 的以下简化版本：

$$r_n = \max\{p_i + r_{n-i} : 1 \le i \le n\} . \tag{14.2}$$

在这个公式中，最优解只体现了对*one* 相关子问题4的解，而不是对余数4的解。

自上而下递归实现

下页的 CUT-ROD 程序以直接、自上而下的递归方式实现了方程 (14.2) 中隐含的计算。它以价格数组 pŒ1 W n 和一个整数 n 作为输入，并返回长度为 n 的杆可能产生的最大收益。对于长度 n D 0，不可能产生任何收益，因此 CUT-ROD 在第 2 行返回 0。第 3 行将最大收益 q 初始化为 1，以便第 4 至第 3 行的 for 循环正确计算

q D max fp $_i$ C CUT-ROD.p; n i / W 1 i ng。然后第 6 行返回此值。对 n 进行简单的归纳证明，使用公式 (14.2)，此答案等于所需答案 r $_n$ 。

```
切割杆.p; n/
1 如果 n ==0 2 返回 0 3 q D 1 4 对于 i D 1 至
n 5 q D max fq; pŒi C CUT-ROD.p; n i /g 6
返回 q
```

如果您用自己喜欢的编程语言编写 CUT-ROD 并在计算机上运行它，您会发现一旦输入大小变得相当大，您的程序就会需要很长时间才能运行。对于 n D 40，您的程序可能需要几分钟甚至一个多小时。对于较大的 n 值，您还会发现每次将 n 增加 1，您的程序的运行时间大约增加一倍。

为什么 CUT-ROD 效率如此低下？问题在于 CUT-ROD 使用相同的参数值一遍又一遍地递归调用自身，这意味着它重复解决相同的子问题。图 14.3 显示了一棵递归树，演示了当 n D 4 时发生的情况：CUT-ROD.p；n/ 对 i D 1；2；：：：；n 调用 CUT-ROD.p；n i /。等效地，CUT-ROD.p；n/ 对每个 j D 0；1；：：：；n 调用 CUT-ROD.p；j /。当此过程递归展开时，作为 n 的函数，所做的工作量会爆炸式增长。

为了分析 CUT-ROD 的运行时间，让 T.n/ 表示对 CUT-ROD.p;n/ 进行的总调用次数，其中 n 为特定值。此表达式等于递归树中根标记为 n 的子树中的节点数。计数包括其根处的初始调用。因此，T.0/D1 和

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) . \tag{14.3}$$

初始的 1 表示根处的调用，项 T .j / 计算由于调用 CUT-ROD.p; n i / 而引起的调用次数（包括递归调用），其中 j D n i 。正如练习 14.1-1 要求您证明的那样，

$$T(n) = 2^n , \tag{14.4}$$

所以 CUT-ROD 的运行时间是 n 的指数。

回想起来，这个指数级的运行时间并不那么令人惊讶。CUT-ROD 明确考虑了切割长度为 n 的杆的所有可能方法。有多少种方法

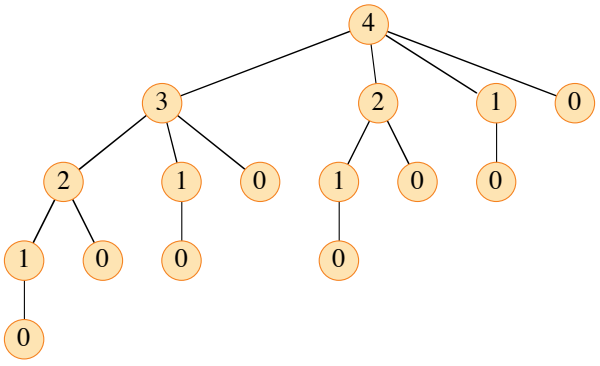


图 14.3 递归树显示了对 n D 4 的调用 C UT-ROD.p; n/ 所导致的递归调用。每个节点标签都给出了相应子问题的大小 n，因此从标签为 s 的父节点到标签为 t 的子节点的一条边对应于切断大小为 s 的初始部分并留下大小为 t 的剩余子问题。从根到叶的路径对应于切断长度为 n 的杆的 2^{n-1} 种方法之一。通常，此递归树有 2^n 个节点和 2^{n-1} 个叶。

有多少个？一根长度为 n 的棒有 n 1 个潜在的可切割位置。每种可能的切割方式都会在这些 n 1 个位置的某个子集上进行切割，包括空集（即无法切割）。将每个切割位置视为 n 1 个元素集合的不同成员，可发现有 2^{n-1} 个子集。图 14.3 的递归树中的每一片叶子都对应着一种切割棒的可能方式。因此，递归树有 2^{n-1} 片叶子。从根到叶子的简单路径上的标签给出了每次切割之前剩余的每个右侧部分的大小。也就是说，标签给出了相应的切割点，从棒的右端测量。

使用动态规划进行最优杆切割

现在，我们来看看如何使用动态规划将C UT-ROD转换成一个有效的算法。

动态规划方法的工作原理如下。与简单递归解决方案不同，动态规划方法不会重复解决相同的子问题，而是将每个子问题安排为 *only once* 进行解决。实际上，有一种显而易见的方法可以做到这一点：第一次解决子问题时，*save its solution*。如果您以后需要再次参考此子问题的解决方案，只需查找它，而不是重新计算它。

保存子问题解决方案需要付出代价：存储解决方案所需的额外内存。因此，动态规划可作为 ***time- memory trade-off*** 的示例。节省的成本可能非常可观。例如，我们即将使用动态规划从指数时间算法开始进行杆切割

归结为 ，.n 2 / 时间算法。当所涉及的 *distinct* 子问题数量是输入大小的多项式时，动态规划方法可以在多项式时间内运行，并且您可以在多项式时间内解决每个这样的子问题。

通常有两种等效的方法来实现动态规划方法。杆切割问题的解法就说明了这两种方法。

第一种方法是 ***top-down*** 和 ***memoization***。[2] 在这种方法中，您以自然的方式递归编写过程，但经过修改以保存每个子问题的结果（通常在数组或哈希表中）。现在，该过程首先检查它是否以前解决过这个子问题。如果是，它会返回保存的值，从而节省此级别的进一步计算。如果没有，该过程将以通常的方式计算该值，但也会保存它。我们称递归过程为 ***memoized***：它 “ 记住” 以前计算过的结果。

第二种方法是 ***bottom-up method***。这种方法通常依赖于子问题 “ 大小 ” 的一些自然概念，这样解决任何特定子问题都只依赖于解决 “ 较小 ” 的子问题。按大小顺序解决子问题，最小子问题优先，在第一次解决每个子问题时存储其解决方案。这样，在解决特定子问题时，已经保存了其解决方案所依赖的所有较小子问题的解决方案。您只需解决每个子问题一次，当您第一次看到它时，您已经解决了其所有先决条件子问题。

这两种方法产生的算法具有相同的渐近运行时间，除非在特殊情况下，自上而下的方法实际上不会递归检查所有可能的子问题。自下而上的方法通常具有更好的常数因子，因为它的过程调用开销较低。

对页上的 MEMOIZED-CUT-ROD 和 MEMOIZED-CUT-ROD-AUX 过程演示了如何记忆自上而下的 CUT-ROD 过程。主过程 MEMOIZED-CUT-ROD 用值 1 初始化一个新的辅助数组 rŒ0Wn ，由于已知的收入值始终为非负数，因此它是表示 “unknown” 的方便选择。然后，MEMOIZED-CUT-ROD 调用其辅助过程 MEMOIZED-CUT-ROD-AUX，它只是指数时间过程 CUT-ROD 的记忆版本。它首先在第 1 行检查所需值是否已知，如果是，则第 2 行返回该值。否则，第 337 行以通常的方式计算所需值 q，第 8 行将其保存在 r Œn 中，第 9 行返回它。

自下而上的版本，即下一页的 BOTTOM-UP-CUT-ROD，甚至更简单。使用自下而上的动态规划方法，BOTTOM-UP-CUT-ROD 利用了子问题的自然排序：

---

[2] The technical term “memoization” is not a misspelling of “memorization.” The word “memoization” comes from “memo,” since the technique consists of recording a value to be looked up later.

记忆切割杆 .p; n/

1 让 r Œ0 W n 成为一个新的数组 //，它将记住 r 中的解决方案值
2 对于 i D 0 到 n 3 r Œi D 1 4 返回 MEMOIZED-CUT-ROD-AU
X .p; n; r /

记忆切割杆辅助 .p; n; r /

1 if r Œn 0 // 已经有长度为 n 的解吗？ 2 return r Œn 3 if n == 0 4 q
D 0 5 else q D 1 6 for i D 1 to n // i 是第一个切口的位置 7 q D max fq;
pŒi C MEMOIZED-CUT-ROD-AUX .p; n i; r /g 8 r Œn D q //
记住长度为 n 的解的值 9 return q

自下而上切割杆 .p; n/

```
1  let r[0 : n] be a new array          // will remember solution values in r
2  r[0] = 0
3  for j = 1 to n                       // for increasing rod length j
4      q = −∞
5      for i = 1 to j                   // i is the position of the first cut
6          q = max {q, p[i] + r[j − i]}
7      r[j] = q                         // remember the solution value for length j
8  return r[n]
```

如果 i < j，则大小 i 比大小为 j 的子问题“小”。因此，该过程按顺序求解大小为 j 的子问题 D 0; 1; : : : ; n。

BOTTOM-UP-CUT-ROD 的第 1 行创建了一个新的数组 rŒ0Wn 来保存子问题的结果，第 2 行将 rŒ0 初始化为 0，因为长度为 0 的杆不赚取任何收入。第 3 至第 6 行按大小递增的顺序求解每个大小为 j 的子问题，其中 jD 1；2；：：：；n。解决特定大小 j 的问题所用的方法与 CUT-ROD 所用的方法相同，只是第 6 行现在直接引用数组条目 rŒj i ，而不是进行递归调用来解决大小为 j 的子问题。第 7 行将大小为 j 的子问题的解保存在 rŒj 中。最后，第 8 行返回 rŒn ，它等于最优值 r_n。

自下而上和自上而下的版本具有相同的渐近运行时间。BOTTOM-UP-CUT-ROD 的运行时间为 ，.n ²/，因为它是双重嵌套的

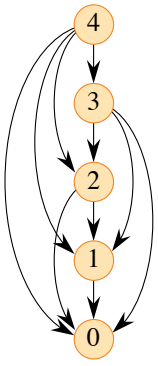


图 14.4 n D 4 的杆切割问题的子问题图。顶点标签给出了相应子问题的大小。有向边 .x; y/ 表示解决子问题 x 需要解决子问题 y。此图是图 14.3 中递归树的简化版本，其中所有具有相同标签的节点都折叠为单个顶点，所有边都从父节点指向子节点。

循环结构。其内部 for 循环的迭代次数（第 536 行）形成一个算术级数。其自上而下的对应部分 MEMOIZED-CUT-ROD 的运行时间也是 ，.n 2/，尽管这个运行时间可能有点难以看出。因为解决先前解决的子问题的递归调用会立即返回，所以 MEMOIZED-CUT-ROD 只解决每个子问题一次。它解决大小为 0；1；:::；n 的子问题。为了解决大小为 n 的子问题，第 637 行的 for 循环迭代 n 次。因此，在 MEMOIZED-CUT-ROD 的所有递归调用中，此 for 循环的总迭代次数形成一个算术级数，总共迭代 ，.n 2/ 次，就像 BOTTOM-UP-CUT-ROD 的内部 for 循环一样。（我们实际上在这里使用了一种聚合分析形式。我们将在 16.1 节中详细了解聚合分析。）

子问题图

当你考虑动态规划问题时，你需要了解所涉及的子问题集以及子问题如何相互依赖。

问题的 ***subproblem graph*** 恰恰体现了这一信息。图 14.4 显示了 n D 4 的杆切割问题的子问题图。它是一个有向图，每个不同的子问题都有一个顶点。如果确定子问题 x 的最优解涉及直接考虑子问题 y 的最优解，则子问题图具有从子问题 x 的顶点到子问题 y 的顶点的有向边。例如，如果自上而下的求解 x 的递归过程直接调用自身来求解 y ，则子问题图包含从 x 到 y 的边。你可以将子问题图视为

自上而下递归方法的递归树的 “reduced” 或 “collapsed” 版本，其中同一子问题的所有节点合并为一个顶点，所有边都从父节点指向子节点。

自下而上的动态规划方法考虑子问题图的顶点的顺序是，在解决子问题 x 之前，先解决与给定子问题 x 相邻的子问题 y。（如 B.4 节所述，有向图中的邻接关系不一定是对称的。）使用我们将在 20.4 节中看到的术语，在自下而上的动态规划算法中，考虑子问题图的顶点的顺序是 “ 逆拓扑排序，” 或 “ 子问题图转置的拓扑排序”。换句话说，在解决所有子问题之前，不会考虑任何子问题。类似地，使用我们将在第 20.3 节中讨论的概念，你可以将动态规划的自上而下方法（带有记忆）视为子问题图的“深度优先搜索”。

子问题图 G D .V; E/ 的大小可以帮助你确定动态规划算法的运行时间。由于你只解决每个子问题一次，因此运行时间是解决每个子问题所需时间的总和。通常，计算子问题解的时间与子问题图中相应顶点的度（出边数）成正比，子问题的数量等于子问题图中顶点的数量。在这种常见情况下，动态规划的运行时间与顶点和边的数量成线性关系。

重建解决方案

程序 MEMOIZED-CUT-ROD 和 BOTTOM-UP-CUT-ROD 返回棒切割问题的最佳解决方案 *value*，但它们不返回解决方案 *itself* ：零件尺寸列表。

让我们看看如何扩展动态规划方法，不仅记录为每个子问题计算出的最优 *value*，还记录导致最优值的 *choice*。有了这些信息，您可以轻松打印出最优解决方案。下一页的 EXTENDED-BOTTOM-UP-CUT-ROD 程序不仅计算每个杆尺寸 j 的最大收益 r_j，还计算 s_j，即要切断的第一块的最佳尺寸。它类似于 BOTTOM-UP-CUT-ROD，不同之处在于它在第 1 行创建数组 s，并在第 8 行更新 sŒj ，以保存在解决尺寸为 j 的子问题时要切断的第一块的最佳尺寸 i。

下页的 PRINT-CUT-ROD-SOLUTION 程序以价格数组 pŒ1 W n 和杆尺寸 n 作为输入。它调用 EXTENDED-BOTTOM-UP-CUT-ROD 来计算最优第一件尺寸数组 sŒ1 W n 。然后，它打印出最优分解中完整的件尺寸列表

长度为 n 的杆。对于图 14.1 中出现的示例价格图表，调用 EXTENDED-BOTTOM-UP-CUT-ROD .p; 10/ 返回以下数组：

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$		1	2	3	2	2	6	1	2	3	10

调用 PRINT-CUT-ROD-SOLUTION .p; 10/ 只会打印 10，但调用 n D 7 会打印切割 1 和 6，这对应于先前给出的 r_7 的第一个最优分解。

延伸底部向上切割杆 .p; n/

1 令 r Œ0 W n 和 sŒ1 W n 为新数组 2 r Œ0 D 0 3 对于 j D 1 至 n //，杆长度增加 j 4 q D 1 5 对于 i D 1 至 j //，i 为第一次切割的位置 6 如果 q < pŒi C r Œj i 7 q D pŒi C r Œj i 8 sŒj D i // 长度 j 的最佳切割位置 9 r Œj D q // 记住长度 j 的解决方案值 10 返回 r 和 s

打印-切割-棒-解决方案 .p; n/

1 .r; s/ D 延伸-底部-向上-切割-杆 .p; n/ 2 while n > 0 3 打印 sŒn // 切割位置的长度 n 4 n D n sŒn // 杆剩余部分的长度

练习

14.1-1
证明方程 (14.4) 由方程 (14.3) 和初始条件 T .0/ D 1 得出。

14.1-2
通过反例说明，以下 “贪婪” 策略并不总是确定切割棒的最佳方法。将长度为 i 的棒的 ***density*** 定义为 p_i / i，即其每英寸的价值。长度为 n 的棒的贪婪策略会切下第一段长度为 i 的棒，其中 1 = i = n，具有最大

密度。然后继续将贪婪策略应用于长度为 n 的剩余部分。

14.1-3
考虑对杆切割问题的一种修改，其中除了每根杆的价格 p_i 之外，每次切割还会产生固定成本 c 。与解决方案相关的收益现在是各部分价格的总和减去切割成本。给出一个动态规划算法来解决这个修改后的问题。

14.1-4
修改 CUT-ROD 和 MEMOIZED-CUT-ROD-AUX，使它们的 for 循环最多只到 bn/2c，而不是最多到 n。你还需要对程序进行哪些其他更改？它们的运行时间会受到怎样的影响？

14.1-5
修改记忆 ZED-CUT-ROD 不仅返回值，还返回 实际的解决方案。

14.1-6
斐波那契数列由第 69 页的递归 (3.31) 定义。给出一个 O.n/-time 动态规划算法来计算第 n 个斐波那契数列。画出子问题图。该图包含多少个顶点和边？

14.2 矩阵链乘法

下一个动态规划示例是解决矩阵链乘法问题的算法。给定一个序列（链）hA 1; A 2; : : : ; A n i，其中有 n 个矩阵需要相乘，其中矩阵不一定是方阵，目标是计算乘积

$$A_1 A_2 \cdots A_n \ . \tag{14.5}$$

使用标准算法 [3] 来乘以矩形矩阵，我们稍后会看到，同时最小化标量乘法的次数。

一旦你用括号括起来，就可以使用乘以矩形矩阵对的算法作为子程序来计算表达式 (14.5)，以解决矩阵相乘的所有歧义。矩阵乘法是结合的，因此所有括号都会产生相同的乘积。

[3] None of the three methods from Sections 4.1 and Section 4.2 can be used directly, because they apply only to square matrices.

如果矩阵是单个矩阵或两个完全用括号括起来的矩阵乘积的乘积，则矩阵是***fully parenthesized***。例如，如果矩阵链是 hA $_1$; A $_2$; A $_3$; A $_4$i，那么可以用五种不同的方式将乘积完全用括号括起来：

.A $_1$ .A $_2$ .A $_3$ A $_4$ ///
; .A $_1$ ..A $_2$ A $_3$ /A $_4$
// ; ..A $_1$ A $_2$ /.A $_3$ A
$_4$ // ; ..A $_1$ .A $_2$ A $_3$ //
A $_4$ / ; ...A $_1$ A $_2$ /A
$_3$ /A $_4$ / .

如何为矩阵链加上括号会对计算乘积的成本产生巨大影响。首先考虑两个矩形矩阵相乘的成本。标准算法由过程 R ECTANGULAR-MATRIX-MULTI PLY 给出，该过程推广了第 81 页的方阵乘法过程 MATRIX-MULTIPLY。R ECTANGULAR-MATRIX-MULTIPLY 过程计算三个矩阵 A D .a $_{ij}$/、B D .b $_{ij}$/ 和 C D .c $_{ij}$/ 的 C D C C AB，其中 A 是 pq，B 是 qr，C 是 pr。

矩形矩阵乘法 .A; B; C; p; q; r /

```
1  for i = 1 to p
2      for j = 1 to r
3          for k = 1 to q
4              c_ij = c_ij + a_ik · b_kj
```

RECTANGULAR-MATRIX-MULTIPLY 的运行时间主要取决于第 4 行中的标量乘法次数，即 pqr 。因此，我们将矩阵相乘的成本视为标量乘法的次数。（即使我们考虑将 C D 0 初始化为仅执行 C D A B，标量乘法的次数仍然占主导地位。）

为了说明矩阵乘积的不同括号引起的不同成本，考虑三个矩阵的链 hA$_1$; A$_2$; A$_3$i 的问题。假设矩阵的维数分别为 10 100、100 5 和 5 50。按照括号. .A$_1$ A$_2$ /A$_3$ /进行乘法运算，将执行 101005D5000次标量乘法来计算105矩阵乘积A$_1$ A$_2$，再加上另外10550D2500次标量乘法来将此矩阵乘以A$_3$，总共执行7500次标量乘法。根据替代括号进行乘法 .A $_1$ .A $_2$ A $_3$ // 执行 100550D25,000 次标量乘法来计算 10050 矩阵乘积 A$_2$ A$_3$，再加上另外 1010050D50,000 次标量乘法来将 A$_1$ 乘以此矩阵，得到

总共 75,000 次标量乘法。因此，根据第一个括号计算乘积的速度要快 10 倍。

我们将 ***matrix-chain multiplication problem*** 表述如下：给定一个由 n 个矩阵组成的链 hA 1; A 2; : : : ; A n i，其中对于 i D 1; 2; : : : ; n，矩阵 A i 的维度为 p i−1 p i，将乘积 A 1 A 2 A n 完全括起来，使标量乘法的次数最少。输入是维度为 hp 0; p 1; p 2; : : : ; p n i 的序列。

矩阵链乘法问题并不涉及实际的矩阵乘法。其目标只是确定一个成本最低的矩阵乘法顺序。通常，确定这个最佳顺序所花费的时间，比实际执行矩阵乘法时节省的时间要多得多（例如，只执行 7500 次标量乘法，而不是 75,000 次）。

计算括号的数量

在用动态规划解决矩阵链乘法问题之前，让我们先说服自己，穷举检查所有可能的括号并不是一个有效的算法。用 P .n/ 表示 n 个矩阵序列的备选括号数。当 n D 1 时，该序列仅由一个矩阵组成，因此只有一种方法可以完全括号化矩阵积。当 n ≈ 2 时，完全括号化的矩阵积是两个完全括号化的矩阵子积的乘积，并且对于任何 k D 1; 2; : : : ; n 1，两个子积之间的分割可能发生在第 k 个和第 .k C 1/st 矩阵之间。因此，我们得到递归

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 . \end{cases} \tag{14.6}$$

第 329 页的习题 12-4 要求你证明类似递归式的解是序列 ***Catalan numbers***，其增长速度为 .4 n /n 3/2 /。一个更简单的练习（参见练习 14.2-3）是证明递归式 (14.6) 的解是 .2 n /。因此，解的数量是 n 的指数，而穷举搜索的强力方法在确定如何最佳地对矩阵链进行括号时是一种糟糕的策略。

应用动态规划

让我们使用动态规划方法来确定如何最佳地对矩阵链进行括号化，遵循本章开头所述的四步顺序：

1. 描述最优解的结构。2. 递归地定义最优解的值。3. 计算最优解的值。4. 根据计算信息构建最优解。

我们将按顺序介绍这些步骤，演示如何将每个步骤应用于解决问题。

步骤 1：最佳括号的结构

在动态规划方法的第一步中，您将找到最优子结构，然后使用它从子问题的最优解构造问题的最优解。为了对矩阵链乘法问题执行此步骤，首先引入一些符号会很方便。设 $A_{i:j}$（其中 $i \# j$）表示通过评估乘积 $A_i A_{i+1} \cdots A_j$ 得到的矩阵。如果问题是非平凡的，即 $i < j$，则为了将乘积 $A_i A_{i+1} \cdots A_j$ 括起来，乘积必须在 A_k 和 A_{k+1} 之间拆分，其中 k 位于 $i \# k < j$ 范围内。也就是说，对于某个 k 值，首先计算矩阵 $A_{i:k}$ 和 $A_{k+1:j}$，然后将它们相乘以得出最终乘积 $A_{i:j}$。以这种方式加括号的成本是计算矩阵 $A_{i:k}$ 的成本，加上计算 $A_{k+1:j}$ 的成本，再加上将它们相乘的成本。

这个问题的最优子结构如下。假设为了最优地括住 $A_i A_{i+1} \cdots A_j$，你将乘积拆分在 A_k 和 A_{k+1} 之间。那么在 $A_i A_{i+1} \cdots A_j$ 的最优括号内括住 “preû x” 子链 $A_i A_{i+1} \cdots A_k$ 的方式必定是 $A_i A_{i+1} \cdots A_k$ 的最优括号。为什么？如果存在一种成本较低的方法来括住 $A_i A_{i+1} \cdots A_k$，那么您可以用该括号替换最佳括号 $A_i A_{i+1} \cdots A_j$ 中的内容，以产生另一种方法来括住 $A_i A_{i+1} \cdots A_j$，其成本低于最佳方法：矛盾。对于如何在 $A_i A_{i+1} \cdots A_j$ 的最佳括号中括住子链 $A_{k+1} A_{k+2} \cdots A_j$，也有类似的观察：它必须是 $A_{k+1} A_{k+2} \cdots A_j$ 的最佳括号。

现在让我们使用最优子结构来展示如何从子问题的最优解构建问题的最优解。矩阵链乘法问题的任何非平凡实例的解都需要拆分乘积，并且任何最优解都包含子问题实例的最优解。因此，要构建矩阵链乘法问题实例的最优解，请将问题拆分为两个子问题（最优地括号化 $A_i A_{i+1} \cdots A_k$ 和 $A_{k+1} A_{k+2} \cdots A_j$），找到两个子问题实例的最优解，然后组合这些最优子问题解。为确保您已经检查了最优拆分，您必须考虑所有可能的拆分。

第 2 步：递归解决方案

下一步是根据子问题的最优解递归地定义最优解的成本。对于矩阵链乘法问题，子问题是确定对 1 = i = j = n 用括号括起来 A $_i$A $_{i+1}$ A $_j$ 的最小成本。给定输入维度 hp $_0$; p $_1$; p $_2$; : : :; p $_n$i，索引对 i; j 指定一个子问题。让 mŒi; j 为计算矩阵 A $_{i:j}$ 所需的最少标量乘法次数。对于完整问题，计算 A $_{1:n}$ 的最低成本方法是 mŒ1; n 。

我们可以按如下方式递归定义 mŒi;j 。如果 i D j，问题很简单：链仅由一个矩阵 A $_{i:i}$ D A $_i$ 组成，因此计算乘积时不需要进行标量乘法。因此，当 i D 1; 2; : : : ; n 时，mŒi;i = D 0。要计算 i < j 时的 mŒi;j，我们利用步骤 1 中的最优解结构。假设最优括号将乘积 A $_i$A $_{i+1}$ A $_j$ 拆分在 A $_k$ 和 A $_{k+1}$ 之间，其中 i = k < j 。那么，mŒi;j 等于最小成本 mŒi; k 用于计算子积 A $_{i:k}$，加上最小成本 mŒk C 1；j 用于计算子积 A $_{k+1:j}$，加上将这两个矩阵相乘的成本。由于每个矩阵 A $_i$ 都是 p $_{i-1}$ p $_i$，因此计算矩阵乘积 A $_{i:k}$ A $_{k+1:j}$ 需要 p $_{i-1}$p $_k$p $_j$ 次标量乘法。因此，我们得到

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j .$$

这个递归方程假设你知道 k 的值。但你还不知道，至少现在还不知道。你必须尝试 k 的所有可能值。有多少个？只有 j i，即 k D i; i C 1; : : : ; j 1。由于最佳括号必须使用这些 k 值之一，因此你只需检查所有值即可找到最佳值。因此，对乘积 A $_i$A $_{i+1}$ A $_j$ 进行括号化的成本最小的递归定义变为

$$m[i, j] = \begin{cases} 0 & \text{if } i = j , \\ \min\{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j : i \le k < j\} & \text{if } i < j . \end{cases} \tag{14.7}$$

mŒi; j 值给出了子问题最优解的成本，但它们并未提供构建最优解所需的全部信息。为了帮助您实现此目的，我们将 sŒi; j 定义为 k 值，在该值上，您将乘积 A $_i$A $_{i+1}$ A $_j$ 拆分为最优括号。也就是说，sŒi; j 等于 k 值，使得 mŒi; j D mŒi; k C mŒk C 1; j C p $_{i-1}$p $_k$p $_j$。

步骤 3：计算最优成本

此时，你可以编写一个基于递归的递归算法（14.7）来计算 A $_1$A $_2$乘以 A $_n$ 的最小成本 mŒ1; n 。但正如我们所见

对于杆切割问题，我们将在 14.3 节中看到，这种递归算法花费的时间是指数级的。这并不比使用蛮力法检查每种括号乘积方法更好。

幸运的是，没有那么多不同的子问题：对于满足 1 i j n 的每个 i 和 j 选择，只有一个子问题，或者总共 ã $_n$ ä C n D ，.n ²/。[4] 递归算法可能会在其递归树的不同分支中多次遇到每个子问题。重叠子问题的这种特性是动态规划适用的第二个标志（第一个标志是最佳子结构）。

我们并不递归地计算递归式（14.7）的解，而是使用表格自下而上的方法来计算最优成本，就像过程 MATRIX-CHAIN-ORDER 中那样。（相应的使用记忆化的自上而下的方法出现在 14.3 节中。）输入是一个矩阵维数的序列 p D hp $_0$; p $_1$; : : : ; p $_n$ i 以及 n，这样对于 i D 1; 2; : : : ; n，矩阵 A $_i$ 的维度为 p $_{i-1}$ p $_i$。该过程使用辅助表 mŒ1 W n; 1 W n 来存储 mŒi; j 成本，使用另一个辅助表 sŒ1 W n 1; 2 W n 记录哪个指标 k 在计算 mŒi; j 时实现了最优成本。该表将有助于构建最佳解决方案。

矩阵链顺序 .p; n/

```
1  let m[1:n, 1:n] and s[1:n − 1, 2:n] be new tables
2  for i = 1 to n                          // chain length 1
3      m[i, i] = 0
4  for l = 2 to n                          // l is the chain length
5      for i = 1 to n − l + 1              // chain begins at A_i
6          j = i + l − 1                   // chain ends at A_j
7          m[i, j] = ∞
8          for k = i to j − 1              // try A_{i:k} A_{k+1:j}
9              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10             if q < m[i, j]
11                 m[i, j] = q             // remember this cost
12                 s[i, j] = k             // remember this index
13 return m and s
```

算法应该按照什么顺序填写表项？为了回答这个问题，让我们看看在计算

[4] The $\binom{n}{2}$ term counts all pairs in which $i < j$. Because i and j may be equal, we need to add in the n term.

成本 mŒi; j 。公式 (14.7) 告诉我们，要计算矩阵乘积 $A_{i:j}$ 的成本，首先需要为所有的 k D i; i C 1; : : : ; j 1 计算乘积 $A_{i:k}$ 和 $A_{k+1:j}$ 的成本。链 $A_i A_{i+1} \quad A_j$ 由 j i C 1 矩阵组成，链 $A_i A_{i+1} : : : A_k$ 和 $A_{k+1} A_{k+2} : : : A_j$ 分别由 k i C 1 和 j k 矩阵组成。由于 k < j，因此 k C 1 矩阵链由少于 j C 1 矩阵组成。同样，由于 k i，j k 个矩阵链由少于 j i C 1 个矩阵组成。因此，算法应该按从短矩阵链到长矩阵链的顺序填充表 m。也就是说，对于最优地将链 $A_i A_{i+1} \quad A_j$ 括起来的子问题，将子问题的大小视为链的长度 j i C 1 是有意义的。

现在，让我们看看 MATRIX-CHAIN-ORDER 过程如何按链长度增加的顺序填写 mŒi; j 个条目。第 233 行对 i D 1; 2; : : : ; n 初始化 mŒi; i = D 0，因为任何只有一个矩阵的矩阵链不需要标量乘法。在第 4312 行的 for 循环中，循环变量 l 表示要计算其最小成本的矩阵链的长度。此循环的每次迭代都使用递归（14.7）来计算 mŒi; i C l 1 for i D 1; 2; : : : ; n l C 1。在第一次迭代中，l D 2，因此循环计算 mŒi; i C1 for i D 1; 2; : : : ; n1：长度为 l D 2 的链的最小成本。第二次循环，计算 i D 1; 2; : : : ; n 2 的 mŒi; i C 2 ：长度为 l D 3 的链的最小成本。依此类推，最后得到长度为 l D n 的单个矩阵链并计算 mŒ1; n 。当第 7312 行计算 mŒi; j 成本时，该成本仅取决于已经计算出的表条目 mŒi; k 和 mŒk C 1; j 。

图 14.5 说明了由 MATRIX-CHAIN-ORDER 过程对 n 个 D 6 矩阵链填写的 m 表和 s 表。由于 mŒi;j 仅对 i = j 定义，因此仅使用表 m 中位于主对角线或主对角线上方的部分。图中将表旋转以使主对角线水平延伸。矩阵链列在底部。使用此布局，将矩阵子链 $A_i A_{i+1} A_j$ 相乘的最小成本 mŒi;j 出现在从 A_i 向东北延伸并与从 A_j 向西北延伸的线的交点处。横读，表中的每个对角线都包含相同长度的矩阵链的条目。 MATRIX-CHAIN-ORDER 计算每行中从下到上、从左到右的行。它使用 k D i; i C 1; : : : ; j 1 的乘积 $p_{i-1} p_k p_j$ 以及 mŒi; j 西南和东南方向的所有条目来计算每个条目 mŒi; j 。

对 MATRIX-CHAIN-ORDER 的嵌套循环结构进行简单检查，可知该算法的运行时间为 $O.n^3/$。循环嵌套深度为三层，每个循环索引（l、i 和 k）最多取 n 1 个值。练习 14.2-5 要求您证明该算法的运行时间实际上也是 $.n^3/$。该算法需要 $.n^2/$ 空间来存储 m 和 s 表。因此，MATRIX-CHAIN-ORDER 比枚举所有可能的括号并检查每一个的指数时间方法高效得多。

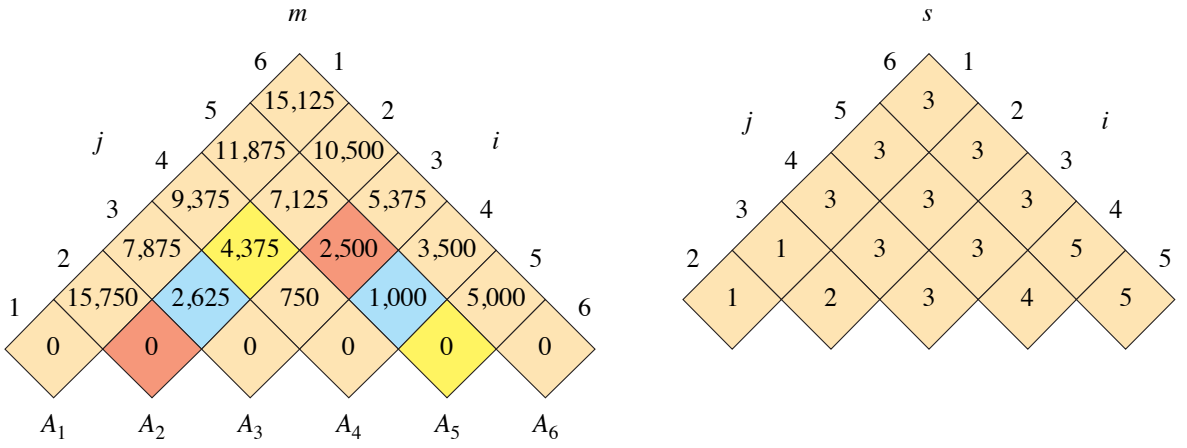


图 14.5 MATRIX-CHAIN-ORDER 针对 n D 6 和以下矩阵维数计算的 m 和 s 表：

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

表格被旋转，使得主对角线水平延伸。m 表仅使用主对角线和上三角，s 表仅使用上三角。将 6 个矩阵相乘所需的标量乘法的最小次数为 mŒ1；6 D 15,125。在非棕褐色的条目中，计算时将具有相同颜色的对在第 9 行一起取

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 &= 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases}$$
$$= 7125\,.$$

步骤 4：构建最优解决方案

尽管矩阵链序 (MATRIX-CHAIN-ORDER) 确定了计算矩阵链积所需的最佳标量乘法次数，但它并未直接显示如何乘以矩阵。表 sŒ1 W n 1; 2 W n 提供了这样做所需的信息。每个条目 sŒi; j 记录了一个 k 值，使得 A $_i$ A $_{i+1}$ A $_j$ 的最佳括号将乘积拆分在 A $_k$ 和 A $_{k+1}$ 之间。计算 A $_{1:n}$ 时，最佳的最后一个矩阵乘法是 A $_{1:s[1,n]}$ A $_{s[1,n]+1:n}$。该表还包含确定先前矩阵乘法所需的信息，使用递归：sŒ1; sŒ1; n 确定计算 A $_{1:s[1,n]}$ 和 sŒsŒ1 时的最后一个矩阵乘法； n C 1; n 确定计算 A $_{s[1,n]+1:n}$ 时的最后一个矩阵乘法。对页上的递归程序 PRINT-OPTIMAL-PARENS 打印矩阵链乘积 A $_i$ A $_{i+1}$ A $_j$ 的最优括号，给定由 MATRIX-CHAIN-ORDER 计算出的 s 表和 in-

对 i 和 j 进行切分。初始调用 PRINT-OPTIMAL-PARENS .s; 1; n/ 打印全矩阵链积 $A_1 A_2 \cdots A_n$ 的最优括号。在图 14.5 的例子中，调用 PRINT-OPTIMAL-PARENS .s; 1; 6/ 打印最优括号 ..A_1.$A_2 A_3$//..$A_4 A_5$/A_6//。

```
打印-最佳-PARENS .s; i; j /
1 如果 i ==j 2 打印 “A” i 3 否则 打印 “(” 4 PRINT
-OPTIMAL-PARENS .s; i; sŒi; j     / 5 PRINT-OP
TIMAL-PARENS .s; sŒi; j      C 1; j / 6 打印 “)”
```

练习

14.2-1
找到矩阵链积的最优括号，其维度序列为 h5；10；3；12；5；50；6i。

14.2-2
给出一个递归算法 MATRIX-CHAIN-MULTIPLY .A; s; i; j / ，该算法实际上执行最优矩阵链乘法，给定矩阵序列 hA_1; A_2; : : :; A_ni，由 MATRIX-CHAIN-ORDER 计算的 s 表，以及索引 i 和 j 。（初始调用是 MATRIX-CHAIN-MULTIPLY .A; s; 1; n / 。）假设调用 RECTANGULAR-MATRIX-MULTIPLY .A; B / 返回矩阵 A 和 B 的乘积。

14.2-3
利用代换法证明递归式 (14.6) 的解为 $.2^n$/。

14.2-4
描述矩阵链乘法的子问题图，输入链长度为 n。它有多少个顶点？它有多少条边，它们是什么边？

14.2-5
令 R.i; j / 为在调用 MATRIX-CHAIN-ORDER 计算其他表项时表项 mŒi; j 被引用的次数。证明整个表的总引用次数为

$$\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) = \frac{n^3 - n}{3} .$$

(*Hint:* 您可能会发现方程（第 1141 页的 A.4) 很有用。）

14.2-6
证明n元素表达式的全括号恰好有n1对括号。

14.3 动态规划的要素

尽管您刚刚看到了动态规划方法的两个完整示例，但您可能仍然想知道该方法何时适用。从工程角度来看，您应该何时寻找问题的动态规划解决方案？在本节中，我们将研究优化问题必须具备的两个关键要素，以便动态规划能够应用：最佳子结构和重叠子问题。我们还将重新审视并更全面地讨论记忆化如何帮助您在自上而下的递归方法中利用重叠子问题属性。

最优子结构

使用动态规划解决优化问题的第一步是描述最优解的结构。回想一下，如果问题的最优解包含子问题的最优解，则问题呈现 ***optimal substructure***。当问题呈现最优子结构时，这是一个很好的线索，表明动态规划可能适用。（然而，正如第 15 章所讨论的，这也可能意味着贪婪策略适用。）动态规划根据子问题的最优解构建问题的最优解。因此，您必须小心确保您考虑的子问题范围包括最优解中使用的子问题。

最优子结构是解决本章中前两个问题的关键。在 14.1 节中，我们观察到切割长度为 n 的杆的最佳方法（如果 Serling Enterprises 进行任何切割）涉及以最优方式切割第一次切割产生的两个部分。在 14.2 节中，我们注意到矩阵链乘积 $A_i A_{i+1} \cdots A_j$ 的最优括号化（将乘积拆分为 A_k 和 A_{k+1}）中包含了括号化 $A_i A_{i+1} \cdots A_k$ 和 $A_{k+1} A_{k+2} \cdots A_j$ 问题的最佳解决方案。

你会发现自己在发现最佳子结构时遵循了一个常见的模式：

1. 你表明，问题的解决方法包括做出选择，例如选择杆的初始切割点或选择拆分矩阵链的索引。做出此选择后，将留下一个或多个子问题需要解决。

2. 你假设对于给定的问题，你有一个选择，可以得到最佳解决方案。你还不必关心如何确定这个选择。你只是假设它已经给了你。

3. 根据这个选择，你可以确定会出现哪些子问题，以及如何最好地描述由此产生的子问题空间。
4. 通过使用“剪切粘贴”技术，您可以证明在问题最优解中使用的子问题解本身必须是最优的。通过假设每个子问题解都不是最优的，然后得出矛盾，您可以做到这一点。具体来说，通过“剪切出”每个子问题的非最优解并“粘贴”最优解，您可以证明您可以得到原始问题的更好解，从而与您已经有最优解的假设相矛盾。如果最优解导致多个子问题，它们通常非常相似，因此您可以修改剪切粘贴论证，让其中一个子问题轻松应用于其他子问题。

要表征子问题空间，一个好的经验法则是尽量保持空间尽可能简单，然后根据需要扩展它。例如，杆切割问题的子问题空间包含针对每个尺寸 i 最佳切割长度为 i 的杆的问题。这个子问题空间效果很好，没有必要尝试更通用的子问题空间。

相反，假设你试图将矩阵链乘法的子问题空间限制为形式为 $A_1 A_2 \cdots A_j$ 的矩阵乘积。与前面一样，最佳括号必须将此乘积拆分在 A_k 和 A_{k+1} 之间，其中 $1 = k < j$。除非你能保证 k 始终等于 j 1，否则你会发现有形式为 $A_1 A_2 \cdots A_k$ 和 $A_{k+1} A_{k+2} \cdots A_j$ 的子问题。此外，后一个子问题不具有形式 $A_1 A_2 \cdots A_j$。要用动态规划解决这个问题，需要允许子问题在“两端”变化。也就是说，在对乘积 $A_i A_{i+1} \cdots A_j$ 加上括号的子问题中，i 和 j 都需要变化。

最佳子结构在不同问题领域中存在两种变化：

1. 原始问题的最优解使用了多少个子问题，以及 2. 在确定最优解中使用哪些子问题时有多少种选择。

在棒切割问题中，切割一根大小为 n 的棒的最优解仅使用一个子问题（大小为 n i），但我们必须考虑 i 的 n 个选择才能确定哪一个能产生最优解。子链 $A_i A_{i+1} \cdots A_j$ 的矩阵链乘法就是一个有两个子问题和 j i 个选择的例子。对于给定矩阵 A_k，其中乘积分裂，出现两个子问题4，括号内为 $A_i A_{i+1} \cdots A_k$ 和括号内为 $A_{k+1} A_{k+2} \cdots A_j$4，我们必须最优地求解其中的 *both* 个。一旦确定了子问题的最优解，我们就从 j i 个候选解中为指标 k 选择。

通俗地说，动态规划算法的运行时间取决于两个因素的乘积：子问题总数和每个子问题要考虑的选择数。在杆切割问题中，我们总共有 ，.n/ 个子问题，每个子问题最多有 n 个选择，因此运行时间为 $O.n^2/$。矩阵链乘法总共有 ，$.n^2/$ 个子问题，每个子问题最多有 n 1 个选择，因此运行时间为 $O.n^3/$（实际上，根据练习 14.2-5，运行时间为 ，$.n^3/$）。

通常，子问题图提供了执行相同分析的另一种方法。每个顶点对应一个子问题，子问题的选择是与该子问题相关的边。回想一下，在杆切割中，子问题图有 n 个顶点，每个顶点最多有 n 条边，运行时间为 $O.n^2/$。对于矩阵链乘法，如果你要绘制子问题图，它将有 $.n^2/$ 个顶点，每个顶点的度最多为 n 1，总共有 $O.n^3/$ 个顶点和边。

动态规划通常以自下而上的方式使用最优子结构。也就是说，首先找到子问题的最优解，在解决了子问题之后，再找到问题的最优解。寻找问题的最优解需要在子问题中选择使用哪个子问题来解决该问题。问题解决的成本通常是子问题的成本加上直接归因于选择本身的成本。例如，在棒切割中，我们首先解决确定切割长度为 i 的棒的最佳方法的子问题，其中 i D 0; 1; : : : ; n 1，然后我们使用公式 (14.2) 确定这些子问题中的哪一个能为长度为 n 的棒提供最优解。归因于选择本身的成本是公式 (14.2) 中的项 p_i。在矩阵链乘法中，我们确定了 $A_i A_{i+1} \cdots A_j$ 子链的最佳括号，然后选择矩阵 A_k 来分割乘积。归因于选择本身的成本是项 $p_{i-1} p_k p_j$。

第 15 章探讨了贪婪算法，它与动态规划有许多相似之处。具体来说，贪婪算法适用的问题具有最优子结构。贪婪算法与动态规划之间的一个主要区别是，贪婪算法不是先找到子问题的最优解，然后做出明智的选择，而是先做出一个贪婪的选择，即当时看起来最好的选择，然后求解由此产生的子问题。

lem，而不必费心解决所有可能相关的较小子问题。令人惊讶的是，在某些情况下，这种策略是有效的！

Subtleties

您应该小心，不要假设最佳子结构适用，而实际情况并非如此。考虑以下两个问题，其输入包括有向图 G D .V; E/ 和顶点 u; v 2 V 。

无权最短路径：[5] 找到一条从 u 到 v 的由最少边组成的路径。这样的路径一定很简单，因为从路径中移除一个循环会产生一条边较少的路径。

无权最长简单路径：找到一条从 u 到 v 的包含最多边的简单路径。（如果不要求路径必须简单，那么这个问题就不难解决，因为反复遍历一个循环会创建包含任意多边的路径。）

无权最短路径问题表现出最优子结构。具体如下。假设 u ¤ v，因此问题并不简单。那么，从 u 到 v 的任何路径 p 都必须包含一个中间顶点，比如 w 。（注意，w 可以是 u 或 v。）然后，我们可以将路径 u $\overset{p}{\rightsquigarrow}$ v 分解为子路径 u $\overset{p_1}{\rightsquigarrow}$ w $\overset{p_2}{\rightsquigarrow}$ v。p 中的边数等于 p_1 中的边数加上 p_2 中的边数。我们声称，如果 p 是从 u 到 v 的最优（即最短）路径，那么 p_1 一定是从 u 到 w 的最短路径。为什么？如前所述，使用 “ 剪切粘贴” 参数：如果有另一条从 u 到 w 的路径，比如 p'_1，其边数较少

比 p_1 少，那么我们可以删掉 p_1 并粘贴 p'_1 以生成一条边数少于 p 的路径 u $\overset{p'_1}{\rightsquigarrow}$ w $\overset{p_2}{\rightsquigarrow}$ v，从而与 p 的最优性相矛盾。同样，p_2 必须是从 w 到 v 的最短路径。因此，要找到从 u 到 v 的最短路径，请考虑所有中间顶点 w，找到从 u 到 w 的最短路径和从 w 到 v 的最短路径，并选择一个中间顶点 w 来产生整体最短路径。第 23.2 节使用这种最佳子结构观察的变体来找到加权有向图上每对顶点之间的最短路径。

您可能倾向于认为，寻找无权最长简单路径的问题也表现出最优子结构。毕竟，如果我们将最长简单路径 u $\overset{p}{\rightsquigarrow}$ v 分解为子路径 u $\overset{p_1}{\rightsquigarrow}$ w $\overset{p_2}{\rightsquigarrow}$ v，那么 p_1 难道不是从 u 到 w 的最长简单路径吗？p_2 难道不是从 w 到 v 的最长简单路径吗？答案是否定的！图 14.6 提供了一个例子。考虑

---

[5] We use the term “unweighted” to distinguish this problem from that of finding shortest paths with weighted edges, which we shall see in Chapters 22 and 23. You can use the breadth-first search technique of Chapter 20 to solve the unweighted problem.

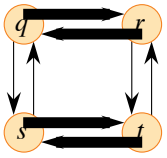


图 14.6 有向图表明在无权有向图中寻找最长简单路径的问题没有最优子结构。路径 q ! r ! t 是从 q 到 t 的最长简单路径，但子路径 q ! r 不是从 q 到 r 的最长简单路径，子路径 r ! t 也不是从 r 到 t 的最长简单路径。

路径 q !r !t ，是从 q 到 t 的最长简单路径。q !r 是从 q 到 r 的最长简单路径吗？不是，因为路径 q !s !t !r 是一条更长的简单路径。r !t 是从 r 到 t 的最长简单路径吗？同样不是，因为路径 r !q !s !t 是一条更长的简单路径。

此示例表明，对于最长简单路径，问题不仅缺乏最优子结构，而且您不一定能从子问题的解中组装出问题的“合法”解。如果将最长简单路径 q !s !t !r 和 r !q !s !t 结合起来，您会得到路径 q !s !t !r !q !s !t ，这并不简单。事实上，寻找无加权最长简单路径的问题似乎不具有任何类型的最优子结构。尚未发现针对此问题的有效动态规划算法。事实上，这个问题是 NP 完全的，4 我们将在第 34 章中看到，4这意味着我们不太可能在多项式时间内找到解决它的方法。

为什么最长简单路径的子结构与最短路径的子结构如此不同？尽管最长路径和最短路径问题的解都使用两个子问题，但寻找最长简单路径的子问题不是 ***inde- pendent***，而最短路径的子问题则是。子问题独立是什么意思呢？我们的意思是，一个子问题的解不会影响同一问题的另一个子问题的解。对于图 14.6 中的例子，我们有寻找从 q 到 t 的最长简单路径的问题，它包含两个子问题：寻找从 q 到 r 和从 r 到 t 的最长简单路径。对于第一个子问题，我们选择了路径 q!s!t!r，它使用了顶点 s 和 t。这些顶点不能出现在第二个子问题的解中，因为两个子问题的解的组合产生的路径并不简单。如果顶点 t 不能出现在第二个问题的解中，那么就没有办法解决它，因为 t 必须位于形成解的路径上，并且它不是子问题解“拼接”在一起的顶点（该顶点是 r ）。因为顶点 s 和 t 出现在一个子问题的解中，所以它们不能出现在另一个子问题的解中。但是，它们中的一个必须出现在另一个子问题的解中，而最优解需要两者。

因此，我们说这些子问题不是独立的。从另一个角度看，在解决一个子问题时使用资源（这些资源是顶点）会导致它们无法用于另一个子问题。

那么，为什么子问题在寻找最短路径时是独立的呢？答案是，子问题本质上不共享资源。我们声称，如果顶点 w 位于从 u 到 v 的最短路径 p 上，那么我们可以将 *any* 最短路径 u ^{p_1} w 和 *any* 最短路径 w ^{p_2} v 拼接在一起，以生成从 u 到 v 的最短路径。我们确信，除了 w 之外，没有顶点可以同时出现在路径 p_1 和 p_2 上。为什么？假设某个顶点 x ¤ w 同时出现在 p_1 和 p_2 中，这样我们可以将 p_1 分解为 u $^{p_{ux}}$ x w，将 p_2 分解为 w x $^{p_{xv}}$ v。根据该问题的最优子结构，路径 p 具有的边数与 p_1 和 p_2 的总和一样多。假设 p 有 e 条边。现在让我们构造一条从 u 到 v 的路径 p′D u $^{p_{ux}}$ x $^{p_{xv}}$ v。因为我们已经切除了从 x 到 w 和从 w 到 x 的路径，每条路径都至少包含一条边，所以路径 p′ 最多包含 e 2 条边，这与 p 是最短路径的假设相矛盾。因此，我们可以确保最短路径问题的子问题是独立的。

14.1 和 14.2 节中研究的两个问题具有独立的子问题。在矩阵链乘法中，子问题是将子链 $A_i A_{i+1} A_k$ 和 $A_{k+1} A_{k+2} A_j$ 相乘。这些子链是不相交的，所以不可能有任何矩阵同时包含在两个子链中。在棒切割中，为确定切割长度为 n 的棒的最佳方法，我们研究了当 i=0;1;:::;n≥1 时切割长度为 i 的棒的最佳方法。因为长度为 n 问题的最优解只包含其中一个子问题解（切掉第一段之后），所以子问题的独立性不是问题。

重叠子问题

优化问题必须具备的第二个要素是，子问题的空间必须“小”，即问题的递归算法会反复解决相同的子问题，而不是总是生成新的子问题。通常，不同子问题的总数是输入大小的多项式。当递归算法反复重访同一问题时，我们说优化问题具有***overlapping subproblems***。[6] 相反，使用分而治之的优化问题

[6] It may seem strange that dynamic programming relies on subproblems being both independent and overlapping. Although these requirements may sound contradictory, they describe two different notions, rather than two points on the same axis. Two subproblems of the same problem are independent if they do not share resources. Two subproblems are overlapping if they are really the same subproblem that occurs as a subproblem of different problems.

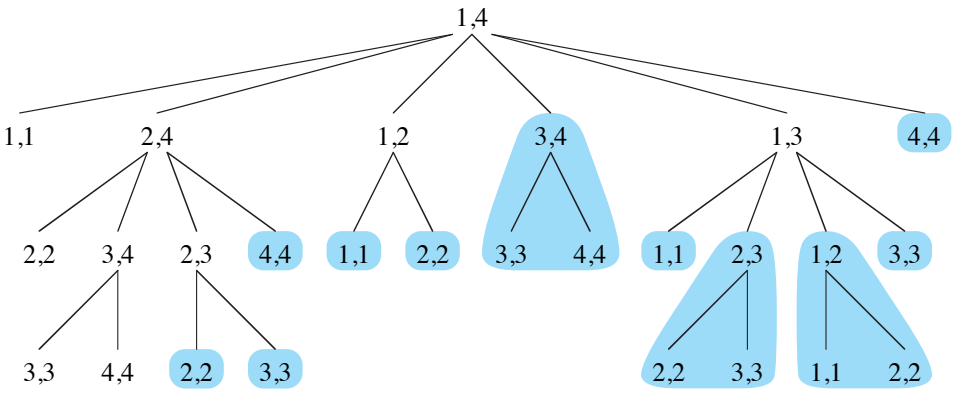


图 14.7 计算 R ECURSIVE-MATRIX-CHAIN.p; 1; 4/ 的递归树。每个节点包含参数 i 和 j 。在蓝色阴影子树中执行的计算由 M EMOIZED-MATRIX-CHAIN 中的单个表查找替换。

征服方法适合在递归的每一步产生全新的问题。动态规划算法通常利用重叠子问题，即解决每个子问题一次，然后将解决方案存储在表中，以便在需要时查找，每次查找只需花费恒定时间。

在第 14.1 节中，我们简要介绍了杆切割的递归解决方案如何以指数方式调用多次来寻找较小子问题的解决方案。动态规划解决方案将运行时间从递归算法的指数时间缩短到二次时间。

为了更详细地说明重叠子问题性质，让我们重新回顾一下矩阵链乘法问题。参考图 14.5，我们可以观察到，在解决较高行中的子问题时，MATRIX-CHAIN-ORDER 会重复查找较低行中子问题的解。例如，它引用条目 mŒ3; 4 四次：在计算 mŒ2; 4 、mŒ1; 4 、mŒ3; 5 和 mŒ3; 6 期间。如果算法每次都重新计算 mŒ3; 4 ，而不是仅仅查找它，则运行时间将显著增加。为了了解如何增加，请考虑上页中确定 mŒi; 的低效递归程序 R ECURSIVE-MATRIX-CHAIN。 j ，计算矩阵链乘积 $A_{i:j}$ D A_i A_{i+1} A_j 所需的最小标量乘法次数。该过程直接基于递归 (14.7)。图 14.7 显示了调用 RECURSIVE-MATRIX-CHAIN.p; 1; 4/ 生成的递归树。每个节点都由参数 i 和 j 的值标记。观察到一些值对出现多次。

事实上，通过这种递归程序计算 mŒ1; n 的时间至少是 n 的指数。为了了解原因，让 T.n/ 表示 RECURSIVE-MATRIX-

递归矩阵链 .p; i; j /

1 如果 i ==j 2 返回 0 3 mŒi; j D 1 4 对于 k D i 到 j 1
5 q D 递归矩阵链 .p; i; k/ C 递归矩阵链 .p; k C 1; j / C p
$_{i-1}$ p $_{k}$ p $_{j}$ 6 如果 q < mŒi; j 7 mŒi; j D q 8 返回 m
Œi; j

CHAIN 计算 n 个矩阵链的最佳括号。由于第 132 行和第 637 行的执行分别需要至少单位时间，第 5 行中的乘法也是如此，因此检查该过程可得出递归

$$T(n) \geq \begin{cases} 1 & \text{if } n = 1 , \\ 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) & \text{if } n > 1 . \end{cases}$$

注意到对于 i D 1; 2; : : : ; n 1，每个项 T .i / 作为 T .k / 出现一次，作为 T .n k / 出现一次，并将求和中的 n 1 个 1 与前面的 1 一起收集，我们可以将递归式重写为

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n . \tag{14.8}$$

让我们用代换法证明 T .n/ D .2 ^{n} /。具体来说，我们将证明对于所有 n 1，T .n/ 2 $^{n-1}$。对于基本情况 n D 1，求和为空，我们得到 T .1/ 2 ^{0}。归纳地，对于 n 2，我们有

$$\begin{aligned} T(n) &\geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \sum_{j=0}^{n-2} 2^{j} + n && \text{(letting } j = i - 1) \\ &= 2(2^{n-1} - 1) + n && \text{(by equation (A.6) on page 1142)} \\ &= 2^{n} - 2 + n \\ &\geq 2^{n-1} , \end{aligned}$$

这就完成了证明。因此，调用 RECURSIVE-MATRIX-CHAIN .p; 1; n/ 所执行的总工作量至少是 n 的指数。

将这种自上而下的递归算法（不使用记忆）与自下而上的动态规划算法进行比较。后者效率更高，因为它利用了重叠子问题的特性。矩阵链乘法只有 n^2 个不同的子问题，而动态规划算法只求解每个子问题一次。另一方面，递归算法必须在每个子问题每次重新出现在递归树中时求解它。每当问题的自然递归解的递归树重复包含相同的子问题，并且不同子问题的总数很少时，动态规划就可以提高效率，有时甚至会显著提高。

重建最优解决方案

实际上，您通常希望将在每个子问题中做出的选择存储在单独的表中，这样就不必从成本表中重建这些信息。

对于矩阵链乘法，当我们需要重建最优解时，表 sŒi; j 可以节省大量工作。假设第 378 页上的 M ATRIX-CHAIN-ORDER 过程不维护 sŒi; j 表，因此它只填充包含最优子问题成本的表 mŒi; j 。在确定在括号 $A_i A_{i+1} \cdots A_j$ 的最优解中使用哪些子问题时，该过程从 j i 个可能性中进行选择，并且 j 不是常数。因此，重建它为给定问题的解决方案选择的子问题将花费 ，.j / D !.1/ 的时间。因为 M ATRIX-CHAIN-ORDER 存储在 sŒi; j 分割乘积 $A_i A_{i+1} \cdots A_j$ 的矩阵索引，第 381 页的 PRINT-OPTIMAL-PARENS 过程可以在 O.1/ 时间内查找每个选择。

记忆化

正如我们在切割杆问题中看到的那样，动态规划还有一种替代方法，它通常能提供自下而上的动态规划方法的效率，同时保持自上而下的策略。这个想法是使用自然但低效的递归算法。与自下而上的方法一样，你维护一个包含子问题解决方案的表格，但用于填写表格的控制结构更像递归算法。

记忆化递归算法在表中为每个子问题的解决方案保留一个条目。每个表条目最初都包含一个特殊值，以表明该条目尚未填写。当递归算法展开时第一次遇到子问题时，将计算其解决方案，然后将其存储在表中。

后续每次遇到这个子问题时，只需查找表中存储的值并返回它。[7]

过程 MEMOIZED-MATRIX-CHAIN 是第 389 页上过程 RECURSIVE-MATRIX-CHAIN 的记忆化版本。请注意，它与第 369 页上用于解决杆切割问题的记忆化自上而下方法的相似之处。

```
记忆矩阵链 .p; n/
1 让 mŒ1 W n; 1 W n    成为新表 2 fo
r i D 1 到 n 3 for j D i 到 n 4 mŒi; j
D 1 5 返回 LOOKUP-CHAIN.m; p; 1;
n/

LOOKUP-CHAIN.m; p; i; j /
1 如果 mŒi; j     < 1 2 返回 mŒi; j     3 如果 i ==j 4 mŒi; j
D 0 5 否则对于 k D i 至 j  1 6 q D LOOKUP-CHAIN.m; p; i; k/ C
LOOKUP-CHAIN.m; p; k C 1; j / C p i−1 p k p j 7 如果 q < mŒi;
j     8 mŒi; j     D q 9 返回 mŒi; j
```

MEMOIZED-MATRIX-CHAIN 过程与第 378 页上的自下而上的 MATRIX-CHAIN-ORDER 过程类似，维护一个表 mŒ1 W n; 1 W n   ，其中包含 mŒi; j   的计算值，即计算矩阵 $A_{i:j}$ 所需的最小标量乘法次数。每个表条目最初都包含值 1，以表明该条目尚未填写。调用 LOOKUP-CHAIN.m; p; i; j / 时，如果第 1 行发现 mŒi; j   < 1，则该过程仅返回先前在第 2 行计算的成本 mŒi; j   。否则，成本按 RECURSIVE-MATRIX-CHAIN 中的方式计算，存储在 mŒi; j   中，然后返回。因此，LOOKUP-CHAIN.m; p; i; j / 始终返回 mŒi; j   的值，但它仅在第一次调用 LOOKUP-CHAIN 时使用这些特定的 i 和 j 值来计算该值。

---

[7] This approach presupposes that you know the set of all possible subproblem parameters and that you have established the relationship between table positions and subproblems. Another, more general, approach is to memoize by using hashing with the subproblem parameters as keys.

图 14.7 说明了 MEMOIZED-MATRIX-CHAIN 与 RECURSIVE-MATRIX-CHAIN 相比如何节省时间。蓝色阴影的子树表示查找而不是重新计算的值。

与自下而上的 MATRIX-CHAIN-ORDER 程序一样，记忆化程序 MEMOIZED-MATRIX-CHAIN 的运行时间为 $O.n^3/$ 次。首先，MEMOIZED-MATRIX-CHAIN 的第 4 行执行了 ，$.n^2/$ 次，这占据了第 5 行对 LOOKUP-CHAIN 的调用之外的运行时间。我们可以将 LOOKUP-CHAIN 的调用分为两类：

1. 调用其中 mŒi; j D 1，以便执行第 339 行，并且
2. 调用其中 mŒi; j < 1，以便 LOOKUP-CHAIN 仅在第 2 行返回。

第一种类型有 $.n^2/$ 次调用，每个表条目一次。第二种类型的所有调用都由第一种类型的调用作为递归调用进行。每当 LOOKUP-CHAIN 的给定调用进行递归调用时，它都会进行 $O.n/$ 次。因此，第二种类型总共有 $O.n^3/$ 次调用。第二种类型的每次调用都需要 $O.1/$ 次时间，而第一种类型的每次调用都需要 $O.n/$ 次时间加上其递归调用所花费的时间。因此，总时间为 $O.n^3/$ 。因此，记忆化将 $.2^n/$ 次算法转变为 $O.n^3/$ 次算法。

我们已经了解了如何在 $O.n^3/$ 时间内通过自上而下的记忆式动态规划算法或自下而上的动态规划算法解决矩阵链乘法问题。自下而上和记忆式方法都利用了重叠子问题的特性。总共只有 ，$.n^2/$ 个不同的子问题，这两种方法都只计算一次每个子问题的解。如果没有记忆式，自然递归算法的运行时间会呈指数级增长，因为已解决的子问题会被重复求解。

在一般实践中，如果所有子问题都必须至少解决一次，那么自下而上的动态规划算法通常比相应的自上而下的记忆算法好上一个常数倍，因为自下而上的算法没有递归开销，维护表的开销也更少。此外，对于某些问题，您可以利用动态规划算法中表访问的常规模式进一步减少时间或空间需求。另一方面，在某些情况下，子问题空间中的某些子问题可能根本不需要解决。在这种情况下，记忆解决方案的优势在于只解决那些肯定需要的子问题。

练习

14.3-1

哪种方法更有效来确定矩阵链乘法问题中的最佳乘法次数：列举所有括号化的方法？

计算乘积并计算每个乘法的次数，还是运行 RECURSIVE-MATRIX-CHAIN ？证明你的答案。

14.3-2
在一个包含 16 个元素的数组上绘制 2.3.1 节中 MERGE-SORT 过程的递归树 。解释为什么记忆化无法加速 MERGE-SORT 等优秀的分治算法。

14.3-3
考虑矩阵链乘法问题的对立变体，其中目标是将矩阵序列括起来，以最大化而不是最小化标量乘法的数量。这个问题是否表现出最佳子结构？

14.3-4
如上所述，在动态规划中，您首先要解决子问题，然后选择其中哪个子问题作为问题的最优解。卡普莱特教授声称，她并不总是需要解决所有子问题才能找到最优解。她建议，她可以通过始终选择矩阵 A_k 来拆分子积 $A_i A_{i+1} A_j$ (，方法是在解决子问题之前选择 k 来最小化数量 $p_{i-1} p_k p_j$)，从而找到矩阵链乘法问题的最优解。找出一个矩阵链乘法问题实例，对于该实例，这种贪婪方法会产生次优解。

14.3-5
假设 14.1 节中的杆切割问题对允许生产的长度为 i 的件数也有限制 l_i，其中 i D 1; 2; : : : ; n。证明 14.1 节中描述的最佳子结构性质不再成立。

14.4 最长公共子序列

生物学应用通常需要比较两个（或更多）不同生物体的 DNA。DNA 链由称为 ***bases*** 的一串分子组成，其中可能的碱基是腺嘌呤、胞嘧啶、鸟嘌呤和胸腺嘧啶。用每个碱基的首字母表示它们，我们可以将 DNA 链表示为 4 元素集 fA；C；G；Tg 上的字符串。（有关字符串的定义，请参阅第 C.1 节。）例如，一个生物体的 DNA 可能是 S_1 D ACCGGTCGAGTGCGCGGAAGCC GGCCGAA ，而另一个生物体的 DNA 可能是 S_2 D GTCGTTCGGAATGCC GTTGCTCTGTAAA 。比较的一个原因是

比较两条 DNA 链的一个主要目的是确定这两条链的 “相似” 程度，以此作为衡量两个生物体关系密切程度的某种量度。我们能够而且确实以许多不同的方式定义相似性。例如，如果一条链是另一条链的子串，我们可以说两条 DNA 链相似。（第 32 章探讨了解决这个问题的算法。）在我们的例子中，S_1 和 S_2 都不是另一条链的子串。或者，如果将一条链变成另一条链所需的变化次数很少，我们可以说两条链相似。（问题 14-5 研究这个概念。）衡量链 S_1 和 S_2 相似性的另一种方法是找到第三条链 S_3，其中 S_3 中的碱基出现在 S_1 和 S_2 中。这些碱基必须以相同的顺序出现，但不一定连续。我们能找到的链 S_3 越长，S_1 和 S_2 就越相似。在我们的例子中，最长的链 S_3 是 GTCGTCGGAAGCCGGCCGAA 。

我们将最后一个相似性概念形式化为最长公共子序列问题。给定序列的子序列就是给定序列中省略 0 个或多个元素。正式地，给定一个序列 $X = \langle x_1, x_2, \ldots, x_m \rangle$，另一个序列 $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 是 X 的 ***subsequence***，如果存在一个严格递增的序列 $\langle i_1, i_2, \ldots, i_k \rangle$，其中 X 的索引为，使得对于所有 $j = 1, 2, \ldots, k$，我们有 $x_{i_j} = z_j$。例如，$Z = \langle B, C, D, B \rangle$ 是 $X = \langle A, B, C, B, D, A, B \rangle$ 的子序列，对应索引序列 $\langle 2, 3, 5, 7 \rangle$。

给定两个序列 X 和 Y ，如果序列 Z 是 X 和 Y 的子序列，则我们说序列 Z 是 X 和 Y 的 ***common sub- sequence***。例如，如果 $X = \langle A, B, C, B, D, A, B \rangle$ 且 $Y = \langle B, D, C, A, B, A \rangle$，则序列 $\langle B, C, A \rangle$ 是 X 和 Y 的公共子序列。但是，序列 $\langle B, C, A \rangle$ 不是 X 和 Y 的 *longest* 公共子序列 (***LCS***)，因为它的长度为 3，而序列 $\langle B, C, B, A \rangle$（也是序列 X 和 Y 的公共子序列）的长度为 4。序列 $\langle B, C, B, A \rangle$ 是 X 和 Y 的 LCS，序列 $\langle B, D, A, B \rangle$ 也是，因为 X 和 Y 没有长度为 5 或更大的公共子序列。

在 ***longest-common-subsequence problem*** 中，输入是两个序列 $X = \langle x_1, x_2, \ldots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \ldots, y_n \rangle$，目标是找到 X 和 Y 的最大长度公共子序列。本节介绍如何使用动态规划有效地解决 LCS 问题。

步骤 1：表征最长公共子序列

你可以用一种蛮力方法解决 LCS 问题：枚举 X 的所有子序列，并检查每个子序列是否也是 Y 的子序列，同时跟踪找到的最长子序列。X 的每个子序列对应于 X 的索引 $\{1, 2, \ldots, m\}$ 的子集。由于 X 有 2^m 个子序列，这种方法需要指数时间，因此对于长序列来说不切实际。

然而，正如下面的定理所示，LCS 问题具有最优子结构性质。如我们所见，子问题的自然类对应于两个输入序列的 “前缀” 对。准确地说，给定一个序列 X D hx $_1$; x $_2$; : : : ; x $_m$ i，对于 i D 0; 1; : : : ; m，我们定义 X 的第 i 个 ***prefix*** 为 X $_i$ D hx $_1$; x $_2$; : : : ; x $_i$ i。例如，如果 X D hA; B; C; B; D; A; B i，则 X $_4$ D hA; B; C; B i 且 X $_0$ 为空序列。

Theorem14.1 (Optimal substructure of an LCS)
令 X D hx $_1$; x $_2$; : : : ; x $_m$ i 和 Y D hy $_1$; y $_2$; : : : ; y $_n$ i 为序列，令 Z D hz $_1$; z $_2$; : : : ; z $_k$ i 为 X 和 Y 的任意 LCS。

1. 如果 x $_m$ D y $_n$，则 z $_k$ D x $_m$ D y $_n$ 且 Z $_{k-1}$ 是 X $_{m-1}$ 的 LCS, d并且 $_1$。
2. 如果 x $_m$ ¤ y $_n$ 且 z $_k$ ¤ x $_m$，则 Z 是 X $_{m-1}$ 和 Y 的 LCS。
3. 如果 x $_m$ ¤ y $_n$ 且 z $_k$ ¤ y $_n$，则 Z 是 X 和 Y $_{n-1}$ 的 LCS。

Proof（1）若 z $_k$ ¤ x $_m$，则我们可以将 x $_m$ D y $_n$ 附加到 Z 以获得长度为 k C 1 的 X 和 Y 的公共子序列，这与 Z 是 X 和 Y 的 *longest* 公共子序列的假设相矛盾。因此，我们必须有 z $_k$ D x $_m$ D y $_n$。现在，前缀 Z $_{k-1}$ 是 X $_{m-1}$ 和 Y $_{n-1}$ 的长度为 k 1/ 的公共子序列。我们希望证明它是一个 LCS。假设为了矛盾的目的，存在X$_{m-1}$和Y$_{n-1}$的公共子序列W，其长度大于k 1。那么，将x$_m$ Dy$_n$附加到W会产生X和Y的公共子序列，其长度大于k，这是矛盾的。

（2）若 z $_k$ ¤ x $_m$，则 Z 是 X $_{m-1}$ 和 Y 的公共子序列。若 X $_{m-1}$ 和 Y 有一个长度大于 k 的公共子序列 W，则 W 也将是 X $_m$ 和 Y 的公共子序列，这与 Z 是 X 和 Y 的 LCS 的假设相矛盾。

（3）证明与（2）对称。 ■

定理 14.1 描述最长公共子序列的方式是，两个序列的 LCS 包含这两个序列前缀的 LCS。因此，LCS 问题具有最优子结构属性。递归解决方案还具有重叠子问题属性，我们稍后会看到。

第 2 步：递归解决方案

定理 14.1 意味着在找到 X D hx $_1$; x $_2$; : : : ; x $_m$ i 和 Y D hy $_1$; y $_2$; : : : ; y $_n$ i 的 LCS 时，应该检查一个或两个子问题。如果 x $_m$ D y $_n$，则需要找到 X $_{m-1}$ 和 Y $_{n-1}$ 的 LCS。将 x $_m$ D y $_n$ 附加到这个 LCS 会得到 X 和 Y 的 LCS。如果 x $_m$ ¤ y $_n$，那么必须解决两个子问题：找到 X $_{m-1}$ 和 Y 的 LCS 以及找到 X 和 Y $_{n-1}$ 的 LCS。

这两个 LCS 中较长的一个就是 X 和 Y 的 LCS。由于这些情况穷尽了所有可能性，因此最佳子问题解之一必须出现在 X 和 Y 的 LCS 中。

LCS 问题具有重叠子问题属性。方法如下。要找到 X 和 Y 的 LCS，您可能需要找到 X 和 Y_{n-1} 的 LCS 以及 X_{m-1} 和 Y 的 LCS。但这些子问题中的每一个都有找到 X_{m-1} 和 Y_{n-1} 的 LCS 的子子问题。许多其他子问题共享子子问题。

与矩阵链乘法问题一样，递归求解 LCS 问题需要建立最优解值的递归。我们定义 cŒi; j 为序列 X_i 和 Y_j 的 LCS 的长度。如果 i D 0 或 j D 0，则其中一个序列的长度为 0，因此 LCS 的长度为 0。LCS 问题的最优子结构给出了递归公式

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max\{c[i,j-1], c[i-1,j]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases} \tag{14.9}$$

在这个递归公式中，问题中的条件限制了要考虑的子问题。当 x_i D y_j 时，你可以而且应该考虑寻找 X_{i-1} 和 Y_{j-1} 的 LCS 这个子问题。否则，你就应该考虑寻找 X_i 和 Y_{j-1} 的 LCS 以及 X_{i-1} 和 Y_j 的 LCS 这两个子问题。在我们之前研究过的用于杆切割和矩阵链乘法的动态规划算法中，我们没有因为问题中的条件而排除任何子问题。寻找 LCS 并不是唯一一种根据问题中的条件排除子问题的动态规划算法。例如，编辑距离问题（参见问题 14-5）就具有此特性。

步骤 3：计算 LCS 的长度

根据公式 (14.9)，你可以编写一个指数时间递归算法来计算两个序列的 LCS 长度。由于 LCS 问题只有 2.mn 个不同的子问题（计算 0 = i = m 和 0 = j = n 的 cŒi; j），因此动态规划可以自下而上地计算解决方案。

下一页的过程 LC S-LENGTH 将两个序列 X D hx 1 ; x 2 ; : : : ; x m i 和 Y D h y 1 ; y 2 ; : : : ; y n i 作为输入，以及它们的长度。它将 cŒi; j 值存储在表 c Œ0 W m; 0 W n 中，并按 ***row- major*** 顺序计算条目。也就是说，过程从左到右填充 c 的第一行，然后是第二行，依此类推。该过程还维护表 bŒ1 W m; 1 W n ，以帮助构建最优解。直观地说，bŒi; j 指向与计算 cŒi; j 时选择的最优子问题解相对应的表条目。该过程返回 b 和 c 表，其中 cŒm; n 包含 X 和 Y 的 LCS 的长度。图 14.8 显示了 LC S-LENGTH 在

序列 X D hA; B; C; B; D; A; B i 和 Y D hB; D; C; A; B; Ai。该过程的运行时间为 ，.mn/，因为每个表条目需要 ，.1/ 的时间来计算。

```
LCS-长度.X; Y; m; n/
 1 让 bŒ1 W m; 1 W n    和 cŒ0 W m; 0 W n    为新表 2 对于 i D
1 到 m 3 cŒi; 0    D 0 4 对于 j D 0 到 n 5 cŒ0; j    D 0 6 对于 i D 1
到 m // 按行主顺序计算表条目 7 对于 j D 1 到 n 8 如果 x i == y j 9
cŒi; j    D cŒi    1; j  1    C 1 10 bŒi; j    D "-" 11 elseif cŒi    1
; j    cŒi; j  1    12 cŒi; j    D cŒi  1; j    13 bŒi; j    D """ 14 el
se cŒi; j    D cŒi; j  1    15 bŒi; j    D "" 16 返回 c 和 b
```

```
PRINT-LCS.b; X; i; j /
 1 如果 i == 0 或 j == 0 2 返回 // LCS 长度为 0 3
 如果 bŒi; j    == "-" 4 PRINT-LCS.b; X; i    1;
j    1/ 5 打印 x i              // 与 y j 相同 6 否则
 如果 bŒi; j    == """ 7 PRINT-LCS.b; X; i    1;
j / 8 否则 PRINT-LCS.b; X; i; j    1/
```

步骤 4：构建 LCS

使用 LC S-LENGTH 返回的 b 表，可以快速构造 X D hx 1 ; x 2 ; : : : ; x m i 和 Y D hy 1 ; y 2 ; : : : ; y n i 的 LCS。从 bŒm; n   开始，按照箭头跟踪整个表格。在条目 bŒi; j   中遇到的每个 "-" 都意味着 x i D y j 是 LCS-LENGTH 找到的 LCS 的一个元素。此方法以相反的顺序给出这个 LCS 的元素。递归过程 PRINT-LCS 以正确的正向顺序打印出 X 和 Y 的 LCS。

i		j	0	1	2	3	4	5	6
			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

图 14.8 LC S-LENGTH 对序列 X D hA; B; C; B; D; A; Bi 和 Y D hB; D; C; A; B; Ai 计算出的 c 和 b 表。第 i 行第 j 列的方块包含 cŒi; j 的值以及表示 bŒi; j 值的相应箭头。表右下角的 c Œ7; 6 中的条目 4 是 X 和 Y 的 LCS hB; C; B; Ai 的长度。对于 i; j > 0，条目 cŒi; j 仅取决于 x_i D y_j 是否以及条目 cŒi 1; j 、cŒi; j 1 和 cŒi 1; j 1 中的值，这些值是在 cŒi; j 之前计算出来的。要重建 LCS 的元素，请从右下角开始沿 bŒi; j 箭头前进，如蓝色阴影序列所示。蓝色阴影序列上的每个 “-” 对应于一个条目（突出显示），其中 x_i D y_j 是 LCS 的成员。

初始调用是 PRINT-LCS.b; X; m; n/。对于图 14.8 中的 b 表，此过程打印 BC BA。该过程花费 O.m C n/ 次，因为它在每次递归调用中至少减少 i 和 j 中的一个。

改进代码

一旦你开发出一种算法，你就会发现你可以改进它所用的时间或空间。一些变化可以简化代码并改进常数因子，但除此之外不会带来性能的渐进改善。其他变化可以带来时间和空间的显著渐进节省。

例如，在 LCS 算法中，你可以完全消除 b 表。每个 cŒi;j 条目仅依赖于其他三个 c 表条目：cŒi 1;j 1 、cŒi 1;j 和 cŒi;j 1 。给定 cŒi;j 的值，你可以在 O.1/ 时间内确定这三个值中的哪一个用于计算 cŒi;j ，而无需检查表 b。因此，你可以使用类似于 PRINT-LC S 的过程在 O.mCn/ 时间内重建 LCS。（练习 14.4-2 要求你给出伪代码。）虽然这种方法节省了 ,. mn/ 空间，但计算的辅助空间要求

LCS 不会渐近减少，因为 c 表无论如何都会占用“ .mn/ ”空间。

但是，您可以减少 LCS-LENGTH 的渐近空间要求，因为它一次只需要表 c 的两行：正在计算的行和前一行。（事实上，正如练习 14.4-4 要求您展示的那样，您只需使用比 c 一行略多一点的空间即可计算 LCS 的长度。）如果您只需要 LCS 的长度，这种改进是有效的。如果您需要重建 LCS 的元素，较小的表无法保留足够的信息来在 O.m C n/ 时间内追溯算法的步骤。

练习

14.4-1
确定 h1；0；0；1；0；1；0；1i 和 h0；1；0；1；1；0；1；1；0i 的 LCS。

14.4-2
给出伪代码，在 O.m C n/ 时间内，从已完成的 c 表和原始序列 X D hx $_1$; x $_2$; : : : ; x $_m$ i 和 Y D hy $_1$; y $_2$; : : : ; y $_n$ i 重建一个 LCS，不使用 b 表。

14.4-3
给出一个在 O.mn/ 时间内运行的 LC S-LENGTH 的记忆版本。

14.4-4
说明如何仅使用 c 表中的 2 个条目加上 O.1/ 个额外空间来计算 LCS 的长度。然后说明如何做同样的事情，但使用 min fm; ng 条目加上 O.1/ 个额外空间。

14.4-5
给出一个 O.n ^{2} / 次算法来找出 n 个数字序列的最长单调递增子序列。

14.4-6
给出一个 O.n lg n/-time 算法来找出 n 个数字序列中最长的单调递增子序列。（*Hint:* 长度为 i 的候选子序列的最后一个元素至少与长度为 i 的候选子序列的最后一个元素一样大 1.通过将候选子序列通过输入序列链接起来来维护它们。）

14.5 最佳二叉搜索树

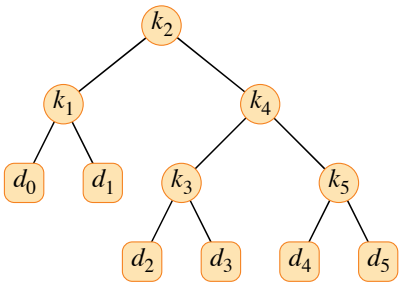
假设您正在设计一个程序，将文本从英语翻译成拉脱维亚语。对于文本中每个英语单词的每次出现，您都需要查找其对应的拉脱维亚语。您可以通过构建一个二叉搜索树来执行这些查找操作，该二叉搜索树以 n 个英语单词为键，以它们的拉脱维亚语对应词为卫星数据。由于您将在树中搜索文本中的每个单词，因此您希望搜索所花费的总时间尽可能短。您可以使用红黑树或任何其他平衡二叉搜索树来确保每次出现的搜索时间为 O.lg n/。但是，单词出现的频率不同，并且经常使用的单词（例如 *the* ）最终可能会出现在远离根的位置，而很少使用的单词（例如 *naumachia*）则出现在根附近。这样的组织会减慢翻译速度，因为在二叉搜索树中搜索键时访问的节点数等于 1 加上包含键的节点的深度。您希望将文本中经常出现的单词放在更靠近根的位置。[8] 此外，文本中的某些单词可能没有拉脱维亚语翻译，[9] 并且这些单词根本不会出现在二叉搜索树中。假设您知道每个单词出现的频率，那么如何组织二叉搜索树以最小化所有搜索中访问的节点数？

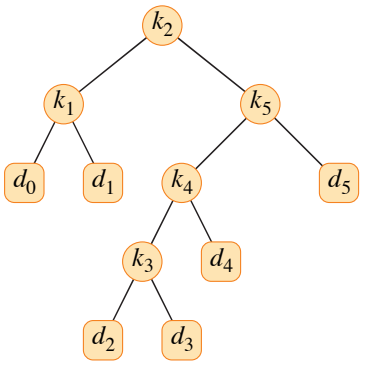
您需要的是 ***optimal binary search tree***。正式地，给定一个序列 KDhk$_1$;k$_2$;:::;k$_n$i，其中包含 n 个不同的键，并且 k$_1$ <k$_2$ <+<k$_n$，构建一个包含它们的二叉搜索树。对于每个键 k$_i$，您都会得到概率 p$_i$，即任何给定的搜索都是针对键 k$_i$。由于某些搜索可能是针对 K 中没有的值，因此您还有 nC1“虚拟”键 d$_0$;d$_1$;d$_2$;:::;d$_n$ 来表示这些值。具体来说，d$_0$ 表示所有小于 k$_1$ 的值，d$_n$ 表示所有大于 k$_n$ 的值，对于 i D 1; 2; : : :; n $\geq$ 1，虚拟键 d$_i$ 表示介于 k$_i$ 和 k$_{i+1}$ 之间的所有值。对于每个虚拟键 d$_i$，有概率 q$_i$ 发生搜索对应于 d$_i$。图 14.9 显示了一组 n D 5 个键的两棵二叉搜索树。每个键 k$_i$ 都是一个内部节点，每个虚拟键 d$_i$ 都是一个叶子节点。由于每次搜索要么成功（找到某个键 k$_i$），要么失败（找到某个虚拟键 d$_i$），因此我们有

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 . \tag{14.10}$$

[8] If the subject of the text is ancient Rome, you might want *naumachia* to appear near the root.

[9] Yes, *naumachia* has a Latvian counterpart: *nomačija*.





node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

(a)

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

(b)

图 14.9 一组 nD5 个密钥的两棵二叉搜索树，其概率如下：

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) 一棵预期搜索成本为 2.80 的二叉搜索树。 (b) 一棵预期搜索成本为 2.75 的二叉搜索树。这棵树是最优的。

了解每个键和每个虚拟键的搜索概率，我们就可以确定给定二叉搜索树 T 中搜索的预期成本。假设搜索的实际成本等于检查的节点数，即在 T 中搜索找到的节点的深度加 1。那么在 T 中搜索的预期成本为

$$\begin{aligned} \mathrm{E}\left[\text{search cost in } T\right] &= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\mathrm{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i \ , \end{aligned} \tag{14.11}$$

其中深度 $_T$ 表示节点在树 T 中的深度。最后一个等式来自公式 (14.10)。图 14.9 显示了如何逐个节点计算预期搜索成本。

对于给定的一组概率，您的目标是构建一个预期搜索成本最小的二叉搜索树。我们将这样的树称为 ***optimal binary search tree***。图 14.9(a) 显示了一棵二叉搜索树，其预期成本为 2:80，针对图标题中给出的概率。图的 (b) 部分显示了一个最佳二叉搜索树，其预期成本为 2:75。此示例表明，最佳二叉搜索树不一定是总高度最小的树。最佳二叉搜索树也不一定总是将具有最大概率的密钥放在根部。这里，密钥 k_5 具有所有密钥中最大的搜索概率，但所示的最佳二叉搜索树的根是 k_2。（任何以 k_5 为根的二叉搜索树的最低预期成本为 2.85。）

与矩阵链乘法一样，对所有可能性进行穷举检查无法产生有效的算法。您可以使用键 $k_1; k_2; : : : ; k_n$ 标记任何 n 节点二叉树的节点以构造二叉搜索树，然后添加虚拟键作为叶子。在第 329 页的问题 12-4 中，我们看到具有 n 个节点的二叉树的数量为 $.4^n/n^{3/2}/$。因此，您需要检查指数数量的二叉搜索树才能执行穷举搜索。我们将看到如何使用动态规划更有效地解决这个问题。

斯蒂p1：最优二分搜索的结构 稀土

为了描述最优二叉搜索树的最优子结构，我们先从对子树的观察开始。考虑二叉搜索树的任意子树。对于某个 1 i j n，它必须包含连续范围 $k_i; : : : ; k_j$ 中的键。此外，包含键 $k_i; : : : ; k_j$ 的子树还必须将虚拟键 $d_{i-1}; : : : ; d_j$ 作为其叶子节点。

现在我们可以陈述最优子结构：如果最优二叉搜索树 T 有一个子树 T′ 包含键 $k_i; : : : ; k_j$，那么这个子树 T′ 对于具有键 $k_i; : : : ; k_j$ 和虚拟键 $d_{i-1}; : : : ; d_j$ 的子问题也必须是最优的。通常的剪切粘贴参数适用。如果有一个子树 T″ 的预期成本低于 T′，那么从 T 中剪切 T′ 并粘贴 T″ 将导致二叉搜索树的预期成本低于 T，从而与 T 的最优性相矛盾。

有了最优子结构，下面是如何从子问题的最优解构造问题的最优解。给定键 $k_i; : : : ; k_j$，其中一个键，比如说 k_r (i r j)，是包含这些键的最优子树的根。根 k_r 的左子树包含键 $k_i; : : : ; k_{r-1}$ (和虚拟键 $d_{i-1}; : : : ; d_{r-1}$)，右子树包含键 $k_{r+1}; : : : ; k_j$ (和虚拟键 $d_r; : : : ; d_j$)。只要检查所有候选根 k_r，

其中 i = r = j，并且确定所有包含 k_i；：：：；k_{r-1} 的最优二叉搜索树以及包含 k_{r+1}；：：：；k_j 的最优二叉搜索树，则保证能找到一棵最优二叉搜索树。

关于 “空” 子树，有一个技术细节值得理解。假设在具有键 $k_i; : : : ; k_j$ 的子树中，选择 k_i 作为根。根据上述论证，k_i 的左子树包含键 $k_i; : : : ; k_{i-1}$：根本没有键。但请记住，子树还包含伪键。我们采用这样的约定：包含键 $k_i; : : : ; k_{i-1}$ 的子树没有实际键，但包含单个伪键 d_{i-1}。对称地，如果选择 k_j 作为根，则 k_j 的右子树包含键 $k_{j+1}; : : : ; k_j$。这个右子树不包含实际的键，但它包含虚拟键 d_j。

第 2 步：递归解决方案

为了递归地定义最优解的值，子问题的范围是找到一个包含键 $k_i; : : : ; k_j$ 的最优二叉搜索树，其中 $i \geq 1$，$j \geq n$，$j \geq i$ 1。（当 $j \geq i$ 时，只有虚拟键 d_{i-1}，没有实际键。）令 eŒi; j 表示搜索包含键 $k_i; : : : ; k_j$ 的最优二叉搜索树的预期成本。您的目标是计算 eŒ1; n，即搜索所有实际键和虚拟键的最优二叉搜索树的预期成本。

简单情况是 j D i $\geq$ 1。此时子问题仅由虚拟密钥 d_{i-1} 组成。预期搜索成本为 eŒi; i $\geq$ 1 D q_{i-1}。

当j≥i时，需要从k_i;:::;k_j中选择一个根k_r，然后构造一棵以键k_i;:::;k_{r-1}为左子树的最优二叉搜索树，以及一棵以键k_{r+1};:::;k_j为右子树的最优二叉搜索树。当子树成为某个节点的子树时，其预期搜索成本会发生什么变化？子树中每个节点的深度增加1。根据公式（14.11），该子树的预期搜索成本增加了子树中所有概率的总和。对于具有键k_i;:::;k_j的子树，将此概率和表示为

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \ . \tag{14.12}$$

因此，如果 k_r 是包含键 $k_i; : : : ; k_j$ 的最佳子树的根，则我们有

$$e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j)) \ .$$

注意到

$$w(i, j) = w(i, r - 1) + p_r + w(r + 1, j) \ ,$$

我们将 eŒi; j 重写为

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) . \tag{14.13}$$

递归方程 (14.13) 假设您知道要使用哪个节点 k $_r$ 作为根。当然，您会选择预期搜索成本最低的根，从而得出最终的递归公式：

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i-1 , \\ \min\{e[i, r-1] + e[r+1, j] + w(i, j) : i \le r \le j\} & \text{if } i \le j . \end{cases} \tag{14.14}$$

eŒi; j 值给出了最优二叉搜索树的预期搜索成本。为了帮助跟踪最优二叉搜索树的结构，定义 *root* Œi; j （对于 1 = i = j = n）为索引 r，其中 k $_r$ 是包含键 k $_i$; : : : ; k $_j$ 的最优二叉搜索树的根。虽然我们将看到如何计算 *root* Œi; j 的值，但从这些值构建最优二叉搜索树留作练习 14.5-1。

步骤 3：计算最佳二叉搜索树的预期搜索成本

至此，您可能已经注意到我们对最优二叉搜索树和矩阵链乘法的特征描述之间存在一些相似之处。对于这两个问题域，子问题都由连续的索引子范围组成。直接递归实现公式 (14.14) 的效率与直接递归矩阵链乘法算法一样低效。相反，您可以将 eŒi; j 值存储在表 eŒ1 W n C 1; 0 W n 中。第一个索引需要运行到 n C 1 而不是 n，因为为了获得仅包含虚拟键 d_n 的子树，您需要计算并存储 eŒn C 1; n 。第二个索引需要从 0 开始，因为为了获得仅包含虚拟键 d_0 的子树，您需要计算并存储 eŒ1; 0 。只有条目 eŒi; j ，其中 j ¹ i 1 已填写。表 *root* Œi; j 记录包含键 k $_i$; : : :; k $_j$ 的子树的根，并且仅使用 1 × i × j × n 的条目。

另一个表使动态规划算法稍微快一点。每次计算 eŒi; j 时，不必从头计算 w.i; j 的值（这需要 1.j 1 i / 加法），而是将这些值存储在表 wŒ1 W n C 1; 0 W n 中。对于基本情况，计算 wŒi; i 1 D q $_{i-1}$ ，其中 1 = i = n C 1。对于 j 1 i ，计算

$$w[i, j] = w[i, j-1] + p_j + q_j . \tag{14.15}$$

因此，你可以每次计算 ，.n ^{2} / 个 wŒi; j 值。

下一页的 OPTIMAL-BST 程序将概率 p $_1$; : : : ; p $_n$ 和 q $_0$; : : : ; q $_n$ 以及大小 n 作为输入，并返回表 e 和 *root* 。根据上述描述以及与 MATRIX-CHAIN-ORDER 程序的相似性

在第 14.2 节中，您会发现这个过程的操作相当简单明了。第 234 行的 for 循环初始化 eŒi; i 1 和 wŒi; i 1 的值。然后第 5314 行的 for 循环使用递归 (14.14) 和 (14.15) 对所有 1 = i = j = n 计算 eŒi; j 和 wŒi; j 。在第一次迭代中，当 l D 1 时，循环对 i D 1; 2; : : : ; n 计算 eŒi; i 和 wŒi; i 。第二次迭代中，当 l D 2 时，对 i D 1; 2; 计算 eŒi; i C 1 和 wŒi; i C 1 。 : : : ; n 1，等等。最内层的 for 循环位于第 103-14 行，它尝试每个候选索引 r 来确定将哪个键 k_r 用作包含键 k_i ；: : : ; k_j 的最佳二叉搜索树的根。此 for 循环在找到更好的键作为根时，将索引 r 的当前值保存在 *root* Œi; j 中。

```
OPTIMAL-BST(p, q, n)
 1  let e[1 : n + 1, 0 : n], w[1 : n + 1, 0 : n],
          and root[1 : n, 1 : n] be new tables
 2  for i = 1 to n + 1                  // base cases
 3      e[i, i − 1] = q_{i−1}           // equation (14.14)
 4      w[i, i − 1] = q_{i−1}
 5  for l = 1 to n
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          e[i, j] = ∞
 9          w[i, j] = w[i, j − 1] + p_j + q_j          // equation (14.15)
10          for r = i to j                             // try all possible roots r
11              t = e[i, r − 1] + e[r + 1, j] + w[i, j]  // equation (14.14)
12              if t < e[i, j]                         // new minimum?
13                  e[i, j] = t
14                  root[i, j] = r
15  return e and root
```

图 14.10 显示了由 OPTIMAL-BST 程序根据图 14.9 所示的密钥分布计算出的表 eŒi; j 、wŒi; j 和 *root* Œi; j 。与图 14.5 中的矩阵链乘法示例一样，这些表被旋转以使对角线水平延伸。OPTIMAL-BST 在每一行中从下到上、从左到右计算行。

OPTIMAL-BST 过程需要 .n^3/ 时间，就像 MATRIX-CHAIN-ORDER 一样。它的运行时间为 O.n^3/，因为它的 for 循环嵌套深度为三层，并且每个循环索引最多取 n 个值。OPTIMAL-BST 中的循环索引与 MATRIX-CHAIN-ORDER 中的循环索引没有完全相同的界限，但它们在所有方向上最多都在 1 以内。因此，与 MATRIX-CHAIN-ORDER 一样，OPTIMAL-BST 过程需要 .n^3/ 时间。

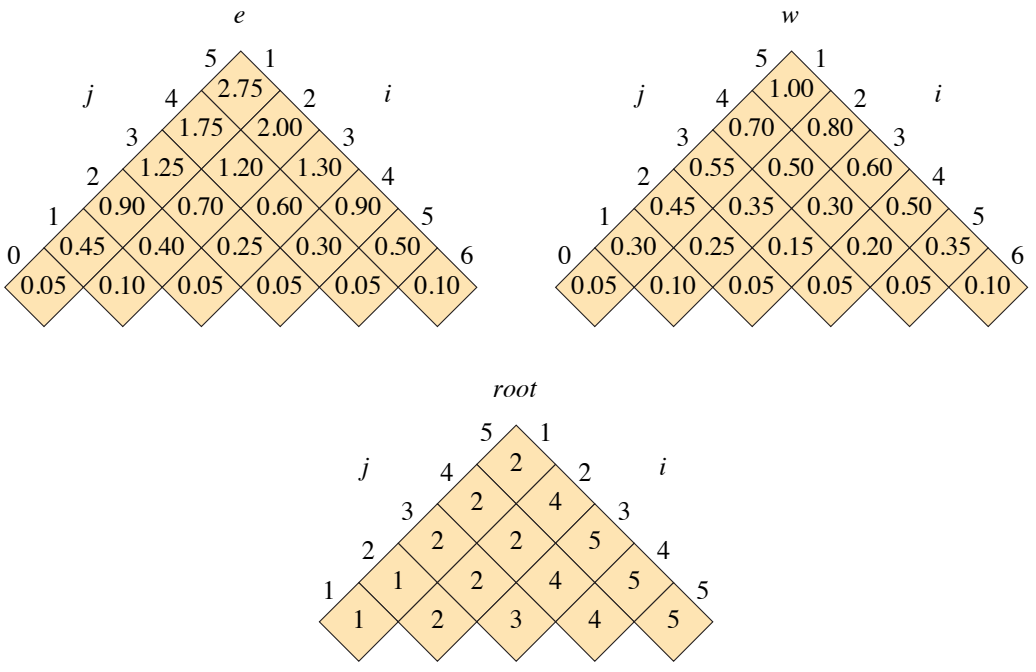


图 14.10 OPTIMAL-BST 根据图 14.9 所示的密钥分布计算表 eŒi; j 、wŒi; j 和 *root*Œi; j 。这些表被旋转，使得对角线水平延伸。

练习

14.5-1

编写 CONSTRUCT-OPTIMAL-BST .*root* ; n/ 程序的伪代码，给定表 *root* Œ1 W n; 1 W n ，输出最优二叉搜索树的结构。对于图 14.10 中的示例，你的程序应打印出结构

k 2 是根 k 1 是 k 2 的左孩子 d 0 是 k 1 的左孩子 d 1 是 k 1 的右孩子 k 5 是 k 2 的右孩子 k 4 是 k 5 的左孩子 k 3 是 k 4 的左孩子 d 2 是 k 3 的左孩子 d 3 是 k 3 的右孩子 d 4 是 k 4 的右孩子 d 5 是 k 5 的右孩子

对应于图 14.9(b) 所示的最佳二叉搜索树。

14.5-2确定一组 n D 7 个密钥的最佳二叉搜索树的成本和结构，概率如下：

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

14.5-3

假设你不去维护表 wŒi; j ，而是直接从 OPTIMAL-BST 第 9 行的公式 (14 .12) 计算 w.i; j / 的值，并在第 11 行中使用这个计算值。这种变化将如何影响 OPTIMAL-BST 的渐近运行时间？

14.5-4

Knuth [264] 证明，对于所有 1 = i < j = n，总存在最优子树的根，使得 *root* Œi; j 1 = *root* Œi; j = *root* Œi C 1; j 。利用这一事实修改 OPTIMAL-B ST 程序，使其运行时间为 ，.n 2 / 时间。

问题

14-1Longest simple path in a directed acyclic graph

给定一个有向无环图 G D .V; E/，其边权重为实值，有两个不同的顶点 s 和 t 。路径的 ***weight*** 是路径中边权重的总和。描述一种动态规划方法，用于找到从 s 到 t 的最长加权简单路径。你的算法的运行时间是多少？

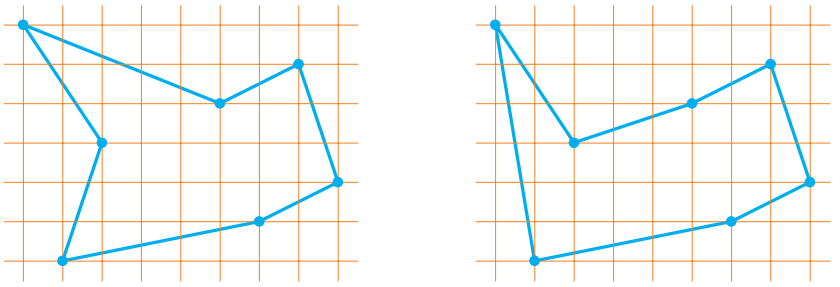
14-2Longest palindrome subsequence

palindrome 是某个字母表上的非空字符串，正读和反读都一样。回文的例子包括所有长度为 1 的字符串、civic、racecar 和 aibohphobia（害怕回文）。

给出一个有效的算法来找出给定输入字符串的最长回文子序列。例如，给定输入字符，你的算法应该返回 carac。你的算法的运行时间是多少？

14-3Bitonic euclidean traveling-salesperson probl em

在 ***euclidean traveling-salesperson problem*** 中，给定平面上的 n 个点的集合，您的目标是找到连接所有 n 个点的最短闭路。



(a) (b)

图 14.11 平面上的七个点，显示在单位网格上。（a）最短闭环路线，长度约为 24:89。此路线不是双调的。（b）同一组点的最短双调路线。其长度约为 25:58。

图 14.11(a) 显示了 7 点问题的解。该一般问题是 NP 难的，因此其解被认为需要超过多项式时间（参见第 34 章）。

J. L. Bentley 建议通过仅考虑 ***bitonic tours*** 来简化问题，即从最左边的点开始，严格向右走到最右边的点，然后严格向左回到起点的路线。图 14.11(b) 显示了相同 7 个点的最短双调路线。在这种情况下，多项式时间算法是可能的。

描述一种 O.n ^{2} / 时间算法来确定最佳双调游览。您可以假设没有两个点具有相同的 x 坐标，并且对实数的所有操作都需要单位时间。（*Hint:* 从左到右扫描，保持游览两个部分的最佳可能性。）

14-4Printing neatly

考虑使用等宽字体（所有字符具有相同的宽度）整齐地打印一个段落的问题。输入文本是 n 个单词的序列，长度为 l $_1$; l $_2$; : : : ; l $_n$ ，以字符为单位，这些单词将整齐地打印在多行上，每行最多包含 M 个字符。没有单词超出行长，因此对于 i D 1; 2; : : : ; n， l $_i$ # M。“整洁度”的标准如下。如果给定行包含单词 i 到 j ，其中 i # j ，并且单词之间恰好出现一个空格，则行尾的额外空格字符数为 M j C i P $_{k=i}^{j}$ l $_k$ ，该值必须为非负数，以便单词 ût 在行上。目标是最小化除最后一行之外所有行的行尾多余空格字符数立方和。给出一个动态规划算法，整齐地打印一段 n 个单词。分析算法的运行时间和空间要求。

14-5Edit distance

为了将源文本字符串 xŒ1 W m　转换为目标字符串 yŒ1 W n　，可以执行各种转换操作。目标是给定 x 和 y，产生一系列将 x 更改为 y 的转换。数组 4假设足够大以保存它需要的所有字符4保存中间结果。最初，z为空，在终止时，应该有 zŒj　D yŒj　，其中 j D 1; 2; : : : ; n。解决这个问题的过程维护 x 中的当前索引 i 和 z中的 j，并且允许操作改变 z和这些索引。最初，i D j D 1。在转换过程中必须检查 x 中的每个字符，这意味着在转换操作序列结束时，i D m C 1。

您可以从六种转换操作中进行选择，每种操作都有一个取决于操作的固定成本：

通过设置 zŒj　D xŒi　，然后增加 i 和 j，将字符从 x 复制到 z。此操作检查 xŒi　，成本为 Q $_C$。

将 x 中的一个字符替换为另一个字符 c，方法是设置 zŒj　D c，然后增加 i 和 j。此操作检查 xŒi　并花费 Q $_R$。

通过增加 i 但不改变 j 来从 x 中删除一个字符。此操作检查 xŒi　，成本为 Q $_D$。

通过设置 zŒj　D c，然后增加 j（但不改变 i）将字符 c 插入到 z中。此操作不检查 x 的任何字符，并且成本为 Q $_I$。

通过将接下来的两个字符从 x 复制到 z但顺序相反来调整（即交换）它们：设置 zŒj　D xŒi C 1　和 zŒj C 1　D xŒi　，然后设置 i D i C 2 和 j D j C 2。此操作检查 xŒi　和 xŒi C 1　，成本为 Q $_T$。

通过设置 i D m C 1 来消除 x 的余数。此操作检查 x 中尚未检查的所有字符。如果执行此操作，则必须是 ûnal 操作。它的成本为 Q $_K$。

图 14.12 给出了将源字符串 algorithm 转换为目标字符串 altruistic 的一种方法。其他几种转换操作序列可以将 algorithm 转换为 altruistic。

假设 Q $_C$ < Q $_D$ C Q $_I$ 和 Q $_R$ < Q $_D$ C Q $_I$，否则，将不会使用复制和替换操作。给定转换操作序列的成本是序列中各个操作成本的总和。对于上述序列，将算法转换为利他主义的成本为 3Q $_C$ C Q $_R$ C Q $_D$ C 4Q $_I$ C Q $_T$ C Q $_K$。

a.给定两个序列 xŒ1 W m　和 yŒ1 W n　以及变换操作的成本，从 x 到 y 的 ***edit distance*** 是最便宜的操作的成本，

Operation	*x*	*z*
initial strings	algorithm	
copy	algorithm	a
copy	algorithm	al
replace by t	algorithm	alt
delete	algorithm	alt
copy	algorithm	altr
insert u	algorithm	altru
insert i	algorithm	altrui
insert s	algorithm	altruis
twiddle	algorithm	altruisti
insert c	algorithm	altruistic
kill	algorithm	altruistic

图 14.12 将源算法转换为目标字符串 altruistic 的运算序列。下划线字符为运算后的 xŒi 和 ŒEj 。

将 x 转换为 y 的编辑序列。描述一个动态规划算法，该算法找出从 xŒ1 W m 到 yŒ1 W n 的编辑距离，并打印出最佳操作序列。分析算法的运行时间和空间要求。

编辑距离问题概括了对齐两个 DNA 序列的问题（例如，参见 Setubal 和 Meidanis [405，第 3.2 节]）。有几种方法可以通过对齐两个 DNA 序列来测量它们的相似性。对齐两个序列 x 和 y 的一种方法包括在两个序列的任意位置（包括两端）插入空格，以使得到的序列 x′ 和 y′ 具有相同的长度，但在相同位置没有空格（即，对于任何位置 j，x′Œj 都不是空格，y′Œj 也不是空格）。然后，我们为每个位置分配一个 “ 分数 ”。位置 j 的得分如下：

C1 如果 x′Œj D y′Œj 且都不是空间，• 1 如果 x′Œj ¤ y′Œj 且都不是空间，• 2 如果 x′Œj 或 y′Œj 是空间。

比对得分是各个位置得分的总和。例如，给定序列 x D GATCGGCAT 和 y D CAATGTGAATC，一个比对是

```
G ATCG GCAT C
AAT GTGAATC
- * ++ * + * +-++
*
```

某个位置下的+表示该位置的得分为C1，-表示得分为1，*表示得分为2，因此该比对的总得分为612142D4。

b.解释如何使用复制、替换、删除、插入、旋转和删除等转换操作的子集将寻找最佳对齐的问题转化为编辑距离问题。

14-6Planning a company party

Blutarsky 教授正在为一家公司总裁提供咨询，该公司正在策划一场公司聚会。该公司采用层级结构，即主管关系形成以总裁为根的树。人力资源部为每位员工的欢乐程度评分，这是一个实数。为了让所有参加者都能享受聚会的乐趣，总裁不希望员工及其直属主管同时参加。

向 Blutarsky 教授提供了一棵描述公司结构的树，使用第 10.3 节中描述的左子节点、右兄弟节点表示法。树的每个节点除了指针之外，还保存员工的姓名和该员工的待客热情等级。描述一种算法来编制一份宾客名单，使宾客待客热情等级的总和最大化。分析算法的运行时间。

14-7Viterbi algorithm

有向图上的动态规划可以在语音识别中发挥作用。带有标记边的有向图 G D .V; E/ 构成了讲受限语言的人的形式模型。每条边 .u; v/ 2 E 都标有来自有限声音集 † 的声音 .u; v/ 。图中从特定顶点 v $_0$ 2 V 开始的每条有向路径都对应于模型产生的可能声音序列，路径的标记是该路径上边的标记的串联。

a.描述一种有效的算法，给定一个边标记有向图 G，其具有可区分的顶点 v $_0$ 和一个来自 † 的声音序列 s D h $_1$； $_2$; : : : ; $_k$ i，返回 G 中从 v $_0$ 开始并以 s 为标签的路径（如果存在这样的路径）。否则，算法应该返回 NO-SUCH-PATH。分析算法的运行时间。（*Hint:* 您可能会发现第 20 章中的概念很有用。）

现在假设每条边 .u; v/ 2 E 都有一个相关的非负概率 p.u; v/ 被遍历，从而产生相应的声音。离开任何顶点的边的概率之和等于 1。路径的概率被定义为其边概率的乘积。想想

从顶点 v_0 开始的路径的概率是从 v_0 开始的 “随机游走” 沿着指定路径的概率，其中离开顶点 u 的边是随机选取的，根据离开 u 的可用边的概率。

b.将你的答案扩展到部分 (a)，以便如果返回一条路径，它是 *most probable path* ，从顶点 v_0 开始，并具有标签 s 。分析算法的运行时间。

14-8Image compression by seam carving

假设您获得了一幅彩色图片，该图片由 mn 像素数组 AŒ1 W m; 1 W n 组成，其中每个像素指定红、绿、蓝 (RGB) 强度的三倍。您想通过从 m 行中各移除一个像素来稍微压缩该图片，以便整个图片变窄一个像素。但是，为了避免不协调的视觉效果，两个相邻行中移除的像素必须位于同一列或相邻列。这样，移除的像素从顶行到底行形成一个 “ 接缝 ”，其中接缝中的连续像素在垂直或对角线上相邻。

a.证明这种可能的接缝的数量在 m 中至少呈指数增长，假设 $n > 1$。

b.现在假设每个像素 AŒi; j 都给出了一个实值破坏度量 d Œi; j ，表示移除像素 AŒi; j 的破坏程度。直观地讲，像素的破坏度量越低，该像素与其相邻像素的相似度就越高。将接缝的破坏度量定义为其像素破坏度量的总和。

给出一个算法来找到具有最低中断度量的接缝。你的算法有多有效？

14-9Breaking a string

某种字符串处理编程语言允许你将字符串拆分成两部分。由于此操作会复制字符串，因此将 n 个字符的字符串拆分成两部分需要花费 n 个时间单位。假设你想将字符串拆分成许多部分。拆分的顺序会影响总使用时间。例如，假设你想在第 2、8 和 10 个字符之后拆分一个 20 个字符的字符串（从左端开始按升序对字符进行编号，从 1 开始）。如果你将拆分编程为按从左到右的顺序进行，那么第一次拆分需要 20 个时间单位，第二次拆分需要 18 个时间单位（在第 8 个字符处将字符串从第 3 个字符拆分到第 20 个字符），第三次拆分需要 12 个时间单位，总共 50 个时间单位。但是，如果你将拆分编程为按从右到左的顺序进行，那么第一次拆分需要 20 个时间单位，

第二次破坏花费 10 个时间单位，第三次破坏花费 8 个时间单位，总共 38 个时间单位。按照另一种顺序，您可以先在 8 处破坏（花费 20），然后在 2 处破坏左侧部分（花费另外 8），最后在 10 处破坏右侧部分（花费 12），总花费为 40。

设计一个算法，给定要在其后中断的字符数，确定对这些中断进行排序的最低成本方法。更正式地讲，给定一个包含 n 个字符的字符串的中断点的数组 LŒ1 W m　，计算中断序列的最低成本以及实现此成本的中断序列。

14-10Planning an investment strategy

您对算法的了解帮助您在一家热门创业公司获得了一份令人兴奋的工作，同时还获得了 10,000 美元的签约奖金。您决定投资这笔钱，以期在 10 年后获得最大回报。您决定聘请投资经理 G. I. Luvcache 来管理您的签约奖金。Luvcache 合作的公司要求您遵守以下规则。它提供 n 种不同的投资，编号为 1 到 n。在每年 j，投资 i 提供的回报率为 r_{ij}。换句话说，如果您在第 j 年在投资 i 中投资 d 美元，那么在第 j 年年底，您将获得 d r_{ij} 美元。回报率是有保证的，也就是说，对于每项投资，您都可以获得未来 10 年的所有回报率。您每年只做一次投资决策。每年年末，您可以将前一年赚到的钱留在相同的投资中，也可以将钱转移到其他投资中，方法是在现有投资之间转移资金或将资金转移到新投资中。如果您在连续两年之间不转移资金，则需要支付 f_1 美元的费用，而如果您转移资金，则需要支付 f_2 美元的费用，其中 $f_2 > f_1$。您每年年末支付一次费用，无论您是将资金转入或转出一项投资，还是转入或转出多项投资，费用都是相同的，即 f_2。

a.如上所述，该问题允许您在每年将资金投入多项投资。证明存在一种最佳投资策略，该策略在每年将所有资金投入一项投资。（回想一下，最佳投资策略会在 10 年后最大化资金量，而不考虑任何其他目标，例如最小化风险。）***b.*** 证明规划最佳投资策略的问题表现出最佳子结构。

c.设计一个算法来规划你的最佳投资策略。你的算法的运行时间是多少？

d.假设 Luvcache 的公司施加了额外的限制，即在任何时候，您的任何一项投资都不能超过 15,000 美元。请证明，在 10 年后最大化您的收入的问题不再表现出最优子结构。

14-11Inventory planning

Rinky Dink 公司生产重铺溜冰场的机器。这类产品的需求每个月都在变化，所以公司需要制定一个策略，在需求虽然波动但可预测的情况下规划生产。公司希望设计一个未来 n 个月的计划。对于每个月 i，公司知道需求 d_i，也就是公司将销售的机器数量。令 D D P $_{i=1}^{n}$ d_i 为未来 n 个月的总需求。公司保留一批全职员工，他们每月提供最多生产 m 台机器的劳动力。如果公司需要在某个月生产多于 m 台机器，它可以雇用额外的兼职劳动力，每台机器的成本为 c 美元。此外，如果公司在月底持有未售出的机器，则必须支付库存成本。公司最多可以持有 D 台机器，持有 j 台机器的成本为函数 h.j / 其中 j D 1; 2；：：：；D 随 j 单调递增。

给出一个算法，计算出一个公司计划，使其成本最小化，同时满足所有需求。运行时间应该是 n 和 D 的多项式。

14-12Signing free-agent baseball players

假设您是一支大联盟棒球队的总经理。在休赛期间，您需要为球队签下一些自由球员。球队老板给了您一笔 $X 的预算，用于签下自由球员。您可以花费少于 $X 的金额，但如果花费超过 $X ，球队老板会惩罚您。

您正在考虑 N 个不同的位置，并且对于每个位置，都有 P 个可以打该位置的自由球员可供选择。[10] 因为您不想让任何位置的球员过多而导致您的阵容超负荷，所以对于每个位置，您最多可以签下一名可以打该位置的自由球员。（如果您没有在某个位置签下任何球员，那么您计划继续使用您已经拥有的该位置的球员。）

---

[10] Although there are nine positions on a baseball team, N is not necessarily equal to 9 because some general managers have particular ways of thinking about positions. For example, a general manager might consider right-handed pitchers and left-handed pitchers to be separate “positions,” as well as starting pitchers, long relief pitchers (relief pitchers who can pitch several innings), and short relief pitchers (relief pitchers who normally pitch at most only one inning).

要确定一名球员的价值，您决定使用棒球统计指标[11]，即“WAR、”或“胜率高于替补。”胜率较高的球员比胜率较低的球员更有价值。签下胜率较高的球员并不一定比签下胜率较低的球员更贵，因为除了球员的价值之外，还有其他因素决定了签下他们的成本。

对于每个可用的自由球员 p，你有三条信息：

球员的位置，• p: *cost*，签下该球员需要花费的金额，以及
• p: *war*，球员的WAR。

设计一个算法，在花费不超过 $X 的情况下最大化您签约球员的总 WAR。您可以假设每个球员的签约金额是 100,000 美元的倍数。您的算法应该输出您签约球员的总 WAR、您花费的总金额以及您签约的球员列表。分析算法的运行时间和空间需求。

章节注释

Bellman [44] 于 1955 年开始系统研究动态规划，并于 1957 年出版了一本相关书籍。“规划”一词在这里和线性规划中都指使用表格求解方法。尽管结合动态规划元素的优化技术早已为人所知，但 Bellman 为该领域提供了坚实的数学基础。

Galil 和 Park [172] 根据表的大小以及每个条目所依赖的其他表条目的数量对动态规划算法进行分类。如果动态规划算法的表大小为 O.n^t/，并且每个条目都依赖于 O.n^e/ 个其他条目，则称该算法为 tD/eD。例如，第 14.2 节中的矩阵链乘法算法为 2D/1D，第 14.4 节中的最长公共子序列算法为 2D/0D。

第 378 页的 MATRIX-CHAIN-ORDER 算法由 Muraoka 和 Kuck [339] 提出。Hu 和 Shing [230, 231] 给出了矩阵链乘法问题的 O.n lg n/-time 算法。

最长公共子序列问题的 O.mn/ 时间算法似乎是一种民间算法。Knuth [95] 提出了次二次算法是否

[11] ***Sabermetrics*** is the application of statistical analysis to baseball records. It provides several ways to compare the relative values of individual players.

存在解决 LCS 问题的算法。Masek 和 Paterson [316] 肯定地回答了这个问题，他们给出了一个运行时间为 O.mn/ lg n/ 的算法，其中 n = m，序列取自一个有界大小的集合。对于输入序列中没有元素出现超过一次的特殊情况，Szymanski [425] 展示了如何在 O..n C m/ lg.n C m// 的时间内解决这个问题。其中许多结果可以扩展到计算字符串编辑距离的问题（问题 14-5）。

Gilbert 和 Moore [181] 发表了一篇关于变长二进制编码的早期论文，该论文包含一个 $O.n^3/$ 时间算法，用于构造所有概率 p_i 为 0 的情况的最优二叉搜索树。Aho、Hopcroft 和 Ullman [5] 介绍了第 14.5 节中的算法。伸展树 [418] 会根据搜索查询修改树，它在未使用频率初始化的情况下位于最优界限的常数因子范围内。练习 14.5-4 是 Knuth [264] 做的。Hu 和 Tucker [232] 设计了一种算法，用于所有概率 p_i 为 0 的情况，该算法使用 $O.n^2/$ 时间和 O.n/ 空间。随后，Knuth [261] 将时间缩短到 O.n lg n/。

问题 14-8 是 Avidan 和 Shamir [30] 的贡献，他们在网上发布了一段精彩的视频，演示了这种图像压缩技术。

15 Greedy Algorithms

优化问题的算法通常要经过一系列步骤，每一步都有一组选择。对于许多优化问题，使用动态规划来确定最佳选择是过度的，更简单、更高效的算法就可以了。***greedy algorithm*** 总是做出目前看起来最好的选择。也就是说，它会做出局部最优的选择，希望这个选择能带来全局最优解。本章探讨了贪婪算法提供最优解的优化问题。在阅读本章之前，您应该阅读第 14 章中有关动态规划的内容，特别是第 14.3 节。

贪婪算法并非总能产生最优解，但对于许多问题来说，它们确实能产生最优解。在第 15.1 节中，我们首先研究一个简单但并非平凡的问题，即活动选择问题，贪婪算法可以有效地计算出该问题的最优解。我们首先考虑一种动态规划方法，然后展示始终做出贪婪选择可以产生最优解，从而得到贪婪算法。第 15.2 节回顾了贪婪方法的基本要素，并给出了证明贪婪算法正确性的直接方法。第 15.3 节介绍了贪婪技术的一个重要应用：设计数据压缩（霍夫曼）码。最后，第 15.4 节表明，为了在缓存中发生未命中时决定替换哪些块，如果事先知道块访问顺序，则 “最远的未来” 策略是最佳的。

贪婪方法非常强大，可以很好地解决各种问题。后面的章节将介绍许多可以视为贪婪方法应用的算法，包括最小生成树算法（第 21 章）、Dijkstra 的单源最短路径算法（第 22.3 节）和贪婪集合覆盖启发式算法（第 35.3 节）。最小生成树算法是贪婪方法的一个经典示例。虽然您可以独立阅读本章和第 21 章，但您可能会发现将它们一起阅读会很有用。

15.1 活动选择问题

我们的第一个例子是调度几项需要独占使用公共资源的竞争活动的问题，目标是选择一组最大大小的相互兼容的活动。想象一下，你负责调度一个会议室。你有 n 个提议的 ***activities*** 的集合 S D fa $_1$; a $_2$; : : : ; a $_n$ g，每个集合都希望预订会议室，并且该房间每次只能用于一项活动。每个活动 a $_i$ 都有 a ***start time*** s $_i$ 和 a ***finish time*** f $_i$，其中 0 = s $_i$ < f $_i$ < 1。如果被选中，活动 a $_i$ 将在半开时间间隔 Œs $_i$; f $_i$/ 内进行。如果间隔 Œs $_i$; f $_i$/ 和 Œs $_j$; f $_j$/ 不重叠，则活动 a $_i$ 和 a $_j$ 为 ***compatible***。也就是说，如果 s $_i$ ¬ f $_j$ 或 s $_j$ ¬ f $_i$，则 a $_i$ 和 a $_j$ 是兼容的。（假设如果你的工作人员需要时间将房间从一个活动切换到下一个活动，则切换时间已计入间隔中。）在 ***activity-selection problem*** 中，你的目标是选择最大大小的相互兼容活动子集。假设活动按完成时间单调递增的顺序排序：

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n \,. \tag{15.1}$$

(我们稍后会看到这一假设带来的优势。)例如，考虑图 15.1 中的活动集。子集 fa $_3$; a $_9$; a $_{11}$ g 由相互兼容的活动组成。但它不是最大子集，因为子集 fa $_1$; a $_4$; a $_8$; a $_{11}$ g 更大。事实上，fa $_1$; a $_4$; a $_8$; a $_{11}$ g 是相互兼容活动的最大子集，另一个最大的子集是 fa $_2$; a $_4$; a $_9$; a $_{11}$ g。

我们将分几个步骤来了解如何解决这个问题。首先，我们将探索一种动态规划解决方案，在确定最佳解决方案中使用哪些子问题时，您需要考虑几种选择。然后，我们会发现您只需考虑一种选择4——贪婪选择4，当您做出贪婪选择时，只剩下一个子问题。基于这些观察，我们将开发一种递归贪婪算法来解决活动选择问题。最后，我们将通过将递归算法转换为迭代算法来完成开发贪婪解决方案的过程。虽然我们在本节中经历的步骤比开发贪婪算法时通常的步骤稍微复杂一些，但它们说明了贪婪算法和动态规划之间的关系。

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

图 15.1 活动集合 fa $_1$; a $_2$; : : : ; a $_{11}$ g。活动 a $_i$ 的开始时间为 s $_i$，结束时间为 f $_i$。

活动选择问题的最优子结构

让我们验证一下活动选择问题是否具有最优子结构。用 S $_{ij}$ 表示在活动 a $_i$ 结束后开始的活动集，以及在活动 a $_j$ 开始之前完成的活动集。假设您想要在 S $_{ij}$ 中找到一组相互兼容的活动，并进一步假设这样的最大集合是 A $_{ij}$，其中包括一些活动 a $_k$。通过将 $_k$ 纳入最佳解决方案，您将面临两个子问题：在集合 S $_{ik}$ (中找到相互兼容的活动，这些活动在活动 a $_i$ 完成后开始，并且在活动 a $_k$ 开始之前完成)；在集合 S $_{kj}$ (中找到相互兼容的活动，这些活动在活动 a $_k$ 完成后开始，并且在活动 a $_j$ 开始之前完成)。设 A $_{ik}$ D A $_{ij}$ \ S $_{ik}$ 和 A $_{kj}$ D A $_{ij}$ \ S $_{kj}$，因此 A $_{ik}$ 包含在 A $_{ij}$ 中在 a $_k$ 开始之前完成的活动，并且 A $_{kj}$ 包含在 A $_{ij}$ 中在 a $_k$ 完成后开始的活动。因此，我们有 A $_{ij}$ D A $_{ik}$ [fa $_k$ g [A $_{kj}$，所以 S $_{ij}$ 中互相兼容活动的最大集合 A $_{ij}$ 由 jA $_{ij}$j D jA $_{ik}$ j C jA $_{kj}$j C 1 个活动组成。

通常的剪切粘贴论证表明，最优解 A $_{ij}$ 还必须包括 S $_{ik}$ 和 S $_{kj}$ 两个子问题的最佳解。如果您可以在 S $_{kj}$ 中找到一组相互兼容的活动 A $'_{kj}$，其中 jA $'_{kj}$j > jA $_{kj}$j，那么您可以在 S $_{ij}$ 子问题的解决方案中使用 A $'_{kj}$ 而不是 A $_{kj}$。您将构建一组 jA $_{ik}$j C jA $'_{kj}$j C 1 > jA $_{ik}$j C jA $_{kj}$j C 1 D jA $_{ij}$j 相互兼容的活动，这与 A $_{ij}$ 是最优解的假设相矛盾。对称论证适用于 S $_{ik}$ 中的活动。

这种描述最优子结构的方法表明，你可以通过动态规划来解决活动选择问题。我们用 cŒi; j 表示集合 S $_{ij}$ 的最优解的大小。然后，动态规划方法给出了递归

$$c[i, j] = c[i, k] + c[k, j] + 1 .$$

当然，如果你不知道集合 S $_{ij}$ 的最优解是否包括活动 a $_k$，则必须检查 S $_{ij}$ 中的所有活动以找出要选择哪一个，以便

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max \{c[i, k] + c[k, j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset . \end{cases} \tag{15.2}$$

然后，您可以开发一个递归算法并对其进行记忆，或者您可以自下而上地进行工作，并在进行过程中填写表格条目。但您会忽略活动选择问题的另一个重要特征，您可以充分利用这一特征。

做出贪婪的选择

如果您可以选择一项活动来添加到最佳解决方案中，而不必先解决所有子问题，会怎么样？这可以让您不必考虑递归（15.2）中固有的所有选择。事实上，对于活动选择问题，您只需考虑一种选择：贪婪选择。

活动选择问题的贪婪选择是什么？直觉告诉你，你应该选择一项活动，让资源尽可能多地用于其他活动。在你最终选择的活动中，其中一项必须是第一个完成的活动。因此，直觉告诉你，选择 S 中完成时间最早的活动，因为这样可以让资源尽可能多地用于其后的活动。（如果 S 中有多个活动完成时间最早，则选择任何这样的活动。）换句话说，由于活动按完成时间单调递增排序，贪婪选择是活动 a_1。选择第一个完成的活动并不是考虑对此问题做出贪婪选择的唯一方法。练习 15.1-3 要求你探索其他可能性。

一旦做出贪婪选择，您只剩下一个子问题需要解决：找到在 a_1 完成后开始的活动。为什么不必考虑在 a_1 开始之前完成的活动？因为 $s_1 < f_1$，并且因为 f_1 是任何活动的最早完成时间，所以没有活动的完成时间可以小于或等于 s_1。因此，与活动 a_1 兼容的所有活动都必须在 a_1 完成后开始。

此外，我们已经确定活动选择问题表现出最优子结构。设 $S_k = \{a_i \in S : s_i \geq f_k\}$ 为活动 a_k 结束后开始的活动集。如果贪婪地选择活动 a_1，则 S_1 仍是唯一需要解决的子问题。[1] 最优子结构表示，如果 a_1 属于最优解，则原始问题的最优解由活动 a_1 和子问题 S_1 最优解中的所有活动组成。

还有一个大问题：这种直觉正确吗？贪婪选择4（即选择第一个活动来完成4）是否总是某种最优解决方案的一部分？以下定理表明确实如此。

---

[1] We sometimes refer to the sets S_k as subproblems rather than as just sets of activities. The context will make it clear whether we are referring to S_k as a set of activities or as a subproblem whose input is that set.

Theorem15.1
考虑任意非空子问题 S_k，令 a_m 为 S_k 中完成时间最早的活动。则 a_m 包含在 S_k 中某个最大大小的相互兼容活动子集中。

Proof令 A_k 为 S_k 中相互兼容活动的最大子集，令 a_j 为 A_k 中完成时间最早的活动。如果 a_j D a_m，则大功告成，因为我们已经证明 a_m 属于 S_k 中相互兼容活动的最大子集。如果 a_j ¤ a_m，则令集合 A'_k D .A_k fa $_j$g/[fa $_m$g 为 A_k，但用 a_m 代替 a_j。 A'_k 中的活动是兼容的，这是因为 A_k 中的活动是兼容的，a_j 是 A_k 中第一个完成的活动，并且 f_m f $_j$。由于 jA'_kj D jA_kj，我们得出结论，A'_k 是 S_k 中相互兼容活动的最大子集，并且它包含 a_m。 ■

虽然你也许能够用动态规划解决活动选择问题，但定理 15.1 指出你不需要这样做。相反，你可以重复选择最先完成的活动，只保留与此活动兼容的活动，并重复进行，直到没有剩余活动。此外，由于你总是选择完成时间最早的活动，因此你选择的活动的完成时间必须严格增加。你可以总体上只考虑每项活动一次，按照完成时间单调递增的顺序。

解决活动选择问题的算法不需要像基于表格的动态规划算法那样自下而上地工作。相反，它可以自上而下地工作，选择一项活动放入它构建的最优解决方案中，然后解决从与已选活动兼容的活动中选择活动的子问题。贪婪算法通常采用这种自上而下的设计：做出选择，然后解决子问题，而不是采用自下而上的方法，即先解决子问题，然后再做出选择。

递归贪婪算法

现在您知道可以绕过动态规划方法，而使用自上而下的贪婪算法，让我们看一个直接的递归程序来解决活动选择问题。下一页上的 RECURSIVE-ACTIVITY-SELECTOR 程序采用活动的开始和结束时间（表示为数组 s 和 f）、[2] 定义要解决的子问题 S_k 的索引 k 以及原始问题的大小 n。它返回一个最大

[2] Because the pseudocode takes s and f as arrays, it indexes into them with square brackets rather than with subscripts.

大小的 S_k 中相互兼容的活动集。该过程假设 n 个输入活动已经根据公式 (15.1) 按单调递增的完成时间排序。如果不是，您可以先在 O.n lg n/ 的时间内按此顺序排列它们，任意打破平局。为了开始，添加活动 a_0 和 f_0 D 0，这样子问题 S_0 就是整个活动集 S。解决整个问题的初始调用是 RECURSIVE-ACTIVITY-SELECTOR .s; f; 0; n/。

递归活动选择器.s; f; k; n/

1 m D k C 1 2 while m n and sŒm < f Œk // 找到 S_k 中的第一个活动以完成 3 m D m C 1 4 if m n 5 return fa $_m$ g [RECURSIVE-ACTIVITY-SELECTOR .s; f; m; n/ 6 else return ;

图 15.2 显示了该算法如何对图 15.1 中的活动进行操作。在给定的递归调用 RECURSIVE-ACTIVITY-SELECTOR .s; f; k; n/ 中，第 233 行的 while 循环查找 S_k 中要完成的第一个活动。该循环检查 a_{k+1} ；a_{k+2} ；: : : ；a_n ，直到找到与 a_k 兼容的第一个活动 a_m ，这意味着 $s_m \approx f_k$ 。如果循环因找到这样的活动而终止，则第 5 行返回 fa $_m$ g 与递归调用 RECURSIVE-ACTIVITY-SELECTOR .s; f; m; n/ 返回的 S_m 的最大大小子集的并集。或者，循环可能因为 m > n 而终止，在这种情况下，该过程已检查了 S_k 中的所有活动，但未找到与 $_k$ 兼容的活动。在这种情况下，S_k D ;，因此第 6 行返回 ;。

假设活动已经按 ûnish 时间排序，调用 RECURSIVE-ACTIVITY-SELECTOR .s; f; 0; n/ 的运行时间为 ，.n/。要了解原因，请观察在所有递归调用中，每个活动在第 2 行的 while 循环测试中只检查一次。特别是，活动 a_i 在最后一次调用中被检查，其中 k < i。

迭代贪婪算法

递归过程可以转换为迭代过程，因为过程 RECURSIVE-ACTIVITY-SELECTOR 几乎是 “ 尾递归 ”（参见问题 7-5）：它以对自身的递归调用结束，然后进行联合操作。将尾递归过程转换为迭代形式通常是一项简单的任务。事实上，某些编程语言的一些编译器会自动执行此任务。

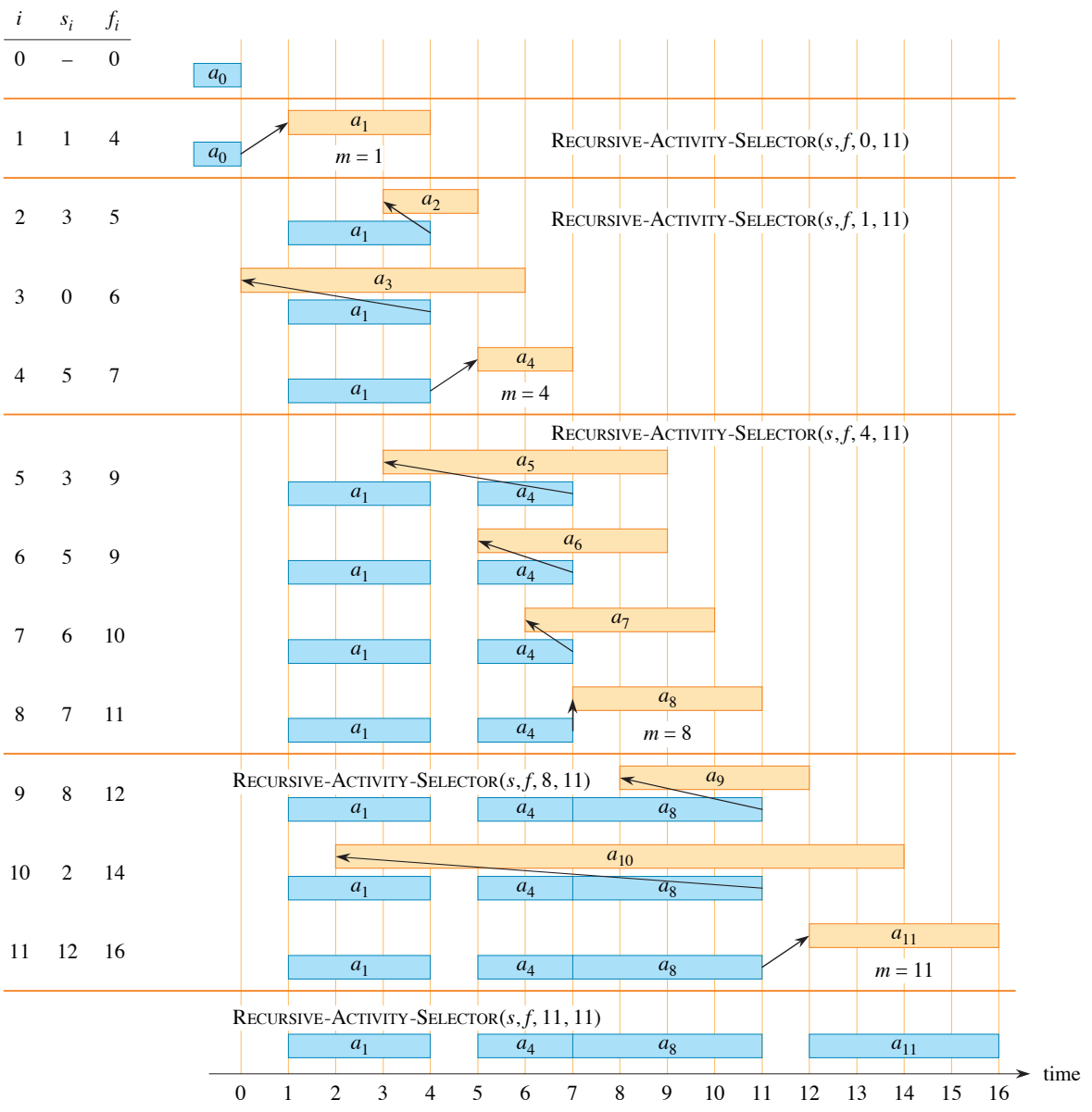


图 15.2 RECURSIVE-ACTIVITY-SELECTOR 对图 15.1 中的 11 个活动的操作。每次递归调用中考虑的活动出现在水平线之间。活动 a 0 在时间 0 完成，初始调用 RECURSIVE-ACTIVITY-SELECTOR.s; f; 0; 11/ 选择活动 a 1 。在每次递归调用中，已经选择的活动为蓝色，正在考虑以棕褐色显示的活动。如果活动的开始时间早于最近添加的活动的结束时间（它们之间的箭头指向左），则拒绝该活动。否则（箭头直接指向上方或右侧），则选择该活动。最后一个递归调用 RECURSIVE-ACTIVITY-SELECTOR.s; f; 11; 11/ 返回；。选定活动的结果集为 f a 1 ; a 4 ; 8 ; 11 g。

GREEDY-ACTIVITY-SELECTOR 程序是 RECURSIVE-ACTIVITY-SELECTOR 程序的迭代版本。它也假设输入活动按单调递增的完成时间排序。它将选定的活动收集到集合 A 中，并在完成后返回该集合。

```
贪婪活动选择器.s; f; n/
1 A D fa 1 g 2 k D 1 3 for m D 2 to n 4 如果 sŒm ¹ f
Œk // 是 S k 中的 m ？ 5 A D A [ fa m g // 是，因此
选择它 6 k D m // 并从那里继续 7 返回 A
```

该过程的工作原理如下。变量 k 索引 A 中的最新添加项，对应于递归版本中的活动 a $_k$。由于该过程按单调递增的完成时间顺序考虑活动，因此 f $_k$ 始终是 A 中任何活动的最大完成时间。也就是说，

$$f_k = \max\{f_i : a_i \in A\} . \tag{15.3}$$

132 行选择活动 a 1，初始化 A 以仅包含此活动，并初始化 k 以索引此活动。336 行的 for 循环查找 S $_k$ 中最早完成的活动。循环依次考虑每个活动 a $_m$，如果活动 a $_m$ 与所有先前选定的活动兼容，则将其添加到 A。这样的活动是 S $_k$ 中最早完成的活动。要查看活动 a $_m$ 是否与 A 中当前的每个活动兼容，只需通过公式 (15.3) 检查（第 4 行）其开始时间 s $_m$ 不早于最近添加到 A 的活动的完成时间 f $_k$ 即可。如果活动 a $_m$ 兼容，则 536 行将活动 a $_m$ 添加到 A 并将 k 设置为 m。调用 GREEDY-ACTIVITY-SELECTOR .s; f / 返回的集合 A 恰好是初始调用 RECURSIVE-ACTIVITY-SELECTOR .s; f; 0; n/ 返回的集合。

和递归版本一样，GREEDY-ACTIVITY-SELECTOR 在 ¸n 时间内安排一组 n 个活动，假设这些活动已经根据其完成时间进行了初步排序。

练习

15.1-1

给出一个基于递归 (15.2) 的活动选择问题的动态规划算法。让您的算法计算上述定义的 cŒi; j 的大小，并生成相互兼容活动的最大子集。

假设输入已经按照公式（15.1）排序。将你的解决方案的运行时间与GREEDY-ACTIVITY-SELECTOR的运行时间进行比较。

15.1-2
假设您不总是选择第一个要完成的活动，而是选择最后一个与所有先前选择的活动兼容的活动来开始。描述这种方法是一种贪婪算法，并证明它能产生最优解。

15.1-3
并非任何贪婪的活动选择问题方法都能产生最大规模的相互兼容活动集。请举一个例子来说明从与先前选择的活动兼容的活动中选择持续时间最短的活动的方法不起作用。对于始终选择与其他剩余活动重叠最少的兼容活动和始终选择开始时间最早的兼容剩余活动的方法，请执行相同的操作。

15.1-4
您需要在大量演讲厅中安排一组活动，其中任何活动都可以在任意演讲厅进行。您希望使用尽可能少的演讲厅来安排所有活动。请给出一个有效的贪婪算法来确定哪些活动应该使用哪个演讲厅。

（该问题也称为 ***interval-graph coloring problem***。它由一个区间图建模，该区间图的顶点是给定的活动，其边连接不兼容的活动。为每个顶点着色所需的最小颜色数，使得没有两个相邻的顶点具有相同的颜色，对应于找到安排所有给定活动所需的最少演讲厅。）

15.1-5
考虑活动选择问题的一种修改，其中每个活动 a_i 除了开始和结束时间之外，还有一个值 v_i。目标不再是最大化计划的活动数量，而是最大化计划活动的总价值。也就是说，目标是选择一组兼容的活动 A，使得 $\sum_{a_k \in A} v_k$ 最大化。给出该问题的多项式时间算法。

15.2 贪婪策略的要素

贪婪算法通过做出一系列选择来获得问题的最佳解决方案。在每个决策点，算法都会做出当时看来最好的选择。这种启发式策略并不总是能产生最佳解决方案，但就像在活动选择问题中一样，有时它会这样做。本节讨论贪婪方法的一些一般属性。

我们在 15.1 节中遵循的开发贪婪算法的过程比典型的过程要复杂一些。它包括以下步骤：

1. 确定问题的最优子结构。2. 开发递归解决方案。（对于活动选择问题，我们制定了递归（15.2），但绕过了仅基于此递归开发递归算法。）3. 表明如果做出贪婪选择，则只剩下一个子问题。4. 证明做出贪婪选择总是安全的。（步骤 3 和 4 可以按任意顺序进行。）5. 开发实现贪婪策略的递归算法。6. 将递归算法转换为迭代算法。

这些步骤非常详细地强调了贪婪算法的动态规划基础。例如，活动选择问题的第一个切入点定义了子问题 S_{ij}，其中 i 和 j 都是变化的。然后我们发现，如果你总是做出贪婪选择，你可以将子问题限制为 S_k 的形式。

另一种方法是考虑贪婪选择来设计最佳子结构，这样选择后只需解决一个子问题。在活动选择问题中，首先删除第二个下标并定义 S_k 形式的子问题。然后证明贪婪选择（第一个活动 a_m 到 S_k 中完成）与剩余兼容活动集 S_m 的最优解相结合，可产生 S_k 的最优解。更一般地，您可以按照以下步骤顺序设计贪婪算法：

1. 将优化问题视为一个你做出选择并剩下一个子问题需要解决的问题。2. 证明对于做出贪婪选择的原始问题始终存在最优解，因此贪婪选择始终是安全的。3. 通过展示做出贪婪选择后剩下的子问题来展示最优子结构，该子问题具有以下特性：如果你将一个

子问题的最优解与你所做的贪婪选择，你会得到原始问题的最优解。

本章后面的部分将使用这种更直接的过程。然而，在每一个贪婪算法背后，几乎总是有一个更麻烦的动态规划解决方案。

如何判断贪婪算法是否能解决特定的优化问题？没有一种方法总是有效的，但贪婪选择属性和最优子结构是两个关键因素。如果你能证明问题具有这些属性，那么你就已经在为它开发贪婪算法的道路上迈出了一大步。

贪婪选择性质

第一个关键要素是 ***greedy-choice property***：您可以通过做出局部最优（贪婪）选择来组合全局最优解决方案。换句话说，当您考虑做出哪种选择时，您会做出在当前问题中看起来最好的选择，而不考虑子问题的结果。

这就是贪婪算法与动态规划不同的地方。在动态规划中，你在每一步都要做出选择，但这个选择通常取决于子问题的解。因此，你通常以自下而上的方式解决动态规划问题，从较小的子问题进展到较大的子问题。（或者，你可以自上而下地解决它们，但要记忆。当然，即使代码是自上而下的，你仍然必须在做出选择之前解决子问题。）在贪婪算法中，你做出当前看起来最好的选择，然后解决剩下的子问题。贪婪算法所做的选择可能取决于迄今为止的选择，但不能取决于任何未来的选择或子问题的解。因此，与先解决子问题再做出第一次选择的动态规划不同，贪婪算法在解决任何子问题之前都会做出它的第一个选择。动态规划算法自下而上进行，而贪婪策略通常自上而下进行，不断做出一个又一个贪婪选择，将每个给定的问题实例简化为更小的实例。

当然，你需要证明每一步的贪婪选择都会产生全局最优解。通常，就像定理 15.1 的情况一样，证明会检查某个子问题的全局最优解。然后，它会展示如何修改解决方案，用其他选择代替贪婪选择，从而得到一个类似但较小的子问题。

通常，与必须考虑更广泛的选择相比，贪婪选择可以更有效地进行。例如，在活动选择问题中，假设活动已经按 ûnish 时间按单调递增顺序排序，则每个活动只需检查一次。通过预处理

输入或通过使用适当的数据结构（通常是优先级队列），您通常可以快速做出贪婪选择，从而产生有效的算法。

最优子结构

如我们在第 14 章中看到的那样，如果问题的最优解包含子问题的最优解，则问题表现出 ***optimal substructure***。此属性是评估动态规划是否适用的关键因素，对于贪婪算法也至关重要。作为最优子结构的一个例子，回想一下第 15.1 节如何证明，如果子问题 S_{ij} 的最优解包括活动 a_k，那么它还必须包含子问题 S_{ik} 和 S_{kj} 的最优解。给定这个最优子结构，我们认为如果知道将哪个活动用作 k，则可以通过选择 k 以及子问题 S_{ik} 和 S_{kj} 最优解中的所有活动来构建 S_{ij} 的最优解。对最优子结构的这种观察导致了描述最优解值的递归 (15.2)。

在将最优子结构应用于贪婪算法时，您通常会使用更直接的方法。如上所述，您可以假设您通过在原始问题中做出贪婪选择而得出子问题。您真正需要做的就是论证子问题的最优解与已经做出的贪婪选择相结合会产生原始问题的最优解。该方案隐式地对子问题使用归纳法来证明在每一步做出贪婪选择都会产生最优解。

贪婪规划与动态规划

由于贪婪和动态规划策略都利用了最优子结构，因此当贪婪解决方案足够时，您可能会倾向于生成问题的动态规划解决方案，或者相反，您可能会误以为贪婪解决方案有效，而实际上需要动态规划解决方案。为了说明这两种技术之间的细微差别，让我们研究一下经典优化问题的两种变体。

0-1 knapsack problem 如下。抢劫商店的小偷想要拿走背包中价值最高的物品，背包最多可以装 W 磅赃物。小偷可以选择拿走商店中 n 件物品的任意子集。第 i 件物品价值 v_i 美元，重 w_i 磅，其中 v_i 和 w_i 为整数。小偷应该拿走哪些物品？（我们称之为 0-1 背包问题，因为对于每件物品，小偷必须拿走或留下。小偷不能拿走物品的零头，也不能拿走一件物品超过一次。）

在 ***fractional knapsack problem*** 中，设置相同，但小偷可以拿走部分物品，而不必对每件物品进行二进制 (0-1) 选择。您可以将 0-1 背包问题中的物品视为金锭，而分数背包问题中的物品则更像是金粉。

两个背包问题都表现出最优子结构性质。对于 0-1 问题，如果最有价值的负载最多为 W 磅，其中包含物品 j，则剩余负载必定是小偷可以从除物品 j 之外的 n 1 个原始物品中拿走的最多重量为 W w_j 磅的最有价值的负载。对于可比较的分数问题，如果最有价值的负载最多为 W 磅，其中包含重量为 w 的物品 j，则剩余负载必定是小偷可以从 n 件原始物品加上重量为 w_j w 磅的物品 j 中拿走的最多重量为 W w 磅的最有价值的负载。

尽管问题相似，但贪婪策略可以解决分数背包问题，但不能解决 0-1 问题。要解决分数问题，首先计算每件物品每磅的价值 v_i/w_i。遵循贪婪策略，小偷首先尽可能多地拿走每磅价值最高的物品。如果该物品耗尽而小偷还能携带更多，那么小偷会尽可能多地拿走每磅价值第二高的物品，依此类推，直到达到重量限制 W。因此，通过按每磅价值对物品进行排序，贪婪算法的运行时间为 O.n lg n/。练习 15.2-1 要求你证明分数背包问题具有贪婪选择属性。

要了解这种贪婪策略不适用于 0-1 背包问题，请考虑图 15.3(a) 中所示的问题实例。此示例有三件物品和一个可容纳 50 磅的背包。物品 1 重 10 磅，价值 60 美元。物品 2 重 20 磅，价值 100 美元。物品 3 重 30 磅，价值 120 美元。因此，物品 1 每磅的价值为 6 美元，大于物品 2（每磅 5 美元）或物品 3（每磅 4 美元）的每磅价值。因此，贪婪策略将首先选择物品 1。但是，从图 15.3(b) 中的案例分析可以看出，最佳解决方案选择物品 2 和 3，而将物品 1 留在后面。选择物品 1 的两个可能解决方案都是次优的。

然而，对于类似的分数问题，贪婪策略（首先选择物品 1）确实会产生最佳解决方案，如图 15.3(c) 所示。在 0-1 问题中，选择物品 1 不起作用，因为小偷无法将背包装满，而空的空间会降低每磅负载的有效价值。在 0-1 问题中，当你考虑是否将某件物品放入背包时，你必须将包含该物品的子问题的解与不包含该物品的子问题的解进行比较，然后才能做出选择。以这种方式表述的问题会产生许多重叠的子问题

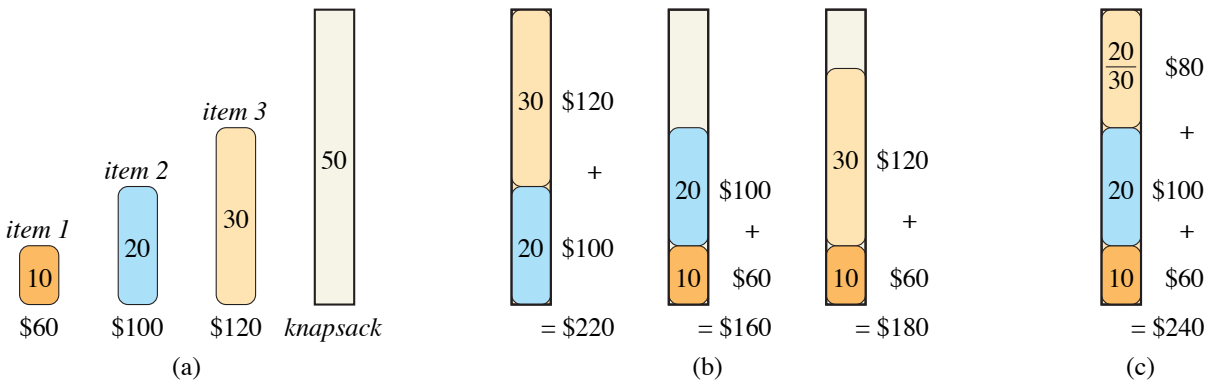


图 15.3 一个例子表明贪婪策略对 0-1 背包问题不起作用。（a）小偷必须从所示的三个物品中选择一个子集，其重量不得超过 50 磅。（b）最佳子集包括物品 2 和 3。任何包含物品 1 的解决方案都是次优的，即使物品 1 的每磅价值最高。（c）对于分数背包问题，按每磅价值最高的顺序选择物品可得到最佳解决方案。

问题4动态规划的标志，事实上，正如练习 15.2-2 要求您展示的那样，您可以使用动态规划来解决 0-1 问题。

练习

15.2-1
证明分数背包问题具有贪婪选择性质。

15.2-2
给出 0-1 背包问题的动态规划解决方案，运行时间为 O.n W / 次，其中 n 是物品数量，W 是小偷可以放入背包中的物品的最大重量。

15.2-3
假设在 0-1 背包问题中，物品按重量增加排序时的顺序与按价值减少排序时的顺序相同。请给出一种有效的算法来找到这种背包问题变体的最优解决方案，并论证你的算法是正确的。

15.2-4
盖科教授一直梦想着滑着直排轮滑穿越北达科他州。教授计划沿着美国 2 号公路穿越该州，这条公路从明尼苏达州东部边界的大福克斯一直延伸到蒙大拿州西部边界附近的威利斯顿。教授可以携带两升水，滑行 1000 公里后

缺水。（由于北达科他州相对平坦，教授不必担心上坡路段的饮水量会比平坦或下坡路段大。）教授将从大福克斯出发，带上两升水。教授有一张北达科他州官方地图，上面标出了美国 2 号公路沿线所有可以补充水的地方以及这些地点之间的距离。

教授的目标是尽量减少穿越州的路线上的补水站数量。请给出一种有效的方法，让教授能够确定在哪些地方设置补水站。请证明你的策略可以产生最佳解决方案，并给出其运行时间。

15.2-5

描述一种有效的算法，给定实线上的一组点 $\{x_1, x_2, \ldots, x_n\}$，确定包含所有给定点的最小单位长度闭区间集。论证你的算法是正确的。

15.2-6

说明如何在 $O(n)$ 时间内解决分数背包问题。

15.2-7

给定两个集合 A 和 B，每个集合包含 n 个正整数。你可以选择按你喜欢的方式重新排序每个集合。重新排序后，令 a_i 为集合 A 的第 i 个元素，令 b_i 为集合 B 的第 i 个元素。然后你将获得收益 $\prod_{i=1}^{n} a_i^{b_i}$。给出一个最大化收益的算法。证明你的算法最大化收益，并说明其运行时间（忽略重新排序集合的时间）。

15.3 霍夫曼编码

霍夫曼编码可以很好地压缩数据：通常可以节省 20% 到 90% 的数据，具体取决于被压缩数据的特征。数据以字符序列的形式到达。霍夫曼贪婪算法使用一个表格来给出每个字符出现的频率（其频率），以建立将每个字符表示为二进制字符串的最佳方式。

假设你有一个包含 100,000 个字符的数据文件，希望将其紧凑地存储起来，并且你知道文件中 6 个不同字符的出现频率如图 15.4 所示。字符 a 出现了 45,000 次，字符 b 出现了 13,000 次，依此类推。

对于如何表示这样的信息域，您有很多选择。在这里，我们考虑设计 ***binary character code***（或简称为 ***code***）的问题

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

图 15.4 字符编码问题。一个包含 100,000 个字符的数据文件仅包含字符 a – f，其频率已标明。由于每个字符由 3 位代码字表示，因此对数据文件进行编码需要 300,000 位。使用所示的可变长度代码，编码仅需要 224,000 位。

其中每个字符都由一个唯一的二进制字符串表示，我们将其称为 ***codeword***。如果使用 ***fixed-length code***，则需要 dlg ne 位来表示 n 2 个字符。因此，对于 6 个字符，需要 3 位：a = 000、b = 001、c = 010、d = 011、e = 100 和 f = 101。此方法需要 300,000 位来编码整个 ûle。您能做得更好吗？

variable-length code 的效果比固定长度代码好得多。这个想法很简单：给频繁出现的字符短代码字，给不频繁出现的字符长代码字。图 15.4 显示了这样的代码。这里，1 位字符串 0 表示 a，4 位字符串 1100 表示 f。此代码需要

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

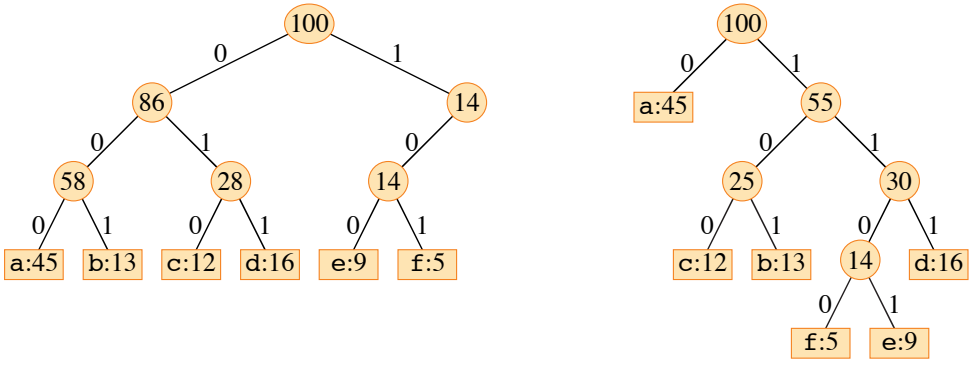
来表示 ûle，节省了大约 25%。事实上，正如我们将看到的，这是此 ûle 的最佳字符代码。

无前缀代码

我们在此仅考虑其中没有代码字也是其他代码字的前缀的代码。此类代码称为 ***prefix-free codes***。虽然我们不会在此证明这一点，但无前缀代码始终可以在任何字符代码中实现最佳数据压缩，因此我们将注意力限制在无前缀代码上不会失去通用性。

对于任何二进制字符代码，编码总是很简单：只需将表示字母表每个字符的代码字连接起来即可。例如，对于图 15.4 中的可变长度无前缀代码，4 个字符的字母表面的编码为 1100 - 0 - 100 - 1101 D 110001001101，其中 “-” 表示连接。

无前缀代码是理想的，因为它们简化了解码。由于没有代码字是其他代码字的前缀，因此编码字开头的代码字是无歧义的。您可以简单地识别初始代码字，将其转换回原始字符，然后对编码字的其余部分重复解码过程。在我们的示例中，字符串 100011001101 唯一解析为 100011001101，解码为 cafe。



(a) (b)

图 15.5 与图 15.4 中的编码方案相对应的树。每片叶子都标有一个字符及其出现频率。每个内部节点都标有其子树中叶子的频率总和。所有频率均以千为单位。（a）对应于固定长度代码 a = 000、b = 001、c = 010、d = 011、e = 100、f = 101 的树。（b）对应于最佳无前缀代码 a = 0、b = 101、c = 100、d = 111、e = 1101、f = 1100 的树。

解码过程需要一种方便的无前缀代码表示，以便您可以轻松地挑选出初始代码字。二叉树提供了一种这样的表示，其叶子是给定的字符。将字符的二进制代码字解释为从根到该字符的简单路径，其中 0 表示 “转到左子节点”，1 表示 “转到右子节点。” 图 15.5 显示了我们示例中的两个代码的树。请注意，这些不是二叉搜索树，因为叶子不必按排序顺序出现，并且内部节点不包含字符键。

一个字母的最优编码总是用一棵 *full* 二叉树表示，其中每个非叶子节点都有两个子节点（见练习 15.3-2）。我们例子中的固定长度编码不是最优的，因为它的树（如图 15.5(a) 所示）不是满二叉树：它包含以 10 开头的代码字，但没有以 11 开头的代码字。由于我们现在可以将注意力限制在满二叉树上，我们可以说，如果 C 是字符的字母表，并且所有字符频率均为正，则最优无前缀编码的树恰好有 jC j 个叶子节点，每个字母一个，并且恰好有 jC j 1 个内部节点（见第 1175 页的练习 B.5-3）。

给定一个对应于无前缀代码的树 T，我们可以计算编码一个字母所需的位数。对于字母表 C 中的每个字符 c，让属性 c: *freq* 表示字母 c 中出现的频率，让 d $_T$.c/ 表示树中 c 叶子的深度。请注意，d $_T$.c/ 也是字符 c 的代码字长度。因此，编码一个字母所需的位数为

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) \,, \tag{15.4}$$

我们将其定义为树 T 的 ***cost***。

构造霍夫曼编码

霍夫曼发明了一种贪婪算法，可以构造最优的无前缀代码，为了纪念他，我们将其称为 ***Huffman code***。根据我们在第 15.2 节中的观察，其正确性证明依赖于贪婪选择属性和最优子结构。我们不会先证明这些属性成立，然后再开发伪代码，而是先给出伪代码。这样做将有助于阐明算法如何进行贪婪选择。

HUFFMAN 过程假设 C 是一组 n 个字符，每个字符 c 2 C 都是一个具有属性 c:*freq* 的对象，该属性给出其频率。该算法自下而上地构建与最佳编码相对应的树 T。它从一组 jC j 叶子开始，执行一系列 jC j 1 “合并” 操作来创建 ûnal 树。该算法使用以 *freq* 属性为关键字的最小优先级队列 Q 来识别要合并在一起的两个最不频繁的对象。合并两个对象的结果是一个新的对象，其频率是合并后的两个对象的频率之和。

HUFFMAN.C /

1 n D jC j 2 Q D C 3 for i D 1 to n 1 4 分配一个新节点z 5 x D EXTRACT-MIN .Q/ 6 y D EXTRACT-MIN .Q/ 7 *z.left* D x 8 *z.right* D y 9 *z.freq* D x: *freq* C y: *freq* 10 I NSERT.Q; z 11 return EXTRACT-MIN.Q/ // 树的根是唯一剩下的节点

对于我们的例子，霍夫曼算法的流程如图 15.6 所示。由于字母表包含 6 个字母，因此初始队列大小为 n D 6，5 个合并步骤构建树。最终树表示最佳无前缀代码。字母的代码字是从根到字母的简单路径上的边标签序列。

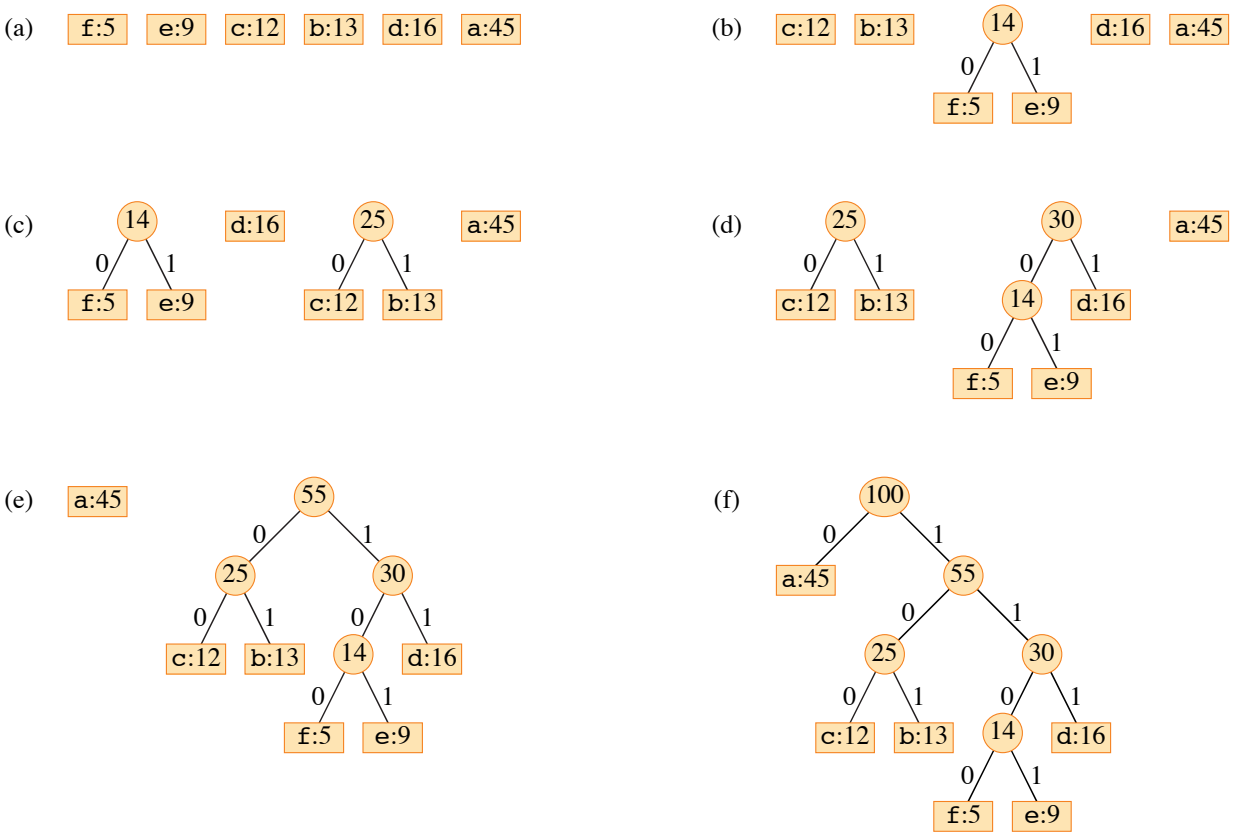


图 15.6 针对图 15.4 中给出的频率，霍夫曼算法的步骤。每个部分都按频率升序排列了队列的内容。每个步骤都会将频率最低的两棵树合并起来。叶子节点显示为包含字符及其频率的矩形。内部节点显示为包含其子节点频率总和的圆圈。如果连接内部节点及其子节点的边是通向左子节点的边，则标记为 0；如果是通向右子节点的边，则标记为 1。字母的代码字是连接根节点和该字母的叶子节点的边上的标签序列。 (a) 初始的 n D 6 个节点集，每个字母一个。 (b) – (e) 中间阶段。 (f) ûnal 树。

HUFFMAN 过程的工作原理如下。第 2 行用 C 中的字符初始化最小优先级队列 Q。第 3310 行的 for 循环重复从队列中提取频率最低的两个节点 x 和 y，并用表示它们合并的新节点 *z* 替换它们。*z* 的频率计算为第 9 行中 x 和 y 频率的总和。节点 *z* 的左子节点是 x，右子节点是 y。（此顺序是任意的。交换任何节点的左子节点和右子节点都会产生具有相同成本的不同代码。）经过 n 1 次合并后，第 11 行返回队列中剩下的一个节点，该节点是代码树的根。

该算法在没有变量 x 和 y 的情况下产生相同的结果，将 EXTRACT-MIN 调用返回的值直接分配给第 7 行和第 8 行中的 *zleft* 和 *zright* ，并将第 9 行更改为 *zfreq* D *zleft* : *freq* C *zright* : *freq*。不过，我们将在正确性证明中使用节点名称 x 和 y，因此我们保留它们。

霍夫曼算法的运行时间取决于最小优先级队列 Q 的实现方式。我们假设它以二进制最小堆的形式实现（参见第 6 章）。对于包含 n 个字符的集合 C，第 6.3 节讨论的 BUILD-MIN-HEAP 过程可以在 O.n/ 的时间内初始化第 2 行中的 Q。第 33-10 行中的 for 循环恰好执行 n 1 次，并且由于每个堆操作都需要 O.lg n/ 的时间，因此该循环对运行时间的贡献为 O.n lg n/。因此，HUFFMAN 在包含 n 个字符的集合上的总运行时间为 O.n lg n/。

霍夫曼算法的正确性

为了证明贪婪算法 HUFFMAN 是正确的，我们将证明确定最优无前缀代码的问题表现出贪婪选择和最优子结构性质。下一个引理表明贪婪选择性质成立。

Lemma15.2 (Optimalprefix-free codes have the greedy-choice p roperty)
假设 C 是一个字母表，其中每个字符 c 2 C 的频率为 c：*freq*。假设 x 和 y 是 C 中频率最低的两个字符。那么存在一个最优的无前缀码，其中 x 和 y 的码字长度相同，仅在最后一位不同。

Proof证明的思路是取代表任意最优无前缀码的树 T，并对其进行修改，使其代表另一个最优无前缀码，使得字符 x 和 y 在新树中作为最大深度的兄弟叶出现。在这样的树中，x 和 y 的码字长度相同，仅在最后一位不同。

假设 a 和 b 是任意两个字符，它们是 T 中最大深度的兄弟叶子。不失一般性，假设 a: *freq* b: *freq* 和 x: *freq* y: *freq*。由于 x: *freq* 和 y: *freq* 是两个最低叶子频率（按顺序），而 a: *freq* 和 b: *freq* 是两个任意频率（按顺序），因此我们有 x: *freq* a: *freq* 和 y: *freq* b: *freq*。

在证明的剩余部分中，我们可能有 x: *freq* D a: *freq* 或 y: *freq* D b: *freq*，但 x: *freq* D b: *freq* 意味着 a: *freq* D b: *freq* D x: *freq* D y: *freq* (参见练习 15.3-1)，引理显然为真。因此，假设 x: *freq* ¤ b: *freq*，这意味着 x ¤ b。

如图 15.7 所示，想象交换 T 中 a 和 x 的位置以生成一棵树 T′，然后交换 T′ 中 b 和 y 的位置以生成一棵

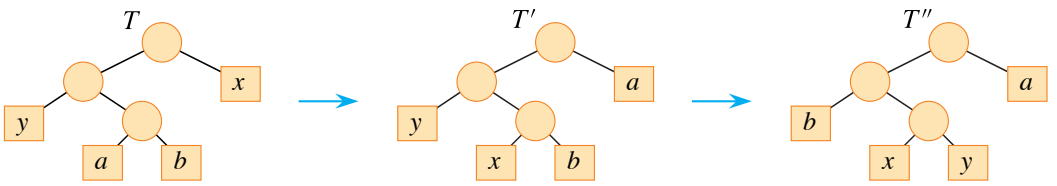


图 15.7 引理 15.2 证明中关键步骤的说明。在最优树 T 中，叶子 a 和 b 是两个最大深度的兄弟。叶子 x 和 y 是出现频率最低的两个字符。它们出现在 T 中的任意位置。假设 x ¤ b，交换叶子 a 和 x 会产生树 T′，然后交换叶子 b 和 y 会产生树 T″。由于每次交换都不会增加成本，因此得到的树 T″ 也是最优树。

树 T″ 中 x 和 y 是最大深度的兄弟叶子。（注意，如果 x D b 但 y ¤ a，则树 T″ 没有 x 和 y 作为最大深度的兄弟叶子。因为我们假设 x ¤ b，所以这种情况不会发生。）根据公式（15.4），T 和 T′ 之间的成本差异为

$$
\begin{aligned}
&乙(T) - B(T') \\
&\quad = \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&\quad = (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\quad \geq 0 ,
\end{aligned}
$$

因为 a: *freq* x: *freq* 和 d $_T$.a/ d $_T$.x/ 均为非负数。更具体地说，a: *freq* x: *freq* 为非负数，因为 x 是最小频率叶子节点，而 d $_T$.a/ d $_T$.x/ 为非负数，因为 a 是 T 中最大深度叶子节点。类似地，交换 y 和 b 不会增加成本，因此 B.T′B .T″为非负数。因此，B.T″#B.T′#B.T /，并且由于 T 是最优的，我们有 B.T /#B.T″，这意味着 B.T″D B.T /。因此，T″ 是一棵最优树，其中 x 和 y 作为最大深度的兄弟叶子出现，引理由此得出。

■

引理 15.2 意味着，通过合并构建最优树的过程可以不失一般性地从贪婪选择合并频率最低的两个字符开始。为什么这是一个贪婪选择？我们可以将单个合并的成本视为合并的两个项目的频率之和。练习 15.3-4 表明，构建树的总成本等于其合并成本之和。在每一步的所有可能合并中，HUFFMAN 选择成本最低的合并。

下一个引理表明，构造最优无前缀码的问题具有最优子结构性质。

Lemma15.3 (Optimal prefix-free codes have the optimal-substructure property) 假设 C 为给定字母表，其频率为 c：*freq*，对每个字符 c 2 C 定义。假设 x 和 y 为 C 中频率最小的两个字符。假设 C′ 为字母表 C 中移除字符 x 和 y 并添加一个新字符 *z*，使得 C′ D .C fx; y g/ [fg. Deûne *freq*，对 C′ 中所有与 C 中值相同的字符定义，以及 *z*: *freq* D x：*freq* C y：*freq*。假设 T′ 为任意一棵树，表示字母表 C′ 的最佳无前缀码。然后，通过将 T′ 的叶节点替换为以 x 和 y 作为子节点的内部节点，可以获得树 T，该树表示字母表 C 的最佳无前缀编码。

Proof 首先，我们通过考虑公式 (15.4) 中的组件成本，说明如何根据树 T′ 的成本 B.T′/ 来表示树 T 的成本 B.T /。对于每个字符 c 2 C fx; y g，我们有 d $_T$.c/ D d $_{T'}$.c/，因此 c: *freq* d $_T$.c/ D c: *freq* d $_{T'}$.c/。由于 d $_T$.x/ D d $_T$.y/ D d $_{T'}$. *z*C 1，我们有

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) , \end{aligned}$$

由此我们得出结论

$$B(T) = B(T') + x.freq + y.freq$$

或者，等价地，

$$B(T') = B(T) - x.freq - y.freq .$$

我们现在用反证法证明引理。假设 T 不代表 C 的最优无前缀代码。那么存在一个最优树 T″，使得 B.T″/ < B.T /。不失一般性（根据引理 15.2），T″ 有 x 和 y 作为兄弟。让 T‴ 成为树 T″，其中 x 和 y 的共同父代被频率为 *z* 的叶子 *z* 替换：*freq* D x: *freq* C y: *freq*。那么

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') , \end{aligned}$$

这与假设 T′ 表示 C′ 的最佳无前缀代码相矛盾。因此，T 必须表示字母表 C 的最佳无前缀代码。

■

Theorem15.4
HUFFMAN 程序生成最佳的无前缀码。

Proof从引理 15.2 和 15.3 直接得出。 ■

练习

15.3-1
解释为什么在引理 15.2 的证明中，如果 x: *freq* D b: *freq*，那么我们就必须有 a: *freq* D b: *freq* D x: *freq* D y: *freq*。

15.3-2证明非满二叉树不能对应最优无前缀码。

15.3-3根据前 8 个斐波那契数，以下频率集的最佳霍夫曼编码是什么？

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21
当频率是前 n 个斐波那契数时，你能否概括你的答案以找到最佳代码？

15.3-4
证明：对于一个代码来说，一棵满二叉树 T 的总成本 B.T / 等于所有内部节点中该节点的两个子节点的频率总和。

15.3-5
给定一个由 n 个字符组成的集合 C 上的最优无前缀代码，您希望使用尽可能少的位来传输代码本身。说明如何仅使用 2n 1 C n dlg ne 位来表示 C 上的任何最优无前缀代码。（*Hint:* 使用 2n 1 位来指定树的结构，如通过遍历树发现的那样。）

15.3-6
将 Huffman 算法推广到三元码字（即使用符号 0、1 和 2 的码字），并证明它能产生最佳三元码。

15.3-7
数据文件包含一个 8 位字符序列，其中所有 256 个字符的出现频率大致相同：最大字符频率小于最小字符频率的两倍。请证明，在这种情况下，霍夫曼编码并不比使用普通的 8 位固定长度代码更有效。

15.3-8
证明没有无损（可逆）压缩方案可以保证对于每个输入文件，对应的输出文件更短。（*Hint:* 将可能的文件数量与可能的编码文件数量进行比较。）

15.4 离线缓存

计算机系统可以通过将主内存的子集存储在 ***cache***（一种较小但速度更快的内存）中来减少访问数据的时间。缓存将数据组织成 ***cache blocks***，通常包含 32、64 或 128 个字节。您还可以将主内存视为虚拟内存系统中磁盘驻留数据的缓存。这里的块称为 ***pages***，典型大小为 4096 字节。

计算机程序在执行时会发出一系列内存请求。假设有 n 个内存请求，请求顺序为块 b_1；b_2；:::；b_n 中的数据。访问序列中的块可能不是不同的，实际上，任何给定的块通常会被访问多次。例如，访问四个不同块 p；q；r；s 的程序可能会对块 s；q；s；q；q；s；p；p；r；s；s；q；p；r；q。缓存最多可容纳固定数量的 k 个缓存块。在第一个请求之前，缓存为空。每个请求最多会导致一个块进入缓存，最多会导致一个块从缓存中逐出。在请求块 b_i 时，可能会发生以下三种情况中的任何一种：

1. 由于先前请求了同一个块，块 b_i 已在缓存中。缓存保持不变。这种情况称为 ***cache hit***。2. 块 b_i 当时不在缓存中，但缓存包含的块少于 k 个。在这种情况下，块 b_i 被放入缓存中，因此缓存包含的块比请求之前多一个。3. 块 b_i 当时不在缓存中，并且缓存已满：它包含 k 个块。块 b_i 被放入缓存中，但在此之前，必须从缓存中逐出一些其他块以腾出空间。

后两种情况，即请求的块尚未在缓存中，称为 ***cache misses***。目标是在整个 n 个请求序列中最小化缓存未命中次数，或者等效地最大化缓存命中次数。在缓存中保存的块少于 k 个时发生的缓存未命中4，即缓存首次被填满4，称为 ***compulsory miss***，因为先前没有决定可以将请求的块保留在缓存中。当发生缓存未命中并且缓存已满时，理想情况下，选择要逐出的块应该允许在整个未来请求序列中缓存未命中的次数尽可能少。

通常，缓存是一个在线问题。也就是说，计算机必须决定在不知道未来请求的情况下将哪些块保留在缓存中。然而，在这里，让我们考虑这个问题的离线版本，其中计算机提前知道整个 n 个请求序列和缓存大小 k，目标是最小化缓存未命中总数。

为了解决这个离线问题，您可以使用一种名为 ***furthest-in- future*** 的贪婪策略，该策略选择逐出缓存中请求序列中下一次访问距离最远的块。直观地说，这种策略是有道理的：如果你暂时不需要某个东西，为什么要保留它？我们将通过展示离线缓存问题表现出最佳子结构并且距离最远具有贪婪选择属性来证明距离最远的策略确实是最佳的。

现在，您可能会想，由于计算机通常无法提前知道请求的顺序，因此研究离线问题毫无意义。实际上，这是有意义的。在某些情况下，您确实可以提前知道请求的顺序。例如，如果您将主内存视为缓存，将整个数据集视为驻留在磁盘（或固态驱动器）上，则存在提前规划整个读写集的算法。此外，我们可以使用最佳算法产生的缓存未命中数作为比较在线算法性能的基准。我们将在第 27.3 节中这样做。

离线缓存甚至可以模拟现实世界的问题。例如，考虑这样一个场景：您事先知道已知位置的 n 个事件的固定时间表。事件可能在某个位置发生多次，不一定是连续发生的。您正在管理一组 k 个代理，您需要确保在事件发生时每个位置都有一个代理，并且您希望尽量减少代理必须移动的次数。在这里，代理就像块，事件就像请求，移动代理类似于缓存未命中。

离线缓存的最优子结构

为了表明离线问题具有最优子结构，我们将子问题 .C; i / 定义为在发生对块 b_i 的请求时，使用缓存配置 C 处理对块 $b_i; b_{i+1}; : : : ; b_n$ 的请求，即 C 是满足 jC j = k 的块集的子集。子问题 .C; i / 的解决方案是一系列决策，这些决策指定在每次请求块 $b_i; b_{i+1}; : : : ; b_n$ 时要驱逐哪个块（如果有）。子问题 .C; i / 的最优解决方案最小化缓存未命中次数。

考虑子问题 .C; i / 的一个最优解 S，令 C′ 为处理解决方案 S 中对块 b_i 的请求之后缓存的内容。令 S′ 为所得子问题 .C′; i C 1 / 的 S 子解。如果对 b_i 的请求导致缓存命中，则缓存保持不变，因此 C′ D C 。如果对块 b_i 的请求导致缓存未命中，则缓存的内容会发生变化，因此 C′ ¤ C 。我们声称在任何一种情况下，S′ 都是子问题 .C′; i C 1 / 的最优解。为什么？如果 S′ 不是子问题 .C′; i C 1/，则存在另一个子问题 .C′; i C 1/ 的解决方案 S″，其缓存未命中次数比 S′ 少。将 S″ 与 S 的决策相结合，以

块 b_i 产生另一个解决方案，该解决方案比 S 产生更少的缓存未命中，这与 S 是子问题 .C; i / 的最优解决方案的假设相矛盾。

要量化递归解决方案，我们需要更多的符号。让 $R_{C,i}$ 成为处理对块 b_i 的请求后可以立即跟在配置 C 之后的所有缓存配置的集合。如果请求导致缓存命中，则缓存保持不变，因此 $R_{C,i} = \{C\}$。如果对 b_i 的请求导致缓存未命中，则有两种可能性。如果缓存未满 ($|C| < k$)，则缓存已满，唯一的选择是将 b_i 插入缓存，因此 $R_{C,i} = \{C \cup \{b_i\}\}$。如果缓存未命中时缓存已满 ($|C| = k$)，则 $R_{C,i}$ 包含 k 种潜在配置：C 中每个可能被逐出并由块 b_i 替换的候选块一个。在这种情况下，$R_{C,i} = \{C - \{x\} \cup \{b_i\} : x \in C\}$。例如，如果请求$C = \{p, q, r\}$、$k = 3$和块s，则$R_{C,i} = \{\{p, q, s\}, \{p, r, s\}, \{q, r, s\}\}$。

令 *miss*.C; i / 表示子问题 .C; i / 解决方案中缓存未命中的最小次数。以下是 *miss*.C; i / 的递归式：

$$miss(C, i) = \begin{cases} 0 & \text{if } i = n \text{ and } b_n \in C\ , \\ 1 & \text{if } i = n \text{ and } b_n \notin C\ , \\ miss(C, i+1) & \text{if } i < n \text{ and } b_i \in C\ , \\ 1 + \min\{miss(C', i+1) : C' \in R_{C,i}\} & \text{if } i < n \text{ and } b_i \notin C\ . \end{cases}$$

贪婪选择性质

为了证明最远未来策略可产生最佳解决方案，我们需要证明最佳离线缓存具有贪婪选择特性。结合最优子结构特性，贪婪选择特性将证明最远未来策略可产生最少的缓存未命中次数。

Theorem15.5 (Optimal offline caching has the greedy-choice property)
考虑一个子问题 .C; i /，此时缓存 C 包含 k 个块，因此缓存已满，并且发生缓存未命中。当请求块 b_i 时，让 $z = b_m$ 成为 C 中下次访问时间最远的块。（如果缓存中的某个块永远不会再被引用，则将任何此类块视为块 z。并为块 $z = b_m = b_{n+1}$ 添加一个虚拟请求。）然后，在请求块 b_i 时驱逐块 z 包含在子问题 .C; i / 的某个最佳解决方案中。

Proof 假设 S 是子问题 .C; i / 的最优解。如果 S 根据对块 b_i 的请求驱逐块 z，那么我们就完成了，因为我们已经证明某些最优解包括驱逐 z。

因此现在假设最优解决方案 S 在请求块 b_i 时驱逐了其他块 x。我们将为子问题 .C; i / 构建另一个解决方案 S′，在

对 b $_i$ 的请求，驱逐块 z 而不是 x，并且不会导致比 S 更多的缓存未命中，因此 S′ 也是最优的。由于不同的解决方案可能会产生不同的缓存配置，因此用 C $_{S,j}$ 表示在请求某个块 b $_j$ 之前解决方案 S 下的缓存配置，解决方案 S′ 和 C $_{S',j}$ 也是如此。我们将展示如何构造具有以下属性的 S′：

1. 对于 j D i C 1; : : : ; m，令 D $_j$ D C $_{S,j}$ \ C $_{S',j}$。然后，jD $_j$j ≥ k 1，这样缓存配置 C $_{S,j}$ 和 C $_{S',j}$ 最多相差一个块。如果它们不同，则对于某个块 y ¤ z，C $_{S,j}$ D D $_j$ [fg 和 C $_{S',j}$ D D $_j$ [fy g。2. 对于块 b $_i$ ；: : : ; b $_{m-1}$ 的每个请求，如果解决方案 S 具有缓存命中，则解决方案 S′ 也具有缓存命中。

3. 对于所有j>m，缓存配置C$_{S,j}$和C$_{S',j}$相同。
4. 在对块 b $_i$ ；：：：；b $_m$ 的请求序列中，解决方案 S′ 产生的缓存未命中次数最多为解决方案 S 产生的缓存未命中次数。

我们将通过归纳法证明这些属性对于每个请求都成立。1. 我们对 j 进行归纳，对于 j D i C 1; : : : ; m。对于基本情况，初始缓存 C $_{S,i}$ 和 C $_{S',i}$ 是相同的。在请求块 b $_i$ 时，解决方案 S 驱逐 x，解决方案 S′ 驱逐 z。因此，缓存配置 C $_{S,i+1}$ 和 C $_{S',i+1}$ 仅相差一个块，C $_{S,i+1}$ D D $_{i+1}$ [fg，C $_{S',i+1}$ D D $_{i+1}$ [fx g，和 x ¤ z。归纳步骤定义了解决方案 S′ 在请求 i C 1 j m 1 的块 b $_j$ 时的行为。归纳假设是当请求 b $_j$ 时，属性 1 成立。因为 zD b $_m$ 是 C $_{S,i}$ 中下一个引用在最远未来的块，所以我们知道 b $_j$ ¤ z。我们考虑几种情况：• 如果 C $_{S,j}$ D C $_{S',j}$ (使得 jD $_j$j D k)，则解决方案 S′ 在请求 b $_j$ 时做出的决定与 S 相同，因此 C $_{S,j+1}$ D C $_{S',j+1}$。• 如果 jD $_j$j D k 1 且 b $_j$ 2 D $_j$，则两个缓存均已包含块 b $_j$，并且解决方案 S 和 S′ 均有缓存命中。因此，C $_{S,j+1}$ D C $_{S,j}$ 和 C $_{S',j+1}$ D C $_{S',j}$。• 如果 jD $_j$j D k 1 且 b $_j$ ··· D $_j$，则因为 C $_{S,j}$ D D $_j$ [fg 和 b $_j$ ¤ z，解决方案 S 发生缓存未命中。它逐出块 z 或某个块 w 2 D $_j$。◦ 如果解决方案 S 逐出块 z，则 C $_{S,j+1}$ D D $_j$ [fb $_j$g。根据 b $_j$ D y 是否成立，有两种情况：· 如果 b $_j$ D y，则解决方案 S′ 缓存命中，因此 C $_{S',j+1}$ D C $_{S',j}$ D D $_j$ [fb $_j$g。因此，C $_{S,j+1}$ D C $_{S',j+1}$。· 如果 b $_j$ ¤ y，则解决方案 S′ 缓存未命中。它逐出块 y，因此 C $_{S',j+1}$ D D $_j$ [fb $_j$g，再次 C $_{S,j+1}$ D C $_{S',j+1}$。

B 如果解决方案 S 驱逐了某个块 w 2 D $_j$，则 C $_{S,j+1}$ D .D $_j$ fwg/ [fb $_j$; g。同样，有两种情况，取决于 b $_j$ D y 是否：· 如果 b $_j$ D y，则解决方案 S′ 有一个缓存命中，因此 C $_{S',j+1}$ D C $_{S',j}$ D D $_j$ [fb $_j$g。由于 w 2 D $_j$ 且 w 未被解决方案 S′ 驱逐，我们有 w 2 C $_{S',j+1}$。因此，w ··· D $_{j+1}$ 和 b $_j$ 2 D $_{j+1}$，因此 D $_{j+1}$ D .D $_j$ fwg/ [fb $_j$g。因此，C $_{S,j+1}$ D D $_{j+1}$ [fg，C $_{S',j+1}$ D D $_{j+1}$ [fwg，并且因为 w ¤ z. 所以当请求块 b $_{j+1}$ 时，属性 1 成立。（换句话说，块 w 替换属性 1 中的块 y。）· 如果 b $_j$ ¤ y，则解决方案 S′ 发生缓存未命中。它逐出块 w，因此 C $_{S',j+1}$ D .D $_j$ fwg/ [fb $_j$; y g。因此，我们有 D $_{j+1}$ D .D $_j$ fwg/ [fb $_j$g 因此 C $_{S,j+1}$ D D $_{j+1}$ [fg 和 C $_{S',j+1}$ D D $_{j+1}$ [fy g。

2. 在上面关于保持属性 1 的讨论中，解决方案 S 可能仅在前两种情况下具有缓存命中，并且当且仅当 S 确实具有缓存命中时，解决方案 S′ 才在这些情况下具有缓存命中。

3. 如果 C $_{S,m}$ D C $_{S',m}$，则解决方案 S′ 在请求块 zD b $_m$ 时做出的决定与 S 相同，因此 C $_{S,m+1}$ D C $_{S',m+1}$。如果 C $_{S,m}$ ¤ C $_{S',m}$，则根据属性 1，C $_{S,m}$ D D $_m$ [fg 和 C $_{S',m}$ D D $_m$ [fy g，其中 y ¤ z. 在这种情况下，解决方案 S 有一个缓存命中，因此 C $_{S,m+1}$ D C $_{S,m}$ D D $_m$ [fg。解决方案 S′ 逐出块 y 并引入块 z. 因此 C $_{S',m+1}$ D D $_m$ [fg D C $_{S,m+1}$。因此，无论 C $_{S,m}$ D C $_{S',m}$ 是否成立，我们都有 C $_{S,m+1}$ D C $_{S',m+1}$，并且从对块 b $_{m+1}$ 的请求开始，解决方案 S′ 只需做出与 S 相同的决策。

4. 根据属性 2，在请求块 b $_i$; : : : ; b $_{m-1}$ 时，只要解决方案 S 有缓存命中，S′ 也会有缓存命中。只剩下对块 b $_m$ D z的请求需要考虑。如果 S 在请求 b $_m$ 时发生缓存未命中，那么无论 S′ 是缓存命中还是缓存未命中，我们都已完成：S′ 最多有与 S 相同数量的缓存未命中。

因此现在假设 S 有一个缓存命中，而 S′ 在请求 b $_m$ 时有一个缓存未命中。我们将证明至少存在一个对块 b $_{i+1}$; : : : ; b $_{m-1}$ 的请求，该请求导致 S 的缓存未命中和 S′ 的缓存命中，从而补偿了请求块 b $_m$ 时发生的缓存命中。证明是通过矛盾来实现的。假设对块 b $_{i+1}$; : : : ; b $_{m-1}$ 的任何请求都不会导致 S 的缓存未命中和 S′ 的缓存命中。

我们首先观察到，一旦缓存 C $_{S,j}$ 和 C $_{S'j}$ 对于某个 j > i 相等，它们此后将保持相等。还观察到，如果 b $_m$ 2 C $_{S,m}$ 和 b $_m$ ··· C $_{S',m}$，则 C $_{S,m}$ ¤ C $_{S',m}$。因此，解决方案 S 不能在请求块 b $_i$; : : : ; b $_{m-1}$ 时驱逐块 z. 因为如果这样做，那么这两个

缓存配置将相等。剩下的可能性是，对于每个请求，对于某个块 y z 我们有 C $_{S,j}$ D D $_j$ [fg，C $_{S',j}$ D D $_j$ [fy g，并且解决方案 S 驱逐了某个块 w 2 D $_j$。此外，由于这些请求均未导致 S 的缓存未命中以及 S′ 的缓存命中，因此 b $_j$ D y 的情况从未发生。也就是说，对于块 b $_{i+1}$ ；：：：；b $_{m-1}$ 的每个请求，请求的块 b $_j$ 永远不会是块 y 2 C $_{S',j}$ C $_{S,j}$。在这些情况下，在处理请求之后，我们有 C $_{S',j+1}$ D D $_{j+1}$ [fy g：两个缓存之间的差异没有变化。现在，让我们回到对块 b $_i$ 的请求，之后我们有 C $_{S',i+1}$ D D $_{i+1}$ [fx g。因为直到请求块 b $_m$ 之前的每个后续请求都没有改变缓存之间的差异，所以我们有 C $_{S',j}$ D D $_j$ [fx g for j D i C 1; : : : ; m。

根据定义，块 x D b $_m$ 在块 x 之后被请求。这意味着块 b $_{i+1}$ ；：：：；b $_{m-1}$ 中至少有一个是块 x。但是对于 j D i C 1；：：：；m，我们有 x 2 C $_{S',j}$ 和 x ⋯ C $_{S,j}$，因此这些请求中至少有一个对 S′ 有缓存命中，对 S 有缓存未命中，这是一个矛盾。我们得出结论，如果解决方案 S 有缓存命中，而解决方案 S′ 在请求块 b $_m$ 时有缓存未命中，则某个较早的请求会产生相反的结果，因此解决方案 S′ 产生的缓存未命中不会比解决方案 S 多。由于假设 S 是最优的，因此 S′ 也是最优的。 ■

除最优子结构性质外，定理 15.5 还告诉我们，最远的未来策略可产生最少的缓存未命中次数。

练习

15.4-1

为使用最远未来策略的缓存管理器编写伪代码。它应将缓存中的块集 C、缓存可容纳的块数 k、请求的块序列 b $_1$ ；b $_2$ ；：：：；b $_n$ 以及所请求块 b $_i$ 序列中的索引 i 作为输入。对于每个请求，它应打印出是发生缓存命中还是缓存未命中，对于每个缓存未命中，它还应打印出哪个块（如果有）被逐出。

15.4-2

真正的缓存管理器不知道未来的请求，因此它们经常使用过去的情况来决定要驱逐哪个块。***least-recently-used*** 或 ***LRU*** 策略会驱逐当前缓存中所有块中最近请求最少的块。（您可以将 LRU 视为“最远过去。”）给出一个请求序列示例，其中 LRU 策略不是最佳的，通过显示它比最远未来策略在相同请求序列中引起的缓存未命中次数更多。

15.4-3

Croesus 教授建议，在定理 15.5 的证明中，属性 1 中的最后一条子句可以改为 C $_{S',j}$ D D $_j$ [fx g，或者，等效地，要求属性 1 中给出的块 y 始终是解决方案 S 在请求块 b $_i$ 时驱逐的块 x。请说明证明在何处因这一要求而失效。

15.4-4

本节假设每次请求一个块时，最多只有一个块被放入缓存中。但是，您可以想象一种策略，其中多个块可以在单个请求时进入缓存。请证明，对于允许多个块在每次请求时进入缓存的每个解决方案，都有另一个解决方案在每次请求时只引入一个块，并且至少同样好。

问题

15-1Coin changing

考虑使用最少数量的硬币找零 n 分的问题。假设每枚硬币的价值都是一个整数。

a.描述一个贪婪算法，用于找回由 25 美分、10 美分、5 美分和 1 美分组成的硬币。证明你的算法能得出最优解。***b.*** 假设可用的硬币的面值为 c 的幂：面值为 c ^{0} ；c ^{1} ；：：：；c ^{k} ，其中 c > 为某些整数 1 和 k 1。证明贪婪算法总能得出最优解。

c.给出一组硬币面额，贪婪算法无法得出最优解。你的集合应该包含一分钱，这样 n 的每个值都有一个解。

d.给出一个 O.nk/-time 算法，使用最少数量的硬币对任意一组 k 种不同面额的硬币进行找零，假设其中一枚硬币是一分钱。

15-2Scheduling to minimize average completion time

给定一个任务集合 S D fa $_1$; a $_2$; : : : ; a $_n$ g，其中任务 a $_i$ 需要 p $_i$ 个处理时间单位才能完成。令 C $_i$ 为任务 a $_i$ 的 ***completion time***，即任务 a $_i$ 完成处理的时间。您的目标是最小化平均完成时间，即最小化 .1/n/ P $_{i=1}^{n}$ C $_i$ 。例如，假设有两个任务 a $_1$ 和 a $_2$ ，其 p $_1$ D 为 3，p $_2$ D 为 5，并考虑时间表

其中，$_2$ 首先运行，然后是 $_1$。然后我们有 C $_2$ D 5、C $_1$ D 8，平均完成时间为 .5 C 8//2 D 6:5。但是，如果任务 a $_1$ 首先运行，那么我们有 C $_1$ D 3、C $_2$ D 8，平均完成时间为 .3 C 8//2 D 5:5。

a.给出一个算法来安排任务，以最小化平均完成时间。每个任务必须非抢占式运行，也就是说，一旦任务 a $_i$ 开始，它必须连续运行 p $_i$ 个单位时间，直到完成。证明你的算法最小化平均完成时间，并分析算法的运行时间。

b.现在假设任务并非全部同时可用。也就是说，每个任务在其 ***release time*** b $_i$ 之前都无法启动。还假设任务可能是 ***preempted***，以便可以暂停任务并在稍后重新启动。例如，处理时间 p $_i$ D 6 和释放时间 b $_i$ D 1 的任务 a $_i$ 可能在时间 1 开始运行并在时间 4 被抢占。然后它可能在时间 10 恢复，但在时间 11 被抢占，并且它可能最终在时间 13 恢复并在时间 15 完成。任务 a $_i$ 总共运行了 6 个时间单位，但其运行时间被分为三部分。给出一个算法来调度任务，以最小化这种新场景中的平均完成时间。证明你的算法最小化了平均完成时间，并分析你的算法的运行时间。

章节注释

关于贪婪算法的更多资料可以在 Lawler [276] 和 Papadimitriou 和 Steiglitz [353] 中找到。贪婪算法首次出现在组合优化文献中，是在 1971 年 Edmonds 的一篇文章中 [131]。

活动选择问题贪婪算法的正确性证明基于 Gavril [179] 的证明。

霍夫曼编码发明于 1952 年[233]。Lelewer 和 Hirschberg [294] 于 1987 年综述了已知的数据压缩技术。

最远未来策略由 Belady [41] 提出，他建议将其用于虚拟内存系统。Lee 等人 [284] 和 Van Roy [443] 的文章提供了最远未来策略是最优的替代证明。

16 Amortized Analysis

假设您加入了 Buff 健身房。Buff 收取每月 60 美元的会员费，外加每次使用健身房的 3 美元。由于您很守纪律，所以您在 11 月份每天都会去 Buff 健身房。除了 11 月份的 60 美元月费外，您当月还需支付 3 美元（30 美元，90 美元）。虽然您可以将费用视为 60 美元的月费和另外 90 美元的日费，但您可以换个角度考虑。总共，您在 30 天内支付 150 美元，平均每天 5 美元。以这种方式查看费用时，您需要将月费分摊到每月的 30 天内，每天分摊 2 美元。

在分析运行时间时，您可以做同样的事情。在 ***amortized analysis*** 中，您将执行一系列数据结构操作所需的时间平均化到所有执行的操作中。通过摊销分析，您可以表明，如果对一系列操作进行平均，那么即使序列中的单个操作可能很昂贵，操作的平均成本也会很小。摊销分析与平均情况分析的不同之处在于不涉及概率。摊销分析保证 *average performance of each operation in the worst case*。

本章的前三节介绍了摊销分析中最常用的三种技术。第 16.1 节从聚合分析开始，在该分析中，您可以确定 n 个操作序列的总成本的上限 T .n/。然后，每个操作的平均成本为 T .n//n。您将平均成本作为每个操作的摊销成本，以便所有操作都具有相同的摊销成本。

16.2 节介绍了会计方法，通过该方法可以确定每个操作的摊销成本。当存在多种类型的操作时，每种类型的操作可能具有不同的摊销成本。会计方法在序列的早期对某些操作收取过高的费用，将过高的费用存储为 “pre-

支付信用额度”用于数据结构中的特定对象。在后续的序列中，信用额度用于支付收费低于实际成本的操作。

16.3 节讨论了势方法，它类似于会计方法，可以确定每项操作的摊销成本，并可能在早期对操作收取过高的费用，以补偿以后收取的不足。势方法将信用作为整个数据结构的 “势能” 来维护，而不是将信用与数据结构中的单个对象相关联。

在本章中，我们将使用两个示例来分别检查这三种方法。一个是带有附加操作 MULTIPOP 的堆栈，该操作可一次弹出多个对象。另一个是二进制计数器，它通过单个操作 I NCREMENT 从 0 开始递增计数。

阅读本章时，请记住摊销分析期间分配的费用仅用于分析目的。它们不需要也不应该出现在代码中。例如，如果您在使用会计方法时为对象 x 分配信用，则无需在代码中为某些属性（例如 x: *credit* ）分配适当的金额。

当你进行摊销分析时，你经常会深入了解特定的数据结构，这种洞察可以帮助你优化设计。例如，第 16.4 节将使用势方法来分析动态扩展和收缩表。

16.1 总体分析

在 ***aggregate analysis*** 中，您表明对于所有 n，n 个操作序列总共需要 T .n/ *worst-case* 时间。在最坏情况下，每个操作的平均成本或 ***amortized cost*** 因此为 T .n//n。此摊销成本适用于每个操作，即使序列中有多种类型的操作也是如此。我们将在本章中研究的另外两种方法，即核算方法和潜在方法，可能会为不同类型的操作分配不同的摊销成本。

堆栈操作

作为聚合分析的第一个示例，让我们分析已添加新操作的堆栈。第 10.1.3 节介绍了两个基本堆栈操作，每个操作都需要 O.1/ 时间：

PUSH.S; x/ 将对象 x 推送到堆栈 S 上。

POP.S / 弹出堆栈 S 的顶部并返回弹出的对象。在空堆栈上调用 POP 会产生错误。

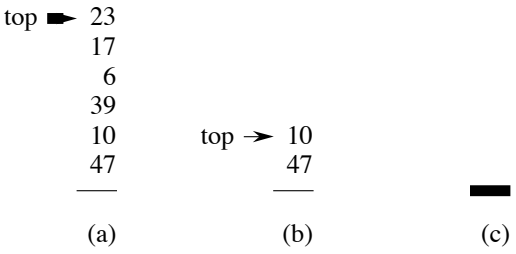


图 16.1 MULTIPOP 对堆栈 S 的操作，最初如图 (a) 所示。顶部 4 个对象由 MULTIPOP.S; 4/ 弹出，其结果如图 (b) 所示。下一个操作是 MULTIPOP.S; 7/，它将清空堆栈 4（如图 (c) 所示），因为剩余的对象少于 7 个。

由于每个操作都需要 O.1/ 的时间，我们假设每个操作的成本为 1。因此，n 个 PUSH 和 POP 操作序列的总成本为 n，n 个操作的实际运行时间为 ‘ .n/ ’ 。现在让我们添加堆栈操作 MULTIPOP .S; k/，该操作移除堆栈 S 的顶部 k 个对象，如果堆栈包含的对象少于 k 个，则弹出整个堆栈。当然，该过程假定 k 为正数，否则，MULTIPOP 操作将保持堆栈不变。在 M ULTIPOP 的伪代码中，如果堆栈上当前没有对象，则操作 STAC K-EMPTY 返回 TRUE，否则返回 FALSE。图 16.1 显示了 MULTIPOP 的一个示例。

多点弹出 .S; k/

1 而非 STAC K-EMPTY .S / 且 k > 0 2 PO
P.S / 3 k D k 1

在包含 s 个对象的堆栈上，MULTIPOP .S; k/ 的运行时间是多少？实际运行时间与实际执行的 POP 操作数成线性关系，因此我们可以从 PUSH 和 PO P 各为 1 的抽象成本的角度来分析 MULTIPOP。while 循环的迭代次数是从堆栈中弹出的对象数 min fs; kg。循环的每次迭代都会在第 2 行调用一次 P O P。因此，MULTIPOP 的总成本为 min fs; kg，实际运行时间是此成本的线性函数。

现在让我们分析一个初始为空的堆栈上的 n 个 PUSH、POP 和 MULTIPOP 操作序列。序列中 MULTIPOP 操作的最坏情况成本为 O.n/，因为堆栈大小最多为 n。因此，任何堆栈操作的最坏情况时间都是 O.n/，因此 n 个操作序列的成本为 $O.n^2/$，因为该序列最多包含 n 个 MULTIPOP 操作，每个操作的成本为 O.n/。

虽然这个分析是正确的，但是通过单独考虑每个操作的最坏情况成本而得出的 O.n ^{2} / 结果并不严密。

是的，单个 MULTIPOP 可能很昂贵，但综合分析表明，在最初为空的堆栈上执行任何 n 次 PUSH、POP 和 MULTIPOP 操作的序列，其成本上限为 O.n/。为什么？除非首先推送对象，否则无法将其从堆栈中弹出。因此，在非空堆栈上调用 POP 的次数（包括在 M ULTIPOP 内的调用）最多为 PUSH 操作的次数，最多为 n 次。对于任何 n 值，任何 n 次 PUSH、POP 和 MULT IPOP 操作的序列总共需要 O.n/ 次。对 n 个操作取平均值可得出每个操作的平均成本为 O.n//n D O.1/。综合分析将每个操作的摊销成本指定为平均成本。因此，在此示例中，所有三个堆栈操作的摊销成本均为 O.1/。

回顾一下：尽管堆栈操作的平均成本（因此也是运行时间）为 O.1/，但分析并不依赖概率推理。相反，分析得出了 n 个操作序列的 *worst-case* 界限为 O.n/。将此总成本除以 n 可得出每个操作的平均成本4（即摊销成本4）为 O. 1/。

增加二进制计数器

作为聚合分析的另一个例子，考虑实现一个从 0 开始向上计数的 k 位二进制计数器的问题。该计数器由一个位数组 AŒ0 W k 1 表示。存储在计数器中的二进制数 x 的最低位在 AŒ0 中，最高位在 AŒk 1 中，因此 x D P $_{i=0}^{k-1}$ AŒi 2 ^{i}。最初，x D 0，因此对于 i D 0; 1; : : : ; k 1，AŒi D 0。要将 1（模 2 ^{k}）添加到计数器中的值，请调用 I NCREMENT 过程。

```
我增加.A; k/
1 i D 0 2 当 i < k 且 AŒi
==1 3 AŒi D 0 4 i D i C 1
5 如果 i < k 6 AŒi D 1
```

图 16.2 显示了当 I NCREMENT 被调用 16 次时二进制计数器会发生什么情况，从初始值 0 开始，到值 16 结束。第 2 到第 3 行中的 while 循环每次迭代都会将 1 添加到位置 i 中。如果 AŒi = D 1，则加 1 会使位置 i 中的位变为 0，并产生一个进位 1，将其添加到

Counter value	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

图 16.2 一个 8 位二进制计数器，其值通过一系列 16 次 I NCREMENT 操作从 0 变为 16。需要移位才能达到下一个值的位以蓝色阴影表示。移位的运行成本显示在右侧。总成本始终小于 I NCREMENT 操作总数的两倍。

在循环的下一次迭代中，将位置 i C 1 置为 0。否则，循环结束，然后，如果 i < k，AŒi 必须为 0，因此第 6 行将 1 添加到位置 i，将 0 置为 1。如果循环以 i D k 结束，则 I NCREMENT 调用将所有 k 位从 1 置为 0。每个 I NCREMENT 操作的成本与置为 0 的位数成线性关系。

与堆栈示例一样，粗略分析得出的界限是正确的，但不严格。在最坏的情况下，一次 I NCREMENT 执行需要 ，.k/ 时间，其中数组 A 中的所有位都是 1。因此，在最初为零的计数器上执行 n 次 I NCREMENT 操作序列在最坏的情况下需要 O.nk/ 时间。

尽管一次 I NCREMENT 调用可能弹出所有 k 位，但并不是每次调用时所有位都会弹出。（请注意与 MULTIPOP 的相似性，一次调用可能弹出许多对象，但并不是每次调用都会弹出许多对象。）如图 16.2 所示，每次调用 I NCREMENT 时，AŒ0 都会弹出。下一位 AŒ1 每隔一次才会弹出：对最初为零的计数器进行 n 次 I NCREMENT 操作会导致 AŒ1 弹出 bn/2c 次。类似地，位 AŒ2 每四次才会弹出一次，或者在 n 次 I NCREMENT 操作序列中弹出 bn/4c 次。通常，对于 i D 0; 1; : : : ; k 1，位 AŒi üips bn /2 ^{i} c 次，在初始为零的计数器上执行 n 次 I INCREMENT 操作。对于 i ¹ k，位 AŒi 不存在，因此它不能 üip。总数

因此序列中的 üips 为

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n ,$$

根据第 1142 页的公式 (A.7)。因此，在最初为零的计数器上执行一系列 n 次 I NCREMENT 操作在最坏情况下需要 O.n/ 时间。每次操作的平均成本（因此每次操作的摊销成本）为 O.n//n D O.1/。

练习

16.1-1
如果堆栈操作集包括 MULTIPUSH 操作（将 k 个项目推送到堆栈上），那么堆栈操作的摊销成本的 O.1/ 界限是否仍然成立？

16.1-2
说明如果在 k 位计数器示例中包含一个 DECREMENT 操作，则 n 次操作可能耗费多达 ，.nk/ 个时间。

16.1-3
使用聚合分析来确定数据结构上一系列 n 个操作的每个操作的摊销成本，其中，如果 i 是 2 的精确幂，则第 i 个操作的成本为 i，否则为 1。

16.2 会计方法

在摊销分析的 ***accounting method*** 中，您可以为不同的操作分配不同的费用，某些操作的收费高于或低于其实际成本。您向操作收取的金额是其 ***amortized cost***。当操作的摊销成本超过其实际成本时，您可以将差额分配给数据结构中的特定对象作为 ***credit***。信用可以帮助支付摊销成本低于实际成本的后续操作。因此，您可以将操作的摊销成本视为在其实际成本和存入或用完的信用之间分配。不同的操作可能有不同的摊销成本。此方法不同于聚合分析，在聚合分析中，所有操作都具有相同的摊销成本。

你必须谨慎选择操作的摊销成本。如果你想使用摊销成本来表明在最坏情况下，每个操作的平均成本是

很小，您必须确保操作序列的总摊销成本为该序列的总实际成本提供了上限。此外，与聚合分析一样，上限必须适用于所有操作序列。让我们用 c $_i$ 表示第 i 个操作的实际成本，用 y c $_i$ 表示第 i 个操作的摊销成本。然后您需要有

$$\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i \tag{16.1}$$

对于所有 n 个操作序列。数据结构中存储的总信用是总摊销成本与总实际成本之间的差额，或 P $_{i=1}^{n}$ y c $_i$ P $_{i=1}^{n}$ c $_i$。根据不等式 (16.1)，与数据结构相关的总信用必须始终为非负数。如果你允许总信用变为负数（早期操作收费过低并承诺以后偿还账户的结果），那么当时发生的总摊销成本将低于发生的总实际成本。在这种情况下，对于截至当时的操作序列，总摊销成本将不是总实际成本的上限。因此，你必须注意数据结构中的总信用永远不要变为负数。

堆栈操作

为了说明摊销分析的会计方法，我们回到堆栈示例。回想一下，操作的实际成本是
PUSH 1,POP 1,MULTIPOP 分
钟 fs; kg,

其中 k 是提供给 MULTIPOP 的参数，s 是调用时的堆栈大小。让我们分配以下摊销成本：
推送 2，弹出 0，多
弹出 0。

MULTIPOP 的摊销成本为常数（0），而实际成本为变量，因此所有三个摊销成本均为常数。一般而言，所考虑操作的摊销成本可能彼此不同，甚至可能渐近不同。

现在让我们看看如何通过收取摊销成本来支付任何堆栈操作序列的费用。让 1 美元代表每个成本单位。首先，堆栈是空的。回想一下第 10.1.3 节中堆栈数据结构与自助餐厅中的一叠盘子之间的类比。将盘子推入堆栈后，使用 1 美元支付

实际推车成本，留下 1 美元信用额（从收取的 2 美元中）。将 1 美元信用额放在车牌顶部。在任何时间点，堆中的每块车牌上都有 1 美元信用额。

盘子上存储的 1 美元用于预付将盘子从堆栈中弹出的成本。POP 操作不产生任何费用：通过从盘子中取出 1 美元信用来支付弹出盘子的实际成本。因此，通过对 PUSH 操作收取更多费用，我们可以将 POP 操作视为免费。

此外，MULTIPOP 操作也不会产生任何费用，因为它只是重复的 POP 操作，每个操作都是免费的。如果 MULTIPOP 操作弹出 k 个盘子，则实际成本由 k 个盘子上存储的 k 美元支付。由于堆栈上的每个盘子上都有 1 美元的信用额度，并且堆栈中的盘子数量始终为非负数，因此信用额度始终为非负数。因此，对于 n 个 PUSH、POP 和 MULTIPOP 操作的 *any* 序列，总摊销成本是总实际成本的上限。由于总摊销成本为 O.n/，因此总实际成本也是如此。

增加二进制计数器

作为会计方法的另一个例证，让我们分析从 0 开始的二进制计数器上的 I N CREMENT 操作。回想一下，此操作的运行时间与被移除的位数成正比，这在本例中是成本。同样，我们将使用 $1 来表示每个成本单位（在本例中为移除一位）。

对于摊销分析，将 0 位设置为 1 的摊销成本为 2 美元。当某个位设置为 1 时，2 美元中的 1 美元用于实际设置该位。第二个 1 美元作为信用额度驻留在该位上，以后如果该位被重置为 0 时使用。在任何时间点，计数器中的每个 1 位上都有 1 美元的信用额度，因此将某个位重置为 0 可以视为不花费任何成本，并且该位上的 1 美元预付了重置费用。

以下是如何确定 I NCREMENT 的摊销成本。在 while 循环中将位重置为 0 的成本由重置位的美元支付。I NCREMENT 过程最多将一位设置为 1（在第 6 行），因此 I NCREMENT 操作的摊销成本最多为 2 美元。计数器中的 1 位数量永远不会变为负数，因此信用额度始终保持非负值。因此，对于 n 个 I NCREMENT 操作，总摊销成本为 O.n/，这限制了总实际成本。

练习

16.2-1

对大小不超过 k 的堆栈执行一系列 PUSH 和 POP 操作。每执行 k 次操作后，整个堆栈的副本都会自动生成。

出于备份目的。通过为各种堆栈操作分配适当的摊销成本，证明 n 个堆栈操作（包括复制堆栈）的成本为 O.n/。

16.2-2
使用会计分析方法重做练习 16.1-3。

16.2-3
您不仅希望增加计数器，还希望将其重置为 0（即，使其中的所有位都为 0）。计算检查或修改位的时间，作为“ .1/ ”，说明如何将计数器实现为位数组，以便任何 n 个 I NCREMENT 和 RESET 操作序列在最初为零的计数器上都需要 O.n/ 时间。（*Hint:* 保持指向高位 1 的指针。）

16.3 势方法

摊销分析的 ***potential method*** 不是将预付工作表示为存储在数据结构中的特定对象的信用，而是将预付工作表示为“潜在能量，”或只是“ 潜力，”，可以释放以支付未来的操作。潜力适用于整个数据结构，而不是数据结构内的特定对象。

势方法的工作原理如下。从初始数据结构 D $_0$ 开始，发生一系列 n 个操作。对于每个 i D 1; 2; : : : ; n，让 c $_i$ 为第 i 个操作的实际成本，D $_i$ 为将第 i 个操作应用于数据结构 D $_{i-1}$ 后产生的数据结构。***potential function*** Φ将每个数据结构 D $_i$ 映射到实数 ΦD $_i$ /，即与 D $_i$ 关联的 ***potential***。关于势函数 Φ的第 i 个操作的 ***amortized cost*** y c $_i$ 定义为

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \tag{16.2}$$

因此，每项操作的摊销成本等于其实际成本加上由于该操作而产生的潜在变化。根据公式 (16.2)，n 项操作的总摊销成本为

$$\begin{aligned} \sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned} \tag{16.3}$$

第二个方程根据第 1143 页的方程 (A.12) 得出，因为 $\Phi(D_i)$ 项是望远镜。

如果你能定义一个势函数 Φ 使得 $\Phi(D_n)$ 和 $\Phi(D_0)$，那么总摊销成本 $\sum_{i=1}^{n} \hat{c}_i$ 给出了总实际成本 $\sum_{i=1}^{n} c_i$ 的上限。实际上，你并不总是知道可能执行多少个操作。因此，如果你要求对于所有 i 都有 $\Phi(D_i)$ 和 $\Phi(D_0)$，那么你就可以保证，就像在会计方法中一样，你已经提前付款了。通常最简单的方法是定义 $\Phi(D_0)$ 为 0，然后证明对于所有 i 都有 $\Phi(D_i)$ 和 0。（参见练习 16.3-1，了解处理 $\Phi(D_0) \neq 0$ 的情况的简单方法。）

直观地看，如果第i个操作的电位差$\Phi(D_i) - \Phi(D_{i-1})$为正，则摊销成本$\hat{c}_i$表示对第i个操作的充电过高，数据结构的电位会增加。如果电位差为负，则摊销成本表示对第i个操作的充电不足，电位的减少将支付操作的实际成本。

方程 (16.2) 和 (16.3) 定义的摊销成本取决于势函数 Φ 的选择。不同的势函数可能产生不同的摊销成本，但仍然会限制实际成本。在选择势函数时，您经常会发现需要权衡利弊。要使用的最佳势函数取决于所需的时间界限。

堆栈操作

为了说明势能法，我们再次回到堆栈操作 PUSH、POP 和 MULTIPOP 的示例。我们将堆栈上的势函数 Φ 定义为堆栈中的对象数。空的初始堆栈 D_0 的势能为 $\Phi(D_0) = 0$。由于堆栈中的对象数永远不会为负，因此第 i 次操作后得到的堆栈 D_i 具有非负势能，因此

$$\begin{aligned} \Phi(D_i) &\geq 0 \\ &= \Phi(D_0) . \end{aligned}$$

因此，相对于 Φ 的 n 个操作的总摊销成本代表了实际成本的上限。

现在让我们计算各种堆栈操作的摊销成本。如果包含 s 个对象的堆栈上的第 i 个操作是 PUSH 操作，则潜在差异为

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 . \end{aligned}$$

根据公式（16.2），此 PUSH 操作的摊销成本为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 .\end{aligned}$$

假设对 s 个对象堆栈的第 i 个操作是 MULTIPOP .S; k/，这会导致 k ′ D min f s; kg 个对象从堆栈中弹出。该操作的实际成本为 k ′，潜在差异为

$$\Phi(D_i) - \Phi(D_{i-1}) = -k' .$$

因此，MULTIPOP 操作的摊销成本为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 .\end{aligned}$$

类似地，普通 POP 操作的摊销成本为 0。

三个操作中每个操作的摊销成本为 O.1/，因此 n 个操作序列的总摊销成本为 O.n/。由于 Φ i / 和 Φ 0 /，n 个操作的总摊销成本是总实际成本的上限。因此，n 个操作的最坏情况成本为 O.n/。

增加二进制计数器

作为势能法的另一个例子，我们重新讨论增加一个 k 位二进制计数器。这次，第 i 次 I INCREMENT 操作之后计数器的势能被定义为第 i 次操作之后计数器中 1 的位数，我们将其表示为 b i。

以下是如何计算 I NCREMENT 操作的摊销成本。假设第 i 个 I NCREMENT 操作将 t i 位重置为 0。因此，该操作的实际成本 c i 最多为 t i C 1，因为除了重置 t i 位之外，它最多将一位设置为 1。如果 b i D 0，则第 i 个操作已将所有 k 位重置为 0，因此 b i 1 D t i D k。如果 b i > 0，则 b i D b i 1 t i C 1。在任一情况下，b i b i 1 t i C 1，潜在差为

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\le (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i .\end{aligned}$$

因此摊销成本为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le (t_i + 1) + (1 - t_i) \\ &= 2 .\end{aligned}$$

如果计数器从 0 开始，则 $\Phi(D_0) = 0$。由于对于所有 i，$\Phi(D_i) \geq 0$ 均为 0，因此 n 个 I NCREMENT 操作序列的总摊销成本是总实际成本的上限，因此 n 个 I NCREMENT 操作的最坏情况成本为 O.n/。

即使计数器不是从 0 开始，潜在方法也提供了一种简单而巧妙的方法来分析计数器。计数器以 b_0 个 1 位开始，经过 n 次 I NCREMENT 操作后，它有 b_n 个 1 位，其中 $0 \leq b_0, b_n \leq k$。将方程 (16.3) 重写为

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) .$$

由于 $\Phi(D_0) = b_0$、$\Phi(D_n) = b_n$ 和 $c_i = \hat{c}_i$ #2 对所有 $1 \leq i \leq n$ 而言，n 个 I NCREMENT 操作的实际总成本为

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 . \end{aligned}$$

具体来说，$b_0 \geq k$ 表示只要 $k = O(n)$，总实际成本就是 O.n/。换句话说，如果至少发生 $n = \Omega(k)$ I NCREMENT 操作，则总实际成本就是 O.n/，无论计数器包含的初始值是什么。

练习

16.3-1

假设有一个势函数 Φ，使得对所有的 i，都有 $\Phi(D_i) \geq \Phi(D_0)$，但 $\Phi(D_0) \geq 0$。证明存在一个势函数 Φ'，使得对所有的 i，都有 $\Phi'(D_0) \geq 0$，$\Phi'(D_i) \geq 0$，并且使用 Φ 的摊销成本与使用 Φ' 的摊销成本相同。

16.3-2

关于使用潜在的分析方法做练习 16.1-3 溶解。

16.3-3

考虑一个支持 I NSERT 和 EXTRACT-MIN 指令的普通二进制最小堆数据结构，当堆中有 n 个项目时，在最坏情况下，每个操作的执行时间为 O.lg n/。给出一个势函数 Φ，使得 I NSERT 的摊销成本为 O.lg n/，EXTRACT-MIN 的摊销成本为 O.1/，并表明您的势函数得出这些摊销时间界限。请注意，在分析中，n 是堆中当前的项目数，并且您不知道堆中可以存储的最大项目数的界限。

16.3-4
假设堆栈以 s_0 个对象开始并以 s_n 个对象结束，那么执行 n 次堆栈操作 PUSH、POP 和 MULTIPOP 的总成本是多少？

16.3-5
说明如何使用两个普通堆栈实现队列（练习 10.1-7），使得每次 ENQUEUE 和每次 DEQUEUE 操作的摊销成本为 O.1/。

16.3-6
设计一个数据结构来支持整数动态多集 S 的以下两种操作，该结构允许重复值：

I NSERT.S; x/ 将 x 插入到 S 中。

DELETE-LARGER-HALF .S / 从 S 中删除最大的 djS j ⁄2e 个元素。

解释如何实现此数据结构，以便任何 m 个 I NSERT 和 DELETE-LARGER-HALF 操作序列在 O.m/ 时间内运行。您的实现还应包括一种在 O.jS j/ 时间内输出 S 元素的方法。

16.4 动态表

当您设计使用表的应用程序时，您并不总是事先知道表将容纳多少项。您可能会为表分配空间，但后来才发现空间不够。然后，程序必须重新分配更大的表，并将原始表中的所有项复制到新的更大的表中。同样，如果已从表中删除许多项，则可能需要重新分配较小的表。本节研究动态扩展和收缩表的问题。摊销分析将显示插入和删除的摊销成本仅为 O.1/，即使操作在触发扩展或收缩时的实际成本很大。此外，您将看到如何保证动态表中未使用的空间永远不会超过总空间的恒定部分。

假设动态表支持 TABLE-INSERT 和 TABLE-DELETE 操作。TABLE-INSERT 向表中插入一个项目，该项目占用一个 ***slot***，即一个项目的空间。同样，TABLE-DELETE 从表中删除一个项目，从而释放一个位置。用于组织表的数据结构方法的细节并不重要：它可以是堆栈（第 10.1.3 节）、堆（第 6 章）、哈希表（第 11 章）或其他东西。

使用第 11.2 节中介绍的概念很方便，我们在其中分析了哈希。非空表 T 的 ***load factor*** $\alpha = T.num / T.size$ 定义为表中存储的项目数除以表的大小（插槽数）。空表（没有插槽的表）的大小为 0，其负载因子定义为 1。如果动态表的负载因子低于常数，则表中未使用的空间永远不会超过总空间量的常数分数。

我们首先分析一个只允许插入的动态表，然后分析支持插入和删除的更一般的情况。

16.4.1 表扩展

假设表的存储分配为一个槽数组。当所有槽都已使用或相当于其负载因子为 1 时，表将被填满。[1] 在某些软件环境中，尝试将项目插入已满的表中时，唯一的选择是中止并显示错误。但是，本节中的场景假设软件环境与许多现代软件环境一样，提供了可以根据请求分配和释放存储块的内存管理系统。因此，在将项目插入已满的表中时，系统可以通过分配一个比旧表具有更多槽的新表来***expand*** 该表。由于表必须始终驻留在连续的内存中，因此系统必须为更大的表分配一个新数组，然后将项目从旧表复制到新表中。

一种常见的启发式方法是分配一个新表，其槽数是旧表的两倍。如果唯一的表操作是插入，则表的负载因子始终至少为 1/2，因此浪费的空间量永远不会超过表中总空间的一半。

下一页上的 TABLE-I NSERT 过程假定 T 是一个表示表的对象。属性 T: *table* 包含指向表示表的存储块的指针，T: *num* 包含表中的项目数，T: *size* 给出表中的总槽数。最初，表是空的：T: *num* D T: *size* D 0。

这里有两种类型的插入：TABLE-INSERT 过程本身和第 6 行和第 10 行中的 ***elementary insertion*** 插入表。我们可以通过为每个基本插入分配 1 的成本来分析 TABLE-INSERT 的运行时间，以基本插入的数量为依据。在大多数计算环境中，第 2 行中分配初始表的开销是恒定的，而第 5 行和第 7 行中分配和释放存储的开销主要由传输成本决定。

---

[1] In some situations, such as an open-address hash table, it's better to consider a table to be full if its load factor equals some constant strictly less than 1. (See Exercise 16.4-2.)

TABLE-I NSERT .T; x/ 1 如果 T: *size* ==0 2 为 T: *table*
分配 1 个槽 3 T: *size* D 1 4 如果 T: *num* ==T: *size* 5 为
new-table 分配 2 个 T: *size* 个槽 6 将 T: *table* 中的所
有项目插入到 *new-table* 中 7 释放 T: *table* 8 T: *table*
D *new-table* 9 T: *size* D 2 T: *size* 10 将 x 插入 T: *table*
11 T: *num* D T: *num* C 1

第 6 行中的 ring 项。因此，TABLE-INSERT 的实际运行时间与基本插入的数量成线性关系。执行第 539 行时发生 ***expansion***。

现在，我们将使用所有三种摊销分析技术来分析在最初为空的表上执行的 n 个 TABLE-INSERT 操作序列。首先，我们需要确定第 i 个操作的实际成本 c_i。如果当前表有空间容纳新项目（或者这是第一个操作），则 c_i D 1，因为执行的唯一基本插入是第 10 行中的插入。但是，如果当前表已满，并且发生扩展，则 c_i D i：第 10 行中基本插入的成本为 1 加上第 6 行中从旧表复制到新表的项目的成本为 i 1。对于 n 个操作，操作的最坏情况成本为 O.n /，这导致 n 个操作的总运行时间的上限为 O.n^2 /。

这个界限并不严格，因为在 n 个 TABLE-INSERT 操作过程中，表很少会扩展。具体来说，只有当 i 1 是 2 的精确幂时，第 i 个操作才会导致扩展。正如聚合分析所示，操作的摊销成本实际上是 O.1/。第 i 个操作的成本为

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$$

因此，n 个 TABLE-INSERT 操作的总成本为

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \qquad \text{(by equation (A.6) on page 1142)} \\ &= 3n, \end{aligned}$$

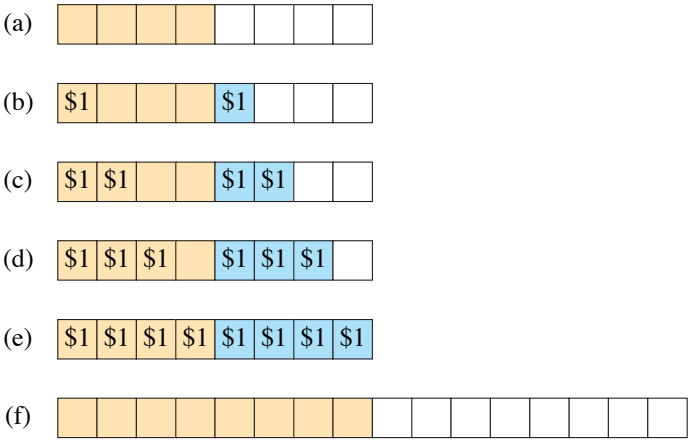


图 16.3 用记帐方法分析表扩展。每次调用 TABLE-I NSERT 都会收取 3 美元，具体如下：1 美元用于支付基本插入费用，1 美元用于插入项作为稍后重新插入的预付款，1 美元用于已经存在于表中的项，也作为重新插入的预付款。（a）扩展后的表，有 8 个槽，4 个项（棕褐色槽），没有存储信用。（b）–（e）在 4 次调用 TABLE-I NSERT 之后，表中都会多出一个项，新项上存储 1 美元，扩展后立即存在的 4 个项之一上存储 1 美元。包含这些新项的槽为蓝色。（f）下次调用 TABLE-I NSERT 时，表已满，因此再次扩展。每个项需要支付 1 美元才能重新插入。现在表格看起来与部分 (a) 中一样，没有存储信用但有 16 个槽和 8 个项目。

因为最多 n 个操作每个成本为 1，其余操作的成本形成一个几何级数。由于 n 个 TABLE-INSERT 操作的总成本限制为 3n，因此单个操作的摊销成本最多为 3。

会计方法可以直观地解释为什么 TABLE-INSERT 操作的摊销成本应为 3。你可以认为每个项目都为三个基本插入付费：将自身插入当前表、在表下次扩展时移动自身以及在表下次扩展时移动表中已有的其他项目。例如，假设扩展后表的大小立即为 m，如图 16.3 中 m D 8 所示。然后表包含 m/2 个项目，并且不包含任何信用。每次调用 TABLE-INSERT 收费 $3。立即发生的基本插入费用为 $1。另一个 $1 作为信用驻留在插入的项目上。第三个 $1 作为信用驻留在表中已有的 m/2 个项目之一上。直到另外 m/2 1 个项目插入后，表才会再次装满，因此，当表包含 m 个项目并已满时，每个项目上都有 $1 的费用，需支付扩展期间重新插入的费用。

现在，让我们看看如何使用潜在方法。我们将在 16.4.2 节中再次使用它来设计一个摊销成本为 O.1/ 的 TABLE-DELETE 操作

就像会计方法在扩展4（即，当 T: *num* D T: *size*/24）后没有立即存储信用一样，让我们将 T: *num* D T: *size*/2 时的势能定义为 0。随着基本插入的发生，势能需要增加到足以支付表下次扩展时发生的所有重新插入。当 T: *num* D T: *size* 时，表在另一次 T: *size*/2 次 TABLE-INSERT 调用后填满。在这些 T: *size*/2 次调用之后的下一次 TABLE-INSERT 调用将触发扩展，成本为 T: *size* 以重新插入所有项目。因此，在 T: *size*/2 次 TABLE-INSERT 调用过程中，势能必须从 0 增加到 T: *size*。为了实现这一增长，让我们设计一个潜在的方法，以便每次调用 TABLE-INSERT 都会增加

$$\frac{T.size}{T.size/2} = 2 ,$$

直到表格展开。你可以看到势函数

$$\Phi(T) = 2(T.num - T.size/2) \tag{16.4}$$

在表扩展后立即等于 0，此时 T: *num* D T: *size*/2，每次插入时它都会增加 2，直到表填满。一旦表填满，即 T: *num* D T: *size*，势能 Φ / 等于 T: *size*。势能的初始值为 0，由于表始终至少为半满，因此 T: *num* ≈ T: *size*/2，这意味着 Φ / 始终为非负值。因此，n 个 TABLE-INSERT 操作的摊销成本之和给出了实际成本之和的上限。

为了分析表操作的摊销成本，可以方便地从每个操作引起的电位变化的角度来思考。令 Φ 表示第 i 个操作后的电位，我们可以将方程 (16.2) 重写为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= c_i + \Delta\Phi_i ,\end{aligned}$$

其中 Φ 是由于第 i 次操作而引起的电位变化。首先，考虑第 i 次插入不会导致表扩展的情况。在这种情况下， Φ 为 2。由于实际成本 c $_i$ 为 1，因此摊销成本为

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi_i \\ &= 1 + 2 \\ &= 3 .\end{aligned}$$

现在，考虑当表在第 i 次插入期间确实扩展时可能出现的变化，因为在插入之前表是满的。令 *num* $_i$ 表示第 i 次操作后存储在表中的项目数，*size* $_i$ 表示第 i 次操作后表的总大小，因此 *size* $_{i-1}$ D *num* $_{i-1}$ D i 1

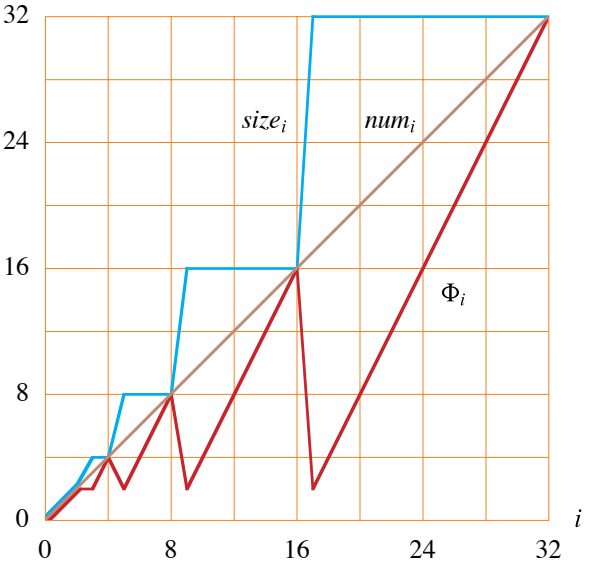


图 16.4 n 次 TABLE-INSERT 操作序列对表中项目数 num_i（棕线）、表中插槽数 $size_i$（蓝线）以及势 ΦD 2.num_i $size_i$/2/（红线）的影响，每个都是在第 i 次操作之后测量的。在扩展之前，势已累积到表中项目数，因此它可以支付将所有项目移动到新表的费用。此后，势降至 0，但在插入导致扩展的项目时，势立即增加 2。

因此 Φ_{-1} D 2.$size_{i-1}$ $size_{i-1}$/2/ D $size_{i-1}$ D i 1. 扩展之后，势能立即降至 0，然后插入新项，导致势能增加至 ΦD 2. 因此，当第 i 次插入触发扩展时，ΦD 2 .i 1/ D 3 i . 当表在第 i 次 TABLE-INSERT 操作中扩展时，实际成本 c_i 等于 i（重新插入 i 1 个项并插入第 i 个项），摊销成本为

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi_i \\ &= i + (3 - i) \\ &= 3 .\end{aligned}$$

图 16.4 绘制了 num_i、$size_i$ 和 Φ的值与 i 的关系。请注意，扩展表格的潜力是如何建立的。

16.4.2 表格扩展和收缩

要实现 TABLE-DELETE 操作，从表中删除指定项非常简单。但是，为了限制浪费的空间量，当加载因子变得太小时，您可能希望删除表。Ta-

表收缩类似于表扩展：当表中的项目数下降得太低时，分配一个新的、较小的表，然后将项目从旧表复制到新表中。然后，您可以通过将旧表返回到内存管理系统来释放旧表的存储空间。为了不浪费空间，同时保持较低的摊销成本，插入和删除过程应保留两个属性：

动态表的负载因子下界为一个正常数，上界为 1，而表操作的摊销成本上界为一个常数•。

每个操作的实际成本等于基本插入或删除的数量。

您可能会认为，如果在将项目插入已满的表中时将表大小加倍，那么在删除会导致表容量不足一半的项目时，应该将表大小减半。此策略确实可以保证表的加载因子永远不会低于 1/2。不幸的是，它也可能导致操作的摊销成本非常大。考虑以下场景。对大小为 n/2 的表 T 执行 n 个操作，其中 n 是 2 的精确幂。前 n/2 个操作是插入操作，根据我们之前的分析，总共花费 ，.n /。在此插入序列结束时，T：*num* D T：*size* D n/2。对于第二个 n/2 个操作，执行以下序列：

插入，删除，删除，插入，删除，删除，插入，删除，插入，插入，...

第一次插入导致表的大小扩大到 n。随后的两次删除导致表的大小缩减回 n/2。另外两次插入导致另一次扩展，依此类推。每次扩展和收缩的成本为 '.n/，并且有 '.n/ 个。因此，n 次操作的总成本为 '.n ²/，使得操作的摊销成本为 '.n/。

这种策略的问题是，在表扩展之后，没有足够的删除操作来支付收缩的费用。同样，在表收缩之后，没有足够的插入操作来支付扩展的费用。

我们如何解决这个问题？让表的负载因子降至 1/2 以下。具体来说，在将项目插入已满的表中时继续将表大小加倍，但在删除项目时将表大小减半会导致表变得小于 1/4 满，而不是像以前那样 1/2 满。因此，表的负载因子被常数 1/4 限制在下方，收缩后负载因子立即为 1/2。

膨胀或收缩应耗尽所有累积势能，因此膨胀或收缩后，当载荷因子为 1/2 时，表的势能为 0。图 16.5 显示了这一想法。当载荷因子偏离 1/2 时，

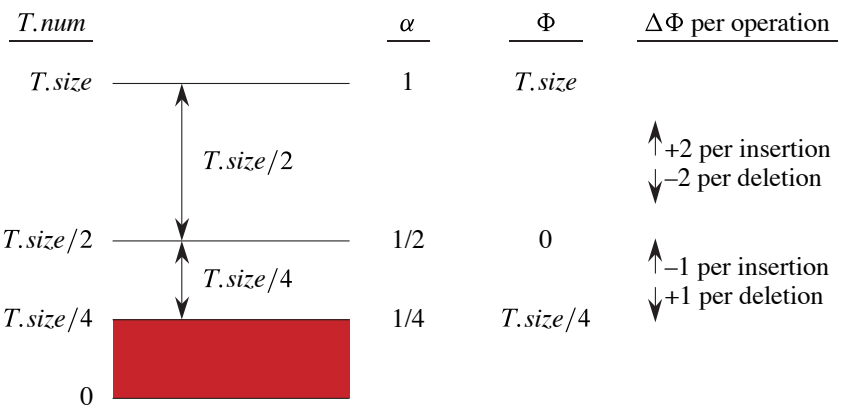


图 16.5 如何思考表插入和删除的势函数 Φ 当负载因子 α为 1/2 时，势为 0。为了在表填满时积累足够的势来支付重新插入所有 T：*size* 个项目的成本，当 α≥ 1/2 时，每次插入时势都需要增加 2。相应地，每次删除时势都会减少 2，剩下 α≥ 1/2。为了在表收缩时积累足够的势来支付重新插入所有 T：*size*/4 个项目的成本，当 α < ≥ 1/2 时，每次删除时势都需要增加 1，相应地，每次插入时势都会减少 1，剩下 α< ≥ 1/2。红色区域表示载荷系数小于 1/4，这是不允许的。

势能增加，因此当发生扩展或收缩时，表已获得足够的势能来支付将所有项目复制到新分配的表中的费用。因此，当负载因子增加到 1 或减少到 1/4 时，势能函数应增长到 T: *num*。在扩展或收缩表后，负载因子立即回到 1/2，表的势能降低回 0。

我们省略了 TABLE-DELETE 的代码，因为它类似于 TABLE-INSERT。我们假设如果在 TABLE-DELETE 期间发生收缩，则它发生在从表中删除项目之后。分析假设每当表中的项目数降至 0 时，表就不会占用存储空间。也就是说，如果 T：*num* D 0，则 T：*size* D 0。

我们如何设计一个势函数，使插入和删除的摊销时间均为常数？当负载因子至少为 1/2 时，我们用于插入的相同势函数 Φ T / D 2.T: *num* T: *size*/2/ 仍然有效。当表至少为半满时，如果表没有扩展，则每次插入都会使势能增加 2；如果每次删除不会导致负载因子降至 1/2 以下，则每次删除都会使势能减少 2。

当负载因子小于 1/2 时，即当 1/4 αT / < 1/2 时，情况又会如何呢？与之前一样，当 αT / D 1/2 时，即 T: *num* D T: *size*/2，潜在的 Φ T / 应为 0。要将负载因子从 1/2 降至 1/4，需要删除 T: *size*/4

发生，此时 T: *num* D T: *size*/4。为了支付所有重新插入的费用，电位必须从 0 增加到 T: *size*/4，以弥补这些 T: *size*/4 次删除。因此，对于每次调用 TABLE-DELETE，直到表收缩，电位应该增加

$$\frac{T.size/4}{T.size/4} = 1 .$$

同样，当 $\alpha < 1/2$ 时，每次调用 TABLE-INSERT 都会使势函数减少 1。当 1/4 = $\alpha(T) < 1/2$ 时，势函数

$$\Phi(T) = T.size/2 - T.num$$

产生这种期望的行为。

将两种情况结合起来，我们得到了势函数

$$\Phi(T) = \begin{cases} 2(T.num - T.size/2) & \text{if } \alpha(T) \geq 1/2 , \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2 . \end{cases} \tag{16.5}$$

空表的势能为 0，并且势能永远不会为负。因此，相对于 Φ的一系列操作的总摊销成本为该序列的实际成本提供了上限。图 16.6 说明了势能函数在一系列插入和删除操作中的表现。

现在，让我们确定每个操作的摊销成本。与前面一样，令 num_i 表示第 i 次操作后存储在表中的项目数，$size_i$ 表示第 i 次操作后表的总大小，α_i D $num_i/size_i$ 表示第 i 次操作后的负载因子，Φ表示第 i 次操作后的潜力，Φ表示由于第 i 次操作而导致的潜力变化。最初，num_0 D 0、$size_0$ D 0 和 Φ_0 D 0。

表不扩展或收缩且加载因子不跨越 αD 1/2 的情况很简单。如我们所见，如果 α_{i-1} 1/2 且第 i 个操作是插入且不会导致表扩展，则 ΦD 2。同样，如果第 i 个操作是删除且 α_i 1/2，则 ΦD 2。此外，如果 $\alpha_{i-1} < 1/2$ 且第 i 个操作是删除操作且不触发收缩，则 ΦD 1；如果第 i 个操作是插入操作且 $\alpha_i < 1/2$ ，则 ΦD 1。换句话说，如果没有发生扩展或收缩，且载荷因子未穿过 αD 1/2，则

如果负载因子保持在 1/2 或以上，则插入的可能性增加 2，删除的可能性减少 2；如果负载因子保持在 1/2 以下，则删除的可能性增加 1，插入的可能性减少 1。

在每种情况下，第 i 个操作的实际成本 c_i 仅为 1，因此

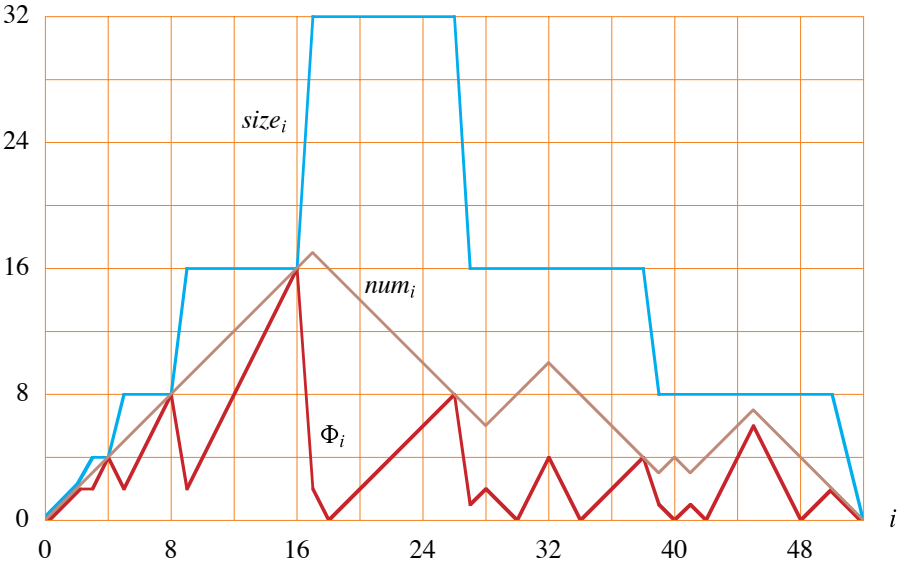


图 16.6 一系列 n 个 TABLE-INSERT 和 TABLE-DELETE 操作对表中项目数 num_i（棕线）、表中插槽数 $size_i$（蓝线）和潜力（红线）的影响

$$\Phi_i = \begin{cases} 2(num_i - size_i/2) & \text{if } \alpha_i \geq 1/2\,, \\ size_i/2 - num_i & \text{if } \alpha_i < 1/2\,, \end{cases}$$

其中 α D $num_i/size_i$，每个都是在第 i 次操作后测量的。在扩张或收缩之前，势能已经积累到表中项目的数量，因此它可以支付将所有项目移动到新表的费用。

如果第 i 个操作是插入，则其摊销成本 y c_i 为 c_i C Φ，如果负载因子保持在 1/2 或以上，则为 1 C 2 D 3，如果负载因子保持在 1/2 以下，则为 1 C .1/ D 0；如果第 i 个操作是删除，则其摊销成本 y c_i 为 c_i C Φ，如果负载因子保持在 1/2 或以上，则为 1 C .2/ D 1，如果负载因子保持在 1/2 以下，则为 1 C 1 D 2。

剩下四种情况：插入操作使负载因子从 1/2 以下变为 1/2，删除操作使负载因子从 1/2 变为 1/2 以下，删除操作导致表收缩，插入操作导致表膨胀。我们在 16.4.1 节末尾分析了最后一种情况，表明其摊销成本为 3。

当第 i 个操作是删除操作导致表收缩时，收缩前有 num_{i-1} D $size_{i-1}/4$，然后删除该项，收缩后最后有 num_i D $size_i/2$ 1。因此，根据公式（16.5），我们有

$$\begin{aligned}\Phi_{i-1} &= size_{i-1}/2 - num_{i-1} \\ &= size_{i-1}/2 - size_{i-1}/4 \\ &= size_{i-1}/4 ,\end{aligned}$$

这也等于删除一个项目并将 $size_{i-1}/4$ 1 个项目复制到新的、较小的表中的实际成本 c_i。由于操作完成后 $num_i = size_i/2$ 1，$\alpha_i < 1/2$，依此类推

$$\begin{aligned}\Phi_i &= size_i/2 - num_i \\ &= 1 ,\end{aligned}$$

给出 $\Phi = 1 \; size_{i-1}/4$。因此，当第 i 个操作是触发收缩的删除时，其摊销成本为

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi_i \\ &= size_{i-1}/4 + (1 - size_{i-1}/4) \\ &= 1 .\end{aligned}$$

最后，我们处理加载因子满足公式 (16.5) 的一种情况（在操作之前）和另一种情况（在操作之后）。我们从删除开始，其中在操作之前有 $num_{i-1} = size_{i-1}/2$，因此 $\alpha_{i-1} = 1/2$，并且在操作之后有 $num_i = size_i/21$，因此 $\alpha_i < 1/2$。因为$\alpha_{i-1} = 1/2$，所以有$\Phi_{i-1} = 0$，又因为$\alpha_i < 1/2$，所以有$\Phi = size_i/2$ $num_i = 1$。由此可得 $\Phi = 1 \; 0 = 1$。由于第i个操作为删除操作，不会引起收缩，因此实际成本c_i等于1，摊销成本$\hat{c}_i$为$c_i + \Phi = 1 + 1 = 2$。

相反，如果第 i 个操作是插入操作，使负载因子从低于 1/2 变为等于 1/2，则潜在变化 Φ等于 1。同样，实际成本 c_i 为 1，现在摊销成本 $\hat{c}_i$ 为 $c_i + \Phi = 1 + 1 = 2$。

总之，由于每个操作的摊销成本都受一个常数的限制，因此动态表上任何 n 个操作序列的实际时间都是 O.n/。

练习

16.4-1
使用势能法，分析第一个表插入的摊销成本。

16.4-2
您希望实现一个动态的开放地址哈希表。当哈希表的负载因子达到某个严格小于 1 的值 α时，为什么您会认为该表已满？简要描述如何使动态开放地址哈希表中的插入操作以这样的方式运行：每个哈希表的摊销成本预期值

插入是 O.1/。为什么每次插入的实际成本的预期值不一定是所有插入的 O.1/？

16.4-3
讨论如何使用会计方法来分析插入和删除操作，假设当表的负载因子超过 1 时，表的大小将加倍，而当表的负载因子低于 1/4 时，表的大小将减半。

16.4-4
假设当表的负载因子低于 1/4 时，不是通过将其大小减半来收缩表，而是当表的负载因子低于 1/3 时，通过将其大小乘以 2/3 来收缩表。使用潜在函数

Φ.T / D j2.T：*num* T：*size*/2/j；
表明使用此策略的 TABLE-DELETE 的摊销成本受常数限制。

问题

16-1Binary reflected Gray code
binary Gray code 表示二进制的非负整数序列，每次从一个整数到下一个整数恰好需要 1 位 üips。***binary reflected Gray code*** 表示整数 0 到 2^k 1 的序列，其中 k 为某个正整数，按照以下递归方法计算：

对于kD1，二进制反射格雷码为h0；1i。
对于 k 2，首先对 k 1 生成二进制反射格雷码，给出 2^{k-1} 个整数 0 到 2^{k-1} 1。然后生成此序列的反射，它只是序列的倒转。（也就是说，序列中的第 j 个整数变为反射中的第 0.2^{k-1} 个整数）。接下来，将 2^{k-1} 添加到反射序列中的每个 2^{k-1} 个整数。最后，连接两个序列。

例如，对于 k D 2，首先形成 k D 1 的二进制反射格雷码 h0;1i。其反射为序列 h1;0i。将 2^{k-1} D 2 添加到反射中的每个整数后得到序列 h3;2i。将这两个序列连接起来得到 h0;1;3;2i，或者以二进制表示为 h00;01;11;10i，因此每个整数都与其前一个整数相差恰好 1 位。对于 k D 3，k D 2 的二进制反射格雷码反射为 h2;3;1;0i，加上 2^{k-1} D 4 后得到 h6;7;5;4i。连接后生成序列 h0;1;3;2;6;7;5;4i，以二进制表示为 h000; 001；011；010；110；111；101；100i。在二进制反射的格雷码中，即使从最后一个整数绕回到第一个整数，也只有一位丢失。

a.将二进制反射格雷码中的整数索引从 0 到 2^k 1，并考虑二进制反射格雷码中的第 i 个整数。要从二进制反射格雷码中的第 i 个整数到第 i 个整数，恰好需要一位 üips。说明如何确定给定索引 i 时哪一位 üips。

b.假设给定一个位数 j，您可以在常数时间内计算一个整数的第 j 位，说明如何在 ，.2^k/ 时间内计算 2^k 个数字的整个二进制反射格雷码序列。

16-2Making binary search dynamic

对已排序数组进行二分查找需要对数搜索时间，但插入新元素的时间与数组大小呈线性关系。您可以通过保留多个已排序数组来改善插入时间。

具体来说，假设您希望在 n 个元素的集合上支持 SEARCH 和 INSERT。令 k≤D≤dlg.nC≤1/e，且 n 的二进制表示形式为 $hn_{k-1}; n_{k-2}; : : :; n_0$ i。维护 k 个排序数组 $A_0; A_1; : : :; A_{k-1}$，其中对于 i≤D≤0; 1; : : :; k≥1，数组 A_i 的长度为 2^i。每个数组要么是满的要么是空的，分别取决于 n_i D≤1 还是 n_i D≤0。因此，所有 k 个数组中保存的元素总数为 $P_{i=0}^{k-1} n_i 2^i$ Dn。虽然每个单独的数组都经过排序，但是不同数组中的元素彼此之间没有特殊的关系。

a.描述如何对此数据结构执行 SEARCH 操作。分析其最坏情况运行时间。***b.*** 描述如何执行 I NSERT 操作。分析其最坏情况和摊销运行时间，假设唯一的操作是 I NSERT 和 SEARCH。***c.*** 描述如何实现 DELETE。分析其最坏情况和摊销运行时间，假设可以有 D ELETE、I NSERT 和 SEARCH 操作。

16-3Amortized weight-balanced trees

考虑一个普通的二叉搜索树，通过向每个节点 x 添加属性 x: *size* 来增强该树，该属性给出了以 x 为根的子树中存储的键的数量。令 α为 $1/2 = \alpha < 1$ 范围内的常数。如果 x: *left*: *size* = α+ x: *size* 且 x: *right* : *size* = α+ x: *size*，则我们称给定节点 x 为 ***α-balanced***。如果树中的每个节点都是 α平衡的，则整棵树为 ***α-balanced***。G. Varghese 提出了以下维护权重平衡树的摊销方法。

a.从某种意义上说，1/2 平衡树是尽可能平衡的。给定任意二叉搜索树中的节点 x，说明如何重建以 x 为根的子树，使其变为 1/2 平衡。您的算法应在 ，.x: *size*/ 时间内运行，并且可以使用 O.x: *size*/ 辅助存储。

b.说明在 n 节点 α平衡二叉搜索树中执行搜索最坏情况需要 O.lg n/ 的时间。

对于该问题的其余部分，假设常数 α严格大于 1/2。假设您像往常一样为 n 节点二叉搜索树实现 I NSERT 和 DELETE，但在每次此类操作之后，如果树中的任何节点不再是 α平衡的，那么您将 “ 重建 ” 以树中最高节点为根的子树，使其变为 1/2 平衡的。

我们将使用潜在方法分析此重建方案。对于二叉搜索树 T 中的节点 x，de ûne

$$\Delta(x) = |x.left.size - x.right.size| \ .$$

解释 T 的潜力

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x),$$

其中 c 是一个足够大的常数，取决于 α。

c.论证任何二叉搜索树都具有非负势能，并且 1/2 平衡树的势能为 0。***d.*** 假设 m 个单位的势能可以用于重建 m 个节点的子树。c 必须以 α为单位有多大才能使重建非 α平衡的子树花费 O.1/ 摊销时间？***e.*** 说明在 n 个节点 α平衡树中插入节点或删除节点需要花费 O.lg n/ 摊销时间。

16-4The cost of restructuring red-black trees

红黑树上有四种基本操作可执行 ***structural modi- fications***：节点插入、节点删除、旋转和颜色变化。我们已经看到，RB-I NSERT 和 RB-DELETE 仅使用 O.1/ 旋转、节点插入和节点删除来维护红黑属性，但它们可能会进行更多颜色变化。

a.描述一个有 n 个节点的合法红黑树，使得调用 RB-I NSERT 添加 .n C 1/st 节点会导致 .lg n/ 颜色变化。然后描述一个合法的

具有 n 个节点的红黑树，在特定节点上调用 RB-DELETE 会导致 .lg n/ 颜色变化。

尽管每次操作的最坏情况下颜色变化次数可能是对数的，但您将证明，在最初为空的红黑树上进行任何 m 次 RB-INSERT 和 RB-DELETE 操作序列都会在最坏情况下导致 O.m/ 结构修改。

b.RB -I NSERT-FIXUP 和 RB-DELETE-FIXUP 代码的主循环处理的一些情况是 ***terminating***：一旦遇到，它们会导致循环在一定数量的附加操作后终止。对于 RB-I NSERT-FIXUP 和 RB-DELETE-F IXUP 的每种情况，指定哪些是终止的，哪些不是。（*Hint:* 查看第 13.3 和 13.4 节中的图 13.5、13.6 和 13.7。）

您将首先分析仅执行插入时的结构修改。假设 T 是红黑树，定义 .T / 为 T 中的红色节点数。假设一个单位的潜力可以支付 RB-INSERT-FIXUP 三种情况中的任何一种所执行的结构修改。

c.令 T ′ 为将 RB-I NSERT-FIXUP 的案例 1 应用于 T 的结果。论证 .T ′ / D .T / 1. ***d.*** 我们可以将 RB-I NSERT 过程的操作分为三个部分。列出由 RB-I NSERT 的 13-16 行、RB-I NSERT-FIXUP 的非终止案例以及 RB-I NSERT-FIXUP 的终止案例所导致的结构修改和潜在变化。***e.*** 使用部分 (d)，论证由任何 RB -I NSERT 调用执行的结构修改的摊销次数为 O.1/。

接下来，你将证明当插入和删除都发生时，存在 O.m/ 结构修改。同样，对于每个节点 x,

$$w.x/D \quad ,,$$

现在重新定义红黑树 T 的势为

$$\Phi(T) = \sum_{x \in T} w(x),$$

其中

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is red,} \\ 1 & \text{if } x \text{ is black and has no red children,} \\ 0 & \text{if } x \text{ is black and has one red child,} \\ 2 & \text{if } x \text{ is black and has two red children.} \end{cases}$$

并让 T′ 成为将 RB-INSERT-FIXUP 或 RB-DELETE-FIXUP 的任何非终止情况应用于 T 所产生的树。

f. 证明对于所有非终止的 RB-I NSERT-FIXUP 情况，Φ.T′/ < Φ.T/1。论证任何 RB-I NSERT-FIXUP 调用所执行的结构修改的摊销次数为 O.1/。***g.*** 证明对于所有非终止的 RB-DELETE-FIXUP 情况，Φ.T′/ < Φ.T/1。论证任何 RB-DELETE-FIXUP 调用所执行的结构修改的摊销次数为 O.1/。***h.*** 完成证明，在最坏情况下，任何 m 个 RB-I NSERT 和 RB-DELETE 操作序列都会执行 O.m/ 次结构修改。

章节注释

Aho、Hopcroft 和 Ullman [5] 使用聚合分析来确定不相交集森林中操作的运行时间。我们将在第 19 章中使用潜在方法分析此数据结构。Tarjan [430] 概述了摊销分析的核算和潜在方法，并介绍了几种应用。他将核算方法归功于几位作者，包括 M. R. Brown、R. E. Tarjan、S. Huddleston 和 K. Mehlhorn。他将潜在方法归功于 D. D. Sleator。术语 “ 摊销 ” 是 D. D. Sleator 和 R. E. Tarjan 的贡献。

势函数对于证明某些类型问题的下界也很有用。对于问题的每种配置，定义一个将配置映射到实数的势函数。然后确定初始配置的势 Φ_{init}、最终配置的势 Φ_{final} 以及由于任何步骤引起的势的最大变化 Φ_{max}。因此，步骤数必须至少为 $|\Phi_{final} - \Phi_{init}| / |\Phi_{max}|$。用于证明 I/O 复杂度下界的势函数示例出现在 Cormen、Sundquist 和 Wisniewski [105]、Floyd [146] 以及 Aggarwal 和 Vitter [3] 的著作中。 Krumme、Cybenko 和 Venkataraman [271] 应用势函数证明了 ***gos- siping*** 的下界：将图中每个顶点的一个唯一项传达给其他每个顶点。

Part V Advanced Data Structures

介绍

本部分将重新研究支持动态集操作的数据结构，但比第三部分更高级。例如，其中一章大量使用了第 16 章中的摊销分析技术。

第 17 章介绍了如何扩充红黑树4（在每个节点中添加附加信息4），以支持除第 12 章和第 13 章中介绍的动态集操作之外的动态集操作。第一个例子扩充了红黑树，以动态维护一组键的顺序统计信息。另一个例子以不同的方式扩充它们以维护实数区间。第 17 章包含一个定理，给出了何时可以扩充红黑树同时保持插入和删除的 O.lg n/ 运行时间的充分条件。

第 18 章介绍了 B 树，它是专门设计用于存储在磁盘上的平衡搜索树。由于磁盘的运行速度比随机存取存储器慢得多，因此 B 树的性能不仅取决于动态集操作所消耗的计算时间，还取决于它们执行的磁盘访问次数。对于每个 B 树操作，磁盘访问次数会随着 B 树的高度而增加，但 B 树操作会保持较低的高度。

第 19 章探讨了不相交集的数据结构。从 n 个元素组成的集合开始，每个元素最初都在自己的单例集合中，UNION 操作将两个集合合并起来。在任何时候，n 个元素都被划分为不相交的集合，即使对 UNION 操作的调用会动态更改集合的成员。查询 FIND-SET 识别当前包含给定元素的唯一集合。将每个集合表示为一个简单的根树会产生令人惊讶的快速操作：m 个操作序列在 O.m ˛.n/// 时间内运行，其中 ˛.n/ 是一个增长速度非常慢的函数 4˛.n/ 在任何可以想象的应用中最多为 4。证明这个时间界限的摊销分析与数据结构一样简单。

本部分涵盖的主题绝不是“高级”数据结构的唯一示例。其他高级数据结构包括以下内容：

Fibonacci heaps [156] 实现了可合并堆（见第 268 页的问题 10-2），INSERT、MINIMUM 和 UNION 操作仅花费 O.1/ 实际和摊销时间，EXTRACT-MIN 和 DELETE 操作花费 O.lg n/ 摊销时间。然而，这些数据结构最显著的优势是 DECREASE-KEY 仅花费 O.1/ 摊销时间。后来开发的 ***Strict Fibonacci heaps*** [73] 使所有这些时间界限都成为现实。因为 DECREASE-KEY 操作花费的摊销时间是常数，所以（严格）斐波那契堆构成了迄今为止一些图问题渐近最快的算法的关键组件。

Dynamic trees [415, 429] 维护一个由不相交的有根树组成的森林。每棵树上的每条边都有一个实值成本。动态树支持查询以查找父节点、根、边成本以及从节点到根的简单路径上的最小边成本。可以通过切割边、更新从节点到根的简单路径上的所有边成本、将根链接到另一棵树以及使节点成为其所在树的根来操作树。动态树的一种实现为每个操作提供了 O.lg n/ 的摊销时间界限，而更复杂的实现则产生 O.lg n/ 的最坏情况时间界限。动态树用于一些渐近最快的网络流算法中。

Splay trees [418, 429] 是一种二叉搜索树，标准搜索树操作在 O.lg n/ 摊销时间内运行。伸展树的一个应用是简化动态树。

Persistent 数据结构允许查询数据结构的过去版本，有时也允许更新。例如，只需花费很少的时间和空间，链接数据结构就可以持久化 [126]。问题 13-1 给出了持久动态集的一个简单示例。

对于有限的键值范围，有几种数据结构可以更快地实现字典操作（INSERT、DELETE 和 SEARCH）。通过利用这些限制，它们能够实现比基于比较的数据结构更好的最坏情况渐近运行时间。如果键是从集合 f0；1；2；：：：；u 1g 中抽取的唯一整数，其中 u 是 2 的精确幂，则递归数据结构（称为 ***van Emde Boas tree*** [440, 441]）支持 SEARCH、INSERT、DELETE、MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 中的每一个操作，时间为 O.lg lg u/s。 ***Fusion trees*** [157] 是第一个允许更快的字典操作的数据结构，当宇宙被限制为整数时，这些操作在 O.lg n/ lg lg n/ 时间内实现。几个后续数据结构，包括 ***expo- nential search trees*** [17]，也对部分或全部给出了改进的界限

字典操作并在本书的章节注释中提到。

Dynamic graph data structures 支持各种查询，同时允许通过插入或删除顶点或边的操作来改变图的结构。它们支持的查询示例包括顶点连通性 [214]、边连通性、最小生成树 [213]、双连通性和传递闭包 [212]。

本书的章节注释中提到了额外的数据结构。