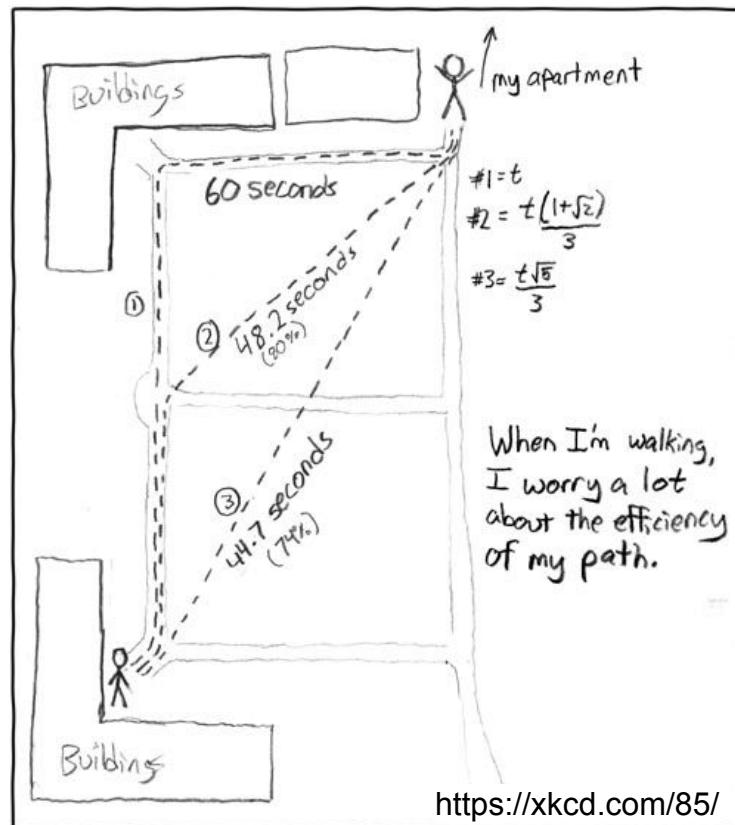


Computational Geometry

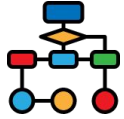
Part 1: Basic Geometry, Convex Hulls



Ben Adams (slides by Richard Lobb & R. Mukundan)

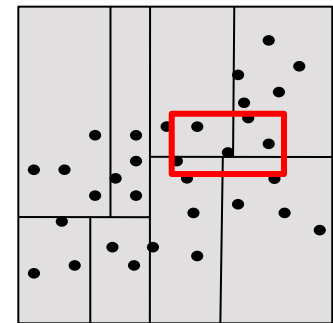
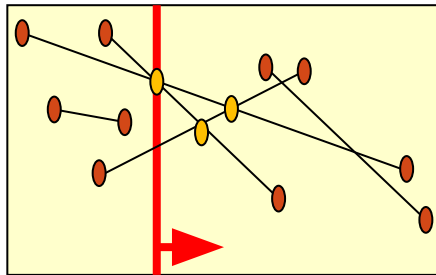
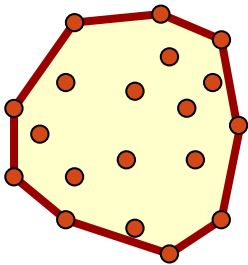
benjamin.adams@canterbury.ac.nz

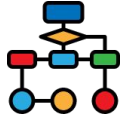
Department of Computer Science and Software Engineering
 University of Canterbury



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.
- The primary emphasis is on **computational complexity**.

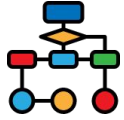




Introduction

Common characteristics of computational geometry problems:

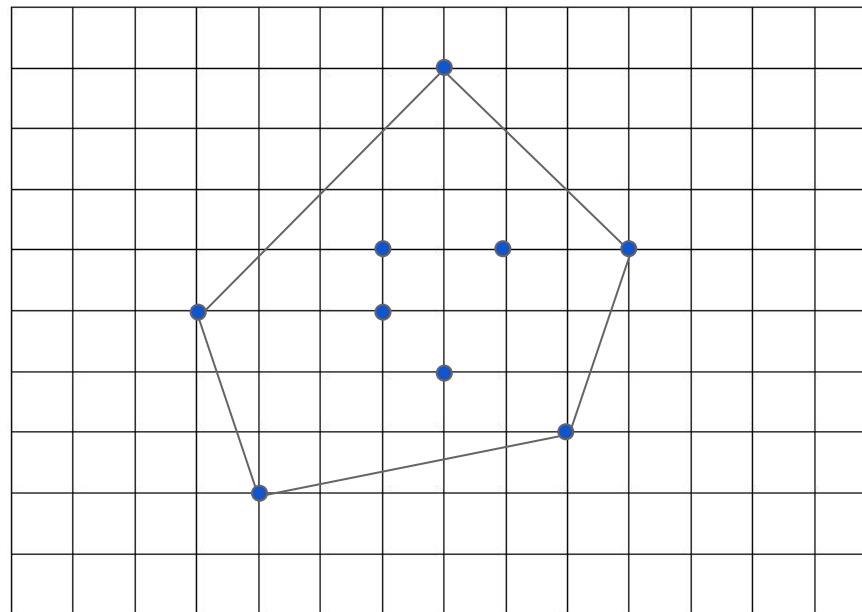
- **Floating point errors:** numerical instabilities arise from things like nearly-parallel lines, rectangles with massive aspect ratios, points perilously close together, etc.
- **Special cases:** Coincident points or lines, several lines intersecting at a point, vertical lines, parallel and overlapping lines.
- **Degenerate cases:** A line segment with equal end points, a triangle with zero area.

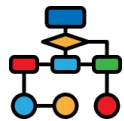


A Simplification

To avoid most of the trickiest issues in Computational Geometry we will (mostly) restrict ourselves to problems without degeneracies that can be solved without floating point calculations

- All points are on an integer cartesian grid.
- We try to avoid division, square roots, trig functions etc.





Plotting points, lines, polygons etc

matplotlib is useful for visualising geometric problems

Documentation:

<https://matplotlib.org/api>

Particularly https://matplotlib.org/api/axes_api.html

Example:

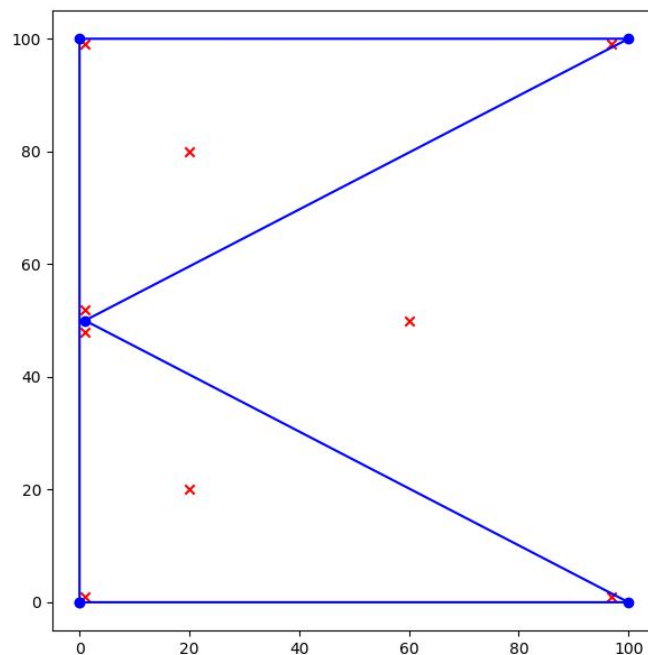
```
import matplotlib.pyplot as plt

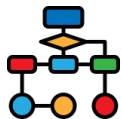
vertices = [(0, 0), (100, 0), (1, 50),
            (100, 100), (0, 100), (0,0)]
vx, vy = zip(*vertices) # Unpack them

points = [(1, 1), (20, 20), (20, 80), (60, 50),
          (97, 1), (1, 48), (1, 52), (97, 99), (1, 99)]
px, py = zip(*points) # Unpack

axes = plt.axes()
axes.plot(vx, vy, color='blue', marker='o')
axes.plot(px, py, color='red', marker='x', linestyle='')

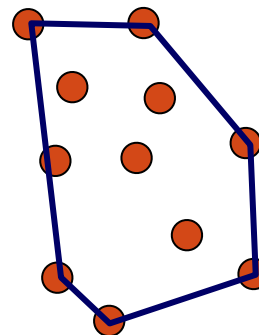
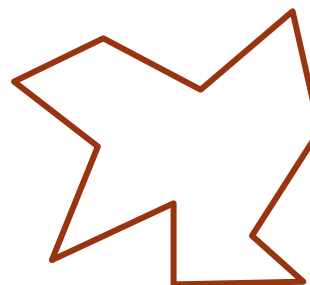
plt.show()
```



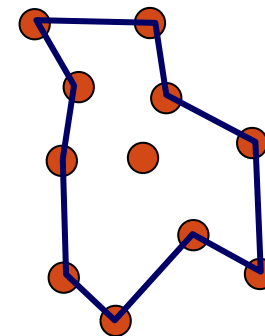


Lecture Outline

- Points, vectors and lines
 - Computer representation
 - Basic operations
- Polygons
 - Convex polygons
 - Point inclusion tests
 - Simple closed paths
- Convex Hulls
 - Properties and applications
 - Gift-Wrap algorithm
 - Graham-Scan algorithm

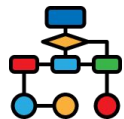


Convex



Non-Convex

Region Boundaries



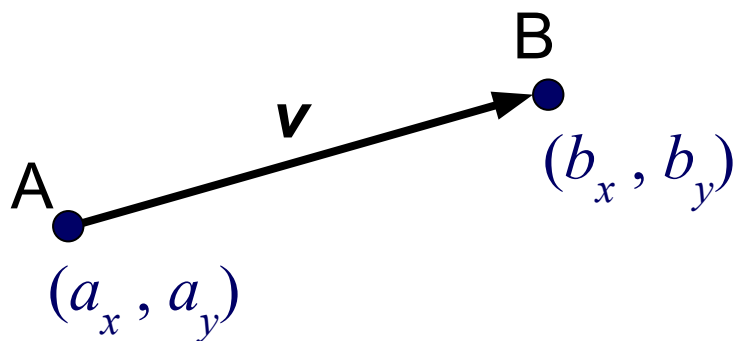
Points and Vectors

A *point* is a position in space, e.g. Christchurch, Dunedin.

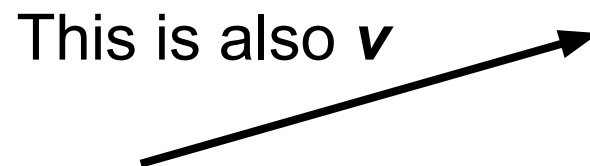
Represented in the computer by coordinates in a cartesian grid, e.g. a 2-element tuple (x, y) for 2D.

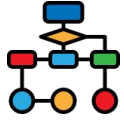
A *vector* is the difference between two points, essentially a $(\Delta x, \Delta y)$ tuple. *It has no position.*

Traditionally said to have *direction* and *magnitude* but we don't usually represent it that way in the computer.



$$\mathbf{v} = (b_x - a_x, b_y - a_y)$$

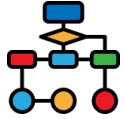




Naming convention

In these notes I (try to) use:

- Upper-case non-bold non-italic letters for points
 - A, B, P, Q etc
- Lower-case bold italic letters for vectors
 - \mathbf{v} , \mathbf{p} , \mathbf{q} etc
- Also I often use the lower case bold italic letter for the *position vector* of a point
 - e.g. \mathbf{p} is the *position vector* of P
 - i.e. the vector from the origin to P



Operations on points and vectors

Operations on points

Can subtract them to get a displacement vector $\mathbf{v} = B - A$

Linear/affine sum $P(\alpha) = (1 - \alpha)A + \alpha B$ represents points on line between A and B

You can't generally add or multiply points e.g., where's Christchurch + Dunedin ?

The linear sum equation looks like an exception, but isn't. It's a single indivisible 2-operand operation.

Operations on vectors

Addition and subtraction

Multiplication by a scalar

Addition of a vector to a point to get another point e.g. $B = A + \mathbf{v}$

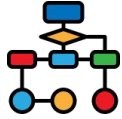
Length by Pythagoras: $|\mathbf{v}| = \sqrt{v_x^2 + v_y^2}$ but use $|\mathbf{v}|^2$ where possible

Dot product

Signed area

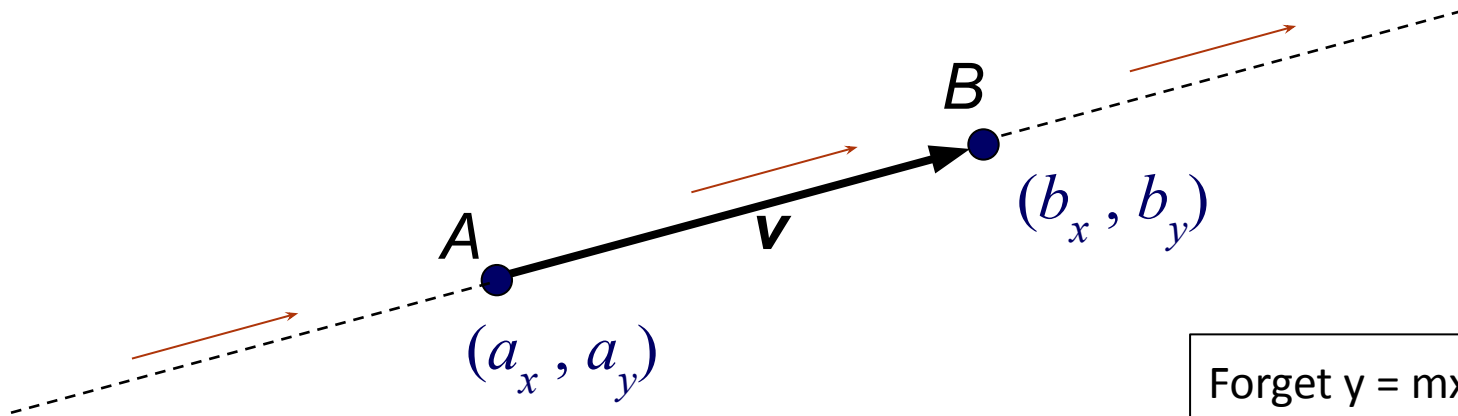
Assumed
knowledge

Warning: although points and vectors are both represented as (x, y) tuples, they *are* different. We cannot *translate* (= "shift") a vector nor *add* points.



Lines

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

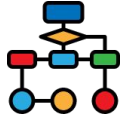


Forget $y = mx + c$.
It's no use to you here!

Can represent any point on the line as

$$P(\alpha) = A + \alpha \mathbf{v} = A + \alpha(B - A) = (1 - \alpha)A + \alpha B$$

α is relative distance from A to B , e.g. 0.5 at mid-point.



Vector dot product ("scalar product")

$$\mathbf{p} = (p_x, p_y), \mathbf{q} = (q_x, q_y): \quad \mathbf{p} \cdot \mathbf{q} = p_x q_x + p_y q_y$$

$$\text{e.g. } (3,6) \cdot (2,4) = (3 \times 2 + 6 \times 4) = 30$$

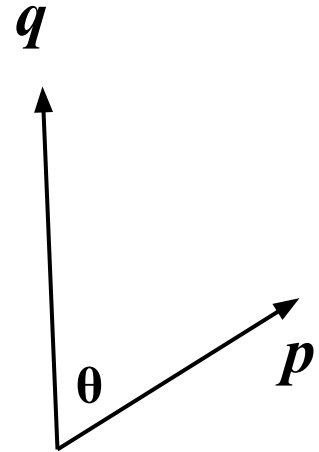
Uses:

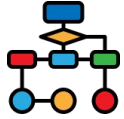
Magnitude: $|\mathbf{p}|^2 = \mathbf{p} \cdot \mathbf{p}$

- Avoid square roots if possible
- They're slow and approximate

Angles

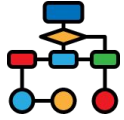
- $\mathbf{p} \cdot \mathbf{q} = |\mathbf{p}| |\mathbf{q}| \cos \theta$
- $\mathbf{p} \cdot \mathbf{q} > 0 \Rightarrow$ angle between directions \mathbf{p} and \mathbf{q} is acute.
- $\mathbf{p} \cdot \mathbf{q} == 0 \Rightarrow \mathbf{p}$ perpendicular to \mathbf{q}





Computer representation

- We don't want to do separate calculations with x and y .
- Ideally have a class *Point* and a class *Vector*.
 - Restrict operations:
 - point - point \rightarrow vector
 - point + point illegal
 - point + vector \rightarrow point
 - vector \pm vector \rightarrow vector
 - etc
- But coding and using such classes is cumbersome
- In this course we'll use a single class *Vec* for vectors
- Represent points by their ***position vectors***, i.e. the vector from the origin to the point
- It's over to you to avoid meaningless operations like *point + point*, translating a vector, rotating a point (about what?) etc



Class *Vec*

Try to minimise use of *x* and *y* attributes - do vector operations where possible.

```
class Vec:
    """A simple vector in 2D. Also used as a position vector for points"""
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vec(self.x + other.x, self.y + other.y)

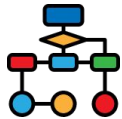
    def __sub__(self, other):
        return Vec(self.x - other.x, self.y - other.y)

    def __mul__(self, scale):
        return Vec(self.x * scale, self.y * scale)

    def dot(self, other):
        return self.x * other.x + self.y * other.y

    def lensq(self):
        return self.dot(self)

    def __repr__(self):
        return "({}, {})".format(self.x, self.y)
```



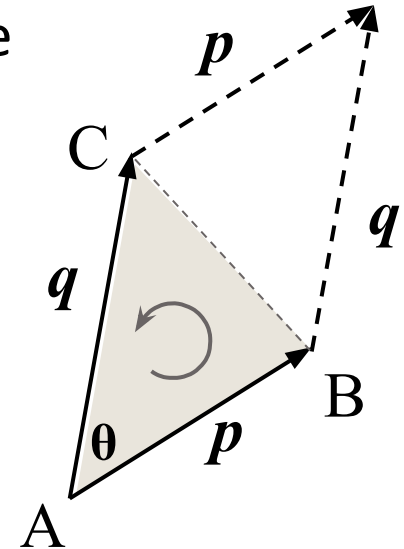
Signed area/turn direction

The area of the parallelogram defined by two edge vectors \mathbf{p} and \mathbf{q} from some point A is

$$\begin{aligned}\text{Area}_{\text{parallelogram}} &= p_x q_y - p_y q_x \\ &= |\mathbf{p}| |\mathbf{q}| \sin \theta\end{aligned}$$

$$\mathbf{p} = \mathbf{B} - \mathbf{A}$$

$$\mathbf{q} = \mathbf{C} - \mathbf{A}$$



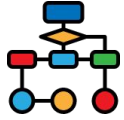
The triangle area is half that.

Area is positive if ABC is CCW (counter- clockwise)

Area is zero if ABC are collinear (\mathbf{p} is parallel to \mathbf{q})

Area is negative otherwise.

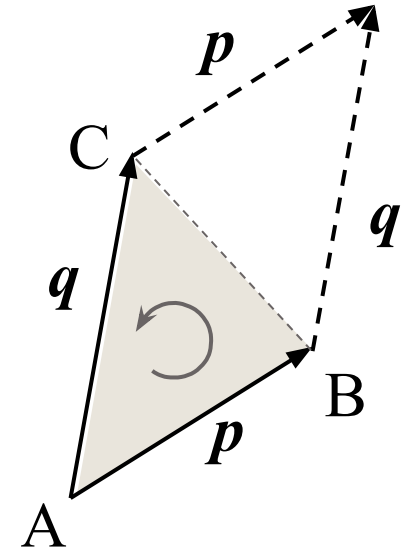
Take absolute value if the **area** is what you want (rather than the turn direction).

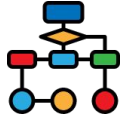


The *signed_area* function

To avoid division (we want to use only integers) we define a function that yields *twice* the area of triangle ABC.

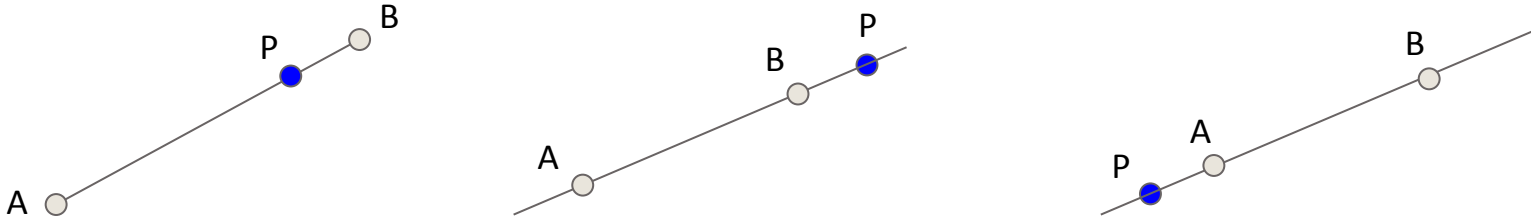
```
def signed_area(a, b, c):
    """Twice the area of the triangle abc.
       Positive if abc are in counter clockwise order.
       Zero if a, b, c are collinear.
       Otherwise negative.
    """
    p = b - a
    q = c - a
    return p.x * q.y - q.x * p.y
```





Point P on line or line segment (A, B)

If P, A and B are collinear, $\text{signed_area}(P, A, B) == 0$.



Additionally, if P is on the line segment (A, B):

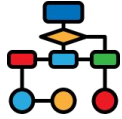
$$|(P, A)| \leq |(A, B)| \text{ and } |(P, B)| \leq |(A, B)|$$



The length of the line segment (P, A)

For efficiency and to avoid floating point, compare *squared* lengths

- Method `Vec.lensq()` e.g. $|(P, A)|^2 = (P - A).\text{lensq}()$



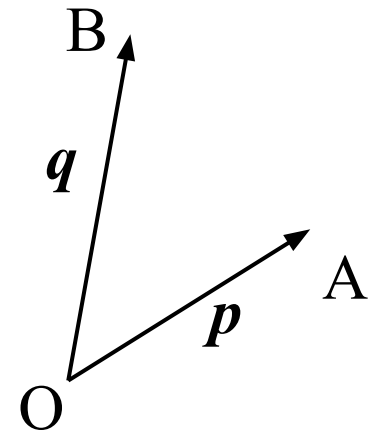
Turn direction

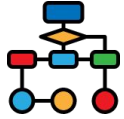
From signed area, we can answer questions like "If I stand at point O, looking towards A, is B on my left or my right?"

Call this "turn direction".

Just compute sign of area OAB.

- Positive \Rightarrow B to left
- Negative \Rightarrow B to right
- Zero \Rightarrow collinear

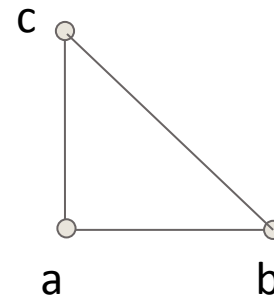


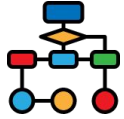


Code

```
def is_ccw(a, b, c):  
    """True iff triangle abc is counter-clockwise"""  
    area = signed_area(a, b, c) # As earlier  
    # May want to throw an exception if area == 0  
    return area > 0
```

```
>>> a = Vec(0, 0)  
>>> b = Vec(1, 0)  
>>> c = Vec(0, 1)  
>>> is_ccw(a, b, c)  
True  
>>> is_ccw(a, c, b)  
False
```





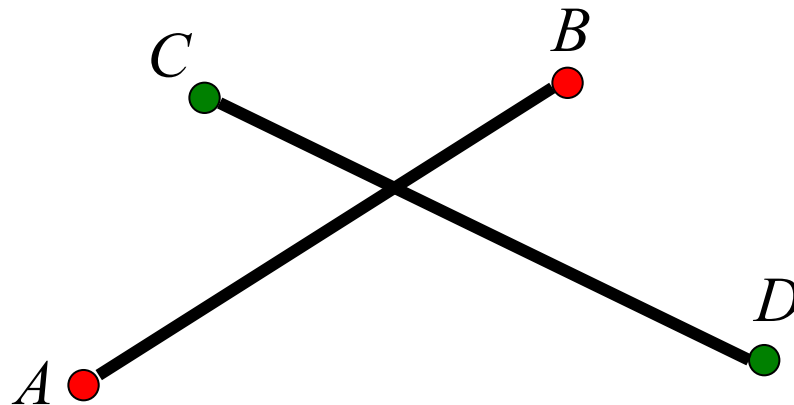
Line Segment Intersection Test

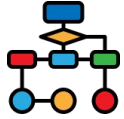
Two line segments AB and CD as shown below intersect only if points C, D are on different sides of line AB

i.e. if $\text{is_ccw}(\text{ADB}) \neq \text{is_ccw}(\text{ACB})$

and points A, B are on different sides of line CD

i.e. if $\text{is_ccw}(\text{CAD}) \neq \text{is_ccw}(\text{CBD})$



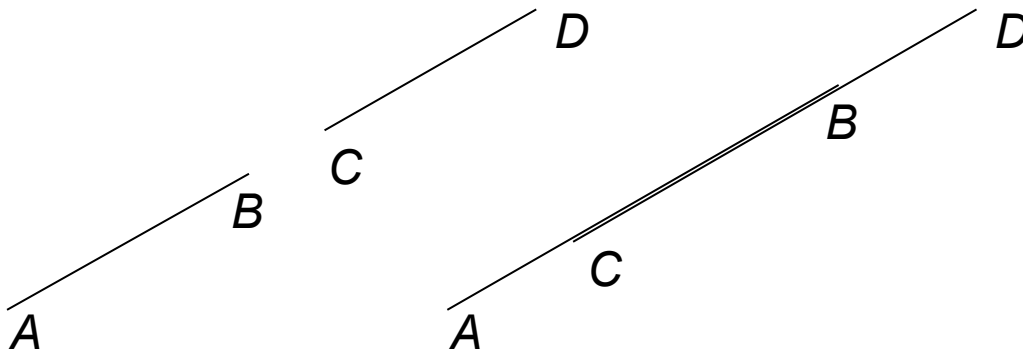


Line Segment Intersection Test

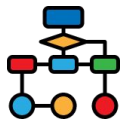
Special Cases: An end point of one line segment lies on the other segment



Degenerate Case: All four points are collinear

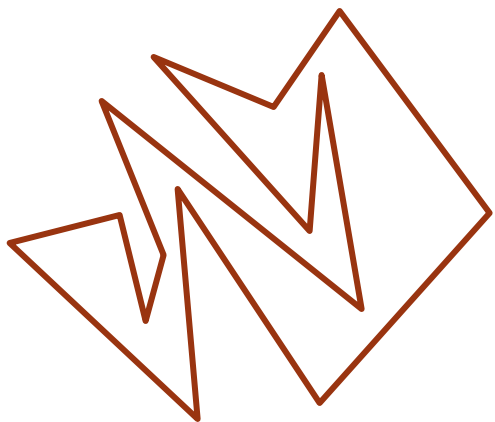


Q: what would happen in the code in these situations?

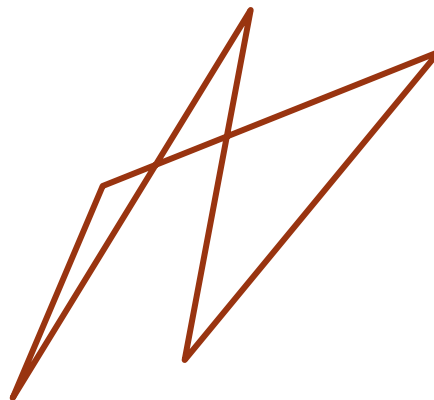


Simple Polygons

A simple polygon is a polygon whose edges do not intersect



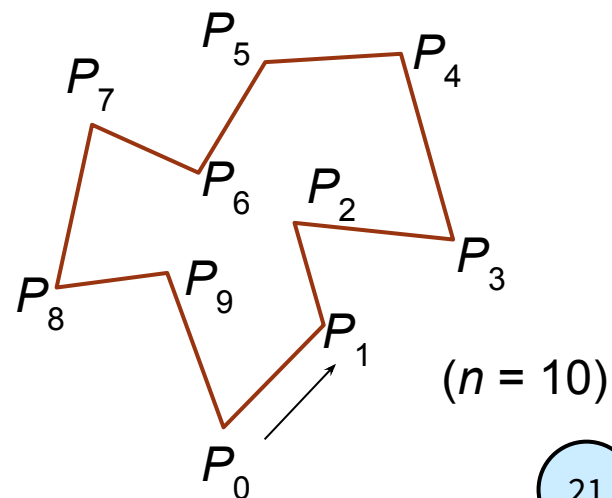
Simple polygon

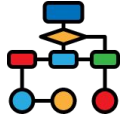


Self-intersecting polygon

A simple polygon can be represented using an ordered list of vertices: $[P_0, P_1, \dots, P_{n-1}]$

Vertices are usually in **counter-clockwise** (CCW) order.

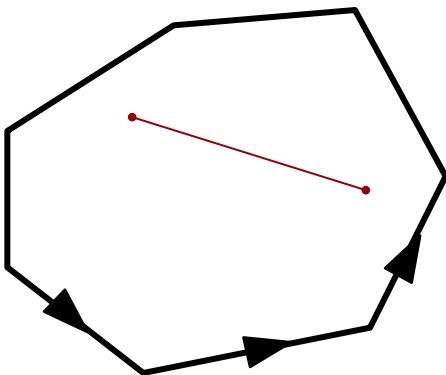




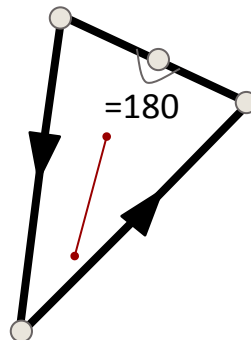
Convex Polygons

A convex polygon satisfies the following properties:

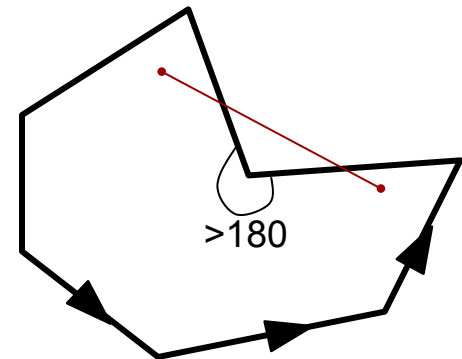
- Any line segment connecting two interior points lies entirely within the polygon
- Every **interior angle** is less than or equal to 180 degs.
 - Or *strictly less than* 180 degs for a *strictly convex* polygon
- Every **counterclockwise traversal** of a convex polygon either continues straight, or **turns left** at every vertex.
 - Use *is_ccw* (allowing area = 0 as well)



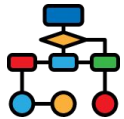
Convex



Convex



Not Convex

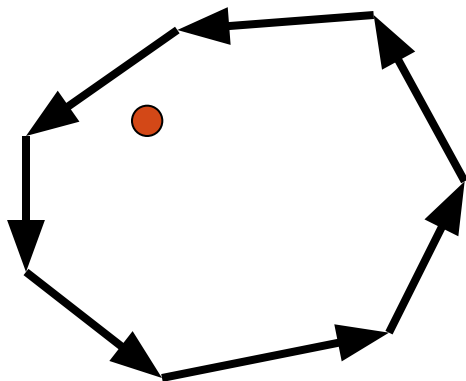


Point Inclusion Test for convex polygon

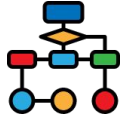
“Point Inclusion Test” refers to the problem of determining whether a given point is inside a polygon.

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.

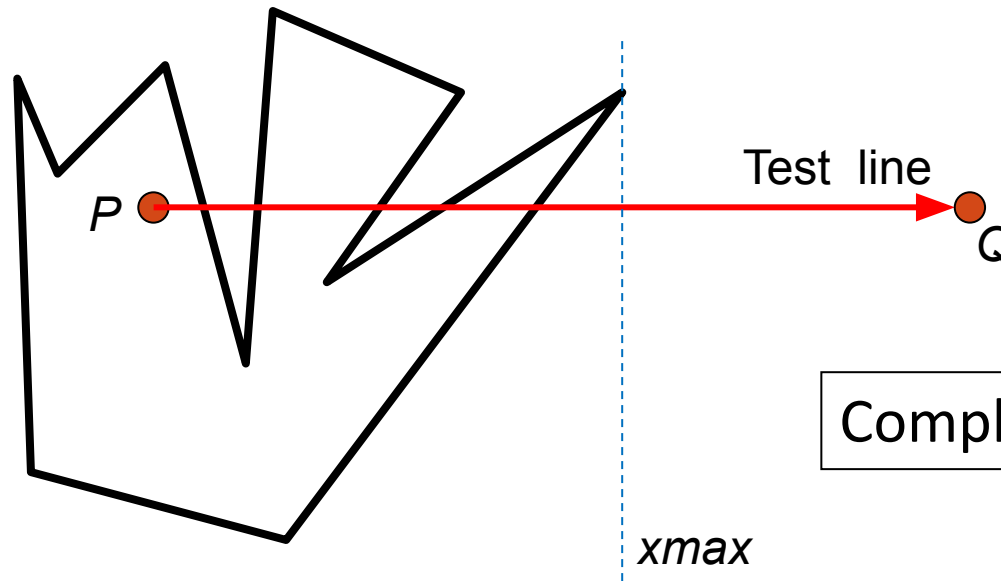


Use *is_ccw* (with $\text{area} > 0$)
Complexity = $O(n)$



Point Inclusion Test for simple polygon

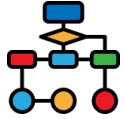
A point P lies inside a simple polygon if and only if a semi-infinite horizontal line emanating from the point **intersects the edges an odd number of times**.



Number of edge
intersections:
 $k = 5$

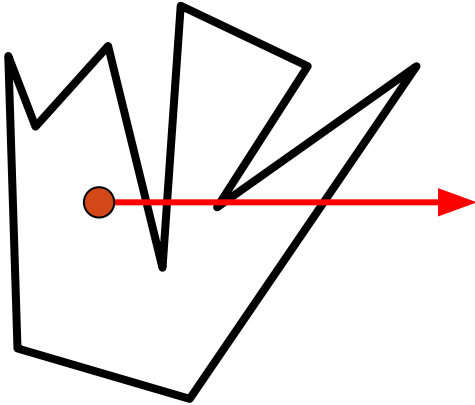
Complexity = $O(n)$

A horizontal “semi-infinite” line from P is actually a finite length segment PQ where the coordinates of Q are chosen such that $q_y = p_y$, $q_x > x_{max}$



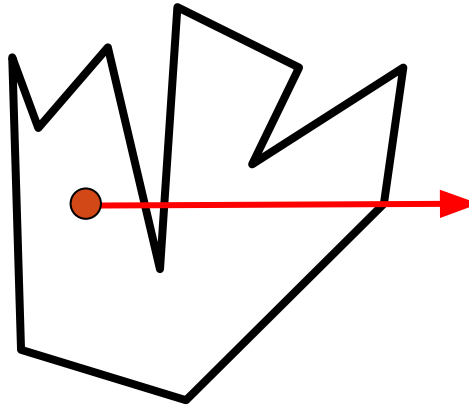
Point Inclusion Test

Special Cases:



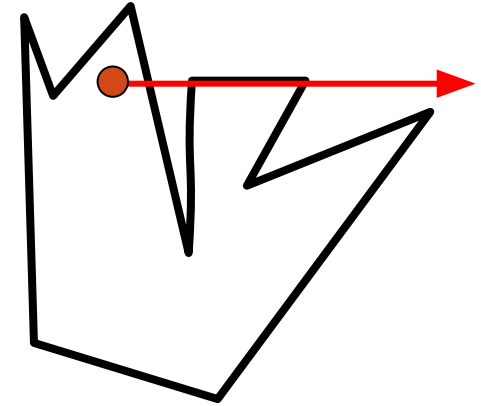
The line passes through a vertex (case a).

The intersection at the vertex should be counted zero times or twice.

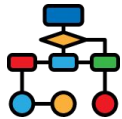


The line passes through a vertex (case b).

The intersection at the vertex should be counted only once



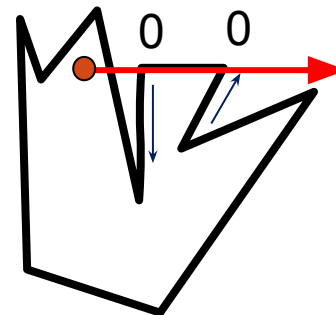
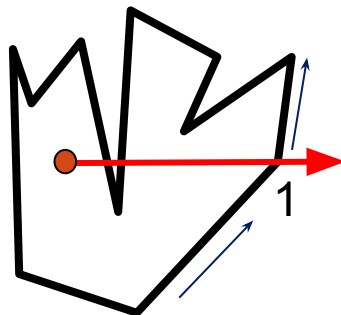
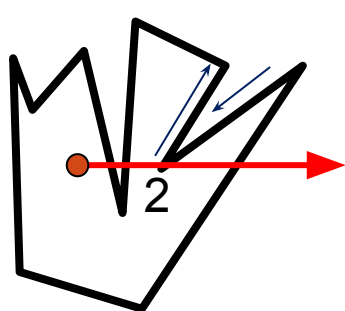
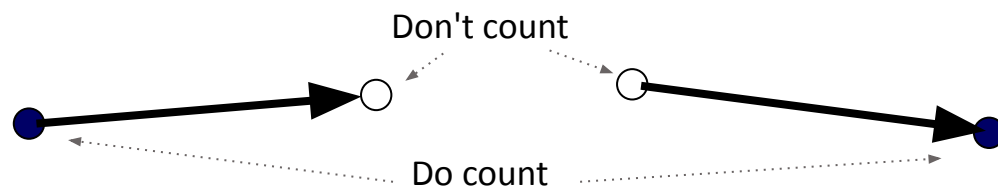
The line and an edge of the polygon overlap.



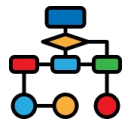
Point Inclusion Test

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

1. Ignore horizontal edges
2. If an edge is directed upward then it contains the start vertex.
3. If the edge is directed downward, it contains the end vertex.



Note: The edge directions given above assume a counterclockwise traversal of the polygon.



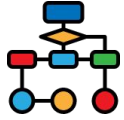
An easier solution

If all points are on an integer grid, just move P and Q very slightly upwards (e.g. by $1.E-08$).

Line PQ cannot now intersect any polygon vertices.


Mathematically not exactly correct but good enough in just about every practical context.

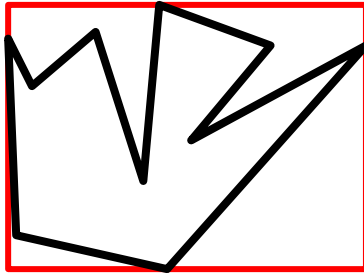
I said “no floating point arithmetic”. I lied.
But `Vec` and `is_ccw` still work fine with the
tweaked coordinates.



Further Improvements

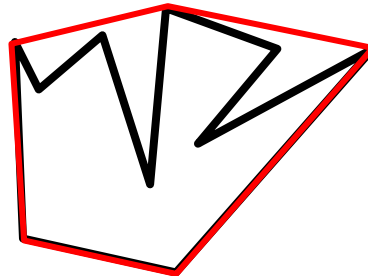
- We should try to minimise redundant computations
- A simple pre-processing step using a *bounding volume* can help in determining if a point is completely outside a rectangular region enclosing the polygon.

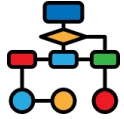
P

 (p_x, p_y)



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

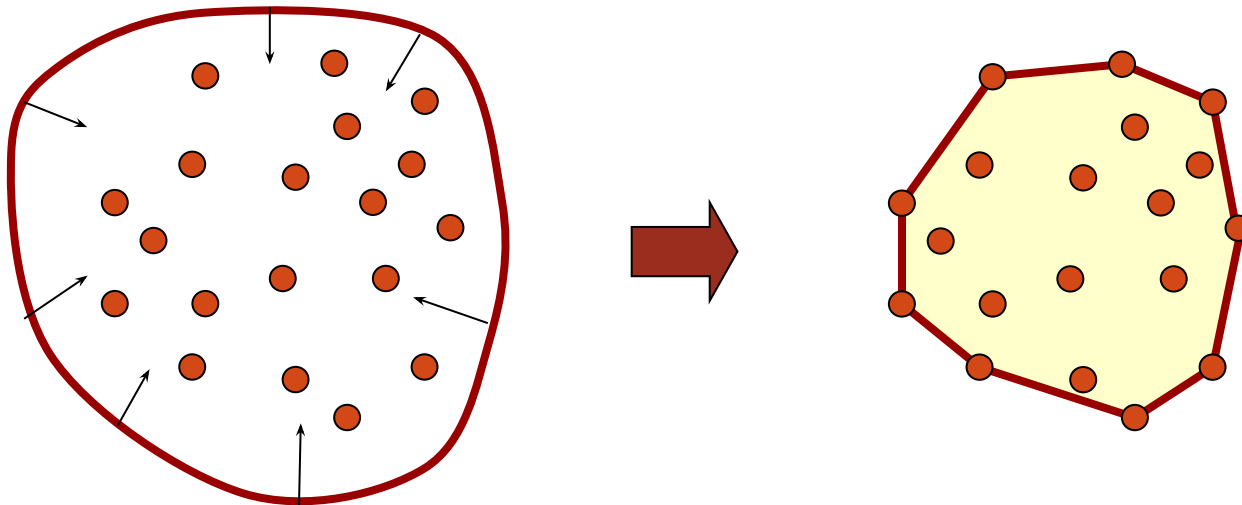
- A better (?) bounding volume:

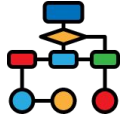




Convex Hulls

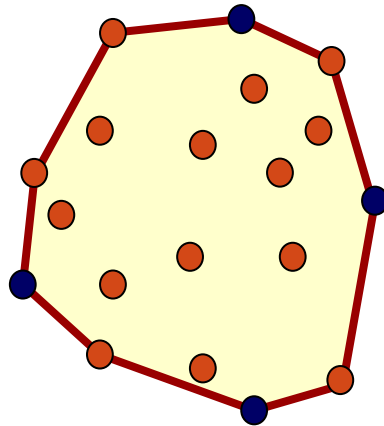
- Given a finite set of points $S = \{P_0, P_1 \dots P_{n-1}\}$, the convex hull of S is the smallest convex polygon enclosing all of the points.
- Visualized as the shape of a stretched rubber band around the points so that every point lies within the band.

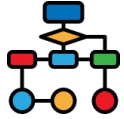




Properties of Convex Hulls

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





Convex combinations

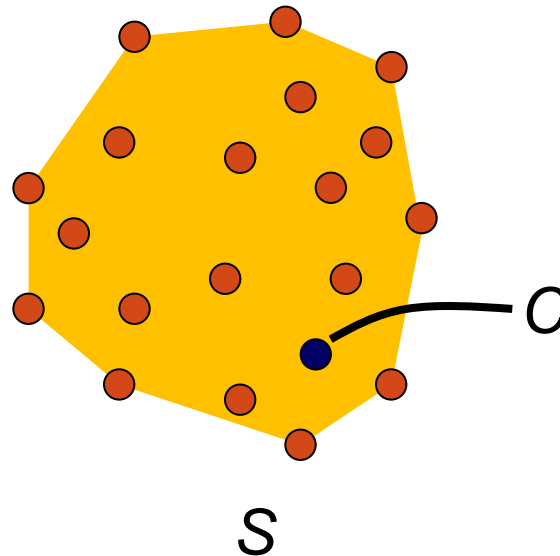
If we construct a new point C using a *convex combination* of points in S , then C lies inside the convex hull of S .

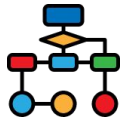
$$C = w_0 P_0 + w_1 P_1 + w_2 P_2 + \dots + w_{n-1} P_{n-1}$$

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



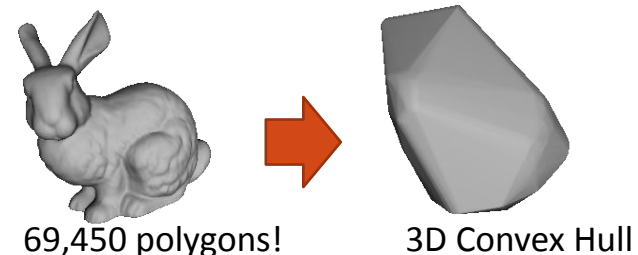
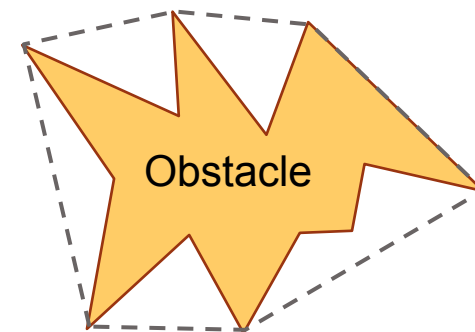
i.e. w_i is a real
number in the range
0 to 1 inclusive

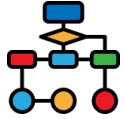




Applications of Convex Hulls

- GIS (geographical information systems)
 - Define regions from point samples, e.g. crime types, disease outbreaks)
- Robotics
 - Path planning
 - Object recognition
- Graphics
 - Bounding volumes for ray tracing and collision detection
- Many others
 - Image processing, pattern recognition ...





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:

`hull = []`

for each point $P \in S$:

for each point $Q \in S, Q \neq P$:

`edge_in_hull = True`

for each point $R \in S, R \neq P, R \neq Q$:

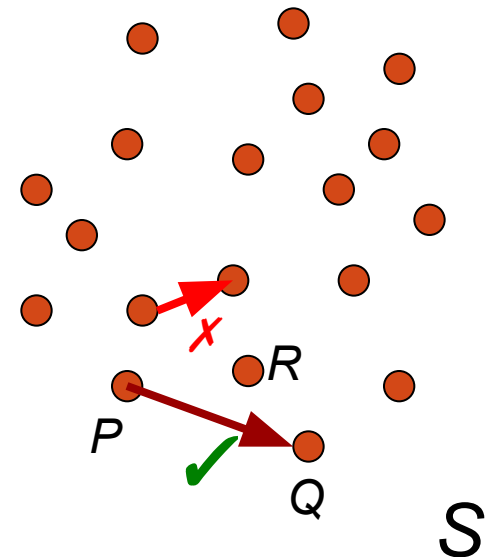
if PQR is *not* CCW

`edge_in_hull = False`

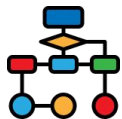
break

if `edge_in_hull`:

`hull.append(edge(P, Q))`



Complexity: $O(n^3)$

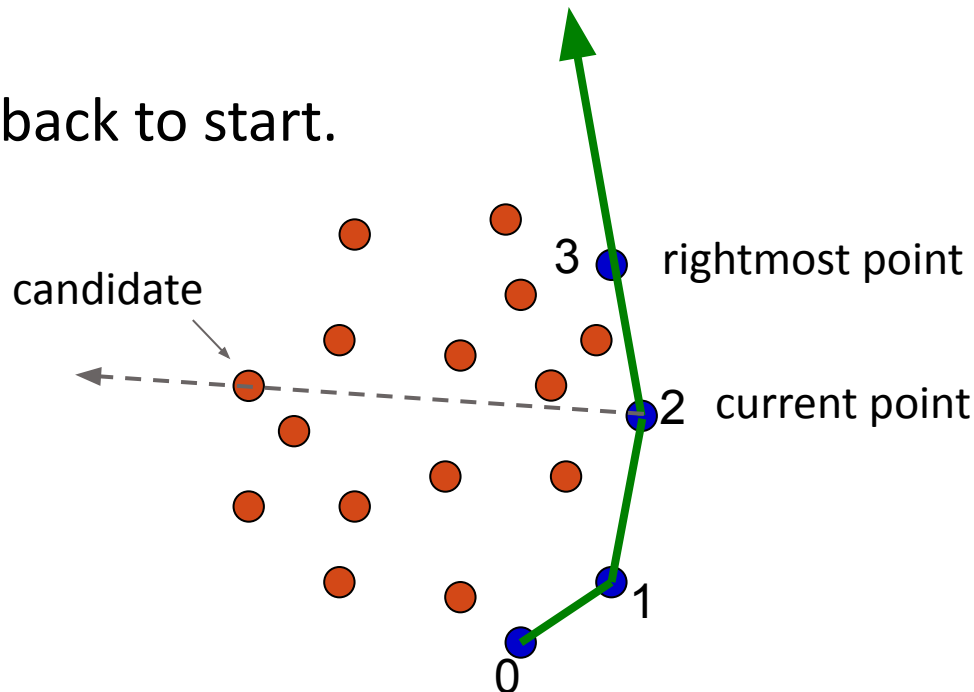


Convex Hull – Gift Wrap

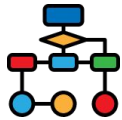
Also known as Package Wrap, or Jarvis March algorithm.

Starting from the point with minimum y -coordinate, select the next point as the “rightmost one” looking from the current point. [Can use *is_ccw* for that.]

Repeat until hull closes back to start.



See on-line visualisation: <https://lobb.nz/convexhullvisualiser/convexhull.html>



Gift Wrap (Pseudo-Code)

bottommost = point in S with minimum
y-coordinate (in case of ties, choose the
left-most point)

hull = [bottommost]

while hull is not a closed loop:

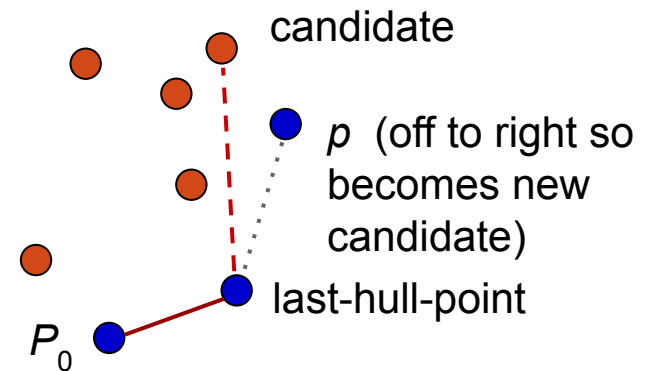
candidate = null

for p **in** {points - last-hull-point}:

if candidate is null **or** p is to the right of
candidate, seen from last-hull-point:

candidate = p

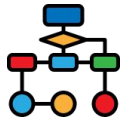
hull.append(candidate)



Output sensitive
complexity!

Complexity: $O(mn)$
Worst case: $O(n^2)$

m = Number of vertices on convex hull

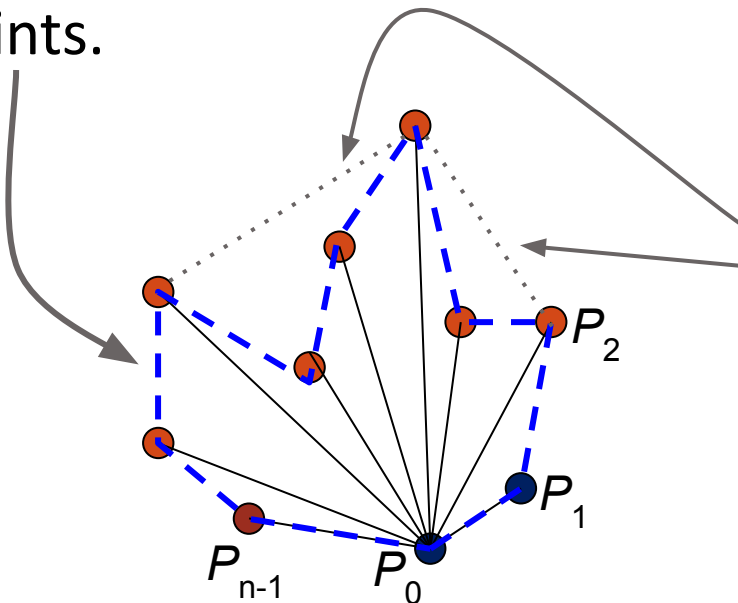


Graham-Scan Algorithm: idea

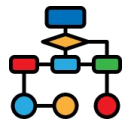
Gift wrap is $O(n^2)$ worst case. But we can do better.

The Graham-scan algorithm is a $O(n \log n)$ algorithm that uses *sorting*.

It first constructs a “Simple Closed Path” that includes all the points.

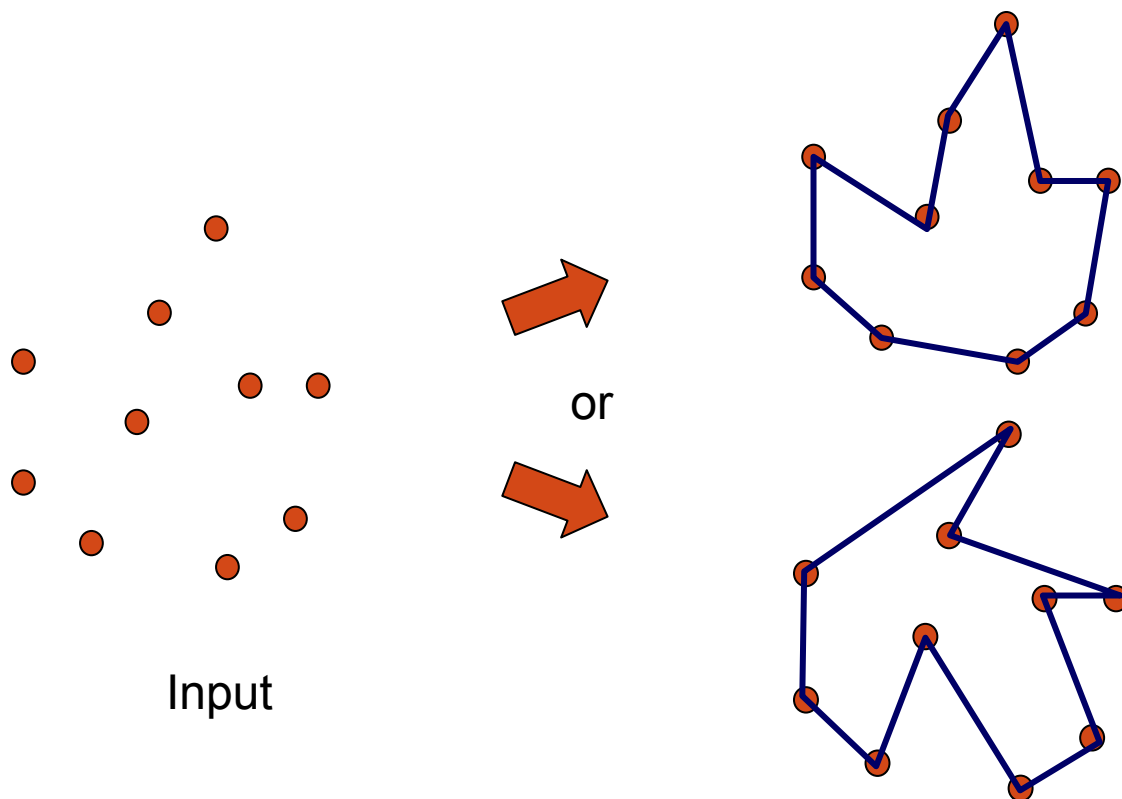


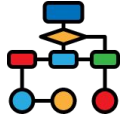
It then traverses the boundary, filling in the concavities



Step 1: Making a simple closed path

Problem: given a set of n points on a plane, compute a simple closed path that passes through all the points and does not intersect itself, i.e. a *simple polygon*

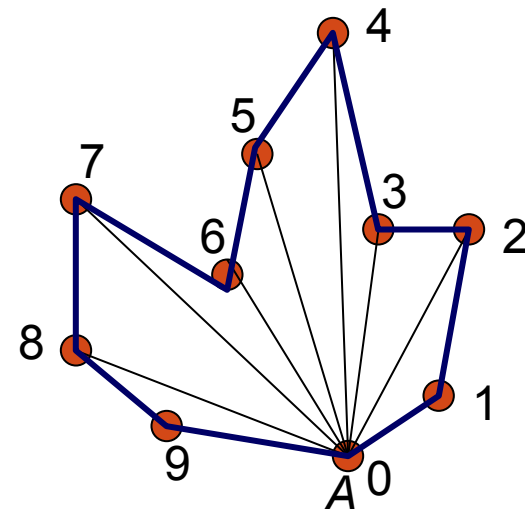
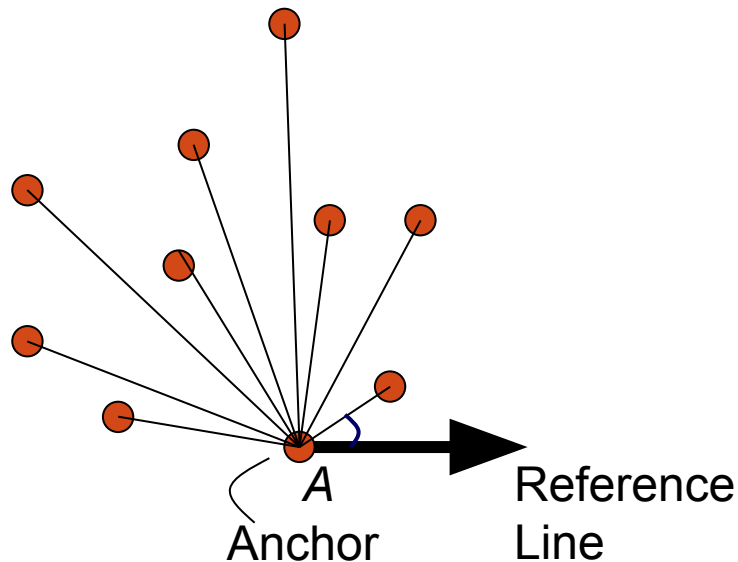


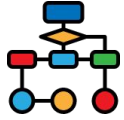


Simple Closed Path algorithm

Algorithm (in principle):

- 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

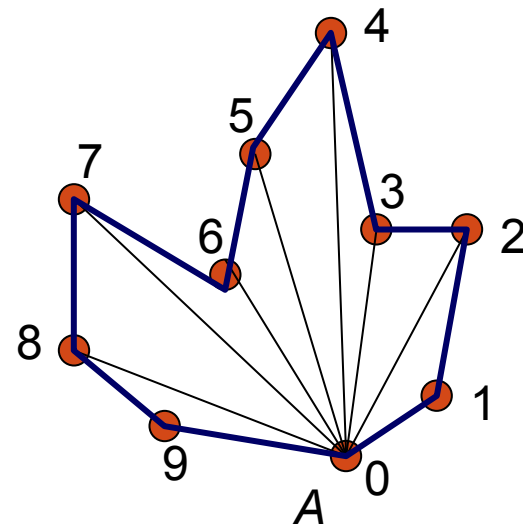
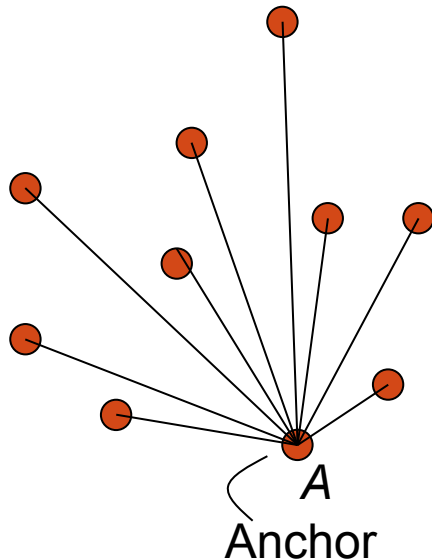


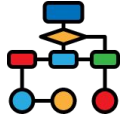


Sorting without calculating angles

Angle calculation requires trig. We're trying to use only integers. So:

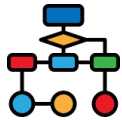
- We don't actually need angles - we only need to know the *ordering*, i.e. that P_i is to **the left** of the line (P_0, P_{i-1}) .
 - Think of this as the '<' operator
- Our *is_ccw* function can do this using integer arithmetic only. 😊





Sorting without trig: details

- Assume we have a list of points $\{p_i\}$.
- First we must identify p with minimum y (and, if multiple such points, minimum x).
 - Call it *anchor*.
- We need to sort all points in a counter-clockwise order as seen from point *anchor*.
- Python *sort* method and *sorted* function both have an optional *key* parameter to define ordering.
- We have an *is_ccw* function which we want to use (somehow) as a sort key.
 - E.g. *is_ccw(anchor, p1, p2)* means $p1 < p2$ for sort purposes
 - But it takes three parameters, whereas *sort key* function takes only one.



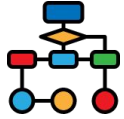
Solution 1: use *cmp_to_key*

```
from functools import cmp_to_key  # Converts a cmp function to a key function

def cmp(p1, p2):
    """Compares two points with respect to a globally defined anchor point.
    Returns a negative, zero or positive value according to whether p1 is
    to the right of p2 ("p1 < p2"), collinear with it ("p1 == p2") or to
    the left ("p1 > p2").
    """
    v1 = p1 - anchor
    v2 = p2 - anchor
    if v1.lensq() == 0:  # Is p1 the anchor point (or a copy of it)?
        return -1      # Yes. Make sure anchor point < everything else
    elif v2.lensq() == 0: # Is p2 the anchor point (or a copy of it)?
        return +1      # Yes, Make sure all other points > anchor
    else: # In all other cases, return the negative of the usual area
        return v2.x * v1.y - v1.x * v2.y  # This is negative if p1 < p2
```

Once *anchor* has been identified and defined globally (or in any outer context of *cmp*) can then sort points:

```
simply_poly = sorted(points, key=cmp_to_key(cmp))
```



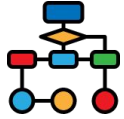
Solution 2: define a special SortKey class

```
class PointSortKey:
    """A class for use as a key when sorting points wrt anchor point"""
    def __init__(self, p, anchor):
        self.vec = p - anchor          # Direction vector from anchor to p
        self.is_anchor = self.vec.lensq() == 0 # True iff p == anchor

    def __lt__(self, other):
        """Compares two sort keys. p1 < p2 means the vector from the
           anchor point to p2 is to the left of the vector from the
           anchor to p1.
        """
        if self.is_anchor:
            return True                # Ensure anchor point < all other points
        elif other.is_anchor:
            return False                # Ensure no other point < the anchor
        else:
            area = self.vec.x * other.vec.y - other.vec.x * self.vec.y
            return area > 0             # area > 0 => is_ccw => p1 < p2
```

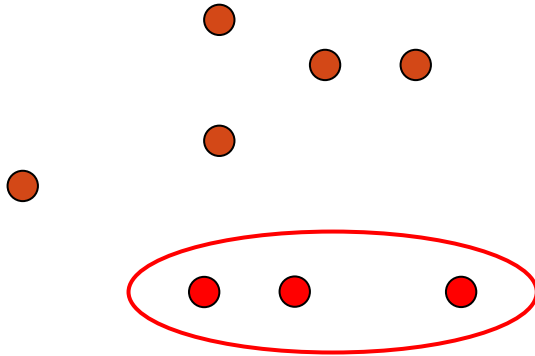
Once *anchor* has been identified can then sort points:

```
simply_poly = sorted(points, key=lambda p: PointSortKey(p, anchor))
```



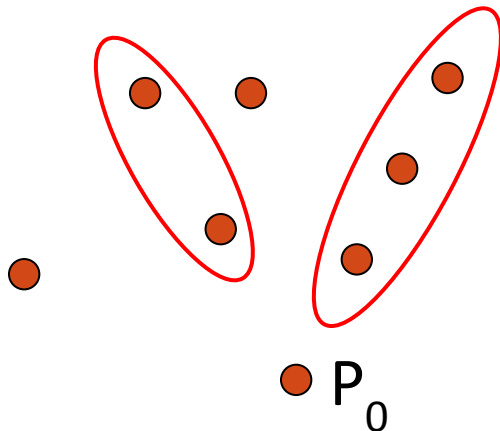
Simple Closed Path: Special Cases

- Several points with the same y -minimum value:

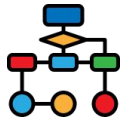


We choose the leftmost bottommost point as anchor in this case.

- Several points collinear with P_0 .



We solve this problem by ignoring it 😊



Graham-Scan Algorithm: step 2

Given our simple polygon $L = P_0, P_1 \dots P_{n-1}$ how do we fill in the concavities?

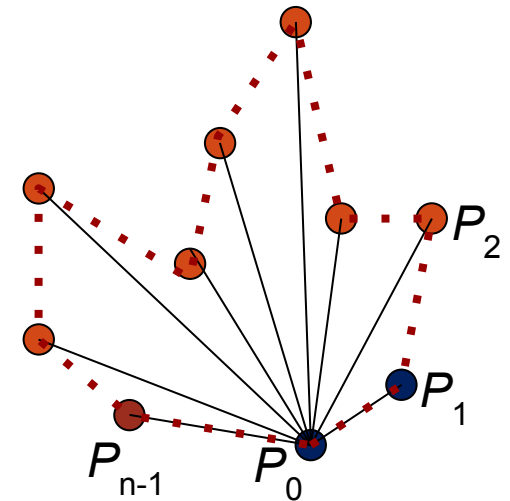
Observe that:

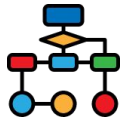
- P_0, P_1 will always be on the convex hull.
- P_0, P_1, P_2 will always make a left turn.
 - But P_2 may not be on the convex hull.

Idea:

Initialise hull H to $[P_0, P_1, P_2]$. It's a *stack*.

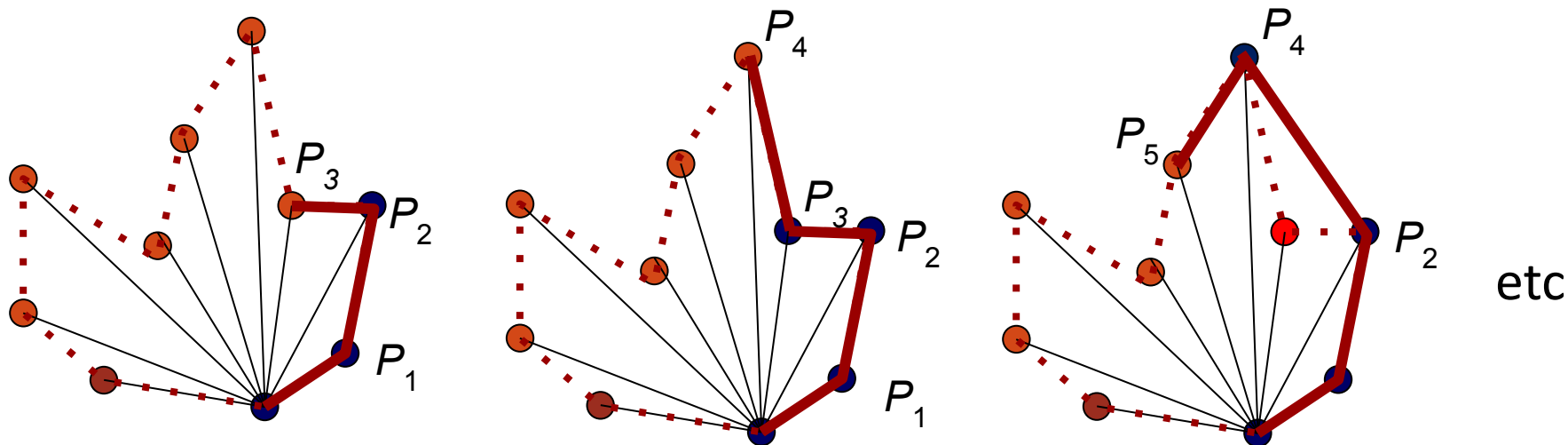
Traverse remaining points in order, adding each one to the hull, but backing up and discarding a point whenever we make a right turn.



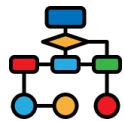


Step 2 in a bit more detail

- At each iteration, the next point in L is checked to see if it is on the left of the line connecting the last two points in H . If it is, the point is added to H , otherwise, the last point of H is popped.
- When all points in L have been processed the stack H contains the vertices of the convex hull in counter-clockwise order.



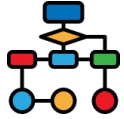
See on-line visualisation: <https://lobb.nz/convexhullvisualiser/convexhull.html>



Graham Scan: full pseudocode

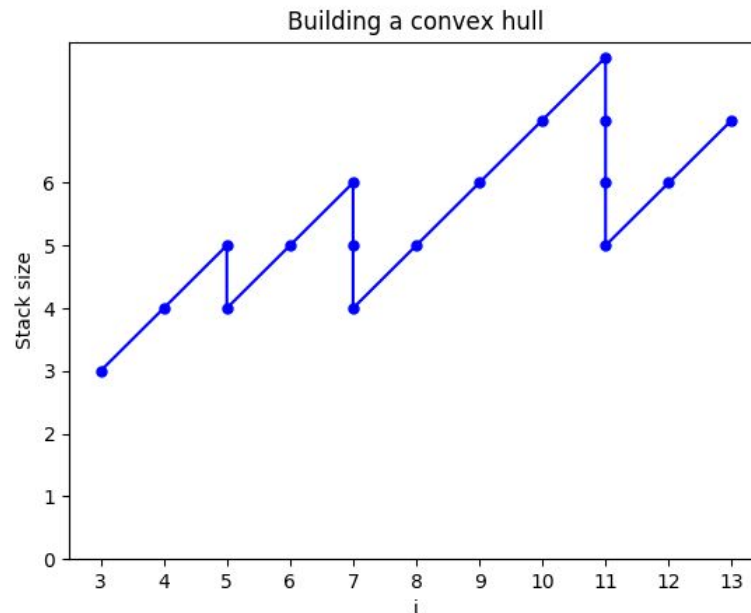
1. P_0 = lowest (and if necessary the left most) point
2. Sort all points in CCW order w.r.t. P_0 to form the list of points $L = [P_0, P_1, \dots, P_{n-1}]$
3. Stack $H = [P_0, P_1, P_2]$
4. **for** $i = 3$ to $n-1$: # Process each remaining point P_i in L
5. **while** not $\text{isCCW}(H[-2], H[-1], P_i)$
6. $H.\text{pop}()$
7. $H.\text{append}(P_i)$

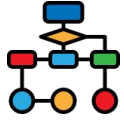
Step 1 takes $O(n)$ time.
 Step 2 takes $O(n \log n)$ time.
 Step 3 takes $O(1)$ time.
 What about loops 4, 5?



Graham Scan: Complexity

- The *for* loop in step 4 adds one element to the stack H .
- 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 leads to a 'sawtooth' pattern of execution:





Graham Scan: Complexity

- The sawtooth 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 $O(n \log n)$.