

- Sedgewick之巨著，与高德纳TAOCP一脉相承
- 几十年多次修订，经久不衰的畅销书
- 涵盖所有程序员必须掌握的50种算法



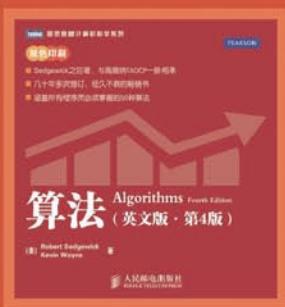
# 算法 (第4版) Algorithms Fourth Edition

[美] Robert Sedgewick 著  
Kevin Wayne

谢路云 译



人民邮电出版社  
POSTS & TELECOM PRESS



## ∞ 延伸阅读 ∞

计算机程序设计艺术

卷1：基本算法（英文版·第3版）

计算机程序设计艺术

卷2：半数值算法（英文版·第3版）

计算机程序设计艺术

卷3：排序与查找（英文版·第2版）

计算机程序设计艺术

卷4A：组合算法（一）（英文版）

计算机体系结构：

量化研究方法（第5版）

具体数学：

计算机科学基础（第2版）

编译器设计（第2版）



# 算法 (第4版) Algorithms Fourth Edition

[美] Robert Sedgewick 著  
Kevin Wayne  
谢路云

人民邮电出版社  
北京



图灵程序设计丛书

# Algorithms, Fourth Edition

# 算法（第4版）

[美] Robert Sedgewick Kevin Wayne 著  
谢路云 译

人民邮电出版社  
北京

## 图书在版编目 (C I P) 数据

算法 : 第4版 / (美) 塞奇威克 (Sedgewick, R.) ,  
(美) 韦恩 (Wayne, K.) 著 ; 谢路云译. -- 北京 : 人民  
邮电出版社, 2012.10

(图灵程序设计丛书)

书名原文: Algorithms, Fourth Edition

ISBN 978-7-115-29380-0

I. ①算… II. ①塞… ②韦… ③谢… III. ①电子计  
算机—算法理论 IV. ①TP301.6

中国版本图书馆CIP数据核字(2012)第220659号

## 内 容 提 要

本书作为算法领域经典的参考书, 全面介绍了关于算法和数据结构的必备知识, 并特别针对排序、搜索、图处理和字符串处理进行了论述。第4版具体给出了每位程序员应知应会的50个算法, 提供了实际代码, 而且这些Java代码实现采用了模块化的编程风格, 读者可以方便地加以改造。配套网站提供了本书内容的摘要及更多的代码实现、测试数据、练习、教学课件等资源。

本书适合用做大学教材或从业者的参考书。

## 图灵程序设计丛书 算法 (第4版)

---

◆ 著 [美] Robert Sedgewick Kevin Wayne

译 谢路云

责任编辑 朱 巍

执行编辑 丁晓昀 刘美英

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号

邮编 100061 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

北京 印刷

◆ 开本: 787×1092 1/16

印张: 40.5

字数: 1115千字 2012年10月第1版

印数: 1~5 000册 2012年10月北京第1次印刷

著作权合同登记号 图字: 01-2011-5236号

---

ISBN 978-7-115-29380-0

定价: 99.00元

读者服务热线: (010)51095186转604 印装质量热线: (010)67129223

反盗版热线: (010)67171154

# 版 权 声 明

Authorized translation from the English language edition, entitled *Algorithms, Fourth Edition*, 978-0-321-57351-3 by Robert Sedgewick, Kevin Wayne, published by Pearson Education, Inc., publishing as Addison Wesley, Copyright © 2011 by Pearson Education, Inc.

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

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. and POSTS & TELECOM PRESS Copyright © 2012.

本书中文简体字版由Pearson Education Asia Ltd.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

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

版权所有，侵权必究。

谨以此书献给Adam、Andrew、Brett、Robbie，并特别感谢Linda。

——Robert Sedgewick

献给Jackie和Alex。

——Kevin Wayne

# 译者序

在计算机领域，算法是一个永恒的主题。即使仅把算法入门方面的书都摆出来，国内外的加起来怕是能铺满整个天安门广场。在这些书中，有几本尤其与众不同，本书就是其中之一。

本书是学生的良师。在翻译的过程中我曾无数次感叹：“要是当年我能拥有这本书那该多好！”应该说本书是为在校学生量身打造的。没有数学基础？没关系，只要你在高中学过了数学归纳法，那么书中95%以上的数学内容你都可以看得懂，更何况书中还辅以大量图例。没学过编程？没关系，第1章会给大家介绍足够多的Java知识，即使你不是计算机专业的学生，也不会遇到困难。整本书的内容编排循序渐进，由易到难，前后呼应，足见作者的良苦用心。没有比本书更专业的算法教科书了。

本书是老师的好帮手。如果老师们还只能照本宣科，只能停留在算法本身一二三四的阶段，那就已经大大落后于这个时代了。算法并不仅仅是计算的方法，探究算法的过程反映出的是我们对这个世界的认知方法：是唯唯诺诺地将课本当做圣经，还是通过“实验—失败—再实验”循环的锤炼？数学是保证，数据是验证。本书通过各种算法，从各个角度，多次说明了这个道理，这也正是第1章是全书内容最多的一章的原因。希望每一位读者都不要错过第1章。无论你有没有编程基础，都会从中得到有益的启示。

本书是程序员的益友。在工作了多年之后，快速排序、霍夫曼编码、KMP等曾经熟悉的概念在你脑中是不是已经凋零成了一个个没有内涵的名词？是时候重新拾起它们了。无论是为手头的工作寻找线索，还是为下一份工作努力准备，这些算法基础知识都是你不能跳过的。本书强调软件工程中的最佳实践，特别适合已有工作经验的程序员朋友。所有的算法都是先有API，再有实现，之后是证明，最后是数据。这种先接口后实现、强调测试的做法，无疑是在工作中摸爬滚打多年的程序员最熟悉的。

本书也有一些遗憾，比如没有介绍动态规划这样重要的思想。但是瑕不掩玉，它仍然是最好的入门级算法书。我强烈地希望能够把本书翻译成中文，但同时也诚惶诚恐，如履薄冰，担心自己的水平不足以准确传达原文的意思。翻译的过程虽然辛苦，但我觉得非常值得。感谢人民邮电出版社图灵公司给了我这个机会，感谢编辑和审稿专家的细心检查。同时感谢我的妻子朱天的全力支持。译者水平有限，bug在所难免，还请读者批评指正。

谢路云  
2012.9.17

# 前　　言

本书力图研究当今最重要的计算机算法并将一些最基础的技能传授给广大求知者。它适合用做计算机科学进阶教材，面向已经熟悉了计算机系统并掌握了基本编程技能的学生。本书也可用于自学，或是作为开发人员的参考手册，因为书中实现了许多实用算法并详尽分析了它们的性能特点和用途。这本书取材广泛，很适合作为该领域的入门教材。

**算法和数据结构的学习**是所有计算机科学教学计划的基础，但它不只是对程序员和计算机系的学生有用。任何计算机使用者都希望计算机能运行得更快一些或是能解决更大规模的问题。本书中的算法代表了近50年来的大量优秀研究成果，是人们工作中必备的知识。从物理中的*N*体模拟问题到分子生物学中的基因序列问题，我们描述的基本方法对科学研究而言已经必不可少；从建筑建模系统到模拟飞行器，这些算法已经成为工程领域极其重要的工具；从数据库系统到互联网搜索引擎，算法已成为现代软件系统中不可或缺的一部分。这仅是几个例子而已，随着计算机应用领域的不断扩张，这些基础方法的影响也会不断扩大。

在开始学习这些基础算法之前，我们先要熟悉全书中都将会用到的栈、队列等低级抽象的数据类型。然后依次研究排序、搜索、图和字符串方面的基础算法。最后一章将会从宏观角度总结全书的内容。

## 独特之处

本书致力于研究有实用价值的算法。书中讲解了多种算法和数据结构，并提供了大量相关的信息，读者应该能有信心在各种计算环境下实现、调试并应用它们。本书的特点涉及以下几个方面。

**算法** 书中均有算法的完整实现，并讨论了程序在多个样例上的运行状况。书中的代码都是可以运行的程序而非伪代码，因此非常便于投入使用。书中程序是用Java语言编写的，但其编程风格方便读者使用其他现代编程语言重用其中的大部分代码来实现相同算法。

**数据类型** 我们在数据抽象上采用了现代编程风格，将数据结构和算法封装在了一起。

**应用** 每一章都会给出所述算法起到关键作用的应用场景。这些场景多种多样，包括物理模拟与分子生物学、计算机与系统工程学，以及我们熟悉的数据压缩和网络搜索等。

**学术性** 我们非常重视使用数学模型来描述算法的性能。我们用模型预测算法的性能，然后在真实的环境中运行程序来验证预测。

**广度** 本书讨论了基本的抽象数据类型、排序算法、搜索算法、图及字符串处理。我们在算法

的讨论中研究数据结构、算法设计范式、归纳法和解题模型。这将涵盖20世纪60年代以来的经典方法以及近年来产生的新方法。

我们的主要目标是将今天最重要的实用算法介绍给尽可能广泛的群体。这些算法一般都十分巧妙奇特，20行左右的代码就足以表达。它们展现出的问题解决能力令人叹为观止。没有它们，创造计算智能、解决科学问题、开发商业软件都是不可能的。

## 本书网站

本书的一个亮点是它的配套网站algs4.cs.princeton.edu。这一网站面向教师、学生和专业人士，免费提供关于算法和数据结构的丰富资料。

**一份在线大纲** 包含了本书内容的结构并提供了链接，浏览起来十分方便。

**全部实现代码** 书中所有的代码均可以在这里找到，且其形式适合用于程序开发。此外，还包括算法的其他实现，例如高级的实现、书中提及的改进的实现、部分习题的答案以及多个应用场景的客户端代码。我们的重点是用真实的应用环境来测试算法。

**习题与答案** 网站还提供了一些附加的选择题（只需要一次单击便可获取答案）、很多算法应用的例子、编程练习和答案以及一些有挑战性的难题。

**动态可视化** 书是死的，但网站是活的，在这里我们充分利用图形类演示了算法的应用效果。

**课程资料** 网站包含和本书及网上内容对应的一整套幻灯片，以及一系列编程作业、核对表、测试数据和备课手册。

**相关资料链接** 网站包含大量的链接，提供算法应用的更多背景知识以及学习算法的其他资源。

我们希望这个站点和本书互为补充。一般来说，建议读者在第一次学习某种算法或是希望获得整体概念时看书，并把网站作为编程时的参考或是在线查找更多信息的起点。

## 作为教材

本书为计算机科学专业进阶的教材，涵盖了这门学科的核心内容，并能让学生充分锻炼编程、定量推理和解决问题等方面的能力。一般来说，此前学过一门计算机方面的先导课程就足矣，只要熟悉一门现代编程语言并熟知现代计算机系统，就都能够阅读本书。

虽然本书使用Java实现算法和数据结构，但其代码风格使得熟悉其他现代编程语言的人也能看懂。我们充分利用了Java的抽象性（包括泛型），但不会依赖这门语言的独门特性。

书中涉及的多数数学知识都有完整的讲解（少数会有延伸阅读），因此阅读本书并不需要准备太多数学知识，不过有一定的数学基础当然更好。应用场景都来自其他学科的基础内容，同样也在书中有完整介绍。

本书涉及的内容是任何准备主修计算机科学、电气工程、运筹学等专业的学生应了解的基础知识，并且对所有对科学、数学或工程学感兴趣的学生也十分有价值。

## 背景介绍

这本书意在接续我们的一本基础教材《Java程序设计：一种跨学科的方法》，那本书对计算机领域做了概括性介绍。这两本书合起来可用做两到三个学期的计算机科学入门课程教材，为所有学生在自然科学、工程学和社会科学中解决计算问题提供必备的基础知识。

本书大部分内容来自Sedgewick的算法系列图书。本质上，本书和该系列的第1版和第2版最接近，但还包含了作者多年教学和学习的经验。Sedgewick的《C算法（第3版）》、《C++算法（第3版）》、《Java算法（第3版）》更适合用做参考书或是高级课程的教材，而本书则是专门为大学一、二年级学生设计的一学期教材，也是最新的基础入门书或从业者的参考书。

## 致谢

本书的编写花了近40年时间，因此想要一一列出所有参与人是不可能的。本书的前几版一共列出了好几十人，其中包括（按字母顺序）Andrew Appel、Trina Avery、Marc Brown、Lyn Dupré、Philippe Flajolet、Tom Freeman、Dave Hanson、Janet Incerpi、Mike Schidlowsky、Steve Summit和Chris Van Wyk。我要感谢他们所有人，尽管其中有些人的贡献要追溯到几十年前。至于第4版，我们要感谢试用了本书样稿的普林斯顿及其他院校的数百名学生，以及通过本书网站发表意见和指出错误的世界各地的读者。

我们还要感谢普林斯顿大学对于高质量教学的坚定支持，这是本书得以面世的基础。

Peter Gordon几乎从本书写作之初就提出了很多有用的建议，这一版奉行的“归本溯源”的指导思想也是他最早提出的。关于第4版，我们要感谢Barbara Wood认真又专业的编辑工作，Julie Nahil对生产过程的管理，以及Pearson出版公司中为本书的付梓和营销辛勤工作的朋友。所有人都在积极地追赶进度，而本书的质量并没有受到丝毫影响。

# 目 录

<b>第1章 基础</b> .....	1
1.1 基础编程模型	4
1.1.1 Java程序的基本结构	4
1.1.2 原始数据类型与表达式	6
1.1.3 语句	8
1.1.4 简便记法	9
1.1.5 数组	10
1.1.6 静态方法	12
1.1.7 API	16
1.1.8 字符串	20
1.1.9 输入输出	21
1.1.10 二分查找	28
1.1.11 展望	30
1.2 数据抽象	38
1.2.1 使用抽象数据类型	38
1.2.2 抽象数据类型举例	45
1.2.3 抽象数据类型的实现	52
1.2.4 更多抽象数据类型的实现	55
1.2.5 数据类型的设计	60
1.3 背包、队列和栈	74
1.3.1 API	74
1.3.2 集合类数据类型的实现	81
1.3.3 链表	89
1.3.4 综述	98
1.4 算法分析	108
1.4.1 科学方法	108
1.4.2 观察	108
1.4.3 数学模型	112
1.4.4 增长数量级的分类	117
1.4.5 设计更快的算法	118
1.4.6 倍率实验	121
1.4.7 注意事项	123
1.4.8 处理对于输入的依赖	124
1.4.9 内存	126
1.4.10 展望	129
1.5 案例研究：union-find算法	136
1.5.1 动态连通性	136
1.5.2 实现	140
1.5.3 展望	148
<b>第2章 排序</b> .....	152
2.1 初级排序算法	153
2.1.1 游戏规则	153
2.1.2 选择排序	155
2.1.3 插入排序	157
2.1.4 排序算法的可视化	159
2.1.5 比较两种排序算法	159
2.1.6 希尔排序	162
2.2 归并排序	170
2.2.1 原地归并的抽象方法	170
2.2.2 自顶向下的归并排序	171
2.2.3 自底向上的归并排序	175
2.2.4 排序算法的复杂度	177
2.3 快速排序	182
2.3.1 基本算法	182
2.3.2 性能特点	185
2.3.3 算法改进	187
2.4 优先队列	195
2.4.1 API	195
2.4.2 初级实现	197
2.4.3 堆的定义	198
2.4.4 堆的算法	199
2.4.5 堆排序	205
2.5 应用	214
2.5.1 将各种数据排序	214
2.5.2 我应该使用哪种排序算法	218
2.5.3 问题的归约	219
2.5.4 排序应用一览	221

<b>第3章 查找</b>	.....	227
3.1 符号表	.....	228
3.1.1 API	.....	228
3.1.2 有序符号表	.....	230
3.1.3 用例举例	.....	233
3.1.4 无序链表中的顺序查找	.....	235
3.1.5 有序数组中的二分查找	.....	238
3.1.6 对二分查找的分析	.....	242
3.1.7 预览	.....	244
3.2 二叉查找树	.....	250
3.2.1 基本实现	.....	250
3.2.2 分析	.....	255
3.2.3 有序性相关的方法与删除操作	.....	257
3.3 平衡查找树	.....	269
3.3.1 2-3查找树	.....	269
3.3.2 红黑二叉查找树	.....	275
3.3.3 实现	.....	280
3.3.4 删除操作	.....	282
3.3.5 红黑树的性质	.....	284
3.4 散列表	.....	293
3.4.1 散列函数	.....	293
3.4.2 基于拉链法的散列表	.....	297
3.4.3 基于线性探测法的散列表	.....	300
3.4.4 调整数组大小	.....	304
3.4.5 内存使用	.....	306
3.5 应用	.....	312
3.5.1 我应该使用符号表的哪种实现	.....	312
3.5.2 集合的API	.....	313
3.5.3 字典类用例	.....	315
3.5.4 索引类用例	.....	318
3.5.5 稀疏向量	.....	322
<b>第4章 图</b>	.....	329
4.1 无向图	.....	331
4.1.1 术语表	.....	331
4.1.2 表示无向图的数据类型	.....	333
4.1.3 深度优先搜索	.....	338
4.1.4 寻找路径	.....	342
4.1.5 广度优先搜索	.....	344
4.1.6 连通分量	.....	349
4.1.7 符号图	.....	352
4.1.8 总结	.....	358
4.2 有向图	.....	364
4.2.1 术语	.....	364
4.2.2 有向图的数据类型	.....	365
4.2.3 有向图中的可达性	.....	367
4.2.4 环和有向无环图	.....	369
4.2.5 有向图中的强连通性	.....	378
4.2.6 总结	.....	385
4.3 最小生成树	.....	390
4.3.1 原理	.....	391
4.3.2 加权无向图的数据类型	.....	393
4.3.3 最小生成树的API和测试用例	.....	396
4.3.4 Prim算法	.....	398
4.3.5 Prim算法的即时实现	.....	401
4.3.6 Kruskal算法	.....	404
4.3.7 展望	.....	407
4.4 最短路径	.....	412
4.4.1 最短路径的性质	.....	413
4.4.2 加权有向图的数据结构	.....	414
4.4.3 最短路径算法的理论基础	.....	420
4.4.4 Dijkstra算法	.....	421
4.4.5 无环加权有向图中的最短路径算法	.....	425
4.4.6 一般加权有向图中的最短路径问题	.....	433
4.4.7 展望	.....	445
<b>第5章 字符串</b>	.....	451
5.1 字符串排序	.....	455
5.1.1 键索引计数法	.....	455
5.1.2 低位优先的字符串排序	.....	458
5.1.3 高位优先的字符串排序	.....	461
5.1.4 三向字符串快速排序	.....	467
5.1.5 字符串排序算法的选择	.....	470
5.2 单词查找树	.....	474
5.2.1 单词查找树	.....	475
5.2.2 单词查找树的性质	.....	483
5.2.3 三向单词查找树	.....	485
5.2.4 三向单词查找树的性质	.....	487
5.2.5 应该使用字符串符号表的哪种实现	.....	489
5.3 子字符串查找	.....	493
5.3.1 历史简介	.....	493
5.3.2 暴力子字符串查找算法	.....	494

5.3.3 Knuth-Morris-Pratt子字符串 查找算法 .....	496
5.3.4 Boyer-Moore字符串查找算 法 .....	502
5.3.5 Rabin-Karp指纹字符串查找 算法 .....	505
5.3.6 总结 .....	509
5.4 正则表达式 .....	514
5.4.1 使用正则表达式描述模式 .....	514
5.4.2 缩略写法 .....	516
5.4.3 正则表达式的实际应用 .....	517
5.4.4 非确定有限状态自动机 .....	518
5.4.5 模拟NFA的运行 .....	520
5.4.6 构造与正则表达式对应的 NFA .....	522
5.5 数据压缩 .....	529
5.5.1 游戏规则 .....	529
5.5.2 读写二进制数据 .....	530
5.5.3 局限 .....	533
5.5.4 热身运动：基因组 .....	534
5.5.5 游程编码 .....	537
5.5.6 霍夫曼压缩 .....	540
第6章 背景 .....	558
索引 .....	611



# 第1章 基础

本书的目的是研究多种重要而实用的算法，即适合用计算机实现的解决问题的方法。和算法关系最紧密的是数据结构，即便于算法操作的组织数据的方法。本章介绍的就是学习算法和数据结构所需要的基本工具。

首先要介绍的是我们的基础编程模型。本书中的程序只用到了 Java 语言的一小部分，以及我们自己编写的用于封装输入输出以及统计的一些库。1.1 节总结了相关的语法、语言特性和书中将会用到的库。

接下来我们的重点是数据抽象并定义抽象数据类型（ADT）以进行模块化编程。在 1.2 节中我们介绍了用 Java 实现抽象数据类型的过程，包括定义它的应用程序编程接口（API）然后通过 Java 的类机制来实现它以供各种用例使用。

之后，作为重要而实用的例子，我们将学习三种基础的抽象数据类型：背包、队列和栈。1.3 节用数组、变长数组和链表实现了背包、队列和栈的 API，它们是全书算法实现的起点和样板。

性能是算法研究的一个核心问题。1.4 节描述了分析算法性能的方法。我们的基本做法是科学式的，即先对性能提出假设，建立数学模型，然后用多种实验验证它们，必要时重复这个过程。

我们用一个连通性问题作为例子结束本章，它的解法所用到的算法和数据结构可以实现经典的 union-find 抽象数据结构。

1  
2  
3

## 算法

编写一段计算机程序一般都是实现一种已有的方法来解决某个问题。这种方法大多和使用的编程语言无关——它适用于各种计算机以及编程语言。是这种方法而非计算机程序本身描述了解决问题的步骤。在计算机科学领域，我们用算法这个词来描述一种有限、确定、有效的并适合用计算机程序来实现的解决问题的方法。算法是计算机科学的基础，是这个领域研究的核心。

要定义一个算法，我们可以用自然语言描述解决某个问题的过程或是编写一段程序来实现这个过程。如发明于 2300 多年前的欧几里德算法所示，其目的是找到两个数的最大公约数：

### 自然语言描述

计算两个非负整数  $p$  和  $q$  的最大公约数：若  $q$  是 0，则最大公约数为  $p$ 。否则，将  $p$  除以  $q$  得到余数  $r$ ， $p$  和  $q$  的最大公约数即为  $q$  和  $r$  的最大公约数。

### Java 语言描述

```
Public static int gcd(int p, int q)
{
    if (q == 0) return p;
    int r = p % q;
    return gcd(q, r);
}
```

欧几里德算法

如果你不熟悉欧几里德算法，那么你应该在学习了1.1节之后完成练习1.1.24和练习1.1.25。在本书中，我们将用计算机程序来描述算法。这样做的一个重要原因之一是可以更容易地验证它们是否如所要求的那样有限、确定和有效。但你还应该意识到用某种特定语言写出一段程序只是表达一个算法的一种方法。数十年来本书中许多算法都曾被表达为多种编程语言的程序，这正说明每种算法都是适合于在任何计算机上用任何编程语言实现的方法。

我们关注的大多数算法都需要适当地组织数据，而为了组织数据就产生了数据结构，数据结构也是计算机科学研究的核心对象，它和算法的关系非常密切。在本书中，我们的观点是数据结构是算法的副产品或是结果，因此要理解算法必须学习数据结构。简单的算法也会产生复杂的数据结构，相应地，复杂的算法也许只需要简单数据结构。本书中我们将会研讨许多数据结构的性质，也许本书就应该叫《算法与数据结构》。

**4** 当用计算机解决一个问题时，一般都存在多种不同的方法。对于小型问题，只要管用，方法的不同并没有什么关系。但是对于大型问题（或者是需要解决大量小型问题的应用），我们就需要设计能够有效利用时间和空间的方法了。

学习算法的主要原因是它们能节约非常多的资源，甚至能够让我们完成一些本不可能完成的任务。在某些需要处理上百万个对象的应用程序中，设计优良的算法甚至可以将程序运行的速度提高数百万倍。在本书中我们将在多个场景中看到这样的例子。与此相反，花费金钱和时间去购置新的硬件可能只能将速度提高十倍或是百倍。无论在任何应用领域，精心设计的算法都是解决大型问题最有效的方法。

在编写庞大或者复杂的程序时，理解和定义问题、控制问题的复杂度和将其分解为更容易解决的子问题需要大量的工作。很多时候，分解后的子问题所需的算法实现起来都比较简单。但是在大多数情况下，某些算法的选择是非常关键的，因为大多数系统资源都会消耗在它们身上。本书的焦点就是这类算法。我们所研究的基础算法在许多应用领域都是解决困难问题的有效方法。

计算机程序的共享已经变得越来越广泛，尽管书中涉及了许多算法，我们也只实现了其中的一小部分。例如，Java库包含了许多重要算法的实现。但是，实现这些基础算法的简化版本有助于我们更好地理解、使用和优化它们在库中的高级版本。更重要的是，我们经常需要重新实现这些基础算法，因为在全新的环境中（无论是硬件的还是软件的），原有的实现无法将新环境的优势完全发挥出来。在本书中，我们的重点是用最简洁的方式实现优秀的算法。我们会仔细地实现算法的关键部分，并尽最大努力揭示如何进行有效的底层优化工作。

**5** 为一项任务选择最合适的算法是困难的，这可能会需要复杂的数学分析。计算机科学中研究这种问题的分支叫做算法分析。通过分析，我们将要学习的许多算法都有着优秀的理论性能；而另一些我们则只是根据经验知道它们是可用的。我们的主要目标是学习典型问题的各种有效算法，但也会注意比较不同算法之间的性能差异。不应该使用资源消耗情况未知的算法，因此我们会时刻关注算法的期望性能。

## 本书框架

接下来概述一下全书的主要内容，给出涉及的主题以及本书大致的组织结构。这组主题触及了尽可能多的基础算法，其中的某些领域是计算机科学的核心内容，通过对这些领域的深入研究，我们找出了应用广泛的基本算法，而另一些算法则来自计算机科学和相关领域比较前沿的研究成果。总之，本书讨论的算法都是数十年来研发的重要成果，它们将继续在快速发展的计算机应用中扮演重要角色。

## 第1章 基础

它讲解了在随后的章节中用来实现、分析和比较算法的基本原则和方法，包括 Java 编程模型、数据抽象、基本数据结构、集合类的抽象数据类型、算法性能分析的方法和一个案例分析。

## 第2章 排序

有序地重新排列数组中的元素是非常重要的基础算法。我们会深入研究各种排序算法，包括插入排序、选择排序、希尔排序、快速排序、归并排序和堆排序。同时我们还会讨论另外一些算法，它们用于解决几个与排序相关的问题，例如优先队列、选举以及归并。其中许多算法会成为后续章节中其他算法的基础。

## 第3章 查找

从庞大的数据集中找到指定的条目也是非常重要的。我们将会讨论基本的和高级的查找算法，包括二叉查找树、平衡查找树和散列表。我们会梳理这些方法之间的关系并比较它们的性能。

## 第4章 图

图的主要内容是对象和它们的连接，连接可能有权重和方向。利用图可以为大量重要而困难的问题建模，因此图算法的设计也是本书的一个主要研究领域。我们会研究深度优先搜索、广度优先搜索、连通性问题以及若干其他算法和应用，包括 Kruskal 和 Prim 的最小生成树算法、Dijkstra 和 Bellman-Ford 的最短路径算法。

6

## 第5章 字符串

字符串是现代应用程序中的重要数据类型。我们将会研究一系列处理字符串的算法，首先是对字符串键的排序和查找的快速算法，然后是子字符串查找、正则表达式模式匹配和数据压缩算法。此外，在分析一些本身就十分重要的基础问题之后，这一章对相关领域的前沿话题也作了介绍。

## 第6章 背景

这一章将讨论与本书内容有关的若干其他前沿研究领域，包括科学计算、运筹学和计算理论。我们会介绍性地讲一下基于事件的模拟、B 树、后缀数组、最大流量问题以及其他高级主题，以帮助读者理解算法在许多有趣的前沿研究领域中所起到的巨大作用。最后，我们会讲一讲搜索问题、问题转化和 NP 完全性等算法研究的支柱理论，以及它们和本书内容的联系。

学习算法是非常有趣和令人激动的，因为这是一个历久弥新的领域（我们学习的绝大多数算法都还不到“五十岁”，有些还是最近才发明的，但也有一些算法已经有数百年的历史）。这个领域不断有新的发现，但研究透彻的算法仍然是少数。本书中既有精巧、复杂和高难度的算法，也有优雅、朴素和简单的算法。在科学和商业应用中，我们的目标是理解前者并熟悉后者，这样才能掌握这些有用的工具并学会算法式思考，以迎接未来计算任务的挑战。

7

## 1.1 基础编程模型

我们学习算法的方法是用 Java 编程语言编写的程序来实现算法。这样做是出于以下原因：

- 程序是对算法精确、优雅和完全的描述；
- 可以通过运行程序来学习算法的各种性质；
- 可以在应用程序中直接使用这些算法。

相比用自然语言描述算法，这些是重要而巨大的优势。

这样做的一个缺点是我们要使用特定的编程语言，这会使分离算法的思想和实现细节变得困难。

我们在实现算法时考虑到了这一点，只使用了大多数现代编程语言都具有且能够充分描述算法所必需的语法。

我们仅使用了 Java 的一个子集。尽管我们没有明确地说明这个子集的范围，但你也会看到我们只使用了很少的 Java 特性，而且会优先使用大多数现代编程语言所共有的语法。我们的代码是完整的，因此希望你能下载这些代码并用我们的测试数据或是你自己的来运行它们。

我们把描述和实现算法所用到的语言特性、软件库和操作系统特性总称为基础编程模型。本节以及 1.2 节会详细说明这个模型，相关内容自成一体，主要是作为文档供读者查阅，以便理解本书的代码。我们的另一本入门级的书籍 *An Introduction to Programming in Java: An Interdisciplinary Approach* 也使用了这个模型。

作为参考，图 1.1.1 所示的是一个完整的 Java 程序。它说明了我们的基础编程模型的许多基本特点。在讨论语言特性时我们会用这段代码作为例子，但可以先不用考虑代码的实际意义（它实现了经典的二分查找算法，并在白名单过滤应用中对算法进行了检验，请见 1.1.10 节）。我们假设你具备某种主流语言编程的经验，因此你应该知道这段代码中的大多数要点。图中的注释应该能够解答你的任何疑问。因为图中的代码某种程度上反映了本书代码的风格，而且对各种 Java 编程惯例和语言构造，在用法上我们都力求一致，所以即使是经验丰富的 Java 程序员也应该看一看。

8

### 1.1.1 Java 程序的基本结构

一段 Java 程序（类）或者是一个静态方法（函数）库，或者定义了一个数据类型。要创建静态方法库和定义数据类型，会用到下面五种语法，它们是 Java 语言的基础，也是大多数现代语言所共有的。

- **原始数据类型：**它们在计算机程序中精确地定义整数、浮点数和布尔值等。它们的定义包括取值范围和能够对相应的值进行的操作，它们能够被组合为类似于数学公式定义的表达式。
- **语句：**语句通过创建变量并对其赋值、控制运行流程或者引发副作用来进行计算。我们会使用六种语句：声明、赋值、条件、循环、调用和返回。
- **数组：**数组是多个同种数据类型的值的集合。
- **静态方法：**静态方法可以封装并重用代码，使我们可以用独立的模块开发程序。
- **字符串：**字符串是一连串的字符，Java 内置了对它们的一些操作。
- **标准输入 / 输出：**标准输入输出是程序与外界联系的桥梁。
- **数据抽象：**数据抽象封装和重用代码，使我们可以定义非原始数据类型，进而支持面向对象编程。

我们将在本节学习前五种语法，数据抽象是下一节的主题。

运行 Java 程序需要和操作系统或开发环境打交道。为了清晰和简洁，我们把这种输入命令执行程序的环境称为虚拟终端。请登录本书的网站去了解如何使用虚拟终端，或是现代系统中许多其他高级的编程开发环境的使用方法。

在例子中，BinarySearch 类有两个静态方法 `rank()` 和 `main()`。第一个方法 `rank()` 含有四条语句：两条声明语句，一条循环语句（该语句中又有一条赋值语句和两条条件语句）和一条返回语句。

第二个方法 `main()` 包含三条语句：一条声明语句、一条调用语句和一个循环语句（该语句中又包含一条赋值语句和一条条件语句）。

要执行一个 Java 程序，首先需要用 `javac` 命令编译它，然后再用 `java` 命令运行它。例如，要运行 `BinarySearch`，首先要输入 `javac BinarySearch.java`（这将生成一个叫 `BinarySearch.class` 的文件，其中含有这个程序的 Java 字节码）；然后再输入 `java BinarySearch`（接着是一个白名单文件名）把控制权移交给这段字节码程序。为了理解这段程序，我们接下来要详细介绍原始数据类型和表达式，各种 Java 语句、数组、静态方法、字符串和输入输出。

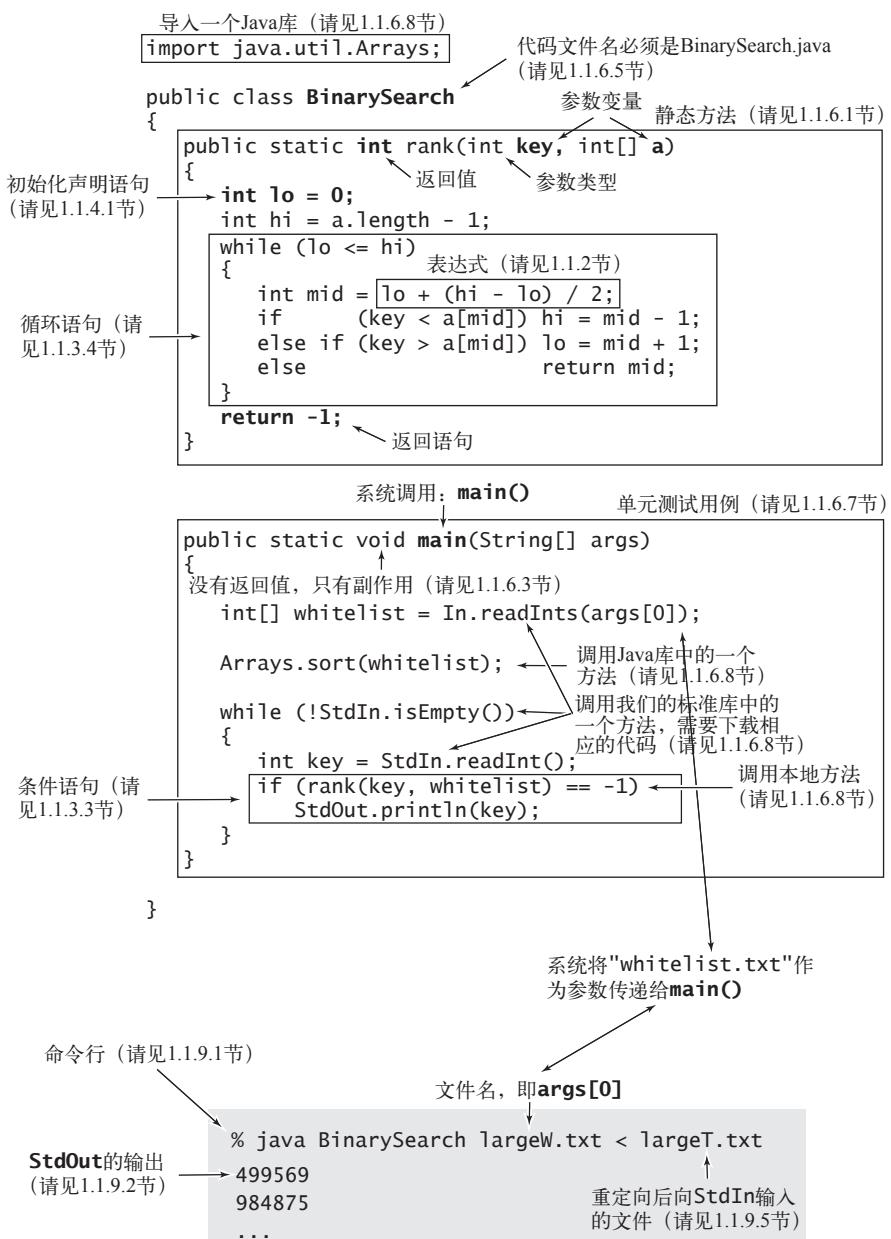


图 1.1.1 Java 程序及其命令行的调用

### 1.1.2 原始数据类型与表达式

数据类型就是一组数据和对其所能进行的操作的集合。首先考虑以下4种Java语言最基本的基本数据类型：

- 整型，及其算术运算符(`int`)；
- 浮点型，及其算术运算符(`double`)；
- 布尔型，它的值`{true, false}`及其逻辑操作(`boolean`)；
- 字符型，它的值是你能够输入的英文字母数字字符和符号(`char`)。

接下来我们看看如何指明这些类型的值和对这些类型的操作。

Java程序控制的是用标识符命名的变量。每个变量都有自己的类型并存储了一个合法的值。在Java代码中，我们用类似数学表达式的表达式来实现对各种类型的操作。对于原始类型来说，我们用标识符来引用变量，用`+`、`-`、`*`、`/`等运算符来指定操作，用字面量，例如`1`或者`3.14`来表示值，用形如`(x+2.236)/2`的表达式来表示对值的操作。表达式的目的就是计算某种数据类型的值。表1.1.1对这些基本内容进行了说明。

表 1.1.1 Java 程序的基本组成

术 语	例 子	定 义
原始数据类型	<code>int double boolean char</code>	一组数据和对其所能进行的操作的集合(Java语言内置)
标识符	<code>a abc Ab\$ a_b ab123 lo hi</code>	由字母、数字、下划线和\$组成的字符串，首字符不能是数字
变量	[任意标识符]	表示某种数据类型的值
运算符	<code>+ - * /</code>	表示某种数据类型的运算
字面量	<code>int 1 0 -42 double 2.0 1.0e-15 3.14 boolean true false char 'a' '+' '9' '\n'</code>	值在源代码中的表示
表达式	<code>int lo + (hi - lo) / 2 double 1.0e-15 * t boolean lo &lt;= hi</code>	字面量、变量或是能够计算出结果的一串字面量、变量和运算符的组合

11

只要能够指定值域和在此值域上的操作，就能定义一个数据类型。表1.1.2总结了Java的`int`、`double`、`boolean`和`char`类型的相关信息。许多现代编程语言中的基本数据类型和它们都很相似。对于`int`和`double`来说，这些操作是我们熟悉的算数运算；对于`boolean`来说则是逻辑运算。需要注意的重要一点是，`+`、`-`、`*`、`/`都是被重载过的——根据上下文，同样的运算符对不同类型会执行不同的操作。这些初级运算的关键性质是运算产生的数据的数据类型和参与运算的数据的数据类型是相同的。这也意味着我们经常需要处理近似值，因为很多情况下由表达式定义的准确值并非参与表达式运算的值。例如，`5/3`的值是`1`而`5.0/3.0`的值是`1.6666666666666667`，两者都很接近但并不准确地等于`5/3`。下表并不完整，我们会在本节最后的答疑部分中讨论更多运算符和偶尔需要考虑到的各种异常情况。

表 1.1.2 Java 中的原始数据类型

类 型	值 域	运 算 符	典型表达式 表 达 式	值
<code>int</code>	<code>-2<sup>31</sup> 至 +2<sup>31</sup>-1 之间的整数(32位,二进制补码)</code>	<code>+(加) -(减) *(乘) /(除) %(求余)</code>	<code>5 + 3 5 - 3 5 * 3 5 / 3 5 % 3</code>	<code>8 2 15 1 2</code>

(续)

类 型	值 域	运 算 符	典型表达式 表 达 式	值
double	双精度实数 (64位, IEEE 754 标准)	+ (加) - (减) * (乘) / (除)	3.141 - 0.03 2.0 - 2.0e-7 100 * 0.015 6.02e23 / 2.0	3.111 1.9999998 1.5 3.01e23
boolean	true 或 false	&& (与)    (或) ! (非) ^ (异或)	true && false false    true !false true ^ true	false true true false
char	字符 (16位)	(算术运算符, 但很少使用)		

12

### 1.1.2.1 表达式

如表 1.1.2 所示, Java 使用的是中缀表达式: 一个字面量(或是一个表达式),紧接着是一个运算符,再接着是另一个字面量(或者另一个表达式)。当一个表达式包含一个以上的运算符时,运算符的作用顺序非常重要,因此 Java 语言规范约定了如下的运算符优先级: 运算符 \* 和 / (以及 %) 的优先级高于 + 和 - (优先级越高, 越早运算); 在逻辑运算符中, ! 拥有最高优先级, 之后是 &&, 接下来是 ||。一般来说, 相同优先级的运算符的运算顺序是从左至右。与在正常的算数表达式中一样, 使用括号能够改变这些规则。因为不同语言中的优先级规则会有些许不同, 我们在代码中会使用括号并用各种方法努力消除对优先级规则的依赖。

### 1.1.2.2 类型转换

如果不会损失信息, 数值会被自动提升为高级的数据类型。例如, 在表达式 `1+2.5` 中, 1 会被转换为浮点数 `1.0`, 表达式的值也为 `double` 值 `3.5`。转换指的是在表达式中把类型名放在括号里将其后的值转换为括号中的类型。例如, `(int)3.7` 的值是 `3` 而 `(double)3` 的值是 `3.0`。需要注意的是将浮点型转换为整型将会截断小数部分而非四舍五入, 在复杂的表达式中的类型转换可能会很复杂, 应该小心并尽量少使用类型转换, 最好是在表达式中只使用同一类型的字面量和变量。

### 1.1.2.3 比较

下列运算符能够比较相同数据类型的两个值并产生一个布尔值: 相等 (`==`)、不等 (`!=`)、小于 (`<`)、小于等于 (`<=`)、大于 (`>`) 和大于等于 (`>=`)。这些运算符被称为混合类型运算符, 因为它们的结果是布尔型, 而不是参与比较的数据类型。结果是布尔型的表达式被称为布尔表达式。我们将会看到这种表达式是条件语句和循环语句的重要组成部分。

### 1.1.2.4 其他原始类型

Java 的整型能够表示  $2^{32}$  个不同的值, 用一个 32 位二进制即可表示 (虽然现在的许多计算机有 64 位二进制, 但整型仍然是 32 位)。与此相似, 浮点型的标准规定为 64 位。这些大小对于一般应用程序中使用的整数和实数已经足够了。为了提供更大的灵活性, Java 还提供了其他五种原始数据类型:

- 64 位整数, 及其算术运算符 (`long`);
- 16 位整数, 及其算术运算符 (`short`);
- 16 位字符, 及其算术运算符 (`char`);
- 8 位整数, 及其算术运算符 (`byte`);
- 32 位单精度实数, 及其算术运算符 (`float`)。

13 在本书中我们大多使用 `int` 和 `double` 进行算术运算，因此我们在此不会再详细讨论其他类似的数据类型。

### 1.1.3 语句

Java 程序是由语句组成的。语句能够通过创建和操作变量、对变量赋值并控制这些操作的执行流程来描述运算。语句通常会被组织成代码段，即花括号中的一系列语句。

- **声明语句：** 创建某种类型的变量并用标识符为其命名。
- **赋值语句：** 将（由表达式产生的）某种类型的数值赋予一个变量。Java 还有一些隐式赋值的语法可以使某个变量的值相对于当前值发生变化，例如将一个整型值加 1。
- **条件语句：** 能够简单地改变执行流程——根据指定的条件执行两个代码段之一。
- **循环语句：** 更彻底地改变执行流程——只要条件为真就不断地反复执行代码段中的语句。
- **调用和返回语句：** 和静态方法有关（见 1.1.6 节），是改变执行流程和代码组织的另一种方式。

程序就是由一系列声明、赋值、条件、循环、调用和返回语句组成的。一般来说代码的结构都是嵌套的：一个条件语句或循环语句的代码段中也能包含条件语句或是循环语句。例如，`rank()` 中的 `while` 循环就包含一个 `if` 语句。接下来，我们逐个说明各种类型的语句。

#### 1.1.3.1 声明语句

声明语句将一个变量名和一个类型在编译时关联起来。Java 需要我们用声明语句指定变量的名称和类型。这样，我们就清楚地指明了能够对其进行的操作。Java 是一种强类型的语言，因为 Java 编译器会检查类型的一致性（例如，它不会允许将布尔类型和浮点类型的变量相乘）。变量可以声明在第一次使用之前的任何地方——一般我们都在首次使用该变量的时候声明它。变量的作用域就是定义它的地方，一般由相同代码段中声明之后的所有语句组成。

#### 1.1.3.2 赋值语句

赋值语句将（由一个表达式定义的）某个数据类型的值和一个变量关联起来。在 Java 中，当我们写下 `c=a+b` 时，我们表达的不是数学等式，而是一个操作，即令变量 `c` 的值等于变量 `a` 的值与变量 `b` 的值之和。当然，在赋值语句执行后，从数学上来说 `c` 的值必然会等于 `a+b`，但语句的目的是改变 `c` 的值（如果需要的话）。赋值语句的左侧必须是单个变量，右侧可以是能够得到相应类型的值的任意表达式。

#### 1.1.3.3 条件语句

大多数运算都需要用不同的操作来处理不同的输入。在 Java 中表达这种差异的一种方法是 `if` 语句：

```
if (<boolean expression>) { <block statements> }
```

这种描述方式是一种叫做模板的形式记法，我们偶尔会使用这种格式来表示 Java 的语法。尖括号 (`<>`) 中的是我们已经定义过的语法，这表示我们可以在指定的位置使用该语法的任意实例。在这里，`<boolean expression>` 表示一个布尔表达式，例如一个比较操作。`<block statements>` 表示一段 Java 语句。我们也可以给出 `<boolean expression>` 和 `<block statements>` 的形式定义，不过我们不想深入这些细节。`if` 语句的意义不言自明：当且仅当布尔表达式的值为真 (`true`) 时代码段中的语句才会被执行。以下 `if-else` 语句能够在两个代码段之间作出选择：

```
if (<boolean expression>) { <block statements> }
else { <block statements> }
```

#### 1.1.3.4 循环语句

许多运算都需要重复。Java 语言中处理这种计算的基本语句的格式是：

```
while (<boolean expression>) { <block statements> }
```

`while` 语句和 `if` 语句的形式相似（只是用 `while` 代替了 `if`），但意义大有不同。当布尔表达式的值为假 (`false`) 时，代码什么也不做；当布尔表达式的值为真 (`true`) 时，执行代码段（和 `if` 一样），然后再次检查布尔表达式的值，如果仍然为真，再次执行代码段。如此这般，只要布尔表达式的值为真，就继续执行代码段。我们将循环语句中的代码段称为循环体。

#### 1.1.3.5 `break` 与 `continue` 语句

有些情况下我们也会需要比基本的 `if` 和 `while` 语句更加复杂的流程控制。相应地，Java 支持在 `while` 循环中使用另外两条语句：

- `break` 语句，立即从循环中退出；
- `continue` 语句，立即开始下一轮循环。

本书很少在代码中使用它们（许多程序员从来都不用），但在某些情况下它们的确能够大大简化代码。

15

#### 1.1.4 简便记法

程序有很多种写法，我们追求清晰、优雅和高效的代码。这样的代码经常会使用以下这些广为流传的简便写法（不仅仅是 Java，许多语言都支持它们）。

##### 1.1.4.1 声明并初始化

可以将声明语句和赋值语句结合起来，在声明（创建）一个变量的同时将它初始化。例如，`int i = 1;` 创建了名为 `i` 的变量并赋予其初始值 1。最好在接近首次使用变量的地方声明它并将其实例化（为了限制它的作用域）。

##### 1.1.4.2 隐式赋值

当希望一个变量的值相对于其当前值变化时，可以使用一些简便的写法。

- 递增 / 递减运算符，`++i`；等价于 `i=i+1`；且表达式为 `i+1`。类似地，`--i`；等价于 `i=i-1`；`i++`；和 `i--`；的意思相同，只是表达式的值为 `i` 的值。
- 其他复合运算符，在赋值语句中将一个二元运算符写在等号之前，等价于将左边的变量放在等号右边并作为第一个操作数。例如，`i/=2`；等价于 `i=i/2`。注意，`i += 1`；等价于 `i = i + 1`；（以及 `++i`）。

##### 1.1.4.3 单语句代码段

如果条件或循环语句的代码段只有一条语句，代码段的花括号可以省略。

##### 1.1.4.4 `for` 语句

很多循环的模式都是这样的：初始化一个索引变量，然后使用 `while` 循环并将包含索引变量的表达式作为循环的条件，`while` 循环的最后一条语句会将索引变量加 1。使用 Java 的 `for` 语句可以更紧凑地表达这种循环：

```
for (<initialize>; <boolean expression>; <increment>)
{
    <block statements>
}
```

除了几种特殊情况之外，这段代码都等价于：

```
<initialize>;
while (<boolean expression>)
{
    <block statements>
    <increment>;
}
```

我们将使用 **for** 语句来表示对这种初始化—递增循环用法的支持。

表 1.1.3 总结了各种 Java 语句及其示例与定义。

表 1.1.3 Java 语句

语句	示例	定义
声明语句	int i; double c;	创建一个指定类型的变量并用标识符命名
赋值语句	a = b + 3; discriminant = b * b - 4.0 * c;	将某一数据类型的值赋予一个变量
声明并初始化	int i = 1; double c = 3.14159265;	在声明时赋予变量初始值
隐式赋值	++i; i += 1;	i = i + 1;
条件语句 ( <b>if</b> )	if (x < 0) x = -x;	根据布尔表达式的值执行一条语句
条件语句 ( <b>if-else</b> )	if (x > y) max = x; else max = y;	根据布尔表达式的值执行两条语句中的一条
循环语句 ( <b>while</b> )	int v = 0; while(v <= N) v = 2 * v; double t = c; while (Math.abs(t - c/t) > 1e-15*t) t = (c/t + t) / 2.0;	执行语句，直至布尔表达式的值变为假 ( <b>false</b> )
循环语句 ( <b>for</b> )	for (int i = 1; i <= N; i++) sum += 1.0/i; for (int i = 0; i <= N; i++) StdOut.println(2*Math.PI*i/N);	<b>while</b> 语句的简化版
调用语句	int key = StdIn.readInt();	调用另一方法（请见 1.1.6.2 节）
返回语句	return false;	从方法中返回（请见 1.1.6.3 节）

17

## 1.1.5 数组

数组能够顺序存储相同类型的多个数据。除了存储数据，我们也希望能够访问数据。访问数组中的某个元素的方法是将其编号然后索引。如果我们有  $N$  个值，它们的编号则为 0 至  $N-1$ 。这样对于 0 到  $N-1$  之间任意的  $i$ ，我们就能够在 Java 代码中用  $a[i]$  唯一地表示第  $i$  个元素的值。在 Java 中这种数组被称为一维数组。

### 1.1.5.1 创建并初始化数组

在 Java 程序中创建一个数组需要三步：

- 声明数组的名字和类型；
- 创建数组；
- 初始化数组元素。

在声明数组时，需要指定数组的名称和它含有的数据的类型。在创建数组时，需要指定数组的长度（元素的个数）。例如，在以下代码中，“完整模式”部分创建了一个有  $N$  个元素的 **double** 数组，

所有的元素的初始值都是 0.0。第一条语句是数组的声明，它和声明一个相应类型的原始数据类型变量十分相似，只有类型名之后的方括号说明我们声明的是一个数组。第二条语句中的关键字 new 使 Java 创建了这个数组。我们需要在运行时明确地创建数组的原因是 Java 编译器在编译时无法知道应该为数组预留多少空间（对于原始类型则可以）。for 语句初始化了数组的 N 个元素，将它们的值置为 0.0。在代码中使用数组时，一定要依次声明、创建并初始化数组。忽略了其中的任何一步都是很常见的编程错误。

### 1.1.5.2 简化写法

为了精简代码，我们常常会利用 Java 对数组默认的初始化来将三个步骤合为一条语句，即上例中的简化写法。等号的左侧声明了数组，等号的右侧创建了数组。这种写法不需要 for 循环，因为在 Java 数组中 double 类型的变量的默认初始值都是 0.0，但如果你想使用不同的初始值，那么就需要使用 for 循环了。数值类型的默认初始值是 0，布尔型的默认初始值是 false。例子中的第三种方式用花括号将一列由逗号分隔的值在编译时将数组初始化。

### 1.1.5.3 使用数组

典型的数组处理代码请见表 1.1.4。在声明并创建数组之后，在代码的任何地方都能通过数组名之后的方括号中的索引来访问其中的元素。数组一经创建，它的大小就是固定的。程序能够通过 a.length 获取数组 a[] 的长度，而它的最后一个元素总是 a[a.length - 1]。Java 会自动进行边界检查——如果你创建了一个大小为 N 的数组，但使用了一个小于 0 或者大于 N-1 的索引访问它，程序会因为运行时抛出 ArrayOutOfBoundsException 异常而终止。

18

完整模式	<pre>double[] a; a = new double[N]; for (int i = 0; i &lt; N; i++)     a[i] = 0.0;</pre>	声明数组 创建数组 初始化数组
简化写法	<pre>double[] a = new double[N];</pre>	
声明初始化	<pre>int[] a = { 1, 1, 2, 3, 5, 8 };</pre>	
		声明、创建并初始化一个数组

表 1.1.4 典型的数组处理代码

任 务	实现（代码片段）
找出数组中最大的元素	<pre>double max = a[0]; for (int i = 1; i &lt; a.length; i++)     if (a[i] &gt; max) max = a[i];</pre>
计算数组元素的平均值	<pre>int N = a.length; double sum = 0.0; for (int i = 0; i &lt; N; i++)     sum += a[i]; double average = sum / N;</pre>
复制数组	<pre>int N = a.length; double[] b = new double[N]; for (int i = 0; i &lt; N; i++)     b[i] = a[i];</pre>
颠倒数组元素的顺序	<pre>int N = a.length; for (int i = 0; i &lt; N/2; i++) {     double temp = a[i];     a[i] = a[N-1-i];     a[N-i-1] = temp; }</pre>

(续)

任 务	实现(代码片段)
矩阵相乘(方阵) a[][] * b[][] = c[][]	<pre>int N = a.length; double[][] c = new double[N][N]; for (int i = 0; i &lt; N; i++)     for (int j = 0; j &lt; N; j++)         { // 计算行 i 和列 j 的点乘             for (int k = 0; k &lt; N; k++)                 c[i][j] += a[i][k]*b[k][j];         }     }</pre>

### 1.1.5.4 起别名

请注意，数组名表示的是整个数组——如果我们将一个数组变量赋予另一个变量，那么两个变量将会指向同一个数组。例如以下这段代码：

```
int[] a = new int[N];
...
a[i] = 1234;
...
int[] b = a;
...
b[i] = 5678; // a[i] 的值也会变成 5678
```

这种情况叫做起别名，有时可能会导致难以察觉的问题。如果你是想将数组复制一份，那么应该声明、创建并初始化一个新的数组，然后将原数组中的元素值挨个复制到新数组，如表 1.1.4 的第三个例子所示。

### 1.1.5.5 二维数组

在 Java 中二维数组就是一维数组的数组。二维数组可以是参差不齐的（元素数组的长度可以不一致），但大多数情况下（根据合适的参数  $M$  和  $N$ ）我们都会使用  $M \times N$ ，即  $M$  行长度为  $N$  的数组的二维数组（也可以称数组含有  $N$  列）。在 Java 中访问二维数组也很简单。二维数组  $a[][]$  的第  $i$  行第  $j$  列的元素可以写作  $a[i][j]$ 。声明二维数组需要两对方括号。创建二维数组时要在类型名之后分别在方括号中指定行数以及列数，例如：

```
double[][] a = new double[M][N];
```

我们将这样的数组称为  $M \times N$  的数组。我们约定，第一维是行数，第二维是列数。和一维数组一样，Java 会将数值类型的数组元素初始化为 0，将布尔型的数组元素初始化为 `false`。默认的初始化对二维数组更有用，因为可以节约更多的代码。下面这段代码和刚才只用一行就完成创建和初始化的语句是等价的：

```
double[][] a;
a = new double[M][N];
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        a[i][j] = 0.0;
```

在将二维数组初始化为 0 时这段代码是多余的，但是如果想要初始化为其他值，我们就需要嵌套的 `for` 循环了。

### 1.1.6 静态方法

本书中的所有 Java 程序要么是数据类型的定义（详见 1.2 节），要么是一个静态方法库。在许多语言中，静态方法被称为函数，因为它们和数学函数的性质类似。静态方法是一组在被调用时会

被顺序执行的语句。修饰符 `static` 将这类方法和 1.2 节的实例方法区别开来。当讨论两类方法共有的属性时我们会使用不加定语的方法一词。

### 1.1.6.1 静态方法

方法封装了由一系列语句所描述的运算。方法需要参数（某种数据类型的值）并根据参数计算出某种数据类型的返回值（例如数学函数的结果）或者产生某种副作用（例如打印一个值）。`BinarySearch` 中的静态函数 `rank()` 是前者的一个例子；`main()` 则是后者的一个例子。每个静态方法都是由签名（关键字 `public static` 以及函数的返回值，方法名以及一串各种类型的参数）和函数体（即包含在花括号中的代码）组成的，如图 1.1.2 所示。静态函数的例子请见表 1.1.5。

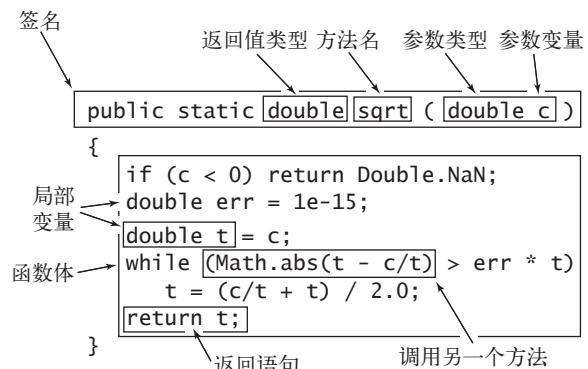


图 1.1.2 静态方法解析

表 1.1.5 典型静态方法的实现

任 务	实 现
计算一个整数的绝对值	<pre>public static int abs(int x) {     if (x &lt; 0) return -x;     else        return x; }</pre>
计算一个浮点数的绝对值	<pre>public static double abs(double x) {     if (x &lt; 0.0) return -x;     else          return x; }</pre>
判定一个数是否是素数	<pre>public static boolean isPrime(int N) {     if (N &lt; 2) return false;     for (int i = 2; i*i &lt;= N; i++)         if (N % i == 0) return false;     return true; }</pre>
计算平方根（牛顿迭代法）	<pre>public static double sqrt(double c) {     if (c &lt; 0) return Double.NaN;     double err = 1e-15;     double t = c;     while (Math.abs(t - c/t) &gt; err * t)         t = (c/t + t) / 2.0;     return t; }</pre>
计算直角三角形的斜边	<pre>public static double hypotenuse(double a, double b) { return Math.sqrt(a*a + b*b); }</pre>
计算调和级数（请见表 1.4.5）	<pre>public static double H(int N) {     double sum = 0.0;     for (int i = 1; i &lt;= N; i++)         sum += 1.0 / i;     return sum; }</pre>

### 1.1.6.2 调用静态方法

调用静态方法的方法是写出方法名并在后面的括号中列出参数值，用逗号分隔。当调用是表达式的一部分时，方法的返回值将会替代表达式中的方法调用。例如，`BinarySearch` 中调用 `rank()` 返回了一个 `int` 值。仅由一个方法调用和一个分号组成的语句一般用于产生副作用。例如，`BinarySearch` 的 `main()` 函数中对系统方法 `Arrays.sort()` 的调用产生的副作用，是将数组中的所有条目有序地排列。调用方法时，它的参数变量将被初始化为调用时所给出的相应表达式的值。返回语句将结束静态方法并将控制权交还给调用者。如果静态方法的目的是计算某个值，返回语句应该指定这个值（如果这样的静态方法在执行完所有的语句之后都没有返回语句，编译器会报错）。

22  
23

### 1.1.6.3 方法的性质

对方法所有性质的完整描述超出了本书的范畴，但以下几点值得一提。

- **方法的参数按值传递：**在方法中参数变量的使用方法和局部变量相同，唯一不同的是参数变量的初始值是由调用方提供的。方法处理的是参数的值，而非参数本身。这种方式产生的结果是在静态方法中改变一个参数变量的值对调用者没有影响。本书中我们一般不会修改参数变量。值传递也意味着数组参数将会是原数组的别名（见 1.1.5.4 节）——方法中使用的参数变量能够引用调用者的数组并改变其内容（只是不能改变原数组变量本身）。例如，`Arrays.sort()` 将能够改变通过参数传递的数组的内容，将其排序。
- **方法名可以被重载：**例如，Java 的 `Math` 包使用这种方法为所有的原始数值类型实现了 `Math.abs()`、`Math.min()` 和 `Math.max()` 函数。重载的另一种常见用法是为函数定义两个版本，其中一个需要一个参数而另一个则为该参数提供一个默认值。
- **方法只能返回一个值，但可以包含多个返回语句：**一个 Java 方法只能返回一个值，它的类型是方法签名中声明的类型。静态方法第一次执行到一条返回语句时控制权将会回到调用代码中。尽管可能存在多条返回语句，任何静态方法每次都只会返回一个值，即被执行的第一条返回语句的参数。
- **方法可以产生副作用：**方法的返回值可以是 `void`，这表示该方法没有返回值。返回值为 `void` 的静态函数不需要明确的返回语句，方法的最后一条语句执行完毕后控制权将会返回给调用方。我们称 `void` 类型的静态方法会产生副作用（接受输入、产生输出、修改数组或者改变系统状态）。例如，我们的程序中的静态方法 `main()` 的返回值就是 `void`，因为它的作用是向外输出。技术上来说，数学方法的返回值都不会是 `void` (`Math.random()` 虽然不接受参数但也有返回值)。

24

2.1 节所述的实例方法也拥有这些性质，尽管两者在副作用方面大为不同。

### 1.1.6.4 递归

方法可以调用自己（如果你对递归概念感到奇怪，请完成练习 1.1.16 到练习 1.1.22）。例如，下面给出了 `BinarySearch` 的 `rank()` 方法的另一种实现。我们会经常使用递归，因为递归代码比相应的非递归代码更加简洁优雅、易懂。下面这种实现中的注释就言简意赅地说明了代码的作用。我们可以用数学归纳法证明这段注释所解释的算法的正确性。我们会在 3.1 节中展开这个话题并为二分查找提供一个这样的证明。

编写递归代码时最重要的有以下三点。

- **递归总有一个最简单的情况——**方法的第一条语句总是一个包含 `return` 的条件语句。
- **递归调用总是去尝试解决一个规模更小的子问题，这样递归才能收敛到最简单的情况。**在下面的代码中，第四个参数和第三个参数的差值一直在缩小。

- 递归调用的父问题和尝试解决的子问题之间不应该有交集。在下面的代码中，两个子问题各自操作的数组部分是不同的。

```

public static int rank(int key, int[] a)
{ return rank(key, a, 0, a.length - 1); }

public static int rank(int key, int[] a, int lo, int hi)
{ //如果key存在于a[]中，它的索引不会小于lo且不会大于hi

    if (lo > hi) return -1;
    int mid = lo + (hi - lo) / 2;
    if      (key < a[mid]) return rank(key, a, lo, mid - 1);
    else if (key > a[mid]) return rank(key, a, mid + 1, hi);
    else
        return mid;
}

```

二分查找的递归实现

25

违背其中任意一条都可能得到错误的结果或是低效的代码（见练习 1.1.19 和练习 1.1.27），而坚持这些原则能写出清晰、正确且容易评估性能的程序。使用递归的另一个原因是我们可以使用数学模型的来估计程序的性能。我们会在 3.2 节的二分查找以及其他几个地方分析这个问题。

### 1.1.6.5 基础编程模型

静态方法库是定义在一个 Java 类中的一组静态方法。类的声明是 `public class` 加上类名，以及用花括号包含的静态方法。存放类的文件的文件名和类名相同，扩展名是 `.java`。Java 开发的基本模式是编写一个静态方法库（包含一个 `main()` 方法）来完成一个任务。输入 `java` 和类名以及一系列字符串就能调用类中的 `main()` 方法，其参数为由输入的字符串组成的一个数组。`main()` 的最后一条语句执行完毕之后程序终止。在本书中，当我们提到用于执行一项任务的 Java 程序时，我们指的是用这种模式开发的代码（可能还包括对数据类型的定义，如 1.2 节所示）。例如，`BinarySearch` 就是一个由两个静态方法 `rank()` 和 `main()` 组成的 Java 程序，它的作用是将输入中所有不在通过命令行指定的白名单中的数字打印出来。

### 1.1.6.6 模块化编程

这个模型的最重要之处在于通过静态方法库实现了模块化编程。我们可以构造许多个静态方法库（模块），一个库中的静态方法也能够调用另一个库中定义的静态方法。这能够带来许多好处：

- 程序整体的代码量很大时，每次处理的模块大小仍然适中；
- 可以共享和重用代码而无需重新实现；
- 很容易用改进的实现替换老的实现；
- 可以为解决编程问题建立合适的抽象模型；
- 缩小调试范围（请见 1.1.6.7 节关于单元测试的讨论）。

例如，`BinarySearch` 用到了三个独立的库，即我们的 `StdOut` 和 `StdIn` 以及 Java 的 `Arrays`，而这三个库又分别用到了其他的库。

### 1.1.6.7 单元测试

Java 编程的最佳实践之一就是每个静态方法库中都包含一个 `main()` 函数来测试库中的所有方法（有些编程语言不支持多个 `main()` 方法，因此不支持这种方式）。恰当的单元测试本身也是很具有挑战性的编程任务。每个模块的 `main()` 方法至少应该调用模块中的其他代码并在某种程度上保

26

证它的正确性。随着模块的成熟，我们可以将 `main()` 方法作为一个开发用例，在开发过程中用它来测试更多的细节；也可以把它编成一个测试用例来对所有代码进行全面的测试。当用例越来越复杂时，我们可能会将它独立成一个模块。在本书中，我们用 `main()` 来说明模块的功能并将测试用例留做练习。

### 1.1.6.8 外部库

我们会使用来自 4 个不同类型的库中的静态方法，重用每种库代码的方式都稍有不同。它们大多都是静态方法库，但也有部分是数据类型的定义并包含了一些静态方法。

- 系统标准库 `java.lang.*`: 这其中包括 `Math` 库，实现了常用的数学函数；`Integer` 和 `Double` 库，能够将字符串转化为 `int` 和 `double` 值；`String` 和 `StringBuilder` 库，我们稍后会在本节和第 5 章中详细讨论；以及其他一些我们没有用到的库。
- 导入的系统库，例如 `java.util.Arrays`: 每个标准的 Java 版本中都含有上千个这种类型的库，不过本书中我们用到的并不多。要在程序的开头使用 `import` 语句导入才能使用这些库（我们也是这样做的）。
- 本书中的其他库：例如，其他程序也可以使用 `BinarySearch` 的 `rank()` 方法。要使用这些库，请在本书的网站上下载它们的源代码并放入你的工作目录中。
- 我们为本书（以及我们的另一本入门教材 *An Introduction to programming in Java: An Interdisciplinary Approach*）开发的标准库 `Std*`: 我们会在下面简要地介绍这些库，它们的源代码和使用方法都能够在本书的网站上找到。

要调用另一个库中的方法（存放在相同或者指定的目录中，或是一个系统标准库，或是在类定义前用 `import` 语句导入的库），我们需要在方法前指定库的名称。例如，`BinarySearch` 的 `main()` 方法调用了系统库 `java.util.Arrays` 的 `sort()` 方法，我们的库 `StdIn` 中的 `readInts()` 方法和 `StdOut` 库中的 `println()` 方法。

我们自己及他人使用模块化方式编写的方法库能够极大地扩展我们的编程模型。除了在 Java 的标准版本中可用的所有库之外，网上还有成千上万各种用途的代码库。为了将我们的编程模型限制在一个可控范围之内，以将精力集中在算法上，我们只会使用以下所示的方法库，并在 1.1.7 节中列出了其中的部分方法。

27

### 1.1.7 API

模块化编程的一个重要组成部分就是记录库方法的用法并供其他人参考的文档。我们会统一使用应用程序编程接口（API）的方式列出本书中使用的每个库方法名称、签名和简短的描述。我们用用例来指代调用另一个库中的方法的程序，用实现描述实现了某个 API 方法的 Java 代码。

#### 1.1.7.1 举例

在表 1.1.6 的例子中，我们用 `java.lang` 中 `Math` 库常用的静态方法说明 API 的文档格式。

这些方法实现了各种数学函数——它们通过参数计算得到某种类型的值（`random()` 除外，它没有对应的数学函数，因为它不接受参数）。它们的参数都是 `double` 类型且返回值也都是 `double` 类型，因此可以将它们看做 `double` 数据类型的扩展——这种扩展的能力正是现代编程语言的特性

系统标准库	<code>Math</code>
	<code>Integer</code> <sup>†</sup>
	<code>Double</code> <sup>†</sup>
	<code>String</code> <sup>†</sup>
	<code>StringBuilder</code>
	<code>System</code>
导入的系统库	<code>java.util.Arrays</code>
我们的标准库	<code>StdIn</code>
	<code>StdOut</code>
	<code>StdDraw</code>
	<code>StdRandom</code>
	<code>StdStats</code>
	<code>In</code> <sup>†</sup>
	<code>Out</code> <sup>†</sup>

<sup>†</sup> 含有静态方法的数据类型的定义

本书使用的含有静态方法的库

之一。API 中的每一行描述了一个方法，提供了使用该方法所需要知道的所有信息。Math 库也定义了常数 PI（圆周率  $\pi$ ）和 E（自然对数 e），你可以在自己的程序中通过这些变量名引用它们。例如，`Math.sin(Math.PI/2)` 的结果是 1.0，`Math.log(Math.E)` 的结果也是 1.0（因为 `Math.sin()` 的参数是弧度而 `Math.log()` 使用的是自然对数函数）。

表 1.1.6 Java 的数学函数库的 API (节选)

<code>public class Math</code>	
<code>static double abs(double a)</code>	$a$ 的绝对值
<code>static double max(double a, double b)</code>	$a$ 和 $b$ 中的较大者
<code>static double min(double a, double b)</code>	$a$ 和 $b$ 中的较小者
注 1: <code>abs()</code> 、 <code>max()</code> 和 <code>min()</code> 也定义了 <code>int</code> 、 <code>long</code> 和 <code>float</code> 的版本。	
<code>static double sin(double theta)</code>	正弦函数
<code>static double cos(double theta)</code>	余弦函数
<code>static double tan(double theta)</code>	正切函数
注 2: 角用弧度表示，可以使用 <code>toDegrees()</code> 和 <code>toRadians()</code> 转换角度和弧度。	
注 3: 它们的反函数分别为 <code>asin()</code> 、 <code>acos()</code> 和 <code>atan()</code> 。	
<code>static double exp(double a)</code>	指数函数 ( $e^a$ )
<code>static double log(double a)</code>	自然对数函数 ( $\log_a$ , 即 $\ln a$ )
<code>static double pow(double a, double b)</code>	求 $a$ 的 $b$ 次方 ( $a^b$ )
<code>static double random()</code>	[0, 1) 之间的随机数
<code>static double sqrt(double a)</code>	$a$ 的平方根
<code>static double E</code>	常数 e (常数)
<code>static double PI</code>	常数 $\pi$ (常数)

其他函数请见本书的网站。

28

### 1.1.7.2 Java 库

成千上万个库的在线文档是 Java 发布版本的一部分。为了更好地描述我们的编程模型，我们只是从中节选了本书所用到的若干方法。例如，BinarySearch 中用到了 Java 的 Arrays 库中的 `sort()` 方法，我们对它的记录如表 1.1.7 所示。

表 1.1.7 Java 的 Arrays 库节选 (java.util.Arrays)

<code>public class Arrays</code>	
<code>static void sort(int[] a)</code>	将数组按升序排序

注：其他原始类型和 `Object` 对象也有对应版本的方法。

29

Arrays 库不在 `java.lang` 中，因此我们需要用 `import` 语句导入后才能使用它，与 BinarySearch 中一样。事实上，本书的第 2 章讲的正是数组的各种 `sort()` 方法的实现，包括 `Arrays.sort()` 中实现的归并排序和快速排序算法。Java 和很多其他编程语言都实现了本书讲解的许多基础算法。例如，Arrays 库还包含了二分查找的实现。为避免混淆，我们一般会使用自己的实现，但对于你已经掌握的算法使用高度优化的库实现当然也没有任何问题。

### 1.1.7.3 我们的标准库

为了介绍 Java 编程、为了科学计算以及算法的开发、学习和应用，我们也开发了若干库来提供一些实用的功能。这些库大多用于处理输入输出。我们也会使用以下两个库来测试和分析我们的实

现。第一个库扩展了 `Math.random()` 方法（见表 1.1.8），以根据不同的概率密度函数得到随机值；第二个库则支持各种统计计算（见表 1.1.9）。

表 1.1.8 我们的随机数静态方法库的 API

<code>public class StdRandom</code>	
<code>static void initialize(long seed)</code>	初始化
<code>static double random()</code>	0 到 1 之间的实数
<code>static int uniform(int N)</code>	0 到 N-1 之间的整数
<code>static int uniform(int lo, int hi)</code>	lo 到 hi-1 之间的整数
<code>static double uniform(double lo, double hi)</code>	lo 到 hi 之间的实数
<code>static boolean bernoulli(double p)</code>	返回真的概率为 p
<code>static double gaussian()</code>	正态分布，期望值为 0，标准差为 1
<code>static double gaussian(double m, double s)</code>	正态分布，期望值为 m，标准差为 s
<code>static int discrete(double[] a)</code>	返回 i 的概率为 a[i]
<code>static void shuffle(double[] a)</code>	将数组 a 随机排序

注：库中也包含为其他原始类型和 `Object` 对象重载的 `shuffle()` 函数。

表 1.1.9 我们的数据分析静态方法库的 API

<code>public class StdStats</code>	
<code>static double max(double[] a)</code>	最大值
<code>static double min(double[] a)</code>	最小值
<code>static double mean(double[] a)</code>	平均值
<code>static double var(double[] a)</code>	采样方差
<code>static double stddev(double[] a)</code>	采样标准差
<code>static double median(double[] a)</code>	中位数

30

`StdRandom` 的 `initialize()` 方法为随机数生成器提供种子，这样我们就可以重复和随机数有关的实验。以上一些方法的实现请参考表 1.1.10。有些方法的实现非常简单，为什么还要在方法库中实现它们？设计良好的方法库对这个问题的标准回答如下。

- 这些方法所实现的抽象层有助于我们将精力集中在实现和测试本书中的算法，而非生成随机数或是统计计算。每次都自己写完成相同计算的代码，不如直接在用例中调用它们要更简洁易懂。
- 方法库会经过大量测试，覆盖极端和罕见的情况，是我们可以信任的。这样的实现需要大量的代码。例如，我们经常需要使用的各种数据类型的实现，又比如 Java 的 `Arrays` 库针对不同数据类型对 `sort()` 进行了多次重载。

这些是 Java 模块化编程的基础，不过在这里可能有些夸张。但这些方法库的方法名称简单、实现容易，其中一些仍然能作为有趣的算法练习。因此，我们建议你到本书的网站上去学习一下 `StdRandom.java` 和 `StdStats.java` 的源代码并好好利用这些经过验证了的实现。使用这些库（以及检验它们）最简单的方法就是从网站上下载它们的源代码并放入你的工作目录。网站上讲解了在各种系统上使用它们的配置目录的方法。

表 1.1.10 StdRandom 库中的静态方法的实现

期望的结果	实 现
随机返回 $[a, b]$ 之间的一个 double 值	<pre>public static double uniform(double a, double b) {   return a + StdRandom.random() * (b-a); }</pre>
随机返回 $[0..N]$ 之间的一个 int 值	<pre>public static int uniform(int N) {   return (int) (StdRandom.random() * N); }</pre>
随机返回 $[lo, hi]$ 之间的一个 int 值	<pre>public static int uniform(int lo, int hi) {   return lo + StdRandom.uniform(hi - lo); }</pre>
根据离散概率随机返回的 int 值 (出现 $i$ 的概率为 $a[i]$ )	<pre>public static int discrete(double[] a) {   // a[] 中各元素之和必须等于 1     double r = StdRandom.random();     double sum = 0.0;     for (int i = 0; i &lt; a.length; i++)     {         sum = sum + a[i];         if (sum &gt;= r) return i;     }     return -1; }</pre>
随机将 double 数组中的元素排序 (请见练习 1.1.36)	<pre>public static void shuffle(double[] a) {     int N = a.length;     for (int i = 0; i &lt; N; i++)     {         // 将 a[i] 和 a[i..N-1] 中任意一个元素交换         int r = i + StdRandom.uniform(N-i);         double temp = a[i];         a[i] = a[r];         a[r] = temp;     } }</pre>

#### 1.1.7.4 你自己编写的库

你应该将自己编写的每一个程序都做一个日后可以重用的库。

- 编写用例，在实现中将计算过程分解成可控的部分。
- 明确静态方法库和与之对应的 API (或者多个库的多个 API)。
- 实现 API 和一个能够对方法进行独立测试的 main() 函数。

这种方法不仅能帮助你实现可重用的代码，而且能够教会你如何运用模块化编程来解决一个复杂的问题。

API 的目的是将调用和实现分离：除了 API 中给出的信息，调用者不需要知道实现的其他细节，而实现也不应考虑特殊的应用场景。API 使我们能够广泛地重用那些为各种目的独立开发的代码。没有任何一个 Java 库能够包含我们在程序中可能用到的所有方法，因此这种能力对于编写复杂的应用程序特别重要。相应地，程序员也可以将 API 看做调用和实现之间的一份契约，它详细说明了每个方法的作用。实现的目标就是能够遵守这份契约。一般来说，做到这一点有很多种方法，而且将调用者的代码和实现的代码分离使我们可以将老算法替换为更新更好的实现。在学习算法的过程中，这也使我们能够感受到算法的改进所带来的影响。

31  
32

33

### 1.1.8 字符串

字符串是由一串字符（`char`类型的值）组成的。一个`String`类型的字面量包括一对双引号和其中的字符，比如“Hello, World”。`String`类型是Java的一个数据类型，但并不是原始数据类型。我们现在就讨论`String`类型是因为它非常基础，几乎所有Java程序都会用到它。

#### 1.1.8.1 字符串拼接

和各种原始数据类型一样，Java内置了一个串联`String`类型字符串的运算符（+）。表1.1.11是对表1.1.2的补充。拼接两个`String`类型的字符串将得到一个新的`String`值，其中第一个字符串在前，第二个字符串在后。

表1.1.11 Java的`String`数据类型

类 型	值 域	举 例	运 算 符	表达式举例	
				表 达 式	值
<code>String</code>	一串字符	"AB" "Hello" "2.5"	+ (拼接)	"Hi," + "Bob" "12" + "34" "1" + "+" + "2"	"Hi, Bob" "1234" "1+2"

#### 1.1.8.2 类型转换

字符串的两个主要用途分别是将用户从键盘输入的内容转换成相应数据类型的值以及将各种数据类型的值转化成能够在屏幕上显示的内容。Java的`String`类型为这些操作内置了相应的方法，而且`Integer`和`Double`库还包含了分别和`String`类型相互转化的静态方法（见表1.1.12）。

表1.1.12 `String`值和数字之间相互转换的API

<code>public class Integer</code>		
	<code>static int parseInt(String s)</code>	将字符串s转换为整数
	<code>static String toString(int i)</code>	将整数i转换为字符串
<code>public class Double</code>		
	<code>static double parseDouble(String s)</code>	将字符串s转换为浮点数
	<code>static String toString(double x)</code>	将浮点数x转换为字符串

#### 1.1.8.3 自动转换

我们很少明确使用刚才提到的`toString()`方法，因为Java在连接字符串的时候会自动将任意数据类型的值转换为字符串：如果加号（+）的一个参数是字符串，那么Java会自动将其他参数都转换为字符串（如果它们不是的话）。除了像“`The square root of 2.0 is` ”+`Math.sqrt(2.0)`这样的使用方式之外，这种机制也使我们能够通过一个空字符串“”将任意数据类型的值转换为字符串值。

#### 1.1.8.4 命令行参数

在Java中字符串的一个重要的用途就是使程序能够接收到从命令行传递来的信息。这种机制很简单。当你输入命令`java`和一个库名以及一系列字符串之后，Java系统会调用库的`main()`方法并将那一系列字符串变成一个数组作为参数传递给它。例如，`BinarySearch`的`main()`方法需要一个命令行参数，因此系统会创建一个大小为1的数组。程序用这个值，也就是`args[0]`，来获取白

名单文件的文件名并将其作为 `StdIn.readInts()` 的参数。另一种在我们的代码中常见的用法是当命令行参数表示的是数字时，我们会用 `parseInt()` 和 `parseDouble()` 方法将其分别转换为整数和浮点数。

字符串的用法是现代程序中的重要部分。现在我们还只是用 `String` 在外部表示为字符串的数字和内部表示为数字类数据类型的值进行转换。在 1.2 节中我们会看到 Java 为我们提供了非常丰富的字符串操作；在 1.4 节中我们会分析 `String` 类型在 Java 内部的表示方法；在第 5 章我们会深入学习处理字符串的各种算法。这些算法是本书中最有趣、最复杂也是影响力最大的一部分算法。

35

### 1.1.9 输入输出

我们的标准输入、输出和绘图库的作用是建立一个 Java 程序和外界交流的简易模型。这些库的基础是强大的 Java 标准库，但它们一般更加复杂，学习和使用起来都更加困难。我们先来简单地了解一下这个模型。

在我们的模型中，Java 程序可以从命令行参数或者一个名为标准输入流的抽象字符流中获得输入，并将输出写入另一个名为标准输出流的字符流中。

我们需要考虑 Java 和操作系统之间的接口，因此我们要简要地讨论一下大多数操作系统和程序开发环境所提供的相应机制。本书网站上列出了关于你所使用的系统的更多信息。默认情况下，命令行参数、标准输入和标准输出是和应用程序绑定的，而应用程序是由能够接受命令输入的操作系统或是开发环境所支持。我们笼统地用终端来指代这个应用程序提供的供输入和显示的窗口。20 世纪 70 年代早期的 Unix 系统已经证明我们可以用这个模型方便直接地和程序以及数据进行交互。我们在经典的模型中加入了一个标准绘图模块用来可视化表示对数据的分析，如图 1.1.3 所示。

#### 1.1.9.1 命令和参数

终端窗口包含一个提示符，通过它我们能够向操作系统输入命令和参数。本书中我们只会用到几个命令，如表 1.1.13 所示。我们会经常使用 `java` 命令来运行我们的程序。我们在 1.1.8.4 节中提到过，Java 类都会包含一个静态方法 `main()`，它有一个 `String` 数组类型的参数 `args[]`。这个数组的内容就是我们输入的命令行参数，操作系统会将它传递给 Java。Java 和操作系统都默认参数为字符串。如果我们需要的某个参数是数字，我们会使用类似 `Integer.parseInt()` 的方法将其转换为适当的数据类型的值。图 1.1.4 是对命令的分析。

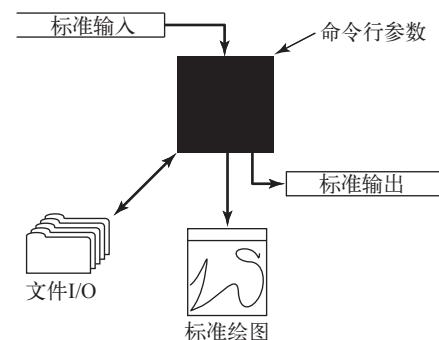


图 1.1.3 Java 程序整体结构

表 1.1.13 操作系统常用命令

命    令	参    数	作    用
<code>javac</code>	.java 文件名	编译 Java 程序
<code>java</code>	.class 文件名（不需要扩展名） 和命令行参数	运行 Java 程序
<code>more</code>	任意文本文件名	打印文件内容

36

### 1.1.9.2 标准输出

我们的 StdOut 库的作用是支持标准输出。一般来说，系统会将标准输出打印到终端窗口。`print()` 方法会将它的参数放到标准输出中；`println()` 方法会附加一个换行符；`printf()` 方法能够格式化输出（见 1.1.9.3 节）。Java 在其 `System.out` 库中提供了类似的方法，但我们会用 StdOut 库来统一处理标准输入和输出（并进行了一些技术上的改进），见表 1.1.4。

表 1.1.14 我们的标准输出库的静态方法的 API

public class StdOut	
static void print(String s)	打印 s
static void println(String s)	打印 s 并接一个换行符
static void println()	打印一个换行符
static void printf(String f, ...)	格式化输出

注：其他原始类型和 `Object` 对象也有对应版本的方法。

要使用这些方法，请从本书的网站上将 `StdOut.java` 下载到你的工作目录，并像 `StdOut.println("Hello, World")`；这样在代码中调用它们。左下方的程序就是一个例子。

### 1.1.9.3 格式化输出

在最简单的情况下 `printf()` 方法接受两个参数。第一个参数是一个格式字符串，描述了第二个参数应该如何在输出中被转换为一个字符串。最简单的格式字符串的第一个字符是 % 并紧跟一个字符表示的转换代码。我们最常使用的转换代码包括 `d`（用于 Java 整型的十进制数）、`f`（浮点型）

```
public class RandomSeq
{
    public static void main(String[] args)
    { // 打印N个(lo, hi)之间的随机值
        int N = Integer.parseInt(args[0]);
        double lo = Double.parseDouble(args[1]);
        double hi = Double.parseDouble(args[2]);
        for (int i = 0; i < N; i++)
        {
            double x = StdRandom.uniform(lo, hi);
            StdOut.printf("%.2f\n", x);
        }
    }
}
```

StdOut 的用例示例

```
% java RandomSeq 5 100.0 200.0
123.43
153.13
144.38
155.18
104.02
```

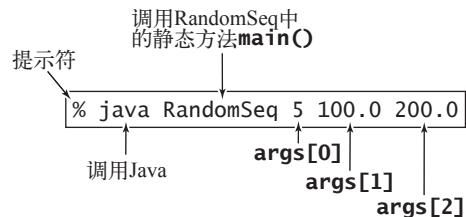


图 1.1.4 命令详解

和 `s`（字符串）。在 % 和转换代码之间可以插入一个整数来表示转换之后的值的宽度，即输出字符串的长度。默认情况下，转换后会在字符串的左边添加空格以达到需要的宽度，如果我们想在右边加入空格则应该使用负宽度（如果转换得到的字符串比设定宽度要长，宽度会被忽略）。在宽度之后我们还可以插入一个小数点以及一个数值来指定转换后的 `double` 值保留的小数位数（精度）或是 `String` 字符串所截取的长度。使用

`printf()` 方法时需要记住最重要的一点就是，格式字符串中的转换代码和对应参数的数据类型必须匹配。也就是说，Java 要求参数的数据类型和转换代码表示的数据类型必须相同。`printf()` 的第一个 `String` 字符串参数也可以包含其他字符。所有非格式字符串的

字符都会被传递到输出之中，而格式字符串则会被参数的值所替代（按照指定的方式转换为字符串）。例如，这条语句：

```
StdOut.printf("PI is approximately %.2f\n", Math.PI);
```

会打印出：

```
PI is approximately 3.14
```

可以看到，在 `printf()` 中我们需要明确地在第一个参数的末尾加上 `\n` 来换行。`printf()` 函数能够接受两个或者更多的参数。在这种情况下，在格式化字符串中每个参数都会有对应的转换代码，这些代码之间可能隔着其他会被直接传递到输出中的字符。也可以直接使用静态方法 `String.format()` 来用和 `printf()` 相同的参数得到一个格式化字符串而无需打印它。我们可以用格式化打印方便地将实验数据输出为表格形式（这是它们在本书中的主要用途），如表 1.1.15 所示。

表 1.1.15 `printf()` 的格式化方式（更多选项请见本书网站）

数据类型	转换代码	举 例	格式化字符串举例	转换后输出的字符串
<b>int</b>	d	512	"%14d" "%-14d"	" 512" "- 512"
<b>double</b>	f e	1595.1680010754388	"%14.2f" "%.7f" "%14.4e"	" 1595.17" "1595.1680011" " 1.5952e+03"
<b>String</b>	s	"Hello, World"	"%14s" "%-14s" "%14.5s"	" Hello, World" "Hello, World " "Hello "

38

#### 1.1.9.4 标准输入

我们的 `StdIn` 库从标准输入流中获取数据，这些数据可能为空也可能是一系列由空白字符分隔的值（空格、制表符、换行符等）。默认状态下系统会将标准输出定向到终端窗口——你输入的内容就是输入流（由 `<ctrl-d>` 或 `<ctrl-z>` 结束，取决于你使用的终端应用程序）。这些值可能是 `String` 或是 Java 的某种原始类型的数据。标准输入流最重要的特点是这些值会在你的程序读取它们之后消失。只要程序读

取了一个值，它就不能回退并再次读取它。这个特点产生了一些限制，但它反映了一些输入设备的物理特性并简化了对这些设备的抽象。有了输入流模型，这个库中的静态方法大都是自文档化的（它们的签名即说明了它们的用途）。右侧列出了 `StdIn` 的一个用例。

表 1.1.16 详细说明了标准输入库中的静态方法的 API。

```
public class Average
{
    public static void main(String[] args)
    { // 取StdIn中所有数的平均值
        double sum = 0.0;
        int cnt = 0;
        while (!StdIn.isEmpty())
        { // 读取一个数并计算累计之和
            sum += StdIn.readDouble();
            cnt++;
        }
        double avg = sum / cnt;
        StdOut.printf("Average is %.5f\n", avg);
    }
}
```

StdIn 的用例举例

```
% java Average
1.23456
2.34567
3.45678
4.56789
<ctrl-d>
Average is 2.90123
```

表 1.1.16 标准输入库中的静态方法的 API

Public class StdIn	
static boolean	isEmpty()
static int	readInt()
static double	readDouble()
static float	readFloat()
static long	readLong()
static boolean	readBoolean()
static char	readChar()
static byte	readByte()
static String	readString()
static boolean	hasNextLine()
static String	readLine()
static String	readAll()

39

### 1.1.9.5 重定向与管道

标准输入输出使我们能够利用许多操作系统都支持的命令行的扩展功能。只需要向启动程序的命令中加入一个简单的提示符，就可以将它的标准输出重定向至一个文件。文件的内容既可以永久保存也可以在之后作为另一个程序的输入：

```
% java RandomSeq 1000 100.0 200.0 > data.txt
```

这条命令指明标准输出流不是被打印至终端窗口，而是被写入一个叫做 `data.txt` 的文件。每次调用 `StdOut.print()` 或是 `StdOut.println()` 都会向该文件追加一段文本。在这个例子中，我们最后会得到一个含有 1000 个随机数的文件。终端窗口中不会出现任何输出：它们都被直接写入了“`>`”号之后的文件中。这样我们就能将信息存储以备下次使用。请注意不需要改变 `RandomSeq` 的任何内容——它使用的是标准输出的抽象，因此它不会因为我们使用了该抽象的另一种不同的实现而受到影响。类似，我们可以重定向标准输入以使 `StdIn` 从文件而不是终端应用程序中读取数据：

```
% java Average < data.txt
```

这条命令会从文件 `data.txt` 中读取一系列数值并计算它们的平均值。具体来说，“`<`”号是一个提示符，它告诉操作系统读取文本文件 `data.txt` 作为输入流而不是在终端窗口中等待用户的输入。当程序调用 `StdIn.readDouble()` 时，操作系统读取的是文件中的值。将这些结合起来，将一个程序的输出重定向为另一个程序的输入叫做管道：

```
40 % java RandomSeq 1000 100.0 200.0 | java Average
```

这条命令将 `RandomSeq` 的标准输出和 `Average` 的标准输入指定为同一个流。它的效果是好像在 `Average` 运行时 `RandomSeq` 将它生成的数字输入了终端窗口。这种差别影响非常深远，因为它突破了我们能够处理的输入输出流的长度限制。例如，即使计算机没有足够的空间来存储十亿个数，我们仍然可以将例子中的 1000 换成 1 000 000 000（当然我们还是需要一些时间来处理它们）。当 `RandomSeq` 调用 `StdOut.println()` 时，它就向输出流的末尾添加了一个字符串；当 `Average` 调用 `StdIn.readInt()` 时，它就从输入流的开头删除了一个字符串。这些动作发生的实际顺序取决于操作系统：它可能会先运行 `RandomSeq` 并产生一些输出，然后再运行 `Average`，来消耗这些输出，或者它也可以先运行 `Average`，直到它需要一些输入然后再运行 `RandomSeq` 来产生一些输出。虽然

最后的结果都一样，但我们的程序就不再需要担心这些细节，因为它们只会和标准输入和标准输出的抽象打交道。

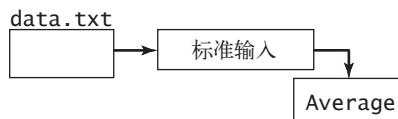
图 1.1.5 总结了重定向与管道的过程。

### 1.1.9.6 基于文件的输入输出

我们的 In 和 Out 库提供了一些静态方法，来实现向文件中写入或从文件中读取一个原始数据类型（或 String 类型）的数组的抽象。我们会使用 In 库中的 `readInts()`、`readDoubles()` 和 `readStrings()` 以及 Out 库中的 `writeInts()`、`writeDoubles()` 和 `writeStrings()` 方法，参数可以是文件或网页如表 1.1.17 所示。例如，借此我们可以在同一个程序中分别使用文件和标准输入达到两种不同的目的，例如 BinarySearch。In 和 Out 两个库也实现了一些数据类型和它们的实例方法，这使我们能够将多个文件作为输入输出流并将网页作为输入流，我们还会在 1.2 节中再次考察它们。

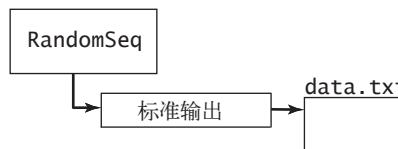
将一个文件重定向为标准输入

% java Average < data.txt



将标准输出重定向到一个文件

% java RandomSeq 1000 100.0 200.0 > data.txt



将一个程序的输出通过管道作为另一个程序的输入

% java RandomSeq 1000 100.0 200.0 | java Average

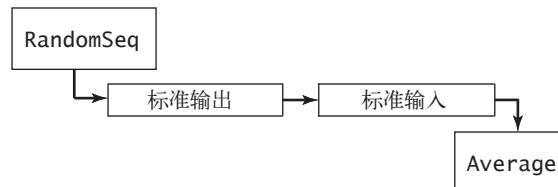


图 1.1.5 命令行的重定向与管道

表 1.1.17 我们用于读取和写入数组的静态方法的 API

<b>public class In</b>	
static int[] readInts(String name)	读取多个 int 值
static double[] readDoubles(String name)	读取多个 double 值
static String[] readStrings(String name)	读取多个 String 值
<b>public class Out</b>	
static void write(int[], String name)	写入多个 int 值
static void write(double[] a, String name)	写入多个 double 值
static void write(String[] a, String name)	写入多个 String 值

注 1：库也支持其他原始数据类型。

注 2：库也支持 StdIn 和 StdOut（忽略 name 参数）。

41

### 1.1.9.7 标准绘图库（基本方法）

目前为止，我们的输入输出抽象层的重点只有文本字符串。现在我们要介绍一个产生图像输出的抽象层。这个库的使用非常简单并且允许我们利用可视化的方式处理比文字丰富得多的信息。和我们的标准输入输出一样，标准绘图抽象层实现在库 `StdDraw` 中，可以从本书的网站上下载 `StdDraw.java` 到你的工作目录来使用它。标准绘图库很简单：我们可以将它想象为一个抽象的能够在二维画布上画出点和直线的绘图设备。这个设备能够根据程序调用的 `StdDraw` 中的静态方法画出

一些基本的几何图形，这些方法包括画出点、直线、文本字符串、圆、长方形和多边形等。和标准输入输出中的方法一样，这些方法几乎也都是自文档化的：`StdDraw.line()` 能够根据参数的坐标画出一条连接点 $(x_0, y_0)$ 和点 $(x_1, y_1)$ 的线段，`StdDraw.point()`能够根据参数坐标画出一个以 $(x, y)$ 为中心的点，等等，如图 1.1.6 所示。几何图形可以被填充（默认为黑色）。默认的比例尺为单位正方形（所有的坐标均在 0 和 1 之间）。标准的实现会将画布显示为屏幕上的一个窗口，点和线为黑色，背景为白色。

42

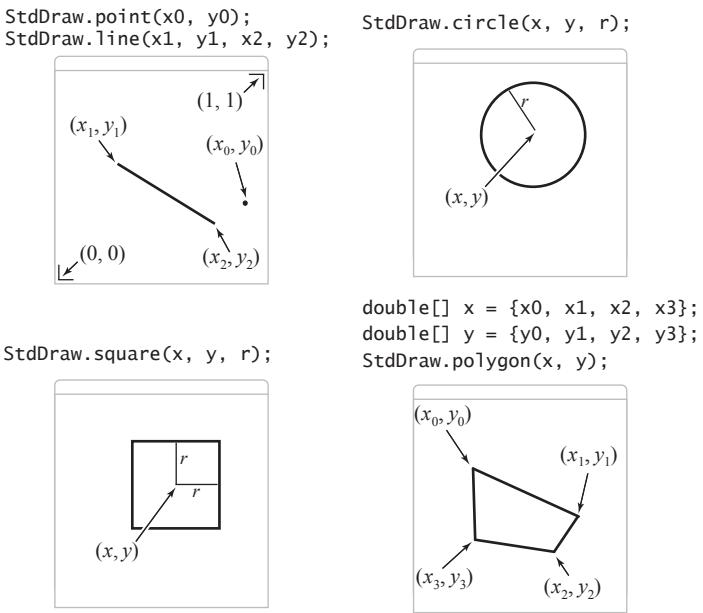


图 1.1.6 StdDraw 的用法举例

表 1.1.18 是对标准绘图库中静态方法 API 的汇总。

表 1.1.18 标准绘图库的静态（绘图）方法的 API

---

```

public class StdDraw
{
    static void line(double x0, double y0, double x1, double y1)
    static void point(double x, double y)
    static void text(double x, double y, String s)
    static void circle(double x, double y, double r)
    static void filledCircle(double x, double y, double r)
    static void ellipse(double x, double y, double rw, double rh)
    static void filledEllipse(double x, double y, double rw, double rh)
    static void square(double x, double y, double r)
    static void filledSquare(double x, double y, double r)
    static void rectangle(double x, double y, double rw, double rh)
    static void filledRectangle(double x, double y, double rw, double rh)
    static void polygon(double[] x, double[] y)
    static void filledPolygon(double[] x, double[] y)
}

```

---

### 1.1.9.8 标准绘图库（控制方法）

标准绘图库中还包含一些方法来改变画布的大小和比例、直线的颜色和宽度、文本字体、绘图时间（用于动画）等。可以使用在 `StdDraw` 中预定义的 `BLACK`、`BLUE`、`CYAN`、`DARK_GRAY`、`GRAY`、`GREEN`、`LIGHT_GRAY`、`MAGENTA`、`ORANGE`、`PINK`、`RED`、`BOOK_RED`、`WHITE` 和 `YELLOW` 等颜色常数作为 `setPenColor()` 方法的参数（可以用 `StdDraw.RED` 这样的方式调用它们）。画布窗口的菜单还包含一个选项用于将图像保存为适于在网上传播的文件格式。表 1.1.19 总结了 `StdDraw` 中静态控制方法的 API。

表 1.1.19 标准绘图库的静态（控制）方法的 API

<code>public class StdDraw</code>	
<code>static void setXscale(double x0, double x1)</code>	将 $x$ 的范围设为 $(x_0, x_1)$
<code>static void setYscale(double y0, double y1)</code>	将 $y$ 的范围设为 $(y_0, y_1)$
<code>static void setPenRadius(double r)</code>	将画笔的粗细半径设为 $r$
<code>static void setPenColor(Color c)</code>	将画笔的颜色设为 $c$
<code>static void setFont(Font f)</code>	将文本字体设为 $f$
<code>static void setCanvasSize(int w, int h)</code>	将画布窗口的宽和高分别设为 $w$ 和 $h$
<code>static void clear(Color c)</code>	清空画布并用颜色 $c$ 将其填充
<code>static void show(int dt)</code>	显示所有图像并暂停 $dt$ 毫秒

43

在本书中，我们会在数据分析和算法的可视化中使用 `StdDraw`。表 1.1.20 是一些例子，我们在本书的其他章节和练习中还会遇到更多的例子。绘图库也支持动画——当然，这个话题只能在本书的网站上展开了。

44

表 1.1.20 `StdDraw` 绘图举例

数 据	绘图的实现（代码片段）	结 果
函数值	<pre>int N = 100; StdDraw.setXscale(0, N); StdDraw.setYscale(0, N*N); StdDraw.setPenRadius(.01); for (int i = 1; i &lt;= N; i++) {     StdDraw.point(i, i);     StdDraw.point(i, i*i);     StdDraw.point(i, i*Math.log(i)); }</pre>	
随机数组	<pre>int N = 50; double[] a = new double[N]; for (int i = 0; i &lt; N; i++)     a[i] = StdRandom.random(); for (int i = 0; i &lt; N; i++) {     double x = 1.0*i/N;     double y = a[i]/2.0;     double rw = 0.5/N;     double rh = a[i]/2.0;     StdDraw.filledRectangle(x, y, rw, rh); }</pre>	
已排序的随机数组	<pre>int N = 50; double[] a = new double[N]; for (int i = 0; i &lt; N; i++)     a[i] = StdRandom.random(); Arrays.sort(a); for (int i = 0; i &lt; N; i++) {     double x = 1.0*i/N;     double y = a[i]/2.0;     double rw = 0.5/N;     double rh = a[i]/2.0;     StdDraw.filledRectangle(x, y, rw, rh); }</pre>	

45

### 1.1.10 二分查找

我们要学习的第一个 Java 程序的示例程序就是著名、高效并且应用广泛的二分查找算法，如下所示。这个例子将会展示本书中学习新算法的基本方法。和我们将要学习的所有程序一样，它既是算法的准确定义，又是算法的一个完整的 Java 实现，而且你还能够从本书的网站上下载它。

#### 二分查找

```
import java.util.Arrays;
public class BinarySearch
{
    public static int rank(int key, int[] a)
    { // 数组必须是有序的
        int lo = 0;
        int hi = a.length - 1;
        while (lo <= hi)
        { // 被查找的键要么不存在，要么必然存在于 a[lo..hi] 之中
            int mid = lo + (hi - lo) / 2;
            if (key < a[mid]) hi = mid - 1;
            else if (key > a[mid]) lo = mid + 1;
            else return mid;
        }
        return -1;
    }
    public static void main(String[] args)
    {
        int[] whitelist = In.readInts(args[0]);
        Arrays.sort(whitelist);
        while (!StdIn.isEmpty())
        { // 读取键值，如果不存在于白名单中则将其打印
            int key = StdIn.readInt();
            if (rank(key, whitelist) < 0)
                StdOut.println(key);
        }
    }
}
```

这段程序接受一个白名单文件（一列整数）作为参数，并会过滤掉标准输入中的所有存在于白名单中的条目，仅将不在白名单上的整数打印到标准输出中。它在 `rank()` 静态方法中实现了二分查找算法并高效地完成了这个任务。

关于二分查找算法的完整讨论，包括它的正确性、性能分析及其应用，请见 3.1 节。

```
% java BinarySearch tinyW.txt < tinyT.txt
50
99
13
```

#### 1.1.10.1 二分查找

我们会在 3.2 节中详细学习二分查找算法，但此处先简单地描述一下。算法是由静态方法 `rank()` 实现的，它接受一个整数键和一个已经有序的 `int` 数组作为参数。如果该键存在于数组中则返回它的索引，否则返回 `-1`。算法使用两个变量 `lo` 和 `hi`，并保证如果键在数组中则它一定在 `a[lo..hi]` 中，然后方法进入一个循环，不断将数组的中间键（索引为 `mid`）和被查找的键比较。如果被查找的键等于 `a[mid]`，返回 `mid`；否则算法就将查找范围缩小一半，如果被查找的键小于 `a[mid]` 就继续在左半边查找，如果被查找的键大于 `a[mid]` 就继续在右半边查找。算法找到被查找的键或是查找范围为空时该过程结束。二分查找之所以快是因为它只需检查很少几个条目（相对于数组的大小）就能够找到目标元素（或者确认目标元素不存在）。在有序数组中进行二分查找的示

例如图 1.1.7 所示。

### 1.1.10.2 开发用例

对于每个算法的实现，我们都会开发一个用例 `main()` 函数，并在书中或是本书的网站上提供一个示例输入文件来帮助读者学习该算法并检测它的性能。在这个例子中，这个用例会从命令行指定的文件中读取多个整数，并会打印出标准输入中所有不存在于该文件中的整数。我们使用了图 1.1.8 所示的几个较小的测试文件来展示它的行为，这些文件也是图 1.1.7 中的跟踪和例子的基础。我们会使用较大的测试文件来模拟真实应用并测试算法的性能（请见 1.1.10.3 节）。

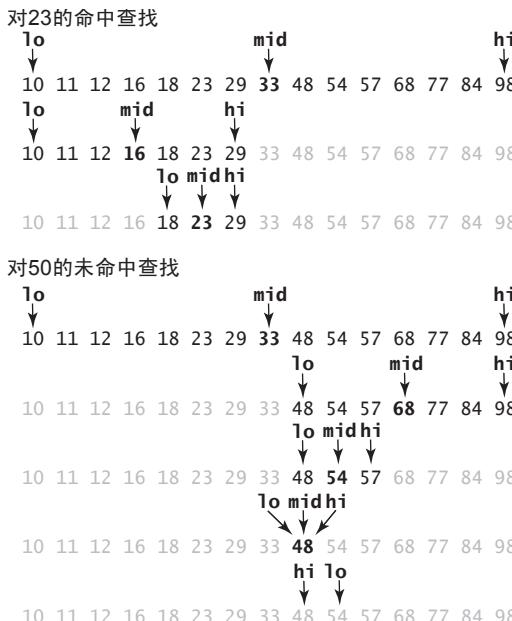


图 1.1.7 有序数组中的二分查找

<code>tinyW.txt</code>	<code>tinyT.txt</code>
84	23
48	50
68	10
10	99
18	18
98	23
12	98
23	84
54	11
57	10
48	48
33	77
16	13
77	54
11	98
29	77
77	68

图 1.1.8 为 BinarySearch 的测试用例准备的小型测试文件

### 1.1.10.3 白名单过滤

如果可能，我们的测试用例都会通过模拟实际情况来展示当前算法的必要性。这里该过程被称为白名单过滤。具体来说，可以想象一家信用卡公司，它需要检查客户的交易账号是否有效。为此，它需要：

- 将客户的账号保存在一个文件中，我们称它为白名单；
- 从标准输入中得到每笔交易的账号；
- 使用这个测试用例在标准输出中打印所有与任何客户无关的账号，公司很可能拒绝此类交易。

在一家有上百万客户的大公司中，需要处理数百万甚至更多的交易都是很正常的。为了模拟这种情况，我们在本书的网站上提供了文件 `largeW.txt` (100 万个整数) 和 `largeT.txt` (1000 万个整数) 其基本情况如图 1.1.9 所示。

### 1.1.10.4 性能

一个程序只是可用往往是不够的。例如，以下 `rank()` 的实现也可以很简单，它会检查数组的每个元素，甚至都不需要数组是有序的：

```
public static int rank(int key, int[] a)
{
    for (int i = 0; i < a.length; i++)
        if (a[i] == key) return i;
    return -1;
}
```

有了这个简单易懂的解决方案，我们为什么还需要归并排序和二分查找呢？你在完成练习 1.1.38 时会看到，计算机用 `rank()` 方法的暴力实现处理大量输入（比如含有 100 万个条目的白名单和 1000 万条交易）非常慢。没有如二分查找或者归并排序这样的高效算法，解决大规模的白名单问题是不可能的。良好的性能常常是极为重要的，因此我们会在 1.4 节中为性能研究做一些铺垫，并会分析我们学习的所有算法的性能特点（包括 2.2 节的归并排序和 3.1 节中的二分查找）。

目前，我们在这里粗略地勾勒出我们的编程模型的目标是，确保你能够在计算机上运行类似于 `BinarySearch` 的代码，使用它处理我们的测试数据并为适应各种情况修改它（比如本节练习中所描述的一些情况）以完全理解它的可应用性。我们的编程模型就是设计用来简化这些活动的，这对各种算法的学习至关重要。

48  
49

### 1.1.11 展望

在本节中，我们描述了一个精巧而完整的编程模型，数十年来它一直在（并且现在仍在）为广大程序员服务。但现代编程技术已经更进一步。前进的这一步被称为数据抽象，有时也被称为面向对象编程，它是我们下一节的主题。简单地说，数据抽象的主要思想是鼓励程序定义自己的数据类型（一系列值和对这些值的操作），而不仅仅是那些操作预定义的数据类型的静态方法。

面向对象编程在最近几十年得到了广泛的应用，数据抽象已经成为现代程序开发的核心。我们在本书中“拥抱”数据抽象的原因主要有三。

- 它允许我们通过模块化编程复用代码。例如，第 2 章中的排序算法和第 3 章中的二分查找以及其他算法，都允许调用者用同一段代码处理任意类型的数据（而不仅限于整数），包括调用者自定义的数据类型。
- 它使我们可以轻易构造多种所谓的链式数据结构，它们比数组更灵活，在许多情况下都是高效算法的基础。
- 借助它我们可以准确地定义所面对的算法问题。比如 1.5 节中的 union-find 算法、2.4 节中的优先队列算法和第 3 章中的符号表算法，它们解决问题的方式都是定义数据结构并高效地实现它们的一组操作。这些问题都能够用数据抽象很好地解决。

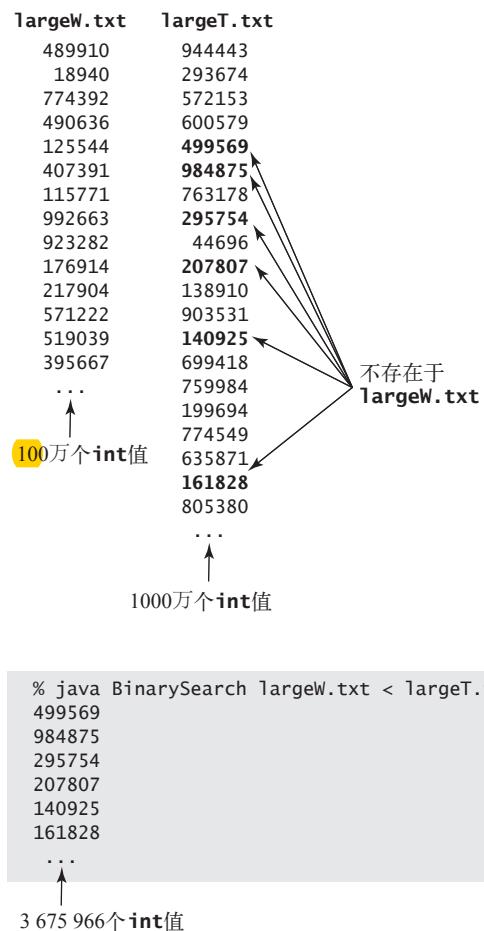


图 1.1.9 为 `BinarySearch` 测试用例准备的大型文件

尽管如此，但我们的重点仍然是对算法的研究。在了解了这些知识以后，我们将学习面向对象编程中和我们的使命相关的另一个重要特性。

50

## 答疑

- 问 什么是 Java 的字节码？  
 答 它是程序的一种低级表示，可以运行于 Java 的虚拟机。将程序抽象为字节码可以保证 Java 程序员的代码能够运行在各种设备之上。
- 问 Java 允许整型溢出并返回错误值的做法是错误的。难道 Java 不应该自动检查溢出吗？  
 答 这个问题在程序员中一直是有争议的。简单的回答是它们之所以被称为原始数据类型就是因为缺乏此类检查。避免此类问题并不需要很高深的知识。我们会使用 `int` 类型表示较小的数（小于 10 个十进制位）而使用 `long` 表示 10 亿以上的数。
- 问 `Math.abs(-2147483648)` 的返回值是什么？  
 答 `-2147483648`。这个奇怪的结果（但的确是真的）就是整数溢出的典型例子。
- 问 如何才能将一个 `double` 变量初始化为无穷大？  
 答 可以使用 Java 的内置常数：`Double.POSITIVE_INFINITY` 和 `Double.NEGATIVE_INFINITY`。
- 问 能够将 `double` 类型的值和 `int` 类型的值相互比较吗？  
 答 不通过类型转换是不行的，但请记住 Java 一般会自动进行所需的类型转换。例如，如果 `x` 的类型是 `int` 且值为 3，那么表达式 (`x<3.1`) 的值为 `true`——Java 会在比较前将 `x` 转换为 `double` 类型（因为 3.1 是一个 `double` 类型的字面量）。
- 问 如果使用一个变量前没有将它初始化，会发生什么？  
 答 如果代码中存在任何可能导致使用未经初始化的变量的执行路径，Java 都会抛出一个编译异常。
- 问 Java 表达式 `1/0` 和 `1.0/0.0` 的值是什么？  
 答 第一个表达式会产生一个运行时除零异常（它会终止程序，因为这个值是未定义的）；第二个表达式的值是 `Infinity`（无穷大）。
- 问 能够使用 `<` 和 `>` 比较 `String` 变量吗？  
 答 不行，只有原始数据类型定义了这些运算符。请见 1.1.2.3 节。
- 问 负数的除法和余数的结果是什么？  
 答 表达式 `a/b` 的商会向 0 取整；`a % b` 的余数的定义是  $(a/b)*b + a \% b$  恒等于 `a`。例如  $-14/3$  和  $14/-3$  的商都是  $-4$ ，但  $-14 \% 3$  是  $-2$ ，而  $14 \% -3$  是  $2$ 。
- 问 为什么使用 `(a && b)` 而非 `(a & b)`？  
 答 运算符 `&`、`|` 和 `^` 分别表示整数的位逻辑操作与、或和异或。因此， $10|6$  的值为  $14$ ， $10^6$  的值为  $12$ 。在本书中我们很少（偶尔）会用到这些运算符。`&&` 和 `||` 运算符仅在独立的布尔表达式中有效，原因是短路求值法则：表达式从左向右求值，一旦整个表达式的值已知则停止求值。
- 问 嵌套 `if` 语句中的二义性有问题吗？  
 答 是的。在 Java 中，以下语句：

```
if <expr1> if <expr2> <stmtA> else <stmtB>
等价于：
if <expr1> { if <expr2> <stmtA> else <stmtB> }
```

51

即使你想表达的是：

```
if <expr1> { if <expr2> <stmtA> } else <stmtB>
```

避免这种“无主的”`else`陷阱的最好办法是显式地写明所有大括号。

问 一个`for`循环和它的`while`形式有什么区别？

答 `for`循环头部的代码和`for`循环的主体代码在同一个代码段之中。在一个典型的`for`循环中，递增变量一般在循环结束之后都是不可用的；但在和它等价的`while`循环中，递增变量在循环结束之后仍然是可用的。这个区别常常是使用`while`而非`for`循环的主要原因。

问 有些Java程序员用`int a[]`而不是`int[] a`来声明一个数组。这两者有什么不同？

答 在Java中，两者等价且都是合法的。前一种是C语言中数组的声明方式。后者是Java提倡的方式，因为变量的类型`int[]`能更清楚地说明这是一个整型的数组。

问 为什么数组的起始索引是0而不是1？

答 这个习惯来源于机器语言，那时要计算一个数组元素的地址需要将数组的起始地址加上该元素的索引。将起始索引设为1要么会浪费数组的第一个元素的空间，要么会花费额外的时间来将索引减1。

问 如果`a[]`是一个数组，为什么`StdOut.println(a)`打印出的是一个十六进制的整数，比如@f62373，而不是数组中的元素呢？

答 问得好。该方法打印出的是这个数组的地址，不幸的是你一般都不需要它。

问 我们为什么不使用标准的Java库来处理输入和图形？

答 我们的确用到了它们，但我们希望使用更简单的抽象模型。`StdIn`和`StdDraw`背后的Java标准库是为实际生产设计的，这些库和它们的API都有些笨重。要想知道它们真正的模样，请查看`StdIn.java`和`StdDraw.java`的代码。

问 我的程序能够重新读取标准输入中的值吗？

答 不行，你只有一次机会，就好像你不能撤销`println()`的结果一样。

问 如果我的程序在标准输入为空之后仍然尝试读取，会发生什么？

答 会得到一个错误。`StdIn.isEmpty()`能够帮助你检查是否还有可用的输入以避免这种错误。

问 这条出错信息是什么意思？

```
Exception in thread "main" java.lang.NoClassDefFoundError: StdIn
```

答 你可能忘记把`StdIn.java`文件放到工作目录中去了。

问 在Java中，一个静态方法能够将另一个静态方法作为参数吗？

答 不行，但问得好，因为有很多语言都能够这么做。

## 练习

1.1.1 给出以下表达式的值：

- a.  $(0 + 15) / 2$
- b.  $2.0e-6 * 100000000.1$
- c. `true && false || true && true`

1.1.2 给出以下表达式的类型和值：

- a.  $(1 + 2.236)/2$
- b.  $1 + 2 + 3 + 4.0$

c. `4.1 >= 4`

d. `1 + 2 + "3"`

- 1.1.3 编写一个程序，从命令行得到三个整数参数。如果它们都相等则打印 `equal`，否则打印 `not equal`。

- 1.1.4 下列语句各有什么问题（如果有的话）？

a. `if (a > b) then c = 0;`

b. `if a > b { c = 0; }`

c. `if (a > b) c = 0;`

d. `if (a > b) c = 0 else b = 0;`

- 1.1.5 编写一段程序，如果 `double` 类型的变量 `x` 和 `y` 都严格位于 0 和 1 之间则打印 `true`，否则打印 `false`。

- 1.1.6 下面这段程序会打印出什么？

```
int f = 0;
int g = 1;
for (int i = 0; i <= 15; i++)
{
    StdOut.println(f);
    f = f + g;
    g = f - g;
}
```

54

- 1.1.7 分别给出以下代码段打印出的值：

a. `double t = 9.0;
while (Math.abs(t - 9.0/t) > .001)
 t = (9.0/t + t) / 2.0;
StdOut.printf("%.5f\n", t);`

b. `int sum = 0;
for (int i = 1; i < 1000; i++)
 for (int j = 0; j < i; j++)
 sum++;
StdOut.println(sum);`

c. `int sum = 0;
for (int i = 1; i < 1000; i *= 2)
 for (int j = 0; j < 1000; j++)
 sum++;
StdOut.println(sum);`

- 1.1.8 下列语句会打印出什么结果？给出解释。

a. `System.out.println('b');`

b. `System.out.println('b' + 'c');`

c. `System.out.println((char) ('a' + 4));`

- 1.1.9 编写一段代码，将一个正整数 `N` 用二进制表示并转换为一个 `String` 类型的值 `s`。

解答：Java 有一个内置方法 `Integer.toBinaryString(N)` 专门完成这个任务，但该题的目的就是给出这个方法的其他实现方法。下面就是一个特别简洁的答案：

```
String s = "";
for (int n = N; n > 0; n /= 2)
    s = (n % 2) + s;
```

55

## 1.1.10 下面这段代码有什么问题?

```
int[] a;
for (int i = 0; i < 10; i++)
    a[i] = i * i;
```

解答：它没有用 new 为 a[] 分配内存。这段代码会产生一个 variable a might not have been initialized 的编译错误。

## 1.1.11 编写一段代码，打印出一个二维布尔数组的内容。其中，使用 \* 表示真，空格表示假。打印出行号和列号。

## 1.1.12 以下代码段会打印出什么结果?

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
    a[i] = 9 - i;
for (int i = 0; i < 10; i++)
    a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
    System.out.println(i);
```

## 1.1.13 编写一段代码，打印出一个 M 行 N 列的二维数组的转置（交换行和列）。

1.1.14 编写一个静态方法 lg()，接受一个整型参数 N，返回不大于 log<sub>2</sub>N 的最大整数。不要使用 Math 库。

## 1.1.15 编写一个静态方法 histogram()，接受一个整型数组 a[] 和一个整数 M 为参数并返回一个大小为 M 的数组，其中第 i 个元素的值为整数 i 在参数数组中出现的次数。如果 a[] 中的值均在 0 到 M-1 之间，返回数组中所有元素之和应该和 a.length 相等。

## 1.1.16 给出 exR1(6) 的返回值：

```
public static String exR1(int n)
{
    if (n <= 0) return "";
    return exR1(n-3) + n + exR1(n-2) + n;
}
```

56

## 1.1.17 找出以下递归函数的问题：

```
public static String exR2(int n)
{
    String s = exR2(n-3) + n + exR2(n-2) + n;
    if (n <= 0) return "";
    return s;
}
```

答：这段代码中的基本情况永远不会被访问。调用 exR2(3) 会产生调用 exR2(0)、exR2(-3) 和 exR2(-6)，循环往复直到发生 StackOverflowError。

## 1.1.18 请看以下递归函数：

```
public static int mystery(int a, int b)
{
    if (b == 0)      return 0;
    if (b % 2 == 0) return mystery(a+a, b/2);
    return mystery(a+a, b/2) + a;
}
```

mystery(2, 25) 和 mystery(3, 11) 的返回值是多少？给定正整数 a 和 b，mystery(a,b) 计算的结果是什么？将代码中的 + 替换为 \* 并将 return 0 改为 return 1，然后回答相同的问题。

## 1.1.19 在计算机上运行以下程序：

```

public class Fibonacci
{
    public static long F(int N)
    {
        if (N == 0) return 0;
        if (N == 1) return 1;
        return F(N-1) + F(N-2);
    }
    public static void main(String[] args)
    {
        for (int N = 0; N < 100; N++)
            StdOut.println(N + " " + F(N));
    }
}

```

57

计算机用这段程序在一个小时之内能够得到  $F(N)$  结果的最大  $N$  值是多少？开发  $F(N)$  的一个更好的实现，用数组保存已经计算过的值。

- 1.1.20 编写一个递归的静态方法计算  $\ln(N!)$  的值。
- 1.1.21 编写一段程序，从标准输入按行读取数据，其中每行都包含一个名字和两个整数。然后用 `printf()` 打印一张表格，每行的若干列数据包括名字、两个整数和第一个整数除以第二个整数的结果，精确到小数点后三位。可以用这种程序将棒球球手的击球命中率或者学生的考试分数制成表格。
- 1.1.22 使用 1.1.6.4 节中的 `rank()` 递归方法重新实现 `BinarySearch` 并跟踪该方法的调用。每当该方法被调用时，打印出它的参数 `lo` 和 `hi` 并按照递归的深度缩进。提示：为递归方法添加一个参数来保存递归的深度。
- 1.1.23 为 `BinarySearch` 的测试用例添加一个参数：+ 打印出标准输入中不在白名单上的值；-，则打印出标准输入中在白名单上的值。
- 1.1.24 给出使用欧几里德算法计算 105 和 24 的最大公约数的过程中得到的一系列  $p$  和  $q$  的值。扩展该算法中的代码得到一个程序 `Euclid`，从命令行接受两个参数，计算它们的最大公约数并打印出每次调用递归方法时的两个参数。使用你的程序计算 1 111 111 和 1 234 567 的最大公约数。
- 1.1.25 使用数学归纳法证明欧几里德算法能够计算任意一对非负整数  $p$  和  $q$  的最大公约数。

58

## 提高题

- 1.1.26 将三个数字排序。假设 `a`、`b`、`c` 和 `t` 都是同一种原始数字类型的变量。证明以下代码能够将 `a`、`b`、`c` 按照升序排列：

```

if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }

```
  - 1.1.27 二项分布。估计用以下代码计算 `binomial(100, 50)` 将会产生的递归调用次数：

```

public static double binomial(int N, int k, double p)
{
    if (N == 0 && k == 0) return 1.0; and if (N < 0 || k < 0) return 0.0;
    return (1.0 - p)*binomial(N-1, k, p) + p*binomial(N-1, k-1);
}

```
- 将已经计算过的值保存在数组中并给出一个更好的实现。
- 1.1.28 删除重复元素。修改 `BinarySearch` 类中的测试用例来删去排序之后白名单中的所有重复元素。

- 1.1.29 等值键。为 `BinarySearch` 类添加一个静态方法 `rank()`，它接受一个键和一个整型有序数组（可能存在重复键）作为参数并返回数组中小于该键的元素数量，以及一个类似的方法 `count()` 来返回数组中等于该键的元素的数量。注意：如果 `i` 和 `j` 分别是 `rank(key, a)` 和 `count(key, a)` 的返回值，那么 `a[i..i+j-1]` 就是数组中所有和 `key` 相等的元素。
- 1.1.30 数组练习。编写一段程序，创建一个  $N \times N$  的布尔数组 `a[][]`。其中当 `i` 和 `j` 互质时（没有相同因子），`a[i][j]` 为 `true`，否则为 `false`。
- 1.1.31 随机连接。编写一段程序，从命令行接受一个整数 `N` 和 `double` 值 `p`（0 到 1 之间）作为参数，在一个圆上画出大小为 0.05 且间距相等的 `N` 个点，然后将每对点按照概率 `p` 用灰线连接。  
[59]
- 1.1.32 直方图。假设标准输入流中含有一系列 `double` 值。编写一段程序，从命令行接受一个整数 `N` 和两个 `double` 值 `l` 和 `r`。将  $(l, r)$  分为 `N` 段并使用 `StdDraw` 画出输入流中的值落入每段的数量的直方图。
- 1.1.33 矩阵库。编写一个 `Matrix` 库并实现以下 API:

---

<code>public class Matrix</code>	
<code>static double dot(double[] x, double[] y)</code>	向量点乘
<code>static double[][] mult(double[][] a, double[][] b)</code>	矩阵和矩阵之积
<code>static double[] transpose(double[][] a)</code>	转置矩阵
<code>static double[] mult(double[][] a, double[] x)</code>	矩阵和向量之积
<code>static double[] mult(double[] y, double[][] a)</code>	向量和矩阵之积

---

编写一个测试用例，从标准输入读取矩阵并测试所有方法。

- 1.1.34 过滤。以下哪些任务需要（在数组中，比如）保存标准输入中的所有值？哪些可以被实现为一个过滤器且仅使用固定数量的变量和固定大小的数组（和 `N` 无关）？在每个问题中，输入都来自于标准输入且含有 `N` 个 0 到 1 的实数。  
[60]
- 打印出最大和最小的数
  - 打印出所有数的中位数
  - 打印出第 `k` 小的数，`k` 小于 100
  - 打印出所有数的平方和
  - 打印出 `N` 个数的平均值
  - 打印出大于平均值的数的百分比
  - 将 `N` 个数按照升序打印
  - 将 `N` 个数按照随机顺序打印

## 实验题

- 1.1.35 模拟掷骰子。以下代码能够计算每种两个骰子之和的准确概率分布：

```
int SIDES = 6;
double[] dist = new double[2*SIDES+1];
for (int i = 1; i <= SIDES; i++)
    for (int j = 1; j <= SIDES; j++)
        dist[i+j] += 1.0;

for (int k = 2; k <= 2*SIDES; k++)
    dist[k] /= 36.0;
```

`dist[i]` 的值就是两个骰子之和为  $i$  的概率。用实验模拟  $N$  次掷骰子，并在计算两个 1 到 6 之间的随机整数之和时记录每个值的出现频率以验证它们的概率。 $N$  要多大才能够保证你的经验数据和准确数据的吻合程度达到小数点后三位？

- 1.1.36 乱序检查。通过实验检查表 1.1.10 中的乱序代码是否能够产生预期的效果。编写一个程序 `ShuffleTest`，接受命令行参数  $M$  和  $N$ ，将大小为  $M$  的数组打乱  $N$  次且在每次打乱之前都将数组重新初始化为  $a[i] = i$ 。打印一个  $M \times M$  的表格，对于所有的列  $j$ ，行  $i$  表示的是  $i$  在打乱后落到  $j$  的位置的次数。数组中的所有元素的值都应该接近于  $N/M$ 。

- 1.1.37 糟糕的打乱。假设在我们的乱序代码中你选择的是一个 0 到  $N-1$  而非  $i$  到  $N-1$  之间的随机整数。证明得到的结果并非均匀地分布在  $N!$  种可能性之间。用上一题中的测试检验这个版本。

- 1.1.38 二分查找与暴力查找。根据 1.1.10.4 节给出的暴力查找法编写一个程序 `BruteForceSearch`，在你的计算机上比较它和 `BinarySearch` 处理 `largeW.txt` 和 `largeT.txt` 所需的时间。 61

- 1.1.39 随机匹配。编写一个使用 `BinarySearch` 的程序，它从命令行接受一个整型参数  $T$ ，并会分别针对  $N=10^3$ 、 $10^4$ 、 $10^5$  和  $10^6$  将以下实验运行  $T$  遍：生成两个大小为  $N$  的随机 6 位正整数数组并找出同时存在于两个数组中的整数的数量。打印一个表格，对于每个  $N$ ，给出  $T$  次实验中该数量的平均值。 62



## 第 2 章 排序

排序就是将一组对象按照某种逻辑顺序重新排列的过程。比如，信用卡账单中的交易是按照日期排序的——这种排序很可能使用了某种排序算法。在计算时代早期，大家普遍认为 30% 的计算周期都用在了排序上。如果今天这个比例降低了，可能的原因之一是如今的排序算法更加高效，而并非排序的重要性降低了。现在计算机的广泛使用使得数据无处不在，而整理数据的第一步通常就是进行排序。所有的计算机系统都实现了各种排序算法以供系统和用户使用。

即使你只是使用标准库中的排序函数，学习排序算法仍然有三大实际意义：

- 对排序算法的分析将有助于你全面理解本书中比较算法性能的方法；
- 类似的技术也能有效解决其他类型的问题；
- 排序算法常常是我们解决其他问题的第一步。

更重要的是这些算法都很经典、优雅和高效。

排序在商业数据处理和现代科学计算中有着重要的地位，它能够应用于事物处理、组合优化、天体物理学、分子动力学、语言学、基因组学、天气预报和很多其他领域。其中一种排序算法（快速排序，见 2.3 节）甚至被誉为 20 世纪科学和工程领域的十大算法之一。

在本章中我们将学习几种经典的排序算法，并高效地实现了“优先队列”这种基础数据类型。我们将讨论比较排序算法的理论基础并在本章结尾总结若干排序算法和优先队列的应用。

## 2.1 初级排序算法

作为对排序算法领域的第一次探索，我们将学习两种初级的排序算法以及其中一种的一个变体。深入学习这些相对简单的算法的原因在于：第一，我们将通过它们熟悉一些术语和简单的技巧；第二，这些简单的算法在某些情况下比我们之后将会讨论的复杂算法更有效；第三，以后你会发现，它们有助于我们改进复杂算法的效率。

### 2.1.1 游戏规则

我们关注的主要对象是重新排列数组元素的算法，其中每个元素都有一个主键。排序算法的目标就是将所有元素的主键按照某种方式排列（通常是按照大小或是字母顺序）。排序后索引较大的主键大于等于索引较小的主键。元素和主键的具体性质在不同的应用中千差万别。在 Java 中，元素通常都是对象，对主键的抽象描述则是通过一种内置的机制（请见 2.1.1.4 节中的 Comparable 接口）来完成的。

“排序算法类模版”中的 Example 类展示了我们的习惯约定：我们会将排序代码放在类的 sort() 方法中，该类还将包含辅助函数 less() 和 exch()（可能还有其他辅助函数）以及一个示例用例 main()。Example 类还包含了一些早期调试使用的代码：测试用例 main() 将标准输入得到的字符串排序，并用私有方法 show() 打印字符数组的内容。我们还会在本章中遇到各种用于比较不同算法并研究它们的性能的测试用例。为了区别不同的排序算法，我们为相应的类取了不同的名字，用例可以根据名字调用不同的实现，例如 Insertion.sort()、Merge.sort()、Quick.sort() 等。

大多数情况下，我们的排序代码只会通过两个方法操作数据：less() 方法对元素进行比较，exch() 方法将元素交换位置。exch() 方法的实现很简单，通过 Comparable 接口实现 less() 方法也不困难。将数据操作限制在这两个方法中使得代码的可读性和可移植性更好，更容易验证代码的正确性、分析性能以及排序算法之间的比较。在学习具体的排序算法实现之前，我们先讨论几个对于所有排序算法都很重要的问题。

244

#### 排序算法类的模板

```
public class Example
{
    public static void sort(Comparable[] a)
    { /* 请见算法2.1、算法2.2、算法2.3、算法2.4、算法2.5或算法2.7*/ }

    private static boolean less(Comparable v, Comparable w)
    { return v.compareTo(w) < 0; }

    private static void exch(Comparable[] a, int i, int j)
    { Comparable t = a[i]; a[i] = a[j]; a[j] = t; }

    private static void show(Comparable[] a)
    { // 在单行中打印数组
        for (int i = 0; i < a.length; i++)
            StdOut.print(a[i] + " ");
        StdOut.println();
    }

    public static boolean isSorted(Comparable[] a)
    { // 测试数组元素是否有序
        for (int i = 1; i < a.length; i++)
            if (!less(a[i], a[i-1])) return false;
    }
}
```

```
% more tiny.txt
S O R T E X A M P L E

% java Example < tiny.txt
A E E L M O P R S T X
```

```

        return true;
    }

    public static void main(String[]
args)
    { // 从标准输入读取字符串，将它们排序并输出
String[] a = In.readStrings();
sort(a);
assert isSorted(a);
show(a);
}
}

```

```

% more words3.txt
bed bug dad yes zoo ... all bad yet
% java Example < words.txt
all bad bed bug dad ... yes yet zoo

```

这个类展示的是数组排序实现的框架。对于我们学习的每种排序算法，我们都会为这样一个类实现一个 `sort()` 方法并将 `Example` 改为算法的名称。测试用例会将标准输入得到的字符串排序，但是这段代码使我们的排序方法适用于任意实现了 `Comparable` 接口的数据类型。

245

### 2.1.1.1 验证

无论数组的初始状态是什么，排序算法都能成功吗？谨慎起见，我们会在测试代码中添加一条语句 `assert isSorted(a);` 来确认排序后数组元素都是有序的。尽管一般都会测试代码并从数学上证明算法的正确性，但在实现每个排序算法时加上这条语句仍然是必要的。需要注意的是，如果我们只使用 `exch()` 来交换数组的元素，这个测试就足够了。当我们直接将值存入数组中时，这条语句无法提供足够的保证（例如，把初始输入数组的元素全部置为相同的值也能通过这个测试）。

### 2.1.1.2 运行时间

我们还要评估算法的性能。首先，要计算各个排序算法在不同的随机输入下的基本操作的次数（包括比较和交换，或者是读写数组的次数）。然后，我们用这些数据来估计算法的相对性能并介绍在实验中验证这些猜想所使用的工具。对于大多数实现，代码风格一致会使我们更容易作出对性能的合理猜想。

**排序成本模型。**在研究排序算法时，我们需要计算比较和交换的数量。对于不交换元素的算法，我们会计算访问数组的次数。

### 2.1.1.3 额外的内存使用

排序算法的额外内存开销和运行时间是同等重要的。排序算法可以分为两类：除了函数调用所需的栈和固定数目的实例变量之外无需额外内存的原地排序算法，以及需要额外内存空间来存储另一份数组副本的其他排序算法。

### 2.1.1.4 数据类型

我们的排序算法模板适用于任何实现了 `Comparable` 接口的数据类型。遵守 Java 惯例的好处是很多你希望排序的数据都实现了 `Comparable` 接口。例如，Java 中封装数字的类型 `Integer` 和 `Double`，以及 `String` 和其他许多高级数据类型（如 `File` 和 `URL`）都实现了 `Comparable` 接口。因此你可以直接用这些类型的数组作为参数调用我们的排序方法。例如，右上方的代码使用了快速排序（请见 2.3 节）来对 N 个随机的 `Double` 数据进行排序。

246

```

Double a[] = new Double[N];
for (int i = 0; i < N; i++)
    a[i] = StdRandom.uniform();
Quick.sort(a);

```

将N个随机值的数组排序

在创建自己的数据类型时，我们只要实现 `Comparable` 接口就能够保证用例代码可以将其排序。要做到这一点，只需要实现一个 `compareTo()` 方法来定义目标类型对象的自然次序，如右侧的 `Date` 数据类型所示（参见表 1.2.12）。

对于  $v < w$ 、 $v = w$  和  $v > w$  三种情况，Java 的习惯是在 `v.compareTo(w)` 被调用时分别返回一个负整数、零和一个正整数（一般是 -1、0 和 1）。为了节约篇幅，我们接下来用  $v > w$  来表示 `v.compareTo(w) > 0` 这样的代码。一般来说，如果 `v` 和 `w` 无法比较或者两者之一是 `null`，`v.compareTo(w)` 将会抛出一个异常。此外，`compareTo()` 必须实现一个完整的比较序列，即：

- 自反性，对于所有的 `v`，`v=v`；
- 反对称性，对于所有的  $v < w$  都有  $v > w$ ，且  $v=w$  时  $w=v$ ；
- 传递性，对于所有的 `v`、`w` 和 `x`，如果  $v \leq w$  且  $w \leq x$ ，则  $v \leq x$ 。

从数学上来说这些规则都很标准和自然，遵守它们应该不难。总之，`compareTo()` 实现了我们的主键抽象——它给出了实现了 `Comparable` 接口的任意数据类型的对象的大小顺序的定义。需要注意的是 `compareTo()` 方法不一定会用到进行比较的实例的所有实例变量，毕竟数组元素的主键很可能只是每个元素的一小部分。

本章剩余篇幅将会讨论对一组自然次序的对象进行排序的各种算法。为了比较和对照各种算法，我们会检查它们的许多性质，包括在各种输入下它们比较和交换数组元素的次数以及额外内存的使用量。通过这些我们能够对它们的性能作出猜想，而这些猜想在过去的数十年间已经在无数的计算机上被验证过了。所有的实现都是需要通过检验的，所以我们也会讨论相关的工具。在研究经典的选择排序、插入排序、希尔排序、归并排序、快速排序和堆排序之后，我们将在 2.5 节讨论一些实际的应用和问题。

## 2.1.2 选择排序

一种最简单的排序算法是这样的：首先，找到数组中最小的那个元素，其次，将它和数组的第一个元素交换位置（如果第一个元素就是最小元素那么它就和自己交换）。再次，在剩下的元素中找到最小的元素，将它与数组的第二个元素交换位置。如此往复，直到将整个数组排序。这种方法叫做选择排序，因为它在不断地选择剩余元素之中的最小者。

如算法 2.1 所示，选择排序的内循环只是在比较当前元素与目前已知的最小元素（以及将当前索引加 1 和检查是否代码越界），这已经简单到了极点。交换元素的代码写在内循环之外，每次交换都能排定一个元素，因此交换的总次数是  $N$ 。所以算法的时间效率取决于比较的次数。

```
public class Date implements Comparable<Date>
{
    private final int day;
    private final int month;
    private final int year;

    public Date(int d, int m, int y)
    {   day = d; month = m; year = y; }

    public int day() { return day; }
    public int month() { return month; }
    public int year() { return year; }

    public int compareTo(Date that)
    {
        if (this.year > that.year) return +1;
        if (this.year < that.year) return -1;
        if (this.month > that.month) return +1;
        if (this.month < that.month) return -1;
        if (this.day > that.day) return +1;
        if (this.day < that.day) return -1;
        return 0;
    }

    public String toString()
    { return month + "/" + day + "/" + year; }
}
```

定义一个可比较的数据类型

**命题 A。**对于长度为  $N$  的数组，选择排序需要大约  $N^2/2$  次比较和  $N$  次交换。

**证明。**可以通过算法的排序轨迹来证明这一点。我们用一张  $N \times N$  的表格来表示排序的轨迹（见算法 2.1 下部的表格），其中每个非灰色字符都表示一次比较。表格中大约一半的元素不是灰色的——即对角线和其上部分的元素。对角线上的每个元素都对应着一次交换。通过查看代码我们可以更精确地得到，0 到  $N-1$  的任意  $i$  都会进行一次交换和  $N-1-i$  次比较，因此总共有  $N$  次交换以及  $(N-1)+(N-2)+\dots+2+1=N(N-1)/2 \sim N^2/2$  次比较。

总的来说，选择排序是一种很容易理解和实现的简单排序算法，它有两个很鲜明的特点。

运行时间和输入无关。为了找出最小的元素而扫描一遍数组并不能为下一遍扫描提供什么信息。这种性质在某些情况下是缺点，因为使用选择排序的人可能会惊讶地发现，一个已经有序的数组或是主键全部相等的数组和一个元素随机排列的数组所用的排序时间竟然一样长！我们将会看到，其他算法会更善于利用输入的初始状态。

数据移动是最少的。每次交换都会改变两个数组元素的值，因此选择排序用了  $N$  次交换——交换次数和数组的大小是线性关系。我们将研究的其他任何算法都不具备这个特征（大部分的增长数量级都是线性对数或是平方级别）。

248

### 算法 2.1 选择排序

```
public class Selection
{
    public static void sort(Comparable[] a)
    { // 将a[]按升序排列
        int N = a.length; // 数组长度
        for (int i = 0; i < N; i++)
        { // 将a[i]和a[i+1..N]中最小的元素交换
            int min = i; // 最小元素的索引
            for (int j = i+1; j < N; j++)
                if (less(a[j], a[min])) min = j;
            exch(a, i, min);
        }
    }
    // less()、exch()、isSorted()和main()方法见“排序算法类模板”
}
```

该算法将第  $i$  小的元素放到  $a[i]$  之中。数组的第  $i$  个位置的左边是  $i$  个最小的元素且它们不会再被访问。

		a [ ]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R

算法在黑色的  
元素中查找最小值

加粗的元  
素都是  $a[min]$

8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

选择排序的轨迹（每次交换后的数组内容）

249

### 2.1.3 插入排序

通常人们整理桥牌的方法是一张一张的来，将每一张牌插入到其他已经有序的牌中的适当位置。在计算机的实现中，为了给要插入的元素腾出空间，我们需要将其余所有元素在插入之前都向右移动一位。这种算法叫做插入排序，实现请见算法 2.2。

与选择排序一样，当前索引左边的所有元素都是有序的，但它们的最终位置还不确定，为了给更小的元素腾出空间，它们可能会被移动。但是当索引到达数组的右端时，数组排序就完成了。

和选择排序不同的是，插入排序所需的时间取决于输入中元素的初始顺序。例如，对一个很大且其中的元素已经有序（或接近有序）的数组进行排序将会比对随机顺序的数组或是逆序数组进行排序要快得多。

**命题 B。**对于随机排列的长度为  $N$  且主键不重复的数组，平均情况下插入排序需要  $\sim N^2/4$  次比较以及  $\sim N^2/4$  次交换。最坏情况下需要  $\sim N^2/2$  次比较和  $\sim N^2/2$  次交换，最好情况下需要  $N-1$  次比较和 0 次交换。

**证明。**和命题 A 一样，通过一个  $N \times N$  的轨迹表可以很容易就得到交换和比较的次数。最坏情况下对角线之下所有的元素都需要移动位置，最好情况下都不需要。对于随机排列的数组，在平均情况下每个元素都可能向后移动半个数组的长度，因此交换总数是对角线之下的元素总数的二分之一。

比较的总次数是交换的次数加上一个额外的项，该项为  $N$  减去被插入的元素正好是已知的最小元素的次数。在最坏情况下（逆序数组），这一项相对于总数可以忽略不计；在最好情况下（数组已经有序），这一项等于  $N-1$ 。

插入排序对于实际应用中常见的某些类型的非随机数组很有效。例如，正如刚才所提到的，想想当你用插入排序对一个有序数组进行排序时会发生什么。插入排序能够立即发现每个元素都已经在合适的位置之上，它的运行时间也是线性的（对于这种数组，选择排序的运行时间是平方级别的）。对于所有主键都相同的数组也会出现相同的情况（因此命题 B 的条件之一就是主键不重复）。

250

### 算法 2.2 插入排序

```
public class Insertion
{
    public static void sort(Comparable[] a)
    { // 将 a[] 按升序排列
        int N = a.length;
        for (int i = 1; i < N; i++)
        { // 将 a[i] 插入到 a[i-1]、a[i-2]、a[i-3]...之中
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
```

```

        exch(a, j, j-1);
    }
}
// less()、exch()、isSorted()和main()方法见“排序算法类模板”
}

```

对于 0 到  $N-1$  之间的每一个  $i$ , 将  $a[i]$  与  $a[0]$  到  $a[i-1]$  中比它小的所有元素依次有序地交换。在索引  $i$  由左向右变化的过程中, 它左侧的元素总是有序的, 所以当  $i$  到达数组的右端时排序就完成了。

		a [ ]										
i	j	0	1	2	3	4	5	6	7	8	9	10
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

插入排序的轨迹（每次插入后的数组内容）

251

我们要考虑的更一般的情况是部分有序的数组。倒置指的是数组中的两个顺序颠倒的元素。比如 EXAMPLE 中有 11 对倒置: E-A、X-A、X-M、X-P、X-L、X-E、M-L、M-E、P-L、P-E 以及 L-E。如果数组中倒置的数量小于数组大小的某个倍数, 那么我们说这个数组是部分有序的。下面是几种典型的部分有序的数组:

- 数组中每个元素距离它的最终位置都不远;
- 一个有序的大数组接一个小数组;
- 数组中只有几个元素的位置不正确。

插入排序对这样的数组很有效, 而选择排序则不然。事实上, 当倒置的数量很少时, 插入排序很可能比本章中的其他任何算法都要快。

**命题 C。** 插入排序需要的交换操作和数组中倒置的数量相同, 需要的比较次数大于等于倒置的数量, 小于等于倒置的数量加上数组的大小再减一。

**证明。** 每次交换都改变了两个顺序颠倒的元素的位置, 相当于减少了一对倒置, 当倒置数量为 0 时, 排序就完成了。每次交换都对应着一次比较, 且 1 到  $N-1$  之间的每个  $i$  都可能需要一次额外的比较 (在  $a[i]$  没有达到数组的左端时)。

要大幅提高插入排序的速度并不难, 只需要在内循环中将较大的元素都向右移动而不总是交换两个元素 (这样访问数组的次数就能减半)。我们把这项改进留做一个练习 (请见练习 2.1.25)。

总的来说，插入排序对于部分有序的数据十分高效，也很适合小规模数组。这很重要，因为这些类型的数组在实际应用中经常出现，而且它们也是高级排序算法的中间过程。我们会在学习高级排序算法时再次接触到插入排序。

#### 2.1.4 排序算法的可视化

在本章中我们会使用一种简单的图示来帮助我们说明排序算法的性质。我们没有使用字母、数字或是单词这样的键值来跟踪排序的进程，而使用了棒状图，并以它们的高矮来排序。这种表示方法的好处是能够使排序过程一目了然。

如图 2.1.1 所示，插入排序不会访问索引右侧的元素，而选择排序不会访问索引左侧的元素。另外，在这种可视化的轨迹图中可以看到，因为插入排序不会移动比被插入的元素更小的元素，它所需的比较次数平均只有选择排序的一半。

用我们的 StdDraw 库画出一张可视轨迹图并不比追踪一次算法的运行轨迹难多少。将 Double 值排序，并在适当的时候指示算法调用 show() 方法（和追踪算法的轨迹时一样），然后开发一个使用 StdDraw 来绘制棒状图而不是打印结果的 show() 方法。最复杂的部分是设置 y 轴的比例以使轨迹的线条符合预期的顺序。请通过练习 2.1.18 来更好地理解可视轨迹图的价值和使用。

将轨迹变成动画，理解起来就更加简单，这样可以看到动态演化到有序状态的过程。产生轨迹动画的过程本质上和上一段所描述的相同，但不需要担心 y 轴的问题（只需每次擦除窗口中的内容并重绘棒状图即可）。尽管我们无法在书中展现这些动画，它们对于理解算法的工作原理也很有帮助，你能通过练习 2.1.17 体会这一点。

#### 2.1.5 比较两种排序算法

现在我们已经实现了两种排序算法，我们很自然地想知道选择排序（算法 2.1）和插入排序（算法 2.2）哪种更快。这个问题在学习算法的过程中会反复出现，也是本书的重点之一。我们已经在第 1 章中讨论过一些基本的概念，这里我们第一次用实践说明我们解决这个问题的办法。一般来说，根据 1.4 节所介绍的方法，我们将通过以下步骤比较两个算法：

- 实现并调试它们；
- 分析它们的基本性质；
- 对它们的相对性能作出猜想；

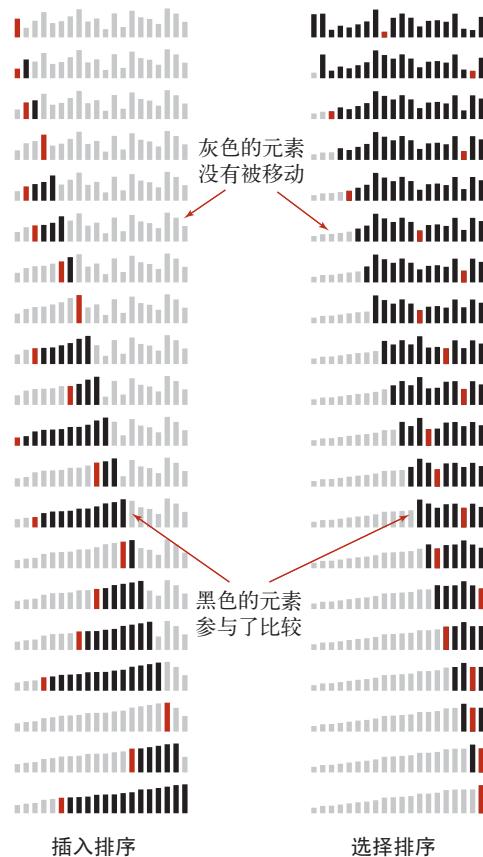


图 2.1.1 初级排序算法的可视轨迹图（另见彩插）

## □ 用实验验证我们的猜想。

这些步骤都是经过时间检验的科学方法，只是现在是运用在算法研究之上。

现在，算法 2.1 和算法 2.2 表示已经实现了第一步，命题 A、命题 B 和命题 C 组成了第二步，下面的性质 D 将是第三步，之后“比较两种排序算法”的 `SortCompare` 类将会完成第四步。这些行为都是紧密相关的。

在这些简洁的步骤之下是大量的算法实现、调试分析和测试工作。每个程序员都知道只有经过长期的调试和改进才能得到这样的代码，每个数学家都知道正确分析的难度，每个科学家也都知道从提出猜想到设计并执行实验来验证它们是多么费心。只有研究那些最重要的算法的专家才会经历完整的研究过程，但每个使用算法的程序员都应该了解算法的性能特性背后的科学过程。

实现了算法之后，下一步我们需要确定一个适当的输入模型。对于排序，命题 A、命题 B 和命题 C 用到的自然输入模型假设数组中的元素随机排序，且主键值不会重复。对于有很多重复主键的应用来说，我们需要一个更加复杂的模型。

如何估计插入排序和选择排序在随机排序数组下的性能呢？通过算法 2.1 和算法 2.2 以及命题 A、命题 B 和命题 C 可以发现，对于随机排序数组，两者的运行时间都是平方级别的。也就是说，在这种输入下插入排序的运行时间和  $N^2$  乘以一个小常数成正比，选择排序的运行时间和  $N^2$  乘以另一个小常数成比例。这两个常数的值取决于所使用的计算机中比较和交换元素的成本。对于许多数据类型和一般的计算机，可以假设这些成本是相近的（但我们也看到一些大不相同的例外）。因此我们直接得出了以下猜想。  
254

**性质 D。**对于随机排序的无重复主键的数组，插入排序和选择排序的运行时间是平方级别的，两者之比应该是一个较小的常数。

**例证。**这个结论在过去的半个世纪中已经在许多不同类型的计算机上经过了验证。在 1980 年本书第 1 版完成之时插入排序就比选择排序快一倍，现在仍然是这样，尽管那时这些算法将 10 万条数据排序需要几个小时而现在只需要几秒钟。在你的计算机上插入排序也比选择排序快一些吗？可以通过 `SortCompare` 类来检测。它会使用由命令行参数指定的排序算法名称所对应的 `sort()` 方法进行指定次数的实验（将指定大小的数组排序），并打印出所观察到的各种算法的运行时间的比例。

为了证明这一点，我们用 `SortCompare`（见“比较两种排序算法”）来做几次实验。我们使用 `Stopwatch` 来计时，右侧的 `time()` 函数的任务是调用本章中的几种简单排序算法。

随机数组的输入模型由 `SortCompare` 类中的 `timeRandomInput()` 方法实现。这个方法会生成随机的 `Double` 值，将它们排

```
public static double time(String alg, Comparable[] a)
{
    Stopwatch timer = new Stopwatch();
    if (alg.equals("Insertion")) Insertion.sort(a);
    if (alg.equals("Selection")) Selection.sort(a);
    if (alg.equals("Shell")) Shell.sort(a);
    if (alg.equals("Merge")) Merge.sort(a);
    if (alg.equals("Quick")) Quick.sort(a);
    if (alg.equals("Heap")) Heap.sort(a);
    return timer.elapsedTime();
}
```

针对给定输入，为本章中的一种排序算法计时

序，并返回指定次测试的总时间。使用 0.0 至 1.0 之间的随机 `Double` 值比使用类似于 `StdRandom.shuffle()` 的库函数更简单有效，因为这样几乎不可能产生相等的主键值（请见练习 2.5.31）。如第 1 章中所讨论的，用命令行参数指定重复次数的好处是能够运行大量的测试（测试次数越多，每遍测试所需的平均时间就越接近于真实的平均数据）并且能够减小系统本身的影响。你应该在自己的计算机上用 `SortCompare` 进行实验，来了解关于插入排序和选择排序的结论是否成立。

### 比较两种排序算法

```
public class SortCompare
{
    public static double time(String alg, Double[] a)
    { /* 请见前面的正文 */

        public static double timeRandomInput(String alg, int N, int T)
        { // 使用算法1将T个长度为N的数组排序
            double total = 0.0;
            Double[] a = new Double[N];
            for (int t = 0; t < T; t++)
            { // 进行一次测试（生成一个数组并排序）
                for (int i = 0; i < N; i++)
                    a[i] = StdRandom.uniform();
                total += time(alg, a);
            }
            return total;
        }

        public static void main(String[] args)
        {
            String alg1 = args[0];
            String alg2 = args[1];
            int N = Integer.parseInt(args[2]);
            int T = Integer.parseInt(args[3]);
            double t1 = timeRandomInput(alg1, N, T); // 算法1的总时间
            double t2 = timeRandomInput(alg2, N, T); // 算法2的总时间
            StdOut.printf("For %d random Doubles\n    %s is", N, alg1);
            StdOut.printf(" %.1f times faster than %s\n", t2/t1, alg2);
        }
    }
}
```

这个用例会运行由前两个命令行参数指定的排序算法，对长度为  $N$ （由第三个参数指定）的 `Double` 随机数组进行排序，元素值均在 0.0 到 1.0 之间，重复  $T$  次（由第四个参数指定），然后输出总运行时间的比例。

```
% java SortCompare Insertion Selection 1000 100
For 1000 random Doubles
    Insertion is 1.7 times faster than Selection
```

255
256

我们故意将性质 D 描述得不够明确——没有说明那个小常量的值，以及对比较和交换的成本相近的假设，这样性质 D 才能广泛适用于各种情况。可能的话，我们会尽量用这样的语言来抓住我们所研究的每个算法的性能的本质。如第 1 章中讨论的那样，我们提出的每个性质都需要在特定的场景中进行科学测试，也许还需要用一个基于相关命题（数学定理）的猜想进行补充。

对于实际应用，还有一个很重要的步骤，那就是用实际数据在实验中验证我们的猜想。我们会在2.5节和练习中再考虑这一点。在这种情况下，当主键有重复或是排列不随机，性质D就可能会不成立。可以使用`StdRandom.shuffle()`来将一个数组打乱，但有大量重复主键的情况则需要更加细致的分析。

我们对算法分析的讨论是抛砖引玉，而非盖棺定论。如果你想到了关于算法性能的其他问题，可以用`SortCompare`等工具来研究它，后面的练习为你提供了许多机会。

插入排序和选择排序的性能比较就讨论到这里，还存在许多比它们快成千上万倍的算法，我们对此会更感兴趣。当然，仍然有必要学习这些初级算法，因为：

- 它们帮助我们建立了一些基本的规则；
- 它们展示了一些性能基准；
- 在某些特殊情况下它们也是很好的选择；
- 它们是开发更强大的排序算法的基石。

因此，不止是排序，对于本书中的每个问题我们都会沿用这种方式，首先学习的就是最初级的相关算法。`SortCompare`这样的程序对于这种渐进式的算法研究十分重要。每一步，我们都能用这类程序来了解新的或是改进后的算法的性能是否产生了预期的进步。

257

### 2.1.6 希尔排序

为了展示初级排序算法性质的价值，接下来我们将学习一种基于插入排序的快速的排序算法。对于大规模乱序数组插入排序很慢，因为它只会交换相邻的元素，因此元素只能一点一点地从数组的一端移动到另一端。例如，如果主键最小的元素正好在数组的尽头，要将它挪到正确的位置就需要 $N-1$ 次移动。希尔排序为了加快速度简单地改进了插入排序，交换不相邻的元素以对数组的局部进行排序，并最终用插入排序将局部有序的数组排序。

希尔排序的思想是使数组中任意间隔为 $h$ 的元素都是有序的。这样的数组被称为 $h$ 有序数组。换句话说，一个 $h$ 有序数组就是 $h$ 个互相独立的有序数组编织在一起组成的一个数组（见图2.1.2）。在进行排序时，如果 $h$ 很大，我们就能将元素移动到很远的地方，为实现更小的 $h$ 有序创造方便。用这种方式，对于任意以1结尾的 $h$ 序列，我们都能够将数组排序。这就是希尔排序。算法2.3的实现使用了序列 $1/2(3^k-1)$ ，从 $N/3$ 开始递减至1。我们把这个序列称为递增序列。算法2.3实时计算了它的递增序列，另一种方式是将递增序列存储在一个数组中。

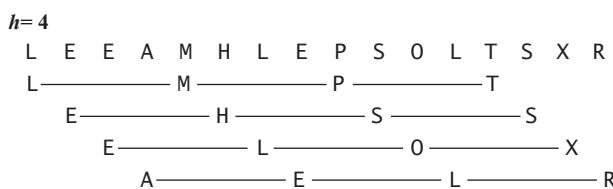


图2.1.2 一个 $h$ 有序数组即一个由 $h$ 个有序子数组组成的数组

实现希尔排序的一种方法是对于每个 $h$ ，用插入排序将 $h$ 个子数组独立地排序。但因为子数组是相互独立的，一个更简单的方法是在 $h$ -子数组中将每个元素交换到比它大的元素之前去（将比它大的元素向右移动一格）。只需要在插入排序的代码中将移动元素的距离由1改为 $h$ 即可。这样，希尔排序的实现就转化为了一个类似于插入排序但使用不同增量的过程。

希尔排序更高效的原因是它权衡了子数组的规模和有序性。排序之初，各个子数组都很短，排

序之后子数组都是部分有序的，这两种情况都很适合插入排序。子数组部分有序的程度取决于递增序列的选择。透彻理解希尔排序的性能至今仍然是一项挑战。实际上，算法 2.3 是我们唯一无法准确描述其对于乱序的数组的性能特征的排序方法。

### 算法 2.3 希尔排序

```
public class Shell
{
    public static void sort(Comparable[] a)
    { // 将a[]按升序排列
        int N = a.length;
        int h = 1;
        while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, 1093, ...
        while (h >= 1)
        { // 将数组变为h有序
            for (int i = h; i < N; i++)
            { // 将a[i]插入到a[i-h], a[i-2*h], a[i-3*h]...之中
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
                    exch(a, j, j-h);
            }
            h = h/3;
        }
    }
    // less(), exch(), isSorted()和main()方法见“排序算法类模板”
}
```

如果我们在插入排序（算法 2.2）中加入一个外循环来将  $h$  按照递增序列递减，我们就能得到这个简洁的希尔排序。增幅  $h$  的初始值是数组长度乘以一个常数因子，最小为 1。

```
% java SortCompare Shell Insertion 100000 100
For 100000 random Doubles
Shell is 600 times faster than Insertion
```

```
输入 S H E L L S O R T E X A M P L E
13-sort P H E L L S O R T E X A M S L E
4-sort L E E A M H L E P S O L T S X R
1-sort A E E E H L L M O P R S S T X
```

希尔排序的轨迹（每遍排序后的数组内容）

258  
259

如何选择递增序列呢？要回答这个问题并不简单。算法的性能不仅取决于  $h$ ，还取决于  $h$  之间的数学性质，比如它们的公因子等。有很多论文研究了各种不同的递增序列，但都无法证明某个序列是“最好的”。算法 2.3 中递增序列的计算和使用都很简单，和复杂递增序列的性能接近。但可以证明复杂的序列在最坏情况下的性能要好于我们所使用的递增序列。更加优秀的递增序列有待我们去发现。

和选择排序以及插入排序形成对比的是，希尔排序也可以用于大型数组。它对任意排序（不一定是随机的）的数组表现也很好。实际上，对于一个给定的递增序列，构造一个使希尔排序运行缓慢的

数组并不容易。希尔排序的轨迹如图 2.1.3 所示，可视轨迹如图 2.1.4 所示。

输入	S H E L L S O R T E X A M P L E
<b>13-sort</b>	P H E L L S O R T E X A M S L E
	P H E L L S O R T E X A M S L E
	P H E L L S O R T E X A M S L E
<b>4-sort</b>	L H E L P S O R T E X A M S L E
	L H E L P S O R T E X A M S L E
	L H E L P S O R T E X A M S L E
	L H E L P S O R T E X A M S L E
	L H E L P S O R T E X A M S L E
	L H E L P S O R T E X A M S L E
	L H E L P S O R T E X A M S L E
	L H E L P S O R T E X A M S L E
	L E E L P H O R T S X A M S L E
	L E E L P H O R T S X A M S L E
	L E E A P H O L T S X R M S L E
	L E E A M H O L P S X R T S L E
	L E E A M H O L P S X R T S L E
	L E E A M H L P S O R T S X E
	L E E A M H L E P S O L T S X R
<b>1-sort</b>	E L E A M H L E P S O L T S X R
	E E L A M H L E P S O L T S X R
	A E E L M H L E P S O L T S X R
	A E E L M H L E P S O L T S X R
	A E E H L M L E P S O L T S X R
	A E E H L M E P S O L T S X R
	A E E E H L M P S O L T S X R
	A E E E H L M P S O L T S X R
	A E E E H L M O P S L T S X R
	A E E E H L L M O P S T S X R
	A E E E H L L M O P S T S X R
	A E E E H L L M O P S S T X R
	A E E E H L L M O P R S S T X
结果	A E E E H L L M O P R S S T X

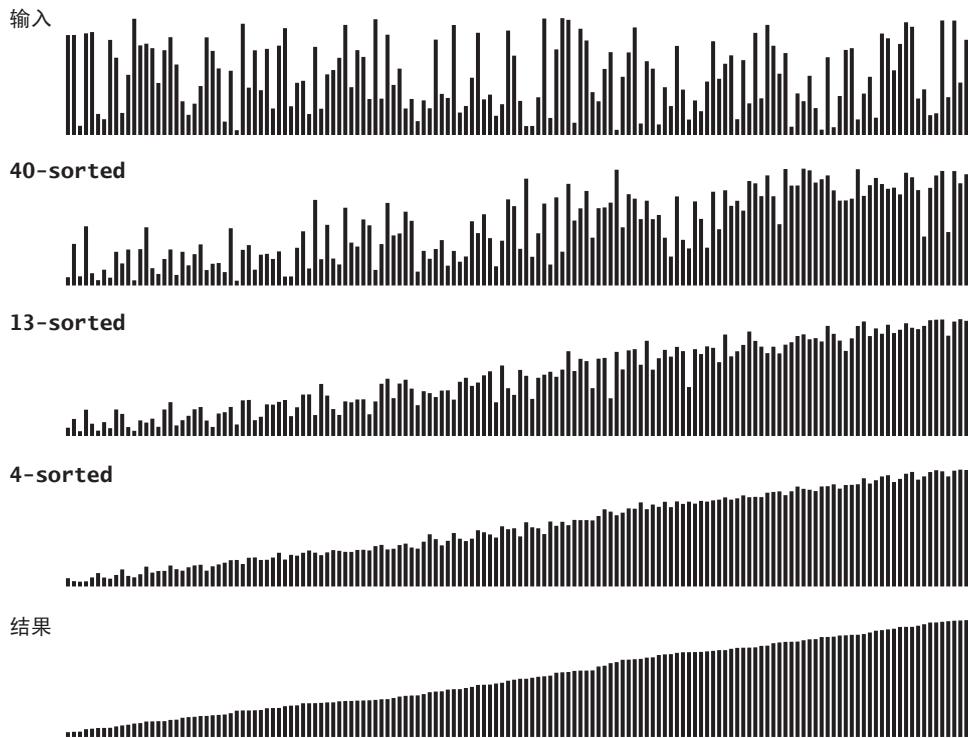
[260]

图 2.1.3 希尔排序的详细轨迹（各种插入）

通过 SortCompare 可以看到，希尔排序比插入排序和选择排序要快得多，并且数组越大，优势越大。在继续学习之前，请在你的计算机上用 SortCompare 比较一下希尔排序和插入排序以及选择排序的性能，数组的大小按照 2 的幂次递增（见练习 2.1.27）。你会看到希尔排序能够解决一些初级排序算法无能为力的问题。这个例子是我们第一次用实际应用说明一个贯穿本书的重要理念：通过提升速度来解决其他方式无法解决的问题是研究算法的设计和性能的主要原因之一。

研究希尔排序性能需要的数学论证超出了本书范围。如果你不相信，可以从证明下面这一点开始：当一个“h 有序”的数组按照增幅 k 排序之后，它仍然是“h 有序”的。至于算法 2.3 的性能，目前最重要的结论是它的运行时间达不到平方级别。例如，已知在最坏的情况下算法 2.3 的比较次数和  $N^{\beta/2}$  成正比。有意思的是，由插入排序到希尔排序，一个小小的改变就突破了平方级别的运行时间

的屏障。这正是许多算法设计问题想要达到的目标。



261

图 2.1.4 希尔排序的可视轨迹

在输入随机排序数组的情况下，我们在数学上还不知道希尔排序所需要的平均比较次数。人们发明了很多递增序列来渐进式地改进最坏情况下所需的比较次数 ( $N^{4/3}$ ,  $N^{5/4}$ ,  $N^{6/5}\dots$ )，但这些结论大多只有学术意义，因为对于实际应用中的  $N$  来说它们的递增序列的生成函数（以及与  $N$  乘以一个常数因子）之间的区别并不明显。

在实际应用中，使用算法 2.3 中的递增序列基本就足够了（或者是本节最后的练习中提供的一个递增序列，它可能可以将性能改进 20% ~ 40%）。另外，很容易就能验证下面这个猜想。

**性质 E。** 使用递增序列 1, 4, 13, 40, 121, 364… 的希尔排序所需的比较次数不会超出  $N$  的若干倍乘以递增序列的长度。

**例证。** 记录算法 2.3 中比较的数量并将其除以使用的序列长度是一道简单的练习（请见练习 2.1.12）。大量的实验证明平均每个增幅所带来的比较次数约为  $N^{1/5}$ ，但只有在  $N$  很大的时候这个增长幅度才会变得明显。这个性质似乎也和输入模型无关。

有经验的程序员有时会选择希尔排序，因为对于中等大小的数组它的运行时间是可以接受的。它的代码量很小，且不需要使用额外的内存空间。在下面的几节中我们会看到更加高效的算法，但除了对于很大的  $N$ ，它们可能只会比希尔排序快两倍（可能还达不到），而且更复杂。如果你需要解决一个排序问题而又没有系统排序函数可用（例如直接接触硬件或是运行于嵌入式系统中的代码），可以先用希尔排序，然后再考虑是否值得将它替换为更加复杂的排序算法。

262

## 答疑

问 排序看起来是个很简单的问题，我们用计算机不是可以做很多更有意思的事情吗？

答 也许吧，但快速的排序算法才使得那些更有意思的事情成为可能。在2.5节以及全书的其他章节你都可以找到很多这样的例子。排序算法今天仍然值得我们学习是因为它易于理解，你能从中领会到许多精妙之处。

问 为什么有这么多排序算法？

答 原因之一是许多排序算法的性能都和输入模型有很大的关系，因此不同的算法适用于不同应用场景中的不同输入。例如，对于部分有序和小规模的数组应该选择插入排序。其他限制条件，例如空间和重复的主键，也都是需要考虑的因素。我们将会在2.5节中再次讨论这个问题。

问 为什么要使用`less()`和`exch()`这些不起眼的辅助函数？

答 它们抽象了所有排序算法都会用到的共同操作，这种抽象使得代码更便于理解。而且它们增强了代码的可移植性。例如，算法2.1和算法2.2中的大部分代码在其他几种编程语言中也是可以执行的。即使是在Java中，只要将`less()`实现为`v < w`，这些算法的代码就可以将不支持`Comparable`接口的基本数据类型排序了。

问 当我运行`SortCompare`时，每次的结果都不一样（而且和书上的也不相同），为什么？

答 对于初学者，你的计算机和我们的计算机不同，操作系统、Java运行时环境等都不一样。这些不同可能导致算法代码生成的机器码不同。每次运行所得结果不同的原因可能在于当时运行的其他程序或是很多其他原因。大量的重复实验可以淡化这种干扰，我们的经验是现如今算法性能的微小差异很难观察。这就是我们要关注较大差异的原因。

[263]

## 练习

- 2.1.1 按照算法2.1所示轨迹的格式给出选择排序是如何将数组E A S Y Q U E S T I O N排序的。
- 2.1.2 在选择排序中，一个元素最多可能会被交换多少次？平均可能会被交换多少次？
- 2.1.3 构造一个含有 $N$ 个元素的数组，使选择排序（算法2.1）运行过程中 $a[j] < a[min]$ （由此 $min$ 会不断更新）成功的次数最大。
- 2.1.4 按照算法2.2所示轨迹的格式给出插入排序是如何将数组E A S Y Q U E S T I O N排序的。
- 2.1.5 构造一个含有 $N$ 个元素的数组，使插入排序（算法2.2）运行过程中内循环（`for`）的两个判断结果总是假。
- 2.1.6 在所有的主键都相同时，选择排序和插入排序谁更快？
- 2.1.7 对于逆序数组，选择排序和插入排序谁更快？
- 2.1.8 假设元素只可能有三种值，使用插入排序处理这样一个随机数组的运行时间是线性的还是平方级别的？或是介于两者之间？
- 2.1.9 按照算法2.3所示轨迹的格式给出希尔排序是如何将数组E A S Y S H E L L S O R T Q U E S T I O N排序的。
- 2.1.10 在希尔排序中为什么在实现 $h$ 有序时不使用选择排序？
- 2.1.11 将希尔排序中实时计算递增序列改为预先计算并存储在一个数组中。
- 2.1.12 令希尔排序打印出递增序列的每个元素所带来的比较次数和数组大小的比值。编写一个测试用例对随机`Double`数组进行希尔排序，验证该值是一个小常数，数组大小按照10的幂次递增，

不小于 100。

264

## 提高题

- 2.1.13 纸牌排序。说说你会如何将一副扑克牌按花色排序（花色顺序是黑桃、红桃、梅花和方片），限制条件是所有牌都是背面朝上排成一列，而你一次只能翻看两张牌或者交换两张牌（保持背面朝上）。
- 2.1.14 出列排序。说说你会如何将一副扑克牌排序，限制条件是只能查看最上面的两张牌，交换最上面的两张牌，或是将最上面的一张牌放到这摞牌的最下面。
- 2.1.15 昂贵的交换。一家货运公司的一位职员得到了一项任务，需要将若干大货箱按照发货时间摆放。比较发货时间很容易（对照标签即可），但将两个货箱交换位置则很困难（移动麻烦）。仓库已经快满了，只有一个空闲的仓位。这位职员应该使用哪种排序算法呢？
- 2.1.16 验证。编写一个 `check()` 方法，调用 `sort()` 对任意数组排序。如果排序成功而且数组中的所有对象均没有被修改则返回 `true`，否则返回 `false`。不要假设 `sort()` 只能通过 `exch()` 来移动数据，可以信任并使用 `Arrays.sort()`。
- 2.1.17 动画。修改插入排序和选择排序的代码，使之将数组内容绘制成为文中所示的棒状图。在每一轮排序后重绘图片来产生动画效果，并以一张“有序”的图片作为结束，即所有圆棒均已按照高度有序排列。提示：使用类似于文中的用例来随机生成 `Double` 值，在排序代码的适当位置调用 `show()` 方法，并在 `show()` 方法中清理画布并绘制棒状图。
- 2.1.18 可视轨迹。修改你为上一题给出的解答，为插入排序和选择排序生成和文中类似的可视轨迹。  
提示：使用 `setYscale()` 函数是一个明智的选择。附加题：添加必要的代码，与文中的图片一样用红色和灰色强调不同角色的元素。
- 2.1.19 希尔排序的最坏情况。用 1 到 100 构造一个含有 100 个元素的数组并用希尔排序和递增序列 1 4 13 40 对其排序，使比较的次数尽可能多。
- 2.1.20 希尔排序的最好情况。最好情况是什么？证明你的结论。

265

- 2.1.21 可比较的交易。用我们的 `Date` 类（请见 2.1.1.4 节）作为模板扩展你的 `Transaction` 类（请见练习 1.2.13），实现 `Comparable` 接口，使交易能够按照金额排序。

解答：

```
public class Transaction implements Comparable<Transaction>
{
    ...
    private final double amount;
    ...
    public int compareTo(Transaction that)
    {
        if (this.amount > that.amount) return +1;
        if (this.amount < that.amount) return -1;
        return 0;
    }
    ...
}
```

- 2.1.22 事务排序测试用例。编写一个 `SortTransaction` 类，在静态方法 `main()` 中从标准输入读取一系列事务，将它们排序并在标准输出中打印结果（请见练习 1.3.17）。

解答：

```

public class SortTransactions
{
    public static Transaction[] readTransactions()
    { // 请见练习 1.3.17 }
    public static void main(String[] args)
    {
        Transaction[] transactions = readTransactions();
        Shell.sort(transactions);
        for (Transaction t : transactions)
            StdOut.println(t);
    }
}

```

[266]

## 实验题

- 2.1.23 纸牌排序。请几位朋友分别将一副扑克牌排序（见练习 2.1.13）。仔细观察并记录他们所使用的方法。
- 2.1.24 插入排序的哨兵。在插入排序的实现中先找出最小的元素并将其置于数组的最左边，这样就能去掉内循环的判断条件  $j > 0$ 。使用 `SortCompare` 来评估这种做法的效果。注意：这是一种常见的规避边界测试的方法，能够省略判断条件的元素通常被称为哨兵。
- 2.1.25 不需要交换的插入排序。在插入排序的实现中使较大元素右移一位只需要访问一次数组（而不用使用 `exch()`）。使用 `SortCompare` 来评估这种做法的效果。
- 2.1.26 原始数据类型。编写一个能够处理 `int` 值的插入排序的新版本，比较它和正文中所给出的实现（能够隐式地用自动装箱和拆箱转换 `Integer` 值并排序）的性能。
- 2.1.27 希尔排序的用时是次平方级的。在你的计算机上用 `SortCompare` 比较希尔排序和插入排序以及选择排序。测试数组的大小按照 2 的幂次递增，从 128 开始。
- 2.1.28 相等的主键。对于主键仅可能取两种值的数组，评估和验证插入排序和选择排序的性能，假设两种主键值出现的概率相同。
- 2.1.29 希尔排序的递增序列。通过实验比较算法 2.3 中所使用的递增序列和递增序列 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, 16 001, 36 289, 64 769, 146 305, 260 609（这是通过序列  $9 \times 4^k - 9 \times 2^k + 1$  和  $4^k - 3 \times 2^k + 1$  综合得到的）。可以参考练习 2.1.11。
- 2.1.30 几何级数递增序列。通过实验找到一个  $t$ ，使得对于大小为  $N=10^6$  的任意随机数组，使用递增序列  $1, \lfloor t \rfloor, \lfloor t^2 \rfloor, \lfloor t^3 \rfloor, \lfloor t^4 \rfloor, \dots$  的希尔排序的运行时间最短。给出你能找到的三个最佳  $t$  值以及相应的递增序列。
- [267] 以下练习描述的是各种用于评估排序算法的测试用例。它们的作用是用随机数据帮助你增进对性能特性的理解。随着命令行指定的实验次数的增大，可以和 `SortCompare` 一样在它们中使用 `time()` 函数来得到更精确的结果。在以后的几节中我们会使用这些练习来评估更加复杂的算法。
- 2.1.31 双倍测试。编写一个能够对排序算法进行双倍测试的用例。数组规模  $N$  的起始值为 1000，排序后打印  $N$ 、估计排序用时、实际排序用时以及在  $N$  增倍之后两次用时的比例。用这段程序验证在随机输入模型下插入排序和选择排序的运行时间都是平方级别的。对希尔排序的性能作出猜想并验证你的猜想。
- 2.1.32 运行时间曲线图。编写一个测试用例，使用 `StdDraw` 在各种不同规模的随机输入下将算法的平均运行时间绘制成一张曲线图。可能需要添加一两个命令行参数，请尽量设计一个实用的工具。
- 2.1.33 分布图。对于你为练习 2.1.33 给出的测试用例，在一个无穷循环中调用 `sort()` 方法将由第三个

命令行参数指定大小的数组排序，记录每次排序的用时并使用 `StdDraw` 在图上画出所有平均运行时间，应该能够得到一张运行时间的分布图。

**2.1.34** 罕见情况。编写一个测试用例，调用 `sort()` 方法对实际应用中可能出现困难或极端情况的数组进行排序。比如，数组可能已经是有序的，或是逆序的，数组的所有主键相同，数组的主键只有两种值，大小为 0 或是 1 的数组。

**2.1.35** 不均匀的概率分布。编写一个测试用例，使用非均匀分布的概率来生成随机排列的数据，包括：

- 高斯分布；
- 泊松分布；
- 几何分布；
- 离散分布（一种特殊情况请见练习 2.1.28）。

评估并验证这些输入数据对本节讨论的算法的性能的影响。

268

**2.1.36** 不均匀的数据。编写一个测试用例，生成不均匀的测试数据，包括：

- 一半数据是 0，一半是 1；
- 一半数据是 0, 1/4 是 1, 1/4 是 2, 以此类推；
- 一半数据是 0, 一半是随机 `int` 值。

评估并验证这些输入数据对本节讨论的算法的性能的影响。

**2.1.37** 部分有序。编写一个测试用例，生成部分有序的数组，包括：

- 95% 有序，其余部分为随机值；
- 所有的元素和它们的正确位置的距离都不超过 10；
- 5% 的元素随机分布在整个数组中，剩下的数据都是有序的。

评估并验证这些输入数据对本节讨论的算法的性能的影响。

**2.1.38** 不同类型的元素。编写一个测试用例，生成由多种数据类型元素组成的数组，元素的主键值随机，包括：

- 每个元素的主键均为 `String` 类型（至少长 10 个字符），并含有一个 `double` 值；
- 每个元素的主键均为 `double` 类型，并含有 10 个 `String` 值（每个都至少长 10 个字符）；
- 每个元素的主键均为 `int` 类型，并含有一个 `int[20]` 值

评估并验证这些输入数据对本节讨论的算法的性能的影响。

269



## 第3章 查 找

现代计算机和网络使我们能够访问海量的信息。高效检索这些信息的能力是处理它们的重要前提。本章描述的都是数十年来在广泛应用中经过实践检验的经典查找算法。没有这些算法，现代信息世界的基础计算设施都无从谈起。

我们会使用符号表这个词来描述一张抽象的表格，我们会将信息（值）存储在其中，然后按照指定的键来搜索并获取这些信息。键和值的具体意义取决于不同的应用。符号表中可能会保存很多键和很多信息，因此实现一张高效的符号表也是一项很有挑战性的任务。

符号表有时被称为字典，类似于那本将单词的释义按照字母顺序排列起来的历史悠久的参考书。在英语字典里，键就是单词，值就是单词对应的定义、发音和词源。**符号表有时又叫做索引**，即书本最后将术语按照字母顺序列出以方便查找的那部分。在一本书的索引中，键就是术语，而值就是书中该术语出现的所有页码。

在说明了基本的 API 和两种重要的实现之后，我们会学习用三种经典的数据类型来实现高效的符号表：**二叉查找树、红黑树和散列表**。在总结中我们会看到它们的若干扩展和应用，它们的实现都有赖于我们在本章中将会学到的高效算法。

## 3.1 符号表

符号表最主要的目的就是将一个键和一个值联系起来。用例能够将一个键值对插入符号表并希望在之后能够从符号表的所有键值对中按照键直接找到相对应的值。本章会讲解多种构造这样的数据结构的方法，它们不光能够高效地插入和查找，还可以进行其他几种方便的操作。要实现符号表，我们首先要定义其背后的数据结构，并指明创建并操作这种数据结构以实现插入、查找等操作所需的算法。

查找在大多数应用程序中都至关重要，许多编程环境也因此将符号表实现为高级的抽象数据结构，包括 Java——我们会在 3.5 节中讨论 Java 的符号表实现。表 3.1.1 给出的例子是在一些典型的应用场景中可能出现的键和值。我们马上会看到一些参考性的用例，3.5 节的目的就是向你展示如何在程序中有效地使用符号表。本书中我们还会在其他算法中使用符号表。

**定义。** 符号表是一种存储键值对的数据结构，支持两种操作：插入（put），即将一组新的键值对存入表中；查找（get），即根据给定的键得到相应的值。

表 3.1.1 典型的符号表应用

应 用	查 找 的 目 的	键	值
字典	找出单词的释义	单词	释义
图书索引	找出相关的页码	术语	一串页码
文件共享	找到歌曲的下载地址	歌曲名	计算机 ID
账户管理	处理交易	账户号码	交易详情
网络搜索	找出相关网页	关键字	网页名称
编译器	找出符号的类型和值	变量名	类型和值

[362]

### 3.1.1 API

符号表是一种典型的抽象数据类型（请见第 1 章）：它代表着一组定义清晰的值以及相应的操作，使得我们能够将类型的实现和使用区分开来。和以前一样，我们要用应用程序编程接口（API）来精确地定义这些操作（如表 3.1.2 所示），为数据类型的实现和用例提供一份“契约”。

表 3.1.2 一种简单的泛型符号表 API

public class ST<Key, Value>	
ST()	创建一张符号表
void put(Key key, Value val)	将键值对存入表中(若值为空则将键 key 从表中删除)
Value get(Key key)	获取键 key 对应的值(若键 key 不存在则返回 null)
void delete(Key key)	从表中删去键 key (及其对应的值)
boolean contains(Key key)	键 key 在表中是否有对应的值
boolean isEmpty()	表是否为空
int size()	表中的键值对数量
Iterable<Key> keys()	表中的所有键的集合

在查看用例代码之前，为了保证代码的一致、简洁和实用，我们要先说明具体实现中的几个设计决策。

### 3.1.1.1 泛型

和排序一样，在没有方法时我们没有指定处理对象的类型，而是使用了泛型。对于符号表，我们更过明确地指出了查找操作的键的类型来区分它们的不同角色，而不是像 2.4 节的优先队列那样将键和值本身混在一起。在考虑了这份基本的 API 后（例如，这里没有说明键的有序性），我们会用 Comparable 的对象来扩展典型的用例，这也会为数据类型带来许多新的方法。

### 3.1.1.2 重复的键

任何的所有实现都遵循以下规则：

□ 每个键只对应一个值（表中不允许存在重复的键）；

□ 当用例代码向表中存入的键值对和表中已有的键（及关联的值）冲突时，新的值会替代旧的值。

这些规则定义了 **关联数组** 的抽象形式。你可以将符号表想象成一个数组，键即索引，值即数组的元素。在一个一般的数组中，键就是整型的索引，我们用它来快速访问数组的内容；在一个关联数组（符号表）中，键可以是任意类型，但我们仍然可以用它来快速访问数组的内容。一些编程语言（非 Java）直接支持程序员使用 `st[key]` 来代替 `st.get(key)`，`st[key]=val` 来代替 `st.put(key, val)`，其中 `key`（键）和 `val`（值）都可以是任意类型的对象。[363]

### 3.1.1.3 空（null）键

键不能为空。和 Java 中的许多其他机制一样，使用空键会产生一个运行时异常（请见本节答疑的第三条）。

### 3.1.1.4 空（null）值

我们还规定不允许有空值。这个规定的直接原因是在我们的 API 定义中，当键不存在时 `get()` 方法会返回空，这也意味着任何不在表中的键关联的值都是空。这个规定产生了两个（我们所期望的）结果：第一，我们可以用 `get()` 方法是否返回空来测试给定的键是否存在于符号表中；第二，我们可以将空值作为 `put()` 方法的第二个参数存入表中来实现删除，也就是 3.1.1.5 节的主要内容。

### 3.1.1.5 删除操作

在符号表中，删除的实现可以有两种方法：**延时删除**，也就是将键对应的值置为空，然后在某个时候删去所有值为空的键；或是**即时删除**，也就是立刻从表中删除指定的键。刚才已经说过，`put(key, null)` 是 `delete(key)` 的一种简单的（延时型）实现。而实现（即时型）`delete()` 就是为了替代这种默认的方案。在我们的符号表实现中不会使用默认的方案，而在本书的网站上 `put()` 实现的开头有这样一句防御性代码：

```
if (val == null) { delete(key); return; }
```

这保证了符号表中任何键的值都不为空。为了节省版面我们没有在本书中附上这段代码（我们也不会在调用 `put()` 时使用 `null`）。

### 3.1.1.6 便捷方法

为了用例代码的清晰，我们在 API 中加入了 `contains()` 和 `isEmpty()` 方法，它们的实现如表 3.1.3 所示，只需要一行。

表 3.1.3 默认实现

方 法	默认实现
<code>void delete(Key key)</code>	<code>put(key, null);</code>
<code>boolean contains(key)</code>	<code>return get(key) != null;</code>
<code>boolean isEmpty()</code>	<code>return size() == 0;</code>

为节省篇幅，我们不想重复这些代码，但我们约定它们存在于所有符号表 API 的实现中，用例  
[364] 程序可以自由使用它们。

### 3.1.1.7 迭代

为了方便用例处理表中的所有键值，我们有时会在 API 的第一行加上 `implements Iterable<Key>` 这句话，强制所有实现都必须包含 `iterator()` 方法来返回一个实现了 `hasNext()` 和 `next()` 方法的迭代器，如 1.3 节的栈和队列所述。但是对于符号表我们采用了一个更简单的方法。我们定义了 `keys()` 方法来返回一个 `Iterable<Key>` 对象以方便用例遍历所有的键。这么做是为了和以后的有序符号表的所有方法保持一致，使得用例可以遍历表的键集的一个指定的部分。

### 3.1.1.8 键的等价性

要确定一个给定的键是否存在于符号表中，首先要确立对象等价性的概念。我们在 1.2.5.8 节深入讨论过这一点。在 Java 中，按照约定所有的对象都继承了一个 `equals()` 方法，Java 也为它的标准数据类型例如 `Integer`、`Double` 和 `String` 以及一些更加复杂的类型，如 `File` 和 `URL`，实现了 `equals()` 方法——当使用这些数据类型时你可以直接使用内置的实现。例如，如果 `x` 和 `y` 都是 `String` 类型，当且仅当 `x` 和 `y` 的长度相同且每个位置上的字母都相同时，`x.equals(y)` 返回 `true`。而自定义的键则需要如 1.2 节所述重写 `equals()` 方法。你可以参考我们为 `Date` 类型（请见 1.2.5.8 节）实现的 `equals()` 方法为自己的数据类型实现 `equals()` 方法。和 2.4.4.5 节中讨论的优先队列一样，最好使用不可变的数据类型作为键，否则表的一致性是无法保证的。

[365]

## 3.1.2 有序符号表

典型的应用程序中，键都是 `Comparable` 的对象，因此可以使用 `a.compareTo(b)` 来比较 `a` 和 `b` 两个键。许多符号表的实现都利用了 `Comparable` 接口带来的键的有序性来更好地实现 `put()` 和 `get()` 方法。更重要的是在这些实现中，我们可以认为符号表都会保持键的有序并大大扩展它的 API，根据键的相对位置定义更多实用的操作。例如，假设键是时间，你可能会对最早的或是最晚的键或是给定时间段内的所有键等感兴趣。在大多数情况下用实现 `put()` 和 `get()` 方法背后的数据结构都不难实现这些操作。于是，对于 `Comparable` 的键，在本章中我们实现了表 3.1.4 中的 API。

表 3.1.4 一种有序的泛型符号表的 API

---

<code>public class ST&lt;Key extends Comparable&lt;key&gt;, Value&gt;</code>	
<code>    ST()</code>	创建一张有序符号表
<code>    void put(Key key, Value val)</code>	将键值对存入表中（若值为空则将键 <code>key</code> 从表中删除）
<code>    Value get(Key key)</code>	获取键 <code>key</code> 对应的值（若键 <code>key</code> 不存在则返回空）
<code>    void delete(Key key)</code>	从表中删去键 <code>key</code> （及其对应的值）
<code>    boolean contains(Key key)</code>	键 <code>key</code> 是否存在于表中
<code>    boolean isEmpty()</code>	表是否为空
<code>    int size()</code>	表中的键值对数量
<code>    Key min()</code>	最小的键
<code>    Key max()</code>	最大的键
<code>    Key floor(Key key)</code>	小于等于 <code>key</code> 的最大键
<code>    Key ceiling(Key key)</code>	大于等于 <code>key</code> 的最小键
<code>    int rank(Key key)</code>	小于 <code>key</code> 的键的数量
<code>    Key select(int k)</code>	排名为 <code>k</code> 的键

---

(续)

<code>public class ST&lt;Key extends Comparable&lt;key&gt;, Value&gt;</code>	
<code>void deleteMin()</code>	删除最小的键
<code>void deleteMax()</code>	删除最大的键
<code>int size(Key lo, Key hi)</code>	[lo..hi] 之间键的数量
<code>Iterable&lt;Key&gt; keys(Key lo, Key hi)</code>	[lo..hi] 之间的所有键, 已排序
<code>Iterable&lt;Key&gt; keys()</code>	表中的所有键的集合, 已排序

366

只要你见到类的声明中含有泛型变量 `Key extends Comparable<Key>`, 那就说明这段程序是在实现这份 API, 其中的代码依赖于 `Comparable` 的键并且实现了更加丰富的操作。上面所有这些操作一起为用例定义了一个有序符号表。

### 3.1.2.1 最大键和最小键

对于一组有序的键, 最自然的反应就是查询其中的最大键和最小键。我们在 2.4 节讨论优先队列时已经遇到过这些操作。在有序符号表中, 我们也有方法删除最大键和最小键 (以及它们所关联的值)。有了这些, 符号表就具有了类似于 2.4 节中 `IndexMinPQ` 的能力。主要的区别在于优先队列中可以存在重复的键但符号表中不行, 而且有序符号表支持的操作更多。

### 3.1.2.2 向下取整和向上取整

对于给定的键, 向下取整( `floor` )操作(找出小于等于该键的最大键)和向上取整( `ceiling` )操作(找出大于等于该键的最小键)有时是很有用的。这两个术语来自于实数的取整函数 (对一个实数  $x$  向下取整即为小于等于  $x$  的最大整数, 向上取整则为大于等于  $x$  的最小整数)。

### 3.1.2.3 排名和选择

检验一个新的键是否插入合适位置的基本操作是排名 ( `rank`, 找出小于指定键的键的数量) 和选择 ( `select`, 找出排名为  $k$  的键)。要测试一下你是否完全理解了它们的作用, 请确认对于 0 到 `size() - 1` 的所有  $i$  都有  $i == rank(select(i))$ , 且所有的键都满足 `key == select(rank(key))`。2.5 节中我们在学习排序时已经遇到过对这两种操作的需求了。对于符号表, 我们的挑战是在实现插入、删除和查找的同时快速实现这两种操作。

有序符号表的操作示例如表 3.1.5 所示。

### 3.1.2.4 范围查找

给定范围内 (在两个给定的键之间) 有多少键? 是哪些? 在很多应用中能够回答这些问题并接受两个参数的 `size()` 和 `keys()` 方法都很有用, 特别是在大型数据库中。能够处理这类查询是有序符号表在实践中被广泛应用的重要原因之一。

### 3.1.2.5 例外情况

当一个方法需要返回一个键但表中却没有合适的键可以返回时, 我们约定抛出一个异常 (另一

表 3.1.5 有序符号表的操作示例

	键	值
<code>min()</code> →	<b>09:00:00</b>	Chicago
	09:00:03	Phoenix
	09:00:13	→ Houston
<code>get(09:00:13)</code> →	09:00:59	Chicago
	09:01:10	Houston
<code>floor(09:05:00)</code> →	<b>09:03:13</b>	Chicago
	09:10:11	Seattle
<code>select(7)</code> →	<b>09:10:25</b>	Seattle
	09:14:25	Phoenix
	<b>09:19:32</b>	Chicago
	<b>09:19:46</b>	Chicago
<code>keys(09:15:00, 09:25:00)</code> →	<b>09:21:05</b>	Chicago
	<b>09:22:43</b>	Seattle
	<b>09:22:54</b>	Seattle
	09:25:52	Chicago
<code>ceiling(09:30:00)</code> →	<b>09:35:21</b>	Chicago
	09:36:14	Seattle
<code>max()</code> →	<b>09:37:44</b>	Phoenix
<hr/>		
	<code>size(09:15:00, 09:25:00) = 5</code>	
	<code>rank(09:10:25) = 7</code>	

367

种合理的方法是在这种情况下返回空）。例如，在符号表为空时，`min()`、`max()`、`deleteMin()`、`deleteMax()`、`floor()` 和 `ceiling()` 都会抛出异常，当  $k < 0$  或  $k >= \text{size}()$  时 `select(k)` 也会抛出异常。

### 3.1.2.6 便捷方法

在基础 API 中我们已经见过了 `contains()` 和 `isEmpty()` 方法，为了用例的清晰我们又在 API 中添加了一些冗余的方法。为了节约版面，除非特别声明，我们约定所有有序符号表 API 的实现都含有如表 3.1.6 所示的方法。

表 3.1.6 有序符号表中冗余有序性方法的默认实现

方 法	默认的实现
<code>void deleteMin()</code>	<code>delete(min());</code>
<code>void deleteMax()</code>	<code>delete(max());</code>
<code>int size(Key lo, Key hi)</code>	<pre>if (hi.compareTo(lo) &lt; 0)     return 0; else if (contains(hi))     return rank(hi) - rank(lo) + 1; else     return rank(hi) - rank(lo);</pre>
<code>Iterable&lt;Key&gt; keys()</code>	<code>return keys(min(), max());</code>

### 3.1.2.7 (再谈) 键的等价性

Java 的一条最佳实践就是维护所有 `Comparable` 类型中 `compareTo()` 方法和 `equals()` 方法的一致性。也就是说，任何一种 `Comparable` 类型的两个值 `a` 和 `b` 都要保证 `(a.compareTo(b)==0)` 和 `a.equals(b)` 的返回值相同。为了避免任何潜在的二义性，我们不会在有序符号表的实现中使用 `equals()` 方法。作为替代，我们只会使用 `compareTo()` 方法来比较两个键，即我们用布尔表达式 `a.compareTo(b)==0` 来表示“`a` 和 `b` 相等吗？”一般来说，这样的比较都代表着在符号表中的一次成功查找（找到了 `b`）。和排序算法一样，Java 为许多经常作为键的数据类型提供了标准的 `compareTo()` 方法，为你自定义的数据类型实现一个 `compareTo()` 方法也不困难（参见 2.5 节）。

### 3.1.2.8 成本模型

无论我们是使用 `equals()` 方法（对于符号表的键不是 `Comparable` 对象而言）还是 `compareTo()` 方法（对于符号表的键是 `Comparable` 对象而言），我们使用比较一词来表示将一个符号表条目和一个被查找的键进行比较操作。在大多数的符号表实现中，这个操作都出现在内循环。在少数的例外中，我们则会统计数组的访问次数。

**查找的成本模型。**在学习符号表的实现时，我们会统计比较的次数（等价性测试或是键的相互比较）。在内循环不进行比较（极少）的情况下，我们会统计数组的访问次数。

符号表实现的重点在于其中使用的数据结构和 `get()`、`put()` 方法。在下文中我们不会总是给出其他方法的实现，因为将它们作为练习能够更好地检验你对实现背后的数据结构的理解程度。为了区别不同的实现，我们在特定的符号表实现的类名前加上了描述性前缀。在用例代码中，除非我们想使用一个特定的实现，我们都会使用 `ST` 表示一个符号表实现。在本章和其他章节中，经过学习和讨论过大量符号表的使用和实现后你会慢慢地理解这些 API 的设计初衷。同时我们也会在答疑和练习中讨论算法设计时的更多选择。

### 3.1.3 用例举例

虽然我们会在 3.5 节中详细说明符号表的更多应用，在学习它的实现之前我们还是应该先看看如何使用它。相应地我们这里考察两个用例：一个用来跟踪算法在小规模输入下的行为测试用例，和一个用来寻找更高效的实现的性能测试用例。

#### 3.1.3.1 行为测试用例

为了在小规模的输入下跟踪算法的行为，我们用以下测试用例测试我们对符号表的所有实现。这段代码会从标准输入接受多个字符串，构造一张符号表来将  $i$  和第  $i$  个字符串相关联，然后打印符号表。在本书中我们假设所有的字符串都只有一个字母。一般我们会使用 "S E A R C H E X A M P L E"。按照我们的约定，用例会将键 S 和 0，键 R 和 3 关联起来，等等。但 E 的值是 12（而非 1 或者 6），A 的值为 8（而非 2），因为我们的关联型数组意味着每个键的值取决于最近一次 `put()` 方法的调用。对于符号表的简单实现（无序），用例的输出中键的顺序是不确定的（这和具体实现有关）；对于有序符号表，用例应该将键按顺序打印出来。这是一种索引用例，它是我们将在 3.5 节中讨论的一种重要的符号表应用的一个特殊情况。

测试用例的实现代码如下所示。测试用例的键、值及输出如图 3.1.1 所示。

键	S	E	A	R	C	H	E	X	A	M	P	L	E		
值	0	1	2	3	4	5	6	7	8	9	10	11	12		
简单的符号表测试用例												370			
图 3.1.1 测试用例的键、值和输出												370			
L	11	简单符号表的 (一种可能的) 输出													
P	10	有序符号 表的输出													
M	9														
X	7														
H	5														
C	4														
R	3														
A	8														
E	12														
S	0														

```

public static void main(String[] args)
{
    ST<String, Integer> st;
    st = new ST<String, Integer>();
    for (int i = 0; !StdIn.isEmpty(); i++)
    {
        String key = StdIn.readString();
        st.put(key, i);
    }
    for (String s : st.keys())
        StdOut.println(s + " " + st.get(s));
}

```

#### 3.1.3.2 性能测试用例

`FrequencyCounter` 用例会从标准输入中得到的一列字符串并记录每个（长度至少达到指定的阈值）字符串的出现次数，然后遍历所有键并找出出现频率最高的键。这是一种字典，我们会在 3.5 节中更加详细地讨论这种应用。这个用例回答了一个简单的问题：哪个（不小于指定长度的）单词在一段文字中出现的频率最高？在本章中，我们会用这个用例以及三段文字来进行性能测试：狄更斯的《双城记》中的前五行 (`tinyTale.txt`)，《双城记》全书 (`tale.txt`)，以及一个知名叫做 Leipzig Corpora Collection 的数据库 (`leipzig1M.txt`)，内容为一百万条随机从网络上抽取的句子。例如，这是 `tinyTale.txt` 的内容：

```
% more tinyTale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
```

小型测试输入

这段文字共有 60 个单词，去掉重复的单词还剩 20 个，其中 4 个出现了 10 次（频率最高）。对于这段文字，`FrequencyCounter` 可能会打印出 `it`、`was`、`the` 或者 `of` 中的某一个单词（具体会打印出哪一个取决于符号表的具体实现），以及它出现的频率 10。表 3.1.7 总结了大型测试输入流的性质。

表 3.1.7 大型测试输入流的性质

	TinyTale.txt		tale.txt		leipzig1M.txt	
	单词数	不同的单词数	单词数	不同的单词数	单词数	不同的单词数
所有单词	60	20	135 635	10 679	21 191 455	534 580
长度大于等于 8 的单词	3	3	14 350	5 737	4 239 597	299 593
长度大于等于 10 的单词	2	2	4 582	2 260	1 610 829	165 555

371

`FrequencyCounter` 用例实现过程如下所示。

### 符号表的用例

```
public class FrequencyCounter
{
    public static void main(String[] args)
    {
        int minlen = Integer.parseInt(args[0]); // 最小键长
        ST<String, Integer> st = new ST<String, Integer>();
        while (!StdIn.isEmpty())
        { // 构造符号表并统计频率
            String word = StdIn.readString();
            if (word.length() < minlen) continue; // 忽略较短的单词
            if (!st.contains(word)) st.put(word, 1);
            else st.put(word, st.get(word) + 1);
        }
        // 找出出现频率最高的单词
        String max = " ";
        for (String w : st.keys())
            if (st.get(w) > st.get(max))
                max = w;
        StdOut.println(max);
    }
}
```

```

    st.put(max, 0);

    for (String word : st.keys())
        if (st.get(word) > st.get(max))
            max = word;

    StdOut.println(max + " " + st.get(max));
}

}

```

这个符号表的用例统计了标准输入中各个单词的出现频率，然后将频率最高的单词打印出来。命令行参数指定了表中的键的最短长度。

```

% java FrequencyCounter 1 < tinyTale.txt
it 10

% java FrequencyCounter 8 < tale.txt
business 122

% java FrequencyCounter 10 < leipzig1M.txt
government 24763

```

372

研究符号表处理大型文本的性能要考虑两个方面的因素：首先，每个单词都会被作为键进行搜索，因此处理性能和输入文本的单词总量必然有关；其次，输入的每个单词都会被存入符号表（输入中不重复单词的总数也就是所有键都被插入以后符号表的大小），因此输入流中不同的单词的总数也是相关的。我们需要这两个量来估计 `FrequencyCounter` 的运行时间（作为开始，请见练习 3.1.6）。我们会在学习了一些算法之后再回头说明一些细节，但你应该对类似这样的符号表应用的需求有一个大致的印象。例如，用 `FrequencyCounter` 分析 `leipzig1M.txt` 中长度不小于 8 的单词意味着，在一个含有数以千计的键值对的符号表中进行上百万次的查找，而互联网中的一台服务器可能需要在含有上百万个键值对的表中处理上亿的交易。

这个用例和所有这些例子都提出了一个简单的问题：我们的实现能够在一张用多次 `get()` 和 `put()` 方法构造出的巨型符号表中进行大量的 `get()` 操作吗？如果我们的查找操作不多，那么任意实现都能够满足需要。但没有一个高效的符号表作为基础是无法使用 `FrequencyCounter` 这样的程序来处理大型问题的。`FrequencyCounter` 是一种极为常见的应用的代表，它的这些特性也是许多其他符号表应用的共性：

- 混合使用查找和插入的操作；
- 大量的不同键；
- 查找操作比插入操作多得多；
- 虽然不可预测，但查找和插入操作的使用模式并非随机。

我们的目标就是实现一种符号表来满足这些能够解决典型的实际问题的用例的需要。

下面，我们将会学习两种初级的符号表实现并通过 `FrequencyCounter` 分别评估它们的性能。在之后的几节中，你会学习一些经典的实现，即使对于庞大的输入和符号表它们的性能仍然非常优秀。

373

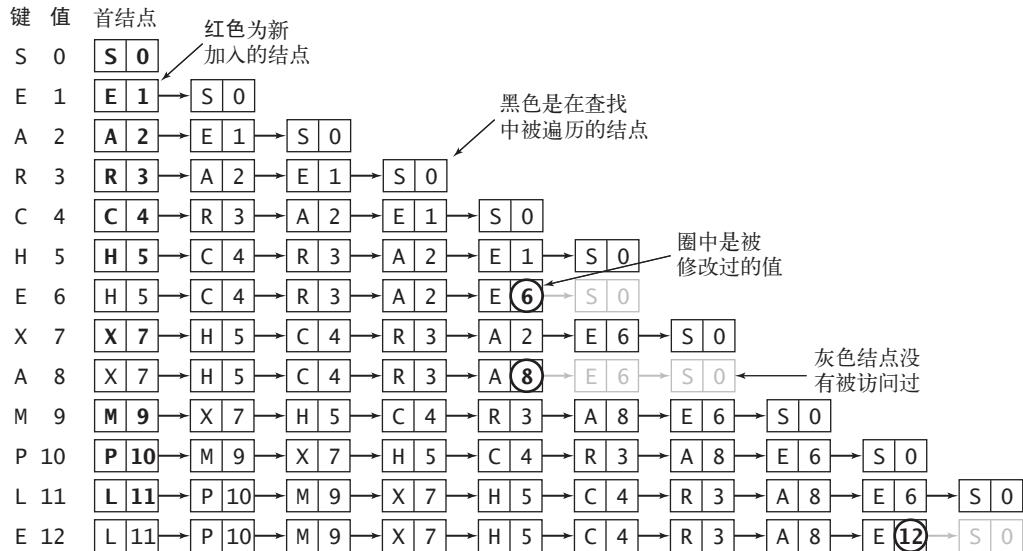
### 3.1.4 无序链表中的顺序查找

符号表中使用的数据结构的一个简单选择是链表，每个结点存储一个键值对，如算法 3.1 中的代码所示。`get()` 的实现即为遍历链表，用 `equals()` 方法比较需被查找的键和每个结点中

的键。如果匹配成功我们就返回相应的值，否则我们返回 `null`。`put()` 的实现也是遍历链表，用 `equals()` 方法比较需被查找的键和每个结点中的键。如果匹配成功我们就用第二个参数指定的值更新和该键相关联的值，否则我们就用给定的键值对创建一个新的结点并将其插入到链表的开头。这种方法也被称为顺序查找：在查找中我们一个一个地顺序遍历符号表中的所有键并使用 `equals()` 方法来寻找与被查找的键匹配的键。

算法 3.1 ( `SequentialSearchST` ) 用链表实现了符号表的基本 API，我们在第 1 章中的基础数据结构中学习过它。这里我们将 `size()`、`keys()` 和即时型的 `delete()` 方法留做练习。这些练习能够巩固并加深你对链表和符号表的基本 API 的理解。

这种基于链表的实现能够用于和我们的用例类似的、需要大型符号表的应用吗？我们已经说过，分析符号表算法比分析排序算法更困难，因为不同的用例所进行的操作序列各不相同。对于 `FrequencyCounter`，最常见的情形是虽然查找和插入的使用模式是不可预测的，但它们的使用肯定不是随机的。因此我们主要研究最坏情况下的性能。为了方便，我们使用命中表示一次成功的查找，未命中表示一次失败的查找。使用基于链表的符号表的索引用例的轨迹如图 3.1.2 所示。



374

图 3.1.2 使用基于链表的符号表的索引用例的轨迹（另见彩插）

### 算法 3.1 顺序查找（基于无序链表）

```
public class SequentialSearchST<Key, Value>
{
    private Node first;           // 链表首结点

    private class Node
    { // 链表结点的定义
        Key key;
        Value val;
    }
}
```

```

Node next;

public Node(Key key, Value val, Node next)
{
    this.key = key;
    this.val = val;
    this.next = next;
}

public Value get(Key key)
{
    // 查找给定的键，返回相关联的值
    for (Node x = first; x != null; x = x.next)
        if (key.equals(x.key))
            return x.val; // 命中
    return null; // 未命中
}

public void put(Key key, Value val)
{
    // 查找给定的键，找到则更新其值，否则在表中新建结点
    for (Node x = first; x != null; x = x.next)
        if (key.equals(x.key))
            { x.val = val; return; } // 命中，更新
    first = new Node(key, val, first); // 未命中，新建结点
}
}

```

符号表的实现使用了一个私有内部 `Node` 类来在链表中保存键和值。`get()` 的实现会顺序地搜索链表查找给定的键（找到则返回相关联的值）。`put()` 的实现也会顺序地搜索链表查找给定的键，如果找到则更新相关联的值，否则它会用给定的键值对创建一个新的结点并将其插入到链表的开头。`size()`、`keys()` 和即时型的 `delete()` 方法的实现留做练习。

375

**命题 A。** 在含有  $N$  对键值的基于（无序）链表的符号表中，未命中的查找和插入操作都需要  $N$  次比较。命中的查找在最坏情况下需要  $N$  次比较。特别地，向一个空表中插入  $N$  个不同的键需要  $\sim N^2/2$  次比较。

**证明。** 在表中查找一个不存在的键时，我们会将表中的每个键和给定的键比较。因为不允许出现重复的键，每次插入操作之前我们都需要这样查找一遍。

**推论。** 向一个空表中插入  $N$  个不同的键需要  $\sim N^2/2$  次比较。

查找一个已经存在的键并不需要线性级别的时间。一种度量方法是查找表中的每个键，并将总时间除以  $N$ 。在查找表中的每个键的可能性都相同的情况下时，这个结果就是一次查找平均所需的比较数。我们将它称为随机命中。尽管符号表用例的查找模式不太可能是随机的，这个模型也总能适应得很好。我们很容易就可以得到随机命中所需的平均比较次数为  $\sim N/2$ ：算法 3.1 中的 `get()` 方法查找第一个键需要 1 次比较，查找第二个键需要 2 次比较，如此这般，平均比较次数为  $(1+2+\dots+N)/N=(N+1)/2\sim N/2$ 。

这些分析完全证明了基于链表的实现以及顺序查找是非常低效的，无法满足 `FrequencyCounter` 处理庞大输入问题的需求。比较的总次数和查找次数与插入次数的乘积成正比。对于《双城记》这个数字大于  $10^9$ ，而对于 Leipzig Corpora 数据库这个数字大于  $10^{14}$ 。

按照惯例，为了验证分析结果我们需要进行一些实验。这里我们用 `FrequencyCounter` 以及命令行参数 8 来分析 `tale.txt`。这将需要 14 350 次 `put()`（已经说过，输入中的每个单词都需要一次 `put()` 操作来更新它的出现频率，`contains()` 方法的调用是可以避免的，这里忽略了它的成本）。符号表将包含 5737 个键，也就是说大约三分之一的操作都将表增大了，其余操作为查找。为了将性能可视化我们使用了 `VisualAccumulator`（请见表 1.2.14）将每次 `put()` 操作转换为两个点：对于第  $i$  次 `put()` 操作，我们会在横坐标为  $i$ ，纵坐标为该次操作所进行的比较次数的位置画一个灰点，以及横坐标为  $i$ ，纵坐标为前  $i$  次 `put()` 操作累计所需的平均比较次数的位置画一个黑点，如图 3.1.3 所示。和所有科学实验数据一样，这其中包含了很多信息供我们研究（这张图含有 14 350 个灰点和 14 350 个黑点）。这里，我们的主要兴趣在于这张表证实了我们关于 `put()` 平均需要访问半条链表的猜想。虽然实际的数据比一半稍少，但对这个事实（以及图表曲线的形状）最好的解释应该是应用的特性，而非算法（请见练习 3.1.36）。

尽管某个具体用例的性能特点可能是复杂的，但只要使用我们准备的文本或者随机有序输入以及我们在第 1 章中介绍的 `DoublingTest` 程序，我们还是能够轻松估计出 `FrequencyCounter` 的性能并测试验证的。我们将这些测试留给练习和接下来将要学习的更加复杂的实现。如果你并不觉得我们需要更快的实现，请一定完成这些练习！（或者用 `FrequencyCounter` 调用 `SequentialSearchST` 来处理 `leipzig1M.txt`！）

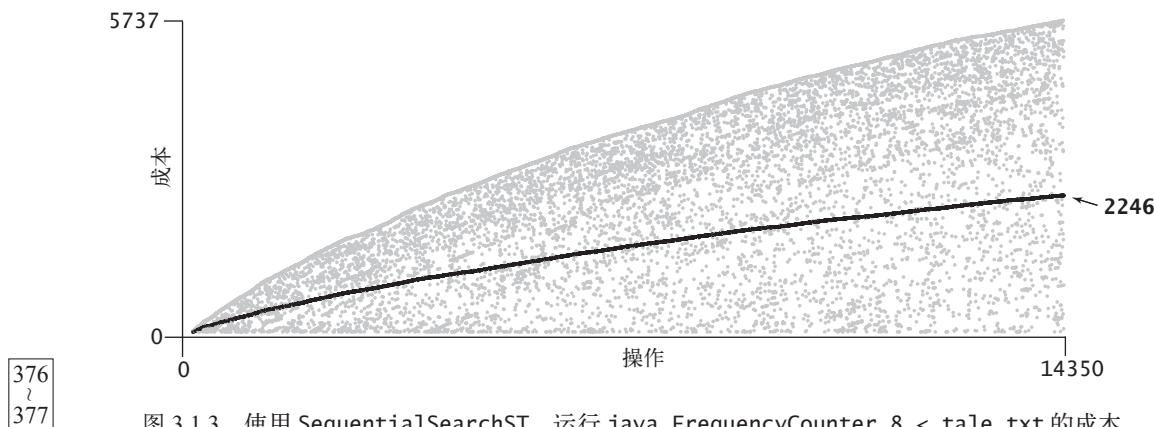


图 3.1.3 使用 `SequentialSearchST`, 运行 `java FrequencyCounter 8 < tale.txt` 的成本

### 3.1.5 有序数组中的二分查找

下面我们要学习有序符号表 API 的完整实现。它使用的数据结构是一对平行的数组，一个存储

键一个存储值。算法 3.2 (BinarySearchST) 可以保证数组中 Comparable 类型的键有序，然后使用数组的索引来高效地实现 get() 和其他操作。

这份实现的核心是 rank() 方法，它返回表中小于给定键的键的数量。对于 get() 方法，只要给定的键存在于表中，rank() 方法就能够精确地告诉我们在哪里能够找到它（如果找不到，那它肯定就不在表中了）。

对于 put() 方法，只要给定的键存在于表中，rank() 方法就能够精确地告诉我们到哪里去更新它的值，以及当键不在表中时将键存储到表的何处。我们将所有更大的键向后移动一格来腾出位置（从后向前移动）并将给定的键值对分别插入到各自数组中的合适位置。结合我们测试用例的轨迹来研究 BinarySearchST 也是学习这种数据结构的好方法。

这段代码为键和值使用了两个数组（另一种方式请见练习 3.1.12）。和我们在第 1 章中对泛型的栈和队列的实现一样，这段代码也需要创建一个 Key 类型的 Comparable 对象的数组和一个 Value 类型的 Object 对象的数组，并在构造函数中将它们转化回 Key[] 和 Value[]。和以前一样，我们可以动态调整数组，使得用例无需担心数组大小（请注意，你会发现这种方法对于大数组实在是太慢了）。

使用基于有序数组的符号表实现的索引用例的轨迹如表 3.1.8 所示。

表 3.1.8 使用基于有序数组的符号表实现的索引用例的轨迹

键	值	keys[]										N	vals[]									
		0	1	2	3	4	5	6	7	8	9		0	1	2	3	4	5	6	7	8	9
S	0	S										1	0									
E	1	E	S									2	1	0								
A	2	A	E	S								3	2	1	0							
R	3	A	E	R	S							4	2	1	3	0						
C	4	A	C	E	R	S						5	2	4	1	3	0					
H	5	A	C	E	H	R	S					6	2	4	1	5	3	0				
E	6	A	C	E	H	R	S					6	2	4	6	5	3	0				
X	7	A	C	E	H	R	S	X				7	2	4	6	5	3	0	7			
A	8	A	C	E	H	R	S	X				7	8	4	6	5	3	0	7			
M	9	A	C	E	H	M	R	S	X			8	8	4	6	5	9	3	0	7		
P	10	A	C	E	H	M	P	R	S	X		9	8	4	6	5	9	10	3	0	7	
L	11	A	C	E	H	L	M	P	R	S	X	10	8	4	6	5	11	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	12	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

378

### 算法 3.2 二分查找（基于有序数组）

```
public class BinarySearchST<Key extends Comparable<Key>, Value>
{
    private Key[] keys;
    private Value[] vals;
    private int N;
    public BinarySearchST(int capacity)
    {   // 调整数组大小的标准代码请见算法 1.1
        keys = (Key[]) new Comparable[capacity];
        vals = (Value[]) new Object[capacity];
    }
}
```

```

    }
    public int size()
    {   return N;  }
    public Value get(Key key)
    {
        if (isEmpty()) return null;
        int i = rank(key);
        if (i < N && keys[i].compareTo(key) == 0) return vals[i];
        else                                         return null;
    }

    public int rank(Key key)
    // 请见算法3.2(续1)

    public void put(Key key, Value val)
    {   // 查找键, 找到则更新值, 否则创建新的元素
        int i = rank(key);
        if (i < N && keys[i].compareTo(key) == 0)
        {   vals[i] = val; return;  }
        for (int j = N; j > i; j--)
        {   keys[j] = keys[j-1]; vals[j] = vals[j-1];  }
        keys[i] = key; vals[i] = val;
        N++;
    }

    public void delete(Key key)
    // 该方法的实现请见练习3.1.16
}

```

这段符号表的实现用两个数组来保存键和值。和1.3节中基于数组的栈一样, `put()`方法会在插入新元素前将所有较大的键向后移动一格。这里省略了调整数组大小部分的代码。

379

### 3.1.5.1 二分查找

我们使用有序数组存储键的原因是, 第1章中作为例子出现的经典二分查找法能够根据数组的索引大大减少每次查找所需的比较次数。我们会使用有序索引数组来标识被查找的键可能存在的子数组的大小范围。在查找时, 我们先将被查找的键和子数组的中间键比较。如果被查找的键小于中间键, 我们就在左子数组中继续查找, 如果大于我们就在右子数组中继续查找, 否则中间键就是我们要找的键。算法3.2(续1)中实现`rank()`方法的代码使用了刚才讨论的二分查找法。这个实现值得我们仔细研究。作为开始, 我们来看这段等价的递归代码。

调用这里的`rank(key, 0, N-1)`所进行的比较和调用算法3.2(续1)的实现所进行的比较完全相同。但如1.1节中讨论的, 这个版本更好地暴露了算法的结构。递归的`rank()`保留了以下性质:

- 如果表中存在该键, `rank()`应该返回该键的位置, 也就是表中小于它的键的数量;

```

public int rank(Key key, int lo, int hi)
{
    if (hi < lo) return lo;
    int mid = lo + (hi - lo) / 2;
    int cmp = key.compareTo(keys[mid]);
    if      (cmp < 0)
        return rank(key, lo, mid-1);
    else if (cmp > 0)
        return rank(key, mid+1, hi);
    else return mid;
}

```

递归的二分查找

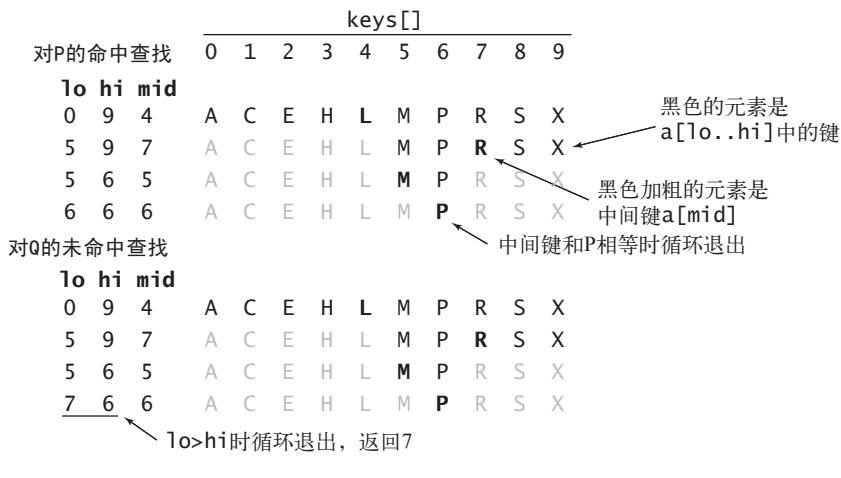
□ 如果表中不存在该键，`rank()` 还是应该返回表中小于它的键的数量。

好好想想算法 3.2（续 1）中非递归的 `rank()` 为什么能够做到这些（你可以证明两个版本的等价性，或者直接证明非递归版本中的循环在结束时 `lo` 的值正好等于表中小于被查找的键的键的数量），所有程序员都能从这些思考中有所收获。（提示：`lo` 的初始值为 0，且永远不会变小）

### 算法 3.2（续 1）基于有序数组的二分查找（迭代）

```
public int rank(Key key)
{
    int lo = 0, hi = N-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if (cmp < 0) hi = mid - 1;
        else if (cmp > 0) lo = mid + 1;
        else return mid;
    }
    return lo;
}
```

该方法实现了正文所述的经典算法来计算小于给定键的键的数量。它首先将 `key` 和中间键比较，如果相等则返回其索引；如果小于中间键则在左半部分查找；大于则在右半部分查找。



在有序数组中使用二分法查找排名的轨迹

### 算法 3.2（续 2）基于二分查找的有序符号表的其他操作

```
public Key min()
{ return keys[0]; }

public Key max()
{ return keys[N-1]; }

public Key select(int k)
{ return keys[k]; }
```

```

public Key ceiling(Key key)
{
    int i = rank(key);
    return keys[i];
}

public Key floor(Key key)
// 请见练习3.1.17

public Key delete(Key key)
// 请见练习3.1.16

public Iterable<Key> keys(Key lo, Key hi)
{
    Queue<Key> q = new Queue<Key>();
    for (int i = rank(lo); i < rank(hi); i++)
        q.enqueue(keys[i]);
    if (contains(hi))
        q.enqueue(keys[rank(hi)]);
    return q;
}

```

这些方法，以及练习 3.1.16 和练习 3.1.17，组成了我们对使用二分查找的有序符号表的完整实现。`min()`、`max()` 和 `select()` 方法都很简单，只需按照给定的位置从数组中返回相应的值即可。`rank()` 方法实现了二分查找，是其他方法的基石。`floor()` 和 `delete()` 方法虽然也不难，但稍微复杂一些，在此留做练习。

### 3.1.5.2 其他操作

因为键被保存在有序数组中，算法 3.2（续 2）中和顺序有关的大多数操作都一目了然。例如，调用 `select(k)` 就相当于返回 `keys[k]`。我们将 `delete()` 和 `floor()` 留做练习。你应该研究一下 `ceiling()` 和带两个参数的 `keys()` 方法的实现，并完成练习来巩固和加深你对有序符号表的 API 及其实现的理解。

380  
382

### 3.1.6 对二分查找的分析

`rank()` 的递归实现还能够让我们立即得到一个结论：二分查找很快，因为递归关系可以说明算法所需比较次数的上界。

**命题 B。** 在  $N$  个键的有序数组中进行二分查找最多需要  $(\lg N + 1)$  次比较（无论是否成功）。

**证明。** 这里的分析和对归并排序的分析（第 2 章的命题 F）类似（但相对简单）。令  $C(N)$  为在大小为  $N$  的符号表中查找一个键所需进行的比较次数。显然我们有  $C(0)=0$ ， $C(1)=1$ ，且对于  $N>0$  我们可以写出一个和递归方法直接对应的归纳关系式：

$$C(N) \leq C(\lfloor N/2 \rfloor) + 1$$

无论查找会在中间元素的左侧还是右侧继续，子数组的大小都不会超过  $\lfloor N/2 \rfloor$ ，我们需要一次比较来检查中间元素和被查找的键是否相等，并决定继续查找左侧还是右侧的子数组。当  $N$  为 2 的幂减 1 时 ( $N=2^n-1$ )，这种递推很容易。首先，因为  $\lfloor N/2 \rfloor=2^{n-1}-1$ ，所以我们有：

$$C(2^n-1) \leq C(2^{n-1}-1) + 1$$

用这个公式代换不等式右边的第一项可得：

$$C(2^n - 1) \leq C(2^{n-2} - 1) + 1 + 1$$

将上面这一步重复  $n-2$  次可得：

$$C(2^n - 1) \leq C(2^0) + n$$

最后的结果即：

$$C(N) = C(2^n) \leq n + 1 < \lg N + 1$$

对于一般的  $N$ , 确切的结论更加复杂, 但不难通过以上论证推广得到 (请见练习 3.1.20)。二分查找所需时间必然在对数范围之内。

刚才给出的实现中, `ceiling()` 只是调用了一次 `rank()`, 而接受两个参数的默认 `size()` 方法调用了两次 `rank()`, 因此这份证明也保证了这些操作 (包括 `floor()`) 所需的时间最多是对数级别的 (`min()`、`max()` 和 `select()` 操作所需的时间都是常数级别的)。

尽管能够保证查找所需的时间是对数级别的, `BinarySearchST` 仍然无法支持我们用类似 `FrequencyCounter` 的程序来处理大型问题, 因为 `put()` 方法还是太慢了。二分查找减少了比较的次数但无法减少运行所需时间, 因为它无法改变以下事实: 在键是随机排列的情况下, 构造一个基于有序数组的符号表所需要访问数组的次数是数组长度的平方级别 (在实际情况下键的排列虽然不是随机的, 但仍然很好地符合这个模型)。`BinarySearchST` 的操作的成本如表 3.1.9 所示。

**命题 B (续)**。向大小为  $N$  的有序数组中插入一个新的元素在最坏情况下需要访问  $\sim 2N$  次数组, 因此向一个空符号表中插入  $N$  个元素在最坏情况下需要访问  $\sim N^2$  次数组。

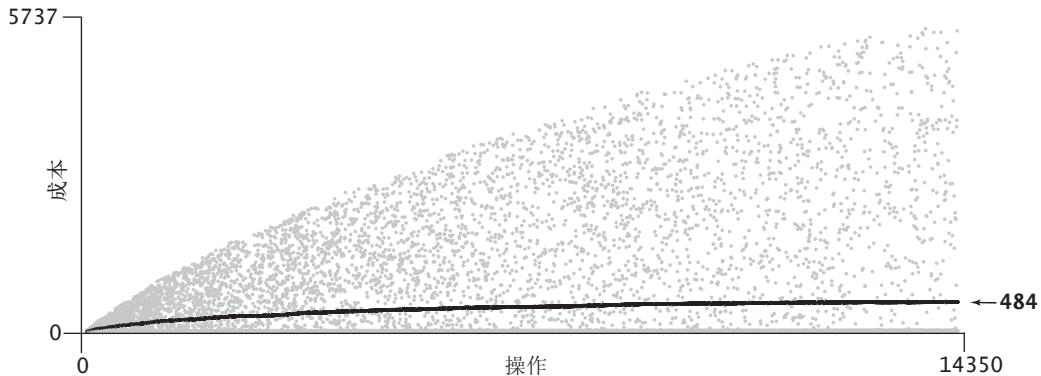
**证明**。同命题 A。

表 3.1.9 `BinarySearchST` 的操作的成本

方法	运行所需时间的增长数量级	383
<code>put()</code>	$N$	
<code>get()</code>	$\log N$	
<code>delete()</code>	$N$	
<code>contains()</code>	$\log N$	
<code>size()</code>	1	
<code>min()</code>	1	
<code>max()</code>	1	
<code>floor()</code>	$\log N$	
<code>ceiling()</code>	$\log N$	
<code>rank()</code>	$\log N$	
<code>select()</code>	1	
<code>deleteMin()</code>	$N$	
<code>deleteMax()</code>	1	

对于含有  $10^4$  个不同键的《双城记》, 构建符号表需要访问数组约  $10^8$  次; 而对于含有  $10^6$  个不同键的 Leipzig 项目则需要访问数组  $10^{11}$  次。虽然现代计算机可勉强实现, 但这样的成本还是过高了。

回头看看 `FrequencyCounter` 在参数为 8 时 `put()` 操作的性能, 我们可以看到平均情况下的比较次数 (包括访问数组的次数) 从 `SequentialSearchST` 的 2246 次降低到了 `BinarySearchST` 的 484 次 (如图 3.1.4 所示)。这比我们在分析中预测的还要更好, 额外的部分可能能够再次通过应用的性质得到解释 (请见练习 3.1.36)。这次改进令人印象深刻, 但你会看到, 我们还能做得更好。



384

图 3.1.4 使用 BinarySearchST, 运行 java FrequencyCounter 8 &lt; tale.txt 的成本

### 3.1.7 预览

一般情况下二分查找都比顺序查找快得多，它也是众多实际应用程序的最佳选择。对于一个静态表（不允许插入）来说，将其在初始化时就排序是值得的，如第1章中的二分查找所示（请见表1.2.15）。即使查找前所有的键值对已知（这在应用程序中是一种常见的情况），为BinarySearchST添加一个能够初始化并将符号表排序的构造函数也是有意义的（请见练习3.1.12）。当然，二分查找也不适合很多应用。例如，它无法处理Leipzig Corpora数据库，因为查找和插入操作是混合进行的，而且符号表也太大了。如我们所强调的那样，现代应用需要同时能够支持高效的查找和插入两种操作的符号表实现。也就是说，我们需要在构造庞大的符号表的同时能够任意插入（也许还有删除）键值对，同时也要能够完成查找操作。

表3.1.10给出了本节中介绍的符号表的初级实现的性能特点。表中给出的是总成本中的最高级项（对于二分查找是数组的访问次数，对于其他则是比较次数），即运行时间的增长数量级。

表 3.1.10 简单的符号表实现的成本总结

算法（数据结构）	最坏情况下的成本 ( $N$ 次插入后)		平均情况下的成本 ( $N$ 次随机插入后)		是否高效地支持有序性相关的操作
	查 找	插 入	查 找	插 入	
顺序查找(无序链表)	$N$	$N$	$N/2$	$N$	否
二分查找(有序数组)	$\lg N$	$2N$	$\lg N$	$N$	是

核心的问题在于我们能否找到能够同时保证查找和插入操作都是对数级别的算法和数据结构。答案是令人兴奋的“可以”！这个答案也正是本章的重点所在。和第2章讨论的高效排序算法一样，能够高效地查找和插入的符号表是算法领域对世界最重要的贡献之一，也是我们今天能够享受的丰富计算性基础设施的开发基础。

我们如何能够实现这个目标呢？要支持高效的插入操作，我们似乎需要一种链式结构。但单链接的链表是无法使用二分查找法的，因为二分查找的高效来自于能够快速通过索引取得任何子数组的中间元素（但得到一条链表的中间元素的唯一方法只能是沿链表遍历）。为了将二分查找的效率和链表的灵活性结合起来，我们需要更加复杂的数据结构。能够同时拥有两者的就是二叉查找树，它也是我们下面两节的主题。我们会将散列表留到3.4节中讨论。

385

在本章中我们会学习 6 种符号表的实现，这里我们先给出一个简单的预览。表 3.1.11 包含一系列数据结构以及它们适用和不适用于某个应用场景的原因，按照我们学习它们的先后顺序排列。

表 3.1.11 符号表的各种实现的优缺点

使用的数据结构	实 现	优 点	缺 点
链表(顺序查找)	SequentialSearchST	适用于小型问题	对于大型符号表很慢
有序数组(二分查找)	BinarySearchST	最优的查找效率和空间需求，能够进行有序性相关操作	插入操作很慢
二叉查找树	BST	实现简单，能够进行有序性相关操作	没有性能上界的保证 链接需要额外的空间
平衡二叉查找树	RedBlackBST	最优的查找和插入效率，能够进行有序性相关操作	链接需要额外的空间
散列表	SeparateChainHashST LinearProbingHashST	能够快速地查找和插入常见类型的数据	需要计算每种类型的数据的散列无法进行有序性相关的操作 链接和空结点需要额外的空间

在学习中我们会仔细了解每种算法和实现的各种性质，这里的简单特性是为了帮助你在学习它们的同时能够从全局的高度来理解它们。一句话，我们有若干种高效的符号表实现，它们能够并且已经被应用于无数程序之中了。

386

## 答疑

- 问 为什么符号表不像 2.4 节中优先队列那样使用一个 `Comparable` 的 `Item` 类型，而是对于键和值使用不同的数据类型？
- 答 这的确是一种可行的办法。这两者代表了将键和值关联起来的两种不同方式——我们可以构造一种将键包含在其中的数据结构来隐式关联键值或是显式地将键和值区分开来。对于符号表，我们选择突出关联数组的抽象形式。同时也请注意，符号表的用例在查找时只会指定一个键，而非一个键值对。
- 问 为什么要用 `equals()`？为什么不一直使用 `compareTo()`？
- 答 并不是所有的数据产生的键值对都能够进行比较，尽管有时候将它们保存在符号表可以。举一个比较极端的例子，你可能会用一幅照片或者一首歌作为键，但没法比较它们，只能知道它们是否相等（也要花点儿工夫）。
- 问 为什么键的值不能为空（`null`）？
- 答 因为我们会用 `Key` 调用 `compareTo()` 或者 `equals()` 方法，因此我们假设它是一个 `Object`。但是当 `a` 为 `null` 时 `a.compareTo(b)` 会抛出一个空指针异常。如果能消除这种可能性，用例的代码能够更简单。
- 问 为什么不和排序一样使用一个类似于 `less()` 的方法？
- 答 在符号表中等价性比较特殊，因此我们还需要一个方法来测试等价性。为了避免增加本质上功能相同的方法，我们使用了 Java 内置的 `equals()` 和 `compareTo()`。
- 问 在 `BinarySearchST` 中的类型转换之前，为什么不将 `key[]` 和 `val[]` 一样声明为 `Object[]`（而是 `Comparable[]`）？

答 问得好。如果你这么做，你会得到一个 `ClassCastException`，因为键只能是 `Comparable` 的（以保证 `key[]` 中的元素都有 `compareTo()` 方法）。因此将 `key[]` 声明为 `Comparable[]` 是必需的。深入程序语言的设计细节来解释这里的原因可能会有些跑题。在本书所有使用泛型的 `Comparable` 对象和数组的代码中我们都会照此办理。

387 问 如果我们需要将多个值关联到同一个键怎么办？例如，如果我们在应用程序中用 `Date` 日期作为键，那不会需要处理重复的键吗？

答 可能会，也可能不会。例如，两列火车不可能同时在同一条轨道上到达同一个车站（但它们可以在不同的铁轨上同时到站）。处理这种情形有两个办法：用其他信息来消除重复或者使用 `Queue` 类型来存储所有有相同键的值。我们会在 3.5 节中详细讨论符号表的应用。

问 3.1.7 节中将表预排序的想法看起来是个好主意，为什么把它留作一道练习（请见练习 3.1.12）？

答 的确，在某些应用中它确实是最佳的选择。但在一个希望实现快速查找的数据结构中为了“图方便”而加入一个低效的插入方法会变成一个性能陷阱，因为一个普通用例可能会在一张很大的表中混合使用查找和插入操作却发现运行所需的时间是平方级别的。这种陷阱太常见了，因此当你使用他人开发的软件，尤其是接口繁多时，你应该加倍小心。当对象含有大量“便捷”方法而导致到处都是性能陷阱，而用例却可能认为所有的方法都同样高效时，这个问题就非常严重了。Java 的 `ArrayList` 类就是这样的一个例子（请见练习 3.5.27）。

388

## 练习

3.1.1 编写一段程序，创建一张符号表并建立字母成绩和数值分数的对应关系，如下表所示。从标准输入读取一系列字母成绩，计算并打印 GPA（字母成绩对应的分数的平均值）。

A+	A	A-	B+	B	B-	C+	C	C-	D	F
4.33	4.00	3.67	3.33	3.00	2.67	2.33	2.00	1.67	1.00	0.00

3.1.2 开发一个符号表的实现 `ArrayST`，使用（无序）数组来实现我们的基本 API。

3.1.3 开发一个符号表的实现 `OrderedSequentialSearchST`，使用有序链表来实现我们的有序符号表 API。

3.1.4 开发抽象数据类型 `Time` 和 `Event` 来处理表 3.1.5 中的例子中的数据。

3.1.5 实现 `SequentialSearchST` 中的 `size()`、`delete()` 和 `keys()` 方法。

3.1.6 用输入中的单词总数  $W$  和不同单词总数  $D$  的函数给出 `FrequencyCounter` 调用的 `put()` 和 `get()` 方法的次数。

3.1.7 对于  $N=10, 10^2, 10^3, 10^4, 10^5$  和  $10^6$ ，在  $N$  个小于 1000 的随机非负整数中 `Frequency Counter` 平均能够找到多少个不同的键？

3.1.8 在《双城记》中，使用频率最高的长度大于等于 10 的单词是什么？

3.1.9 在 `FrequencyCounter` 中添加追踪 `put()` 方法的最后一次调用的代码。打印出最后插入的那个单词以及在此之前总共从输入中处理了多少个单词。用你的程序处理 `tale.txt` 中长度分别大于等于 1、8 和 10 的单词。

3.1.10 给出用 `SequentialSearchST` 将键 `E A S Y Q U E S T I O N` 插入一个空符号表的过程的轨迹。一共进行了多少次比较？

3.1.11 给出用 `BinarySearchST` 将键 `E A S Y Q U E S T I O N` 插入一个空符号表的过程的轨迹。一

共进行了多少次比较？

- 3.1.12 修改 `BinarySearchST`，用一个 `Item` 对象的数组而非两个平行数组来保存键和值。添加一个构造函数，接受一个 `Item` 的数组为参数并将其归并排序。389
- 3.1.13 对于一个会随机混合进行  $10^3$  次 `put()` 和  $10^6$  次 `get()` 操作的应用程序，你会使用本节中的哪种符号表的实现？说明理由。
- 3.1.14 对于一个会随机混合进行  $10^6$  次 `put()` 和  $10^3$  次 `get()` 操作的应用程序，你会使用本节中的哪种符号表的实现？说明理由。
- 3.1.15 假设在一个 `BinarySearchST` 的用例程序中，查找操作的次数是插入操作的 1000 倍。当分别进行  $10^3$ 、 $10^6$  和  $10^9$  次查找时，请估计插入操作在总耗时中的比例。
- 3.1.16 为 `BinarySearchST` 实现 `delete()` 方法。
- 3.1.17 为 `BinarySearchST` 实现 `floor()` 方法。
- 3.1.18 证明 `BinarySearchST` 中 `rank()` 方法的实现的正确性。
- 3.1.19 修改 `FrequencyCounter`，打印出现频率最高的所有单词，而非其中之一。提示：请用 `Queue`。
- 3.1.20 补全命题 B 的证明（证明  $N$  的一般情况）。提示：先证明  $C(N)$  的单调性，即对于所有的  $N > 0$ ， $C(N) \leq C(N+1)$ 390

## 提高题

- 3.1.21 内存使用。基于 1.4 节中的假设，对于  $N$  对键值比较 `BinarySearchST` 和 `SequentialSearchST` 的内存使用情况。不需要记录键值本身占用的内存，只统计它们的引用。对于 `BinarySearchST`，假设数组大小可以动态调整，数组中被占用的空间比例为 25% ~ 100%。
- 3.1.22 自组织查找。自组织查找指的是一种能够将数组元素重新排序使得被访问频率较高的元素更容易被找到的查找算法。请修改你为练习 3.1.2 给出的答案，在每次查找命中时：将被找到的键值对移动到数组的开头，将所有中间的键值对向右移动一格。这个启发式的过程被称为前移编码。
- 3.1.23 二分查找的分析。请证明对于大小为  $N$  的符号表，一次二分查找所需的最大比较次数正好是  $N$  的二进制表示的位数，因为右移一位的操作会将二进制的  $N$  变为二进制的  $[N/2]$ 。
- 3.1.24 插值法查找。假设符号表的键支持算术操作（例如，它们可能是 `Double` 或者 `Integer` 类型的值）。编写一个二分查找来模拟字典的行为，例如当单词的首字母在字母表的开头时我们也会在字典的前半部分进行查找。具体来说，设  $k_{lo}$  为符号表的第一个键， $k_{hi}$  为符号表的最后一个键，当要查找  $k_x$  时，先和  $\lfloor k_x - k_{lo} \rfloor / (k_{hi} - k_{lo})$  进行比较，而非取中间元素。用 `SearchCompare`<sup>①</sup> 调用 `FrequencyCounter` 来比较你的实现和 `BinarySearchST` 的性能。
- 3.1.25 缓存。因为默认的 `contains()` 的实现中调用了 `get()`，所以 `FrequencyCounter` 的内循环会将同一个键查找两三遍：

```
if (!st.contains(word)) st.put(word, 1);
else st.put(word, st.get(word) + 1);
```

为了能够提高这样的用例代码的效率，我们可以用一种叫缓存的技术手段，即将访问最频繁的键的位置保存在一个变量中。修改 `SequentialSearchST` 和 `BinarySearchST` 来实现这个点子。391

<sup>①</sup> `SearchCompare` 应该是一个类似于 `SortCompare` 的类，但实际上正文中并没有任何关于这个 `SearchCompare` 类的内容。——译者注

- 3.1.26 基于字典的频率统计。修改 `FrequencyCounter`，接受一个字典文件作为参数，统计标准输入中出现在字典中的单词的频率，并将单词和频率打印为两张表格，一张按照频率高低排序，一张按照字典顺序排序。
- 3.1.27 小符号表。假设一段 `BinarySearchST` 的用例插入了  $N$  个不同的键并会进行  $S$  次查找。当构造表的成本和所有查找的总成本相同时，给出  $S$  的增长数量级。
- 3.1.28 有序的插入。修改 `BinarySearchST`，使得插入一个比当前所有键都大的键只需要常数时间（这样在构造符号表时有序地使用 `put()` 插入键值对就只需要线性时间了）
- 3.1.29 测试用例。编写一段测试代码 `TestBinarySearch.java` 来测试正文中 `min()`、`max()`、`floor()`、`ceiling()`、`select()`、`rank()`、`deleteMin()`、`deleteMax()` 和 `keys()` 的实现。可以参考 3.1.3.1 节的索引用例，添加代码使其在适当的情况下接受更多的命令行参数。
- 3.1.30 验证。向 `BinarySearchST` 中加入断言（`assert`）语句，在每次插入和删除数据后检查算法的有效性和数据结构的完整性。例如，对于每个索引必有 `i==rank(select(i))` 且数组应该总是有序的。

392

## 实验题

- 3.1.31 性能测试。编写一段性能测试程序，先用 `put()` 构造一张符号表，再用 `get()` 进行访问，使得表中的每个键平均被命中 10 次，且有大致相同次数的未命中访问。键为长度从 2 到 50 不等的随机字符串。重复这样的测试若干遍，记录每遍的运行时间，打印平均运行时间或将它们绘制成图。
- 3.1.32 练习。编写一段练习程序，用困难或者极端的但在实际应用中可能出现的情况来测试我们的有序符号表 API。一些简单的例子包括有序的键列、逆序的键列、所有键全部相同或者只含有两种不同的值。
- 3.1.33 自组织查找。编写一段程序调用自组织查找的实现（请见练习 3.1.22），用 `put()` 构造一个大小为  $N$  的符号表，然后根据预先定义好的概率分布进行  $10N$  次命中查找。对于  $N=10^3$ 、 $10^4$ 、 $10^5$  和  $10^6$ ，用这段程序比较你在练习 3.1.22 中的实现和 `BinarySearchST` 的运行时间，在预定义的概率分布中查找命中第  $i$  小的键的概率为  $1/2^i$ 。
- 3.1.34 Zipf 法则。用命中第  $i$  小的键的概率为  $1/(iH_N)$  的分布重新完成上一道练习，其中  $H_N$  为调和级数（请见表 1.4.6）。这种分布被称为 Zipf 法则。比较前移编码和上一道练习中的在特定分布下的最优安排，该安排将所有键按升序排列（即按照它们的期望频率的降序排列）。
- 3.1.35 性能验证 I。用各种不同的  $N$  运行双倍测试，取《双城记》的前  $N$  个单词，验证 `FrequencyCounter` 在使用 `SequentialSearchST` 时所需的运行时间是  $N$  的平方级别的猜想。
- 3.1.36 性能验证 II。解释 `FrequencyCounter` 在使用 `BinarySearchST` 时比使用 `SequentialSearchST` 时的性能提高程度好于预期的原因。
- 3.1.37 `put/get` 的比例。当 `FrequencyCounter` 使用 `BinarySearchST` 在 100 万个长度为  $M$  位的随机整数中统计每个值的出现频率时，根据经验判断 `BinarySearchST` 中 `put()` 操作和 `get()` 操作的耗时比，其中  $M=10$ 、 $20$  和  $30$ 。再统计 `tale.txt` 并评估耗时比，并比较两次的结果。
- 3.1.38 均摊成本图。修改 `FrequencyCounter`、`SequentialSearchST` 和 `BinarySearchST`，统计计算中每次 `put()` 操作的成本并生成类似本节所示的图。

393

- 3.1.39 实际耗时。修改 `FrequencyCounter`, 用 `Stopwatch` 和 `StdDraw` 绘图, 其中  $x$  轴为 `get()` 和 `put()` 的调用次数之和,  $y$  轴为总运行时间, 每次调用时就根据已运行时间画一个点。分别用 `SequentialSearchST` 和 `BinarySearchST` 处理《双城记》并讨论运行的结果。注意: 曲线上突然的跳跃可能是缓存导致的, 这已经超出了这个问题的讨论范围。
- 3.1.40 二分查找的临界点。找出使用二分查找比顺序查找要快 10 000 倍和 1000 倍的  $N$  值。分析并预测  $N$  的大小并通过实验验证它。
- 3.1.41 插值查找的临界点。找出使用插值查找比二分查找要快 1 倍、2 倍和 10 倍的  $N$  值, 其中假设所有键为随机的 32 位整数 (请见练习 3.1.24)。分析并预测  $N$  的大小并通过实验验证它。

394



## 第4章 图

在许多计算机应用中，由相连的结点所表示的模型起到了关键的作用。这些结点之间的连接很自然地会让人们产生一连串的疑问：沿着这些连接能否从一个结点到达另一个结点？有多少个结点和指定的结点相连？两个结点之间最短的连接是哪一条？

要描述这些问题，我们要使用一种抽象的数学对象，叫做图。本章中，我们会详细研究图的基本性质，为学习各种算法并回答这种类型的疑问作好准备。这些算法是解决许多重要的实际问题的基础，没有优秀的算法，这些问题的解决无法想象。

图论作为数学领域中的一个重要分支已经有数百年的历史了。人们发现了图的许多重要而实用的性质，发明了许多重要的算法，其中许多困难问题的研究仍然十分活跃。本章中，我们会介绍一系列基础的图算法，它们在各种应用中都十分重要。

和我们已经研究过的许多其他问题域一样，关于图的算法研究相对来说才开始不久。尽管有些基础的算法在几个世纪前就已发现了，但大多数有趣的结论都是近几十年才被发现。得益于我们已经学习过的那些算法，即使是由最简单的图论算法得到的程序也是很有用的，而那些我们将要学习的复杂算法则都是已知的最优美和最有意思的算法的一部分。

为了展示图论应用的广泛领域，在探索这片富饶之地之前，我们先来看以下几个示例。

514  
515

**地图。**正在计划旅行的人也许想知道“从普罗维登斯到普林斯顿的最短路线”。对最短路径上经历过交通堵塞的旅行者可能会问：“从普罗维登斯到普林斯顿的哪条路线最快？”要回答这些问题，我们都要处理有关结点（十字路口）之间多条连接（公路）的信息。

**网页信息。**当我们在浏览网页时，页面上都会包含其他网页的引用（链接）。通过单击链接，我们可以从一个页面跳到另一个页面。整个互联网就是一张图，结点是网页，连接就是超链接。图算法是帮助我们在网络上定位信息的搜索引擎的关键组件。

**电路。**在一块电路板上，晶体管、电阻、电容等各种元件是精密连接在一起的。我们使用计算机来控制制造电路板的机器并检查电路板的功能是否正常。我们既要检查短路这类简单问题，也要检查这幅电路图中的导线在蚀刻到芯片上时是否会出现交叉等复杂问题。第一类问题的答案仅取决于连接（导线）的属性，而第二个问题则会涉及导线、各种元件以及芯片的物理特性等详细信息。

**任务调度。**商品的生产过程包含了许多工序以及一些限制条件，这些条件会决定某些任务的先后次序。如何安排才能在满足限制条件的情况下用最少的时间完成这些生产工序呢？

**商业交易。**零售商和金融机构都会跟踪市场中的买卖信息。在这种情形下，一条连接可以表示现金和商品在买方和卖方之间的转移。在此情况下，理解图的连接结构原理可能有助于增强人们对市场的理解。

**配对。**学生可以申请加入各种机构，例如社交俱乐部、大学或是医学院等。这里结点就对应学生和机构，而连接则对应递交的申请。我们希望找到申请者与他们感兴趣的空位之间配对的方法。

**516** 计算机网络。计算机网络是由能够发送、转发和接收各种消息的站点互相连接组成的。我们感兴趣的是这种互联结构的性质，因为我们希望网络中的线路和交换设备能够高效率地处理网络流量。

软件。编译器会使用图来表示大型软件系统中各个模块之间的关系。图中的结点即构成整个系统的各种类和模块，连接则为类的方法之间的可能调用关系（静态分析），或是系统运行时的实际调用关系（动态分析）。我们需要分析这幅图来决定如何以最优的方式为程序分配资源。

社交网络。当你在使用社交网站时，会和你的朋友之间建立起明确的关系。这里，结点对应人而连接则联系着你和你的朋友或是关注者。分析这些社交网络的性质是当前图算法的一个重要应用。对它感兴趣的不止是社交网络的公司，还包括政治、外交、娱乐、教育、市场等许多其他机构（参见表 4.0.1）。

表 4.0.1 图的典型应用

应    用	结    点	连    接
地图	十字路口	公路
网络内容	网页	超链接
电路	元器件	导线
任务调度	任务	限制条件
商业交易	客户	交易
配对	学生	申请
计算机网络	网站	物理连接
软件	方法	调用关系
社交网络	人	友谊关系

这些示例展示了图作为一种抽象模型的应用范围以及我们在处理图时可能会遇到的各种计算问题。人们研究过的关于图的问题数以千计，但它们大多数都能用一些简单的图模型解决——本章我们将会学习几个最重要的模型。在实际应用中，处理庞大的数据是很常见的，因此解决方法是否可行完全取决于算法的效率。

**517** 在本章中，我们会依次学习 4 种最重要的图模型：无向图（简单连接）、有向图（连接有方向性）、加权图（连接带有权值）和加权有向图（连接既有方向性又带有权值）。

## 4.1 无向图

在我们首先要学习的这种图模型中，边（edge）仅仅是两个顶点（vertex）之间的连接。为了和其他图模型相区别，我们将它称为无向图。这是一种最简单的图模型，我们先来看一下它的定义。

**定义。**图是由一组顶点和一组能够将两个顶点相连的边组成的。

就定义而言，顶点叫什么名字并不重要，但我们需要一个方法来指代这些顶点。一般使用 $0$ 至 $V-1$ 来表示一张含有 $V$ 个顶点的图中的各个顶点。这样约定是为了方便使用数组的索引来编写能够高效访问各个顶点中信息的代码。用一张符号表来为顶点的名字和 $0$ 到 $V-1$ 的整数值建立一一对应的关系并不困难（请见4.1.7节），因此直接使用数组索引作为结点的名称更方便且不失一般性（也不会损失什么效率）。我们用 $v-w$ 的记法来表示连接 $v$ 和 $w$ 的边， $w-v$ 是这条边的另一种表示方法。

在绘制一幅图时，用圆圈表示顶点，用连接两个顶点的线段表示边，这样就能直观地看出图的结构。但这种直觉有时也可能误导我们，因为图的定义和绘出的图像是无关的。例如，图4.1.1中的两组图表示的是同一幅图，因为图的构成只有（无序的）顶点和边（顶点对）。

**特殊的图。**我们的定义允许出现两种简单而特殊的情况，参见图4.1.2：

- 自环，即一条连接一个顶点和其自身的边；
- 连接同一对顶点的两条边称为平行边。

数学家常常将含有平行边的图称为多重图，而将没有平行边或自环的图称为简单图。一般来说，实现允许出现自环和平行边（因为它们会在实际应用中出现），但我们不会将它们作为示例。因此，我们用两个顶点就可以指代一条边了。

### 4.1.1 术语表

和图有关的术语非常多，其中大多数定义都很简单，我们在这里集中介绍。

当两个顶点通过一条边相连时，我们称这两个顶点是相邻的，并称该连接依附于这两个顶点。某个顶点的度数即为依附于它的边的总数。子图是由一幅图的所有边的一个子集（以及它们所依附的所有顶点）组成的图。许多计算问题都需要识别各种类型的子图，特别是由能够顺序连接一系列顶点的边所组成的子图。

**定义。**在图中，路径是由边顺序连接的一系列顶点。简单路径是一条没有重复顶点的路径。环是一条至少含有一条边且起点和终点相同的路径。简单环是一条（除了起点和终点必须相同之外）不含有重复顶点和边的环。路径或者环的长度为其中所包含的边数。

大多数情况下，我们研究的都是简单环和简单路径并会省略掉简单二字。当允许重复的顶点时，我们指的都是一般的路径和环。当两个顶点之间存在一条连接双方的路径时，我们称一个顶点和另

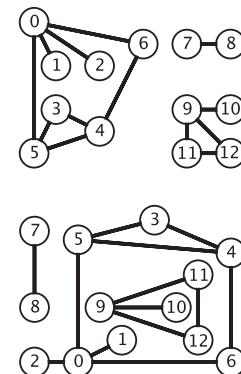


图 4.1.1 同一幅图的两种表示

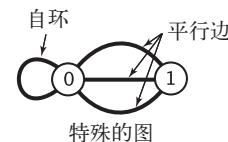


图 4.1.2 特殊的图

一个顶点是连通的。我们用类似  $u-v-w-x$  的记法来表示  $u$  到  $x$  的一条路径，用  $u-v-w-x-u$  表示从  $u$  到  $v$  到  $w$  到  $x$  再回到  $u$  的一条环。我们会学习几种查找路径和环的算法。另外，路径和环也会帮我们从整体上考虑一幅图的性质，参见图 4.1.3。

**定义。**如果从任意一个顶点都存在一条路径到达另一个任意顶点，我们称这幅图是连通图。一幅非连通的图由若干连通的部分组成，它们都是其极大连通子图。

直观上来说，如果顶点是物理存在的对象，例如绳节或是念珠，而边也是物理存在的对象，例如绳子或是电线，那么将任意顶点提起，连通图都将是一个整体，而非连通图则会变成两个或多个部分。一般来说，要处理一张图就需要一个个地处理它的连通分量（子图）。

无环图是一种不包含环的图。我们将要学习的几个算法就是要找出一幅图中满足一定条件的无环子图。我们还需要一些术语来表示这些结构。

**定义。**树是一幅无环连通图。互不相连的树组成的集合称为森林。连通图的生成树是它的一幅子图，它含有图中的所有顶点且是一棵树。图的生成树森林是它的所有连通子图的生成树的集合，参见图 4.1.4 和图 4.1.5。

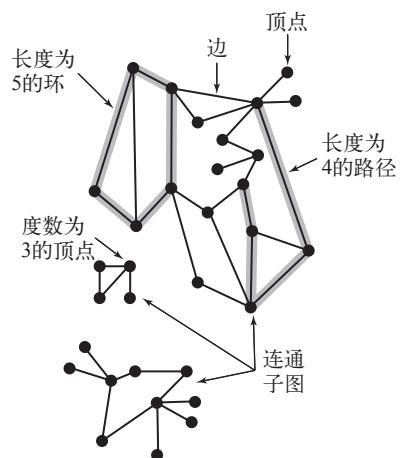


图 4.1.3 图的详解

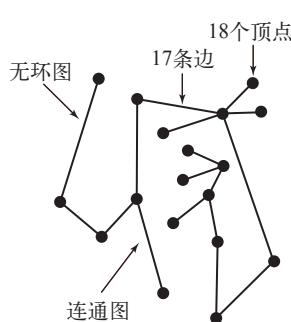


图 4.1.4 一棵树

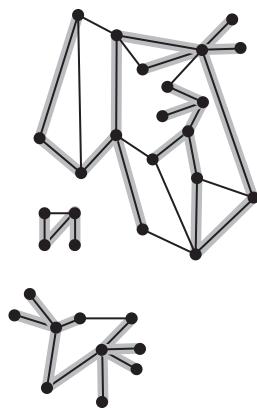


图 4.1.5 生成树森林

树的定义非常通用，稍做改动就可以变成用来描述程序行为的（函数调用层次）模型和数据结构（二叉查找树、2-3 树等）。树的数学性质很直观并且已被系统地研究过，因此我们就不给出它们的证明了。例如，当且仅当一幅含有  $V$  个结点的图  $G$  满足下列 5 个条件之一时，它就是一棵树：

- $G$  有  $V-1$  条边且不含有环；
- $G$  有  $V-1$  条边且是连通的；
- $G$  是连通的，但删除任意一条边都会使它不再连通；
- $G$  是无环图，但添加任意一条边都会产生一条环；
- $G$  中的任意一对顶点之间仅存在一条简单路径。

我们会学习几种寻找生成树和森林的算法，以上这些性质在分析和实现这些算法的过程中扮演着重要的角色。

图的密度是指已经连接的顶点对占所有可能被连接的顶点对的比例。在稀疏图中，被连接的顶点对很少；而在稠密图中，只有少部分顶点对之间没有边连接。一般来说，如果一幅图中不同的边的数量只占顶点总数  $V$  的一小部分，那么我们就认为这幅图是稀疏的，否则则是稠密的，参见图 4.1.6。这条经验规律虽然会留下一片灰色地带（比如当边的数量为  $\sim cV^{3/2}$  时），但实际应用中稀疏图和稠密图之间的区别是十分明显的。我们将会遇到的应用使用的几乎都是稀疏图。

二分图是一种能够将所有结点分为两部分的图，其中图的每条边所连接的两个顶点都分别属于不同的部分。图 4.1.7 即为一幅二分图的示例，其中红色的结点是一个集合，黑色的结点是另一个集合。二分图会出现在许多场景中，我们会在本节的最后详细研究其中的一个场景。

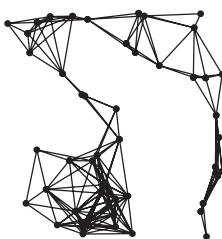
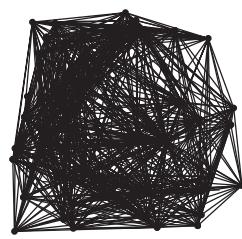
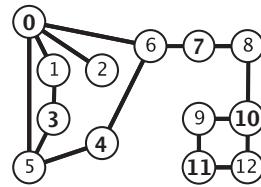
稀疏图 ( $E=200$ )稠密图 ( $E=1000$ )图 4.1.6 两幅图 ( $V=50$ )

图 4.1.7 二分图 (另见彩插)

现在，我们已经做好了学习图处理算法的准备。我们首先会研究一种表示图的数据类型的 API 及其实现，然后会学习一些查找图和鉴别连通分量的经典算法。最后，我们会考虑真实世界中的一些图的应用，它们的顶点的名字可能不是整数并且会含有数目庞大的顶点和边。

519  
521

## 4.1.2 表示无向图的数据类型

要开发处理图问题的各种算法，我们首先来看一份定义了图的基本操作的 API，参见表 4.1.1。有了它我们才能完成从简单的基本操作到解决复杂问题的各种任务。

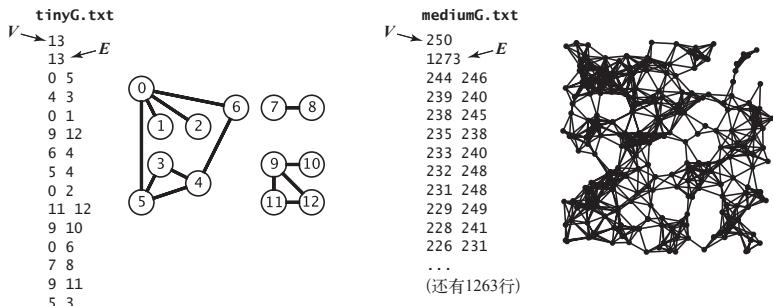
表 4.1.1 无向图的 API

public class	Graph	
	Graph(int V)	创建一个含有 $V$ 个顶点但不含有边的图
	Graph(Iterable<Integer> in)	从标准输入流 $in$ 读入一幅图
int	V()	顶点数
int	E()	边数
void	addEdge(int v, int w)	向图中添加一条边 $v-w$
Iterable<Integer>	adj(int v)	和 $v$ 相邻的所有顶点
String	toString()	对象的字符串表示

这份 API 含有两个构造函数，有两个方法用来分别返回图中的顶点数和边数，有一个方法用来添加一条边，`toString()` 方法和 `adj()` 方法用来允许用例遍历给定顶点的所有相邻顶点（遍历顺序不确定）。值得注意的是，本节将学习的所有算法都基于 `adj()` 方法所抽象的基本操作。

第二个构造函数接受的输入由  $2E+2$  个整数组成：首先是  $V$ ，然后是  $E$ ，再然后是  $E$  对 0 到  $V-1$  之间的整数，每个整数对都表示一条边。例如，我们使用了由图 4.1.8 中的 `tinyG.txt` 和 `mediumG.txt` 所描述的两个示例。

调用 Graph 的几段用例代码请见表 4.1.2。



[522]

图 4.1.8 Graph 的构造函数的输入格式（两个示例）

表 4.1.2 最常用的图处理代码

任 务	实 现
计算 v 的度数	<pre>public static int degree(Graph G, int v) {     int degree = 0;     for (int w : G.adj(v)) degree++;     return degree; }</pre>
计算所有顶点的最大度数	<pre>public static int maxDegree(Graph G) {     int max = 0;     for (int v = 0; v &lt; G.V(); v++)         if (degree(G, v) &gt; max)             max = degree(G, v);     return max; }</pre>
计算所有顶点的平均度数	<pre>public static double avgDegree(Graph G) { return 2 * G.E() / G.V(); }</pre>
计算自环的个数	<pre>public static int numberofSelfLoops(Graph G) {     int count = 0;     for (int v = 0; v &lt; G.V(); v++)         for (int w : G.adj(v))             if (v == w) count++;     return count/2; // 每条边都被记过两次 }</pre>
图的邻接表的字符串表示 (Graph 的实例方法)	<pre>public String toString() {     String s = V + " vertices, " + E + " edges\n";     for (int v = 0; v &lt; V; v++)     {         s += v + ": ";         for (int w : this.adj(v))             s += w + " ";         s += "\n";     }     return s; }</pre>

[523]

#### 4.1.2.1 图的几种表示方法

我们要面对的下一个图处理问题就是用哪种方式（数据结构）来表示图并实现这份 API，这包含以下两个要求：

- 它必须为可能在应用中碰到的各种类型的图预留出足够的空间；
  - **Graph** 的实例方法的实现一定要快——它们是开发处理图的各种用例的基础。
- 这些要求比较模糊，但它们仍然能够帮助我们在三种图的表示方法中进行选择。

- 邻接矩阵。我们可以使用一个  $V$  乘  $V$  的布尔矩阵。当顶点  $v$  和顶点  $w$  之间有相连接的边时，定义  $v$  行  $w$  列的元素值为 **true**，否则为 **false**。这种表示方法不符合第一个条件——含有上百万个顶点的图是很常见的， $V^2$  个布尔值所需的空间是不能满足的。
- 边的数组。我们可以使用一个 **Edge** 类，它含有两个 **int** 实例变量。这种表示方法很简洁但不满足第二个条件——要实现 **adj()** 需要检查图中的所有边。
- 邻接表数组。我们可以使用一个以顶点为索引的列表数组，其中的每个元素都是和该顶点相邻的顶点列表，参见图 4.1.9。这种数据结构能够同时满足典型应用所需的以上两个条件，我们会在本章中一直使用它。

除了这些性能目标之外，经过缜密的检查，我们还发现了另一些在某些应用中可能会很重要的东西。例如，允许存在平行边相当于排除了邻接矩阵，因为邻接矩阵无法表示它们。

#### 4.1.2.2 邻接表的数据结构

非稠密图的标准表示称为邻接表的数据结构，它将每个顶点的所有相邻顶点都保存在该顶点对应的元素所指向的一张链表中。我们使用这个数组就是为了快速访问给定顶点的邻接顶点列表。这里使用 1.3 节中的 **Bag** 抽象数据类型来实现这个链表，这样我们就可以在常数时间内添加新的边或遍历任意顶点的所有相邻顶点。后面框注“**Graph** 数据类型”中的 **Graph** 类的实现就是基于这种方法，而图 4.1.9 中所示的正是用这种方法处理 **tinyG.txt** 所得到的数据结构。要添加一条连接  $v$  与  $w$  的边，我们将  $w$  添加到  $v$  的邻接表中并把  $v$  添加到  $w$  的邻接表中。因此，在这个数据结构中每条边都会出现两次。这种 **Graph** 的实现的性能有如下特点：

- 使用的空间和  $V+E$  成正比；

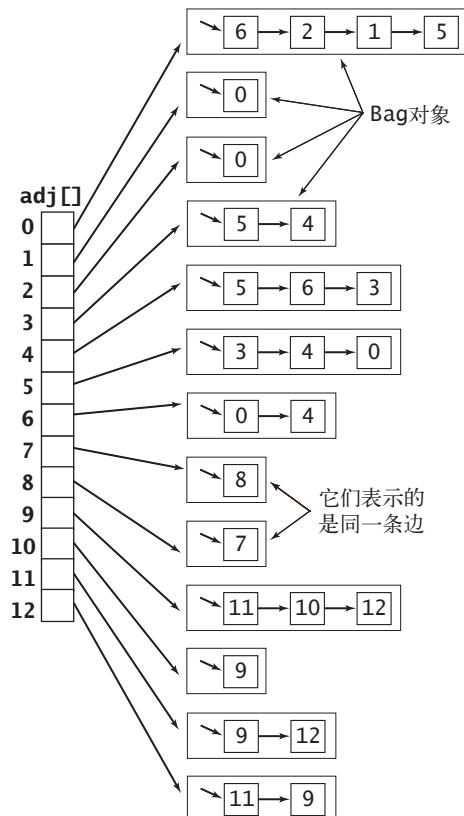


图 4.1.9 邻接表数组示意（无向图）

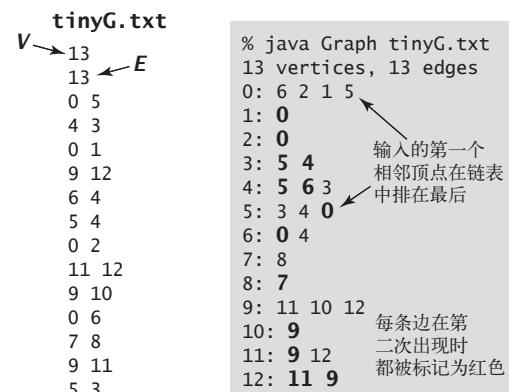
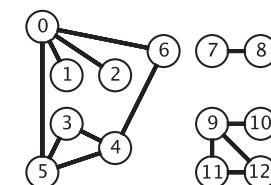


图 4.1.10 由边得到的邻接表（另见彩插）

- 添加一条边所需的时间为常数；
- 遍历顶点  $v$  的所有相邻顶点所需的时间和  $v$  的度数成正比（处理每个相邻顶点所需的时间为常数）。

对于这些操作，这样的特性已经是最优的了，这已经可以满足图处理应用的需要，而且支持平行边和自环（我们不会检测它们）。注意，边的插入顺序决定了 `Graph` 的邻接表中顶点的出现顺序，参见图 4.1.10。多个不同的邻接表可能表示着同一幅图。当使用构造函数从标准输入中读入一幅图时，这就意味着输入的格式和边的顺序决定了 `Graph` 的邻接表数组中顶点的出现顺序。因为算法在使用 `adj()` 来处理所有相邻的顶点时不会考虑它们在邻接表中的出现顺序，这种差异不会影响算法的正确性，但在调试或是跟踪邻接表的轨迹时我们还是需要注意这一点。为了简化操作，假设 `Graph` 有一个测试用例来从命令行参数指定的文件中读取一幅图并将它打印出来（参见表 4.1.2 中的 `toString()` 方法的实现），以显示邻接表中的各个顶点的出现顺序，这也是算法处理它们的顺序（请见练习 4.1.7）。

525

### Graph 数据类型

---

```

public class Graph
{
    private final int V;           // 顶点数目
    private int E;                 // 边的数目
    private Bag<Integer>[] adj;   // 邻接表
    public Graph(int V)
    {
        this.V = V; this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V];      // 创建邻接表
        for (int v = 0; v < V; v++)               // 将所有链表初始化为空
            adj[v] = new Bag<Integer>();
    }
    public Graph(In in)
    {
        this(in.readInt());          // 读取V并将图初始化
        int E = in.readInt();         // 读取E
        for (int i = 0; i < E; i++)
        { // 添加一条边
            int v = in.readInt();     // 读取一个顶点
            int w = in.readInt();     // 读取另一个顶点
            addEdge(v, w);           // 添加一条连接它们的边
        }
    }
    public int V() { return V; }
    public int E() { return E; }
    public void addEdge(int v, int w)
    {
        adj[v].add(w);             // 将w添加到v的链表中
        adj[w].add(v);             // 将v添加到w的链表中
        E++;
    }
    public Iterable<Integer> adj(int v)
    { return adj[v]; }
}

```

这份 `Graph` 的实现使用了一个由顶点索引的整型链表数组。每条边都会出现两次，即当存在一条连接  $v$  与  $w$  的边时， $w$  会出现在  $v$  的链表中， $v$  也会出现在  $w$  的链表中。第二个构造函数从输入流中读取一幅图，开头是  $V$ ，然后是  $E$ ，再然后是一列整数对，大小在 0 到  $V-1$  之间。`toString()` 方法请见表 4.1.2。

526

在实际应用中还有一些操作可能是很有用的，例如：

- 添加一个顶点；
- 删除一个顶点。

实现这些操作的一种方法是扩展之前的 API，使用符号表( ST )来代替由顶点索引构成的数组( 这样修改之后就不需要约定顶点名必须是整数了 )。我们可能还需要：

- 删除一条边；
- 检查图是否含有边  $v-w$ 。

要实现这些方法(不允许存在平行边)，我们可能需要使用 SET 代替 Bag 来实现邻接表。我们称这种方法为邻接集。本书中不会使用这些数据结构，因为：

- 用例代码不需要添加顶点、删除顶点和边或是检查一条边是否存在；
- 当用例代码需要进行上述操作时，由于频率很低或者相关的邻接链表很短，因此可以直接使用穷举法遍历链表来实现；
- 使用 SET 和 ST 会令算法的实现变得更加复杂，分散了读者对算法本身的注意力；
- 在某些情况下，它们会使性能损失  $\log V$ 。

使我们的算法适应其他设计(例如，不允许出现平行边或是自环)并避免不必要的性能损失并不困难。表 4.1.3 总结了之前提到过的所有其他实现方法的性能特点。常见的应用场景都需要处理庞大的稀疏图，因此我们会一直使用邻接表。

表 4.1.3 典型 Graph 实现的性能复杂度

数据结构	所需空间	添加一条边 $v-w$	检查 $w$ 和 $v$ 是否相邻	遍历 $v$ 的所有相邻顶点
边的列表	$E$	1	$E$	$E$
邻接矩阵	$V^2$	1	1	$V$
邻接表	$E+V$	1	$degree(v)$	$degree(v)$
邻接集	$E+V$	$\log V$	$\log V$	$\log V + degree(v)$

527

#### 4.1.2.3 图的处理算法的设计模式

因为我们会讨论大量关于图处理的算法，所以设计的首要目标是将图的表示和实现分离开来。为此，我们会为每个任务创建一个相应的类，用例可以创建相应的对象来完成任务。类的构造函数一般会在预处理中构造各种数据结构，以有效地响应用例的请求。典型的用例程序会构造一幅图，将图传递给实现了某个算法的类(作为构造函数的参数)，然后调用用例的方法来获取图的各种性质。作为热身，我们先来看看这份 API，参见表 4.1.4。

表 4.1.4 图处理算法的 API (热身)

public class Search		
	Search(Graph G, int s)	找到和起点 $s$ 连通的所有顶点
boolean	marked(int v)	$v$ 和 $s$ 是连通的吗
int	count()	与 $s$ 连通的顶点总数

我们用起点( source )区分作为参数传递给构造函数的顶点与图中的其他顶点。在这份 API 中，构造函数的任务是找到图中与起点连通的其他顶点。用例可以调用 marked() 方法和 count() 方

528

法来了解图的性质。方法名 `marked()` 指的是这种基本算法使用的一种实现方式，本章中会一直使用到这种算法：在图中从起点开始沿着路径到达其他顶点并标记每个路过的顶点。后面框注中的图处理用例 `TestSearch` 接受由命令行得到的一个输入流的名称和起始结点的编号，从输入流中读取一幅图（使用 `Graph` 的第二个构造函数），用这幅图和给定的起始结点创建一个 `Search` 对象，然后用 `marked()` 打印出图中和起点连通的所有顶点。它也调用了 `count()` 并打印了图是否是连通的（当且仅当搜索能够标记图中的所有顶点时图才是连通的）。

```
public class TestSearch
{
    public static void main(String[] args)
    {
        Graph G = new Graph(new In(args[0]));
        int s = Integer.parseInt(args[1]);
        Search search = new Search(G, s);

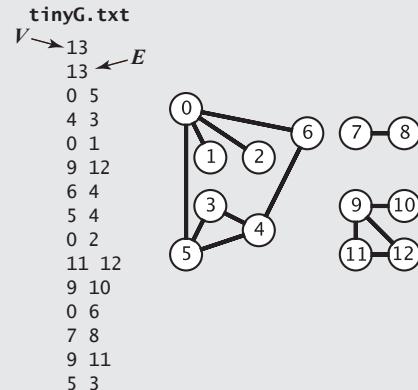
        for (int v = 0; v < G.V(); v++)
            if (search.marked(v))
                StdOut.print(v + " ");
        StdOut.println();

        if (search.count() != G.V())
            StdOut.print("NOT ");
        StdOut.println("connected");
    }
}
```

图处理的用例（热身）

```
% java TestSearch tinyG.txt 0
0 1 2 3 4 5 6
NOT connected

% java TestSearch tinyG.txt 9
9 10 11 12
NOT connected
```

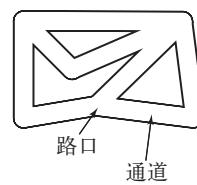


我们已经见过 `Search API` 的一种实现：第 1 章中的 union-find 算法。它的构造函数会创建一个 `UF` 对象，对图中的每一条边进行一次 `union()` 操作并调用 `connected(s, v)` 来实现 `marked(v)` 方法。实现 `count()` 方法需要一个加权的 `UF` 实现并扩展它的 API，以便使用 `count()` 方法返回 `wt[find(v)]`（请见练习 4.1.8）。这种实现简单而高效，但下面我们要学习的实现还可以更进一步。它基于的是深度优先搜索（DFS）的。这是一种重要的递归方法，它会沿着图的边寻找和起点连通的所有顶点。深度优先搜索是本章中将学习的好几种关于图的算法的基础。

### 4.1.3 深度优先搜索

我们常常通过系统地检查每一个顶点和每一条边来获取图的各种性质。要得到图的一些简单性质（比如，计算所有顶点的度数）很容易，只要检查每一条边即可（任意顺序）。但图的许多其他性质和路径有关，因此一种很自然的想法是沿着图的边从一个顶点移动到另一个顶点。尽管存在各种各样的处理策略，但后面将要学习的几乎所有与图有关的算法都使用了这个简单的抽象模型，其中最

迷宫



图

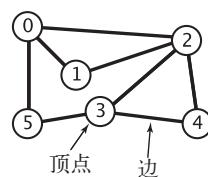


图 4.1.11 等价的迷宫模型

简单的就是下面介绍的这种经典的方法。

#### 4.1.3.1 走迷宫

思考图的搜索过程的一种有益的方法是，考虑另一个和它等价但历史悠久而又特别的问题——在一个由各种通道和路口组成的迷宫中找到出路。有些迷宫的规则很简单，但大多数迷宫则需要很复杂的策略才行。用迷宫代替图、通道代替边、路口代替顶点仅仅只是一些文字游戏，但就目前来说，这么做可以帮助我们直观地认识问题，参见图 4.1.11。探索迷宫而不迷路的一种古老办法(至少可以追溯到忒修斯和米诺陶的传说)叫做 Tremaux 搜索，参见图 4.1.12。要探索迷宫中的所有通道，我们需要：

- 选择一条没有标记过的通道，在你走过的路上铺一条绳子；
- 标记所有你第一次路过的路口和通道；
- 当来到一个标记过的路口时(用绳子)回退到上个路口；
- 当回退到的路口已没有可走的通道时继续回退。

绳子可以保证你总能找到一条出路，标记则能保证你不会两次经过同一条通道或者同一个路口。要知道是否完全探索了整个迷宫需要的证明更复杂，只有用图搜索才能够更好地处理问题。Tremaux 搜索很直接，但它与完全搜索一张图仍然稍有不同，因此我们接下来看看图的搜索方法。

```
public class DepthFirstSearch
{
    private boolean[] marked;
    private int count;

    public DepthFirstSearch(Graph G, int s)
    {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        count++;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

    public boolean marked(int w)
    {
        return marked[w];
    }

    public int count()
    {
        return count;
    }
}
```

深度优先搜索

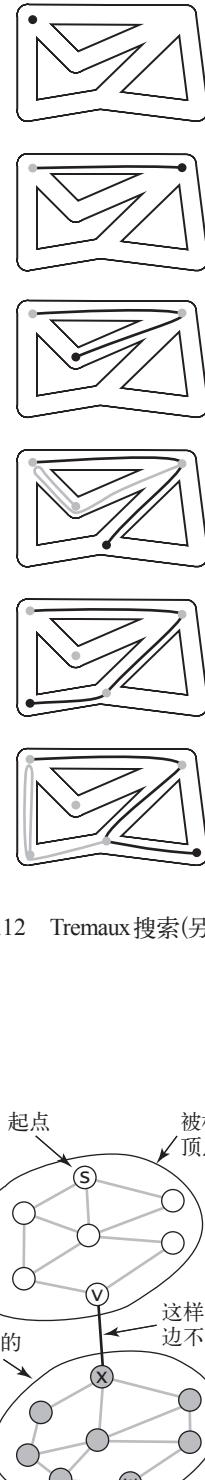
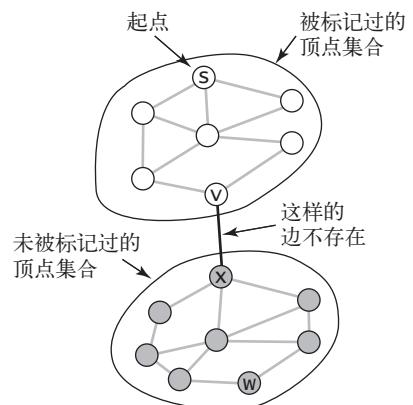


图 4.1.12 Tremaux 搜索(另见彩插)



### 4.1.3.2 热身

搜索连通图的经典递归算法（遍历所有的顶点和边）和 Tremaux 搜索类似，但描述起来更简单。要搜索一幅图，只需用一个递归方法来遍历所有顶点。在访问其中一个顶点时：

- 将它标记为已访问；
- 递归地访问它的所有没有被标记过的邻居顶点。

这种方法称为深度优先搜索（DFS）。Search API 的一种实现使用了这种方法，如深度优先搜索框注所示。它使用一个 `boolean` 数组来记录和起点连通的所有顶点。递归方法会标记给定的顶点并调用自己来访问该顶点的相邻顶点列表中所有没有被标记过的顶点。如果图是连通的，每个邻接链表中的元素都会被检查到。

**命题 A。** 深度优先搜索标记与起点连通的所有顶点所需的时间和顶点的度数之和成正比。

**证明。** 首先，我们要证明这个算法能够标记与起点  $s$  连通的所有顶点（且不会标记其他顶点）。因为算法仅通过边来寻找顶点，所以每个被标记过的顶点都与  $s$  连通。现在，假设某个没有被标记过的顶点  $w$  与  $s$  连通。因为  $s$  本身是被标记过的，由  $s$  到  $w$  的任意一条路径中至少有一条边连接的两个顶点分别是被标记过的和没有被标记过的，例如  $v-x$ 。根据算法，在标记了  $v$  之后必然会出现  $x$ ，因此这样的边是不存在的。前后矛盾。每个顶点都只会被访问一次保证了时间上限（检查标记的耗时和度数成正比）。

529  
531

### 4.1.3.3 单向通道

代码中方法的调用和返回机制对应迷宫中绳子的作用：当已经处理过依附于一个顶点的所有边时（搜索了路口连接的所有通道），我们就只能“返回”（`return`，两者的意义相同）。为了更好地与迷宫的 Tremaux 搜索对应起来，我们可以想象一座完全由单向通道构造的迷宫（每个方向都有一个通道）。和在迷宫中会经过一条通道两次（方向不同）一样，在图中我们也会路过每条边两次（在它的两个端点各一次）。在 Tremaux 搜索中，要么是第一次访问一条边，要么是沿着它从一个被标记过的顶点退回。在无向图的深度优先搜索中，在碰到边  $v-w$  时，要么进行递归调用（ $w$  没有被标记过），要么跳过这条边（ $w$  已经被标记过）。第二次从另一个方向  $w-v$  遇到这条边时，总是会忽略它，因为它的另一端  $v$  肯定已经被访问过了（在第一次遇到这条边的时候）。

### 4.1.3.4 跟踪深度优先搜索

通常，理解算法的最好方法是在一个简单的例子中跟踪它的行为。深度优先算法尤其是这样。在跟踪它的轨迹时，首先要注意的是，算法遍历边和访问顶点的顺序与图的表示是有关的，而不只是与

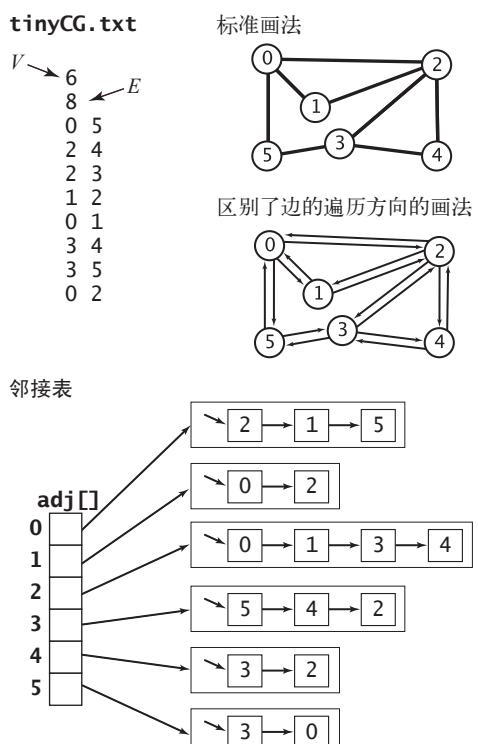


图 4.1.13 一幅连通的无向图

图的结构或是算法有关。因为深度优先搜索只会访问和起点连通的顶点，所以使用图 4.1.13 所示的一幅小型连通图为例。在示例中，顶点 2 是顶点 0 之后第一个被访问的顶点，因为它正好是 0 的邻接表的第一个元素。要注意的第二点是，如前文所述，深度优先搜索中每条边都会被访问两次，且在第二次时总会发现这个顶点已经被标记过。这意味着深度优先搜索的轨迹可能会比你想象的长一倍！示例图仅含有 8 条边，但需要追踪算法在邻接表的 16 个元素上的操作。

#### 4.1.3.5 深度优先搜索的详细轨迹

图 4.1.14 显示的是示例中每个顶点被标记后算法使用的数据结构，起点为顶点 0。查找开始于构造函数调用递归的 `dfs()` 来标记和访问顶点 0，后续处理如下所述。

- 因为顶点 2 是 0 的邻接表的第一个元素且没有被标记过，`dfs()` 递归调用自己来标记并访问顶点 2（效果是系统会将顶点 0 和 0 的邻接表的当前位置压入栈中）。
- 现在，顶点 0 是 2 的邻接表的第一个元素且已经被标记过了，因此 `dfs()` 跳过了它。接下来，顶点 1 是 2 的邻接表的第二个元素且没有被标记，`dfs()` 递归调用自己来标记并访问顶点 1。
- 对顶点 1 的访问和前面有所不同：因为它的邻接表中的所有顶点（0 和 2）都已经被标记过了，因此不需要再进行递归，方法从 `dfs(1)` 中返回。下一条被检查的边是 2-3（在 2 的邻接表中顶点 1 之后的顶点是 3），因此 `dfs()` 递归调用自己来标记并访问顶点 3。
- 顶点 5 是 3 的邻接表的第一个元素且没有被标记，因此 `dfs()` 递归调用自己来标记并访问顶点 5。
- 顶点 5 的邻接表中的所有顶点（3 和 0）都已经被标记过了，因此不需要再进行递归。
- 顶点 4 是 3 的邻接表的下一个元素且没有被标记过，因此 `dfs()` 递归调用自己来标记并访问顶点 4。这是最后一个需要被标记的顶点。

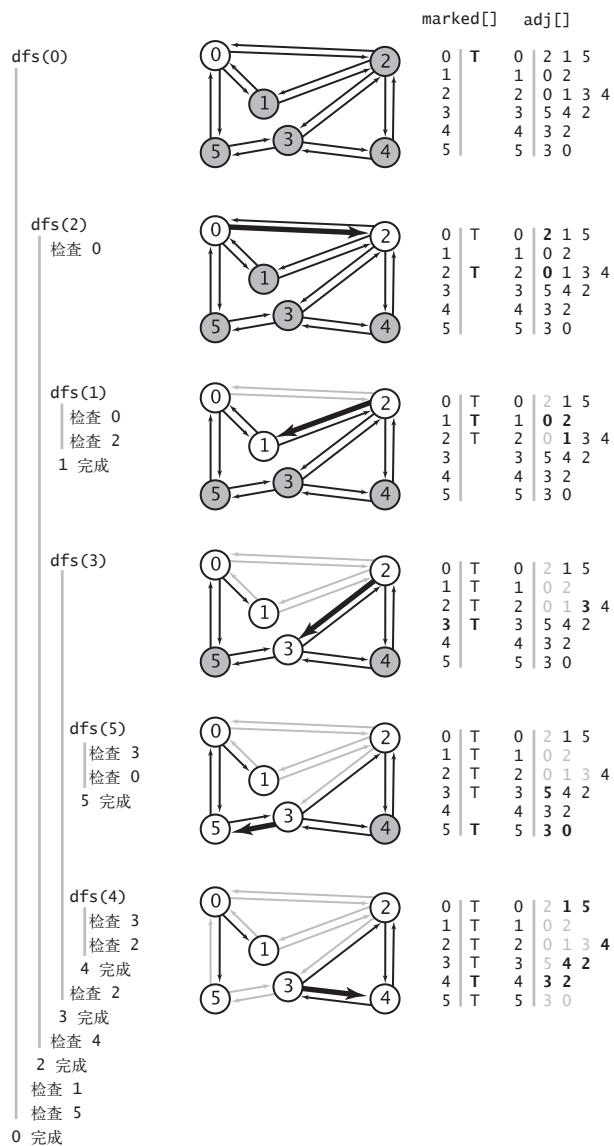


图 4.1.14 使用深度优先搜索的轨迹，寻找所有和顶点 0 连通的顶点（另见彩插）

532  
533

- 在顶点 4 被标记了之后，`dfs()` 会检查它的邻接表，然后再检查 3 的邻接表，然后是 2 的邻接表，然后是 0 的，最后发现不需要再进行任何递归调用，因为所有的顶点都已经被标记过了。

这种简单的递归模式只是一个开始——深度优先搜索能够有效处理许多和图有关的任务。例如，本节中，我们已经可以用深度优先搜索来解决在第 1 章首次提到的一个问题。

**连通性。**给定一幅图，回答“两个给定的顶点是否连通？”或者“图中有多少个连通子图？”等类似问题。

我们可以轻易地用处理图问题的标准设计模式给出这些问题的答案，还要将这些解答与在 1.5 节中学习的 union-find 算法进行比较。

问题“两个给定的顶点是否连通？”等价于“两个给定的顶点之间是否存在一条路径？”，也许也可以叫做路径检测问题。但是，在 1.5 节学习的 union-find 算法的数据结构并不能解决找出这样一条路径的问题。深度优先搜索是我们已经学习过的几种方法中第一个能够解决这个问题的算法。它能够解决的另一个问题如下所述。

**单点路径。**给定一幅图和一个起点  $s$ ，回答“从  $s$  到给定目的顶点  $v$  是否存在一条路径？如果有，找出这条路径。”等类似问题。

深度优先搜索算法之所以极为简单，是因为它所基于的概念为人所熟知并且非常容易实现。事实上，它是一个既小巧而又强大的算法，研究人员用它解决了无数困难的问题。上述两个问题只是我们将要研究的许多问题的开始。

534

#### 4.1.4 寻找路径

单点路径问题在图的处理领域中十分重要。根据标准设计模式，我们将使用如下 API（请见表 4.1.5）。

表 4.1.5 路径的 API

public class Paths	
<code>Paths(Graph G, int s)</code>	在 $G$ 中找出所有起点为 $s$ 的路径
<code>boolean hasPathTo(int v)</code>	是否存在从 $s$ 到 $v$ 的路径
<code>Iterable&lt;Integer&gt; pathTo(int v)</code>	$s$ 到 $v$ 的路径，如果不存在则返回 null

构造函数接受一个起点  $s$  作为参数，计算  $s$  到与  $s$  连通的每个顶点之间的路径。在为起点  $s$  创建了 `Paths` 对象后，用例可以调用 `pathTo()` 实例方法来遍历从  $s$  到任意和  $s$  连通的顶点的路径上的所有顶点。现在暂时查找所有路径，以后会实现只查找具有某些属性的路径。

```
% java Paths tinyCG.txt 0
0 to 0: 0
0 to 1: 0-2-1
0 to 2: 0-2
0 to 3: 0-2-3
0 to 4: 0-2-3-4
0 to 5: 0-2-3-5
```

535

```
public static void main(String[] args)
{
    Graph G = new Graph(new In(args[0]));
    int s = Integer.parseInt(args[1]);
    Paths search = new Paths(G, s);
    for (int v = 0; v < G.V(); v++)
    {
        StdOut.print(s + " to " + v + ": ");
        if (search.hasPathTo(v))
            for (int x : search.pathTo(v))
                if (x == s) StdOut.print(x);
                else StdOut.print("-" + x);
            StdOut.println();
    }
}
```

Paths 实现的测试用例

上一页右下角框注中的用例从输入流中读取了一个图并从命令行得到一个起点，然后打印出从起点到与它连通的每个顶点之间的一条路径。

#### 4.1.4.1 实现

算法 4.1 基于深度优先搜索实现了 `Paths`。它扩展了 4.1.3.2 节中的热身代码 `DepthFirstSearch`，添加了一个实例变量 `edgeTo[]` 整型数组来起到 Tremaux 搜索中绳子的作用。这个数组可以找到从每个与 `s` 连通的顶点回到 `s` 的路径。它会记住每个顶点到起点的路径，而不是记录当前顶点到起点的路径。为了做到这一点，在由边 `v-w` 第一次访问任意 `w` 时，将 `edgeTo[w]` 设为 `v` 来记住这条路径。换句话说，`v-w` 是从 `s` 到 `w` 的路径上的最后一条已知的边。这样，搜索的结果是一棵以起点为根结点的树，`edgeTo[]` 是一棵由父链接表示的树。算法 4.1 的代码的右侧是一个小示例。要找出 `s` 到任意顶点 `v` 的路径，算法 4.1 实现的 `pathTo()` 方法用变量 `x` 遍历整棵树，将 `x` 设为 `edgeTo[x]`，就像 1.5 节中的 union-find 算法一样，然后在到达 `s` 之前，将遇到的所有顶点都压入栈中。将这个栈返回为一个 `Iterable` 对象帮助用例遍历 `s` 到 `v` 的路径。

#### 算法 4.1 使用深度优先搜索查找图中的路径

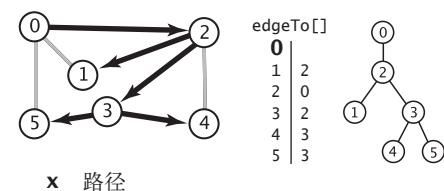
```
public class DepthFirstPaths
{
    private boolean[] marked; // 这个顶点上调用过dfs()了吗?
    private int[] edgeTo; // 从起点到一个顶点的已知路径上的最后一个顶点
    private final int s; // 起点

    public DepthFirstPaths(Graph G, int s)
    {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        this.s = s;
        dfs(G, s);
    }

    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w])
            {
                edgeTo[w] = v;
                dfs(G, w);
            }
    }

    public boolean hasPathTo(int v)
    { return marked[v]; }

    public Iterable<Integer> pathTo(int v)
    {
        if (!hasPathTo(v)) return null;
        Stack<Integer> path = new Stack<Integer>();
        for (int x = v; x != s; x = edgeTo[x])
            path.push(x);
        path.push(s);
        return path;
    }
}
```



x	路径
5	5
3	3 5
2	2 3 5
0	0 2 3 5

pathTo(5) 的计算轨迹

这段 `Graph` 的用例使用了深度优先搜索，以找出图中从给定的起点 `s` 到它连通的所有顶点的路径。

536

来自 `DepthFirstSearch` (4.1.3.2 节) 的代码均为灰色。为了保存到达每个顶点的已知路径，这段代码使用了一个以顶点编号为索引的数组 `edgeTo[]`，`edgeTo[w]=v` 表示  $v-w$  是第一次访问  $w$  时经过的边。`edgeTo[]` 数组是一棵用父链接表示的以  $s$  为根且含有所有与  $s$  连通的顶点的树。

#### 4.1.4.2 详细轨迹

图 4.1.15 显示的是示例中每个顶点被标记后 `edgeTo[]` 的内容，起点为顶点 0。`marked[]` 和 `adj[]` 的内容与 4.1.3.5 节中的 `DepthFirstSearch` 的轨迹相同，递归调用和边检查的详细描述也完全一样，这里不再赘述。深度优先搜索向 `edgeTo[]` 数组中顺序添加了 0-2、2-1、2-3、3-5 和 3-4。这些边构成了一棵以起点为根结点的树并提供了 `pathTo()` 方法所需的信息，使得调用者可以按照前文所述的方法找到从 0 到顶点 1、2、3、4、5 的路径。

`DepthFirstPaths` 与 `DepthFirstSearch` 的构造函数仅有几条赋值语句不同，因此 4.1.3.2 节中的命题 A 仍然适用。另外，我们还有以下命题。

**命题 A（续）。** 使用深度优先搜索得到从给定起点到任意标记顶点的路径所需的时间与路径的长度成正比。

**证明。** 根据对已经访问过的顶点数量的归纳可得，`DepthFirstPaths` 中的 `edgeTo[]` 数组表示了一棵以起点为根结点的树。`pathTo()` 方法构造路径所需的时间和路径的长度成正比。

537

#### 4.1.5 广度优先搜索

深度优先搜索得到的路径不仅取决于图的结构，还取决于图的表示和递归调用的性质。我们很自然地还经常对下面这些问题感兴趣。

单点最短路径。给定一幅图和一个起点  $s$ ，回答“从  $s$  到给定目的顶点  $v$  是否存在一条路径？如果有，找出其中最短的那条（所含边数最少）。”等类似问题。

解决这个问题的经典方法叫做广度优先搜索 (BFS)。它也是许多图算法的基石，因此我们会

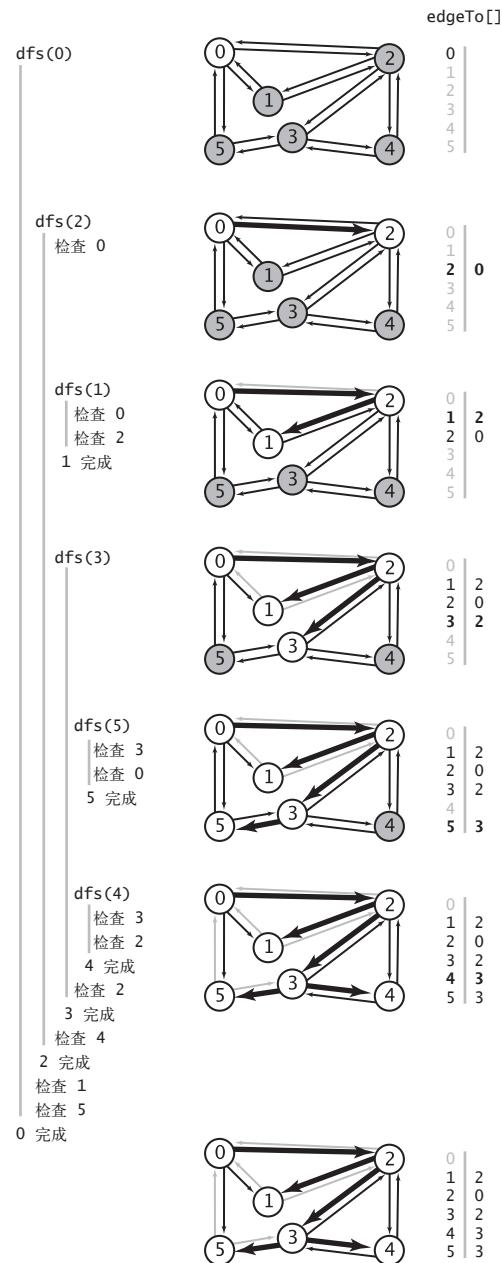


图 4.1.15 使用深度优先搜索的轨迹，寻找所有起点为 0 的路径（另见彩插）

在本节中详细学习。深度优先搜索在这个问题上没有什么作为，因为它遍历整个图的顺序和找出最短路径的目标没有任何关系。相比之下，广度优先搜索正是为了这个目标才出现的。要找到从  $s$  到  $v$  的最短路径，从  $s$  开始，在所有由一条边就可以到达的顶点中寻找  $v$ ，如果找不到我们就继续在与  $s$  距离两条边的所有顶点中查找  $v$ ，如此一直进行。深度优先搜索就好像是一个人在走迷宫，广度优先搜索则好像是一组人在一起朝各个方向走这座迷宫，每个人都有自己的绳子。当出现新的叉路时，可以假设一个探索者可以分裂为更多的人来搜索它们，当两个探索者相遇时，会合二为一（并继续使用先到达者的绳子），参见图 4.1.16。

在程序中，在搜索一幅图时遇到有多条边需要遍历的情况时，我们会选择其中一条并将其他通道留到以后再继续搜索。在深度优先搜索中，我们用了一个可以下压的栈（这是由系统管理的，以支持递归搜索方法）。使用 LIFO（后进先出）的规则来描述压栈和走迷宫时先探索相邻的通道类似。从有待搜索的通道中选择最晚遇到过的那条。在广度优先搜索中，我们希望按照与起点的距离的顺序来遍历所有顶点，看起来这种顺序很容易实现：使用（FIFO，先进先出）队列来代替栈（LIFO，后进先出）即可。我们将从有待搜索的通道中选择最早遇到的那条。

### 实现

算法 4.2 实现了广度优先搜索算法。它使用了一个队列来保存所有已经被标记过但其邻接表还未被检查过的顶点。先将起点加入队列，然后重复以下步骤直到队列为空：

- 取队列中的下一个顶点  $v$  并标记它；
- 将与  $v$  相邻的所有未被标记过的顶点加入队列。

538

算法 4.2 中的 `bfs()` 方法不是递归的。不像递归中隐式使用的栈，它显式地使用了一个队列。和深度优先搜索一样，它的结果也是一个数组 `edgeTo[]`，也是一棵用父链接表示的根结点为  $s$  的树。它表示了  $s$  到每个与  $s$  连通的顶点的最短路径。用例也可以使用算法 4.1 中为深度优先搜索实现的相同的 `pathTo()` 方法得到这些路径。

图 4.1.17 和图 4.1.18 显示了用广度优先搜索处理样图时，算法使用的数据结构在每次循环的迭代开始时的内容。首先，顶点 0 被加入队列，然后循环开始搜索。

- 从队列中删去顶点 0 并将它的相邻顶点 2、1 和 5 加入队列中，标记它们并分别将它们在 `edgeTo[]` 中的值设为 0。
- 从队列中删去顶点 2 并检查它的相邻顶点 0 和 1，发现两者都已经被标记。将相邻的顶点 3 和 4 加入队列，标记它们并分别将它们在 `edgeTo[]` 中的值设为 2。
- 从队列中删去顶点 1 并检查它的相邻顶点 0 和 2，发现它们都已经被标记了。
- 从队列中删去顶点 5 并检查它的相邻顶点 3 和 0，发现它们都已经被标记了。
- 从队列中删去顶点 3 并检查它的相邻顶点 5、4 和 2，发现它们都已经被标记了。
- 从队列中删去顶点 4 并检查它的相邻顶点 3 和 2，发现它们都已经被标记了。

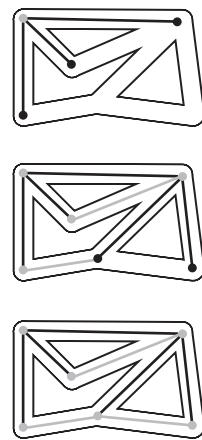


图 4.1.16 广度优先的迷宫搜索

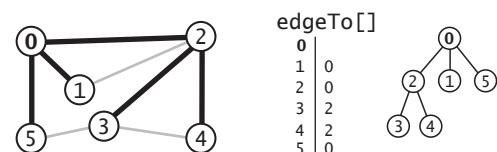
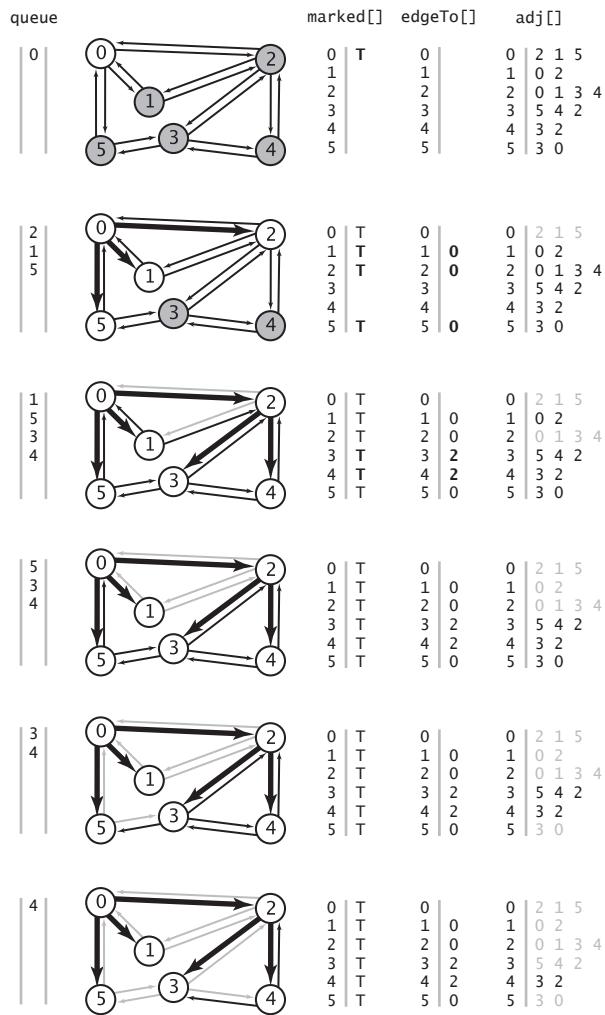


图 4.1.17 使用广度优先搜索寻找所有起点为 0 的路径的结果



[539]

图 4.1.18 使用广度优先搜索的轨迹，寻找所有起点为 0 的路径（另见彩插）

#### 算法 4.2 使用广度优先搜索查找图中的路径

```
public class BreadthFirstPaths
{
    private boolean[] marked; // 到达该顶点的最短路径已知吗?
    private int[] edgeTo; // 到达该顶点的已知路径上的最后一个顶点
    private final int s; // 起点

    public BreadthFirstPaths(Graph G, int s)
    {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        this.s = s;
        bfs(G, s);
    }
}
```

```

private void bfs(Graph G, int s)
{
    Queue<Integer> queue = new Queue<Integer>();
    marked[s] = true;           // 标记起点
    queue.enqueue(s);          // 将它加入队列
    while (!queue.isEmpty())
    {
        int v = queue.dequeue(); // 从队列中删去下一顶点
        for (int w : G.adj(v))
            if (!marked[w])      // 对于每个未被标记的相邻顶点
            {
                edgeTo[w] = v;   // 保存最短路径的最后一条边
                marked[w] = true; // 标记它，因为最短路径已知
                queue.enqueue(w); // 并将它添加到队列中
            }
    }
}

public boolean hasPathTo(int v)
{ return marked[v]; }

public Iterable<Integer> pathTo(int v)
// 和深度优先搜索中的实现相同（请见算法4.1）
}

```

```

% java BreadthFirstPaths
tinyCG.txt 0
0 to 0: 0
0 to 1: 0-1
0 to 2: 0-2
0 to 3: 0-2-3
0 to 4: 0-2-4
0 to 5: 0-5

```

这段 `Graph` 的用例使用了广度优先搜索，以找出图中从构造函数得到的起点 `s` 到与其他所有顶点的最短路径。`bfs()` 方法会标记所有与 `s` 连通的顶点，因此用例可以调用 `hasPathTo()` 来判定一个顶点与 `s` 是否连通并使用 `pathTo()` 得到一条从 `s` 到 `v` 的路径，确保没有其他从 `s` 到 `v` 的路径所含的边比这条路径更少。

540

对于这个例子来说，`edgeTo[]` 数组在第二步之后就已经完成了。和深度优先搜索一样，一旦所有的顶点都已经被标记，余下的计算工作就只是在检查连接到各个已被标记的顶点的边而已。

**命题 B。** 对于从 `s` 可达的任意顶点 `v`，广度优先搜索都能找到一条从 `s` 到 `v` 的最短路径（没有其他从 `s` 到 `v` 的路径所含的边比这条路径更少）。

**证明。** 由归纳易得队列总是包含零个或多个到起点的距离为  $k$  的顶点，之后是零个或多个到起点的距离为  $k+1$  的顶点，其中  $k$  为整数，起始值为 0。这意味着顶点是按照它们和 `s` 的距离的顺序加入或者离开队列的。从顶点 `v` 加入队列到它离开队列之前，不可能找到到 `v` 的更短的路径，而在 `v` 离开队列之后发现的所有能够到达 `v` 的路径都不可能短于 `v` 在树中的路径长度。

**命题 B（续）。** 广度优先搜索所需的时间在最坏情况下和  $V+E$  成正比。

**证明。** 和命题 A 一样（请见 4.1.3.2 节），广度优先搜索标记所有与 `s` 连通的顶点所需的时间也与它们的度数之和成正比。如果图是连通的，这个和就是所有顶点的度数之和，也就是  $2E$ 。

注意，我们也可以用广度优先搜索来实现已经用深度优先搜索实现的 Search API，因为它检查所有与起点连通的顶点和边的方法只取决于查找的能力。

我们在本章开头说过，深度优先搜索和广度优先搜索是我们首先学习的几种通用的图搜索的算

法之一。在搜索中我们都会先将起点存入数据结构中，然后重复以下步骤直到数据结构被清空：

- 取其中的下一个顶点并标记它；
- 将  $v$  的所有相邻而又未被标记的顶点加入数据结构。

这两个算法的不同之处仅在于从数据结构中获取下一个顶点的规则（对于广度优先搜索来说是最早加入的顶点，对于深度优先搜索来说是最晚加入的顶点）。这种差异得到了处理图的两种完全不同的视角，尽管无论使用哪种规则，所有与起点连通的顶点和边都会被检查到。

图 4.1.19 和图 4.1.20 显示了深度优先搜索和广度优先搜索处理样图 mediumG.txt 的过程，它们清晰地展示了两种方法中搜索路径的不同。深度优先搜索不断深入图中并在栈中保存了所有分叉的顶点；广度优先搜索则像扇面一般扫描图，用一个队列保存访问过的最前端的顶点。深度优先搜索探索一幅图的方式是寻找离起点更远的顶点，只在碰到死胡同时才访问近处的顶点；广度优先搜索则会首先覆盖起点附近的顶点，只在临近的所有顶点都被访问了之后才向前进。深度优先搜索的路径通常较长而且曲折，广度优先搜索的路径则短而直接。根据应用的不同，所需要的性质也会有所不同（也许路径的性质也会变得无关紧要）。在 4.4 节中，我们会学习 Paths 的 API 的其他实现来寻找有特定属性的路径。

541  
542

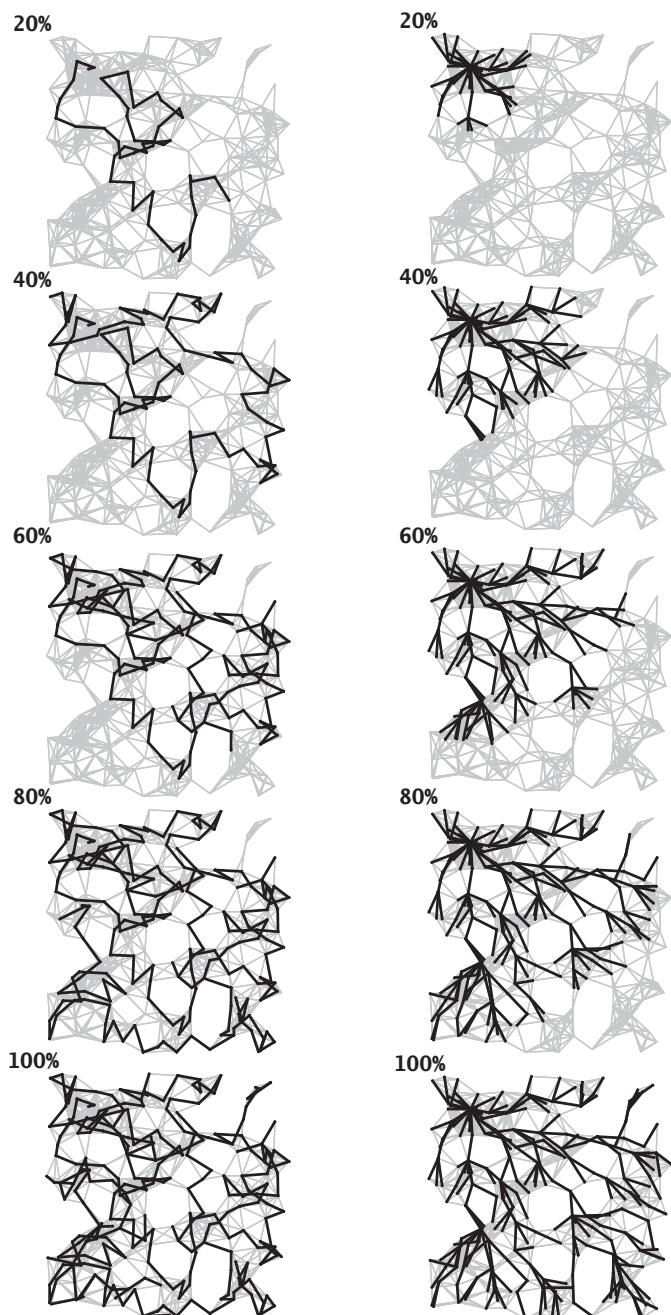


图 4.1.19 使用深度优先搜索查找路径（250 个顶点）

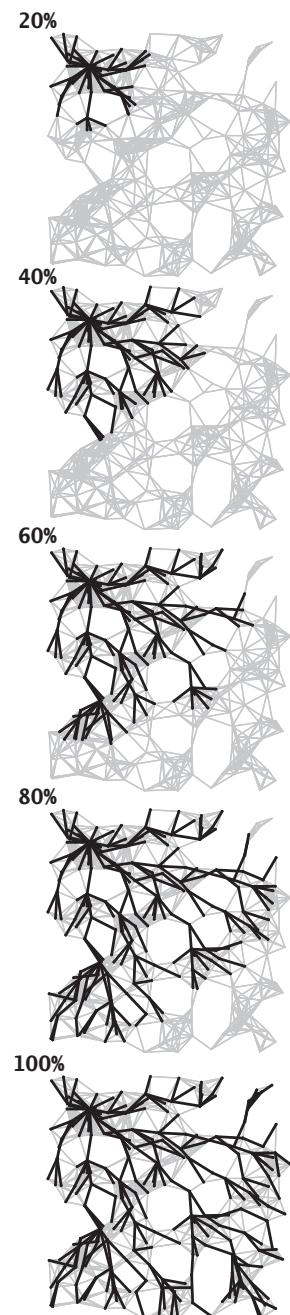


图 4.1.20 使用广度优先搜索查找最短路径（250 个顶点）

### 4.1.6 连通分量

深度优先搜索的下一个直接应用就是找出一幅图的所有连通分量。回忆 1.5 节中“与……连通”是一种等价关系，它能够将所有顶点切分为等价类（连通分量）。对于这个常见的任务，我们定义如下 API（请见表 4.1.6）。

表 4.1.6 连通分量的 API

<code>public class CC</code>	
<code>    CC(Graph G)</code>	预处理构造函数
<code>    boolean connected(int v, int w)</code>	v 和 w 连通吗
<code>    int count()</code>	连通分量数
<code>    int id(int v)</code>	v 所在的连通分量的标识符 (0 ~ count()-1)

用例可以用 `id()` 方法将连通分量用数组保存，如框注中的用例所示。它能够从标准输入中读取一幅图并打印其中的连通分量数，其后是每个子图中的所有顶点，每行一个子图。为了实现这些，它使用了一个 `Bag` 对象数组，然后用每个顶点所在的子图的标识符作为数组的索引，以将所有顶点加入相应的 `Bag` 对象中。当我们希望独立处理每个连通分量时这个用例就是一个模型。

#### 4.1.6.1 实现

CC 的实现（请见算法 4.3）使用了 `marked[]` 数组来寻找一个顶点作为每个连通分量中深度优先搜索的起点。递归的深度优先搜索第一次调用的参数是顶点 0——它会标记所有与 0 连通的顶点。然后构造函数中的 `for` 循环会查找每个没有被标记的顶点并递归调用 `dfs()` 来标记和它相邻的所有顶点。另外，它还使用了一个以顶点作为索引的数组 `id[]`，将同一个连通分量中的顶点和连通分量的标识符关联起来（`int` 值）。这个数组使得 `connected()` 方法的实现变得十分简单，和 1.5 节中的 `connected()` 方法完全相同（只需检查标识符是否相同）。这里，标识符 0 会被赋予第一个连通分量中的所有顶点，1 会被赋予第二个连通分量中的所有顶点，依此类推。这样所有的标识符都会如 API 中指定的那样在 0 到 `count()-1` 之间。这个约定使得以子图作为索引的数组成为可能，如右侧框注用例所示。

```
public static void main(String[] args)
{
    Graph G = new Graph(new In(args[0]));
    CC cc = new CC(G);

    int M = cc.count();
    StdOut.println(M + " components");

    Bag<Integer>[] components;
    components = (Bag<Integer>[]) new Bag[M];
    for (int i = 0; i < M; i++)
        components[i] = new Bag<Integer>();
    for (int v = 0; v < G.V(); v++)
        components[cc.id(v)].add(v);
    for (int i = 0; i < M; i++)
    {
        for (int v: components[i])
            StdOut.print(v + " ");
        StdOut.println();
    }
}
```

查找连通分量API的测试用例

543

#### 算法 4.3 使用深度优先搜索找出图中的所有连通分量

```
public class CC
{
    private boolean[] marked;
    private int[] id;
    private int count;
```

```

public CC(Graph G)
{
    marked = new boolean[G.V()];
    id = new int[G.V()];
    for (int s = 0; s < G.V(); s++)
        if (!marked[s])
            {
                dfs(G, s);
                count++;
            }
    }

private void dfs(Graph G, int v)
{
    marked[v] = true;
    id[v] = count;
    for (int w : G.adj(v))
        if (!marked[w])
            dfs(G, w);
}

public boolean connected(int v, int w)
{ return id[v] == id[w]; }

public int id(int v)
{ return id[v]; }

public int count()
{ return count; }

}

```

```

% more tinyG.txt
13 vertices, 13 edges
0: 6 2 1 5
1: 0
2: 0
3: 5 4
4: 5 6 3
5: 3 4 0
6: 0 4
7: 8
8: 7
9: 11 10 12
10: 9
11: 9 12
12: 11 9

% java CC tinyG.txt
3 components
6 5 4 3 2 1 0
8 7
12 11 10 9

```

这段 `Graph` 的用例使得它的用例可以独立处理一幅图中的每个连通分量。来自 `DepthfirstSearch` (请见 4.1.3.2 节) 的代码均为灰色。这里的实现是基于一个由顶点索引的数组 `id[]`。如果 `v` 属于第 `i` 个连通分量，则 `id[v]` 的值为 `i`。构造函数会找出一个未被标记的顶点并调用递归函数 `dfs()` 来标记并区分出所有和它连通的顶点，如此重复直到所有的顶点都被标记并区分。`connected()`、`count()` 和 `id()` 方法的实现非常简单 (另见图 4.1.21)。

544

**命题 C。** 深度优先搜索的预处理使用的时间和空间与  $V+E$  成正比且可以在常数时间内处理关于图的连通性查询。

**证明。** 由代码可以知道每个邻接表的元素都只会被检查一次，共有  $2E$  个元素 (每条边两个)。实例方法会检查或者返回一个或两个变量。

#### 4.1.6.2 union-find 算法

CC 中基于深度优先搜索来解决图连通性问题的方法与第 1 章中的 union-find 算法相比孰优孰劣？理论上，深度优先搜索比 union-find 法快，因为它能保证所需的时间是常数而 union-find 算法不行；但在实际应用中，这点差异微不足道。union-find 算法其实更快，因为它不需要完整地构造并表示一幅图。更重要的是，union-find 算法是一种动态算法 (我们在任何时候都能用接近常数的时间检查两个顶点是否连通，甚至是在添加一条边的时候)，但深度优先搜索则必须要对图进行预处理。因此，我们在完成只需要判断连通性或是需要完成有大量连通性查询和插入操作混合等类似的任务时，更倾向使用 union-find 算法，而深度优先搜索则更适合实现图的抽象数据类型，因为它能更有效地利用已有的数据结构。

		marked[]									id[]																
		0	1	2	3	4	5	6	7	8	9	10	11	12	0	1	2	3	4	5	6	7	8	9	10	11	12
dfs(0)		0	T								0				0												
dfs(6)		0	T					T			0				0												
检查 0																											
dfs(4)		0	T				T	T			0				0		0	0	0								
dfs(5)		0	T			T	T	T			0				0		0	0	0								
dfs(3)		0	T			T	T	T	T		0				0		0	0	0	0							
检查 5																											
检查 4																											
3 完成																											
检查 4																											
检查 0																											
5 完成																											
检查 6																											
检查 3																											
4 完成																											
6 完成																											
dfs(2)		0		T	T	T	T	T	T		0				0	0	0	0	0	0							
检查 0																											
2 完成																											
dfs(1)		0			T	T	T	T	T	T	0				0	0	0	0	0	0	0						
检查 0																											
1 完成																											
检查 5																											
0 完成																											
dfs(7)		1			T	T	T	T	T	T	0	0	0	0	0	0	0	1									
dfs(8)		1				T	T	T	T	T	T	0	0	0	0	0	0	0	1	1							
检查 7																											
8 完成																											
7 完成																											
dfs(9)		2				T	T	T	T	T	T	0	0	0	0	0	0	0	1	2							
dfs(11)		2					T	T	T	T	T	T	T	T	0	0	0	0	0	0	1	2					
检查 9																											
dfs(12)		2						T	T	T	T	T	T	T	T	0	0	0	0	0	0	1	2	2			
检查 11																											
检查 9																											
12 完成																											
11 完成																											
dfs(10)		2							T	T	T	T	T	T	T	T	0	0	0	0	0	0	1	2	2		
检查 9																											
10 完成																											
检查 12																											
9 完成																											

图 4.1.21 使用深度优先搜索的轨迹，寻找所有连通分量

我们已经用深度优先搜索解决了几个非常基础的问题。这种方法很简单，递归实现使我们能够进行复杂的运算并为一些图的处理问题给出简洁的解决方法。在表 4.1.7 中，我们为下面两个问题作出了解答。

检测环。给定的图是无环图吗？

双色问题。能够用两种颜色将图的所有顶点着色，使得任意一条边的两个端点的颜色都不相同吗？这个问题也等价于：这是一幅二分图吗？

深度优先搜索和已学习过的其他算法一样，它简洁的代码下隐藏着复杂的计算。因此，研究这些例子、在样图中跟踪算法的轨迹并加以扩展、用算法来解决环和着色的问题都是非常值得的（留作练习）。

表 4.1.7 使用深度优先搜索处理图的其他示例

任 务	实 现
G 是无环图吗? (假设不存在自环或平行边)	<pre> public class Cycle {     private boolean[] marked;     private boolean hasCycle;     public Cycle(Graph G)     {         marked = new boolean[G.V()];         for (int s = 0; s &lt; G.V(); s++)             if (!marked[s])                 dfs(G, s, s);     }      private void dfs(Graph G, int v, int u)     {         marked[v] = true;         for (int w : G.adj(v))             if (!marked[w])                 dfs(G, w, v);             else if (w != u) hasCycle = true;     }      public boolean hasCycle()     { return hasCycle; } } </pre>
G 是二分图吗? (双色问题)	<pre> public class TwoColor {     private boolean[] marked;     private boolean[] color;     private boolean isTwoColorable = true;     public TwoColor(Graph G)     {         marked = new boolean[G.V()];         color = new boolean[G.V()];         for (int s = 0; s &lt; G.V(); s++)             if (!marked[s])                 dfs(G, s);     }      private void dfs(Graph G, int v)     {         marked[v] = true;         for (int w : G.adj(v))             if (!marked[w])             {                 color[w] = !color[v];                 dfs(G, w);             }             else if (color[w] == color[v]) isTwoColorable = false;     }      public boolean isBipartite()     { return isTwoColorable; } } </pre>

[547]

### 4.1.7 符号图

在典型应用中, 图都是通过文件或者网页定义的, 使用的是字符串而非整数来表示和指代顶点。为了适应这样的应用, 我们定义了拥有以下性质的输入格式:

- 顶点名为字符串;

- 用指定的分隔符来隔开顶点名（允许顶点名中含有空格）；
- 每一行都表示一组边的集合，每一条边都连接着这一行的第一个名称表示的顶点和其他名称所表示的顶点；
- 顶点总数  $V$  和边的总数  $E$  都是隐式定义的。

图 4.1.22 是一个简单的示例。Routes.txt 文件表示的是一个小型运输系统的模型，其中表示每个顶点的是美国机场的代码，连接它们的边则表示顶点之间的航线。文件只是一组边的列表。图 4.1.23 所示的是一个更庞大的例子，取自 movies.txt，即 3.5 节中介绍的互联网电影数据库。还记得吗？这个文件的每一行都列出了一个电影名以及出演该部电影的一系列演员。从图的角度来说，我们可以将它看作一幅图的定义，电影和演员都是顶点，而邻接表中的每一条边都将电影和它的表演者联系起来。注意，这是一幅二分图——电影顶点之间或者演员结点之间都没有边相连。

#### 4.1.7.1 API

表 4.1.8 中，API 定义的 Graph 用例可以直接使用已有的图算法来处理这种文件定义的图。

表 4.1.8 用符号作为顶点名的图的 API

<code>public class SymbolGraph</code>	
<code>SymbolGraph(String filename,               String delim)</code>	根据 <code>filename</code> 指定的文件构造图，使用 <code>delim</code> 来分隔顶点名
<code>boolean contains(String key)</code>	<code>key</code> 是一个顶点吗
<code>int index(String key)</code>	<code>key</code> 的索引
<code>String name(int v)</code>	索引 <code>v</code> 的顶点名
<code>Graph G()</code>	隐藏的 Graph 对象

548

这份 API 定义了一个构造函数来读取并构造图，用 `name()` 方法和 `index()` 方法将输入流中的顶点名和图算法使用的顶点索引对应起来。

#### 4.1.7.2 测试用例

下一页框注所示的是符号图的测试用例，它用第一个命令行参数指定的文件（第二个命令行参数指定了分隔符）来构造一幅图并从标准输入接受查询。用户可以输入一个顶点名并得到该顶点的相邻结点的列表。这个用例提供的正好是 3.5 节中研究过的反向索引的功能。以 routes.txt 为例，你可以输入一个机场的代码来查找能从该机场直飞到达的城市，但这些信息并不是直接就能从文件中得到的。对于 movies.txt，你可以输入一个演员的名字来查看数据库中他所出演的影片列表。输入一部电影的名字来得到它的演员列表，这不过是在照搬文件中对应行数据，但输入演员的名字来得到影片的列表则相当于查找反向索引。尽管数据库的构造是为了将电影名连接到演员，二分图模型同时也意味着将演员连接到电影名。二分图的性质自动完成了反向索引。以后我们将会看到，这将成为处理更复杂的和图有关的问题的基础。

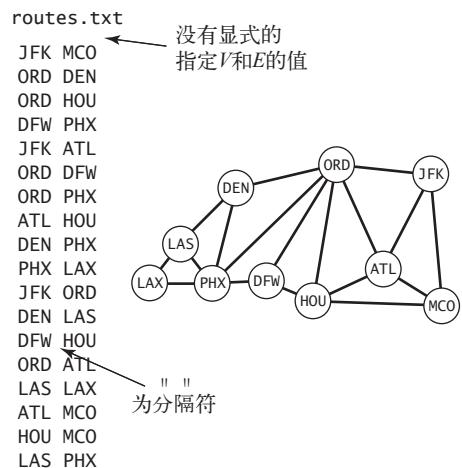


图 4.1.22 符号图示例（边的列表）

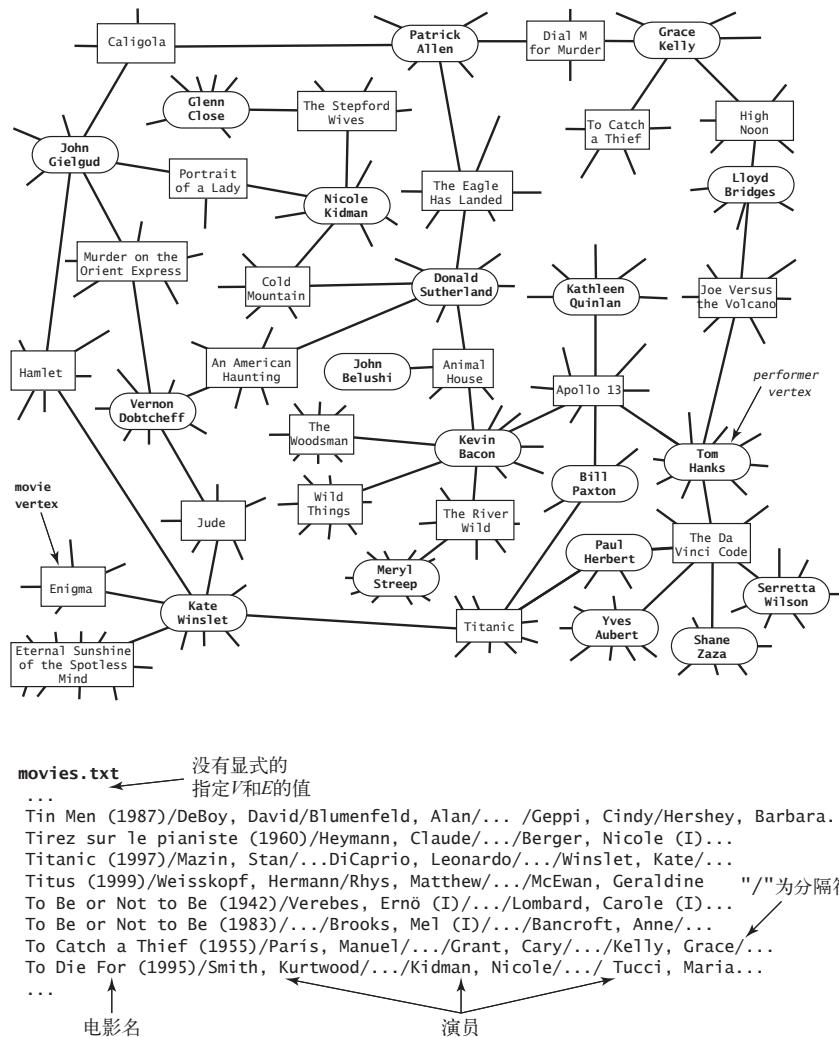


图 4.1.23 符号图示例（邻接表）

```

public static void main(String[] args)
{
    String filename = args[0];
    String delim = args[1];
    SymbolGraph sg = new SymbolGraph(filename, delim);

    Graph G = sg.G();

    while (StdIn.hasNextLine())
    {
        String source = StdIn.readLine();
        for (int w : G.adj(sg.index(source)))
            StdOut.println(" " + sg.name(w));
    }
}

```

符号图API的测试用例

```
% java SymbolGraph routes.txt " "
JFK
ORD
ATL
MCO
LAX
LAS
PHX
```

```
% java SymbolGraph movies.txt "/"
Tin Men (1987)
DeBoy, David
Blumenfeld, Alan
...
Geppi, Cindy
Hershey, Barbara
...
Bacon, Kevin
Mystic River (2003)
Friday the 13th (1980)
Flatliners (1990)
Few Good Men, A (1992)
...
```

549  
550

很显然，这种方法适用于我们遇到过的所有图算法：用例可以用 `index()` 将顶点名转化为索引并在图的处理算法中使用，然后将处理结果用 `name()` 转化为顶点名以方便在实际应用中使用。

#### 4.1.7.3 实现

`SymbolGraph` 的完整实现请见下面的框注“符号图的数据类型”。它用到了以下 3 种数据结构，参见图 4.1.24。

- 一个符号表 `st`，键的类型为 `String`（顶点名），值的类型为 `int`（索引）；
- 一个数组 `keys[]`，用作反向索引，保存每个顶点索引所对应的顶点名；
- 一个 `Graph` 对象 `G`，它使用索引来引用图中顶点。

`SymbolGraph` 会遍历两遍数据来构造以上数据结构，这主要是因为构造 `Graph` 对象需要顶点总数  $V$ 。在典型的应用中，在定义图的文件中指明  $V$  和  $E$ （见本节开头 `Graph` 的构造函数）可能会有些不便，而有了 `SymbolGraph`，我们就可以方便地在 `routes.txt` 或者 `movies.txt` 中添加或者删除条目而不用担心需要维护边或顶点的总数。

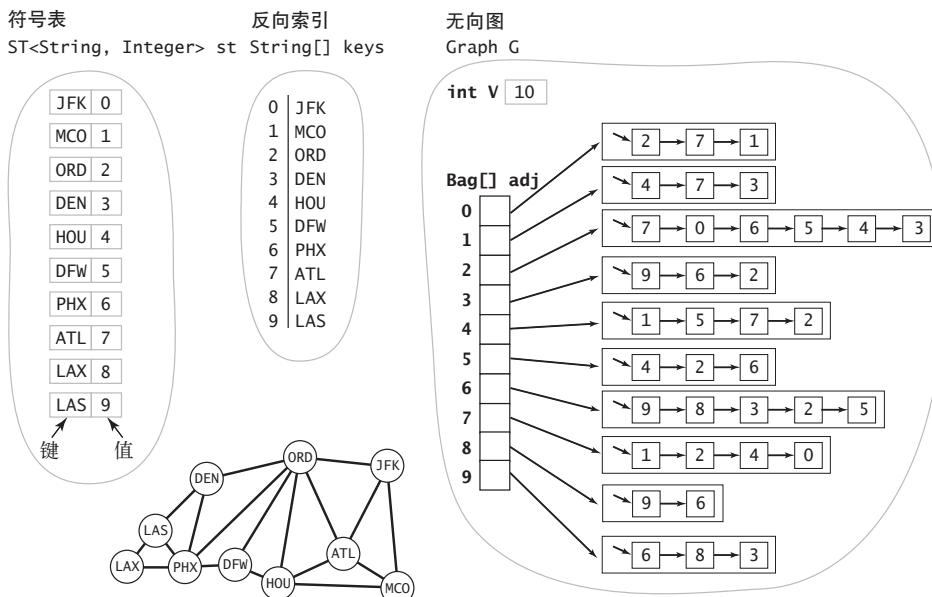


图 4.1.24 符号图中用到的数据结构

### 符号图的数据类型

```

public class SymbolGraph
{
    private ST<String, Integer> st;           // 符号名 → 索引
    private String[] keys;                      // 索引 → 符号名
    private Graph G;                           // 图

    public SymbolGraph(String stream, String sp)
    {
        st = new ST<String, Integer>();
        In in = new In(stream);                  // 第一遍
        while (in.hasNextLine())                // 构造索引
        {
            String[] a = in.readLine().split(sp); // 读取字符串
            for (int i = 0; i < a.length; i++)   // 为每个不同的字符串关联一个索引
                if (!st.contains(a[i]))
                    st.put(a[i], st.size());
        }
        keys = new String[st.size()];           // 用来获得顶点名的反向索引是一个数组
        for (String name : st.keys())
            keys[st.get(name)] = name;

        G = new Graph(st.size());
        in = new In(stream);                  // 第二遍
        while (in.hasNextLine())                // 构造图
        {
            String[] a = in.readLine().split(sp); // 将每一行的顶点和该行的其他顶点相连
            int v = st.get(a[0]);
            for (int i = 1; i < a.length; i++)
                G.addEdge(v, st.get(a[i]));
        }
    }

    public boolean contains(String s) { return st.contains(s); }
    public int index(String s)      { return st.get(s); }
    public String name(int v)       { return keys[v]; }
    public Graph G()               { return G; }
}

```

这个 `Graph` 实现允许用字符串代替数字索引来表示图中的顶点。它维护了实例变量 `st`（符号表用来映射顶点名和索引）、`keys`（数组用来映射索引和顶点名）和 `G`（使用索引表示顶点的图）。为了构造这些数据结构，代码会将图的定义处理两遍（定义的每一行都包含一个顶点及它的相邻顶点列表，用分隔符 `sp` 隔开）。

552

#### 4.1.7.4 间隔的度数

图处理的一个经典问题就是，找到一个社交网络之中两个人间隔的度数。为了弄清楚概念，我们用一个最近很流行的名为 *Kevin Bacon* 的游戏来说明这个问题。这个游戏用到了刚才讨论的“电影 – 演员”图。*Kevin Bacon* 是一个活跃的演员，曾出演过许多电影。我们为图中的每个演员赋一个 *Kevin Bacon* 数：*Bacon* 本人为 0，所有和 *Kevin Bacon* 出演过同一部电影的人的值为 1，所有（除了 *Kevin Bacon*）和 *Kevin Bacon* 数为 1 的演员出演过同一部电影的其他演员的值为 2，依次类推。例如，*Meryl Streep* 的 *Kevin Bacon* 数为 1，因为她和 *Kevin Bacon* 一同出演过 *The River Wild*。

Nicole Kidman 的值为 2，因为她虽然没有和 Kevin Bacon 同台演出过任何电影，但她和 Tom Cruise 一起演过 *Days of Thunder*，而 Tom Cruise 和 Kevin Bacon 一起演过 *A Few Good Men*。给定一个演员的名字，游戏最简单的玩法就是找出一系列的电影和演员来回溯到 Kevin Bacon。例如，有些影迷可能知道 Tom Hanks 和 Lloyd Bridges 一起演过 *Joe Versus the Volcano*，而 Bridges 和 Grace Kelly 一起演过 *High Noon*，Kelly 又和 Patrick Allen 一起演过 *Dial M for Murder*，Allen 和 Donald Sutherland 一起演过 *The Eagle has Landed*，Sutherland 和 Kevin Bacon 一起出演了 *Animal House*。但知道这些也并不足以确定 Tom Hanks 的 Kevin Bacon 数。（他的值实际上应该是 1，因为他和 Kevin Bacon 在 *Apollo 13* 中合作过）。你可以看到 Kevin Bacon 数必须定义为最短电影链的长度，因此如果不使用计算机，人们很难知道游戏中到底谁赢了。当然，如后面标注“间隔的度数”中 **SymbolGraph** 的用例 **DegreesOfSeparation** 所示，**BreadthFirstPaths** 才是我们所要的程序，它通过最短路径来找出 movies.txt 中任意演员的 Kevin Bacon 数。这个程序从命令行得到一个起点，从标准输入中接受查询并打印出一条从起点到被查询顶点的最短路径。因为 movies.txt 所构造的是一幅二分图，每条路径上都会交替出现电影和演员的顶点。打出的结果可以证明这样的路径是存在的（但并不能证明它是最短的——你需要向你的朋友证明命题 B 才行）。**DegreesOfSeparation** 也能够在非二分图中找到最短路径。例如，在 routes.txt 中，它能够用最少的边找到一种从一个机场到达另一个机场的方法。

```
% java DegreesOfSeparation movies.txt "/" "Bacon, Kevin"
Kidman, Nicole
Bacon, Kevin
Few Good Men, A (1992)
Cruise, Tom
Days of Thunder (1990)
Kidman, Nicole
Grant, Cary
Bacon, Kevin
Mystic River (2003)
Willis, Susan
Majestic, The (2001)
Landau, Martin
North by Northwest (1959)
Grant, Cary
```

553

你可能会发现用 **DegreesOfSeparation** 来回答一些关于电影行业的问题很有趣。例如，你不但可以找到演员和演员之间的间隔，还可以找到电影和电影之间的间隔。更重要的是，间隔的概念在其他许多领域也被广泛研究。例如，数学家也会玩这个游戏，但他们的图是用一些论文的作者到 P.Erdős (20 世纪的一位多产的数学家) 的距离来定义的。类似地，似乎新泽西州的每个人的 Bruce Springsteen 数都为 2，因为每个人都声称自己认识某个认识 Bruce 的人。要玩 Erdős 的游戏，你需要一个包含所有数学论文的数据库；要玩 Sprintsteen 的游戏还要困难一些。从更严肃的角度来说，间隔度数的理论在计算机网络的设计以及理解各个科学领域中的自然网络中都能起到重要的作用。

554

```
% java DegreesOfSeparation movies.txt "/" "Animal House (1978)"
Titanic (1997)
    Animal House (1978)
    Allen, Karen (I)
    Raiders of the Lost Ark (1981)
    Taylor, Rocky (I)
    Titanic (1997)
    To Catch a Thief (1955)
    Animal House (1978)
    Vernon, John (I)
    Topaz (1969)
    Hitchcock, Alfred (I)
    To Catch a Thief (1955)
```

### 间隔的度数

```
public class DegreesOfSeparation
{
    public static void main(String[] args)
    {
        SymbolGraph sg = new SymbolGraph(args[0], args[1]);
        Graph G = sg.G();

        String source = args[2];
        if (!sg.contains(source))
        { StdOut.println(source + " not in database."); return; }

        int s = sg.index(source);
        BreadthFirstPaths bfs = new BreadthFirstPaths(G, s);

        while (!StdIn.isEmpty())
        {
            String sink = StdIn.readLine();
            if (sg.contains(sink))
            {
                int t = sg.index(sink);
                if (bfs.hasPathTo(t))
                    for (int v : bfs.pathTo(t))
                        StdOut.println(" " + sg.name(v));
                else StdOut.println("Not connected");
            }
            else StdOut.println("Not in database.");
        }
    }
}
```

```
% java
DegreesOfSeparation
routes.txt " " JFK
LAS
    JFK
    ORD
    PHX
    LAS
DFW
    JFK
    ORD
    DFW
```

这段代码使用了 `SymbolGraph` 和 `BreadthFirstPath` 来查找图中的最短路径。对于 `movies.txt`，可以用它来玩 Kevin Bacon 游戏。

555

### 4.1.8 总结

在本节中，我们介绍了几个基本的概念，本章的其余部分会继续扩展并研究：

- 图的术语；
- 一种图的表示方法，能够处理大型而稀疏的图；
- 和图处理相关的类的设计模式，其实现算法通过在相关的类的构造函数中对图进行预处理、构造所需的数据结构来高效支持用例对图的查询；

- 深度优先搜索和广度优先搜索；
- 支持使用符号作为图的顶点名的类。

表 4.1.9 总结了我们已经学习过的所有图算法的实现。这些算法非常适合作为图处理的入门学习。随后学习更加复杂类型的图以及处理更加困难的问题时，我们还会用到这些代码的变种。在考虑了边的方向以及权重之后，同样的问题会变得困难得多，但同样的算法仍然奏效并将成为解决更加复杂问题的起点。

表 4.1.9 本节中得到解决的无向图处理问题

问 题	解决方法	参 阅
单点连通性	<code>DepthFirstSearch</code>	4.1.3.2 节
单点路径	<code>DepthFirstPaths</code>	算法 4.1
单点最短路径	<code>BreadthFirstPaths</code>	算法 4.2
连通性	<code>CC</code>	算法 4.3
检测环	<code>Cycle</code>	表 4.1.7
双色问题（图的二分性）	<code>TwoColor</code>	表 4.1.7

556

## 答疑

问 为什么不把所有的算法都实现在 `Graph.java` 中？

答 可以这么做，可以向基本的 `Graph` 抽象数据类型的定义中添加查询方法（以及它们需要的私有变量和方法等）。尽管这种方式可以用到一些我们所使用的数据结构的优点，它还是有一些严重的缺陷，因为图处理的成本比 1.3 节中遇到那些基本数据结构要高得多。这些缺点主要有：

- 在图处理中，需要实现的操作还有很多，我们无法在一份 API 中全部精确地定义它们；
- 简单任务的 API 和复杂任务所使用的 API 是相同的；
- 一个方法将可以访问另外一个方法专用的变量，这有悖我们需要遵守的封装原则。

这种情况并不罕见：这种 API 被称为宽接口（请见 1.2.5.2 节）。本章包含如此众多的图算法，将导致这种 API 变得非常宽。

问 `SymbolGraph` 真需要将图的定义遍历两遍吗？

答 不，你也可以将用时增加  $\lg N$  并直接用 `ST` 而非 `Bag` 来实现 `adj()`。我们的另一本书 *An Introduction to Programming in Java: An Interdisciplinary Approach* 中含有使用这种方法的一个实现。

557

## 练习

- 4.1.1 一幅含有  $V$  个顶点且不含有平行边的图中至多含有多少条边？一幅含有  $V$  个顶点的连通图中至少含有多少条边？
- 4.1.2 按照正文中示意图的样式（请见图 4.1.9）画出 `Graph` 的构造函数在处理图 4.1.25 的 `tinyGex2.txt` 时构造的邻接表。
- 4.1.3 为 `Graph` 添加一个复制构造函数，它接受一幅图 `G` 然后创建并初始化这幅图的一个副本。`G` 的用例对它作出的任何改动都不应该影响到它的副本。
- 4.1.4 为 `Graph` 添加一个方法 `hasEdge()`，它接受两个整型参数 `v` 和 `w`。如果图含有边 `v-w`，方法返回 `true`，否则返回 `false`。

- 4.1.5 修改 Graph，不允许存在平行边和自环。
- 4.1.6 有一张含有四个顶点的图，其中的边为 0-1、1-2、2-3 和 3-0。给出一种邻接表数组，无论以任何顺序调用 addEdge() 来添加这些边都无法创建它。
- 4.1.7 为 Graph 编写一个测试用例，用命令行参数命名并从输入流中接受一幅图，然后用 toString() 方法将其打印出来。
- 4.1.8 按照正文中的要求，用 union-find 算法实现 4.1.2.3 中搜索的 API。
- 4.1.9 使用 dfs() 处理由 Graph 的构造函数从 tinyGex2.txt (请见练习 4.1.2) 得到的图并按照 4.1.3.5 节的图 4.1.14 的样式给出详细的轨迹。同时，画出 edgeTo[] 所表示的树。
- 4.1.10 证明在任意一幅连通图中都存在一个顶点，删去它（以及和它相连的所有边）不会影响到图的连通性，编写一个深度优先搜索的方法找出这样一个顶点。提示：留心那些相邻顶点全部都被标记过的顶点。
- 4.1.11 使用算法 4.2 中的 bfs(G, 0) 处理由 Graph 的构造函数从 tinyGex2.txt (请见练习 4.1.2) 得到的图并画出 edgeTo[] 所表示的树。
- 4.1.12 如果 v 和 w 都不是根结点，能够由广度优先搜索得到的树中计算它们之间的距离吗？
- 4.1.13 为 BreadthFirstPaths 的 API 添加并实现一个方法 distTo(), 返回从起点到给定的顶点的最短路径的长度，它所需的时间应该为常数。
- 4.1.14 如果用栈代替队列来实现广度优先搜索，我们还能得到最短路径吗？
- 4.1.15 修改 Graph 的输入流构造函数，允许从标准输入读入图的邻接表（方法类似于 SymbolicGraph），如图 4.1.26 的 tinyGadj.txt 所示。在顶点和边的总数之后，每一行由一个顶点和它的所有相邻顶点组成。

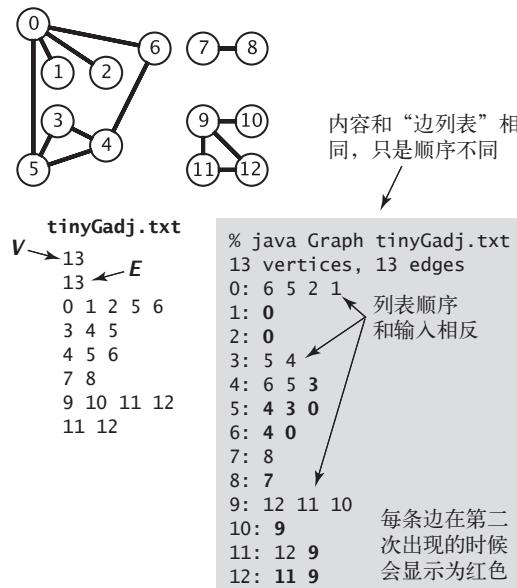
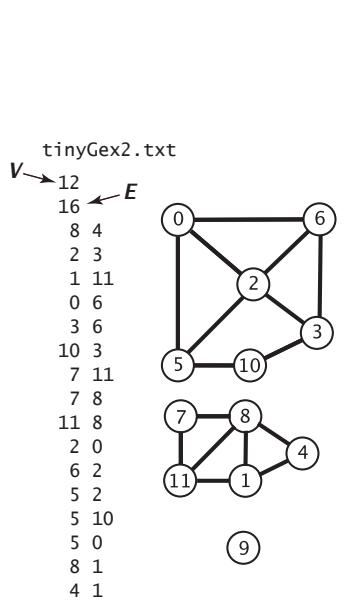


图 4.1.25

图 4.1.26

- 4.1.16 顶点 v 的离心率是它和离它最远的顶点的最短距离。图的直径即所有顶点的最大离心率，半径为所有顶点的最小离心率，中点为离心率和半径相等的顶点。实现以下 API，如表 4.1.10 所示。

表 4.1.10

<b>public class GraphProperties</b>	
<b>GraphProperties(Graph G)</b>	构造函数 (如果 G 不是连通的, 抛出异常)
<b>int eccentricity(int v)</b>	v 的离心率
<b>int diameter()</b>	G 的直径
<b>int radius()</b>	G 的半径
<b>int center()</b>	G 的某个中点

- 4.1.17 图的周长为图中最短环的长度。如果是无环图, 则它的周长为无穷大。为 **GraphProperties** 添加一个方法 **girth()**, 返回图的周长。提示: 在每个顶点都进行广度优先搜索。含有 s 的最小环为 s 到某个顶点 v 的最短距离加上 v 到 s 的最短距离。
- 4.1.18 使用 **CC** 找出由 **Graph** 的输入流构造函数从 **tinyGex2.txt** (请见练习 4.1.2) 得到的图中的所有连通分量并按照图 4.1.21 的样式给出详细的轨迹。
- 4.1.19 使用 **Cycle** 在由 **Graph** 的输入流构造函数从 **tinyGex2.txt** (请见练习 4.1.2) 得到的图中找到的一个环并按照本节示意图的样式给出详细的轨迹。在最坏情况下, **Cycle** 构造函数的运行时间的增长数量级是多少?
- 4.1.20 使用 **TwoColor** 给出由 **Graph** 的构造函数从 **tinyGex2.txt** (请见练习 4.1.2) 得到的图的一个着色方案并按照本节示意图的样式给出详细的轨迹。在最坏情况下, **TwoColor** 构造函数的运行时间的增长数量级是多少?
- 4.1.21 用 **SymbolGraph** 和 **movie.txt** 找到今年获得奥斯卡奖提名的演员的 Kevin Bacon 数。
- 4.1.22 编写一段程序 **BaconHistogram**, 打印一幅 Kevin Bacon 数的柱状图, 显示 **movies.txt** 中 Kevin Bacon 数为 0、1、2、3……的演员分别有多少。将值为无穷大的人归为一类(不与 Kevin Bacon 连通)。
- 4.1.23 计算由 **movies.txt** 得到的图的连通分量的数量和包含的顶点数小于 10 的连通分量的数量。计算最大的连通分量的离心率、直径、半径和中点。Kevin Bacon 在最大的连通分量之中吗?
- 4.1.24 修改 **DegreesOfSeparation**, 从命令行接受一个整型参数 y, 忽略上映年数超过 y 的电影。
- 4.1.25 编写一个类似于 **DegreesOfSeparation** 的 **SymbolGraph** 用例, 使用深度优先搜索代替广度优先搜索来查找两个演员之间的路径, 输出类似如右侧框注所示的数据格式。
- 4.1.26 使用 1.4 节中的内存使用模型评估用 **Graph** 表示一幅含有 V 个顶点和 E 条边的图所需的内存。
- 4.1.27 如果重命名一幅图中的顶点就能够使之变得和另一幅图完全相同, 这两幅图就是同构的。画出含有 2、3、4、5 个顶点的所有非同构的图。
- 4.1.28 修改 **Cycle**, 允许图含有自环和平行边。

558  
559

```
% java DegreesOfSeparationDFS movies.txt
Source: Bacon, Kevin
Query: Kidman, Nicole
Bacon, Kevin
Mystic River (2003)
O'Hara, Jenny
Matchstick Men (2003)
Grant, Beth
...
[123 movies]()
Law, Jude
Sky Captain... (2004)
Jolie, Angelina
Playing by Heart (1998)
Anderson, Gillian (I)
Cock and Bull Story, A (2005)
Henderson, Shirley (I)
24 Hour Party People (2002)
Eccleston, Christopher
Gone in Sixty Seconds (2000)
Balahoutis, Alexandra
Days of Thunder (1990)
Kidman, Nicole
```

560  
561

## 提高题

4.1.29 欧拉环和汉密尔顿环。考虑以下4组边定义的图：

```
0-1 0-2 0-3 1-3 1-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
0-1 0-2 0-3 1-3 0-3 2-5 5-6 3-6 4-7 4-8 5-8 5-9 6-7 6-9 8-8
0-1 1-2 1-3 0-3 0-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
4-1 7-9 6-2 7-3 5-0 0-2 0-8 1-6 3-9 6-3 2-8 1-5 9-8 4-5 4-7
```

哪几幅图含有欧拉环（恰好包含了所有的边且没有重复的环）？哪几幅图含有汉密尔顿环（恰好包含了所有的顶点且没有重复的环）？

- 4.1.30 图的枚举。含有  $V$  个顶点和  $E$  条边（不含平行边）的不同的无向图共有多少种？
- 4.1.31 检测平行边。设计一个线性时间的算法来统计图中的平行边的总数。
- 4.1.32 奇环。证明一幅图能够用两种颜色着色（二分图）当且仅当它不含有长度为奇数的环。
- 4.1.33 符号图。实现一个 `SymbolGraph`（不一定必须使用 `Graph`），只需要遍历一遍图的定义数据。由于需要查找符号表，实现中图的各种操作时耗可能会变为原来的  $\log V$  倍。
- 4.1.34 双向连通性。如果任意一对顶点都能由两条不同（没有重叠的边或顶点）的路径连通则图就是双向连通的。在一幅连通图中，如果一个顶点被删掉后图不再连通，该顶点就被称为关节点。证明没有关节点的图是双向连通的。提示：给定任意一对顶点  $s$  和  $t$  和一条连接两点的路径，由于路径上没有任何顶点为关节点，构造另一条不同的路径连接  $s$  和  $t$ 。
- 4.1.35 边的连通性。在一幅连通图中，如果一条边被删除后图会被分为两个独立的连通分量，这条边就被称为桥。没有桥的图称为边连通图。开发一种基于深度优先搜索算法的数据类型，判断一个图是否是边连通图。
- 4.1.36 欧拉图。为平面上的图设计并实现一份叫做 `EulideanGraph` 的 API，其中图所有顶点均有坐标。实现一个 `show()` 方法，用 `StdDraw` 将图绘出。
- 4.1.37 图像处理。在一幅图像中将所有相邻的、颜色相同的点相连就可以得到一幅图，为这种隐式定义的图实现填充（flood fill）操作。

562  
563

## 实验题

- 4.1.38 随机图。编写一个程序 `ErdosRenyiGraph`，从命令行接受整数  $V$  和  $E$ ，随机生成  $E$  对 0 到  $V-1$  之间的整数来构造一幅图。注意：生成器可能会产生自环和平行边。
- 4.1.39 随机简单图。编写一个程序 `RandomSimpleGraph`，从命令行接受整数  $V$  和  $E$ ，用均等的几率生成含有  $V$  个顶点和  $E$  条边的所有可能的简单图。
- 4.1.40 随机稀疏图。编写一个程序 `RandomSparseGraph`，根据精心选择的一组  $V$  和  $E$  的值生成随机的稀疏图，以便用它对由 Erdős-Renyi 模型得到的图进行有意义的经验性测试。
- 4.1.41 随机欧拉图。编写一个 `EulideanGraph` 的用例（请见练习 4.1.36）`RandomEulideanGraph`，用随机在平面上生成  $V$  个点的方式生成随机图，然后将每个点和在以该点为中心半径为  $d$  的圆内的其他点相连。注意：如果  $d$  大于阈值  $\sqrt{\lg V / \pi V}$ ，那么得到的图几乎必然是连通的，否则得到的图几乎必然是不连通的。
- 4.1.42 随机网格图。编写一个 `EulideanGraph` 的用例 `RandomGridGraph`，将  $\sqrt{V}$  乘  $\sqrt{V}$  的网格中的所有顶点和它们的相邻顶点相连（参考练习 1.5.18）。修改代码为图额外添加  $R$  条随机的边。对于较大的  $R$ ，缩小网格使得总边数保持在  $V$  个左右。添加一个选项，使得出现一条从顶点  $s$  到顶

点  $v$  的边的概率与  $s$  到  $t$  的欧拉距离成反比。

- 4.1.43 真实世界中的图。从网上找出一幅巨型加权图——可以是一张标记了距离的地图，或者是标明了费用的电话连接，或是航班价目表。编写一段程序 `RandomRealGraph`，从这些顶点构成的子图中随机选取  $V$  个顶点，然后再从这些顶点构成的子图中随机选取  $E$  条边来构造一幅图。
- 4.1.44 随机区间图。考虑数轴上的  $V$  个区间的集合。这样的一个集合定义了一幅区间图，图中的每个顶点都对应一个区间，而边则对应两个区间的交集（大小不限）。编写一段程序，随机生成大小均为  $d$  的  $V$  个区间，然后构造相应的区间图。提示：使用二分查找树。564
- 4.1.45 随机运输图。定义运输系统的一种方法是定义一个顶点链的集合，每条顶点链都表示一条连接了多个顶点的路径。例如，链 0-9-3-2 定义了边 0-9、9-3 和 3-2。编写一个 `EulideanGraph` 的用例 `RandomTransportation`，从一个输入文件中构造一幅图，文件的每行均为一条链，使用符号名。编辑一份合适的输入使得程序能够从中构造一幅和巴黎地铁系统相对应的图。  
测试所有的算法并研究所有图模型的所有参数是不现实的。请为下面的每一道题都编写一段程序来处理从输入得到的任意图。这段程序可以调用上面的任意生成器并对相应的图模型进行实验。可以根据上次实验的结果自己作出判断来选择不同实验。陈述结果以及由此得出的任何结论。
- 4.1.46 深度优先搜索中的路径长度。对于各种图的模型，运行实验并根据经验判断 `DepthFirstPaths` 在两个随机选定的顶点之间找到一条路径的概率并计算找到的路径的平均长度。
- 4.1.47 广度优先搜索中的路径长度。对于各种图的模型，运行实验并根据经验判断 `BreadthFirstPaths` 在两个随机选定的顶点之间找到一条路径的概率并计算找到的路径的平均长度。
- 4.1.48 连通分量。运行实验随机生成大量的图并画出柱状图，根据经验判断各种类型的随机图中连通分量的数量的分布情况。
- 4.1.49 双色问题。大多数的图都无法用两种颜色着色，深度优先搜索能够很快发现这一点。对于各种图模型，使用经验性的测试来研究 `TwoColor` 检查的边的数量。565

### **Robert Sedgewick**

斯坦福大学博士，导师为Donald E. Knuth，从1985年开始一直担任普林斯顿大学计算机科学系教授，曾任该系主任，也是Adobe Systems公司董事会成员，曾在Xerox PARC、国防分析研究所（Institute for Defense Analyses）和法国国家信息与自动化研究所（INRIA）从事研究工作。他的研究方向包括解析组合学、数据结构和算法的分析与设计、程序可视化等。

### **Kevin Wayne**

康奈尔大学博士，普林斯顿大学计算机科学系高级讲师，研究方向包括算法的设计、分析和实现，特别是图和离散优化。



# 算法 (第4版)

Algorithms Fourth Edition

本书全面讲述算法和数据结构的必备知识，具有以下几大特色。

◆ 算法领域的经典参考书

Sedgewick畅销著作的最新版，反映了经过几十年演化而成的算法核心知识体系

◆ 内容全面

全面论述排序、搜索、图处理和字符串处理的算法和数据结构，涵盖每位程序员应知应会的50种算法

◆ 全新修订的代码

全新的Java实现代码，采用模块化的编程风格，所有代码均可供读者使用

◆ 与实际应用相结合

在重要的科学、工程和商业应用环境下探讨算法，给出了算法的实际代码，而非同类著作常用的伪代码

◆ 富于智力趣味性

简明扼要的内容，用丰富的视觉元素展示的示例，精心设计的代码，详尽的历史和科学背景知识，各种难度的练习，这一切都将使读者手不释卷

◆ 科学的方法

用合适的数学模型精确地讨论算法性能，这些模型是在真实环境中得到验证的

◆ 与网络相结合

配套网站algs4.cs.princeton.edu提供了本书内容的摘要及相关的代码、测试数据、编程练习、教学课件等资源

PEARSON

[www.pearson.com](http://www.pearson.com)

图灵社区：[www.ituring.com.cn](http://www.ituring.com.cn)

新浪微博：@图灵教育 @图灵社区

反馈/投稿/推荐信箱：[contact@turingbook.com](mailto:contact@turingbook.com)

热线：(010)51095186转604

**分类建议** 计算机/计算机科学

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)

ISBN 978-7-115-29380-0



ISBN 978-7-115-29380-0

定价：99.00元