programming pearls

by Jon Bentley

A LITTLE PROGRAM, A LOT OF FUN

Small computer programs are often educational and entertaining. This column tells the story of a tiny program that, in addition to those qualities, proved quite useful to a small company.

The Problem

The company had just purchased several personal computers. After I got their primary system up and running, I encouraged people to keep an eye out for tasks around the office that could be done by a program. The firm's business was public opinion polling, and an alert employee suggested automating the task of drawing a random sample from a (printed) list of precincts. Because doing the job by hand required a boring hour with a table of random numbers, she proposed the following program:

> I'd like a program to which the user types a list of precinct names and an integer M. Its output is a list of M of the precincts chosen at random. There are usually a few hundred precinct names (each of which is an alphanumeric string of at most a dozen characters), and M is typically between 20 and 40.

That's the user's idea for a program. Do you have any suggestions about the problem definition before we dive into coding?

My primary response was that it was a great idea; the task was ripe for automation. I then pointed out that typing several hundred names, while perhaps easier than dealing with long columns of random numbers, was still a tedious and error-prone task. In general, it's foolish to prepare a lot of input when the program is going to ignore the bulk of it anyway. I therefore suggested an alternative program:

> The input consists of two integers M and N, with $M < N$. The output is a sorted list of M random integers in the range $1..N$ in which no integer occurs more than once. For probability buffs, we desire a sorted selection without replacement in which each selection occurs equiprobably.

When $M = 20$ and $N = 200$, the program might produce a 20-element sequence that starts 4, 15, 17, . . . The user then draws a sample of size 20 from 200 precincts by counting through the list and marking the 4^{th}, 15^{th}, and 17^{th} names, and so on. (The numbers are required to be sorted because the hard copy list isn't numbered.)

That specification met with the approval of its potential users. After the program was implemented, the task that previously required an hour could be accomplished in a few minutes.

Now look at the problem from the other side: how would you implement the program? Assume that your system provides a function *RandInt*(I, J) that returns a random integer chosen uniformly in the range $I..J$, and a function *RandReal*(A, B) that returns a random real number chosen uniformly in the interval $[A, B)$.

One Solution

As soon as we settled on the problem to be solved, I ran to my nearest copy of Knuth's *Seminumerical Algorithms* (having copies of Knuth's three volumes both at home and at work has been well worth the investment). Because I had studied the book carefully a decade earlier, I knew that it contains several algorithms for problems like this. After spending a minute considering several possible designs that we'll study shortly, I realized that Algorithm S in Knuth's Section 3.4.2 was the ideal solution to this problem.

The algorithm considers the integers 1, 2, . . . , N in order, and selects each one by an appropriate random test. By visiting the integers in order, we guarantee that the output will be sorted.

To understand the selection criterion, let's consider the example that $M = 2$ and $N = 5$. We should select the integer 1 with probability 2/5; a program implements that by a statement like

```
if RandReal(0,1)<2/5 then . . .
```

Unfortunately, we can't select 2 with the same probability: doing so might or might not give us a total of 2 out of the 5 integers. We will therefore bias the decision and select 2 with probability 1/4 if 1 was chosen but with probability 2/4 if 1 was not chosen. In general, to select S numbers out of R remaining, we'll select the next number with probability S/R.

This probabilistic idea results in Program 1.

```
Select := M; Remaining := N
for I := 1 to N do
    if RandReal(0,1)<Select/Remaining
            then
        print I; Select := Select-1
    Remaining := Remaining-1
```

As long as $M \leq N$, the program selects exactly M integers: it can't select more because when *Select* goes to zero no integer is selected and it can't select fewer because when *Select/Remaining* goes to one an integer is always selected. The `for` statement ensures that the integers are printed in sorted order. The above description should help you believe that each subset is equally likely to be picked; Knuth gives a probabilistic proof.

Knuth's second volume made the program easy to write. Even including titles, range checking, and the like, the final program required only 13 lines of BASIC. It was finished within half an hour of when the problem was defined, and has been used for several years without problems.

The Design Space

I just described one part of a programmer's job: solving today's problem. Another, and perhaps more important, part of the job is to prepare for solving tomorrow's problems. Sometimes that preparation involves taking classes or studying books like Knuth's. More often, though, we programmers learn by the simple mental exercise of asking how we might have solved a problem differently. Let's do that now by exploring the space of possible designs for a program to select M integers at random from 1..N.

We'll start by evaluating Program 1. The algorithmic idea is straightforward, the code is short, it uses just a few words of space, and the run time is fine for this application. The run time might, however, be a problem in other applications: to select a single integer from the range $1..2^{31} - 1$, for instance, would take hours on a supercomputer. It's therefore worth a few minutes of our time to study other ways of solving the problem. Sketch as many high-level designs as you can before reading on; don't worry about implementation details yet.

One solution inserts random integers into an initially empty set until there are enough. In pseudocode, it is

```
Initialize set S to empty
Size := 0
while Size<M do
    T := RandInt(1,N)
    if T is not in S then
        Insert T in S
        Size := Size + 1
Print the elements of S in sorted
    order
```

The algorithm is not biased towards any particular element; its output is random. We are still left with the problem of implementing the set S; think about an appropriate data structure.

The bitmap data structure is particularly easy to implement. We represent the set S by an array of bits in which the I^{th} bit is one if and only if the integer I is in the set. We initialize it by the subroutine *InitToEmpty*:

```
for I := 1 to N do
    Bit[I] := 0
```

The function *Member*(T) tells whether T is in S by returning *Bit*[T], and the procedure *Insert*(T) inserts T in S by the assignment *Bit*[T]: = 1. Finally, *PrintInOrder* prints the elements of S:

```
for I := 1 to N do
    if Bit[I]=1 then
        print I
```

These subroutines allow us to write more precise pseudocode for Program 2.

```
InitToEmpty
Size := 0
while Size<M do
    T := RandInt(1,N)
    if not Member(T) then
        Insert(T)
        Size := Size + 1
PrintInOrder
```

The bitmaps in Program 2 use N/b words of b-bit memory. The obvious implementations of the initialization and printing routines both require time proportional to N, but that can be reduced to N/b by operating on the b bits in a word simultaneously (this holds as long as $M < N/b$; we'll soon consider what to do when M is close to N). There are always exactly M calls to the *Insert* procedure, but there may be more calls to *Member* because some of *RandInt*'s random numbers may already be in the set. Problem 2 shows that as long as $M < N/2$, the expected number of *Member* tests is less than $2M$. Both *Member* and *Insert* require constant time per operation, so the total cost of these operations is proportional to M. Thus the expected total run time of Program 2 is $O(N/b)$.

Although the performance analysis assumed that the set was implemented by a bitmap, nothing in Program 2 says so: the *InitToEmpty*, *Member*, *Insert*, and *PrintInOrder* operations all refer to the "Abstract Data Type" of sets. Replacing those four subroutines can change the representation of the sets and thereby change the performance of the program. Figure 1 illustrates several possible data structures at the end of a run in which $M = 5$, $N = 10$, and *RandInt*(1, 10) returns the sequence 3, 1, 4, 1, 5, 9.

Binary search trees are described in most texts on algorithms and data structures. Because the insertions into the tree are in random order, it is unlikely to get too far out of balance; complex balancing schemes are therefore not needed in this application. The M bins can be viewed as a kind of hashing in which the integers in the range 1..N/M are placed in the first bin, and the integer I is placed in bin (roughly) $I \times M/N$. The bins are implemented as an array of linked lists, each of expected length one. The average performance of the various schemes, when $M < N/b$, is given in Table 1.

Beware of the constant factors hiding in the big-ohs: the array operations are usually cheap compared to some implementations of the bit vector accesses, the pointer operations on binary trees, and the divisions

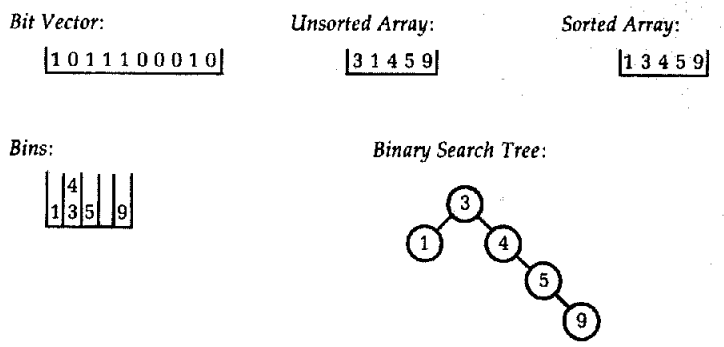

FIGURE 1. Data Structures for Representing Sets

used by bins. To understand the performance issues, let's consider the case that $N = 1{,}000{,}000$ and $b = 32$. When $M = 5{,}000$, bins are probably the most efficient structure; when $M = 50{,}000$, bitmaps are faster and take less space; when $M = 500{,}000$, Program 1 uses much less space and is also faster. When $M = 999{,}995$, though, we would do better to represent the five elements *not* selected; either kind of array would be easy to code and fast for this task.

Yet another approach to generating a sorted subset of random integers is to shuffle an N-element array that contains the numbers $1..N$, and then sort the first M to be the output. Knuth's Algorithm P in Section 3.4.2 shuffles the array $X[1..N]$:

```
for I := 1 to N do
    swap(X[I],X[RandInt(I,N)])
```

Ashley Shepherd and Alex Woronow of the University of Houston observed that in this problem we need only shuffle the first M elements of the array. The complete program is thus Program 3:

```
for I := 1 to N do X[I]:=I
for I := 1 to M do
    swap(X[I],X[RandInt(I,N)])
sort(1,M)
```

The sorted list is in $X[1..M]$. The algorithm uses N words of memory and $O(N + M \log M)$ time. We can view this algorithm as an alternative to Program 2 in which we represent the set of selected elements in $X[1..I]$ and the set of unselected elements in $X[I + 1..N]$. By explicitly representing the unselected elements we don't have to test whether the newly selected element is already chosen.

Although Programs 1, 2, and 3 offer many different solutions to the problem, they by no means cover the possible design space. One approach generates the "gaps" between successive integers in the set. Although I don't know how to make it work, the method might be used to generate M random integers in $O(M)$ time and constant space. Problem 8 describes an approach in which a multiple-pass algorithm yields a time-space trade-off.

Principles

This column illustrates several important steps in the programming process. Although the following discussion presents the stages in a natural order, the design process is more active: we hop from one activity to another, visiting each many times before arriving at an acceptable solution.

Understand the Perceived Problem. Talk with the user about the context in which the problem arises. Problem statements often include ideas about solutions; like all early ideas, they should be considered but not followed slavishly.

Specify an Abstract Problem. A clean, crisp problem statement helps us first to solve this problem and then to see how this solution applies to other problems.

Explore the Design Space. Too many programmers jump too quickly to "the" solution to their problem; they think for a minute and code for a day rather than thinking for an hour and coding for an hour. Using informal high-level languages helps us to describe de-

TABLE 1. The Performance of the Data Structures

Set Representation	O(Time per Operation)				Total Time	Space in Words
	Init	Member	Insert	Print		
Bit Vector	N/b	1	1	N/b	$O(N/b)$	N/b
Unsorted Array	1	M	1	$M \log M$	$O(M^2)$	M
Sorted Array	1	$\log M$	M	M	$O(M^2)$	M
Binary Tree	1	$\log M$	$\log M$	M	$O(M \log M)$	$3M$
Bins	M	1	1	M	$O(M)$	$3M$

signs succinctly: pseudocode represents control flow and "Abstract Data Types" represent the crucial data structures (such as the sets in Program 2). A thorough knowledge of the literature is invaluable at this stage of the design process.

Implement One Solution. On lucky days our exploration of the design space shows that one program is far superior to the rest; at other times we have to prototype the top few to choose the best. We should strive to implement the chosen design in straightforward and succinct code.*

Retrospect. Polya's delightful *How to Solve It* can help any programmer become a better problem solver. On page 15 he observes that "There remains always something to do; with sufficient study and penetration, we could improve any solution, and, in any case, we can always improve our understanding of the solution." His hints are particularly helpful for looking back at programming problems.

Problems

1. The problem specified that all M-element subsets be chosen with equal probability, which is a stronger requirement than choosing each integer with probability M/N. Describe an algorithm that chooses each element equiprobably, but chooses some subsets with greater probability than others.
2. Show that when $M < N/2$, the expected number of *Member* tests made by Program 2 before finding a number not in the set is less than 2.
3. Counting the *Member* tests in Program 2 leads to many interesting problems in combinatorics and probability theory. How many *Member* tests does the program make on the average as a function of M and N? How many does it make when $M = N$? When is it likely to make more than M tests?
4. This column described several algorithms for a single problem. Make a table describing when each is appropriate as a function of constraints on run time, space, coding time, etc.
5. [*Class Exercise*] I assigned the problem of generating sorted subsets twice in an undergraduate course on algorithms. Before the unit on sorting and searching, students had to write a program for $M = 20$ and $N = 400$; the primary grading criterion was a short, clean program—run time was not an issue. After the unit on sorting and searching they had to solve the problem again with $M = 2{,}000$ and $N = 1{,}000{,}000$, and the grade was based primarily on run time.
6. [V.A. Vyssotsky] Algorithms for generating combinatorial objects are often profitably expressed as recursive procedures. Program 1 can be written as

```
procedure RandSelect(M,N)
    if M>0 then
        if RandReal(0,1)<M/N then
            print N
            RandSelect(M-1,N-1)
        else
            RandSelect(M,N-1)
```

This program prints the random integers in decreasing order; how could you make them appear in increasing order? Argue the correctness of the resulting program. How could you use the basic recursive structure of this program to generate all M-element subsets of $1..N$?

7. How would you generate a random selection of M integers from $1..N$ with the constraint that the final output must appear in random order? How would you generate a sorted list if duplicate integers were allowed in the list? What if both duplicates and a random order were desired?
8. Describe a k-pass algorithm that uses $O(kM)$ expected time but only $O(kM^{1/k})$ space to generate a sorted list of M random integers from $1..N$, with duplicate integers allowed.
9. [M.I. Shamos] A promotional game consists of a card containing 10 spots, which hide a random permutation of the integers 1..10. The player rubs the dots off the card to expose the hidden integers. If the integer three is ever exposed, then the card loses; if one and two (in either order) are revealed, then the card wins. Describe the steps you would take to compute the probability that randomly choosing a sequence of spots wins the game; assume that you may use at most one hour of CPU time.

Further Reading

Combinatorial Algorithms for Computers and Calculators by Nijenhuis and Wilf (second edition published by Academic Press in 1978) describes a large collection of algorithms both abstractly and as ready-to-run FORTRAN programs. It is particularly strong in algorithms for generating combinatorial objects.

Solution for a November Problem

1. The sequential program reads 10,000 blocks at 200 blocks/second, so it always requires 50 seconds. The on-line program reads R records in $R/20$ seconds, so when $R = 100$ it takes five seconds; when $R = 10{,}000$ it takes about eight minutes.

* Problem 5 describes a class exercise that I graded on programming style. Most students turned in one-page solutions and received mediocre grades. Two students who had spent the previous summer on a large software development project turned in beautifully documented five-page programs, broken into about a dozen procedures, each with an elaborate heading. The students were upset when I gave their code failing grades. My reason was simple: my program worked in five lines of code, and the inflation factor of 60 was too much. John Mashey eloquently described the two styles: code that simply does the job versus voluminous declarations of good intent.

For Correspondence: Jon Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Avenue, Murray Hill, NJ 07974.