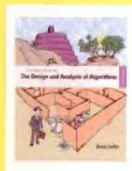


PEARSON

Introduction to The Design and Analysis of Algorithms Third Edition

# 算法设计与分析基础

(美) Anany Levitin 著 (第3版)  
潘彦 译



- ② 畅销十余年的国外经典教材，影响全球数十万名读者
- ② 条理清晰，逻辑严密，透过经典算法来揭示算法精髓
- ② 全面覆盖十大通用算法设计技术
- ② 600多道习题有难有易又有趣，涉及ACM竞赛题、算法谜题和面试题

PEARSON

清华大学出版社

---

本书仅提供部分阅读，如需完整版，请联系QQ: 461573687

提供各种书籍pdf下载，如有需要，请联系 QQ: 461573687

PDF制作说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ: 461573687, 或者 QQ: 2404062482。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！

**备用QQ:2404062482**



Introduction to The Design and Analysis of Algorithms Third Edition

# 算法设计与分析基础

(第3版)

(美) Anany Levitin 著  
潘彦译

清华大学出版社  
北京

## 内 容 简 介

作者基于丰富的教学经验,开发了一套全新的算法分类方法。该分类法站在通用问题求解策略的高度,对现有大多数算法准确分类,从而引领读者沿着一条清晰、一致、连贯的思路来探索算法设计与分析这一迷人领域。本书作为第3版,相对前版调整了多个章节的内容和顺序,同时增加了一些算法,并扩展了算法的应用,使得具体算法和通用算法设计技术的对应更加清晰有序;各章累计增加了70道习题,其中包括一些有趣的谜题和面试问题。

本书十分适合作为算法设计和分析的基础教材,也适合任何有兴趣探究算法奥秘的读者使用,只要读者具备数据结构和离散数学的知识即可。

Simplified Chinese edition copyright © 2015 by PEARSON EDUCATION ASIA LIMITED and TSINGHUA UNIVERSITY PRESS.

Original English language title: Introduction to the Design and Analysis of Algorithms, 3rd Edition by Anany Levitin, Copyright © 2012

EISBN: 9780132316811

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Pearson Education, Inc.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).

本书中文简体翻译版由 Pearson Education 授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行。

北京市版权局著作权合同登记号 图字: 01-2014-2598

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。侵权举报电话: 010-62782989 13701121933

### 图书在版编目(CIP)数据

算法设计与分析基础/(美)莱维汀(Levitin, A.)著;潘彦译. —3版. —北京:清华大学出版社, 2015

书名原文: Introduction to the Design and Analysis of Algorithms

ISBN 978-7-302-38634-6

I. ①算… II. ①莱… ②潘… III. ①算法设计 ②算法分析 IV. TP301.6

中国版本图书馆 CIP 数据核字(2014)第 279849 号

责任编辑:文开琪 汤涌涛

封面设计:杨玉兰

责任校对:周剑云

责任印制:宋 林

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, [c-service@tup.tsinghua.edu.cn](mailto:c-service@tup.tsinghua.edu.cn)

质 量 反 馈:010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印 刷 者:三河市君旺印务有限公司

装 订 者:三河市新茂装订有限公司

经 销:全国新华书店

开 本:185mm×260mm 印 张:27.5 字 数:657千字

版 次:2004年6月第1版 2015年2月第3版 印 次:2015年2月第1次印刷

印 数:1~3500

定 价:69.00元



# 译者序

十年前，本书第 1 版面世。

十年后，迎来了第 3 版。

十年不长。作者 Anany Levitin 仍然在维拉诺瓦大学从事算法基础教学，兢兢业业不断更新和完善着这本算法经典教材。清华大学出版社的诸位仍然辛勤耕耘在教材出版的第一线，在行业并不十分景气的情况下，恪守职业尊严，努力为大家奉献一部部优秀的教材和读物。正是由于这些作者、编者多年不变的持续付出，计算机教育事业才有了不断发展下去的动力。

十年也不短。十年前的读者想必已经从莘莘学子成为了企业骨干，很多已经成家立业，事业有成了吧？大家有没有在从事和算法有关的工作？算法学习给大家带来了什么有益的改变？多想听听大家的心声。作为译者本人来说，翻译第 1 版时刚刚三十岁，而现在已过不惑之年。当年接手本书的初衷仅仅是希望提供一本易懂的翻译教材，尽量减少读者阅读的障碍。但实际上，从这本书受益最大的可能还是译者本人。首先，翻译本书的过程提高了自身的综合能力。其次，有机会逐字逐句精读这样一本严谨的教材是一种很好的学术训练，为本人后来的博士生涯增益不少。最后，本人目前从事算法交易，尽管很少用到现成算法，但本书提供的算法专业训练还是使我获益良多。

茫茫历史长河中，一本书的好坏可能并不重要，但如果每个人都能专注做好自己的事情，对人对己就会产生非常有益的影响。捧起本书的读者们，我衷心希望大家认真做事，做正确的事。因为，下一个十年你不会后悔这样的付出。

我要感谢本书原著者，让我有机会和一本好书一起成长。我要感谢第 1、第 2 版的读者，他们通过互联网对本书做出了非常积极的评价，还有读者不吝指出书中的错误，和大家交流非常开心。我要感谢出版社的领导，继续给予我信任，并容忍我并不算快的进度。我还要感谢本书的编辑，她十年如一日，以一贯的严谨为本书提供了质量保证，尽管从未谋面，我想我们已经是老朋友了。我最后要感谢爱人李靓的支持，她理解翻译工作的意义，为我提供了很多实际的帮助。

从作者本版的修订风格来看，第 3 版不会是最后一版，希望我有幸再次为广大读者执起译笔。

祝大家学习顺利！

潘彦

phil\_pan@hotmail.com

# 前言

一个人在接受科技教育时能得到的最珍贵的收获是能够终身受用的通用智能工具。<sup>①</sup>

——乔治·福赛思

无论是计算科学还是计算实践，算法都在其中扮演着重要角色。因此，这门学科中出现了大量的教材。它们在介绍算法的时候，基本上都选择了以下两种方案中的一种。第一种方案是按照问题的类型对算法进行分类。这类教材安排了不同的章节分别讨论排序、查找、图等算法。这种做法的优点是，对于解决同一问题的不同算法，它能够立即比较这些算法的效率。其缺点在于，由于过于强调问题的类型，它忽略了对算法设计技术的讨论。

第二种方案围绕着算法设计技术来组织章节。在这种结构中，即使算法来自于不同的计算领域，如果它们采用了相同的设计技术，就会被编成一组。从各方(例如[BaY95])获得的信心使我相信，这种结构更适合于算法设计与分析的基础课程。强调算法设计技术有三个主要原因。第一，学生们在解决新问题时，可以运用这些技术设计出新的算法。从实用的角度看，这使得学习算法设计技术颇有价值。第二，学生们会试图按照算法的内在设计方法对已知的众多算法进行分类。计算机科学教育的一个主要目的，就是让学生们知道如何发掘不同应用领域的算法间的共性。毕竟，每门学科都会倾向于把它的重要主题归纳为几个甚至一个规则。第三，依我看来，算法设计技术作为问题求解的一般性策略，在解决计算机领域以外的问题时，也能发挥相当大的作用。

遗憾的是，无论是从理论还是从教学的角度，传统的算法设计技术分类法都存在一些严重的缺陷。其中最显著的缺陷就是无法对许多重要的算法进行分类。由于这种局限性，这些书的作者不得不在按照设计技术进行分类的同时，另外增加一些章节来讨论特殊的问题类型。但这种改变导致课程缺乏一致性，而且很可能会使学生感到迷惑。

## 算法设计技术的新分类法

传统算法设计技术分类法的缺陷令我感到失望，它激发我开发一套新的分类法([Lev99])，这套分类法就是本书的基础。以下是这套新分类法的几个主要优势。

- 新分类法比传统分类法更容易理解。它包含的某些设计策略，例如蛮力法、减治法、变治法、时空权衡和迭代改进，几乎从不曾被看作重要的设计范例。
- 新分类法很自然地覆盖了许多传统方法无法分类的经典算法(欧几里得算法、堆排序、查找树、散列法、拓扑排序、高斯消去法、霍纳法则等，不胜枚举)。所以，新分类法能够以一种连贯的、一致的方式表达这些经典算法的标准内容。
- 新分类法很自然地容纳了某些设计技术的重要变种(例如，它能涵盖减治法的3个

---

<sup>①</sup> 译注：出自 *What to do till the computer scientist comes*(1968)。乔治·福赛思(George Forsythe, 1917—1972)是一名数学家，他认为最重要的三大工具依次是自然语言、数学和计算机科学。

变种和变治法的3个变种)。

- 在分析算法效率时,新分类法与分析方法结合得更好(参见附录B)。

## 设计技术作为问题求解的一般性策略

在本书中,主要将设计技术应用于计算机科学中的经典问题(这里唯一的创新是引入了一些数值算法的内容,我们也是用同样的通用框架来表述这些算法的)。但把这些设计技术看作问题求解的一般性工具时,它们的应用就不仅限于传统的计算问题和数学问题了。有两个因素令这一点变得尤其重要。第一,越来越多的计算类应用超越了它们的传统领域,并且有足够的理由使人相信,这种趋势会愈演愈烈。第二,人们渐渐认识到,提高学生们的問題求解能力是高等教育的一个主要目标。为了满足这个目标,在计算机科学课程体系中安排一门算法设计和分析课程是非常合适的,因为它会告诉学生如何应用一些特定的策略来解决问题。

虽然我并不建议将算法设计和分析课程变成一门教授一般性问题求解方法的课程,但我深信,我们不应错过算法设计和分析课程提供的这样一个独一无二的机会。为了这个目标,本书包含了一些和谜题相关的应用。虽然利用谜题来教授算法课程绝不是我的创新,但本书打算通过引进一些全新的谜题来系统地实现这个思路。

## 如何使用本书

我的目标是写一本既不泛泛而谈,又可供学生们独立阅读的教材。为了实现这个目标,本书做了如下努力。

- 根据乔治·福赛思的观点(参见前面的引文),我试图着重强调隐藏在算法设计和分析背后的主要思想。在选择特定的算法来阐述这些思想的时候,我并不倾向于涉及大量的算法,而是选择那些最能揭示其内在设计技术或分析方法的算法。幸运的是,大多数经典算法满足这个要求。
- 第2章主要分析算法的效率,该章将分析非递归算法的方法和分析递归算法的典型方法区别开来。这一章还花了一些篇幅介绍算法经验分析和算法可视化。
- 书中系统地穿插着一些面向读者的提问。其中有些问题是经过精心设计的,而且答案紧随其后,目的是引起读者的注意或引发疑问。其余问题的用意是防止读者走马观花,不能充分理解本书的内容。
- 每一章结束时都会对本章最重要的概念和结论做一个总结。
- 本书包含600多道习题。有些习题是为了给大家练习,另外一些则是为了指出书中正文部分所涉及内容的重要意义,或是为了介绍一些书中没有涉及的算法。有一些习题利用了因特网上的资源。较难的习题数量不多,会在教师用书中用一种特殊的记号标注出来(因为有些学生可能没有勇气做那些有难度标注的习题,所以本书没有对习题标注难度)。谜题类的习题用一种特殊的图标做标注。
- 本书所有的习题都附有提示。除了编程练习,习题的详细解法都能够在教师资源中找到。请发送邮件到 [coo@netease.com](mailto:coo@netease.com), 申请教师相关资源(也可联系培生公司的当地销



售代表, 或者访问 [www.pearsonhighered.com/irc](http://www.pearsonhighered.com/irc))。本书的任何读者都可以在 CS 支持网站 <http://cssupport.pearsoncmg.com> 上找到 PowerPoint 格式的幻灯片文件。如果对算法有兴趣, 欢迎加入 QQ 群“算法学习交流”, 群号: 425283001。

## 第 3 版的变化

第 3 版有若干变化。其中最重要的变化是介绍减治法和分治法的先后顺序。第 3 版会先介绍减治法, 后介绍分治法, 这样做有以下几个优点。

- 较之分治法, 减治法更简单。
- 在求解问题方面, 减治法应用更广。
- 这样的编排顺序便于先介绍插入排序, 后介绍合并排序和快速排序。
- 数组划分的概念通过选择性问题引入, 这次利用 Lomuto 算法的单向扫描来实现, 而将 Hoare 划分方法的双向扫描留至后文与快速排序一并介绍。
- 折半查找归入介绍减常量算法的章节。

另一个重要变化是重新编排第 8 章关于动态规划的内容, 具体如下所述。

- 导述部分的内容是全新的。在前两版中用计算二项式系数的例子来引入动态规划这一重要技术, 但在第 3 版中会介绍 3 个基础性示例, 这样介绍的效果更好。
- 8.1 节的习题是全新的, 包括一些在前两版中没有涉及的流行的应用。
- 第 8 章其他小节的顺序也做了调整, 以便达到由浅入深、循序渐进的效果。

此外, 还有其他一些变化。增加了不少与本书所述算法相关的应用。遍历图算法不再随减治法介绍, 而是纳入蛮力算法和穷举查找的范畴, 我认为这样更合理。在介绍生成组合对象的算法时, 新增了格雷码算法。对求解最近对问题的分治法有更深入地探讨。改进的内容包括算法可视化和求解旅行商问题的近似算法, 当然参考文献也有相应的更新。

第 3 版一共新增约 70 道习题, 其中涉及算法谜题和面试问题。

## 先修课程

本书假定读者已经学习了离散数学的标准课程和一门基础性的编程课程。有了这样的知识背景, 读者应该能够掌握本书的内容而不会遇到太大的困难。尽管如此, 1.4 节、附录 A 和附录 B 仍然对基本的数据结构以及必须用到的求和公式与递推关系分别进行复习和回顾。只有 3 个小节(2.2 节、11.4 节和 12.4 节)会用到一些简单的微积分知识, 如果读者缺少必要的微积分知识, 完全可以跳过这 3 个涉及微积分的小节, 这并不妨碍对本书其余部分的理解。

## 课程进度安排

如果打算开设一门围绕算法设计技术来讲解算法设计和分析理论的基础课程, 可以采用本书作为教材。但要想在一个学期内完成该课程, 本书涵盖的内容可能过于丰富了。大体上来说, 跳过第 3~12 章的部分内容不会影响读者对后面部分的理解。本书的任何一个部分都可以安排学生自学。尤其是 2.6 节和 2.7 节, 它们分别介绍了经验分析和算法可视化,

这两小节的内容可以结合课后练习<sup>①</sup>布置给学生。

下面给出了针对一个学期课程的教学计划，这是按照 40 课时的集中教学来设计的。

课 次	主 题	小 节
1	课程简介	1.1~1.3
2, 3	分析框架：常用符号 $O$ 、 $\Theta$ 和 $\Omega$	2.1, 2.2
4	非递归算法的数学分析	2.3
5, 6	递归算法的数学分析	2.4, 2.5(+附录 B)
7	蛮力算法	3.1, 3.2(+3.3)
8	穷举查找	3.4
9	深度优先查找和广度优先查找	3.5
10~11	减一算法：插入排序、拓扑排序	4.1, 4.2
12	折半查找和其他减常量算法	4.4
13	减变量算法	4.5
14~15	分治法：合并排序、快速排序	5.1~5.2
16	其他分治法示例	5.3、5.4 或 5.5
16	减变量算法	5.6
17~19	实例化简：预排序、高斯消去法、平衡查找树	6.1~6.3
20	改变表现：堆和堆排序或者霍纳法则和二进制幂	6.4 或 6.5
21	问题化简	6.6
22~24	时空权衡：串匹配、散列法、B 树	7.2~7.4
25~27	动态规划算法	8.1~8.4(选 3 节)
28~30	贪婪算法：Prim 算法、Kruskal 算法、Dijkstra 算法、哈夫曼算法	9.1~9.4
31~33	迭代改进算法	10.1~10.4(选 3 节)
34	下界的参数	11.1
35	决策树	11.2
36	$P$ 、 $NP$ 和 $NP$ 完全问题	11.3
37	数值算法	11.4(+12.4)
38	回溯法	12.1
39	分支界限法	12.2
40	$NP$ 困难问题的近似算法	12.3

<sup>①</sup> 译注：“练习”的原文为 project，一般应该翻译成“项目”，但国外一般将布置在课后完成的、较大型的、要求实际演练的习题称为 project，国内没有相应的称呼，所以姑且译为“练习”。

## 致谢

我要向本书的评审表达衷心的感谢，还要感谢本书前两版的许多读者，他们提供了许多宝贵的意见和建议，帮助本书得以改进和完善。本书第3版尤其得益于下列人士的评审，包括 Andrew Harrington(芝加哥洛约拉大学)、David Levine(圣文德大学)、Stefano Lombardi(加州大学河滨分校)、Daniel McKee(宾州曼斯菲尔德大学)、Susan Brilliant(弗吉尼亚州立联邦大学)、David Akers(菩及海湾大学)以及两名匿名评审。

我要感谢培生出版社所有为本书付出不懈努力的工作人员和相关人士。尤其要感谢本书编辑 Matt Goldstein、编务助理 Chelsea Bell、市场经理 Yez Alayan 和产品总监 Kayla Smith-Tarbox。我还要感谢 Richard Camp 为本书审稿，Windfall Software 的 Paul Anagnostopoulos 和 Jacqui Scarlott 为本书排版并提供项目管理支持，以及 MaryEllen Oliver 为本书进行校对。

最后，我要感谢两位家人。另一半整天都在写书比自己本人写书更让人崩溃，我的妻子 Maria 已容忍我多年并任劳任怨地帮助我，本书 400 多幅插图以及教师手册都是凭她一己之力完成的。女儿 Miriam 是我多年的英语老师，她不但阅读了本书大量篇幅，还帮我为每章找到了合适的名人名言。

Anany Levitin  
anany.levitin@villanova.edu



# 目 录

第 1 章 绪论.....	1	2.1.1 输入规模的度量 .....	33
1.1 什么是算法.....	2	2.1.2 运行时间的度量单位 .....	34
习题 1.1.....	6	2.1.3 增长次数 .....	35
1.2 算法问题求解基础.....	7	2.1.4 算法的最优、最差和 平均效率 .....	36
1.2.1 理解问题.....	8	2.1.5 分析框架概要 .....	38
1.2.2 了解计算设备的性能.....	8	习题 2.1 .....	39
1.2.3 在精确解法和近似解法 之间做出选择.....	9	2.2 渐近符号和基本效率类型.....	40
1.2.4 算法的设计技术.....	9	2.2.1 非正式的介绍 .....	40
1.2.5 确定适当的数据结构.....	9	2.2.2 符号 $O$ .....	41
1.2.6 算法的描述.....	10	2.2.3 符号 $\Omega$ .....	42
1.2.7 算法的正确性证明.....	10	2.2.4 符号 $\Theta$ .....	42
1.2.8 算法的分析.....	11	2.2.5 渐近符号的有用特性 .....	43
1.2.9 为算法写代码.....	12	2.2.6 利用极限比较增长次数 .....	44
习题 1.2.....	13	2.2.7 基本的效率类型 .....	45
1.3 重要的问题类型.....	14	习题 2.2 .....	46
1.3.1 排序.....	15	2.3 非递归算法的数学分析.....	48
1.3.2 查找.....	16	习题 2.3 .....	52
1.3.3 字符串处理.....	16	2.4 递归算法的数学分析.....	54
1.3.4 图问题.....	16	习题 2.4 .....	59
1.3.5 组合问题.....	17	2.5 例题：计算第 $n$ 个斐波那契数.....	62
1.3.6 几何问题.....	17	习题 2.5 .....	65
1.3.7 数值问题.....	18	2.6 算法的经验分析.....	66
习题 1.3.....	18	习题 2.6 .....	69
1.4 基本数据结构.....	20	2.7 算法可视法.....	70
1.4.1 线性数据结构.....	20	小结.....	73
1.4.2 图.....	22	第 3 章 蛮力法 .....	75
1.4.3 树.....	25	3.1 选择排序和冒泡排序.....	76
1.4.4 集合与字典.....	28	3.1.1 选择排序 .....	76
习题 1.4.....	29	3.1.2 冒泡排序 .....	77
小结.....	30	习题 3.1 .....	78
第 2 章 算法效率分析基础.....	32	3.2 顺序查找和蛮力字符串匹配.....	80
2.1 分析框架.....	33	3.2.1 顺序查找 .....	80

3.2.2 蛮力字符串匹配.....	81	第5章 分治法 .....	131
习题 3.2.....	82	5.1 合并排序.....	133
3.3 最近对和凸包问题的蛮力算法.....	83	习题 5.1 .....	135
3.3.1 最近对问题.....	83	5.2 快速排序.....	136
3.3.2 凸包问题.....	84	习题 5.2 .....	140
习题 3.3.....	87	5.3 二叉树遍历及其相关特性.....	141
3.4 穷举查找.....	89	习题 5.3 .....	143
3.4.1 旅行商问题.....	89	5.4 大整数乘法和 Strassen 矩阵乘法 .....	144
3.4.2 背包问题.....	90	5.4.1 大整数乘法 .....	145
3.4.3 分配问题.....	91	5.4.2 Strassen 矩阵乘法 .....	146
习题 3.4.....	93	习题 5.4 .....	148
3.5 深度优先查找和广度优先查找.....	94	5.5 用分治法解最近对问题和 凸包问题.....	149
3.5.1 深度优先查找.....	94	5.5.1 最近对问题 .....	149
3.5.2 广度优先查找.....	96	5.5.2 凸包问题 .....	151
习题 3.5.....	98	习题 5.5 .....	153
小结.....	100	小结.....	154
第4章 减治法 .....	101	第6章 变治法 .....	155
4.1 插入排序.....	103	6.1 预排序.....	156
习题 4.1.....	105	习题 6.1 .....	158
4.2 拓扑排序.....	106	6.2 高斯消去法.....	160
习题 4.2.....	109	6.2.1 LU 分解 .....	164
4.3 生成组合对象的算法.....	111	6.2.2 计算矩阵的逆 .....	165
4.3.1 生成排列.....	111	6.2.3 计算矩阵的行列式 .....	166
4.3.2 生成子集.....	113	习题 6.2 .....	167
习题 4.3.....	114	6.3 平衡查找树.....	168
4.4 减常因子算法.....	115	6.3.1 AVL 树 .....	169
4.4.1 折半查找.....	116	6.3.2 2-3 树 .....	173
4.4.2 假币问题.....	117	习题 6.3 .....	174
4.4.3 俄式乘法.....	118	6.4 堆和堆排序.....	175
4.4.4 约瑟夫斯问题.....	119	6.4.1 堆的概念 .....	176
习题 4.4.....	120	6.4.2 堆排序 .....	180
4.5 减可变规模算法.....	122	习题 6.4 .....	181
4.5.1 计算中值和选择问题.....	122	6.5 霍纳法则和二进制幂.....	182
4.5.2 插值查找.....	125	6.5.1 霍纳法则 .....	182
4.5.3 二叉查找树的查找和插入.....	126	6.5.2 二进制幂 .....	184
4.5.4 拈游戏.....	127	习题 6.5 .....	186
习题 4.5.....	128	6.6 问题化简.....	187
小结.....	129		

6.6.1 求最小公倍数.....	188	第 9 章 贪婪技术.....	243
6.6.2 计算图中的路径数量.....	189	9.1 Prim 算法.....	245
6.6.3 优化问题的化简.....	189	习题 9.1 .....	249
6.6.4 线性规划.....	190	9.2 Kruskal 算法.....	250
6.6.5 简化为图问题.....	192	习题 9.2 .....	255
习题 6.6.....	193	9.3 Dijkstra 算法.....	256
小结.....	194	习题 9.3 .....	259
第 7 章 时空权衡.....	196	9.4 哈夫曼树及编码.....	260
7.1 计数排序.....	197	习题 9.4 .....	264
习题 7.1.....	199	小结.....	265
7.2 字符串匹配中的输入增强技术.....	200	第 10 章 迭代改进.....	266
7.2.1 Horspool 算法.....	201	10.1 单纯形法.....	267
7.2.2 Boyer-Moore 算法.....	204	10.1.1 线性规划的几何解释 .....	267
习题 7.2.....	207	10.1.2 单纯形法概述 .....	270
7.3 散列法.....	209	10.1.3 单纯形法其他要点 .....	275
7.3.1 开散列(分离链).....	210	习题 10.1 .....	276
7.3.2 闭散列(开式寻址).....	211	10.2 最大流量问题.....	278
习题 7.3.....	213	习题 10.2 .....	285
7.4 B 树.....	214	10.3 二分图的最大匹配.....	286
习题 7.4.....	217	习题 10.3 .....	291
小结.....	218	10.4 稳定婚姻问题.....	292
第 8 章 动态规划.....	219	习题 10.4 .....	295
8.1 三个基本例子.....	220	小结.....	296
习题 8.1.....	224	第 11 章 算法能力的极限.....	297
8.2 背包问题和记忆功能.....	226	11.1 如何求下界.....	298
8.2.1 背包问题.....	226	11.1.1 平凡下界 .....	298
8.2.2 记忆化.....	227	11.1.2 信息论下界 .....	299
习题 8.2.....	229	11.1.3 敌手下界 .....	299
8.3 最优二叉查找树.....	230	11.1.4 问题化简 .....	300
习题 8.3.....	234	习题 11.1 .....	302
8.4 Warshall 算法和 Floyd 算法.....	235	11.2 决策树.....	302
8.4.1 Warshall 算法.....	235	11.2.1 排序的决策树 .....	303
8.4.2 计算完全最短路径的 Floyd 算法 .....	238	11.2.2 查找有序数组的决策树 .....	305
习题 8.4.....	241	习题 11.2 .....	306
小结.....	242	11.3 $P$ 、 $NP$ 和 $NP$ 完全问题.....	308
		11.3.1 $P$ 和 $NP$ 问题.....	308



11.3.2 $NP$ 完全问题 .....	311	12.3 $NP$ 困难问题的近似算法 .....	339
习题 11.3.....	314	12.3.1 旅行商问题的近似算法 .....	340
11.4 数值算法的挑战.....	316	12.3.2 背包问题的近似算法 .....	349
习题 11.4.....	322	习题 12.3 .....	352
小结.....	323	12.4 解非线性方程的算法.....	353
<b>第 12 章 超越算法能力的极限 .....</b>	<b>325</b>	12.4.1 平分法 .....	355
12.1 回溯法.....	325	12.4.2 试位法 .....	357
12.1.1 $n$ 皇后问题.....	326	12.4.3 牛顿法 .....	358
12.1.2 哈密顿回路问题.....	328	习题 12.4 .....	360
12.1.3 子集和问题.....	328	小结.....	361
12.1.4 一般性说明.....	329	<b>跋 .....</b>	<b>363</b>
习题 12.1.....	331	<b>附录 A 算法分析的实用公式 .....</b>	<b>366</b>
12.2 分支界限法.....	332	<b>附录 B 递推关系简明指南 .....</b>	<b>369</b>
12.2.1 分配问题.....	332	<b>习题提示 .....</b>	<b>380</b>
12.2.2 背包问题.....	335	<b>参考文献 .....</b>	<b>414</b>
12.2.3 旅行商问题.....	336		
习题 12.2.....	338		

# 第1章 绪 论

有两种思想，就像摆放在天鹅绒上的宝石那样熠熠生辉，一个是微积分，另一个就是算法。微积分以及在微积分基础上建立起来的数学分析体系造就了现代科学，而算法则造就了现代世界。<sup>①</sup>

——大卫·柏林斯基

为什么要学习算法？如果你想成为一名计算机专业人士，无论从理论还是从实践的角度，学习算法都是必需的。从实践的角度来看，我们必须了解计算领域中不同问题的一系列标准算法。此外，我们还要具备设计新算法和分析其效率的能力。从理论的角度来看，对算法的研究(有时称为“**算法学**”，英文为 *algorithmics*)已经被公认为是计算机科学的基石。大卫·哈雷尔(David Harel)写了一本非常好的书，他直截了当地将其命名为《**算法学——计算精髓**》(*Algorithmics: the Spirit of Computing*)。书中是这样阐述这个问题的：

算法不只是计算机科学的一个分支。它是计算机科学的核心。而且，可以毫不夸张地说，它与绝大多数科学、商业和技术都密切相关。([Har92], p. 6)

即使不是计算机相关专业的学生，学习算法的理由也是非常充分的。坦率地说，没有算法，就没有计算机程序。而且，随着计算机日益渗透到我们日常工作和生活的方方面面，需要学习算法的人也越来越多。

学习算法的另一个理由是可以用它来培养人们的分析能力。毕竟，算法可以看作解决问题的一类特殊方法——它虽然不是问题的答案，但它是经过准确定义以获得答案的过程。因此，无论是否涉及计算机，特定的算法设计技术都能看作问题求解的有效策略。当然，算法思想天生固有的精确性限制了它能够解决的问题种类。例如，我们找不到一种使人幸福快乐的算法，也找不到一种使人功成名就的算法。但另一方面，从教育角度来看，这种必要的精确性却是很重要的。唐纳德·克努特(Donald Knuth)，算法学历史上最卓越的计算机科学家之一，是这样论述这个问题的：

受过良好训练的计算机科学家知道怎样处理算法：如何构造算法、操作算法、理解算法以及分析算法。这些知识远不止为了写出良好的计算机程序而准备的。算法是一种一般性的智能工具，它必定有助于我们对其他学科的理解，不管是化学、语言学或音乐，还是其他学科。为什么算法会有这种作用呢？我们可以这样理解：人们常说，一个人只有把知识教给别人，才能真正掌握它。实际上，一个人只有把知识教给“计算机”，才能“真正”掌握它，也就是说，将知识表述为一种算法……比起简单按照常规去理解事物，尝试用算法将其形式化能使我们的理解更加深刻。([Knu96], p. 9)

---

<sup>①</sup> 译注：引自《算法的诞生》(*The Advent of the Algorithm*, 2000)，作者是大卫·柏林斯基(David Berlinski)，他的另一部代表作是《微积分之旅》。

我们从 1.1 节开始涉及算法的概念。作为例子,我们将针对同一问题(求最大公约数)使用三种不同的算法。这样做有几个理由。首先,这是大家在中学时代就非常熟悉的问题。其次,它揭示了一种重要的观点:同样的问题往往能用多种算法来解决。这三种算法之所以具有典型意义,是因为它们的解题思路不同,复杂程度不同,解题效率也各不相同。再次,我希望将其中一种算法作为本书最先介绍的算法,这不仅因为它历史悠久(早在两千多年前它就出现在了欧几里得的著作中),还因为它不朽的力量和重要性。最后,对这三种算法的研究,能使我们从总体上观察一种算法通常具有的若干重要特性。

1.2 节讲述算法解题的分析和计算问题。我们将讨论有关算法设计和分析的一些重要内容。内容涉及算法解题的不同方面,包括问题的分析、如何正确表述算法以及算法的效率分析。这一节不能传授秘方为任意问题设计算法。这是一个公认的事实,即这种秘方是不存在的。但当大家日后进行自己的算法设计和分析工作时,肯定会用到 1.2 节的内容。

1.3 节专门讨论几种重要的问题类型。经验证明,无论对于算法教学还是对于算法应用,这些问题类型都是极其重要的。实际上,有些教材(例如[Sed88])就是围绕这些问题类型来组织内容的。尽管包括我在内的许多人都认为围绕算法设计技术来组织内容更有优势,但无论如何,了解这些重要的问题类型都是十分必要的。一是因为这些类型是实际应用中最常见的,二是由于本书从头到尾都会用它们来演示一些特殊的算法设计技术。

1.4 节回顾基本的数据结构。安排这一节的目的与其说是为了讨论这方面的内容,还不如说是希望把它作为读者的参考材料。如果需要了解更详细的内容,可以参考很多关于该主题的优秀书籍,其中大多数都是和某一种编程语言相关的。

## 1.1 什么是算法

虽然对于这个概念没有一个大家公认的定义,但我们对它的含义还是有基本共识的:

**算法(algorithm)**是一系列解决问题的明确指令,也就是说,对于符合一定规范的输入,能够在有限时间内获得要求的输出。

这个定义可以用一幅简单的图(图 1.1)来说明。

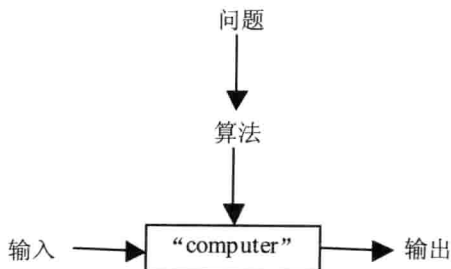


图 1.1 算法的概念

定义中使用了“指令”这个词,这意味着有人或物能够理解和执行所给出的命令,我

们将这种人或物称为 computer。请记住，在电子计算机发明以前，computer 是指那些从事数学计算的人。现在，computer 当然是特指那些做每件事情都越发不可或缺的、无所不在的电子设备。但要注意的是，虽然绝大多数算法最终要靠计算机来执行，但算法概念本身并不依赖于这样的假设。

为了阐明算法的概念，本节将以三种方法为例来解决同一个问题，即计算两个整数的最大公约数。这些例子会帮助我们阐明以下要点。

- 算法的每一个步骤都必须没有歧义，不能有半点儿含糊。
- 必须认真确定算法所处理的输入的值域。
- 同一算法可以用几种不同的形式来描述。
- 同一问题，可能存在几种不同的算法。
- 针对同一问题的算法可能基于完全不同的解题思路，而且解题速度也会有显著不同。

还记得最大公约数的定义吗？两个不全为 0 的非负整数  $m$  和  $n$  的最大公约数记为  $\gcd(m, n)$ ，代表能够整除(即余数为 0) $m$  和  $n$  的最大正整数。古希腊数学家、亚历山大港的欧几里得(公元前 3 世纪)所著的《几何原本》，以系统论述几何学而著称，在其中的一卷里，他简要描述了一个最大公约数算法。用现代数学的术语来表述，欧几里得算法(Euclid's algorithm)采用的方法是重复应用下列等式，直到  $m \bmod n$  等于 0。

$$\gcd(m, n) = \gcd(n, m \bmod n) \quad (m \bmod n \text{ 表示 } m \text{ 除以 } n \text{ 之后的余数})$$

因为  $\gcd(m, 0) = m$ (为什么?)， $m$  最后的取值也就是  $m$  和  $n$  的初值的最大公约数。

举例来说， $\gcd(60, 24)$ 可以这样计算：

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12$$

如果你对这个算法还没有足够的认识，可以做本节习题第 6 题，试着求一些较大数的最大公约数。

下面是该算法的一个更加结构化的描述：

用于计算  $\gcd(m, n)$ 的欧几里得算法

**第一步：**如果  $n = 0$ ，返回  $m$  的值作为结果，同时过程结束；否则，进入第二步。

**第二步：** $m$  除以  $n$ ，将余数赋给  $r$ 。

**第三步：**将  $n$  的值赋给  $m$ ，将  $r$  的值赋给  $n$ ，返回第一步。

我们也可以使用伪代码来描述这个算法：

```
算法 Euclid( $m, n$ )
//使用欧几里得算法计算  $\gcd(m, n)$ 
//输入：两个不全为 0 的非负整数  $m, n$ 
//输出： $m, n$  的最大公约数
while  $n \neq 0$  do
     $r \leftarrow m \bmod n$ 
     $m \leftarrow n$ 
     $n \leftarrow r$ 
return  $m$ 
```

我们怎么知道欧几里得算法最终一定会结束呢?通过观察,我们发现,每经过一次循环,参加运算的两个算子中的后一个都会变得更小,而且绝对不会变成负数。确实,下一次循环时, $n$ 的新值是 $m \bmod n$ ,这个值总是比 $n$ 小。所以,第二个算子的值最终会变成0,此时,这个算法也就结束了。

就像其他许多问题一样,最大公约数问题也有多种算法。让我们看看解这个问题的另外两种方法。第一个方法只基于最大公约数的定义: $m$ 和 $n$ 的最大公约数就是能够同时整除它们的最大正整数。显然,这样一个公约数不会大于两数中的较小者,因此,我们先有: $t = \min\{m, n\}$ 。我们现在可以开始检查 $t$ 是否能够整除 $m$ 和 $n$ :如果能, $t$ 就是最大公约数;如果不能,我们就将 $t$ 减1,然后继续尝试(我们如何确定该算法最终一定会结束呢? )。例如,对于60和24这两个数来说,该算法会先尝试24,然后是23,这样一直尝试到12,算法就结束了。

#### 用于计算 $\gcd(m, n)$ 的连续整数检测算法

**第一步:** 将  $\min\{m, n\}$  的值赋给  $t$ 。

**第二步:**  $m$  除以  $t$ 。如果余数为0,进入第三步;否则,进入第四步。

**第三步:**  $n$  除以  $t$ 。如果余数为0,返回  $t$  的值作为结果;否则,进入第四步。

**第四步:** 把  $t$  的值减1。返回第二步。

注意,和欧几里得算法不同,按照这个算法的当前形式,当它的一个输入为0时,计算出来的结果是错误的。这个例子说明了为什么必须认真、清晰地规定算法输入的值域。

求最大公约数的第三种过程,我们应该在中学时代就很熟悉了。

#### 中学时计算 $\gcd(m, n)$ 的过程

**第一步:** 找到  $m$  的所有质因数。

**第二步:** 找到  $n$  的所有质因数。

**第三步:** 从第一步和第二步求得的质因数分解式找出所有的公因数(如果  $p$  是一个公因数,而且在  $m$  和  $n$  的质因数分解式分别出现过  $p_m$  和  $p_n$  次,那么应该将  $p$  重复  $\min\{p_m, p_n\}$  次)。

**第四步:** 将第三步中找到的质因数相乘,其结果作为给定数字的最大公约数。

这样,对于60和24这两个数,我们得到:

$$60 = 2 \times 2 \times 3 \times 5$$

$$24 = 2 \times 2 \times 2 \times 3$$

$$\gcd(60, 24) = 2 \times 2 \times 3 = 12$$

虽然学习这个方法的那段中学时光是令人怀念的,但我们仍然注意到,第三个过程比欧几里得算法要复杂得多,也慢得多(下一章中,我们将会讨论对算法运行时间进行求解和比较的方法)。撇开低劣的性能不谈,以这种形式表述的中学求解过程还不能称为一个真正意义上的算法。为什么?因为其中求质因数的步骤并没有明确定义:该步骤要求得到一个质因数的列表,但我们十分怀疑中学里是否曾教过如何求这样一个列表。必须承认,这并不是鸡蛋里挑骨头。除非解决了这个问题,否则我们不能下结论说,能够写一个实现这个



过程的程序。顺便说一句，第三步也没有定义清楚。当然，它的不明确性要比求质因数的步骤更容易纠正一些。想想我们是如何求两个有序列表的公共元素的。

所以，我们要介绍一个简单的算法，用来产生一个不大于给定整数  $n$  的连续质数序列。它很可能是古希腊人发明的，称为“埃拉托色尼筛选法”<sup>①</sup>(sieve of Eratosthenes)。该算法一开始初始化一个  $2 \sim n$  的连续整数序列，作为候选质数。然后，在算法的第一个循环中，它将类似 4 和 6 这样的 2 的倍数从序列中消去。然后，它指向列表中的下一个数字 3，又将其倍数消去(我们这里的做法过于直接，增加了不必要的开销。因为有一些数字，拿 6 来说吧，被消去了不止一次)。不必处理数字 4，因为 4 本身和它的倍数都是 2 的倍数，它们已经在前面的步骤中被消去了。第三步处理序列中剩下的下一个元素 5。该算法以这个方式不断做下去，直到序列中已经没有可消的元素为止。序列中剩下的整数就是我们要求的质数。

作为一个例子，我们尝试用这个算法找出  $n$  不大于 25 的质数序列。

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	3		5		7				11		13				17		19				23		25
2	3		5		7				11		13				17		19					23	

对于这个例子来说，更多步骤已经多余了，因为它们只会消去在算法的前面循环中已经消去的数字。序列中剩下的数字就是小于等于 25 的连续质数。

其倍数仍未消去的最大数  $p$  应该满足什么条件呢<sup>②</sup>？在回答这个问题之前，我们先要注意到：如果当前步骤中，我们正在消去  $p$  的倍数，那么第一个值得考虑的倍数是  $p \times p$ ，因为其他更小的倍数  $2p, \dots, (p-1)p$  已经在先前的步骤中从序列里消去了。了解这个事实可以帮助我们避免多次消去相同的数字。显然， $p \times p$  不会大于  $n$ ， $p$  也不会大于  $\sqrt{n}$  向下取整的值(记作  $\lfloor \sqrt{n} \rfloor$ <sup>③</sup>，称为“向下取整函数”)。在下面这段伪代码中，我们假设有一个函数可以计算  $\lfloor \sqrt{n} \rfloor$ ，当然，我们也能以不等式  $p \times p \leq n$  作为判断循环是否继续的条件。

#### 算法 Sieve( $n$ )

```

//实现“埃拉托色尼筛选法”
//输入：一个正整数  $n > 1$ 
//输出：包含所有小于等于  $n$  的质数的数组  $L$ 
for  $p \leftarrow 2$  to  $n$  do  $A[p] \leftarrow p$ 
for  $p \leftarrow 2$  to  $\lfloor \sqrt{n} \rfloor$  do //参见伪代码前的说明
    if  $A[p] \neq 0$  //  $p$  没有被前面的步骤消去
         $j \leftarrow p * p$ 
        while  $j \leq n$  do
             $A[j] \leftarrow 0$  //将该元素标记为已经消去
             $j \leftarrow j + p$ 
//将  $A$  中剩余的元素复制到质数数组  $L$  中

```

① 译注：埃拉托色尼出生于昔勒尼(在今利比亚)，此算法诞生于约公元前 200 年。

② 译注： $p$  以后，消去过程就可以停止了。

③ 译注：舍去小数部分后的整数值。

```
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

这样就能够将“埃拉托色尼筛选法”应用在中学时的求解过程中了,我们得到了一个计算两个正整数的最大公约数的正规算法。注意,还必须关注其中一个输入参数为 1 或者两个都为 1 的情况:因为严格来讲,数学家并不认为 1 是一个质数,所以这个方法是无法处理这种输入的。

在结束本节之前,还需要做一些说明。虽然我们这里举的例子有一些数学味道,但当今所使用的大多数算法(即使是那些已经应用于计算机程序的算法)都不涉及数学问题。大家可以看到,无论是工作中还是生活中,算法每天都在帮助我们处理各种事务。算法在当今社会是无所不在的,它是信息时代的魔术引擎,希望这个事实能够使大家下定决心,深入学习算法课程。

## 习题 1.1

1. 研究一下 al-Khorezmi(或者称为 al-Khwarizmi,译名为阿尔·花刺子模),“算法”(algorithm)一词起源于这个名字。研究过程中我们还会发现,“算法”一词的起源和“代数”(algebra)一词的起源是相同的。
2. 如果告诉你,设立美国专利体系的基本目的是促进“有用的技术”,那么你认为算法在这个国家能够申请到专利吗?算法是否应该允许申请专利呢?
3.
  - a. 按照算法要求的精确性写出你从学校到家里的驾驶指南。
  - b. 按照算法要求的精确性写出你最喜欢的菜的烹饪方法。
4. 设计一个计算  $\lfloor \sqrt{n} \rfloor$  的算法,  $n$  是任意正整数。除了赋值和比较运算,该算法只能用到基本的四则运算操作。
5. 设计一个算法,在已经排序的两个列表中,找出所有相同的元素。例如,列表 2, 5, 5, 5 和 2, 2, 3, 5, 5, 7, 应该输出 2, 5, 5。如果给定的两个列表的长度分别为  $m$  和  $n$ ,你设计的算法的最大比较次数是多少?
6.
  - a. 用欧几里得算法求  $\text{gcd}(31415, 14142)$ 。
  - b. 用欧几里得算法求  $\text{gcd}(31415, 14142)$ , 速度是检查  $\min\{m, n\}$  和  $\text{gcd}(m, n)$  间连续整数的算法的多少倍? 请估算一下。
7. 证明等式  $\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$  对每一对正整数  $(m, n)$  都成立。
8. 对于第一个数小于第二个数的一对数字,欧几里得算法将会如何处理?该算法在处理这种输入的过程中,上述情况最多会发生几次?
9.
  - a. 对于所有  $m \geq 1, n \leq 10$  的输入,欧几里得算法最少要做几次除法?
  - b. 对于所有  $m \geq 1, n \leq 10$  的输入,欧几里得算法最多要做几次除法?
10.
  - a. 在欧几里得的书里,欧几里得算法用的不是整数除法,而是减法。请用伪代码描述这个版本的欧几里得算法。



- b. 欧几里得游戏(参见[Bog]) 一开始,板上写有两个不相等的正整数。两个玩家交替写数字,每一次,当前玩家都必须在板上写出任意两个板上数字的差,而且这个数字必须是新的,也就是说,不能与板上任何一个已有的数字相同。当玩家再也写不出新数字时,他就输了。请问,你是选择先行动还是后行动呢?
11. 扩展欧几里得算法 不仅能够求出两个正整数  $m$  和  $n$  的最大公约数  $d$ , 还能求出两个整数  $x$  和  $y$  (不一定为正), 使得  $mx + ny = d$ 。
- a. 在参考资料中查阅扩展欧几里得算法的描述(参见[KnuI]), 然后任选一种语言实现它。
- b. 改写上述程序以对丢番图方程  $ax + by = c$  求解, 系数  $a, b, c$  为任意整数。
12. 带锁的门 在走廊上有  $n$  个带锁的门, 从 1 到  $n$  依次编号。最初所有的门都是关着的。我们从门前经过  $n$  次, 每一次都从 1 号门开始。在第  $i$  次经过时( $i = 1, 2, \dots, n$ ) 我们改变  $i$  的整数倍号锁的状态: 如果门是关的, 就打开它; 如果门是打开的, 就关上它。在最后一次经过后, 哪些门是打开的, 哪些门是关上的? 有多少打开的门?

## 1.2 算法问题求解基础

让我们重申一下在本章概述中已经提出的一个重要观点:

可以认为算法是问题的程序化解决方案。

这些解决方案本身并不是答案, 而是获得答案的精确指令。正是对于精确定义的结构化过程的强调, 才使计算机科学有别于其他学科, 特别是有别于理论数学。理论数学一般仅满足于证明某个问题是否有解, 或者对解的性质进行研究。

现在列出在算法设计分析过程中经历的一系列典型步骤(见图 1.2), 并做简要讨论。

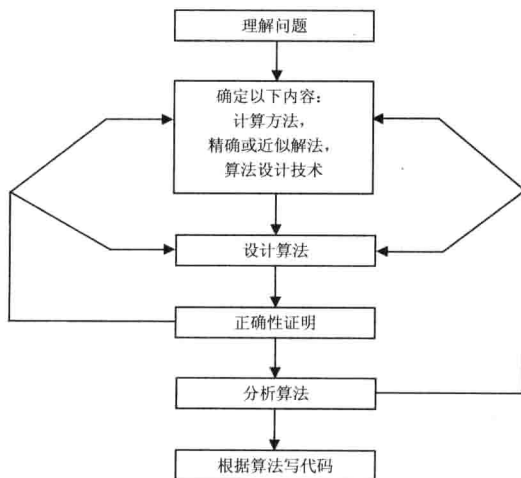


图 1.2 算法的设计和分析过程

## 1.2.1 理解问题

从实践角度看,在设计算法之前,我们首先需要对给定的问题有完全的理解。我们应该仔细阅读问题描述,有疑惑就提出来。试着手工处理一些小规模的例子,考虑一下特殊的情况,必要时再继续提出疑问。

有几类问题会频繁出现在计算机应用中,我们会在下一节中进行讨论。如果待解问题属于其中的一类,就可以用一个已知的算法来求解。当然,了解这些算法如何运作及其优缺点,对解决问题是有帮助的,尤其是在我们不得不从几个可用的算法中选择一个时。但更常见的情况是我们无法找到一个完全可用的算法而不得不自己设计,这往往是一项有趣而又困难的工作。这时,本节所介绍的一系列步骤会有所帮助。

算法的输入,确定了该算法所解问题的一个**实例(instance)**。严格确定算法需要处理的实例的范围是非常重要的(例如,回忆一下前一节中讨论的三种最大公约数算法,它们能处理的实例范围是不同的)。如果不这样做,算法也许能够正确处理大多数输入,但遇到某些“边界值”时就会出错。记住,正确的算法不仅应该能处理大多数常见情况,而且应该能正确处理所有合法的输入。

因此,不要对算法解题的第一步敷衍了事。否则,就要冒不得不返工的风险。

## 1.2.2 了解计算设备的性能

一旦完全了解了待处理的问题,我们还要搞清楚将要运行算法的计算设备的性能。如今,类冯·诺依曼的机器(约翰·冯·诺依曼<sup>①</sup>、A. 博克斯和 H. 戈尔斯坦于 1946 年合作提出的一种计算机体系结构)仍是计算机的主流,我们使用的大多数算法的代码仍然注定要运行在这种系统上。这个体系结构的根本在于**随机存取机(random-access machine, RAM)**。它最主要的假设是:指令逐条运行,每次执行一步操作。相应地,设计在这种机器上运行的算法称为**顺序算法(sequential algorithm)**。

一些更新式的计算机打破了 RAM 模型的核心假设,它们可以在同一时间执行多条操作,即并行计算。能够利用这种计算能力的算法称为**并行算法(parallel algorithm)**。尽管如此,在可预见的未来,RAM 模型下的算法设计和分析的经典技术仍然是算法学的基础。

在算法当中是否需要考虑计算机的计算速度和存储容量呢?如果把设计算法作为科学实验,答案可以说是“否”:就像我们将在 2.1 节讲到的,绝大多数计算机科学家倾向于以一种独立于特定机型的方式来研究算法。如果把算法作为实用工具来设计,答案可能取决于所要解决的问题。今天,即使是一台很“慢”的计算机,它的速度也是快得不可思议的。所以,在很多情况下,我们并不需要担心计算机的速度无法胜任所要处理的任务。然而,总有一些重要的问题,它们原本就是非常复杂的,可能不得不处理海量的数据,或者处理一些对时间很敏感的应用。在这些情况下,认识到特定计算机系统的速度和存储限制是非常必要的。

---

<sup>①</sup> 约翰·冯·诺依曼(John von Neumann, 1903—1957), 20 世纪最杰出的科学家之一。

### 1.2.3 在精确解法和近似解法之间做出选择

下一个重要问题是选择精确解题还是近似解题。前者所对应的算法称为**精确算法(exact algorithm)**,后者则称为**近似算法(approximation algorithm)**。为什么有时要选择近似算法呢?首先,有一些重要的问题在很多情况下的确无法求得精确解,例如求平方根、解非线性方程和求定积分。其次,由于某些问题固有的复杂性,用已知的精确算法来解决该问题可能会慢得让人难以忍受。这种情况往往发生在一个问题涉及数量庞大的选择时。我们将在第3章、第11章和第12章里看到此类难题的一些例子。最后,一个近似算法可以作为更复杂的精确算法的一部分。

### 1.2.4 算法的设计技术

现在,算法解题的必要条件都已具备了,如何设计一个算法来解决一个给定的问题呢?这正是本书希望解答的主要问题,我们会讲一些一般性的设计方法。

那么,什么是算法设计技术呢?

算法设计技术(也称为“策略”或者“范例”)是用算法解题的一般性方法,用于解决不同计算领域的多种问题。

查看本书的目录,你会发现本书大多数章节都是专门介绍某一设计技术的。它们提取出一些已被证实对算法设计非常有用的关键思想。基于以下原因,我们认为学习这些技术是非常重要的。

第一,在为新问题(没有令人满意的已知算法可以解决)设计算法时,它们能够给予指导。所以,学习这样的技术正如“授人以鱼,不如授人以渔”。当然,这并不是说,我们遇到的每个问题都必须应用所有的设计技术。但是放在一起,它们便构成一组强大的工具,为我们日后的学习和工作提供便利。

第二,算法是计算机科学的基础。每一门学科都倾向于对其主要研究对象进行分类,计算机科学也不例外。算法设计技术让我们可以按照内在设计理念对算法进行分类,所以,设计技术使我们能够以一种自然的方式对算法进行分类和研究。

### 1.2.5 确定适当的数据结构

尽管算法设计技术提供了一套通用的方法来对问题算法求解,但为特定问题设计算法仍然是一项具有挑战性的任务。有些设计技术不适用于目标问题;有时多种设计技术需要结合起来解决特定问题;还有一些问题,很难确定是不是特定算法设计技术的具体应用。即使特定的设计技术能够应用于具体问题,设计算法仍然需要设计人员精心构思。当然,选择算法设计技术或者编写算法都可以熟能生巧,但这两者本身并不是简单的工作。

当然,设计人员需要根据算法执行的操作为算法选择适合的数据结构。例如,在1.1节介绍的“埃拉托色尼筛选法”,如果实现时使用链表而不是数组,它的运行时间会更长(为



什么? )。同时请注意,第6章和第7章所讨论的一些算法设计技术,它们非常依赖于对问题实例的数据进行构造和重构。很多年以前,一本很有影响的教材就预言了算法和数据结构将会成为计算机编程的重要基础,它的书名也很贴切,就叫《算法+数据结构=程序》([Wir76])。在面向对象编程的新领域,数据结构对于算法的设计和分析仍然是至关重要的。我们在1.4节将会复习基础的数据结构。

### 1.2.6 算法的描述

我们一旦设计了一个算法,就需要用一定的方式对它进行详细描述。1.1节给出了一个例子,我们已经用文字(虽然较随意,但也是按照一步一步的形式)和伪代码描述了欧几里得算法。这是当今描述算法的两种最常用的做法。

使用自然语言描述算法显然很有吸引力。然而,自然语言固有的不严密性使得我们很难做到简单清晰地描述算法。不过,这也是我们在学习算法的过程中需要努力掌握的一个重要技巧。

伪代码(pseudocode)是自然语言和类编程语言组成的混合结构。伪代码往往比自然语言更精确,而且用伪代码描述的算法往往会更简洁。令人惊讶的是,计算机科学家从来没有就伪代码的形式达成过共识,而是让教材的作者去设计他们自己的“方言”。值得庆幸的是,这些方言彼此还十分相似,任何熟悉一门现代编程语言的人都完全能够理解。

本书选择的方言力求不给读者带来任何困难。出于对简单性的偏好,我们忽略了对变量的定义,并使用缩进来表示 **for**, **if** 和 **while** 语句的作用域。正如大家在前一节里看到的,我们将使用箭头“ $\leftarrow$ ”表示赋值操作,用双斜线“//”表示注释。

在计算机应用早期,描述算法的主要工具是流程图(flowchart)。流程图使用一系列相连的几何图形来描述算法,几何图形内部包含对算法步骤的描述。实践证明,除了一些非常简单的算法以外,这种表示方法使用起来非常不便。如今,我们只能在早期的算法教材里找到它的踪影。

当代计算机技术还不能将自然语言或伪代码形式的算法描述直接“注入”计算机。我们需要把算法变成用特定编程语言编写的程序。尽管这种程序应当属于算法的具体实现,但我们也能将其看作算法的另一种表述方式。

### 1.2.7 算法的正确性证明

一旦完成对算法的描述,我们就必须证明它的正确性(correctness)。也就是说,我们必须证明对于每一合法输入,该算法都会在有限的时间内输出一个需要的结果。举例来说,计算最大公约数的欧几里得算法的正确性依赖于以下条件:等式  $\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$  的正确性(这需要证明,参见习题1.1的第7题);该算法每做一次循环,第二个数字就会变得更小;算法会在第二个数字变为0时停止。

对于某些算法来说,正确性证明是十分简单的;而对于另一些算法来说,却可能是十分复杂的。证明正确性的一般方法是使用数学归纳法,因为算法的迭代过程原本就符合这种证明所需要的一系列步骤。值得一提的是,虽然根据一些特定输入来追踪算法操作的做

法很有意义，但它并不能最终证明该算法的正确性。而为了证明算法是不正确的，则只需给出一个算法不能正确处理的输入实例就足够了。

对近似算法的正确性定义则没有精确算法那么直接。对于一个近似算法来说，我们常常试图证明该算法所产生的误差不超出预定义的范围。第12章会对这方面的研究举一些例子。

## 1.2.8 算法的分析

我们常常希望算法具有许多良好的特性。除了正确性，最重要的特性就是效率(efficiency)了。实际上，有两种算法效率：时间效率(time efficiency)，指出算法运行有多快；空间效率(space efficiency)，说明算法需要多少额外的存储空间<sup>①</sup>。第2章提出了一个分析算法效率的通用框架和一些特殊技术。

算法应该具有的另一个特性是简单性(simplicity)。和效率不同，效率能够用数学的严密性进行精确定义和研究论证，而简单性就像“美”一样，很大程度上取决于审视者的眼光。举例来说，大多数人都承认，在计算  $\gcd(m, n)$  时，欧几里得算法比中学里的计算过程更简单，但欧几里得算法是否比连续整数检验算法更简单则不是那么一目了然的。然而简单性仍然是一个重要的算法特性，值得我们孜孜以求。为什么？因为简单的算法更容易理解和实现，因而相应的程序也往往包含较少的 bug<sup>②</sup>。当然，对于简单性的美学诉求也是让人无法抗拒的。对于同样的问题，有时简单算法的效率比复杂算法更高。遗憾的是，情况并不总是如此，在这种情况下，我们就需要进行谨慎的权衡。

我们希望拥有的另一个算法特性是一般性(generality)。其实，它包含两层意思：算法所解决问题的一般性和算法所接受输入的一般性。对于第一个方面，我们应该注意到，有时候以更一般的形式出现的问题，反而更容易解决。考虑一个两个整数互质的例子，即判断它们是否只拥有唯一的公约数1。实际上这样做更容易：设计一个更一般的算法，用它来计算两个整数的最大公约数，然后解决前面的问题——检查一下最大公约数是否为1。然而，有些情况下，设计一个更一般的算法不仅没有必要，甚至可能是困难的或是完全不可能的。例如，没有必要为了找出  $n$  个数字的中值(即其中第  $\lceil n/2 \rceil$  个最小的元素)而对整个数列排序。又如，解二次方程的标准公式不可能推广到求解任意次数的多项式方程。

至于输入的范围，我们主要关心的是设计这样一个算法，它能够很自然地处理问题可能涉及的输入。例如，对于一个最大公约数算法来说，不把等于1的整数作为可能的输入就很不自然。另一方面，虽然解二次方程根的标准公式能够处理系数是复数的情况，但一般情况下，我们不会将它推广到这种程度，除非明确要求这样做。

如果我们对于某个算法的效率、简单性或一般性不满意，则必须重新设计算法。其实，即使我们对算法做出了肯定的评价，再去探寻另一种算法仍然是有意义的。回想一下前一节中用来计算最大公约数的三种不同算法。一般来说，不要指望依靠一次尝试就能找到最好的算法，最起码，我们应该试着对已有的算法进行优化。例如，相对于1.1节中的最初

① 译注：除了存储算法本身，另外需要的空间。

② 译注：程序设计中的小错误。

版本, 我们已经对“埃拉托色尼筛选法”的实现做了不少改进(你知道是哪些改进吗?)。法国作家、飞行员和飞机设计师安东尼·德·圣埃克苏佩里(Antoine de Saint-Exupéry)有一句名言, 如果我们把它记在心中, 就会更上一层楼: “不是在无以复加, 而是在无以复减的时候, 设计师才知道他已经达到了完美的境界。”<sup>①</sup>

## 1.2.9 为算法写代码

绝大多数算法注定最终以计算机程序的形式实现。为算法编程既是挑战, 也是机遇。挑战在于, 为算法编写的程序可能出现错误或者效率低下。一些有影响的计算机科学家坚信, 除非计算机程序的正确性能够以数学的严密性来证明, 否则我们不能认为程序是正确的。他们开发了一些特殊的技术来实现这种证明([Gri81]), 但到目前为止, 这些形式化验证技术只能处理一些非常小型的程序。

就实用性来说, 对程序的验证还是要依赖测试。测试计算机程序与其说是一门科学, 还不如说是一门艺术。但这并不意味着我们就无需学习这方面的知识。我们可以查看一些专门讲述测试和调试技术的书籍, 但更重要的是, 无论我们实现何种算法, 都要对程序进行彻底的测试及调试。

另一个需要注意的问题是, 本书自始至终都假设算法的输入都在事先确定的范围内, 从而不进行检验。当算法的程序实现用于实际应用时, 这样的检验还是必不可少的。

当然, 算法的正确实现是必要的, 但它还不是全部: 我们当然不愿意用缺乏效率的实现来削弱算法的威力。现代编译器的确为这种需求提供了一定的保障, 尤其当它们处于代码优化模式时。但我们仍然需要掌握一些标准的技巧, 例如: 在循环之外计算循环中的不变式(表达式的值不会随情况而改变); 合并公共的子表达式; 用低开销操作代替高开销操作等(参见[Ker99]和[Ben00], 它们对代码优化和算法编程的其他相关问题做了很好的讨论)。一般来说, 这样的改进对算法速度的影响仅仅是一个常数因子, 而一个更好的算法会使运行时间产生数量级的差异。对于一个已选定的算法, 如果能提高 10%~50%的速度, 上述努力将是值得的。

对于实际程序, 我们还可以利用经验分析来研究它内在算法的效率。这种分析的基本原理是: 提供若干输入, 计算程序的运行时间, 然后对结果进行分析。我们将在 2.6 节讨论经验分析方法的利与弊。

最后, 我们再强调一下图 1.2 中过程的主要含义:

一个好的算法是不懈努力和反复修正的结果, 这是一条规律。

所以, 即使够运气, 获得了一个看似完美的算法思路, 也应该尝试着改进它。

实际上, 这是一件好事, 因为这会让最终结果充满更多的乐趣(的确, 我曾经考虑将这本书命名为《算法的乐趣》)。但另一方面, 我们怎么知道何时应该停止这种努力呢? 现实生活中, 迫使我们停下来的往往是项目进度表和老板的耐心。其实原本也该如此, 完美的

---

<sup>①</sup> 这段对简洁设计的呼唤是在乔恩·本特利(Jon Bentley)的论文集([Ben00])中找到的。这些论文讨论了算法设计和实现的多种问题, 被恰当地命名为《编程珠玑》(*Programming Pearls*)。我真诚地向大家推荐乔恩·本特利和安东尼·德·圣埃克苏佩里的作品。



代价往往是高昂的，而且并不总是提倡的。设计算法是一种工程行为，需要在资源有限的情况下，在互斥的目标之间做权衡，设计者的时间就是这样一种资源。

在学术领域，算法的最优性(optimality)问题引发了有趣而又艰苦的研究。实际上，该问题与某一算法的效率无关，而与所解决问题的复杂度有关：对于给定的问题，任一算法最少需要花费多少气力呢？有些时候，上述问题的答案是已知的。例如，对于长度为  $n$  的数组，任何用比较元素值来对数组进行排序的算法，都需要做大约  $n \log_2 n$  次比较(参见 11.2 节)。但对于许多貌似简单的问题，计算机科学家还无法给出一个最终答案，例如矩阵的乘法。

算法问题求解的另一个重要疑问是：是不是每个问题都能够用算法的方法来解决？我们当然不是在讨论问题无解的情况，例如在判别式为负时求二次方程的实根。在这种情况下，我们能得到的结果，或者说期望得到的结果，应该是算法指出该问题无解。我们也不是在讨论定义模糊的问题。我们所说的是，即使是一些明确定义的问题，它们只要求回答“是”或“否”，可能也是“不可判定”的，即不能用任何算法解决。11.3 节将介绍这种问题的一个重要例子。幸运的是，在实际计算中，绝大多数问题都能够用算法来解决。

图 1.2 的流程图可能过于呆板了，但在结束这一节之前，我们希望读者不要误以为设计算法很无聊。一个千真万确的事实是：发明(或者发现)算法是一个非常具有创造性和非常值得付出的过程。本书的目的就是证明这个事实。

## 习题 1.2

-  1. **古代谜题** 一个农夫带着一只狼、一只羊和一棵白菜来到河边。他需要用船把它们带到河对岸。然而，这艘船只能容下农夫本人和另外一样东西(要么是狼，要么是羊，要么是白菜)。如果农夫不在场的话，狼就会吃掉羊，羊也会吃掉白菜。请为农夫解决这个问题，或者证明它无解(为了有助于解决这个问题，我们假设农夫是一位不爱吃白菜的素食主义者，所以他既不吃羊，也不吃白菜。而且，我们也不假设这只狼是一种受保护的动物)。
-  2. **现代谜题** 有 4 个人打算过桥，他们都在桥的某一端。我们有 17 分钟让他们全部到达大桥的另一头。时间是晚上，他们只有一只手电筒。一次最多只能有两个人同时过桥，而且必须携带手电筒。必须步行将手电筒带来带去，即扔来扔去是不行的。每个人走路的速度不同：甲过桥要用 1 分钟，乙要用 2 分钟，丙要用 5 分钟，丁要用 10 分钟。两个人一起走的速度等于其中走得慢的那个人的速度。(注意，根据网上传言，西雅图附近一家著名软件公司的主考官就是用这个问题来考面试者的。)
3. 当三角形的边长分别是给定的正数  $a, b, c$  时，下面哪个公式可以作为计算三角形面积的算法？
- a.  $S = \sqrt{p(p-a)(p-b)(p-c)}$ ,  $p = (a+b+c)/2$
  - b.  $S = \frac{1}{2}bc \sin A$ ,  $A$  是  $b$  边和  $c$  边的夹角
  - c.  $S = \frac{1}{2}ah_a$ ,  $h_a$  是  $a$  边上的高

4. 用伪代码写一个算法来求方程  $ax^2 + bx + c = 0$  的实根,  $a, b, c$  是任意实系数。(可以假设  $\text{sqrt}(x)$  是求平方根的函数。)
5. 写出将十进制正整数转换为二进制整数的标准算法。
  - a. 用文字描述。
  - b. 用伪代码描述。
6. 写出你最喜欢用的 ATM 在提款时所用的算法(可以依据喜好选用文字或伪代码描述)。
7.
  - a. 求  $\pi$  值问题能够精确求解吗?
  - b. 该问题存在几个实例?
  - c. 在网上查找该问题的算法。
8. 除计算最大公约数问题外, 列出一个你已知有多种算法的问题。其中哪个算法更简单? 哪个算法效率更高?
9. 考虑下面这个算法, 它求的是数值数组中大小最接近的两个元素的差。

算法  $\text{MinDistance}(A[0..n-1])$

//输入: 数字数组  $A[0..n-1]$

//输出: 数组中两个大小相差最少的元素的差值

$dmin \leftarrow \infty$

for  $i \leftarrow 0$  to  $n-1$  do

for  $j \leftarrow 0$  to  $n-1$  do

if  $i \neq j$  and  $|A[i] - A[j]| < dmin$

$dmin \leftarrow |A[i] - A[j]|$

return  $dmin$

尽可能改进该算法(如果有必要, 完全可以抛弃该算法; 否则, 请改进该算法)。

10. 匈牙利籍数学家乔治·波利亚(George Polya, 1887—1985)写了一本书, 名为《怎样解题: 数学思维的新方法》<sup>①</sup>(参见[Pol57]), 这是关于问题求解的最有影响的书籍之一。波利亚将他的观点总结为 4 点。请到网上查找这段话, 或者最好直接在他的书中找, 然后将他的思想和我们在 1.2 节中概括的方法进行比较, 看看它们之间有什么共同之处, 有什么不同之处。

## 1.3 重要的问题类型

计算中能遇到无数种问题, 但只有少数领域的问题引起了研究人员的特殊关注。大体来讲, 这些问题要么具有非常重要的使用价值, 要么具有一些非常重要的特征, 从而使它们成为令人感兴趣的研究课题。幸运的是, 在大多数情况下, 这两种动因往往可以相互强化。

在本节中, 我们开始讲述最重要的问题类型:

- 排序
- 查找

<sup>①</sup> 原书名为 *How to Solve it*, 中译本由上海科技教育出版社于 2011 年出版。



- 字符串处理
- 图问题
- 组合问题
- 几何问题
- 数值问题

本书后面的章节将利用这些问题来阐明不同的算法设计技术和算法分析方法。

### 1.3.1 排序

**排序问题(sorting problem)**要求我们按照升序重新排列给定列表中的数据项。当然,为了这个问题有意义,列表中的数据项应该能够排序(数学家可能会说,这里需要一种全序关系)。在实践中,我们常常需要对数字、字符和字符串的列表进行排序,最重要的是,类似于学校维护的学生信息、图书馆维护的图书信息以及公司维护的员工信息的记录也需要按照数字或者字符的顺序进行排序。在对记录排序的时候,我们需要选取一段信息作为排序的依据。例如,我们可以按照学生姓名的字母顺序,也可以按照学号或者学生个人的平均分数来对学生记录进行排序。这段特别选定的信息称为**键(key)**。计算机科学家常常只关心如何对键的列表进行排序,哪怕表中的元素不是记录,也许仅仅是整数。

我们为什么需要有序列表呢?首先,有序列表可能是所求解问题的输出要求,例如对网上搜索结果进行排序,或对学生的平均成绩进行排序。其次,排序使我们更容易求解和列表相关的问题。其中最重要的是查找问题:这就是为什么字典、电话簿和班级名册都是排好序的。在 6.1 节中我们将会看到一些例子来说明预排序列表的好处。同样原因,在很多其他领域的重要算法(例如几何算法和数据压缩)中,排序也被作为一个辅助步骤。贪婪算法是本书后续章节将要讨论的一个重要算法设计技术,它也要求有序的输入。

到目前为止,计算机科学家已经开发出了几十种不同的排序算法。实际上,有人形象地把发明一种新的排序算法比喻为像设计一种更棒的捕鼠器那么困难。但我们很高兴地告诉大家,寻找更好的“捕鼠器”这个行动还在继续。这种百折不挠的精神是令人钦佩的,原因如下:一方面,有少数不错的排序算法,只需要做大约  $n \log_2 n$  次比较就能完成长度为  $n$  的任意数组的排序;另一方面,没有一种基于“键”值比较(相对比较键值的部分内容而言)的排序算法能在本质上超过它们。

排序领域有着那么多的算法,为什么还会出现这种困境呢?这不是没有原因的。虽然有些算法的确比其他算法更好,但没有一种算法在任何情况下都是最优的。有些算法比较简单,但速度相对较慢;另外一些速度比较快,但更复杂。有些算法比较适合随机排列的输入,而另一些则更适合基本有序的列表。有些算法仅适合排列驻留在快速存储器中的列表,而另一些可以用来对存储在磁盘上的大型文件排序,如此等等。

排序算法有两个特性特别值得一提。如果一个排序算法保留了等值元素在输入中的相对顺序,就可以说它是**稳定的(stable)**。换句话说,如果一个输入列表包含两个相等的元素,它们的位置分别是  $i$  和  $j$ ,  $i < j$ , 而在排好序的列表中,它们的位置分别为  $i'$  和  $j'$ , 那么  $i' < j'$  肯定就是成立的。这种特性很有用,例如,有一个按照字母排序的学生列表,现在我们打算以学生个人的平均成绩来排序:一个稳定算法输出的列表将会把成绩相同的学生仍然

按照字母顺序排列。一般来说,将相隔很远的键交换位置的算法虽然不稳定,但往往速度很快。在后面的章节中,一些重要的排序算法将会证实这一说法。

对于排序算法来说,第二个值得注意的特性是算法需要的额外存储空间。如果一个算法不需要额外的存储空间(除了个别存储单元以外),我们就说它是在位的(in-place)。重要的排序算法有些是在位的,有些则不是。

### 1.3.2 查找

**查找问题**(searching problem)就是在给定的集合(或者是多重集,它允许多个元素具有相同的值)中找一个给定的值[我们称之为**查找键**(search key)]。有许多查找算法可供选择,其中既包括直截了当的顺序搜索,也包括效率极高但应用受限的折半查找,还有那些将原集合用另一种形式表示以方便查找的算法。最后一类算法对于现实应用具有特别重要的价值,因为它们对于大型数据库的信息存取来说是不可或缺的。

对于查找来说,也没有一种算法在任何情况下都是最优的。有些算法比其他算法速度快,但需要较多的存储空间;有些算法速度非常快,但仅适用于有序的数组。和排序算法不同,查找算法没有稳定性问题,但会发生其他问题。具体来说,如果应用里的数据相对于查找次数频繁变化,查找问题就必须结合另外两种操作一起考虑:在数据集合中添加和删除元素的操作。在这种情况下,必须仔细选择数据结构和算法,以便在各种操作的需求之间达到一个平衡。而且,对于用于高效查找(以及添加和删除)的特大型数据集合来说,如何组织其结构是一个不同寻常的挑战,而这对实际应用具有非常重要的意义。

### 1.3.3 字符串处理

近些年来,处理非数值数据的应用增长迅速,引发了研究人员和业界对字符串处理算法的极大兴趣。**字符串**(string)是字母表中的符号所构成的序列。我们尤其关心文本串,它是由字母、数字以及特殊符号构成的;位串是由“0”和“1”构成的;基因序列也可以用字符串模型来表示,只不过它的字母表只包含4个字母,即{A, C, G, T}。但需要指出的是,由于编程语言以及编译的需要,字符串处理算法在计算机科学中一直都非常重要。

如何在文本中查找一个给定的词,这一特殊问题引起了研究人员的特别关注,他们称其为**字符串匹配**(string matching)问题。针对此类查找的特性,人们发明了好几种算法。我们会在第3章中介绍一种非常简单的算法,在第7章中讨论另外两个算法,它们分别基于R. 博伊尔(R. Boyer)和J. 摩尔(J. Moore)的卓越思想。

### 1.3.4 图问题

算法中最古老也最有趣的领域是图算法。通俗地讲,可以认为**图**(graph)是由一些称为顶点的点构成的集合,其中某些顶点由一些称为边的线段相连(下一节将给出一个更严格的定义)。图之所以成为一个令人感兴趣的对象,既有理论上的原因,也有实践上的原因。图可以对广泛的、各种各样的实际应用进行建模,包括交通、通信、社会和经济网络,项目

日程安排以及各种比赛。研究互联网在技术和社会层面的各种具体问题，是现阶段计算机科学家、经济学家和社会学家共同关注的热点问题(参见[Eas10])。

基本的图算法包括图的遍历算法(如何能一次访问到网络中的所有节点)、最短路线算法(两个城市之间的最佳路线是哪条?)以及有向图的拓扑排序(一系列课程的预备课程是相互一致的,还是自相矛盾的?)。幸运的是,这些算法可以用来阐明一些通用的算法设计技术。因此,我们会在本书的相应章节中详细讲述。

有一些图问题在计算上是非常困难的。其中最广为人知的恐怕要数旅行商问题和图填色问题了。**旅行商问题(traveling salesman problem, TSP)**就是找出访问  $n$  个城市的最短路径,并且保证每个城市只访问一次。它的主要应用包括路径规划,主要出现在一些现代应用中,例如电路板和超大规模集成电路的制造, X 射线晶体学以及基因工程等。**图填色问题(graph-coloring problem)**就是要用最少的种类的颜色为图中的顶点填色,并保证任何两个邻接顶点的颜色都不同。这个问题源于若干应用,例如安排事务进度:如果用以边相连的顶点来代表事务,当且仅当独立事务无法排定同时发生时,图填色问题的解才能生成一张最优的日程表。

### 1.3.5 组合问题

从更抽象的角度来看,旅行商问题和图填色问题都是**组合问题(combinatorial problems)**的特例。有一些问题要求(明确地或者隐含地)寻找一个组合对象,例如一个排列、一个组合或者一个子集,这些对象能够满足特定的条件并具有我们想要的特性,如价值最大化或者成本最小化。

一般说来,无论从理论角度还是实践角度来看,组合问题都是计算领域中最难的问题。这是出于以下原因。第一,通常,随着问题规模的增大,组合对象的数量增长极快,即使是中等规模的实例,其组合的规模也会达到不可思议的数量级。第二,还没有一种已知算法能在可接受的时间内,精确地解决绝大多数这类问题。而且,大多数计算机科学家认为这样的算法是不存在的。但这个猜想既没被证实,也没被证伪,所以这仍然是计算机科学理论领域中悬而未决的最重要问题。我们将在 11.3 节中对这个主题做更深入的讨论。

有些组合问题用高效的算法求解,但我们应该把它们当作幸运的例外。这些例外中就包含前面提到的最短路径算法。

### 1.3.6 几何问题

**几何算法(geometric algorithm)**处理类似于点、线、多面体这样的几何对象。古希腊人非常热衷于开发一些过程(当然,他们不会称其为算法<sup>①</sup>)来解决各种各样的几何问题,包括用没有刻度的尺和圆规绘出简单的几何图形,如三角形、圆形等。二千多年以后,对几何算法一度消失的强烈兴趣,在计算机时代复兴了,虽然没有尺和圆规,但有比特、字节以及和古人相同的良好创造力。当然,如今人们对几何算法感兴趣是因为一些完全不同的应

---

① 译注:那时候,花剌子模数学家(al-Khwarizmi)还没有出生呢。

用,例如计算机图形学、机器人技术和断层 X 摄像技术等。

本书只讨论两个经典的计算几何问题:最近对问题和凸包问题。顾名思义,最近对问题(closest-pair problem)求的是给定平面上的  $n$  个点中,距离最近的两个点。凸包问题(convex-hull problem)要求找一个能把给定集合中所有点都包含在里面的最小凸多边形。如果对其他几何算法感兴趣,可以找到大量这方面的专著,例如[deB10], [ORo98], [Pre85]。

### 1.3.7 数值问题

数值问题(numerical problem)是另一个广阔的具体应用领域,涉及具有连续性<sup>①</sup>的数学问题:像解方程和方程组,计算定积分以及求函数的值等。对于大多数这样的数学问题,我们都只能近似求解。另外一个主要的困难是因为这样一个事实:这类问题一般都要操作实数,而实数在计算机内部只能近似表示。而且,对近似数的大量算术操作可能会将大量的舍入误差叠加起来,导致一个看似可靠的算法输出的是被严重歪曲的结果。

多年以来,人们在这个领域中开发出大量成熟的算法,这些算法在许多科学和工程应用中一直扮演着至关重要的角色。但在过去的三十年中,计算机业界将注意力转移到了商业应用。这些新的应用主要需要另外一些算法,它们能对信息存储、取出,再通过网络传输和呈现给用户。作为这种革命性变化的结果,数值分析丧失了它在计算机科学界和业界的应用统治地位。然而,对于计算机专业人士来说,至少掌握数值算法的基本概念还是非常重要的。我们将在 6.2 节、11.4 节和 12.4 节中讨论几个经典的数值算法。

### 习题 1.3

1. 考虑这样一个排序算法:对于待排序的数组中的每一个元素,统计小于它的元素个数,然后利用这个信息,将各个元素放到有序数组的相应位置上去。

**算法** ComparisonCountingSort( $A[0..n-1]$ )

//用比较计数对数组排序

//输入:可排序数组  $A[0..n-1]$

//输出:数组  $S[0..n-1]$ ,  $A$  的元素在其中按照非降序排列

**for**  $i \leftarrow 0$  **to**  $n-1$  **do**

$\text{Count}[i] \leftarrow 0$

**for**  $i \leftarrow 0$  **to**  $n-2$  **do**

**for**  $j \leftarrow i+1$  **to**  $n-1$  **do**

**if**  $A[i] < A[j]$

$\text{Count}[j] \leftarrow \text{Count}[j] + 1$

**else**  $\text{Count}[i] \leftarrow \text{Count}[i] + 1$

**for**  $i \leftarrow 0$  **to**  $n-1$  **do**

$S[\text{Count}[i]] \leftarrow A[i]$

**return**  $S$

- a. 应用该算法对列表“60, 35, 81, 98, 14, 47”进行排序。

<sup>①</sup> 译注:非离散的。

b. 该算法稳定吗?

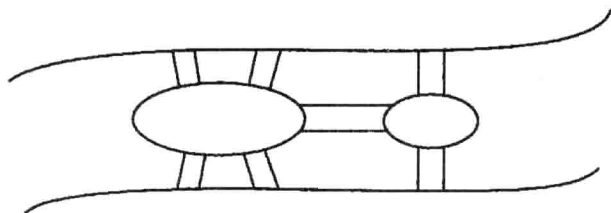
c. 该算法在位吗?

2. 写出你所知道的有关查找问题的算法名称。用简洁的文字描述每个算法。如果一个查找算法都不了解, 正好借此机会自己设计一个。

3. 为字符串匹配问题设计一个简单的算法。



4. **七桥问题** 大家公认, 图论诞生于七桥问题。出生于瑞士的伟大数学家欧拉 (Leonhard Euler, 1707—1783) 解决了该问题。该问题如下: 一个人是否可能在一次步行中穿越柯尼斯堡城中全部的七座桥后回到起点, 且每座桥只经过一次。下面是河以及河上的两个岛和七座桥的草图。

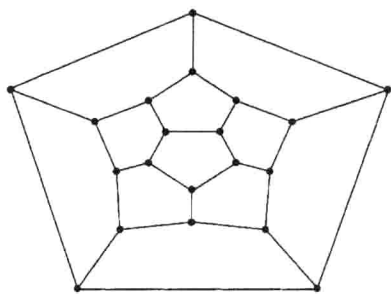


a. 用图的语言定义该问题。

b. 该问题有解吗? 如果认为有解, 请画出步行路线图; 如果认为无解, 解释一下原因并指出, 为了使这种步行路线成为可能, 我们最少需要增加几座新桥。



5. **环游世界游戏(Icosian)** 在欧拉的发现(参见第4题)一个世纪以后, 著名的爱尔兰数学家威廉·哈密顿(William Hamilton, 1805—1865)创造了另一个著名的问题, 它被称为环游世界游戏。这个游戏是在一块圆形的木板上玩的, 板上刻的图如下所示:



寻找哈密顿回路(Hamiltonian circuit)——在回到起点之前, 这一路径能够访问该图的所有顶点并且只访问一次。

6. 考虑以下问题: 假设身处华盛顿特区和伦敦那样发达的地铁系统中, 为地铁乘客设计一个算法, 找出从一个指定车站到另一个车站的最优路径。

a. 该问题的定义有一些模糊, 现实生活中的问题往往就是这样的。对于这个问题来说, 有什么合理的标准可以用来定义“最优路径”?

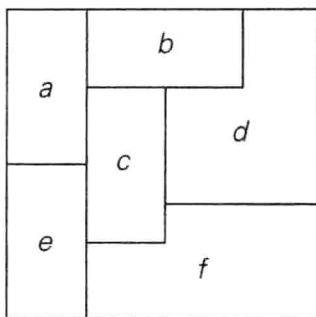
b. 如何用图来对该问题建模?

7. a. 用组合对象的术语重新描述旅行商问题。

b. 用组合对象的术语重新描述图填色问题。



8. 考虑以下地图。



a. 解释一下如何根据图填色问题对该地图着色，使得相邻区域颜色不同。

b. 利用 a 的答案，用最少种类的颜色对该地图着色。

9. 为以下问题设计一个算法：对于  $x$ - $y$  坐标平面上的  $n$  个点的集合，判断它们是不是都落在同一条圆周线上。

10. 写一个程序判断两条线段是否有交点，程序的输入是两条线段  $P_1Q_1$  和  $P_2Q_2$  的端点的  $(x, y)$  坐标。

## 1.4 基本数据结构

由于绝大多数算法关心的是对数据的操作，数据的特殊组织方法在算法设计和分析中扮演了一个至关重要的角色。我们可以将**数据结构(data structure)**定义为对相关的数据项进行组织的特殊架构。数据项的性质是由手头的问题所决定的，它的范围可以从基础的数据类型(例如，整数和字符)到数据结构(例如，我们可以用以一维数组为元素的一维数组来实现矩阵)。事实证明，有一些数据结构对计算机算法尤其重要。由于大家对这些结构已经非常熟悉了，因此这里仅提供一个快速的回顾。

### 1.4.1 线性数据结构

两种最重要的基本数据结构是数组和链表。(一维)**数组(array)**是  $n$  个相同数据类型的元素构成的序列，它们连续存储在计算机的存储器中，我们只要指定数组的**下标(index)**就能够访问这些元素(见图 1.3)。

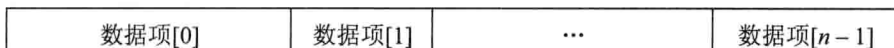


图 1.3  $n$  个元素的数组

大多数情况下，下标都是整数，不是介于 0 到  $n-1$  之间就是介于 1 到  $n$  之间(如图 1.3 所示)。有些计算机语言允许下标介于两个整数边界 **low** 和 **high** 之间，有些甚至允许非数字下标，例如，以月份名作为下标，来索引与每年 12 个月相对应的数据项。

无论位于数组的什么位置，都能用相等的常量时间访问数组中的任何元素。这个特性

是链表所不具备的, 参见下文。

数组可以实现多种其他数据结构, 其中比较出名的是**字符串**。字符串是来自于字母表的字符序列, 并以一个特殊字符来标识字符串的结束。由 0 和 1 组成的字符串称为**二进制串(binary string)**或者**位串(bit string)**。字符串对于文本数据处理、计算机语言定义、程序编译以及抽象计算模型研究都是不可或缺的。字符串的常见操作不同于其他数组(例如, 数字数组)的典型操作。字符串的典型操作包括计算字符串的长度, 按照**字典序(lexicographic order)**, 即字母顺序确定两个字符串在排序时的顺序, 以及连接两个字符串(根据两个给定的字符串构造一个新字符串, 即将第二个字符串附加在第一个字符串的尾部)。

**链表(linked list)**是 0 个或多个称为**节点(node)**的元素构成的序列, 每个节点包含两类信息: 一类是数据; 另一类是一个或多个称为**指针(pointer)**的链接, 指向链表中其他元素(我们用一种称为 null 的特殊指针表明某个节点没有后继元素)。在**单链表(singly linked list)**中, 除了尾节点, 每个节点都包含一个指向下一元素的指针(见图 1.4)。

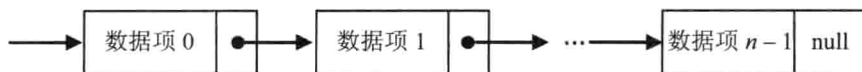


图 1.4  $n$  个元素的单链表

为了访问链表中的某个特定元素, 我们从链表的首节点开始, 沿指针链向前遍历, 直到访问到该特定元素为止。因此, 和数组不同, 访问单链表元素所需要的时间依赖于该元素在链表中的位置。但从积极的方面来说, 链表不需要事先分配任何存储空间, 而且插入和删除的效率也非常高, 只要对相关指针进行重新连接就可以了。

为了增强链表结构的灵活性, 我们可以采取多种方式。例如, 为了方便起见, 链表常常从一个称为**表头(header)**的特殊节点开始。这个节点常常包含着一些关于链表的信息, 例如链表的当前长度。它还能包含其他信息, 例如, 除了包含一个指向头元素的指针外, 还可以包含一个指向尾元素的指针。

另一种扩展结构称为**双链表(doubly linked list)**, 其中除了首尾两个节点, 每一个节点都同时包含指向前趋的指针和指向后继的指针(见图 1.5)。

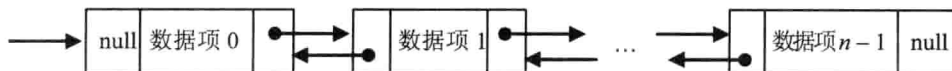


图 1.5  $n$  个元素的双链表

数组和链表都属于一种称为**线性列表(或者简称为列表)**的更抽象的数据结构, 也是最主要的两种表现形式。**列表(list)**是由数据项构成的有限序列, 即按照一定的线性顺序排列的数据项集合。这种数据结构的基本操作包括对元素的查找、插入和删除。

栈和队列是两种特殊类型的列表, 它们尤其重要。**栈(stack)**是一种插入和删除操作都只能在端部进行的列表, 这一端称为**栈顶(top)**。因为在我们脑海中栈的形象往往不是水平的, 而是垂直的, 就像一叠盘子就能很好地模拟栈操作。因此, 当我们在栈中添加一个元素(进栈)或者删除一个元素(出栈)时, 该结构按照一种“后进先出”(last-in-first-out, LIFO)的方式运转, 非常类似于我们对一叠盘子的操作, 我们只能移走最顶部的盘子, 或者在一叠盘子的顶部再加上一个盘子。栈应用很广, 尤其对于实现递归算法来说是不可缺少的。

另一方面, 队列(queue)也是一种列表, 只是删除元素在列表的一头进行, 这一头称为队头(front)[这种删除操作称为出队(dequeue)], 插入元素在表的另一头进行, 这一头称为队尾(rear)[这种插入操作称为入队(enqueue)]。因此, 队列是按照一种“先进先出”(first-in-first-out, FIFO)的方式运行的(就像排在银行柜员前的一个顾客队列)。队列也有许多重要的应用, 其中包括一些图问题的算法。

许多重要的应用, 要求从一个动态改变的候选集合中选择一个优先级最高的元素。有一种数据结构可以满足这类应用, 称为优先队列。优先队列(priority queue)是数据项的一个集合, 这些数据项都来自于一些全序域(最常见的是整数或实数)。对优先队列的主要操作包括查找最大元素、删除最大元素和插入新的元素。当然, 实现优先队列时, 必须使后两种操作产生一个新队列。我们可以直接基于数组或者有序数组来实现优先队列, 但这两者都不是效率最高的解决方案。优先队列还有更好的实现方法, 它基于一种精巧的数据结构, 我们称之为堆(heap)。6.4 节将讨论堆(以及一个基于堆的重要排序算法)。

### 1.4.2 图

就像上一节提到的, 通俗地说, 图可以看作平面上的“顶点”或者“节点”构成的集合, 某些顶点被称为“边”或者“弧”的线段连接。严格来说, 一个图  $G = \langle V, E \rangle$  由两个集合来定义: 一个有限集合  $V$ , 它的元素称为顶点(vertex); 另一个有限集合  $E$ , 它的元素是一对顶点, 称为边(edge)。如果每对顶点之间都没有顺序, 也就是说, 顶点对  $(u, v)$  等同于顶点对  $(v, u)$ , 我们说顶点  $u$  和  $v$  相互邻接(adjacent), 它们通过无向边  $(u, v)$  相连接(无向边英文为 undirected edge)。我们把顶点  $u$  和  $v$  称为边  $(u, v)$  的端点(endpoint), 称  $u$  和  $v$  和该边相关联(incident); 当然, 边  $(u, v)$  也和它的端点  $u$  和  $v$  相关联。如果图  $G$  中的所有边都是无向的, 我们称之为无向图(undirected graph)。

如果顶点对  $(u, v)$  不等同于顶点对  $(v, u)$ , 我们说边  $(u, v)$  的方向是从顶点  $u$  到顶点  $v$ , 其中  $u$  称为尾(tail),  $v$  称为头(head)。也可以说边  $(u, v)$  离开  $u$  进入  $v$ 。如果图的每一条边都是有向的, 则图本身是有向的(directed)。有向的图也称为有向图(digraph)。

为了方便起见, 通常会把图或者有向图的顶点标上字母或整数, 或者按照应用的要求标上字符串(见图 1.6)。图 1.6(a)包含 6 个顶点和 7 条边:

$$V = \{a, b, c, d, e, f\}, E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}$$

图 1.6(b)中包含 6 个顶点和 8 条有向边:

$$V = \{a, b, c, d, e, f\}, E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}$$

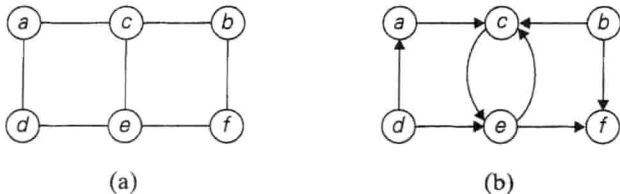


图 1.6 (a)无向图, (b)有向图

我们对图的定义没有禁止圈(loop),即连接顶点自身的边。现在开始,除非另外明确定义,我们将只考虑不含圈的图。因为我们的定义不允许无向图的同一对顶点间拥有多条边,对于 $|V|$ 个顶点的无圈无向图,它可能包含的边的数量 $|E|$ 可以用这个不等式表示:

$$0 \leq |E| \leq |V|(|V|-1)/2$$

(如果每个顶点 $|V|$ 和所有其他 $|V|-1$ 个顶点之间都有边相连,图中边的数量就会达到最大。但是,我们必须对 $|V|(|V|-1)$ 的积除以2,因为每条边被包含了两次。)

任意两个顶点之间都有边相连的图称为**完全(complete)图**。如果完全图具有 $|V|$ 个顶点,它的标准符号是 $K_{|V|}$ 。如果图中所缺的边数量相对较少,我们称它为**稠密(dense)图**;如果图中的边相对顶点来说数量较少,我们称它为**稀疏(sparse)图**。我们处理的是稀疏图还是稠密图,可能会影响图的表示方法,从而影响我们所设计或使用的算法的运行时间。

### 1. 图的表示法

在计算机算法中,图的表示常常在两种方法中选择其一:邻接矩阵或邻接链表。 $n$ 个顶点的**邻接矩阵(adjacency matrix)**是一个 $n \times n$ 的布尔矩阵,图中每个顶点都由一行和一列来表示。如果从第 $i$ 个顶点到第 $j$ 个顶点之间有连接边,则矩阵中第 $i$ 行第 $j$ 列的元素等于1;如果没有这条边,则等于0。例如,图1.6(a)所对应的邻接矩阵可以参见图1.7(a)。

注意,一个无向图的邻接矩阵总是对称的(为什么?),也就是说,当 $i \geq 0, j \leq n-1$ 时, $A[i, j] = A[j, i]$ 。

图或者有向图的**邻接链表(adjacency list)**是邻接链表的一个集合,其中每一个顶点用一个邻接链表表示,该链表包含了和这个顶点邻接的所有顶点(即所有和该顶点有边相连的顶点)。通常,这样一个表由一个表头开始,表头指出该链表表示的是哪一个顶点。例如,图1.6(b)所对应的邻接链表可以参见图1.7(b)。也可以这样解释,对于一个给定的顶点,它的邻接链表指出了邻接矩阵中值为1的列。

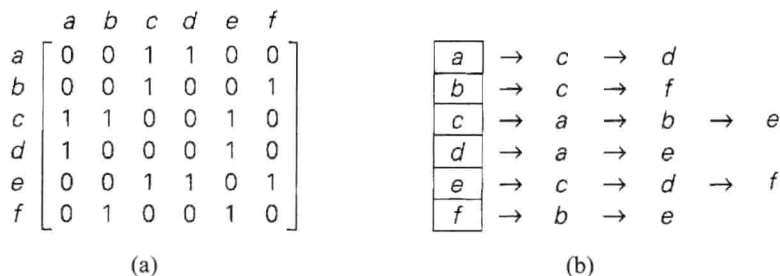


图 1.7 图 1.6(a)所对应的(a)邻接矩阵, (b)邻接链表

如果图是稀疏的,尽管链表中的指针会占用额外的存储器,但相对于相应的邻接矩阵来说,邻接链表占用的空间还是较少。如果是稠密图,情况就正好相反了。一般来说,采用哪种表示法更方便取决于问题的性质,取决于用哪种算法来解决问题,还可能取决于输入图的类型(稀疏的还是稠密的)。

### 2. 加权图

**加权图(weighted graph)**,又称**加权有向图(weighted digraph)**,是一种给边赋了值的图(或

有向图)。这些值称为边的**权重(weight)**或**成本(cost)**。之所以要研究这种图是由于数目众多的现实应用,例如寻找交通网络或者通信网络两点间的最短路径,又如前面提到过的旅行商问题。

图的两种主要表示方法都可以方便地表示加权图。如果用邻接矩阵表示加权图,当存在一条从第  $i$  个节点到第  $j$  个节点的边时,矩阵元素  $A[i, j]$  可以简单地包含这条边的权重;当不存在这样一条边时,则包含一个特殊符号,例如  $\infty$ 。这种矩阵称为**权重矩阵(weight matrix)**或**成本矩阵(cost matrix)**。图 1.8(b)演示了如何用该方法表示图 1.8(a)中的权重矩阵(对于有些应用来说,在邻接矩阵的主对角线上放上 0 会更方便)。加权图的邻接链表在它们的节点中不仅必须包含邻接节点的名字,还必须包含相应的边的权重(图 1.8(c))。

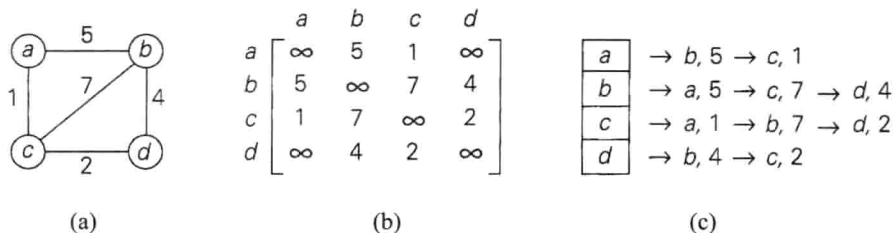


图 1.8 (a)加权图, (b)它的邻接矩阵, (c)它的邻接链表

### 3. 路径和环

图有许多令人感兴趣的特性,但有两个特性对于许多应用都是非常重要的:**连通性(connectivity)**和**无环性(acyclicity)**。两者都基于**路径(path)**的概念。从顶点  $u$  到顶点  $v$  的路径可以这样定义:它是图  $G$  中始于  $u$  止于  $v$  的邻接(以一条边连接)顶点序列。如果一条路径上所有的顶点都是互不相同的,我们说这条路径是**简单(simple)路径**。路径的**长度(length)**就是将路径代表的顶点序列中的顶点数目减 1,恰好和路径所包含的边的数目一致。例如,在图 1.6(a)中,  $a, c, b, f$  是从  $a$  到  $f$  的长度为 3 的简单路径,而  $a, c, e, c, b, f$  是从  $a$  到  $f$  的长度为 5 的路径(非简单路径)。

如果是有向图,我们常常会对有向路径感兴趣。**有向路径(directed path)**是顶点的一个序列,序列中的每一对连续顶点都被一条边连接起来,边的方向等于从第一个顶点指向下一个顶点的方向。例如,图 1.6(b)中,  $a, c, e, f$  是从  $a$  到  $f$  的一条有向路径。

如果对于图中的每一对顶点  $u$  和  $v$ , 都有一条从  $u$  到  $v$  的路径,我们说该图是**连通的(connected)**。如果把连通图的模型定义为代表边的绳子连接着代表顶点的小球,那么所有的东西都应该是连在一起的。如果图是非连通的,这样一个模型会包含几个自我连通的部分,称为该图的**连通分量(connected component)**。严格地说,连通分量是给定图的极大连通子图(不能通过加进某个邻接顶点来扩充)<sup>①</sup>。例如,图 1.6(a)和图 1.8(a)是连通的,而图 1.9 是不连通的,因为,像  $a$  到  $f$ , 它们之间并没有路径。图 1.9 有两个连通分量,分别包含顶点  $\{a, b, c, d, e\}$  和  $\{f, g, h, i\}$ 。

包含若干连通分量的图常常出现在现实应用中。美国州际公路系统的示意图就是一个典型例子。(为什么?)

<sup>①</sup> 给定图  $G = \langle V, E \rangle$  的子图(subgraph)是图  $G' = \langle V', E' \rangle$ , 使得  $V' \subseteq V$  并且  $E' \subseteq E$ 。

对于许多应用来说,知道所考虑的图是否包含回路是非常重要的。**回路(cycle)**是这样一种路径,它的起点和终点都是同一顶点,长度大于0,而且绝不会将同一条边包含两次。例如,  $f, h, i, g, f$  是图 1.9 的一条回路。不包含回路的图称为**无环(acyclic)**图。下一小节将讨论无环图。

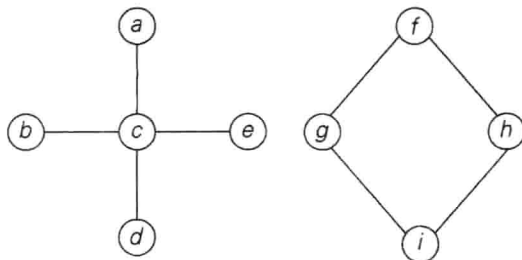


图 1.9 不连通的图

### 1.4.3 树

**树(tree)**,更精确地说,**自由树(free tree)**就是连通无回路图(图 1.10(a))。无回路但不一定连通的图称为**森林(forest)**:它的每一个连通分量是一棵树(图 1.10(b))。

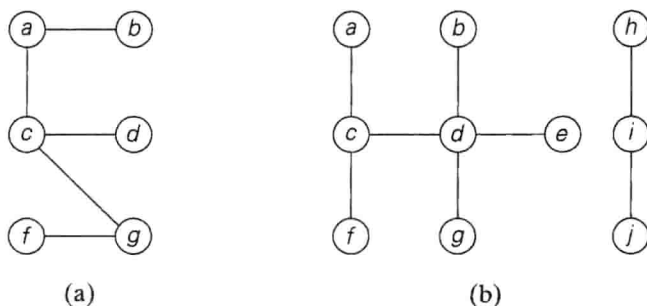


图 1.10 (a)树, (b)森林

树具有其他图没有的一些重要特性。具体来说,树的边数总是比它的顶点数少 1:

$$|E| = |V| - 1$$

就像图 1.9 所示,要使图成为树,这个特征是必要的,但不是充分的。但对于连通图来说,它就是充分的了,而且它可以作为检验连通图是否包含回路的简便方法。

#### 1. 有根树

树的另外一个非常重要的特性就是:树的任意两个顶点之间总是恰好存在一条从一个顶点到另一个顶点的简单路径。这个性质使得以下做法成为可能:任选自由树中的一个顶点,将它作为所谓**有根树(rooted tree)**的**根(root)**。在对有根树的描述中,根常常放在最顶上(树的第 0 层),邻接根的顶点放在根的下面(第 1 层),再下面是和根距离两条边的顶点(第 2 层),然后依次类推。图 1.11 表述了如何将自由树转换为有根树。



有根树在计算机科学中扮演了一个非常重要的角色,这个角色远远要比自由树重要。实际上,为了简单起见,它们常常简称为“树”。树的最典型应用是用来描述层次关系,从文件目录到企业的组织架构。还有很多较不明显的应用,如字典的实现(参见下文),超大型数据集的高效存储(7.4节)以及数据编码(9.4节)。就像第2章中将提到的,树对于分析递归算法也是很有帮助的。限于篇幅,树的应用难以一一列举,但我们应该提一下所谓的**状态空间树**(state-space tree),它强调了两种重要的算法设计技术:回溯和分支界限(12.1节和12.2节)。

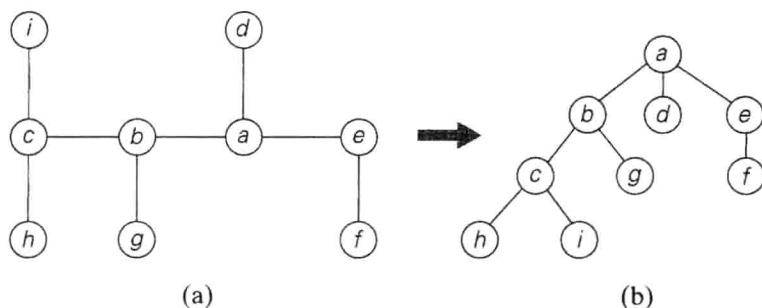


图 1.11 (a)自由树, (b)转换为有根树

对于树  $T$  中的任意顶点  $v$ , 从根到该顶点的简单路径上的所有顶点都称为  $v$  的**祖先**(ancestor)。一般也将顶点本身作为它自己的祖先, 顶点本身以外的所有祖先顶点的集合称为**真祖先**(proper ancestor)集合。如果  $(u, v)$  是从根到顶点  $v$  的简单路径上的最后一条边, 则  $u$  称为  $v$  的**父母**(parent),  $v$  称为  $u$  的**子女**(child)。具有相同父母的顶点称为**兄弟**(sibling)。没有子女的顶点称为**叶**(leaf)节点, 至少有一个子女的顶点称为**父**(parental)节点。所有以顶点  $v$  为祖先的顶点称为  $v$  的**子孙**(descendant), 而  $v$  的**真子孙**(proper descendant)则不包括顶点  $v$  本身。顶点  $v$  的所有的子孙以及连接子孙的边构成了  $T$  的以  $v$  为根的**子树**(subtree)。因而, 对于图 1.11(b)中的树来讲, 该树的根是  $a$ , 顶点  $d, g, f, h, i$  是叶节点, 而顶点  $a, b, e, c$  是父节点,  $b$  的父母是  $a$ ,  $b$  的子女是  $c$  和  $g$ ,  $b$  的兄弟是  $d$  和  $e$ , 以  $b$  为根的子树中的顶点是  $\{b, c, g, h, i\}$ 。

顶点  $v$  的**深度**(depth)是从根到  $v$  的简单路径的长度。树的**高度**(height)是从根到叶节点的最长简单路径的长度。例如, 在图 1.11(b)的树中, 顶点  $c$  的深度是 2, 树的高度是 3。因此, 如果我们约定根的层数是 0, 然后从上到下地计算树的层数, 那么顶点的深度就是它在树中的层数, 而且树的高度就是顶点的最大层数(有一个事实需要提醒大家注意, 有些教材的作者把树的高度定义为树所包含的层的数量, 不像本书把高度定义为从根到叶的最长简单路径的长度。前者定义的高度要比后者大 1)。

## 2. 有序树

**有序树**(ordered tree)是一棵有根树, 树中每一顶点的所有子女都是有序的。为方便起见, 我们可以假设在这种树的图例中, 所有的子女都是从左到右有序排列的。

也可以把一棵**二叉树**(binary tree)定义为有序树, 但其中所有顶点的子女个数都不超过两个, 并且每个子女不是父母的**左子女**(left child)就是父母的**右子女**(right child)。二叉树也

可能为空。图 1.12(a)给出了一棵二叉树的例子。如果一棵二叉树的根是另一棵二叉树的顶点的左(右)子女,则称其为该顶点的左(右)子树。由于左右子树也是二叉树,二叉树其实可以递归定义。这使得很多涉及二叉树的问题可以用递归算法来解决。

在图 1.12(b)中,对图 1.12(a)中二叉树的顶点分配了一些数字。注意,分配给每个父母顶点的数字都比它左子树中的数字大,比右子树中的数字小。这种树称为**二叉查找树(binary search tree)**。二叉树和二叉查找树在计算机科学中有着极其广泛的应用,本书中也能发现不少。尤其是,二叉查找树能够推广为一种更一般的查找树,称为**多路查找树(multiway search tree)**,这种结构对于磁盘上超大数据集的高效存取是必不可少的。

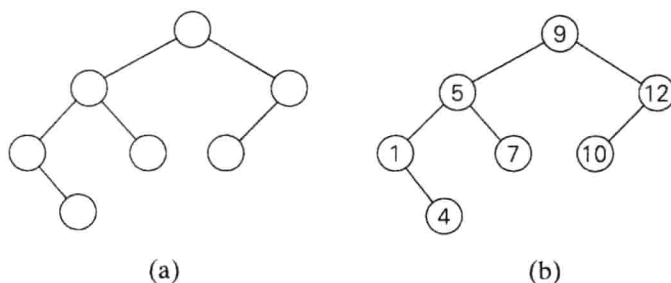


图 1.12 (a)二叉树, (b)二叉查找树

就像本书后面将会提到的,大多数有关二叉查找树的重要算法及其扩展算法的效率取决于这些树的高度。所以,下面的不等式对于分析这些算法是十分重要的。对于高度为  $h$ , 具有  $n$  个顶点的二叉树,我们有以下不等式:

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

出于计算方便,在实现时一棵二叉树常常由代表树顶点的一系列节点来表示。每个节点包含相关顶点的某些信息(顶点的名字或是顶点的值)以及两个分别指向该节点左子女和右子女的指针。图 1.13 说明了图 1.12(b)中二叉查找树的一个实现。

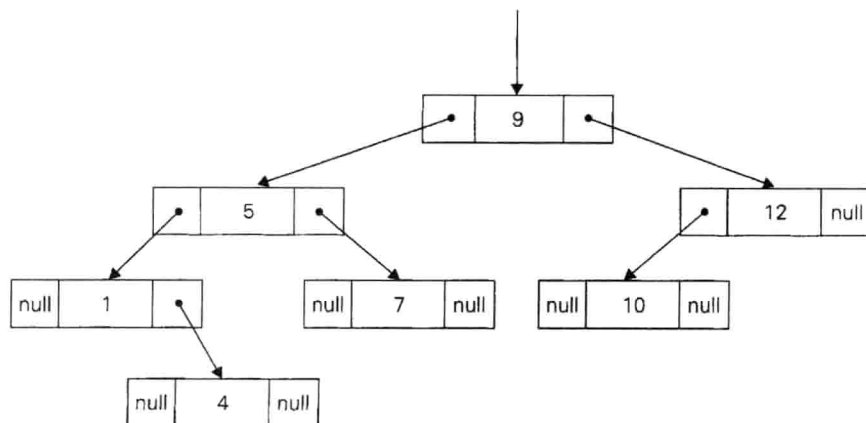


图 1.13 图 1.12(b)中二叉查找树的标准实现

在计算机中,任意一棵有序树可以这样表示:简单地在—个父节点中加入与子女相同

数量的指针。但如果不同节点的子女数目相差很大,这种表示法将变得很不方便。如果像对待二叉树一样,每个节点只包含两个指针,就能避免这种不便。此时,左指针仍然指向节点的第一个子女,而右指针则指向节点的下一个兄弟。因此,这种方法称为**先子女后兄弟表示法**(first child-next sibling representation)。这样,一个顶点的所有兄弟都被一个单独的链表(通过节点的右指针)链接起来了,而且该链表的第一个元素也被它们父节点的左指针指着。当图 1.11(b)中的树应用了这种表示法以后,就会如图 1.14(a)所示。显而易见,这种表示法以一种高效的方式将一棵有序树改造成了一棵二叉树,我们说后者是前者的关联二叉树。只要把指针顺时针“转动” $45^\circ$ ,就能把树变成这种表现形式(参见图 1.14(b))。

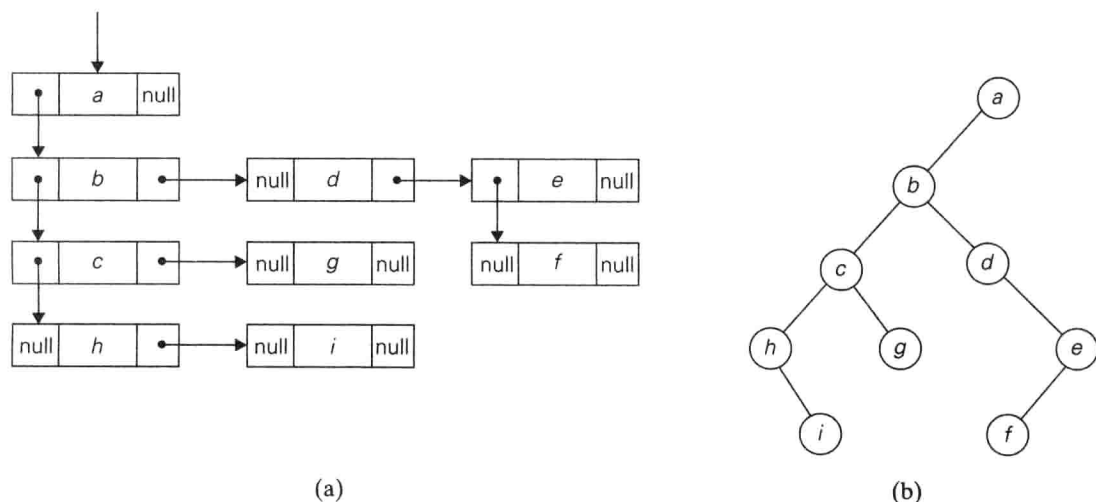


图 1.14 (a)图 1.11(b)的先子女后兄弟表示, (b)它的二叉树表示

### 1.4.4 集合与字典

在数学中,集合概念是一个中心角色。我们可以这样描述**集合(set)**:它是互不相同项的无序组合(可以为空),这些项被称为集合的**元素(element)**。一个特定的集合应该这样定义:要么直接列出元素的确切列表(例如,  $S = \{2, 3, 5, 7\}$ );要么指出集合的特殊属性,也就是集合所有元素都满足,并且只有它们才满足的特性(例如,  $S = \{n: n \text{ 为小于 } 10 \text{ 的质数}\}$ )。最重要的集合运算包括:检查一个给定项是不是给定集合的成员(给定项是不是在集合的元素当中);求两个集合的并集(该集合包含所有属于其中一个集合的元素或者同时属于两个集合的元素);求两个集合的交集,该集合包含所有同时属于两个集合的元素。

集合在计算机应用中可以用两种方法实现。第一种方法只考虑一些称为**通用集合(universal set)**的大集合  $U$  的子集。如果集合  $U$  具有  $n$  个元素,那么  $U$  的任何子集  $S$  能够用一个长度为  $n$  的位串[称为**位向量(bit vector)**]来表示。当且仅当  $U$  的第  $i$  个元素包含在  $S$  中时,向量中第  $i$  个元素为 1。举个例子,如果  $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ ,那么  $S = \{2, 3, 5, 7\}$  应该用位串 011010100 表示。这种集合表示法可以实现非常快速的标准集合运算,但这是以使用大量存储空间为代价的。

为了计算的便捷,我们更常用第二种方法表示集合:用线性列表的结构来表示集合元素。当然,只有对有限集合这个方法才是可行的。幸运的是,与数学不同,大多数计算机应用需要的正是这种集合。但是,要注意集合和列表的两个主要差别。第一,集合不能包含相同的元素,而列表可以。有时,我们可以引进**多重集(multiset)**或者**包(bag)**的概念,来绕过对唯一性的要求,多重集和包是可重复项的无序组合。第二,集合是元素的无序组合,所以,改变集合元素的顺序并不会改变集合。列表——定义为元素的有序组合,则正好相反。这是一个重要的理论上的差别,但幸运的是,它对于许多应用来说并不重要。还有一点需要提醒一下,如果用线性表来表示一个集合,根据手头应用的情况,维护线性表的有序排列可能是必要的。


在计算时,我们对集合或者多重集做的最多的操作,就是从集合中查找一个给定元素、增加新元素和删除一个元素。能够实现这三种操作的数据结构称为**字典(dictionary)**。请注意这种数据结构和 1.3 节中提到的查找问题的关系,显然,我们现在处理的是动态内容的查找。因此,一个字典的高效实现必须在查找的效率和两种操作的效率之间达到一种平衡。有许多方法可以实现字典,范围从简单地使用数组(有序的或无序的)到类似散列法和平衡查找树的复杂技术,我们会在后面对这些技术进行讨论。

许多计算机应用要求动态地把  $n$  个元素的集合划分为一系列不相交的子集。我们可以把集合初始化为  $n$  个单元素的子集以后,再对它做一系列合并和查找的操作。这个问题称为**集合合并问题(set union problem)**。我们会在 9.2 节中结合它的一个最重要的应用来讨论解决该问题的一个高效算法。

大家可能会注意到,在复习基本数据结构的时候,我们总是提到对这些数据结构通常所做的特定操作。很久以前,计算机科学家就认识到了数据和操作之间的这种紧密关系。这种关系催生了**抽象数据类型(abstract data type, ADT)**这一思想。抽象数据类型是由一个表示数据项的抽象对象集合和一系列对这些对象所做的操作构成的。作为抽象数据类型的例子,大家可以再读一读我们对优先队列和字典的定义。虽然抽象数据类型可以用 Pascal 这样传统的面向过程语言来实现(参见[Aho83]),但用 C++和 Java 这样的面向对象语言来实现要更为方便,这些语言用**类(class)**来支持抽象数据类型。

## 习题 1.4

1. 请分别描述一下应该如何实现下列对数组的操作,使得操作时间不依赖于数组的长度  $n$ 。
  - a. 删除数组的第  $i$  个元素( $1 \leq i \leq n$ )。
  - b. 删除有序数组的第  $i$  个元素(当然,原先的数组必须保持有序)。
2. 如果要解决包含  $n$  个元素的线性表的查找问题,已知该线性表是有序的,我们应该如何利用这个特性?请针对下列情况分别解答:
  - a. 该线性表是一个数组。
  - b. 该线性表是一个链表。

3. a. 给出一个空栈在依次进行了以下操作之后的状态:  
push(*a*), push(*b*), pop, push(*c*), push(*d*), pop  
b. 给出一个空队列在依次进行了以下操作之后的状态:  
enqueue(*a*), enqueue(*b*), dequeue, enqueue(*c*), enqueue(*d*), dequeue
4. a. *A* 是一个无向图的邻接矩阵。请说明当邻接矩阵具有何种特征时意味着图具有下列特性:
  - i. 图是完全图。
  - ii. 图具有圈, 即某些顶点具有指向自己的边。
  - iii. 图具有一个孤立顶点, 即没有边和该顶点相连。
 b. 针对图的邻接链表回答同样的问题。
5. 请详细描述这样一个算法, 它将自由树转换为以该树的给定顶点为根的有根树。
6. 证明下列关于  $n$  个顶点二叉树高度的不等式:
 
$$\lfloor \log_2 n \rfloor \leq h \leq n-1$$
7. 请指出如何用下列数据结构实现 ADT 优先队列:
  - a. (无序)数组。
  - b. 有序数组。
  - c. 二叉查找树。
8. 如何实现一个相对较小, 长度为  $n$  的字典(例如, 美国的 50 个州的州名), 已知所有的元素都是唯一的。详细说明每个字典操作的实现。
9. 指出对于下列每个应用来说最适合的数据结构。
  - a. 按照来电的优先顺序接电话。
  - b. 按照客户订单的接收顺序向客户发货。
  - c. 实现一个计算简单算术表达式的计算器。
10.  变位词 设计一个检查两个单词是否为变位词的算法, 也就是说, 是不是能够通过改变一个单词的字母顺序, 来得到另一个单词。例如单词 tea 和 eat 是变位词。

## 小 结

- “算法”是在有限的时间内, 对问题求解的一个清晰的指令序列。算法的每个输入确定了该算法求解问题的一个实例。
- 算法可以用自然语言或者伪代码表示, 也可以用计算机程序的方式实现。
- 在对算法进行分类的各种方法中, 最主要的两种方法如下:
  - ◆ 按照求解问题的类型对算法进行分组。
  - ◆ 按照其内在的设计技术对算法进行分组。
- 重要的问题类型包括: 排序、查找、字符串处理、图问题、组合问题、几何问题和数值问题。

- “算法设计技术”(也称为“策略”或者“范例”)是用算法解题的一般性方法,适用于解决不同计算领域的多种问题。
- 虽然设计算法无疑是一种具有创造性的工作,但我们仍然能够确立一系列涉及这一过程的相互关联的活动。图 1.2 对此进行了描述。
- 一个好的算法常常是不懈努力和反复修正的结果。
- 解决同一个问题的算法常常有好几种。例如,对于计算两个整数的最大公约数,我们给出了三种算法:欧几里得算法、连续整数检测算法以及中学时代的算法。作为对最后一种算法的改进,我们用“埃拉托色尼筛选法”来产生一组质数。
- 算法操作的是数据,这使得数据结构成为决定算法解题效率的关键因素。最重要的基本数据结构是“数组”和“链表”。它们可以用来表示一些更抽象的数据结构,如线性表、栈、队列、图(通过图的邻接矩阵或者邻接链表表示)、二叉树以及集合。
- 一个表示数据项的抽象对象集合和一系列对这些对象所做的操作合称为“抽象数据类型”(ADT)。线性表、栈、队列、优先队列以及字典都是抽象数据类型的重要例子。现代面向对象语言用类来支持 ADT 的实现。



---

本书仅提供部分阅读，如需完整版，请联系QQ: 461573687

提供各种书籍pdf下载，如有需要，请联系 QQ: 461573687

PDF制作说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ: 461573687, 或者 QQ: 2404062482。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！

**备用QQ:2404062482**