Geometry

13

This chapter discusses algorithm techniques related to geometry. The general goal of the chapter is to find ways to *conveniently* solve geometric problems, avoiding special cases and tricky implementations.

Section 13.1 introduces the C++ complex number class which has useful tools for geometric problems. After this, we will learn to use cross products to solve various problems, such as testing whether two line segments intersect and calculating the distance from a point to a line. Finally, we discuss ways to calculate polygon areas and explore special properties of Manhattan distances.

Section 13.2 focuses on sweep line algorithms which play an important role in computational geometry. We will see how to use such algorithms for counting intersection points, finding closest points, and constructing convex hulls.

13.1 Geometric Techniques

A challenge when solving geometric problems is how to approach the problem so that the number of special cases is as small as possible and there is a convenient way to implement the solution. In this section, we will go through a set of tools that make solving geometric problems easier.

13.1.1 Complex Numbers

A *complex number* is a number of the form $x + yi$, where $i = \sqrt{-1}$ is the *imaginary unit*. A geometric interpretation of a complex number is that it represents a two-dimensional point (x, y) or a vector from the origin to a point (x, y). For example, Fig. 13.1 illustrates the complex number $4 + 2i$.

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_13

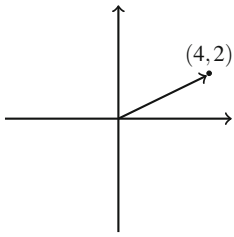


Fig. 13.1 Complex number $4 + 2i$ interpreted as a point and a vector

The C++ complex number class `complex` is useful when solving geometric problems. Using the class we can represent points and vectors as complex numbers, and use the features of the class to manipulate them. To do this, let us first define a coordinate type `C`. Depending on the situation, a suitable type is `long long` or `long double`. As a general rule, it is good to use integer coordinates whenever possible, because calculations with integers are exact.

Here are possible coordinate type definitions:

```
typedef long long C;
```

```
typedef long double C;
```

After this, we can define a complex type `P` that represents a point or a vector:

```
typedef complex<C> P;
```

Finally, the following macros refer to x and y coordinates:

```
#define X real()
#define Y imag()
```

For example, the following code creates a point $p = (4, 2)$ and prints its x and y coordinates:

```
P p = {4,2};
cout << p.X << "" << p.Y << "\n"; // 4 2
```

Then, the following code creates vectors $v = (3, 1)$ and $u = (2, 2)$, and after that calculates the sum $s = v + u$.

```
P v = {3,1};
P u = {2,2};
P s = v+u;
cout << s.X << "" << s.Y << "\n"; // 5 3
```

Functions The `complex` class also has functions that are useful in geometric problems. The following functions should only be used when the coordinate type is `long double` (or another floating point type).

The function `abs(v)` calculates the length $|v|$ of a vector $v = (x, y)$ using the formula $\sqrt{x^2 + y^2}$. The function can also be used for calculating the distance between points (x_1, y_1) and (x_2, y_2), because that distance equals the length of the vector $(x_2 - x_1, y_2 - y_1)$. For example, the following code calculates the distance between points $(4, 2)$ and $(3, -1)$

```
P a = {4,2};
P b = {3,-1};
cout << abs(b-a) << "\n"; // 3.16228
```

The function `arg(v)` calculates the angle of a vector $v = (x, y)$ with respect to the x-axis. The function gives the angle in radians, where r radians equals $180r/\pi$ degrees. The angle of a vector that points to the right is 0, and angles decrease clockwise and increase counterclockwise.

The function `polar(s, a)` constructs a vector whose length is s and that points to an angle a, given in radians. A vector can be rotated by an angle a by multiplying it by a vector with length 1 and angle a.

The following code calculates the angle of the vector $(4, 2)$, rotates it $1/2$ radians counterclockwise, and then calculates the angle again:

```
P v = {4,2};
cout << arg(v) << "\n"; // 0.463648
v *= polar(1.0,0.5);
cout << arg(v) << "\n"; // 0.963648
```

13.1.2 Points and Lines

The *cross product* $a \times b$ of vectors $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is defined to be $x_1 y_2 - x_2 y_1$. It tells us the direction to which b turns when it is placed directly after a. There are three cases illustrated in Fig. 13.2:

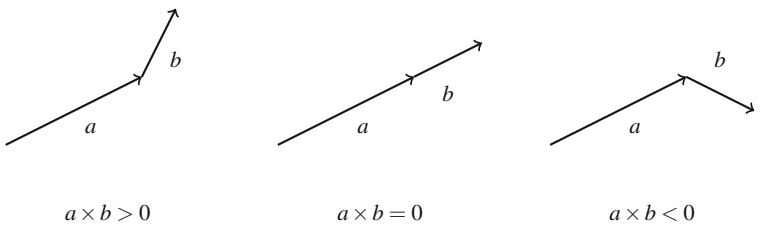


Fig. 13.2 Interpretation of cross products

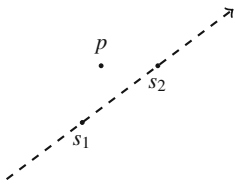


Fig. 13.3 Testing the location of a point

- $a \times b > 0$: b turns left
- $a \times b = 0$: b does not turn (or turns 180 degrees)
- $a \times b < 0$: b turns right

For example, the cross product of vectors $a = (4, 2)$ and $b = (1, 2)$ is $4 \cdot 2 - 2 \cdot 1 = 6$, which corresponds to the first scenario of Fig. 13.2. The cross product can be calculated using the following code:

```
P a = {4,2};
P b = {1,2};
C p = (conj(a)*b).Y; // 6
```

The above code works, because the function `conj` negates the y coordinate of a vector, and when the vectors $(x_1, -y_1)$ and (x_2, y_2) are multiplied together, the y coordinate of the result is $x_1 y_2 - x_2 y_1$.

Next we will go through some applications of cross products.

Testing Point Location Cross products can be used to test whether a point is located on the left or right side of a line. Assume that the line goes through points s_1 and s_2, we are looking from s_1 to s_2 and the point is p. For example, in Fig. 13.3, p is located on the left side of the line.

The cross product $(p - s_1) \times (p - s_2)$ tells us the location of the point p. If the cross product is positive, p is located on the left side, and if the cross product is negative, p is located on the right side. Finally, if the cross product is zero, the points s_1, s_2, and p are on the same line.

Line Segment Intersection Next, consider the problem of testing whether two line segments ab and cd intersect. It turns out that if the line segments intersect, there are three possible cases:

Case 1: The line segments are on the same line and they overlap each other. In this case, there is an infinite number of intersection points. For example, in Fig. 13.4, all points between c and b are intersection points. To detect this case, we can use cross products to test if all points are on the same line. If they are, we can then sort them and check whether the line segments overlap each other.

Case 2: The line segments have a common vertex that is the only intersection point. For example, in Fig. 13.5 the intersection point is $b = c$. This case is easy to check, because there are only four possibilities for the intersection point: $a = c$, $a = d$, $b = c$, and $b = d$.

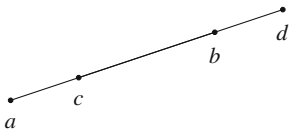


Fig. 13.4 Case 1: the line segments are on the same line and overlap each other

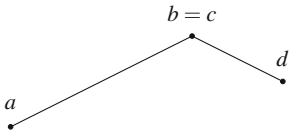


Fig. 13.5 Case 2: the line segments have a common vertex

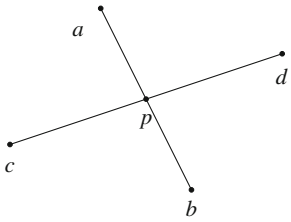


Fig. 13.6 Case 3: the line segments have an intersection point that is not a vertex

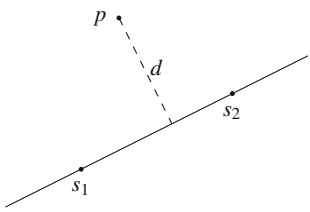


Fig. 13.7 Calculating the distance from p to the line

Case 3: There is exactly one intersection point that is not a vertex of any line segment. In Fig. 13.6, the point p is the intersection point. In this case, the line segments intersect exactly when both points c and d are on different sides of a line through a and b, and points a and b are on different sides of a line through c and d. We can use cross products to check this.

Distance from a Point to a Line Another property of cross products is that the area of a triangle can be calculated using the formula

$$\frac{|(a-c)\times(b-c)|}{2},$$

where a, b, and c are the vertices of the triangle. Using this fact, we can derive a formula for calculating the shortest distance between a point and a line. For example, in Fig. 13.7, d is the shortest distance between the point p and the line that is defined by the points s_1 and s_2.

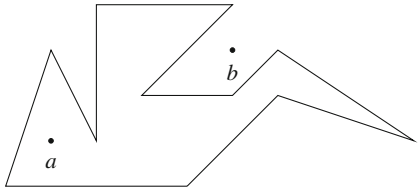


Fig. 13.8 Point a is inside and point b is outside the polygon

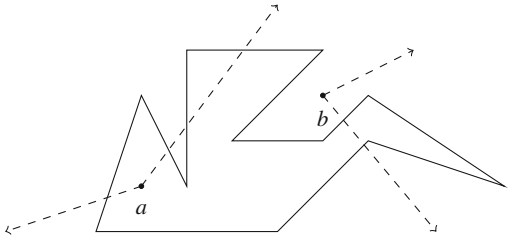


Fig. 13.9 Sending rays from points a and b

The area of a triangle whose vertices are s_1, s_2, and p can be calculated in two ways: it is both $\frac{1}{2}|s_2 - s_1|d$ (the standard formula taught in school) and $\frac{1}{2}((s_1 - p) \times (s_2 - p))$ (the cross product formula). Thus, the shortest distance is

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

Point in a Polygon Finally, consider the problem of testing whether a point is located inside or outside a polygon. For example, in Fig. 13.8, point a is inside the polygon and point b is outside the polygon.

A convenient way to solve the problem is to send a *ray* from the point to an arbitrary direction and calculate the number of times it touches the boundary of the polygon. If the number is odd, the point is inside the polygon, and if the number is even, the point is outside the polygon.

For example, in Fig. 13.9, the rays from a touch 1 and 3 times the boundary of the polygon, so a is inside the polygon. In a similar way, the rays from b touch 0 and 2 times the boundary of the polygon, so b is outside the polygon.

13.1.3 Polygon Area

A general formula for calculating the area of a polygon, sometimes called the *shoelace formula*, is as follows:

$$\frac{1}{2}|\sum_{i=1}^{n-1}(p_i \times p_{i+1})| = \frac{1}{2}|\sum_{i=1}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)|.$$

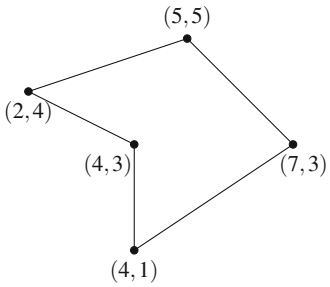


Fig. 13.10 A polygon whose area is 17/2

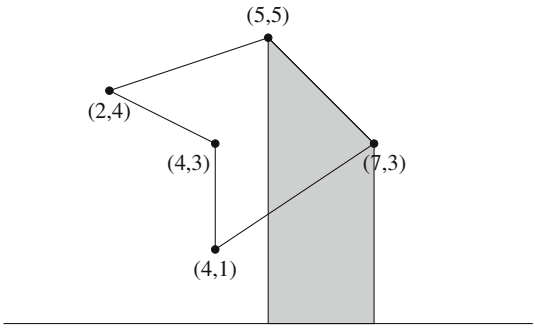


Fig. 13.11 Calculating the area of the polygon using trapezoids

Here the vertices are $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, $\ldots$, $p_n = (x_n, y_n)$ in such an order that p_i and p_{i+1} are adjacent vertices on the boundary of the polygon, and the first and last vertex is the same, i.e., $p_1 = p_n$.

For example, the area of the polygon in Fig. 13.10 is

$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = 17/2.$$

The idea behind the formula is to go through trapezoids whose one side is a side of the polygon, and another side lies on the horizontal line $y = 0$. For example, Fig. 13.11 shows one such trapezoid. The area of each trapezoid is

$$(x_{i+1} - x_i)\frac{y_i + y_{i+1}}{2},$$

where the vertices of the polygon are p_i and p_{i+1}. If $x_{i+1} > x_i$, the area is positive, and if $x_{i+1} < x_i$, the area is negative. Then, the area of the polygon is the sum of areas of all such trapezoids, which yields the formula

$$\left|\sum_{i=1}^{n-1}(x_{i+1} - x_i)\frac{y_i + y_{i+1}}{2}\right| = \frac{1}{2}\left|\sum_{i=1}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)\right|.$$

Note that the absolute value of the sum is taken, because the value of the sum may be positive or negative, depending on whether we walk clockwise or counterclockwise along the boundary of the polygon.

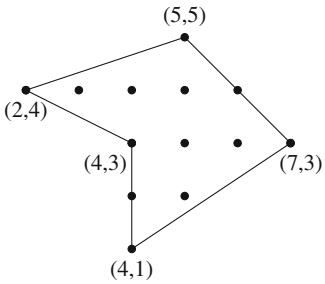


Fig. 13.12 Calculating the polygon area using Pick's theorem

Pick's Theorem *Pick's theorem* provides another way to calculate the area of a polygon, assuming that all vertices of the polygon have integer coordinates. Pick's theorem tells us that the area of the polygon is

$$a + b/2 - 1,$$

where a is the number of integer points inside the polygon and b is the number of integer points on the boundary of the polygon. For example, the area of the polygon in Fig. 13.12 is

$$6 + 7/2 - 1 = 17/2.$$

13.1.4 Distance Functions

A *distance function* defines the distance between two points. The usual distance function is the *Euclidean distance* where the distance between points (x_1, y_1) and (x_2, y_2) is

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

An alternative distance function is the *Manhattan distance* where the distance between points (x_1, y_1) and (x_2, y_2) is

$$|x_1 - x_2| + |y_1 - y_2|.$$

For example, in Fig. 13.13, the Euclidean distance between the points is

$$\sqrt{(5-2)^2 + (2-1)^2} = \sqrt{10}$$

and the Manhattan distance is

$$|5 - 2| + |2 - 1| = 4.$$

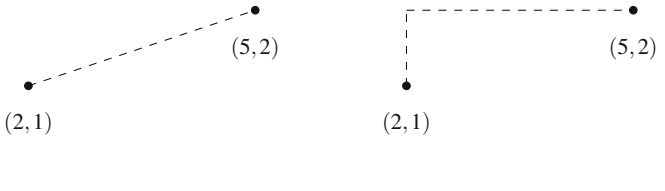


Fig. 13.13 Two distance functions

Fig. 13.14 Regions within a distance of 1

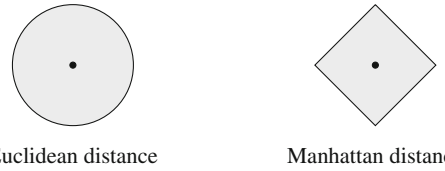


Fig. 13.15 Points B and C have the maximum Manhattan distance

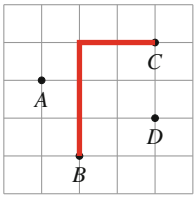


Fig. 13.16 Maximum Manhattan distance after transforming the coordinates

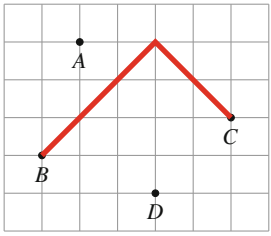


Figure 13.14 shows regions that are within a distance of 1 from the center point, using the Euclidean and Manhattan distances.

Some problems are easier to solve if Manhattan distances are used instead of Euclidean distances. As an example, given a set of points in the two-dimensional plane, consider the problem of finding two points whose Manhattan distance is maximum. For example, in Fig. 13.15, we should select points B and C to get the maximum Manhattan distance 5.

A useful technique related to Manhattan distances is to transform the coordinates so that a point (x, y) becomes $(x + y, y - x)$. This rotates the point set 45° and scales it. For example, Fig. 13.16 shows the result of the transformation in our example scenario.

Then, consider two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ whose transformed coordinates are $p_1' = (x_1', y_1')$ and $p_2' = (x_2', y_2')$. Now there are two ways to express the Manhattan distance between p_1 and p_2:

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x_1' - x_2'|, |y_1' - y_2'|)$$

For example, if $p_1 = (1, 0)$ and $p_2 = (3, 3)$, the transformed coordinates are $p_1' = (1, -1)$ and $p_2' = (6, 0)$ and the Manhattan distance is

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

The transformed coordinates provide a simple way to operate with Manhattan distances, because we can consider x and y coordinates separately. In particular, to maximize the Manhattan distance, we should find two points whose transformed coordinates maximize the value of

$$\max(|x_1' - x_2'|, |y_1' - y_2'|).$$

This is easy, because either the horizontal or vertical difference of the transformed coordinates has to be maximum.

13.2 Sweep Line Algorithms

Many geometric problems can be solved using *sweep line* algorithms. The idea in such algorithms is to represent an instance of the problem as a set of events that correspond to points in the plane. Then, the events are processed in increasing order according to their x or y coordinates.

13.2.1 Intersection Points

Given a set of n line segments, each of them being either horizontal or vertical, consider the problem of counting the total number of intersection points. For example, in Fig. 13.17, there are five line segments and three intersection points.

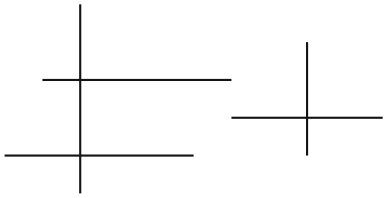


Fig. 13.17 Five line segments with three intersection points

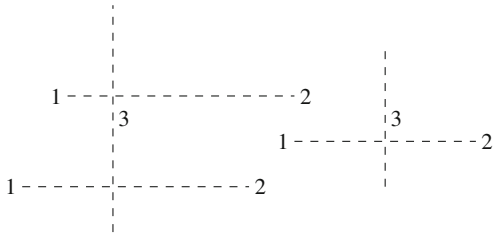


Fig. 13.18 Events that correspond to the line segments

It is easy to solve the problem in $O(n^2)$ time, because we can go through all possible pairs of line segments and check if they intersect. However, we can solve the problem more efficiently in $O(n \log n)$ time using a sweep line algorithm and a range query data structure. The idea is to process the endpoints of the line segments from left to right and focus on three types of events:

(1) horizontal segment begins
(2) horizontal segment ends
(3) vertical segment

Figure 13.18 shows the events in our example scenario.

After creating the events, we go through them from left to right and use a data structure that maintains the y coordinates of the active horizontal segments. At event 1, we add the y coordinate of the segment to the structure, and at event 2, we remove the y coordinate from the structure. Intersection points are calculated at event 3: when processing a vertical segment between points y_1 and y_2, we count the number of active horizontal segments whose y coordinate is between y_1 and y_2, and add this number to the total number of intersection points.

To store y coordinates of horizontal segments, we can use a binary indexed or segment tree, possibly with index compression. Processing each event takes $O(\log n)$ time, so the algorithm works in $O(n \log n)$ time.

13.2.2 Closest Pair Problem

Given a set of n points, our next problem is to find two points whose Euclidean distance is minimum. For example, Fig. 13.19 shows a set of points, where the closest pair is painted black.

This is another example of a problem that can be solved in $O(n \log n)$ time using a sweep line algorithm.[1] We go through the points from left to right and maintain

[1] Creating an efficient algorithm for the closest pair problem was once an important open problem in computational geometry. Finally, Shamos and Hoey [26] discovered a divide and conquer algorithm that works in $O(n \log n)$ time. The sweep line algorithm presented here has common elements with their algorithm, but it is easier to implement.

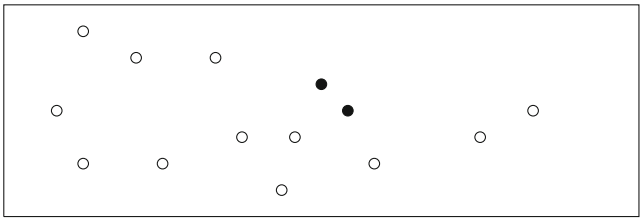


Fig. 13.19 An instance of the closest pair problem

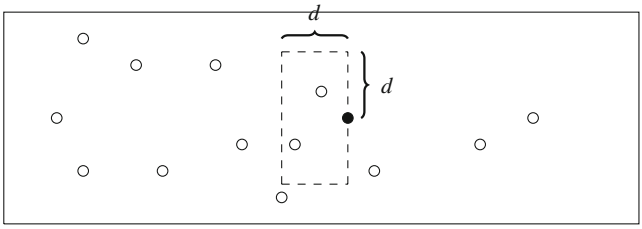
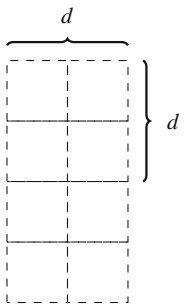


Fig. 13.20 Region where the closest point must lie

Fig. 13.21 Closest point region contains $O(1)$ points



a value d: the minimum distance between two points seen so far. At each point, we find its nearest point to the left. If the distance is less than d, it is the new minimum distance and we update the value of d.

If the current point is (x, y) and there is a point to the left within a distance of less than d, the x coordinate of such a point must be between $[x - d, x]$ and the y coordinate must be between $[y - d, y + d]$. Thus, it suffices to only consider points that are located in those ranges, which makes the algorithm efficient. For example, in Fig. 13.20, the region marked with dashed lines contains the points that can be within a distance of d from the active point.

The efficiency of the algorithm is based on the fact that the region always contains only $O(1)$ points. To see why this holds, consider Fig. 13.21. Since the current minimum distance between two points is d, each $d/2 \times d/2$ square may contain at most one point. Thus, there are at most eight points in the region.

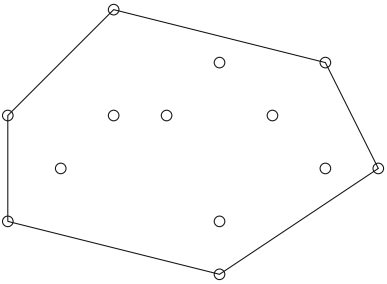


Fig. 13.22 Convex hull of a point set

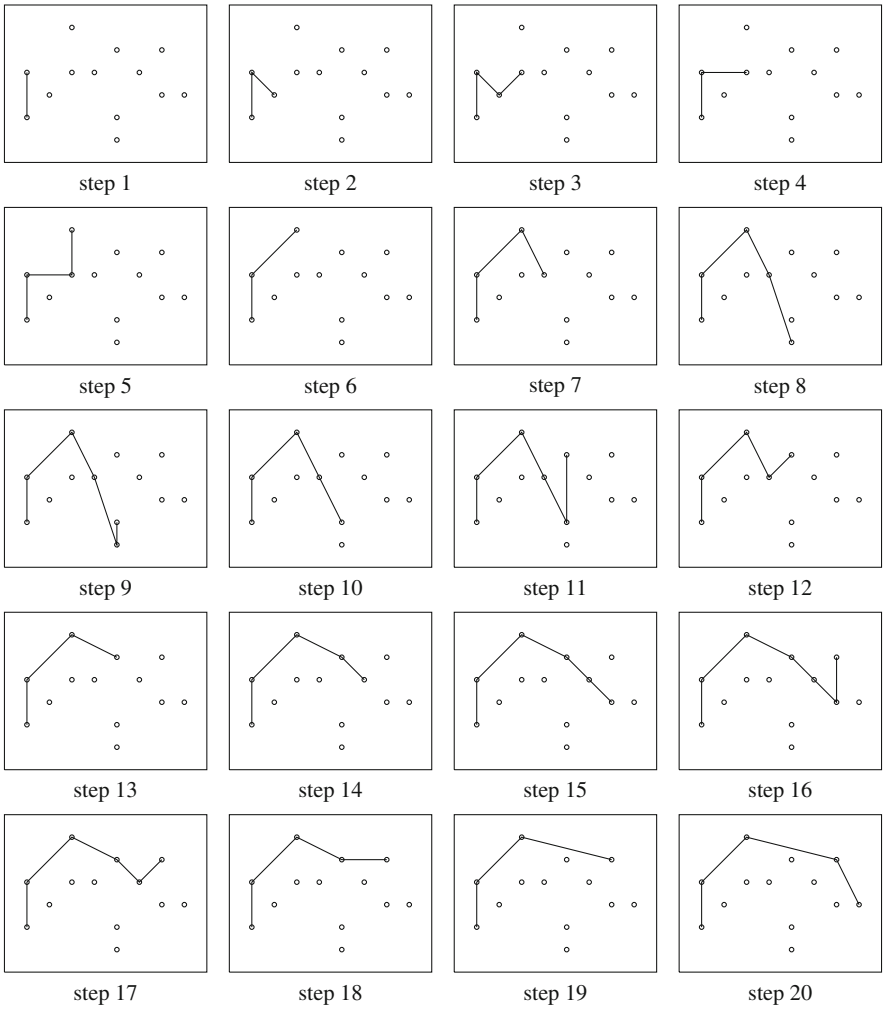


Fig. 13.23 Constructing the upper part of the convex hull using Andrew's algorithm

We can go through the points in the region in $O(\log n)$ time by maintaining a set of points whose x coordinates are between $[x-d, x]$ so that the points are sorted in increasing order according to their y coordinates. The time complexity of the algorithm is $O(n \log n)$, because we go through n points and determine for each point its nearest point to the left in $O(\log n)$ time.

13.2.3 Convex Hull Problem

A *convex hull* is the smallest convex polygon that contains all points of a given point set. Here convexity means that a line segment between any two vertices of the polygon is completely inside the polygon. For example, Fig. 13.22 shows the convex hull of a point set.

There are many efficient algorithms for constructing convex hulls. Perhaps the simplest among them is *Andrew's algorithm* [2], which we will describe next. The algorithm first determines the leftmost and rightmost points in the set, and then constructs the convex hull in two parts: first the upper hull and then the lower hull. Both parts are similar, so we can focus on constructing the upper hull.

First, we sort the points primarily according to x coordinates and secondarily according to y coordinates. After this, we go through the points and add each point to the hull. Always after adding a point to the hull, we make sure that the last line segment in the hull does not turn left. As long as it turns left, we repeatedly remove the second last point from the hull. Figure 13.23 shows how Andrew's algorithm creates the upper hull for our example point set.