COSC 262
Algorithms

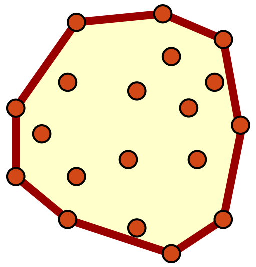
1 Computational Geometry

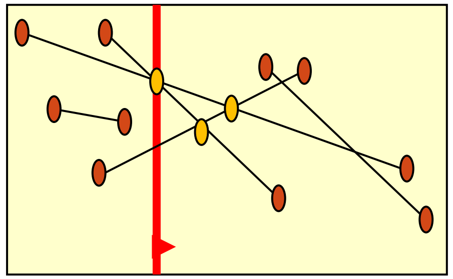
1.1 Convex Hulls

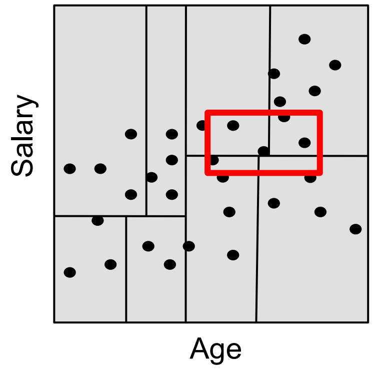
R. Mukundan
mukundan@canterbury.ac.nz
Department of Computer Science and Software Engineering
University of Canterbury

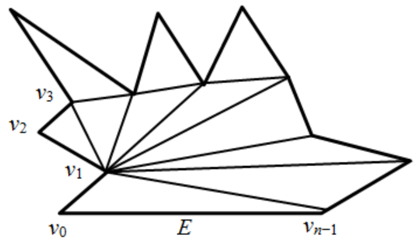
Introduction

- Computational Geometry (or Geometric Algorithms) deals with the design and analysis of efficient algorithms for
  - problems involving geometric inputs such as points, lines, triangles, pixels.
  - problems that can be solved in the geometric domain (eg. database search)
- The primary emphasis is on **computational complexity**.
- Several applications in Computer Graphics, Robotics, Geographical Information Systems CAD/CAM, Visualization and Computer Vision.





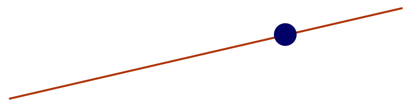




Introduction

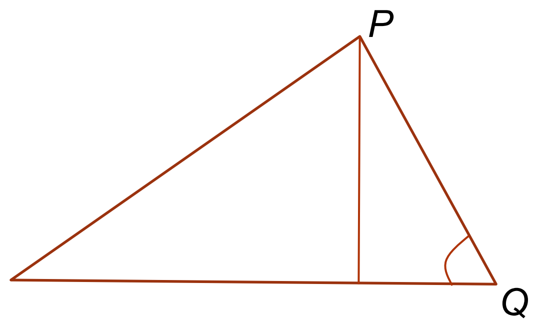
Common characteristics of computational geometry problems:

- **Error thresholds**: Usually required in comparisons between geometrical elements.

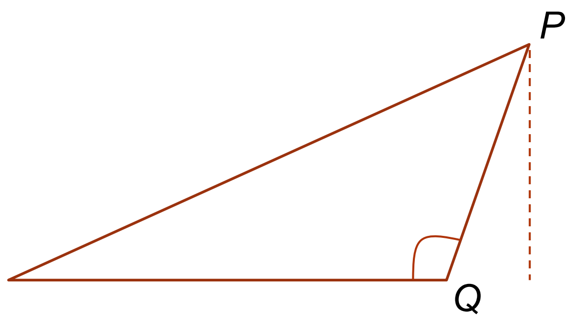


Does the point lie on the line?

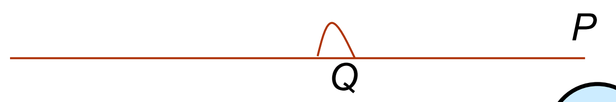
- **Special cases**: Coincident points or lines, Several lines intersecting at a point, Vertical lines, Parallel and overlapping lines.
- **Degenerate cases**: A line segment with coinciding end points, A triangle with zero area.



Division of a triangle



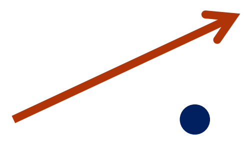
Special Case

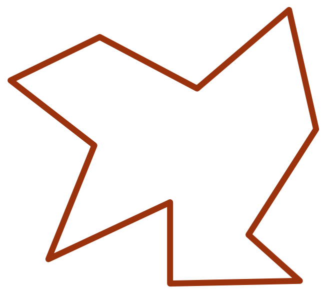


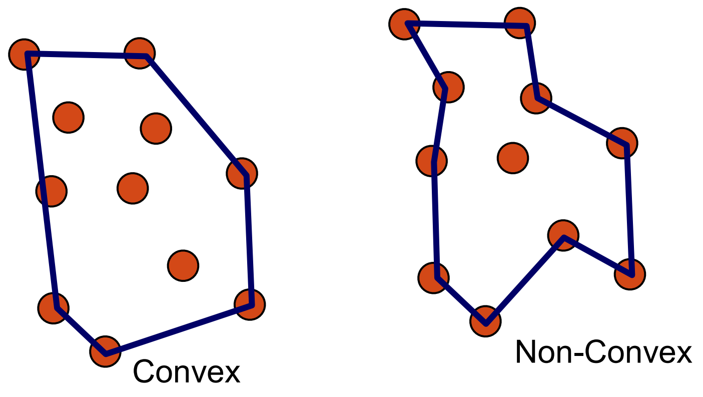
Degenerate Case

Lecture Outline

- Line Segments
  - Definition of a directed line segment
  - Orientation of 3 points
  - Line segment intersection
- Polygons
  - Convex polygons
  - Point inclusion tests
  - Simple closed paths
  - Angle approximation
- Convex Hulls
  - Properties and applications
  - Gift-Wrap algorithm
  - Graham-Scan algorithm



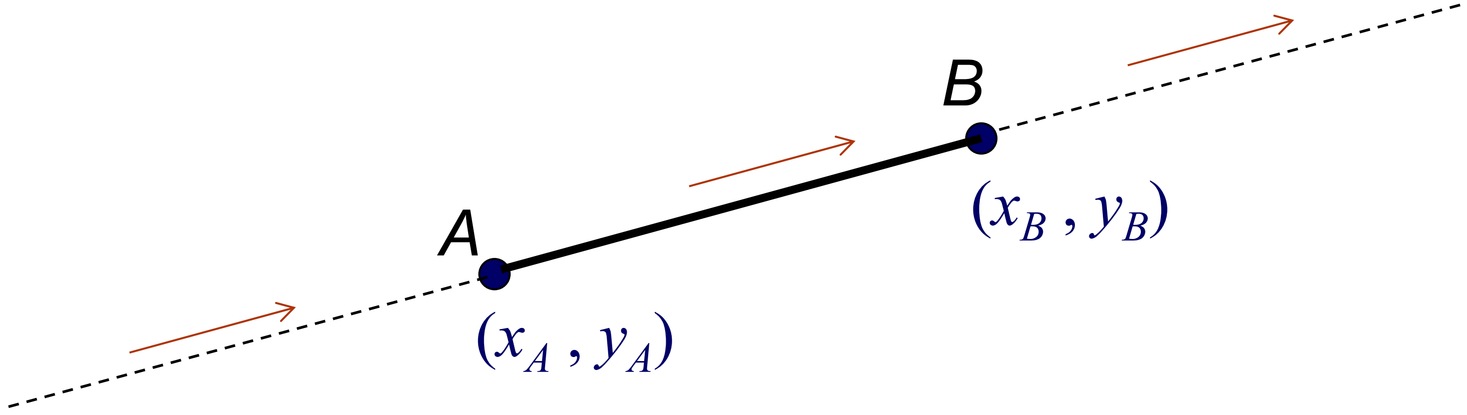




Region Boundaries

Directed Line

A directed line is defined by an **ordered pair** of points (*A*, *B*), *A*≠*B*. It is an infinite line having a specific direction ("from *A* to *B*").



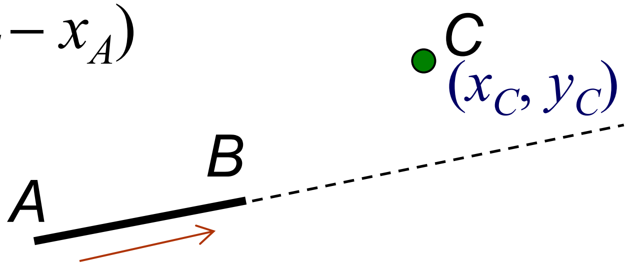
We can associate the following function with the directed line:

$$L_{AB}(\mathbf{x}, \mathbf{y}) = (x_B - x_A)\,(\mathbf{y} - y_A) - (y_B - y_A)\,(\mathbf{x} - x_A)$$

Directed Line Function

For any point $C = (x_C, y_C)$, the line function $L_{AB}(C)$ gives a value

$$L_{AB}(C) = (x_B - x_A)\,(\boldsymbol{y_C} - y_A) - (y_B - y_A)\,(\boldsymbol{x_C} - x_A)$$



Python:

```
def lineFn(ptA, ptB, ptC):
    return (
          (ptB[0]-ptA[0])*(ptC[1]-ptA[1]) -
          (ptB[1]-ptA[1])*(ptC[0]-ptA[0]) )
```

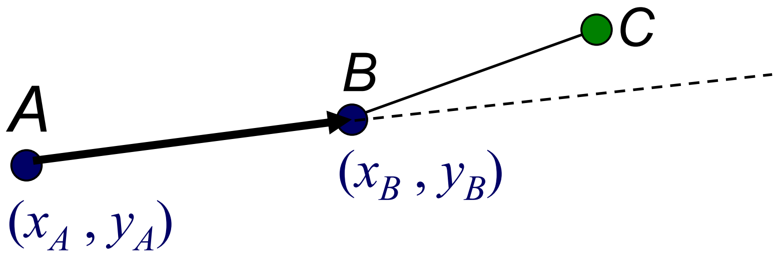
If the point *C* lies on the infinite line defined by *A*, *B*, $L_{AB}(C) = 0$.

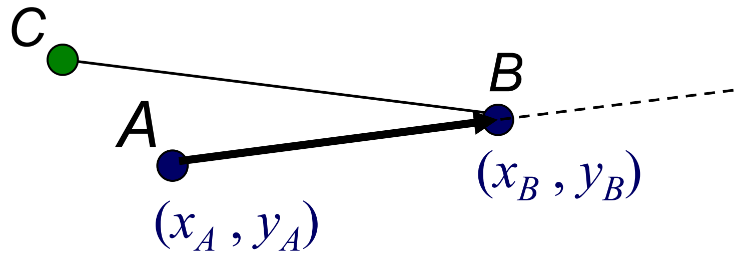
Directed Line : Orientation of 3 Points

If the point *C* lies on the **left** of the directed line *AB*, then

$$L_{AB}(C) = (x_B - x_A)(y_C - y_A) - (y_B - y_A)(x_C - x_A) > 0$$

The points *A, B, C* make a **counter-clockwise** (ccw) turn.

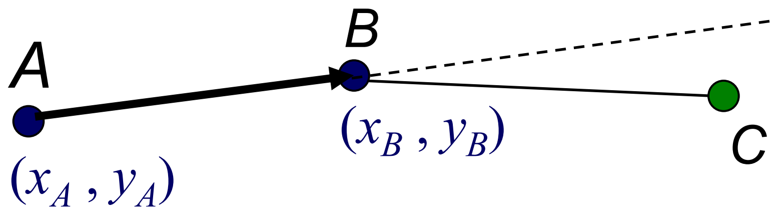


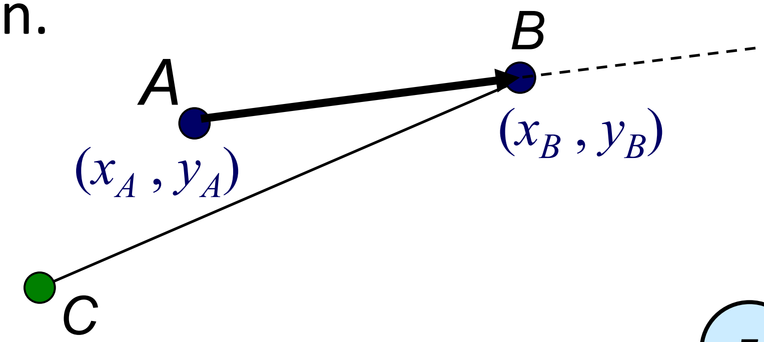


If the point *C* lies on the **right** of the directed line *AB*, then

$$L_{AB}(C) = (x_B - x_A)(y_C - y_A) - (y_B - y_A)(x_C - x_A) < 0$$

The points *A, B, C* make a **clockwise** turn.

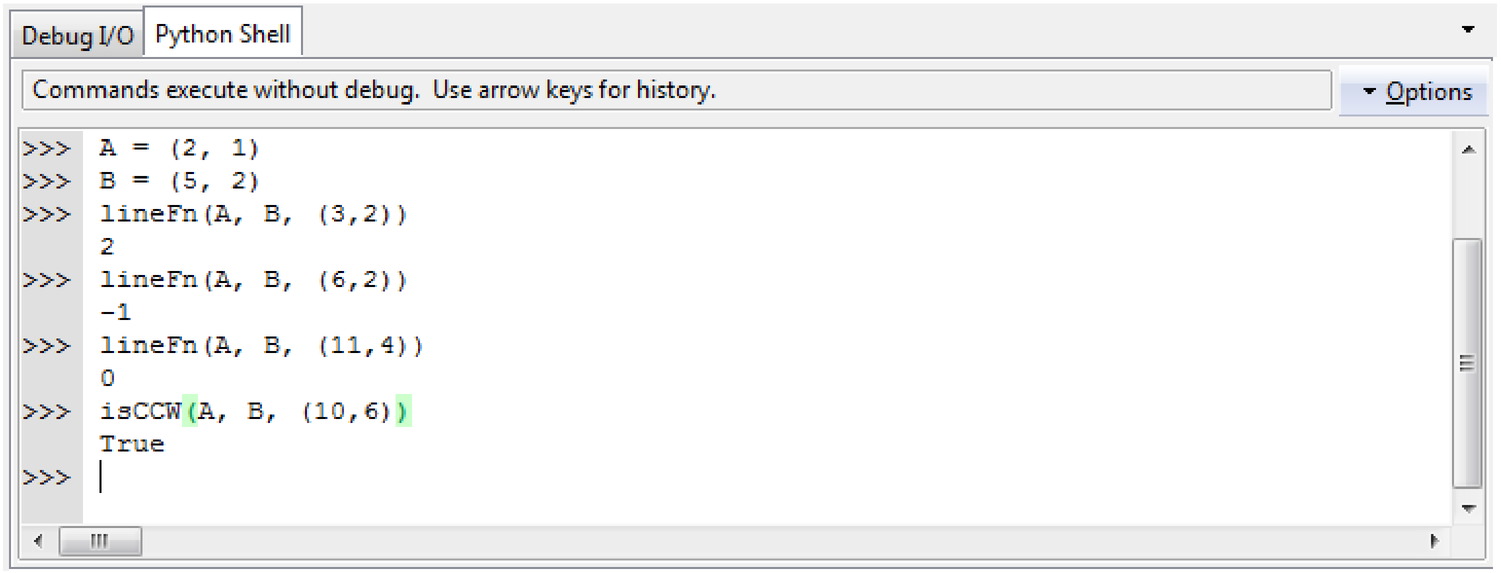




Sample Implementation: $L_{AB}(C)$

```
def lineFn(ptA, ptB, ptC):
    return (
        (ptB[0]-ptA[0])*(ptC[1]-ptA[1]) -
        (ptB[1]-ptA[1])*(ptC[0]-ptA[0]) )

def isCCW(ptA, ptB, ptC):
    return lineFn(ptA, ptB, ptC) > 0
```



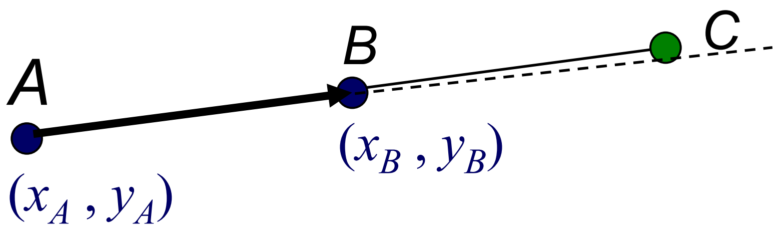
Commands execute without debug. Use arrow keys for history.

```
>>> A = (2, 1)
>>> B = (5, 2)
>>> lineFn(A, B, (3,2))
    2
>>> lineFn(A, B, (6,2))
    -1
>>> lineFn(A, B, (11,4))
    0
>>> isCCW(A, B, (10,6))
    True
>>> 
```

Troublesome Triples

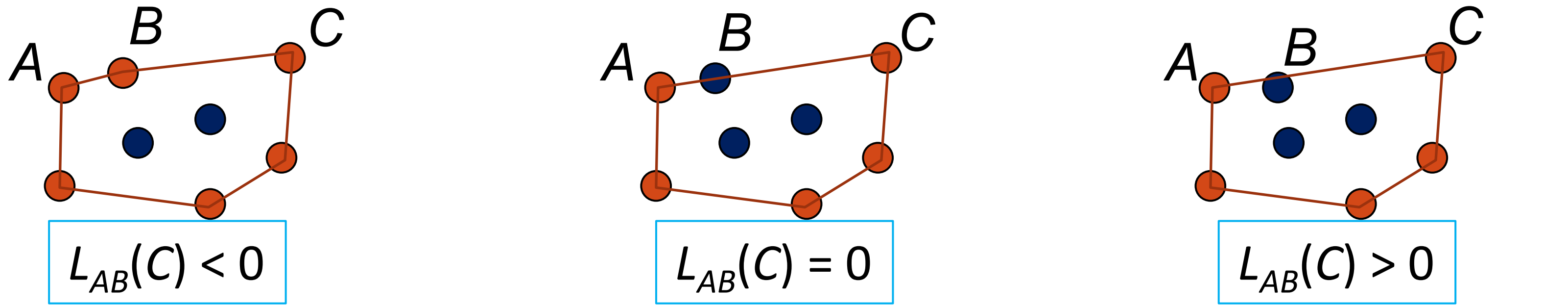
In many computational geometry algorithms, the collinearity of 3 points leads to a special case that should be handled carefully and correctly.

- Numerical precision of computations play an important role.
- An error threshold is often used.
- Proper decisions must be made in such situations.



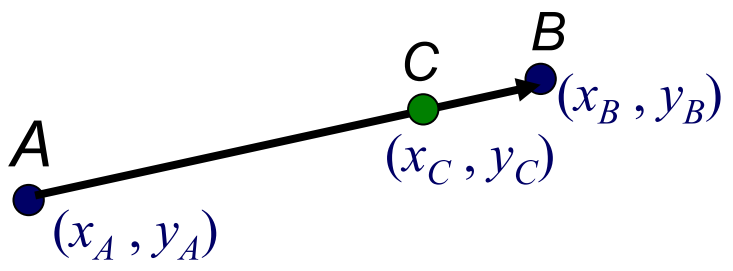
Is $L_{AB}(C) = 0$?

Convex hull example:



Point on Line Segment

A point *C* lies on the line **segment** *AB* if and only if the following conditions are satisfied:



$L_{AB}(C) = 0$

and

$\min(x_a, x_b) \le x_c \le \max(x_a, x_b)$

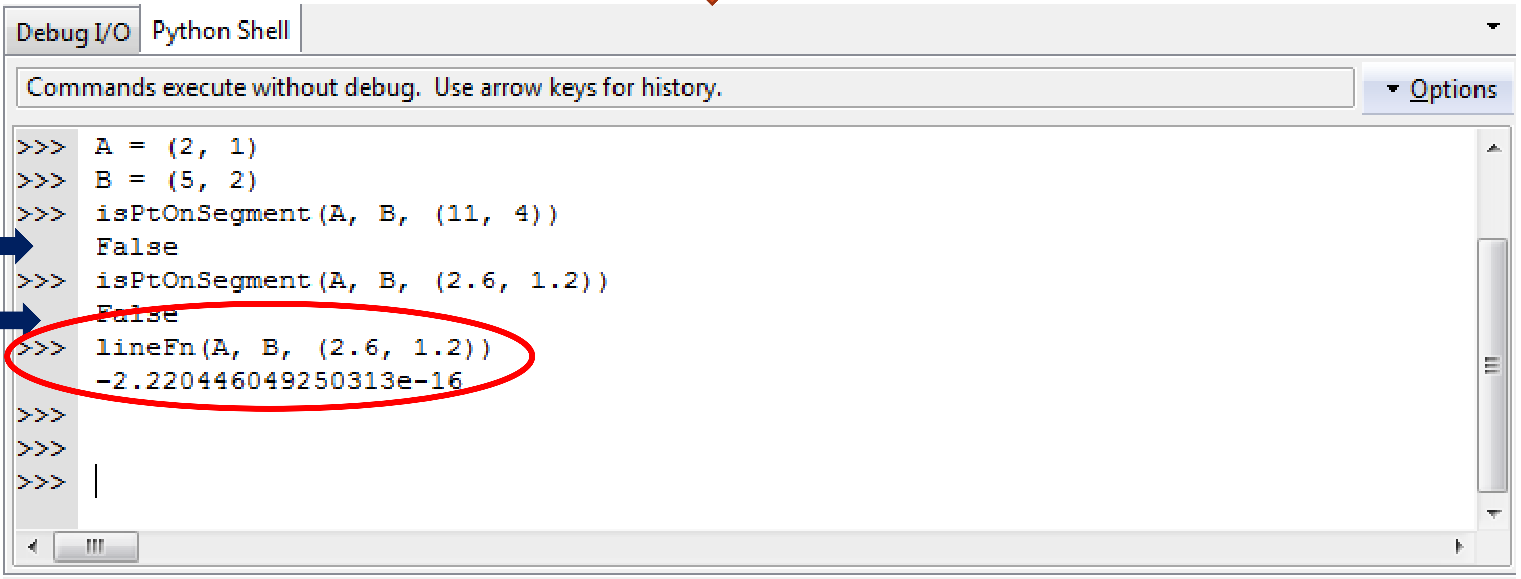
and

$\min(y_a, y_b) \le y_c \le \max(y_a, y_b)$

Point on Segment

Use a small threshold `1.e-6`

```
def isPtOnSegment(ptA, ptB, ptC):
    return (
        lineFn(ptA, ptB, ptC) == 0
        and min(ptA[0], ptB[0]) <= ptC[0]
        and ptC[0] <= max(ptA[0], ptB[0])
        and min(ptA[1], ptB[1]) <= ptC[1]
        and ptC[1] <= max(ptA[1], ptB[1]) )
```



Commands execute without debug. Use arrow keys for history.

```
>>> A = (2, 1)
>>> B = (5, 2)
>>> isPtOnSegment(A, B, (11, 4))
False
>>> isPtOnSegment(A, B, (2.6, 1.2))
False
>>> lineFn(A, B, (2.6, 1.2))
-2.220446049250313e-16
>>>
>>>
>>>
```

✓

✗

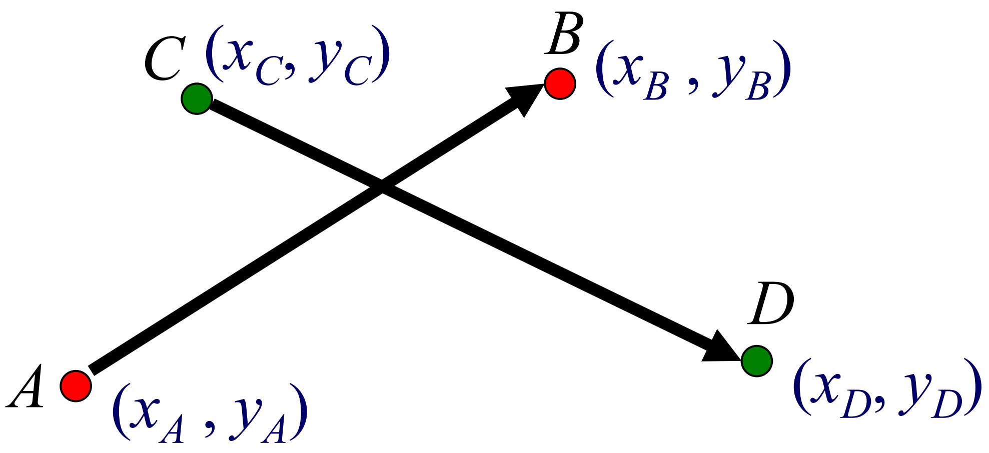
Line Segment Intersection Test

Two line segments *AB* and *CD* as shown below intersect only if

Points C, D are on different sides of line *AB*: $L_{AB}(C)\, L_{AB}(D) < 0$

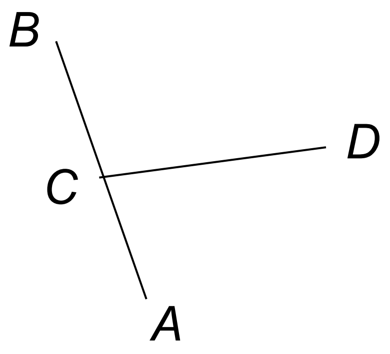
and

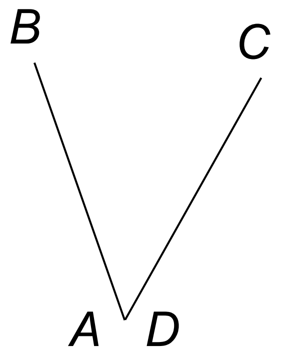
Points A, B are on different sides of line *CD*: $L_{CD}(A)\, L_{CD}(B) < 0$



Line Segment Intersection Test

Special Cases: An end point of one line segment lies on the other segment

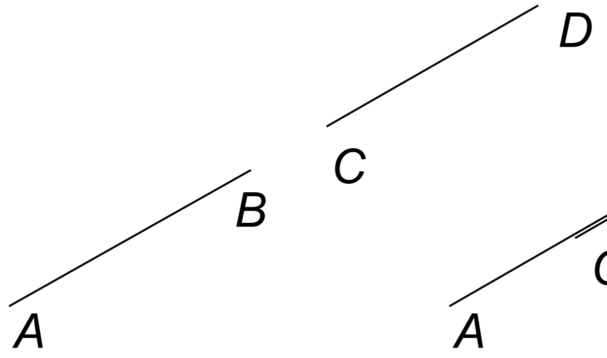


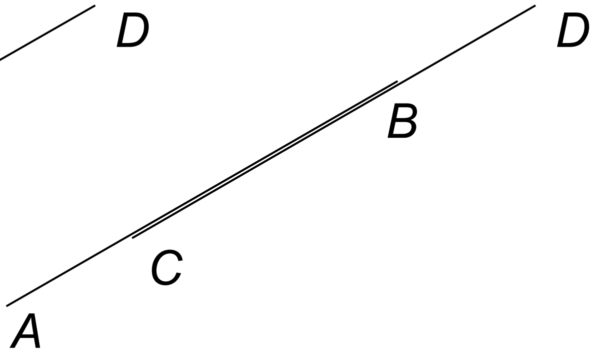


$L_{AB}(C)\,L_{AB}(D) = 0$
$L_{CD}(A)\,L_{CD}(B) < 0$

$L_{AB}(C)\,L_{AB}(D) = 0$
$L_{CD}(A)\,L_{CD}(B) = 0$

Degenerate Case: All four points are collinear

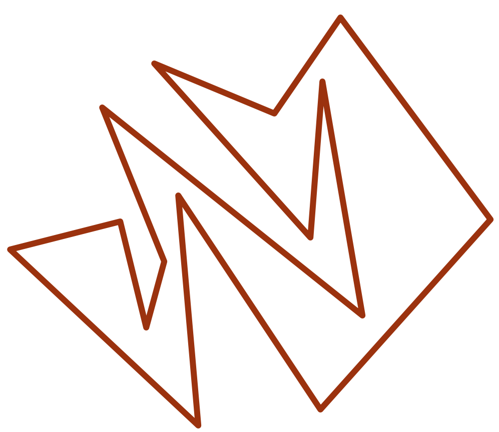




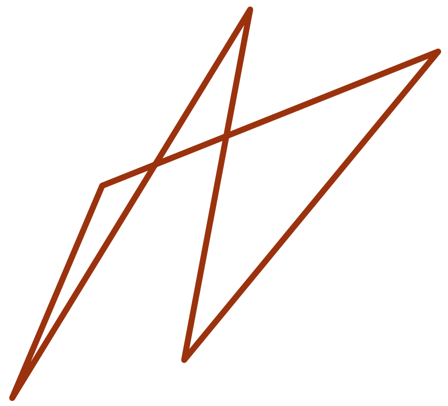
In both the cases shown here,
$L_{AB}(C)\,L_{AB}(D) = 0$
$L_{CD}(A)\,L_{CD}(B) = 0$

Simple Polygons

A simple polygon is a polygon whose edges do not intersect



Simple polygon

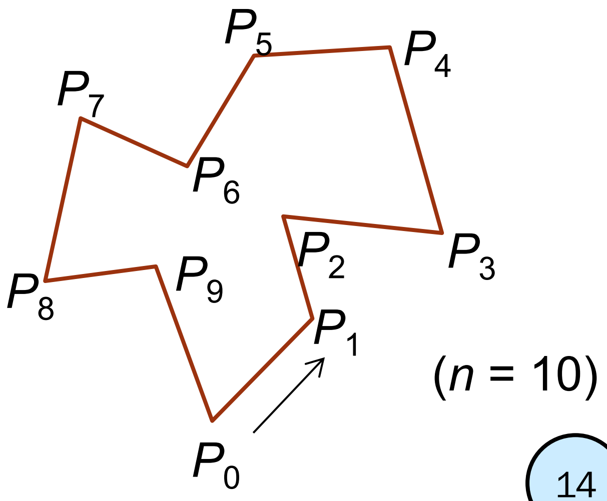


Self-intersecting polygon

A simple polygon can be represented using an ordered list of vertices (each vertex is a tuple (x, y)) :

$[P_0, P_1, \ldots, P_{n-1}]$

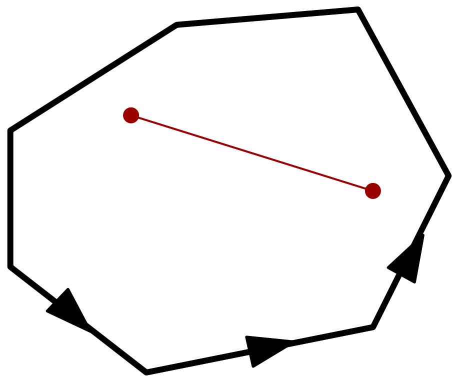
We usually perform an ***anti-clockwise*** traversal of the vertex list.



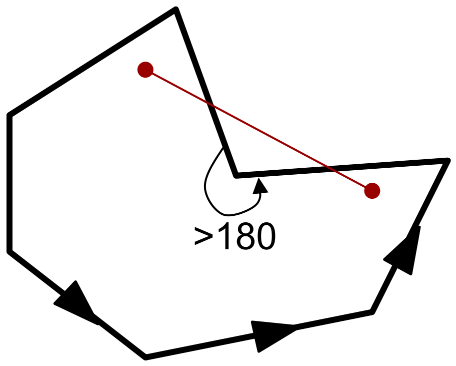
Convex Polygons

A convex polygon satisfies the following properties:

- Any line segment connecting two interior points lies entirely within the polygon
- Every **interior angle** is less than or equal to 180 degs.
- Every **anticlockwise traversal** of a convex polygon either continues straight, or **turns left** at every vertex.



Convex



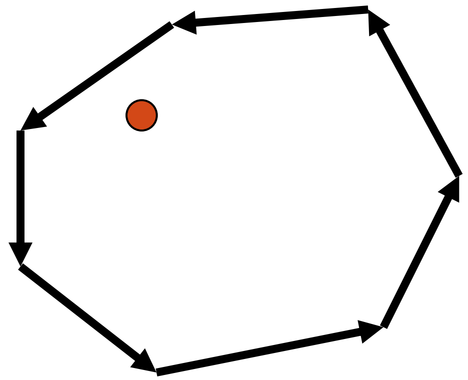
Not Convex

Point Inclusion Test

"Point Inclusion Test" refers to the problem of determining whether a given point is inside a polygon.

The problem is also referred to as the "Point-in-Polygon" (PIP) problem.

A point lies inside a **convex polygon** if and only **if it is on the left side of every edge for an anticlockwise traversal** of the polygon.

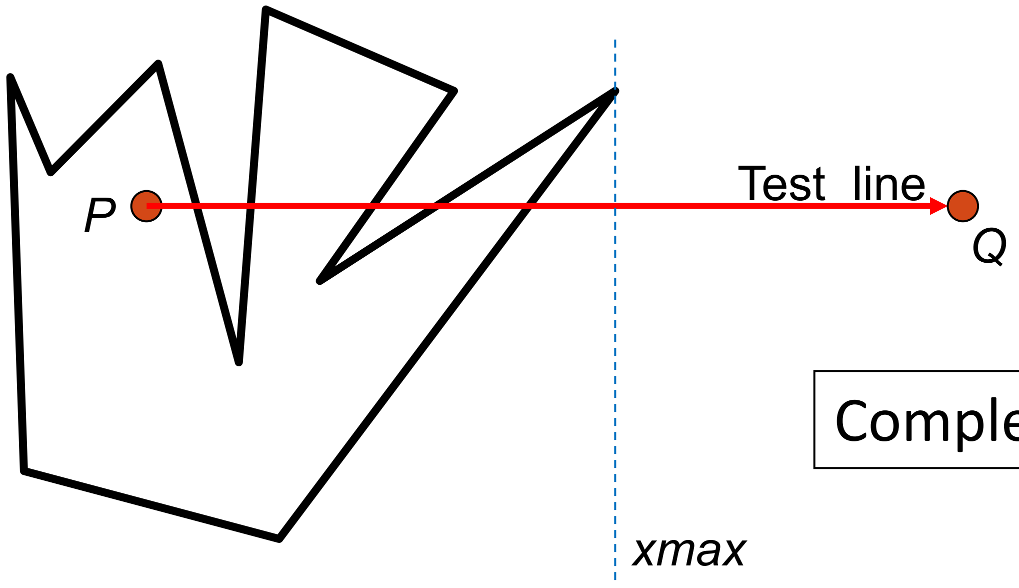


Complexity = $O(n)$

Note: More efficient $O(\log n)$ algorithms also exist!

Point Inclusion Test

A point *P* lies inside a simple polygon if and only if a semi-infinite horizontal line emanating from the point **intersects the edges an odd number of times.**



Number of edge intersections:
k = 5

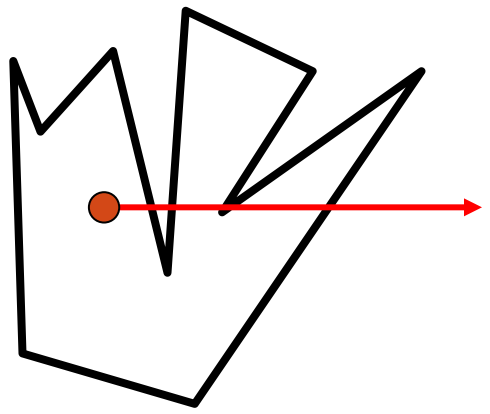
Complexity = *O*(*n*)

A horizontal "semi-infinite" line from *P* (x_p, y_p) is actually a finite length segment *PQ* where the coordinates (x_q, y_q) of *Q* are chosen such that

$y_q = y_p, \qquad x_q > x_{max}$

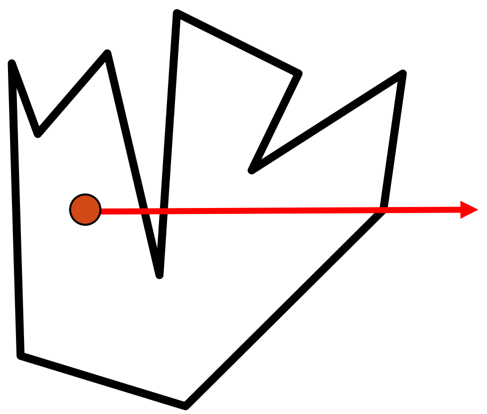
Point Inclusion Test

Special Cases:



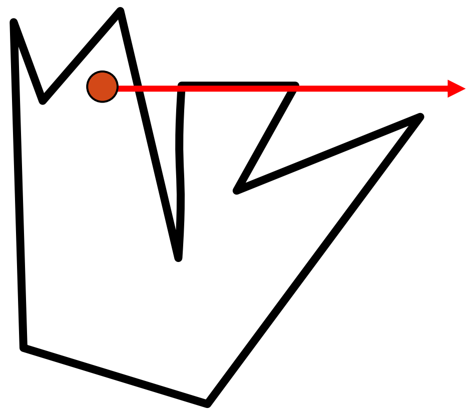
The line passes through a vertex.

Should the intersection at the vertex be counted twice?



The line passes through a vertex.

Should the intersection at the vertex be counted only once?

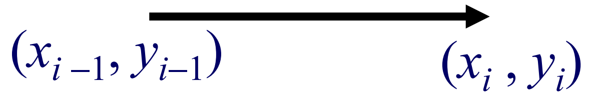


The line and an edge of the polygon overlap.

Point Inclusion Test

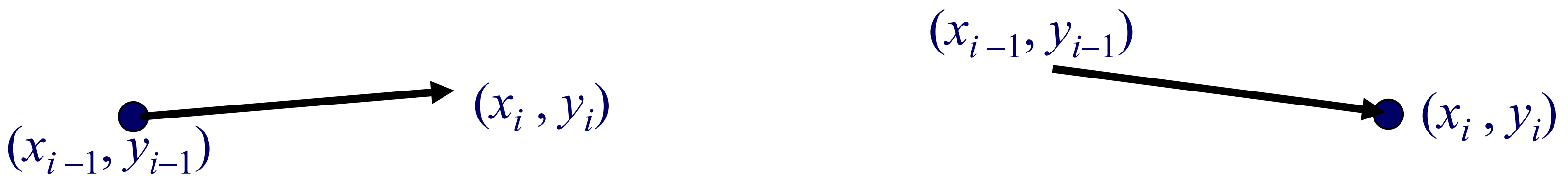
Several solutions exist to handle the special cases shown on previous slide. One approach is discussed below:

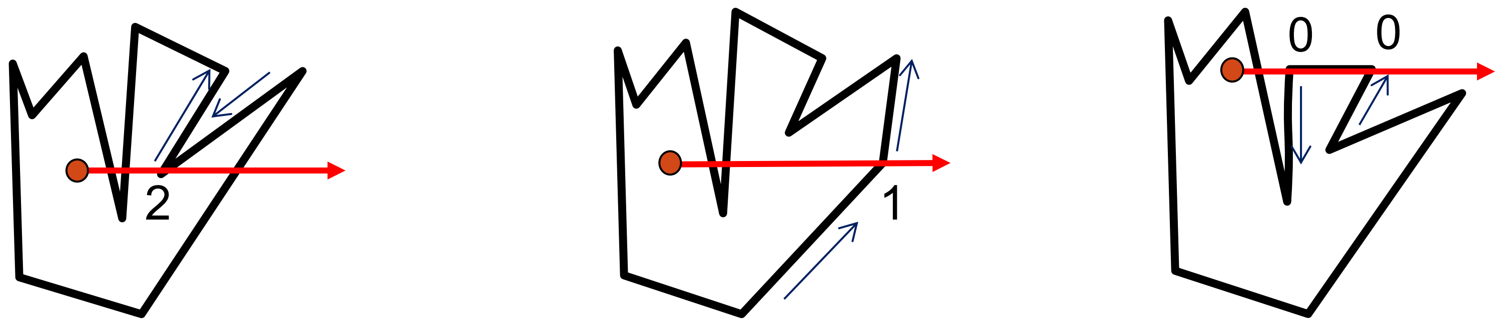
Ignore horizontal edges ($y_i = y_{i-1}$)



If the edge is directed upward then it contains the start vertex.

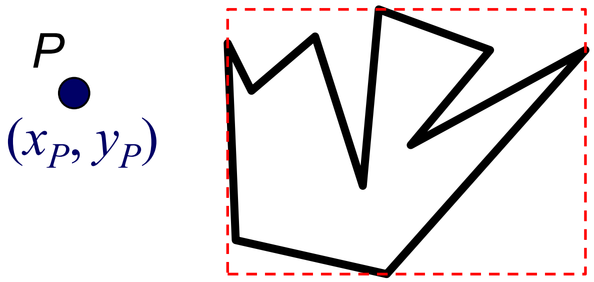
If the edge is directed downward, it contains the end vertex.





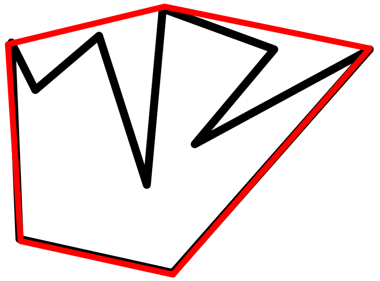
Further Improvements

- We should try to minimise redundant computations
- Is it necessary to perform line-edge comparisons for every point on the plane containing the polygon?
  - A simple pre-processing step using a *bounding volume* can help in determining if a point is completely outside a rectangular region enclosing the polygon.



P is outside the polygon if $(x_p < x_{min})$ or $(x_p > x_{max})$ or $(y_p < y_{min})$ or $(y_p > y_{max})$

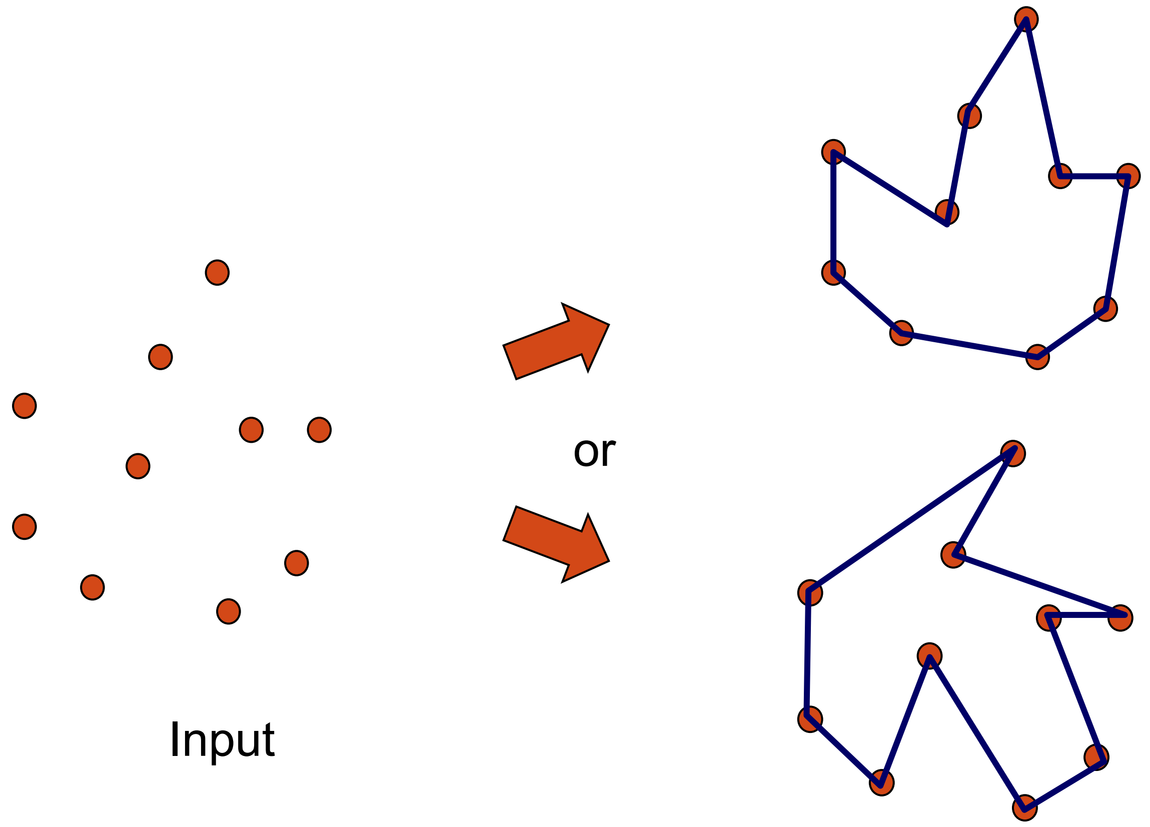
  - A better bounding volume:



Simple Closed Path: Joining the dots

Converting a point set to a **simple polygon** (see Slide 14):

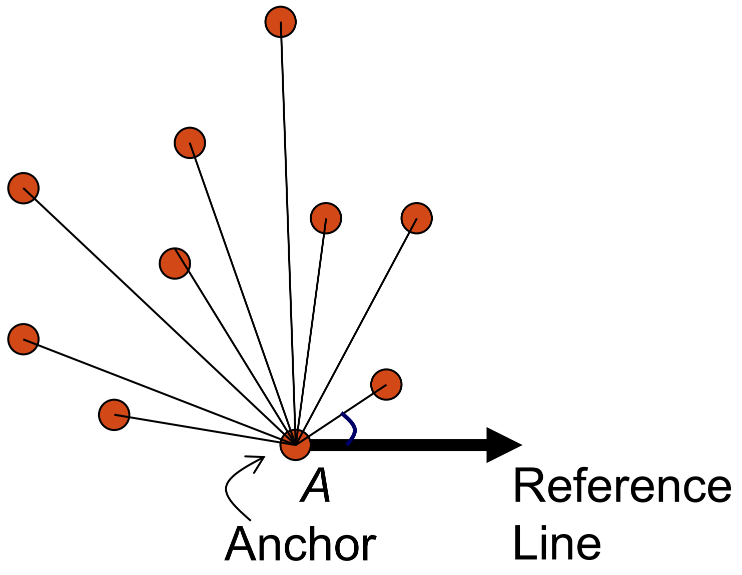
Given a set of *n* points on a plane, compute a simple closed path (or a closed polygonal line) that passes through all the points and does not intersect itself.

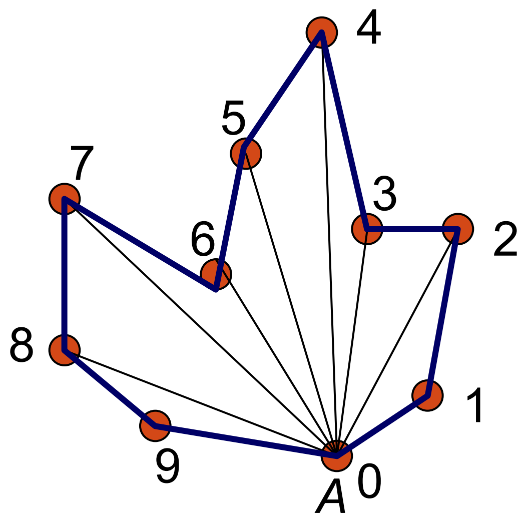


Simple Closed Path

Algorithm:

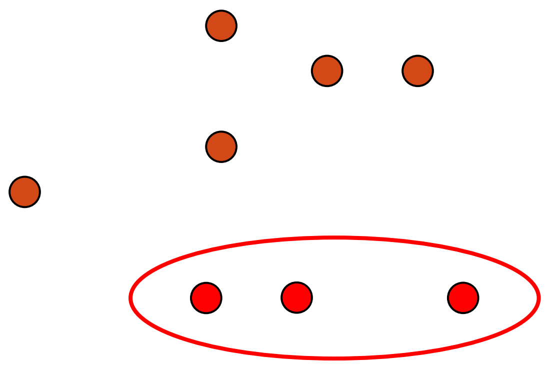
- Select the point *A* with minimum *y* value as the starting point (**anchor**).
- For each point, compute the angle between the line segment from *A* to that point and a horizontal (**reference**) line through *A*.
- Sort the points according to the angle.
- Connect adjacent points in the sorted list



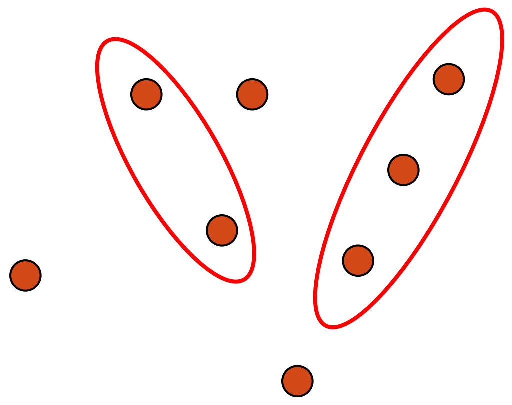


Simple Closed Path: Special Cases

- Several points with the same *y*-minimum value:

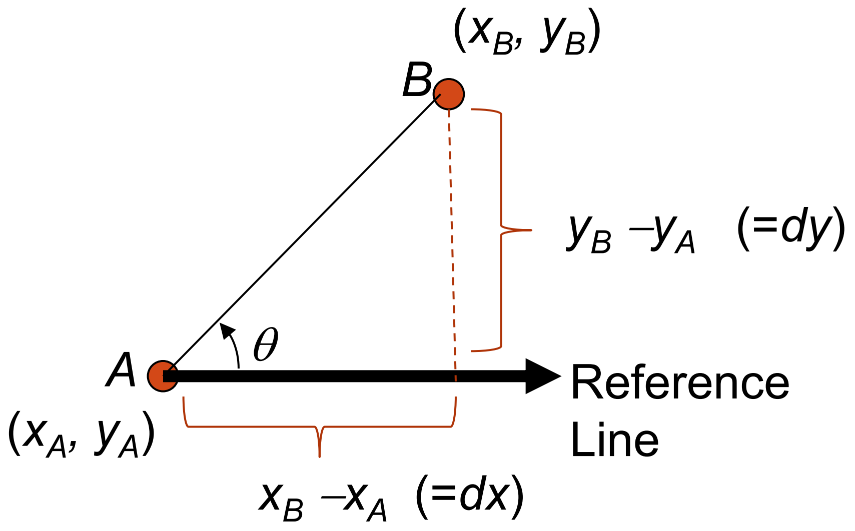


- Several points making equal angle with the reference line.



Simple Closed Path

- The solution for the simple closed path takes *O*(*n*log*n*) time.
- It requires the computation of the angle between a line and a horizontal line using the inverse trigonometric function $\tan^{-1}$.

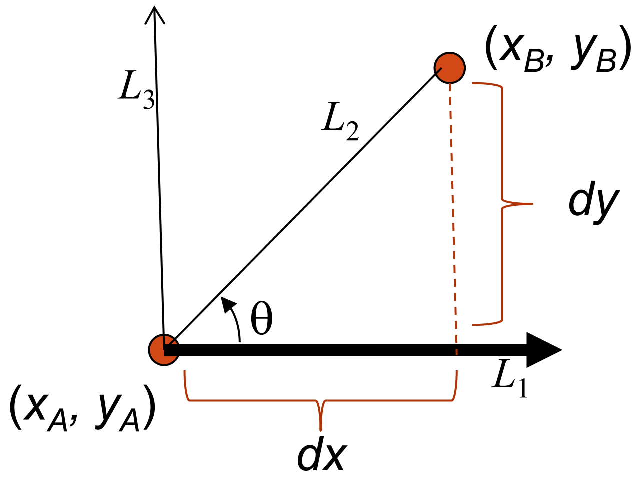


$$\theta = \tan^{-1}\left(\frac{y_B - y_A}{x_B - x_A}\right)$$
$$= \tan^{-1}\left(\frac{dy}{dx}\right)$$

- In the previous problem, the angle is used only for sorting. Therefore we could use any parameter that increases with the angle. We do not actually require the exact value of the angle!

An Approximation for θ

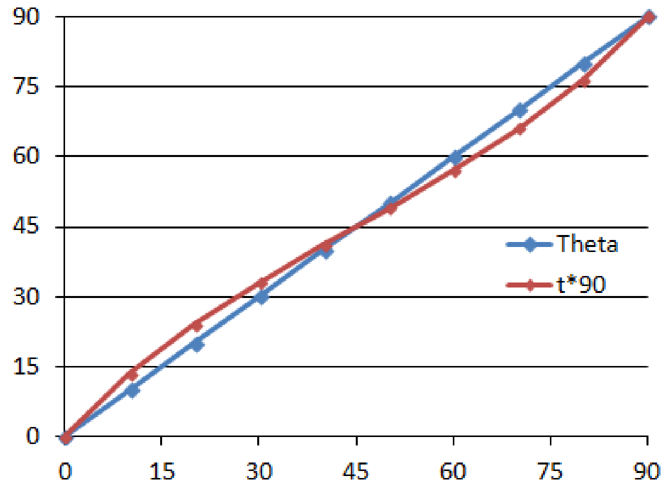
We can construct a simple **approximation** for θ that does not require evaluating trigonometric functions.



Define $$t = \frac{dy}{|dx| + |dy|}$$

Point on line L_1	$dy = 0$	$\theta = 0$	$t = 0$
Point on line L_2	$dy = dx$	$\theta = 45°$	$t = 0.5$
Point on line L_3	$dx = 0$	$\theta = 90°$	$t = 1$

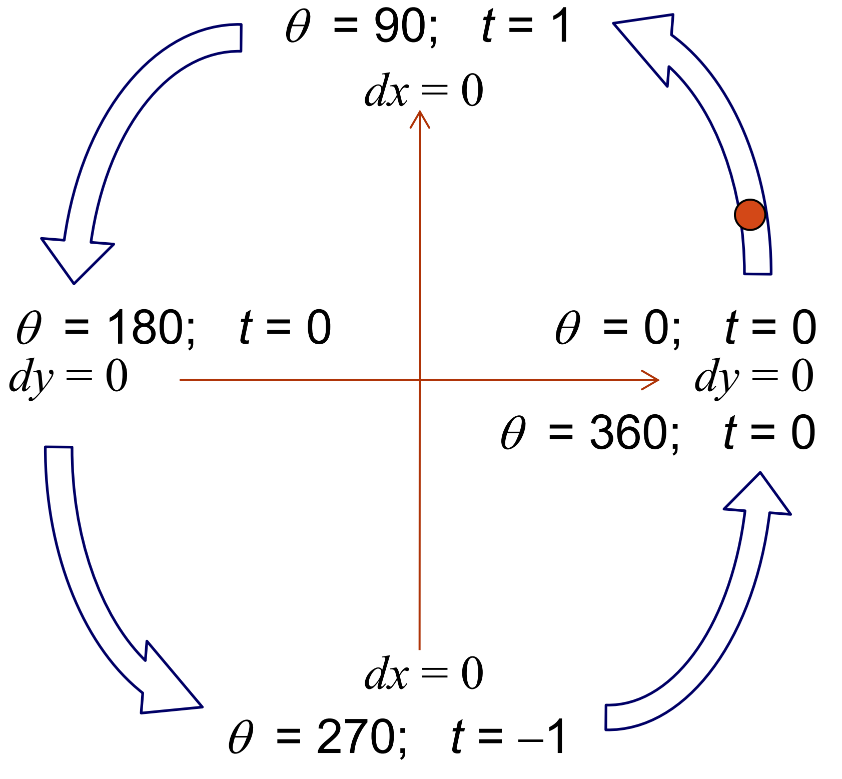
$\theta \approx t * 90$



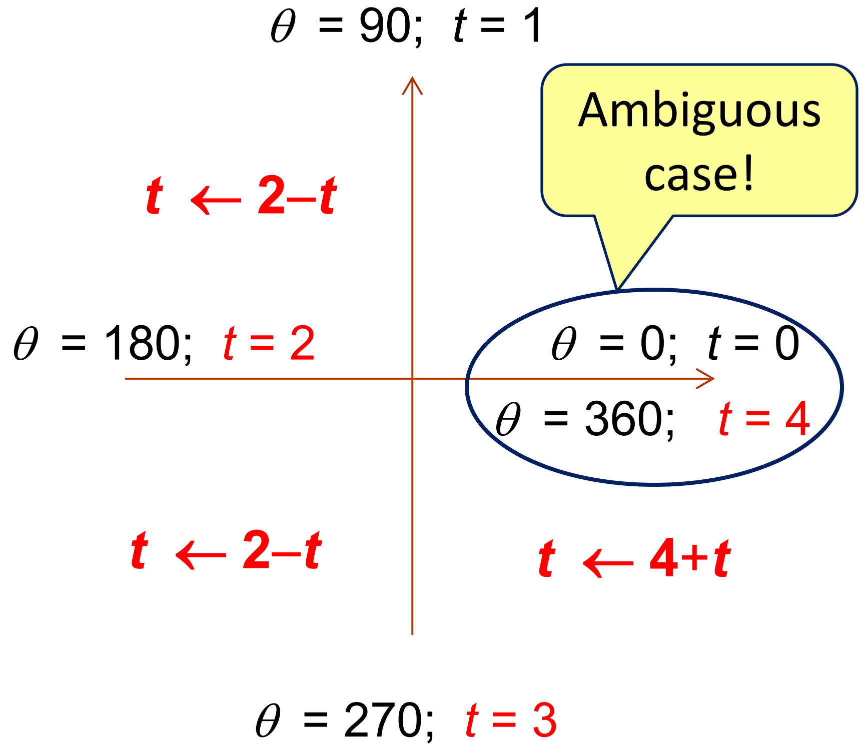
This approximation is valid only in the range $0 \le \theta \le 90°$

What about other values of θ?

An Approximation for θ



$$t = \frac{dy}{|dx| + |dy|}$$



As θ increases from 0 to 360 degs,
t increases from 0 to 4.
Therefore, $\theta \approx t * 90$.

An Approximation for θ (Code)

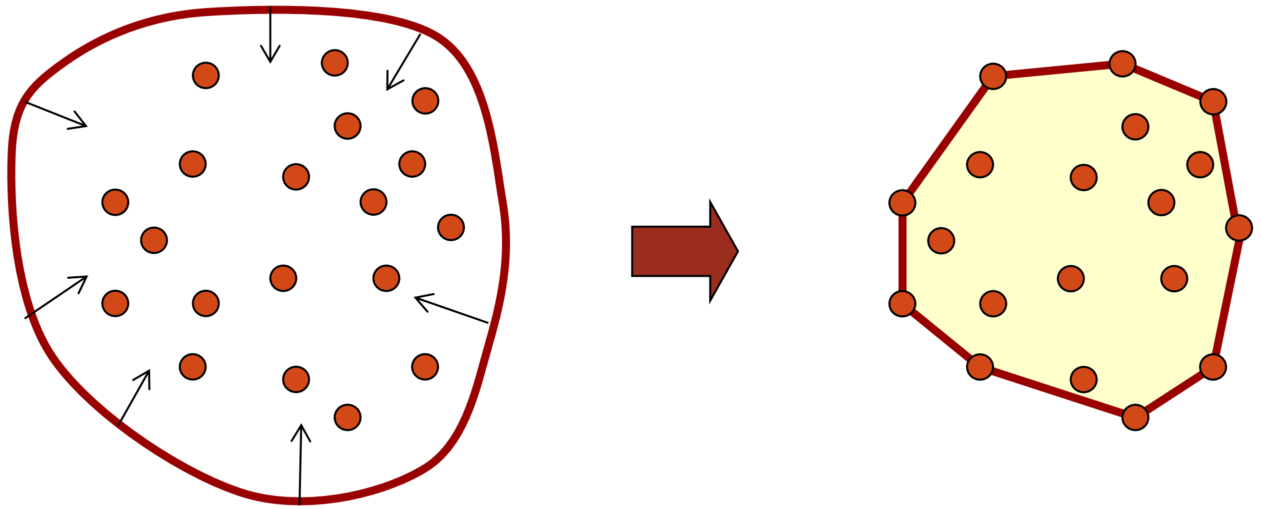
```
def theta(pointA, pointB):
    ''' Computes an approximation of the angle between
        the line AB and a horizontal line through A.
    '''
    dx = pointB[0] - pointA[0]
    dy = pointB[1] - pointA[1]
    if abs(dx) < 1.e-6 and abs(dy) < 1.e-6:
        t = 0
    else:
        t = dy/(abs(dx) + abs(dy))

    if  dx < 0:
        t = 2 - t
    elif  dy < 0:
        t = 4 + t
    return t*90
```

Degenerate case!

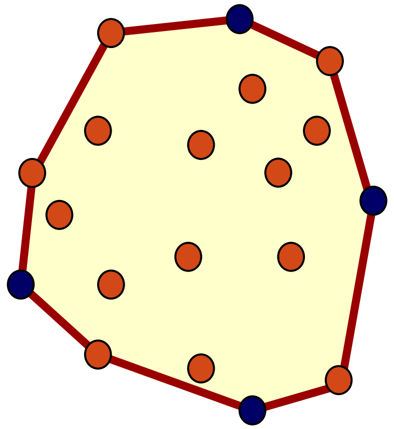
Convex Hulls

- Given a finite set of points $S = \{P_0, P_1 \ldots P_{n-1}\}$, the convex hull of S is the smallest convex polygon enclosing all of the points.
- Visualized as the shape of a stretched rubber band around the points so that every point lies within the band.



Properties of Convex Hulls

- A convex hull of a set of points *S* is a *unique* convex polygon that contains every point of *S*, and whose vertices are only points in *S.*
- A convex hull is the intersection of all convex polygons containing *S*.
- A convex hull is the union of all triangles determined by points of *S*.
- Points in *S* with minimum *x*, maximum *x*, minimum *y*, and maximum *y* coordinates are all vertices of the convex hull of *S*.

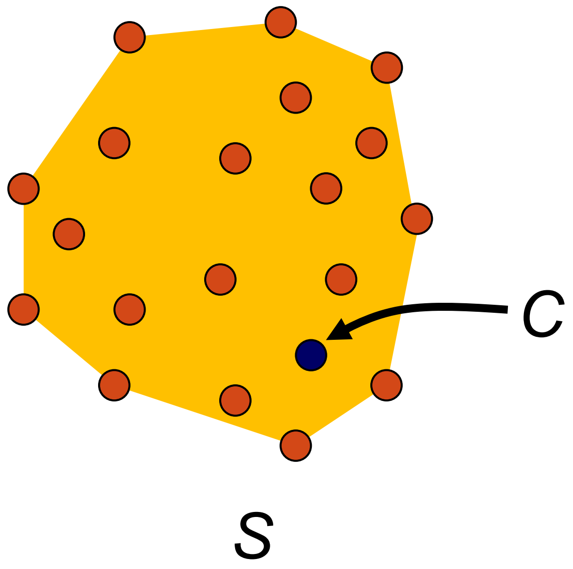


Properties of Convex Hulls

If we construct a new point *C* using a *convex combination* of points in *S*, then *C* lies inside the convex hull of *S*.

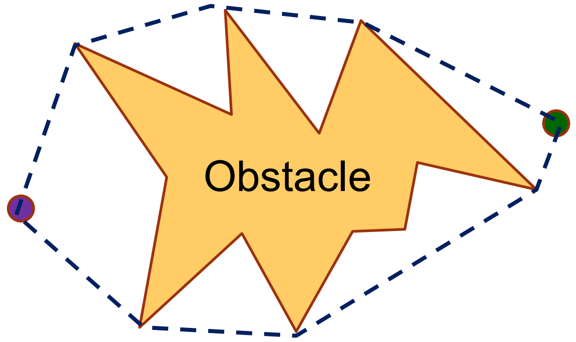
$$C = w_0 P_0 + w_1 P_1 + w_2 P_2 + \ldots + w_{n-1} P_{n-1}$$

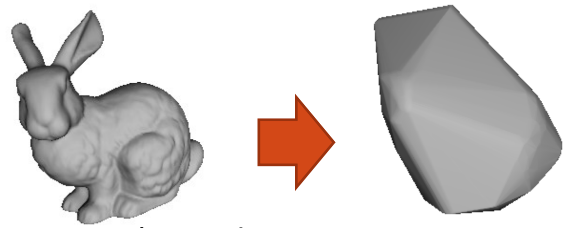
$w_i \in [0, 1]$ for all i, and $w_0 + w_1 + \ldots + w_{n-1} = 1.$

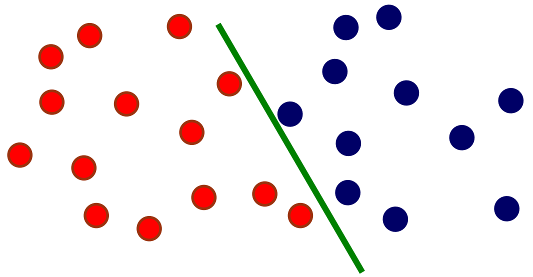


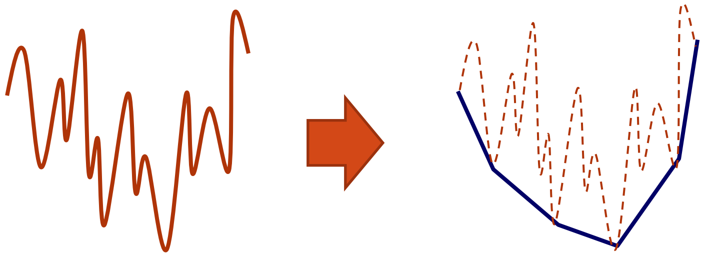
Applications of Convex Hulls

- Robotics
  - Path planning
  - Object recognition
- Graphics
  - Bounding volumes for ray tracing and collision detection
- Geometry Algorithms
  - Voronoi diagrams
  - Delaunay triangulations
- Pattern Recognition
  - Template matching









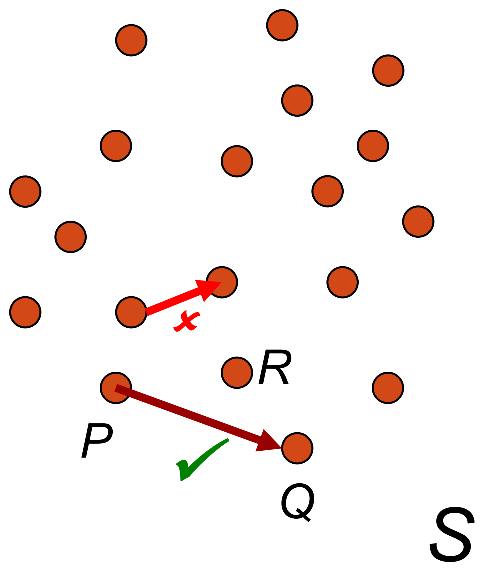
Convex Hull: Naïve Method

Given a set *S* of points,

- Construct *all possible* edges using pairs of points in *S*,
- Check if *every* point in *S* lies on the left side of the edge. If so, the edge belongs to the convex hull.

Pseudo Code:

```
For each point $P \in S$ {
    For each point $Q \in S,\ Q \neq P$ {
        For each point $R \in S,\ R \neq P,\ R \neq Q$ {
            If $L_{PQ}(R) \geq 0$ for every $R \in S$
            then $PQ$ is an edge of the convex hull
        }
    }
}
```

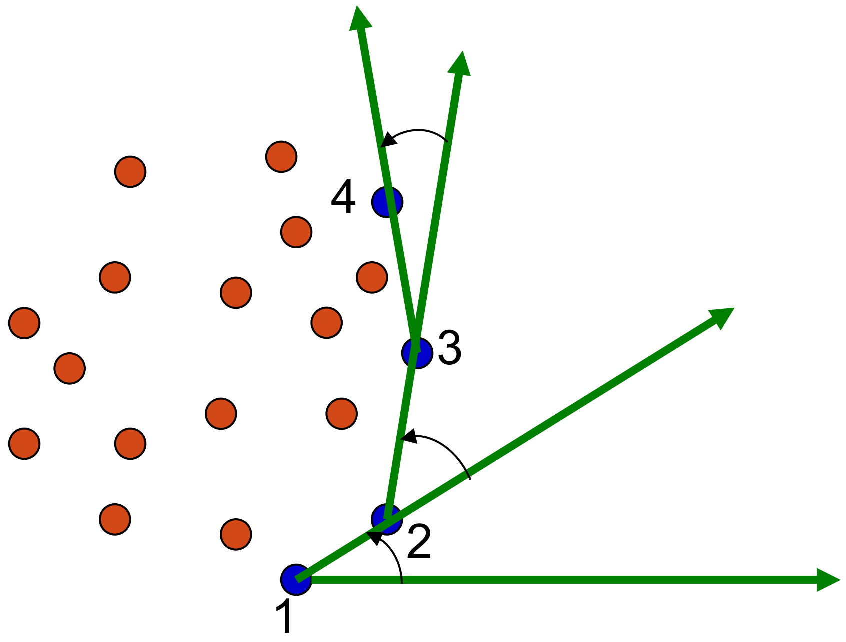


Complexity: $O(n^3)$

Convex Hull – Gift Wrap

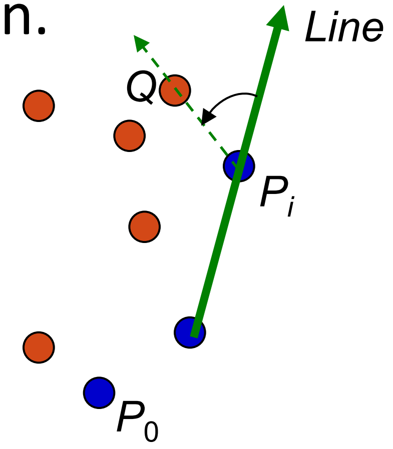
Also known as Package wrap, or Jarvis March algorithm.

Starting from the point with minimum *y*-coordinate, select the next point by identifying the edge that makes minimum angle with the current direction.



Gift Wrap (Pseudo-Code)

```
$P_0$ = point in $S$ with minimum $y$-coordinate (in case of ties, choose
        the right-most point)
$Line$ = A horizontal line through $P_0$ towards $+x$ direction.
i = 0
Repeat {
    For every point $Q \in S$, $Q \neq P_i$ {
        Compute angle between $Line$ and $P_iQ$.
        Choose $Q$ that gives minimum angle (In case of ties, choose
        the point farthest from $P_i$)
    }
    $P_{i+1} = Q$
    $Line = P_iP_{i+1}$
    i++;
} until ($P_i == P_0$)
```



Output sensitive algorithm!

Complexity: $O(mn)$

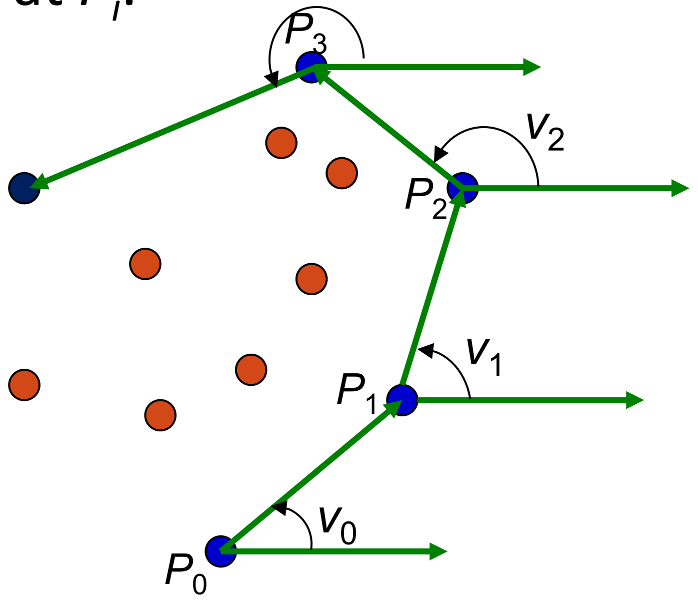
m = Number of vertices on convex hull

Gift Wrap

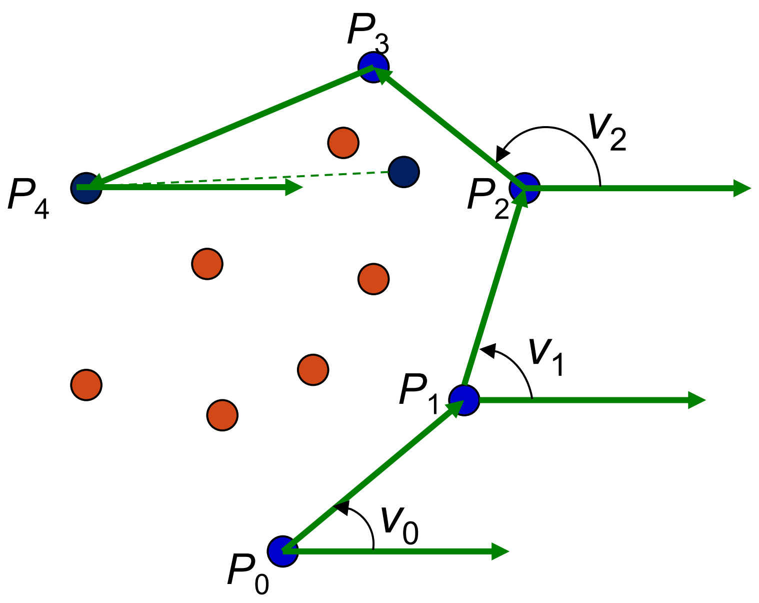
The previous method requires the computation of angles between two lines.

Can we select the point *Q* using the minimum value of the angle between the line P_iQ and a *horizontal line* through P_i?

- If so, we can use the theta function (Slide 27) to compute an approximation of the angle between P_iQ and the horizontal line at P_i.



Correct results from P_0 to P_4



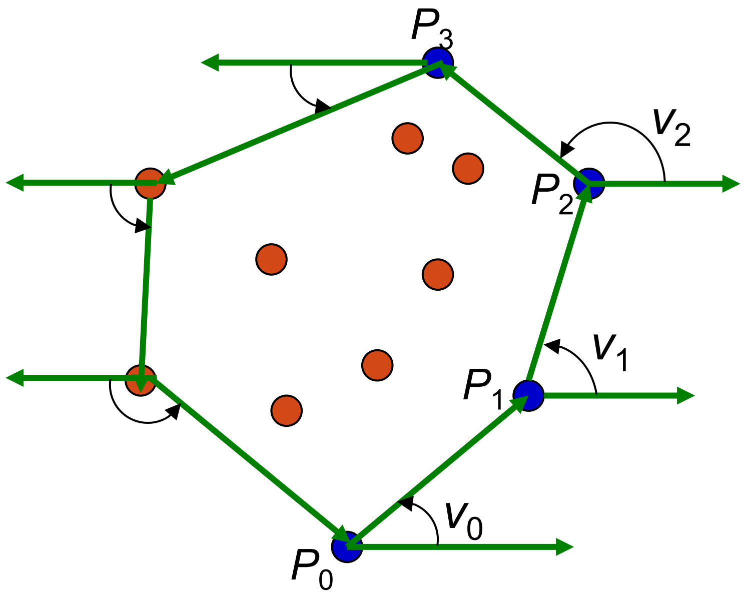
Wrong result at P_4 !

Gift Wrap

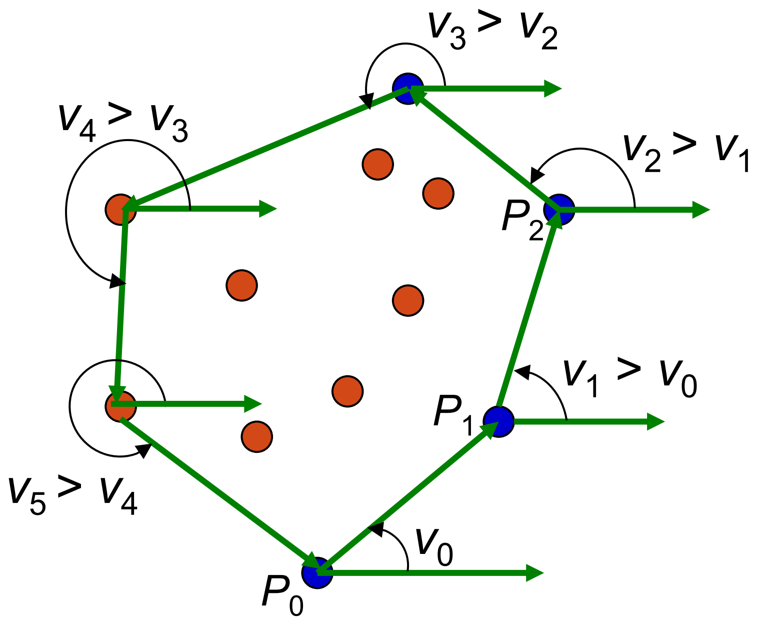
There are two possible solutions to the problem discussed on the previous slide.

Solution-1: First, complete only the right hull (P_0 to P_3). Then, complete the left hull (P_3 to P_0) using horizontal lines in −*x* direction.

Solution-2: Complete the whole hull using horizontal lines in +*x* direction, but choose the minimum angle which is *greater than the previous minimum angle.*



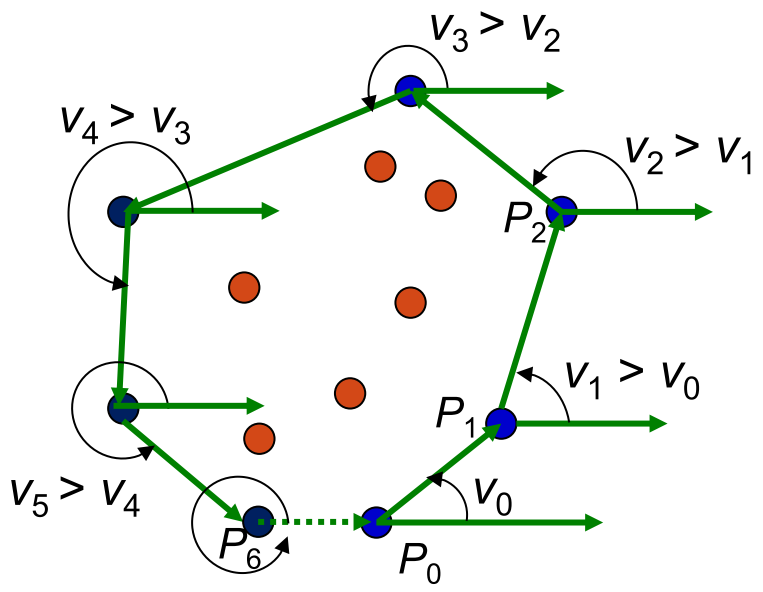
Solution-1



Solution-2

Gift Wrap: Special Case

- If there are more than one point with minimum-y coordinate, we select the rightmost point as P_0. Therefore every vertex of the convex hull should return an angle > 0.
- In the above case, we require an angle of 360 degs for the last segment (P_6P_0 in the following figure). The `theta()` function may be modified to return 360 degs if the angle is 0 degs.



Gift Wrap (Pseudo Code)

```
Input: Points $P_0, P_1, \ldots P_{n-1}$ stored in an array or list pts
Find the point $P_k$ with minimum $y$-value (In case of ties, select the right-most
   point)
$i = 0$;   $v = 0$;   pts[$n$]= $P_k$  (append $P_k$ to the list)
Repeat {
    Swap pts[$i$] and pts[$k$]
     minAngle = 361
     For ( $j = i+1,\ldots,n$) {
         angle = theta(pts[$i$], pts[$j$])
        If (angle < minAngle  and  angle > $v$  and pts[$j$] $\neq$ pts[$i$] ) {
            minAngle = angle;   $k = j$
         }
     }
     $i = i + 1$;   $v$ = minAngle
} until ($k == n$)
```

See slide 27

See slide 36
(Solution-2)

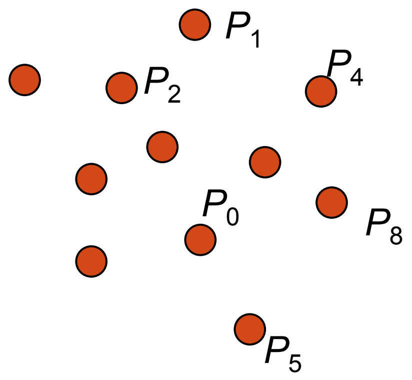
The first few steps of the algorithm are shown on the following slides.

Gift Wrap: Implementation

Input to the program: An array `pts` of points $P_0,\ldots P_{n-1}$.

pts												
	P_0	P_1	P_2	P_3	P_4	P_5						P_{n-1}

pts[0]	pts[1]	pts[2]	pts[3]								pts[n-1]
P_0	P_1	P_2	P_3	P_4	P_5						P_{n-1}



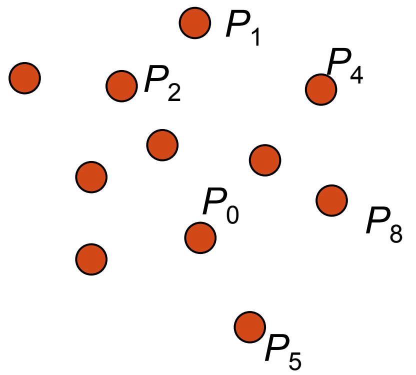
Gift Wrap: Implementation

Initialization:

- i = current position in the array = 0.
- Find the index k of the point with minimum *y*-value. In the example given below, k = 5.
- Append the list with this point.
- Angle v = 0

i

pts[0]	pts[1]	pts[2]	pts[3]								pts[n-1]	pts[n]
P_0	P_1	P_2	P_3	P_4	P_5						P_{n-1}	P_5

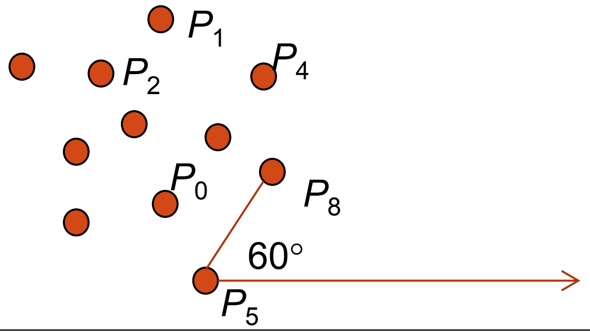


Gift Wrap: Implementation

Iteration (i = 0):

- Swap `pts`[i] with `pts`[k].
- For the remaining points with indices j = i+1…n, compute theta(`pts`[i], `pts`[j])
- k is the index of the point that gives the minimum angle, with the additional condition, angle > v (previous minimum)
- In the example below, k = 8. Minimum angle = 60
- $v \leftarrow 60$, $i \leftarrow 1$.

pts[0]	pts[1]	pts[2]	pts[3]		pts[5]						pts[n-1]	pts[n]
P_5	P_1	P_2	P_3	P_4	P_0						P_{n-1}	P_5

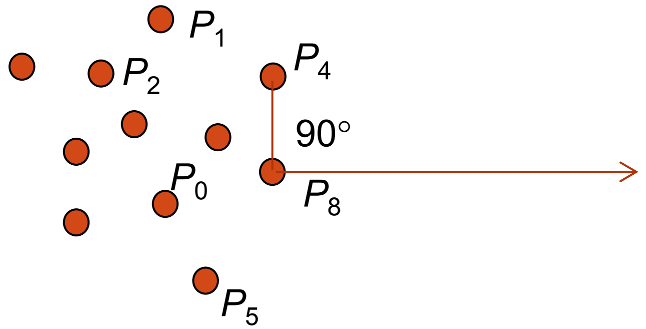


Gift Wrap: Implementation

Iteration (i = 1):

- Swap `pts`[i] with `pts`[k].
- For the remaining points with indices j = i+1…n, compute theta(`pts`[i], `pts`[j])
- k is the index of the point that gives the minimum angle, with the additional condition angle > v (previous minimum)
- In the example below, k = 4. Minimum angle = 90
- $v \leftarrow 90$, $i \leftarrow 2$.

pts[0]	pts[1]	pts[2]	pts[3]		pts[5]			pts[8]			pts[n-1]	pts[n]
P_5	P_8	P_2	P_3	P_4	P_0			P_1			P_{n-1}	P_5



Gift Wrap: Implementation

Iteration (i = 2):

pts[0]	pts[1]	pts[2]	pts[3]		pts[5]			pts[8]			pts[n-1]	pts[n]
P_5	P_8	P_4	P_3	P_2	P_0			P_1			P_{n-1}	P_5

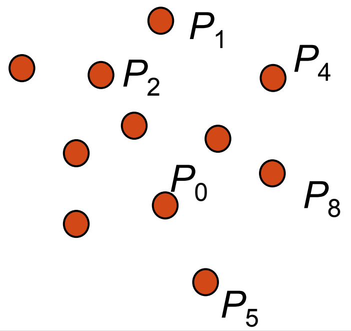
k = 8, i = 3

Iteration (i = 3):

pts[0]	pts[1]	pts[2]	pts[3]		pts[5]			pts[8]			pts[n-1]	pts[n]
P_5	P_8	P_4	P_1	P_2	P_0			P_3			P_{n-1}	P_5

… and so on.

The algorithm stops when the value of k equals n.



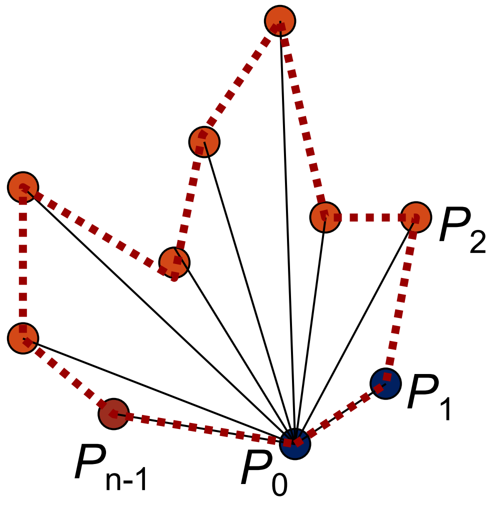
Graham-Scan Algorithm

The Graham-scan algorithm first constructs a "Simple Closed Path" (Slide 22) to obtain a sequential ordering L of the input points, where P_0 is the point with minimum y coordinate.

- With this ordering of points, P_0, P_1 will always be on the convex hull.
- The points P_0, P_1 , P_2 will always make a left turn. Note: P_2 *need not* necessarily lie on the convex hull.
- Remove the first three points from L and add them to a stack, with $P2$ as the top-of-stack element.

$L = [P_3, \ldots P_{n-1}]$

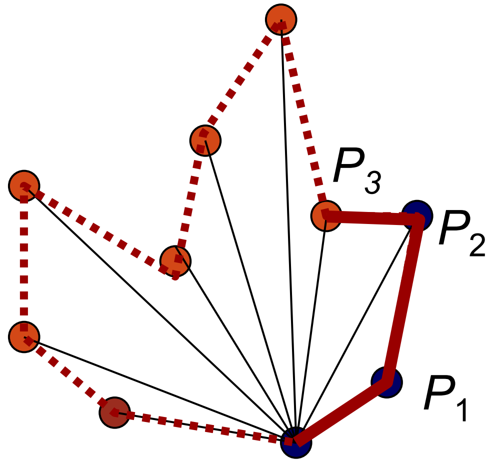
$S = [P_0, P_1, P_2]$ ← Top of stack



Graham-Scan Algorithm

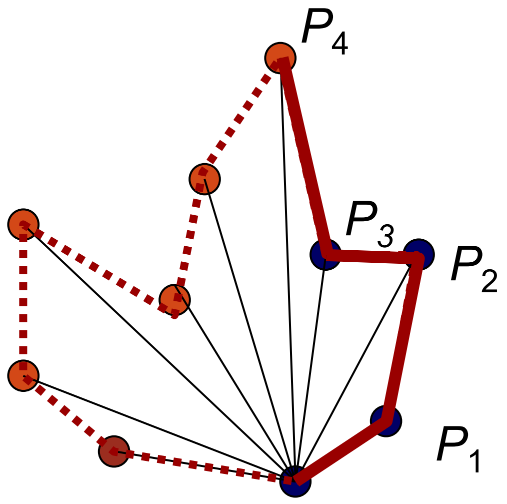
- In each iteration, the next point is removed from *L* and checked if it is on the left of line connecting the last two points in *S*. If it is, the point is added to *S*, otherwise, the last point (top-of-stack element) in *S* is removed.
- The algorithm terminates when the last element in *L* has been added to *S*. The stack *S* would then contain the vertices of the convex hull in counter-clockwise order.

Remove P_3 from L: $[P_4, \ldots P_9]$
IsCCW(P_1, P_2 , P_3)? YES
Add P_3 to S.
$S = [P_0, P_1, P_2, P_3]$

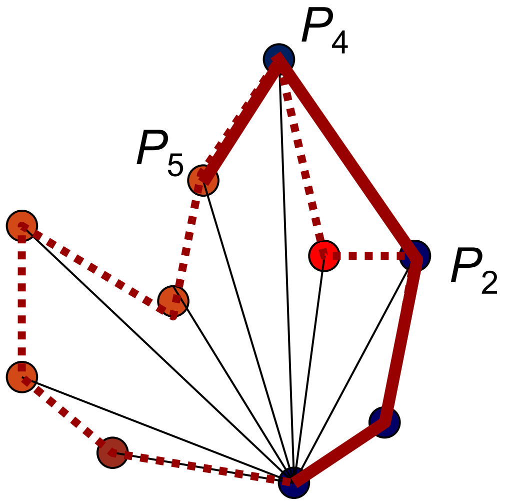


Graham-Scan Algorithm

Remove P_4 from L: [P_5, … P_9]
IsCCW(P_2, P_3 , P_4)? **NO**
Remove top-of-stack element.
S = [P_0, P_1, P_2]
IsCCW(P_1, P_2 , P_4)? YES
Add P_4 to S.
S = [P_0, P_1, P_2 , P_4]

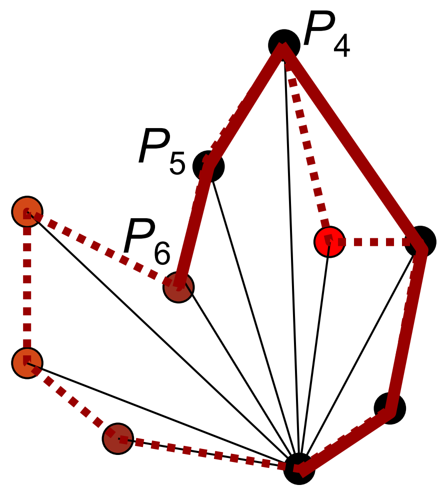


Remove P_5 from L: [P_6, … P_9]
IsCCW(P_2, P_4 , P_5)? YES
Add P_5 to S.
S = [P_0, P_1, P_2 , P_4 , P_5]

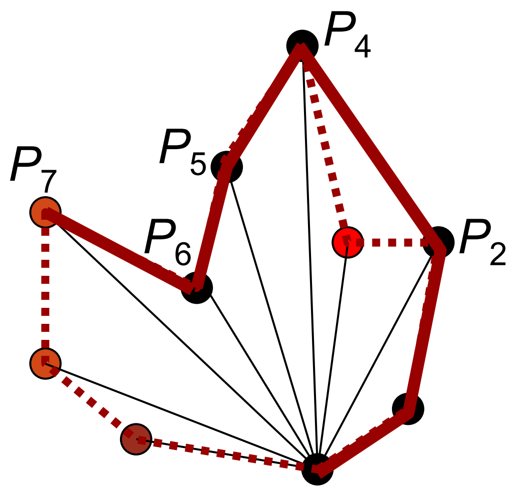


Graham-Scan Algorithm

Remove P_6 from L: $[P_7, P_8, P_9]$
IsCCW(P_4, P_5, P_6)? YES
Add P_6 to S.
$S = [P_0, P_1, P_2, P_4, P_5, P_6]$

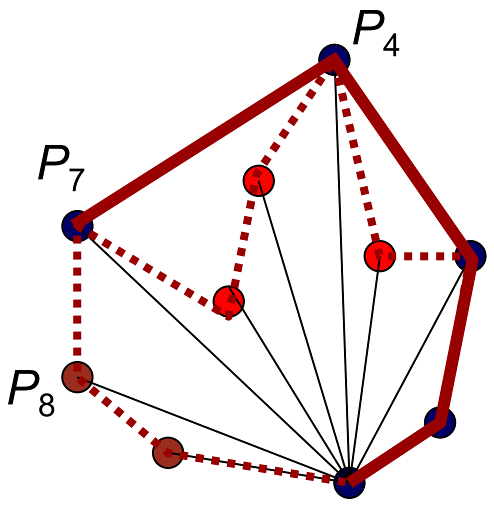


Remove P_7 from L: $[P_8, P_9]$
IsCCW(P_5, P_6, P_7)? **NO**
Remove top-of-stack element.
$S = [P_0, P_1, P_2, P_4, P_5]$
IsCCW(P_4, P_5, P_7)? **NO**
Remove top-of-stack element.
$S = [P_0, P_1, P_2, P_4]$
IsCCW(P_2, P_4, P_7)? YES
Add P_7 to S.
$S = [P_0, P_1, P_2, P_4, P_7]$

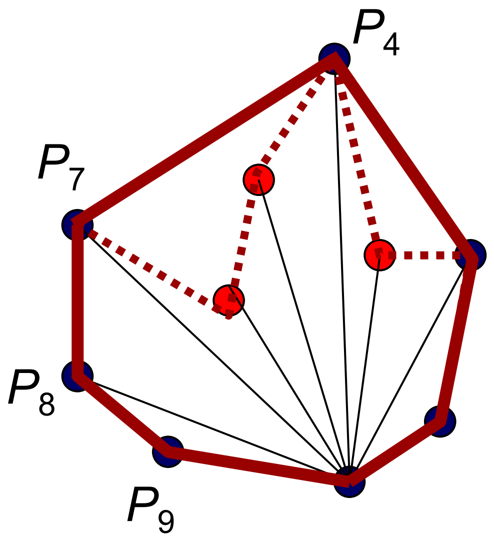


Graham-Scan Algorithm

Remove P_8 from L: $[P_9]$
IsCCW(P_4, P_7 , P_8)? YES
Add P_8 to S.
$S = [P_0, P_1, P_2 , P_4 , P_7 , P_8]$



Remove P_9 from L: []
IsCCW(P_7, P_8 , P_9)? YES
Add P_9 to S.
$S = [P_0, P_1, P_2 , P_4, P_7 , P_8 , P_9]$



Graham Scan: Pseudo Code

Requires fn `theta()` Slide 27

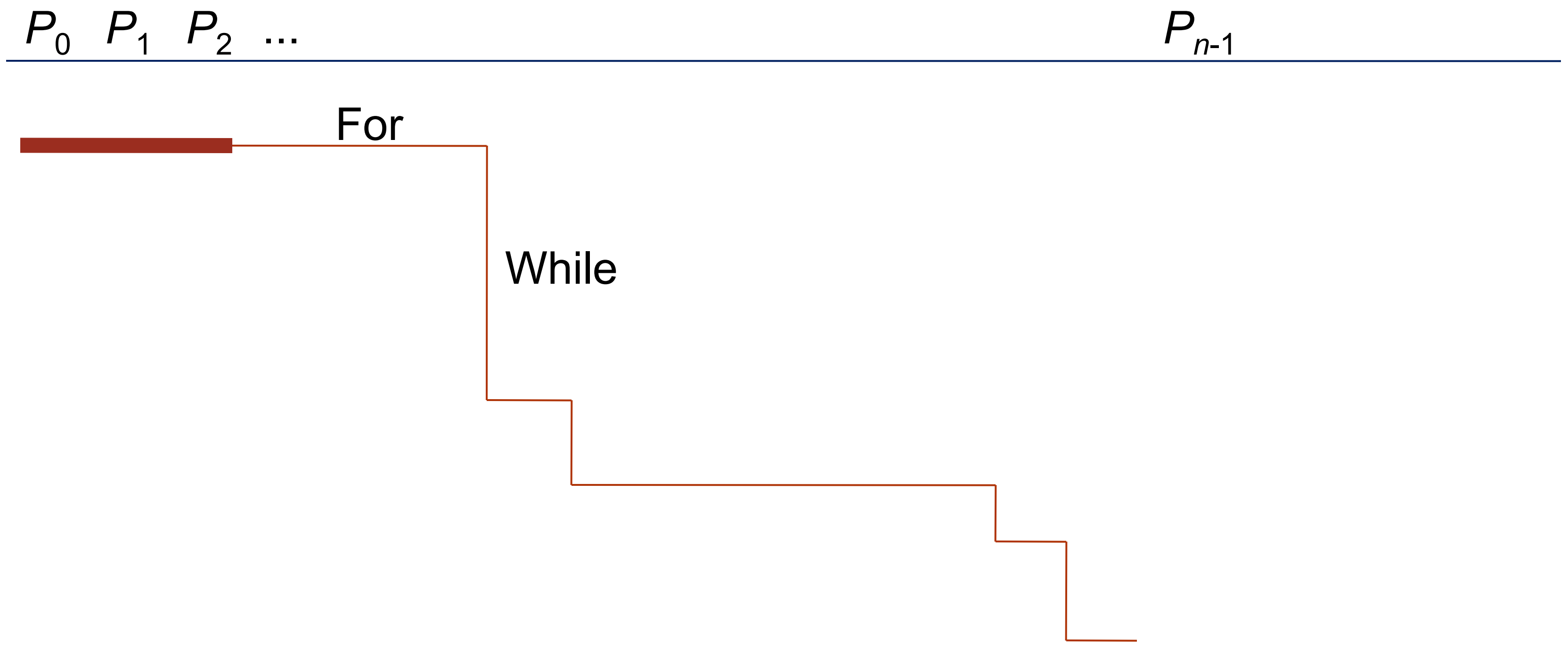
```
1. $P_0$ = Rightmost lowest point
2. Sort other points by angle about the horizontal line through $P_0$
   and form the list of points $L = [P_3, \ldots P_{n-1}]$
3. Stack $S = [P_0, P_1, P_2]$
4. For ($i = 3$ to $n-1$) {            #Process each point in $L$
5.     While not isCCW($S[-2]$, $S[-1]$, $P_i$)
6.         $S$.Pop();
7.     $S$.Append($P_i$)
8. }
```

See slide 8

Step 1 takes $O(n)$ time.
Step 2 takes $O(n\log n)$ time.
Step 3 takes $O(1)$ time.
What about loops 4, 5?

Graham Scan: Complexity

- The ‘For’ loop in step 4 adds one element to the stack.
- The nested ‘While’ loop may execute several times removing elements from the stack, but it can only remove elements that were previously added.
- The above type of nested loops lead to a ‘staircase’ pattern of execution as shown below:



Graham Scan: Complexity

- The staircase pattern on the previous slide can have a maximum height of n. The execution of the nested loops can be completed in at most $2n$ steps, *i.e.*, $O(n)$ time.
- Thus the running time of Graham Scan algorithm is $O(n\log n)$.