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:

```
void shell(unsigned long n, float a[])
Sorts an array a[1..n] into ascending numerical order by Shell's method (diminishing increment
sort). n is input; a is replaced on output by its sorted rearrangement.
{
    unsigned long i,j,inc;
    float v;
    inc=1;                              Determine the starting increment.
    do {
        inc *= 3;
        inc++;
    } while (inc <= n);
    do {                                Loop over the partial sorts.
        inc /= 3;
        for (i=inc+1;i<=n;i++) {        Outer loop of straight insertion.
            v=a[i];
            j=i;
            while (a[j-inc] > v) {      Inner loop of straight insertion.
                a[j]=a[j-inc];
                j -= inc;
                if (j <= inc) break;
            }
            a[j]=v;
        }
    } while (inc > 1);
}
```

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.)

For speed of execution, we do not implement Quicksort using recursion. Thus the algorithm 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]:

```
#include "nrutil.h"
#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;
#define M 7
#define NSTACK 50
```
Here M is the size of subarrays sorted by straight insertion and NSTACK is the required auxiliary storage.

```
void sort(unsigned long n, float arr[])
```
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.

```
{
    unsigned long i,ir=n,j,k,l=1,*istack;
    int jstack=0;
    float a,temp;

    istack=lvector(1,NSTACK);
    for (;;) {                              Insertion sort when subarray small enough.
        if (ir-l < M) {
            for (j=l+1;j<=ir;j++) {
                a=arr[j];
                for (i=j-1;i>=l;i--) {
                    if (arr[i] <= a) break;
                    arr[i+1]=arr[i];
```

```
            }
            arr[i+1]=a;
        }
        if (jstack == 0) break;
        ir=istack[jstack--];          Pop stack and begin a new round of parti-
        l=istack[jstack--];               tioning.
    } else {
        k=(l+ir) >> 1;                Choose median of left, center, and right el-
        SWAP(arr[k],arr[l+1])             ements as partitioning element a. Also
        if (arr[l] > arr[ir]) {           rearrange so that a[l] ≤ a[l+1] ≤ a[ir].
            SWAP(arr[l],arr[ir])
        }
        if (arr[l+1] > arr[ir]) {
            SWAP(arr[l+1],arr[ir])
        }
        if (arr[l] > arr[l+1]) {
            SWAP(arr[l],arr[l+1])
        }
        i=l+1;                        Initialize pointers for partitioning.
        j=ir;
        a=arr[l+1];                   Partitioning element.
        for (;;) {                    Beginning of innermost loop.
            do i++; while (arr[i] < a);       Scan up to find element > a.
            do j--; while (arr[j] > a);       Scan down to find element < a.
            if (j < i) break;         Pointers crossed. Partitioning complete.
            SWAP(arr[i],arr[j]);      Exchange elements.
        }                             End of innermost loop.
        arr[l+1]=arr[j];              Insert partitioning element.
        arr[j]=a;
        jstack += 2;
        Push pointers to larger subarray on stack, process smaller subarray immediately.
        if (jstack > NSTACK) nrerror("NSTACK too small in sort.");
        if (ir-i+1 >= j-l) {
            istack[jstack]=ir;
            istack[jstack-1]=i;
            ir=j-1;
        } else {
            istack[jstack]=j-1;
            istack[jstack-1]=l;
            l=i;
        }
    }
  }
  free_lvector(istack,1,NSTACK);
}
```

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

```
#include "nrutil.h"
#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;
#define M 7
#define NSTACK 50

void sort2(unsigned long n, float arr[], float brr[])
Sorts an array arr[1..n] into ascending order using Quicksort, while making the corresponding
rearrangement of the array brr[1..n].
{
    unsigned long i,ir=n,j,k,l=1,*istack;
    int jstack=0;
    float a,b,temp;
```

```
istack=lvector(1,NSTACK);
for (;;) {                              Insertion sort when subarray small enough.
    if (ir-l < M) {
        for (j=l+1;j<=ir;j++) {
            a=arr[j];
            b=brr[j];
            for (i=j-1;i>=l;i--) {
                if (arr[i] <= a) break;
                arr[i+1]=arr[i];
                brr[i+1]=brr[i];
            }
            arr[i+1]=a;
            brr[i+1]=b;
        }
        if (!jstack) {
            free_lvector(istack,1,NSTACK);
            return;
        }
        ir=istack[jstack];              Pop stack and begin a new round of parti-
        l=istack[jstack-1];                 tioning.
        jstack -= 2;
    } else {
        k=(l+ir) >> 1;                  Choose median of left, center and right el-
        SWAP(arr[k],arr[l+1])               ements as partitioning element a. Also
        SWAP(brr[k],brr[l+1])               rearrange so that a[l] ≤ a[l+1] ≤ a[ir].
        if (arr[l] > arr[ir]) {
            SWAP(arr[l],arr[ir])
            SWAP(brr[l],brr[ir])
        }
        if (arr[l+1] > arr[ir]) {
            SWAP(arr[l+1],arr[ir])
            SWAP(brr[l+1],brr[ir])
        }
        if (arr[l] > arr[l+1]) {
            SWAP(arr[l],arr[l+1])
            SWAP(brr[l],brr[l+1])
        }
        i=l+1;                          Initialize pointers for partitioning.
        j=ir;
        a=arr[l+1];                     Partitioning element.
        b=brr[l+1];
        for (;;) {                      Beginning of innermost loop.
            do i++; while (arr[i] < a);     Scan up to find element > a.
            do j--; while (arr[j] > a);     Scan down to find element < a.
            if (j < i) break;           Pointers crossed. Partitioning complete.
            SWAP(arr[i],arr[j])         Exchange elements of both arrays.
            SWAP(brr[i],brr[j])
        }                               End of innermost loop.
        arr[l+1]=arr[j];                Insert partitioning element in both arrays.
        arr[j]=a;
        brr[l+1]=brr[j];
        brr[j]=b;
        jstack += 2;
        Push pointers to larger subarray on stack, process smaller subarray immediately.
        if (jstack > NSTACK) nrerror("NSTACK too small in sort2.");
        if (ir-i+1 >= j-l) {
            istack[jstack]=ir;
            istack[jstack-1]=i;
            ir=j-1;
        } else {
            istack[jstack]=j-1;
            istack[jstack-1]=l;
            l=i;
        }
```

```
        }
    }
}
```

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 $a_i,\ i = 1,\dots,N$, is said to form a "heap" if it satisfies the relation

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

Here the division in $j/2$ means "integer divide," i.e., is an exact integer or else is rounded down to the closest integer. Definition (8.3.1) will make sense if you think of the numbers a_i as being arranged in a binary tree, with the top, "boss," node being a_1, the two "underling" nodes being a_2 and a_3, *their* four underling nodes being a_4 through a_7, etc. (See Figure 8.3.1.) In this form, a heap has every "supervisor" greater than or equal to its two "supervisees," down through the levels of the hierarchy.

If you have managed to rearrange your array into an order that forms a heap, then sorting it is very easy: You pull off the "top of the heap," which will be the largest element yet unsorted. Then you "promote" to the top of the heap its largest underling. Then you promote *its* largest underling, and so on. The process is like what happens (or is supposed to happen) in a large corporation when the chairman of the board retires. You then repeat the whole process by retiring the new chairman of the board. Evidently the whole thing is an $N \log_2 N$ process, since each retiring chairman leads to $\log_2 N$ promotions of underlings.

Well, how do you arrange the array into a heap in the first place? The answer is again a "sift-up" process like corporate promotion. Imagine that the corporation starts out with $N/2$ employees on the production line, but with no supervisors. Now a supervisor is hired to supervise two workers. If he is less capable than one of his workers, that one is promoted in his place, and he joins the production line.