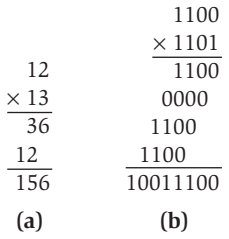

Figure 5.8 The elementary-school algorithm for multiplying two integers, in (a) decimal and (b) binary representation.

even as an algorithmic question. But, in fact, elementary schoolers are taught a concrete (and quite efficient) algorithm to multiply two n-digit numbers x and y. You first compute a "partial product" by multiplying each digit of y separately by x, and then you add up all the partial products. (Figure 5.8 should help you recall this algorithm. In elementary school we always see this done in base-10, but it works exactly the same way in base-2 as well.) Counting a single operation on a pair of bits as one primitive step in this computation, it takes $O(n)$ time to compute each partial product, and $O(n)$ time to combine it in with the running sum of all partial products so far. Since there are n partial products, this is a total running time of $O(n^2)$.

If you haven't thought about this much since elementary school, there's something initially striking about the prospect of improving on this algorithm. Aren't all those partial products "necessary" in some way? But, in fact, it is possible to improve on $O(n^2)$ time using a different, recursive way of performing the multiplication.

Designing the Algorithm

The improved algorithm is based on a more clever way to break up the product into partial sums. Let's assume we're in base-2 (it doesn't really matter), and start by writing x as $x_1 \cdot 2^{n/2} + x_0$. In other words, x_1 corresponds to the "high-order" $n/2$ bits, and x_0 corresponds to the "low-order" $n/2$ bits. Similarly, we write $y = y_1 \cdot 2^{n/2} + y_0$. Thus, we have

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0. \end{aligned} \tag{5.1}$$

Equation (5.1) reduces the problem of solving a single n-bit instance (multiplying the two n-bit numbers x and y) to the problem of solving four $n/2$-bit instances (computing the products x_1y_1, x_1y_0, x_0y_1, and x_0y_0). So we have a first candidate for a divide-and-conquer solution: recursively compute the results for these four $n/2$-bit instances, and then combine them using Equation

(5.1). The combining of the solution requires a constant number of additions of $O(n)$-bit numbers, so it takes time $O(n)$; thus, the running time $T(n)$ is bounded by the recurrence

$$T(n) \leq 4T(n/2) + cn$$

for a constant c. Is this good enough to give us a subquadratic running time?

We can work out the answer by observing that this is just the case $q = 4$ of the class of recurrences in (5.3). As we saw earlier in the chapter, the solution to this is $T(n) \leq O(n^{\log_2 q}) = O(n^2)$.

So, in fact, our divide-and-conquer algorithm with four-way branching was just a complicated way to get back to quadratic time! If we want to do better using a strategy that reduces the problem to instances on $n/2$ bits, we should try to get away with only *three* recursive calls. This will lead to the case $q = 3$ of (5.3), which we saw had the solution $T(n) \leq O(n^{\log_2 q}) = O(n^{1.59})$.

Recall that our goal is to compute the expression $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$ in Equation (5.1). It turns out there is a simple trick that lets us determine all of the terms in this expression using just three recursive calls. The trick is to consider the result of the single multiplication $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$. This has the four products above added together, at the cost of a single recursive multiplication. If we now also determine x_1y_1 and x_0y_0 by recursion, then we get the outermost terms explicitly, and we get the middle term by subtracting x_1y_1 and x_0y_0 away from $(x_1 + x_0)(y_1 + y_0)$.

Thus, in full, our algorithm is

```
Recursive-Multiply(x,y):
  Write x = x_1 · 2^{n/2} + x_0
        y = y_1 · 2^{n/2} + y_0
  Compute x_1 + x_0 and y_1 + y_0
  p = Recursive-Multiply(x_1 + x_0, y_1 + y_0)
  x_1y_1 = Recursive-Multiply(x_1, y_1)
  x_0y_0 = Recursive-Multiply(x_0, y_0)
  Return x_1y_1 · 2^n + (p − x_1y_1 − x_0y_0) · 2^{n/2} + x_0y_0
```

Analyzing the Algorithm

We can determine the running time of this algorithm as follows. Given two n-bit numbers, it performs a constant number of additions on $O(n)$-bit numbers, in addition to the three recursive calls. Ignoring for now the issue that $x_1 + x_0$ and $y_1 + y_0$ may have $n/2 + 1$ bits (rather than just $n/2$), which turns out not to affect the asymptotic results, each of these recursive calls is on an instance of size $n/2$. Thus, in place of our four-way branching recursion, we now have

a three-way branching one, with a running time that satisfies

$$T(n) \leq 3T(n/2) + cn$$

for a constant c.

This is the case $q = 3$ of (5.3) that we were aiming for. Using the solution to that recurrence from earlier in the chapter, we have

(5.13) *The running time of* `Recursive-Multiply` *on two n-bit factors is* $O(n^{\log_2 3}) = O(n^{1.59})$.

5.6 Convolutions and the Fast Fourier Transform

As a final topic in this chapter, we show how our basic recurrence from (5.1) is used in the design of the *Fast Fourier Transform*, an algorithm with a wide range of applications.

The Problem

Given two vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$, there are a number of common ways of combining them. For example, one can compute the sum, producing the vector $a + b = (a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1})$; or one can compute the inner product, producing the real number $a \cdot b = a_0b_0 + a_1b_1 + \cdots + a_{n-1}b_{n-1}$. (For reasons that will emerge shortly, it is useful to write vectors in this section with coordinates that are indexed starting from 0 rather than 1.)

A means of combining vectors that is very important in applications, even if it doesn't always show up in introductory linear algebra courses, is the *convolution* $a * b$. The convolution of two vectors of length n (as a and b are) is a vector with $2n - 1$ coordinates, where coordinate k is equal to

$$\sum_{\substack{(i,j):i+j=k \\ i,j<n}} a_i b_j.$$

In other words,

$$a * b = (a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_1b_1 + a_2b_0, \ldots, a_{n-2}b_{n-1} + a_{n-1}b_{n-2}, a_{n-1}b_{n-1}).$$

This definition is a bit hard to absorb when you first see it. Another way to think about the convolution is to picture an $n \times n$ table whose (i, j) entry is a_ib_j, like this,

$$\begin{array}{ccccc} a_0b_0 & a_0b_1 & \dots & a_0b_{n-2} & a_0b_{n-1} \\ a_1b_0 & a_1b_1 & \dots & a_1b_{n-2} & a_1b_{n-1} \\ a_2b_0 & a_2b_1 & \dots & a_2b_{n-2} & a_2b_{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n-1}b_0 & a_{n-1}b_1 & \dots & a_{n-1}b_{n-2} & a_{n-1}b_{n-1} \end{array}$$

and then to compute the coordinates in the convolution vector by summing along the diagonals.

It's worth mentioning that, unlike the vector sum and inner product, the convolution can be easily generalized to vectors of different lengths, $a = (a_0, a_1, \dots, a_{m-1})$ and $b = (b_0, b_1, \dots, b_{n-1})$. In this more general case, we define $a * b$ to be a vector with $m + n - 1$ coordinates, where coordinate k is equal to

$$\sum_{\substack{(i,j):i+j=k \\ i<m, j<n}} a_i b_j.$$

We can picture this using the table of products a_ib_j as before; the table is now rectangular, but we still compute coordinates by summing along the diagonals. (From here on, we'll drop explicit mention of the condition $i < m, j < n$ in the summations for convolutions, since it will be clear from the context that we only compute the sum over terms that are defined.)

It's not just the definition of a convolution that is a bit hard to absorb at first; the motivation for the definition can also initially be a bit elusive. What are the circumstances where you'd want to compute the convolution of two vectors? In fact, the convolution comes up in a surprisingly wide variety of different contexts. To illustrate this, we mention the following examples here.

- A first example (which also proves that the convolution is something that we all saw implicitly in high school) is polynomial multiplication. Any polynomial $A(x) = a_0 + a_1x + a_2x^2 + \cdots a_{m-1}x^{m-1}$ can be represented just as naturally using its vector of coefficients, $a = (a_0, a_1, \dots, a_{m-1})$. Now, given two polynomials $A(x) = a_0 + a_1x + a_2x^2 + \cdots a_{m-1}x^{m-1}$ and $B(x) = b_0 + b_1x + b_2x^2 + \cdots b_{n-1}x^{n-1}$, consider the polynomial $C(x) = A(x)B(x)$ that is equal to their product. In this polynomial $C(x)$, the coefficient on the x^k term is equal to

  $$c_k = \sum_{(i,j):i+j=k} a_i b_j.$$

  In other words, the coefficient vector c of $C(x)$ is the convolution of the coefficient vectors of $A(x)$ and $B(x)$.
- Arguably the most important application of convolutions in practice is for *signal processing*. This is a topic that could fill an entire course, so

we'll just give a simple example here to suggest one way in which the convolution arises.

Suppose we have a vector $a = (a_0, a_1, \ldots, a_{m-1})$ which represents a sequence of measurements, such as a temperature or a stock price, sampled at m consecutive points in time. Sequences like this are often very noisy due to measurement error or random fluctuations, and so a common operation is to "smooth" the measurements by averaging each value a_i with a weighted sum of its neighbors within k steps to the left and right in the sequence, the weights decaying quickly as one moves away from a_i. For example, in *Gaussian smoothing*, one replaces a_i with

$$a'_i = \frac{1}{Z} \sum_{j=i-k}^{i+k} a_j e^{-(j-i)^2},$$

for some "width" parameter k, and with Z chosen simply to normalize the weights in the average to add up to 1. (There are some issues with boundary conditions—what do we do when $i - k < 0$ or $i + k > m$?—but we could deal with these, for example, by discarding the first and last k entries from the smoothed signal, or by scaling them differently to make up for the missing terms.)

To see the connection with the convolution operation, we picture this smoothing operation as follows. We first define a "mask"

$$w = (w_{-k}, w_{-(k-1)}, \ldots, w_{-1}, w_0, w_1, \ldots, w_{k-1}, w_k)$$

consisting of the weights we want to use for averaging each point with its neighbors. (For example, $w = \frac{1}{Z}(e^{-k^2}, e^{-(k-1)^2}, \ldots, e^{-1}, 1, e^{-1}, \ldots, e^{-(k-1)^2}, e^{-k^2})$ in the Gaussian case above.) We then iteratively position this mask so it is centered at each possible point in the sequence a; and for each positioning, we compute the weighted average. In other words, we replace a_i with $a'_i = \sum_{s=-k}^{k} w_s a_{i+s}$.

This last expression is essentially a convolution; we just have to warp the notation a bit so that this becomes clear. Let's define $b = (b_0, b_1, \ldots, b_{2k})$ by setting $b_\ell = w_{k-\ell}$. Then it's not hard to check that with this definition we have the smoothed value

$$a'_i = \sum_{(j,\ell):j+\ell=i+k} a_j b_\ell.$$

In other words, the smoothed sequence is just the convolution of the original signal and the reverse of the mask (with some meaningless coordinates at the beginning and end).

- We mention one final application: the problem of combining histograms. Suppose we're studying a population of people, and we have the following two histograms: One shows the annual income of all the men in the population, and one shows the annual income of all the women. We'd now like to produce a new histogram, showing for each k the number of *pairs* (M, W) for which man M and woman W have a combined income of k.

  This is precisely a convolution. We can write the first histogram as a vector $a = (a_0, \ldots, a_{m-1})$, to indicate that there are a_i men with annual income equal to i. We can similarly write the second histogram as a vector $b = (b_0, \ldots, b_{n-1})$. Now, let c_k denote the number of pairs (m, w) with combined income k; this is the number of ways of choosing a man with income a_i and a woman with income b_j, for any pair (i, j) where $i + j = k$. In other words,

$$c_k = \sum_{(i,j):i+j=k} a_i b_j.$$

  so the combined histogram $c = (c_0, \ldots, c_{m+n-2})$ is simply the convolution of a and b.

  (Using terminology from probability that we will develop in Chapter 13, one can view this example as showing how convolution is the underlying means for computing the distribution of the sum of two independent random variables.)

Computing the Convolution Having now motivated the notion of convolution, let's discuss the problem of computing it efficiently. For simplicity, we will consider the case of equal length vectors (i.e., $m = n$), although everything we say carries over directly to the case of vectors of unequal lengths.

Computing the convolution is a more subtle question than it may first appear. The definition of convolution, after all, gives us a perfectly valid way to compute it: for each k, we just calculate the sum

$$\sum_{(i,j):i+j=k} a_i b_j$$

and use this as the value of the k^{th} coordinate. The trouble is that this direct way of computing the convolution involves calculating the product $a_i b_j$ for every pair (i, j) (in the process of distributing over the sums in the different terms) and this is $\Theta(n^2)$ arithmetic operations. Spending $O(n^2)$ time on computing the convolution seems natural, as the definition involves $O(n^2)$ multiplications $a_i b_j$. However, it's not inherently clear that we have to spend quadratic time to compute a convolution, since the input and output both only have size $O(n)$.