23 Geometric algorithms

Geometry is the science of correct reasoning on incorrect figures.
– George Pólya

Geometry existed before creation.
– Plato

In this chapter, I introduce algorithms for solving some common geometric problems. These algorithms find immense applications in several practical and engineering problems, most notably in the areas of computer graphics, image processing, robotics and geographic information systems. For simplicity, I restrict the discussion only to plane geometry. Geometry in three and higher dimensions is also important and may be learned from standard textbooks on computational geometry.

23.1 Basic definitions

Let us denote the set of all real numbers by the blackboard bold letter $\mathbb{R}$. The two-dimensional *plane* is denoted by $\mathbb{R}^2$ and is typically represented by two orthogonal coordinates x and y.

A *point* $P = (x, y)$ in the plane is specified by two coordinates x and y. We typically use a floating-point variable (`float` or `double`) for representing each coordinate. For a point P, $x(P)$ denotes its x-coordinate and $y(P)$ denotes its y-coordinate.

Working on floating-point variables is necessarily accompanied by floating-point approximations that may, in unfortunate situations, lead to erroneous conclusions. In applications where such errors are likely and/or undesirable, better representations of real numbers may be adopted. For example, if the computation is restricted only to rational numbers, one may use a pair of multiple-precision integers in order to represent each coordinate. Custom-designed higher-precision floating-point numbers may also be used. More sophisticated techniques (like use of algebraic numbers) may also be useful in situations. Whatever we do, we cannot represent *all* real numbers exactly. That is a painful consequence of simple counting arguments.[23.1] All we can achieve is reduction of errors (of course, at the cost of expensive arithmetic operations). In this chapter, however, we completely ignore the issues of floating-point instability and pretend that every operation is exact.

The plane $\mathbb{R}^2$ is endowed with a conventional notion of distance. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points in $\mathbb{R}^2$. The (*Euclidean*) *distance* between P_1 and P_2 is given by

$$d(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Other notions of distance can also be conceived of. For every $r \in \mathbb{N}$, we may define a distance

$$d_r(P_1, P_2) = \sqrt[r]{|x_1 - x_2|^r + |y_1 - y_2|^r}.$$

The standard Euclidean distance corresponds to $r = 2$. The *Manhattan distance* or the *taxi cab distance* corresponds to $r = 1$, and is defined as

$$d_1(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2|.$$

[23.1] The real line $\mathbb{R}$ is uncountable, whereas we can represent only countably many numbers using a finite representation alphabet.

We can extend this notion of distance for $r = \infty$ as

$$d_\infty(P_1, P_2) = \max(|x_1 - x_2|, |y_1 - y_2|).$$

Unless explicitly mentioned, I will use the term distance to indicate the Euclidean distance $d = d_2$.

Example

Let $P_1 = (-1, 2)$ and $P_2 = (3, 4)$. The Euclidean distance between P_1 and P_2 is $d(P_1, P_2) = \sqrt{(-1-3)^2 + (2-4)^2} = 2\sqrt{5} = 4.4721359549\ldots$. Notice that, the representation $2\sqrt{5}$ is exact, whereas any floating-point representation of it must be approximate, since $\sqrt{5}$ is irrational. We also have

$$\begin{aligned} d_1(P_1, P_2) &= |x_1 - x_2| + |y_1 - y_2| = |-1-3| + |2-4| = 4 + 2 = 6, \\ d_3(P_1, P_2) &= \sqrt[3]{|x_1 - x_2|^3 + |y_1 - y_2|^3} = \sqrt[3]{4^3 + 2^3} = 2\sqrt[3]{9} = 4.1601676461\ldots, \\ d_\infty(P_1, P_2) &= \max(|x_1 - x_2|, |y_1 - y_2|) = \max(|-1-3|, |2-4|) = 4. \end{aligned}$$

The distance d_1 corresponds to walking only parallel to the two axes for reaching P_2 from P_1, whereas d_∞ reflects the maximum of the differences between the corresponding coordinates.

A (straight) *line* (also called an *infinite line* or a *full line*) is specified by a linear equation

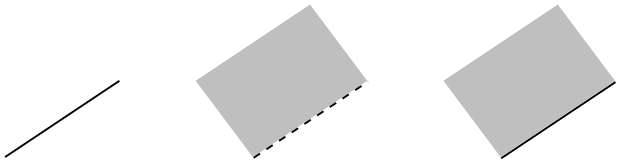
$$ax + by + c = 0,$$

where a, b are real numbers, not both zero. If we remove this line from $\mathbb{R}^2$, we obtain two *open half planes* specified as

$$\begin{aligned} H_1 &= \{(x, y) \in \mathbb{R}^2 \mid ax + by + c > 0\}, \\ H_2 &= \{(x, y) \in \mathbb{R}^2 \mid ax + by + c < 0\}. \end{aligned}$$

Adding the line to each of these open half planes gives us the *closed half planes*

$$\begin{aligned} \overline{H_1} &= \{(x, y) \in \mathbb{R}^2 \mid ax + by + c \geqslant 0\}, \\ \overline{H_2} &= \{(x, y) \in \mathbb{R}^2 \mid ax + by + c \leqslant 0\}. \end{aligned}$$



(a) A straight line (b) Open half plane (c) Closed half plane

A line is fully specified by two distinct points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on it. The equation of the line can be given as

$$(y_2 - y_1)x + (x_1 - x_2)y = x_1 y_2 - x_2 y_1.$$

A *semi-infinite line* (or a *half line* or a *ray*) is a connected subset of a full line L, that starts at some particular point on L and extends infinitely to only one of the two possible directions.

A (*line*) *segment* connecting two points P_1 and P_2 is the (finite) part between P_1 and P_2 (both inclusive) of the line passing through the given points. A line segment is typically specified by (the coordinates of) its two end points.

Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two distinct points in $\mathbb{R}^2$. The line segment L connecting P_1 and P_2 can be *parametrically* represented as

$$(1-t)P_1 + tP_2 = \big((1-t)x_1 + tx_2, (1-t)y_1 + ty_2\big) \text{ for } 0 \leqslant t \leqslant 1. \qquad \text{[Line segment]}$$

Putting $t = 0$ gives the point P_1 and taking $t = 1$ gives the point P_2. As the real-valued parameter t varies from 0 to 1, the corresponding point $(1-t)P_1 + tP_2$ moves along the line segment L from P_1 to P_2.

Of course, we may let the parameter t accept values outside the interval $0 \leqslant t \leqslant 1$. If t assumes all non-negative values, we get the half line starting at P_1 and passing through P_2 towards infinity:

$$(1-t)P_1 + tP_2 = \big((1-t)x_1 + tx_2, (1-t)y_1 + ty_2\big) \text{ for } t \geqslant 0. \qquad \text{[Half line]}$$

On the other hand, if we let t vary over the entire real line, we get the full line passing through P_1 and P_2:

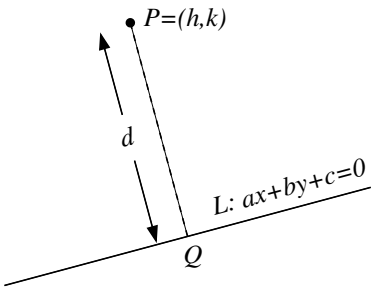
$$(1-t)P_1 + tP_2 = \big((1-t)x_1 + tx_2, (1-t)y_1 + ty_2\big) \text{ for } t \in \mathbb{R}. \qquad \text{[Full line]}$$

Let L_1 and L_2 be two distinct lines given as

$$L_1: \quad a_1x + b_1y + c_1 = 0,$$
$$L_2: \quad a_2x + b_2y + c_2 = 0.$$

These two equations have a simultaneous solution if and only if $a_1b_2 - a_2b_1 \neq 0$. The solution equals the coordinates of the *point of intersection* of L_1 and L_2. If $a_1b_2 - a_2b_1 = 0$, the two lines are parallel and do not intersect in the real plane.

Two line segments (or half lines), however, need not intersect even if they are not parallel. In order to solve the intersection problem for two segments, we may first compute the equations of the underlying lines, find the point of intersection of the lines, and compute the values of the parameter t corresponding to this point of intersection for each of the segments. If both the values of t lie in the admissible range ($0 \leqslant t \leqslant 1$), then and only then the two segments intersect. A similar criterion can be developed for the intersection of two half lines (or of a half line with a line segment).



Distance between a point and a line

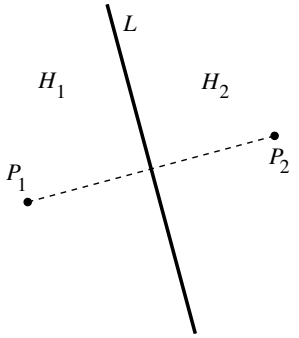
Let $P = (h, k)$ be a point and $L : ax + by + c = 0$ a line. The distance between P and L is the length of the shortest segment joining P to a point on L. We know that the shortest distance is achieved by dropping a perpendicular to L from P and is given by

$$d = \frac{|ah + bk + c|}{\sqrt{a^2 + b^2}}. \qquad (23.1.1)$$

The set of all points equidistant from two given points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ is a straight line L which happens to be the perpendicular bisector of the segment P_1P_2. The equation for L is

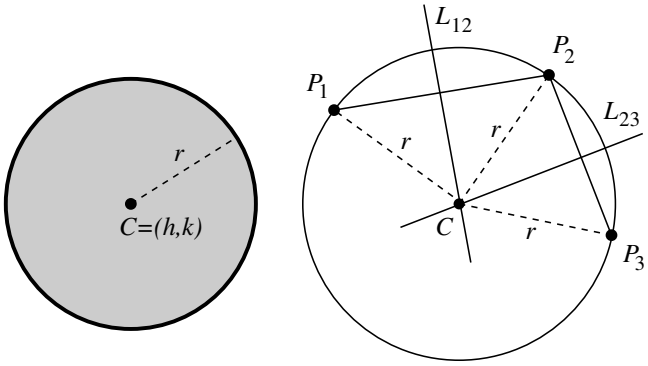
$$(x_1 - x_2)x + (y_1 - y_2)y + \frac{(x_2^2 + y_2^2) - (x_1^2 + y_1^2)}{2} = 0. \tag{23.1.2}$$

The line L cuts the plane in two open half planes. The half plane H_1 containing P_1 consists of precisely those points that are closer to P_1 than to P_2. Analogously, the half plane H_2 containing P_2 consists of precisely those points that are closer to P_2 than P_1.



Points equidistant from two points

The set of all points that are at a given distance $r \geqslant 0$ from a given point $C = (h, k)$ is the *circle* of radius r and with center at C. The region enclosed by the circle (including the circle itself) is called the *closed ball* with radius r and center C. If we exclude the perimeter from the closed ball, we get the *open ball* with radius r and center C.

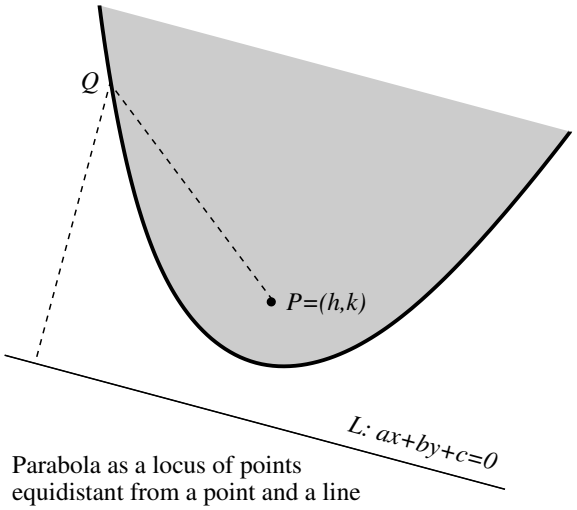


(a) circle as a locus (b) circle through three points

A unique circle passes through three points P_1, P_2 and P_3 if and only if the points are not collinear. In order to determine the circle, we first compute the perpendicular bisector L_{12} of the segment $\overline{P_1P_2}$. All points on L_{12} are equidistant from P_1 and P_2. Likewise, we compute the perpendicular bisector L_{23} of the segment $\overline{P_2P_3}$. All points on L_{23} are equidistant from P_2 and P_3. Since P_1, P_2, P_3 are not collinear, the two lines L_{12} and L_{23} are not parallel and so intersect at a unique point C. Clearly, C

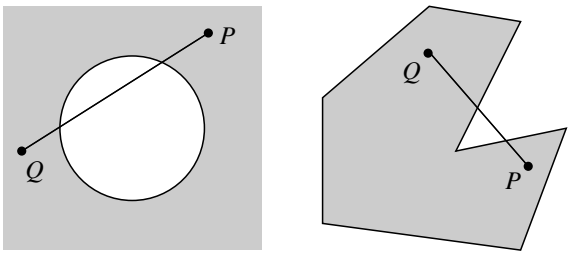
is equidistant from P_1, P_2 and P_3 and is thus the center of the desired circle. The radius of the circle can be computed as $r = d(C,P_1)$. If P_1,P_2,P_3 are collinear, the lines L_{12} and L_{23} are parallel and do not meet in the real plane. Consequently, no circle passes through these three points.

Let a point $P = (h,k)$ and a line $L : ax+by+c = 0$ be given. The locus of all points Q equidistant from P and L is a parabola. P is called the *focus* and L the *directrix* of the parabola. The shaded region in the figure below consists of precisely the points that are closer to P than to L.



Parabola as a locus of points
equidistant from a point and a line

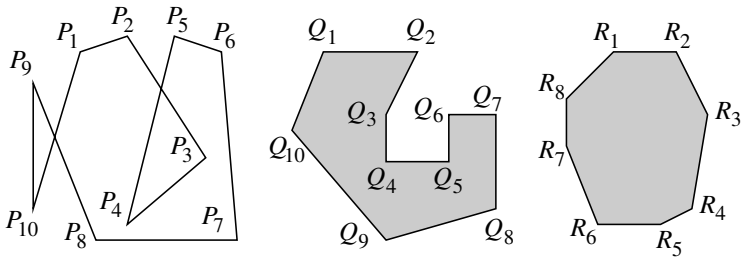
A region X in $\mathbb{R}^2$ is called *convex* if, for any two points $P,Q \in X$, the entire line segment $\overline{PQ}$ lies in X. For example, the entire plane $\mathbb{R}^2$, open and closed half planes, open and closed balls, straight lines and line segments, the *insides* of parabolas are all convex regions. On the other hand, the outside of a circle or a parabola, the Euclidean plane minus a point or minus a straight line are not convex (also see the figure below).



Non−convex regions

A *polygon* is specified by its vertices $P_1,P_2,\ldots,P_n$. Unless otherwise stated, we assume that the vertices $P_1,P_2,\ldots,P_n,P_1$ appear in that order during a closed walk along the polygon (starting at P_1). A polygon is said to be *simple* if no two non-consecutive edges (segments) of the polygon intersect. Examples of simple and non-simple polygons are given in the figure below.

A simple polygon cuts the plane in two regions. The bounded region is called the *interior* of the polygon (shown by the shaded regions in the figure below), whereas the unbounded region is called the *exterior* of the polygon. If the interior of a simple polygon is convex, we call the polygon a *convex polygon*. A polygon is convex if and only if all its interior angles are $\leqslant 180^0$.



(a) non–simple polygon (b) simple polygon (c) convex polygon

The vertices of a convex polygon are usually specified in the clockwise order starting from any arbitrary vertex. A specification of the n vertices of an n-gon requires a storage for $2n$ floating-point values. Any other representation of an n-gon, that needs $\mathrm{O}(n)$ storage space, can also be used. For example, in addition to the coordinates of the vertices, some extra information may be stored. These extra information may speed up some algorithms on convex polygons and/or make analysis of the algorithms easier. Since the input size is n here, we remain happy so long as the additional storage overhead fulfills the $\mathrm{O}(n)$ bound.

Suppose that a convex n-gon $P_1P_2\ldots P_n$ (with vertices listed in the clockwise order) and a point P is given. Our task is to check whether P lies inside the polygon or not. In order to solve this (and many other) problems, let me introduce the concept of *orientation*.

Let L be a straight line passing through two (distinct) points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$. We specify whether the line is oriented from P_1 to P_2 or from P_2 to P_1. An orientation of L is a selection between these two choices.

Let L be oriented from P_1 to P_2. Assume that $P = (h,k)$ is any point in the plane. Consider the determinant

$$\mathrm{side}(P_1,P_2,P) = \det\begin{pmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & h & k \end{pmatrix}. \tag{23.1.3}$$

The point P lies on, to the left, or to the right of the oriented line L according as the determinant $\mathrm{side}(P_1,P_2,P)$ is zero, positive, or negative, respectively.

Back to our old question, that is, whether a point P lies inside a convex polygon $P_1P_2\ldots P_n$. It is evident that P lies inside the polygon if and only if P lies to the right of all of the oriented edges $\overline{P_1P_2}, \overline{P_2P_3}, \ldots, \overline{P_{n-1}P_n}, \overline{P_nP_1}$ of the polygon. The check whether a point lies to the right of an oriented line can be performed by computing a 3×3 determinant. Since an n-gon has n edges, it follows that the check whether P lies inside the polygon can be performed in $\mathrm{O}(n)$ time.

Unfortunately, this is far from optimal. This problem can be solved in $\mathrm{O}(\log n)$ time using a procedure similar to the binary search. However, a naive representation of the polygon using $2n$ coordinates of its vertices does not immediately divulge how this binary search can be performed.

This is our first motivation for studying computational geometry. We need to exercise utmost care on our data structures and algorithms in order to arrive at good solutions to geometric problems.

Before we end this elementary lesson and jump to sophisticated problems, one comment is in order. In geometry, we often talk about objects in *general position*. Depending on the problem at hand, this phrase bears different meanings. For example, if we are given a set of points in general position, we usually mean that no three of the given points are collinear. In addition, we may also mean that no four of the given points lie on a circle, and there is no repetition of a coordinate value in the list. For a collection of lines (or segments) in general position, we typically mean that no two of the given lines are parallel to one another and perhaps also that no given line is parallel to the x- or y-axes. In a demanding situation, we may also require that the end points of the segments do not share any coordinate value and also that a pair of intersecting segments do not intersect at one (or two) endpoints of the segments. As a final example, suppose that we are given a collection of circles in general position. We then know that no two of the circles are concentric, and perhaps some other nice things about the circles.

A collection of randomly chosen objects may be *statistically* expected to be in general position. However, when we solve practical problems, this statistical nicety may be absent. For example, when we are asked to compute the path of a robot from a point to another inside a rectangular room, we may expect that some obstacles in the room have standard geometric shapes (such as squares and rectangles) and are placed aesthetically in the room (say, rectangles with sides parallel to the walls). A collection of objects not is general position is referred to as *degenerate*.

There are several ways to handle degeneracy. For certain kinds of degeneracies, we may convert the given collection to one in general position using suitable coordinate transformations (like translation, scaling and rotation). If the important attributes and properties of the collection are preserved by the transformation, we apply our algorithm(s) in the transformed space, and finally convert the solution back to the original space by the reverse transformation. Unfortunately, this trick is too naive to work in general. A better strategy is to handle degeneracies separately. In other words, we first design an algorithm for objects in general position. We subsequently investigate what kinds of degeneracies may occur and how they affect the algorithm. We then try to patch up our algorithm so that it can handle the degenerate cases. This attempt may increase the complexity of the algorithm in some unavoidable situations. For simplicity, I will avoid discussions on degeneracy in this book.

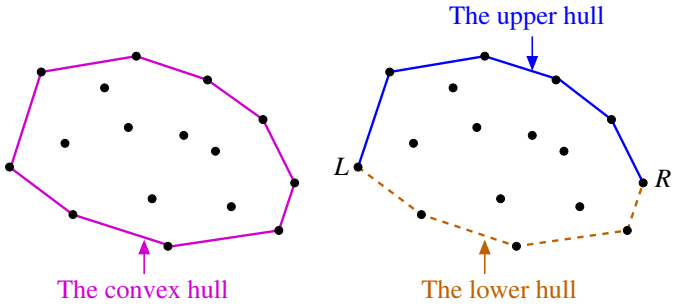
23.2 Convex hulls

We are given a (finite) set points in the plane. Our task is to compute the smallest convex region that contains all these points. It is easy to conceive that this smallest convex region must be a region enclosed by a convex polygon (since the given set of points is finite). We call this convex polygon the *convex hull* of the given set of points.

You may visualize a convex hull as follows. Imagine that a nail is struck on a flat ground at each of the given points. We are given a sufficiently stretchable elastic rubber band. Our task is to place the rubber band in such a way that it encloses all the nails, and the total area enclosed by the stretched band is as small as possible. Figure 85 shows the convex hull of a set of 16 points.

Being a convex polygon, a convex hull can be represented by a sequence of vertices appearing in the clockwise order. We can break this sequence in two parts. Let L and R be the leftmost and

Figure 85: Convex hull of a set of 16 points



rightmost points in this polygon (clearly also in the given set of points). The clockwise listing of vertices on the polygon starting from L and ending at R is called the *upper hull* of the given points. Analogously, the clockwise listing of the vertices of the convex hull starting at R and ending at L is the *lower hull* of the given points. If n points are given, then the convex hull contains $O(n)$ vertices (and edges). It then easily follows that given the convex hull, we can compute the upper and lower hulls in $O(n)$ time. Conversely, given the upper and lower hulls, we can compute the entire convex hull in $O(n)$ time. It, therefore, suffices to compute the upper and lower hulls individually.

Let me introduce some notations. Let S be the given set of points. The convex hull of S is denoted by $\mathrm{CH}(S)$, the upper hull of S by $\mathrm{UH}(S)$ and the lower hull by $\mathrm{LH}(S)$.

In the rest of this section, I describe some well-known algorithms for computing $\mathrm{CH}(S)$ (or equivalently both $\mathrm{UH}(S)$ and $\mathrm{LH}(S)$) for a set S of n points. The most naive strategy takes $O(n^3)$ running time which is far poorer than what we can achieve. I present two $O(n \log n)$ algorithms for computing $\mathrm{CH}(S)$. I end this section by explaining an $O(nh)$ algorithm for computing $\mathrm{CH}(S)$, where h is the number of vertices in $\mathrm{CH}(S)$. We will assume that the points in S are in general position. In this context, this assertion indicates that the x-coordinates of the points in S are (pairwise) distinct, and also that no three of these points are collinear.

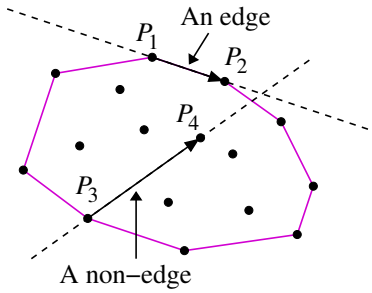
23.2.1 A naive approach

The naive approach first discovers all the edges of the convex hull. Let P, Q be two given points. The oriented segment $\overline{PQ}$ is an edge of $\mathrm{CH}(S)$ if and only if all points in S (other than P, Q) lie to the right of $\overline{PQ}$. Once all the edges are discovered, it is an easy matter to determine a clockwise listing of the vertices in the convex hull. All of the given points lie to the *right* of each edge $\overline{PQ}$. The segment $\overline{QP}$ with the opposite orientation has all the points to its left, and so is not an oriented edge of the resulting convex hull. Figure 86 demonstrates this approach. The edge $\overline{P_1P_2}$ belongs to the convex hull, whereas the edge $\overline{P_3P_4}$ does not.

A high-level description of this naive algorithm follows.

```
Initialize to empty a list L of line segments;
for each P ∈ S {
   for each Q ∈ S\{P} {
```

Figure 86: Explaining the naive algorithm for computing convex hulls




```
        edgeFound = 1;
        for each R ∈ S \ {P,Q} {
            if (side(P,Q,R) > 0) edgeFound = 0;
        }
        if (edgeFound) add PQ to L;
    }
}
Pick an arbitrary edge P1P2 from L;
Initialize CH to the ordered list P1,P2;
Set P = P2;
while (1) {
    Find the edge PQ from L;
    if (Q equals P1) break;
    Insert Q at the end of CH;
    P = Q;
}
return CH;
```

Let us derive the running time of the above algorithm. The triply nested loop at the beginning runs over all triples (P,Q,R) of pairwise distinct points and so takes a running time of $O(n^3)$. (Recall that each computation of side(P,Q,R) takes $O(1)$ time.) The running time of the loop that converts L to CH depends on how L is implemented. If L is implemented as an ordinary linked list, this conversion takes $O(n^2)$ time. Suitable data structures for L reduces this running time to $O(n \log n)$. Nonetheless, this is asymptotically smaller than the running time of the edge discovery phase. To sum up, our naive algorithm runs in $O(n^3)$ time.

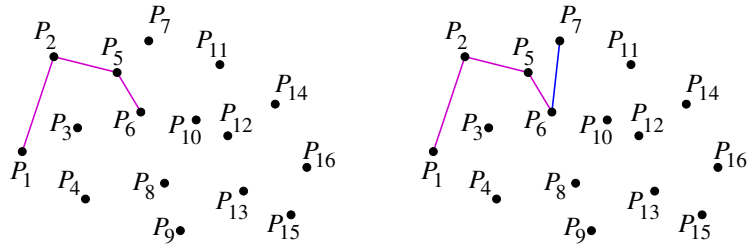
23.2.2 Graham's scan

Now, we discuss another iterative algorithm for computing CH(S) for a set S of n points in the plane. This algorithm computes the upper and lower hulls separately. Because of symmetry, we concentrate on the construction of the upper hull UH(S) only.

First, we sort the points of S in ascending order of their x coordinates. (Under the assumption that the points in S are in general position, there is no repetition of coordinates.) Let $P_1, P_2, \ldots, P_n$ be the sorted list of points. Sorting n points takes $O(n \log n)$ time. We compute the upper hull from this sorted sequence in $O(n)$ time. Thus, we get an $O(n \log n)$-time algorithm for computing $\mathrm{UH}(S)$.

We maintain an ordered list L of points. We initialize the list to the ordered list P_1, P_2. Next, we keep on considering the remaining points P_i for $i = 3, 4, \ldots, n$ (in that order) in a loop. At the beginning of the i-th iteration of the loop, the list L consists of the points of the upper hull of $P_1, P_2, \ldots, P_{i-1}$. This condition is satisfied for $i = 3$. Inside the loop body, we bring the point P_i into consideration, and when the i-th iteration terminates, the list L is updated to the upper hull of $P_1, P_2, \ldots, P_i$. The loop terminates when $i = n + 1$. At that point, L stores the vertices of $\mathrm{UH}(P_1, P_2, \ldots, P_n)$, as desired.

Figure 87: One iteration of Graham's scan



(a) Points $P_1 - P_6$ processed

(b) P_7 added, left turn detected

(c) P_6 deleted, left turn detected

(d) P_5 deleted, iteration complete

It remains to explain the body of the loop that transforms the upper hull of $P_1, \ldots, P_{i-1}$ to the upper hull of $P_1, \ldots, P_i$. The basic idea is illustrated in Figure 87 (with $i = 7$). Being the rightmost among the points considered so far, P_i must be a vertex of the upper hull of $P_1, \ldots, P_i$. However, the introduction of P_i may violate convexity as follows. Let U, V be the last two vertices of L. If the points U, V, P_i (in that order) produce a left turn (or equivalently, if $\mathrm{side}(U, V, P_i) > 0$), the point V cannot be present in the upper hull. So we remove V from the list. The convexity condition may again be violated (See Part (c) of Figure 87). If so, we throw away another point from L. We keep on doing this, until convexity is restored or there is only one point left in L (this must be the leftmost

point P_1). Finally, we add P_i at the end of L and proceed to the next iteration with the next value for i. A high-level description of the loop follows.

```
Initialize to P1,P2 an ordered list L of points;
for (i=3, i<=n; ++i) {
    while (1) {
        if (L contains only one point) break;
        Let U,V be the last two members of L;
        if (side(U,V,Pi) < 0) break;
        Delete the last element V from L;
    }
    Add Pi at the end of the list L;
}
Output the elements of L from beginning to end;
```

In order that the insert and delete operations on L can be performed efficiently (in $O(1)$ time per operation), we maintain L as an array or a doubly linked list. The inner `while` loop may require many (even $\Theta(n)$) deletions in an iteration. But the important point to observe here is that each point can be deleted at most once from L (after it is inserted in L). So a simple amortization argument reveals that preparing the final list $L = \mathrm{UH}(P_1,\ldots,P_n)$ takes a total of $O(n)$ time.

It is worthy to highlight here that the $O(n\log n)$ running time of Graham's scan is attributed to the sorting step and not to the creation of the list L. The natural question that arises is whether a running time of $o(n\log n)$ can be achieved by avoiding sorting altogether. It can, however, be proved (Exercise!) that any algorithm that computes the convex hull of n points in a plane must take $\Omega(n\log n)$ time (as long as the convex hull contains $\Theta(n)$ vertices). In other words, Graham's scan is an optimal algorithm.

23.2.3 Preparata and Hong's divide-and-conquer algorithm

I now describe a recursive algorithm for computing $\mathrm{CH}(P_1,\ldots,P_n)$ that runs in $O(n\log n)$ time. Since sorting n points satisfies the same time bound, we may assume, without loss of generality, that the points $P_1,\ldots,P_n$ are already sorted with respect to their x-coordinates. Moreover, we assume that the points are in general position so that the x-coordinates of $P_1,\ldots,P_n$ form a strictly increasing sequence. Consider the following recursive algorithm.

```
if n ⩽ 3, compute CH(P1,...,Pn) by the naive algorithm and return this hull;
Recursively compute C1 = CH(P1,...,P⌈n/2⌉);
Recursively compute C2 = CH(P⌈n/2⌉+1,...,Pn);
Merge C1 and C2 to C = CH(P1,...,Pn);
Return C;
```

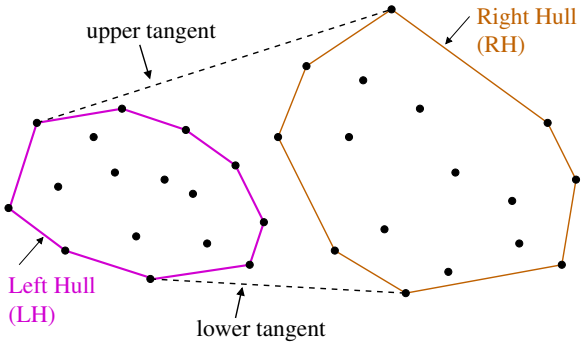
Let $T(n)$ be the running time for this recursive algorithm. Assume that C_1 and C_2 can be merged in $O(n)$ time. We then have

$$T(n) = T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor) + O(n) \text{ for } n > 3.$$

We have seen that this divide-and-conquer recurrence has the solution $T(n) = O(n\log n)$.

I need to supply an algorithm that can merge C_1 and C_2 in linear time. Since the points are sorted and have distinct x-coordinates, the left hull C_1 is separated from the right hull C_2 by a vertical strip. In view of this, an idea illustrated in Figure 88 works.

Figure 88: Merging of the left and the right hulls



A line L is called a *tangent* (or a *supporting line*) to a convex polygon if it touches the convex polygon and all vertices of the convex polygon lie on or to one side of L. We plan to compute the upper and the lower tangents common to both C_1 and C_2. We discard from both C_1 and C_2 all the points lying strictly between these two tangents. The remaining points can be easily listed in the clockwise sequence representing the merged hull C.

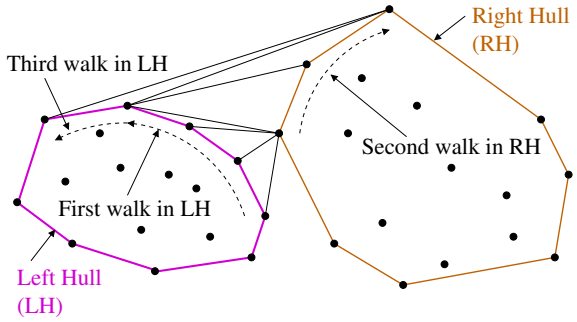
The merging algorithm runs in $O(n)$ time, provided that we can compute the two tangents in $O(n)$ time. Here, I describe an algorithm for the computation of the upper tangent only. The determination of the lower tangent can be symmetrically handled.

Figure 89 illustrates the computation of the upper tangent. We let two points P_1 and P_2 march along C_1 and C_2 respectively. Initially, P_1 is the rightmost point of C_1 and P_2 the leftmost point of C_2. The point P_1 jumps from a vertex in C_1 to the next vertex in the counterclockwise order, whereas P_2 moves in the clockwise order along C_2. At every intermediate instant, we check whether the oriented segment $\overline{P_1P_2}$ is an upper tangent to C_1 and the oriented segment $\overline{P_2P_1}$ is an upper tangent to C_2. This means that we check whether both the neighboring points of P_1 on C_1 lie to the right of $\overline{P_1P_2}$ and both the neighboring points of P_2 on C_2 lie to the left of $\overline{P_2P_1}$. If this condition is not satisfied at P_1 (resp. P_2), we move P_1 (resp. P_2) to the next vertex in the counterclockwise (resp. clockwise) direction along C_1 (resp. C_2). Eventually, P_1 and P_2 reach the vertices defining the upper tangent. As Figure 89 illustrates, the vertices defining the common tangent need not be the topmost vertices of the two hulls (see the left hull).

The following code snippet summarizes the algorithm for computing the upper tangent.

```
Initialize P1 to the rightmost point of the left hull C1;
Initialize P2 to the leftmost point of the right hull C2;
while (P1P2 is not a common tangent to C1 and C2) {
    while (P1P2 is not an upper tangent of C1)
        advance P1 to the next (counterclockwise) vertex of C1;
    while (P2P1 is not an upper tangent of C2)
        advance P2 to the next (clockwise) vertex of C2;
}
Return (P1,P2);
```

Figure 89: Computing the upper tangent



Although intuitively clear, it demands a formal proof to settle that the above algorithm correctly computes the upper tangent, that is, the moving points P_1 and P_2 do not overshoot the respective vertices of tangency on C_1 and C_2. I leave the proof to the reader as an exercise and concentrate on the running time of the algorithm.

Let h_1 and h_2 be the numbers of vertices in C_1 and C_2 respectively. Although we have a nested loop in the code, every step in the walk advances either P_1 or P_2. Therefore, after at most $h_1 + h_2$ steps, the walk stops. Since $h_1 + h_2 \leqslant n$, we conclude that the upper tangent can be computed in $\mathrm{O}(n)$ time.

23.2.4 Jarvis's march

Let C be the convex hull of a set S of n points in the plane. Suppose h is the number of vertices in C. Irrespective of the value of h, our naive algorithm takes $\mathrm{O}(n^3)$ time, whereas Graham's scan and Preparata and Hong's algorithm take $\mathrm{O}(n \log n)$ time. Now, I discuss another algorithm for computing C, that runs in $\mathrm{O}(nh)$ time. In general, $h = \mathrm{O}(n)$, so the running time of this new algorithm is poor (namely $\mathrm{O}(n^2)$). However, if h is small, in particular, if $h = \mathrm{o}(\log n)$, then we have an $\mathrm{o}(n \log n)$ algorithm. An algorithm whose running time depends on the (representation) complexity of the output is called *output-sensitive*. Such algorithms are often useful in computational geometry.

Figure 90: An iteration of Jarvis's march

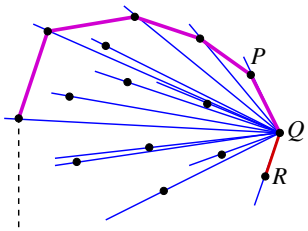


Figure 90 illustrates the working of the new algorithm known as *Jarvis's march*. It starts growing the hull from the leftmost point in S (this point can be determined in linear time). During each iteration, the algorithm discovers the next vertex in the hull. Suppose that during the beginning of some iteration, the convex hull has grown until the vertex Q. Let P be the vertex on the hull immediately before Q. Let L denote the oriented line $\overline{PQ}$. If P happens to be the first vertex (this corresponds to the first iteration), we take L to be the vertical line passing upward through P.

In order to compute the vertex R that follows Q in the hull, we compute the angles from $\overline{PQ}$ to all points of S. The point R that has the smallest angle with $\overline{PQ}$ is the desired next vertex. The following pseudocode implements Jarvis's march.

```
Determine the leftmost point P_1 in S;
Initialize a list C of points to contain the only point P_1;
Let L be the line passing through P_1 and oriented upward;
while (1) {
    θ_min = 180^0;
    for each point Q' of S {
        θ = the angle between PQ and QQ';
        if (θ < θ_min) {
            θ_min = θ; R = Q';
        }
    }
    if (R is the same as P_1) break;
    Add R to the end of the list C;
    Let L be the directed line QR;
}
return C;
```

The `while` loop is executed exactly h times, since each iteration of the loop discovers a new edge of the hull. In each iteration, the algorithm computes $n-1$ angles. There is no need to sort these angles; finding only the minimum of these angles suffices and can be accomplished in $O(n)$ time. Thus, Jarvis's march runs in $O(nh)$ time, as claimed earlier.

23.3 The sweep paradigm

The sweep paradigm has been proved to be a very useful method for solving several geometric problems. In the sweep method, a geometric object (like a line, ray, plane or rectangle) sweeps continuously from one position to another following a specific route. During the movement of the sweeping object, several events occur. An event is a geometric phenomenon that has the potential of determining the output of the algorithm. Some or all of these events contribute to the generation of the desired output.

There is an infinite (in fact, uncountable) number of positions of the sweeping object between its start and end positions. It is impossible to consider all these positions. Instead, we fix the sweeping object only at the positions at which the events occur. We handle each event individually so as to extract the information relevant for generating the output. If the number of events is finite, our algorithm terminates eventually.

This description of the sweep paradigm is too general to make sufficient sense. So I am getting ready to supply you a couple of examples illustrating the paradigm. In the section on Voronoi diagrams, a more complicated example will be provided.

23.3.1 Line segment intersection

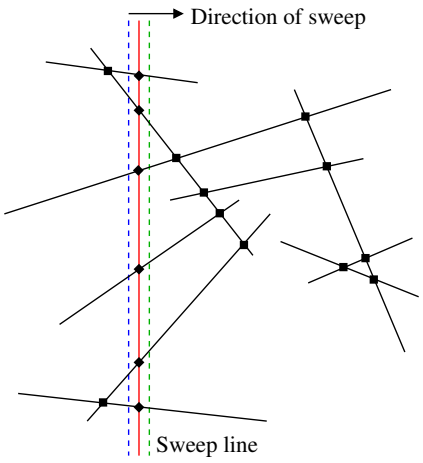
We are given a set of n line segments $L_1, \ldots, L_n$ specified by their end points. Our task is to determine all the points of intersection of these segments. We assume that the lines are in general position. In this context, this means the following.

- No two of the given segments are parallel (or collinear).
- No given segment is parallel to the y-axis.
- The end points and the points of intersection of the given segments have pairwise distinct x-coordinates.

We know that the intersection point of two line segments can be computed in $O(1)$ time. Thus, considering each pair (L_i, L_j), $i \neq j$, of segments gives us a simple algorithm for solving the line-segment intersection problem. Since the number of pairs of segments is $\binom{n}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$, this algorithm takes $\Theta(n^2)$ running time.

Let h be the actual number of intersection points for the given set. Since h may be as large as $\Theta(n^2)$, the above algorithm is optimal (reporting the h intersection points itself takes $\Theta(n^2)$ time). However, think of a situation when h is small. In this case, we hope to do better than spending $\Theta(n^2)$ effort. Now, I describe an $O((n+h)\log n)$-time algorithm for this problem. If $h = O(n)$, this running time is $O(n \log n)$. This algorithm is based on the sweep paradigm.

Figure 91: Line segment intersection



A vertical line L (that is, a line parallel to the y-axis) sweeps from $x = -\infty$ to $x = +\infty$ (see Figure 91). Let the left and right end points of L_i be P_i and Q_i respectively. Also let $P_{\text{left}} = \min(x(P_1), \ldots, x(P_n))$ and $Q_{\text{right}} = \max(x(Q_1), \ldots, x(Q_n))$ be the smallest and largest x-coordinates of the end points. As long as the sweep line L lies to the left of P_{left}, no relevant phenomenon occurs, that may lead to the discovery of an intersection point. Similarly, when the sweep line leaves Q_{right} and proceeds rightward, it has already visited all the intersection points, and there is nothing more to do. Therefore, it suffices to watch the movement of L between P_{left} and Q_{right}. Still, there are infinitely many positions for L to check. Luckily, all these positions of L are not important for the discovery of intersection points.

Throughout the sweep of L, we maintain a list of the segments that intersect L. We store the segments in a sorted order (say, from top to bottom) of the intersection points. The relative order of the segments intersecting L is important rather than their exact points of intersection with L. Indeed, it is not necessary to store the y-coordinates of the intersection points of the segments with L.

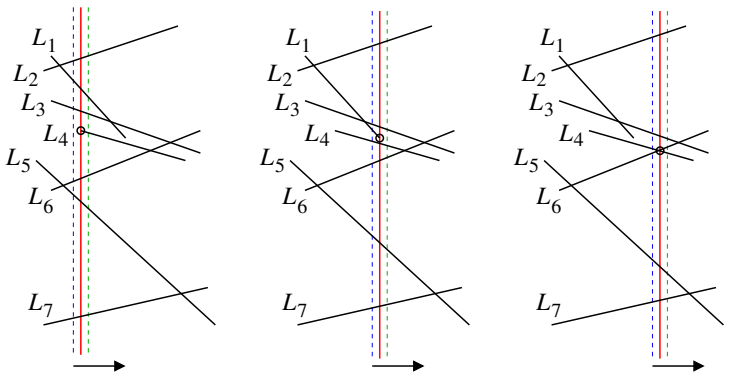
Figure 91 shows a *general* position of the sweep line. It also shows the positions of the sweep line a little before and a little after the current position (see the dotted lines). Notice that the positions of the intersecting segments relative to the sweep line do not change during this span of the movement of L. In other words, no event occurs during this span.

The information stored against the sweep line changes if and only if one of the following events occurs.

- *Enter segment:* The sweep line L meets the left end point of a new segment.
- *Leave segment:* L leaves a segment that has been intersecting L for some time in the past.
- *Intersection point:* L reaches an intersection point of two given line segments.

These three events are illustrated in Figure 92. The positions of the sweep line immediately before and immediately after the events are shown by dotted vertical lines. These lines indicate what changes we need to make in the sweep-line information in order to attend different events.

Figure 92: Events in the line sweep algorithm for line segment intersection



(a) Enter segment (b) Leave segment (c) Intersection point

We maintain an *event queue* for storing the x-coordinates of the events that are yet to be processed, that is, those that lie to the right of the current position of the sweep line. Since each end point of each given segment yields an event, we initialize the event queue by the $2n$ end points $P_1,\ldots,P_n,Q_1,\ldots,Q_n$. Intersection points also lead to events. But we do not make an attempt to insert all intersection points in the queue initially. (That will, anyway, defeat our original intention of achieving an output-sensitive algorithm.) We will gradually insert these points in the queue as the sweep line progresses rightward.

Let L_i and L_j be two given segments that are currently intersecting with L. Suppose that these intersection points lie consecutively in the list of intersection points on L. Finally, suppose that L_i and L_j intersect at a point that lies to the right of the current position of L. This is the only situation where the (x-coordinate of the) intersection point of L_i and L_j resides in the event queue. For brevity, we use the notation $\bigcap(L_i,L_j)$ to denote the point of intersection of L_i and L_j (or the x-coordinate of the point).

The event queue is kept sorted in the increasing order. As long as the queue is not empty, we look at the next event, delete it from the queue and modify the sweep line information and the event queue appropriately.

Part (a) of Figure 92 describes an *enter segment* event, where L meets the left end point of L_4. We insert L_4 between L_3 and L_6 in the sweep line. Just before this event, L_3 and L_6 were adjacent on the sweep line, but are not so now. So we delete $\bigcap(L_3,L_6)$ from the event queue. Now, (L_3,L_4) and (L_4,L_6) are adjacent pairs. L_3 and L_4 do not intersect, whereas L_4 and L_6 intersect to the right of L. So we insert $\bigcap(L_4,L_6)$ in the event queue.

In Part (b) of Figure 92, L leaves the right end point of L_1. This exposes L_3 and L_4 as new neighbors. Consequently, we need to look at $\bigcap(L_3,L_4)$ for possible insertion in the event queue. In our example, the segments L_3 and L_4 do not intersect.

In Part (c) of Figure 92, the sweep line meets the intersection point of L_4 and L_6. After this intersection, the positions of the two segments L_4 and L_6 are swapped on the sweep line. Moreover, the neighbor of L_3 changes from L_4 to L_6, and the neighbor of L_5 changes from L_6 to L_4. So the intersection points $\bigcap(L_3,L_4)$ and $\bigcap(L_5,L_6)$ (if existent and to the right of L) are deleted from the event queue, whereas $\bigcap(L_3,L_6)$ and $\bigcap(L_5,L_4)$ (if existent and to the right of L) are inserted in the event queue.

```
Initialize E (the event queue) to P_1,...,P_n,Q_1,...,Q_n in sorted order;
Initialize S (the sweep line information) to empty;
while (E is not empty) {
    Pick up the next event (the event with smallest x-coordinate) from E;
    if it is an enter segment event {
        Suppose that the left end point of L_i has triggered the event;
        Insert L_i in the appropriate position of S (kept sorted);
        Let L_s be the immediate predecessor of L_i in S;
        Let L_t be the immediate successor of L_i in S;
        If ⋂(L_s,L_t) exists and lies to the right of L, delete ⋂(L_s,L_t) from E;
        If ⋂(L_s,L_i) exists and lies to the right of L, insert ⋂(L_s,L_i) in E;
        If ⋂(L_i,L_t) exists and lies to the right of L, insert ⋂(L_i,L_t) in E;
    } else if it is a leave segment event {
        Suppose that the right end point of L_i has triggered the event;
```

```
        Let L_s be the immediate predecessor of L_i in S;
        Let L_t be the immediate successor of L_i in S;
        Delete L_i from S;
        If ⋂(L_s,L_t) exists and lies to the right of L, insert ⋂(L_s,L_t) in E;
    } else if it is an intersection point event {
        Suppose that ⋂(L_i,L_j) has triggered the event;
        Print ⋂(L_i,L_j);
        Let L_s be the immediate predecessor of L_i in S;
        Let L_t be the immediate successor of L_j in S;
        Interchange L_i and L_j in S;
        If ⋂(L_s,L_i) exists and lies to the right of L, delete ⋂(L_s,L_i) from E;
        If ⋂(L_t,L_j) exists and lies to the right of L, delete ⋂(L_t,L_j) from E;
        If ⋂(L_s,L_j) exists and lies to the right of L, insert ⋂(L_s,L_j) in E;
        If ⋂(L_i,L_t) exists and lies to the right of L, insert ⋂(L_i,L_t) in E;
    }
}
```

In the above code, I have assumed, for the sake of simplicity, that the predecessor L_s and the successor L_t (in S) always exist. This need not be true in all cases. The reader is urged to fill out the trivial details to handle the general situation.

Let us now look into the organization of the event queue E and the sweep line information S so as to achieve efficient operations on them. The basic operations required on E are insertion, deletion, and finding and deleting minimum elements. On the other hand, S should support insertion, deletion, interchanging two consecutive elements, and finding the predecessors and successors of elements. In view of this, we represent each of S and E as a balanced binary search tree (like AVL tree). Notice that the deletion of an arbitrary element from a heap is, in general, costly. So we avoid using a heap (that is, a priority queue) to implement E.

E stores those $2n$ end points and some of the h intersection points, that lie to the right of the current position of the sweep line. The sorting is with respect to the x-coordinates of the points. In addition to these coordinates, we need also to store a reference to the line segment(s) against each coordinate. This means that against each end point (its x-coordinate actually), we should also store to which segment this end point belongs and also whether it is a left end point or a right end point. Similarly, for an intersection point in E, we need to store the references of the two line segments having this intersection point. There are at most $2n$ end points to the right of any position of L. Moreover, an intersection point is present in E only if it corresponds to two consecutive segments on L. Therefore, the maximum size of E is $2n+(n-1)$. A balanced binary search tree storing these events has a height of $\mathrm{O}(\log n)$.

The other structure S stores references to the line segments that currently intersect with L. The list is kept sorted in the decreasing order of y-coordinates of the intersection points of L with the active segments. We do not need to store these y-coordinates explicitly. First, after every movement (even if infinitesimal) of L, these coordinates change. Second, the relative positions of the active segments with respect to L are important, not their exact locations.

A problem, however, arises in this context. Let us think about an insertion operation in S necessitated by an *enter segment* event. A new segment now comes to the picture. We know the y-coordinate of its left end point. But we must also know the relative position of this y-coordinate

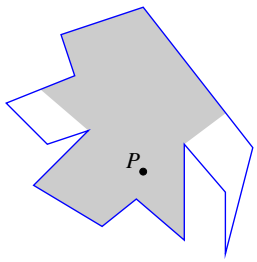
with respect to the current intersection points of L with the active segments. In order to tackle this situation, we keep on *computing* the y-coordinates of the intersection points of the active segments with L, as we traverse down the tree representing S. First, we compute the exact point of intersection of L with the segment at the root of the tree. Depending on whether this point is above or below the left end point of the segment to be inserted, we choose an appropriate sub-tree, and once again compute the exact point of intersection of L with the root of this sub-tree. Proceeding in this way, we eventually reach the correct insertion position. This search path is of logarithmic height, that is, we need to compute at most $O(\log n)$ intersection points. Since each intersection point can be computed in $O(1)$ time, the total insertion cost continues to remain as $O(\log n)$.

Let us now investigate the overall complexity of the line-sweep algorithm. Initially, the $2n$ end points need to be inserted in the balanced binary search tree E. This takes $O(n \log n)$ time. Subsequently, the sweep line encounters a total of $2n+h$ events. Each event can be handled in $O(\log n)$ time under the representation of E and S, discussed above. Thus, the line-sweep algorithm runs in $O((n+h)\log n)$ time. The space complexity of the line sweep algorithm is $O(n)$.

23.3.2 Visibility polygon

A person P (considered as a point) is sitting on a chair in the interior of a closed room whose boundary is a simple polygon. The walls are assumed to be opaque. The person P can rotate in his revolving chair, but cannot move from one place to another. The question is to determine what area of the room the person can see. Figure 93 illustrates this situation.

Figure 93: Region of visibility from P

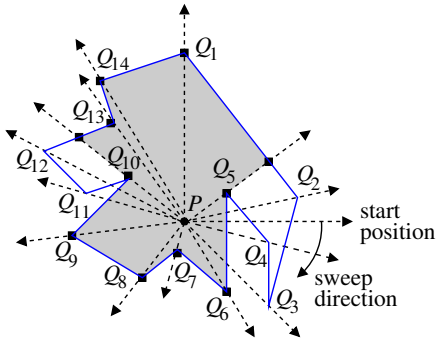


If the room in question is convex, the entire interior of the room is visible from P. If not, the region of visibility—yet another simple polygon—may be a strict subset of the interior of the room. Our task is to determine this visibility polygon. We assume that the room is provided as a clockwise list of the vertices of its boundary. Here *clockwise* means that as we traverse the boundary following the list, the interior always remains to our right.

We use a sweep algorithm to solve this problem. Here, a ray emanating from P rotates clockwise starting from the horizontal right position. As the ray makes a complete rotation of 360^0, it gathers sufficient information for determining the visibility polygon. However, there are infinitely many positions of the ray. We cannot take care of all these positions. This is not necessary, anyway. We instead consider only the specific positions at which some potentially important changes occur

during the sweep of the ray. Figure 94 illustrates these event points. It is evident from the figure that it suffices to consider only those positions of the sweeping ray, at which the ray meets the vertices of the boundary polygon. For simplicity, we assume that the vertices of the boundary polygon are in general position. Here, this means that P is not collinear with two (or more) vertices of the polygon.

Figure 94: Event positions for the sweeping ray



We maintain two data structures for an efficient execution of the algorithm. The first is an event queue that holds the angular positions of the unvisited vertices of the boundary polygon. All angles are calculated from the initial position of the ray in the clockwise sense and have values between 0 and 360 degrees. The event queue is initialized by these angular positions of all the vertices of the boundary. As the sweep ray encounters vertices in the increasing order of their angular positions, the vertices are removed one by one from the queue. When all the vertices are encountered, the event queue becomes empty and the algorithm terminates.

We also maintain a list S of edges of the boundary of the room, with which the sweep ray intersects. We call these edges *active* at the current position of the ray. The active edges are kept sorted in the increasing order of the distances between P and the intersection points of the active edges with the ray. We do not store the exact values of these distances. The relative order of intersection of the active edges with the sweep ray is important in this context.

As soon as the sweep ray hits a vertex Q of the room boundary, an event occurs. During each event we determine zero, one or two vertices of the output visibility polygon. Let Q_- and Q_+ be the vertices respectively previous and next to Q on the room boundary. There are four different types of events. The type of an event is determined by the relative positions of Q_- and Q_+ with respect to the ray $\overline{PQ}$. The types are summarized in the following table.

	Direction with respect to $\overline{PQ}$	
Type of event	**of Q_-**	**of Q_+**
LR	left	right
RL	right	left
LL	left	left
RR	right	right

Let me now describe how to handle these four types of events. Let us denote the edge Q_-Q by e_-

and the edge QQ_+ by e_+. We treat these edges as non-oriented, at least as long as their inclusion in S is concerned.

Event of type LR

In this case, the sweep ray leaves the edge e_- and enters the edge e_+. We check in S whether e_- happens to be the closest to P among all active edges. If so, we output Q as the next vertex of the visibility polygon. If not, we do not output any vertex. In both the cases, e_- is deleted from S and e_+ is inserted in S.

For example, this event occurs at the vertices Q_9 and Q_{12} in Figure 94. The edge Q_8Q_9 was closest to the sweep ray immediately before the event Q_9 occurs, so Q_9 is output as a vertex of the visibility polygon. Moreover, Q_8Q_9 is deleted from S and Q_9Q_{10} is added to S. During the event Q_{12}, on the other hand, the edge $Q_{11}Q_{12}$ is not closest to P in S. So Q_{12} is not output as a vertex of the visibility polygon. However, $Q_{11}Q_{12}$ is deleted from S and $Q_{12}Q_{13}$ is added to S.

Event of type RL

Here, the sweep ray leaves the edge e_+ and enters the edge e_-. This case looks symmetric to the case *LR*, but there is a subtle difference here in that e_- cannot be closest to P along the sweep ray immediately before the event Q occurs (for if e_- were closest to P, traversing the boundary from Q_- to Q_+ via Q would leave the interior of the room to the left of the direction of traversal). We only remove e_+ from S and add e_- to S.

This event occurs at Q_4 in Figure 94. We delete Q_4Q_5 from S and add Q_3Q_4 to S.

Event of type LL

In this case, the ray leaves both the edges e_- and e_+. If neither of these edges is closest to P along the sweep ray immediately before the event occurs, we delete these two edges from S and proceed to the next event. Otherwise, we output Q as the next vertex of the visibility polygon, and then delete e_- and e_+ from S. After the deletion, we determine the closest edge along the ray, and output the intersection of the ray with this closest edge as the next vertex of the visibility polygon.

Consider the event Q_3 in Figure 94. Neither Q_2Q_3 nor Q_3Q_4 was closest to P along the sweep ray just before this event. So we delete these two edges from S. On the other hand, when the event Q_{10} occurs, we delete Q_9Q_{10} and $Q_{10}Q_{11}$ from S and output Q_{10} and the intersection of PQ_{10} with the closest remaining edge $Q_{12}Q_{13}$ as the next two vertices of the visibility polygon.

Event of type RR

In this case, the sweep ray enters both the edges e_- and e_+. We first determine the point R on the room boundary, that is closest to P along the sweep ray. This information can be gathered from S. We then insert e_- and e_+ in S. At this instant, however, both these edges are equidistant from P along the sweep ray. So we plan to be futuristic. Using the other end points Q_- and Q_+ (or, equivalently, considering the angles PQQ_- and PQQ_+), we determine which of the edges will be closer to P along the sweep ray in the immediate future.

After the insertion of the two edges, we check whether Q is closer to P than the point R. If so, we output R and Q (in that sequence) as the next two vertices of the visibility polygon. Otherwise, we output no vertex.

Once again, look at Figure 94. An event of type *RR* occurs at Q_5. Since the angle PQ_5Q_4 is larger than the angle PQ_5Q_6, the edge Q_4Q_5 is considered more distant from P than the edge Q_5Q_6,

when these two edges are inserted in S. After the insertions, Q_5Q_6 is closer to the previously closest edge Q_1Q_2. So the intersection of Q_1Q_2 and PQ_5 is output as a vertex of the visibility polygon. This is followed by Q_5 as the next output vertex.

An event of type *RR* occurs also at Q_{11}. We insert $Q_{10}Q_{11}$ and $Q_{11}Q_{12}$ in S, and $Q_{10}Q_{11}$ is treated closer to P than $Q_{11}Q_{12}$ during the insertion. No vertices are output in this case, since Q_{11} is more distant from P than the current intersection of the sweep ray with the room boundary.

That completes the description of how to handle each of the four types of events. I leave a high-level description of the algorithm to the reader. Let me instead highlight how the event queue and the sweep ray information should be maintained so as to handle the events efficiently.

The event queue should support arbitrary insertion and deletion of minimum only. So a heap (priority queue) is well suited to implement the event queue. The sweep ray information S, on the other hand, must support arbitrary insertions and deletions and also minimum extractions. As a result, S should be realized as a height-balanced search tree (like the AVL tree).

Let n be the number of vertices in the boundary polygon. The event queue stores a maximum of n vertices. So each insertion or deletion of minimum can be performed in $O(\log n)$ time. In particular, initializing the event queue takes $O(n \log n)$ time. Also S stores a list of edges of the boundary, and so the number of elements stored in S is always $O(n)$. Each event can be handled in $O(\log n)$ time, since doing so involves $O(1)$ operations on the event queue and on the sweep ray structure S. Finally, there are exactly n events, each corresponding to a vertex in the boundary. To sum up, the above ray sweep algorithm runs in $O(n \log n)$ time and requires $O(n)$ space.

23.4 Voronoi diagrams

Let $S = \{P_1, P_2, \ldots, P_n\}$ be a set of n points in the plane. Imagine that the points represent grocery shops in a city. Assume that the shops sell items at identical prices and also that all the shops offer equally horrible customer services. In that situation, the only criterion for an inhabitant to select a shop for marketing is geographic proximity. In other words, a shop P_i attracts customers residing at a point P if and only if P_i is closer to P than any other shop P_j, $j \neq i$. Here, the distance between two points is measured by the standard Euclidean metric. This is not perfectly practical, since human beings usually walk or drive along roads, and the geometry of the roads in the city does have a bearing in determining the closest shop. However, you may assume that this is a fairy tale, and human beings have wings. When a person plans to move from a point to another, (s)he flies along the line segment joining the source and the destination points.

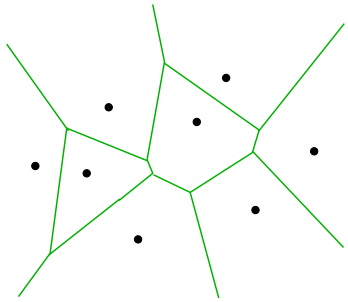
Our task is clear now. For each i, we need to determine the points P in the plane such that $d(P,P_i) \leqslant d(P,P_j)$ for all $j \neq i$. The region consisting of all such points is called the *Voronoi cell* for P_i. We denote this by $\mathrm{VCell}(P_i)$. For a pair (i,j) with $i \neq j$, let H_{ij} denote the closed half plane comprising points not more distant from P_i than from P_j, that is, $H_{ij} = \{Q \mid d(Q,P_i) \leqslant d(Q,P_j)\}$. We clearly have

$$\mathrm{VCell}(P_i) = \bigcap_{\substack{j=1 \\ j \neq i}}^{n} H_{ij}\,.$$

Since each H_{ij} is convex, we conclude that $\mathrm{VCell}(P_i)$ is convex too. Moreover, this cell is the intersection of a finite number of closed half planes. We assume that the points of S are in general

position and $n \geqslant 3$. It follows that each $\mathrm{VCell}(P_i)$ is either a convex polygonal region or a convex region enclosed by line segments and two semi-infinite lines. Figure 95 shows the Voronoi cells for 8 points in the plane.

Figure 95: Voronoi diagram of a set of eight sites



The subdivision of $\mathbb{R}^2$ into n Voronoi cells $\mathrm{VCell}(P_i)$, $i = 1, \ldots, n$, is called the *Voronoi diagram* of S, denoted $\mathrm{Vor}(S)$. The diagram is specified by the vertices and the edges of the Voronoi cells. The given points P_i are called *sites* (since they are the sites of the grocery shops in the city). The vertices and edges of the Voronoi cells of the sites are called vertices and edges of $\mathrm{Vor}(S)$. Note that each edge of $\mathrm{Vor}(S)$ is either finite or semi-infinite.

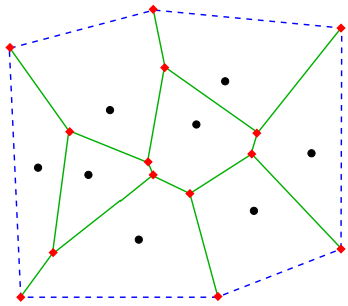
In this section, I describe two algorithms for computing the Voronoi diagram of a set S of n points. Both these algorithms run in $\mathrm{O}(n \log n)$ time. In order the make the exposition simple and compact, I omit some long rigorous proofs of geometric facts, and rely on the intuition of the readers. Cynic students are urged to fill out the missing details using rigorous mathematical arguments.

Notice that $\mathrm{VCell}(P_i)$ may share an edge with each of the other cells $\mathrm{VCell}(P_j)$, $j \neq i$. Thus, a cell may contain as large as $n-1$ edges. Since there are n sites, the total number of edges in $\mathrm{Vor}(S)$ must be $\leqslant n(n-1)/2 = \mathrm{O}(n^2)$. If this number is $\Omega(n^2)$ too, it is out of question to arrive at an $\mathrm{O}(n \log n)$ time bound for any algorithm for computing $\mathrm{Vor}(S)$ (since any such algorithm must output all the edges of $\mathrm{Vor}(S)$). The upper bound $\mathrm{O}(n^2)$ on the size of $\mathrm{Vor}(S)$ is indeed not tight. The size is actually $\Theta(n)$. I will prove this assertion using a property of planar graphs. An independent proof is given as Exercise 5.244.

We construct an undirected graph G from $\mathrm{Vor}(S)$. Note that $\mathrm{Vor}(S)$ contains semi-infinite edges and so does not immediately represent a graph. We consider a finite region (say, polygonal) big enough so that only the semi-infinite edges of $\mathrm{Vor}(S)$ intersect with the boundary of the region (see Figure 96). The boundary of the enclosing region is shown by dotted segments.

The vertices of $\mathrm{Vor}(S)$ and all the points of intersection of the boundary of the enclosing region with the semi-infinite edges of $\mathrm{Vor}(S)$ constitute the vertex set for G. The edge set of G comprises all finite edges of $\mathrm{Vor}(S)$, the finite segments of the semi-infinite edges of $\mathrm{Vor}(S)$ inside the enclosing region, and the lines joining consecutive vertices in the boundary of the enclosing region. Let V and E denote the number of vertices and edges of this graph G. We assume that no three points of S are collinear. This implies that no two edges of $\mathrm{Vor}(S)$ are parallel, and consequently the graph G is connected.

Figure 96: Size of a Voronoi diagram



Such a graph G is called *planar*, since the graph can be drawn in the plane (it is already drawn) with no non-trivial intersections of edges. Removal of the graph (actually, its planar drawing) breaks the plane in several connected regions exactly one of which is infinite. Each such connected piece of $\mathbb{R}^2$ is called a *face* of the graph. Let F be the number of faces of the planar graph G constructed above. We now use a well-known property of connected planar graphs, namely *Euler's formula*:

$$V - E + F = 2\,.$$

Each finite face of G corresponds to a Voronoi cell. There is also an unbounded face (the outside of the enclosing region). Therefore, $F = n+1$. Since no four points in S lie on a common circle, each vertex of G has degree 3, and so, by the degree-sum formula, we have $3V = 2E$. Combining these results gives

$$E = 3E - 2E = 3E - 3V = 3(E-V) = 3(F-2) = 3(n+1-2) = 3n-3\,,$$

and consequently $V = \frac{2}{3}E = 2n-2$. Since V and E are upper bounds on the number of vertices and edges of $\mathrm{Vor}(S)$, the size of $\mathrm{Vor}(S)$ is bounded above by a linear function of n.

23.4.1 Shamos and Hoey's divide-and-conquer algorithm

I now describe a divide-and-conquer algorithm for computing the Voronoi diagram of a set S of n sites in the plane. The complete details of this algorithm are quite complex and frighteningly gory. So I plan to remain happy with only a sketchy overview of the algorithm.

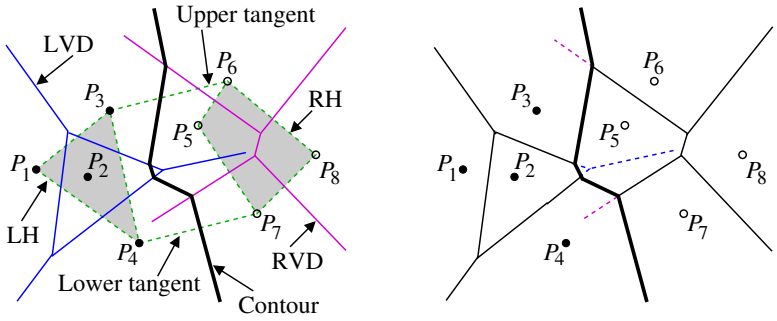
Since n points can be sorted in $O(n \log n)$ time, we first perform this step (only once) and assume that the sites $P_1, \ldots, P_n$ of S are provided in the increasing order of x-coordinates. We break up S in two subsets of (almost) equal sizes: $L = \{P_1, \ldots, P_{\lceil n/2 \rceil}\}$ and $R = \{P_{\lceil n/2 \rceil + 1}, \ldots, P_n\}$. We recursively compute the Voronoi diagrams $\mathrm{Vor}(L)$ and $\mathrm{Vor}(R)$. Finally, we merge the two Voronoi diagrams so as to obtain $\mathrm{Vor}(S)$.

The merging step is evidently the most critical step of the algorithm. We assume that the sites in S are in general position, that is, the sites have pairwise distinct x-coordinates and no four of the sites lie on the same circle. The first assumption implies that L and R are separated horizontally. Moreover, no edge of $\mathrm{Vor}(S)$ is horizontal. The second condition implies that all vertices of $\mathrm{Vor}(S)$ have degree 3.

Both $\mathrm{Vor}(L)$ and $\mathrm{Vor}(R)$ are subdivisions of the entire plane $\mathbb{R}^2$. So too is $\mathrm{Vor}(S)$. In other words, while merging the left and right diagrams, each point in $\mathbb{R}^2$ comes either from $\mathrm{Vor}(L)$ or from $\mathrm{Vor}(R)$. The basic step of the merging process is the computation of a piece-wise linear curve called the *contour*. The contour starts and ends with semi-infinite lines and contains, between these lines, a connected set of line segments. Each segment or half line of the contour is along the perpendicular bisector of the segment joining a point of L with a point of R.

We start generating the contour from bottom to top. We first determine the half line at the bottom and move along the line until we discover another segment in the diagram. We then leave the half line and start moving along the new segment. We then discover the next segment in the contour and change our direction of walk to along this segment. This process is repeated until we eventually reach the other half line of the contour extending infinitely upward. Throughout this entire walk along the contour, the y-coordinate increases monotonically. Figure 97 illustrates the merging step. $\mathrm{Vor}(L)$ and $\mathrm{Vor}(S)$ are shown as LVD and RVD. The contour is shown as the bold line.

Figure 97: Merging of Voronoi diagrams



(a) Computation of the contour

(b) The merged diagram

We first determine the semi-infinite lines on the contour. To that end, we compute the convex hulls $\mathrm{LH} = \mathrm{CH}(L)$ and $\mathrm{RH} = \mathrm{CH}(R)$ and subsequently the lower and upper tangents. In our example, these are P_4P_7 and P_3P_6. These two segments are on the convex hull of S. By Exercise 5.240, the perpendicular bisectors of these segments contribute semi-infinite lines in $\mathrm{Vor}(S)$.

We move upward along the perpendicular bisector of P_4P_7. This bisector demarcates $\mathrm{VCell}(P_4)$ and $\mathrm{VCell}(P_7)$ in $\mathrm{Vor}(S)$. Eventually, we meet an edge of either $\mathrm{Vor}(L)$ or $\mathrm{Vor}(R)$. In our example, we meet the perpendicular bisector of P_5P_7. We now need to take the decision whether we are going to enter $\mathrm{VCell}(P_5)$ in $\mathrm{Vor}(R)$ or not. If we do not do so, we remain in $\mathrm{VCell}(P_7)$ in $\mathrm{Vor}(R)$. This means that we walk back along the semi-infinite line along which we came. So we must enter $\mathrm{VCell}(P_5)$ in $\mathrm{Vor}(R)$. The point where the contour meets the perpendicular bisector of P_5P_7 is equidistant from P_4, P_5 and P_7. The points in the interior of $\mathrm{VCell}(P_5)$ in $\mathrm{Vor}(R)$ are closer to P_5 than to P_7. So we forget P_7 and move along the perpendicular bisector of P_4P_5. This leads to the discovery of the point where $\mathrm{VCell}(P_4)$, $\mathrm{VCell}(P_5)$ and $\mathrm{VCell}(P_7)$ meet in $\mathrm{Vor}(S)$. We also discover the edge between $\mathrm{VCell}(P_4)$ and $\mathrm{VCell}(P_5)$ in $\mathrm{Vor}(S)$. The new vertex discovered is of degree three: two

edges are determined by the contour, the third by the segment separating VCell(P_5) and VCell(P_7) in Vor(R).

Movement along the discovered edge lets the contour meet the boundary between VCell(P_2) and VCell(P_4) in Vor(L). We enter VCell(P_2) (where else can we go?) thereby relinquishing proximity from P_4. Another vertex of Vor(S) is thus discovered, and the contour now changes direction to along the perpendicular bisector of P_2P_5.

We eventually leave VCell(P_2) and enter VCell(P_3) of Vor(L) along the perpendicular bisector of P_3P_5. Thus, yet another vertex and another edge in Vor(S) are discovered.

Finally, we leave VCell(P_3) in Vor(R) and proceed along the perpendicular bisector of P_3P_6 in VCell(P_6) of Vor(R). But P_3P_6 is the upper tangent. Thus, we have already reached the final semi-infinite line of the contour. The contour does not change direction any more and continues along this line infinitely upward.

After the contour is fully determined, the part of Vor(L) to the right of the contour is discarded, so also is the part of Vor(R) to the left of the contour. The resulting merged diagram is shown in Part (b) of the above figure. The portions discarded from Vor(L) and Vor(R) are shown by dotted lines.

This example illustrates the basic procedure outlined below.

```
Determine the lower and upper tangents;
Initialize the contour to the perpendicular bisector of the lower tangent;
Let ℓ be the last segment (or half line) discovered in the contour;
Initialize ℓ to the perpendicular bisector of the lower tangent;
Initialize y_curr to −∞;
while (ℓ is not the perpendicular bisector of the upper tangent) {
    Let ℓ be the perpendicular bisector of P_iP_j, P_i ∈ L, P_j ∈ R;
    Determine the first intersection Q of ℓ above y_curr
              with a segment ℓ′ in Vor(L) or Vor(R);
    if (ℓ′ belongs to Vor(L)) {
        Let ℓ′ be the perpendicular bisector of P_iP_i′;
        Add the intersection point Q to the contour;
        Add the perpendicular bisector ℓ″ of P_i′P_j to the contour;
    } else {
        Let ℓ′ be the perpendicular bisector of P_jP_j′;
        Add the intersection point Q to the contour;
        Add the perpendicular bisector ℓ″ of P_iP_j′ to the contour;
    }
    Update y_curr to the y-coordinate of Q;
    Also update ℓ to ℓ″;
}
Discard the portion of Vor(L) lying to the right of the contour;
Discard the portion of Vor(R) lying to the left of the contour;
```

As I intim(id)ated the reader earlier, the proof that the above algorithm correctly merges two horizontally separated Voronoi diagrams is complicated. Interested students may look at textbooks on computational geometry. Let me now investigate the running time of the merging step.

Since the points of S are sorted, one can run Graham's scan to compute the left and right hulls in $O(n)$ time. The two tangents can also be computed in $O(n)$ time (see Preparata and Hong's divide-and-conquer algorithm for computing convex hulls). The `while` loop of the above code snippet can be implemented to run in $O(n)$ time. In order to achieve this, we need to maintain sophisticated data structures, a study of which is well beyond the scope of this book. We also make use of the fact that both $\mathrm{Vor}(L)$ and $\mathrm{Vor}(R)$ have sizes bounded by linear functions of n. The final step of discarding portions from the two sub-diagrams can, therefore, be implemented in $O(n)$ time too. If $T(n)$ denotes the running time for computing $\mathrm{Vor}(S)$, we have

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n).$$

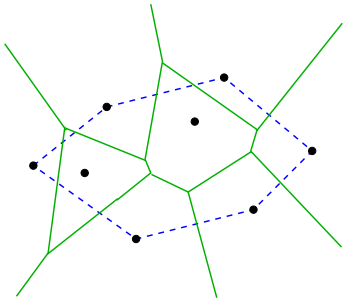
We know that this recurrence has the solution $T(n) = O(n \log n)$.

23.4.2 Fortune's line sweep algorithm

Steven Fortune's line sweep algorithm is an iterative way to compute Voronoi diagrams in the plane. This algorithm runs in $O(n \log n)$ time. I hope to supply a more concrete treatment of this algorithm than the above divide-and-conquer algorithm, since the sweep algorithm, although somewhat complicated, is reasonably intuitive and is not too demanding regarding the use of data structures. As usual, we plan to compute $\mathrm{Vor}(S)$, where $S = \{P_1, \ldots, P_n\}$ is a set of n sites in the plane. Fortune's algorithm requires the sites to be in general position. In this context, this means that no two of the sites have the same x-coordinate or the same y-coordinate, no three of the sites are collinear, and no four of the sites lie on a common circle.

Fortune's algorithm detects the finite vertices of $\mathrm{Vor}(S)$. In order to handle the semi-infinite lines, we start by computing the convex hull of S. This precomputation can be done in $O(n \log n)$ time using Graham's scan or the Preparata–Hong algorithm. The relation between $\mathrm{CH}(S)$ and $\mathrm{Vor}(S)$ is that two Voronoi cells $\mathrm{VCell}(P_i)$ and $\mathrm{VCell}(P_j)$ share a semi-infinite line if and only if P_iP_j is an edge of $\mathrm{CH}(S)$ (Exercise 5.240). Figure 98 illustrates this phenomenon.

Figure 98: Voronoi diagram and convex hull



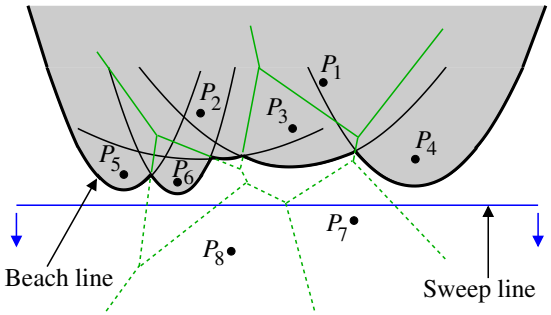
The convex hull of S identifies all the semi-infinite lines of $\mathrm{Vor}(S)$. Since no two sites have the same x-coordinate, $\mathrm{Vor}(S)$ does not contain any horizontal edge (finite or semi-infinite). In particular, the perpendicular bisectors of the edges of the upper hull extend to $y = +\infty$, and the perpendicular bisectors of the edges of the lower hull extend to $y = -\infty$. We say that one end of

such a semi-infinite line is at $+\infty$ or $-\infty$ as the case is. We do not know its finite end. On the other hand, we know neither of the ends of the finite edges of $\mathrm{Vor}(S)$. Fortune's algorithm furnishes this unknown information. If no end of an edge of $\mathrm{Vor}(S)$ is known, we say that the edge is not yet opened. If only one end of an edge is known, we say that the edge is opened. Finally, if both the ends of an edge are known, we say that edge is closed. We start Fortune's algorithm with only the semi-infinite edges opened.

In Fortune's algorithm, a line sweeps across the plane in a specific direction, like left-to-right or right-to-left or top-to-bottom or bottom-to-top. For better psychological impact, let the sweep line stay parallel to the x-axis and move from top to bottom. Imagine that you are sitting on the sweep line, and ocean waves are coming from the top in order to engulf you. Luckily, you will always remain safe on the sweep line as we will see shortly, unless you accidentally are in a site (for marketing perhaps) at the instant when the sweep line meets that site.

The sweep algorithms discussed earlier in this chapter process events one by one as they are encountered. The part of the plane already visited by the sweep object requires no future attention. In the case of Fortune's algorithm, on the other hand, a site below the sweep line may influence the geometry of the diagram above the sweep line. This means that a region already swept by the sweep line may require future attention. However, there is a part of the region swept already by the sweep line, that cannot be affected by future events that the sweep line encounters. This region is the ocean mentioned above. Below the ocean remains the land that contains the sweep line. The boundary between the ocean and the land is called the *beach line*. The geometry of the beach line is determined by the position of the sweep line and the locations of the sites. Figure 99 illustrates the concept. The part of the Voronoi diagram already determined by the algorithm lies in the ocean (the shaded region). The dotted portion of the Voronoi diagram is yet to be determined.

Figure 99: Fortune's sweep algorithm



Let me now provide a mathematical description of the beach line. The locus of all points equidistant from a given point and a given line is a parabola. The points *inside* the parabola are precisely those that are closer to the given point than the given line. Consider a site, say, P_5 in Figure 99. The set of points equidistant from P_5 and the sweep line is a parabola that opens vertically upward. From the site P_5, any point on or inside this parabola is strictly closer than any point below the sweep line. Thus, any unforeseen site below the sweep line cannot alter the portion of $\mathrm{VCell}(P_5)$ inside or on the parabola with P_5 at the focus. We take the union of the interiors of all the parabolas

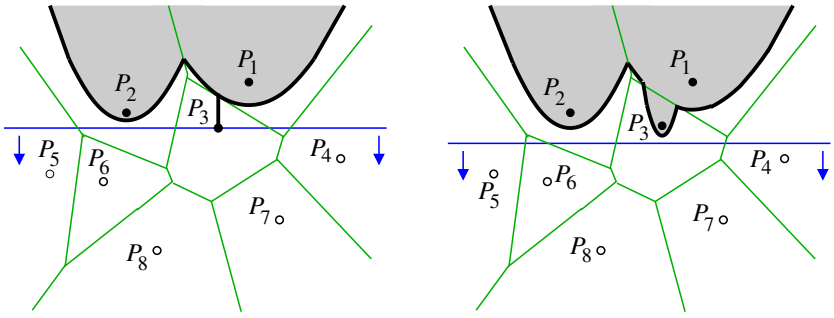
corresponding to the sites (on or) above the sweep line. This union constitutes the ocean. The part of Vor(S), that currently lies inside the ocean, cannot be affected by future encounters of the sweep line with sites (like P_7 or P_8).

The boundary of the ocean is the beach line. It consists of a sequence of parabolic arcs (which are parts of the boundaries of the parabolic waves focused at the sites above the sweep line). The beach line is x-monotone, which means that during a walk along the beach line keeping the ocean to the left, the x-coordinate increases monotonically. I will mention later how to store the beach line. In fact, the beach line need not be stored explicitly. Its only use is to make the understanding of the sweep algorithm simpler and more intuitive.

There is another very useful observation in this context. Consider two consecutive parabolas on the beach line, like those corresponding to P_5 and P_6 in Figure 99. The point at which these parabolas intersect is equidistant from P_5 and P_6 and so lies on the perpendicular bisector of the segment P_5P_6. As the sweep line moves down, the beach line advances downward appropriately, and the intersection point of two consecutive parabolas proceeds along the perpendicular bisector of the segment joining the corresponding sites. This is how edges of Vor(S) are generated by the sweep algorithm.

We, however, do not output the coordinates of Voronoi edges continuously. In fact, an edge of Vor(S) is specified by its two end points. So it suffices to identify only the end points of the segments. A situation where such an end point is generated is a type of events in this sweep algorithm. There are actually two types of events in Fortune's algorithm. I now characterize these events formally.

Figure 100: A site event in Fortune's algorithm



(a) A site event occurs (b) After the site event

Site event

A site event occurs when the sweep line encounters a new site. The situation is illustrated in Figure 100. Before this event occurs, the parabolas for the sites P_2 and P_1 were consecutive on the beach line, and the intersection of these parabolas was tracing the perpendicular bisector of P_1P_2. When the sweep line meets the site P_3, another parabola comes into the picture. The new parabola (for P_3) has its focus on the directrix at this moment, and so is degenerate, that is, a vertical half line starting at P_3. Introduction of this parabola in the beach line has several impacts. First, the parabolic arc for P_3 (now only a segment) breaks the arc for P_1 in two pieces and is itself inserted

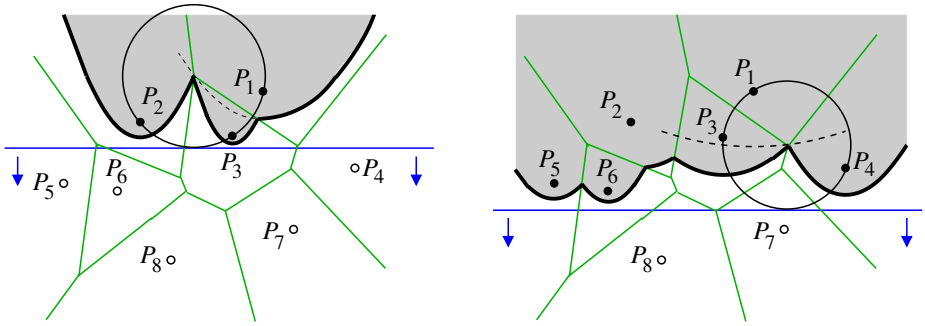
between the two pieces. Second, a new edge is detected in the Voronoi diagram. This is precisely the edge between VCell(P_3) and VCell(P_1). As the sweep line moves further down, the two points of intersection of the parabolas for P_3 and P_1 trace the perpendicular bisector of P_3P_1 (see Part (b) of the above figure). Notice that no vertex of Vor(S) is generated during a site event. We nevertheless have to handle site events in order to keep the beach line adequately updated.

Circle event

Circle events generate all the vertices of the Voronoi diagram. During a site event, a new parabolic arc is introduced in the beach line. Now, think of the converse situation, that is, a parabolic arc disappears from the beach line. This happens when an arc for a site is subsumed by the two neighboring arcs in the beach line.

Figure 101 illustrates this situation. Consider the situation shown in Part (b) of Figure 100. The introduction of the arc for P_3 creates a narrow arc for P_1 between the arcs for P_2 and P_3. As the sweep line moves further down, a moment comes when the length of this narrow arc reduces to zero (see Part (a) of Figure 101). After this happens, the intersection of the arcs for P_2 and P_3 moves along the perpendicular bisector of P_2P_3. The intersection of the arcs for P_3 and P_1 continues to move along the perpendicular bisector of P_3P_1.

Figure 101: Circle events in Fortune's algorithm



(a) A fork–out circle event

(b) A fork–in circle event

Another situation is depicted in Part (b) of Figure 101. Before this event occurs, the arcs for P_3, P_1, P_4 were consecutive on the beach line, and the perpendicular bisectors of P_3P_1 and P_1P_4 were traced. At the moment shown in the figure, the intermediate arc disappears. The arcs for P_3 and P_4 now become adjacent, and their intersection starts to trace the perpendicular bisector of P_3P_4.

Let us have a closer look at the geometry of a circle event. During the occurrence of this event, three parabolic arcs meet at a point; call it C. This point C is equidistant from the three sites to which the arcs belong. Such a point C must be a vertex in the Voronoi diagram (the sites are in general position, so only three sites are equidistant from a Voronoi vertex). The distance ρ from C to either of these points is the radius of the circle passing through the three sites. Moreover, ρ is also the distance of C (the center of the circle) from the sweep line. Thus, the circle touches the sweep line at this moment. This is the reason why this event is called a circle event. A future circle event can be predicted by three consecutive arcs on the beach line.

Not only does a circle event generate a vertex of the Voronoi diagram, it also identifies which edge(s) terminate and which edge(s) begin. Suppose that three consecutive arcs P_i, P_j, P_k participate in a circle event. We check whether the edges along the perpendicular bisectors of P_iP_j and P_jP_k are already opened at a finite point (above the common point of intersection of the three parabolas) or at $+\infty$. If so, it is a fork-in circle event. We have seen the other (common) end of the edges, so we close the edges corresponding to these perpendicular bisectors. We have also identified one end of the edge along the perpendicular bisector of P_iP_k. If that edge is opened at $-\infty$, we close it. Otherwise we open it. Its other finite end will be discovered later by a future circle event.

On the other hand, if the edges along the perpendicular bisectors of P_iP_j and P_jP_k are either not opened or opened at $-\infty$, we have a fork-out circle event. We close the segment along the perpendicular bisector of P_iP_k. For each of the perpendicular bisectors of P_iP_j and P_jP_k, we either open it if it is not yet opened, or close it if it is opened at $-\infty$.

Now, we plan to implement Fortune's algorithm. As usual, we maintain two data structures. The first one is the event queue Q which stores the y-coordinates of all future events. For a site event, the y-coordinate of the corresponding site is stored. For a circle event corresponding to three sites, the y-coordinate of the bottommost point of the circle passing through the sites is stored.

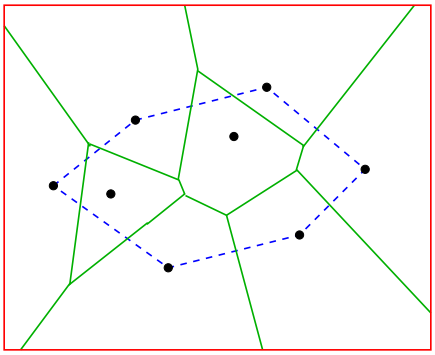
The second data structure B stores the information relevant to the sweep line. In this case, however, the important information lies on the beach line. (Of course, the current position of the sweep line is also important.) In B, we store all the parabolic arcs on the beach line from left to right. The arcs are not explicitly stored, but references to the sites to which the arcs belong are stored. Since a site may contribute multiple arcs on the beach line, multiple references to the same site may have to be stored in B. These different references are to be treated differently, since they correspond to different arcs of the same parabola.

The algorithm starts by initializing Q by all of the n site events. Subsequently, a loop is executed, which iteratively checks the next event (that is, the event with the largest y-coordinate in Q), deletes the event from Q, and handles the event appropriately. The loop stops when Q becomes empty. I now explain how different events are handled inside the loop body.

First, let us consider a site event. Let the sweep line meet the site P_i. We know that currently the parabola for P_i is a vertical line through P_i. We consult the data structure B in order to know which arc is intersected by this degenerate parabola. Let this arc correspond to P_s. From B, we also find out the immediate predecessor P_r and the immediate successor P_t of P_s on the beach line. We make a copy of (the reference to) P_s in B and insert P_i between these copies. That is, the relevant portion of B changes from P_r, P_s, P_t to P_r, P_s, P_i, P_s, P_t. After this insertion, the arcs for P_r, P_s, P_t are no longer consecutive on the beach line. So we delete from Q the circle event corresponding to the triple P_r, P_s, P_t. We also insert the circle events corresponding to the triples P_r, P_s, P_i and P_i, P_s, P_t. Notice that after the insertion we also have the consecutive arcs for P_s, P_i, P_s, but this triple cannot generate a circle event, since it involves only two sites P_i and P_s.

Handling a circle event is also straightforward. Let the circle event correspond to the triple P_i, P_j, P_k of sites. They contribute consecutive parabolic arcs on the beach line. Let P_s precede this sequence and P_t follow the sequence, that is, the relevant part of B is P_s, P_i, P_j, P_k, P_t. Now, P_j is removed from B, and the sequence becomes P_s, P_i, P_k, P_t. We delete from Q the circle events corresponding to the triples P_s, P_i, P_j and P_j, P_k, P_t. The circle event for the consecutive triple P_i, P_j, P_k need not be separately removed, since this is the circle event we are handling now and this will be

Figure 102: Enclosing the Voronoi diagram in a bounding box



deleted by the event handler anyway. Finally, we insert in Q the circle events corresponding to the triples P_s, P_i, P_k and P_i, P_k, P_t.

The nature of the event handling procedures highlights that we should implement both B and Q as height-balanced binary search trees. A heap is not good for implementing Q, since we have to delete circle events from the queue—such a circle event need not be the next event in the queue.

When the sweep line meets the topmost site, a single degenerate parabolic arc occupies the entire beach line. Subsequently, a site event replaces one arc by three, whereas a circle event deletes one arc from the beach line. The maximum number of arcs on the beach line (that is, in B) can be $2n-1$. The event queue Q may contain at most n site events and at most $2n-3$ circle events. So the sizes of both B and Q are $\mathrm{O}(n)$, and so the heights of the corresponding trees are $\mathrm{O}(\log n)$. Handling each event calls for $\mathrm{O}(1)$ basic operations on B and Q. Moreover, there are $\mathrm{O}(n)$ events to handle. Thus, Fortune's algorithm runs in $\mathrm{O}(n \log n)$ time, and its space requirement is $\mathrm{O}(n)$.

One final task remains about handling the semi-infinite edges of $\mathrm{Vor}(S)$. Currently, these have ends at $+\infty$ or $-\infty$. This is not a complete geometric description of these edges. To complete the description, we choose a bounding region (a square or a rectangle or a circle) large enough to enclose all the sites and all the finite vertices of $\mathrm{Vor}(S)$. For each edge P_iP_j of $\mathrm{CH}(S)$, we compute the perpendicular bisector of P_iP_j, and determine its intersection with the boundary of the bounding region. We end the semi-infinite segment at that point of intersection with the understanding that this actually crosses the boundary and proceed toward infinity. This final stage is illustrated in Figure 102, and can be completed in $\mathrm{O}(n)$ time.