Group 9

Geometry Algorithm

Phạm Thạch Thanh Trúc
Phan Huỳnh Ngọc Trâm

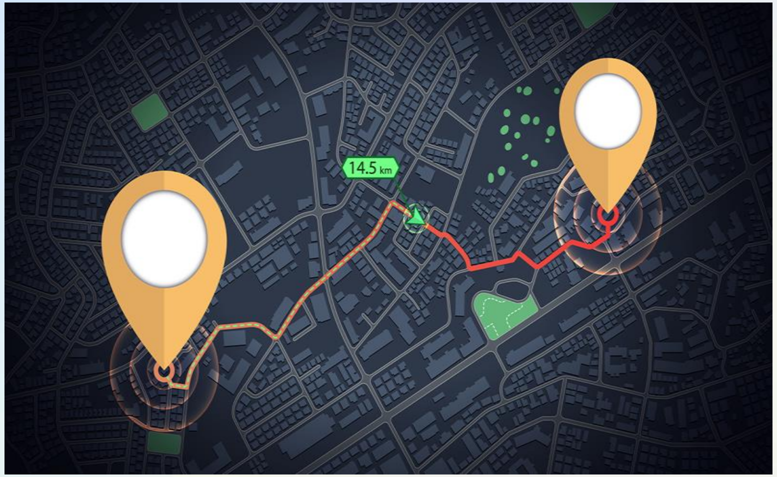
Table of contents

01

Overview

1. Overview

Q: What are some example's of geometric problem in our daily life?



1. Overview

Applications:

- VLSI design.
- Computer vision.
- Mathematical models
- Models of physical world
- Astronomical simulation
- Geographic information systems

1. Overview

Do you know: Most geometry algorithm are only 50-years-old !

For clarity:

- Graham's Scan, Jarvis March (Gift Wrapping), Quick Hull, and Divide and Conquer-based approaches have been around since the 1970s and 1980s.

- Divide and Conquer-based approaches for Closest Pair of Points were developed in 1970s.

02

Geometric Primitives

2. Geometric Primitives

- **Points:** Exact location in space. Determined by x and y on a 2-D plane.

```
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

- **Line:** a one-dimensional figure defined by points satisfying a linear equation:

$$ax + by + c = 0$$

```
class Line():
    def __init__(self, p1, p2):
        self.a = p1.y - p2.y
        self.b = p2.x - p1.x
        self.c = -(self.a * p1.x + self.b * p1.y)
```

2. Geometric Primitives

- **Line Segment:** Bounded by finite points along an infinite straight line that has two endpoints.

```
class LineSegment():
    def __init__(self, p, q):
        self.p = p
        self.q = q

    def length(self):
        return hypot(self.p.x - self.q.x,
            self.p.y - self.q.y) # Euclidean

A = Point(0, 0)
B = Point(2, 1)
line_segment = LineSegment(A, B)
```

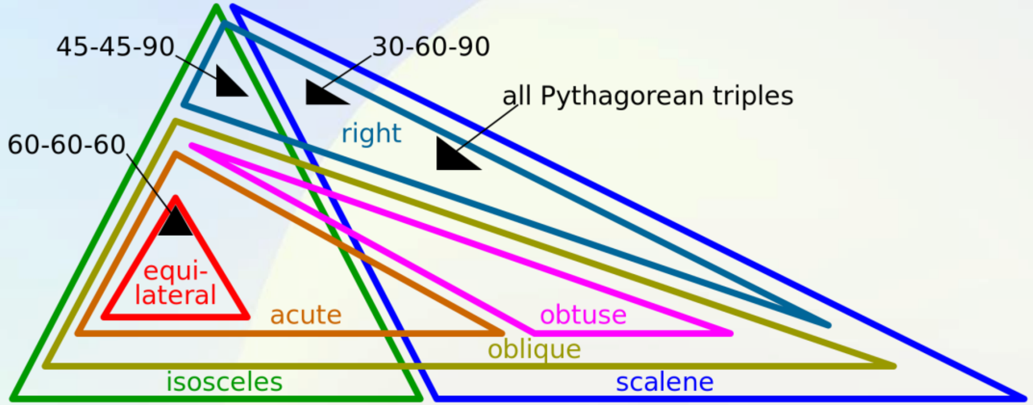
```
class Vector:
  def __init__(self, p, q):
    self.x = q.x - p.x
    self.y = q.y - p.y

  def magnitude(self):
    return hypot(self.x, self.y)
```

- **Vector:** Characterized by a magnitude and direction, emanating from a specific point in space and extends in the indicated direction.

2. Geometric Primitives

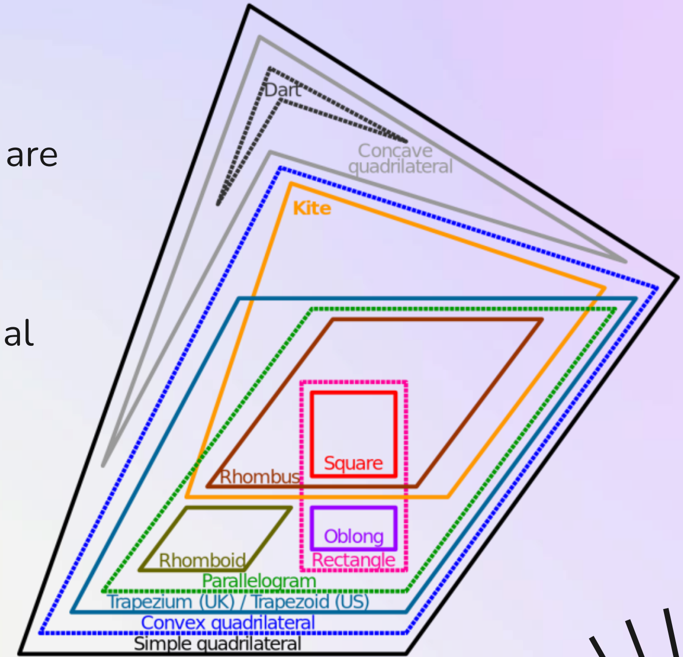
- **Triangle:** A polygon with 3 vertices and 3 edges. There are four main types:

  1. **Equilateral:** Three equal-length edges and three 60 degrees interior angle.
  2. **Isoscele:** Two equal-length edges and two equal interior angle.
  3. **Scalene:** All edges have different length
  4. **Right:** One 90 degrees angle (right angle).



2. Geometric Primitives

- ❑ **Quadrilaterals:** A polygon with 4 sides, 4 vertices and 4 angles. Some other special quadrilaterals:
  1. **Rhombus:** all four sides of equal length, but the angles are not necessarily right angles.
  2. **Parallelogram:** opposite sides that are parallel and equal in length, and opposite angles that are equal.
  3. **Trapezoid:** at least one pair of parallel sides. The other two sides may or may not be equal in length.



03

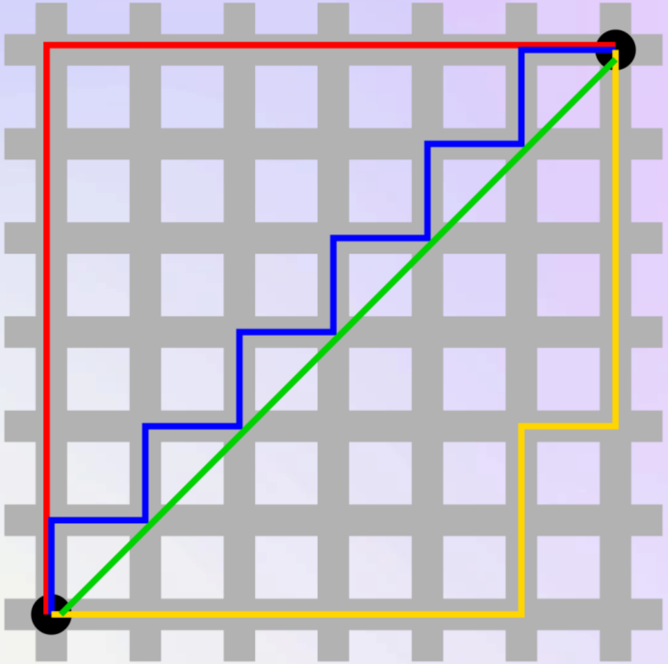
Geometry Function

3. Geometry Function

- ❑ Distance from Point to Point:

1. Euclid Distance: $\sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2}$

2. Manhattan Distance: $| x_Q - x_P | + | y_Q - y_P |$



3. Geometry Function

- ❑ **Dot product**

```
def DotProduct(v1, v2):
  return v1.x * v2.x + v1.y * v2.y
```

$$\vec{u} \cdot \vec{v} = x_1 x_2 + y_1 y_2$$

$$\vec{u} \cdot \vec{v} = |\vec{u}| \cdot |\vec{v}| \cdot \cos(\theta)$$

- ❑ **Cross product**

$$\vec{a} \times \vec{b} = \vec{n} \cdot |\vec{a}| \cdot |\vec{b}| \cdot \sin(\theta)$$

```
def CrossProduct(v1, v2):
  return v1.x * v2.y - v1.y * v2.x
```

3. Geometry Function

- ❑ Distance from points to line

```
def PointToLine(A, B, C, isSegment = False):
  AB, BA = vector(A, B), vector(B, A)
  AC, CA = vector(A, C), vector(C, A)
  BC, CB = vector(B, C), vector(C, B)
  if isSegment:
    if DotProduct(AB, CB) < 0:
      return BC.magnitude()
    if DotProduct(BA, CA) < 0:
      return AC.magnitude()
  dist = abs( CrossProduct(BA, CA)
  / AB.magnitude())
  return dist
```

- ❑ Angle between 2 vectors

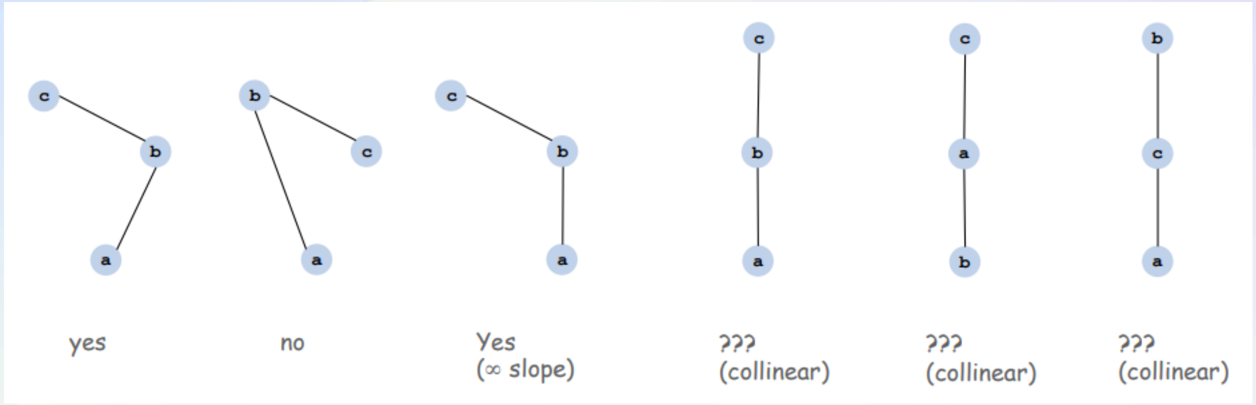
$$\cos(\theta) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| \cdot |\vec{v}|}$$

```
def getAngle(v1, v2):
  return acos( DotProduct(v1, v2) /
(v1.magnitude()*v2.magnitude()) )

O = Point(0,0)
A = Point(-6,8)
B = Point(5,12)
print(degrees(getAngle(vector(O,A),
vector(O,B))))
```

3. Geometry Function

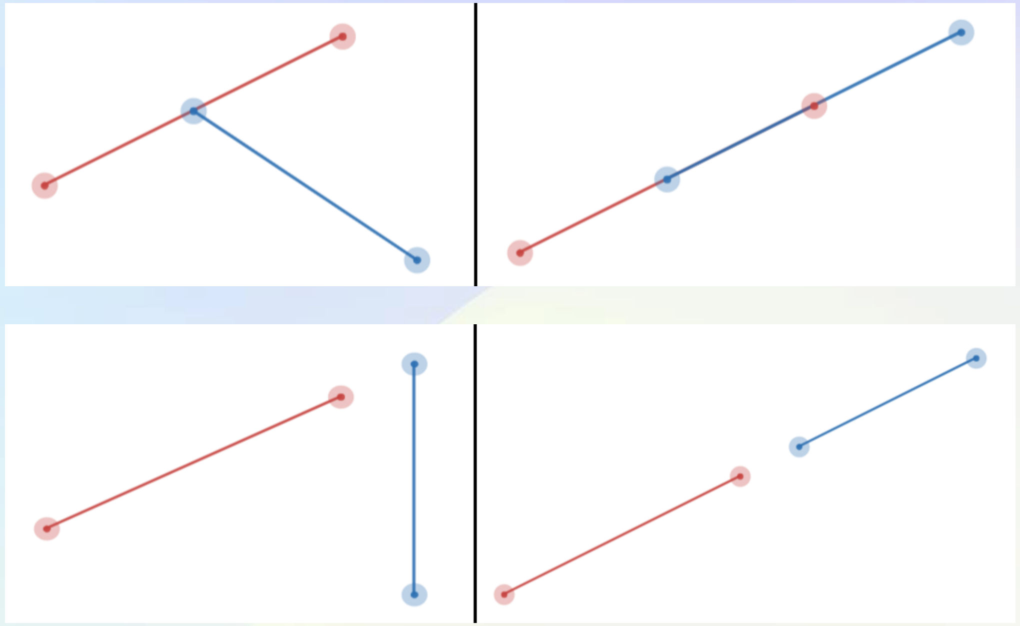
- **Clockwise & Counter-Clockwise**
  - Given three points **a**, **b** and **c**, is it in **CCW turn or not?**

```
def CCW(A, B, C):
    crossProduct = CrossProduct(A, B, C)
    if crossProduct == 0:
        return 0
    return 1 if crossProduct > 0 else -1
```



3. Geometry Function

❑ **Line intersection:**



- If 3 points collinear, check if one endpoint belongs to the other
- If no points collinear, segment AB and CD intersect when:
  - ✓ **C** and **D** have different orientation compares to **AB**
  - ✓ **A** and **B** have different orientation compares to **CD**

3. Geometry Function

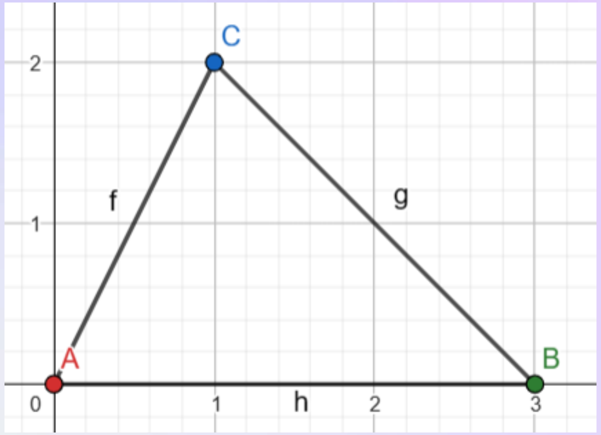
- ❑ **Polygon Presentation:** Enumerate the vertices of the polygon in either clockwise or counter-clockwise.
- ❑ **Perimeter of Polygon:** The sum of distances between ordered vertices

```
def PolygonPerimeter(polygon):
  result = 0.0
  for i in range(len(polygon) - 1):
    result += LineSegment(polygon[i], polygon[i + 1])
  return result
```

2. Geometric Primitives

Initialize a Triangle:

```
class Polygon():
  def __init__(self):
    self.points = []


Triangle = Polygon()  # Define a Triangle

Triangle.points.append(Point(0,0))
Triangle.points.append(Point(3,0))
Triangle.points.append(Point(1,2))

Triangle.points.append(Triangle.points[0])
```



3. Geometry Function

- **Area of Polygon:** Compute the determinant of the matrix

$$A = \frac{1}{2} \times \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_{n-1} & y_{n-1} \end{bmatrix} = \frac{1}{2} \times (x_0 \times y_1 + x_1 \times y_2 + \cdots + x_{n-1} \times y_0 - x_1 \times y_0 + x_2 \times y_1 + \cdots + x_0 \times y_{n-1}$$

```python
def PolygonArea(polygon):
  result = 0
  for i in range(len(polygon) - 1):
    x1 = polygon[i].x
    y1 = polygon[i].y
    x2 = polygon[i + 1].x
    y2 = polygon[i + 1].y
    result += (x1 * y2 - x2 * y1)
  return abs(result) / 2.0
```

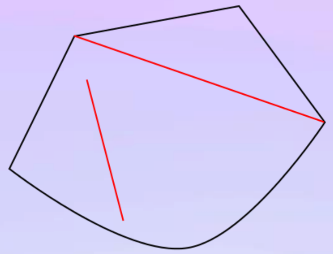
3. Geometry Function

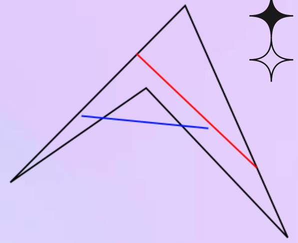
- **Check point is inside Polygon:** Compute the sum of angles between three points

$$\{P[i], pt, P[i+1]\}$$

```
def isPointInPolygon(point, polygon):
  size = len(polygon)
  if size <= 3:
    return False
  sum = 0
  for i in range(len(polygon) - 1):
    if CCW(point, polygon[i], polygon[i + 1]):
      sum += angle( polygon[i], point, polygon[i + 1])
    else:
      sum -= angle( polygon[i], point, polygon[i + 1])
  return abs(abs(sum) - 360) <= 1e-9
```

3. Geometry Function


convex


not convex

❑ Convex Polygon condition:

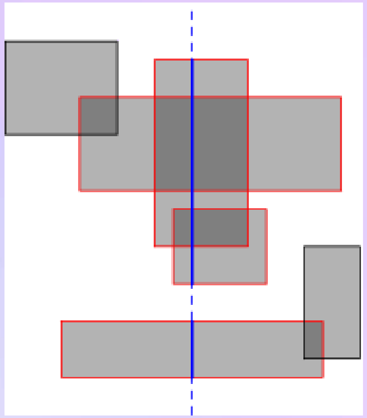
- Any line segment drawn inside the Polygon does not intersect any edge of Polygon
- Otherwise, it is called Concave

```
def isConvex(polygon):
    size = len(polygon)
    if size <= 3:
      return False
    isLeft = CCW(polygon[0], polygon.y,polygon[2])

    for i in range(1, size - 1):
        if CCW(polygon[i], polygon[i + 1],
           polygon[(1 if i+2 == size else i+2)]) != isLeft:
          return False

    return True
```

04

Sweep Line

4. Sweep Line



❑ **Algorithm’s strategy:**

✓ Represent an instance of the problem as a set of events that correspond to points in the plane.

✓ Don't need to keep track of the sweep line at all possible positions - only at the "critical" positions

✓ The events are processed in increasing order according to their **x** or **y** coordinates.

❑ **Complexity:**

✓ The running time is **O(n log n)**, because sorting the events takes **O(n log n)** time and the rest of the algorithm takes **O(n)** time.

4. Sweep Line

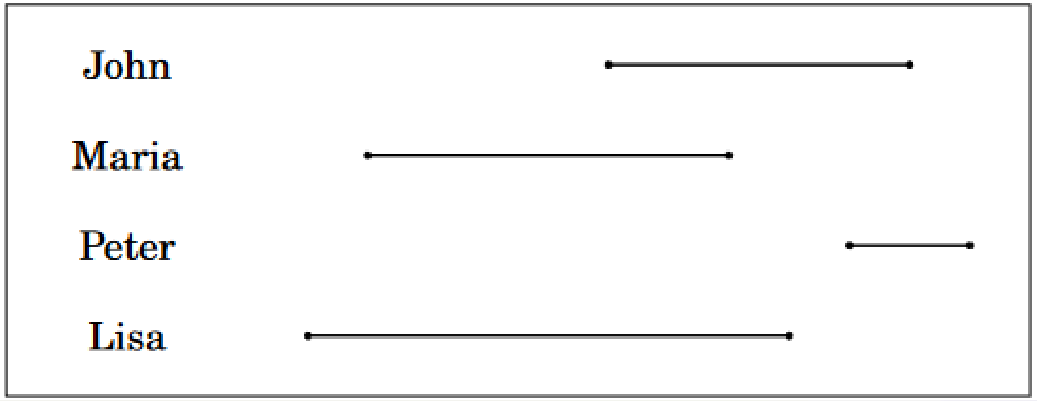
- **For example:**

- Kory and Aphe own a company that has **n employees**, and we know for each employee their arrival and leaving times on a certain day.
- Our task is to calculate the maximum number of employees that were in the office at the same time.

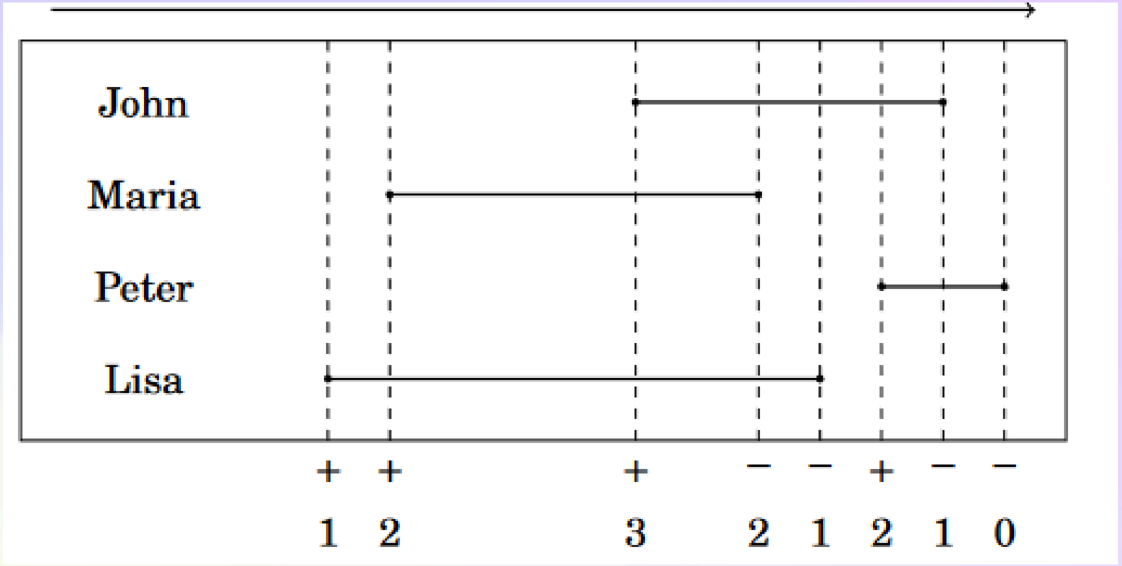
4. Sweep Line

person	arrival time	leaving time
John	10	15
Maria	6	12
Peter	14	16
Lisa	5	13

corresponds to the following events:



4. Sweep Line



For the following steps:

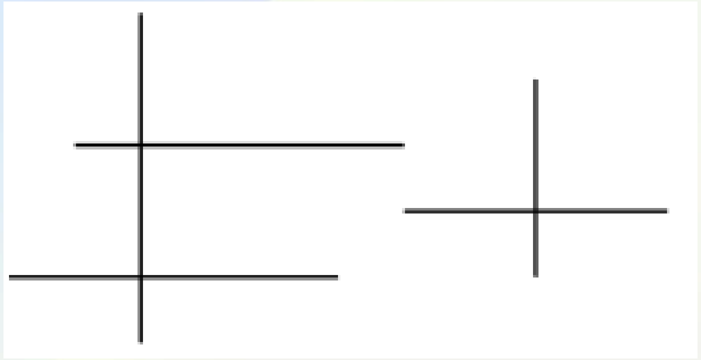
1. Set a counter from left to right
2. Increase the counter by 1 if a person arrives
3. Decrease the counter by 1 if a person leaves
4. Answer is the maximum value of the counter

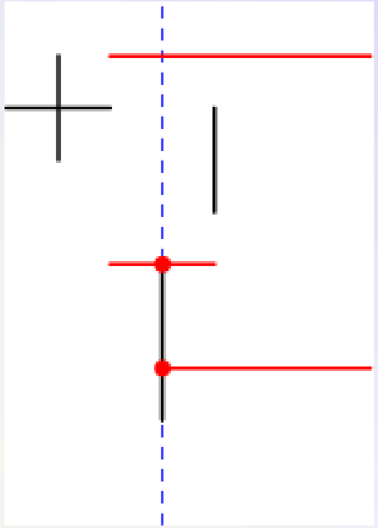
4. Sweep Line

❑ Intersection points

Task: Given a set of **n** line segments. Consider the problem of counting the total number of intersection points.

For example:

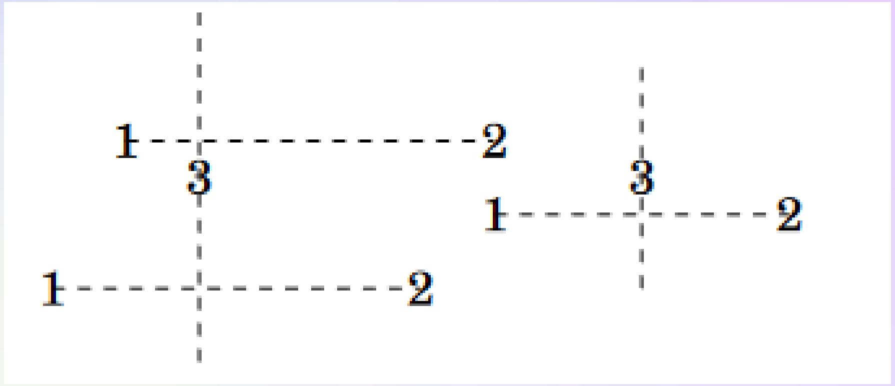




4. Sweep Line

Idea: Process the endpoints of the line segments from left to right

- Focus on three types of events:
  1. Horizontal segment begins (1)
  2. Horizontal segment ends (2)
  3. Vertical segment (3)



- Go from left to right:
  - At (1): Add **y** coordinate to the set
  - At (2): Remove **y** coordinate from the set
  - At (3): Check if there's a **y** coordinate between **y1** and **y2** (of the vertical line)

4. Sweep Line

- ✓ Easy to solve in **O(N²)**

Go through all possible pairs of line segments

and check if they intersect

- ✓ Can solve in **O(n log n)** using Line Sweep

4. Sweep Line

❑ Closest Pair of Points

Task: Find the closest Pair of Point in a set of *n* points

- ➢ Brute Force
- ➢ Divide and Conquer

Application:

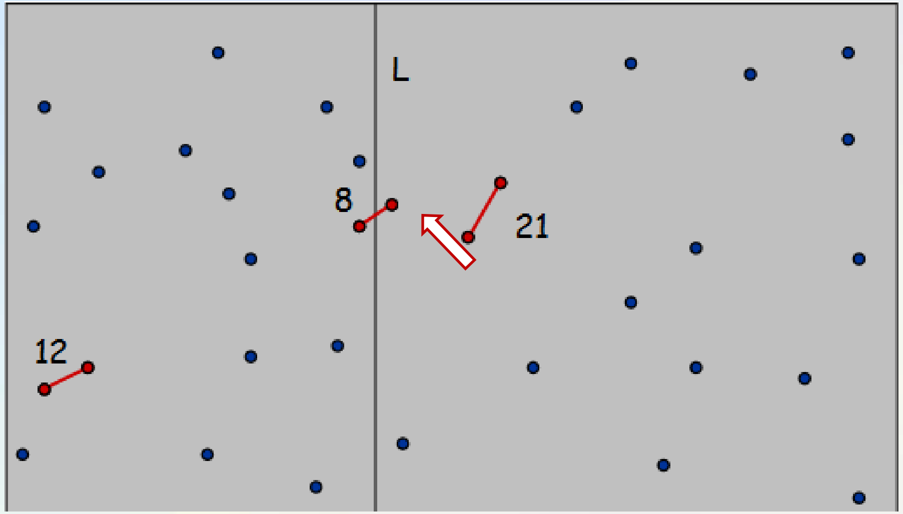
- ➢ In aviation, find the closest pair of aircraft to each other that are likely to have a collision.
- ➢ In the postal service, find the 2 post offices closest together to close either.
- ➢ In machine learning, used in cluster analysis in statistics (Hierarchical Clustering).

4. Sweep Line

Divide and Conquer:

Q: There are multiple points in 2D plane. Find the closet distance between two points?

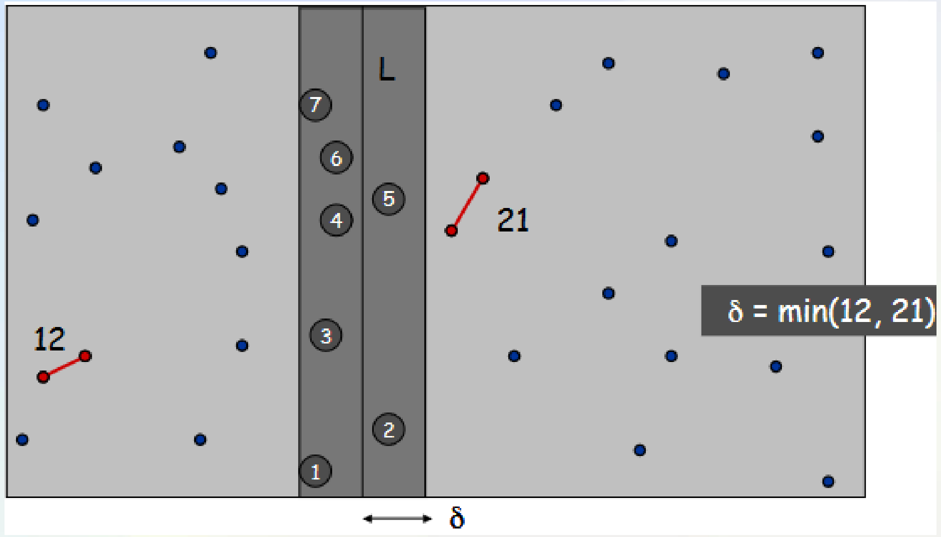
Approach: Divide into sub-planes and find closet pair in each side recursively



What if the correct answer is the distance between two points in different sub-plane?

4. Sweep Line

- Call the left side is **A**, the right side is **B**
- Assuming the minimum distance of points between left is δ_A and right sides is δ_B
- Let $\delta = \min(\delta_A, \delta_B)$



- Observe points lie within δ of line **L**
- Sort points in strip by their y coordinate
- Check distances of those within the sorted list

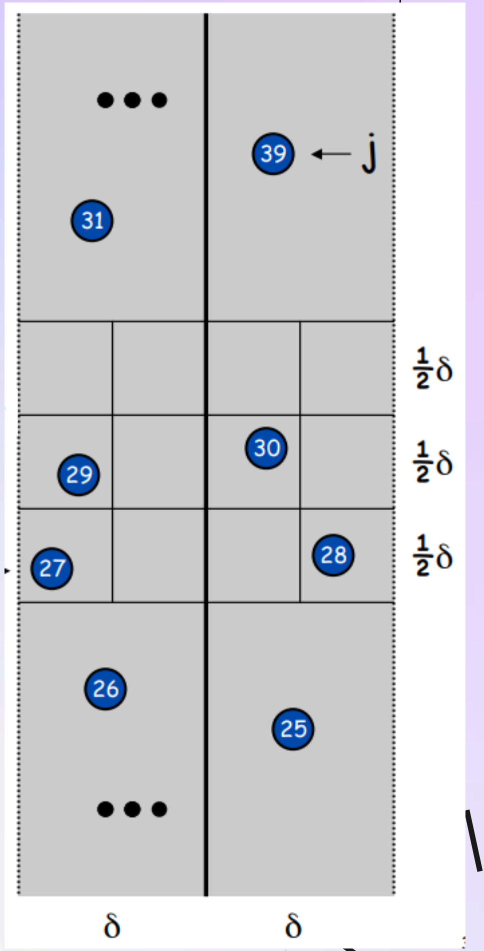
4. Sweep Line

- What is the running time of this?
- How many point do we need to check for each **Pi**?
- **Scanning the strip:**

```
For i = 1 to r:
    For j = i + 1 to min(i + 7, r):
        Check pair si, sj
```

4. Sweep Line

- **Claim:**
  - If $|i - j| \le 7$, then the distance between **si** and **sj** is at least δ
- **Proof:**
  - No two points lie in same **½δ-by-½δ box**.
  - Two points at least 2 rows apart have **distance ≥ 2(½δ)**



4. Sweep Line

```
Closest-Pair(p1, ..., pn) {
    Compute separation line L # half the points are on one side and half on the other      O(n log n)
    δ1 = Closest-Pair(left half)                                                          2T(n / 2)
    δ2 = Closest-Pair(right half)
    δ = min(δ1, δ2)
    Delete all points further than δ from separation line L                               O(n)
    Sort remaining points by y-coordinate.                                                O(n log n)
    Scan points in y-order and compare distance between
    each point and next 11 neighbors.                                                     O(n)
    If any of these distances is less than δ, update δ.
    return δ.
}
```

4. Sweep Line

- **Running time:**

$$T(n) \le 2T(\frac{n}{2}) + O(n \log n)$$
$$\Rightarrow T(n) = O(n \log^2 n)$$

- **Q:** Can we improve the running time?

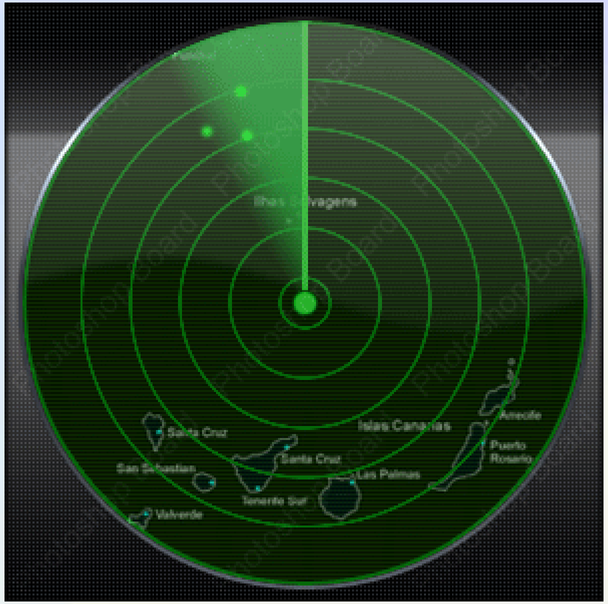
  - Sort all points by y at the beginning
  - Divide preserves the y-order of points

$$\Rightarrow O(n)$$

Finally, it becomes $T(n) = O(n \log n)$

4. Sweep Line

Radial Sweep involves a ray that rotates around a Central Point (like a Radar screen)



- ✓ In this case, we sort points / events by their bearing instead of by their **x**- and **y**-coordinates.
- ✓ Besides that, the mechanics are the same as those of normal line sweep.

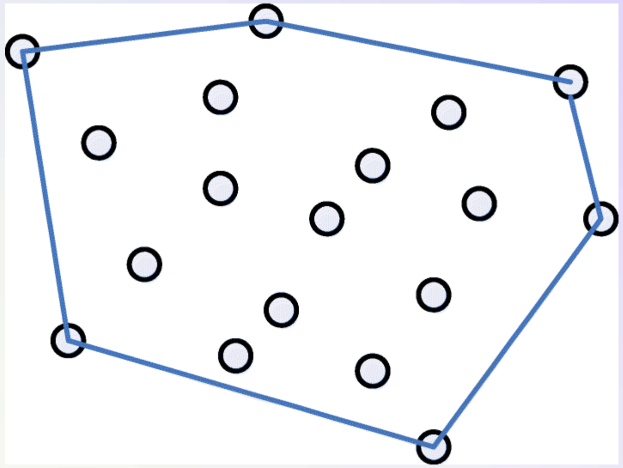
05

Convex Hull

5. Convex Hull

Task: Find the Convex Hull of the set of points.

- ❑ The convex hull is defined as a smallest convex polygon containing this entire set of points
- ❑ Two main ways to solve:
  - Brute Force
  - Graham Scan



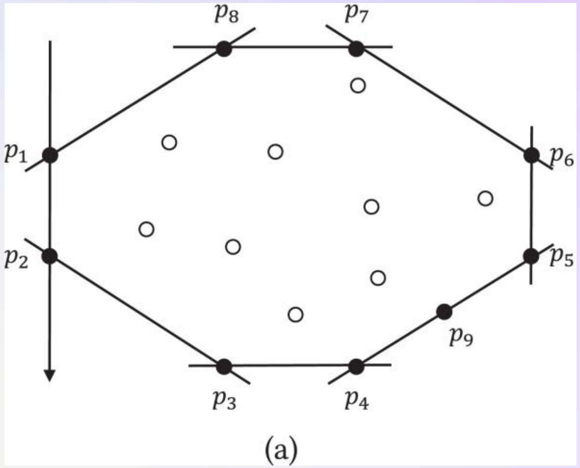
5. Convex Hull

- **Brute Force:** Given that a Line segment belongs to a Convex Hull if and only if all the remaining points are on the same side of the Line segment

  - Line segment equation: $ax + by - c = 0$
  - The equation divide into 2 sides:

$$ax + by - c > 0$$
$$ax + by - c < 0$$

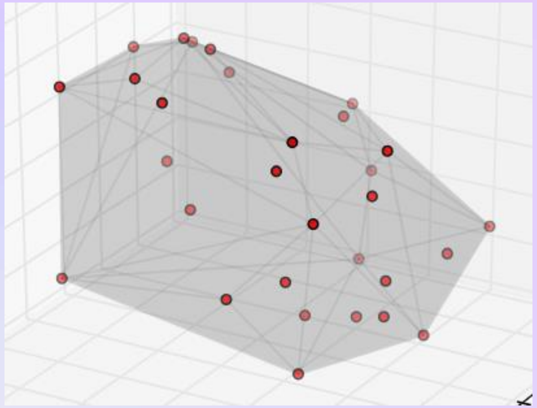
  - If all other points are lying on one side, that Line segment is an edge of Convex Hull



5. Convex Hull

- Brute Force

1. List all possible pair of points (**n/2 pairs**)
2. For each **pair (p1, p2)**, check if all other points lie to one side. If yes, add the pair to Convex Hull
3. Repeat for all pairs, then remove the duplicate edges
4. Connect the edges in order to form Convex Hull

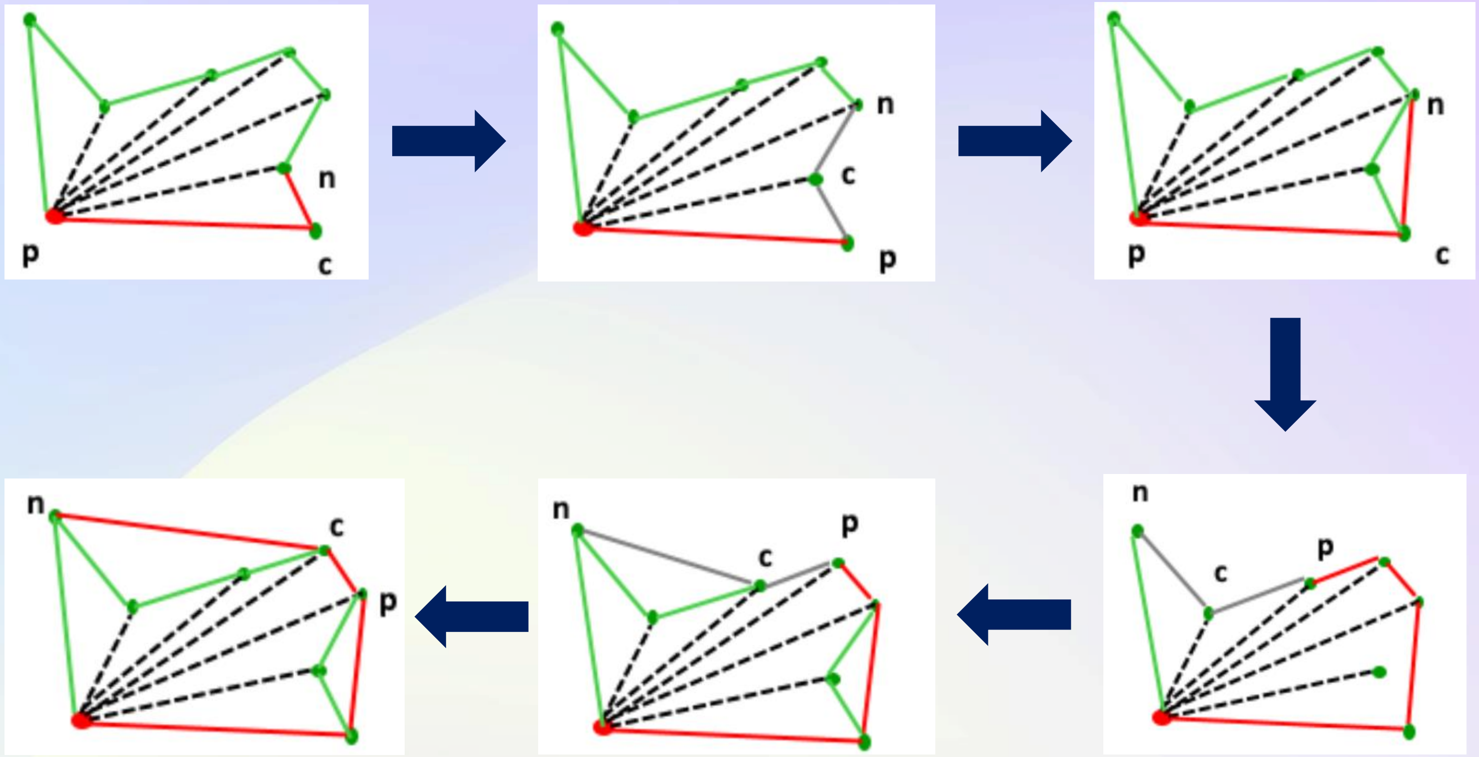


5. Convex Hull

- **Graham Scan:**

1. Define **Point O** which belong to convex hull as an anchor (usually bottom left point)
2. Call the current Convex Hull set **H**
3. Start from **O**, iterate through points with each point called **P**
4. If the line segment created by **P** and the last point in **H** is Clockwise, remove **P**, return to the last point in **H** and choose another point
5. If it is Counter Clockwise, move to the next point. Repeat until back to **O**

5. Convex Hull



5. Convex Hull

```
let points be the list of points
let stack = empty_stack()
find the lowest y-coordinate and leftmost point, called P
sort points by polar angle with P, if several points have the same polar angle then only
keep the farthest


for point in points:
    # pop the last point from the stack if we turn clockwise to reach this point
    while count stack > 1 and ccw(next_to_top(stack), top(stack), point) <= 0:
        pop stack
    push point to stack
end
```

THANKS FOR LISTENING