大师签名系列

Programming Pearls

Second Edition

编程珠玑

（第三版）

MASTER SIGNATURES SERIES 大师签名系列

[美] Jon Bentley 著
谢君英 石朝江 译

- 近20年来众多大师级程序员一致推崇的作品

中国电力出版社
www.infopower.com.cn

Programming Pearls

编程珠玑（第二版）

“《编程珠玑》第一版是我职业生涯早期阅读过的最有影响力的书籍之一 [illegible]”

——Steve McConnell

《Code Complete》等多部畅销书作者

如果让程序员们列出他们最喜欢的书籍，Jon Bentley的《编程珠玑》通常可以位于经典之列。如同珍珠来自于曾经折磨牡蛎的沙粒，程序设计的珍珠也来自曾经折磨程序员的实际问题。Bentley的珍珠建立在坚实的工程学基础上，在洞察力和创造力的王国中为那些恼人的问题提供了独特而巧妙的解决方案。通过一些精心设计的有趣而且颇具指导意义的程序，本书对众多实用程序设计技巧及基本设计原则作了清晰而机智的描述。因此，《编程珠玑》得到各个层次程序员的青睐并不让人感觉意外。

Bentley彻底更新了第一版中的大多数素材，以反映当今的程序设计方法和环境。此外还增加了以下三个方面的新内容：

- 测试、调试和计时
- 集合表示
- 字符串问题

原来的所有程序都重新进行了改写，并生成了等量的新代码。您可以从本书网站（www.programmingpearls.com）获取所有程序的C或C++实现。

新版本中保持不变的是Bentley对于程序设计问题本质的关注，以及他针对这些问题给出的优美解决方案。不论您是第一次阅读Bentley的经典，还是想再次领略他作品中的新观点，这本书都肯定会成为您最喜爱的图书之一。

Jon Bentley是位于新泽西州Murray Hill的朗讯贝尔实验室计算机科学研究中心的技术委员会委员。Jon自1998年就成为《Dr. Dobb's Journal》杂志的特约编辑。他的“编程珠玑”专栏多年来一直是顶级学术杂志《Communications of the ACM》最风行的特色专栏之一，而本书正是建立在这些专栏的基础之上。

大师签名系列 MASTER SIGNATURE SERIES

责任编辑 / 高 军 宋恩从
封面设计 / 王红卫

ISBN 7-5083-1914-1

9 787508 319148 >

ISBN 7-5083-1914-1
定价：28.00元

大师签名系列

Programming Pearls
Second Edition

编程珠玑
（第二版）

[美] Jon Bentley 著
谢君英 石朝江 译

中国电力出版社
www.infopower.com.cn

Programming Pearls 2nd Edition (ISBN 0-201-65788-0)
Jon Bentley

图书在版编目（CIP）数据
编程珠玑 /（美）本特利著；谢君英，石朝江译. 北京：中国电力出版社，2003
（大师签名系列）
书名原文：Programming Pearls
ISBN 7-5083-1914-1
Ⅰ.编... Ⅱ.①本...②谢...③石... Ⅲ.程序设计 Ⅳ.TP311.1
中国版本图书馆 CIP 数据核字（2003）第 115737 号

丛 书 名：大师签名系列
书　　名：编程珠玑
编　　著：（美）Jon Bentley
翻　　译：谢君英　石朝江
责任编辑：高军　朱恩从
出版发行：中国电力出版社
　　地址：北京市三里河路6号　　邮政编码：100044
　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：汇鑫印务有限公司
开　　本：787×1092　1/16　　印　　张：14.5　　字　　数：276千字
版　　次：2004年4月北京第1版　　2004年4月第1次印刷
定　　价：28.00 元

前　言

计算机程序设计涉及诸多方面。Fred Brooks 在《*Mythical Man Month*》中描述了一幅广阔的画卷；他的作品侧重于讲述管理在大型软件项目中所扮演的关键角色。更为具体一点的是 Steve McConnell 在《*Code Complete*》中介绍的优美的程序设计风格。书中涉及的那些主题对于优秀的软件和程序员来说都是至关重要的。但不幸的是，合理软件工程原则指导下的精巧的应用程序并非总是可以打动人心——除非软件按时全部完成并运转正常。

关于本书

本书涉及的主题是计算机专业领域中更为迷人的一个方面：这是一些超出了可靠工程学范畴、位于洞察力和创造力王国中的程序设计珍珠。如同珍珠来自于曾经折磨牡蛎的沙粒，程序设计的珍珠也来自于曾经折磨程序员的实际问题。书中这些有趣的程序将会教给您重要的程序设计技巧和基本的设计原则。

书中的内容大多出自发表在《*Communications of the ACM*》上的 Programming Pearls 专题，它们被整理、修订，并于 1986 年作为本书的第一版出版。本书对第一版十三章中的十二章做了大量的编辑更动，并加入了三个新的主题。

阅读本书的惟一要求是读者有过使用高级语言进行程序设计的经验。一些高级的技术（比如 C++模板）在书中也会偶尔出现，对这些主题不熟悉的读者可以直接跳到下一章节，而不会造成阅读障碍。

尽管书中的每一章都可以自成体系，但在整体上仍然存在逻辑上的分组。第 1 部分的第 1～5 章回顾了程序设计的基础知识：问题定义、算法、数据结构，以及程序验证和测试。第 2 部分内容主要围绕效率问题展开，效率问题不仅本身十分重要，而且常常是进入有趣的程序设计问题的最好跳板。第 3 部分将会把这些技巧应用到几个关于排序、查找和字符串处理的基本问题中。

给读者的提示：不要读得太快。请仔细阅读，并尝试解决书中提到的问题——某些问题看似容易，但可能需要您花费一到两个小时的精力才能解决。然后，请认真解答每章末尾提出的问题：通过思考并得出解决方案，您从本书学到的大部分知识将得以巩固。如果可能的话，在翻看书后给出的提示和解决方案之前最好和朋友或同事讨论您的观点。每章末尾给出的附加阅读材料并不是代表学术界观点的参考资料列表；我推荐了一些优

秀的参考书目，它们也是我个人书库中重要的组成部分。

本书是为程序员所写。我希望这些问题、提示、解决方案和附加阅读材料将会给程序员带来帮助。事实上，本书已经在多门课程中发挥了它的作用，其中包括算法、程序验证和软件工程。附录 1 中的算法目录是给职业程序员的一个参考，同时说明了如何将本书应用于算法和数据结构方面的课程。

代码

本书第一版中的伪码程序其实已全部实现了，但只有我一个人可以看到这些真正的代码。在第二版中，我重新编写了原有的程序并编写了同样数量的新代码。您可以从本书的网站（www.programmingpearls.com）上获得这些程序。代码中包含了许多用于测试、调试和对函数计时的基本框架。网站上也提供一些其他相关资料。因为现在程序员可以从网上获得十分多的软件，所以这一版本的一个新主题就是介绍如何评价和使用软件组件。

本书的程序采用了简洁的代码风格：短变量名、较少的空行、少数或没有错误检查。在大型软件项目中这是不合适的，但它有利于表达算法的关键思想。答案 5.1 给出了关于这种风格的更详细的背景。

书中包含了一些真正的 C 和 C++程序，但大多数函数都表示成伪码的风格：这样就可以使用较少的篇幅并避免不优雅的语法。符号 for i = [0, n)将 i 从 0 迭代到 n-1。在这种风格的 for 循环中，左右圆括号表示开放的范围（其中不包含边界值），而左右方括号则表示封闭的范围（其中包含边界值）。表达式 function(i, j) 仍然表示调用一个带有参数 i 和 j 的函数，array[i, j]也仍然会访问一个数组元素。

书中列出了许多程序在“我的机器”上的运行情况，该机器的配置是奔腾 II 400MHz 处理器、128MB 内存、Windows NT 4.0 操作系统。我还记录了程序在其他几台机器上的运行情况，书中给出了我所观察到的一些根本区别。所有实验都采用了最大程度的编译器优化。我建议您在自己的机器上进行计时，并打赌您将会找出类似的运行时间比例。

致第一版读者

我希望你翻阅本书的第一反应是“看起来的确与第一版很像。”几分钟之后，我希望你将会发觉“我从未看过此书。”

这一版与第一版关注的问题相同，但它被置于更大的背景之中。计算机技术已经在关键领域（如数据库、网络、用户界面）发生了根本的变化，大多数程序员都应该熟悉这些技术。但在各个领域的中心，编程问题依然是核心。第一版的程序所关注的主题在本书中仍然得以保留。但与第一版相比，本书像是一个更大池塘中的一条更大的鱼。

原书第 4 章中关于实现二进制查找的一节现在被放到了介绍测试、调试和计时的第 5 章中。原来的第 11 章得到了扩展并被拆分为新的第 12 章（关于原来的问题）和第 13 章（关于集合表示）。原来的第 13 章介绍了一种运行在 64KB 地址空间上的拼写检查；现在它已经被去掉了，但它的思想仍然存在于第 13.8 节中。新的第 15 章是关于字符串问题的。许多新节被添加到原来的章中，另外一些节则被删掉了。由于引入了新的问题、新的解决方案以及四个新的附录，本书比上一版本在长度上增加了 25%。

许多旧的案例研究在这一版本中没有发生变化，但也有一些老故事根据现在的情况被重新改写。

第一版致谢

我要感谢许多人对我的支持。在《*Communications of the ACM*》上设立专题的想法最初来自 Peter Denning 和 Stuart Lynn。Peter 勤奋地为 ACM 工作，他使这个专题成为可能并录用我从事此项工作。ACM 总部的成员，尤其是 Roz Steier 和 Nancy Adriance 给了我许多支持，他们让这些专题文章能以原有的形式发表。我尤其要感谢 ACM 的是：它鼓励专题以原有的形式发表；同时感谢许多 *CACM* 的读者，是你们对专题的热心评论使这个扩充版本变得必要并有可能完成。

Al Aho、Peter Denning、Mike Garey、David Johnson、Brian Kernighan、John Linderman、Doug McIlroy 和 Don Stanat 在百忙中抽出时间仔细阅读了本书的每一个章节。我还要感谢以下诸位提出的宝贵意见：Henry Baird、Bill Cleveland、David Gries、Eric Grosse、Lynn Jelinski、Steve Johnson、Bob Melville、Bob Martin、Arno Penzias、Marilyn Roper、Chris Van Wyk、Vic Vyssotsky 和 Pamela Zave。另外，Al Aho、Andrew Hume、Brian Kernighan、Ravi Sethi、Laura Skinger 和 Bjarne Stroustrup 在成书过程中提供了巨大的帮助；西点军校 EF 485 基地的学员测试了本书手稿的倒数第二个草稿。感谢所有人。

第二版致谢

Dan Bentley、Russ Cox、Brian Kernighan、Mark Kernighan、John Linderman、Steve McConnell、Doug McIlroy、Rob Pike、Howard Trickey 和 Chris Van Wyk 仔细阅读了本书第二版。我还要感谢以下诸位提出的宝贵意见：Paul Abrahams、Glenda Childress、Eric Grosse、Ann Martin、Peter McIlroy、Peter Memishian、Sundar Narasimhan、Lisa Ricker、Dennis Ritchie、Ravi Sethi、Carol Smith、Tom Szymanski 和 Kentaro Toyama。感谢 Peter Gordon 以及他在 Addison-Wesley 的同事，他们为这一版的出版提供了许多帮助。

目　录

第 3 部分 产 品

第1部分　预备知识

这5章首先回顾了编程基础。第1章介绍了单个问题的历史。谨慎的问题定义与直接的编程技巧相结合，造就了一流的解决方案。该章说明了本书的中心主题：认真思考分析真实的案例是一件有趣的事，并且很可能在实践中获得收益。

第2章考察了三个问题，强调了算法上的理解如何才能产生简单而有效的代码。第3章概述了数据结构在软件设计中所起的关键作用。

第4章将程序验证作为编写正确代码的一款工具引入。在第9章、第11章以及第14章中，当我们派生出更复杂（而且更快速）的函数时，验证技术将得到广泛使用。第5章向大家展示了我们是如何在实际的代码中实现那些抽象程序的：我们使用脚手架来检测某个函数，使用测试例子来验证函数，以及度量其性能。

第1章 开　篇

程序员提出的问题很简单："我该如何对磁盘文件进行排序？"在我向大家述说我回答这个问题时所犯的第一次错误情形之前，我先给大家一次比我做得更好的机会。你会怎么回答这个问题呢？

1.1 一次友好的对话

我犯的错误就是回答了程序员的问题。我向他简略地介绍了一下如何在磁盘中实现合并排序（Merge Sort）之后，我建议他先钻研一下算法课本，但这并不能满足他的热情。实际上他更关心的还是怎样解决该问题，而不是进一步的学习。后来我给他讲述了流行编程书籍中的磁盘排序程序。该程序包括大约有10多个函数，200行程序代码；我估计实现和测试这些代码最多花费程序员一个星期的时间。

我想我已经解决了他的问题，但是他的犹豫导致我走了回头路。我们之间的对话如下所示，我的问题用斜体字表示。

为什么你老是希望编写自己的排序程序？为什么不使用系统提供的排序程序呢？

我需要在一个大型系统中进行排序，但由于技术不明朗的原因，我不能使用系统文件排序程序。

需要排序的内容究竟是什么？文件中有多少记录？每个记录的格式是什么？

该文件包含至多10000000个记录，每条记录都是一个7位整数。

等一下。假如文件那么小的话，为什么还要费力地使用磁盘排序呢？为什么不在主存储器中对它进行排序呢？

尽管机器有很多MB的主存储器，但是该排序功能属于某个大型系统中的一部分。我想实际上我可能只有1MB的空闲主存。

你能将有关记录方面的内容说得更详细一点吗？

每个记录都是一个7位正整数，并且没有其他的关联数据，每个整数至多只能出现一次。

通过对话，问题描述得更清晰了。在美国，电话号码由3位区号与7位其他号码组成。拨打包含免费区号800的电话是不收费的（当时这是惟一一类免费号码）。实际的免

费电话号码数据库包含有大量的信息，包括免费电话号码、拨打的实际号码（有时有几个号码，还有号码拨打区域以及时间方面的规定）、用户名称和地址，等等。

程序员正在建立一个小小的系统角落（corner）来处理这一类数据库，将要进行排序的整数就是那些免费电话号码。输入文件是一个号码列表（其他信息全被删除了），并且同一号码出现两次以上将是一个错误。预期的输出是一个包含大量号码，并且以升序的方式进行排序的文件。实际环境同时也定义了性能需求。在与系统进行长时间的会话期间，用户请求排序文件的频率大约是每小时一次。在完成排序之前，用户不能做任何事情。因此排序时间不能太长，最合适的运行时间是10秒钟。

1.2 精确的问题陈述

对于程序员来说，这些需求综合起来就是说："如何进行磁盘文件排序？"在着手解决这个问题之前，让我们不带偏见地以更有用的形式进行需求分析：

输入：

所输入的是一个文件，至多包含n个正整数，每个正整数都要小于n，这里$n=10^7$。如果输入时某一个整数出现了两次，就会产生一个致命的错误。这些整数与其他任何数据都不关联。

输出：

以增序形式输出经过排序的整数列表。

约束：

至多（大概）只有1MB的可用主存；但是可用磁盘空间非常充足。运行时间至多只允许几分钟；10秒钟是最适宜的运行时间。

请花片刻时间思考一下这个问题的规格说明。现在你会给该程序员一点什么建议呢？

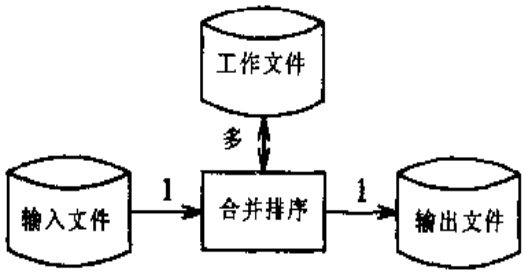
1.3 程序设计

显而易见，该程序使用普通的基于磁盘的合并排序作为起始点，但是对它进行了调整，以反映出我们将要排序整数这一事实。那样一来就将原来的200行代码减少了几十行，同时使代码的执行速度更快了。它大概仍然需要几天的时间使代码调试通过并运行起来。

第二个解决方案甚至更加利用了这个排序问题的特定本性。如果将每个号码存储在7个字节里，那么我们就能够在可用的1M空间中存储大约143000个号码。可是如果我

们将每个号码表示成 32 位整数的话，我们就可以在 1MB 空间中存储 250000 个号码。因此，我们将使用一个在输入文件中带有 40 个通道的程序。在第一个通道中它将 0 到 249999 之间的任意整数读到内存中，并（至多）对 250000 个整数进行排序，然后将他们写到输出文件中。第二个通道对 250000 到 499999 之间的整数进行排序，依此类推，直到第 40 个通道，它将排序 9750000 到 9999999 之间的整数。快速排序（Quicksort）在主存中相当有效，它只需要 20 行代码（我们将在第 11 章中看到这一点）。因此整个程序只需 1 到 2 页的代码即可实现，并且该程序还有一个令人满意的特性，即我们不必担心使用中间磁盘文件的问题。但是不幸的是，为了完成这个程序，我们付出了读取整个输入文件 40 次的代价。

合并排序程序从输入中一次性读取文件，然后在工作文件的辅助下进行排序（注意工作文件会进行多次读取与写入操作），之后再对文件进行一次性写入操作。



40 个通道的算法不使用中间文件，需要多次读取输入文件，但只进行一次输出文件的写入操作。

我们可能更愿意采用下面所示的方案。这个方案结合了前两个方案各自的优势。它对输入只进行一次操作，并且不使用中间文件。

只有将输入文件中的所有整数都表示在可用的主存空间中时，我们才能这么做。这样该问题就归结为我们是否可以将至多一千万个不同整数表示在大约八百万可用的比特中。请根据自己的情况选择一种合适的方法。

1.4 实现纲要

我们考察的只是简单问题，在现实中，位图和位向量很常见，它们的表示方法会更加让我们惊讶。我们可以使用一个20位的字符串来表示一个小型的小于20的非负整数集合。例如，我们可以将集合{1, 2, 3, 5, 8, 13}存储在下面这个字符串中：

```
0 1 1 1 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0
```

集合中代表数字的各个位设置为1，而其他的位全部都设为0。

在现实问题中，每个整数的7个十进制数字表示了一个小于千万的数字。我们将使用一个具有一千万位的字符串表示该文件，在该字符串中，当且仅当整数i在该文件中时，第i个位才打开（设为1）。（程序员发现有200万个空闲的位；问题5仔细研究了严格限制在1MB空间时所出现的问题。）这种表示法使用了这个问题（该问题在排序问题中不太常见）中的三个属性：输入的范围相对要小一些，并且还不包含重复数据，而且没有数据与单个整数以外的每一记录相关联。

给定了表示文件中整数集合的位图数据结构后，我们可以将编写该程序的过程分成三个自然阶段。第一个阶段关闭所有的位，将集合初始化为空集。第二个阶段读取文件中的每个整数，并打开相应的位，建立该集合。第三个阶段检查每个位，如果某个位是1，就写出相应的整数，从而创建已排序的输出文件。如果n是向量中位的数目（在本例中是10000000），该程序可以用伪码如下表示：

```
/* phase 1: initialize set to empty */
    for i = [0, n)
        bit[i] = 0
/* phase 2: insert present elements into the set */
    for each i in the input file
        bit[i] = 1
/* phase 3: write sorted output */
    for i = [0, n)
        if bit[i] == 1
            write i on the output file
```

（请回顾一下序言中的表示法，for i=[0, n)表示i从0到n-1进行迭代。）

对于该程序员来说，本节内容解决他的问题已经足够了。其中还将面临到一些实现方面的细节，这将在问题2、5和7中进行描述。

1.5 原则

一位程序员在电话中向我诉说了他所碰到的问题。花大约一刻钟的时间，我们触及到实际问题，并找到位图解决方案。这样，该程序才用了区区几十行代码，花了他两个

小时来实现。这比在打电话以前我们恐惧设想的编写数百行代码来实现，花费数周编程时间要好得多。该程序的运行速度变得飞快：当磁盘合并排序也许会花许多分钟时，该程序只比读取输入与写入输出所花的时间多一点，大约是10秒。答案3包含了与数个程序有关的计时详细信息，这些程序都是为完成该任务而设计的。

在此案例分析中我们可以发现，这些事实所包含的第一个教训是：仔细分析小问题有时可以带来巨大的实际好处。在本例中，花几分钟的时间来仔细分析，导致了代码长度、编程时间和运行时间减少了一个数量级。Chunk Yeager将军（第一个飞行速度超过音速的人）使用“简单、较少的零部件、易于维护、非常强壮”这样的词汇来赞扬飞机的引擎系统。该程序就包含了这些属性。然而，如果需求规格的某些方面发生变化的话，程序的特定结构将难于修改。本例除了极力宣扬巧妙编程外，还示范了下面的普遍原则：

恰当的问题。问题的定义要占这场战役的90%左右。让人高兴的是，程序员并不满足于我所描述的第一个程序。如果你提出了恰当的问题，问题10、11和12就有非常好的解答；在参考提示和解答之前，请仔细思考一下这些问题。

位图数据结构。此数据结构代表了有限域中的稠集（dense set），每一个元素至少出现一次，没有其他的数据和元素相关联。即使不满足这些条件（例如存在多重元素或额外数据时），也可以使用有限域中的键作为表索引（表具有更复杂的条目）；请参考问题6和8。

多通道（Multiple-Pass）算法。这些算法具有多个输入数据的通道，每读一次就向完成前进一步。在第1.3节中，我们看到了一个具有40个通道的算法；问题5鼓励你开发一个两通道算法。

时间和空间之间的权衡，二者不可偏废。有关时间和空间权衡的编程实践和原理：程序可以通过使用更多的运行时间来节省空间。例如，问题5答案中的两通道算法加倍了程序的运行时间，以减半其空间。可是，就我个人最常有的体验来说，减少程序空间同时也要求减少其运行时间[1]。位图的有效空间结构极大地减少了排序的运行时间。空间的减少导致时间减少有两个原因：需要处理的数据减少意味着处理它的时间也减少了；将数据保存在主存中而不是在磁盘中，可以避免访问磁盘时所花费的时间开销。当然，只有初始设计离最优状态差得很多时，才可能使时间和空间都得到改善成为可能。

简单的设计。法国作家和航空器设计师Antoine de Saint-Exupéry说：“设计师的至高境界不是他不能再往作品中添加什么东西，而是他不能再从中取走什么东西。”更多的

1 权衡（tradeoff）常见于各种各样的工程规范；例如汽车设计时就可能需要考虑通过添加较重的部件以获得更好的加速性能而减少汽车的行驶里程。尽管如此，相互改善还是首选的。在对我曾经驾驶过的小汽车进行检查时，我注意到：汽车基本结构的重量节省可以导致各种不同底盘构件重量的进一步减少，甚至可以消除对某些人来说需要的部分，比如动力转向。

程序员都应该按照这个标准来评价他们的作品。与复杂程序相比，简单程序通常要更加可靠、安全、健壮和有效些，构建和维护时也要更加容易一些。

程序设计阶段。本例示范了将在第12.4节中详细描述的设计过程。

1.6 问题

本书后面将提供部分问题的提示和解答。

1. 如果内存不紧缺，你将如何用一种语言（该语言可以使用库来表示和排序集合）来进行排序？

2. 你会如何使用位逻辑运算实现位向量（比如“与”运算、“或”运算以及“移位”运算）？

3. 运行时效率是一个重要的设计目标，由此产生的程序也具有足够的效率。请在系统中实现位图排序并测试其运行时间；与系统排序及问题1中的排序方法相比较，这种排序方法如何？假设n等于10000000，并且输入文件中包含1000000个整数。

4. 如果你认真地做了问题3，你将碰到生成k个小于n并且互不重复的整数的麻烦。最简单的方法就是使用前k个正整数。这个极端的数据集合不会很大地改变位图方法的运行时间，但它可能会歪曲系统排序的运行时间。你如何可以生成一个包含k个整数的文件，要求这些整数都是惟一的，并且在0到n-1之间随机出现的，次序也是随机的？请力求编写出一个简短高效的程序。

5. 程序员说他有1MB的空闲内存，但是我们拟编写的代码有1.25M。他可以搜集额外的空间，这一点问题都没有。如果1MB是一个严格的分界线，你会推荐怎么办呢？你的算法所需要的运行时间是多少？

6. 如果某程序员告诉你每个整数至多只能出现10次，而不是每一个整数至多只能出现一次，你会给程序员什么样的建议呢？你的解决方案如何变为一个可用存储量的函数？

7. [R. Weil]所勾画的程序有几个缺陷。首先，它假定所输入的整数不会重复出现。如果某一整数出现多次，会发生什么情况呢？在那种情况下，如何可以对程序进行修改以调用错误函数？输入整数小于0、大于或等于n时会发生什么情况呢？如果输入不是数字，那怎么办？程序应该怎么处理这些情况？程序还可采用其他什么稳健的检测措施呢？请描述一下测试该程序的各种小数据集，包括对这些情况以及其他行为不规则情况的适当处理。

8. 程序员面对的问题是，美国的所有免费电话号码区号都是800。免费号码现在包含800、877以及888，并且还在增长。只用1MB空间，你如何去排序所有的免费号码？你如何存储一个免费号码集合以允许实施快速查找，从而确定给定的免费号码是可用的

还是已经占用？

9. 用更多的空间来换取更少的时间所带来的一个问题就是，初始化空间本身可能要占用大量的时间。请介绍一下如何通过某个技巧绕开这个问题，以便在首次访问某向量时就将其条目初始化为 0。你的方案应该使用固定的初始化时间，以及对每个向量固定的访问时间，还要使用和向量大小成比例的额外空间。因为此方法通过使用更多的空间来减少初始化的时间，所以只能在空间比较便宜，时间比较昂贵，而且向量比较稀疏的情况下考虑使用。

10. 在低成本昼夜递送的时代出现以前，商店允许客户通过电话下订单，所订商品几天之后客户才能拿到手。商店的数据库使用客户电话号码作为主键进行检索（客户知道自己的号码并且该键近乎独一无二）。你将如何组织商店的数据库，以便允许高效插入并检索订单呢？

11. 在 20 世纪 80 年代早期，洛克希德公司（Lockheed）的工程师们每天都要将大量的图纸从他们在美国加利福尼亚州桑尼维尔工厂中的计算机辅助设计（CAD）系统运送到圣他库斯（Santa Cruz）测试站。尽管设备之间的距离只有 25 英里，汽车快递服务要花大约一小时的时间(由于交通堵塞以及山路的原因)，每天都要花费 100 美元的成本。请提出一个可选的数据传输方案，并评估需要花费的成本。

12. 载人航天的先驱们很快就意识到在极端的太空环境中也能进行流畅书写的需要。有一则很流行的都市传闻声称，在花费百万美元的研究之后，美国宇航局(NASA)已开发出一支特殊的笔来解决这个问题。根据这则传闻，苏联会如何解决同样的问题呢？

1.7 进阶阅读

这次小练习只是浮光掠影地给出了与程序说明有关的迷人话题。要深入掌握这些关键任务，请参看 Michael Jackson 所著的《*Software Requirements & Specifications*》（Addison-Wesley 于 1995 年出版）。该书的主题独立性极好，并且还辅以一些小小的评论。

在本章所描述的案例分析中，程序员的主要问题不一定是技术上的，更可能是心理上的：因为他正试图解决一个错误的问题，所以他不能取得进步。通过打破概念上的障碍，转而解决一个更简单的问题，这样我们最终解决他的问题了。James L. Adams 所著的《*Conceptual Blockbusting*》（第三版已由 Perseus 于 1986 年出版）研究了这种跳跃，通常它可以说是一种通向创造性思维的令人愉悦的激励。尽管它不是按照程序员的思维编写的，但是许多课程都特别适合于编程问题。Adams 将概念性障碍定义为“妨碍问题解决者正确认识问题或获得解答的心理屏障”；问题 10、11 和 12 鼓励你打破某些屏障。

第2章　啊哈！算法

研究算法对于正在工作的程序员来说是很有用的。有关该主题的课程将使学员掌握重要任务的功能以及解决新问题的技术。在后面的章节中我们将看到高级算法工具有时候会如何对软件系统产生重要的影响，它不仅减少了开发时间，而且加快了执行速度。

算法和那些复杂的思想一样非常关键，对于比较普通级别的编程来说它起着较重要的作用。在马丁·加德纳所著的《Aha!Insight（啊哈！灵机一动）》（我就是将这个书名窃用为本章标题的）一书中，他描述了和我一致的观点："看起来很困难的问题解决起来可能很简单，并且还很可能出人意料之外。"与某些高级方法不一样的是，只有经过广泛的研究之后才能对算法具备那种出神入化的理解力。任何愿意在编码前、编码期间以及编码后认真思考的程序员都可具备这种能力。

2.1　三个问题

为一般起见，本章将围绕三个问题进行展开，在继续阅读之前请尝试回答这些问题。

A. 给定一个包含32位整数的顺序文件，它至多只能包含40亿个这样的整数，并且整数的次序是随机的。请查找一个此文件中不存在的32位整数（至少必有一个遗漏，为什么？）。在有足够主存的情况下，你会如何解决这个问题？如果你可以使用若干外部临时文件但可用主存却只有上百字节，你会如何解决这个问题呢？

B. 请将一个具有n个元素的一维向量向左旋转i个位置。例如，假设n=8，i=3，那么向量abcdefgh旋转之后得到向量defghabc。简单编码使用一个具有n个元素的中间向量分n步即可完成此作业。你可以仅使用几十字节的微小内存，花费与n成比例的时间来旋转该向量吗？

C. 给定一本英语单词词典，请找出所有的变位词集。例如，因为"pots"、"stop"、"tops"相互之间都是由另一个词的各个字母改变序列而构成的，因此这些词相互之间就是变位词。

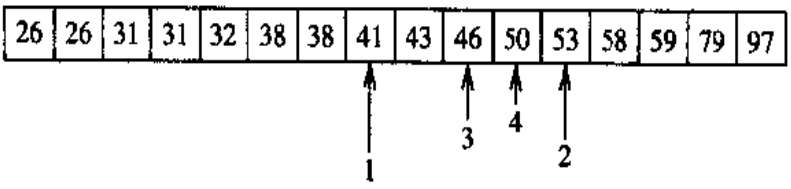
2.2　无所不在的二分查找法

我心里想着1到100之间的某个整数，你猜猜是哪个。50？太小了。75？太大了。

以后依次类推，直到你猜出我所想的那个数为止。如果我所想的整数最初在 1 到 n 之间，那么你可能需要猜 $\log_2 n$ 次。如果 n 是 1000，猜 10 次就行了，如果 n 是 1000000，那么你至多需要猜 20 次。

本例阐述了一种解决众多编程问题的技术：二分查找法。我们最初知道某一对象在一个给定的范围之内，探测操作告诉我们该对象在给定位置的下方、刚好在该位置上或在给定位置的上方。二分查找通过重复探测当前范围中间位置的方式定位该对象。如果探测操作没有找到该对象，那么我们减半当前范围并继续进行下一轮探测。当我们找到我们正在查找的对象，或者范围变为空时，即停止探测。

在编程中，二分查找最常见的应用就是在排序数组中查找某个元素。如果正在查找的元素是 50，那么算法将进行以下探测。



二分查找程序很难编写正确，这已经是众所皆知的事情了。我们将在第 4 章详细研究二分查找程序的代码。

要在具有 n 个元素的表中进行查找的话，顺序查找平均要进行 n/2 次比较，而二分查找的比较次数决不会超过 $\log_2 n$ 次。这在系统性能上可以导致很大的差距；下面这段逸史摘自描述《*Communications of ACM*》的“TWA 预定系统”个案研究，很具典型性：

> 我们有一个程序，它在一大块内存中进行线性查找，一秒钟大约 100 次。随着网络的增长，每条消息的平均 CPU 时间要上升 0.3 毫秒，这猛地使我们大吃一惊。我们发现是线性查找出现了问题，遂将应用程序改为使用二分查找法，结果问题排除了。

我在很多其他的系统中也碰到过同样的故事。程序员一开始可使用简单的顺序查找数据结构。顺序查找通常也够快了。如果程序变得太慢了，那么对表进行排序然后使用二分查找法往往可以排除这个瓶颈。

但是有关二分查找的故事并没有以快速查找排序数组而结束。Roy Weil 在清理一个包含大约 1000 行的输入文件时应用了该技术，这 1000 行里面有一个单个的坏行。不幸的是，坏行不能一眼看出；只能通过运行程序一部分文件，并看到一个错误明显的答案来进行识别，这通常要花费几分钟的时间。他的前辈在调试时试图通过一次只在程序中运行几行代码的方式发现这个问题，但他们求解的过程太慢了，跟蜗牛爬行一样。Weil 是怎样只运行程序十次就找出“元凶”的呢？

有了这些“热身运动”，我们就可以解决问题 A 了。输入是一个顺序文件（请考虑一下磁带或磁盘，尽管磁盘可以随机存取，但从头到尾读取文件通常要快得多）。该文件至多包含 40 亿个次序随机的 32 位整数，我们打算查找一个在文件中并不存在的 32 位整数（因为一共有 2^{32} 个或 4294967296 个这样的整数，所以至少会出现一个遗漏）。如果具有足够的主存，我们就可以使用第 1 章中介绍的位图技术，将 536870912 个 8 位字节分配给一个位图，用于表示此范围内的整数。然而，问题还问到：如果只有上百字节的主存以及若干备用的顺序文件，我们该如何找到遗漏的整数呢？为了使用二分查找，我们必须定义范围、范围内各个元素的表示法以及探测方法，以确定哪半个范围包含了遗漏的整数。我们可以采取什么措施呢？

我们应该使用一个至少包含一个遗漏整数的整数序列作为范围，实际上我们是使用包含所有整数的文件来表示该范围。关键在于我们可以通过计数中点上下元素个数的方式来探测某一范围：上下范围至多包含总范围内元素个数的一半。因为总范围内有一个遗漏元素，所以较小的一半范围内必定包含该遗漏元素。这些就是对问题实施二分查找算法时的大部分要素；在了解答案之前请先将这些要素整理一下，看看 Ed Reingold 是如何进行二分查找的。

二分查找在这些方面的使用只触及到它在编程应用中的一点皮毛而已。求根程序使用二分查找，通过连续插入中间值的方式求解一元方程式；数值分析家将这称为二分法。答案 11.9 中的选择算法随机选择一个元素进行分割，然后对该元素一侧的所有元素递归调用算法自身，它所使用的就是“随机化”二分查找。二分查找的其他用法还包括树数据结构以及程序调试（程序一言不发地死去时，你会在哪里探测源文本，以找到故障语句呢？）。从这些例子中的每一个都可以看出，可将每个程序都看作是对基本二分查找算法的润色，这样就为程序员提供了一种“万金油”算法！

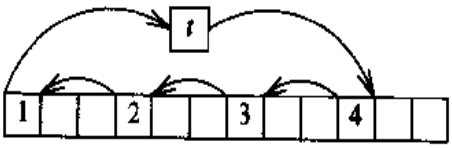
2.3 原语的力量

二分查找是一个寻找问题的解决方案；我们现在就来研究一个具有若干解决方案的问题。问题 B 将一个具有 n 个元素的向量 x 向左旋转 i 个位置，时间上与 n 成比例，并且只有几十字节的的额外空间。这个问题会导致千姿百态的应用程序。有些编程语言提供了旋转作为对向量的原语运算。更重要的是，旋转相当于交换不同大小的相邻内存块：无论什么时候拖拉文本块到文件中的其他什么位置，你都要请求程序交换两个内存块。在许多应用程序中，时间和空间的约束都是很重要的。

有人可能会通过下面的方式解决这个问题：设法将 x 中的前 i 个元素复制到一个临时数组中，接着将余下的 n-i 个元素左移 i 个位置，然后再将前 i 个元素从临时数组中复

制回 x 中的后面位置。然而这个方案使用了 i 个额外的位置，这使得它太浪费空间了。而在另一个不同方法中，我们可以定义一个函数来将 x 向左旋转一个位置（时间上与 n 成比例）；然后调用该函数 i 次，但这又太费时间。

要在有限的资源中解决这个问题的话，很显然程序需要更加复杂一些。有个成功的方法堪称巧妙的杂技表演：先将 x[0]移到临时变量 t 中，然后将 x[i]移到 x[0]中、x[2i]移到 x[i]中，依次类推（将 x 中的所有下标都对 n 取模），直到我们又回到从 x[0]中提取元素，不过在这时我们要从 t 中提取元素，然后结束该过程。当 i=3，n=12 时，该阶段将以下面的次序移动各个元素。



如果该过程不能移动所有的元素，那么我们再从 x[1]开始移动，一直依次进行下去，直到移动了所有的元素时为止。问题 3 要求你将这个想法转换为编码，请务必小心！

不同的算法源自对问题的不同看法：旋转向量 x 实际上就是将向量 ab 的两个部分交换为向量 ba，这里 a 代表 x 的前 i 个元素。假设 a 比 b 短。将 b 分割成 b_l 和 b_r，使 b_r 的长度和 a 的长度一样。交换 a 和 b_r，将 ab_lb_r 转换成 b_rb_la。因为序列 a 已在它的最终位置了，所以我们可以集中精力交换 b 的两个部分了。由于这个新问题和原先的问题是一样的，所以我们以递归的方式进行解决。根据这个算法可以得到一个非常优雅的程序（答案 3 描述了 Gries 和 Mills 的迭代解决方法），但是该算法要求编码细腻，还需要深思熟虑，以确保程序具有足够的效率。

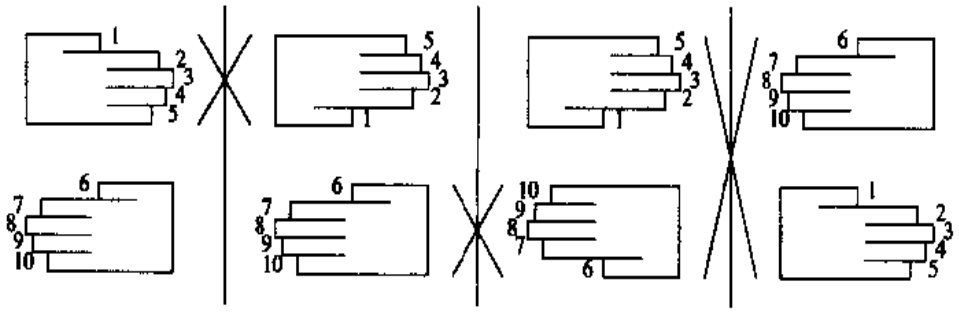
在你最终“啊哈!”领悟之前，该问题看起来似乎很困难：让我们将这个问题看作是把数组 ab 转换成数组 ba 吧，但同时也假定我们具有在数组中转置指定部分元素的函数。我们先从 ab 开始，转置 a 得到 a^rb，再转置 b 得到 a^rb^r，然后再转置整个 a^rb^r 得到$(a^rb^r)^r$，实际上就是 ba。这就导致了下面产生旋转作用的代码；注释显示了 abcdefgh 向左旋转三个元素的结果。

```
reverse(0, i-1)     /* cbadefgh */
reverse(i, n-1)     /* cbahgfed */
reverse(0, n-1)     /* defghabc */
```

Doug McIlroy 给出了将一个具有 10 个元素的数组向上旋转 5 个位置的手摇法例子：先是掌心对着你自己，左手在右手上面，如下页的图所示。

该转置代码在时间和空间上都很有效，并且是这么简短和简单，想出错都很难。Brian Kernighan 和 P. J. Plauger 在其 1981 年的《*Software Tools in Pascal*》中正是使用这段代码在文本编辑器中移动各行的。Kernighan 介绍说第一次执行时这段代码即运行成功了，

而他们前面用于完成类似任务、基于链表的代码却包含好几个 bug。这段代码在好几个文本处理系统中得到了使用，包括最初我用来输入本章的文本编辑器。Ken Thompson 在 1971 年编写了编辑器和转置代码，还声称即使在那个时候，那也可编入神话故事了。



2.4 归拢：排序

现在让我们转向问题 C。给定一本英语单词词典（每个输入行一个单词，字母都用小写），我们必须找出所有的变位词类。有几个不错的理由促使我们研究这个问题。第一个在于技术上：该解决方案是持有正确观点与使用正确工具绝妙配合的结果。第二个理由更具说服力：聚会上，难道你喜欢成为惟一一个不知道“deposit”、“dopiest”、“posited”以及“topside”是变位词的人吗？如果这些理由的说服力还不够，问题 6 描述了一个在真实系统中的类似问题。

这个问题有许多解决方法，但效果都是惊人地差，并且极其复杂。无论什么方法，只要它考虑如何求解某个单词中各个字母的所有置换，那么它注定是要失败的。单词“cholecystoduodenostomy”（这在我的词典中是“duodenocholecystostomy”的变位词）具有 22!个变位词，简单的乘法表明 $22! \approx 1.124 \times 10^{21}$。即使假定每次置换的速度是 1 皮秒那样的快速，这也需要花费 1.1×10^{9} 秒的时间。经验法则“π秒是十亿分之一世纪”（请参见 7.1 节）告诉我们 1.1×10^{9} 秒是几十年。任何希望比较所有单词对的方法在我的机器上至少要运行一个通宵——在我的词典中，大约有 230000 个单词，并且即使一次简单的变位词比较至少也需要花一微秒的时间，所以总的时间粗略算来就是：

230000 个单词×230000 次比较/一个单词×1 微秒/一次比较=52900×10^{6} 微秒=52900 秒≈14.7 小时

你能找到同时避免上述两个缺陷的方法吗？

你“啊哈！”领悟到了将词典中的每个单词都进行签名，这样同一变位词类中的单词会具有相同的签名，然后将具有相同签名的单词归拢到一起。这就将原来的变位词问题转换为两个子问题：选择一个签名；收集具有相同签名的单词。在进一步阅读之前请仔

细考虑一下这些问题。

对于第一个问题，我们将在排序的基础上使用签名[1]：按字母顺序排序单词中的字母。“deposit”的签名就是“deiopst”，“deiopst”也是“dopiest”以及该类其他任意单词的签名。为解决第二个问题，我们可以按照它们的签名排序各个单词。我所听到过的有关这个算法的最佳描述是 Tom Cargill 的手摇式算法：先这样排序（手水平摆动），然后再那样排序（手垂直摆动）。第 2.8 节描述了该算法的一种实现。

2.5 原则

排序。排序最明显的用法就是生成已排序的输出，这既可作为系统规范的一部分，也可作为另一个程序（多半是使用二分查找法的程序）的准备条件。但在变位词这个问题中，排序是极其烦人的；我们进行排序时要将相等的元素（在这种情况下是签名）整理到一起。那些签名还可应用在另一排序应用场合：对单词内的字母进行排序，这为变位词类之内的各个单词提供了一个规范形式。通过在每条记录上放置一个额外的键，然后按照那些键进行排序，排序函数可以被当成一匹驮马一样，不知疲倦地重新排序磁盘文件中的数据。我们在第 3 部分将多次返回到排序这个话题。

二分查找。在已排序表中查找某个元素的算法非常有效，可以在主存和磁盘中使用；其惟一的缺点就是整个表必须是已知的，并且还必须预先进行排序。这个简单算法下面所隐含的策略也常用在众多其他的应用场合。

签名。等价关系定义了各个类时，定义一个签名是很有帮助的，这样类中的每一个项都具有相同的签名，而其他项则没有。对单词内的各个字母进行排序可为变位词类产生一个签名；其他签名则通过排序并计数表示字母重复次数的方式给出（所以“mississippi”的签名可能是“i4m1p2s4”，如果省略 1 的话就成为“i4mp2s4”），或者通过保存一个具有 26 个整数的数组以指示每个字母出现次数的方式给出。签名的其他应用还包括美国联邦调查局用于索引指纹的方法以及对于识别那些听起来类似但是拼写起来不一样的名字具有启发作用的 Soundex 探测法。

名　字	探测法签名
Smith	s530
Smythe	s530
Schultz	s243
Shultz	s432

1 此程序的算法已由多人各自独立地发现了，有记载的日期至少可以回溯到 20 世纪 60 年代中期。

Knuth 在他所著的《*Sorting and Searching*》一书的第 6 章介绍了探测法。

问题定义。我们在第 1 章中已指明，在编程过程中确定用户真正想做什么是编程的一个基本部分。本章的主题就是问题定义中的下一个步骤：我们将使用什么原语来解决该问题呢？在每一案例情况下，“啊哈！”的领悟都会定义一个新的基本运算，使问题变得更普通。

问题解决者的观点。优秀的程序员都有点懒：他们会舒适地靠背坐着，等待灵感出现，而不急于将自己的第一思想转化成编码。当然这必须与在适当的时候直觉性地进行编码相权衡。但是真正的技术就在于对这个适当时候的把握。判断力只会伴随着解决问题以及反省问题解决方案那样的实际体验而出现。

2.6 问题

1. 请思考一下找到给定输入单词的所有变位词这个问题。如果提供了单词和词典，你会如何解决这个问题呢？在回答和查询之前，如果你可以花些时间和空间处理一下该词典，结果又怎样呢？

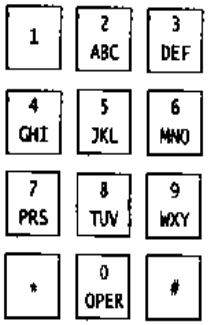
2. 给定一个包含 4300000000 个 32 位整数的顺序文件，请问你如何可以找到一个至少出现了两次的整数？

3. 我们浏览了两个向量的旋转算法，它们都要求代码精细，并且都实现为程序 f。请问 i 和 n 的最大公约数是怎么出现在程序中的？

4. 有些读者指出，尽管三个旋转算法所需要的时间全都和 n 成比例，但是杂耍算法的速度显然是转置算法的两倍：它存储和检索数组元素的次数只有一次，而转置算法则要两次。请在实际的机器上体验一下这些函数，比较它们的速度；尤其要注意内存引用的局部性问题。

5. 向量旋转函数将向量 ab 更改为 ba；那你应如何将向量 abc 转换为 cba 呢？（这个问题建模了如何交换非邻接内存块的问题。）

6. 在 20 世纪 70 年代后期，贝尔实验室部署了一个“用户操作的电话簿辅助”程序，允许员工使用标准的按钮电话查找公司电话簿中的号码。



要查找系统设计者 Mike Lesk 的号码，你只需按下“LESK*M*”（也即“5375*6*”），系统即会说出他的号码。这一设备现在随处可见。这一系统有一个问题，那就是不同的名字可能具有相同的按钮编码；这种现象在 Lesk 系统中发生时，它会要求用户提供更多的信息。给定一个大的名字文件，比如大城市的电话簿，你会如何确定这些“错误的匹配”呢？（但 Lesk 在这样的电话簿上做这个试验时，他发现错误匹配发生率只有 0.2%。）你如何实现给定某一名字按钮编码，然后返回可能匹配名字的集合这一功能呢？

7. 在 20 世纪 60 年代早期，Vic Vyssotsky 和某个程序员一起共事，该程序员需要转置一个存储在磁带上的 4000×4000 矩阵（每一条记录都有相同的格式，并且有几十个字节的大小）。这位同事最初提出了一个程序可能需要花费 50 个小时的运行时间；Vyssotsky 是如何将运行时间减少到半小时的呢？

8. [J. Ullman]给定一个具有 n 个元素的实数集、一个实数 t 以及一个整数 k，请问你如何快速地确定该实数集是否存在一具有 k 个元素的子集，其中各元素的总和至多只能为 t。

9. 顺序查找和二分查找代表了查找时间和预处理时间之间的一种权衡。在具有 n 个元素的表中，需要执行多少次二分查找才能弥补对表进行排序时所需要的预处理时间？

10. 在一位新研究员报到为托马斯·爱迪生做事的那一天，爱迪生要他计算一个空电灯泡壳的体积。这位新雇员使用测径器和微积分算了几个小时之后，得到的答案是 150 立方厘米。过了没几秒钟，爱迪生就算出了“接近 155 立方厘米”的答案，他是怎么做的呢？

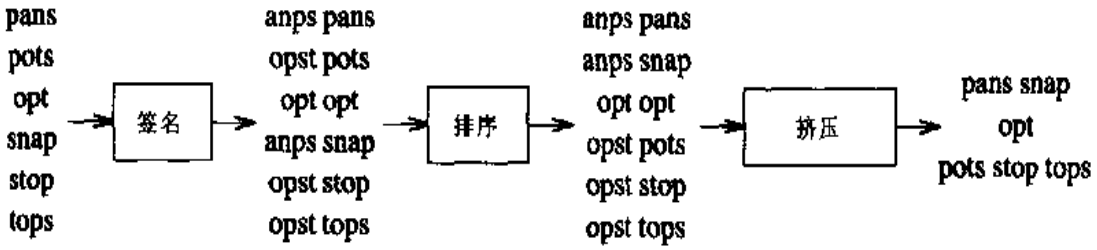
2.7 进阶阅读

第 8.8 节描述了几本算法方面的好书。

2.8 实现变位词程序［补充材料］[2]

我将我的变位词程序组织成三段式的“管道”结构，其中一个程序的输出文件将是下一个程序的输入文件。第一个程序对单词进行签名，第二个程序对已签名的文件进行排序，而第三个程序则将同一个变位词类中的各个单词挤压到同一行中。下面是在具有六个单词的词典中进行处理时的情形。

2 在杂志专栏中，补充材料一般都偏移原文，并印在页面上原文的旁边。尽管它们不是专栏的基本部分，但它们提供了对材料的一些理解。在本书中，它们出现在每一章的后面，并标记为“补充材料”。



该输出包含了三个变位词类。

下面C语言的sign程序假定单词的长度不超过100个字母，并且输入文件只包含小写字母和换行符。（因此，我使用一行命令对词典进行预处理，将大写字母转换为小写字母。）

```
int charcomp(char *x, char *y) { return *x - *y; }

#define WORDMAX 100
int main(void)
{   char word[WORDMAX], sig[WORDMAX];
    while (scanf("%s", word) != EOF) {
        strcpy(sig, word);
        qsort(sig, strlen(sig), sizeof(char), charcomp);
        printf("%s %s\n", sig, word);
    }
    return 0;
}
```

while循环一次将一个字符串读到word中，直到读到文件末尾时为止。strcpy函数将输入单词复制到单词sig中，随即调用C语言标准库中的qsort函数对里面的字符进行排序（qsort函数的参数是待排序的数组、数组长度、每个排序项的字节数以及比较两个项的函数名称，在这里是比较单词内的字符）。最后，printf语句依次输出签名、单词本身和换行。

系统sort程序将具有相同签名的单词全部整理在一起；squash程序将它们输出在单个行中。

```
int main(void)
{   char word[WORDMAX], sig[WORDMAX], oldsig[WORDMAX];
    int linenum = 0;
    strcpy(oldsig, "");
    while (scanf("%s %s", sig, word) != EOF) {
        if (strcmp(oldsig, sig) != 0 && linenum > 0)
            printf("\n");
        strcpy(oldsig, sig);
        linenum++;
        printf("%s ", word);
    }
    printf("\n");
    return 0;
}
```

大部分的工作将由第二个 printf 语句来完成：对于每一个输入行，它都要输出第二个字段和一个空格。if 语句判断签名之间的变化：假如 sig 与 oldsig（当前值）不同的话，随即输出换行（只要该记录不是文件中的第一条记录）。最后一个 printf 输出最后一个换行符。

利用小型输入文件测试完这些简单的程序之后，我可键入下面的命令构建一个变位词列表：

```
sign <dictionary | sort | squash >gramlist
```

该命令将把文件 dictionary 传到程序 sign 中，接着将 sign 的输出传给 sort 程序，然后再将 sort 的输出传给 squash，最后将 squash 的输出写入文件 gramlist 中。该程序运行了 18 秒：sign 花了 4 秒；sort 花了 11 秒；squash 花了 3 秒。

我在包含 230000 个单词的词典上运行该程序，但是该词典没有包含很多的-s 和-ed 词尾。下面是一些比较有趣的变位词类。

```
subessential suitableness
canter creant cretan nectar recant tanrec trance
caret carte cater crate creat creta react recta trace
destain instead sainted satined
adroitly dilatory idolatry
least setal slate stale steal stela tales
reins resin rinse risen serin siren
constitutionalism misconstitutional
```

第3章　数据结构程序

大多数程序员都看到过数据结构程序，而大多数好的程序员都意识到自己起码编写过一个数据结构程序。这些数据结构程序本该短小、干净、漂亮，而它们实际却是那种巨大的、混乱的、丑陋不堪的程序。我曾经见过几个程序，它们可以归结为类似以下所示的代码：

```
if (k ==   1) c001++
if (k ==   2) c002++
   ...
if (k == 500) c500++
```

尽管程序实际所完成的任务要稍微更复杂一些，但是不会将它们误认为是要对 1 到 500 之间的每一个整数在文件中出现的次数进行计数。每个程序所包含的代码都超过 1000 行。今天大多数程序员马上都会意识到，如果使用另一个不同的数据结构——一个具有 500 个元素的数组——来代替 500 个单独的变量，那么使用一个只有原来程序零头大小的程序就可以完成该任务。

因此，本章标题的含义就是：合适的数据视图确实构造了程序。本章描述了各种不同的程序，通过重构它们的内部数据，使其变得更加简短（而且更好）。

3.1　调查程序

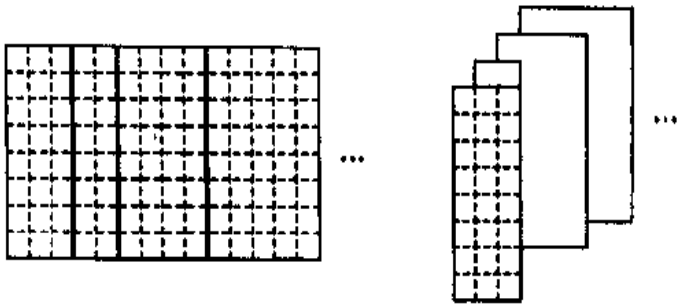
我们将要研究的下一个程序总结了由某一特定大学的学生所填写的两万份问卷。部分输出结果如下所示：

	Total	US Citi	Perm Visa	Temp Visa	Male	Female
African American	1289	1239	17	2	684	593
Mexican American	675	577	80	11	448	219
Native American	198	182	5	3	132	64
Spanish Surname	411	223	152	20	224	179
Asian American	519	312	152	41	247	270
Caucasian	16272	15663	355	33	9367	6836
Other	225	123	78	19	129	92
Totals	19589	18319	839	129	11231	8253

对于每一个种群而言，男性数目与女性数目之和要稍微小于总数，因为有些人没有回答某些问题。实际的输出更加复杂。我显示了所有7行以及总计行，但只显示了6列，表示总计和两个其他的类别，即国籍和性别。在实际的问题中，共有25列，表示8个类别以及三个类似的输出页面：其中两个页面各自用于两个独立的校园；另一个页面用于那两个页面的总和。还有其他少数关系密切的将要输出的表，比如拒绝回答每一个问题的学生的数目。每一道问卷都用一个记录进行表示。记录中的条目0包含了编码为0~7之间整数的种群（有7个类别和“拒绝”），条目1包含校园（0~2之间的整数），条目2包含国籍，等等，直到条目8。

程序员从系统分析员的概要设计出发编码该程序；进展两个月之后产生了上千行的代码，而他估计任务也才完成了一半的样子。看了他的设计之后我就理解他的这种困境了：该程序大约有350个截然不同的变量——25列×7行×2页。变量声明之后，该程序包含了一个老鼠窝式的逻辑嵌套，该逻辑决定了读取每个输入记录时需要增加哪些变量。考虑一下，想想你会怎样编写该程序。

关键的决定在于应该将这些数字存储为一个数组。作出下一个决定要更困难一些：应该按照输出结构（沿着校园的3个维度、种群以及25个列）或者输入结构（沿着校园的4个维度、种群、类别以及类别之内的值）对数组进行布局吗？不计校园的维度，这些方法可以看作是：



以上两个方法都是可行的；在我的程序中，三维视图（左边）导致了读数据时工作量稍微大些，写数据时工作量稍微小些。该程序占用了150行的代码：80行用于构建变量；30行用于产生我所描述的输出；40行用于产生其他的表。

计数程序和调查程序都是两个不必要的大型程序；两个都包含了大量被单个数组替换掉的变量。代码的长度减少了一个数量级，从而就产生了正确的程序，这样的程序开发速度快、易于测试与维护。尽管在任一应用中不太要紧，但是与大型程序相比，两个小程序在运行时间和空间方面效率都更高。

既然小程序能够完成任务，那么为什么程序员还要去编写大程序呢？原因之一就是程序员缺乏第2.5节中所提到的那种重要的惰性；他们总是匆匆忙忙将他们的第一灵感编写出来。但是在我所描述的两种情况下，有一个更深层的问题：在程序员思考问题时所采用的

语言中，数组一般用作在程序一开始就已初始化并且不再更改的固定表。在第 1.7 节中所描述的 James Adams 的书籍中，他会说程序员对于使用动态计数器数组具有“概念上的障碍”。

有很多其他的原因会导致程序员犯这样的错误。在准备撰写本章时，我在我自己的调查程序代码中发现了一个类似的例子。主输入循环具有 8 个语句块，每块有 5 个语句，共有 40 行代码。其中前两个块可以表示为：

```
ethnicgroup = entry[0]
campus = entry[1]
if entry[2] == refused
    declined[ethnicgroup, 2]++
else
    j = 1 + entry[2]
    count[campus, ethnicgroup, j]++
if entry[3] == refused
    declined[ethnicgroup, 3]++
else
    j = 4 + entry[3]
    count[campus, ethnicgroup, j]++
```

初始化数组 offset 以包含 0、0、1、4、6、…之后，我可以使用 6 行代码替换原先的 40 行代码：

```
for i = [2, 8]
    if entry[i] == refused
        declined[ethnicgroup, i]++
    else
        j = offset[i] + entry[i]
        count[campus, ethnicgroup, j]++
```

我对代码的长度减少了一个数量级感到非常满意，以致我都看不到有个人正看着我。

3.2 表单字母编程

你刚才已经键入了姓名和密码，以登录到你中意的购物站点。你所看到的下一页就是如下所示的样子：

```
Welcome back, Jane!
We hope that you and all the members
of the Public family are constantly
reminding your neighbors there
on Maple Street to shop with us.
As usual, we will ship your order to
    Ms. Jane Q. Public
    600 Maple Street
    Your Town, Iowa 12345
  ...
```

作为一个程序员，你应该意识到计算机从数据库中查询你的姓名并取回以下字段数据：

```
Public|Jane|Q|Ms.|600|Maple Street|Your Town|Iowa|12345
```

但是程序该如何精确地从你的数据库记录中构建那个定制的 Web 页面呢？草率的程序员可能很想像下面那样开始编写程序：

```
read lastname, firstname, init, title, streetnum,
    streetname, town, state, zip
print "Welcome back,", firstname, "!"
print "We hope that you and all the members"
print "of the", lastname, "family are constantly"
print "reminding your neighbors there"
print "on", streetname, "to shop with us."
print "As usual, we will ship your order to"
print "    ", title, firstname, init ".", lastname
print "    ", streetnum, streetname
print "    ", town ",", state, zip
  ...
```

这一类程序很迷人，但是很冗长。

更好的方法就是编写一个依赖于下面这样的表单字母模式（form letter schema）的表单字母生成器（form letter generator）：

```
Welcome back, $1!
We hope that you and all the members
of the $0 family are constantly
reminding your neighbors there
on $5 to shop with us.
As usual, we will ship your order to
    $3 $1 $2. $0
    $4 $5
    $6, $7 $8
  ...
```

表示法$i 表示记录中的第 i 个字段，所以$0 表示姓，等等。下面的伪码将解释该模式。这段伪码假定字母$字符在输入模式中写为$$。

```
read fields from database
loop from start to end of schema
    c = next character in schema
    if c != '$'
        printchar c
    else
        c = next character in schema
        case c of
            '$':        printchar '$'
            '0' - '9': printstring field[c]
            default:    error("bad schema")
```

该模式在程序中描述成了一个长长的字符数组。在该数组中，文本行由换行符结尾（Perl和其他脚本语言在这方面甚至更加简单；我们可以使用诸如$lastname那样的变量）。

与编写明显的程序相比，编写生成器和模式或许更加简单些。将数据从控件中分离开来可以使你大大受益：如果字母重新设计，那么可以在文本编辑器中操作该模式，第二个特定的页面准备起来也要简单些。

报表模式的概念可以大大简化我曾经维护过的一个具有5300行代码的COBOL程序。该程序的输入描述了一个家庭的财务状态；其输出是一个目录单，汇总了该状态并推荐未来的策略。某些数字是：120个输入字段；在18个页面中共有400个输出行；300行用于清理输入数据的代码；800行用于计算；还有4200行用于编写输出。我估计4200行的输出代码可以使用一个至多几十行代码的解释程序和一个400行的模式来代替；计算代码可以保持不变。重新以这种形式编写程序所产生的COBOL代码至多只有原来大小的三分之一，并且维护起来也更加容易。

3.3 数组例子

菜单。我希望我的Visual Basic程序的用户可以单击某个菜单项来进行选择。我随意翻看了一下大量的优秀示例程序，发现其中有一个程序，竟允许用户在八个选择项之间进行选择。在审查菜单幕后的代码时，我发现item0看起来有点类似下面所示样子：

```
sub menuitem0_click()
    menuitem0.checked = 1
    menuitem1.checked = 0
    menuitem2.checked = 0
    menuitem3.checked = 0
    menuitem4.checked = 0
    menuitem5.checked = 0
    menuitem6.checked = 0
    menuitem7.checked = 0
```

item1仅作以下更改，其他几乎都是一样的：

```
sub menuitem1_click()
    menuitem0.checked = 0
    menuitem1.checked = 1
    ...
```

item2到item7也是一样的。总之，选择菜单项占用大约100行的代码。

我自己也像那样编写过代码。一开始一个菜单中只有两个项，代码还是比较合理的。当我添加第三个、第四个以及以后的项时，我对代码幕后的功能性如此兴奋，以至于代码变得越来越混乱。

稍微观察一下，我们就可以将大多数代码移到单个函数 uncheckall 当中，该函数将每一个 checked 字段都设置为 0。之后，第一个函数就变为：

```
sub menuitem0_click()
    uncheckall
    menuitem0.checked = 1
```

但我们仍然还有 7 个其他相类似的函数。

幸运的是，Visual Basic 支持菜单项数组，所以我们可以用下面这一个函数来代替 8 个类似的函数：

```
sub menuitem_click(int choice)
    for i = [0, numchoices)
        menuitem[i].checked = 0
    menuitem[choice].checked = 1
```

将重复代码聚集到一个通用的函数中使代码由 100 行减少到了 25 行，而明智地使用数组将代码减少到了 4 行。添加下一个选择项时更加容易，而且也完全清除了潜在的错误代码。只用寥寥几行代码，这种方法就解决了我的问题。

错误消息。“脏”系统的代码里面散布着许许多多的错误消息，并且这些错误消息还和其他的输出语句混杂在一起；但是“干净”系统只通过单个函数访问这些错误消息。让我们看一看在“脏”组织和“干净”组织下执行以下请求的难度：产生一个所有可能错误消息的完整列表；将每个“严重”错误消息改为发声报警；将错误消息翻译成法语或德语。

日期函数。给定某一年以及该年中的某一日，返回其所在月和月中的日子；例如，2004 年中的第 61 天就是 3 月 1 日。Kernighan 和 Plauger 所著的《*Elements of Programming Style*》一书，为完成此任务给出了一个 55 行代码的程序，该程序直接取自其他人的编程文本。然后他们又通过一个带有 26 个整数的数组，为该任务提供了一个只有 5 行代码的程序。问题 4 将介绍日期函数中大量存在的表示问题。

单词分析。许多计算问题都是在对英语单词进行分析的过程中产生的。在第 13.8 节中我们将看到，拼写检查程序如何使用“后缀剥离（suffix stripping）”来浓缩其词典：它们存储单个单词“laugh”时不会存储所有的末尾变形（“-ing”、“-s”、“-ed”，等等）。语言学家为这一类任务提出了实体规则（substantial body of rules）。在 1973 年构建第一个实时文本语音合成器时，Doug McIlroy 认识到是代码误导了这一规则；相反，他使用 1000 行的代码和一个 400 行的表来编写该程序。当某人不是通过添加表而是去修改程序时，其结果是用 2500 行额外的代码来完成增加的 20%的工作。McIlroy 断言，通过添加更多的表，现在他可以使用不到 1000 行的代码来完成扩展的任务。需要自己尝试一下类似的规则集的话，请参见问题 5。

3.4 构造数据

什么是构造良好的数据？随着时间的推移，这个标准也在稳步上升。在前些年，构造数据的意思就是选择良好的变量名称。一旦程序员在哪里使用过并行数组或寄存器偏移量，随后，语言就会将记录或结构以及指向它们的指针合并在一起。我们学会了使用具有诸如 insert 或 search 那样的名称的函数替换操作数据的代码：那将有助于我们更改表示而不会破坏其余部分代码。David Parnas 扩展了该方法，他观察到通过考察系统将处理的数据可以对良好的模块结构有更深的了解这样一个事实。

“面向对象的编程”采取了下一步。程序员学会了在设计中标识各种基本的对象，向世人公布某一抽象概念及其基本运算，并从视图中隐藏其实现细节。诸如 Smalltalk 和 C++那样的语言允许我们将那些对象封装在类中；在第 13 章中我们研究集合（set）的抽象和实现时将详细讨论这种方法。

3.5 针对特定数据的强大工具

在过去艰难的日子里，程序员构建每一个程序都需要从头开始。现代化的工具使程序员（以及其他人）轻而易举地就可以构建各种各样的应用程序。本小节列出了一个简短的工具列表，它只是象征性的，不是一个完整的工具列表。每一个工具都利用某一数据视图来解决某一特定但又是常见的问题。诸如 Visual Basic、Tcl 以及各种不同 shell 等语言提供了用于连接这些对象的“胶水”。

超文本。在 20 世纪 90 年代早期，当时还只有几千个 Web 站点，我就对从 CD-ROM 转移到 Web 上的参考资料着了迷。数据收集功能之强大让人震惊：百科全书、词典、年鉴、电话簿、古典文学、教科书、系统参考手册，还有更多的东西，都掌握在我的手掌心了。不幸的是，各种不同数据集的用户界面同样让人震惊：每一个程序都有自己的花样。今天，所有那样的数据（以及更多东西）我都是在 CD 和 Web 上进行访问，所选的界面通常是 Web 浏览器。这将使用户的生活更惬意，对于实施者来说也是类似的。

名称-值对。书目数据库可能具有如下所示的条目：

```
%title      The C++ Programming Language, Third Edition
%author     Bjarne Stroustrup
%publisher  Addison-Wesley
%city       Reading, Massachusetts
%year       1997
```

Visual Basic 采取这种方法来描述界面上的控件。窗体左上角的文本框可以使用以下属性（名称）和设置（值）进行描述：

Height	495
Left	0
Multiline	False
Name	txtSample
Top	0
Visible	True
Width	215

（完整的文本框包含36个对。）例如，需要加宽该文本框的话，我可以使用鼠标拖拽文本框的右边缘；或者输入一个更大的整数，替换掉215；或者使用运行时赋值的方式：

```
txtSample.Width = 400
```

程序员可以选择最方便的方法来操作这个虽然简单但功能又很强大的结构。

电子表格。记录我们组织的预算对我来说似乎很困难。出于习惯，我为该作业构建了一个大型的程序，用户界面很笨拙。邻近的程序员粗略地看了一下，将该程序实现为一个电子表格，还补充了Visual Basic中的几个函数。整个界面对于会计这样的主要用户来说相当自然。（如果今天要让我编写大学调查程序的话，因为调查结果只不过是一个数字数组，这一事实足以促使我尝试将它放到一个电子表格之中。）

数据库。许多年以前，在文件记录簿中胡乱记录了头十二次跳伞的详细信息之后，一名程序员决心要自动化记录他的高空跳伞记录。在那时的几年以前，这还涉及到如何布局复杂的记录格式，以及手工构建程序以进行数据的输入、更新和检索。当时，他使用一个新奇的商业数据库包来完成该任务，我和他都对此充满敬意：他可以在几分钟而不是几天之内定义全新的数据库操作屏幕。

域特定的语言。图形用户界面（GUI）有幸代替了许多古老而又笨拙的文本语言，但是特殊目的的语言在某些应用场合仍然有用。当实际上我希望直接键入类似下面所示的数学式子时，我会厌恶必须使用鼠标在虚假的计算器上敲来敲去：

```
n = 1000000
47 * n * log(n)/log(2)
```

我不喜欢使用怪模怪样的文本框和运算符按钮的组合来指定某个查询，反而喜欢按照类似下面所示的语言进行编写：

```
(design or architecture) and not building
```

窗口以前是由好几百行可执行代码指定的，现在可以只用几十行超文本标记语言（HTML）来描述。对于一般的用户输入来说，语言可能已不流行了，但是它们在某些

应用场合仍然是强有力的手段。

3.6 原则

尽管本章中所介绍的故事横跨几十年，也涉及到多种语言，但是每个故事的寓意都是一样的：*可用小程序的话就不要编写大程序*。大多数结构都例证了被 Polya 在其《*How to Solve It*》一书中称为发明家的悖论的观点："问题越一般化，解决起来可能也就越容易"。对于编程来说，这就意味着直接解决一个 23 种情况的问题，要比编写一个处理 n 种情况的通用程序，然后将该程序应用到 n=23 时的情况更加困难。

本章只集中介绍了数据结构对软件的一个贡献：将大程序缩减为小程序。数据结构还有其他许多正面的影响，包括时间和空间的缩减、增加可移植性和可维护性。Fred Brook 在其《*Mythical Man Month*》一书的第 9 章中专门对空间缩减进行了评论，但对于同样期望缩减其他属性的程序员来说，这不失为一条不错的忠告：

> *程序员在对空间缺乏无能为力时，往往会脱离代码的纠缠，回过头去凝神考虑他的数据，这样会找到更好的方法。表示法是编程的精华。*

下面介绍你回头思考时需要考虑的几个原则。

将重复性代码改写到数组中。使用最简单的数据结构——数组——来表示一段冗长的相类似的代码往往能达到最佳效果。

封装复杂的结构。当你需要一个复杂的数据结构时，使用抽象的术语对它进行定义，并将那些操作表示成一个类。

尽可能地使用高级工具。超文本、名称-值对、电子表格、数据库、语言以及类似的工具在其专门的问题域内都属于功能相当强大的工具。

让数据去构造程序。本章的主题是使用适当的数据结构去替换复杂的代码，这可以使数据起到构造某个程序的效果。尽管细节更改了，但是主题仍然保留：在编写代码之前，好的程序员通常都会通篇理解构建程序时所围绕的输入数据结构、输出数据结构以及中间数据结构。

3.7 问题

1. 本书第二版将要出版时，美国的个人所得税分 5 个不同的比率，最高的大约是 40%。以前分法更为复杂，比率也更高。有一段编程代码给出了以下 25 个 if 语句，作为一个合理的方法来计算 1978 年美国联邦所得税。比率序列 0.14、0.15、0.16、0.17、0.18、…

序列每次增加0.01。请对此做出评论。

```
if income <= 2200
    tax = 0
else if income <= 2700
    tax =          .14 * (income - 2200)
else if income <= 3200
    tax =    70 + .15 * (income - 2700)
else if income <= 3700
    tax =   145 + .16 * (income - 3200)
else if income <= 4200
    tax =   225 + .17 * (income - 3700)
    ...
else
    tax = 53090 + .70 * (income - 102200)
```

2. 第k阶常系数线性递归定义了一个序列如下所示：

$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + c_{k+1}$,

这里的c_1、…、c_{k+1}是实数。请编写一个程序，输入k、a_1、…、a_k、c_1、…、c_{k+1}和m，输出a_1、…、a_m。与对一个特定的50阶递归进行求值并且不使用数组的程序相比，请问该程序有何难度？

3. 请编写一个“banner（标语）”函数。输入一个大写字母，输出一个字符数组，该字符数组用图形方式描绘该字母。

4. 请编写处理下列日期问题的函数：给定两个日子，计算这两个日子之间的天数；给定某个日子，返回它在一周中属于第几天；给定某年某月，以字符数组的形式产生该月的日历。

5. 本问题将处理一小部分用连字符连接的英语单词方面的问题。下面的规则列表描述了一些以字母c结尾的单词的有效连字符连接：

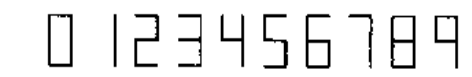
et-ic al-is-tic s-tic p-tic -lyt-ic ot-ic an-tic n-tic c-tic at-ic h-nic n-ic m-ic l-lic b-lic -clic l-ic h-ic f-ic d-ic -bic a-ic -mac i-ac

应用该规则时必须按照上述次序进行；从而导致了连字符连接“eth-nic”（遵循规则“h-nic”）和“clinic”（不满足前一规则，但遵循规则“n-ic”）。如果在某个函数中，给定一个单词，你必须返回后缀连字符连接，那么你该如何表示这样的规则呢？

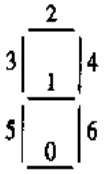
6. 请构建一个“表单字母生成器”，要求该字母生成器可将数据库中的每一条记录都合成到一个定制的文档（这就是通常被称为“邮件合并”的特性）。请设计小型的模式和输入文件，测试一下程序的正确性。

7. 一般的词典都允许人们查阅单词的定义，问题2.1描述了一种允许人们查阅单词变位词的词典。请设计查阅单词正确拼写以及查阅单词押韵的词典。讨论一下查阅一个整数序列（比如1、1、2、3、5、8、13、21、…）、化学结构或者歌曲韵律结构的词典。

8. [S.C. Johnson]七段装置提供了一种廉价的十进制数字显示法：



这七个段通常按以下所示进行编号：



请编写一个程序，要求用 5 个七段数字显示一个 16 位（bit）的正整数。输出结果是一个具有 5 个字节的数组；当且仅当数字 j 的第 i 段打开时，字节 j 中的位 i 才是 1。

3.8 进阶阅读

数据可以构造程序，但是只有聪明的程序员才能构造大型的软件系统。Steve McConnell 所著的《*Code Complete*》由微软出版社于 1993 年出版。其副标题精确描述了这本 860 页的大部头书籍：*A Practical Handbook of Software Construction*。该书快速引导你领略了程序员的智慧。

第 8 章到第 12 章是关于数据的，和本章内容关系最大。McConnell 从诸如声明数据、选择数据名称那样的基础内容一直介绍到包括表驱动的程序以及抽象数据类型那样的高级话题。他的第 4 章到第 7 章是关于设计方面的，详细介绍了本章的主题。

构建一个软件项目所需要的知识跨度很大，从设计小函数雅致大方的结构到管理大型的软件项目。尤其是在将《*Rapid Development*》（微软出版社，1996 年）和《*Software Project Survival Guide*》（微软出版社，1998 年）结合在一起时，McConnell 论及了那两个极端以及这两个极端之间的大部分领域。他的著作阅读时意趣横生，但是你决不会忘记他所讲述的那些来之不易的个人体验。

第4章　编写正确的程序

在20世纪60年代后期，人们一直都在谈论验证其他程序正确性的程序的前途问题。不幸的是，在中间的几十年间，很少出现异议，谈论来谈论去还是自动化验证系统的问题。然而，尽管是不能实现的想法，有关程序验证方面的研究还是给我们提供了一些有价值的东西，这些东西远远不只是一个将程序囫囵吞下然后闪现“好”或者“坏”的黑箱子——现在我们对计算机编程已有一个基本的理解了。

本章的目的在于向你展示这种基本理解如何可以帮助现实中的程序员编写正确的程序。一位读者这样特征化地描述了大多数程序员成长时所伴随的软件编程方法：“编写代码，然后将它抛过墙去，让QA（质量保证）或QT（质量测试）去处理bug问题”。本章描述了一个可选的方法。在开始着手这个主题自身以前，我们必须正确地记住它。编码技能只构成编写正确程序的一小部分。大部分任务是前面三章中的主题：问题定义、算法设计以及数据结构选择。如果那些任务你都完成得比较出色，那么编写正确的代码一般来说也就比较轻松了。

4.1　二分查找的挑战

即使有了最优秀的设计，程序员常常还必须编写精细的代码。本章就是关于一个要求特别小心编码的问题：二分查找。在分析完该问题并勾画出算法之后，我们将使用验证原则编写该程序。

在第2.2节中我们已首次碰到过这个问题了；我们打算确定已排序的数组x[0..n-1]是否包含目标元素t。[1]准确地说，我们知道$n \geqslant 0$，并且$x[0] \leqslant x[1] \leqslant x[2] \leqslant x[3] \leqslant \cdots \leqslant x[n-1]$；当n=0时，该数组是空的。t的类型与x的元素的类型是一样的；伪码对于整数、浮点数或字符串应该都能起作用。答案存储在整数p（位置）中：p=-1时，目标t不在x[0..n-1]中，否则$0 \leqslant p \leqslant n-1$并且t=x[p]。

二分查找法通过记录数组中包含t（假如t在数组中的某个位置）的元素范围的方式

1 如果你在评判这些短变量名称、二分查找函数的定义形式、错误处理以及其他风格问题时需要帮助，请参见问题5.1和答案5.1。上述问题对于成功地完成大型软件项目来说是很关键的。

来解决该问题。最初，这个范围是整个数组。将数组的中间元素和 t 进行比较，去掉该范围的一半以缩小范围。连续进行该过程，直到在该数组中发现 t 为止或直到发现包含 t 的范围不存在时为止。在具有 n 个元素的表中，二分查找大约要进行 $\log_2 n$ 次比较。

大多数程序员都认为，有了上述描述作为指导，编写代码应该容易了。他们错了。若不信，你可立刻放下本章的学习，开始自己编写该代码。你试试看。

我曾经在课程教学中将这个问题布置给专业程序员。学生们花了两个小时的时间按照他们所选的语言将上述描述转化为程序；高级伪码是不会错的。在指定时间结束之时，几乎所有程序员都说他们编写出了完成该任务的正确代码。接着，我们花了三十分钟的时间检查他们的代码，而程序员则使用测试用例对代码进行测试。在若干班级，大约超过 100 名程序员当中，结果几乎都是一样的：百分之九十的程序员在他们的程序中都发现了 bug（并且我还一直怀疑那些没有发现 bug 的代码的正确性）。

我很惊骇：给了那么充裕的时间，只有大约 10%的专业程序员才能够正确地编写出这个小程序。但是他们不是惟一发现这是一道难题的人：历史上，Knuth 在其《*Sorting and Searching*》一书的第 6.2.1 节中指出：尽管第一个二分查找程序于 1946 年就已经公布了，但是第一个没有 bug 的二分查找程序在 1962 年才出现。

4.2 编写程序

二分查找的关键概念在于我们总是知道如果 t 在数组 x[0..n-1]中的某处，那么它必定在 x 的某个范围中。我们使用简写形式 mustbe（range）表示如果 t 在数组中，那么它必定在 range 中。我们可以使用这种表示法将上述有关二分查找的描述转换为程序草图。

```
initialize range to 0..n-1
loop
    { invariant: mustbe(range) }
    if range is empty,
        break and report that t is not in the array
    compute m, the middle of the range
    use m as a probe to shrink the range
        if t is found during the shrinking process,
        break and report its position
```

本程序的关键部分就是 loop invariant，即用花括号括起来的部分。有关程序状态的这个断言（assertion）被称为不变式（invariant），因为每一次循环迭代的开始和结尾它都是真值（true）；这就形式化了我们上述的直观表示。

现在我们将对该程序进行精化，以确保所有的动作都和这个不变式相关联。我们必须面对的第一个问题就是 range 的表示问题：我们将使用两个下标 l 和 u（分别表示下限

和上限）来表示范围 l..u（第 9.3 节中的二分查找函数用起始位置和长度表示某一范围）。逻辑函数 mustbe（l,u）表示：如果 t 在数组中的某处，它必定在（封闭的）范围 x[l..u]内。

我们的下一步工作就是初始化；l 和 u 应该选什么值才能保证 mustbe（l,u）是真呢？很明显，应该选择 0 和 n-1：mustbe（0,n-1）表示如果 t 在 x 中，那么它必定也在 x[0..n-1]中，这正是我们在程序一开始就了解到的。因此，初始化过程就是分别对 l 和 u 赋值 0 和 n-1 的过程：l=0；u=n-1。

下一个任务就是检查范围是不是空的，并计算新的中间点 m。如果 l>u，则说范围 l..u 是空的。这时我们将特殊值-1 存储到 p 中，同时终止循环，也即：

```
if l > u
    p = -1; break
```

break 语句将终止封闭循环。下面的语句将计算范围中间点 m：

```
m = (l + u) / 2
```

"/"运算符实现整数除法：6/2 等于 3，和 7/2 是一样的。下面就是我们已经进行扩展了的程序：

```
l = 0; u = n-1
loop
    { invariant: mustbe(l, u) }
    if l > u
        p = -1; break
    m = (l + u) / 2
    use m as a probe to shrink the range l..u
        if t is found during the shrinking process,
        break and note its position
```

下一步就是对循环体中的最后三行进行细化，这将涉及到对 t 和 x[m]进行比较，并采取适当的动作保持该不变式。这样该代码就具有以下所示的一般形式：

```
case
    x[m] <  t:  action a
    x[m] == t:  action b
    x[m] >  t:  action c
```

对于动作 b，因为我们已经知道 t 的位置是 m，所以我们将 p 设置为 m 并终止循环。因为其他两种情况是对称的，所以我们将集中介绍第一种情况，并且我们相信后一种情况将可按照对称的方式得到（这也是我们将在下一节中精确地验证该代码的部分缘由）。

我们知道，如果 x[m]<t，那么 x[0]≤x[1]≤…≤x[m]<t，所以 t 不可能在 x[0..m]中。另外我们还知道 t 也不可能在 x[l..u]的外面，如果 t 在 x 中某处的话，那它必定在 x[m+1..u]中，我们将之写为 mustbe(m+1,u)。然后我们将 l 设置为 m+1，重新设立不变式 mustbe(l,u)。

将这些情况插入到前面的代码草图中就得到了最终的函数。

```
l = 0; u = n-1
loop
    { mustbe(l, u) }
    if l > u
        p = -1; break
    m = (l + u) / 2
    case
        x[m] <  t:  l = m+1
        x[m] == t:  p = m; break
        x[m] >  t:  u = m-1
```

这是一个短程序，包括10多行的代码和一个不变式断言。程序验证的基本技术是先精确指定该不变式，并在我们编写每一行代码时密切关注以保持该不变式。在我们将算法草图转换成伪码时，这种技术对我们帮助极大。这个过程多少使我们对程序有了一点信心。但我们决不能因此而确信程序正确无误。在继续往下阅读之前，请花几分钟的时间确认一下该代码的表现是否与所要求的一致。

4.3 理解程序

在面对微妙的编程问题时，如我们刚看到的那样，我通常会尝试在细节层面上得出代码。接着我会使用验证方法来增强我对程序正确性的信心。在第9、11和14章中，我们都将在此层面使用验证。

在本节中，我们将以吹毛求疵的态度在细节层面上对二分查找代码的验证进行学习和讨论，实际上在工作中我所做的分析要比这少得多。下一页的程序（极）大量地使用了断言注释，从而形式化了最初编写代码时所使用的直观表示。

尽管代码的开发是自顶向下的(先从一般概念开始,然后再细化到一行一行的代码)，但是这里的正确性分析是自底向上的：我们将从个别的代码行开始，研究它们如何一起协作以解决该问题。

> **警告**
>
> 下面是一些单调沉闷的内容。如果你觉得昏昏欲睡的话，请跳转到第4.4节。

我们将从第1行～第3行开始。第一行的断言是，按照mustbe的定义，mustbe(0,n-1)是真的：如果t在数组中的某处，那么它必定也在x[0..n-1]中。第2行中有两个赋值语

句：l=0 以及 u=n-1，这就给第三行代码提供了断言：mustbe(l,u)。

现在我们来分析一下难点部分：第 4 行到第 27 行中的循环。我们对其正确性的讨论分三部分进行，每一部分都与循环不变式紧密相关：

初始化。循环首次执行时，不变式为真。

保存。如果不变式在迭代一开始就保存并且循环体已执行，那么等循环体完成执行之后，该不变式仍将保持为真。

终止。循环将终止，并保存预期的结果（在这种情况下，预期结果即为 p 保存了正确的值）。显示这一点将用到不变式所确立的事实根据。

对于初始化来说，我们注意到，第 3 行中的断言和第 5 行是一样的。为了确立其他两个属性，我们将从第 5 行开始一直推究到第 27 行。讨论第 9 行和第 21 行（break 语句）时，我们将确立终止属性；如果我们沿路推究到第 27 行，因为这一行和第 5 行是一样的，所以我们将确立保存。

```
 1.  { mustbe(0, n-1) }
 2.  l = 0; u = n-1
 3.  { mustbe(l, u) }
 4.  loop
 5.      { mustbe(l, u) }
 6.      if l > u
 7.          { l > u && mustbe(l, u) }
 8.          { t is not in the array }
 9.          p = -1; break
10.      { mustbe(l, u) && l <= u }
11.      m = (l + u) / 2
12.      { mustbe(l, u) && l <= m <= u }
13.      case
14.          x[m] <  t:
15.                  { mustbe(l, u) && cantbe(0, m) }
16.                  { mustbe(m+1, u) }
17.                  l = m+1
18.                  { mustbe(l, u) }
19.          x[m] == t:
20.                  { x[m] == t }
21.                  p = m; break
22.          x[m] >  t:
23.                  { mustbe(l, u) && cantbe(m, n) }
24.                  { mustbe(l, m-1) }
25.                  u = m-1
26.                  { mustbe(l, u) }
27.      { mustbe(l, u) }
```

如果第 6 行测试成功的话，那么将得到第 7 行的断言：如果 t 在数组中的某处，那么它必定在位置 l 和 u 之间，并且 l>u。这些事实也就暗示了第 8 行的成立：t 不在数组中。这样我们就可以在设置 p 为-1 之后，在第 9 行正确地终止该循环。

如果第6行测试失败，我们将转到第10行。不变式仍然保持原样（我们没有更改它），并且因为测试失败了，我们知道l≤u。第11行将m设置为小于或等于l与u的平均值的最大整数。因为平均值始终在两个值之间，所以m绝不可能小于l，于是我们得到第12行中的断言。

第13行到第27行是对case语句进行分析，这里仔细考虑了三种可能选择中的每一种选择。其中最简单的就是第19行的第二个case可选项。由于第20行断言的原因，我们正确地将p设置为m，并且终止循环。这是循环终止所在的两个位置中的一个，两个位置都正确地终止了循环，所以我们确立了循环的终止正确性。

下一步我们来分析一下两个对称的case语句分支；因为开发代码时我们集中介绍了第一个分支，现在我们将主要精力转向第22行到第26行的代码。请仔细研究一下第23行中的断言。第一个子句是不变式，循环还没有更改此不变式。因为t<x[m]≤x[m+1] ≤... ≤x [n-1]，因而第二个子句是真的，所以我们知道t在数组中不可能位于大于m-1的某个位置：这可以表示成简写形式cantbe(m,n-1)。

从逻辑上来说，如果t必定在l和u之间，并且不是m或在m之上，那么它必定在l和m-1之间（如果t在x中的某处）；因此得到第24行。如果第24行是真的话，执行第25行将导致第26行也是真的，这就是赋值的定义。因此这个case语句分支在第27行重新确立了不变式。

第14行到第18行中的讨论在形式上是一样的，至此我们将case语句中的三种可能的选择全部分析完了。其中一个恰当地终止了循环，另两个维持了该不变式。

对这段代码的上述分析表明：循环终止时它将在p中设置一个正确的值。然而，仍然可能出现无限循环；实际上，专业程序员在编写程序时也会犯这种常见的错误。

我们能够终止循环在于我们使用了可变范围 l..u。该范围最初设置为某一有限大小（n），第6行到第9行确保了范围内不包含任何元素时循环将终止。因此为了验证循环会终止，我们必须表明每一次循环迭代之后范围都会逐渐缩小。第12行告诉我们m始终在当前范围之内。循环到两个case语句分支（第14行和第22行）时将把m位置的值从当前范围中排除掉，因而范围的大小至少减少了1。因此该程序必将终止。

有了这些背景知识，我对我们可以进一步学习这个函数就相当有信心了。下一章恰好涉及到该主题：用C语言实现该函数，然后对其进行测试，确保该函数的正确性和有效性。

4.4 原则

该练习显示了程序验证的诸多长处：问题是重要的；要求仔细为其编写代码；程序开发要在验证观念的指导下进行；利用一般的工具即可进行正确性分析。该练习的主要

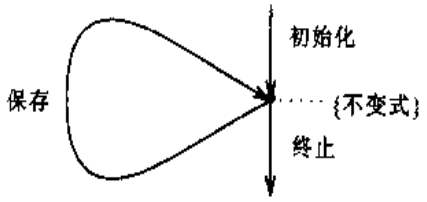
弱点在于它处于细节层面；在实际工作中，我是在一个更不正式的层面上进行的。幸运的是，这些细节阐明了大量一般性的原理，例如：

断言。输入、程序变量以及输出之间的关系描述了程序的状态；断言允许程序员精确地说明这些关系。它们在整个程序生存期中的任务将在下一节中进行讨论。

顺序控制结构。控制程序最简单的结构就是对语句采取逐条执行的形式。我们可在各个语句之间插入一些断言，并单独分析每一步程序过程，以理解这一类结构。

选择控制结构。这类结构包含各种形式的 if 和 case 语句；在执行期间，可以在多个选项之间选择一个选项继续执行。我们对若干选项中的每一选项都分别进行考虑，以显示这一类结构的正确性。事实上，选择特定选项允许我们将某个断言作为证明。例如，如果我们执行 if　i>f 后的语句，我们就可以断言 i>f，并使用这个事实衍生出下一个相关的断言。

迭代控制结构。为了证明循环的正确性，我们必须确立三个属性：



我们先证明初始化过程确立了循环不变式，然后再说明每一次迭代都保存其真值。这两步采用数学归纳法的形式证明了不变式在每一次循环迭代前和迭代后都是真的。第三步证明了无论循环执行在什么时候终止，预期结果都是真值。将这几步合在一起即证实了无论循环何时终止，都能以恰当的方式终止。我们必须证明它是通过其他方法终止的（二分查找的终止证明使用了典型的三段论中的小前提法）。

函数。为了验证一个函数，我们首先需要通过两个断言陈述其目的。它的*前置条件*就是在调用之前必须保持为真；其*后置条件*就是函数要保证终止。这样我们就可以如下所示说明一个用 C 语言编写的二分查找函数：

```
int bsearch(int t, int x[], int n)
 /* precondition: x[0] <= x[1] <= ... <= x[n-1]
    postcondition:
        result == -1     =>  t not present in x
        0 <= result < n  =>  x[result] == t
  */
```

这些条件与其说是事实的描述，还不如说是契约：它们假定如果调用该函数时满足前置条件的话，那么函数的执行将确立其后置条件。一旦证明了函数体具有此属性，我就可以使用前置条件和后置条件之间的这种陈述关系，而不需要再次考虑实现问题。这种软件开发方法经常称为“契约编程”。

4.5 程序验证的任务

某一程序员试图让另一名程序员确信某段代码是正确无误时，所使用的主要工具通常都是测试用例：手工输入然后执行该程序。那是一个功能强大的工具：它适用于检测bug，使用简单，也好理解。然而，这样做很显然需要程序员对程序有很深的理解。如果对程序没有很好的理解，那他们决不能先行编写出程序来。程序验证的好处之一就是它为程序员提供了一门语言，程序员可以用它来表达对程序的理解。

在本书的后面，尤其是第9、11和14章，在开发各种精细的程序时，我们将应用到验证技术。编写代码时，我们将使用验证语言来解释每一行代码；在勾画每一个循环的不变式时验证技术尤其有帮助。程序文本结尾的重要解释将作为断言；确定现实世界中的软件应该包含什么样的断言是一门艺术，这门艺术只能从实践中学来。

验证语言经常用在代码第一次编写之后，并且在代码走查期间开始执行。测试期间，如果违反了断言语句的话，那将指明出现bug了；对违规形式进行分析你就知道该如何排除某一bug而不会再引入另一个bug。请注意，当你在调试、修改代码或错误的断言语句时，要完全地理解代码，抵御那种“改变代码，只要能让它运行起来就行”的冲动。下一章讲述了断言在程序测试和调试中扮演的几种角色。断言在程序维护期间很关键：当你捡起以前从没有看过，而且几年来也没有其他人看过的代码时，程序状态断言对我们理解程序很有帮助。

这些技术仅仅是编写正确程序的一小部分；保持代码的简单性通常是正确性的关键。另一方面，有些熟悉这些技术的专业程序员给我的感觉是：当他们构建一个程序时，难的部分通常先通过，而错误往往就在容易的部分。这在我自己的编程经历中太常见了。当他们碰到难的部分时他们就会耐下心来，成功地运用那些功能强大的正规技术。而在那些容易的部分，他们往往会回退到古老的编程方式中，因而得到的结果也是老的。以前我还没有碰到过这种现象，自己碰到过之后才相信这种现象。这种难堪的现象也是对经常使用这些正规技术的一个良好触动。

4.6 问题

1. 二分查找证明起来很费劲，按照某些标准来说，它仍然未完成。你如何可以证明程序不会出现运行时错误呢（比如0作除数、字溢出、变量超出声明范围或者数组下标超出界限）？如果你懂得一点离散数学方面的背景知识的话，你可以将证据形式化到一个逻辑系统中吗？

2. 如果最初的二分查找对你来说太简单了，那么请你试一下其变型：在 p 中返回 t 在数组 x 中第一次出现时的位置（如果 t 在数组中多次出现的话，原先的算法所返回的是众多位置中的任意一个）。你的代码应该对数组元素进行对数次比较，可能要进行 $\log_2 n$ 次这样的比较才能完成此二分查找。

3. 编写并验证递归二分查找程序。请指出代码和证明的哪一部分与迭代版本的程序相同，哪一部分不相同？

4. 请往你的二分查找程序中添加虚拟的“计时变量”，以便计数比较次数；并使用程序验证技术证明其运行时间确实是成对数的。

5. 请证明当输入 x 是正整数时，此程序将终止。

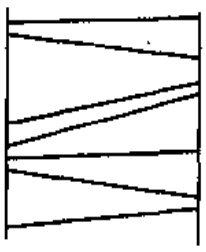
```
while x != 1 do
    if even(x)
        x = x/2
    else
        x = 3*x+1
```

6. [C. Scholten]David Gries 在他所著的《*The Science of Programming*》中将这称为“咖啡罐问题”。起初给你提供一个盛装了一些黑豆和白豆的咖啡罐以及一大堆额外的黑豆。然后你重复进行以下过程，直到罐中只剩下一粒豆子时为止：

随机从罐中选择两粒豆子。如果它们的颜色一样，就将它们都扔掉，并且在罐中放入一颗额外的黑豆。如果它们的颜色不同，则将白豆返回罐中，同时扔掉黑豆。

请证明该过程会终止。当一开始罐子里既有黑豆又有白豆时，你能说出罐子里最后剩下的豆子是什么颜色的吗（表示成黑豆数和白豆数的函数）？

7. 一位同事在编写一个在点阵显示器上绘制直线的程序时碰到了下面的问题。n 对实数（a_i，b_i）定义了 n 条直线 $y_i=a_ix+b_i$。对于所有在 0 到 n-2 之间的 i 值并且 x 的所有值在[0,1]内时，则这些直线以 $y_i<y_{i+1}$ 的有序方式排列在 x 闭区间[0,1]中：



如果条件更不严格的话，这些直线不会和垂直线接触。给定一个点(x,y)，当 $0\leqslant x\leqslant 1$ 时，他希望确定包含这个点的两条直线。如何可以快速地解决这个问题呢？

8. 二分查找基本上比顺序查找更快一些：如果要在一具有 n 个元素的表中进行查找的话，二分查找粗略说来需要进行 $\log_2 n$ 次比较，而顺序查找粗略地说需要进行 n/2 次。虽然二分查找通常已经够快了，在少数情况下，查找还必须更快一些。尽管在算法上你

不能减少比较的逻辑次数，但是你能否重写二分查找代码，使它更快些吗？为明确起见，假设你需要在一个具有1000个整数的排序表中进行查找。

9. 作为一个程序验证的练习，请精确指定下列每一个程序片断的输入/输出行为，并指出满足这些说明的代码。第一个程序实现向量的加法运算：a=b+c。

```
i = 0
while i < n
    a[i] = b[i] + c[i]
    i = i+1
```

（这段代码和下面两个片断将“for i=[0,n)”循环展开为末尾使用加1运算的while循环）。下一片断将计算数组x中的最大值。

```
max = x[0]
i = 1
while i < n do
    if x[i] > max
        max = x[i]
    i = i+1
```

下面的顺序查找程序将返回t在数组x[0..n-1]中第一次出现的位置。

```
i = 0
while i < n && x[i] != t
    i = i+1
if i >= n
    p = -1
else
    p = i
```

下面这个程序将以与n的对数成比例的时间计算出x的n次幂。这个递归程序将直接进行编码与验证；迭代版本更复杂些，留作附加问题。

```
function exp(x, n)
        pre  n >= 0
        post result = x^n
    if n = 0
        return 1
    else if even(n)
        return square(exp(x, n/2))
    else
        return x*exp(x, n-1)
```

10. 在二分查找函数中引入一个错误，并通过验证这些包含错误的代码来看看这些错误是否及如何被查出来的？

11. 使用C或C++语言编写下面递归二分查找，并证明其正确性：

```
int binarysearch(DataType x[], int n)
```

请单独使用这个函数，不要调用任何其他的递归函数。

4.7　进阶阅读

David Gries 所著的《*The Science of Programming*》是一本在编程验证方面相当出色的入门书。这本书在 1987 年由 Springer-Verlag 出版社以平装本出版。它先从逻辑开始说起，继而对程序验证和开发进行了正规介绍，最后讨论了常见语言的编程问题。在本章中我已设法勾画了程序验证的好处；大多数程序员可以有效使用验证技术的惟一方法就是学习诸如 Gries 所著的那样的书籍。

第5章　编程中的次要问题

迄今为止，一切进展尚为顺利。你进行深入地发掘，定义了正确的问题；小心地选择了算法和数据结构，权衡了真正的需求，还使用程序验证技术编写出了一流的伪码（这一点你有百分之百的把握确信它是正确无误的）。那么你怎么将得到的这些“珍品”合并到你的大系统中呢？好，剩下的都是编程中的小问题了。

程序员都是乐观主义者，他们倾向于采取简单的途径：编写出函数代码；然后将它插入到系统中，强烈希望它能运行。这有时能够奏效。但是1000次中有其他的999次会导致灾难的发生：人们不得不在巨型系统中到处摸索，而目的仅仅是为了操作这个小小的函数。

聪明的程序员会先构建脚手架，从而为访问函数提供更加容易的方法。本章主要致力于将上一章中用伪码描述的二分查找实现为一个可靠的C语言函数（该代码非常类似于C++或Java的实现，该方法对大多数其他的语言都适合）。一旦有了代码，我们就可以使用脚手架对它进行探测了，然后继续对它进行更加全面的测试并在运行时进行试验。对于极小的函数来说，这个过程算是冗长了。尽管如此，这样做得到的结果却是我们可以信赖该程序。

5.1　从伪码到C语言

我们假定数组x和目标项t都属于类型DataType。DataType由C语言中typedef语句定义，如下所示：

```
typedef int DataType;
```

定义的类型可以是长整型、浮点型，或任何其他类型。数组可通过两个全局变量实现：

```
int n;
DataType x[MAXN];
```

（尽管对于大多数的C语言程序来说，这样的风格比较低劣，但它反映了我们访问C++类中数据的方法；全局变量也可以使用更小的脚手架。）我们的目标就是下面的C语言

函数：

```
int binarysearch(DataType t)
 /* precondition: x[0] <= x[1] <= ... <= x[n-1]
    postcondition:
        result == -1     =>  t not present in x
        0 <= result < n  =>  x[result] == t
  */
```

第4.2节中的大多数伪码语句都可以逐行转换成C语言语句（还可转换成其他大多数语言）。在伪码将值保存在答案变量 p 中的任何位置，C 程序都将返回该值。使用 C 语言中的无限循环 for(;;)替换伪码循环得到下面的代码：

```
for (;;) {
    if (l > u)
        return -1;
    ... rest of loop ...
}
```

倒转一下测试我们可以将之转换为 while 循环：

```
while (l <= u) {
    ... rest of loop ...
}
return -1;
```

最终的程序如下所示：

```
int binarysearch(DataType t)
    /* return (any) position if t in sorted x[0..n-1] or
        -1 if t is not present */
{   int l, u, m;
    l = 0;
    u = n-1;
    while (l <= u) {
        m = (l + u) / 2;
        if (x[m] < t)
            l = m+1;
        else if (x[m] == t)
            return m;
        else /* x[m] > t */
            u = m-1;
    }
    return -1;
}
```

5.2 测试装备

运用该函数的第一步当然就是手工走查少量的测试用例。通常极小的用例（具有 0

个、1个、2个元素的数组）就足以查出bug。对于更大些的数组，这一类测试就会变得冗长乏味了，所以下一步就要构建一个调用该函数的测试驱动程序了。下面5行的C语言脚手架可以完成该任务：

```
while (scanf("%d %d", &n, &t) != EOF) {
    for (i = 0; i < n; i++)
        x[i] = 10*i;
    printf(" %d\n", binarysearch(t));
}
```

我们可以在开始先测试一个大约有24行代码的C程序：binarysearch函数以及main函数中的上述代码。随着附加脚手架的添加，程序会不断增长。

当我键入输入行“2 0”时，该程序即生成具有两个元素的数组，其中x[0]=0,x[1]=10，然后报告（在下一个预期行中）查找0的结果是其位置为0：

```
2 0
 0
2 10
 1
2 -5
 -1
2 5
 -1
2 15
 -1
```

在上述代码中键入的输入始终以斜体字的形式出现。下两行显示已正确定位10到位置1了。最后6行描述了三次标准的不成功查找。这样该程序恰当地处理了具有两个不同元素的数组的所有可能情形。随着程序通过不同规模输入的类似测试，我对程序的正确性越来越自信了，而对这种艰辛的手工测试也越来越厌烦了。下一节将描述自动化此项工作的脚手架。

不是所有的测试都能顺利完成。下面是由若干专业程序员提议的二分查找：

```
int badsearch(DataType t)
{   int l, u, m;
    l = 0;
    u = n-1;
    while (l <= u) {
        m = (l + u) / 2;
/* printf("  %d %d %d\n", l, m, u); */
        if (x[m] < t)
            l = m;
        else if (x[m] > t)
            u = m;
        else
            return m;
```

```
        }
        return -1;
    }
```

（我们不久将讨论到注释掉的 printf 语句。）你能发现代码中有什么问题吗？

该程序通过了前两次小的测试。在具有 5 个元素的数组中，它在位置 2 处找到了 20，位置 3 处找到了 30：

```
5 20
 2
5 30
 3
5 40
 ...
```

尽管如此，当我查找 40 时，该程序就进入了无限循环。为什么？

为了解决这个问题，我在上面插入了一个 printf 语句作为注释（该语句向左缩排，以显示它是脚手架）。它显示了每次查找时 l、m 和 u 值的顺序：

```
5 20
  0 2 4
 2
5 30
  0 2 4
  2 3 4
 3
5 40
  0 2 4
  2 3 4
  3 3 4
  3 3 4
  3 3 4
  ...
```

第一次查找时首次探测就发现了 20，第二次查找是在第二次探测时才发现 30。第三次查找时发现前两次探测都很好，但是第三次探测就开始进入无限循环了。当我们设法证明循环的终止性（实际上是徒劳无益的）时，我们就应该发现那个 bug 了。

当不得不调试一个深深嵌入到一个大型程序中的小算法时，我有时会使用诸如单步调试那样的调试工具来调试该大型程序。但是，当我像上面那样使用脚手架调试一个算法时，使用 printf 语句通常实现要更快一些，比起复杂调试工具来说也要更加有效一些。

5.3 断言的艺术

在我们开发第 4 章中的二分查找时，断言起了几个关键的作用：它们引导我们开发代码；允许我们对其正确性进行求证。现在我们将把它们插入到代码中，确保运行时的

表现符合我们的理解。

我们将使用断言来陈述我们相信某个逻辑表达式是正确的。如果 n 大于或等于 0，语句 assert（n≥0）将不做任何事情，但是如果 n 是负数的话，它将引发某些错误（多半是调用某个调试器了）。在报告二分查找发现目标之前，我们可能会作如下断言：

```
    ...
else if (x[m] == t) {
    assert(x[m] == t);
    return m;
} else
    ...
```

这个微弱的断言只是重复 if 语句中的条件。我们可能希望对它进行增强，断言该返回值在输入范围内：

```
assert(0 <= m && m < n && x[m] == t);
```

当循环结束而又没有找到目标值时，我们了解到 l 和 u 已经交错，因此我们也就知道该元素不在数组中了。我们可能极想断言我们发现了一对相邻的包含目标 t 的元素：

```
assert(x[u] < t && x[u+1] > t);
return -1;
```

从逻辑上来说，如果我们在已排序表中发现了 1 和 3 是两个相邻的元素，那么我们可以确信 2 不在此表中了。尽管如此，此断言有时即使对一个正确的程序来说也会失败，为什么？

当 n=0 时，变量 u 初始化为-1，这时下标会访问数组外面的元素。因此，为了确保断言的实用性，我们必须通过边界测试，对其进行弱化：

```
assert((u < 0 || x[u] < t) && (u+1 >= n || x[u+1] > t));
```

此语句在一些不完善的查找中确实会发现某些 bug。

每次迭代时范围都会减小，这就证明了该查找一定会停止。我们可以使用少量附加的计算和断言在执行期间检验一下该属性。我们将 size 初始化为 n+1，接着在 for 语句之后插入以下代码：

```
oldsize = size;
size = u - l + 1;
assert(size < oldsize);
```

如果让我公开承认只是因为待查找的数组未排序，就使得我尝试很多次二分查找都不成功，那么我会感到无地自容的。一旦我们定义了该函数：

```
int sorted()
{   int i;
    for (i = 0; i < n-1; i++)
```

```
        if (x[i] > x[i+1])
            return 0;
    return 1;
}
```

我们就可以断言 assert(sorted())。尽管如此，我们还是应该小心地在所有查找之前进行一次这种昂贵的测试。在主循环中包含测试可能会导致二分查找的运行时间和 n log n 相当。

当我们在脚手架中测试函数，以及从组件测试移到系统测试中时，断言是很有帮助的。有些项目使用预处理器定义断言，所以断言可以分开编译，不会引发运行时开销。另一方面，Tony Hoare 曾经观察到，程序员测试时使用断言，而在生产期间将它们关闭，这就像水手在岸上训练时穿上救生衣，在海里面却将其脱下。

Steve Maguire 所著的《*Writing Solid Code*》（微软出版社 1993 年出版）一书的第 2 章就致力于研究断言在工业软件中的使用。他详细描述了在微软的各个产品和库中有关断言使用方面的几个纷争。

5.4 自动化测试

你摆弄这个程序已经足够多了，可以相当确信地说它已经是正确的了；你也已经厌烦手工传递测试用例给它了。下一步就是构建脚手架，对它进行自动“攻击”。测试函数的主循环从最小的可能值（0）开始一直到运行最大合理值 n：

```
for n = [0, maxn]
    print "n=", n
    /* test value n */
```

print 语句将报告测试进展。有些程序员讨厌这个语句：它产生一些混乱的而不是实质性的信息。另一些人则在观看测试进程中寻找慰藉，当你发现第一个 bug 时，它对于你了解已经通过了哪些测试可能会有用。

测试循环的第 1 部分就是分析用例，要求所有元素都互不相同（它还要在数组顶部放置一个多余的元素，以确保查找不会定位它）。

```
/* test distinct elements (plus one at the end) */
for i = [0, n]
    x[i] = 10*i
for i = [0, n)
    assert(s(10*i)     == i)
    assert(s(10*i - 5) == -1)
assert(s(10*n - 5) == -1)
assert(s(10*n)     == -1)
```

为了使测试各种不同的函数更加简单，我们定义了下面待测试的函数：

```
#define s binarysearch
```

下面这些断言将测试每一个可能的位置，不管是成功的查找，还是不成功的查找；同时也测试元素在数组中但同时又在查找边界之外的例子。

测试循环的下一部分将探测相等元素数组：

```
/* test equal elements */
for i = [0, n)
    x[i] = 10
if n == 0
    assert(s(10) == -1)
else
    assert(0 <= s(10) && s(10) < n)
assert(s(5) == -1)
assert(s(15) == -1)
```

这将查找在数组中某处的元素，以及小于或大于该元素的元素。

这些测试将在大部分程序中进行。将 n 从 0 到 100 取值进行测试覆盖了空数组、bug（0、1 和 2）的常见大小、2 的几次幂，以及 2 的幂减去几个数。手动进行这些测试的话将非常烦人（并且因此容易出错），但是它们只使用微不足道的计算机时间。如果将 maxn 设置为 1000，那么在我的机器中，测试时间将用不了几秒钟。

5.5 计时

在进行广泛的测试之后我越来越相信该查找是正确的。我们如何可以类似地举出新证据，使我们相信完成该工作，大约需要进行 $\log_2 n$ 次比较？下面是计时脚手架的主循环：

```
while read(algnum, n, numtests)
    for i = [0, n)
        x[i] = i
    starttime = clock()
    for testnum = [0, numtests)
        for i = [0, n)
            switch (algnum)
                case 1: assert(binarysearch1(i) == i)
                case 2: assert(binarysearch2(i) == i)
    clicks = clock() - starttime
    print algnum, n, numtests, clicks,
        clicks/(1e9 * CLOCKS_PER_SEC * n * numtests)
```

这段代码计算了在具有 n 个互不相同的元素的数组中成功完成一次二分查找时所需要的平均运行时间。它首先初始化该数组，然后针对数组中的每个元素一共执行 numtests 次查找。switch 语句选择待测试的算法（脚手架应该始终准备着测试几个变量）。print 语句汇报三个输入值和两个输出值：clicks 的原始数字（对这些值进行检验无论什么时候

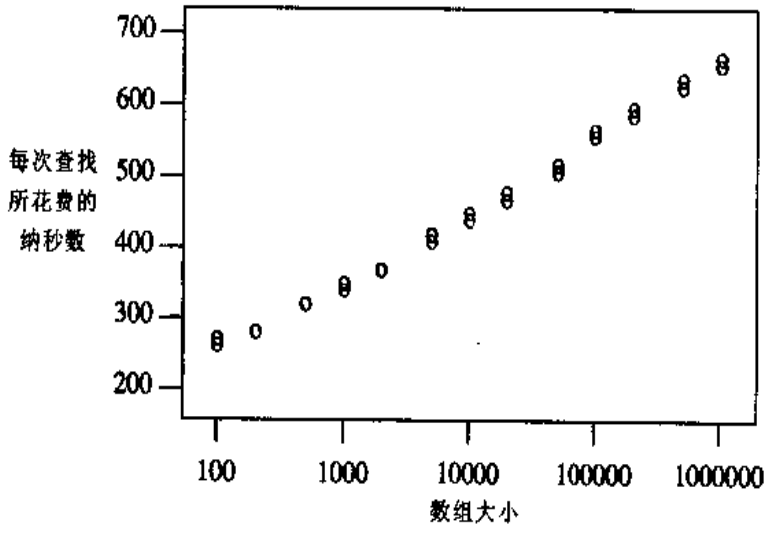
都很关键），以及一个更加易于解释的值（在本例中，是指以纳秒计时的查找的平均成本。这个成本由 print 语句中的转换因子 1E9 给出）。

下面是与 400MHz 奔腾 II 中的程序进行的会话，与往常一样，斜体表示输入：

```
1 1000 10000
1       1000    10000   3445    344.5
1 10000 1000
1       10000   1000    4436    443.6
1 100000 100
1       100000  100     5658    565.8
1 1000000 10
1       1000000 10      6619    661.9
```

第一行在大小为 1000 的数组上测试算法 1（也即我们目前为止已研究过的 binarysearch），并且执行 10000 次测试。它花了 3445 个时钟节拍（在本系统中是以毫秒进行汇报的），转换之后得到每次查找的平均开销约为 344.5 纳秒。后续的三次实验中，每次都将 n 增加 10 倍，而测试的次数则减少 10 倍。每次查找的运行时间看来大约是 $50+30\log_2 n$ 纳秒。

接下来我编写了三行程序，以生成计时脚手架的输出。输出被打点在图上，该图显示，平均查找成本确实以 log n 增长的方式而增加。问题 7 对本脚手架中的一个潜在的计时 bug 作了评述：在充分信任这一类数字之前请一定要研究一下这个问题。



5.6 完整的程序

现在我相信这个二分查找的 C 语言实现是正确的。为什么？因为我利用方便的语言，很小心地得出了伪码，然后又使用分析技术验证了其正确性。我将其逐行转换为 C 语言程序，给出程序输入并观察输出。我在整个代码中布设了断言，确保我的理论分析和实际表现相吻合。计算机完成它所擅长的任务，使用各个测试用例去检验该程序。最后，

简单的实验表明其运行时间和理论预测的一样低。

有了这些背景知识，我就会对在大程序中使用该代码来搜索一个排序数组感到心安理得一些了。如果这段C代码出现了一个逻辑bug，那么我会相当惊讶的。但是如果发现大量其他类型的bug，我则不会感到震惊。调用者记得对表进行排序吗？查找项不在表中时预期的返回值是-1吗？如果查找目标项在表中出现多次的话，这段代码将随意返回一个下标；用户真的希望返回第一个或最后一个下标吗？诸如此类等等。

你能信任这段代码吗？你可以相信我的话（如果你愿意相信我的话，现在我有一个你可能愿意购买的“桥梁”）。另外，你还可以从本书的站点复制该程序。它包含我们迄今为止所看到的所有函数，还包含我们将在第9章中研究的几个二分查找的变体。它的主函数看起来类似下面的样子：

```
int main(void)
{   /* probe1(); */
    /* test(25); */
    timedriver();
    return 0;
}
```

注释掉除一个调用之外的所有函数之后，你可以大致考虑一些特定的输入，使用测试用例对其调试，或运行计时试验。

5.7 原则

本章投入了大量的精力去解决一个小问题。该问题可能小些，但是不太容易。请回想一下第4.1节所说的，尽管第一个二分查找程序于1946年就已公布了，但对所有n值都运行正常的第一个二分查找程序直到1962年才出现。如果以前的程序员采用了本章所叙述的方法的话，得出一个正确的二分查找程序可能就用不着花16年的时间了。

脚手架。最好的脚手架通常就是那种最容易构建的脚手架。对于某些任务来说，最简单的脚手架由一个图形用户界面组成，实现它的语言有Visual Basic、Java或Tcl等。在其中的每一种语言中，我已经采用point-and-click控件和别致的可视化输出，花了半小时来测试那些小程序。但是对于许多算法任务而言，我发现摒弃那些功能强大的工具，使用本章我们所看到的那些更简单的（移植性更好的）命令行技术或许要更好一些。

编码。我发现对于高难度的函数来说，最简单的方法就是先使用便利的高级伪码为其描绘骨架，然后将之转换成实现语言。

测试。测试组件时，在脚手架中进行测试要比在一个大系统中简单和全面得多。

调试。程序在脚手架中隔离时调试起来会很困难，并且在把它嵌入到真实环境中时

调试甚至会更加困难。第 5.10 节讲述了有关调试大系统的几个故事。

计时。如果对运行时间的要求不高的话，采用线性查找要比采用二分查找简单得多。编写线性查找程序时，许多程序员都是一次成功的。因为运行时间对于我们引入二分查找的附加复杂性来说是很重要的，所以我们应该进行试验，确保得到我们预期的性能。

5.8 问题

1. 请总体评述一下本章、本书的编程风格。解决诸如变量名称、二分查找函数的格式和规格说明、代码布局等方面的风格问题。

2. 请将二分查找的伪码转换成 C 语言以外的其他语言，并构建脚手架测试和调试你的实现。语言和系统方面对此有什么帮助和妨碍？

3. 请在二分查找函数中引入错误。测试过程是如何捕获错误的？脚手架如何帮助你找出 bug？（这个练习最好作为一个双人游戏来完成，袭击者引入 bug，然后防卫者必须追踪。）

4. 重复问题 3，但这次保持二分查找代码的正确，在调用它的函数中引入错误（比如忘记排序数组）。

5. [R. S. Cox] 常见的 bug 就是将二分查找应用于未排序的数组。在每一次查找之前对数组是否已排序进行完整检测要花费 n-1 次额外比较的高昂代价，你如何在函数中添加部分检测，以显著地降低成本呢？

6. 请在研究二分查找时实现图形用户界面。为增加调试有效性而导致增加开发时间，这合算吗？

7. 第 5.5 节中的计时脚手架有一个潜在的计时 bug：通过依次查找每个元素的方式，我们得到了非常有利的缓存行为。如果我们知道潜在应用中的查找会展现类似的位置，那么这就是一个精确的框架（但是二分查找也许不是适合于该任务的工具）。但是如果我们期望那些查找法能随机探测数组，那么我们或许还应该初始化并打乱某一排列向量：

```
for i = [0, n)
    p[i] = i
scramble(p, n)
```

然后随机执行查找

```
assert(binarysearch1(p[i]) == p[i])
```

请比较一下这两个版本，看看在观测到的运行时间方面是否有什么不同。

8. 脚手架未得到充分的使用，并且人们也很少公开地描述这个问题。请检验一下任何你可以找到的脚手架；绝望的时候你可以会去找本书的站点。请构建脚手架，测试一

个你已编写的复杂函数。

9. 请从本书的站点中下载 search.c 脚手架程序，试试找出二分查找在你机器上的运行时间。你会使用什么工具生成输入并存储和分析其输出呢？

5.9 进阶阅读

Kernighan 和 Pike 所著的《*Practice of Programming*》在 1999 年由 Addison Wesley 出版。他们用了 50 页的篇幅来讨论调试（第 5 章）和测试（第 6 章）。他们介绍了几个超出本章范围的重要主题，比如非可再生的 bug 和回归测试。

在这本书中，有九章内容相当有趣，一定会引起每一个搞实际开发的程序员的注意。除了上述提到的两章之外，其他各章的标题包括编程风格、算法和数据结构、设计和实现、界面、性能、可移植性及表示法。他们的书籍对两个熟练程序员的技巧和风格提供了有价值的深入见解。

该书的第 3.8 节介绍了 Steve McConnell 所著的《*Code Complete*》。该书的第 25 章讲述了“单元测试”，第 26 章讲述了“调试”。

5.10 调试［补充材料］

每个程序员都知道，调试是一件困难的事情。但是熟练的调试人员可以使这项工作看起来很简单。一些程序员心烦意乱地描述他们已经花了几个小时找 bug，而主管人员问了几个问题，结果数分钟之后，程序员在瞪着眼睛看那段有故障的代码了。专业调试人员决不会忘记，不管系统行为初看起来有多么神秘，里头必定存在一定的逻辑解释。

在 IBM 的约克城研究中心中就有一段逸史说明了那种态度。一位程序员最近安装了一个新的工作站。当他坐下来时一切都正常，但是当他站起来时他就不能登录系统了。那种行为有百分之百的可重复性：坐下时可以登录而站起来时从来都不能登录。

我们当中的大多数人都只是把头靠在椅背上，对这样的情况百思不得其解。那个工作站怎么知道那个可怜的家伙是站着呢还是坐着呢？但是优秀的调试员知道里头必定大有文章。最容易怀疑到的就是电气理论了。地毯下的电线松了吗？存在静电问题吗？但也不一定就是电气方面的问题。一个机灵的同事最后问对了一个问题：程序员站着和坐着时分别是如何登录的？伸出手再试试。

问题就在键盘上：两个键的键帽松动了。程序员坐下时因为他是触摸打字，所以没有注意到这个问题；但是当他站起来时，是在寻找和敲击键盘，他就要犯错误了。

了解这点之后，专业调试员拿了把改锥，拧紧了那两个晃晃悠悠的键帽，之后一切都正常了。

芝加哥的一个银行系统已经正常运作好几个月了，但是想不到的是，第一次处理国际数据就停止工作了。程序员花了几天的时间追溯代码，但他们不能找到任何停止该程序的命令。当更仔细地观察这个问题时，他们发现当他们输入有关厄瓜多尔的某些数据时，该程序将停止。更加仔细的检测显示当用户键入首都名字（Quito，基多）时该程序会将它解释为请求停止运行程序！

Bob Martin 曾经看到一个系统“work once twice”。该系统在第一次处理事务时都是正确的，然后在后续的事务中无不例外地出现小错误。系统重新启动之后，它在处理第一次事务时又是很正常，但所有的后续事务处理都失败了。当 Martin 将该行为戏称为“work once twice”时，开发人员立刻知道去查找某个变量，程序加载时这个变量正确初始化了，但是第一次事务之后就没有正确地重新设定了。

无论在什么情况下，恰当的提问都会引导聪明的程序员迅速找出令人讨厌的 bug：“你站着和坐着的时候做了不同的事情吗？我可以看看你每次的登录方式吗？”“准确地说，退出程序之前你输入了什么？”“程序在开始失败之前工作正常过吗？有多少次？”

Rick Lemons 说在调试中他上过的最好的一堂课就是观看魔术表演了。魔术师耍了 6 个于理不通的戏法，Lemons 发现他自己也倾向于相信它们了。然后他提醒自己这些于理不通的事情绝对是不可能的，他还仔细观察每一个特技，以便解开它明显的矛盾。他先从他熟悉的基本原理——物理定律——开始，从那开始着手找出每一个戏法的简单解释。这种态度使 Lemons 成为我曾经见过的最优秀的调试员之一。

我所见过的有关排除故障（debugging）方面的最出色的书籍要数由 Berton Roueché 编著的《*The Medical Detectives*》了（该书于 1991 年由 Penguin 出版社出版）。该书的男主人公们为各种复杂的有机体消魔去病，其中有轻微恶心患者，也有重病的城镇。他们所采用的解决问题的方法可以直接应用于调试计算机系统。这些真实的故事和任何小说一样，非常具有吸引力。

第2部分　性　　能

简单而且功能强大的程序可以让用户高兴并且也不会让程序构建者烦恼。这是程序员的终极目标，也是前面五章所强调的重点。

现在我们将注意力转向那些令人愉悦的程序的某一特定方面：效率。用户使用效率低的程序时需要长时间等待，还会错失许多大好时机，这样会使他们觉得极其郁闷。因此接下来这几章将描述提高性能的几个途径。

第6章介绍了各种方法，以及它们之间是如何互相作用的。后面三章按照常用的次序分别讨论了三个改善运行时间的方法。

第7章介绍了在早期的设计过程中所使用的“封底”计算（“back-of-the-envelope” calculations）如何可以确保基本的系统结构具有足够的效率。

第8章是有关算法设计技术方面的内容，这些技术有时能够显著地降低模块的运行时间。

第9章讨论了代码优化，这个过程通常是在系统实现后期完成的。

结束第2部分时，第10章转向了性能的另一个重要方面：空间效率。

研究效率有三个很好的理由。第一个就是其在许多应用场合中固有的重要性。我敢打赌，本书的每一个读者都曾经盯着监视器发过呆，迫切希望程序运行得快一些。我认识的一位软件经理估计她有一半的开发预算都用在性能改善上了。许多程序，包括实时程序、巨型数据库系统以及交互软件，都具有严格的时间要求。

研究性能的第二个原因就是教育意义。除了实际的好处之外，效率是一个极好的培训基地。这些章节覆盖了各种不同的思想，包括算法理论以及诸如“封底”计算那样的常识性技巧。主旋律在于思维的流动性；尤其是第6章，它鼓励我们从多个不同的观点去看待同一个问题。

还有许多介绍其他主题的类似课程。这些章节可能涉及到用户界面、系统健壮性或安全性。效率具有可度量的优点：我们全都可以赞成某一程序要比另一程序快2.5倍，然而有些讨论，例如有关用户界面方面的讨论，经常要陷于个人喜好的泥沼之中。

研究性能最重要的原因就是追求速度，这一点在1986年的电影《Top Gun》中用不朽的语言做了最佳的描述：“我觉得需要……需要速度！”

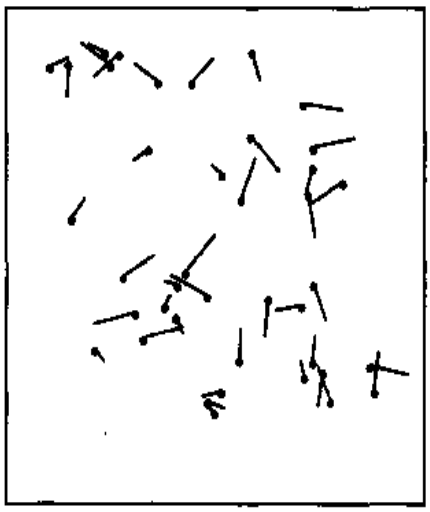
第6章 性能透视

接下来的三章将描述提高运行时效率的三种不同方法。在本章中，我们将看到这些部分如何装配在一起组成一个整体：每一种技术的应用层次都是若干计算机系统构建层次中的一种。我们先研究一个特殊的程序，然后转向研究系统设计层次，对其进行更加系统化的了解。

6.1 案例研究

Andrew Appel 在 1985 年 1 月份的《*SIAM Journal on Scientific and Statistic Computing*》第 6 卷第 1 期的第 85～103 页中描述了“多体仿真高效程序”。在不同的层次上对该程序进行研究之后，他将一年的运行时间减少为一天。

该程序主要解决经典的计算引力场中相互作用的多体问题。该程序在给定 n 个物体的质量、初始位置以及初始速度的情况下，模拟它们在三维空间中的运动；将这些物体看作是行星、恒星或者星系。在两维空间中，其输入可能是下面这样：



Appel 的论文描述了当 n=10000 时天体物理学方面的两个问题：物理学家通过研究模拟运行，可以测试理论研究与天文观测的吻合程度。（若要了解有关该问题的更多细节信息以及后来在 Appel 方法基础之上形成的解决方法，请参见由 Pfalzner 和 Gibbon 合著的并由剑桥大学出版社于 1996 年出版的《*Many-Body Tree Methods in Physics*》。

该模拟程序明细地将时间分为一些很小的“步”，并计算每一步中每个物体的进展情

况。因为它计算每个物体对其他各个物体的吸引力，所以时间步的成本大约与 n^2 相当。Appel 估计对于这一类算法来说，如果 n=10000 的话，1000 个时间步在他的计算机上意味着需要一年左右的时间。

Appel 最终解决这个问题的程序所花时间不到一天（即 400 的加速系数）。许多物理学家后来都使用他的技术。下面将简要总结一下他的这个程序，所忽略的许多重要详细信息在他的那篇论文中都能找到。重要的是，惊人的加速是通过在若干不同的层次上努力得到的。

算法和数据结构。Appel 第一个想要考虑的就是寻找一个高效的算法。通过将物理对象表示成二叉树的叶子的形式，他可以将每个时间步由 $O(n^2)$的成本减少到 $O(n \log n)$[1]。那些高一级的节点代表对象簇。作用在特定对象上的力可以使用由大簇所施加的力近似表示；Appel 指出这种近似性不会背离模拟。粗略地说，该树有 log n 级，结果得到 O(n log n)算法在某种意义上类似于第 8.3 节的分治法。这种更改将该程序运行时间减少了 12 倍。

算法优化。这个简单的算法始终使用那些极小的时间步来处理少见的两个粒子靠得很近时的情形。树数据结构允许这样的粒子对可由一个特殊的函数来识别和处理。那将加倍时间步的尺寸，因而减半了该程序的运行时间。

数据结构重组。表示对象初始设置的树不便于表示后期的设置。每个时间步都要重新组织数据结构。这要花费少量的时间，但是这减少了局部计算的次数，因而也减半了总的运行时间。

代码优化。由于树提供了额外数值精度，64 位双精度浮点数可由 32 位单精度数字进行替换，这种更改可以将运行时间减半。对该程序进行开销分析表明 98%的运行时间都花在某一函数上了；使用汇编语言重写那段代码可以将其速度增加 2.5 倍。

硬件。作了上述全部更改之后，该程序在造价 25 万美元的部门机器上运行仍然需要两天的时间，并且某些次运行已经达到了预期的效果。因此 Appel 将该程序移到一个稍贵一点的配备有浮点加速器的机器上运行，这次又将运行时间减少了一半。

将上面所描述的那些更改中的加速系数相乘，总共可得到 400 的加速系数；Appel 最后的程序运行 10000 个物体的模拟大约花了一天的时间。但是加速并不是不花代价的。简单的算法可由不多的几十行代码表示，然而，快速的程序需要 1200 行代码。快速程序的设计和实现花了 Appel 几个月的时间。下表汇总了那几方面的加速：

---

1 大O表示法 $O(n^2)$可以看作是"和 n^2 相当"；$15n^2+100n$ 和 $n^2/2-100$ 都是 $O(n^2)$。从更加形式上的定义来说，$f(n)=O(g(n))$ 意味着对于某些常数 c 和足够大的 n 值来说，$f(n)<c\ g(n)$。该表示法形式上的定义可以在算法设计或离散数学教科书中找到；第 8.5 节演示了该表示法和程序设计的相关性。

设计层次	加速系数	修改
算法和数据结构	12	二叉树，将时间由 $O(n^2)$减少为 O(n log n)
算法优化	2	使用更大的时间步
数据结构重组	2	产生适合树算法的簇
系统独立性代码优化	2	使用单精度浮点数替换双精度浮点数
系统依赖性代码优化	2.5	使用汇编语言重新编码关键函数
硬件	2	使用浮点加速器
总计	400	

此表说明了几种加速之间的依赖性。树形数据结构是主要的加速因素，它为接下来的三个更改打开了方便之门。最后两个加速因素分别是更改成汇编代码以及使用浮点加速器，在这种情况下它们和树无关。在超级计算机上，树结构对时间的影响很小（因为管道体系结构非常适合于那种简单的算法）；算法加速不必是独立于硬件的。

6.2 设计层次

计算机系统的设计层次有很多，上至高级的软件结构，下至硬件中的晶体管。下面概述直观地说明了以下各个设计层次，请不要将这看作是正式的分类。[2]

问题定义。在追求快速系统这场战斗中，可能在指定将要解决的问题上打胜仗，也可能在这上面打败仗。在我写作这一段的那天，一位提供商告诉我因为采购订单在我所在机构和采购部门之间的什么地方丢失了，他不能给我们发货。采购部门也被类似的订单淹没了，因为我所在机构的 50 个人分别单独下了订单。经过我所在机构的管理人员和采购部门的一番友好会谈之后，这 50 份订单被整理成了一份大订单。除了同时方便两个机构的管理之外，这样也可将一小块计算机系统加速 50 倍。优秀的系统分析员一定会注意到系统部署前后的这一类时间节省。

有时候良好的说明会很好地提供用户认为是必要的东西。在第 1 章中我们看到将有关输入的不多的几个重要事实合并到排序程序中会如何同时减少一个数量级的运行时间和代码长度。问题说明和效率具有微妙的交互作用；例如，良好的错误恢复可能导致编译器稍微减慢，但是它可以通过减少编译数量来减少整个时间。

系统结构。将大系统分解为模块可能是确定性能时最重要的单个因素。为整个系统提出一个骨架之后，设计者应该做一个“封底”估计，以确保其性能近乎符合要求；这一类

2 我是从 Raj Reddy 和 Allen Newell 的论文中学到本章中的这个主题的。这篇论文的题目是“*Multiplicative Speedup of Systems*（系统的成倍增加速）”（发表于由 A. K. Jones 编辑，Academic Press 出版社于 1977 年出版的《*Perspectives on Computer Science*（计算机科学透视）》。他们的论文描述了不同设计层次下的加速，其中关于硬件和系统软件方面的加速的内容尤其丰富。

计算在第7章中将作为主题来介绍。因为在新系统中获得效率要比在现有系统中进行效率改进容易多了，所以性能分析在系统设计期间是很关键的。

算法和数据结构。构建快速模块的关键通常是结构，这些结构表示了数据以及作用于数据的算法。在Appel程序中，最大的单项改进就是将$O(n^2)$算法替换为O(n log n)算法；第2章和第8章描述了类似的加速。

代码优化。Appel对代码做了小小的更改之后获得了5倍加速；第9章将详细介绍这个主题。

系统软件。有时候更改系统所依赖的软件要比更改系统本身还简单。对于此系统中的查询来说，新数据库系统是否更快呢？其他不同的操作系统是不是更加适合于此任务的实时约束呢？所有可能的编译器优化是否已启用？

硬件。运行速度更快的硬件可以提高性能。通用计算机通常都足够快；在同一个处理器或多处理器上提高时钟速度可以获得加速。声卡、视频加速器以及其他的卡将中心处理器上的工作分担到小而快速的专用处理器上。游戏设计师非常善于使用这些设备来加速。例如，专用数字信号处理器（DSP）使得一些便宜的玩具以及家用器具能进行交流。Appel的解决方法中包括给现有机器添加浮点加速器，这在那两个极端中属于折衷的解决方案。

6.3 原则

因为预防胜于治疗，所以我们应该谨记Gardon Bell在为DEC（数字设备公司）设计计算机时所作的观测报告。

计算机系统最便宜、最快速、最可靠的部件就是那些不在那儿的部件。

那些丢失的部件也是最精确（它们决不会出错）、最安全（它们不能被侵占）的部件，还是设计、文档说明、测试和维护起来最简单的部件。不能过分强调简单设计的重要性。

但是当不能回避性能问题时，考虑设计层次可以有助于集中程序员的精力。

假如你需要少许加速，请研究最好的层次。大多数程序员对效率都有自己的非条件反射："更改算法"或"优化排队纪律"马上就会脱口而出。在你决定深入研究任意给定的层次之前，请考虑一下所有可能的层次，并选择能够得到最大加速而所需的精力又最少的那个层次。

假如你需要大的加速，请深入研究多重层次。诸如Appel那样巨大的加速只能通过深入研究问题各个不同的方面得到。取得那些突破通常都需要花费大量的精力。当某一层次的更改和其他层次的更改无关（通常是这样，但也不尽然）时，各个不同的加速可

以相乘。

第7章、第8章和第9章在三个不同的设计层次讨论了加速；请在考虑单个加速时保持一定的洞察力。

6.4 问题

1. 假定现在计算机比 Appel 进行试验时要快 1000 倍。如果使用的总计算时间是一样的（大约一天），那么对于 $O(n^2)$算法和 $O(n \log n)$算法来说，问题规模 n 将增加到多少呢？

2. 请在各种不同的层次上对下面一些问题的加速进行讨论：对 500 位的整数进行因式分解；傅立叶分析；模拟 VLSI 电路；以及对大文本文件作给定字符串的搜索。请讨论各种加速的相关性。

3. Appel 发现，将双精度浮点数更改为单精度浮点数进行计算加倍了他的程序速度。请选择合适的测试并度量在你系统上的这种加速。

4. 本章集中讨论了运行时间的效率问题。其他常见的性能度量方法有容错性、可靠性、安全性、成本、性价比、精确性以及对用户错误的健壮性。请讨论一下如何可以在不同设计层次触及到这些问题中的每一个问题。

5. 请讨论一下在各个不同的设计层次利用最新技术时的成本问题，包括开发时间（日历和人员）、可维护性和费用。

6. 一个古老而又广为人知的谚语说道："正确第一、效率第二——如果答案错了，程序的速度就变得无关紧要了"。正确还是错误？

7. 请讨论一下日常生活中问题如何可以在不同的层次进行处理，比如在汽车事故中遭受的伤害。

6.5 进阶阅读

Butler Lampson 的"Hints for Computer System Design（计算机系统设计提示）"一文在 1984 年 1 月 1 号的《*IEEE Software*》期刊中发表了。其中的许多提示都是有关性能问题的；它的论文对集成软硬件的系统设计尤其有用。本书出版之时，可以在 www. research. microsoft. com /~lampson/获取该论文的副本。

第7章 封底计算

当 Bob Martin 问我“密西西比河一天的流量有多少”时，已是我们就软件工程问题进行的令人神往的对话的中途阶段了。因为发现他评论的深度达到了极高的层次，我赶紧礼貌地止住我的真实反应并说到“对不起”。当他再次问我时我意识到除了迁就一下这个可怜的小伙子之外，我别无选择了。小伙子显然已在管理一家大型软件公司的压力之下垮掉了。

我的反应有点类似这样。我算了一下，在靠近河口的地方大约有 1 英里宽，20 英尺深（或者大约 1/250 英里）。我估计水流速度大约 5 英里/小时，或者 120 英里/天。乘一下得到：

1 英里×1/250 英里×120 英里/天≈1/2 立方英里/天

这显示密西西比河每天的排水量大约是 0.5 立方英里，处在一个数量级以内。但那又怎样？

这时 Martin 从桌上拿起一份有关通信系统（这是他公司为夏季奥林匹克运动会构建的一个通信系统）的提议，还仔细检查了类似的一系列计算。通过测量给他发送一个一个字符的邮件所需的时间，他在我们谈话期间估计了一个关键的参数。其他数字都直接取自提议，因而相当精确。他的计算就像计算密西西比河的流量那样简单，但是要更有启发作用。计算表明，在一般假设下，提议的系统只有在每一分钟都至少包含 120 秒时才能工作。前一天他已经将设计返回去重新绘制了。（谈话是在奥运会之前一年进行的，最终的系统在奥运会中用到了，没有一点故障。）

那就是 Bob Martin 介绍“封底”计算工程技术的精彩（古怪）方式。该思想在工程学校中是标准食粮，但对大多数从业工程师来说，则是面包和黄油了。不幸的是，忽视计算的现象太常见了。

7.1 基本技能

下面这些提示可能有助于你进行封底计算。

*两个答案要比一个答案好。*当我问 Peter Weinberger 密西西比河每天流出多少水时，他答道：“和流进来的一样。”然后他估计密西西比盆地纵横大约有 1000 英里，那里的年

降雨量大约有 1 英尺（或者 1/5000 英里）。这就得到：

1000 英里×1000 英里×1/5000 英里/年≈200 立方英里/年

（200 立方英里/年）/（400 天/年）≈1/2 立方英里/天

或每天比 0.5 立方英里多一点。双重检测所有的计算是很重要的，对于快速计算尤其如此。

作为带有些许欺骗性的第三重检测，一部年鉴记载了该河的排水量是 640000 立方英尺/秒。根据这进行计算得知：

640000 立方英尺/秒×3600 秒/小时≈2.3×10^9 立方英尺/小时

2.3×10^9 立方英尺/小时×24 小时/天≈6×10^{10} 立方英尺/天

6×10^{10} 立方英尺/天/(5000 英尺/英里)^3≈6×10^{10} 立方英尺/天/(125×10^9 立方英尺/立方英里)

≈60/125 立方英里/天

≈1/2 立方英里/天

两次估计和另一个都很近似，尤其是年鉴的那个答案，这是一个纯粹侥幸的良好例子。

快速检测。Polya 在其《*How to Solve It*》中用三页的篇幅叙述了“按照量纲进行测试”，他将这描述为“一个著名的快速而高效的几何或物理公式检测方法”。第一个法则是求和的量纲必须是一样的，这个量纲也就是和中的量纲。你可以将英尺加在一起得到英尺，但是你不能将秒和英镑相加。第二个法则是乘积的量纲就是量纲的乘积。上述例子同时遵循这两个规则。下面的乘法：

(英里+英里)×英里×英里/天=立方英里/天

除常数之外，形式上是正确的。

使用一个简单的表可帮助我们对上述那样的复杂表达式中的量纲一目了然。进行 Weinberger 的计算时，我们先写下这三个原始的因子。

1000 英里	1000 英里	1 英里
		5000 年

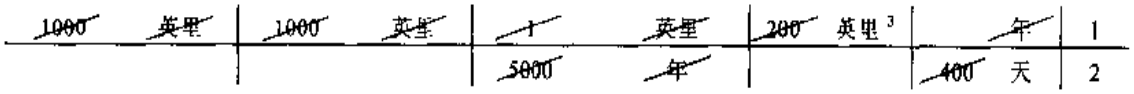
下一步我们通过删除项的方式简化该表达式，得到 200 立方英里/年的输出结果。

~~1000 英里~~	~~1000 英里~~	~~1 英里~~	20 英里^3
		~~5000 年~~	

现在我们乘以每年约有 400 天的恒等式。

~~1000 英里~~	~~1000 英里~~	~~1 英里~~	20 英里^3	年
		~~5000 年~~		400 天

删除得到（这时很熟悉的）答案：0.5 立方英里/天。



这些表格计算有助于你对量纲一目了然。

量纲测试检查了等式的形式。请使用计算尺时代的老法子检查一下你的乘除法：独立计算主要的数字和指数。人们可以对加法进行若干次快速检查。

```
  3142      3142      3142
  2718      2718      2718
 +1123     +1123     +1123
  ----      ----      ----
   983      6982      6973
```

第一次求和的位数少了，第二次求和在最末位有效位出现了错误。“舍九法”技术揭示了第三个例子中的错误：被加数中的数字加起来是 8 对 9 取模，而结果中的数字加起来是 7 对 9 取模。在正确的加法中，在去掉加起来等于 9 的数字之后，各加数的数字之和与总和的数字之和是相等的。

最重要的是，请不要忘记常识：一定要怀疑任何显示密西西比河每天排水量是 100 加仑那样的类似计算。

经验法则。我先在会计课程中学习了“72 法则”。假定你投入了一笔钱，时间是 y 年，利率每年是 r%。该规则在财务上的描述就是：如果 r×y=72，那么大致说来你投入的钱会翻番的。这个近似数相当精确：如果你投资 1000 美元，时间是 12 年，利息是 6%，那么届时你将得到 2012 美元；而花 9 年的时间，利息为 8%时，投资 1000 美元，将得到 1999 美元。

72 法则对于估计指数过程的增长来说非常便利。如果盘子里的细菌每小时增长率是 3%，那么每天它的数量都会翻倍。翻倍会使程序员回想起这个熟悉的经验法则：因为 2^{10}=1024，10 次翻倍大约就是 1000，20 次翻倍大约就是 1 百万，30 次翻倍大约就是 10 亿了。

假设一个指数程序花了 10 秒钟来解决一个规模为 n=40 的问题，并且 n 增加 1 的话就增加 12%的运行时间（我们或许可以按照对数标度将它的增长情况打点出来，从而了解这方面的内容）。72 法则告诉我们，n 每增加 6，运行时间就将翻倍，或者 n 每增加 60 时，运行时间上涨 1000 倍。因此该程序在 n=100 时应该花费 10000 秒，或几小时。但是 n 增加到 160 时会出现什么情况呢？时间会增加到 10^7 秒吗？这是花多少时间呢？

你可能发现要记住一年有 3.155×10^7 秒会很困难。另一方面，要忘记 Tom Duff 总结的下面这个便利经验法则也是很困难的，其精确度在 0.5 以内。

π 秒是一纳世纪（nanocentury）。

因为该指数程序要花 10^7 秒的时间，所以我们应该准备等上大约四个月的时间。

实践。和许多活动一样，你的估计能力只能在实践中得到提高。请练习一下本章末尾的几个问题，以及附录 2 中的估计测验（我曾经进行过类似的测验，发现我对自己的估计能力估计过高了）。第 7.8 节描述了日常生活中的快速计算。大多数工作场所都提供了大量封底估计的机会。那个盒子中有多少“包装花生”？你公司的人们每天要花多少时间排队喝早晨咖啡、吃午饭、影印以及类似事情？那些时间要花掉公司多少薪水？而下一次你确实已厌烦在桌边吃午餐时，去问你的同事密西西比河每天要流出多少水。

7.2 性能估计

现在让我们转向介绍计算中的快速计算。在你的数据结构（可以是链表或散列表）中，各个节点包含一个整数和一个指向某节点的指针：

```
struct node { int i; struct node *p; };
```

封底测验：你 128MB 主存的计算机能满足两百万个这样的节点吗？

查看我的性能监视器，发现我那 128MB 主存的机器一般只有 85MB 的空闲主存（我通过运行第 2 章中的向量旋转代码看看磁盘会在什么时候崩溃，对那个问题进行了确认）。但是一个节点到底占用多少内存呢？在老式的 16 位机器中，一个指针和一个整数将占用 4 个字节。当我编写本书时，32 位整数和指针已经极其常用了。这时应该是 8 个字节。有时我还会在 64 位模式下编译，所以它也可能是 16 个字节。我们可以用下面的一行 C 语言代码找出针对任何特定系统的答案：

```
printf("sizeof(struct node)=%d\n", sizeof(struct node));
```

正如我所预期的那样，我那个系统以 8 个字节表示每个记录。85MB 的空闲内存对于 16MB 来说应该很宽裕了。

那么，当我使用两百万个那类 8 字节记录时，为什么我那 128MB 的机器会发狂似地崩溃呢？关键在于我是使用 C 语言中的 malloc 函数（类似于 C++中的 new 运算符）动态分配它们的。我假定 8 字节记录可能具有另外的 8 个字节的开销；于是预计那些节点总计应该占用大约 32MB 的空间。实际上，每个节点都需要另外的 40 个字节的开销，从而导致每个节点总计占用 48 个字节。因此两百万个记录总计就要使用 96MB 的空间了（但是在其他系统和编译器中，那些记录每个只有 8 个字节的开销）。

附录 3 描述了一个程序，用于探究几个常见结构的内存花费。它所产生的第一行是用 sizeof 运算符得出的：

```
sizeof(char)=1  sizeof(short)=2  sizeof(int)=4
sizeof(float)=4  sizeof(struct *)=4  sizeof(long)=4
sizeof(double)=8
```

我从32位编译器那里也精确地预计到了这些值。进一步试验测出了由内存分配符返回的顺序指针之间的差别，这就可以对记录的大小进行大致的猜测了（我们还应该使用其他的工具验证这一类粗略的猜测）。现在我理解了使用这种分配符时，1到12个字节之间的记录将花费48个字节的内存，13到28个字节的记录将花费64个字节，等等。在第10章和13章，我们将回过来看这个空间模型。

让我们再来试一个快速计算测验的例子。你知道某个数值算法的运行时间是由n^3次平方根运算决定的，在本例中n=1000。那么程序计算10亿个平方根需要花费多长的时间？

为了在我的系统中找到答案，我先从下面的小C程序开始着手：

```
#include <math.h>
int main(void)
{   int i, n = 1000000;
    float fa;
    for (i = 0; i < n; i++)
        fa = sqrt(10.0);
    return 0;
}
```

我运行了该程序，其中带有一个报告程序运行时间的命令（我使用老式的数字钟检测了这一时间，我把它放在我的计算机边上。它有根针坏了，但是秒表还能使用）。我发现该程序花了0.2秒的时间计算一百万个平方根，2秒钟的时间计算1千万个，20秒的时间计算1亿个。我估计计算10亿个平方根要花200秒的时间。

但是真实程序中求一次平方根的计算会花费200纳秒的时间吗？它可能要慢得多：多半是因为平方根函数将大多数当前参数都高速缓存为起始值。使用同样的参数重复调用这一类函数可能给它带来意想不到的好处。而且，实际上该函数可能会更快些：我编译该程序时禁用了优化（优化会删除主循环，所以它总是运行0次）。附录3描述了如何展开这个小程序，以产生一页说明，描述对给定系统的原语C运算的时间成本。

联网有多快呢？为了确定有多快，我键入了ping machine-name。ping同一建筑物中的机器花了几毫秒，所以也就代表了启动时间。如果哪天运气好的话，我花大约70毫秒的时间就可以跨美国ping通另一海岸的机器（以光速来回行走5000英里大约要花27毫秒的时间）。哪天运气不好的话，我可能要延时到1000毫秒以上。复制大文件时的时间度量显示，10兆比特以太网1秒大约移动1兆字节（也就是说，它得到了其潜在带宽的80%）。类似的，100兆比特以太网速度也是飞快的，1秒可移动10兆字节。

做个小小的试验就可以看出点端倪。数据库设计者应该知道读取和写入记录的时间，

而对于各种不同窗体的连接来说，图形程序员应该知道关键屏幕操作的成本。今天你做这一类小试验所需要的短短的时间会使你在今后作出明智的决策时能节省多得多的时间。

7.3 安全系数

任何计算的输出最多只和其输入一样好。如果数据良好的话，简单的计算就可以得到精确的答案，有时这些答案是非常有用的。Don Knuth 曾经编写了一个磁盘排序包，发现运行时它花的时间是计算所预期的时间的两倍。经过努力的检测之后他发现了一个缺陷：由于软件 bug 的原因，系统中那些买来一年了的硬盘在其整个生命期中一直都只以标榜速度的一半在运行。当 bug 得到修复之后，Knuth 的排序包如预期那样运行了，并且其他所有与磁盘紧密相关的程序也运行得更快了。

但是，通常随意的输入就可以达到目的（附录 2 中的估计测验可能会有助于你判断自己的猜测能力）。如果你这里猜大约是 20%，那里猜大约是 50%，那么你仍然会发现设计和说明之间的上下差异还是 100 倍，附加的精确度是不需要的。但是在充分相信 20% 的误差幅度以前，请考虑一下 Vic Vyssotsky 在多个场合的谈话中曾提到的建议。

Vyssotsky 说："你们当中大多数人会回忆起 Galloping Gertie 来，它就是奈洛斯海峡桥，在 1940 年的一场暴风雨中断裂了。在 Galloping Gertie 以前的大约 8 年中，有几架悬浮桥也遭到了同样方式的破坏。那是气动上升现象的原因。如果需要对各个力进行正确的工程计算（这涉及到急剧的非线性）的话，你必须使用数学以及 Kolmogorov 的概念对涡旋谱进行建模。没有人真正知道如何确切地对其进行建模，直到大约 20 世纪 50 年代，这个问题才得到解决。那么，为什么 Brooklyn Bridge 不会像 Galloping Gertie 那样自己断裂呢？"

"那是因为 John Roebling 充分意识到他不知道某些东西。他那些有关 Brooklyn Bridge 设计的笔记和信件仍然存在，它们是优秀工程师意识到自己知识局限性的比较吸引人的例子。他了解到作用在吊桥上的气动上升现象之后就去观察它，并且他知道他掌握知识还不够，不足以对其进行建模。所以他将 Brooklyn Bridge 桥上的铁索强度设计成基于已知的静态和动态负载的常规计算所要求强度的 6 倍"。

"当 Roebling 被问到他提议的桥会不会像其他那么多的桥那样跨掉时，他答道：'不会的，因为我将它的强度设计到六倍于它的必要强度，这样就可以防止跨掉。'"

"Roebling 是一名优秀的工程师，他使用一个巨大的安全系数来补偿他对某些方面的无知，从而建筑了一座令人满意的桥。我们确实需要那样做吗？我承认在对我们的实时软件系统进行性能计算时，我们必须按照 2、4 或 6 的系数降低性能，以补偿我们的无知。在进行可靠性/可用性承诺时，我们应该对我们认为能够满足的目标保留一个 10 的

系数，以补偿我们的无知。在估计规模、成本以及进度时，我们应该保留2或4的系数，以弥补我们在某个方面的缺漏。我们应该按照John Roebling那样的方法进行设计，而不是和与他同时代的其他设计者那样进行设计。据我所知，在美国，迄今为止没有一座由Roebling同时代的其他人所建筑的吊桥还未跨掉，19世纪70年代在美国建筑的各种类型的桥中，有四分之一是在自建成后算起十年之内就跨掉的。”

“我想知道我们是像John Roebling那样的工程师吗？”

7.4 利特尔法则

多数封底计算都使用明显的规则：总成本是单位成本乘以个数。然而，有时我们需要更加敏感的领悟能力。Bruce Weide对一个令人称奇的万能规则做了以下笔记。

“由Denning和Buzen所引入的‘运营分析’（参见《*Computing Surveys*》第10卷，第3期，1978年11月，第225～261页）要比计算机系统的排队网络模型更加一般一些。他们的解释很出色，但是由于文章主题中心所限的原因，他们没有阐明利特尔法则的一般性。该证明方法与队列或计算机系统之间没有任何关系。利特尔法则阐述到：‘系统中物体的平均数量就是系统中物体离开系统的平均比率和每个物体在系统中所花费的平均时间的乘积。’”（如果物体进入和离开系统存在一个总体上的流量平衡的话，出去率也就是进入率。）

“我在俄亥俄州立大学的计算机体系结构课上讲授这种性能分析的技术。但是我想强调一下，这个结果是系统论中一般性的定律，也可以应用于多种其他的系统。例如，你正在排队等待进入一个非常火爆的夜总会，你可能会花一会儿时间算计一下你必须站在那儿等多长的时间，企图估计一下夜总会的进入率。但是有了利特尔法则，你就可以推出‘这个地方可以容纳60个人，平均来说，进入的人们会在里头呆上大约3个小时，所以进入率大约就是每小时20个人。队列上已排着20个人了，所以我们可能还要等一个小时。让我们先回家看一看《编程珠玑》再说。’现在你了解大致情况了吧。”

Peter Denning简洁地将这个法则用短语概括为“队列中的平均物体数量是进入率和平均滞留时间之乘积。”他将这个法则应用于他的酒窖：“我在地下室中存有150个盛酒容器，每年我都要喝完（和买回）25个容器的酒。请问每个容器我要保存多少年？利特尔法则告诉我们用150个容器除以25个容器/年即得到6年。

接着他转向了更加严肃的应用。“可以使用利特尔法则定律和流量平衡证明用于多用户系统的响应时间准则。假设平均思考时间为z的n个用户连接到了一个任意的系统中，其响应时间为r。每个用户都在思考和等待响应之间循环，所以元系统（由用户和计算机系统组成）中的总计作业数都固定在n上。如果你切断系统输出到用户之间的路

径，你将看到一个元系统，其平均负载为 n，平均响应时间为 z+r，吞吐量为 x（按每个时间单位的作业数度量）。利特尔法则表明 n=x×(z+r)，解析一下得到 r=n/x-z。”

7.5 原则

当你使用封底计算时，一定要回忆一下爱因斯坦的名言。

任何事都应该做到尽可能的简单，除非没有更简单的了。

我们知道，考虑安全系数以弥补我们估计参数时的错误，以及对手边问题的无知，简单计算也将不再是一件很简单的事情。

7.6 问题

附录 2 中的测验包含了附加题。

1. 尽管贝尔实验室距离巨大的密西西比河大约有一千英里，但是我们离平时样和的帕塞伊克河只有两英里。1992 年 6 月 10 日，在下了一周的特大暴雨之后，《*Star-Ledger*》的编辑引用一个工程师的话说：“这条河的流速大约是 200 英里/小时，要比平均流速快 5 倍”。请问你对此有什么评论？

2. 在多远距离下，骑自行车的邮差传送可移动媒介的速度可以比高速数据线传递信息更快？

3. 请问通过打字填满一张软盘需要花费你多长的时间？

4. 假设整个世界的运转按照 1000000 的系数变慢了，你的计算机执行一条指令时需要花多长的时间？你的磁盘旋转一周呢？你的磁盘臂跨盘寻找呢？你键入你的名字呢？

5. 请证明为什么“舍九法”能够测试加法。你如何可以进一步测试 72 法则？你可以证明什么呢？

6. 联合国估计 1998 年世界上的人口数量将达到 59 亿，年增长率是 1.33%。这个增长率还会持续增长吗？到 2050 年时，人口数量会达到多少呢？

7. 附录 3 描述了对你系统进行时间和空间成本建模的程序。读取有关模型之后，请写下你对系统成本的猜测。从本书的网站上可以获取这些程序。请在你的机器上运行这些程序，将得到的估计值和你的猜测比较一下。

8. 请使用快速计算估计一下本书所提出的那些设计框架的运行时间。

a）估计一下这些程序和设计的时间和空间需求。

b）大 O 算法可以看作是快速计算的形式化——它捕获增长率，忽略常系数。请对

第 6、8、11、12、13、14 和 15 章中的算法使用大 O 运行时间，估计一下将它们作为程序的形式实现时所需要的运行时间。请将你的估计和各章节中所叙述的试验结果比较一下。

9. 假设系统处理一个事务需要访问 100 次磁盘（尽管完成每次事务时，有些系统需要的次数可能更少，但有些系统需要对磁盘访问几百次）。该系统在每个磁盘中每小时可以处理多少事务？

10. 请估计一下你的城市的死亡率（用每年人口的百分率度量）。

11. [P. J. Denning]请给出证明利特尔法则的大纲。

12. 你可能会在报纸文章中看到“一枚美国 25 美分硬币的平均寿命是 30 年”那样的新闻。你可如何检验那条声明呢？

7.7 进阶阅读

有关数学常识方面我最钟爱的一本书就是 Darrel Huff 写于 1954 年的经典书籍《*How To Lie With Stastics*》，这本书在 1993 年由诺顿出版社重新出版。现在看来，书中的例子相当离奇（这些富人中有些人当时每年可以挣 2.5 万美元这么一个天文数字！），但是里面的原理却是永恒的。John Allen Paulo 的《*Innumeracy: Mathematical Illiteracy and Its Consequences*》则讲述了 1990 年解决类似问题时所采用的方法（由 Farrar、Straus 和 Giroux 出版社出版）。

物理学家也知道这个话题。Jan Wolitzky 在《*Communications of the ACM*》的专栏中写到：

我经常听到有人根据物理学家的名字，将“封底计算”称为“Fermi Approximations（费米近似）”。下面所说的就是这个故事。恩里科·费米（Enrico Fermi）、罗伯特·奥本海默（Robert Oppenheimer）以及其他一些曼哈顿项目的骨干人员藏在一堵低矮的冲击波墙之后，等待几千码之外的第一个核装置的爆炸。Fermi 将几张纸撕成小碎片，当看到火光一闪时，即把碎片撒向空中，冲击波过去之后，他用步子量出纸片飞过的距离，然后进行快速的“封底计算”，得出了炸弹的爆炸当量。这个数字在很久以后得到了昂贵监视设备的确认。

搜索字符串“back of the envelope”以及“Fermi problems”可以找到大量相关的 Web 页面。

7.8 日常生活中的快速计算[补充材料]

在《*Communications of the ACM*》中发表专栏文章后，引来了许多有趣的信件。一

位读者提到曾听一则广告说某位销售员曾经驾驶新车一年行走了100000英里，于是要他儿子验证一下这个说法能不能站稳脚跟。这里有一个快速的答案，每年有2000个工作小时（50周×40小时/周），销售员每小时可能行驶50英里，但这忽略了销售时所花费的实际时间，而它确实包含在这个说法中了。因此这个说法超出了可信范围。

日常生活为我们提供了许多磨练我们快速计算技能的机会。例如，去年一年中你花了多少钱下馆子吃饭？当听到一位纽约市人经过快速计算后说他和他的妻子每个月花在出租车上的钱要比花在房租上的钱更多时，我非常吃惊。对于加利福尼亚的读者而言（他们可能不知道什么是出租车），用花园浇花的水管向游泳池注水，要多长时间才能注满呢？

有几个读者说快速计算大约在儿童阶段就已经教过了。Roger Pinkham写到：

> 我是一个教师，一直试图向每一个听讲的人讲授封底计算，这已经有好几年了。可是我却不可思议地失败了。看来需要具有怀疑主义者的气质才行。
>
> 我父亲反复向我灌输这种思想。我出生自缅因州海岸，还是孩提时代，我就私下知道了我父亲和朋友Homer Potter之间的谈话。Homer坚持说两位来自康涅狄格州的夫人一天吃了200磅龙虾。我父亲说，“让我们来看看。如果你15分钟吃一盘，每一盘约3磅，那么每小时吃12磅，或者每天吃约100磅。所以我不相信！”
>
> Homer发誓道：“那确实是真的！你什么都不相信！”父亲不信他的话，决不改变。两个星期后，Homer说，“Fred，你知道那两个女子？她们一天只吃了20磅。”
>
> 父亲毋庸置疑地嘟哝着：“这样的话我就信了。”

其他几个读者也在讨论如何同时从父母和孩子的观点将这种态度教给孩子们。孩子们常见的问题就是“走路到华盛顿特区你需要花多长时间？”以及“今年我们用耙子耙了多少叶子？”通过适当地引导，这类问题可以鼓励孩子们保持长达一生的好奇，而付出就是经常为孩子们的问题所困扰。

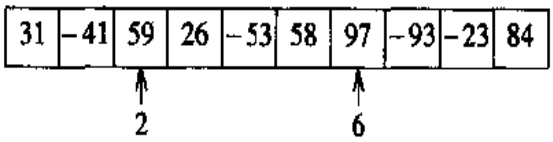
第8章　算法设计技术

第2章描述了算法设计对程序员的日常影响：算法领悟可以使程序更加简单。在本章中，我们将看到出现得更少但是效果更加显著的一方面：复杂算法有时可以极大地提高性能。

本章就一个小问题研究了四种不同的算法，重在强调设计它们时所采用的各种技术。有些算法可能稍微有点复杂，但绝对具有合理性。尽管我们研究的第一个程序要花15天的时间解决一个规模为100000的问题，但是最后一个程序解决同一问题只需花5毫秒的时间。

8.1　问题和简单算法

问题出现在一维模式识别中；我们随后将考察有关它的一些历史。问题的输入是一个具有n个浮点数字的向量x；其输出是在输入的任何相邻子向量中找出的最大和。例如，如果输入向量包含下面10个元素：



那么该程序将返回x[2..6]的总和，或187。所有数字都是正数时问题很简单，最大的子向量就是整个输入向量。当其中一些数字是负数时麻烦就出现了：我们是否应该包含负数，只是希望另一边的正数会弥补它吗？为了使问题定义更加完整，我们认为当所有的输入都是负数时，最大总和子向量是空向量，空向量的总和为0。

完成该任务的程序显然迭代了所有满足$0 \leqslant i \leqslant j < n$的i和j整数对；对每一个整数对，它都要计算x[i..j]的总和，并检查总和是否大于迄今为止的最大总和。算法1的伪码如下所示：

```
maxsofar = 0
for i = [0, n)
    for j = [i, n)
        sum = 0
        for k = [i, j]
```

```
        sum += x[k]
    /* sum is sum of x[i..j] */
    maxsofar = max(maxsofar, sum)
```

这段代码比较简短、直观并且易于理解。不幸的是，它也比较慢。例如在我的机器上，如果 n=10000，该程序需要花 22 分钟的运行时间；如果 n 为 100000，则要花费 15 天的时间。我们将在第 8.5 节中看到计时方面的详细信息。

这些时间就跟轶事一样；对于使用第 6.1 节中所描述的大 O 表示法对算法效率进行分析，我们具有不同类型的感受。外部循环恰好执行 n 次，在每次外部循环执行期间，中间循环至多执行 n 次。将这两个系数 n 相乘之后显示中间循环将执行 $O(n^2)$次。中间循环之内的内部循环执行的次数不会超过 n 次，因此，它的成本是 O(n)。将内部循环的成本和执行次数相乘之后得到整个程序的成本与 n 的立方相当。因此我们称之为立方算法。

这个例子演示了大 O 分析技术以及它的许多优点和缺点。其主要的缺点就是我们实际上仍不知道对于任意特定的输入来说，程序需要进行的具体执行次数；我们只知道步骤的数量是 $O(n^3)$。那个缺点的不足之处通常可通过该方法的另外两个长处来弥补。大 O 分析（像上面那样）比较容易实现，对于使用封底计算来确定程序是否适合于给定的应用场合来说，渐近运行时间通常已经足够了。

接下来的几节利用渐近运行时间作为程序效率的惟一衡量标准。如果你不喜欢这些内容，请直接跳到第 8.5 节，该节展示了这样的分析对于该问题是相当详尽的。但是在继续往下阅读之前，请花几分钟时间尝试着找到一个更快的算法。

8.2 两个二次算法

大多数程序员对算法 1 都有类似的反应："有一个明显的方法可以使它运行起来快得多。"实际上有两个明显的方法，如果有一个对给定程序员来说是显而易见的，那么另一个通常则不那么明显。两个算法都是二次的，对于输入规模 n 来说，需要执行 $O(n^2)$步。这两种算法都是通过以下的方式得到其运行时间的：以固定的步骤数而不是以算法 1 的 j-i+1 来计算 x[i..j]中的总和。但是这两个二次算法在以常数时间计算总和时使用了极为不同的方法。

第一个二次算法注意到 x[i..j]中的总和与前面已计算的 x[i..j-1]的总和密切相关，从而快速计算了总和。利用那种关系导致了算法 2 的产生。

```
maxsofar = 0
for i = [0, n)
    sum = 0
    for j = [i, n)
```

```
        sum += x[j]
        /* sum is sum of x[i..j] */
        maxsofar = max(maxsofar, sum)
```

第一个循环内部的语句需要执行 n 次，而每次执行外部循环时，第二个循环内的语句至多需要执行 n 次，所以总的运行时间是 $O(n^2)$。

另一个备选的二次算法是通过访问在外部循环执行之前就已构建的数据结构的方式在内部循环中计算总和。cumarr 的第 i 个元素包含 x[0..i]中的各个值的累加和，所以 x[i..j]中各个值的总和可以通过计算 cumarr[j]-cumarr[i-1]得到。这就得到了以下的算法 2b 的代码：

```
cumarr[-1] = 0
for i = [0, n)
    cumarr[i] = cumarr[i-1] + x[i]
maxsofar = 0
for i = [0, n)
    for j = [i, n)
        sum = cumarr[j] - cumarr[i-1]
        /* sum is sum of x[i..j] */
        maxsofar = max(maxsofar, sum)
```

（问题 5 解决了我们访问 cumarr[-1]的问题。）这段代码花费的时间为 $O(n^2)$；分析结果恰好和算法 2 相一致。

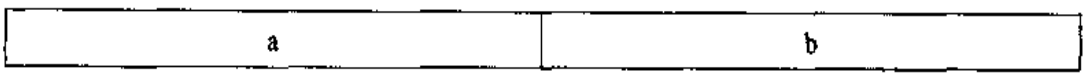
到此为止，我们所看到的算法检测了子向量中所有可能的起始值和末尾值的值对，并估计了那个子向量中各个数的总和。因为存在 $O(n^2)$个子向量，任何检测所有那些值的算法都至少必须花费二次方时间。你能想一个办法避开这个问题并且得到需要更少运行时间的方法吗？

8.3 分治算法

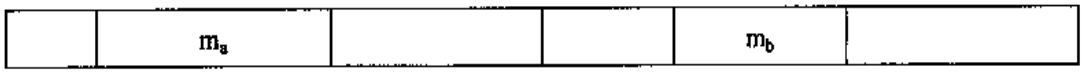
我们的第一个子二次算法是很复杂的；如果你陷入了它的内部细节，那么跳到下节时你也不会损失多少。它的基础就是下面的分治法：

> 要解决规模为 n 的问题，可递归解决两个规模近似为 n/2 的子问题，然后将它们的答案进行合并以得到整个问题的答案。

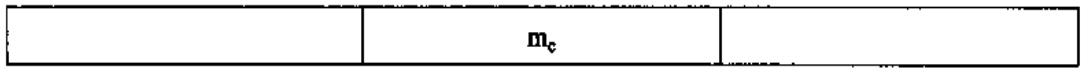
在这种情况下，初始问题要处理大小为 n 的向量，所以将它划分为两个子问题的最自然的方法就是创建两个大小相当的子向量，我们将之称为 a 和 b。



然后我们递归找出 a 和 b 中元素和最大的子向量，我们将它们称为 m_a 和 m_b。



现在很容易就可以想到我们找到问题的解了，因为在整个向量中最大总和的子向量必定是 m_a 或 m_b。这基本上接近正确。事实上，最大值要么整个在 a 中，要么整个在 b 中，或跨越 a 和 b 之间的边界；我们将跨越边界的最大值称为 m_c。



这样我们的分治算法将递归计算 m_a 或 m_b，并通过其他的方法计算 m_c，然后返回三个中最大的那一个。

上述描述足够我们编写代码了。剩下需要描述的就是我们该如何处理小向量，以及如何计算 m_c。前者比较简单：只有一个元素的向量的最大值就是该向量中的惟一值（如果该数是负数，就是 0），零元素向量的最大值定义为 0。要计算 m_c，我们观察到，其左边是从边界开始到达 a 的最大子向量，右边在 b 中的情况与此类似。将这些因素集中到一起得到下面算法 3 的代码：

```
float maxsum3(l, u)
    if (l > u)  /* zero elements */
        return 0
    if (l == u)  /* one element */
        return max(0, x[l])

    m = (l + u) / 2
    /* find max crossing to left */
    lmax = sum = 0
    for (i = m; i >= l; i--)
        sum += x[i]
        lmax = max(lmax, sum)
    /* find max crossing to right */
    rmax = sum = 0
    for i = (m, u]
        sum += x[i]
        rmax = max(rmax, sum)

    return max(lmax+rmax, maxsum3(l, m), maxsum3(m+1, u))
```

算法 3 最初通过下面的方式进行调用：

```
answer = maxsum3(0, n-1)
```

该代码相当微妙，很容易出错，但是它解决问题的时间复杂度是 O(n log n)。我们可以通过多种方式对此加以证明。一个非正式的讨论观察到，该算法在每一 O(log n)级递归上完成的工作量是 O(n)。通过使用递推关系，可以使该讨论更精确。如果 T(n)表示解决规模为 n 的问题所花的时间，那么 T(1)= O(n)且 T(n)=2T(n/2) + O(n)。

问题 15 显示了这种递推具有解 T(n)=O(n log n)。

8.4 扫描算法

我们现在就采用操作数组时最简单的算法：它从最左端（元素 x[0]）开始，一直扫描到最右端（元素 x[n-1]），记下所碰到过的最大总和子向量。最大值最初是 0。假设我们已解决了针对于 x[0..i-1]的问题，我们如何将其扩展为包含 x[i]的问题呢？我们使用类似分治算法中的道理：前 i 个元素中，最大总和子数组要么在 i-1 个元素中（我们将把它存储在 maxsofar 中），要么截止到位置 i（我们将其存储在 maxendinghere 中）。

使用类似算法 3 那样的代码重新开始计算 maxendinghere 得到另一个二次算法。我们可以使用导出算法 2 的技术避开这一点：我们不计算截止于位置 i 的最大子向量，反而使用截止于位置 i-1 的最大子向量。这就导出了算法 4。

```
maxsofar = 0
maxendinghere = 0
for i = [0, n)
    /* invariant: maxendinghere and maxsofar
       are accurate for x[0..i-1] */
    maxendinghere = max(maxendinghere + x[i], 0)
    maxsofar = max(maxsofar, maxendinghere)
```

理解这个程序的关键就是变量 maxendinghere。在该循环的第一个赋值语句前，maxendinghere 包含了截止于位置 i-1 的最大子向量的值；赋值语句修改它以包含截止于位置 i 的最大子向量的值。只要这样做能够保持其为正值，该语句将向它增加一个值 x[i]；当它变为负值时，就将它重新设为 0（因为截止于 i 的最大子向量现在是空向量了）。尽管代码相当微妙，但它比较简短，运行起来也比较快：它的运行时间是 O(n)，这样我们就可称之为线性算法。

8.5 重要性

到此为止我们对大 O 分析法已经演练得非常娴熟了，该是和盘托出和研究程序运行时间的时候了。我是用 C 语言在主频 400MHz 的奔腾 II 计算机上实现这四个基本算法的，对它们进行计时，然后外推观测到的运行时间得到下表（算法 2b 的运行时间一般在算法 2 的 10%之内，所以此表未包含它）。

此表说明了大量的问题，其中最重要的是合适的算法设计可以极大地降低运行时间，这一点在中间几行中突出强调了。最后两行显示了问题规模的增加如何与运行时间的增加相关联。

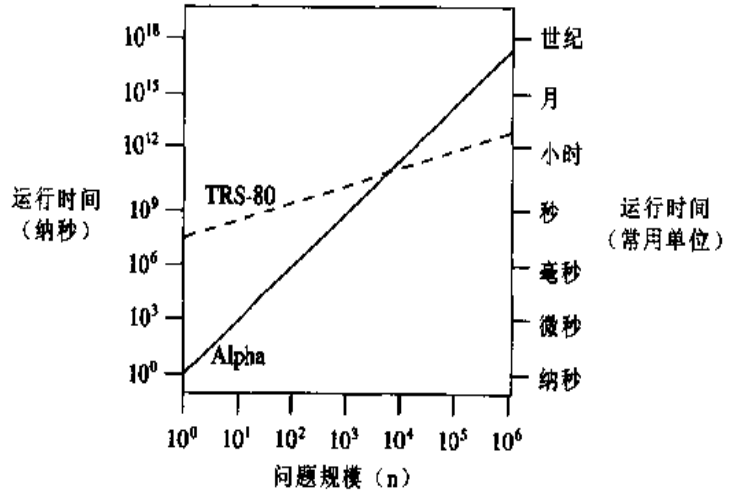
算法		1	2	3	4
运行时间（纳秒）		$1.3n^3$	$10n^2$	$47n \log_2 n$	$48n$
解决规模如右边所列出的问题时所需的时间	10^3	1.3 秒	10 毫秒	0.4 毫秒	0.05 毫秒
	10^4	22 分钟	1 秒	6 毫秒	0.5 毫秒
	10^5	15 天	1.7 分钟	78 毫秒	5 毫秒
	10^6	41 年	2.8 小时	0.94 秒	48 毫秒
	10^7	41 千年	1.4 周	11 秒	0.48 秒
单位时间内能够解决的最大问题规模	秒	920	10000	1.0×10^6	2.1×10^7
	分	3600	77000	4.9×10^7	1.3×10^9
	小时	14000	6.0×10^5	2.4×10^9	7.6×10^{10}
	天	41000	2.9×10^6	5.0×10^{10}	1.8×10^{12}
如果 n 成 10 倍地增加，时间将增加的倍数为		1000	100	10+	10
如果时间成 10 倍地增加，n 的增加倍数为		2.15	3.16	10-	10

另一个重点是，当我们将立方算法、二次算法以及线性算法进行相互比较时，程序中的常系数关系不是很大（第 2.4 节中对 O(n!)算法的讨论表明，常系数在不断加快的函数中的影响还不及在多项式中的影响那么大）。为了强调这一点，我进行了一次试验，使两个算法的常系数尽可能地不同。为了得到一个巨大的常系数，我在 Radio Shack TRS-80 Model III（1980 年的个人电脑，使用 Z-80 处理器，主频为 2.03MHz）上实现算法 4。为了进一步减慢那只可怜的老古董，我使用解释语言 Basic，这种 Basic 代码比编译代码慢 1～2 个数量级。作为另一极端，我在 Alpha 21164（主频为 533MHz）上实现了算法 1。我得到了我所希望的不均匀性：立方算法的运行时间可以度量为 $0.58n^3$ 纳秒，然而线性算法的运行时间只是 19.5n 毫秒，或者 19500000n 纳秒（也就是说，每秒大约处理 50 个元素）。下表显示了对于各种不同问题规模来说，那些表达式是如何转换为运行时间的。

n	Alpha 21164，C，立方算法	TRS-80，Basic，线性算法
10	0.6 微秒	200 毫秒
100	0.6 毫秒	2.0 秒
1000	0.6 秒	20 秒
10000	10 分钟	3.2 分钟
100000	7 天	32 分钟
1000000	19 年	5.4 小时

3300 万的常系数差异允许立方算法一开始要快一些，但是要追上线性算法是不可能的。

两种算法打个平手的点在 5800 附近，在这个位置，每种算法所花的运行时间都低于两分钟。



8.6 原则

该问题的历史可以把算法设计技术清楚明白地显示出来。该问题出现在布朗大学的 Ulf Grenander 所面对的模式匹配问题中；最初的问题如问题 13 所描述的那样是二维形式的。在那个版本中，最大总和子数组就是数字化图形中某个特定种类模式的最大可能估计量。因为二维问题要求的时间太多以至于不能解决，所以 Grenander 将它简化为一维以对其结构有更好的了解。

Grenander 观察到，算法 1 的立方时间惊人地慢，于是得出了算法 2。他于 1977 年将该问题描述给了 Michael Shamos，这个家伙花一个通宵就设计出了算法 3。不久以后当 Shamos 向我介绍这个问题时，我们一致认为那也许是可能的算法中最好的；有些研究已经表明解决若干类似的问题需要与 n log n 相当的时间。几天之后，Shamos 出席了 Carnegie Mellon 研讨会，在该会议上向统计学家 Jay Kadane 描述了该问题以及相关历史。Jay Kadane 不出几分钟就给出了算法 4 的提纲。幸运的是，我们知道没有更快的算法了：任何正确的算法都必须花费 O(n)的时间（请参见问题 6）。

虽然一维问题得到了完满的解决，Grenander 最初的二维问题在其提出二十年之后仍没有得到解决，而此时本书就要出版了。由于所有已知算法计算开销的原因，Grenander 不得不放弃那个解决他的模式匹配问题的方法。如果哪个读者觉得对于一维问题来说，线性时间算法是“显而易见”的，那么这里也鼓励他去找一找问题 13 的“显而易见”的算法。

此故事中的算法说明了几个重要的算法设计技术。

保存状态，避免重新计算。算法 2 和 4 使用了简单的动态编程形式。通过使用一些空间来保存各个结果，我们就可以避免因重新计算而浪费时间。

将信息预处理到数据结构中。算法 2b 中的 cumarr 结构允许对子向量中的总和进行快速计算。

分治算法。算法 3 使用了简单的分治法形式；有关算法设计方面的教科书介绍了更多高级的分治法形式。

扫描算法。有关数组的问题经常可以通过询问“我如何可以将 x[0..i-1]的解决方案扩展为 x[0..i]的解决方案？”的方式得到解决。算法 4 同时保存旧的答案和一些辅助数据，以计算新答案。

累积。算法 2b 使用了累积表，在累积表中，第 i 个元素包含 x 中前 i 个值的总和；在处理范围时，这一类表很常见。例如，从本年初到 10 月份为止的销售额中减去本年初到 2 月份为止的销售额，业务分析师可以确定 3 月份到 10 月份之间的销售额。

下限。只有了解到他们的算法已经是可能算法中的最佳时，算法设计师才可能安安稳稳地睡个好觉。为此，他们必须证明某个匹配下限。本问题的线性下限就是问题 6 的主题，更加复杂的下限的求解可能会十分困难。

8.7 问题

1. 算法 3 和算法 4 使用的代码相当微妙，很容易出错。请使用第 4 章中的程序验证技术证明一下该代码的正确性；指定循环变量时请务必小心。

2. 请在你的机器上对本章中的四个算法进行计时，创建一个如第 8.5 节中所示的表。

3. 我们对四个算法的分析仅限于大 O 这个层次。请尽可能精确地分析每个算法所使用的 max 函数的数量；这道练习对你分析这些程序的运行时间有什么启示吗？每个算法需要多少空间？

4. 如果输入数组中的各个元素都是从[-1,1]中选出来的随机的实数，那么最大子向量的预期值会是多少呢？

5. 为简单起见，算法 2b 访问 cumarr[-1]。你会如何使用 C 语言处理这个问题呢？

6. 请证明一下，任何计算最大子向量的正确的算法都必须检测所有 n 个输入。（有些问题的算法恰好会忽略某些输入；请思考一下答案 2.2 中的 Saxe 的算法以及 Boyer 和 Moore 的子字符串查找算法。）

7. 当我第一次实现这些算法时，我总是使用脚手架将各种不同算法所产生的答案和算法 4 中所产生的答案进行比较。当看到脚手架报告算法 2b 和算法 3 中的错误时，我心意极其烦乱，但是当我仔细研究这些数值答案时，我发现它们尽管不一样，但非常接近。

接下去会是什么呢？

8. 请修改算法3（分治算法），以在最坏的线性时间下运行。

9. 我们已将负数数组的最大子向量定义为0，也即空向量的总和。假设我们更换一下，将最大子向量定义为最大元素的值，你会如何修改各个程序呢？

10. 假设我们希望查找总和最接近0的子向量，而不是查找具有最大总和的子向量。你能设计哪种完成此任务的最有效的算法呢？可以使用哪些算法设计技术呢？如果我们希望查找一个总和最接近某一给定实数的子向量，结果又怎样呢？

11. 收费公路是由n个收费站之间的n-1段公路组成；每一段都和行驶费用挂钩。仅使用费用数组按照O(n)时间，或使用具有$O(n^2)$个项的表按照固定时间来描述任何两站之间的行驶费用都是毫无价值的。请描述一个数据结构，它需要O(n)的空间度，但它允许按照固定的时间计算任何路段的费用。

12. 对数组x[0..n-1]进行初始化，每一个元素都变为0之后，将执行下面n个运算：

```
for i = [l, u]
    x[i] += v
```

这里的l、u和v是每次运算时的参数（l和u是满足$0 \leqslant l \leqslant u < n$的整数，v是实数）。完成这n次运算之后，x[0..n-1]中的各个值就按顺序排列了。刚刚所提的方法的时间度是$O(n^2)$。你能给出一个更快的算法吗？

13. 在最大子数组问题中，给定$n \times n$的实数数组，我们必须找出任意具有最大总和的矩形子数组。该问题的复杂度如何呢？

14. 给定整数m、n和实向量x[n]，请找出使总和x[i]+…+x[i+m]最接近于0的整数i（$0 \leqslant i < n-m$）。

15. 当T(1)=0和n是2的幂时，请问递推公式T(n)=2T(n/2)+cn的解是什么？请用数学归纳法证明你所得的结果。如果T(1)=c，结果又怎样呢？

8.8 进阶阅读

只有经过广泛的研究和实践，你才能熟练地使用算法设计技术；大多数程序员只是从有关算法方面的课程或教科书中获得这些知识。Aho、Hopcroft和Ullman的《*Data Structures and Algorithms*》是一本很优秀的大学教材，由Addison-Wesley出版社于1983年出版。第10章是关于“Algorithms Design Techniques（算法设计技术）”的内容，与本章内容关系尤为紧密。

Cormen、Leiserson和Rivest的《*Introduction to Algorithms*》由MIT出版社于1990年出版。这本上千页的大部头作品对这个领域进行了全方位的概述。第I、II和III部分

涵盖了基础、排序以及查找。第 IV 部分涉及“高级设计和分析技术”，和本章主题的关系尤其紧密。第 V、VI 和 VII 部分仔细研究了高级数据结构、图算法和其他选定的话题。

这些书与其他七本书集中在一张名为《*Dr. Dobb's Essential Books on Algorithms and Data Structures*》的 CD-ROM 中。该 CD 在 1999 年由 Miller-Freeman 有限公司发行。这对所有对算法和数据结构感兴趣的程序员来说都是一本无价的指南书籍。在本书即将出版之际，可从 Dr. Dobb 站点 www.ddj.com 上订购完整的电子版，其价格仅为一本实际的书的价格。

第9章 代码优化

有些程序员对效率倾注太多的关心了：因为太在乎小小的最佳优化，所以他们创建了过分聪明然而却难以维护的程序。而另一些人则很少在意这点，他们有着漂亮的结构化程序，但是这些程序最终效率极低，因此没有实用价值。优秀的程序员会保持代码效率：效率在软件中只是众多问题中的一个，但是有时也是极其重要的一个。

前面一章已经讨论了提高效率的高层方法：问题定义、系统结构、算法设计以及数据结构选择。本章将介绍一个低层方法。“代码优化”确定现有程序中的开销昂贵的部分，然后作一些小小更改，提高其速度。“代码优化”并不总是适当的方法，并且很少能吸引人，但是它有时确实可以使程序的性能大不一样。

9.1 一个典型的故事

有一天午后不久，我和 Chris Van Wyk 在一起谈论代码优化的问题；然后他就去改善一个 C 程序了。几小时之后，他将一个具有 3000 行代码的图形程序的运行时间减少了一半。

尽管典型图像的运行时间更短了，该程序处理某些复杂的图片时仍然要花 10 分钟。Van Wyk 所采取的第一步就是绘制程序轮廓，确定每个函数需要花费的时间（下一页将介绍一个类似，但更小的程序的轮廓）。在 10 个典型的测试图片上运行该程序显示几乎 70%的运行时间都花在内存分配函数 malloc 上了。

Van Wyk 的第二步就是研究内存分配程序。因为他的程序通过单个提供错误检测的函数访问 malloc，所以他可以修改该函数，而不用分析 malloc 的源代码。他插入了少数几行记账代码，这些代码显示最流行的记录种类被分配了 30 次，比第二名更加频繁。如果你知道该程序的大部分时间都花在审核单个记录类型的内存之上，你会如何进行修改使其运行得更快呢？

Van Wyk 应用缓存原理解决了这个问题：访问最频繁的数据访问起来也应该最便宜。他修改了他的程序，将最常用类型的空闲记录捕获在一个链表中。以后在处理常见的请求时，他就可以快速访问那个链表而不需调用通用的内存分配程序；这就将程序的总运行时间减少到以前运行时间的 45%（所以内存分配程序现在大约占用总时间的 30%）。

另一个额外的好处就是修改后的分配程序减少了存储碎片，这比原来的分配程序能够更加有效地使用主存。答案 2 显示了这个古老技术的又一种实现；我们在第 13 章中将多次使用类似的方法。

这个故事很好地演示了代码优化艺术。花费不多的几小时测量 3000 行代码的程序并向其中添加 20 行代码之后，Van Wyk 加倍了其运行的速度，并且没有改变程序的用户视图，也没有增加维护的难度。他是使用一般的工具得到这种加速的：通过描述轮廓识别出程序中的“热点”，而缓存则减少在那儿所花的时间。

下面是一个典型的小 C 程序的轮廓，在形式和内容上和 Van Wyk 的轮廓都很相似：

```
    Func        Func+Child          Hit
    Time   %       Time      %      Count  Function
-----------------------------------------------------
1413.406  52.8   1413.406  52.8   200002 malloc
 474.441  17.7   2109.506  78.8   200180 insert
 285.298  10.7   1635.065  61.1   250614 rinsert
 174.205   6.5   2675.624 100.0        1 main
 157.135   5.9    157.135   5.9        1 report
 143.285   5.4    143.285   5.4   200180 bigrand
  27.854   1.0     91.493   3.4        1 initbins
```

这次运行时，大部分时间都花在 malloc 上了。问题 2 鼓励你缓存各个节点，减少该程序的运行时间。

9.2 第一个辅助采样器

现在我们从一个大的程序转向几个小的函数。每一个都描述了我曾在不同的背景下看到过的问题。这些问题在其应用场合花费了大部分的运行时间，而且解决方案也使用了一般原理。

*问题 1——整数求余。*第 2.3 节简要描述了旋转一个向量可应用的三种算法。答案 2.3 在内部循环中使用下面的运算实现了“杂耍”算法：

```
k = (j + rotdist) % n;
```

附录 3 中的成本模型显示 C 语言中的求余运算符%的开销是十分昂贵的：尽管大多数算术运算需要花费大约 10 纳秒的时间，但是%运算符花费的时间接近 100 纳秒。通过下面的代码实现该运算，我们也许可以减少程序的运行时间。

```
k = j + rotdist;
if (k >= n)
    k -= n;
```

上述代码使用比较及（很少使用的）减法运算符替换了昂贵的%运算符。但那样做会使

整个函数有什么不同吗？

我第一次运行该程序时将旋转距离 rotdist 设置为 1，运行时间从 119n 纳秒下降到 57n 纳秒。程序几乎快了两倍，62 纳秒的加速接近于成本模型中的预测。

第二次我将 rotdist 设置为 10，当看到两个算法的运行时间都是 206n 纳秒时，我感到很震惊。通过进行类似答案 2.4 中图那样的试验，我很快找到了原因：rotdist=1 时，该算法是以顺序的方式访问内存，求余运算符花费了主要的时间。但是当 rotdist=10 时，代码是按照每 10 个字访问一次的方式访问内存，因此从 RAM 检索到高速缓冲占用了主要的时间。

以前的程序员知道，如果某个程序将时间主要花费在输入输出之上，企图加速该程序中的计算将是毫无价值的。在现代的体系结构中，有这么多的周期花在访问内存上时，企图减少计算时间同样是毫无用处的。

*问题 2——函数、宏以及内联代码。*在整个第 8 章中，我们不得不计算两个值中的最大值。例如，在第 8.4 节中，我们使用类似下面所示的代码：

```
maxendinghere = max(maxendinghere, 0);
maxsofar = max(maxsofar, maxendinghere);
```

max 函数返回这两个参数中的最大值：

```
float max(float a, float b)
{   return a > b ? a : b; }
```

这个程序的运行时间是 89n 纳秒。

以前的 C 语言程序员可能会下意识地使用宏来替换这个特殊的函数：

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

这当然更加难看并且更容易出错。对于许多优化编译器来说，它根本就没有什么区别（这一类编译器以内联的方式编写各个小函数）。然而，在我的系统中，这种更改减少了算法 4 的运行时间，从 89n 纳秒减少到了 47n 纳秒，加速系数几乎为 2。

当我衡量将第 8.3 节中的算法 3 改变为使用宏这一更改效果时，我那因为上述加速而得意的高兴劲儿顿时不见了：n=10000 时，它的运行时间从 10 毫秒增加到了 100 秒，减速系数达到 10000。宏明显增加了算法 3 的运行时间，从当前的 O(n log n)增加到有点接近于 $O(n^2)$。我很快就发现，宏那种按名称调用的语法导致了算法 3 递归调用自身达两次以上，因此增加了其渐近运行时间。问题 4 提供了一个更加极端的这一类减速的例子。

当 C 程序员不得不为在性能和正确性之间进行权衡而烦扰时，C++程序员正在享受鱼和熊掌兼得的快乐。C++允许人们请求某一函数进行内联编译，这就兼有了函数干净利落的语义和宏的低开销。

在好奇心的驱使下，我同时避免使用宏和函数，在 if 语句中写出该计算：

```
if (maxendinghere < 0)
    maxendinghere = 0;
if (maxsofar < maxendinghere)
    maxsofar = maxendinghere;
```

运行时间基本上没有变化。

问题 3——顺序查找。现在我们转向研究表（很可能未排序）的顺序查找：

```
int ssearch1(t)
    for i = [0, n)
        if x[i] == t
            return i
    return -1
```

平均来说，这段简洁的代码花了 4.06n 纳秒的时间来查找位于数组 x 中的某一元素。因为在一次典型的成功查找中，它只检索数组中一半的元素，所以它需要在每一个表元素上花费大约 8.1 纳秒的时间。

该循环线条比较清晰，稍微有点肥的地方可能已经裁减掉了。内部循环有两个测试：第一个测试 i 是否在数组尾了，第二个测试 x[i]是否是预期的元素。我们可以在该数组的末尾加上一个标记值，从而使用第二个测试来替换第一个测试：

```
int ssearch2(t)
    hold = x[n]
    x[n] = t
    for (i = 0; ; i++)
        if x[i] == t
            break
    x[n] = hold
    if i == n
        return -1
    else
        return i
```

这就将运行时间降低到了 3.87n 纳秒，大约加速了 5%。上述代码假设该数组已经分配，这样 x[n]可能会被临时覆盖掉。保存 x[n]以及在查找之后对其进行恢复时都要很小心；这在大多数应用场合中都是不必要的，所以在下一个版本中会被删掉。

现在最内层循环只包含一个加法、一个数组访问以及一个测试了。还有其他进一步减少运行时间的方法吗？我们最终的顺序查找将循环展开 8 次，删除了加法；进一步展开并不会加速其运行速度。

```
int ssearch3(t)
    x[n] = t
    for (i = 0; ; i += 8)
        if (x[i  ] == t) {          break }
        if (x[i+1] == t) { i += 1; break }
```

```
        if (x[i+2] == t) { i += 2; break }
        if (x[i+3] == t) { i += 3; break }
        if (x[i+4] == t) { i += 4; break }
        if (x[i+5] == t) { i += 5; break }
        if (x[i+6] == t) { i += 6; break }
        if (x[i+7] == t) { i += 7; break }
    if i == n
        return -1
    else
        return i
```

这段代码将时间减少到了 1.70n 纳秒，下降率大约是 56%。在老式机器中，降低开销可以加速 10%或 20%。但是在现代的机器中将循环展开有助于避免流水线阻塞，减少分支，增加指令级的并行。

问题 4——计算球面距离。最后一个问题就是地理或几何数据处理方面的典型应用场合。输入的第 1 部分就是由球面上 5000 个点组成的 S 集；每个点都由经度和纬度表示。将这些点存储在我们选定的数据结构中之后，该程序就读取输入中的第 2 部分：由 20000 个点组成的序列，每个点都由经度和纬度表示。对于该序列中的每个点，程序必须指出 S 集中哪个点最接近它，其中的距离是以球体中心到两个点的连线之间的角度来度量的。

在 20 世纪 80 年代早期，Margaret Wright 在斯坦福大学对地图进行计算，总结与某些特定的基因特征在全球分布的数据时就碰到过类似的问题。她的解决方案很直观，将 S 表示成包含经度和纬度值的数组。与序列中某点最接近的邻居是通过计算该点到 S 中每一个点的距离确定的，当然这需要使用复杂的，包含 10 个 sin 和 cos 函数的三角公式。尽管该程序编码很简单，并且对小型数据集也能得到不错的结果；但是对于大型地图来说，即使在大型机上运行也需要花费几个小时。这完全超出了项目的预算。

因为我以前已处理过几何方面的问题，所以 Wright 请我试试这个问题。花了大半个周末的时间之后，我开发出了几个别出心裁的算法和数据结构来解决这个问题。幸运的是（回顾过去），每一个别出心裁的算法都需要好几百行的代码，所以我没有去试任一种算法。当我向 Andrew Appel 描述这些数据结构时，他发现了一个关键点：为什么要在数据结构的层次解决这个问题呢？为什么不使用简单的数据结构，将这些点保存在一个数组中，通过优化代码来减少点之间距离计算的成本呢？你采用他这个思想会怎样呢？

更改点的表示可以大大地降低成本：我们不使用经度和纬度来表示点，我们使用 x、y 和 z 坐标表示球面上的点。这样，该数据结构就是一个数组，它不仅包含了每个点的经度和纬度，还包含了该点的笛卡儿坐标。当程序处理序列中的每个点时，不多的几个三角函数将其经度和纬度转换成 x、y 和 z 坐标，然后我们就可以计算它到 S 集中每个点

的距离。它与S集中的点的距离计算为三个维度上各个差值的平方和，这样的开销通常要比计算一个三角函数的开销更加便宜，更不用说10个三角函数了（附录3中运行时间成本模型提供了一个系统中的详细情况）。因为两个点之间的角度随着它们欧几里得距离的平方单调增加，所以此方法计算的答案是正确的。

尽管这个方法确实需要附加的内存，但它带来了巨大的好处：当Wright将该变化合并到他的程序中时，对于复杂地图来说，运行时间由几小时降低为半分钟。在这种情况下，代码优化只通过使用二十几行的代码就解决了该问题。然而，算法和数据结构更改可能需要好几百行的代码。

9.3 主要的外科手术——二分查找

现在我们转向研究我所知道的有关代码优化方面最极端的例子之一。细节问题可以从问题4.8中得知：我们打算在一个具有1000个整数的表中执行二分查找。在我们经过该过程时，请记住代码优化在二分查找中通常不是必需的——二分查找算法的效率已经较高了，对其进行代码优化经常都是多余的。因此，在第4章中我们忽略了微观效率，集中于获得一个简单、正确以及可维护的程序。但是有时优化过的查找可能会在某个系统中产生很大的性能差异。

我们将在连续的四个程序中开发快速二分查找。它们是很微妙的，但是这里有一个很好的理由要开发这四个程序：最终的程序通常要比第4.2节中的二分查找快2或3倍。在继续往下阅读之前，你能发现下面这段代码中某些明显的浪费吗？

```
l = 0; u = n-1
loop
    /* invariant: if t is present, it is in x[l..u] */
    if l > u
        p = -1; break
    m = (l + u) / 2
    case
        x[m] <  t:  l = m+1
        x[m] == t:  p = m; break
        x[m] >  t:  u = m-1
```

我们先从修改过的问题开始我们的快速二分查找开发，这个修改过的问题就是确定整数数组x[0..n-1]中整数t第一次出现的位置；在t多次出现的情况下，上述代码可能会返回其中的任意一个位置（我们只需要第15.3节中的这一查找）。该程序的主循环类似于上面所述的那个：我们将把下标l和u保存在包含t的位置的数组中，但是我们将使用不变式关系$x[l]<t\leq x[u]$和$l<u$。我们假设$n\geq 0$，$x[-1]<t$以及$x[n]\geq t$（但是该程序决不会去访问那两个假想的元素）。现在二分查找的代码如下面这样：

```
l = -1; u = n
while l+1 != u
    /* invariant: x[l] < t && x[u] >= t && l < u */
    m = (l + u) / 2
    if x[m] < t
        l = m
    else
        u = m
/* assert l+1 = u && x[l] < t && x[u] >= t */
p = u
if p >= n || x[p] != t
    p = -1
```

第一行代码初始化不变式。循环重复时，该不变式通过 if 语句进行维护；很容易检测这两分支是否保持不变。循环终止时，我们知道 t 是否在数组中的某处，那么第一个出现的位置就在位置 u 中；这个事实在 assert 注释中更加正式地声明了。如果 t 在 x 中，那么最后两个语句将 p 设置为 t 第一次出现的下标；如果 t 不在数组中，即将 p 设置为-1。

尽管这个二分查找程序解决的问题要比前一个程序所解决的更加难些，但它潜在地更有效率：在每次循环迭代中，它只将 t 与 x 中的某个元素比较一次。前一程序有时必须测试两个这样的结果。

该程序的下一版将首次利用我们知道的 n=1000 这个事实。它使用了一个不同的范围表示方法：我们不使用它的上下限值来表示 l..u，而使用它的下限值 l 以及增量 i 来表示，比如 l+i=u。代码将确保 i 总是 2 的幂；该属性一旦具有了就容易保持，但是一开始难以获得（因为数组的大小 n 等于 1000）。因此该程序由一个赋值语句和一个 if 语句处理，以确保正在查找的范围初始大小保持为 512，即比 1000 小的最大的 2 的幂。这样 l 和 l+i 一起要么表示-1..511，要么表示 488..1000。将前面的二分查找程序转换为这个范围的新表示方法得到下面的代码：

```
i = 512
l = -1
if x[511] < t
    l = 1000 - 512
while i != 1
    /* invariant: x[l] < t && x[l+i] >= t && i = 2^j */
    nexti = i / 2
    if x[l+nexti] < t
        l = l + nexti
        i = nexti
    else
        i = nexti
/* assert i == 1 && x[l] < t && x[l+i] >= t */
p = l+1
if p > 1000 || x[p] != t
    p = -1
```

该程序的正确性证明和前一程序的证明恰好一样。这段代码通常要比其预处理程序更慢一些，但它为将来的加速开启了一扇大门。

下一程序是上述程序的简化，它合并了智能编译器可能执行的某些优化。第一个 if 语句已经简化，变量 nexti 已经删除，并将 nexti 赋值语句从内层的 if 语句中删除掉了。

```
i = 512
l = -1
if x[511] < t
    l = 1000 - 512
while i != 1
    /* invariant: x[l] < t && x[l+i] >= t && i = 2^j */
    i = i / 2
    if x[l+i] < t
        l = l + i
/* assert i == 1 && x[l] < t && x[l+i] >= t */
p = l+1
if p > 1000 || x[p] != t
    p = -1
```

虽然对该代码的正确性证明仍然具有相同的结构，但现在我们可以理解它是在更直观的层次上运行的了。当第一次测试失败且 l 保持为 0 时，程序依次计算 p 中的各个位，并且最高有效位优先计算。

该代码的最后一个版本并不是最好的。它展开了整个循环，从而消除了循环控制和 i 被 2 除的开销。因为 i 假定程序中只有 10 个互不相同的值，所以我们可以将它们全部写在代码中，从而避免在运行时重复计算。

```
l = -1
if (x[511]   < t) l = 1000 - 512
    /* assert x[l] < t && x[l+512] >= t */
if (x[l+256] < t) l += 256
    /* assert x[l] < t && x[l+256] >= t */
if (x[l+128] < t) l += 128
if (x[l+64 ] < t) l += 64
if (x[l+32 ] < t) l += 32
if (x[l+16 ] < t) l += 16
if (x[l+8  ] < t) l += 8
if (x[l+4  ] < t) l += 4
if (x[l+2  ] < t) l += 2
    /* assert x[l] < t && x[l+2  ] >= t */
if (x[l+1  ] < t) l += 1
    /* assert x[l] < t && x[l+1  ] >= t */
p = l+1
if p > 1000 || x[p] != t
    p = -1
```

我们可以插入诸如包含 x[l+256]测试那样的完整的断言字符串来理解这段代码。一旦进行两种情况分析以查看 if 语句的行为方式，那么所有其他的 if 语句就都解决了。

我曾在各个不同的系统中对第 4.2 节中的原始二分查找和这个优化过的二分查找进行比较。本书的第一版报告了跨四个机器、五种编程语言以及若干个优化层次下的各个运行时间。加速的范围也不尽相同，有的减少了 38%的运行时间，有的加速系数达到了 5（运行时间减少了 80%）。当我在我的机器上度量时，我感到一阵子惊喜，因为 n=1000 时，每次查找时的查找时间从 350 纳秒减少到了 125 纳秒（减少了 60%）。

这个加速由于表现太好，所以给人一种虚幻的感觉，但它确实是这样。经过仔细检查之后显示，我的计时脚手架是依次查找每个数组元素的：首先 x[0]，然后 x[1]，依次向前。这就给二分查找提供了特别有利的内存访问模式以及极好的分支预测。因此我将脚手架更改为随机查找元素。原始二分查找花费时间为 418 纳秒，而循环展开之后程序只要花 266 纳秒，加速了 36%。

这种推导是在最极端的情况下进行代码优化的理想化情况。我们使用一个超瘦的实质上也是更快的程序来替换这个明显的二分查找程序（该程序在外观上并不显得那么罗嗦）。（自从 20 世纪 60 年代早期起，此函数就已经在计算基础领域小有名气了。我是在 20 世纪 80 年代早期从 Guy Steele 那里学会它的；他是在 MIT　那里学会它的，这个算法在 20 世纪 60 年代后期就在 MIT 开始出名了。Vic Vysstosky 于 1961 年在贝尔实验室使用了这段代码；伪码中的每一个 if 语句都是以三个 IBM 7090 的指令实现的。）

第 4 章中的程序验证工具在该任务中起到了关键的作用。因为我们使用它们，所以我们可以相信最终的程序是不会错的；当我第一次看到这个最终的代码时，它既没有推导，也没有验证，我还以为它是什么魔术。

9.4 原则

有关代码优化最重要的原则就是尽量少用代码优化。总的来说可以概括为以下几点。

效率的角色。软件的许多其他属性和效率一样重要，甚至更重要。Don Knuth 观察到不成熟的优化是大量编程灾害的根源；它会危及程序的正确性、功能性以及可维护性。当这一点变得很重要时，请考虑适当将效率放一放。

度量工具。当效率变得重要时，第一步就是对系统进行配置，找出花费时间的位置。对程序进行轮廓描述时一般都会显示大多数的时间都花在少量的热点位置上了，而余下的代码则很少执行（例如，在第 6.1 节中，单个函数占用了 98%的运行时间）。对程序进行剖析将指出关键的区域；对于其他区域，我们遵循有名的格言“没有坏的话就不要修复它”。诸如附录 3 中那样的运行时间成本模型有助于程序员理解为什么某些特定的运算和函数成本比较高。

设计层次。我们在第 6 章中已看到效率问题的解决可以有多种方法。在优化代码以

前，我们应该确保其他方法不会提供更加有效的解决方案。

当加速的另一面是减慢时。有时候，使用 if 语句替换%求余运算符可以得到两倍的加速；而其他时候，对运行时间的影响则没有什么差别。将函数转换为宏可以两倍加速某一函数，但对于另一函数又可能减慢 10000 倍。进行改善之后，很重要的一点就是需要度量一下对典型输入的效果。由于这些故事以及更多类似故事的原因，我们必须小心地注意 Jurg Nievergelt 对代码优化人员的警告：咬人的人也应该提防被咬。

上述讨论考虑了代码优化的时机和位置；一旦我们决定进行代码优化，余下的问题就是如何进行优化了。附录 4 包含了一系列代码优化的一般规则。我们曾看到的所有例子都可以按照这些原则来解释；现在我就来示范一下，规则名称使用斜体字表示。

Van Wyk 的图形程序。Van Wyk 解决这个问题的一般策略就是*利用常用情况*。他那特定的应用包括*高速缓存*一列最常见类型的记录。

问题 1——整数求余。该解决方案*利用代数恒等式*，以便使用一个便宜的比较运算符来替换成本较高的求余运算符。

问题 2——函数、宏以及内联代码。通过使用宏来替换函数来实现*压缩过程层次结构*，几乎可以加速两倍，但是将代码写成内联的形式并没有进一步的差异。

问题 3——顺序查找。使用标记值*进行测试合并*可以获得大约 5%的加速。*循环展开*可以得到大约 56%的额外加速。

问题 4——计算球面距离。将经度、纬度和笛卡儿坐标保存在一起是*扩展数据结构*的例子；使用更简单的欧几里德距离而不是角度距离可以*利用代数恒等式*。

二分查找。*合并测试*将每次内部循环的数组比较次数从两次减少到一次；*利用代数恒等式*将上下限的表示方法更改为下限和增量表示法；*循环展开*将程序展开以消除所有的循环开销。

迄今为止，我们进行代码优化的目的都是减少 CPU 时间。我们也可以将代码优化用于其他目的，比如减少分页或增加高速缓存命中率。除了减少运行时间之外，进行代码优化最常见的使用或许就是减少程序所需要的空间了，下一章即探讨这个话题。

9.5 问题

1. 对你自己的某一程序进行剖析，然后设法使用本章中所描述的方法减少热点位置的运行时间。

2. 本书的站点包含了引言中已分析过的 C 程序；它实现了我们将在第 13 章中看到的 C++程序中的一个小小子集。请尝试在你的系统上对其进行分析。除非你有一个特别有效的 malloc 函数，否则很可能绝大多数时间都会花在 malloc 上面。请尝试一下通过实

现诸如 Van Wyk 那样的节点缓存以减少它的运行时间。

3. “杂耍”旋转算法的哪个特殊属性允许我们使用 if 语句不是使用成本更高的 while 语句替换%求余运算符？请亲自确定一下什么时候值得使用 while 语句替换%求余运算符。

4. 当n是一个代表数组最大尺寸的正整数时，下面的递归C函数将返回数组x[0..n-1]中的最大值：

```
float arrmax(int n)
{   if (n == 1)
        return x[0];
    else
        return max(x[n-1], arrmax(n-1));
}
```

当 max 是一个函数时，不到几毫秒，它就可以找出具有 10000 个元素的数组中的最大元素。当 max 是下面这个 C 宏时：

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

这个算法在具有 27 个元素的数组中找出最大值时需要花费 6 秒钟；如果具有 28 个元素，则要 12 秒钟。请提供一个返回这种可怕行为的输入，并用数学的方法分析一下这个运行时间。

5. 如果这些各不相同的二分查找算法应用于未排序的数组（违反说明），那会有什么表现呢？

6. C 和 C++库提供了字符函数，比如 isdigit、isupper 以及 islower 来确定字符的类型。你会如何实现这些函数呢？

7. 给定一个非常长的字节序列（假设有十亿或万亿），你会如何有效地计算 1 位的总数呢？（也就是说，在整个序列中有多少个位是打开的。）

8. 如何在程序中使用标记值找出数组中的最大元素？

9. 因为顺序查找比二分查找简单，所以对于那些小的表来说通常会效率更高。另一方面，在二分查找中进行比较所需的对数次数也暗示了对于那些较大的表来说，它要比顺序查找的线性时间更快一些。平衡点就是对每个程序进行优化的程度。你会使该平衡点多高或多低呢？对两个程序进行同等的优化时，在你的机器上平衡点又是多少呢？

10. D. B. Lomet 观察到散列法解决具有 1000 个整数的查找程序时可能比二分查找效率更高。请实现一个快速散列法程序并将它和优化过的二分查找进行比较，按照速度和空间来说，它们的比较情况如何？

11. 在 20 世纪 60 年代早期，Vic Berecz 发现在 Sikorsky 飞机的模拟程序中，大多数的时间都花在计算三角函数上了。进一步调查显示，只有在角度为 5 度的整数倍时才

计算这些函数。他是如何减少运行时间的？

12. 人们有时通过考虑数学而不是代码问题来对程序进行优化。为了计算下面的多项式：

$$y = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0$$

下面的代码使用了 2n 个乘法。请给出一个更快的函数。

```
y = a[0]
xi = 1
for i = [1, n]
    xi = x * xi
    y = y + a[i]*xi
```

9.6　进阶阅读

第 3.8 节描述了 Steve McConnell 的《*Code Complete*》。该书的第 28 章讲述了“代码优化策略”；它概述了一般的性能，详细描述了代码优化方法。第 29 章很好地收集了代码优化的一些规则。

本书的附录 4 提供了有关的代码优化的规则，还描述了它们在本书中的使用方式。

第10章 压缩空间

如果你跟我所知道的那几个人一样，那么读到这个题目时首先想到的就是“压缩多好啊!”在以前恶劣的计算年代里，这样的事情经常发生，程序员受限于小的机器，但这样的年代一去不复返了。新的观点是“这里1G字节，那里1G字节，很快你就要考虑到真正的内存了。”那种观点确实有点道理——许多程序员都使用大型机器，很少需要考虑从程序中压缩空间。

但时常努力地考虑压缩程序是很有利的。有时这种思考会带来新的启示，使程序变得更加简单。减少空间通常带来运行时间上合理的副作用：程序越小，加载的时候也越快，也越容易填充到高速缓存中；需要操作的数据越少，操作时所花的时间通常也就越少。跨网络传送数据时所需的时间通常和数据大小成正比。即使相对于便宜的内存来说，空间也可能是关键的。微型机器（比如在玩具和家电中看到的那些）仍然只有很小的内存。当使用巨型机来解决巨大的问题时，我们必须小心地使用内存。

注意到其重要性，让我们来概括一下减少空间的几项重要技术。

10.1 关键——简单性

简单性可以产生功能性、健壮性以及速度和空间。Dennis Ritchie和Ken Thompson最初是在具有8192个18位字的机器上开发Unix操作系统的。他们在有关系统的论文中说到“在系统及其程序方面，总是存在相当严重的大小约束。假设部分反方意见期望合理的效率和表现能力，那么大小约束不仅鼓励节省，而且还鼓励设计得优雅些。”

在20世纪50年代，当Fred Brooks为一国有公司编写一个付款程序时，他观察了简化的功能。该程序的瓶颈就出现在肯塔基州州政府征收的所得税的表示上。税收在法律中由一个二维表表示，收入作为一维，豁免数量作为另一维。显式地存储该表需要几千个内存字，比机器的容量要大。

Brooks所尝试的第一步就是为整个税表匹配一个数学函数，但是它太参差不齐了，没有哪个简单的函数可以接近。在知道这个表是由不偏爱数学函数的立法者创建的之后，Brooks咨询了一些肯塔基州的立法情况，以了解是什么论据导致这样奇异的表。他发现肯塔基州的州税是抽取联邦税之后剩余收入的简单函数。因此他的程序是从现有的程序

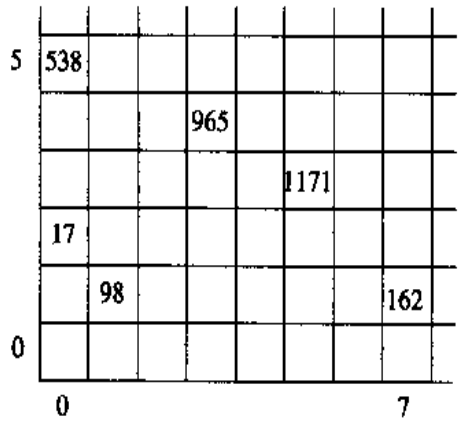
中计算联邦税，然后使用剩余的收入以及只占用几十个字内存的表来确定肯塔基的州税。

通过研究问题出现的上下文，Brooks 可以使用一个更简单的问题来替换待解决的原始问题。尽管原始问题表面看来需要好几千个字的数据空间，但是修改之后的问题却以微不足道的内存使问题得到解决。

简单性也可以减少代码的空间。第 3 章描述了几个大型程序，使用合适的数据结构可以将它们替换成更小的程序。在那些情况下，使程序更简单的观点会将源代码从几千行降低到几百行，或许还可以将对象代码的大小再减少一个数量级。

10.2 一个演示问题

在 20 世纪 80 年代早期，我查阅了一个系统，它在几何图形数据库中存储邻居。两千个邻居中的每一个在范围 0..1999 中都有一个编号，并在地图中描述为一个点。该系统通过触摸输入板，允许用户访问其中的任意一个点。该程序将选定的实际位置转换为整数对 x 和 y（均在 0..199 范围内）。输入板大致四英尺见方，该程序使用平方英寸解析度。然后程序使用（x,y）对指出用户选中了两千个点中的哪个点（如果有的话）。因为在同一位置（x,y）不可能存在两个点，所以负责该模块的程序员就使用 200×200 的点标识符（在范围 0..1999 之间的整数；如果在那个位置不存在点，则是-1）矩阵来表示该地图。该数组的左下角可能如下所示，空的方格表示该位置没有点。

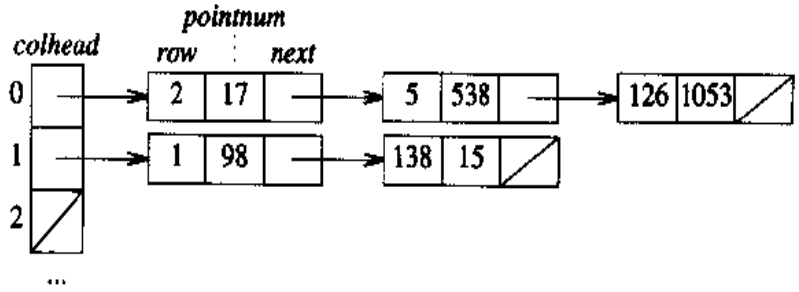


在相应的地图上，点 17 在位置（0，2），点 538 在（0，5），第一列中其他四个可见的位置是空的。

该数组很容易实现，也能得到很快的访问时间。程序员可以选择按照 16 位或 32 位来实现每个整数。要是选择 32 位整数的话，200×200=40000 个元素将花费 160 千字节。因此它选择了更短的表示法，该数组使用了 80 千字节或六分之一的半兆字节内存。在系统生命期的早期阶段那是没有什么问题的。但是随着系统的增长，它就开始用完空间了。程序员问我们如何可以减少花在这个结构上的内存。你会给该程序员什么样的建议呢？

这是一个经典的使用稀疏数据结构的机会。这个例子很古老，但最近在一个具有上百兆字节的计算机上表示一个具有100万个活动项的10000×10000的矩阵时，我看到了同样的例子。

对于稀疏矩阵来说，一个明显的表示法就是使用一个数组表示所有的列，使用链表表示给定列中的活动元素。为获得更好看的排版格式，已将此图顺时钟旋转了90°：



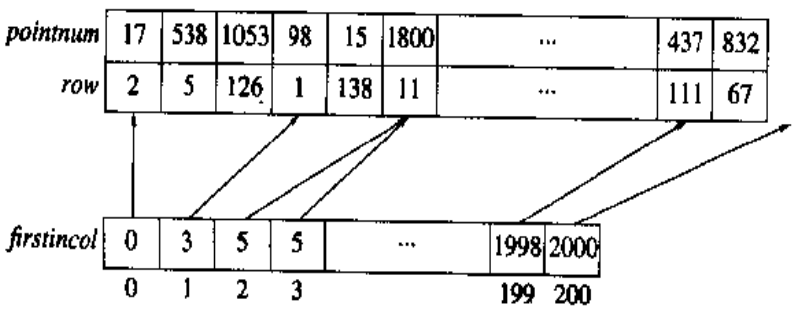
此图显示了第一列中的三个点：点17在（0，2）中，点538在（0,5）中，点1053在（0，126）中。两个点在第二列，第三列没有点。我们使用下面的代码查找点（i,j）：

```
for (p = colhead[i]; p != NULL; p = p->next)
    if p->row == j
        return p->pointnum
return -1
```

在最坏的情况下，查找某一数组元素至多要访问200个节点，但平均来说大约只要访问10个节点。

这个结构使用了具有200个指针以及2000个记录的数组。每个记录都有一个整数和两个指针。附录3中的空间成本模型告诉我们该指针将占用800个字节。如果我们将这些记录分配为具有2000个元素的数组，那么它们每个记录将占用12个字节，总的来说需要24800个字节。（如果我们打算使用该附录中所描述的默认malloc，每个记录可能要消耗48个字节。事实上，整个结构将从最初的80千字节增加到96.8千字节。）

程序员不得不在不支持指针和结构的Fortran版本中实现该结构。因此我们使用一个具有201个元素的数组表示这些列，两个具有2000个元素的并行数组表示这些点。下面就是这三个数组，底部数组中的整数下标也描述为箭头（为与本书中的其他数组保持一致性，Fortran数组以1为基数的下标已经改为使用以0为基数了）。



第 i 列中的点由位置 firstcol[i]和 firstcol[i+1]-1 之间的 row 和 pointnum 数组表示；虽然只有 200 个列，但是定义了 firstcol[200]以满足该条件。为确定位置（i,j）中存储了哪个点，我们使用下面的伪码：

```
for k = [firstincol[i], firstincol[i+1])
    if row[k] == j
        return pointnum[k]
return -1
```

这个版本使用了两个具有 2000 个元素的数组和一个具有 201 个元素的数组。该程序员使用一些 16 位整数（总字节数为 8402 个字节）刚好实现了这个结构。它要比使用完整的矩阵稍微慢些（平均来说大约需要访问节点 10 次）。即使如此，该程序满足用户的要求是没有问题的。由于系统具有良好的模块结构，通过更改一些函数，该方法几个小时之后就合并成功。我们观察到运行时间没有什么不良变化，获得了非常需要的 70 千字节。

要是该结构仍然花费空间，我们甚至可以进一步地减少它的空间。因为 row 数组的元素全部都少于 200 个，所以它们每个都可存储在单字节无符号 char 中，这就将空间减少到 6400 个字节了。如果行本身已经存储了点，那么我们甚至可以完全删除 row 数组：

```
for k = [firstincol[i], firstincol[i+1])
    if point[pointnum[k]].row == j
        return pointnum[k]
return -1
```

这就将空间减少到 4400 个字节了。

在真实系统中，快速的查找时间对于用户交互很关键，因为其他函数通过相同的界面来查找点。如果运行时间不重要，并且这些点具有 row 和 col 字段，那么我们可以通过顺序查找数组中的点，将最终的空间减少为 0 个字节。即使这些点没有这两个字段，该结构的空间也可以通过“键值索引”而减少到 4000 个字节：我们扫描整个数组，其中的第 i 个元素包含有两个单字节的字段，用于提供点 i 的 row 和 col 值。

此问题举例说明了有关数据结构方面的几个普通点。该问题很经典：稀疏数组表示[所谓稀疏数组是指其中大多数项都具有同一值（通常为 0）的矩阵]。这个问题的解决方案在概念上很简单，实现起来也很容易。我们使用了大量节省空间的方法。我们不需要 lastincol 数组和 firstincol 配对，而是利用下面的事实：此列中的最后一个点直接在下一列的第一个点之前。这是一个重新计算而非存储的普通例子。类似地，也不存在和 row 配对的 col 数组；因为我们只在整个 firstincol 数组中访问 row，所以我们总是知道当前列。尽管 row 一开始是 32 位，但是我们将它的表示减少为 16 位以及最终减少为 8 位。我们先从各个记录开始着手，但是最终要转向数组，以节省几千字节的空间。

10.3 数据空间技术

尽管简化通常是解决问题的最简单的方法，但是对某些难一些的问题它就无能为力了。在本小节中，我们将研究各种减少程序所需数据的存储空间的技术。在下一小节中，我们将考虑减少执行期间保存程序时所用的内存。

不要保存，重新计算。如果无论什么时候，在我们需要某一对象时，我们都不保存它，而是对它进行重新计算，那么保存该对象所需的空间可以急剧地减少。这恰好就是取消点阵和每次重新开始执行顺序查找的幕后思想。一个素数表将被一个检索素数函数所取代。此方法牺牲更多的运行时间来获得更少的空间。这种方法只适用于待存储的对象可以从其描述中重新计算时的情形。

这一类“生成器程序”被经常应用到在相同的随机输入上执行若干程序，其目的是进行性能比较及正确性的衰减测试。取决于应用场合，随机对象可能是具有随机产生文本行的文件或随机产生边数的图形。我们不保存整个对象，只保存它的生成器程序以及定义了特定对象的随机种子，访问它们时稍微多花点时间，巨大的对象就可以在很少的几个字节中表示出来。

当PC软件的用户从CD-ROM或DVD-ROM中安装软件时，可能会面对这一类选择。“典型安装”可能会在可以快速读取的系统硬盘中保存几百兆的数据。而“最小化安装”将把那些文件保留在更慢的设备中，但是不会占用磁盘空间。这一类安装在每次调用程序时，会花费更多的时间来读取数据，从而节省了磁盘空间。

对于许多跨网络运行的程序来说，对数据大小最追切的关心就是传输它们时要花费的时间。遵循双重建议“保存，不进行重新传输”，有时我们可以通过本地缓存方式减少需要传输的数据量。

稀疏数据结构。第10.2节已介绍过这些结构了。在第3.1节中，我们通过将一个参差不齐的三维表保存在一个二维数组中的方式节省了空间。如果我们使用将要作为索引而保存在表中的键，那么我们不需要键本身；相反，我们只保存其有关的属性，比如它被查看的次数计数。附录1中的算法一览表中描述了这种键索引技术的一些应用。在上述稀疏矩阵例子中，贯穿firstincol数组的键索引允许我们在没有col数组的情况下进行索引。

尽管程序员在修改共享对象（对象拥有者全都期望修改）时必须小心谨慎，但是将指针保存在共享的大型对象（比如长文本字符串）中会消除保存同一对象的众多副本时所需花费的成本。我在我的桌面历书中就使用了这种技术，它提供了从1821年一直到2080年之间的日历：它不是列出260个不同的日历，而是给出了14个标准日历[1月1

日是星期中的哪一天（有7种可能）乘以闰年或非闰年（2种可能）]以及一个提供260年中每一年日历编号的表。

一些电话系统将语音会话看作为稀疏结构以节省通信带宽。当音量在某一方向上下降到某一临界水平上时，即用简洁的表示法来发送静音，如此节省下来的带宽就可用来发送其他的会话。

数据压缩。信息理论告诉我们，通过压缩的方式编码对象可以减少空间。例如，在稀疏矩阵例子中，我们将行号表示从32位压缩为16位，继而再压缩为8位。在个人电脑的早期阶段，我构建了一个程序，它在读写很长的十进制数字字符串时需要花费很多的时间。我对其进行更改，通过整数 c=10×a+b，将两个十进制数字 a 和 b 编码在一个字节中。该信息可以通过以下两个语句进行解码：

```
a = c / 10
b = c % 10
```

这个简单的方案将输入输出时间减少了一半，同时也将数值数据文件压缩在一张软盘而不是两张软盘中。这一类编码可以减少单个记录所需要的空间，但是那些小的记录在编码和解码时花的时间可能要更多些（请参见问题6）。

信息理论也指出，我们可以压缩正通过某一通道（比如磁盘文件和网络）发送的记录流。可以通过记录两个通道（立体声）（其中精度为16位，采样频率为44100赫兹）的方式按照CD音效对声音进行录音。使用这种方法时，一秒的声音需要176400个字节。MP3标准将典型的声音文件（尤其是音乐）压缩为按CD标准压缩后大小的很小一部分。问题10.10要求你度量一下表示文本、图像、声音以及类似东西时几种常见格式的效率。有些程序员为他们的软件构建特殊目的的压缩算法；第13.8节纲要性地提出了如何将一个具有75000个英语单词的文件压缩到52千字节的空间中。

分配策略。有时你使用多少空间并不和你如何使用这些空间同等重要。例如，假设你的程序使用了三个不同类型的记录 x、y 和 z，所有记录都有同样的大小。在某些语言中，你的第一冲动可能就是声明这三种类型中每一种都有10000个对象。但是如果你使用10001个 x 对象，而没有 y 和 z 的话，结果又怎样呢？用到第10001个记录之后，该程序可能会用完空间，即使20000个其他对象完全未使用。通过在需要时才分配记录的方式动态分配记录可以避免这一类明显的浪费现象。

动态分配显示，在需要什么东西之前，我们不必请求那些东西；可变长度记录的策略告诉我们当我们确实需要请求某样东西时，我们应该根据需要量的多少来申请。在以前80列记录的穿孔卡片时代中，磁盘上有一半以上的字节拖着长长的空白是很常见的。可变长度文件使用换行符来指示一行的结束，因此加倍了这一类磁盘的存储量。我曾经使用可变长度记录三倍加速了具有输入输出限制的程序：记录长度最大是 250，但平均

来说只使用了其中的大约 80 个字节。

垃圾回收。回收废弃的内存之后，那些旧的位就又像新的一样了。第 14.4 节中的堆排序算法重叠了两个逻辑数据结构，它们在不同的时间应用于相同的物理存储位置。

在 20 世纪 70 年代早期，Brian Kernighan 编写了一个旅行销售员程序，使用一个共享内存的方法将空间用于两个 150×150 的矩阵。我们将这两个矩阵分别称为 a 和 b，它们用于表示点之间的距离。因此 Kernighan 知道它们的对角线是 0 值（a[i,i]=0），而其他值则是对称的（a[i,j] = a[j, i]）。因此它让两个三角矩阵共享某一方形矩阵 c 中的空间，下图是其中的一角：

0	b[0,1]	b[0,2]	b[0,3]	
a[1,0]	0	b[1,2]	b[1,3]	
a[2,0]	a[2,1]	0	b[2,3]	
a[3,0]	a[3,1]	a[3,2]	0	

随后 Kernighan 可以通过下面的代码引用 a[i,j]了：

```
c[max(i, j), min(i, j)]
```

对于 b 也是类似的，但是应该将 min 和 max 调换一下。自从那时起，这个表示法就已经在各种不同的程序中得到使用了。该技术使 Kernighan 的程序在某种程度上编写起来更加难些，运行也稍微慢些，但是在一个具有 30000 个字的机器上，将两个 22500 个字的矩阵减少成一个的效果是很显著的。而如果矩阵是 30000×30000 的话，那么在今天具有 1G 字节的机器上，同样的更改可能具有同样的效果。

在现代计算系统中，具有决定性的可能就是使用高速缓存敏感的内存布局了。尽管我研究这个理论已有许多年了，但是当我第一次使用某些多碟 CD 软件时，我会表现出发自内心深处的欣赏。全国电话簿和全国地图使用起来是一件使人高兴的事情，只在很少的时候我才会不得不替换 CD，这时我已从这个国家的一部分浏览到另一部分了。但是当我使用我的第一部两张盘的百科全书时，我发现交换 CD 这么频繁以至于我又回到以前一张 CD 的年代了，内存布局对于我的访问模式来说并不敏感。答案 2.4 图形解释了三个具有截然不同内存访问模式的算法的性能。我们将在第 13.2 节中看到一个应用，其中即使数组接触的数据比链表要多，它们也要更快，这是因为它们的顺序内存访问和系统高速缓存之间交互作用时效率更高。

10.4 编码空间技术

有时空间瓶颈不在于数据，而在于程序本身的大小。在以前条件恶劣的年代，我看到有的图形程序具有成页成页类似下面所示的代码：

```
for i = [17, 43] set(i, 68)
for i = [18, 42] set(i, 69)
for j = [81, 91] set(30, j)
for j = [82, 92] set(31, j)
```

在这里，set（i,j）打开屏幕位置（i,j）处的图片元素。使用合适的函数，比方说用于绘制水平线和垂直线的 hor 和 ver 函数，可以如下所示替换那段代码：

```
hor(17, 43, 68)
hor(18, 42, 69)
vert(81, 91, 30)
vert(82, 92, 31)
```

上述代码又可以用一个解释器来替换，这个解释器从类似下面的数组中读取命令：

```
h 17 43 68
h 18 42 69
v 81 91 30
v 82 92 31
```

如果上面代码仍然占用太多的空间，那么可以为命令（h、v 或其他两个）分配两个位以及为三个数字（这些数字是范围 0..1023 中的整数）中的每个数字都分配 10 个位，这些行中的每一行都可以用一个具有 32 位的字来表示(这种转换可以当然地由程序来进行)。这种假设的情况演示了用于减少代码空间的几种通用技术。

函数定义。通过用函数替换代码中的公共模式简化了上述程序，因而也就减少了它的空间需求，并增加了它的清晰性。这是一个“自底向上”设计的普通例子。尽管我们不能忽略自顶向下方法，但是由良好的原始对象组件和函数给出一个单一的视图，可以使系统维护起来更加简单，同时也减少了空间。

微软删除了很少使用的函数，将它的整个 Windows 系统向下压缩为更加紧凑的 Windows CE，使其能在具有更小内存的“移动计算平台”上运行。这个较小的用户界面（UI）在具有狭小屏幕的小型机器上（包括嵌入式系统、掌上电脑）运行得很好，熟悉的界面会给用户带来很大的好处。较小的应用编程接口(API)使得系统对于 Windows API 程序员来说很熟悉（并且对于许多程序来说，即使不是兼容，也是非常接近）。

解释器。在图形程序中，我们用 4 字节解释器命令去替换一行长的程序文本。第 3.2 节描述了一个用于格式信函编程的解释器，尽管它的主要目的是使程序的构造和维护更加简单，但是它意外地也减少了程序的空间。

Kernighan 和 Pike 在他们的《*Practice of Programming*》一书（在本书第 5.9 节中已描述）中将第 9.4 节用于介绍“解释器、编译器和虚拟机”。他们列举了例子来验证他们的结论：“虚拟机是以前的一个有趣想法，最近借助于 Java 和 Java 虚拟机（Java Virtual Machine，JVM）又重新流行起来了；对于使用高级语言编写的程序来说，它们提供了一种简易的方法生成可移植的、高效的程序表示。”

转换成机器语言。大多数程序员对空间减少控制得相当少的一个方面是，将源语言转换成机器语言。对编译器进行一些较次要的更改会将 Unix 系统早期版本下的代码空间减少 5 个百分点。作为最后的手段，程序员可能会考虑到将大型系统中的关键部分用汇编语言进行手工编码。然而这个昂贵且易出错的过程只会带来一点点好处，它经常用于一些内存宝贵的系统，比如数字信号处理器。

Apple Macintosh 于 1984 年引入时，是一款令人称奇的机器，这款小小的计算机（128K 字节的 RAM）具有令人吃惊的用户界面和一个功能强大的软件集。设计小组预期将发运好几百万份副本，并且只提供 64K 字节的 ROM。通过谨慎的函数定义（包括通用化运算符、合并函数和删除功能特性）以及使用汇编语言手工编码整个 ROM，该小组将难以置信的众多系统功能放到了一个极微小的 ROM 中。他们估计，他们那些经过极端优化的代码，加上谨慎的寄存器分配和指令选择，大小只有从高级语言编译过来的同样代码的一半（尽管编译器已经进行了大大的改善）。紧凑的汇编代码运行起来也非常快。

10.5 原则

空间成本。如果某一程序使用的内存超过 10%，结果又怎样呢？在某些系统中，这一类增加不会产生什么花费：先前浪费的位现在又可以使用了。在一些非常小的系统中，该程序可能根本就不能运行了：它会用完内存的。如果数据正在通过网络进行传输，在到达之前可能花费额外的 10%时间。在一些缓存和分页系统中，因为前面和 CPU 较接近的数据现在已经逆行到二级高速缓存 RAM 或磁盘了，所以运行时间可能会急剧增加（请参见第 13.2 节和答案 2.4）。在着手减少空间成本之前，你应该先了解空间成本的概念。

空间中的“热点”。第 9.4 节描述了程序的运行时间通常如何聚集在一些热点上：代码中的少量部分经常要占用大部分的运行时间。反之，就代码所需的内存来说也是这样：无论一条指令执行了 10 亿次还是根本就没有执行，它都需要相同的保存空间（除非大部分的代码从来就没有交换到主存中和小的高速缓存中）。事实上数据也可以具有热点：少数常见类型的记录经常要占用大部分的内存。例如，在稀疏矩阵例子中，在一个半兆字节内存的机器中，单个数据结构就要占用 15%的内存。使用一个只有十分之一大小的结

构替换它对系统有着重大的影响：将一个只有 1K 字节的结构在大小上减少 100 倍产生的影响基本可以忽略不计。

*度量空间。*大多数系统都提供了性能监视器，它允许程序员观察程序运行时内存的使用情况。附录 3 描述了一个用 C++语言编写的空间成本模型，和性能监视器结合使用时尤其有帮助。各种专用工具有时会有所帮助。当程序开始变得不可思议地大时，Doug Mcilroy 将连接程序的输出和源文件合并显示，目的是显示每一行耗费了多少个字节（有些宏会扩展成几百行代码）；这就允许它去裁减目标代码了。有一次通过观看由内存分配程序返回的内存块电影（“算法动画”），我发现程序中存在内存泄漏。

*权衡。*有时程序员必须权衡放弃性能、功能性或可维护性以获得内存，这一类工程决策只适用于在所有可选办法都研究过之后才能作出。本章中的几个例子介绍了减少空间如何对其他因素产生积极的影响。在第 1.4 节中，位图数据结构允许记录集保存在内存中而不是磁盘中，因此将运行时间从几分钟减少到几秒钟，代码也从几百行减少为几十行。出现这种情况的原因主要是因为原先的解决方案远远没有得到优化，但是我们那些还不太熟练的程序员会经常发现自己的代码就处于这种状态。无论何时，在我们权衡放弃任意希望得到的特性之前，我们都应该寻找各种技术，努力改善我们解决方案中的方方面面。

*和环境协作。*编程环境对于程序的空间效率具有重要的影响。重要部分包括编译器和运行时系统所使用的表示方式、内存分配策略以及分页策略。类似附录 3 中的空间成本模型可有助于确保你不会对你的系统产生消极影响。

*使用合适的作业工具。*我们已经学习过四种减少数据空间的技术（重新计算、稀疏结构、信息理论以及分配策略）、三种减少代码空间的技术（函数定义、解释器以及转换）和一条重写原则（简单性）。当内存是关键因素时，请务必全面考虑你的这些选项。

10.6　问题

1. 20 世纪 70 年代晚期，Stu Feldman 构建了一个 Fortran 77 编译器，它只适于 64K 字节的代码空间。为了节省空间，他将一些关键记录中的整数塞到一些具有四个位的字段中。当他删除包装，将这些字段保存在八个位中时，他发现尽管数据空间增加了数百个字节，但是程序的整个大小下降了好几千个字节。为什么？

2. 请问你会如何编写一个程序，构建如第 10.2 节中所描述的稀疏矩阵数据结构？你可以为该任务寻找其他简单的但空间上更有效的数据结构吗？

3. 请问你的系统总共有多大的磁盘空间？当前可用的有多少？RAM 有多少？RAM 中一般有多少是可用的？你可以度量一下你系统中各个高速缓存的大小吗？

4. 请研究一下非计算机应用程序（比如年鉴以及其他参考书）中的数据，举例说明

空间压缩。

5. 在以前的编程日子里，Fred Brooks 还面临着有关在小型计算机表示一个大表的另外一个问题（超出了本书第 10.1 节中的那些内容）。因为每一个表项只留有很少几个位的空间（实际上，对于每个项都存在一个可用的十进制数字——在早些时候我已经说过这些话了！），所以他不能在数组中存储整个表。他采用的第二个方法是利用数值分析找出匹配该表的函数。这就导致了一个非常接近于真实表的函数（没有哪个项比真实的项多一两个单元），并且该函数需要的内存总量也可忽略不计，但是税法约束意味着那种近似程度不够高。Brooks 如何可以在有限的空间中获得所需要的精确性呢？

6. 在第 10.3 节中对数据压缩的讨论曾提及到使用/和%运算符解码 10×a+b。请探讨一下使用逻辑运算和表查询替换那些运算时所涉及的时间和空间上的权衡。

7. 在常见类型的配置文件中，程序计数器的值是在常规的基础上采样的，比方说第 9.1 节中的例子。请设计一个保存一些值的数据结构，这些值在时间和空间上都具有高效率并且还提供有用的输出。

8. 明显的数据表示为数据（MMDDYYYY）分配了 8 个字节，为社会保障号（DDD-DD-DDDD）分配了 9 个字节，为名字分配了 25 个字节（其中姓占用 14 个字节、名 10 个字节、中间名 1 个字节）。如果空间紧缺，你会如何大大削减那些需求呢？

9. 尽可能地往小处压缩一本在线英语词典。计算空间时，请同时度量一下该数据文件以及解释该数据的程序。

10. 原始声音文件（比如.wav）可以压缩成.mp3 文件；原始图像文件（比如.bmp）可以压缩成.gif 或.jpeg 文件；原始动画图片文件（比如.avi）可以压缩成.mpg 文件。请试验一下这些文件格式，评估一下它们的压缩有效性。这些专用的压缩格式的压缩有效性和压缩成一般的模式（比如.gzip）相比会怎样呢？

11. 一位读者评论到："提到现代程序时，常常不是指你所编写的代码，而是指你所使用的大代码"。请研究一下你的程序，看看连接之后它会多大。你如何可以减少其空间？

10.7 进阶阅读

Fred Brooks 所著《*Mythical Man Month*》一书的 20 周年纪念版于 1995 年由 Addision-Wesley 出版。它重新印刷了原书中一些比较悦目的短文，同时也添加了几篇新的短文，包括他那比较有影响力的"No Silver Bullet——Essence and Accident in Software Engineering"。该书第 9 章的标题是"Ten pounds in a five-pound sack"，它集中介绍了在一些大型项目中对空间进行管理控制。他提出了一些重要的问题，如规模预算、函数说明以及以空间换取功能或时间。

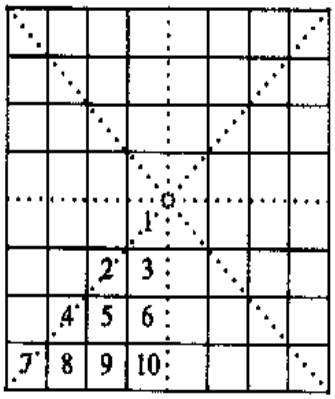
本书第 8.8 节所引用的书籍中，许多书籍都描述了以空间有效性算法和数据结构为基础的科学和技术。

10.8 巨大的压缩[补充材料]

在 20 世纪 80 年代早期，Ken Thompson 构建了一个两段式程序，它用于解决给定条件下国际象棋残局问题，比如一个王和两个相与一个王和一个马对敌（此程序和前世界计算机冠军 Belle 国际象棋机器截然不同，该机器是由 Thompson 和 Joe Condon 一起开发的）。该程序的学习阶段通过从所有可能的“将死”（象棋的最后一着）往回走来计算所有可能的走法到将死的距离，计算机科学家将这种技术称为动态编程，而国际象棋专家则称之为复盘分析。由此得到的数据库使程序对于给定的棋子无所不晓。所以在游戏阶段，它对残局下得非常出色。国际象棋专家用下面的词汇来描绘它所玩的游戏：“复杂、流畅、漫长以及困难”以及“难以忍受的缓慢和神秘”，它打乱了既定的国际象棋“教规”。

显式保存所有可能的棋盘在空间上是贵得惊人的。因此 Thompson 将棋盘编码用作键以便对棋盘信息磁盘文件进行索引；文件中的每一条记录都包含了 12 个位，包括从该位置开始到将死的步数。因为棋盘上有 64 个格子，因此五个固定的棋子位置可以编码为 5 个整数，范围在 0..63 之间，这些整数规定了每个棋子的位置。由此得到的键具有 30 个位，这就暗示着数据库中有一个具有 2^{30} 个元素的表或具有大约 10.7 亿个具有 12 个位的记录，在那时，这已经超过了可用的磁盘容量。

Thompson 的关键洞察力在于棋盘。在任何虚线的周围，它都是一个镜像图像，如下图所示。棋盘具有相同的值，没有必要在数据库进行重复。



因此他的程序假设白王位于十个已编号方格中的一个；对于任意的棋盘，按顺序至多三次镜像就可以摆放成这种格式。这种规范化将磁盘文件的大小减小到 10×64^4 或 10×2^{24} 个具有 12 个位的记录。Thompson 进一步观察到：因为黑王不能和白王相邻，因此对于两个王来说只有 454 种合法的摆棋位置，其中白王位于上述已标记的十个方格中的一个。

利用那个事实，他的数据库缩小到了 454×64^3 或大约 12100 万条具有 12 个位的记录，这就很适合于保存在单个专用的磁盘中了。

即使 Thompson 知道他的程序只可能有一个副本，他还是将该文件压缩到了一张磁盘中。Thompson 利用数据结构中的对称性将磁盘空间减少了 8 倍，这对于他整个程序的成功来说是很关键的。压缩空间也减少了程序的运行时间：通过减少在残局程序中需要分析的位置的数量，将学习阶段的时间从好几个月减少到了几周时间。

第3部分 产 品

现在开始介绍有趣的内容。第1部分和第2部分打下了基础，接下来的五章将利用这些材料来创建有趣的程序。问题本身很重要，并且它们提供了在实际应用中前述章节探讨的那些技术所聚集的焦点。

第11章讲述了几个一般用途的排序算法。第12章介绍了一个来自实际应用（产生随机整数）的特定问题，并展示了如何以各种不同方法解决这个问题。一种方法是将其看成是集合表示中的一个问题，这是第13章的主题。第14章介绍堆数据结构，并展示了它如何为排序和优先队列获得高效的算法。第15章解决了在非常长的文本字符串中查找单词或词组时所涉及的几个问题。

第11章 排　　序

如何按序排列一组记录？答案非常简单：*使用库排序函数*。我们在答案 1.1 中采用了这个方法，在第 2.8 节的变位词程序中也两次使用了该方法。不幸的是，它并不总是有效的。有的时候，现有的排序方法使用起来非常麻烦，或在解决特定问题的时候比较慢（正如我们在第 1.1 节中看到的。）在这种情况下，程序员没有选择，只能自己动手编写排序函数。

11.1 插入排序

很多玩纸牌的人都采用插入排序来排列他们手中的纸牌。他们将发到的牌按序排列，拿到新牌时，就将这张新牌插到合适的位置。假设现在要将数组 x[n]按增序排列，那么首先将第一个元素作为已排序子数组 x[0..0]，然后插入元素 x[1]，…，x[n-1]，如下面的伪码所示：

```
for i = [1, n)
    /* invariant: x[0..i-1] is sorted */
    /* goal: sift x[i] down to its
         proper place in x[0..i] */
```

下面四行体现了该算法在一个具有 4 个元素的数组上的整个应用过程。"|" 代表变量 i，它左边的元素已经排过序，而它右边的元素则是按原始顺序排列的，还没有排序。

```
3|1 4 2
1 3|4 2
1 3 4|2
1 2 3 4|
```

移动通过一个从左到右的循环实现，该循环使用变量 j 跟踪被移动的元素。只要某个元素具有前趋元素（也就是 j>0）并且没有到达其最终位置（即该元素小于它的前趋元素），循环就交换该元素和它的前趋元素。

因此，完整的排序 isort1 如下所示：

```
for i = [1, n)
    for (j = i; j > 0 && x[j-1] > x[j]; j--)
        swap(j-1, j)
```

我很少自己编写排序函数，而上面的插入排序就是我所编写的第一个函数，只有简单的三行代码。

想要优化代码的程序员可能会觉得内层循环的 swap 函数调用看起来非常不舒服。可以通过内联写入函数体以提高代码的运行效率，当然很多优化编译器会帮我们完成这项工作。使用下面的代码替换 swap 函数，该段代码通过变量 t 交换 x[j]和 x[j-1]。

```
t = x[j];  x[j] = x[j-1];  x[j-1] = t
```

在我自己的机器上，isort2 的运行时间仅是 isort1 的三分之一。

这次改动又为下一步的加速提供了思路。由于在上面的函数体中，不断地给变量 t 赋予相同的值（x[i]中的初始值），因此，可以将给 t 赋值的语句和将 t 值赋给其他变量的语句移出循环，并修改比较语句，从而得到 isort3：

```
for i = [1, n)
    t = x[i]
    for (j = i; j > 0 && x[j-1] > t; j--)
        x[j] = x[j-1]
    x[j] = t
```

只要 t 小于数组值，该段代码就将元素右移一个位置，并最终将 t 移到它的正确位置。这个五行代码的函数比前面的要复杂一些，但是在我的系统上，它要比 isort2 函数快 15%。

在随机数据和最差情况下，插入排序的运行时间和 n^2 成正比。下面列出了当输入是 n 个随机整数时，上面三个程序的运行时间：

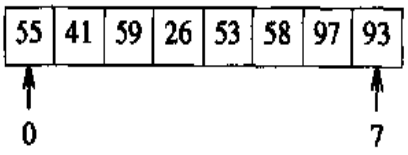
程序	C 代码行数	纳秒
插入排序 1（isort1）	3	11.9 n^2
插入排序 2（isort2）	5	3.8 n^2
插入排序 3（isort3）	5	3.2 n^2

第三个排序程序需要几毫秒的时间排序 n=1000 个整数，三分之一秒的时间排序 n=10000 个整数，将近一个小时的时间排序 100 万个整数。下面我们会介绍只需要一秒就能排序 100 万个整数的代码。如果输入数组本来的顺序就比较好，基本上是有序的，那么插入排序就非常快，因为每个元素只需要移动几步。第 11.3 节中的算法就体现了这个属性。

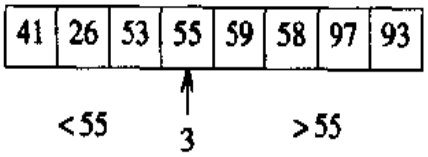
11.2 简单快速排序

C. A. R. Hoare 在 1962 年 4 月的《*Computer Journal*》第 5 卷第 1 期的第 10～15 页

上发表的一篇经典论文“快速排序”介绍了这个算法。该算法用到了第 8.3 节中介绍的分治法：排序数组时，将数组划分成两个较小的部分，然后递归排序它们。例如，假设现在要排序一个具有 8 个元素的数组：



在第一个元素 55 处划分，这样所有小于 55 的元素都位于该元素左边，所有大于 55 的元素都位于它的右边。



如果分别递归排序子数组 0 到 2 和子数组 4 到 7，那么整个数组就排序好了。

该算法的平均运行时间大大小于插入排序的 $O(n^2)$时间，因为排序该数列时需要不断执行划分操作。划分了 n 个元素之后，大概有一半元素的值大于划分值，一半元素的值小于划分值。在相同的运行时间中，插入排序中的移动操作只能将一个元素移动到适当的位置。

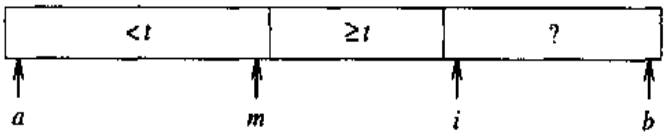
现在我们对递归函数有了大致的了解。下面使用下标 l 和 u 分别表示数组活动部分的上下界。当数组的元素小于 2 时，终止递归。代码如下所示：

```
void qsort(l, u)
    if l >= u then
        /*最多一个元素，什么都不做 */
        return
    /*目标：根据特定的值，划分数组，最终将该值放
      在正确的位置 p
     */
    qsort(l, p-1)
    qsort(p+1, u)
```

我采用了从 Nico Lomuto 处学到的简单方法将数组根据值 t 进行划分。下一节将介绍一个能够更快地完成该任务的程序[1]，由于本节提供的这个函数很容易理解，所以基本上不会出错，当然速度也肯定比较慢。给定了值 t 之后，重新组合 x[a..b]，计算下标 m（“中间值”的下标），这样所有小于 t 的元素在 m 的一端，而所有大于 t 的元素在 m 的

1 在下一节中，使用两端划分表示法来表示快速排序。虽然该段代码的基本思路非常简单，但是实现起来很需要技巧——有一次，我花了两天的时间跟踪一个 bug，结果发现问题出在一个简短的划分循环上。看过本章内容草稿的一个读者认为这个标准方法要比 Lomuto 的要简单，并马上写出一些代码来证明他的观点，而我发现了两个 bug 后就停止查看其代码了。

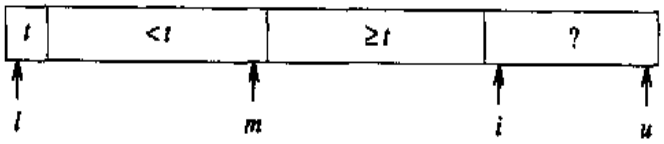
另一端。下面通过一个简单的 for 循环实现该任务，for 循环从左到右扫描数组，使用变量 i 和 m 指向数组 x 中的不变式。



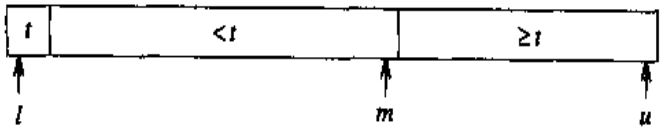
代码在检查第 i 个元素时必须考虑两种情况。如果 x[i]≥t，那么一切都很正常，不变式仍然为真。另一方面，当 x[i]<t 时，可以通过增加 m 重新获得不变式（这就需要指向小元素的新位置），然后交换 x[i]和 x[m]。完整的划分代码如下所示：

```
m = a-1
for i = [a, b]
    if x[i] < t
        swap(++m, i)
```

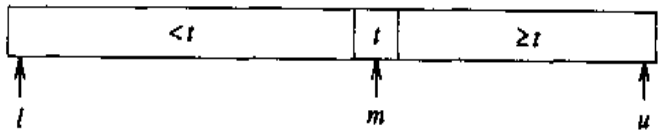
在快速排序中，我们会根据值 t=x[l]划分数组 x[l..u]，这样 a 等于 l+1，b 等于 u。因此，划分循环的不变式如下所示：



循环终止时得到：



然后交换[2]x[l]和 x[m]得到：



接着可以使用参数(l，m−1)和(m+1，u)两次递归调用该函数。

最终得到了第一个完整的快速排序 qsort1 的代码。假设调用函数 qsort1(0,n−1)排序数组 x[n]。

```
void qsort1(l, u)
    if (l >= u)
        return
```

2 本来想要忽略这个步骤，使用参数（l,m）和（m+1,u）进行递归，不幸的是，当 t 是子数组中的最大元素时，这将成为死循环。我在验证终止条件的时候发现了这个 bug，但是读者可能会猜测我究竟是如何找到这个 bug 的，Miriam Jacob 给出了一个优雅的不正确性的证明：由于从来不移动 x[l]，因此，只有当数组中的最小元素从 x[0]开始才能使用该排序。

```
m = l
for i = [l+1, u]
    /* invariant: x[l+1..m]   <  x[l] &&
                  x[m+1..i-1] >= x[l] */
    if (x[i] < x[l])
        swap(++m, i)
swap(l, m)
/* x[l..m-1] < x[m] <= x[m+1..u] */
qsort1(l, m-1)
qsort1(m+1, u)
```

问题 2 中介绍了 Bob Sedgewick 对该划分代码的修改，修改之后得到 qsort2，它要稍微快一些。

该程序的正确性证明绝大部分是推导出来的（当然是在合适的地方）。通过归纳进行证明：最外面的 if 语句正确地处理了空数组和单元素数组，而划分代码为递归调用正确地建立了较大的数组。该程序不可能造成无穷递归调用，因为每次请求都排除了元素 x[m]，这和第 4.3 节中证明二分查找终止条件是同一个问题。

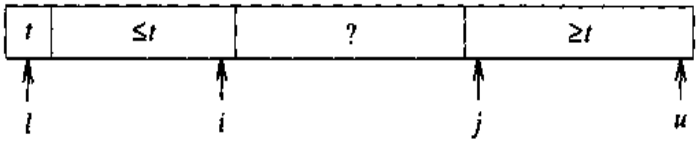
当输入数组是一个随机数列并且各个元素都不相同时，快速排序平均所需时间为 O(n log n)，栈空间为 O(log n)。它的数学原理和第 8.3 节中的相同。很多算法书都分析了快速排序的运行时间，并且证明任何基于比较的排序最少必须使用 O(n log n)次比较。因此，快速排序近乎是最优的。

qsort1 函数是到目前为止我所知道的最简单的快速排序，但是它体现了该算法的很多重要特征。第一，它确实非常快：在我的系统上，该函数只需要一秒左右的时间就能够排序 100 万个随机整数，大概比优化过的 C 库 qsort 函数快两倍（那个函数具有昂贵的通用接口）。该排序比较适用于一些表现良好的应用程序，但是它具有很多快速排序法都具有的另一个性质：在一些常见的输入下，它可能需要二次方的运行时间。下一节进一步研究更为强壮的快速排序。

11.3　更好的快速排序

qsort1 函数能够快速排序一个随机整数数组，但是如果输入不是随机的，那它的性能会怎样呢？如第 2.4 节所示，程序员经常排序很多相等的元素。因此，必须考虑一种极端情况：n 个元素都相等的数组。在这种输入情况下，插入排序的性能非常好：每个元素需要移动 0 个位置，所以总的运行时间为 O(n)。但是，qsort1 函数却不一样，在这种情况下，它的性能非常糟糕。n-1 次划分中每次都需要 O(n)时间来去掉单个元素，所以总的运行时间为 $O(n^2)$。当 n=1000000 时，运行时间从一秒一下子变成了两个小时。

通过使用下面的循环不变式，谨慎处理两端的划分可以避免这个问题：



下标 i 和 j 的初始值为要划分的数组的两个极值。主循环中有两个内部循环。第一个内部循环从左边开始，将 i 移过小元素，遇到大元素时停止。第二个内部循环从右边开始，将 j 移到大元素之后，直到遇到一个较小的元素。然后，主循环判断这两个下标，看是否交叉，然后交换两个值。

但是，如果两个元素相同，代码该如何处理呢？第一件要做的事情就是扫描右边的元素以避免做多余的工作，但是当所有的输入都相同时，这会导致时间复杂度接近二次方。这里，遇到相同的元素时会停止扫描，并交换元素。虽然这样做增加了交换的次数，但是它将所有元素都相同的最差情况变成了需要 $n \log_2 n$ 次比较的最好情况。下面的代码实现了这个划分：

```
void qsort3(l, u)
    if l >= u
        return
    t = x[l]; i = l; j = u+1
    loop
        do i++ while i <= u && x[i] < t
        do j-- while x[j] > t
        if i > j
            break
        swap(i, j)
    swap(l, j)
    qsort3(l, j-1)
    qsort3(j+1, u)
```

这样做除了能够控制所有元素都相同的情况外，它的平均交换次数也比 qsort1 少。

到目前为止我们看到的快速排序都是根据数组的第一个元素进行划分。当输入非常随机时，这样做很好，但是它需要多余的时间和空间放置常见的输入。例如，如果数组早就按递增顺序排列，那么它就可能根据最小的元素划分，然后是第二小的元素，以此类推直到整个数组，需要时间 $O(n^2)$。这样的话随机选择一个划分元素要比这样做好得多，我们通过 x[i]与 x[l..u]中的一个随机项相交换来实现：

```
swap(l, randint(l, u));
```

如果你没有现成的 randint 函数，问题 12.1 研究了你自己该如何编写这样的函数。但是不论使用怎样的代码，都要特别注意，randint 返回的值在范围[l, u]之间——如果返回的值超出了范围，就会存在一个潜在的 bug。将随机划分元素和两路划分代码结合起来，则现在得到的快速排序的运行时间就和 $n \log n$ 成正比，对任何 n 个元素的数组输入都是如此。随机性能边界不是对输入分布的假设，而是调用随机数生成器的结果。

这个快速排序使用大量的时间来排序一个非常小的子数组。使用插入排序这类简单的方法来排序这些子数组速度要快好多。Bob Sedgewick 开发了一个特别聪明的程序代码来实现这个观点。当在小的子数组上调用快速排序时（也就是当 l 和 u 非常接近时），不需要执行任何操作。通过将函数中的第一个 if 语句改成

```
if u-l > cutoff
    return
```

可以实现这一点。其中，cutoff 是一个非常小的整数。程序结束时，数组并不是有序的，但是被组合成一小块一小块随机顺序的值，并且满足这样的条件，即某一块中的元素小于它右边的任何块中的元素。必须通过另一个排序方法完成块中的排序。由于数组的排序快要完成了，比较合适选用插入排序。我们通过下面的代码排序整个数组：

```
qsort4(0, n-1)
isort3()
```

问题 3 主要研究 cutoff 的最佳值。

作为代码优化的最后一步，我们将扩展内部循环中的 swap 函数的代码（因为另外两个对 swap 的调用不在内部循环中，扩展它们变成内联代码形式对速度的影响微乎其微）。下面是快速排序的最终代码 qsort4：

```
void qsort4(l, u)
    if u - l < cutoff
        return
    swap(l, randint(l, u))
    t = x[l]; i = l; j = u+1
    loop
        do i++; while i <= u && x[i] < t
        do j--; while x[j] > t
        if i > j
            break
        temp = x[i]; x[i] = x[j]; x[j] = temp
    swap(l, j)
    qsort4(l, j-1)
    qsort4(j+1, u)
```

问题 4 和 11 提到了进一步提高快速排序性能的方法。

下表总结了快速排序的各个版本。右边一列给出了排序 n 个随机整数所需的平均运行时间，单位纳秒，在某些输入下，很多函数的运行时间都会劣化为到平方次。

程序	代码行数	纳秒
C 库 qsort	3	$137n\log_2 n$
快速排序 1（qsort1）	9	$60n\log_2 n$
快速排序 2（qsort2）	9	$56n\log_2 n$

续表

程序	代码行数	纳秒
快速排序 3（qsort3）	14	$44n\log_2 n$
快速排序 4（qsort4）	15+5	$36n\log_2 n$
C++库 sort	1	$30n\log_2 n$

qsort4 函数使用 15 行 C 代码，以及 5 行 isort3。如果排序 100 万个随机整数，那么运行时间在 0.6 秒（C++库 sort）到 2.7 秒（C 库 qsort）之间。在第 14 章中，我们将介绍一个新的算法，该算法能够在 O(n log n)时间内排序 n 个整数，甚至是在最差的情况下也能保证这个数量级的时间。

11.4 原则

本练习介绍了很多特定排序和一般编程的重要理论。

C 库 qsort 非常简单并且相对比较快，它比手写的快速排序慢仅仅是因为它的通用而且灵活的接口使用函数调用进行每个比较。C++库 sort 具有最简单的接口：通过调用 sort(x, x+n)排序数组 x；它也具有很高的执行效率。如果一个系统排序能够满足你的需求，那么就不需要自己编写代码。

插入排序编写起来非常简单并且当排序任务较小时速度比较快。在我的系统上使用 isort3 排序 10000 个整数需要 1/3 秒。

如果 n 很大，快速排序的 O(n log n)运行时间就非常关键。第 8 章中的算法设计技巧给出了分治法的基本观点，第 4 章中介绍的程序验证技巧帮助我们在精简高效的代码中实现了这些基本观点。

虽然更改算法能够大大提高程序的效率，但第 9 章中介绍的代码优化技巧也能提高插入排序和快速排序的效率，它们的加速系数分别是 4 和 2。

11.5 问题

1. 就像其他许多强大的工具一样，我们也经常会在不该使用排序的时候使用排序，而在应该使用排序的时候却不使用排序了。解释一下，在计算一个包括 n 个浮点数的数组的最大值、最小值、平均值、中值和模这些统计值时，怎样会导致过度使用排序或没有充分利用排序。

2. [R. Sedgewick]将 x[l]用作标记值来提高 Lomuto 的划分方法的效率。说明该方法

是如何在循环之后将 swap 移除的。

3. 如何找到特定系统上的最佳 cutoff 值？

4. 虽然快速排序平均仅使用了 O(log n)栈空间，但是在最差情况下，它可能使用线性空间。解释一下原因，然后修改程序，使得在最差情况下仅使用对数空间。

5. [M. D. McIlroy]证明究竟如何使用 Lomuto 的划分方法才能保证可变长度的位字符串的排序时间和它们的长度之和成正比。

6. 使用本章介绍的技巧实现其他排序算法。选择排序首先将最小的值放在 x[0]中，然后将剩余的最小值放在 x[1]中，以此类推。Shell 排序（或“递减增量排序”）类似于插入排序，但其是将元素移动 h 个位置而不是一个位置。h 值开始的时候很大，然后慢慢减小。

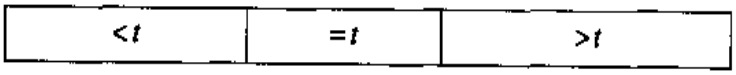
7. 本书的网站提供了本章介绍的各种排序的程序实现。统计在你的系统上运行各个排序函数所需的时间，然后将统计值制成类似于第 11.3 节中的表。

8. 起草一个一页左右的指南，告诉你系统的用户如何选择排序方法。确保你的方法考虑到了运行时间、空间、程序员时间（开发和维护所需的时间）、通用性（如果我想排序代表罗马数字的字符串会怎样呢？）、稳定性（相同的值应该保持相应的顺序）及输入数据的特殊特征等等的重要性。使用第 1 章中介绍的排序问题对你的方法进行极值测试。

9. 编写程序，让该程序在 O(n)时间内从数组 x[0..n-1]中找出第 k 个最小的元素。你的算法可以排列 x 的元素。

10. 收集并显示快速排序程序运行时间的经验数据。

11. 根据后续条件编写一个“胖支点”划分函数：



如何才能将这个函数和快速排序程序结合起来？

12. 研究非计算机应用中使用的排序方法（如邮件分拣室和零钱排序器）。

13. 本章介绍的快速排序程序随机选择一个划分元素。研究更好的选择，如数组样本的中间值。

14. 本章的快速排序使用两个整型下标表示子数组，在 Java 这类语言中必须这样做，因为它没有指向数组的指针。然而，在 C 或 C++中，可以在初始调用和所有的递归调用中都使用类似于下面的函数来排序一个整型数组：

```
void qsort(int x[], int n)
```

修改本章中的算法，让它使用这个接口。

11.6 进阶阅读

自 1973 年第一版在 Addison-Wesley 出版以来，Don Knuth 的《*Art of Computer Programming , Volume 3: Sorting and Searching*》一书就是关于本章这个主题的当之无愧的必选参考书。他详细介绍了所有重要的算法，从数学角度分析了它们的运行时间，并用汇编代码实现了它们。练习和参考中介绍了许多基本算法的重要变形。在 1998 年，Knuth 更新并修订了这本书，出版了该书的第二版。他使用的 MIX 汇编语言有些过时了，但是代码中体现的基本原理是永远都不会过时的。

Bob Sedgewick 在他的不朽的《*Algorithms*》一书的第三版中对排序和查找处理方法给出了一个更加现代的描述。第 1 到第 4 部分介绍了基础、数据结构、排序和查找。《*Algorithms in C*》由 Addison-Wesley 在 1997 年出版，《*Algorithms in C++*》（C++顾问为 Chris Van Wyk）在 1998 年出版，《*Algorithms in Java*》（Java 顾问为 Tim Lindholm）在 1999 年出版。他强调了有用算法的实现（使用你自己选择的语言），并从直观上解释了它们的性能。

这两本书是介绍排序的本章、介绍查找的第 13 章和介绍堆的第 14 章的主要参考书。

第12章 抽样问题

通常，小的计算机程序更能寓教于乐。本章介绍了一个非常小的程序，它不仅非常具有教育意义，而且对某家公司也非常有用。

12.1 一个实际问题

在1980年初，一家公司购买了他们的第一台个人电脑。我帮他们安装了基本的系统并运行之后，告诉公司里的人注意一下办公室中哪些工作是能够通过程序完成的。这家公司的业务主要涉及到公众民意调查，一个机敏的雇员建议最好能够从打印的列表区域中自动绘制出随机样本。由于手动完成该项工作非常麻烦枯燥，需要一个多小时才能处理一张随机表，她建议开发符合如下要求的一些程序：

> 输入一组区域名和一个整数m。输出结果是随机选择的m个区域组成的列表。通常有几百个区域名（每个区域名最多有12个字符），而一般m的范围是20到40。

这就是用户所需要的程序。在进行编码之前，你对这个程序的定义还有没有什么疑问？

我最初的反应就是这是一个好主意，这个任务适用于自动化。然后，我指出输入几百个名字可能要比处理一列很长的随机数方便，但是这仍然是一项非常枯燥并容易出错的工作。总的来说，当程序要忽略绝大部分内容时，准备很多输入是一个非常愚蠢的行为。因此，我提出了另一个方案。

> 输入包含两个整数m和n，并且m<n。输出一个由m个随机数字组成的有序列表，这些随机数的范围是0..n-1[1]，并且每个整数最多出现一次。就概率上而言，我们希望得到不需要替换的选择，其中每个选择出现的可能性都相同。

当m=20并且n=200时，程序可能产生一个20个元素的序列，分别为4、15、17…。然后，用户就从200个区域中找出这20个样例，在列表中标出第4、第15和第17的名字，

---

[1] 真正的程序产生范围1..n之间的m个整数。在本章中，为了跟本书其他章节中的范围保持一致，我将范围变成是基于零的，这样就能够使用该程序产生一个C数组的随机样本。程序员可以从0开始计算，但是民意调查人员是从1开始的。

诸如此类。(要求将输出结果排好序，因为硬拷贝的列表没有编号。)

这个规格说明得到了它的潜在用户的认同。程序实现之后，本来需要一个多小时才能完成的任务，现在只需要几分钟就能完成了。

下面从另一个角度来看这个问题：如何实现程序？假设有一个函数 bigrand()，该函数返回一个大的随机整数（比 m 和 n 大很多），而函数 randint(i, j)返回在范围 i..j 之间均匀选择的随机整数。问题 1 研究了这类函数的实现。

12.2 一种解决方案

一旦定下了需要解决的问题，我马上找出最新的 Knuth 的《*Seminumerical Algorithms*》的副本（在家里和办公室都放上 Knuth 三卷本的副本，工作时你会觉得你的投资很合算）。由于在十几年前我就深入研究了这本书，我隐约记得书中有几个解决类似问题的算法。花了几分钟研究了几个可能的设计之后，我认为 Knuth 在上述提到的书的第 3.4.2 节中的算法就是我面对的问题的理想解决方法。

该算法按顺序考虑整数 0，1，2，…，n-1，并通过合适的随机测试选择每个元素。通过按序访问整数，就能保证输出结果是有序的。

下面通过 m=2 和 n=5 这个例子来理解选择条件。选择第一个整数 0，概率为 2/5，程序通过下面的语句来实现：

```
if (bigrand() % 5) < 2
```

不幸的是，我们不能使用相同的概率选择 1：这样做有可能不能从 5 个整数中选择出 2 个整数。因此决策会有些偏差，如果 0 已经被选中了，那么选中 1 的概率就是 1/4，如果 0 没有选中，那么就有 2/4 的概率会选中 1。总的来说，从 r 个剩余的元素中选择 s 个元素，那么下一个元素选中的概率为 s/r。下面就是相应的伪码：

```
select = m
remaining = n
for i = [0, n)
    if (bigrand() % remaining) < select
        print i
        select--
    remaining--
```

只要 m≤n，程序就选出 m 个整数：不能选择更多的整数，因为 select 变为 0 时不能选中更多的整数；并且选择的整数也不可能少于 m，因为当 select/remaining 变为 1 时总会选中一个整数。for 语句能确保整数按序输出。通过上面的描述你可能已经发现每个子集被选中的可能性是相同的。Knuth 给出了它的概率证明。

Knuth 的第二卷使得我们能够更加容易地编写该程序。即使包含了标题、输入、输

出、范围检查及其他类似的内容，最终的程序仍然只需要 13 行 Basic 代码。在定义了问题之后，只需要半个小时的时间就编写完成了该程序，而且这个程序使用了多半年也没有出现任何问题。下面使用 C++实现该程序：

```
void genknuth(int m, int n)
{   for (int i = 0; i < n; i++)
        /* select m of remaining n-i */
        if ((bigrand() % (n-i)) < m) {
            cout << i << "\n";
            m--;
        }
}
```

这个程序仅使用了几十个字节的内存，并且能够快速地解决公司的问题。然而，当 n 非常大时，该代码可能非常慢。例如，在我的机器上使用该算法生成一些随机的 32 位正整数（也就是 $n=2^{32}$）大概需要 12 分钟。封底测试：使用该代码生成一个随机的 48 或 64 位整数需要多长时间？

12.3　设计空间

程序员的一个任务就是解决当前存在的问题。另一个更为重要的任务就是准备解决将来可能出现的问题。有时，这种准备包括参加培训或读书（如 Knuth 的书）。然而更通常的情况是，通过询问如何才能通过不同的方式解决问题往往会使你能够学到更多的东西。下面通过研究抽样问题可能的设计空间进行学习。

我在 West Point 举行讲座的时候，跟一个班级的学员讨论到了这个问题，我要求他们给出一个比原问题语句（为程序输入 200 个名字）更好的方法。一个学员建议将区域列表拍下来，使用切纸机剪切这个副本，在纸袋子里摇动纸片，然后挑出所需的号码。这个学生的这种方法体现了“概念爆炸”，这是第 1.7 节[2]引用的 Adam 的书中的主题。

从现在开始，主要是要编写一个程序，该程序按随机顺序从 0 到 n-1 中选择输出 m 个整数。首先来评价程序 1。该程序的算法相当简单，代码很短，只需要很少的空间，对于这个应用来说，运行时间也是比较合适的。运行时间和 n 成比例，但是对一些应用来说，仍不能被接受。因此需要几分钟的时间来研究解决这个问题的其他方法。在阅读下面内容之前，尽可能多地想出一些高层设计，现在并不用担心具体的实现细节。

一个解决方法就是在一个初始为空的集合中插入随机整数，直到填入足够的整数。伪码如下所示：

2 那本书的第 57 页概述了 Arthur Koestler 观点中的三个闪光点。“Ah!”表示他的名字非常具有创造性，“aha!（啊哈!）”洞察力表示发现活动。他将这个学员的解决方法叫做“haha!”。这表示：使用技术含量很低的答案来解答高技术含量的问题是一种非常有意思的启发（如答案 1.10、1.11 和 1.12 所示）。

```
initialize set S to empty
size = 0
while size < m do
    t = bigrand() % n
    if t is not in S
        insert t into S
        size++
print the elements of S in sorted order
```

该算法在选择元素时能够保证所有的元素都具有相同的选中概率，它的输出是随机的。但是还有一个问题，那就是集合 S 的实现，所以必须要考虑使用合适的数据结构。

以前，我总是考虑排序链表、二分查找树和其他所有常用数据结构的优势。现在，我可以利用 C++标准模板库中现有的东西，并将集合称为 set：

```
void gensets(int m, int n)
{   set<int> S;
    while (S.size() < m)
        S.insert(bigrand() % n);
    set<int>::iterator i;
    for (i = S.begin(); i != S.end(); ++i)
        cout << *i << "\n";
}
```

我非常高兴地看到实际的代码和伪码的长度相同。在我的机器上，这个程序只需要 20 秒的时间就能够产生并输出 100 万个排好序的不重复的 31 位随机整数。由于大概需要 12.5 秒的时间就能生成并输出 100 万个没有排过序的整数，而且这些整数肯定不会重复出现，因而集合操作就大概需要 7.5 秒的时间。

C++标准模板库规范能够保证在 O(log m)时间内完成每个插入操作，集合内部迭代需要 O(m)，所以整个程序需要的时间为 O(m log m)（和 n 相比，m 较小时）。然而，该数据结构的空间消耗相对较大：当 m=1700000 时，我的 128MB 内存的机器就不行了。下一章主要讨论集合的实现。

产生随机整数的排序子集的另一个方法是弄乱一个 n 个元素数组，这个数组包含数值的范围是 0..n-1，然后排序前 m 个元素并输出。第 3.4.2 节中介绍的 Knuth 的算法 P 的主要作用就是弄乱数组 x[0..n-1]。

```
for i = [0, n)
    swap(i, randint(i, n-1))
```

Ashley Shepherd 和 Alex Woronow 观察发现，在这个问题中，只需要搅乱数组前面 m 个元素，其给出了如下的 C++程序：

```
void genshuf(int m, int n)
{   int i, j;
    int *x = new int[n];
    for (i = 0; i < n; i++)
        x[i] = i;
```

```
    for (i = 0; i < m; i++) {
        j = randint(i, n-1);
        int t = x[i]; x[i] = x[j]; x[j] = t;
    }
    sort(x, x+m);
    for (i = 0; i < m; i++)
        cout << x[i] << "\n";
}
```

该算法使用了 n 个字的内存并需要 O(n+m log m)时间，但是如果使用了问题 1.9 中的技术，就能将时间降低到 O(m log m)。可以使用该算法替代程序 2，在程序 2 中，选中的元素放在 x[0..i-1]中，没有选中的元素放在 x[i..n-1]中。通过显式表示未选中的元素，就不需要测试元素以前是否已经选中。不幸的是，由于这个方法需要 O(n)的时间和内存，因此 Knuth 的算法总是占据优势。

到目前为止，我们看到的函数提供了对该问题的多个不同解决方法，但是，毫无疑问它们涉及到了设计空间。例如，假设 n 是 100 万而 m 是 n-10。我们可能产生一个包含 10 个元素的有序随机样本，然后从 n 个元素中找出不在这个样本里的元素。接着，假设 n 为 1000 万，m 为 2^{31}。我们能够产生 1100 万个整数，然后将它们排序，通过扫描删除重复的整数，最后生成一个包含 1000 万个元素的排好序的样本。答案 9 介绍了一种特别聪明的算法，这个算法基于 Bob Floyd 提出的查找方法。

12.4　原则

本章介绍了编写程序的几个重要步骤。虽然下面是按照自然顺序介绍各个阶段的，但是设计阶段更加主动：从一个活动跳跃到另一个活动，并可能需要迭代很多次才能得到一个比较合理的解。

理解观察到的问题。和用户讨论问题出现的环境。通常地，问题描述中就包含了解决方法的基本思想；就跟以前一样，必须考虑它们，但是不能排除其他解决方法。

指定一个抽象问题。一个简洁、明确的问题描述能够帮助我们在解决这个问题的基础上，进一步考虑如何将这个解决方法应用到其他问题上。

利用设计空间。很多程序员一下子就找到了问题的“所谓”解决方法，他们往往宁愿使用一分钟的时间来思考，一天的时间来编码，也不乐意思考一个小时，编码一小时。可以用非正式的高级语言来描述设计：伪码代表控制流，抽象数据类型代表关键的数据结构。理解这一点在设计过程阶段非常重要。

实现解决方法。幸运的时候，通过研究设计空间就能够发现某个程序比其他的要好；而有时则只好从较好的几个解决方法中选择最好的。应该尽量使用简单直白的代码实现

选中的设计，使用可用的最强大操作[3]。

回顾。Polya 的《*How to Solve It*》能够帮助所有的程序员，使他们成为更好的问题解决人员。在该书的第 15 页，他发现“总有事情可做，在深入研究和理解之后，能够改进任何解决方法，并且，在任何一种情况下，都能够进一步理解解决方法。”他的提示对于回顾编程问题非常有用。

12.5 问题

1. 一般 C 库中的 rand()函数大概随机返回 15 个随机位。使用该函数实现函数 bigrand()，使其至少返回 30 个随机位；并实现函数 randint(l,u)，使其返回一个范围在[l,u]中的随机整数。

2. 第 12.1 节规定所有具有 m 个元素的子集都必须具有相同的选中概率，这个条件比每个整数具有 m/n 的选中概率要强。写出一个算法，在该算法中所有的元素具有相同的选中概率，但是某些子集的选中概率要比其他一些大。

3. 证明当 m<n/2 时，在找出不在集合中的元素之前，基于集合的算法进行的成员测试次数小于 2。

4. 在基于集合的程序中，统计成员测试产生了很多组合概率学上的有趣问题。以 m 和 n 的函数来表示，程序平均要进行多少次成员测试？当 m=n 时又会进行多少次呢？什么时候测试次数可能超过 m？

5. 本章介绍了一个问题的多个算法，在本书的网站上提供了这些算法。在你自己的系统上，测试这些算法的性能，并说出每个算法适合使用的运行时间、空间等约束条件。

6. ［课堂练习］在本科生的算法课上，我布置了一道题目，让他们两次生成有序子集。在讲述排序和查找课程单元之前，在学生必须编写 m=20 且 n=400 的程序，基本的打分标准是，程序要简短、清晰——运行时间不是问题。在学习了排序和查找之后，他们必须再次解决同样的问题，此时 m=5000000 且 n=1000000000，主要基于运行时间进行打分。

7. [V. A. Vyssotsky]通常使用递归函数来表示生成组合对象的算法会比较合适。Knuth 的算法可以写成：

---

3 问题 6 介绍了一个课堂练习，我主要按编程风格对此进行打分。多数学生交上来的程序大概有一页，我给了一个比较中等的分数。但是有两个学生在上一个暑假参加了一个大型软件开发项目，他们交上来的程序有五页，书写得非常漂亮，该程序有很多函数，每个函数都有复杂的标题。我给他们不及格。最好的程序只需要 5 行代码，膨胀 60 倍以上的代码就不能及格了。当这两个人跟我抱怨说，他们使用了标准的软件工程工具时，我就引用了 Pamela Zave 的话来回答他们：“软件工程的目的是为了控制复杂度，而不是增加复杂度。”多花几分钟时间来寻找一个简单的程序往往能节省几个小时编写复杂程序的时间。

```
void randselect(m, n)
        pre   0 <= m <= n
        post  m distinct integers from 0..n-1 are
              printed in decreasing order
    if m > 0
        if (bigrand() % n) < m
            print n-1
            randselect(m-1, n-1)
        else
            randselect(m, n-1)
```

该程序按递减顺序输出随机整数。如何才能按照递增顺序输出结果？讨论按此要求所得到的程序的正确性。如何使用该程序的基本递归结构生成0..n-1的所有m个元素的子集？

8. 如何从0..n-1中随机选择m个整数，并且按照随机顺序输出最后的结果？如果列表中允许出现重复的整数，如何才能生成一个有序列表？如果既允许重复又需要按照随机顺序输出结果，那该怎样？

9. [R. W. Floyd]当m接近n时，基于集合的算法会产生很多集合中早就存在的整数，因此需要去掉这些整数。你能不能写出一个算法，即便是在最差情况下，该算法也只需要m个随机数值？

10. 如何随机从n个对象中选择一个对象，这n个对象是按序排列的，但是在此之前你并不知道n的值？具体些说，在事先并不知道行数的情况下，如何读取一个文本文件，随机选择并输出一行？

11. [M. I. Shamos]某个升级游戏有一张卡，卡上有16个点，这其中隐藏了整数1..16的随机排列。玩家将卡上的点擦开以查看里面的整数。如果出现整数3，那么玩家就失败了；如果1和2（不考虑顺序）都出现了，那么玩家就赢了。描述计算随机选择一个点序列而赢得游戏的概率；假设你最多可以使用一个小时的CPU时间。

12. 本章中介绍的那个程序的第一个版本有一个很大的漏洞，当m=0时，程序就死掉了。如果m为其他值，它看起来像是产生了随机结果，但是实际上并不是这样子的。如何测试一个样本生成程序，确保它的输出确实是随机的？

12.6 进阶阅读

Don Knuth 的《*Art of Computer Programming*》的第二卷是《*Seminumerical Algorithms*》。第三版是由Addison-Wesley在1998年出版的。第3章（本书的前半部分）是关于随机数的，第4章（后半部分）主要介绍算法。第3.4.2节的“随机取样和混合”和本章关系特别密切。如果你想要创建一个随机数生成器或是执行高级算法的函数，你就必须阅读这本书。

第13章 查 找

很多方面都涉及到查找问题。编译器通过查找某个变量名，从而找到它的类型和地址。拼写检查器通过查看字典来判断单词拼写是否正确。目录程序通过查找订阅者的名字找到他的电话号码。Internet 域名服务器通过名字找到它的 IP 地址。这些应用程序以及很多类似的程序都需要通过查找一组数据找到和特定项相关的数据。

本章主要深入研究这样一个查找问题：在不存储其他相关数据的情况下，如何存储一组整数？虽然这个问题很小，但它却引发了很多数据结构实现中的关键问题。我们先从明确定义任务开始，并使用它来研究最常见的集合表示。

13.1 接口

这里接着讨论上一章的问题：生成一个范围在[0,maxval]之间的 m 个随机整数的有序序列，不存在重复的整数。我们的任务是实现下列伪码：

```
initialize set S to empty
size = 0
while size < m do
    t = bigrand() % maxval
    if t is not in S
        insert t into S
        size++
print the elements of S in sorted order
```

我们将这个数据结构叫做 IntSet，这是个整数集合。我们将这个接口定义成具有以下公共成员的 C++类：

```
class IntSetImp {
public:
    IntSetImp(int maxelements, int maxval);
    void insert(int t);
    int size();
    void report(int *v);
};
```

构造函数 IntSetImp 将集合初始化为空。该函数有两个参数，这两个参数分别表示集合中的最大的元素数量和每个元素的最大值，在特定的实现中可以忽略其中一个参数，也

可以两个都忽略。insert 函数向集合中添加一个新元素（如果这个元素本来不在集合中）。size 函数说出当前元素数量，report 函数将元素写入向量 v 中（按顺序写入）。

很明显，这个小接口仅具有学习意义，其缺少对一个工业强度的类来说很关键的构成部分，如错误处理程序和析构函数。熟练的 C++程序员可能会使用带有虚函数的抽象类指定这个接口，然后将每个实现写成派生类。我们将采用更加简单的方法（有时，它的效率更高），使用 IntSetArr 这类名字作为数组的实现，IntSetList 作为列表的实现，等等。并使用名字 IntSetImp 代表任意实现。

在这个函数中，该 C++代码使用这样的数据结构来生成一个随机整数的有序集合：

```
void gensets(int m, int maxval)
{   int *v = new int[m];
    IntSetImp S(m, maxval);
    while (S.size() < m)
        S.insert(bigrand() % maxval);
    S.report(v);
    for (int i = 0; i < m; i++)
        cout << v[i] << "\n";
}
```

由于 insert 函数并不会在集合中放入重复的元素，因此不需要在插入之前测试元素是否在集合中。

最简单的 IntSet 实现使用强大和通用的 C++标准模板库（Standard Template Library，STL）中的 set 模板：

```
class IntSetSTL {
private:
    set<int> S;
public:
    IntSetSTL(int maxelements, int maxval) { }
    int size() { return S.size(); }
    void insert(int t) { S.insert(t); }
    void report(int *v)
    {   int j = 0;
        set<int>::iterator i;
        for (i = S.begin(); i != S.end(); ++i)
            v[j++] = *i;
    }
};
```

构造函数忽略了它的两个参数。IntSet、size 和 insert 函数在 STL 中有与之相对应的部分。report 函数使用标准迭代器将集合元素按序写入数组。这个通用结构不错，但不是很好。下面马上就可以看到实现是决定这个特定任务是否更富效率的时间和空间的五个因素之一。

13.2 线性结构

下面将使用最简单的结构建立第一个集合实现：一个整型数组。我们的类将当前的元素数存放在整数 n 中，而将整数本身存放在向量 x 中：

```
private:
    int n, *x;
```

（附录 5 中给出了所有类的完整实现。）该 C++构造函数的伪码版本分配了数组（另有一个元素用作标记）并将 n 赋为 0：

```
IntSetArray(maxelements, maxval)
    x = new int[1 + maxelements]
    n = 0
    x[0] = maxval
```

由于必须按序输出元素，我们准备总是按这种方式存储它们（在其他一些应用程序中，更适合使用无序数组），并将标记元素 maxval 放置在有序元素的最后；maxval 大于集合中的任何一个元素。这就使用查找更大元素（我们早就需要了）这样一个测试替换对是否运行到列表结尾处的测试。这将简化插入代码，并能够提高它的速度：

```
void insert(t)
    for (i = 0; x[i] < t; i++)
        ;
    if x[i] == t
        return
    for (j = n; j >= i; j--)
        x[j+1] = x[j]
    x[i] = t
    n++
```

第一个循环扫描出小于插入值 t 的数组元素。如果新元素等于 t，则说明它早就在集合中，因此马上返回。否则，将较大的元素（包括标记元素）移动一位，将 t 插入到结果空位中，并将 n 加一。这需要 O(n)时间。

所有实现中的 Size 函数都相同：

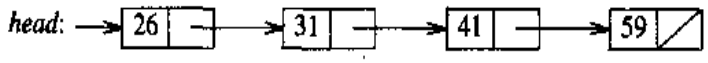
```
int size()
    return n
```

report 函数在 O(n)时间内将所有元素（除了标记元素）复制到输出数组中：

```
void report(v)
    for i = [0, n)
        v[i] = x[i]
```

如果事先知道集合的大小，那么数组这种结构用来比较适合实现集合。因为数组是有序的，所以能用二分查找建立一个运行时间为 O(log n)的成员函数。在本节结尾部分将详细讨论数组的运行时间。

如果事先不知道集合的大小，那么链表将是表示集合的首选；链表消除了插入时元素移动的成本。



类 IntSetList 将使用该私有数据：

```
private:
    int n;
    struct node {
        int val;
        node *next;
        node(int v, node *p) { val = v; next = p; }
    };
    node *head, *sentinel;
```

链表中的每个结点都具有一个整型值和一个指向链表中下一结点的指针。Node 构造函数将其两个参数的值赋给那两个字段。

跟有序数组的原理一样，链表也必须有序。就像在数组中一样，链表将使用标记结点，它的值大于所有真实的值。构造函数建立这样一个结点，并让 head 指向它。

```
IntSetList(maxelements, maxval)
    sentinel = head = new node(maxval, 0)
    n = 0
```

report 函数遍历链表，将排好序的元素放在输出向量中：

```
void report(int *v)
    j = 0
    for (p = head; p != sentinel; p = p->next)
        v[j++] = p->val
```

为了在有序链表中插入一项，必须遍历整个链表，直到找到那个元素（并且马上返回），或找到一个更大的值，在该点插入这个值。不幸的是，情况的多样化总是导致结点更为复杂，参见答案 4。我知道的完成这个任务的最简单的代码是一个递归函数，原来叫做：

```
void insert(t)
    head = rinsert(head, t)
```

递归部分非常清晰：

```
node *rinsert(p, t)
    if p->val < t
        p->next = rinsert(p->next, t)
    else if p->val > t
        p = new node(t, p)
        n++
    return p
```

当编程问题隐藏在众多特殊情况下，使用递归通常能够简化代码，如上所示。

当使用上面两个结构生成 m 个随机整数时，平均来说，m 个查找中的每一次运行时间和 m 成正比。因此，每个结构的总时间和 m^2 成正比。我认为链表版本要比数组版本相对快一些：它使用了额外的空间（指针用的）来避免超出数组的上界。下面是 n 为 1000000，m 为 10000 到 40000 之间时的运行时间：

结构	集合大小（m）		
	10000	20000	40000
数组	0.6	2.6	11.1
简单链表	5.7	31.2	170.0
链表（消除递归）	1.8	12.6	73.8
链表（组分配）	1.2	5.7	25.4

跟我所预计的一样，数组的运行时间成平方数增加，带有一个非常合理的常量。但是我实现的第一个链表，其开始时的增长幅度小于数组，但后来增长得比 n^2 要快。肯定有哪个地方存在问题。

我的第一个反应就是将问题归结于递归。除了递归调用的开销外，rinsert 函数的递归深度就是找到元素的位置，为 O(n)。在整个递归中，代码将原来的值赋回给几乎所有的指针。当我将递归函数转换成答案 4 中介绍的迭代版本时，运行时间几乎降低了 3 倍。

我的下一个反应就是使用问题 5 中的技术改变存储分配：构造函数分配了一个具有 m 个结点的单个块，insert 根据需要获取它们，而不是为每个插入分配一个新的结点。这在下面两个不同的方面都是很大的改进：

附录 3 中的时间成本模型表明跟简单的操作相比，存储分配所消耗的时间要高出两个数量级。我们使用一个操作替换掉了 m 个昂贵的操作。

附录 3 中的空间成本模型表明，如果将结点作为块来分配，每个结点只需消耗 8 个字节（4 个用于整数，4 个用于指针）：40000 个结点消耗 320 个千字节，我机器的二级缓存正好够用。然而，如果单独分配结点，则每个结点就要消耗 48 个字节，它们总共要消耗 1.92 兆字节，这超出了二级缓存的容量。

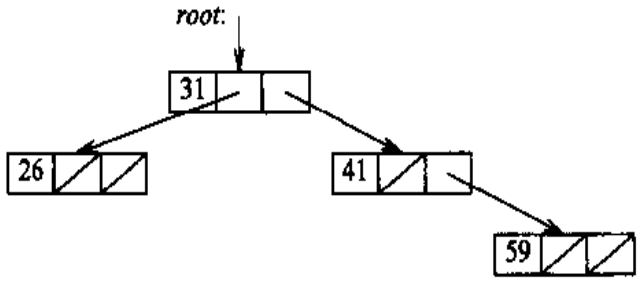
在具有更高效率的分配器的不同系统中，消除递归能够将加速系数变为 5，但是当使用单独分配仅能减少 10%。类似于很多代码优化技巧，缓存和递归消除有时会带来很多好处，但是有的时候却没什么用处。

数组插入算法查找整个队列，以找到插入目标值的合适位置，然后再压入更大的值。链表插入算法只完成了第一部分，并没有完成第二部分。所以，如果链表只是完成了一

半工作，那为什么它需要两倍的时间呢？部分原因是它需要两倍的内存：大链表必须将8个字节的结点读入高速缓存以便来访问4个字节的整数。另一部分原因是，可以很容易地预测数组对数据的访问，而链表的访问模式则可能占用所有的内存。

13.3 二分查找树

下面要从线性结构转向支持快速查找和插入的结构。下图是插入整数31、41、59和26之后的二分查找树，顺序为：



IntSetBST类定义了结点和根：

```
private:
    int n, *v, vn;
    struct node {
        int val;
        node *left, *right;
        node(int i) { val = i; left = right = 0; }
    };
    node *root;
```

初始化该树的时候将根设为空，并通过调用递归函数执行其他操作。

```
IntSetBST(int maxelements, int maxval) { root = 0; n = 0; }
void insert(int t) { root = rinsert(root, t); }
void report(int *x) { v = x; vn = 0; traverse(root); }
```

插入函数遍历这棵树，直到找到所需的值（查找结束），或在整棵树中没有找到所需的值（插入结点）：

```
node *rinsert(p, t)
    if p == 0
        p = new node(t)
        n++
    else if t < p->val
        p->left = rinsert(p->left, t)
    else if t > p->val
        p->right = rinsert(p->right, t)
    // do nothing if p->val == t
    return p
```

由于应用程序中的元素是按随机顺序插入的，所以不用担心复杂的平衡方法（问题1表明随机集合上的其他算法会导致树的极大不平衡）。

inorder遍历[1]首先处理左子树，然后输出结点本身，接着处理右子树：

```
void traverse(p)
    if p == 0
        return
    traverse(p->left)
    v[vn++] = p->val
    traverse(p->right)
```

它使用变量vn来索引向量v中下一个可用元素。

这张表给出了在第13.1节中看到的STL set结构、二分查找树以及下一节中要介绍的其他几个结构的运行时间（这是我机器上运行的情况）。最大的整数为$n=10^8$，m可以尽可能地大，直到系统内存不够而必须使用磁盘时为止。

结 构	集合大小（m）					
	1000000		5000000		10000000	
	秒	MB	秒	MB	秒	MB
STL（标准模板库）	9.38	72				
二分查找树	7.30	56				
二分查找树*	3.71	16	25.26	80		
桶	2.36	60				
桶*	1.02	16	5.55	80		
位向量	3.72	16	5.70	32	8.36	52

这些时间都没有包含输出结果的时间，包含了输出结果后的时间要略微大于STL实现所需的时间。简单的二分查找树避免了STL所使用的复杂的平衡方法（STL规范能够保证在最差情况下获得较好的性能），因此，它会稍微快一些，同时使用的空间也更少一些。当m=1600000时，STL就开始崩溃了，而第一个BST则在m=1900000时崩溃。标为“BST*”的行介绍了结合几个优化后的二分查找树运行情况。最重要的是它一下子就分配所有的结点（如问题5）。这就大大降低了树的空间需求，运行时间大致降低了三分之一。该代码也将递归转化为迭代（如问题4），同时使用了问题7中介绍的标记结点，这些大概能提速25%。

1 我的这个程序的第一个版本有一个奇怪的bug：编译器报告内部不一致并死掉了。我关闭了优化，这个“bug”就没有了，然后我就责怪写优化器的人。后来我发现，问题出在，当我快速编写遍历代码时，我忘了对p进行是否为null的if判断。优化器尽量将尾递归转化为循环，当不能找到一个判断终止循环时就会死掉。

13.4　整数结构

下面来介绍最后两个现实中常用的结构，它们利用集合来代表整数。位向量是第 1 章的老内容了，下面是它们的私有数据和函数：

```
enum { BITSPERWORD = 32, SHIFT = 5, MASK = 0x1F };
int n, hi, *x;
void set(int i)  {       x[i>>SHIFT] |=  (1<<(i & MASK)); }
void clr(int i)  {       x[i>>SHIFT] &= ~(1<<(i & MASK)); }
int  test(int i) { return x[i>>SHIFT] &   (1<<(i & MASK)); }
```

构造函数分配数组并关闭所有位：

```
IntSetBitVec(maxelements, maxval)
    hi = maxval
    x = new int[1 + hi/BITSPERWORD]
    for i = [0, hi)
        clr(i)
    n = 0
```

问题 8 表明通过一次操作一个字的数据也许能够提高这个速度。在 report 函数中进行类似的提速：

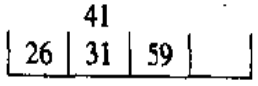
```
void report(v)
    j = 0
    for i = [0, hi)
        if test(i)
            v[j++] = i
```

最后，insert 函数打开位并增加 n，但是只有在这个位以前是关闭的情况下才能这样做：

```
void insert(t)
    if test(t)
       return
    set(t)
    n++
```

上一节的表表明如果最大值 n 足够小，使得位向量能融入主存中，那么这个结构的效率就非常高（问题 8 说明了如何更好地提高效率）。不幸的是，如果 n 是 2^{32}，位向量需要 0.5GB 的主存。

最后的数据结构将链表和位向量的优点结合起来了。它在桶序列中放入整数。如果在范围 0..99 中有四个整数，就将它们放在四个桶中。桶 0 包含范围 0..24 内的整数，桶 1 代表了 25..49，桶 2 代表了 50..74，桶 3 代表了 75..99：



可以将 m 桶视为散列。每个桶中的整数由有序链表来表示。由于整数的分布均匀，所以

每个链表的长度都为 1。

该结构具有下面的私有数据：

```
private:
    int n, bins, maxval;
    struct node {
        int val;
        node *next;
        node(int v, node *p) { val = v; next = p; }
    };
    node **bin, *sentinel;
```

构造函数使用大值来分配桶数组和标记元素，并初始化每个桶，让它们指向标记元素：

```
IntSetBins(maxelements, pmaxval)
    bins = maxelements
    maxval = pmaxval
    bin = new node*[bins]
    sentinel = new node(maxval, 0)
    for i = [0, bins)
        bin[i] = sentinel
    n = 0
```

insert 函数需要将整数 t 放入合适的桶中。明显的映射 t * bins/maxval 将导致数值溢出（就我个人的痛苦经验来说，这是糟糕的调试）。在该段代码中我们将使用更安全的映射：

```
void report(v)
    j = 0
    for i = [0, hi)
        if test(i)
            v[j++] = i
```

rinsert 类似于链表。类似的，report 函数实际上是按顺序应用在每个桶上的链表代码：

```
void report(v)
    j = 0
    for i = [0, bins)
        for (node *p = bin[i]; p != sentinel; p = p->next)
            v[j++] = p->val
```

最后一节中的表说明桶很快。标有“桶*”的行描述了修改后的桶在初始化过程中分配所有结点的运行时间（如问题 5）；修改后的结构大概使用四分之一的空间和初始时间的一半，删除递归后运行时间又降低了 10%。

13.5 原则

前面已经介绍了 5 种表示集合的重要数据结构。当 m 相对 n 来说很小时，这些结构的平均性能如下表所示（b 表示每个字的位数）：

集合表示	O（每个操作的时间）			总时间	空间（字）
	Init	Insert	report		
有序数组	1	m	m	$O(m^2)$	m
有序链表	1	m	m	$O(m^2)$	2m
二分树	1	log m	m	O(mlog m)	3m
桶	m	1	m	O(m)	3m
位向量	n	1	n	O(n)	n/b

这张表仅仅涉及到集合表示这类问题的表面。答案 10 提到了其他可能的方法。第 15.1 节介绍了查找单词集合的数据结构。

虽然本章主要介绍了集合表示的数据结构，但是我们已从中学习到了许多编程中都会应用到的基本原理。

库的作用。C++标准模板库提供了一个很容易实现、维护及扩展的通用解决方法。当你面临的问题涉及到数据结构时，你的第一个反应应该是去查找解决问题的通用工具。然而，在本例中，专用代码将充分利用特定问题所具备的特征，加快运行速度。

空间重要性。第 13.2 节中我们看到优化得很好的链表完成数组的一半工作，但是需要两倍的时间。为什么？因为数组中每个元素只需要使用一半的内存，并顺序访问内存。在第 13.3 节中，可以看到在二分查找树中使用定制的内存分配，将空间减小了 3 倍，时间减小了 2 倍。当超过 0.5MB（我的机器上的二级高速缓存）的内存边界，并接近 80MB（空闲 RAM 量）时，运行时间急剧增加。

代码优化技巧。最大的改进就是通过单独分配一个大块来替换通用内存分配。这就消除了很多昂贵的调用，并能更高效地利用空间。通过重写递归函数，将它变成迭代函数，对链表来说，其加速系数为 3，但是对于桶，只能加速 10%。多数结构通过标记元素能够得到清晰简单的代码，偶尔也会减少运行时间。

13.6 问题

1. 答案 12.9 介绍了生成随机整数的有序集合的 Bob Floyd 算法。你能否通过本节的 IntSets 实现该算法？这些结构在 Floyd 算法生成非随机分布时，其执行情况如何？

2. 如何更改 IntSet 接口使得它更强壮？

3. 使用 find 函数扩充集合类，find 函数主要用来判断给定的元素是否在集合中。能不能让该函数比 insert 的效率更高？

4. 将链表、桶和二分查找树的递归插入函数改写成迭代函数，并测量它们之间在运行时间上的差别。

5. 第9.1节和答案9.2介绍了Chris Van Wyk是如何将可用结点保留在自己的结构中从而避免很多对存储分配器的调用的。考虑一下如何将这一原理应用到链表、桶和二分查找树实现的IntSets上。

6. 在各种IntSet实现上，通过对下面的段进行计时，你能够学习到什么？

```
IntSetImp S(m, n);
for (int i = 0; i < m; i++)
    S.insert(i);
```

7. 我们的数组、链表和桶都使用了标记元素。说明一下如何将这一技巧用于二分查找树。

8. 证明一下如何通过同时在很多位上进行并行操作来加速位向量的初始化和输出操作。这在操作char、short、int、long或其他一些单元时是不是最高效的？

9. 证明如何通过使用廉价的逻辑位移替代昂贵的除操作来加速桶。

10. 如何使用其他数据结构表示上下文中的整数集合，就像生成随机整数一样？

11. 建立一个最快的完整函数来生成没有重复的随机整数的有序数组（可以使用前面介绍的任何接口来表示集合）。

13.7 进阶阅读

第11.6节介绍了Knuth和Sedgewick的一本优秀的算法书。查找是Knuth的《*Sorting and Searching*》第6章（第2部分）的主题，也是Sedgewick的《*Algorithms*》的第4部分（最后一部分）的主题。

13.8 实际查找问题［补充内容］

本章中的小型结构给我们提供了研究严谨的数据结构的基础。本部分研究了Doug McIlroy于1978年写的spell程序中用于表示字典的著名结构。在1980年初写本书这章内容时，我使用McIlroy的程序对它们进行了拼写检查。在本版中，我再次使用了spell，发现这仍然是一个非常有用的工具。在《*IEEE Transactions on Communications*》COM-30第1期（1982年1月，第91～99页）“Development of a spelling list”一文中可以找到McIlroy程序的详细内容。我的字典将珍珠一词定义成“正确的选择或非常珍贵”，这个程序是合乎这个标准的。

McIlroy面对的第一个问题是组成单词列表。他首先将没有缩减的字典（为了有效性）和有百万个单词的Brown Univerisity corpus（为了通用性）相交。这是一个很合理

的开端，但是仍然需要完成很多工作。

专有名词查询充分说明了 McIlroy 的方法，而多数字典都没有考虑到这一点。首先考虑人名：大型电话簿中最常见的 1000 个姓，一组男孩和女孩的名字，名人（如 Dijkstra 和 Nixon），以及 Bulfinch 目录中虚构的名字。观察到 Xerox 和 Texaco 这类拼写错误后，他增加了财富 500 列表中的公司名。引用资料中有大量出版社的名字，所以没有包含它们。接着考虑地理名：国家和它们的首都，州和它们的首府，美国和世界上前 100 个最大的城市，同时也不要忘了海洋、行星和星球。

他还添加了动植物的常用名，以及化学、解剖学和（用于本地消耗）计算中的专有名词。但是他很注意，尽量不增加太多：他没有考虑生活中误拼的但有效的单词（如地质术语 cwm），并且仅仅包含了几个可选择的拼写中的一个（因此是 traveling 而不是 travelling）。

McIlroy 的技巧在于查看实际运行时的 spell 的输出。有些时候，spell 自动将输出复制一份发送给他（过去对隐私和效力的看法与现在不同）。当他找到问题所在时，他将使用最广泛的解决方法。最终的结果有 75000 个单词：它几乎包含了文档中所有的单词，但是仍然发现了我的拼写错误。

程序使用词缀分析从单词中去除前后缀。这样做很有必要并且也非常方便。有必要是因为英语单词列表中没有这些东西，拼写检查器必须猜测这是 misrepresented 这类词的派生还是一大堆有效英文单词的拼写错误。词缀分析具有很好的副作用，那就是减小了词典的容量。

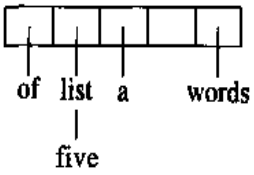
词缀分析的目标是将 misrepresented 降为 sent，剥去 mis-、re-、pre-和-ed（虽然 represent 并不表示“再次出现”，present 的含义并不是“实现发送”，spell 使用重合降低词典的容量）。程序的表中包含 40 个前缀规则和 30 个后缀规则。1300 个例外终止的“停止列表”能够终止良好但不正确的猜测，如将 entend（intend 的误拼）变为 en-+tend，这种分析就将 75000 个单词列表分为 30000 个单词。McIlroy 的程序在每个单词上循环，剥去前后缀并查看结果，直到它找到匹配的单词或没有任何前后缀了（这个单词就是错误的）。

封底分析表明了将字典保存在内存中的重要性。对于 McIlroy 来说这一点更为困难，因为最初的时候他是在 PDP-11 上开发该程序的，仅有 64KB 的地址空间。其文章的摘要概述了他的空间压缩：“去除前后缀将列表的大小变为原来大小的三分之一，通过散列又去掉了剩下的 60%的位，然后再使用数据压缩又压缩了一半。”因此，只需要 26000 个 16 位计算机字就能表示 75000 个英语单词（有很多原型）。

McIlroy 通过散列使用 27 位表示 30000 个英语单词中的每个单词（后面将会介绍为什么使用 27）。下面以小型单词列表为例，展示该方案：

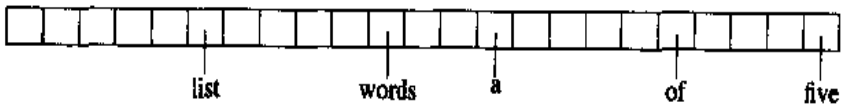
a list of five words

第一个散列方法使用 n 个元素的散列表，大致等于列表的大小，并使用了一个散列函数，该函数将一个字符串映射到[0, n)范围中的一个整数(在第 15.1 节中将看到一个字符串的散列函数)。表的第 i 项指向一个链表，该链表包括散列到 i 的所有字符串。如果空单元表示空列表，并且散列函数生成 h(a)=2，h(list)=1，等等，那么 5 个元素的表就如下所示：



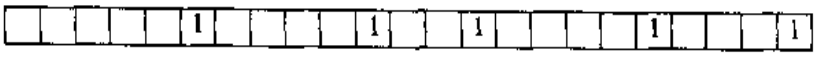
为了查找单词 w，我们对第 h(w)个单元指向的列表执行顺序查找。

下一个方案使用了一张更大的表。n=23 后，使得绝大多数的散列单元仅仅包含一个函数成为可能。在本例中，h(a)=13 并且 h(list)=5。



spell 程序中 $n=2^{27}$（大约等于 134 百万），几乎所有的非空列表都仅仅包含单个元素。

下一步非常大胆：McIlroy 在每个表项中仅仅存放了单个位，并没有使用单词链表。这就大大减少了空间，但是非常容易出错。这张图和前例使用了相同的散列函数，并使用空单元表示零位。



为了查找单词 w，程序访问了表中的第 h(w)位。如果该位为 0，那么程序就正确地报告说单词 w 不在表中。如果该位为 1，程序就认为 w 在表中。有的时候，不好的单词碰巧散列到了有效位，但是出现这种错误的概率仅有 $30000/2^{27}$ 或大约等于 1/4000。因此，平均来说，每 4000 个不良单词就有一个漏网，被认为是有效的。McIlroy 观察到，典型的草稿的错误很少超过 20 个，所以，每一百次中最多出现一次这类错误——这就是他选择 27 的原因。

使用 $n=2^{27}$ 位的字符串表示散列表将消耗 1600 万字节。因此，程序仅仅表示了一个位。在上面的例子中，他存储了下列散列值：

5 10 13 18 22

如果 h(w)存在，那么单词 w 就在表中。这些值的表示需要 30000 个 27 位，但是 McIlroy 的机器的地址空间中仅有 32000 个 16 位。因此，他排序列表，并使用了长度可变的代码来表示后续散列值之间的区别。假设从值 0 开始，上面的列表就压缩成：

5 5 3 5 4

McIlroy 的 spell 平均每个使用了 13.6 位来表示区别。这就剩下几百个额外的单词指向压缩列表中的有用的起始位置，因此能够加速顺序查找。结果是，一个 64KB 的字典能快速访问并且很少出错。

前面早就考虑了 spell 两个方面的性能：它输出有用的结果，并能存放在 64KB 的地址空间中。它的速度也非常快。即便在第一次编译它的老机器上，它也能在半分钟内检查 10 页文档的拼写，而与本书厚度差不多的一本书检查起来也只需要 10 分钟（在那时候这看来是非常快的）。因为能够从磁盘上快速阅读小字典，所以在几秒内就能完成单个单词的拼写检查。

第14章 堆

本章主要介绍“堆”，我们可以使用堆这个数据结构解决两个重要的问题。

排序。通过堆排序来排序 n 个元素的数组所需的时间肯定不会超过 O(n log n)，并且它只需要使用几个字的额外空间。

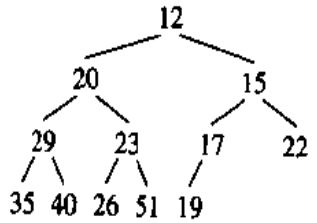
优先队列。堆通过在集合中插入新元素或从集合中提取最小的元素来维护元素集合。每个操作所需的时间为 O(log n)。

使用堆处理这两个问题时，编码非常简单并且计算效率很高。

本章按照自底而上的结构介绍：首先介绍细节，然后介绍总的框架。接着两节介绍了堆数据结构和对其进行操作的两个函数。随后的两节使用这些工具解决上面提到的问题。

14.1 数据结构

堆是用来表示元素集合的一种数据结构[1]。例子中的堆将用来表示数值，但是堆中的元素可以是任何有序类型。下面是由 12 个整数构成的堆：

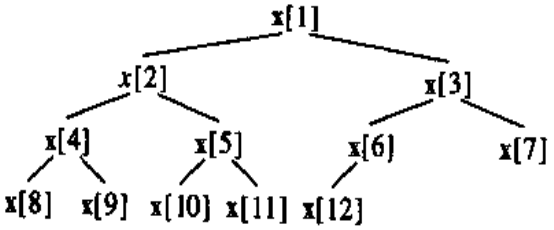


从根本上来说，由该二叉树的两个属性可以判断出它是一个堆。第一个属性是*顺序*：任何结点的值都小于或等于其子结点的值。这就意味着集合的最小元素就是树根（本例中为 12），但是它没有说明左右子结点的相对大小。第二个属性是*形状*，如下所示

1 在其他计算环境中，“堆”是指能够分配可变大小的结点的一块很大的内存段，在本章中将忽略这个定义。

总的来说，具有上面形状属性的二叉树至多在两层上具有终止结点，最底层终止结点尽量靠左。树中没有“洞”，因而如果它包含 n 个结点，就没有哪个结点和根之间的距离会超过 $\log_2 n$。下面将介绍这两个属性是如何具有足够的限制性以帮助我们找到集合中的最小元素，同时其结构也非常松散，可以使我们在插入或删除元素之后高效地重新组织结构。

下面将注意力从堆的抽象属性转向它们的实现。二叉树主要用于表示记录和指针。我们将实现仅具有形状属性的二叉树，但是在这种特殊情况下，其效率很高。使用 12 个元素的数组 x[1..12]表示具有形状属性的一棵 12 个元素的树，如下所示：

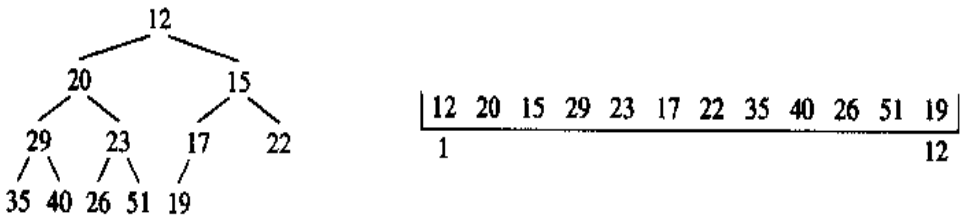


注意，堆使用的是从下标 1 开始的数组；C 中最简单的方法就是声明 x[n+1]并浪费元素 x[0]。这是二叉树的隐式表示，根位于 x[1]，它的两个子结点是 x[2]和 x[3]，等等。树的典型函数定义如下所示：

```
root = 1
value(i) = x[i]
leftchild(i) = 2*i
rightchild(i) = 2*i+1
parent(i) = i / 2
null(i) = (i < 1) or (i > n)
```

n 个元素的隐形树必须具有形状属性：它不为缺少的元素做任何准备。

该图显示了一个 12 个元素的堆和它的实现，这是通过包含 12 个元素的数组的隐形树实现的。



由于形状属性是通过表示方法来保证的，从现在开始，“堆”这个词就意味着任何结点的值都大于或等于其父结点的值。精确些说，如果

$$\forall_{2 \le i \le n}\ x[i/2] \le x[i]$$

那么数组 x[1..n]就具有堆的属性。

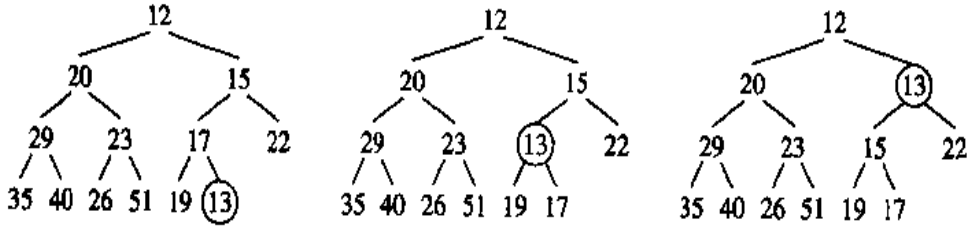
记住，“/”整除操作符会取整，所以 4/2 和 5/2 都得到 2。在下一节将介绍具有堆属性的子数组 x[l..u]（它是一个早就具有形状属性的变量）；可以通过数学方法定义 heap(l,u)，如下所示：

$\forall_{2l \le i \le u}\ x[i/2] \le x[i]$

14.2 两个关键函数

本节主要研究两个函数，这两个函数主要用于修补在一端或另一端中断的堆属性。这两个函数的效率都很高：它们大概需要 log n 步来重新组织具有 n 个元素的堆。由于本章讲述的风格是自底而上的，因此将在这里定义函数，然后在下一节使用它们。

当 x[1..n-1]是堆时，在 x[n]上随便放一个元素可能不能产生 heap(1,n)；重新建立该属性是函数 siftup 的任务。它的名字就描述了它的策略：尽可能将新元素上移到树中，不断将它和它的父结点进行交换(本节将使用典型的堆定义来确定通过哪条路是向上移：堆的根为 x[1]，位于树的顶部，因此，x[n]在数组的底部)。下图演示了这个过程（从左到右），显示了新元素 13 在树中上移直到移到合适的位置成为根的右子结点的整个过程。



该过程一直持续到带圈的结点大于或等于它的父结点（如本例所示），或它已位于树的根部。如果过程开始时 heap(1,n-1)为真，那么 heap(1,n)为真。

有了这个直观的背景知识后，下面可以开始编写代码了。由于筛选移位过程调用了一个循环，所以从循环不变式开始。在上图中，除了带圈的结点和它的父结点之间外，树的所有地方都保留了堆属性。如果 i 是带圈结点的下标，那么就可以使用不变式：

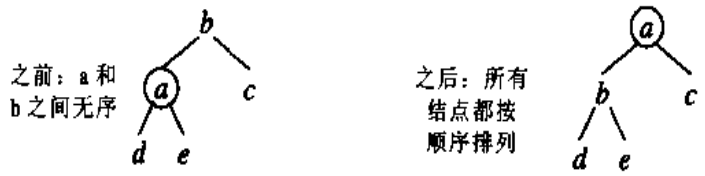
```
loop
    /* invariant: heap(1, n) except perhaps
          between i and its parent */
```

由于开始时具有堆 heap(1,n-1)，因此可以通过赋值语句 i=n 初始化循环。

循环必须检查有没有结束（带圈的结点要么位于堆的顶部，要么大于或等于它的父结点），如果没有结束，则让过程继续。不变式表明除了 i 和它的父结点之间外，其他地方都满足堆属性。如果 i==1 为真，那么 i 没有父结点且所有地方都满足堆属性，因此可以终止循环。当 i 没有父结点时，可以通过赋值语句 p=i/2 让父结点的下标为 p。如果 x[p]

≤x[i]，那么所有地方都满足堆属性，并且循环可以终止。

另一方面，如果 i 和它的父结点之间顺序不对，那么可以交换 x[i]和 x[p]。下图显示了该步骤，其中的关键字是单个字母，结点 i 带有圈。



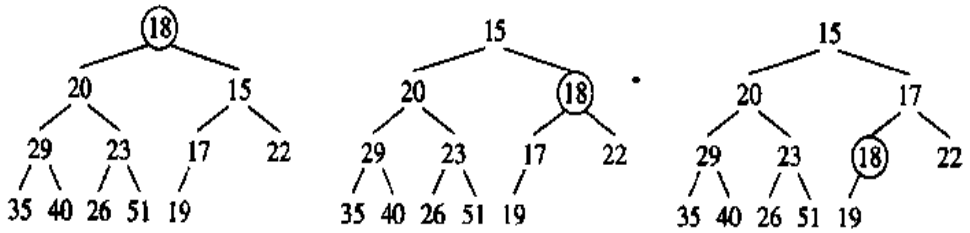
交换之后，5 个元素的顺序都是正确的：b<d 并且 b<e，因为 b 原来位于堆[2]的最上面；a<b，因为不满足条件 x[p]≤x[i]；结合了 a<b 和 b<c 之后，得到 a<c。这就确保了除了 p 和它的父结点之外，其他地方都满足堆属性，因此通过赋值语句 i=p 就能重新获得不变式。

这个过程显示在 siftup 代码中，它的运行时间和 log n 成正比，因为堆具有 log n 层。

```
void siftup(n)
        pre  n > 0 && heap(1, n-1)
        post heap(1, n)
    i = n
    loop
        /* invariant: heap(1, n) except perhaps
                between i and its parent */
        if i == 1
            break
        p = i / 2
        if x[p] <= x[i]
            break
        swap(p, i)
        i = p
```

在第 4 章中，“pre”和“post”行特征化该函数：如果在函数调用之前前序条件为真，那么函数返回之后，后续条件也为真。

现在从 siftup 转向 siftdown。当 x[1..n]是一个堆时，给 x[1]分配一个新值得到 heap(2,n)；函数 siftdown 使得 heap(1,n)为真。它将 x[1]下移，直到它没有子结点或小于/等于它的子结点。下图给出了 18 在堆中下移直到最后小于它的单个子结点 19 的整个过程。



2 在循环不变式中没有说明这个重要的属性。Don Knuth 观察到，也为了更加精确，应该加强不变式使其变为“如果 i 没有父结点，就保存 heap(1,n)；否则，如果 x[i]被 x[p]替换，那么也将保存 heap(1, n)，其中 p 是 i 的父结点”。在稍后的 siftdown 循环中，也要做类似的准备。

当一个元素上移时，它总是移向根。下移比较复杂：将无序的元素和比它小的子结点交换。

该图显示了 siftdown 循环的不变式：除了带圈结点和它的子结点之间可能不满足堆属性外其他部分都符合。

```
loop
    /* invariant: heap(1, n) except perhaps between
          i and its (0, 1 or 2) children */
```

这个循环和 siftup 的非常相似。首先检查 i 是否具有子结点，如果它没有子结点就终止循环。下面就涉及到比较微妙的部分了：如果 i 确实具有子结点，那么将设置变量 c 指向比 i 更小的子结点。最后，或者满足 x[i]≤x[c]终止循环，或者交换 x[i]和 x[c]并赋值 i=c 继续进行循环，直到底部为止。

```
void siftdown(n)
        pre   heap(2, n) && n >= 0
        post  heap(1, n)
    i = 1
    loop
        /* invariant: heap(1, n) except perhaps between
              i and its (0, 1 or 2) children */
        c = 2*i
        if c > n
            break
        /* c is the left child of i */
        if c+1 <= n
            /* c+1 is the right child of i */
            if x[c+1] < x[c]
                c++
        /* c is the lesser child of i */
        if x[i] <= x[c]
            break
        swap(c, i)
        i = c
```

如上对 siftup 所做的案例分析表明交换操作保证除了 c 和它的子结点外其他所有地方的堆属性都为真。类似于 siftup，这个函数所需时间和 log n 成正比，因为它在堆的每层都要完成固定数量的工作。

14.3 优先队列

每个数据结构都有两面。从外部来看，它的*规范*说明了它所完成的工作——队列通过 insert 和 extract 操作维护元素序列。从内部来看，它的实现说明了它是如何完成工作的——可以使用数组或链表来实现队列。在研究优先队列之前，首先指定它们的抽象属

性，然后再研究它们的实现。

优先队列操作一个初始为空[3]的元素集合，该集合叫做 S。insert 函数在集合中插入一个新元素，可以在前后序条件中精确定义它们。

```
void insert(t)
        pre   |S| < maxsize
        post  current S = original S ∪ {t}
```

函数 extractmin 删除集合中的最小元素并通过单个参数 t 返回该值。

```
int extractmin()
        pre   |S| > 0
        post  original S = current S ∪ {result}
                && result = min(original S)
```

当然，可以修改这个函数以产生最大元素，或总顺序中的任何极值。

可以使用模板为该任务指定一个 C++类，该模板指定了队列中元素的类型 T：

```
template<class T>
class priqueue {
public:
    priqueue(int maxsize);  // init set S to empty
    void insert(T t);       // add t to S
    T extractmin();         // return smallest in S
};
```

在许多应用程序中优先队列都非常有用。操作系统可以使用这样一个结构来表示一组任务：按任意顺序插入它们，然后进行提取：

```
priqueue<Task> queue;
```

在单个事件模拟中，元素就是事件的时间：模拟循环提取下一个事件并且可能在队列中添加更多的事件：

```
priqueue<Event> eventqueue;
```

在两个应用中，必须使用集合中元素之外的其他信息扩展基本的优先队列。在下面的讨论中将忽略“实现细节”，但是 C++类通常能很好地处理它们。

很明显，比较适合使用数组或链表这类顺序结构来实现优先队列。如果队列是有序的，那么提取最小值的操作就非常简单，但是很难插入一个新元素。而在没有排过序的结构中情况则相反。下表显示了有 n 个元素的集合上的各个结构的性能。

虽然二分查找能够在 O(log n)时间内找到一个新元素的位置，但是移动原来的元素从而给新元素空出空间来需要 O(n)步。如果您忘记了 $O(n^2)$和 O(n log n)算法之间的区别，请查看第 8.5 节：当 n 是 100 万时，需要 3 小时零 1 秒来运行这些程序。

3 由于集合可以包含同一元素的多个副本，因此可以更为精确地将它叫做“多集合”或“包”。由于定义了并操作符所以{2,3}∪{2}={2，2，3}。

数据结构	运行时间		
	1次 insert	1次 extractmin	n次 insert / extractmin
有序队列	O(n)	O(1)	$O(n^2)$
堆	O(log n)	O(log n)	O(n log n)
无序队列	O(1)	O(n)	$O(n^2)$

优先队列的堆实现提供了两个顺序极值之间的折衷。它使用具有堆属性的数组x[1..n]表示n个元素的集合，其中，在C或C++中将x声明成x[maxsize+1]（我们不准备使用x[0]）。将n赋为0，这就将集合初始化为空了。为了插入一个元素，将n加1，然后将新元素放置在x[n]上。这样，siftup就被应用到heap(1,n-1)上。因此，插入代码如下所示：

```
void insert(t)
    if n >= maxsize
        /* report error */
    n++
    x[n] = t
    /* heap(1, n-1) */
    siftup(n)
    /* heap(1, n) */
```

函数extractmin找出了集合中的最小元素，删除它，然后重新组织数组，让它具有堆属性。由于该数组是一个堆，所以最小元素为x[1]。集合中剩下的n-1个元素位于x[2..n]中，它也具有堆属性。通过两步可重新得到heap(1,n)。第一步，首先将x[n]移动到x[1]，并将n减1；现在集合中的元素位于x[1..n]中，并且heap(2,n)为真。第二步调用siftdown。代码非常简单，如下所示：

```
int extractmin()
    if n < 1
        /* report error */
    t = x[1]
    x[1] = x[n--]
    /* heap(2, n) */
    siftdown(n)
    /* heap(1, n) */
    return t
```

当将insert和extractmin应用到包含n个元素的堆时，都需要O(log n)时间。

下面是优先队列完整的C++实现：

```
template<class T>
class priqueue {
private:
    int n, maxsize;
    T *x;
    void swap(int i, int j)
    {   T t = x[i]; x[i] = x[j]; x[j] = t; }
```

```
public:
    priqueue(int m)
    {   maxsize = m;
        x = new T[maxsize+1];
        n = 0;
    }
    void insert(T t)
    {   int i, p;
        x[++n] = t;
        for (i = n; i > 1 && x[p=i/2] > x[i]; i = p)
            swap(p, i);
    }
    T extractmin()
    {   int i, c;
        T t = x[1];
        x[1] = x[n--];
        for (i = 1; (c = 2*i) <= n; i = c) {
            if (c+1 <= n && x[c+1] < x[c])
                c++;
            if (x[i] <= x[c])
                break;
            swap(c, i);
        }
        return t;
    }
};
```

这个小型接口没有错误检查或析构函数，但是却简明地表达了算法的本质内容。当我们的伪码使用冗长代码风格时，这种密集代码却是另一个极端。

14.4 排序算法

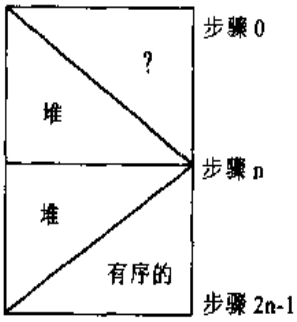
优先队列为排序向量提供了一个简单的算法，首先在优先队列中按序插入每个元素，然后按序删除它们。在C++中使用了类 priqueue 之后编码就相当简单了：

```
template<class T>
void pqsort(T v[], int n)
{   priqueue<T> pq(n);
    int i;
    for (i = 0; i < n; i++)
        pq.insert(v[i]);
    for (i = 0; i < n; i++)
        v[i] = pq.extractmin();
}
```

n 次 insert 和 extractmin 操作在最差情况下的成本是 O(n log n)，优于第 11 章中快速排序的 $O(n^2)$这个最差情况下的时间。不幸的是，堆使用的数组 x[0..n]需要 n+1 个字的额外主存。

现在来看看堆排序，它大大改善了这个方法。它使用了较少的代码，并且由于它不需要辅助数组，因此使用的空间也较少，此外其使用的时间较少。根据该算法的目的，假设修改了 siftup 和 siftdown，让它们操作堆，其中堆的最大元素在顶部，通过交换“<”和“>”符号很容易就能实现这一点。

简单算法使用了两个数组，一个用于优先队列，另一个用于将要排序的元素。堆排序仅使用了一个数组，因而大大节省了空间。单个实现数组 x 表示两个抽象结构：左端是堆，右端是元素序列，一开始的时候顺序是随意的，而最后则是有序的。下图展示了数组 x 的演化过程，数组是水平绘制的，而时间是纵轴。



堆排序算法分两个阶段：前 n 步将数组建立到堆中，后 n 步按照降序方式提取元素并建立最终的有序序列，从右到左。

第一个阶段建立堆。它的不变式如下所示：

该段代码通过将元素在数组中上移，建立了 heap(1,n)。

```
for i = [2, n]
    /* invariant: heap(1, i-1) */
    siftup(i)
    /* heap(1, i) */
```

第二个阶段是使用堆建立有序数列。它的不变式如下所示：

循环体在两个操作中都保存了不变式。由于 x[1]是前 i 个元素中最大的元素，将它和 x[i]交换就将有序序列扩展了一位。这个交换损害了堆属性，我们通过将新的顶部元素下移重新获得堆属性。第二阶段的代码如下所示：

```
for (i = n; i >= 2; i--)
    /* heap(1, i)   && sorted(i+1, n) && x[1..i]   <= x[i+1..n] */
    swap(1, i)
    /* heap(2, i-1) && sorted(i, n)   && x[1..i-1] <= x[i..n]   */
```

```
    siftdown(i-1)
    /* heap(1, i-1) && sorted(i, n)   && x[1..i-1] <= x[i..n]   */
```

有了前面建立的函数后，完整的堆排序算法仅需要五行代码。

```
for i = [2, n]
    siftup(i)
for (i = n; i >= 2; i--)
    swap(1, i)
    siftdown(i-1)
```

由于算法使用了 n-1 次 siftup 和 siftdown 操作，而每个操作的成本最多 O(log n)，所以运行时间为 O(n log n)，即便是在最差情况下。

答案 2 和 3 描述了加速（同时也简化）堆排序算法的几种方法。虽然堆排序保证了最差情况下仍具有 O(n log n)性能，但对于典型的输入数据，最快的堆排序通常也比第 11.2 节中的简单快速排序慢。

14.5 原则

效率。形状属性保证了堆中的所有结点和根之间的距离在 $\log_2 n$ 层之间。由于树是平衡的，所以函数 siftup 和 siftdown 的运行效率很高。堆排序通过在同一个实现数组中交叉两个抽象结构（堆和数列）从而避免了使用额外的空间。

正确性。为循环编写代码首先必须精确说出它的不变式。循环在保持不变式的同时不断地向终止方向前进。*形状*和*顺序*属性表示了一个不同类型的不变式：它们是堆数据结构的不变属性。一个在堆上操作的函数可以假设其在结构上操作时属性为真，并且也必须确保结束的时候这些属性也为真。

抽象。好的工程师能够区分出某个组件完成什么功能（用户看到的抽象概念）以及它是如何完成这个功能的（黑盒内部的实现）。本章将黑盒按两种不同的方式打包：过程抽象和抽象数据类型。

过程抽象。您可以在不知道排序函数实现细节情况下，通过它来排序一个数组：即将排序视为单个操作。函数 siftup 和 siftdown 提供了类似级别的抽象：在建立优先队列和堆排序时，我们并不关心函数是如何工作的，但是我们知道它们做了哪些工作（修补在一端或另一端终止的数组的堆属性）。好的工程允许我们一次定义这些黑盒组件，然后多次使用它们组成两种不同类型的工具。

抽象数据类型。数据类型完成什么功能是由它的方法和它们的规范决定的；而如何实现这些功能则是由它们的实现决定的。我们可以仅仅使用本章的 C++ priqueue 类或最后一章的 C++ IntSet 类的规范来推断它们的正确性。当然，它们的实现肯定对程序的性能有影响。

14.6 问题

1. 实现基于堆的优先队列，并尽可能提高它的运行速度。当n为什么值时它比顺序结构快？

2. 修改siftdown满足下列规范。

```
void siftdown(l, u)
        pre   heap(l+1, u)
        post  heap(l, u)
```

该代码的运行时间是多少？说说如何使用它在O(n)时间内构建一个n个元素组成的堆，以及一个更快的，使用更少代码的堆排序。

3. 让堆排序运行得尽可能快。并将它和第11.3节表格中的排序算法进行比较。

4. 如何使用优先队列的堆实现解决下列问题？当输入是有序的时您的答案有什么变化？

a）构建哈夫曼代码（绝大多数关于信息理论的书和许多关于数据结构的书都详细讨论了这类代码）。

b）计算大型浮点数集合的和。

c）在具有10亿个数值的文件中找出最大的100万个数值。

d）将多个小型有序文件合并到一个大型有序文件中（在实现基于磁盘的合并排序程序时可能出现这类问题，如第1.3节中所介绍的）。

5. 桶打包问题将n个权值分配给最小数量的单位容量桶（每个值都在0和1之间）。对该问题，第一个符合条件的探测方法是按序考虑权值，并将每个权值放置在能满足它的第一个桶中，按递增顺序扫描桶。在他的MIT的论文中，David Johnson观察到类似于堆的结构能够在时间O(n log n)内实现该探测方法。说说如何实现。

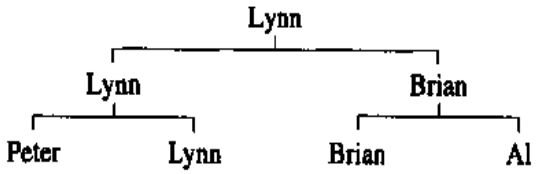
6. 磁盘上的顺序文件的普通实现让每块都指向它的后续，它的后续可能是磁盘上的任意块。该方法要求在固定时间内写入一个块（文件初始化时），读取文件中的第一块，读取文件中的第i块（一旦读取了第i-1块）。因此从头开始读取第i块需要的时间和i成比例。当Ed McCreight在Xerox Palo Alto Research Center设计一个磁盘控制器时，他观察到通过为每个结点增加一个附加指针，就能保留所有其他属性，但是允许在与log i成比例的时间内读取第i块。如何实现这一点？解释一下这个读取第i块的算法在哪些方面和问题4.9中在与log i成比例关系的时间内得到i次幂的数值的代码相一致。

7. 在一些机器上，除以2来找到当前范围的中点是二分查找程序最昂贵的部分。假设正确构建了被查找的数组，证明如何使用乘以2的操作来替代除。给出建立并查找这样一张表的算法。

8. 当队列的平均大小大于 k 时，如何较好地实现表示(0,k)范围内的整数的优先队列？

9. 证明在优先队列的堆实现中，insert 和 extractmin 的同时对数的运行时间在一个最佳常数因子范围内。

10. 运动迷们都很熟悉堆的基本观点。假设在半决赛中，Brian 打败了 Al 并且 Lynn 打败了 Peter，并且在总决赛中，Lynn 胜过了 Brian。将这些结果绘制如下：



这样一棵“锦标赛”树在网球锦标赛和足球、棒球和篮球的季后夺标总决赛中很常见。假设比赛的结果是一致的（在体育运动中这种假设是无效的），那么第二名在冠亚军决赛中夺冠的几率有多大？根据参赛者在锦标赛预赛中的排名给出算法的种子。

11. 在 C++标准模板库中如何实现堆、优先队列和堆排序？

14.7 进阶阅读

第 11.6 节中介绍了 Knuth 和 Sedgewick 编写的优秀的算法教材。在 Knuth 的《*Sorting and Searching*》一书的第 5.2.3 节中介绍了堆和堆排序。在 Sedgewick 的《*Algorithms*》一书的第 9 章中介绍了优先队列和堆排序。

第 15 章　珍珠字符串

我们周围都是字符串。位字符串构成了整数和浮点数，数字字符串构成了电话号码，字符字符串构成了单词，较长的字符字符串构成了 Web 页面，更长的字符串就形成了书。而在遗传数据库和本书很多读者的细胞内部存在很多由 A、C、G 和 T 构成的极长的字符串。

程序对这类字符串执行各类操作。排序、统计、查找并分析它们的具体风格。本章通过查看字符串的一些常见问题介绍这些主题。

15.1　单词

我们的第一个问题就是要生成文档中包含的单词列表（当在几百本书上执行该程序时，你会发现你得到了一本很好的字典的雏形）。但是，什么是单词呢？在这里，单词的定义是由空格包围的一个字符系列，这就意味着 Web 页上包含了很多“单词”，如“<html>”、“<body>”和“ ”。问题 1 就是如何避免这类问题。

我们的第一个 C++程序使用标准模板库的 set 和 string，对答案 1.1 中的程序稍做修改：

```
int main(void)
{   set<string> S;
    set<string>::iterator j;
    string t;
    while (cin >> t)
        S.insert(t);
    for (j = S.begin(); j != S.end(); ++j)
        cout << *j << "\n";
    return 0;
}
```

先通过 while 循环读取输入并将每个单词插入集合 S（通过 STL 规范，忽略重复的单词）。然后通过 for 循环迭代整个集合，并按顺序输出单词。该程序编写得非常漂亮，并且效率很高（下面将会详细介绍这一主题）。

下一步就是计算每个单词在文档中出现的次数。下面是在《King James Bible》中出现频率最高的 21 个单词，按数字顺序递减排列，为了节省空间将这 21 个单词分为三列显示：

```
the  62053      shall 9756      they 6890
and  38546      he    9506      be   6672
of   34375      unto  8929      is   6595
to   13352      I     8699      with 5949
And  12734      his   8352      not  5840
that 12428      a     7940      all  5238
in   12154      for   7139      thou 4629
```

在这本书中，789616 个单词中大概有 8%是单词“the”（在这个列表中，“the”占到了 16%）。根据对单词的定义，“and”和“And”是两个不同的单词，需要分别统计。

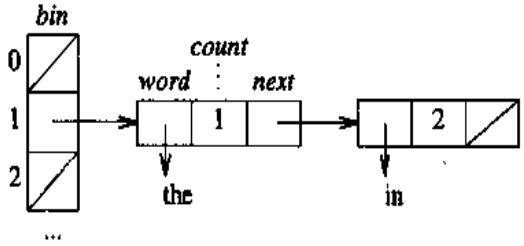
上面的统计是通过下列 C++程序实现的，该程序使用标准模板库的 map 将每个字符串与相应的整数统计值相对应：

```
int main(void)
{   map<string, int> M;
    map<string, int>::iterator j;
    string t;
    while (cin >> t)
        M[t]++;
    for (j = M.begin(); j != M.end(); ++j)
        cout << j->first << " " << j->second << "\n";
    return 0;
}
```

while 语句将每个单词 t 插入到映射 M 并将相关的计数器加 1（这个计数器初始为 0）。for 语句按顺序对单词进行迭代并输出每个单词（首先）和它们的计数值（其次）。

这段 C++代码相当简单，简明并且快速。在我的机器上，它需要 7.6 秒来处理《Bible》这本书。大概需要 2.4 秒来读入这本书，4.9 秒执行插入操作，0.3 秒输出结果。

通过建立自己的散列表，使用包含指向单词的指针、该单词出现频率及指向散列表中下一个结点的指针的结点能够减少处理时间。下面是插入字符串“in”、“the”和“in”之后的散列表，在极特殊的事件中两个字符串都散列为 1：



我们通过下面的 C 结构实现散列表：

```
typedef struct node *nodeptr;
typedef struct node {
    char *word;
    int count;
    nodeptr next;
} node;
```

虽然我们对“单词”的定义非常宽松，但是《Bible》中还是只有29131个不同的单词。我们还是使用原来的方法，使用一个跟它很接近的基本数值来定义散列表的大小，并将乘法器定义为31：

```
#define NHASH 29989
#define MULT 31
nodeptr bin[NHASH];
```

我们的散列函数将一个字符串映射到小于NHASH的一个正整数：

```
unsigned int hash(char *p)
    unsigned int h = 0
    for ( ; *p; p++)
        h = MULT * h + *p
    return h % NHASH
```

使用unsigned（无符号）整数来确保h为正。

main函数将每个结点初始化为NULL，读取单词并将结点的计数器加1，然后迭代整个散列表以输出（无序）单词和计数值：

```
int main(void)
    for i = [0, NHASH)
        bin[i] = NULL
    while scanf("%s", buf) != EOF
        incword(buf)
    for i = [0, NHASH)
        for (p = bin[i]; p != NULL; p = p->next)
            print p->word, p->count
    return 0
```

这项工作是由incword完成的，它根据输入单词增加计数器（并且如果原来没有这个计数器就初始化它）：

```
void incword(char *s)
    h = hash(s)
    for (p = bin[h]; p != NULL; p = p->next)
        if strcmp(s, p->word) == 0
            (p->count)++
            return
    p = malloc(sizeof(hashnode))
    p->count = 1
    p->word = malloc(strlen(s)+1)
    strcpy(p->word, s)
    p->next = bin[h]
    bin[h] = p
```

for循环查看具有相同散列值的每个结点。如果找到了该单词，就将该节点的计数值加1并且函数返回。如果没有找到这个单词，函数就创建一个新结点，分配空间并复制字符串（有经验的C程序员会使用strdup完成该任务），并将该结点插入到链表的前端。

这个C程序大概需要2.4秒读取输入（和C++版本一样），但是只需要0.5秒执行插

入操作（从4.9秒降为0.5秒），并且只需要0.06秒就能输出结果（而原来需要0.3秒）。整个运行时间为3.0秒（原来需要7.6秒），处理时间为0.55秒（原来需要5.2秒）。定制的散列表（C代码的第30行）比从C++标准模板库中进行映射要快一个数量级。

这个小练习表明，主要可以通过两种方法来表示单词集合。平衡查找树将字符串作为不可分割的对象进行处理；在多数STL的集合和映射中都使用了这些结构。由于它们总是保持元素的有序性，所以它们能够高效地执行查找前序和按序输出元素这类操作。另一方面，散列查看字符串内部以完成散列函数，然后将关键字分配到大表中。总的来说速度很快，但是它不能保证平衡树的最差性能，也不能支持其他涉及到顺序的操作。

15.2 词组

单词是文章的基本组成部分，通过查找单词能够解决很多重要的问题。然而，有时需要在长字符串中查找词组（文档、帮助文件或Web页面，甚至是整个Web），如“substring searching”或“implicit data structures”。

如果你以前没有看过某一本书，则如何在这本很厚的书中查找“a phrase of several words”？没有任何选择，只能从头开始扫描整个输入内容，很多算法书中都介绍了各种各样解决“子字符串查找问题”的方法。

假设在执行查找之前，你有一个机会预处理书的内容，则可以通过建立一个散列表（或查找树）来索引文章中的每个不同的单词，并存储每个单词的每次出现。这样生成的“反向索引”使得程序能够快速地查到给定的单词。可以通过交叉获得的单词列表查找词组，但是其实现比较复杂并且执行效率不高（然而，有些Web搜索引擎就是采用这个方法）。

下面将要介绍一个高效的数据结构，并将它应用在一个很小的问题上：给定一个输入文本文件，查找其中最长的重复子字符串。例如，“Ask not what your country can do for you,but what you can do for your country”中最长的重复字符串就是“can do for you”，“your country”是第二个位置。如何编写解决这个问题的程序？

这个问题让我们想起来第2.4节中的变位词程序。如果输入字符串存储在c[0..n-1]中，那么就可以使用类似于下面的伪码比较每对子字符串：

```
maxlen = -1
for i = [0, n)
    for j = (i, n)
        if (thislen = comlen(&c[i], &c[j])) > maxlen
            maxlen = thislen
            maxi = i
            maxj = j
```

当作为comlen函数参数的两个字符串长度相等时，该函数便返回这个长度值，从第一个字符开始：

```
int comlen(char *p, char *q)
    i = 0
    while *p && (*p++ == *q++)
        i++
    return i
```

由于该算法查看所有的子字符串对，所以它的时间和n^2成正比。我们可以通过使用散列表来查找词组中的单词而实现提速，但是下面，我们将采用一种全新的方法。

我们的程序至多可以处理MAXN个字符，这些字符被存储在数组c中：

```
#define MAXN 5000000
char c[MAXN], *a[MAXN];
```

我们将使用一个简单的数据结构“后缀数组”；虽然是从1990年开始才提出这个术语，但至少在1970年就开始使用这个结构了。这个结构是一个指向字符的指针数组a。当我们读取输入时，首先初始化a，这样，每个元素就都指向输入字符串中的相应字符：

```
while (ch = getchar()) != EOF
    a[n] = &c[n]
    c[n++] = ch
c[n] = 0
```

数组c中的最后一个元素是一个空字符，它终止了所有字符串。

元素a[0]指向整个字符串，下一个元素指向以第二个字符开始的数组的后缀，等等。在输入字符串“banana”后，该数组将表示这些后缀：

```
a[0]: banana
a[1]: anana
a[2]: nana
a[3]: ana
a[4]: na
a[5]: a
```

由于数组a中的指针分别指向字符串中的每个后缀，所以将数组a命名为“后缀数组”。

如果在数组c中两次出现长字符串，它就有两个不同的后缀。因此，我们将排序数组以找到相同的后缀（就好像第2.4节中的排序将变位词集中到一起）。“banana”数组排序成：

```
a[0]: a
a[1]: ana
a[2]: anana
a[3]: banana
a[4]: na
a[5]: nana
```

然后就可以扫描该数组比较邻接元素，以找出最长重复的字符串，本例中为“ana”。

我们使用 qsort 函数排序后缀数组：

```
qsort(a, n, sizeof(char *), pstrcmp)
```

比较函数 pstrcmp 将一个间接层添加到库函数 strcmp 中。这个对数组的扫描使用 comlen 函数统计两个邻接单词中相同的字符数：

```
for i = [0, n)
    if comlen(a[i], a[i+1]) > maxlen
        maxlen = comlen(a[i], a[i+1])
        maxi = i
printf("%.*s\n", maxlen, a[maxi])
```

printf 语句使用“*”精确地输出字符串的 maxlen 字符。

运行最终的程序，在 Samuel Butler 对 Homer 的《Iliad》一书的翻译版的 807503 个字符中查找最长的重复字符串。该程序需要 4.8 秒来定位这个字符串：

whose sake so many of the Achaeans have died at Troy, far from their homes? Go about at once among the host, and speak fairly to them , man by man, that they draw not their ships into the sea.

这段文字第一次出现在 Juno 建议 Minerva 将希腊人和特罗伊人隔离开来；之后不久它又出现了，这时候是 Minerva 将这段话一字不差地重复给尤利西兹听。在这个和其他典型的具有 n 个字符的文本文件中，算法运行的时间是 O(n log n)，这主要是由排序导致的。

后缀数组使用文本本身和 n 个附加指针来表示输入文本中的 n 个字符的每个子字符串。问题 6 研究了如何使用后缀数组来解决子字符串查找问题。下面将介绍后缀数组更为复杂的一个应用。

15.3　生成文本

如何才能生成随机文本？一个典型的方法是将那只可怜的猴子放在它那陈旧的打字机上。如果这只猴子单击每个小写字母或空格键的概率是一样的，那么输出可能如下所示：

uzlpcbizdmddk njsdzyyvfgxbgjjgbtsak rqvpgnsbyputvqqdtmgltz ynqotqigexjumq- phuj cfwn ll jiexpyqzgsdllgcoluphl sefsrvqqytjakmav bfusvirsjl wprwqt

这段文字没有任何意义。

如果统计一下单词游戏中的字母数（如 Scrabble™ 或 Boggle™），你会发现，各个不同字母出现的次数是有所不同。例如，A 比 Z 多。猴子通过统计文档中的字符能够生成更有意义的文字。如果在这段文字中 A 出现了 300 次，而 B 出现了 100 次，那么猴子

输入A的可能性是输入B的3倍。这就让我们进一步接近英语：

saade ve mw hc n entt da k eethetocusosselalwo gx fgrsnoh, tvettaf aetnlbilo fc lhd'
okleutsndyeoshtbogo eet ib nheaoopefni ngent

多数事件出现在上下文中。假设我们想要随机生成某年的华式温度数据。0～100范围内的365个随机整数数列不能愚弄一般的观察者。最好将今天的温度变成昨天温度的（随机）函数：如果今天是85度，那么明天就不太可能是15度。

在英语单词中也有同样的道理：如果这个字母是Q，那么下一个字母很有可能就是U。将一个字母做成其前趋的一个随机函数能够生成更有趣的文本。因此，我们能够读取一个样本，然后计算A之后每个字母出现的次数，B之后每个字母出现的次数，...，字母表中各个字母之后各字母出现的次数。在书写随机文本时，我们使用当前字母的一个随机函数产生下一个字母。“order-1”文本就是通过这种方法产生的：

Order-1:t I amy, vin.id wht omanly heay atuss n macon aresethe hired boutwhe t,tl, ad torurest t plur I wit hengamind tarer-plarody thishand.

Order-2: Ther I the heingoind of-pleat, blur it dwere wing waske hat trooss. Yout lar on wassing, an sit.“ ”Yould,“ ”I that vide was nots ther.

Order-3: I has them the saw the secorrow.And wintaikls on my my ent, thinks, fore voyager lanated the been cksed helder was of him a very free bottlemarkable.

Order-4: His heard. “ ”Exactly he very glad trouble, and by Hopkins! That it on on the who difficentralia. He rushed likely? “ ”Blood night that.

我们可以将这个观点扩展到更长的字母序列上。Order-2文本是通过前两个字母的函数来生成每个字母的（通常将一个字母对叫做二字母组）。例如，在英语中二字母组TH后面总是跟着元音A、E、I、O、U或Y，很少跟R和W，基本不会跟有其他字母。Order-3文本的下一个字母是通过前三个字母的函数生成的（三连字母）。在Order-4文本中，绝大多数单词都是英语，而且你会发现它们来自Sherlock Holems小说中(《*The Adventure of Abbey Grange*》)。当受过标准教育的读者阅读完本章内容之后，会想起从古英语到维多利亚式的英语的演变过程。

具有数学背景的读者会认为这个过程是一个马尔科夫链。一个状态表示了每个k链字母，并且从一个链到另一个链的几率不会改变，所以，这是一个“具有固定转换可能性的有限马尔科夫链”。

我们也能够在单词级别生成随机文本。最笨的方法就是按随机顺序输出字典中的每个单词。稍微好一些的方法就是读取一个文档，统计其中每个单词出现的次数，然后选

择下一个合适可能性的单词输出（第 15.1 节中的程序使用适合这类任务的工具）。通过在使用马尔科夫链生成下一个单词的时候考虑一些前趋单词，我们能够获得更加有趣的文本。下面是阅读本书前 14 章的草稿后生成的随机文本：

> Order-1: The table shows how many contexts; it uses two or equal to the sparse matrices were not chosen. In Section 13.1, for a more efficient that"the more time was published by calling recursive structure translates to build search).
> Order-2: The program is guided by verification ideas, and the second errs in the STL implementation (which guarantees good worst-case performance), and is especially rich in speedups due to Gordon Bell. Everything should be to use a macro:for *n* = 10,000,its run time; that point Martin picked up from his desk
> Order-3: A Quicksort would be quite efficient for the main-memory sorts, and it requires only a few distinct values in this particular problem, we can write them all down in the program, and they were making progress towards a solution at a snail's pace.

我们能够很响亮地读出 order-1 中的文本，但是 order-3 中的文本却包含原文输入中很长的词组，中间的过渡非常随机。为了模仿散文，order-2 中的文本有些耸人听闻。

我首先查看了 Shannon 1948 年经典的《*Mathematical Theory of Communication*》一书中英语文本的字母级别和单词级别 order-k 近似值。Shannon 写道：

> "To construct [order-1 letter-level text] for example, one opens a bood at random and selects a letter at random on the page. This letter is recorded. The book is then opened to another page and one reads until this letter is encountered. The succeeding letter is then recorded. Turning to another page this second letter is searched for and the succeeding letter recorded, etc. Asimilar process was used for [order-1 and order-2 letter-level text, and order-0 and order-1 word-level text]. It would be inter-esting if further approximations could be constructed, but the labor involved becomes enormous at the next stage."

程序能够自动完成这项枯燥的工作。我们生成 order-k 马尔科夫链的 C 程序最少将 5 兆字节的文本存放在数组 inputchars 中：

```
int k = 2;
char inputchars[5000000];
char *word[1000000];
int nword = 0;
```

我们可以通过扫描整个输入文本来实现 Shannon 的算法从而生成每个单词（虽然当文本很大时，这个速度非常慢）。我们将数组 word 作为一个指向字母的后缀数组，只是它仅从单词的边界开始（通常的修改）。变量 nword 保存了单词的数目。我们使用下面的代码读取文件：

```
word[0] = inputchars
while scanf("%s", word[nword]) != EOF
    word[nword+1] = word[nword] + strlen(word[nword]) + 1
    nword++
```

将每个单词添加到 inputchars 中（不需要其他存储分配），并通过 scanf 提供的 null 字符终止。

在读取输入之后，我们对 word 数组进行排序，将所有指向同一个 k 单词序列的指针收集起来。该函数进行下列比较：

```
int wordncmp(char *p, char* q)
    n = k
    for ( ; *p == *q; p++, q++)
        if (*p == 0 && --n == 0)
            return 0
    return *p - *q
```

当字符相同时，它就扫描两个字符串。每次遇到 null 字母，它就将计数器 n 减 1，并在查找到 k 个相同的单词后返回相同。当它找到不同的字母时，返回不同。

读取输入之后，就增加 k 个 null 字符（这样，比较函数就不会结束运行），输出文档的前 k 个单词（开始随机输出），并调用排序：

```
for i = [0, k)
    word[nword][i] = 0
for i = [0, k)
    print word[i]
qsort(word, nword, sizeof(word[0]), sortcmp)
```

通常，sortcmp 函数为它的指针添加了一个间接层。

我们的空间效率结构现在包含了大量关于文本中的 k 链字母。如果 k 为 1 并且输入文本为“of the people, by the people, for the people”，word 数组如下所示：

```
word[0]: by the
word[1]: for the
word[2]: of the
word[3]: people
word[4]: people, for
word[5]: people, by
word[6]: the people,
word[7]: the people
word[8]: the people,
```

为了清晰起见，上面仅仅显示了数组 word 中的每个元素所指向的前 k+1 个单词，

虽然后面还有更多单词。如果查找“the”后跟的单词，就在后缀数组中查找它，有三个选择：“people”、“twice”和“people”。

现在，可以通过下面的方法生成没用的文本：

```
phrase = first phrase in input array
loop
    perform a binary search for phrase in word[0..nword-1]
    for all phrases equal in the first k words
        select one at random, pointed to by p
    phrase = word following p
    if k-th word of phrase is length 0
        break
    print k-th word of phrase
```

通过将 phrase 设置成输入的第一个字符初始化循环（记住，这些单词早就在输出文件中）。二分查找使用第 9.3 节中的代码定位 phrase 的第一次出现（找到第一次出现非常关键；第 9.3 节的二分查找就实现这个功能）。下一个循环扫描所有相同的词组，并使用答案 12.10 随机选择其中的一个。因为该词组的第 k 个单词长度为 0，所以当前词组在文档的最后，终止循环。

下列完整的伪码实现了这些观点，并给它将要生成的单词数定一个上界：

```
phrase = inputchars
for (wordsleft = 10000; wordsleft > 0; wordsleft--)
    l = -1
    u = nword
    while l+1 != u
        m = (l + u) / 2
        if wordncmp(word[m], phrase) < 0
            l = m
        else
            u = m
    for (i = 0; wordncmp(phrase, word[u+i]) == 0; i++)
        if rand() % (i+1) == 0
            p = word[u+i]
    phrase = skip(p, 1)
    if strlen(skip(phrase, k-1)) == 0
        break
    print skip(phrase, k-1)
```

Kernighan 和 Pike 的《*Practice of Programming*》的第 3 章（第 5.9 节中介绍的）主要涉及到“设计和实现”这个主题。由于“这是一个经典的程序：输入一些数据，输出一些数据，并根据一些技巧进行处理。”，所以他们围绕着单词级别的马尔科夫文本的生成组织该章内容。他们说明了有关这个问题的一些有趣的历史，并使用 C、Java、C++、Awk 和 Perl 语言实现了该程序。

将本节中完成该任务的程序和它们的 C 程序进行比较，该代码大概是它们的一半：

通过指向 k 个连续的单词的指针来表示词组能够节省空间并且实现起来比较方便。当输入大小接近 1MB 时，两个程序的速度大致相同。由于 Kernighan 和 Pike 使用了较大的数据结构，并大量使用了效率很低的 malloc，所以在我的系统上，本章中的程序使用的内存数量级较小。如果结合答案 14 的加速，并使用散列表替换二分查找和排序，本章中的程序将成为两个加速因素中的一个（并且内存使用大约增加了 50%）。

15.4 原则

字符串问题。编译器如何在符号表中查找变量名？在你输入查找字符串的每个单词时，你的帮助系统如何快速查找整个 CD-ROM？Web 搜索引擎如何搜索一个词组？在解决这些实际的问题的过程中都使用到了本章概要介绍的小型问题中的一些技巧。

字符串的数据结构。我们已经看到了表示字符串的一些最为重要的数据结构。

散列。总体来说，这个结构比较快并且较容易实现。

平衡树。这些结构保证了即便是在输入很少的时候也有较好的性能，并且已经将它们打包在 C++标准模板库的 set 和 map 中。

后缀数组。初始化文本中指向每个字符的指针数组，将它们排序，这样就得到了一个后缀数组。然后，你就能够扫描它，找到最近的字符串或使用二分查找查找单词或词组。

在第 13.8 节中，使用了其他几种结构表示字典中的单词。

库还是定制的组件？C++的 set、map 和 string 使用起来都比较方便，但是它们的通用而强大的接口意味着它们没有专用的散列函数效率高。其他库组件的效率很高：散列使用 strcmp，后缀数组使用 qsort。我查看了 bsearch 和 strcmp 的库实现，并建立了二分查找和马尔科夫程序中的 wordncmp 函数。.

15.5 问题

1. 本章我们使用了单词的简单定义，那就是，单词是以空格为分隔符的。而在很多实际的文档中，如 HTML 或 RTF 中，却还包含格式化命令。如何处理这类命令？你是否还需要进行其他处理？

2. 在具有大量主存的机器上，如何通过 C++ STL 的 set 或 map 解决第 13.8 节中的查找问题？跟 McIlroy 的结构进行比较，它需要多少内存？

3. 如果将答案 9.2 中的专用 malloc 包含到第 15.1 节中的散列程序中，能够提速多少？

4. 当散列表很大，散列函数均匀分布数据时，表中的每一列的元素都很少。如果这

些条件都不冲突，那么查找所需的时间就比较固定。当第 15.1 节中的散列表中没有找到新的字符串时，就将它放置在列表前端。为了模拟散列存在的问题，将 NHASH 设置为 1，并将其和别的列表策略进行试验，如添加到列表最后，将最新找到的元素放置到列表前端。

5. 在查看第 15.1 节中输出的单词频率时，最好将单词按照递减顺序输出。如何修改 C 和 C++程序以实现该任务？如何仅仅输出 M 个最常用的单词（其中 M 是 10 或 1000 这类常量）？

6. 给定一个新的输入字符串，如何在后缀数组中找到所存储文本中的最长匹配？如何通过建立一个 GUI 接口来完成该任务？

7. 对于“典型”的输入来说，我们的程序能够快速找到重复的字符串，但是在某些输入条件下，其速度很慢。计算这类输入所需的时间。在实际应用中会不会出现的这类输入？

8. 如何修改查找重复字符串的程序，以找出出现次数超过 M 次的最长的字符串？

9. 在给定的两个输入文本中，找出在两段文本中都出现的最长的字符串。

10. 说说如何通过仅指向从单词边界开始的后缀来减少重复程序中的指针的数目。这对程序产生的输出有何影响？

11. 编写一个程序，生成字母级别的马尔科夫文本。

12. 如何使用第 15.1 节中的工具和技巧生成（order-0 或非马尔科夫）随机文本？

13. 本书的 Web 站点上提供了单词级别的马尔科夫文本生成程序。在你的一些文档上测试该程序。

14. 如何使用散列提速马尔科夫程序？

15. 第 15.3 节中对 Shannon 的引用描述了他用来构建马尔科夫文本的算法。编写程序实现该算法。它给出了马尔科夫频率的近似值，但不是精确的形式。解释为什么不是精确的形式。编写程序从头开始扫描整个字符串以生成每个单词（因此使用真实的频率）。

16. 如何使用本章介绍的技巧组合成词典的单词列表（这是第 13.8 节中 Doug McIlroy 面临的问题）？如何不使用词典就建立一个拼写检查器？如何不使用语法规则就建立一个语法检查器？

17. 调查一下在速度识别和数据比较这类应用程序中，k 链字母分析技巧的使用情况。

15.6　进阶阅读

第 8.8 节中引用的很多书都包含表示和处理字符串的效率算法和数据结构的材料。

第一版本的尾声

当时看来，对作者的一次采访是本书第一版本的最好总结。它仍然介绍了这本书，所以，这里再次使用了它。

Q：谢谢你同意我进行这次采访。

A：不客气——现在我的这段时间归你安排。

Q：我看到在《*Communications of the ACM*》中早就有这些章节的内容了，为什么你还要将它们收集到这本书中？

A：主要出于几个小原因：我修改了其中几个小错误，并做出了几百个小的改进，添加了一些新的章节。书中有50%的问题、答案和图片都是新的。而且，我认为将很多章节收集在一本书中要比出现在几十本杂志中更方便。然而，最主要的原因是，将它们收集在一本书中之后，可以很容易地读书中的理论，整体大于局部之和。

Q：本书中都有哪些理论？

A：最重要的就是认真考虑编程问题既实用又有趣。这项工作不仅仅涉及到从正式的需求文档到系统的程序开发。如果这本书导致哪怕仅仅是一个程序员幡然醒悟、重新爱上他/她的工作，那么这本书就达到了它的目的了。

Q：这个回答比较空洞。本书是不是通过一些技术线索连接起来的？

A：第 2 部分主要讨论了性能，这是所有章节的主题。在好多章中都用程序进行了验证。附录 1 总结了本书的算法。

Q：很多章节都强调了设计过程。你能不能总结一下该主题？

A：我很乐意回答这个问题。在这次采访之前我正在准备出一个列表。该列表的内容是对程序员的 10 条建议。

处理正确的问题；

展示解决方法的设计空间；

查看数据；

使用封底；

利用对称性；

使用组件进行设计；

建立原型；

必要时进行权衡；

尽量简单；

让程序尽量优美。

最初是在编程中讨论这些问题，但最终它们却应用在工程上。

Q：这让我想起了一个问题，这个问题一直困扰着我：在本书中，很容易就能简化一些小程序，那么在实际的软件中，这项技巧同样有用么？

A：我有三个答案：是的、不是和可能。“是的”，它们同样起作用；例如，第3.4节[在第一版本中]中介绍了一个大型软件项目，这个项目最终简化成仅需要80个员工每年。一个同样迂腐的答案就是“不是”：如果你简化得适当，你就能避免建立庞大的系统，并且不需要按比例使用这些技巧。虽然两种观点都有好处，但实际情况往往存在于两者之间，这就是“可能”的来源。有些软件必然很大，本书的一些理论比较适合用于这些系统上。Unix系统是一个很好的例子，它的强大就来自于简单而优美的各个组成部分。

Q：你在书中讨论了另一个Bell Lab系统。这些章节是不是有些偏？

A：可能是有一点。我主要使用了我实际应用中的一些材料，这就使得这本书有些偏向于我的实际情况。积极些说，这些章节的很多材料都是我的同事提供的，他们应该受到表扬（或批评）。我在Bell Lab中从研究人员和开发人员那里学到了很多东西。那里具有很好的合作氛围，研究和开发之间的交互非常好。所以，很多你觉得比较偏的东西，实际上是我对我的使用者的一种真诚表现。

Q：让我们回到现实中来。本书少了哪些内容？

A：我本来希望书中能够有一个大型系统，它由很多程序构成，但是我不能在一章中通过十页左右就描述出任何有意义的系统。更实际些说，我希望还有以后的章节讨论这些主题“程序员的计算机科学”（类似于第4章的程序认证和第8章的算法设计），以及“计算工程技术”（类似于第7章的封底计算）。

Q：如果你深入研究“科学”和“工程”，那么为什么这些章节不是偏重理论和表，而是故事情节？

A：请注意——自己再回过头去看的人应该可以不注重写作风格。

第二版的尾声

一些传统由于自身的内涵而流传。另外一些则是因为其他原因。

Q：谢谢你能够再次接受我的采访，已经过了很多年了。

A：14 年。

Q：让我们继续上次的采访。为什么要出这本书的新版本？

A：我喜欢这本书，非常喜欢。这本书写起来有很大的乐趣，并且这么多年读者一直都非常支持我。书中的原理经受了时间的考验，但是第一版本中的很多例子都过时了。现在的读者很难将“巨型”计算机和半兆的主存联想起来。

Q：那么，你在这个新版本中作了哪些修改呢？

A：很多，我在序言中都说明了。在采访之前，你没有看过么？

Q：噢，对不起。我在网站上看了你关于如何获得本书代码的讨论。

A：编写那些代码是我写这个版本中觉得最有乐趣的事情。在第一版中我使用了绝大多数程序，但是我仅是想让读者看看真实的代码。在这个版本中，我大概写了 2500 行的 C 和 C++代码。

Q：你宣称这些代码是公开的？我阅读了一部分，很糟糕的风格！微观变量名、奇怪的函数定义、应该是参数的全局变量，等等。你难道不觉得让真正的软件工程师看到这些代码比较难堪么？

A：实际上，在大型软件项目中我使用的风格能够大大改善。然而，本书不是一个大型软件项目。这甚至不是一本大型的书。答案 5.1 介绍了简要的代码风格和我选择它的原因。如果我想写一本几千页的书，我就会采用长一些的代码风格。

Q：说到长代码，你的 sort.cpp 程序对应了 C 标准库 qsort、C++标准模板库 sort 和几个手写的快速排序。你不能下定决心么？程序员应该使用库函数还是自己重新编写代码？

A：Tom Duff很好地回答了这个问题："尽可能盗用别人的代码。"库非常伟大，任何时候都尽可能使用它们。使用系统库，然后在其他库中查找适当的函数。在任何工程活动中，不是所有的人工制品适用于所有的客户。我希望本书提供的伪码（以及网站上的真实的代码）对于那些必须自己编写程序的人来说比较有用。我觉得本书的脚手架和经验方法将帮助这些程序员评估大量的算法并从中选出一个最佳方法用于他们的程序中。

Q：除了一些共有代码和更新的故事外，这个版本中真正有新意的地方是什么？

A：我尽量在高速缓存和指令层并发行中面对代码优化。在较大的层面上，新的三章内容反映了本版中的三个主要的改变：第5章描述了真实的代码和脚手架。第13章给出了数据结构的细节，第15章派生出了高级算法。本书的很多观点在印刷之前早就存在了，但是附录3中的成本模型及第15.3节中的马尔科夫文本算法第一次出现在这儿。并将新的马尔科夫文本算法和Kernighan和Pike经典算法进行了比较。

Q：你接触了很多贝尔实验室的人。我们上一次讨论的时候，你对那个地方非常有感情，但是你就在那儿呆了几年的时间。在最近14年，实验室变化很大。你对这个地方有什么感想，对那些改变有什么想法？

A：我在编写本书的第一章的时候，贝尔实验室是贝尔系统的一部分。第一版本出版后，我们是AT&T的一部分。现在我们是朗讯科技的一部分。公司、通信产业和计算领域在这段时间内变化都很大。贝尔实验室也进行了这些革新，并且总是先驱。我进入这个实验室是因为我喜欢平衡理论和应用，我想构建产品并出书。我在实验室的这段时间内，钟摆来回摆动，但是我的管理总是涉及到大量的活动。

本书第一版本的一位评论人员说到"Bentley每天的工作环境就是一个编程涅槃。他是新西兰莫累山贝尔实验室技术部的一名成员，直接访问了边缘软件和软件技术，并和世界上很多最优秀的软件开发者位于同一个自助餐馆。" 贝尔实验室仍然是这样一个地方。

Q：每天都生活在天堂中么？

A：每天都好像生活在天堂中，但是不在天堂中的日子也很美好。

附录1 算法分类

本书从另一个完全不同的角度介绍了大学算法课程中的很多题材——本书更加强调应用和编码而不是材料分析。本附录将材料和更加典型的框架结合起来。

排序

问题定义。输出序列是输入序列的有序数列。当输入是文件时，输出通常是一个不同的文件。当输入为数组时，输出通常是同一个数组。

应用。该列表仅仅表明了排序应用的多样性。

- 输出需求。一些用户需要有序输出；参见第 1.1 节并考虑您的电话簿和每月查看的账单。二分查找这类函数要求输入是有序的。
- 收集相同项。程序员通过排序收集数列中的相同项：第 2.4 节和第 2.8 节中的变位词程序收集同一变位词类中的单词。第 15.2 节和第 15.3 节中的后缀数组收集相同的文本词组。同时请参考问题 2.6、8.10 和 15.8。
- 其他应用。第 2.4 节和第 2.8 节中的变位词程序将排序作为单词中字母的规范次序，因此将它作为变位词类的一个标记。问题 2.7 通过排序重新组织磁带上的数据。

通用函数。下列算法对 n 元素数列进行排序。

- 插入排序。第 11.1 节中的程序在最差情况下和随机输入时的运行时间为 $O(n^2)$。并且在该节的表中介绍了多个变量的运行时间。第 11.3 节使用插入排序在 O(n) 时间内排序一个本来就几乎是有序的数组。这是本书介绍的惟一一个稳定排序：具有相同值的输出元素和输入具有相同的相对顺序。
- 快速排序。第 11.2 节中的简单快速排序在 n 个不同元素的数组上运行时所需要的运行时间为 O(n log n)。它是递归的，平均使用对数栈空间。在最差的情况下，它所需要的时间为 $O(n^2)$，并且需要的栈空间为 O(n)。在相同元素的数组上，其运行时间为 $O(n^2)$。第 11.3 节中的改进后的版本在任何一个数组上的运行时间都是 O(n log n)。第 11.3 节中的表提供了有关快速排序的多个实现的运行时间的经验数据。通常，C 标准库的 qsort 就是使用该算法实现的。该算法还应用于第 2.8

节、第 15.2 节、第 15.6 节，以及答案 1.1 中。通常，C++标准库的 sort 也是使用该算法实现的，在第 11.3 节中对它进行了计时。

- 堆排序。第 14.4 节中的堆排序在任何 n 元素数组上的运行时间都是 O(n log n)，该算法没有使用递归，仅仅使用了固定的额外空间。答案 14.1 和答案 14.2 介绍了更快的堆排序。
- 其他排序算法。在第 1.3 节中概述了归并排序算法，该算法在排序文件的时候具有非常高的效率。问题 14.4.d 中概述了归并算法。答案 11.6 中给出了选择排序和 shell 排序的伪码。

答案 1.3 中给出了多个排序算法的运行时间。

专用函数。通过使用这些函数，能够产生特定输入上的简短而高效的程序。

- 基数排序。问题 11.5 中 McIlroy 的位字符串排序能够生成较大字母表上的有序字符串（例如，字节）。
- 位图排序。第 1.4 节中的位图排序利用了要排序的整数通常在一个很小的范围内并且没有重复的元素也没有多余的数据这一事实。答案 1.2、1.3、1.5 和 1.6 给出了实现细节和扩展。
- 其他排序。第 1.3 节中的多通道排序多次读取输入，通过时间获取空间。第 12 章和第 13 章生成了随机整数的有序集合。

查找

问题定义。查找函数判断它的输入是不是给定集合的成员，并能检索出相关信息。

应用。问题 2.6 中查找 Lesk 的电话号码簿，将（编码）姓名转换成电话号码。第 10.8 节中的 Thompson 的程序的最后阶段通过查找棋盘来执行最优的移动。第 13.8 节中 McIlroy 的拼写检查器查找目录判断某个单词的拼写是否正确。结合该函数介绍了其他一些应用程序。

通用函数。下面的算法用于查找任意 n 元素集合。

- 顺序查找。第 9.2 节给出了顺序查找数组的一个简单而优化过的版本。第 13.2 节给出了数组和链表中的顺序查找。在连字符单词（问题 3.5）、平滑的地理数据（第 9.2 节）、表示一个稀疏矩阵（第 10.2 节）、生成随机集合（第 13.2 节）、排序压缩目录（第 13.8 节）、桶压缩（问题 14.5），以及查找所有相同的文本词组（第 15.3 节）中使用了该算法。第 3 章和问题 3.1 的简介部分介绍了顺序查找的两个比较愚蠢的实现。
- 二分查找。第 2.2 节介绍了一个大概通过 $\log_2 n$ 次比较来查找有序数组的算法，

并且在本书的第 4.2 节中开发了该算法相应的代码。第 9.3 节扩展了该代码，找出所有相同元素的第一次出现并优化了它的性能。在一个预定系统（第 2.2 节）、错误输入行（第 2.2 节）、输入单词的变位词（问题 2.1）、电话号码（问题 2.6）、线段中某个点的位置（问题 4.7）、稀疏数组中某个项的下标（答案 10.2）、随机整数（问题 13.3）和词组（答案 15.2 和答案 15.3）这些应用中都包含了对记录的查找。问题 2.9 和问题 9.9 讨论了二分查找和顺序查找之间的利弊权衡。

- 散列化。问题 1.10 讨论了电话号码的散列计数，问题 9.10 讨论了整数集合的散列技术，第 13.4 节讨论了使用桶散列整数集合的技术，第 13.8 节则讨论了字典中的单词散列技术。第 15.1 节使用散列技术统计了文档中的单词。
- 二分查找树。第 13.3 节使用（非平衡）二分查找树表示了一个随机的整数集合。平衡树主要用来实现 C++标准模板库中的 set 模板，我们在第 13.1 节、第 15.1 节，以及答案 1.1 中都使用了平衡树。

专用函数。这些函数生成了特定输入下的简短而高效的程序。

- 关键字索引。可以将一些关键字用作数组值的索引。第 13.4 节中的桶和位向量使用整数关键字作为索引。关键字索引包括电话号码（第 1.4 节）、字符（答案 9.6）、三角函数的参数（问题 9.11）、稀疏数组的下标（第 10.2 节）、程序计数值（问题 10.7）、棋盘（第 10.8 节）、随机整数（第 13.4 节）、字符串的散列值（第 13.8 节）和优先队列中的整数值（问题 14.8）。问题 10.5 通过关键字索引和数值函数减少了空间。
- 其他方法。第 8.1 节描述了如何通过将常用元素保存在高速缓存中来减少查找所需的时间。第 10.1 节描述了在理解了上下文的情况下如何简化对税务表的查找。

其他集合算法

这些问题用于处理一个可能包含重复值的、n 个元素的集合。

优先队列。优先队列中包含一组元素，通过插入任意元素和删除最小元素来操作该队列。第 14.3 节介绍了完成该任务的两个顺序结构，并用堆给出了一个高效实现优先队列的 C++类。问题 14.4、14.5 和 14.8 介绍了它的应用。

选择。问题 2.8 介绍了一个问题，在这个问题中，我们必须选择出集合里的前 k 个最小的元素。答案 11.9 介绍了完成该任务的一个高效算法，也可以使用问题 2.8、11.1 和 14.4.c 中提到的算法。

与字符串相关的算法

第2.4节和第2.8节计算字典中的变位词集合。答案9.6介绍了将字符分类的几种方法。第15.1节列出了文件中不同的单词并统计了文件中各个单词的出现次数，首先使用C++标准模板库组件，然后使用定制的散列表。第15.2节用后缀数组查找了文本文件中最长的重复子串，第15.3节使用可变后缀数组根据马尔科夫模型生成了随机文本。

向量和矩阵算法

第2.3节和问题2.3、2.4讨论了交换向量中的子序列的算法，答案2.3提供了实现该算法的代码。问题2.5介绍了交换向量中的非邻接子序列的算法。问题2.7使用排序转置磁带上的一个矩阵。问题4.9、9.4和9.8介绍了计算向量中的最大值的程序。第10.3节和第14.4节介绍了共享空间的向量和矩阵算法。第13.1节、第10.2节和第13.8节讨论了稀疏向量和矩阵。问题1.9介绍了初始化第11.3节中使用的稀疏向量的方法。第8章介绍了计算向量中的最大和子序列的五种算法，第8章中的一些问题涉及到了向量和矩阵。

随机对象

全书都用到了伪随机整数的生成函数。在答案12.1中实现了这些函数。第12.3节介绍了“移动”数组中元素的算法。第12.1节到第12.3节介绍了选择集合中随机子集的算法（同时请参见问题12.7和12.9）。问题1.4给出了该算法的应用。答案12.10给出了随机选择一个未知编码的对象的算法。

数值算法

答案2.3给出了计算两个整数的最大公因子的欧几里德算法。问题3.7概述了使用常量系数评估线性递归的算法。问题4.9给出了计算数值正整数次幂的高效算法的代码。问题9.11通过表查找计算三角函数。答案9.12介绍了Horner的计算多项式的方法。问题11.1和14.4b介绍了如何计算大量浮点数之和。

附录2 估算测试

第7章的封底计算都是从基本的数量开始的。在问题的规格说明（如需求文档）中通常能够看到这些数值，但在其他时候它们可能是估算值。

设计这个小测试是为了帮助您评估您数值估算的熟练程度。请您根据个人观点，填上每个问题的上下界，给您90%的机会将真值包含在其中。尽量不要将范围设置得太大或太小。我想，这个测试大概需要5~10分钟。请尽量认真对待（下一个读者可以使用本页的复印件）。

[_____, _____]2000年1月1日，美国的人口数量，单位百万。

[_____, _____]拿破仑出生日。

[_____, _____]密西西比河的长度，单位英里。

[_____, _____]波音747大型客机最大载重，单位磅。

[_____, _____]从地球到月球的声音信号传播所需秒数。

[_____, _____]伦敦的纬度。

[_____, _____]航天飞机围绕地球旋转一周所需秒数。

[_____, _____]金门桥上两个塔台之间的长度，单位英尺。

[_____, _____]独立宣言的签名人数。

[_____, _____]成年人身上骨头的数量。

完成该测试之后，请翻到下一页查看答案和解释。

请在翻到下一页之前先回答上面的问题。

如果您还没有独立完成上面的所有题目，请回过去继续完成。下面是答案，来自于年历或类似的资源。

2000 年 1 月 1 日美国的人口总数为 272.5 百万。

拿破仑出生于 1769 年。

密西西比河的长度是 3710 英里。

波音 747-400 客运飞机最大的载重是 875000 磅。

从地球到月球的声音信号传播需要 1.29 秒。

伦敦的纬度是 51.1 度。

航天飞机围绕地球旋转一周需要 91 秒。

金门桥上两个塔台之间的长度是 4200 英尺。

独立宣言的签名人数是 56。

成年人身上有 206 根骨头。

请计算一下有多少范围中包含了正确的答案。由于您使用了 90%的信任区间，所以您的答案中必须有 9～10 个是正确的。

如果您的所有答案都是正确的，那么您是一个非常优秀的估算者。同时，您的范围也可能非常大，那么您可能什么都能猜对。

如果在您的答案中，正确的个数小于或等于 6 个，那么您就跟我第一次做类似的估算测试一样，情况比较糟糕。希望您能够通过一些练习提高估算水平。

如果您有 7～8 个答案是正确的，那么您是一个较好的估算者。以后请记住，将您 90%的范围再扩展一些。

如果您正好有 9 个答案是正确的，那么您是一个优秀的估算者。或者，对于前 9 个问题而言您的范围是无限大，而最后一个问题的范围则为零。如果这样，那么您应该感到羞愧。

附录 3　时间和空间成本模型

第 7.2 节介绍了两个用来估算各类原语操作的时间和空间消耗的小程序。本附录显示了如何将它们扩展成能够生成一页时间和空间估算的有用程序。在网站（www.Programmingpearls.com）上提供了这两个程序的完整源代码。

程序 spacemod.cpp 生成了 C++中各结构所消耗的空间的模型。程序的第 1 部分使用了如下语句序列

```
cout << "sizeof(char)=" << sizeof(char);
cout << " sizeof(short)=" << sizeof(short);
```

来精确衡量原语对象：

```
sizeof(char)=1  sizeof(short)=2  sizeof(int)=4
sizeof(float)=4  sizeof(struct *)=4  sizeof(long)=4
sizeof(double)=8
```

该程序使用这些例子中的简单命名转换定义了很多结构：

```
struct structc { char c; };
struct structic { int i; char c; };
struct structip { int i; structip *p; };
struct structdc { double d; char c; };
struct structc12 { char c[12]; };
```

该程序使用宏输出结构的 sizeof，然后估算如下所示的 new 分配的字节数：

```
structc    1   48 48 48 48 48 48 48 48 48 48
structic   8   48 48 48 48 48 48 48 48 48 48
structip   8   48 48 48 48 48 48 48 48 48 48
structdc   16  64 64 64 64 64 64 64 64 64 64
structcd   16  64 64 64 64 64 64 64 64 64 64
structcdc  24  -3744 4096 64 64 64 64 64 64 64 64
structiii  12  48 48 48 48 48 48 48 48 48 48
```

第一个数值由 sizeof 给出，后面十个数值报告了 new 返回的后续指针之间的区别。这个输出结果非常典型：绝大多数数值都保持了一致，但是分配器时不时会间隔一下。

这个宏输出了一行内容：

```
#define MEASURE(T, text) {                \
    cout << text << "\t";                 \
    cout << sizeof(T) << "\t";            \
    int lastp = 0;                        \
    for (int i = 0; i < 11; i++) {        \
```

```
        T *p = new T;                          \
        int thisp = (int) p;                   \
        if (lastp != 0)                        \
            cout << " " << thisp - lastp; \
        lastp = thisp;                         \
    }                                          \
    cout << "\n";                              \
}
```

调用这个宏时，在结构名之后的引号内跟着相同的名字，如下所示：

```
MEASURE(structc, "structc");
```

（我的第一份草稿使用了带有结构类型参数的 C++模板，但是 C++实现的人为因素导致了它的衡量偏差很大。）

下表总结了我机器上的程序的输出结果：

结构	sizeof	new 的开销空间
int	4	48
structc	1	48
structic	8	48
structip	8	48
structdc	16	64
structcd	16	64
structcdc	24	64
structiii	12	48
structiic	12	48
structc 12	12	48
structc 13	13	64
structc 28	28	64
structc 29	29	80

左边一列数字帮助我们估算结构的 sizeof。首先总结类型的 sizeof；它解释了 structip 的 8 字节。我们也必须考虑对齐问题；虽然它的组成部分总共需要 10 个字节（两个 char 和一个 double），但是 structcdc 消耗了 24 个字节。

右边一列给出了 new 操作符开销的空间。显然，所有 sizeof 为 12 或小于 12 的结构都将消耗 48 个字节。

13～28 个字节的结构消耗 64 个字节。总的来说，分配块的大小是 16 的倍数，大概有 36～47 个字节的开销。这个开销相当昂贵，我使用的其他系统仅仅使用 8 个字节的开

销来表示一条 8 个字节的记录。

第 7.2 节也介绍了估算特定 C 操作成本的小程序。我们可以将它推广到一个生成一组 C 操作的一页长的成本模型的 timemod.c 程序中（在 1991 年，Brian Kernighan、Chris Van Wyk 和我编写了该程序的蓝本）该程序的 main 函数包含了一系列的 T（标题）行，后面跟着 M 行来衡量操作的成本：

```
T("Integer Arithmetic");
M({});
M(k++);
M(k = i + j);
M(k = i - j);
    ...
```

这些行（以及类似的行）会产生如下输出：

```
Integer Arithmetic (n=5000)
 {}          250   261   250   250   251   10
 k++         471   460   471   461   460   19
 k = i + j   491   491   500   491   491   20
 k = i - j   440   441   441   440   441   18
 k = i * j   491   490   491   491   490   20
 k = i / j  2414  2433  2424  2423  2414   97
 k = i % j  2423  2414  2423  2414  2423   97
 k = i & j   491   491   480   491   491   20
 k = i | j   440   441   441   440   441   18
```

第一列给出了循环内部执行的操作：

```
for i = [1, n]
    for j = [1, n]
        op
```

后面的五列是该循环五次执行的时钟的原始时间(本系统为毫秒)（这些时间是一致的，不一致的数值能够帮助找出可疑的运行)。最后一列给出了每个操作的平均成本，单位为纳秒。表中的第一行说明需要 10 个纳秒来执行包含空操作的循环。下一行说明增加变量 k 大概额外消耗了 9 纳秒。除了除和余数操作之外，所有的算术和逻辑操作开销大致相同，除和余数操作具有更高的数量级。

我使用该方法对我的机器进行了大致的估算，很难解释清楚。我查看了整个试验，关闭了所有的优化。当我启用这些选项时，优化器删除了时间循环，所有的时间都为零。

这项工作是通过 M 宏完成的，如下面的伪码所示：

```
#define M(op)
    print op as a string
    timesum = 0
    for trial = [0, trials)
        start = clock()
        for i = [1, n]
```

```
            op
        t = clock()-start
        print t
        timesum += t
    print 1e9*timesum / (n*n * trials * CLOCKS_PER_SEC)
```

本书的网站上提供了该成本模型的完整代码。

现在，我们看看该程序在我机器上的输出结果。由于时钟点击都是一致的，我们将其忽略，仅仅报告用纳秒表示的平均时间。

```
Floating Point Arithmetic (n=5000)
 fj=j;                  18
 fj=j; fk = fi + fj     26
 fj=j; fk = fi - fj     27
 fj=j; fk = fi * fj     24
 fj=j; fk = fi / fj     78
Array Operations (n=5000)
 k = i + j              17
 k = x[i] + j           18
 k = i + x[j]           24
 k = x[i] + x[j]        27
```

浮点操作开始时将整数 j 赋给浮点数 fj（大概需要 8 纳秒）；外循环将 i 赋给浮点数 fi。浮点操作本身的成本大概和它们的整数部分相同，数组操作的开销同样也不是很大。

下面的文本能够让我们从总体上了解控制流和一些特定的排序操作：

```
Comparisons (n=5000)
 if (i < j) k++           20
 if (x[i] < x[j]) k++     25
Array Comparisons and Swaps (n=5000)
 k = (x[i]<x[k]) ? -1:1   34
 k = intcmp(x+i, x+j)     52
 swapmac(i, j)            41
 swapfunc(i, j)           65
```

比较和交换的函数版本每个比它们的内联部分要多使用 20 纳秒。第 9.2 节比较了使用函数、宏和内联代码计算两个值中的最大值的成本：

```
Max Function, Macro and Inline (n=5000)
 k = (i > j) ? i : j   26
 k = maxmac(i, j)      26
 k = maxfunc(i, j)     54
```

相对来说，rand 函数的成本较小（虽然每次调用 bigrand 函数都将调用两次 rand），平方根的数量级大于基本的算术操作（虽然只是除操作的两倍），简单的三角函数操作是它的两倍，而高级三角函数操作需要微秒时间。

```
Math Functions (n=1000)
 k = rand()          40
 fk = j+fi           20
```

```
fk = sqrt(j+fi)     188
fk = sin(j+fi)      344
fk = sinh(j+fi)    2229
fk = asin(j+fi)     973
fk = cos(j+fi)      353
fk = tan(j+fi)      465
```

由于这些操作的代价都很高，我们缩小了 n 的值。但是内存分配却更加昂贵，应该使用一个更加小的 n：

```
Memory Allocation (n=500)
 free(malloc(16))        2484
 free(malloc(100))       3044
 free(malloc(2000))      4959
```

附录 4　代码优化规则

我在 1982 年出的书《*Writing Efficient Programs*》中建立了代码优化的 27 条规则。这本书现在已经不出版了，所以我在这里重复一下那些规则（仅做了一些微小的改动），同时给出了它们在本书中的应用例子。

用空间换取时间规则

扩展数据结构。通常，通过给结构增加其他信息或改变结构内部的信息让它访问得更快能够减少对数据的常用操作所需的时间。

在第 9.2 节中，Wright 想要在地球表面的一组点中找出最近的一个邻接点（按角度来说），这些点都是使用经度和纬度表示的，这项工作涉及到昂贵的三角函数操作。Appel 扩展了她的数据结构，使用了 x、y 和 z 坐标，这样就能够以更少的计算时间来使用欧几里德距离。

存储预先计算好的结果。计算函数一次，然后存储计算结果能够减少昂贵函数的重新计算所需的成本。以后对该函数的请求就只需要通过表查找来完成，而不需要重新计算该函数。

第 8.2 节和答案 8.11 中的累加数组使用两张表的查找和减法替代了一系列的加法。

答案 9.7 通过一次查找一个字节或单词加速了程序对位的计算。

答案 10.6 使用表查找替代了位移和逻辑操作。

高速缓存。必须降低经常访问的数据的访问成本。

第 9.1 节介绍了 Van Wyk 是如何将最常用的结点大小缓存起来以避免对系统存储分配器的昂贵调用。答案 9.2 给出了一个结点缓存的细节。

第 13 章为列表、段和二分查找树缓存了结点。

如果在基本数据中没有表示位置，那么高速缓存可能会逆火并增加程序的运行时间。

懒惰计算法。除非需要，否则该策略永远都不会计算某个元素，这样可以避免计算不必要的元素。

用时间换取空间规则

压缩。密集存储表示能够通过增加存储和检索数据所需的时间来降低存储成本。

第 10.2 节中的稀疏数组表示通过增加访问该结构所需时间大大减少了存储成本。

第 13.8 节中的 McIlroy 的拼写检查器词典将 75000 个英语单词压缩到 52KB。

第 10.3 节中的 Kernighan 的数组和第 14.4 节中的堆排序都使用了交叉技术，通过排序同一内存空间中从来都不会同时调用的数据项减少了数据空间。

虽然压缩有时是通过牺牲时间来获取空间，但是通常能够快速处理小的表示。

解释程序。通常，使用解释程序能够减少表示程序所需的空间，解释程序压缩表示相同的操作序列。

第 3.2 节使用了“形式字母编程”解释程序，第 10.4 节使用指向简单图形程序的解释程序。

循环规则

将代码移出循环。最好不要在循环的每次迭代中都执行特定的操作，而是将它放在循环外部，仅仅执行一次。

第 11.1 节将对变量 t 的赋值移到了 isort2 的主循环的外部。

合并测试条件。高效的内部循环应该尽量少包含测试条件，最好只有一个。因此，程序员应该尽量使用其他退出条件模拟循环的一些退出条件。

哨兵是该规则最常见的应用：在数据结构的边界上放一个哨兵来减少对是否已经查完整个结构的测试。第 9.2 节通过哨兵线性查找一个数组。第 13 章使用哨兵产生数组、链表、桶和二分查找树的清晰代码（同时也是高效的）。答案 14.1 在堆的结束部分放置了一个哨兵。

循环的解开。解开一个循环能够消除修改循环下标的成本，同时也能避免管道拖延、减少分支，并增加指令层的并发。

解开第 9.2 节的顺序查找大概能将它的运行时间减少 50%，第 9.3 节中解开了一个二分查找将它的运行时间大致减少到 35%～65%。

传输驱动的循环解开。如果在普通的赋值中使用了一个成本很高的内部循环，那么通常通过重复这些代码改变对变量的使用能够消除这些赋值。特别是，删除赋值 i=j，后面的代码必须将 j 视为 i。

消除无条件分支。在快速的循环中不应该包含无条件分支。通过“旋转”循环，在

底部加上一个条件分支，能够消除循环结束处的无条件分支。

通常由优化编译器完成该操作。

循环合并。如果两个邻近的循环操作作用在同一个元素集上，那么最好合并这两个操作部分，仅仅使用一个循环控制操作。

逻辑规则

利用代数恒等式。如果逻辑表达式的计算非常昂贵，就使用比较廉价的代数等式表达式来替代它。

简化的单调函数。测试几个变量的单调非递减函数是不是超过了特定的阈值，一旦达到了这个阈值就不需要计算任何变量。

该规则的一个更加复杂的应用就是一旦达到了循环的目的就退出循环。第 10 章、第 13 章和第 15 章中的查找循环都是一旦找到了所需的元素就终止。

重新排序测试。在组织逻辑测试的时候，应该将廉价的经常成功的测试放在昂贵的很少成功的测试前面。

答案 9.6 概述了可能需要重新排序的一系列测试。

预计算逻辑函数。可以使用表示域的表的查找替代一个小的有限域中的逻辑函数。

答案 9.6 介绍了如何使用表查找替代标准的 C 库字符分类函数。

消除布尔变量。我们可以通过用 if-else 语句替代对布尔变量的 v 赋值来消除布尔变量，在 if-else 语句中一个分支表示 v 为真的情况，其他的表示 v 为假的情况。

过程规则

压缩函数层次。通常，重写函数并绑定过去的变量能够减少调用本身（非递归）函数集合元素的运行时间。

使用宏替代第 9.2 节中的 max 函数能够将加速系数提高 2。

编写第 11.1 节内部的 swap 函数，能够将它的运行速度提高三分之一。编写第 11.3 节的 swap 内联，基本不能加速。

利用常用情况。应该组织函数正确处理所有情况并高效处理普通情况。

第 9.1 节中，Van Wyk 的存储分配器能正确处理所有结点大小，而且能够特别高效地处理最常用的结点大小。

第 6.1 节中，Appel 使用专用的、小的时间步处理昂贵的邻近对象，这就允许其程序

的其他部分使用更为高效的、大的时间步。

协同例程。通常，使用协同例程能够将多通道算法转换为单通道算法。

第2.8节中的变位词程序使用了管道，这能通过一组协同例程来实现。

递归函数变化。通过下面的转换能够减少递归函数的运行时间：

将递归重写为迭代，如第13章中的链表和二分查找树。通过使用不同的程序栈将递归转化为迭代（如果某个函数仅仅包含一个对自身的递归调用，那么就没有必要将返回地址存储在栈中）。

如果函数的最后一步是递归调用自身，那么使用一个到其第一条语句的分支来替换该调用，这通常叫做消除尾递归，能够使用答案11.9中的代码实现该转换。通常能够将该分支转换为循环，由编译器来执行这步优化。

通常使用辅助过程能够更高效地解决这些小的子问题，而不是将问题的大小变为0或1。第11.3节中的qsort4函数使用削减了近50的值。

- *并发*。在基本的硬件条件下，构建的程序应该能够尽可能地利用并发。

表示规则

初始化编译时间。在程序执行之前，尽可能初始化变量。

利用代数恒等式。如果表达式的计算非常昂贵，就应使用较为便宜的代数恒等表达式替换它。

第9.2节中，Appel使用乘法和加法替代了昂贵的三角函数操作，同时使用单调函数消除了昂贵的平方根操作。

第9.2节中使用廉价的if语句替换内部循环中昂贵的C求余操作符%。

通常，我们可以使用向左或向右移位来实现幂的乘或除。答案13.9使用位移段替换了任意的除。答案10.6将10移位4来替换除操作。

在第6.1节中，Appel利用数据结构的加法数值精度来用更为快速的32位数值替换了64位浮点数值。

尽量减少数组元素上的迭代循环，使用加法替代乘法。很多编译器进行了该项优化。该技巧生成了递增算法的一个大类。

消除通用子表达式。如果连续两次计算了同一个表达式，并且它的所有变量都没有任何改动，那么就应该避免第二次计算，只需要排序第一次的计算结果并将它用于第二次计算中。

现在，编译器都能消除不包含函数调用的常用子表达式。

配对计算。如果总是同时计算两个类似的表达式，那么就应该建立一个新的过程，

将它们成对计算。

在第 13.1 节中，我们的第一个伪码总是使用 member 和 insert 函数。如果 insert 的参数早就在集合中，C++代码就使用不完成任何操作的 insert 替代这两个函数。

利用单词并发。使用基本计算机体系结构的完整数据路径宽度计算昂贵的表达式。

问题 13.8 显示了如何通过操作 char 或 int 一次完成很多位上的位向量。

答案 9.7 并发统计位数。

附录5　C++中的查找类

下面是在第 13 章中讨论的 C++整数集合表示类的完整清单。网站（www.Programmingpearls.com）上提供了完整的代码。

```
class IntSetSTL {
private:
    set<int> S;
public:
    IntSetSTL(int maxelms, int maxval) { }
    int size() { return S.size(); }
    void insert(int t) { S.insert(t);}
    void report(int *v)
    {   int j = 0;
        set<int>::iterator i;
        for (i = S.begin(); i != S.end(); ++i)
            v[j++] = *i;
    }
};


class IntSetArray {
private:
    int n, *x;
public:
    IntSetArray(int maxelms, int maxval)
    {   x = new int[1 + maxelms];
        n = 0;
        x[0] = maxval;
    }
    int size() { return n; }
    void insert(int t)
    {   for (int i = 0; x[i] < t; i++)
            ;
        if (x[i] == t)
            return;
        for (int j = n; j >= i; j--)
            x[j+1] = x[j];
        x[i] = t;
        n++;
    }
    void report(int *v)
    {   for (int i = 0; i < n; i++)
            v[i] = x[i];
    }
};
```

```
class IntSetList {
private:
    int n;
    struct node {
        int val;
        node *next;
        node(int v, node *p) { val = v; next = p; }
    };
    node *head, *sentinel;
    node *rinsert(node *p, int t)
    {   if (p->val < t) {
            p->next = rinsert(p->next, t);
        } else if (p->val > t) {
            p = new node(t, p);
            n++;
        }
        return p;
    }
public:
    IntSetList(int maxelms, int maxval)
    {   sentinel = head = new node(maxval, 0);
        n = 0;
    }
    int size() { return n; }
    void insert(int t) { head = rinsert(head, t); }
    void report(int *v)
    {   int j = 0;
        for (node *p = head; p != sentinel; p = p->next)
            v[j++] = p->val;
    }
};



class IntSetBST {
private:
    int n, *v, vn;
    struct node {
        int val;
        node *left, *right;
        node(int v) { val = v; left = right = 0; }
    };
    node *root;
    node *rinsert(node *p, int t)
    {   if (p == 0) {
            p = new node(t);
            n++;
        } else if (t < p->val) {
            p->left = rinsert(p->left, t);
        } else if (t > p->val) {
            p->right = rinsert(p->right, t);
        } // do nothing if p->val == t
        return p;
    }
    void traverse(node *p)
```

```
    {   if (p == 0)
            return;
        traverse(p->left);
        v[vn++] = p->val;
        traverse(p->right);
    }
public:
    IntSetBST(int maxelms, int maxval) { root = 0; n = 0; }
    int size() { return n; }
    void insert(int t) { root = rinsert(root, t); }
    void report(int *x) { v = x; vn = 0; traverse(root); }
};

class IntSetBitVec {
private:
    enum { BITSPERWORD = 32, SHIFT = 5, MASK = 0x1F };
    int n, hi, *x;
    void set(int i)  {       x[i>>SHIFT] |=  (1<<(i & MASK)); }
    void clr(int i)  {       x[i>>SHIFT] &= ~(1<<(i & MASK)); }
    int  test(int i) { return x[i>>SHIFT] &   (1<<(i & MASK)); }
public:
    IntSetBitVec(int maxelms, int maxval)
    {   hi = maxval;
        x = new int[1 + hi/BITSPERWORD];
        for (int i = 0; i < hi; i++)
            clr(i);
        n = 0;
    }
    int size() { return n; }
    void insert(int t)
    {   if (test(t))
            return;
        set(t);
        n++;
    }
    void report(int *v)
    {   int j = 0;
        for (int i = 0; i < hi; i++)
            if (test(i))
                v[j++] = i;
    }
};
class IntSetBins {
private:
    int n, bins, maxval;
    struct node {
        int val;
        node *next;
        node(int v, node *p) { val = v; next = p; }
    };
    node **bin, *sentinel;
    node *rinsert(node *p, int t)
    {   if (p->val < t) {
            p->next = rinsert(p->next, t);
        } else if (p->val > t) {
            p = new node(t, p);
            n++;
```

```
        }
        return p;
    }
public:
    IntSetBins(int maxelms, int pmaxval)
    {   bins = maxelms;
        maxval = pmaxval;
        bin = new node*[bins];
        sentinel = new node(maxval, 0);
        for (int i = 0; i < bins; i++)
            bin[i] = sentinel;
        n = 0;
    }
    int size() { return n; }
    void insert(int t)
    {   int i = t / (1 + maxval/bins);
        bin[i] = rinsert(bin[i], t);
    }
    void report(int *v)
    {   int j = 0;
        for (int i = 0; i < bins; i++)
            for (node *p = bin[i]; p != sentinel; p = p->next)
                v[j++] = p->val;
    }
};
```

部分问题的答案提示

第 1 章

4. 阅读第 12 章。

5. 考虑两通道算法。

6、8、9. 使用键索引。

10. 考虑散列，并且不要将自己局限在计算系统中。

11. 针对鸟类讨论该问题。

12. 不使用笔如何写字？

第 2 章

1. 考虑排序、二分查找和签名。

2. 争取获得运行时间为线性的算法。

5. 利用恒等式：$cba=(a^Tb^Tc^T)^T$。

7. Vyssotsky 使用了一个系统工具和两个一次性程序，他编写这两个程序仅仅是为了重新组织磁带上的数据。

8. 考虑集合中 k 个最小的元素。

9. s 顺序查找的成本和 sn 成正比。s 二分查找的总成本是查找的成本加上排序表所需的时间。在考虑各类算法的常量因子前，请看问题 9.9。

10. 阿基米德是如何判断出皇冠不是纯金的？

第 3 章

2. 使用一个数组表示递归的系数，使用另一个数组表示前面 k 个值。程序在一个循环内部包含另一个循环。

4. 只有一个函数是需要从头开始编写的；其他两个函数都可以调用它。

第4章

2. 使用精确的不变式。考虑在数组中添加两个空元素初始化不变式：x[-1]=-∞并且x[-1]=∞。

5. 如果您解决了这个问题，跑到最近的数学部门并要求得到一位博士学位。

6. 查看过程保留的不变式，并将 can 的初始条件和它的终止条件关联起来。

7. 再次阅读第 2.2 节。

9. 使用下面的循环不变式，在 while 语句测试之前，它们为真。对于向量加法，

$i \leq n \quad \&\& \quad \forall_{1 \leq j < i}\, a[j] = b[j] + c[j]$

对于顺序查找，

$i \leq n \quad \&\& \quad \forall_{1 \leq j < i}\, x[j] \neq t$

11. 参见答案 11.14 中的递归函数，该函数将一个指针传递给数组。

第5章

3. 查找“mutation testing”这类术语。

5. 在 O(log n)或 O(1)额外比较中能够获得什么？

6. 本书的网站上包含了一个 GUI Java 程序，使用该程序学习排序算法。

9. 脚手架的 Tab 分隔的输出格式兼容多数的电子表格。我通常将一系列的相关实验存储在同一页电子表格上，同时还存储了它们的性能以及我为什么做这个实验和能从中学到什么的注释。

第6章

1. 查看第 8.5 节。

3. 修改附录 3 中介绍的成本模型的运行时间，衡量双精度操作的成本。

7. 通过检查驾驶培训、严格限制开车速度、限制最小驾车年龄，严惩酒后驾车，以及建立一个良好的公共交通运输系统来避免机动车交通事故。如果确实发生了交通事故，通过成员座仓、扣上安全带（可能要通过法律进行强制）和安全气囊可以减少乘客受伤程度。如果乘客受伤了，通过现场的护理人员、救护直升机的快速送往医院、外伤中心和校正手术能够减小伤害造成的后果。

第7章

5. 首先使用函数$(1+x/100)^{72/x}$，然后使用电子表格绘制$(1+0.72/x)^x$。为了证明 72 法则的属性，记住 $\lim_{n\to\infty}(1+c/n)^n = e^c$，2 的自然对数大约为 0.693，并且渐近线并不总是最佳逼近线。

8. 认真思考问题 2.7、8.10、8.12、8.13、9.4、10.10、11.6、12.7、12.9、12.11、13.3、13.6、13.11、15.4、15.5、15.7、15.9 和 15.15，以及第 1.3、2.2、2.4、2.8、10.2、12.3、13.2、13.3、13.8、14.3、14.4、15.1、15.2 和 15.3 节中的设计和程序。

第8章

4. 绘制随机遍历的累加和。

7. 浮点加法不一定需要交换。

8. 除了计算区域中的最大和，返回数组每端最大向量结束的信息。

10、11、12 使用累加数组。

13. 显式算法的运行时间为 $O(n^4)$；获得立方算法的时间度。

第9章

3. 由于最多只能将 k 增加 n-1，所以 k 肯定小于 2n。

9. 要使得即便是当 n 非常小的时候，二分查找也比顺序查找更具有竞争力，只要让比较操作的代价十分昂贵就可以了。

第10章

1. 编译器生成了什么代码访问压缩字段？

5. 混合并匹配函数和表。

7. 通过考虑特定范围的等价内存减少数据。这些范围可以是固定长度的块（如 64 字节）或函数边界。

第11章

2. 将循环下标 i 从高值变化到低值，这样它逐渐接近 x[l]中的已知值 t。

4. 当你需要解决两个子问题时，你应该马上解决哪个问题？应该将哪个问题留给栈以后返回——大的还是小的？

9. 修改快速排序，让它仅在包含 k 的子范围内递归。

第 12 章

4. 找到一个统计学家并使用“Coupon Collector's Problem”和“Birthday Paradox”这类词汇向他发问。

11. 该问题说了你能够使用计算机，但并不是说必须使用计算机。

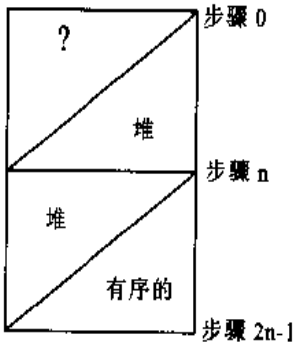
第 13 章

2. 必须进行错误检查，以确保被插入的整数在正确的范围内并且数据结构还没有满。析构函数应该返回所有分配的存储空间。

3. 使用二分查找测试某个元素是否在有序数组中。

第 14 章

2. 目标是具有如下结构的堆排序。



3. 仔细查看问题 2，并且考虑将代码移出循环。

6. 堆中具有从结点 i 到结点 2i 的隐含指针；在磁盘文件中使用同一技术。

7. x[0..6]中的二分查找使用根为 x[3]的隐形树。如何使用第 14.1 节中的隐形树？

9. 在排序中使用 O(n log n)下边界。如果 insert 和 extractmin 的运行时间小于 $O(\log n)$，那么排序时间可以小于 O(n log n)；说明如何更快地排序。

第15章

15. 假设，现在我们正从仅在单个词组“x y x z”中包含单词x、y和z且具有100万个单词的文档生成order-1马尔科夫文本。在一半情况下x后面跟着y，另一半情况下可能跟着z。Shannon的算法给出了哪些可能性？

16. 如何使用字母和单词的k链计数？

17. 一些商业语音识别程序基于三链统计。

部分问题的答案

第 1 章的答案

1. 该 C 程序使用标准库 qsort 排序一个整数文件。

```
int intcomp(int *x, int *y)
{   return *x - *y; }

int a[1000000];
int main(void)
{   int i, n=0;
    while (scanf("%d", &a[n]) != EOF)
        n++;
    qsort(a, n, sizeof(int), intcomp);
    for (i = 0; i < n; i++)
        printf("%d\n", a[i]);
    return 0;
}
```

这个 C++程序使用标准模板库中的 set 容器完成相同的任务。

```
int main(void)
{   set<int> S;
    int i;
    set<int>::iterator j;
    while (cin >> i)
        S.insert(i);
    for (j = S.begin(); j != S.end(); ++j)
        cout << *j << "\n";
    return 0;
}
```

答案 3 概述了两个程序的性能。

2. 这些函数使用常量来设置、清除并测试位值：

```
#define BITSPERWORD 32
#define SHIFT 5
#define MASK 0x1F
#define N 10000000
int a[1 + N/BITSPERWORD];

void set(int i) {       a[i>>SHIFT] |=  (1<<(i & MASK)); }
void clr(int i) {       a[i>>SHIFT] &= ~(1<<(i & MASK)); }
int  test(int i){ return a[i>>SHIFT] &   (1<<(i & MASK)); }
```

3. 该C代码使用答案2中定义的函数实现排序算法。

```
int main(void)
{   int i;
    for (i = 0; i < N; i++)
        clr(i);
    while (scanf("%d", &i) != EOF)
        set(i);
    for (i = 0; i < N; i++)
        if (test(i))
            printf("%d\n", i);
    return 0;
}
```

我使用答案4中的程序生成包含100万个不重复正整数的文件，每个正整数都小于1000万。下表报告了使用系统命令行排序、答案1中的C++和C程序、位图代码对它们进行排序所需的成本：

	系统排序	C++/STL	C/qsort	C/位图
总时间（秒）	89	38	12.6	10.7
计算时间（秒）	79	28	2.4	0.5
MB	0.8	70	4	1.25

第一行是总时间，第二行将读写文件的输入/输出时间减少了10.2秒。虽然通用C++程序使用的内存和CPU时间是专用C程序的50倍，但是它仅需要一半的代码，并能很容易地扩展到其他问题上。

4. 参见第12章，特别是问题12.8。该代码假设randint(l，u)返回1..u之间的一个随机整数。

```
for i = [0, n)
    x[i] = i
for i = [0, k)
    swap(i, randint(i, n-1))
    print x[i]
```

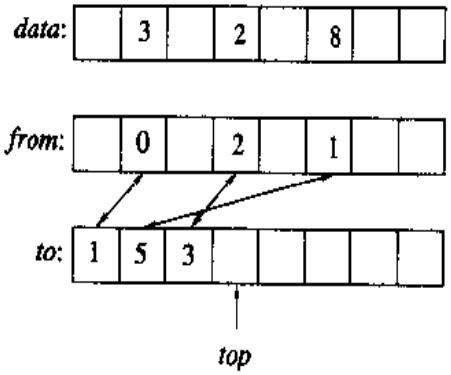
swap函数交换了x中的两个元素。在第12.1节中详细讨论了Randint函数。

5. 使用位图表示1000万个数值需要很多位，或125万个字节。利用没有哪个电话号码以数字0或1开始这个事实能够将内存需求减少为100万个字节。另外，双通道算法首先使用5000000/8=625000个字的内存排序0～4999999之间的整数，然后在第二个通道中排序5000000～9999999之间的整数。K通道算法在kn时间和n/k空间内排序n个不重复的正整数。

6. 如果每个整数最多出现10次，那么我们就可以使用半个字节4位（或者四位组）统计它出现的次数。使用问题5的答案，我们可以使用10000000/2字节在单通道排序整

个文件，或使用 10000000/2 k 字节在 k 通道内排序整个文件。

9. 使用另外两个 n 元素向量 from 和 to，以及整数 top 中包含的符号能够初始化向量 data[0..n-1]。如果已经初始化元素 data[i]，那么 from[i]<top 并且 to[form[i]] = i。因此，from 是一个简单的符号，to 和 top 一起确保了 from 不会写上内存的随机内容。下图中没有初始化 data 的空条目：



变量 top 初始化为 0，下面的代码首先访问数组元素 i：

```
from[i] = top
to[top] = i
data[i] = 0
top++
```

这个问题和答案来自 Aho、Hopcroft 和 Ullman 编写的《*Design and Analysis of Computer Algorithms*》（由 Addison-Wesley 在 1974 年出版）中的练习 2.12。他结合了关键字索引和深奥的签名方法，适用于矩阵和向量。

10. 商店将纸质订单表格放在 10×10 的桶数组中，使用客户电话号码的最后两位作为散列索引。当客户打电话来预定时，就将它放置在适当的桶中。当客户来取商品时，销售人员就按序查看对应桶中的订单——这就是经典的“顺序查找带有冲突解决的开放式散列”。电话号码的最后两个数字非常随机，因此是非常优秀的散列函数，而前面两个数字却是可怕的散列函数——为什么呢？一些市民使用类似的方法来在记事本中记录各种信息。

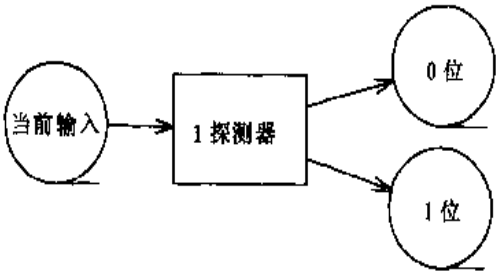
11. 两台设备上的计算机是通过微波连接的，但是当时测试站打印图片所需的打印机却是非常昂贵的。因此，该团队将图片绘制在主设备上，拍摄它们，然后将 35mm 的底片通过信鸽发送到测试站，在那里放大并打印图片。鸽子 45 分钟来回一次，是汽车所需时间的一半，并且每天只需要几美元。在项目开发的 16 个月中，信鸽传递了几百卷的底片，仅丢失了两卷（当地的传播习惯，没有传送分类信息）。由于现在打印机比较便宜，所以使用微波连接解决该问题。

12. 根据都市传闻，苏联人用铅笔就解决了这个问题。至于这个真实故事的背景，

请查看www.spacepen.com。Fisher Space Pen 公司成立于 1948 年，俄国航天署、水底探测人员和喜马拉雅山脉登山者使用了它的书写设备。

第 2 章的答案

A. 查看以 32 位表示每个整数的二分查找法非常有用。我们在算法的第一个通道读取了 40 亿个输入整数，并在这些输入整数前加上一个 0 位，然后输出到一个顺序文件中，在这些输入整数前加一位 1 并输出到另一个文件中。



因为这两个文件中的一个最多包含 20 亿个整数，所以，我们接着将该文件用作当前输入，并重复探测过程，但是这次探测的是第二位。如果原来的输入文件包含 n 个元素，那么第一个通道将读取 n 个整数，第二个通道最多读取 n/2 个整数，第三个通道最多读取 n/4 个整数，以此类推，所以总的运行时间和 n 成正比。通过排序文件及扫描能够找到丢失的整数，但是这样做后运行时间将和 n log n 成正比。Illinois 大学的 Ed Reingold 将这个问题作为一次测验的题目。

B. 参见第 2.3 节。

C. 参见第 2.4 节。

1. 为了找出给定单词的所有变位词，我们首先计算它的符号。如果不能进行任何预处理，那么只能顺序读取整个目录，计算每个单词的符号，并比较两个符号。如果能够进行预处理，就可以在一个包含根据符号排序的(符号、单词)对中执行二分查找。Musser 和 Saini 在他们的《*STL Tutorial and Reference Guide*》(这本书是在 1996 年由 Addison-Wesley 出版的）一书的第 12 章～第 15 章实现了几个变位词程序。

2. 二分查找在一个包含一半以上整数的子区间递归查找至少出现两次的单词。我原来的解决方法不能保证每次迭代都能将整数数目折半，所有 $\log_2 n$ 通道的最差情况下的运行时间和 n log n 成正比。Jim Saxe 观察到查找能够避免带有太多的重复元素，这样就将运行时间减为线性时间。当他的查找程序发现在当前 m 个整数中可能有重复元素时，它就会在当前工作磁带上存储 m+1 个整数；如果磁带上可能出现更多的整数，他的程序就会删除它们。虽然他的方法经常忽略输入变量，它的策略却非常保守，能够确保至少

找到一个重复元素。

3. 下面的代码很具有技巧性，它通过 rotdist 将 x[n]左旋。

```
for i = [0, gcd(rotdist, n))
    /* move i-th values of blocks */
    t = x[i]
    j = i
    loop
        k = j + rotdist
        if k >= n
            k -= n
        if k == i
            break
        x[j] = x[k]
        j = k
    x[j] = t
```

rotdist 和 n 的最大公因子是要置换的循环数（在现代代数学中，就是旋转归纳中的旁系数）。

下一个程序来自 Gries 的《*Science of Programming*》的第 18.1 节。它假设函数 swap(a,b,m)将 x[a..a+m-1]和 x[b..b+m-1]交换。

```
if rotdist == 0 || rotdist == n
    return
i = p = rotdist
j = n - p
while i != j
    /* invariant:
        x[0  ..p-i  ] in final position
        x[p-i..p-1  ] = a (to be swapped with b)
        x[p  ..p+j-1] = b (to be swapped with a)
        x[p+j..n-1  ] in final position
     */
    if i > j
        swap(p-i, p, j)
        i -= j
    else
        swap(p-i, p+j-i, i)
        j -= i
swap(p-i, p, i)
```

第 4 章中介绍了循环不变式。

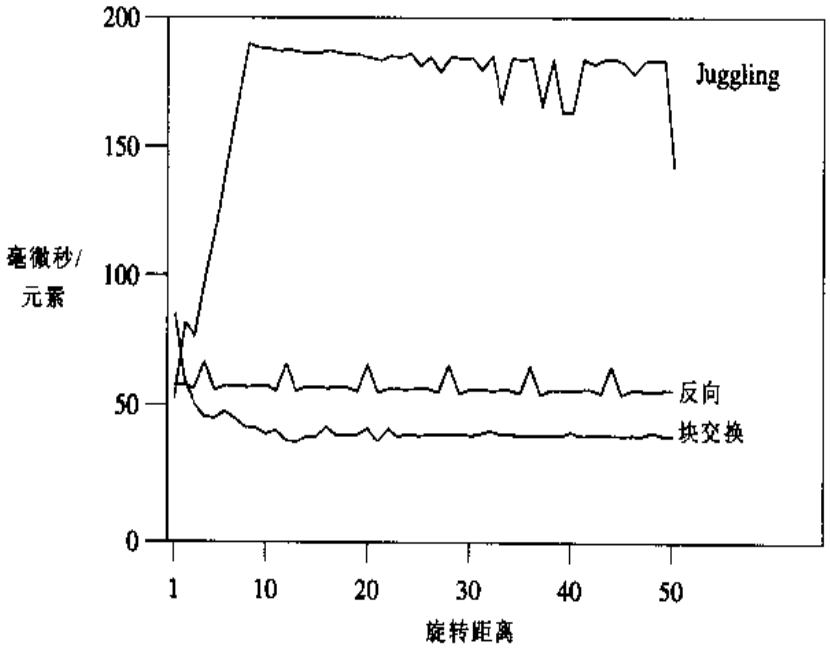
该代码和计算 i 和 j 的最大公因子的欧几里德算法是同构的（虽然慢但是正确），假设所有的输入都不为零。

```
int gcd(int i, int j)
    while i != j
        if i > j
            i -= j
```

```
        else
            j -= i
    return i
```

Gries 和 Mills 研究了《Cornell University Computer Science Technical Report 81-452》中的“交换区域”里的三个旋转算法。

4. 我在一台 400MHz Pentium II 机器上运行这三个算法，n 为 1000000，旋转距离为 1～50。该图绘制了在每个数据集上 50 次运行的平均时间：



反向代码的运行时间比较一致，大概每个元素 58 纳秒，当旋转距离为 4、12、20 及所有模 8 余 4 的整数时，运行时间大概跳到 66 纳秒（这可能和 32 个字节的缓存大小交互作用）。块交换算法是多数昂贵的算法的开始（可能是由于交换单个元素块的函数调用引起的），但是它具有良好的高速缓存性能，这使得当旋转距离大于 2 时它具有最快的执行效率。Juggling 算法是最廉价的，但是它的高速缓存性能很差（访问每个 32 字节高速缓存线的单个元素），这就导致当旋转距离为 8 时运行时间就达到了 200 纳秒。它的时间在 190 纳秒左右浮动，稍微有所下降（当旋转距离为 1000 时，它的运行时间降为 105 纳秒，然后马上恢复到 190）。在 20 世纪 80 年代中叶，这段相同的代码将旋转距离设置为页大小从而利用了页的性能。

6. 签名是按钮编码，所以签名“LESK*M*”就成为“5375*6*”。为了在字典中找到错误的匹配，我们给每个按钮编码签名，并根据签名排序（并根据相同签名的名字），然后顺序读取有序文件，输出具有不同名字的相同签名。为了检索给定了按钮编码之后的名字，就可以使用包含签名和其他数据的结构。我们可以排序该结构，然后使用二分查找排序按钮编码，在真实的系统中，可以使用散列技术或数据库系统。

7. 为了转置以行为主的矩阵，Vyssotsky 预先考虑了每条记录的列和行，然后调用系统磁带排序按列排序，再按行排序，然后使用另一个程序删除列号和行号。

8. 该问题的“啊哈!”洞察力就是，当且仅当子集包含 k 个最小元素时，k 元素子集之和最大为 t。如果通过排序原集合来查找子集，那么所需的时间就和 n log n 成正比，如果使用选择算法（查看答案 11.9）来查找子集，所需的时间就和 n 成正比。当 Ullman 将这个作为课堂作业布置时，学生所设计的算法的运行时间各式各样，有上面提到的，还有 O(n log k)、O(nk)、$O(n^2)$和 $O(n^k)$。你能否找出一个具有上面运行时间的自然算法？

10. 爱迪生在灯泡中充满了水，然后将这些水倒到一个具有刻度的圆柱体中（如果你注意提示可能就会发现，阿基米德也使用水来计算体积，在他那个时代，“啊哈!”洞察力受到了大声的欢迎）。

第 3 章的答案

1. 税收表中的每一项都包含三个值：该等级的下界、基本税收及根据下界对收入收税时的税率。在表中增加一个具有“无限”下界的最终哨兵项能够简化顺序查找代码的编写，并提高执行效率（参见第 9.2 节）；当然也可以使用二分查找。这些技术能够用于任何分段线性函数。

3. 该块字母“I”

```
XXXXXXXXX
XXXXXXXXX
XXXXXXXXX
   XXX
   XXX
   XXX
   XXX
   XXX
   XXX
XXXXXXXXX
XXXXXXXXX
XXXXXXXXX
```

可以编码成：

```
3 lines 9 X
6 lines 3 blank  3 X  3 blank
3 lines 9 X
```

或更加紧凑：

```
3 9 X
6 3 b 3 X 3 b
3 9 X
```

4. 如果想要找出两个日期之间相差多少天，就需要计算这两个日期所在年份的天数，然后用后面一个减去前面一个（可能来自年），然后加上 365 乘上年份之差，再加上一，表示平年。如果想要找出给定某天是星期几，就计算给定日期和已知星期日之间的天数，然后使用模块算法将它转化为星期几。如果要准备给定年份的某个月的日历，就需要知道该月有多少天（注意正确处理二月份），以及该月 1 日是星期几。Dershowitz 和 Reingold 将这些内容编写成了整本书《*Calendrical Calculations*》（1997 年由剑桥大学出版社出版）。

5. 由于对单词的比较是从右到左进行的，所以要将单词按相反顺序存储（从右到左）可能需要付出一些代价。可以使用字符二维数组（通常比较浪费）、使用终止符分隔后缀的单字符数组、具有指向各个单词的指针数组的字符数组。

6. Aho、Kernighan 和 Weinberger 在他们《*Awk Programming Language*》（1988 年由 Addison Wesley 出版）一书的第 101 页给出了一个仅九行的程序，该程序的作用是生成信件。

第 4 章的答案

1. 证明，当在不变式中添加条件 0≤l≤n 和-1≤u＜n 之后不会出现溢出现象，然后我们就能够连通 l+u。也可以使用相同的条件来证明永远都不会访问数组边界之外的元素。如果我们在第 9.3 节中定义了虚构的边界元素 x[-1]和 x[n]，那么就能正式将 mustbe(l,u)定义成 x[l-1]<t 并且 x[u_l]>t。

2. 参见第 9.3 节。

5. 若要查看这个著名的、公开的数学问题的介绍，请参见 1984 年 1 月 B.Hayes 在 Scientific American 上的 Computer Recreations 专栏中发表的《On the ups and downs of hailstone numbers》。如果想进一步讨论技术问题，请参见 1985 年 1 月 J.C.Lagarias 在 American Mathematical Monthly 上发表的《The 3x+1 problem and its generalizations》。在本书出版之际，Lagarias 大概有 30 页的文献，其中有大约 100 个参考了 www.research.att.com/~jcl/3x+1.html 上的内容。

6. 由于每一步都将罐子中的豆子减少一粒，所以该过程最终能够终止。它总是从咖啡罐中拿掉零个或两个白豆，所以不变式就是白豆数目的奇偶校验。因此，当且仅当罐子中原来的白豆数为奇数时，最后剩下的豆子才可能是白色的。

7. 由于构成梯子的线段在 y 方向是递增的，所以可以通过二分查找找到给定点的两端范围。查找中的基本比较说明了这个点是在给定线段之下、之上还是就在线段上。你会如何编写该函数呢？

8. 参见第 9.3 节。

第 5 章的答案

1. 编写大型程序时，我使用长名称表示全局变量（10 或 12 个字符）。本章使用 x、n 和 t 这类短变量名。在多数软件项目中，最短的合理名称可能类似于 elem、nelems 和 target。我发现建立脚手架的时候使用短名字比较方便，并且在第 4.3 节中类似的数学证明中也必须使用短名称。将类似的规则应用到数学中：对数学不熟悉的人可能希望听到“直角三角形相邻两条直角边的平方和等于斜边的平方”，而处理该问题的人通常会说“$a^2+b^2=c^2$”。

我尽量保持 Kernighan 和 Ritchie 的 C 编码风格，但是我在第一行代码中放置了函数的开始花括号，并删除了其他空行，以节省版面（对于本书的小函数而言，这占了很大一个百分比）。

如果不存在该值，那么第 5.1 节中的二分查找就返回整数-1，如果存在这个值，那么就指向这个值。Steve McConnell 建议查找应该返回两个值：一个布尔值，说明是否存在该值，仅当布尔值为真时，返回一个下标：

```
boolean BinarySearch(DataType TargetValue, int *TargetIndex)
 /* precondition: Element[0] <= Element[1] <=
        ... <= Element[NumElements-1]
    postcondition:
        result == false =>
            TargetValue not in Element[0..NumElements-1]
        result == true  =>
            Element[*TargetIndex] == TargetValue
 */
```

McConnell 的《*Code Complete*》一书的第 402 页的程序清单 18.3 是一个 Pascal 插入排序，占一页（一大页）；代码和注释加起来总共 41 行。该风格比较适合于大型软件项目。本书的第 11.1 节给出了一个仅需 5 行的代码。

只有很少的程序具有错误检查。一些函数将文件中的数据读入到大小为 MAX 的数组中，并且 scanf 调用极容易溢出缓存。作为参数的数组参数为全局变量。

在本书中，我使用了适用于教科书和脚手架的快捷方式，但是不适用于大型软件项目。如 Kernighan 和 Pike 在他们的《*Practice of Pragramming*》一书的第 1.1 节中观察到的“清晰总是通过简明扼要获得的。”即便这样，我的多数代码都避免使用第 14.3 节所演示的 C++代码中难以置信的密集风格。

7. 当 n=1000，每次按序查找整个数组需要 351 纳秒，但是如按随机顺序查找它，就会将成本提高到 418 纳秒（大约减速 20%）。当 $n=10^6$，即便二级缓存，试验也溢出，

并且减速因子为 2.7。对于第 8.3 节中高度优化的二分查找，有序查找能够在 125 纳秒内查找 n=1000 个元素的表，但是随机查找需要 266 纳秒的时间，减速因子为 2。

第 6 章的答案

4. 您是否希望您的系统是可靠的？那么，在设计初期就必须注意可靠性，如果一开始的时候没有注意，以后就很难弥补。设计的数据结构必须要达到当结构受到损害的时候，能够恢复信息。通过浏览和遍历审查代码，并广泛测试代码。在可靠的操作系统上及使用错误纠正内存的冗余硬件上运行您的软件。设定一个计划，当系统崩溃时（它当然会崩溃）能够快速恢复。详细记录每个错误，以便以后学习。

6. “首先是要确保它能够运行，其次才是提高运行效率”这是一个非常好的建议。然而，Bill Wulf 仅仅花了几分钟时间就让我相信了这个古老的名言并没有我以前想像得那么正确。他使用了文档产品系统这个例子来说服我，这个系统需要几个小时才能准备一本书。Wulf 的论点是：“这个程序，类似于任何其他大型系统，今天发现了 10 个小错误。下个月将发现 10 个不同的错误。如果让您选择是纠正 10 个当前的错误还是让程序运行速度加快 10 倍，您会选择哪一个？”

第 7 章的答案

在本书付诸出版时，下面这些答案猜测的数字可能会偏离正确值 2 倍，但是不会偏离得更多。

1. 即便是从新泽西州帕特森美丽的瀑布上 80 英尺的高处流下来，帕塞伊克河的流速也达不到 200 英里每小时。但是我怀疑工程师确实告诉记者该河流最大的流速能够达到 200 英里每天，比一般的 40 英里每天要快五倍，一般流速不到 2 英里每小时。

2. 老式的可移动硬盘有 100MB。ISDN 线每秒钟传递 112 千位，或每小时传输 50MB。这就相当于在骑自行车的人的口袋中放入一个盘，然后让他骑车 2 小时或绕半径 15 英里的圆一周。在一个更加有趣的比赛中，将 100 个 DVD 放在骑自行车的人的背包里，而它的带宽系数上升了 17000；将 ATM 的线更新为 155MB/秒，能够将系数增加 1400。这就给骑车者另一个系数 12，或要多骑一天。（在我写完这段文字的第二天，在同事的办公桌上发现堆着 200 张 5GB 一次写入的唱片。在 1999 年，出现了非标准的万亿媒体）。

3. 软盘的容量为 1.44MB。老实说，我的打字速度为每分钟 55 个单词（或 300 字节）。因此在 4800 分钟，或 80 小时之内我能够填满整个软盘（本书的输入文本仅有 0.5MB，

但是我却需要三天以上的时间才能输入完毕)。

4. 我希望的答案是每行 10 纳秒的指令大约需要百分之一秒，以及 3 小时的 11 毫秒磁盘转速（5400 转/分钟），进行 6 个小时的 20 毫秒查找，并且执行大概一个月的两秒的名字输入。一个聪明的读者写到，“它需要多长时间？如果时钟也慢下来，那么时间就和前面差不多。”

5. 速率在 5%和 10%之间，72 法则估算的精确度可以达到 1%以内。

6. 由于 72/1.33 近似于 54，我们能够预算到 2052 年人口将翻倍（UN 估计值号召大量降低人口增长率）。

9. 忽略由于查询而导致的减速，每次磁盘操作 20 毫秒（查找时间）的话，那么每个事务耗时 2 秒，也就是说每小时处理 1800 个事务。

10. 你可以通过统计报纸上的死亡通告估算本地的死亡率以及当地的人口数。一个更为简单的方法是使用利特尔法则估算死亡年龄。例如，如果死亡年龄为 70 年，那么每年的死亡人口就为 1/70 或 1.4%。

11. Peter Denning 对利特尔法则的证明可以分为两部分。“首先，定义 λ =A/T，即达到速率，其中 A 是在 T 这段观察时间内到达的数目。定义 X=C/T，即输出速率，其中 C 是 T 时间内的完成数。N(t)表示在[o,T]内的时间 t 卜系统中的数目。W 是 n(t)下的区域，在 item-seconds 的单位中，表示观察期间系统中所有元素的等待时间总和。每个元素完成的平均响应时间定义为 R=W/C，单位为(item-seconds)/(item)。系统的平均数是 n(t)的平均高度，即 L=W/T，单位为(item-seconds)/(second)。现在很明显 L=RX。这个公式仅就输出速率而言。没有必要进行流量平衡，例如，具有相同的输出流量（符号，λ=X)。如果你添加了这个假设条件，公式就变成 L=λ*R，这是查询和系统理论中遇到的公式。”

12. 当我读到一枚 25 分硬币的“平均寿命为 30 年”时，我觉得这个数字对我来说太大了，我记得我没看到过多少古老的硬币。因此，我就伸进口袋，找到 12 枚 25 分的硬币。下面是它们的年龄：

4 5 7 9 9 12 17 17 19 20 34

平均年龄为 13 年，这和硬币的平均寿命为它的 2 倍相一致（在分布相当一致的年龄中)。我找到了大量年龄都少于 5 年的硬币，我可以进一步研究这个问题。然而，这次我认为这篇论文的结论是正确的。同一论文报告说“至少制造了 7.5 亿枚 25 分的新泽西州硬币”，还说到每 10 周就会发布一种新的 25 分的州硬币。这乘起来就得到了每年有 40 亿枚 25 分的硬币，或每个美国居民有 12 枚新的 25 分的硬币。每个硬币具有 30 年的寿命就意味着每个人有 360 枚 25 分的硬币。如果将这些硬币放在口袋里就太多了，但是如果我们考虑了汽车和家里的零钱、收银机、投币式自动售货机和银行，就很有可能有这么多了。

第 8 章的答案

1. David Gries 在《*Science of Computer Programming* 2》第 207～214 页的“A Note on the Standard Strategy for Developing Loop Invariants and Loops”中系统地推导并验证了算法 4。

3. 算法 1 大致对函数 max 进行了 $n^3/6$ 次调用，算法 2 大致使用了 $n^2/2$ 次调用，算法 4 大致使用了 2n 次调用。算法 2b 为累加数组使用了线性额外空间，算法 3 在栈中使用了对数额外空间。其他算法仅使用了常数额外空间。算法 4 是在线的：一次输入完毕它就计算出答案，这特别适用于处理磁盘上的文件。

5. 如果将 cumarr 声明成

```
float *cumarr;
```

那么赋值

```
cumarr = realarray+1
```

将意味着 cumarr[-1]指向 realarray[0]。

9. 使用赋值 maxsofar=-∞替换 maxsofar=0。如果-∞的使用让你迷惑，也可以使用 maxsofar=x[0]，为什么？

10. 初始化累加数组 cum，使得 cum[I]=x[0]+…+x[I]。如果 cum[l-1]=cum[u]，那么子向量之和就接近于 0。因此，能够通过定位 cum 中两个最接近的元素找到和接近于零的子向量，排序数组之后，能够在 O(n log n)时间内完成该任务。该运行时间接近于最优常量，因为任何能够解决这个问题的算法都能够用于解决“元素惟一性”问题，“元素惟一性”问题主要是判断数组中是否包含重复元素（Dobkin 和 Lipton 表明该问题所需的时间比决策树模型计算的最差情况要更多）。

11. 线性收费站 i 和 j 之间的总成本为 cum[j]-cum[i-1]，其中 cum 是个累加数组，如上所示。

12. 该答案使用另一个累加数组。可以使用赋值语句：

```
for i = [l, u]
    x[i] += v
```

替代循环

```
cum[u] += v
cum[l-1] -= v
```

它象征性地将 x[0..u]加 v，然后从 x[0..l-1]中减去它。执行了所有这些求和之后，可以使用下面的代码计算数组 x：

```
for (i = n-1; i >= 0; i--)
    x[i] = x[i+1] + cum[i]
```

这将最差情况下求 n 个数之和的时间从 $O(n^2)$降为 O(n)。这个问题出现在收集第 6.1 节中介绍的 Appel 的 n 体程序的统计数字中。使用了该解决方法之后将统计函数的运行时间从 4 小时减少为 20 分钟。当需要一年才能执行完这个程序时，这种加速就显得不是很重要了，但是如果仅需要一天就能执行完程序，那么这个加速就非常重要了。

13. 使用算法 2 在长度 m 方向的技巧和算法 4 在长度 n 方向的技巧，可以在时间 $O(m^2n)$内找到 m*n 维数组的子数组的最大值之和。因此，能够在时间 $O(n^3)$内解决 n*n 问题，这是二十几年来最好的结果。Tamaki 和 Tokuyama 于 1998 年在《*Symposium on Discrete Algorithms*》（第 446～452 页）发表了一个速度更快一些的算法，它的运行时间为 $O(n^3[(\log \log n)/(\log n)]^{1/2})$。同时，他们还给出了一个 $O(n^2 \log n)$的近似算法，用来查找总和至少为最大值一半的子数组，并且介绍了它在数据挖掘中的应用。最佳下界和 n^2 成正比。

第 9 章的答案

2. 这些变量有助于实现 Van Wyk 方法的变种。该方法使用 nodesleft 跟踪 freenode 指向的大小为 NODESIZE 的结点的数目。当其变为零时，它分配另一个大小为 NODEGROUP 的组。

```
#define NODESIZE 8
#define NODEGROUP 1000
int nodesleft = 0;
char *freenode;
```

使用对下面 private 版本的调用替换对 malloc 的调用：

```
void *pmalloc(int size)
{   void *p;
    if (size != NODESIZE)
        return malloc(size);
    if (nodesleft == 0) {
        freenode = malloc(NODEGROUP*NODESIZE);
        nodesleft = NODEGROUP;
    }
    nodesleft--;
    p = (void *) freenode;
    freenode += NODESIZE;
    return p;
}
```

如果该调用没有指定大小，那么该函数将马上调用系统 malloc。当 nodesleft 为 0 时，它分配另一个组。如果使用第 9.1 节中配置的相同输入，那么总的运行时间就由 2.67 秒降为 1.55 秒，并且花费在 malloc 上的时间由 1.41 秒降为.31 秒（或新运行时间的 19.7%）。

如果该程序也释放结点，那么一个新变量就指向一个自由结点的单向链表。当释放结点时，就将该结点放到链表的最前面。当链表为空时，该算法分配一组结点，并且通过链表将它们连接起来。

4. 如果提供的一组值按递减顺序排列，那么该算法大致需要使用 2^n 个操作。

5. 如果二分查找算法输出说找到了值 t，那么这个值就在表中。虽然，应用到无序表中时，该算法有时可能输出不存在 t，而实际上 t 在表中。在这种情况下，算法定位一对邻接元素，它能证明如果表是有序的，t 不在表中。

6. 例如，有人可能会使用下面的条件判断一个字符是否是数字

```
if c >= '0' && c <= '9'
```

如果想要判断某个字符是否为字母数字就要进行一系列复杂的比较；如果性能有问题，那么就应该使用最有可能成功的测试条件。通常，使用 256 元素表比较简单并且速度相对比较快：

```
#define isupper(c) (uppertable[c])
```

绝大多数系统在表的每个元素中存储几位，然后通过逻辑提取它们，并且：

```
#define isupper(c) (bigtable[c] & UPPER)
#define isalnum(c) (bigtable[c] & (DIGIT|LOWER|UPPER))
```

C 和 C++程序员可能更喜欢通过查看文件 ctype.h 来看看他们的系统是如何解决这个问题的。

7. 第一个方法是计算每个输入单元（可能是 8 位字符或 32 位整数）中为 1 的位数，然后将它们相加。通过按序查看每位，或者迭代为 on 的位（使用 b&=(b-1)这类语句），或者在一个表中（例如 2^{16}=65536 个元素的表）执行查找，可以找出 16 位整数中为 1 的位数。高速缓存的大小对你时单元的选择有何影响？

第二个方法是计算输入中每个输入单元中的数值，然后将该数值乘以该单元中为 1 的位数。

8. R.G.Dromey 编写了下面的代码来计算 x[0..n-1]中的最大元素，使用 x[n]作为哨兵：

```
i = 0
while i < n
    max = x[i]
    x[n] = max
    i++
    while x[i] < max
        i++
```

11. 使用几个 72 元素表替代函数计算能够将该程序在 IBM 7090 上的运算时间从 30 分钟减少为 1 分钟。由于计算直升飞机的旋翼叶片需要运行该程序三百次，这几百个额外的内存字将 CPU 时间从一周降低为几个小时。

12. Horner 的方法通过

```
y = a[n]
for (i = n-1; i >= 0; i--)
    y = x*y + a[i]
```

计算多项式。他使用 n 乘法，并且比前面的代码快两倍。

第 10 章的答案

1. 将所有访问压缩字段的高级语言指令编译成很多机器指令，而访问未压缩字段值需要很少的指令。将记录解压缩之后，Feldman 稍微增加了数据空间，但是也大大减少了代码空间和运行时间。

2. 一些读者建议存储（x, y, pointnum）三元组时如果 x 相同则根据 y 排序，然后可以使用二分查找来查找给定的（x, y）对。如果已经根据 x 值（如上，在 x 相同的情况下根据 y 排序）排序输入，那么本书中介绍的数据结构很容易建立。在 row 数组的 firstincol[i]和 firstincol [i+1]-1 之间格式化二分查找能够更快地查找该结构。注意，这些 y 值按递增顺序出现，而且二分查找必须正确处理查找空子数组的情况。

4. Almanacs 将城市间的距离作为三角数组存储在表中，这能将它们的空间减少一半。有时，数学表仅存储函数的最少有效位数字。并且对于每行值，仅一次给出最多有效位数字。电视节目表仅仅说明节目开始时间，从而节省空间（没有按照每 30 分钟的间隔列出所有的节目）。

5. 在该表中，Brooks 使用了两种表示方法。函数在真实的答案之间，并且存储在数组中的单个十进制数字说明了它们之间的区别。阅读了这个问题和答案之后，该版本的两个读者认为，最近他们通过增补不同表中的近似函数解决了一些问题。

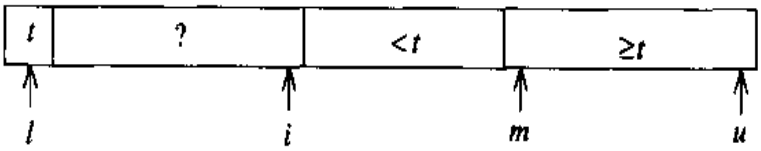
6. 原始文件需要 300KB 的磁盘容量。将两个数字压缩到一个字节中能够将容量减小到 150KB，但是增加了读文件所需的时间（在这个时代，“单面双密度”5.25 英寸软盘的容量为 184KB）。使用表查找替代昂贵的/和%操作，这将消耗 200 个字节的主存，但是将读取时间基本上减少到原来的成本。因此，200 字节的主存换取了 150KB 的磁盘容量。很多读者都建议我使用编码 c=(a<<4)|b；这个值能够被语句 a=c>>4 和 b=c&0xF 解码。John Linderman 观察到“移位和掩码不仅能够比乘和除快，而且 hex dump 这类常用工具能够使用可读形式显示解码的数据”。

第 11 章的答案

1. 通常会过度使用排序来查找 n 个浮点数中的最小值和最大值。答案 9 说明了如何

在不排序的情况下更加快速地找到中间值，但是在一些系统上，可能使用排序更为简单一些。排序非常适合于查找模式，但是散列的速度更快。查找平均值的代码的执行时间和 n 成正比，首先进行排序操作的方案可能具有更高的数值精确度；参见问题 14.4.b。

2. Bob Sedgewick 观察到能够使用下面的不变式修改 Lomuto 的划分方案，从而从右到左执行：



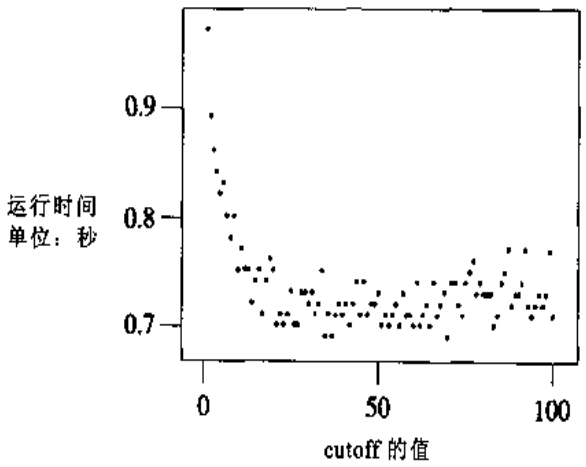
划分代码如下所示：

```
m = u+1
for (i = u; i >= l; i++)
    if x[i] >= t
        swap(--m, i)
```

因为终止时 x[m]=t，所以可以使用参数(l,m-1)和(m+1,u)进行递归，不需要其他 swap。Sedgewick 也使用 x[l]作为哨兵，从而删除内循环中的一个测试条件：

```
m = i = u+1
do
    while x[--i] < t
        ;
    swap(--m, i)
while i != l
```

3. 为了确定 cutoff 的最佳值，我将 cutoff 分别设置成 1～100 之内的所有值，将 n 设置成 1000000，并针对每个 cutoff 值分别运行该程序。下图给出了各个 cutoff 值对应的程序运行时间。



较好的 cutoff 值为 50，30 和 70 之间的值节省了几个百分点。

4. 参见第 11.6 节中引用的参考文献。

5. McIlroy 的程序的运行时间和排序的数据量成正比，这就优化了最差情况。它假设 x[0..n-1]中的每个记录有一个整数 length 和指向数组 bit[0..length-1]的指针。

```
void bsort(l, u, depth)
    if l >= u
        return
    for i = [l, u]
        if x[i].length < depth
            swap(i, l++)
    m = l
    for i = [l, u]
        if x[i].bit[depth] == 0
            swap(i, m++)
    bsort(l, m-1, depth+1)
    bsort(m, u, depth+1)
```

该函数原来是由 bsort(0,n-1,1)调用的。注意，该程序给参数和定义 for 循环的变量赋值。线性运行时间依赖于 swap 操作移动指向位字符串的指针而不是移动位字符串本身这一事实。

6. 该段代码实现选择排序：

```
void selsort()
    for i = [0, n-1)
        for j = [i, n)
            if x[j] < x[i]
                swap(i, j)
```

该段代码实现 Shell 排序：

```
void shellsort()
    for (h = 1; h < n; h = 3*h + 1)
        ;
    loop
        h /= 3
        if (h < 1)
            break
        for i = [h, n)
            for (j = i; j >= h; j -= h)
                if (x[j-h] < x[j])
                    break
                swap(j-h, j)
```

9. 该选择算法是由 C.A.R.Hoare 设计的；这段代码是在对 qsort4 稍作修改的基础上获得的。

```
void select1(l, u, k)
        pre l <= k <= u
        post x[l..k-1] <= x[k] <= x[k+1..u]
    if l >= u
```

```
        return
    swap(l, randint(l, u))
    t = x[l]; i = l; j = u+1
    loop
        do i++; while i <= u && x[i] < t
        do j--; while x[j] > t
        if i > j
            break
        temp = x[i]; x[i] = x[j]; x[j] = temp
    swap(l, j)
    if j < k
        select1(j+1, u, k)
    else if j > k
        select1(l, j-1, k)
```

由于函数在最后部分执行递归，因此可以将它转变为一个 while 循环。在*排序和查找*的问题 5.2.2～5.2.32 中，Knuth 说明程序平均通过 3.4n 次比较才能找到 n 个元素的中间值。概率参数类似于答案 2.A 中的最差情况下的参数。

14. 该版本的快速排序使用指向数组的指针。由于它仅仅使用两个参数 x 和 n，我发现只要读者理解概念 x+j+1 表示数组从位置 x[j+1]开始，它甚至比 qsort1 简单。

```
void qsort5(int x[], int n)
{   int i, j;
    if (n <= 1)
        return;
    for (i = 1, j = 0; i < n; i++)
        if (x[i] < x[0])
            swap(++j, i, x);
    swap(0, j, x);
    qsort5(x, j);
    qsort5(x+j+1, n-j-1);
}
```

第 12 章的答案

1. 这两个函数分别返回一个较大的随机数（通常为 30 位）和一个指定输入范围内的随机数：

```
int bigrand()
{   return RAND_MAX*rand() + rand(); }

int randint(int l, int u)
{   return l + bigrand() % (u-l+1); }
```

2. 如果想从范围 0...n-1 中选择 m 个整数，从范围内随机选择数值 i，然后输出数值 i，i+1，…，i+m-1，可能包含 0。该方法中的每个整数的选中概率为 m/n，但是偏向于特定的子集。

3. 如果到目前选择的整数数目少于 n/2，那么随机选中的整数不再被选中的概率大于 1/2。也就是说，根据按平均来说必须两次抛硬币才能获得人头面的逻辑，获得未选中的整数的平均数要少于 2。

4. 现在将集合 S 视为 n 个初始为空的瓮。每个调用 randint 都将选中一个瓮，我们将一个球抛入该瓮中。如果这个瓮中以前就有球了，那么 member 判断为真。统计学家需要知道球数，从而确认每个至少包含一个球的瓮，这就称为“Coupon Collector's Problem”（我必须收集多少篮球卡才能确保我拥有所有的 n？）答案是大致为 nlnn。当所有的球进入不同的瓮时，该算法需要执行 m 次测试；判断什么时候两个小球可能进入同一个瓮，这是“生日悖论”（在任何 23 个或更多的人群中，两个人有可能同一天生日）。总的来说，如果总共有 $O(\sqrt{n})$个球，那么两个球可能共享 n 个瓮中的一个。

7. 为了按增序输出值，可以将 print 语句放置在递归调用之后，或输出 n+1-i 而不是 i。

8. 为了按随机顺序输出各个整数，在第一次生成的时候就输出每个整数；参见答案 1.4。为了按序输出重复的整数，删除判断整数是否已经在集合中的测试条件。为了按随机顺序输出重复的整数，使用通用程序

```
for i = [0, m)
    print bigrand() % n
```

9. Bob Floyd 研究基于集合的算法时，他没有解决一个问题，那就是如何抛弃一些生成的随机整数。因此他派生了另一个基于集合的算法，下面是该算法的 C++实现代码：

```
void genfloyd(int m, int n)
{   set<int> S;
    set<int>::iterator i;
    for (int j = n-m; j < n; j++) {
        int t = bigrand() % (j+1);
        if (S.find(t) == S.end())
            S.insert(t); // t not in S
        else
            S.insert(j); // t in S
    }
    for (i = S.begin(); i != S.end(); ++i)
        cout << *i << "\n";
}
```

答案 13.1 使用不同的集合接口实现了同一个算法。Floyd 的算法最初出现在 1986 年 8 月出版的《*Communications of the ACM*》的 *Programming Pearls* 专栏中，在我 1988 年的《More *Programming Pearls*》的第 13 章再次出现。这些参考包含了对其正确性的简单证明。

10. 我们总是选择第一行，并使用二分之一的概率选择第二行，使用三分之一的概

率选择第三行，以此类推。在该过程结束的时候，每一行具有相同的选中概率（1/n，其中 n 是文件中的总行数）：

```
i = 0
while more input lines
    with probability 1.0/++i
        choice = this input line
print choice
```

11. 在“应用算法设计”一课的家庭作业中我提出了这个问题。如果学生想出的方法能够在几分钟的 CPU 时间内计算出答案，他们就得零分。如果回答是“我要同我的概率统计老师讨论”就能够得到一半的分数，而最佳答案是：

> 4..16 之间的数值对游戏没有任何影响，所以能够忽略它们。如果在 3 之前选择了 1 和 2 就能赢得游戏。这也就是 3 是最后选中的情况，这种情况的概率为三分之一。因此随机顺序赢的概率为 1/3。

不要被问题的陈述所迷惑。不能因为能够使用 CPU 时间所以你就使用 CPU 时间。

12. 第 5.9 节介绍了 Kernighan 和 Pike 的《*Practice of Programming*》。他们书中的第 6.8 节介绍了他们是怎样测试概率程序的（在第 15.3 节中我们将查看完成同一任务的不同程序）。

第 13 章的答案

1. 按此风格使用 IntSet 类能够实现答案 12.9 中的 Floy 算法：

```
void genfloyd(int m, int maxval)
{   int *v = new int[m];
    IntSetSTL S(m, maxval);
    for (int j = maxval-m; j < maxval; j++) {
        int t = bigrand() % (j+1);
        int oldsize = S.size();
        S.insert(t);
        if (S.size() == oldsize) // t already in S
            S.insert(j);
    }
    S.report(v);
    for (int i = 0; i < m; i++)
        cout << v[i] << "\n";
}
```

当 m 和 maxval 相等时，按递增顺序插入元素，这是二分查找树的最差情况。

4. 链表的这个迭代插入算法比相应的递归算法要长，因为它复制了在 head 之后和列表之后插入结点的案例分析：

```
void insert(t)
    if head->val == t
        return
    if head->val > t
        head = new node(t, head)
        n++
        return
    for (p = head; p->next->val < t; p = p->next)
        ;
    if p->next->val == t
        return
    p->next = new node(t, p->next)
    n++
```

这段代码更为简单，通过使用指向指针的指针删除了重复项：

```
void insert(t)
    for (p = &head; (*p)->val < t; p = &((*p)->next))
        ;
    if (*p)->val == t
        return
    *p = new node(t, *p)
    n++
```

它跟前一版本一样快。只需要对该段代码做出很小的修改就能应用在桶上。答案 7 将该方法应用于二分查找树中。

5. 需要一个指向下一可用结点的指针，才能用单个分配替换多个分配：

```
node *freenode;
```

我们在构建类的时候分配所有需要的结点：

```
freenode = new node[maxelms]
```

在插入函数中根据需要进行删除：

```
if (p == 0)
    p = freenode++
    p->val = t
    p->left = p->right = 0
    n++
else if ...
```

在桶中使用同一技巧。答案 7 在二分查找树中使用了这一技巧。

6. 按照递增顺序插入结点能够衡量数组和列表的查找成本，并且仅会引入很少的插入开销。该序列将引发箱和二分查找的最差性能。

7. 前一版本中为 null 的指针现在都将指向标记结点。在构造函数中将它初始化：

```
root = sentinel = new node
```

插入结点首先将目标值 t 放入标记结点，然后使用一个指针的指针（如答案 4 中所介绍的）遍历树直到找到 t。那时，它使用答案 5 中的技巧插入一个新结点。

```
void insert(t)
    sentinel->val = t
    p = &root
    while (*p)->val != t
        if t < (*p)->val
            p = &((*p)->left)
        else
            p = &((*p)->right)
    if *p == sentinel
        *p = freenode++
        (*p)->val = t
        (*p)->left = (*p)->right = sentinel
        n++
```

声明并初始化变量 node：

```
node **p = &root;
```

9. 使用位移替换除法，使用如下所示的伪码初始化变量：

```
goal = n/m
binshift = 1
for (i = 2; i < goal; i *= 2)
    binshift++
nbins = 1 + (n >> binshift)
```

这个插入函数从该结点开始：

```
p = &(bin[t >> binshift])
```

10. 可以通过混合并比较大量数据结构来表示随机集合。由于我们很清楚每个桶中将包含多少项，例如，我们可以使用第 13.2 节中的结果，使用小的数组来表示多数桶中的元素（然后当桶满时将多余的元素放入链表中）。在《*Communications of the* ACM》1986 年 5 月的“*Programming Pearls*”专栏中，Don Knuth 在介绍他用于文档化 Pascal 程序的 Web 系统时描述到“有序散列表”。在他 1992 年出版的《*Literate Programming*》书的第 5 章中再次提到了这篇文章。

第 14 章的答案

1. 如果将 swap 中对临时变量的赋值和从中取值的语句移到循环外部，那么就能够加快 siftdown 函数的执行速度。将代码移出循环，并在 x[0]位置放置一个哨兵元素来替代 if==1 的测试能够进一步加快 siftup 函数的执行速度。

2. 对书中的 siftdown 函数版本稍作修改就得到新的 siftdown 函数，使用 i=l 替换了赋值 i=1，并将跟 n 的比较替换成跟 u 的比较。结果，函数的运行时间为 O(log u-log l)。该段代码在 O(n)时间内建立了一个堆：

```
for (i = n-1; i >= 1; i--)
    /* 不变式: maxheap(i+1, n) */
    siftdown(i, n)
    /* maxheap(i, n) */
```

由于对于所有的整数 l>n/2 来说，maxheap(l, n)为真，因此能够将 for 循环中的边界 n-1 更改为 n/2。

3. 使用答案 1 和 2 中的函数，堆排序如下所示：

```
for (i = n/2; i >= 1; i--)
    siftdown1(i, n)
for (i = n; i >= 2; i--)
    swap(1, i)
    siftdown1(1, i-1)
```

它的运行时间仍然是 O(n log n)，但是比原来的堆排序的系数要小。网站（www.Programmingpearls.com）提供了堆排序的几个实现版本。

4. 在所有问题中，堆使用 O(log n)过程替换 O(n)过程。

a）构建哈夫曼代码的迭代过程中，选择集合中的两个最小的元素，然后将它们合并到一个新的结点中，这是通过两个 extractmins 加上一个 insert 来实现的。如果输入频率是有序的，那么就能够在线性时间内计算哈夫曼代码，具体的实现细节留作练习。

b）使用简单算法将浮点大数和浮点小数相加可能会造成精度误差。更高级的算法总是将集合中的两个最小的数值相加，这类似于前面提到的哈夫曼代码的算法。

c）具有百万个元素的堆（顶部是最小值）代表了目前看到的 100 万个最大的数值。

d）可以使用堆来表示每个文件中的下一个元素，从而合并有序文件。迭代步骤从堆中选择最小的元素并将它的后续元素插入到堆中。能够在 O(log n)时间内从 n 个文件中选择出要输出的下一个元素。

5. 在桶数列上有一个类似于堆的结构。堆中的每个结点说明了它的子孙中最不满的桶的剩余空间。判断在哪里放置新权时，查找会尽可能向左进行（也就是说，左边最不满的箱中有足够的空间容纳它），只有在必要的情况下才向右执行，该操作所需要的时间和堆的深度的 O(log n)成正比。插入了权之后，重新遍历该路径将权放置在堆中。

6. 磁盘上通常是通过让块 i 指向块 i+1 来实现顺序文件的。McCreight 观察到如果结点 i 也指向结点 2i，那么最多通过 O(log n)次访问就能够找到任意结点 n。下面的递归函数输出了访问的路径。

```
void path(n)
        pre   n >= 0
        post  path to n is printed
    if n == 0
        print "start at 0"
    else if even(n)
```

```
        path(n/2)
        print "double to ", n
    else
        path(n-1)
        print "increment to ", n
```

注意，这和问题 4.9 中的程序在 O(logn)步骤内计算 x^n 是相同的。

7. 修改后的二分查找从 i=1 开始，每次迭代都将 i 设置为 2i 或 2i+1。元素 x[1]包含中间元素，x[2]包含第一个四分位点，x[3]是第三个四分位点，以此类推。S.R.Mahaney 和 J.I.Munro 发现在 O(n)时间和 O(1)额外空间内将具有 n 元素的有序数组放入“堆查找”中的算法。作为该方法的先驱，还考虑了将大小为 2^k-1 的有序数组复制到一个“堆查找”数组 b 中：a 中奇数位的元素按序进入 b 中后半部分的位置，逢 2 的位置模 4 得到 b 的第二个四分之一位，等等。

11. C++标准模板库支持堆的 make_heap、push_heap、pop_heap 和 sort_heap 这类操作。您可以组合这些操作以尽量简化堆排序：

```
make_heap(a, a+n);
sort_heap(a, a+n);
```

STL 同样也提供了 priority_queue 适配器。

第 15 章的答案

1. 很多文档系统提供了检索出所有格式命令并查看输入的原始文本表示的方法。在长文本上运行第 15.2 节的字符串复制程序时发现，该程序对文本的格式非常敏感。程序需要 36 秒来处理 King James Bible 中的 4460056 个字符，并且最长的重复字符串中有 269 个字符。如果删除每行的行号从而标准化输入文本后，长字符串就能够跨越行界，这时最长的字符串能够达到 563 个字符，该程序找出这个字符串需要的运行时间和前面基本相同。

3. 由于该程序每次执行插入操作时都需要执行很多相关的查找操作，因此只有很少一部分时间是用于内存分配的。使用专用内存分配器能够将处理时间大致减少 0.06 秒，这能将这个阶段提速 10%，但是只能将整个程序提速 2%。

5. 在 C++程序中添加另一个 map，从而将一系列的单词和它们的计数结合起来。在 C 程序中，我们可以通过计数排序数组，然后迭代它（由于一些单词的统计很大，因此这个数组要比输入的文本小得多）。对于一般文本，我们可以使用关键字索引，保存一个范围在 1..1000 的链表数组以作计数用。

7. 算法书上警告过“aaaaaaaa”这类输入，即重复几千次。我发现很容易就能够计算出处理文件中新行的程序所需的运行时间。程序需要 2.09 秒来处理 5000 个新行，8.90

秒处理 10000 个新行，37.90 秒处理 20000 个新行。这个增长速度要比平方快一些，可能和 $n\log_2 n$ 成正比，每个都和 n 成比例。添加大型输入文件的两个副本可能会出现更差的情况。

8. 子数组 a[i..i+M]代表了 M+1 个字符串。由于数组是有序的，通过在第一个和最后一个字符串上调用 comlen，我们能够很快判断出 M+1 个字符串中有几个相同的字符：

```
comlen(a[i], a[i+M])
```

网站（www.Programmingpearls.com）提供了该算法的实现代码。

9. 将第一个字符串读入数组 c，注意其终止的位置，使用 null 字符结束它，然后读入第二个字符串并结束它。跟前面一样进行排序。在扫描数组时，使用“异或”确保其中一个字符串是在过渡点之前开始的。

14. 该函数散列了 k 个单词，以 null 字符结束：

```
unsigned int hash(char *)
    unsigned int h = 0
    int n
    for (n = k; n > 0; p++)
        h = MULT * h + *p
        if (*p == 0)
            n--
    return h % NHASH
```

网站（www.programmingpearls.com）提供的程序使用该散列函数替换 Markov 文本生成算法中的二分查找，这样就将时间从 O(n log n)缩减到 O(n)。程序使用散列表中的元素列表表示方法来添加正好 nwords 个额外的 32 位整数，其中 nwords 是输入的单词数。