

**Figure 33.10** The definition of a star-shaped polygon, for use in Exercise 33.3-4. (a) A star-shaped polygon. The segment from point  $p$  to any point  $q$  on the boundary intersects the boundary only at  $q$ . (b) A non-star-shaped polygon. The shaded region on the left is the shadow of  $q$ , and the shaded region on the right is the shadow of  $q'$ . Since these regions are disjoint, the kernel is empty.

star-shaped polygon  $P$  specified by its vertices in counterclockwise order, show how to compute  $\text{CH}(P)$  in  $O(n)$  time.

### 33.3-5

In the *on-line convex-hull problem*, we are given the set  $Q$  of  $n$  points one point at a time. After receiving each point, we compute the convex hull of the points seen so far. Obviously, we could run Graham's scan once for each point, with a total running time of  $O(n^2 \lg n)$ . Show how to solve the on-line convex-hull problem in a total of  $O(n^2)$  time.

### 33.3-6 ★

Show how to implement the incremental method for computing the convex hull of  $n$  points so that it runs in  $O(n \lg n)$  time.

---

## 33.4 Finding the closest pair of points

We now consider the problem of finding the closest pair of points in a set  $Q$  of  $n \geq 2$  points. “Closest” refers to the usual euclidean distance: the distance between points  $p_1 = (x_1, y_1)$  and  $p_2 = (x_2, y_2)$  is  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ . Two points in set  $Q$  may be coincident, in which case the distance between them is zero. This problem has applications in, for example, traffic-control systems. A system for controlling air or sea traffic might need to identify the two closest vehicles in order to detect potential collisions.

A brute-force closest-pair algorithm simply looks at all the  $\binom{n}{2} = \Theta(n^2)$  pairs of points. In this section, we shall describe a divide-and-conquer algorithm for

this problem, whose running time is described by the familiar recurrence  $T(n) = 2T(n/2) + O(n)$ . Thus, this algorithm uses only  $O(n \lg n)$  time.

### The divide-and-conquer algorithm

Each recursive invocation of the algorithm takes as input a subset  $P \subseteq Q$  and arrays  $X$  and  $Y$ , each of which contains all the points of the input subset  $P$ . The points in array  $X$  are sorted so that their  $x$ -coordinates are monotonically increasing. Similarly, array  $Y$  is sorted by monotonically increasing  $y$ -coordinate. Note that in order to attain the  $O(n \lg n)$  time bound, we cannot afford to sort in each recursive call; if we did, the recurrence for the running time would be  $T(n) = 2T(n/2) + O(n \lg n)$ , whose solution is  $T(n) = O(n \lg^2 n)$ . (Use the version of the master method given in Exercise 4.6-2.) We shall see a little later how to use “presorting” to maintain this sorted property without actually sorting in each recursive call.

A given recursive invocation with inputs  $P$ ,  $X$ , and  $Y$  first checks whether  $|P| \leq 3$ . If so, the invocation simply performs the brute-force method described above: try all  $\binom{|P|}{2}$  pairs of points and return the closest pair. If  $|P| > 3$ , the recursive invocation carries out the divide-and-conquer paradigm as follows.

**Divide:** Find a vertical line  $l$  that bisects the point set  $P$  into two sets  $P_L$  and  $P_R$  such that  $|P_L| = \lceil |P|/2 \rceil$ ,  $|P_R| = \lfloor |P|/2 \rfloor$ , all points in  $P_L$  are on or to the left of line  $l$ , and all points in  $P_R$  are on or to the right of  $l$ . Divide the array  $X$  into arrays  $X_L$  and  $X_R$ , which contain the points of  $P_L$  and  $P_R$  respectively, sorted by monotonically increasing  $x$ -coordinate. Similarly, divide the array  $Y$  into arrays  $Y_L$  and  $Y_R$ , which contain the points of  $P_L$  and  $P_R$  respectively, sorted by monotonically increasing  $y$ -coordinate.

**Conquer:** Having divided  $P$  into  $P_L$  and  $P_R$ , make two recursive calls, one to find the closest pair of points in  $P_L$  and the other to find the closest pair of points in  $P_R$ . The inputs to the first call are the subset  $P_L$  and arrays  $X_L$  and  $Y_L$ ; the second call receives the inputs  $P_R$ ,  $X_R$ , and  $Y_R$ . Let the closest-pair distances returned for  $P_L$  and  $P_R$  be  $\delta_L$  and  $\delta_R$ , respectively, and let  $\delta = \min(\delta_L, \delta_R)$ .

**Combine:** The closest pair is either the pair with distance  $\delta$  found by one of the recursive calls, or it is a pair of points with one point in  $P_L$  and the other in  $P_R$ . The algorithm determines whether there is a pair with one point in  $P_L$  and the other point in  $P_R$  and whose distance is less than  $\delta$ . Observe that if a pair of points has distance less than  $\delta$ , both points of the pair must be within  $\delta$  units of line  $l$ . Thus, as Figure 33.11(a) shows, they both must reside in the  $2\delta$ -wide vertical strip centered at line  $l$ . To find such a pair, if one exists, we do the following:

1. Create an array  $Y'$ , which is the array  $Y$  with all points not in the  $2\delta$ -wide vertical strip removed. The array  $Y'$  is sorted by  $y$ -coordinate, just as  $Y$  is.
2. For each point  $p$  in the array  $Y'$ , try to find points in  $Y'$  that are within  $\delta$  units of  $p$ . As we shall see shortly, only the 7 points in  $Y'$  that follow  $p$  need be considered. Compute the distance from  $p$  to each of these 7 points, and keep track of the closest-pair distance  $\delta'$  found over all pairs of points in  $Y'$ .
3. If  $\delta' < \delta$ , then the vertical strip does indeed contain a closer pair than the recursive calls found. Return this pair and its distance  $\delta'$ . Otherwise, return the closest pair and its distance  $\delta$  found by the recursive calls.

The above description omits some implementation details that are necessary to achieve the  $O(n \lg n)$  running time. After proving the correctness of the algorithm, we shall show how to implement the algorithm to achieve the desired time bound.

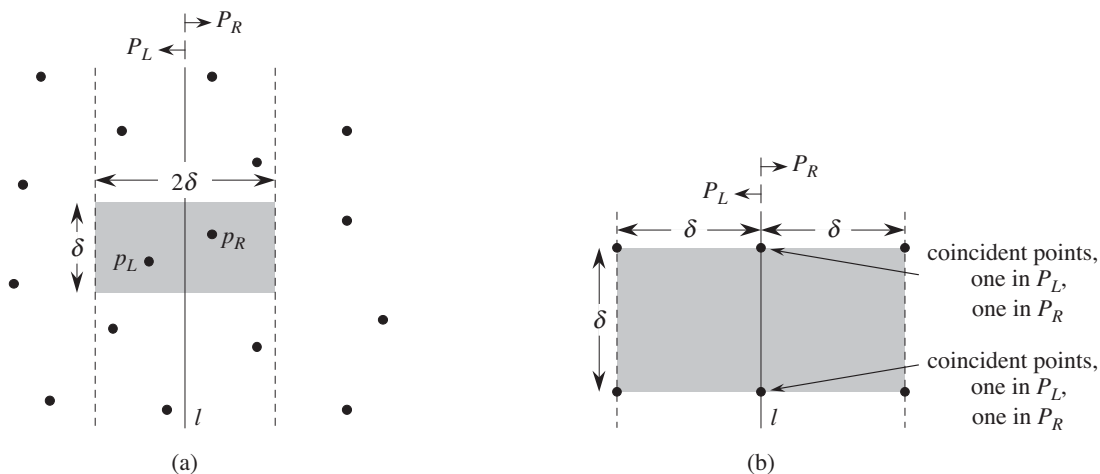
### Correctness

The correctness of this closest-pair algorithm is obvious, except for two aspects. First, by bottoming out the recursion when  $|P| \leq 3$ , we ensure that we never try to solve a subproblem consisting of only one point. The second aspect is that we need only check the 7 points following each point  $p$  in array  $Y'$ ; we shall now prove this property.

Suppose that at some level of the recursion, the closest pair of points is  $p_L \in P_L$  and  $p_R \in P_R$ . Thus, the distance  $\delta'$  between  $p_L$  and  $p_R$  is strictly less than  $\delta$ . Point  $p_L$  must be on or to the left of line  $l$  and less than  $\delta$  units away. Similarly,  $p_R$  is on or to the right of  $l$  and less than  $\delta$  units away. Moreover,  $p_L$  and  $p_R$  are within  $\delta$  units of each other vertically. Thus, as Figure 33.11(a) shows,  $p_L$  and  $p_R$  are within a  $\delta \times 2\delta$  rectangle centered at line  $l$ . (There may be other points within this rectangle as well.)

We next show that at most 8 points of  $P$  can reside within this  $\delta \times 2\delta$  rectangle. Consider the  $\delta \times \delta$  square forming the left half of this rectangle. Since all points within  $P_L$  are at least  $\delta$  units apart, at most 4 points can reside within this square; Figure 33.11(b) shows how. Similarly, at most 4 points in  $P_R$  can reside within the  $\delta \times \delta$  square forming the right half of the rectangle. Thus, at most 8 points of  $P$  can reside within the  $\delta \times 2\delta$  rectangle. (Note that since points on line  $l$  may be in either  $P_L$  or  $P_R$ , there may be up to 4 points on  $l$ . This limit is achieved if there are two pairs of coincident points such that each pair consists of one point from  $P_L$  and one point from  $P_R$ , one pair is at the intersection of  $l$  and the top of the rectangle, and the other pair is where  $l$  intersects the bottom of the rectangle.)

Having shown that at most 8 points of  $P$  can reside within the rectangle, we can easily see why we need to check only the 7 points following each point in the array  $Y'$ . Still assuming that the closest pair is  $p_L$  and  $p_R$ , let us assume without



**Figure 33.11** Key concepts in the proof that the closest-pair algorithm needs to check only 7 points following each point in the array  $Y'$ . **(a)** If  $p_L \in P_L$  and  $p_R \in P_R$  are less than  $\delta$  units apart, they must reside within a  $\delta \times 2\delta$  rectangle centered at line  $l$ . **(b)** How 4 points that are pairwise at least  $\delta$  units apart can all reside within a  $\delta \times \delta$  square. On the left are 4 points in  $P_L$ , and on the right are 4 points in  $P_R$ . The  $\delta \times 2\delta$  rectangle can contain 8 points if the points shown on line  $l$  are actually pairs of coincident points with one point in  $P_L$  and one in  $P_R$ .

loss of generality that  $p_L$  precedes  $p_R$  in array  $Y'$ . Then, even if  $p_L$  occurs as early as possible in  $Y'$  and  $p_R$  occurs as late as possible,  $p_R$  is in one of the 7 positions following  $p_L$ . Thus, we have shown the correctness of the closest-pair algorithm.

### Implementation and running time

As we have noted, our goal is to have the recurrence for the running time be  $T(n) = 2T(n/2) + O(n)$ , where  $T(n)$  is the running time for a set of  $n$  points. The main difficulty comes from ensuring that the arrays  $X_L$ ,  $X_R$ ,  $Y_L$ , and  $Y_R$ , which are passed to recursive calls, are sorted by the proper coordinate and also that the array  $Y'$  is sorted by  $y$ -coordinate. (Note that if the array  $X$  that is received by a recursive call is already sorted, then we can easily divide set  $P$  into  $P_L$  and  $P_R$  in linear time.)

The key observation is that in each call, we wish to form a sorted subset of a sorted array. For example, a particular invocation receives the subset  $P$  and the array  $Y$ , sorted by  $y$ -coordinate. Having partitioned  $P$  into  $P_L$  and  $P_R$ , it needs to form the arrays  $Y_L$  and  $Y_R$ , which are sorted by  $y$ -coordinate, in linear time. We can view the method as the opposite of the MERGE procedure from merge sort in

Section 2.3.1: we are splitting a sorted array into two sorted arrays. The following pseudocode gives the idea.

```

1  let  $Y_L[1 \dots Y.length]$  and  $Y_R[1 \dots Y.length]$  be new arrays
2   $Y_L.length = Y_R.length = 0$ 
3  for  $i = 1$  to  $Y.length$ 
4      if  $Y[i] \in P_L$ 
5           $Y_L.length = Y_L.length + 1$ 
6           $Y_L[Y_L.length] = Y[i]$ 
7      else  $Y_R.length = Y_R.length + 1$ 
8           $Y_R[Y_R.length] = Y[i]$ 

```

We simply examine the points in array  $Y$  in order. If a point  $Y[i]$  is in  $P_L$ , we append it to the end of array  $Y_L$ ; otherwise, we append it to the end of array  $Y_R$ . Similar pseudocode works for forming arrays  $X_L$ ,  $X_R$ , and  $Y'$ .

The only remaining question is how to get the points sorted in the first place. We *presort* them; that is, we sort them once and for all *before* the first recursive call. We pass these sorted arrays into the first recursive call, and from there we whittle them down through the recursive calls as necessary. Presorting adds an additional  $O(n \lg n)$  term to the running time, but now each step of the recursion takes linear time exclusive of the recursive calls. Thus, if we let  $T(n)$  be the running time of each recursive step and  $T'(n)$  be the running time of the entire algorithm, we get  $T'(n) = T(n) + O(n \lg n)$  and

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 3, \\ O(1) & \text{if } n \leq 3. \end{cases}$$

Thus,  $T(n) = O(n \lg n)$  and  $T'(n) = O(n \lg n)$ .

## Exercises

### 33.4-1

Professor Williams comes up with a scheme that allows the closest-pair algorithm to check only 5 points following each point in array  $Y'$ . The idea is always to place points on line  $l$  into set  $P_L$ . Then, there cannot be pairs of coincident points on line  $l$  with one point in  $P_L$  and one in  $P_R$ . Thus, at most 6 points can reside in the  $\delta \times 2\delta$  rectangle. What is the flaw in the professor's scheme?

### 33.4-2

Show that it actually suffices to check only the points in the 5 array positions following each point in the array  $Y'$ .