Computational Geometry

Jie Wang

University of Massachusetts Lowell
Department of Computer Science

Algorithmic Issues and Techniques: A General Discussion

- We are dealing with problems that are always solvable by brute force. **Objective**: Solve them as efficient as possible.
- Algorithmic issues can be roughly divided into three categories:

1. Technique-specific. For example, sorting-searching-traversing, divide-and-conquer, dynamic programming, greedy selection, linear programming. These techniques can be applied to different kinds of applications in different domains.
2. Domain-specific. For example, graph algorithms deal with graph problems; maximum flow algorithms deal with maximum-flow problems; FFT deals with multiplications of polynomials; computational geometry deals with geometric problems. Solving these problems may require using different algorithmic techniques, or inventing new ones.
3. Model-specific. For example, algorithms for sequential models (i.e., deterministic Turing machines, RAM); algorithms for distributed/parallel models (e.g., multi-threaded algorithms); algorithms for quantum computers.

Computational Geometry Overview

- Similar to graph algorithms, computational geometry (CG) algorithms use a number of techniques with special domain knowledge in geometry.
- Inputs to a CG problem are often descriptions of geometric objects, including sets of points, line segments, polygons, ...
- For example, input objects may be represented by a set of of points $\{\mathbf{p}_1, \mathbf{p}_2, \ldots\}$ in a d-dimensional space R^d. For instance, we can represent a triangle by specifying 3 points in R^2:

$$\{(0,0),(0,3),(3,0)\}.$$

- Outputs to a CG problem are inquiries about the inputs. For example:
  - Do lines intersect?
  - What is the smallest convex polygon that encloses a set of points?

Properties of Line Segments

- A line segment defined by two end points $\mathbf{p}_1$ and $\mathbf{p}_2$ is often denoted by $\overline{\mathbf{p}_1\mathbf{p}_2}$.
- **Convex combination**. Given two points $\mathbf{p}_1 = (x_1, y_1)$ and $\mathbf{p}_2 = (x_2, y_2)$, a convex combination of any point $\mathbf{p}_3 = (x_3, y_3)$ is defined below for some α in the range $0 \leq \alpha \leq 1$:

$$x_3 = \alpha x_1 + (1 - \alpha)x_2,$$
$$y_3 = \alpha y_1 + (1 - \alpha)y_2.$$

- A line in the 2D space is a convex combination of two points $\mathbf{p}_1$ and $\mathbf{p}_2$.
- A point is said to be on the line segment if it lies along the line defined by $\mathbf{p}_1$ and $\mathbf{p}_2$ and the point is between $\mathbf{p}_1$ and $\mathbf{p}_2$.
- A directed line segment is denoted by $\overrightarrow{\mathbf{p}_1\mathbf{p}_2}$.
  - If p_1 is the origin $(0, 0)$, the directed segment is simply denoted as a vector $\mathbf{p}_2$.

Typical Questions

1. Given two directed line segments $\overrightarrow{\mathbf{p}_0\mathbf{p}_1}$ and $\overrightarrow{\mathbf{p}_0\mathbf{p}_2}$, is $\overrightarrow{\mathbf{p}_0\mathbf{p}_1}$ clockwise from $\overrightarrow{\mathbf{p}_0\mathbf{p}_2}$ or counterclockwise with respect to their common end point $\mathbf{p}_0$?
2. Given two line segments $\overline{\mathbf{p}_0\mathbf{p}_1}$ and $\overline{\mathbf{p}_1\mathbf{p}_2}$, will traversing $\overline{\mathbf{p}_0\mathbf{p}_1}$ and then $\overline{\mathbf{p}_1\mathbf{p}_2}$ make a left or right turn at point $\mathbf{p}_1$?
3. Do line segments $\overline{\mathbf{p}_1\mathbf{p}_2}$ and $\overline{\mathbf{p}_3\mathbf{p}_4}$ intersect?

These questions can be answered in $O(1)$ time, and we want to do so without using division or trigonometric functions for the following reasons:

- Division is expensive and error prone, so are trigonometric functions.
- For example, to determine if two line segments intercept, the "straightforward" method is to solve the line equations first and then check if the intersection is on both line segments. This method needs to use division to find the point of intersection.
- When the line segments are nearly parallel, this method is sensitive to the precision of the division operation on real computers.

Cross Products

- The cross product of two points $\mathbf{p}_1 = (x_1, y_1)$ and $\mathbf{p}_2 = (x_2, y_2)$, denoted by $\mathbf{p}_1 \times \mathbf{p}_2$, is defined by

$$\begin{aligned}\mathbf{p}_1 \times \mathbf{p}_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -\mathbf{p}_2 \times \mathbf{p}_1.\end{aligned}$$

- If $\mathbf{p}_1 \times \mathbf{p}_2 > 0$, then $\mathbf{p}_1$ is clockwise from $\mathbf{p}_2$ w.r.t. the origin (0,0); if $\mathbf{p}_1 \times \mathbf{p}_2 < 0$, then $\mathbf{p}_1$ is counterclockwise from $\mathbf{p}_2$.
- If $\mathbf{p}_1 \times \mathbf{p}_2 = 0$ then, $\mathbf{p}_1$ and $\mathbf{p}_2$ are **colinear**, namely, they are along the same line, either in the same direction or opposite.

Geometric Interpretation of the Cross Product

The cross product is the (shaded) area of a parallelogram formed by points $(0, 0), \mathbf{p}_1, \mathbf{p}_2$, and $\mathbf{p}_1 + \mathbf{p}_2 = (x_1 + x_2, y_1 + y_2)$.

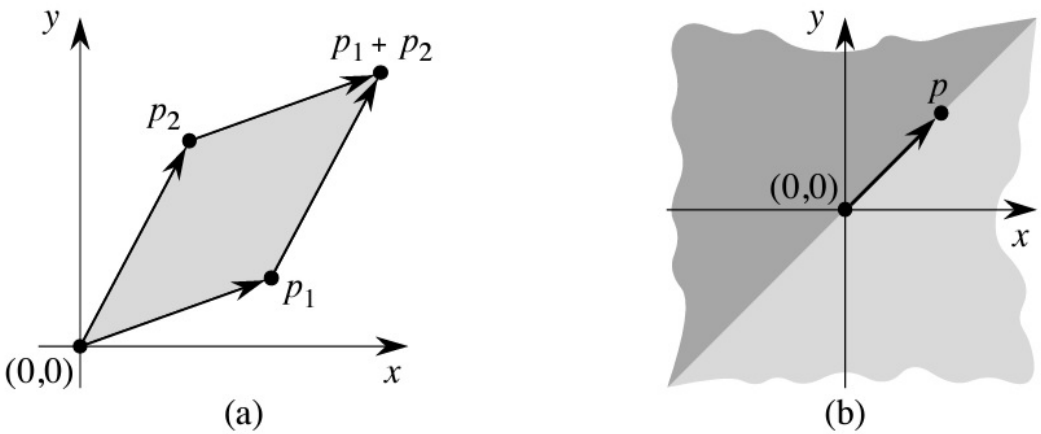


Figure 33.1 **(a)** The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. **(b)** The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.

Line Segments Starting at Arbitrary Points

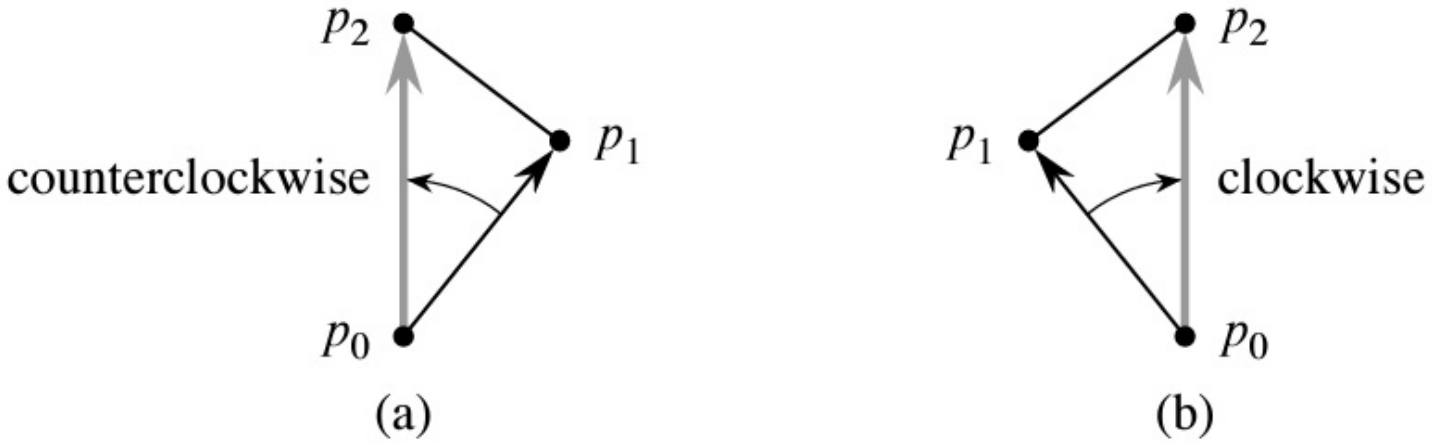
To determine the relationship between two directed line segments $\overrightarrow{\mathbf{p}_0\mathbf{p}_1}$ and $\overrightarrow{\mathbf{p}_0\mathbf{p}_2}$,

- Idea: Translate the directed line segments to start at the origin. This can be achieved by computing $\mathbf{p}_1' = \mathbf{p}_1 - \mathbf{p}_0$ and $\mathbf{p}_2' = \mathbf{p}_2 - \mathbf{p}_0$.
- Vector subtraction is subtraction componentwise.
- Compute $\mathbf{p}_1' \times \mathbf{p}_2'$ and check the sign of the result.

Turns

The cross product can be used to determine if one walks along consecutive line segments whether they turn left or right.

- Just need to figure out if $\mathbf{p}_0\overrightarrow{\mathbf{p}_2}$ is clockwise or counterclockwise from $\mathbf{p}_0\overrightarrow{\mathbf{p}_1}$.
  - Namely, compute $(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)$ and use the sign of the cross product to determine turns.
- To see how this works geometrically, consider the following example:



Intersection

- **Seek** a solution that does **not** use division at all.
- The line segment $\overline{\mathbf{p}_1\mathbf{p}_2}$ **straddles** a line $\overline{\mathbf{p}_3\mathbf{p}_4}$ if point $\mathbf{p}_1$ is on one side of the line and $\mathbf{p}_2$ is on the other side of the line.
- Two segments intersect iff either (or both) of the following conditions hold:
  1. Each segment straddles the line containing the other.
  2. An endpoint of one segment lies on the other segment.
- Given segments $A = \overline{\mathbf{p}_1\mathbf{p}_2}$ and $B = \overline{\mathbf{p}_3\mathbf{p}_4}$, compute

$$
\begin{aligned}
d_1 &= (\mathbf{p}_4 - \mathbf{p}_3) \times (\mathbf{p}_1 - \mathbf{p}_3), \\
d_2 &= (\mathbf{p}_4 - \mathbf{p}_3) \times (\mathbf{p}_2 - \mathbf{p}_3), \\
d_3 &= (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1), \\
d_4 &= (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_4 - \mathbf{p}_1).
\end{aligned}
$$

Use Cross Products

- The line segments are colinear.
  - This means $d_1 = d_2 = d_3 = d_4 = 0$.
  - Need to check if one of the points lies within the endpoints of the line segment. If one of them does, the segments intersect.
- Line segment A has one endpoint counterclockwise from line segment B and the other endpoint clockwise from segment B. This is true iff

$$d_1 \cdot d_2 < 0 \text{ and } d_3 \cdot d_4 < 0.$$

- Line segment A has one endpoint either clockwise or counterclockwise from line segment B and the other end point is somewhere on the line extending line segment B.

Pseudo Code

SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)

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

DIRECTION(p_i, p_j, p_k)

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

ON-SEGMENT(p_i, p_j, p_k)

```
1  if min(xi, xj) ≤ xk ≤ max(xi, xj) and min(yi, yj) ≤ yk ≤ max(yi, yj)
2      return TRUE
3  else return FALSE
```

Case Analysis

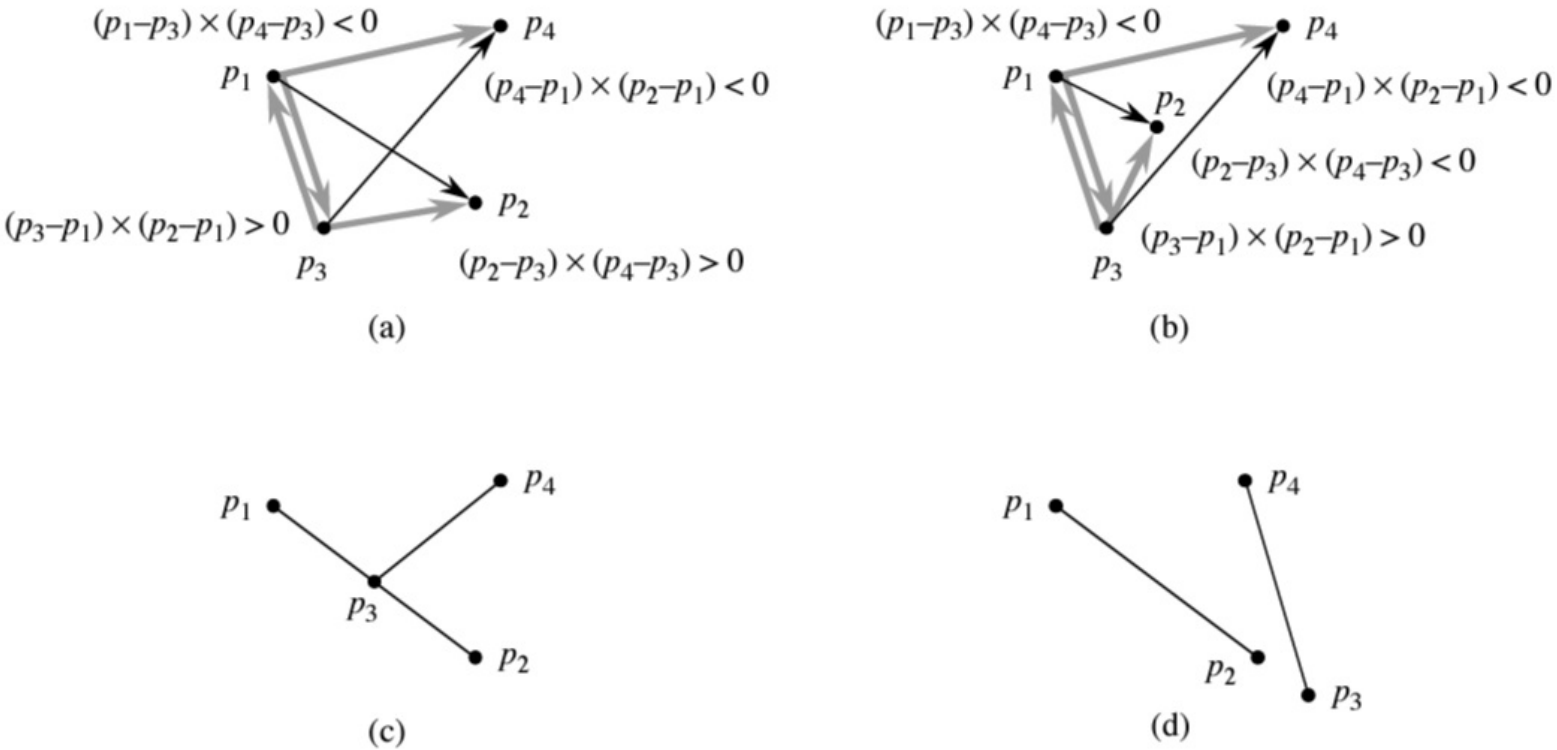


Figure 33.3 Cases in the procedure SEGMENTS-INTERSECT. **(a)** The segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ straddle each other's lines. Because $\overline{p_3p_4}$ straddles the line containing $\overline{p_1p_2}$, the signs of the cross products $(p_3 - p_1) \times (p_2 - p_1)$ and $(p_4 - p_1) \times (p_2 - p_1)$ differ. Because $\overline{p_1p_2}$ straddles the line containing $\overline{p_3p_4}$, the signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ differ. **(b)** Segment $\overline{p_3p_4}$ straddles the line containing $\overline{p_1p_2}$, but $\overline{p_1p_2}$ does not straddle the line containing $\overline{p_3p_4}$. The signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ are the same. **(c)** Point p_3 is colinear with $\overline{p_1p_2}$ and is between p_1 and p_2. **(d)** Point p_3 is colinear with $\overline{p_1p_2}$, but it is not between p_1 and p_2. The segments do not intersect.

A Collection of Line Segments

- Given a collection of n line segments, we would like to know if there are two line segments that intersect.
- A brute-force $O(n^2)$-time algorithm: Check every pair of line segments.
- Can improve this bound to $O(n \log n)$.
- **Line-sweep algorithms**.
  - In a line-sweep algorithm, an imaginary vertical (or horizontal) line moves across the plane (left-to-right or top-to-bottom).
  - When the sweep line intersects a line segment, a computation is performed and sweep the line continues sweeping.
  - Use an efficient dynamic data structure to store the resulting order of objects.

Assumptions and Terminologies

1. No vertical vertical segments (or horizontal segments).
2. No three segments intersect at the same point.

- If the sweep-line is at location x, then all segments with $x_i = x$ are called **comparable**.
- One segment s_1 is **above** another segment s_2 at x, denoted by $s_1 \succcurlyeq_x s_2$, if $y_1 > y_2$, where (x, y_1) is on s_1 and (x, y_2) is on s_2.
- The relation $\succcurlyeq$ is transitive and reflexive.
- A segment **enters** the ordering when its left endpoint is hit by the sweep line. The segment **leaves** the ordering when its right end point is passed.
- **An important observation**: Line-sweeping specifies an order for all line segments. But when two segments intersect, they reverse their order.

Line-Sweeping Detail

If two segments s_1 and s_2 intersect, we have the following:

- There is a point u where the sweep line has passed through the intersection. Namely, $s_1 \succcurlyeq_u s_2$ or $s_2 \succcurlyeq_u s_1$.
- W.l.o.g, before the intersection, assume that $s_1 \succcurlyeq_v s_2$ at v, and after the interaction point, lets say point w, we have $s_2 \succcurlyeq_w s_2$.
  - Notice that the ordering of these segments have changed.
  - Since we assume that no three segments intersect at the same point, there must be some vertical sweep line x for which intersecting segments are consecutive in the order $\succcurlyeq_x$.

Line-Sweeping Continued

To accomplish their work the line-sweeping process must manage two sets of data

1. The **sweep-line status** gives the relationship among the objects the sweep line intersects.
2. The **event-point schedule**
   - A sequence of points called **event points**.
   - These are ordered left to right according to their x coordinates.
   - When the sweep line intersects an event point's x coordinate, the sweep halts, processes the event points, and resume sweeping.
   - Changes to the sweep line status **only** occur at event points.

Event-Point Schedule

- Each segment endpoint is an event point.
- Sort them in increasing order by x coordinate from left to right.
- If two event points share the same x coordinate they are called **covertical**.
- Covertical points are handled as follows:
  1. Place left endpoints before right endpoints.
  2. Order the resulting left endpoints by y coordinate.
  3. Order the resulting right endpoints by y coordinate.
- Use a balanced binary-search T (e.g., AVL tree, red-black tree) that can perform the following operations in $O(\log n)$ time:
  - $\textsc{Insert}(T, s)$: Insert segment s into T.
  - $\textsc{Delete}(T, s)$: Delete segment s from T.
  - $\textsc{Above}(T, s)$: Return the segment immediately above segment s in T.
  - $\textsc{Below}(T, s)$: Return the segment immediately below segment s in T.

Algorithm Detail

- When a left event point is encountered, call $\textsc{Insert}(T, s)$, where s is the corresponding line segment.
- When two segments are consecutive in the ordering, checked using $\textsc{Above}(T, s)$ and $\textsc{Below}(T, s)$, check if they intersect.
- When a right event point is encountered, call $\textsc{Delete}(T, s)$, where s is the corresponding segment.

Pseudo Code

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

An Illustration

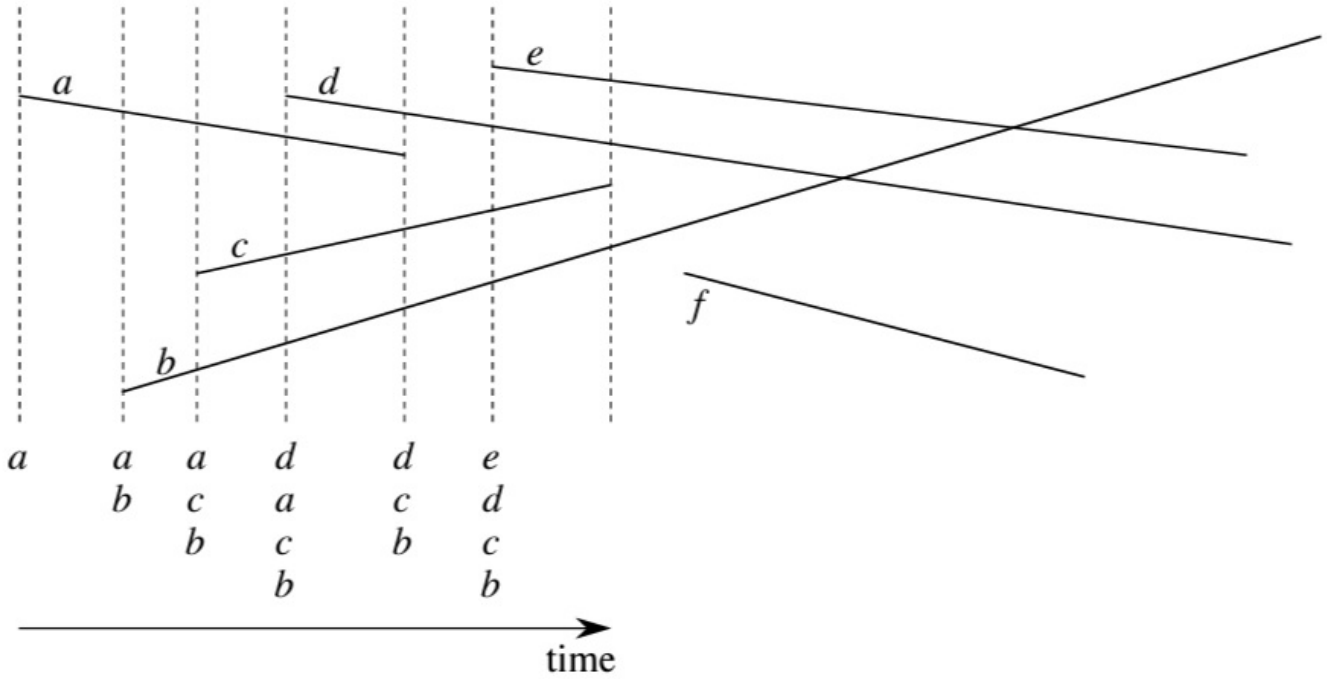


Figure 33.5 The execution of ANY-SEGMENTS-INTERSECT. Each dashed line is the sweep line at an event point. Except for the rightmost sweep line, the ordering of segment names below each sweep line corresponds to the total preorder T at the end of the **for** loop processing the corresponding event point.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

Runtime Analysis

- The best known algorithm to build a balanced binary search tree for n numbers incurs $O(n \log n)$ time.
- The loop runs once for each event point.
- There are $2n$ event points since each line segment is defined by exactly two endpoints.
- The body of the loop performs $O(\log n)$ work per event point.
  - There are a constant number of order data structure operations each costing $O(\log n)$ time.
  - Intersection calculations can be performed in $O(1)$ time.
- The runtime is $O(n \log n)$, where n is the number of segments.

The Convex Hull Problem

- The convex hull problem is the most important problem in computational geometry; a number of CG algorithms begin by computing the convex hull.
- Applications include
  - Collision detection in video games.
  - Path planing.
  - Processing satellite maps for accessibility.
  - Detecting statistical outliers.
  - Computing the diameter of a set of points (i.e. the distance between the two farthest points).
- A convex hull is a boundary around a collection of points and is defined below:
  Given a set S of points, the convex hull for S is the smallest convex polygon P for which each point of S is either on the boundary of P or in its interior.

An Example

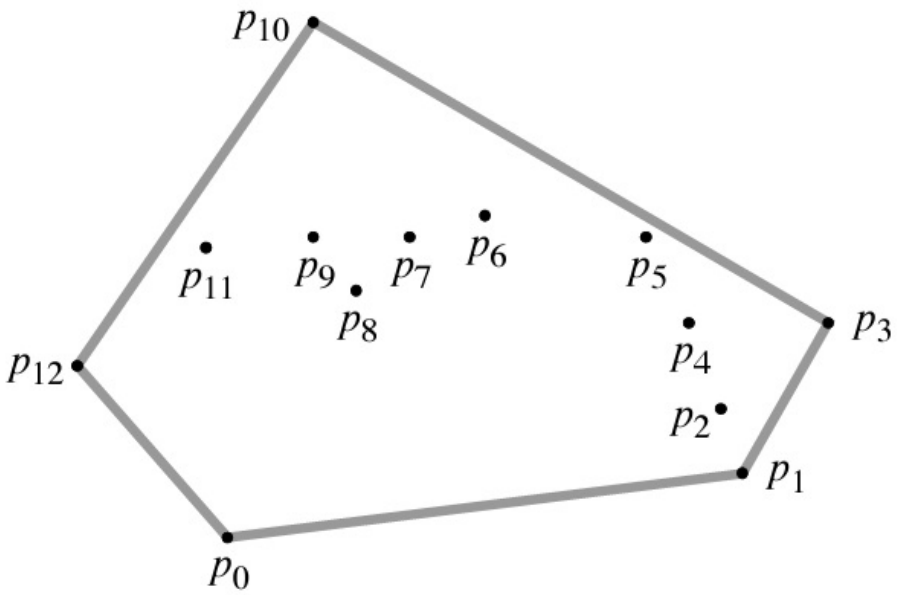


Figure 33.6 A set of points $Q = \{p_0, p_1, \ldots, p_{12}\}$ with its convex hull $\mathrm{CH}(Q)$ in gray.

Convex Polygons

- A convex polygon, defined below, is a polygon with no "dents" in it. A polygon is **convex** if given any two points on its boundary or in its interior, all points on the line segment drawn between them are contained in the polygon's interior or boundary.
- A set of points (finite or infinite) in the plane is called **convex** if for any two points **p** and **q** in the set, the entire line segment $\overline{\mathbf{pq}}$ belongs to the set.
- The convex hull of a set S of points is the smallest convex set containing S.
  - The smallest requirement means that the convex hull of S must be a subset of any convex set containing S.
- **Theorem**. A convex hull of any set S of $n > 2$ points not all on the same line is a convex polygon with the vertices as some of the points of S.

The Convex Hull Problem

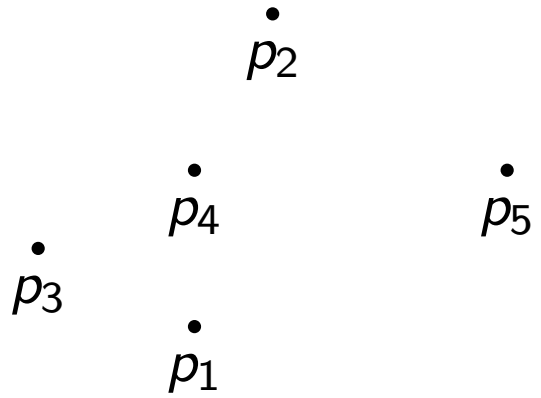
Input: A set S of n points.
Output: The list of extreme points that form the vertices of the convex polygon

- We call a point an **extreme point** of a set S if it is a point that is not a middle point of any line segment with endpoints in S.
  - Example: extreme points for a triangle are its vertices.
  - Example: extreme points for a circle are the points on its circumference.

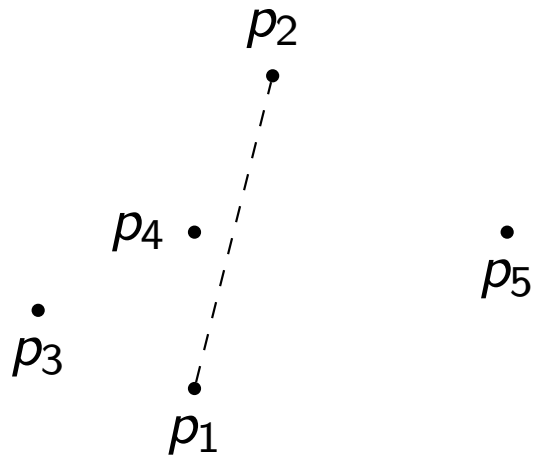
  Observation: A line segment connecting two points $\mathbf{p}_i$ and $\mathbf{p}_j$ of a set of n points is part of the convex hull's boundary iff all other points of the set lie on the same side of the straight line through these two points.

Brute Force

- Simply repeat the test in our observation for every pair of points in the set S.
- Let's work through a few steps on the following set of points:

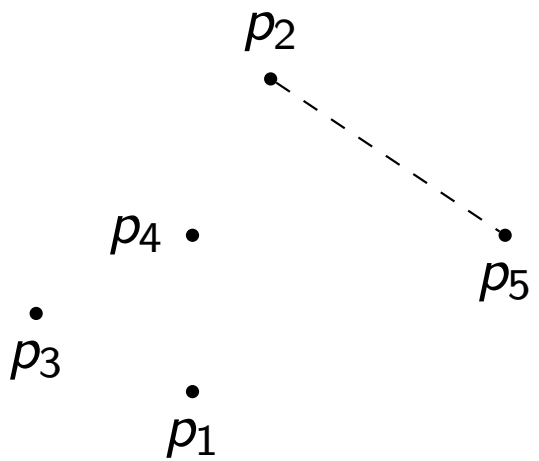


- **Bad Segment**: Test the line segment between p_1 and p_2



Brute Force Continued

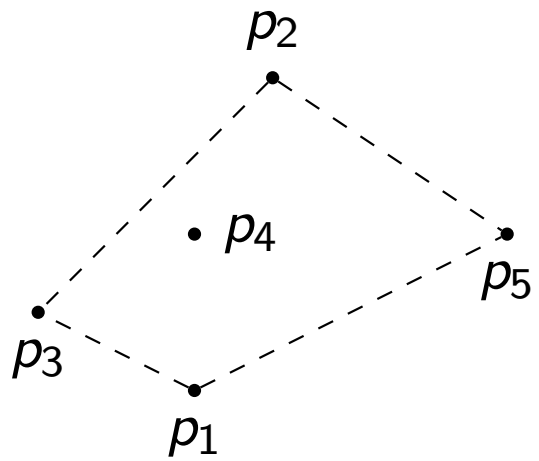
- The line segment in the previous slide produces points on both sides of the line segment, and so fail the test.
- **Good Segment**: Test the line segment between p_2 and p_5



- Every point is all on the same side of the line segment (such a geometric construct is called a **half-plane**). This segment is part of the convex hull.

Brute Force Continued

- The convex hull for the entire set:



- This is an inefficient way of solving the problem. We need to check $\binom{n}{2} = \frac{n(n-1)}{2}$ line segments. Each line segment requires to check $n-2$ points, yielding the following runtime:

$$(n-2)\frac{n(n-1)}{2} = \frac{(n-2)(n^2-n)}{2} = \frac{n^3-3n^2+2n}{2} = O(n^3).$$

- The best-known algorithms can solve convex hull in $O(n \log n)$ time.

Rotational Sweep

- Use the polar coordinate system, where a point is defined by two parameters:
  1. A distance r from the origin;
  2. An angle θ from a reference pole.
- A coordinate is defined by (r, θ), where r is the radial distance and θ is the radial angle from the reference pole
  - The common reference angle pole is the positive x axis.
  - The origin is simple the origin of the Cartesian plane.
- Given a polar coordinate (r, θ) one can compute the corresponding Cartesian coordinate (x, y) by observing

$$x = r \cos \theta,$$
$$y = r \sin \theta.$$

Grahm Scan

- Use a stack to perform the following operations:
  - PUSH – add an element to the stack
  - PEEK – return the top element of the stack **without** removing it.
  - POP – remove the top element from the stack.
  - NEXTTOTOP – returns the second from top element from the stack (without removing it).
- The Grahm scan stores the points of the convex hull on the stack.
- The Grahm scan first finds the point in set Q with the minimum y-coordinate, call it point $\mathbf{p}_0$.
  Should there be multiple such points, find the left most point.
- Use point $\mathbf{p}_0$ as the origin, sort all remaining points in increasing order counterclockwise according to their polar angles around it with the x-axis. Should there be multiple points with the same angle, remove all but the furthest away points for each angle.

Sorting

- Use the cross product to determine direction.
- To compare two points $\mathbf{p}_i$ and $\mathbf{p}_j$ with respect to $\mathbf{p}_0$, compute

$$c = (\mathbf{p}_i - \mathbf{p}_0) \times (\mathbf{p}_j - \mathbf{p}_0).$$

  - If $c > 0$, then $\mathbf{p}_i$ has a smaller angle than $\mathbf{p}_j$.
  - If $c < 0$, then $\mathbf{p}_i$ has a larger angle than $\mathbf{p}_j$.
  - No need to worry about the case when $c = 0$, for colinear points have been removed already.

Grahm Scan Detail

- W.l.o.g, write sorted points in the remaining points in Q as

$$\mathbf{p}_0 < \mathbf{p}_1 < \cdots < \mathbf{p}_m.$$

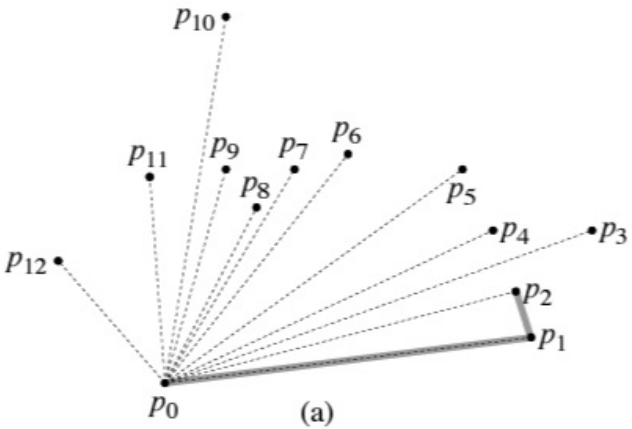
  Note: $m \leq n - 1$.

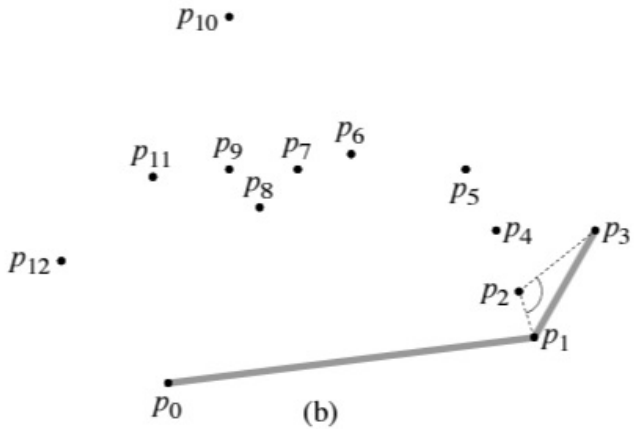
- The scan pushes $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$ onto the stack in that order.
- Loop from point $\mathbf{p}_3$ to $\mathbf{p}_m$. During each iteration,
  - Use a while loop to continually pop the top point off of the stack if $\mathbf{p} = \textsc{Peek}(S)$ and $\mathbf{p}' = \textsc{NextToTop}(S)$ makes a right turn from $\mathbf{p}_i$. That is, pop the stack if

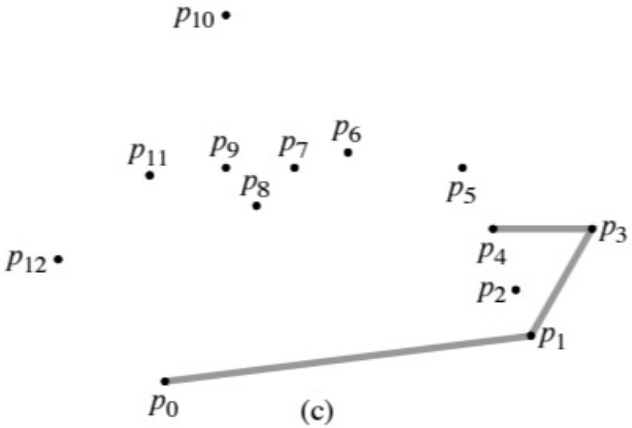
$$(\mathbf{p}' - \mathbf{p}_i) \times (\mathbf{p} - \mathbf{p}_i) > 0.$$

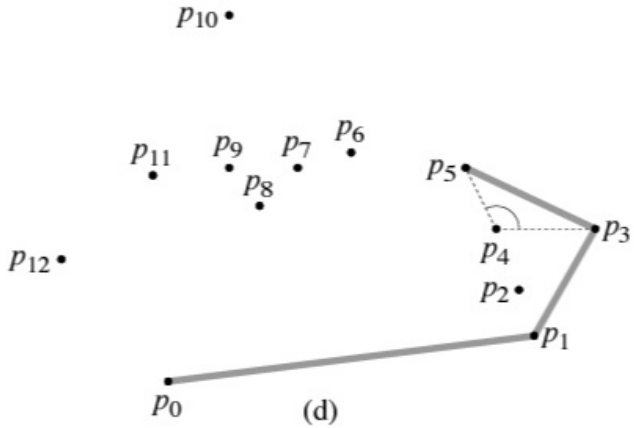
  - After the loop concludes, push $\mathbf{p}_i$ onto the stack.
- The remaining elements on the stack are the vertices of the convex hull.

Example

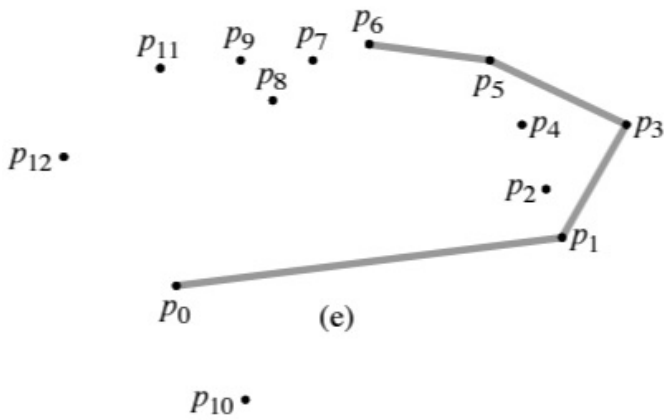

p_{10}
p_{11}
p_9
p_7
p_6
p_8
p_5
p_4
p_3
p_{12}
p_2
p_1
p_0
(a)

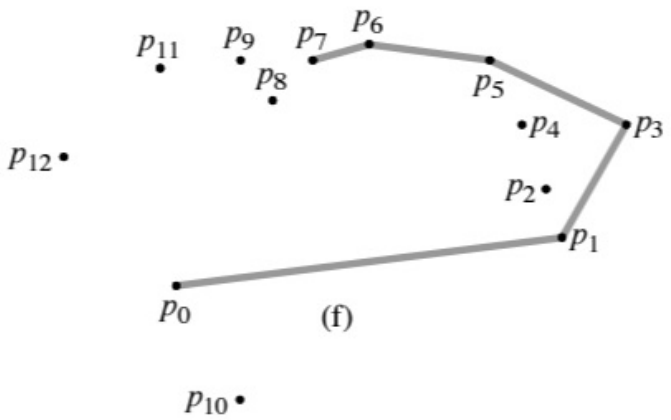

p_{10}
p_{11}
p_9
p_7
p_6
p_8
p_5
p_4
p_3
p_{12}
p_2
p_1
p_0
(b)

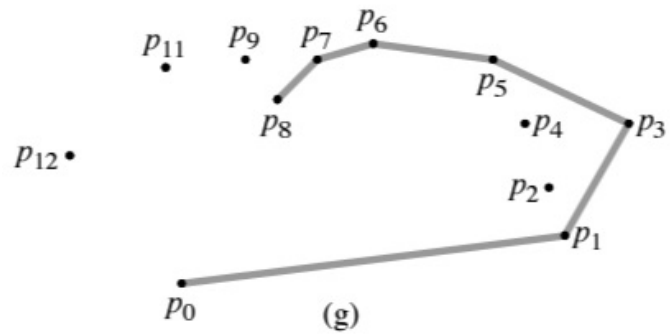

p_{10}
p_{11}
p_9
p_7
p_6
p_8
p_5
p_3
p_4
p_{12}
p_2
p_1
p_0
(c)

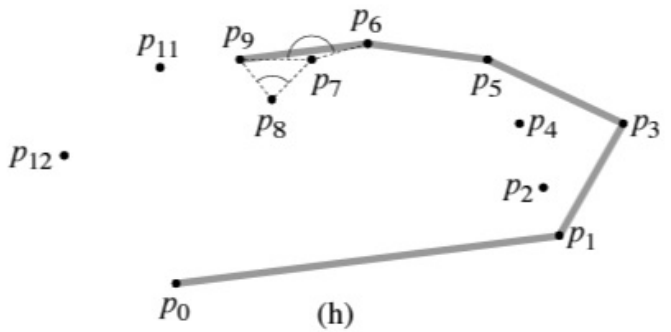

p_{10}
p_{11}
p_9
p_7
p_6
p_8
p_5
p_3
p_4
p_{12}
p_2
p_1
p_0
(d)

Example Continued

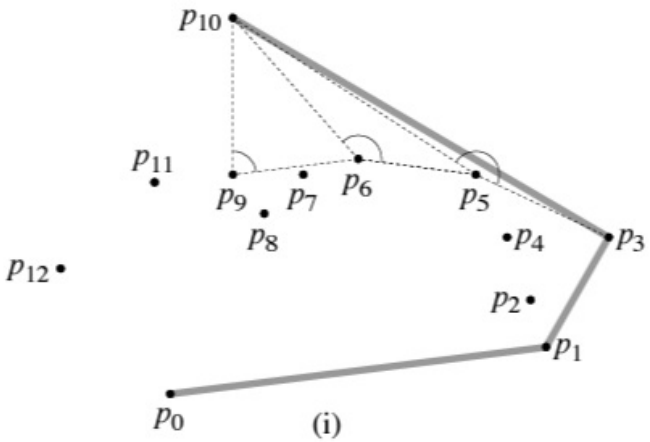


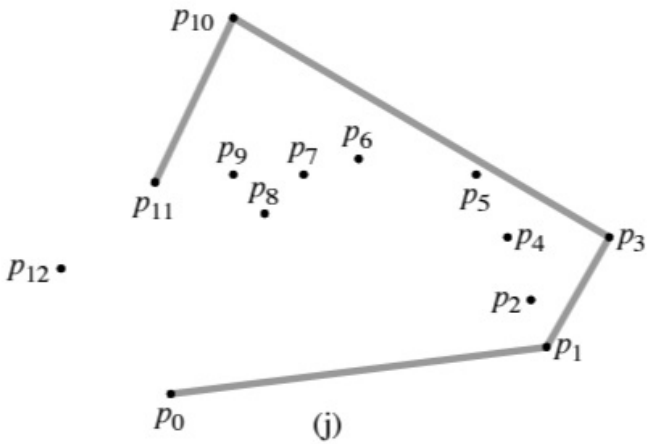


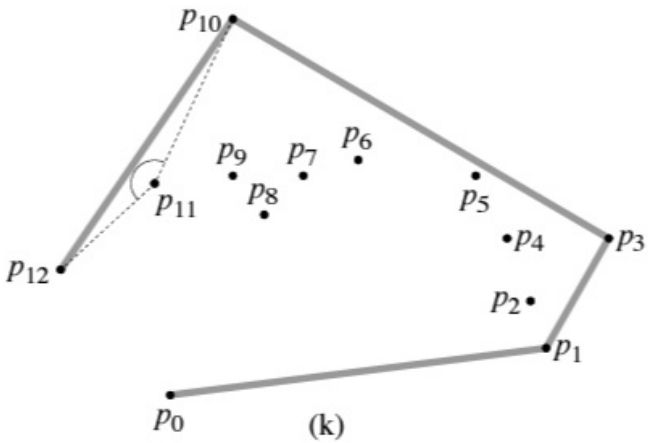


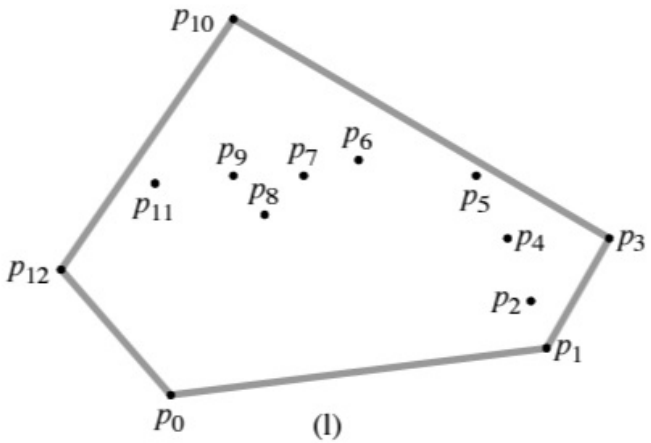


Example Continued









Correctness

- Each time through the inner loop, the Grahm scan finds the next edge to add to the polygon.
- If a right turn is made based on the current set of vertices, it won't be convex. Must backtrack by popping the last edge off and rechecking.
- When the scan finds an edge that does not involve a right turn from the last edge, we know that all points we have popped are to the left of this edge and thus in the interior of the polygon.
- This is the "tightest" polygon, as we are processing the edges based on angular path from the lowest vertex in the plane.

Runtime Analysis

- Finding $\mathbf{p}_0$ and removing all colinear points can be done in $O(n)$ time.
- Sorting takes $O(n \log n)$ time.
- The scan only inspects at most n points, so the outer loop runs n times.
- Using amortized analysis, the inner most loop can only pop what has been pushed (points $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_l$ are not popped, where $\mathbf{p}_l$ is the point with the largest index remaining in the stack.
  - This means at most $l - 2$ $\textsc{Pop}$ operations must have occurred.
  - We know that $l \leq m \leq n - 1$.
  - This means across all iterations of the while loop it has run at most $O(n)$ time.
- Thus, the Grahm scan incurs $O(n \log n)$ time.

Jarvis' March

- Jarvis' March uses a technique known as **gift wrapping** to determine the convex hull.
- It runs in time $O(nh)$, where h is the number of vertices in the convex hull.
  - Observe if $h = o(\log n)$, then Jarvis' March is asymptotically faster than the Grahm scan.
- The intuition of Jarvis' March is as follows:
  1. Start from the lowest point in the set, which is the same $\mathbf{p}_0$ as in the Grahm scan. If there is a tight, select the left-most point. This is the first point in the convex hull.
  2. Starting from the previous point of the convex hull, find the point with the right-most angle, this is the next point for the convex hull.
  3. Repeat until $\mathbf{p}_0$ is reached.

Example

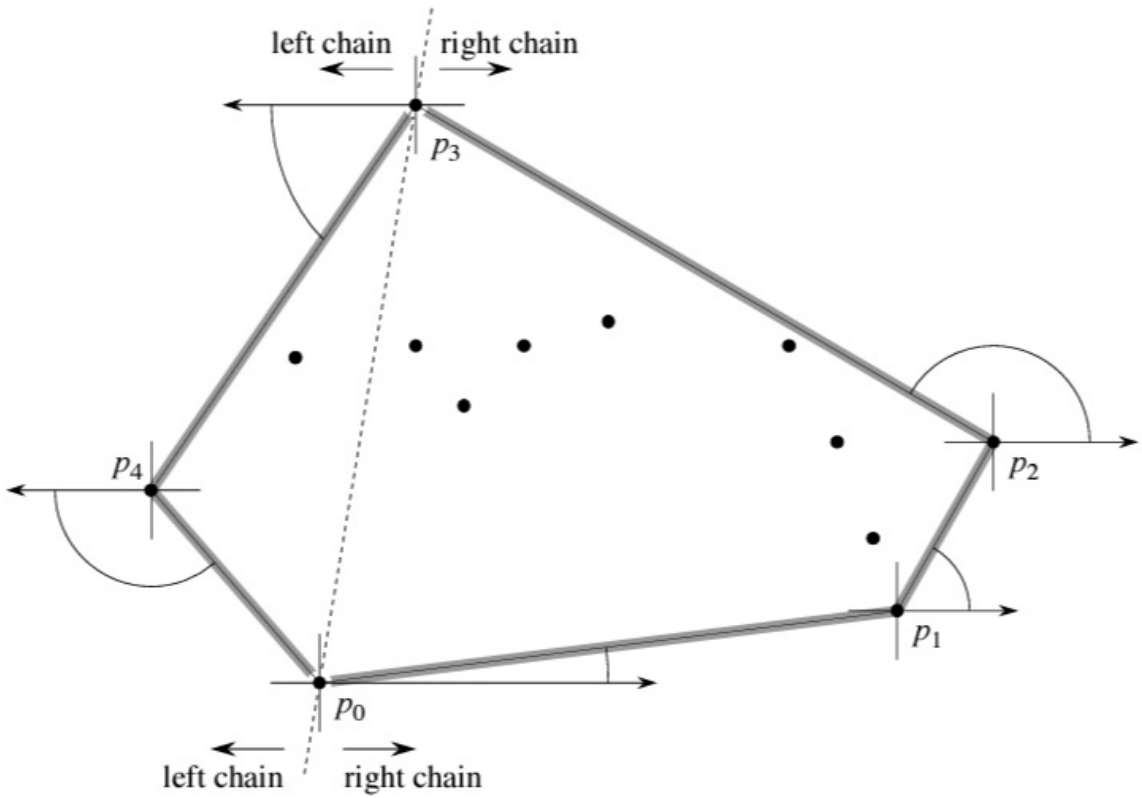


Figure 33.9 The operation of Jarvis's march. We choose the first vertex as the lowest point p_0. The next vertex, p_1, has the smallest polar angle of any point with respect to p_0. Then, p_2 has the smallest polar angle with respect to p_1. The right chain goes as high as the highest point p_3. Then,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

The Closest Pair of Points

Input: A set $Q = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n\}$ of n points in the d-dimensional space and a metric $\delta : Q \times Q \to R$.
Output: The distance between the two closest points $\mathbf{p}_i$ and $\mathbf{p}_j$ according to the metric δ.

- For our purposes we will only consider the 2D Euclidean plane and with the metric

$$\delta(\mathbf{p}_i, \mathbf{p}_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

- Brute-force approach: Compute the distance between each pair of distinct points and find the pair with the smallest distance.
- Need to check $\binom{n}{2}$ pairings, resulting in a $O(n^2)$-time algorithm.
- Can solve in in $O(n \log n)$-time using a nice divide-and-conquer algorithm.

Divide and Conquer

- Let X and Y be arrays containing the points in Q sorted in increasing order by x coordinate and y coordinate respectively.
- Let $P \subseteq Q$. If $|P| \leq 3$, use the brute-force approach to solve this tiny instance. If $|P| > 3$,

  Divide: Find a vertical line l that bisects the point set P into two sets P_L and P_R such that the sets are almost equal in size.
  - Divide array X into arrays X_L and X_R according to P_L and P_R, sorted in increasing order by x coordinate.
  - Do the same with array Y to obtain Y_L and Y_R.

  Conquer:
  - Find the closet pair of points in P_L, and denote its distance by δ_L.
  - Find the closest pair of points in P_R, and denote its distance by δ_R.

Divide-and-Conquer Continued

- The closes pair of points in P can only be one of the following:
  - the closest pair of points in P_L,
  - the closest pair of points in P_R, or
  - the closest pair of points with one point in P_L and one point in P_R.

  Combine: The distance between this pair of points is at most

$$\delta = \min\{\delta_L, \delta_R\}.$$

  This means both points must be within δ of l. We look at the points in the 2δ-wide strip centered at l using the following procedure:

  1. Create an array Y', which is equal to array Y with all points not in the 2δ strip removed.
  2. For each point $\mathbf{p}$ in array Y', find the points in Y' that are within δ units of $\mathbf{p}$. Denote the distance of the closest pair by δ'.
  3. If $\delta' < \delta$, the closest pair is in the 2δ strip, return δ' and the associated points.

Correctness

It suffices to only consider the case of $|P| > 3$.

- At some level of recursion, let $p_L \in P_L$ and $p_r \in P_R$ be the closest pairs of of points with distance $\delta' < \delta$.
  - This means each point is less then δ from l.
  - p_L and p_R must be within δ of each other vertically.
- Thus, these points are in a $\delta \times 2\delta$ rectangle, which can be divided in two side-by-side squares with $\delta \times \delta$ dimension.
- In each $\delta \times \delta$ square there can only be at most 4 points.
  1. They can only be at the corners of the squares (otherwise they are too far from l or too far from the other sides y coordinate).
  2. Moreover, the points in each square must be at least δ units apart (otherwise they would be closer then δ').
- W.l.o.g., assume that p_L precedes p_R in Y'. Then, even if p_L occurs as early as possible in Y' and p_R occurs as late as possible, p_R is within 7 positions following p_L.
- The pair p_L and p_R become the closes pair with distance δ'.

Example

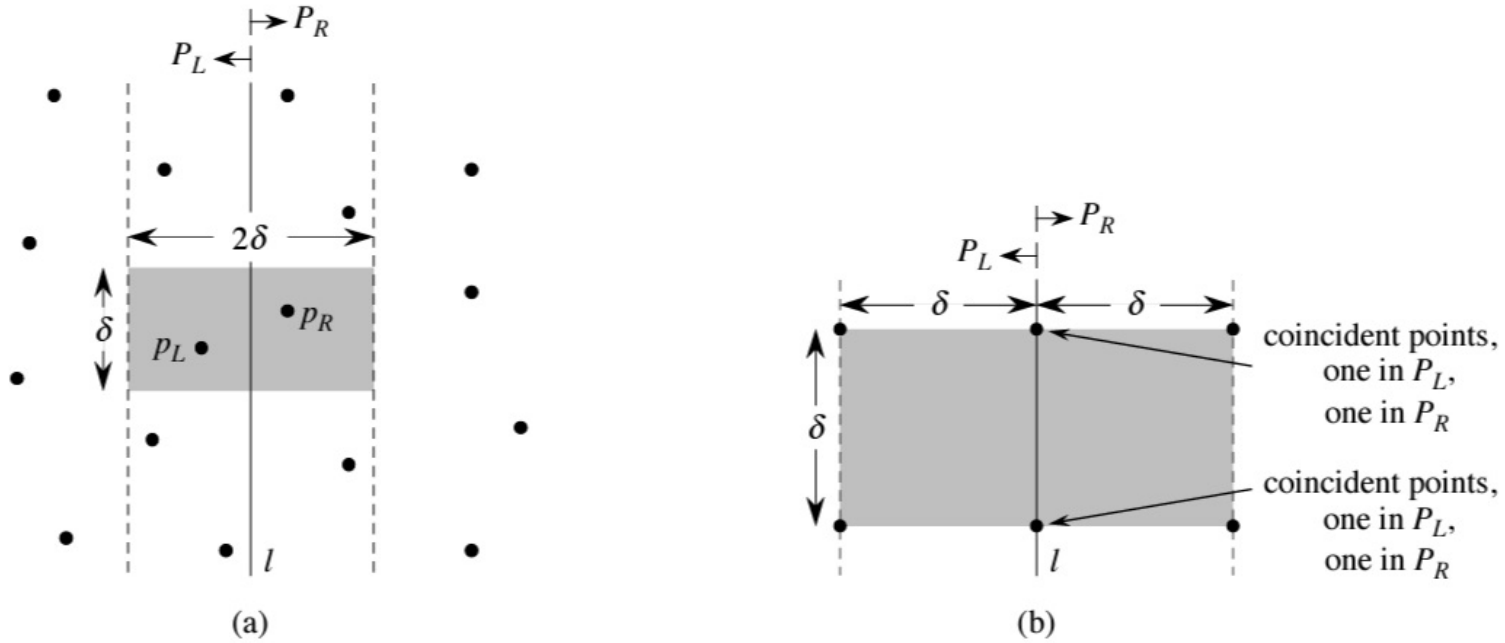


Figure 33.11 Key concepts in the proof that the closest-pair algorithm needs to check only 7 points following each point in the array Y'. **(a)** If $p_L \in P_L$ and $p_R \in P_R$ are less than δ units apart, they must reside within a $\delta \times 2\delta$ rectangle centered at line l. **(b)** How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

Runtime Analysis

- The runtime is given by the following recurrence

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n) & \text{If } n > 3, \\ O(1) & \text{If } n \leq 3. \end{cases}$$

  - The first term comes from the recursive call.
  - The work at each level is just to maintain sorted order for the partitions of X and Y, which incurs $O(n)$ time since X and Y were presorted. This only adds a one time cost of $O(n \log n)$ to the runtime.

- $$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^{\log 2} \log n) = \Theta(n \log n).$$