by Jon L. Bentley

HOW TO SORT

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

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.[1] In such cases, a programmer has no choice but to write a sort routine.

Sort routines are old hat—they have been standard fare in *Communications of the ACM* for a quarter of a century and can be found in many textbooks. Why have I chosen to review this subject one more time?

- Too many programmers don't consult the literature. Instead, they use a "sort logic" passed on from buddy to buddy, each one adding an efficiency bell or whistle to an inefficient algorithm. If you know such a programmer, please copy these pages and pass them on.
- Books and papers on sorting algorithms often describe super-duper algorithms that are difficult to implement. This column is a first-aid kit for sorting that describes a few algorithms that are simple to understand and to implement.

In addition to being useful, the development of these routines is good clean programming fun.

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 a sort and efficiency isn't an issue, that's the routine I use: it's just five lines of easy code. (There are a few zealously protective systems on which 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 optimization. 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

[1] The August 1983 column described how a system sort couldn't be used because the routine was needed in the middle of a large system. As an example of a performance problem, consider a hypothetical programmer who calls a system sort in the innermost loop of his program to sort a few integers. Unfortunately, some sorts would do the job by writing the integers to a file, swapping the user program out to disk, reading in the sort program, sorting the file on disk, swapping back in the user program, and reading in the sorted file. Replacing such a system sort with a simple procedure could easily increase the speed of the program by a factor of a million.

©1984 ACM 0001-0782/84/0400-0287 75¢

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. In BASIC, it becomes the subroutine

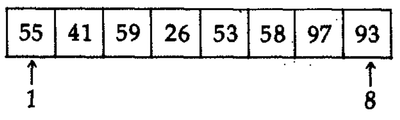
```
1000 ' SORT X(1..N)
1010 FOR I=2 TO N
1015   ' INVARIANT: X(1..I-1) IS SORTED
1020   J=I
1030   T=X(J)
1040   IF J<=1 OR X(J-1)<=T THEN 1080
1050      X(J)=X(J-1)
1060      J=J-1
1070     GOTO 1040
1080   X(J)=T
1090 NEXT I
1100 RETURN
```

When I compared the running time of this program with an "efficient" sort from a 1982 BASIC text (which 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.

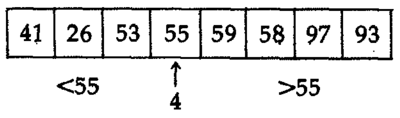
On random data as well as in the worst case, the time of Insertion Sort on an N-element array is proportional to N^2. Fortunately, if the data in the array is already in almost sorted order, the program is much faster because each element sifts down just a short distance.

Quicksort—An $O(N \log N)$ Algorithm

The Quicksort algorithm was invented by C.A.R. Hoare in the early 1960s. It uses *divide-and-conquer*: 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:



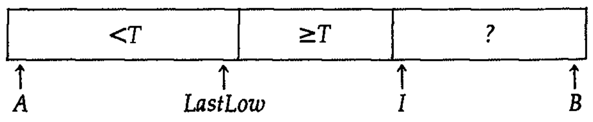
We can then recursively sort the subarray from 1 to 3 and the subarray from 5 to 8, independently, and wind up with the entire array being sorted.

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.

It's not hard to turn the above idea into 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
        /* array contains at most
           one element, do nothing */
      else
        /* partition array around
           a given value, which is
           eventually placed in
           position M */
        QSort(L, M-1)
        QSort(M+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[2], but this routine is so easy to understand that it is hard to get wrong, and it is by no means slow. Given the value T, we are to rearrange X[$A..B$] and compute an index (which we'll call *LastLow* for reasons to become clear soon) such that all elements less than T are to one side of *LastLow*, 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 *LastLow* to maintain the following invariant in array X:



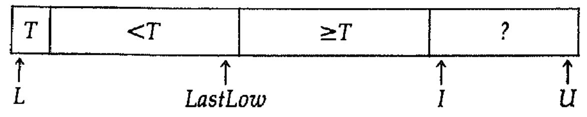
When the code inspects the I^{th} element there are two cases to consider. If X[I] $\geq T$ then all is fine; the invariant is still true. On the other hand, when X[I] $< T$ we regain the invariant by incrementing *LastLow* to index the new home of the little guy, and then swapping him

[2] Most discussions 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 straightforward, 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.

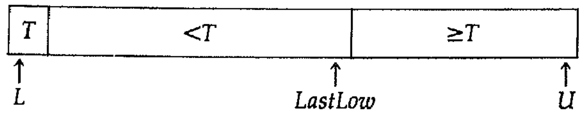
with the big guy already there. The code is

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

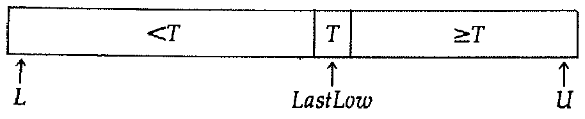
In Quicksort we'll partition the array *X*[*L*..*U*] around the value *T* = *X*[*L*], so *A* will be *L* + 1 and *B* will be *U*. Thus the invariant of the partitioning loop is depicted as



When the loop terminates we have



We then swap *X*[*L*] with *X*[*LastLow*] to give[3]



We can now recursively call the routine with the parameters (*L*, *LastLow* − 1) and (*LastLow* + 1, *U*).

The above algorithm always partitions around the first element in the array. While I won't give the analysis here, it is easy to show that this choice can require excessive time and space for some very 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*]. Thus the complete Quicksort is

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

[3] It is tempting to ignore this step and to recur with parameters (*L*, *LastLow*) and (*LastLow* + 1, *U*). Doing so leads to an infinite loop if *T* happens to be the strictly greatest element in the array. The astute reader can guess how I discovered this.

[4] 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 can be an insidious bug to track down.

If you don't have a *RandInt* function, you can make one using a function *Rand* that returns a random real distributed uniformly in [0, 1) by the expression *L* + *trunc*(*Rand* × (*U* + 1 − *L*)).[4] To sort the array X[1..*N*] we call the procedure

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

Most of the proof of correctness of this program was given in its derivation (which is, of course, its proper place). The proof itself 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 at least one element is excluded at each invocation (*X*[*LastLow*]); this is almost exactly the same argument we used in the December 1983 column to show that binary search terminates. I won't give the details here, but it is pretty easy to show that this program runs in *O*(*N* log *N*) average time and *O*(log *N*) average stack space *for any input array with distinct elements*. That is, the random performance is a result of calling the random number generator, rather than an assumption about the distribution of inputs. (The running time can be longer if the array has duplicated elements; see Problem 3.)

There are many ways to tune the Quicksort code to make it faster. Perhaps the simplest is to 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 (such as Insertion Sort) rather than spending the time to fire up all the machinery of Quicksort. Bob Sedgewick developed a particularly clever implementation of this idea. If Quicksort is called on a small subarray (that is, if *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. Thus we sort the entire array by the code

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

On my system the best choice for *CutOff* was 15; this reduced the time of the program to half of what it was originally (or another twenty-five percent reduction after writing the *swap* procedure in line). Values of *CutOff* between 8 and 30 gave savings to within a few percent of that; the best value for your system can l e found by experiment.

Principles

The programs we've studied are summarized in the following table. They were written in the C language on a VAX-11/750, they were timed on random 32-bit integers, and all logarithms are base two. Insertion Sort 1 is the first sort given; Insertion Sort 2 wrote the *swap* code in line and moved assignments to and from T out of the loop. Quicksort 1 was the first Quicksort; Quicksort 2 wrote the *swap* code in line and sorted small subarrays by calling Insertion Sort 2 after the recursive call on (1,N). The System Sort is the UNIX[5] routine *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	$63N \log N$	0.05	0.63	7.8
Quicksort 2	20	$32N \log N$	0.03	0.32	4.3
System Sort	1	$97N \log N$	0.06	1.0	13.6

The table underscores the following points.

- The system sort is easy and fast.[6] If a system command is available that meets your needs, don't even consider writing your own 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 about ten minutes on my system).
- For large N, the $O(N \log N)$ running time of Quicksort becomes crucial. The techniques of algorithm design gave us the basic idea for this divide-and-conquer algorithm, and the techniques of program verification helped us implement the idea in straightforward, succinct and efficient code.
- Even though the big speedups of sorting are achieved by changing algorithms, the code tuning techniques discussed in the February 1983 column are also useful. They speed up Insertion Sort by a factor of 3 and Quicksort by a factor of 2.

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.

[5] UNIX is a Trademark of AT&T Bell Laboratories.
[6] The system Quicksort is slower than the hand-made Quicksorts because its (general and flexible) interface uses a procedure call to make each comparison.

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
```

On many systems 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 your system do on this code? 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 $X[1] = X[2] = \cdots = X[N]$; explain why. That problem is avoided in the following Quicksort, which is adapted from Sedgewick's paper cited under further reading.

```
procedure QSort(L,U)
    if U>L 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])
          endloop
        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. Because of that and the fact that this code makes fewer swaps, this program is almost twice as fast as Quicksort 1, even on arrays of distinct elements. Use invariants to prove that this program is correct. How does it solve the problem of duplicate keys?

4. [R. Sedgewick] Modify Lomuto's partitioning scheme to remove the *swap* after the loop and to increase the speed by using a sentinel. (Hint: let the loop indices move from right to left, so that they approach the known value T in $X[L]$.)
5. 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) and Heap Sort (good speed in the worst case). Does your table support the same conclusions?
6. 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?), sta-

bility (items with equal keys should retain their relative order), special properties of the input data, etc. To test your procedure, try feeding it the sorting problem described in the August 1983 column.

7. I'd like to include in a future column a collection of sorting programs that are small and lucid, perhaps even at a slight cost in performance. W.D. Maurer has a 19-line FORTRAN implementation of Heap Sort that is quite efficient, and M.D. McIlroy has a 12-line program to sort variable-length strings in time proportional to the sum of their lengths. Do you know other programs along these lines, or cleaner versions of the Insertion Sort and Quicksort in the text?

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

Further Reading

What to read about sorting depends on why you're reading. If you want to learn more about the subject in general, you should refer to the algorithms texts mentioned in previous columns. In keeping with the how-to-theme of this column, I'll list some references particularly helpful for programmers who find themselves writing sort routines.

- If you want to write the ultimate primary-memory sort routine, Sedgewick's "Implementing Quicksort Programs" is the ideal reference (it appeared in *CACM 21*, 10, October 1978, pp. 847–857).
- If your job is to write a simple disk-based sorting package, you should certainly see how Kernighan and Plauger wrote theirs in *Software Tools* and *Software Tools in Pascal* (published in 1976 and 1981, respectively, by Addison-Wesley).
- Programmers who are about to spend several months (or more) writing a quality system sort should study Knuth's *Art of Computer Programming, volume 3: Sorting and Searching.*

COMPUTER HUMOR

My only incentive not to celebrate April Fool's day in this column is the fact that most computer humor isn't funny. As you will observe shortly, that wasn't enough. With apologies,

Have you heard how the new ⟨*your choice*⟩ computer implements a branch instruction? It holds the program counter constant and moves memory.

The CRAY-3 is so fast that it can execute an infinite loop in less than two minutes.

Why is it that most hackers confuse Halloween and Christmas? Because $31_{OCT} = 25_{DEC}$.

Although I have not been able to trace it down, I have heard that there is a paper about one of the hardest problems in Artificial Intelligence and Signal Processing entitled "How to wreck a nice beach." (Hint: say it aloud.)

The Hacker's Dictionary by Steele, Woods, Finkel, Crispin, Stallman and Goodfellow (published in 1983 by Harper and Row) does in fact contain computer humor that is both funny and educational. A brief sample, with an emphasis on edification:

Creeping Featurism *noun.* The tendency for anything complicated to become even more complicated because people keep saying, "Gee, it would be even better if it had this feature too." The result is usually a patchwork, because it grew one *ad hoc* step at a time, rather than being planned. Planning is a lot of work, but it's easy to add just one extra little feature to help someone. . . . And then another . . . and another. . . . Usually this term is used to describe computer programs, but it could also be applied to the federal government, the IRS 1040 form, and new cars.

Software Rot *noun.* A hypothetical disease the existence of which has been deduced from the observation that unused programs or FEATURES will stop working after sufficient time has passed even if "nothing has changed". Synonym: BIT DECAY.

And in their entry for JOCK, they give the following $O(N^3)$ sorting method.

An example of a brute-force program is one that sorts ten thousand numbers of examining them all, picking the smallest one, and saving it in another table; then examining all the numbers again and picking the smallest one except for the one it already picked; and in general choosing the next number by examining all ten thousand numbers and choosing the smallest one that hasn't yet been picked (as determined by examining all the ones already picked).

They go on to mention that even on fast computers this algorithm could easily require 40 days to sort 10,000 numbers. Now, that's a *funny* sorting algorithm.