COLUMN 11: SEARCHING

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.

11.1 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 open 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 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 output is required to be sorted because the hardcopy 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 $[A,B)$.

11.2 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 contained 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 $M \le N$, 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 thirteen lines of BASIC. It was finished half an hour after the problem was defined, and has been used for several years without problems.

11.3 The Design Space

One part of a programmer's job is 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 the sampling problem.

When I talked about the problem at West Point, I asked for a better approach than the first problem statement (typing all 200 names to the program). One student suggested photocopying the precinct list, cutting the copy with a paper slicer, shaking the slips in a paper bag, and then pulling out the required number of slips. That cadet showed the "conceptual blockbusting" that is the subject of Adams's book cited in Section 1.7.†

From now on we'll confine our search to a program to write M sorted 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 dozen integers 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

---

† Page 57 of that book sketches Arthur Koestler's views on three kinds of creativity. *Ah!* insights are his name for originality, and *aha!* insights are acts of discovery. He would call this cadet's solution a *haha!* insight: the low-tech answer to a high-tech question is an act of comic inspiration (as in Solution 1.10).

```
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 described in Section 1.4 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*, which turns off all bits.

```
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, the routine *PrintInOrder* prints the elements of *S*.

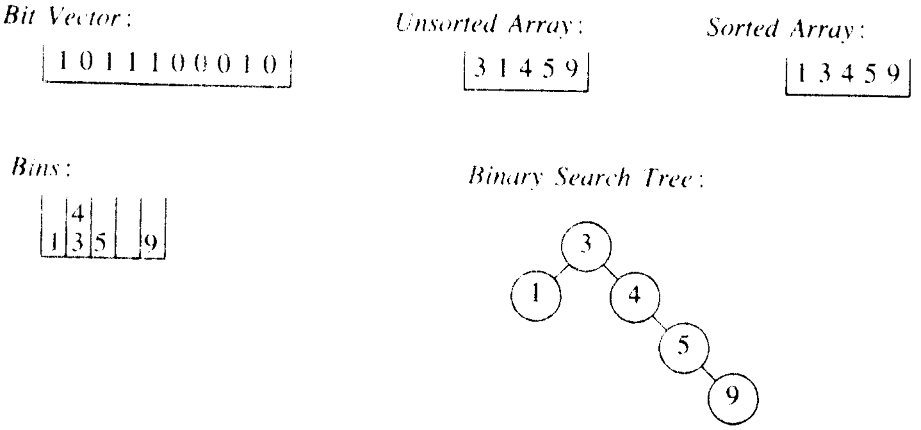
```
for I := 1 to N do
    if Bit[I] = 1 then
        print I
```

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 simultaneously operating on all *b* bits in a word (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 2*M*. Both *Member* and *Insert* require constant time per operation, so their total cost 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 an "abstract data type" of sets (a set with these operations is usually called a *dictionary*, more on this in Section 12.5). Replacing those four subroutines can change the representation of the sets and thereby change the performance of the program. This figure 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 hashes to bin (roughly) $I \times M/N$. The bins are implemented as an array of linked lists. Because the integers are uniformly distributed, each linked list has expected length one. The average performance of the various schemes, when $M<N/b$, is as follows.

SET	O(TIME PER OPERATION)				TOTAL	SPACE IN
REPRESENTATION	Init	Member	Insert	Print	TIME	WORDS
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	$O(M)$	$3M$

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 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 shuffle only the first M elements of the array, which gives 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, but the technique of Problem 1.8 reduces this to $O(M \log M)$ 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 avoid testing whether the new element was previously chosen.

Programs 1, 2 and 3 offer different solutions to the problem, but they by no means cover the possible design space. Yet another approach generates the "gaps" between successive integers in the set. J. S. Vitter's "Faster Methods for Random Sampling" in the July 1984 *Communications of the ACM* generate M sorted random integers in $O(M)$ time and constant space; those resource bounds are within a constant factor of optimal.

11.4 Principles

This column illustrates several important steps in the programming process. Although the following discussion presents the stages in one natural order, the design process is more active: we hop from one activity to another, usually iterating through 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 can be applied to other problems.