

# **Geometry Introduction**

# Topic

- Introduction
- Two lines Intersection Test
- Point inside polygon
- Convex hull
- Line Segments Intersection Algorithm

# Geometry

## Components

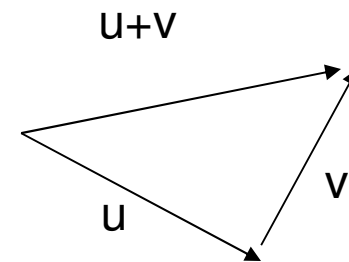
- Scalar (S)
- Point (P)
- Free vector (V)

## Allowed operations

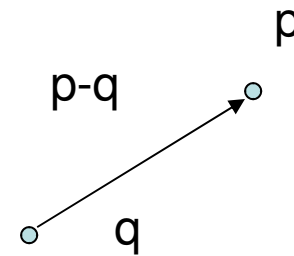
- $S * V \rightarrow V$
- $V + V \rightarrow V$
- $P - P \rightarrow V$
- $P + V \rightarrow P$

# Examples

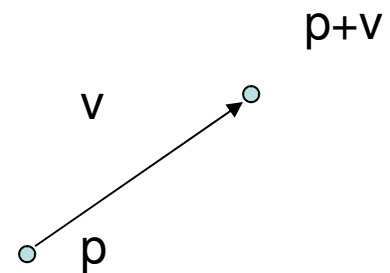
Vector addition

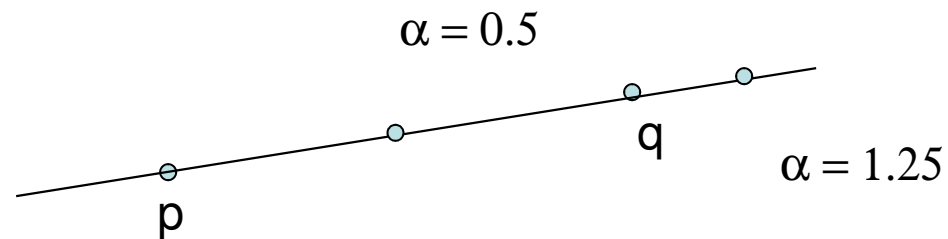


Point subtraction



Point-vector addition





จุดใด ๆ บนเส้นตรงที่ลากผ่านจุด **p** และ **q** สามารถสร้างได้จาก  
affine combination:

$$\mathbf{r} = \mathbf{p} + \alpha \cdot (\mathbf{q} - \mathbf{p})$$

# Euclidean Geometry

- In affine geometry, angle and distance are not defined.
- Euclidean geometry is an extension providing an additional operation called “inner product”
- There are other types of geometry that extends affine geometry such as projective geometry, hyperbolic geometry...

Dot product is a mapping from two vectors to a real number.

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_d \end{pmatrix}, \mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_d \end{pmatrix}$$

Length

$$|\mathbf{u}| = \sqrt{\mathbf{u} \cdot \mathbf{u}}$$

Angle

$$\text{ang}(\mathbf{u}, \mathbf{v}) = \cos^{-1} \left( \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}| |\mathbf{v}|} \right)$$

Orthogonality:  $\mathbf{u}$  and  $\mathbf{v}$  are orthogonal when

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{0}$$

Dot product

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^d u_i v_i$$

Distance

$$\text{dist}(\mathbf{P}, \mathbf{Q}) = |\mathbf{P} - \mathbf{Q}|$$

# Topic

- Introduction
- Two lines Intersection Test
- Point inside polygon
- Convex hull
- Line Segments Intersection Algorithm

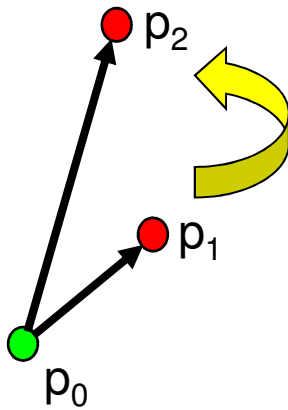


# Cross-Product-Based Geometric Primitives

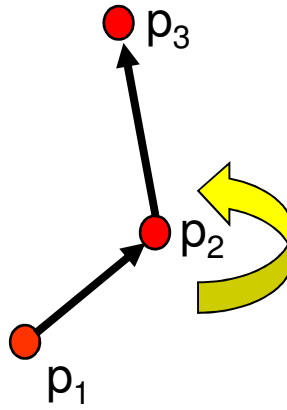
Some fundamental geometric questions:

1. Given two directed segments  $\overrightarrow{p_0p_1}$  and  $\overrightarrow{p_0p_2}$ , is  $\overrightarrow{p_0p_1}$  clockwise from  $\overrightarrow{p_0p_2}$  with respect to their common endpoint  $p_0$ ?
2. Given two line segments  $\overline{p_1p_2}$  and  $\overline{p_2p_3}$ , if we traverse  $\overline{p_1p_2}$  and then  $\overline{p_2p_3}$ , do we make a left turn at point  $p_2$ ?
3. Do line segments  $\overline{p_1p_2}$  and  $\overline{p_3p_4}$  intersect?

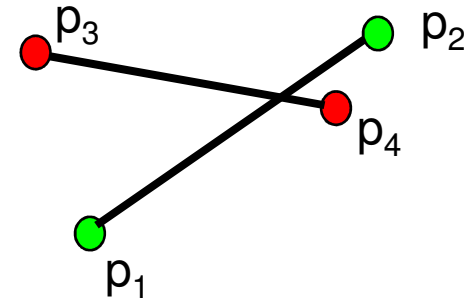
source: 91.503 textbook Cormen et al.



(1)

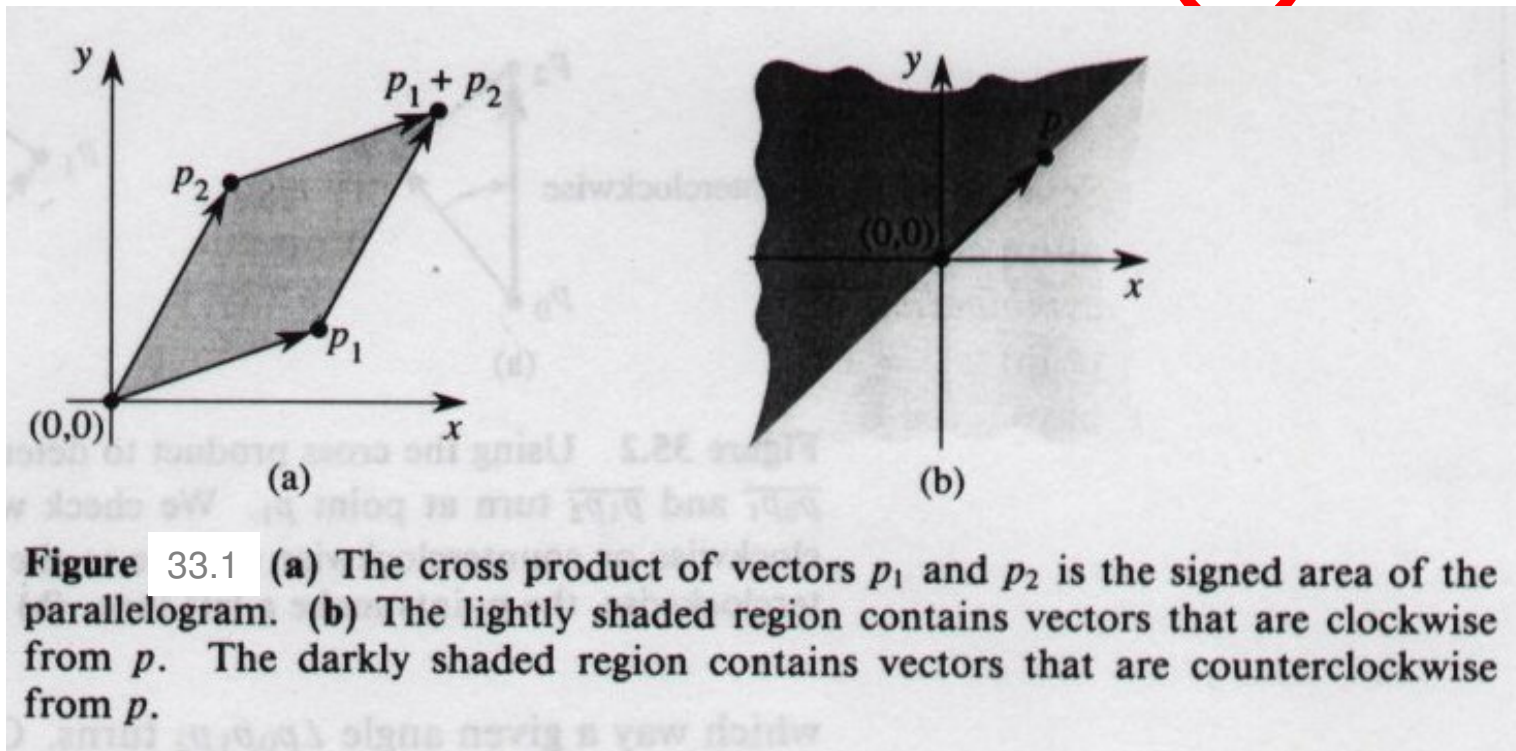


(2)

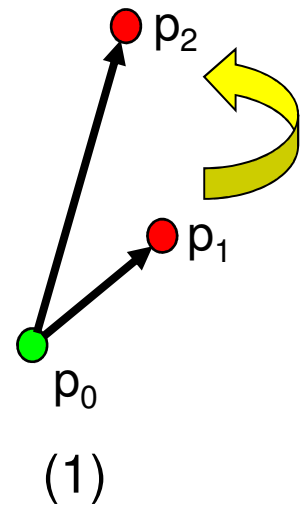


(3)

# Cross-Product-Based Geometric Primitives: (1)



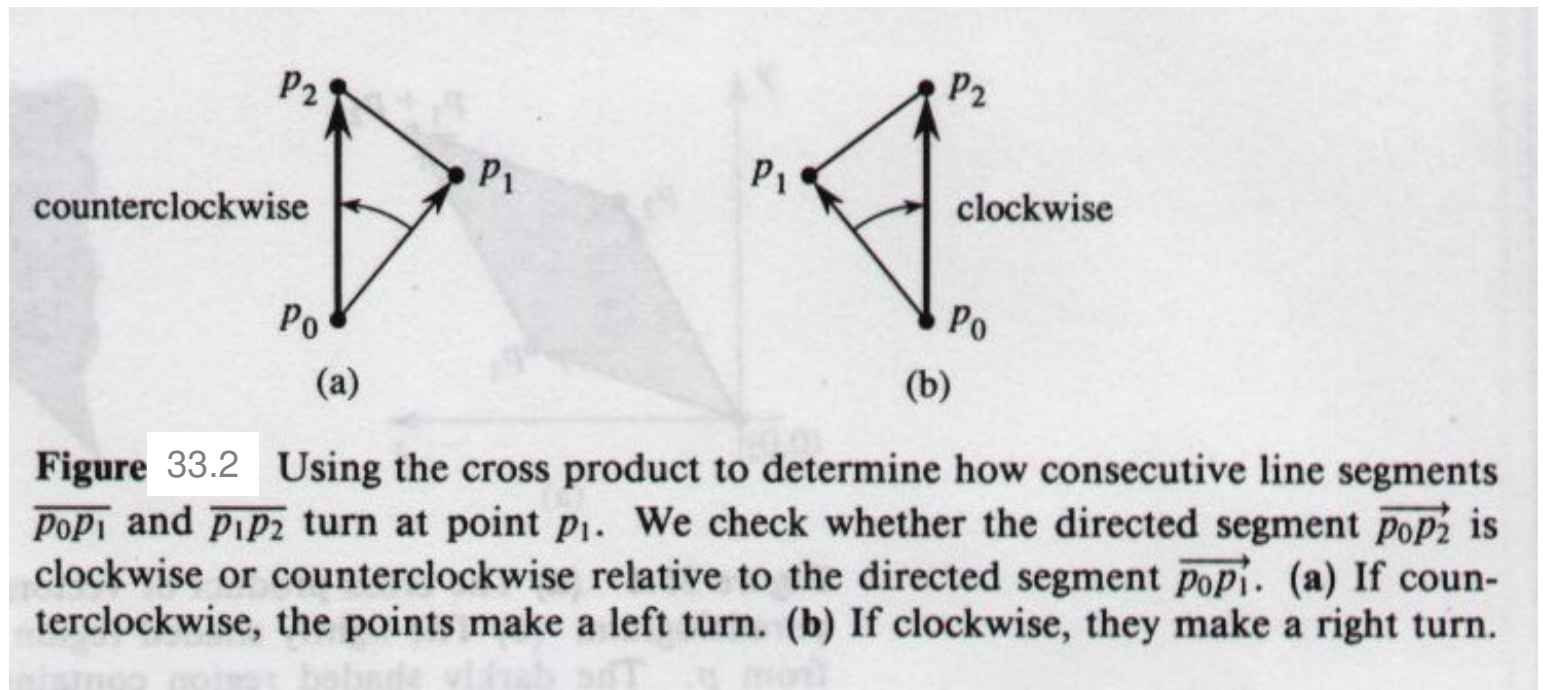
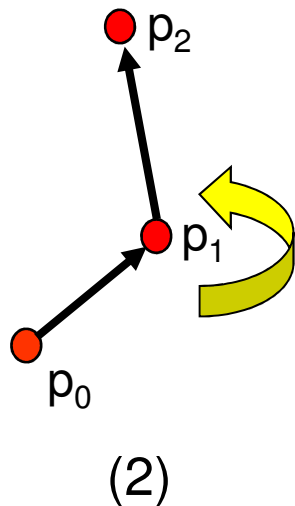
$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1$$



$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

Advantage: less sensitive to accumulated round-off error *source: 91.503 textbook Cormen et al.*

# Cross-Product-Based Geometric Primitives: (2)

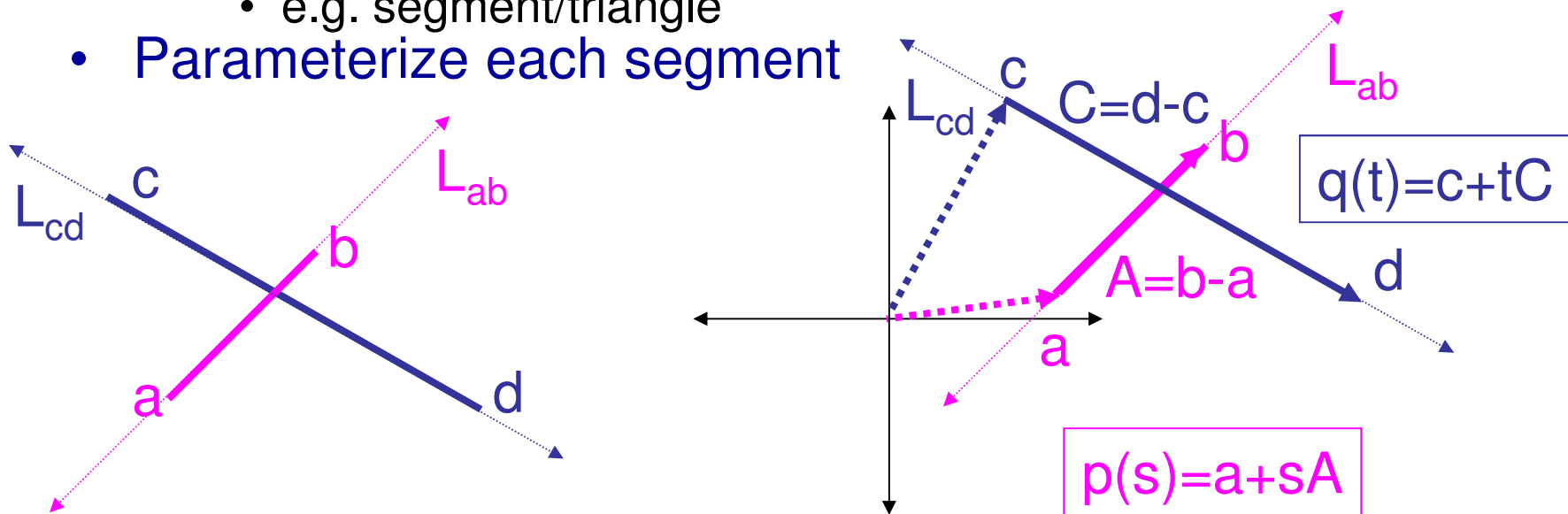


# isLeft()

```
// isLeft(): tests if a point is Left|On|Right of an infinite line.
//      Input:  three points P0, P1, and P2
//      Return: >0 for P2 left of the line through P0 and P1
//              =0 for P2 on the line
//              <0 for P2 right of the line
int isLeft( Point P0, Point P1, Point P2 )
{
    return ( (P1.x - P0.x) * (P2.y - P0.y)
             - (P2.x - P0.x) * (P1.y - P0.y) );
}
```

# Segment-Segment Intersection

- Finding the actual intersection point
- Approach: parametric vs. slope/intercept
  - parametric generalizes to more complex intersections
    - e.g. segment/triangle
- Parameterize each segment



Intersection: values of  $s$ ,  $t$  such that  $p(s) = q(t)$  :  $a + sA = c + tC$

2 equations in unknowns  $s$ ,  $t$  : 1 for  $x$ , 1 for  $y$

source: O'Rourke, Computational Geometry in C

Assume that  $a = (x_1, y_1)$   $b = (x_2, y_2)$   $c = (x_3, y_3)$   $d = (x_4, y_4)$

$$s = \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)}$$

$$t = \frac{(x_2 - x_1)(y_1 - y_3) - (y_2 - y_1)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)}$$

# Code

```
typedef struct point { double x; double y;} point;
typedef struct line { point p1; point p2;} line;

int check_lines(line *line1, line *line2, point *hitp)
{
    double d    =    (line2->p2.y - line2->p1.y)*(line1->p2.x-line1->p1.x) -
                    (line2->p2.x - line2->p1.x)*(line1->p2.y-line1->p1.y);

    double ns =    (line2->p2.x - line2->p1.x)*(line1->p1.y-line2->p1.y) -
                    (line2->p2.y - line2->p1.y)*(line1->p1.x-line2->p1.x);

    double nt =    (line1->p2.x - line1->p1.x)*(line1->p1.y - line2->p1.y) -
                    (line1->p2.y - line1->p1.y)*(line1->p1.x - line2->p1.x);

    if(d == 0) return 0;

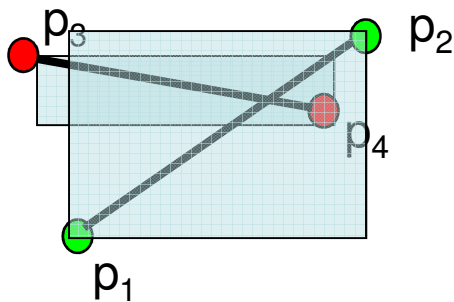
    double s= ns/d;
    double t = nt/d;

    return (s >=0 && s <= 1 && t >= 0 && t <= 1));
}
```



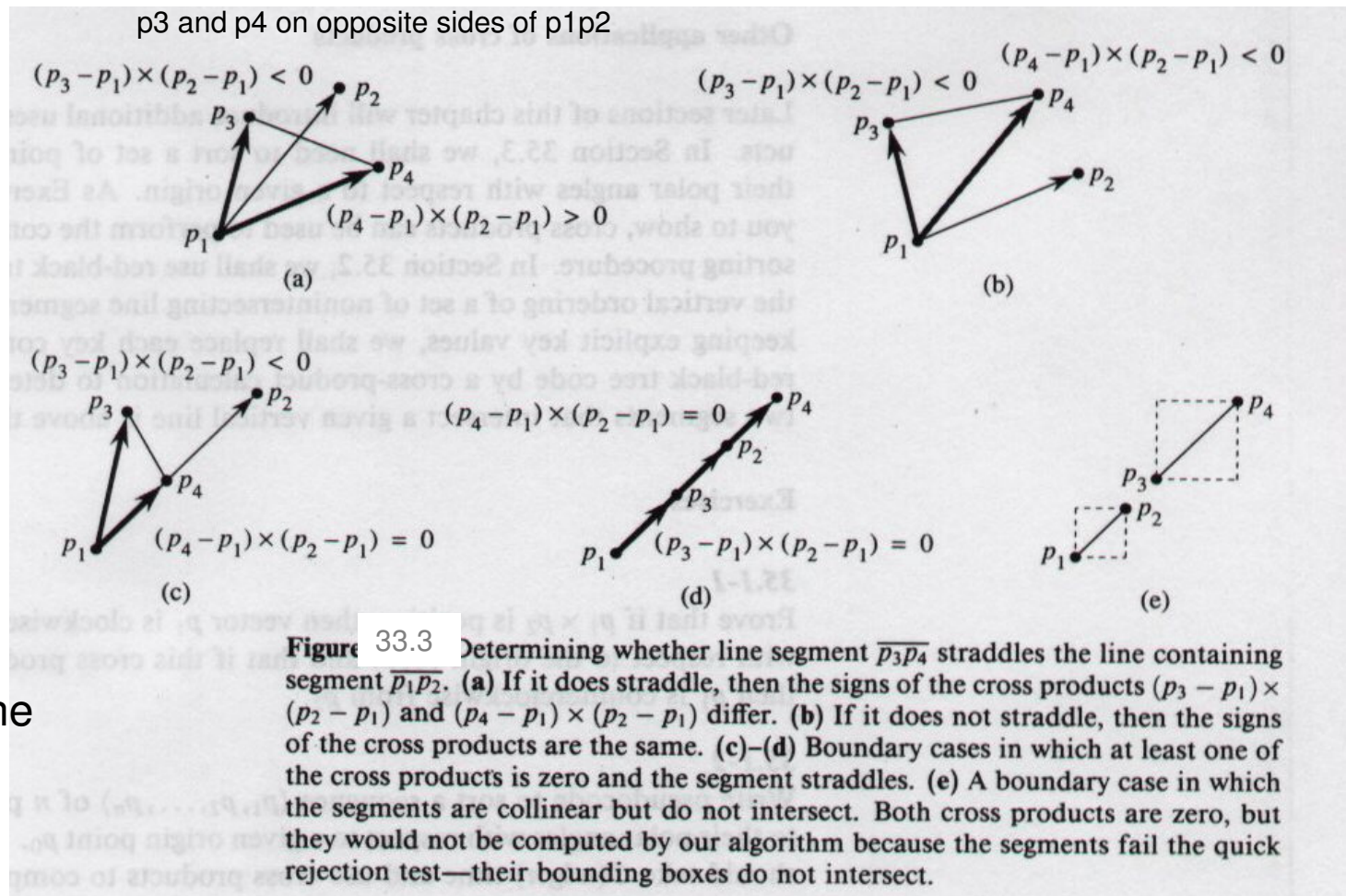
# Intersection of 2 Line Segments

Step 1:  
Bounding Box  
Test



(3)

Step 2: Does each  
segment straddle the  
line containing the  
other?



source: 91.503 textbook Cormen et al.

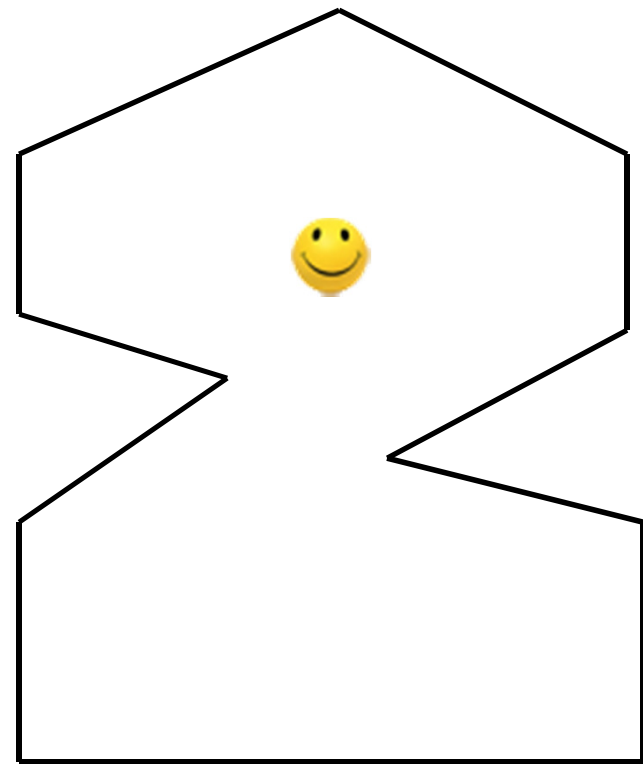


# Topic

- Introduction
- Two lines Intersection Test
- Point inside polygon
- Convex hull
- Line Segments Intersection Algorithm

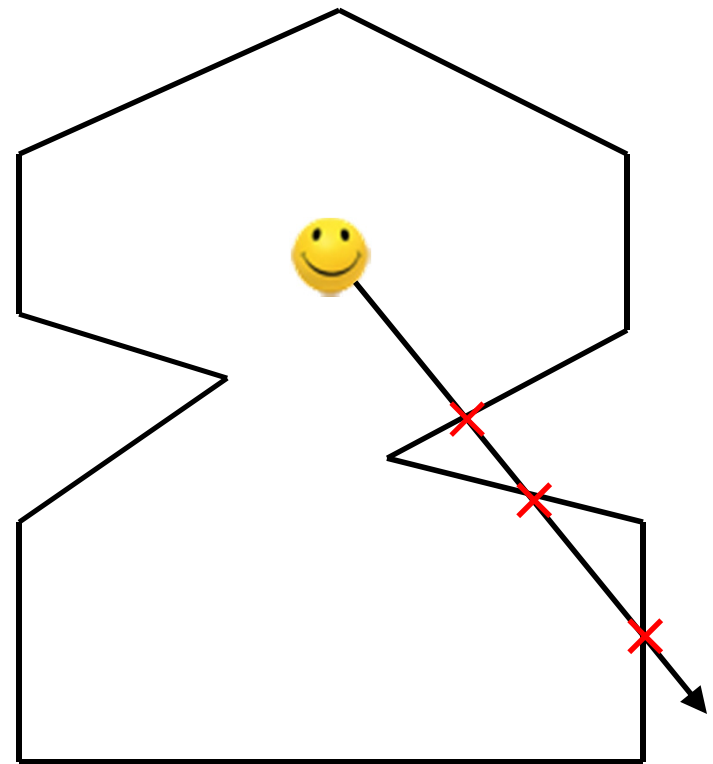
# Point Inside Polygon Test

- Given a point, determine if it lies inside a polygon or not



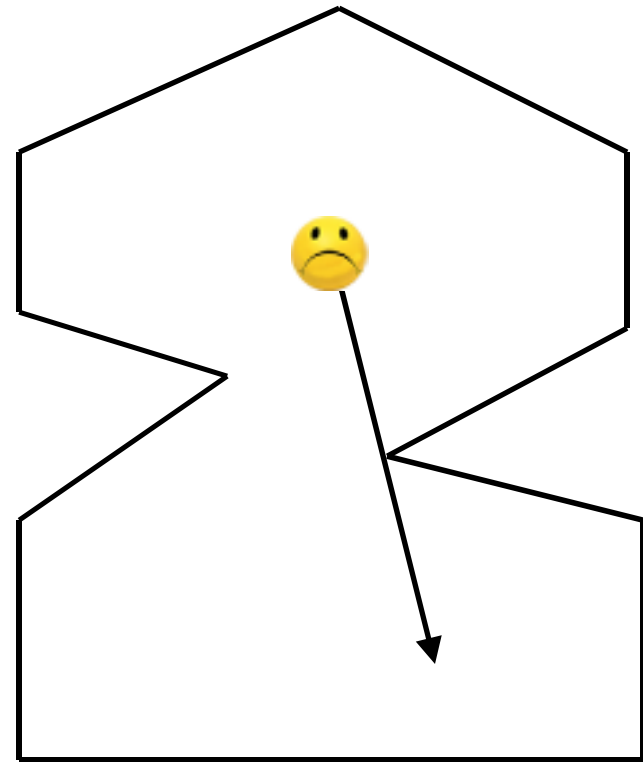
# Ray Test

- Fire ray from point
- Count intersections
  - Odd = inside polygon
  - Even = outside polygon



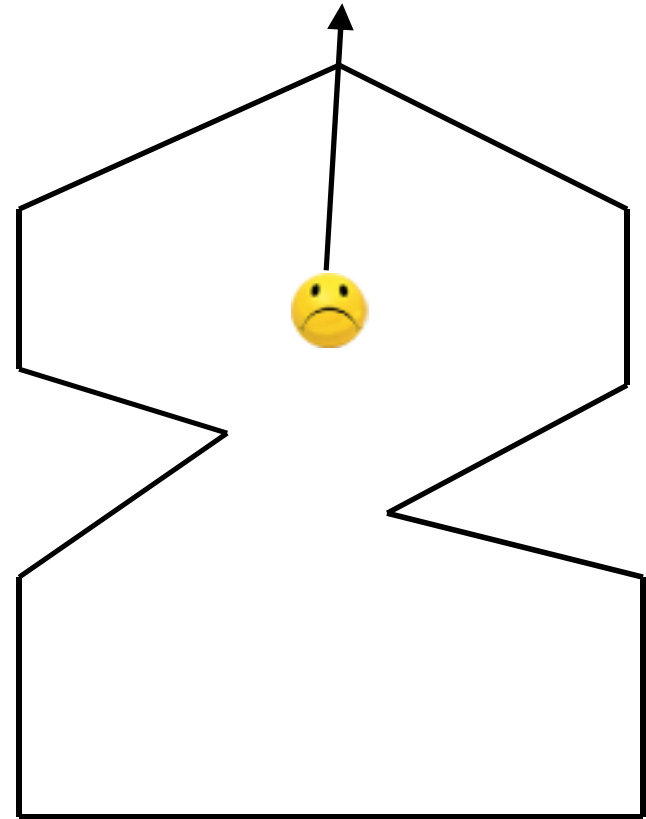
# Problems With Rays

- Fire ray from point
- Count intersections
  - Odd = inside polygon
  - Even = outside polygon
- Problems
  - Ray through vertex



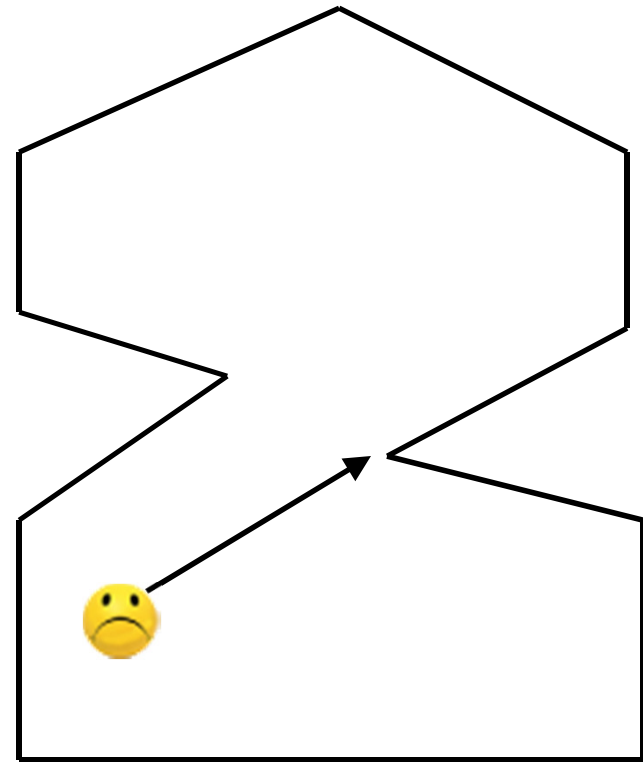
# Problems With Rays

- Fire ray from point
- Count intersections
  - Odd = inside polygon
  - Even = outside polygon
- Problems
  - Ray through vertex



# Problems With Rays

- Fire ray from point
- Count intersections
  - Odd = inside polygon
  - Even = outside polygon
- Problems
  - Ray through vertex
  - Ray parallel to edge



# Solution

- Edge Crossing Rule
  - an upward edge includes its starting endpoint, and excludes its final endpoint;
  - a downward edge excludes its starting endpoint, and includes its final endpoint;
  - horizontal edges are excluded; and
  - the edge-ray intersection point must be strictly right of the point P.
- Use horizontal ray for simplicity in computation

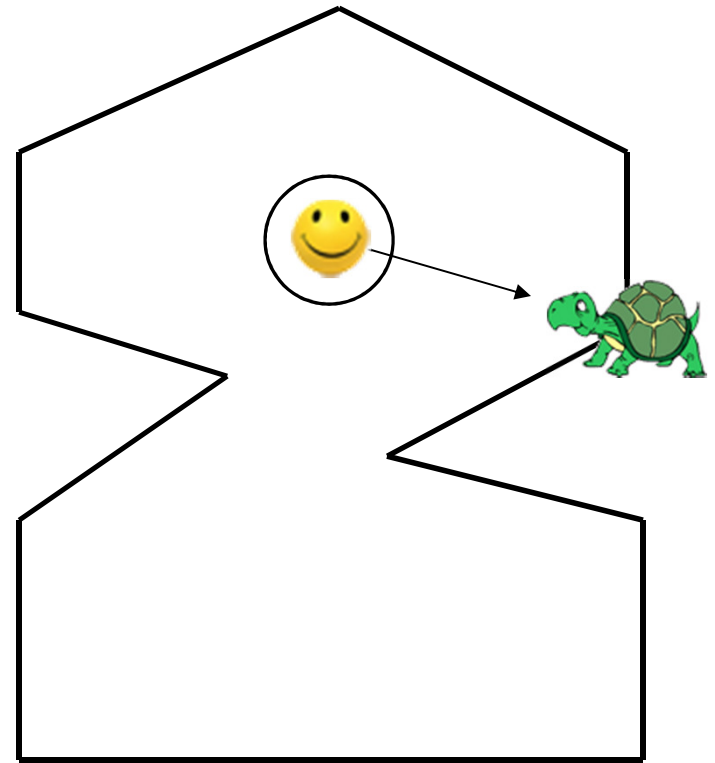
# Code

```
// cn_PnPoly(): crossing number test for a point in a polygon
//      Input:   P = a point,
//              V[] = vertex points of a polygon V[n+1] with V[n]=V[0]
//      Return:  0 = outside, 1 = inside
// This code is patterned after [Franklin, 2000]
int cn_PnPoly( Point P, Point* V, int n )
{
    int    cn = 0;    // the crossing number counter

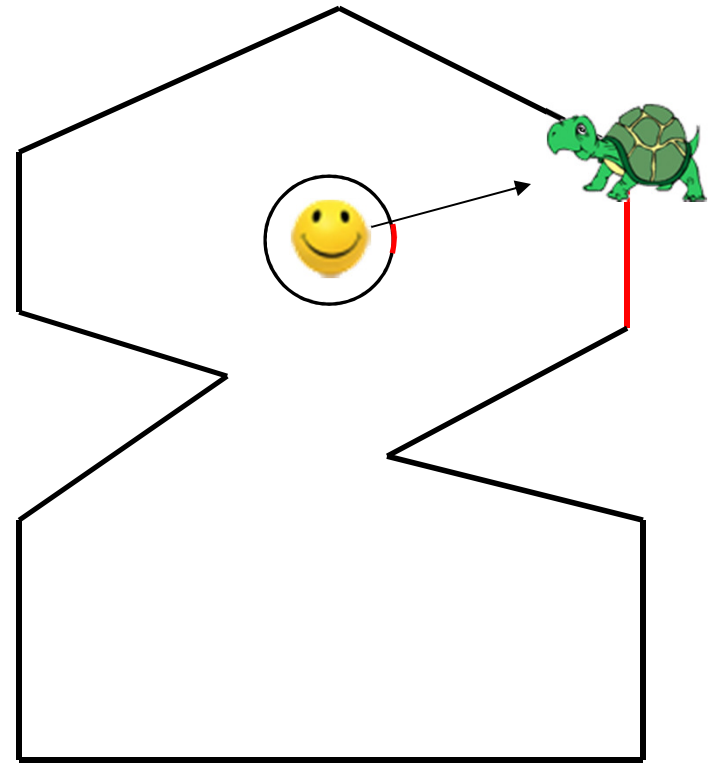
    // loop through all edges of the polygon
    for (int i=0; i<n; i++) {    // edge from V[i] to V[i+1]
        if (((V[i].y <= P.y) && (V[i+1].y > P.y))    // an upward crossing
            || ((V[i].y > P.y) && (V[i+1].y <= P.y))) { // a downward crossing
            // compute the actual edge-ray intersect x-coordinate
            float vt = (float)(P.y - V[i].y) / (V[i+1].y - V[i].y);
            if (P.x < V[i].x + vt * (V[i+1].x - V[i].x)) // P.x < intersect
                ++cn;    // a valid crossing of y=P.y right of P.x
        }
    }
    return (cn&1);    // 0 if even (out), and 1 if odd (in)
}
```



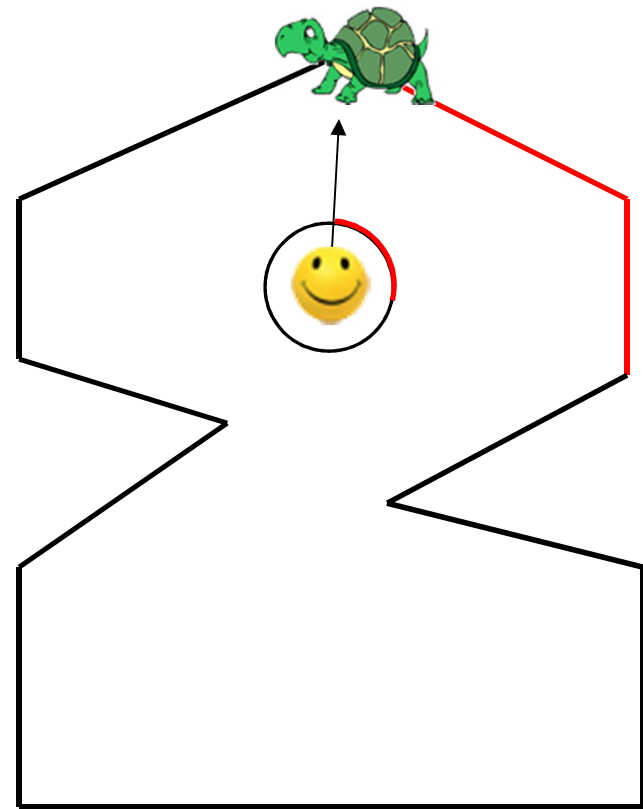
# A Better Way



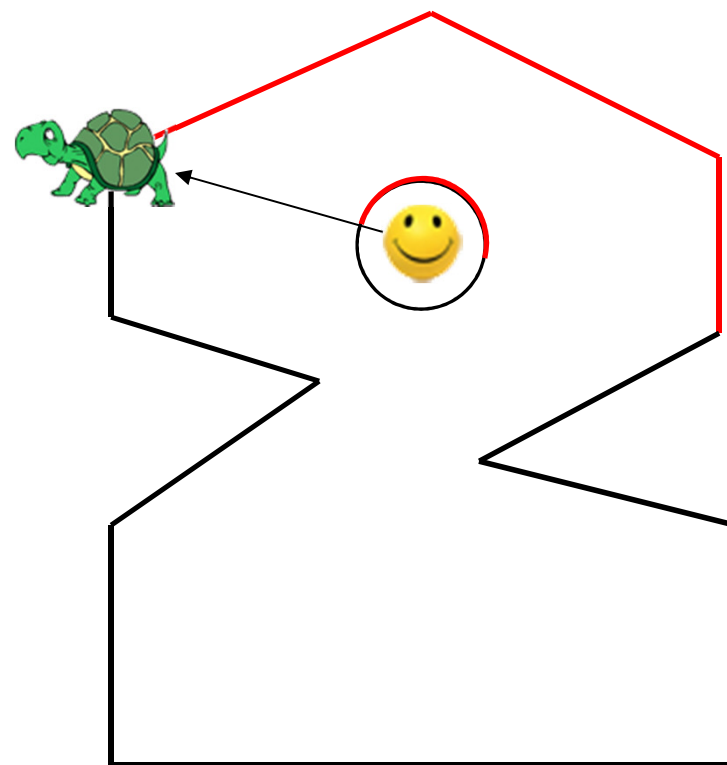
# A Better Way



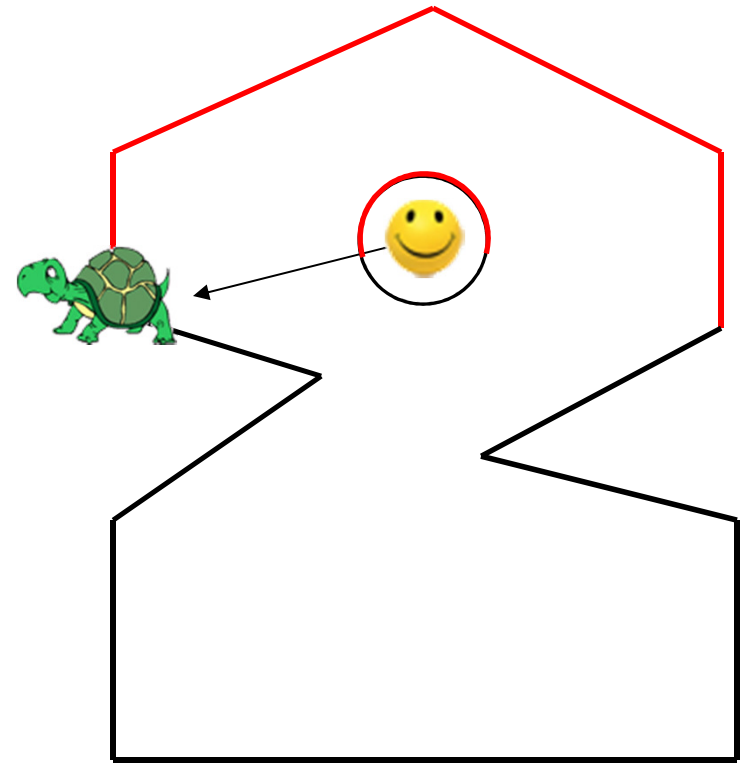
# A Better Way



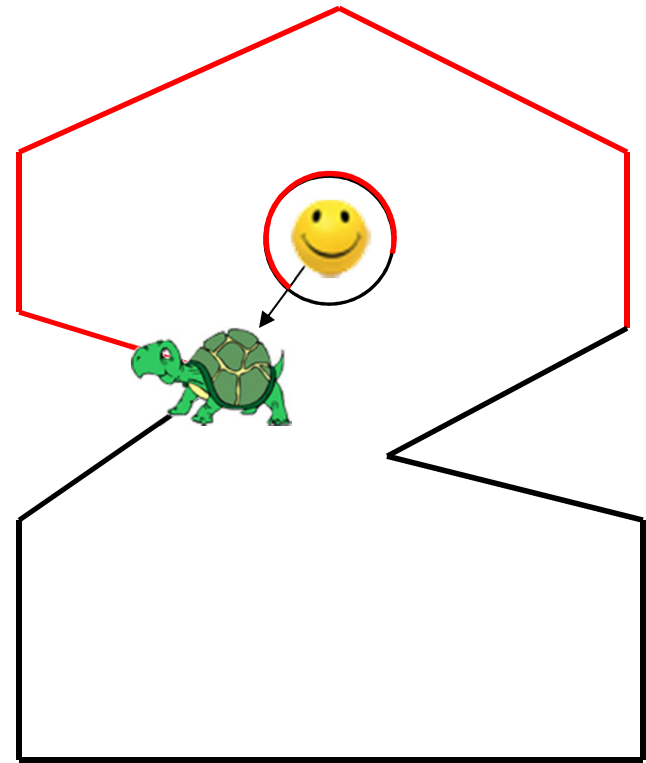
# A Better Way



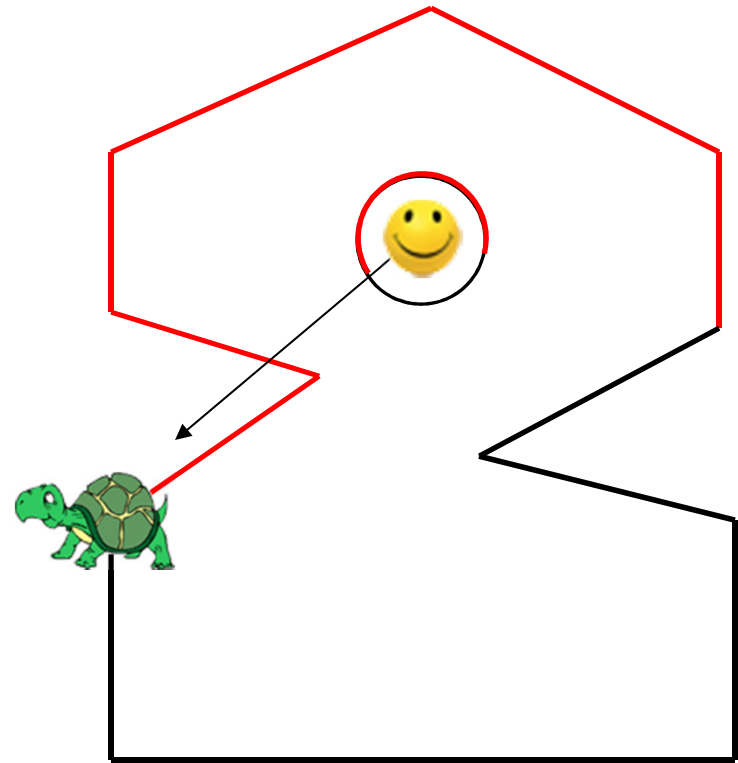
# A Better Way



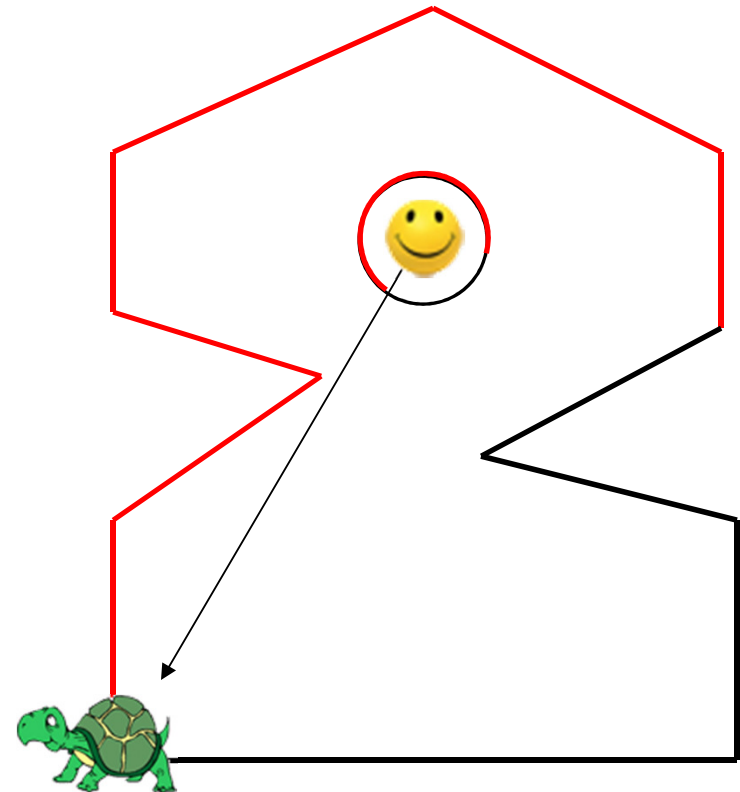
# A Better Way



# A Better Way

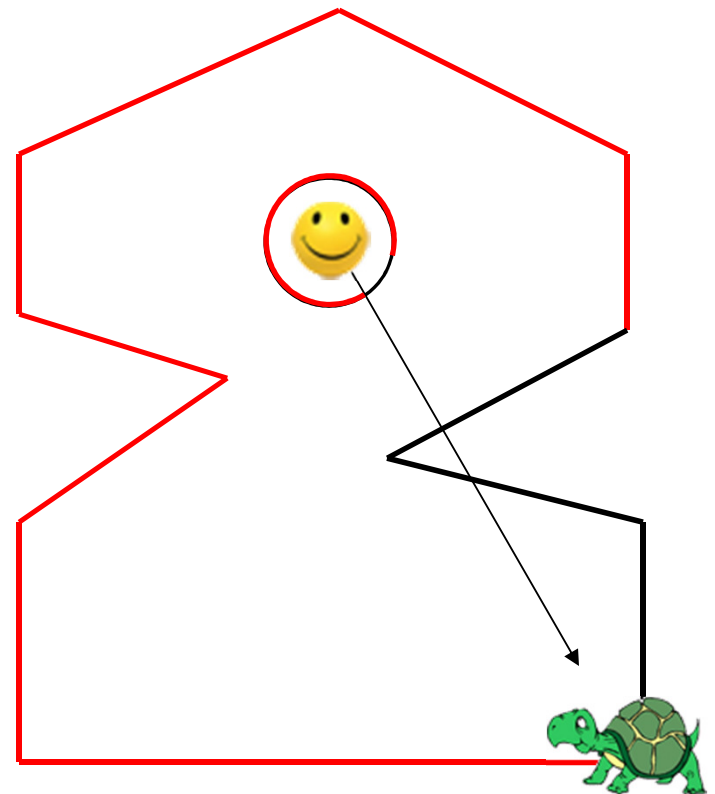


# A Better Way

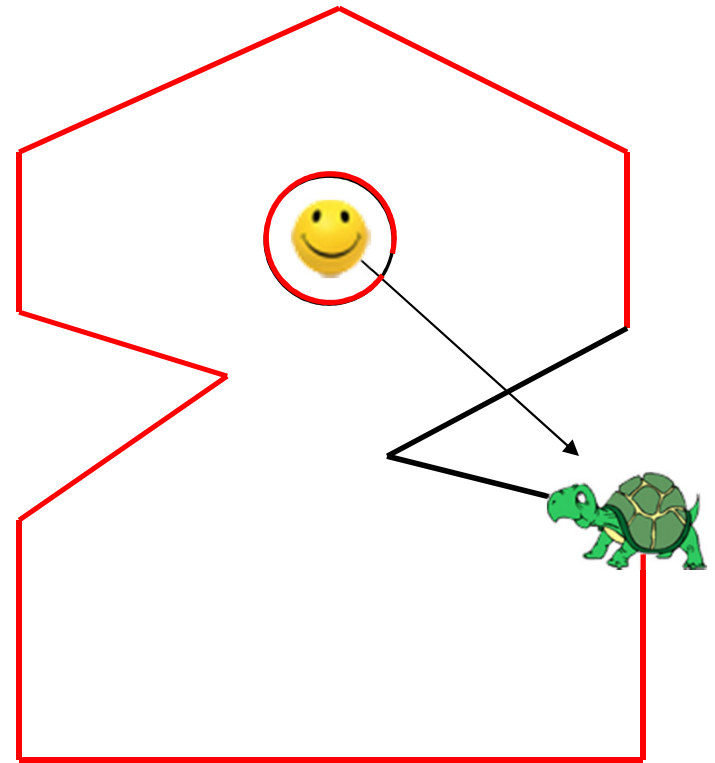




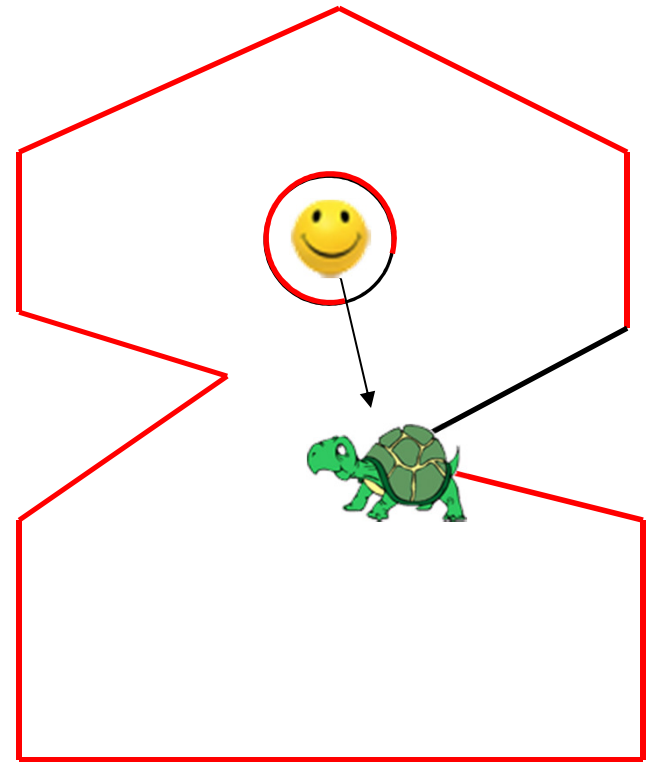
# A Better Way



# A Better Way

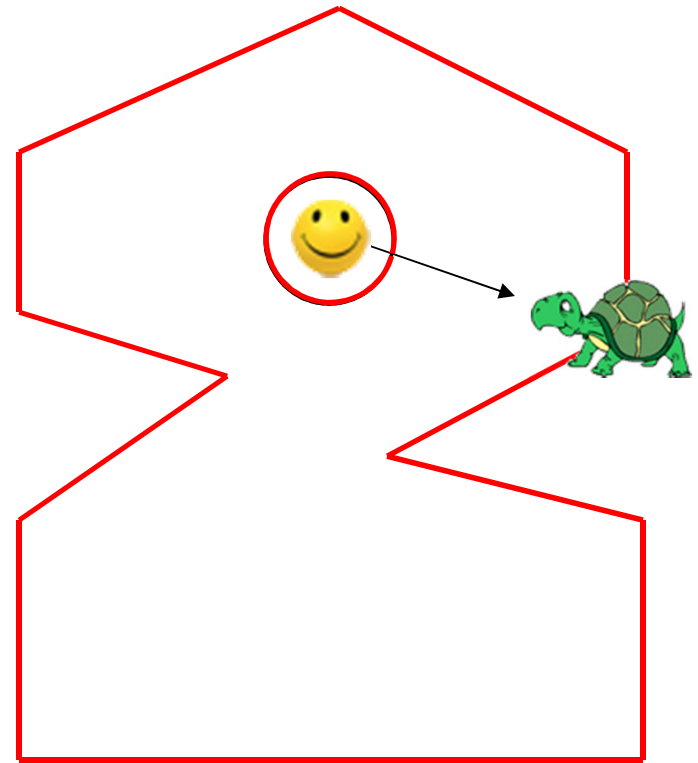


# A Better Way

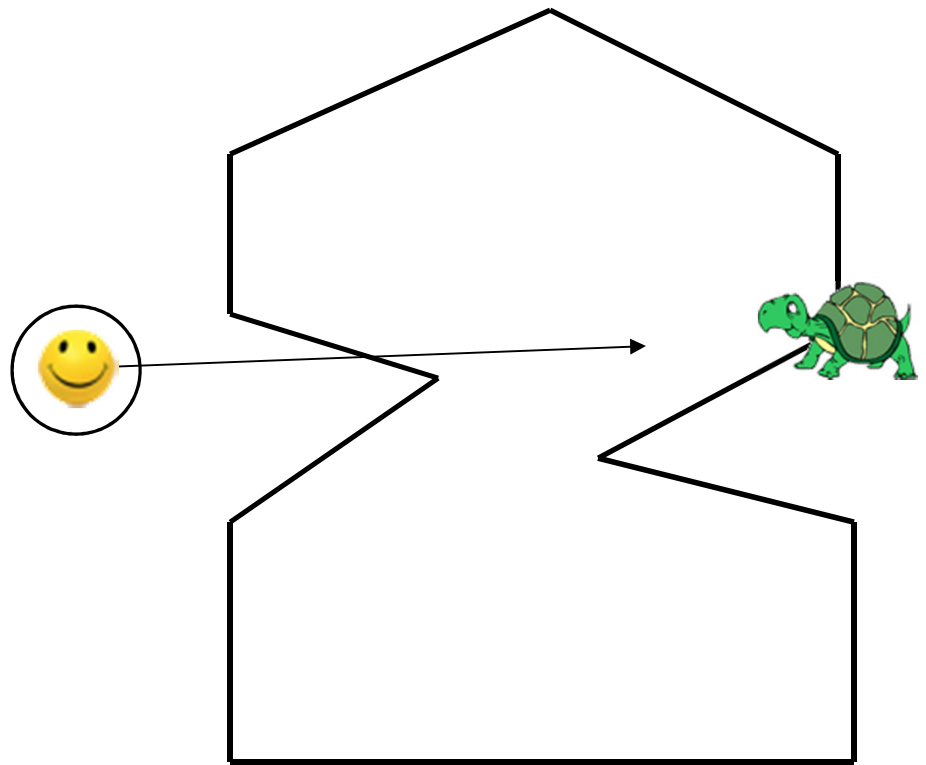


# A Better Way

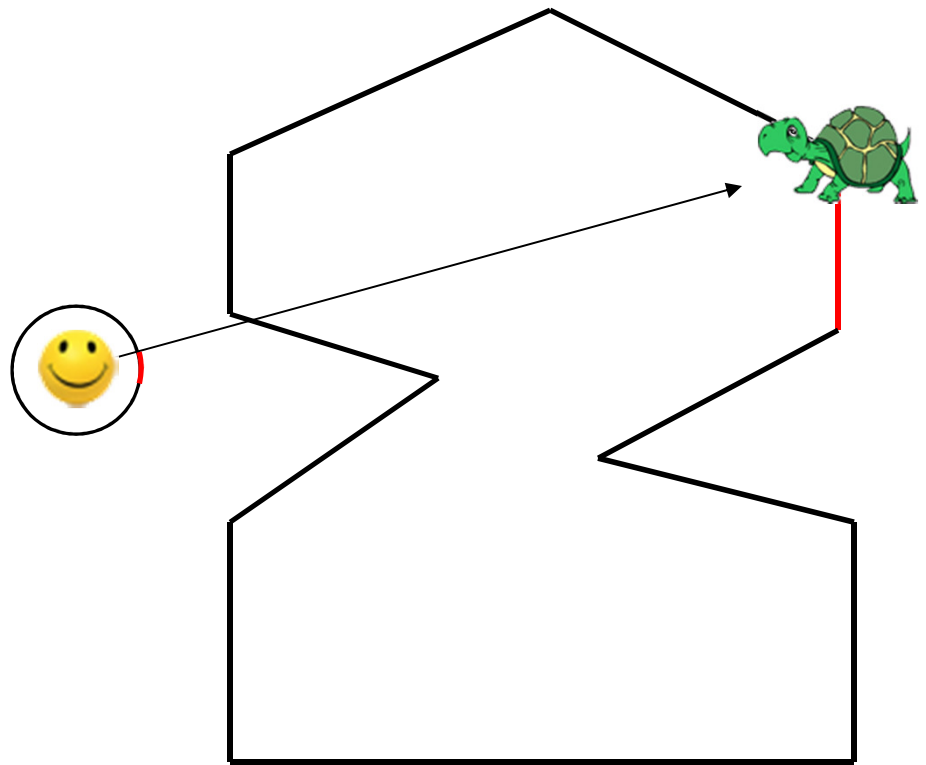
- One winding = inside



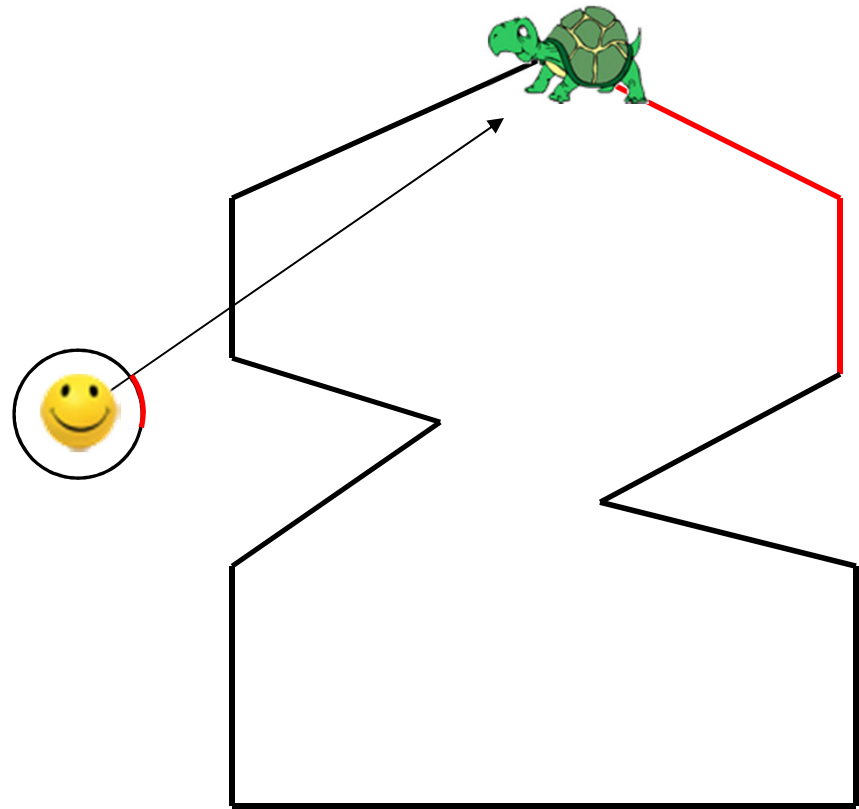
# A Better Way



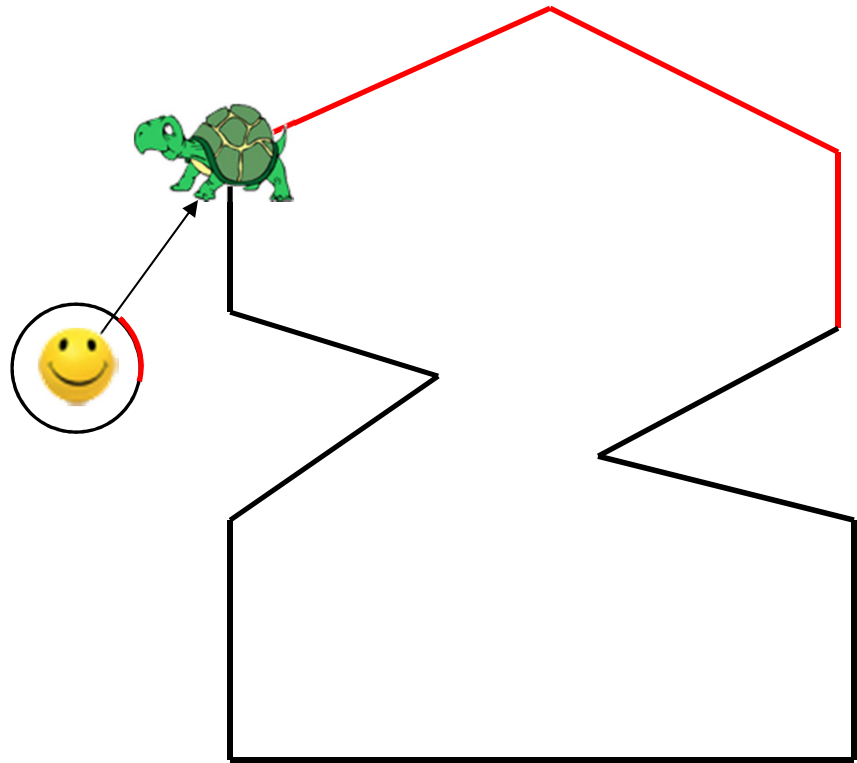
# A Better Way



# A Better Way

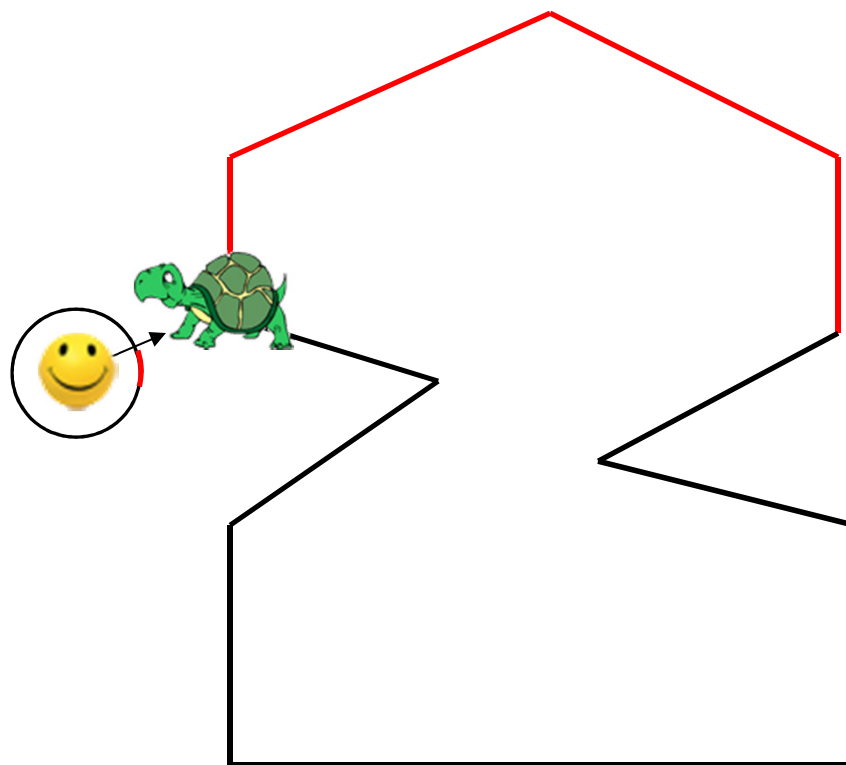


# A Better Way

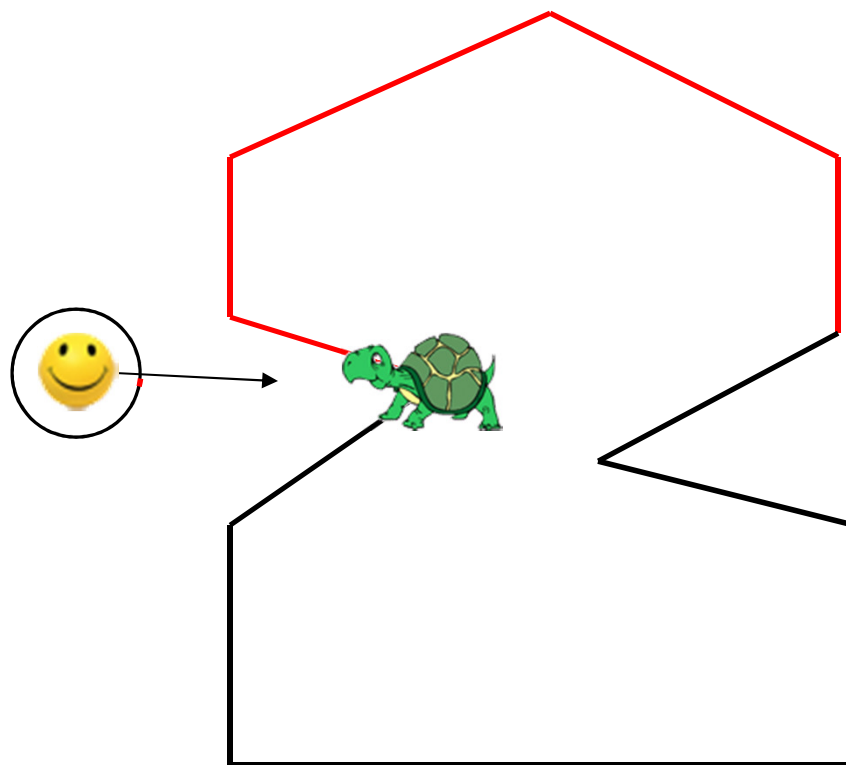




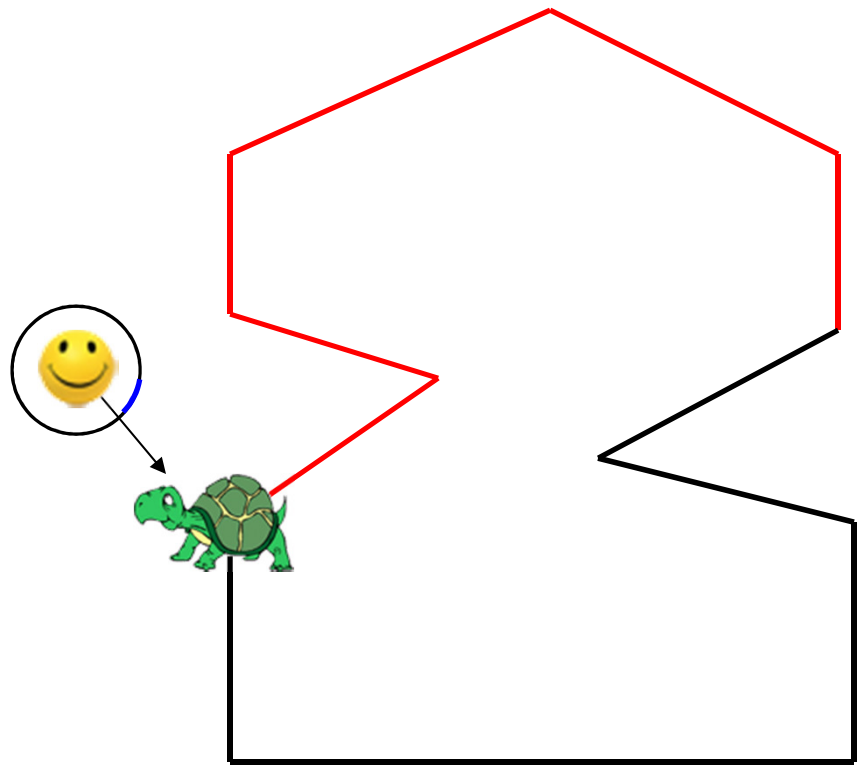
# A Better Way



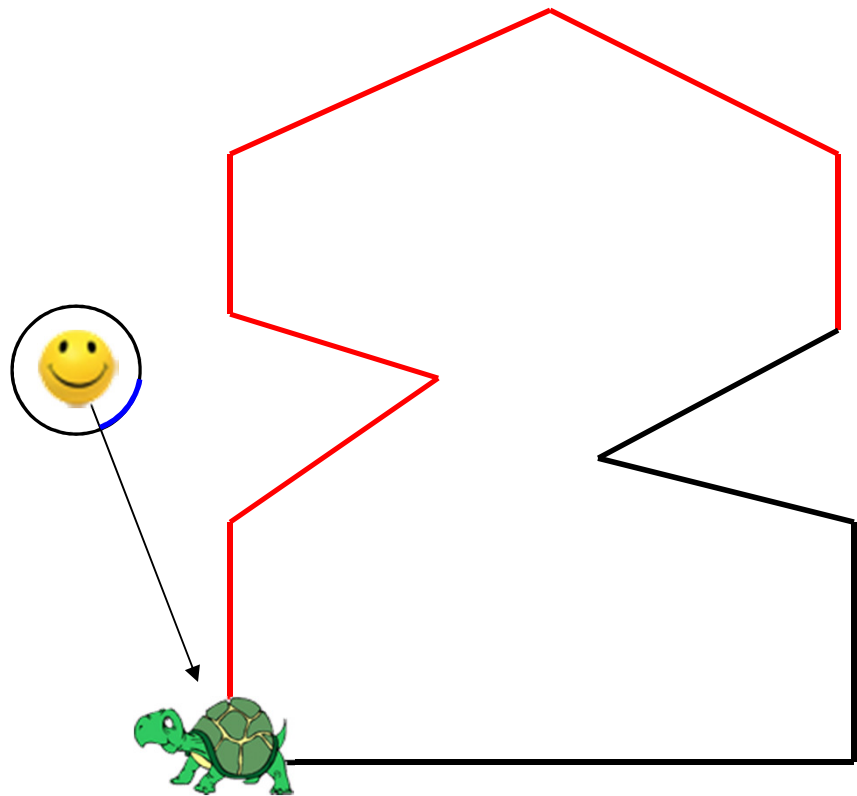
# A Better Way



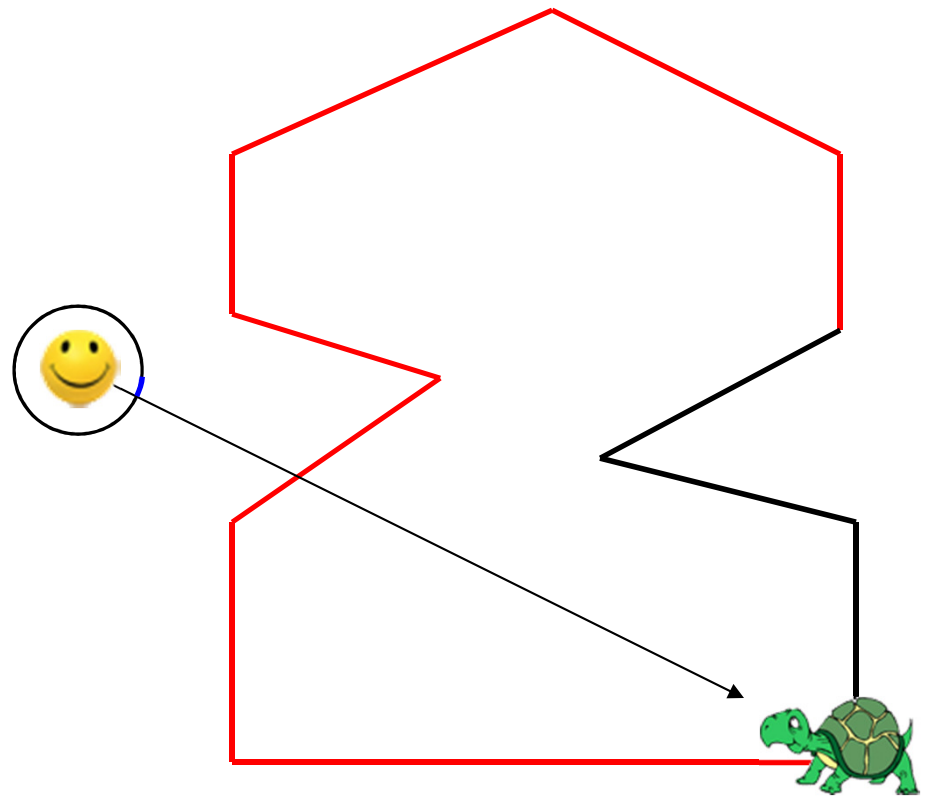
# A Better Way



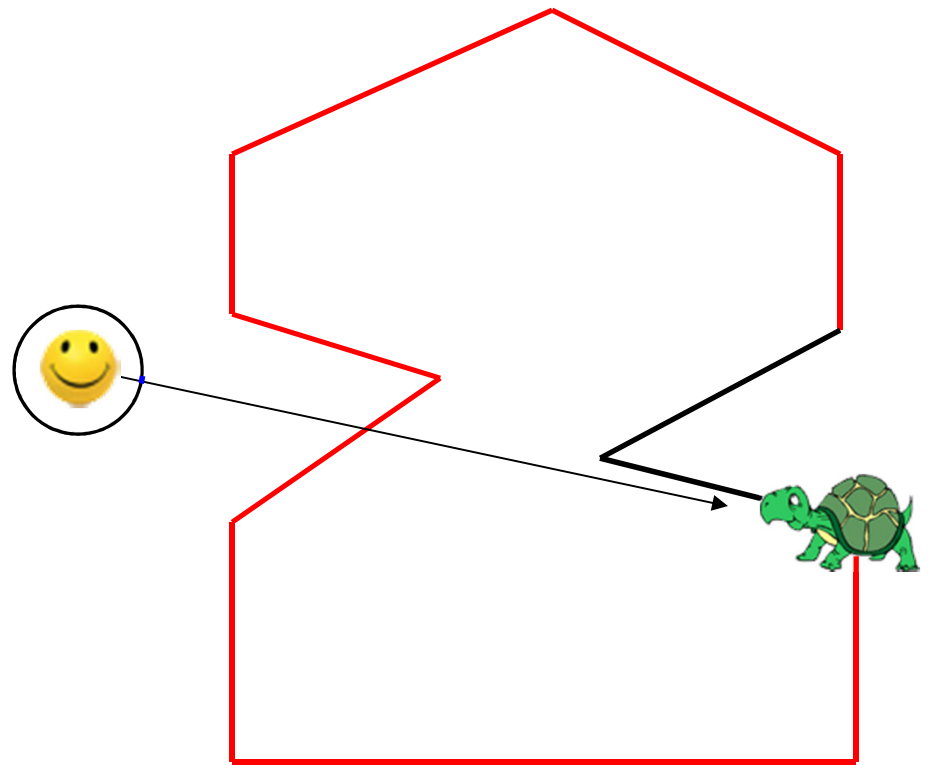
# A Better Way



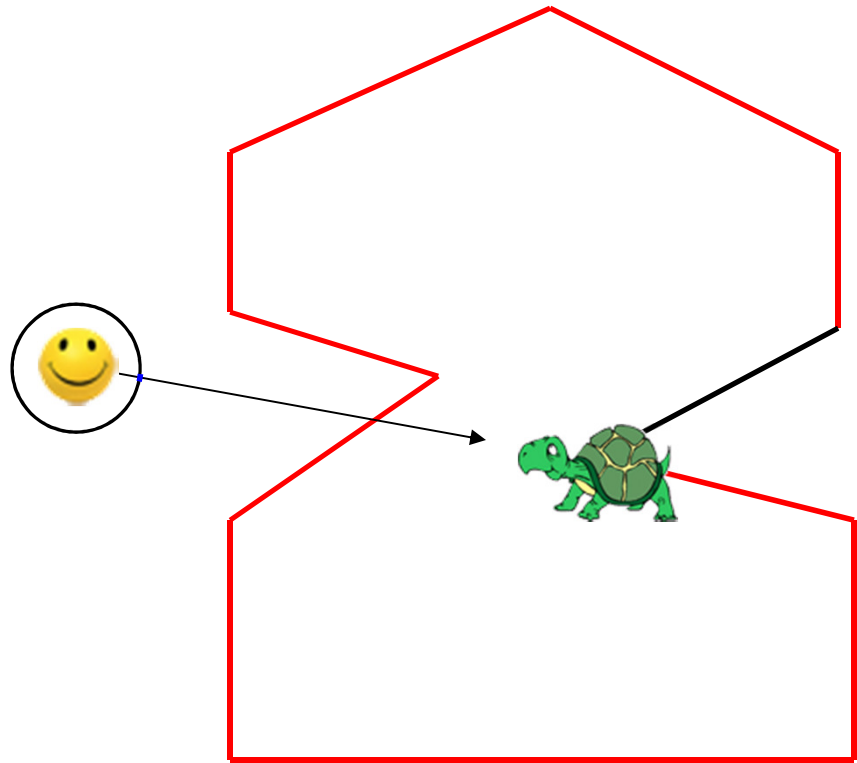
# A Better Way



# A Better Way

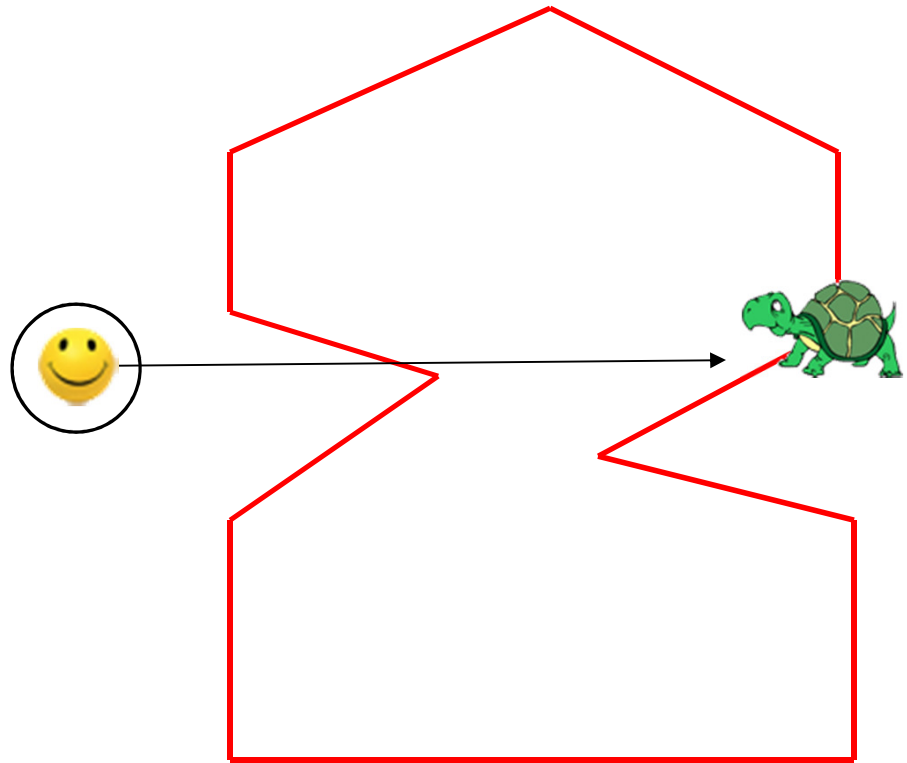


# A Better Way



# A Better Way

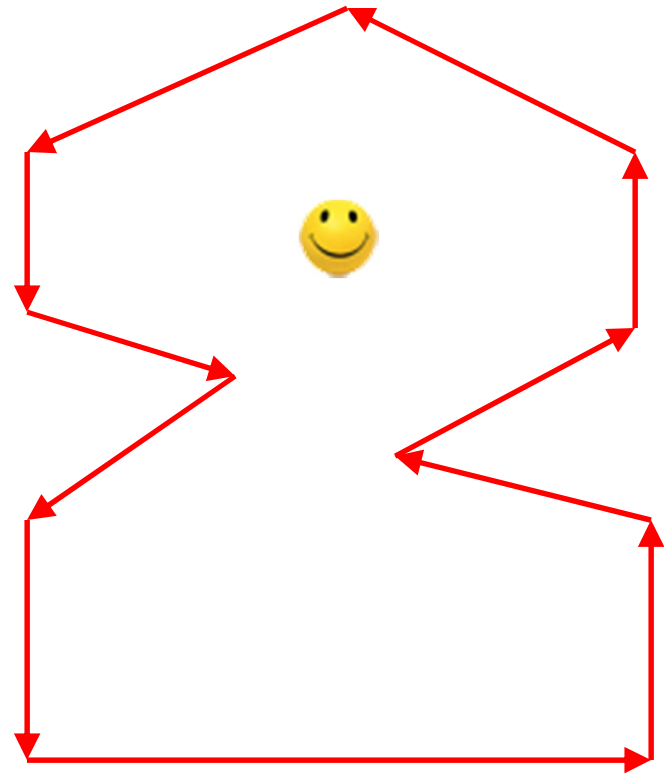
- zero winding = outside





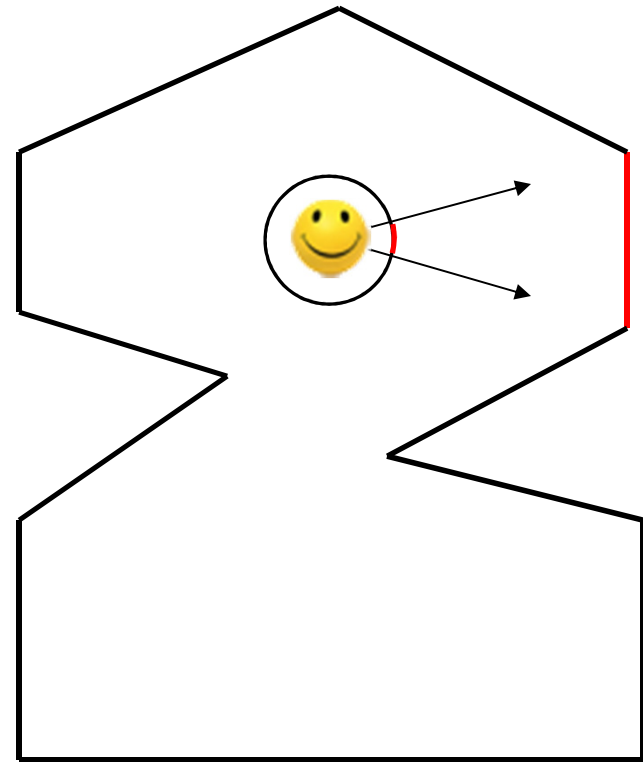
# Requirements

- Oriented edges
- Edges can be processed in any order



# Advantages

- Extends to 3D!
- Numerically stable
- Even works on models with holes:
  - Odd  $k$ : inside
  - Even  $k$ : outside
- No ray casting



# **Actual Implementation**

# Winding Number

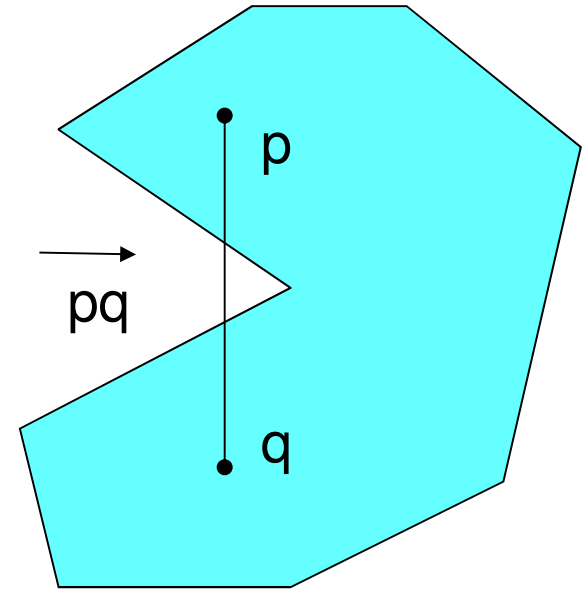
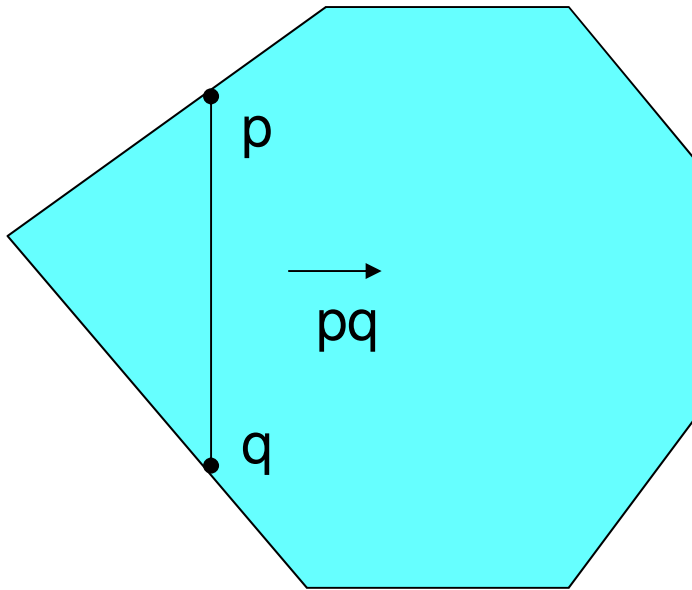
```
Int wn_PnPoly( Point P, Point* V, int n )
{
    int    wn = 0;    // the winding number counter

    // loop through all edges of the polygon
    for (int i=0; i<n; i++) {    // edge from V[i] to V[i+1]
        if (V[i].y <= P.y) {    // start y <= P.y
            if (V[i+1].y > P.y)    // an upward crossing
                if (isLeft( V[i], V[i+1], P) > 0)    // P left of edge
                    ++wn;    // have a valid up intersect
        }
        else {    // start y > P.y (no test needed)
            if (V[i+1].y <= P.y)    // a downward crossing
                if (isLeft( V[i], V[i+1], P) < 0)    // P right of edge
                    --wn;    // have a valid down intersect
        }
    }
    return wn;
}
```

# Topic

- Introduction
- Two lines Intersection Test
- Point inside polygon
- Convex hull
- Line Segments Intersection Algorithm

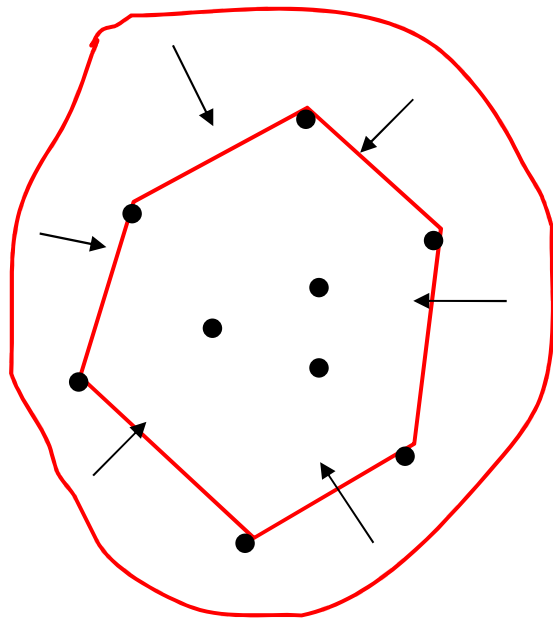
# Convex Hulls



Subset of  $S$  of the plane is convex, if for all pairs  $p, q$  in  $S$  the line segment  $pq$  is completely contained in  $S$ .

The **Convex Hull**  $CH(S)$  is the smallest convex set, which contains  $S$ .

# Convex hull of a set of points in the plane



Rubber band experiment

The convex hull of a set  $P$  of points is the unique convex polygon whose vertices are points of  $P$  and which contains all points from  $P$ .

# Convexity & Convex Hulls

source: O'Rourke, *Computational Geometry in C*

- A convex combination of points  $x_1, \dots, x_k$  is a sum of the form  $\alpha_1 x_1 + \dots + \alpha_k x_k$  where

$$\alpha_i \geq 0 \quad \forall i \quad \text{and} \quad \alpha_1 + \dots + \alpha_k = 1$$

- Convex hull of a set of points is the set of all convex combinations of points in the set.

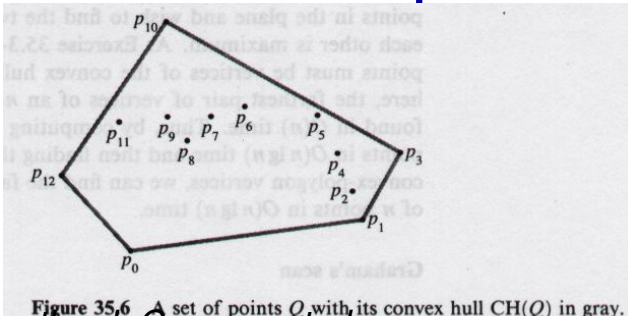
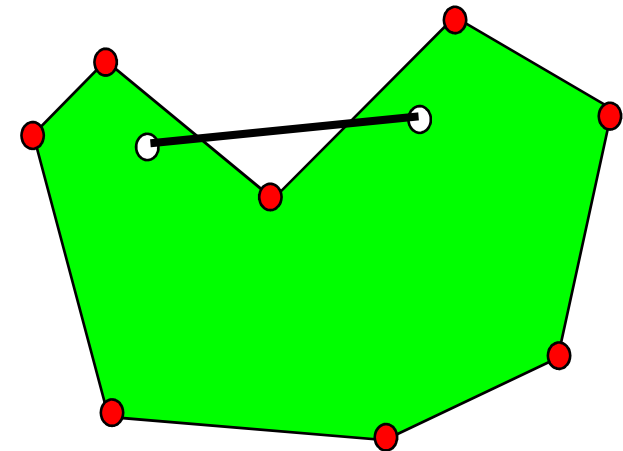
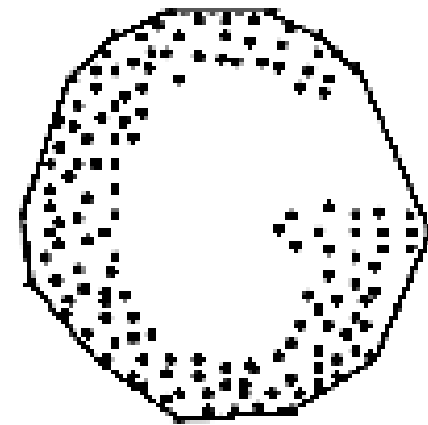


Figure 35.6 A set of points  $Q$  with its convex hull  $CH(Q)$  in gray.

source: 91.503 textbook Cormen et al.



nonconvex polygon

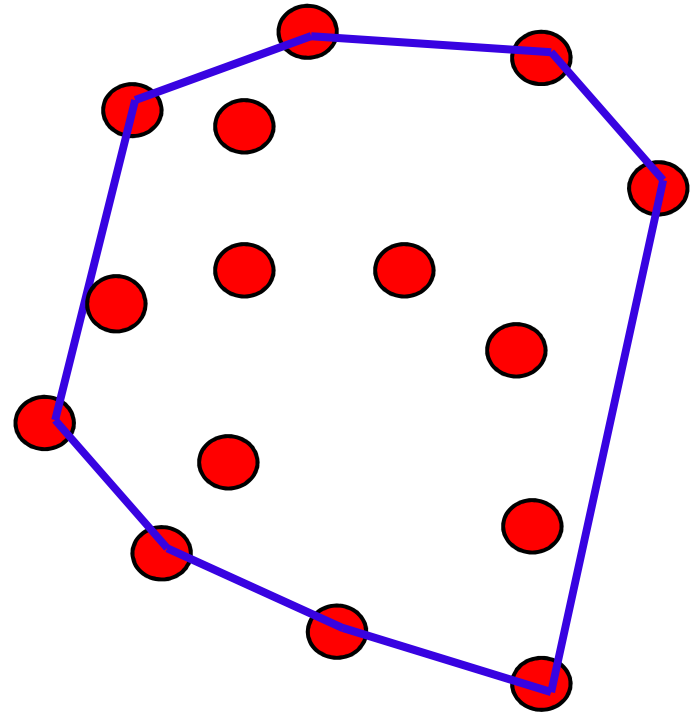


convex hull of a point set



# Convex Hull

- Input:
  - Set  $S = \{s_1, \dots, s_n\}$  of  $n$  points
- Output:
  - Find its convex hull
- Many algorithms:
  - Naïve –  $O(n^3)$
  - Insertion –  $O(n \log n)$
  - Divide and Conquer –  $O(n \log n)$
  - Gift Wrapping –  $O(nh)$ ,  $h$  = no of points on the hull
  - Graham Scan –  $O(n \log n)$



# Naive Algorithms for Extreme Points

## **Algorithm:** INTERIOR POINTS

```
for each i do
  for each j  $\neq$  i do
    for each k  $\neq$  j  $\neq$  i do
      for each L  $\neq$  k  $\neq$  j  $\neq$  i do
        if  $p_L$  in triangle( $p_i, p_j, p_k$ )
          then  $p_L$  is nonextreme
```

**$O(n^4)$**

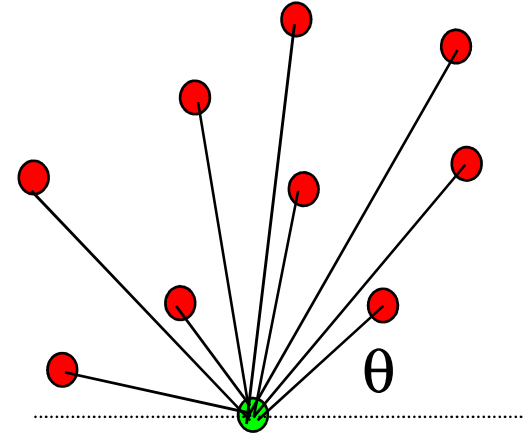
## **Algorithm:** EXTREME EDGES

```
for each i do
  for each j  $\neq$  i do
    for each k  $\neq$  j  $\neq$  i do
      if  $p_k$  is not left or on ( $p_i, p_j$ )
        then ( $p_i, p_j$ ) is not extreme
```

**$O(n^3)$**

# Algorithms: 2D Gift Wrapping

- Use one extreme edge as an anchor for finding the next



## Algorithm: GIFT WRAPPING

$i_0 \leftarrow$  index of the lowest point

$i \leftarrow i_0$

repeat

  for each  $j \neq i$

    Compute counterclockwise angle  $\theta$  from previous hull edge

$k \leftarrow$  index of point with smallest  $\theta$

  Output  $(p_i, p_k)$  as a hull edge

$i \leftarrow k$

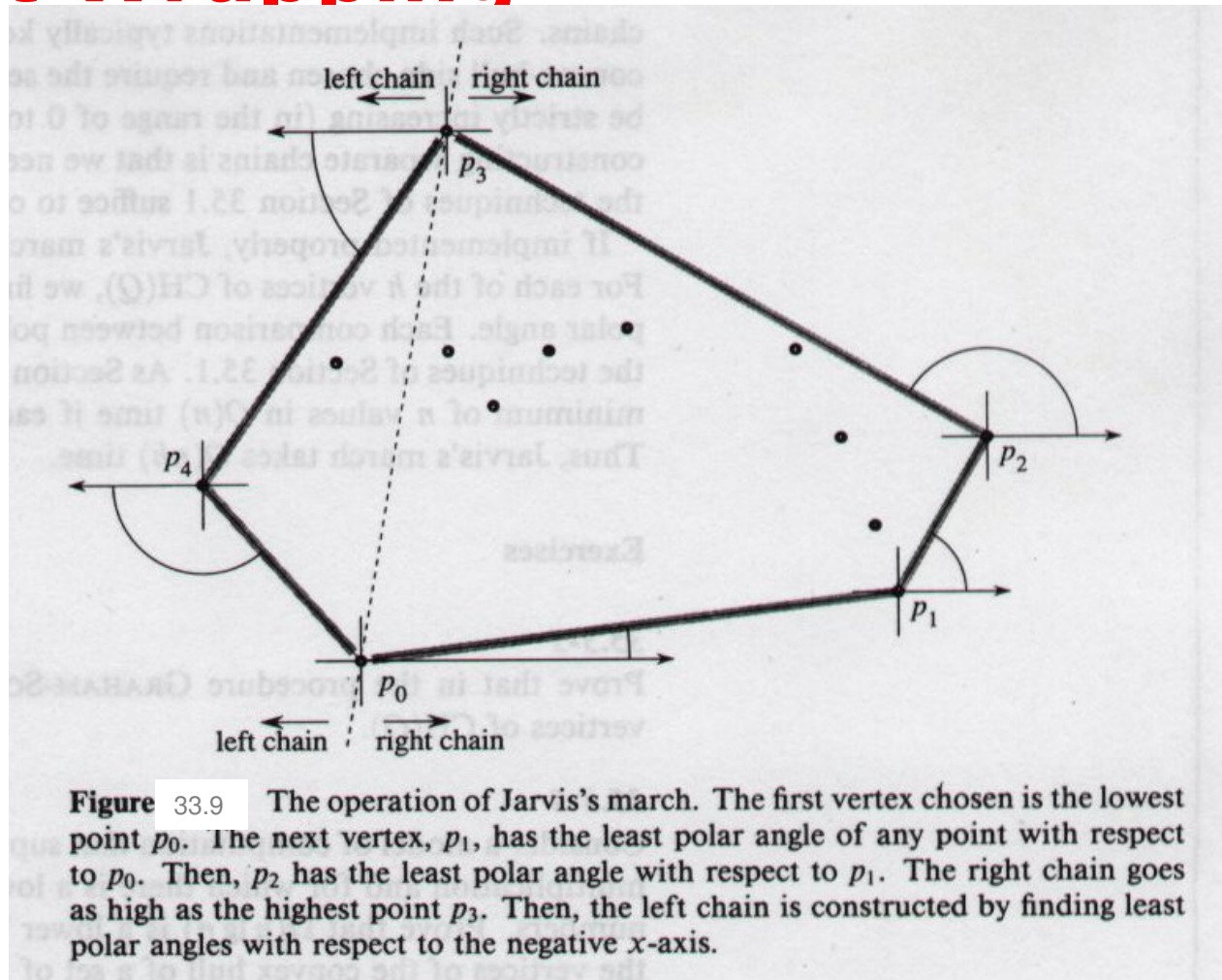
until  $i = i_0$

source: O'Rourke, Computational Geometry in C

$O(n^2)$

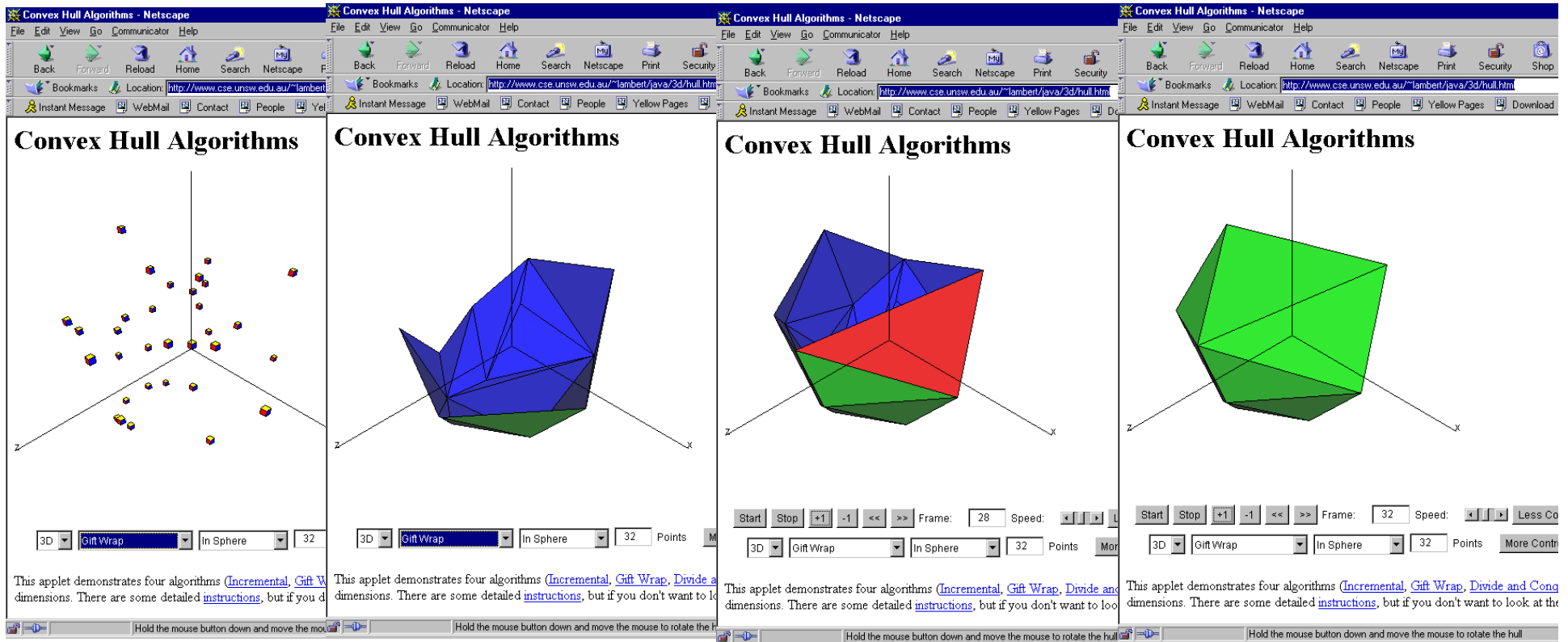
# Gift Wrapping

source: 91.503 textbook Cormen et al.



Output Sensitivity:  $O(n^2)$  run-time is actually  $O(nh)$  where  $h$  is the number of vertices of the convex hull.

# Algorithms: 3D Gift Wrapping



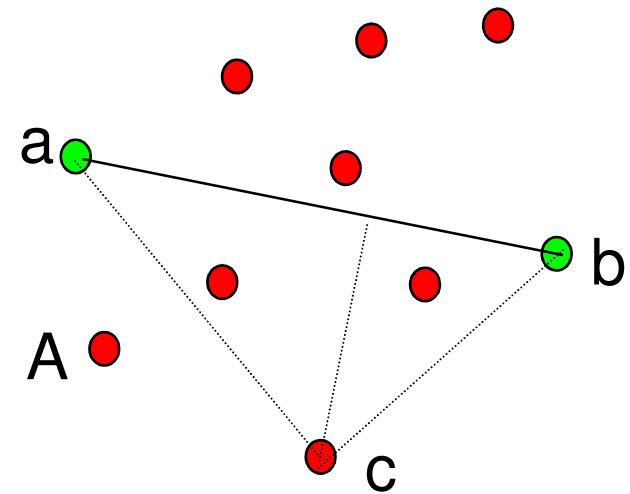
$O(n^2)$  time

[output sensitive:  $O(nF)$  for  $F$  faces on hull]

*CxHull Animations:* <http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html>

# Algorithms: 2D QuickHull

- Concentrate on points close to hull boundary
- Named for similarity to Quicksort



## Algorithm: QUICK HULL

*finds one of upper or lower hull*

function QuickHull(a,b,S)

  if  $S = \emptyset$  return()

  else

    c  $\leftarrow$  index of point with max distance from ab

    A  $\leftarrow$  points strictly right of (a,c)

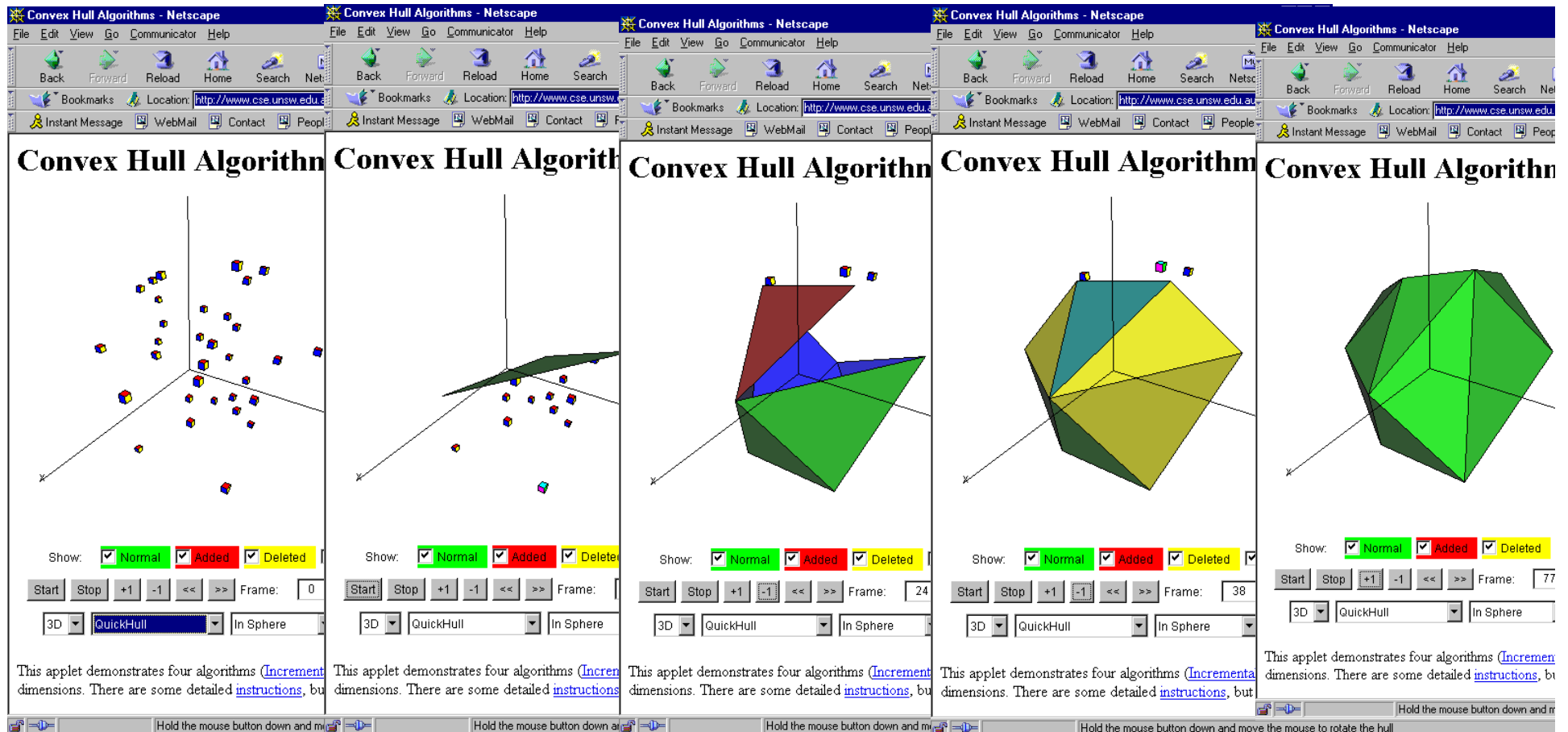
    B  $\leftarrow$  points strictly right of (c,b)

  return QuickHull(a,c,A) + (c) + QuickHull(c,b,B)

**$O(n^2)$**

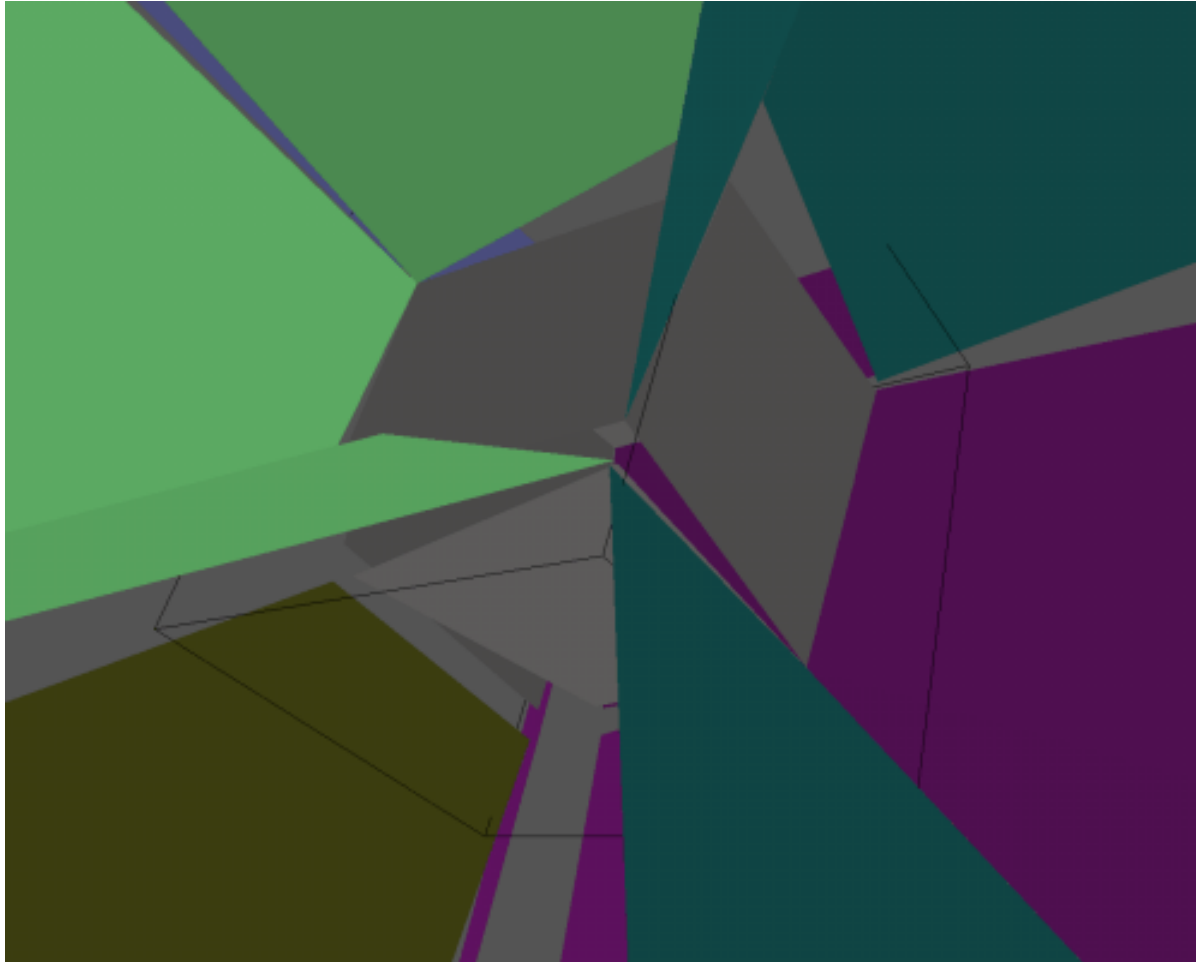
*source: O'Rourke, Computational Geometry in C*

# Algorithms: 3D QuickHull



CxHull Animations: <http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html>

## Algorithms: $\geq 2D$



Convex Hull  
boundary is  
intersection of  
hyperplanes, so  
worst-case  
combinatorial size  
(not necessarily running  
time) complexity is  
in:  $\Theta(n^{\lfloor d/2 \rfloor})$

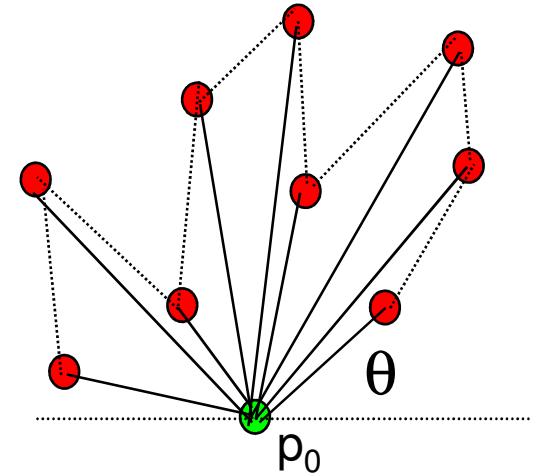
Qhull: <http://www.qhull.org/>



# Graham's Algorithm

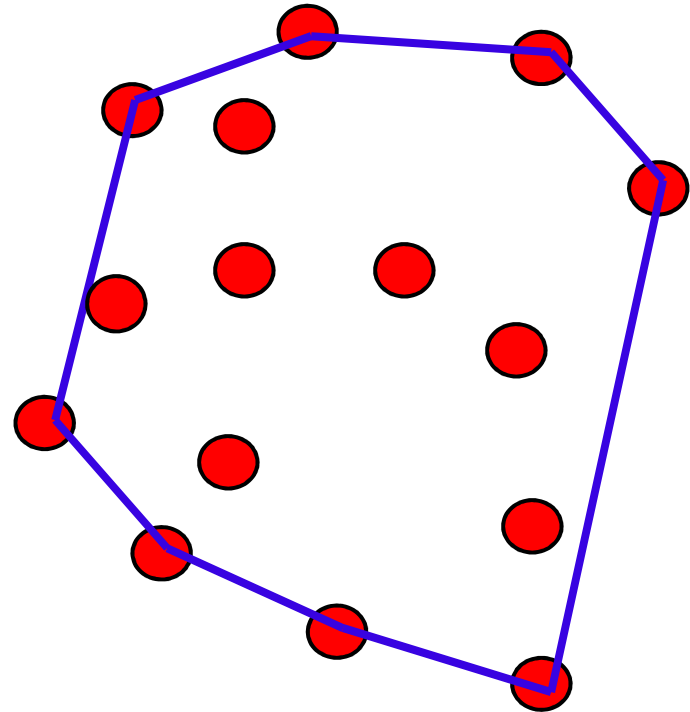
*source: O'Rourke, Computational Geometry in C*

- Points sorted angularly provide “star-shaped” starting point
- Prevent “dents” as you go via convexity testing



# Graham Scan

- Polar sort the points around a point inside the hull
- Scan points in counter-clockwise (CCW) order
  - Discard any point that causes a clockwise (CW) turn
    - If CCW, advance
    - If !CCW, discard current point and back up

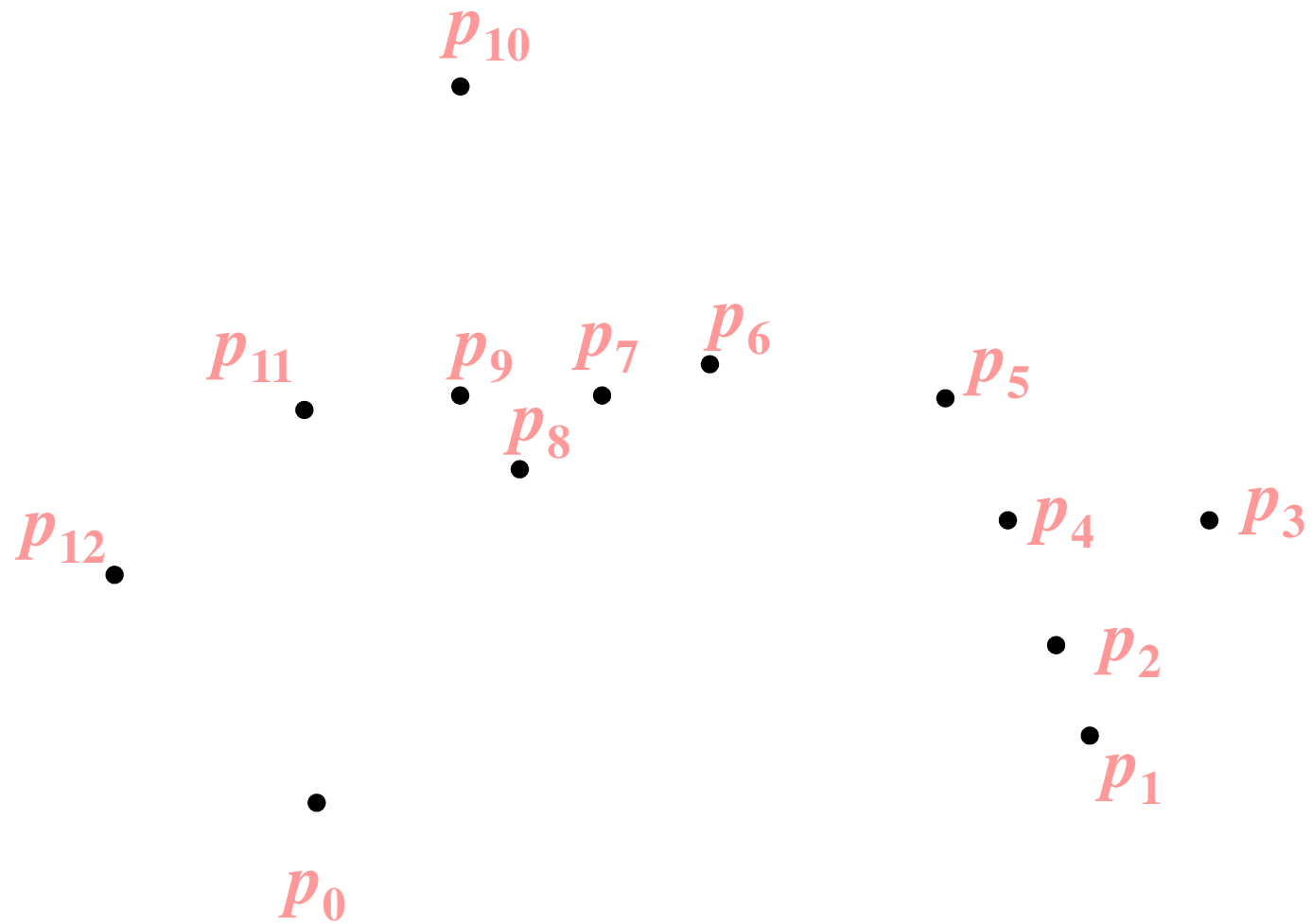


# Graham Scan

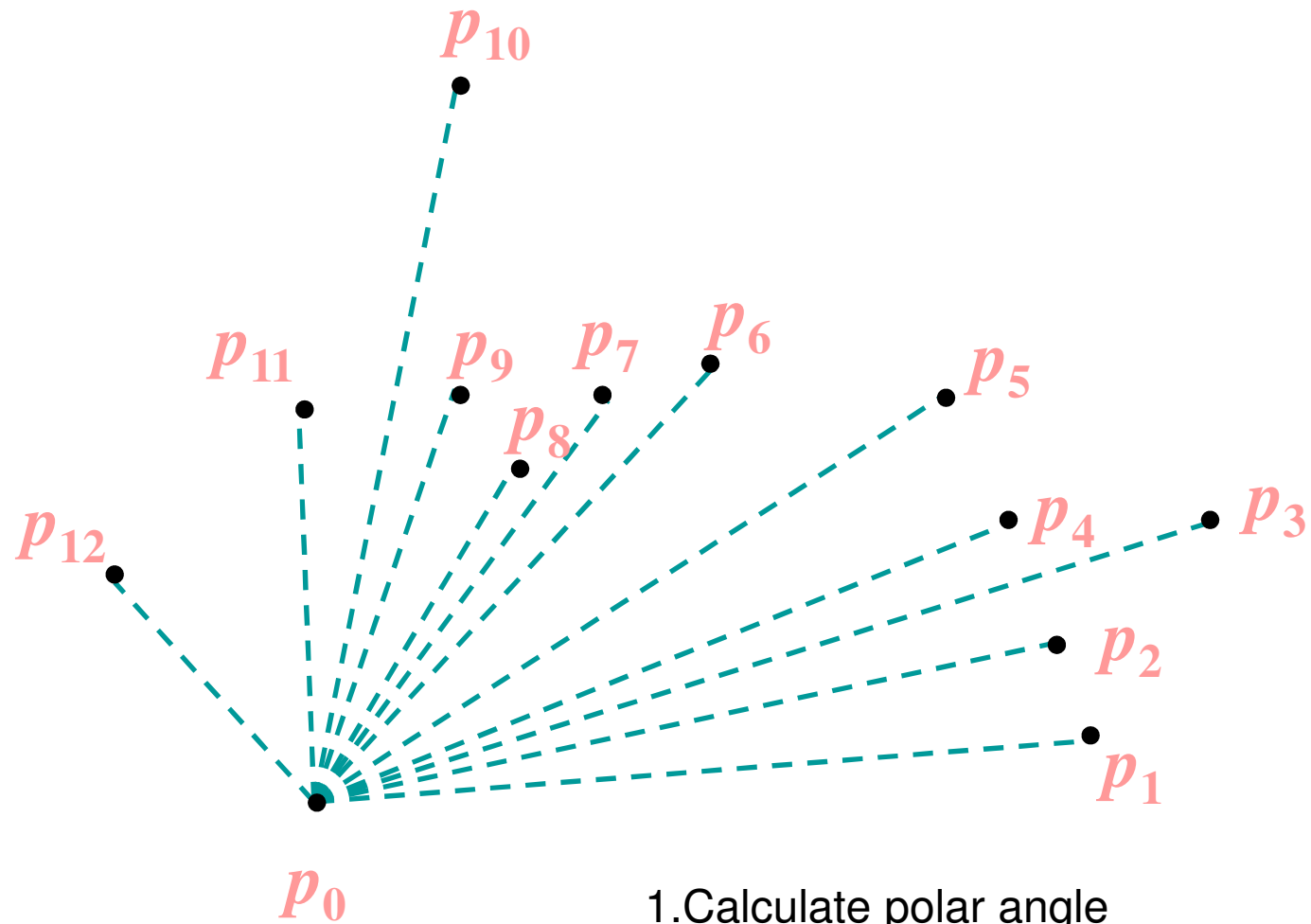
GRAHAM-SCAN( $Q$ )

```
1  let  $p_0$  be the point in  $Q$  with the minimum  $y$ -coordinate,  
   or the leftmost such point in case of a tie  
2  let  $\langle p_1, p_2, \dots, p_m \rangle$  be the remaining points in  $Q$ ,  
   sorted by polar angle in counterclockwise order around  $p_0$   
   (if more than point has the same angle, remove all but  
   the one that is farthest from  $p_0$ )  
3   $top[S] \leftarrow 0$   
4  PUSH( $p_0, S$ )  
5  PUSH( $p_1, S$ )  
6  PUSH( $p_2, S$ )  
7  for  $i \leftarrow 3$  to  $m$   
8      do while the angle formed by points NEXT-TO-TOP( $S$ ),  
               TOP( $S$ ), and  $p_i$  makes a nonleft turn  
9          do POP( $S$ )  
10     PUSH( $S, p_i$ )  
11  return  $S$ 
```

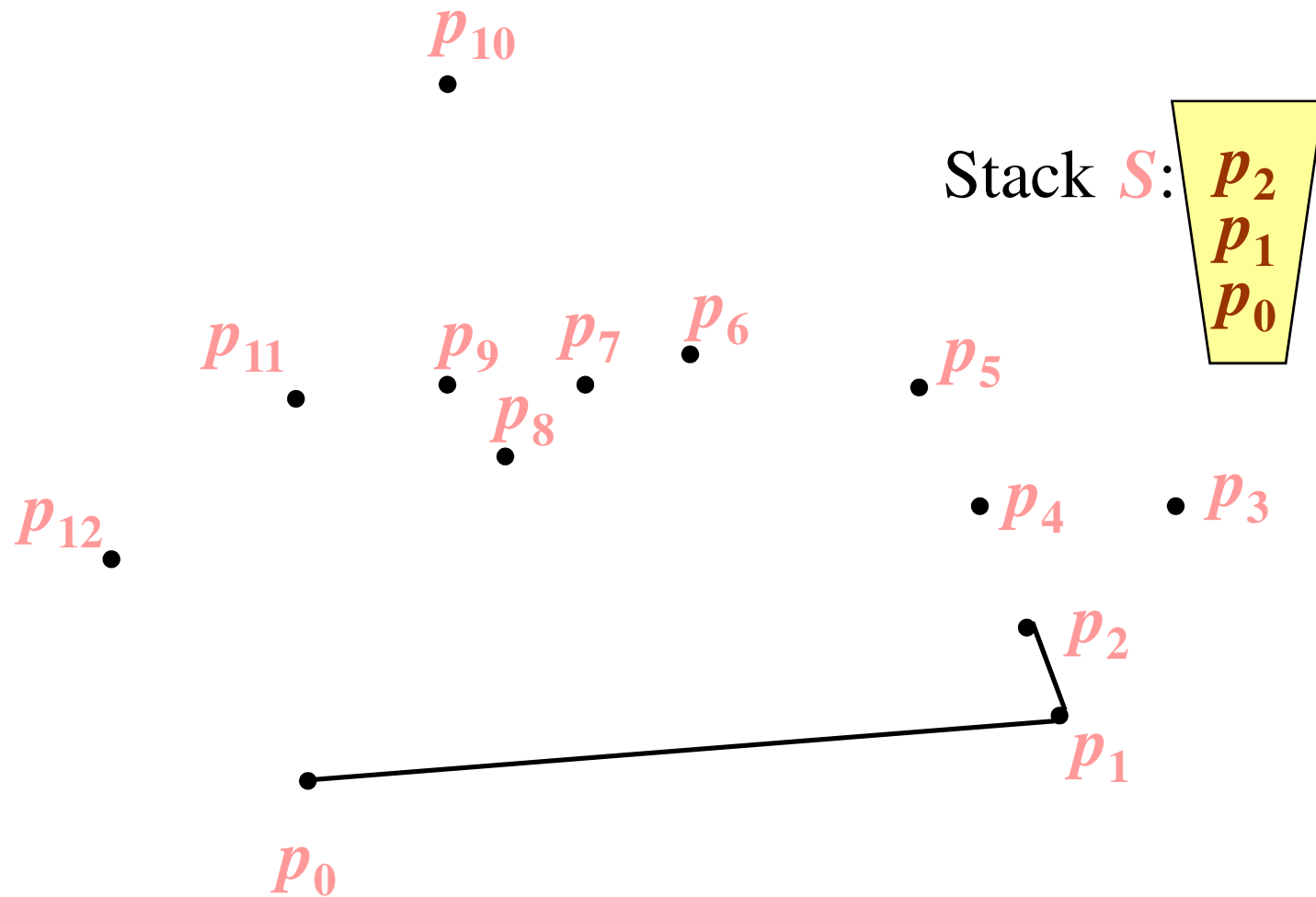
# Graham-Scan : (1/11)



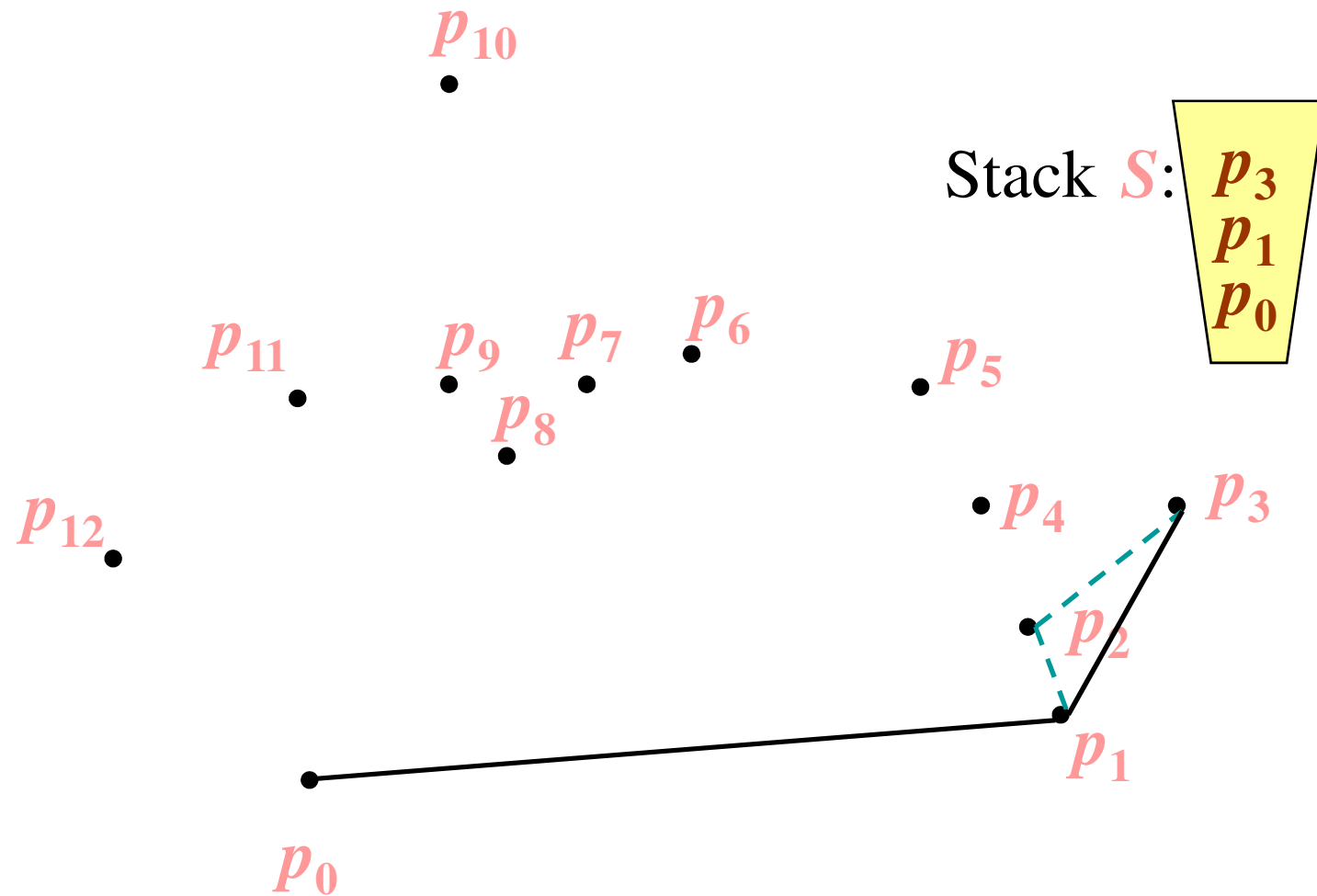
# Graham-Scan :(1/11)



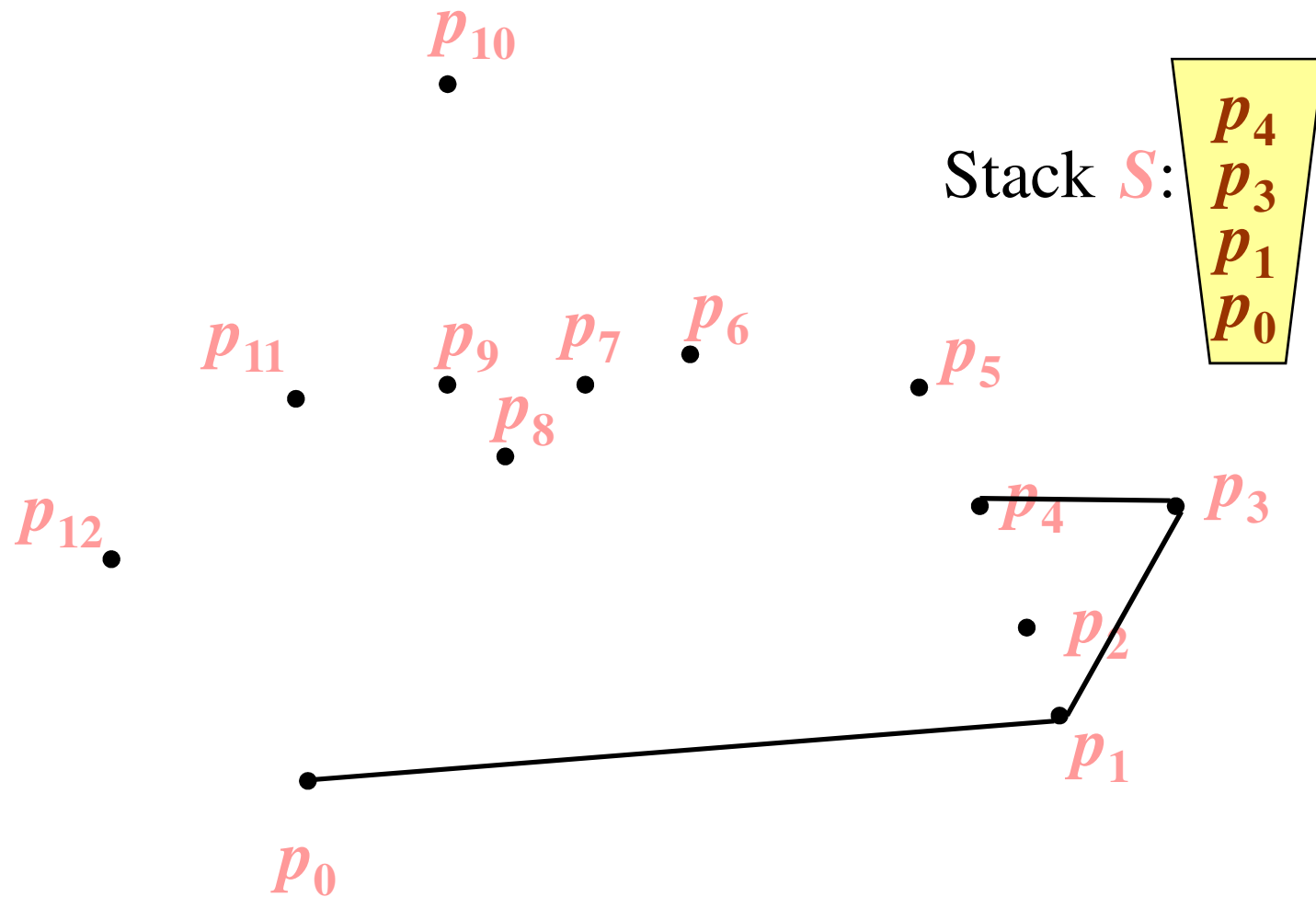
# Graham-Scan : (2/11)



# Graham-Scan : (3/11)

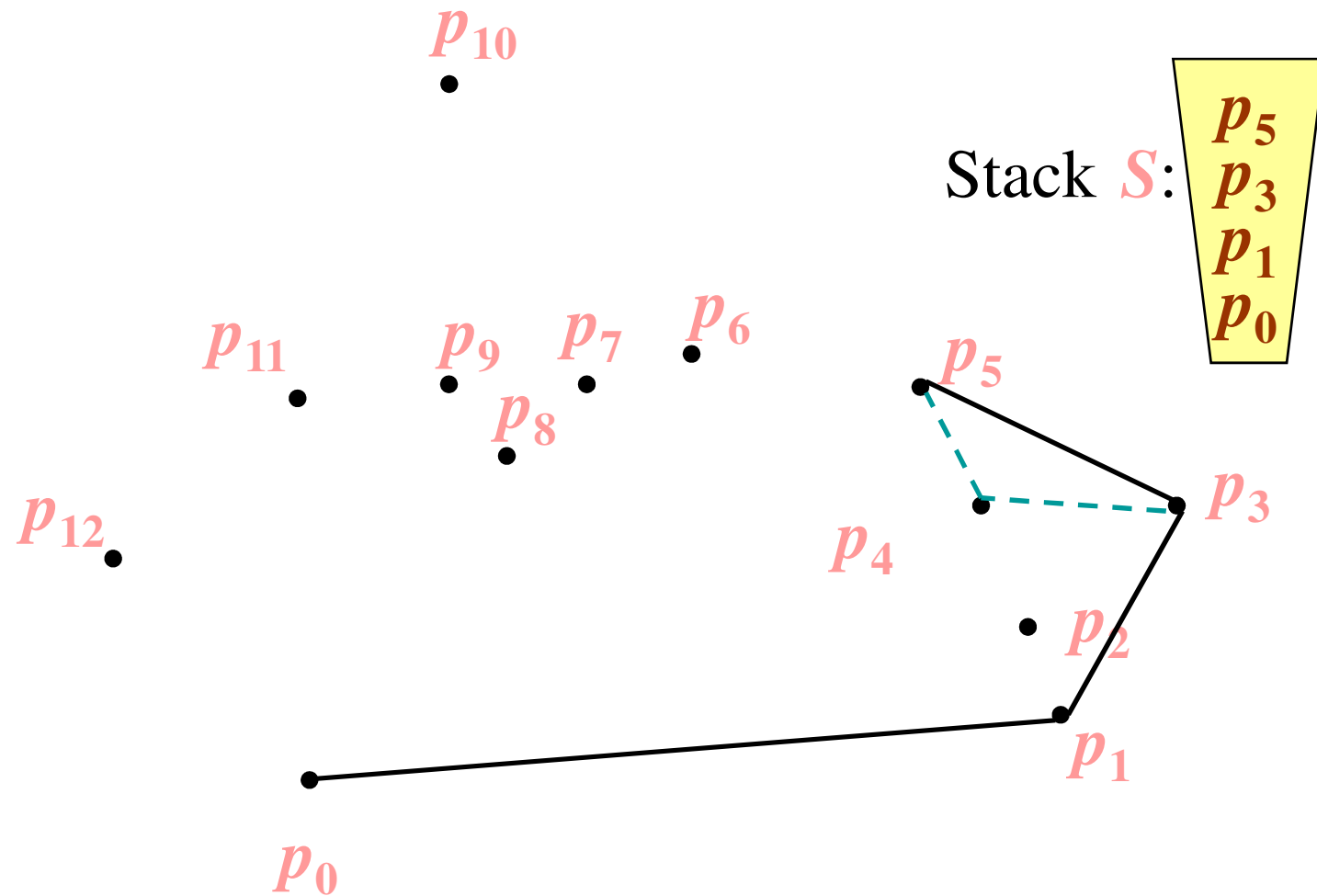


# Graham-Scan : (4/11)

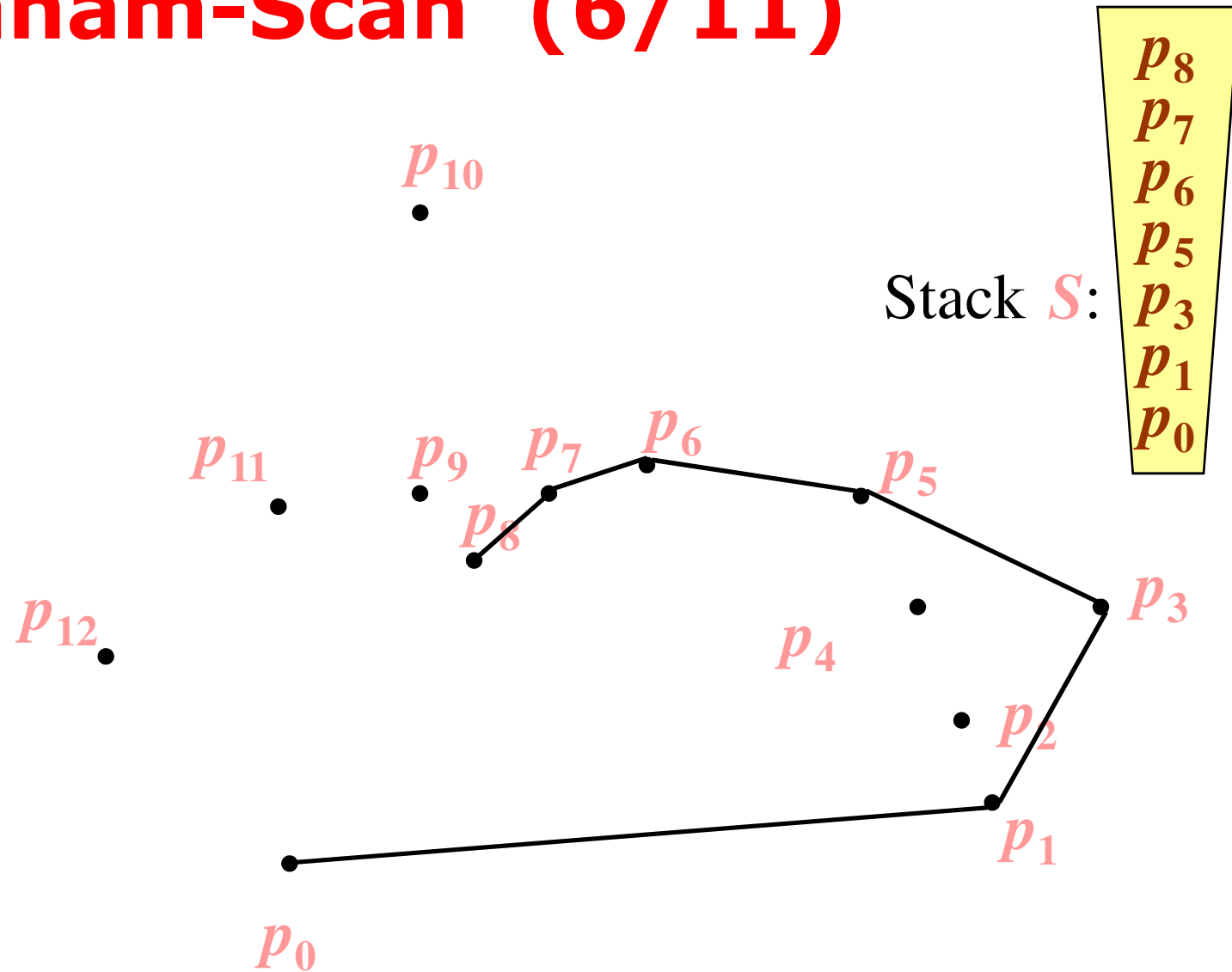




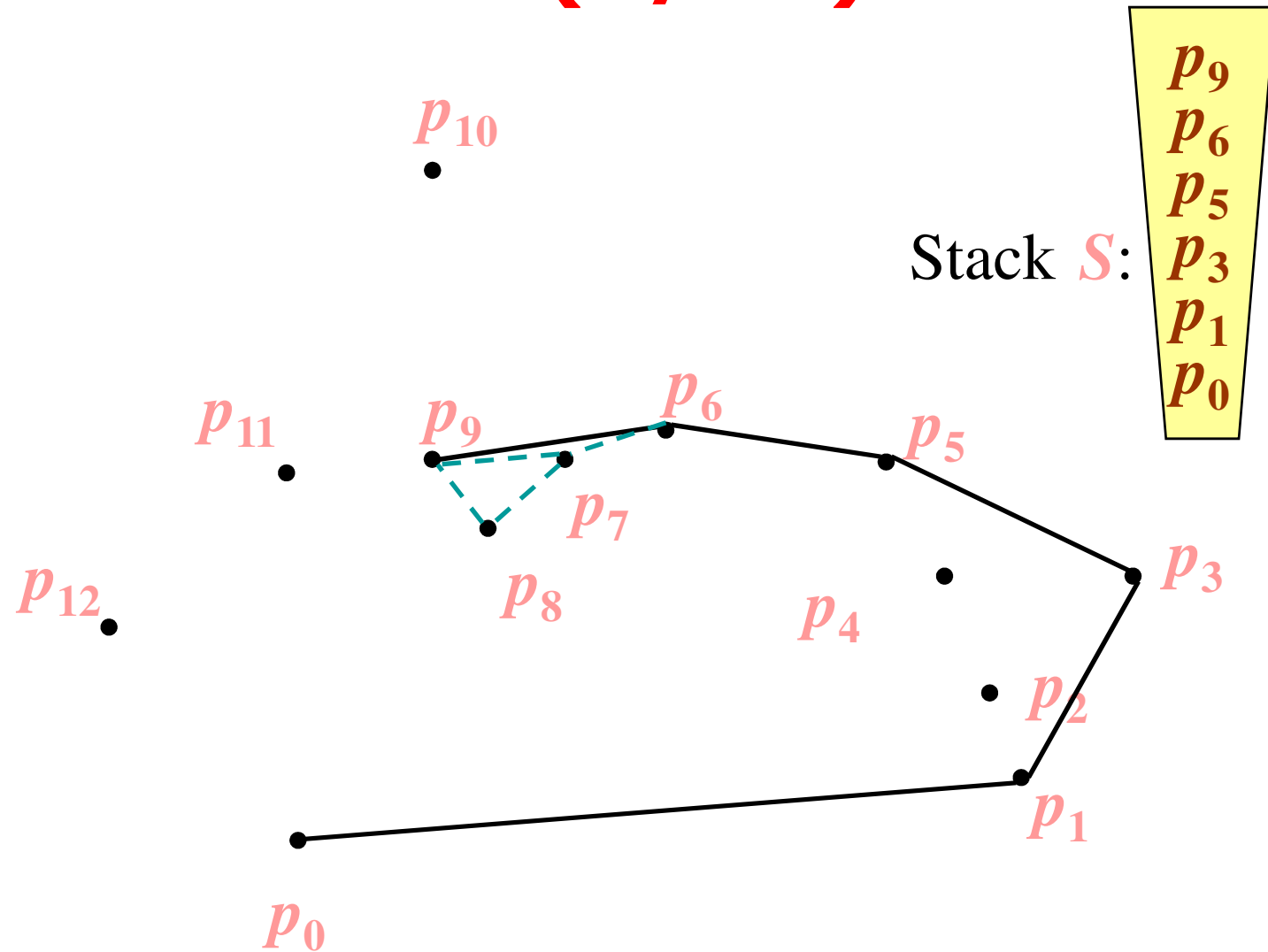
# Graham-Scan (5/11)



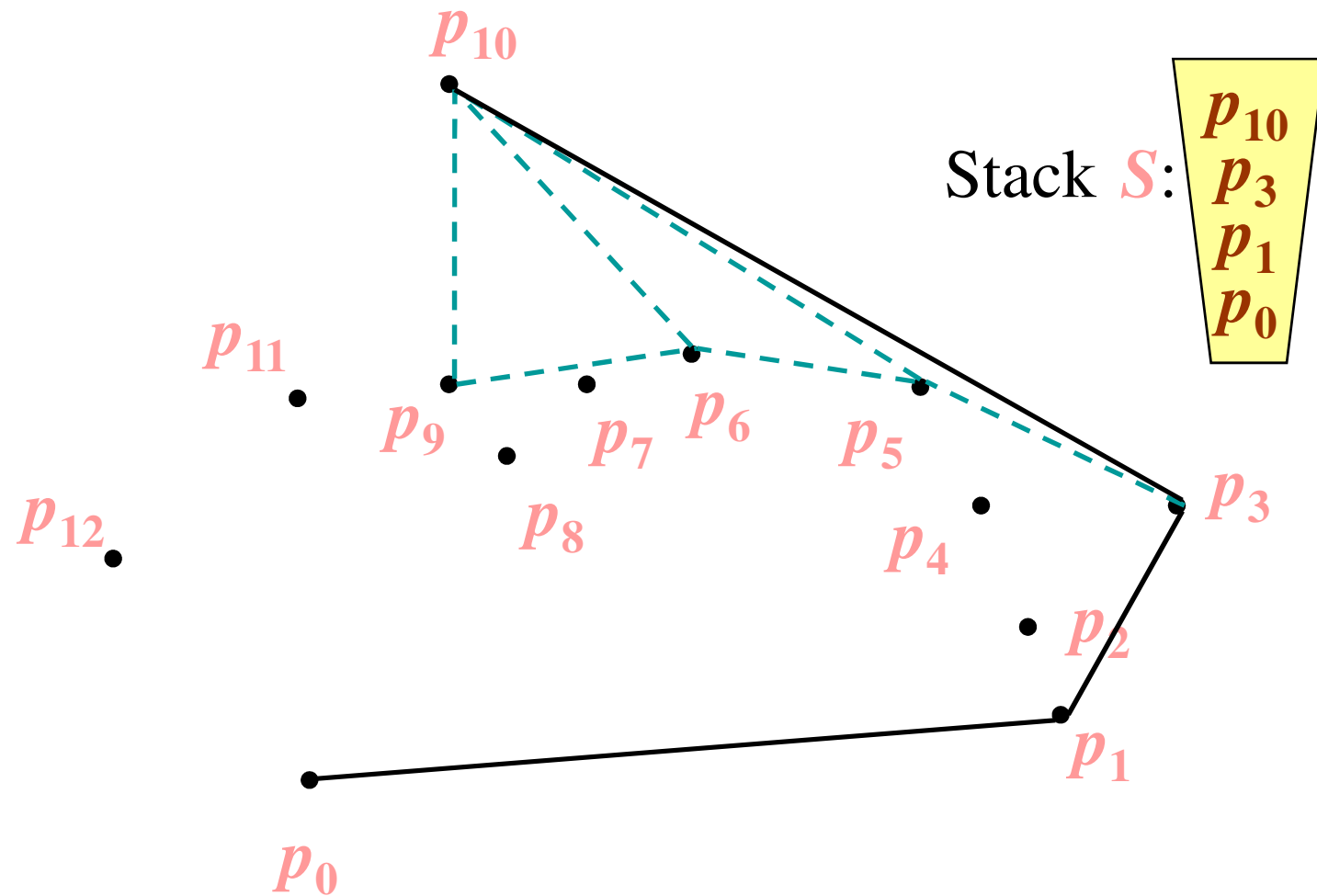
# Graham-Scan (6/11)



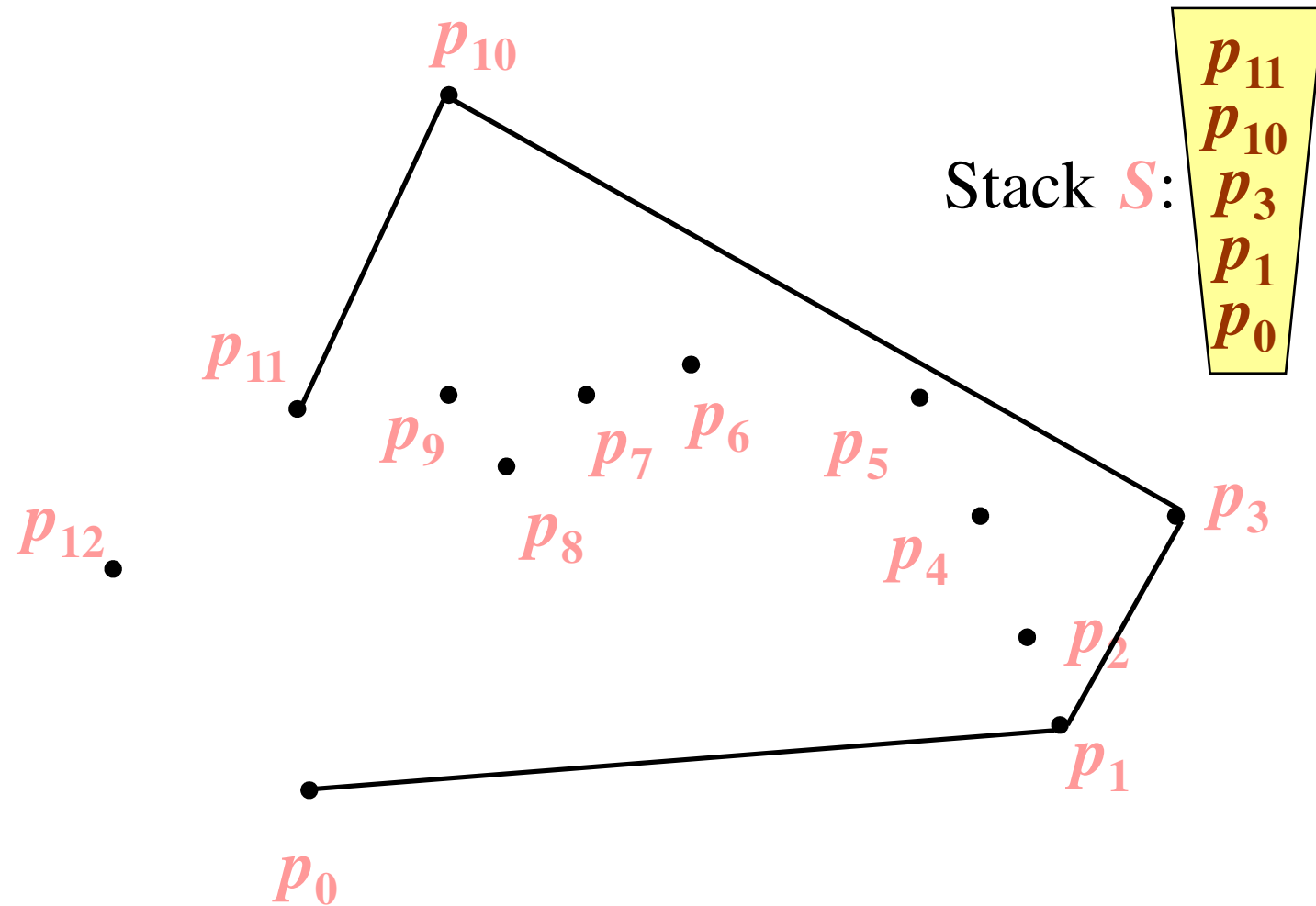
# Graham-Scan (7/11)



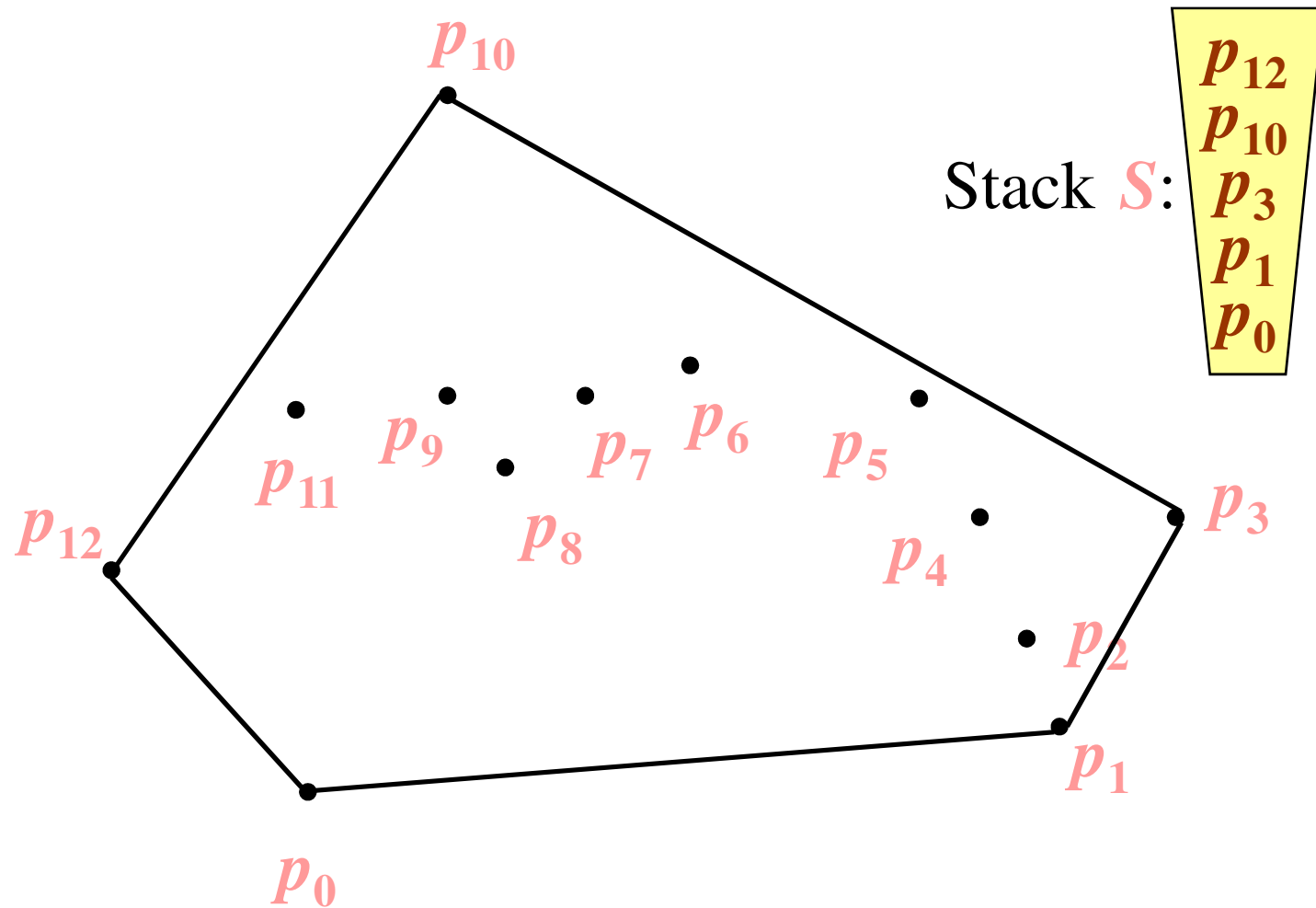
# Graham-Scan (8/11)



# Graham-Scan (9/11)



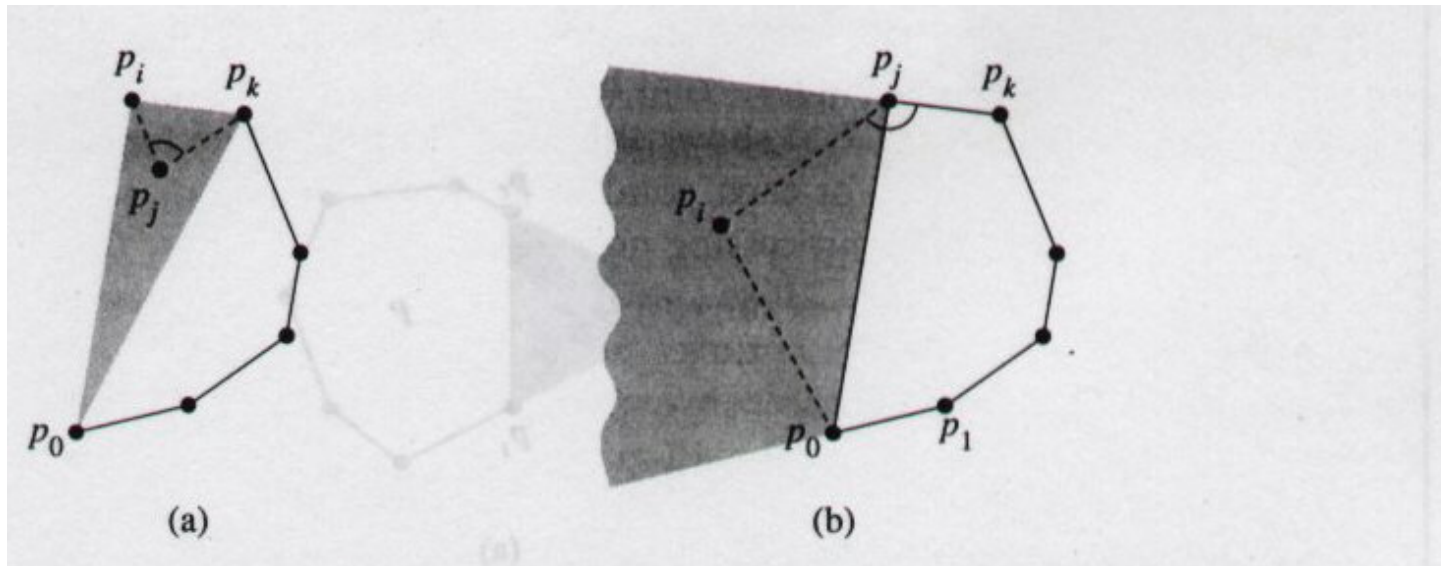
# Graham-Scan (10/11)



# Time complexity Analysis

- Graham-Scan
  - Sorting in step 2 needs  $O(n \log n)$ .
  - Time complexity of stack operation is  $O(2n)$
  - The overall time complexity in **Graham-Scan** is  $O(n \log n)$ .
- Demo:
  - <http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html>

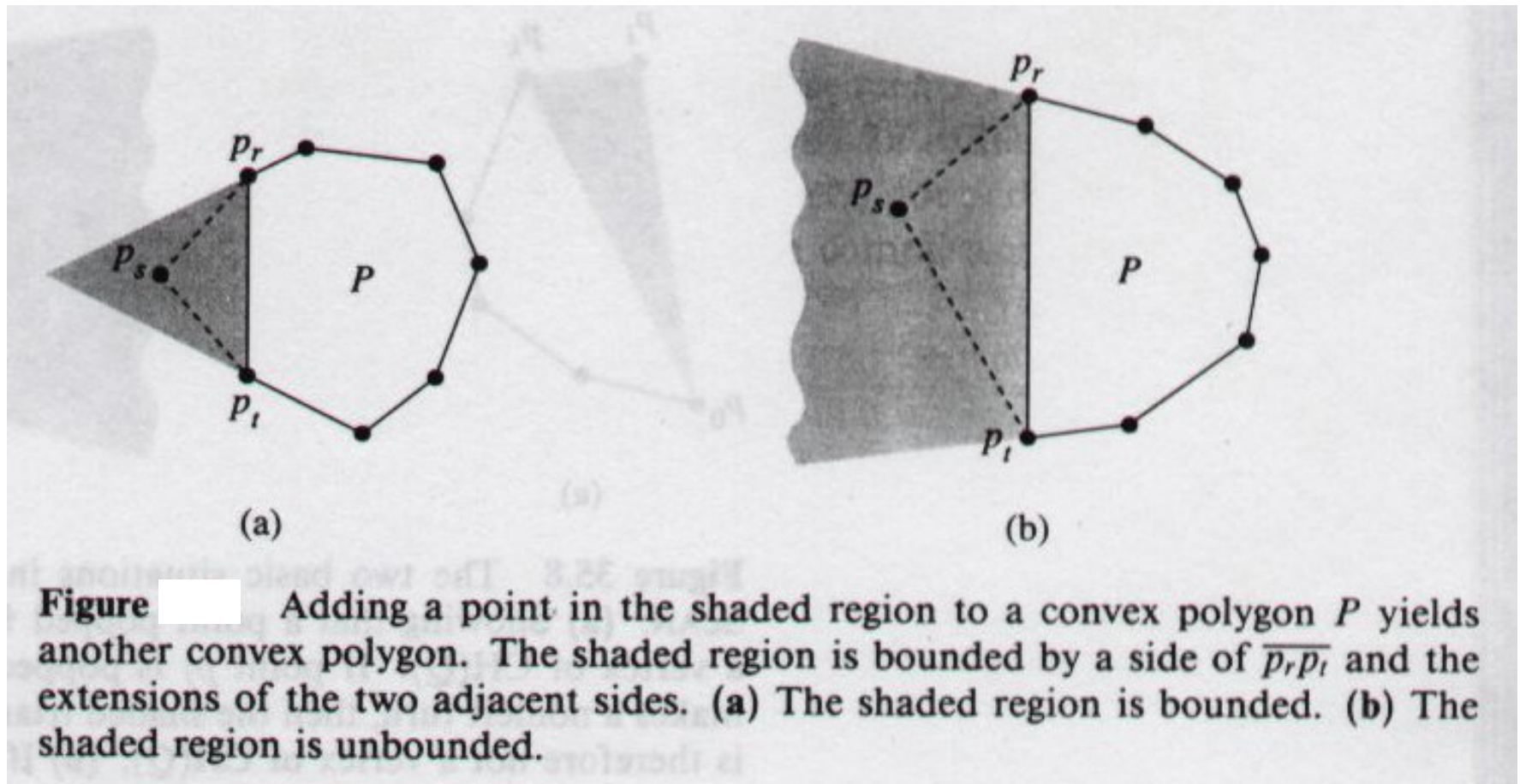
# Graham Scan



**Figure** The two basic situations in the proof of correctness of GRAHAM-SCAN. (a) Showing that a point popped from the stack in GRAHAM-SCAN is not a vertex of  $CH(Q)$ . If point  $p_j$  is popped from the stack because angle  $\angle p_k p_j p_i$  makes a nonleft turn, then the shaded triangle  $\triangle p_0 p_k p_i$  contains point  $p_j$ . Point  $p_j$  is therefore not a vertex of  $CH(Q)$ . (b) If point  $p_i$  is pushed onto the stack, then there must be a left turn at angle  $\angle p_k p_j p_i$ . Because  $p_i$  follows  $p_j$  in the polar-angle ordering of points and because of how  $p_0$  was chosen,  $p_i$  must be in the shaded region. If the points on the stack form a convex polygon before the push, then they must form a convex polygon afterward.



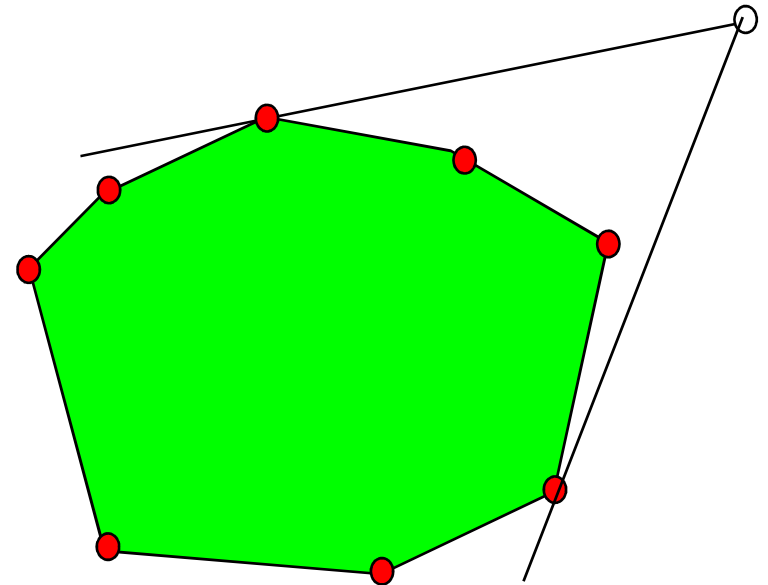
# Graham Scan



# Algorithms: 2D Incremental

source: O'Rourke, *Computational Geometry in C*

- Add points, one at a time
  - update hull for each new point
- Key step becomes adding a single point to an existing hull.
  - Find 2 tangents
    - Results of 2 consecutive LEFT tests differ
- Idea can be extended to 3D.



**Algorithm:** INCREMENTAL ALGORITHM

Let  $H_2 \leftarrow \text{ConvexHull}\{p_0, p_1, p_2\}$

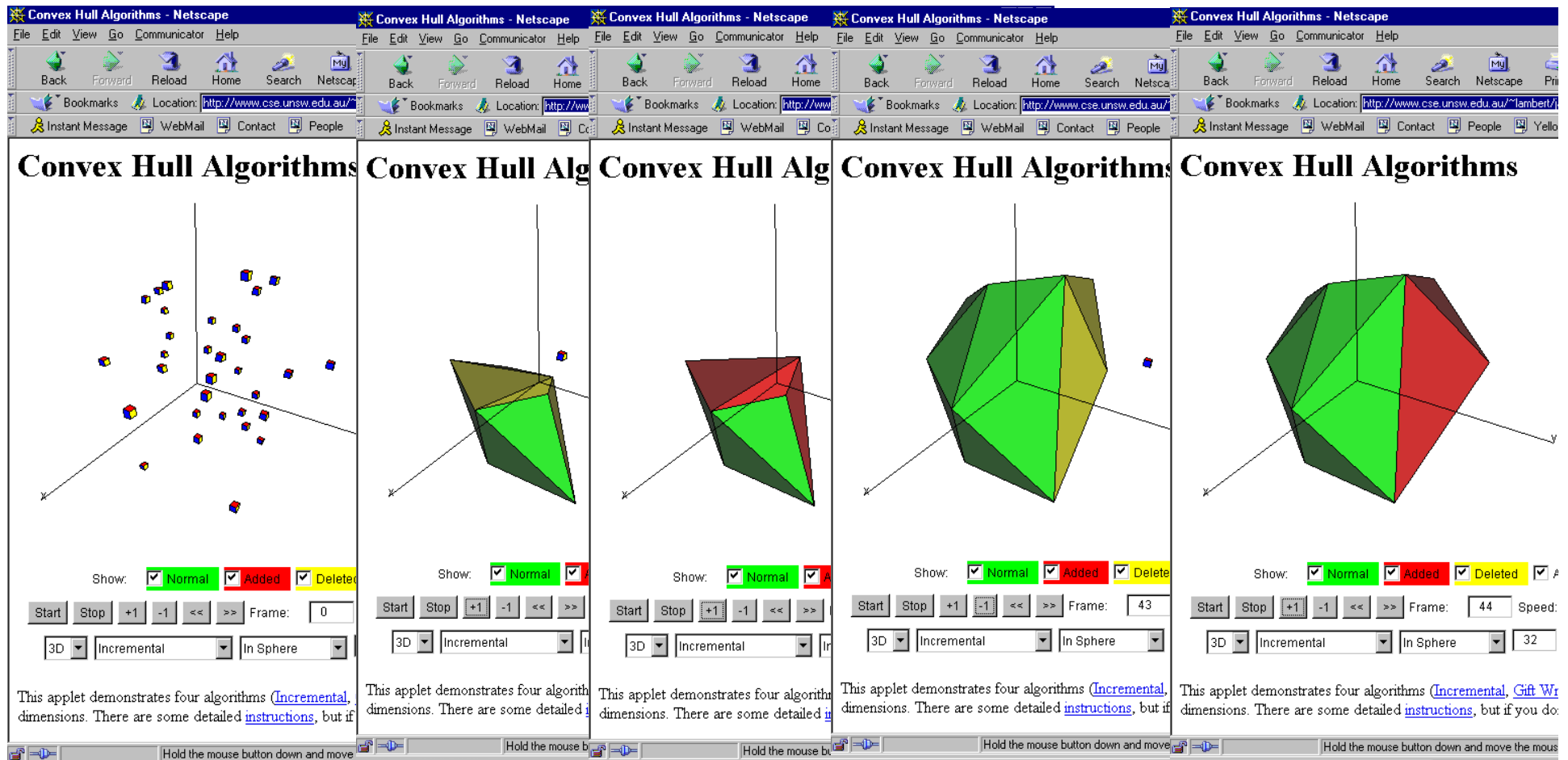
for  $k \leftarrow 3$  to  $n - 1$  do

$H_k \leftarrow \text{ConvexHull}\{H_{k-1} \cup p_k\}$

$O(n^2)$

*can be improved to  $O(n \lg n)$*

# Algorithms: 3D Incremental



$O(n^2)$  time

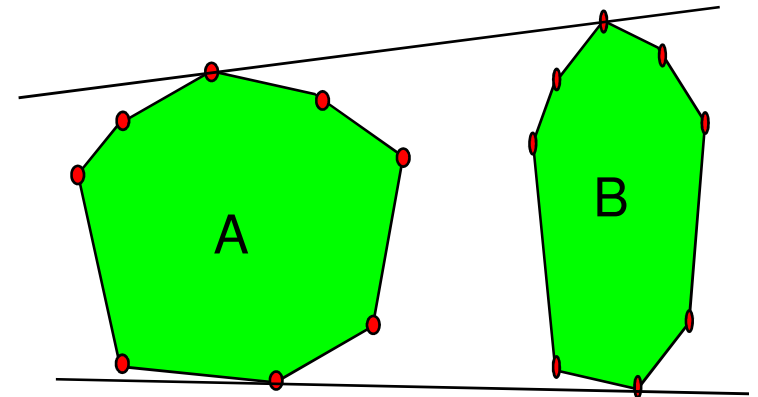
*CxHull Animations:* <http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html>

# Algorithms:

## 2D Divide-and-Conquer

*source: O'Rourke, Computational Geometry in C*

- Divide-and-Conquer in a geometric setting
- $O(n)$  merge step is the challenge
  - Find upper and lower tangents
  - Lower tangent: find rightmost pt of A & leftmost pt of B; then “walk it downwards”
- Idea can be extended to 3D.



### Algorithm: DIVIDE-and-CONQUER

Sort points by x coordinate

Divide points into 2 sets A and B:

A contains left  $\lceil n/2 \rceil$  points

B contains right  $\lfloor n/2 \rfloor$  points

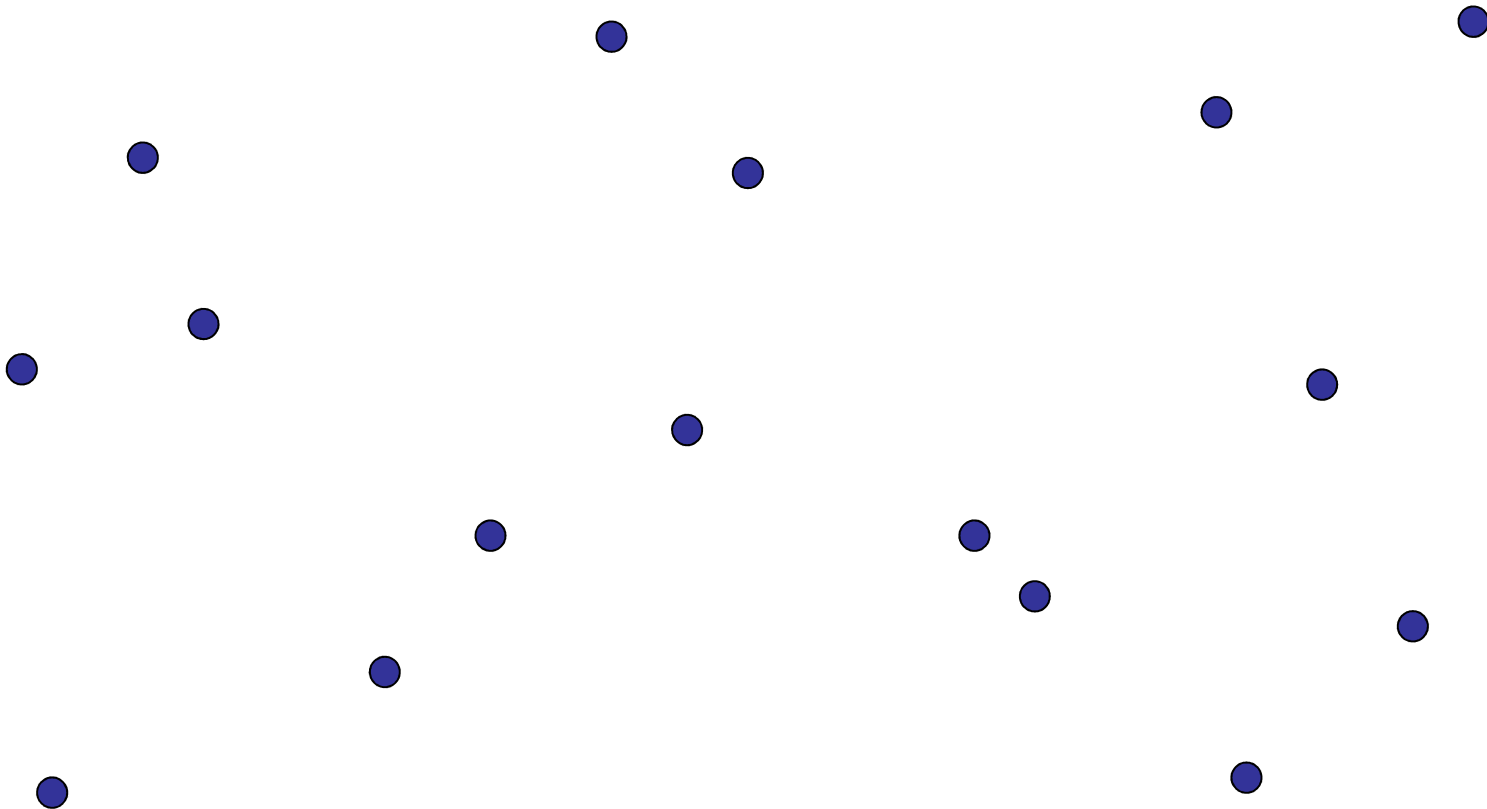
Compute ConvexHull(A) and ConvexHull(B) recursively

Merge ConvexHull(A) and ConvexHull(B)

**$O(n \lg n)$**

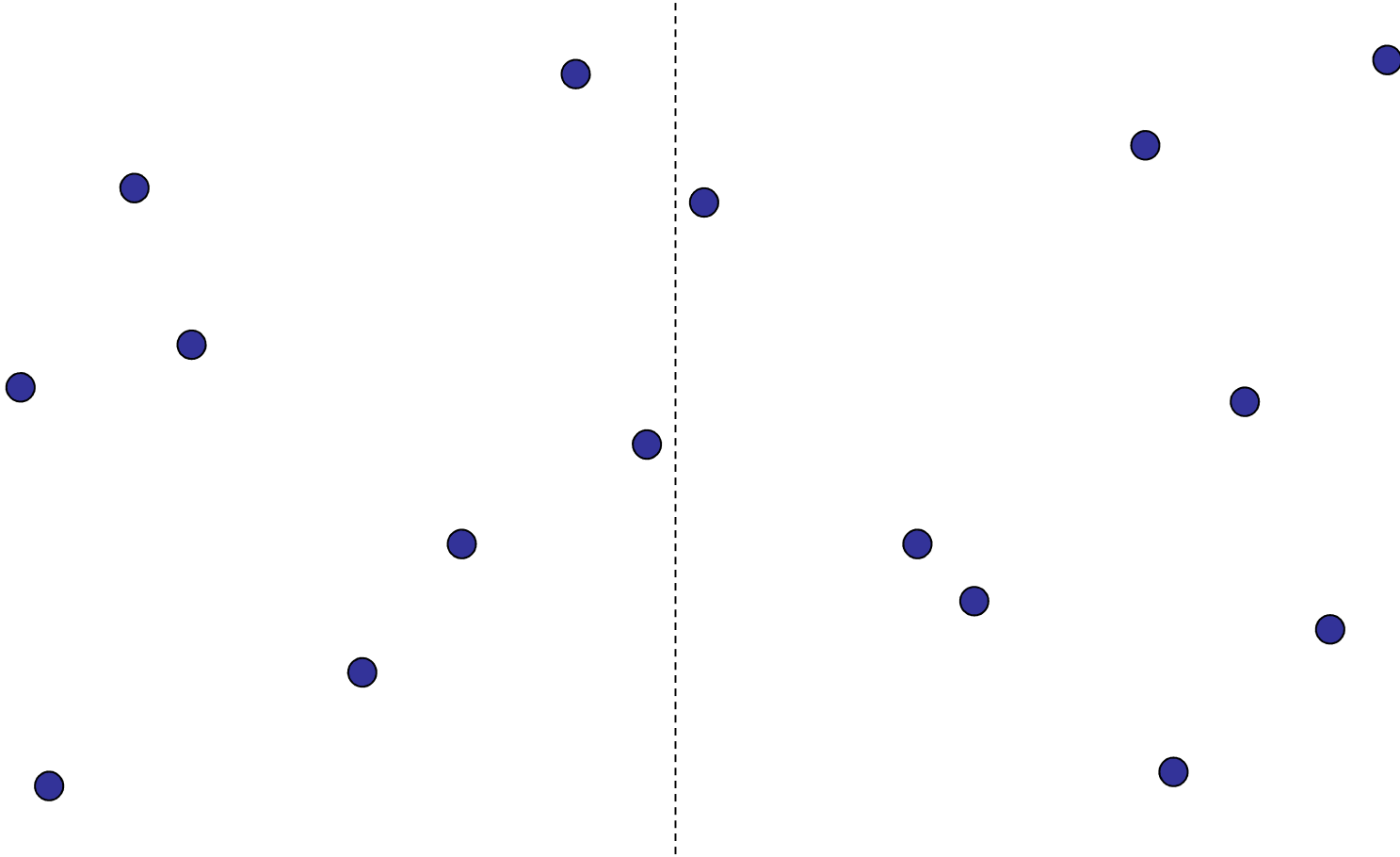
# Convex Hull – Divide & Conquer

- Split set into two, compute convex hull of both, combine.



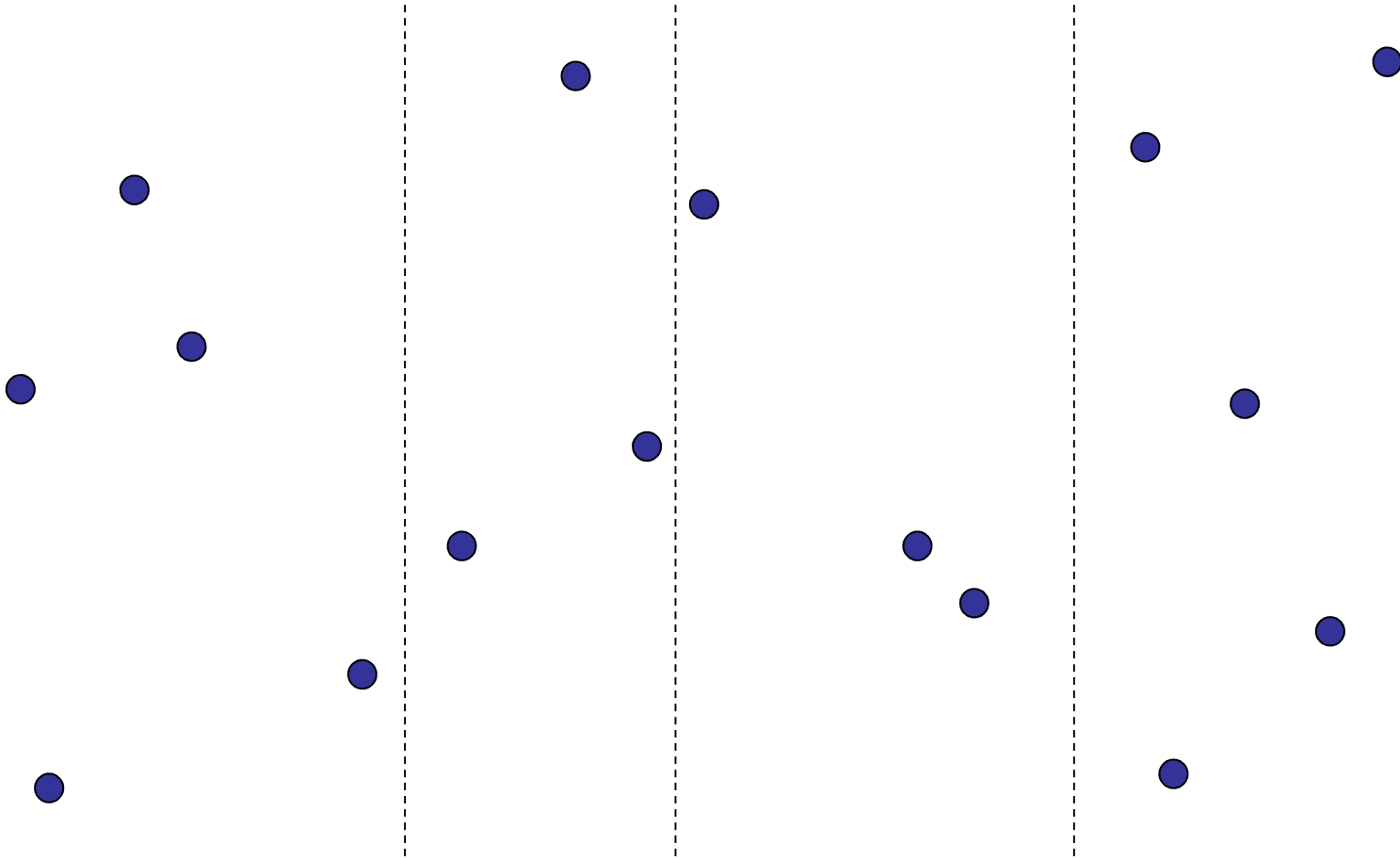
# Convex Hull – Divide & Conquer

- Split set into two, compute convex hull of both, combine.



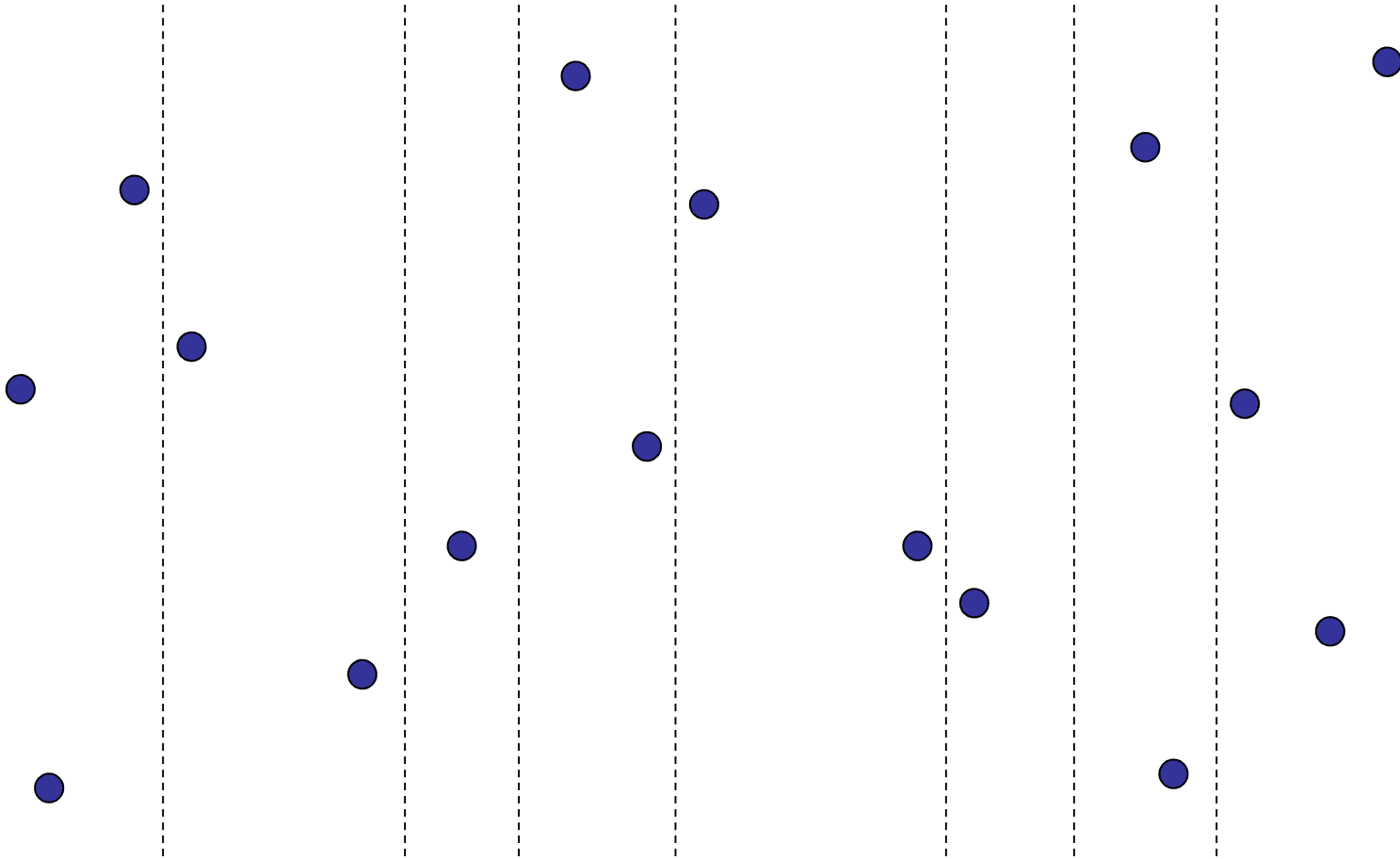
# Convex Hull – Divide & Conquer

- Split set into two, compute convex hull of both, combine.



# Convex Hull – Divide & Conquer

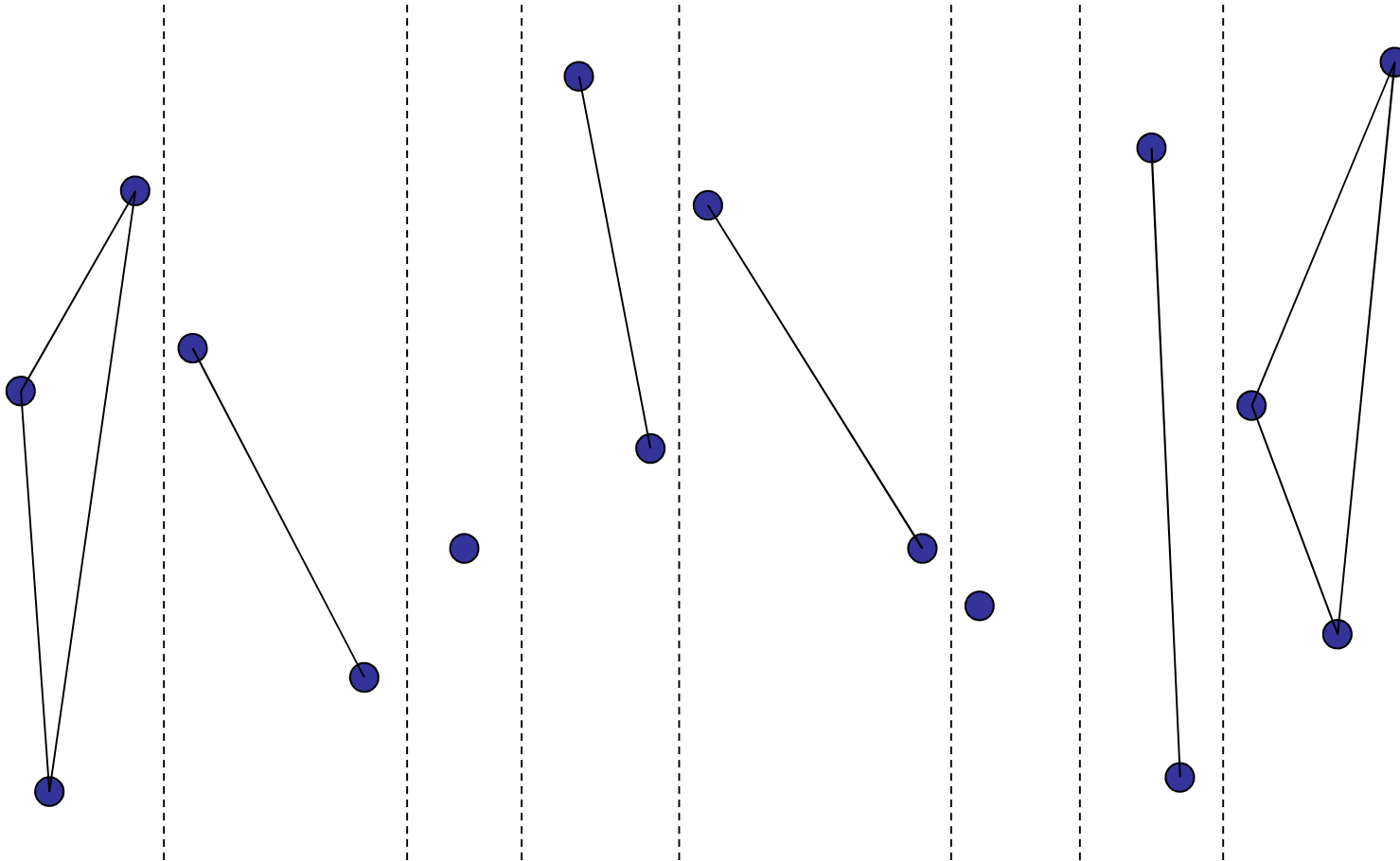
- Split set into two, compute convex hull of both, combine.





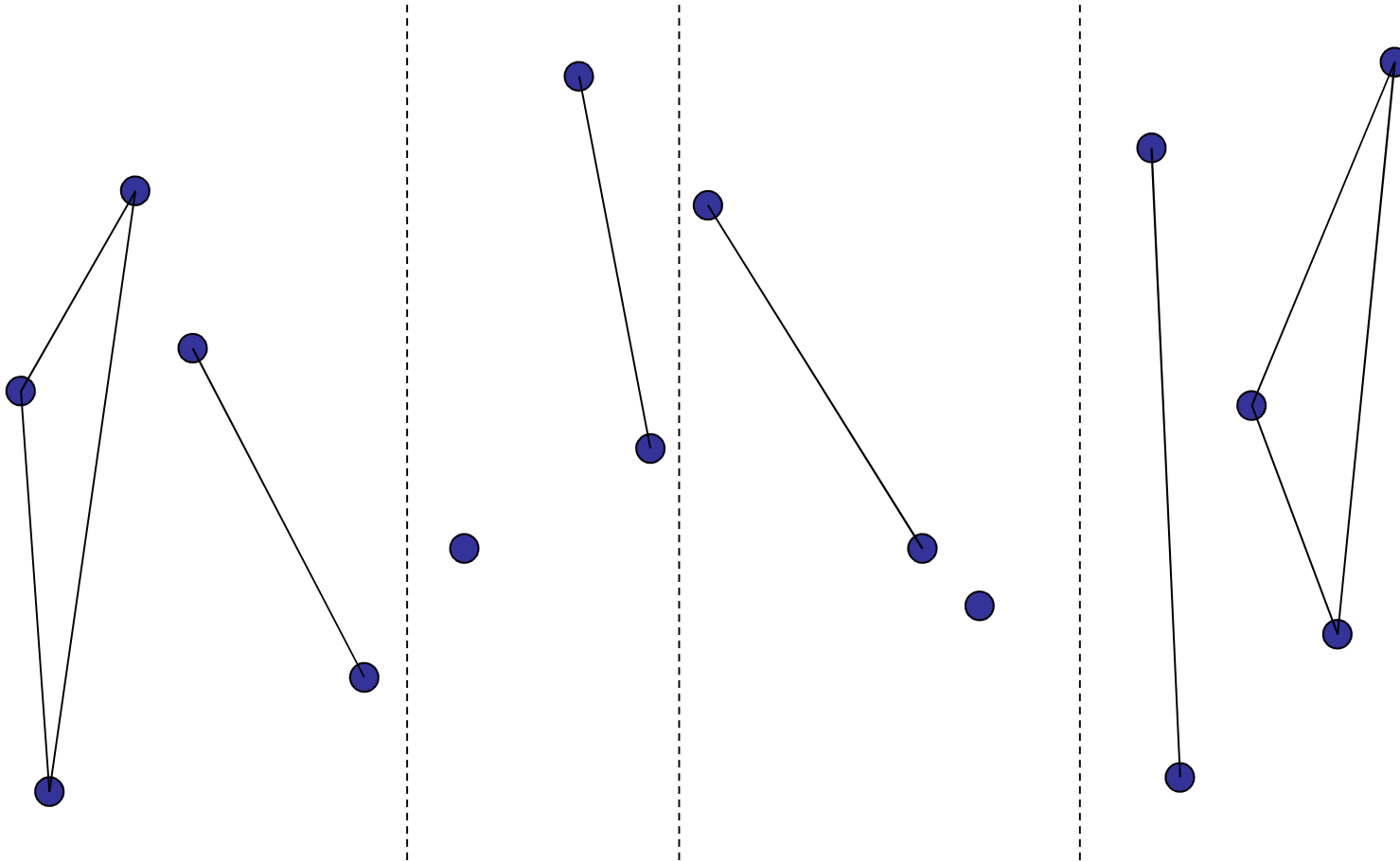
# Convex Hull – Divide & Conquer

- Split set into two, compute convex hull of both, combine.



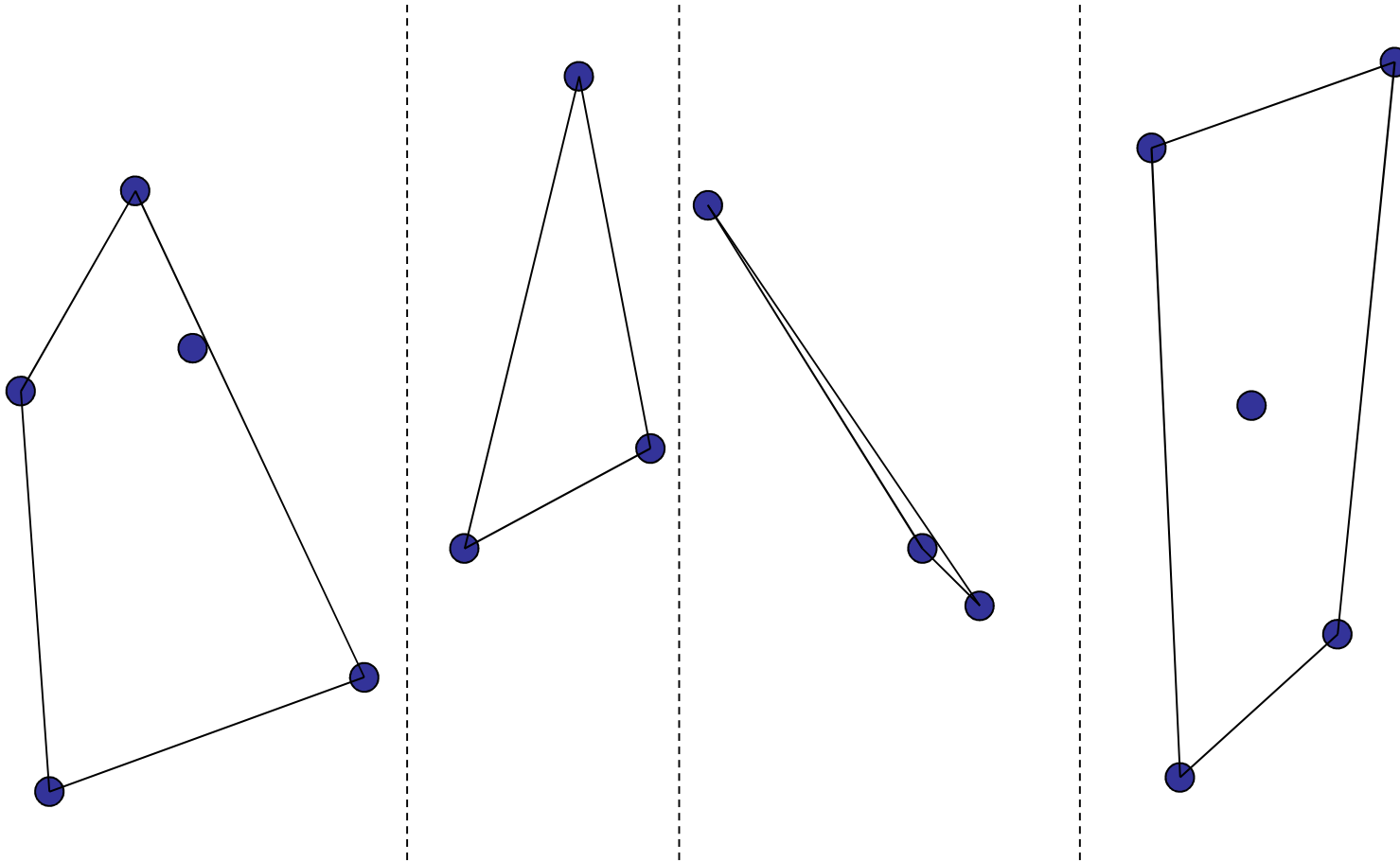
# Convex Hull – Divide & Conquer

- Split set into two, compute convex hull of both, combine.



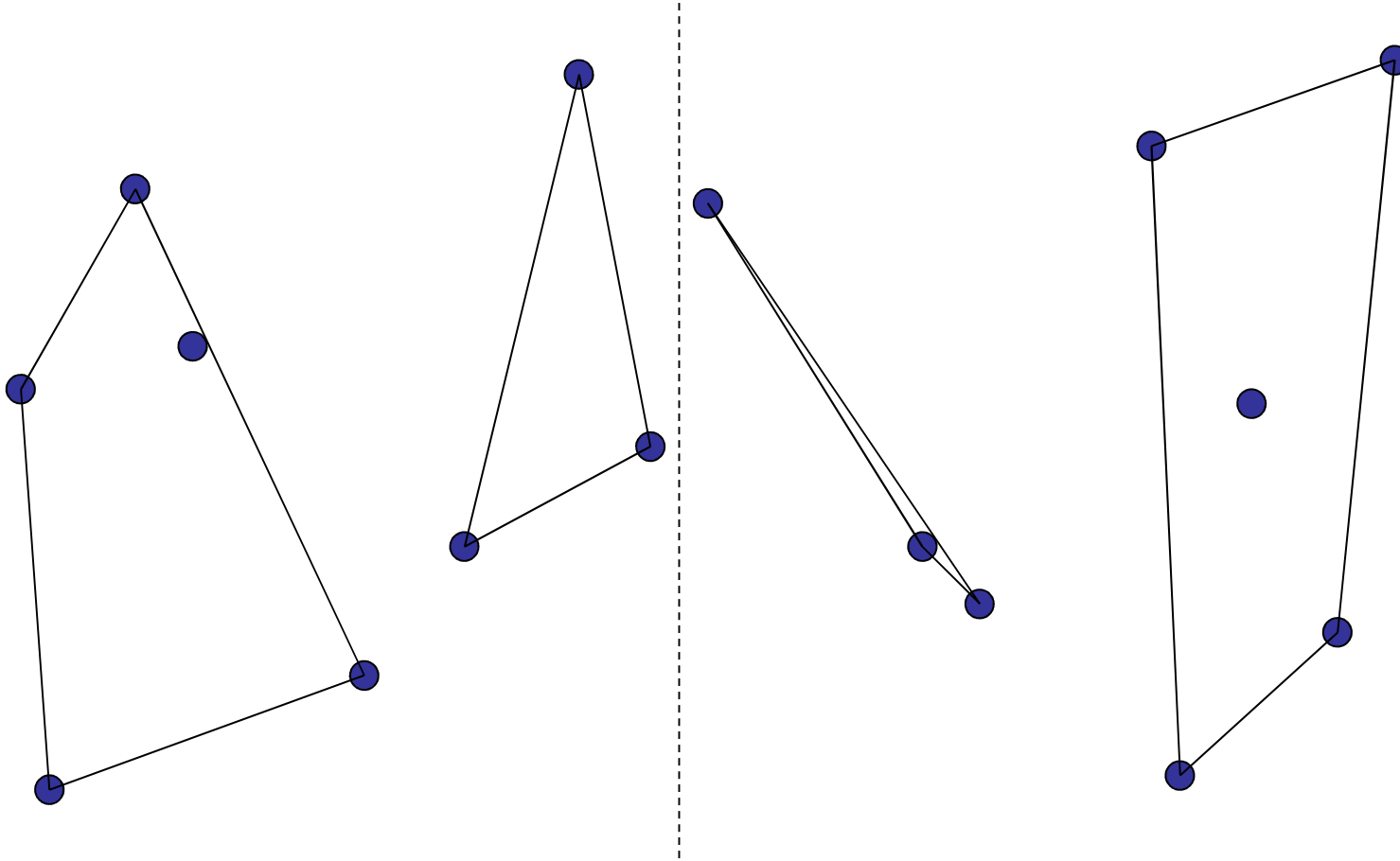
# Convex Hull – Divide & Conquer

- Split set into two, compute convex hull of both, combine.



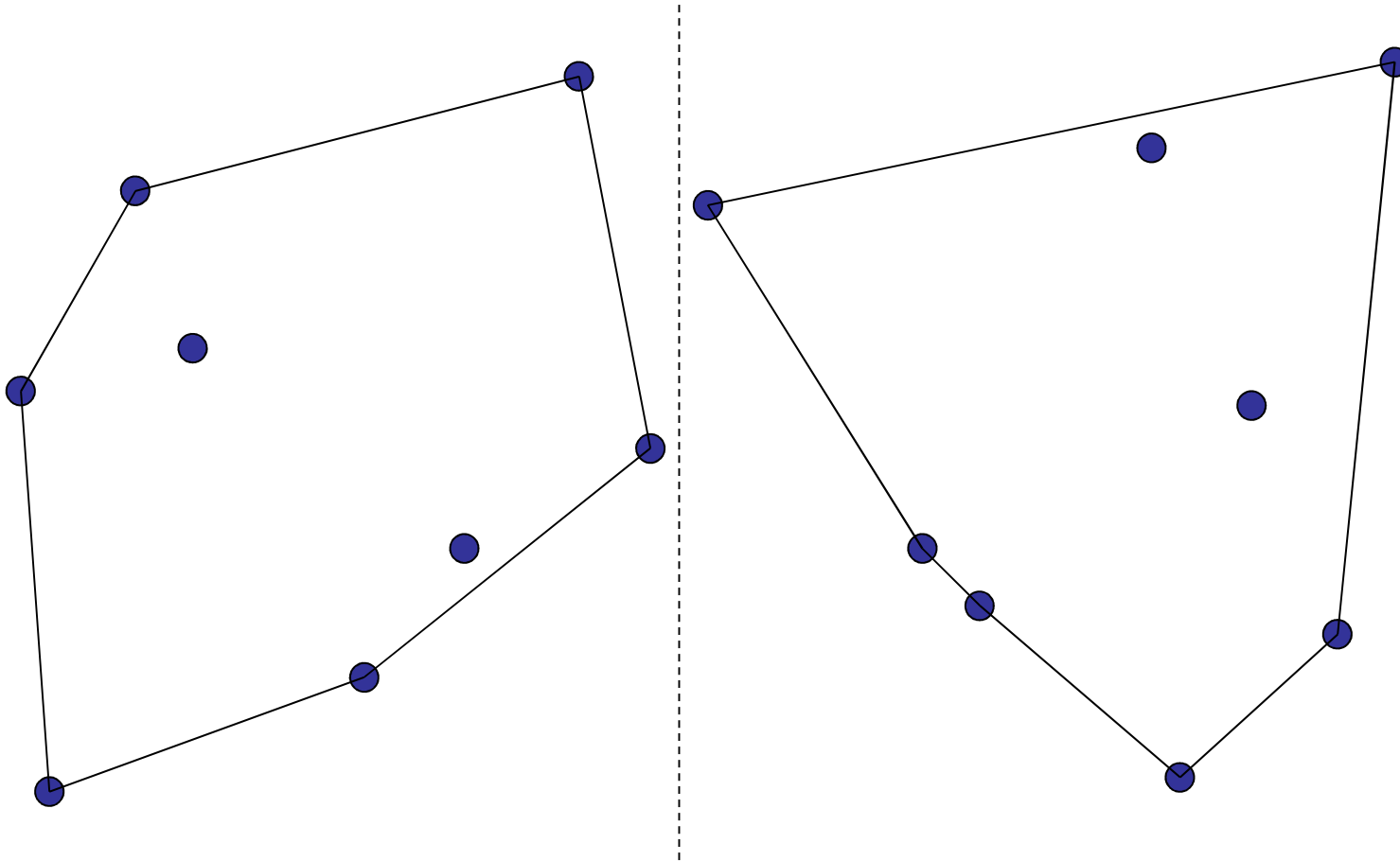
# Convex Hull – Divide & Conquer

- Split set into two, compute convex hull of both, combine.



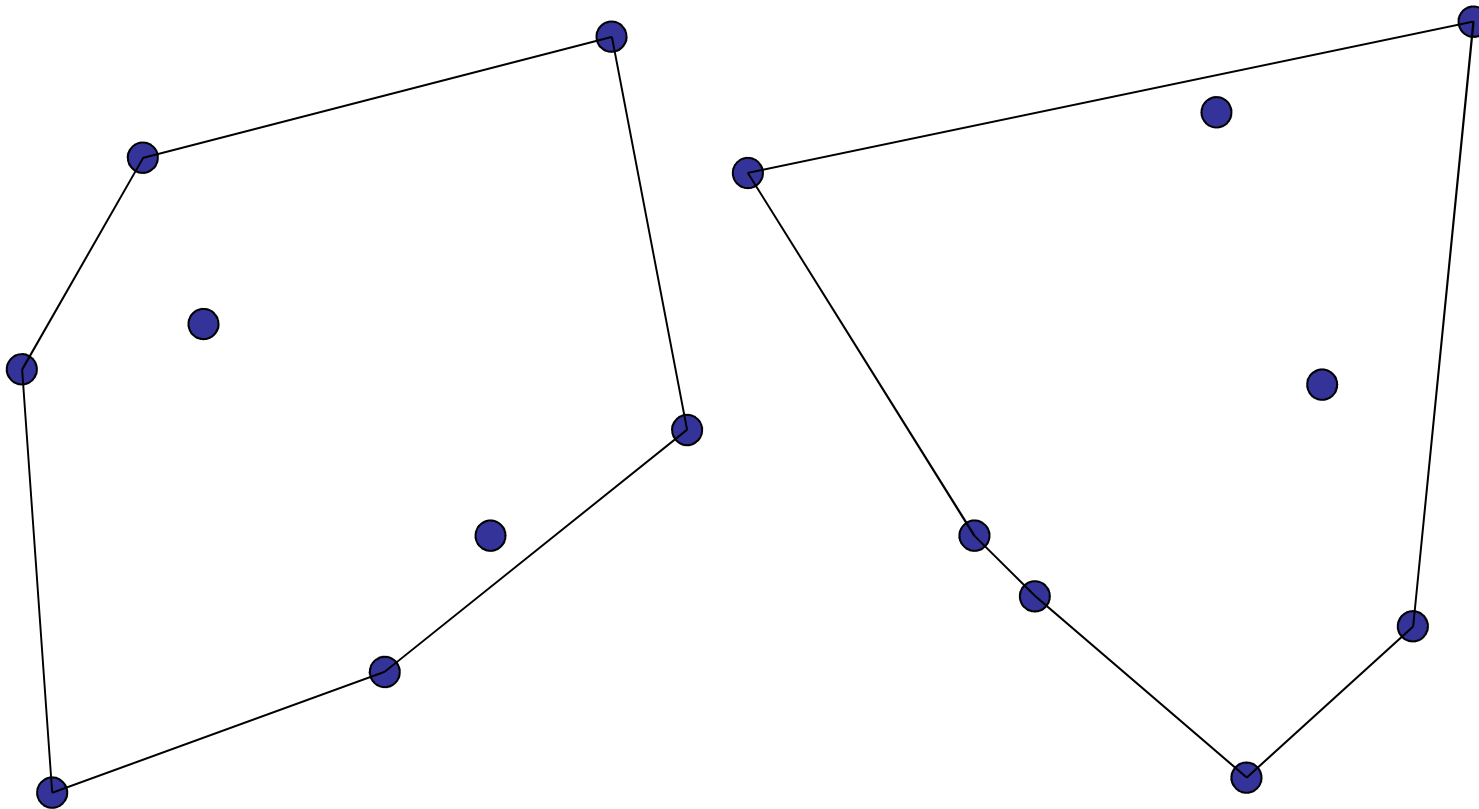
# Convex Hull – Divide & Conquer

- Split set into two, compute convex hull of both, combine.



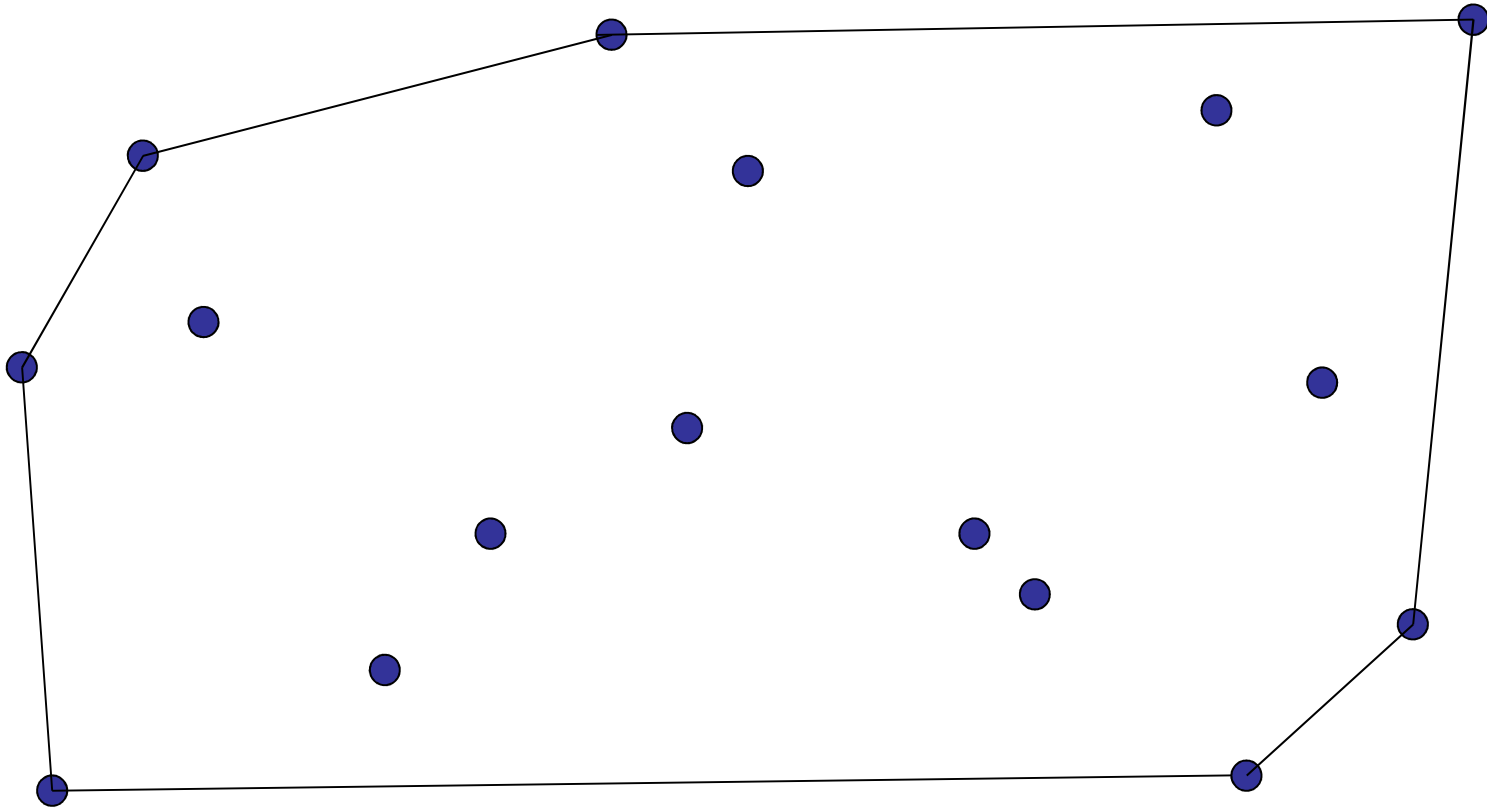
# Convex Hull – Divide & Conquer

- Split set into two, compute convex hull of both, combine.



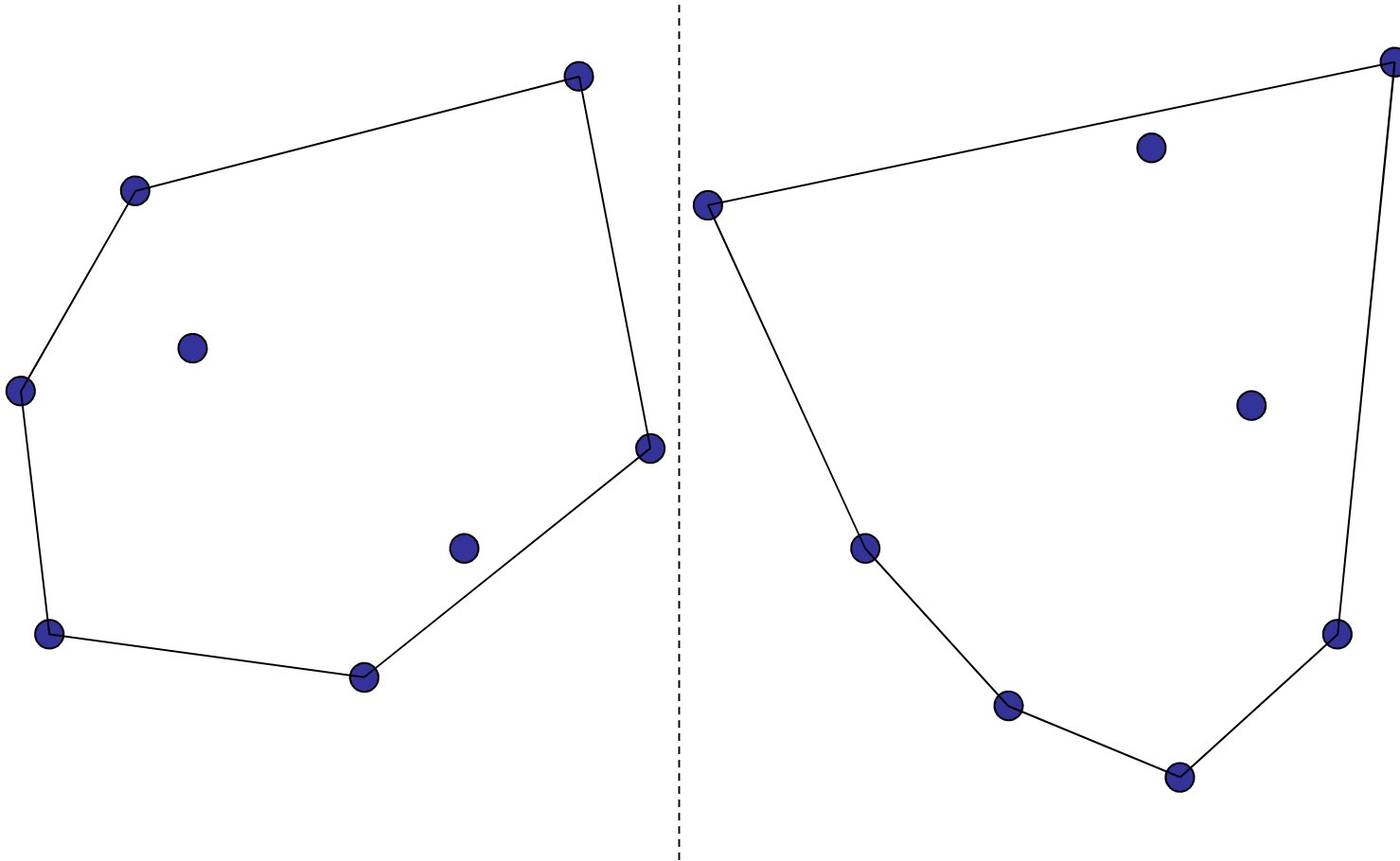
# Convex Hull – Divide & Conquer

- Split set into two, compute convex hull of both, combine.



# Convex Hull – Divide & Conquer

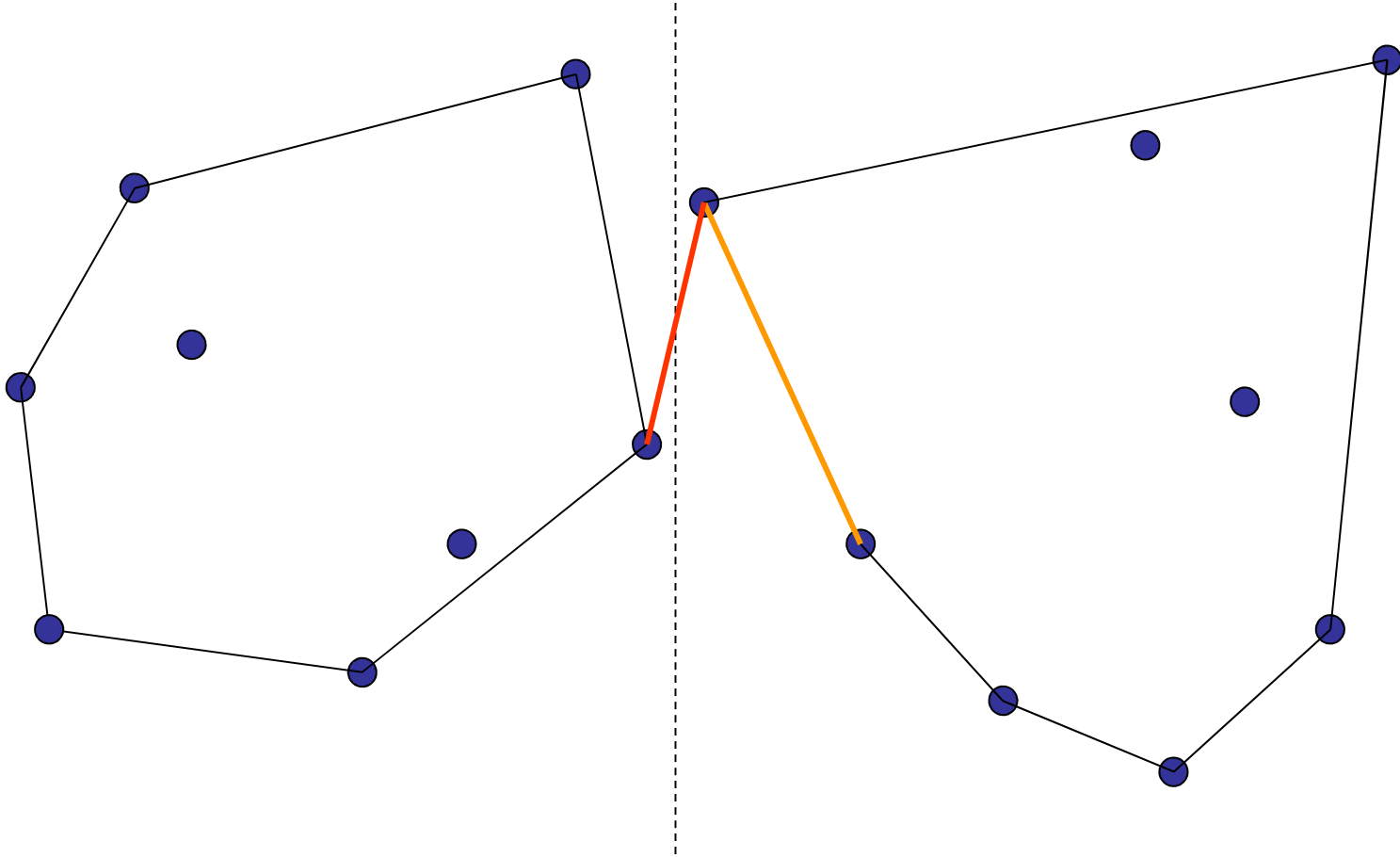
- Merging two convex hulls.





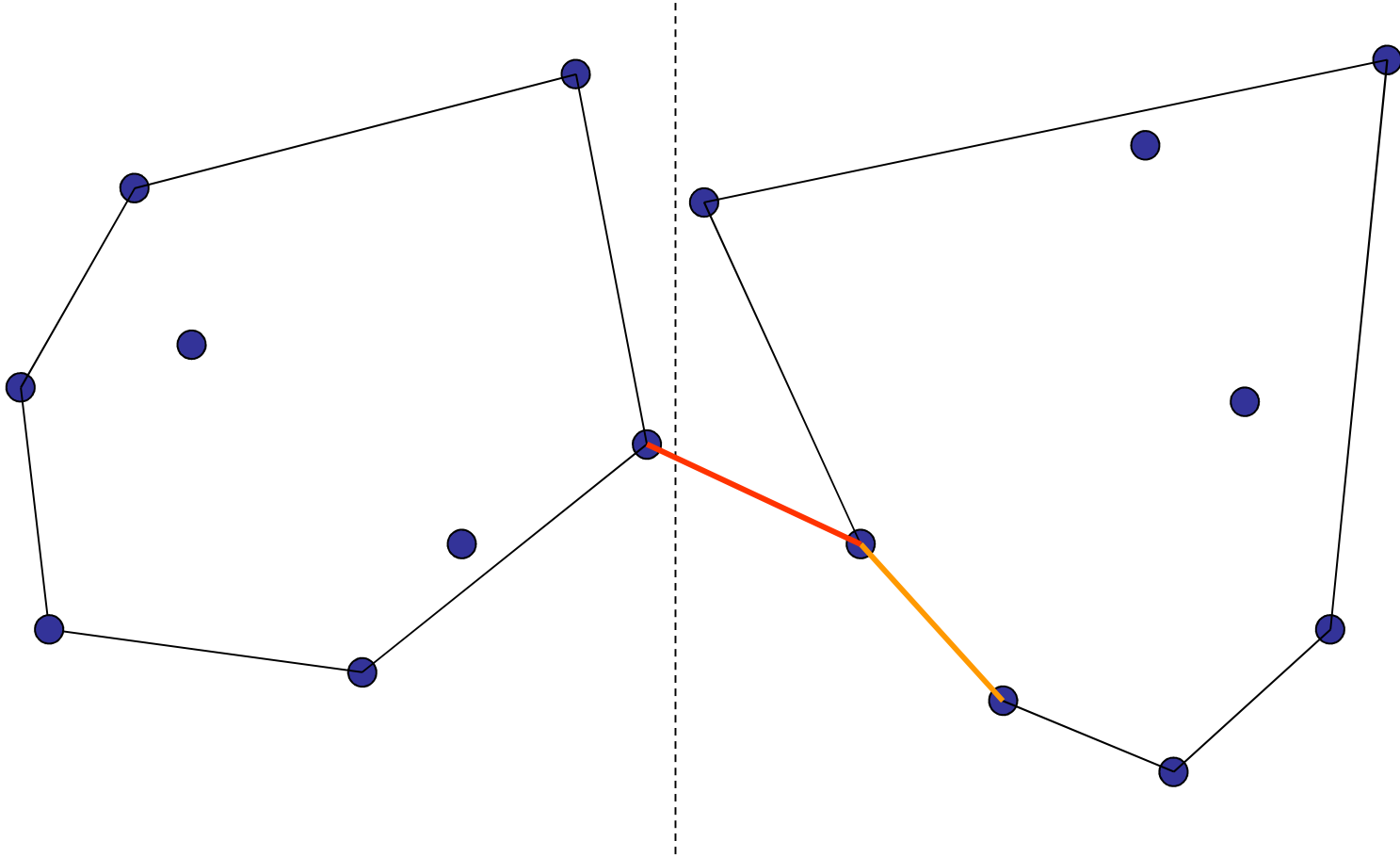
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (i) Find the **lower** tangent.



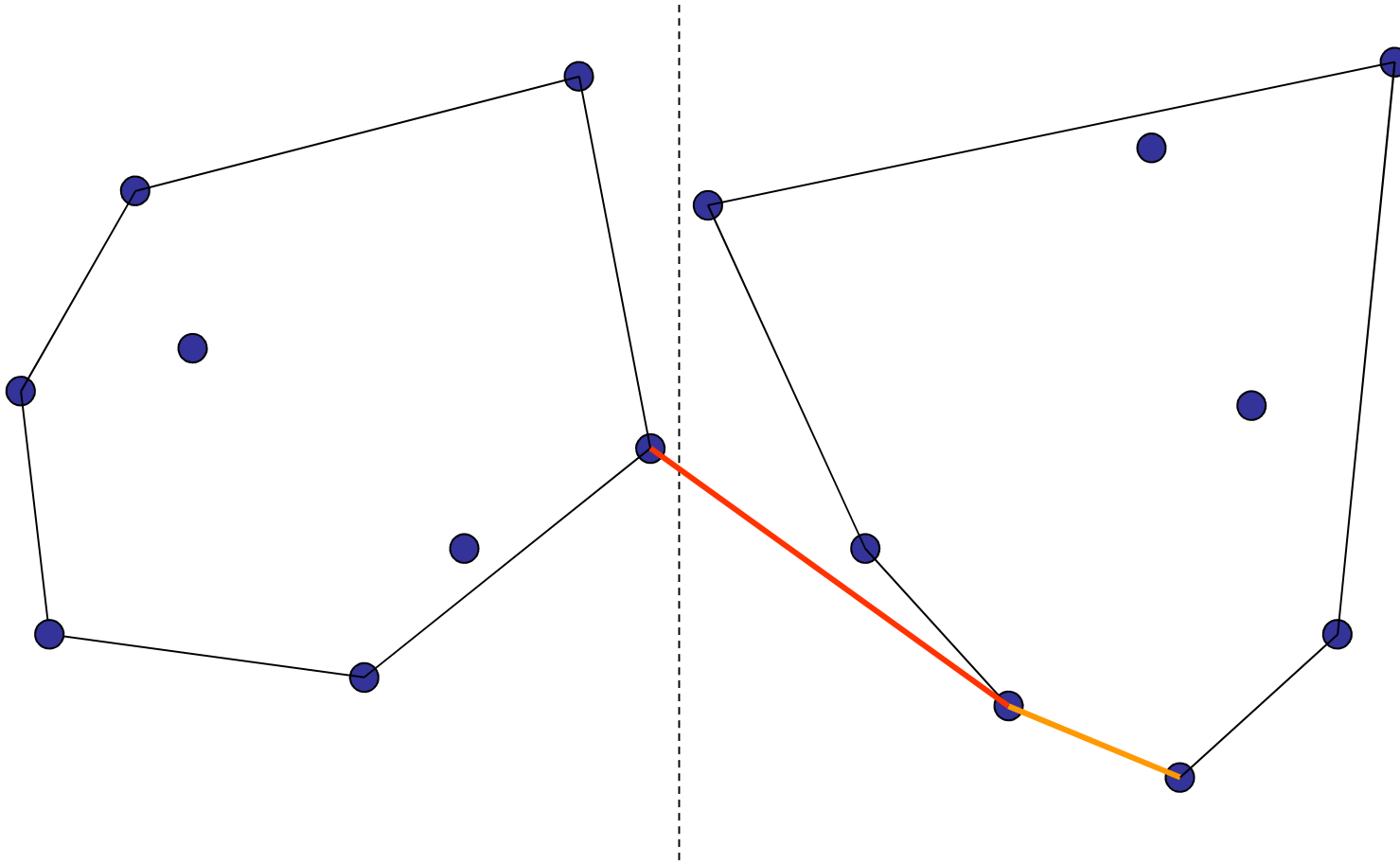
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (i) Find the **lower** tangent.



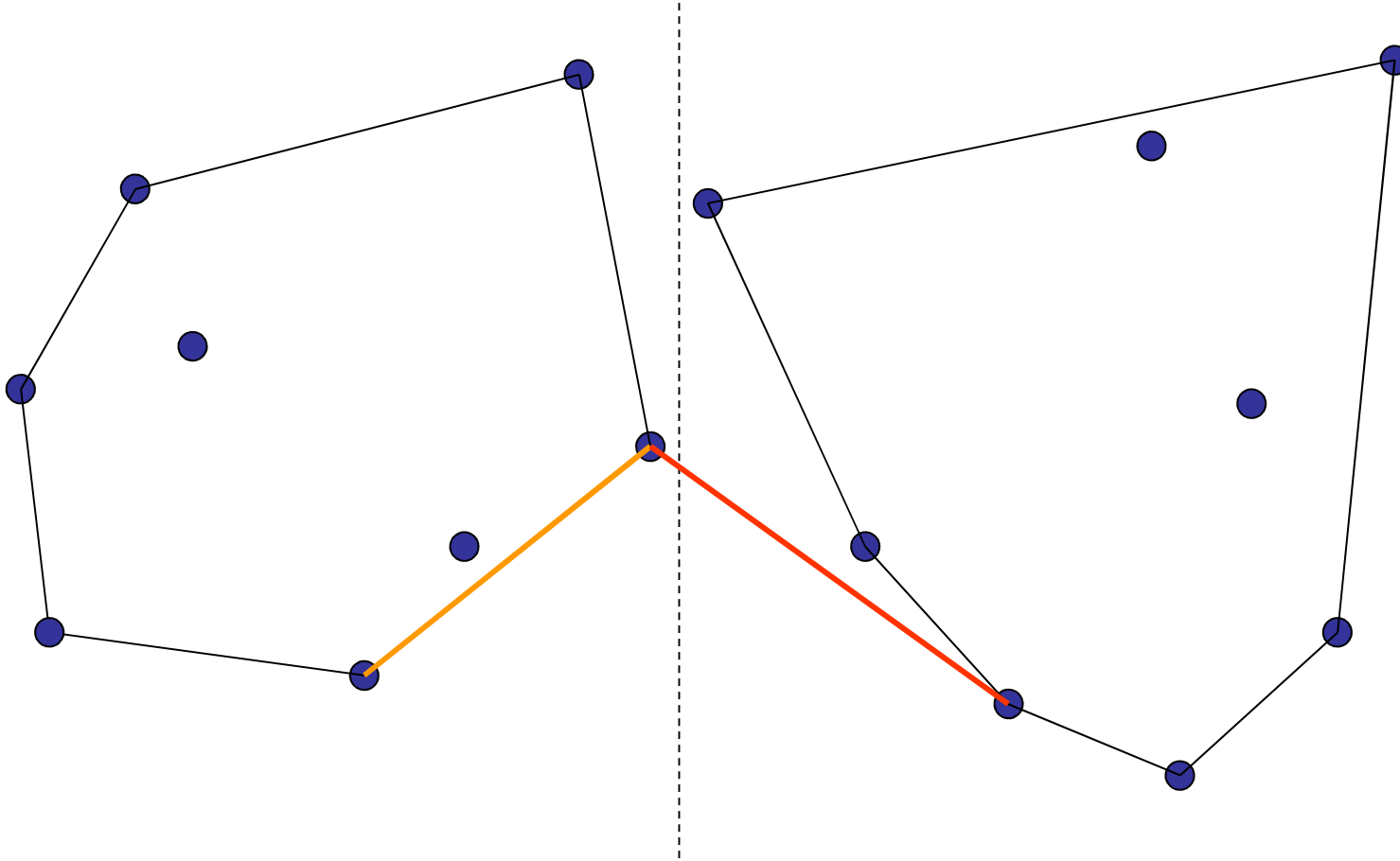
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (i) Find the **lower** tangent.



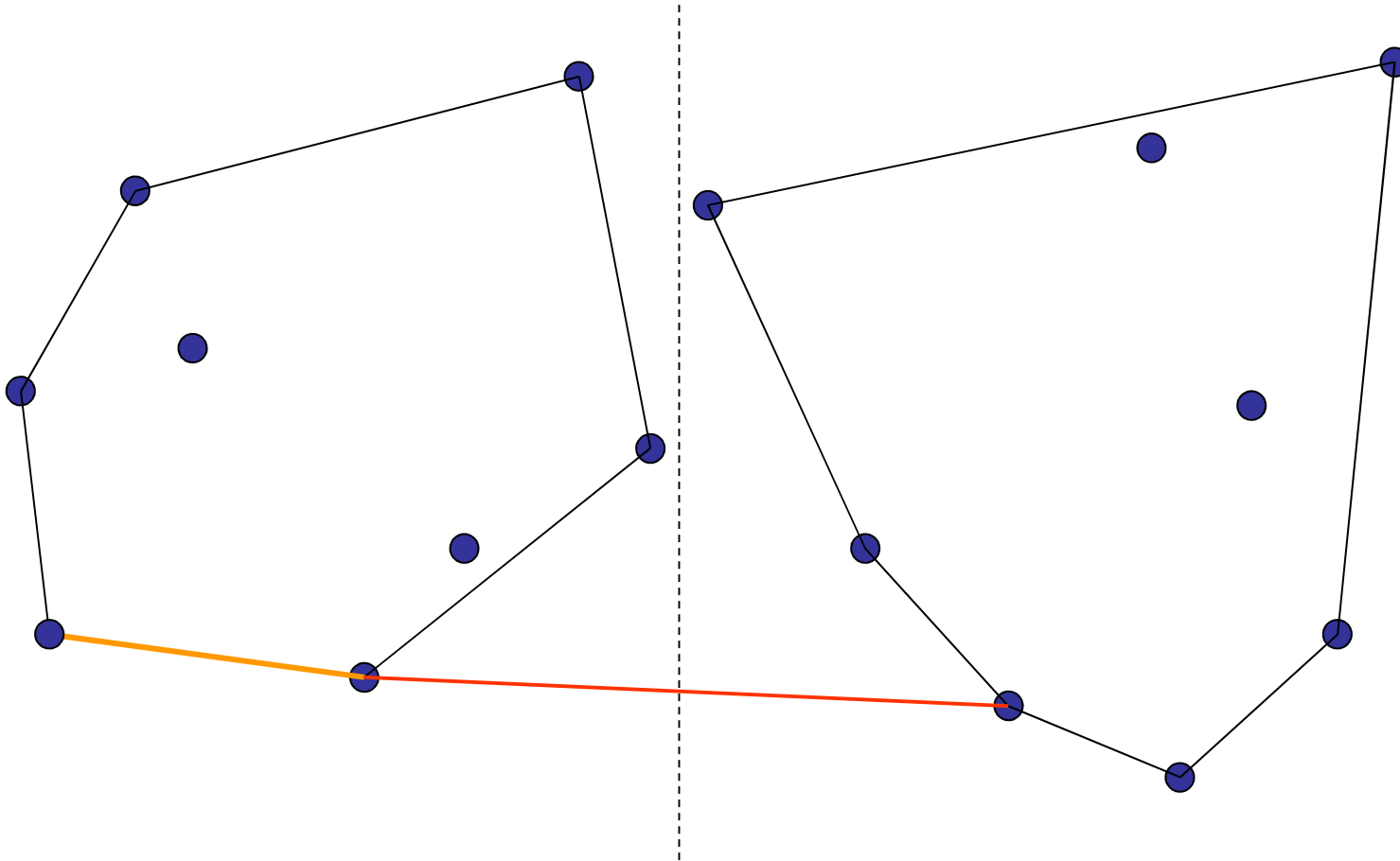
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (i) Find the **lower** tangent.



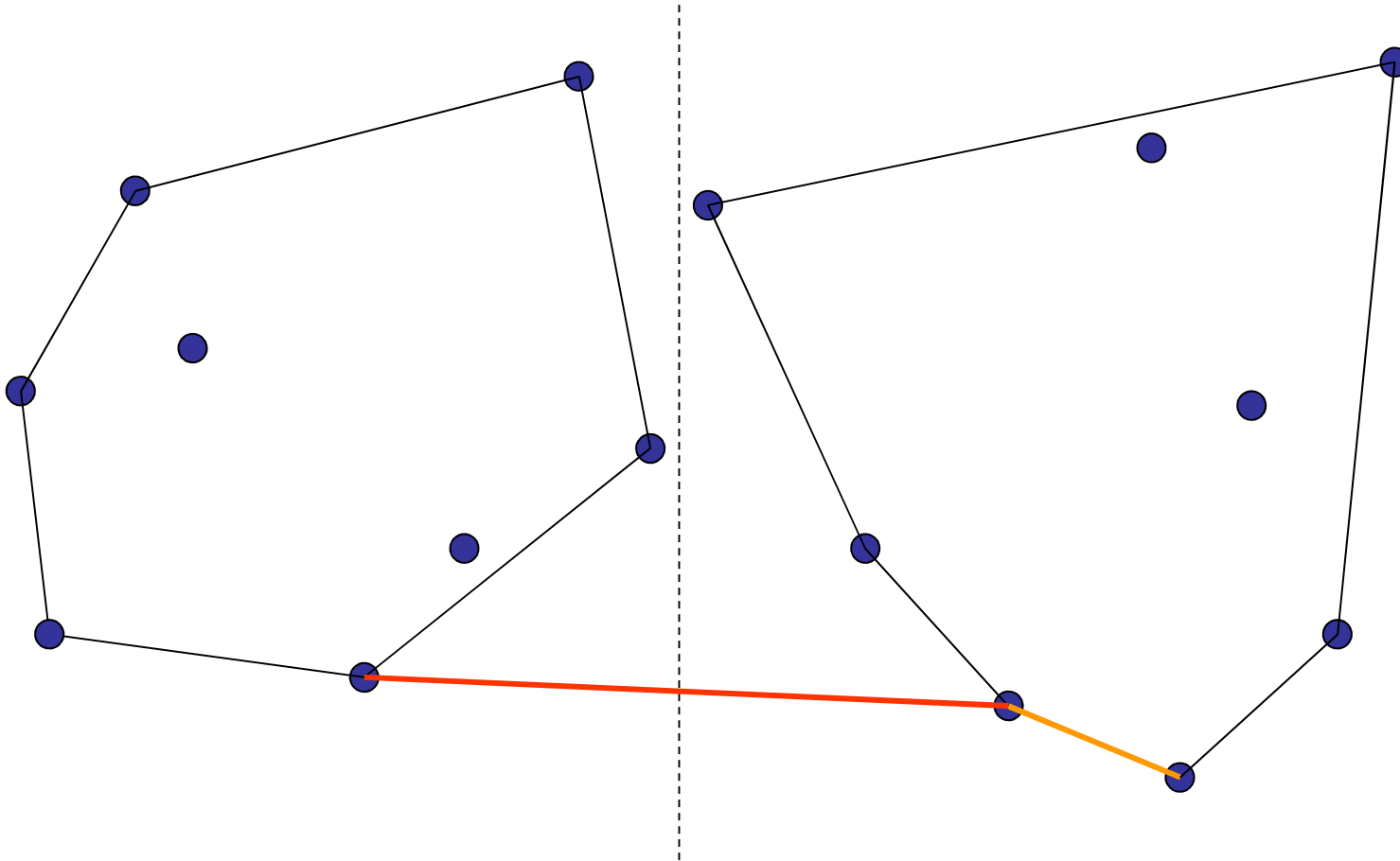
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (i) Find the **lower** tangent.



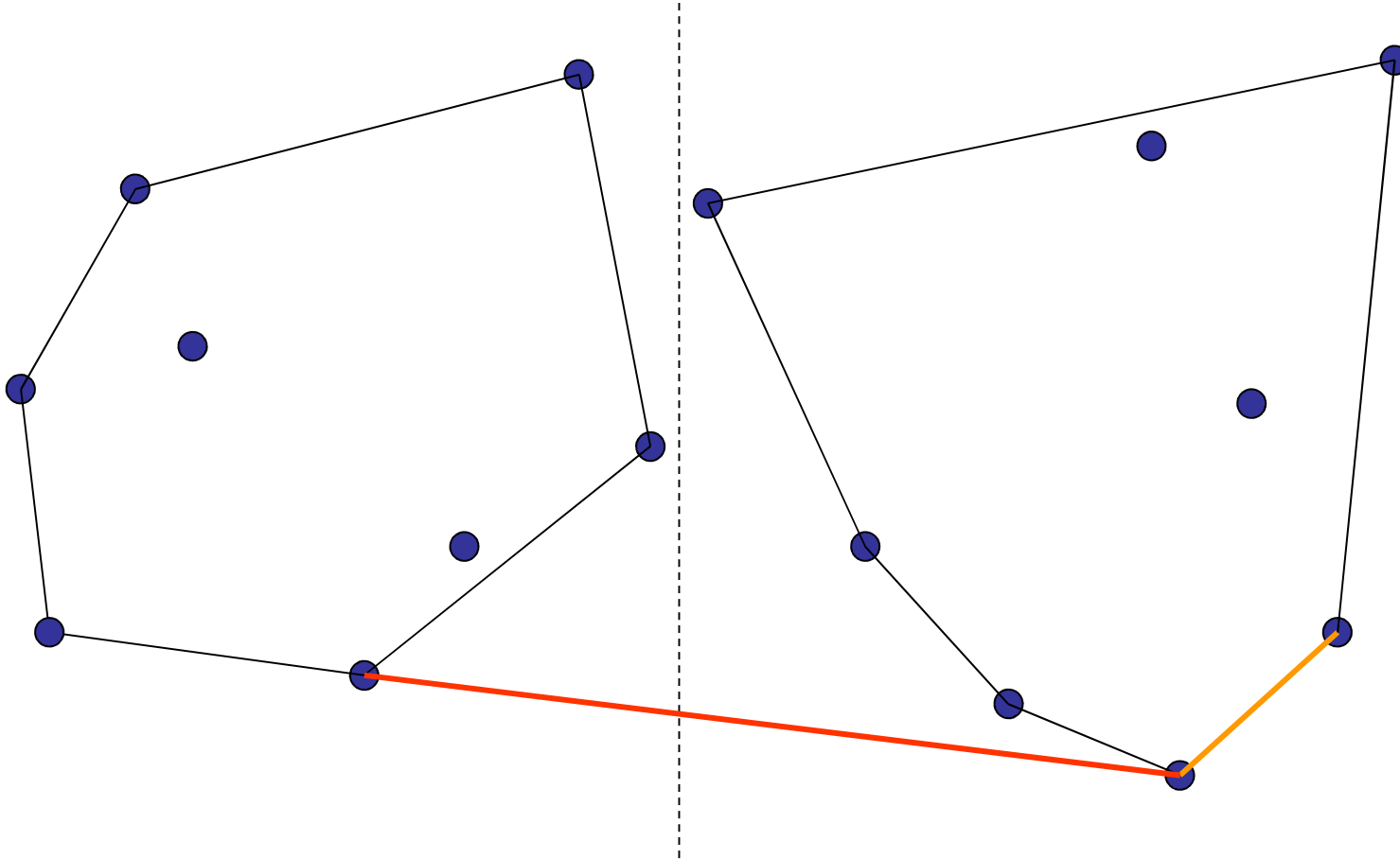
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (i) Find the **lower** tangent.



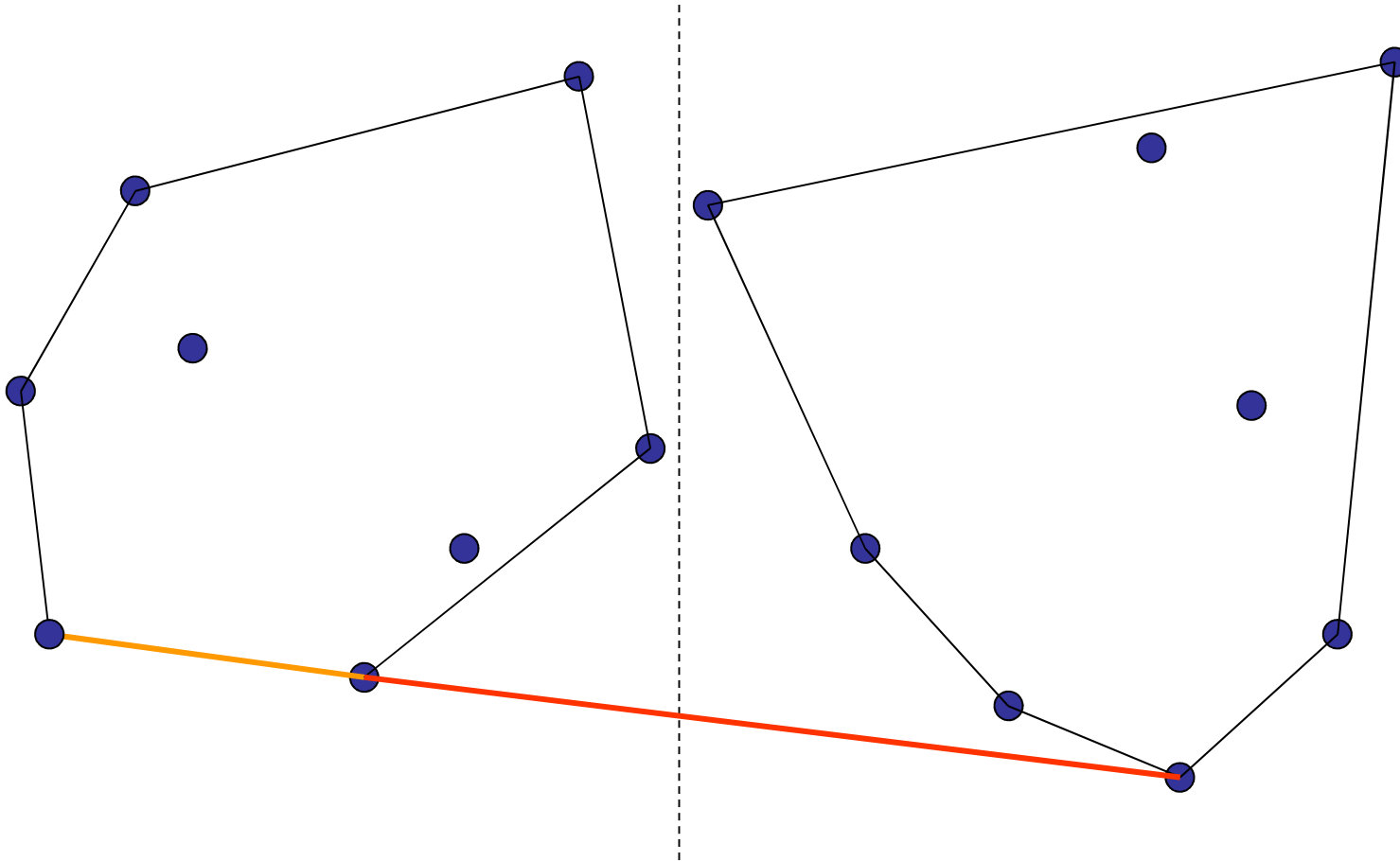
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (i) Find the **lower** tangent.



# Convex Hull – Divide & Conquer

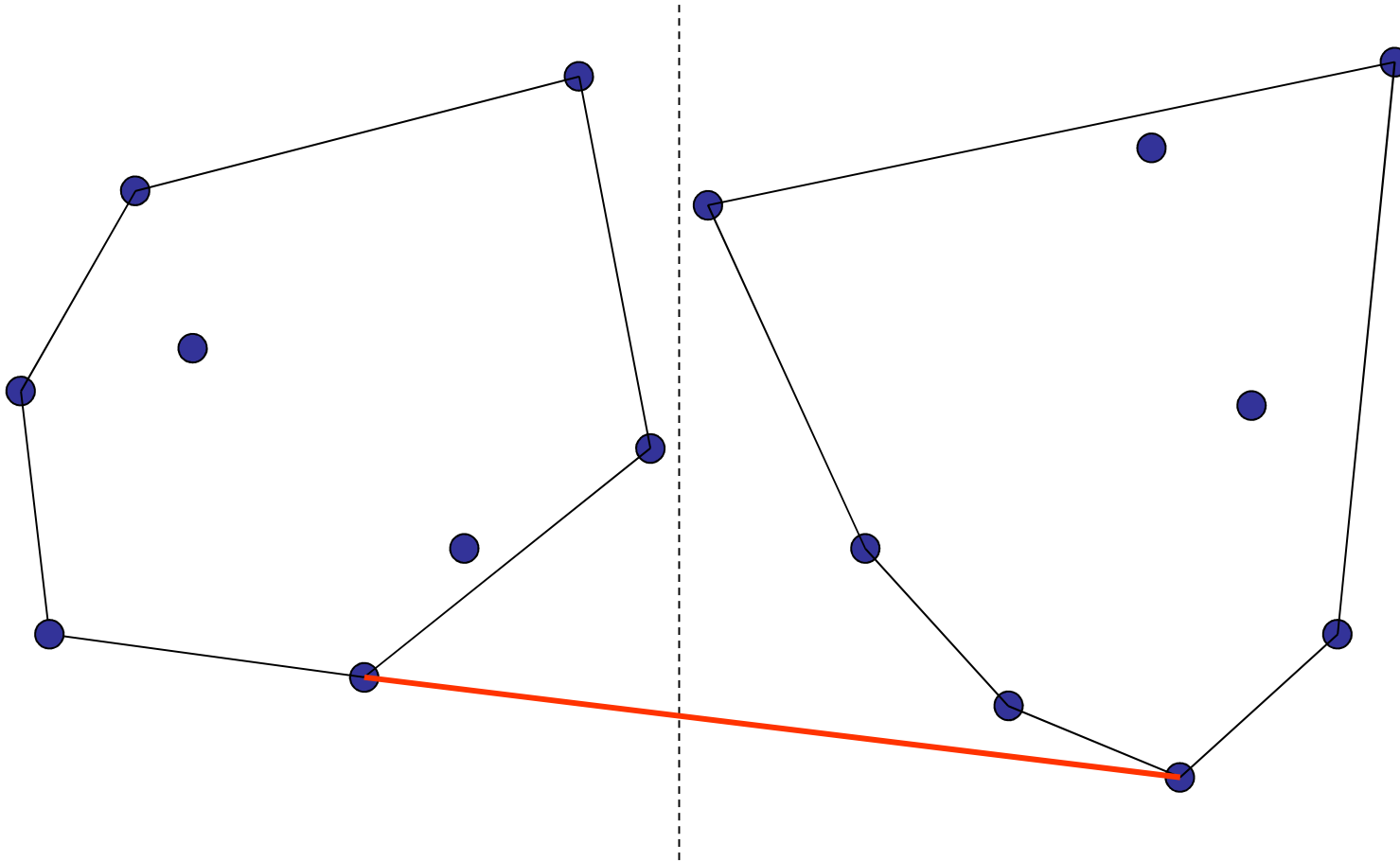
- Merging two convex hulls: (i) Find the **lower** tangent.





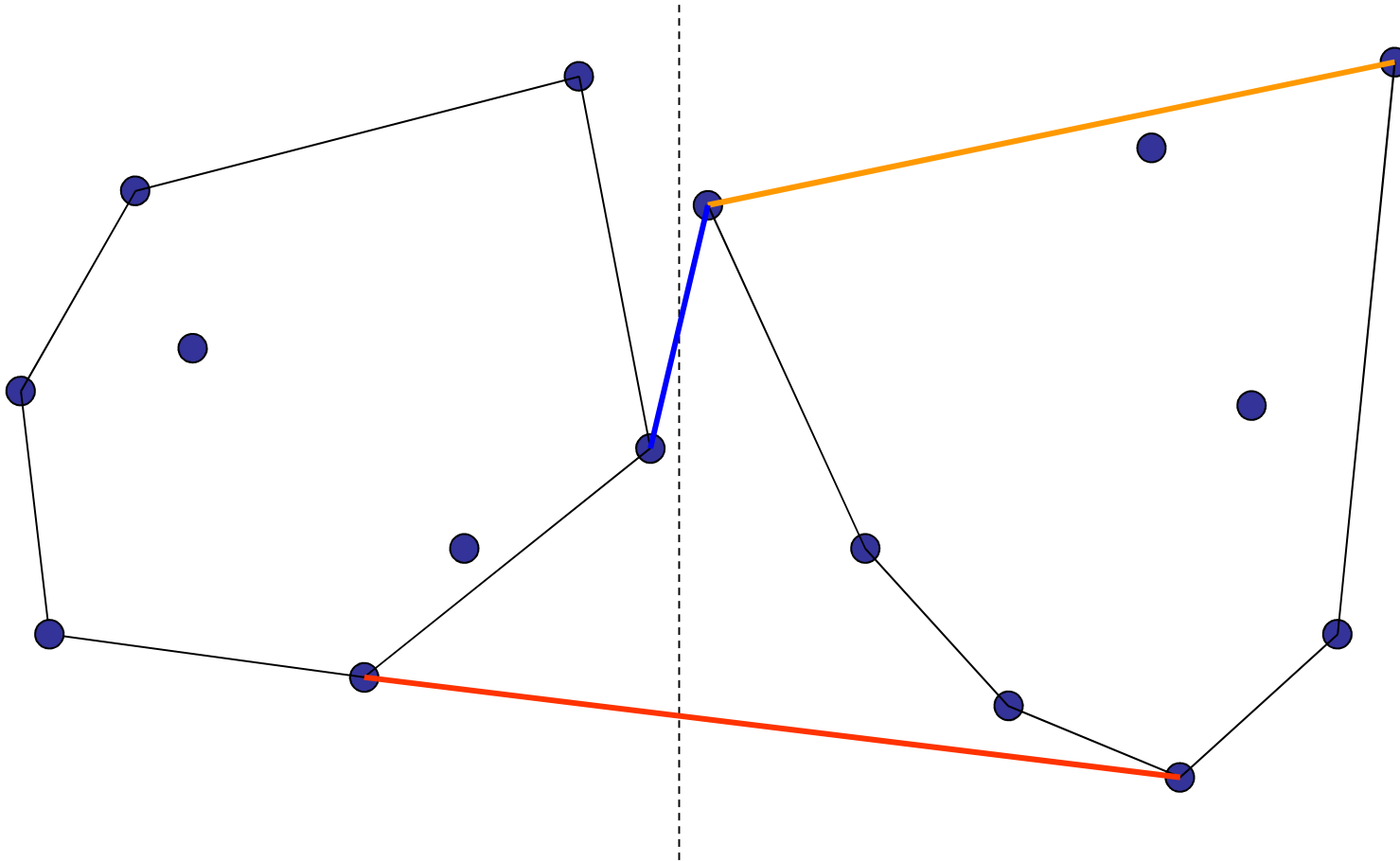
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (i) Find the **lower** tangent.



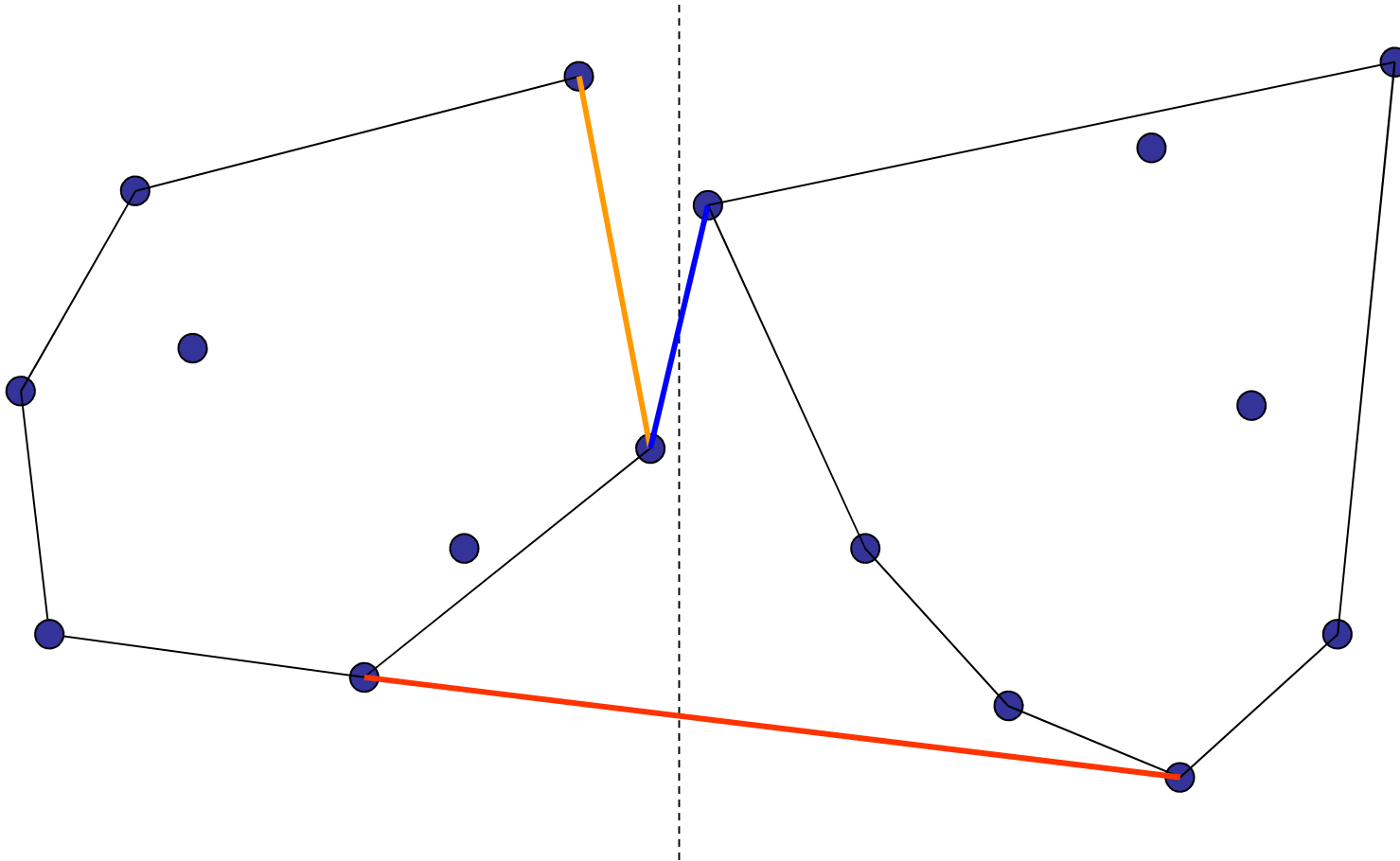
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (ii) Find the **upper** tangent.



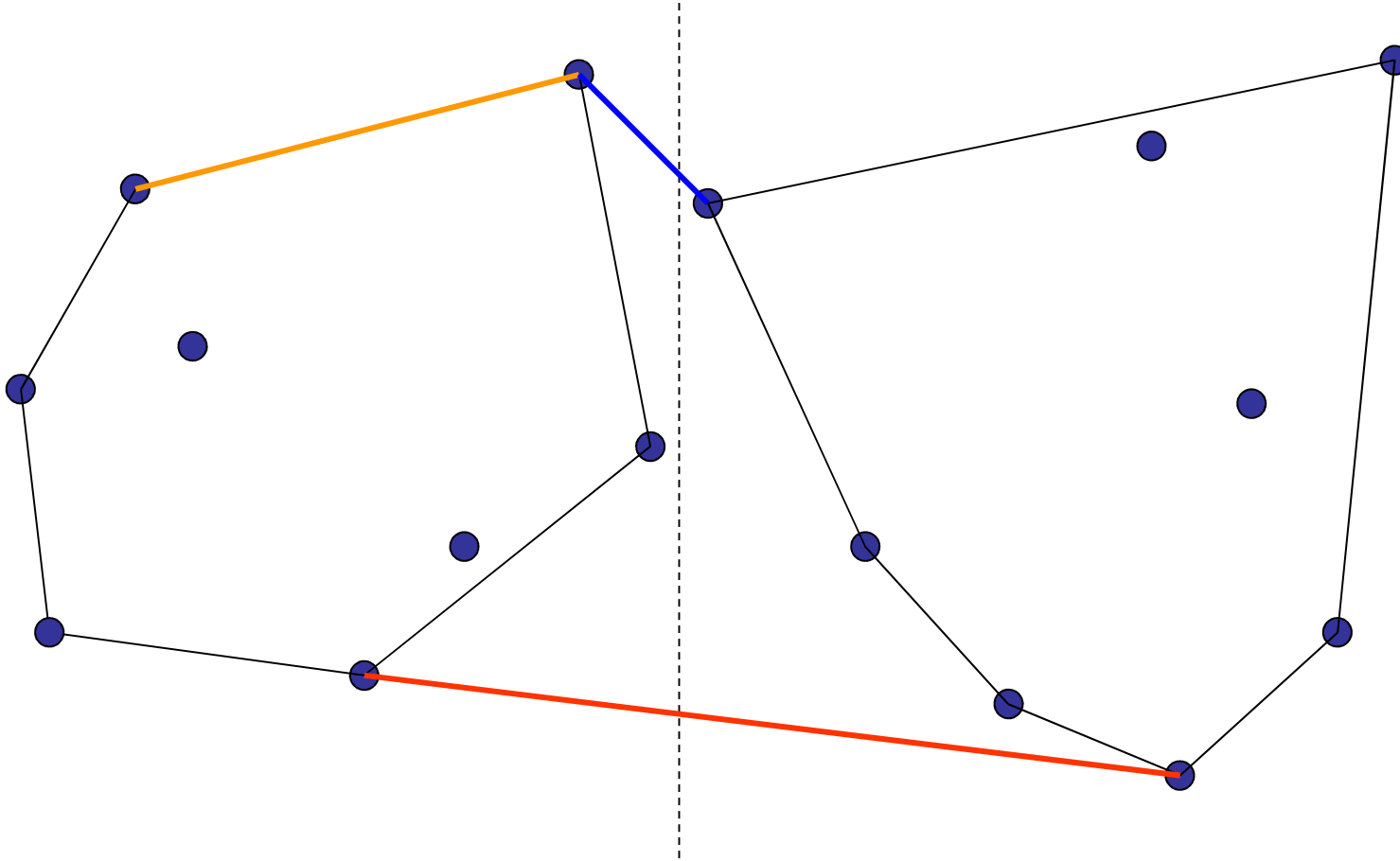
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (ii) Find the **upper** tangent.



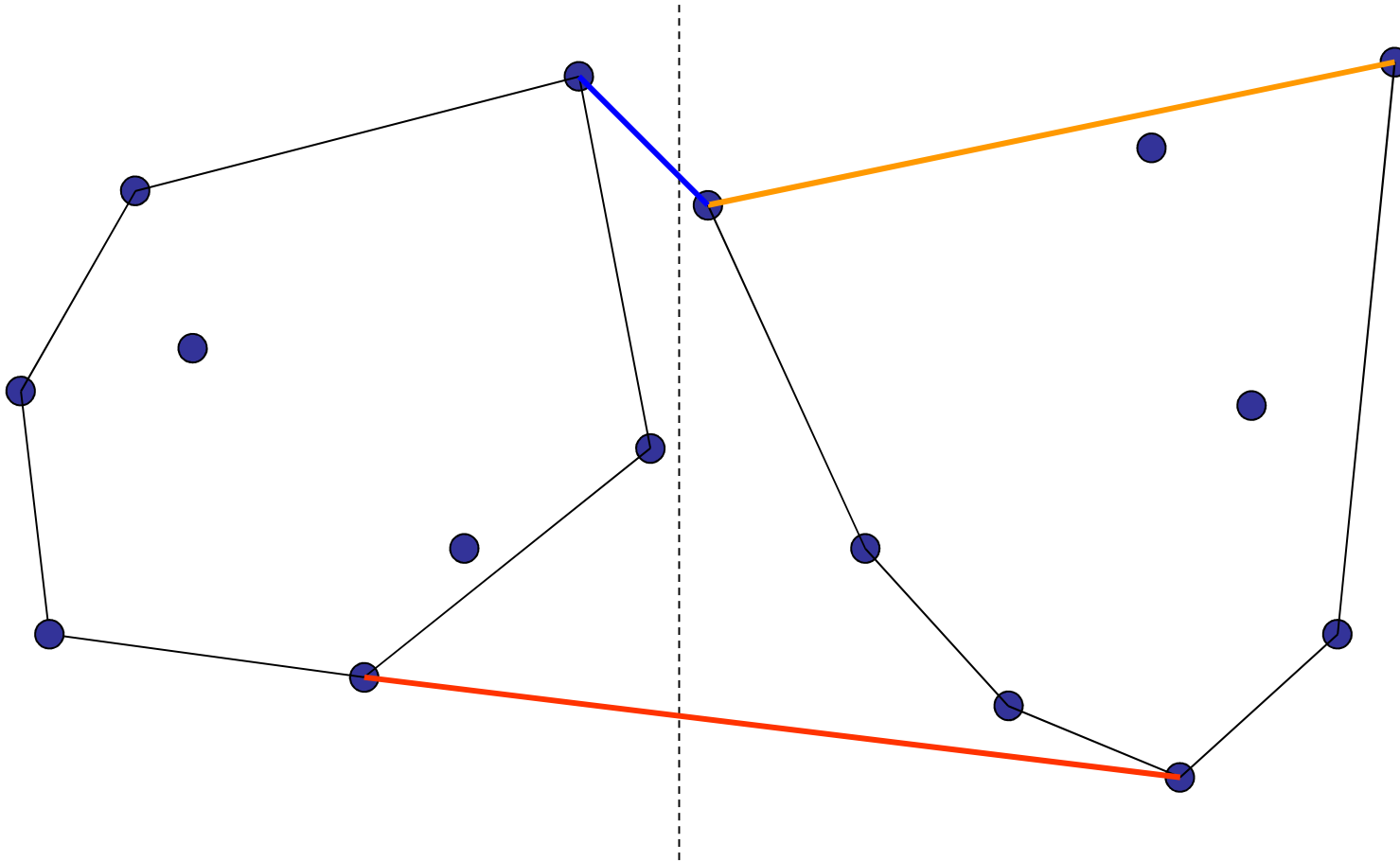
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (ii) Find the **upper** tangent.



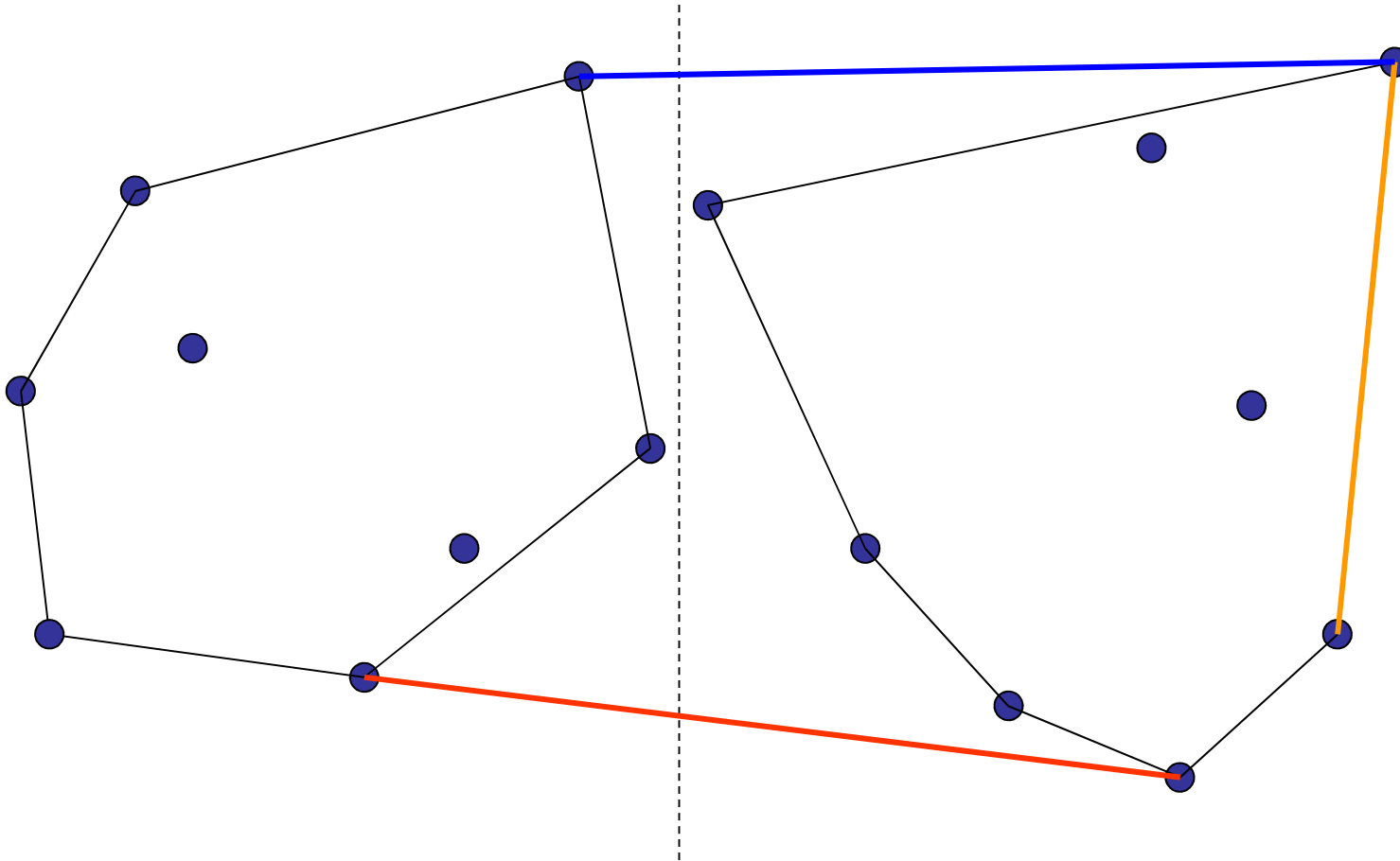
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (ii) Find the **upper** tangent.



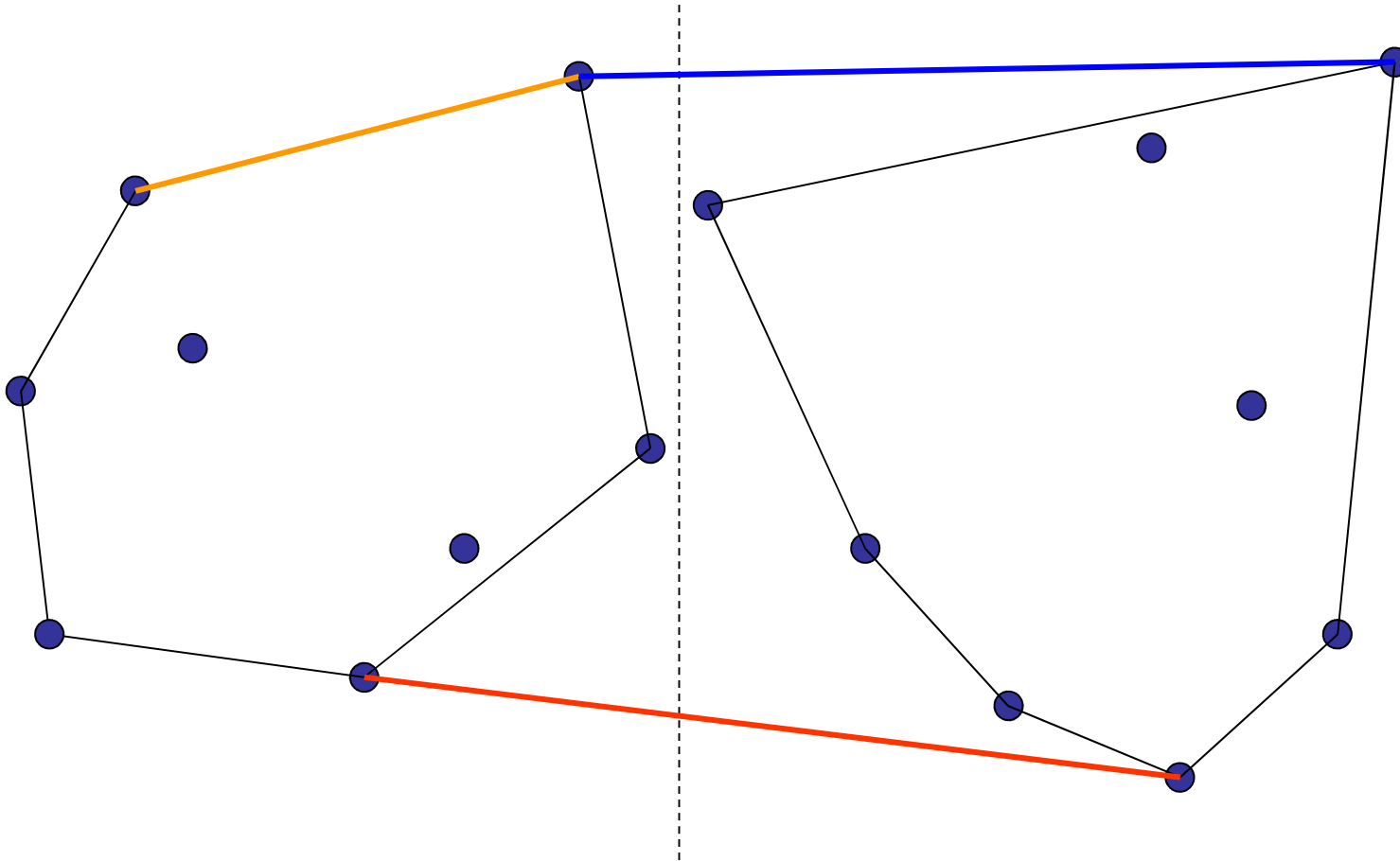
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (ii) Find the **upper** tangent.



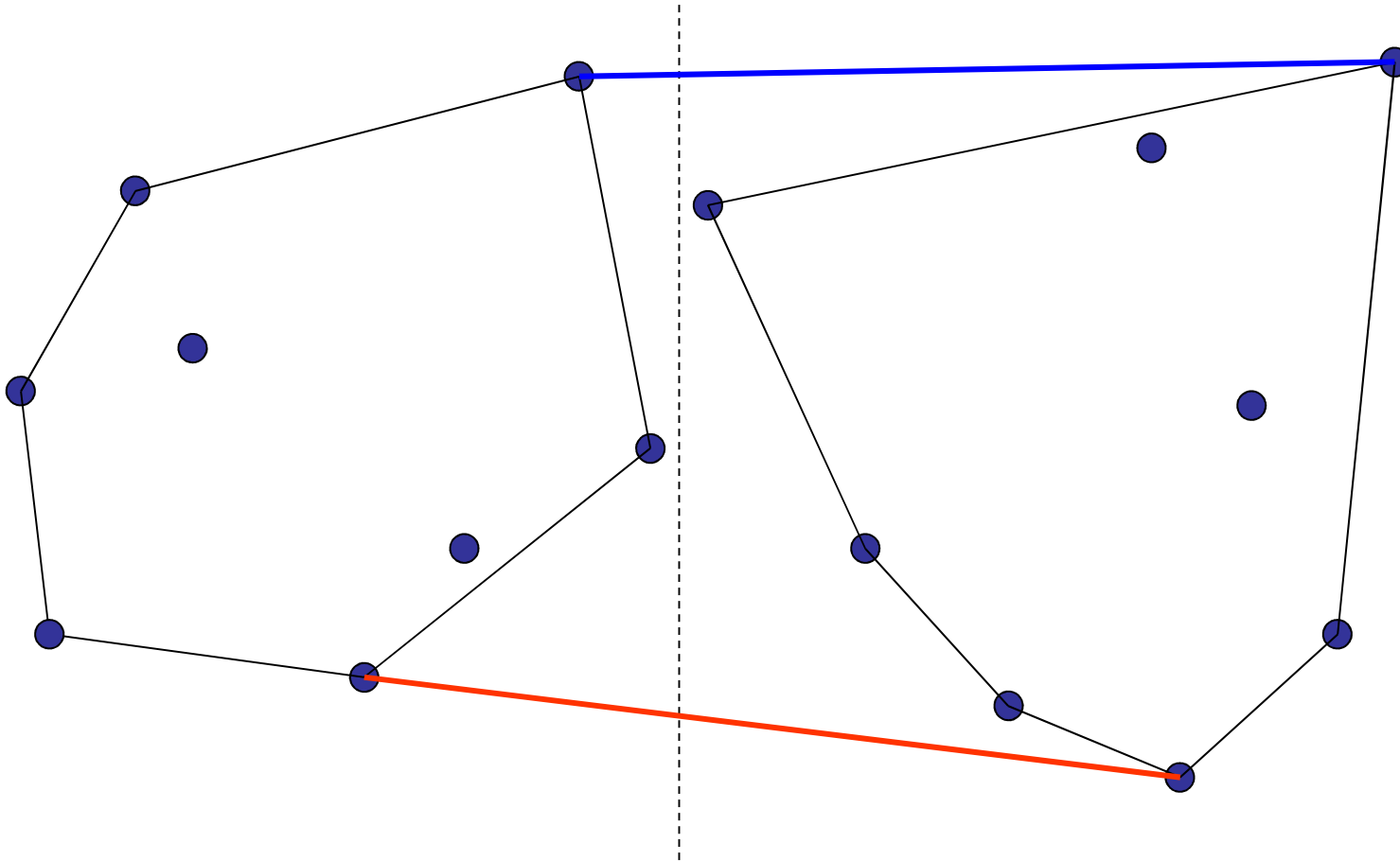
# Convex Hull – Divide & Conquer

- Merging two convex hulls: (ii) Find the **upper** tangent.



# Convex Hull – Divide & Conquer

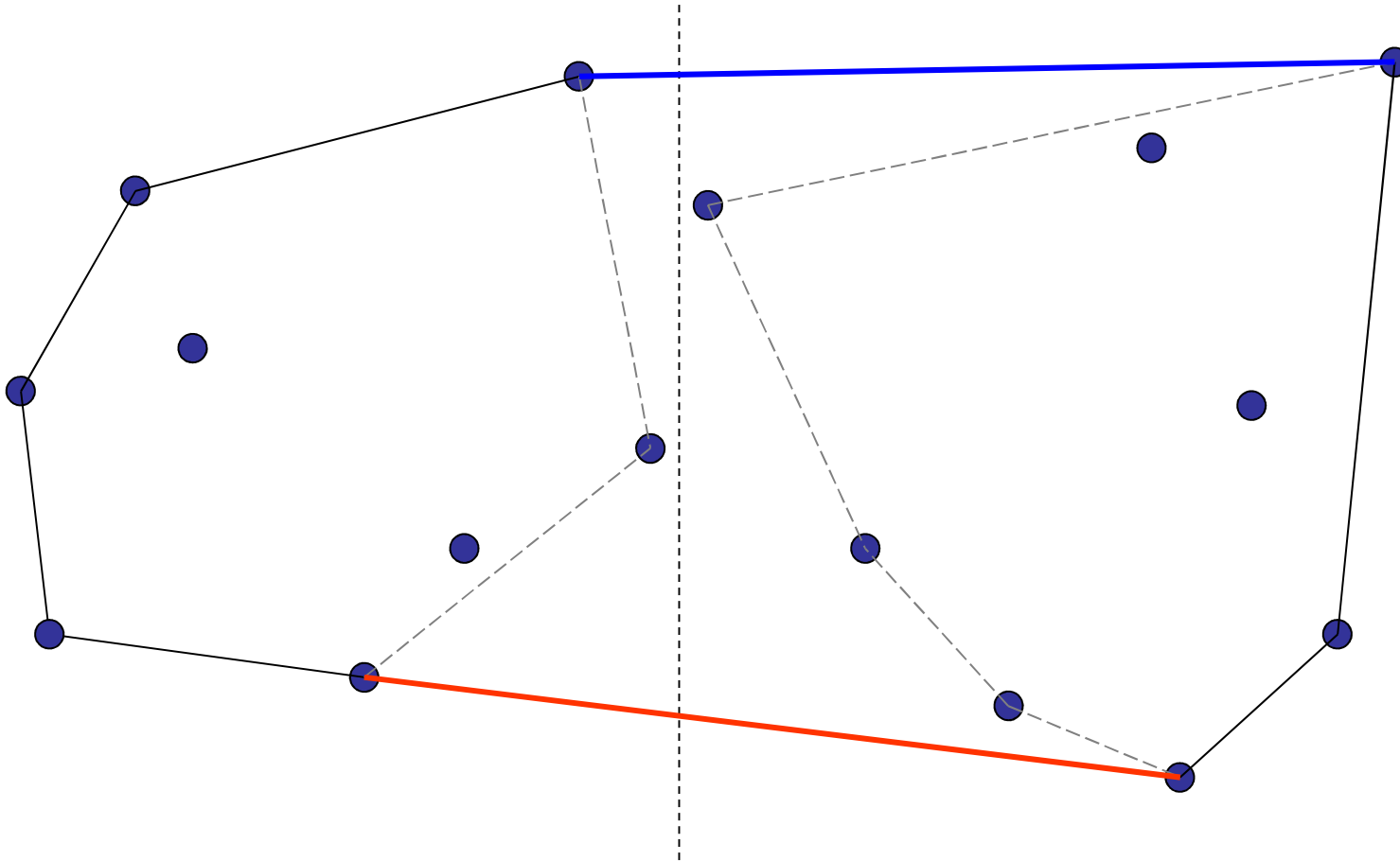
- Merging two convex hulls: (ii) Find the **upper** tangent.



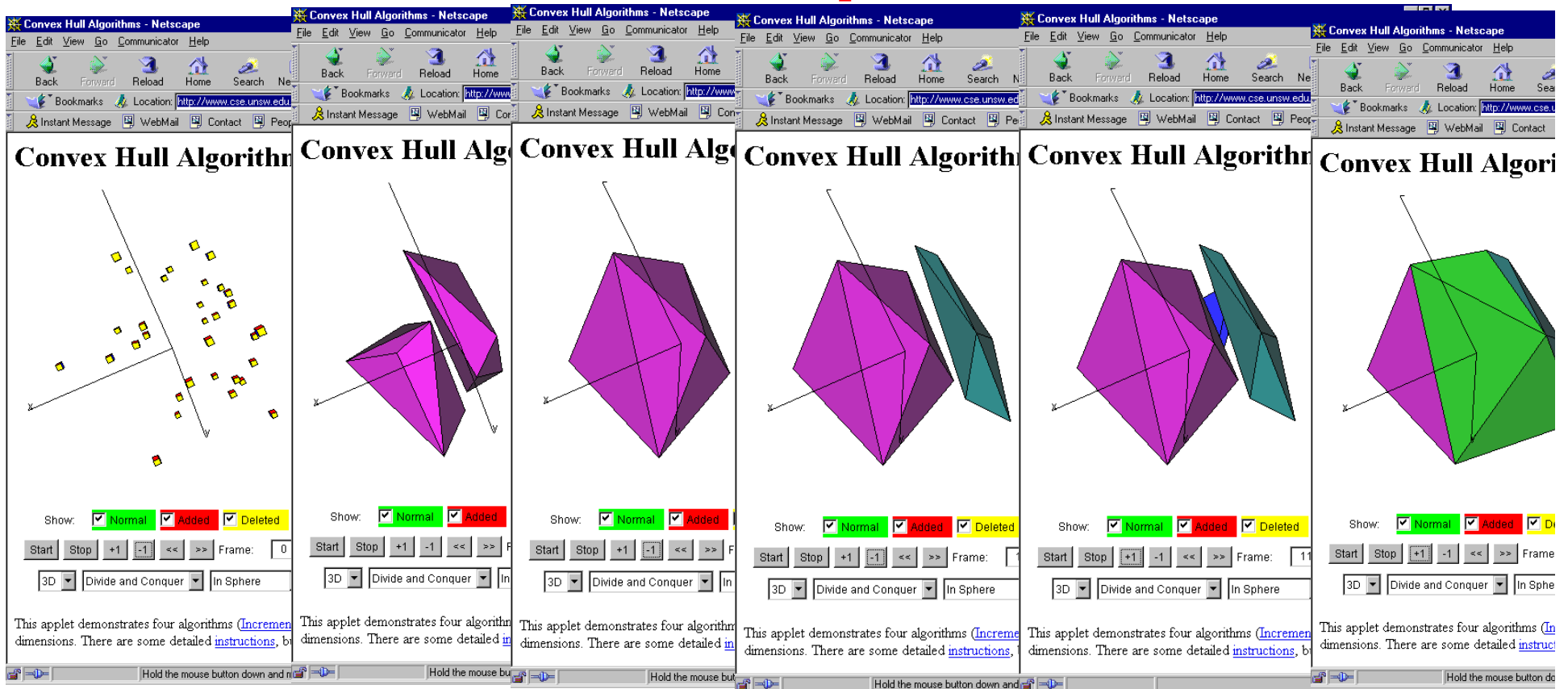


# Convex Hull – Divide & Conquer

- Merging two convex hulls: (iii) Eliminate non-hull edges.



# Algorithms: 3D Divide and Conquer



$O(n \log n)$  time !

CxHull Animations: <http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html>

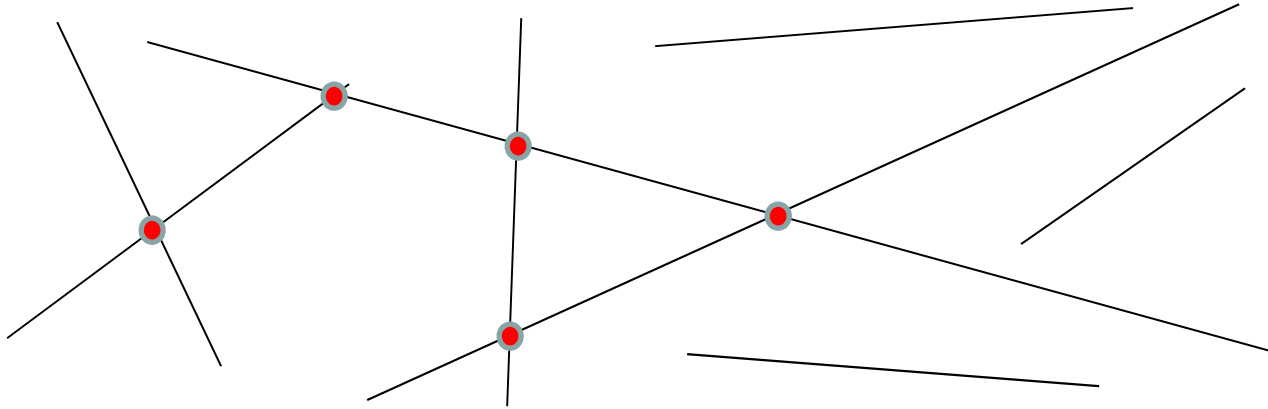
# Topic

- Introduction
- Two lines Intersection Test
- Point inside polygon
- Convex hull
- Line Segments Intersection Algorithm

# Line segment intersection

- Input:

- Set  $S = \{s_1, \dots, s_n\}$  of  $n$  line segments,  $s_i = (x_i, y_i)$



- Output:

- $k$  = All intersection points among the segments in  $S$

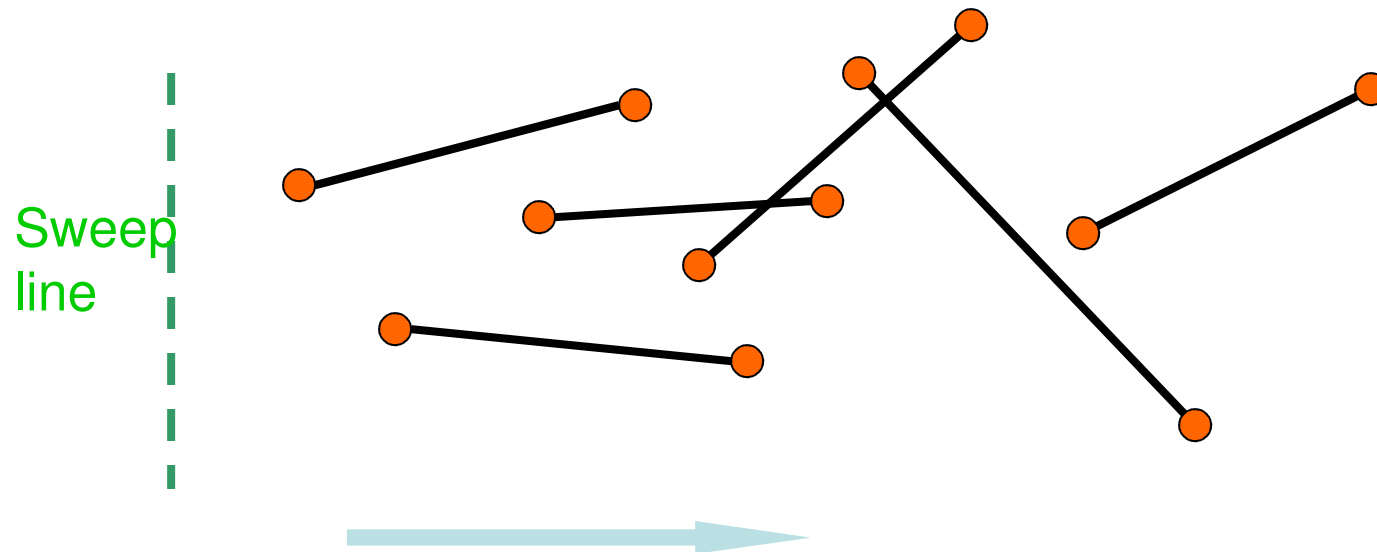
# Line segment intersection

- Worst case:
  - $k = n(n-1)/2 = O(n^2)$  intersections
- Sweep line algorithm (near optimal algorithm):
  - $O(n \log n + k)$  time and  $O(n)$  space
  - $O(n)$  space

# Sweep Line Algorithm

Avoid testing pairs of segments that are far apart.

**Idea:** *imagine* a vertical sweep line passes through the given set of line segments, from left to right.



# Assumption on Non-degeneracy

No segment is vertical. // the sweep line always hits a segment at  
// a point.

If a segment is vertical, imagine we rotate it clockwise by a tiny angle.  
This means:

For each vertical segment, we will consider its lower  
endpoint before upper point.

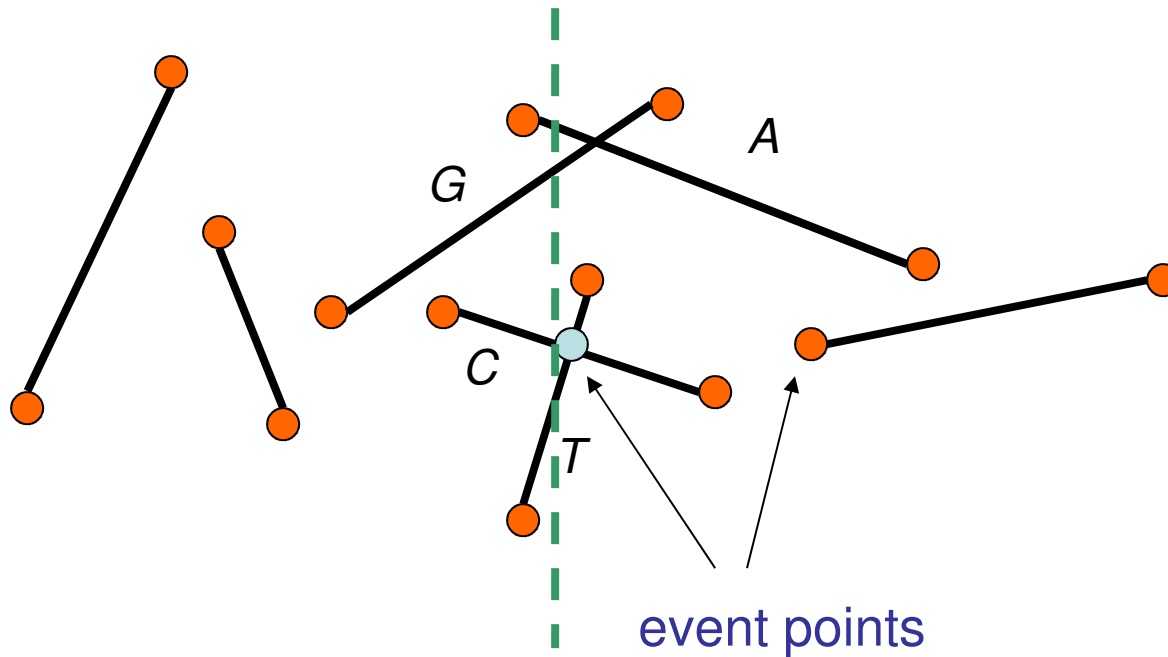
# Sweep Line Status

*The set of segments intersecting the sweep line.*

It changes as the sweep line moves, but *not continuously*.

Updates of status happen only at *event points*.

{ left endpoints  
right endpoints  
intersections

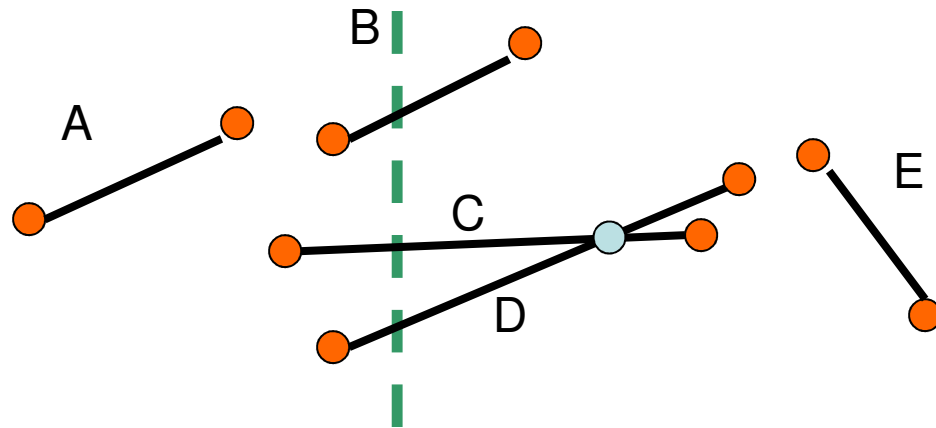




# Ordering Segments

A *total order* over the segments that intersect the current position of the sweep line:

- Based on which parts of the segments we are currently interested in



$B > C > D$   
(A and E not in the ordering)

$C > D$   
(B drops out of the ordering)

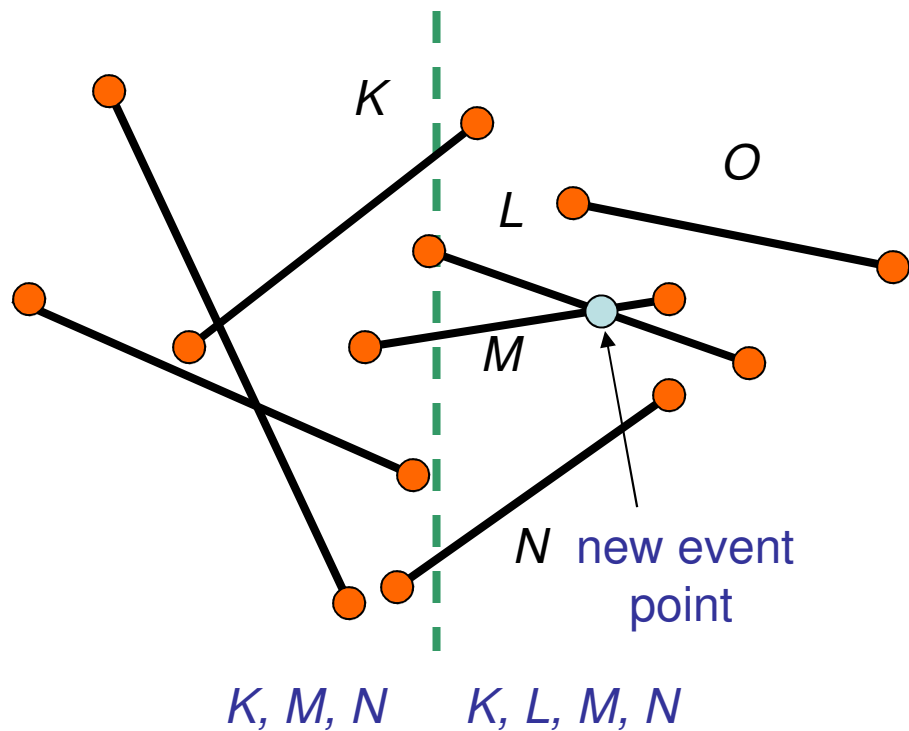
$D > C$   
(C and D swap their positions)

At an event point, the sequence of segments changes:

- ◆ Update the status.
- ◆ Detect the intersections.

# Status Update (1)

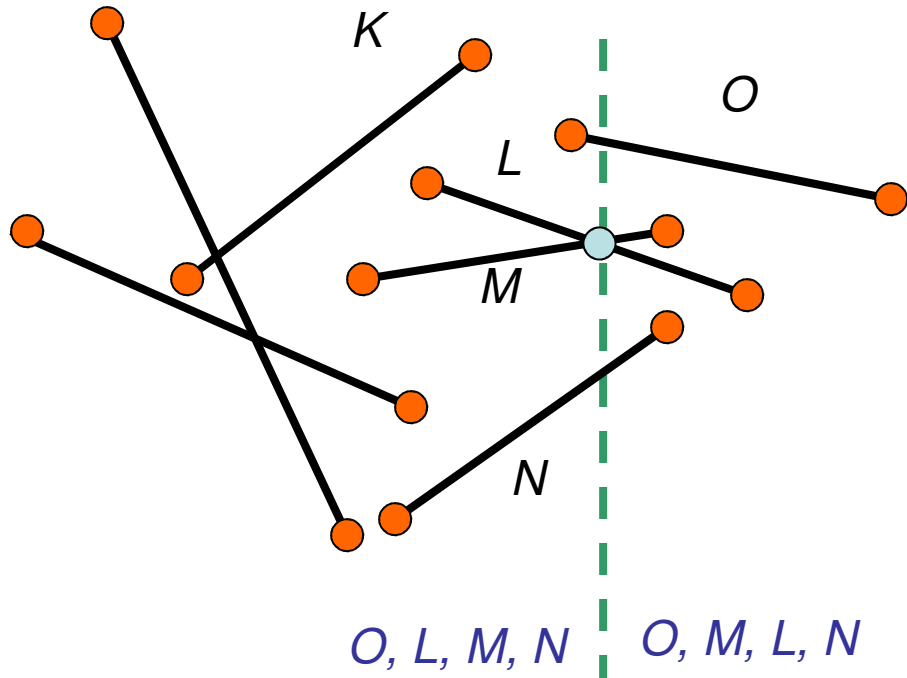
Event point is the left endpoint of a segment.



- ◆ A new segment *L* intersecting the sweep line
- ◆ Check if *L* intersects with the segment above (*K*) and the segment below (*M*).
- ◆ Intersection(s) are new event points.

# Status Update (2)

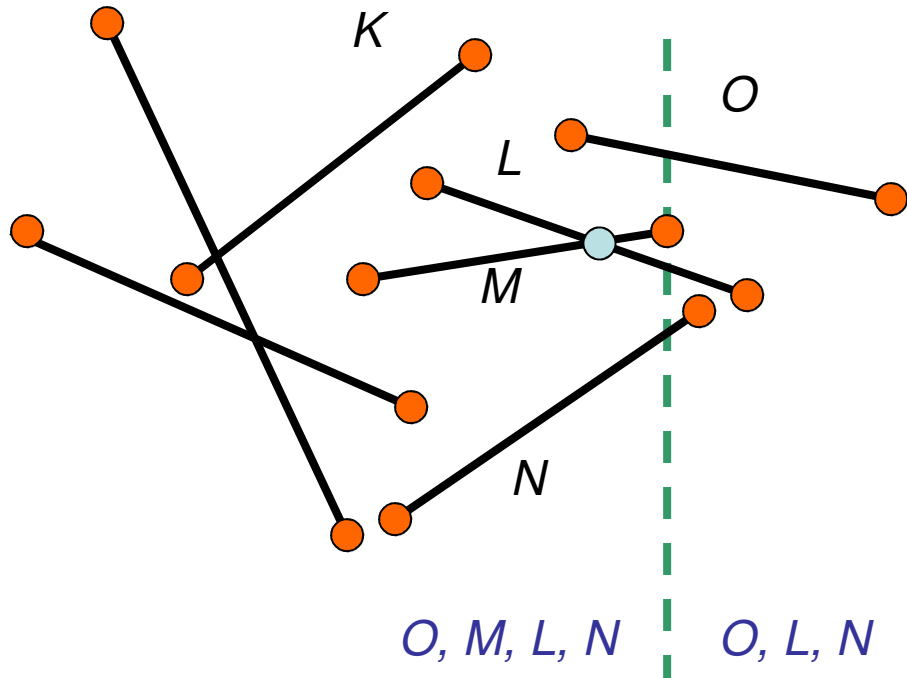
Event point is an intersection.



- ◆ The two intersecting segments ( $L$  and  $M$ ) change order.
- ◆ Check intersection with new neighbors ( $M$  with  $O$  and  $L$  with  $N$ ).
- ◆ Intersection(s) are new event points.

# Status Update (3)

Event point is a lower endpoint of a segment.



- ◆ The two neighbors ( $O$  and  $L$ ) become adjacent.
- ◆ Check if they ( $O$  and  $L$ ) intersect.
- ◆ Intersection is new event point.

# Data Structure for Event Queue

Ordering of event points:

- ✦ by  $x$ -coordinates
- ✦ by  $y$ -coordinates in case of a tie in  $x$ -coordinates.

Supports the following operations on a segment  $s$ .

- fetching the next event  $// O(\log m)$
- inserting an event  $// O(\log m)$

Every event point  $p$  is stored with all segments starting at  $p$ .

Data structure: balanced binary search tree (e.g., red-black tree).

$m = \text{\#event points in the queue}$

# Data Structure for Sweep-line Status

✧ Describes the relationships among the segments intersected by the sweep line.

✧ Use a balanced binary search tree  $T$  to support the following operations on a segment  $s$ .

Insert( $T, s$ )

Delete( $T, s$ )

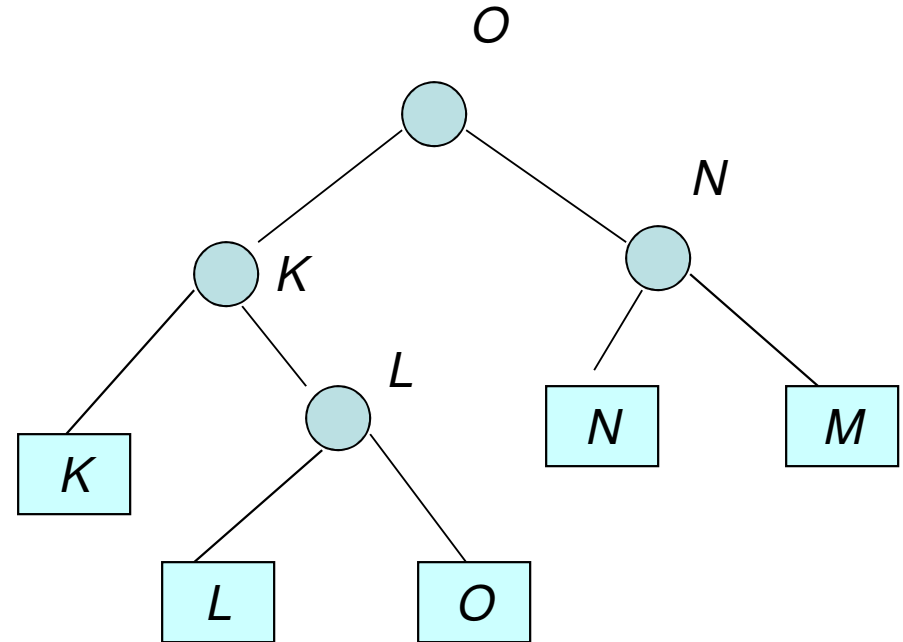
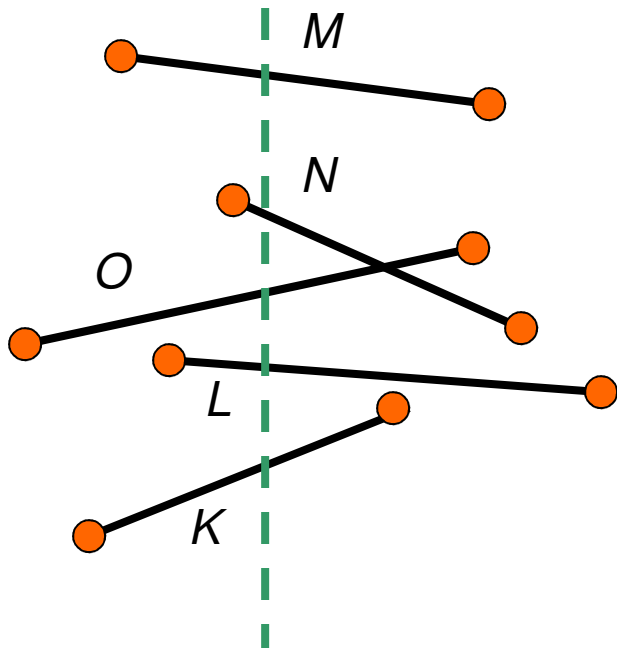
Above( $T, s$ ) // segment immediately above  $s$

Below( $T, s$ ) // segment immediately below  $s$

✧ e.g, Red-black trees, splay trees (key comparisons replaced by cross-product comparisons).

✧  $O(\log n)$  for each operation.

# An Example

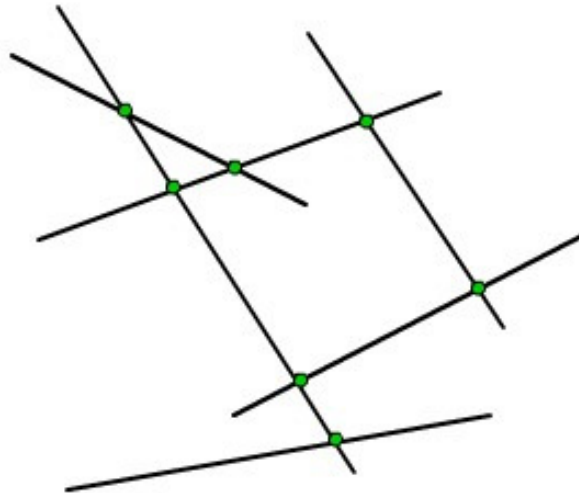


- ◆ The bottom-up order of the segments correspond to the *left-to-right* order of the leaves in the tree  $T$ .
- ◆ Each internal node stores the segment from the rightmost leaf in its left subtree.

# Line segment intersection

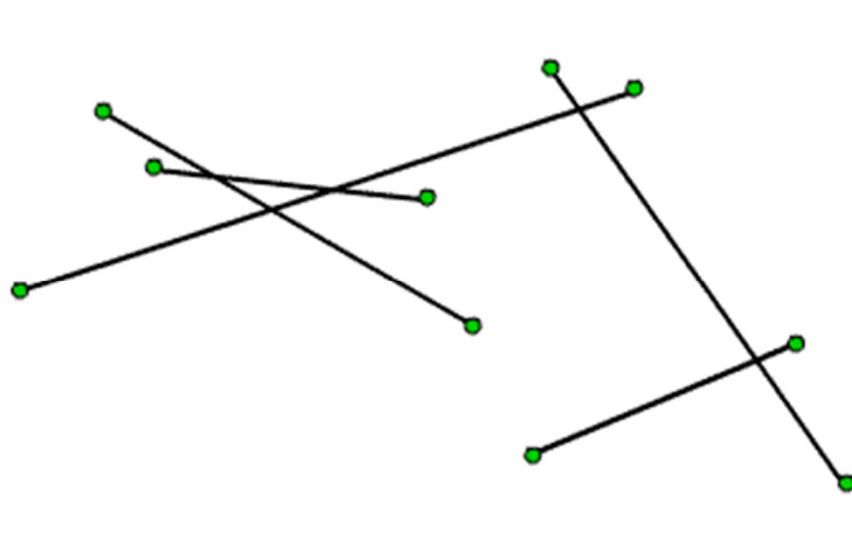
Input:  $n$  line segments

Output: all intersection points





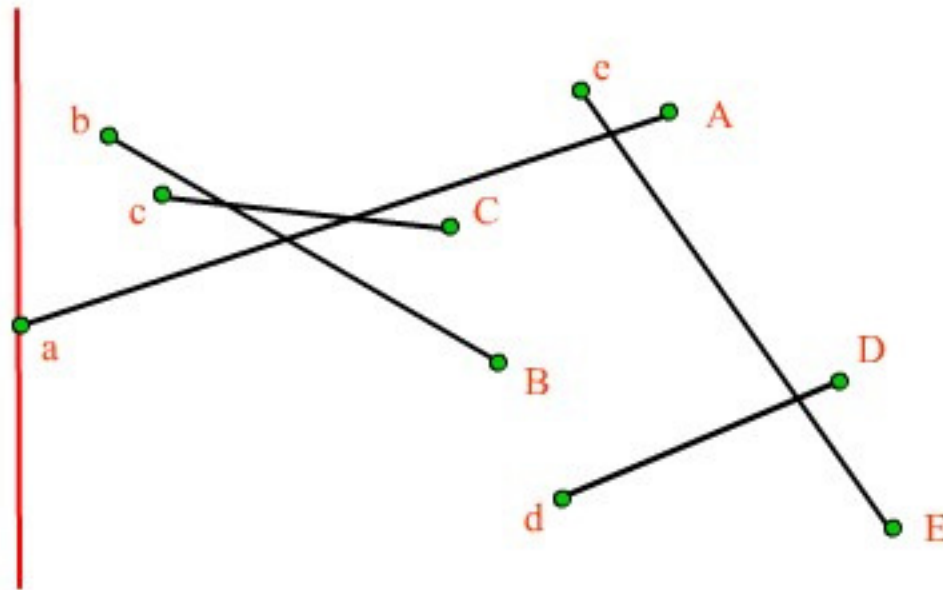
# Sweeping...



**Let's trace...**

Intersect:

aA



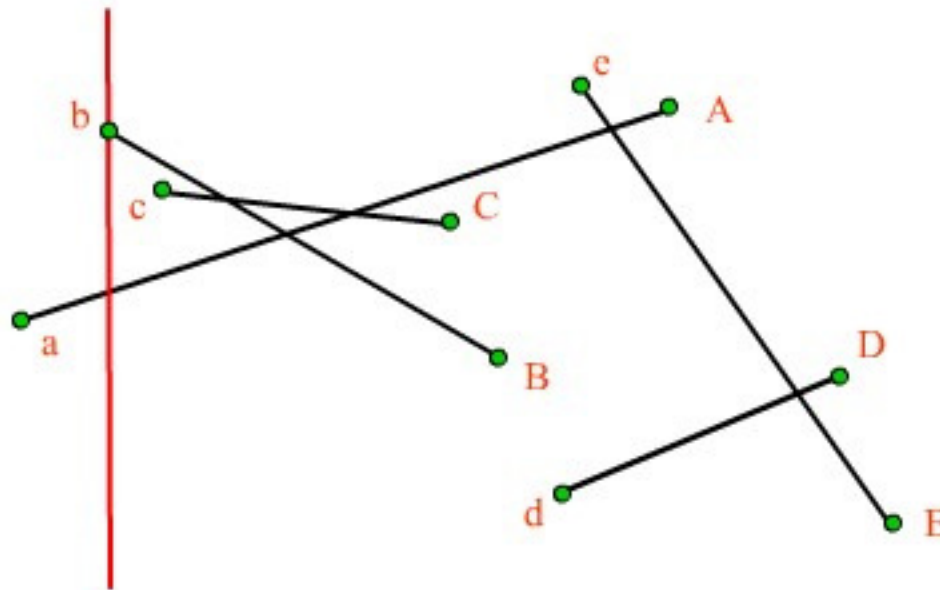
Event: a b c C B d e A D E

# Let's trace...

Intersect:

aA

Insert ab  
Add bB



Event: b c C B d e A D E

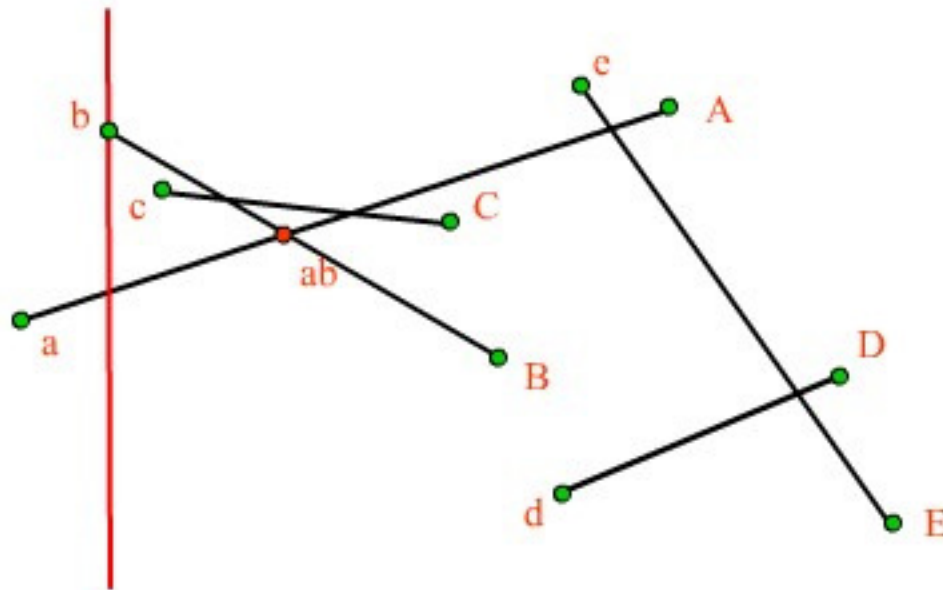
**Key:** two segments intersect, they must be adjacent in the intersection list at certain moment.

# Let's trace...

Intersect:

bB

aA



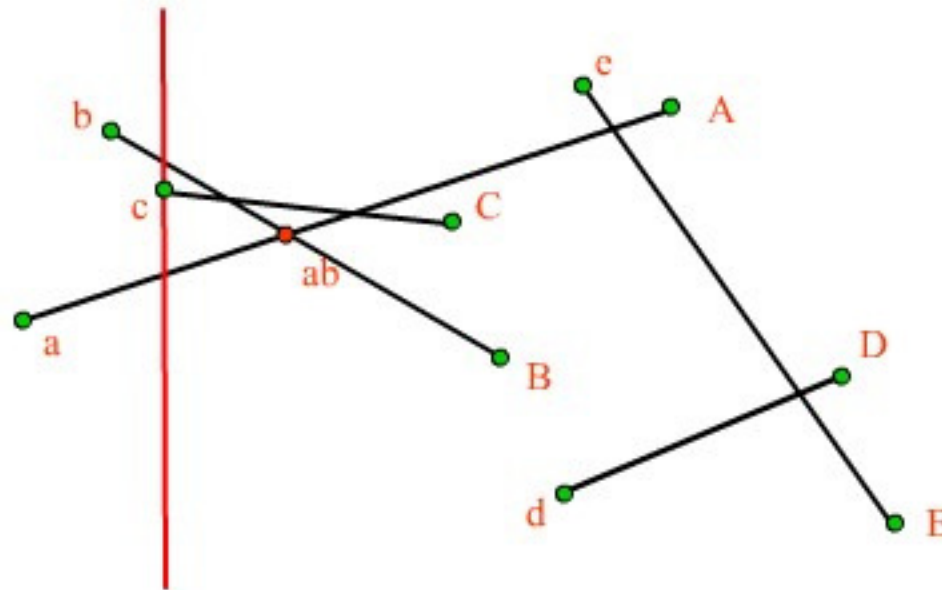
Event: b c **ab** C B d e A D E

## Let's trace ...

Intersect:

bB

aA



Insert  
bc  
Insert  
ac  
Add cC

Event: c ab C B d e A D E

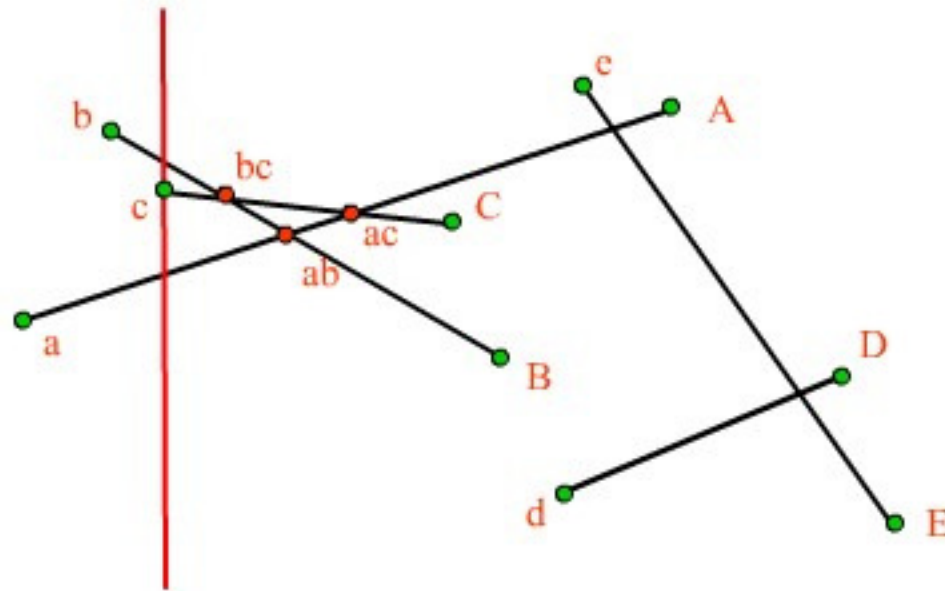
# Let's trace ...

Intersect:

bB

cC

aA



Event: c bc ab ac C B d e A D E

## Let's trace ...

Intersect:

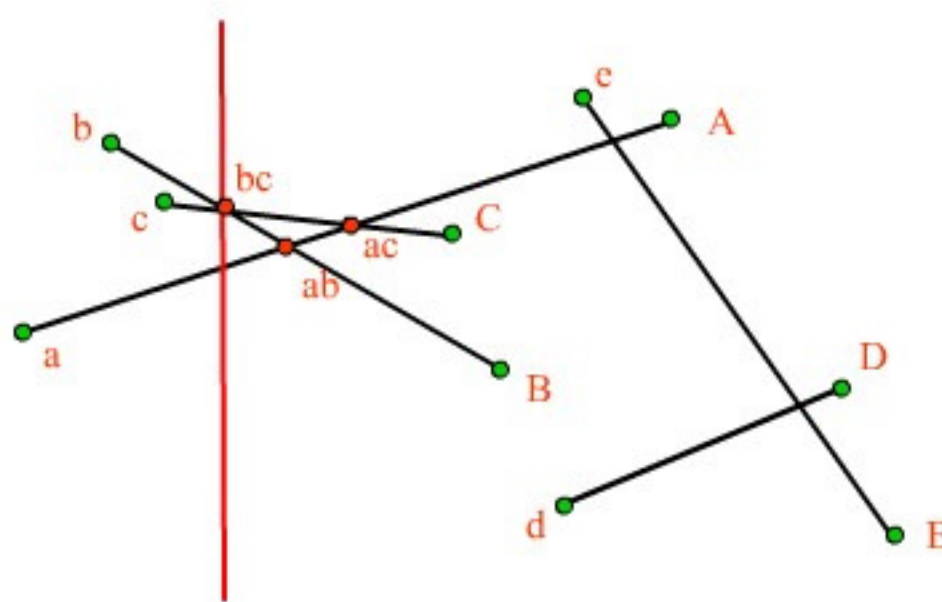
bB

cC

aA

Count bc

Swap bB-cC



Event: bc ab ac C B d e A D E



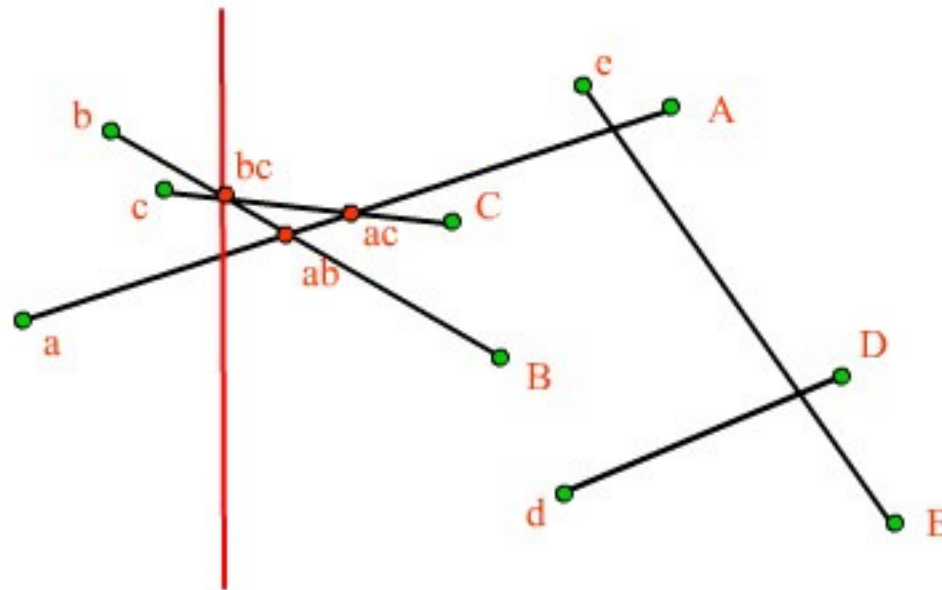
## Let's trace ...

Intersect:

cC

bB

aA



Event: bc ab ac C B d e A D E

## Let's trace ...

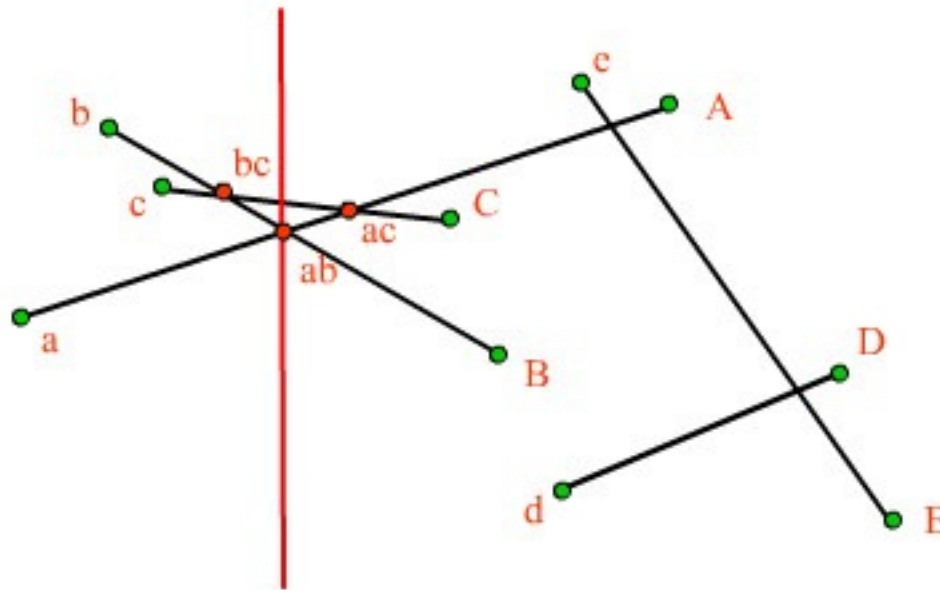
Intersect:

cC

bB

aA

Count ab  
Swap aA-bB



Event: ab ac C B d e A D E

## Let's trace ...

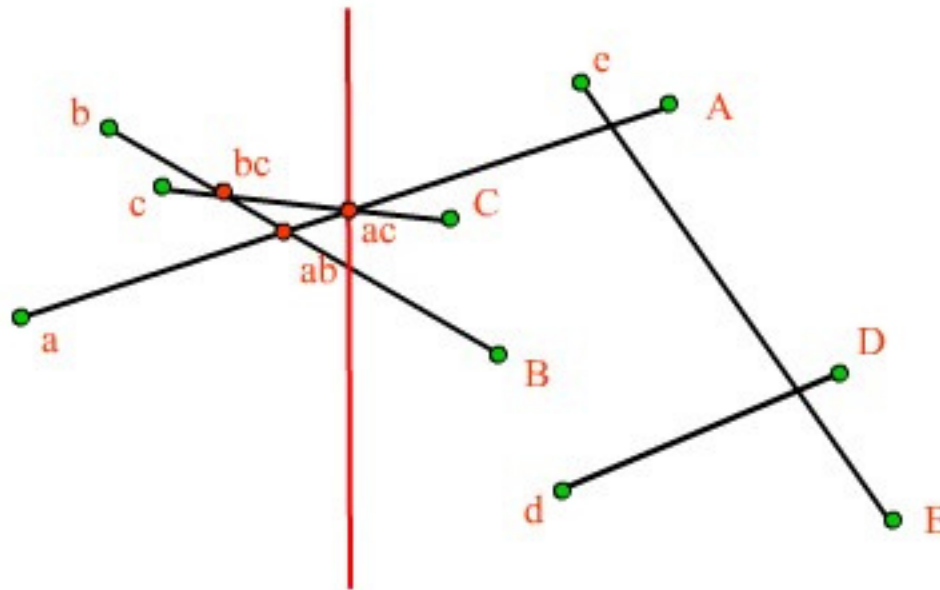
Intersect:

cC

aA

bB

Count ac  
Swap aA-cC



Event: **ac** C B d e A D E

## Let's trace ...

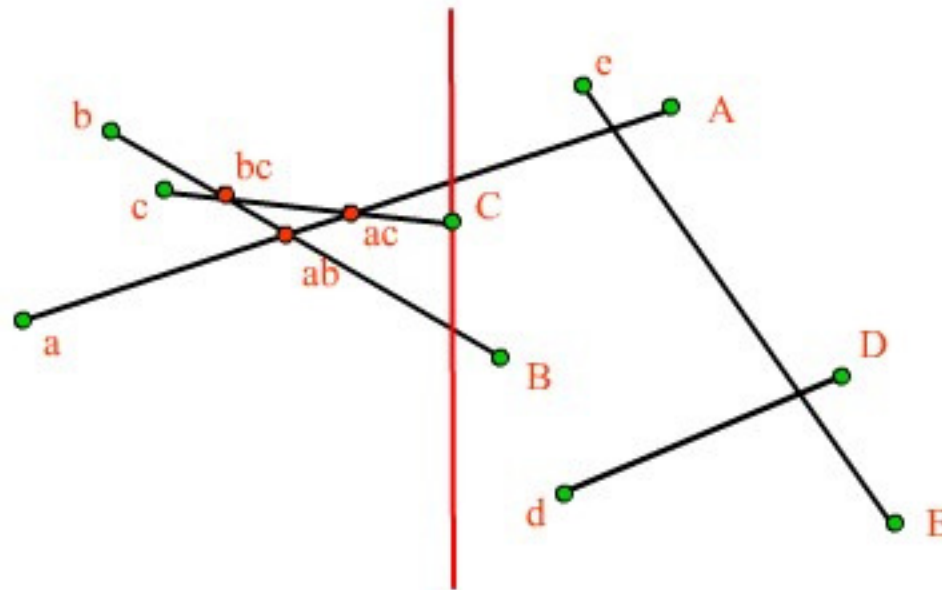
Intersect:

aA

cC

bB

Remove cC



Event: C B d e A D E

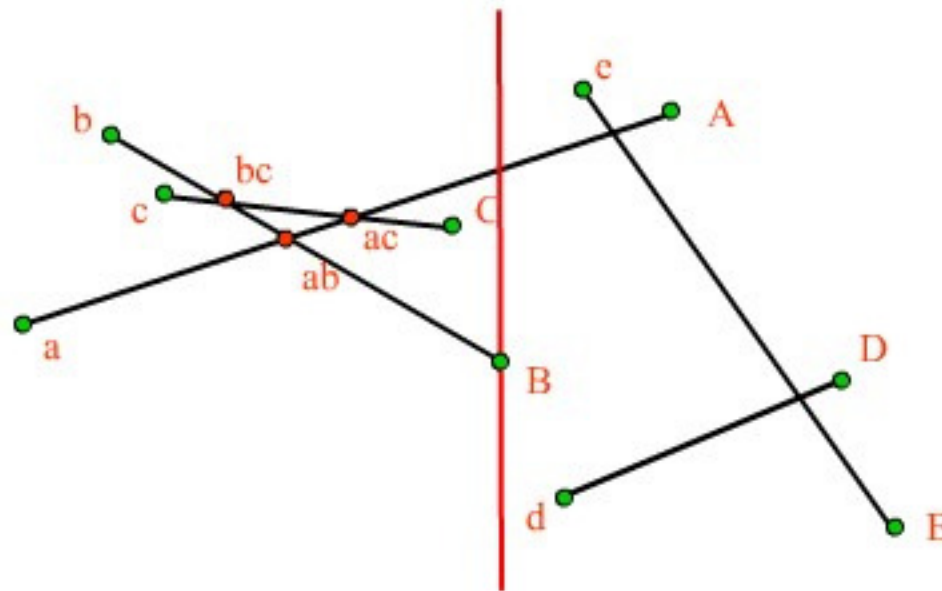
## Let's trace ...

Intersect:

aA

bB

Remove bB



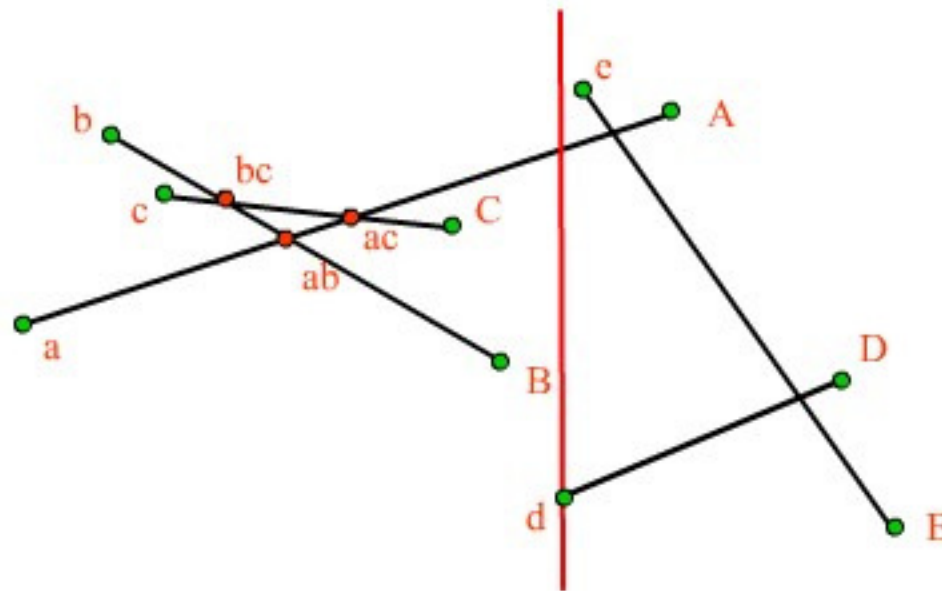
Event: B d e A D E

# Let's trace ...

Intersect:

aA

Add dD



Event: d e A D E

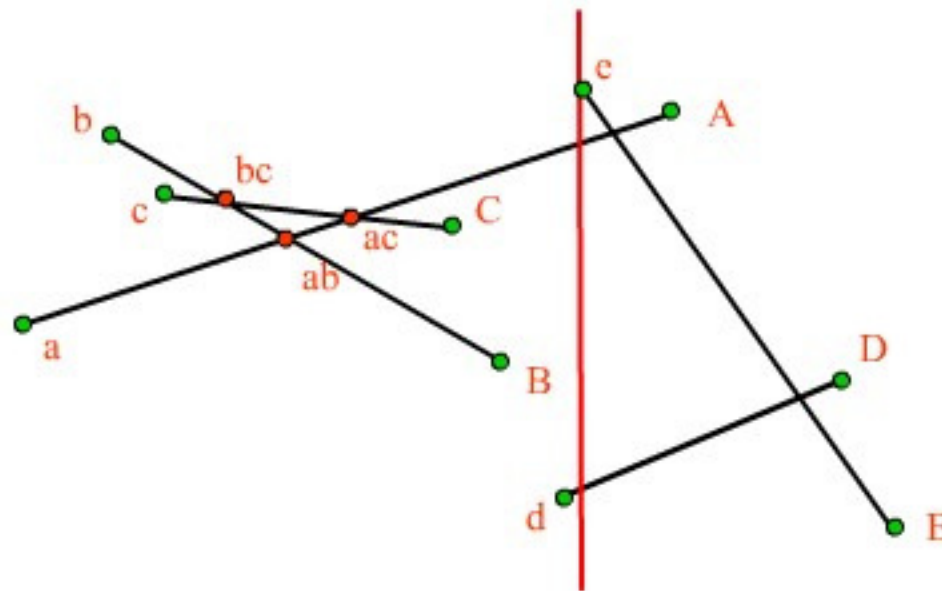
## Let's trace ...

Intersect:

aA

dD

Add eE  
Insert ae



Event: e A D E

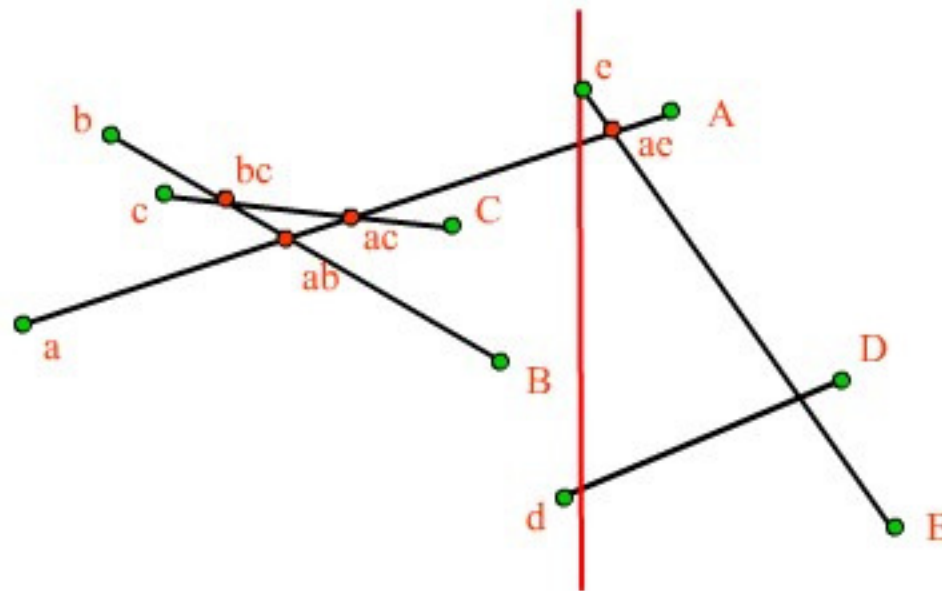
## Let's trace ...

Intersect:

eE

aA

dD



Event: e ae A D E



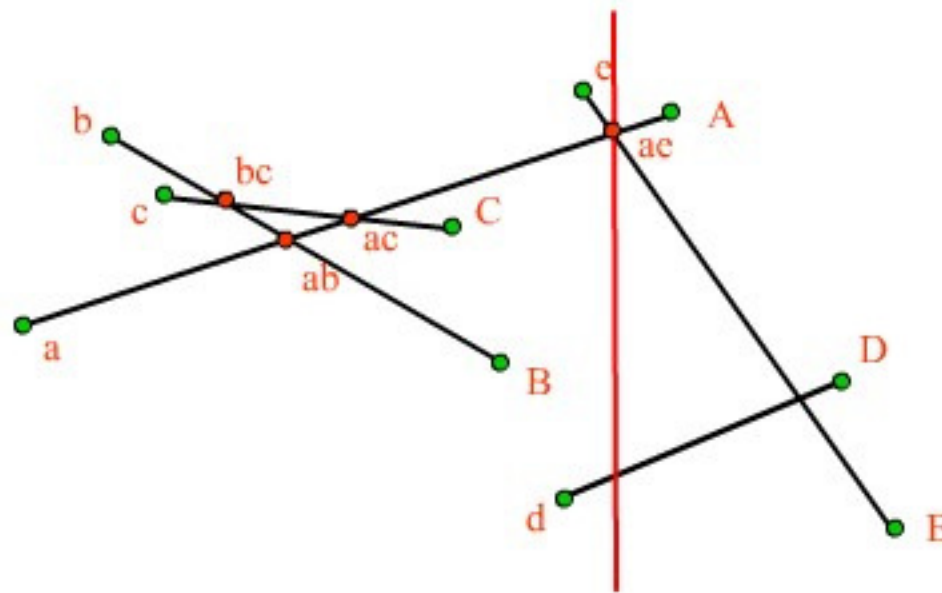
## Let's trace ...

Intersect:

eE

aA

dD



Count ae  
Swap eE-a/  
Insert de

Event: **ae** A D E

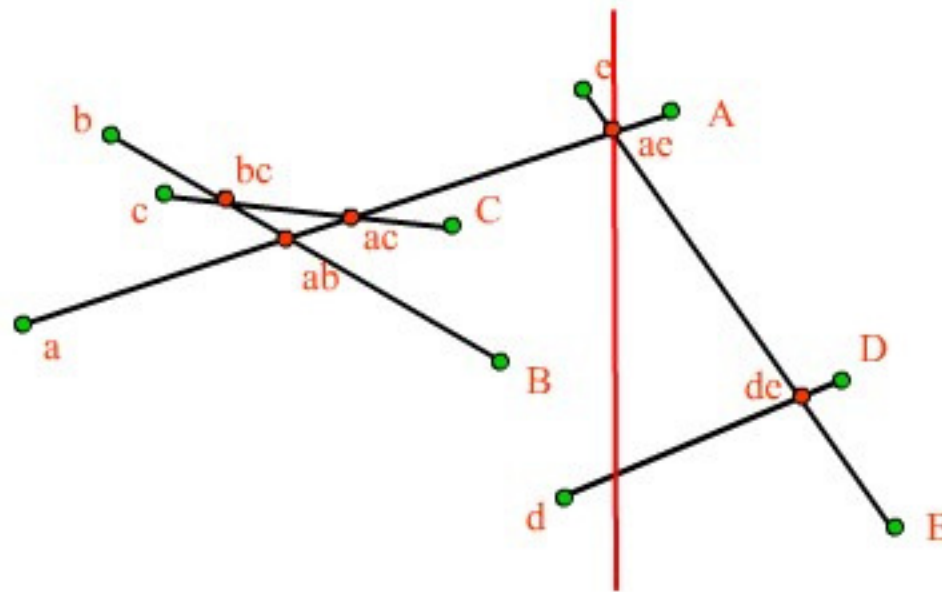
## Let's trace ...

Intersect:

aA

eE

dD



Event: ae A de D E

## Let's trace ...

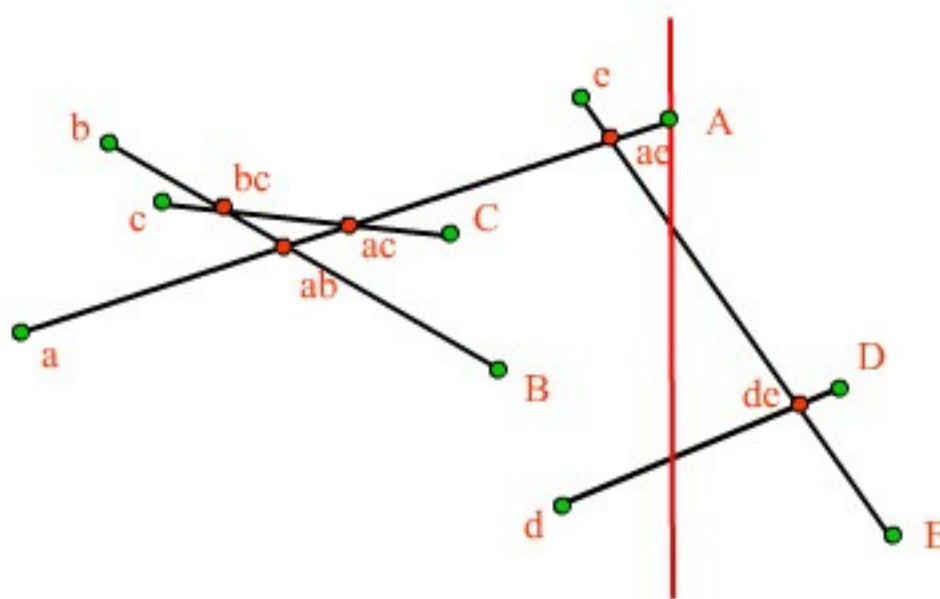
Intersect:

aA

eE

dD

Remove aA



Event: A de D E

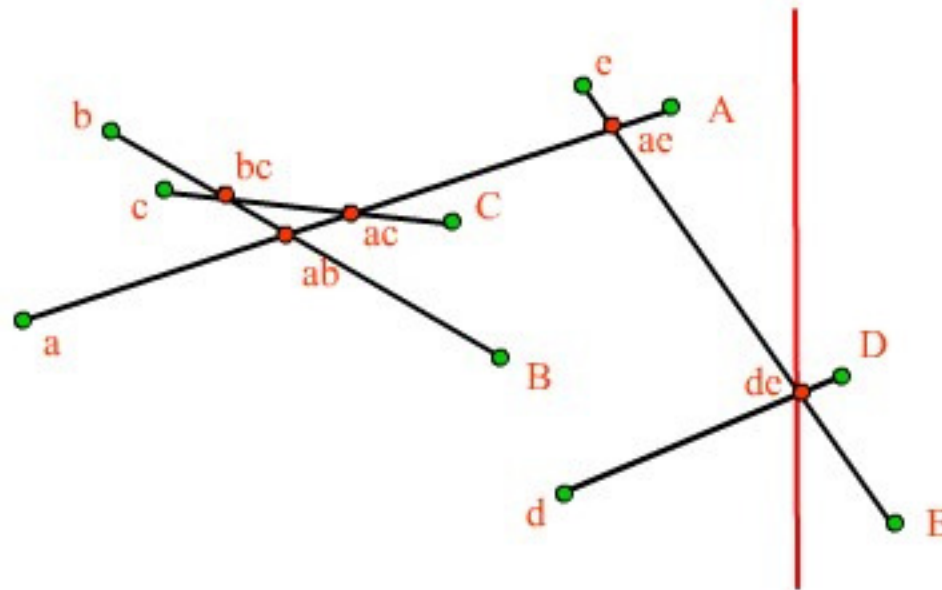
## Let's trace ...

Intersect:

eE

dD

Count de  
Swap dD-el



Event: de D E

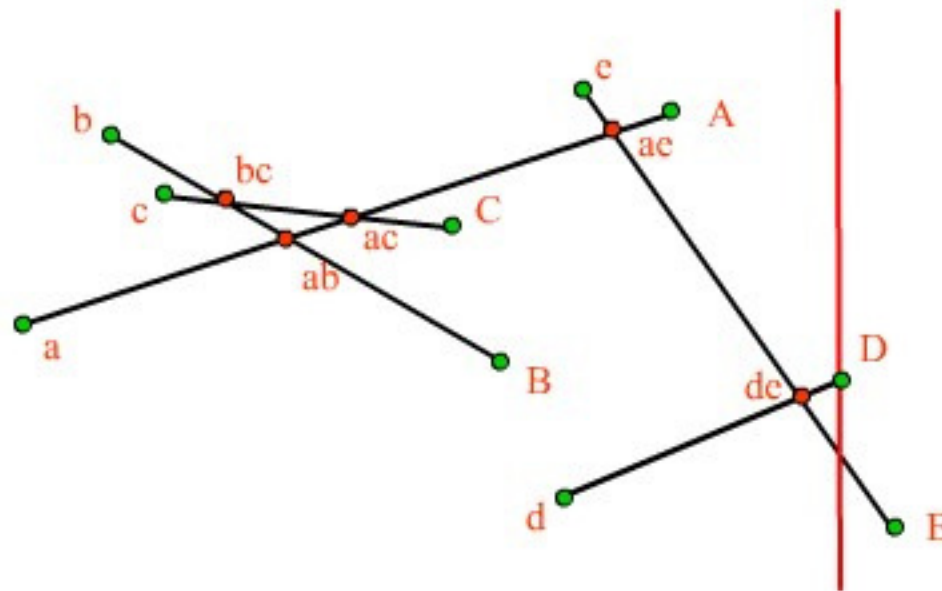
## Let's trace ...

Intersect:

dD

eE

Remove dD



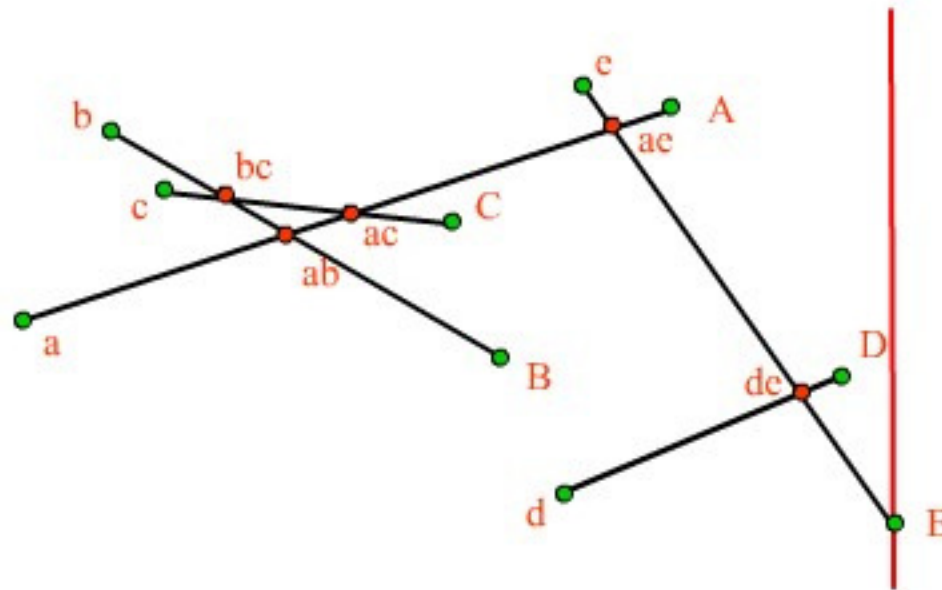
Event: D E

# Let's trace ...

## Intersect:

eE

## Remove eE



Event:  $E$

# The Algorithm

FindIntersections( $S$ )

**Input:** a set  $S$  of line segments

**Output:** all intersection points and for each intersection the segment containing it.

1.  $Q \leftarrow \emptyset$  // initialize an empty event queue
2. Insert the segment endpoints into  $Q$  // store with every left endpoint  
// the corresponding segments
3.  $T \leftarrow \emptyset$  // initialize an empty status structure
4. **while**  $Q \neq \emptyset$
5.     **do** extract the next event point  $p$
6.          $Q \leftarrow Q - \{p\}$
7.         HandleEventPoint( $p$ )

# Handling Event Points

Status updates (1) – (3) presented earlier.

**Degeneracy:** several segments are involved in one event point (tricky).

