Algorithms
Chapter 33
Computational Geometry

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Line-segment properties**
- Determining whether any pair of segments intersects
- Finding the convex hull
- Finding the closest pair of points

Overview

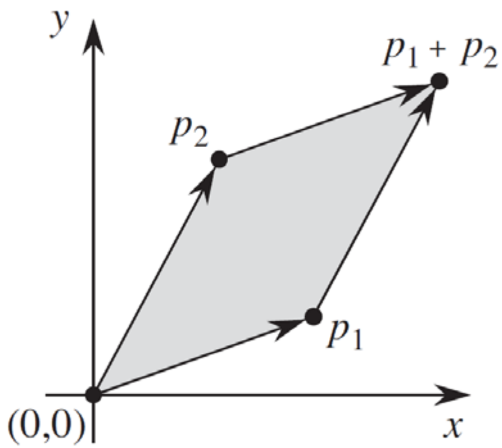
- **Computational geometry**: study algorithms for solving geometric problems such as
  - computer graphics,
  - robotics,
  - VLSI design, and
  - computer aided design.

- In this chapter, each input object is represented as a set of points $\{p_1, p_2, p_3, \ldots\}$, where each $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbf{R}$.
  - For example, an n-vertex polygon $P = \langle p_0, p_1, p_2, \ldots, p_{n-1} \rangle$.

Line-segment properties

- A **convex combination** of two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is any point $p_3 = (x_3, y_3)$ such that for some α in the range $0 \le \alpha \le 1$, we have
  - $x_3 = \alpha x_1 + (1 - \alpha)x_2$, and
  - $y_3 = \alpha y_1 + (1 - \alpha)y_2$.
- We also write that $p_3 = \alpha p_1 + (1 - \alpha)p_2$.
- The **line segment** $\overline{p_1 p_2}$ is the set of convex combinations of p_1 and p_2.
- We call p_1 and p_2 the **endpoints o**f segment $\overline{p_1 p_2}$.
- If p_1 is the **origin** (0, 0), then we can treat the **directed segment** $\overrightarrow{p_1 p_2}$ as the **vector** p_2.

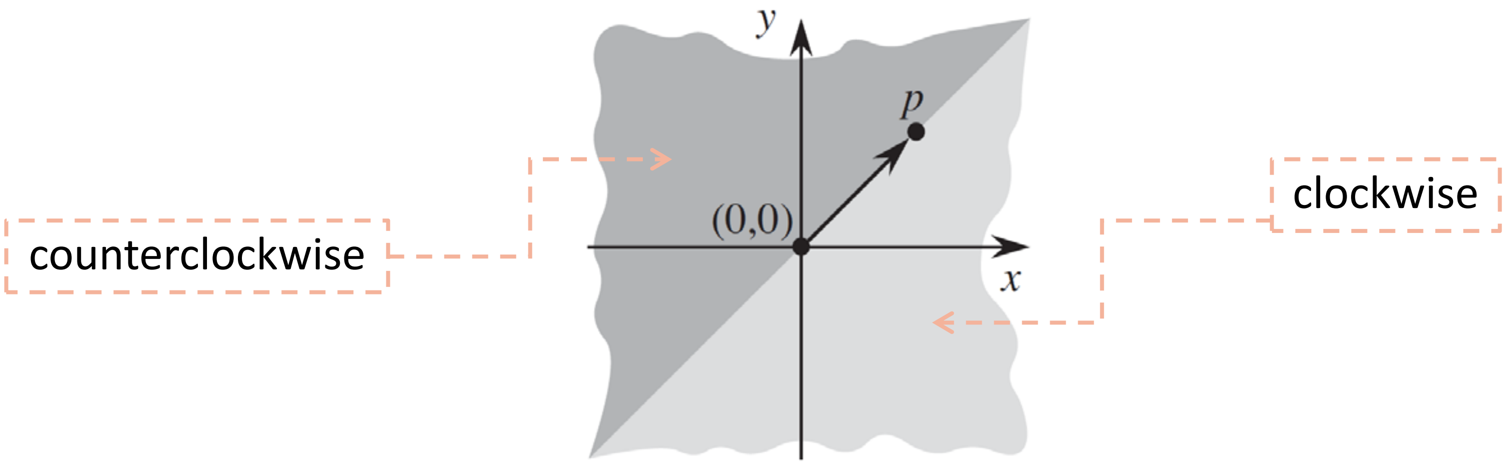
Cross products



- Consider vectors p_1 and p_2. The **cross product** $p_1 \times p_2$ of p_1 and p_2 is the signed area of the parallelogram formed by the points (0, 0), p_1, p_2, and $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$.
- An equivalent definition:

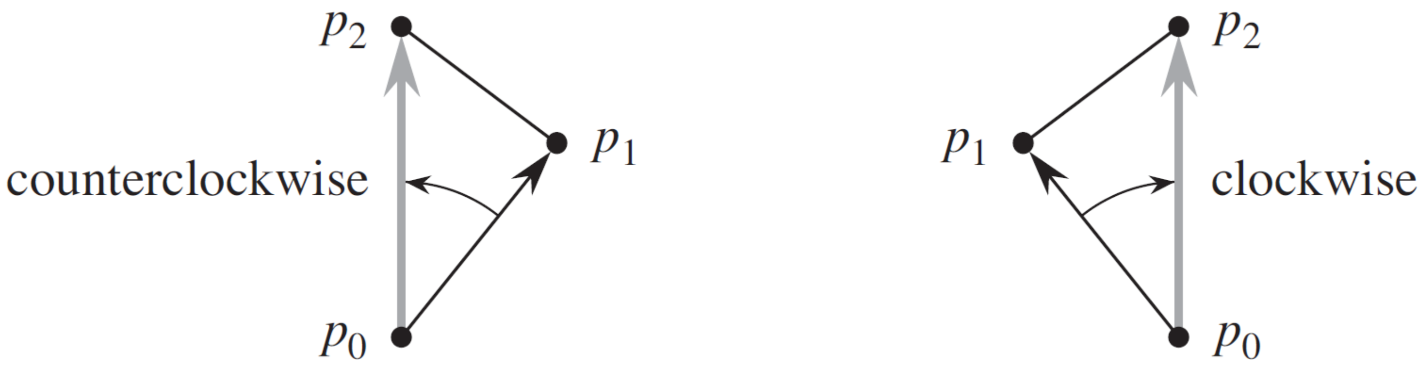
$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 . \end{aligned}$$

Clockwise, counterclockwise, or collinear ?



- **Question 1**: Given two vectors p_1 and p_2, is p_1 clockwise from p_2 with respect to their common endpoint p_0? If $p_1 \times p_2$ is
  - **positive**, then p_1 is clockwise from p_2.
  - **negative**, then p_1 is counterclockwise from p_2.
  - **0**, then the vectors are **collinear**, pointing in either the same or opposite directions.

Turn left or right ?



- **Question 2**: Given two line segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$, if we traverse $\overline{p_0p_1}$ and then $\overline{p_1p_2}$, do we make a left turn at point *p*1?
  - Check whether $\overrightarrow{p_0p_2}$ is clockwise or counterclockwise relative to $\overrightarrow{p_0p_1}$.
  - If **counterclockwise**, the points make a left turn.
  - If **clockwise**, they make a right turn.

Whether two line segments intersect ?

- **Question 3**: Do line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$ intersect ?
- A segment $\overline{p_1 p_2}$ **straddles** a line if point p_1 lies on one side of the line and point p_2 lies on the other side.
  - A boundary case arises if p_1 or p_2 lies directly on the line.
- Two line segments intersect if and only if either (or both) of the following conditions holds:
  - Each segment straddles the line containing the other.
  - An endpoint of one segment lies on the other segment.
    (This condition comes from the boundary case.)

Pseudocode

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

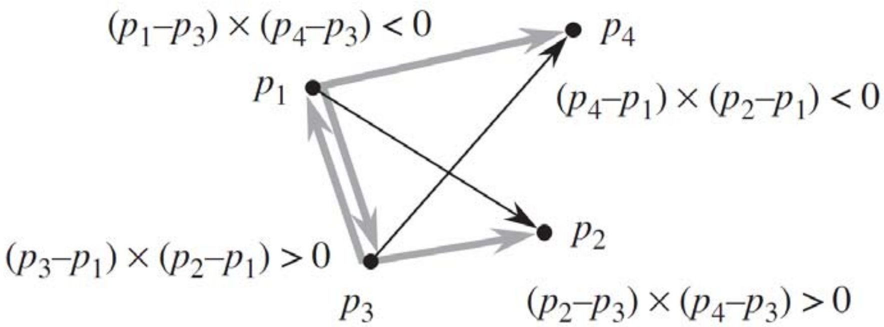
$d_1 \leftarrow$ DIRECTION(p_3, p_4, p_1)
$d_2 \leftarrow$ DIRECTION(p_3, p_4, p_2)
$d_3 \leftarrow$ DIRECTION(p_1, p_2, p_3)
$d_4 \leftarrow$ DIRECTION(p_1, p_2, p_4)
if (($d_1 > 0$ and $d_2 < 0$) or ($d_1 < 0$ and $d_2 > 0$)) and
  (($d_3 > 0$ and $d_4 < 0$) or ($d_3 < 0$ and $d_4 > 0$))
  return TRUE
elseif $d_1 = 0$ and ON-SEGMENT(p_3, p_4, p_1)
  return TRUE
elseif $d_2 = 0$ and ON-SEGMENT(p_3, p_4, p_2)
  return TRUE
elseif $d_3 = 0$ and ON-SEGMENT(p_1, p_2, p_3)
  return TRUE
elseif $d_4 = 0$ and ON-SEGMENT(p_1, p_2, p_4)
  return TRUE
else return FALSE

DIRECTION(p_i, p_j, p_k)

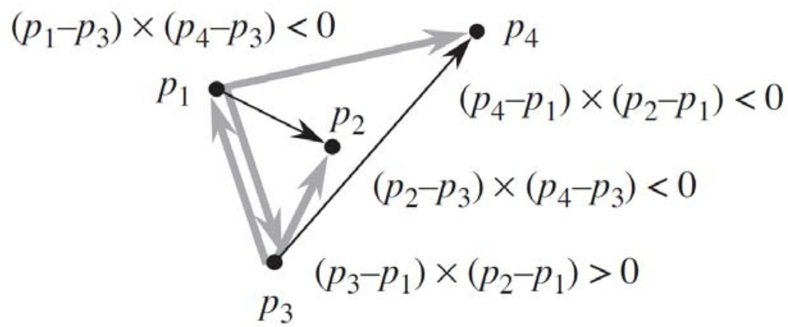
return $(p_k - p_i) \times (p_j - p_i)$

ON-SEGMENT (p_i, p_j, p_k)

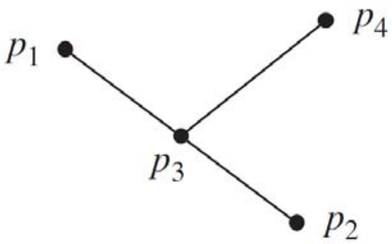
if $\min(x_i, x_j) \le x_k \le \max(x_i, x_j)$ and
  $\min(y_i, y_j) \le y_k \le \max(y_i, y_j)$
  return TRUE
else return FALSE



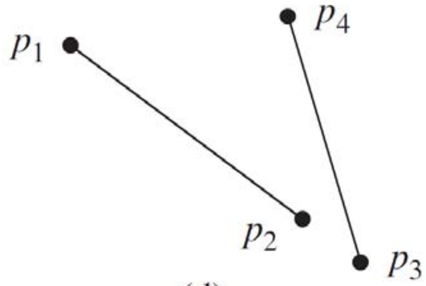
(a)



(b)



(c)



(d)

- Two line segments intersect if and only if conditions (a) or (c) holds.
- In (b), segment $\overline{p_3p_4}$ straddles the line containing $\overline{p_1p_2}$, but segment $\overline{p_1p_2}$ does not straddle the line containing $\overline{p_3p_4}$.
- In (d), p_3 is collinear with $\overline{p_1p_2}$, but it is not between p_1 and p_2. The segments do not intersect.

Outline

- Line-segment properties
- **Determining whether any pair of segments intersects**
- Finding the convex hull
- Finding the closest pair of points

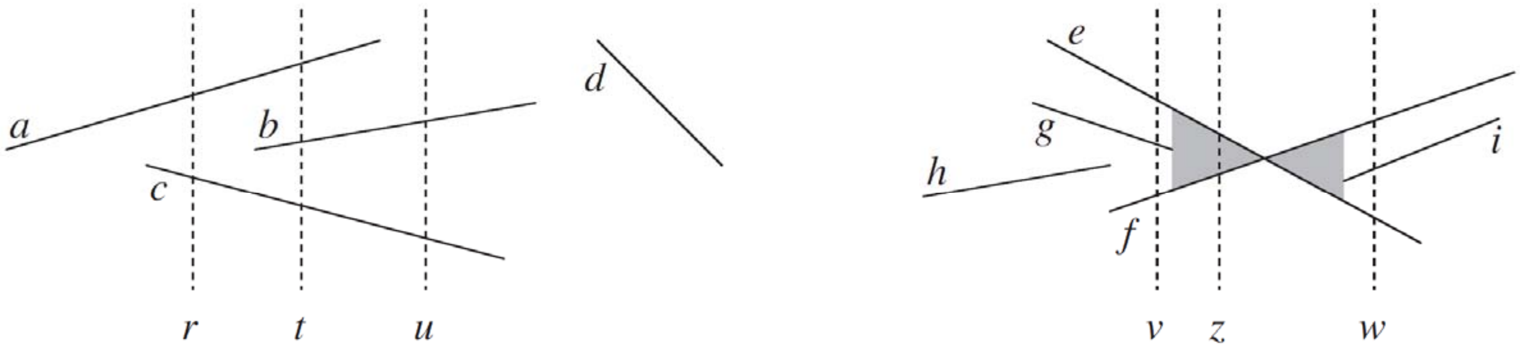
Determining if two line segments intersect ?

- This section presents an algorithm for determining whether any two line segments in a set of segments intersect.
- The algorithm uses a technique known as **sweeping**.
- The algorithm runs in $O(n \lg n)$ time, where n is the number of segments we are given.
- In **sweeping**, an imaginary vertical **sweep line** passes through the given set of geometric objects, usually from left to right.
- We assume that
  - no input segment is vertical; and
  - no three input segments intersect at a single point.

Ordering segments & Moving the sweep line$_{1/2}$

- Two segments s_1 and s_2, are **comparable** at x if the vertical sweep line with x-coordinate x intersects both of them.
- We say that s_1 is **above** s_2 at x, written $s_1 \geq_x s_2$, if
  - the intersection of s_1 with the sweep line at x is higher than the intersection of s_2 with the same sweep line; or
  - if s_1 and s_2 intersect at the sweep line.
- Sweeping algorithms typically manage two sets of data:
  - The **sweep-line status** gives the relationships among the objects intersected by the sweep line.
  - The **event-point schedule** is a sequence of points, called **event point**, ordered from left to right, that defines the halting positions of the sweep line.

Ordering segments & Moving the sweep line$_{2/2}$



(a) (b)

- In (a), we have
  - $a \geq_r c$, $a \geq_t b$, $b \geq_t c$, $a \geq_t c$, and $b \geq_u c$.
  - segment *d* is comparable with no other segment shown.
- In (b), one can see that
  - when segments *e* and *f* intersect, their orders are reversed: we have $e \geq_v f$ but $f \geq_w e$.

Event-point schedule & Sweep-line status

- Event-point schedule:
  - Each segment endpoint is an event point.
  - We sort the segment endpoints by increasing *x*-coordinate and proceed from left to right.
- When we encounter a segment's
  - Left endpoint: insert the segment into the sweep-line status;
  - Right endpoint: delete the segment into the sweep-line status.
- Whenever two segments first become consecutive, we check whether they intersect.

Operations for sweep-line status

- We require the following operations for sweep-line status T:
  - INSERT(T, s): insert segment s into T.
  - DELETE(T, s): delete segment s from T.
  - ABOVE(T, s): return the segment immediately above segment s in T.
  - BELOW(T, s): return the segment immediately below segment s in T.
- Each of the above operations can be performed in $O(\lg n)$ time using red-black trees.
- Recall that the red-black-tree operations in Chapter 13 involve comparing keys.
  - We can replace the key comparisons by comparisons that use cross products to determine the relative ordering of two segments (see Exercise 33.2-2).

Segment-intersection pseudocode

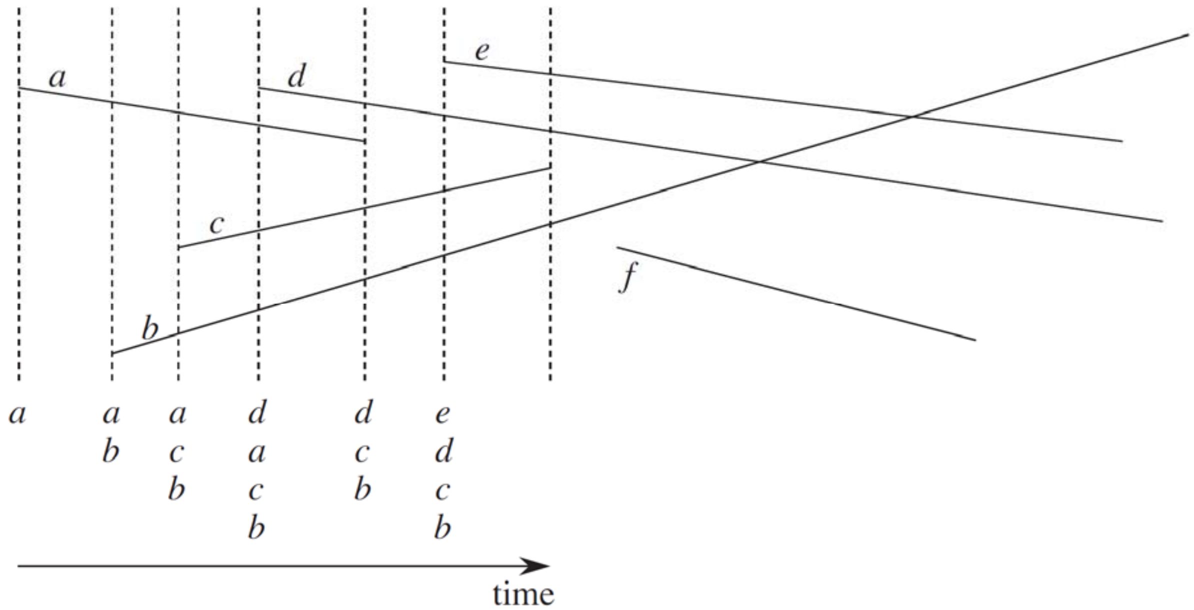
```
ANY-SEGMENTS-INTERSECT(S)
1.  T ← Ø
2.  sort the endpoints of the segments in S from left to right,
      breaking ties by putting left endpoints before right endpoints
      and breaking further ties by putting points with lower
      y-coordinates first
3.  for each point p in the sorted list of 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
```

Lines 2: $O(n \log n)$

Lines 3–11: $2n \cdot (O(\log n) + O(1))$

Time complexity: $O(n \log n)$

The execution of ANY-SEGMENTS-INTERSECT



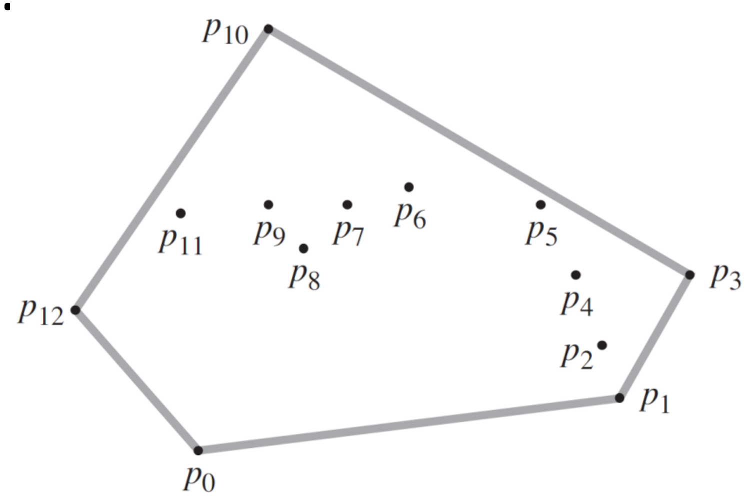
- Each dashed line is the sweep line at an event point.
- The intersection of segments *d* and *b* is found when segment *c* is deleted.

Outline

- Line-segment properties
- Determining whether any pair of segments intersects
- **Finding the convex hull**
- Finding the closest pair of points

Finding the convex hull

- The **convex hull** of a set *Q* of points is the smallest convex polygon *P* for which each point in *Q* is either on the boundary of *P* or in its interior.



- Two algorithms:
  - Graham's scan, runs in $O(n \lg n)$ time, n is the number of points.
  - Jarvis's march, runs in $O(nh)$ time, where h is the number of vertices of the convex hull.

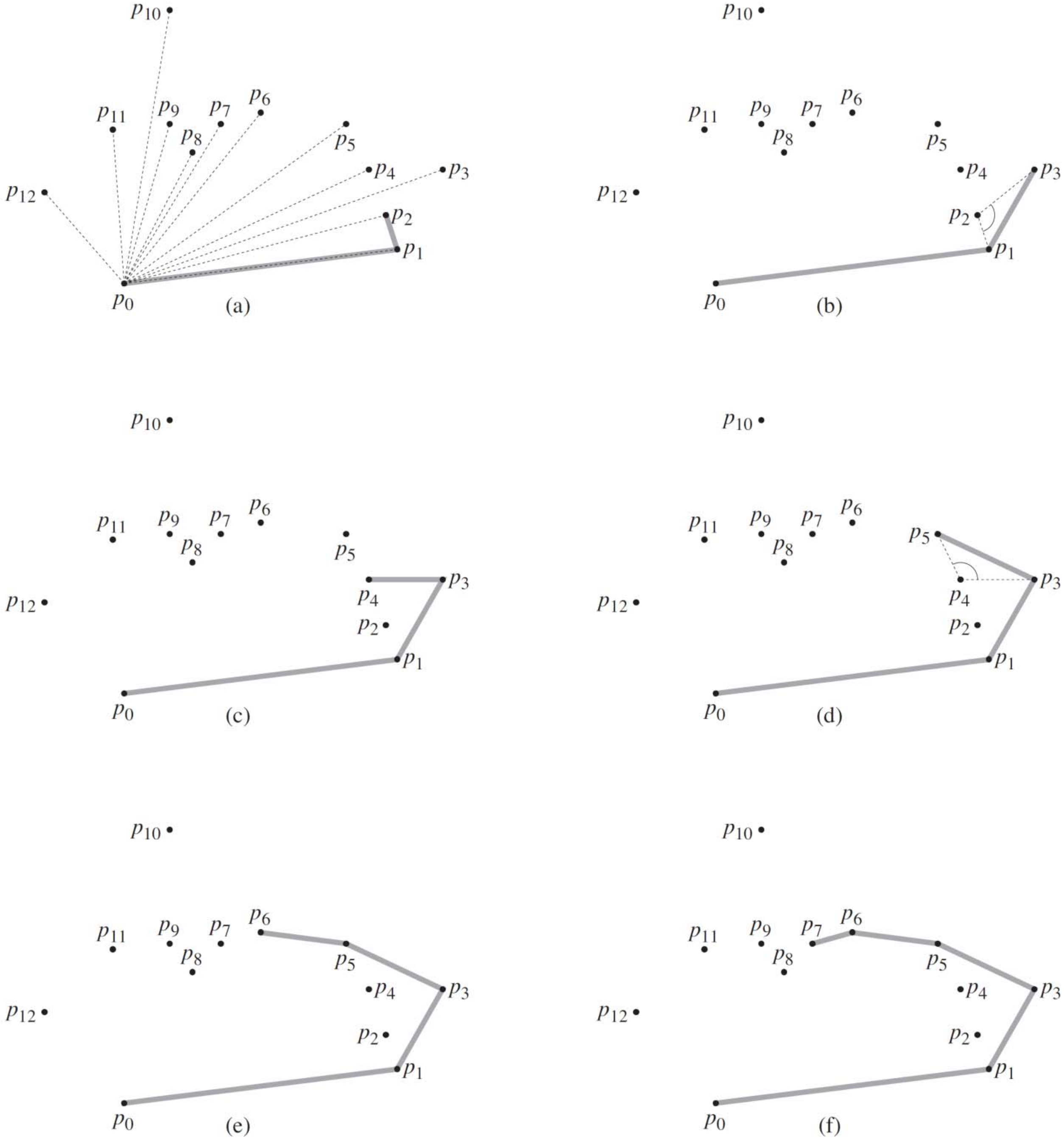
Graham's scan

- Both Graham's scan and Jarvis's march use a technique called **rotational sweep**, processing vertices in the order of the polar angles.
- Graham's scan :
  - By maintaining a stack *S* of candidate points.
  - Each point of the input set *Q* is pushed once onto the stack.
  - The points that are not vertices of CH(*Q*) are eventually popped from the stack.
  - When the algorithm terminates, stack *S* contains exactly the vertices of CH(*Q*).

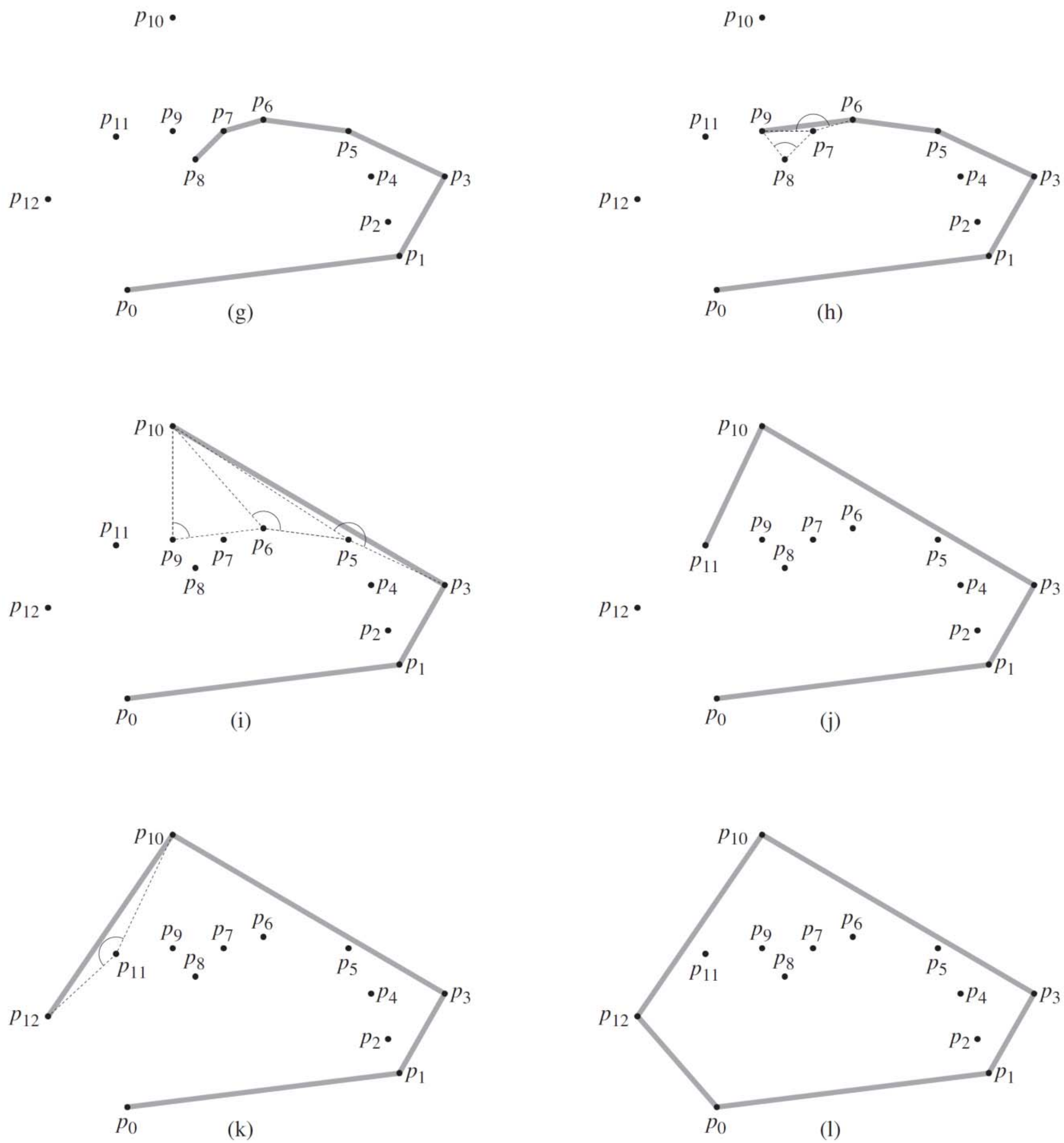
Graham's scan pseudocode

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

Time complexity: $O(n \log n)$



p_{10}
p_{11} p_9 p_7 p_6
p_8 p_5
p_{12} p_4 p_3
p_2
p_1
p_0
(a)
(b)
(c)
(d)
(e)
(f)

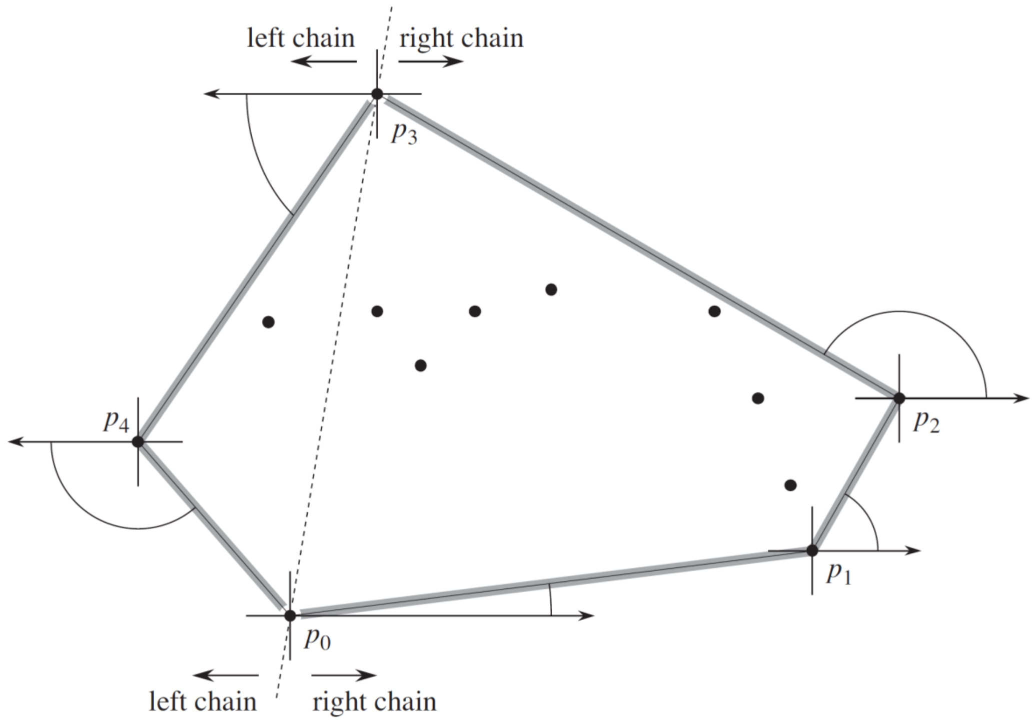


In (h), the right turn at angle $\angle p_7 p_8 p_9$ causes p_8 to be popped, and then the right turn at angle $\angle p_6 p_7 p_9$ causes p_7 to be popped.

Jarvis's march$_{1/2}$

- Jarvis's march computes the convex hull of a set Q of points by a technique known as **package wrapping** (or **gift wrapping**).
- Jarvis's march :
  - Find the lowest point p_0 and the highest point p_k.
  - Construct the **right chain** of CH(Q).
    - We start with p_0, the next convex hull vertex p_1 has the smallest polar angle with respect to p_0.
    - Similarly, p_2 has the smallest polar angle with respect to p_1, and so on.
    - When we reach the highest vertex p_k, we have constructed the right chain of CH(Q).
  - Construct the **left chain** of CH(Q) similarly.

Jarvis's march$_{2/2}$



- Time complexity: $O(nh)$, where h is the # of vertices of CH(Q).
  - Each comparison between polar angles takes $O(1)$ time.

Outline

- Line-segment properties
- Determining whether any pair of segments intersects
- Finding the convex hull
- **Finding the closest pair of points**

Finding the closest pair of points

- Consider the problem of finding the closest pair of points in a set Q of $n \geq 2$ points.
  - **"Closest"** refers to the usual euclidean distance: the distance between points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is

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

- A **brute-force algorithm** simply looks at all the $\binom{n}{2}$ pairs of points.
- In this section, we shall describe a divide-and-conquer algorithm whose running time is described by the familiar recurrence $T(n) = 2T(n/2) + O(n)$.
- Thus, this algorithm uses only $O(n \lg n)$ time.

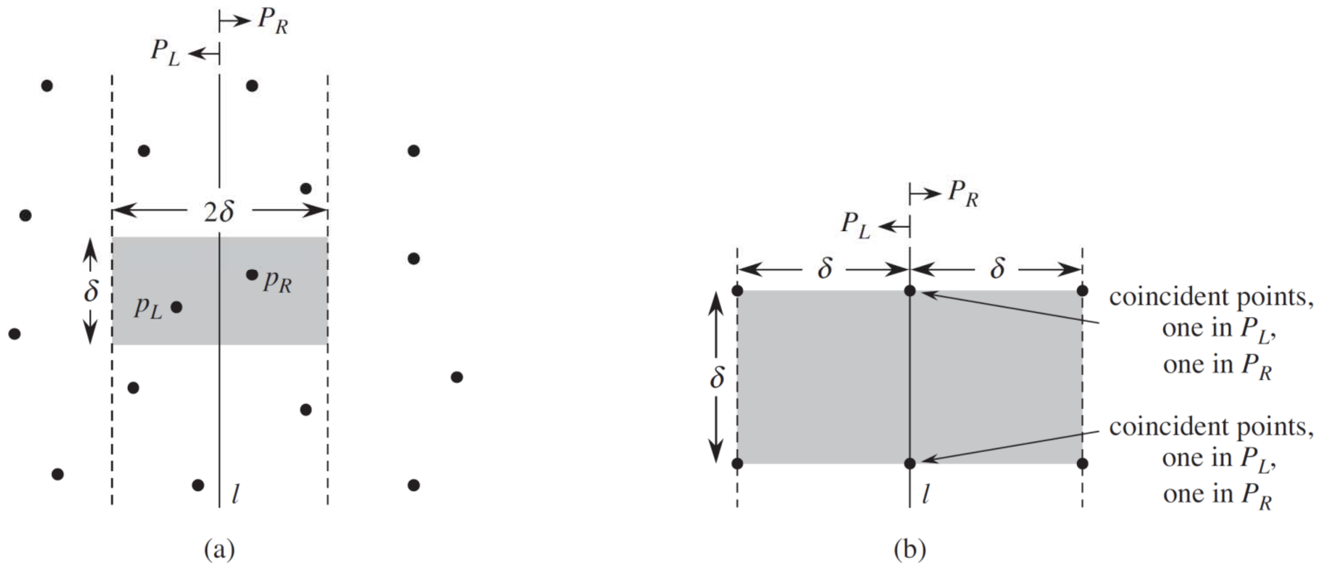
The divide-and-conquer algorithm$_{1/3}$

- The input of each recursive:
  - $P \subseteq Q$.
  - X : contains all the points in P and the points is sorted by monotonically increasing x-coordinates.
  - Y : contains all the points in P and the points is sorted by monotonically increasing y-coordinates.
- If $|P| \leq 3$, perform the brute-force method.
- If $|P| > 3$, recursive invocation carries out the divide-and-conquer paradigm as follows.

The divide-and-conquer algorithm$_{2/3}$

- **Divide:**
  - Find a vertical line ℓ that bisects the point set P into two sets P_L and P_R such that $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$.
  - Divide X into arrays X_L and X_R.
  - Divide Y into arrays Y_L and Y_R.
- **Conquer:**
  - Let the closest-pair distances returned for P_L and P_R be δ_L and δ_R, respectively, and let $\delta = \min(\delta_L, \delta_R)$.
- **Combine:**
  - The closest pair is either the pair with distance δ, or one point in P_L and the other in P_R whose distance is less than δ.
  - If the latter happens, both points of the pair must be within δ units of line ℓ.
  - To find such a pair, if one exists, the algorithm does the following:

The divide-and-conquer algorithm$_{3/3}$



1. It creates an array Y', which is the array Y with all points not in the 2δ-wide vertical strip removed.
2. For each point p in the array Y', try to find points in Y' that are within δ units of p. (Only the **7** points in Y' that follow p need to be considered.)
3. Suppose δ' is closest-pair distance found over all pairs of points in Y'. If $\delta' < \delta$, then return δ'. Otherwise, return δ.

Implementation$_{1/2}$

- Main difficulty:
  - Ensure that arrays X_L, X_R, Y_L, and Y_R, which are passed to recursive calls, are sorted by the proper coordinate.
  - Ensure that array Y' is sorted by y-coordinate.

Implementation$_{2/2}$

- Method:
  - Presort the pints in Q by the proper coordinate to get X and Y before the first recursive call.
  - In each recursive call:
    - Divide P into P_L and P_R ➔ $O(n)$ time.
    - The following pseudocode gives the idea to get Y_L, and Y_R from Y.

$length[Y_L] \leftarrow length[Y_R] \leftarrow 0$
for $i \leftarrow 1$ **to** $length[Y]$
    if $Y[i] \in P_L$
        then $length[Y_L] \leftarrow length[Y_L] + 1$
                $Y_L[length[Y_L]] \leftarrow Y[i]$
        else $length[Y_R] \leftarrow length[Y_R] + 1$
                $Y_R[length[Y_R]] \leftarrow Y[i]$

    - Similar pseudocode works for forming arrays X_L, X_R, and Y'.

Running time

- We get $T'(n) = T(n) + O(n \lg n)$.
  - $T(n)$: the running time of each recursive step.
  - $T'(n)$: the running time of the entire algorithm.

- We can rewrite the recurrence as

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

- Thus, $T(n) = O(n \lg n)$ and $T'(n) = O(n \lg n)$.