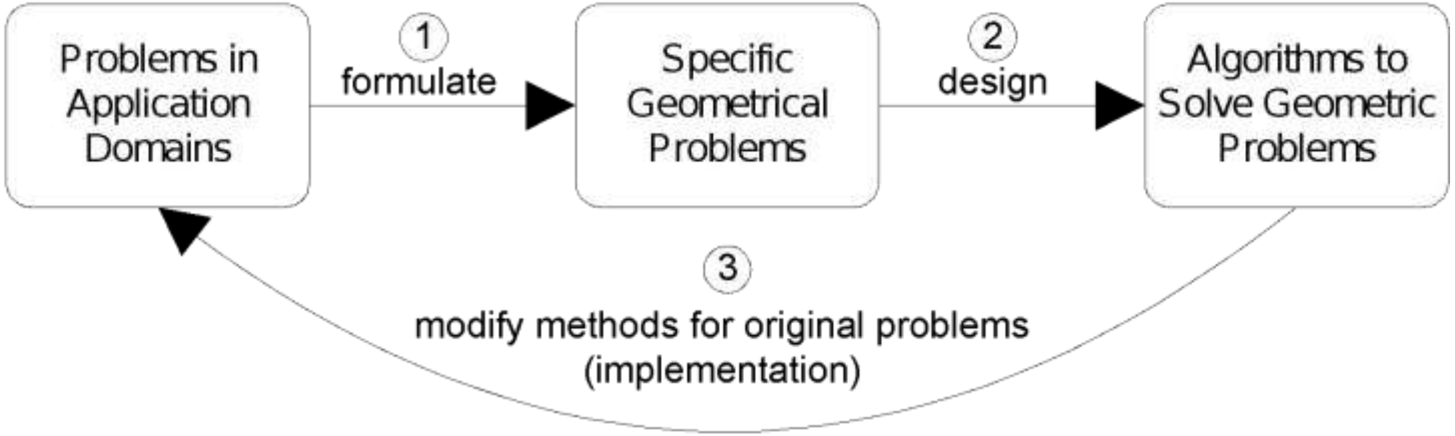
Computational Geometry

Unit 5

What is Computational Geometry?

- Study of Data Structures and Algorithms for Geometric Problems



- 1970′s: Need for Computational Geometry Recognized; Progress made on 1 and 2
- Today: "Mastered" 1 and 2, not so successful with 3

Application Domains

- Computer Graphics and Virtual Reality
  - 2-D & 3-D: intersections, hidden surface elimination, ray tracing
  - Virtual Reality: collision detection (intersection)
- Robotics
  - Motion planning, assembly orderings, collision detection, shortest path finding
- Global Information Systems (GIS)
  - Large Data Sets □ data structure design
  - Overlays □ Find points in multiple layers
  - Interpolation □ Find additional points based on values of known points
  - Voronoi Diagrams of points

Application Domains continued

- Computer Aided Design and Manufacturing (CAD / CAM)
  - Design 3-D objects and manipulate them
    - Possible manipulations: merge (union), separate, move
  - "Design for Assembly"
    - CAD/CAM provides a test on objects for ease of assembly, maintenance, etc.
- Computational Biology
  - Determine how proteins combine together based on folds in structure
    - Surface modeling, path finding, intersection

Geometry

Plane Geometry: the geometry that deals with figures in a two-dimensional PLANE.

Solid Geometry: the geometry that deals with figures in three-dimensional space.

Spherical Geometry: the geometry that deals with figures on the surface of a sphere.

Euclidean Geometry: the geometry (plane and solid) based on Euclid's postulates.

Non-Euclidean Geometry: any geometry that changes Euclid's postulates.

Analytic Geometry: the geometry that deals with the relation between ALGEGRA and geometry, using GRAPHS and EQUATIONS of lines, curves, and surfaces to develop and prove relationships.

Computational Geometry

- Inclusion problems:
  - locating a point in a planar subdivision,
  - reporting which point among a given set are contained in a specified domain, etc.
- Intersection problems:
  - finding intersections of line segments, polygons, circles, rectangles, polyhedra, half spaces, etc.
- Proximity problems:
  - determining the closest pair among a set of given points,
  - computing the smallest distance from one set of points to another.
- Construction problems:
  - identify the convex hull of a polygon,
  - obtaining the smallest box that includes a set of points, etc.

COMPUTATIONAL GEOMETRY

- Computational geometry is the branch of computer science that studies algorithms for solving geometric problems.
- In modern engineering and mathematics, computational geometry has applications in, among other fields, computer graphics, robotics, VLSI design, computer-aided design, and statistics.
- The input to a computational-geometry problem is typically a description of a set of geometric objects, such as a set of points, a set of line segments, or the vertices of a polygon in counter clock-wise order.
- The output is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the convex hull (smallest enclosing convex polygon) of the set of points.

COMPUTATIONAL GEOMETRY

- In this chapter, we look at a few computational-geometry algorithms in two dimensions, that is, in the plane.
- Each input object is represented as a set of points $\{p_i\}$, where each $p_i = (x_i, y_i)$ and $x_i, y_i \in$ R. For example, an n-vertex polygon P is represented by a sequence $\{p_0, p_1, p_2, \ldots, p_n\text{-}1\}$ of its vertices in order of their appearance on the boundary of P.
- Next we see how to answer simple questions about line segments efficiently and accurately: **whether one segment is clockwise or counter clockwise** from another that shares an endpoint, which **way we turn when traversing two adjoining line segments, and whether two line segments intersect.**

Line-segment properties

- Several of the computational-geometry algorithms in this chapter will require answers to questions about the properties of line segments.
- 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 $0 <= \alpha <= 1$, 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 $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Intuitively, p_3 is any point that is on the line passing through p_1 and p_2 and is on or between p_1 and p_2 on the line.
- Given two distinct points p_1 and p_2, the ***line segment*** $\overrightarrow{p_1 p_2}$ is the set of convex combinations of p_1 and p_2.
- We call p_1 and p_2 the ***endpoints*** of segment $\overrightarrow{p_1 p_2}$. Sometimes the ordering of p_1 and p_2 matters, and we speak of the ***directed segment*** $\overrightarrow{p_1 p_2}$. 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.

Line-segment properties

- In this section, we shall explore the following questions:

1. Given two directed segments $\overrightarrow{p_0 p_1}$ and $\overrightarrow{p_0 p_2}$, is $\overrightarrow{p_0 p_1}$ clockwise from $\overrightarrow{p_0 p_2}$ with respect to their common endpoint p_0?
2. Given two line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$, if we traverse $\overline{p_0 p_1}$ and then $\overline{p_1 p_2}$, do we make a left turn at point p_1?
3. Do line segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersect?

- There are no restrictions on the given points.
- We can answer each question in $O(1)$ time, which should come as no surprise since the input size of each question is $O(1)$. Moreover, our methods will use only additions, subtractions, multiplications, and comparisons. We need neither division nor trigonometric functions, both of which can be computationally expensive and prone to problems with round-off error.

Cross Products

- Computing cross products is at the heart of our line-segment methods. Consider vectors p_1 and p_2, shown in figure.
- The ***cross product*** $p_1 \times p_2$ can be interpreted as the signed area of the parallelogram formed by the points (0, 0), p_1, p_2, and $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$.

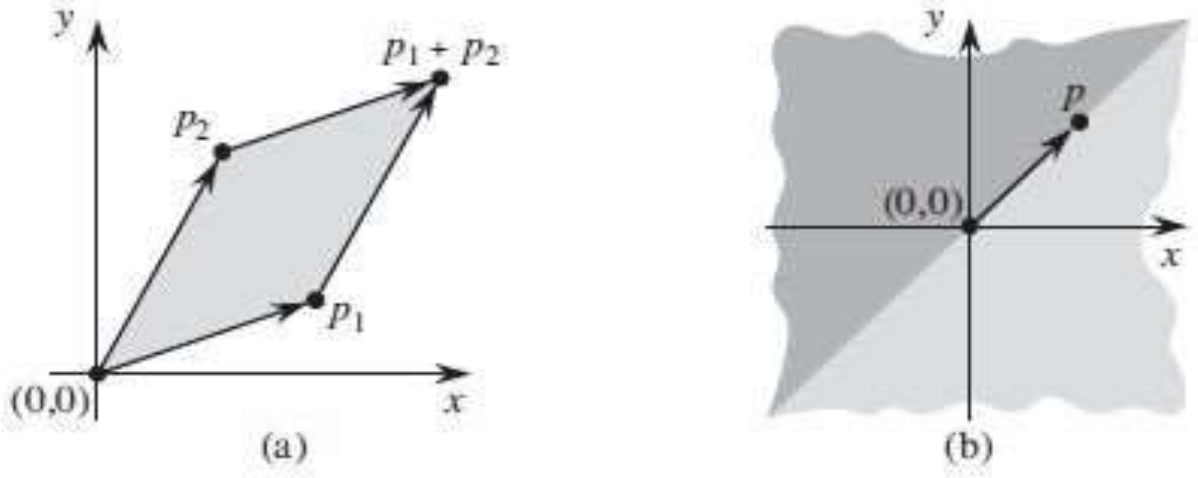


Figure 33.1 (a) The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. (b) The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.

Cross Products

- An equivalent, but more useful, definition gives the cross product as the determinant of a matrix.

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

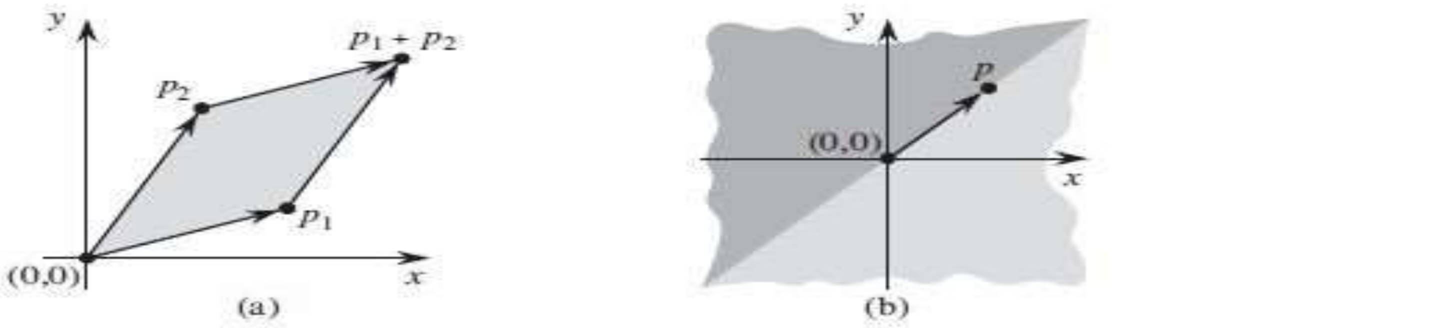


Figure 33.1 **(a)** The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. **(b)** The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.

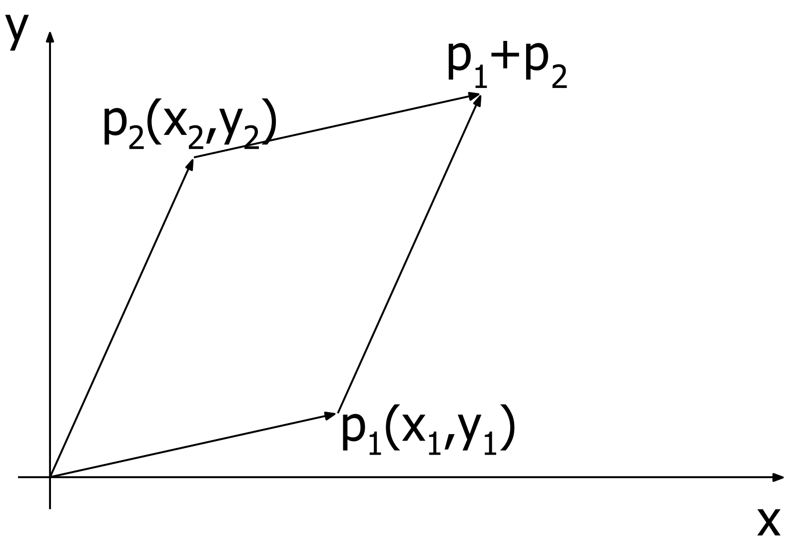
Cross Products

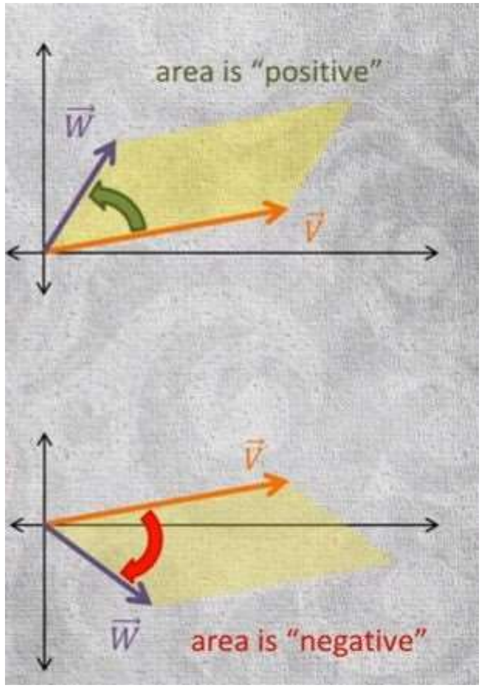
- Can solve many geometric problems using the cross product.
- Two points: $p_1=(x_1,y_1)$, $p_2=(x_2,y_2)$
- Cross product of the two points is defined by

  $p_1 \times p_2$ = the determinant of a matrix $\begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix}$

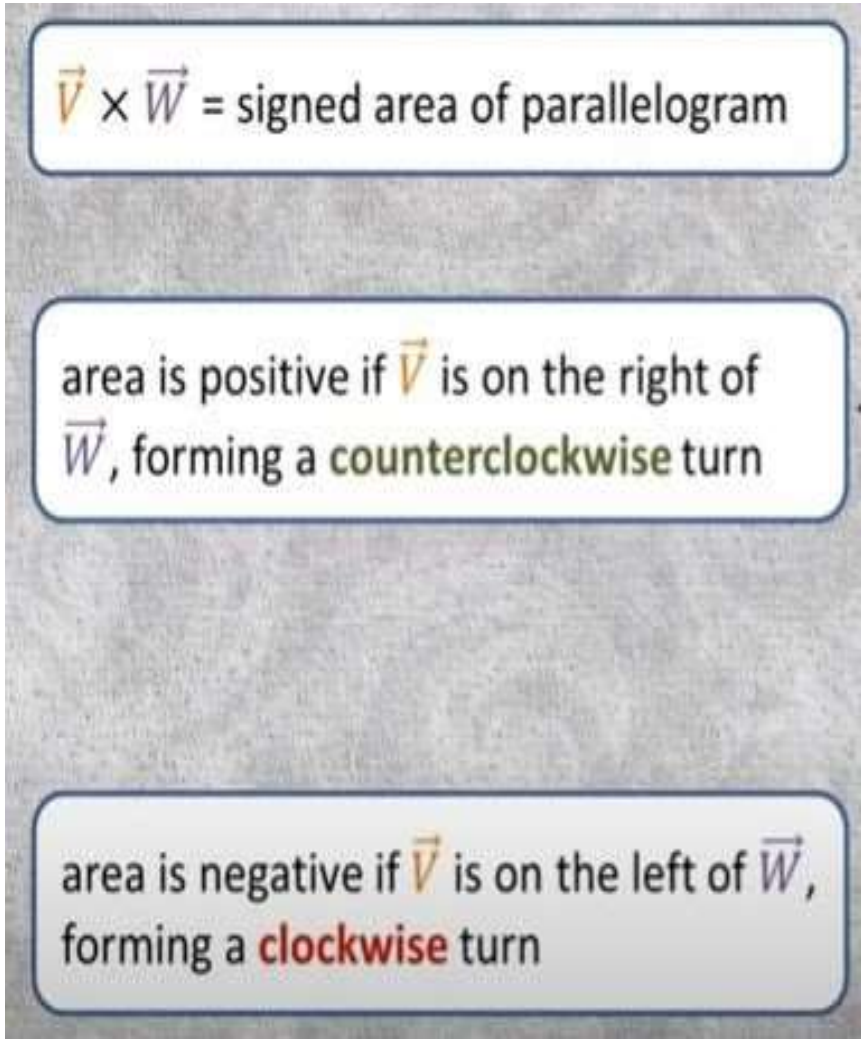
  $= x_1y_2 - x_2y_1$

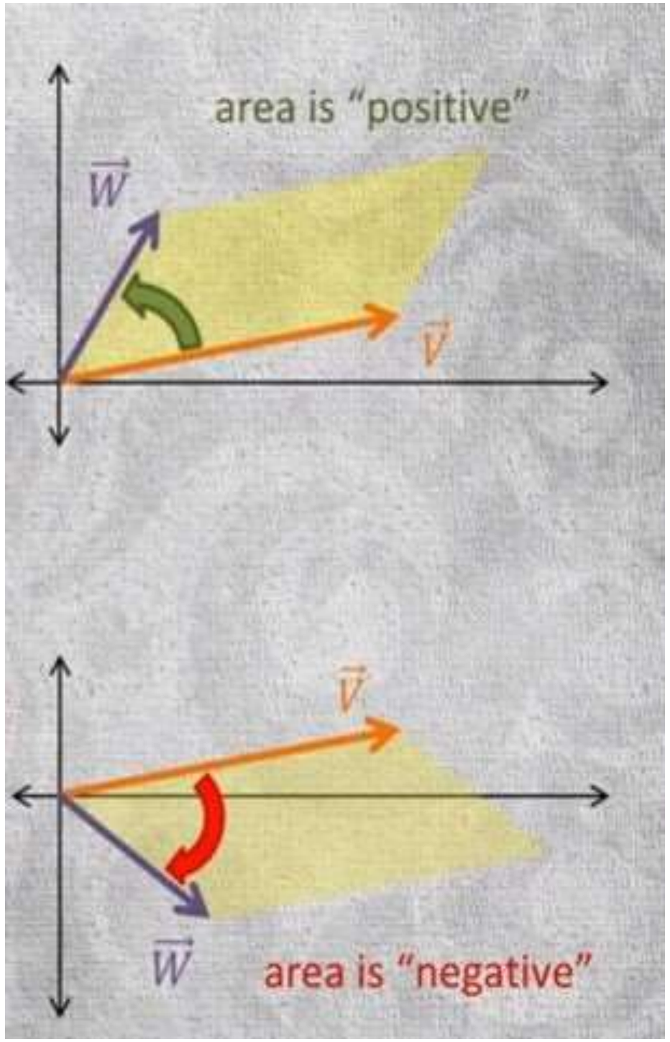
  = the signed area of the parallelogram of four points: (0,0), p_1, p_2, p_1+p_2
- What happens if $p_1 \times p_2 = 0$?
- If $p_1 \times p_2$ is positive then p_1 is clockwise from p_2
- If $p_1 \times p_2$ is negative then p_1 is counterclockwise from p_2





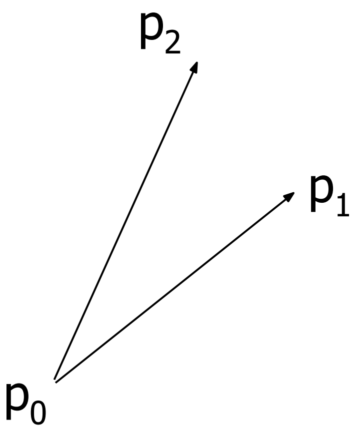
Cross Products



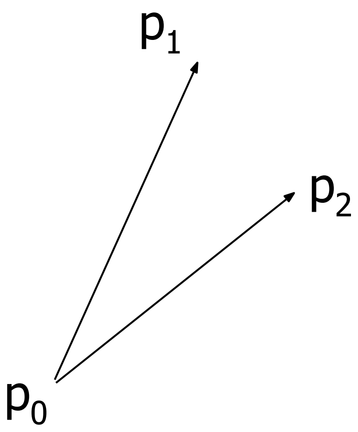


Clockwise or Counterclockwise

- Problem definition
  - Determine whether a directed segment p_0p_1 is clockwise from a directed segment p_0p_2 w.r.t. p_0



(a) p_0p_2 is counterclockwise from p_0p_1: $(p_2-p_0)x(p_1-p_0) < 0$



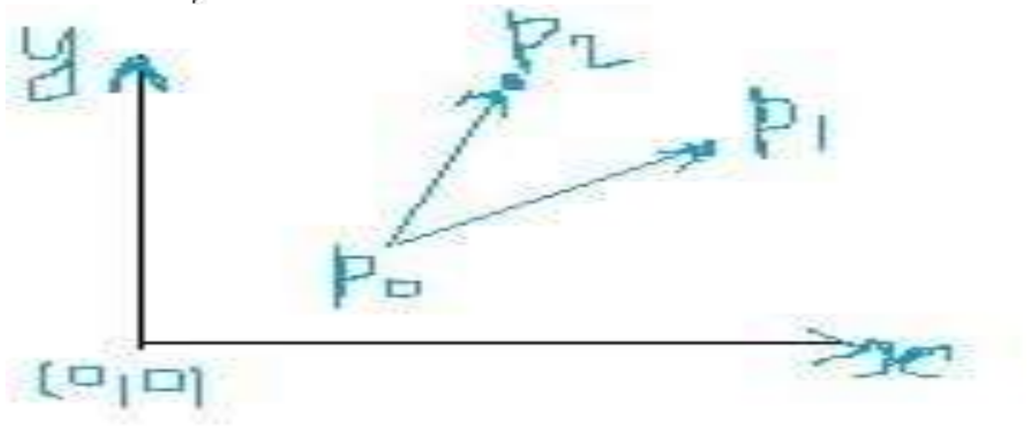
(b) p_0p_2 is clockwise from p_0p_1: $(p_2-p_0)x(p_1-p_0) > 0$

Clockwise or Counterclockwise

- Problem definition
  - Determine whether a directed segment p_0p_1 is clockwise from a directed segment p_0p_2 w.r.t. p_0
- Solution
  1. Map p_0 to (0,0), p_1 to p_1', p_2 to p_2'
     - $p_1' = p_1 - p_0$, $p_2' = p_2 - p_0$

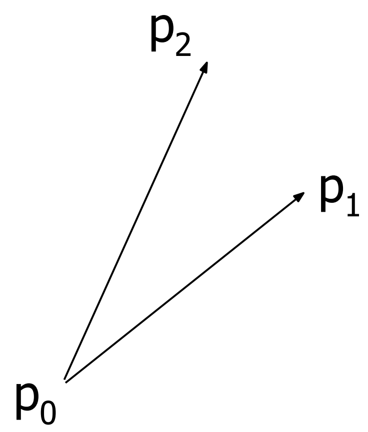
let $p_1 - p_0$ denote the vector $p_1' = (x_1', y_1')$, where $x_1' = x_1 - x_0$ and $y_1' = y_1 - y_0$, and we define $p_2 - p_0$ similarly. We then compute the cross product

$$\begin{pmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{pmatrix} = (p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

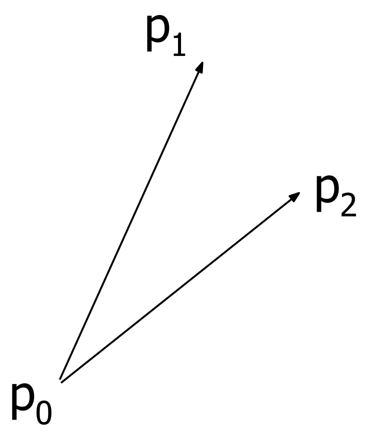


Clockwise or Counterclockwise

- Problem definition
  - Determine whether a directed segment p_0p_1 is clockwise from a directed segment p_0p_2 w.r.t. p_0
- Solution
  1. Map p_0 to (0,0), p_1 to p_1', p_2 to p_2'
     - $p_1' = p_1 - p_0$, $p_2' = p_2 - p_0$
  2. If $p_1' \times p_2' > 0$ then the segment p_0p_1 is clockwise from p_0p_2
  3. else counterclockwise



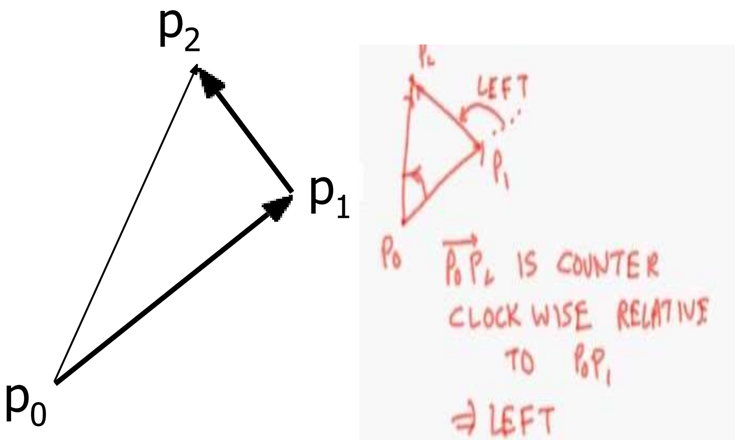
(a) p_0p_2 is counterclockwise from p_0p_1: $(p_2-p_0)x(p_1-p_0) < 0$



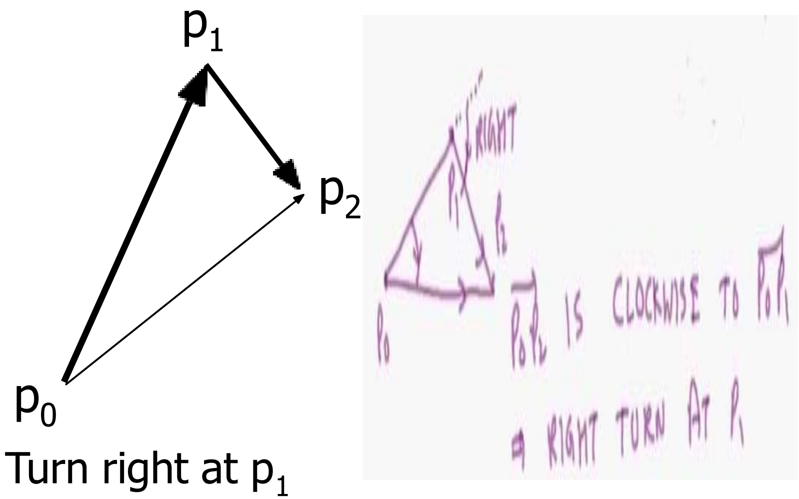
(b) p_0p_2 is clockwise from p_0p_1: $(p_2-p_0)x(p_1-p_0) > 0$

Turn Left or Turn Right

- Problem definition
  - Determine whether two consecutive line segments p_0p_1 and p_1p_2 turn left or right at the common point p_1.
- Solution
  - Determine p_0p_2 is clockwise or counterclockwise from p_0p_1 w.r.t. p_0
  - If counterclockwise then turn left at p_0
  - else turn right at p_0



(a) Turn left at p_1
$(p_2-p_0)x(p_1-p_0) < 0$

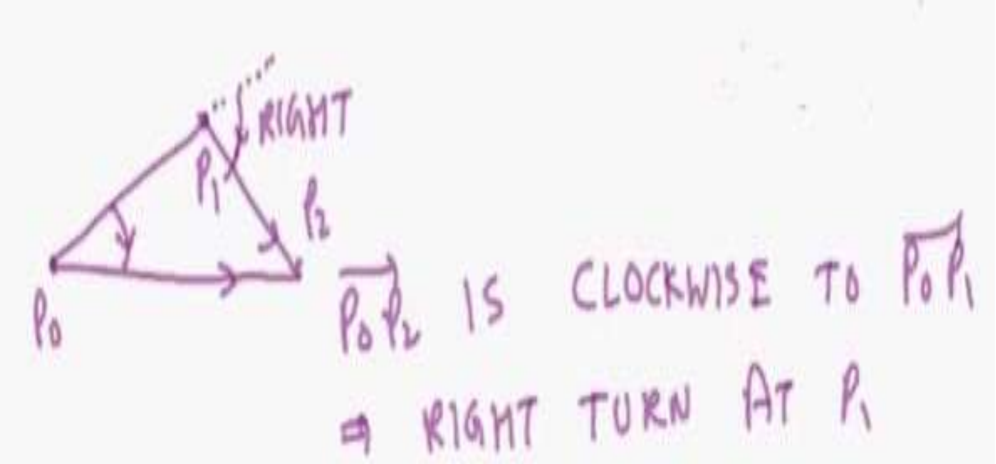


(b) Turn right at p_1
$(p_2-p_0)x(p_1-p_0) > 0$

Turn Left or Turn Right

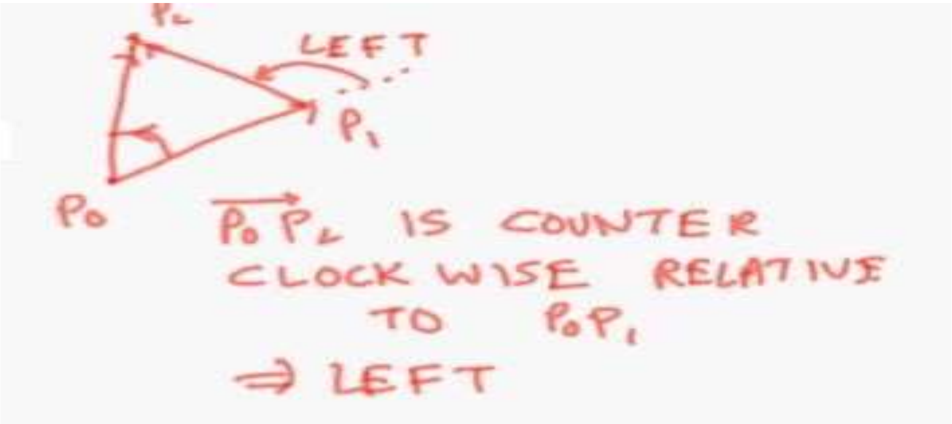
$(p_2\text{-}p_0)x(p_1\text{-}p_0)$

- If zero then collinear
- If positive then turn right
- If negative then turn left



(a) Turn right at p_1

$(p_2\text{-}p_0)x(p_1\text{-}p_0) > 0$

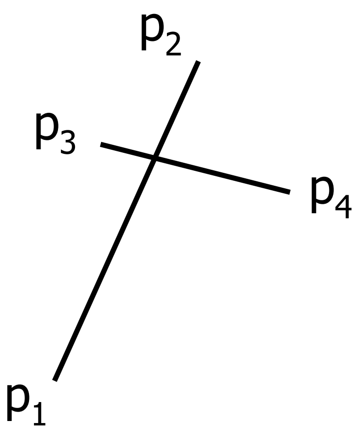


(b) Turn left at p_1

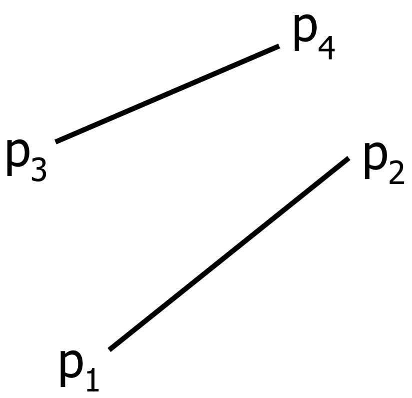
$(p_2\text{-}p_0)x(p_1\text{-}p_0) < 0$

Two Segments Intersect

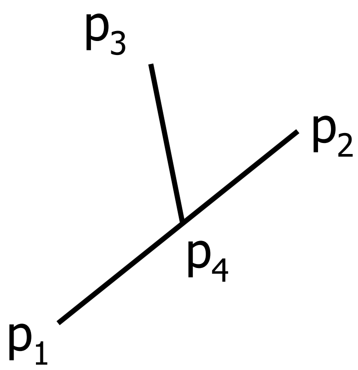
- Five cases



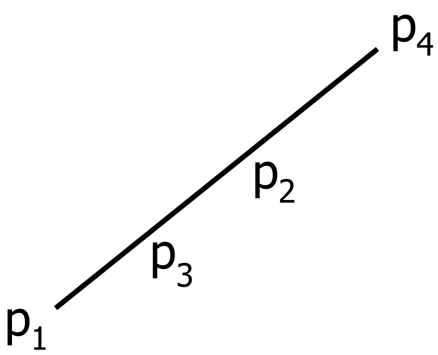
(a) p_1p_2, p_3p_4 intersect



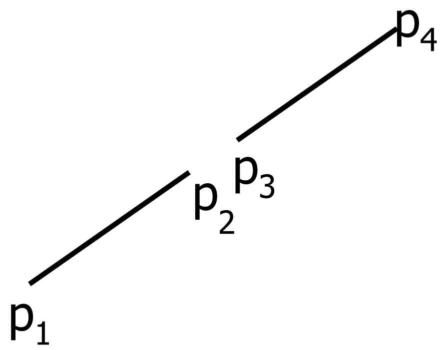
(b) p_1p_2, p_3p_4 do not intersect



(c) p_1p_2, p_3p_4 intersect



(d) p_1p_2, p_3p_4 intersect

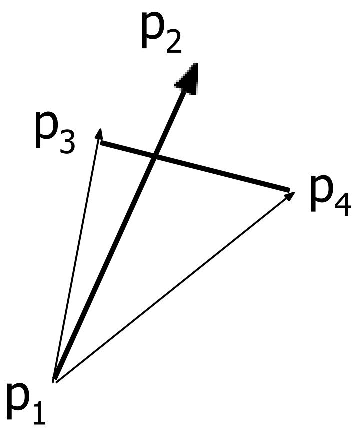


(e) p_1p_2, p_3p_4 do not intersect

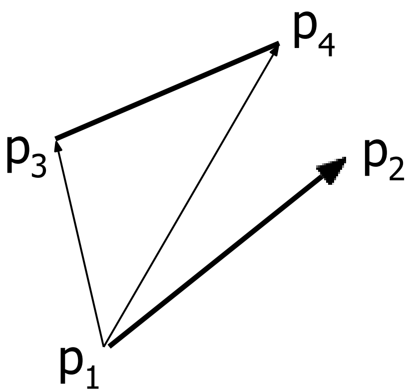
Two Segments Intersect

- Consider two cross products:

1. $(p_3 - p_1) \times (p_2 - p_1)$
2. $(p_4 - p_1) \times (p_2 - p_1)$

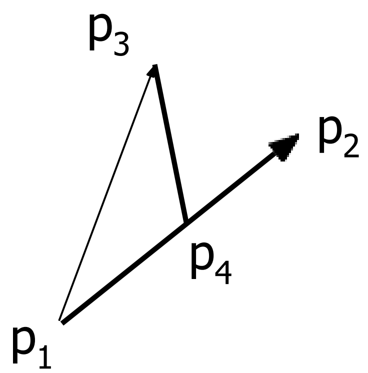


(a) p_1p_2, p_3p_4 intersect:
$(p_3 - p_1) \times (p_2 - p_1) < 0$
$(p_4 - p_1) \times (p_2 - p_1) > 0$



(b) p_1p_2, p_3p_4 do not intersect:
$(p_3 - p_1) \times (p_2 - p_1) < 0$
$(p_4 - p_1) \times (p_2 - p_1) < 0$

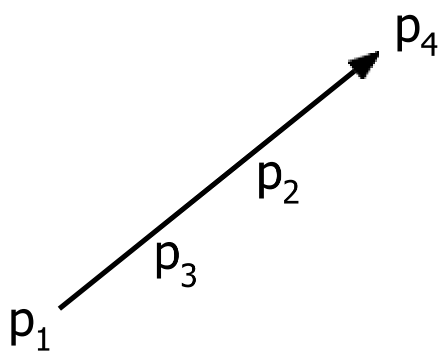
Two Segments Intersect



(c) p_1p_2, p_3p_4 intersect

$(p_3 - p_1) \times (p_2 - p_1) < 0$

$(p_4 - p_1) \times (p_2 - p_1) = 0$



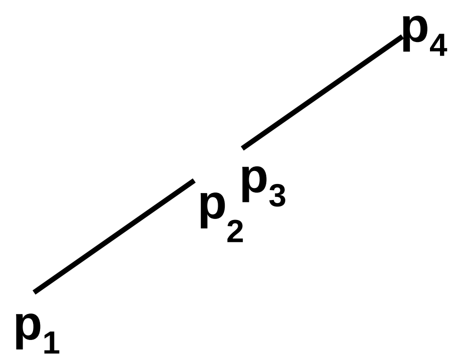
(d) p_1p_2, p_3p_4 intersect

$(p_3 - p_1) \times (p_2 - p_1) = 0$

$(p_4 - p_1) \times (p_2 - p_1) = 0$

Two Segments Intersect

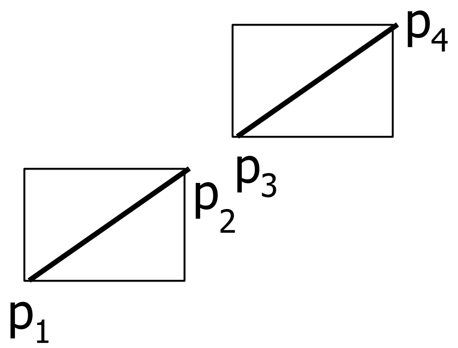
- Case (e)
  - What are the cross products ?
    - $(p_3 - p_1) \times (p_2 - p_1) = 0$
    - $(p_4 - p_1) \times (p_2 - p_1) = 0$
  - The cross products are zero's, but they do not intersect
  - Same result with Case (d)



(e) p_1p_2, p_3p_4 do not intersect

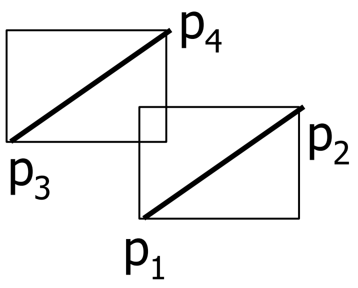
Bounding Boxes

- Definition
  - Given a geometric object, the bounding box is defined by the smallest rectangle that contains the object
- Given two line segments p_1p_2 and p_3p_4
- The bounding box of the line segment p_1p_2
  - The rectangle with lower left point=$(x_1',y_1')=(\min(x_1,x_2), \min(y_1,y_2))$ and
  - upper right point=$(x_2',y_2')= (\max(x_1,x_2), \max(y_1,y_2))$
- The bounding box of the line segment p_3p_4 is
  - The rectangle with lower left point=$(x_3',y_3')=(\min(x_3,x_4), \min(y_3,y_4))$ and
  - upper right point=$(x_4',y_4')=(\max(x_3,x_4), \max(y_3,y_4))$



Bounding Boxes

- What is the condition for the two bounding boxes intersect?
  - $(x_3' \le x_2')$ and $(x_1' \le x_4')$ and $(y_3' \le y_2')$ and $(y_1' \le y_4')$
- If two bounding boxes do not intersect, then the two line segments do not intersect.
- But it is not always true that
  - if two bounding boxes intersect, then the two line segments intersect
  - e.g., the figure



Two Segments Intersect (6)

- Case summary
  - (a) p_1p_2, p_3p_4 intersect
    - $(p_3 - p_1) \times (p_2 - p_1) < 0$
    - $(p_4 - p_1) \times (p_2 - p_1) > 0$
  - (b) p_1p_2, p_3p_4 do not intersect
    - $(p_3 - p_1) \times (p_2 - p_1) < 0$
    - $(p_4 - p_1) \times (p_2 - p_1) < 0$
  - (c) p_1p_2, p_3p_4 intersect
    - $(p_3 - p_1) \times (p_2 - p_1) < 0$
    - $(p_4 - p_1) \times (p_2 - p_1) = 0$
  - (d) p_1p_2, p_3p_4 intersect
    - $(p_3 - p_1) \times (p_2 - p_1) = 0$
    - $(p_4 - p_1) \times (p_2 - p_1) = 0$
  - (e) p_1p_2, p_3p_4 do not intersect
    - $(p_3 - p_1) \times (p_2 - p_1) = 0$
    - $(p_4 - p_1) \times (p_2 - p_1) = 0$

Two Segments Intersect - Algorithm

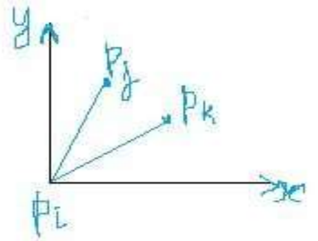
```
SEGMENT-INTERSECT(p1, p2, p3, p4)
d1 = DIRECTION(p3, p4, p1) ; d2 = DIRECTION(p3, p4, p2)
d3 = DIRECTION(p1, p2, p3); d4 = DIRECTION(p1, p2, p4)
 if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and ((d3 > 0
 and d4 < 0) or (d3 < 0 and d4 > 0))
        return TRUE
 else if d1 == 0 and ON-SEGMENT(p3, p4, p1)
        return TRUE
 else if d2 == 0 and ON-SEGMENT(p3, p4, p2)
        return TRUE
 else if d3 == 0 and ON-SEGMENT(p1, p2, p3)
        return TRUE
 else if d4 == 0 and ON-SEGMENT(p1, p2, p4)
        return TRUE
 else return FALSE
```

Two Segments Intersect - Algorithm

```
SEGMENT-INTERSECT(p1, p2, p3, p4)
  d1 = DIRECTION(p3, p4, p1) ; d2 = DIRECTION(p3, p4, p2)
  d3 = DIRECTION(p1, p2, p3); d4 = DIRECTION(p1, p2, p4)
   if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and ((d3 > 0
   and d4 < 0) or (d3 < 0 and d4 > 0))
          return TRUE
   else if d1 == 0 and ON-SEGMENT(p3, p4, p1)
          return TRUE
   else if d2 == 0 and ON-SEGMENT(p3, p4, p2)
          return TRUE
   else if d3 == 0 and ON-SEGMENT(p1, p2, p3)
          return TRUE
   else if d4 == 0 and ON-SEGMENT(p1, p2, p4)
          return TRUE
   else return FALSE
```

DIRECTION(p_i, p_j, p_k)

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

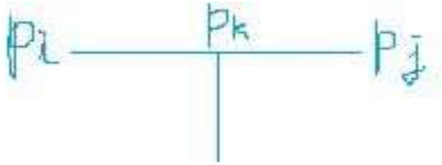


ON-SEGMENT(p_i, p_j, p_k)

1 **if** $\min(x_i, x_j) \le x_k \le \max(x_i, x_j)$ and $\min(y_i, y_j) \le y_k \le \max(y_i, y_j)$

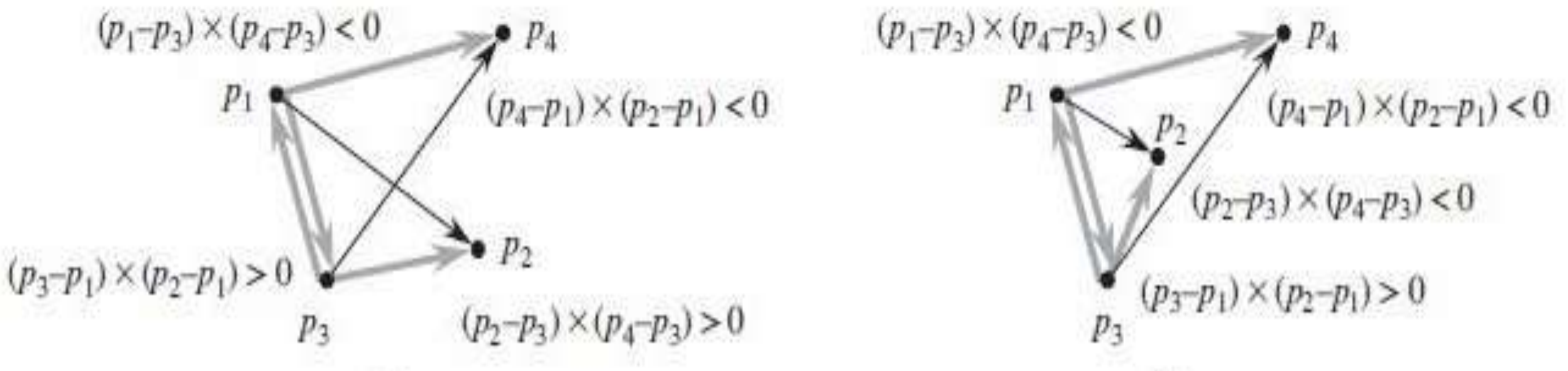
2 **return** TRUE

3 **else return** FALSE



Two Segments Intersect - Algorithm

```
d1 = DIRECTION(p3, p4, p1) ; d2 = DIRECTION(p3, p4, p2)
d3 = DIRECTION(p1, p2, p3); d4 = DIRECTION(p1, p2, p4)
 if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and ((d3 > 0
 and d4 < 0) or (d3 < 0 and d4 > 0))
        return TRUE
```

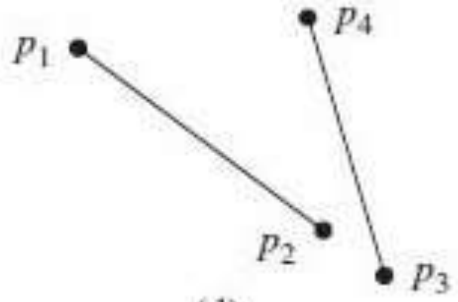


(a) (b)

Two Segments Intersect - Algorithm

```
else if d_1 == 0 and ON-SEGMENT(p_3, p_4, p_1)
        return TRUE
else if d_2 == 0 and ON-SEGMENT(p_3, p_4, p_2)
        return TRUE
else if d_3 == 0 and ON-SEGMENT(p_1, p_2, p_3)
        return TRUE
else if d_4 == 0 and ON-SEGMENT(p_1, p_2, p_4)
        return TRUE
else return FALSE
```

(c)



(d)

Line Segment Intersection Problem

- The Problem: Given n line segments, is any pair of them insect?
- Clearly, doing pairwise intersection testing takes $O(n^2)$ time. By a sweeping technique, we can solve it in

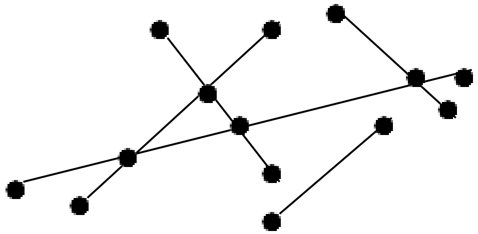
  $O(n \log n)$ time (without printing all the intersections).
- In the worst case, there are $\Omega(n^2)$ intersections.
- Simplifying assumptions
  - No input segment is vertical.
  - No three input segments intersect at a single point.

Determining whether any pair of segments intersects

Problem Definition:

Input : $S=\{s_1, s_2, \ldots, s_n\}$ of n segments in plane.

Output: set I of intersection points among segments in S. (with segments containing each intersection pt)



How many intersections possible?

In worst case, $\binom{n}{2} = \theta(n^2)$

Idea

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," which is common to many computational-geometry algorithms.

In *sweeping*, an imaginary vertical *sweep line* passes through the given set of geometric objects, usually from left to right. We treat the spatial dimension that the sweep line moves across, in this case the x-dimension, as a dimension of time. Sweeping provides a method for ordering geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of relationships among them. The line-segment-intersection algorithm in this section considers all the line-segment endpoints in left-to-right order and checks for an intersection each time it encounters an endpoint.

Idea

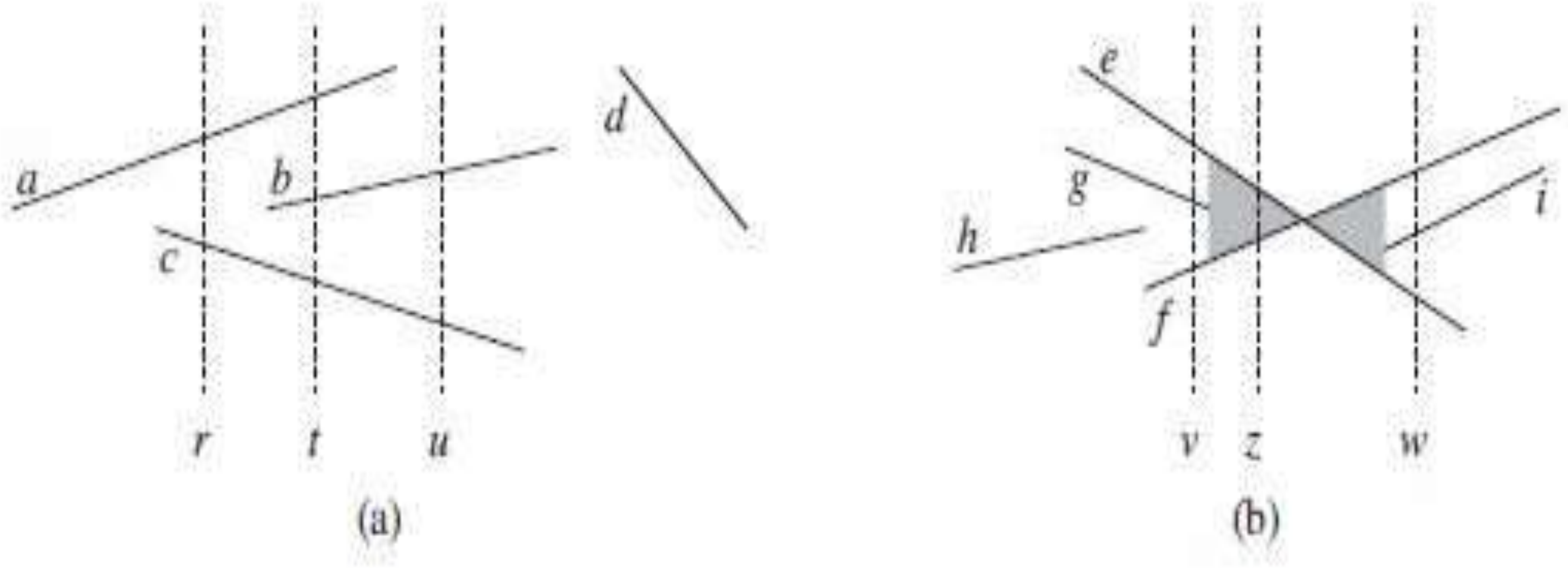


Figure 33.4 The ordering among line segments at various vertical sweep lines. **(a)** We have $a \succcurlyeq_r c$, $a \succcurlyeq_t b$, $b \succcurlyeq_t c$, $a \succcurlyeq_t c$, and $b \succcurlyeq_u c$. Segment d is comparable with no other segment shown. **(b)** When segments e and f intersect, they reverse their orders: we have $e \succcurlyeq_v f$ but $f \succcurlyeq_w e$. Any sweep line (such as z) that passes through the shaded region has e and f consecutive in the ordering given by the relation $\succcurlyeq_z$.

Assumption

To describe and prove correct our algorithm for determining whether any two of n line segments intersect, we shall make two simplifying assumptions. First, we assume that no input segment is vertical. Second, we assume that no three input segments intersect at a single point.

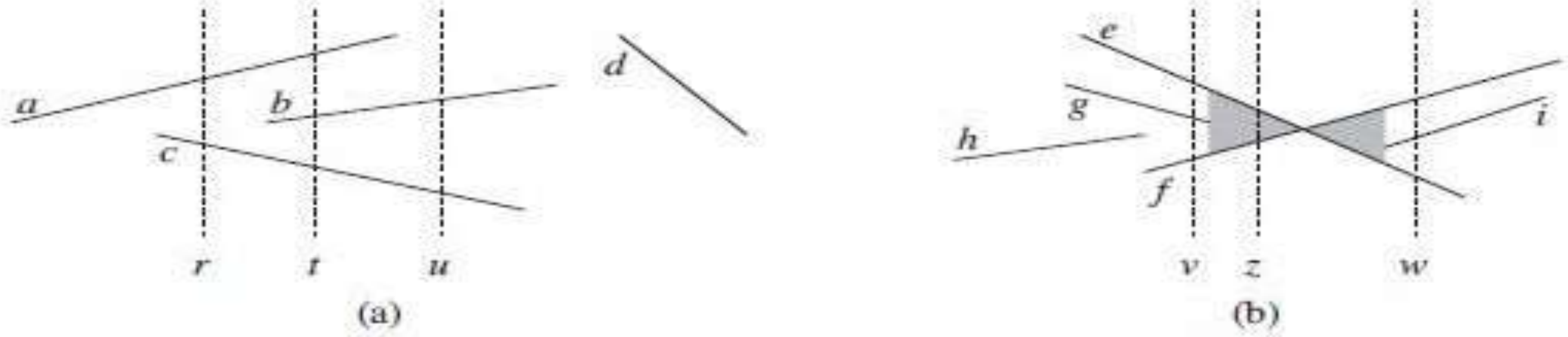


Figure 33.4 The ordering among line segments at various vertical sweep lines. **(a)** We have $a \succeq_r c$, $a \succeq_t b$, $b \succeq_t c$, $a \succeq_t c$, and $b \succeq_u c$. Segment d is comparable with no other segment shown. **(b)** When segments e and f intersect, they reverse their orders: we have $e \succeq_v f$ but $f \succeq_w e$. Any sweep line (such as z) that passes through the shaded region has e and f consecutive in the ordering given by the relation $\succeq_z$.

Plane Sweep Algorithm

Plane Sweep Algorithm:

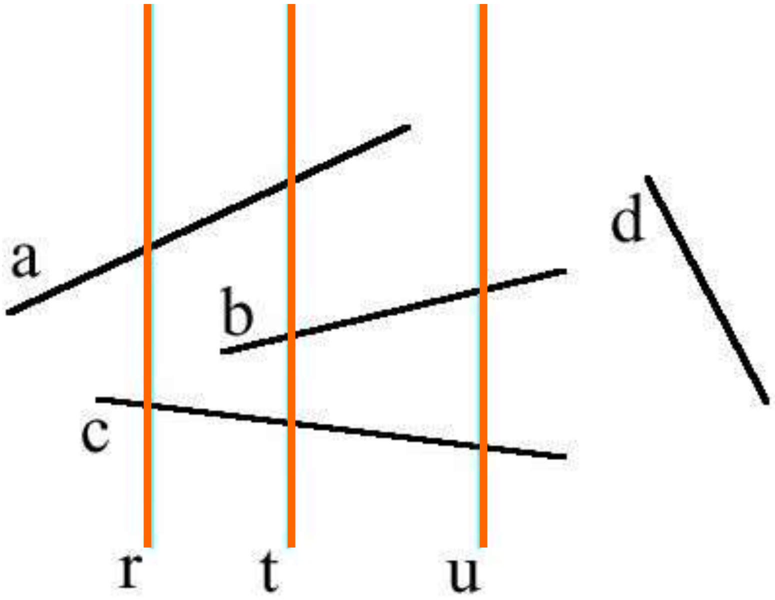
- L is vertical sweep line initially left of all segments.
- Sweep L right over segments, and keep track of all segments intersecting it.
- Status T of sweep line is the set of segments currently intersecting L .
- Events are points where status changes.
- At each event point
  - Update status of sweep line: add/remove segments from T
  - Perform intersection tests

Ordering Segments

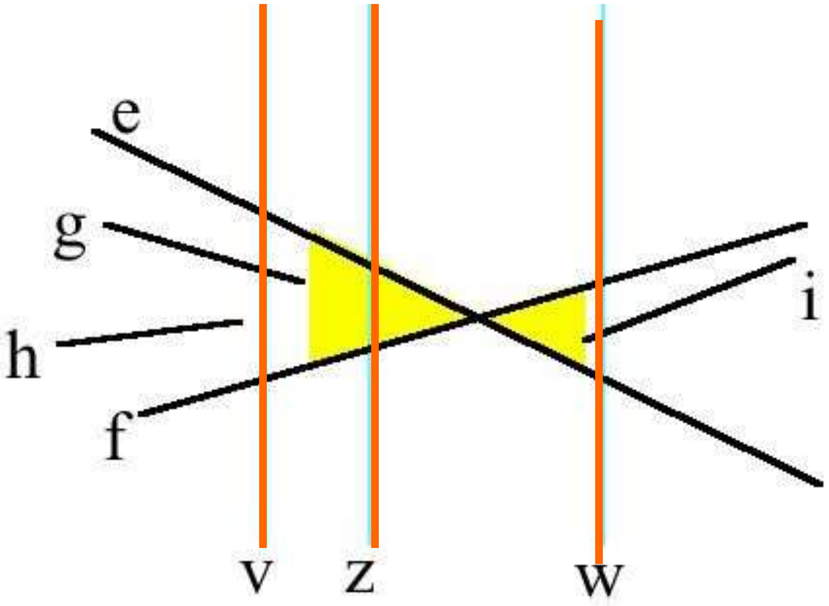
- Two nonintersecting segments s_1 and s_2 are comparable at x if the vertical sweep line with x-coordinate x intersects both of them.

- s_1 is above s_2 at x, written $s_1 >_x s_2$, if s_1 and s_2 are comparable at x and the intersection of s_1 with the sweep line at x is higher than the intersection of s_2 with the same sweep line.

Example



$a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$, and $b >_u c$.
d is comparable with no other segment shown.



When segments e and f intersect, their orders are reversed: we have $e >_v f$ but $f >_w e$.

Moving the Sweep Line

- Sweeping algorithms typically manage two sets of data:
  1. The **sweep-line status** gives the relationships among the objects intersected by the sweep line.
  2. The **event-point schedule** is a sequence of x-coordinates, ordered from left to right, that defines the halting positions (event points) of the sweep line. Changes to the sweep-line status occur only at event points.

Moving the Sweep Line

- Sort the segment endpoints by increasing x-coordinate and proceed from left to right.
- Insert a segment into the sweep-line status when its left endpoint is encountered, and delete it from the sweep-line status when its right endpoint is encountered.
- Whenever two segments first become consecutive in the total order, we check whether they intersect.

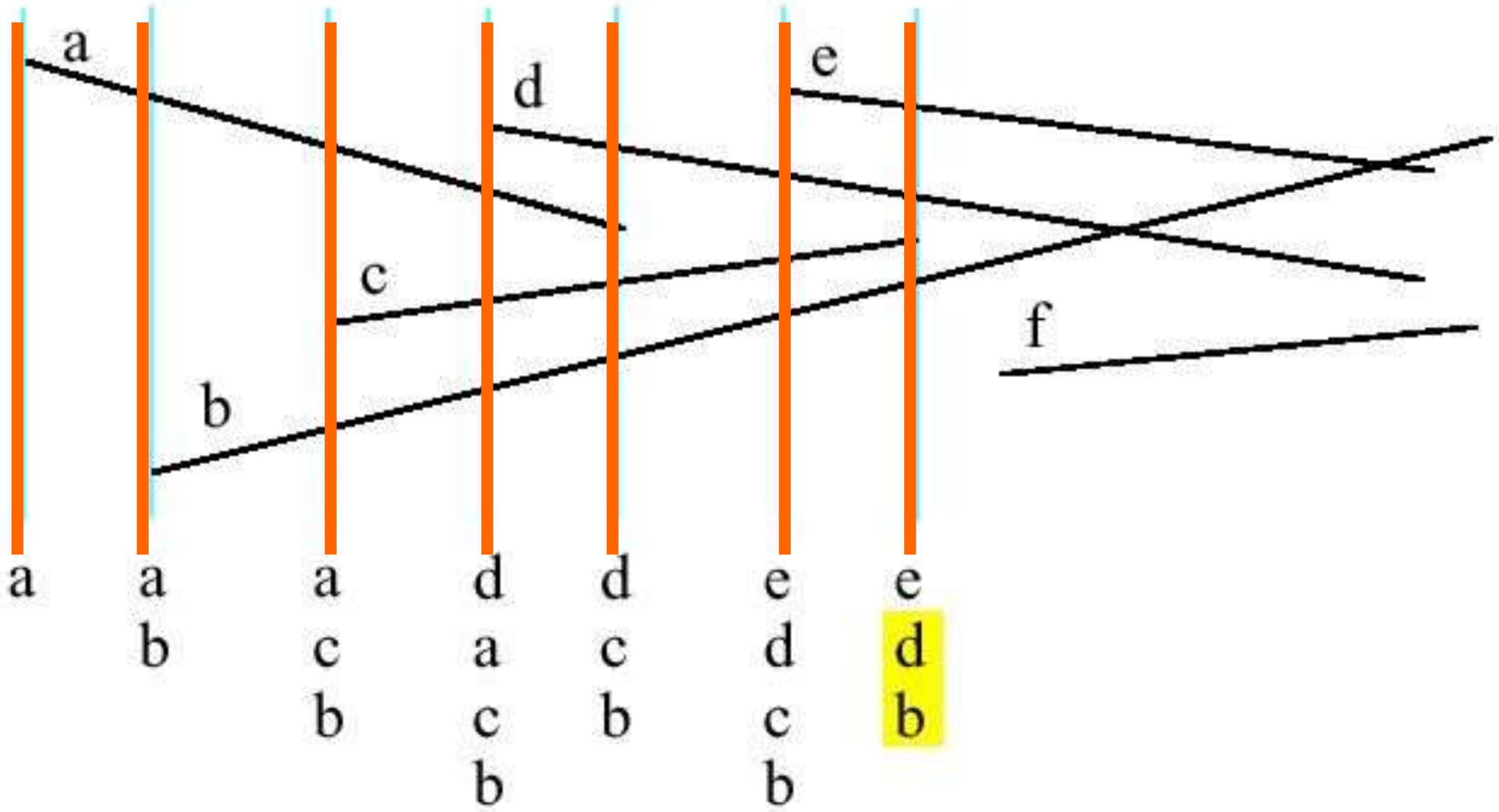
Moving the Sweep Line

- The sweep-line status is a total order T. for which we require the following operations:
  - 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.
- If there are n segments in the input, we can perform each of the above operations in O(log n) time using red-black trees.
- Replace the key comparisons by cross-product comparisons that determine the relative ordering of two segments.

Segment-intersection Pseudocode

```
ANY-SEGMENTS-INTERSECT(S)
T ⇐ ∅
sort the endpoints of the segments in S from left to right,
breaking ties by putting points with lower y-coordinates first
for each point p in the sorted list of endpoints
     do if p is the left endpoint of a segment s
            then INSERT(T, s)
                 if (ABOVE(T, s) exists and intersects s)
                    or (BELOW(T, s) exists and intersects s)
                 then return TRUE
          if p is the right endpoint of a segment s
            then if both ABOVE(T, s) and BELOW(T, s) exist
                    and ABOVE(T, s) intersects BELOW(T, s)
                 then return TRUE
                 DELETE(T, s)
return FALSE
```

- The above algorithm takes as input a set S of n line segments, returning TRUE if any pair in S intersects, and FALSE otherwise. The total order T is implemented by a red-black tree.



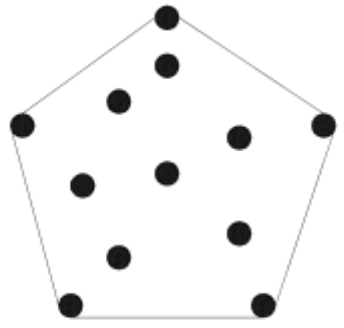
-- The intersection of segments d and b is found when segment c is deleted.
-- Running time is O(n log n).

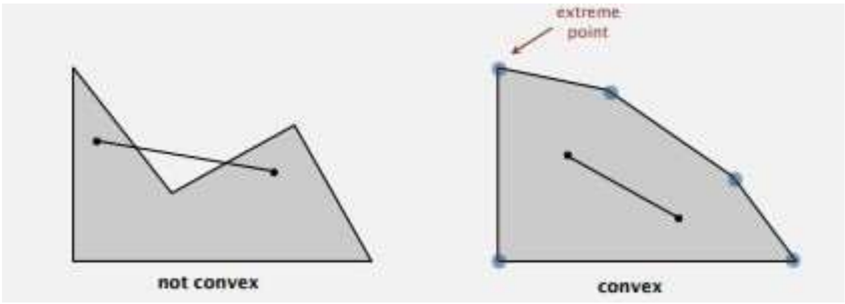
Convex Hulls

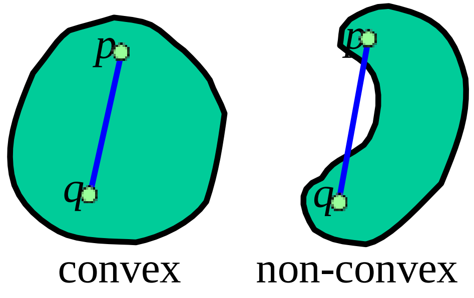
- Definitions
  - A subset P of the plane is convex iff for every p,q ∈P line segment pq is completely contained in P.
  - The Convex Hull of a set Q of points is the smallest convex polygon P, for which each point in Q is either on the boundary of P or in its interior.

Intuition:

If there is a plane Q , consisting of nails sticking out from a board. Then the Convex Hull of Q can be thought of the shape formed by a tight rubber band that surrounds all the nails.

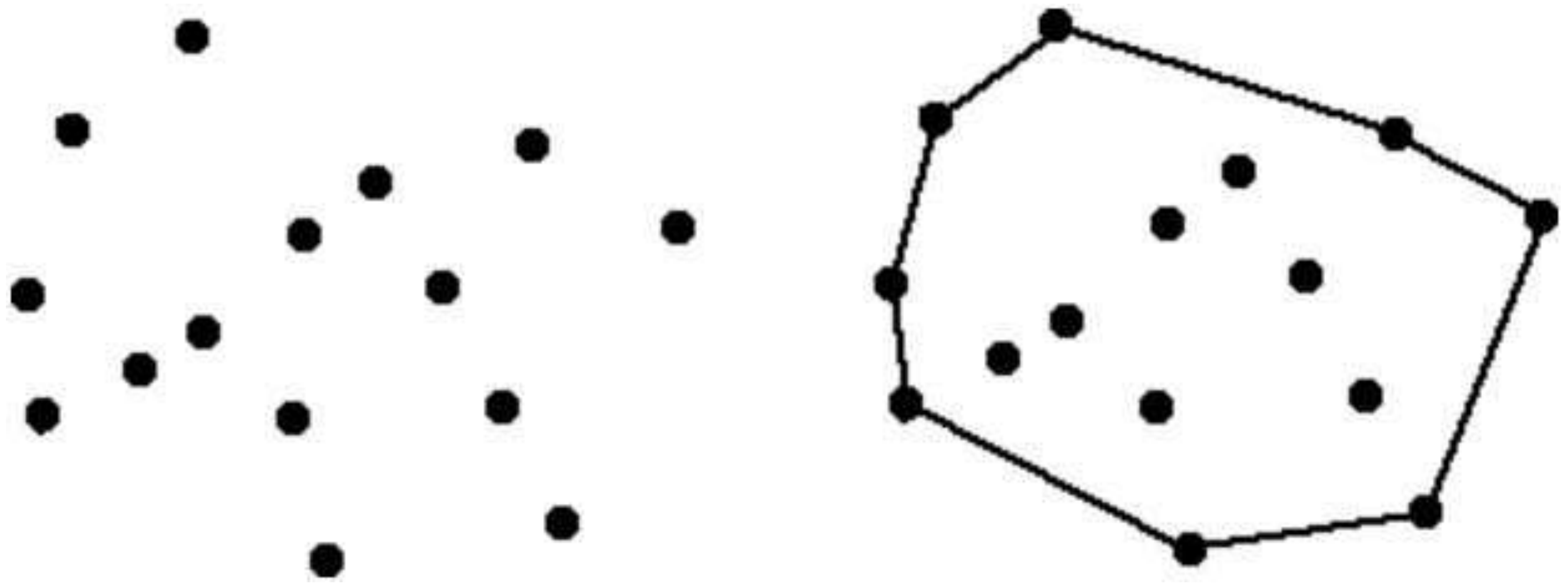




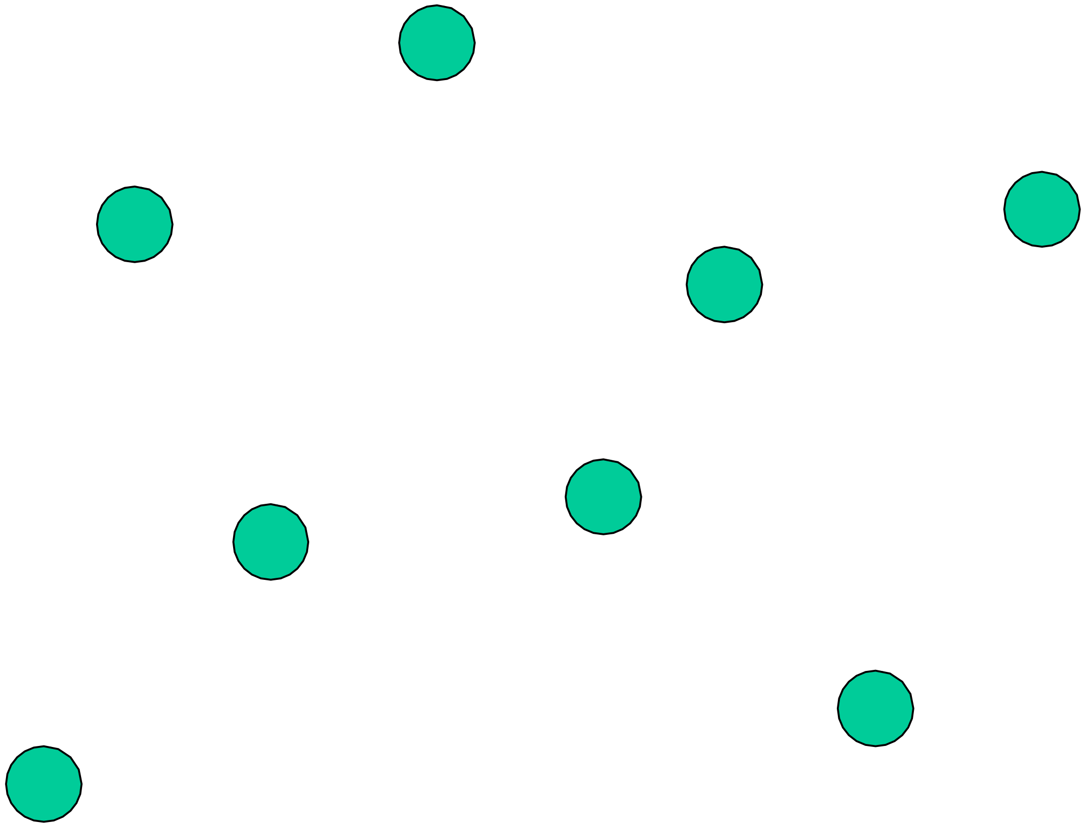


Finding the Convex Hull

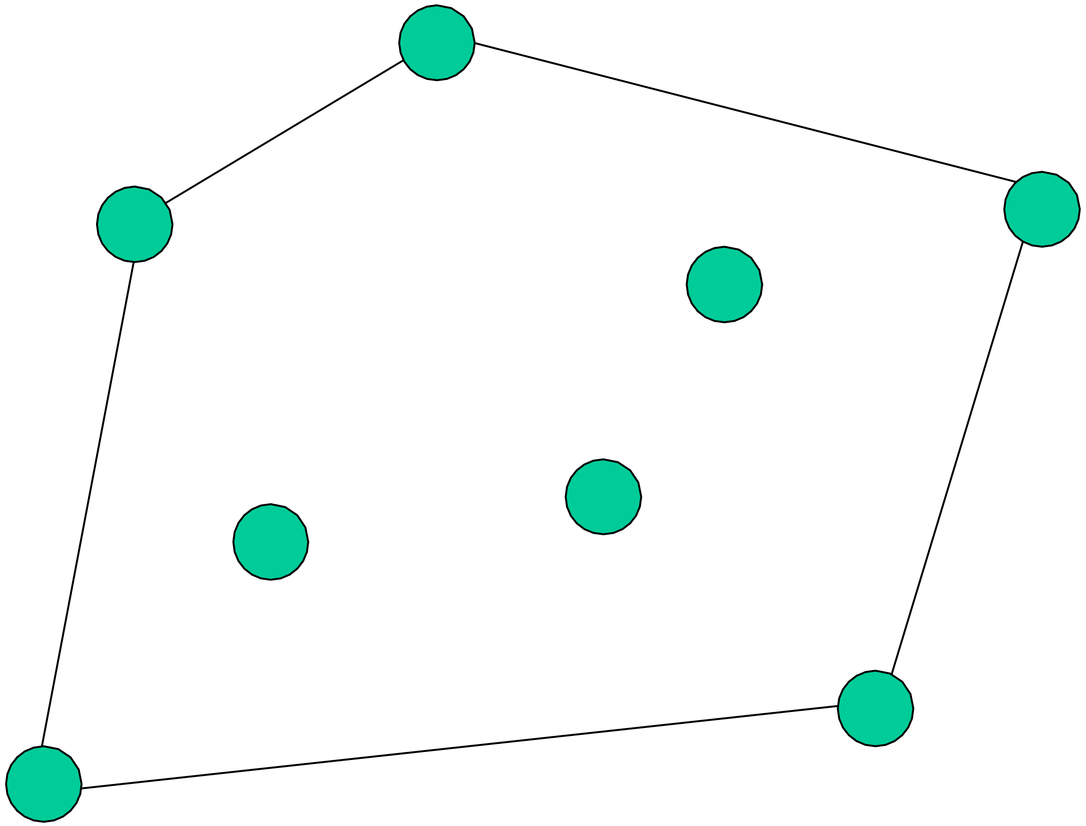
- A convex hull of n given points is defined as the smallest convex polygon containing them all



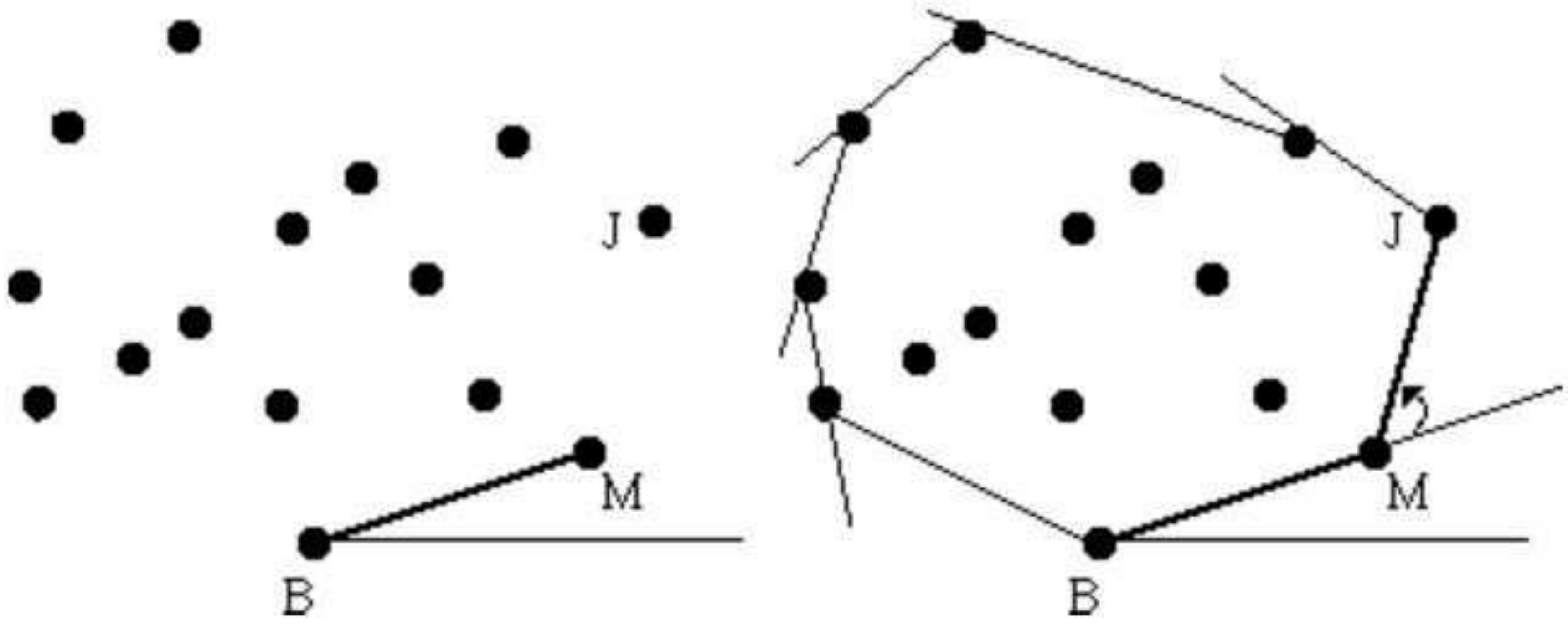
Convex Hull



Convex Hull



Idea



Method 1: The Graham Scan

```
GRAHAM-SCAN(Q)
let p0 be the point in Q with the minimum y-coordinate,
        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
        (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)
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
            POP(S)
        PUSH(pi, S)
return S
```

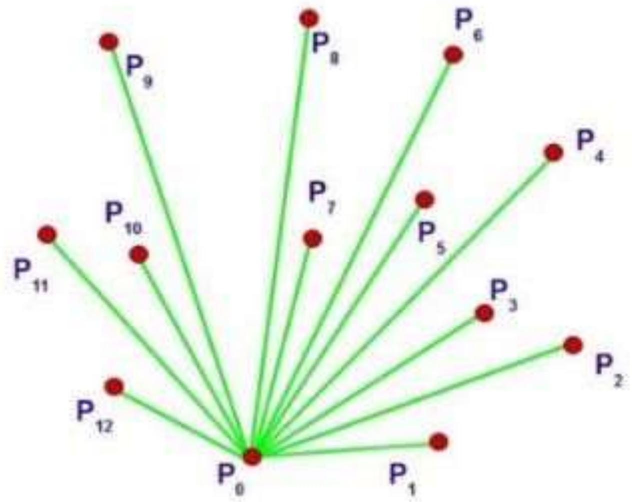
Method 1: The Graham Scan

- Graham Scan algorithm starts by taking the point with the lowest y-coordinate (picking leftmost in case of a tie)
- From this point calculate the angles to all other points

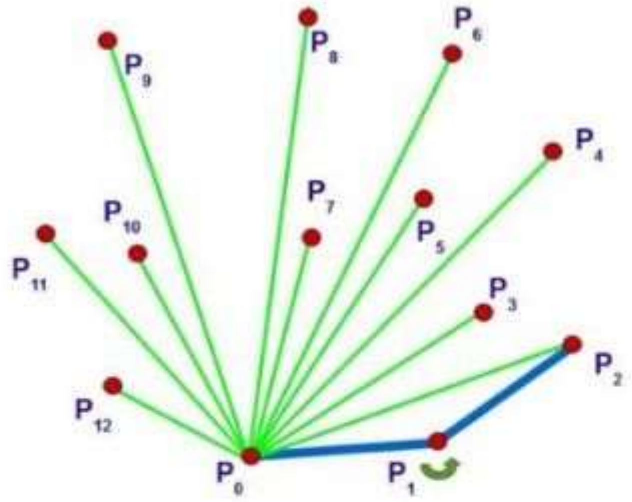
- Sort all the angles

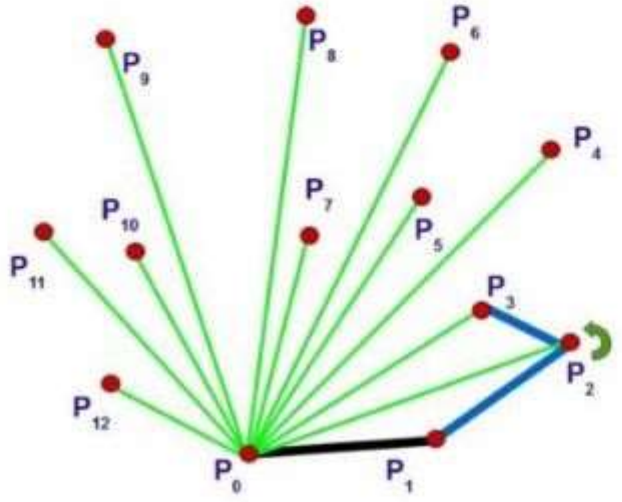
- Start plotting to the next points

- Whenever it takes right turns it backtracks and re-joins those points that makes the shortest path.

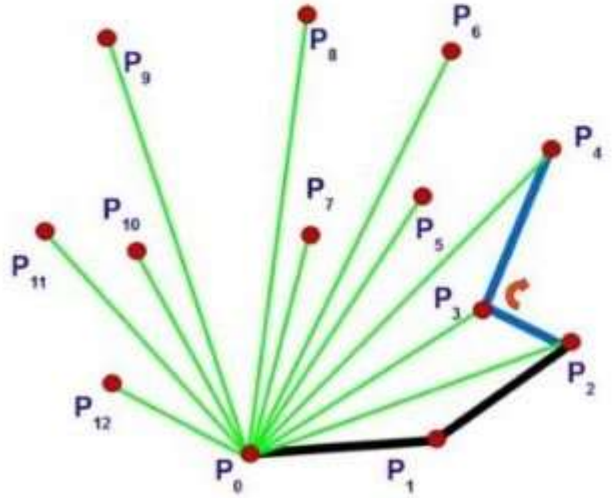


P9
P8
P6
P4
P7
P10
P5
P11
P3
P2
P12
P0
P1

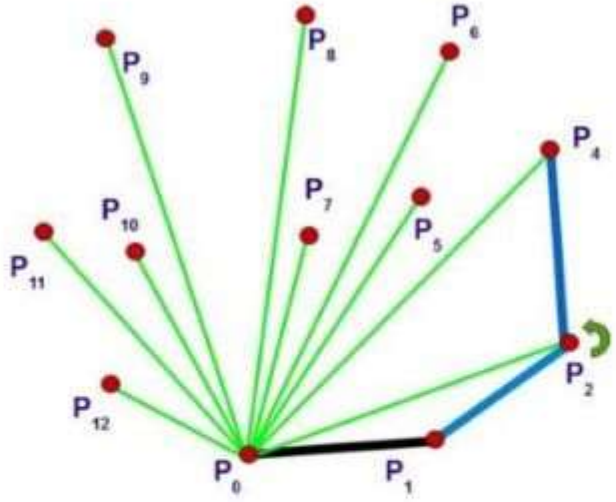

P9
P8
P6
P4
P7
P10
P5
P11
P3
P2
P12
P0
P1

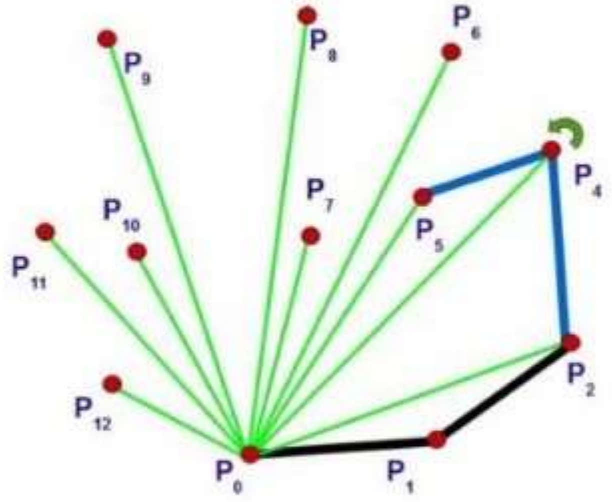


P_9
P_8
P_6
P_4
P_7
P_{10}
P_5
P_{11}
P_3
P_2
P_{12}
P_0
P_1

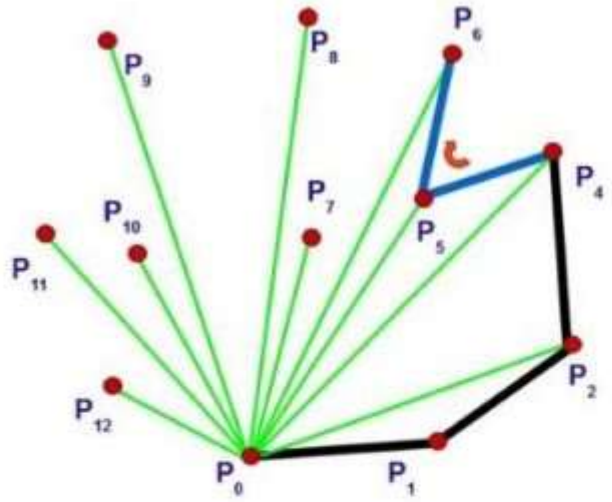


P_9
P_8
P_6
P_4
P_7
P_{10}
P_5
P_{11}
P_3
P_2
P_{12}
P_0
P_1

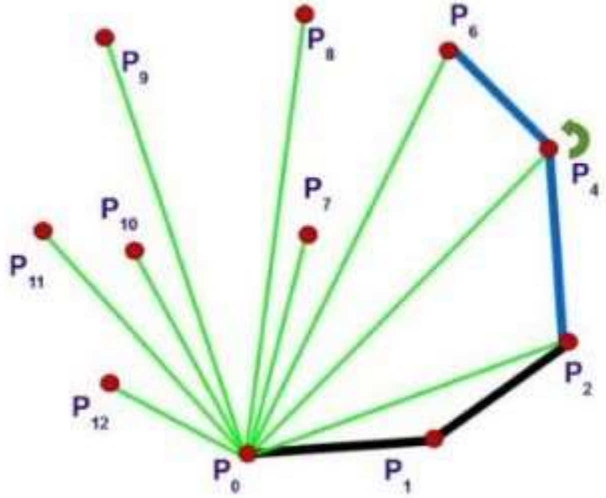

P_0
P_1
P_2
P_4
P_5
P_6
P_7
P_8
P_9
P_{10}
P_{11}
P_{12}



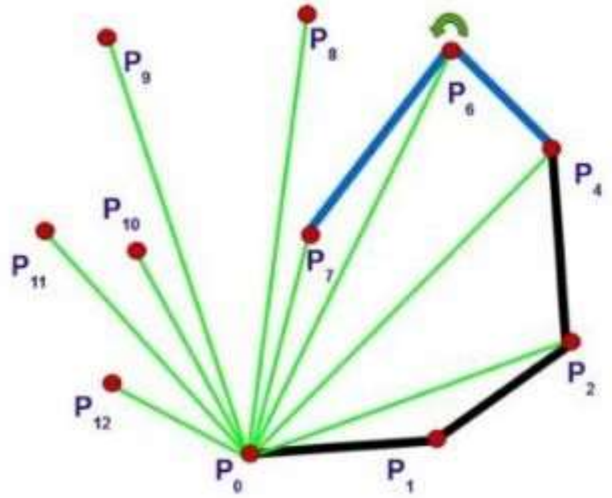
P6
P8
P9
P4
P7
P10
P5
P11
P2
P12
P0
P1

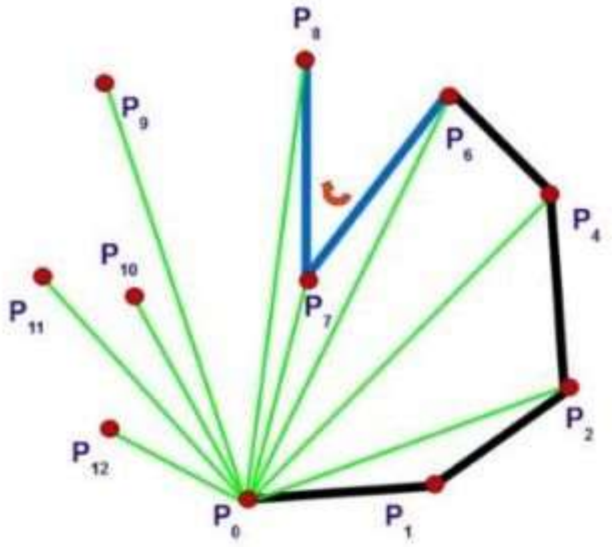


P6
P8
P9
P4
P7
P10
P5
P11
P2
P12
P0
P1

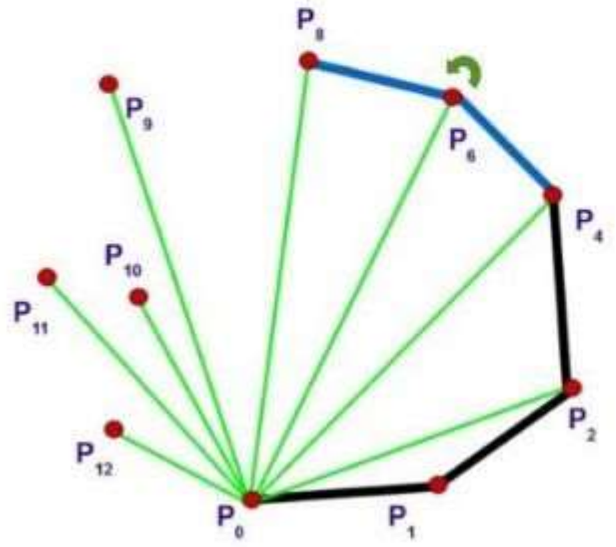


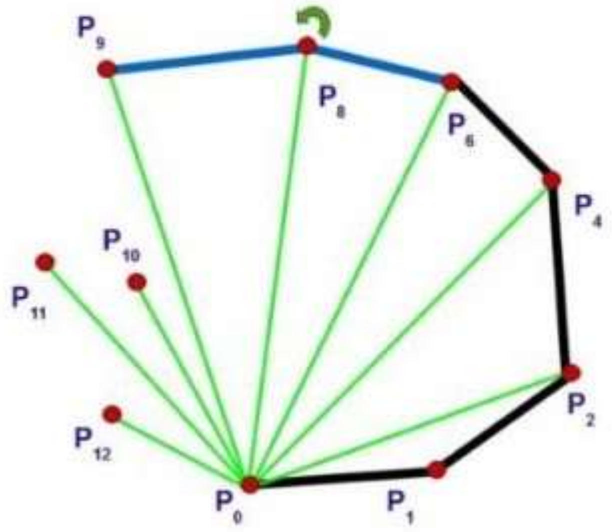
P6
P8
P9
P4
P7
P10
P11
P2
P12
P0
P1


P9
P8
P6
P4
P10
P11
P7
P2
P12
P0
P1

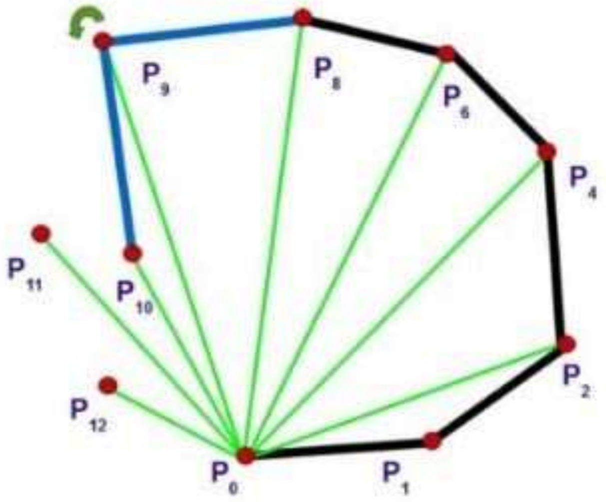


P_8
P_9
P_6
P_4
P_{10}
P_{11}
P_7
P_2
P_{12}
P_0
P_1

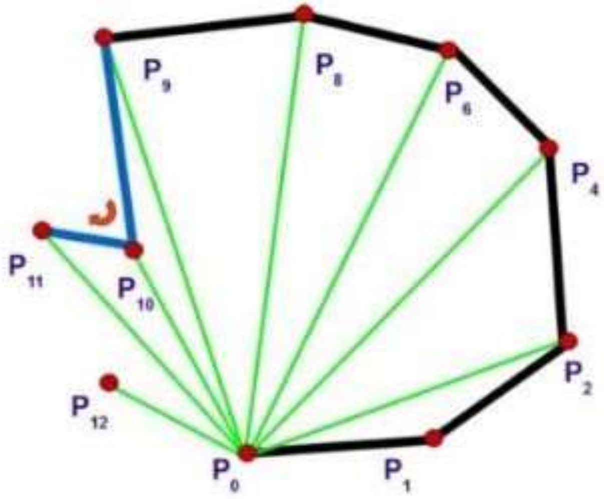

P_8
P_9
P_6
P_4
P_{10}
P_{11}
P_2
P_{12}
P_0
P_1



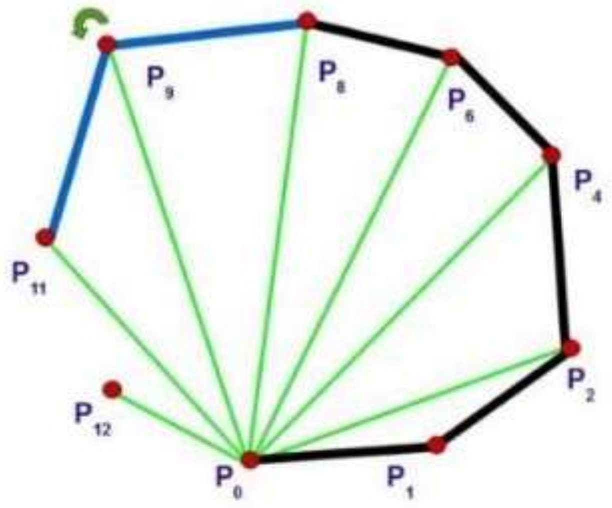
P9
P8
P6
P4
P10
P11
P2
P12
P0
P1



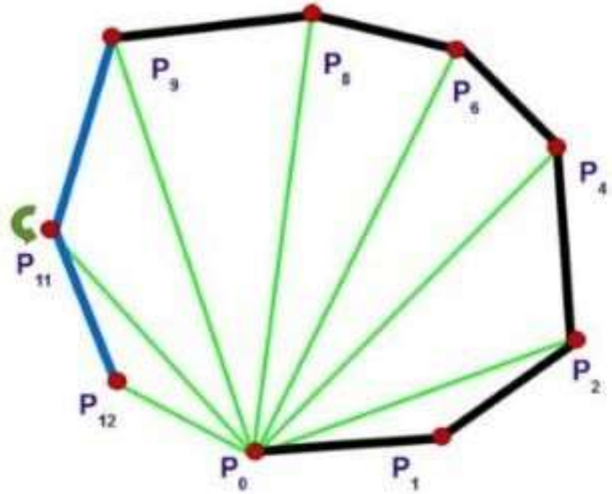
P9
P8
P6
P4
P11
P10
P2
P12
P0
P1



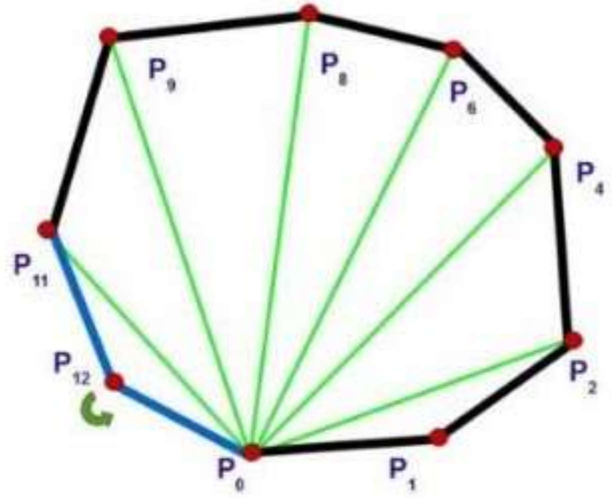
P_9
P_8
P_6
P_4
P_{11}
P_{10}
P_2
P_{12}
P_0
P_1

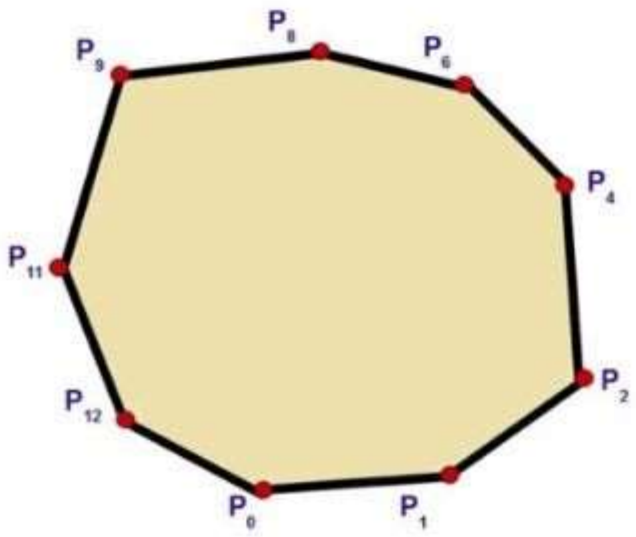


P9
P8
P6
P4
P11
P2
P12
P0
P1



P9
P8
P6
P4
P11
P2
P12
P0
P1


P9
P8
P6
P4
P11
P12
P2
P0
P1


P9
P8
P6
P4
P11
P2
P12
P0
P1

GRAHAM-SCAN(Q)

```
let p0 be the point in Q with the minimum y-coordinate,
        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
        (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)
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
            POP(S)
        PUSH(pi, S)
return S
```

Method 2: *Jarvis's march* (Package Wrapping)

Jarvis March computes the Convex Hull of a set Q of points by a technique called package wrapping or gift wrapping.

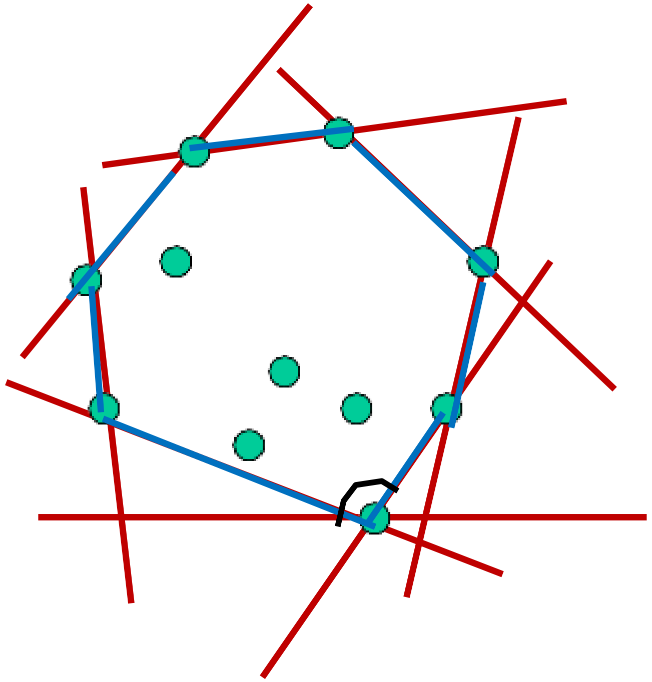
Intuitively Jarvis March simulates a taut piece of paper around the set Q. To get the turning we take an "anchor" point then make a line with every other point and select the one with the least angle and keep on repeating.

The algorithm runs in time O(nh) where h is the number of vertices

Method 2: *Jarvis's march* (Package Wrapping)

- Pick a point on convex hull.
- Loop through all points and find the one that forms the minimum sized anticlockwise angle off the horizontal axis from the previous point.
- Continue until you encounter the first point.

Method 2: *Jarvis's march* (Package Wrapping)



Jarvis's March

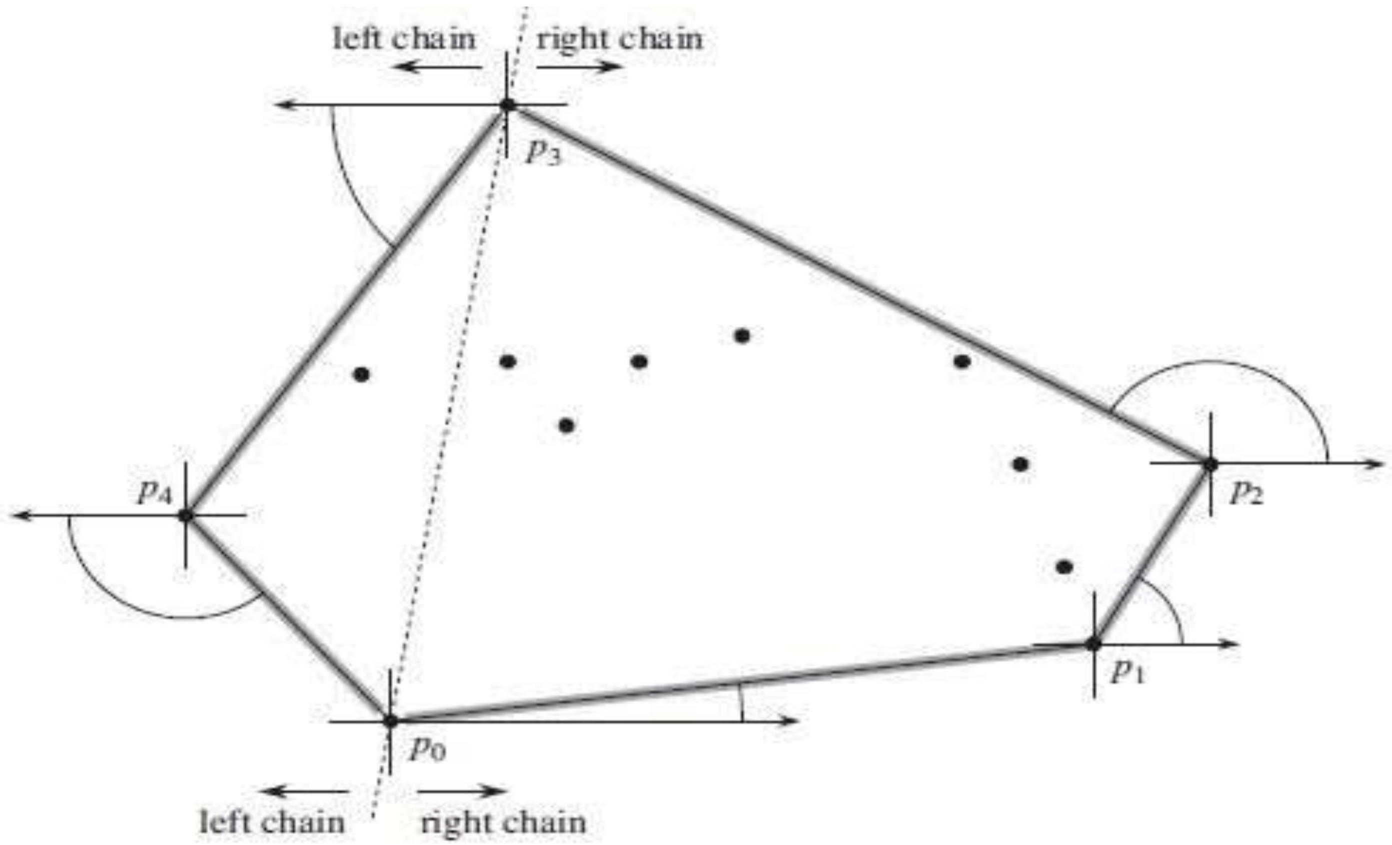


Figure 33.9 The operation of Jarvis's march. We choose the first vertex as the lowest point p_0. The next vertex, p_1, has the smallest polar angle of any point with respect to p_0. Then, p_2 has the smallest polar angle with respect to p_1. The right chain goes as high as the highest point p_3. Then,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

The Closest Pair Problem

The problem: Let Q ={p_1,..., p_n} be a set of n points in d-dimensional space, determine the closest pair of points in S.

-- The distance between two points (x_1,..., x_d) and (y_1,..., y_d) is defined as $\sqrt[q]{\sum_{i=1}^{d} \text{abs}(x_i - y_i)^q}$, where q ≥ 2 is a given integer.

-- Sequential algorithms

- Best known lower bound: $\Omega(n)$.
- Best known algorithm: $O(n^2)$. brute-force testing
- For d = 2 (and q = 2): $O(n \cdot \log n)$ divide and conquer

- The divide-and-conquer algorithm: Closest-Pair(P, X, Y)

  -- Each recursive invocation of the algorithm takes as input a subset $P \subseteq Q$ and arrays X and Y.

  -- Each of which contains all the points of the input subset P. The points in array X are sorted so that their x-coordinates are monotonically increasing. Similarly, array Y is sorted by monotonically increasing y-coordinate.

  -- A given recursive invocation with inputs P, X, and Y first checks whether $|P| \leq 3$. If so, the invocation simply performs the brute-force method and return the closest pair. If $|P| > 3$, the recursive invocation carries out the divideand-conquer paradigm as follows.

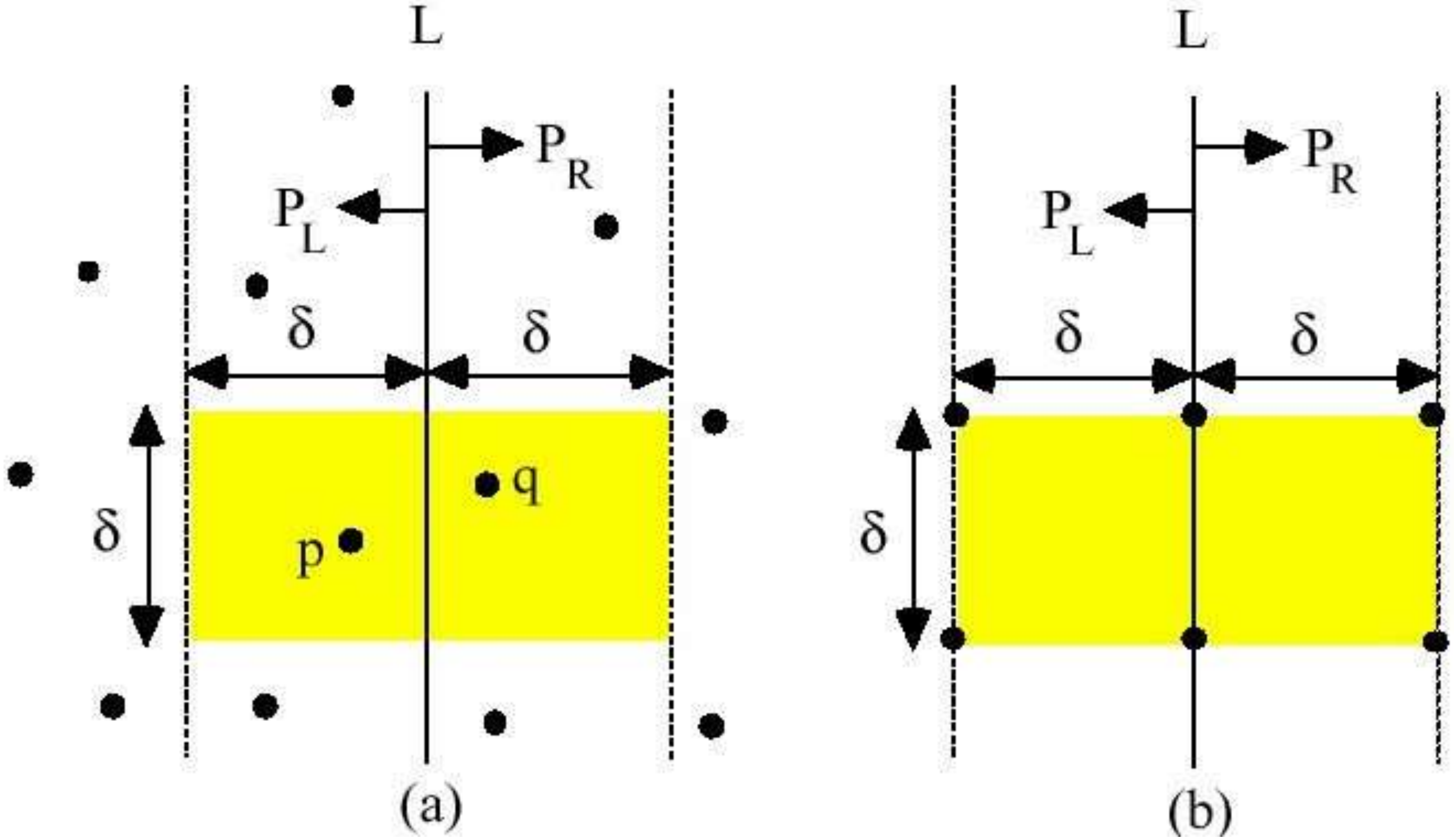
- Divide:
  - -- Find a vertical line L that bisects the point set P into two sets P_L and P_R such that $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, all points in P_L are on or to the left of line L, and all points in P_R are on or to the right of L.
  - -- Divide X into arrays X_L and X_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing x-coordinate. Similarly, divide Y into arrays Y_L and Y_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing y-coordinate.
- Conquer:
  - -- δ_L = Closest-Pair(P_L, X_L, Y_L)
  - -- δ_R = Closest-Pair(P_R, X_R, Y_R)
  - -- $\delta = \min(\delta_L, \delta_R)$

Combine:

- -- The closest pair is either the pair with distance δ found by one of the recursive calls, or it is a pair of points with one point in P_L and the other in P_R.
- -- Observe that if there is a pair of points with distance less than δ, 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.

1. 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 Y', computes the distance from p to the 7 points in Y' that follow p and keeps track of the closest-pair distance δ' found over all pairs of points in Y'.
3. If $\delta' < \delta$, then the vertical strip does indeed contain a closer pair than was found by the recursive calls. This pair and its distance δ' are returned. Otherwise, the closest pair and its distance δ found by the recursive calls are returned.

Correctness



Correctness

(a) If $p \in P_L$ and $q \in P_R$ 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. There can be 8 points in the $\delta \times 2\delta$ rectangle if the points shown on line L are actually pairs of coincident points with one point in P_L and one in P_R.

Implementation and Running Time

- Our goal is to have the recurrence for the running time be
  $T(n) = 2T(n/2) + O(n)$.

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

  -- Note that if the array X that is received by a recursive call is already sorted, then the division of set P into P_L and P_R is easily accomplished in linear time.

- Key observation: In each call, we wish to form a sorted subset of a sorted array. For example, a particular invocation is given the subset P and the array Y. sorted by y-coordinate. Having partitioned P into P_L and P_R, it needs to form the arrays Y_L and Y_R, which are sorted by y-coordinate. Moreover, these arrays must be formed in linear time. The method can be viewed as the opposite of the merge procedure: we are splitting a sorted array into two sorted arrays.

```
length[Y_L] = length[Y_R] = 0
for i = 1 to length[Y]
    do if Y[i] ∈ P_L
        then length[Y_L] = length[Y_L] + 1
             Y_L[length[Y_L]] = Y[i]
        else length[Y_R] = length[Y_R] + 1
             Y_R[length[Y_R]] = Y[i]
```

-- Similar pseudocode works for forming arrays X_L and X_R.

- To get the points sorted in the first place, simply presorting them.
- $T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 3 \\ O(1) & \text{if } n \leq 3 \end{cases}$

  $T'(n) = O(n \log n) + T(n) = O(n \log n)$

- marjorie-lazos -ppt
- http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap35.htm
- https://nptel.ac.in/courses/106/102/106102011/
- https://www.youtube.com/watch?v=1z8hzaOUL_w
- https://www.youtube.com/watch?v=R08OY6yDNy0
- https://www.slideserve.com/kaseem-burgess/computational-geometry-powerpoint-ppt-presentation
- https://www.youtube.com/watch?v=B2AJoQSZf4M
- https://www.youtube.com/watch?v=_j1Qd9suN0s
- https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/