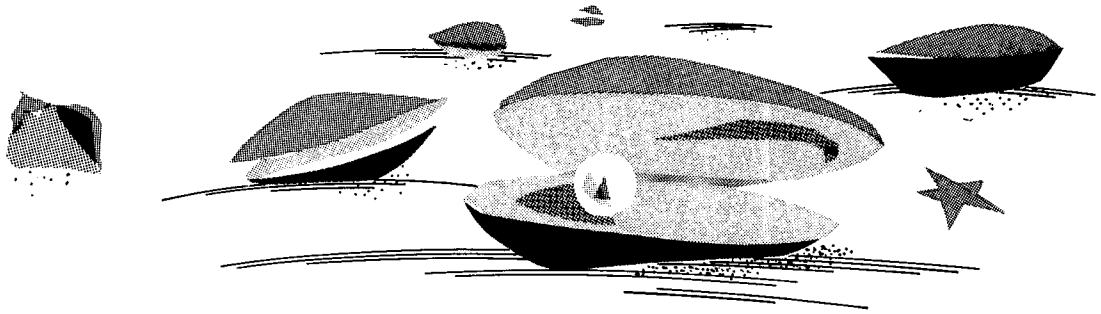
ANNOUNCING

programming pearls

By Jon Bentley



I am pleased to announce the inauguration of *Programming Pearls*, a new department of *Communications* devoted to the seemingly small things that distinguish great programs from other programs. *Programming Pearls* will be written by contributing editor Jon Bentley. It will appear in these pages every other month after an initial set of four columns monthly.

Jon Bentley is an Associate Professor of Computer Science at Carnegie-Mellon University, now on leave at Bell Laboratories in Murray Hill, New Jersey. Jon has a long interest in algorithms and data structures, especially those making up highly efficient programs. He believes that elegance means both simplicity in principle and efficiency in practice. He believes that every program can be elegant and that every programmer is capable of designing efficient elegant programs. He has written many papers on this subject. His book, *Writing Efficient Programs*, was published by Prentice-Hall (Englewood Cliffs, N.J.) in 1982.

Jon says: "Much of the art of writing code boils down to common sense and good taste. Unfortunately, the workmanlike application of the engineering principles isn't always exciting.

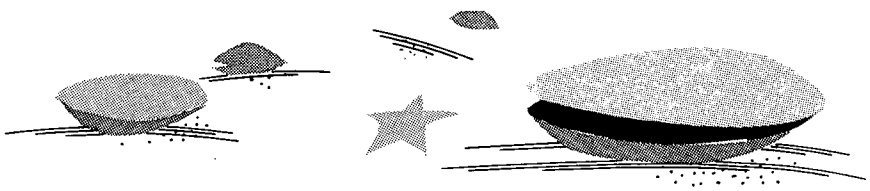
"My column will focus on a more glamorous aspect of the profession: *Programming Pearls* whose origins lie beyond solid engineering, in the realm of insight and craftsmanship. There are at least two reasons to study these programs: they're fun, and they teach techniques that belong in the tool box of every software engineer. Remember, though, that the columns describe only one small—but sometimes crucial—part of the programmer's task.

"I chose the metaphor of *pearls* very carefully. The pearl is a small hard object of great value to the users of the oyster in which it is found. It is not easily formed but was well worth the effort. It protects the oyster from destruction by a grain of dirty sand."

The topics of the column will vary widely, subject to the requirement that each pearl be beautiful and valuable. Beauty will be measured by conceptual simplicity and the soundness of underlying scientific principle. Value will be measured by the improvement to the quality and efficiency of the software. Readers are invited to send pearls they have discovered to Jon Bentley, Bell Laboratories, Room 2C-317, 600 Mountain Avenue, Murray Hill, NJ 07974. The best pearls will be written up in Jon's columns.

At the end of each column, Jon will include a few problems for you to think about as you seek deeper understanding of a pearl. He will publish solutions in the next column. Please do not send your solutions to Jon: he already knows them and does not plan to publish readers' solutions. Nonetheless, I urge you to work hard on devising solutions. Discuss them with colleagues. And try to use them in your own programming.

Peter J. Denning
Editor-in-Chief



CRACKING THE OYSTER

•●•

The programmer's question was simple: "How do I sort on disk?" Before I tell you how I made my first mistake, let me give you a chance to make yours: what would you have said?

My mistake was to answer his question; I gave him a thumbnail sketch on how to sort on disk. My suggestion that he dig into Knuth's classic *Sorting and Searching* met with less than enthusiasm; he was more concerned about solving the problem at hand than with furthering his education. I then told him about the disk sorting program in Chapter 4 of Kernighan and Plauger's *Software Tools*. Their Ratfor program consists of over two hundred lines of code in twelve procedures; translating that into Fortran and testing the resulting several hundred lines of code would have taken about a week. I thought that I had solved his problem; it was only his hesitation that 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 a sort routine at all? Why not use the system sort?
I need the sort in the middle of a large system, and the operating system doesn't provide a way to escape from a user program to a system routine.

What exactly are you sorting? How many records are in the file? What is the format of each record?
The file consists of at most 27,000 records; each record is a 16-bit integer.

Wait a minute; if the file is that small, why bother going to disk at all? Why not just sort the whole thing in main memory?
Although the machine has half a megabyte of main memory, this routine is in the middle of a big program. I expect that I'll have only about a thousand 16-bit words free at that point.

Is there anything else you can tell me about the records?
Each one is an integer between 1 and 27,000 (inclusive), and no integer can appear more than once.

The context makes the problem clearer: this system was used for political redistricting (automated gerrymandering), and the numbers being sorted were indices of precincts that comprise a political district. Each precinct within a state was given a unique number from 1 to 27,000 (the number of precincts in the largest state), and it was illegal to include the same precinct twice in one district. The context also defines the performance requirements: since the user interrupted his design session roughly once an hour to invoke the sort and could do nothing until it was completed, the sort couldn't take more than a few minutes, while a few seconds was a more desirable run time.

Precise Problem Statement

Those are the facts. In the mind of the programmer they added up to "How do I sort on disk?" Before we attack the problem, let's arrange those facts in a more accessible form.

Input:
A file containing at most 27,000 integers in the range 1 . . . 27,000; it is a fatal error condition 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) one thousand 16-bit words of storage are available in main memory; disk buffers in main memory and ample disk storage are available. The run time can be at most several minutes; a run time of ten seconds need not be decreased.

Before we try solving this problem together, think about it. How would you advise the programmer now?

Program Design

The most obvious solution is to use Kernighan and Plauger's general disk sorting program as a base but trim it to exploit the fact that we are sorting integers (their program sorts text lines). That wouldn't reduce the amount of code in their program substantially (maybe by about 30 lines out of over 200), but it would make it run faster. It would still take quite a while to get the code up and running.

A second solution makes even more use of the particular nature of this sorting problem; its main loop makes 27 passes over the input file. On the first pass it reads into memory any integer between 1 and 1000, sorts the (at most) 1000 integers and writes them to the output file. The second pass deals with the integers from 1001 to 2000, and so on to the twenty-seventh pass, which takes care of 26,001 to 27,000. Kernighan and Plauger's quicksort program would be quite efficient for the in-core sorts, and it requires only about 40 lines of Ratfor code. The entire program could therefore be implemented in Fortran in about 80 lines 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 many times.

Figure 1 illustrates the two solutions we have seen so far and suggests a third scheme that combines their positive attributes: we read the entire input file into main memory (once), manipulate the elements in main memory, and then write the output file. We can do this only if we can somehow represent all the integers in the input file in the roughly 1000 available words of main memory. Thus the problem boils down to whether we can represent the 27,000 distinct integers in *about* 16,000 available bits. Think for a minute about an appropriate representation.

Viewed in this light, the *bitmap* representation of a set screams out to be used. We will represent the file of integers by a string of 27,000 bits in which the *I*th bits is on if and only if the integer *I* is in the file. This representation makes crucial use of three attributes of this problem that are not usually found in sorting problems: the input is from a small range, it contains no duplicates, and no data is associated with each record beyond the integer.

Given the bitmap data structure to represent the set of integers in the file, the program is easy to write as three 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. Finally, the third phase produces the sorted output file by inspecting each bit and writing out the appropriate integer if the bit is one. Expressed in high-level pseudocode, the program becomes the following (where

the constant N represents the number of bits in the vector, in this case 27,000):

```
/* Phase 1: Initialize set to
            empty */
for I: = 1 to N do
    Bit[I] := 0
/* Phase 2: Insert present ele-
            ments into the set */
for each integer I in the file do
    Bit[I] := 1
/* Phase 3: Output sorted files */
for I: = 1 to N do
    if Bit [I] = 1 then
      write I on the output file
```

The only barrier standing between the above code sketch and a complete Fortran program is the implementation of bitmaps. While this would be trivial in the many programming languages that support bitmaps as a primitive data type, we must usually implement them ourselves in Fortran. Suppose that Fortran did provide bitwise logical operations on words (such as ANDing or ORing two words to give a third); how would you advise the programmer to implement bitmaps? What if those operations were not available; how would you implement bitmaps in that case?

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 routine if this happened? What happens when an input integer is less than one or greater than N? How should that case be handled? The third flaw is potentially more serious. The programmer said that he had about 1000 words of free storage, but this code uses 27,000/16 or 1688 words. When I described this code to the programmer he was able to scrounge the extra space without much trouble. What course of action would you recommend if the 1000 words of space had in fact been a hard and fast boundary? (See Problem 1 for a hint.)

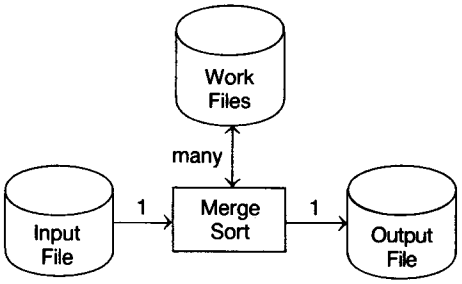
Lessons from the Case Study

The programmer told me about this problem in a phone call; it took us 15 minutes to get to the real problem and find the bitmap solution. After that he implemented the program in a few dozen lines of Fortran in a couple of hours. That compares quite nicely with the hundreds of lines of Fortran and the week of programming time we had feared at the start of the phone call. And the program was lightning fast: while I had estimated that a merge sort on disk might take several minutes to sort the data, this program takes little more than the time to read the input and to write the output—usually less than a dozen seconds.

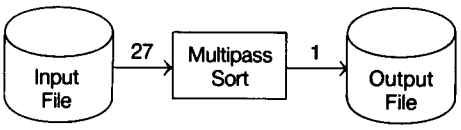
Those facts contain the first lesson from this case study: careful analysis of a small problem can sometimes yield tremendous practical benefits. In this case a few minutes of careful study led to an order of magnitude reduction in code length, programmer time, and run time. Furthermore, the fact that the code was short and clean meant that it was easy to maintain and bug-free. This case illustrates a number of general principles:

- *The right problem.* Defining the problem was 90 percent of this battle. I'm glad the programmer didn't settle for the first program I described.
- *The bitmap data structure.* This data structure is a useful representation for dense sets of elements over a finite domain when each element occurs at most once and there is no other data associated with the element. If those conditions aren't satisfied (when there are multiple elements or extra data) then as long as the key is from a finite domain, you might still be able to use it as an index into a table with more complicated entries.
- *Multiple-pass algorithms.* These algorithms make several passes over their input data, accomplishing a little bit more each time. We saw a 27-pass algorithm earlier; Problem 1 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. Problem 1 describes such a tradeoff: by using more time, a two-pass algorithm makes do with less space. More frequently, though, by reducing the space used by a program, a programmer reduces its run time as well. This certainly happened when we used bitmaps to represent the set we were sorting: that space-efficient structure 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 meant less time to process it, and keeping data in main memory rather than on disk avoided the extremely high overhead of disk accesses. (Of course, the mutual improvement was strongly related to the original design being far from optimal).
- *A sparse design.* Antoine de Saint-Exupéry, a French writer and aircraft designer, said that "a designer knows he

The 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 27-pass algorithm reads the input file many times and writes the output just once, but without using any intermediate files:



We would prefer the following scheme, which combines the advantages of the previous two: it reads the input just once, and sorts without intermediate disk files:

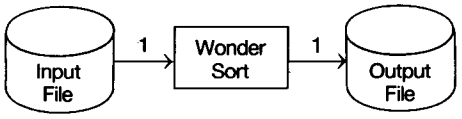


FIGURE 1. Three Designs for the Sorting Program.

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.

Problems

Problems 1 through 4 describe extensions to the program we studied above. Their purpose is to let you test your understanding of that program and try out some of the techniques on your own. Solutions to all the problems will be sketched in the next column.

1. The bitmap sorting algorithm uses 27,000/16 or 1688 sixteen-bit words of main memory. Show how a two-pass algorithm can reduce the space required to less than 1000 words. What is the run time of the new algorithm?

2. Run-time efficiency was an important part of our design goal, and the resulting program was efficient enough. Implement the bitmap program on your system and measure its run time. How does it compare to the system sort on the same file?

3. If you take Problem 2 seriously, then you will face the problem of generating K integers between 1 and 27,000 without duplicates. The simplest approach is to use the first K positive integers. This extreme data set won't alter the run time of the bitmap method by much, but it might lead to a system sort that is much faster than on typical data. How could you generate a file K unique random integers between 1 and N in random order? Strive for a short program that is also efficient.

4. 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 10 times? How would your solution to this variant change as a function of the amount of storage he told you was available?

The next problems are more related to the techniques we've considered than to the sorting problem per se. As you consider programs to solve these problems, keep in mind the techniques of multiple-pass algorthims, mutual time and space reductions (or sometimes tradeoffs), and key indexing (a generalization of bitmaps: instead of just storing a single bit to say whether the record is present, we store other data in an array indexed by the key).

5. In building a part of a college registration system, a programmer wanted a data structure to represent the number of seats available in the various courses. Each course had a unique four-digit identification number (from 0000 to 9999) and a three-digit seat count (from 000 to 999); there were 6000 total courses. After building the data structure from a file containing a list of the course numbers and the seat counts, the program was to process a tape of about 80,000 requests for courses. Each request for a valid course number was either denied (if the seat count was zero) or approved (in which case the seat count was decremented by one); requests for invalid course numbers were marked as such. The system had about 30 kilobytes of main memory available to the user (after allocating program space buffers, etc.). In his first design the programmer considered using a disk-based retrieval package in which each course was represented as a seven-byte disk record (four for course number and three for seat count); the disk operations required to access each record would have made processing the request tape quite expensive. Is there a better way to organize the course information?

6. 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 this problem can be circumvented 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. You may 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 very cheap and time is very dear. (This problem is from Exercise 2.12 of *The Design and Analysis of Computer Algorithms* by Aho, Hopcroft, and Ullman.)

The next problems are practice. We'll study several solutions for each of them in the next column.

7. Given a tape that contains exactly one million twenty-bit integers in random order, find a 20-bit integer that isn't on the tape (and there must be at least one missing—why?). How would you solve this problem with ample quantities of main memory? How would you solve it if you had several tape drives (say, half a dozen, just to make sure you don't run out) but only a few dozen words of main memory?

8. Rotate a vector of N elements left by I positions. For instance, with $N = 8$ and $I = 3$, the vector *ABCDEFGH is rotated to be DEFGHABC*. It is easy to do this rotation in N steps if we have available an N-element intermediate vector. Can you rotate the vector in time proportional to N using only a few extra words of storage?

9. Given a dictionary of English words, find all sets of anagrams. For instance, "pots", "stop" and "tops" are all anagrams of one another because each can be formed by permuting the letters of the others. Assume that each word is given on a separate input line in lower case letters.

Further Reading

The theme of this column is small (but difficult and rewarding) problems that arise in the construction of software. *Software Tools* by Kernighan and Plauger is an excellent reference in which to pursue this topic further (originally published in 1976 by Addison-Wesley, a later version with the same basic theme but many important changes appeared in 1981 as *Software Tools in Pascal*). That book describes a tool-building approach to software engineering that can change the way you think about programming. Beyond the basic philosophy, the book contains many fine examples of programming the way it should be: the straightforward construction of programs that are easy to use and to maintain.

In many places, though, the authors rise above mere good practice to subtle solutions to hard problems. The index entries of "algorithms" and "data structures" point to many of these pearls. My only complaint is that the authors sometimes present a subtle idea in such a straightforward and obvious way that the reader is misled into believing that it would naturally pop into his head in a similar situation. Be sure to take time to appreciate the book's pearls: they're beautiful, and they are based on techniques that belong in every programmer's tool box.

The main problem the programmer had in the case study described in this column was not so much technical as psychological: he couldn't make progress because he was trying to solve the wrong problem. We finally solved his problem by breaking through his conceptual block and solving an easier problem. *Conceptual Blockbusting* by Adams (Second Edition published by Norton in 1979) studies this kind of leap, and is generally a pleasant prod towards more creative thinking. Although not written with programmers in mind, many of its lessons are appropriate for programming problems.