reasonable places to look for bugs are where N is 2^k, 2^k-1, and 2^k+1. Of course one boundary is always the trivial or null case, here when N equals one. And we hit the jackpot.

But by now we should have learned to be suspicious: where one bug is found, there may be an infestation. So we looked at another boundary, the case where the table contained a match for the input entry. And again we hit the jackpot.

For practice, let us examine the boundaries of the following routine, which computes the mean of a set of numbers.

```
      DIMENSION X(201)
      READ (5, 100) TEST
100 FORMAT (F10.0)
C INITIALIZE
      SUM = 0.0
      COUNT = 0.0
C ADVANCE COUNTER ON EACH RETURN TO THIS POINT
    4 COUNT = COUNT + 1.0
      I = COUNT
C READ A DATA ITEM
      READ (5, 100) X(I)
C CHECK FOR END-OF-DECK SENTINEL
      IF (X(I) .EQ. TEST) GO TO 9
C PROCESS THE ITEM
      SUM = SUM + X(I)
      GO TO 4
C COMPUTE THE MEAN
    9 AVG = SUM / COUNT
      . . .
```

The value TEST is read first; the next occurrence of this value signals the end of the input.

The most obvious boundary in any program, and often the easiest to test, is what it does when presented with no data at all. So what happens if we try to find the average of zero items?

Rather surprisingly, the program does not attempt to divide by zero (which is good), yet the very fact that it divides by 1 instead of zero is suspicious, since no special precautions were taken. The next boundary, also easy, is for one data item. And this time we see what is wrong. When there is one data item, the program divides the running sum by 2 instead of 1. From here it is easy to verify that COUNT, which counts the number of data items, is always 1 too high when the average is computed. The program is only asymptotically correct.

Dividing by COUNT-1 instead of COUNT is a quick fix (except when COUNT is 1) but it is usually wiser to rewrite than to patch, just to make sure that all the problems have been found and that the code is still clean. The root of the trouble here is that there are two counters, I and COUNT, yet only one thing to count, and the states of the counters have become confused. Programs often have redundant variables, perhaps for historical reasons, that serve only to add unnecessary complexity. Considerable energy can be spent just trying to keep them in phase. COUNT and I are both incremented *before* a new value is read in. This is done so the new value can be put into X(I) before testing if it is an end of file signal. Of course COUNT should only be incremented after the new value is known not to be an end of file signal. Interestingly enough, the book from which this example is taken shows a