More generally, a sample, S, of s elements is chosen from the N elements. Let δ be some number, which we will choose later so as to minimize the average number of comparisons used by the procedure. We find the $(v_1 = ks/N - \delta)$th and $(v_2 = ks/N + \delta)$th smallest elements in S'. Almost certainly, the kth smallest element in S will fall between v_1 and v_2, so we are left with a selection problem on 2δ elements. With low probability, the kth smallest element does not fall in this range, and we have considerable work to do. However, with a good choice of s and δ, we can ensure, by the laws of probability, that the second case does not adversely affect the total work.

If an analysis is performed, we find that if $s = N^{2/3}\log^{1/3} N$ and $\delta = N^{1/3}\log^{2/3} N$, then the expected number of comparisons is $N + k + O(N^{2/3}\log^{1/3} N)$, which is optimal except for the low-order term. (If $k > N/2$, we can consider the symmetric problem of finding the $(N - k)$th largest element.)

Most of the analysis is easy to do. The last term represents the cost of performing the two selections to determine v_1 and v_2. The average cost of the partitioning, assuming a reasonably clever strategy, is equal to N plus the expected rank of v_2 in S, which is $N + k + O(N\delta/s)$. If the kth element winds up in S', the cost of finishing the algorithm is equal to the cost of selection on S', namely, $O(s)$. If the kth smallest element doesn't wind up in S', the cost is $O(N)$. However, s and δ have been chosen to guarantee that this happens with very low probability $o(1/N)$, so the expected cost of this possibility is $o(1)$, which is a term that goes to zero as N gets large. An exact calculation is left as Exercise 10.22.

This analysis shows that finding the median requires about $1.5N$ comparisons on average. Of course, this algorithm requires some floating-point arithmetic to compute s, which can slow down the algorithm on some machines. Even so, experiments have shown that if correctly implemented, this algorithm compares favorably with the quickselect implementation in Chapter 7.

10.2.4 Theoretical Improvements for Arithmetic Problems

In this section we describe a divide-and-conquer algorithm that multiplies two N-digit numbers. Our previous model of computation assumed that multiplication was done in constant time, because the numbers were small. For large numbers, this assumption is no longer valid. If we measure multiplication in terms of the size of numbers being multiplied, then the natural multiplication algorithm takes quadratic time. The divide-and-conquer algorithm runs in subquadratic time. We also present the classic divide-and-conquer algorithm that multiplies two N-by-N matrices in subcubic time.

Multiplying Integers

Suppose we want to multiply two N-digit numbers, X and Y. If exactly one of X and Y is negative, then the answer is negative; otherwise it is positive. Thus, we can perform this check and then assume that $X, Y \geq 0$. The algorithm that almost everyone uses when multiplying by hand requires $\Theta(N^2)$ operations, because each digit in X is multiplied by each digit in Y.

If $X = 61{,}438{,}521$ and $Y = 94{,}736{,}407$, $XY = 5{,}820{,}464{,}730{,}934{,}047$. Let us break X and Y into two halves, consisting of the most significant and least significant digits,

respectively. Then $X_L = 6{,}143$, $X_R = 8{,}521$, $Y_L = 9{,}473$, and $Y_R = 6{,}407$. We also have $X = X_L 10^4 + X_R$ and $Y = Y_L 10^4 + Y_R$. It follows that

$$XY = X_L Y_L 10^8 + (X_L Y_R + X_R Y_L)10^4 + X_R Y_R$$

Notice that this equation consists of four multiplications, $X_L Y_L$, $X_L Y_R$, $X_R Y_L$, and $X_R Y_R$, which are each half the size of the original problem ($N/2$ digits). The multiplications by 10^8 and 10^4 amount to the placing of zeros. This and the subsequent additions add only $O(N)$ additional work. If we perform these four multiplications recursively using this algorithm, stopping at an appropriate base case, then we obtain the recurrence

$$T(N) = 4T(N/2) + O(N)$$

From Theorem 10.6, we see that $T(N) = O(N^2)$, so, unfortunately, we have not improved the algorithm. To achieve a subquadratic algorithm, we must use less than four recursive calls. The key observation is that

$$X_L Y_R + X_R Y_L = (X_L - X_R)(Y_R - Y_L) + X_L Y_L + X_R Y_R$$

Thus, instead of using two multiplications to compute the coefficient of 10^4, we can use one multiplication, plus the result of two multiplications that have already been performed. Figure 10.37 shows how only three recursive subproblems need to be solved.

Function	Value	Computational Complexity
X_L	6,143	Given
X_R	8,521	Given
Y_L	9,473	Given
Y_R	6,407	Given
$D_1 = X_L - X_R$	−2,378	$O(N)$
$D_2 = Y_R - Y_L$	−3,066	$O(N)$
$X_L Y_L$	58,192,639	$T(N/2)$
$X_R Y_R$	54,594,047	$T(N/2)$
$D_1 D_2$	7,290,948	$T(N/2)$
$D_3 = D_1 D_2 + X_L Y_L + X_R Y_R$	120,077,634	$O(N)$
$X_R Y_R$	54,594,047	Computed above
$D_3 10^4$	1,200,776,340,000	$O(N)$
$X_L Y_L 10^8$	5,819,263,900,000,000	$O(N)$
$X_L Y_L 10^8 + D_3 10^4 + X_R Y_R$	5,820,464,730,934,047	$O(N)$

Figure 10.37 The divide-and-conquer algorithm in action

It is easy to see that now the recurrence equation satisfies

$$T(N) = 3T(N/2) + O(N)$$

and so we obtain $T(N) = O(N^{\log_2 3}) = O(N^{1.59})$. To complete the algorithm, we must have a base case, which can be solved without recursion.

When both numbers are one-digit, we can do the multiplication by table lookup. If one number has zero digits, then we return zero. In practice, if we were to use this algorithm, we would choose the base case to be that which is most convenient for the machine.

Although this algorithm has better asymptotic performance than the standard quadratic algorithm, it is rarely used, because for small N the overhead is significant, and for larger N there are even better algorithms. These algorithms also make extensive use of divide and conquer.

Matrix Multiplication

A fundamental numerical problem is the multiplication of two matrices. Figure 10.38 gives a simple $O(N^3)$ algorithm to compute $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are $N \times N$ matrices. The algorithm follows directly from the definition of matrix multiplication. To compute $C_{i,j}$, we compute the dot product of the ith row in $\mathbf{A}$ with the jth column in $\mathbf{B}$. As usual, arrays begin at index 0.

```
1   /**
2    * Standard matrix multiplication.
3    * Arrays start at 0.
4    * Assumes a and b are square.
5    */
6   matrix<int> operator*( const matrix<int> & a, const matrix<int> & b )
7   {
8       int n = a.numrows( );
9       matrix<int> c{ n, n };
10
11      for( int i = 0; i < n; ++i )    // Initialization
12          for( int j = 0; j < n; ++j )
13              c[ i ][ j ] = 0;
14
15      for( int i = 0; i < n; ++i )
16          for( int j = 0; j < n; ++j )
17              for( int k = 0; k < n; ++k )
18                  c[ i ][ j ] += a[ i ][ k ] * b[ k ][ j ];
19
20      return c;
21  }
```

Figure 10.38 Simple $O(N^3)$ matrix multiplication