33 Computational Geometry

Computational geometry is the branch of computer science that studies algorithms for solving geometric problems. In modern engineering and mathematics, computational geometry has applications in such diverse fields as computer graphics, robotics, VLSI design, computer-aided design, molecular modeling, metallurgy, manufacturing, textile layout, forestry, and statistics. The input to a computational-geometry problem is typically a description of a set of geometric objects, such as a set of points, a set of line segments, or the vertices of a polygon in counterclockwise order. The output is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the convex hull (smallest enclosing convex polygon) of the set of points.

In this chapter, we look at a few computational-geometry algorithms in two dimensions, that is, in the plane. We represent each input object by a set of points $\{p_1, p_2, p_3, \ldots\}$, where each $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$. For example, we represent an n-vertex polygon P by a sequence $\langle p_0, p_1, p_2, \ldots, p_{n-1} \rangle$ of its vertices in order of their appearance on the boundary of P. Computational geometry can also apply to three dimensions, and even higher-dimensional spaces, but such problems and their solutions can be very difficult to visualize. Even in two dimensions, however, we can see a good sample of computational-geometry techniques.

Section 33.1 shows how to answer basic questions about line segments efficiently and accurately: whether one segment is clockwise or counterclockwise from another that shares an endpoint, which way we turn when traversing two adjoining line segments, and whether two line segments intersect. Section 33.2 presents a technique called "sweeping" that we use to develop an $O(n \lg n)$-time algorithm for determining whether a set of n line segments contains any intersections. Section 33.3 gives two "rotational-sweep" algorithms that compute the convex hull (smallest enclosing convex polygon) of a set of n points: Graham's scan, which runs in time $O(n \lg n)$, and Jarvis's march, which takes $O(nh)$ time, where h is the number of vertices of the convex hull. Finally, Section 33.4 gives

an $O(n \lg n)$-time divide-and-conquer algorithm for finding the closest pair of points in a set of n points in the plane.

33.1 Line-segment properties

Several of the computational-geometry algorithms in this chapter require answers to questions about the properties of line segments. A ***convex combination*** of two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is any point $p_3 = (x_3, y_3)$ such that for some α in the range $0 \le \alpha \le 1$, we have $x_3 = \alpha x_1 + (1 - \alpha)x_2$ and $y_3 = \alpha y_1 + (1 - \alpha) y_2$. We also write that $p_3 = \alpha p_1 + (1 - \alpha) p_2$. Intuitively, p_3 is any point that is on the line passing through p_1 and p_2 and is on or between p_1 and p_2 on the line. Given two distinct points p_1 and p_2, the ***line segment*** $\overline{p_1 p_2}$ is the set of convex combinations of p_1 and p_2. We call p_1 and p_2 the ***endpoints*** of segment $\overline{p_1 p_2}$. Sometimes the ordering of p_1 and p_2 matters, and we speak of the ***directed segment*** $\overrightarrow{p_1 p_2}$. If p_1 is the ***origin*** $(0, 0)$, then we can treat the directed segment $\overrightarrow{p_1 p_2}$ as the ***vector*** p_2.

In this section, we shall explore the following questions:

1. Given two directed segments $\overrightarrow{p_0 p_1}$ and $\overrightarrow{p_0 p_2}$, is $\overrightarrow{p_0 p_1}$ clockwise from $\overrightarrow{p_0 p_2}$ with respect to their common endpoint p_0?
2. Given two line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$, if we traverse $\overline{p_0 p_1}$ and then $\overline{p_1 p_2}$, do we make a left turn at point p_1?
3. Do line segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersect?

There are no restrictions on the given points.

We can answer each question in $O(1)$ time, which should come as no surprise since the input size of each question is $O(1)$. Moreover, our methods use only additions, subtractions, multiplications, and comparisons. We need neither division nor trigonometric functions, both of which can be computationally expensive and prone to problems with round-off error. For example, the "straightforward" method of determining whether two segments intersect—compute the line equation of the form $y = mx + b$ for each segment (m is the slope and b is the y-intercept), find the point of intersection of the lines, and check whether this point is on both segments—uses division to find the point of intersection. When the segments are nearly parallel, this method is very sensitive to the precision of the division operation on real computers. The method in this section, which avoids division, is much more accurate.

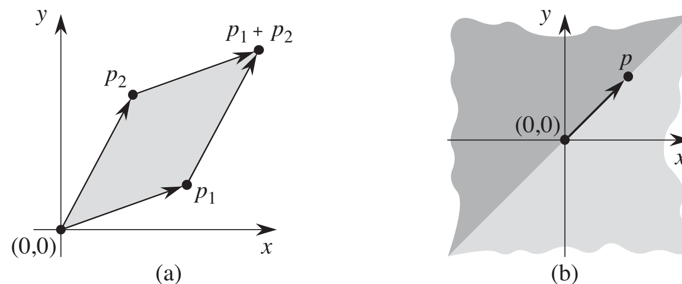

Figure 33.1 **(a)** The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. **(b)** The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.

Cross products

Computing cross products lies at the heart of our line-segment methods. Consider vectors p_1 and p_2, shown in Figure 33.1(a). We can interpret the ***cross product*** $p_1 \times p_2$ as the signed area of the parallelogram formed by the points $(0,0)$, p_1, p_2, and $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. An equivalent, but more useful, definition gives the cross product as the determinant of a matrix:[1]

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 . \end{aligned}$$

If $p_1 \times p_2$ is positive, then p_1 is clockwise from p_2 with respect to the origin $(0,0)$; if this cross product is negative, then p_1 is counterclockwise from p_2. (See Exercise 33.1-1.) Figure 33.1(b) shows the clockwise and counterclockwise regions relative to a vector p. A boundary condition arises if the cross product is 0; in this case, the vectors are ***colinear***, pointing in either the same or opposite directions.

To determine whether a directed segment $\overrightarrow{p_0 p_1}$ is closer to a directed segment $\overrightarrow{p_0 p_2}$ in a clockwise direction or in a counterclockwise direction with respect to their common endpoint p_0, we simply translate to use p_0 as the origin. That is, we let $p_1 - p_0$ denote the vector $p'_1 = (x'_1, y'_1)$, where $x'_1 = x_1 - x_0$ and $y'_1 = y_1 - y_0$, and we define $p_2 - p_0$ similarly. We then compute the cross product

---

[1] Actually, the cross product is a three-dimensional concept. It is a vector that is perpendicular to both p_1 and p_2 according to the "right-hand rule" and whose magnitude is $|x_1 y_2 - x_2 y_1|$. In this chapter, however, we find it convenient to treat the cross product simply as the value $x_1 y_2 - x_2 y_1$.

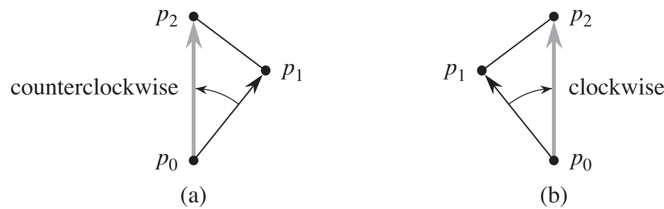

Figure 33.2 Using the cross product to determine how consecutive line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$ turn at point p_1. We check whether the directed segment $\overrightarrow{p_0 p_2}$ is clockwise or counterclockwise relative to the directed segment $\overrightarrow{p_0 p_1}$. **(a)** If counterclockwise, the points make a left turn. **(b)** If clockwise, they make a right turn.

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) .$$

If this cross product is positive, then $\overrightarrow{p_0 p_1}$ is clockwise from $\overrightarrow{p_0 p_2}$; if negative, it is counterclockwise.

Determining whether consecutive segments turn left or right

Our next question is whether two consecutive line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$ turn left or right at point p_1. Equivalently, we want a method to determine which way a given angle $\angle p_0 p_1 p_2$ turns. Cross products allow us to answer this question without computing the angle. As Figure 33.2 shows, we simply check whether directed segment $\overrightarrow{p_0 p_2}$ is clockwise or counterclockwise relative to directed segment $\overrightarrow{p_0 p_1}$. To do so, we compute the cross product $(p_2 - p_0) \times (p_1 - p_0)$. If the sign of this cross product is negative, then $\overrightarrow{p_0 p_2}$ is counterclockwise with respect to $\overrightarrow{p_0 p_1}$, and thus we make a left turn at p_1. A positive cross product indicates a clockwise orientation and a right turn. A cross product of 0 means that points p_0, p_1, and p_2 are colinear.

Determining whether two line segments intersect

To determine whether two line segments intersect, we check whether each segment straddles the line containing the other. A segment $\overline{p_1 p_2}$ ***straddles*** a line if point p_1 lies on one side of the line and point p_2 lies on the other side. A boundary case arises if p_1 or p_2 lies directly on the line. Two line segments intersect if and only if either (or both) of the following conditions holds:

1. Each segment straddles the line containing the other.
2. An endpoint of one segment lies on the other segment. (This condition comes from the boundary case.)

The following procedures implement this idea. SEGMENTS-INTERSECT returns TRUE if segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersect and FALSE if they do not. It calls the subroutines DIRECTION, which computes relative orientations using the cross-product method above, and ON-SEGMENT, which determines whether a point known to be colinear with a segment lies on that segment.

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
 1  d1 = DIRECTION(p3, p4, p1)
 2  d2 = DIRECTION(p3, p4, p2)
 3  d3 = DIRECTION(p1, p2, p3)
 4  d4 = DIRECTION(p1, p2, p4)
 5  if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
        ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
 6      return TRUE
 7  elseif d1 == 0 and ON-SEGMENT(p3, p4, p1)
 8      return TRUE
 9  elseif d2 == 0 and ON-SEGMENT(p3, p4, p2)
10      return TRUE
11  elseif d3 == 0 and ON-SEGMENT(p1, p2, p3)
12      return TRUE
13  elseif d4 == 0 and ON-SEGMENT(p1, p2, p4)
14      return TRUE
15  else return FALSE
```

```
DIRECTION(pi, pj, pk)
1  return (pk − pi) × (pj − pi)
```

```
ON-SEGMENT(pi, pj, pk)
1  if min(xi, xj) ≤ xk ≤ max(xi, xj) and min(yi, yj) ≤ yk ≤ max(yi, yj)
2      return TRUE
3  else return FALSE
```

SEGMENTS-INTERSECT works as follows. Lines 1–4 compute the relative orientation d_i of each endpoint p_i with respect to the other segment. If all the relative orientations are nonzero, then we can easily determine whether segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersect, as follows. Segment $\overline{p_1 p_2}$ straddles the line containing segment $\overline{p_3 p_4}$ if directed segments $\overrightarrow{p_3 p_1}$ and $\overrightarrow{p_3 p_2}$ have opposite orientations relative to $\overrightarrow{p_3 p_4}$. In this case, the signs of d_1 and d_2 differ. Similarly, $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$ if the signs of d_3 and d_4 differ. If the test of line 5 is true, then the segments straddle each other, and SEGMENTS-INTERSECT returns TRUE. Figure 33.3(a) shows this case. Otherwise, the segments do not straddle

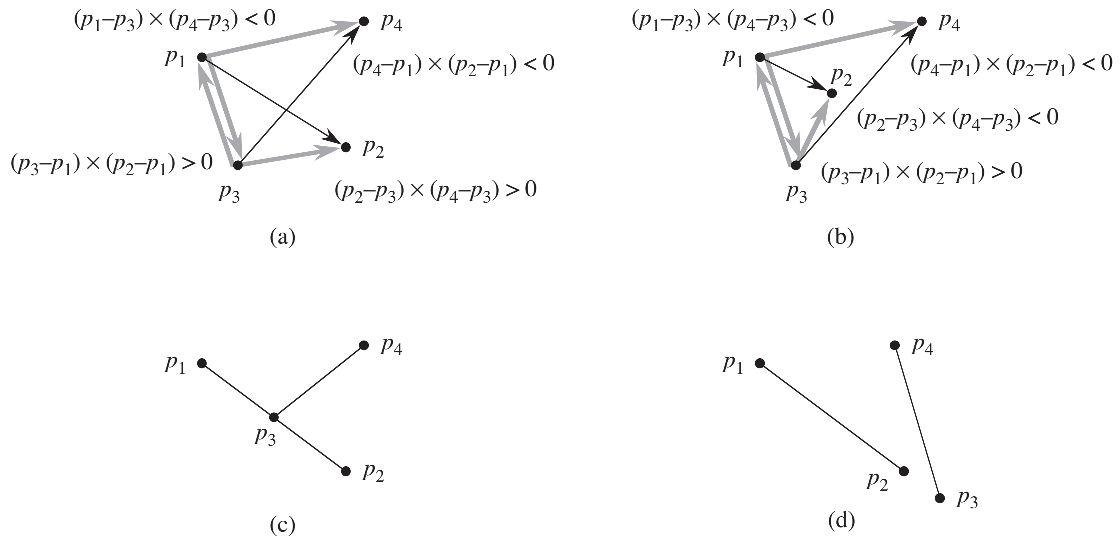

Figure 33.3 Cases in the procedure SEGMENTS-INTERSECT. **(a)** The segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ straddle each other's lines. Because $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, the signs of the cross products $(p_3 - p_1) \times (p_2 - p_1)$ and $(p_4 - p_1) \times (p_2 - p_1)$ differ. Because $\overline{p_1 p_2}$ straddles the line containing $\overline{p_3 p_4}$, the signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ differ. **(b)** Segment $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, but $\overline{p_1 p_2}$ does not straddle the line containing $\overline{p_3 p_4}$. The signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ are the same. **(c)** Point p_3 is colinear with $\overline{p_1 p_2}$ and is between p_1 and p_2. **(d)** Point p_3 is colinear with $\overline{p_1 p_2}$, but it is not between p_1 and p_2. The segments do not intersect.

each other's lines, although a boundary case may apply. If all the relative orientations are nonzero, no boundary case applies. All the tests against 0 in lines 7–13 then fail, and SEGMENTS-INTERSECT returns FALSE in line 15. Figure 33.3(b) shows this case.

A boundary case occurs if any relative orientation d_k is 0. Here, we know that p_k is colinear with the other segment. It is directly on the other segment if and only if it is between the endpoints of the other segment. The procedure ON-SEGMENT returns whether p_k is between the endpoints of segment $\overline{p_i p_j}$, which will be the other segment when called in lines 7–13; the procedure assumes that p_k is colinear with segment $\overline{p_i p_j}$. Figures 33.3(c) and (d) show cases with colinear points. In Figure 33.3(c), p_3 is on $\overline{p_1 p_2}$, and so SEGMENTS-INTERSECT returns TRUE in line 12. No endpoints are on other segments in Figure 33.3(d), and so SEGMENTS-INTERSECT returns FALSE in line 15.

Other applications of cross products

Later sections of this chapter introduce additional uses for cross products. In Section 33.3, we shall need to sort a set of points according to their polar angles with respect to a given origin. As Exercise 33.1-3 asks you to show, we can use cross products to perform the comparisons in the sorting procedure. In Section 33.2, we shall use red-black trees to maintain the vertical ordering of a set of line segments. Rather than keeping explicit key values which we compare to each other in the red-black tree code, we shall compute a cross-product to determine which of two segments that intersect a given vertical line is above the other.

Exercises

33.1-1
Prove that if $p_1 \times p_2$ is positive, then vector p_1 is clockwise from vector p_2 with respect to the origin $(0, 0)$ and that if this cross product is negative, then p_1 is counterclockwise from p_2.

33.1-2
Professor van Pelt proposes that only the x-dimension needs to be tested in line 1 of ON-SEGMENT. Show why the professor is wrong.

33.1-3
The ***polar angle*** of a point p_1 with respect to an origin point p_0 is the angle of the vector $p_1 - p_0$ in the usual polar coordinate system. For example, the polar angle of $(3, 5)$ with respect to $(2, 4)$ is the angle of the vector $(1, 1)$, which is 45 degrees or $\pi/4$ radians. The polar angle of $(3, 3)$ with respect to $(2, 4)$ is the angle of the vector $(1, -1)$, which is 315 degrees or $7\pi/4$ radians. Write pseudocode to sort a sequence $\langle p_1, p_2, \ldots, p_n \rangle$ of n points according to their polar angles with respect to a given origin point p_0. Your procedure should take $O(n \lg n)$ time and use cross products to compare angles.

33.1-4
Show how to determine in $O(n^2 \lg n)$ time whether any three points in a set of n points are colinear.

33.1-5
A ***polygon*** is a piecewise-linear, closed curve in the plane. That is, it is a curve ending on itself that is formed by a sequence of straight-line segments, called the ***sides*** of the polygon. A point joining two consecutive sides is a ***vertex*** of the polygon. If the polygon is ***simple***, as we shall generally assume, it does not cross itself. The set of points in the plane enclosed by a simple polygon forms the ***interior*** of

the polygon, the set of points on the polygon itself forms its ***boundary***, and the set of points surrounding the polygon forms its ***exterior***. A simple polygon is ***convex*** if, given any two points on its boundary or in its interior, all points on the line segment drawn between them are contained in the polygon's boundary or interior. A vertex of a convex polygon cannot be expressed as a convex combination of any two distinct points on the boundary or in the interior of the polygon.

Professor Amundsen proposes the following method to determine whether a sequence $\langle p_0, p_1, \ldots, p_{n-1} \rangle$ of n points forms the consecutive vertices of a convex polygon. Output "yes" if the set $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \ldots, n-1\}$, where subscript addition is performed modulo n, does not contain both left turns and right turns; otherwise, output "no." Show that although this method runs in linear time, it does not always produce the correct answer. Modify the professor's method so that it always produces the correct answer in linear time.

33.1-6

Given a point $p_0 = (x_0, y_0)$, the ***right horizontal ray*** from p_0 is the set of points $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ and } y_i = y_0\}$, that is, it is the set of points due right of p_0 along with p_0 itself. Show how to determine whether a given right horizontal ray from p_0 intersects a line segment $\overline{p_1 p_2}$ in $O(1)$ time by reducing the problem to that of determining whether two line segments intersect.

33.1-7

One way to determine whether a point p_0 is in the interior of a simple, but not necessarily convex, polygon P is to look at any ray from p_0 and check that the ray intersects the boundary of P an odd number of times but that p_0 itself is not on the boundary of P. Show how to compute in $\Theta(n)$ time whether a point p_0 is in the interior of an n-vertex polygon P. (*Hint:* Use Exercise 33.1-6. Make sure your algorithm is correct when the ray intersects the polygon boundary at a vertex and when the ray overlaps a side of the polygon.)

33.1-8

Show how to compute the area of an n-vertex simple, but not necessarily convex, polygon in $\Theta(n)$ time. (See Exercise 33.1-5 for definitions pertaining to polygons.)

33.2 Determining whether any pair of segments intersects

This section presents an algorithm for determining whether any two line segments in a set of segments intersect. The algorithm uses a technique known as "sweeping," which is common to many computational-geometry algorithms. Moreover, as

the exercises at the end of this section show, this algorithm, or simple variations of it, can help solve other computational-geometry problems.

The algorithm runs in $O(n \lg n)$ time, where n is the number of segments we are given. It determines only whether or not any intersection exists; it does not print all the intersections. (By Exercise 33.2-1, it takes $\Omega(n^2)$ time in the worst case to find *all* the intersections in a set of n line segments.)

In ***sweeping***, an imaginary vertical ***sweep line*** passes through the given set of geometric objects, usually from left to right. We treat the spatial dimension that the sweep line moves across, in this case the x-dimension, as a dimension of time. Sweeping provides a method for ordering geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of relationships among them. The line-segment-intersection algorithm in this section considers all the line-segment endpoints in left-to-right order and checks for an intersection each time it encounters an endpoint.

To describe and prove correct our algorithm for determining whether any two of n line segments intersect, we shall make two simplifying assumptions. First, we assume that no input segment is vertical. Second, we assume that no three input segments intersect at a single point. Exercises 33.2-8 and 33.2-9 ask you to show that the algorithm is robust enough that it needs only a slight modification to work even when these assumptions do not hold. Indeed, removing such simplifying assumptions and dealing with boundary conditions often present the most difficult challenges when programming computational-geometry algorithms and proving their correctness.

Ordering segments

Because we assume that there are no vertical segments, we know that any input segment intersecting a given vertical sweep line intersects it at a single point. Thus, we can order the segments that intersect a vertical sweep line according to the y-coordinates of the points of intersection.

To be more precise, consider two segments s_1 and s_2. We say that these segments are ***comparable*** at x if the vertical sweep line with x-coordinate x intersects both of them. We say that s_1 is ***above*** s_2 at x, written $s_1 \succsim_x s_2$, if s_1 and s_2 are comparable at x and the intersection of s_1 with the sweep line at x is higher than the intersection of s_2 with the same sweep line, or if s_1 and s_2 intersect at the sweep line. In Figure 33.4(a), for example, we have the relationships $a \succsim_r c$, $a \succsim_t b$, $b \succsim_t c$, $a \succsim_t c$, and $b \succsim_u c$. Segment d is not comparable with any other segment.

For any given x, the relation "$\succsim_x$" is a total preorder (see Section B.2) for all segments that intersect the sweep line at x. That is, the relation is transitive, and if segments s_1 and s_2 each intersect the sweep line at x, then either $s_1 \succsim_x s_2$ or $s_2 \succsim_x s_1$, or both (if s_1 and s_2 intersect at the sweep line). (The relation $\succsim_x$ is

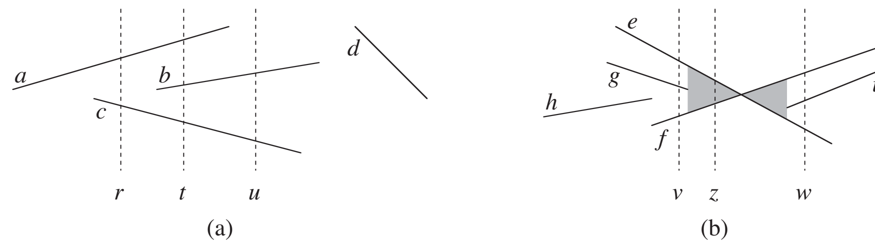

Figure 33.4 The ordering among line segments at various vertical sweep lines. **(a)** We have $a \succcurlyeq_r c$, $a \succcurlyeq_t b$, $b \succcurlyeq_t c$, $a \succcurlyeq_t c$, and $b \succcurlyeq_u c$. Segment d is comparable with no other segment shown. **(b)** When segments e and f intersect, they reverse their orders: we have $e \succcurlyeq_v f$ but $f \succcurlyeq_w e$. Any sweep line (such as z) that passes through the shaded region has e and f consecutive in the ordering given by the relation $\succcurlyeq_z$.

also reflexive, but neither symmetric nor antisymmetric.) The total preorder may differ for differing values of x, however, as segments enter and leave the ordering. A segment enters the ordering when its left endpoint is encountered by the sweep, and it leaves the ordering when its right endpoint is encountered.

What happens when the sweep line passes through the intersection of two segments? As Figure 33.4(b) shows, the segments reverse their positions in the total preorder. Sweep lines v and w are to the left and right, respectively, of the point of intersection of segments e and f, and we have $e \succcurlyeq_v f$ and $f \succcurlyeq_w e$. Note that because we assume that no three segments intersect at the same point, there must be some vertical sweep line x for which intersecting segments e and f are *consecutive* in the total preorder $\succcurlyeq_x$. Any sweep line that passes through the shaded region of Figure 33.4(b), such as z, has e and f consecutive in its total preorder.

Moving the sweep line

Sweeping algorithms typically manage two sets of data:

1. The ***sweep-line status*** gives the relationships among the objects that the sweep line intersects.
2. The ***event-point schedule*** is a sequence of points, called ***event points***, which we order from left to right according to their x-coordinates. As the sweep progresses from left to right, whenever the sweep line reaches the x-coordinate of an event point, the sweep halts, processes the event point, and then resumes. Changes to the sweep-line status occur only at event points.

For some algorithms (the algorithm asked for in Exercise 33.2-7, for example), the event-point schedule develops dynamically as the algorithm progresses. The algorithm at hand, however, determines all the event points before the sweep, based

solely on simple properties of the input data. In particular, each segment endpoint is an event point. We sort the segment endpoints by increasing x-coordinate and proceed from left to right. (If two or more endpoints are ***covertical***, i.e., they have the same x-coordinate, we break the tie by putting all the covertical left endpoints before the covertical right endpoints. Within a set of covertical left endpoints, we put those with lower y-coordinates first, and we do the same within a set of covertical right endpoints.) When we encounter a segment's left endpoint, we insert the segment into the sweep-line status, and we delete the segment from the sweep-line status upon encountering its right endpoint. Whenever two segments first become consecutive in the total preorder, we check whether they intersect.

The sweep-line status is a total preorder T, for which we require the following operations:

- INSERT(T, s): insert segment s into T.
- DELETE(T, s): delete segment s from T.
- ABOVE(T, s): return the segment immediately above segment s in T.
- BELOW(T, s): return the segment immediately below segment s in T.

It is possible for segments s_1 and s_2 to be mutually above each other in the total preorder T; this situation can occur if s_1 and s_2 intersect at the sweep line whose total preorder is given by T. In this case, the two segments may appear in either order in T.

If the input contains n segments, we can perform each of the operations INSERT, DELETE, ABOVE, and BELOW in $O(\lg n)$ time using red-black trees. Recall that the red-black-tree operations in Chapter 13 involve comparing keys. We can replace the key comparisons by comparisons that use cross products to determine the relative ordering of two segments (see Exercise 33.2-2).

Segment-intersection pseudocode

The following algorithm takes as input a set S of n line segments, returning the boolean value TRUE if any pair of segments in S intersects, and FALSE otherwise. A red-black tree maintains the total preorder T.

```
ANY-SEGMENTS-INTERSECT(S)
 1  T = ∅
 2  sort the endpoints of the segments in S from left to right,
        breaking ties by putting left endpoints before right endpoints
        and breaking further ties by putting points with lower
        y-coordinates first
 3  for each point p in the sorted list of endpoints
 4      if p is the left endpoint of a segment s
 5          INSERT(T, s)
 6          if (ABOVE(T, s) exists and intersects s)
                or (BELOW(T, s) exists and intersects s)
 7              return TRUE
 8      if p is the right endpoint of a segment s
 9          if both ABOVE(T, s) and BELOW(T, s) exist
                and ABOVE(T, s) intersects BELOW(T, s)
10              return TRUE
11          DELETE(T, s)
12  return FALSE
```

Figure 33.5 illustrates how the algorithm works. Line 1 initializes the total preorder to be empty. Line 2 determines the event-point schedule by sorting the $2n$ segment endpoints from left to right, breaking ties as described above. One way to perform line 2 is by lexicographically sorting the endpoints on (x, e, y), where x and y are the usual coordinates, $e = 0$ for a left endpoint, and $e = 1$ for a right endpoint.

Each iteration of the **for** loop of lines 3–11 processes one event point p. If p is the left endpoint of a segment s, line 5 adds s to the total preorder, and lines 6–7 return TRUE if s intersects either of the segments it is consecutive with in the total preorder defined by the sweep line passing through p. (A boundary condition occurs if p lies on another segment s'. In this case, we require only that s and s' be placed consecutively into T.) If p is the right endpoint of a segment s, then we need to delete s from the total preorder. But first, lines 9–10 return TRUE if there is an intersection between the segments surrounding s in the total preorder defined by the sweep line passing through p. If these segments do not intersect, line 11 deletes segment s from the total preorder. If the segments surrounding segment s intersect, they would have become consecutive after deleting s had the **return** statement in line 10 not prevented line 11 from executing. The correctness argument, which follows, will make it clear why it suffices to check the segments surrounding s. Finally, if we never find any intersections after having processed all $2n$ event points, line 12 returns FALSE.

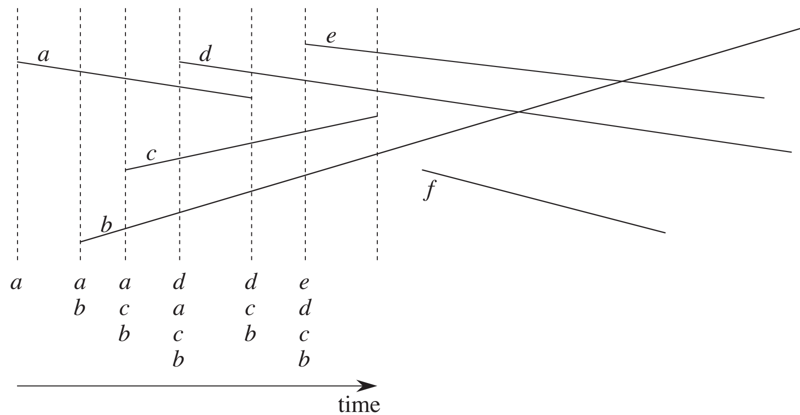

Figure 33.5 The execution of ANY-SEGMENTS-INTERSECT. Each dashed line is the sweep line at an event point. Except for the rightmost sweep line, the ordering of segment names below each sweep line corresponds to the total preorder T at the end of the **for** loop processing the corresponding event point. The rightmost sweep line occurs when processing the right endpoint of segment c; because segments d and b surround c and intersect each other, the procedure returns TRUE.

Correctness

To show that ANY-SEGMENTS-INTERSECT is correct, we will prove that the call ANY-SEGMENTS-INTERSECT(S) returns TRUE if and only if there is an intersection among the segments in S.

It is easy to see that ANY-SEGMENTS-INTERSECT returns TRUE (on lines 7 and 10) only if it finds an intersection between two of the input segments. Hence, if it returns TRUE, there is an intersection.

We also need to show the converse: that if there is an intersection, then ANY-SEGMENTS-INTERSECT returns TRUE. Let us suppose that there is at least one intersection. Let p be the leftmost intersection point, breaking ties by choosing the point with the lowest y-coordinate, and let a and b be the segments that intersect at p. Since no intersections occur to the left of p, the order given by T is correct at all points to the left of p. Because no three segments intersect at the same point, a and b become consecutive in the total preorder at some sweep line z.[2] Moreover, z is to the left of p or goes through p. Some segment endpoint q on sweep line z

[2]If we allow three segments to intersect at the same point, there may be an intervening segment c that intersects both a and b at point p. That is, we may have $a \succcurlyeq_w c$ and $c \succcurlyeq_w b$ for all sweep lines w to the left of p for which $a \succcurlyeq_w b$. Exercise 33.2-8 asks you to show that ANY-SEGMENTS-INTERSECT is correct even if three segments do intersect at the same point.

is the event point at which a and b become consecutive in the total preorder. If p is on sweep line z, then $q = p$. If p is not on sweep line z, then q is to the left of p. In either case, the order given by T is correct just before encountering q. (Here is where we use the lexicographic order in which the algorithm processes event points. Because p is the lowest of the leftmost intersection points, even if p is on sweep line z and some other intersection point p' is on z, event point $q = p$ is processed before the other intersection p' can interfere with the total preorder T. Moreover, even if p is the left endpoint of one segment, say a, and the right endpoint of the other segment, say b, because left endpoint events occur before right endpoint events, segment b is in T upon first encountering segment a.) Either event point q is processed by ANY-SEGMENTS-INTERSECT or it is not processed.

If q is processed by ANY-SEGMENTS-INTERSECT, only two possible actions may occur:

1. Either a or b is inserted into T, and the other segment is above or below it in the total preorder. Lines 4–7 detect this case.
2. Segments a and b are already in T, and a segment between them in the total preorder is deleted, making a and b become consecutive. Lines 8–11 detect this case.

In either case, we find the intersection p and ANY-SEGMENTS-INTERSECT returns TRUE.

If event point q is not processed by ANY-SEGMENTS-INTERSECT, the procedure must have returned before processing all event points. This situation could have occurred only if ANY-SEGMENTS-INTERSECT had already found an intersection and returned TRUE.

Thus, if there is an intersection, ANY-SEGMENTS-INTERSECT returns TRUE. As we have already seen, if ANY-SEGMENTS-INTERSECT returns TRUE, there is an intersection. Therefore, ANY-SEGMENTS-INTERSECT always returns a correct answer.

Running time

If set S contains n segments, then ANY-SEGMENTS-INTERSECT runs in time $O(n \lg n)$. Line 1 takes $O(1)$ time. Line 2 takes $O(n \lg n)$ time, using merge sort or heapsort. The **for** loop of lines 3–11 iterates at most once per event point, and so with $2n$ event points, the loop iterates at most $2n$ times. Each iteration takes $O(\lg n)$ time, since each red-black-tree operation takes $O(\lg n)$ time and, using the method of Section 33.1, each intersection test takes $O(1)$ time. The total time is thus $O(n \lg n)$.

Exercises

33.2-1
Show that a set of n line segments may contain $\Theta(n^2)$ intersections.

33.2-2
Given two segments a and b that are comparable at x, show how to determine in $O(1)$ time which of $a \succcurlyeq_x b$ or $b \succcurlyeq_x a$ holds. Assume that neither segment is vertical. (*Hint:* If a and b do not intersect, you can just use cross products. If a and b intersect—which you can of course determine using only cross products—you can still use only addition, subtraction, and multiplication, avoiding division. Of course, in the application of the $\succcurlyeq_x$ relation used here, if a and b intersect, we can just stop and declare that we have found an intersection.)

33.2-3
Professor Mason suggests that we modify ANY-SEGMENTS-INTERSECT so that instead of returning upon finding an intersection, it prints the segments that intersect and continues on to the next iteration of the **for** loop. The professor calls the resulting procedure PRINT-INTERSECTING-SEGMENTS and claims that it prints all intersections, from left to right, as they occur in the set of line segments. Professor Dixon disagrees, claiming that Professor Mason's idea is incorrect. Which professor is right? Will PRINT-INTERSECTING-SEGMENTS always find the leftmost intersection first? Will it always find all the intersections?

33.2-4
Give an $O(n \lg n)$-time algorithm to determine whether an n-vertex polygon is simple.

33.2-5
Give an $O(n \lg n)$-time algorithm to determine whether two simple polygons with a total of n vertices intersect.

33.2-6
A ***disk*** consists of a circle plus its interior and is represented by its center point and radius. Two disks intersect if they have any point in common. Give an $O(n \lg n)$-time algorithm to determine whether any two disks in a set of n intersect.

33.2-7
Given a set of n line segments containing a total of k intersections, show how to output all k intersections in $O((n + k) \lg n)$ time.

33.2-8
Argue that ANY-SEGMENTS-INTERSECT works correctly even if three or more segments intersect at the same point.

33.2-9
Show that ANY-SEGMENTS-INTERSECT works correctly in the presence of vertical segments if we treat the bottom endpoint of a vertical segment as if it were a left endpoint and the top endpoint as if it were a right endpoint. How does your answer to Exercise 33.2-2 change if we allow vertical segments?

33.3 Finding the convex hull

The ***convex hull*** of a set Q of points, denoted by CH(Q), is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior. (See Exercise 33.1-5 for a precise definition of a convex polygon.) We implicitly assume that all points in the set Q are unique and that Q contains at least three points which are not colinear. Intuitively, we can think of each point in Q as being a nail sticking out from a board. The convex hull is then the shape formed by a tight rubber band that surrounds all the nails. Figure 33.6 shows a set of points and its convex hull.

In this section, we shall present two algorithms that compute the convex hull of a set of n points. Both algorithms output the vertices of the convex hull in counterclockwise order. The first, known as Graham's scan, runs in $O(n \lg n)$ time. The second, called Jarvis's march, runs in $O(nh)$ time, where h is the number of vertices of the convex hull. As Figure 33.6 illustrates, every vertex of CH(Q) is a

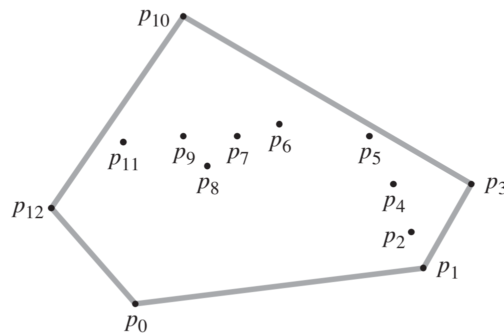

Figure 33.6 A set of points $Q = \{p_0, p_1, \ldots, p_{12}\}$ with its convex hull CH(Q) in gray.

point in Q. Both algorithms exploit this property, deciding which vertices in Q to keep as vertices of the convex hull and which vertices in Q to reject.

We can compute convex hulls in $O(n \lg n)$ time by any one of several methods. Both Graham's scan and Jarvis's march use a technique called "rotational sweep," processing vertices in the order of the polar angles they form with a reference vertex. Other methods include the following:

- In the ***incremental method***, we first sort the points from left to right, yielding a sequence $\langle p_1, p_2, \ldots, p_n \rangle$. At the ith stage, we update the convex hull of the $i-1$ leftmost points, $\mathrm{CH}(\{p_1, p_2, \ldots, p_{i-1}\})$, according to the ith point from the left, thus forming $\mathrm{CH}(\{p_1, p_2, \ldots, p_i\})$. Exercise 33.3-6 asks you how to implement this method to take a total of $O(n \lg n)$ time.
- In the ***divide-and-conquer method***, we divide the set of n points in $\Theta(n)$ time into two subsets, one containing the leftmost $\lceil n/2 \rceil$ points and one containing the rightmost $\lfloor n/2 \rfloor$ points, recursively compute the convex hulls of the subsets, and then, by means of a clever method, combine the hulls in $O(n)$ time. The running time is described by the familiar recurrence $T(n) = 2T(n/2) + O(n)$, and so the divide-and-conquer method runs in $O(n \lg n)$ time.
- The ***prune-and-search method*** is similar to the worst-case linear-time median algorithm of Section 9.3. With this method, we find the upper portion (or "upper chain") of the convex hull by repeatedly throwing out a constant fraction of the remaining points until only the upper chain of the convex hull remains. We then do the same for the lower chain. This method is asymptotically the fastest: if the convex hull contains h vertices, it runs in only $O(n \lg h)$ time.

Computing the convex hull of a set of points is an interesting problem in its own right. Moreover, algorithms for some other computational-geometry problems start by computing a convex hull. Consider, for example, the two-dimensional ***farthest-pair problem***: we are given a set of n points in the plane and wish to find the two points whose distance from each other is maximum. As Exercise 33.3-3 asks you to prove, these two points must be vertices of the convex hull. Although we won't prove it here, we can find the farthest pair of vertices of an n-vertex convex polygon in $O(n)$ time. Thus, by computing the convex hull of the n input points in $O(n \lg n)$ time and then finding the farthest pair of the resulting convex-polygon vertices, we can find the farthest pair of points in any set of n points in $O(n \lg n)$ time.

Graham's scan

Graham's scan solves the convex-hull problem by maintaining a stack S of candidate points. It pushes each point of the input set Q onto the stack one time,

and it eventually pops from the stack each point that is not a vertex of CH(Q). When the algorithm terminates, stack S contains exactly the vertices of CH(Q), in counterclockwise order of their appearance on the boundary.

The procedure GRAHAM-SCAN takes as input a set Q of points, where $|Q| \geq 3$. It calls the functions TOP(S), which returns the point on top of stack S without changing S, and NEXT-TO-TOP(S), which returns the point one entry below the top of stack S without changing S. As we shall prove in a moment, the stack S returned by GRAHAM-SCAN contains, from bottom to top, exactly the vertices of CH(Q) in counterclockwise order.

```
GRAHAM-SCAN(Q)
 1  let p0 be the point in Q with the minimum y-coordinate,
        or the leftmost such point in case of a tie
 2  let ⟨p1, p2, ..., pm⟩ be the remaining points in Q,
        sorted by polar angle in counterclockwise order around p0
        (if more than one point has the same angle, remove all but
        the one that is farthest from p0)
 3  let S be an empty stack
 4  PUSH(p0, S)
 5  PUSH(p1, S)
 6  PUSH(p2, S)
 7  for i = 3 to m
 8      while the angle formed by points NEXT-TO-TOP(S), TOP(S),
                and pi makes a nonleft turn
 9          POP(S)
10      PUSH(pi, S)
11  return S
```

Figure 33.7 illustrates the progress of GRAHAM-SCAN. Line 1 chooses point p_0 as the point with the lowest y-coordinate, picking the leftmost such point in case of a tie. Since there is no point in Q that is below p_0 and any other points with the same y-coordinate are to its right, p_0 must be a vertex of CH(Q). Line 2 sorts the remaining points of Q by polar angle relative to p_0, using the same method—comparing cross products—as in Exercise 33.1-3. If two or more points have the same polar angle relative to p_0, all but the farthest such point are convex combinations of p_0 and the farthest point, and so we remove them entirely from consideration. We let m denote the number of points other than p_0 that remain. The polar angle, measured in radians, of each point in Q relative to p_0 is in the half-open interval $[0, \pi)$. Since the points are sorted according to polar angles, they are sorted in counterclockwise order relative to p_0. We designate this sorted sequence of points by $\langle p_1, p_2, \ldots, p_m \rangle$. Note that points p_1 and p_m are vertices

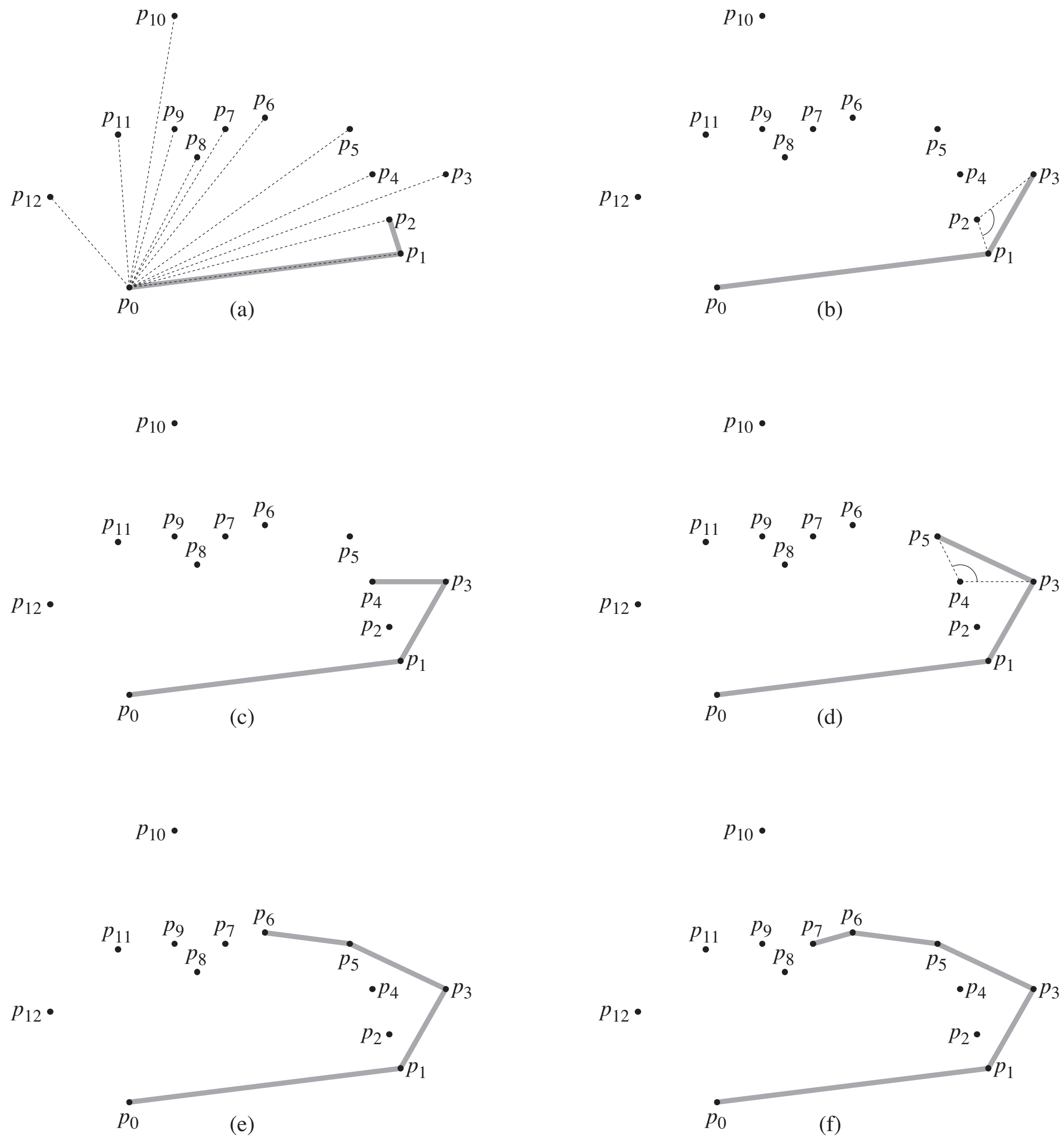

Figure 33.7 The execution of GRAHAM-SCAN on the set Q of Figure 33.6. The current convex hull contained in stack S is shown in gray at each step. **(a)** The sequence $\langle p_1, p_2, \ldots, p_{12} \rangle$ of points numbered in order of increasing polar angle relative to p_0, and the initial stack S containing p_0, p_1, and p_2. **(b)–(k)** Stack S after each iteration of the **for** loop of lines 7–10. Dashed lines show nonleft turns, which cause points to be popped from the stack. In part (h), for example, the right turn at angle $\angle p_7 p_8 p_9$ causes p_8 to be popped, and then the right turn at angle $\angle p_6 p_7 p_9$ causes p_7 to be popped.

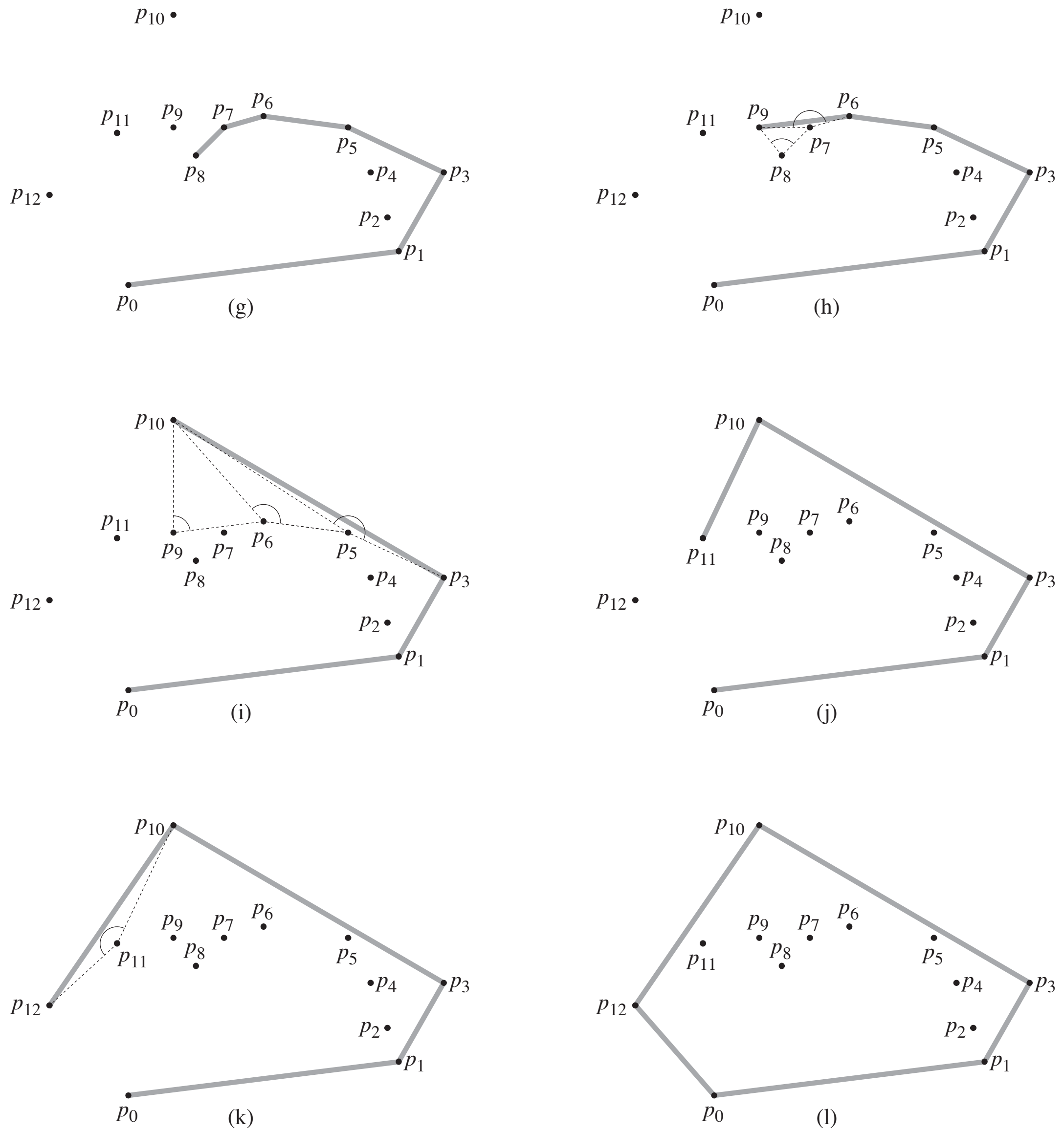

Figure 33.7, continued **(l)** The convex hull returned by the procedure, which matches that of Figure 33.6.

of CH(Q) (see Exercise 33.3-1). Figure 33.7(a) shows the points of Figure 33.6 sequentially numbered in order of increasing polar angle relative to p_0.

The remainder of the procedure uses the stack S. Lines 3–6 initialize the stack to contain, from bottom to top, the first three points p_0, p_1, and p_2. Figure 33.7(a) shows the initial stack S. The **for** loop of lines 7–10 iterates once for each point in the subsequence $\langle p_3, p_4, \ldots, p_m \rangle$. We shall see that after processing point p_i, stack S contains, from bottom to top, the vertices of CH($\{p_0, p_1, \ldots, p_i\}$) in counterclockwise order. The **while** loop of lines 8–9 removes points from the stack if we find them not to be vertices of the convex hull. When we traverse the convex hull counterclockwise, we should make a left turn at each vertex. Thus, each time the **while** loop finds a vertex at which we make a nonleft turn, we pop the vertex from the stack. (By checking for a nonleft turn, rather than just a right turn, this test precludes the possibility of a straight angle at a vertex of the resulting convex hull. We want no straight angles, since no vertex of a convex polygon may be a convex combination of other vertices of the polygon.) After we pop all vertices that have nonleft turns when heading toward point p_i, we push p_i onto the stack. Figures 33.7(b)–(k) show the state of the stack S after each iteration of the **for** loop. Finally, GRAHAM-SCAN returns the stack S in line 11. Figure 33.7(l) shows the corresponding convex hull.

The following theorem formally proves the correctness of GRAHAM-SCAN.

Theorem 33.1 (Correctness of Graham's scan)
If GRAHAM-SCAN executes on a set Q of points, where $|Q| \geq 3$, then at termination, the stack S consists of, from bottom to top, exactly the vertices of CH(Q) in counterclockwise order.

Proof After line 2, we have the sequence of points $\langle p_1, p_2, \ldots, p_m \rangle$. Let us define, for $i = 2, 3, \ldots, m$, the subset of points $Q_i = \{p_0, p_1, \ldots, p_i\}$. The points in $Q - Q_m$ are those that were removed because they had the same polar angle relative to p_0 as some point in Q_m; these points are not in CH(Q), and so CH(Q_m) $=$ CH(Q). Thus, it suffices to show that when GRAHAM-SCAN terminates, the stack S consists of the vertices of CH(Q_m) in counterclockwise order, when listed from bottom to top. Note that just as p_0, p_1, and p_m are vertices of CH(Q), the points p_0, p_1, and p_i are all vertices of CH(Q_i).

The proof uses the following loop invariant:

> At the start of each iteration of the **for** loop of lines 7–10, stack S consists of, from bottom to top, exactly the vertices of CH(Q_{i-1}) in counterclockwise order.

Initialization: The invariant holds the first time we execute line 7, since at that time, stack S consists of exactly the vertices of $Q_2 = Q_{i-1}$, and this set of three

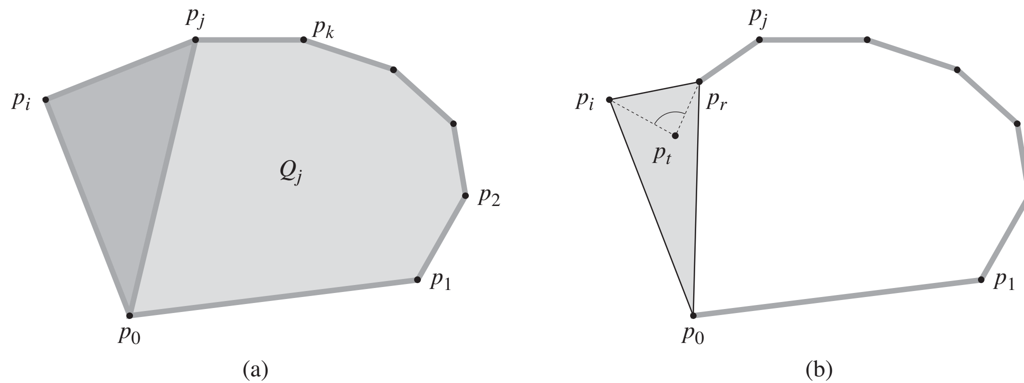

Figure 33.8 The proof of correctness of GRAHAM-SCAN. **(a)** Because p_i's polar angle relative to p_0 is greater than p_j's polar angle, and because the angle $\angle p_k p_j p_i$ makes a left turn, adding p_i to $\text{CH}(Q_j)$ gives exactly the vertices of $\text{CH}(Q_j \cup \{p_i\})$. **(b)** If the angle $\angle p_r p_t p_i$ makes a nonleft turn, then p_t is either in the interior of the triangle formed by p_0, p_r, and p_i or on a side of the triangle, which means that it cannot be a vertex of $\text{CH}(Q_i)$.

vertices forms its own convex hull. Moreover, they appear in counterclockwise order from bottom to top.

Maintenance: Entering an iteration of the **for** loop, the top point on stack S is p_{i-1}, which was pushed at the end of the previous iteration (or before the first iteration, when $i = 3$). Let p_j be the top point on S after executing the while loop of lines 8–9 but before line 10 pushes p_i, and let p_k be the point just below p_j on S. At the moment that p_j is the top point on S and we have not yet pushed p_i, stack S contains exactly the same points it contained after iteration j of the **for** loop. By the loop invariant, therefore, S contains exactly the vertices of $\text{CH}(Q_j)$ at that moment, and they appear in counterclockwise order from bottom to top.

Let us continue to focus on this moment just before pushing p_i. We know that p_i's polar angle relative to p_0 is greater than p_j's polar angle and that the angle $\angle p_k p_j p_i$ makes a left turn (otherwise we would have popped p_j). Therefore, because S contains exactly the vertices of $\text{CH}(Q_j)$, we see from Figure 33.8(a) that once we push p_i, stack S will contain exactly the vertices of $\text{CH}(Q_j \cup \{p_i\})$, still in counterclockwise order from bottom to top.

We now show that $\text{CH}(Q_j \cup \{p_i\})$ is the same set of points as $\text{CH}(Q_i)$. Consider any point p_t that was popped during iteration i of the **for** loop, and let p_r be the point just below p_t on stack S at the time p_t was popped (p_r might be p_j). The angle $\angle p_r p_t p_i$ makes a nonleft turn, and the polar angle of p_t relative to p_0 is greater than the polar angle of p_r. As Figure 33.8(b) shows, p_t must

be either in the interior of the triangle formed by p_0, p_r, and p_i or on a side of this triangle (but it is not a vertex of the triangle). Clearly, since p_t is within a triangle formed by three other points of Q_i, it cannot be a vertex of $\text{CH}(Q_i)$. Since p_t is not a vertex of $\text{CH}(Q_i)$, we have that

$$\text{CH}(Q_i - \{p_t\}) = \text{CH}(Q_i) . \tag{33.1}$$

Let P_i be the set of points that were popped during iteration i of the **for** loop. Since the equality (33.1) applies for all points in P_i, we can apply it repeatedly to show that $\text{CH}(Q_i - P_i) = \text{CH}(Q_i)$. But $Q_i - P_i = Q_j \cup \{p_i\}$, and so we conclude that $\text{CH}(Q_j \cup \{p_i\}) = \text{CH}(Q_i - P_i) = \text{CH}(Q_i)$.

We have shown that once we push p_i, stack S contains exactly the vertices of $\text{CH}(Q_i)$ in counterclockwise order from bottom to top. Incrementing i will then cause the loop invariant to hold for the next iteration.

Termination: When the loop terminates, we have $i = m + 1$, and so the loop invariant implies that stack S consists of exactly the vertices of $\text{CH}(Q_m)$, which is $\text{CH}(Q)$, in counterclockwise order from bottom to top. This completes the proof. ■

We now show that the running time of GRAHAM-SCAN is $O(n \lg n)$, where $n = |Q|$. Line 1 takes $\Theta(n)$ time. Line 2 takes $O(n \lg n)$ time, using merge sort or heapsort to sort the polar angles and the cross-product method of Section 33.1 to compare angles. (We can remove all but the farthest point with the same polar angle in total of $O(n)$ time over all n points.) Lines 3–6 take $O(1)$ time. Because $m \leq n - 1$, the **for** loop of lines 7–10 executes at most $n - 3$ times. Since PUSH takes $O(1)$ time, each iteration takes $O(1)$ time exclusive of the time spent in the **while** loop of lines 8–9, and thus overall the **for** loop takes $O(n)$ time exclusive of the nested **while** loop.

We use aggregate analysis to show that the **while** loop takes $O(n)$ time overall. For $i = 0, 1, \ldots, m$, we push each point p_i onto stack S exactly once. As in the analysis of the MULTIPOP procedure of Section 17.1, we observe that we can pop at most the number of items that we push. At least three points—p_0, p_1, and p_m—are never popped from the stack, so that in fact at most $m - 2$ POP operations are performed in total. Each iteration of the **while** loop performs one POP, and so there are at most $m - 2$ iterations of the **while** loop altogether. Since the test in line 8 takes $O(1)$ time, each call of POP takes $O(1)$ time, and $m \leq n - 1$, the total time taken by the **while** loop is $O(n)$. Thus, the running time of GRAHAM-SCAN is $O(n \lg n)$.

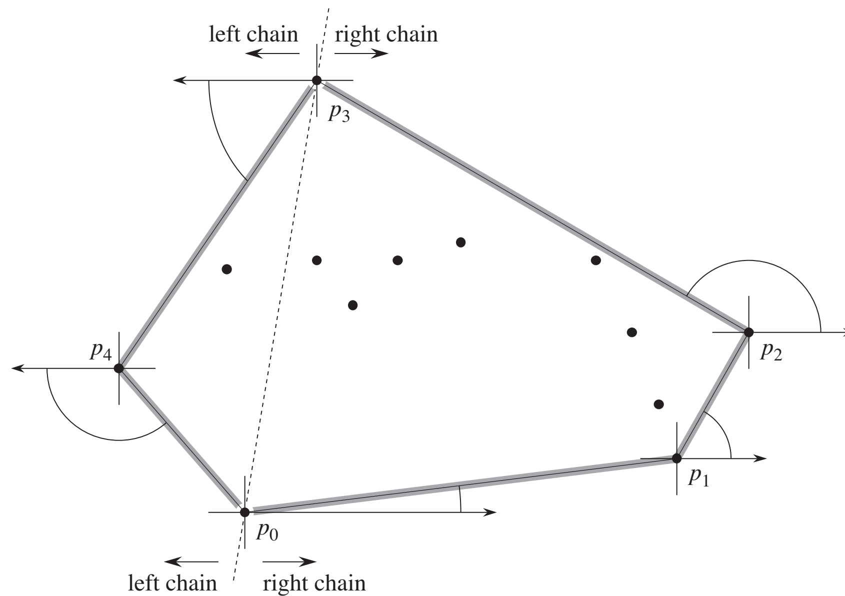

Figure 33.9 The operation of Jarvis's march. We choose the first vertex as the lowest point p_0. The next vertex, p_1, has the smallest polar angle of any point with respect to p_0. Then, p_2 has the smallest polar angle with respect to p_1. The right chain goes as high as the highest point p_3. Then, we construct the left chain by finding smallest polar angles with respect to the negative x-axis.

Jarvis's march

Jarvis's march computes the convex hull of a set Q of points by a technique known as ***package wrapping*** (or ***gift wrapping***). The algorithm runs in time $O(nh)$, where h is the number of vertices of $\text{CH}(Q)$. When h is $o(\lg n)$, Jarvis's march is asymptotically faster than Graham's scan.

Intuitively, Jarvis's march simulates wrapping a taut piece of paper around the set Q. We start by taping the end of the paper to the lowest point in the set, that is, to the same point p_0 with which we start Graham's scan. We know that this point must be a vertex of the convex hull. We pull the paper to the right to make it taut, and then we pull it higher until it touches a point. This point must also be a vertex of the convex hull. Keeping the paper taut, we continue in this way around the set of vertices until we come back to our original point p_0.

More formally, Jarvis's march builds a sequence $H = \langle p_0, p_1, \ldots, p_{h-1} \rangle$ of the vertices of $\text{CH}(Q)$. We start with p_0. As Figure 33.9 shows, the next vertex p_1 in the convex hull has the smallest polar angle with respect to p_0. (In case of ties, we choose the point farthest from p_0.) Similarly, p_2 has the smallest polar angle

with respect to p_1, and so on. When we reach the highest vertex, say p_k (breaking ties by choosing the farthest such vertex), we have constructed, as Figure 33.9 shows, the ***right chain*** of CH(Q). To construct the ***left chain***, we start at p_k and choose p_{k+1} as the point with the smallest polar angle with respect to p_k, but *from the negative x-axis*. We continue on, forming the left chain by taking polar angles from the negative x-axis, until we come back to our original vertex p_0.

We could implement Jarvis's march in one conceptual sweep around the convex hull, that is, without separately constructing the right and left chains. Such implementations typically keep track of the angle of the last convex-hull side chosen and require the sequence of angles of hull sides to be strictly increasing (in the range of 0 to 2π radians). The advantage of constructing separate chains is that we need not explicitly compute angles; the techniques of Section 33.1 suffice to compare angles.

If implemented properly, Jarvis's march has a running time of $O(nh)$. For each of the h vertices of CH(Q), we find the vertex with the minimum polar angle. Each comparison between polar angles takes $O(1)$ time, using the techniques of Section 33.1. As Section 9.1 shows, we can compute the minimum of n values in $O(n)$ time if each comparison takes $O(1)$ time. Thus, Jarvis's march takes $O(nh)$ time.

Exercises

33.3-1

Prove that in the procedure GRAHAM-SCAN, points p_1 and p_m must be vertices of CH(Q).

33.3-2

Consider a model of computation that supports addition, comparison, and multiplication and for which there is a lower bound of $\Omega(n \lg n)$ to sort n numbers. Prove that $\Omega(n \lg n)$ is a lower bound for computing, in order, the vertices of the convex hull of a set of n points in such a model.

33.3-3

Given a set of points Q, prove that the pair of points farthest from each other must be vertices of CH(Q).

33.3-4

For a given polygon P and a point q on its boundary, the ***shadow*** of q is the set of points r such that the segment $\overline{qr}$ is entirely on the boundary or in the interior of P. As Figure 33.10 illustrates, a polygon P is ***star-shaped*** if there exists a point p in the interior of P that is in the shadow of every point on the boundary of P. The set of all such points p is called the ***kernel*** of P. Given an n-vertex,

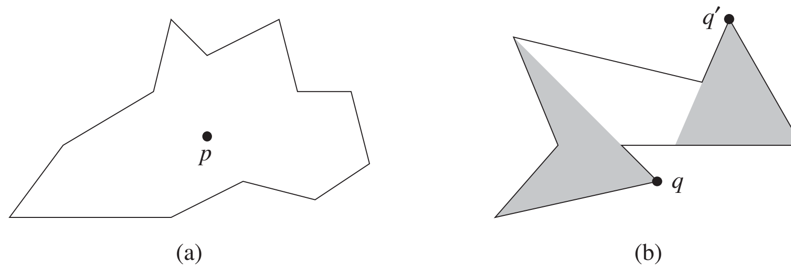

Figure 33.10 The definition of a star-shaped polygon, for use in Exercise 33.3-4. **(a)** A star-shaped polygon. The segment from point p to any point q on the boundary intersects the boundary only at q. **(b)** A non-star-shaped polygon. The shaded region on the left is the shadow of q, and the shaded 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 $\mathrm{CH}(P)$ 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 δ_L and δ_R, respectively, and let $\delta = \min(\delta_L, \delta_R)$.

Combine: The closest pair is either the pair with distance δ 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 δ. Observe that if a pair of points has distance less than δ, both points of the pair must be within δ units of line l. Thus, as Figure 33.11(a) shows, they both must reside in the 2δ-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 2δ-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 δ 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 δ' 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 δ'. Otherwise, return the closest pair and its distance δ 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 δ' between p_L and p_R is strictly less than δ. Point p_L must be on or to the left of line l and less than δ units away. Similarly, p_R is on or to the right of l and less than δ units away. Moreover, p_L and p_R are within δ 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 δ 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

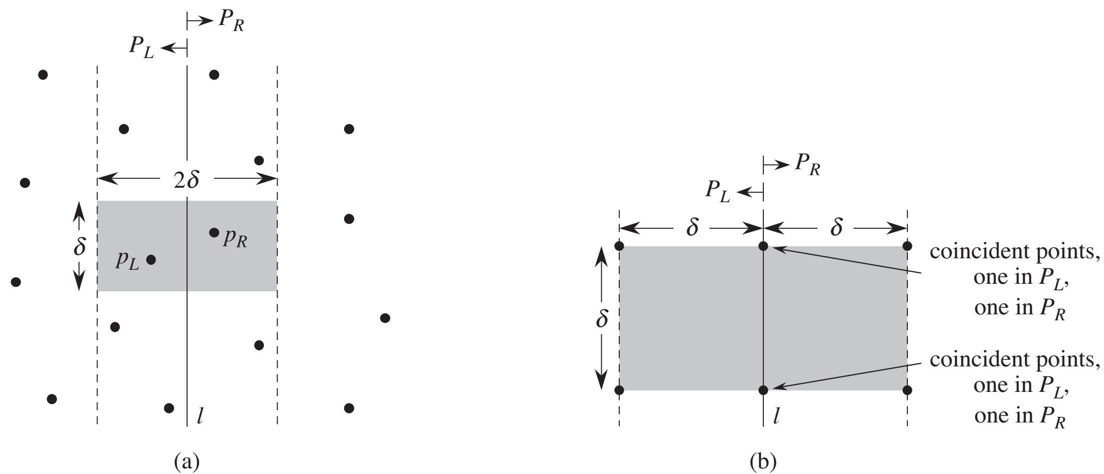

Figure 33.11 Key concepts in the proof that the closest-pair algorithm needs to check only 7 points following each point in the array Y'. **(a)** If $p_L \in P_L$ and $p_R \in P_R$ are less than δ units apart, they must reside within a $\delta \times 2\delta$ rectangle centered at line l. **(b)** How 4 points that are pairwise at least δ 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 .. Y.length] and Y_R[1 .. Y.length] be new arrays
2  Y_L.length = Y_R.length = 0
3  for i = 1 to Y.length
4      if Y[i] ∈ 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

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

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'.

33.4-3

We can define the distance between two points in ways other than euclidean. In the plane, the ***L_m-distance*** between points p_1 and p_2 is given by the expression $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$. Euclidean distance, therefore, is L_2-distance. Modify the closest-pair algorithm to use the L_1-distance, which is also known as the ***Manhattan distance***.

33.4-4

Given two points p_1 and p_2 in the plane, the L_∞-distance between them is given by $\max(|x_1 - x_2|, |y_1 - y_2|)$. Modify the closest-pair algorithm to use the L_∞-distance.

33.4-5

Suppose that $\Omega(n)$ of the points given to the closest-pair algorithm are covertical. Show how to determine the sets P_L and P_R and how to determine whether each point of Y is in P_L or P_R so that the running time for the closest-pair algorithm remains $O(n \lg n)$.

33.4-6

Suggest a change to the closest-pair algorithm that avoids presorting the Y array but leaves the running time as $O(n \lg n)$. (*Hint:* Merge sorted arrays Y_L and Y_R to form the sorted array Y.)

Problems

33-1 Convex layers

Given a set Q of points in the plane, we define the ***convex layers*** of Q inductively. The first convex layer of Q consists of those points in Q that are vertices of $\text{CH}(Q)$. For $i > 1$, define Q_i to consist of the points of Q with all points in convex layers $1, 2, \ldots, i-1$ removed. Then, the ith convex layer of Q is $\text{CH}(Q_i)$ if $Q_i \neq \emptyset$ and is undefined otherwise.

a. Give an $O(n^2)$-time algorithm to find the convex layers of a set of n points.

b. Prove that $\Omega(n \lg n)$ time is required to compute the convex layers of a set of n points with any model of computation that requires $\Omega(n \lg n)$ time to sort n real numbers.

33-2 Maximal layers

Let Q be a set of n points in the plane. We say that point (x, y) ***dominates*** point (x', y') if $x \geq x'$ and $y \geq y'$. A point in Q that is dominated by no other points in Q is said to be ***maximal***. Note that Q may contain many maximal points, which can be organized into ***maximal layers*** as follows. The first maximal layer L_1 is the set of maximal points of Q. For $i > 1$, the ith maximal layer L_i is the set of maximal points in $Q - \bigcup_{j=1}^{i-1} L_j$.

Suppose that Q has k nonempty maximal layers, and let y_i be the y-coordinate of the leftmost point in L_i for $i = 1, 2, \ldots, k$. For now, assume that no two points in Q have the same x- or y-coordinate.

a. Show that $y_1 > y_2 > \cdots > y_k$.

Consider a point (x, y) that is to the left of any point in Q and for which y is distinct from the y-coordinate of any point in Q. Let $Q' = Q \cup \{(x, y)\}$.

b. Let j be the minimum index such that $y_j < y$, unless $y < y_k$, in which case we let $j = k + 1$. Show that the maximal layers of Q' are as follows:

- If $j \leq k$, then the maximal layers of Q' are the same as the maximal layers of Q, except that L_j also includes (x, y) as its new leftmost point.
- If $j = k + 1$, then the first k maximal layers of Q' are the same as for Q, but in addition, Q' has a nonempty $(k + 1)$st maximal layer: $L_{k+1} = \{(x, y)\}$.

c. Describe an $O(n \lg n)$-time algorithm to compute the maximal layers of a set Q of n points. (*Hint:* Move a sweep line from right to left.)

d. Do any difficulties arise if we now allow input points to have the same x- or y-coordinate? Suggest a way to resolve such problems.

33-3 Ghostbusters and ghosts

A group of n Ghostbusters is battling n ghosts. Each Ghostbuster carries a proton pack, which shoots a stream at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits the ghost. The Ghostbusters decide upon the following strategy. They will pair off with the ghosts, forming n Ghostbuster-ghost pairs, and then simultaneously each Ghostbuster will shoot a stream at his chosen ghost. As we all know, it is *very* dangerous to let streams cross, and so the Ghostbusters must choose pairings for which no streams will cross.

Assume that the position of each Ghostbuster and each ghost is a fixed point in the plane and that no three positions are colinear.

a. Argue that there exists a line passing through one Ghostbuster and one ghost such that the number of Ghostbusters on one side of the line equals the number of ghosts on the same side. Describe how to find such a line in $O(n \lg n)$ time.

b. Give an $O(n^2 \lg n)$-time algorithm to pair Ghostbusters with ghosts in such a way that no streams cross.

33-4 Picking up sticks

Professor Charon has a set of n sticks, which are piled up in some configuration. Each stick is specified by its endpoints, and each endpoint is an ordered triple giving its (x, y, z) coordinates. No stick is vertical. He wishes to pick up all the sticks, one at a time, subject to the condition that he may pick up a stick only if there is no other stick on top of it.

a. Give a procedure that takes two sticks a and b and reports whether a is above, below, or unrelated to b.

b. Describe an efficient algorithm that determines whether it is possible to pick up all the sticks, and if so, provides a legal order in which to pick them up.

33-5 Sparse-hulled distributions

Consider the problem of computing the convex hull of a set of points in the plane that have been drawn according to some known random distribution. Sometimes, the number of points, or size, of the convex hull of n points drawn from such a distribution has expectation $O(n^{1-\epsilon})$ for some constant $\epsilon > 0$. We call such a distribution ***sparse-hulled***. Sparse-hulled distributions include the following:

- Points drawn uniformly from a unit-radius disk. The convex hull has expected size $\Theta(n^{1/3})$.
- Points drawn uniformly from the interior of a convex polygon with k sides, for any constant k. The convex hull has expected size $\Theta(\lg n)$.
- Points drawn according to a two-dimensional normal distribution. The convex hull has expected size $\Theta(\sqrt{\lg n})$.

a. Given two convex polygons with n_1 and n_2 vertices respectively, show how to compute the convex hull of all $n_1 + n_2$ points in $O(n_1 + n_2)$ time. (The polygons may overlap.)

b. Show how to compute the convex hull of a set of n points drawn independently according to a sparse-hulled distribution in $O(n)$ average-case time. (*Hint:* Recursively find the convex hulls of the first $n/2$ points and the second $n/2$ points, and then combine the results.)

Chapter notes

This chapter barely scratches the surface of computational-geometry algorithms and techniques. Books on computational geometry include those by Preparata and Shamos [282], Edelsbrunner [99], and O'Rourke [269].

Although geometry has been studied since antiquity, the development of algorithms for geometric problems is relatively new. Preparata and Shamos note that the earliest notion of the complexity of a problem was given by E. Lemoine in 1902. He was studying euclidean constructions—those using a compass and a ruler—and devised a set of five primitives: placing one leg of the compass on a given point, placing one leg of the compass on a given line, drawing a circle, passing the ruler's edge through a given point, and drawing a line. Lemoine was interested in the number of primitives needed to effect a given construction; he called this amount the "simplicity" of the construction.

The algorithm of Section 33.2, which determines whether any segments intersect, is due to Shamos and Hoey [313].

The original version of Graham's scan is given by Graham [150]. The package-wrapping algorithm is due to Jarvis [189]. Using a decision-tree model of computation, Yao [359] proved a worst-case lower bound of $\Omega(n \lg n)$ for the running time of any convex-hull algorithm. When the number of vertices h of the convex hull is taken into account, the prune-and-search algorithm of Kirkpatrick and Seidel [206], which takes $O(n \lg h)$ time, is asymptotically optimal.

The $O(n \lg n)$-time divide-and-conquer algorithm for finding the closest pair of points is by Shamos and appears in Preparata and Shamos [282]. Preparata and Shamos also show that the algorithm is asymptotically optimal in a decision-tree model.