Figure 94 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) {
    $\theta_{\min} = 180^0$;
    for each point $Q'$ of $S$ {
        $\theta =$ the angle between $\overline{PQ}$ and $\overline{QQ'}$;
        if ($\theta < \theta_{\min}$) {
            $\theta_{\min} = \theta$; $R = Q'$;
        }
    }
    if ($R$ is the same as $P_1$) break;
    Add $R$ to the end of the list $C$;
    Let $L$ be the directed line $\overline{QR}$;
}
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

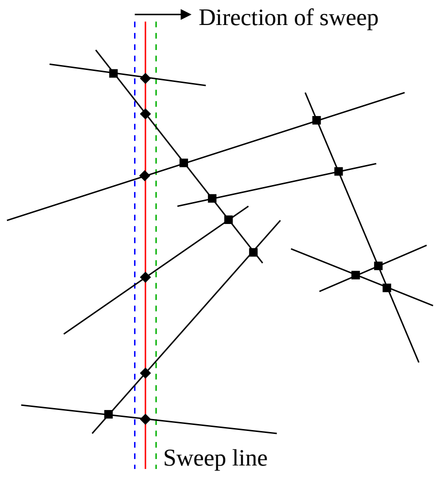
We are given a set of n line segments $L_1, \ldots, L_n$ specified by their end points. Our task is to determine all the points of intersection of these segments. We assume that the lines are in general position. In this context, this means the following.

- 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 95: 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 95). 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.

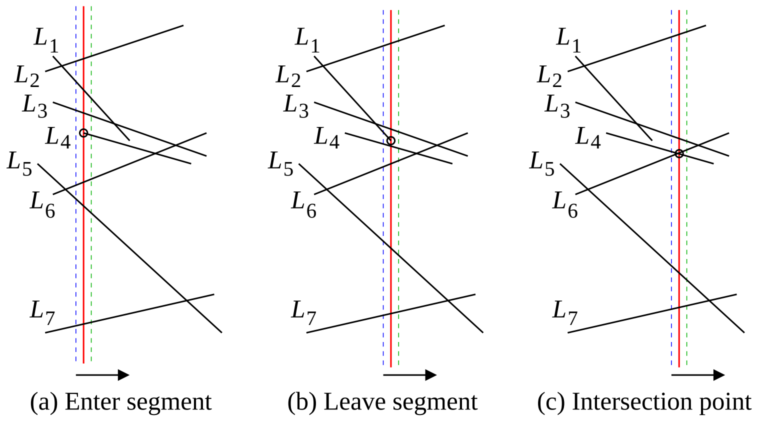
Figure 95 shows a *general* position of the sweep line. It also shows the positions of the sweep line a little before and a little after the current position (see the dotted lines). Notice that the positions of the intersecting segments relative to the sweep line do not change during this span of the 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 96. 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 96: Events in the line sweep algorithm for line segment intersection



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 96 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 96, 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 96, 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 $P_1,\ldots,P_n,Q_1,\ldots,Q_n$ 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 $\bigcap(L_s,L_t)$ exists and lies to the right of $L$, delete $\bigcap(L_s,L_t)$ from $E$;
      If $\bigcap(L_s,L_i)$ exists and lies to the right of $L$, insert $\bigcap(L_s,L_i)$ in $E$;
      If $\bigcap(L_i,L_t)$ exists and lies to the right of $L$, insert $\bigcap(L_i,L_t)$ 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 $\bigcap(L_s,L_t)$ exists and lies to the right of $L$, insert $\bigcap(L_s,L_t)$ in $E$;
        } else if it is an intersection point event {
            Suppose that $\bigcap(L_i,L_j)$ has triggered the event;
            Print $\bigcap(L_i,L_j)$;
            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 $\bigcap(L_s,L_i)$ exists and lies to the right of $L$, delete $\bigcap(L_s,L_i)$ from $E$;
            If $\bigcap(L_t,L_j)$ exists and lies to the right of $L$, delete $\bigcap(L_t,L_j)$ from $E$;
            If $\bigcap(L_s,L_j)$ exists and lies to the right of $L$, insert $\bigcap(L_s,L_j)$ in $E$;
            If $\bigcap(L_i,L_t)$ exists and lies to the right of $L$, insert $\bigcap(L_i,L_t)$ 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 $O(\log n)$.

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.

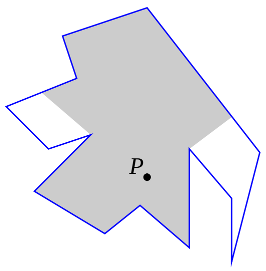
A problem, however, arises in this context. Let us think about an insertion operation in S necessitated by an *enter segment* event. A new segment now comes to the picture. We know the y-coordinate of its left end point. But we must also know the relative position of this y-coordinate with respect to 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 97 illustrates this situation.

Figure 97: 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