PEARSON

Addison Wesley

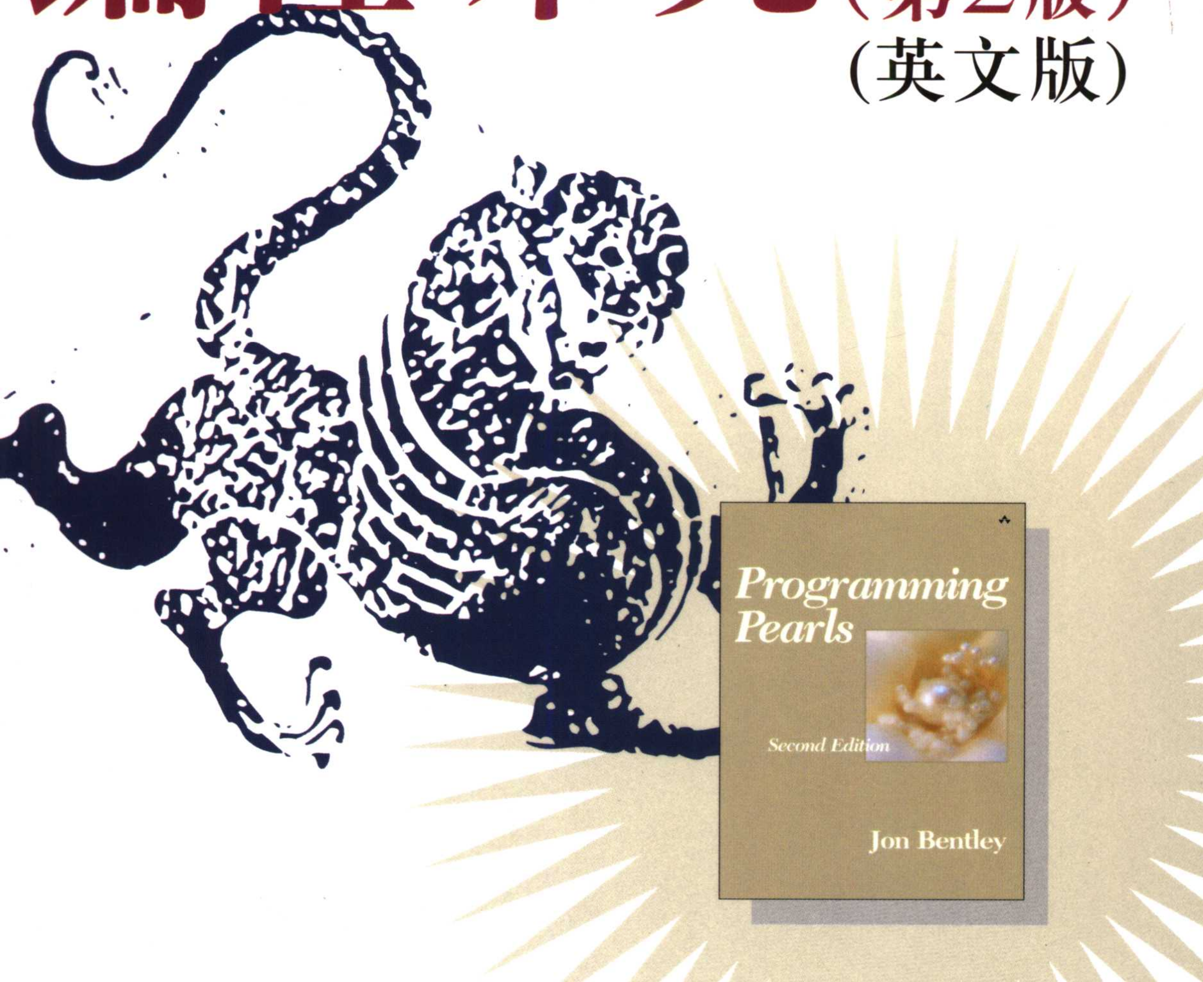
典藏原版书苑

Programming Pearls

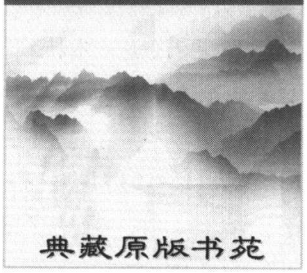
Second Edition

[美] Jon Bentley 著

编程珠玑（第2版）（英文版）



人民邮电出版社
POSTS & TELECOM PRESS



编 程 珠 玑

（第2版）（英文版）

Programming Pearls (Second Edition)

[美] Jon Bentley 著

图书在版编目（CIP）数据

编程珠玑 /（美）本特利（Bentley, J.）著．—2 版．
—北京：人民邮电出版社，2006.11
（典藏原版书苑）
ISBN 7-115-15171-7

Ⅰ．编… Ⅱ．本… Ⅲ．程序设计—英文 Ⅳ．TP311.1

中国版本图书馆 CIP 数据核字（2006）第 101389 号

典藏原版书苑

编程珠玑（第 2 版）（英文版）

◆ 著　　[美] Jon Bentley
　责任编辑　刘映欣
◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
　邮编　100061　　电子函件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京铭成印刷有限公司印刷
　新华书店总店北京发行所经销
◆ 开本：700×1000　1/16
　印张：15.5
　字数：280 千字　　　2006 年 11 月第 1 版
　印数：1 – 3 000 册　　　2006 年 11 月北京第 1 次印刷

著作权合同登记号　图字：01-2006-5089 号

ISBN 7-115-15171-7/TP・5644

定价：28.00 元

读者服务热线：(010)67132705　印装质量热线：(010)67129223

内容提要

本书针对程序设计人员探讨了一系列的实际问题，这些问题是对现实中常见问题的归纳总结。作者虽然没有给出解决这些问题的具体代码，但始终非常富于洞察力和创造力地围绕着这些折磨程序员的实际问题展开讨论，从而引导读者理解问题并学会解决问题的技能，这些都是程序员实际编程生涯中的基本技能。为此，本书给出了一些精心设计的有趣而且颇具指导意义的程序，这些程序能够为那些复杂的编程问题提供清晰而且完备的解决思路，书中还充满了对实用程序设计技巧及基本设计原则的清晰而睿智的描述。

本书在第一版的基础上增加了 3 个方面的新内容：测试、调试和计时，集合表示，字符串问题，并对第一版的所有程序都进行了改写，生成了等量的新代码。

前　言

计算机程序设计涉及很多方面。Fred Brooks 在《人月神话》中描述了一幅广阔的画面，他的作品着重介绍了在大型软件项目中关键角色的管理。更具体一点的是 Steve McConnell 在《代码大全》中介绍的良好的程序设计风格，书中所涉及的主题对于优秀的软件和程序员都至关重要。不幸的是，那些遵循合理软件工程原则的应用程序有时候令人生畏——除非软件按时交付并能正常运转。

关于本书

本书涉及的主题是计算机专业领域中更具魅力的一个方面：超越于纯粹工程学范畴、富于洞察力和创造力的编程珠玑。正如珍珠来自于曾经折磨牡蛎的沙粒，编程珠玑也来自曾经折磨程序员的实际问题。书中的程序不仅能够引发您的兴趣，而且可以教给您重要的程序设计技巧和基本的设计原则。

书中的内容多数来源于发表在 *Communications of the Association for Computing Machinery* 上的 Programming Pearls 专栏文章，经整理、修订后于 1986 年作为本书的第一版出版。本书在重新编辑第一版大部分内容的基础上，还加入了 3 个新的主题。

本书对读者的惟一要求是，需要有过使用高级语言编程的经验。一些高级的技术（例如 C++模版）在书中也会偶尔出现，不熟悉这些主题的读者可以直接跳过相关章节，不会造成阅读障碍。

尽管书中的每一章都可以自成体系，但在整体上仍然进行了逻辑上的分组。第 I 部分的第 1 章到第 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 II 部分内容主要围绕效率问题进行展开，效率问题本身不仅十分重要，而且常常是开始进行有趣的程序设计最好的出发点。第III部分将会把这些技巧应用到关于排序、查找和字符串处理的基本问题上。

建议读者不要读得太快，请仔细阅读，并尝试解决书中提到的问题，一些问题看似容易，但可能需要花费很多的精力去解决。接下来请认真解答每章后提出的问题，通过对解决方案的思考，从本书学到的大部分知识将得以巩固。如果愿意，可以在阅读书后给出的提示和解决方案之前与周围的朋友或同事进行讨论。每章后面列出的阅

读材料并不代表最权威的参考资料，我只是推荐了一些优秀的参考书目，这些书也是我极有价值的个人收藏。

本书是为程序员所著。希望这些问题、提示、解决方案和参考资料将会给程序员带来帮助。事实上，本书已经在高等院校的多门课程中被采用，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，同时也说明了如何将本书应用于算法和数据结构方面的课程。

关于代码

本书第一版中的伪码程序其实已全部实现，但只有我可以看到这些真正的代码。在这一版中，我重新编写了原来的程序并生成了等量的新代码。可以从本书的网站（www.programmingpearls.com）获得这些程序代码，其中包含了许多用于测试、调试和定时的架构平台。网站上也提供一些其他相关资料。因为现在可以从网上获得大量软件，所以这一版本的一个新话题就是介绍如何评估并使用软件组件。

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适用的，但这样有利于传达算法的关键思想。5.1 节的答案中介绍了关于这种风格的背景。

书中包含了一些现实中的 C 和 C++程序，但大多数函数都表示成伪码的形式：采用较少的空格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，左右圆括号表示开区间（区间中不包含边界值），而左右方括号则表示闭区间（区间中包含边界值）。表达式 function{ i ,j}表示调用一个带参数 i 和 j 的函数，array[i ,j]会访问一个数组元素。

本书列出了许多程序在“我的机器”上运行的情况，“我的机器”的配置是奔腾 II 400MHz 处理器、128MB 内存、Windows NT 4.0 操作系统。书中还记录了程序在其他几台机器上的运行情况，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可能程度的编译器优化。建议您在自己的机器上计时，我敢肯定您将会找出相似的运行规律。

致第一版读者

希望您阅读本书的第一反应是“看起来的确与第一版很像”，稍后，希望您将会有“我从未看过此书”的感觉。

这一版与第一版所关注的问题相同，但被置于更大的背景之中。计算机技术已经在一些重要领域发生了根本性的变化，例如数据库、网络、用户界面。尽管在这些领域中仍然存在着编程方面的难题，但大多数程序员都应该熟悉这些技术。第一版的程序在本书中仍然得以保留，与第一版相比，本书像是一个更大池塘中更大一条的鱼。

第一版第 4 章中关于实现二进制查找的一节在这一版本中被放到了第 5 章介绍测

试、调试和定时中。第一版第 11 章的内容经扩展后被拆成这一版本的第 12 章（关于原来的问题）和第 13 章（关于集合表示）。第一版的第 13 章介绍了一种运行在 64KB 地址空间上的拼写检查，在这一版本中被删掉了，但它的思想在 13.8 节中仍有保留。新的第 15 章是关于字符串问题的，并增减了一些新的小节。由于增加了新的问题、新的解决方案以及 4 个附录，这一版本比上一版本增加了 25%的篇幅。

许多旧的案例研究在这一版本中没有发生变化，但也有一些老的问题根据现在的技术发展状况进行了重新改写。

第一版致谢

在此感谢许多人对我的支持，是 Peter Denning 和 Stuart Lynn 提出了在 *Communications of the ACM* 上办专栏的想法。Peter 勤奋地为 ACM 工作，他使这个专栏被立项并让我从事此项工作。ACM 总部的成员，尤其是 Roz Steier 和 Nancy Adriance 给了我许多支持，他们让这些专栏文章发表时保持最初的风格。我尤其要感谢 ACM 的是，它支持专栏文章发表时保持最初的风格。同时感谢许多 CACM 的读者，是他们在专栏上的热心评论使这个新版本变得必要并有可能出版。

感谢 Al Aho、Peter Denning、Mike Garey、David Johnson、Brian Kernighan、John Linderman、Doug McIlroy 和 Don Stanat 在百忙中抽出宝贵时间仔细阅读了本书的每一个章节。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。Al Aho、Andrew Hume、Brian Kernighan、Ravi Sethi、Laura Skinger 和 Bjarne Stroustrup 在本书的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的倒数第二个草稿。在此感谢所有帮助过我的人。

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 仔细阅读了这一版本。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 和他的 Addison-Wesley 同事为本书的出版所提供的热情帮助。

CONTENTS

PART I: PRELIMINARIES

These five columns review the basics of programming. Column 1 is the history of a single problem. A combination of careful problem definition and straightforward programming techniques led to an elegant solution. The column illustrates the central theme of this book: thinking hard about a real case study can be fun and can also lead to practical benefits.

Column 2 examines three problems, with an emphasis on how algorithmic insights can yield simple and effective code. Column 3 surveys the crucial role that the structure of data can play in software design.

Column 4 introduces program verification as a tool for writing correct code. Verification techniques are used extensively as we derive subtle (and fast) functions in Columns 9, 11 and 14. Column 5 shows how we implement those abstract programs in real code: we use scaffolding to probe a function, to bombard it with test cases, and to measure its performance.

COLUMN 1: **CRACKING THE OYSTER**

The programmer's question was simple: ''How do I sort a disk file?'' Before I tell you how I made my first mistake, let me give you a chance to do better than I did. What would you have said?

1.1 A Friendly Conversation

My mistake was to answer his question. I gave him a thumbnail sketch of how to implement a Merge Sort on disk. My suggestion that he dig into an algorithms text met with less than enthusiasm — he was more concerned about solving the problem than furthering his education. I then told him about a disk sorting program in a popular programming book. The program consisted of about two hundred lines of code in a dozen functions; I estimated that implementing and testing the code would have taken the programmer at most a week.

I thought that I had solved his problem, but his hesitation led me back to the right track. The conversation then went something like this, with my questions in *italics*.

Why do you want to write your own sort at all? Why not use a sort provided by your system?

I need the sort in the middle of a large system, and for obscure technical reasons, I can't use the system file-sorting program.

What exactly are you sorting? How many records are in the file? What is the format of each record?

The file contains at most ten million records; each record is a seven-digit integer.

Wait a minute. If the file is that small, why bother going to disk at all? Why not just sort it in main memory?

Although the machine has many megabytes of main memory, this function is part of a big system. I expect that I'll have only about a megabyte free at that point.

Is there anything else you can tell me about the records?

Each one is a seven-digit positive integer with no other associated data, and no integer can appear more than once.

The context makes the problem clearer. In the United States, telephone numbers consist of a three-digit ''area code'' followed by seven additional digits. Telephone calls to numbers with the ''toll-free'' area code of 800 (the only such code at the time) were not charged. A real database of toll-free telephone numbers includes a great deal of information: the toll-free telephone number, the real number to which calls are routed (sometimes several numbers, with rules on which calls go where when), the name and address of the subscriber, and so on.

The programmer was building a small corner of a system for processing such a database, and the integers to be sorted were toll-free telephone numbers. The input file was a list of numbers (with all other information removed), and it was an error to include the same number twice. The desired output was a file of the numbers, sorted in increasing numeric order. The context also defines the performance requirements. During a long session with the system, the user requested a sorted file roughly once an hour and could do nothing until the sort was completed. The sort therefore couldn't take more than a few minutes, while ten seconds was a more desirable run time.

1.2 Precise Problem Statement

To the programmer these requirements added up to, ''How do I sort a disk file?'' Before we attack the problem, let's arrange what we know in a less biased and more useful form.

Input: A file containing at most n positive integers, each less than n, where $n = 10^7$. It is a fatal error if any integer occurs twice in the input. No other data is associated with the integer.

Output: A sorted list in increasing order of the input integers.

Constraints: At most (roughly) a megabyte of storage is available in main memory; ample disk storage is available. The run time can be at most several minutes; a run time of ten seconds need not be decreased.

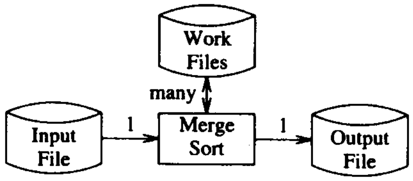
Think for a minute about this problem specification. How would you advise the programmer now?

1.3 Program Design

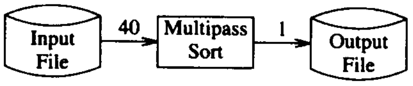
The obvious program uses a general disk-based Merge Sort as a starting point but trims it to exploit the fact that we are sorting integers. That reduces the two hundred lines of code by a few dozen lines, and also makes it run faster. It might still take a few days to get the code up and running.

A second solution makes even more use of the particular nature of this sorting problem. If we store each number in seven bytes, then we can store about 143,000 numbers in the available megabyte. If we represent each number as a 32-bit integer, though, then we can store 250,000 numbers in the megabyte. We will therefore use a program that makes 40 passes over the input file. On the first pass it reads into memory any integer between 0 and 249,999, sorts the (at most) 250,000 integers and writes them to the output file. The second pass sorts the integers from 250,000 to 499,999, and so on to the 40th pass, which sorts 9,750,000 to 9,999,999. A Quicksort would be quite efficient for the main-memory sorts, and it requires only twenty lines of code (as we'll see in Column 11). The entire program could therefore be implemented in a page or two of code. It also has the desirable property that we no longer have to worry about using intermediate disk files; unfortunately, for that benefit we pay the price of reading the entire input file 40 times.

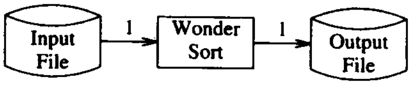
A Merge Sort program reads the file once from the input, sorts it with the aid of work files that are read and written many times, and then writes it once.



The 40-pass algorithm reads the input file many times and writes the output just once, using no intermediate files.



We would prefer the following scheme, which combines the advantages of the previous two. It reads the input just once, and uses no intermediate files.



We can do this only if we represent all the integers in the input file in the available megabyte of main memory. Thus the problem boils down to whether we can represent at most ten million distinct integers in *about* eight million available bits. Think about an appropriate representation.

1.4 Implementation Sketch

Viewed in this light, the *bitmap* or *bit vector* representation of a set screams out to be used. We can represent a toy set of nonnegative integers less than 20 by a string of 20 bits. For instance, we can store the set {1, 2, 3, 5, 8, 13} in this string:

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

The bits representing numbers in the set are 1, and all other bits are 0.

In the real problem, the seven decimal digits of each integer denote a number less than ten million. We'll represent the file by a string of ten million bits in which the i^{th} bit is on if and only if the integer i is in the file. (The programmer found two million spare bits; Problem 5 investigates what happens when a megabyte is a firm limit.) This representation uses three attributes of this problem not usually found in sorting problems: the input is from a relatively small range, it contains no duplicates, and no data is associated with each record beyond the single integer.

Given the bitmap data structure to represent the set of integers in the file, the program can be written in three natural phases. The first phase initializes the set to empty by turning off all bits. The second phase builds the set by reading each integer in the file and turning on the appropriate bit. The third phase produces the sorted output file by inspecting each bit and writing out the appropriate integer if the bit is one. If n is the number of bits in the vector (in this case 10,000,000), the program can be expressed in pseudocode as:

```
/* 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
```

(Recall from the preface that the notation *for* i = [0, n) iterates i from 0 to $n-1$.)

This sketch was sufficient for the programmer to solve his problem. Some of the implementation details he faced are described in Problems 2, 5 and 7.

1.5 Principles

The programmer told me about his problem in a phone call; it took us about fifteen minutes to get to the real problem and find the bitmap solution. It took him a couple of hours to implement the program in a few dozen lines of code, which was far superior to the hundreds of lines of code and the week of programming time that we had feared at the start of the phone call. And the program was lightning fast: while a Merge Sort on disk might have taken many minutes, this program took little more than the time to read the input and to write the output — about ten seconds. Solution 3 contains timing details on several programs for the task.

Those facts contain the first lesson from this case study: careful analysis of a small problem can sometimes yield tremendous practical benefits. In this case a few minutes of careful study led to an order of magnitude reduction in code length, programmer time and run time. General Chuck Yeager (the first person to fly faster than sound) praised an airplane's engine system with the words ''simple, few parts, easy to maintain, very strong''; this program shares those attributes. The program's specialized structure, however, would be hard to modify if certain dimensions of the specifications were changed. In addition to the advertising for clever programming, this case illustrates the following general principles.

The Right Problem. Defining the problem was about ninety percent of this battle

— I'm glad that the programmer didn't settle for the first program I described. Problems 10, 11 and 12 have elegant solutions once you pose the right problem; think hard about them before looking at the hints and solutions.

The Bitmap Data Structure. This data structure represents a dense set over a finite domain when each element occurs at most once and no other data is associated with the element. Even if these conditions aren't satisfied (when there are multiple elements or extra data, for instance), a key from a finite domain can be used as an index into a table with more complicated entries; see Problems 6 and 8.

Multiple-Pass Algorithms. These algorithms make several passes over their input data, accomplishing a little more each time. We saw a 40-pass algorithm in Section 1.3; Problem 5 encourages you to develop a two-pass algorithm.

A Time-Space Tradeoff and One That Isn't. Programming folklore and theory abound with time-space tradeoffs: by using more time, a program can run in less space. The two-pass algorithm in Solution 5, for instance, doubles a program's run time to halve its space. It has been my experience more frequently, though, that reducing a program's space requirements also reduces its run time.† The space-efficient structure of bitmaps dramatically reduced the run time of sorting. There were two reasons that the reduction in space led to a reduction in time: less data to process means less time to process it, and keeping data in main memory rather than on disk avoids the overhead of disk accesses. Of course, the mutual improvement was possible only because the original design was far from optimal.

A Simple Design. Antoine de Saint-Exupéry, the French writer and aircraft designer, said that, "A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away." More programmers should judge their work by this criterion. Simple programs are usually more reliable, secure, robust and efficient than their complex cousins, and easier to build and to maintain.

Stages of Program Design. This case illustrates the design process that is described in detail in Section 12.4.

1.6 Problems

Hints for and solutions to selected problems can be found in sections at the back of the book.

1. If memory were not scarce, how would you implement a sort in a language with libraries for representing and sorting sets?

† Tradeoffs are common to all engineering disciplines; automobile designers, for instance, might trade reduced mileage for faster acceleration by adding heavy components. Mutual improvements are preferred, though. A review of a small car I once drove observed that "the weight saving on the car's basic structure translates into further weight reductions in the various chassis components — and even the elimination of the need for some, such as power steering".

2. How would you implement bit vectors using bitwise logical operations (such as and, or and shift)?
3. Run-time efficiency was an important part of the design goal, and the resulting program was efficient enough. Implement the bitmap sort on your system and measure its run time; how does it compare to the system sort and to the sorts in Problem 1? Assume that *n* is 10,000,000, and that the input file contains 1,000,000 integers.
4. If you take Problem 3 seriously, you will face the problem of generating *k* integers less than *n* without duplicates. The simplest approach uses the first *k* positive integers. This extreme data set won't alter the run time of the bitmap method by much, but it might skew the run time of a system sort. How could you generate a file of *k* unique random integers between 0 and $n-1$ in random order? Strive for a short program that is also efficient.
5. The programmer said that he had about a megabyte of free storage, but the code we sketched uses 1.25 megabytes. He was able to scrounge the extra space without much trouble. If the megabyte had been a hard and fast boundary, what would you have recommended? What is the run time of your algorithm?
6. What would you recommend to the programmer if, instead of saying that each integer could appear at most once, he told you that each integer could appear at most ten times? How would your solution change as a function of the amount of available storage?
7. [R. Weil] The program as sketched has several flaws. The first is that it assumes that no integer appears twice in the input. What happens if one does show up more than once? How could the program be modified to call an error function in that case? What happens when an input integer is less than zero or greater than or equal to *n*? What if an input is not numeric? What should a program do under those circumstances? What other sanity checks could the program incorporate? Describe small data sets that test the program, including its proper handling of these and other ill-behaved cases.
8. When the programmer faced the problem, all toll-free phone numbers in the United States had the 800 area code. Toll-free codes now include 800, 877 and 888, and the list is growing. How would you sort all of the toll-free numbers using only a megabyte? How can you store a set of toll-free numbers to allow very rapid lookup to determine whether a given toll-free number is available or already taken?
9. One problem with trading more space to use less time is that initializing the space can itself take a great deal of time. Show how to circumvent this problem by designing a technique to initialize an entry of a vector to zero the first time it is accessed. Your scheme should use constant time for initialization and for each vector access, and use extra space proportional to the size of the vector. Because this method reduces initialization time by using even more space, it should be considered only when space is cheap, time is dear and the vector is sparse.