```
Else
   Return (r*_0, r*_1)
Endif
```

Analyzing the Algorithm

We first prove that the algorithm produces a correct answer, using the facts we've established in the process of designing it.

(5.11) *The algorithm correctly outputs a closest pair of points in P.*

Proof. As we've noted, all the components of the proof have already been worked out, so here we just summarize how they fit together.

We prove the correctness by induction on the size of P, the case of $|P| \leq 3$ being clear. For a given P, the closest pair in the recursive calls is computed correctly by induction. By (5.10) and (5.9), the remainder of the algorithm correctly determines whether any pair of points in S is at distance less than δ, and if so returns the closest such pair. Now the closest pair in P either has both elements in one of Q or R, or it has one element in each. In the former case, the closest pair is correctly found by the recursive call; in the latter case, this pair is at distance less than δ, and it is correctly found by the remainder of the algorithm. ■

We now bound the running time as well, using (5.2).

(5.12) *The running time of the algorithm is $O(n \log n)$.*

Proof. The initial sorting of P by x- and y-coordinate takes time $O(n \log n)$. The running time of the remainder of the algorithm satisfies the recurrence (5.1), and hence is $O(n \log n)$ by (5.2). ■

5.5 Integer Multiplication

We now discuss a different application of divide and conquer, in which the "default" quadratic algorithm is improved by means of a different recurrence. The analysis of the faster algorithm will exploit one of the recurrences considered in Section 5.2, in which more than two recursive calls are spawned at each level.

The Problem

The problem we consider is an extremely basic one: the multiplication of two integers. In a sense, this problem is so basic that one may not initially think of it

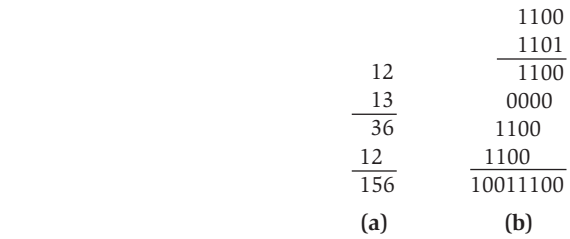

Figure 5.8 The elementary-school algorithm for multiplying two integers, in (a) decimal and (b) binary representation.

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,