香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Computational Geometry

Ethen Yuen {ethening}

2023-04-15

Introduction

- Tasks that involves Computational Geom is usually a time sink, demanding high level of implementation accuracy.
- Issues that you will run into: (included but not limited to)
  - Mess up the handling on <= and <, >= and >
  - Precision error
  - Special case where the geometric object degenerate (plane -> line, line -> point)
  - Special case where slope is equal to infinity
  - Special case where some points may be collinear and would cause problems
  - ...
  - Why is my program still incorrect??
- So what should you do when you see Geom task? ~~Run~~

Introduction

- Luckily, it is a rare sight to see such tasks appear in secondary-level OI contests (often appear in ICPC).
- Also, IOI syllabus have a rather limited scope on the items tested. (see next page)
- You should have the basic knowledge on how to approach this kind of tasks.

IOI Syllabus

AL10. Geometric algorithms

In general, the ISC has a strong preference towards problems that can be solved using integer arithmetics to avoid precision issues. This may include representing some computed values as exact fractions, but extensive use of such fractions in calculations is discouraged.

Additionally, if a problem uses two-dimensional objects, the ISC prefers problems in which such objects are rectilinear.

- ✓ Representing points, vectors, lines, line segments.
- ✓ Checking for collinear points, parallel/orthogonal vectors and clockwise turns (for example, by using dot products and cross products).
- ✓ Intersection of two lines.
- ✓ Computing the area of a polygon from the coordinates of its vertices.[19]
- ✓ Checking whether a (general/convex) polygon contains a point.
- ✓ Coordinate compression.
- ✓ $\mathcal{O}(n \log n)$ time algorithms for convex hull
- ✓ Sweeping line method
- ✗ Point-line duality
- ✗ Halfspace intersection, Voronoi diagrams, Delaunay triangulations.
- ✗ Computing coordinates of circle intersections against lines and circles.
- ✗ Linear programming in 3 or more dimensions and its geometric interpretations.
- ✗ Center of mass of a 2D object.
- ✗ Computing and representing the composition of geometric transformations if the knowledge of linear algebra gives an advantage.

5.3 Other Areas in Mathematics

- ✗ Geometry in three or more dimensions.

Today's Scope

[Mostly in 2-dimensional space]

- Point, line, line segment
- Dot Product & Cross Product
- Line Intersection
- Polygon
- Convex Hull
- Common tricks on Computational Geometry

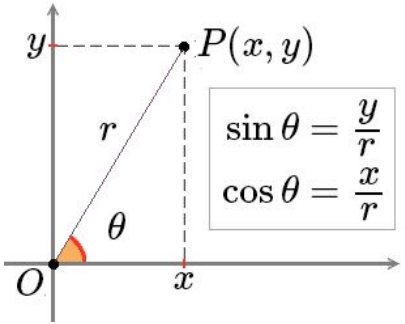
- We are going to use vector for most of the calculations, so it will be introduced as well.

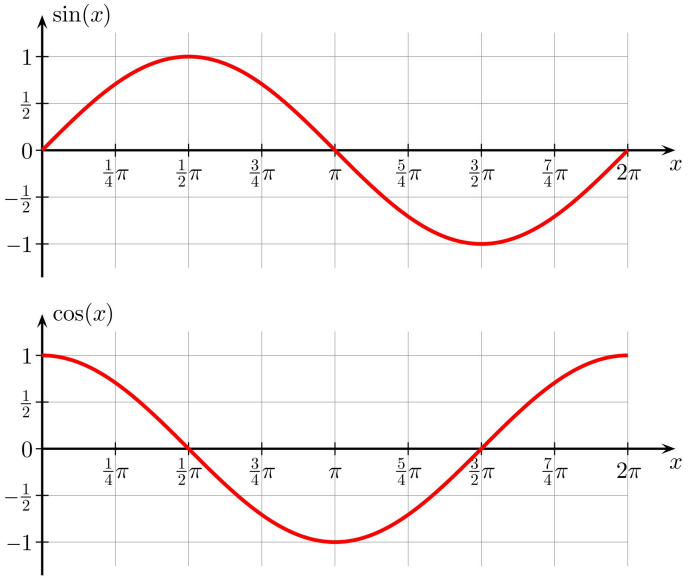
Practice Task

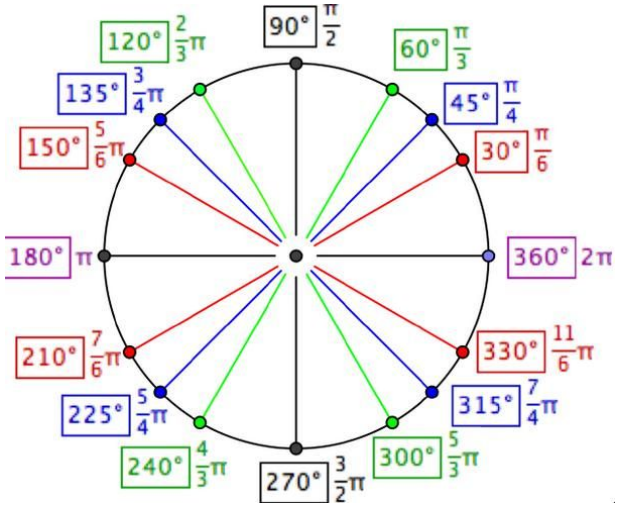
- If you think you have already learned everything in this lecture: JOI Spring Camp 2014 Day 4 T1 “Constellation 2”
- You may submit and read translation (Chinese) in LOJ: https://loj.ac/p/2882

Some Prerequisites

- Hopefully you have learn about Trigonometry and angle in radians







Some Prerequisites

- And know about some basic **precision handling** techniques:
  - If the problem allows, try to use **int** or **long long** to avoid precision errors!
  - Float number only store a approximation of the number due to its representation, and when you do arithmetic with approximation, the result is also approximation (it is not necessary the value you want it to be, just close enough!), e.g. `0.1 + 0.2 == 0.3` would return false.
  - Handle the precision errors carefully, usually by using ε (EPS):
  - We usually set const double EPS = 1e-9 (or 1e-6)
    - `a == b → fabs(a - b) < EPS`
    - `a < b → a + EPS < b`
    - `a <= b → a < b + EPS`
- For example,
  - `operator==(const Point& T) const { return fabs(x - T.x) < EPS && fabs(y - T.y) < EPS; }`

Point

- The most important foundation of it all, simply use 2 **double** to store the x- and y-coordinates of a point.
- Sometimes, **int** or **fraction** is used instead to avoid precision issue.

```
struct Point {
    double x = 0, y = 0;
    Point(double x = 0, double y = 0) : x(x), y(y) {}
};
```

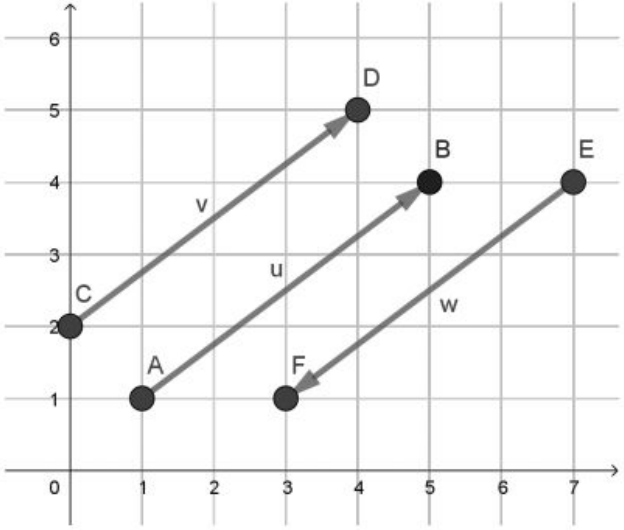
Line

- The most common way to store a line is composed of two points that lies on that line.
- Whether it refers to a line (infinite line) or a line segment (bounded by 2 endpoints) is up to you.
- There are other forms to store a line, but this is the most simple and versatile one.

```
struct Line {
    Point p1, p2;
    Line(Point p1, Point p2) : p1(p1), p2(p2) {}
};
```

Vector

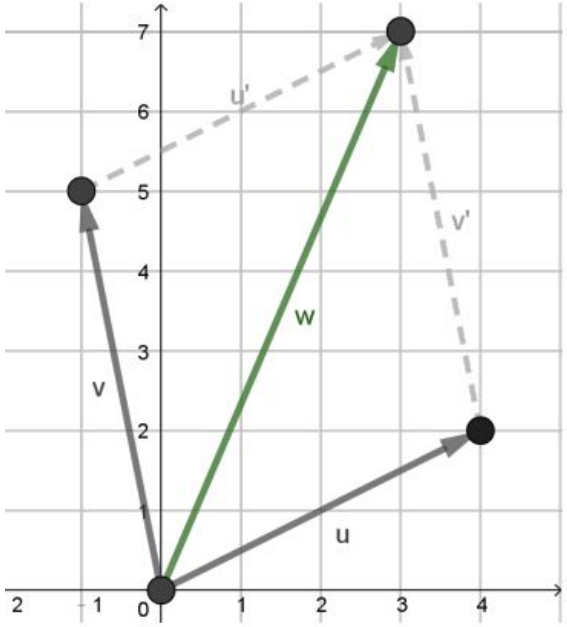
- A quantity with magnitude (length) and direction.
  - Opposed to Scalar, a quantity with only magnitude
- It has no positional information alone, but it can be "attached" onto a initial point.
- Geometrically, the vector from A to B is denoted by $\overrightarrow{AB}$
- Computationally, 2D vector is usually represented by 2 values, the displacements of x and y, denoted by **<dx, dy>**.
- It can be viewed interchangeably with a 2D point, because a 2D point can be represented by a vector attached onto the origin **O(0, 0)**.



- $\overrightarrow{AB} = \vec{u} = (+4, +3)$
- $\overrightarrow{CD} = \vec{v} = (+4, +3)$
- $\overrightarrow{EF} = \vec{w} = (-4, -3)$
- $\vec{u} = \vec{v} \neq \vec{w}$, while $\vec{w} = -\vec{u}$

Vector Calculation

- The good thing about vector is that it comes with many predefined arithmetic rules.
- Vector addition: **<a, b> + <c, d> = <a + c, b + d>**
- Vector subtraction: **<a, b> - <c, d> = <a - c, b - d>**
- Scalar multiplication: **k<a, b> = <ka, kb>**
- Norm (Length): **|<a, b>| = sqrt($a^2 + b^2$)**
- Dot Product, Cross Product: (explain later)



- $\vec{u} = (+4, +2), \vec{v} = (-1, +5)$
- $\vec{w} = \vec{u} + \vec{v} = (4 - 1, 2 + 5) = (+3, +7)$
- $\vec{u} = \vec{w} - \vec{v} = \vec{w} + (-\vec{v})$
- $|\vec{u}| = \sqrt{4^2 + 2^2} = 2\sqrt{5}, |\vec{v}| = \sqrt{(-1)^2 + 5^2} = \sqrt{26}$

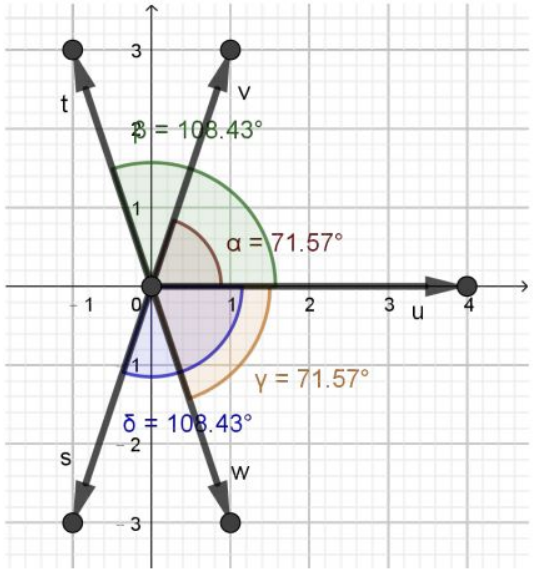
Point

- We can define some function and overload some more operators.

```
struct Point {
    double x = 0, y = 0;
    Point(double x = 0, double y = 0) : x(x), y(y) {}
    Point operator+(Point p) const { return Point(x + p.x, y + p.y); }
    Point operator-(Point p) const { return Point(x - p.x, y - p.y); }
    Point operator*(double s) const { return Point(x * s, y * s); }
    Point operator/(double s) const { return Point(x / s, y / s); }
    double dist2() const { return x * x + y * y; }
};
```

Dot Product

- Dot product of two vectors: $\vec{u} \cdot \vec{v}$
- Computationally, **<a, b> · <c, d> = ac + bd**
  - Could be extended into more dimensions:
    <a1, a2, …, am> · <b1, b2, …, bm> = a1 b1 + a2 b2 +…+ am bm
- The result is a scalar, instead of a vector
- Geometrically, $\vec{u} \cdot \vec{v} = |\vec{u}||\vec{v}|cos(\theta)$
  where **θ** is the angle between the two vectors.
- Notably, when the 2 vectors are orthogonal (angle is **π/2**), $\vec{u} \cdot \vec{v} = 0$
- And when a vector dot product with itself:
  $\vec{u} \cdot \vec{u} = |u|^2$



- $\vec{u} \cdot \vec{s} = \vec{u} \cdot \vec{t} = 4\sqrt{10}\cos 108.43° = -4$
- $\vec{u} \cdot \vec{v} = \vec{u} \cdot \vec{w} = 4\sqrt{10}\cos 71.57° = 4$

Dot Product

- Quick question to make sure you are following!
- Given n vectors in m dimensions: V1, V2, …, Vn
  (1 <= n <= 1e5, 1 <= m <= 10, -1e4 <= Vij <= 1e4)
- Could you find the maximum dot product of any 2 vectors,
  max(1 <= x <= y <= n) Vx · Vy?

- Could you do this in a fast enough method?

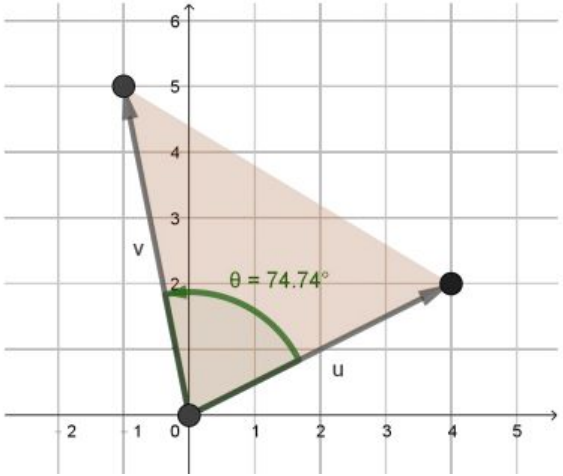
Dot Product

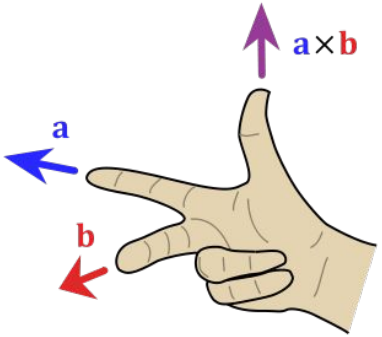
- Could you find the maximum dot product of any 2 vectors,
  max(1 <= x <= y <= n) Vx · Vy

- It can be shown that the maximum dot product must come from the dot product of a vector and itself. Why?
- Think about the geometric definition of dot product:
  $\vec{u} \cdot \vec{v} = |\vec{u}||\vec{v}|cos\theta$

Cross Product

- Cross product of two vectors: $\vec{u} \times \vec{v}$
- Computationally, **<a, b> x <c, d> = ad - bc**
  - (the result is a vector with a direction of z)
  - The order of the operands matters
  - Defined in 3-dimensional space
- Geometrically, $\vec{u} \times \vec{v} = |\vec{u}||\vec{v}|sin(\theta)\vec{n}$
  where **θ** is the angle **from u to v**, and **n** is a unit vector perpendicular to plane with u, v.
- In 2D Geometry though, we only care about the sign and value, since there are only 2 direction (into the plane, or out from the plane)





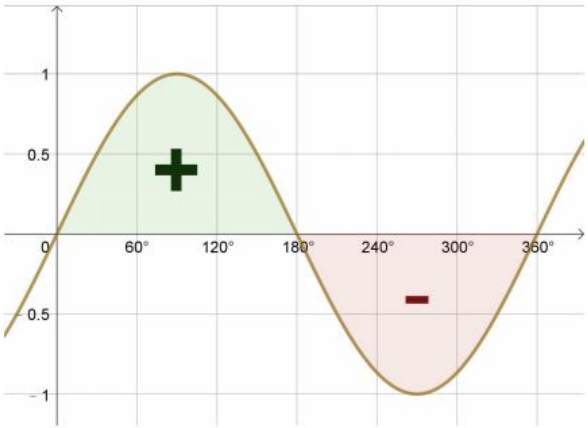
Products

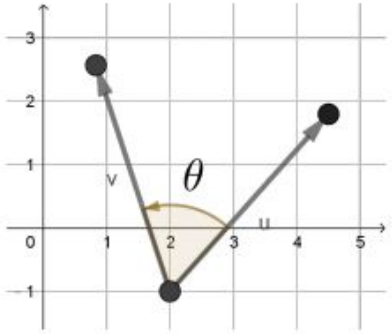
- You could keep adding these functions to your template.

```
struct Point {
    double x = 0, y = 0;
    Point(double x = 0, double y = 0) : x(x), y(y) {}
    Point operator+(Point p) const { return Point(x + p.x, y + p.y); }
    Point operator-(Point p) const { return Point(x - p.x, y - p.y); }
    Point operator*(double s) const { return Point(x * s, y * s); }
    Point operator/(double s) const { return Point(x / s, y / s); }
    double dist2() const { return x * x + y * y; }
    double dot(Point p) const { return x * p.x + y * p.y; }
    double cross(Point p) const { return x * p.y - y * p.x; }
};
```

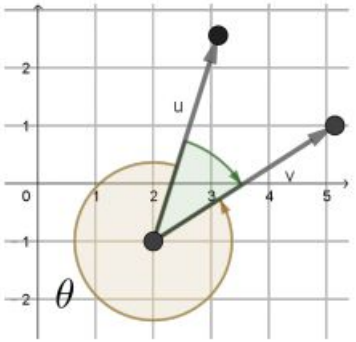
Turning Direction

- Let's focus on the sign of the cross product first: $\vec{u} \times \vec{v} = |\vec{u}||\vec{v}|sin(\theta)\vec{n}$
- u and v are parallel (same or opposite direction): $\vec{u} \times \vec{v} = 0$
- u -> anticlockwise turn -> v: $\vec{u} \times \vec{v} > 0$
- u -> clockwise turn -> v: $\vec{u} \times \vec{v} < 0$



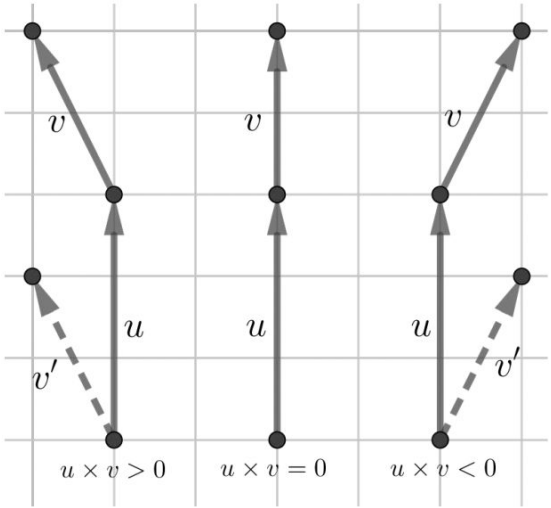


$\vec{u} \times \vec{v} > 0$



$\vec{u} \times \vec{v} < 0$

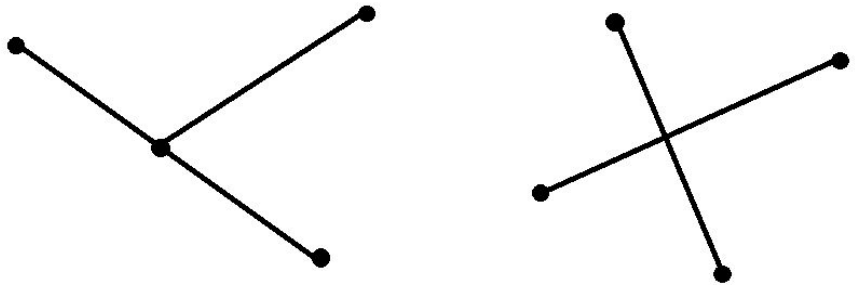
Turning Direction




```
int turn(Point a, Point b, Point c) {
    double res = (b - a).cross(c - b);
    if (res < 0) return 1; // a -> b -> c: right turn
    else if (res == 0) return 0;  // a -> b -> c: no turn
    else return -1; // a -> b -> c: left turn
}

int main() {
    Point P0(0, 0), P1(0, 3);
    cout << turn(P0, P1, Point(-1, 5)) << endl; // -1
    cout << turn(P0, P1, Point(0, 5)) << endl; // 0
    cout << turn(P0, P1, Point(1, 5)) << endl; // 1
    return 0;
}
```

Line Intersection

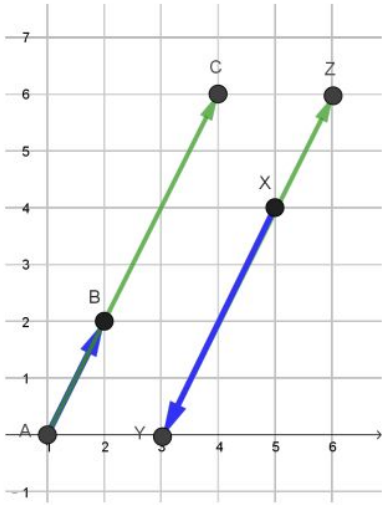
- Practice question!
- Given 2 line segments (2 pairs of endpoints), determine if they are intersected.
- Two cases to be considered:
- 1. Intersect but not crossing, 2. Intersect and crossing



Line Intersection

1. Intersect but not crossing

- This is easy to check if we know how to check if a point lies on another segment



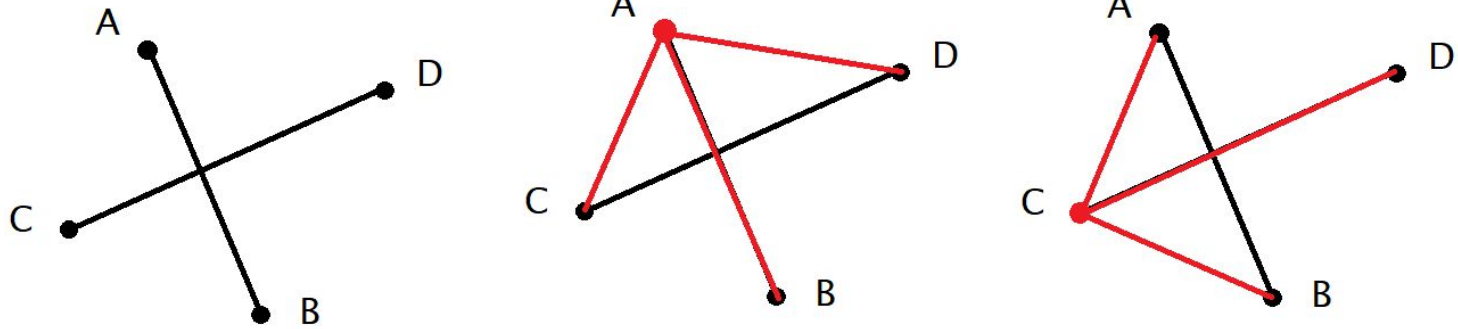

```cpp
bool onSegment(Point p, Line L) {
    double cross = (L.p1 - p).cross(L.p2 - p);
    double dot = (L.p1 - p).dot(L.p2 - p);
    // Parallel but in opposite direction
    return cross == 0 && dot <= 0;
}


int main() {
    Point A(1, 0), B(2, 2), C(4, 6);
    Point X(5, 4), Y(3, 0), Z(6, 6);

    cout << onSegment(A, Line(B, C)) << endl; // 0
    cout << onSegment(B, Line(A, C)) << endl; // 1

    cout << onSegment(X, Line(Y, Z)) << endl; // 1
    cout << onSegment(Y, Line(Z, X)) << endl; // 0
    cout << onSegment(Y, Line(Y, Z)) << endl; // 1
    return 0;
}
```

Line Intersection

2. Intersect and crossing

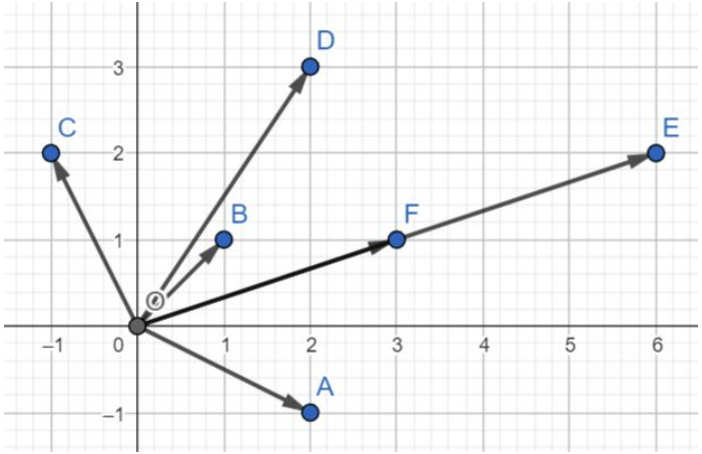
- Using A as base point, B should be in middle of C and D
- Using C as base point, D should be in middle of A and B

  (AD x AB)(AB x AC) > 0 && (CA x CD)(CD x CB) > 0
- [Consecutive clockwise turn or anticlockwise turn]



Polar Sort

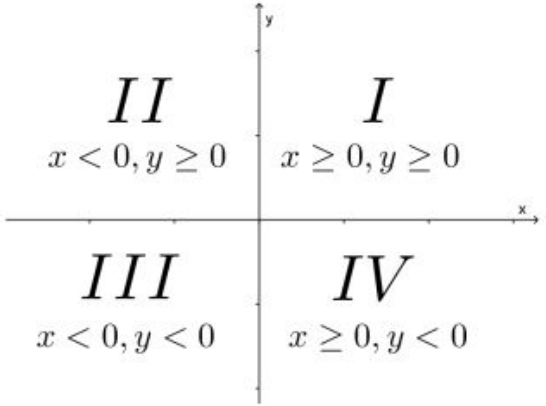
- In many tasks, it is useful to sort the points with their polar angle, and cross product provide a way without calculating the angle




```cpp
int main() {
    Point A(2, -1), B(1, 1), C(-1, 2);
    Point D(2, 3), E(6, 2), F(3, 1);
    vector<Point> vec = {A, B, C, D, E, F};
    sort(vec.begin(), vec.end(), [](Point &u, Point&v) {
        if (u.cross(v) == 0) return u.x < v.x; //collinear
        else return u.cross(v) > 0;
    });
    for (auto pt : vec) {
        cout << pt.x << " " << pt.y << endl;
    }
    /*  2 -1 -> A
        3 1 -> F
        6 2 -> E
        1 1 -> B
        2 3 -> D
        -1 2 -> C */
}
```

Polar Sort

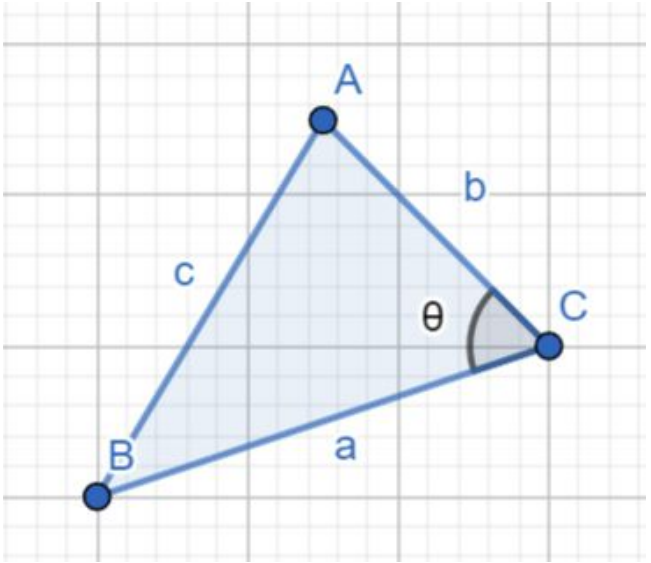
- However, the previous version only works when all of the points are within a 180° angle, because the sorting order warp around (and you would reach the contradiction that a point should be put in front of itself)
- This is still useful in many cases where all the points are on the same side of a line.



- A version that handles this issue is to first compare the point by the quadrant it was in, then sort by cross product. (Try to think about this yourself!)

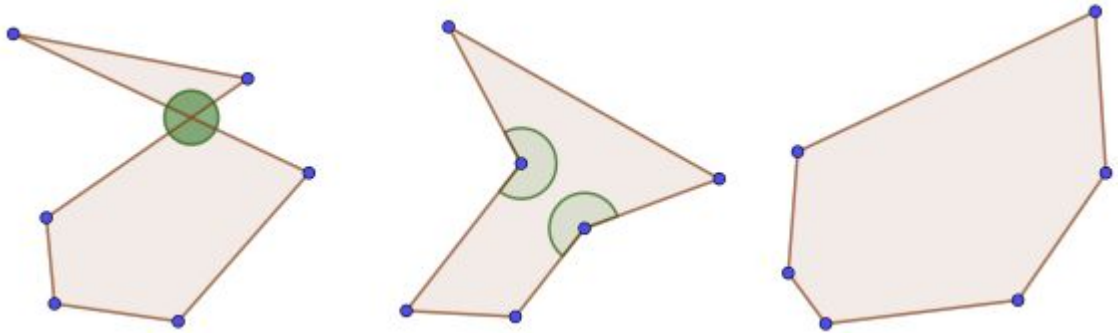
More on Cross Product

- Beside the sign of the cross product, the value actually have a meaning too.
- Recall that in DSE Maths, you should have learn this area formula: **Area = 1/2 ab sin θ**
- We also know that **CA x CB = ab sin θ**
- **Signed Area =** $\frac{1}{2}(\vec{u} \times \vec{v})$
- Where the sign is decided by the vector orientation. This fact will be useful later.



Polygon

- First, you must know how to distinguish between the following polygon:
- **Simple polygon:** no self-intersection except at vertices connecting neighboring line segments
- **Convex polygon:** A simple polygon with every interior angle no more than π (or 180°)
- **Strictly convex polygon:** A simple polygon with every interior angle less than π (or 180°)



Non-simple polygon Concave polygon Convex polygon

Polygon

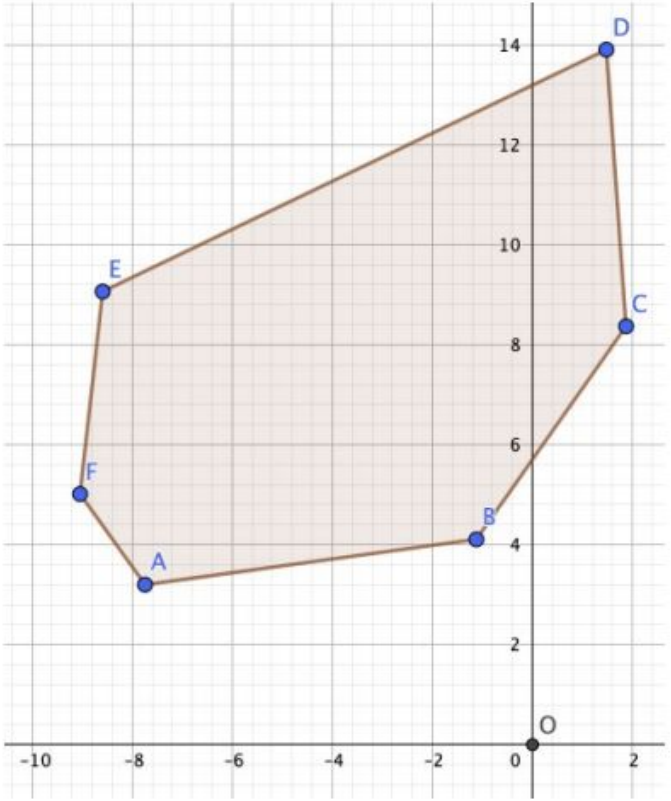
- How to store a polygon with a program?
- Simply, as a list of points: **p0, p1, ..., pN-1**
- Where there is a line segment connecting
  - pi and pi+1 for all 0 <= i < N - 1
  - pN-1 and p0
- We usually stored the points in counterclockwise direction

- The starter question for polygon is: how to calculate its area given its points?

Polygon

- For a polygon, we can divide the polygon into N triangles and sum their signed areas together.

$$Area = \frac{1}{2} \sum_{i=0}^{N-1} \overrightarrow{OP_i} \times \overrightarrow{OP_{i+1}}$$

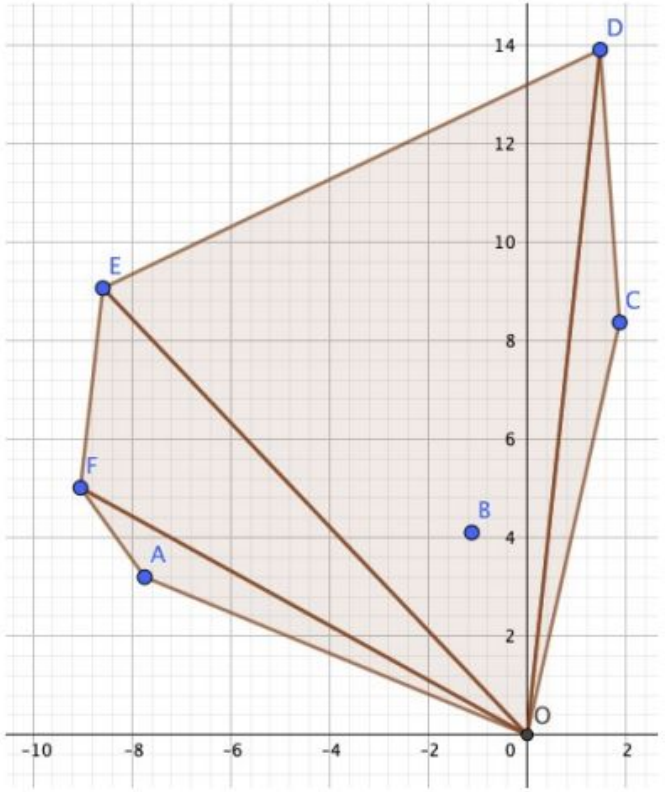


Polygon

- For a polygon, we can divide the polygon into N triangles and sum their signed areas together.

$$Area = \frac{1}{2} \sum_{i=0}^{N-1} \overrightarrow{OP_i} \times \overrightarrow{OP_{i+1}}$$

- The triangles with **positive** areas are:

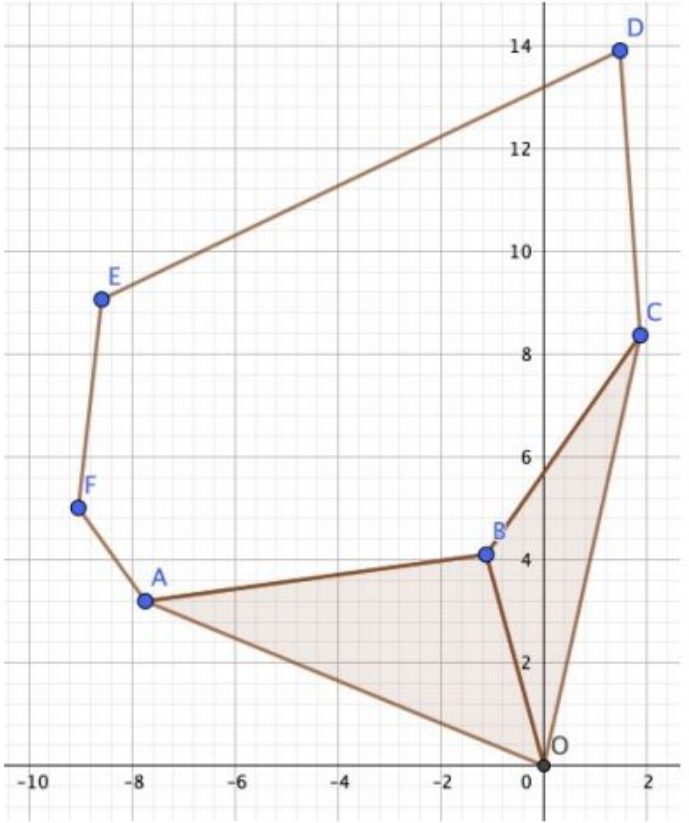


Polygon

- For a polygon, we can divide the polygon into N triangles and sum their signed areas together.

$$Area = \frac{1}{2} \sum_{i=0}^{N-1} \overrightarrow{OP_i} \times \overrightarrow{OP_{i+1}}$$

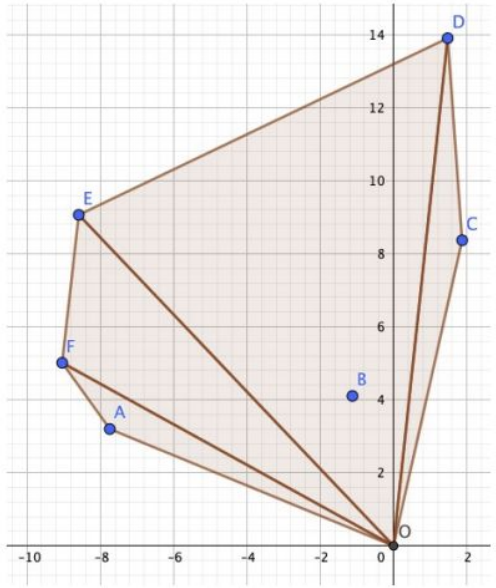
- The triangles with **negative** areas are:

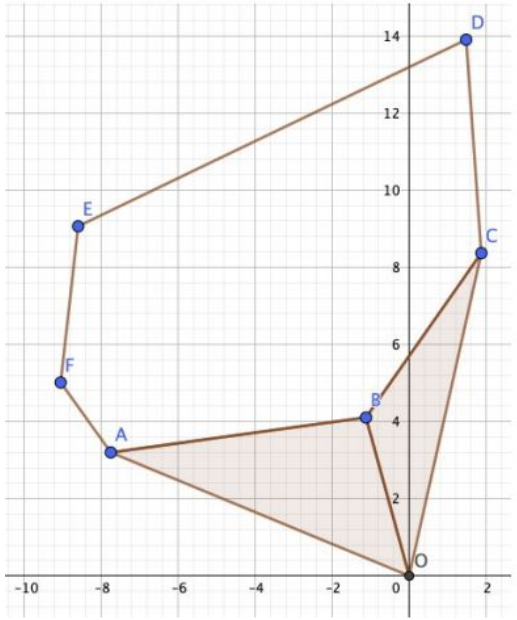


Polygon

- You could see how these triangles sum up to be the whole polygon.
- This works for Concave Polygon as well (shown in the next example).
- But it would not work for non-simple polygon

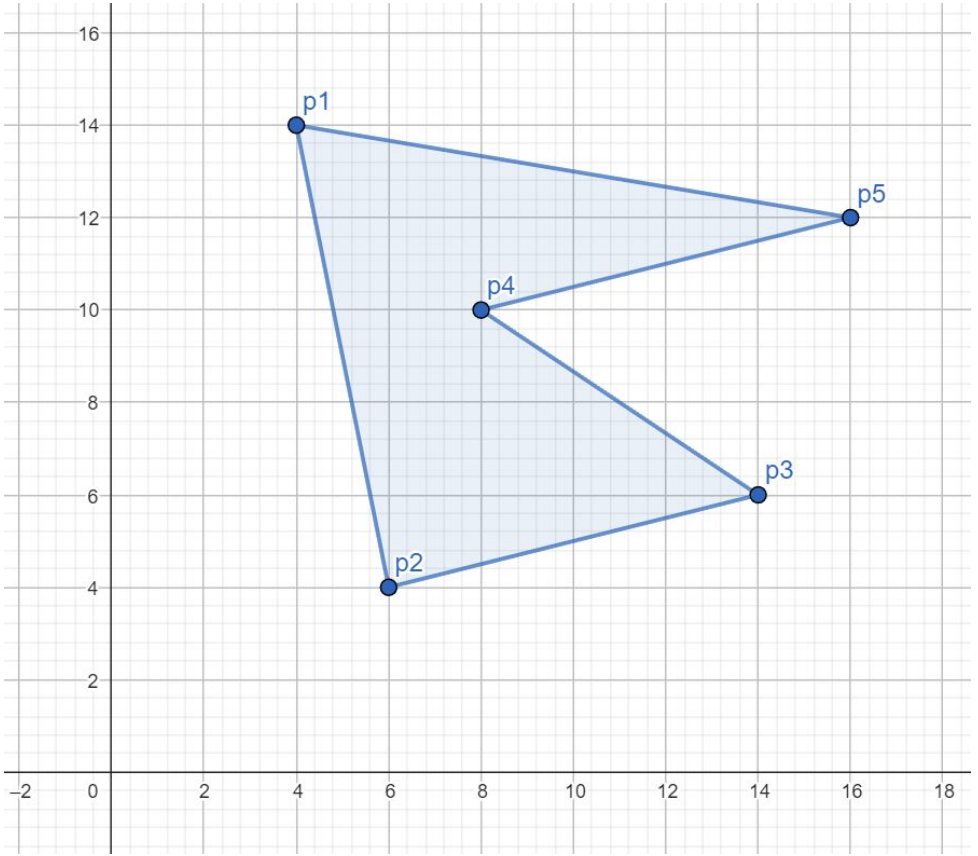
$$Area = \frac{1}{2} \sum_{i=0}^{N-1} \overrightarrow{OP_i} \times \overrightarrow{OP_{i+1}}$$





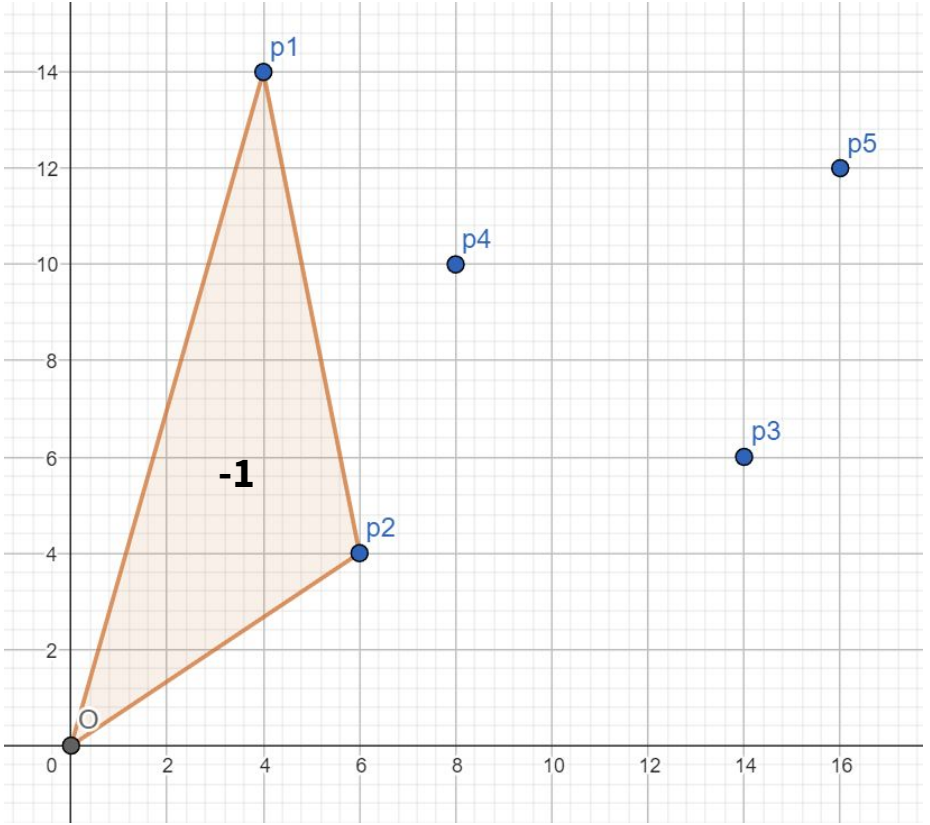
Polygon

- Concave polygon example



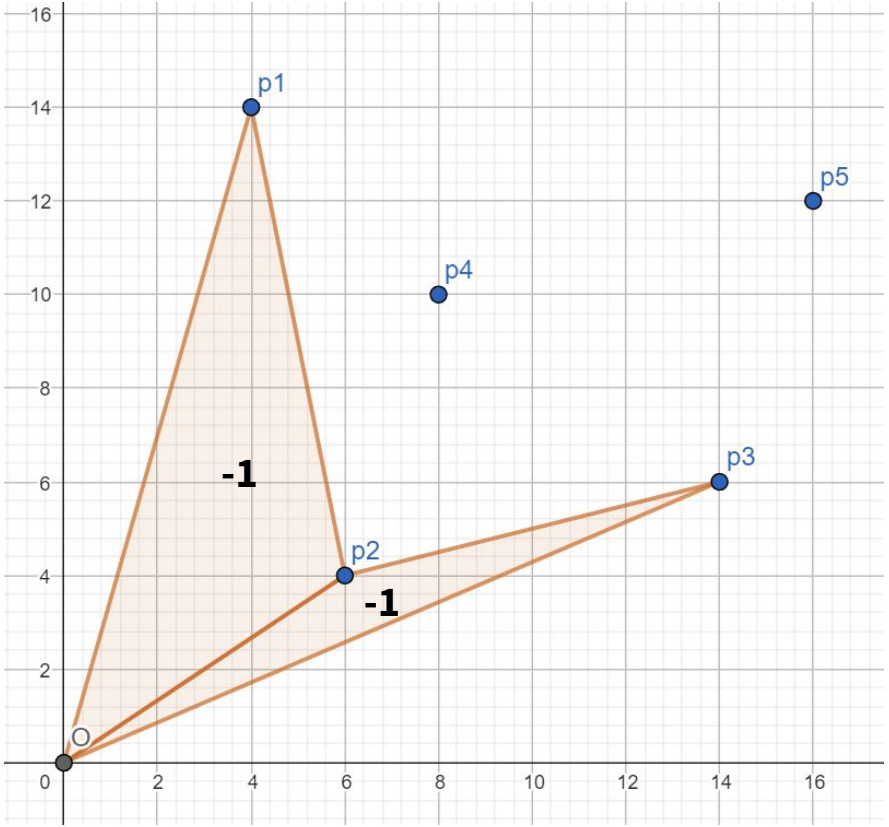
Polygon

- Concave polygon example
- Negative Area **OP1P2**



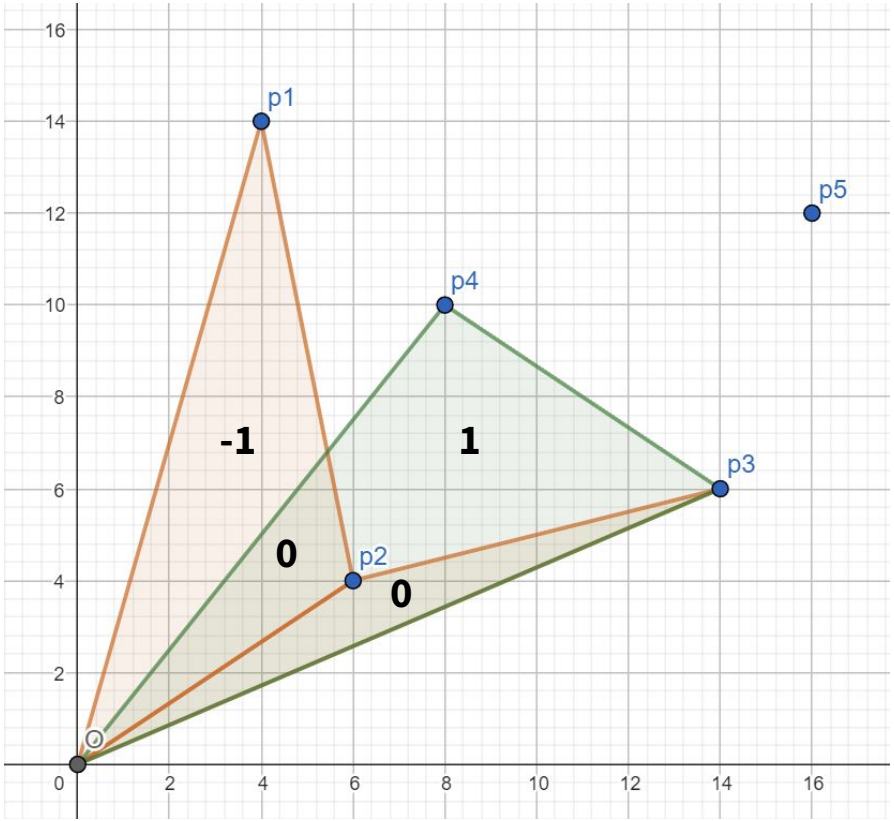
Polygon

- Concave polygon example
- Negative Area **OP2P3**



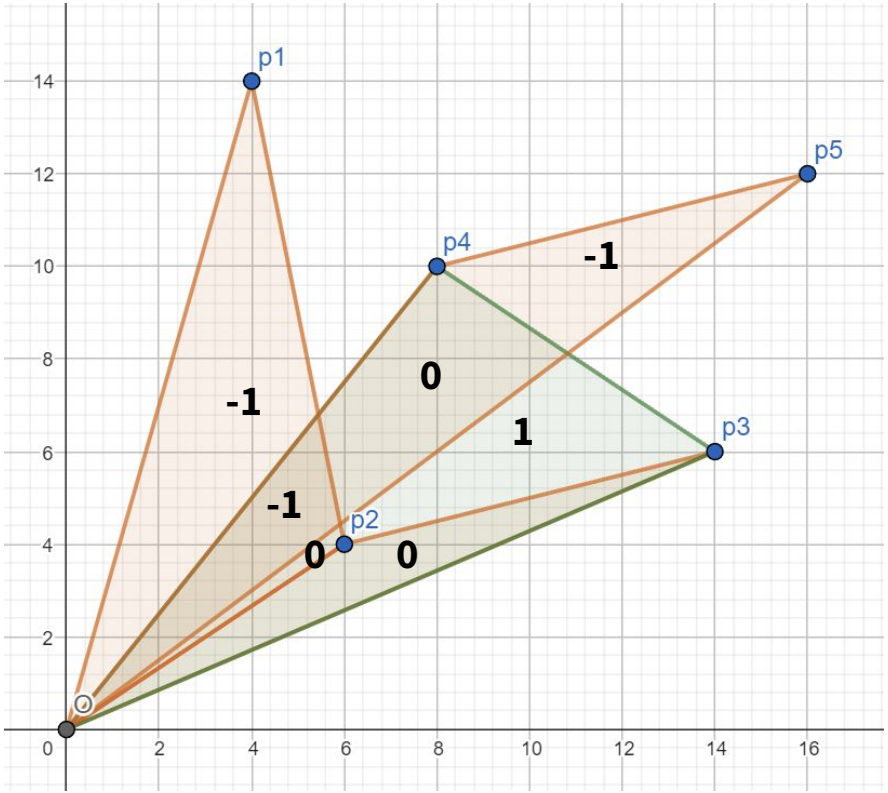
Polygon

- Concave polygon example
- Positive Area **OP3P4**



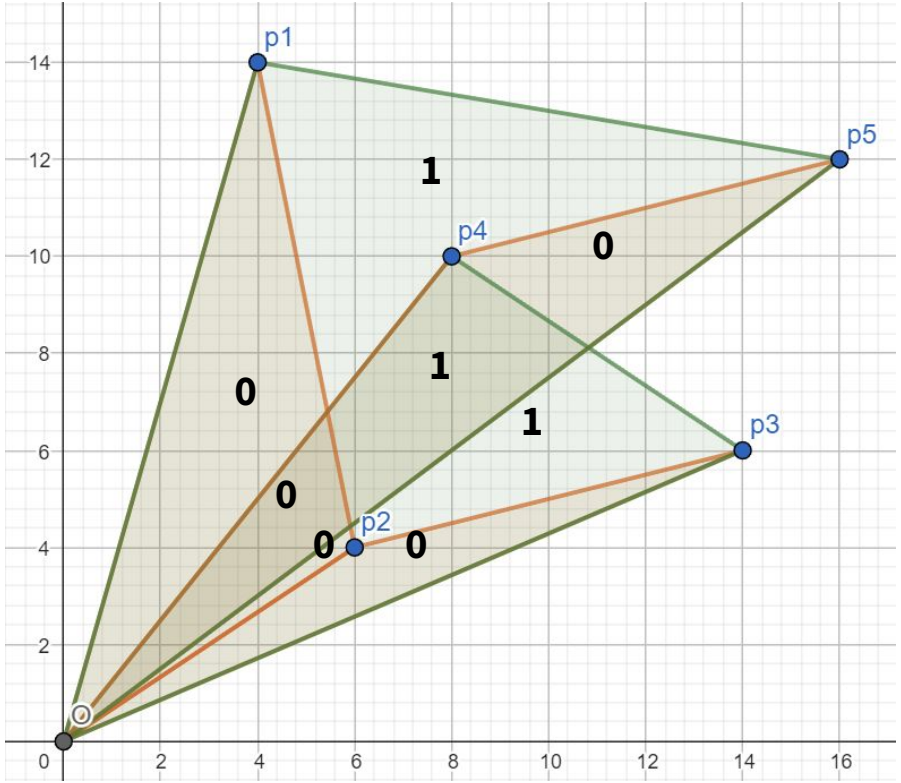
Polygon

- Concave polygon example
- Negative Area **OP4P5**



Polygon

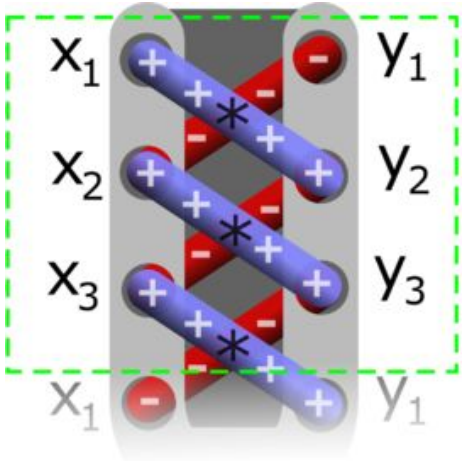
- Concave polygon example
- Positive Area **OP5P1**



Polygon

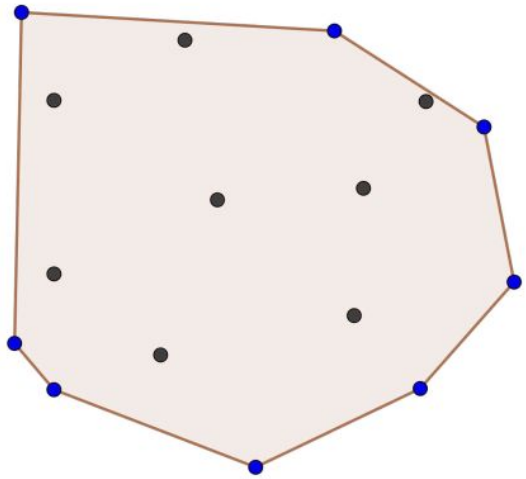
- This area calculation method is also called the shoelace formula.
- Because the product terms make it looks like tying a shoelace.

$$Area = \frac{1}{2} \sum_{i=0}^{N-1} \overrightarrow{OP_i} \times \overrightarrow{OP_{i+1}}$$



Convex Hull

- Given a bunch of points, a **convex hull** is the smallest convex polygon that contains all of them.
- We can always choose a subset of points to form the hull (why?)
- How to find these points in a quick enough way?



Convex Hull

- There are many solutions to how to find a convex hull.
- The one we are going to teach, is an O(n log n) algorithm called **Andrew's Monotone Chain Algorithm**, based on Graham's scan.

- There exists O(n log h) algorithm (h = number of points on the hull).

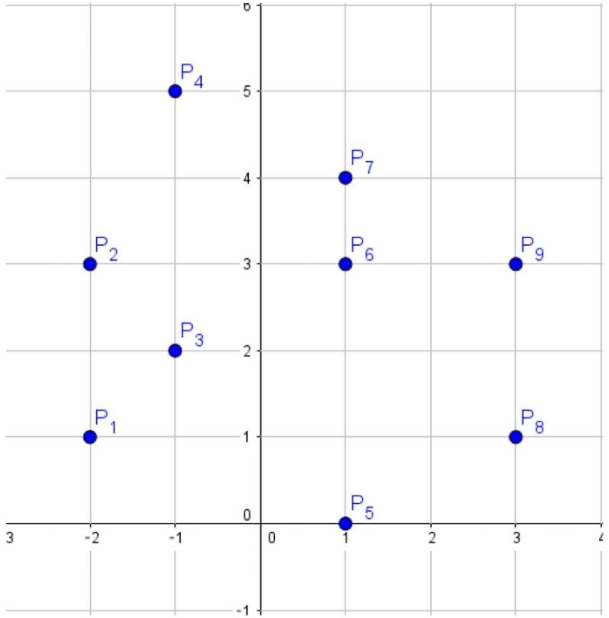
Convex Hull

- There are many solutions to how to find a convex hull.
- The one we are going to teach, is an O(n log n) algorithm called **Andrew's Monotone Chain Algorithm**, based on Graham's scan.

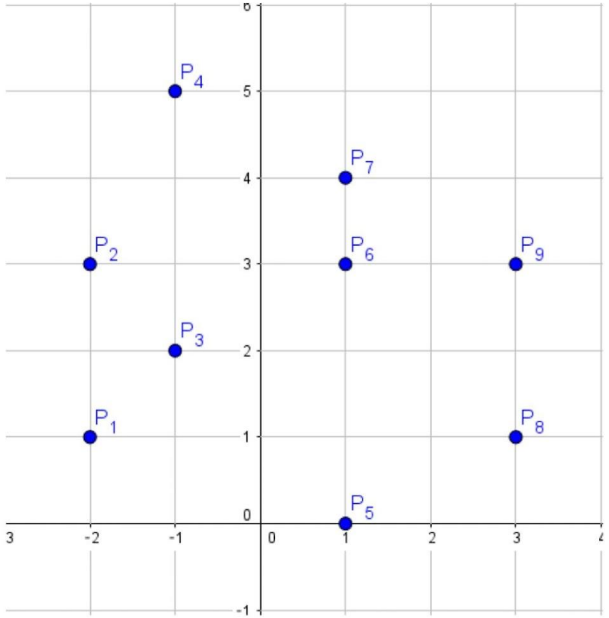
- There exists O(n log h) algorithm (h = number of points on the hull).

- The following algorithm demonstration is from Alex Tung's 2017 HKOI Lecture.

- Step 1: sort points by x then by y



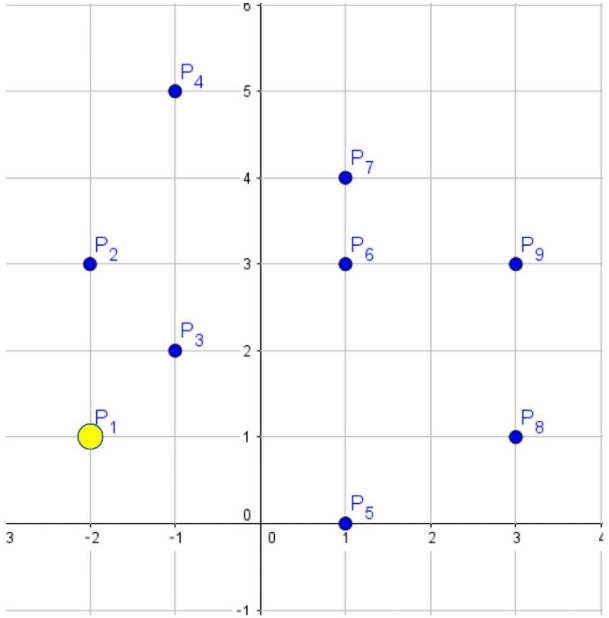
- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 1$

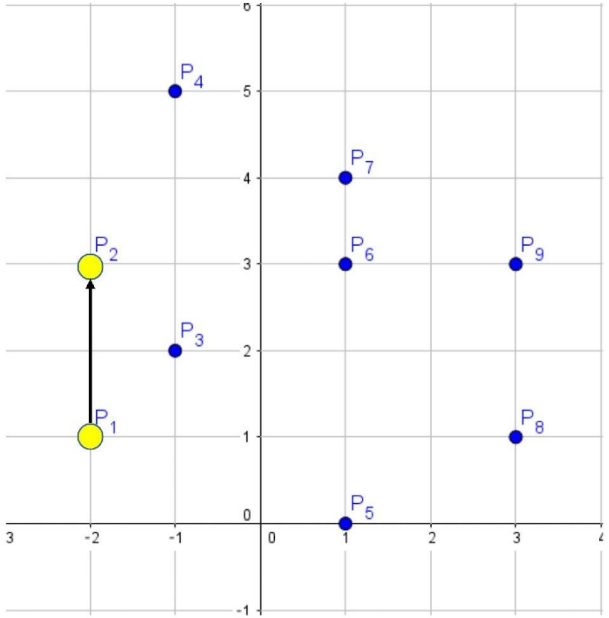
$STACK = [P_1]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 2$

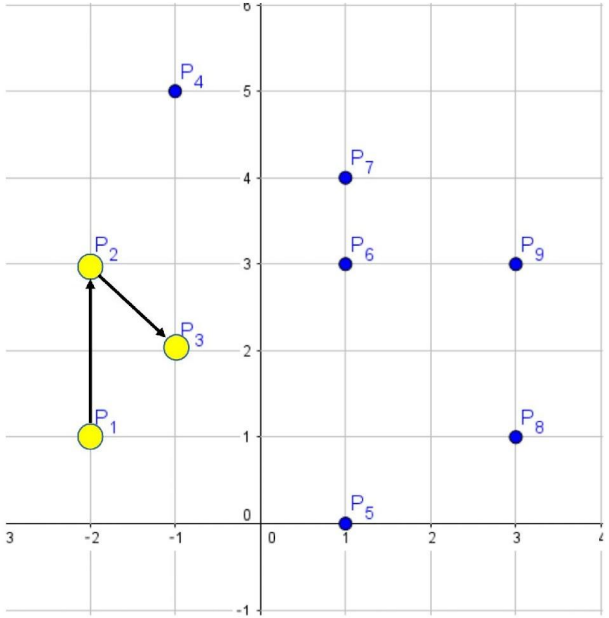
$STACK = [P_1, P_2]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 3$

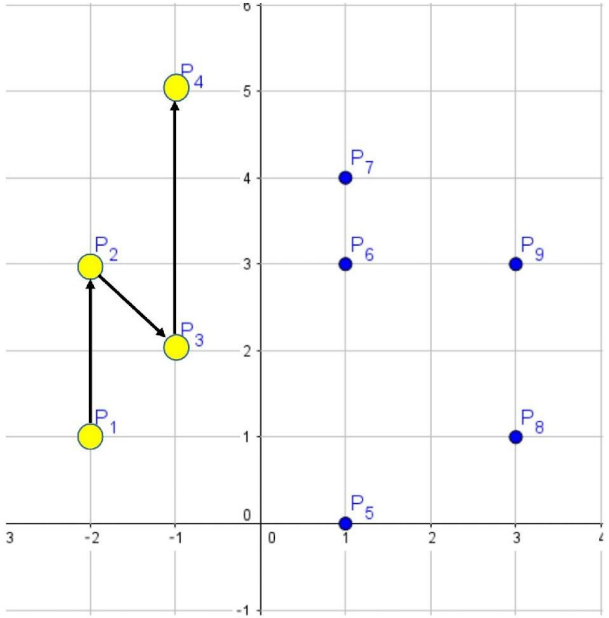
$\text{STACK} = [P_1, P_2, P_3]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 4$

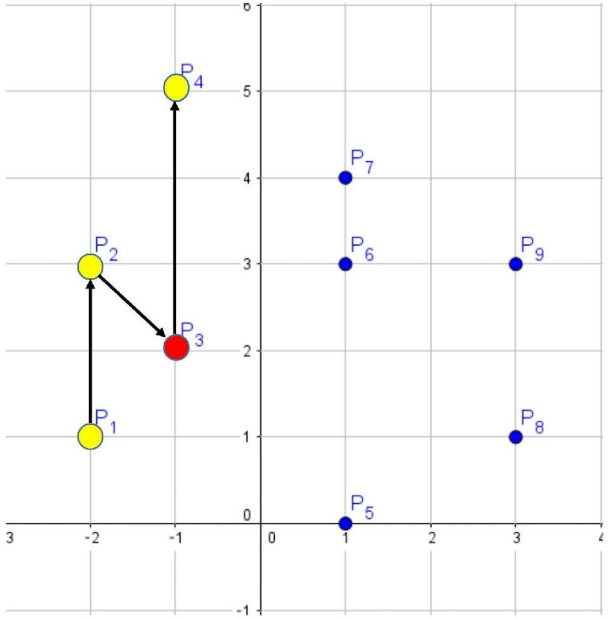
$STACK = [P_1, P_2, P_3, P_4]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 4$

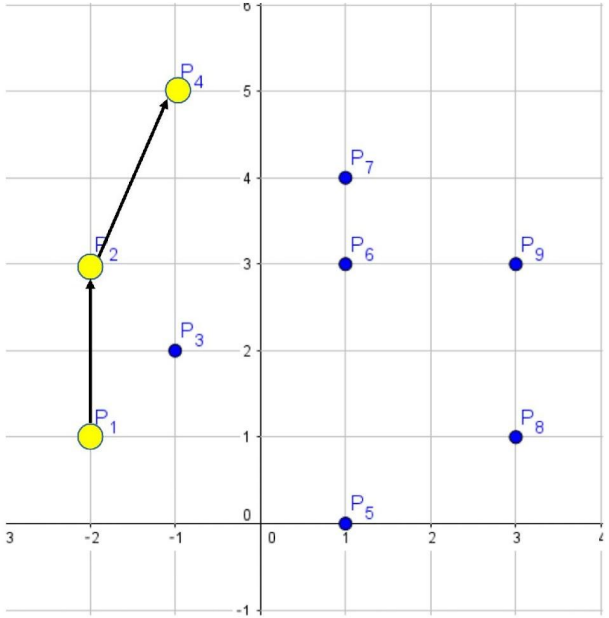
$\text{STACK} = [P_1, P_2, P_3, P_4]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 4$

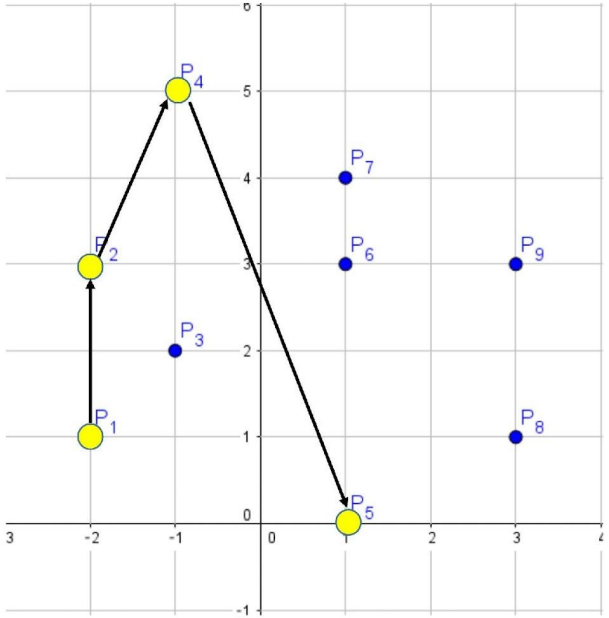
$\text{STACK} = [P_1, P_2, P_4]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 5$

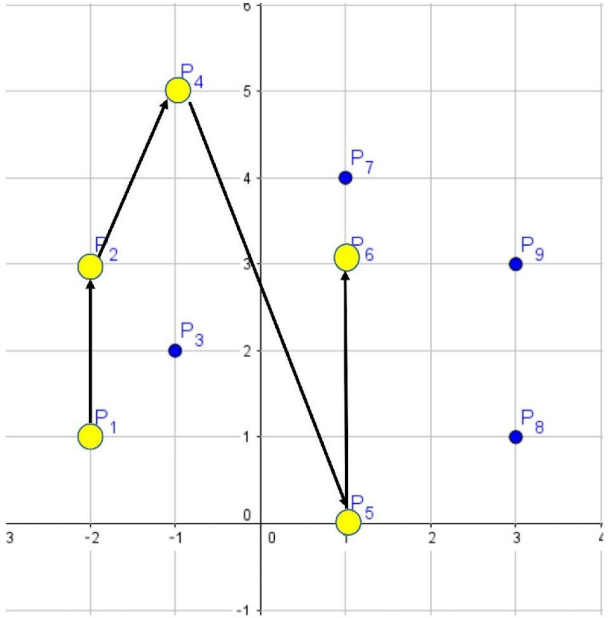
$\text{STACK} = [P_1, P_2, P_4, P_5]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 6$

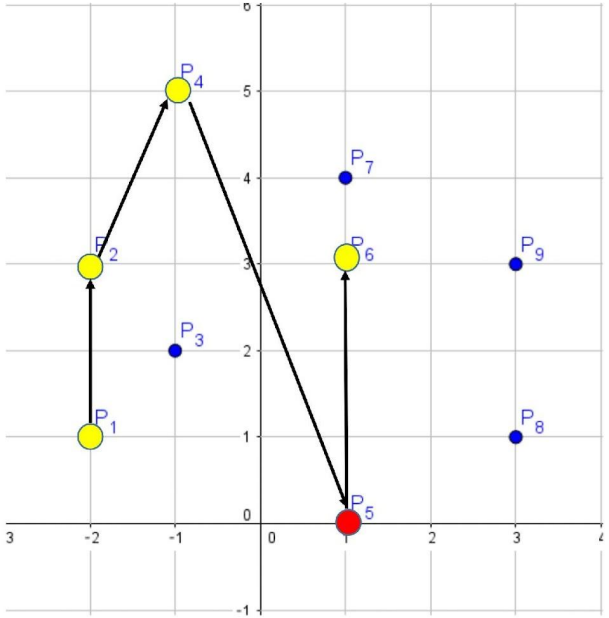
$\text{STACK} = [P_1, P_2, P_4, P_5, P_6]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 6$

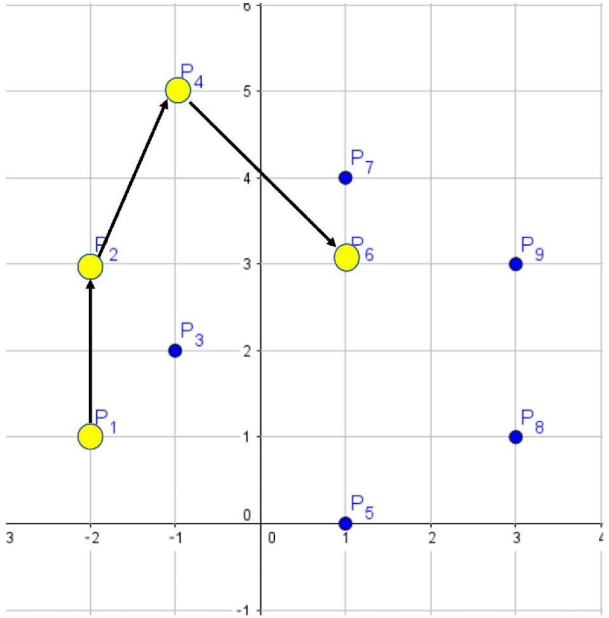
$\text{STACK} = [P_1, P_2, P_4, P_5, P_6]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 6$

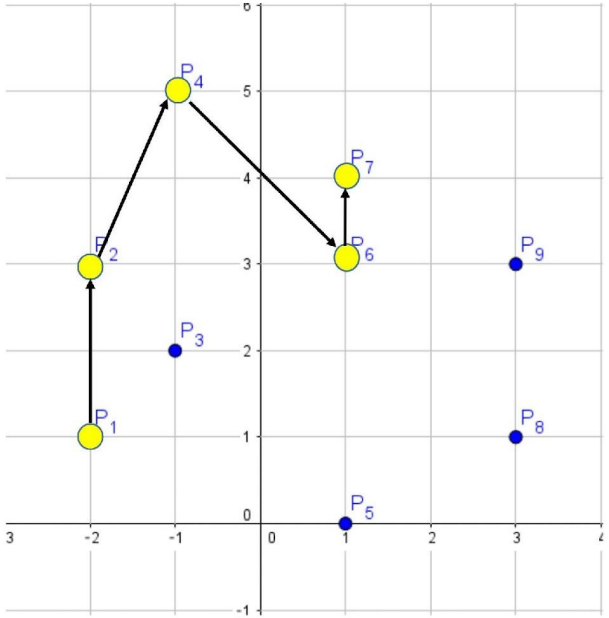
$\text{STACK} = [P_1, P_2, P_4, P_6]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 7$

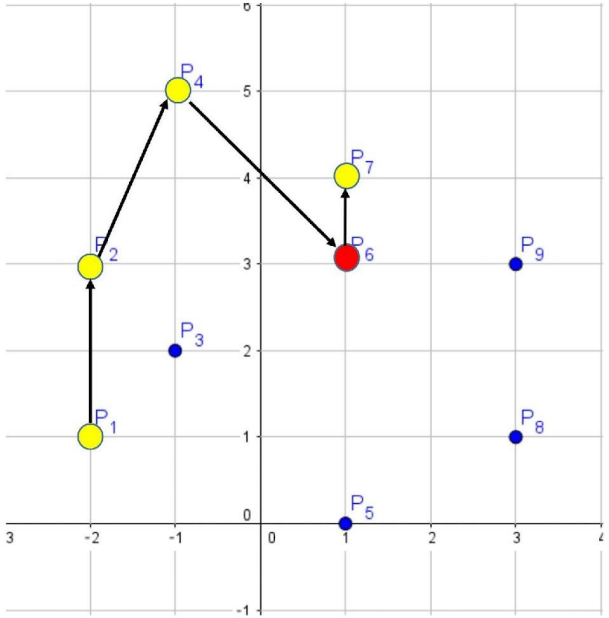
$\text{STACK} = [P_1, P_2, P_4, P_6, P_7]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

i = 7

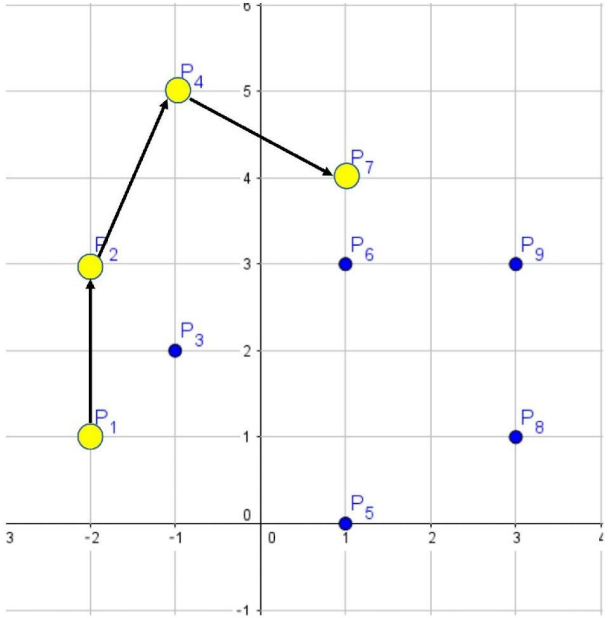
STACK = $[P_1, P_2, P_4, P_6, P_7]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 7$

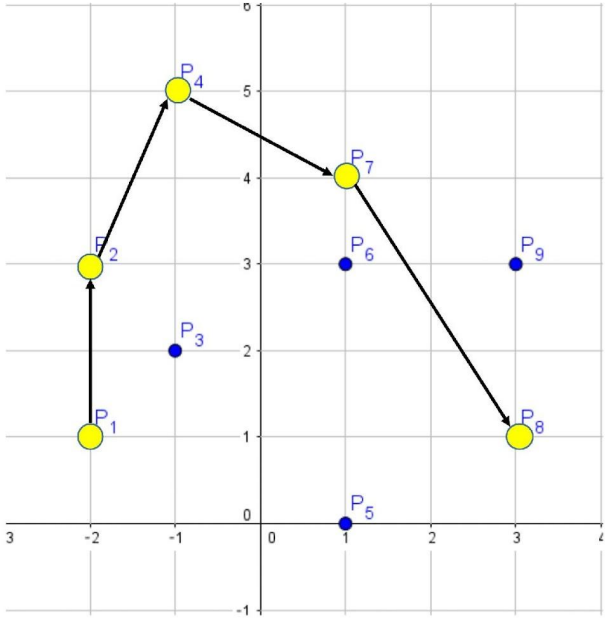
$\text{STACK} = [P_1, P_2, P_4, P_7]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 8$

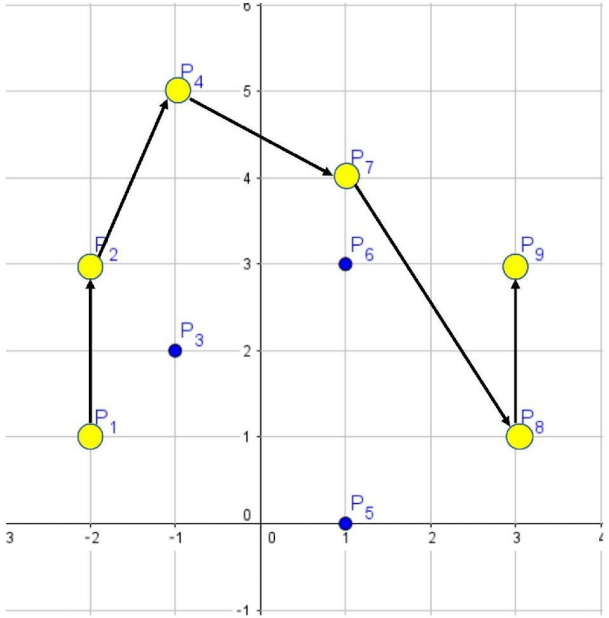
$\text{STACK} = [P_1, P_2, P_4, P_7, P_8]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 9$

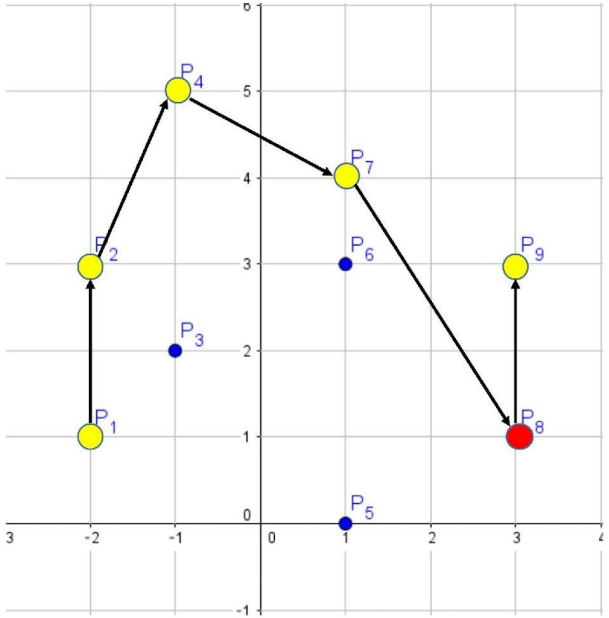
$\text{STACK} = [P_1, P_2, P_4, P_7, P_8, P_9]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 9$

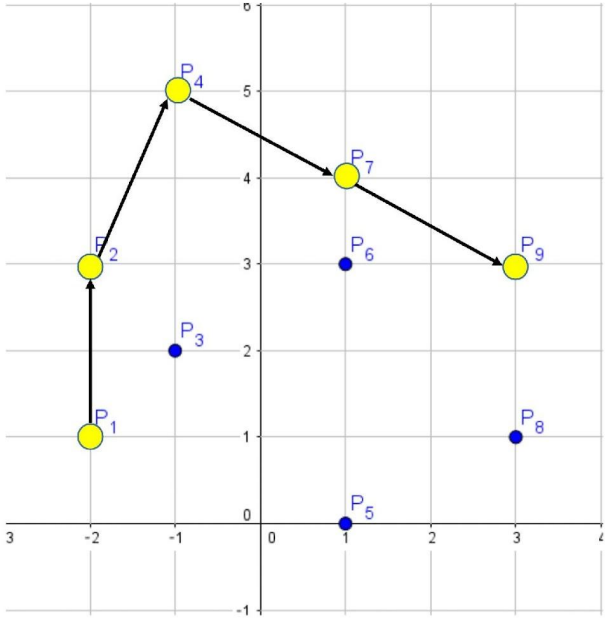
$\text{STACK} = [P_1, P_2, P_4, P_7, P_8, P_9]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 9$

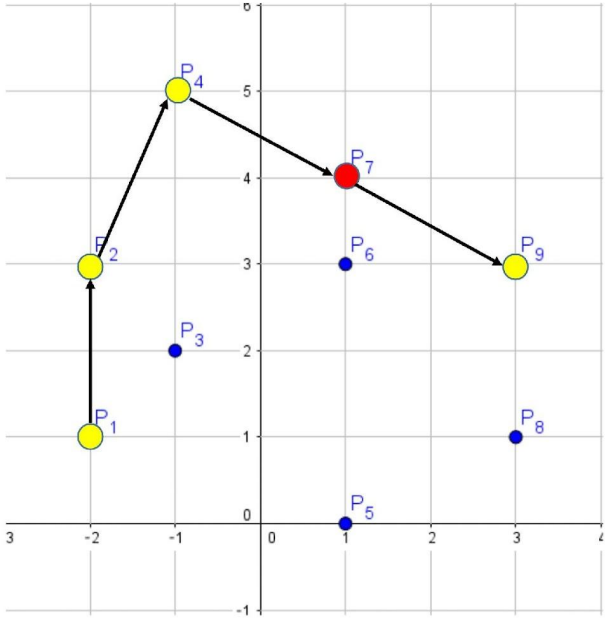
$\text{STACK} = [P_1, P_2, P_4, P_7, P_9]$



- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 9$

$\text{STACK} = [P_1, P_2, P_4, P_7, P_9]$

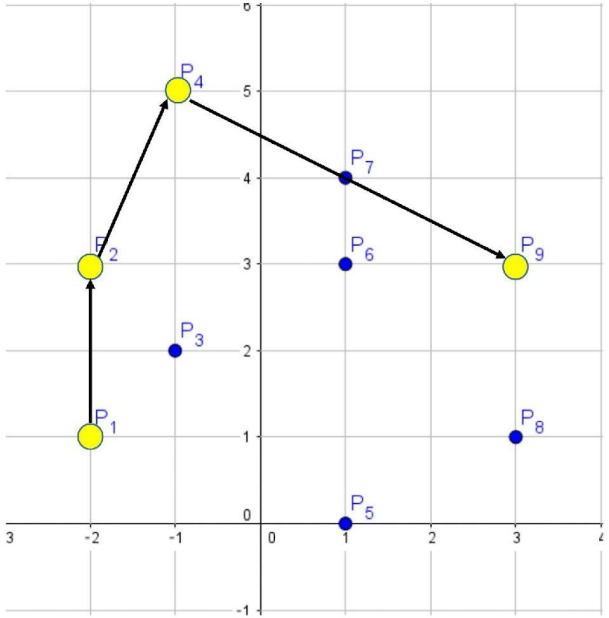


- Step 2: Start with an empty stack. Maintain the stack s.t. every turn is a right turn

$i = 9$

$\text{STACK} = [P_1, P_2, P_4, P_9]$

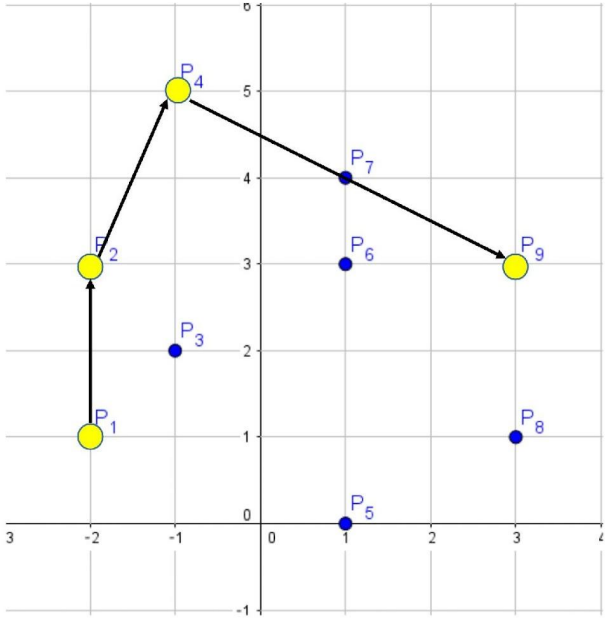
(Step 2 gives us the "upper hull" of the points)



- Step 3: Similar to step 2, but process points in reverse order

$i = 9$

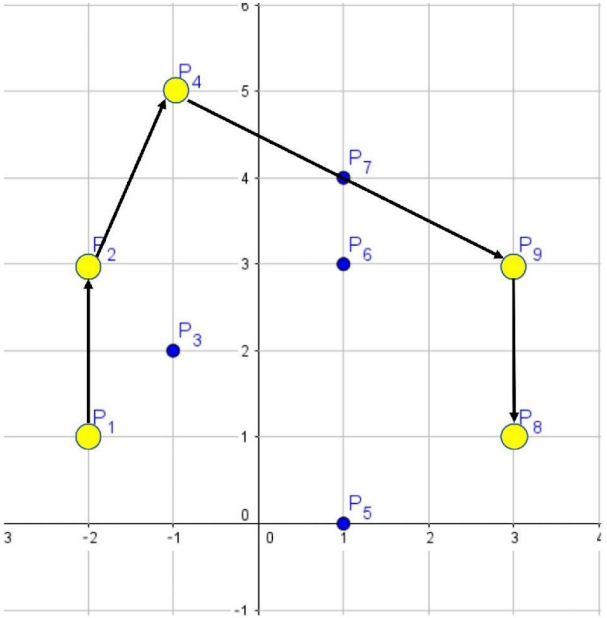
$\text{STACK} = [P_1, P_2, P_4, P_9]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 8$

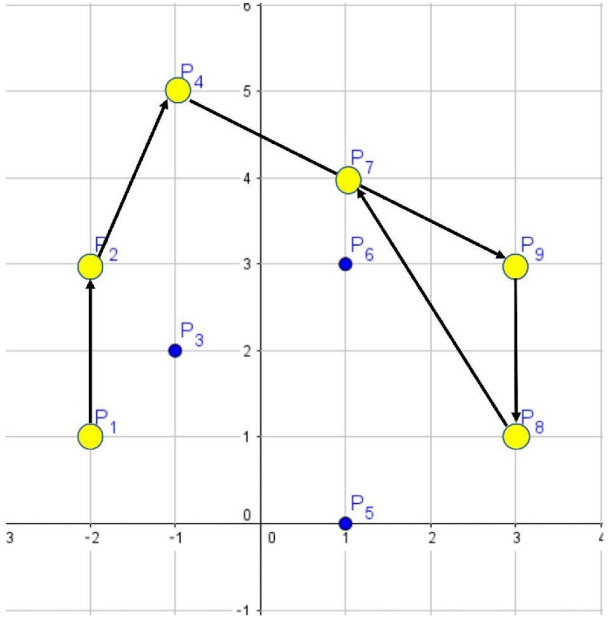
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 7$

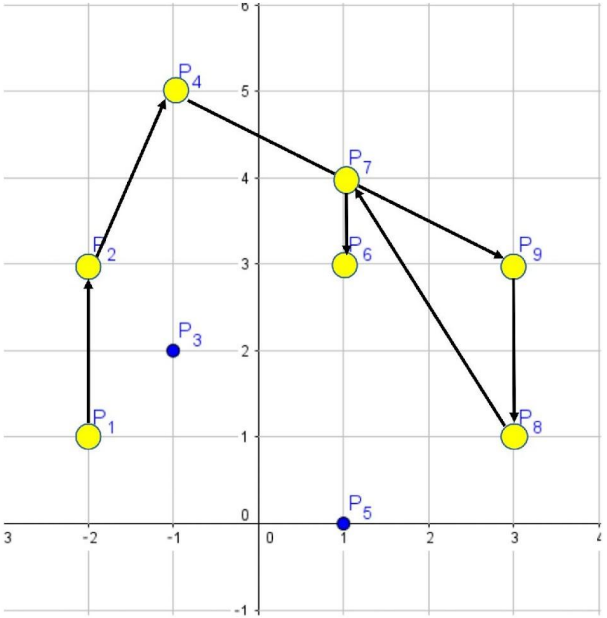
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_7]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 6$

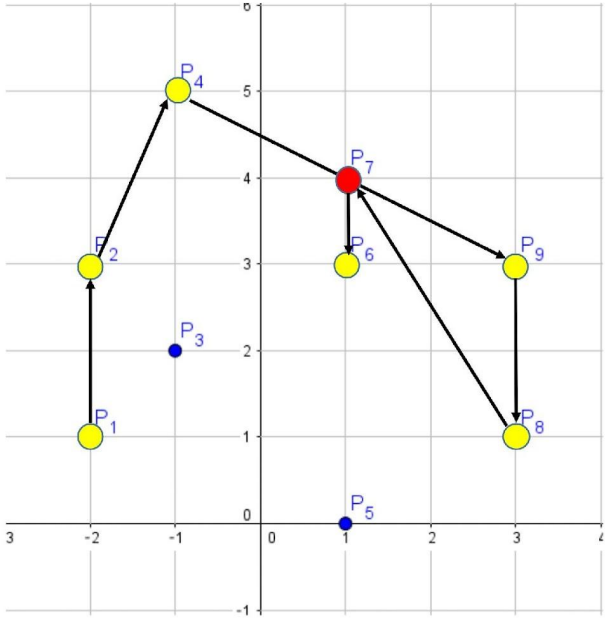
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_7, P_6]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 6$

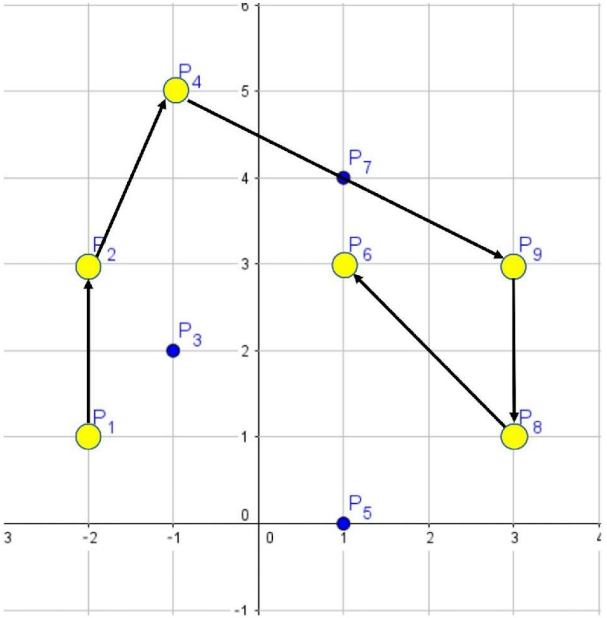
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_7, P_6]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 6$

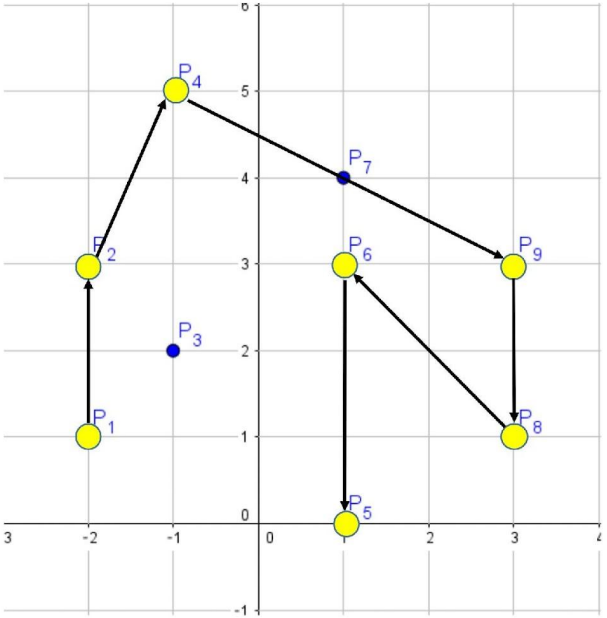
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_6]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 5$

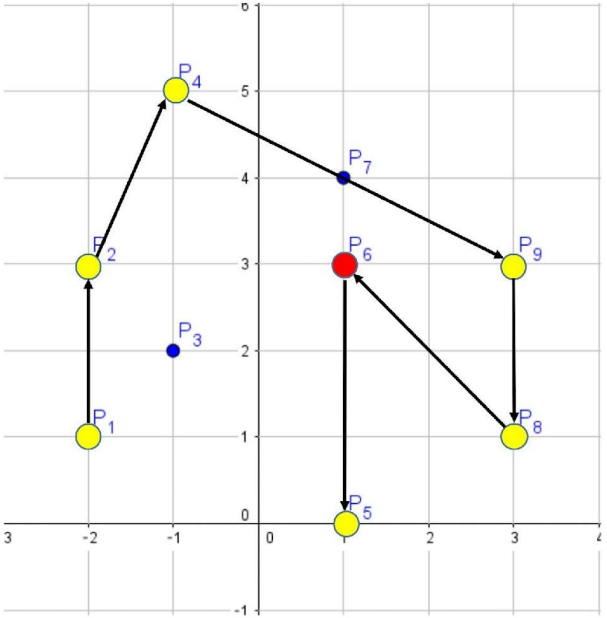
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_6, P_5]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 5$

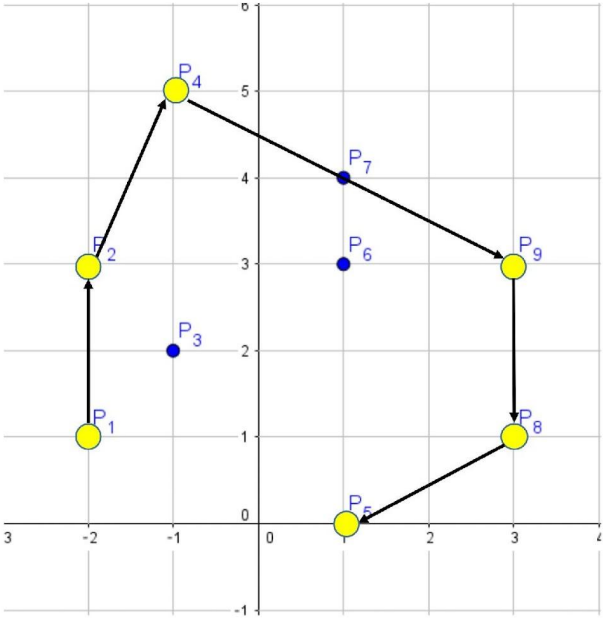
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_6, P_5]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 5$

$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5]$

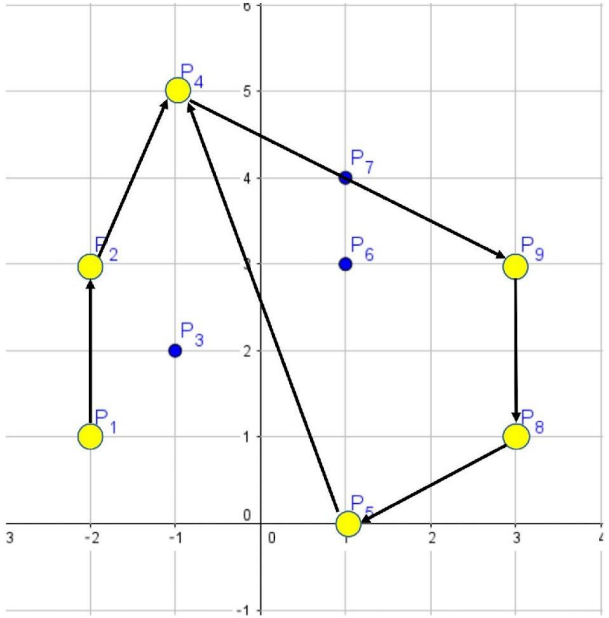


- Step 3: Similar to step 2, but process points in reverse order

i = 4

STACK = $[P_1, P_2, P_4, P_9, P_8, P_5, P_4]$

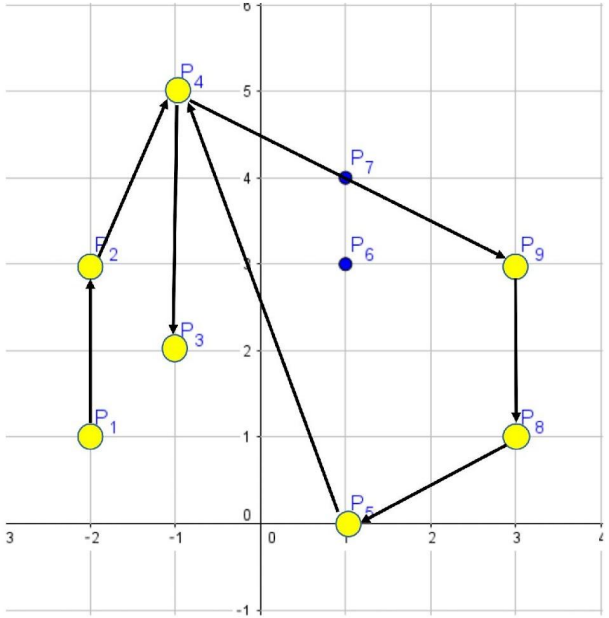
Don't worry about P_4; it will be fixed automatically :)



- Step 3: Similar to step 2, but process points in reverse order

$i = 3$

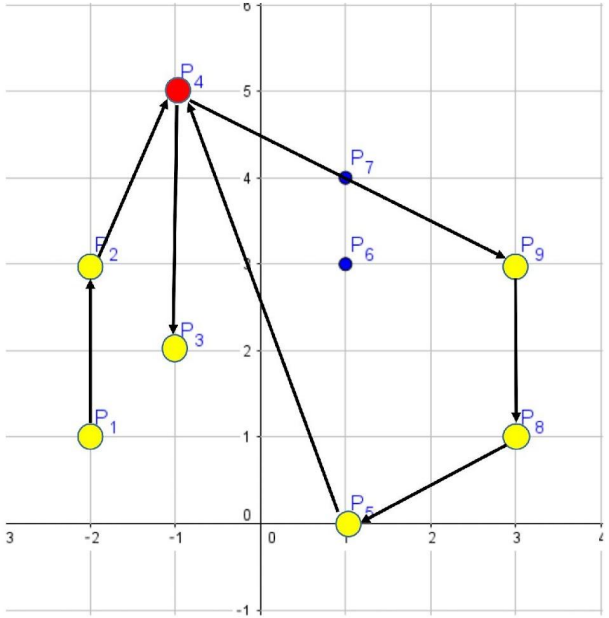
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5, P_4, P_3]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 3$

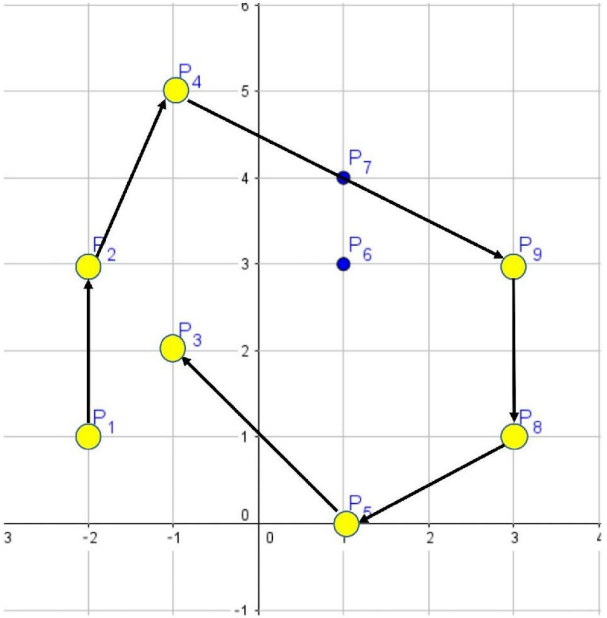
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5, P_4, P_3]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 3$

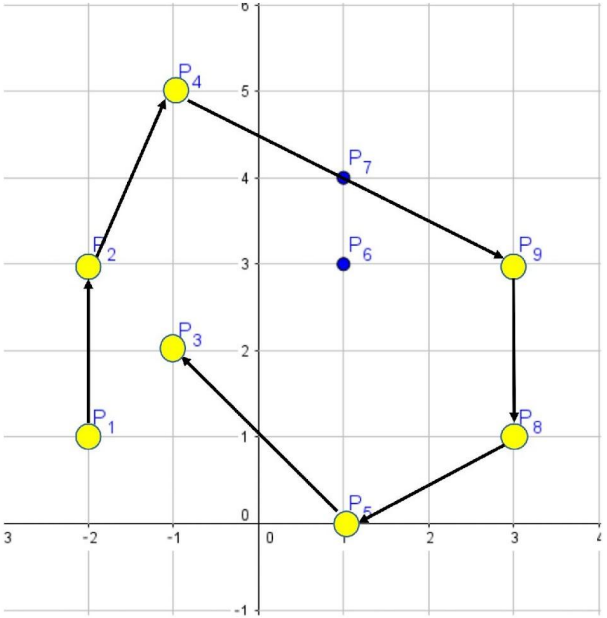
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5, P_3]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 3$

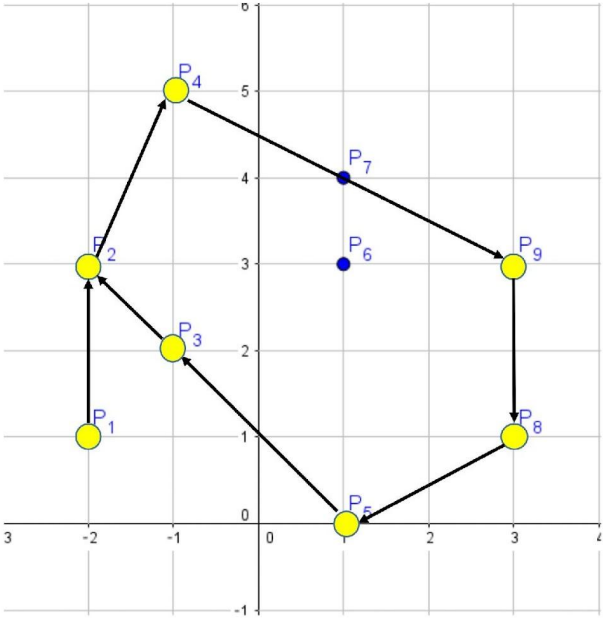
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5, P_3]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 2$

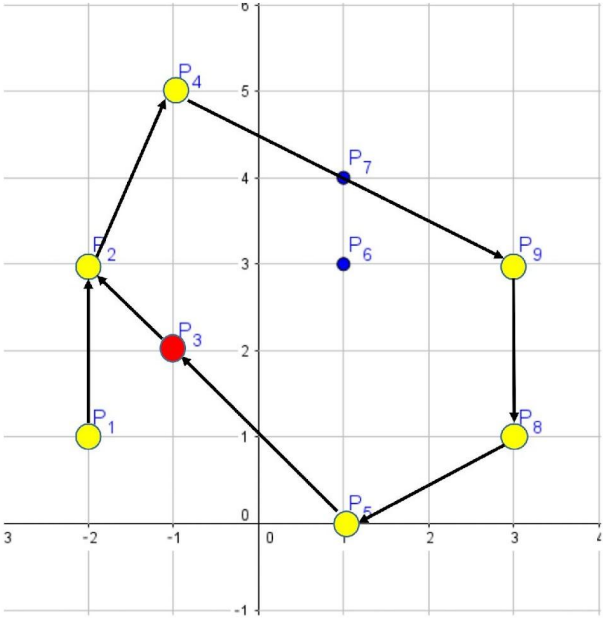
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5, P_3, P_2]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 2$

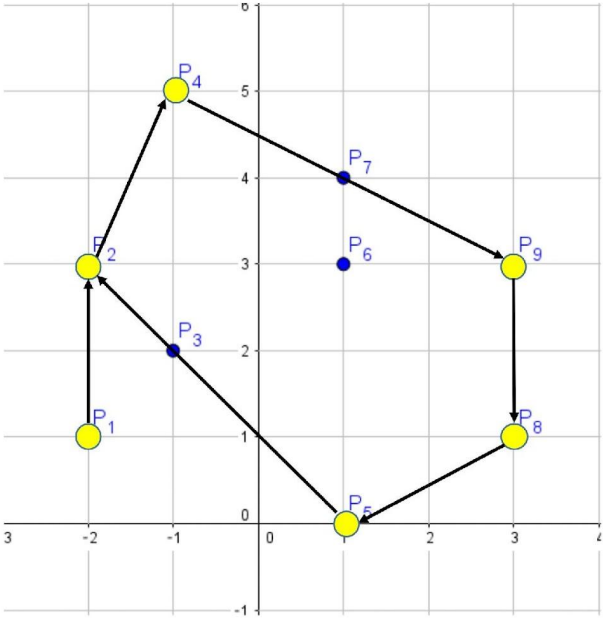
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5, P_3, P_2]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 2$

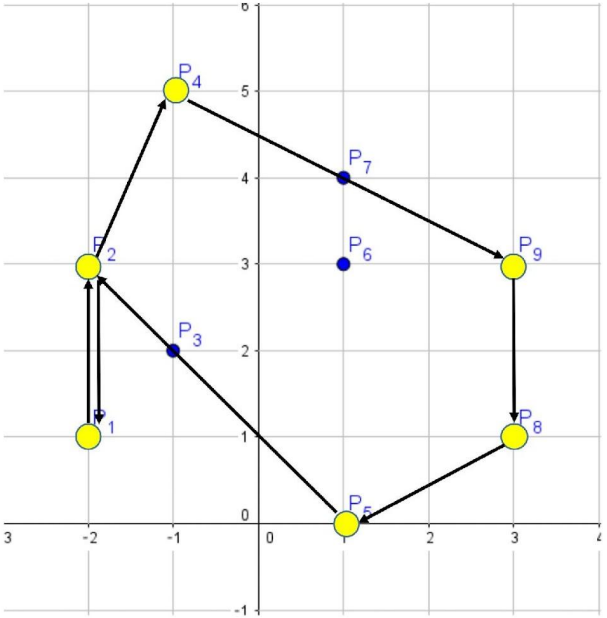
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5, P_2]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 1$

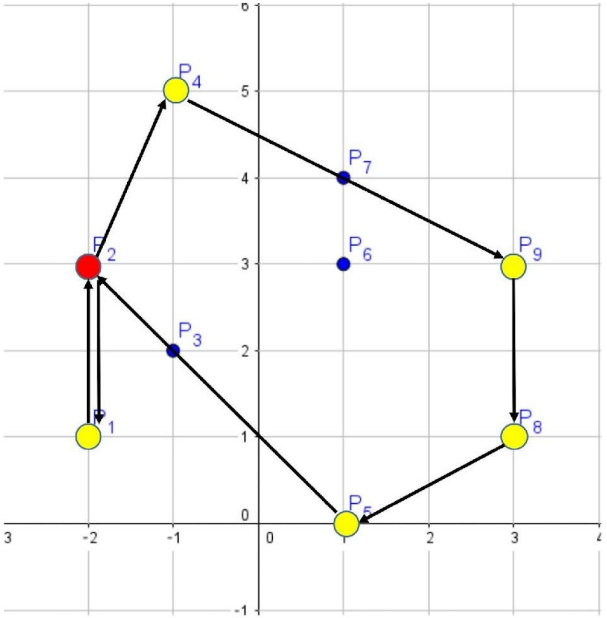
$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5, P_2, P_1]$



- Step 3: Similar to step 2, but process points in reverse order

$i = 1$

$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5, P_2, P_1]$

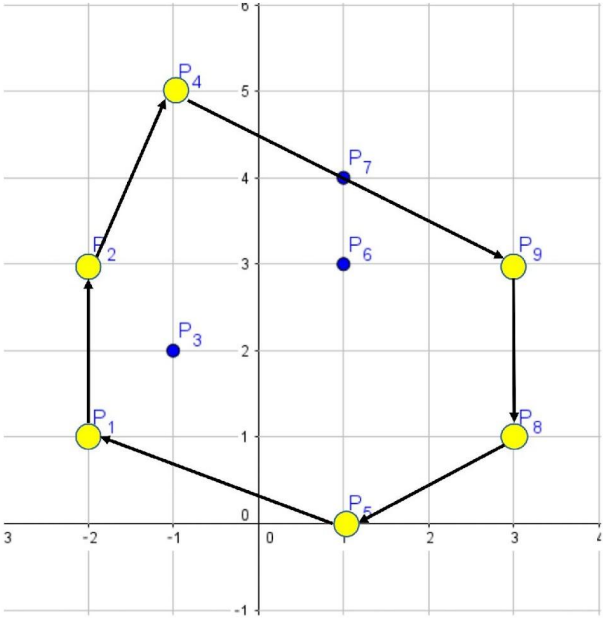


- Step 3: Similar to step 2, but process points in reverse order

$i = 1$

$\text{STACK} = [P_1, P_2, P_4, P_9, P_8, P_5, P_1]$

End!



ICPC 1999 WF Problem D The Fortified Forest

- POJ1873 or Vjudge (UVA811)
- Problem: Given N trees (2 <= N <= 15), each tree are positioned at (Xi, Yi) with value Vi, and cutting it down would give length = Li fences.
- What is the minimum value of trees to be cut down such that you can build fences around the remaining trees?

ICPC 1999 WF Problem D The Fortified Forest

- POJ1873 or Vjudge (UVA811)
- Problem: Given N trees (2 <= N <= 15), each tree are positioned at (Xi, Yi) with value Vi, and cutting it down would give length = Li fences.
- What is the minimum value of trees to be cut down such that you can build fences around the remaining trees?

- Solution: Since N is small, we could just exhaust which tree to cut with dfs. Then we can calculate the perimeter needed to contain the remaining tree by running a simple convex hull algorithm.

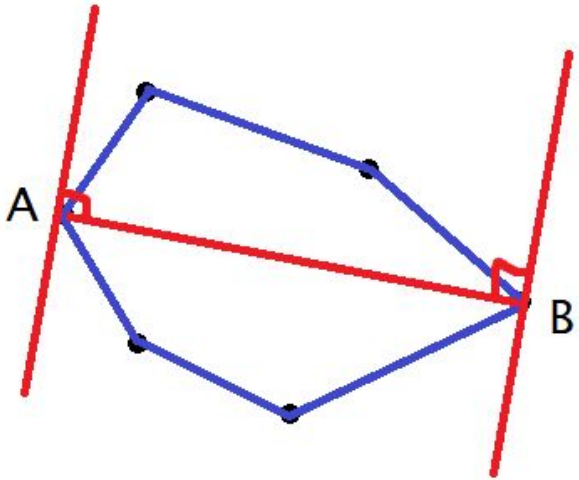
Largest Distance between Points

- Problem: Given N points on a 2D plane, what is the **largest** distance between 2 points?

Largest Distance between Points

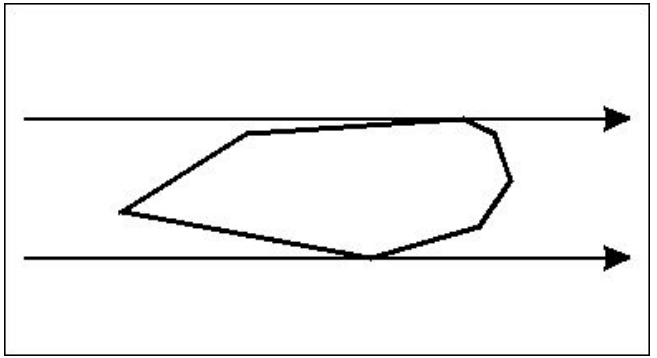
- Problem: Given N points on a 2D plane, what is the **largest** distance between 2 points?

- An observation is, these 2 points must lie on the Convex Hull. Why?
- A property of a point on convex hull is that, you can draw a line through this point, such that every given points are **on one side of this line**.
- Suppose you have a largest distance pair point AB, the perpendicular line to AB through point A and point B are a line that satisfies that property.



Rotating Calipers (旋轉卡尺)

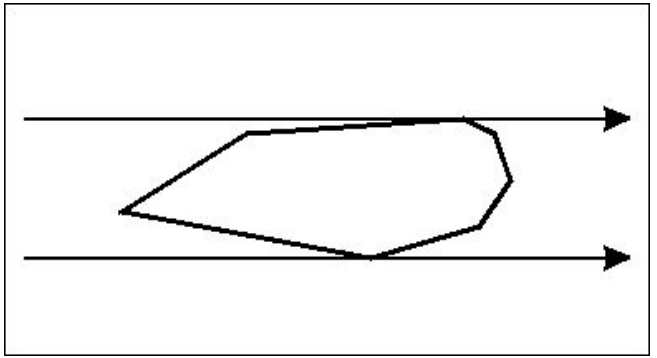
- When 2 parallel line are tangent to 2 points without crossing through any other line of the polygon, the pair of points are **antipodal**. (at most 3N/2 pairs)
- So, a way to find the largest distance is by rotating two parallel lines to generate all antipodal points, and take the largest distance one.



GIF, view here:
http://www-cgrl.cs.mcgill.ca/~godfried/research/calipers.html

Rotating Calipers (旋轉卡尺)

- The antipodal pairs can be generated by maintaining a 2-pointers and update by some area calculation.
- If you are interested in knowing the implementation, you can refer to the wiki page here.



GIF, view here: http://www-cgrl.cs.mcgill.ca/~godfried/research/calipers.html

Rotating Calipers (旋轉卡尺)

- There is a huge list of problems that this technique could solve:

Applications [edit]

Pirzadeh[5] describes various applications of rotating calipers method.

Distances [edit]

- Diameter (maximum width) of a convex polygon[6][7]
- Width (minimum width) of a convex polygon[8]
- Maximum distance between two convex polygons[9][10]
- Minimum distance between two convex polygons[11][12]
- Widest empty (or separating) strip between two convex polygons (a simplified low-dimensional variant of a problem arising in support vector machine based machine learning)
- Grenander distance between two convex polygons[13]
- Optimal strip separation (used in medical imaging and solid modeling)[14]

Bounding boxes [edit]

- Minimum area oriented bounding box
- Minimum perimeter oriented bounding box

Triangulations [edit]

- Onion triangulations
- Spiral triangulations
- Quadrangulation
- Nice triangulation
- Art gallery problem
- Wedge placement optimization problem[15]

Multi-polygon operations [edit]

- Union of two convex polygons
- Common tangents to two convex polygons
- Intersection of two convex polygons[16]
- Critical support lines of two convex polygons
- Vector sums (or Minkowski sum) of two convex polygons[17]
- Convex hull of two convex polygons

Traversals [edit]

- Shortest transversals[18][19]
- Thinnest-strip transversals[20]

Others [edit]

- Non parametric decision rules for machine learned classification[21]
- Aperture angle optimizations for visibility problems in computer vision[22]
- Finding longest cells in millions of biological cells[23]
- Comparing precision of two people at firing range
- Classify sections of brain from scan images

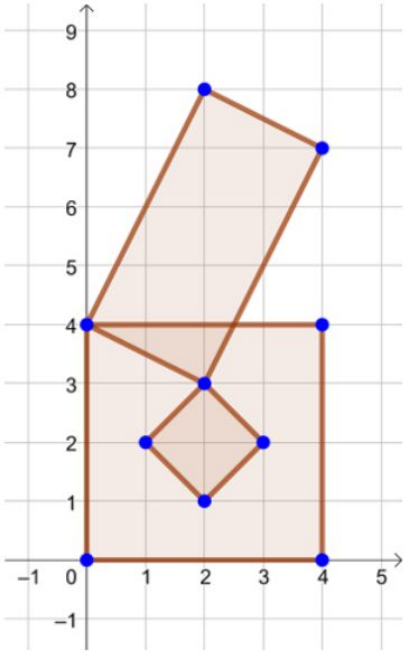
Common Tricks

- Although knowing different algorithms to handle Geometry problem is important, the core of many geometry problems depends Exhaustion (What to exhaust / How to exhaust).

- Also, Divide and conquer is useful in various cases, it can help you speed up your exhaustion process.

M2104 Social Distancing and Lockdown Countdown

- https://judge.hkoi.org/task/M2104
- Problem: Given N points on a 2D plane, find the number of different rectangles formed by these points. Two rectangles are considered different if at least one of four corners are different.
- 1 <= N <= 2500, -1e9 <= x, y <= 1e9



M2104 Social Distancing and Lockdown Countdown

Adapted From 2021 MC0 Solutions

- For subtask 1, N is pretty small (≤30), exhaust all possible combinations of 4 points in $O(N^4)$
- To check if it forms a rectangle, we need to check the lengths and angles, which you should have known how to do so.

- Remember to remove duplicated counts (consider how many combinations we considered for one rectangle)
- If we exhaust 4 indices (i, j, k, l) and:
  - only consider (i, j, k), (j, k, l), (k, l, i), (l, i, j) form right angles, the duplicate factor is 8 (or 4 if it's only clockwise right angle)

Expected score: 5/20

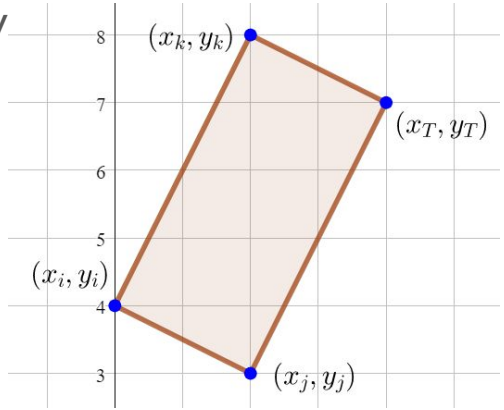
M2104 Social Distancing and Lockdown Countdown

Adapted From 2021 MC0 Solutions

To do better, we could exhaust first 3 points in $O(N^3)$, and check if the remaining points exist

Suppose we have exhausted (i, j, k):

- First we have to confirm the points j, i, k forms a right angle
- If so, the target point coordinates can be calculated by
  - $x_T = x_k - x_i + x_j$ (similar for y)
- And we can check if (x_T, y_T) exists if:
  - we first insert the N given points into a std::set (or similar)
  - Check in O(lgN) (or O(1) if some other data structures)



Expected score: 8/20

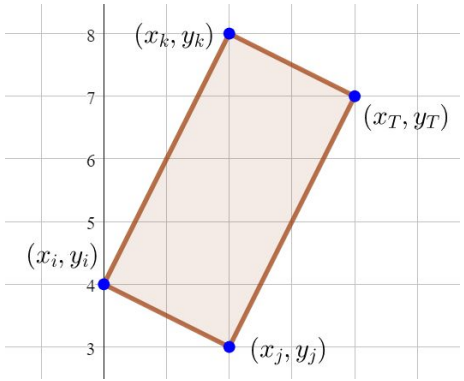
M2104 Social Distancing and Lockdown Countdown

Adapted From 2021 MC0 Solutions

From the previous solution, we know that when considering point i, only pairs (j, k) that form a right angle by point i is meaningful

For every point i exhausted, sort the N-1 points by its angle from point i

- Maintain two pointers to keep angle between points = 90 degrees
- Exhaust all pairs of points (j, k) along these two lines
  - Check if the target point T exist (same as previous slide)



Expected score: 12/20 ~ 20/20

The maximum number of triples that form a right angle is in fact $O(N^2 \lg N)$:
[János Pach, Micha Sharir, Repeated angles in the plane and related problems]

M2104 Social Distancing and Lockdown Countdown

Adapted From 2021 MC0 Solutions

From subtask 2 to 3, we're expecting $O(N^2 \lg N)$ or $O(N^2)$ solutions, the actual score you get highly depends on optimizations of your implementation :)

In the following slides,
we will discuss a different solution that could be implemented easier,
also easier to pass the tight TL.

M2104 Social Distancing and Lockdown Countdown

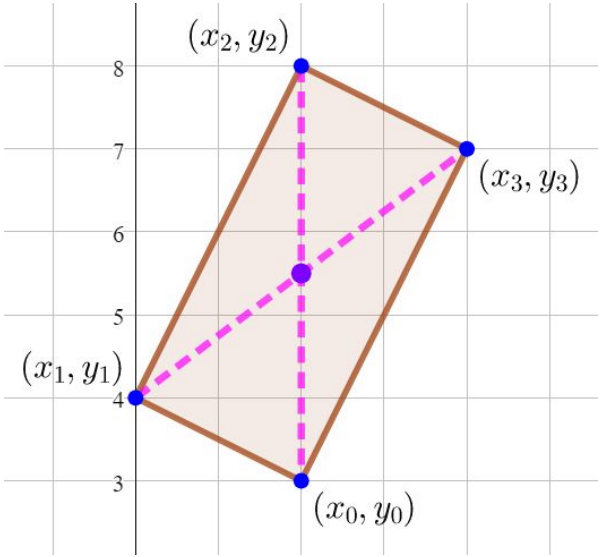
Adapted From 2021 MC0 Solutions

The key observation is that, to form a rectangle, the 4 points must come from 2 pairs of points that:

- Share the same midpoint
- Have the same length

e.g. 1st pair: (x_0, y_0), (x_2, y_2); 2nd pair (x_1, y_1), (x_3, y_3)

(also if the above conditions are satisfied, they must form a rectangle)



M2104 Social Distancing and Lockdown Countdown

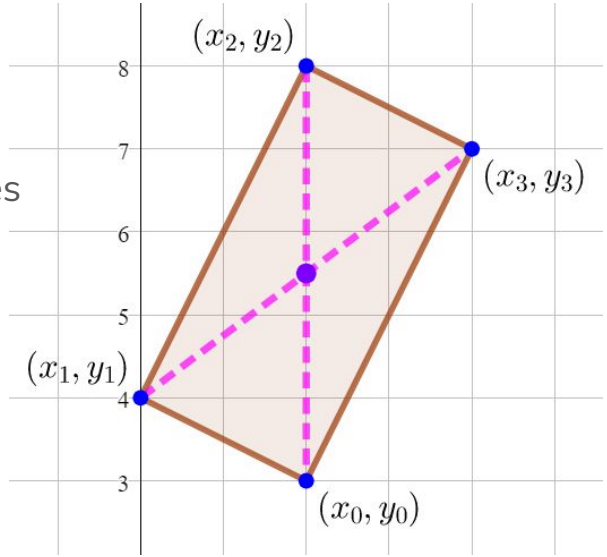
Adapted From 2021 MC0 Solutions

So the idea of full solution is:

- exhaust all pairs of points
- Record their (midpoint, distance) e.g. ((2, 5.5), 5)
- Count the number of such (midpoint, distance)
  - If the count is C, there are C*(C-1)/2 ways to form rectangles

We can record them with std::map / std::multiset / etc., or we can store them in an array and sort to count

Expected score: 14/20 ~ 20/20

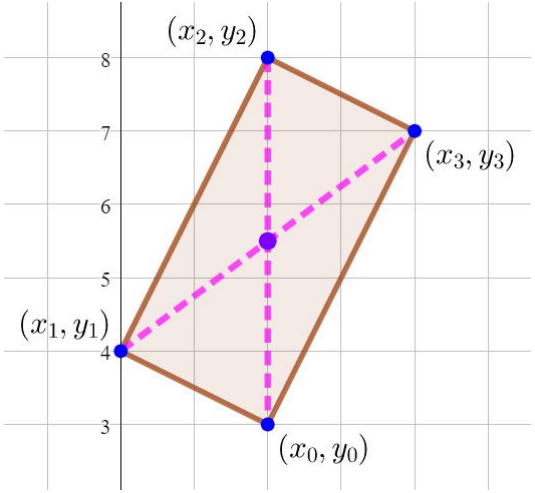


M2104 Social Distancing and Lockdown Countdown

Adapted From 2021 MC0 Solutions

To avoid precision errors, the best practice is to store everything in integers:

- Instead of storing midpoints, store midpoint*2, i.e. sum of two points
- Do not take sqrt of the distance



Exhaustion in Computational Geometry

Throughout the previous solution, what we are exhausting keep changing (from most obvious to less obvious):

- 4 Corners
- 3 Corners
- 3 Corners where the 2nd and 3rd corners form 90-degree angle with 1st
- 2 Corners on a diagonal

- We could see that, by exhausting the right thing with the correct geometric property, we could yield faster solution

Smallest Distance between Points

- Problem: Given N points on a 2D plane, what is the **smallest** distance between 2 points? (also known as Closest Point Pair problem)

Smallest Distance between Points

- Problem: Given N points on a 2D plane, what is the **smallest** distance between 2 points? (also known as Closest Point Pair problem)

- A simple naive solution is to exhaust every point pair and calculate their distance, but of course, we could do better.

Smallest Distance between Points

- The **Scanning Line** approach
- First sort the points by their x-coord (can also then sort by y too but not important)
- Iterate through each point, suppose the current minimum distance is d, we can only consider the second point in the range of [x - d, x + d].

- By sorting the points this already require O(n log n). While it works great with random cases, it would take $O(n^2)$ time if all points are on the same x-coord.

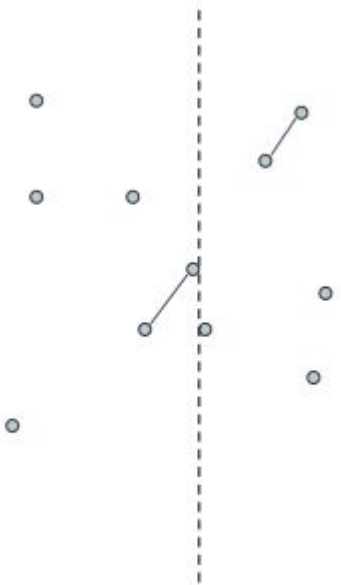
Smallest Distance between Points

Adapted From 2022 AD&C Lecture

- Divide and Conquer approach

- **Divide:** Split the points by the median x-coordinate into **P_l** and **P_r**
- **Conquer:** Solve recursively for both parts

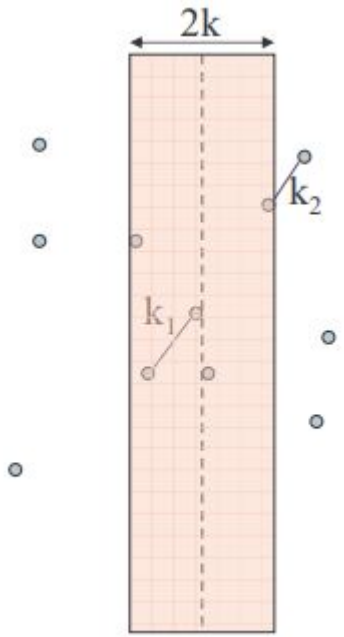
- **Combine:** Find min distance across boundary (Main focus)



Smallest Distance between Points

Adapted From 2022 AD&C Lecture

- Let **k_1** and **k_2** be the min distance in **P_l** and **P_r** respectively and **k = min(k_1, k_2)**
- Note that we are only interested in points with horizontal distance less than or equal to **k** to the median.

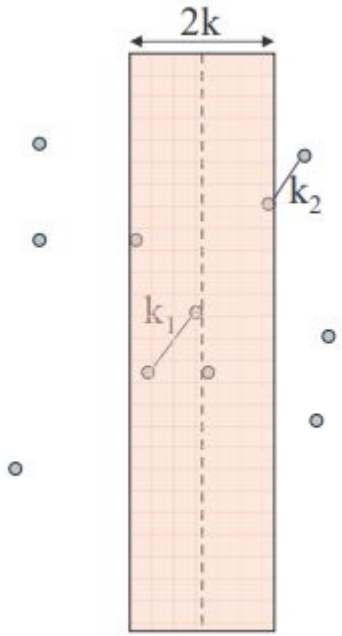


Smallest Distance between Points

Adapted From 2022 AD&C Lecture

- Combining:
- Sort all points in the strip by y-coordinate

```
for i in range(len(strip)):
        for j in range(i, len(strip)):
                if strip[j].y - strip[i].y < k:
                        update_min
                else:
                        break
```

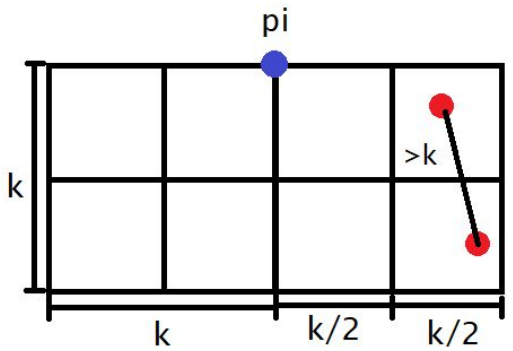


Smallest Distance between Points

Adapted From 2022 AD&C Lecture

- Claim: the inner loop iterates at most 7 times
- Key observations: All points in the same strip must be **> k** away from each other
- Let **p_i** be the point we are considering now, every square in the right diagram can **at most** have 1 point
- Worst case: Total 8 points (corners of squares) - pi = 7 points

```
for i in range(len(strip)):
    for j in range(i, len(strip)):
        if strip[j].y - strip[i].y < k:
            update_min
        else:
            break
```



Smallest Distance between Points

Adapted From 2022 AD&C Lecture

- Time Complexity:
  $T(n) = 2\,T(n/2) + O(\text{sorting cost}) + O(\text{combining})$
  $= 2\,T(n/2) + O(n \log n) + O(n)$
  $= O(n \log^2 n)$
- Improvement:
  Instead of sorting by y at every layer, we could simply perform merge (like in merge sort) on the left and right set, which take O(n)
  Sorting cost becomes linear
- $T(n) = O(n \log n)$

Smallest Distance between Points

- There is still a randomized solution with **O(N) expected time complexity**:
- Suppose the points are randomly shuffled.
- Let's say we know the smallest distance between first **i-1** points is **s**, divide the whole grid into ***s*s cell**. Check the adjacent 9-cells of point **i**, since each cell can have at most 4 points, this part is **O(1)**.
- If the answer is updated, rebuild the grid, else we do nothing.

- Within the first **i** points, the probability that **i** is in the closest pair is **O(1/i)**. Rebuilding the grid cost **O(i)**, so the expected cost is **O(1)** when inserting point **i**. Thus the total time complexity is expected **O(N)**.

Conclusion

- There are still a lot more to be explored in Computational Geometry!
  - Point in Polygon Problem
  - Circle Intersection
  - Half-plane Intersection
  - Pick's Theorem
  - Delaunay Triangulation
  - Minkowski Sum
  - 3D Geometry
- As I have mentioned in the beginning of the lecture, most probably you won't ever see these stuff in secondary school level OI competitions
- But it never hurts to learn more! (someday it may be useful, like in ICPC)

Reference

- Previous Computational Geometry Lecture by various trainers:
- 2022 (Vincent), 2019 (Percy), 2017 (AT)
  - They went in a more technical way with more proofs and formula, while I went in a more application way. Important to read if you need to understand it in-depth!
- Blog posts by Jvruo [Simplified Chinese]:
  https://www.jvruo.com/category/Computational-geometry/
  - Nice to read since his blog explains complicated ideas in simple ways