```
      SUBROUTINE ARRFAC(A, N)
      INTEGER A(N)
      DO 20 I = 1, N
         K = MAX0(A(I), 1)
         A(I) = 1
         DO 10 J = 1, K
            A(I) = A(I) * J
   10    CONTINUE
   20 CONTINUE
      RETURN
      END
```

If the utmost in speed were of any importance here, the way to get it is obviously not to test whether K is one, but to pre-compute a table of factorials, and index into that directly. The table will not be very big on any current computer, since the factorials rapidly get too large to store as exact integers. We have remarked several times that a change in data representation often simplifies control structure more profoundly than any amount of tweaking. Efficiency likewise depends strongly on data representation.

Make it clear before you make it faster.

This brings us to another important point: simplicity and clarity are often of more value than the microseconds possibly saved by clever coding. For instance, one text suggests that the loop

```
      DO 4 J = 1,1000
    4 X(J) = J
```

be replaced by

```
      Z = 0.0
      DO 4 J = 1,1000
      Z = Z + 1.0
    4 X(J) = Z
```

presumably to avoid a thousand conversions to floating point.

This is an excellent example of nit-picking. One compiler we tried compiled the first loop into nine machine instructions, the second loop into eight. If all instructions were to take about the same time, the time saving would be a little over ten per cent. If this loop is ten per cent of an entire program (which seems high, since it is clearly just an initialization), the "improved" program would run one percent faster. As a matter of fact, the "improved" code uses a higher proportion of floating point instructions, which are more time consuming, so any saving is debatable. A second compiler *increased* the number of instructions for the "improved" version from ten to eleven, making the program slower! Trivia rarely affect efficiency. Are all the machinations worth it, when their primary effect is to make the code less readable?