The two interchange sorts do the same number of exchanges. But, although the "efficient" sort does fewer comparisons than the simple sort, the saving does not offset the cost of all the other operations. With some confidence we can conclude that the simple sort is faster because its bookkeeping is simpler. The Shell sort column shows conclusively why it is faster than the others for large *n*.

Beware of preconceptions about where a program spends its time. This avoids the error of looking in the wrong place for improvements. Of course, you have to have some working idea of which part of a program has the most effect on overall speed, but changes designed to improve efficiency should be based on solid measurement, not intuition.

A useful and cheap way to measure how a program spends its time is to count how many times each statement is executed. The resulting set of counts is called the program's "profile" (a term first used by D. E. Knuth in an article in *Software Practice and Experience,* April, 1971). Some enlightened computer centers make available a "profiler" to do this automatically for your program. It works by temporarily adding "N=N+1" statements to appropriate parts of the program.

Instrumentation such as counts, profiles, and subroutine timings helps you concentrate effort on those parts of the code which really need improvement. Although we have already obtained improvements without the aid of the profiler, we can illustrate its potential. Here is a program that collates grades, counting right and wrong answers for each student:

```
      ISUM=0
      DO 3 I=1,5
      IF(CORANS(I).EQ.STUANS(I))GO TO 4
      ICHECK(I)=0
      GO TO 30
4     ICHECK(I)=1
      ISUM=ISUM+1
30    IWRONG=5-ISUM
3     CONTINUE
```

If we take the profile of this code, we observe that the statement

```
30    IWRONG=5-ISUM
```

is executed five times for each student. Why is this necessary? Clearly the number of wrong answers need only be computed once, after we know how many were right (assuming there are no other possibilities).

The code contains a "performance bug" — although correct, it does more work than necessary, because of misplaced code. The statement should be outside the loop, with its label removed and the GO TO 30 changed to GOTO 3.

Instrument your programs.
Measure before making "efficiency" changes.

The cost of computing hardware has steadily decreased; software cost has steadily increased. "Efficiency" should concentrate on reducing the expensive parts of computing. To summarize the main points of this chapter.