

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

数据结构（C++语言版）

邓俊辉

清华大学出版社

二零一一年十月 · 北京

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

丛书序

“清华大学计算机系列教材”已经出版发行了 30 余种，包括计算机科学与技术专业的基础数学、专业技术基础和专业等课程的教材，覆盖了计算机科学与技术专业本科生和研究生的主要教学内容。这是一批至今发行数量很大并赢得广大读者赞誉的书籍，是近年来出版的大学计算机专业教材中影响比较大的一批精品。

本系列教材的作者都是我熟悉的教授与同事，他们长期在第一线担任相关课程的教学工作，是一批很受本科生和研究生欢迎的任课教师。编写高质量的计算机专业本科生（和研究生）教材，不仅需要作者具备丰富的教学经验和科研实践，还需要对相关领域科技发展前沿的正确把握和了解。正因为本系列教材的作者们具备了这些条件，才有了这批高质量优秀教材的产生。可以说，教材是他们长期辛勤工作的结晶。本系列教材出版发行以来，从其发行的数量、读者的反映、已经获得的国家级与省部级的奖励，以及在各个高等院校教学中所发挥的作用上，都可以看出本系列教材所产生的社会影响与效益。

计算机学科发展异常迅速，内容更新很快。作为教材，一方面要反映本领域基础性、普遍性的知识，保持内容的相对稳定性；另一方面，又需要跟踪科技的发展，及时地调整和更新内容。本系列教材都能按照自身的需要及时地做到这一点。如王爱英教授等编著的《计算机组成与结构》、戴梅萼教授等编著的《微型计算机技术及应用》都已经出版了第四版，严蔚敏教授的《数据结构》也出版了三版，使教材既保持了稳定性，又达到了先进性的要求。

本系列教材内容丰富，体系结构严谨，概念清晰，易学易懂，符合学生的认知规律，适合于教学与自学，深受广大读者的欢迎。系列教材中多数配有丰富的习题集、习题解答、上机及实验指导和电子教案，便于学生理论联系实际地学习相关课程。

随着我国进一步的开放，我们需要扩大国际交流，加强学习国外的先进经验。在大学教材建设上，我们也应该注意学习和引进国外的先进教材。但是，“清华大学计算机系列教材”的出版发行实践以及它所取得的效果告诉我们，在当前形势下，编写符合国情的具有自主版权的高质量教材仍具有重大意义和价值。它与国外原版教材不仅不矛盾，而且是相辅相成的。本系列教材的出版还表明，针对某一学科培养的要求，在教育部等上级部门的指导下，有计划地组织任课教师编写系列教材，还能促进对该学科科学、合理的教学体系和内容的研究。

我希望今后有更多、更好的我国优秀教材出版。

清华大学计算机系教授
中国科学院院士

张钹

序

为适应快速发展的形势，计算机专业基础课的教学必须走内涵发展的道路，扎实的理论基础、计算思维能力和科学的方法论是支撑该学科从业人员进行理性思维和理性实践的重要基础。“程序设计基础”、“面向对象技术”、“离散数学”以及“数据结构”等相关课程，构成了清华大学计算机系专业基础课程体系中的一条重要脉络。近年来为强化学生在计算思维和实践能力方面的训练力度，课程组通过研究，探索和实践，着力对该课程系列的教学目标、内容、方法和各门课的分工，以及如何衔接等进行科学而系统的梳理，进一步明确了教学改革的方向。在这样的背景下，由邓俊辉撰写的《数据结构（C++语言版）》正式出版了。

为了体现教材的先进性，作者研读并参考了计算学科教学大纲（ACM/IEEE Computing Curricula），结合该课程教学的国际发展趋势和对计算机人才培养的实际需求，对相关知识点做了精心取舍，从整体考虑加以编排，据难易程度对各章节内容重新分类，给出了具体的教学计划方案。

为了不失系统性，作者依据多年教学积累，对各种数据结构及其算法，按照分层的思想精心进行归纳和整理，并从数据访问方式、数据逻辑结构、算法构成模式等多个角度，理出线索加以贯穿，使之构成一个整体，使学生在学习数据结构众多知识点的同时，获得对这门学问相关知识结构的系统性和全局性的认识。

计算机学科主张“抽象第一”，这没有错，但弄不好会吓倒或难倒学生。本书从具体实例入手，运用“转换-化简”、“对比-类比”等手法，借助大量插图和表格，图文并茂地展示数据结构组成及其算法运转的内在过程与规律，用形象思维帮助阐释抽象过程，给出几乎所有算法的具体实现，并通过多种版本做剖析和对比，引领读者通过学习提升抽象思维能力。

计算机学科实践性极强，不动手是学不会的。为了强化实践，本书除了每章都布置人人必做的习题和思考题外，还有不少于授课学时的上机编程要求，旨在培养学理性思维和理性实践的动脑动手能力。

中国计算机科学与技术学科教程 2002 曾批评国内有关程序设计类的课，一是淡化算法，二是“一开始就扎进程序设计的语言细节中去”。本书十分重视从算法的高度来讲述数据结构与算法的相互依存关系，在书的开篇就用极其精彩的例子讲清了算法效率和算法复杂度度量的基本概念和方法，这就给全书紧密结合算法来讲数据结构打下了很好的基础。

这本书是精心策划和撰写的，结构严整，脉络清晰，行文流畅，可读性强。全书教学目标明确，内容丰富，基本概念和基本方法的阐述深入浅出，最大的特点是将算法知识、数据结构和编程实践有机地融为一体。我以为，引导学生学好本书，对于奠定扎实的学科基础，提高计算思维能力能够起到良好的作用。



前言

背景

伴随着计算学科（**Computing Discipline**）近二十多年来的迅猛发展，相关专业方向不断细化和分化，相应地在计算机教育方面，人才培养的定位与目标呈现明显的多样化趋势，在知识结构与专业素养方面对人才的要求也在广度与深度上拓展到空前的水平。以最新版计算学科教学大纲（**ACM/IEEE Computing Curricula**, 以下简称 CC 大纲）为例，2001 年制定的 **CC2001** 因只能覆盖狭义的计算机科学方向而更多地被称作 **CS2001**。所幸的是，**CC2001** 的意义不仅在于针对计算机科学方向的本科教学提出了详细的指导意见，更在于构建了一个开放的 **CC2001** 框架（**CC2001 Model**）。按照这一规划，首先应该顺应计算学科总体发展的大势，沿着计算机科学（**CS**）、计算机工程（**CE**）、信息系统（**IS**）、信息技术（**IT**）和软件工程（**SE**）以及更多潜在的新学科方向，以分卷的形式制定相应的教学大纲计划，同时以综述报告的形式概括统领；另外，不宜仍拘泥于十年的周期，而应更为频繁地调整和更新大纲，以及时反映计算领域研究的最新进展，满足应用领域对人才的现实需求。

饶有意味的是，无论从此后发表的综述报告还是各分卷报告都可看出，作为计算学科知识结构的核心与技术体系的基石，数据结构与算法的基础性地位不仅没有动摇，反而得到进一步的强化和突出，依然是计算学科研究开发人员的必备素养，以及相关应用领域专业技术人员的看家本领。以 CC 大纲的综述报告（**Computing Curricula 2005 - The Overview Report**）为例，在针对以上五个专业方向本科学位所归纳的共同要求中，数据结构与算法作为程序设计概念与技能的核心，紧接在数学基础之后列第二位。这方面的要求可进一步细分为五个层次：对数据结构与算法核心地位的充分理解与认同，从软件视角对处理器、存储器及显示器等硬件资源的透彻理解，通过编程以软件方式实现数据结构与算法的能力，基于恰当的数据结构与算法设计并实现大型结构化组件及其之间通讯接口的能力，运用软件工程的原理与技术确保软件鲁棒性、可靠性及其面向特定目标受众的针对性的能力。

自上世纪末起，我有幸参与和承担清华大学计算机系以及面向全校“数据结构”课程的教学工作，在学习和吸收前辈们丰富而宝贵教学经验的同时，通过悉心体会与点滴积累，逐步摸索和总结出一套较为完整的教学方法。作为数据结构与算法一线教学工作者中的一员，我与众多的同行一样，在为此类课程的重要性不断提升而欢欣鼓舞的同时，更因其对计算学科人才培养决定性作用的与日俱增而倍感责任重大。尽管多年来持续推进的教学改革已经取得巨大的进展，但面对新的学科发展形势和社会发展需求，为从根本上提高我国计算机理论及应用人才的培养质量，我们的教学理念、教学内容与教学方法仍然有待于进一步突破。

与学校“高素质、高层次、多样化、创造性”人才培养总体目标相呼应，我所在的清华大学计算机系长期致力于培养“面向基础或应用基础的科学技术问题，具备知识创新、技术创新或集成创新能力的研究型人才”。沿着这个大方向，近年来我与同事们从讲授、研讨、作业、实践、考核和教材等方面入手，在系统归纳已有教学资源和成果的基础上，着力推进数据结构的课程建设与改革。其中，教材既为所授知识提供了物化的载体，也为传授过程指明了清晰的脉络，更为教师与学生之间的交流建立了统一的平台，其中重要性不言而喻。继

2006 年出版《数据结构与算法（Java 语言描述）》之后，本教材的出版也是编者对自己数据结构与算法教学工作的又一次系统总结与深入探索。

原则

在读者群体定位、体例结构编排以及环节内容取舍等方面，全书尽力贯彻以下原则。

■ 兼顾基础不同、目标不同的多样化读者群体

全书十二章按四大部分组织，既相对独立亦彼此呼应，难度较大的章节以星号标注，教员与学生可视具体情况灵活取舍。其中第一章绪论旨在尽快地将背景各异的读者引导至同一起点，为此将系统地引入计算与算法的一般性概念，确立时空复杂度的度量标准，并以递归为例介绍算法设计的一般模式；第二至七章为基础部分，涵盖序列、树、图、初级搜索树等基本数据结构及其算法的实现方法及性能分析，这也是多数读者在实际工作中最常涉及的内容，属于研读的重点；第八至十章为进阶部分，介绍高级搜索树、词典和优先级队列等高级数据结构，这部分内容对更加注重计算效率的读者将很有帮助；最后两章分别以串匹配和高级排序算法为例，着重介绍算法性能优化以及针对不同应用需求的调校方法与技巧，这部分内容可以帮助读者深入理解各类数据结构与算法在不同实际环境中适用性的微妙差异。

■ 注重整体认识，着眼系统思维

全书体例参照现代数据结构普遍采用的分类规范进行编排，其间贯穿以具体而本质的线索，帮助读者在了解各种具体数据结构之后，通过概括与提升形成对数据结构家族的整体性认识。行文从多个侧面体现“转换-化简”的技巧，引导读者逐步形成和强化计算思维（computational thinking）的意识与习惯，从方法论的高度掌握利用计算机求解问题的一般性规律与方法。

比如从逻辑结构的角度，按照线性、半线性和非线性三个层次对数据结构进行分类，并以遍历算法为线索，点明不同层次之间相互转换的技巧。又如，通过介绍动态规划、减而治之、分而治之等算法策略，展示如何将人所擅长的概括化简思维方式与计算机强大的枚举迭代能力相结合，高效地求解实际应用问题。再如，从数据元素访问形式的角度，按照循秩访问（call-by-rank）、循位置访问（call-by-position）或循链接访问（call-by-link）、循关键码访问（call-by-key）、循值访问（call-by-value）、循优先级访问（call-by-priority）等方式对各种数据结构做了归类，并指明它们之间的联系与区别。

通过引入代数判定树模型以及对应的下界等概念，并讲解如何针对具体计算模型确定特定问题的复杂度下界，破除了部分读者对计算机计算能力的盲目迷信。按照 CC 大纲综述报告的归纳结论，这也是对计算学科所有专业本科毕业生共同要求中的第三点——不仅需要了解计算机技术可以做什么（possibilities）以及如何做，更需要了解不能做什么（limitations）以及为什么。

■ 尊重认知规律，放眼拓展提升

在相关学科众多的专业基础课程中，数据结构与算法给学生留下的印象多是内容深、难度大，而如何让学生打消畏难情绪从而学有所乐、学有所获，则是摆在每位任课教师面前的课题。计算机教学有其独特的认知规律，整个过程大致可以分为记忆（remember）、理解（understand）、应用（apply）、分析（analyze）、评估（evaluate）和创造（create）等若干阶段，本书也按照这一脉络，在叙述方式上做了一些粗浅的尝试。

为加深记忆与理解，凡重要的知识点均配有插图。全书共计 230 多组 300 余幅插图，借助视觉通道，从原理、过程、实例等角度使晦涩抽象的知识点得以具体化、形象化，也就是鲁迅先生“五到”读书法中的第一条“眼到”。

为加深对类似概念或系列概念的综合理解，完成认识上的提升，还普遍采用“对比”的手法。例如，优先级队列接口不同实现方式之间的性能对比、快速排序算法不同版本在适用范围上的对比，等等。又如，通过 Dijkstra 算法和 Prim 算法的横向对比，提炼和抽象出更具一般性的优先级搜索框架，并反过来基于这一认识实现统一的搜索算法模板。

为强化实践能力的培养，多从具体的应用问题入手，经逐步分析导出具体的解决方法。所列 230 余段代码，均根据讲述的侧重按模块划分，在力求简洁的同时也配有详实的备注解说。读者可以下载代码，边阅读边编译执行，真正做到“手到”和“心到”。几乎所有实现的数据结构均符合对应的抽象数据类型接口标准，在强化接口规范的同时，从习惯与方式上为读者日后的团队协作做铺垫与准备。

在分析与评估方面，介绍了算法复杂度的典型层次及分析技巧，包括常规的最坏情况和平均情况分析，以及分摊分析。针对递归算法，还着重介绍了递归跟踪法与递推方程法。另外从实用的角度，还引入了稳定性、就地性等更为精细的性能评估尺度，并结合部分算法做了相关的分析对比。

数据结构与算法这二者之间相辅相成的关系，也是本书着重体现的一条重要线索。为此，本书的体例与多数同类教材不尽相同。以排序算法为例，除最后一章外，大部分排序算法都作为对应数据结构的应用实例，分散编入相应的章节：其中起泡排序、归并排序、插入排序、选择排序等算法以排序器的形式归入序列部分；桶排序和基数排序归入散列部分；而堆排序则归入优先级队列部分。再如，图算法及其基本实现均前置到第 6 章，待到后续章节引入高级数据结构时再介绍其优化方法，如此前后呼应。行文讲述中也着力突出数据结构对高效算法的支撑作用，以及源自应用的算法问题对数据结构发展的反向推动作用，优先级队列之于 Huffman 编码算法、完全二叉堆之于就地堆排序、伸展树之于基于局部性原理的缓存算法、散列表之于数值空间与样本空间规模差异的弥合算法等，均属于这方面的实例。

与许多课程的规律一样，习题对于数据结构与算法而言也是强化和提升学习效果的必由之途，否则无异于“入宝山而空返”。当然，好的习题不应仅限于对讲授内容的重复与堆砌，而应更多地侧重于拓展与反思。本书的拓展型习题既包括对书中数据结构接口的扩充、算法性能的改进，也包括通过查阅文献资料补充相关的知识点。另外，一些难度极大或者难度不大但过程繁琐的内容，在这里也以习题的形式留待课后进一步探讨。在求知求真的过程中，质疑与批判是难能可贵的精神，反诘与反思更是创造创新的起点。从吸收到反思，在某种意义上也就是学习（learning）与反学习（unlearning）反复迭代、不断上升的过程。为此，部分习题的答案并非简单地重复正文的结论，甚至并不具有固定的答案，以给读者日后灵活的运用与创新留下足够的空间。

说明

书中凡重要的专业词汇均注有原文，插图中的标注也多以英文给出，因为编者认为这都是进一步钻研以及与国际同行交流的基础。公式多采用接近代码的风格，而非严格的数学格式，以利于理解算法。

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

前言

书中涉及的所有代码以及大量尚未在书中列出的辅助代码，均按 Visual Studio 工程形式分成 50 多组，并统一到名为 DSACPP 的解决方案之下，完整的代码包可从本书主页下载后直接编译执行。

为精简篇幅、突出重点，在一定程度上牺牲了软件规范性甚至计算效率，读者不必盲目效仿。比如，为尽量利用页面宽度和便于投影式播放，全文源代码统一采用 Java 风格编排，但代码的层次感却因此有所削弱，代码片段的切分也有过度之嫌。同样出于简化的考虑，代码中一些本可优化但可能影响总体思路的细节也被忽略。另外，对错误与意外的处理也采用了简化的处理方式。

限于编者的水平与经验，书中一定不乏纰漏与谬误之处，恳请读者及专家批评指正。

龙俊峰

2011 年夏末于清华园

教学计划编排方案建议

采用本书作为教材时，视学生基础、专业方向、教学目标及允许课时总量的不同，授课教师可参照以下典型方案分配课内学时，通常还需另外设置约 50% 的课外编程实验学时。

教学内容			教学方案与课内学时分配						
部分	章	节	方案 A	方案 B	方案 C	方案 D	方案 E	方案 F	方案 G
一 基础知识	第 1 章 绪论	1.1 + 1.2 + 1.3 + 1.5	2.5	2.5	3.5	4.5	3.5	2.5	3
		1.4 [*]	1.5	1.5	2	2.5	2.5		2
二 基本数据 结构	第 2 章 向量	2.1 ~ 2.6	3	3	3	4	3	2.5	3
		2.7 [*]	1		1.5	2	1		
		2.8	2	2	2	3	2	2	2
	第 3 章 列表	3.1 ~ 3.4	2	2	3	4	3	2	3
		3.5	2	2	3	4	3	2	3
	第 4 章 栈与队列	4.1 ~ 4.3	2	2	2	3	2	2	2
		4.4 [*]	3	3	3	3	3		3
		4.5 ~ 4.6	1	1	1	2	1	2	1
	第 5 章 二叉树	5.1 + 5.3	2	2	2	2	2	2	2
		5.2 + 5.4	2	2	3	3	3		3
		5.5	2	2	3	3	3	2	2
三 高级数据 结构	第 6 章 图	6.1 ~ 6.4	1.5	1.5	2	2	2	2	2
		6.5 ~ 6.8	2.5	2.5	2	4	3	3	3
		6.9 [*]	1	1	2	2			
		6.10 ~ 6.12	2	2	2	4	3	2	2
	第 7 章 搜索树	7.1 ~ 7.2	2	2	3	6	4	3	3
		7.3 ~ 7.4	2	2	3	6	4	3	3
	第 8 章 高级搜索树	8.1 ~ 8.2	2	2	3				
		8.3 [*] ~ 8.4 [*]	3		3				
	第 9 章 词典	9.1 + 9.3	2	2	2				
		9.2 [*] + 9.4 [*]	4		4				
四 算法	第 10 章 优先级队列	10.1 ~ 10.2	4	4	4				
		10.3 [*]	2		2				
	第 11 章 串	11.1 ~ 11.3	2	2					2
		11.4 [*] ~ 11.5 [*]	2						2
	第 12 章 排序	12.1	2	2					2
		12.2 [*] ~ 12.3 [*]	4						

本书所有相关教学资料均向公众开放，包括勘误、讲义、插图、代码及部分习题解答等。

欢迎访问教材主页：<http://166.111.138.40/~deng/dsacpp/>

致谢

感谢严蔚敏教授，廿多年前是她引领我进入数据结构的殿堂；感谢吴文虎教授，在追随他参与信息学相关竞赛组织工作的过程中，我更加切实地感受到了算法之宏之美。感谢殷人昆、王宏、朱仲涛、徐明星等老师，在与他们的教学合作过程中我获益良多。感谢众多的同行，与他们的交流和探讨每每令我思路顿开。感谢数以千计的学生，他们是我写作的最终动机与不竭动力，无论是在课堂或是课后，与他们相处的时光都属于我在清华园最美好的记忆。

历年的助教研究生不仅出色地完成了繁重的课外辅导与资源建设工作，他们丰富的想象力和创造力更是我重要的灵感来源，在此特别感谢他们对我的帮助（按担任助教时间先后，截至 2011 年）：王智、李云翀、赵乐、肖晶、刘汝佳、高岳、沈超慧、李锐喆、于泽、白彦冰、夏龙、向阳、姚姜源。

感谢国家 863 高技术研究发展计划课题（2010AA012402）的资助。

感谢龙启铭先生及出版社各位编辑，无论偏执于完美的我每次将“定稿”改得如何面目全非，他们始终报以宽容和耐心。正是依靠他们的鼎力支持，本书才得以顺利出版。

随着最后一个字符的敲出，编辑器的记录显示，本书的写作已逾 1000 小时，这还不包括纸面修订以及编写调试代码的用时。因此我要特别感谢我的父母、太太还有女儿，离开他们的关心和鼓励，很难想象自己如何能够坚持到底，并最终完成这一时间复杂度高达三年的艰巨工程。我欠他们的太多。

今年恰逢清华大学百年华诞，南昌二中（心远中学）也迎来 110 周年校庆，谨奉上此书，权作对母校培育之恩的点滴回报。

简要目录

第 1 章 绪论	1	习题	113
§1.1 计算机与算法	2		
§1.2 复杂度度量	8		
§1.3 复杂度分析	11		
§1.4 *递归	16		
§1.5 抽象数据类型	25		
习题	25		
第 2 章 向量	29		
§2.1 从数组到向量	30		
§2.2 接口	31		
§2.3 构造与析构	34		
§2.4 动态空间管理	35		
§2.5 常规向量	39		
§2.6 有序向量	45		
§2.7 *排序与下界	55		
§2.8 排序器	58		
习题	63		
第 3 章 列表	67		
§3.1 从向量到列表	68		
§3.2 接口	69		
§3.3 列表	72		
§3.4 有序列表	79		
§3.5 排序器	80		
习题	85		
第 4 章 栈与队列	87		
§4.1 栈	88		
§4.2 栈与递归	90		
§4.3 典型应用	93		
§4.4 *试探回溯法	103		
§4.5 队列	109		
§4.6 队列应用	111		
第 5 章 二叉树	115		
§5.1 二叉树及其表示	116		
§5.2 编码树	119		
§5.3 二叉树的实现	122		
§5.4 Huffman 编码	129		
§5.5 遍历	142		
习题	152		
第 6 章 图	155		
§6.1 概述	156		
§6.2 抽象数据类型	159		
§6.3 邻接矩阵	161		
§6.4 邻接表	164		
§6.5 图遍历算法概述	165		
§6.6 广度优先搜索	165		
§6.7 深度优先搜索	168		
§6.8 拓扑排序	171		
§6.9 *双连通域分解	174		
§6.10 优先级搜索	178		
§6.11 最小支撑树	180		
§6.12 最短路径	183		
习题	186		
第 7 章 搜索树	189		
§7.1 查找	190		
§7.2 二叉搜索树	191		
§7.3 平衡二叉搜索树	197		
§7.4 AVL 树	200		
习题	207		
第 8 章 高级搜索树	209		
§8.1 伸展树	210		
§8.2 B-树	220		

§8.3 *红黑树	234
§8.4 *kd-树	244
习题	250
第 9 章 词典	251
§9.1 词典	252
§9.2 *跳转表	254
§9.3 散列表	264
§9.4 *散列应用	281
习题	285
第 10 章 优先级队列	287
§10.1 优先级队列	288
§10.2 堆	292
§10.3 *左式堆	303
习题	308
第 11 章 串	311
§11.1 串及串匹配	312
§11.2 蛮力算法	315
§11.3 KMP 算法	317
§11.4 *BM 算法	323
§11.5 *Karp-Rabin 算法	331
习题	335
第 12 章 排序	337
§12.1 快速排序	338
§12.2 *选取与中位数	344
§12.3 *希尔排序	353
习题	358
附录	359
插图索引	360
表格索引	367
算法索引	368
代码索引	369
关键词索引	375

详细目录

第1章 绪论	1		
§1.1 计算机与算法	2	2.2.2 操作实例	32
1.1.1 古埃及人的绳索	2	2.2.3 Vector 模板类	32
1.1.2 欧几里德的尺规	3	§2.3 构造与析构	34
1.1.3 起泡排序	4	2.3.1 默认构造方法	34
1.1.4 算法	5	2.3.2 基于复制的构造方法	34
1.1.5 算法效率	7	2.3.3 析构方法	35
§1.2 复杂度度量	8	§2.4 动态空间管理	35
1.2.1 时间复杂度	8	2.4.1 静态空间管理	35
1.2.2 渐进复杂度	9	2.4.2 可扩充向量	36
1.2.3 空间复杂度	11	2.4.3 扩容	36
§1.3 复杂度分析	11	2.4.4 分摊分析	37
1.3.1 常数 $\mathcal{O}(1)$	11	2.4.5 缩容	38
1.3.2 对数 $\mathcal{O}(\log n)$	12	§2.5 常规向量	39
1.3.3 线性 $\mathcal{O}(n)$	13	2.5.1 直接引用元素	39
1.3.4 多项式 $\mathcal{O}(\text{polynomial}(n))$	14	2.5.2 置乱器	39
1.3.5 指数 $\mathcal{O}(2^n)$	14	2.5.3 判等器与比较器	40
1.3.6 复杂度层次	15	2.5.4 无序查找	40
1.3.7 输入规模	15	2.5.5 插入	41
§1.4 *递归	16	2.5.6 删除	42
1.4.1 线性递归	16	2.5.7 唯一化	43
1.4.2 递归分析	17	2.5.8 遍历	44
1.4.3 递归模式	18	§2.6 有序向量	45
1.4.4 递归消除	21	2.6.1 比较器	45
1.4.5 二分递归	22	2.6.2 有序性甄别	46
§1.5 抽象数据类型	25	2.6.3 唯一化	46
习题	25	2.6.4 查找	48
第2章 向量	29	2.6.5 二分查找 (版本 A)	48
§2.1 从数组到向量	30	2.6.6 Fibonacci 查找	51
2.1.1 数组	30	2.6.7 二分查找 (版本 B)	53
2.1.2 向量	31	2.6.8 二分查找 (版本 C)	54
§2.2 接口	31	§2.7 *排序与下界	55
2.2.1 ADT 接口	31	2.7.1 有序性	55

2.7.5 估计下界.....	58	4.1.2 操作实例	89
§2.8 排序器.....	58	4.1.3 Stack 模板类	89
2.8.1 统一入口.....	58	§4.2 栈与递归	90
2.8.2 起泡排序.....	59	4.2.1 递归的实现.....	90
2.8.3 归并排序.....	60	4.2.2 避免递归	92
习题	63	§4.3 典型应用	93
第3章 列表	67	4.3.1 逆序输出	93
§3.1 从向量到列表	68	4.3.2 递归嵌套	94
3.1.1 从静态存储到动态存储	68	4.3.3 延迟缓冲	97
3.1.2 由秩到位置	69	4.3.4 逆波兰表达式.....	101
3.1.3 列表	69	§4.4 *试探回溯法	103
§3.2 接口	69	4.4.1 试探与回溯.....	103
3.2.1 列表节点	69	4.4.2 八皇后	104
3.2.2 列表	70	4.4.3 迷宫寻径	106
§3.3 列表	72	§4.5 队列	109
3.3.1 头、尾节点	72	4.5.1 概述.....	109
3.3.2 默认构造方法	73	4.5.2 ADT 接口.....	110
3.3.3 由秩到位置的转换	73	4.5.3 操作实例	110
3.3.4 查找	74	4.5.4 Queue 模板类	111
3.3.5 插入	74	§4.6 队列应用	111
3.3.6 基于复制的构造	76	4.6.1 循环分配器.....	111
3.3.7 删除	77	4.6.2 银行服务模拟.....	112
3.3.8 析构	77	习题	113
3.3.9 唯一化	78		
3.3.10 遍历	79		
§3.4 有序列表	79	第5章 二叉树	115
3.4.1 唯一化	79	§5.1 二叉树及其表示	116
3.4.2 查找	80	5.1.1 树	116
§3.5 排序器	80	5.1.2 二叉树	117
3.5.1 统一入口	80	5.1.3 多叉树	117
3.5.2 插入排序	81	§5.2 编码树	119
3.5.3 选择排序	82	5.2.1 二进制编码	119
3.5.4 归并排序	83	5.2.2 二叉编码树	121
习题	85	§5.3 二叉树的实现	122
第4章 栈与队列	87	5.3.1 二叉树节点	122
§4.1 栈	88	5.3.2 二叉树节点操作接口	125
4.1.1 ADT 接口	88	5.3.3 二叉树	126
		§5.4 Huffman 编码	129
		5.4.1 PFC 编码	129
		5.4.2 最优编码树	132

5.4.3 Huffman 编码树	135	6.8.3 算法	172
5.4.4 Huffman 编码算法	137	6.8.4 实现	173
§5.5 遍历	142	6.8.5 实例	174
5.5.1 递归式遍历	142	6.8.6 复杂度	174
5.5.2 *迭代版先序遍历	144	§6.9 *双连通域分解	174
5.5.3 *迭代版中序遍历	146	6.9.1 关节点与双连通域	174
5.5.4 *迭代版后序遍历	149	6.9.2 蛮力算法	175
5.5.5 层次遍历	151	6.9.3 算法	175
习题	152	6.9.4 实现	176
第6章 图	155	6.9.5 实例	177
§6.1 概述	156	6.9.6 复杂度	178
§6.2 抽象数据类型	159	§6.10 优先级搜索	178
6.2.1 操作接口	159	6.10.1 优先级与优先级数	178
6.2.2 Graph 模板类	159	6.10.2 基本框架	179
§6.3 邻接矩阵	161	6.10.3 复杂度	179
6.3.1 原理	161	§6.11 最小支撑树	180
6.3.2 实现	161	6.11.1 支撑树	180
6.3.3 时间性能	163	6.11.2 最小支撑树	180
6.3.4 空间性能	163	6.11.3 歧义性	180
§6.4 邻接表	164	6.11.4 蛮力算法	181
6.4.1 原理	164	6.11.5 Prim 算法	181
6.4.2 复杂度	164	§6.12 最短路径	183
§6.5 图遍历算法概述	165	6.12.1 最短路径树	183
§6.6 广度优先搜索	165	6.12.2 歧义性	184
6.6.1 策略	165	6.12.3 Dijkstra 算法	184
6.6.2 实现	166	习题	186
6.6.3 实例	167	第7章 搜索树	189
6.6.4 复杂度	167	§7.1 查找	190
6.6.5 应用	167	7.1.1 循关键码访问	190
§6.7 深度优先搜索	168	7.1.2 词条	191
6.7.1 策略	168	7.1.3 序与比较器	191
6.7.2 实现	168	§7.2 二叉搜索树	191
6.7.3 实例	169	7.2.1 顺序性	191
6.7.4 复杂度	171	7.2.2 中序遍历序列	192
6.7.5 应用	171	7.2.3 BST 模板类	192
§6.8 拓扑排序	171	7.2.4 查找算法及其实现	193
6.8.1 应用	171	7.2.5 插入算法及其实现	194
6.8.2 有向无环图	172	7.2.6 删除算法及其实现	196

§7.3 平衡二叉搜索树	197
7.3.1 树高与性能	197
7.3.2 理想平衡与适度平衡	199
7.3.3 等价二叉搜索树	199
7.3.4 等价变换与局部调整	200
§7.4 AVL 树	200
7.4.1 AVL 树	200
7.4.2 节点插入	202
7.4.3 节点删除	204
7.4.4 统一重平衡算法	206
习题	207
第 8 章 高级搜索树	209
§8.1 伸展树	210
8.1.1 局部性	210
8.1.2 逐层伸展	211
8.1.3 双层伸展	212
8.1.4 *分摊分析	214
8.1.5 伸展树的实现	216
§8.2 B-树	220
8.2.1 多路平衡查找	220
8.2.2 ADT 接口及其实现	223
8.2.3 关键码查找	224
8.2.4 性能分析	225
8.2.5 关键码插入	226
8.2.6 上溢与分裂	227
8.2.7 关键码删除	229
8.2.8 下溢与合并	230
§8.3 *红黑树	234
8.3.1 概述	234
8.3.2 红黑树接口定义	236
8.3.3 节点插入算法	237
8.3.4 节点删除算法	240
§8.4 *kd-树	244
8.4.1 范围查询	244
8.4.2 kd-树	247
8.4.3 基于 2d-树的范围查询	248
习题	250

第 9 章 词典

251

§9.1 词典	252
9.1.1 操作接口	252
9.1.2 操作实例	253
9.1.3 接口定义	254
9.1.4 实现方法	254
§9.2 *跳转表	254
9.2.1 SkipList 模板类	254
9.2.2 总体逻辑结构	255
9.2.3 四联表	256
9.2.4 查找	257
9.2.5 空间复杂度	259
9.2.6 时间复杂度	259
9.2.7 插入	260
9.2.8 删除	263
§9.3 散列表	264
9.3.1 完美散列	265
9.3.2 装填因子与空间利用率	265
9.3.3 散列函数	266
9.3.4 散列表	269
9.3.5 冲突及其排解	271
9.3.6 开放定址策略	273
9.3.7 查找与删除	275
9.3.8 插入	276
9.3.9 更多开放定址策略	278
9.3.10 散列码转换	280
§9.4 *散列应用	281
9.4.1 桶排序	281
9.4.2 最大间隙	283
9.4.3 基数排序	283
习题	285
第 10 章 优先级队列	287
§10.1 优先级队列	288
10.1.1 优先级与优先级队列	288
10.1.2 关键码、比较器与偏序关系	289
10.1.3 操作接口	289

10.1.4 操作实例：选择排序 289	11.3.7 性能分析 320
10.1.5 接口定义 290	11.3.8 继续改进 321
10.1.6 应用实例：Huffman 编码树 290	§11.4 *BM 算法 323
§10.2 堆 292	11.4.1 思路与框架 323
10.2.1 完全二叉堆 292	11.4.2 坏字符策略 324
10.2.2 元素插入 295	11.4.3 好后缀策略 326
10.2.3 元素删除 296	11.4.4 复杂度 330
10.2.4 建堆 298	11.4.5 算法纵览 330
10.2.5 就地堆排序 300	§11.5 *Karp-Rabin 算法 331
§10.3 *左式堆 303	11.5.1 构思 331
10.3.1 堆合并 303	11.5.2 算法与实现 332
10.3.2 单侧倾斜 303	习题 335
10.3.3 PQ_LeftHeap 模板类 304	
10.3.4 空节点路径长度 304	第 12 章 排序 337
10.3.5 左倾性与左式堆 305	§12.1 快速排序 338
10.3.6 右侧链 305	12.1.1 分治策略 338
10.3.7 合并算法 305	12.1.2 轴点 338
10.3.8 实例 306	12.1.3 快速排序算法 339
10.3.9 合并操作 merge() 的实现 306	12.1.4 快速划分算法 339
10.3.10 复杂度 307	12.1.5 复杂度 341
10.3.11 基于合并的插入和删除 307	12.1.6 应对退化 343
习题 308	§12.2 *选取与中位数 344
第 11 章 串 311	12.2.1 概述 344
§11.1 串及串匹配 312	12.2.2 主流数 345
11.1.1 串 312	12.2.3 归并向量的中位数 347
11.1.2 串匹配 313	12.2.4 基于优先级队列的选取 350
11.1.3 测评标准与策略 314	12.2.5 基于快速划分的选取 350
§11.2 蛮力算法 315	12.2.6 k-选取算法 351
11.2.1 算法描述 315	§12.3 *希尔排序 353
11.2.2 算法实现 315	12.3.1 递减增量策略 353
11.2.3 时间复杂度 316	12.3.2 增量序列 355
§11.3 KMP 算法 317	习题 358
11.3.1 构思 317	
11.3.2 next 表 318	附录 359
11.3.3 KMP 算法 318	插图索引 360
11.3.4 next[0] = -1 319	表格索引 367
11.3.5 next[j + 1] 319	算法索引 368
11.3.6 构造 next 表 320	代码索引 369
	关键词索引 375

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

[详细目录](#)

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第1章

绪论

作为万物之灵的人，与动物的根本区别在于理性，而计算则是理性的一种重要而具体的表现形式。计算机是人类从事计算的工具，是抽象计算模型的具体物化。基于图灵模型的现代计算机，既是人类现代文明的标志与基础，更是人脑思维的拓展与延伸。

尽管计算机的性能日益提高，但这种能力在解决实际应用问题时能否真正得以发挥，决定性的关键因素仍在于人类自身。具体地，通过深入思考与分析获得对问题本质的透彻理解，按照长期积淀而成的框架与模式设计出合乎问题内在规律的算法，选用、改进或定制足以支撑算法高效实现的数据结构，并在真实的应用环境中充分测试、调校和改进，构成了应用计算机高效求解实际问题的典型流程与不二法门。任何一位有志于驾驭计算机的学生，都应该从这些方面入手，不断学习，反复练习，勤于总结。

本章将介绍与计算相关的基本概念，包括算法构成的基本要素、算法效率的衡量尺度、计算复杂度的分析方法与界定技巧、算法设计的基本框架与典型模式，这些也构成了全书所讨论的各类数据结构及相关算法的基础与出发点。

§ 1.1 计算机与算法

1946 年问世的 ENIAC 开启了现代电子数字计算机的时代，计算机科学（computer science）也在随后应运而生。计算机科学的核心在于研究计算方法与过程的规律，而不仅仅是作为计算工具的计算机本身，因此 E. Dijkstra 及其追随者更倾向于将这门科学称作计算科学（computing science）。

实际上，人类使用不同工具从事计算的历史可以追溯到更为久远的时代，计算以及计算工具始终与我们如影相随地穿越漫长的时光岁月，不断推动人类及人类社会的进化发展。从最初颜色各异的贝壳、长短不一的刻痕、周载轮回的日影、粗细有别的绳结^①，以至后来的直尺、圆规和算盘，都曾经甚至依然是人类有力的计算工具。

1.1.1 古埃及人的绳索

古埃及人以其复杂而浩大的建筑工程而著称于世，在长期规划与实施此类工程的过程中，他们逐渐归纳并掌握了基本的几何度量和测绘方法。考古研究发现，公元前 2000 年的古埃及人已经知道如何解决如下实际工程问题：通过直线 1 上给定的点 P，作该直线的垂线。

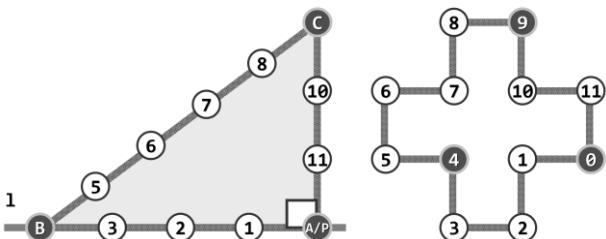


图1.1 古埃及人使用的绳索计算机及其算法

他们的方法如图 1.1 所示，翻译成现代的算法语言，可描述如下。

^① 易经辞云：上古结绳而治，后世圣人，易之以书契

perpendicular(l, P)

输入：直线l及其上一点P

输出：经过P且垂直于l的直线

1. 取12段等长绳索，依次首尾联结成环 //联结处称作“结”，按顺时针方向编号为0..11
2. 奴隶A看管0号结，将其固定于点P处
3. 奴隶B牵动4号结，将绳索沿直线l方向尽可能地拉直
4. 奴隶C牵动9号结，将绳索尽可能地拉直
5. 经过0号和9号结，绘制一条直线

算法1.1 过直线上给定点作直角

以上由奴隶与绳索组成这一系统，就是古希腊人发明的一套计算工具。尽管乍看起来与现代的电子计算机相去甚远，但就其本质而言，二者之间的相似之处远多于差异，它们同样都是用于支持和实现计算过程的物理机制，亦即广义的计算机。因此就这一意义而言，将四千年前的这一计算工具称作“绳索计算机”毫不过份^②。

1.1.2 欧几里德的尺规

欧几里德几何是现代公理系统的鼻祖。从计算的角度来看，针对不同的几何问题，欧氏几何都分别给出了一套几何作图流程，也就是具体的算法。比如，经典的线段三等分过程可描述为如算法 1.2 所示。该算法的一个典型的执行实例如图 1.2 所示。

tripartition(AB)

输入：线段AB

输出：将AB三等分的两个点C和D

1. 从A发出一条与AB不重合的射线ρ
2. 任取ρ上三点C'、D'和B'，使 $|AC'| = |C'D'| = |D'B'|$
3. 联接B'B
4. 过D'做B'B的平行线，交AB于D
5. 过C'做B'B的平行线，交AB于C

算法1.2 三等分给定线段

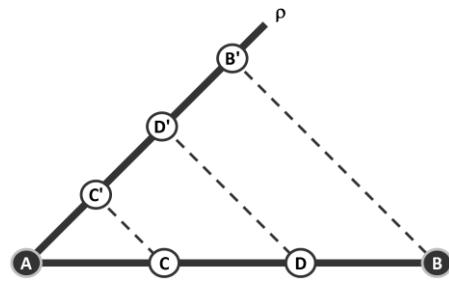


图1.2 古希腊人的尺规计算机

在以上算法中，输入为所给的直线段 AB，输出为将其三等分的 C 和 D 点。我们知道，欧氏几何还给出了大量过程与功能更为复杂的几何作图算法，为将这些算法变成可行的实际操作序列，欧氏几何使用了两种相互配合的基本工具——直尺和圆规——其中前者无需刻度，后者的跨度半径则不受限制。同样地，从计算的角度来看，由直尺和圆规构成的这一物理机制也不妨可以称作“尺规计算机”。在尺规计算机中，可行的基本操作不外乎以下五类：

- 1 过两个点作一直线
- 2 确定两条直线的交点
- 3 以任一点为圆心，以任意半径作一个圆
- 4 确定任一直线和任一圆的交点（若二者的确相交）
- 5 确定两个圆的交点（若二者的确相交）

每一欧氏作图算法均可分解为一系列上述操作的组合，故称之为基本操作恰如其分。

^② 事实上，这类计算机功能之强大远远超出我们的直觉，在某些方面它们甚至超过了现代的电子计算机

1.1.3 起泡排序

D. Knuth^③曾指出，四分之一以上的 CPU 时间都用于执行同一类型的计算：按照某种次序，将给定的一组元素顺序排列，比如将 n 个整数排成一个非降序列。这类操作统称排序（sorting）。就广义而言，我们今天借助计算机所完成的计算任务中，有更高的比例都可归入此类。例如，从浩如烟海的万维网中找出与特定关键词最相关的前 100 个页面，就是此类计算的一种典型形式。排序问题在算法设计与分析中扮演着重要的角色，以下不妨就此做一讨论。为简化起见，这里暂且只讨论对整数的排序。

■ 局部有序与整体有序

在由一组整数组成的序列 $A[0, n-1]$ 中，满足 $A[i-1] \leq A[i]$ 的相邻元素称作顺序的；否则是逆序的。不难看出，有序序列中每一对相邻元素都是顺序的，亦即，对任意 $1 \leq i < n$ 都有 $A[i-1] \leq A[i]$ ；反之，所有相邻元素均顺序的序列，也必然整体有序。

■ 扫描交换

由有序序列的上述特征，我们可以通过不断改善局部的有序性实现整体的有序：从前向后依次检查每一对相邻元素，一旦发现逆序即交换二者的位置。对于长度为 n 的序列，共需做 $n-1$ 次比较和不超过 $n-1$ 次交换，这一过程称作一趟扫描交换。以图 1.3(a) 中由 7 个整数组成的序列 $A[0, 6] = \{5, 2, 7, 4, 6, 3, 1\}$ 为例，在第一趟扫描交换过程中，{5, 2} 交换位置，{7, 4, 6, 3, 1} 循环交换位置，扫描交换后的结果如图(b) 所示。

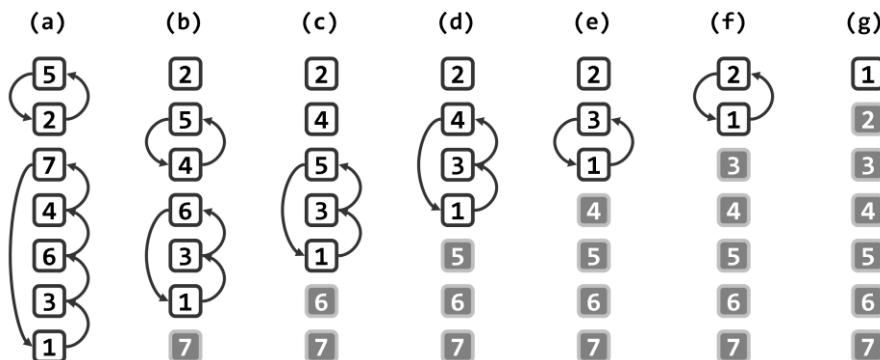


图1.3 通过6趟扫描交换对七个整数排序（其中已就位的元素以深色示意）

■ 起泡排序

可见，经过这样的一趟扫描，序列未必达到整体有序。果真如此，则可对该序列再做一趟扫描交换，比如，图(b) 再经一趟扫描交换的结果如图(c)。事实上，很有可能如图(c~f) 所示需要反复进行多次扫描交换，直到如图(g) 所示在序列中不再含有任何逆序的相邻元素。多数的这类交换操作，都会使得越小（大）的元素朝上（下）方移动（习题[3]），直至它们抵达各自应处的位置。

排序过程中，所有元素朝各自最终位置亦步亦趋的移动过程，犹如气泡在水中的上下沉浮，起泡排序（bubblesort）算法也因此得名。

^③ D. E. Knuth, The Art of Computer Programming, Vol. 3, p.3, ISBN 7-302-05816-4

■ 实现

上述起泡排序的思路，可准确描述和实现为如代码 1.1 所示的函数 bubblesort()。

```
1 void bubblesort1A(int A[], int n) { //起泡排序算法（版本1A）: 0 <= n
2     bool sorted = false; //整体排序标志，首先假定尚未排序
3     while (!sorted) { //在尚未确认已全局排序之前，逐趟进行扫描交换
4         sorted = true; //假定已经排序
5         for (int i = 1; i < n; i++) { //自左向右逐对检查当前范围A[0, n)内的各相邻元素
6             if (A[i-1] > A[i]) { //一旦A[i-1]与A[i]逆序，则
7                 swap(A[i-1], A[i]); //交换之，并
8                 sorted = false; //因整体排序不能保证，需要清除排序标志
9             }
10        }
11        n--; //至此末元素必然就位，故可以缩短待排序序列的有效长度
12    }
13 } //借助布尔型标志位sorted，可及时提前退出，而不致蛮力地做n-1趟扫描交换
```

代码1.1 整数数组的起泡排序

1.1.4 算法

以上三例都可称作算法。那么，究竟什么是算法呢？所谓算法，是基于特定的计算模型，旨在解决某一信息处理问题而设计的一个指令序列。比如，针对“过直线上一点作垂直线”这一问题，基于由绳索和奴隶构成的计算模型，由古埃及人设计的算法 1.1；针对“三等分线段”这一问题，基于由直尺和圆规构成的计算模型，由欧几里德设计的算法 1.2；以及，针对“将若干元素按大小排序”这一问题，基于图灵机模型而设计的 bubblesort() 算法。一般地，本书所说的算法还应必须具备以下要素。

■ 输入与输出

待计算问题的任一实例，都需要以某种方式交给对应的算法，对所求解问题特定实例的这种描述统称为输入（**input**）。对于上述三个例子而言，输入分别是“某条直线及其上一点”、“某条线段”以及“由 n 个整数组成的某一序列”。其中，第三个实例的输入具体地由 $A[]$ 与 n 共同描述和定义，前者为存放待排序整数的数组，后者为整数的总数。

经计算和处理之后得到的信息，即针对输入问题实例的答案，称作输出（**output**）。比如，对于上述三个例子而言，输出分别是“垂直线”、“三等分点”以及“有序序列”。在物理上，输出有可能存放于单独的存储空间中，也可能直接存放于原输入所占的存储空间中。比如，第三个实例即属于后一情形，经排序的整数将按非降次序存放在数组 $A[]$ 中。

■ 基本操作、确定性与可行性

所谓确定性和可行性是指，算法应可描述为一个由若干语义明确的基本操作组成的指令序列，而且每一基本操作在对应的计算模型中均可兑现。

以上述绳索计算机的算法 1.1 为例，整个求解过程可以明白无误地描述为一系列可行的基本操作，比如“取等长绳索”、“联结绳索”、“将绳结固定于指定点”以及“拉直绳索”等。再如尺规计算的算法 1.2 中，“从一点任意发出一条射线”、“在直线上任取三个等距点”、“联接指定两点”等也都属于利用尺规可以兑现的基本操作。细心的读者可能会

注意到，算法 1.2 中涉及的操作并不都是基本的，比如，最后两句都要求“过直线外一点作其平行线”，这本身就是一个几何作图问题。幸运的是，借助基本操作的适当组合，这一子问题也可以圆满解决。因此，解决这一子问题的算法不妨可以称作是算法 1.2 的“子算法”。

从现代程序设计语言的角度，可以更加便捷而准确地理解算法的确定性与可行性。具体地，一个算法满足确定性与可行性，当且仅当它可以通过程序设计语言精确地描述，比如，起泡排序算法可以具体地描述和实现为代码 1.1 中的函数 `bubblesort()`，其中“读取某一元素的内容”、“修改某一元素的内容”、“比较两个元素的大小”、“逻辑表达式求值”以及“根据逻辑判断确定分支转向”等等，都属于现代电子计算机所支持的基本操作。

■ 有穷性与正确性

不难理解，任意算法都应在执行有限次基本操作之后终止并给出输出，此即所谓算法的有穷性（**finiteness**）。进一步地，算法不仅应该迟早会终止，而且所给的输出还应该能够符合由问题本身在事先确定的条件，此即所谓算法的正确性（**correctness**）。

对以上前两个算法实例而言，在针对任一输入实例的计算过程中，每条基本操作语句仅执行一次，故其有穷性不证自明。另外，根据勾股定理以及平行等比原理，其正确性也一目了然。然而对于更为复杂的算法，这两条性质的证明往往颇需费些周折（习题[28]和[29]），有些问题甚至尚无定论（习题[30]）。即便是简单的起泡排序，`bubblesort()` 算法的有穷性和正确性也不是由代码 1.1 自身的结构直接保证的。以下就以此为例做一分析。

■ 起泡排序

图 1.3 给出了 `bubblesort()` 的一次具体执行过程和正确结果，然而严格地说，这远不足以证明起泡排序就是一个名副其实的算法。比如，对于任意一组整数，经过若干趟的起泡交换之后该算法是否总能完成排序？事实上，即便是其有穷性也值得怀疑。就代码结构而言，只有在前一趟扫描交换中未做任何元素交换的情况下，外层循环才会因条件“`!sorted`”不再满足而退出。但是，这一情况对任何输入实例都总能出现吗？反过来，是否存在某一（某些）输入序列，无论做多少趟起泡交换也无济于事？这种担心并非毫无道理。细心的读者或许已注意到，在起泡交换的过程中，尽管多数时候元素会朝着各自的最终位置不断靠近，但有的时候某些元素也的确会暂时朝着远离自己应处位置的方向移动（习题[3]）。

证明算法有穷性和正确性的一个重要技巧，就是从适当的角度审视整个计算过程，并找出其所具有的某种不变性和单调性。其中，单调性通常是指问题的有效规模不断递减。不变性则不仅应在算法初始状态下自然满足，而且应与最终的正确性相呼应——当问题的有效规模缩减到 0 时，不变性应随即等价于正确性。那么具体地，`bubblesort()` 算法的单调性和不变性如何体现呢？

仔细观察图 1.3 之后不难看出，每经过一趟扫描，尽管并不能保证序列达到整体有序，但从“待求解问题的规模”这一角度来看，整体的有序性必然有所改善。以全局最大的元素（图 1.3 中的整数 7）为例，在第一趟扫描交换的过程中，一旦触及该元素，它必将与后续的所有元素一次交换。于是如图 1.3(b) 所示，经过第一趟扫描之后，该最大元素必然就位；而且在此后的各轮扫描交换中，该元素将绝不会参与任何交换。这就意味着，经过一趟扫描交换之后，我们只需关注前面更小的那 $n-1$ 个元素。实际上，这一结论对后续的每一趟扫描交换也都成立，这可由图(c~g)中的 6~2 得到印证。

于是，起泡排序算法的不变性和单调性可分别概括为：经过 k 趟扫描交换之后，最大

的 k 个元素必然就位；经过 k 趟扫描交换之后，待求解问题的有效规模将缩减至 $n-k$ 。如代码 1.1 实现的 `bubblesort()` 算法中，外层 `while` 循环之所以可以不断缩减待排序序列的有效长度 n ，正是基于这一性质。

特别地，在算法初始状态下 $k = 0$ ，这两条性质都自然满足。另一方面由以上单调性可知，无论输入序列如何，至多经 $n-1$ 趟扫描交换后问题的有效规模必将缩减至 1。此时，仅含单个元素的序列的有序性不言而喻；而由该算法的不变性，其余 $n-1$ 个元素在此前的 $n-1$ 次迭代中业已陆续就位。因此，算法不仅必然终止，而且输出的序列必然整体有序，算法的有穷性与正确性由此得证。

■ 退化与鲁棒性

同一问题往往不限于一种算法，而同一算法也常常会有多种实现方式，因此除了以上必须具备的基本属性，在应用环境中还需从实用的角度对不同算法及其不同版本做更为细致考量和取舍。这些细致的要求尽管应纳入软件工程的范畴，但也不失为成熟算法的重要标志。

比如其中之一就是，除一般性情况外，实用的算法还应能够处理各种极端的输入实例。仍以排序问题为例，极端情况下待排序序列的长度可能不是正数（参数 $n = 0$ 甚至 $n < 0$ ），或者反过来长度达到或者超过系统支持的最大值（ $n = \text{INT_MAX}$ ），或者 $A[]$ 中的元素不见得互异甚至全体相等，以上种种都属于所谓的退化（degeneracy）情况。算法所谓的鲁棒性（robustness），就是要求能够尽可能充分地应对此类情况。请读者自行验证，对于以上退化情况，代码 1.1 中 `bubblesort()` 算法依然可以正确返回而不致出现异常。

■ 重用性

从实用角度评判不同算法及其不同实现方式的另一标准着眼于，算法的总体框架能否便捷地推广至其它场合。仍以起泡排序为例。实际上，起泡算法的正确性与所处理序列中元素的类型关系不大，无论是对于 `float`、`char` 或其它类型，只要元素之间可以比较大小，算法的整体框架依然可以沿用。某一算法模式可推广并适用于不同类型基本元素的这种特性，即是重用性的一种典型形式。很遗憾，代码 1.1 中的 `bubblesort()` 算法尚不满足这一要求；稍后在第 2 章和第 3 章中，我们将使包括起泡排序在内的各种排序算法具有这一特性。

1.1.5 算法效率

■ 可计算性

相信本书的读者大都已学习并掌握了至少一种高级程序设计语言，如 C、C++ 或 Java 等。学习程序设计语言的目的，在于学会如何编写合法（即合乎特定程序语言的语法）的程序，从而保证所编写的程序或者能够经过编译器的编译和链接生成可最终的执行代码，或者能够由解释器解释执行。然而从利用计算机解决实际问题的角度来看，这只是第一个层次，仅仅满足这一基本要求还远不足。

以前面提到的有穷性为例。遗憾的是，作为对算法的一项基本而重要的要求，完全合乎语法的程序却往往未必能够满足。几乎所有的读者都应有过这样的体验：很多合法的程序可以顺利编译链接，但在具体运行过程中却因无穷循环或递归溢出导致异常。更糟糕的是，针对许多问题根本不可能设计出必然终止的算法，因此就这个意义而言它们是不可解的。当然，这类问题应归入可计算性（computability）理论的研究范畴，本书不予过多地涉及。

■ 难解性

实际上，我们不仅需要确定算法对任何输入都能够在有穷次操作后终止并输出结果，而且希望计算过程所需的时间尽可能缩短。很遗憾，很多算法尽管可以保证对任意输入都必然终止，但在终止之前所花费的时间成本太高。比如，理论研究的成果已几乎证明，大量问题求解的最低时间成本远远超出实际系统所能提供的计算能力。同样地，此类难解性(*intractability*)问题在本书中也不予过多讨论。

■ 计算效率

在“编写合法程序”这一基础之上，本书将更多地关注于非“不可解和难解”的一般性问题，并讨论如何高效率地解决这一层面的计算问题。为此，首先需要确立一种尺度，用以从时间和空间等方面度量算法的计算成本，进而依此尺度对不同算法进行比较和评判。当然，更重要的是研究和归纳算法设计与实现过程中的一般性规律与技巧，以编写出效率更高、能够处理更大规模数据的程序。这两点既是本书的基本主题，也是贯穿始终的主体脉络。

■ 数据结构

由上可知，无论是算法的初始输入、中间结果还是最终输出，在计算机中都可以数据的形式表示。对于数据的存储、组织、转移及变换等操作，不同计算模型和平台环境所支持的具体形式不尽相同，其执行效率将直接影响和决定算法的整体效率。数据结构这一学科正是以“数据”这一信息的表现形式为研究对象，旨在建立支持高效算法的数据信息处理策略、技巧与方法。要做到根据实际应用需求自如地设计、实现和选用适当的数据结构，必须首先对算法设计的技巧以及相应数据结构的特性了然于心，这些也是本书的重点与难点。

§ 1.2 复杂度度量

算法的计算成本涵盖诸多方面，为确定计算成本的度量标准，我们不妨先从计算速度这一主要因素入手。具体地，如何度量一个算法所需的计算时间呢？

1.2.1 时间复杂度

上述问题并不容易直接回答，原因在于，运行时间是由多种因素综合作用而决定的。首先，即使是同一算法，对于不同的输入所需的运行时间并不相同。以排序问题为例，输入序列的规模、其中各元素的数值以及次序均不确定，这些因素都将影响到排序算法最终的运行时间。为针对运行时间建立起一种可行、可信的评估标准，我们不得不首先考虑其中最为关键的因素。其中，问题实例的规模往往是决定计算成本的主要因素。一般地，问题规模越接近，相应的计算成本也越接近；而随着问题规模的扩大，计算成本通常也呈上升趋势。

如此，本节开头所提的问题即可转化为：随着输入规模的扩大，算法的执行时间将如何增长？执行时间的这一变化趋势可表示为输入规模的一个函数，称作该算法的时间复杂度(*time complexity*)。具体地，特定算法处理规模为 n 的问题所需的时间可记作 $T(n)$ 。

细心的读者可能注意到，根据规模并不能唯一确定具体的输入，规模相同的输入通常都有多个，而算法对其进行处理所需时间也不尽相同。仍以排序问题为例，由 n 个元素组成的输入序列有 $n!$ 种，有时所有元素都需交换，有时却无需任何交换(习题[3])。故严格说来，以上定义的 $T(n)$ 并不明确。为此需要再做一次简化，即从保守估计的角度出发，在规模为 n 的所有输入中选择执行时间最长者作为 $T(n)$ ，并以 $T(n)$ 度量该算法的时间复杂度。

1.2.2 渐进复杂度

至此，对于同一问题的两个算法 A 和 B，通过比较其时间复杂度 $T_A(n)$ 和 $T_B(n)$ ，即可评价二者对应于同一输入规模 n 的计算效率高低。然而，藉此还不足以对其性能优劣做出总体性的评判，比如对于某些问题，有的算法更适用于小规模输入，而另一些则相反（习题[5]）。

幸运的是，在评价算法运行效率时，我们往往倾向于忽略其处理小规模问题的能力，转而关注其在处理更大规模问题时的表现。其中的原因不难理解，小规模问题所需的处理时间相对更少，不同算法在效率方面的实际差异并不明显；而在处理更大规模的问题时，效率的些许差异都将对实际执行效果产生巨大的影响。这种着眼长远、更为注重时间复杂度的渐进变化趋势和增长速度的策略与方法，即所谓的渐进分析（**asymptotic analysis**）。因此，我们的问题可以进一步明确为：在输入规模 n 足够大之后，算法的执行时间 $T(n)$ 的渐进变化趋势和增长速度如何？

■ 大 \mathcal{O} 记号

同样处于保守的估计，我们首先关注 $T(n)$ 的渐进上界。为此可引入所谓“大 \mathcal{O} 记号”（**big- \mathcal{O} notation**）。具体地，若存在正的常数 c 和函数 $f(n)$ ，使得对任何 $n \gg 2$ 都有

$$T(n) \leq c \cdot f(n)$$

则可认为在 n 足够大之后， $f(n)$ 给出了 $T(n)$ 增长速度的一个渐进上界。此时，记之为

$$T(n) = \mathcal{O}(f(n))$$

由这一定义，可导出大 \mathcal{O} 记号的以下性质：

- (1) 对于任一常数 $c > 0$ ，有 $\mathcal{O}(f(n)) = \mathcal{O}(c \cdot f(n))$
- (2) 对于任意常数 $a > b > 0$ ，有 $\mathcal{O}(n^a + n^b) = \mathcal{O}(n^a)$

前一性质意味着，在大 \mathcal{O} 记号的意义下，函数各项正的常系数可以忽略并等同于 1。后一性质则意味着，多项式中的低次项均可忽略，只需保留最高次项。可以看出，大 \mathcal{O} 记号的这些性质的确体现了对函数总体渐进增长趋势的关注和刻画。

■ 环境差异

在实际环境中直接测得的执行时间 $T(n)$ ，虽不失为衡量算法性能的一种指标，但作为评判不同算法性能优劣的标准，其可信度值得推敲。事实上，即便是同一算法、同一输入，在不同的硬件平台上、不同的操作系统中甚至不同的时间，所需要的计算时间都不尽相同。因此，有必要按照超脱于具体硬件平台的某一客观标准，来度量时间复杂度并进而评价不同算法的效率差异。

■ 基本操作

一种自然且可行的解决办法是，将算法的时间复杂度理解为算法中各条指令语句所对应的执行时间之和。在通常的计算模型中，这些语句都可以再分解为若干次基本操作，比如算术运算、比较、分支、子程序调用与返回等；而且，在大多数实际的计算环境中，每一次这类基本操作都可在常数时间内完成。如此，不妨将 $T(n)$ 定义为算法所执行基本操作的总次数。也就是说， $T(n)$ 决定于组成算法的所有语句各自的执行次数，以及其中所含基本操作的数目。以代码 1.1 中起泡排序 **bubblesort()** 算法为例，若将该算法处理长度为 n 的序列所需的时间记作 $T(n)$ ，则按照上述分析，只需统计出该算法所执行基本操作的总次数，即可确定 $T(n)$ 的上界。

■ 起泡排序

`bubblesort()` 算法由内、外两层循环组成。内循环从前向后依次比较各对相邻元素的大小，如有必要则交换逆序的元素对。故在每一轮内循环中，需要扫描和比较 $n-1$ 对元素，至多需要交换 $n-1$ 对元素。无论元素的比较还是元素的交换都属于基本操作，故每一轮内循环至多需要执行 $2(n-1)$ 次基本操作。另外，根据 1.1.4 节对该算法正确性的分析结论，外循环至多执行 $n-1$ 轮。因此，总共需要执行的基本操作不会超过 $2(n-1)^2$ 次。若以此来度量该算法的时间复杂度，则有

$$T(n) = \mathcal{O}(2(n-1)^2)$$

根据大 \mathcal{O} 记号的性质，可进一步简化和整理为

$$T(n) = \mathcal{O}(2n^2 - 4n + 2) = \mathcal{O}(2n^2) = \mathcal{O}(n^2)$$

■ 最坏、最好与平均情况

由上可见，以大 \mathcal{O} 记号形式表示的时间复杂度，实质上是对算法执行时间的一种保守估计——对于规模为 n 的任意输入，算法的运行时间都不会超过 $\mathcal{O}(f(n))$ 。比如，“起泡排序算法复杂度 $T(n) = \mathcal{O}(n^2)$ ” 意味着，该算法处理任何序列所需的时间绝不会超过 $\mathcal{O}(n^2)$ 。的确需要这么长计算时间的输入实例，称作最坏实例或最坏情况（*worst case*）。

需强调的是，这种保守估计并不排斥更好情况甚至最好情况（*best case*）的存在和出现。比如，对于某些输入序列，起泡排序算法的内循环的执行轮数可能少于 $n-1$ ，甚至只需执行一轮（习题[3]）。当然，有时也需要考查所谓的平均情况（*average case*），也就是按照某种约定的概率分布，将规模为 n 的所有输入对应的计算时间加权平均。

比较而言，“最坏情况复杂度”是人们最为关注且使用最多的，在一些特殊的场合甚至成为唯一的指标。比如控制核电站运转、管理神经外科手术室现场的系统而言，从最好或平均角度评判算法的响应速度都不具有任何意义，在最坏情况下的响应速度才是唯一的指标。

■ 大Ω记号

为了对算法的复杂度最好情况做出估计，需要借助另一个记号。如果存在正的常数 c 和函数 $g(n)$ ，使得对于任何 $n \gg 2$ 都有

$$T(n) \geq c \cdot g(n)$$

就可以认为，在 n 足够大之后， $g(n)$ 给出了 $T(n)$ 的一个渐进下界。此时，我们记之为

$$T(n) = \Omega(g(n))$$

这里的 Ω 称作“大Ω记号”（*big-Ω notation*）。与大 \mathcal{O} 记号恰好相反，大Ω记号是对算法执行效率的乐观估计——对于规模为 n 的任意输入，算法的运行时间都不低于 $\Omega(g(n))$ 。比如，即便在最好情况下，起泡排序也至少需要 $T(n) = \Omega(n)$ 的计算时间（习题[4]）。

■ 大Θ记号

借助大 \mathcal{O} 记号、大Ω记号，可以对算法的时间复杂度作出定量的界定，亦即，从渐进的趋势看， $T(n)$ 介于 $\Omega(g(n))$ 与 $\mathcal{O}(f(n))$ 之间。若恰巧出现 $g(n) = f(n)$ 的情况，则可以使用另一记号来表示。

如果存在正的常数 $c_1 < c_2$ 和函数 $h(n)$ ，使得对于任何 $n \gg 2$ 都有

$$c_1 \cdot h(n) \leq T(n) \leq c_2 \cdot h(n)$$

就可以认为在 n 足够大之后， $h(n)$ 给出了 $T(n)$ 的一个确界。此时，我们记之为

$$T(n) = \Theta(h(n))$$

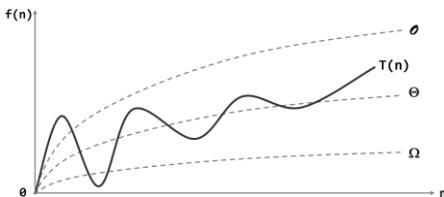


图1.4 大 O 记号、大 Ω 记号和大 Θ 记号

这里的 Θ 称作“大 Θ 记号”(big- Θ notation)，它是对算法复杂度的准确估计——对于规模为 n 的任何输入，算法的运行时间 $T(n)$ 都与 $\Theta(h(n))$ 同阶。

以上主要的这三种渐进复杂度记号，可形象地由图1.4示意。

1.2.3 空间复杂度

除了执行时间的长短，算法所需存储空间的多少也是衡量其性能的另一重要方面，此即所谓空间复杂度(space complexity)。自然地，对于同等规模的输入，在时间复杂度大体相当的前提下，算法所占用的空间越少越好。实际上，本节所引入的几种记号也可用以对空间复杂度的度量，原理及方法相同，不再赘述。

需指出的是，通常我们都更多地甚至仅仅关注于算法的时间复杂度，而多数时候不必对空间复杂度做专门分析。能够如此处理的依据来自于以下事实：就渐进复杂度而言，在任一算法的任何一次运行过程中，实际消耗的存储空间都不多于其间所执行基本操作的总次数。

实际上根据定义，每次基本操作所涉及的存储空间都不会超过常数规模，纵然每次基本操作所占用和访问的存储空间都是新开辟的，整个算法所需的空间总量也不过与基本操作的次数同价。从这个意义上说，时间复杂度本身就是空间复杂度的一个天然的上界。

当然，对空间复杂度的分析与比较也有其自身的意义，尤其在对空间效率非常在乎的应用场合，或者当问题的输入规模极为庞大时，这也会成为一个极为重要的指标。本书的后续章节，将结合一些实际问题介绍相关的方法与技巧。

§1.3 复杂度分析

在明确了算法复杂度的度量标准之后，具体地如何分析各种算法的复杂度呢？1.2.2节所引入的三种记号中，大 O 记号是最基本的，也是最常用到的。从渐进分析的角度，大 O 记号将各算法的复杂度由低到高划分为若干层次级别。以下依次介绍若干典型的复杂度级别，并介绍主要的分析方法与技巧。

1.3.1 常数 $O(1)$

■ 问题与算法

首先考查如下问题：任给整数子集 S ， $|S| = n \geq 3$ ，从中找出一个元素 $a \in S$ ，使得 $a \neq \max(S)$ 且 $a \neq \min(S)$ 。亦即，在最大、最小者之外任取一个元素，称作“非极端元素”或“平常元素”。这一问题可由算法1.3解决。

```
ordinaryElement(S[], n)
```

输入：由 n 个整数构成的集合 S ；输出：其中的任一非极端元素

任取的三个元素 $x, y, z \in S$ ；//既然 S 是集合，这三个元素必互异

通过比较对它们做排序；//设排序结果为： $\min\{x, y, z\}$, $\text{median}(x, y, z)$, $\max\{x, y, z\}$

输出 $\text{median}(x, y, z)$ ；

算法1.3 取非极端元素

■ 复杂度

算法 1.3 的正确性不言而喻，但它需要运行多少时间呢？与输入的规模 n 有何联系？

既然 S 是有限集，故其中的最大、最小元素各有且仅有一个。因此，无论 S 的规模有多大，在任意三个元素中至少都有一个是非极端元素。不妨取前三个元素 $x = S[0]$ 、 $y = S[1]$ 和 $z = S[2]$ ，这一步只需执行三次（从特定单元读取元素的）基本操作，耗费 $\mathcal{O}(3)$ 时间。接下来，为确定这三个元素的大小次序，最多需要做三次比较（习题[7]），也需 $\mathcal{O}(3)$ 时间。最后，输出居中的非极端元素只需 $\mathcal{O}(1)$ 时间。因此综合起来，算法 1.3 的运行时间为：

$$T(n) = \mathcal{O}(3) + \mathcal{O}(3) + \mathcal{O}(1) = \mathcal{O}(7) = \mathcal{O}(1)$$

也就是说，算法 1.3 具有常数量级的时间复杂度。

运行时间 $T(n) = \mathcal{O}(1)$ 的算法统称“常数时间复杂度算法”（constant-time algorithm）。此类算法已是较为理想的，因为不可能奢望更快的算法。一般地，仅含一次或常数次基本操作的算法（如算法 1.1 和算法 1.2）均属此类。尽管此类算法通常不含循环、分支、子程序调用（含递归）等，但也不能仅凭语法结构的表面形式一概而论（习题[8]）。

由 1.2.3 节的分析方法不难看出，除了输入数组等参数之外，该算法仅需常数规模的辅助空间。此类仅需 $\mathcal{O}(1)$ 辅助空间的算法，亦称作就地算法（in-place algorithm）。

1.3.2 对数 $\mathcal{O}(\log n)$

■ 问题与算法

试考查如下问题：给定任一非负整数，统计其对应二进制展开中数位 1 的总数。比如，5 的二进制展开为“101”，其中共有 2 个“1”；又如，441 的二进制展开为“110111001”，其中共有 6 个“1”。这一问题可由代码 1.2 中的 `countOnes()` 算法解决。

```
1 int countOnes(unsigned int n) { //统计整数n的二进制展开中数位1的总数 : O(logn)
2     int ones = 0; //计数器复位
3     while (0 < n) { //在n缩减至0之前，反复地
4         ones += (1 & n); //检查最低位，若为1则计数
5         n >>= 1; //右移一位
6     }
7     return ones; //返回计数
8 }
```

代码1.2 整数二进制展开中数位1总数的统计

该算法使用一个计数器 `ones` 记录数位 1 的数目，初始为 0。随后进入一个循环：通过二进制位的与运算，检查 n 的二进制展开的最低位，若该位为 1 则累计至 `ones`。由于每次循环都将 n 的二进制展开右移一位，故整体效果等同于逐个检验所有数位是否为 1，该算法的正确性由此得证。

以 $n = 441_{(10)} = 110111001_{(2)}$ 为例，按照以上算法 n 与 `ones` 的变化过程如表 1.1 所示。

表1.1 `countOnes(441)` 的执行过程

十进制	二进制	数位 1 计数
441	110111001	0
220	11011100	1
110	1101110	1
55	110111	1
27	11011	2
13	1101	3
6	110	4
3	11	4
1	1	5
0	0	6

■ 复杂度

根据右移运算的性质，每右移一位， n 都至少缩减一半。也就是说，至多经过 $1 + \lfloor \log_2 n \rfloor$ 次循环， n 必然缩减至 0，从而算法终止。实际上从另一角度来看， $1 + \lfloor \log_2 n \rfloor$ 恰为 n 二进制展开的总位数，每次循环都将其右移一位，总的循环次数自然也应是 $1 + \lfloor \log_2 n \rfloor$ 。后一解释，也可以从表 1.1 中 n 的二进制展开一列清晰地看出。

无论是该循环体之前、之内还是之后，均只涉及常数次（逻辑判断、位与运算、加法、右移等）基本操作。因此，`countOnes()` 算法的执行时间主要由循环的次数决定，亦即：

$$\mathcal{O}(1 + \lfloor \log_2 n \rfloor) = \mathcal{O}(\lfloor \log_2 n \rfloor) = \mathcal{O}(\log_2 n)$$

由大 \mathcal{O} 记号定义，在用函数 $\log_r n$ 界定渐进复杂度时，常底数 r 的具体取值无所谓（习题[9]），故通常不予专门标出而笼统地记作 $\log n$ 。比如，尽管此处底数为常数 2，却可直接记作 $\mathcal{O}(\log n)$ 。此类算法称“具有对数时间复杂度”（*logarithmic-time algorithm*）。

实际上，代码 1.2 中的 `countOnes()` 算法仍有巨大的改进余地（习题[13]）。

■ 对数多项式复杂度

更一般地，运行时间 $T(n) = \mathcal{O}(\log^c n)$ 的算法（常数 $c > 0$ ）统称“对数多项式时间复杂度的算法”（*polylogarithmic-time algorithm*）。上述 $\mathcal{O}(\log n)$ 即 $c = 1$ 的特例。此类算法的效率虽不如常数复杂度算法理想，但从多项式的角度看仍能无限接近于后者（习题[10]），故也是极为高效的一类算法。

1.3.3 线性 $\mathcal{O}(n)$

■ 问题与算法

考查如下问题：计算给定 n 个整数的总和。该问题可由代码 1.3 中的算法 `sumI()` 解决。

```
1 int sumI(int A[], int n) { //数组求和算法(迭代版)
2     int sum = 0; //初始化累计器, O(1)
3     for (int i = 0; i < n; i++) //对全部共O(n)个元素, 逐一
4         sum += A[i]; //累计, O(1)
5     return sum; //返回累计值, O(1)
6 } //O(1) + O(n)*O(1) + O(1) = O(n+2) = O(n)
```

代码1.3 数组元素求和算法sumI()

■ 复杂度

`sumI()` 算法的正确性一目了然，它需要运行多少时间呢？首先，对 s 的初始化需要 $\mathcal{O}(1)$ 时间。算法的主体部分是一个循环，每一轮循环中只需进行一次累加运算，这属于基本操作，可在 $\mathcal{O}(1)$ 时间内完成。每经过一轮循环，都将一个元素累加至 s ，故总共需要做 n 轮循环，于是该算法的运行时间应为：

$$\mathcal{O}(1) + \mathcal{O}(1) \times n = \mathcal{O}(n+1) = \mathcal{O}(n)$$

运行时间 $T(n) = \mathcal{O}(n)$ 的此类算法，统称“线性时间复杂度算法”（*linear-time algorithm*）。比如，算法 1.2 只需略加修改，即可解决“ n 等分给定线段”问题，此扩充版本相对于输入 n 就是一个线性时间复杂度的算法。若将执行时间平均划分，则可大致地理解为，此类算法对于输入的每一单元都消耗了常数时间。不难理解，对于大多数问题，在对输入的每一单元至少访问一次之前，往往不可能输出最终的解答。以这里的数组求和问题为

例，在未能确定数组中每一元素的具体数值之前，绝不可能确定其总和。就此意义而言，此类算法的效率亦完全足以令人满意。

1.3.4 多项式 $\mathcal{O}(\text{polynomial}(n))$

若运行时间 $T(n) = \mathcal{O}(f(n))$ 且 $f(x)$ 为多项式，则对应的算法统称“多项式时间复杂度算法”（**polynomial-time algorithm**）。比如根据 1.2.2 节的分析，1.1.3 节所实现起泡排序 **bubblesort()** 算法的时间复杂度为 $T(n) = \mathcal{O}(n^2)$ ，即属此类。当然，此前各线性时间复杂度的算法也属于多项式时间复杂度的特例，其中多项式 $f(n) = n$ 的次数为 1。

在算法复杂度理论中，多项式时间复杂度被视作具有特别意义的一个复杂度级别。一般地，某问题若存在一个复杂度在此界限以内的算法，则称该问题可以有效求解。请注意，这里仅仅要求多项式的次数为一个正的常数，而并未对其设置任何具体上限，故实际上该复杂度级别涵盖了很大的一类算法。比如，复杂度分别为 $\mathcal{O}(n^2)$ 和 $\mathcal{O}(n^{2011})$ 算法都同属此类，尽管二者实际的计算效率有天壤之别。之所以如此，是因为相对于以下的指数复杂度，二者之间不超过多项式规模的差异只不过是小巫见大巫。

1.3.5 指数 $\mathcal{O}(2^n)$

■ 问题与算法

考查如下问题：给定非负整数 n ，计算幂 2^n 。代码 1.4 给出了一个迭代式蛮力算法。

```
1 __int64 power2BF_I(int n) { //幂函数2^n算法(蛮力迭代版), n >= 0
2     __int64 pow = 1; //O(1):累积器初始化为2^0
3     while (0 < n --) //O(n):迭代n轮,每轮都
4         pow <= 1; //O(1):将累积器翻倍
5     return pow; //O(1):返回累积器
6 } //O(n) = O(2^r), r为输入指数n的比特位数
```

代码1.4 幂函数算法（蛮力迭代版）

■ 复杂度

算法 **power2BF_I()** 由 n 轮迭代组成，各需做一次累乘和一次递减，均属于基本操作，故整个算法共需 $\mathcal{O}(n)$ 时间。若以输入指数 n 的二进制位数 $r = 1 + \lfloor \log_2 n \rfloor$ 作为输入规模，则运行时间为 $\mathcal{O}(2^r)$ 。稍后在 1.4.3 节我们将看到，该算法仍有巨大的改进余地。

一般地，运行时间可表示为 $T(n) = \mathcal{O}(a^n)$ 的算法 ($a > 1$)，均属于“指数时间复杂度算法”（**exponential-time algorithm**）。

■ 从多项式到指数

从常数、对数、线性、平方到多项式时间复杂度，算法的效率不断下降，但就实际应用而言，这类算法的效率通常还在可接受的范围。然而，在多项式时间复杂度与指数时间复杂度之间，却有着一道巨大的鸿沟。当问题规模较大后，指数复杂度将急剧上升，计算时间之长很快就会达到令人难以忍受的地步。因此通常认为，指数复杂度算法无法应用于实际问题中，它们不是有效的算法，甚至不能称作算法。

需指出的是，在问题规模不大时，指数复杂度反而可能在较长一段区间内均低于多项式

复杂度。比如，在 $1 \leq n \leq 116690$ 以内，指数复杂度 1.0001^n 反而低于多项式复杂度 $n^{1.0001}$ ；但前者迟早必然超越后者，且随着 n 的进一步增大，二者的差距无法保持在多项式倍的范围。因此，从渐进复杂度的角度看，多项式与指数是无法等量齐观的两个截然不同的量级。

实际上，绝大多数计算问题并不存在多项式时间的算法，也就是说，试图求解此类问题的任一算法都至少需要运行指数量级的时间。特别地，很多问题甚至需要无穷的时间，由于有穷性不能满足，也可以说不存在解决这些问题的算法。这类问题均不属于本书的讨论范围。

1.3.6 复杂度层次

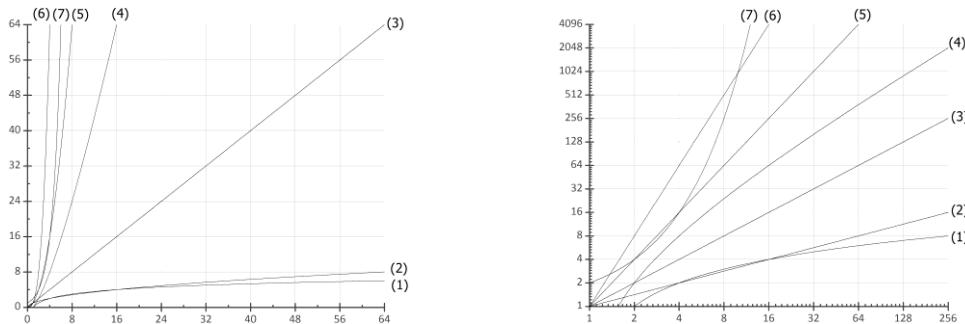


图1.5 复杂度的典型层次：(1)~(7)依次为 $\mathcal{O}(\log n)$ 、 $\mathcal{O}(\sqrt{n})$ 、 $\mathcal{O}(n)$ 、 $\mathcal{O}(n\log n)$ 、 $\mathcal{O}(n^2)$ 、 $\mathcal{O}(n^3)$ 和 $\mathcal{O}(2^n)$

利用大 \mathcal{O} 记号，不仅可以定量地把握算法复杂度的主要部分，而且可以定性地由低至高将复杂度划分为若干层次。典型的复杂度层次包括 $\mathcal{O}(1)$ 、 $\mathcal{O}(\log^* n)$ 、 $\mathcal{O}(\log\log n)$ 、 $\mathcal{O}(\log n)$ 、 $\mathcal{O}(\sqrt{n})$ 、 $\mathcal{O}(n)$ 、 $\mathcal{O}(n \log^* n)$ 、 $\mathcal{O}(n \log\log n)$ 、 $\mathcal{O}(n \log n)$ 、 $\mathcal{O}(n^2)$ 、 $\mathcal{O}(n^3)$ 、 $\mathcal{O}(n^c)$ 、 $\mathcal{O}(2^n)$ 等，图 1.5 绘出了其中七个层次复杂度函数对应的渐进增长趋势。

请注意，在图 1.5 的左图中，层次(7)的 2^n 显得比层次(6)的 n^3 更低，但这只是在问题规模 n 较小时的暂时现象。从覆盖更大范围的右图可以看出，当问题规模不小于 10 之后，层次(7)的复杂度将远远高于层次(6)。另外，右图还采用了双对数坐标，将层次(6)、(5)、(3)和(2)表示为直线，从而更为清晰地显示出各层次之间的高低关系。

1.3.7 输入规模

对算法复杂度的界定都是相对于问题的输入规模而言的，但细心的读者可能已注意到，关于“输入规模”的理解和定义可能不尽相同，因此也可能导致复杂度分析的结论有所差异。比如，1.3.2 节中“`countOnes()` 算法的复杂度为 $\mathcal{O}(\log n)$ ”的结论，是相对于输入整数本身数值 n 而言的；若以 n 二进制展开的宽度 $r = 1 + \lfloor \log_2 n \rfloor$ 作为输入规模，则应为线性复杂度 $\mathcal{O}(r)$ 。再如，1.3.5 节中“`power2BF_I()` 算法的复杂度为 $\mathcal{O}(2^r)$ ”的结论，是相对于输入指数 n 的二进制数位 r 而言的；若以 n 作为输入规模，却应为线性复杂度 $\mathcal{O}(n)$ 。

严格地说，待计算问题的输入规模应定义为“用以描述输入所需的空间的规模”。因此就上述两个例子而言，将输入参数 n 二进制展开的宽度 r 作为输入规模更为合理。也就是说，这两个算法的复杂度应界定为 $\mathcal{O}(r)$ 和 $\mathcal{O}(2^r)$ 。反过来，这种情况下以参数 n 为基准所得出的 $\mathcal{O}(\log n)$ 和 $\mathcal{O}(n)$ 复杂度，分别称作伪对数的（pseudo-logarithmic）和伪线性的（pseudo-linear）复杂度。

§ 1.4 *递归

递归是多数高级程序语言都支持的一个重要特性，它允许程序中的函数或过程自我调用。比如在 C++ 中，递归调用（**recursive call**）就是指某一方法调用自身。这种自我调用通常是直接的，即在函数体中包含一条或多条调用自身的语句。递归也可能是间接实现的，即某个方法首先调用其它方法，再辗转通过其它方法的相互调用，最终调用起始的方法自身。

递归的价值在于，许多应用问题都可简洁而准确地描述为递归形式。以操作系统为例，多数文件系统的目录结构都是递归定义的。具体地，每个文件系统都有一个最顶层的目录，其中可以包含若干文件和下一层的子目录；而在每一子目录中，也同样可能包含若干文件和再下一层的子目录；如此递推，直至不含任何下层的子目录。通过如此的递归定义，文件系统中的目录就可以根据实际应用的需要嵌套任意多层（只要系统的存储资源足以支持）。

递归也是一种基本而典型的算法设计模式。这一模式可以对实际问题中反复出现的结构和形式做高度概括，并从本质层面加以描述与刻画，进而导出高效的算法。从程序结构的角度看，递归模式能够统筹纷繁多变的具体情况，避免复杂的分支以及嵌套的循环，从而更为简明地描述和实现算法，减少代码量，提高算法的可读性，保证算法的整体效率。

以下将从递归的基本模式入手，循序渐进地介绍如何选择和应用（线性递归、二分递归和多分支递归等）不同的递归形式，以实现（遍历、分治等）算法策略，以及如何利用递归跟踪和递推方程等方法分析递归算法的复杂度。

1.4.1 线性递归

■ 数组求和

以下仍以 1.3.3 节数组求和问题为例，采用线性递归模式设计另一算法。首先注意到，若 $n = 0$ 则总和必为 0，这也是最终的平凡情况。否则一般地，数组的总和可理解为前 $n-1$ 个整数（即 $A[0, n-2]$ ）之和，再加上 $A[]$ 的最后一个元素（即 $A[n-1]$ ）。按这一思路，可设计出 **sum()** 算法如代码 1.5 所示。

```
1 int sum(int A[], int n) { //数组求和算法（线性递归版）
2     if (1 > n) //平凡情况，递归基
3         return 0; //直接（非递归式）计算
4     else //一般情况
5         return sum(A, n - 1) + A[n-1]; //递归：前n-1项之和，再累计第n-1项
6 } //O(1)*递归深度 = O(1)*(n+1) = O(n)
```

代码 1.5 数组求和算法（线性递归版）

由此实例可看出递归算法保证有穷性的基本技巧。具体地，首先必须判断并处理 $n = 0$ 之类的平凡情况，以免因无限递归而导致系统溢出。这类平凡情况统称“递归基”（**base case of recursion**）。可能有多种平凡情况，但至少要有一种，且这类情况迟早必出现。比如，算法 **sum()** 的递归基只包含一种情况，只需简单地判断 n 是否已经减小到 0。

■ 线性递归

算法 **sum()** 是通过在更深一层的自我调用来实现的，而且该函数的每一实例对自身的调用至多一次。于是，在每一层次上至多只有一个实例，且它们构成一个线性的次序关系。此类递归模式因而称作“线性递归”（**linear recursion**），它也是递归的最基本形式。

应用线性递归通常必须具备两个条件。其一，应用问题可分解为两个独立的子问题，一个对应于待处理数据中的某一特定元素，因而可以直接求解；另一个对应于剩余部分，且其结构与原问题雷同。在此 `sum()` 算法中，子问题分别为 $A[n-1]$ 和 $\text{sum}(A, n-1)$ 。其二，由于问题的解，可便捷地合并得到原问题的解。在这里，就是将两个子问题的解累加起来。

■ 减而治之

线性递归模式往往对应于所谓减而治之（*decrease-and-conquer*）的算法策略：递归每深入一层，待求解问题的规模都缩减一个常数，直至最终蜕化为平凡的小（简单）问题。

按照减而治之策略，此处随着递归的深入，调用参数将单调地线性递减。因此无论最初输入的 n 有多大，递归调用的总次数都是有限的，故算法的执行迟早会终止，即满足有穷性。当抵达递归基时，算法将执行非递归的计算（这里是返回 0 ）。

1.4.2 递归分析

递归算法时间和空间复杂度的分析与常规算法很不一样，有其自身的规律和特定的技巧，以下介绍递归跟踪与递推方程这两种主要的方法。

■ 递归跟踪

作为一种直观可视的方法，递归跟踪（*recursion trace*）可用以分析递归算法的总体运行时间与空间。具体地，就是按以下原则将递归算法的执行过程整理为图的形式：1) 算法的每一递归实例都表示为一个方框，其中注明了该实例调用的参数；2) 若实例 M 调用实例 N ，则在 M 与 N 对应的方框之间添加一条有向联线，指示二者之间的调用与被调用关系。

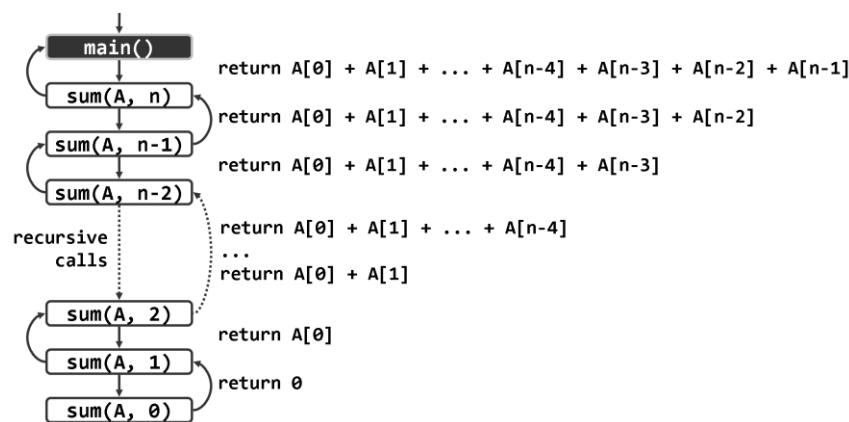


图 1.6 对 $\text{sum}(A, 5)$ 的递归跟踪分析

按上述约定，代码 1.5 中 `sum()` 算法的递归跟踪如图 1.6 所示。其中，`sum()` 算法的每一递归实例分别对应于一个方框，并标有相应的调用参数。每发生一次递归调用，就从当前实例向下引出一条有向边，指向子层对应于新实例的方框。

该图清晰地给出了算法执行的整个过程：首先对参数 n 进行调用，再转向对参数 $n-1$ 的调用，再转向对参数 $n-2$ 的调用，...，直至最终的参数 0 。在抵达递归基后不再递归，而是将平凡的解（长度为 0 数组的总和 0 ）返回给对参数 1 的调用；累加上 $A[0]$ 之后，再返回给对参数 2 的调用；累加上 $A[1]$ 之后，继续返回给对参数 3 的调用；...；如此依次返回，直到最终返回给对参数 n 的调用，此时，只需累加 $A[n-1]$ 即得到整个数组的总和。

从图 1.6 可清楚地看出，整个算法所需的计算时间，应该等于所有递归实例的创建、执行和销毁所需的时间总和。每一递归实例的创建、销毁均由操作系统负责完成，通常因为

近似为常数而可忽略。每个实例所需的执行时间中，递归调用语句所需的时间实际上已计入下层递归实例的账上，故只需统计各递归实例中非递归调用部分所需的时间。

具体就 `sum()` 算法而言，每一递归实例中非递归部分所涉及的计算无非三类（判断 n 是否为 0、累加 $sum(n-1)$ 与 $A[n-1]$ 、返回当前总和）且至多各执行一次。鉴于它们均属于常数时间的基本操作，每个递归实例实际所需计算时间应为常数 $O(3)$ 。由图 1.6 还可知，数组长度为 n 时，递归深度应为 $n+1$ ，故 `sum()` 算法共需运行 $(n+1) \times O(3) = O(n)$ 时间。

那么，`sum()` 算法的空间复杂度又是多少呢？由图 1.6 不难看出，在创建了最后一个递归实例（即到达递归基）时，占用的空间量达到最大——准确地说，等于所有递归实例各自所占空间量的总和。这里每一递归实例所需存放的数据，无非是调用参数（数组 A 的起始地址和长度 n ）以及用于累加总和的临时变量。这些数据各自只需常数规模的空间，其总量也应为常数。故此可知，`sum()` 算法的空间复杂度线性正比于其递归的深度，亦为 $O(n)$ 。

■ 递推方程

递归算法复杂度分析的另一常用方法，即所谓递推方程（recurrence equation）法。与递归跟踪分析相反，该方法无需绘出具体的调用过程，而是通过对递归模式的数学归纳，导出关于复杂度定界函数的递推方程（组）及其边界条件，从而将复杂度分析的任务转化为递归方程（组）的求解。在思路上，该方法与微分方程法颇为相似：很多复杂函数的显式表示通常不易直接获得，但是它们的微分形式却往往遵循一些相对简洁的方程，通过求解这组微分方程即可导出原函数的显式表示。通常，微分方程的解并不唯一，除非给定足够的边界条件。类似地，为使复杂度定界函数的递推方程能够给出确定的解，也需要给定某些边界条件——我们将看到，这一任务通常由递归基完成。

仍以代码 1.5 中线性递归版 `sum()` 算法为例，将该算法处理长度为 n 的数组所需的时间记作 $T(n)$ 。我们将该算法的思路重新表述如下：为解决问题 `sum(A, n)`，需递归地解决问题 `sum(A, n-1)`，然后累加上 $A[n-1]$ 。按照这一理解，求解 `sum(A, n)` 所需的时间，应等于求解 `sum(A, n-1)` 的时间，另加一次整数加法运算的时间。这可表示为如下递推关系：

$$T(n) = T(n-1) + O(1) = T(n-1) + c_1, \text{ 其中 } c_1 \text{ 为常数}$$

而抵达递归基时，求解平凡问题 `sum(A, 0)` 只需（用于直接返回 0 的）常数时间，即

$$T(0) = O(1) = c_2, \text{ } c_2 \text{ 为常数}$$

联立以上两个方程，可以解得：

$$T(n) = c_1 n + c_2 = O(n)$$

这一结论与递归跟踪分析殊途同归。

另外，运用以上方法，同样也可以得出 `sum()` 的空间复杂度（习题[19]）。

1.4.3 递归模式

■ 多递归基

为保证有穷性，递归算法必须设有递归基，且确保对应的语句总能执行到。实际上，针对算法中任一可能的平凡情况，都需设置对应的递归基，因此同一算法的递归基可能（显式或隐式地）不止一个。

下面来看一个数组倒置的实例。所谓“倒置”，就是将数组 $A[]$ 中 n 个元素的前后次序翻转。这里先介绍其算法的递归版本，1.4.4 节还将介绍等效的迭代版。无论哪种实现，均

由如下 `reverse()` 函数作为统一的启动入口。

```
1 void reverse(int*, int, int); //重载的倒置算法原型
2 void reverse(int* A, int n) //数组倒置(算法的初始入口, 调用的可能是reverse()的递归版或迭代版)
3 { reverse(A, 0, n - 1); } //由重载的入口启动递归或迭代算法
```

代码1.6 数组倒置算法的统一入口

借助线性递归不难解决这一问题，为此只需注意到并利用如下事实：为得到整个数组的倒置，可以先对换其首、末元素，然后递归地倒置除这两个元素以外的部分。算法的具体实现如代码 1.7 所示。

```
1 void reverse(int* A, int lo, int hi) { //数组倒置(多递归基递归版)
2     if (lo < hi) {
3         swap(A[lo], A[hi]); //交换A[lo]和A[hi]
4         reverse(A, lo + 1, hi - 1); //递归倒置A[lo+1..hi-1]
5     } //else隐含了两种递归基
6 } //O(hi - lo + 1)
```

代码1.7 数组倒置的递归算法

可见，每深入递归一层，待倒置区间的长度 $hi - lo + 1$ 都缩短 2 个单元。因此，所有递归实例所对应区间长度的奇偶性一致。需要特别留意的是，此处递归基实际上分为两种情况： $lo = hi$ （原数组长度为奇数）或 $lo = hi + 1$ （原数组长度为偶数）。当然，无论如何 `reverse()` 算法都必然会终止于这两种平凡情况之一，因此递归的深度应为：

$$\lceil (n + 1) / 2 \rceil = O(n)$$

在算法终止之前，递归每深入一层都会通过一次对换使得当前的 $A[lo]$ 与 $A[hi]$ 就位，因此该算法的时间复杂度也应线性正比于递归深度，即 $O(n)$ 。

■ 实现递归

在设计递归算法时，应从多个角度尝试，以确定对问题的输入及规模的最佳划分方式。有时，还可能需要重新定义和描述原问题，使得分解后的子问题与原问题具有相同的形式。例如，代码 1.7 线性递归版 `reverse()` 算法通过引入参数 `lo` 和 `hi`，使得对全数组以及其后各子数组的递归调用都统一为相同的语法形式。另外，还利用 C++ 的函数重载(`overload`)机制定义了名称相同、参数表有别的另一函数 `reverse(A, n)`，作为统一的初始入口。

■ 多向递归

在递归算法中，不仅递归基可能有多个，递归调用也可能有多种可供选择的分支。这里先介绍一个简单的例子，其中每一递归实例虽然有多个可能的递归方向，但只能从中选择其一。由于其中各层次的递归实例依然构成一个线性次序关系，这种情况依然属于线性递归。至于允许一个递归实例执行多次递归的情况，稍后将于 1.4.5 节再做介绍。

重新讨论 1.3.5 节的幂函数计算问题：计算 $\text{power}(2, n) = 2^n$ ，其中 n 为非负整数，其二进制展开表示由 $r = 1 + \lfloor \log_2 n \rfloor$ 个比特位组成。按照线性递归的构思，该函数可以重新定义和表述如下：

$$\text{power2}(n) = \begin{cases} 1 & \text{若 } n = 0 \\ 2 \cdot \text{power2}(n-1) & \text{否则} \end{cases}$$

由此不难直接导出一个线性递归的算法，其复杂度与代码 1.4 中蛮力的 `power2BF_I()`

算法完全一样，总共需要做 $\mathcal{O}(n)$ 次递归调用（习题[14]）。但实际上，若能从其它角度分析该函数并给出新的递归定义，完全可以更为快速地完成幂函数的计算。以下就是一例：

$$\text{power2}(n) = \begin{cases} 1 & (\text{若 } n = 0) \\ \text{power2}(\lfloor n/2 \rfloor)^2 \times 2 & (\text{若 } n > 0 \text{ 且为奇数}) \\ \text{power2}(\lfloor n/2 \rfloor)^2 & (\text{若 } n > 0 \text{ 且为偶数}) \end{cases}$$

按照这一新的表述和理解，可按二进制展开 n 之后的各比特位，通过反复的平方运算和加倍运算得到 $\text{power2}(n)$ 。比如：

$$\begin{aligned} 2^1 &= 2^{\text{001}}_{(2)} = (2^{\text{2}})^0 \times (2^{\text{2}})^0 \times 2^1 = (((1 \times 2^0)^2 \times 2^0)^2 \times 2^1) \\ 2^2 &= 2^{\text{010}}_{(2)} = (2^{\text{2}})^0 \times (2^{\text{2}})^1 \times 2^0 = (((1 \times 2^0)^2 \times 2^1)^2 \times 2^0) \\ 2^3 &= 2^{\text{011}}_{(2)} = (2^{\text{2}})^0 \times (2^{\text{2}})^1 \times 2^1 = (((1 \times 2^0)^2 \times 2^1)^2 \times 2^1) \\ 2^4 &= 2^{\text{100}}_{(2)} = (2^{\text{2}})^1 \times (2^{\text{2}})^0 \times 2^0 = (((1 \times 2^1)^2 \times 2^0)^2 \times 2^0) \\ 2^5 &= 2^{\text{101}}_{(2)} = (2^{\text{2}})^1 \times (2^{\text{2}})^0 \times 2^1 = (((1 \times 2^1)^2 \times 2^0)^2 \times 2^1) \\ 2^6 &= 2^{\text{110}}_{(2)} = (2^{\text{2}})^1 \times (2^{\text{2}})^1 \times 2^0 = (((1 \times 2^1)^2 \times 2^1)^2 \times 2^0) \\ 2^7 &= 2^{\text{111}}_{(2)} = (2^{\text{2}})^1 \times (2^{\text{2}})^1 \times 2^1 = (((1 \times 2^1)^2 \times 2^1)^2 \times 2^1) \\ \dots & \end{aligned}$$

一般地，若 n 的二进制展开式为 $b_1b_2b_3\dots b_k$ ，则有

$$2^n = (\dots(((1 \times 2^{b1})^2 \times 2^{b2})^2 \times 2^{b3})^2 \dots \times 2^{bk})$$

若 n_{k-1} 和 n_k 的二进制展开式分别为 $b_1b_2\dots b_{k-1}$ 和 $b_1b_2\dots b_{k-1}b_k$ ，则有

$$2^{n_k} = (2^{n_{k-1}})^2 \times 2^{bk}$$

由此可以归纳得出如下递推式：

$$\text{power2}(n_k) = \begin{cases} \text{power2}(n_{k-1})^2 \times 2 & (\text{若 } b_k = 1) \\ \text{power2}(n_{k-1})^2 & (\text{若 } b_k = 0) \end{cases}$$

基于这一递推式，即可如代码 1.8 所示实现幂函数的多向递归版本 $\text{power2}()$ ：

```
1 inline __int64 sqr(__int64 a) { return a * a; }
2 __int64 power2(int n) { //幂函数2^n算法（优化递归版），n >= 0
3     if (0 == n) return 1; //递归基
4     return (n & 1) ? sqr(power2(n >> 1)) << 1 : sqr(power2(n >> 1)); //视n的奇偶分别递归
5 } //O(logn) = O(r), r为输入指数n的比特位数
```

代码1.8 优化的幂函数算法（线性递归版）

请注意，尽管其中有（奇、偶指数）两种可能的递归方向，但每个递归实例都只能沿其中之一深入到下层递归，故依然属于线性递归。

由其递归跟踪分析图可见，递归每深入一层，指数 n 都会递减至少一半。这意味着，问题规模将按几何级数的速率递减，故经过 $\mathcal{O}(\log n)$ 次递归调用即可抵达递归基；此后，再经过同等次数的算术运算与递归返回，即可得到最终的计算结果。另一方面，每一递归实例中的非递归操作仅需常数时间，故改进之后 $\text{power2}()$ 算法的时间复杂度为：

$$\mathcal{O}(\log n) \times \mathcal{O}(1) = \mathcal{O}(r)$$

与此前代码 1.4 中蛮力版本的 $\mathcal{O}(n) = \mathcal{O}(2^r)$ 相比，计算效率得到了极大提高。

1.4.4 递归消除

由上可见，按照递归的思想可使我们得以从宏观上理解和把握应用问题的实质，深入挖掘和洞悉算法过程的主要矛盾和一般性模式，并最终设计和编写出简洁优美且精确紧凑的算法。然而，递归模式并非十全十美，其众多优点的背后也隐含着某些代价。

■ 空间成本

比如，较之同一算法的迭代式版本，递归版本往往需要耗费更多的空间，并进而影响到实际的运行速度。从递归跟踪分析的角度不难看出，递归算法所消耗的空间量主要取决于递归的最大深度（习题[18]），因此在需要深入递归多层时，空间复杂度将急剧攀升。另外就操作系统而言，与通常函数调用的机制一样，为实现递归调用也需花费大量额外的时间创建、维护和销毁各递归实例，这些也会令计算负担雪上加霜。有鉴于此，在对运行速度要求极高、存储空间需精打细算的场合，往往应将算法的递归版本改写成等价的非递归版本。

一般的转换思路，无非是利用第4章将要介绍的栈结构模拟操作系统的工作过程，从而将递归算法转换为迭代版本。这类的通用方法已超出本书的范围，这里仅结合一种简单而常见的情况略作介绍。

■ 尾递归及其消除

在线性递归算法中，若递归调用恰好出现在最后一步操作，则称之为尾递归（tail recursion）。比如代码1.7中reverse(A, lo, hi)算法的最后一步操作，是对去除了首、末元素之后总长缩减两个单元的子数组进行递归倒置，即属于典型的尾递归。实际上，属于尾递归形式的算法，均可以简捷地转换为等效的迭代版本。

仍以代码1.7中reverse(A, lo, hi)算法为例。如代码1.9所示，首先在起始位置插入一个跳转标志next，然后将尾递归语句调用替换为一条指向next标志的跳转语句。

```
1 void reverse(int* A, int lo, int hi) { //数组倒置(直接改造而得的迭代版)
2     next: //算法起始位置添加跳转标志
3     if (lo < hi) {
4         swap(A[lo], A[hi]); //交换A[lo]和A[hi]
5         lo++; hi--; //收缩待倒置区间
6         goto next; //跳转至算法体的起始位置，迭代地倒置A[lo+1, hi-1]
7     } //else隐含了迭代的终止
8 } //O(hi - lo + 1)
```

代码1.9 由递归版改造而得的数组倒置算法(迭代版)

新的迭代版与原递归版功能等效，但其中使用的goto语句有悖于结构化程序设计的原则。这一语句虽仍不得不被C++等高级语言保留，但最好还是尽力回避。为此可如代码1.10所示，将next标志与if判断综合考查，并代之以一条逻辑条件等价的while语句。

```
1 void reverse(int* A, int lo, int hi) { //数组倒置(规范整理之后的迭代版)
2     while (lo < hi) //用while替换跳转标志和if，完全等效
3         swap(A[lo++], A[hi--]); //交换A[lo]和A[hi]，收缩待倒置区间
4 } //O(hi - lo + 1)
```

代码1.10 进一步调整代码1.9的结构，消除goto语句

请注意，尾递归的判断应依据对算法实际执行过程的分析，而不仅仅是算法外在的语法

形式。比如，递归语句出现在代码体的最后一行，并不见得就是尾递归；严格地说，只有当该算法（除平凡递归基外）任一实例都终止于这一递归调用时，才属于尾递归。以代码 1.5 中线性递归版 `sum()` 算法为例，尽管从表面看似乎最后一行是递归调用，但实际上却并非尾递归——实质的最后一次操作是加法运算。有趣的是，此类算法的非递归化转换方法仍与尾递归如出一辙，相信读者不难将其改写为类似于代码 1.3 中 `sumI()` 算法的迭代版本。

1.4.5 二分递归

■ 分而治之

面对输入规模庞大的应用问题，每每感慨于头绪纷杂而无从下手的你，不妨从先哲孙子的名言中获取灵感——“凡治众如治寡，分数是也”。是的，解决此类问题的有效方法之一，就是将其分解为若干规模更小的子问题，再通过递归机制分别求解。这种分解持续进行，直到子问题规模缩减至平凡情况。这也就是所谓的分而治之（divide-and-conquer）策略。

当然，与减而治之策略一样，这里也要求从形式上对原问题重新描述，以保证子问题与原问题在接口形式上的一致。相应地，每一递归实例都有可能做多次深入递归，故这种模式称作“多路递归”（multi-way recursion）。通常都是将大型问题一分为二，故称作“二分递归”（binary recursion）。需强调的是，无论是分解为两个还是更大常数个子问题，对算法总体的渐进复杂度并无实质影响。

■ 数组求和

以下就采用分而治之的策略，按照二分递归的模式再次解决数组求和问题。新算法的思路是：以居中的元素为界将数组一分为二；递归地对子数组分别求和；最后，子数组之和相加即为原数组的总和。具体过程可描述如代码 1.11，算法入口的调用形式为 `sum(A, 0, n)`。

```
1 int sum(int A[], int lo, int hi) { //数组求和算法（二分递归版，入口为sum(A, 0, n-1)）
2     if (lo == hi) //如遇递归基（区间长度已降至1），则
3         return A[lo]; //直接返回该元素
4     else { //否则（一般情况下lo < hi），则
5         int mi = (lo + hi) >> 1; //以居中单元为界，将原区间一分为二
6         return sum(A, lo, mi) + sum(A, mi + 1, hi); //递归对各子数组求和，然后合计
7     }
8 } //O(hi-lo+1)，线性正比于区间的长度
```

代码1.11 通过二分递归计算数组元素之和

该算法的正确性无需解释。为分析其复杂度，不妨只考察 $n = 2^m$ 形式的长度。

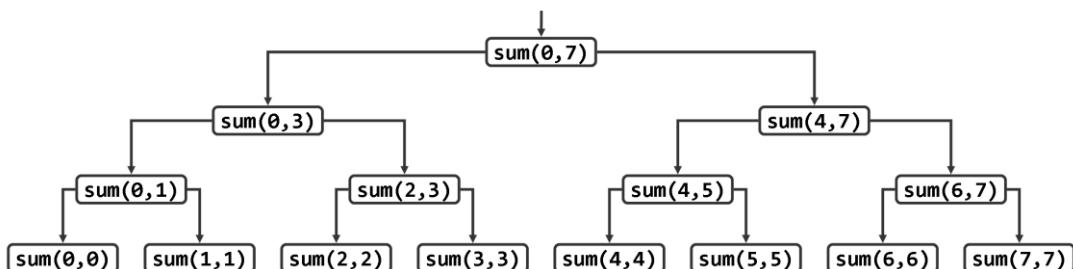


图1.7 对`sum(A, 0, 7)`的递归跟踪分析

图 1.7 针对 $n = 8$ 的情况给出了 $\text{sum}(A, 0, 7)$ 执行过程的递归跟踪。其中，各方框都标注有对应的 lo 和 hi 值，即子数组区间的起、止单元。可见，按照调用的关系及次序，该方法的所有实例构成一个层次结构（即第 5 章将要介绍的树形结构）。沿着这个层次结构每下降一层，每个递归实例 $\text{sum}(lo, hi)$ 都分裂为一对更小的实例 $\text{sum}(lo, mi)$ 和 $\text{sum}(mi+1, hi)$ ——准确地说，每经过一次递归调用，数组区间的长度 $hi - lo + 1$ 都将减半。

算法启动后经连续 $m = \log_2 n$ 次递归调用，数组区间的长度从最初的 n 首次缩减至 1，并到达第一个递归基。实际上，刚到达任一递归基时，已执行的递归调用总是比递归返回多 $m = \log_2 n$ 次。更一般地，到达区间长度为 2^k 的任一递归实例之前，已执行的递归调用总是比递归返回多 $m-k$ 次。因此，递归深度（即任一时刻的活跃递归实例的总数）不会超过 $m+1$ 。鉴于每个递归实例仅需常数空间，故除数组本身所占的空间，该算法只需要 $O(m+1) = O(\log n)$ 的附加空间。我们还记得，代码 1.5 中线性递归版 $\text{sum}()$ 算法共需 $O(n)$ 的附加空间，就这一点而言，新的二分递归版 $\text{sum}()$ 算法有很大改进。

与线性递归版 $\text{sum}()$ 算法一样，此处每一递归实例中的非递归计算都只需要常数时间。递归实例共计 $2n-1$ 个，故新算法的运行时间为 $O(2n-1) = O(n)$ ，与线性递归版相同。

此处每个递归实例可向下深入递归两次，故属于多路递归中的二分递归。二分递归与此前介绍的线性递归有很大区别。比如，在线性递归中整个计算过程仅出现一次递归基，而在二分递归过程中递归基的出现相当频繁，总体而言有超过半数的递归实例都是递归基。

■ 效率

当然，并非所有问题都适宜于采用分治策略。实际上除了递归，此类算法的计算消耗主要来自两个方面。首先是子问题划分，即把原问题分解为形式相同、规模更小的多个子问题，比如代码 1.11 中 $\text{sum}()$ 算法将待求和数组分为前、后两段。其次是子解答合并，即由递归所得子问题的解，得到原问题的整体解，比如由子数组之和累加得到整个数组之和。

为使分治策略真正有效，不仅必须保证以上两方面的计算都能高效地实现，还必须保证子问题之间相互独立——各子问题可独立求解，而无需借助其它子问题的原始数据或中间结果。否则，或者子问题之间必须传递数据，或者子问题之间需要相互调用，无论如何都会导致时间和空间复杂度的无谓增加。以下就以 Fibonacci 数列的计算为例说明这一点。

■ Fibonacci 数：二分递归

考查 Fibonacci 数列第 n 项 $\text{fib}(n)$ 的计算问题，该数列递归形式的定义如下：

$$\text{fib}(n) = \begin{cases} n & (\text{若 } n \leq 1) \\ \text{fib}(n-1) + \text{fib}(n-2) & (\text{若 } n \geq 2) \end{cases}$$

据此定义，可直接导出如代码 1.12 所示的二分递归版 $\text{fib}()$ 算法：

```
1 __int64 fib(int n) { //计算Fibonacci数列的第n项(二分递归版) : O(2^n)
2     return (2 > n) ?
3         (__int64)n //若到达递归基, 直接取值
4         : fib(n - 1) + fib(n - 2); //否则, 递归计算前两项, 其和即为正解
5 }
```

代码1.12 通过二分递归计算Fibonacci数

基于 Fibonacci 数列原始定义的这一实现，不仅正确性一目了然，而且简洁自然。然而不幸的是，在这种场合采用二分递归策略的效率极其低下。实际上，该算法需要运行 $O(2^n)$

时间才能计算出第 n 个 Fibonacci 数。这一指数复杂度的算法，在实际环境中毫无价值。

为确切地界定该算法的复杂度，不妨将计算 $\text{fib}(n)$ 所需的时间记作 $T(n)$ 。按该算法的思路，为计算出 $\text{fib}(n)$ ，先花费 $T(n-1)$ 时间计算出 $\text{fib}(n-1)$ ，再花费 $T(n-2)$ 时间计算 $\text{fib}(n-2)$ ，最后花费一个单位的时间将它们累加起来。由此，可得 $T(n)$ 的递推式如下：

$$T(n) = \begin{cases} 1 & (\text{若 } n \leq 1) \\ T(n-1) + T(n-2) + 1 & (\text{否则}) \end{cases}$$

若令 $S(n) = [T(n) + 1]/2$ ，则有

$$S(n) = \begin{cases} 1 & (\text{若 } n \leq 1) \\ S(n-1) + S(n-2) & (\text{否则}) \end{cases}$$

我们发现， $S(n)$ 的递推形式与 $\text{fib}(n)$ 完全一致，只是起始项不同：

$$S(0) = (T(0)+1)/2 = 1 = \text{fib}(1)$$

$$S(1) = (T(1)+1)/2 = 1 = \text{fib}(2)$$

亦即， $S(n)$ 整体上相对于 $\text{fib}(n)$ 提前了一个单元。由此可知：

$$S(n) = \text{fib}(n+1) = (\Phi^{n+1} - \bar{\Phi}^{n+1})/\sqrt{5}, \quad \Phi = (1+\sqrt{5})/2, \quad \bar{\Phi} = (1-\sqrt{5})/2$$

$$T(n) = 2 \cdot S(n) - 1 = 2 \cdot \text{fib}(n+1) - 1 = \mathcal{O}(\Phi^{n+1}) = \mathcal{O}(2^n)$$

实际上，这一版本 $\text{fib}()$ 算法的空间复杂度也高达指数量级。究其原因在于，计算过程中所出现递归实例的重复度极高（习题[20]）。只需画出递归跟踪分析图的前几层，即不难验证这一点；若需更为精确地界定，可借助递推方程。

■ Fibonacci 数：线性递归

以上之所以首先采用二分递归模式实现 $\text{fib}()$ 算法，是受到该问题原始定义的表面特征—— $\text{fib}(n)$ 由 $\text{fib}(n-1)$ 和 $\text{fib}(n-2)$ 共同决定——的误导。然而进一步分析可见， $\text{fib}(n-1)$ 和 $\text{fib}(n-2)$ 并非相互独立的子问题，故该问题的本质应是线性递归而非二分递归。

当然，为了应用线性递归的模式，首先需从改造 Fibonacci 数的递归定义入手。比如，可定义并使用另一个递归函数，以计算一对相邻的 Fibonacci 数 ($\text{fib}(n-1)$, $\text{fib}(n)$)。若约定 $\text{fib}(-1) = 1$ ，则可得到如代码 1.13 所示的线性递归版 $\text{fib}()$ 算法。

```
1 __int64 fib(int n, __int64& prev) { //计算Fibonacci数列第n项（线性递归版）：入口形式fib(n, pred)
2     if (0 == n) //若到达递归基，则
3         { prev = 1; return 0; } //直接取值：fib(-1) = 1, fib(0) = 0
4     else { //否则
5         __int64 prevPrev; prev = fib(n - 1, prevPrev); //递归计算前两项
6         return prevPrev + prev; //其和即为正解
7     }
8 } //用辅助变量记录前一项，返回数列的当前项，O(n)
```

代码1.13 通过线性递归计算Fibonacci数

以上算法的结构明显呈现线性递归的特征：每个递归实例至多向下递归一次。另外，递归每深入一层，参数 n 都会减一，直至 $n = 0$ 时到达递归基。故即使计入最顶层，先后出现的递归实例总共也不过 $n + 1$ 个。鉴于每个递归实例中的非递归计算均只需常数时间，该算法总体的时间复杂度应为 $\mathcal{O}(n)$ 。无论就复杂度层次还是实际运行速度而言，较之该算

法的二分递归版本都有巨大的改进（习题[22]）。

■ Fibonacci 数：动态规划

以上线性递归版 `fib()` 算法，尽管在执行过程中不再会出现雷同的递归实例，但仍会因递归深度线性正比于输入 n 而需要使用 $\mathcal{O}(n)$ 规模的附加空间。通常，为部分或完全消除算法中的递归成分，往往可以采用动态规划（dynamic programming）策略。其基本技巧之一是，借助很少量的辅助空间，在计算过程中记录下业已处理过的子问题的解答。这样，此后一旦再次遇到相同的子问题，即可通过查阅记录直接获得结果而不必重新计算。

按照动态规划的思路，可将以上线性递归版 `fib()` 算法改写为代码 1.14 中的迭代版。

```
1 __int64 fibI(int n) { //计算Fibonacci数列的第n项(迭代版): O(n)
2     __int64 f = 1, g = 0; //初始化: fib(-1)=1, fib(0)=0
3     while (0 < n--) { f = f + g; g = f - g; } //依据原始定义, 通过n次加法和减法计算fib(n)
4     return g; //返回
5 }
```

代码1.14 基于动态规划策略计算Fibonacci数

这里借助中间变量 `f` 和 `g` 记录一对相邻的 Fibonacci 数；以迭代形式按该数列的原始定义不断更新其数值，直到得出所需的 `fib(n)`。如此，整个算法仅需线性次迭代，时间复杂度为 $\mathcal{O}(n)$ 。更重要的是，该版本仅需常数规模的附加空间，空间效率也有极大提高。

§ 1.5 抽象数据类型

各种数据结构都可看作是由若干数据项组成的集合，同时对数据项定义一组标准的操作。现代数据结构普遍遵从“信息隐藏”的理念，通过统一接口和内部封装，分层次从整体上加以设计、实现与使用。

所谓封装，就是将数据项与相关的操作结合为一个整体，并将其从外部的可见性划分为若干级别，从而将数据结构的外部特性与其内部实现相分离，提供一致且标准的对外接口，隐藏内部的实现细节。于是，数据集合及其对应的操作可超脱于具体的程序设计语言、具体的实现方式，即构成所谓的抽象数据类型（abstract data type, ADT）。抽象数据类型的理论催生了现代面向对象的程序设计语言，而支持封装也是此类语言的基本特征。

本书将尽可能遵循抽象数据类型的规范来设计、实现并分析各种数据结构。具体地，将从各数据结构的对外功能接口（interface）出发，以 C++ 语言为例逐层讲解其内部具体实现（implementation）的原理、方法与技巧，并就不同实现方式的效率及适用范围进行分析与比较。为体现数据结构的通用性，也将普遍采用模板类的描述模式。

习题

- [1] 试以基本的几何作图操作描述一个算法过程，实现“过直线外一点作其平行线”的功能。
- [2] 《海岛算经》讨论了遥测海岛高度的问题：今有望海岛，立两表，齐高三丈，前后相去千步，令后表与前表参相直。从前表却行一百二十三步，人目著地取望岛峰，与表末参合。从后表却行一百二十七步，人目著地取望岛峰，亦与表末参合。问岛高及去表各几何？刘徽给出的解法是：以表高乘表间为实；相多为法，除之。所得加表高，即得岛高。
 - a) 该算法的原理是什么？b) 试以伪代码形式描述该算法的过程。c) 该算法借助了哪些计算工具？

作者保留所有版权，未经授权不得转载 谢绝各种网络电子文库

- [3] 试分别举出实例说明，在对包含 n 个元素的序列做起泡排序的过程中，可能发生以下情况：
a) 任何元素都无需移动（从而内循环仅执行一轮即可终止算法）；b) 某元素有时会逆向移动；
c) 某元素与自己应处的位置相邻，却需要参与 $n-1$ 次交换；d) 所有元素都需要参与 $n-1$ 次交换。
- [4] 对 n 个整数的排序，能否在少于 $\mathcal{O}(n)$ 的时间内完成？为什么？
- [5] 随着问题输入规模的扩大，同一算法所需的计算时间通常都呈单调递增趋势，但情况也并非总是如此。试举实例说明，随着输入规模的扩大，同一算法所需的计算时间可能
a) 呈波动形式增加；b) 呈波动形式稳定不变；甚至 c) 呈波动形式不确定。
- [6] 在一台速度为 1G flops 的电脑上使用代码 1.1 中的 bubblesort() 算法，大致需要多长时间才能完成对全国人口记录的排序？
- [7] 算法 1.3 中，在选出三个数之后还需对它们做排序。试证明：
a) 至多只需比对元素的大小三次，即可完成排序；
b) 在最坏情况下，的确至少需要比对元素的大小三次，才能完成排序。
- [8] 试用 C++ 语言描述一个包含循环、分支、子函数调用甚至递归结构的算法，要求具有常数时间复杂度。
- [9] 试证明，在用对数函数界定渐进复杂度时，常底数的具体取值无所谓。
- [10] 试证明，对于任何 $\epsilon > 0$ ，都有 $\log n = \mathcal{O}(n^\epsilon)$ 。
- [11] 试证明，在大 \mathcal{O} 记号的意义下
a) 等差级数之和与其中最大一项的平方同阶；b) 等比级数之和与其中最大一项同阶。
- [12] 若 $f(n) = \mathcal{O}(n^2)$ 且 $g(n) = \mathcal{O}(n)$ ，则以下结论是否正确：
a) $f(n) + g(n) = \mathcal{O}(n^2)$ ；b) $f(n)/g(n) = \mathcal{O}(n)$ ；c) $g(n) = \mathcal{O}(f(n))$ ；d) $f(n)*g(n) = \mathcal{O}(n^3)$
- [13] 改进代码 1.2 中 countOnes() 算法，使得时间复杂度降至
a) $\mathcal{O}(\text{countOnes}(n))$ ，线性正比于数位 1 的实际数目；b) $\mathcal{O}(\log_2 W)$ ， $W = \mathcal{O}(\log_2 n)$ 为整数的位宽。
比如，目前环境中 `unsigned int` 位宽多为 $W = 32$ 位，于是 $\mathcal{O}(\log_2 32) = \mathcal{O}(1)$ ，仅需常数时间！
- [14] 实现代码 1.4 的递归版，要求时间复杂度保持为 $\mathcal{O}(r) = \mathcal{O}(2^n)$ 。
- [15] 实现代码 1.8 中 power2() 算法的迭代版，要求时间复杂度保持为 $\mathcal{O}(\log r) = \mathcal{O}(n)$ 。
- [16] 考查最大元素问题：从 n 个整数中找出最大者。
a) 试分别采用迭代和递归两种模式设计算法，在线性时间内解决该问题；
b) 用 C++ 语言实现你的算法，并分析它们的复杂度。
- [17] 考查如下问题：设 S 为一组共 n 个正整数，其总和为 $2m$ ，判断是否可将 S 划分为两个不相交的子集，且各自总和均为 m ？美国总统选举即是该问题的一个具体实例。若仅有两位候选人参赛并争夺 $n = 51$ 个选举人团（50 个州加 1 个特区）的共计 $2m = 538$ 张选举人票，是否可能因两人各得 $m = 269$ 张而不得不重新选举？
a) 设计并实现一个对应的算法，时间复杂度为 $\mathcal{O}(2^n)$ ；
b) 若对整数范围不做任何限定，该问题可在多项式时间内求解？
- [18] 证明：若各递归实例仅需常数规模的空间，则递归算法所需空间总量线性正比于递归跟踪树的深度。
- [19] 试采用递推方程法，分析代码 1.5 中线性递归版 sum() 算法的空间复杂度。
- [20] 考查如代码 1.12 所示的二分递归版 fib(n) 算法。
a) 算法过程中，对任 $1 \leq k \leq n$ ，形如 $\text{fib}(k)$ 的递归实例会先后重复出现 $\text{fib}(n-k+1)$ 次；
b) 该算法的空间和时间复杂度均为指数量级。

[21] 考查 Fibonacci 数的计算。

a) 试证明，任意算法哪怕只是直接打印输出 $\text{fib}(n)$ ，也至少需要 $\Omega(n)$ 的时间

(提示：只需证明，无论以任何常数为进制， $\text{fib}(n)$ 均由 $\Theta(n)$ 个数位组成)；

b) 试参考代码 1.8 中 $\text{power2}()$ 算法设计一个算法，在 $O(\log n)$ 时间内计算出 $\text{fib}(n)$ ；

c) 以上结论是否矛盾？为什么？

[22] 考查 $\text{fib}()$ 算法的二分递归版、线性递归版和迭代版。

a) 分别编译这些算法，针对 $n = 64$ 实际运行并测试对比；b) 三者的运行速度有何差别？为什么？

[23] 参照代码 1.14 中迭代版 $\text{fib}()$ 算法，实现支持如下接口的 Fib 类，注意时间复杂度。

```
class Fib {  
public:  
    Fib(int n); // 初始化为不小于n的最小fib()项(比如Fib(6)将初始化为8)，O(logφ(n))时间  
    int get(); // 获取当前项(比如8)，O(1)时间  
    int next(); // 转至下一项(若当前为8，则转至13)，O(1)时间  
    int prev(); // 获取并转至后一项(若当前为8，则转至5)，O(1)时间  
};
```

[24] 《九章算术》记载的“中华更相减损术”可快速地计算正整数 a 和 b 的最大公约数，其计算过程如下：

- | | |
|--|--|
| 01) 令 $p = 1$ | 07) 若 t 为奇数，则转 10) |
| 02) 若 a 和 b 不都是偶数，则转 5) | 08) 令 $t = t/2$ |
| 03) 令 $p = p \times 2$, $a = a/2$, $b = b/2$ | 09) 转 7) |
| 04) 转 2) | 10) 若 $a \geq b$ ，则令 $a = t$ ；否则，令 $b = t$ |
| 05) 令 $t = a - b $ | 11) 转 5) |
| 06) 若 $t = 0$ ，则返回并输出 $a \times p$ | |

a) 按照上述流程，编写一个算法 $\text{int gcd(int a, int b)}$ ，计算 a 和 b 的最大公约数；

b) 与功能相同的欧几里得算法相比，这一算法有何优势？

[25] 法国数学家 Edouard Lucas 于 1883 提出的 Hanoi 塔问题可形象地描述如下：有 n 个中心带孔的圆盘贯穿在直立于地面的一根柱子上，各圆盘的半径自底而上不断缩小；需要利用另一根柱子将它们转运至第三根柱子，但在整个转运的过程中，游离于这些柱子之外的圆盘不得超过一个，且每根柱子上的圆盘半径都须保持上小下大。试将转运过程描述为递归形式，并进而实现一个递归算法。

[26] 试运用递归跟踪法，分析代码 1.7 中 $\text{reverse}()$ 算法和代码 1.8 中 $\text{power2}()$ 算法的时间复杂度。

[27] 是设计并实现一个就地算法 $\text{shift}(\text{int } A[], \text{int } n, \text{int } k)$ ，在 $O(n)$ 时间内将任一数组 $A[0, n)$ 中的元素整体循环左移 k 位。例如若 $A[] = \{1, 2, 3, 4, 5, 6\}$ ，则经 $\text{shift}(A, 6, 2)$ 后 $A[] = \{3, 4, 5, 6, 1, 2\}$ 。提示：利用代码 1.7 中 $\text{reverse}()$ 算法。

[28] 试实现一个递归算法，对任意非负整数 m 和 n ，计算以下 Ackermann 函数值：

$$\text{Ackermann}(m, n) = \begin{cases} n + 1 & (\text{若 } m = 0) \\ \text{A}(m - 1, 1) & (\text{若 } m > 0 \text{ 且 } n = 0) \\ \text{A}(m - 1, \text{A}(m, n - 1)) & (\text{若 } m > 0 \text{ 且 } n > 0) \end{cases}$$

对于每一 (m, n) 组合，这个算法是否必然终止？

[29] 考查所谓咖啡罐游戏 (Coffee Can Game)：在咖啡罐中放有 n 颗黑豆与 m 颗白豆，每次取出两颗：若同色，则扔掉它们，然后放入一颗黑豆；若异色，则扔掉黑豆，放回白豆。

- a) 试证明该游戏必然终止（当罐中仅剩一颗豆子时）；
b) 对于哪些(n, m)的组合，最后剩下的必是白豆？

[30] 序列 Hailstone(n)是从 n 开始，按照以下规则依次生成的一组自然数：

$$\text{Hailstone}(n) = \begin{cases} \{1\} & (\text{若 } n=1) \\ \{n\} \cup \text{Hailstone}(n/2) & (\text{若 } n \text{ 为偶数}) \\ \{n\} \cup \text{Hailstone}(3n+1) & (\text{若 } n \text{ 为奇数}) \end{cases}$$

比如， $\text{Hailstone}(7) = \{7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1\}$ 。
试编写一个递归程序^④，计算 $\text{Hailstone}(n)$ 的长度 $\text{hailstone}(n)$ 。

[31] 若假定机器字长无限，移位操作只需单位时间，递归不会溢出，且 $\text{rand}()$ 为理想的随机数发生器。
试分析以下函数 $F(n)$ ，并以大 O 记号的形式确定其渐进复杂度的紧上界。

```
a) void F(int n) {  
    for (int i = 0; i < n; i ++)  
        for (int j = i; j < n; j ++);  
}  
  
b) void F(int n) {  
    for (int i = 0; i < n; i ++)  
        for (int j = 1; j < 2011; j <<= 1);  
}  
  
c) void F(int n) {  
    for (int i = 1; i < n; i ++)  
        for (int j = 0; j < n; j += i);  
}  
  
d) void F(int n) { for (int i = 1, r = 1; i < n; i <<= r, r <<= 1); }  
e) void F(int n) { for (int i = 0, j = 0; i < n; i += j, j ++); }  
f) void F(int n) { for (int i = 1; i < n; i = 1 << i); }  
g) int F(int n) { return (n > 0) ? G(2, F(n - 1)) : 1; }  
    int G(int n, int m) { return (m > 0) ? n + G(n, m - 1) : 0; }  
h) int F(int n) { return (n > 0) ? G(G(n - 1)) : 0; }  
    int G(int n) { return (n > 0) ? G(n - 1) + 2*n - 1 : 0; }  
i) int F(int n) { return (n > 3) ? F(n >> 1) + F(n >> 2) : n; }  
j) void F(int n) {  
    for (int i = n; 0 < i; i --)  
        if (0 == rand() % i)  
            for (int j = 0; j < n; j ++);  
}  
  
k) void F(int n) { for (int i = 1; i < n/G(i, 0); i ++); }  
    int G(int n, int k) { return (n < 1) ? k : G(n - 2*k - 1, k + 1); }
```

^④ 据本书作者所知，“序列 $\text{Hailstone}(n)$ 长度必然有限”的结论尚未得到证明，故你编写的程序可能并不是一个算法。

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第2章

向量

数据结构是数据项的结构化集合，其结构性表现为数据项之间的相互联系及作用，也可以理解为定义于数据项之间的某种逻辑次序。根据这种逻辑次序的复杂程度，大致可以将各种数据结构划分为线性结构、半线性结构与非线性结构三大类。在线性结构中，各数据项按照一个线性次序构成一个整体。最基本的线性结构统称为序列（sequence），根据其中数据项的逻辑次序与其物理存储地址的对应关系不同，又可进一步地将序列区分为向量（vector）和列表（list）。在向量中，所有数据项的物理存放位置与其逻辑次序完全吻合，此时的逻辑次序也称作秩（rank）；而在列表中，逻辑上相邻的数据项在物理上未必相邻，而是采用间接定址的方式通过封装后的位置（position）相互引用。

本章的讲解将围绕向量结构的高效实现而逐步展开，包括其作为抽象数据类型的接口规范以及对应的算法，尤其是高效维护动态向量的技巧。此外，还将针对有序向量，系统介绍经典的查找与排序算法，并就其性能做一分析对比，这也是本章的重点与难点所在。最后，还将引入复杂度下界的概念，并通过建立比较树模型，针对基于比较式算法给出复杂度下界的统一界定方法。

§ 2.1 从数组到向量

2.1.1 数组

C、C++和Java等程序设计语言都将数组作为一种内置的数据类型，支持对一组相关元素的存储组织与访问操作。具体地，若集合S由n个元素组成，且各元素之间构成一个线性的前、后次序，则可将它们存放于起始于地址A、物理位置连续的一段存储空间，并统称作数组（array），通常以A作为该数组的标识。具体地，数组A[]中的每一元素都唯一对应于某一下标编号，在多数高级程序设计语言中，一般都是从0开始编号，依次是0号、1号、2号、...、n-1号元素，记作

$$A = \{a_0, a_1, \dots, a_{n-1}\} = \{A[0], A[1], \dots, A[n-1]\}$$

其中，对于任何 $0 \leq i < j < n$ ， $A[i]$ 都是 $A[j]$ 的前驱（predecessor）， $A[j]$ 都是 $A[i]$ 的后继（successor）。特别地，对于任何 $i \geq 1$ ， $A[i-1]$ 称作 $A[i]$ 的直接前驱（immediate predecessor）；对于任何 $i \leq n-2$ ， $A[i+1]$ 称作 $A[i]$ 的直接后继（immediate successor）。任一元素的所有前驱构成其前缀（prefix），所有后继构成其后缀（suffix）。

按照这一编号规范，不仅使得每个元素都可以通过下标唯一指代，而且可以使我们直接访问到任一元素。这里所说的“访问”包含读取、修改等基本操作，而“直接”则是指这些操作都可以在常数时间内完成。事实上，只要从数组所在空间的起始地址A出发，即可根据每一元素的编号，经过一次乘法运算和一次加法运算获得待访问元素的物理地址。具体地，若数组A[]存放空间的起始地址为A，且每个元素占用s个单位的空间，则元素A[i]将存放于物理地址 $A + i \times s$ 处。在此类数组中，元素的物理存放地址与其下标之间满足这种线性关系，故亦称作线性数组（linear array）。

2.1.2 向量

按照面向对象思想中的数据抽象原则，可对以上的数组结构做一般性推广，使得其以上特性更具普遍性。向量（**vector**）就是线性数组的一种抽象与泛化，它也是由具有线性次序的一组元素构成的集合 $V = \{v_0, v_1, \dots, v_{n-1}\}$ ，其中的元素分别由秩相互区分。

各元素的秩（**rank**）互异，且均为 $[0, n)$ 内的整数。具体地，若元素 e 的前驱元素共计 r 个，则其秩就是 r 。以此前介绍的线性递归为例，运行过程中所出现过的所有递归实例，按照相互调用的关系可构成一个线性序列。在此序列中，各递归实例的秩反映了它们各自被创建的时间先后，每一递归实例的秩等于早于它出现的实例总数。反过来，通过 r 亦可唯一确定 $e = v_r$ 。这是向量特有的元素访问方式，称作“循秩访问”（**call-by-rank**）。

经如此抽象之后，我们不再限定同一向量中的各元素都属于同一基本类型，它们本身可以是来自于更具一般性的某一类的对象。另外，各元素也不见得同时具有某一数值属性，故而并不保证它们之间能够相互比较大小。

以下首先从向量最基本接口出发，设计并实现与之对应的向量模板类。然后在元素之间具有大小可比性的假设前提下，通过引入通用比较器或重载对应的操作符明确定义元素之间的大小判断依据，并强制要求它们按此次序排列，从而得到所谓有序向量（**sorted vector**），并介绍和分析此类向量的相关算法及其针对不同要求的各种实现版本。

§ 2.2 接口

2.2.1 ADT 接口

作为一种抽象数据类型，向量对象应支持如下操作接口。

表2.1 向量ADT支持的操作接口

操作接口	功能	适用对象
<code>size()</code>	报告向量当前的规模（元素总数）	向量
<code>get(r)</code>	获取秩为 r 的元素	向量
<code>put(r, e)</code>	用 e 替换秩为 r 元素的数值	向量
<code>insert(r, e)</code>	e 作为秩为 r 元素插入，原后继元素依次后移	向量
<code>remove(r)</code>	删除秩为 r 的元素，返回该元素中原存放的对象	向量
<code>disordered()</code>	判断所有元素是否已按非降序排列	向量
<code>sort()</code>	调整各元素的位置，使之按非降序排列	向量
<code>find(e)</code>	查找等于 e 且秩最大的元素	向量
<code>search(e)</code>	查找目标元素 e ，返回不大于 e 且秩最大的元素	有序向量
<code>deduplicate()</code>	剔除重复元素	向量
<code>uniquify()</code>	剔除重复元素	有序向量
<code>traverse()</code>	遍历向量并统一处理所有元素，处理方法由函数对象指定	向量

以上向量操作接口，可能有多种具体的实现方式，计算复杂度也不尽相同。而在引入秩的概念并将外部接口与内部实现分离之后，无论采用何种具体的方式，符合统一外部接口规范的任一实现均可直接地相互调用和集成。

2.2.2 操作实例

按照表 2.1 定义的 ADT 接口，表 2.2 给出了一个整数向量从被创建开始，通过 ADT 接口依次实施一系列操作的过程。请留意观察，向量内部各元素秩的逐步变化过程。

表2.2 向量操作实例

操作	输出	向量组成(自左向右)	操作	输出	向量组成(自左向右)
初始化			disordered()	3	4 3 7 4 9 6
insert(0, 9)		9	find(9)	4	4 3 7 4 9 6
insert(0, 4)		4 9	find(5)	-1	4 3 7 4 9 6
insert(1, 5)		4 5 9	sort()		3 4 4 6 7 9
put(1, 2)		4 2 9	disordered()	0	3 4 4 6 7 9
get(2)	9	4 2 9	search(1)	-1	3 4 4 6 7 9
insert(3, 6)		4 2 9 6	search(4)	2	3 4 4 6 7 9
insert(1, 7)		4 7 2 9 6	search(8)	4	3 4 4 6 7 9
remove(2)	2	4 7 9 6	search(9)	5	3 4 4 6 7 9
insert(1, 3)		4 3 7 9 6	search(10)	5	3 4 4 6 7 9
insert(3, 4)		4 3 7 4 9 6	uniquify()		3 4 6 7 9
size()	6	4 3 7 4 9 6	search(9)	4	3 4 6 7 9

2.2.3 Vector 模板类

按照表 2.1 确定的向量 ADT 接口，可定义 Vector 模板类如代码 2.1 所示。

```
1 typedef int Rank; //秩
2 #define DEFAULT_CAPACITY 3 //默认的初始容量(实际应用中可设置为更大)
3
4 template <typename T> class Vector { //向量模板类
5 private:
6     Rank _size; int _capacity; T* _elem; //规模、容量、数据区
7 protected:
8     void copyFrom(T* const A, Rank lo, Rank hi); //复制数组区间A[lo, hi)
9     void expand(); //空间不足时扩容
10    void shrink(); //装填因子过小时压缩
11    bool bubble(Rank lo, Rank hi); //扫描交换算法
12    void bubbleSort(Rank lo, Rank hi); //起泡排序算法
13    void merge(Rank lo, Rank mi, Rank hi); //归并算法
14    void mergeSort(Rank lo, Rank hi); //归并排序算法
15    Rank partition(Rank lo, Rank hi); //轴点构造算法
16    void quickSort(Rank lo, Rank hi); //快速排序算法
17    void heapSort(Rank lo, Rank hi); //堆排序(稍后结合完全堆讲解)
18 public:
```

```
19 // 构造函数
20 Vector(int c = DEFAULT_CAPACITY) { _elem = new T[_capacity = c]; _size = 0; } //默认
21 Vector(T* A, Rank lo, Rank hi) { copyFrom(A, lo, hi); } //数组区间复制
22 Vector(T* A, Rank n) { copyFrom(A, 0, n); } //数组整体复制
23 Vector(Vector<T> const& V, Rank lo, Rank hi) { copyFrom(V._elem, lo, hi); } //向量区间复制
24 Vector(Vector<T> const& V) { copyFrom(V._elem, 0, V._size); } //向量整体复制
25 // 析构函数
26 ~Vector() { delete [] _elem; } //释放内部空间
27 // 只读访问接口
28 Rank size() const { return _size; } //规模
29 bool empty() const { return _size <= 0; } //判空
30 int disordered() const; //判断向量是否已排序
31 Rank find(T const& e) const { return find(e, 0, (Rank)_size); } //无序向量整体查找
32 Rank find(T const& e, Rank lo, Rank hi) const; //无序向量区间查找
33 Rank search(T const& e) const //有序向量整体查找
34 { return (0 >= _size) ? -1 : search(e, (Rank)0, (Rank)_size); }
35 Rank search(T const& e, Rank lo, Rank hi) const; //有序向量区间查找
36 // 可写访问接口
37 T& operator[](Rank r) const; //重载下标操作符，可以类似于数组形式引用各元素
38 Vector<T> & operator=(Vector<T> const&); //重载赋值操作符，以便直接克隆向量
39 T remove(Rank r); //删除秩为r的元素
40 int remove(Rank lo, Rank hi); //删除秩在区间[lo,hi)之内的元素
41 Rank insert(Rank r, T const& e); //插入元素
42 Rank insert(T const& e) { return insert(_size, e); } //默认作为末元素插入
43 void sort(Rank lo, Rank hi); //对[lo, hi)排序
44 void sort() { sort(0, _size); } //整体排序
45 void unsort(Rank lo, Rank hi); //对[lo, hi)置乱
46 void unsort() { unsort(0, _size); } //整体置乱
47 int deduplicate(); //无序去重
48 int uniquify(); //有序去重
49 // 遍历
50 void traverse(void (*)(T&)); //遍历（使用函数指针，只读或局部性修改）
51 template <typename VST> void traverse(VST&); //遍历（使用函数对象，可全局性修改）
52 }; //Vector
```

代码2.1 向量模板类Vector

这里通过模板参数 `T` 指定向量元素的类型，于是，以 `Vector<int>` 或 `Vector<float>` 之类的形式可便捷地引入存放整数或浮点数的向量，而以 `Vector<Vector<char>>` 之类的形式则可直接定义存放字符的二维向量等，这一技巧可提高数据结构选用的灵活性和效率并减少出错，因此将在本书中频繁使用。

在表 2.1 所列基本操作接口的基础上，这里还扩充了一些接口。比如，基于 `size()` 直接实现的判空接口 `empty()`，还有区间删除接口 `remove(lo, hi)`、区间查找接口 `find(e, lo, hi)` 等。它们多为上述基本接口的扩展或变型，在实际应用中可使代码更为简洁易读。

这里还提供了 `sort()` 接口，以将向量转化为有序向量，为此可有多种排序算法供选用，本章及后续章节将陆续介绍它们的原理、实现并分析其效率。排序之后，向量的很多操作都可更加高效地完成，其中最基本和最常用的莫过于查找。因此，这里还针对有序向量提供了 `search()` 接口，并将介绍若干相关的算法。为便于对 `sort()` 算法的测试，这里还设有一个 `unsort()` 接口以将向量随机置乱。在讨论这些接口之前，我们首先介绍基本接口的实现。

§ 2.3 构造与析构

由代码 2.1 可见，向量可在内部维护一个私有数组 `_elem[]`，其容量由私有变量 `_capacity` 指示，其中有效元素的数量（即向量当前的实际规模）则由 `_size` 指示。因此，向量对象的构造与析构主要围绕这些私有变量和数据区的初始化与销毁展开。

2.3.1 默认构造方法

在面向对象的程序设计语言中，所有对象在可使用之前都需要先被系统创建，这一过程称作初始化（`initialization`），向量亦是如此。在 C++ 语言中，对象的创建由构造函数（`constructor`）来完成，同一对象的构造函数可能重载有多个。由代码 2.1 可见，此处向量的默认构造方法是，首先根据创建者指定的初始容量向系统申请空间，以创建内部私有数组 `_elem[]`；若容量未明确指定，则使用默认值 `DEFAULT_CAPACITY`。接下来，鉴于初生的向量尚不含任何元素，故将指示规模的变量 `_size` 初始化为 0。

整个过程顺序进行，没有任何迭代，若不计用于分配数组空间的时间，共需常数时间。

2.3.2 基于复制的构造方法

向量的另一典型创建方式，是以已有向量或数组（的局部或整体）为蓝本克隆。代码 2.1 中虽为此重载了多个版本，但无论是已封装的向量或未封装的数组，无论是整体还是区间子集，在入口参数合法的前提下，都可归于如下统一的 `copyElem()` 方法：

```
1 template <typename T> //元素类型 (T为基本类型，或已重载赋值操作符'=')
2 void Vector<T>::copyFrom(T* const A, Rank lo, Rank hi) { //基于数组复制的构造
3     _elem = new T[_capacity = 2*(hi-lo)]; _size = 0; //分配空间，规模清零
4     while (lo < hi) //A[lo, hi)内的元素逐一
5         _elem[_size++] = A[lo++]; //复制至_elem[0, hi-lo)
6 }
```

代码2.2 基于复制的向量构造器

如代码 2.2 所示，`copyElem()` 首先由待复制区间的前后边界 `lo` 和 `hi` 换算出新向量的初始规模，再以双倍的容量为内部数组 `_elem[]` 申请空间。这一策略可保证在此后足够长的一段时间内，不会因容量不足而导致溢出。随后，通过循环完成区间内各元素的顺次复制。

鉴于共需做线性次复制操作，若忽略为开辟新空间所需的时间，运行时间应正比于复制区间的长度。若每次赋值仅需常数时间，则总体时间复杂度应为 $O(hi-lo) = O(_size)$ 。

需强调的是，由于向量内部含有动态分配的空间，编译器默认的运算符“=”不足以支持向量之间的相互赋值。例如，后面的 6.3 节将以二维向量形式实现图邻接表，其主向量中的每一元素本身都是一维向量，此时默认的赋值运算符并不能复制向量内部的数据区。为适应

此类赋值操作的需求，可如代码 2.3 所示重载向量的赋值运算符。

```
1 template <typename T> Vector<T>& Vector<T>::operator=(Vector<T> const& V) { //重载赋值操作符
2     if (_elem) delete [] _elem; //释放原有内容
3     copyFrom(V._elem, 0, V.size()); //整体复制
4     return *this; //返回当前对象的引用，以便链式赋值
5 }
```

代码2.3 重载向量赋值操作符

2.3.3 析构方法

与所有对象一样，不再需要的向量对象需及时清理（`cleanup`），以释放其占用的系统资源。在 C++ 语言中，对象的销毁由析构函数（`destructor`）完成。与构造函数不同，同一对象只能有一个不可重载的析构函数。

向量对象的析构过程很简单，如代码 2.1 所示的方法 `~Vector()`，只需释放用于存放元素的内部数组 `_elem[]`，将其占用的空间交还操作系统。`_capacity` 和 `_size` 之类的内部变量无需做任何处理，它们将作为向量对象自身的一部分被系统回收，此后既无需也无法被引用。若此，若不计系统用于空间回收的时间，整个析构过程只需 $O(1)$ 时间。

同样地，向量中的元素可能不是程序语言直接支持的基本类型。比如，可能是指向动态分配对象的指针或引用，故在向量析构之前应该提前释放对应的空间。出于简化的考虑，这里约定并遵照“谁申请谁释放”的原则，由上层调用者负责确定，究竟应释放掉向量各元素所指的对象，还是需要保留这些对象以便通过其它指针继续引用它们。

§ 2.4 动态空间管理

2.4.1 静态空间管理

如代码 2.1 实现的向量，在内部使用动态数组 `_elem[]` 封装一组类型为 `T` 的元素，其容量为 `_capacity`，当前的实际规模则由 `_size` 指示。如此，可在向量元素的秩与数组单元的逻辑编号以及物理地址之间建立起直接的对应关系：向量中秩为 `r` 的元素对应于下标为 `r` 的单元 `_elem[r]`，相应的物理地址为 `_elem + r`。

此类仿照常规数组开辟并使用一段地址连续的物理空间的存储方式，称作静态空间管理策略。很遗憾，就空间使用效率而言，这一方式存在两点不足。一方面，向量容量 `_capacity` 一旦固定，就可能在此后的某一时刻无法加入更多的新元素，亦即出现所谓上溢（`overflow`）的情况。例如，若使用此类向量结构来记录网络服务器的访问日志，则由于插入操作远多于删除操作，不用多久就会溢出。注意，导致溢出的原因并非系统不能提供更多的空间，而是因为向量结构的容量已事先确定。反过来，即便愿意为降低这种风险而预留过多的空间，也很难在程序执行之前明确界定一个合理的空间预留量。在数据规模相对较小的应用中，预留量过大又可能导致空间利用率不足甚至极低。

向量实际规模与内部数组容量的比值（即 `_size/_capacity`）称作装填因子（`load factor`），反映了空间利用率的高低。上述难题可归纳为：如何才能保证向量的装填因子既不至于超过 1，也不至于太接近于 0？为此，需要采用适当的动态空间管理策略。

2.4.2 可扩充向量

解决上述矛盾的一种有效方法，即所谓的可扩充向量（extendable vector）。

可扩充向量的生长方式类似于蝉：身体在每经过一段时间的生长之后若无法继续为其外壳所容纳，就蜕去原先的外壳而换上一身更大的外壳。那么，如何才能动态地调整向量内部数组的容量呢？比如，在溢出即将发生时，可否适当地扩大内部数组的容量？

直接在原数组基础上追加空间的思路并不现实，因为数组特有的定址方式本身要求存放元素的空间地址必须连续，而通常的操作系统并不能保证原数据区的尾部总是预留有足够的空间。为此，必须专门设计和实现一套可行的管理算法。

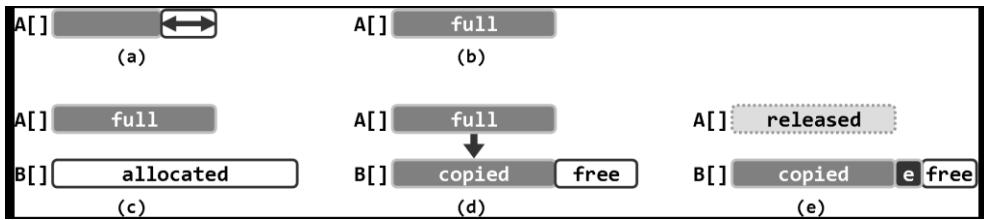


图2.1 可扩充向量的溢出处理

具体地，扩容过程如图 2.1 所示。多数时候内部数组的初始容量仍然够用，此时插入操作可直接执行。当然，每经一次插入（删除），可用空间都会减少（增加）一个单元（图(a))。在经一系列操作过程后，若因插入远多于删除而致使数组 A[]充满，则继续插入将会导致溢出（图(b))。此时，为保证新元素的顺利插入，可另行申请一个容量更大的数组 B[]（图(c))，并将原数组中的成员集体搬迁至新的空间（图(d))，然后即可顺利地插入新元素 e（图(e))。当然，原先所用的数组则需要释放并交还操作系统。

2.4.3 扩容

按照以上思路，可实现向量内部数组动态扩容算法 `expand()` 如下。

```
1 template <typename T> void Vector<T>::expand() { //向量空间不足时扩容
2     if (_size < _capacity) return; //尚未满员时，不必扩容
3     if (_capacity < DEFAULT_CAPACITY) _capacity = DEFAULT_CAPACITY; //不低于最小容量
4     T* oldElem = _elem; _elem = new T[_capacity <= 1]; //容量加倍
5     for (int i = 0; i < _size; i++)
6         _elem[i] = oldElem[i]; //复制原向量内容 (T为基本类型，或已重载赋值操作符'=')
7     delete [] oldElem; //释放原空间
8 }
```

代码2.4 向量内部数组动态扩容算法`expand()`

在调用 `insert()` 接口往向量中插入新元素之前，都要先调用 `expand()` 算法检查内部数组的剩余容量，若果真当前数据区已充满 (`_size == _capacity`)，则按照上述思路将原数组替换为一个更大的数组。请注意，新数组的容量总是取作原数组的两倍。

请注意，新数据区的地址完全取决于操作系统的分配，与原数据区没有直接关系。若直接引用数组，往往导致共同指向原数据区的其它指针失效，成为野指针（wild pointer）。经如上封装之后，不仅可经向量继续准确地引用各元素，更可有效地避免野指针的风险。

2.4.4 分摊分析

■ 时间代价

与基本的数组实现相比，容量可变的可扩充向量可以更高效地利用存储空间；另外，只要系统还有可利用的空间，向量的最大规模将不再受限于数组的初始容量。不过，为实现空间方面的这一改进，我们不得不在时间方面付出相应的代价——每次扩容时都需要花费额外的时间，将原数组的元素逐一复制到新的数组中。即便假定每一元素的复制仅需常数时间，为实现一次规模由 n 到 $2n$ 的扩容，也需要花费 $\mathcal{O}(2n) = \mathcal{O}(n)$ 的时间。

尽管表面看来这一扩充数组的策略似乎效率很低，但实际上这不过是一种错觉。请注意，按照此处的约定，每花费 $\mathcal{O}(n)$ 时间实施一次扩容之后，数组的容量都会加倍。这就意味着，至少要再经过 n 次插入操作，才有可能因重新溢出而再次扩容。也就是说，随着向量规模的扩大，插入操作时需要进行扩容的概率将迅速降低，故而从某种平均的意义而言，用于扩容的时间成本不至很高。以下不妨就此做一严格的分析。

■ 分摊复杂度

实际应用中插入、删除操作极其随机，扩容操作何时执行并不确定，因此通过常规方法难以准确地度量和分析其复杂度。为此需要考查对同一数据结构相继执行的足够多次操作：整个过程所需的运行时间分摊至其间执行的所有操作，即为单次操作的分摊运行时间（**amortized running time**）。请注意，这一概念与平均时间（**average running time**）有本质的区别。后者是基于某种概率分布假设，对各种情况下所需执行时间的加权平均，故亦称作期望运行时间（**expected running time**）；分摊运行时间则要求参与分摊的操作必须构成和来自一个足够长的真实可行的操作序列。因此相对而言，分摊复杂度可以对计算成本做出更为客观而准确的估计（习题[1]）。作为评定算法性能的重要尺度和方法，分摊分析（**amortized analysis**）的相关方法与技巧将在后续章节陆续介绍。

■ $\mathcal{O}(1)$ 分摊时间

以可扩充向量为例，可以考查对该结构的连续 n 次（查询、插入或删除等）操作，将所有操作中用于内部数组扩容的时间累计起来，然后除以 n 。只要 n 足够大，这一平均时间就是用于扩容处理的分摊计算时间。以下我们将看到，即便排除查询和删除操作而仅考查插入操作，可扩充向量单次操作中用于扩容处理的分摊计算时间也不过 $\mathcal{O}(1)$ 。

假定数组的初始容量为常数 N 。既然是估计复杂度的上界，故不妨设向量的初始规模也为 N ——即将溢出。另外不难看出，除插入操作外，向量其余的接口操作既不会直接导致溢出，也不会增加此后溢出的可能性，因此不妨考查最坏的情况，假设在此后需要连续地进行 n 次 `insert()` 操作， $n \gg N$ 。首先定义如下函数：

`size(n)` = 连续插入 n 个元素后向量的规模

`capacity(n)` = 连续插入 n 个元素后数组的容量

`T(n)` = 为连续插入 n 个元素而花费于扩容的时间

其中，向量规模从 N 开始随着操作的进程线性增长，故有：

`size(n)` = $N + n$

另外，数组的容量应足以容纳 n 个元素，故装填因子不至超过 100%。同时，这里的扩容策略属于懒惰类型——只有在的确即将发生溢出时，才不得不将容量加倍——因此装填因

子也始终大于 50%。概括起来，应有

$$\text{size}(n) \leq \text{capacity}(n) < 2 \cdot \text{size}(n)$$

考虑到 N 为常数，故有

$$\text{capacity}(n) = \Theta(\text{size}(n)) = \Theta(n)$$

容量以 2 为比例按指数速度增长，在容量达到 $\text{capacity}(n)$ 之前，共做过 $\Theta(\log_2 n)$ 次扩容，每次扩容所需时间线性正比于当时的容量（或规模），且同样以 2 为比例按指数速度增长。因此，消耗于扩容的时间累计不过

$$T(n) = 2N + 4N + 8N + \dots + \text{capacity}(n) < 2 \cdot \text{capacity}(n) = \Theta(n)$$

将其分摊到其间的连续 n 次操作，单次操作所需的分摊运行时间应为 $O(1)$ 。

■ 其它扩容策略

以上分析确凿地说明，基于加倍策略的动态扩充数组不仅可行，而且就分摊复杂度而言效率也足以令人满意。当然，并非任何扩容策略都能保证如此高的效率。比如，早期可扩充数组多采用另一简单策略：一旦有必要，则追加固定数目的单元。实际上，无论采用的常数多大，在最坏情况下，此类数组单次操作的分摊时间复杂度都将高达 $\Omega(n)$ （习题[2]）。

2.4.5 缩容

导致低效率的另一情况是，向量的实际规模可能远远小于内部数组的容量。比如在连续的一系列操作过程中，若删除操作远多于插入操作，则装填因子极有可能远远小于 100%，甚至非常接近于 0。当装填因子低于某一阈值时，我们称数组发生了下溢（underflow）。

尽管下溢不属于必须解决的问题，但在格外关注空间利用率的场合，发生下溢时也有必要适当缩减内部数组容量。代码 2.5 给出了一个动态缩容 `shrink()` 算法：

```
1 template <typename T> void Vector<T>::shrink() { //装填因子过小时压缩向量所占空间
2     if (_capacity < DEFAULT_CAPACITY << 1) return; //不致收缩到DEFAULT_CAPACITY以下
3     if (_size << 2 > _capacity) return; //以25%为界
4     T* oldElem = _elem; _elem = new T[_capacity >>= 1]; //容量减半
5     for (int i = 0; i < _size; i++) _elem[i] = oldElem[i]; //复制原向量内容
6     delete [] oldElem; //释放原空间
7 }
```

代码2.5 向量内部功能`shrink()`

可见，每次删除操作之后，一旦空间利用率已降至某一阈值以下，该算法随即申请一个容量减半的新数组，将原数组中的元素逐一搬迁至其中，最后将原数组所占空间交还操作系统。这里以 25% 作为装填因子的下限，但在实际应用中，为避免出现频繁交替扩容和缩容的情况，可以选用更低的阈值，甚至取作 0（相当于禁止缩容）。

与 `expand()` 操作类似，尽管单次 `shrink()` 操作需要线性量级的时间，但其分摊复杂度亦为 $O(1)$ （习题[3]）。实际上 `shrink()` 过程等效于 `expand()` 的逆过程，这两个算法相互配合，在不致实质地增加接口操作复杂度的前提下，保证了向量内部空间的高效利用。当然，就单次扩容或缩容操作而言，所需时间的确会高达 $\Omega(n)$ ，所以以上策略并不适用于对每次操作的执行速度都极其敏感的应用场合。

§ 2.5 常规向量

2.5.1 直接引用元素

与数组直接通过下标访问元素的方式(形如“`A[i]`”)相比，向量 ADT 所设置的 `get()` 和 `put()` 接口都显得不甚自然。毕竟，前一访问方式不仅更为我们所熟悉，同时也更加直观和便捷。那么，在经过封装之后，对向量元素的访问可否沿用数组的方式呢？答案是肯定的。

解决的方法之一就是重载(`overload`)操作符“`[]`”，具体如代码 2.6 所示。

```
1 template <typename T> T& Vector<T>::operator[](Rank r) const //重载下标操作符
2 { return _elem[r]; } // assert: 0 <= r < _size
```

代码2.6 重载向量操作符[]

2.5.2 置乱器

■ 置乱算法

可见，经重载后操作符“`[]`”返回的是对数组元素的引用，这就意味着它既可以取代 `get()` 操作(通常作为赋值表达式的右值)，也可以取代 `set()` 操作(通常作为左值)。例如，采用这种形式，可以简明清晰地描述和实现如代码 2.7 所示的向量置乱算法。

```
1 template <typename T> void permute(Vector<T>& V) { //随机置乱向量，使各元素等概率出现于每一位置
2     for (int i = V.size(); i > 0; i--) //自后向前
3         swap(V[i-1], V[rand() % i]); //V[i-1]与V[0, i-1]中某一随机元素交换
4 }
```

代码2.7 向量整体置乱算法permute()

该算法的主体结构是，从待置乱区间的末元素开始逆序向前扫描。对于每一当前元素 `V[i-1]`，通过调用 `rand()` 函数在 `[0, i)` 之间等概率地随机选取一个元素，并将二者交换。注意，这里的交换操作 `swap()` 隐含了三次基于重载操作符“`[]`”的赋值。

在软件测试等应用中，随机向量的生成都属于重要的基本操作。使用算法 `permute()`，不仅可均匀地重排已有向量中各元素的位置，而且可枚举出同一向量的各种排列(习题[4])。

■ 区间置乱接口

为便于对各种向量算法的测试与比较，这里不妨将以上 `permute()` 算法封装至向量 ADT 中，对外提供向量的置乱操作接口 `Vector::unsort()`。

```
1 template <typename T> void Vector<T>::unsort(Rank lo, Rank hi) { //等概率随机置乱向量区间[lo,hi)
2     T* V = _elem + lo; //将子向量_elem[lo, hi)视作另一向量V[0, hi-lo)
3     for (Rank i = hi - lo; i > 0; i--) //自后向前
4         swap(V[i-1], V[rand() % i]); //将V[i-1]与V[0, i-1]中某一元素随机交换
5 }
```

代码2.8 向量区间置乱接口unsort()

如代码 2.8 所示，通过该接口可均匀地置乱任一向量区间 `[lo, hi)` 内的元素，故通用性有所提高。可见，只要将该区间等效地视作另一向量 `V`，即可从形式上完整地套用以上 `permute()` 算法的流程。尽管如此，还请特别留意代码 2.8 与代码 2.7 的微妙差异，此处是通过下标直接地访问内部数组的元素，而不是通过下标间接地访问向量的元素。

2.5.3 判等器与比较器

从算法的角度来看，“判断两个对象是否相等”与“判断两个对象的大小”都是至关重要的操作，它们直接控制着算法执行的分支方向，也因此构成了算法的“灵魂”。为以示区别，前者多称作“比对”操作，后者多称作“比较”操作。当然，这两种操作之间既有联系也有区别，不能相互替代。比如，有些对象只能比对但不能比较；反之，支持比较的对象未必支持比对。不过，出于简化的考虑，本书在多数场合并不会严格地区分二者。

在强调算法实现的简洁性与通用性时，我们关注的一个重要方面在于，算法的某种实现可否适用于任何类型的可比较或可比对数据对象，而不必关心其大小或相等关系的具体含义及实现方式。若能如此，我们就可以将比对和比较操作的具体实现分离出来，独立于算法的流程和数据类型之外。为此，通常可以采用两种方法：其一，将比对和比较操作封装成通用的判等器和比较器；其二，在定义对应的数据类型时，通过重载“<”和“==”之类的操作符具体定义相等和大小关系。本书将主要采用后一方式。为节省篇幅，书中只会给出涉及到的判等和比较操作符，读者可以根据实际需要，参照给出的代码加以扩充。

```
1 template <typename T> static bool lt(T* a, T* b) { return lt(*a, *b); } //less than
2 template <typename T> static bool lt(T& a, T& b) { return a < b; } //less than
3 template <typename T> static bool eq(T* a, T* b) { return eq(*a, *b); } //equal
4 template <typename T> static bool eq(T& a, T& b) { return a == b; } //equal
```

代码2.9 重载比较器以便比较对象指针

在一些复杂的数据结构中，元素的内部类型可能是指向其它对象的指针，而从外部更多关注的往往是其所指对象的大小。若不加处理，直接根据指针的数值（即被指对象的物理地址）所做的比较将毫无意义。为此，这里不妨通过如代码 2.9 所示的机制，约定这种情况下统一按照被指对象的大小做出判断。

2.5.4 无序查找

■ 判等器

代码 2.1 中 `Vector::find(e)` 接口的功能语义为“查找与数据对象 `e` 相等的元素”，这也是向量结构最常用的操作接口之一。这一操作同时也暗示着向量中的元素之间可以通过比对判等是否相等，比如按照这里的约定，假设类型 `T` 或为基本类型，或已经重载了操作符“`==`”或“`!=`”。尽管如此，这并不意味着此类元素之间必然可以比较大小，更不意味着它们已在向量中按某一次序排列。这类向量，也称作无序向量（`unsorted vector`）。

■ 顺序查找

在无序向量中查找任意指定元素 `e` 时，最坏情况下在所有元素都被访问过之前我们无法确定该元素是否的确存在。因此，不妨从末元素起自后向前，逐一取出各个元素并与目标元素 `e` 做比对，直至发现与之匹配者（查找成功），或者直至检查过所有元素之后仍未找到（查找失败）。这种依次逐个比对的查找方式称作顺序查找（`sequential search`）。

■ 实现

在代码 2.1 中，对应于向量的整体或区间分别定义了一个顺序查找操作的入口，其中前者作为特例可直接通过调用后者而实现。因此，只需如代码 2.10 所示实现针对向量区间查找的版本。

```
1 template <typename T> //无序向量的顺序查找：返回最后一个元素e的位置；失败时，返回lo - 1
2 Rank Vector<T>::find(T const& e, Rank lo, Rank hi) const { //assert: 0 <= lo < hi <= _size
3     while ((lo < hi--) && (_elem[hi] != e)); //从后向前，顺序查找
4     return hi; //若hi < lo，则意味着失败；否则hi即命中元素的秩
5 }
```

代码2.10 无序向量元素查找接口find()

这里采用了自后向前的查找次序。颠倒次序虽未尝不可，但相对而言这一次序更为便利。比如，当向量中含有多个命中元素时，这样将总是返回其中秩最大者，从而减少后续（插入之类）操作所需的时间。正因如此，表2.1也对find()接口做此约定。另外，查找失败时统一返回-1而非向量的长度n，这不仅统一和简化了对查找结果的判别，同时也使得查找失败时的返回结果更加易于理解——只要假想着在秩为-1处存入一个与任何对象都匹配的元素（即通配符），则返回该虚拟元素的秩（-1）当且仅当查找失败。

最后还有一处需要留意。其中while循环的控制逻辑由两部分组成，首先判断是否已抵达通配符处，再判断当前元素与目标元素是否相等。得益于C/C++语言中逻辑表达式的短路求值特性，在前一判断非真后循环会立即终止，而不致因试图引用已越界的秩i而出错。

■ 复杂度

对于区间[lo, hi)中的每个元素，整个过程至多访问一次，故累计执行时间线性正比于区间的宽度hi-lo。计入其余操作所需的常数时间，总体复杂度为 $\mathcal{O}(lo-hi+1) = \mathcal{O}(n)$ 。

2.5.5 插入

■ 实现

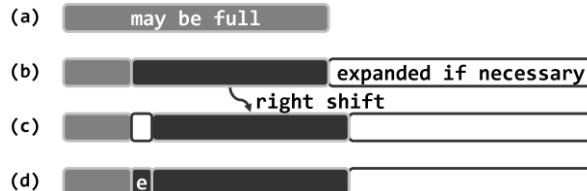


图2.2 向量元素插入操作insert(r, e)的过程

按照代码2.1的ADT定义，插入操作insert(r, e)负责将元素e插到秩为r的单元，为此如图2.2所示，该单元及其后继都将顺次向后移动一个单元。整个操作过程可具体实现如代码2.11所示：

```
1 template <typename T> //将e作为秩为r元素插入
2 Rank Vector<T>::insert(Rank r, T const& e) { //assert: 0 <= r <= size
3     expand(); //若有必要，扩容
4     for (int i = _size; i > r; i--) _elem[i] = _elem[i-1]; //自后向前，后继元素顺次后移一个单元
5     _elem[r] = e; _size++; //置入新元素并更新容量
6     return r; //返回秩
7 }
```

代码2.11 向量元素插入接口insert()

■ 扩容

首先统一调用expand()，若有必要则加倍扩容。扩容的条件是，如(a)所示内部数组已满。扩容之后的内部数组如(b)所示。按照数组固有的要求，各元素须以物理地址连续的方式依次存放。故如(c)所示，随后需要自后向前地将位于r之后的元素_elem[r, _size)

(如果有的话)后移一个单元。请留意后继元素搬迁的次序,自前向后的搬迁将会因元素被覆盖而造成数据丢失。在单元`_elem[r]`腾出之后,方可将待插入对象`e`置入其中。

■ 复杂度

若不计扩容, `insert(r, e)`的计算主要集中于后继元素后移的循环。循环次数取决于位于插入位置之后的元素数目`_size - r`。计入其余部分的常数时间,总体时间复杂度为 $\mathcal{O}(_size - r + 1)$ ——被插入元素越靠后(前)所需时间越短(长)。特别地,当`r`取最大值`_size`时为最好情况,只需 $\mathcal{O}(_size - _size + 1) = \mathcal{O}(1)$ 时间。若插入位置等概率分布,则平均时间复杂度为 $\mathcal{O}(_size) = \mathcal{O}(n)$ (习题[6]),线性正比于向量的实际规模。

2.5.6 删除

删除操作重载有两个接口, `remove(lo, hi)`用以删除区间`[lo, hi)`内的元素,而`remove(r)`用以删除秩为`r`的单个元素。乍看起来,利用后者即可实现前者——令`r`从`hi-1`到`lo`递减,并反复调用`remove(r)`。不幸的是,这一思路似是而非。按照数组元素地址连续的约定,每删除一个元素,所有后继元素都需集体向前移动一个单元。若后继元素共有`m = _size - hi`个,则这类移动共需做`m`次。若按以上思路反复调用`remove(r)`,则元素移动的次数将累计达到`m*(hi-lo)`,与后继元素的数目、区间宽度均呈线性正比关系。

实际可行的思路恰好相反,应将单元素删除视作区间删除的特例,并基于后者实现前者。稍后就会看到,依此思路,移动操作的总次数可控制在`m`以内,而与被删除区间的宽度无关。

■ 区间删除: `remove(lo, hi)`

向量区间删除接口`remove(lo, hi)`的实现如代码2.12所示。

```
1 template <typename T> int Vector<T>::remove(Rank lo, Rank hi) { //删除区间[lo, hi)
2     if (lo == hi) return 0; //出于效率考虑,单独处理退化情况,比如remove(0, 0)
3     while (hi < _size) _elem[lo++] = _elem[hi++]; // [hi, _size)顺次前移hi-lo个单元
4     _size = lo; //更新规模,直接丢弃尾部[lo, _size = hi)区间
5     shrink(); //若有必要,则缩容
6     return hi - lo; //返回被删除元素的数目
7 }
```

代码2.12 向量区间删除接口`remove(lo, hi)`

如图2.3,设`[lo, hi)`(a)为原内部数组(图(a))的合法区间(图(b)),则原秩不小于`hi`的所有后继元素直接自前向后逐个前移`hi - lo`个单元(图(c))。

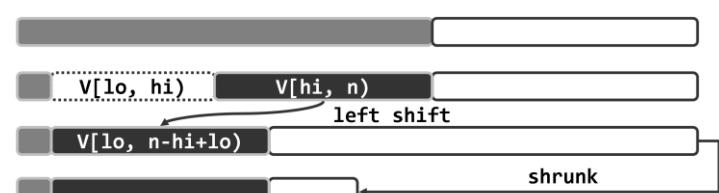


图2.3 向量区间删除操作`remove(lo, hi)`的过程

注意,与插入算法中区间搬迁一样,后继元素的移动次序不能颠倒,否则在某些时候也会因不慎覆盖而导致部分元素数值丢失(习题[7])。前移完毕,即可将向量规模更新为`_size - hi + lo`。当然,最后还要调用`shrink()`,若有必要则做缩容处理(图(d))。

■ 单元素删除 `remove(r)`

利用以上`remove(lo, hi)`通用接口,通过重载即可实现另一同名接口`remove(r)`。

```
1 template <typename T> T Vector<T>::remove(Rank r) { //删除向量中秩为r的元素，0 <= r < size
2     T e = _elem[r]; //备份被删除元素
3     remove(r, r + 1); //调用区间删除算法，等效于对区间[r, r + 1)的删除
4     return e; //返回被删除元素
5 }
```

代码2.13 向量单元素删除接口remove()

■ 复杂度

可见，`remove(lo, hi)`的计算成本主要消耗于依次前移后续元素的循环，其余操作总计不过常数倍常数复杂度。准确地，循环次数取决于位于删除区间之后的元素数目，即 $m = _size - hi$ 。因此，总的时间复杂度应为 $\mathcal{O}(_size - hi + 1) = \mathcal{O}(m)$ 。这与此前的预期吻合：区间删除操作所需的时间只取决于后继元素的数目，而与被删除区间的宽度无关。

特别地，基于该接口实现的`remove(r)`接口需耗时 $\mathcal{O}(_size - r + 1)$ 。也就是说，被删除元素越靠后（前）所需时间越短（长），最好为 $\mathcal{O}(1)$ ，最坏为 $\mathcal{O}(n) = \mathcal{O}(_size)$ 。

■ 意外处理

请注意，上述操作接口对输入有一定的限定，其中设定的待删除区间必须落在合法范围 $[0, _size)$ 之内，为此输入参数必须满足 $0 \leq lo \leq hi \leq _size$ 。

一般地，输入参数超出合法范围属于所谓错误（error）或意外（exception）的典型情况。在真正严谨且可用的实现中，应尽可能对此类情况做周全的处理，比如采用流行的“try .. catch .. throw”模式捕获并处理意外。尽管如此，本书还是统一沿用了相对简化的方式，将入口参数合法性检查的责任统一交由上层调用例程。事实上，这一简化的处理原则不仅有利于保证算法整体的效率，同时在一定程度上仍不失为一种规范的方式。更重要地，这种处理方式可使本书在讲解与叙述过程中的目标和重点更为突出。

2.5.7 唯一化

很多应用场合中，在进一步处理向量之前都要求其中的元素互异。例如，网络搜索引擎均由多个计算节点协同实现，在它们返回各自的局部搜索结果之后，往往还需要剔除其中重复的项目，以合并为一份完整的报表并提交给网络搜索用户。与此类似，所谓向量的唯一化处理，就是从向量中剔除重复元素的过程，即表2.1所列`deduplicate()`接口的功能。

■ 实现

根据向量是否已做排序，此项功能可有两种实现方式，以下先介绍针对无序向量的版本`deduplicate()`，具体实现如代码2.14所示。

```
1 template <typename T> int Vector<T>::deduplicate() { //删除无序向量中重复元素（高效版）
2     int oldSize = \_size; //记录原规模
3     Rank i = 1; //从_elem[1]开始
4     while (i < \_size) //自前向后逐一考查各元素_elem[i]
5         (0 > find(_elem[i], 0, i)) ? //在其前缀中寻找与之雷同者（至多一个）
6             i++ : remove(i); //若无雷同则继续考查其后继，否则删除雷同者
7     return oldSize - \_size; //向量规模变化量，即被删除元素总数
8 }
```

代码2.14 无序向量清除重复元素接口deduplicate()

该算法自前向后逐一考查各元素 `_elem[i]`，并通过调用 `find()` 接口尝试在其前缀中寻找与之雷同者。若找到雷同者则删除，否则转而考查当前元素的后继。

■ 正确性

算法的正确性由以下不变性保证：在 `while` 循环中，当前元素前缀 `_elem[0, i)` 中的所有元素互异。初次进入循环时 $i = 1$ ，只有唯一的前驱 `_elem[0]`，故不变性自然满足。

一般地如图 2.4(a) 所示，假设在转至元素 $e = _elem[i]$ 之前此不变性一直成立。于是经过针对该元素的一次迭代之后，无非两种结果。

若元素 e 的前缀 `_elem[0, i)` (白色区域) 中不含与之雷同的元素，则如图(b)，在做过 $i++$ 之后，新前缀 `_elem[0, i)` 依然继续满足不变性，而且规模增加一个单位。

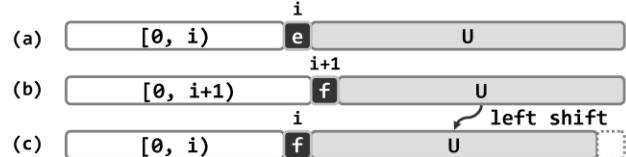


图2.4 无序向量 `deduplicate()` 算法原理

反之，若含存在与 e 雷同的元素，则由此前仍满足的不变性可知这样的雷同元素不超过一个，因此如图(c)，在删除 e 之后，前缀 `_elem[0, i)` 依然保持不变性。

■ 复杂度

由图 2.4(a) 和 (b) 也可看出该算法的单调性：随着循环的进行，当前元素的后继不断严格减少。因此，经过 $n-2$ 次迭代之后该算法必然终止。

每次迭代所需的时间主要消耗于对 `find()` 和 `remove()` 接口的调用。根据 2.5.4 节的分析结论，前一部分时间应线性正比于查找区间的宽度，亦即当前节点的前驱总数；根据 2.5.6 节的分析结论，后一部分时间应线性正比于当前节点的后继总数。因此，最坏情况下每一迭代所需时间为 $\mathcal{O}(n)$ ，总体复杂度应为 $\mathcal{O}(n^2)$ 。

在 2.6.3 节我们将看到，在通过排序将无序向量转化为有序向量之后，`uniquify()` 算法可在 $\mathcal{O}(n)$ 时间内剔除其中的雷同项。由于排序可在 $\mathcal{O}(n \log n)$ 时间内完成，故借助 `uniquify()` 算法，无序向量的 `deduplicate()` 算法的效率可以进一步提高（习题[9]）。

2.5.8 遍历

■ 功能

在很多算法中，往往需要将向量作为一个整体，对其中所有元素实施某种统一的操作，比如输出向量中的所有元素，或者按照某种运算流程统一修改所有元素的数值（习题[10]）。针对此类操作，可为向量专门设置一个遍历接口 `traverse()`。

■ 实现

代码 2.15 针对向量给出了遍历操作接口的具体实现。

```
1 template <typename T> void Vector<T>::traverse(void (*visit)(T)) //利用函数指针机制的遍历
2 { for (int i = 0; i < _size; i++) visit(_elem[i]); }
3
4 template <typename T> template <typename VST> //元素类型、操作器
5 void Vector<T>::traverse(VST& visit) //利用函数对象机制的遍历
6 { for (int i = 0; i < _size; i++) visit(_elem[i]); }
```

代码2.15 向量遍历接口 `traverse()`

可见，`traverse()`遍历的过程，实质上就是自前向后逐一对各元素实施同一基本操作，而具体采用何种操作，可通过两种方式指定。前一种方式借助函数指针`*visit()`指定某一函数，该函数只有一个参数，其类型为对向量元素的引用，故通过该函数即可直接访问或修改向量元素。另外，也可通过另一重载的接口，以函数对象的形式指定具体的遍历操作。

相比较而言，后一形式的功能更强，适用范围更广。比如，函数对象的形式支持对向量元素的关联修改。也就是说，对各元素的修改不仅可以相互独立地进行，也可以根据某个(些)元素的数值相应地修改另一元素。前一形式虽也可实现这类功能，但要繁琐很多。

■ 实例

在代码 2.16 中，`Increase<T>()`即是按函数对象形式指定的基本操作，其功能是将作为参数的引用对象的数值加一（假定元素类型 `T` 可直接递增或已重载操作符“`++`”）。于是可如 `increase()` 函数那样，以此基本操作做遍历即可使向量内所有元素的数值同步加一。

```
1 template <typename T> struct Increase //函数对象：递增一个T类对象
1   { virtual void operator()(T& e) { e++; } }; //假设T可直接递增或已重载++
2
3 template <typename T> void increase(Vector<T> & V) //统一递增向量中的各元素
4 { V.traverse(Increase<T>()); } //以Increase<T>()为基本操作进行遍历
```

代码2.16 基于遍历实现increase()功能

■ 复杂度

遍历操作本身只包含一层线性的循环迭代，故除了向量规模的因素之外，遍历所需时间应线性正比于所统一指定的基本操作所需的时间。比如在上例中，统一的基本操作 `Increase<T>()` 只需常数时间，故这一遍历的总体时间复杂度为 $O(n)$ 。

§ 2.6 有序向量

若向量 $S[0, n]$ 中所有元素不仅按线性次序存放，而且其数值大小也按此次序单调分布，则称之为有序向量（sorted vector）。例如，所有学生的记录可按学号构成一个有序向量（学生名单），使用同一跑道的所有航班可按起飞时间构成一个有序向量（航班时刻表），第二十九届奥运会男子跳高决赛中各选手的记录可按最终跳过的高度构成一个（非增）序列（名次表）。与通常的向量一样，有序向量依然不要求元素互异，故通常约定其中的元素自前（左）向后（右）构成一个非降序列，即对任意 $0 \leq i < j < n$ 都有 $S[i] \leq S[j]$ 。

2.6.1 比较器

当然，除了与无序向量一样支持元素之间的“判等”操作，有序向量的定义中实际上还隐含了另一更强的先决条件：各元素之间能够比较大小。这一条件构成有序向量中“次序”概念的基础，否则所谓的“有序”将无从谈起。多数高级程序语言中支持的基本数据类型都满足这一条件，比如整型、浮点型和字符型等。然而，字符串、复数、矢量以及更为复杂的类型则不见得直接提供某种自然的大小比较规则，为保持大小比较的意义及可行性，常见的一种方法是在内部指定某一（些）可比较的数据项，并由此确立比较的规则。这里按照 2.5.3 节的约定，假设复杂数据对象已经重载了“`<`”和“`<=`”等操作符。

2.6.2 有序性甄别

作为无序向量的特例，有序向量自然可以沿用无序向量的查找算法。然而，得益于元素之间的有序性，有序向量的查找、唯一化等操作都可更快地完成。在首次调用针对有序向量的这些接口之前，都有必要先判断当前向量是否已经有序。若是，则可采用更为高效的方法；否则，需调用排序算法使之有序。代码 2.17 给出了一个鉴别向量是否尚未排序的算法。

```
1 template <typename T> int Vector<T>::disordered() const { //返回向量中逆序相邻元素对的总数
2     int n = 0; //计数器
3     for (int i = 1; i < _size; i++) //逐一检查_size-1对相邻元素
4         if (_elem[i-1] > _elem[i]) n++; //逆序则计数
5     return n; //向量有序当且仅当n = 0
6 }
```

代码2.17 有序向量甄别算法disordered()

该算法的原理与 1.1.3 节介绍的起泡排序算法相同：向量尚未整体排序，当且仅当其中某对相邻元素逆序。为此，只需顺次扫描整个向量，并逐一比较对每一相邻元素。逆序元素对的总数，将返回给上层调用者。

2.6.3 唯一化

相对于无序向量，有序向量中清除重复元素的操作更为重要。正如 2.5.7 节所指出的，出于效率的考虑，为清除无序向量中的重复元素，一般做法往往是首先将其转化为有序向量。

■ 低效版

首先来看唯一化算法如代码 2.18 所示的一个版本。

```
1 template <typename T> int Vector<T>::uniquify() { //有序向量重复元素剔除算法（低效版）
2     int oldSize = _size; int i = 0; //当前比对元素的秩，起始于首元素
3     while (i < _size - 1) //从前向后，逐一比对各对相邻元素
4         (_elem[i] == _elem[i + 1]) ? remove(i + 1) : i++; //若雷同，则删除后者；否则，转至下一元素
5     return oldSize - _size; //向量规模变化量，即被删除元素总数
6 }
```

代码2.18 有序向量uniquify()接口的平凡实现

该算法的正确性基于如下事实：在有序向量中，重复元素必然相互前后紧邻。于是，可以自前向后地逐一检查各对相邻的元素：若二者雷同则调用 `remove()` 接口删除靠后者，否则转向下一对相邻的元素。如此，在扫描结束时，向量将不再含有重复元素。

运行时间主要消耗于 `while` 循环。若记向量规模 $_size = n$ ，则共需循环 $n - 1$ 次。此外，在最坏情况下每次循环都需实施一次 `remove()` 操作，由 2.3 节的分析结论，其复杂度线性正比于被删除元素的后继元素总数。如图 2.5，当大量甚至所有元素雷同时，用于所有 `remove()` 操作的时间总量将高达：

$$(n-2) + (n-3) + \dots + 2 + 1 = O(n^2)$$

图2.5 低效版uniquify()算法的最坏情况

此效率竟与向量未排序时相同，说明该方法未能充分利用此时向量的有序性。

■ 改进思路

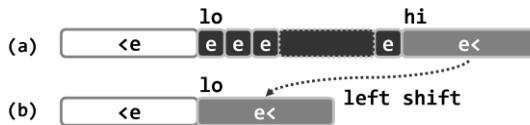


图2.6 有序向量中的重复元素可批量删除

稍加分析即不难看出，以上唯一化过程复杂度过高的根源是，在对 `remove()` 接口的各次调用中，同一元素可能作为后继元素向前移动多次，且每次仅移动一个单元。

如上所言，此时的重复元素必然前后紧邻地集中分布。因此可如图 2.6，以区间为单位成批地删除紧邻的重复元素，并将其后继元素（若存在）统一地大跨度前移。若 $V[lo, hi)$ 为一组紧邻的重复元素，则后继元素 $V[hi, _size)$ 可整体地前移 $hi - lo - 1$ 个单元。

■ 高效版

按照这一思路，可如代码 2.19 所示得到唯一化算法的新版本。

```
1 template <typename T> int Vector<T>::uniquify() { //有序向量重复元素剔除算法(高效版)
2     Rank i = 0, j = 0; //各对互异“相邻”元素的秩
3     while (++j < _size) //逐一扫描，直至末元素
4         if (_elem[i] != _elem[j]) //跳过雷同者
5             _elem[++i] = _elem[j]; //发现不同元素时，向前移至紧邻于前者右侧
6     _size = ++i; shrink(); //直接截除尾部多余元素
7     return j - i; //向量规模变化量，即被删除元素总数
8 }
```

代码2.19 有序向量uniquify()接口的高效实现

图 2.7 为该算法的执行实例。

同样地，既然构成同一子区间
的重复元素物理地址连续，故只需
依次保留各区间的起始元素。

于是，这里借助变量 i 和 j ，
每经过若干次移动，都可使 i 和 j
分别指向下一对相邻子区间的首元
素；在将后者移动至前者的后继位
置之后，即可以重复上述过程。具
体地如图(a)，初始时 $i = 0$ 和 $j =$

1 分别指向最靠前两个元素。



图2.7 在有序向量中查找互异的相邻元素

接下来，逐位后移 j ，直至指向 $A[j=4] = 5 \neq A[i=0]$ 。如图(b)，此时可见， i 和 j 的确分别指向 3 和 5 所在分组的首元素。接下来，令 $i = 1$ ，并将 $A[j=4] = 5$ 前移至 $A[i=1]$ 处。此时的 i 指向刚被前移的 $A[1] = 5$ ；令 $j = j+1 = 5$ 指向待扫描的下一元素 $A[5] = 5$ ，并继续比较。如图(c)，此轮比较终止于 $A[j=9] = 8 \neq A[i=1] = 5$ 。

于是，令 $i = i+1 = 2$ ，并将 $A[j=9] = 8$ 前移至 $A[i=2]$ 处。此时的 i 指向刚被前移的 $A[2] = 8$ ；令 $j = j+1 = 10$ 指向待扫描的下一元素 $A[10] = 8$ ，并继续比较。如图(d)，此轮比较终止于 $A[12] = 13 \neq A[i=2] = 8$ 。于是，令 $i = i+1 = 3$ ，并将 $A[j=12] = 13$ 前移至 $A[i=3]$ 处。此时的 i 指向刚被前移的 $A[3] = 13$ ；令 $j = j+1 = 13$ 指向待扫描的

下一元素 $A[13] = 13$ ，并继续比较。如图(e)，至 $j = 16 \geq _size$ 时，循环结束。最后如图(f)，只需将向量规模更新为 $_size = i+1 = 4$ ，算法随即结束。鉴于在删除重复元素之后内部数组的空间利用率可能下降很多，故需调用 `shrink()`，如有必要则做缩容处理。

■ 复杂度

`while` 循环的每次迭代，仅需比较元素数值一次，向后移动一到两个位置指针，并至多向前复制一个元素，故只需常数时间。而在整个算法过程中，每经过一次迭代秩 j 都必然加一，鉴于 j 不能超过向量的规模 n ，故共需迭代 n 次。由此可知，`uniquify()` 算法的时间复杂度应为 $O(n)$ ，较之 `uniquifySlow()` 的 $O(n^2)$ ，效率整整提高了一个线性因子。

反过来，在遍历所有元素之前不可能确定向量是否已无重复元素，故就渐进复杂度而言，能在 $O(n)$ 时间内完成向量的唯一化处理已是最优方案了。当然，之所以能够做到这一点，关键在于向量已经排序。

2.6.4 查找

对于有序向量，查找操作可以更加高效地完成。为区别于无序向量的查找接口 `find()`，有序向量的查找接口统一命名为 `search()`。

与 `find()` 一样，代码 2.1 也针对有序向量的整体或区间查找定义了两个接口，且前者作为特例可直接调用后者。因此，只需如代码 2.20 所示实现其中的区间查找接口。

```
1 template <typename T> //在有序向量的区间[lo,hi)内，确定不大于e的最后一个节点的秩
2 Rank Vector<T>::search(T const& e, Rank lo, Rank hi) const { //assert: 0 <= lo < hi <= _size
3     return (rand() % 2) ? //按各50%的概率随机使用
4         binSearch(_elem, e, lo, hi) : fibSearch(_elem, e, lo, hi); //二分查找或Fibonacci查找
5 }
```

代码2.20 有序向量各种`search()`算法的统一接口

鉴于有序查找的算法多样且各具特点，为便于测试，这里的接口不妨随机选择查找算法。实际应用中可根据问题的特点具体确定，并做适当微调。以下介绍两类典型的查找算法。

2.6.5 二分查找（版本 A）

■ 减而治之

循秩访问的特点加上有序性，使得我们可将“减而治之”策略运用于有序向量的查找。

具体地如图 2.8 所示，假设在区间 $S[lo, hi)$ 中查找目标元素 e 。

以任一元素 $S[mi] = x$ 为界，(b)
都可将区间分为三部分，且根据此时
的有序性必有：

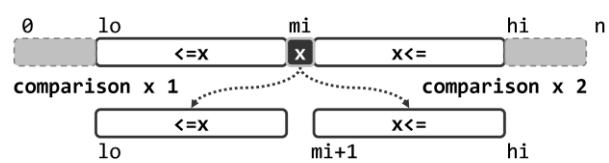


图2.8 基于减治策略的有序向量二分查找算法（版本A）

$$S[lo, mi] \leq S[mi] \leq S[mi+1, hi)$$

于是，只需将目标元素 e 与 x 做一比较，即可视比较结果分三种情况做进一步处理：1) 若 $e < x$ ，则目标元素如果存在，必属于左侧子区间 $S[lo, mi)$ ，故可深入其中继续查找；2) 若 $x < e$ ，则目标元素如果存在，必属于右侧子区间 $S[mi+1, hi)$ ，故也可深入其中继续查找；3) 若 $e = x$ ，则意味着已经在此处命中，故查找随即终止。

也就是说，每经过至多两次比较操作，我们或者已经找到目标元素，或者可以将查找问题简化为一个规模更小的新问题。如此，借助递归机制即可便捷地描述和实现此类算法。实际上，以下将要介绍的各种查找算法都可归入这一模式，不同点仅在于其对切分点 mi 的选取策略，以及每次深入递归之前所做比较操作的次数。

■ 实现

按上述思路实现的第一个算法如代码 2.21 所示。为区别于同类算法，不妨记作版本 A。

```
1 // 二分查找算法(版本A)：在有序向量的区间[lo, hi)内查找元素e, 0 <= lo <= hi <= _size
2 template <typename T> static Rank binSearch(T* A, T const& e, Rank lo, Rank hi) {
3     while (lo < hi) { //每次迭代可能要做两次比较判断，有三个分支
4         Rank mi = (lo + hi) >> 1; //以中点为轴点
5         if (e < A[mi]) hi = mi; //深入前半段[lo, mi)继续查找
6         else if (A[mi] < e) lo = mi + 1; //深入后半段[mi+1, hi)继续查找
7         else return mi; //在mi处命中
8     } //成功查找可以提前终止
9     return -1; //查找失败
10 } //有多个命中元素时，不能保证返回秩最大者；查找失败时，简单地返回-1，而不能指示失败的位置
```

代码2.21 二分查找算法(版本A)

为在有序向量的区间 $[lo, hi)$ 中查找元素 e ，该算法以中点 $mi = (lo+hi)/2$ 为界，将其大致平均地分为前、后两个子向量。随后通过一至两次比较操作，确定问题转化的方向。通过快捷的整数移位操作回避了相对更加耗时的除法运算。另外，通过引入 lo 、 hi 和 mi 等变量，将减治算法通常的递归模式改成了迭代模式。

■ 实例

如图 2.9 左侧所示，设在有序向量 $S[0, 7)$ 中查找目标元素 8。

第一次迭代如图(a1)，取 $mi = (0+7)/2 = 3$ ，经 2 次比较后深入后半段 $S[4, 7)$ 。第二次迭代如图(a2)，取 $mi = (4+7)/2 = 5$ ，经 1 次比较后深入前半段 $S[4, 5)$ 。最后一次迭代如图(a3)，经 2 次比较在 $mi = (4+5)/2 = 4$ 处成功命中。

(a1)  (b1) 

(a2)  (b2) 

(a3)  (b3) 

(b4)  (b4) 

图2.9 二分查找算法(版本A)实例：search(8, 0, 7)

图 2.9 右侧(b1~b4)给出了查找目标元素 3 的过程，虽属于查找失败，但原理同上。

■ 复杂度

尽可能在中点切分的益处在于，无论沿哪个方向转化，新问题的规模都将缩小一半，故此类算法亦称作二分查找 (binary search)。

也就是说，随着不断的迭代，有效的查找范围将按 $1/2$ 的比例以几何级数的速度递减；经至多 $\log_2(hi-lo)$ 次迭代后，算法必然终止。鉴于每次迭代仅需常数时间，故总体时间复杂度为 $\mathcal{O}(\log_2(hi-lo)) = \mathcal{O}(\log n)$ 。与代码 2.10 中顺序查找算法的 $\mathcal{O}(n)$ 复杂度相比， $\mathcal{O}(\log n)$ 几乎改进了一个线性因子。

■ 查找长度

以上迭代过程所涉及的计算主要包括元素的大小比较、秩的算术运算及赋值。就复杂度的常系数而言，元素比较操作的权重往往更大：复杂对象的比较操作往往非常耗时，某些场合中元素的比较操作甚至未必能在常数时间内完成（习题[13]）。因此，查找效率将主要取决于关键码的比较次数，即所谓查找长度（**search length**）。

通常，可针对查找成功或失败等情况，从最好、最坏和平均情况等角度分别进行评估。

■ 成功查找长度

仍以图 2.9 为例，在成功查找 $\text{search}(8, 0, 7)$ 的过程中，共需做 $2 + 1 + 2 = 5$ 次比较操作。实际上，成功查找长度的分布仅取决于有序向量的长度 n ，而与各元素的具体数值无关。

当 $n = 7$ 时由图 2.10 不难验证，各元素所对应的成功查找长度分别应为 $\{4, 3, 5, 2, 5, 4, 6\}$ 。若假定查找的目标元素按等概率分布，则平均查找长度为：

$$\begin{aligned} & (4 + 3 + 5 + 2 + 5 + 4 + 6) / 7 \\ & = 29 / 7 = 4.14 \end{aligned}$$

为估计一般情况的成功查找长度，不妨仍在等概率条件下考查长度 $n = 2^k - 1$ 的有序向量，并将其对应的最短、最长和平均成功查找长度分别记作 $c_{\text{best}}(k)$ 、 $c_{\text{worst}}(k)$ 和 $c_{\text{average}}(k)$ ，将所有元素对应的查找长度总和记作 $C(k) = c_{\text{average}}(k) \cdot (2^k - 1)$ 。在 $k=1$ 时，向量长度 $n=1$ ，成功查找仅有一种情况，故有边界条件：

$$c_{\text{best}}(1) = c_{\text{worst}}(1) = c_{\text{average}}(1) = C(1) = 2$$

按递推分析，对长度为 $n = 2^k - 1$ 向量的查找共有三个分支：经 1 次比较后转化为一个规模为 $2^{k-1} - 1$ 的新问题，其中包含最好情况对应的最短查找长度（图 2.10 中最左侧分支）；经 2 次比较后因成功而终止；经 2 次比较后转化为另一个规模为 $2^{k-1} - 1$ 的新问题，其中包含最坏情况下的最长查找长度（图 2.10 中最右侧分支）。由此，可得递推式如下：

$$\begin{aligned} c_{\text{best}}(k) &= c_{\text{best}}(k-1) + 1, \quad c_{\text{worst}}(k) = c_{\text{worst}}(k-1) + 2 \\ C(k) &= [C(k-1) + (2^{k-1}-1)] + 2 + [C(k-1) + 2 \times (2^{k-1}-1)] \dots \quad (\text{式 2-1}) \end{aligned}$$

结合以上边界条件，可以解得：

$$c_{\text{best}}(k) = k, \quad c_{\text{worst}}(k) = 2k$$

$$C(k) = 2 \cdot C(k-1) + 3 \cdot 2^{k-1} - 1 =^{\circledR} (3k-2) \cdot 2^{k-1} + 1$$

$$c_{\text{average}}(k) = C(k) / 2^{k-1} = 3k/2 - 1 + 3k/2/(2^{k-1}) = 3k/2 - 1 + O(\varepsilon)$$

也就是说，忽略末尾趋于收敛的波动项，平均查找长度为 $O(1.5k) = O(1.5 \cdot \log_2 n)$ 。

■ 失败查找长度

按照代码 2.21，失败查找的终止条件必然是“ $lo \geq hi$ ”，也就是说，只有在有效区

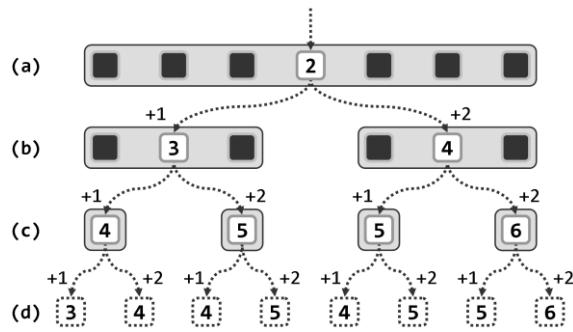


图2.10 二分查找算法（版本A）的查找长度
(成功、失败查找分别以实线、虚线白色方框示意)

^① 令 $F(k) = C(k) - 3k \cdot 2^{k-1} - 1$ ，则有 $F(k) = 2 \cdot F(k-1) = -2^k$

间宽度缩减至 0 时方能终止。因此，失败查找的时间复杂度应为稳定的 $\Theta(\log n)$ 。不难发现，就查找过程中分支方向的组合而言，失败查找可能的情况恰好比成功查找多一种。仍以图 2.10 为例，共有 $7 + 1 = 8$ 种失败的查找。如图 2.10 所示，其对应的查找长度分别为 {3, 4, 4, 5, 4, 5, 5, 6}。其中，最好、最坏情况下分别需要做 3 次、6 次元素比较。若假定查找的目标元素按等概率分布，则平均查找长度为：

$$(3 + 4 + 4 + 5 + 4 + 5 + 5 + 6) / 8 = 36 / 8 = 4.50$$

■ 不足

尽管二分查找算法（版本 A）即便在最坏情况下也可保证 $\mathcal{O}(\log n)$ 的渐进时间复杂度，但就其常系数 1.5 而言，仍有改进余地。以成功查找为例，即便是迭代次数相同的情况下，查找长度也不尽相等。究其根源在于，在每次迭代中，为确定左、右分支方向而需做的元素比较次数分别为 1 和 2 次，从而造成不同情况所对应查找长度的不均衡。尽管该版本从表面上看完全均衡，但以上分析已确凿地表明，最短和最长分支所对应的查找长度相差约两倍。

那么，能否实现更好的均衡呢？具体又应如何实现呢？

2.6.6 Fibonacci 查找

■ 递推方程

递推方程法既是复杂度分析的重要方法，也是我们优化算法时确定突破口的有力武器。为改进以上二分查找算法的版本 A，不妨从刻画查找长度随向量长度递推变化的式 2-1 入手。

实际上，最终求解所得平均复杂度在很大程度上取决于这一等式，准确地讲，主要取决于 $(2^{k-1} - 1)$ 和 $2 \times (2^{k-1} - 1)$ 两项，其中的系数 1 和 2 就是算法为深入前、后子向量而需做的比较操作次数，而 $(2^{k-1} - 1)$ 则是子向量的宽度。此前所指出的二分查找算法版本 A 的均衡性缺陷，根源正在于这两项大小不匹配。理解到这一层后，就不难提出解决问题的方法。具体地不外乎两种：其一，调整前、后区域的宽度，适当加长（缩短）前（后）子向量；其二，统一沿两个方向深入所做的比较次数，比如都统一为一次。

后一思路的实现将在稍后介绍，以下首先介绍前一思路的具体实现。

■ Fibonacci 查找

实际上，减治策略本身并不要求子向量切分点 mi 必须居中，故按上述改进思路，不妨按黄金分割比来确定 mi 。为简化起见，不妨设向量长度 $n = \text{fib}(k) - 1$ 。

于是如图 2.11 所示，
`fibSearch(e, 0, n)` 查找可
以 $mi = \text{fib}(k-1)-1$ 作为前、
后子向量的切分点。如此，前、
后子向量的长度将分别是：
$$\begin{array}{c} \text{(a)} \quad \begin{array}{ccccccc} 0 & & \text{fib}_{k-1}-1 & & \text{fib}_k-1 \\ \text{fib}(k-1) - 1 & x & \text{fib}(k-2) - 1 & & & & \\ \text{comparison } x 1 & & & & \text{comparison } x 2 & & \end{array} \\ \text{(b)} \quad \begin{array}{ccccc} 0 & & \text{fib}_{k-1} & & \text{fib}_k-1 \\ \text{fib}(k-1) - 1 & & & & \text{fib}(k-2) - 1 \\ & & & & \end{array} \end{array}$$

$$\text{fib}(k-1) - 1$$

$$\text{fib}(k-2) - 1 = (\text{fib}(k) - 1) - (\text{fib}(k-1) - 1) - 1$$

于是，无论朝哪个方向深入，新向量的长度从形式上都依然是某个 Fibonacci 数减一，故这一处理手法可以反复套用，直至因在 $S[mi]$ 处命中或向量长度收缩至零而终止。这种查找算法故此称作 Fibonacci 查找（Fibonaccian search）。

图 2.11 Fibonacci 查找算法原理

■ 实现

按照以上思路，可实现 Fibonacci 查找算法如代码 2.22 所示。

```
1 #include "../fibonacci/Fib.h" //引入Fib数列类
2 // Fibonacci查找算法（版本A）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= _size
3 template <typename T> static Rank fibSearch(T* A, T const& e, Rank lo, Rank hi) {
4     Fib fib(hi - lo); //用O(log_phi(n=hi-lo))时间创建Fib数列
5     while (lo < hi) { //每次迭代可能要做两次比较判断，有三个分支
6         while (hi - lo < fib.get()) fib.prev(); //通过向前顺序查找（分摊O(1)）——至多迭代几次？
7         Rank mi = lo + fib.get() - 1; //确定形如Fib(k)-1的轴点
8         if (e < A[mi]) hi = mi; //深入前半段[lo, mi]继续查找
9         else if (A[mi] < e) lo = mi + 1; //深入后半段[mi+1, hi)继续查找
10        else return mi; //在mi处命中
11    } //成功查找可以提前终止
12    return -1; //查找失败
13 } //有多个命中元素时，不能保证返回秩最大者；失败时，简单地返回-1，而不能指示失败的位置
```

代码2.22 Fibonacci查找算法

算法主体框架与二分查找大致相同，主要区别在于以黄金分割点取代中点作为切分点。为此，需要借助一个 Fib 对象（习题[15]）实现对 Fibonacci 数的高效设置与获取。

尽管以下的分析多以长度为 $\text{fib}(k) - 1$ 的向量为例，但这一实现完全可适用于长度任意的向量中的任意子向量。为此，只需在进入循环之前调用构造器 $\text{Fib}(n = \text{hi} - \text{lo})$ ，将初始长度设置为“不小于 n 的最小 Fibonacci 项”。这一步所需花费的 $O(\log_{\phi} n)$ 时间，分摊到后续的 $O(\log_{\phi} n)$ 次迭代中，并不影响算法整体的渐进复杂度。

■ 定性比较

可见，Fibonacci 查找倾向于适当加长（缩短）需 1 次（2 次）比较方可确定的 (a) 前端（后端）子向量。故定性估计，应可在（在常系数的意义上）进一步提高查找的效率。 (b)

为验证这一推断，不妨仍以 $n = \text{fib}(6) - 1 = 7$ 为例，就平均查找长度与二分查找做一对比。如图 2.12，7 种成功情况的查找长度分别为 $\{5, 4, 3, 5, 2, 5, 4\}$ ，(c) 8 种失败情况的查找长度分别为 $\{4, 5, 4, 4, 5, 4, 5, 4\}$ 。若依然假定各成功情况等概率，各失败情况亦等概率，则平均成功查找长度和平均失败查找长度分别为：

$$(5 + 4 + 3 + 5 + 2 + 5 + 4) / 7 = 28 / 7 = 4.00$$

$$(4 + 5 + 4 + 4 + 5 + 4 + 5 + 4) / 8 = 35 / 8 = 4.38$$

与 2.6.5 节二分查找算法（版本 A）对应的实例（图 2.10）的 4.14 和 4.50 相比，平均查找长度的确均有所改进。

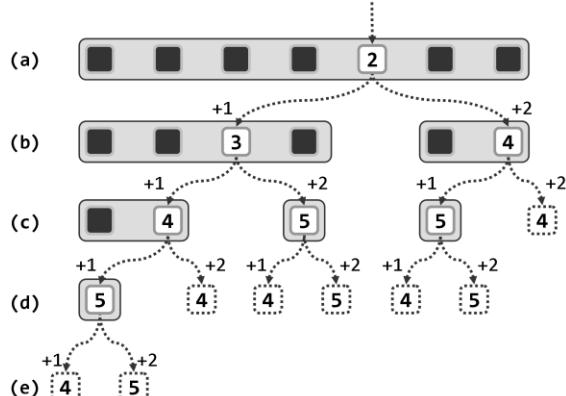


图2.12 Fibonacci查找算法的查找长度
(成功、失败查找分别以实线、虚线白色方框示意)

■ 定量分析

参照 2.6.5 节的方法，也可对 Fibonacci 查找算法的成功查找长度做出最更为精确的分析。其中关于最好、最坏情况的结论完全一致，故以下仅讨论等概率条件下的平均情况。

依然将长度为 $n = \text{fib}(k) - 1$ 的有序向量的平均成功查找长度记作 $c_{\text{average}}(k)$ ，将所有元素对应的查找长度总和记作 $C(k) = c_{\text{average}}(k) \cdot (\text{fib}(k) - 1)$ 。

同理，可得边界条件及递推式如下：

$$c_{\text{average}}(2) = C(2) = 0, c_{\text{average}}(3) = C(3) = 2$$

$$\begin{aligned} C(k) &= [C(k-1) + (\text{fib}(k-1) - 1)] + 2 + [C(k-2) + 2 \times (\text{fib}(k-2) - 1)] \\ &= C(k-2) + C(k-1) + \text{fib}(k-2) + \text{fib}(k) - 1 \end{aligned}$$

结合以上边界条件，可以解得：

$$C(k) = ^{\circledast} k \cdot \text{fib}(k) - \text{fib}(k+2) + 1 = (k - \Phi^2) \cdot \text{fib}(k) + 1 + O(\varepsilon)$$

$$\text{其中, } \Phi = (\sqrt{5} + 1) / 2 = 1.618$$

于是

$$\begin{aligned} c_{\text{average}}(k) &= C(k) / (\text{fib}(k) - 1) \\ &= k - \Phi^2 + 1 + (k - \Phi^2) / (\text{fib}(k) - 1) + O(\varepsilon) = k - \Phi^2 + 1 + O(\varepsilon) \end{aligned}$$

也就是说，忽略末尾趋于收敛的波动项，平均查找长度的增长趋势为：

$$O(k) = O(\log_{\Phi} n) = O(\log_2 2 \cdot \log_2 n) = O(1.44 \cdot \log_2 n)$$

较之 2.6.5 节二分查找算法（版本 A）的 $O(1.50 \cdot \log_2 n)$ ，效率略有提高。

2.6.7 二分查找（版本 B）

■ 从三分支到两分支

2.6.6 节开篇曾指出，二分查找算法版本 A 的不均衡性体现为复杂度递推式中 $(2^{k-1}-1)$ 和 $2 \times (2^{k-1}-1)$ 两项的不均衡。为此，Fibonacci 查找算法已通过采用黄金分割点，成功地降低了时间复杂度的常系数。实际上还有另一更为直接的方法，即令以上两项的常系数同时等于 1。也就是说，无论朝哪个方向深入，都只需做 1 次元素的大小比较。相应地，算法在每次迭代中（或递归层次上）都只有两个分支方向，而不再是三个。

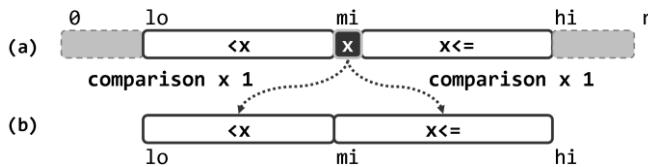


图2.13 基于减治策略的有序向量二分查找算法（版本 B）

具体过程如图 2.13 所示，完全类似于二分查找算法。在每个切分点 $A[mi]$ 处，仅做一次元素比较。若 $e < A[mi]$ ，则深入前端子向量 $A[lo, mi]$ 继续查找；否则，深入后端子向量 $A[mi, hi]$ 继续查找。

■ 实现

按照上述思路，可将二分查找算法改进为如代码 2.23 所示的版本 B。

^② 令 $F(k) = -C(k) + k \cdot \text{fib}(k) + 1$ ，则有 $F(0) = 1, F(1) = 2, F(k) = F(k-1) + F(k-2) = \text{fib}(k+2)$

```
1 // 二分查找算法 (版本B) : 在有序向量的区间[lo, hi)内查找元素e, 0 <= lo <= hi <= _size
2 template <typename T> static Rank binSearch(T* A, T const& e, Rank lo, Rank hi) {
3     while (1 < hi - lo) { //每次迭代仅需做一次比较判断, 有两个分支; 成功查找不能提前终止
4         Rank mi = (lo + hi) >> 1; //以中点为轴点
5         (e < A[mi]) ? hi = mi : lo = mi; //经比较后确定深入[lo, mi)或[mi, hi)
6     } //出口时hi = lo + 1, 查找区间仅含一个元素A[lo]
7     return (e == A[lo]) ? lo : -1; //查找成功时返回对应的秩; 否则统一返回-1
8 } //有多个命中元素时, 不能保证返回秩最大者; 查找失败时, 简单地返回-1, 而不能指示失败的位置
```

代码2.23 二分查找算法 (版本B)

可见, 这里依然以查找区间的中点作为切分点。与版本 A 的差异在于, 每次仅比较一次, 即可相应地更新有效区间的右边界 (`hi = mi`) 或左边界 (`lo = mi`)。

■ 性能

尽管版本 B 中的后端子向量需要加入 `A[mi]`, 但得益于 `mi` 总是位于中央位置, 整个算法 $\mathcal{O}(\log n)$ 的渐进复杂度不受任何影响。

在这一版本中, 只有在向量有效区间宽度缩短至 1 时算法才会终止, 而不能如版本 A 那样可在命中时及时返回。因此, 最好情况下的效率有所倒退。当然, 作为补偿, 最坏情况下的效率相应地有所提高。实际上无论是成功查找或失败查找, 版本 B 各分支的查找长度更加接近, 故整体性能更趋稳定。

■ 进一步的要求

在更多细微之处, 此前实现的二分查找算法 (版本 A 和 B) 及 Fibonacci 查找算法仍有改进的余地。比如, 当目标元素在向量中重复出现时, 它们只能“随机”地报告其一, 具体选取何者取决于算法的分支策略以及当时向量的组成。而在很多场合中, 重复元素之间往往会有隐含地定义有某种优先级次序, 而且算法调用者的确可能希望得到其中优先级最高者。比如, 根据表 2.1 所定义向量的 `search()` 接口, 在有多个命中元素时, 应该以它们的秩为优先级, 并返回其中最靠后者。

这种进一步的要求并非多余。以有序向量的元素插入操作为例, 若通过查找操作不仅能够确定适当的插入位置, 而且在同时存在多个可行的插入位置时能保证返回其中秩最大者, 则不仅能减少需移动的后继元素, 更可保证重复的元素按其插入的相对次序排列。这对于向量的插入排序等算法 (第 3 章习题[7]) 的稳定性 (`stability`) 至关重要。

另外, 对失败查找的处理方式也可以改进。查找失败时, 以上算法都是简单地统一返回一个标识 “-1”。同样地, 若在插入新元素 `e` 之前通过查找确定适当的插入位置, 则希望在查找失败时返回不大 (小) 于 `e` 的最后 (前) 一个元素, 以便将 `e` 作为其后继 (前驱) 插入向量。同样地, 此类约定也使得插入排序等算法的实现更为便捷和自然。

2.6.8 二分查找 (版本 C)

■ 实现

在版本 B 的基础上略作修改, 即可得到如代码 2.24 所示二分查找算法的版本 C。

```
1 // 二分查找算法 (版本C) : 在有序向量的区间[lo, hi)内查找元素e, 0 <= lo <= hi <= _size
2 template <typename T> static Rank binSearch(T* A, T const& e, Rank lo, Rank hi) {
```

```
3     while (lo < hi) { //每次迭代仅需做一次比较判断，有两个分支
4         Rank mi = (lo + hi) >> 1; //以中点为轴点
5         (e < A[mi]) ? hi = mi : lo = mi + 1; //经比较后确定深入[lo, mi]或[mi+1, hi)
6     } //成功查找不能提前终止
7     return --lo; //循环结束时，lo为大于e的元素的最小秩，故lo-1即不大于e的元素的最大秩
8 } //有多个命中元素时，总能保证返回秩最大者；查找失败时，能够返回失败的位置
```

代码2.24 二分查找算法（版本C）

该版本的主体结构与版本B一致，故不难理解，二者的时间复杂度相同。

■ 正确性

从形式上看，版本C与版本B的差异主要有三点。首先，只有当向量有效区间的宽度缩短至0（不再是1）时，迭代及算法才告终止。另外，在每次转入后端分支时，子向量的左边界取作 $mi+1$ （不再是 mi ）。表面上看，后一调整可能存在错误的风险——此时只能确定切分点 $A[mi] \leq e$ ，而“贸然”将 $A[mi]$ 排除在进一步的查找范围之外，似乎会因遗漏目标元素而导致本应成功的查找以失败告终。然而这种担心大可不必。

版本C的正确性源自其中循环体的如下不变性：如图2.14所示， $A[0, lo](A[hi, n])$ 中元素皆不大于（大于） e 。首次迭代时 $lo = 0$ 且 $hi = n$ ， $A[0, lo]$ 和 $A[hi, n]$ 均为空，不变性自然成立。以下数学归纳，设如图(a)在某次进入循环时以上不变性成立，根据算法流程无非两种情况。若 $e < A[mi]$ ，则如图(b)，在令 $hi = mi$ 并使 $A[hi, n]$ 向左扩展之后，该区间内的元素皆不小于 $A[mi]$ ，当然也仍然大于 e 。反之，若 $A[mi] \leq e$ ，则如图(c)，在令 $lo = mi+1$ 并使 $A[0, lo]$ 向右拓展之后，该区间内的元素皆不大于 $A[mi]$ ，当然也仍然不大于 e 。总之无论如何，每次循环之后上述不变性依然得以延续，并直至循环终止时。

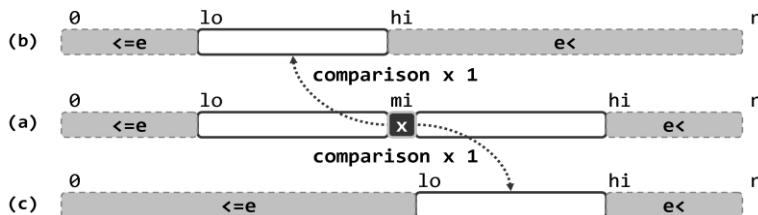


图2.14 基于减治策略的有序向量二分查找算法（版本C）

循环终止时， $lo = hi$ 。考查此时的元素 $A[lo-1]$ 和 $A[lo]$ ：作为 $A[0, lo]$ 内的最后一个元素， $A[lo-1]$ 必不大于 e ；作为 $A[lo, n] = A[hi, n]$ 内的第一个元素， $A[lo]$ 必大于 e 。也就是说， $A[lo-1]$ 即是原向量中不大于 e 的最后一个元素。因此在循环结束之后，无论成功与否，只需返回 $lo-1$ 即可——这也是版本C与版本B的第三点差异。

§ 2.7 *排序与下界

2.7.1 有序性

“天之道，损有余而补不足”，自然万物发展的规律，大体倾向于消除差异。无独有偶，热力学第二定律也指出：任一封闭系统都会朝着熵增加的方向发展。从信息论的角度来看，也就是倾向于更加无序。然而，“人之道，则不然，损不足以奉有余”，人总是偏好有序。

从数据处理的角度看，有序性在很多场合都能极大地提高计算的效率。以查找算法为例，对于无序向量而言，在最坏情况下，在每一元素都至少被检查过之前，将不能判定待查找的元素是否存在，因此如代码 2.10 所实现 `Vector::find()` 接口的 $\mathcal{O}(n)$ 复杂度已属最优。而对于有序向量，如代码 2.20 所实现的 `Vector::search()` 接口的效率则可优化到 $\mathcal{O}(\log n)$ 。实际上，此时之所以可以实施二分查找，正是因为所有元素已按次序排列。

2.7.2 排序及其分类

由以上介绍可见，有序向量的诸如查找等操作，效率远高于一般向量。因此在解决许多应用问题时我们普遍采用的一种策略就是，首先将向量转换为有序向量，再调用有序向量支持的各种高效算法。这一过程的本质就是向量的排序。为此，正如 2.6.1 节所指出的，向量元素之间必须能够定义某种全序关系，以保证它们可相互比较大小。

排序算法是个庞大的家族，可从多个角度对其中的成员进行分类。比如，根据其处理数据的规模与存储的特点不同，可分为内部和外部排序算法：前者处理的数据规模相对不大，内存足以容纳；后者处理的数据规模很大，必须将借助外部甚至分布式存储器，在排序计算过程的任一时刻，内存中只能容纳其中一小部分数据。又如，根据输入形式的不同，排序算法也分为离线算法（**offline algorithm**）和在线算法（**online algorithm**）。前一情况下，待排序的数据是以批处理的形式整体给出的；而在网络计算之类的环境中，待排序的数据通常需要实时生成，在排序算法启动后数据才陆续到达。再如，针对所依赖的体系结构不同，又可分为串行和并行两大类排序算法。另外，根据排序算法是否采用随机策略，还有确定式和随机式之分。本书讨论的范围，主要集中于确定式串行脱机的内部排序算法。

2.7.3 下界

根据 1.2.2 节的分析，1.1.3 节起泡排序算法的复杂度为 $\mathcal{O}(n^2)$ 。那么，这一效率是否已经足够高？能否以更快的速度完成排序？实际上，在着手优化算法之前，这都是首先必须回答的问题。以下结合具体实例，从复杂度下界的角度介绍回答此类问题的一般性方法。

■ 苹果鉴别

考虑如下问题：三只苹果外观一样，其中两只重量相同另一只不同，利用一架天平如何从中找出重量不同的那只？一种直观的方法可以描述为算法 2.1。

```
identifyApple(A, B, C)
输入：三只苹果A、B和C，其中两只重量相同，另一只不同
输出：找出重量不同的那只苹果
{
    称量A和B；若A和B重量相等，则返回C；
    称量A和C；若A和C重量相等，则返回B；
    否则，返回A；
}
```

算法2.1 从三个苹果中选出重量不同者

该算法的可行性、正确性毋庸置疑。该算法在最好情况下仅需执行一次比对操作，最坏情况下两次。那么，是否存在其它算法，即便在最坏情况下也至多只需一次比对呢？

■ 复杂度下界

尽管很多算法都可以优化，但有一个简单的事实却往往为人所忽略：对任一特定的应用问题，随着算法的不断改进，其效率的提高必然存在某一极限。毕竟，我们不能奢望不劳而获。这一极限不仅必然存在，而且其数值应取决于应用问题本身和采用的计算模型。

比如上例中所提出的问题，就是从最坏情况的角度，质疑“2次比对操作”是否为解决这一问题的最低复杂度。一般地，在任一应用问题的所有算法（如果的确存在的话）中，最坏情况下的最低时间复杂度亦称作该问题的复杂度下界（lower bound）。一旦算法的性能达到对应问题的下界，就意味着该算法就渐进意义而言已是最坏情况下最优的（worst-case optimal）。可见，尽早确定一个问题的复杂度下界，对相关算法的优化无疑会有巨大的裨益。以下结合比较树模型，介绍界定问题复杂度下界的一种重要方法。

2.7.4 比较树

■ 基于比较的分支

如果用节点（圆圈）表示算法过程中的不同状态，用有方向的边（直线段或弧线段）表示不同状态之间的相互转换，就可以将以上算法 2.1 转化为图 2.15 的树形结构（第 5 章）。

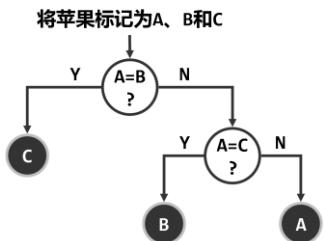


图 2.15 从三只苹果中挑出重量不同者

这一转化方法也推广并适用于多数其它算法。一般地，树根节点对应于算法入口处的起始状态（如此处三个苹果已做好标记）；内部节点（即非末端节点，图中以白色大圈示意）对应于过程中的某步计算，通常属于基本操作；叶节点（即末端节点，图中以黑色小圈示意）则对应于经一系列计算后某次运行的终止状态。如此借助这一树形结构，可以涵盖对应算法所有可能的执行流程。

仍以图 2.15 为例，从根节点到叶节点 C 的路径对应于，在经过一次称量比较并确定 A 与 B 等重后，即可断定 C 是所要查找的苹果。再如，从根节点到叶节点 B 的路径对应于，在经过两次称量比较并确定 A 与 B 不等重、A 与 C 等重之后，即可判定 B 是所要查找的苹果。

■ 比较树

概括而言，以上树形表示形式具有以下特点：

- ① 算法所有可能的执行过程，都可涵盖于这一树形结构中；
- ② 每一内部节点各对应于一次比对（称量）操作；
- ③ 内部节点的左、右分支，分别对应于在两种比对结果（是否等重）下的执行方向；
- ④ 叶节点（或等效地，根到叶节点的路径）对应于算法某次执行的完整过程及其输出；
- ⑤ 反过来，算法的每一运行过程都对应于从根到某一叶节点的路径。

按上述规则与算法相对应的树，称作比较树（comparison tree）。

不难理解，无论什么算法，只要其中所有分支都如算法 2.1 那样完全取决于两个变量（或常量）的相等比对（或大小比较）结果，则该算法所有可能的执行过程都可表示和概括为一棵比较树。反之，凡可如此描述的算法，都称作基于比较式算法（comparison-based algorithm），简称 CBA 式算法。比如在本书中，除散列之外的算法大多属于此类。

以下我们将看到，CBA 式算法在最坏情况下的最低执行成本，由对应的比较树界定。

2.7.5 估计下界

■ 最小树高

考查任一 CBA 式算法 A, 设 CT(A) 为与之对应的比较树。由比较树的性质, 算法 A 每次运行所需的时间, 将取决于其对应叶节点到根节点的距离(称作叶节点的深度); 而算法 A 在最坏情况下的运行时间, 将取决于比较树中所有叶节点的最大深度(称作该树的高度, 记作 $h(CT(A))$)。因此就渐进的意义而言, 算法 A 的时间复杂度应不低于 $\Omega(h(CT(A)))$ 。

对于存在 CBA 式算法的计算问题, 既然其任一 CBA 式算法均对应于某棵比较树, 该问题的复杂度下界就应等于这些比较树的最小高度。问题在于, 如何估计这一最小高度呢?

界定树高的一种主要方法是通过树中包含的叶节点数目, 即可能的输出结果数目。具体地, 在一棵高度为 h 的二叉树中, 叶节点的数目不可能多于 2^h 。反之若某一问题的输出结果不少于 N 种, 则比较树中叶节点也不可能少于 N 个, 树高 h 不可能低于 $\log_2 N$ 。

■ 苹果鉴别

仍以算法 2.1 为例。就该问题而言, 可能的输出结果共计 $N = 3$ 种(不同的苹果分别为 A、B 或 C), 故解决该问题的任一 CBA 式算法所对应比较树的高度为

$$h \geq \lceil \log_2 3 \rceil = 2$$

因此, 只要是采用 CBA 式算法来求解该问题, 则无论如何优化, 在最坏情况下都至少需要 2 次称量——尽管最好情况下的确仍可能仅需 1 次。这也意味着, 算法 2.1 虽朴实无华, 却已是解决苹果鉴别问题的最佳 CBA 式算法。

■ 排序

再以 CBA 式排序算法为例。就 n 个元素的排序问题而言, 可能的输出共有 $N = n!$ 种。与上例略有不同之处在于, 元素之间不仅可以判等而且可以比较大小, 故此时的比较树应属于三叉树, 即每个内部节点都有三个分支(分别对应小于、相等和大于的情况)。不过, 这并不影响上述分析方法的运用。按照以上思路, 任一 CBA 式排序算法所对应比较树的高度

$$h \geq \lceil \log_3(n!) \rceil = \lceil \log_3 e \cdot \ln(n!) \rceil =^{\circ} \Omega(n \log n)$$

可见, 最坏情况下 CBA 式排序算法至少需要 $\Omega(n \log n)$ 时间, 其中 n 为待排序元素数目。

需强调的是, 这一 $\Omega(n \log n)$ 下界是针对比较树模型而言的。事实上, 还有很多不属此类的排序算法(比如 9.4.1 节的桶排序算法和 9.4.3 节的基数排序算法), 且其中一些算法的最坏情况运行时间可能低于这一下界, 但与上述结论并不矛盾。

§ 2.8 排序器

2.8.1 统一入口

鉴于排序在算法设计与实际应用中的重要地位和作用, 排序操作自然应当纳入向量基本接口的范围。这类接口也是将无序向量转换为有序向量的基本方法和主要途径。

代码 2.25 给出了向量排序器的统一入口, 可对向量的任意合法区间做排序, 其中提供了起泡排序、归并排序、快速排序和堆排序等多种算法。为测试对比, 这里暂以随机方式确

^③ 由 Stirling 逼近公式, $n! \sim \sqrt{2\pi n} \cdot (n/e)^n$

定每次调用的具体算法，在具体应用中可根据应用需求与算法特点灵活选用。

```
1 template <typename T> void Vector<T>::sort(Rank lo, Rank hi) { //向量区间[lo, hi)排序
2     switch (rand() % 4) { //随机选取排序算法。可根据具体问题的特点灵活选取或扩充
3         case 1: bubbleSort(lo, hi); break; //起泡排序
4         case 2: mergeSort(lo, hi); break; //归并排序
5         case 3: heapSort(lo, hi); break; //堆排序（稍后介绍）
6         default: quickSort(lo, hi); break; //快速排序（稍后介绍）
7     }
8 }
```

代码2.25 向量排序器接口

以下首先将起泡排序算法集成至向量 ADT 中，然后讲解归并排序算法的原理、实现并分析其复杂度，堆排序和快速排序则分别留待 10.2 节和 12.1 节再做介绍。

2.8.2 起泡排序

起泡排序算法已在 1.1.3 节讲解并实现，这里只需将其集成至向量 ADT 中。

■ 起泡排序

```
1 template <typename T> //向量的起泡排序
2 void Vector<T>::bubbleSort(Rank lo, Rank hi) //assert: 0 <= lo < hi <= size
3 { while (!bubble(lo, hi--)); } //逐趟做扫描交换，直至全序
```

代码2.26 向量的起泡排序

代码 2.26 给出了起泡排序的主体框架，其功能等效于代码 1.1 中的外层循环：反复调用单趟扫描交换算法，直至逆序现象完全消除。

■ 扫描交换

```
1 template <typename T> bool Vector<T>::bubble(Rank lo, Rank hi) { //一趟扫描交换
2     bool sorted = true; //整体有序标志
3     while (++lo < hi) //自左向右，逐一检查各对相邻元素
4         if (_elem[lo-1] > _elem[lo]) { //若逆序，则
5             sorted = false; //意味着尚未整体有序，并需要
6             swap(_elem[lo-1], _elem[lo]); //通过交换使局部有序
7         }
8     return sorted; //返回有序标志
9 }
```

代码2.27 单趟扫描交换

单趟扫描交换算法如代码 2.27 所示，其功能等效于代码 1.1 中的内层循环：依次比较所有相邻元素，一旦逆序则做交换；一旦经某趟扫描后未发现任何逆序现象，则通过返回标志“sorted”告知上一层次，以便及时终止算法。

■ 重复元素与稳定性

稳定性（**stability**）是对排序算法更为细致的一项要求，它重在考查算法对重复元素的处理效果，要求尽可能不打乱大小相等的重复元素。

在采用某一算法对向量 A 排序并生成有序向量 S 之后，假设 A[i] 对应于 S[k_i]。若对

于 A 中每一对重复元素 $A[i] = A[j]$ (以及对应的 $S[k_i] = S[k_j]$)，都有 $i < j$ 当且仅当 $k_i < k_j$ ，则称该排序算法是稳定算法 (**stable algorithm**)。简而言之，稳定算法的特征是，重复元素之间的相对次序在排序前后保持一致。反之，不具有这一特征的排序算法都是不稳定算法 (**unstable algorithm**)。

依此角度反观起泡排序可以发现，该算法过程中元素相对位置有所调整的唯一可能是，某元素 $\text{elem}[i-1]$ 严格大于其后继 $\text{elem}[i]$ 。也就是说，在这种亦步亦趋的交换过程中，重复元素虽可能相互靠拢，但绝对不会相互跨越。由此可知，起泡排序属于稳定算法。

2.8.3 归并排序

■ 历史与发展

归并排序^④ (**mergesort**) 的构思朴实却亦深刻，作为一个算法既古老又仍不失生命力。在排序算法发展的历史上，归并排序具有特殊的地位，它是第一个可以在最坏情况下依然保持 $O(n \log n)$ 运行时间的确定性排序算法。时至今日，当计算机在早期发展过程中曾经遇到的一些难题从某种意义上讲再次呈现时，归并排序又重新焕发了青春。比如，早期计算机的存储能力有限，以至于或者高速存储器不能容纳所有的数据，或者只能使用磁带机或卡片之类的顺序存储设备，这些促进了归并排序的诞生，同时也为归并排序大显身手提供了舞台。信息化无处不在的今天，我们再次发现，人类所拥有信息之庞大，不仅迫使我们更多地将它们存放和组织于分布式平台之上，而且对海量信息的处理也必须首先考虑如何在跨节点的环境中高效地协同计算，而许多新算法和技术的背后都可以看到归并排序的影子。

■ 有序向量的二路归并

与起泡排序通过反复调用单趟扫描交换类似，归并排序也可以理解为是通过反复调用所谓二路归并 (**2-way merge**) 算法而实现的。所谓二路归并，就是将两个有序序列合并成为一个有序序列。这里的序列既可以是向量，也可以是第 3 章将要介绍的列表。这一算法的执行时间也主要消耗于各趟二路归并之中。这里首先考虑有序向量。

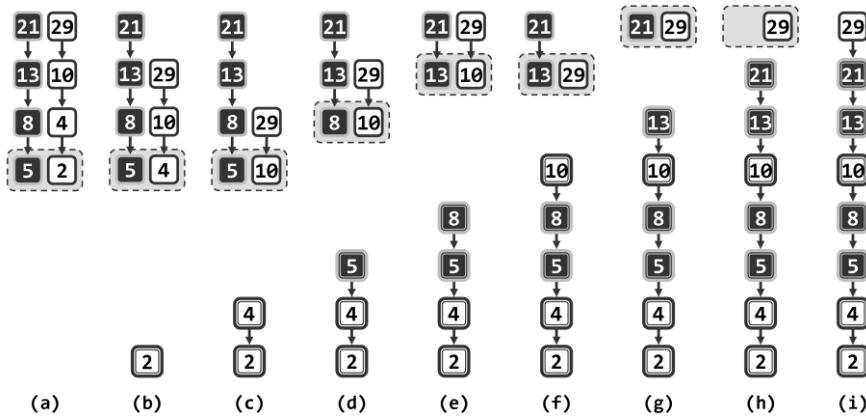


图2.16 有序向量的二路归并实例

(来自两个向量的元素分别以黑、白方框区分，其各自的当前首元素则以灰色长方形示意)

^④ 1945 年由冯·诺依曼在 EDVAC 上首次编程实现

二路归并通过迭代，将各元素按最终次序逐一取出并归入输出向量。每一次迭代中，都需首先比较待归并向量的首元素，再将小者取出并追加到输出向量的末尾，最后更新原向量的首元素记录以便继续迭代。

如图 2.16(a)，考查待归并的有序向量{5, 8, 13, 21}和{2, 4, 10, 29}。首轮迭代经比较，取出右侧向量首元素 2 并归入输出向量，同时原向量首元素更新为 4（图(b)）。此后各次迭代类似，反复比较首元素，将小者取出。如此，即可最终实现整体归并（图(i)）。

可见，该算法在任何时刻只需载入两个向量的首元素，故总体只需要常数规模的内存。因此，无论待排序数据来自磁带机等顺序存储器，还是来自磁盘等随机存储器，均十分高效。

■ 分治策略

归并排序的主体结构属典型的分治策略，可递归地描述和实现如代码 2.28 所示。

```
1 template <typename T> //向量归并排序
2 void Vector<T>::mergeSort(Rank lo, Rank hi) { //assert: 0 < lo <= hi <= size
3     if (hi - lo < 2) return; //单元素区间自然有序，否则...
4     int mi = (lo + hi) >> 1; //以中点为界
5     mergeSort(lo, mi); mergeSort(mi, hi); merge(lo, mi, hi); //分别对前、后半段排序，然后归并
6 }
```

代码2.28 向量的归并排序

为将向量 $S[lo, hi]$ 转换为有序向量，可均匀地将其划分为两个子向量 $S[lo, mi]$ 和 $S[mi, hi]$ 。以下，只要通过自我递归调用将它们分别转换为有序向量，即可借助二路归并算法，得到与原向量 S 对应的整个有序向量。请注意，这里的递归终止条件是当前向量长度 $n = hi - lo = 1$ 。既然仅含单个元素的向量自然有序，这一处理分支的确可作为递归基。

■ 实例

图 2.17 以向量 $S = \{6, 3, 2, 7, 1, 5, 8, 4\}$ 为例，给出了归并算法的完整执行过程。从递归的角度，也可将图 2.17 看作对整个算法执行过程的递归跟踪，其中给出了算法执行期间出现过的所有递归实例，并按照递归调用关系将其排列成一个层次化结构。

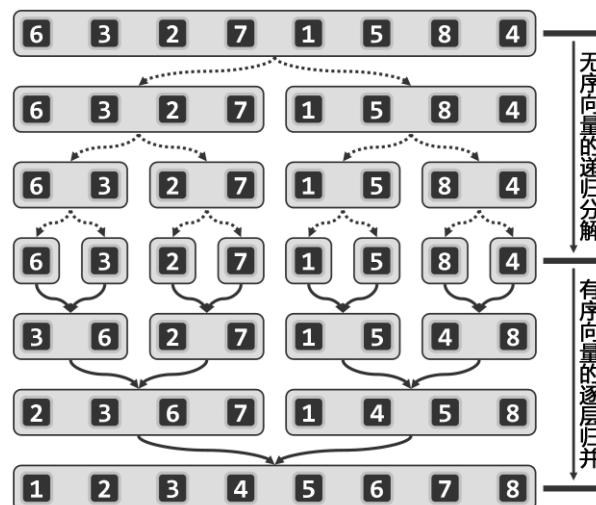


图2.17 归并排序实例

可以看出，上半部分对应于递归的不断深入过程：不断地均匀划分（子）向量，直到其规模缩减至 1 从而抵达递归基。此后如图中下半部分所示，开始递归返回。通过反复调用二路归并算法，相邻且等长的子向量不断地捉对合并为规模更大的有序向量，直至最终得到整个有序向量。由此可见，归并排序可否实现、可否高效实现，关键在于二路归并算法。

■ 二路归并接口的实现

代码 2.29 针对有序向量结构，给出了二路归并算法的一种实现。

```
1 template <typename T> //有序向量的归并
2 void Vector<T>::merge(Rank lo, Rank mi, Rank hi) { //以mi为界、各自有序的子向量[lo, mi)和[mi, hi)
3     T* A = _elem + lo; //合并后的向量A[0, hi-lo) = _elem[lo, hi)
4     int lb = mi - lo; T* B = new T[lb]; //前子向量B[0, lb) = A[lo, mi)
5     for (Rank i = 0; i < lb; B[i] = A[i++]); //复制前子向量
6     int lc = hi - mi; T* C = _elem + mi; //后子向量C[0, lc) = A[mi, hi)
7     Rank i, j, k; i = j = k = 0;
8     while ((j < lb) && (k < lc)) { //反复比较B和C的首元素
9         while ((j < lb) && B[j] <= C[k]) A[i++] = B[j++]; //将小者续至A末尾
10        while ((k < lc) && C[k] <= B[j]) A[i++] = C[k++]; //将小者续至A末尾
11    } //出口时，B或C为空，即j = lb或k = lc
12    while (j < lb) A[i++] = B[j++]; //B非空时须将剩余部分续至A末尾
13    delete [] B; //释放临时空间B
14 } //归并后得到完整的有序向量[lo, hi)
```

代码2.29 有序向量的二路归并

这里约定，参与归并的子向量在原向量中总是前、后相邻的，故借助三个入口参数即可界定其范围[*lo*, *mi*)和[*mi*, *hi*)。

另外，这里使用临时数组 *B*[]存放前一向量 *S*[*lo*, *mi*)的副本，并在归并完成之后将这部分空间归还给系统。故除输入向量本身，该算法还需 $\mathcal{O}(mi - lo) = \mathcal{O}(n)$ 辅助空间。

■ 归并时间

代码 2.29 中二路归并算法 *merge()* 的时间成本主要消耗于其中的三轮迭代。第一轮迭代旨在复制待归并的前一子向量，易见仅需 $\mathcal{O}(mi - lo) = \mathcal{O}(n)$ 时间。

第二轮迭代及其中嵌套的两个迭代旨在不断确定并取出当前首元素中的更小者，直到某一子向量为空。第三轮迭代则负责在后一子向量变空时，将前一子向量中剩余的元素直接转入输出向量。尽管后两轮迭代的形式略微复杂，但就其功能而言，每一次迭代都无非是在经 1~4 次比较后，将某一元素移至正确的位置。因涉及的元素总共不过 $n = hi - lo$ 个，故对应的时间复杂度不过 $\mathcal{O}(n)$ 。

请注意，二路归并虽可在少于 $\Omega(n \log n)$ 的时间内完成排序，但这与 2.7.3 节的结论并不矛盾——毕竟，这里的输入并非一组完全随机的元素，而是分为各自有序的两组，故就总体而言已具有一定程度的有序性。

另外，二路归并只需线性时间的结论，并不限于相邻且等长的子向量。实际上，即便分界点 *mi* 不是位于 *lo* 和 *hi* 的中央，以致待合并子向量的长度相差悬殊，*merge()* 算法依然可行且仅需 $\mathcal{O}(hi - lo)$ 时间。甚至，即便对于起始地址毫无关联的子向量，*merge()* 算法略作修改后亦可适用。

更重要地，正如我们将在后面（第3章之3.5.4节）看到的，这一算法框架也可以同样的效率应用于另一类典型的序列结构——列表。

最后，根据二路归并算法的流程，无论子向量的内部组成如何，也无论它们之间的相对大小如何，该算法都需要经过以上迭代处理，因此其在最好情况下的运行时间也是 $\Omega(n)$ 。概括起来，二路归并的时间复杂度应为 $\Theta(n)$ 。

■ 排序时间

那么，基于以上二路归并的线性算法，归并排序算法的时间复杂度又是多少呢？

不妨采用递推方程分析法，为此首先将归并排序算法处理长度为 n 的向量所需的时间记作 $T(n)$ 。根据算法构思与流程，为对长度为 n 的向量归并排序，需递归地对长度各为 $n/2$ 的两个子向量做归并排序，再花费线性时间做一次二路归并。如此，可得以下递推关系：

$$T(n) = 2 \times T(n/2) + O(n)$$

另外，当子向量长度缩短到1时，递归即可终止并直接返回该向量。故有边界条件

$$T(1) = O(1)$$

联立以上递推式，可以解得（习题[19]）：

$$T(n) = O(n \log n)$$

也就是说，归并排序算法可在 $O(n \log n)$ 时间内对长度为 n 的向量完成排序。因二路归并算法的效率稳定在 $\Theta(n)$ ，故更准确地讲，归并排序算法的时间复杂度应为 $\Theta(n \log n)$ 。

实际上，图2.17所绘出的整个算法执行过程的递归跟踪图，也已给出了相同的答案。为此只需如该图所示，按照各递归实例的规模将其分层排列。既然每次二路归并均只需线性时间，故同层的所有二路归并也只需线性时间。当然，这两个线性时间并不相同：前者是指线性正比于一对待归并子向量长度之和，后者则是线性正比于所有参与归并的子向量长度之和。由图不难看出，原向量中的每个元素在同一层次出现且仅出现一次，故同层递归实例所消耗时间之和应为 $\Theta(n)$ 。另外，各层递归实例的规模以2为倍数按几何级数变化，故共有 $\Theta(\log_2 n)$ 层，共计 $\Theta(n \log n)$ 时间。

习题

[1] 考查“算法A的分摊时间复杂度为 $O(n)$ ”与“算法A的平均时间复杂度为 $O(n)$ ”两个论断。

试说明，前者对算法效率高低的评判通常更加可信。

[2] 假设将代码2.4中expand()算法的扩容策略改为“每次追加固定数目的单元”。

a) 试证明，在最坏情况下，单次扩容操作的分摊时间复杂度为 $\Omega(n)$ ，其中 n 为向量规模；

b) 试举例说明，这种最坏情况的确可能发生。

[3] 试证明，代码2.5中shrink()算法的分摊复杂度为常数。

[4] 考查代码2.7中的permute()算法。

a) 试证明，从理论上讲，该算法的确可以将任一向量区间 $V[0, n)$ 按等概率地置乱；

（提示：亦即，各元素最终落在任一位置的概率均为 $1/n$ 。）

b) 试证明，从理论上讲，通过反复调用permute()算法，可以按等概率生成向量V的 $n!$ 种排列；

c) 试证明，a)和b)的结论在目前的实际计算环境中并不能兑现；

（提示：比如，绝大多数的排列根本就不可能被该算法生成。）

d) 针对c)所指出的不足，可以如何改进该算法？

[5] 考查代码 2.10 中 `find()` 算法。

- a) 在最好情况下，该算法需要运行多长时间？为什么？
- b) 若仅考查成功的查找，则平均需要运行多少时间？为什么。

[6] 考查代码 2.11 中无序向量插入算法 `insert(r, e)`。

试证明，若插入位置 r 等概率分布，则该算法的平均时间复杂度为 $\mathcal{O}(n)$ ， n 为向量的规模。

[7] 考查如代码 2.12 所示的 `remove(lo, hi)` 算法。

- a) 若以自后向前的次序逐个前移后继元素，可能出现什么问题？
- b) 何时出现这类问题？试举一例（提示：后继元素多于待删除元素时部分单元相互覆盖）。

[8] `Vector::deduplicate()` 算法的如下实现是否正确？为什么？

```
template <typename T> int Vector<T>::deduplicate() {  
    int oldSize = _size; //记录原规模  
    for (Rank i = 1; i < _size; i++) //逐一检查V[i]，若其某一前驱V[j]与之雷同，则删除V[j]  
    { Rank j = find(_elem[i], 0, i); if (0 <= j) remove(j); }  
    return oldSize - _size; //向量规模变化量，即被删除元素总数  
}
```

[9] 考查如代码 2.14 所示的 `Vector::deduplicate()` 算法。

- a) 试证明，即便在最好情况下，该算法也需要运行 $\Omega(n^2)$ 时间；
- b) 试改进无序向量的 `deduplicate()` 算法，使其时间复杂度降至 $\mathcal{O}(n \log n)$ ；
- c) 这一效率是否还有改进的余地？为什么？

[10] 试参照代码 2.16 中 `increase()` 方法的实现方式，基于无序向量的遍历接口 `traverse()` 实现以下操作（假定向量元素类型支持算术运算）：

- a) `decrease()`：所有元素数值减一；
- b) `doubleUp()`：所有元素数值加倍。

[11] 字符串、复数、矢量等类型没有提供自然的比较规则，但仍能人为地对其强制定义某种大小关系（即次序关系）。试分别为这三种类型的对象定义“人工的”次序。

[12] 试针对以下情况验证，如代码 2.24 实现的 `search(e, lo, hi)` 版本 C 返回的秩是否符合接口规范：

- a) $[lo, hi]$ 中的元素均小于 e ；
- b) $[lo, hi]$ 中的元素均等于 e ；
- c) $[lo, hi]$ 中的元素均大于 e ；
- d) $[lo, hi]$ 中既包含小于 e 的元素，也包含大于 e 的元素，但不含等于 e 的元素。

[13] 考虑用向量存放一组字符串。

为在其中进行二分查找，可依据字典序（lexicographical order）确定字符串之间的次序：

设字符串 $a = a_1a_2\dots a_n$ 和 $b = b_1b_2\dots b_m$ ，则 $a < b$ 当且仅当 $n = 0 < m$ ，或者 $a_1 < b_1$ ，或者 $a_1 = b_1$ 且 $a_2\dots a_n < b_2\dots b_m$ 。

- a) 试实现一个字符串类，并提供相应的比较器，或者重载对应的操作符；
- b) 若共有 n 个字符串，二分查找的复杂度是否仍为 $\mathcal{O}(\log n)$ ？

[14] 设采用二分查找 `binSearch()` 算法如代码 2.21 实现的版本 A，针对独立均匀分布于 $[0, 2n]$ 内的整数目标关键码，在固定的有序向量 $\{1, 3, 5, \dots, 2n-1\}$ 中查找。

- a) 若将平均的成功和失败查找长度分别记作 S 和 F ，试证明： $(S+1) \cdot n = F \cdot (n+1)$ ；

- b) 上述结论，是否适用于 `binSearch()` 算法的其它版本？为什么？
- c) 上述结论，是否适用于 `fibSearch()` 算法的各个版本？为什么？
- d) 若目标关键码按照其它的随机规律分布，以上结论又应如何调整？

[15] 试按如下接口说明及性能要求设计并实现一个 `Fib` 类，以高效地支持 Fibonacci 查找：

```
class Fib { //Fibonacci数列类
public:
    Fib(int n); //初始化为不小于n的最小Fibonacci项(如, Fib(6) = 8)。O(logφ(n))时间
    int get(); //获取当前Fibonacci项(如, 若当前为8, 则返回8)。O(1)时间
    int next(); //转至下一Fibonacci项(如, 若当前为8, 则转至13)。O(1)时间
    int prev(); //转至上一Fibonacci项(如, 若当前为8, 则转至5)。O(1)时间
};
```

[16] 为做 Fibonacci 查找，未必非要严格地将向量整理为 $\text{fib}(n)-1$ 形式的长度。比如可考虑以下策略：

- a) 按照黄金分割比，取 $mi = \lfloor 0.382*lo + 0.618*hi \rfloor$ ；
- b) 按照近似的黄金分割比，取 $mi = \lfloor (lo + 2*hi) / 3 \rfloor$ ；
- c) 按照近似的黄金分割比，取 $mi = (lo + lo<<1 + hi + hi<<2) >> 3$ 。

这几种替代策略，综合性能孰优孰劣？为什么？

[17] 设 $A[0, n)$ 为一个非降的正整数向量。试设计并实现算法 `expSearch(int x)`，对于任意给定的正整数 $x \leq A[n-1]$ ，从该向量中找出一个元素 $A[k]$ ，使得 $A[k] \leq x \leq A[\min(n-1, k^2)]$ 。

如果有多个满足这一条件的 k ，只需返回其中任何一个，但查找时间不得超过 $O(\log(\log k))$ 。

[18] 设 $A[0, n][0, n)$ 为整数矩阵（即二维向量）， $A[0][0] = 0$ 且任何一行（列）都严格递增。

- a) 试设计一个算法，对于任一整数 $x \geq 0$ ，在 $O(r + s + \log x)$ 时间内，从该矩阵中找出并报告所有值为 x 的元素（的位置），其中 $A[r][0]$ ($A[0][s]$) 为第 r 行（列）中不大于 x 的最大者；
- b) 若 A 的各行（列）只是非减（而不是严格递增），你的算法需做何调整？复杂度有何变化？

[19] 试根据 2.8.3 节所给出的递推关系以及边界条件证明，如代码 2.28 所示 `mergeSort()` 算法的运行时间 $T(n) = O(n \log n)$ 。

[20] 自学 C++ STL 中 `vector` 容器的使用方法，阅读对应的源代码。

[21] 自学 Java 中 `java.util.ArrayList` 和 `java.util.Vector` 类的使用方法，并阅读对应的源代码。

[22] 位图（Bitmap）是一种特殊的序列结构，其长度无限，且其中每个元素的取值均为布尔型（初始均为 `false`），支持如下操作接口：

```
void set(int i); //将第 i 位置为 true
void clear(int i); //将第 i 位置为 false
bool test (int i); //测试第 i 位是否为 true
```

- a) 试给出 Bitmap 类的具体实现；
- b) 并分析其时间和空间复杂度；
- c) 创建 Bitmap 对象时，如何节省下为初始化所有元素所需的时间？
(提示：以一定的空间为代价)

[23] 利用 Bitmap 类设计算法，在线性时间内剔除 ASCII 字符串中的重复字符，各字符仅保留一份。

[24] 利用 Bitmap 类设计算法，快速地计算不大于 10^8 的所有素数。

(提示：Eratosthenes 筛法)

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

- [25] 试分别针对二分查找算法版本 A (代码 2.21) 及 Fibonacci 算法 (代码 2.22), 推导其失败查找长度的显式公式, 并就此方面的性能对二者做一对比。
- [26] 代数判定树 (algebraic decision tree, ADT) 是比较树的推广, 其中每个节点对应于根据某一代数表达式做出的判断。比如, 比较树节点的 “ $a == b$ ” 式判断或 “ $a < b$ ” 式比较, 都可以统一为根据一阶代数表达式 “ $a - b$ ” 取值符号的判断, 故比较树可认为是代数判定树中最简单的特例。
a) 对应于 2.7.4 节所列比较树的性质, 代数判定树有哪些相仿的性质?
b) 2.7.5 节中基于比较树模型的下界估计方法, 可否推广至代数判定树? 如何推广?
- [27] 任给 12 个互异的整数, 其中 10 个已组织为一个有序序列, 现需要插入剩余的两个以完成整体排序。若采用 CBA 式算法, 最坏情况下至少需做几次比较? 为什么?
- [28] 2.8.3 节的向量归并排序算法是稳定的吗? 若是, 请给出证明; 否则, 试举一实例。

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第3章

列表

上一章介绍的向量结构中，各数据项的物理存放位置与逻辑次序完全吻合，故可通过秩直接访问对应的元素，此即所谓“循秩访问”（call-by-rank）。这种访问方式，如同根据具体的城市名、街道名和门牌号找到某人。本章将要介绍的列表，与向量同属序列结构的范畴，其中的元素也构成一个线性逻辑次序；但与向量极为不同的是，元素的物理地址可以任意。相应地，为保证对列表元素访问的可行性，逻辑上相邻的元素之间必须维护某种索引关系，并藉此访问各元素（逻辑上）的前驱和后继。列表元素间的索引，可理解为被索引元素的位置（position），故列表元素是“循位置访问”（call-by-position）的；也可理解为通往被索引元素的链接（link），故亦称作“循链接访问”（call-by-link）。这种访问方式，如同根据已知的某人找到他/她的某位亲朋、他/她的亲朋的亲朋、...。注意，位置是就逻辑次序而言的，并非指物理（地理）上的邻居——后者是向量中秩的概念。

本章的讲解将围绕列表结构的高效实现逐步展开，包括其 ADT 接口规范及对应的算法。此外，还将针对有序列表，系统地介绍经典的查找与排序算法，并就其性能做一分析对比。

§ 3.1 从向量到列表

不同数据结构内部的存储与组织方式各具特色，表现在其支持的操作接口上，相应的操作方式及性能也不尽相同。在设计或选用数据结构时，应更多地从实际应用的需求出发，先确定对应的功能接口及性能指标，再落实到具体的存储方式。比如，提出并实现列表结构的目的，就是弥补向量结构在解决某些应用问题时的不足。二者的差异，表面上体现为来自外部的操作方式不同，但根源则在于其内部存储方式的不同。

3.1.1 从静态存储到动态存储

一般地，对数据结构的操作可分为静态和动态两类，前者仅从中获取存储的信息，后者则会修改数据结的构局部甚至整体。以第 2 章基于数组的向量结构为例，其 `size()` 和 `get()` 等静态操作均可在常数时间内完成，而 `insert()` 和 `remove()` 等动态操作在最坏情况下却都需线性时间。究其原因，在于数组结构“各元素物理地址连续”的约定。得益于此，可在 $O(1)$ 时间内由待访问元素的秩确定其物理地址。但反过来，正因为必须保持逻辑上相邻元素的物理地址连续，在添加（删除）元素之前（之后）才不得不移动 $O(n)$ 个后继元素。

与操作方式相对应地，向量内部按“静态存储”策略来存储和组织，即强制各元素的物理存储次序完全对应于其逻辑关系。尽管如此可使静态操作的效率达到极致，但就动态操作而言，每一局部的修改都可能引起大范围甚至整个数据结构的调整。内部存储方式决定外部访问的方式与效率，故不足为怪，采用静态存储策略的向量注定难以高效率地支持动态操作。

列表（list）结构采用的则是所谓“动态存储”策略。具体地，在其生命期内，此类数据结构将根据内部数据更新的需要，相应地分配或回收局部的数据空间。如此，各元素之间的逻辑关系得以延续，但却不再与其物理存储次序相关。作为补偿，此类结构将通过指针或引用等机制来确定各元素的实际物理地址。例如，链表（linked list）就是一种典型的

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第4章

栈与队列

本章将定制并实现更加基本且更为常用的两类数据结构——栈与队列。与此前介绍的向量和列表一样，它们也属于线性序列结构，故其中存放的数据对象之间也具有某种线性次序。相对于一般的序列结构，栈与队列的数据操作范围仅限于逻辑上的特定某端。然而，得益于其简洁性与规范性，它们既成为构建更复杂、更高级数据结构的基础，同时也是算法设计的基本出发点，甚至常常作为标准配置的基本数据结构以硬件形式直接实现。因此无论就工程或理论而言，其基础性地位都是其它结构无法比拟的。

在信息处理领域的各个层面，栈与队列的身影都随处可见。许多程序语言本身就是建立于栈结构之上，无论 PostScript 或者 Java，其实时运行环境都是基于栈结构的虚拟机。再如，网络浏览器多会将用户最近访问过的地址组织为一个栈。这样，用户每访问一个新页面，其地址就会被存放至栈顶；而用户每按下一次“后退”按钮，即可沿相反的次序返回此前刚访问过的页面。类似地，主流的文本编辑器也大都支持编辑操作的历史记录功能，用户的编辑操作被依次记录在一个栈中。一旦出现误操作，用户只需按下“撤销”按钮，即可取消最近一次操作并回到此前的编辑状态。

在需要公平且经济地对各种自然或社会资源做管理或分配的场合，无论是调度银行和医院的服务窗口，还是管理轮耕的田地和轮伐的森林，队列都可大显身手。甚至计算机及其网络自身内部的各种计算资源，无论是多进程共享的 CPU 时间，还是多用户共享的打印机，也都需要借助队列结构实现合理和优化的分配。

相对于向量和列表，栈与队列的外部接口更为简化和紧凑，故亦可视作向量与列表的特例，因此 C++ 的继承与封装机制在此可以大显身手。得益于此，本章的重点将不再拘泥于对数据结构内部实现的展示，而是更多地从外部特性出发，结合实际问题介绍栈与队列在具体应用问题中的应用。

在栈的应用方面，本章将在 1.4 节的基础上，结合函数调用栈的机制介绍一般函数调用的实现方式与过程，并将其推广至递归调用。然后以降低空间复杂度的目标为线索，介绍通过显式地维护栈结构解决应用问题的典型方法和基本技巧。此外，还将着重介绍如何利用栈结构，实现基于试探回溯策略的高效搜索算法。在队列的应用方面，本章将介绍如何实现基于轮值策略的通用循环分配器，并以银行窗口服务为例实现基本的调度算法。

§ 4.1 栈

4.1.1 ADT 接口

■ 入栈与出栈

栈 (stack) 是存放数据对象的一种特殊容器，其中的数据对象按线性的逻辑次序排列，故也可定义首、末元素。尽管栈结构也支持对象的插入和删除操作，但操作的范围仅限于栈的某一特定端。也就是说，如果约定新的元素只能从某一端插入其中，反过来也只能从同一端删除已有的元素。

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第5章

二叉树

通过前面的章节我们已经了解了一些基本的数据结构，根据其实现方式，这些数据结构大致分为两种类型：基于数组的实现与基于链表的实现。正如我们已经看到的，就其效率而言，这两种实现方式各有长短。具体来说，基于数组实现的结构允许我们通过下标或秩，在常数的时间内找到目标对象，并读取或更新其内容。然而，一旦需要对这类结构进行修改，那么无论是插入还是删除，都需要耗费线性的时间。反过来，基于链表实现的结构允许我们借助引用或位置对象，在常数的时间内插入或删除元素；但是为了找出居于特定次序的元素，我们不得不花费线性的时间对整个结构进行遍历查找。能否将这两类结构的优点结合起来，并回避其不足呢？本章将要介绍的树结构，将正面回答这一问题。

实际上，从更广的视角来看，此前介绍的结构都属于线性结构（**linear structure**），亦即，在其中元素之间存在一个自然的线性次序。树结构则不然，其中的元素之间并不存在天然的直接后继或直接前驱关系，因此属于非线性结构（**non-linear structure**）。不过，正如我们马上就要看到的，只要附加上某种约束（比如遍历），也可以在树结构中的元素之间确定某种线性次序，因此也有人称之为半线性结构（**semi-linear structure**）。

无论如何，随着从线性结构转入树结构，我们的思维方式也将有个飞跃，相应地，算法设计的策略与模式也会因此有所变化，许多基本的算法也将得以更加高效地实现。以第7章和第8章将要介绍的平衡二叉搜索树为例，若其中包含n个元素，则每次查找、更新、插入或删除操作都可在 $\mathcal{O}(\log n)$ ^①时间内完成，而每次遍历都可在 $\mathcal{O}(n)$ 时间内完成。

树结构在算法理论与实际应用中始终都扮演着最为关键的角色，并且有着不计其数的变种。究其原因，在于其独特而又普适的逻辑结构。树是一种自组织的分层结构，而层次化这一特征几乎蕴含于所有事物及其之间的关系当中，成为它们的本质属性之一。从信息处理领域的文件系统、互联网域名系统和数据库系统，一直到地球生态系统以至于人类社会系统，层次化特征以及层次结构均无所不在。

有趣的是，作为树的特例，二叉树实际上并不失其一般性。本章将指出，无论就逻辑结构或算法功能而言，任何有序多叉树都可等价地转化并实现为二叉树。因此，本章讲解的重点也将放在二叉树上。我们将以通讯编码算法的实现这一应用实例作为线索贯穿全章。

§ 5.1 二叉树及其表示

5.1.1 树

■ 有根树

116

从图论的角度看，树等价于连通无环图。因此与一般的图相同，树也由一组顶点（**vertex**）以及联接与其间的若干条边（**edge**）组成。在计算机科学中，往往还会在此基

^① 由第1章习题[10]，就多项式意义而言 $\mathcal{O}(\log n)$ 与 $\mathcal{O}(1)$ 无限接近，而 $\mathcal{O}(\log n)$ 与 $\mathcal{O}(n)$ 相比则几乎改进了一个线性因子。因此从这一角度来看，树结构的确能够将数组和链表的优点结合起来

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第6章



4.4 节曾仿效古希腊英雄特修斯，以栈等基本数据结构模拟线绳和粉笔，展示了试探回溯策略的应用技巧。实际上，这一技巧可进一步推广至更为一般性的场合，包括可以图结构描述的应用问题，从而导出一系列对应的图算法。

特修斯取得成功的关键在于，借助线绳掌握迷宫内各通道之间的联接关系。事实上在很多应用问题中，能否有效描述和利用这类信息都是至关重要的。总体而言，这类信息往往可以理解为定义于一组对象之间的二元关系，比如城市交通图中联接于各公交站之间的街道，或者互联网中联接于 IP 节点之间的路由等。尽管在某种程度上，第 5 章所介绍的树结构也可用以表示这种二元关系，但树中的这类关系仅限于父、子节点之间。相互之间均可能存在二元关系的一组对象，从数据结构的角度看已不再对应于任何线性次序，故属于非线性结构。这类更具一般性的二元关系，属于图论（Graph Theory）的研究范畴。当然，正如以下将看到的，从算法的角度对此类结构的处理策略与上一章相仿，也是通过遍历（traversal）将其转化为非线性结构，进而借助我们对树结构的处理方法和技巧，最终解决问题。

以下首先简要介绍图论的概念，已有相关基础的读者可直接跳过。接下来，介绍如何实现作为抽象数据类型的图结构，主要讨论邻接矩阵和邻接表两种实现方式。然后，从遍历的角度介绍将图转化为树的典型方法，包括广度优先搜索和深度优先搜索。进而，分别以拓扑排序和双连通域分解为例，介绍利用基本数据结构并基于遍历模式设计图算法的总体方法。最后，依照数据结构决定遍历访问次序的观点，将所有遍历算法概括并统一为最佳优先遍历这一模式。如此，我们不仅能够更加准确和深刻地理解不同图算法之间的共性与联系，更可以学会通过选择和改进数据结构高效地实现各种图算法——这也是本章的重点与精髓。

§ 6.1 概述

■ 图

图结构是描述和解决实际应用问题的一种基本而有力的工具，交通图、航线图、电路图、冲突图以及可见性图等都属于这方面的实例。所谓的图（graph）可表示为 $G = (V, E)$ 。其中，集合 V 中的对象称作顶点（vertex）；集合 E 中的元素分别对应于 V 中的某对顶点，以表明它们之间存在某种关系，故这类定义于顶点之间的关系亦形象地称作边（edge）。可见，此处所讨论的“图”，与日常所指的图形、报表或者数学上的函数图像等有本质区别。当然，从计算的需求出发，我们约定 V 和 E 均为有限集，通常记 $n = |V|$ 和 $e = |E|$ 。

与图相关的术语在各种文献中不尽相同，比如有的将顶点称作节点（node），或将边称作弧（arc），本章将统一称作顶点和边。为直观地显示图结构，一种行之有效的方法是用小圆圈或小方块代表顶点，用联接于其间的直线段或曲线表示对应的边。

■ 无向图、有向图及混合图

若边 (u, v) 所对应顶点 u 和 v 的次序无所谓，则称作无向边（undirected edge），描述城际高速公路或同学关系等信息的边即属此类。反之若 u 和 v 不对等，则称 (u, v) 为有向边（directed edge），描述企业之间资金流向、程序之间相互调用关系或类之间继承

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第7章

搜索树

从本章开始，讨论的重点将集中于查找技术。实际上，此前章节中已经就此做过一些讨论，在向量与列表等结构中甚至已经提供并实现了对应的接口。遗憾的是，此前这类接口的效率无法令人满意。

以代码 2.1 中向量模板类 `Vector` 为例，其中针对无序和有序向量，分别提供了 `find()` 和 `search()` 接口。前者的实现策略只不过是将目标对象与向量内存放的对象逐个比对，最坏情况下需要运行 $\mathcal{O}(n)$ 时间。后者利用二分查找策略尽管可以确保在 $\mathcal{O}(\log n)$ 时间内完成单次查找，但一旦向量本身需要修改，无论是插入还是删除，在最坏情况下每次也需 $\mathcal{O}(n)$ 时间。列表模板类 `List` 的情况类似。总之，若要求在自身组成可高效率动态变化的对象集合中进行高效率的查找，则此前所介绍的数据结构均难以胜任。

那么，究竟有无可能同时支持高效率的动态修改和高效率的静态查找？如有可能，应该采用什么样的数据结构？这些正是以下两章将要回答的问题。

因为这部分内容所涉及的数据结构变种较多、各具特色且各有所长，故按基本和高级两章分别讲解，相关内容之间的联系如图 7.1 所示。

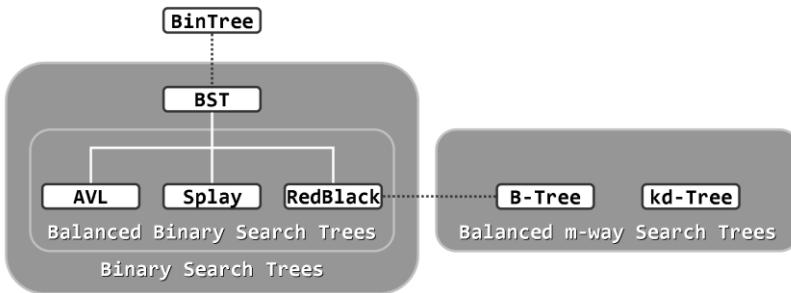


图7.1 第7章、第8章内容纵览

本章将首先介绍树式查找的基本算法及数据结构，通过对二分查找策略的推广，定义并实现二叉搜索树结构。尽管就最坏情况下的渐进时间复杂度而言，这一方法与此前并无本质的提高，但这部分内容依然十分重要，因为这一基于树形结构的构思正是这两章后续内容的基础和出发点。比如，本章的后半部分将基于这一构思提出平衡二叉搜索树的概念，引入并实现 `AVL` 树这一典型的平衡二叉搜索树，借助精巧的平衡调整算法，保证单次动态修改与静态查找均可在 $\mathcal{O}(\log n)$ 时间内完成。

以上作为树形查找算法的基础，已可较为圆满地从正面回答前述问题。

7.1.1 循关键码访问

所谓查找或搜索（`search`），指按照事先约定的规则，在一组数据对象中找到符合特定条件者，这是构建算法的一种基本而重要的操作。其中的数据对象统一地表示和实现为词

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第8章

高级搜索树

包括 AVL 树在内，平衡二叉搜索树家族拥有众多成员，它们各有其适用的场合和范围。本章将按照图 7.1 的总体框架，延续上一章介绍更多的变种。比如，针对实际应用中普遍存在的数据访问的局部性特点，我们将按照“最常用者优先”的原则，引入并实现伸展树结构。尽管最坏情况下其单次操作需要 $\mathcal{O}(n)$ 时间，但就分摊意义而言其复杂度仍在 $\mathcal{O}(\log n)$ 以内。构思之深邃、实现之简洁加上适用之广泛，集三者于一身的伸展树具有别样的魅力。

通过对平衡二叉搜索树的推广，本章将引入平衡多路搜索树，并着重讨论为其中典型代表的 B-树。我们也将体会到此类结构不可替代的作用——借助此类结构，因何与如何能够有效地弥合不同存储级别之间在访问速度上的巨大差异。

以 4 阶 B-树为参照，本章将引入并实现红黑树。红黑树不仅能保持整树平衡从而有效控制单次操作的成本，而且可以将旋转之类结构性调整操作的次数控制在常数以内，这也是该树有别于其它变种的关键特性。这一特性不仅保证了更高的计算效率，更为持久性结构（*persistent structure*）之类高级数据结构的实现，提供了直接而有效的方法。

最后，将结合平面范围查询，介绍基于平面子区域正交划分的 kd-树结构。该结构是对四叉树（quadtree）和八叉树（octree）等结构的一般性推广，为计算几何方面应用问题的求解提供了基本的模式和有效的方法。

§ 8.1 伸展树

前一章介绍的 AVL 树是平衡二叉搜索树的一种完美实现，但实际上平衡二叉搜索树的实现方式远不止于此，比如本节将要介绍的伸展树（splay tree）^①。相对于 AVL 树，伸展树的实现更为简捷。首先，无需对节点实施显式的平衡化操作，转而通过一系列直观而简捷操作的重复，将最近被访问的节点推至树根。另外，伸展树不需要对节点本身的结构做任何附加的要求或改动，不再需要记录平衡因子或高度之类的额外信息，故适用范围更广。

8.1.1 局部性

信息处理的一种典型模式是，将所有数据项整体地视作一个集合，并将其组织为某种适宜的数据结构，进而借助接口反复地高效访问其中的元素。本书介绍的搜索树、词典和优先级队列等都属于典型的此类实例。仅从算法角度来看，无论读取或修改，各次访问操作之间相互独立，次序随机且等概率，故对数据结构效率的分析往往局限于单次操作的复杂度。

然而这一假定往往并不足以反映实际情况。比如，对于实际的数据结构，不仅各操作之间具有极强的相关性，访问次序的规律性很强，而且不同操作发生的几率也极不均衡。同一数据结构的一组连续访问之间所具有的这类相关性，称作“数据局部性”（*data locality*）。

^① 1985 年由 D. D. Sleator 和 R. E. Tarjan 发明：

D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Trees. JACM, 32:652-686, 1985.

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第9章

词典

借助数据结构来表示和组织待处理信息，可以数据集合为单位将其视作一个整体，进而提高信息访问接口的规范性以及信息处理的效率。如今，借助关键码直接查找和访问数据元素的形式已经为越来越多的数据结构所采用，这也成为现代数据结构的一个重要特征。

词典（**dictionary**）结构即是其中最典型的例子，逻辑上它是一组数据元素的集合，其中数据元素都是由关键码和数据项合成的词条（**entry**）。映射（**map**）结构与词典结构一样也是词条的集合，二者的差别仅仅在于，映射要求不同词条的关键码互异，而词典则允许多个词条拥有相同的关键码^①。除了静态查找，映射和词典都支持动态更新，二者统称作符号表（**symbol table**）。实际上，“是否允许雷同关键码”应从语义层面而非 **ADT** 接口的层面予以界定，故本章将不再过份强调二者的差异，而是笼统地称之为词典，并以跳转表和散列表为例，按照“允许雷同”和“禁止雷同”的语义分别实现其统一的接口。

尽管此处词典和映射中的数据元素仍表示和实现为词条形式，但这一做法并非必须的。与第 7 章和第 8 章的搜索树相比，符号表结构并不要求词条之间可以根据关键码比较大小；与第 10 章的优先级队列相比，其查找对象亦不仅限于最大或最小词条。符号表并不按照大小次序来组织数据项，无论各数据项之间是否可以定义某种大小关系。实际上，以散列表为典型代表的符号表结构，将转而根据数据项的数值直接做逻辑查找和物理定位，此即所谓的循值访问（**call-by-value**）。相对于此前各种方式，这一方式更为自然，适用性更强。

既然已经抛开了大小的概念，相关计算过程自然不再属于基于比较式的类型。因此我们将不再束缚于此前关于 **CBA** 式算法局限的结论，从此踏上通往高效算法的另一条大道。比如在第 9.4 节我们将看到，散列式排序算法将不再服从第 2.7 节给出的复杂度下界。

当然，为支持循值访问的方式，在符号表的内部，必须强制地在数据对象的数值与其物理地址之间建立某种关联。所谓散列，讨论和研究的即是在兼顾空间与时间效率的前提下，设计并实现这种关联的一般性原则、技巧与方法，这也是本章的核心与重点。

§ 9.1 词典

9.1.1 操作接口

除通用的接口之外，词典结构主要的操作接口可归纳为表 9.1。

表9.1 词典ADT支持的标准操作接口

操作接口	功能描述
get(key)	若词典中存在以 key 为关键码的词条，则返回该词条的数据对象；否则，返回 null
put(key,value)	插入词条(key, value)，并报告是否成功
remove(key)	若词典中存在以 key 为关键码的词条，则删除之并返回 true；否则，返回 false

^① 事实上，也有一些文献中定义的词典和映射与此约定恰好相反

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第10章

优先级队列

此前的搜索树结构和词典结构，都支持对数据全集的查找。也就是说，其中包含的每一个数据对象都可能作为查找目标并被成功返回。为此，搜索树结构需要在所有元素之间定义并维护一个显式的全序（**full order**）关系；词典结构中的数据对象之间尽管不必支持比较大小，但在散列表之类的具体实现中，都在内部强制地在对象的数值与其对应的秩之间建立起某种关联（尽管实际上这种关联越“随机”越好），从而隐式地定义了一个全序次序。

就对外接口功能而言，本章将要介绍的优先级队列较之此前的数据结构反而有所削弱。具体地，这类结构将操作对象限定于当前的全局极值者。这种根据数据对象之间相对优先级对其进行访问的方式，与此前的也有本质区别，称作循优先级访问（**call-by-priority**）。比如，在全体北京市民中查找年龄最长者，或者在所有鸟类中查找种群规模最小者，等等。

当然，“全局极值”本身就隐含了“所有元素可相互比较”这一性质。然而，优先级队列并不会也不必忠实地动态维护这个全序，却转而维护一个偏序（**partial order**）关系。其高明之处在于，如此不仅足以高效地支持仅针对极值对象的接口操作，更可有效地控制整体计算成本。正如我们将要看到的，对于常规的查找、插入或删除等操作，优先级队列的效率并不低于此前的结构；而对于数据集的批量构建及相互合并等操作，其性能却更胜一筹。作为不失高效率的轻量级数据结构，优先级队列在许多领域都是扮演着不可替代的角色。

§ 10.1 优先级队列

10.1.1 优先级与优先级队列

除了作为存放数据的容器，数据结构还应能够按某种约定的次序动态地组织数据，以支持高效的查找和修改操作。比如 4.5 节的队列结构，可用以描述和处理日常生活中的很多问题：在银行排队等候接受服务的客户，提交给网络打印机的打印任务，等等。在这类问题中，无论客户还是打印任务，接受服务或处理的次序完全取决于其出现的时刻——先到的客户优先接受服务，先提交的打印任务优先执行——此即所谓“先进先出”原则。

然而在更多实际应用环境中，这一简单公平的原则并不能保证整体效率必然达到最高。试想，若干病人正在某所医院的门诊处排队等候接受治疗时，忽然送来一位骨折的病人。要是固守“先进先出”的原则，那么他只能咬牙坚持到目前已经到达的每位病人都已接受过治疗之后。显然，那样的话该病人将承受更长时间的痛苦，甚至贻误治疗的最佳时机。因此，医院在此时都会灵活变通，优先治疗这位骨折的病人。同理，若此时又送来一位心脏病突发的患者，那么医生肯定也会暂时把骨折病人放在一边（如果暂时没有更多医生的话），转而优先抢救心脏病人。

由此可见，在决定病人接受治疗次序时，除了他们到达医院的先后次序，更应考虑到病情的急缓程度，优先接受治疗的应是病情最危重的病人。在数据结构与算法设计中，类似的例子也屡见不鲜。在 3.5.3 节的选择排序算法中，每次迭代都要调用 `selectMax()` 从未排序区间选出最大者。在 5.4.3 节的 Huffman 编码算法中，每次迭代都要调用 `minHChar()`

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第11章

串

串或字符串（**string**）属于线性结构，自然地也可直接利用第2章的向量或第3章的列表等序列结构加以实现，故就串结构本身而言无需更多讨论。但字符串这类数据结构的特点也极其鲜明，可以归纳为结构简单、规模庞大、元素（字符）重复率高。所谓结构简单，是指字符表本身的规模不大，甚至可能极小。以生物信息序列为例，参与蛋白质（文本）合成的常见氨基酸（字符）只有20种，而构成DNA序列（文本）的碱基（字符）仅有4种。就其规模而言，微软Windows系统逾4000万行的源代码长度累计达到40GB，地球系统模式的单个输出文件长达1~100GB，而从本质上讲，这些数据都不过是由ASCII字符组成的。因此，以字符串形式表示的海量文本数据的高效处理技术一直都是相关领域的研究重点。鉴于字符串结构的上述特点，本章将直接利用C++本身提供的字符数组，转而将讲述的重点集中于各种串匹配算法**indexOf()**的基本原理与高效实现。

§ 11.1 串及串匹配

11.1.1 串

■ 字符串

一般地，由n个字符构成的串记作 $S = "a_0 \ a_1 \ \dots \ a_{n-1}"$ ，其中 $a_i \in \Sigma, 0 \leq i < n$ 。这里的 Σ 是所有可用字符的集合，称作字符表（**alphabet**）。常见的字符表有ASCII字符集或Unicode字符集；就二进制表示形式而言，字符表 $\Sigma = \{0, 1\}$ ；而在生物信息学中， Σ 则可能是构成DNA序列的所有碱基，或者组成蛋白质的所有氨基酸。

字符串S所含字符的总数n称作S的长度，记作 $|S| = n$ 。这里只考虑长度有限的串，即 $n < \infty$ 。特别地，长度为零的串称作空串（**null string**）。请注意，空串不同于由空格' '组成的串。

■ 子串

在处理字符串S时，经常需要取出其中某一连续的片段，称作S的子串（**substring**）。具体地，由串S中起始于位置i的连续k个字符组成的子串，记作

substr(S, i, k) = "a_i \ a_{i+1} \ \dots \ a_{i+k-1}", $0 \leq i < n, 0 \leq k$

k即子串的长度。有两种特殊子串：起始于位置0、长度为k的子串称为前缀（**prefix**），终止于位置n-1、长度为k的子串称为后缀（**suffix**），分别记作

prefix(S, k) = substr(S, 0, k);

suffix(S, k) = substr(S, n-k, k)。

由上述定义可直接导出以下结论：空串是任何字符串的子串，也是任何字符串的前缀和后缀；任何字符串都是自己的子串，也是自己的前缀和后缀。此类子串、前缀和后缀分别称作平凡子串（**trivial substring**）、平凡前缀（**trivial prefix**）和平凡后缀（**trivial suffix**）。反之，字符串本身之外的所有非空子串、前缀和后缀，分别称作真子串（**proper substring**）、真前缀（**proper prefix**）和真后缀（**proper suffix**）。

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

第12章

排序

此前诸章中，我们已循序渐进地介绍过多种基本的排序算法：2.8节和3.5节分别针对向量和列表，以排序器的形式实现过起泡排序、归并排序、插入排序以及选择排序等算法；9.4.1节也曾仿效散列的思路与手法实现过桶排序算法，9.4.3节还将其推广至基数排序算法；10.2.5节也曾完美地利用完全堆结构的特长，实现过就地堆排序算法。

本章着重于高级排序算法。与以上基本算法一样，其构思与技巧各具特色，在不同应用中的效率也各有千秋。因此在学习过程中，唯有更多地关注不同算法之间细微而精妙的差异，留意体会其优势与不足，方能做到运用自如，并针对实际问题的需要合理地取舍与改进。

§ 12.1 快速排序

12.1.1 分治策略

与归并排序算法一样，快速排序（quicksort）算法^①也是分治策略的典型应用，但二者之间也有很大的区别。2.8.3节曾指出，归并排序的计算量主要消耗于有序子向量的归并操作，而子向量的划分却几乎不费时间。快速排序恰好相反，它可以在 $\mathcal{O}(1)$ 时间内由子问题的解直接得到原问题的解，但为了将原问题划分为两个子问题却需要 $\mathcal{O}(n)$ 时间。

需特别指出的是，在进行子任务划分时，虽然快速排序算法能够确保子任务是相互独立的，并且规模总和保持在线性量级，但该算法却不能保证由同一任务划分出来的子任务的规模大体相当，实际上甚至有可能极不平衡。因此，快速排序算法并不能保证最坏情况下的 $\mathcal{O}(n \log n)$ 时间复杂度。尽管如此，它仍然受到人们的青睐，并在实际应用中往往成为首选的排序算法。究其原因在于，快速排序算法易于实现，代码结构紧凑简练，而且对于按通常规律随机分布的输入序列，快速排序算法的平均运行时间较之同类算法更少。

下面结合向量介绍该算法的原理，并针对实际需求相应地给出不同的实现版本。

12.1.2 轴点

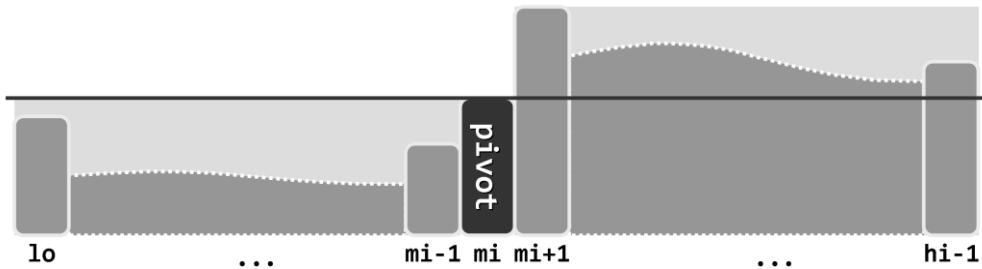


图12.1 序列的轴点（这里用高度来表示各元素的大小）

^① 由英国计算机科学家、1980年图灵奖得主C. A. R. Hoare爵士（1934～）于1960年发明

考查任一向量区间 $S[lo, hi]$ 。对于任何 $lo \leq mi < hi$, 以元素 $p = S[mi]$ 为界都可分割出前、后子向量 $S_1 = S[lo, mi]$ 和 $S_2 = S[mi+1, hi]$ 。如图 12.1 所示, 若 S_1 中的元素均不大于 p , 且 S_2 中元素均不小于 p , 则元素 p 称作向量 S 的一个轴点 (pivot)。

由以上定义, 元素 $p = S[mi]$ 是轴点, 当且仅当恰有 mi 个元素不大于它。这就意味着, 在最终排序所得的有序向量 S' 中 p 应恰有 mi 个前驱, 即该元素的秩依然保持为 mi 。进一步地, 若元素 p 是某一向量 S 的轴点, 则 S 的以下属性将在 S' 中得以延续:

- 0) p 的秩 (mi) ;
- 1) p 的前驱的最大秩 ($mi-1$) ;
- 2) p 的后继的最小秩 ($mi+1$) 。

因此, 不仅以轴点 p 为界, 前、后子向量 S_1 和 S_2 的排序可各自独立地进行, 而且更重要的是, 一旦前、后子向量完成排序, 即可立即 (在 $\mathcal{O}(1)$ 时间内) 得到整个向量的排序结果。故采用分治策略, 递归地利用轴点的以上特性实现子序列的划分与合并, 即可完成原向量的整体排序——这正是快速排序的总体构思。

12.1.3 快速排序算法

按照以上思路, 可作为向量的一种排序器, 实现如代码 12.1 所示的快速排序算法。

```
1 template <typename T> //向量快速排序
2 void Vector<T>::quickSort(Rank lo, Rank hi) { //assert: 0 < lo <= hi <= size
3     if (hi - lo < 2) return; //单元素区间自然有序, 否则...
4     Rank mi = partition(lo, hi - 1); //在[lo, hi-1]内构造轴点
5     quickSort(lo, mi); //对前缀递归排序
6     quickSort(mi + 1, hi); //对后缀递归排序
7 }
```

代码12.1 向量的快速排序

可见, 算法的关键在于确定轴点位置的 `partition()` 算法。轴点的位置一旦确定, 则只需以轴点为界分别递归地对前、后子向量实施快速排序; 子向量的排序结果返回后, 原向量的整体排序即告完成。

12.1.4 快速划分算法

■ 反例

然而事情远非如此简单, 我们首先遇到的困难在于, 并非每个向量都必然含有轴点。以如图 12.2 所示长度为 9 的向量为例, 不难验证, 其中任何元素都不是轴点。

1	2	3	4	5	6	7	8	0
0	1	2	3	4	5	6	7	8

图12.2 有序向量经循环左移一个单元后, 将不含任何轴点

事实上根据此前的分析, 元素 p 作为轴点的必要条件之一是, 其在初始向量与排序后向量中的秩应当相同。因此一般地, 只要向量中所有元素都是错位的——即所谓的错排序列——则任何元素都不可能是轴点。

尽管如此, 通过适当地调整向量中元素的位置, 依然可以“人为地”构造出一个轴点。

■ 思路

为在区间 $[lo, hi]$ 内构造出一个轴点，可任取某一元素作为“培养对象”。

如图 12.3(a)所示，不妨选首元素

$p = S[lo]$ ，将其从向量中取出并做备份，腾出的空闲单元便于其它元素的位置调整。然后如图(b)，不断试图移动 lo 和 hi 并使之相互靠拢。当然，整个移动过程中均需保证 lo (hi)左侧(右侧)的元素均不大于(不小于) p 。最后如图(c)所示，当 lo 与 hi 重合时，只需将原备份的 p 放回到这一位置，则 p 就成为一个名副其实的轴点。

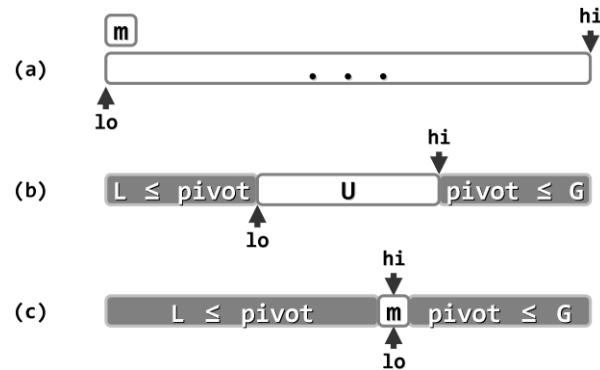


图12.3 轴点构造算法的构思

以上过程在构造出轴点的同时，也按照相对于轴点的大小，将原向量划分为左、右两个子向量，故亦称作快速划分（quick partitioning）算法。

■ 实现

按照以上思路，可以实现轴点构造算法如代码 12.2 所示。

```
1 template <typename T> //轴点构造算法：通过调整元素位置构造区间[lo,hi]的轴点，并返回其秩
2 Rank Vector<T>::partition(Rank lo, Rank hi) { //版本A：基本形式
3     swap(_elem[lo], _elem[lo + rand() % (hi-lo+1)]); //任选一个元素与首元素交换
4     T pivot = _elem[lo]; //以首元素为候选轴点——经以上交换，等效于随机选取
5     while (lo < hi) { //从向量的两端交替地向中间扫描
6         while ((lo < hi) && (pivot <= _elem[hi])) //在不小于pivot的前提下
7             hi--; //向左拓展右端子向量
8         _elem[lo] = _elem[hi]; //小于pivot者归入左侧子序列
9         while ((lo < hi) && (_elem[lo] <= pivot)) //在大小于pivot的前提下
10            lo++; //向右拓展左端子向量
11         _elem[hi] = _elem[lo]; //大于pivot者归入右侧子序列
12     } //assert: lo == hi
13     _elem[lo] = pivot; //将备份的轴点记录置于前、后子向量之间
14     return lo; //返回轴点的秩
15 }
```

代码12.2 轴点构造算法（版本A）

为便于和稍后的改进版本做比较，不妨称之为版本 A。

■ 过程

可见，算法的主体框架为循环迭代；主循环的内部，通过两轮迭代交替地移动 lo 和 hi 。

如图 12.4(a~b)，首先将 $_elem[hi]$ 与 $pivot$ 做比较，只要前者不小于后者就不断向前移动 hi ，直至遇到小于 $pivot$ 的某一元素 $_elem[hi]$ 时终止。此时应如图(b~c)，将该元素挪至 $_elem[lo]$ 并归入左侧子向量。对称地如图(c~d)，随后将 $_elem[lo]$ 与 $pivot$ 做比较，只要前者不大于后者就不断向后移动 lo ，直至遇到大于 $pivot$ 的某一元素 $_elem[lo]$ 。此时应如图(d~e)，将该元素挪至 $_elem[hi]$ 并归入右侧子向量。

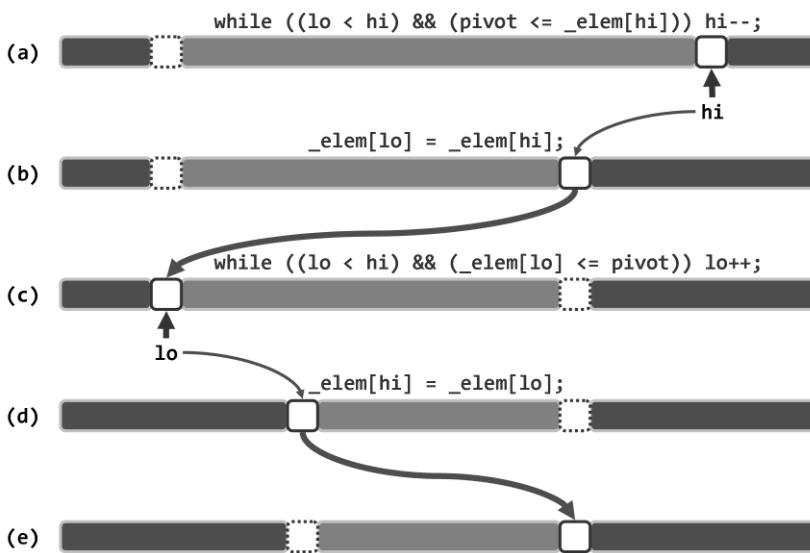


图12.4 轴点构造过程

经如上两轮移动之后，`lo` 与 `hi` 的间距必然缩短，故该算法迟早会终止，且运行时间线性正比于被移动元素的数目，即线性正比于原向量的规模 $hi - lo = O(n)$ 。

当然，如图(e)所示若 `lo` 与 `hi` 仍未重合，则需再执行一次主循环（即两轮内循环）。不难验证，在算法执行过程中的任一时刻，在以 `lo` 和 `hi` 为界的三个子向量中，左、右子向量中的元素分别满足 12.1.2 节所指出的轴点充要条件 1) 和 2)。随着算法的推进，左、右子向量的范围不断扩张，中间子向量的范围则不断压缩。当主循环退出时 `lo` 和 `hi` 重合，充要条件 0) 也随即满足。因此，这时只需将 `pivot` “镶嵌”于左、右子向量之间，即实现了对原向量的一次轴点划分。

■ 稳定性

由于 `lo` 和 `hi` 的移动方向相反，故原处于右（左）端较小（大）的元素将按相反的次序转移至左（右）端；特别地，原处于右（左）端较小（大）的重复元素也将按相反的次序转移至左（右）端，从而不再保持其原有的相对次序。由此可见，快速排序算法并不稳定。

12.1.5 复杂度

■ 最坏情况

上节的分析结论指出，采用代码 12.2 中的 `partition()` 算法，可在线性时间内将原向量的排序问题分解为两个相互独立、总体规模保持线性的子向量排序问题；而且根据轴点的性质，由各自排序后的子向量，可在常数时间内得到整个有序向量。也就是说，分治策略得以高效实现的两个必要条件——子问题划分的高效性及其相互之间的独立性——均可保证。然而尽管如此，另一项关键的必要条件——子任务规模接近——在这里却无法保证。事实上，由 `partition()` 算法划分出的子任务在规模上不仅不能保证接近，而且可能相差悬殊。

反观 `partition()` 算法不难发现，其划分所得子序列的长度与划分的具体过程无关，而是完全取决于入口处所选的候选轴点。具体地，若在最终有序向量中该候选元素的秩为 `r`，

则子向量的规模必为 r 和 $n-r-1$ 。特别地， $r = 0$ 时子向量规模分别为 0 和 $n-1$ ——左侧子向量为空，而右侧子向量与原向量几乎等长。当然，对称的 $r = n-1$ 亦属最坏情况。

更糟糕的是，这类最坏情况可能持续发生。比如，若每次都是简单地选择最左端元素 `_elem[lo]` 作为候选轴点，则对于完全（或几乎完全）有序的输入向量，每次（或几乎每次）划分的结果都是如此。这种情况下，若将快速排序算法处理规模为 n 的向量所需的时间记作 $T(n)$ ，则如下递推关系始终成立：

$$T(n) = T(0) + T(n-1) + \mathcal{O}(n) = T(n-1) + \mathcal{O}(n)$$

综合考虑到其常数复杂度的递归基，与以上递推关系联立即可解得：

$$T(n) = T(n-2) + 2\cdot\mathcal{O}(n) = \dots = T(0) + n\cdot\mathcal{O}(n) = \mathcal{O}(n^2)$$

也就是说，其效率居然低到与起泡排序相近。

■ 降低最坏情况概率

那么，如何才能降低最坏情况出现的概率呢？读者可能已注意到，代码 12.2 的 `partition()` 算法在入口处增加了 `swap()` 一句，在区间内任选一个元素与 `_elem[lo]` 交换。就其效果而言，这使得后续的处理等同于随机选择一个候选轴点，从而在一定程度上降低上述最坏情况出现的概率。这种方法称作随机法。

类似地，也可采用所谓三者取中法：从待排序向量中随机选取三个元素，将其中数值居中者作为候选轴点。理论分析及实验统计均表明，较之固定选取某个元素或随机选取单个元素的策略，如此选出的轴点在最终有序向量中秩过小或过大的概率更低——尽管还不能彻底避免最坏情况的发生。

■ 平均运行时间

以上关于最坏情况下效率仅为 $\mathcal{O}(n^2)$ 的结论不免令人沮丧，难道快速排序名不副实？实际上，更为细致的分析与实验统计都一致地显示，在大多数情况下，快速排序算法的平均效率依然可以达到 $\mathcal{O}(n \log n)$ ；而且较之其它排序算法，其时间复杂度中的常系数更小。以下就以最常见的场景为例，对采用随机法确定候选轴点的快速排序算法的平均效率做一估算。

假设待排序的元素服从独立均匀随机分布。于是，`partition()` 算法在经过 $n-1$ 次比较和至多 $n+1$ 次移动操作之后，对规模为 n 的向量的划分结果无非 n 种可能，划分所得左侧子序列的长度分别是 $0, 1, \dots, n-1$ ，分别决定于所取候选元素在最终有序序列中的秩。按假定条件，每种情况的概率均为 $1/n$ ，故若将算法的平均运行时间记作 $\hat{T}(n)$ ，则有：

$$\begin{aligned}\hat{T}(n) &= (n+1) + (1/n) \times \sum_{k=1}^n [\hat{T}(k-1) + \hat{T}(n-k)] \\ &= (n+1) + (2/n) \times \sum_{k=1}^n \hat{T}(k-1)\end{aligned}$$

等式两侧同时乘以 n ，则有：

$$n \cdot \hat{T}(n) = (n+1)n + 2 \cdot \sum_{k=1}^n \hat{T}(k-1)$$

以及同理：

$$(n-1) \cdot \hat{T}(n-1) = (n-1)n + 2 \cdot \sum_{k=1}^{n-1} \hat{T}(k-1)$$

以上两式相减，即得：

$$\begin{aligned}n \cdot \hat{T}(n) - (n-1) \cdot \hat{T}(n-1) &= 2n + 2 \cdot \hat{T}(n-1) \\n \cdot \hat{T}(n) &= (n+1) \cdot \hat{T}(n-1) + 2n \\\hat{T}(n)/(n+1) &= \hat{T}(n-1)/n + 2/(n+1) \\&= \hat{T}(n-2)/(n-1) + 2/(n+1) + 2/n \\&= \hat{T}(n-3)/(n-2) + 2/(n+1) + 2/n + 2/(n-1) \\&= \dots \\&= \hat{T}(0)/1 + 2/(n+1) + 2/n + 2/(n-1) + \dots + 2/2 \\&= 2 \cdot \sum_{k=1}^{n+1} (1/k) - 1 \\&\stackrel{(2)}{=} O(2 \cdot \ln n) = O(2 \cdot \ln 2 \cdot \log_2 n) = O(1.386 \cdot \log_2 n)\end{aligned}$$

正因为其良好的平均性能，加上其形象直观和易于实现的特点，快速排序算法自诞生起就一直受到人们的青睐，并被集成到 Linux 和 STL 等环境中。

12.1.6 应对退化

■ 重复元素

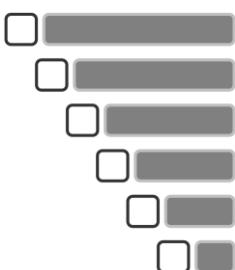


图12.5 `partition()`算法的退化情况也是最坏情况

考查所有(或几乎所有)元素均重复的退化情况。对照代码 12.2 不难发现，`partition()`算法的版本 A 对此类输入的处理完全等效于此前所举的最坏情况。事实上对于此类向量，主循环内部前一子循环的条件中“`pivot <= _elem[hi]`”形同虚设，故该子循环将持续执行，直至“`lo < hi`”不再满足。当然，在此之后另一内循环及主循环也将随即结束。

如图 12.5 所示，如此划分的结果必然是以最左端元素为轴点，原向量被分为极不对称的两个子向量。更糟糕的是，这一最坏情况还可能持续发生，从而使整个算法过程等效地退化为线性递归，递归深度为 $O(n)$ ，导致总体运行时间高达 $O(n^2)$ 。

当然，可以在每次深入递归之前做统一核验，若属于以上退化情况，则无需继续递归而直接返回。不过，在重复元素不多的场合，如此不仅不能改进性能，反而会增加额外的计算量，总体权衡后得不偿失。

■ 改进

轴点构造算法可行的一种改进方案如代码 12.3 所示。为与如代码 12.2 所示同名算法版本 A 相区别，不妨称之为版本 B。

```
1 template <typename T> //轴点构造算法：通过调整元素位置构造区间[lo,hi]的轴点，并返回其秩
2 Rank Vector<T>::partition(Rank lo, Rank hi) { //版本B：可优化处理多个关键码雷同的退化情况
3     swap(_elem[lo], _elem[lo + rand() % (hi-lo+1)]); //任选一个元素与首元素交换
4     T pivot = _elem[lo]; //以首元素为候选轴点——经以上交换，等效于随机选取
5     while (lo < hi) { //从向量的两端交替地向中间扫描
6         while (lo < hi)
```

^② 若记 $h(n) = 1 + 1/2 + 1/3 + \dots + 1/n$ ，则有 $\ln(n+1) = \int_{i=1}^{n+1} (1/x) < h(n) < 1 + \int_{i=1}^n (1/x) = 1 + \ln n$

```
7     if (pivot < _elem[hi]) //在大于pivot的前提下
8         hi--; //向左拓展右端子向量
9     else //直至遇到不大于pivot者
10        { _elem[lo++] = _elem[hi]; break; } //将其归入左端子向量
11    while (lo < hi)
12        if (_elem[lo] < pivot) //在小于pivot的前提下
13            lo++; //向右拓展左端子向量
14        else //直至遇到不小于pivot者
15            { _elem[hi--] = _elem[lo]; break; } //将其归入右端子向量
16    } //assert: lo == hi
17    _elem[lo] = pivot; //将备份的轴点记录置于前、后子向量之间
18    return lo; //返回轴点的秩
19 }
```

代码12.3 轴点构造算法（版本B）

较之版本 A，版本 B 主要是调整了两个内循环的终止条件。以前一内循环为例，原条件

```
pivot <= _elem[hi]
```

在此更改为

```
pivot < _elem[hi]
```

也就是说，一旦遇到元素重复的退化情况，版本 B 将随即终止右端子向量的拓展，并将右端重复的元素转移至左端。因此，如果说此前版本 A 采取的策略是“勤于拓展、懒于交换”，那么改进后版本 B 采用的策略则是“懒于拓展、勤于交换”。

■ 效果及性能

对照代码 12.3 不难验证，对于由重复元素构成的输入向量，以上版本 B 将交替地将右（左）侧元素转移至左（右）侧，并最终恰好将轴点置于正中央的位置。这就意味着，退化的输入向量能够始终被均衡的切分，如此反而转为最好情况，排序所需时间为 $\mathcal{O}(n \log n)$ 。

当然，以上改进并非没有代价。比如，单趟 `partition()` 算法需做更多的元素交换操作。好在这并不影响该算法的线性复杂度。另外，版本 B 倾向于反复交换重复的元素，故它们在原输入向量中的相对次序更难保持，快速排序算法稳定性的不足更是雪上加霜。

§ 12.2 *选取与中位数

12.2.1 概述

■ k-选取

考查如下问题：在任意一组可比较大小的元素中，如何找出由小到大次序为 k 者？如图 12.6(a) 所示，也就是要从与这组元素对应的有序序列 S 中找出秩为 k 的元素 $S[k]$ 。

(a) 0 k n-1

(b) 0 [n/2] n-1

图12.6 选取与中位数

以无序向量 $A = \{3, 13, 2, 5, 8\}$ 为例，对应的有序向量为 $S = \{2, 3, 5, 8, 13\}$ 。故 A 中秩为 0 至 4 的元素依次为 2、3、5、8 和 13。这类计算旨在从与数据集对应的有序

序列中找出具有某一特定秩的元素，故称作选取（selection）问题。通常，若将目标元素的秩记作 k ，对应的问题也称作 k -选取（ k -selection）。

作为 k -选取问题的特例， 0 -选取即通常的最小值问题，而 $(n-1)$ -选取问题即最大值问题。这两个问题都有平凡的最优解，例如代码 3.21 中的 `List::selectMax()`。

注意，在允许元素重复的场合，秩为 k 的元素可能同时存在多个副本。此时不妨约定，其中任何一个都可作为解答输出。

■ 中位数

如图 12.6(b)所示，在长度为 n 的有序序列 S 中，位序居于中央的元素 $S[\lfloor n/2 \rfloor]$ 称作中值或中位数（median）。例如，有序序列 $S = \{2, 3, 5, 8, 13\}$ 的中位数为元素 $S[\lfloor 5/2 \rfloor] = S[2] = 5$ 。对尚未排序的序列也可定义中位数，也就是经排序所得与之对应的有序序列的中位数。例如，无序序列 $A = \{3, 13, 2, 5, 8\}$ 的中位数为元素 $A[3] = 5$ 。

不难看出，中位数的查找即是选取问题在 $k = \lfloor n/2 \rfloor$ 时的特例。由于中位数可将原数据集均衡地划分为大小关系明确的两个子集，故能否高效地确定中位数，将直接关系到快速排序之类基于分治策略算法的高效实现。实际上正如稍后将看到的，作为选取问题的特例，中位数查找也是其中难度最大者。

■ 蛮力算法

查找中位数的一个直觉算法不难理解与实现：对所有元素做排序并得到有序序列 S ，根据定义 $S[\lfloor n/2 \rfloor]$ 即为所要找的中位数。然而根据 2.7.5 节的结论，通常的排序算法在最坏情况下都需要 $\Omega(n \log n)$ 时间。如此，基于中位数的分治算法的时间复杂度将是：

$$T(n) = n \log n + 2 \cdot T(n/2) = O(n \log^2 n)$$

反而不如常规的 $O(n \log n)$ 算法，故这一效率难以令人接受。

因此，以下先结合若干特定情况讨论中位数的定位算法，然后再回到一般性的选取问题。

12.2.2 主流数

■ 问题

为达到热身的目的，不妨先来讨论中位数问题的一个简化版本。在无序向量 A 中，若有一半以上元素的数值同为 m ，则称之为 A 的主流数（majority）。例如，向量 $\{5, 3, 9, 3, 3, 2, 3, 3\}$ 的主流数为 3 ，而向量 $\{5, 3, 9, 3, 1, 2, 3, 3\}$ 则没有主流数。

那么，任给无序向量，如何快速判断其中是否存在主流数，并在存在时将其找出？尽管只是以整数向量为例，以下算法不难推广至元素类型支持判等和比较操作的任意向量。

■ 必要性与充分性

一个不难理解但容易忽略的事实是，如果主流数存在，则必然同时也是中位数。否则，在排序所得的有序向量中，总数超过半数的主流数必然被中位数切分为非空的两部分，这与与此时向量的有序性相悖。这一事实，可做为主流数的一项必要条件。

```
1 template <typename T> bool majority(Vector<T> A, int& maj) { //主流数查找算法：T可比较可判等
2     maj = majEleCandidate(A); //必要性：选出候选者maj
3     return majEleCheck(A, maj); //充分性：验证maj是否的确当选
4 }
```

345

代码12.4 主流数查找算法主体框架

因此，可如代码 12.4 所示调用 `majEleCandidate()` 在向量 A 中找到中位数，并将其作为主流数的唯一候选者。然后再如代码 12.5 所示调用 `majEleCheck()` 扫描一遍向量，通过统计中位数出现的次数，以验证其作为主流数的充分性，从而判断主流数是否的确存在。

```
1 template <typename T> bool majEleCheck(Vector<T> A, T maj) { //验证候选者是否确为主流数
2     int occurrence = 0; //maj在A[]中出现的次数
3     for (int i = 0; i < A.size(); i++) //逐一遍历A[]个元素
4         if (A[i] == maj) occurrence++; //每遇到一次maj，均更新计数器
5     return 2 * occurrence > A.size(); //根据最终的计数值，即可判断是否的确当选
6 }
```

代码12.5 候选主流数核对算法

那么，在尚未得到高效的中位数查找算法之前，又该如何解决主流数问题呢？

■ 减而治之

关于主流数的另一重要事实如图 12.7 所示：在向量 A 的（长度为偶数的）前缀 P 中，若某一元素 x 出现的次数恰占半数，则 A 有主流数仅当对应的后缀 A-P



图12.7 通过减治策略计算主流数

实际上，因为最终会对充分性另作核对，所以 A 不含主流数的情况不必担心。因此，需要验证的无非是 A 的确含有主流数的两种情况。首先，若元素 x 就是 A 的主流数，则在剪除前缀 P 之后，x 与非主流数减少相同的数目，两类元素数量的相对差距在后缀 A-P 中保持不变。反过来，若 P 中另一非 x 的元素为 A 的主流数，则在剪除前缀 P 之后，该主流数减少的数目也不多于其余的非主流数，两类元素数量的相对差距在后缀 A-P 中也不会缩小。

■ 实现

基于以上减而治之的思路，可通过如代码 12.6 所示的 `majEleCandidate()` 算法，确定唯一一个满足以上必要条件的候选者。

```
1 template <typename T> T majEleCandidate(Vector<T> A) { //选出具备必要条件的主流数候选者
2     T maj; //主流数候选者
3     // 线性扫描，并借助计数器c，记录主流数候选者与其它元素的数量差额
4     for (int c = 0, i = 0; i < A.size(); i++)
5         if (0 == c) { //若c归零，则意味着此时的前缀P可剪除
6             maj = A[i]; c = 1; //主流数候选者改为新的当前元素
7         } else //否则
8             (maj == A[i]) ? c++ : c--; //相应地更新差额计数器
9     return maj; //至此，原向量的主流数若存在，则只能是maj —— 尽管反之不然
10 }
```

代码12.6 候选主流数选取算法

该算法自左向右扫描向量一遍。扫描过程中维护一个计数器 c，c 的每次归零，都等效于剪除一段前缀并相应地缩小问题的规模。

变量 `maj` 则始终记录当前前缀中某一出现次数不低于一半的元素：该元素每出现一次，`c` 就相应地增加计数；反之，每次遇到其它的元素，则 `c` 相应地减少计数。

根据以上的分析，一旦 c 归零则意味着如图 12.7(b)所示，在当前向量中找到了一个可剪除的前缀 P 。在剪除该前缀之后，只需将 maj 再次初始化为后缀的首元素，并令 $c = 1$ ，即可继续重复该算法。当整个向量均已扫描过后， maj 即是唯一满足必要条件的候选者。

12.2.3 归并向量的中位数

■ 问题

本节继续讨论中位数问题的另一简化版本。考查如下问题：任给两个有序向量 S_1 和 S_2 ，如何快速找出它们归并后所得有序向量 $S = S_1 \cup S_2$ 的中位数？

■ 蛮力算法

```
1 // 中位数算法蛮力版：效率低，仅适用于max(n1, n2)较小的情况
2 template <typename T> //子序列S1[lo1, lo1+n1)和S2[lo2, lo2+n2)分别有序，数据项可能重复
3 T trivialMedian(Vector<T>& S1, int lo1, int n1, Vector<T>& S2, int lo2, int n2) {
4     Vector<T> S; int i1 = 0, i2 = 0;
5     while ((i1 < n1) && (i2 < n2)) {
6         while ((i1 < n1) && S1[lo1 + i1] <= S2[lo2 + i2]) S.insert(S.size(), S1[lo1 + i1++]);
7         while ((i2 < n2) && S2[lo2 + i2] <= S1[lo1 + i1]) S.insert(S.size(), S2[lo2 + i2++]);
8     }
9     while (i1 < n1) S.insert(S.size(), S1[lo1 + i1++]);
10    while (i2 < n2) S.insert(S.size(), S2[lo2 + i2++]);
11    return S[(n1+n2)/2];
12 }
```

代码12.7 中位数蛮力查找算法

诚然，有序向量 S 中的元素 $S[\lfloor (n_1 + n_2)/2 \rfloor]$ 即为中位数，但若果真按代码 12.7 中蛮力算法 `trivialMedian()` 将二者归并，则需花费 $\mathcal{O}(n_1 + n_2)$ 时间。这一效率虽不算太低，但毕竟未能充分利用“两个子向量已经有序”的条件。那么，能否更快地完成这一任务呢？

以下首先针对 S_1 和 S_2 长度同为 n 的情况给出算法，然后推广至不等长的情况。

■ 减而治之

这里，我们将再次采用减而治之的策略。如图 12.8 所示，考查前一向量的中位数 $m_1 = S_1[\lfloor n/2 \rfloor]$ 和后一向量的逆向中位数 $m_2 = S_2[\lceil n/2 \rceil]$ ，并比较二者的大小。其中图(a)和图(b)分别对应于向量长度 n 为偶数和奇数的情况。

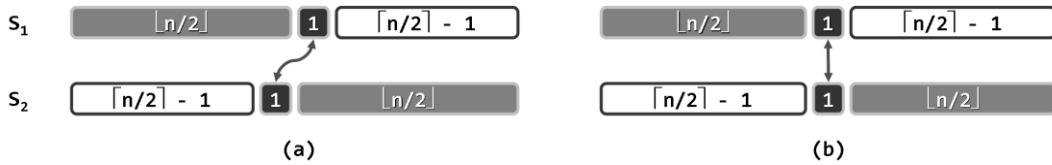


图12.8 采用减治策略，计算等长有序向量归并后的中位数

若 $m_1 = m_2$ ，则意味着在 $S = S_1 \cup S_2$ 中，各有 $\lfloor n/2 \rfloor + (\lceil n/2 \rceil - 1) = n - 1$ 个元素不大于和不小于它们，亦即 m_1 和 m_2 就是 S 的中位数。若 $m_1 < m_2$ ，则意味着在 S 中各有 $\lfloor n/2 \rfloor$ 个元素不大于和不小于它们。这些元素所属区间在图中以灰色示意。可见，这些元素或者不是 S 的中位数，或者与 m_1 或 m_2 同为 S 的中位数。总之无论如何，在清除这些元素之后， S

中位数的数值均保持不变。 $m_1 > m_2$ 的对称情况与此类似。

综合以上分析，只需进行一次比较，即可将原问题的规模缩减大致一半。利用这一性质反复递归，问题的规模将以 $1/2$ 为比例按几何级数的速度递减，直至平凡的递归基。整个算法呈线性递归的形式，递归深度不超过 $\log_2 n$ ，每一递归实例仅需常数时间，故总体时间复杂度为 $O(\log n)$ ，这一效率远远高于蛮力算法。

■ 实现

基于以上减而治之的思路，可实现如代码 12.8 所示的 median() 算法。

```
1 template <typename T> //等长子序列S1[lo1, lo1+n)和S2[lo2, lo2+n)分别有序, n > 0, 数据项可能重复
2 T median(Vector<T>& S1, int lo1, Vector<T>& S2, int lo2, int n) { //中位数算法(高效版)
3     if (n < 3) return trivialMedian(S1, lo1, n, S2, lo2, n); //递归基
4     int mi1 = lo1 + n / 2, mi2 = lo2 + (n - 1) / 2; //长度(接近)减半
5     if (S1[mi1] < S2[mi2])
6         return median(S1, mi1, S2, lo2, n + lo1 - mi1); //取S1右半、S2左半
7     else if (S1[mi1] > S2[mi2])
8         return median(S1, lo1, S2, mi2, n + lo2 - mi2); //取S1左半、S2右半
9     else
10        return S1[mi1];
11 }
```

代码12.8 等长有序向量归并后中位数算法

这里，在向量长度均小于 3 之后，即调用蛮力算法 trivialMedian 直接计算中位数。在向量依然足够长时，分别取出 m_1 和 m_2 ，并按三种情况继续线性递归。因属于尾递归，故不难将其改写为迭代形式（习题[6]）。

■ 一般情况

以上算法可如代码 12.9 所示推广至一般情况，即允许有序向量 S_1 和 S_2 的长度不等。

```
1 template <typename T> //子序列S1[lo1, lo1+n1)和S2[lo2, lo2+n2)分别有序, 数据项可能重复
2 T median(Vector<T>& S1, int lo1, int n1, Vector<T>& S2, int lo2, int n2) { //中位数算法
3     if (n1 > n2) return median(S2, lo2, n2, S1, lo1, n1); //确保n1 <= n2
4     /////////////////////////////////
5     if (n2 < 6) //递归基: 1 <= n1 <= n2 <= 5
6         return trivialMedian(S1, lo1, n1, S2, lo2, n2);
7     /////////////////////////////////
8     // n1 << n2
9     /////////////////////////////////
10    //          lo1          lo1 + n1/2          lo1 + n1 - 1
11    //          |              |                  |
12    //          X >>>>>>>>>> X >>>>>>>>>> X
13    // Y .. trimmed .. Y >>>>>>>>>> Y >>>>>>>>>> Y .. trimmed .. Y
14    // |          |          |          |          |
15    // lo2      lo2 + (n2-n1)/2      lo2 + n2/2      lo2 + (n2+n1)/2      lo2 + n2 - 1
16    /////////////////////////////////
17    if (2 * n1 < n2) //子序列长度相差悬殊时, 长者的两翼可直接截除
```

```
18     return median(S1, lo1, n1, S2, lo2 + (n2 - n1 - 1) / 2, n1 + 2 - (n2 - n1) % 2);
19     ///////////////////////////////////////////////////////////////////
20     //    lo1           lo1 + n1/2           lo1 + n1 - 1
21     //    |             |                   |
22     //    X >>>>>>>>>>>>>>> X >>>>>>>>>>>>>>>> X
23     //                           |
24     //                           m1
25     ///////////////////////////////////////////////////////////////////
26     //                           mi2b
27     //                           |
28     // lo2 + n2 - 1           lo2 + n2 - 1 - n1/2
29     //   |           |
30     //   Y <<<<<<<<<<<<< Y ...
31     //
32     //
33     //
34     //
35     //
36     //
37     //
38     //           ...
39     //           |           |
40     //           lo2 + (n1-1)/2           lo2
41     //           |
42     //           mi2a
43     ///////////////////////////////////////////////////////////////////
44     int mi1 = lo1 + n1 / 2;
45     int mi2a = lo2 + (n1 - 1) / 2;
46     int mi2b = lo2 + n2 - 1 - n1 / 2;
47     if (S1[mi1] > S2[mi2b]) //取S1左半、S2右半
48         return median(S1, lo1, n1 / 2 + 1, S2, mi2a, n2 - (n1 - 1) / 2);
49     else if (S1[mi1] < S2[mi2a]) //取S1右半、S2左半
50         return median(S1, mi1, (n1 + 1) / 2, S2, lo2, n2 - n1 / 2);
51     else //S1保留，S2左右同时缩短
52         return median(S1, lo1, n1, S2, mi2a, n2 - (n1 - 1) / 2 * 2);
53 }
```

代码12.9 不等长有序向量归并后中位数算法

这一算法与代码 12.8 中同名算法的思路基本一致，请参照注释分析和验证其功能。

同样地，由于这里也采用了减而治之的策略，可使问题的规模大致按几何级数递减，故总体复杂度亦为 $\mathcal{O}(\log(n_1 + n_2))$ 。实际上更精确地，其复杂度应为 $\mathcal{O}(\log(\min(n_1, n_2)))$ （习题[7]）——也就是说，子向量长度相等或接近时，此类问题的难度更大。

12.2.4 基于优先级队列的选取

■ 信息量与计算成本

回到一般性的选取问题。与中位数一样，直接基于排序的蛮力算法无法令人满意的原因可以解释为，“一组元素中第 k 大的元素”所包含的信息量，远远少于经全排序得到的有序序列。如此看来，花费足以全排序的计算成本仅得到少量的局部信息，自然属于得不偿失。

对这一现象的逆向理解，为我们提供了新的启示：既然只需获取原数据集的局部信息，为何不采用适宜于这类计算需求的优先级队列结构呢？

■ 堆

以堆结构为例。如图 12.9 所示，基于堆结构的选取算法大致有三种。

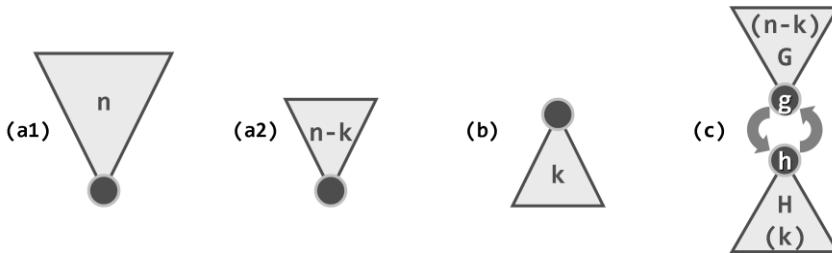


图12.9 基于堆结构的选取算法

第一种算法如图(a1)。首先花费 $\mathcal{O}(n)$ 时间将全体元素组织为一个小顶堆，然后经过 k 次 $\text{delMin}()$ 操作，则如图(a₂)得到位序为 k 的元素。这一算法的运行时间为：

$$\mathcal{O}(n) + k \cdot \mathcal{O}(\log n) = \mathcal{O}(n + k \log n)$$

另一算法如图(b)。任取 k 个元素并用 $\mathcal{O}(k)$ 时间将其组织为大顶堆，然后交替执行 $\text{insert}()$ 和 $\text{delMax}()$ 操作各 $n-k$ 次，依次插入剩余的 $n-k$ 个元素，并随即通过删除操作使堆的规模保持为 k 。所有元素处理完毕后，堆顶即为目标元素。该算法的运行时间为：

$$\mathcal{O}(k) + 2(n-k) \cdot \mathcal{O}(\log k) = \mathcal{O}(k + 2(n-k)\log k)$$

最后一种方法如图(c)。首先将全体元素分为两组，分别构建一个规模为 $n-k$ 的小顶堆 G 和一个规模为 k 的大顶堆 H 。接下来，反复比较它们的堆顶 g 和 h ，只要 $g < h$ 则将二者交换并重新调整两个堆。如此， G 的堆顶 g 不断增大， H 的堆顶 h 不断减小。当 $g \geq h$ 时， h 即为所要找的元素。这一算法的运行时间为：

$$\mathcal{O}(n-k) + \mathcal{O}(k) + \min(k, n-k) \cdot 2 \cdot (\mathcal{O}(\log k + \log(n-k)))$$

在目标元素的秩很小或很大（即 $|n/2 - k| \approx n/2$ ）时，上述算法的性能都还不错。比如， $k \approx 0$ 时，前两种算法均只需 $\mathcal{O}(n)$ 时间。然而很遗憾，当 $k \approx n/2$ 时，以上算法的复杂度均退化至蛮力算法的 $\mathcal{O}(n \log n)$ 。因此，我们不得不转而从其它角度寻找突破口。

12.2.5 基于快速划分的选取

■ 秩、轴点与快速划分

选取问题所查找元素的位序 k ，就是其在对应的有序序列中的秩。就这一性质而言，该元素与轴点颇为相似。尽管 12.1.4 节的快速划分算法只能随机地构造一个轴点，但若反复应用这一算法，应该可以逐步逼近目标 k 。

■ 逐步逼近

以上构思可细化如下。首先，调用算法 `partition()` 构造向量 A 的一个轴点 $A[i] = x$ 。若 $i = k$ ，则该轴点恰好就是待选取的目标元素，即可直接将其返回。

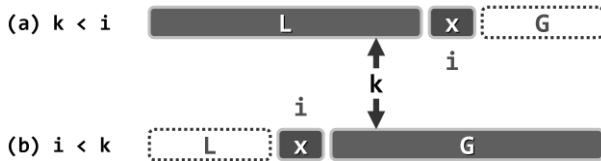


图12.10 基于快速划分算法逐步逼近选取目标元素

反之，若如图 12.10 所示 $i \neq k$ ，则无非两种情况。若如图(a)， $k < i$ ，则选取的目标元素不可能（仅）来自于处于 x 右侧、不小于 x 的子向量（白色）G 中。此时，不妨将子向量 G 剪除，然后递归地在剩余区间继续做 k -选取。反之若如图(b)， $i < k$ ，则选取的目标元素不可能（仅）来自于处于 x 左侧、不大于 x 的子向量（白色）L 中。同理，此时也可将子向量 L 剪除，然后递归地在剩余区间继续做($k-i$)-选取。

■ 实现

基于以上减而治之、逐步逼近的思路，可实现 `quickSelect()` 算法如代码 12.10 所示。

```
1 template <typename T> void quickSelect(Vector<T> & A, Rank k) { //基于快速划分的k选取算法
2     for (Rank lo = 0, hi = A.size() - 1; lo < hi; ) {
3         Rank i = lo, j = hi; T pivot = A[lo];
4         while (i < j) { //O(hi-lo+1) = O(n)
5             while ((i < j) && (pivot <= A[j])) j--; A[i] = A[j];
6             while ((i < j) && (A[i] <= pivot)) i++; A[j] = A[i];
7         } //assert: i == j
8         A[i] = pivot;
9         if (k <= i) hi = i - 1;
10        if (i <= k) lo = i + 1;
11    } //A[k] is now a pivot
12 }
```

代码12.10 基于快速划分的k-选取算法

该算法中主循环体内部的流程与代码 12.2 中的 `partition()` 算法（版本 A）如出一辙。每经过一次主迭代，都会构造出一个新的轴点 $A[i]$ ，然后 lo 或 hi 会相应地通过移动朝对方靠拢，选取目标的查找范围也进一步收缩。当轴点的秩 i 恰为 k 时， lo 和 hi 同时移动并跨越对方，算法随即终止。如此， $A[k]$ 即是查找的目标元素。

尽管内循环仅需 $\mathcal{O}(hi-lo+1)$ 时间，但很遗憾，外循环的次数却无法有效控制。与快速排序算法一样，最坏情况下外循环需执行 $\Omega(n)$ 次（习题 [11]），总体运行时间为 $\mathcal{O}(n^2)$ 。

12.2.6 k-选取算法

以上从多个角度所做的尝试尽管有所收获，但就 k -选取问题在最坏情况下的求解效率这一最终指标而言，均无实质性的突破。本节将延续以上 `quickSelect()` 算法的思路，介绍一个在最坏情况下运行时间依然为 $\mathcal{O}(n)$ 的 k -选取算法。

■ 算法

该算法的主要流程可描述如下。

```
select(A, k)
输入：规模为n的无序序列A，秩k ≥ 0
输出：A所对应有序序列中秩为k的元素
{
    0) if (n = |A| < Q) return trivialSelection(A, k); //递归基：序列规模不大时直接使用蛮力算法
        1) 将A均匀地划分为n/Q个子序列，各含Q个元素；//Q为一个不大的常数，其具体数值稍后给出
        2) 各子序列分别排序，计算中位数，并将这些中位数组成一个序列；//可采用任何排序算法，比如选择排序
        3) 通过递归调用select()，计算出中位数序列的中位数，记作M；
        4) 根据其相对于M的大小，将A中元素分为三个子集：L（小于）、E（相等）和G（大于）；
        5) if (|L| ≥ k) return select(L, k);
            else if (|L| + |E| ≥ k) return M;
            else return select(G, k - |L| - |E|);
}
```

算法12.1 线性时间的k-选取

■ 正确性

该算法正确性的关键在于其中第5步中所涉及的递归。

实际上如图12.11所示，在第4步依据全局中位数M对所有元素做过分类之后，可以假想地将三个子序列L、E和G按照大小次序自左向右排列。尽管这三个子集都有可能是空集，但无论如何，k-选取目标元素的位置无非三种可能。

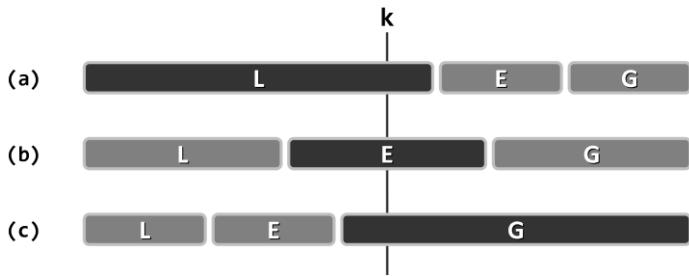


图12.11 k-选取目标元素所处位置的三种可能情况

其一如图(a)，子序列L足够长($|L| \geq k$)。此时，子序列E和G的存在与否与k-选取的结果无关，故可将它们剪除，并在L中继续做递归的k-选取。

其次如图(b)，子序列L长度不足k，但在加入子序列E之后可以覆盖k。此时，E中任何一个元素(均等于全局中位数M)都是所要查找的目标元素，故可直接返回M。

最后如图(c)，子序列L和E的长度总和仍不足k。此时，目标元素必然落在子序列G中，故可将L和E剪除，并在G中继续做递归的($k - |L| - |E|$)-选取。

■ 复杂度

将该select()算法在最坏情况下的运行时间记作T(n)，其中n为输入序列A的规模。

显然，第1步只需 $\mathcal{O}(n)$ 时间。既然Q为常数，故第2步中每一子序列的排序及中位数的计算只需常数时间，累计不过 $\mathcal{O}(n)$ 。第3步为递归调用，因子序列长度为 n/Q ，故经过 $T(n/Q)$ 时间即可得到全局中位数M。第4步依据M对所有元素做分类，为此只需做一趟线性遍历，累计亦不过 $\mathcal{O}(n)$ 时间。

那么，第5步需要运行多少时间呢？

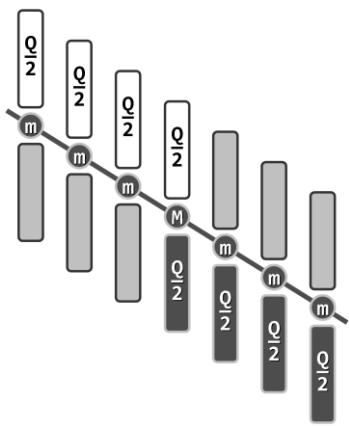


图12.12 各子序列的中位数以及全局中位数

考查第 2 步所得各子序列的中位数。若按照这 n/Q 个中位数的大小次序将其所属子序列顺序排列，大致应如图 12.12 所示。其中居中者，即为第 3 步计算出的全局中位数 M 。

由该图不难发现，至少有一半的子序列中有半数的元素不小于 M （在图中以白色示意）。同理，也至少有一半的子序列中有半数的元素不大于 M （在图中以黑色示意）。反过来，这两条性质也意味着，严格大于（小于） M 的元素在全体元素中所占比例不会超过 75%。由此可知，子序列 L 与 G 的规模均不超过 $3n/4$ 。也就是说，算法的第 5 步尽管会发生递归，但需进一步处理的序列的规模不超过原序列的 $3/4$ 。

综上，可得递推关系如下：

$$T(n) = cn + T(n/Q) + T(3n/4), c \text{ 为常数}$$

若取 $Q = 5$ ，则有

$$T(n) = cn + T(n/5) + T(3n/4) = O(20cn) = O(n)$$

■ 综合评价

上述 `selection()` 算法从理论上证实，的确可以在线性时间内完成 k -选取。然而很遗憾，其线性复杂度中的常系数项过大，以致在通常规模的应用中难以真正体现出效率的优势。

该算法的核心技巧在于第 2 和 3 步，通过高效地将元素分组、分别计算中位数并递归计算出这些中位数的中位数 M ，使问题的规模得以按几何级数的速度递减，从而最终优化算法的整体性能。由此也可看出，中位数算法在一般性 k -选取问题的求解过程中扮演着关键性角色，尽管前者只不过是后者的一个特例，但反过来也是其中难度最大者。

§ 12.3 *希尔排序

12.3.1 递减增量策略

■ 增量

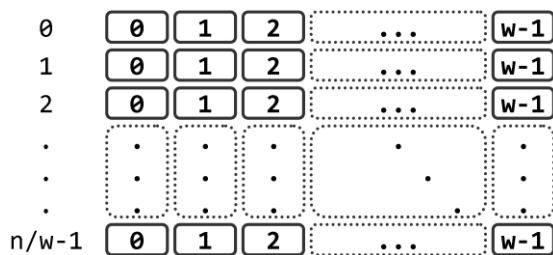


图12.13 将待排序序列表视作二维矩阵

希尔排序^③（Shell sort）算法首先将整个待排序序列 $A[]$ 等效地视作一个二维矩阵 $B[][]$ 。于是如图 12.13 所示，若原一维序列为 $A[0, n)$ ，则对于任一固定的矩阵宽度 w ， A 与 B 中元素之间总有一一对应关系：

$$B[i][j] = A[i + jw]$$

$$A[k] = B[k \% w][k / w]$$

^③ 最初版本由 Donald L. Shell 与 1959 年发明

形象地说，也就是从上到下、自左而右地将原序列 A 中元素依次填入二维矩阵 B 的各个单元。为简化起见，这里不妨假设 w 整除 n。如此，B 中同属一列的元素自上而下对应于 A 中以 w 为间隔的 n/w 个元素。因此，矩阵的宽度 w 亦称作增量（increment）。

■ 算法框架

希尔排序的算法框架可以扼要地描述如下：

```
ShellSort(A, n)
输入：规模为n的无序序列A
输出：A对应的有序序列
{
    取一个递增的增量序列： $\pi = \{w_1 = 1, w_2, w_3, \dots, w_k, \dots\}$ 
    设  $k = \max\{i \mid w_i < n\}$ , 即  $w_k$  为增量序列  $\pi$  中小于  $n$  的最后一项
    for ( $t = k; t > 0; t--$ ) {
        将序列A视作以  $w_t$  为宽度的矩阵  $B_t$ 
        对  $B_t$  的每一列分别排序： $B_t[i], i = 0, 1, \dots, w_t - 1$ 
    }
}
```

算法12.2 希尔排序

■ 增量序列

如图 12.14 所示，希尔排序是个迭代重复的过程。每次迭代中，都从事先设定的某个整数序列中取出一项，并以该项为宽度将输入序列整理为对应宽度的二维矩阵，然后逐列分别排序。当然，各次迭代并不需要真地重组原序列，事实上借助以上一一对应关系即可便捷地根据在 $B[][]$ 中的下标访问统一保存于 $A[]$ 中的元素。

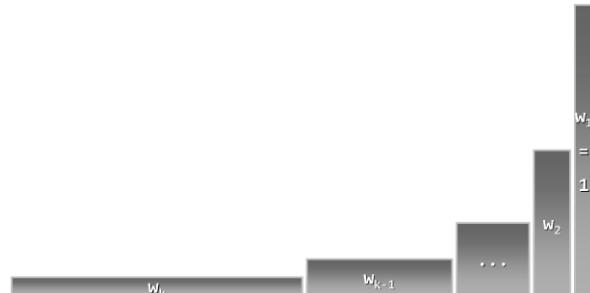


图12.14 递减增量、逐渐逼近策略

因为增量序列中的各项是逆向取出的，所以各次迭代中矩阵的宽度呈缩减的趋势，直至最终使用 $w_1 = 1$ 。矩阵每缩减一次并逐列排序一轮，序列整体的有序性就得以改善。当增量缩减至 1 时，如图最右侧所示，矩阵退化为单独的一列，故最后一次迭代中的“逐列排序”等效于对整个序列执行一次排序。这种通过不断缩减矩阵宽度而逐渐逼近的策略，称作递减增量（diminishing increment）法，也是希尔排序的另一名称。

■ 有序性与效率

由于希尔排序最后一轮必然对整个序列做一次排序，所以无论此前各次迭代如何，最终的结果必然是所需的有序序列，故希尔排序的正确性毋庸置疑。然而反过来我们不禁有个疑问：既然如此，此前各次迭代中的逐列排序有何必要？为何不直接做最后一次排序呢？

有趣的是，希尔排序中逐列排序时所用的算法不必过于讲究，3.5.2 节中的插入排序之类 $O(n^2)$ 算法足矣。事实上， $O(n^2)$ 只是这类算法在最坏情况下的复杂度；随着输入序列的有序性不断提高，此类算法所需的时间将会锐减。比如由第 3 章习题 [9]，当序列中逆序元

素之间的距离均不超过 k 时，插入排序的运行时间仅为 $\mathcal{O}(kn)$ 。这一特性对希尔排序而言至关重要，只要充分加以利用，即可通过前面的迭代不断改善输入序列的有序性，从而降低后续各次迭代的成本，并最终将总体运行时间控制在足以令人满意的范围。

如算法 12.2 所示，希尔排序算法的主体框架已经固定，唯一可以调整的只是增量序列的设计与选用。事实上这一点也的确十分关键，不同的增量序列对插入排序以上特性的利用程度各异，算法的整体效率也相应地差异极大。

12.3.2 增量序列

■ Shell 序列

首先考查 Shell 本人在提出希尔算法之初所使用的序列：

$$\pi_{\text{shell}} = \{1, 2, 4, 8, 16, 32, \dots, 2^k, \dots\}$$

我们将看到，若使用这一序列，希尔排序算法在最坏情况下的性能并不好。

为构造这样一个最坏的输入序列实例，不妨取 $[0, 2^N]$ 内所有的 $n = 2^N$ 个整数。接下来，将这些整数分为 $[0, 2^{N-1}]$ 和 $[2^{N-1}, 2^N]$ 两组，再分别打乱次序后组成两个随机子序列，最后将两个子序列按照交替的次序归并为一个序列。比如， $N = 4$ 时得到的序列可能如下（为便于分辨，来自两个子序列的元素的位置分别做了适当的下移和提升）：

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

请注意，在 π_{shell} 中，首项之外的其余各项均为偶数。因此，在最后一次迭代之前，这两组元素的秩依然保持最初的奇偶性不变。如果把它们分别比作井水与河水，则尽管井水与河水各自都在流动，但毕竟“井水不犯河水”。

特别地，在经过倒数第二次迭代 ($w_2 = 2$) 之后，尽管两组元素已经分别排序，但二者依然恪守各自的秩的奇偶性。仍以 $N = 4$ 为例，此时序列中各元素应排列如下：

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

准确地，此时元素 k 的秩为 $(2k+1) \% (2^{N-1})$ 。对于每一 $1 \leq k \leq 2^{N-1}$ ，与其在最终有序序列中相距 k 个单元的元素各有 2 个，故最后一轮插入排序所做比较操作次数共计：

$$2 \times (1 + 2 + 3 + \dots + 2^{N-1}) = 2^{N-1} \cdot (2^{N-1} + 1) = \mathcal{O}(n^2)$$

反观这一实例可以看出，导致最后一轮排序的低效率的直接原因在于，此前的各次迭代尽管可以改善两组元素各自内部的有序性，但对二者之间有序性的改善却于事无补。究其根源在于，序列 π_{shell} 中除首项外各项均被 2 整除。由此我们可以得到启发——为改进希尔排序的总体性能，首先必须尽可能减少不同增量值之间的公共因子。为此，一种彻底的方法就是保证它们之间两两互素。

不过，为更好地理解和分析如此设计的其它增量序列，需要略做一番准备。

■ 邮资问题

考查如下问题：假设在某个国家，邮局仅发行面值分别为 3 分和 7 分的两种邮票，那么准备邮寄平信的你，可否用这两种邮票组合出对应的 16 分邮资？

略作思考后不难给出解答：使用三张 3 分面值的邮票，另加一张 7 分的。那么进一步地，如果换成邮资为 11 分的明信片，又该如何呢？

事实情况是，这一回无论你如何绞尽脑汁，也不可能给出一种恰好得的组合方案。

■ 线性组合

用数论的语言，以上问题可描述为： $3m + 7n = 11$ 是否存在自然数（非负整数）解？

对于任意自然数 g 和 h ，只要 m 和 n 也是自然数，则 $f = mg + nh$ 都称作 g 和 h 的一个组合（combination）。我们将不能由 g 和 h 组合生成出来的最大自然数记作 $x(g, h)$ 。

这里需要用到数论的一个基本结论：如果 g 和 h 互素，则必有

$$x(g, h) = (g-1)(h-1) - 1 = gh - g - h$$

就以上邮资问题而言， $g = 3$ 与 $h = 7$ 互素，故有

$$x(3, 7) = 2 \times 6 - 1 = 11$$

也就是说，11 恰为无法由 3 和 7 组合生成的最大自然数。

■ h -有序与 h -排序

在序列 $\{a_0, a_1, \dots, a_{n-1}\}$ 中，若 $a_i \leq a_{i+h}$ 对于任何 $0 \leq i < n-h-1$ 均成立，则称该序列 h -有序 (h -ordered)。也就是说，其中相距 h 个单元的每对元素之间均有序。

考查希尔排序中对应于任一增量 h 的迭代。如前所述，该次迭代需将原序列“折叠”成宽度为 h 的矩阵，并对各列分别排序。就效果而言，这等同于在原序列中以 h 为间隔排序，故这一过程称作 h -排序 (h -sorting)。不难看出，经 h -排序之后的序列必然 h -有序。

关于 h -有序和 h -排序，Knuth^④给出了一个重要结论： g -有序的序列再经 h -排序之后，依然保持 g -有序。也就是说，此时该序列既是 g -有序的也是 h -有序的，称作 (g, h) -有序。

反复运用以上结论不难得出以下推论： (g, h) -有序序列必然 $(g+h)$ -有序；同时，对于任意自然数 m 和 n ，该序列也必然 $(mg+nh)$ -有序。

■ 有序性的改善

不难看出，随着 h 不断递减， h -有序序列整体的有序性逐步提高。特别地， 1 -有序的序列即是全局有序的序列。为更准确地验证以上判断，可以如图 12.15 所示，考查与任一元素 a_i 逆序的后继元素。



图12.15 经多次迭代，逆序元素可能的范围必然不断缩小

在分别做过 g -排序与 h -排序之后，根据 Knuth 的结论，该序列应已 (g, h) -有序。根据以上推论，对于 g 和 h 的任一组合 $mg+nh$ ，该序列也应 $(mg+nh)$ -有序。因此反过来， a_i 的后继元素若与之逆序，则该元素与 a_i 的间距必不可能是 g 和 h 的组合。于是根据此前所引的数论结论，倘若 g 和 h 互素，则逆序元素与 a_i 的间距不可能大于 $(g-1)(h-1)$ 。由此可见，在希尔排序的过程中，序列的有序性的确在不断改进。

■ (g, h) -有序与排序成本

设某序列已 (g, h) -有序，现假设 g 和 h 的数值均处于 $\mathcal{O}(d)$ 数量级，以下考查对该序列做 d -排序的时间成本。

据其定义， d -排序共涉及长度均为 $\mathcal{O}(n/d)$ 的 d 个子序列。由以上分析，在 (g, h) -有

^④ D. E. Knuth, The Art of Computer Programming, Vol.3, p.90, Theorem K

序序列中逆序元素的距离不超过 $(g-1)(h-1)$ ，故在这些子序列中逆序元素的距离不超过 $(g-1)(h-1)/d = O(d)$

再次根据第 3 章习题[9]的结论，使用插入排序可在 $O(d \cdot n/2) = O(n)$ 时间内完成每一子序列的排序，于是所有子序列的排序总体消耗的时间应不超过 $O(dn)$ 。

■ Papernov-Stasevic 序列

现在，可以回到增量序列的优化设计问题。按照此前“尽力避免增量值之间公共因子”的思路，Papernov 和 Stasevic 于 1965 年提出了另一增量序列：

$$\mathcal{H}_{ps} = \{1, 3, 7, 15, 31, 63, \dots, 2^{k-1}, \dots\}$$

不难看出，其中各项的确两两互素。我们将看到，采用这一增量序列，希尔排序算法的性能可以改进至 $O(n^{3/2})$ ，其中 n 为待排序序列的规模。

在序列 \mathcal{H}_{ps} 的各项中，设 w_t 为与 $n^{1/2}$ 最接近者，亦即 $w_t = \Theta(n^{1/2})$ 。以下将希尔排序算法过程中的所有迭代分为两类，分别估计其运行时间。

首先，考查在 w_t 之前执行的各次迭代。这类迭代所对应的增量均满足 $w_k > w_t$ ，或等价地， $k > t$ 。在每一次这类迭代中，矩阵共有 w_k 列，各列包含 $O(n/w_k)$ 个元素。因此，若采用插入排序算法，各列分别耗时 $O((n/w_k)^2)$ ，所有列共计耗时 $O(n^2/w_k)$ 。于是，此类迭代各自所需的时间 $O(n^2/w_k)$ 构成一个大致以 2 为比例的几何级数，其总和也应线性正比于其中最大的一项，亦即不超过

$$O(2 \cdot n^2/w_t) = O(n^{3/2})$$

对称地，再来考查 w_t 之后的各次迭代。这类迭代所对应的增量均满足 $w_k < w_t$ ，或等价地， $k < t$ 。考虑到此前刚刚完成 w_{k+1} -排序和 w_{k+2} -排序，而来自 \mathcal{H}_{ps} 序列的 w_{k+1} 和 w_{k+2} 必然互素，且与 w_k 同处一个数量级。因此根据此前结论，每一次这样的迭代至多需要 $O(n \cdot w_k)$ 时间。同样地，这类迭代所需的时间 $O(n \cdot w_k)$ 也构成一个大致以 2 为比例的几何级数，其总和也应线性正比于其中最大的一项，亦即不超过

$$O(2 \cdot n \cdot w_t) = O(n^{3/2})$$

综上可知，采用 \mathcal{H}_{ps} 序列的希尔排序算法，在最坏情况下的运行时间不超过 $O(n^{3/2})$ 。

■ Pratt 序列

Pratt 于 1971 年也提出了自己的增量序列：

$$\mathcal{H}_{pratt} = \{1, 2, 3, 4, 6, 8, 9, 12, 16, \dots, 2^p 3^q, \dots\}$$

可见，其中各项除 2 和 3 外均不含其它素因子。

可以证明，采用 \mathcal{H}_{pratt} 序列，希尔排序算法至多运行 $O(n \log^2 n)$ 时间（习题[12]）。

■ Sedgewick 序列

尽管 Pratt 序列效率很高，但因其中各项的间距太小，导致迭代趟数过多。为此，Sedgewick 吸收 Papernov-Stasevic 序列与 Pratt 序列的优点，提出了以下增量序列：

$$\mathcal{H}_{sedgewick} = \{1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, \dots\}$$

其中各项，均为 $9 \times 4^k - 9 \times 2^k + 1$ 或 $4^k - 3 \times 2^k + 1$ 的形式。

如此改进之后，希尔排序算法在最坏情况下的时间复杂度为 $O(n^{4/3})$ ，平均复杂度为 $O(n^{7/6})$ 。更重要的是，这一增量序列在通常应用环境中的综合效率最佳。

习题

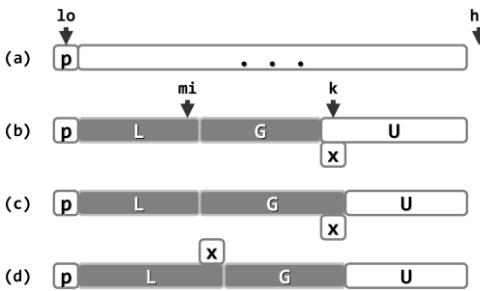
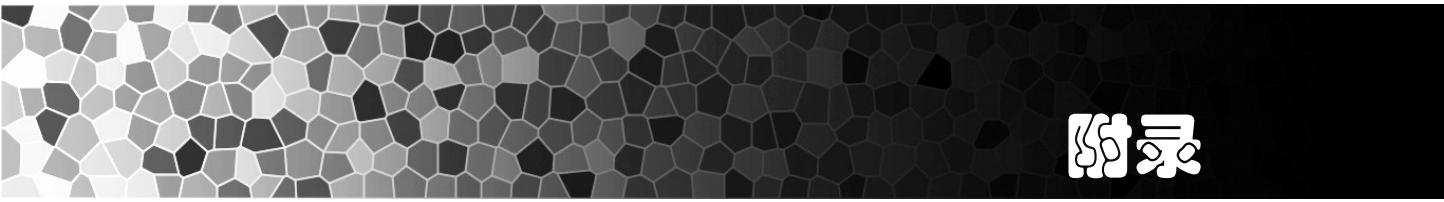


图12.16 轴点构造算法(版本C)

- [1] 图 12.16 给出了轴点构造的另一思路：将整个向量分为三个区间 $L = V[lo, mi]$ 、 $G = V[mi, k]$ 和 $U = V[k, hi]$ ，其中 L/G 中元素均不大/不小于轴点， U 中元素大小未知。初始时 $k = mi = lo$ ；此后随着 k 的不断递增，逐一检查各 $V[k]$ ，并根据 $V[k]$ 相对于候选轴点的大小，相应地扩展区间 L 或 G ，压缩区间 U 。最终，当 $k = hi$ 时， U 不含任何元素，将候选轴点放至 $V[mi]$ 即成为轴点。
- 试依此思路，实现对应的划分算法 `Vector::partition()`；
 - 基于该算法的快速排序是否稳定？
 - 基于该算法的快速排序，能否处理大量元素重复之类的退化情况？
- [2] 考查如代码 12.6 所示 `majEleCandidate()` 算法的返回值 `maj`。
- 该候选者尽管不见得必然是主流数，但是否一定是原向量中出现最频繁者？为什么？
 - 该返回值在向量中出现的次数最少可能是多少？试举一长度不小于 12 的向量实例。
- [3] 按照 12.2.2 节的定义，主流数应严格地多于其它元素。若将“多于”改为“不少于”，则
- 该节所设计的算法框架是否依然可以沿用？或者，需如何调整？
 - 如代码 12.6 所示的 `majEleCandidate()` 算法申报可以沿用？或者，需如何调整？
- [4] 微软 Office 套件中，Excel 提供 `large(range, rank)`、`median(range)` 和 `mode(range)` 等函数。
- 试查阅手册了解其功能；
 - 这些功能分别对应于本章所讨论的哪些问题？
- [5] 实际上，如代码 12.7 所示的 `trivialMedian()` 算法只需迭代至第 $(n_1 + n_2)/2$ 次时即可终止。
- 照此思路改进该算法；
 - 如此改进之后，算法的渐进复杂度是否降低？
- [6] 如代码 12.8 所示的 `median()` 算法属于尾递归 (tail recursion) 形式，试将其改写为迭代形式。
- [7] 如代码 12.9 所示的 `median()` 算法中，针对子向量长度相差悬殊的情况，做了针对性的优化处理。
- 分析该方法的原理并证明其正确性；
 - 试证明，复杂度的精确上界应为 $\mathcal{O}(\log \min(n_1, n_2))$ 。
- [8] 若输入的有序序列 S_1 和 S_2 以列表 (而非向量) 的方式实现，则
- 如代码 12.8 和代码 12.9 所示的 `median()` 算法应做哪些调整？
 - 调整之后的计算效率如何？
- [9] 若输入的有序序列 S_1 和 S_2 以平衡二叉搜索树 (而非序列) 的方式实现，则
- 如代码 12.8 和代码 12.9 所示的 `median()` 算法应做哪些调整？
 - 调整之后的计算效率如何？
- [10] a) 在如代码 12.9 所示 `median()` 算法的基础上添加整型输入参数 k ，在 $S_1 \cup S_2$ 中选取第 k 个元素；
b) 新算法的时间复杂度是多少？
- [11] 考查如代码 12.10 所示的 `quickSelect()` 算法。
- 试举例说明，最坏情况下该算法的外循环需要执行 $\Omega(n)$ 次；
 - 在各元素独立等概率分布的条件下，该算法的平均时间复杂度是多少？
- [12] 试证明，采用 M_{pratt} 序列，希尔排序算法的时间复杂度为 $\mathcal{O}(n \log^2 n)$ 。

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库



附录

插图索引

图 1.1 古埃及人使用的绳索计算机及其算法	2
图 1.2 古希腊人的尺规计算机	3
图 1.3 通过 6 趟扫描交换对七个整数排序（其中已就位的元素以深色示意）	4
图 1.4 大 σ 记号、大 Ω 记号和大 Θ 记号	11
图 1.5 复杂度的典型层次：(1)~(7)依次为 $\sigma(\log n)$ 、 $\sigma(\sqrt{n})$ 、 $\sigma(n)$ 、 $\sigma(n \log n)$ 、 $\sigma(n^2)$ 、 $\sigma(n^3)$ 和 $\sigma(2^n)$	15
图 1.6 对 sum(A, 5) 的递归跟踪分析	17
图 1.7 对 sum(A, 0, 7) 的递归跟踪分析	22
图 2.1 可扩充向量的溢出处理	36
图 2.2 向量元素插入操作 insert(r, e) 的过程	41
图 2.3 向量区间删除操作 remove(lo, hi) 的过程	42
图 2.4 无序向量 deduplicate() 算法原理	44
图 2.5 低效版 uniquify() 算法的最坏情况	46
图 2.6 有序向量中的重复元素可批量删除	47
图 2.7 在有序向量中查找互异的相邻元素	47
图 2.8 基于减治策略的有序向量二分查找算法（版本 A）	48
图 2.9 二分查找算法（版本 A）实例：search(8, 0, 7)	49
图 2.10 二分查找算法（版本 A）的查找长度（成功、失败查找分别以实线、虚线白色方框示意）	50
图 2.11 Fibonacci 查找算法原理	51
图 2.12 Fibonacci 查找算法的查找长度（成功、失败查找分别以实线、虚线白色方框示意）	52
图 2.13 基于减治策略的有序向量二分查找算法（版本 B）	53
图 2.14 基于减治策略的有序向量二分查找算法（版本 C）	55
图 2.15 从三只苹果中挑出重量不同者	57
图 2.16 有序向量的二路归并实例（来自两个向量的元素分别以黑、白方框区分，其各自的当前首元素则以灰色长方形示意）	60
图 2.17 归并排序实例	61
图 3.1 首（末）节点是头（尾）节点的直接后继（前驱）	73
图 3.2 刚创建的 List 对象	73
图 3.3 ListNode::insertAsPred() 算法	75
图 3.4 List::remove() 算法	77
图 3.5 序列的插入排序	81
图 3.6 序列的选择排序	82
图 4.1 一摞椅子即是一个栈	89
图 4.2 栈操作	89
图 4.3 函数调用栈实例：主函数 main() 调用 funcA()，funcA() 调用 funcB()，funcB() 再自我调用	90
图 4.4 facR(7) 的递归调用栈（为简洁起见，这里省略了各帧中记录程序返回地址的指针）	92
图 4.5 进制转换算法流程	93
图 4.6 栈混洗实例：从 {1, 2, 3, 4} 到 {3, 2, 4, 1}（上方左侧为栈 A，右侧为栈 B；下方为栈 S）	94

图 4.7 迭代式括号匹配算法实例（上方为输入表达式；下方为辅助栈的演变过程；虚框表示在（右）括号与栈顶（左）括号匹配时对应的出栈操作）	96
图 4.8 通过剪枝排除候选解子集	104
图 4.9 (a)皇后的控制范围，(b)8 皇后问题的一个解	105
图 4.10 四皇后问题求解过程	106
图 4.11 迷宫寻径算法实例	109
图 4.12 在球桶中顺序排列的一组羽毛球可视作一个队列	110
图 4.13 队列操作	110
图 5.1 有根树的逻辑结构	117
图 5.2 二叉树	117
图 5.3 多叉树的“父节点”表示法	118
图 5.4 多叉树的“孩子节点”表示法	118
图 5.5 多叉树的“父节点+孩子节点”表示法	118
图 5.6 多叉树的“长子+兄弟”表示法（长子和兄弟指针分别以垂直的实线和水平的虚线示意）	119
图 5.7 完整的通讯过程由预处理、编码和解码阶段组成	119
图 5.8 二叉树中每个节点都由根通路串唯一确定	121
图 5.9 $\Sigma = \{'A', 'E', 'G', 'M', 'S'\}$ 两种编码方案对应的二叉编码树	121
图 5.10 BinNode 模板类的逻辑结构	123
图 5.11 二叉树节点左孩子插入过程：(a)插入前；(b)插入后	125
图 5.12 二叉树右节点插入过程：(a)插入前；(b)插入后	127
图 5.13 二叉树节点右子树接入过程：(a)接入前；(b)接入后	128
图 5.14 为实现 PFC 编码和解码过程所需的数据结构和算法	129
图 5.15 子集的 PFC 编码树合并后，即是全集的一棵 PFC 编码树	129
图 5.16 最优编码树的双子性	133
图 5.17 最优编码树的层次性	133
图 5.18 通过节点交换提高编码效率	134
图 5.19 完全二叉树	134
图 5.20 满二叉树	134
图 5.21 考虑字符出现频率，以平均带权深度衡量编码效率	135
图 5.22 若考虑出现频率，完全二叉树或满树未必最优	136
图 5.23 若考虑出现频率，最优编码树往往不是完全二叉树	136
图 5.24 最优编码树的层次性	137
图 5.25 最优编码树中底层兄弟节点合并后，依然是最优编码树	137
图 5.26 Huffman 树构造算法实例	138
图 5.27 二叉树遍历的全局次序由局部次序规则确定	142
图 5.28 二叉树先序遍历序列：{i, d, c, a, b, h, f, e, g, l, k, j, n, m, p, o}	143
图 5.29 二叉树的后序遍历序列：{b, a, c, e, g, f, h, d, j, k, m, o, p, n, l, i}	143
图 5.30 二叉树的中序遍历序列：{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p}	144
图 5.31 迭代式先序遍历实例（出栈节点以深色示意）	145

图 5.32 先序遍历过程：先沿左侧通路自顶而下访问沿途节点，再自底而上依次遍历这些节点的右子树	145
图 5.33 中序遍历过程：顺着左侧通路，自底而上依次访问沿途各节点及其右子树	146
图 5.34 迭代式中序遍历实例（出栈节点以深色示意）	147
图 5.35 中序遍历过程中，在无右孩子的节点处需做回溯	149
图 5.36 后序遍历过程也可划分为模式雷同的若干段	150
图 5.37 迭代式后序遍历实例（出栈节点以深色示意，发生 gotoHLVFL() 调用的节点以大写字母示意）	151
图 5.38 二叉树的层次遍历序列：{i,d,l,c,h,k,n,a,f,j,m,p,b,e,g,o}	151
图 5.39 层次遍历实例（出队节点以深色示意）	152
图 6.1 (a)无向图、(b)混合图和(c)有向图	157
图 6.2 通路与简单通路	158
图 6.3 环路与简单环路	158
图 6.4 欧拉环路与哈密尔顿环路	158
图 6.5 邻接矩阵（空白单元的值均取作矩阵最左上角的数值，比如 0 或 ∞ ）	161
图 6.6 以邻接表方式描述和实现图	164
图 6.7 广度优先搜索示例	167
图 6.8 深度优先搜索实例	170
图 6.9 活跃期与祖先-后代关系之间的对应关系	170
图 6.10 拓扑排序	171
图 6.11 利用“DAG 必有零入度顶点”的特性，实现拓扑排序	172
图 6.12 基于 DFS 搜索的拓扑排序实例	174
图 6.13 关节点	174
图 6.14 双连通域	174
图 6.15 DFS 树根节点是关节点，当且仅当它拥有多个分支	175
图 6.16 内部节点 C 是关节点，当且仅当 C 的某棵极大真子树不（经后向边）联接到 C 的真祖先	175
图 6.17 基于 DFS 搜索的双连通域分解实例（细边框白色、双边框白色和黑色分别示意处于 UNDISCOVERED、DISCOVERED 和 VISITED 状态的顶点，粗边框白色示意当前顶点；dTime 和 fTime (hca) 标签分别标注于各顶点的左、右）	178
图 6.18 支撑树	180
图 6.19 极小支撑树与最小支撑树	180
图 6.20 最小支撑树总是采用最短跨越边联接每一割	181
图 6.21 Prim 算法示例（阴影区域示意不断扩展的子树 T_k ，粗线示意树边）	182
图 6.22 有向带权图及对应的最短路径树	184
图 6.23 最短路径的任一前缀也是最短路径	184
图 6.24 最短路径子树序列	185
图 6.25 Dijkstra 算法示例（阴影区域示意不断扩展的子树 T_k ，粗线示意树边）	186
图 7.1 第 7 章、第 8 章内容纵览	190
图 7.2 二叉搜索树即处处满足顺序性的二叉树	192
图 7.3 二叉搜索树的实例（左三）与反例（右三）	192
图 7.4 二叉搜索树 T（左）的中序遍历序列 S(T)（右）必按非降序排列	192

图 7.5 二叉搜索树的查找过程（查找所经过通路以粗线条示意）	193
图 7.6 二叉搜索树节点插入算法实例	195
图 7.7 二叉搜索树节点删除算法实例	196
图 7.8 由三个关键码{1, 2, 3}的 6 种全排列生成的二叉搜索树	198
图 7.9 由同一组共 11 个节点组成，相互等价的两棵二叉搜索树（阴影部分为二者在拓扑上的差异）	199
图 7.10 zig(v)：顺时针旋转操作	200
图 7.11 zag(v)：逆时针旋转操作	200
图 7.12 在高度固定为 h 的前提下，节点最少的 AVL 树	201
图 7.13 经节点删除和插入操作后，AVL 树可能失衡（加减号示意平衡因子，双圈表示失衡节点）	202
图 7.14 节点插入后通过单旋操作使 AVL 树重新平衡	203
图 7.15 节点插入后通过连续的两次旋转操作使 AVL 树重新平衡	203
图 7.16 节点删除后经一次旋转恢复局部平衡	205
图 7.17 节点删除后通过两次旋转恢复局部平衡	205
图 7.18 节点插入后的统一重新平衡	206
图 8.1 通过自下而上的一系列等价变换，可使任一节点上升至树根	211
图 8.2 简易伸展树的最坏情况	211
图 8.3 通过 zig-zig 操作，将节点 v 上推两层	212
图 8.4 通过 zig-zag 操作，将节点 v 上推两层	212
图 8.5 通过 zig 操作，将节点 v 上推一层（成为树根）	213
图 8.6 双层调整策略的高度折半效果	213
图 8.7 伸展树中较深的节点一旦被访问到，对应分支的长度将随即减半	214
图 8.8 伸展树的节点插入	218
图 8.9 伸展树的节点删除	219
图 8.10 二叉搜索树与四路搜索树	221
图 8.11 B-树的宏观结构（外部节点以深色示意，深度完全一致，且都同处于最底层）	222
图 8.12 (a) 4 阶 B-树；(b) B-树的紧凑表示；(c) B-树的最紧凑表示	222
图 8.13 B-树的查找过程	224
图 8.14 通过分裂修复上溢节点	227
图 8.15 3 阶 B-树插入操作实例	229
图 8.16 下溢节点向父亲“借”一个关键码，父亲再向左兄弟“借”一个关键码	230
图 8.17 下溢节点向父亲“借”一个关键码，父亲再向右兄弟“借”一个关键码	230
图 8.18 下溢节点向父亲“借”一个关键码，然后与左兄弟“粘接”成一个节点	231
图 8.19 3 阶 B-树删除操作实例	234
图 8.20 将二叉树扩展为真二叉树	234
图 8.21 红黑树到 4 阶 B-树的等价转换（在彩色版尚未出版之前本书约定，分别以圆形、正方形、八角形表示红黑树的黑节点、红节点和颜色未定节点，以长方形表示 B-树节点）	235
图 8.22 红黑树的黑高度不低于高度的一半；反之，高度不超过黑高度的两倍	236
图 8.23 双红修正第一种情况（RR-1）及其调整方法	237
图 8.24 双红修正第二种情况（RR-2）及其调整方法	238

图 8.25 双红修正流程图	238
图 8.26 删除节点之后，红黑树条件(4)可能依然满足（图(a)），或经重染色后重新满足（图(b)），也可能不再满足（图(c)）。	240
图 8.27 双黑修正（情况 BB-1）	241
图 8.28 双黑修正（情况 BB-2-R）	241
图 8.29 双黑修正（情况 BB-2-B）	242
图 8.30 双黑修正（情况 BB-3）	242
图 8.31 双黑修正流程图	243
图 8.32 一维范围查询	244
图 8.33 通过预处理排序，高效地解决一维范围查询	245
图 8.34 平面范围查询（planar range query）	245
图 8.35 将待查询的一维点集预处理为一棵平衡二叉搜索树	246
图 8.36 借助平衡二叉搜索树解决一维范围查询问题	246
图 8.37 2d-树中每一节点，均对应于平面上某一左（下）开右（上）闭的矩形	247
图 8.38 2d-树的构造过程，就是对平面递归划分的过程	248
图 8.39 基于 2d-树的平面范围查询实例	249
图 9.1 三国人物的词典结构	253
图 9.2 跳转表的总体逻辑结构	255
图 9.3 跳转表节点插入过程(a~d)，也是节点删除的逆过程(d~a)	262
图 9.4 四联表节点插入过程	262
图 9.5 直接使用线性数组实现电话簿词典	265
图 9.6 散列函数	266
图 9.7 除余法	267
图 9.8 素数表长可降低冲突的概率并提高空间的利用率	268
图 9.9 MAD 法可消除散列过程的连续性	268
图 9.10 通过槽位细分排解散列冲突	272
图 9.11 利用建立独立链排解散列冲突	272
图 9.12 利用公共溢出区解决散列冲突	273
图 9.13 线性试探法对应的查找链	274
图 9.14 通过设置懒惰删除标记，无需大量词条的重排即可保证查找链的完整	274
图 9.15 线性试探法会加剧聚集现象，而平方试探法则会快速跳离聚集区段	278
图 9.16 平方试探法	279
图 9.17 即便散列表长取为素数（M = 11），在装填因子 $\lambda > 50\%$ 时仍可能找不到实际存在的空桶	279
图 9.18 分两步将任意类型的关键码映射为散列地址	280
图 9.19 利用散列表对一组互异整数排序	282
图 9.20 利用散列表对一组可能重复的整数排序	282
图 9.21 利用散列法，在线性时间内确定 n 个共线点之间的最大间隙	283
图 10.1 以获奖先后为优先级，由前 12 届图灵奖得主构成的一个堆结构	292
图 10.2 按照层次遍历序列对完全二叉树节点的编号（其中圆形表示内部节点，方形表示外部节点）	293

图 10.3 完全二叉堆词条插入过程	295
图 10.4 完全二叉堆词条插入操作实例	296
图 10.5 完全二叉堆词条删除过程	297
图 10.6 完全二叉堆词条删除操作实例	297
图 10.7 堆合并算法原理	299
图 10.8 Floyd 算法实例 (虚线示意下滤过程中的交换操作)	300
图 10.9 就地堆排序	301
图 10.10 就地堆排序实例：建堆	301
图 10.11 就地堆排序实例：迭代	302
图 10.12 堆合并	303
图 10.13 整体结构向左倾斜，右侧节点不超过 $\mathcal{O}(\log n)$ 个	303
图 10.14 空节点路径长度 (其中双圈节点违反左倾性)	304
图 10.15 左式堆实例	305
图 10.16 左式堆的右侧链	305
图 10.17 左式堆合并算法原理	305
图 10.18 左式堆合并算法实例	306
图 10.19 基于堆合并操作实现删除接口	307
图 10.20 基于堆合并操作实现词条插入算法	308
图 11.1 串模式匹配的蛮力算法	315
图 11.2 蛮力算法的最坏情况 (也是基于坏字符策略 BM 算法的最好情况)	316
图 11.3 蛮力算法的最好情况 (也是基于坏字符策略 BM 算法的最坏情况)	316
图 11.4 利用以往的成功比对所提供的信息，可以避免主串字符指针的回退	317
图 11.5 利用以往的成功比对所提供的信息，有可能使模式串大跨度地右移	317
图 11.6 利用此前成功比对所提供的信息，在安全的前提下尽可能大跨度地右移模式串	318
图 11.7 $P[j] = P[\text{next}[j]]$ 时，必有 $\text{next}[j + 1] = \text{next}[j] + 1$	319
图 11.8 $P[j] \neq P[\text{next}[j]]$ 时，必有 $\text{next}[j + 1] = \text{next}[\dots\text{next}[j]\dots] + 1$	320
图 11.9 按照此前定义的 next 表，仍有可能进行多次本不必要的字符比对操作	321
图 11.10 坏字符策略：通过右移模式串 P ，使 $T[i+j]$ 重新得到匹配	324
图 11.11 好后缀策略：通过右移模式串 P ，使与 P 后缀 U 匹配的 W 重新得到匹配	327
图 11.12 典型串匹配算法的复杂度概览	330
图 11.13 随着单次比对成功概率 (横轴) 的提高，串匹配算法的运行时间 (纵轴) 通常亦将增加 (此处只是大致示意，实际的增长趋势未必是线性的)	331
图 11.14 Karp-Rabin 串匹配算法实例：模式串指纹 $\text{hash}("82818") = 77$	333
图 11.15 Karp-Rabin 串匹配算法实例：模式串指纹 $\text{hash}("18284") = 48$	334
图 11.16 相邻子串内容及指纹的相关性	334
图 12.1 序列的轴点 (这里用高度来表示各元素的大小)	338
图 12.2 有序向量经循环左移一个单元后，将不含任何轴点	339
图 12.3 轴点构造算法的构思	340
图 12.4 轴点构造过程	341

图 12.5 partition() 算法的退化情况也是最坏情况	343
图 12.6 选取与中位数	344
图 12.7 通过减治策略计算主流数	346
图 12.8 采用减治策略，计算等长有序向量归并后的中位数	347
图 12.9 基于堆结构的选取算法	350
图 12.10 基于快速划分算法逐步逼近选取目标元素	351
图 12.11 k-选取目标元素所处位置的三种可能情况	352
图 12.12 各子序列的中位数以及全局中位数	353
图 12.13 将待排序序列视作二维矩阵	353
图 12.14 递减增量、逐渐逼近策略	354
图 12.15 经多次迭代，逆序元素可能的范围必然不断缩小	356
图 12.16 轴点构造算法（版本 C）	358

表格索引

表 1.1 countOnes(441)的执行过程	12
表 2.1 向量 ADT 支持的操作接口	31
表 2.2 向量操作实例.....	32
表 3.1 列表节点 ADT 支持的操作接口	69
表 3.2 列表 ADT 支持的操作接口	70
表 3.3 插入排序算法实例	81
表 3.4 选择排序算法实例	82
表 4.1 栈 ADT 支持的操作接口	89
表 4.2 栈操作实例	89
表 4.3 表达式求值算法实例	100
表 4.4 RPN 表达式求值算法实例（当前字符以方框注明，操作数栈的底部靠左。）	103
表 4.5 队列 ADT 支持的操作接口	110
表 4.6 队列操作实例.....	110
表 5.1 $\Sigma = \{'A', 'E', 'G', 'M', 'S'\}$ 的一份二进制编码表	120
表 5.2 二进制解码过程	120
表 5.3 $\Sigma = \{'A', 'E', 'G', 'M', 'S'\}$ 的另一份二进制编码表	120
表 5.4 按照表 5.3 “确定”的编码协议，可能有多种解码结果	120
表 5.5 在一篇典型的英文文章中，各字母出现的次数	135
表 5.6 由 6 个字符构成的字符集 Σ ，以及各字符的出现频率	138
表 6.1 图 ADT 支持的边操作接口	159
表 6.2 图 ADT 支持的顶点操作接口	159
表 8.1 双红修正算法所涉及局部操作的统计	238
表 8.2 双黑修正算法所涉及局部操作的统计	243
表 9.1 词典 ADT 支持的标准操作接口	252
表 9.2 词典结构操作实例	253
表 9.3 基数排序实例	284
表 10.1 优先级队列 ADT 支持的操作接口	289
表 10.2 优先级队列操作实例：选择排序	290
表 11.1 串 ADT 支持的操作看接口	313
表 11.2 串操作实例	313
表 11.3 next 表实例：假想地附加一个通配符 P[-1]	319
表 11.4 next 表仍有待优化的实例	321
表 11.5 改进后的 next 表实例	322
表 11.6 模式串 P = "DATA STRUCTURES"及其对应的 BC 表	325
表 11.7 模式串 P = "ICED RICE PRICE"及其对应的 GS 表	328

算法索引

算法 1.1 过直线上给定点作直角	3
算法 1.2 三等分给定线段	3
算法 1.3 取非极端元素	11
算法 2.1 从三个苹果中选出重量不同者	56
算法 4.1 RPN 表达式求值	102
算法 4.2 利用队列结构实现的循环分配器	111
算法 8.1 构造 2d-树	247
算法 8.2 基于 2d-树的平面范围查询	249
算法 12.1 线性时间的 k-选取	352
算法 12.2 希尔排序	354

代码索引

代码 1.1 整数数组的起泡排序	5
代码 1.2 整数二进制展开中位数 1 总数的统计	12
代码 1.3 数组元素求和算法 sumI()	13
代码 1.4 幂函数算法 (蛮力迭代版)	14
代码 1.5 数组求和算法 (线性递归版)	16
代码 1.6 数组倒置算法的统一入口	19
代码 1.7 数组倒置的递归算法	19
代码 1.8 优化的幂函数算法 (线性递归版)	20
代码 1.9 由递归版改造而得的数组倒置算法 (迭代版)	21
代码 1.10 进一步调整代码 1.9 的结构, 消除 goto 语句	21
代码 1.11 通过二分递归计算数组元素之和	22
代码 1.12 通过二分递归计算 Fibonacci 数	23
代码 1.13 通过线性递归计算 Fibonacci 数	24
代码 1.14 基于动态规划策略计算 Fibonacci 数	25
代码 2.1 向量模板类 Vector	33
代码 2.2 基于复制的向量构造器	34
代码 2.3 重载向量赋值操作符	35
代码 2.4 向量内部数组动态扩容算法 expand()	36
代码 2.5 向量内部功能 shrink()	38
代码 2.6 重载向量操作符 []	39
代码 2.7 向量整体置乱算法 permute()	39
代码 2.8 向量区间置乱接口 unsort()	39
代码 2.9 重载比较器以便比较对象指针	40
代码 2.10 无序向量元素查找接口 find()	41
代码 2.11 向量元素插入接口 insert()	41
代码 2.12 向量区间删除接口 remove(lo, hi)	42
代码 2.13 向量单元素删除接口 remove()	43
代码 2.14 无序向量清除重复元素接口 deduplicate()	43
代码 2.15 向量遍历接口 traverse()	44
代码 2.16 基于遍历实现 increase() 功能	45
代码 2.17 有序向量甄别算法 disordered()	46
代码 2.18 有序向量 uniquify() 接口的平凡实现	46
代码 2.19 有序向量 uniquify() 接口的高效实现	47
代码 2.20 有序向量各种 search() 算法的统一接口	48
代码 2.21 二分查找算法 (版本 A)	49
代码 2.22 Fibonacci 查找算法	52

代码 2.23 二分查找算法（版本 B）	54
代码 2.24 二分查找算法（版本 C）	55
代码 2.25 向量排序器接口	59
代码 2.26 向量的起泡排序	59
代码 2.27 单趟扫描交换	59
代码 2.28 向量的归并排序	61
代码 2.29 有序向量的二路归并	62
代码 3.1 列表节点模板类	70
代码 3.2 列表模板类	72
代码 3.3 列表类内部方法 init()	73
代码 3.4 重载列表类的下标操作符	74
代码 3.5 无序列表元素查找接口 find()	74
代码 3.6 列表节点插入接口	75
代码 3.7 ListNode::insertAsPred() 算法	75
代码 3.8 ListNode::insertAsSucc() 算法	76
代码 3.9 列表类内部方法 copyNodes()	76
代码 3.10 基于复制的列表构造方法	76
代码 3.11 列表节点删除接口 remove()	77
代码 3.12 列表析构方法	78
代码 3.13 列表清空方法 clear()	78
代码 3.14 无序列表剔除重复节点接口 deduplicate()	78
代码 3.15 列表遍历接口 traverse()	79
代码 3.16 有序列表剔除重复节点接口 uniquify()	79
代码 3.17 有序列表查找接口 search()	80
代码 3.18 有序列表基于排序的构造方法	80
代码 3.19 列表的插入排序	81
代码 3.20 列表的选择排序	83
代码 3.21 列表最大节点的定位	83
代码 3.22 有序列表的二路归并	84
代码 3.23 列表的归并排序	85
代码 4.1 Stack 模板类	90
代码 4.2 阶乘函数算法（递归版）	92
代码 4.3 阶乘函数算法（迭代版）	92
代码 4.4 进制转换算法（递归版）	93
代码 4.5 进制转换算法（迭代版）	94
代码 4.6 括号匹配算法（递归版）	95
代码 4.7 括号匹配算法（迭代版）	96
代码 4.8 运算符优先级关系的定义	97
代码 4.9 运算符优先级关系的判定	98

代码 4.10 表达式的求值及 RPN 转换	99
代码 4.11 操作数的解析	99
代码 4.12 皇后类	105
代码 4.13 N 皇后算法	106
代码 4.14 迷宫格点类	107
代码 4.15 查询相邻格点	107
代码 4.16 转入相邻格点	108
代码 4.17 迷宫寻径	108
代码 4.18 Queue 模板类	111
代码 4.19 顾客对象	112
代码 4.20 银行服务模拟	112
代码 4.21 查找最短队列	113
代码 5.1 二叉树节点模板类 BinNode	123
代码 5.2 以宏的形式对基于 BinNode 的操作做一归纳整理	124
代码 5.3 二叉树节点左、右孩子的插入	125
代码 5.4 二叉树中序遍历算法的统一入口	125
代码 5.5 二叉树模板类 BinTree	126
代码 5.6 二叉树节点的高度更新	127
代码 5.7 二叉树根、左、右节点的插入	127
代码 5.8 二叉树子树的接入	128
代码 5.9 二叉树子树的删除	129
代码 5.10 二叉树子树的分离	129
代码 5.11 基于二叉树的 PFC 编码	130
代码 5.12 实现 PFC 编码所需的数据结构	130
代码 5.13 初始化 PFC 森林	131
代码 5.14 构造 PFC 编码树	131
代码 5.15 生成 PFC 编码表	131
代码 5.16 PFC 编码	132
代码 5.17 PFC 解码	132
代码 5.18 基于二叉树的 Huffman 编码	139
代码 5.19 HuffChar 结构	139
代码 5.20 Huffman 编码树结构	139
代码 5.21 Huffman 森林结构	139
代码 5.22 Huffman 二进制编码串	140
代码 5.23 Huffman 编码表	140
代码 5.24 Huffman 算法：字符出现频率的样本统计	140
代码 5.25 初始化 Huffman 森林	140
代码 5.26 构造 Huffman 编码树	141
代码 5.27 生成 Huffman 编码表	141

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

附录

代码索引

代码 5.28 Huffman 编码	142
代码 5.29 Huffman 解码	142
代码 5.30 二叉树先序遍历算法（递归版）	143
代码 5.31 二叉树后序遍历算法（递归版）	143
代码 5.32 二叉树中序遍历算法（递归版）	144
代码 5.33 二叉树先序遍历算法（迭代版#1）	144
代码 5.34 二叉树先序遍历算法（迭代版#2）	146
代码 5.35 二叉树中序遍历算法（迭代版#1）	147
代码 5.36 二叉树中序遍历算法（迭代版#2）	147
代码 5.37 二叉树节点直接后继的定位	148
代码 5.38 二叉树中序遍历算法（迭代版#3）	149
代码 5.39 二叉树后序遍历算法（迭代版）	150
代码 5.40 二叉树层次遍历算法	152
代码 6.1 图 ADT 操作接口	160
代码 6.2 基于邻接矩阵实现的图结构	163
代码 6.3 BFS 算法	166
代码 6.4 DFS 算法	168
代码 6.5 基于 DFS 搜索框架实现拓扑排序算法	173
代码 6.6 基于 DFS 搜索框架实现双连通域分解算法	176
代码 6.7 优先级搜索算法框架	179
代码 6.8 Prim 算法的顶点优先级更新器	183
代码 6.9 Dijkstra 算法的顶点优先级更新器	186
代码 7.1 词条模板类 Entry	191
代码 7.2 由 BinTree 派生的二叉搜索树模板类 BST	193
代码 7.3 二叉搜索树 searchIn() 算法	194
代码 7.4 二叉搜索树 search() 接口	194
代码 7.5 二叉搜索树 insert() 接口	195
代码 7.6 二叉搜索树 remove() 接口	196
代码 7.7 二叉搜索树 removeAt() 算法	197
代码 7.8 基于 BST 定义的 AVL 树接口	201
代码 7.9 用于简化 AVL 树算法描述的宏	201
代码 7.10 恢复平衡的调整方案，决定于失衡节点的更高孩子、更高孙子节点的方向	203
代码 7.11 AVL 树节点的插入	204
代码 7.12 AVL 树节点的删除	205
代码 7.13 “3+4”重构	206
代码 7.14 AVL 树的统一重平衡	207
代码 8.1 基于 BST 定义的伸展树接口	217
代码 8.2 伸展树节点的调整	218
代码 8.3 伸展树节点的查找	218

代码 8.4 伸展树节点的插入	219
代码 8.5 伸展树节点的删除	220
代码 8.6 B-树节点	223
代码 8.7 B-树	224
代码 8.8 B-树关键码的查找	225
代码 8.9 B-树关键码的插入	226
代码 8.10 B-树节点的上溢处理	228
代码 8.11 B-树关键码的删除	229
代码 8.12 B-树节点的下溢处理	233
代码 8.13 基于 BST 定义的红黑树接口	236
代码 8.14 用以简化红黑树算法描述的宏	236
代码 8.15 红黑树节点的黑高度更新	237
代码 8.16 红黑树 insert() 接口	237
代码 8.17 双红修正 solveDoubleRed()	239
代码 8.18 红黑树 remove() 接口	240
代码 8.19 双黑修正 solveDoubleBlack()	244
代码 9.1 词典结构的操作接口规范	254
代码 9.2 SkipList 模板类	255
代码 9.3 Quadlist 模板类	256
代码 9.4 QuadlistNode 模板类	257
代码 9.5 Quadlist 对象的创建	257
代码 9.6 SkipList::get() 查找	258
代码 9.7 SkipList::skipSearch() 查找	258
代码 9.8 SkipList::put() 插入	261
代码 9.9 Quadlist::insertSuccAbove() 插入	262
代码 9.10 QuadlistNode::insertAsSuccAbove() 插入	262
代码 9.11 SkipList::remove() 删除	263
代码 9.12 Quadlist::remove() 删除	264
代码 9.13 基于散列表实现的映射结构	270
代码 9.14 散列表构造	270
代码 9.15 确定散列表的素数表长	271
代码 9.16 散列表析构	271
代码 9.17 散列表的查找	275
代码 9.18 散列表的查找 probe4Hit()	276
代码 9.19 散列表元素删除 (采用懒惰删除策略)	276
代码 9.20 散列表元素插入	277
代码 9.21 散列表的查找 probe4Free()	277
代码 9.22 散列表的重散列	278
代码 9.23 散列码转换函数 hashCode()	281

作者保留所有版权，未经授权不得转载
谢绝各种网络电子文库

附录

代码索引

代码 10.1 优先级队列标准接口	290
代码 10.2 利用统一的优先级队列接口，实现通用的 Huffman 编码	291
代码 10.3 为简化完全二叉堆算法的描述及实现而定义的宏	294
代码 10.4 完全二叉树接口	294
代码 10.5 完全二叉堆 getMax() 接口	295
代码 10.6 完全二叉堆 insert() 接口的主体框架	295
代码 10.7 完全二叉堆的上滤	296
代码 10.8 完全二叉堆 delMax() 接口的主体框架	297
代码 10.9 完全二叉堆的下滤	298
代码 10.10 Floyd 建堆算法	299
代码 10.11 基于向量的就地堆排序	302
代码 10.12 左式堆 PQ_LeftHeap 模板类定义	304
代码 10.13 左式堆合并接口 merge()	307
代码 10.14 左式堆节点删除接口 delMax()	308
代码 10.15 左式堆节点插入接口 insert()	308
代码 11.1 蛮力串匹配算法（版本一）	315
代码 11.2 蛮力串匹配算法（版本二）	316
代码 11.3 KMP 主算法	319
代码 11.4 next 表的构造	320
代码 11.5 改进的 next 表构造算法	322
代码 11.6 BM 主算法	323
代码 11.7 BC 表的构造	325
代码 11.8 GS 表的构造	330
代码 11.9 Karp-Rabin 算法相关的预定义	332
代码 11.10 Karp-Rabin 算法主体框架	332
代码 11.11 指纹相同时还需逐个字符地比对	334
代码 11.12 串指纹的快速更新	335
代码 11.13 提前计算 Mm-1	335
代码 12.1 向量的快速排序	339
代码 12.2 轴点构造算法（版本 A）	340
代码 12.3 轴点构造算法（版本 B）	344
代码 12.4 主流数查找算法主体框架	345
代码 12.5 候选主流数核对算法	346
代码 12.6 候选主流数选取算法	346
代码 12.7 中位数蛮力查找算法	347
代码 12.8 等长有序向量归并后中位数算法	348
代码 12.9 不等长有序向量归并后中位数算法	349
代码 12.10 基于快速划分的 k-选取算法	351

关键词索引

A

AVL 树 (AVL tree) 200

B

B-树 (B-tree) 222, 234

F

Fibonacci 查找 (Fibonaccian search) 51

H

h-有序 (h-ordered) 356

h-排序 (h-sorting) 356

K

kd-树 (kd-tree) 247

k 叉树 (k-ary tree) 117

k-选取 (k-selection) 345

M

MAD 法 (multiply-add-divide method) 268

二划

二叉树 (binary tree) 117

二叉树节点 (binary tree node) 122

二叉搜索树 (binary search tree) ... 191

二分查找 (binary search) 49

二分递归 (binary recursion) 22

入队 (enqueue) 110

三划

入边 (incoming edge) 157

入度 (in-degree) 157

入栈 (push) 89

八叉树 (octree) 210

四划

不稳定算法 (unstable algorithm) 60

中位点 (median point) 247

中位数 (median) 345

公共溢出区 (overflow area) 273

内部节点 (internal node) 117

分而治之 (divide-and-conquer) 22

分摊分析 (amortized analysis) 37

分摊运行时间 (amortized running time)

..... 37

切割节点 (cut vertex) 174

双红 (double red) 237

双连通域 (bi-connected component) . 174

双黑 (double black) 240

双端队列 (deque) 114

开放定址 (open addressing)	273
支撑树 (spanning tree)	180
无向边 (undirected edge)	156
无向图 (undigraph)	157
无序向量 (unsorted vector)	40
比较树 (comparison tree)	57
父节点 (parent)	117, 118
计算机科学 (computer science)	2
计算科学 (computing science)	2
队头 (front)	109, 114
队列 (queue)	109
队尾 (rear)	109, 114

五 划

主流数 (majority)	345
出队 (dequeue)	110
出边 (outgoing edge)	157
出度 (out-degree)	157
出栈 (pop)	89
半线性结构 (semi-linear structure)	116
可计算性 (computability)	7
可扩充向量 (extendable vector)	36
可达分量 (reachable component)	166
右侧链 (rightmost chain)	305
叶节点 (leaf)	117
四叉树 (quadtree)	210
四联表 (quadlist)	256
外部节点 (external node)	117, 193, 222
头节点 (header)	72
头顶点 (head)	157
对外功能接口 (interface)	25
对数多项式时间复杂度算法 (polylogarithmic-time algorithm) .	13
对数时间复杂度算法 (logarithmic-time algorithm)	13
左式堆 (leftist heap)	303
左侧通路 (left branch)	145
平凡子串 (trivial substring)	312

平凡后缀 (trivial suffix)	312
平凡前缀 (trivial prefix)	312
平方取中法 (mid-square)	269
平方试探 (quadratic probing)	279
平均运行时间 (average running time)	37
平均带权深度 (weighted average depth)	135
平均情况 (average case)	10
平面范围查询 (planar range query) .	245
平衡二叉搜索树 (balanced binary search tree, BBST)	199
平衡因子 (balance factor)	200
归并排序 (mergesort)	60, 84
末节点 (last node)	72
正确性 (correctness)	6
节点 (node) ...	69, 117, 118, 156, 193
节点的分裂 (split)	227
节点的合并 (merge)	60, 231
边 (edge)	116, 156

六 划

优先级 (priority)	289
优先级队列 (priority queue)	289
优先级搜索 (Priority-First Search, PFS)	179
优先级数 (priority number)	179
伪对数的 (pseudo-logarithmic)	15
伪线性的 (pseudo-linear)	15
伪随机试探法 (pseudo-random probing)	279
先进先出 (first-in-first-out, FIFO)	110
全序 (full order)	288
关节点 (articulation point)	174
关联 (incident)	157
关联矩阵 (incidence matrix)	186
关联数组 (associative array)	253
关键码 (key)	289
再散列 (double hashing)	280

列表 (list) 30, 68
动态规划 (dynamic programming) 25
合成数 (composite number) 187
后代 (descendant) 117
后向边 (back edge) 169
后继 (successor) 30, 69
后缀 (suffix) 30, 312
后缀表达式 (postfix) 101
向量 (vector) 30, 31
回溯 (backtracking) 104
在线算法 (online algorithm) ... 56, 122
地址空间 (address space) 265
多叉堆 (d-heap) 309
多项式时间复杂度算法 (polynomial-time algorithm) 14
多项式散列码 (polynomial hash code) 281
多路递归 (multi-way recursion) 22
多路搜索树 (multi-way search tree) 221
多槽位法 (multiple slots) 272
好后缀 (good suffix) 326
字典序 (lexicographical order) 64
字符串 (string) 312
字符表 (alphabet) 120, 312
成本 (cost) 180
执行栈 (execution stack) 91
有向无环图 (directed acyclic graph, DAG)
..... 158, 172
有向边 (directed edge) 156
有向图 (digraph) 157
有序二叉树 (ordered binary tree) .. 117
有序列表 (sorted list) 79
有序向量 (sorted vector) 31, 45
有序树 (ordered tree) 119
有穷性 (finiteness) 6
有根树 (rooted tree) 117
权重 (weight) 158
红黑树 (red-black tree) 234, 235
网络 (network) 158
自环 (self-loop) 157

自调整列表 (self-adjusting list) ... 85
闭散列 (closed hashing) 273

七划

串模式匹配 (string pattern matching)
..... 314
伸展 (splaying) 211
伸展树 (splay tree) 210
位异或法 (xor) 269
位置 (position) 30, 68, 123
低位字段优先 (least significant digit first) 284
初始化 (initialization) 34
坏字符 (bad character) 324
完全二叉树 (complete binary tree) . 134
完全二叉堆 (complete binary heap) . 292
完美散列 (perfect hashing) 265
尾节点 (trailer) 72, 260
尾顶点 (tail) 157
尾递归 (tail recursion) 21, 358
层 (level) 255
层次遍历 (level-order traversal) .. 151
希尔排序 (Shellsort) 353
序列 (sequence) 30
快速划分 (quick partitioning) 340
快速排序 (quicksort) 338
折叠法 (folding) 269
时间复杂度 (time complexity) 8
极小支撑树 (minimal spanning tree, MST)
..... 181
词条 (entry) 191, 252, 289
词典 (dictionary) 252
返回地址 (return address) 91
连通分量 (connected component) 166
邻接表 (adjacency list) 164
邻接矩阵 (adjacency matrix) 161

八 划

- 具体实现 (implementation) 25
势能 (potential) 215
势能分析法 (potential analysis) ... 214
咖啡罐游戏 (Coffee Can Game) 27
图 (graph) 156
图搜索 (graph search) 165
底层 (bottom) 255
建堆 (heapification) 298
弧 (arc) 156
抽象数据类型 (abstract data type, ADT)
..... 25
拓扑排序 (topological sorting) 172
构造函数 (constructor) 34
析构函数 (destructor) 35
欧拉环路 (Eulerian tour) 158
环路 (cycle) 158
画家算法 (painter's algorithm) 326
直接后继 (intermediate successor) .. 30
直接前驱 (intermediate predecessor) 30
空节点路径长度 (null path length) 123,
304
空串 (null string) 312
空间复杂度 (space complexity) 11
线性时间复杂度算法 (linear-time
algorithm) 13
线性试探 (linear probing) 273
线性结构 (linear structure) 116
线性递归 (linear recursion) 16
线性数组 (linear array) 30
组合 (combination) 356
终点 (destination) 157
范围查询 (range query) 245
试探 (probing) 104
轮值 (round robin) 111
非线性结构 (non-linear structure) . 116
顶层 (top) 255

九 划

- 顶点 (vertex) 116, 156
前向边 (forward edge) 169
前驱 (predecessor) 30, 69
前线 (frontier) 165
前缀 (prefix) 30, 312
前缀无歧义编码 (prefix-free code) . 121
复杂度下界 (lower bound) 57
孩子节点 (child) 117
客户 (client) 111
带权图 (weighted graph) 158
带权深度 (weighted depth) 135
帧 (frame) 91
度数 (degree) 117, 157
持久性结构 (persistent structure) . 210
指数时间复杂度算法 (exponential-time
algorithm) 14
映射 (map) 252
查找长度 (search length) 50
查找链 (probing chain) 274
标志 (tag) 113
栈 (stack) 88
栈底 (stack bottom) 89
栈顶 (stack top) 89
栈混洗 (stack permutation) 94
树边 (tree edge) 166, 168
活跃函数实例 (active function instance)
..... 91
活跃期 (active duration) 169
独立链 (separate chaining) 272
相邻或邻接 (adjacent) 157
祖先 (ancestor) 117
轴点 (pivot) 339
退化 (degeneracy) 7
逆波兰表达式 (reverse Polish notation,
RPN) 101
选取 (selection) 345

选择排序 (selectionsort) 82, 289
重写 (override) 127, 193, 284
重载 (overload) 19, 39, 105, 139, 281, 291
重散列 (rehashing) 277
除余法 (division method) 267
顺序查找 (sequential search) 40
首节点 (first node) 72

十 划

哨兵节点 (sentinel node) 73
旅行商问题 (traveling salesman problem)
..... 104
根 (root) 117
根通路串 (root path string) 121
桥 (bridge) 181
真二叉树 (proper binary tree) 117, 234
真子串 (proper substring) 312
真后代 (proper descendant) 117
真后缀 (proper suffix) 312
真前缀 (proper prefix) 312
真祖先 (proper ancestor) 117
离线算法 (offline algorithm) 56
秩 (rank) 30, 31
调用栈 (call stack) 91
起泡排序 (bubblesort) 4, 59
起点 (origin) 157
起点或源点 (source) 184
递归调用 (recursive call) 16
递归基 (base case of recursion) 16
递归跟踪 (recursion trace) 17
递减增量 (diminishing increment) .. 354
递推方程 (recurrence equation) 18
通路或路径 (path) 157
难解性 (intractability) 8
高度 (height) 117, 123, 152

十一划

偏序 (partial order) 288
减而治之 (decrease-and-conquer) 17
剪枝 (pruning) 104
基于比较式算法 (comparison-based
algorithm) 57
基数排序 (radixsort) 284
堆 (heap) 292
堆排序 (heapsort) 300
常数时间复杂度算法 (constant-time
algorithm) 12
排队论 (queuing theory) 112
排序 (sorting) 4
斜堆 (skew heap) 309
桶 (bucket) 265
桶排序 (bucketsort) 282
桶数组 (bucket array) 265
深度 (depth) 117
深度优先搜索树 (DFS tree) 169
混合图 (mixed graph) 157
清理 (cleanup) 35
渐进分析 (asymptotic analysis) 9
符号表 (symbol table) 252
野指针 (wild pointer) 36
随机生成 (randomly generated by) .. 198
随机组成 (randomly composed of) ... 198

十二划

塔 (tower) 255
就地算法 (in-place algorithm) . 12, 149
循优先级访问 (call-by-priority) ... 288
循关键码访问 (call-by-key) 191
循位置访问 (call-by-position) 68
循环移位散列码 (cyclic shift hash code)
..... 281
循值访问 (call-by-value) 252

循秩访问 (call-by-rank)	68
循链接访问 (call-by-link)	68
插入 (insert)	224
插入排序 (insertionsort)	81
搜索 (search)	190, 224
散列 (hashing)	265
散列冲突 (collision)	267
散列地址 (hashing address)	265
散列函数 (hash function)	265
散列码 (hash code)	280
散列表 (hashtable)	265
最大的间隙 (maximum gap)	283
最小支撑树 (minimum spanning tree, MST)	181
最好情况 (best case)	10
最低共同祖先 (lowest common ancestor, LCA)	246
最坏情况 (worst case)	10
最坏情况下最优的 (worst-case optimal)	57
最佳优先搜索 (Best-First Search, BFS)	179
最高左侧可见叶节点 (highest leaf visible from left, HLVFL)	150
最高连通祖先 (highest connected ancestor, HCA)	176
期望运行时间 (expected running time)	37
稀疏图 (sparse graph)	164
编码 (encoding)	119
装填因子 (load factor)	35, 266, 277
遍历 (traversal)	142, 156
遍历树 (traversal tree)	165
链表 (linked list)	68
链接 (link)	68
鲁棒性 (robustness)	7
黑高度 (black height)	235
黑深度 (black depth)	235

十三划

意外 (exception)	43
数字分析法 (selecting digits)	269
数组 (array)	30
数据局部性 (data locality)	210
满二叉树 (full binary tree)	135
简单图 (simple graph)	157
简单环路 (simple cycle)	158
简单通路 (simple path)	158
解码 (decoding)	119
跨边 (cross edge)	166, 169
跨越边 (crossing edge)	181
跳转表 (skip list)	254
输入 (input)	5
输出 (output)	5
输出敏感的 (output sensitive)	245
错误 (error)	43

十四划

模式计数 (pattern counting)	314
模式定位 (pattern location)	314
模式枚举 (pattern enumeration)	314
模式检测 (pattern detection)	314
稳定性 (stability)	54, 59, 284
稳定算法 (stable algorithm)	60, 81, 282
聚集 (clustering)	268

十五划

增量 (increment)	354
槽位 (slot)	272

十六划

懒惰删除 (lazy removal)	275
---------------------------	-----

内容简介

本书按照面向对象程序设计的思想，根据作者多年教学积累，系统介绍各类数据结构的功能、表示和实现，对比各类数据结构适用的应用环境；结合实际问题展示算法设计的一般性模式与方法、算法实现的主流技巧，以及算法效率的评判依据和分析方法；以高度概括的体例为线索贯穿全书，并通过对比和类比揭示数据结构与算法的内在联系，帮助读者形成整体性认识。

书中穿插大量验证型、拓展型和反思型习题，以激发读者的求知欲，培养自学能力和独立思考习惯；230 多组 300 余幅插图结合简练的叙述，230 余段代码配合详尽而简洁的注释，使深奥抽象的概念和过程得以具体化并便于理解和记忆。

结合学生基础、专业方向、教学目标及允许课时总量等因素，本书提供了若干种典型教学进度及学时分配方案，供授课教师视具体情况参考和选用。勘误表、插图、代码、部分习题解答以及讲义等相关教学资料，均以电子版形式向公众开放，读者可从本书主页（<http://166.111.138.40/~deng/dsacpp/>）直接下载。