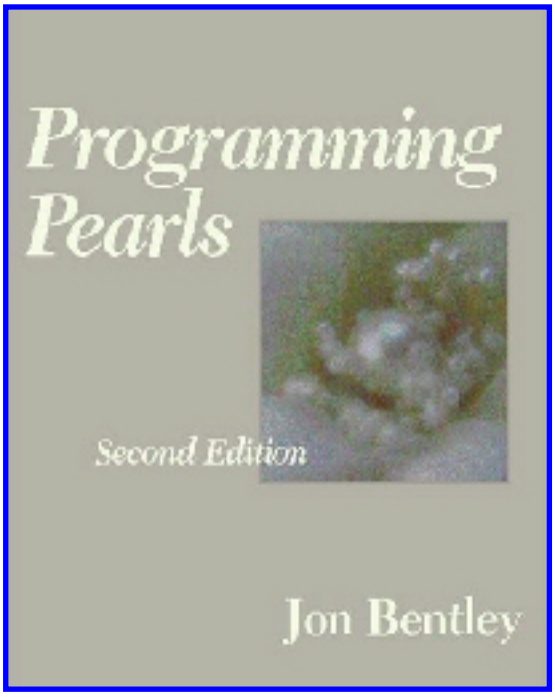
Cracking the Oyster (Column 1 of Programming Pearls)



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.

The Rest of the Column

These are the remaining sections in the column.

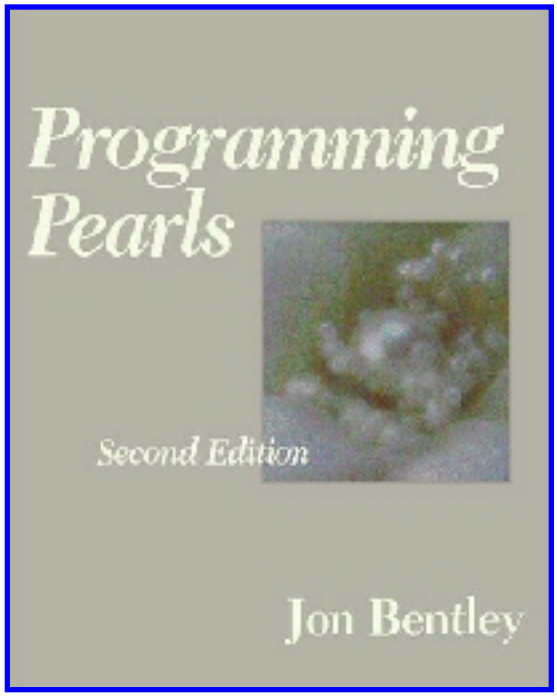
The Solutions to Column 1 give answers for some of the Problems.

Several of the programs in the column and the solutions can be found with the other source code for this book.

The web references describe web sites related to the topic.

An excerpt from this column appears in the November 1999 issue of *Dr. Dobb's Journal*.

Precise Problem Statement (Section 1.2 of Programming Pearls)



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.

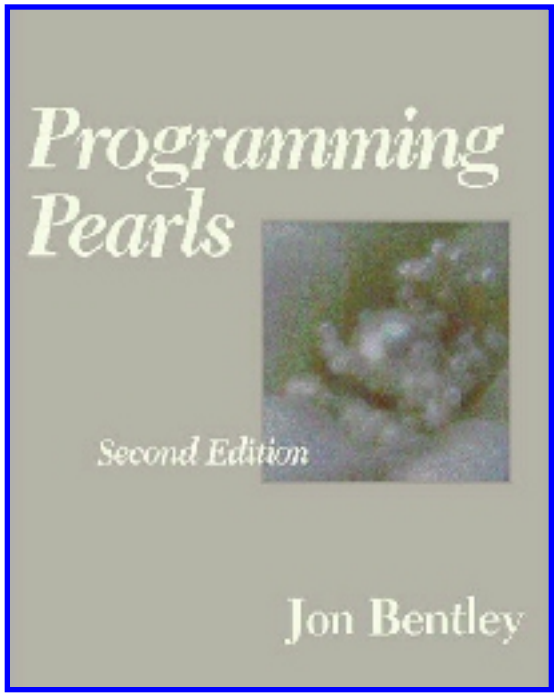
Output:
A sorted list in increasing order of the input integers.

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?

Next: Section 1.3. Program Design.

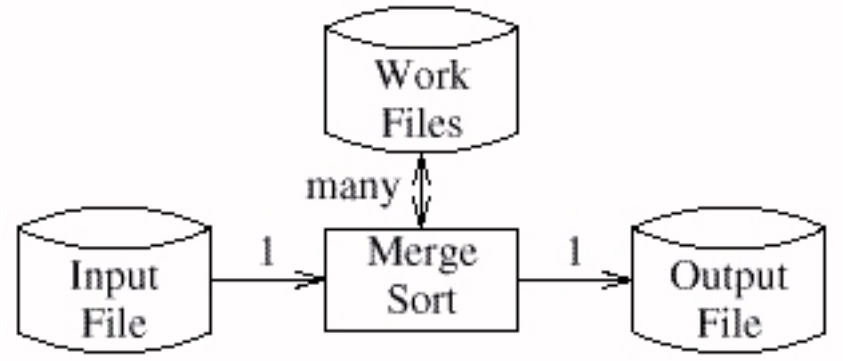
Program Design (Section 1.3 of Programming Pearls)



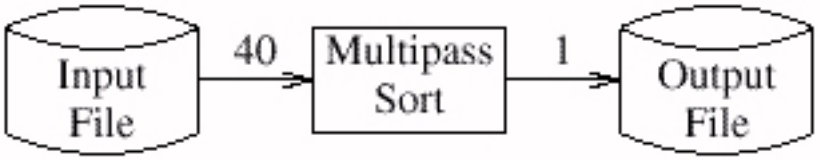
The obvious program uses a general disk-based Merge Sort as a starting point but trims it to exploit the fact that we are sorting integers. That reduces the two hundred lines of code by a few dozen lines, and also makes it run faster. It might still take a few days to get the code up and running.

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.

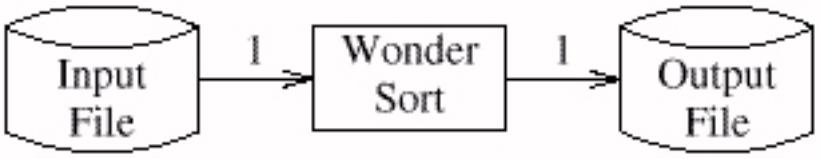
A Merge Sort program reads the file once from the input, sorts it with the aid of work files that are read and written many times, and then writes it once.



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



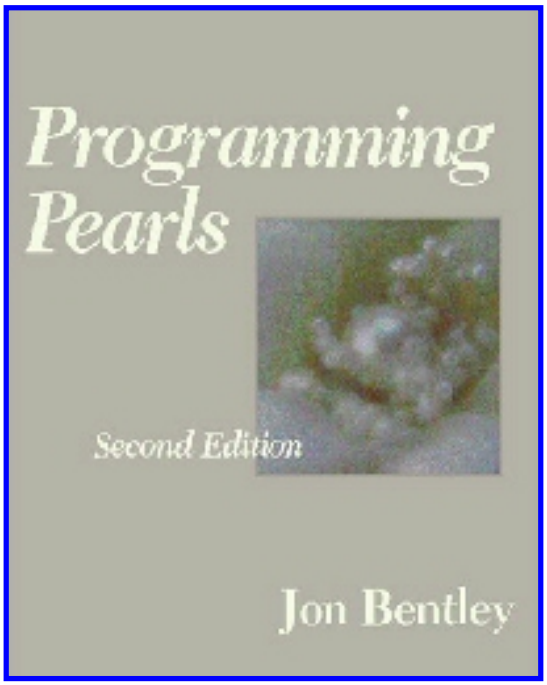
We would prefer the following scheme, which combines the advantages of the previous two. It reads the input just once, and uses no intermediate files.



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.

Next: Section 1.4. Implementation Sketch.

Implementation Sketch (Section 1.4 of Programming Pearls)



Viewed in this light, the *bitmap 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:

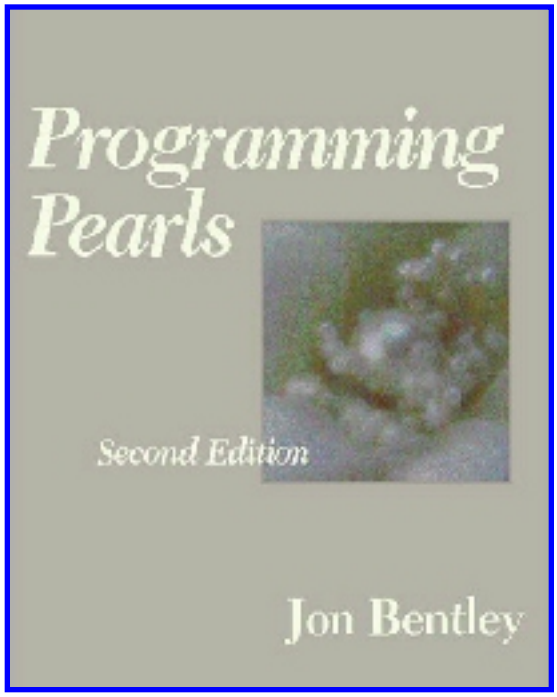
```
/* phase 1: initialize set to empty */
    for i = [0, n)
        bit[i] = 0
/* phase 2: insert present elements into the set */
    for each i in the input file
        bit[i] = 1
/* phase 3: write sorted output */
    for i = [0, n)
        if bit[i] == 1
            write i on the output file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

(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.

Next: Section 1.5. Principles.

Principles (Section 1.5 of Programming Pearls)



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. (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''.) 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

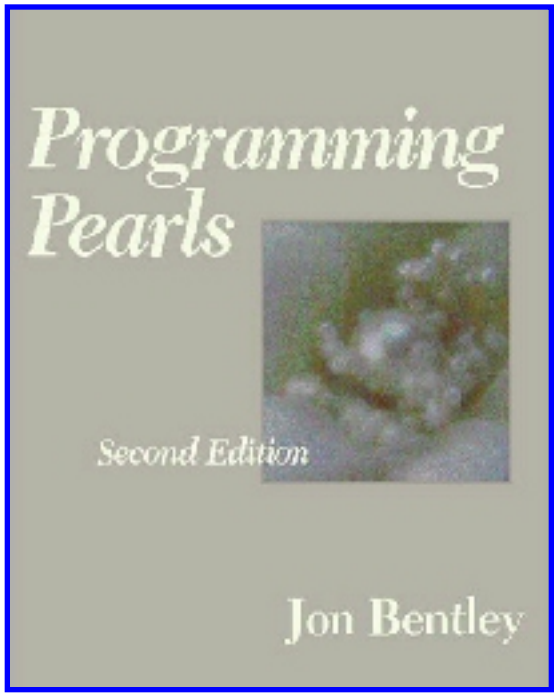
data to process means less time to process it, and keeping data in main memory rather than on disk avoids the overhead of disk accesses. Of course, the mutual improvement was possible only because the original design was far from optimal.

A Simple Design. Antoine de Saint-Exupery, 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.

Next: Section 1.6. Problems.

Problems (Section 1.6 of Programming Pearls)



The Solutions to Column 1 give answers for some of these problems.

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

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 *c* 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.

10. Before the days of low-cost overnight deliveries, a store allowed customers to order items over the telephone, which they picked up a few days later. The store's database used the customer's telephone number as the primary key for retrieval (customers know their phone numbers and the keys are close to unique). How would you organize the store's database to allow orders to be inserted and retrieved efficiently?

11. In the early 1980's Lockheed engineers transmitted daily a dozen drawings from a Computer Aided Design (CAD) system in their Sunnyvale, California, plant to a test station in Santa Cruz. Although the facilities were just 25 miles apart, an automobile courier service took over an hour (due to traffic jams and mountain roads) and cost a hundred dollars per day. Propose alternative data transmission schemes and estimate their cost.

12. Pioneers of human space flight soon realized the need for writing implements that work well in the extreme environment of space. A popular urban legend asserts that the United States National Aeronautics and Space Administration (NASA) solved the problem with a million dollars of research to develop a special pen. According to the legend, how did the Soviets solve the same problem?

Next: Section 1.7. Further Reading.