predict that for large values of n, the Shell sort will be substantially faster than any interchange sort, for its run time grows as no more than $n^{1.5}$ instead of n^2. Common sense says that for small n, interchange sorts will be faster because they are simpler.

Neither theory nor common sense tells us where the cross-over takes place; that depends on programming. Measurements show that, for our particular programs, the transition takes place for n around 50, but that the disparity is not impractical even when n is 300.

These measurements are obtained by using a timing package to time a particular piece of code, like this:

```
CALL TICK(TIME)
code to be timed
CALL TICK(TIME)
```

TIME is set to the elapsed computation time since the last call to TICK, to whatever resolution the operating system provides. Most computer systems provide such a service. (An even better service, less commonly available, times each subroutine without any need to explicitly reference the timing package in the program being timed.)

Timing is not always sufficient. For example, precisely why is the simple sort faster than the "efficient" one? The real work of each is in comparisons and exchanges; the rest is bookkeeping. Could it be that the simple sort is faster because somehow it does much less real work, or does it just do less bookkeeping?

Instrumenting the program to make a simple measurement gives us the clue. We add two counters, NCOMP and NEXCH, to each program:

```
      ...
C COUNT COMPARISONS
      NCOMP = NCOMP+1
      IF (X(I) .GE. X(J)) GOTO 10
C OUT OF ORDER; EXCHANGE AND COUNT
         NEXCH = NEXCH+1
         SAVE = X(I)
         ...
```

The counters are initialized and printed outside the sort. Here are the results (including the Shell sort):

size	"efficient"	simple	Shell	
10	43	54	60	comparisons
	22	22	13	exchanges
50	1020	1280	880	
	570	570	150	
300	41000	45100	11500	
	22400	22400	1500	
2000	1920000	2000000	147000	
	1000000	1000000	18600	