The basic idea of the Shell sort is that in the early stages far-apart elements are compared, instead of adjacent ones. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. For each value of the interval between compared elements, if no exchanges have been made, the interval is decreased, until it reaches one, at which point it effectively becomes a simple interchange sort. If no exchanges are made when the interval is one, the data are sorted.

```
      SUBROUTINE SHELL(X, N)
      REAL X(N)
C  SORTS UP.  IF THERE ARE NO EXCHANGES (IEX=0) ON A SWEEP
C  THE COMPARISON GAP (IGAP) IS HALVED FOR THE NEXT SWEEP
      IGAP = N
    5 IF (IGAP .LE. 1) RETURN
      IGAP = IGAP/2
      IMAX = N-IGAP
   10    IEX = 0
         DO 20 I = 1,IMAX
            IPLUSG = I+IGAP
            IF (X(I) .LE. X(IPLUSG)) GOTO 20
               SAVE = X(I)
               X(I) = X(IPLUSG)
               X(IPLUSG) = SAVE
               IEX = 1
   20    CONTINUE
         IF (IEX .NE. 0) GOTO 10
      GOTO 5
      END
```

Here are the run time comparisons, in milliseconds:

size	"efficient"	simple	Shell
10	1	1	1.7
50	22	19	20
300	850	670	260
2000	38500	29200	3200

The run times speak for themselves — not only is the Shell sort faster by a factor of nine at 2000 elements, but the rate of increase is lower. (Be it noted that the Shell sort is *not* the fastest sort available; it is merely an easy step up from the usual interchange sorts.)

There are two lessons. First, time spent selecting a good algorithm is certain to pay larger dividends than time spent polishing an implementation of a poor method. Second, for any given algorithm, polishing is not likely to significantly improve a fundamentally sound, clean implementation. It may even make things worse.

Don't diddle code to make it faster — find a better algorithm.

Our conclusions about the sort programs are based on measurements, not on *a priori* notions of what will or will not be efficient. For example, theoretical studies