

COLUMN 10: **SORTING**

How should you sort a sequence of records into order? The answer is usually straightforward:

Use a sort command provided by the system.

Unfortunately, this plan doesn't always work. Some systems don't have a sort command, and existing sorts may not be general enough or efficient enough to solve a particular problem (as in Section 1.1). In such cases, a programmer has no choice but to write a sort routine.

10.1 Insertion Sort — An $O(N^2)$ Algorithm

Insertion Sort is the method most card players use to sort their cards. They keep the cards dealt so far in sorted order, and as each new card arrives they insert it into its proper relative position. To sort the array $X[1..N]$ into increasing order we'll start with the sorted subarray $X[1..1]$ and then insert the elements $X[2]$, ..., $X[N]$, as in the following pseudocode.

```
for I := 2 to N do
  /* Invariant:  $X[1..I-1]$  is sorted */
  /* Goal: sift  $X[I]$  down to its
     proper place in  $X[1..I-1]$  */
```

The following four lines show the progress of the algorithm on a four-element array. The "*" represents the variable I ; elements to its left are sorted, while elements to its right are in their original order.

```
3*1 4 2
1 3*4 2
1 3 4*2
1 2 3 4*
```

The sifting down is accomplished by a right-to-left loop that uses the variable J to keep track of the element being sifted. The loop swaps the element with its predecessor in the array as long as there is a predecessor (that is, $J > 1$) and the element hasn't reached its final position (that is, it is out of order with its predecessor). Thus the entire sort is

```

for I := 2 to N do
  /* Invariant: X[1..I-1] is sorted */
  J := I
  while J > 1 and X[J-1] > X[J] do
    Swap(X[J], X[J-1])
    J := J-1

```

When I need to sort and efficiency isn't an issue, that's the routine I use; it's just five lines of easy code. (In a few zealously protective languages this code may generate a run-time error; see Problem 2.)

If you don't have a *Swap* routine handy, the following assignments use the variable *T* to exchange *X[J]* and *X[J-1]*.

```

T := X[J]; X[J] := X[J-1]; X[J-1] := T

```

This code opens the door for a simple exercise in the code tuning of Column 8. Because the variable *T* is assigned the same value over and over (the value originally in *X[I]*), we can move the assignments to and from *T* out of the loop, and change the comparison as follows.

```

for I := 2 to N do
  /* Invariant: X[1..I-1] is sorted */
  J := I
  T := X[J]
  while J > 1 and X[J-1] > T do
    X[J] := X[J-1]
    J := J-1
  X[J] := T

```

This code shifts elements right into the hole vacated by *X[I]*, and finally moves *T* into the hole once it is in its final position. It is seven lines long and a little more subtle than the simple Insertion Sort, but on my system it takes just one third the time of the first program.

This routine is easy to translate, even into primitive languages such as this dialect of BASIC.

```

1000 ' SORT X(1..N), N>1
1010 FOR I=2 TO N
1015   ' INVARIANT: X(1..I-1) IS SORTED
1020   J=I
1030   T=X(J)
1040   IF J<=1 OR X(J-1)<=T THEN 1080
1050     X(J)=X(J-1)
1060     J=J-1
1070     GOTO 1040
1080   X(J)=T
1090 NEXT I
1100 RETURN

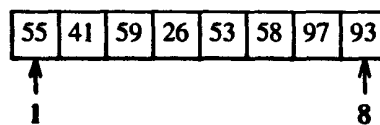
```

When I compared the run time of this program with an “efficient” sort from a 1982 BASIC text (whose sneaky logic used twice as many lines of code), I found that this simple routine required less than half the run time of its more complex cousin.

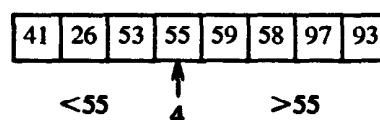
On random data as well as in the worst case, the time of Insertion Sort on an N -element array is proportional to N^2 . Fortunately, if the array is already almost sorted, the program is much faster because each element sifts down just a short distance

10.2 Quicksort — An $O(N \log N)$ Algorithm

This algorithm was described by C. A. R. Hoare in his classic paper “Quicksort” in the *Computer Journal* 5, 1, April 1962, pp. 10-15. It uses the *divide-and-conquer* schema of Section 7.3: to sort an array we divide it into two smaller pieces and sort those recursively. For instance, to sort the eight-element array



we *partition* it around the first element (55) so that all elements less than 55 are to the left of it, while all greater elements are to its right



If we then recursively sort the subarray from 1 to 3 and the subarray from 5 to 8, independently, the entire array is sorted.

The average run time of this algorithm is much less than the $O(N^2)$ time of Insertion Sort because a partitioning operation goes a long way towards sorting the sequence. After a typical partitioning of N elements, there are about $N/2$ elements above the partition value and $N/2$ elements below it. In a similar amount of run time, the sift operation of Insertion Sort manages to get just one more element into the right place.

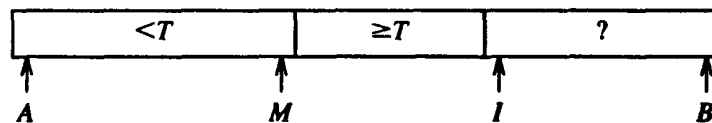
The above idea leads to a sketch of a recursive subroutine. We'll represent the portion of the array we're dealing with by the two indices L and U , for the lower and upper limits. The recursion stops when we come to an array with fewer than two elements. So the code is

```

procedure QSort(L,U)
  if L >= U then
    /* at most one element, do nothing */
  else
    /* partition array around a given
       value, which is eventually
       placed in its correct position P
    */
    QSort(L, P-1)
    QSort(P+1, U)

```

To partition the array around the value T we'll use a simple scheme that I learned from Nico Lomuto of Alsys, Inc. There are faster programs for this job[†], but this routine is so easy to understand that it's hard to get wrong, and it is by no means slow. Given the value T , we are to rearrange $X[A..B]$ and compute the index M (for "middle") such that all elements less than T are to one side of M , while all other elements are on the other side. We'll accomplish the job with a simple `for` loop that scans the array from left to right, using the variables I and M to maintain the following invariant in array X .



When the code inspects the I^{th} element there are two cases to consider. If $X[I] \geq T$ then all is fine; the invariant is still true. On the other hand, when $X[I] < T$, we can regain the invariant by incrementing M to index the new location of the small element, and then swapping that element with $X[I]$. The complete partitioning code is

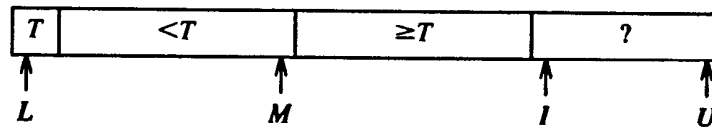
```

M := A-1
for I := A to B do
  if X[I] < T then
    M := M+1
    Swap(X[M], X[I])

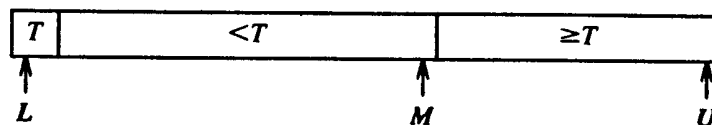
```

[†] Most presentations of Quicksort use a partitioning scheme based on two approaching indices, like the one described in Problem 3. Although the basic idea of that scheme is simple, I have always found the details tricky — I once spent the better part of two days chasing down a bug hiding in a short partitioning loop. A reader of a preliminary draft complained that the standard two-index method is in fact simpler than Lomuto's, and sketched some code to make his point; I stopped looking after I found two bugs.

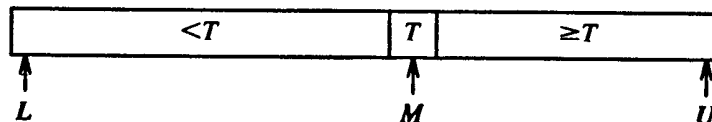
In Quicksort we'll partition the array $X[L..U]$ around the value $T = X[L]$, so A will be $L+1$ and B will be U . Thus the invariant of the partitioning loop is depicted as



When the loop terminates we have



We then swap $X[L]$ with $X[M]$ to give†



We can now recursively call the routine with the parameters $(L, M-1)$ and $(M+1, U)$.

The above algorithm always partitions around the first element in the array. This choice can require excessive time and space for some common inputs — for instance, arrays that are already sorted. We do far better to choose a partitioning element at random; we accomplish this by swapping $X[L]$ with a random entry in $X[L..U]$. If you don't have the *RandInt* function used in the code below, you can make one using a function *Rand* that returns a random real distributed uniformly in $[0,1)$ by the expression $L + \text{int}(\text{Rand} \times (U + 1 - L))$. In the unlikely event that your system doesn't even have that routine, consult Knuth's *Seminumerical Algorithms*. But whether you use a system routine or make your own, be careful that *RandInt* returns a value in the range $L..U$ — a value out of range is an insidious bug.

The final code, Quicksort 1, is presented on the next page. To sort the array $X[1..N]$ we call the procedure

QSort(1,N)

† It is tempting to ignore this step and to recur with parameters (L, M) and $(M+1, U)$; this gives an infinite loop when T is the strictly greatest element in the subarray. I would have caught the bug had I tried to verify termination, but the astute reader can guess how I really discovered it. Miriam Jacob found an elegant proof of incorrectness: since $X[L]$ is never moved, the sort can only work if the minimum element in the array starts in $X[1]$.

```

procedure QSort(L,U)
  if L < U then
    Swap(X[L], X[RandInt(L,U)])
    T := X[L]
    M := L
    for I := L+1 to U do
      /* Invariant: X[L+1..M] < T and
        X[M+1..I-1] >= T */
      if X[I] < T then
        M := M+1
        Swap(X[M], X[I])
    Swap(X[L], X[M])
    /* X[L..M-1] < X[M] <= X[M+1..U] */
    QSort(L, M-1)
    QSort(M+1, U)

```

Most of the proof of correctness of this program was given in its derivation (which is, of course, its proper place). The proof proceeds by induction: the outer *if* statement correctly handles empty and single-element arrays, and the partitioning code correctly sets up larger arrays for the recursive calls. The program can't make an infinite sequence of recursive calls because the element $X[M]$ is excluded at each invocation; this is the same argument that Section 4.3 used to show that binary search terminates.

Let's turn now to the performance of the program. I won't give the details here, but Quicksort runs in $O(N \log N)$ time and $O(\log N)$ stack space on the average, for any input array with distinct elements. The mathematical arguments are similar to those in Section 7.3, and Solution 10 contains data on one implementation. The random performance is a result of calling the random number generator, rather than an assumption about the distribution of inputs. Problems 3, 5 and 11 show ways of improving Quicksort's worst-case performance. Most algorithms texts analyze Quicksort's run time mathematically, and also prove the lower bound that any comparison-based sort must use $O(N \log N)$ comparisons; Quicksort is therefore close to optimal.

Fans of Column 8 have probably noticed several ways to tune the Quicksort code to make it faster. The simplest is indicated in Solution 8.10: we should expand the code for the *Swap* procedure in the inner loop (because the other two calls to *Swap* aren't in the inner loop, writing them in line would have a negligible impact on the speed). On my system this reduced the run time to two-thirds of what it was previously. We might also observe that a great deal of time is spent sorting very small subarrays. It would be faster to sort those using a simple method like Insertion Sort rather than firing up all the machinery of Quicksort.

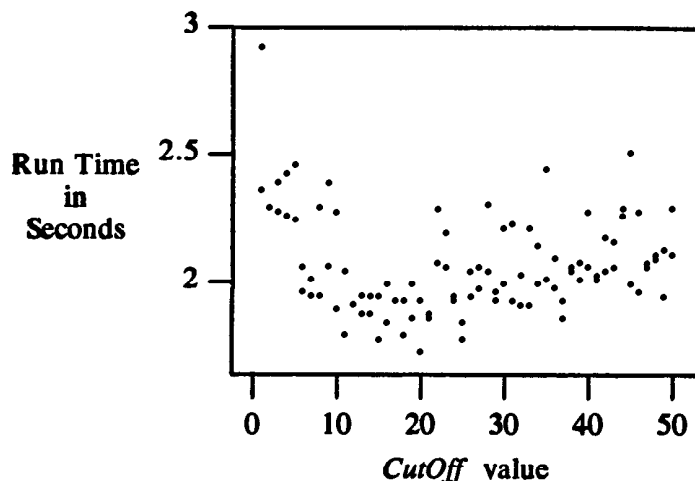
Bob Sedgewick developed a particularly clever implementation of this idea. When Quicksort is called on a small subarray (that is, when U and L are near), we do nothing. We implement this by changing the first *if* statement in the Quicksort procedure to

```
if U-L > CutOff then
```

where *CutOff* is a small integer. When the program finishes the array will not be sorted, but it will be grouped into small clumps of randomly ordered values such that the elements in one clump are less than elements in any clump to its right. We must clean up within the clumps by another sort method; because the array is almost sorted, Insertion Sort is just right for the job. We sort the entire array by the code

```
QSort(1,N)
InsertionSort
```

To determine the best choice for *CutOff*, I ran the program twice at all values of *CutOff* from 1 to 50, with *N* fixed at 5000. This graph plots the results.



A good value of *CutOff* was 15; values between 10 and 20 give savings to within a few percent of that. This change reduced the program, which we'll call Quicksort 2, to half of its original time, or another twenty-five percent reduction after writing the *Swap* procedure in line.

10.3 Principles

The programs we've studied are summarized in the following table. They were implemented in C on a VAX-11/750 and timed on random 32-bit integers; the logarithms are base two. Insertion Sort 1 is the first sort given; Insertion Sort 2 writes the *Swap* code in line and moves assignments to and from *T* out of the loop. Quicksort 1 is the first Quicksort; Quicksort 2 writes the *Swap* code in line and sorts small subarrays by calling Insertion Sort 2 after the recursive call on (1,*N*). The System Sort is the UNIX system's *qsort*. The run-time functions are the result of fitting the known form of the functions to the observed times in the table.

PROGRAM	LINES OF C CODE	RUN TIME IN MICROSECONDS	TIME IN SECONDS FOR SIZE		
			100	1000	10000
Insertion Sort 1	5	$17N^2$	0.17	17.3	1730
Insertion Sort 2	7	$6N^2$	0.06	5.7	570
Quicksort 1	11	$63N \log_2 N$	0.05	0.63	7.8
Quicksort 2	20	$32N \log_2 N$	0.03	0.32	4.3
System Sort	1	$97N \log_2 N$	0.06	1.0	13.6

We'll see yet another $O(N \log N)$ sort in Section 12.4.

There are several important lessons to be learned from the table, about both sorting in particular and programming in general.

The system sort is easy and relatively fast; it is slower than the hand-made Quicksorts only because its general and flexible interface uses a procedure call for each comparison. If a system sort can meet your needs, don't even consider writing your own code. (Section 2.8 describes two sorts available on the UNIX system: the `sort` program and the `qsort` routine.)

Insertion Sort is simple to code and may be fast enough for small sorting jobs. Sorting 10,000 integers with Insertion Sort 2 requires just ten minutes on my system.

For large N , the $O(N \log N)$ run time of Quicksort is crucial. The algorithm design techniques of Column 7 gave us the basic idea for this divide-and-conquer algorithm, and the program verification techniques of Column 4 helped us implement the idea in straightforward, succinct and efficient code.

Even though the big speedups are achieved by changing algorithms, the code tuning techniques of Column 8 speed up Insertion Sort by a factor of 3 and Quicksort by a factor of 2.

10.4 Problems

1. Like any other powerful tool, sorting is often used when it shouldn't be and often not used when it should be. Explain how sorting could be either overused or underused when calculating the following statistics of an array of N floating point numbers: minimum, maximum, mean, median and mode.
2. Suppose that $X[1..10]$ and T are declared to be integers; what happens when the following code is executed?

```

I := 11
if I <= 10 and X[I] < T then I := I+1

```

In many languages the code will execute gracefully without altering I . On some systems, though, the code might abort because the array index I is out

of bounds. What would yours do? Why is this an issue in the various Insertion Sorts? How can the problem be fixed in those sorts?

3. The Quicksort program in the text runs in time proportional to N^2 if $X[1]=X[2]=\dots=X[N]$; explain why. That problem is avoided in Problem 11 and in the following Quicksort, which is adapted from Sedgewick's paper cited in Section 10.5.

```

procedure QSort(L, U)
  if L < U then
    Swap(X[L], X[RandInt(L,U)])
    I := L; J := U+1; T := X[L]
    loop
      repeat I := I+1 until X[I] >= T
      repeat J := J-1 until X[J] <= T
      if J < I then break
      Swap(X[I], X[J])
    Swap(X[L], X[J])
    QSort(L, J-1)
    QSort(I, U)

```

This code assumes that no key in X is greater than $X[N+1]$; it uses that position as a sentinel element to increase the speed of the inner loop. On arrays of distinct elements, this code makes fewer swaps than Quicksort 1 and is therefore almost twice as fast. Use the techniques of Column 4 to prove that this program is correct. How does it solve the problem of duplicate keys?

4. [R. Sedgewick] Speed up Lomuto's partitioning scheme by using $X[L]$ as a sentinel like that described in Problem 8.5. Show how this scheme allows you to remove the *Swap* after the loop.
5. Although Quicksort uses only $O(\log N)$ stack space on the average, it can use linear space in the worst case. Explain why, then modify the program to use only logarithmic space in the worst case.
6. [M. D. McIlroy] Show how to use Lomuto's partitioning scheme to sort varying-length bit strings in time proportional to the sum of their lengths.
7. Implement several sorting programs and summarize them in a table like that in the text. In addition to Insertion Sort and Quicksort, you may want to consider Shell Sort (fair speed with simple code), Heapsort (minimal extra space and good worst-case speed — see Section 12.4), and Radix Sort (applicable only in special cases, but fast for those — see Solution 6). Does your table support the same conclusions?
8. Sketch a one-page procedure to show a user of your system how to select a sorting routine. Make sure that your method considers the importance of run time, space, programmer time (development and maintenance), generality (what if I want to sort character strings that represent Roman numerals?), stability (items with equal keys should retain their relative

order), special properties of the input data, etc. As an extreme test of your procedure, try feeding it the sorting problem described in Column 1.

9. Write a program for finding the K^{th} -smallest element in the array $X[1..N]$ in $O(N)$ expected time. Your algorithm may permute the elements of X . Present data on its run time.
10. Gather and display empirical data on the run time of a Quicksort program.
11. Write a “fat pivot” partitioning routine with the postcondition

$<T$	$=T$	$>T$
------	------	------

How would you incorporate the routine into a Quicksort program?

12. Study sorting methods used in non-computer applications (such as mail rooms and change sorters).
13. The Quicksort programs in this column choose a partitioning element at random. Study better choices, such as the median element of a sample from the array.

10.5 Further Reading

What to read about sorting depends on why you’re reading. To learn more about the subject in general, see the algorithms texts cited in Columns 2 and 7. The following references are particularly helpful for programmers writing sort routines.

If you want to write the ultimate primary-memory sort routine, see Sedgewick’s “Implementing Quicksort Programs” in the October 1978 *Communications of the ACM*.

If your job is to write a simple disk-based sort, see Chapter 4 of Kernighan and Plauger’s *Software Tools* or *Software Tools in Pascal*.

Programmers who are about to spend several months writing a quality system sort should study Knuth’s *Art of Computer Programming, volume 3: Sorting and Searching*. Linderman’s “Theory and practice in the construction of a working sort routine” (in *The Bell Laboratories Technical Journal* 63, 8, part 2, pp. 1827-1843) tells about putting that material to work in a real system sort.