PROJECT PRESENTATION

Dr.Ghada Noureldein

Comp 411

TEAM NAMES

presented by:

Shahd Saeed Abd-Ellatif

Farida Emad Ibrahim

Soliman Waleed Soliman

Abdelrahman Hamada Elsayed

Hams Ashraf Abd-Elhakim

Aliaa Abdel-samie El-Shahat

Nourhan Taha Mohamed

Hana Eslam Adel

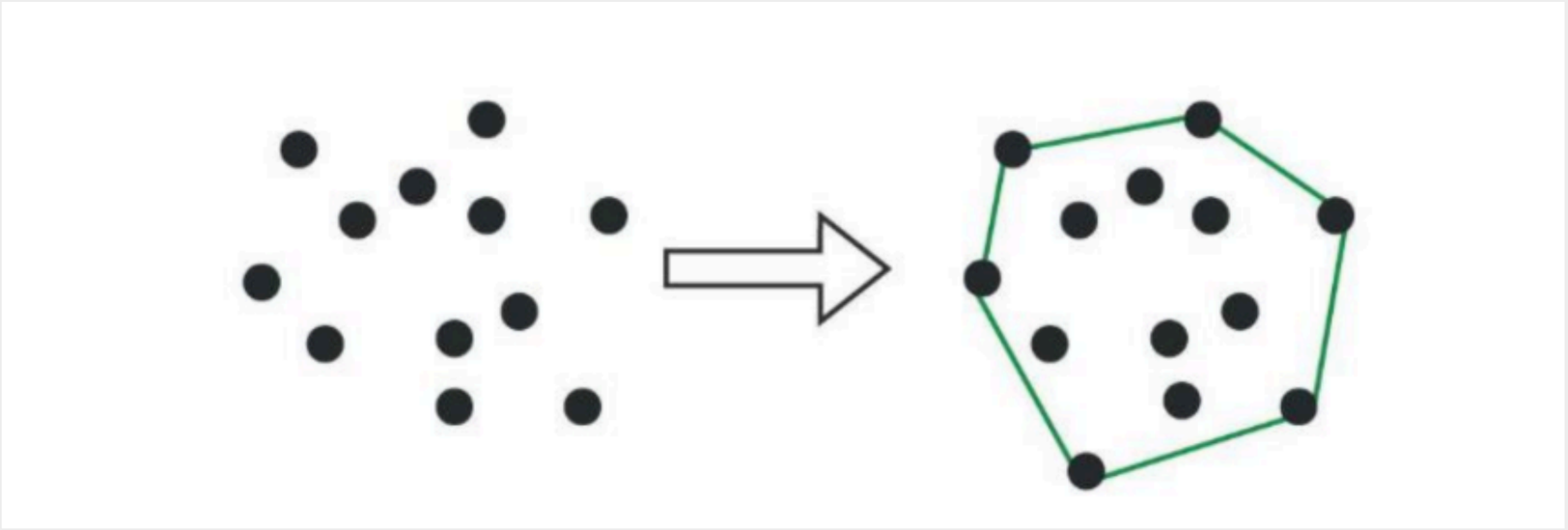
Dina Abdel-Kereem Hussein

Sara Younan Fouad

INTRODUCTION

What is Convex Hull?

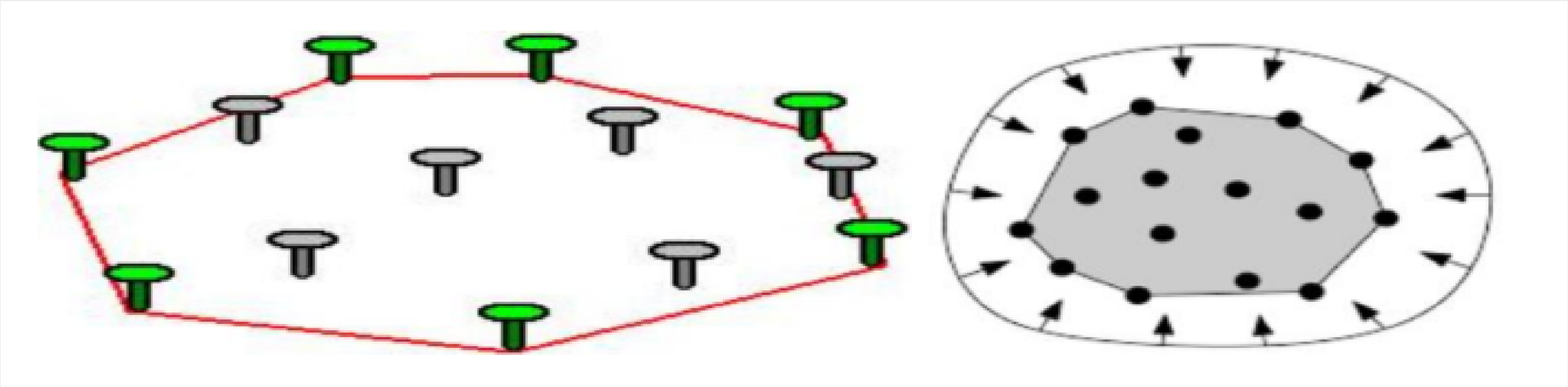
The convex hull of a set of points in a Euclidean space is the smallest convex polygon that encloses all of the points. In two dimensions (2D), the convex hull is a convex polygon.
The below image shows a 2-D convex polygon:



CONCEPT EXPLANATION

Idea: A Rubber Band Analogy

Imagine you have a board with nails hammered into it at random points. Now, if you
take a rubber band and stretch it around the entire set of nails and then let it go, it will
snap into a specific shape, wrapping tightly around the outermost nails.



CONCEPT EXPLANATION

Formal Definition

The Convex Hull of a set of points is the smallest convex set that contains all the points.

Let's unpack the key terms:

• Convex Set: A shape where for any two points inside the shape, the entire straight line segment connecting them is also inside the shape. Think of a solid
circle or a filled triangle—no indentations or "dents."

• Smallest: It's the most "efficient" or tightest-fitting convex shape that still contains all the original points. No smaller convex shape will do the job.

The Convex Hull itself is the polygon defined by the sequence of these outermost points
(the nails the rubber band touches).

CONCEPT EXPLANATION

Key Characteristics

1. It's Always Convex: By definition, it has no dents or concave bays.
2. It's a Polygon: The hull is defined by a subset of the original points, connected by edges.
3. It Contains All Points: Every point from the original set is either on the boundary of the hull or inside it.

CONCEPT EXPLANATION

How is convex hull Calculated?

While the rubber band analogy is intuitive, a computer needs an algorithm. Common ones include:

• Graham Scan: Sorts the points by their angle around a pivot point (typically the bottom-most point) and then builds the hull by checking for "turns."
• Jarvis March (or Gift Wrapping): Finds the left-most point and then "wraps" around the set by repeatedly finding the point that is most to the right (from the current point's perspective), mimicking the rubber band.
In summary, the Convex Hull is a fundamental geometric concept that provides a simple, convex boundary for a complex set of points, with wide-ranging applications from robotics and gaming to data science.
How to calculate the convex Hull using Graham Scan with examples
Of course! Let's dive into the Graham Scan algorithm for calculating the Convex Hull. It's one of the most elegant and efficient algorithms for this task.

ALGORITHM

Graham Scan Algorithm

The Graham Scan works by first finding a guaranteed starting point on the hull, then sorting all other points based on their angle around this point, and finally building the hull by traversing these sorted points and eliminating points that would create a concave turn.

STEPS

Step-by-Step Process

Let's go through the steps with a clear example. Consider these points:

P0 = (0, 0), P1 = (1, 2), P2 = (2, 1), P3 = (2, 4), P4 = (3, 3), P5 = (4, 0),P6 = (1,1)

Step 1: Find the Anchor Point

Find the point with the lowest y-coordinate (and if tied, the leftmost one). This point is guaranteed to be on the convex hull.

In our example:

- P0 = (0, 0) has the lowest y-coordinate → Anchor Point

STEPS

Step 2: Sort by Polar Angle

Sort all remaining points by their polar angle relative to the anchor point. If two points have the same angle, keep only the one farthest from the anchor.

Calculating angles relative to P0(0, 0): $\theta = \tan^{-1}(y/x)$ in radiance

- P1(1, 2): angle = 63.4°
- P2(2, 1): angle = 26.6°
- P3(2, 4): angle = 63.4°
- P4(3, 3): angle = 45°
- P5(4, 0): angle = 0°
- P6(1, 1): angle = 45°

then p3=(2,4) and discard p1=(1,2) and take p4=(3,3) and discard p6=(1,1)

Sorted order: p0(0,0), P5(4, 0), P2(2, 1), P4(3, 3), P3(2, 4)

STEPS

Step 3: Build the Hull

Use a stack to build the convex hull. Process points in sorted order, and for each point, check if moving to it creates a clockwise turn (which would make the hull concave).

Initial stack: [P0, P5]

Process P2=(2, 1):

• Check turn from P5→P2:

$D=(x2-x1)(y3-y1)-(y2-y1)(x3-x1)$
$D=(4-0)(1-0)-(0-0)(2-0)=+4$ Counter-clockwise ✓

• Push P2 → Stack: [P0, P5, P2]

STEPS

Process P4=(3, 3):

D=(2-4)(3-0)-(1-0)(3-4)=-5 clockwise

•Now, Pop P2 from stack

•And, Check turn from P5→P4:
D=(4-0)(3-0)-(0-0)(3-0)=+12 Counter-clockwise ✓

• Push P4 → Stack: [P0, P5, P4]

Process P3=(2, 4):

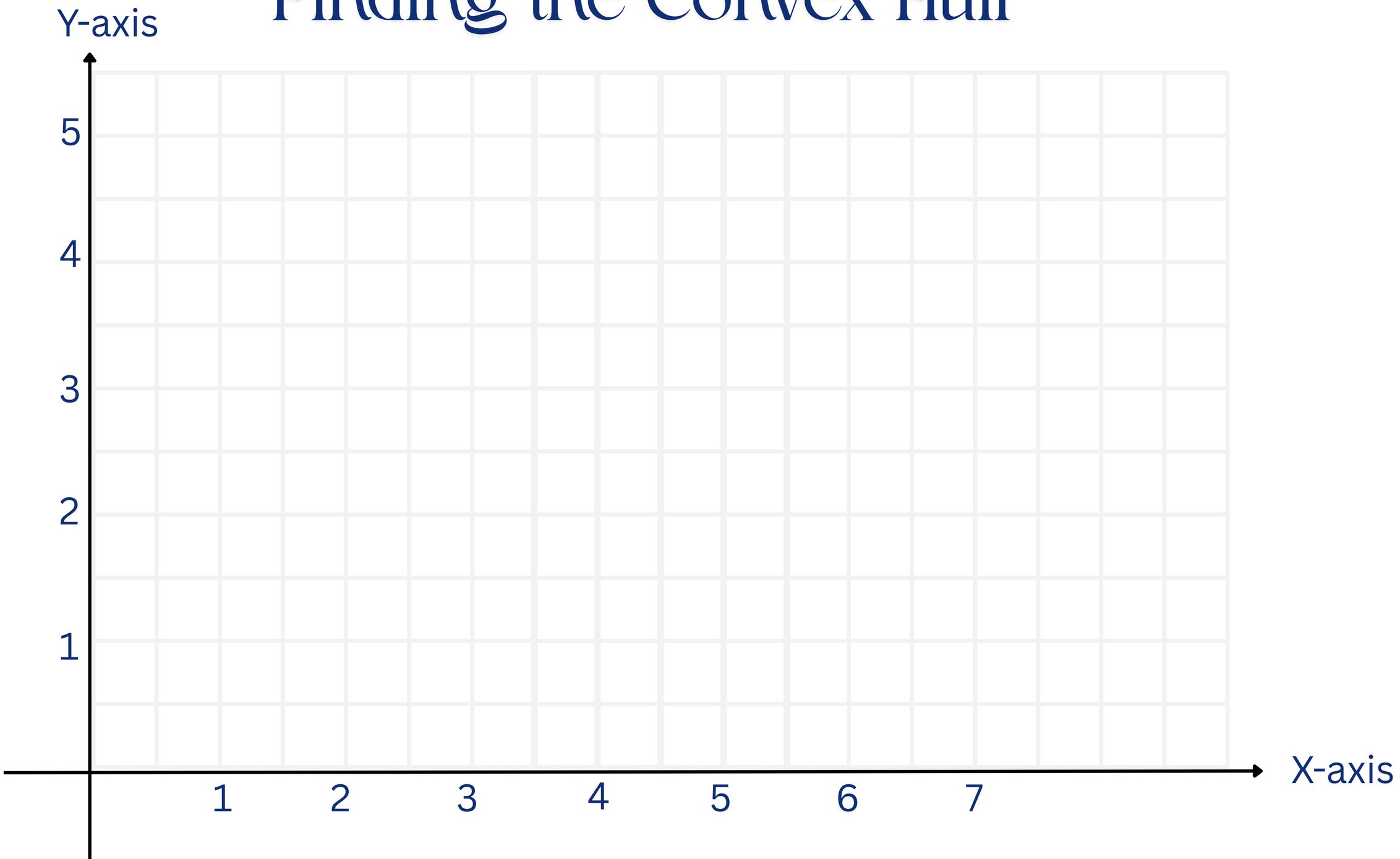
• Check turn from P4→P3:

D=(3-4)(4-0)-(3-0)(2-4)=+2 Counter-clockwise ✓

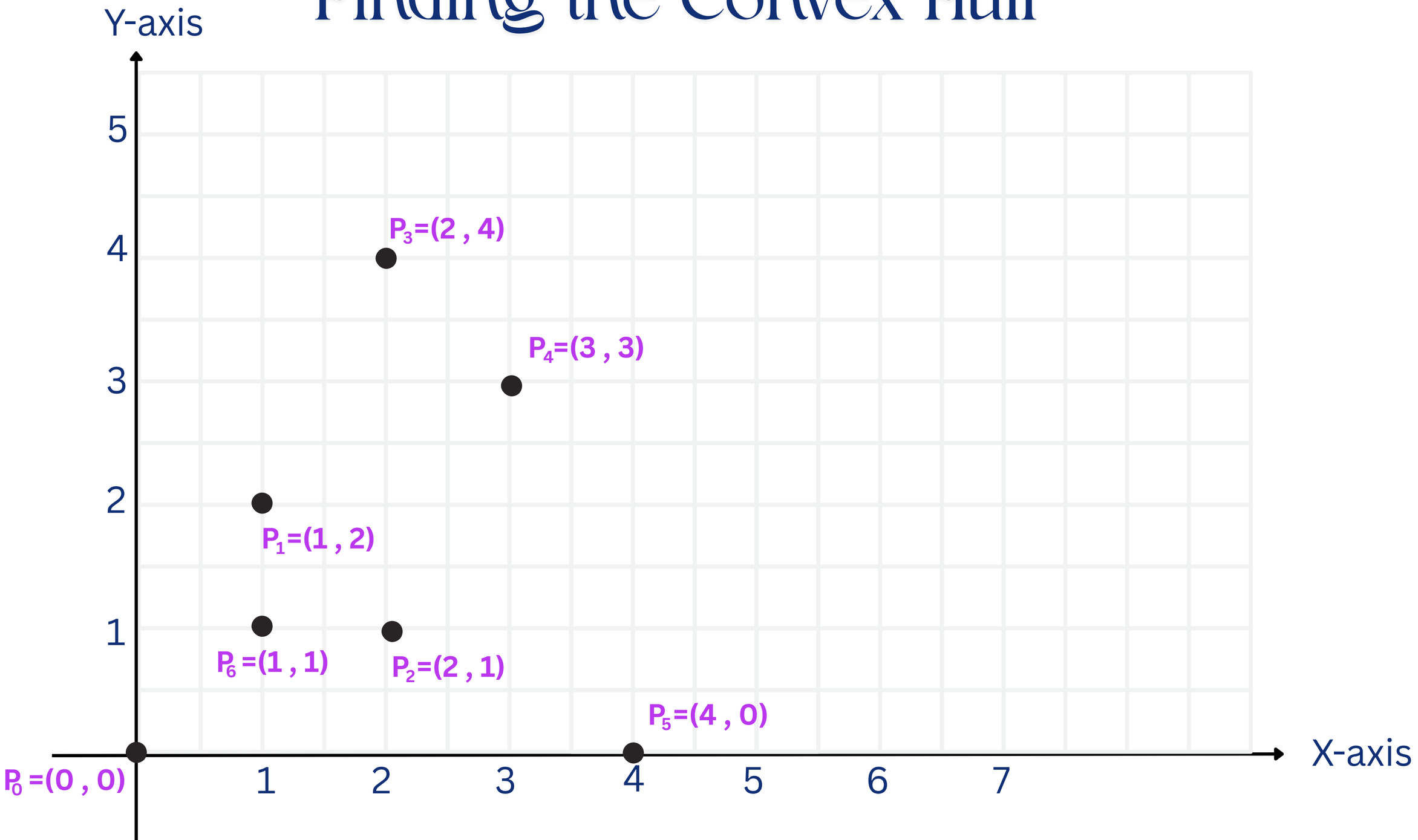
• Push P3 → Stack: [P0, P5, P4, P3]

Final Hull: P0 → P5 → P4 → P3

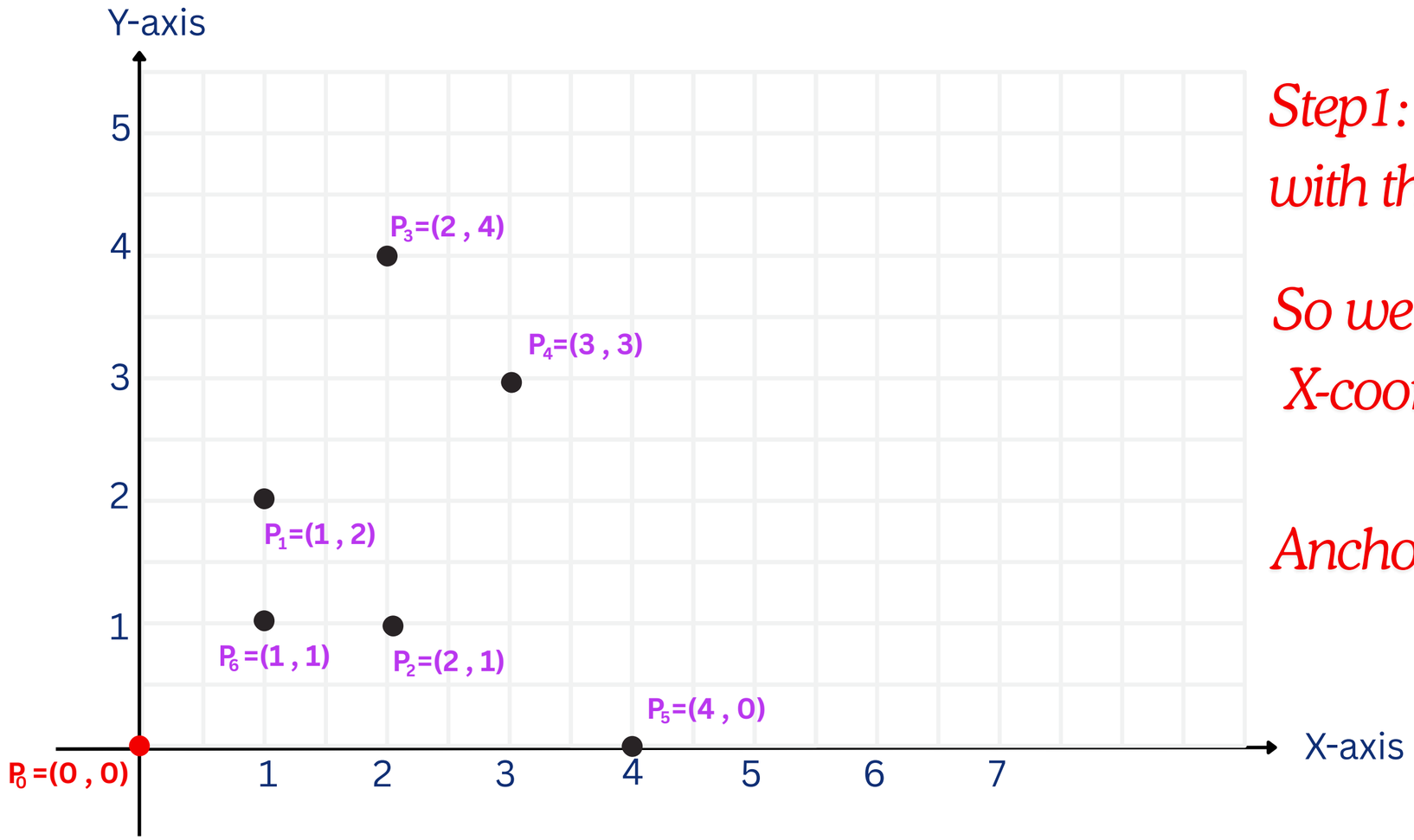
Finding the Convex Hull



Finding the Convex Hull



Finding the Convex Hull

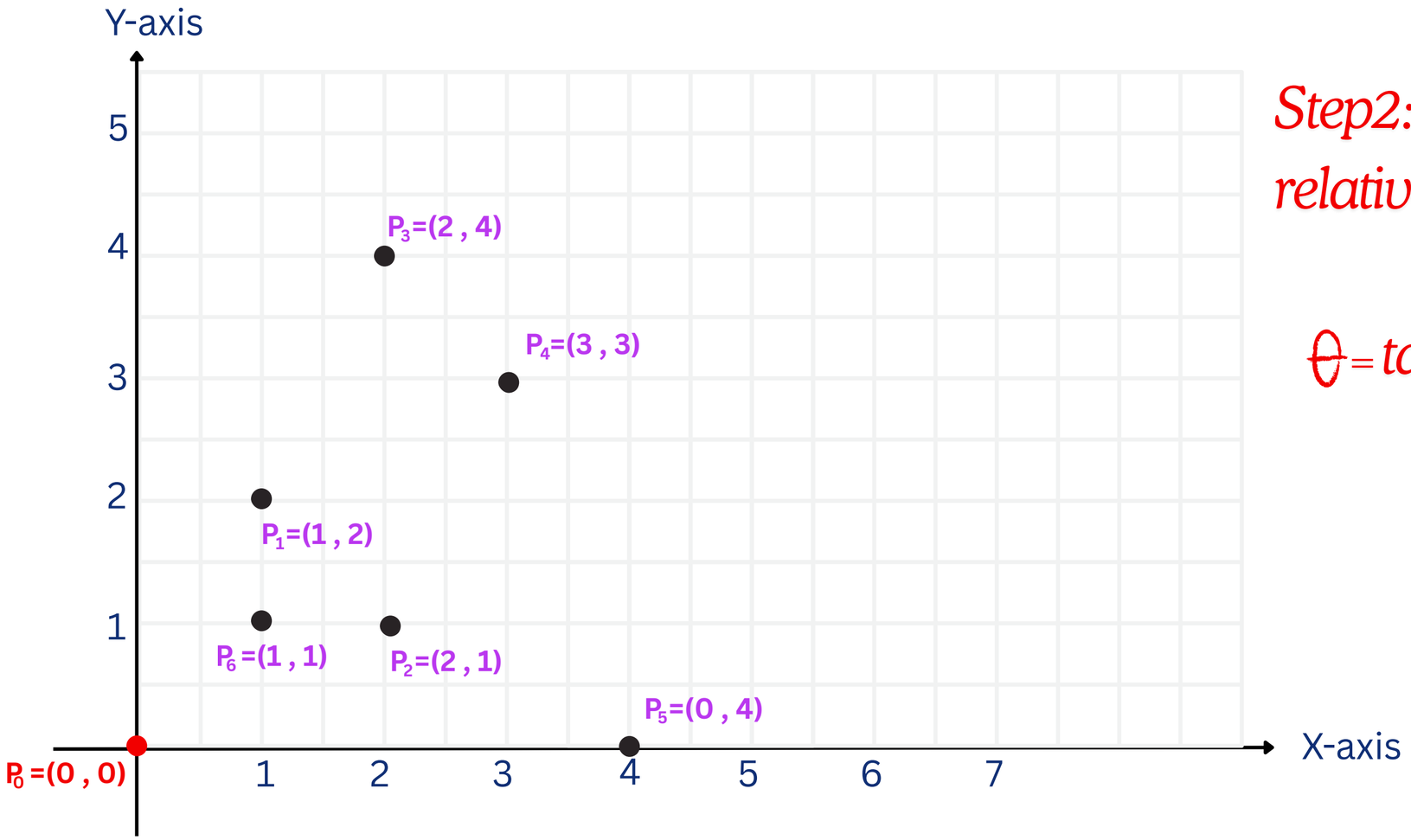


Step1: Finding the Anchor point with the lowest Y-coordinate coordinate

So we choose the point with lowest X-coordinate

Anchor point : P0 (0,0)

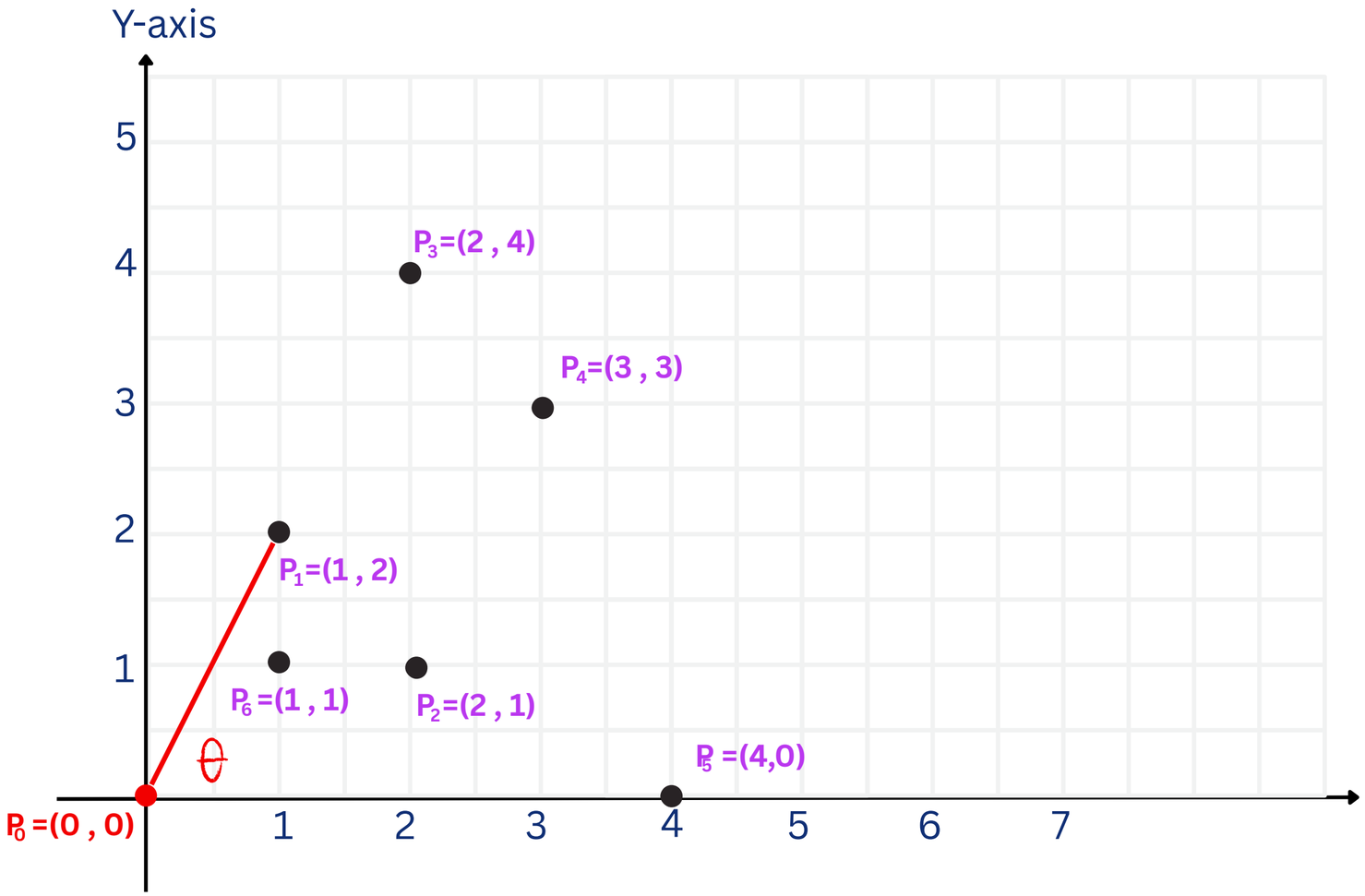
Finding the Convex Hull



Step2: Calculating polar angles relatives to Anchor point

$$\theta = \tan^{-1}\left(\frac{y_2 - y_1}{x_2 - x_1}\right)$$

Finding the Convex Hull

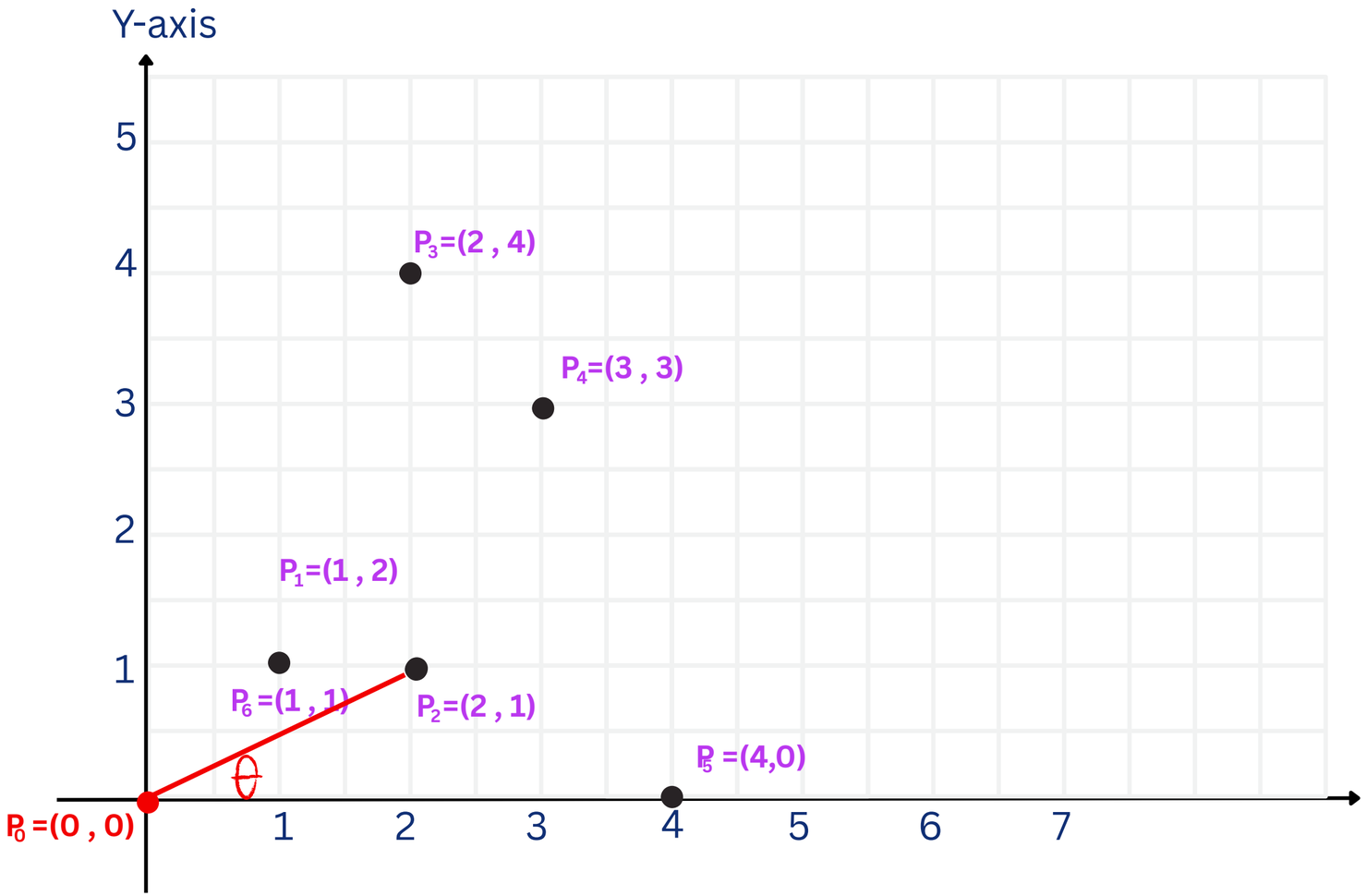


Step2: Calculating polar angles relative to P0=(0 , 0)

$$P1(1 , 2)$$

$$\theta = \tan^{-1}\left(\frac{2-0}{1-0}\right) = 63.43^{o}$$

Finding the Convex Hull

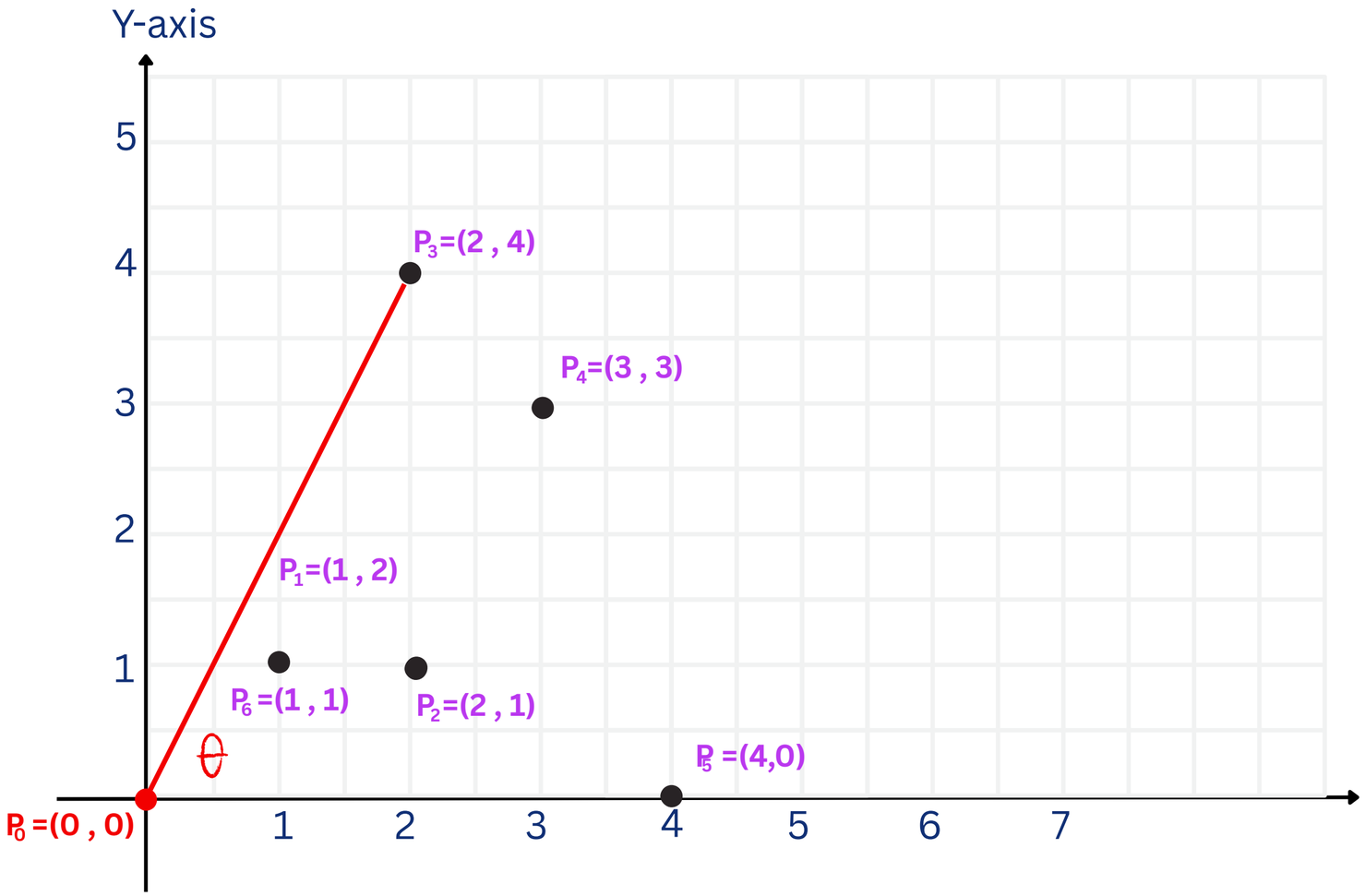


Step2: Calculating polar angles relative to P0=(0 , 0)

$$P2\ (2\ ,\ 1)$$

$$\theta = \tan^{-1}\left(\frac{1 - 0}{2 - 0}\right) = 26.57^{o}$$

Finding the Convex Hull

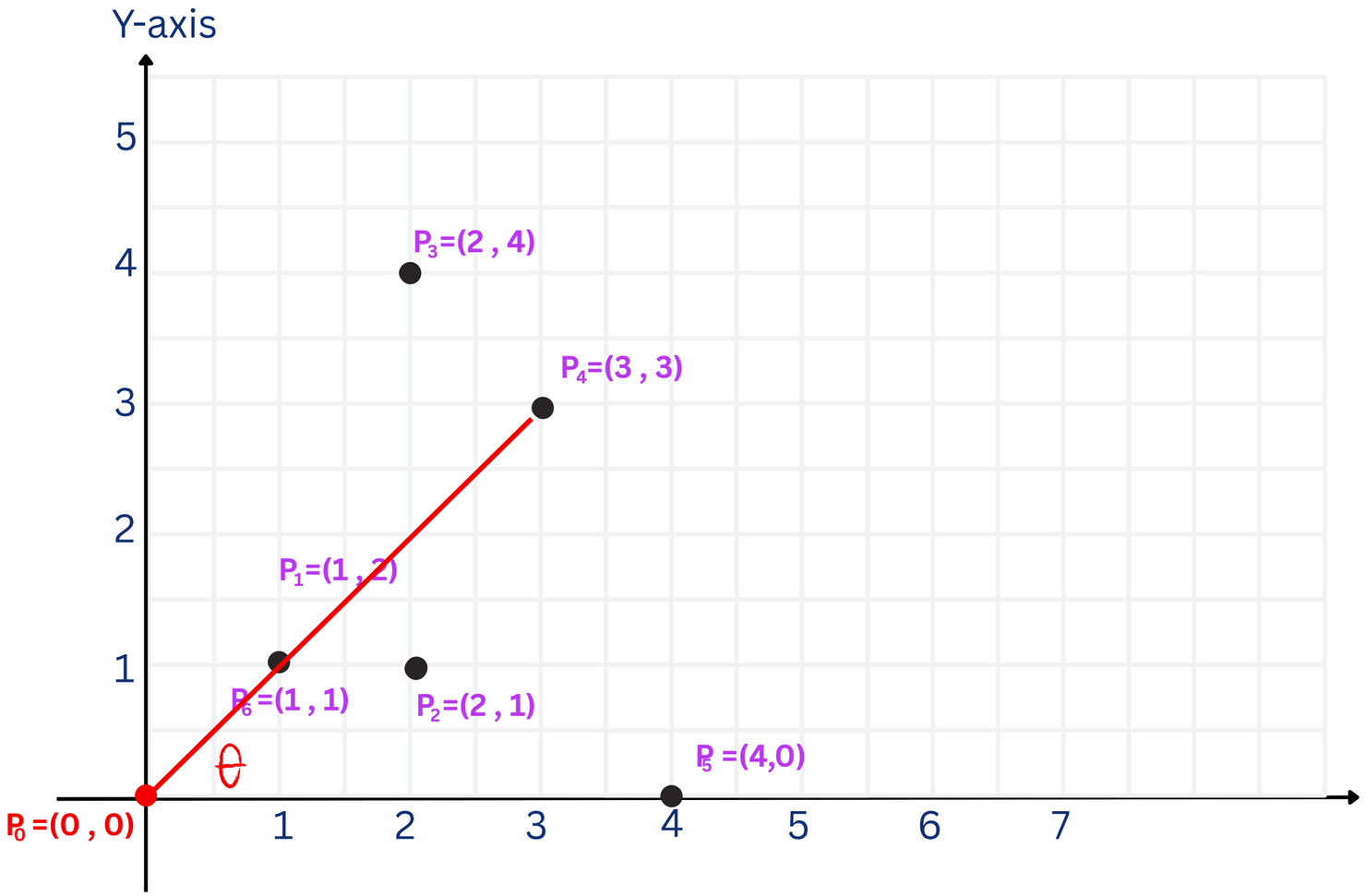


Step2: Calculating polar angles relative to P0=(0 , 0)

$$P3\ (2\ ,\ 4)$$

$$\theta = \tan^{-1}\left(\frac{4 - 0}{2 - 0}\right) = 63.43^{o}$$

Finding the Convex Hull

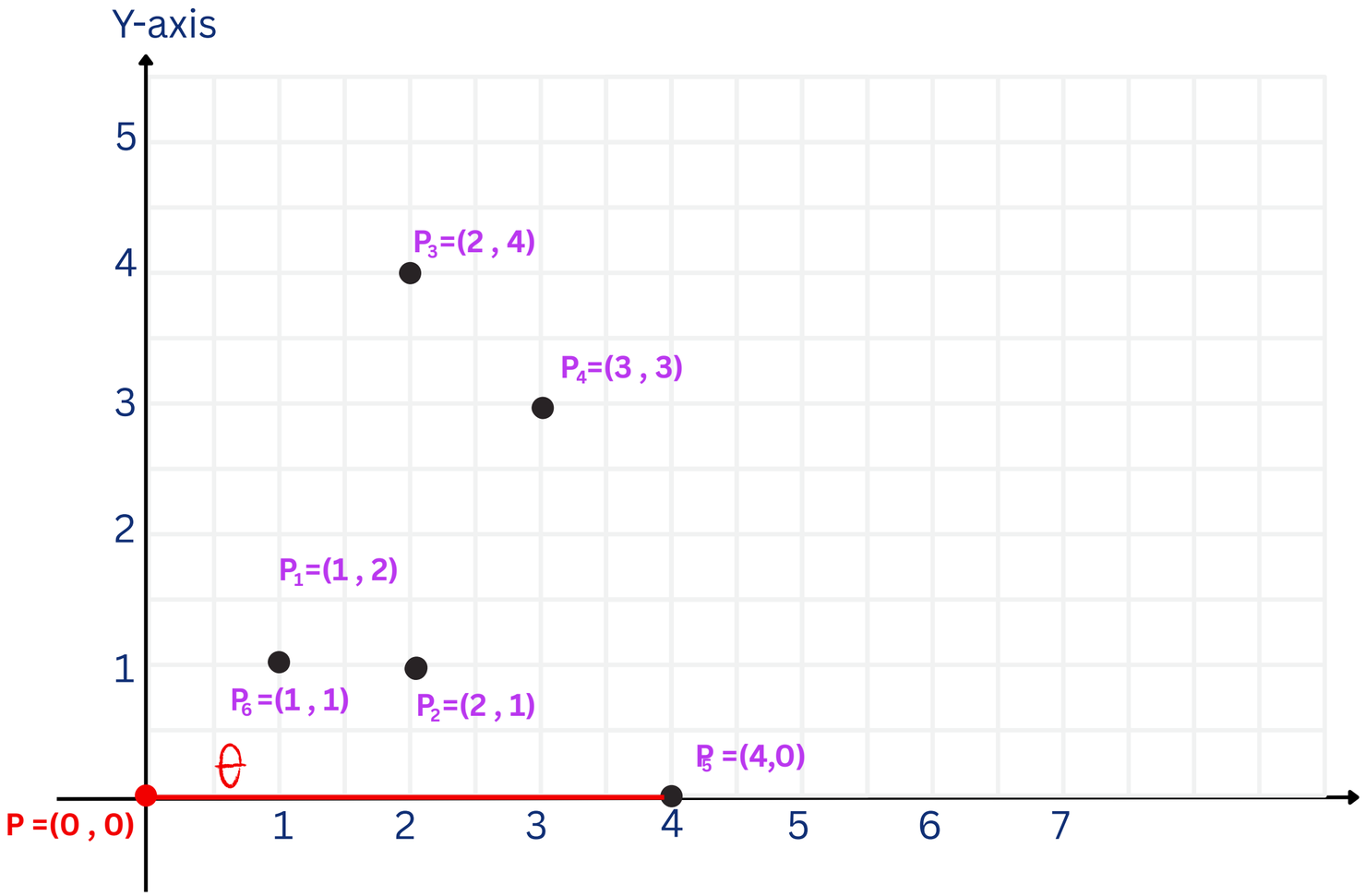


Step2: Calculating polar angles relative to P0=(0 , 0)

$$P4\ (3\ ,\ 3)$$

$$\theta = \tan^{-1}\left(\frac{3 - 0}{3 - 0}\right) = 45^{o}$$

Finding the Convex Hull

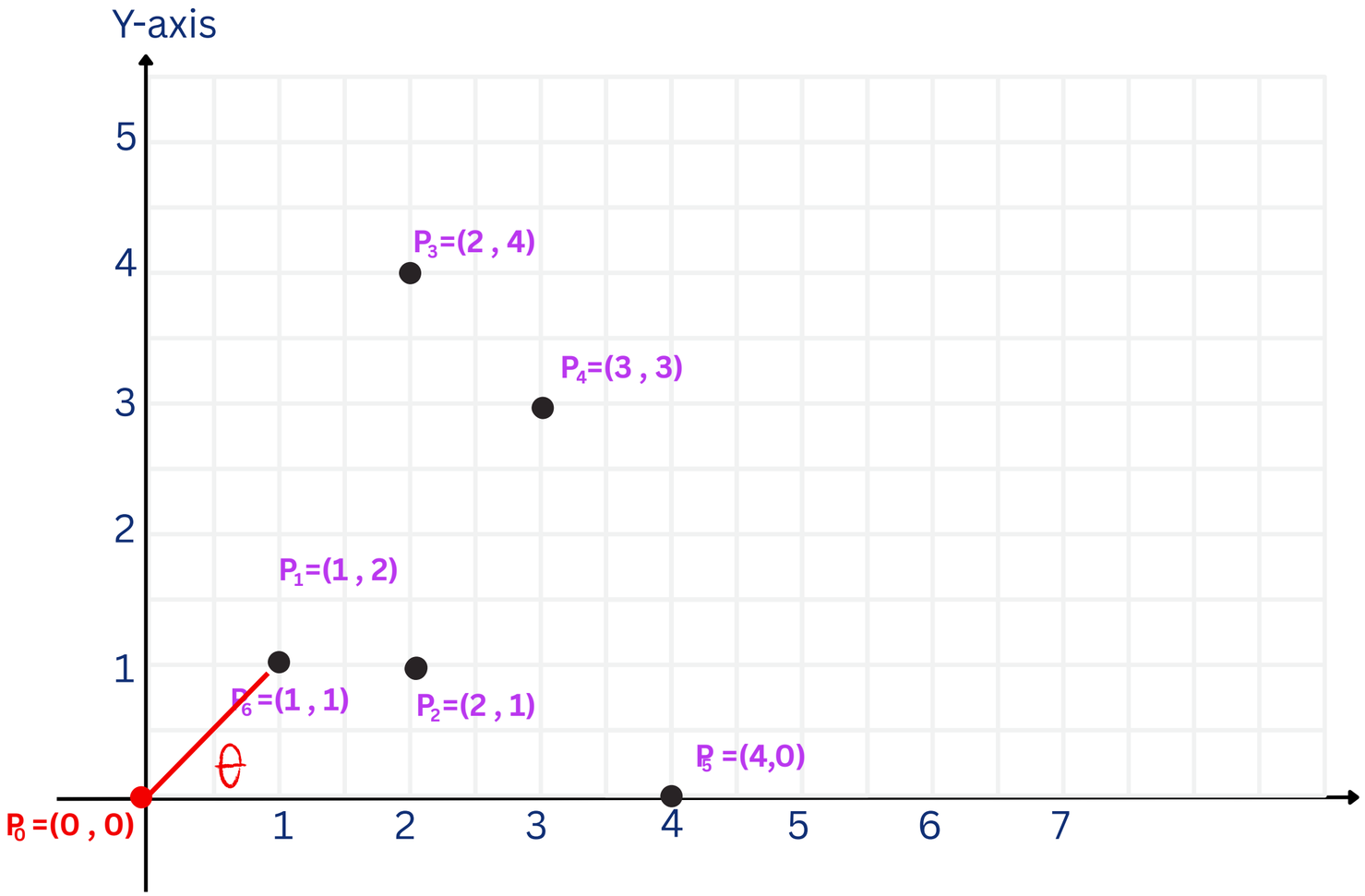


Step2: Calculating polar angles relative to P0=(0 , 0)

$$P5\ (4\ ,\ 0)$$

$$\theta = \tan^{-1}\left(\frac{0 - 0}{4 - 0}\right) = 0.0^{o}$$

Finding the Convex Hull



Step2: Calculating polar angles relative to P0=(0 , 0)

$P6\ (1\ ,\ 1)$

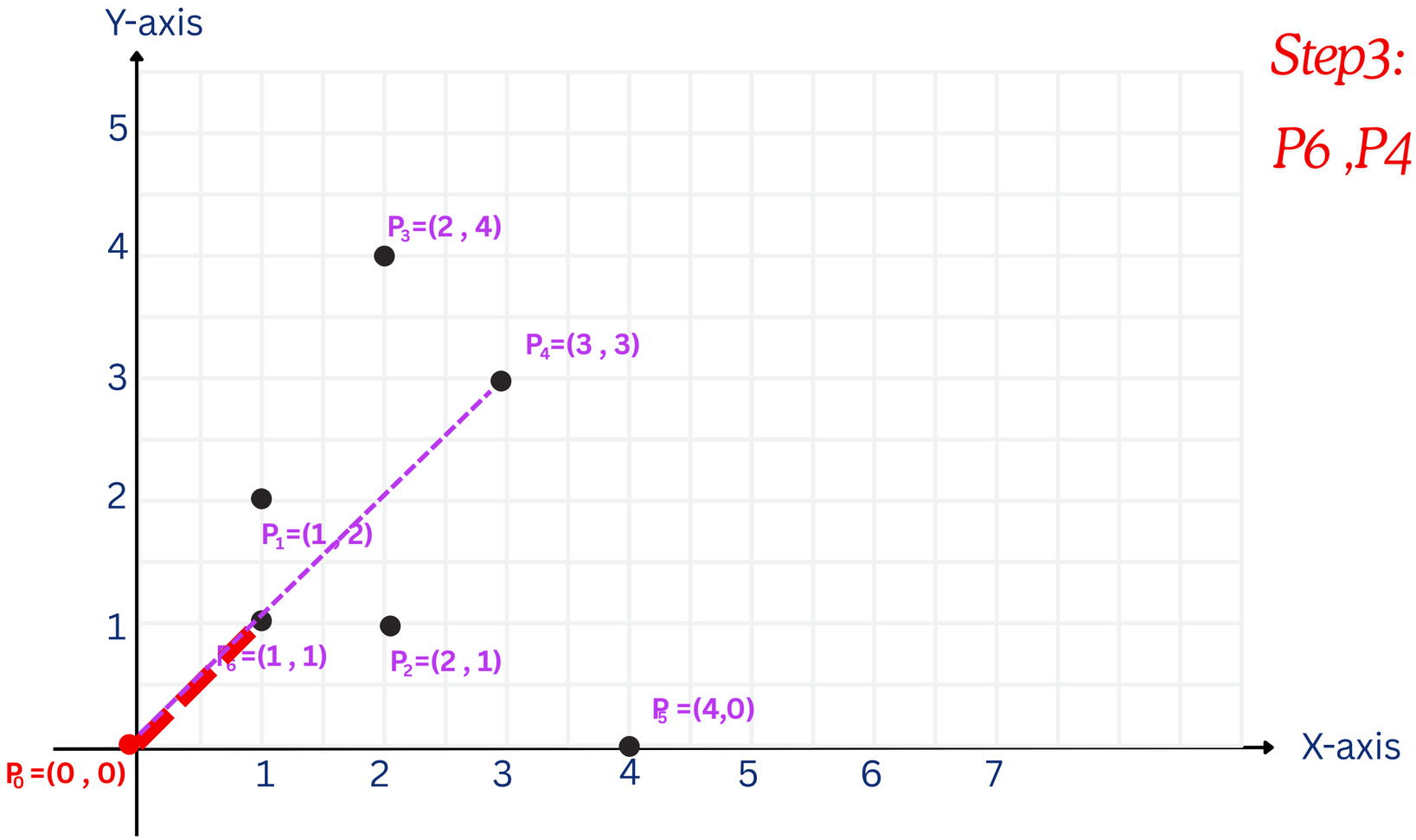
$$\theta = \tan^{-1}\left(\frac{1 - 0}{1 - 0}\right) = 45^{o}$$

Finding the Convex Hull

Step3: sort points by polar angle :

point	Angle(θ)
P5 (4 , 0)	*0.0°*
P2 (2 , 1)	*26.57°*
P6 (1 , 1)	*45°*
P4 (3 , 3)	*45°*
P3 (2 , 4)	*63.43°*
P1 (1 , 2)	*63.43°*

Finding the Convex Hull

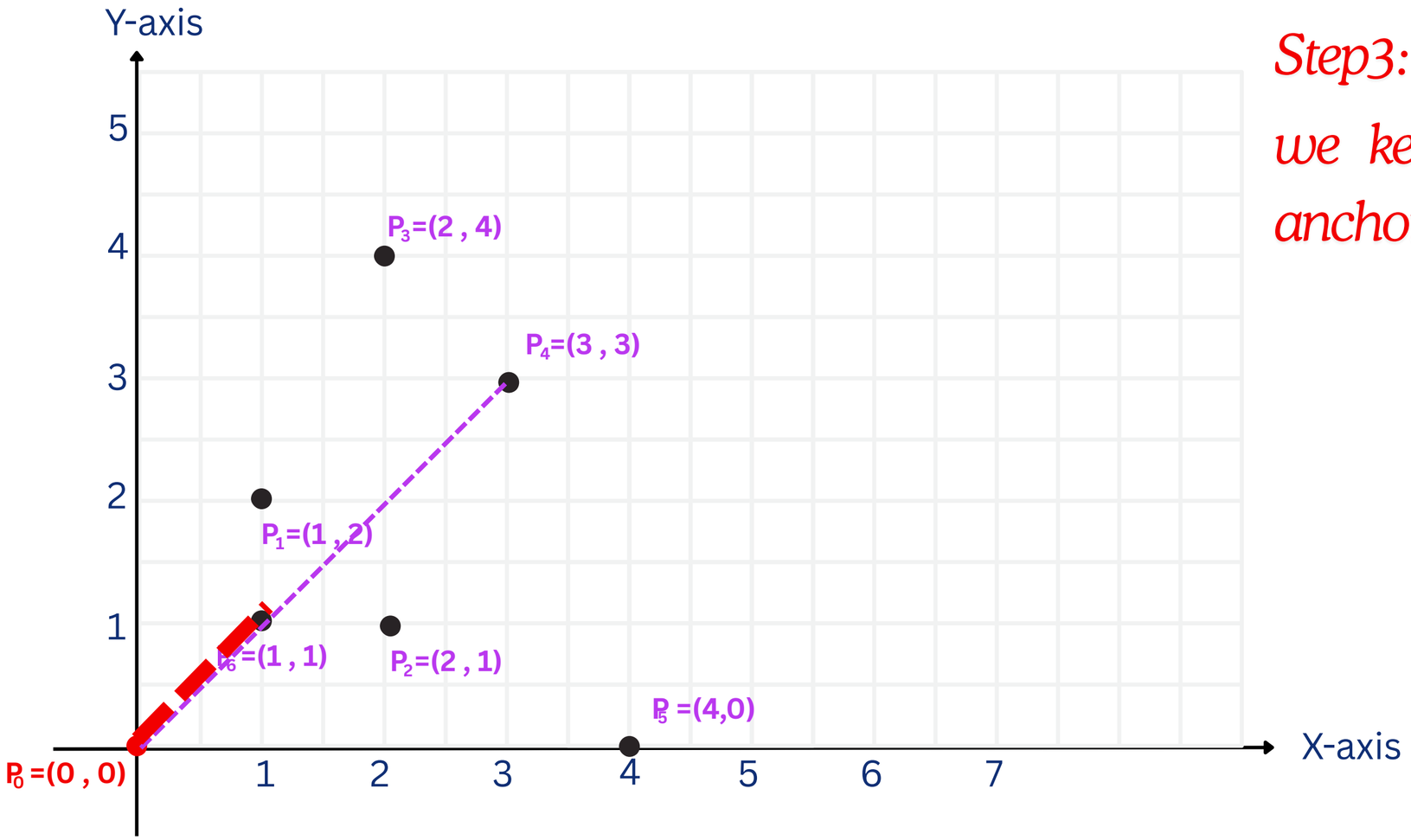


Step3: sort points by polar angle :

P6 ,P4 have the same angle

point	Angle(θ)
P5 (4 , 0)	0.0^o
P2 (2 , 1)	26.57^o
P6 (1 , 1)	45^o
P4 (3 , 3)	45^o
P3 (2 , 4)	63.43^o
P1 (1 , 2)	63.43^o

Finding the Convex Hull

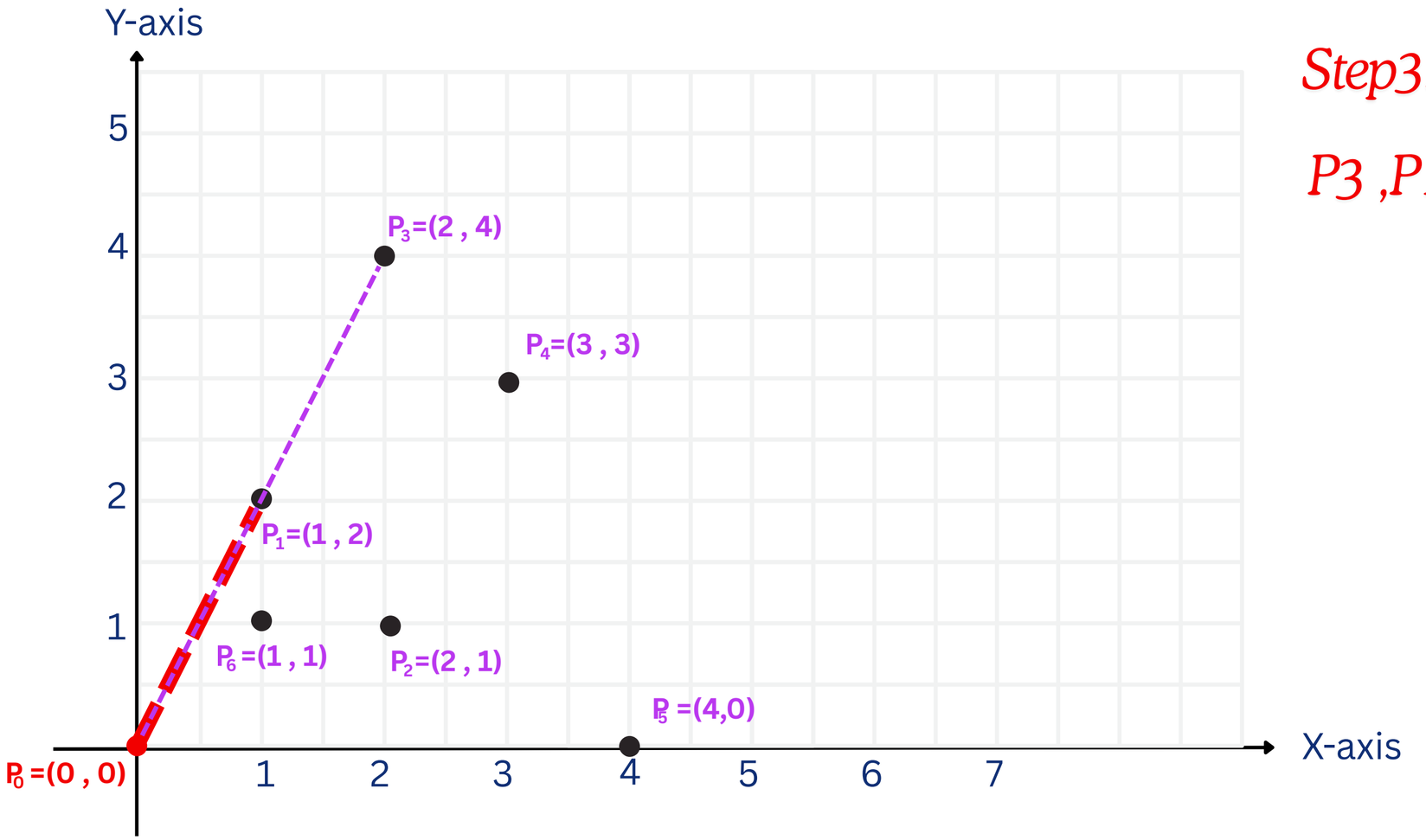


Step3: sort points by polar angle :

we keep only the farthest from the anchor -> P4

point	Angle(θ)
P5 (4 , 0)	*0.0°*
P2 (2 , 1)	*26.57°*
P4 (3 , 3)	*45 °*
P3 (2 , 4)	*63.43°*
P1 (1 , 2)	*63.43°*

Finding the Convex Hull

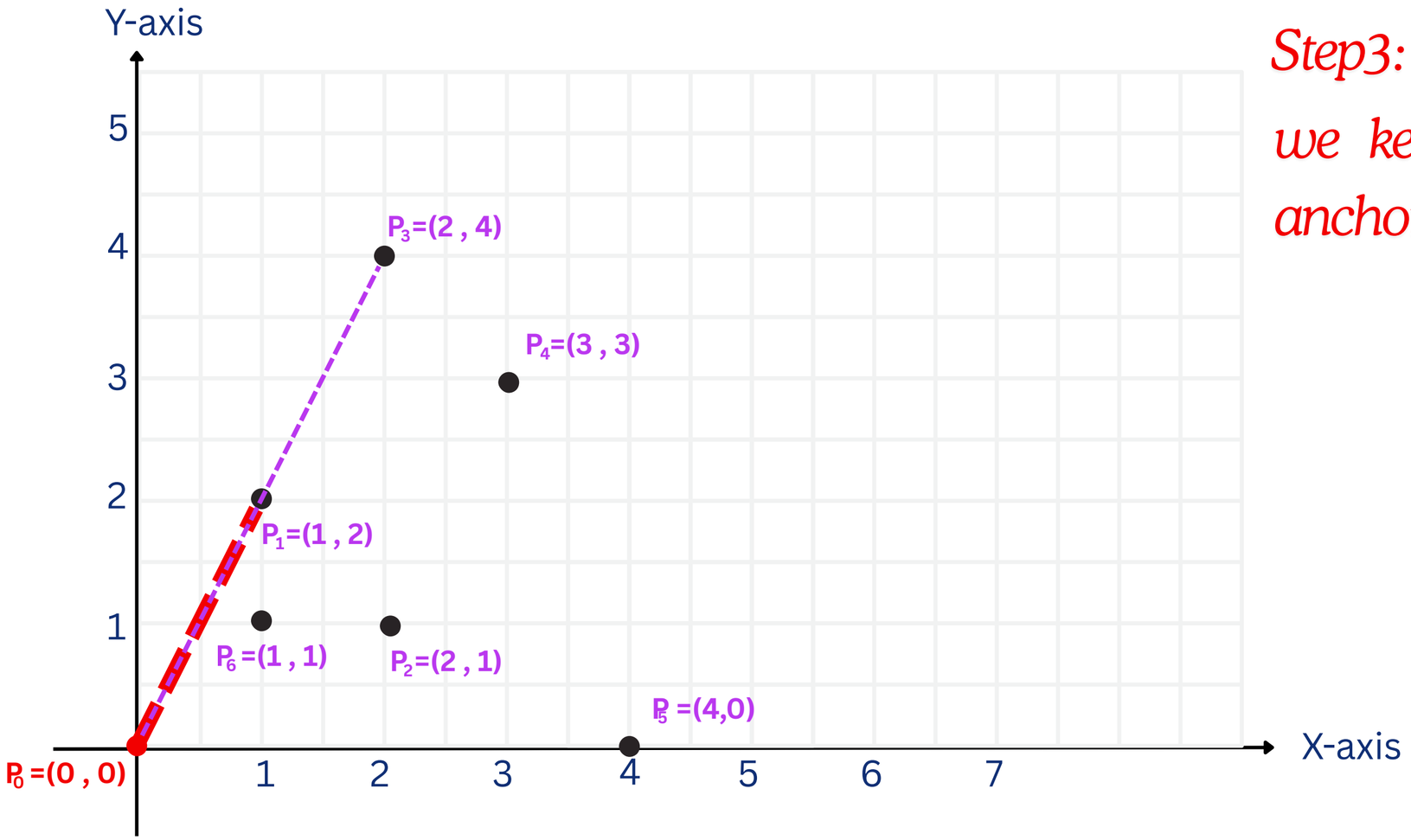


Step3: sort points by polar angle :

P3 ,P1 have the same angle

point	Angle(θ)
P5 (4 , 0)	*0.0 °*
P2 (2 , 1)	*26.57 °*
P4 (3 , 3)	*45 °*
P3 (2 , 4)	*63.43 °*
P1 (1 , 2)	*63.43 °*

Finding the Convex Hull



Step3: sort points by polar angle :

we keep only the farthest from the anchor -> P3

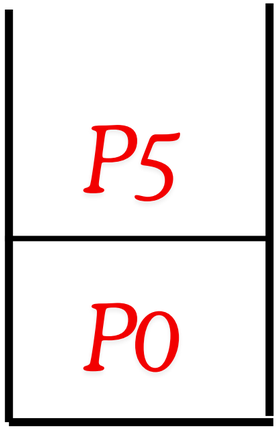
point	Angle(θ)
P5 (4 , 0)	*0.0 °*
P2 (2 , 1)	*26.57 °*
P4 (3 , 3)	*45 °*
P3 (2 , 4)	*63.43°*

Finding the Convex Hull

Step4: Build the hull

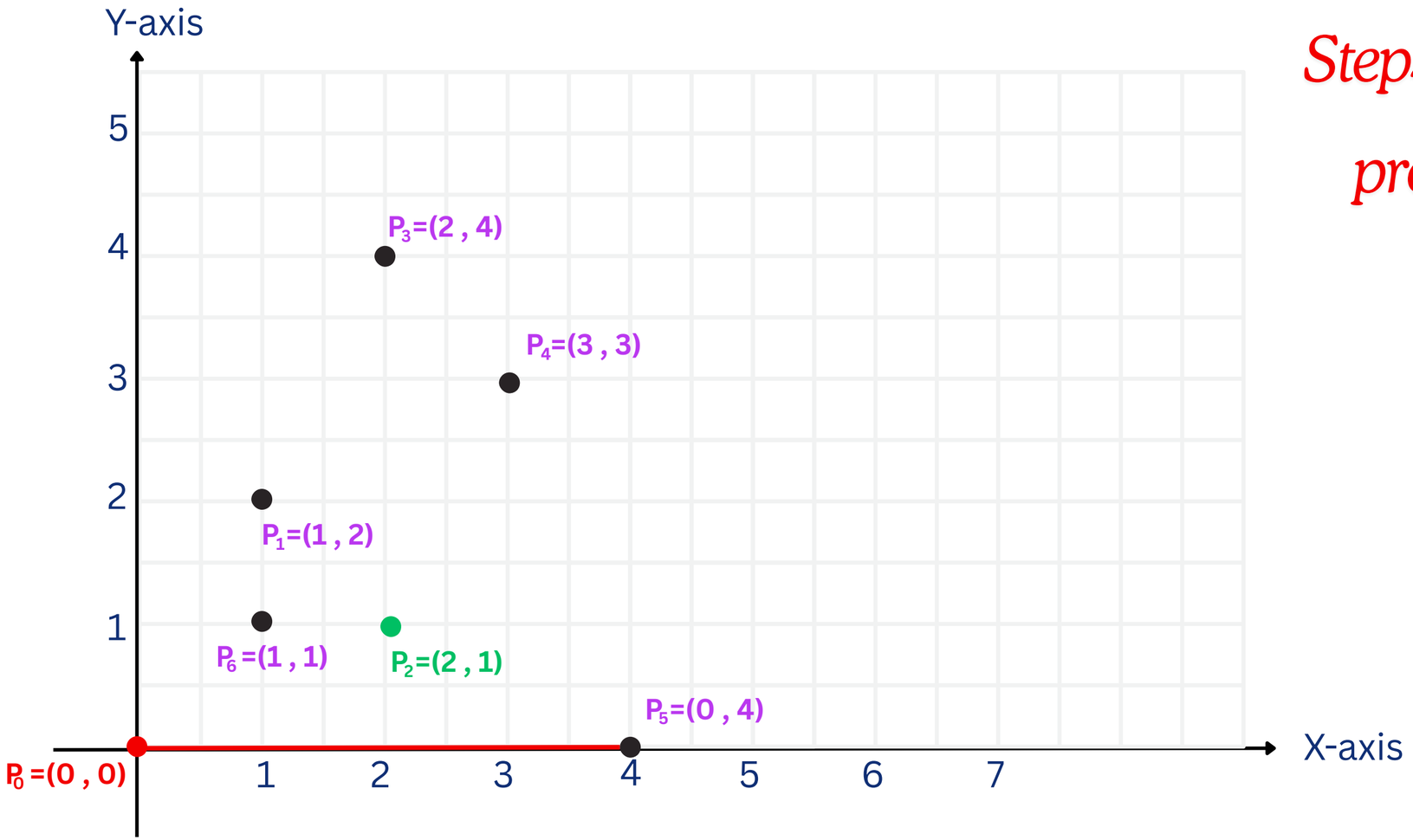
By using a stack we will build convex hull and process each sorted point and check oriantation for each point .

Stack



point	Angle(θ)
P5 (4 , 0)	0.0 ^{o}
P2 (2 , 1)	26.57 ^{o}
P4 (3 , 3)	45 ^{o}
P3 (2 , 4)	63.43 ^{o}

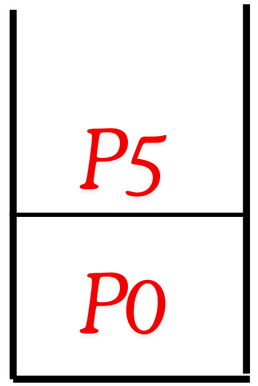
Finding the Convex Hull



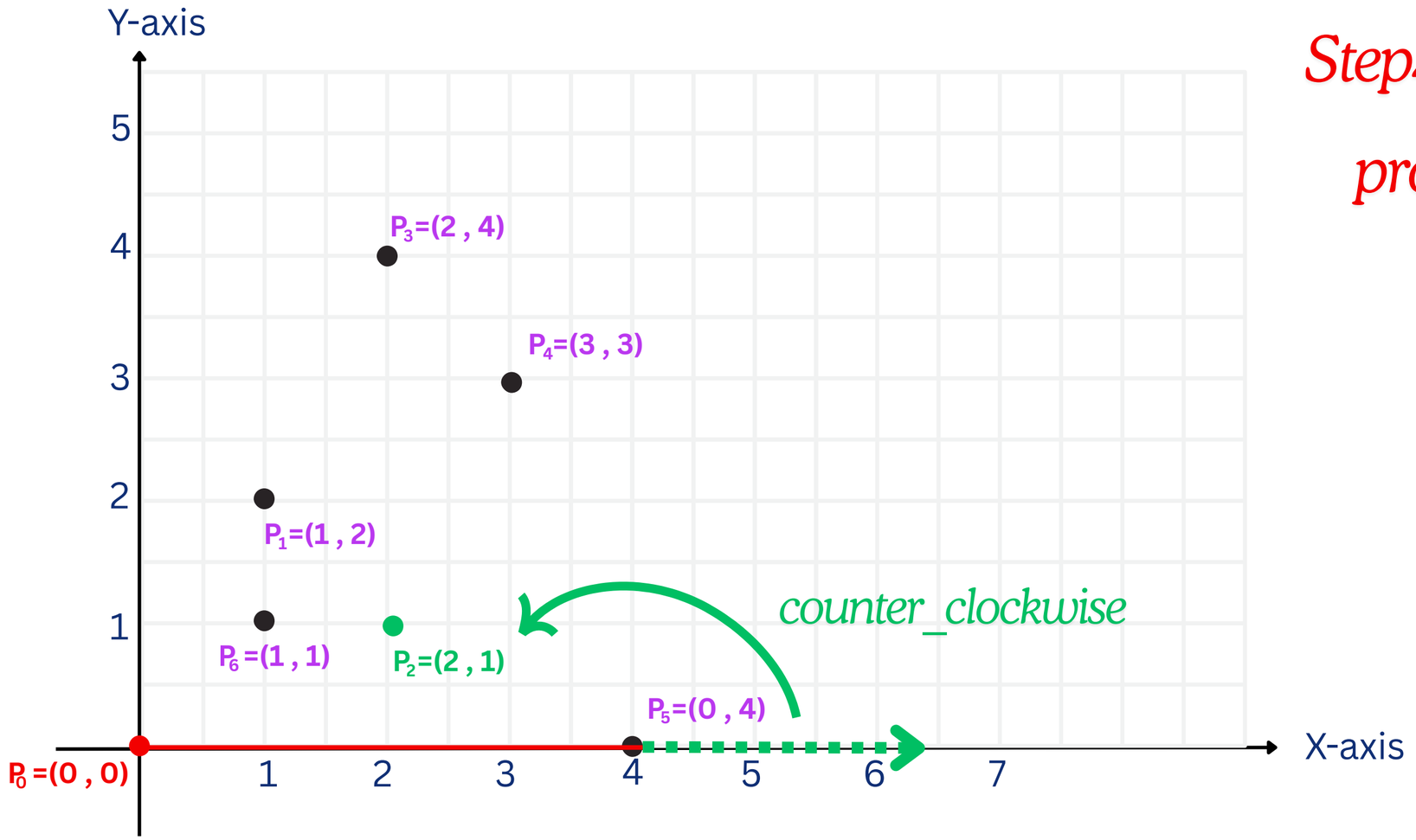
Step4 : build the hull

process P2 (2 , 1)

Stack



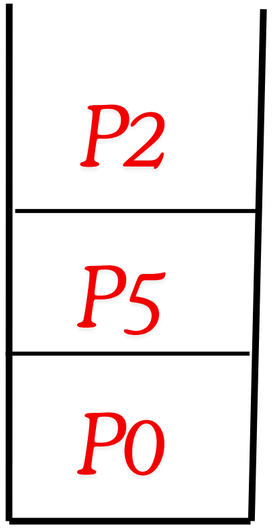
Finding the Convex Hull



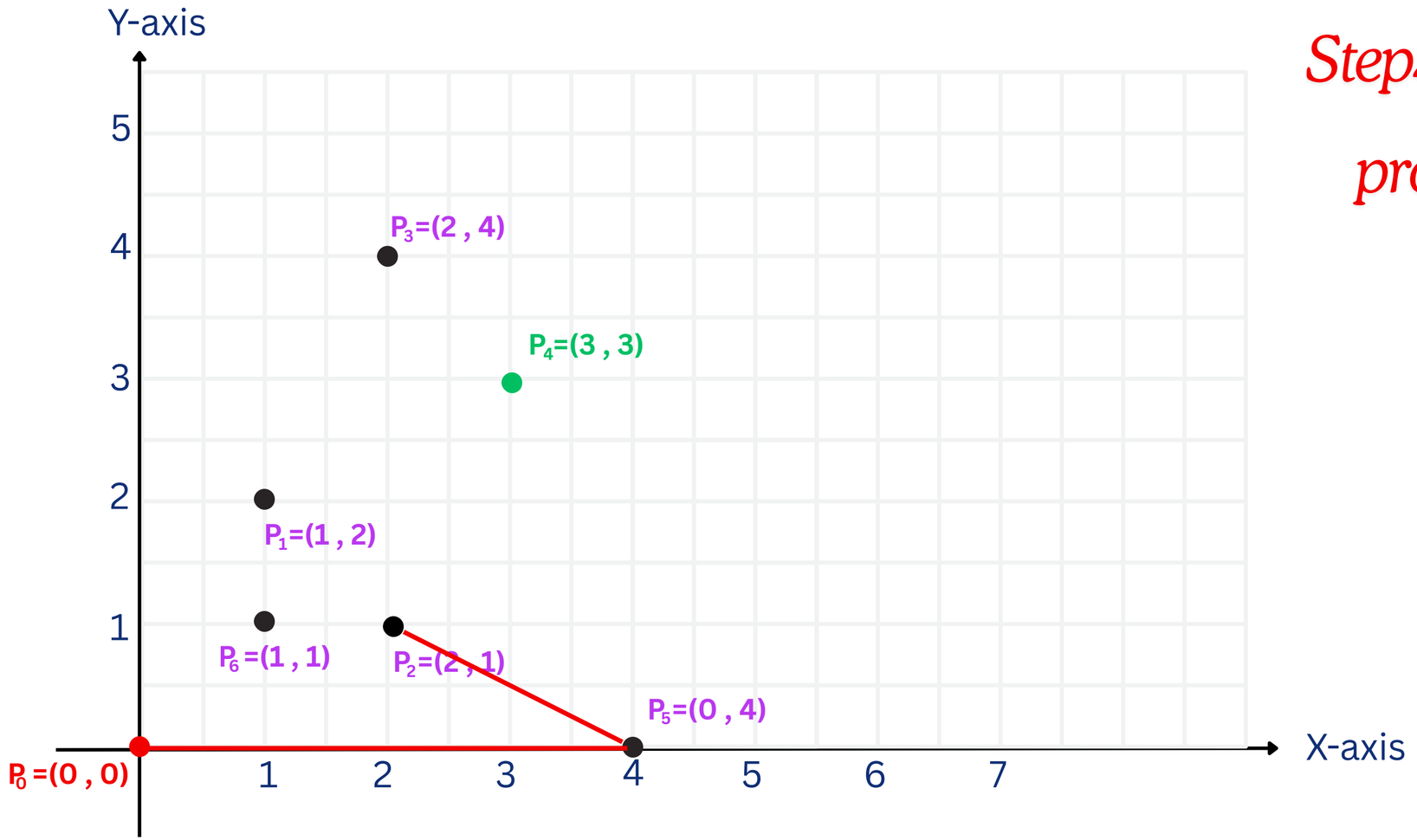
Step4 : build the hull

process P2 (2 , 1)

Stack



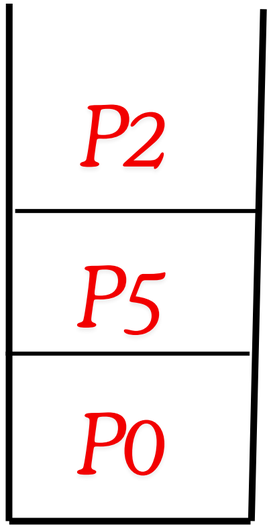
Finding the Convex Hull



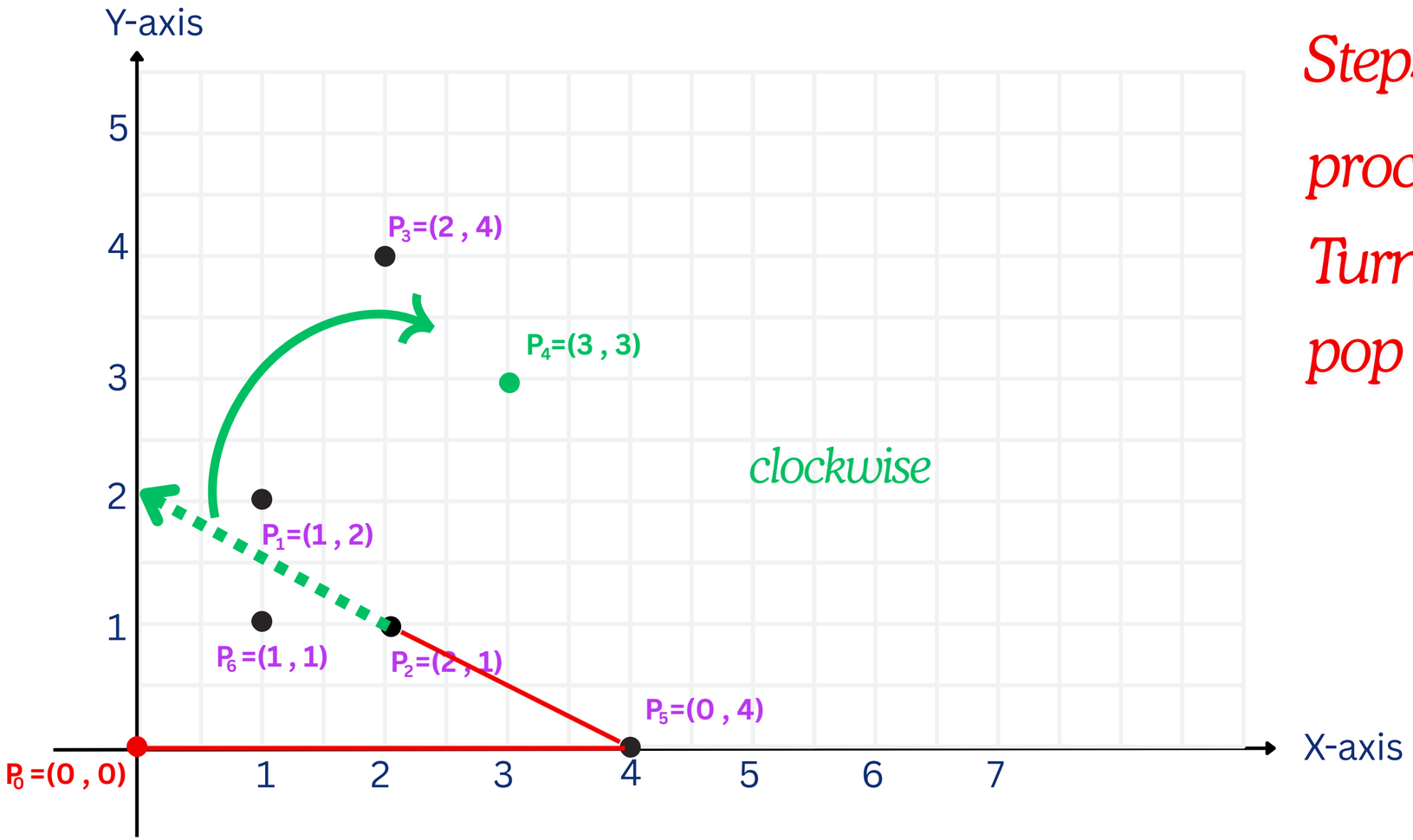
Step4 : build the hull

process P4 (3 , 3)

Stack



Finding the Convex Hull

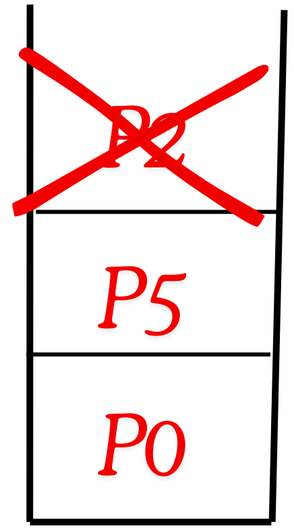


Step4 : build the hull

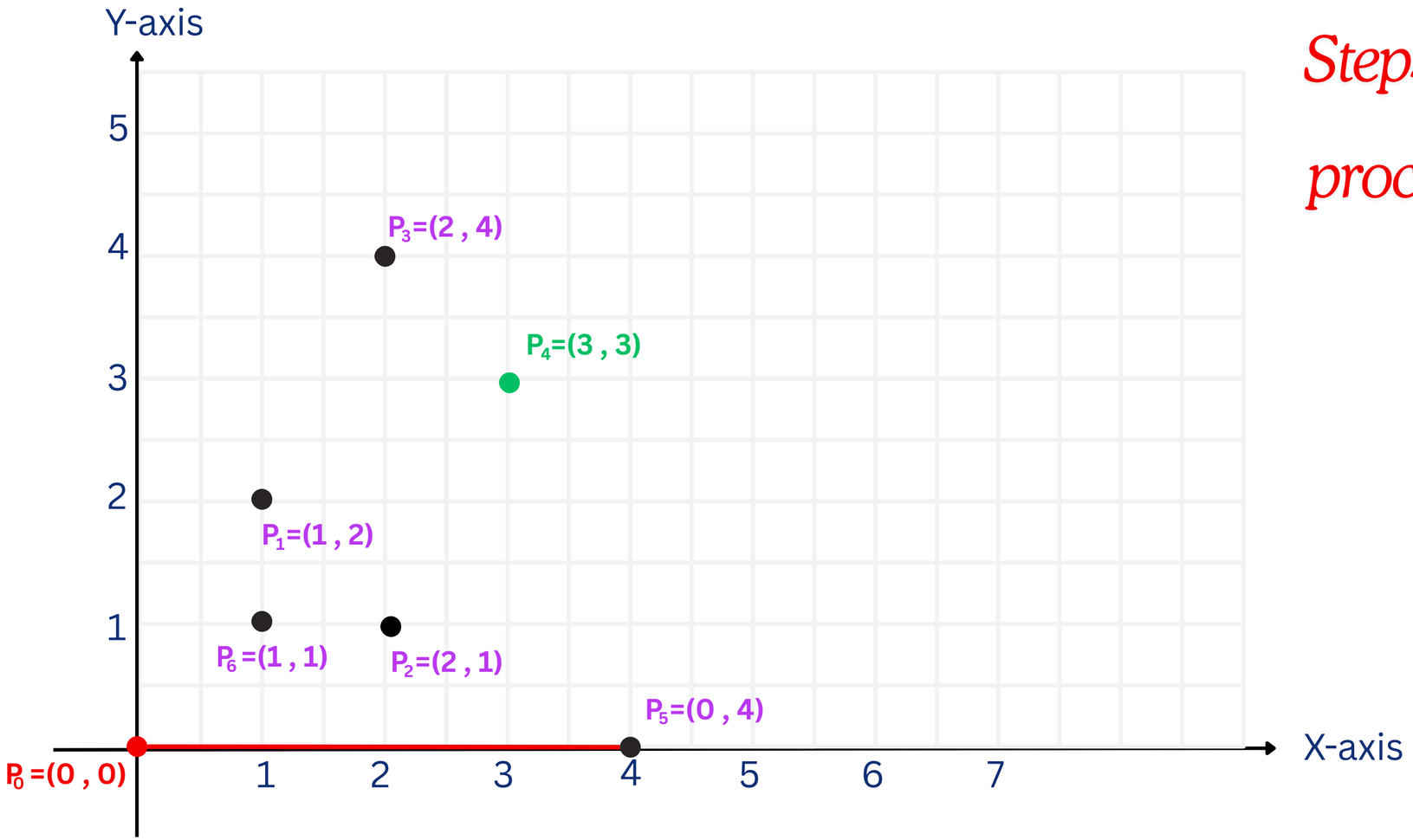
process P4 (3 , 3)

Turn is clockwise so we pop P2 from stack

Stack



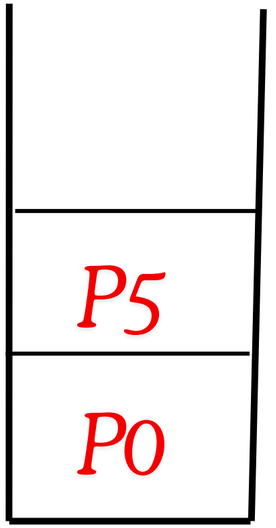
Finding the Convex Hull



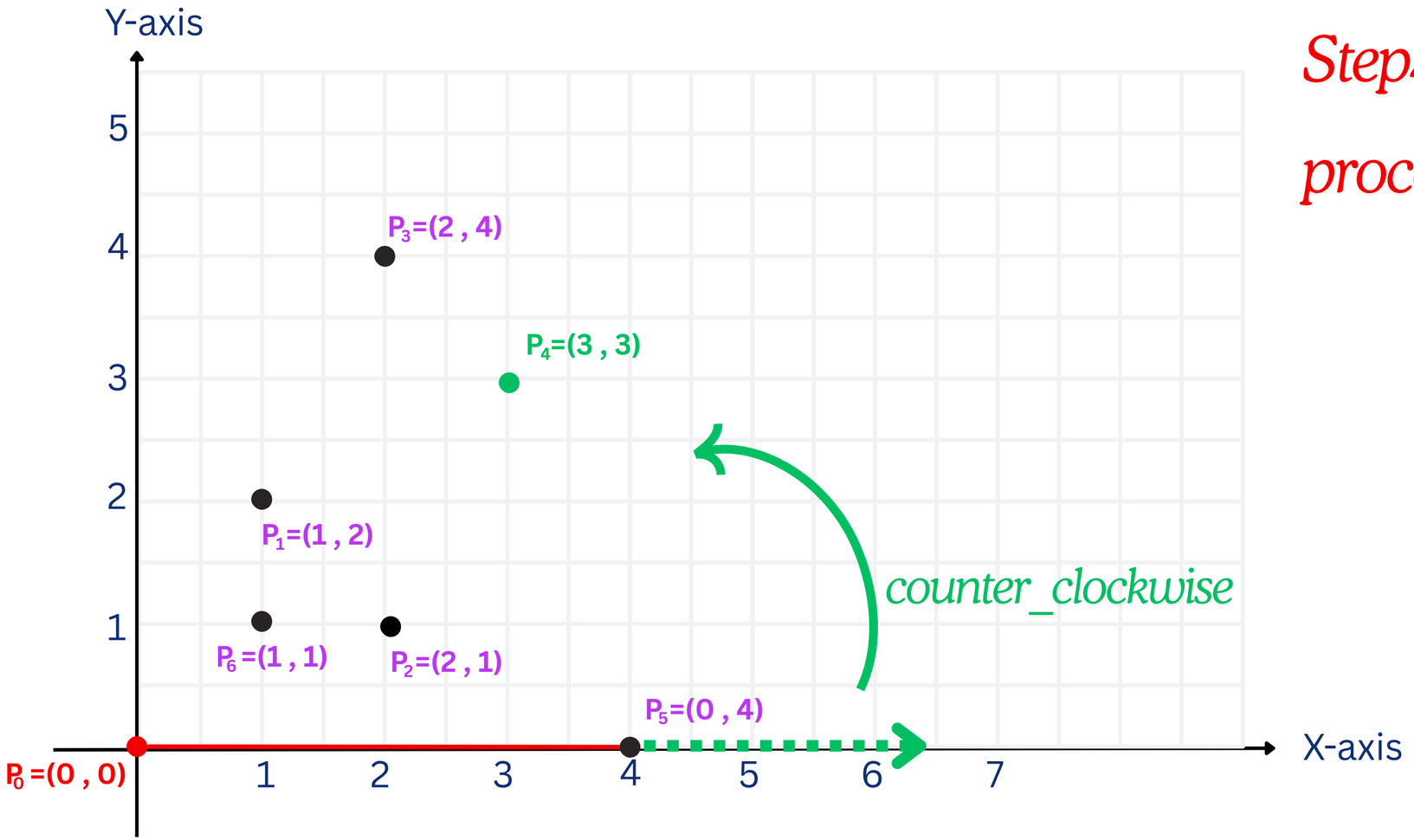
Step4 : build the hull

process P4 (3 , 3)

Stack



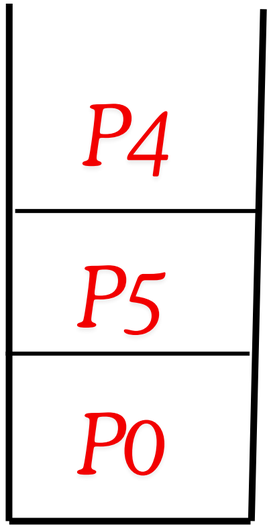
Finding the Convex Hull



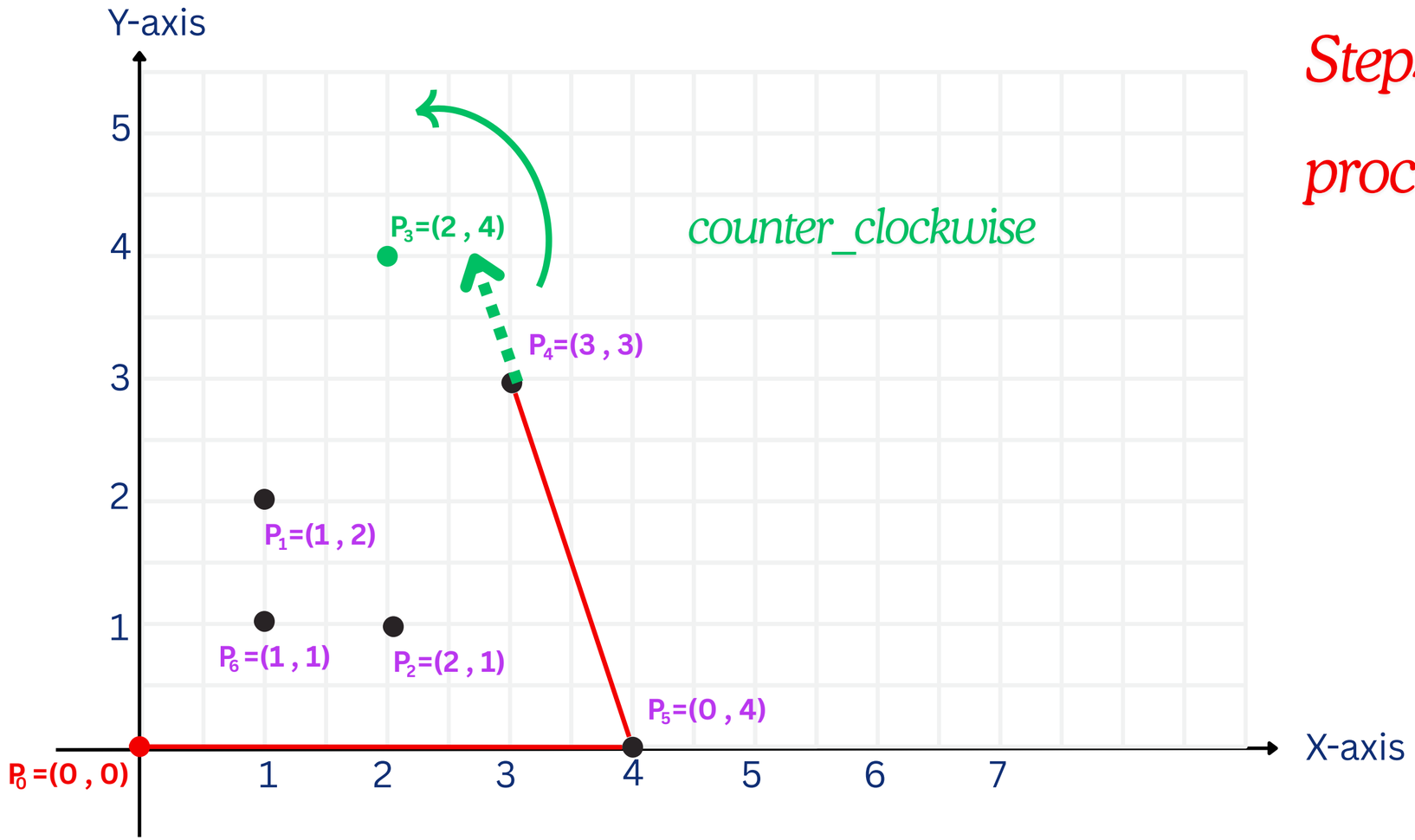
Step4 : build the hull

process P4 (3 , 3)

Stack



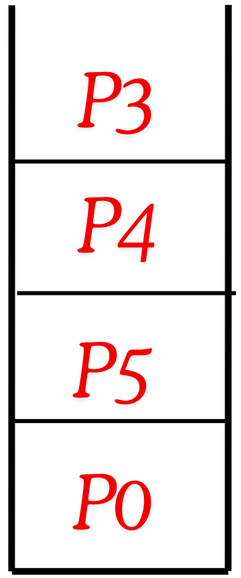
Finding the Convex Hull



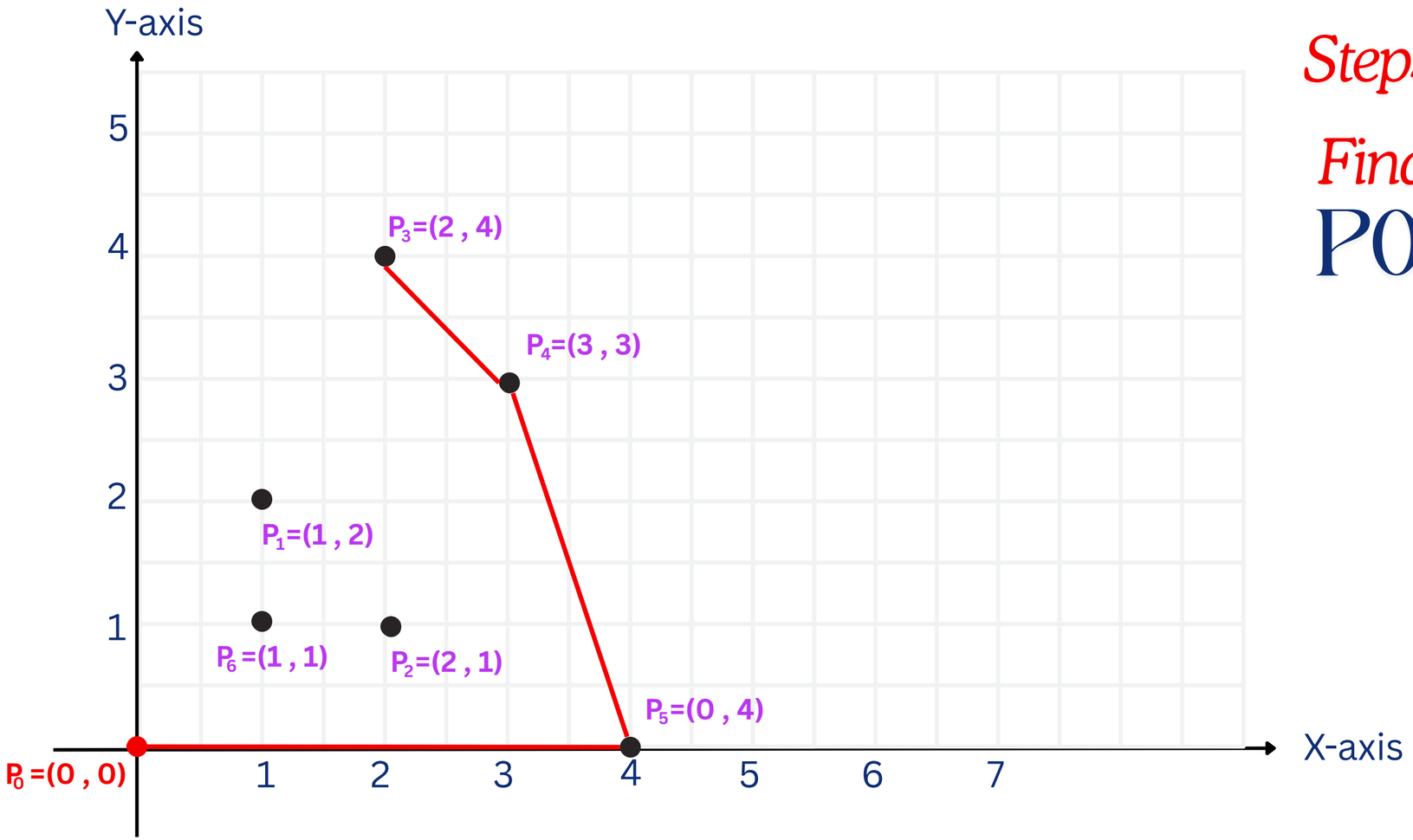
Step4 : build the hull

process P3 (2 , 4)

Stack



Finding the Convex Hull

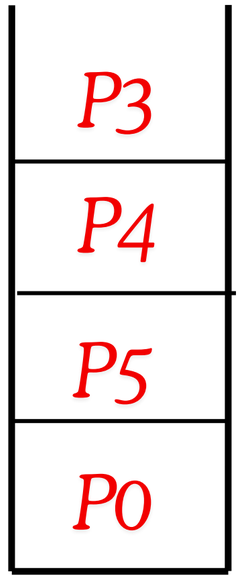


Step4 : build the hull

Final hull

P0->P5->P4->P3->P0

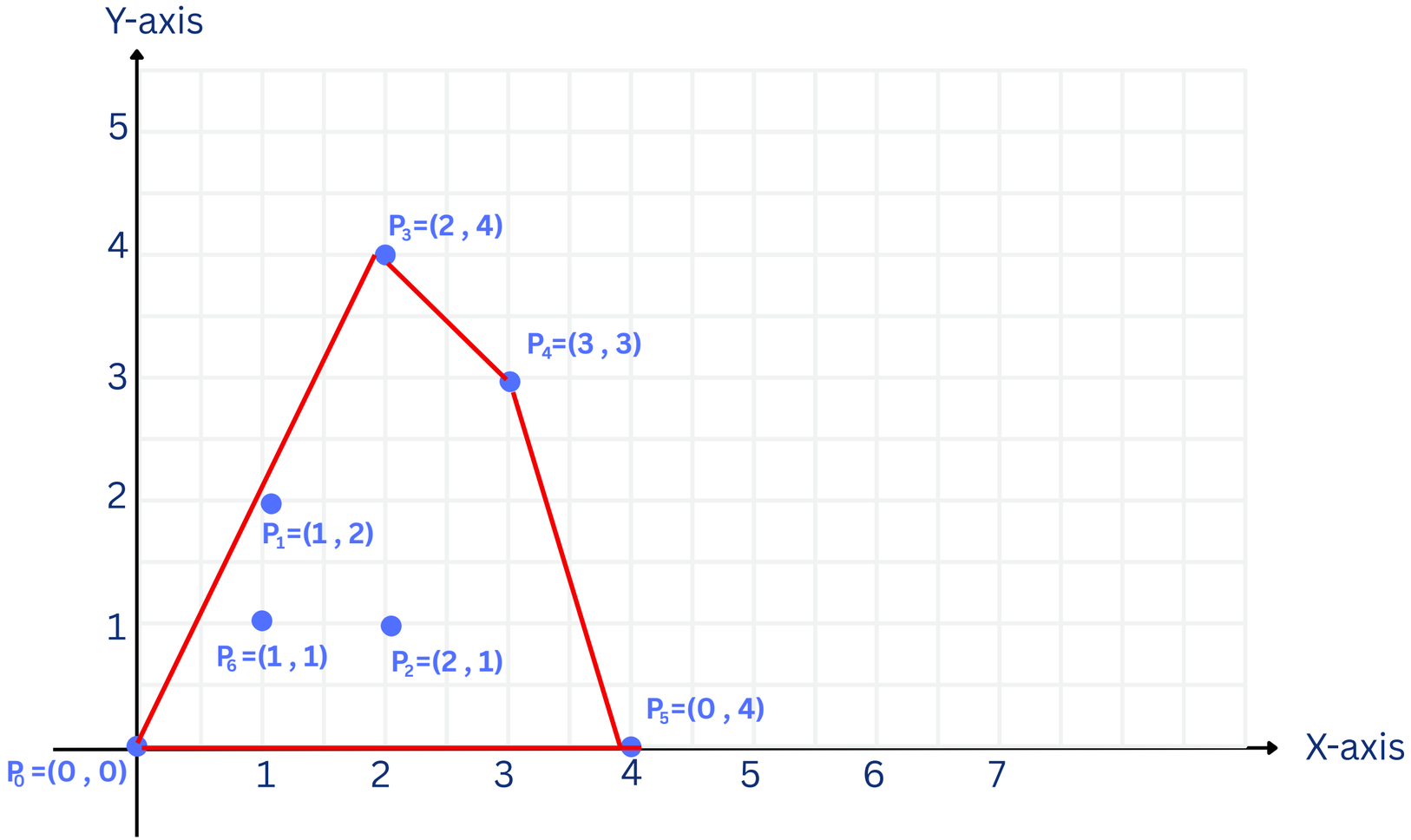
Stack



Finding the Convex Hull

Step4 : build the hull

Final hull P0->P5->P4->P3->P0



CODE BY C++

- **Step 1: Find the Anchor Point**

Find the point with the lowest y-coordinate (and if tied, the leftmost one).

This point is guaranteed to be on the convex hull.

->The function compareY() is use to compare between all points by Y and return the lowest point.

```cpp
// Compare points by Y (and X if tie) O(1)
bool compareY(const Point &p1, const Point &p2)
{
    if (p1.y == p2.y)
        return p1.x < p2.x;
    return p1.y < p2.y;
}
```

->The function findAnchor() is use to Finds the anchor point by selecting the point with the lowest Y-coordinate (and lowest X if there's a tie).

```cpp
// Step 1: Find the anchor point (lowest Y, then lowest X) O(n)
Point findAnchor(vector<Point> &points)
{
    Point anchorPoint = points[0];
    for (const auto &p : points)
    {
        if (compareY(p, anchorPoint))
            anchorPoint = p;
    }
    return anchorPoint;
}
```

->The anchor Calls findAnchor(points) to get the lowest point .

Finds the anchor's position in the list.

Swaps it with the first element so the algorithm always starts from the anchor point.

```
131     // Step 1
132     // Find anchor
133     anchor = findAnchor(points);
134
135     // Move anchor to front
136     int anchorIndex = find(points.begin(), points.end(), anchor) - points.begin();
137     swap(points[0], points[anchorIndex]);
138
```

- **Step 2: Sort by Polar Angle**

Sort all remaining points by their polar angle relative to the anchor point. If two points have the same angle, keep only the one farthest from the anchor.

->The function polarAngle() use to Computes the angle

between each point and the anchor to know its direction

```
51    // calculate polar angle between anchor and point O(1)
52    double polarAngle(const Point &p, const Point &anchor = anchor)
53    {
54        return atan2(p.y - anchor.y, p.x - anchor.x);
55    }
```

->The function sortByAngleAndDistance() sorts all points based on their polar angle relative to the anchor point. If two points have the same angle, it compares their distance from the anchor to keep the closer one first.

```
58  void sortByAngleAndDistance(vector<pair<Point, double>> &pointAngles)
59  {
60      sort(pointAngles.begin(), pointAngles.end(),
61           [](const pair<Point, double> &a, const pair<Point, double> &b)
62           {
63               if (fabs(a.second - b.second) > 1e-9)
64                   return a.second < b.second; // Sort by angle
65               // If angles are equal, sort by distance from anchor
66               return distanceSq(anchor, a.first) < distanceSq(anchor, b.first);
67           });
68  }
```

->The function distanceSq() is calculates distance between two points (without sqrt) to compare which is farther.

```
26  double distanceSq(const Point &a, const Point &b)
27  {
28      return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
29  }
```

->The sortByPolarAngle() function calculates the angle of each anchor point and sorts them in ascending order to build the structure.

```
83  void sortByPolarAngle(vector<Point> &points)
84  {
85      vector<pair<Point, double>> pointAngles;
86
87      for (size_t i = 1; i < points.size(); i++)
88      {
89          double angle = polarAngle(points[i], anchor);
90          pointAngles.push_back({points[i], angle});
91      }
92
93      // Sort
94      sortByAngleAndDistance(pointAngles);
```

->After sorting, if two or more points share the same polar angle from the anchor, we keep only the farthest one.

This prevents overlapping directions and ensures each angle is represented by a single outermost point.

```
97      vector<Point> filtered;
98      filtered.push_back(points[0]);
99
100     for (size_t i = 0; i < pointAngles.size(); i++)
101     {
102         // If next angle is same, keep the farthest one
103         if (i < pointAngles.size() - 1 && fabs(pointAngles[i].second - pointAngles[i + 1].second) < 1e-9)
104         {
105             double d1 = distanceSq(anchor, pointAngles[i].first);
106             double d2 = distanceSq(anchor, pointAngles[i + 1].first);
107             if (d1 > d2)
108                 filtered.push_back(pointAngles[i].first);
109             else
110                 filtered.push_back(pointAngles[i + 1].first);
111             i++; // skip next because we already compared
112         }
113         else
114         {
115             filtered.push_back(pointAngles[i].first);
116         }
117     }
118     points = filtered;
119 }
```

->Sorts all points around the anchor by their polar angle for hull construction.

```
141     sortByPolarAngle(points);
```

- **Step 3: Build the Hull**

Use a stack to build the convex hull. Process points in sorted order, and for each point, check if moving to it creates a clockwise turn (which would make the hull concave).

->Checks if three points make a left turn, right turn, or lie on the same line.

```
31  // Determine orientation of ordered triplet (p, q, r)
32  // Returns:
33  // 0 -> collinear
34  // 1 -> clockwise
35  // 2 -> counter-clockwise
36  int orientation(const Point &p, const Point &q, const Point &r)
37  {
38      double val = (q.x - p.x) * (r.y - p.y) -
39                   (q.y - p.y) * (r.x - p.x);
40
41      if (fabs(val) < 1e-9)
42          return 0;           // collinear
43
44      return (val > 0) ? 2 : 1; // CCW -> 2, CW -> 1
45  }
```

->Uses a stack to keep points that form the outer boundary by checking turn directions.

```
143     // Step 3
144     // Use stack to build hull
145     stack<Point> hull;
146     hull.push(points[0]);
147     hull.push(points[1]);
148
149     for (int i = 2; i < points.size(); i++)
150     {
151         while (hull.size() > 1)
152         {
153             Point top = hull.top();
154             hull.pop();
155             Point nextToTop = hull.top();
156
157             if (orientation(nextToTop, top, points[i]) == 2)
158             { // counter-clockwise
159                 hull.push(top);
160                 break;
161             }
162         }
163         hull.push(points[i]);
164     }
```

->After building the hull, the points stored in the stack are moved into a vector to form the final convex hull.

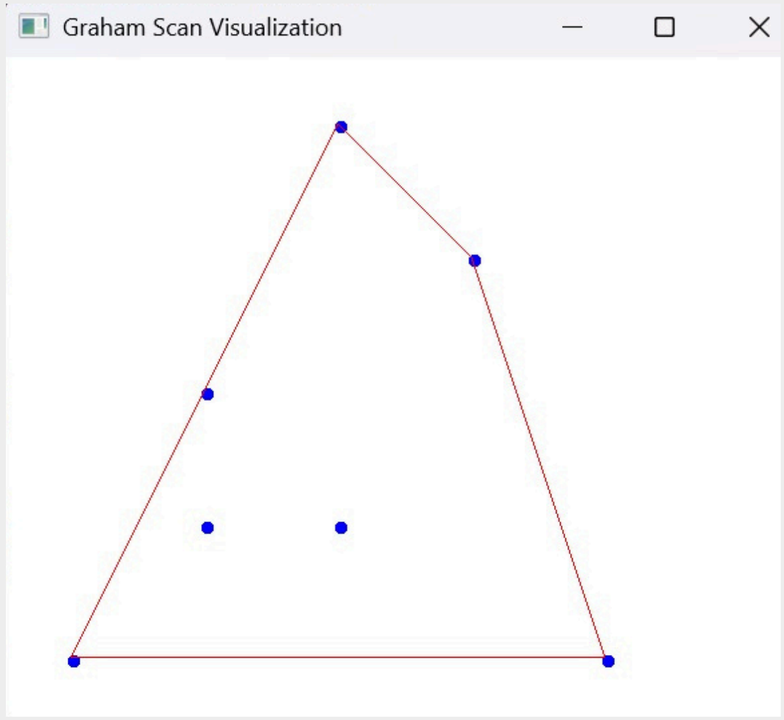
The vector is then reversed to restore the correct order of the hull points.

```
166     // Convert stack to vector for output
167     vector<Point> convexHull;
168     while (!hull.empty())
169     {
170         convexHull.push_back(hull.top());
171         hull.pop();
172     }
173     reverse(convexHull.begin(), convexHull.end());
174     return convexHull;
175 }
```

->Test the code with the given question.

```
177  int main()
178  {
179      vector<Point> points = {
180          {0, 0}, {1, 2}, {2, 1}, {2, 4}, {3, 3}, {4, 0}, {1, 1}};
181
182      vector<Point> hull = grahamScan(points);
183
184      cout << "Convex Hull Points (in order):\n";
185      for (auto &p : hull)
186      {
187          cout << "(" << p.x << ", " << p.y << ")\n";
188      }
```

```
Convex Hull Points (in order):
(0, 0)
(4, 0)
(3, 3)
(2, 4)
```



TIME COMPLEXITY

- Time Complexity of Graham Scan Algorithm
- Finding the anchor point (lowest Y-coordinate): O(n)
- Sorting the points by polar angle: O(n log n)
- Constructing the convex hull using a stack: O(n)

Total Time Complexity: O(n log n)

- The sorting step is the most time-consuming part, which dominates the overall complexity.

THANK YOU