always uses the midpoint. However, it would be silly to search for *Hank Aaron* in the middle of the phone book; somewhere near the start clearly would be more appropriate. Thus, instead of `mid`, we use `next` to indicate the next item we will try to access.

Here's an example of what might be good. Suppose the range contains 1,000 items, the low item in the range is 1,000, the high item in the range is 1,000,000, and we are searching for an item of value 12,000. If the items are uniformly distributed, then we expect to find a match somewhere near the twelfth item. The applicable formula is

$$next = low + \left\lceil \frac{x - a[low]}{a[high] - a[low]} \times (high - low - 1) \right\rceil.$$

The subtraction of 1 is a technical adjustment that has been shown to perform well in practice. Clearly, this calculation is more costly than the binary search calculation. It involves an extra division (the division by 2 in the binary search is really just a bit shift, just as dividing by 10 is easy for humans), multiplication, and four subtractions. These calculations need to be done using floating-point operations. One iteration may be slower than the complete binary search. However, if the cost of these calculations is insignificant when compared to the cost of accessing an item, this is immaterial; we care only about the number of iterations.

Interpolation search has a better Big-Oh bound on average than does binary search, but it has limited practicality and a bad worst case.

In the worst case, where data is not uniformly distributed, the running time could be linear and every item might be examined. Exercise 5.18 asks you to construct such a case. However, under the assumption that the items are reasonably distributed, as with a phone book, the average number of comparisons has been shown to be $O(\log\log N)$. This means that we apply the logarithm twice in succession. For $N = 4$ billion, $\log N$ is about 32 and $\log\log N$ is roughly 5. Of course, there are some hidden constants in Big-Oh notation, but the extra logarithm can lower the number of iterations considerably, as long as a bad case does not crop up. Proving the result rigorously, however, is quite complicated.

5.7 Checking an Algorithm Analysis

Once we have performed an algorithm analysis, we want to see if it is correct and as good as possible. One way to do this is to code the program and see if the empirically observed running time matches the running time predicted by the analysis.

When N increases by a factor of ten, the running time goes up by a factor of ten for linear programs, 100 for quadratic programs, and 1,000 for cubic programs. Programs that run in $O(N \log N)$ take slightly more than ten times as long to run under the same circumstances. These increases can be hard to spot if the lower-order terms have relatively large coefficients and N is not large enough. An example is the jump from $N = 10$ to $N = 100$ in the running time for the various