For small N one does better to use an algorithm whose operation count goes as a higher, i.e., poorer, power of N, if the constant in front is small enough. For $N < 20$, roughly, the method of *straight insertion* (§8.1) is concise and fast enough. We include it with some trepidation: It is an N^2 algorithm, whose potential for misuse (by using it for too large an N) is great. The resultant waste of computer time is so awesome, that we were tempted not to include any N^2 routine at all. We *will* draw the line, however, at the inefficient N^2 algorithm, beloved of elementary computer science texts, called *bubble sort*. If you know what bubble sort is, wipe it from your mind; if you don't know, make a point of never finding out!

For $N < 50$, roughly, *Shell's method* (§8.1), only slightly more complicated to program than straight insertion, is competitive with the more complicated Quicksort on many machines. This method goes as $N^{3/2}$ in the worst case, but is usually faster.

See references [1,2] for further information on the subject of sorting, and for detailed references to the literature.

CITED REFERENCES AND FURTHER READING:

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

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapters 8–13. [2]

8.1 Straight Insertion and Shell's Method

Straight insertion is an N^2 routine, and should be used only for small N, say < 20.

The technique is exactly the one used by experienced card players to sort their cards: Pick out the second card and put it in order with respect to the first; then pick out the third card and insert it into the sequence among the first two; and so on until the last card has been picked out and inserted.

```
      SUBROUTINE piksrt(n,arr)
      INTEGER n
      REAL arr(n)
         Sorts an array arr(1:n) into ascending numerical order, by straight insertion. n is input;
         arr is replaced on output by its sorted rearrangement.
      INTEGER i,j
      REAL a
      do 12 j=2,n                        Pick out each element in turn.
        a=arr(j)
        do 11 i=j-1,1,-1                 Look for the place to insert it.
          if(arr(i).le.a)goto 10
          arr(i+1)=arr(i)
        enddo 11
        i=0
10      arr(i+1)=a                       Insert it.
      enddo 12
      return
      END
```

What if you also want to rearrange an array `brr` at the same time as you sort `arr`? Simply move an element of `brr` whenever you move an element of `arr`:

```
      SUBROUTINE piksr2(n,arr,brr)
      INTEGER n
      REAL arr(n),brr(n)
         Sorts an array arr(1:n) into ascending numerical order, by straight insertion, while making
         the corresponding rearrangement of the array brr(1:n).
      INTEGER i,j
      REAL a,b
      do 12 j=2,n                          Pick out each element in turn.
          a=arr(j)
          b=brr(j)
          do 11 i=j-1,1,-1                 Look for the place to insert it.
              if(arr(i).le.a)goto 10
              arr(i+1)=arr(i)
              brr(i+1)=brr(i)
          enddo 11
          i=0
10        arr(i+1)=a                       Insert it.
          brr(i+1)=b
      enddo 12
      return
      END
```

For the case of rearranging a larger number of arrays by sorting on one of them, see §8.4.

Shell's Method

This is actually a variant on straight insertion, but a very powerful variant indeed. The rough idea, e.g., for the case of sorting 16 numbers $n_1 \dots n_{16}$, is this: First sort, by straight insertion, each of the 8 groups of 2 (n_1, n_9), (n_2, n_{10}), ..., (n_8, n_{16}). Next, sort each of the 4 groups of 4 (n_1, n_5, n_9, n_{13}), ..., $(n_4, n_8, n_{12}, n_{16})$. Next sort the 2 groups of 8 records, beginning with $(n_1, n_3, n_5, n_7, n_9, n_{11}, n_{13}, n_{15})$. Finally, sort the whole list of 16 numbers.

Of course, only the *last* sort is *necessary* for putting the numbers into order. So what is the purpose of the previous partial sorts? The answer is that the previous sorts allow numbers efficiently to filter up or down to positions close to their final resting places. Therefore, the straight insertion passes on the final sort rarely have to go past more than a "few" elements before finding the right place. (Think of sorting a hand of cards that are already almost in order.)

The spacings between the numbers sorted on each pass through the data (8,4,2,1 in the above example) are called the *increments*, and a Shell sort is sometimes called a *diminishing increment sort*. There has been a lot of research into how to choose a good set of increments, but the optimum choice is not known. The set $\dots, 8, 4, 2, 1$ is in fact not a good choice, especially for N a power of 2. A much better choice is the sequence

$$(3^k - 1)/2, \dots, 40, 13, 4, 1 \tag{8.1.1}$$

which can be generated by the recurrence

$$i_1 = 1, \qquad i_{k+1} = 3i_k + 1, \quad k = 1, 2, \dots \tag{8.1.2}$$

It can be shown (see [1]) that for this sequence of increments the number of operations required in all is of order $N^{3/2}$ for the worst possible ordering of the original data.

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