For "randomly" ordered data, the operations count goes approximately as $N^{1.25}$, at least for $N < 60000$. For $N > 50$, however, Quicksort is generally faster. The program follows:

```
      SUBROUTINE shell(n,a)
      INTEGER n
      REAL a(n)
        Sorts an array a(1:n) into ascending numerical order by Shell's method (diminishing in-
        crement sort). n is input; a is replaced on output by its sorted rearrangement.
      INTEGER i,j,inc
      REAL v
      inc=1                                  Determine the starting increment.
1     inc=3*inc+1
      if(inc.le.n)goto 1
2     continue                               Loop over the partial sorts.
        inc=inc/3
        do 11 i=inc+1,n                      Outer loop of straight insertion.
          v=a(i)
          j=i
3         if(a(j-inc).gt.v)then              Inner loop of straight insertion.
            a(j)=a(j-inc)
            j=j-inc
            if(j.le.inc)goto 4
          goto 3
          endif
4         a(j)=v
        enddo 11
      if(inc.gt.1)goto 2
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.1. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 8.

8.2 Quicksort

Quicksort is, on most machines, on average, for large N, the fastest known sorting algorithm. It is a "partition-exchange" sorting method: A "partitioning element" a is selected from the array. Then by pairwise exchanges of elements, the original array is partitioned into two subarrays. At the end of a round of partitioning, the element a is in its final place in the array. All elements in the left subarray are $\leq$ a, while all elements in the right subarray are $\geq$ a. The process is then repeated on the left and right subarrays independently, and so on.

The partitioning process is carried out by selecting some element, say the leftmost, as the partitioning element a. Scan a pointer up the array until you find an element $>$ a, and then scan another pointer down from the end of the array until you find an element $<$ a. These two elements are clearly out of place for the final partitioned array, so exchange them. Continue this process until the pointers

cross. This is the right place to insert a, and that round of partitioning is done. The question of the best strategy when an element is equal to the partitioning element is subtle; we refer you to Sedgewick [1] for a discussion. (Answer: You should stop and do an exchange.)

Quicksort requires an auxiliary array of storage, of length $2\log_2 N$, which it uses as a push-down stack for keeping track of the pending subarrays. When a subarray has gotten down to some size M, it becomes faster to sort it by straight insertion (§8.1), so we will do this. The optimal setting of M is machine dependent, but $M = 7$ is not too far wrong. Some people advocate leaving the short subarrays unsorted until the end, and then doing one giant insertion sort at the end. Since each element moves at most 7 places, this is just as efficient as doing the sorts immediately, and saves on the overhead. However, on modern machines with paged memory, there is increased overhead when dealing with a large array all at once. We have not found any advantage in saving the insertion sorts till the end.

As already mentioned, Quicksort's *average* running time is fast, but its *worst case* running time can be very slow: For the worst case it is, in fact, an N^2 method! And for the most straightforward implementation of Quicksort it turns out that the worst case is achieved for an input array that is already in order! This ordering of the input array might easily occur in practice. One way to avoid this is to use a little random number generator to choose a random element as the partitioning element. Another is to use instead the median of the first, middle, and last elements of the current subarray.

The great speed of Quicksort comes from the simplicity and efficiency of its inner loop. Simply adding one unnecessary test (for example, a test that your pointer has not moved off the end of the array) can almost double the running time! One avoids such unnecessary tests by placing "sentinels" at either end of the subarray being partitioned. The leftmost sentinel is $\leq$ a, the rightmost $\geq$ a. With the "median-of-three" selection of a partitioning element, we can use the two elements that were not the median to be the sentinels for that subarray.

Our implementation closely follows [1]:

```
      SUBROUTINE sort(n,arr)
      INTEGER n,M,NSTACK
      REAL arr(n)
      PARAMETER (M=7,NSTACK=50)
        Sorts an array arr(1:n) into ascending numerical order using the Quicksort algorithm. n
        is input; arr is replaced on output by its sorted rearrangement.
        Parameters: M is the size of subarrays sorted by straight insertion and NSTACK is the required
        auxiliary storage.
      INTEGER i,ir,j,jstack,k,l,istack(NSTACK)
      REAL a,temp
      jstack=0
      l=1
      ir=n
1     if(ir-l.lt.M)then                 Insertion sort when subarray small enough.
        do 12 j=l+1,ir
          a=arr(j)
          do 11 i=j-1,l,-1
            if(arr(i).le.a)goto 2
            arr(i+1)=arr(i)
          enddo 11
          i=l-1
2         arr(i+1)=a
        enddo 12
```

```
        if(jstack.eq.0)return
        ir=istack(jstack)           Pop stack and begin a new round of partitioning.
        l=istack(jstack-1)
        jstack=jstack-2
    else
        k=(l+ir)/2                  Choose median of left, center, and right elements as par-
        temp=arr(k)                     titioning element a. Also rearrange so that a(l) ≤
        arr(k)=arr(l+1)                 a(l+1) ≤ a(ir).
        arr(l+1)=temp
        if(arr(l).gt.arr(ir))then
            temp=arr(l)
            arr(l)=arr(ir)
            arr(ir)=temp
        endif
        if(arr(l+1).gt.arr(ir))then
            temp=arr(l+1)
            arr(l+1)=arr(ir)
            arr(ir)=temp
        endif
        if(arr(l).gt.arr(l+1))then
            temp=arr(l)
            arr(l)=arr(l+1)
            arr(l+1)=temp
        endif
        i=l+1                       Initialize pointers for partitioning.
        j=ir
        a=arr(l+1)                  Partitioning element.
3       continue                    Beginning of innermost loop.
            i=i+1                   Scan up to find element > a.
        if(arr(i).lt.a)goto 3
4       continue
            j=j-1                   Scan down to find element < a.
        if(arr(j).gt.a)goto 4
        if(j.lt.i)goto 5            Pointers crossed. Exit with partitioning complete.
        temp=arr(i)                 Exchange elements.
        arr(i)=arr(j)
        arr(j)=temp
        goto 3                      End of innermost loop.
5       arr(l+1)=arr(j)             Insert partitioning element.
        arr(j)=a
        jstack=jstack+2
          Push pointers to larger subarray on stack, process smaller subarray immediately.
        if(jstack.gt.NSTACK)pause 'NSTACK too small in sort'
        if(ir-i+1.ge.j-l)then
            istack(jstack)=ir
            istack(jstack-1)=i
            ir=j-1
        else
            istack(jstack)=j-1
            istack(jstack-1)=l
            l=i
        endif
    endif
    goto 1
    END
```

As usual you can move any other arrays around at the same time as you sort `arr`. At the risk of being repetitious:

```
      SUBROUTINE sort2(n,arr,brr)
      INTEGER n,M,NSTACK
      REAL arr(n),brr(n)
      PARAMETER (M=7,NSTACK=50)
```

Sorts an array `arr(1:n)` into ascending order using Quicksort, while making the corresponding rearrangement of the array `brr(1:n)`.

```
      INTEGER i,ir,j,jstack,k,l,istack(NSTACK)
      REAL a,b,temp
      jstack=0
      l=1
      ir=n
1     if(ir-l.lt.M)then                 Insertion sort when subarray small enough.
        do 12 j=l+1,ir
          a=arr(j)
          b=brr(j)
          do 11 i=j-1,l,-1
            if(arr(i).le.a)goto 2
            arr(i+1)=arr(i)
            brr(i+1)=brr(i)
          enddo 11
          i=l-1
2         arr(i+1)=a
          brr(i+1)=b
        enddo 12
        if(jstack.eq.0)return
        ir=istack(jstack)               Pop stack and begin a new round of partitioning.
        l=istack(jstack-1)
        jstack=jstack-2
      else
        k=(l+ir)/2                      Choose median of left, center and right elements as partitioning element a. Also rearrange so that a(l) ≤ a(l+1) ≤ a(ir).
        temp=arr(k)
        arr(k)=arr(l+1)
        arr(l+1)=temp
        temp=brr(k)
        brr(k)=brr(l+1)
        brr(l+1)=temp
        if(arr(l).gt.arr(ir))then
          temp=arr(l)
          arr(l)=arr(ir)
          arr(ir)=temp
          temp=brr(l)
          brr(l)=brr(ir)
          brr(ir)=temp
        endif
        if(arr(l+1).gt.arr(ir))then
          temp=arr(l+1)
          arr(l+1)=arr(ir)
          arr(ir)=temp
          temp=brr(l+1)
          brr(l+1)=brr(ir)
          brr(ir)=temp
        endif
        if(arr(l).gt.arr(l+1))then
          temp=arr(l)
          arr(l)=arr(l+1)
          arr(l+1)=temp
          temp=brr(l)
          brr(l)=brr(l+1)
          brr(l+1)=temp
        endif
        i=l+1                           Initialize pointers for partitioning.
        j=ir
        a=arr(l+1)                      Partitioning element.
        b=brr(l+1)
```

```
3       continue                        Beginning of innermost loop.
            i=i+1                       Scan up to find element > a.
        if(arr(i).lt.a)goto 3
4       continue
            j=j-1                       Scan down to find element < a.
        if(arr(j).gt.a)goto 4
        if(j.lt.i)goto 5                Pointers crossed. Exit with partitioning complete.
        temp=arr(i)                     Exchange elements of both arrays.
        arr(i)=arr(j)
        arr(j)=temp
        temp=brr(i)
        brr(i)=brr(j)
        brr(j)=temp
        goto 3                          End of innermost loop.
5       arr(l+1)=arr(j)                 Insert partitioning element in both arrays.
        arr(j)=a
        brr(l+1)=brr(j)
        brr(j)=b
        jstack=jstack+2
          Push pointers to larger subarray on stack, process smaller subarray immediately.
        if(jstack.gt.NSTACK)pause 'NSTACK too small in sort2'
        if(ir-i+1.ge.j-l)then
            istack(jstack)=ir
            istack(jstack-1)=i
            ir=j-1
        else
            istack(jstack)=j-1
            istack(jstack-1)=l
            l=i
        endif
    endif
    goto 1
    END
```

You could, in principle, rearrange any number of additional arrays along with `brr`, but this becomes wasteful as the number of such arrays becomes large. The preferred technique is to make use of an index table, as described in §8.4.

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1978, *Communications of the ACM*, vol. 21, pp. 847–857. [1]

8.3 Heapsort

While usually not quite as fast as Quicksort, Heapsort is one of our favorite sorting routines. It is a true "in-place" sort, requiring no auxiliary storage. It is an $N \log_2 N$ process, not only on average, but also for the worst-case order of input data. In fact, its worst case is only 20 percent or so worse than its average running time.

It is beyond our scope to give a complete exposition on the theory of Heapsort. We will mention the general principles, then let you refer to the references [1,2], or analyze the program yourself, if you want to understand the details.

A set of N numbers $a_i,\ i = 1, \ldots, N$, is said to form a "heap" if it satisfies the relation

$$a_{j/2} \geq a_j \quad \text{for} \quad 1 \leq j/2 < j \leq N \tag{8.3.1}$$