```
Once one list is empty, append the remainder of the other list
    to the output
Return Count and the merged list
```

The running time of Merge-and-Count can be bounded by the analogue of the argument we used for the original merging algorithm at the heart of Mergesort: each iteration of the While loop takes constant time, and in each iteration we add some element to the output that will never be seen again. Thus the number of iterations can be at most the sum of the initial lengths of A and B, and so the total running time is $O(n)$.

We use this Merge-and-Count routine in a recursive procedure that simultaneously sorts and counts the number of inversions in a list L.

```
Sort-and-Count(L)
  If the list has one element then
      there are no inversions
  Else
      Divide the list into two halves:
         A contains the first ⌈n/2⌉ elements
         B contains the remaining ⌊n/2⌋ elements
      (r_A, A) = Sort-and-Count(A)
      (r_B, B) = Sort-and-Count(B)
      (r, L) = Merge-and-Count(A, B)
   Endif
   Return r = r_A + r_B + r, and the sorted list L
```

Since our Merge-and-Count procedure takes $O(n)$ time, the running time $T(n)$ of the full Sort-and-Count procedure satisfies the recurrence (5.1). By (5.2), we have

(5.7) *The* Sort-and-Count *algorithm correctly sorts the input list and counts the number of inversions; it runs in $O(n \log n)$ time for a list with n elements.*

5.4 Finding the Closest Pair of Points

We now describe another problem that can be solved by an algorithm in the style we've been discussing; but finding the right way to "merge" the solutions to the two subproblems it generates requires quite a bit of ingenuity.

The Problem

The problem we consider is very simple to state: Given n points in the plane, find the pair that is closest together.

The problem was considered by M. I. Shamos and D. Hoey in the early 1970s, as part of their project to work out efficient algorithms for basic computational primitives in geometry. These algorithms formed the foundations of the then-fledgling field of *computational geometry*, and they have found their way into areas such as graphics, computer vision, geographic information systems, and molecular modeling. And although the closest-pair problem is one of the most natural algorithmic problems in geometry, it is surprisingly hard to find an efficient algorithm for it. It is immediately clear that there is an $O(n^2)$ solution—compute the distance between each pair of points and take the minimum—and so Shamos and Hoey asked whether an algorithm asymptotically faster than quadratic could be found. It took quite a long time before they resolved this question, and the $O(n \log n)$ algorithm we give below is essentially the one they discovered. In fact, when we return to this problem in Chapter 13, we will see that it is possible to further improve the running time to $O(n)$ using randomization.

Designing the Algorithm

We begin with a bit of notation. Let us denote the set of points by $P = \{p_1, \ldots, p_n\}$, where p_i has coordinates (x_i, y_i); and for two points $p_i, p_j \in P$, we use $d(p_i, p_j)$ to denote the standard Euclidean distance between them. Our goal is to find a pair of points p_i, p_j that minimizes $d(p_i, p_j)$.

We will assume that no two points in P have the same x-coordinate or the same y-coordinate. This makes the discussion cleaner; and it's easy to eliminate this assumption either by initially applying a rotation to the points that makes it true, or by slightly extending the algorithm we develop here.

It's instructive to consider the one-dimensional version of this problem for a minute, since it is much simpler and the contrasts are revealing. How would we find the closest pair of points on a line? We'd first sort them, in $O(n \log n)$ time, and then we'd walk through the sorted list, computing the distance from each point to the one that comes after it. It is easy to see that one of these distances must be the minimum one.

In two dimensions, we could try sorting the points by their y-coordinate (or x-coordinate) and hoping that the two closest points were near one another in the order of this sorted list. But it is easy to construct examples in which they are very far apart, preventing us from adapting our one-dimensional approach.

Instead, our plan will be to apply the style of divide and conquer used in Mergesort: we find the closest pair among the points in the "left half" of

P and the closest pair among the points in the "right half" of P; and then we use this information to get the overall solution in linear time. If we develop an algorithm with this structure, then the solution of our basic recurrence from (5.1) will give us an $O(n \log n)$ running time.

It is the last, "combining" phase of the algorithm that's tricky: the distances that have not been considered by either of our recursive calls are precisely those that occur between a point in the left half and a point in the right half; there are $\Omega(n^2)$ such distances, yet we need to find the smallest one in $O(n)$ time after the recursive calls return. If we can do this, our solution will be complete: it will be the smallest of the values computed in the recursive calls and this minimum "left-to-right" distance.

Setting Up the Recursion Let's get a few easy things out of the way first. It will be very useful if every recursive call, on a set $P' \subseteq P$, begins with two lists: a list P'_x in which all the points in P' have been sorted by increasing x-coordinate, and a list P'_y in which all the points in P' have been sorted by increasing y-coordinate. We can ensure that this remains true throughout the algorithm as follows.

First, before any of the recursion begins, we sort all the points in P by x-coordinate and again by y-coordinate, producing lists P_x and P_y. Attached to each entry in each list is a record of the position of that point in both lists.

The first level of recursion will work as follows, with all further levels working in a completely analogous way. We define Q to be the set of points in the first $\lceil n/2 \rceil$ positions of the list P_x (the "left half") and R to be the set of points in the final $\lfloor n/2 \rfloor$ positions of the list P_x (the "right half"). See Figure 5.6. By a single pass through each of P_x and P_y, in $O(n)$ time, we can create the

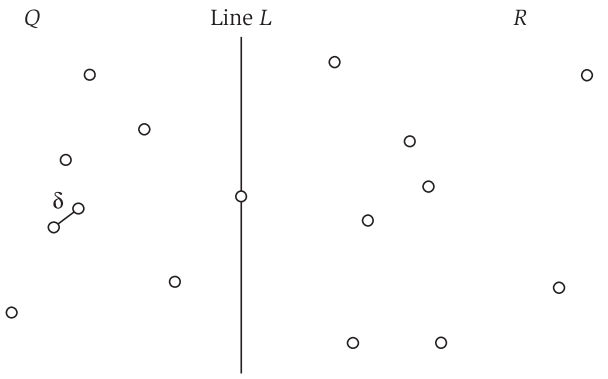

Figure 5.6 The first level of recursion: The point set P is divided evenly into Q and R by the line L, and the closest pair is found on each side recursively.

following four lists: Q_x, consisting of the points in Q sorted by increasing x-coordinate; Q_y, consisting of the points in Q sorted by increasing y-coordinate; and analogous lists R_x and R_y. For each entry of each of these lists, as before, we record the position of the point in both lists it belongs to.

We now recursively determine a closest pair of points in Q (with access to the lists Q_x and Q_y). Suppose that q_0^* and q_1^* are (correctly) returned as a closest pair of points in Q. Similarly, we determine a closest pair of points in R, obtaining r_0^* and r_1^*.

Combining the Solutions The general machinery of divide and conquer has gotten us this far, without our really having delved into the structure of the closest-pair problem. But it still leaves us with the problem that we saw looming originally: How do we use the solutions to the two subproblems as part of a linear-time "combining" operation?

Let δ be the minimum of $d(q_0^*,q_1^*)$ and $d(r_0^*,r_1^*)$. The real question is: Are there points $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$? If not, then we have already found the closest pair in one of our recursive calls. But if there are, then the closest such q and r form the closest pair in P.

Let x^* denote the x-coordinate of the rightmost point in Q, and let L denote the vertical line described by the equation $x = x^*$. This line L "separates" Q from R. Here is a simple fact.

(5.8) *If there exists $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$, then each of q and r lies within a distance δ of L.*

Proof. Suppose such q and r exist; we write $q = (q_x, q_y)$ and $r = (r_x, r_y)$. By the definition of x^*, we know that $q_x \le x^* \le r_x$. Then we have

$$x^* - q_x \le r_x - q_x \le d(q,r) < \delta$$

and

$$r_x - x^* \le r_x - q_x \le d(q,r) < \delta,$$

so each of q and r has an x-coordinate within δ of x^* and hence lies within distance δ of the line L. ■

So if we want to find a close q and r, we can restrict our search to the narrow band consisting only of points in P within δ of L. Let $S \subseteq P$ denote this set, and let S_y denote the list consisting of the points in S sorted by increasing y-coordinate. By a single pass through the list P_y, we can construct S_y in $O(n)$ time.

We can restate (5.8) as follows, in terms of the set S.

(5.9) *There exist $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$ if and only if there exist $s, s' \in S$ for which $d(s, s') < \delta$.*

It's worth noticing at this point that S might in fact be the whole set P, in which case (5.8) and (5.9) really seem to buy us nothing. But this is actually far from true, as the following amazing fact shows.

(5.10) *If $s, s' \in S$ have the property that $d(s, s') < \delta$, then s and s' are within 15 positions of each other in the sorted list S_y.*

Proof. Consider the subset Z of the plane consisting of all points within distance δ of L. We partition Z into *boxes*: squares with horizontal and vertical sides of length $\delta/2$. One *row* of Z will consist of four boxes whose horizontal sides have the same y-coordinates. This collection of boxes is depicted in Figure 5.7.

Suppose two points of S lie in the same box. Since all points in this box lie on the same side of L, these two points either both belong to Q or both belong to R. But any two points in the same box are within distance $\delta \cdot \sqrt{2}/2 < \delta$, which contradicts our definition of δ as the minimum distance between any pair of points in Q or in R. Thus each box contains at most one point of S.

Now suppose that $s, s' \in S$ have the property that $d(s, s') < \delta$, and that they are at least 16 positions apart in S_y. Assume without loss of generality that s has the smaller y-coordinate. Then, since there can be at most one point per box, there are at least three rows of Z lying between s and s'. But any two points in Z separated by at least three rows must be a distance of at least $3\delta/2$ apart—a contradiction. ■

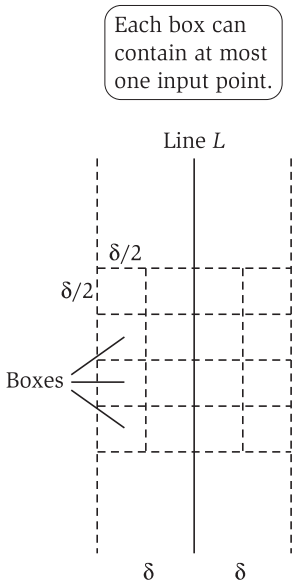

Figure 5.7 The portion of the plane close to the dividing line L, as analyzed in the proof of (5.10).

We note that the value of 15 can be reduced; but for our purposes at the moment, the important thing is that it is an absolute constant.

In view of (5.10), we can conclude the algorithm as follows. We make one pass through S_y, and for each $s \in S_y$, we compute its distance to each of the next 15 points in S_y. Statement (5.10) implies that in doing so, we will have computed the distance of each pair of points in S (if any) that are at distance less than δ from each other. So having done this, we can compare the smallest such distance to δ, and we can report one of two things: (i) the closest pair of points in S, if their distance is less than δ; or (ii) the (correct) conclusion that no pairs of points in S are within δ of each other. In case (i), this pair is the closest pair in P; in case (ii), the closest pair found by our recursive calls is the closest pair in P.

Note the resemblance between this procedure and the algorithm we rejected at the very beginning, which tried to make one pass through P in order

of y-coordinate. The reason such an approach works now is due to the extra knowledge (the value of δ) we've gained from the recursive calls, and the special structure of the set S.

This concludes the description of the "combining" part of the algorithm, since by (5.9) we have now determined whether the minimum distance between a point in Q and a point in R is less than δ, and if so, we have found the closest such pair.

A complete description of the algorithm and its proof of correctness are implicitly contained in the discussion so far, but for the sake of concreteness, we now summarize both.

Summary of the Algorithm A high-level description of the algorithm is the following, using the notation we have developed above.

```
Closest-Pair(P)
  Construct P_x and P_y  (O(n log n) time)
  (p_0*, p_1*) = Closest-Pair-Rec(P_x,P_y)

Closest-Pair-Rec(P_x, P_y)
  If |P| ≤ 3 then
    find closest pair by measuring all pairwise distances
  Endif

  Construct Q_x, Q_y, R_x, R_y (O(n) time)
  (q_0*,q_1*) = Closest-Pair-Rec(Q_x, Q_y)
  (r_0*,r_1*) = Closest-Pair-Rec(R_x, R_y)

  δ = min(d(q_0*,q_1*), d(r_0*,r_1*))
  x* = maximum x-coordinate of a point in set Q
  L = {(x,y) : x = x*}
  S = points in P within distance δ of L.

  Construct S_y (O(n) time)
  For each point s ∈ S_y, compute distance from s
     to each of next 15 points in S_y
     Let s, s' be pair achieving minimum of these distances
     (O(n) time)

  If d(s,s') < δ then
     Return (s,s')
  Else if d(q_0*,q_1*) < d(r_0*,r_1*) then
     Return (q_0*,q_1*)
```

```
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

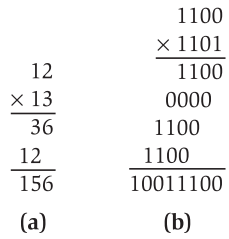

Figure 5.8 The elementary-school algorithm for multiplying two integers, in (a) decimal and (b) binary representation.

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,