Task Planine

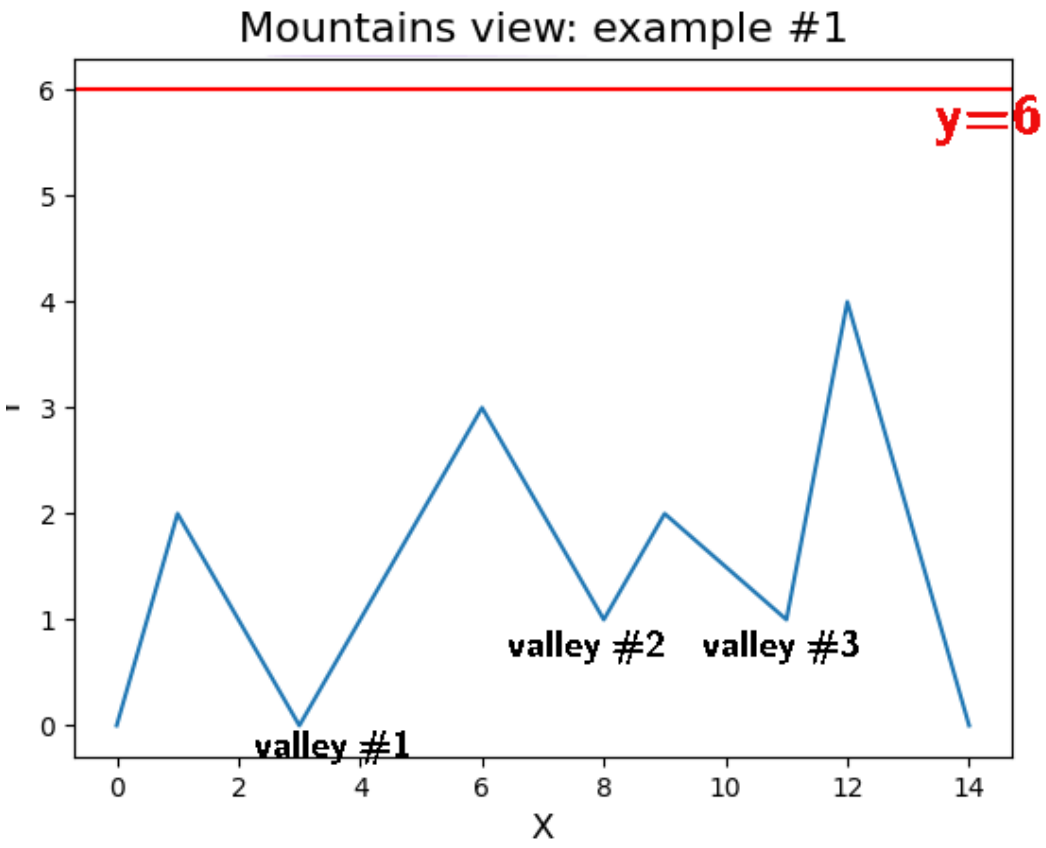
Observation

Let’s go, first, throw the example #1 from the problem description, in order to get the main idea behind the solution.

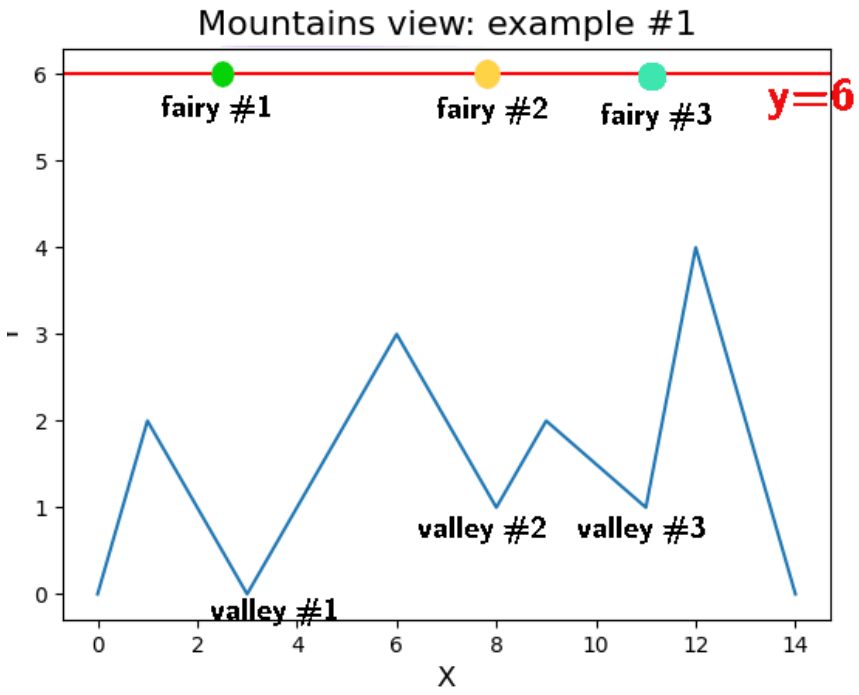
input

```
9 6
0 0
1 2
3 0
6 3
8 1
9 2
11 1
12 4
14 0
```

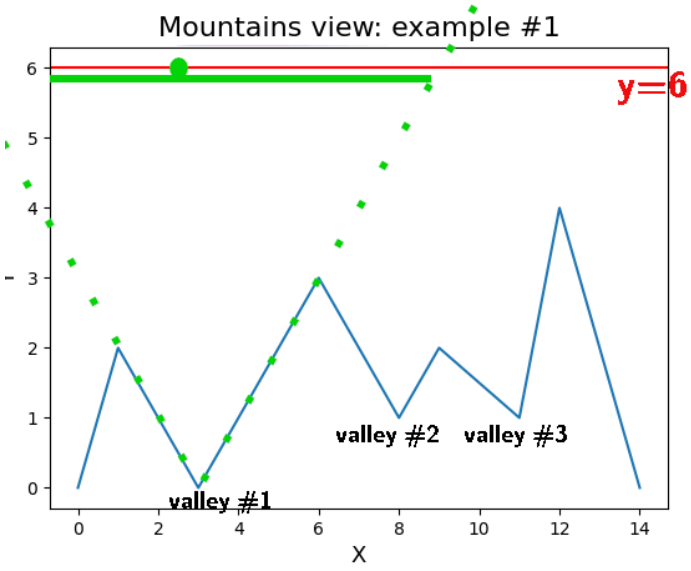
Graphical representation



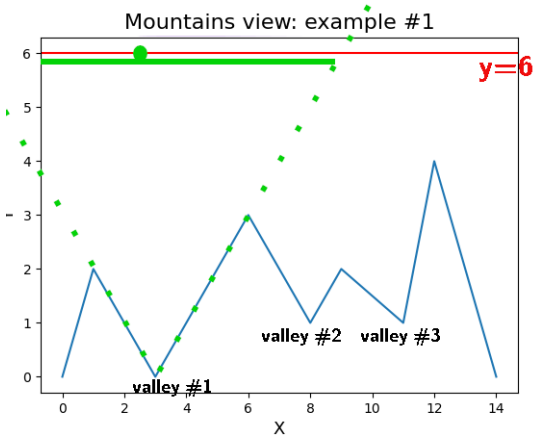
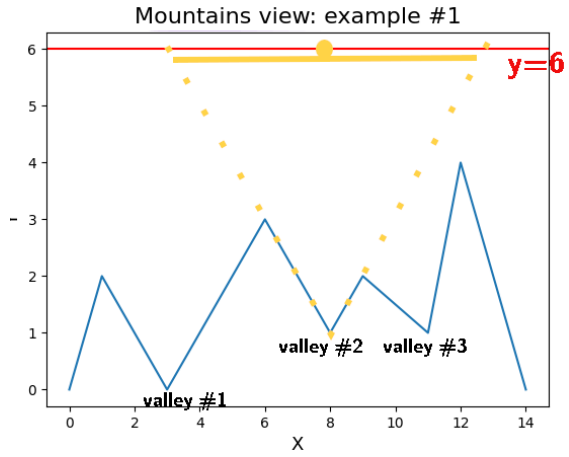
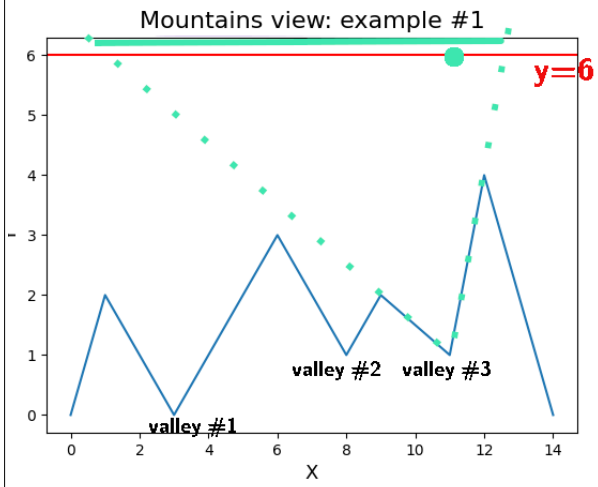
If we,naively, put a fairy for each valley, we gonna have three fairies:



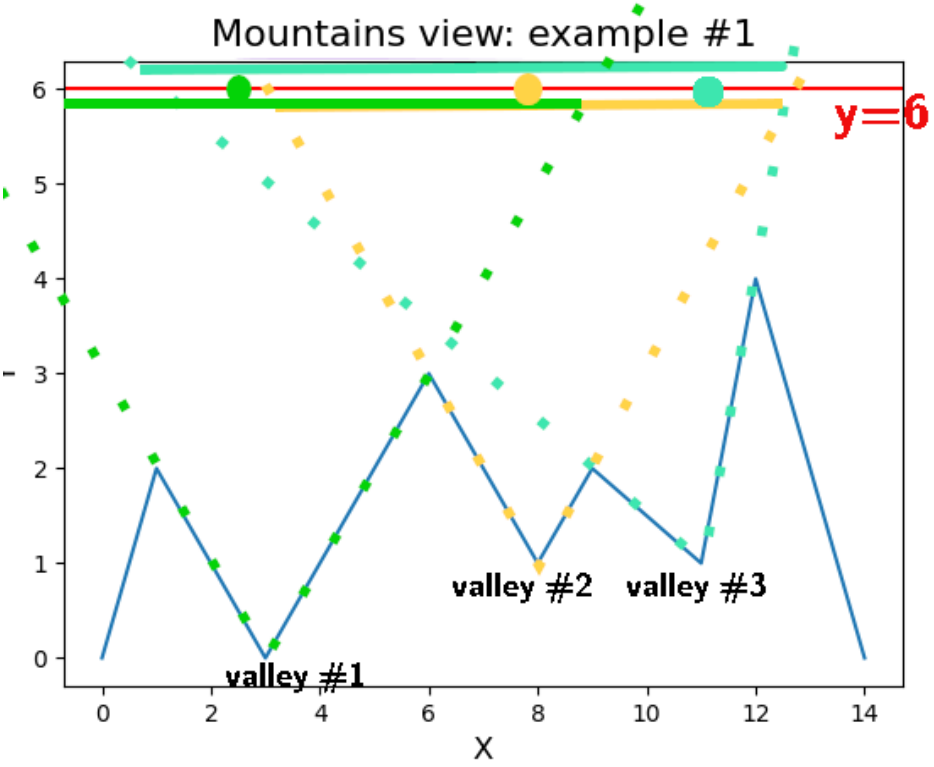
Too know which part of the fairies line y from where a fairy can illuminate its valley, we draw a two lines from each side of the valley, to the line of fairies y , we get a segment on y :

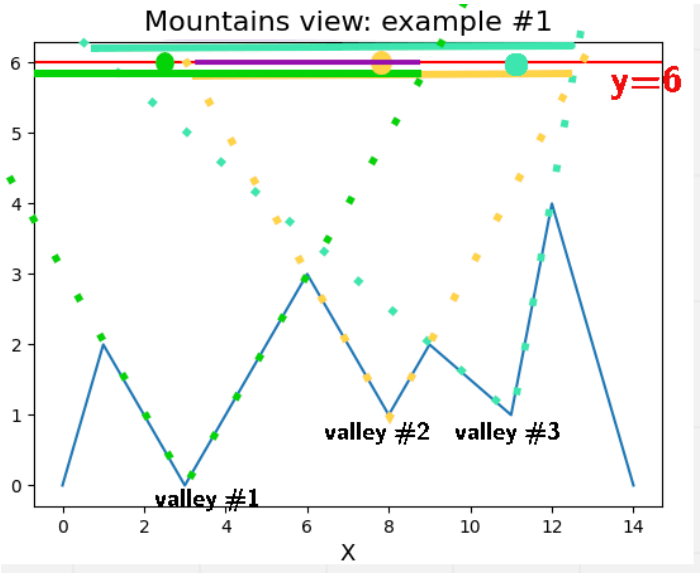


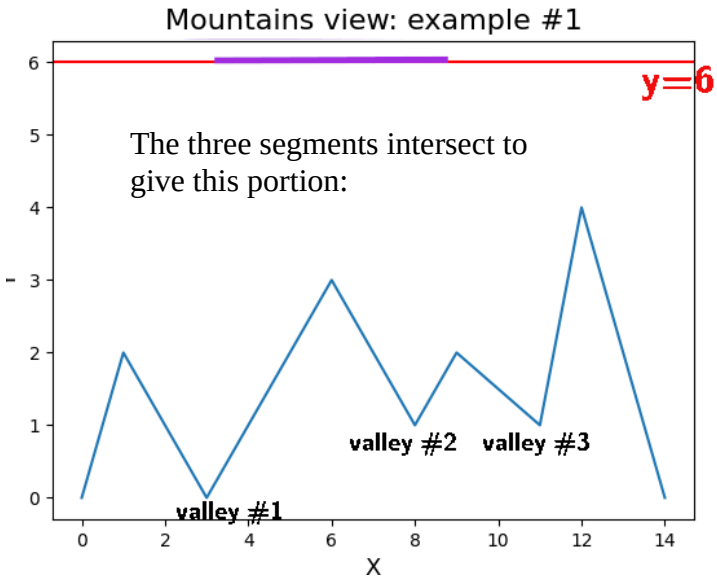
What does that mean?

Fairy #1 illuminate valley #1	Fairy #2 illuminate valley #2	Fairy #3 illuminate valley #3
		
All positions of the fairy #1 on the segment (green segment) are good to illuminate valley #1	All positions of the fairy #2 on the segment (yellow segment) are good to illuminate valley #2	All positions of the fairy #3 on the segment (cyan segment) are good to illuminate valley #3

If we put all this together:

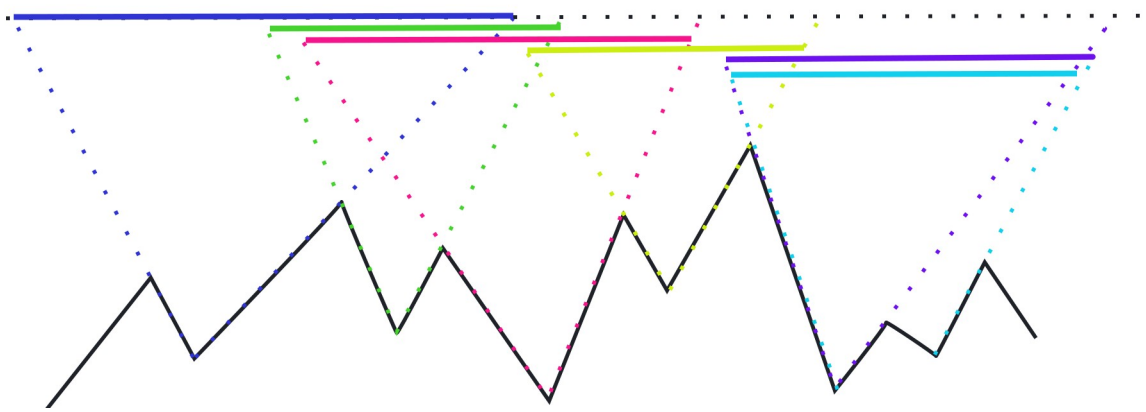




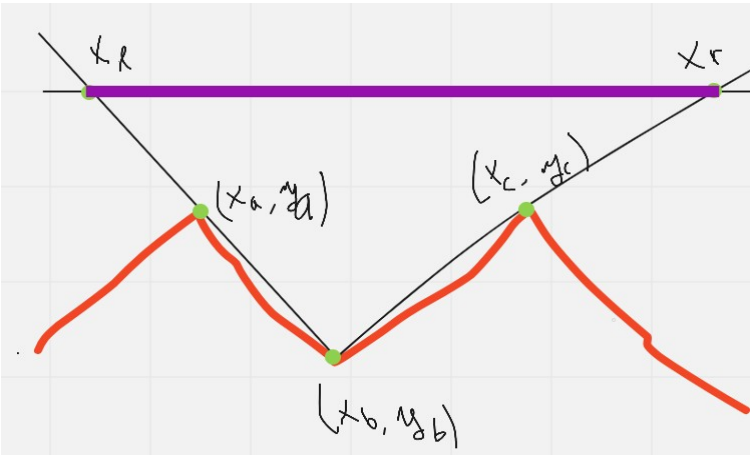


if a fairy is in that segment (purple one), it will illuminate all the three valleys. So the answer here, we need only one fairy.

In general case, our first subtask is to determine all segments for all valleys where a fairy can illuminate that a valley from any position on its segment.



These segments are determined by the intersection of the lines of each side of the valley and the fairies line y .



x_l : left x-coordinate of the segment

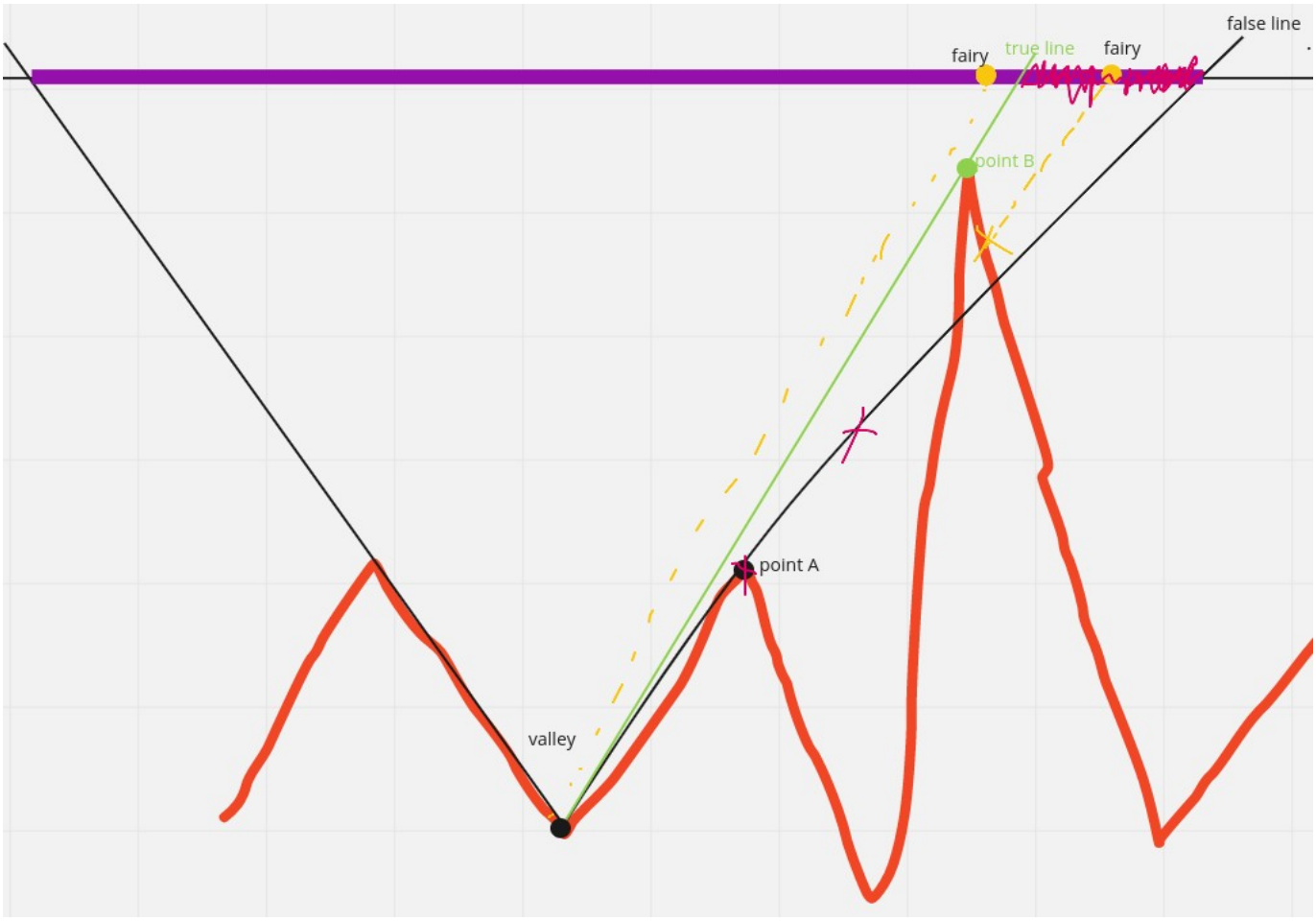
x_r : right x-coordinate of the segment

(x_a , y_a): first upper point of the valley

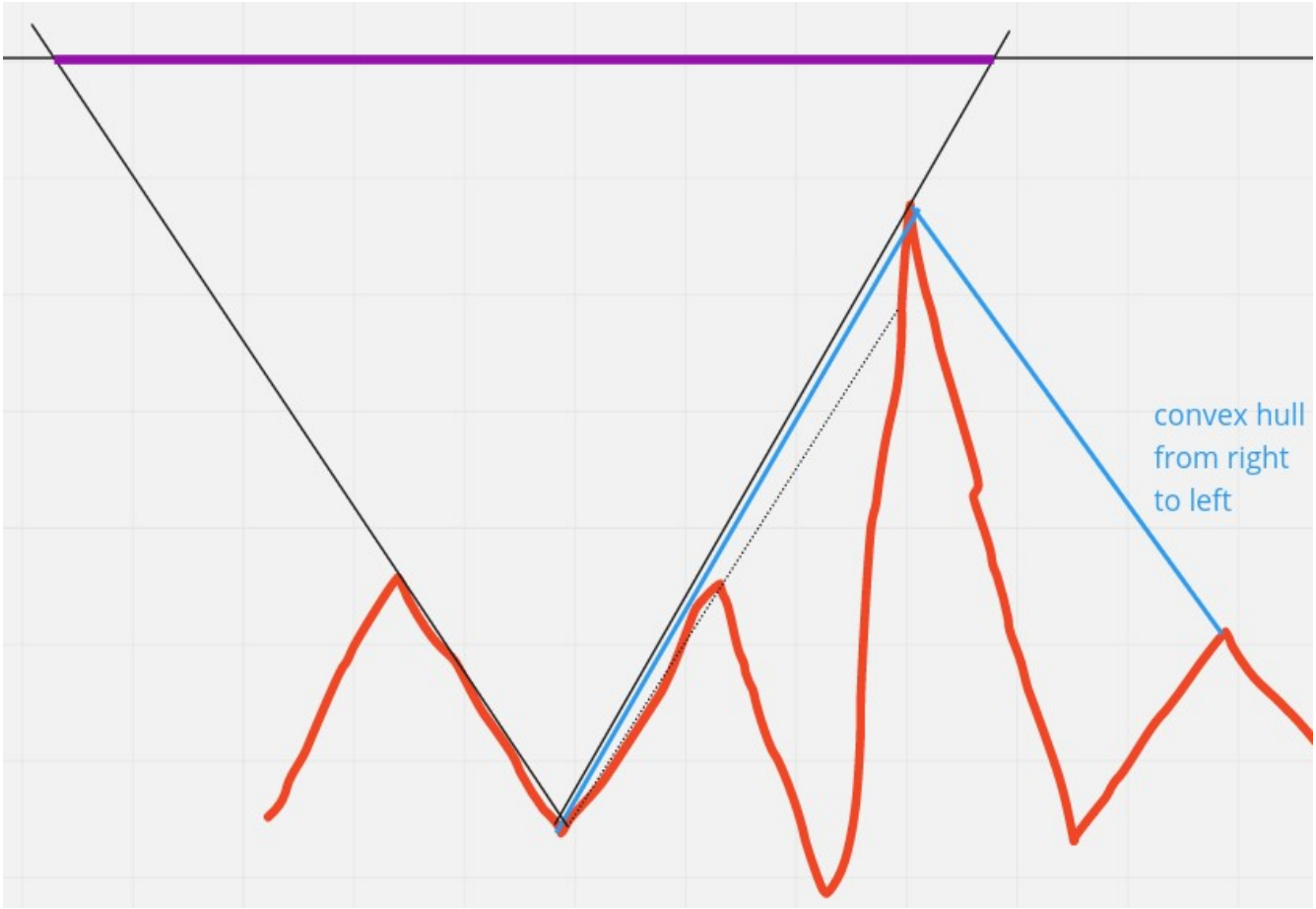
(x_b , y_b): the bottom point of the valley

(x_c , y_c): second upper point of the valley

But there is an issue, when the mountain tops have different heights, it can happen that a non-adjacent mountain top "blocks the view".



To obtain to right point of the segment, we must compute the convex hull points from right to left of that current valley:



The left point point of the segment of the current valley is obtained by computed the convex hull from left to right:

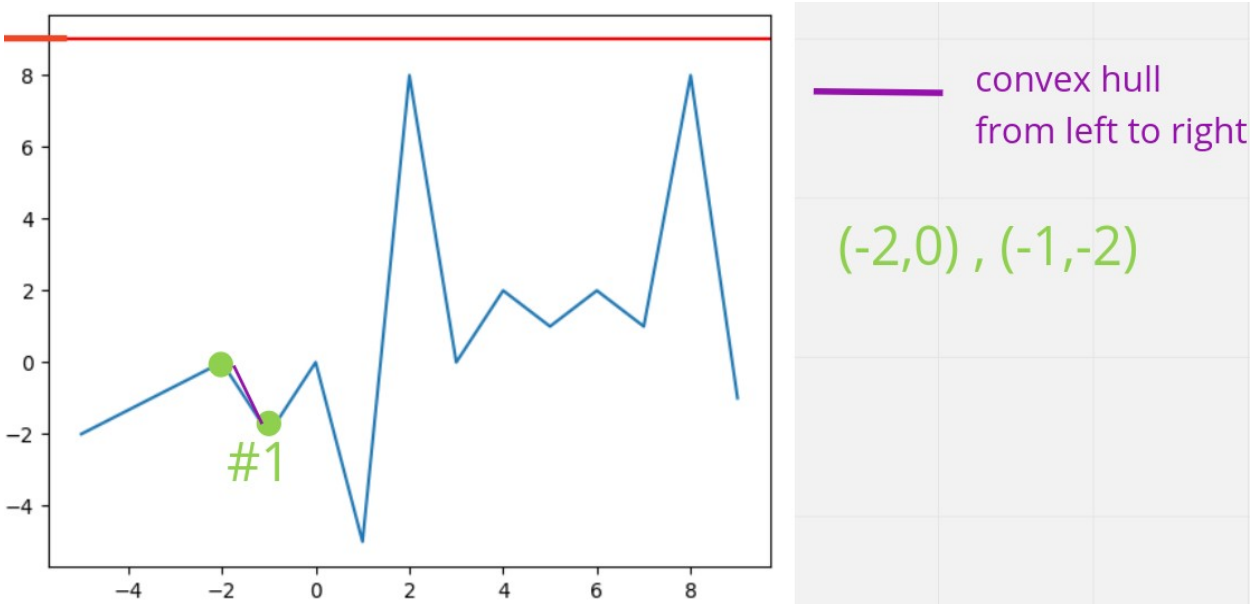
After we find the segments, we have the well-known problem: Compute the minimum number of points covering a set of segments.

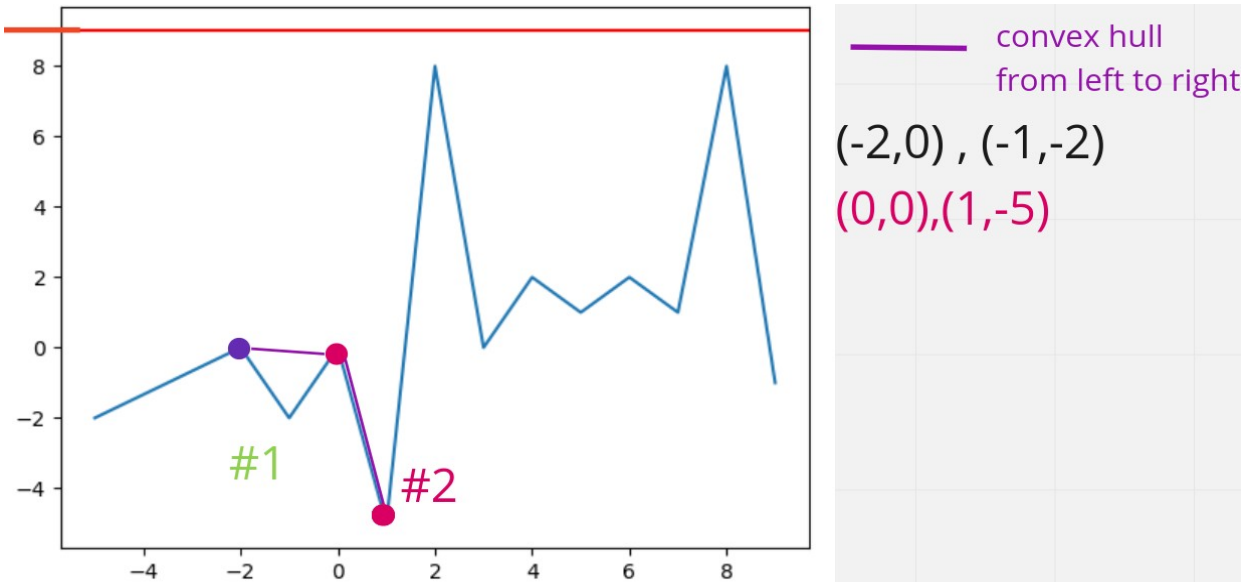
Let’s take an example and see how to do:

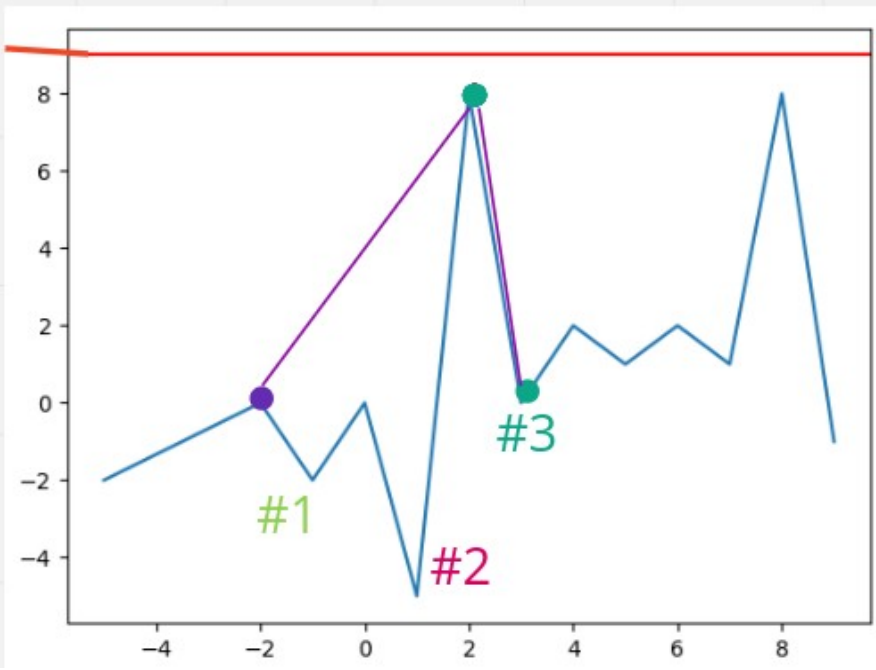
13 9
-5 -2
-2 0
-1 -2
0 0
1 -5
2 8
3 0
4 2
5 1
6 2
7 1
8 8
9 -1

Sub task #1

a) compute the left side points by drawing the upper convex hull from left to right:

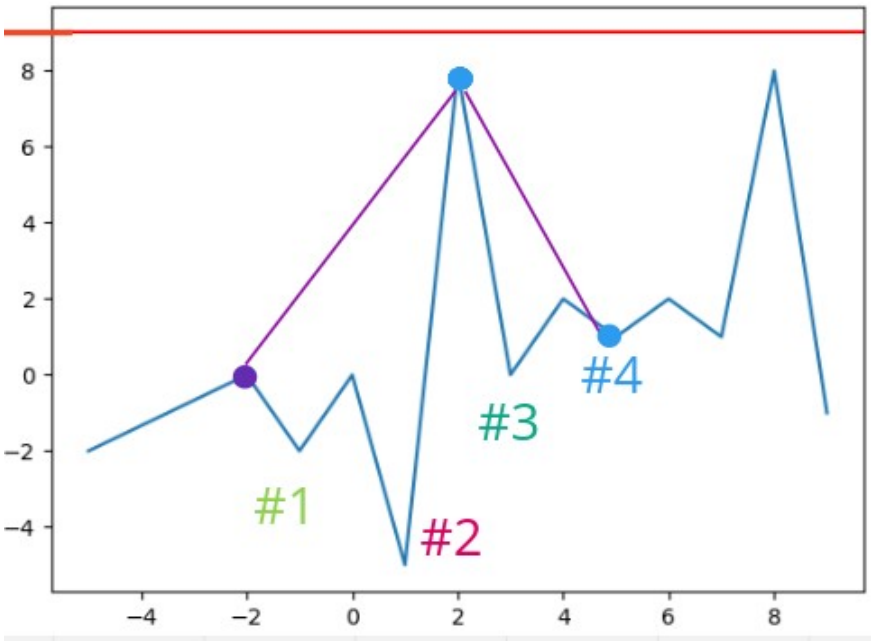






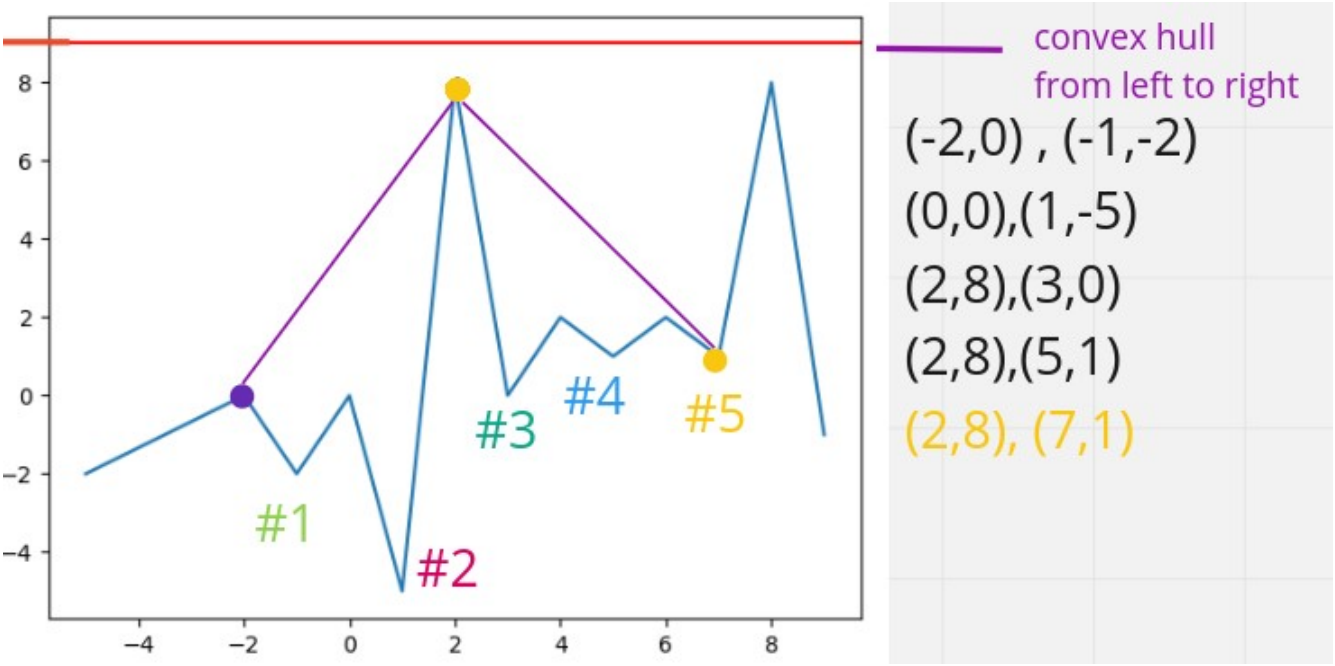
convex hull
from left to right

(-2,0) , (-1,-2)
(0,0),(1,-5)
(2,8),(3,0)

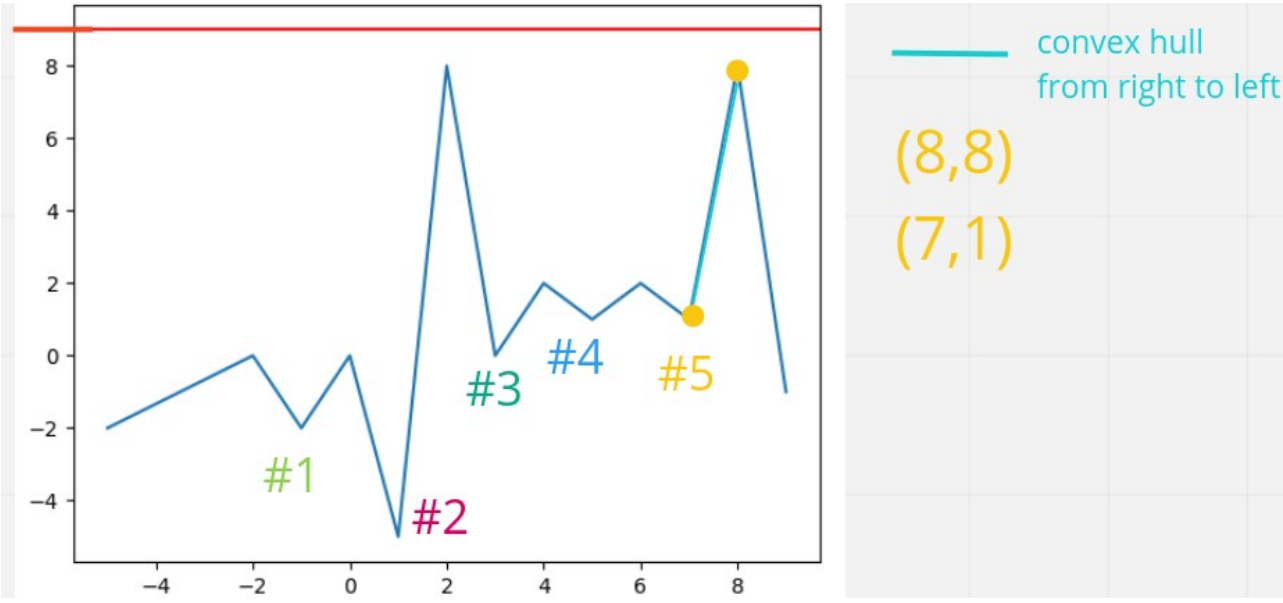


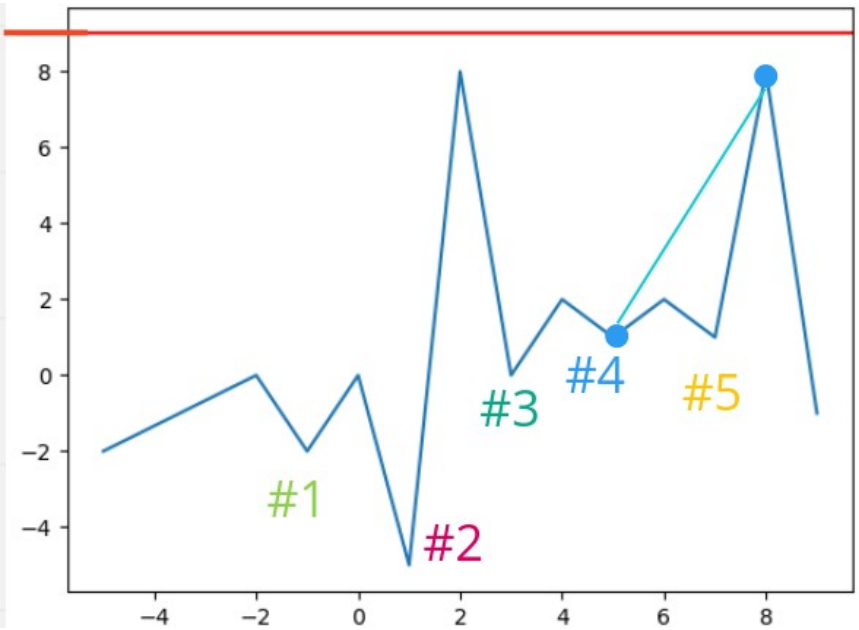
convex hull
from left to right

(-2,0) , (-1,-2)
(0,0),(1,-5)
(2,8),(3,0)
(2,8),(5,1)



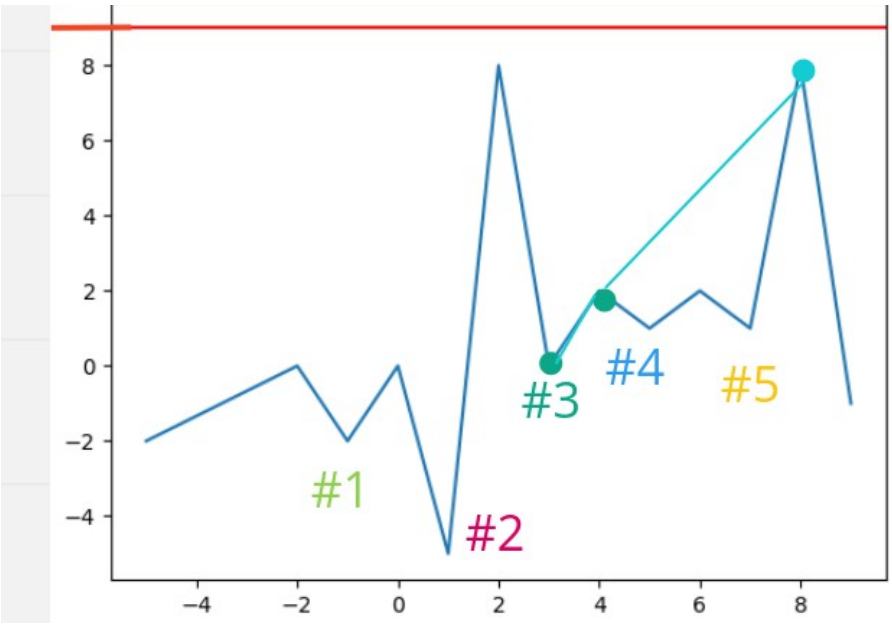
b) compute the right side points by drawing the upper convex hull from right to left:





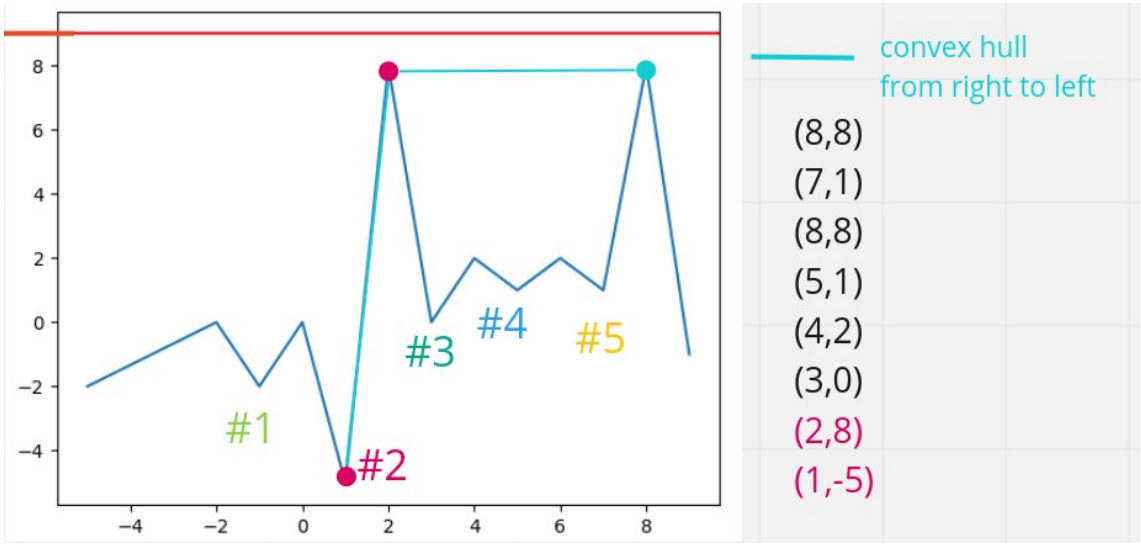
convex hull
from right to left

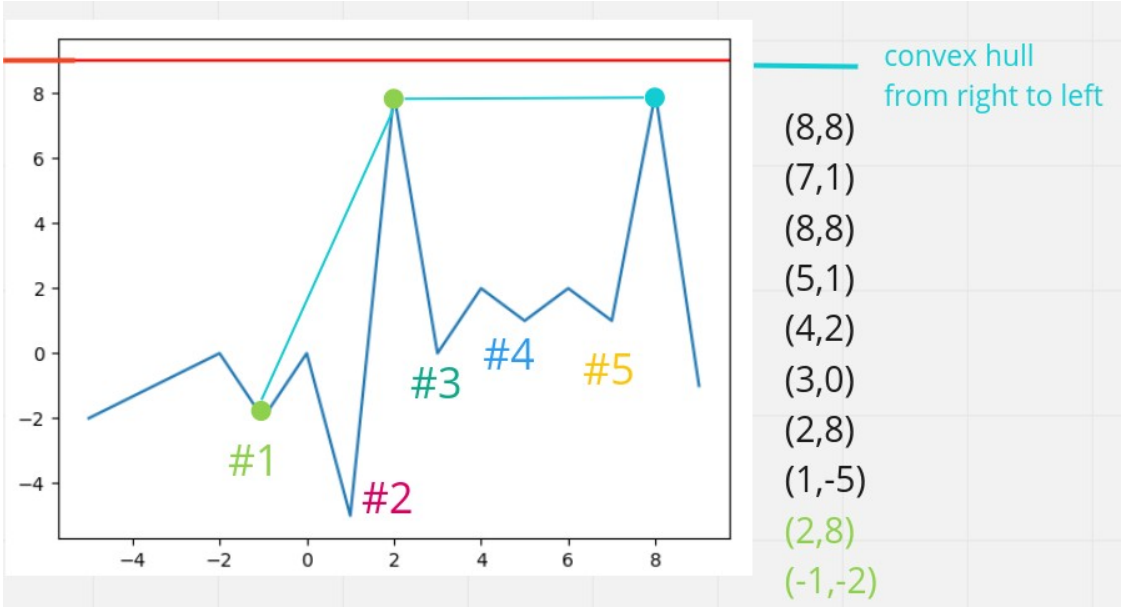
(8,8)
(7,1)
(8,8)
(5,1)



convex hull
from right to left

(8,8)
(7,1)
(8,8)
(5,1)
(4,2)
(3,0)





The time complexity of this subtask is $O(n+(n\log n))$ (n is the size of the inputs), and the space complexity is $O(2\times \text{nb\_valleys})$

$n\log n$, because we need to sort the input

This subtask return two arrays: one contains the top left point and the bottom point of each valley, the other contains the top right point and the bottom point of each valley.

Sub task #2

compute the intersection two points between the both side lines of the current valley and the fairies line y , these two points are the segment $[x_{\text{left}}, x_{\text{right}}]$ of the current valley

Here we manually compute the segment of valley #1:

- points of valley #1: (-2,0) , (-1,-2) , (2,8)
- fairies line: yf = 9

compute x_{left} **:**

compute the slope a of the left line of the valley:

$$\begin{cases} 0 = -2a + b \\ -2 = -a + b \end{cases} \Rightarrow \quad 2 = -a \quad \Rightarrow \quad a = -2$$

compute the constant b of the left line of the valley:

$$b = -4$$

we have the affine function, compute x_{left}:

$$yf = 9 = -2x_{\text{left}} - 4 \quad \Rightarrow \quad x_{\text{left}} = \frac{13}{-2} = -6.5$$

compute x_{right} **:**

compute the slope a of the right line of the valley:

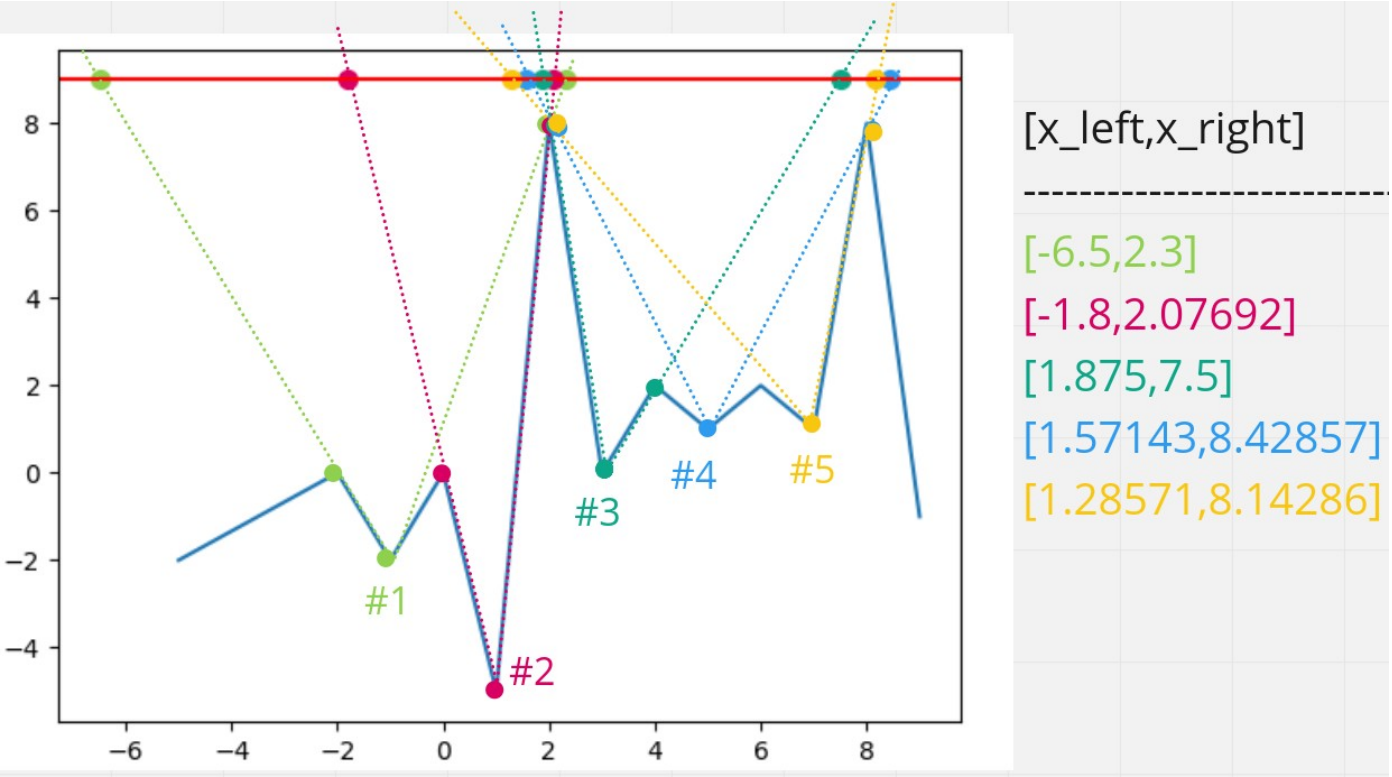
$$\begin{cases} 8 = 2a + b \\ -2 = -a + b \end{cases} \Rightarrow \quad 10 = 3a \quad \Rightarrow \quad a = \frac{10}{3}$$

compute the constant b of the right line of the valley:

$$b = \frac{4}{3}$$

we have the affine function, compute x_{right}:

$$yf = 9 = \frac{10}{3}x_{\text{right}} + \frac{4}{3} \quad \Rightarrow \quad 27 = 10x_{\text{right}} + 4 \quad \Rightarrow \quad x_{\text{right}} = 2.3$$

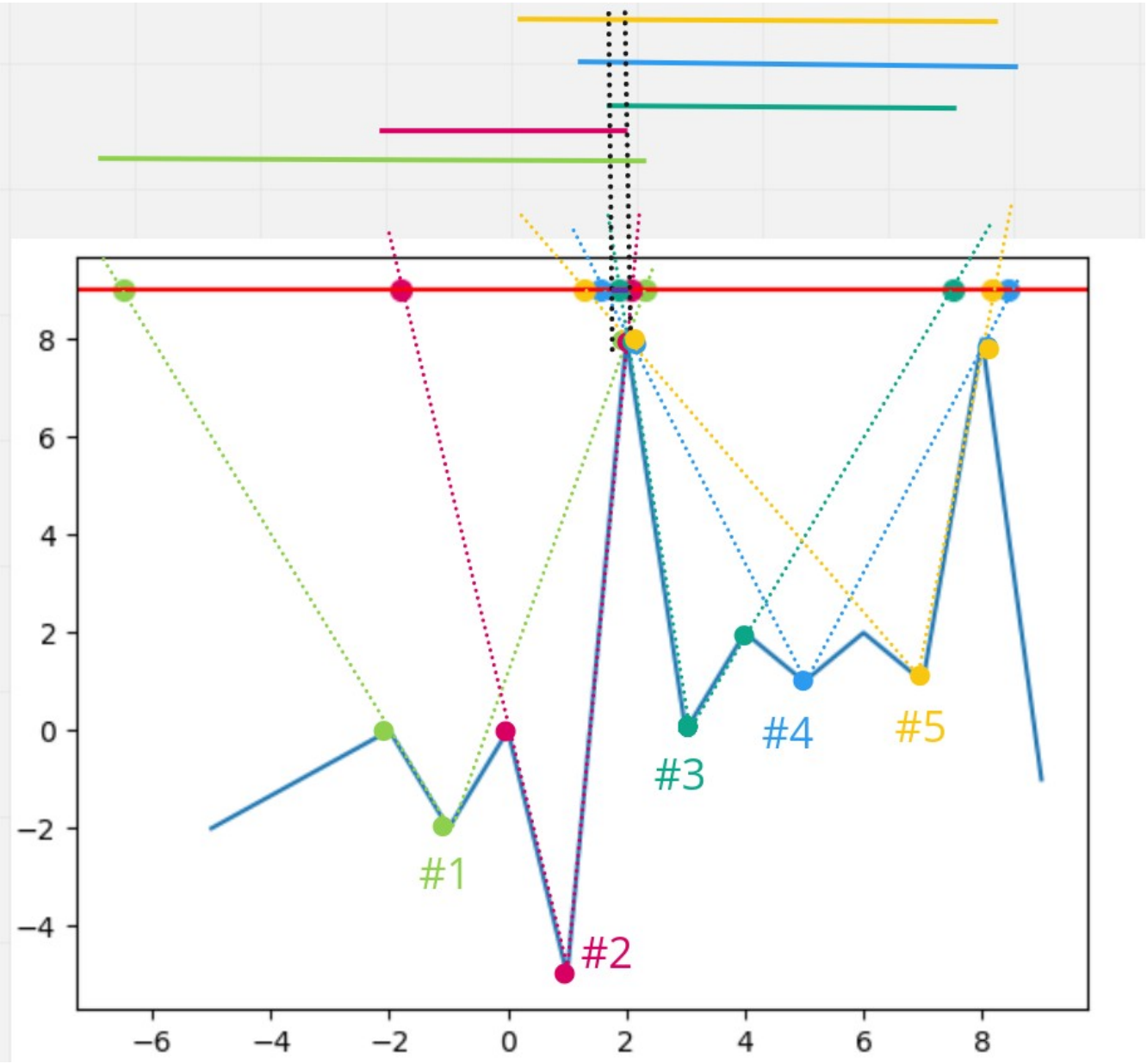


The time and space complexity of subtask2 are $O(1)$

At the end of this task, we get a set points, each one represents the start and end of segment.

Sub task #3

Compute the minimum number of points covering a set of segments:



We just need **one fairy** at any position on the result segment to illuminate all the path.

The time complexity for the subtask #3 is
$O(\text{nb\_segments} + (\text{nb\_segments} \times \log \text{nb\_segments}))$

The space complexity is $O(1)$

At the end of this subtask, we have the final answer.

Skills that we need

As you see we used the **convex hull** to compute the points of each valley. But, to know how draw or determine a convex hull, we need to know about a vector concept which is the **cross product**.

We need also some math to determine the segments (**affine function**, **intersection BTW two lines**)

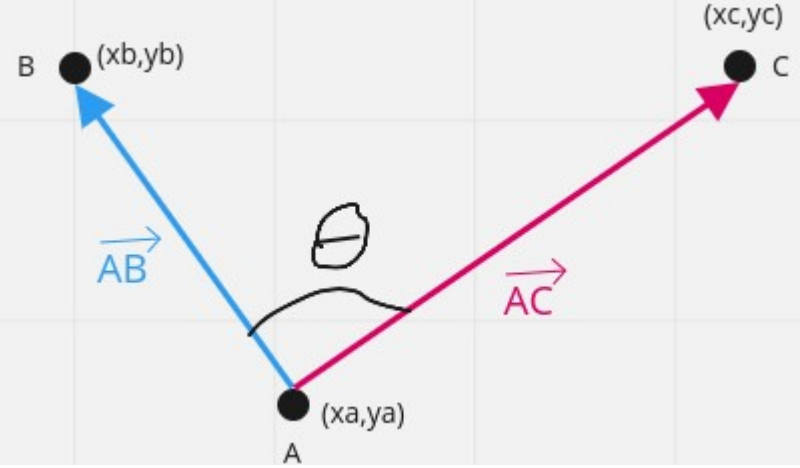
Also, he have to know how to resolve **the minimum points that cover a set of segments**.

Cross product and orientation

Before seeing what is a convex hull, we have to learn about very important concepts in vectors, which are cross product and orientation.

Cross product

If we have three points $A(x_a, y_a)$, $B(x_b, y_b)$ and $C(x_c, y_c)$ in 2-D plane:



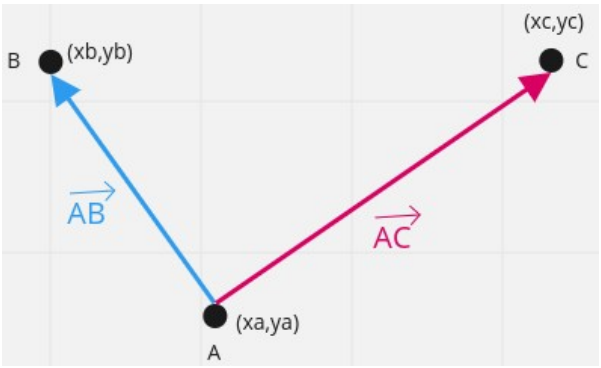
$$cross\ product = \vec{AB} \times \vec{AC} = \begin{vmatrix} (x_c - x_a) & (x_b - x_a) \\ (y_c - y_a) & (y_b - y_a) \end{vmatrix} = (x_c - x_a)(y_b - y_a) - (x_b - x_a)(y_c - y_a)$$

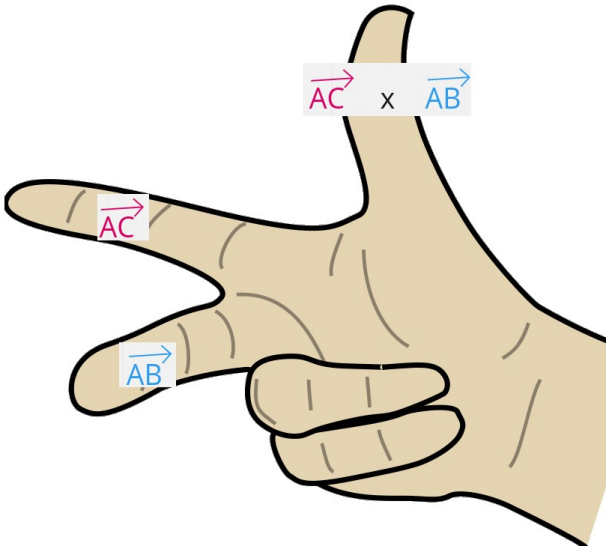
The cross product could be positive, negative or null (all three points in same line)

Trick:

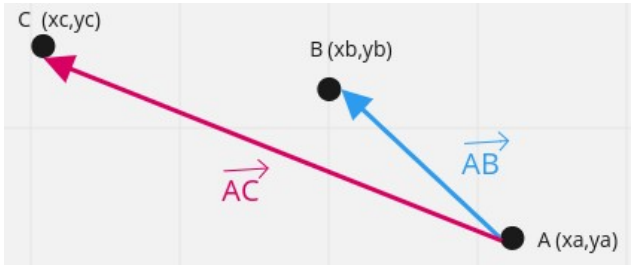
we can use the right hand-Rule to know the sign of the cross product:

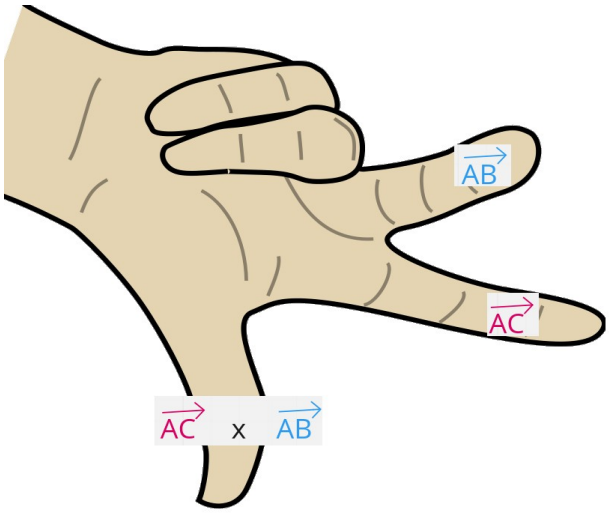
Thumb is up means cross product is positive





Thumb is down means cross product is negative

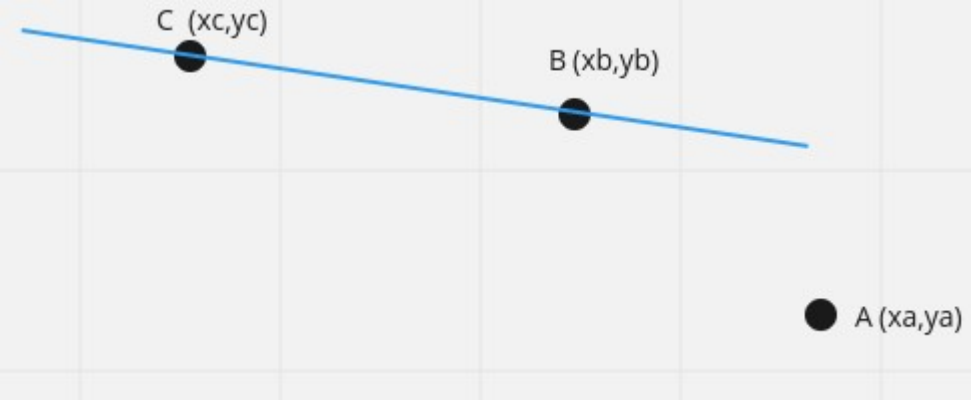




Orientation

The orientation is about to know the orientation of a point by respect of a line.

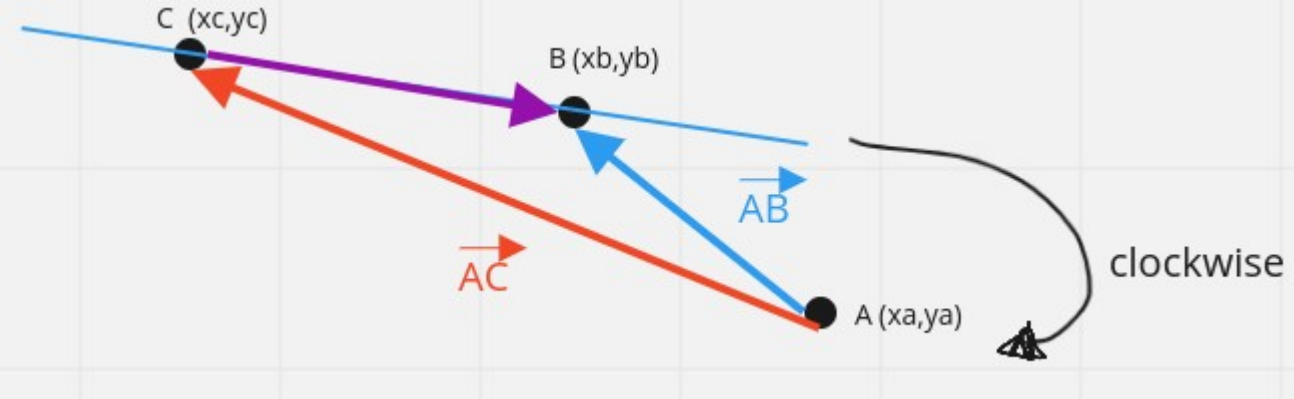
What is the orientation of A by respect of the line (CB) ?

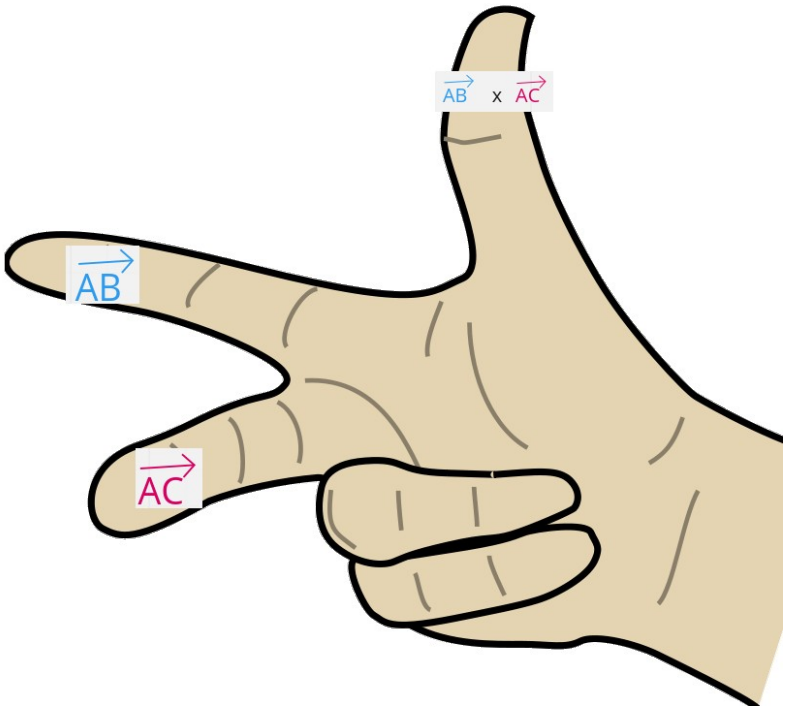


There are two cases:

- If you move from C to B
- if you move from B to C

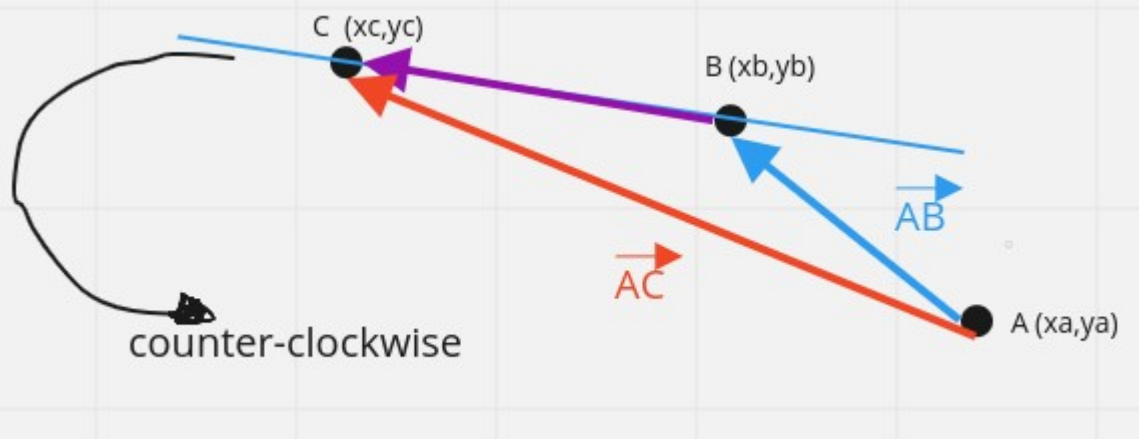
From C to B

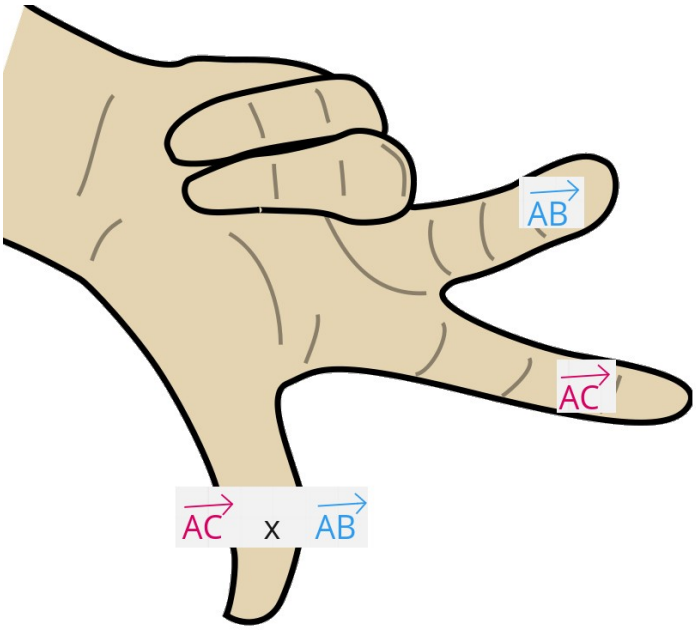




$\vec{AB} \times \vec{AC} > 0$, so the orientation of A by respect of $\vec{CB}$ is positive or clockwise.

From B to C





$\vec{AC} \times \vec{AB} < 0$, so the orientation of A by respect of $\vec{BC}$ is negative or anti-clockwise.

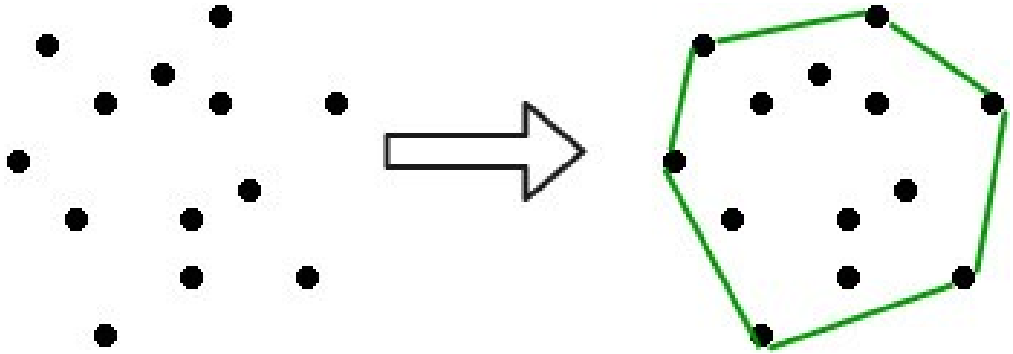
Convex hull?

Given n points in plane (2D or multi-D)

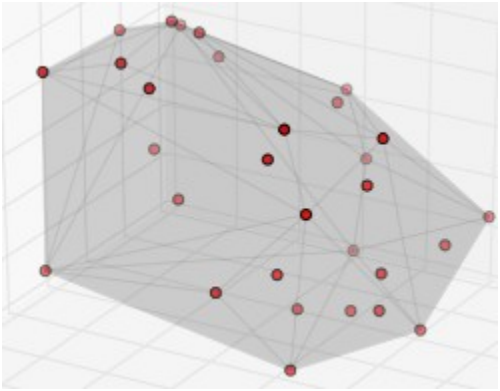
$$S=\{(x_i,y_i)|1\leq i\leq n\}$$

A convex hull is the smallest convex polygon containing all points in S .

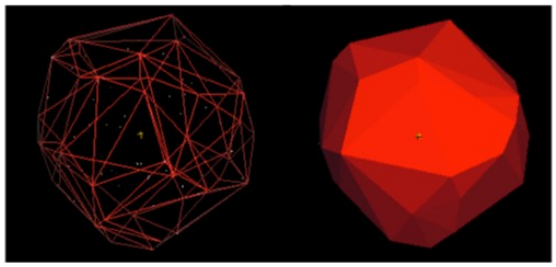
It is represented by a sequence of points on the boundary in clockwise/anti-clockwise order as doubly linked list.



Convex hull in two dimensional plane
Source: https://media.geeksforgeeks.org/wp-content/uploads/Convex_hull_1.jpg



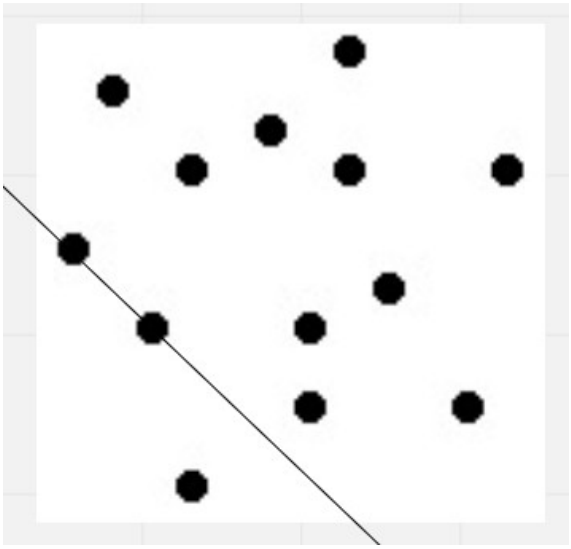
Convex hull in three dimensional plane
Source: https://en.wiktionary.org/wiki/File:Convex_hull_in_3D.svg



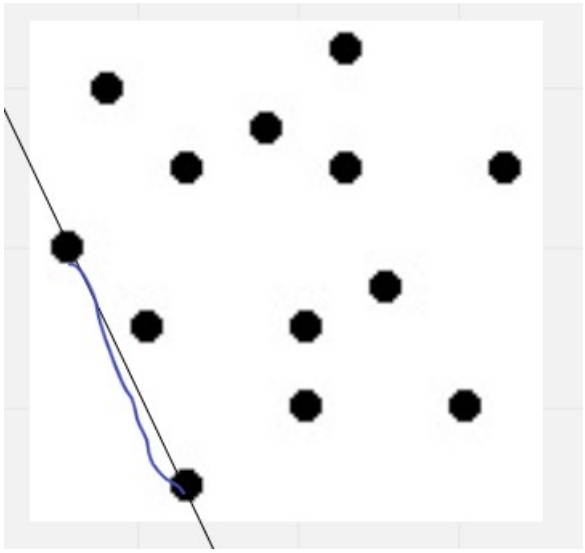
Convex hull in three dimensional plane
Source: https://en.wikipedia.org/wiki/Convex_hull#/media/File:3D_Convex_Hull.tiff

Brute force

1. Take two points
2. Draw a line
3. If all other points are in one side of that line, the two points took in 1. are part of the convex hull



Not all points are in one side of the line



All the points are in one side, the two points are part of the convex hull

```
brute_force(points[]){
  n ← points.size
  if (n ≤ 3) return points;
  for i ∈[0,n-2]{
    for j ∈[i+1,n-1]  {
      if ( are_all_in_one_side(points,points[i],points[j]) ) {
        insert t[i] in S
        insert t[j] in S
      }
  }
  return S
}


are_all_in_one_side(points[], point a, point b){
  n ← points.size
  nb_on_the_same_side ← 0
  nb_on_same_line ← 0

  for each point p in points{
    if (p ≠ a and p ≠ b){
      cp ← cross_product(a,b,p)
      if (cp > 0) nb_on_the_same_side ← nb_on_the_same_side + 1
      else if (cp < 0) nb_on_the_same_side ← nb_on_the_same_side - 1
      else nb_on_same_line ← nb_on_same_line + 1
    }
  }

  return nb_on_the_same_side + nb_on_same_line = n-2 or  nb_on_the_same_side - nb_on_same_line = -(n-2)

}
```

$\vec{ov} \times \vec{ou}$

```
cross_product(point u, point v, point o){

  return (v.x-o.x) * (u.y-o.y) - (u.x-o.x) * (v.y-o.y)

}
```

Graham scan algorithm combined with divide and conquer concept

General concept

The main idea is to build the lower convex hull side by side with the upper convex hull.

Before anything, we have to choose a direction (left to right or right to left)

1. Hulls contain the two first left most points (if left to right direction is chosen)
2. Always take the left most point (if left to right direction is chosen),
3. repeat: check if the actual point can make a valid convex hull: delete de last point of the current hull, if no
4. Add it to the hull, if yes

update the lower convex hull based on 2.

update the upper convex hull based on 2.

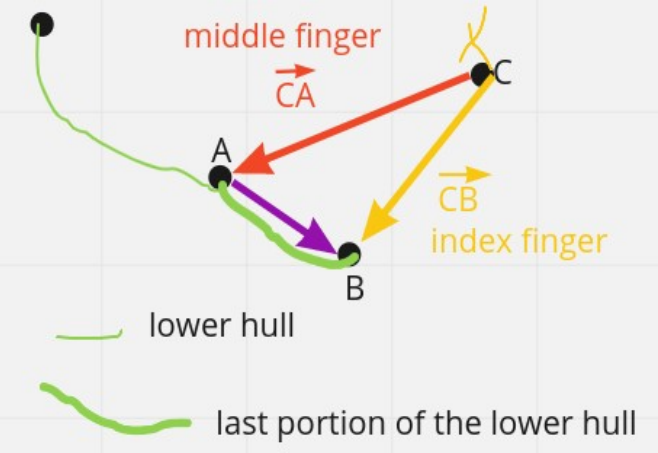
} Divide and conquer

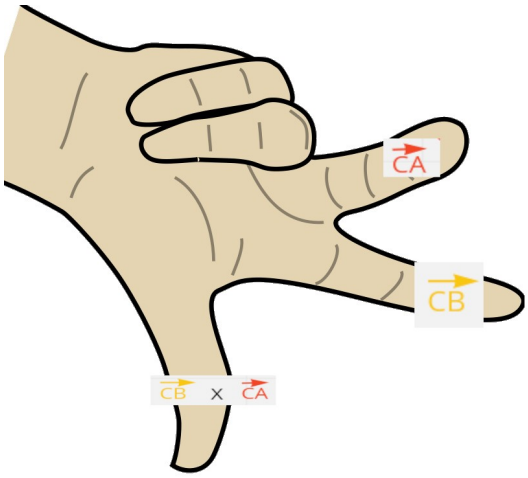
In this section, I assume that you already know about **cross product** and of **orientation**.

Before we start, how to know id a point could be added to the lower or the upper convex hull?

Bu computing its orientation with respect of the last two points of the convex hull (upper and lower)

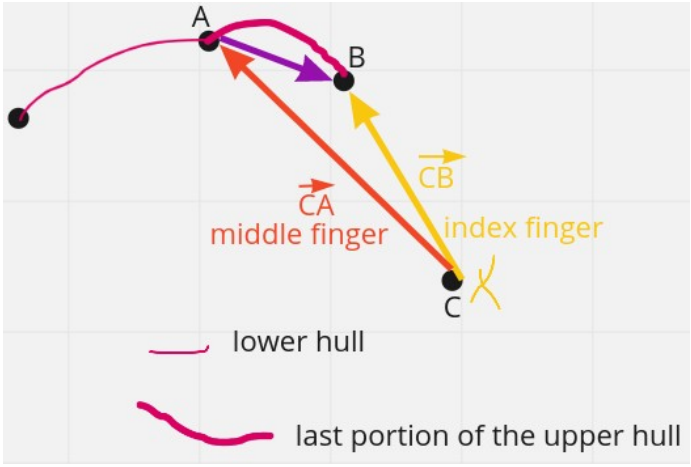
Orientation of the lower hull

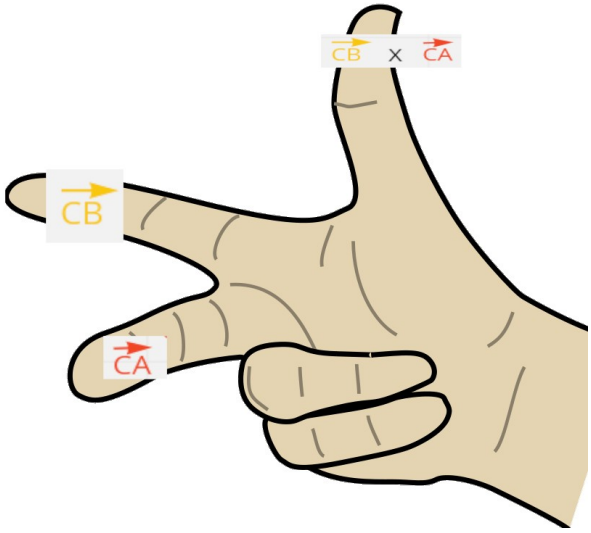




$\vec{CB}$ must be below $\vec{CA}$, that means the cross product $\vec{CB} \times \vec{CA}$ is always negative

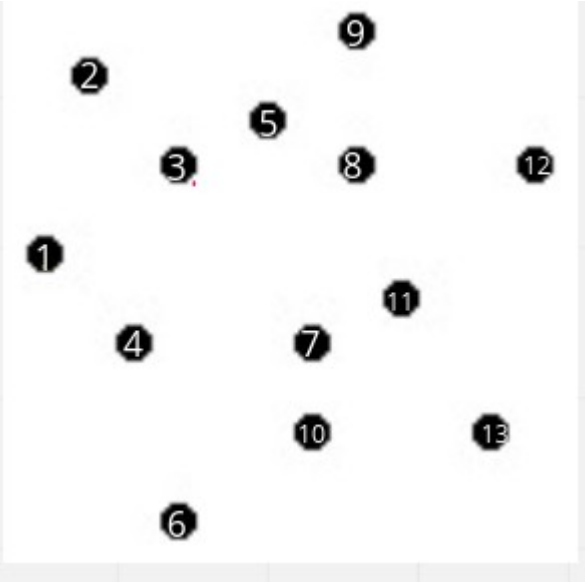
Orientation of the upper hull





$\vec{CB}$ must be above $\vec{CA}$, that means the cross product $\vec{CB} \times \vec{CA}$ is always positive.

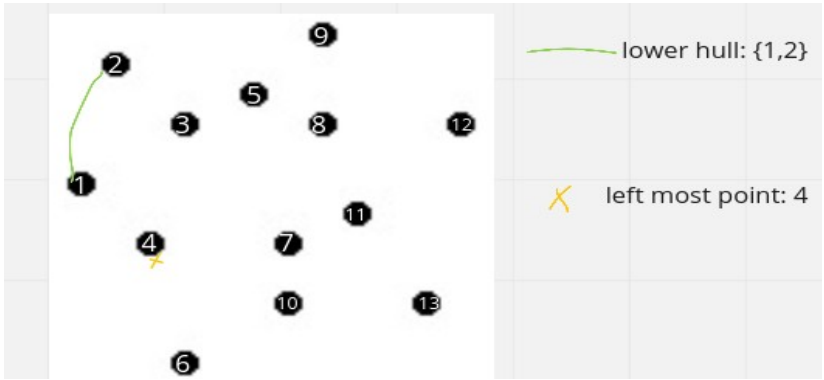
Let’s take the list of points below:

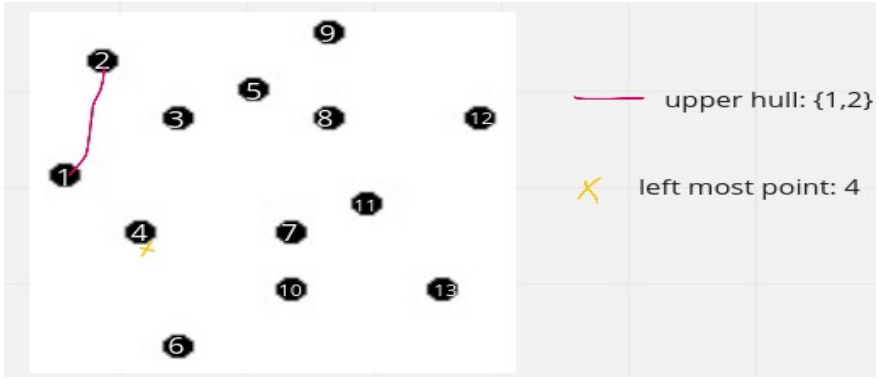


Our task is to build the convex hull using Graham scan algorithm with the divide and conquer concept.

First at all, we order the list of points from the leftmost to the rightmost point. Because here we have a general case (we don't have the x-coordinate of the points), we gonna take this order: $1,2,4,3,6,5,7,10,8,9,11,13,12$.

At the beginning we put the first two leftmost points in both hulls



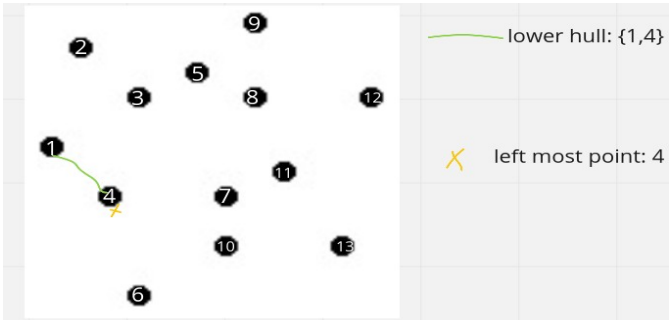


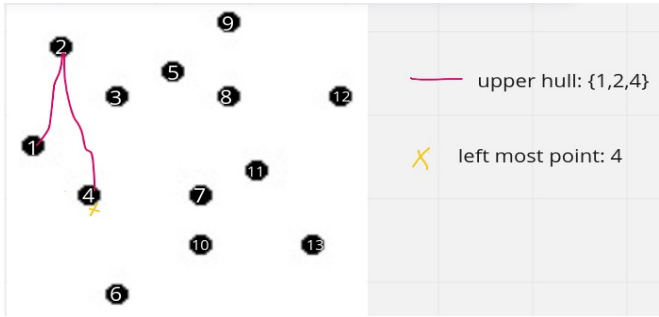
For each point

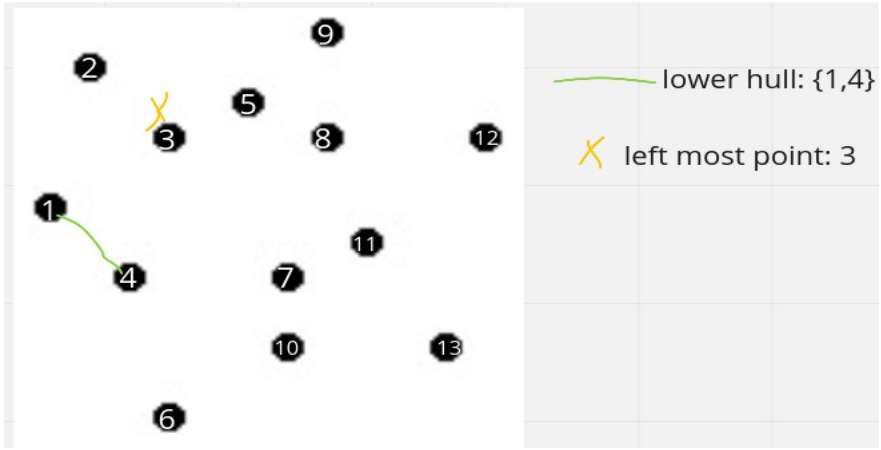
$\vec{4\_2} \times \vec{4\_1} > 0$: can not build lower hull by adding 4 to last portion of lower hull $\{\vec{1,2}\}$:

- delete 2
- lower hull contains one point: add 4

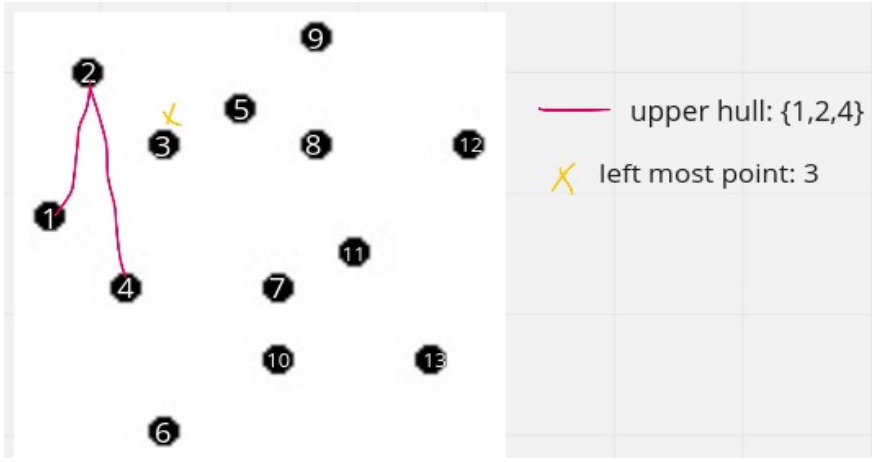
$\vec{4\_2} \times \vec{4\_1} > 0$: we can build upper hull by adding 4 to last portion of upper hull $\{\vec{1,2}\}$



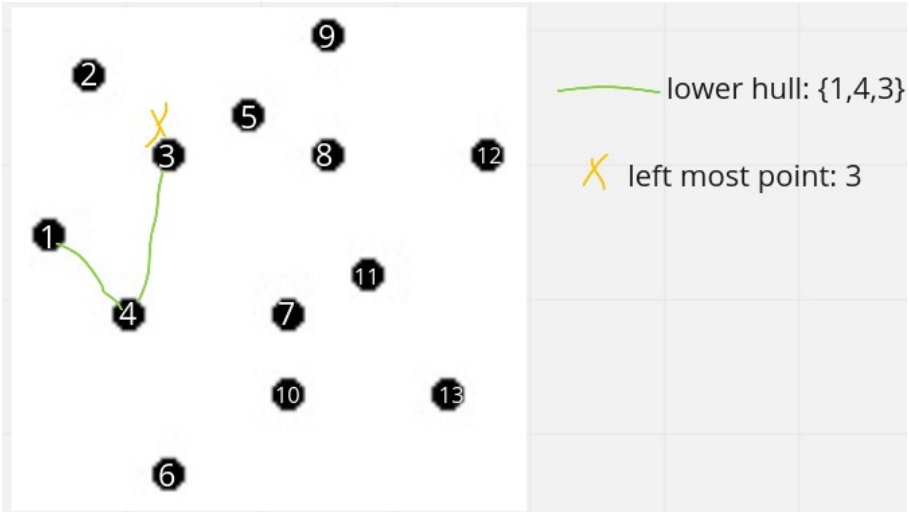


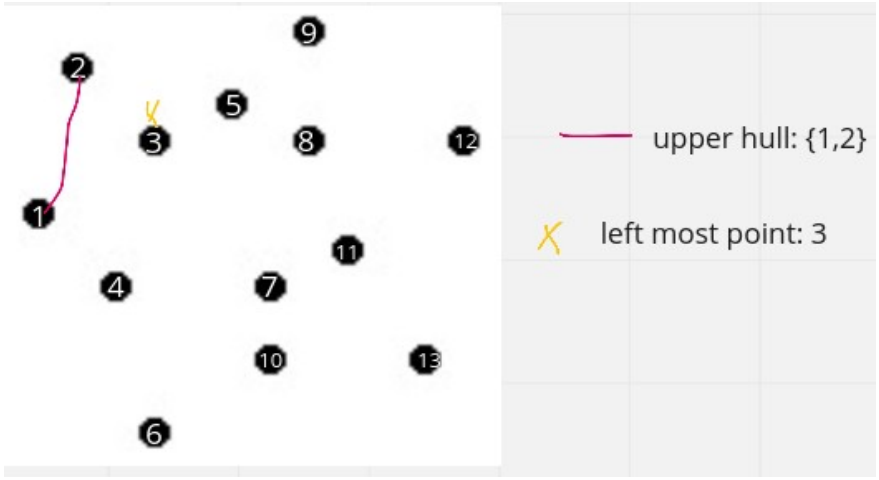


$\vec{3}\vec{4} \times \vec{3}\vec{1} < 0$: we can build lower hull by adding 3 to the last portion of the lower hull $\{\vec{1,4}\}$

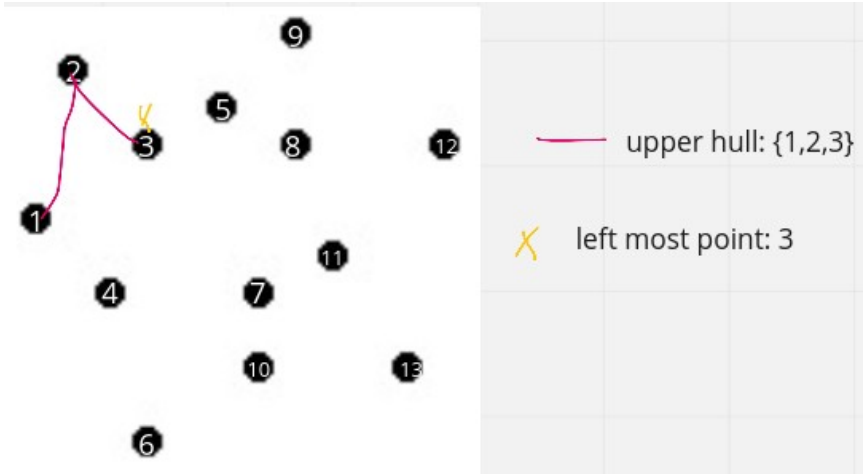


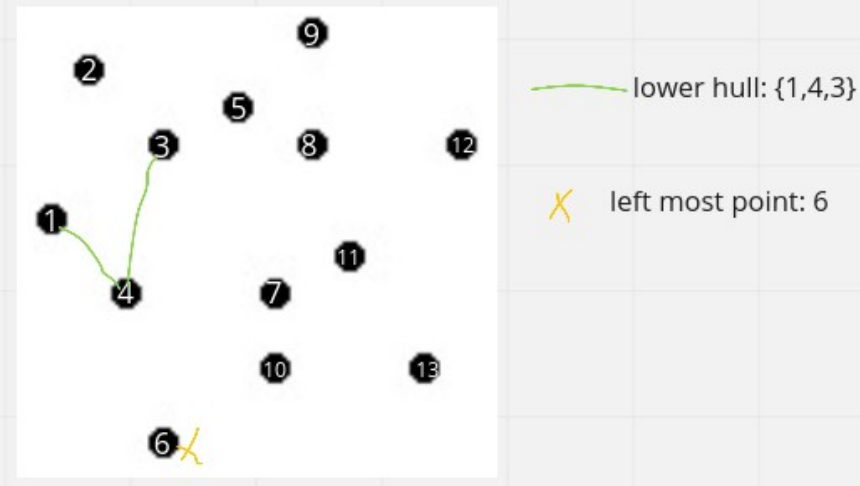
$\vec{3}\vec{4}\, x\, \vec{3}\vec{2} < 0$: we can not build upper hull by adding 3 to the last portion of the upper hull $\{\vec{2,4}\}$: delete 4



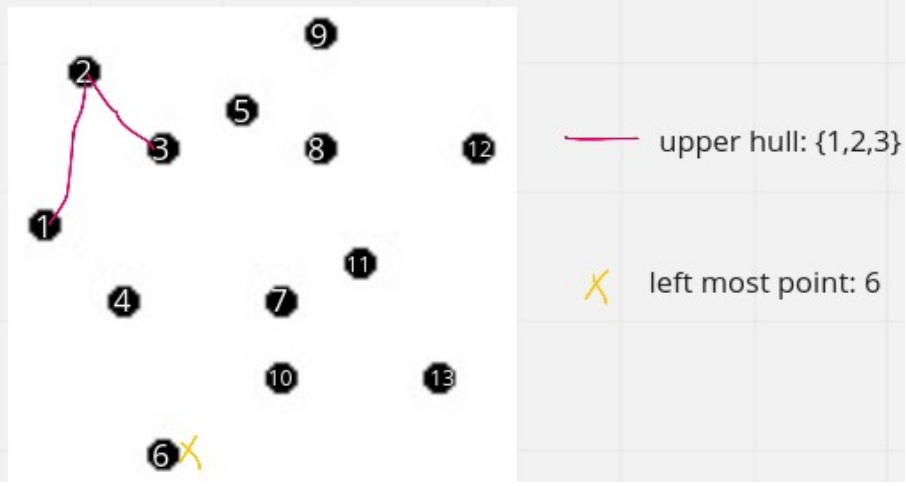


$\vec{3\_2} \times \vec{3\_1} > 0$: we can build upper hull by adding 3 to the last portion of the upper hull $\vec{\{1,2\}}$:

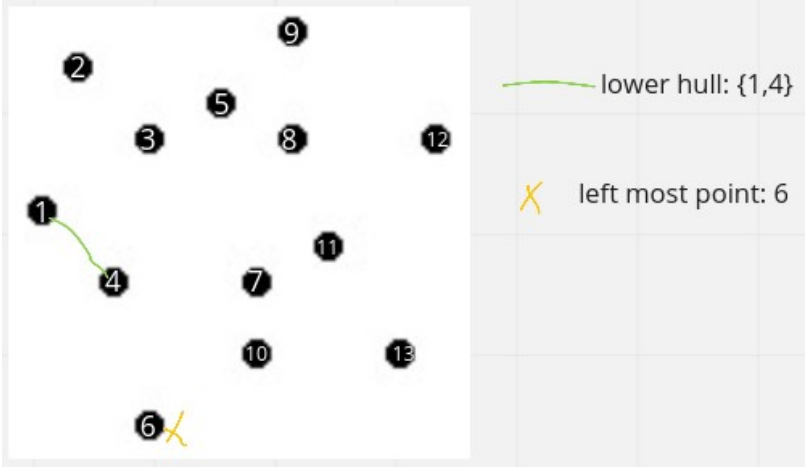


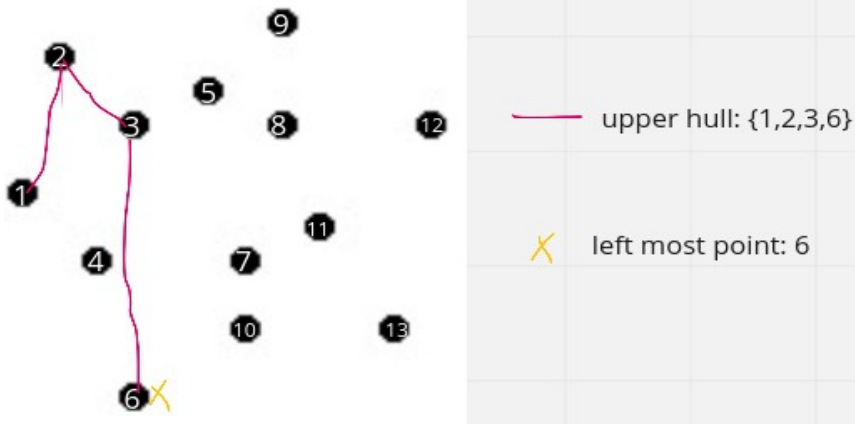


$\vec{6\_3} \times \vec{6\_4} > 0$: we can not build lower hull by adding 6 to last portion of lower hull $\{\vec{4,3}\}$: delete 3



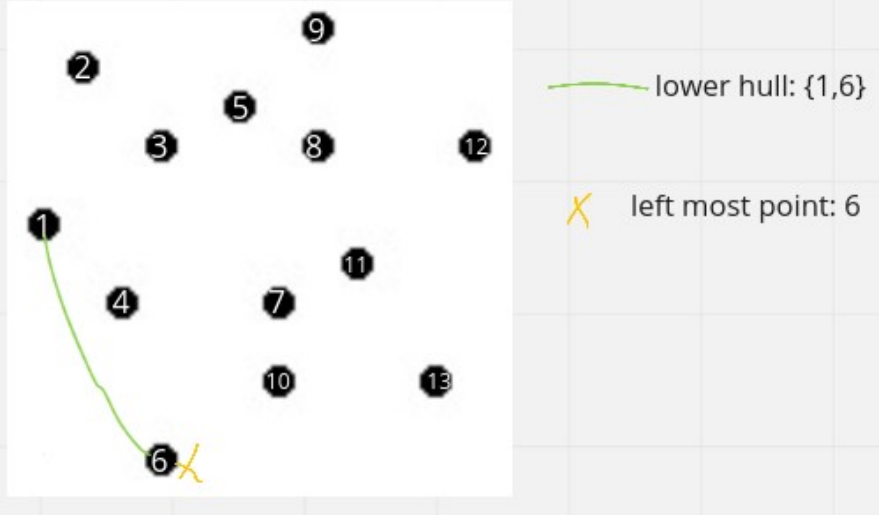
$\vec{6\_3} \times \vec{6\_2} > 0$: we can build upper hull by adding 6 to the last portion the upper hull $\{\vec{2,3}\}$

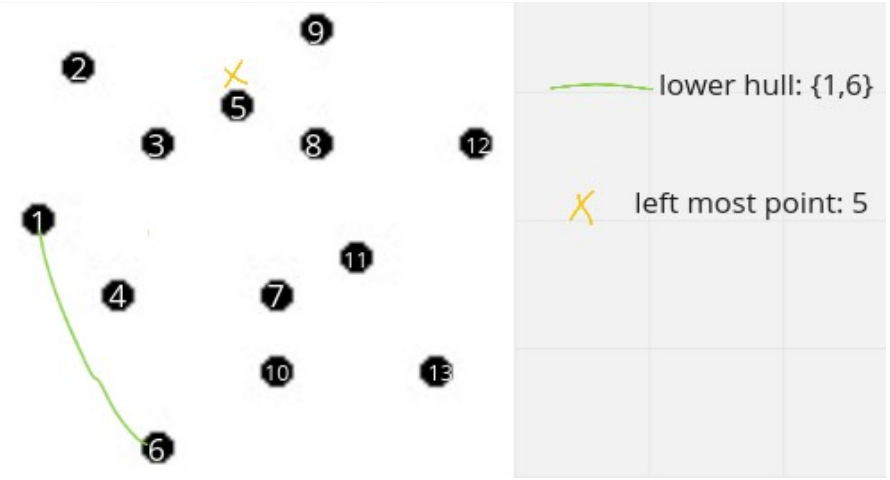




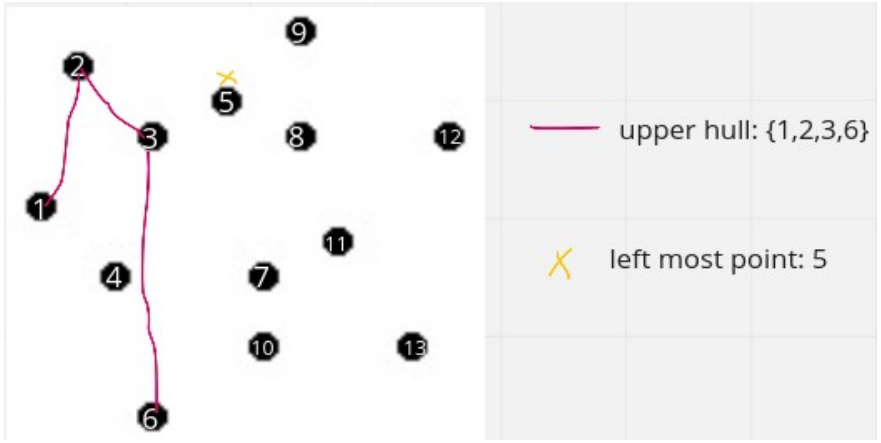
$\vec{6\_4} \, x \, \vec{6\_1} > 0$: we can not build lower hull by adding 6 to $\{\vec{1,4}\}$:

- delete 4
- lower hull contains one point: add 6

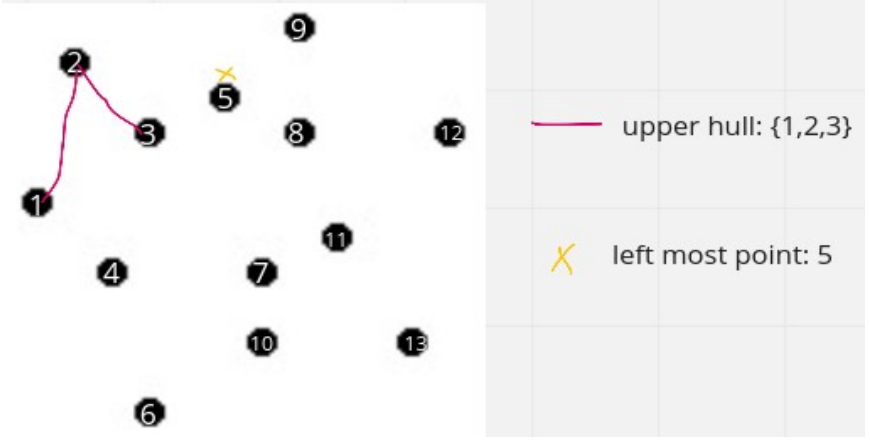




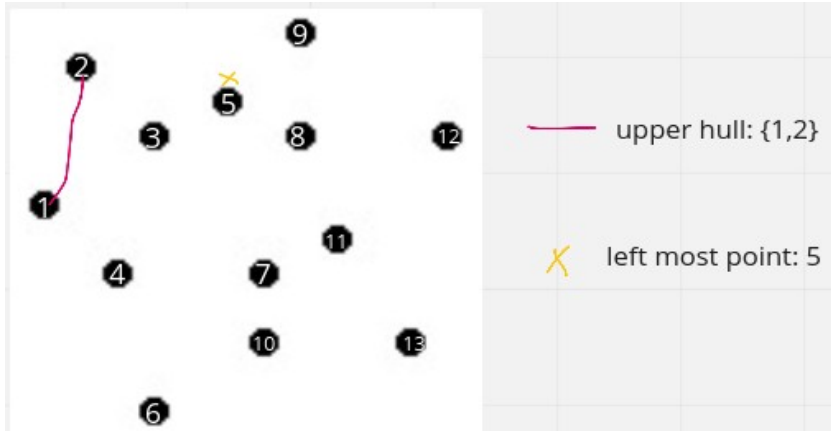
$\vec{5\_6} \times \vec{5\_1} < 0$: we can build lower hull by adding 5 to last portion of the lower hull $\{\vec{1,6}\}$



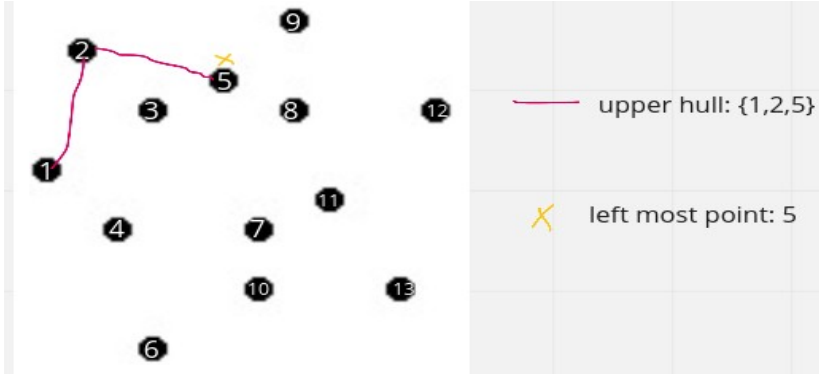
$\vec{5\_6} \times \vec{5\_3} < 0$: we can not build upper hull by adding 5 to last portion of the upper hull $\{\vec{3,6}\}$: delete 6



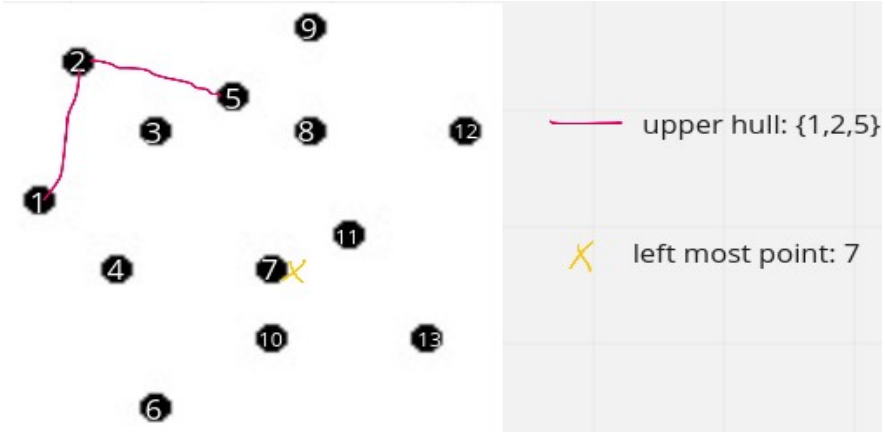
$\vec{5\_3} \times \vec{5\_2} < 0$: we can not build upper hull by adding 5 to the last portion of the upper hull $\{\vec{2,3}\}$: delete 3



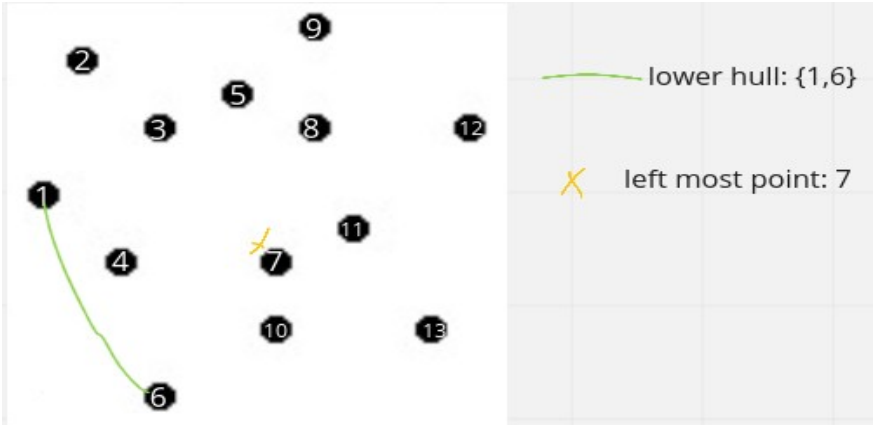
$\vec{5\_2} \times \vec{5\_1} > 0$: we can build upper hull by adding 5 to $\{\vec{1,2}\}$



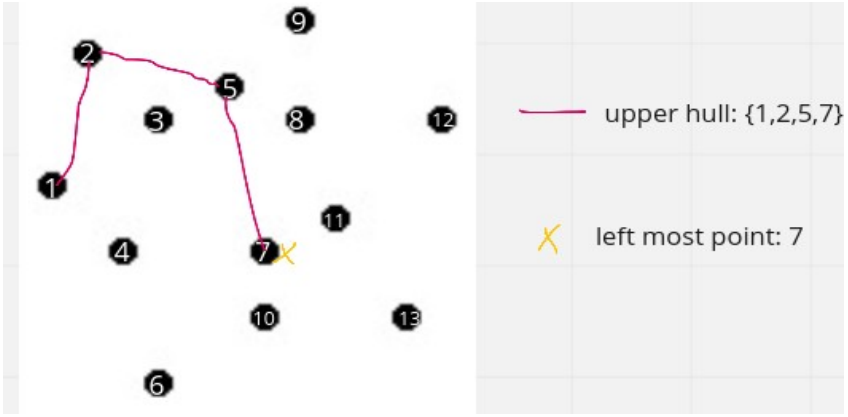
$\vec{7\_5} \times \vec{7\_6} > 0$: we can not build lower hull by adding 7 to the last portion of the lower hull $\{\vec{6,5}\}$: delete 5

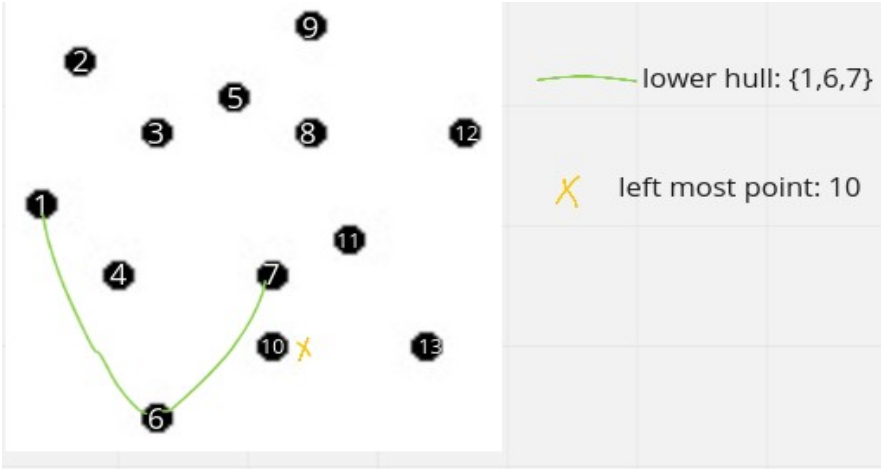


vex $\vec{7\_5} \times \vec{7\_2} > 0$: we can build upper hull by adding 7 to the last portion of the upper hull $\{\vec{2,5}\}$:

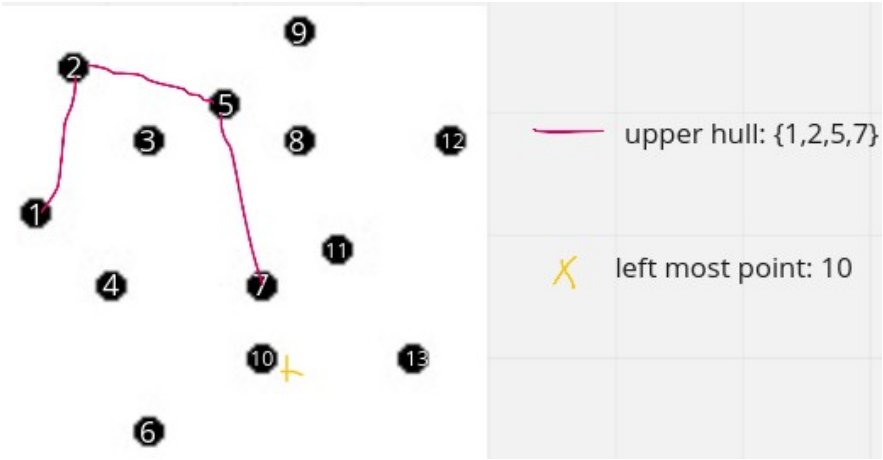


$\vec{7\_6} \times \vec{7\_1} < 0$: we can build lower hull by adding 7 to the last portion of the lower hull $\{\vec{1,6}\}$

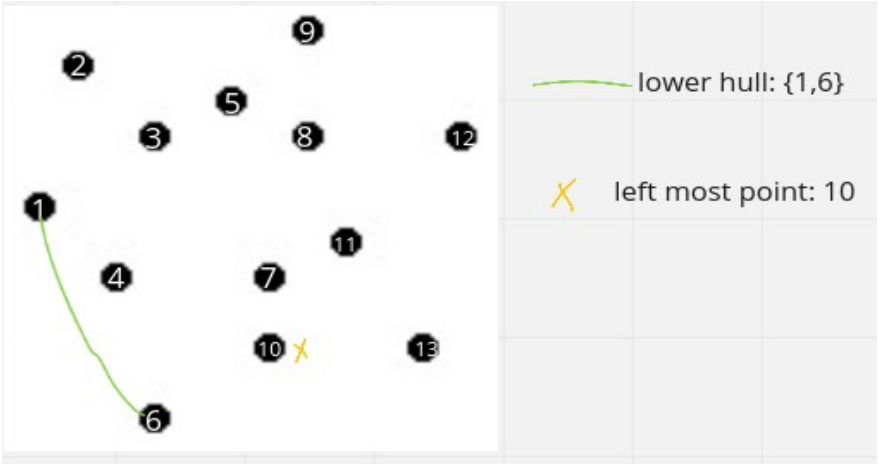


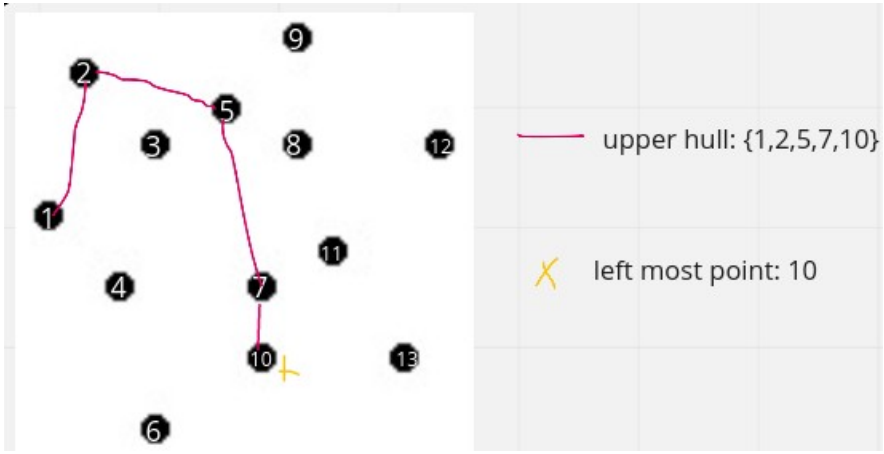


$\vec{10\_7} \times \vec{10\_6} > 0$: we can not build lower hull by adding 10 to the last portion of the lower hull $\{\vec{6,7}\}$: delete 7

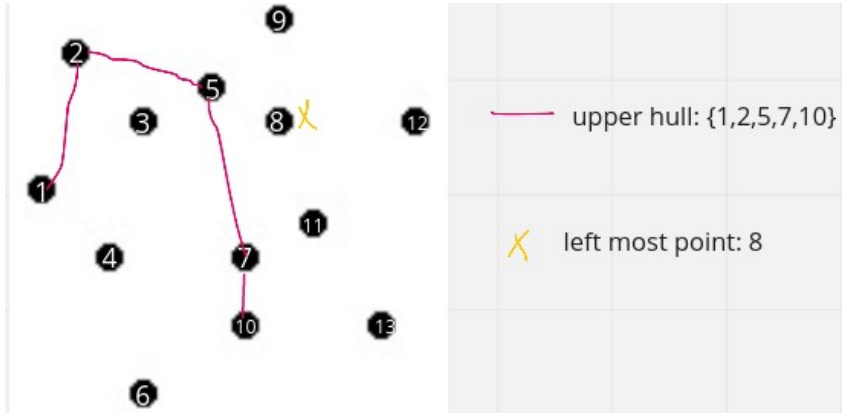


$\vec{10\_7} \times \vec{10\_5} > 0$: we can build upper hull by adding 10 to last portion of upper hull $\{\vec{5,7}\}$:



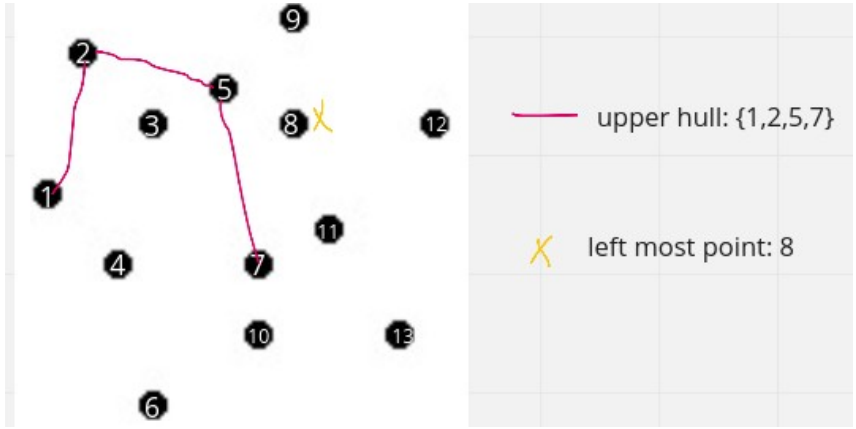


$\vec{10}\_6 \times \vec{10}\_1 < 0$: we can build lower hull by adding 10 to the last portion of the lower hull $\{1, \vec{6}\}$

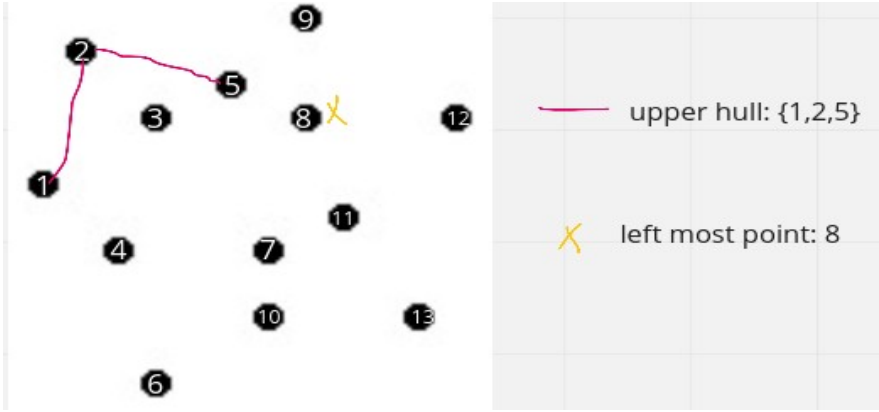


$\vec{8\_10} \times \vec{8\_6} < 0$: we can build lower hull by adding 8 to last portion of the lower hull $\{6, \vec{10}\}$

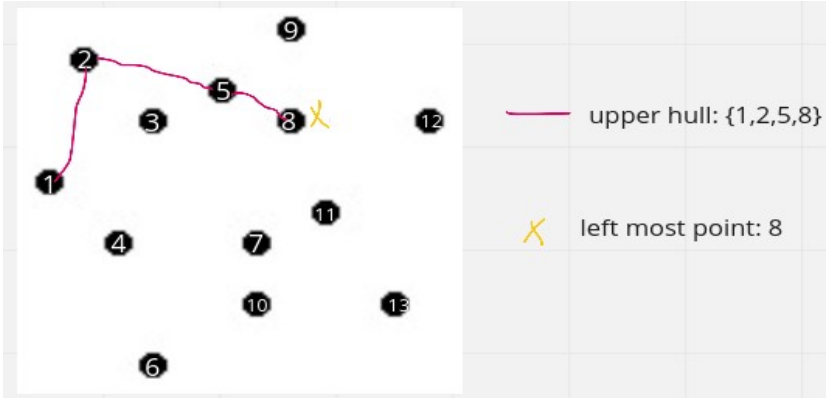
$\vec{8\_10} \times \vec{8\_7} < 0$: we can not build upper hull by adding 8 to last portion of the upper hull $\{7, \vec{10}\}$: delete 10

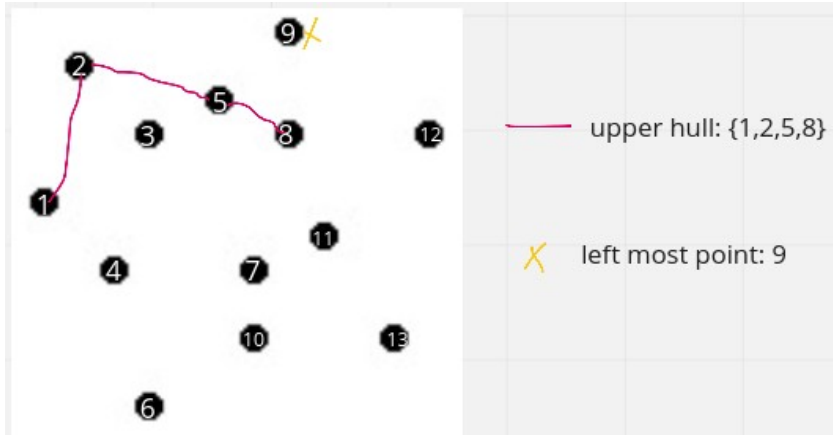


$\vec{8\_7} \times vc\, 8\_5 < 0$: we can not build upper hull by adding 8 to the last portion of the upper hull $\{\vec{5,7}\}$: delete 7



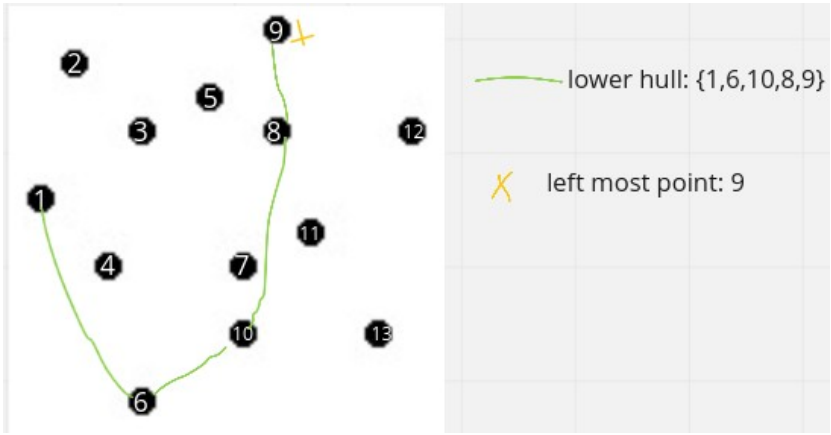
$\vec{8\_5} \times \vec{8\_2} > 0$: we can build upper hull by adding 8 to last portion of the upper hull $\{\vec{2,5}\}$:

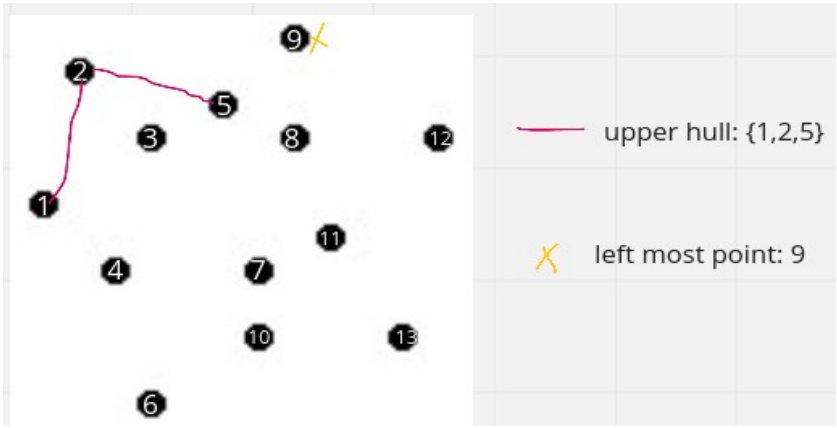




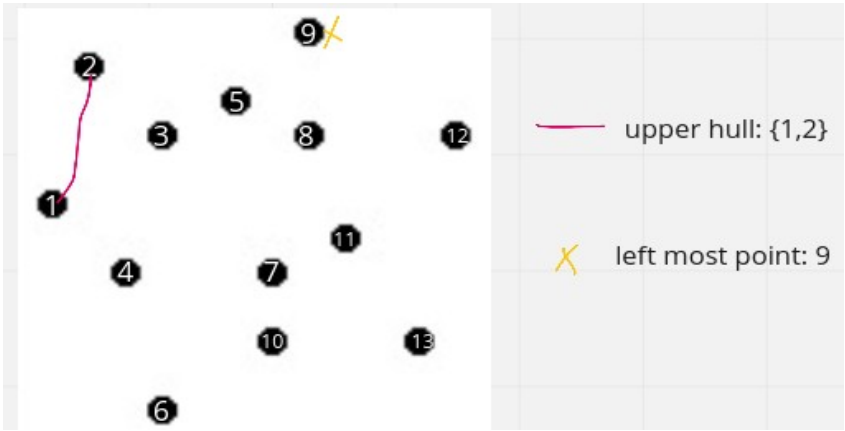
$\vec{9\_8} \times \vec{9\_10} < 0$: we can build lower hull by adding 9 to the last portion of the lower hull $\{\vec{10},8\}$

$\vec{9\_8} \times \vec{9\_5} < 0$: we can not build upper hull by adding 9 to last portion of the upper null $\{\vec{5},8\}$: delete 8

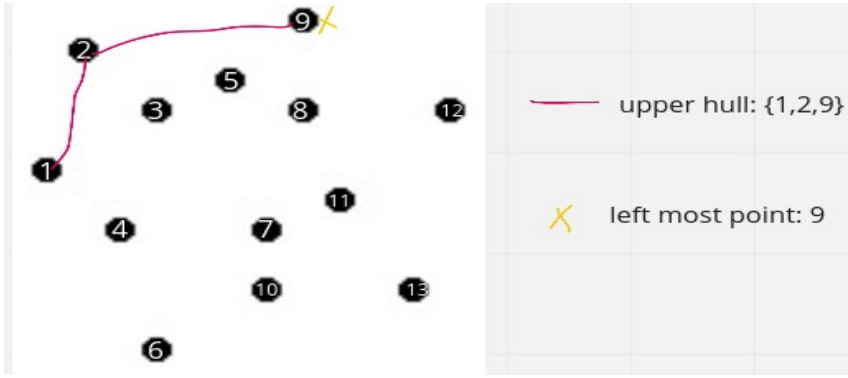


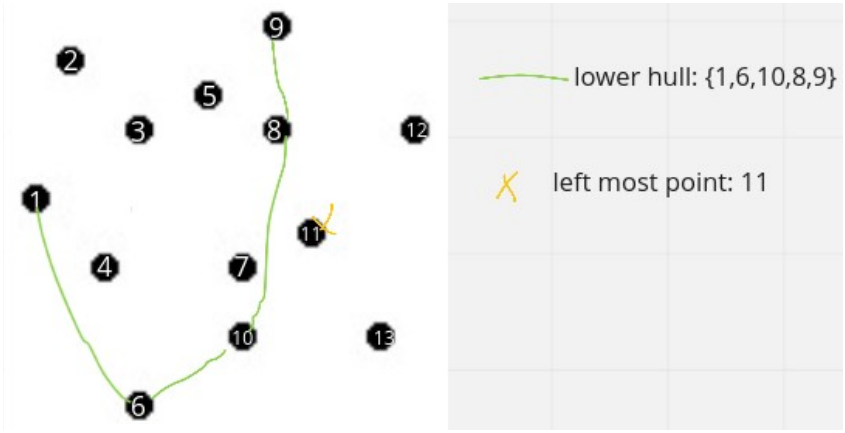


$\vec{9\_5} \times \vec{9\_2} < 0$: we can not build upper hull by adding 9 to the last portion $\{\vec{2,5}\}$: delete 5

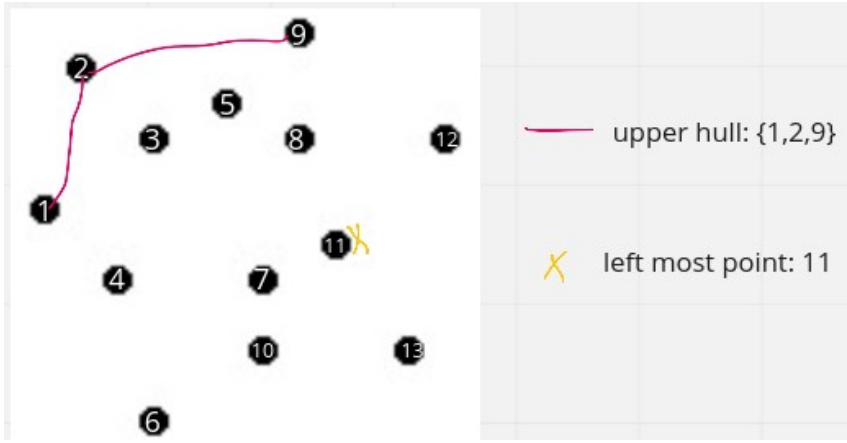


$\vec{9\_2} \times \vec{9\_1} > 0$: we can build upper hull by adding 9 to the last portion of the upper hull $\{\vec{1,2}\}$:



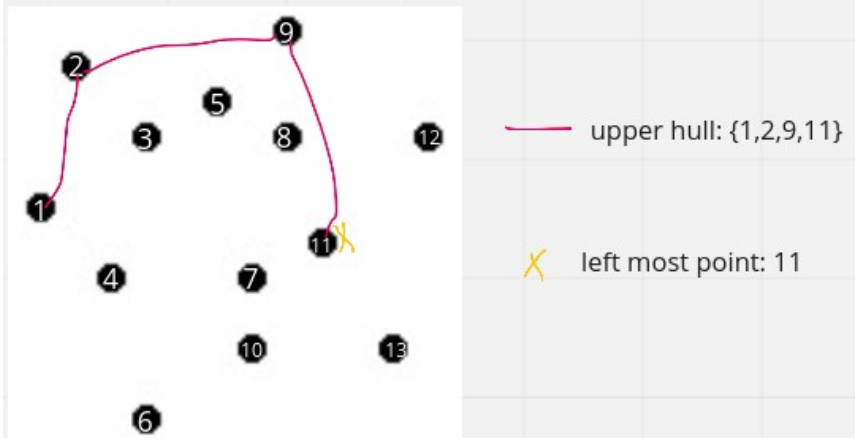


$\vec{11\_9} \times \vec{11\_8} > 0$: we can not build lower hull by adding 11 to last portion of the lower hull $\{\vec{8,9}\}$: delete 9



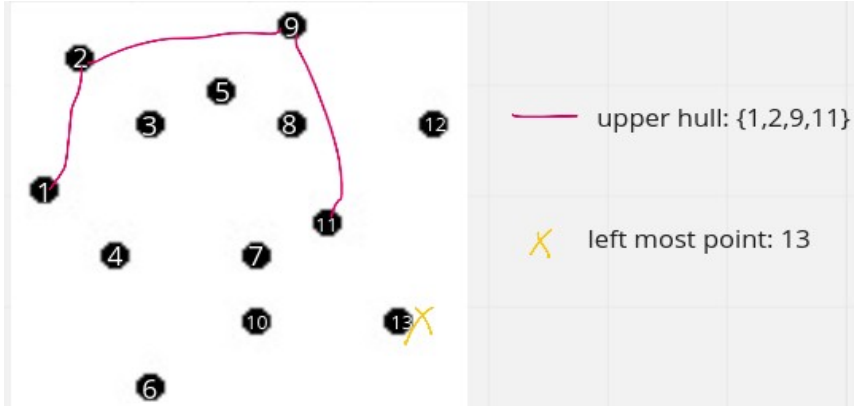
$\vec{11\_9} \times \vec{11\_2} > 0$: we can build upper hull by adding 11 to last portion of the upper hull $\{\vec{2,9}\}$:

$\vec{11\_8} \times \vec{11\_10} > 0$: we can not build lower hull by adding 11 to last portion $\{\vec{10,8}\}$: delete 8



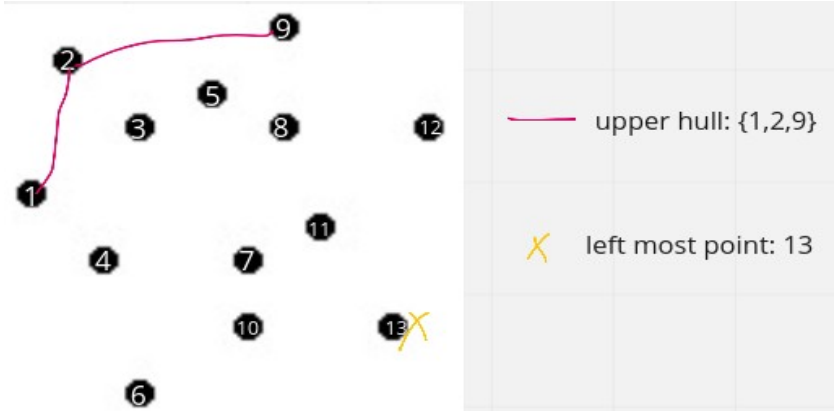
$\vec{11}\_10 \times \vec{11}\_6 < 0$: we can build lower hull by adding 11 to last portion of the lower hull $\{6, \vec{10}\}$

$\vec{13}\_11 \times \vec{13}\_10 > 0$: we can not build lower hull by adding 13 to last portion of the lower hull $\{\vec{10},11\}$: delete 11

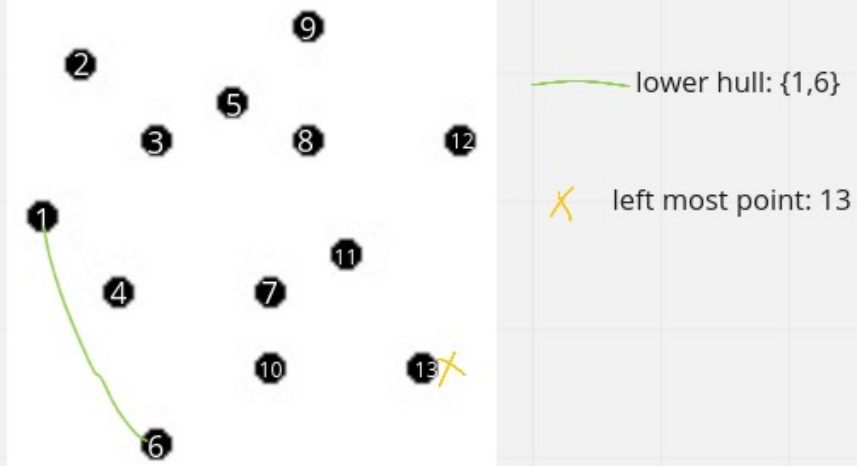


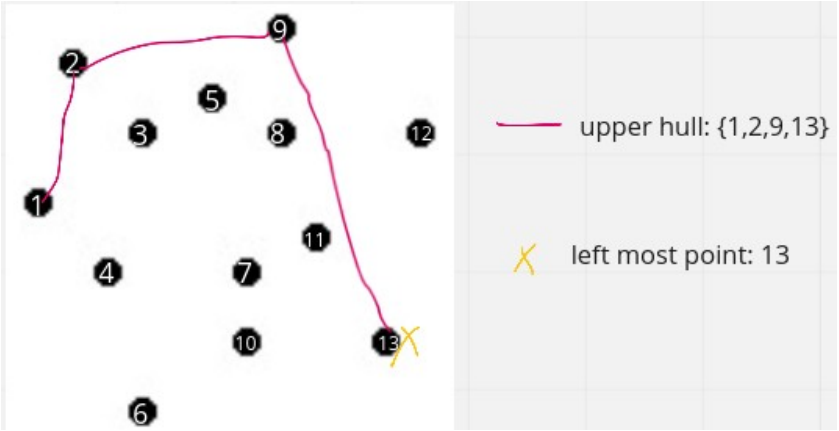
$\vec{13}\_11 \times \vec{13}\_9 < 0$: we cannot build upper hull by adding 13 to the last portion of the upper hull $\{\vec{9},11\}$: delete 11

$\vec{13}\_10 \times \vec{13}\_6 > 0$: we can not build lower hull by adding 13 to last portion of the lower hull $\{\vec{6},10\}$: delete 10

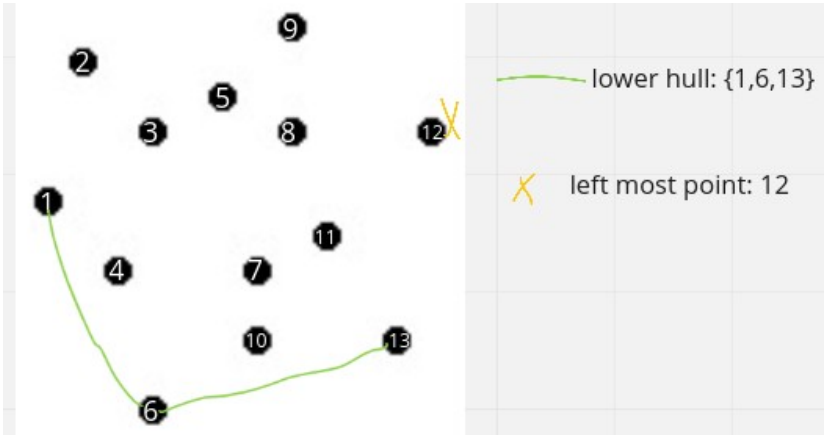


$\vec{13}\_9 \times \vec{13}\_2 > 0$: we can build upper hull by adding 13 to the last portion of the upper hull $\{\vec{2},9\}$:

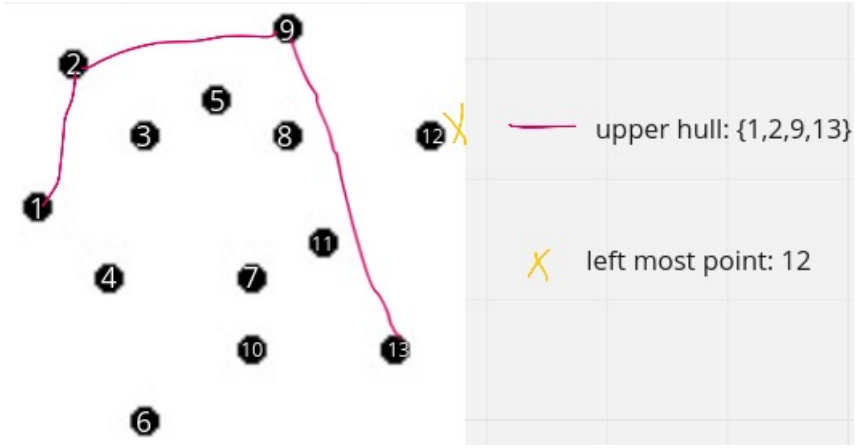




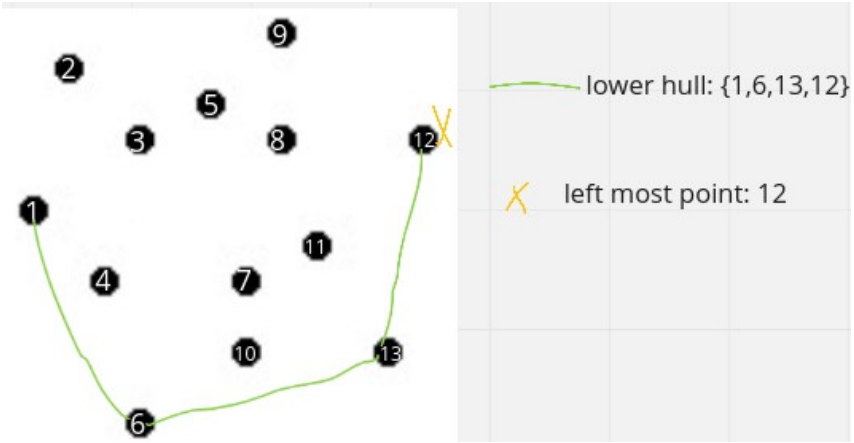
$\vec{13}\_6 \times \vec{13}\_1 < 0$: we can build lower hull by adding 13 to the last portion of the lower hull $\{\vec{1,6}\}$

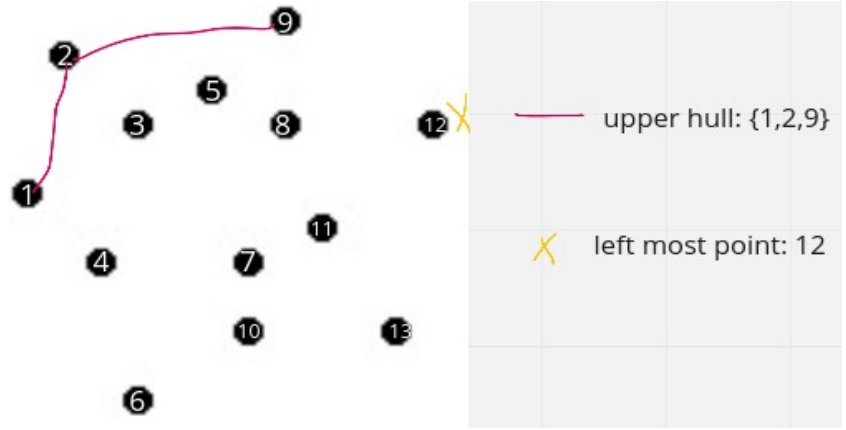


$\vec{12\_13} \times \vec{12\_6} < 0$: we can build lower hull by adding 12 to the last portion of the lower hull $\{6, \vec{13}\}$

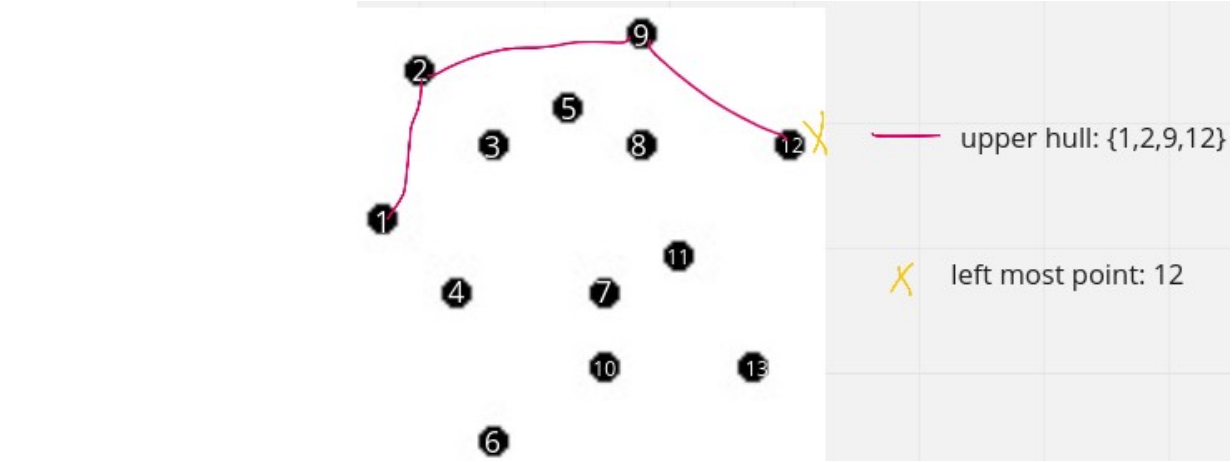


$\vec{12\_13} \times \vec{12\_9} < 0$: we cannot build upper hull by adding 12 to the last portion of the upper hull $\{9, \vec{13}\}$: delete 13





$\vec{12}\_9 \times \vec{12}\_2 > 0$: we can build upper hull by adding 12 to last portion $\{\vec{2},9\}$:



Pseudo code

```
graham_scan(points[]){
    lower[]: will contain the points for lower hull
    upper[]: will contain the points for upper hull
    for each point p in points{
        while lower.size >= 2 and cross_product(lower[lower.size-2],lower[lower.size-1],p > 0){
            delete lower[lower.size-1] from lower[]
        }
        while upper.size >= 2 and cross_product(upper[upper.size-2],upper[upper.size-1],p < 0){
            delete upper[upper.size-1] from upper[]
        }
        add p to lower
        add p to upper
    }
    return (merge lower[] and upper[] without duplication)
}
```

$\vec{ov} \times \vec{ou}$

```
cross_product(point u, point v, point o){
    return (v.x-o.x) * (u.y-o.y) – (u.x-o.x) * (v.y-o.y)
}
```

Practice

To better understand the concept of the convex hull, let practice with a problem from leetcode.

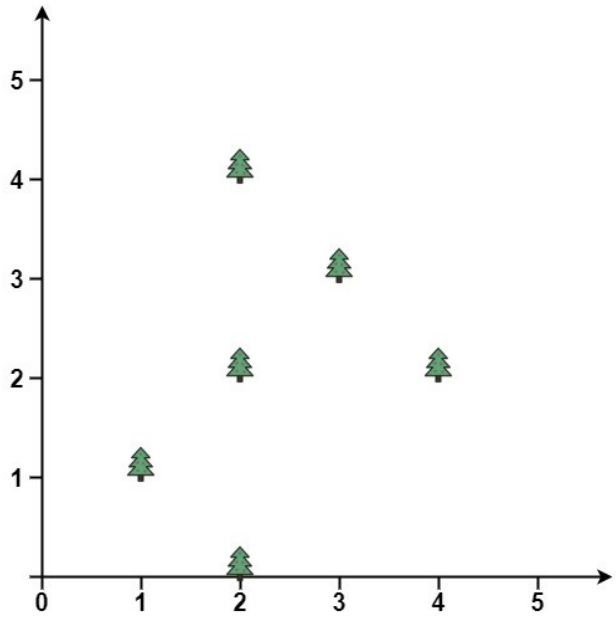
https://leetcode.com/problems/erect-the-fence/

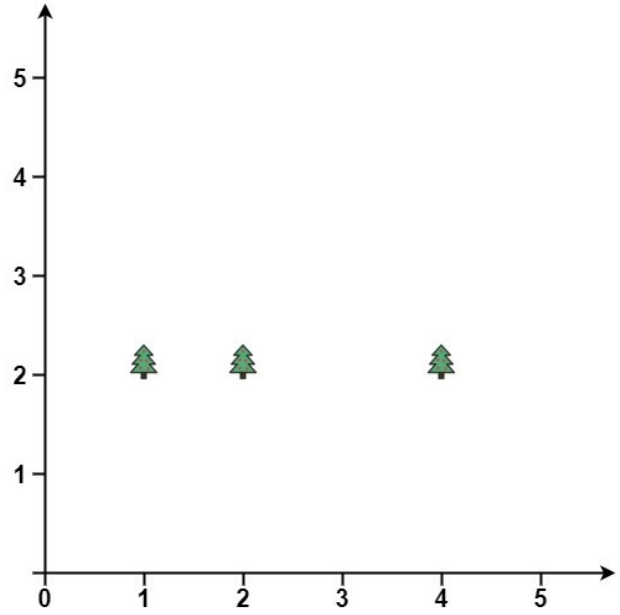
You are given an array `trees` where `trees[i] = [xi, yi]` represents the location of a tree in the garden.

You are asked to fence the entire garden using the minimum length of rope as it is expensive. The garden is well fenced only if **all the trees are enclosed**.

Return the coordinates of trees that are exactly located on the fence perimeter.

Example 1:





Input: points = [[1,1],[2,2],[2,0],[2,4],[3,3],[4,2]]

Output: [[1,1],[2,0],[3,3],[2,4],[4,2]]

Input: points = [[1,2],[2,2],[4,2]]

Output: [[4,2],[2,2],[1,2]]

Constraints:

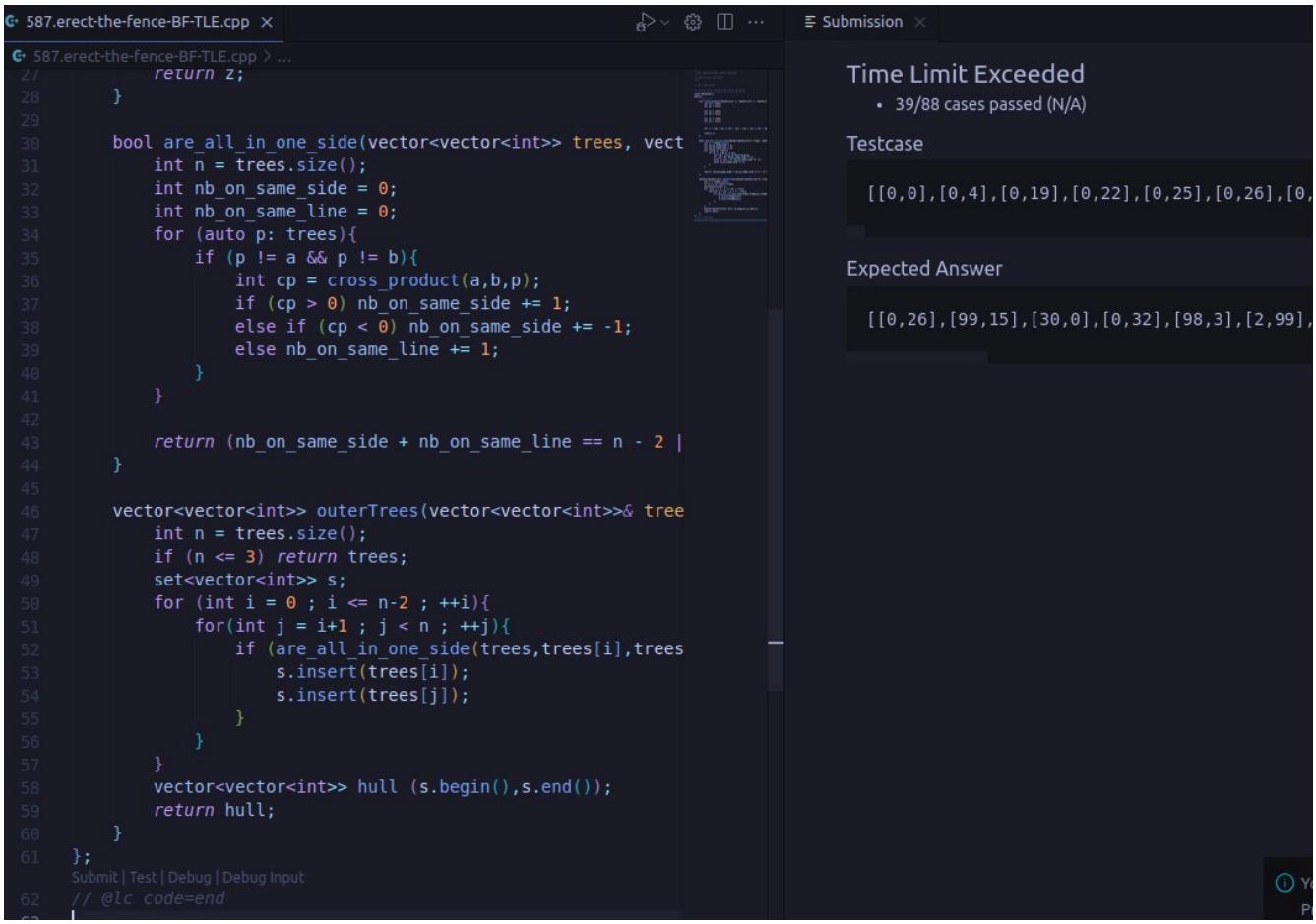
- `1 <= points.length <= 3000`
- `points[i].length == 2`
- `0 <= xi, yi <= 100`
- `All the given points are unique`

Brute force

C++: TLE: $O(n^3)$

```
/*
 * @lc app=leetcode id=587 lang=cpp
 *
 * [587] Erect the Fence
 */

// @lc code=start

// [[1,1],[2,2],[2,0],[2,4],[3,3],[4,2]]\n
// [[0,2],[1,1],[2,2],[2,4],[4,2],[3,3]]\n
class Solution {
public:
    // ov x ou
    int cross_product(vector<int> u, vector<int> v, vector<int> o){
        int xo = o[0];
        int yo = o[1];

        int xu = u[0];
        int yu = u[1];

        int xv = v[0];
        int yv = v[1];


        int z = (yu - yo) * (xv - xo) - (xu - xo) * (yv - yo);

        return z;
    }
```

```
    bool are_all_in_one_side(vector<vector<int>> trees, vector<int> a, vector<int> b){
        int n = trees.size();
        int nb_on_same_side = 0;
        int nb_on_same_line = 0;
        for (auto p: trees){
            if (p != a && p != b){
                int cp = cross_product(a,b,p);
                if (cp > 0) nb_on_same_side += 1;
                else if (cp < 0) nb_on_same_side += -1;
                else nb_on_same_line += 1;
            }
        }

        return (nb_on_same_side + nb_on_same_line == n - 2 || nb_on_same_side - nb_on_same_line == -(n - 2));
    }

    vector<vector<int>> outerTrees(vector<vector<int>>& trees) {
        int n = trees.size();
        if (n <= 3) return trees;
        set<vector<int>> s;
        for (int i = 0 ; i <= n-2 ; ++i){
            for(int j = i+1 ; j < n ; ++j){
                if (are_all_in_one_side(trees,trees[i],trees[j])) {
                    s.insert(trees[i]);
                    s.insert(trees[j]);
                }
            }
        }
        vector<vector<int>> hull (s.begin(),s.end());
        return hull;
    }
};
// @lc code=end
```

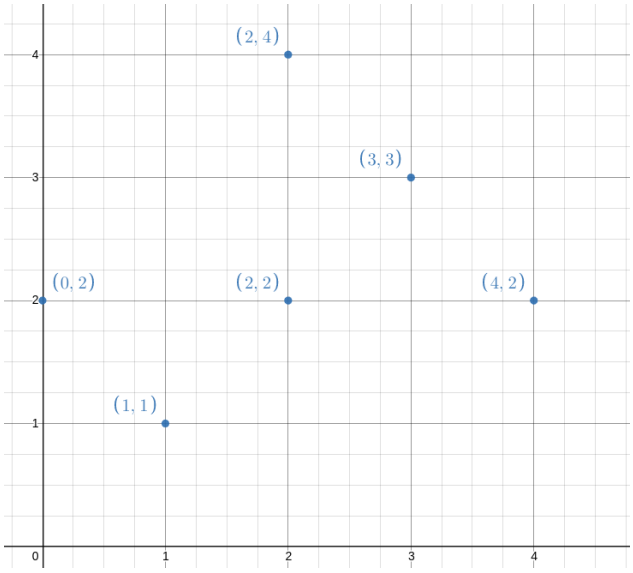

587.erect-the-fence-BF-TLE.cpp
return z;
}
bool are_all_in_one_side(vector<vector<int>> trees, vect
int n = trees.size();
int nb_on_same_side = 0;
int nb_on_same_line = 0;
for (auto p: trees){
if (p != a && p != b){
int cp = cross_product(a,b,p);
if (cp > 0) nb_on_same_side += 1;
else if (cp < 0) nb_on_same_side += -1;
else nb_on_same_line += 1;
}
}
return (nb_on_same_side + nb_on_same_line == n - 2 |
}
vector<vector<int>> outerTrees(vector<vector<int>>& tree
int n = trees.size();
if (n <= 3) return trees;
set<vector<int>> s;
for (int i = 0 ; i <= n-2 ; ++i){
for(int j = i+1 ; j < n ; ++j){
if (are_all_in_one_side(trees,trees[i],trees
s.insert(trees[i]);
s.insert(trees[j]);
}
}
}
vector<vector<int>> hull (s.begin(),s.end());
return hull;
}
};
Submit | Test | Debug | Debug Input
// @lc code=end
Submission
Time Limit Exceeded
39/88 cases passed (N/A)
Testcase
[[0,0],[0,4],[0,19],[0,22],[0,25],[0,26],[0,
Expected Answer
[[0,26],[99,15],[30,0],[0,32],[98,3],[2,99],

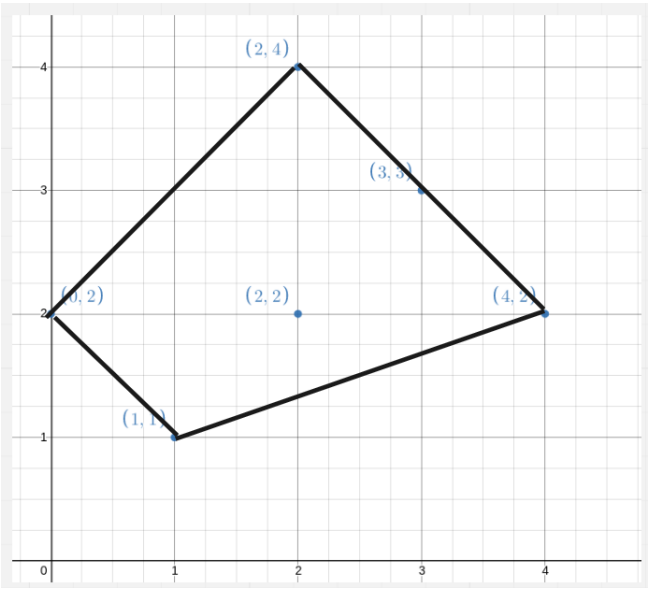
Graham scan algorithm combined with divide and conquer concept

Example

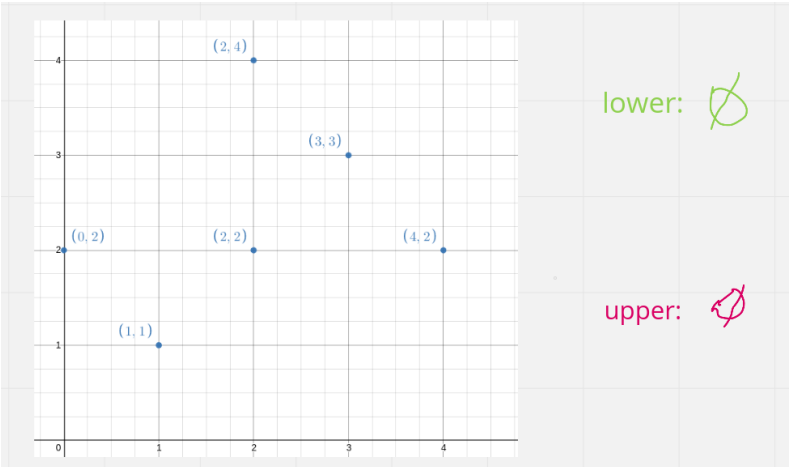
Let’s go throw this example:

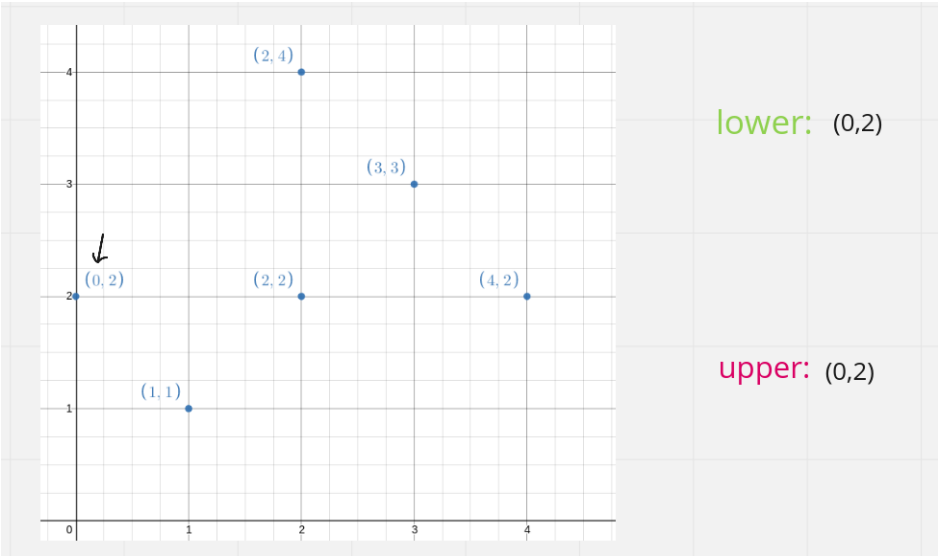
trees:	[0,2]	[1,1]	[2,2]	[2,4]	[4,2]	[3,3]
	0	1	2	3	4	



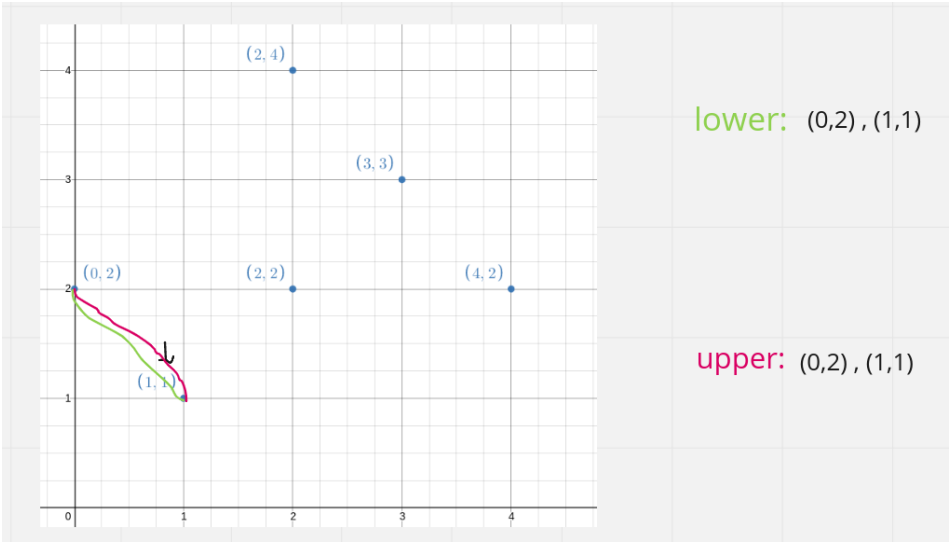


Overview example explanation



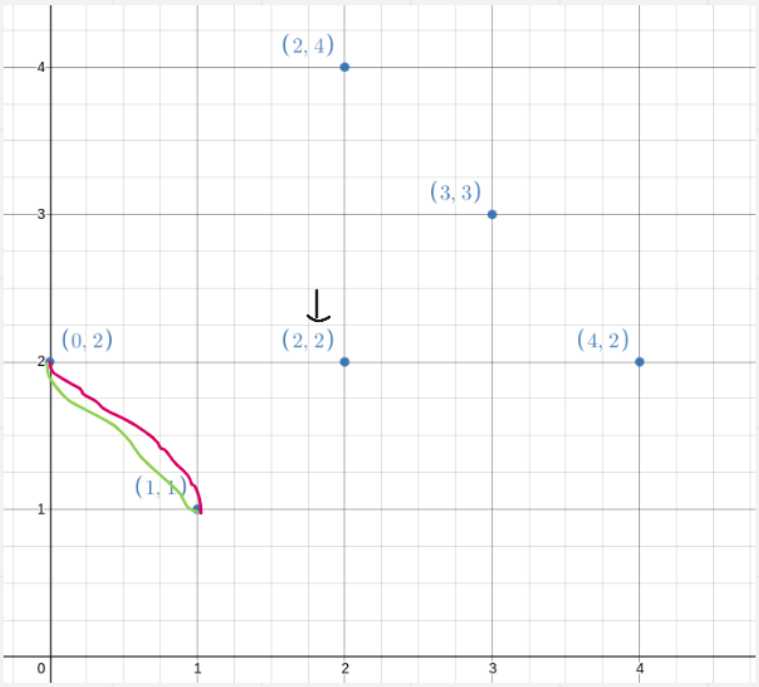


Left most point is (0,2). This the 1st point of both convex hulls



Left most point is (1,1). This the 2nd point of both convex hull

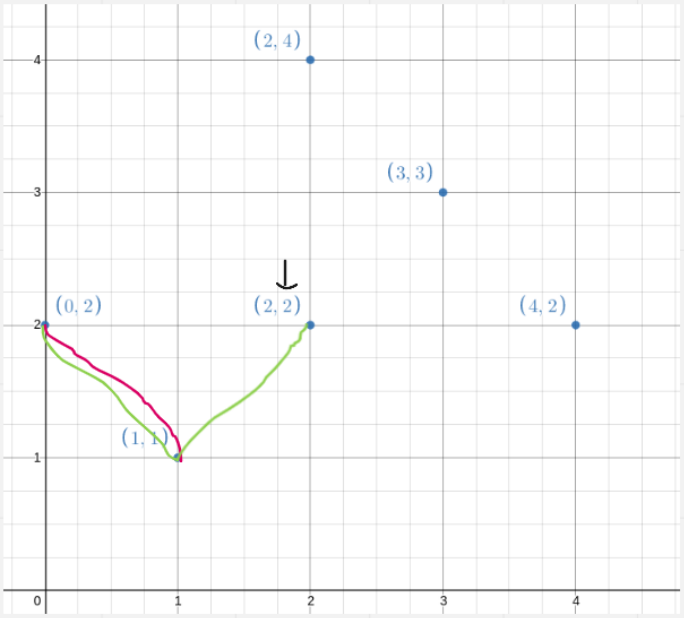
Left most point is (2,2).



lower: (0,2) , (1,1)

upper: (0,2) , (1,1)

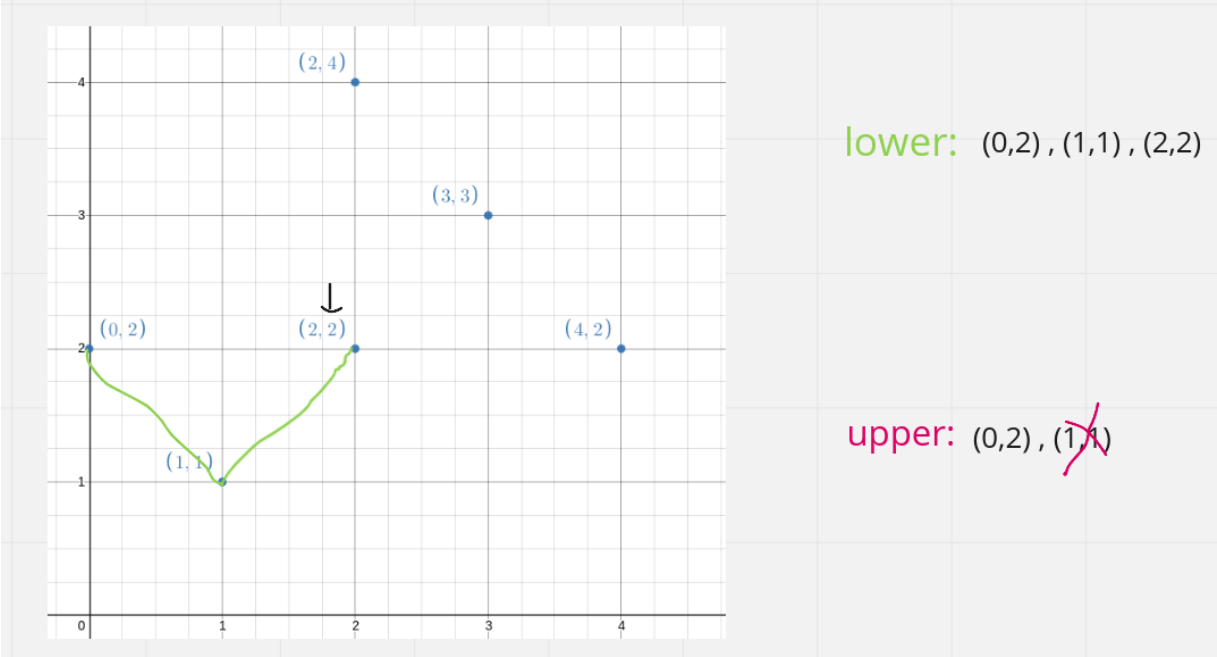
Can we build a lower convex hull with the point (2,2) from (1,1)? Yes: add (2,2)



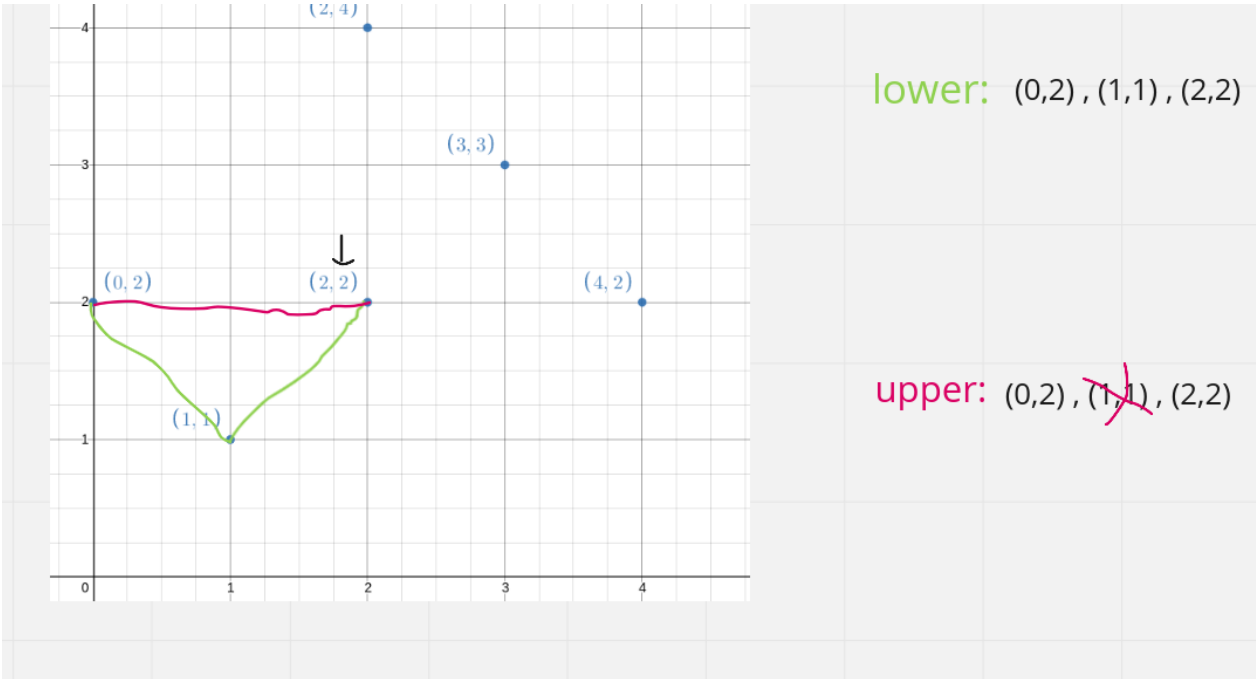
lower: (0,2) , (1,1) , (2,2)

upper: (0,2) , (1,1)

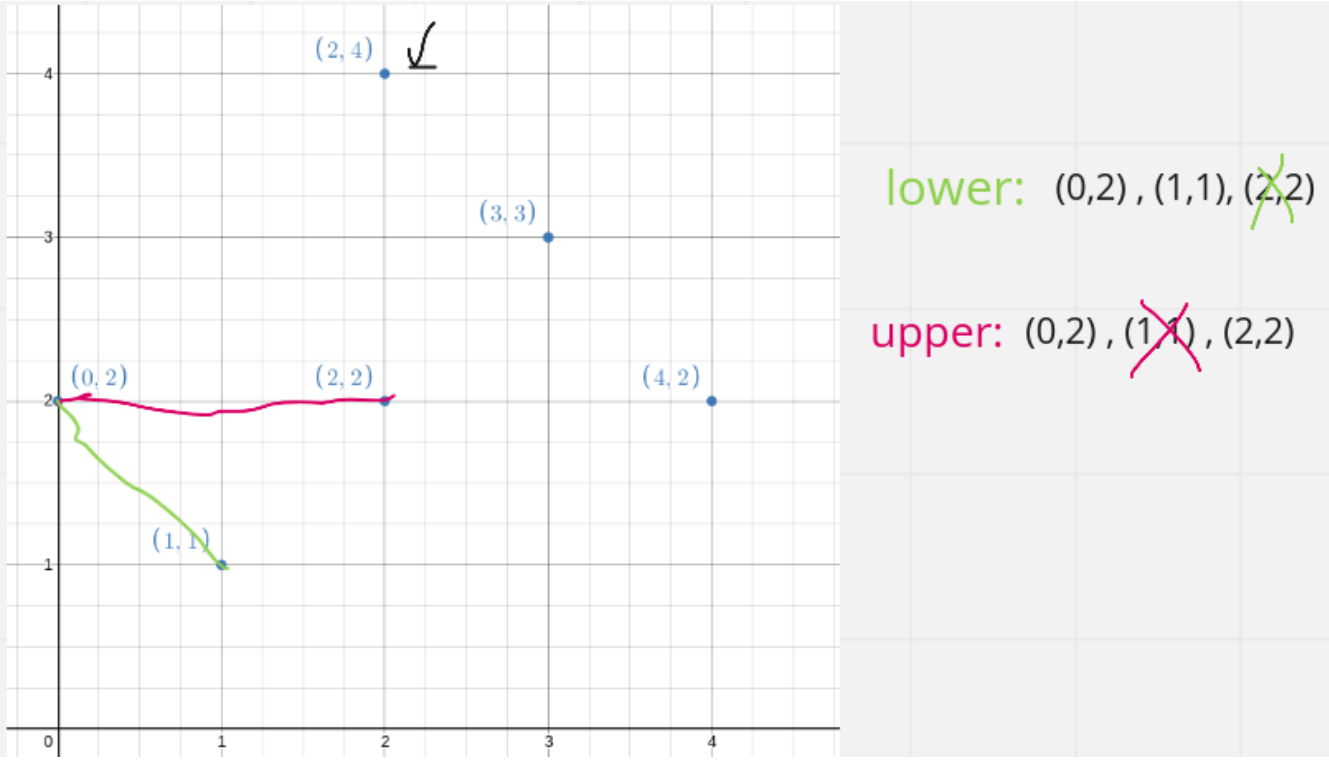
Can we build a upper convex hull with the point (2,2) from (1,1)? No: delete (1,1)



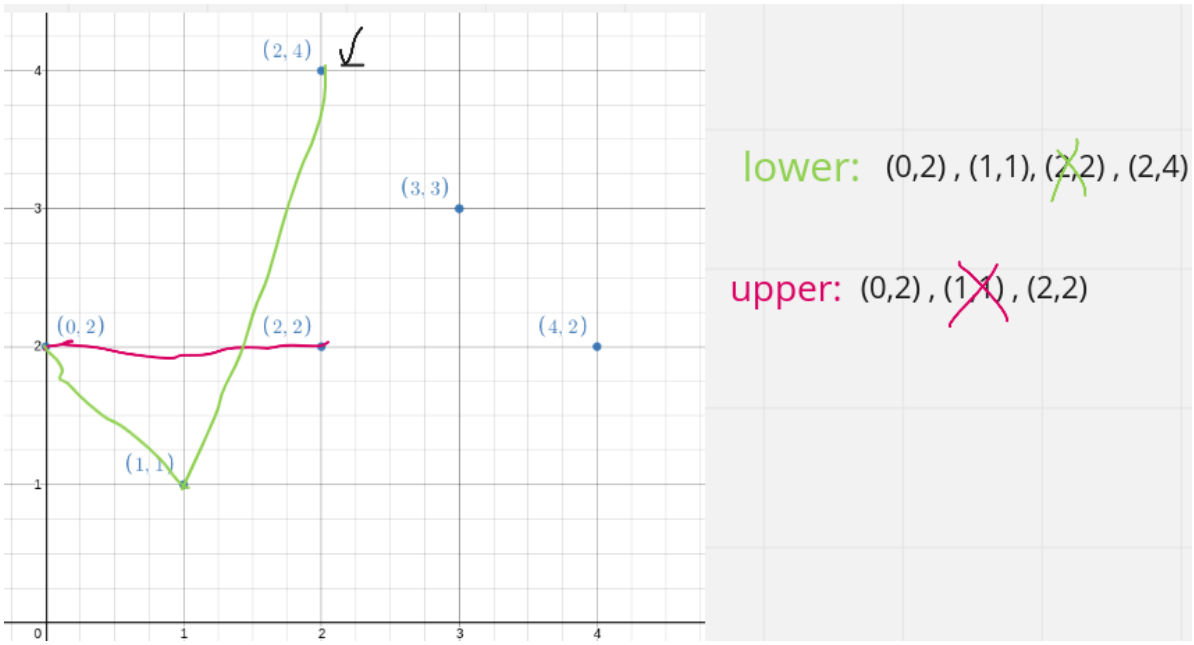
Just one point is in the upper convex hull, so add (2,2)



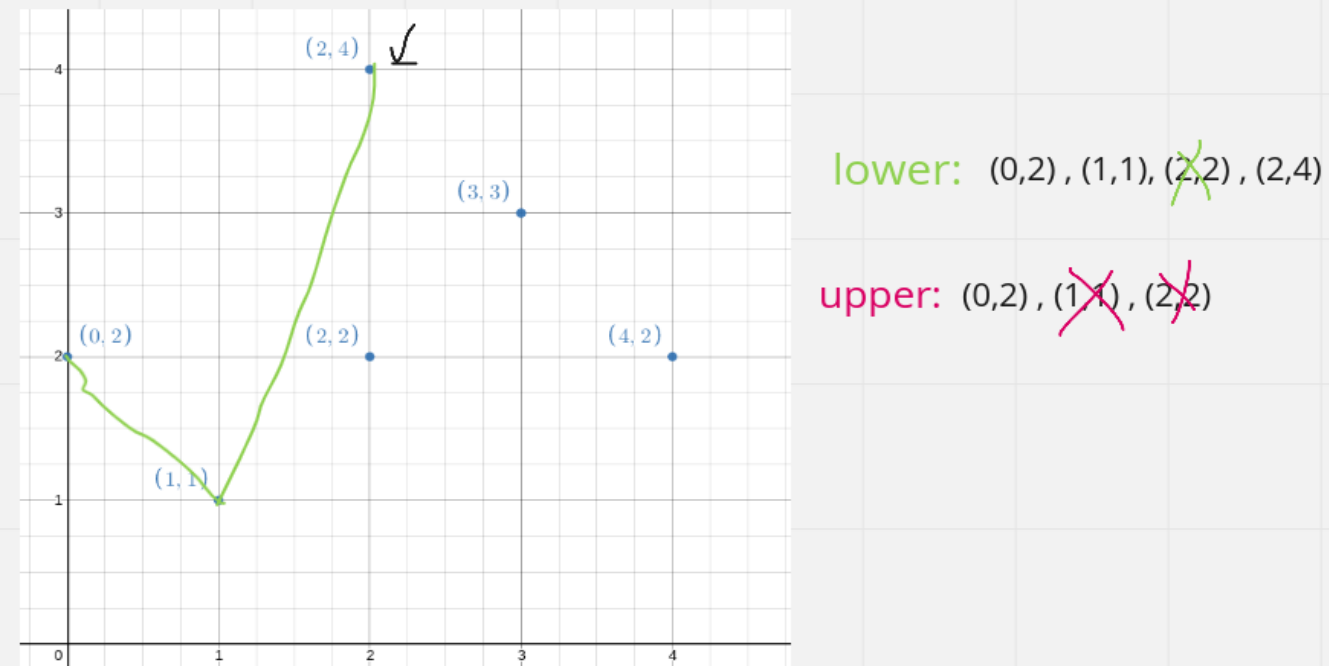
Left most point is (2,4).
Can we build a lower convex hull with the point (2,4) from {(1,1),(2,2)}? No: delete (2,2)



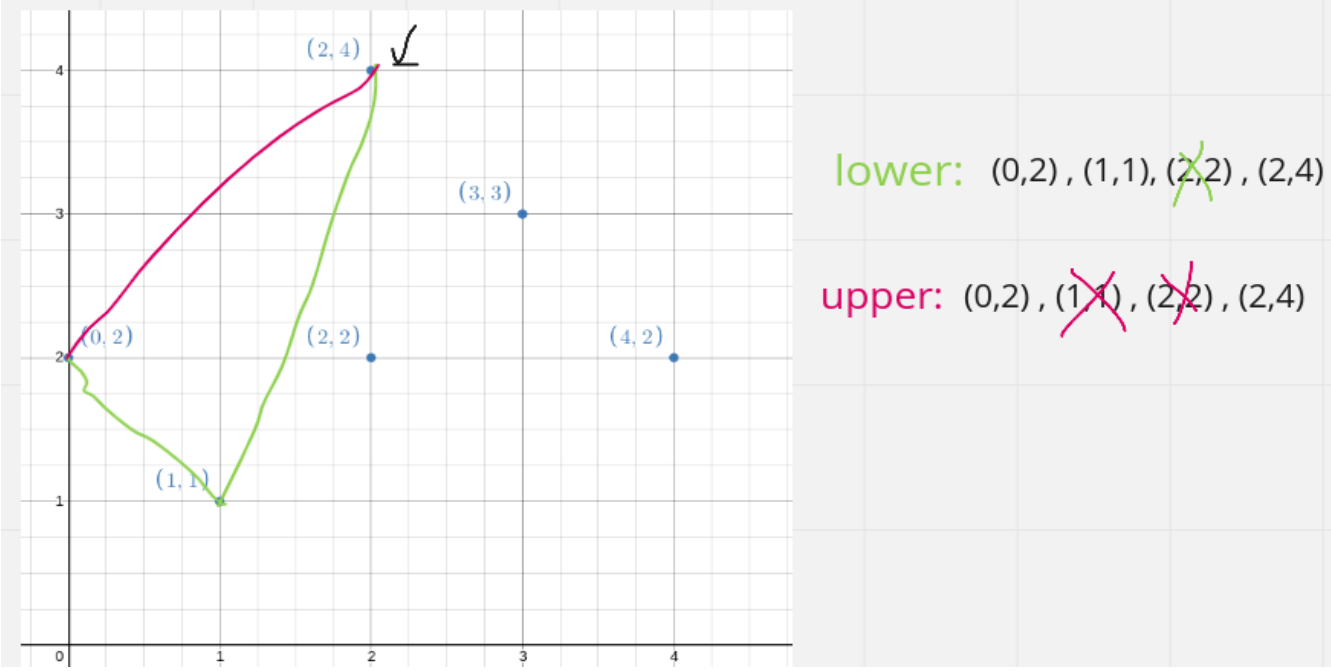
Can we build a lower convex hull with the point (2,4) from {(0,2),(1,1)}? Yes: add (2,4)



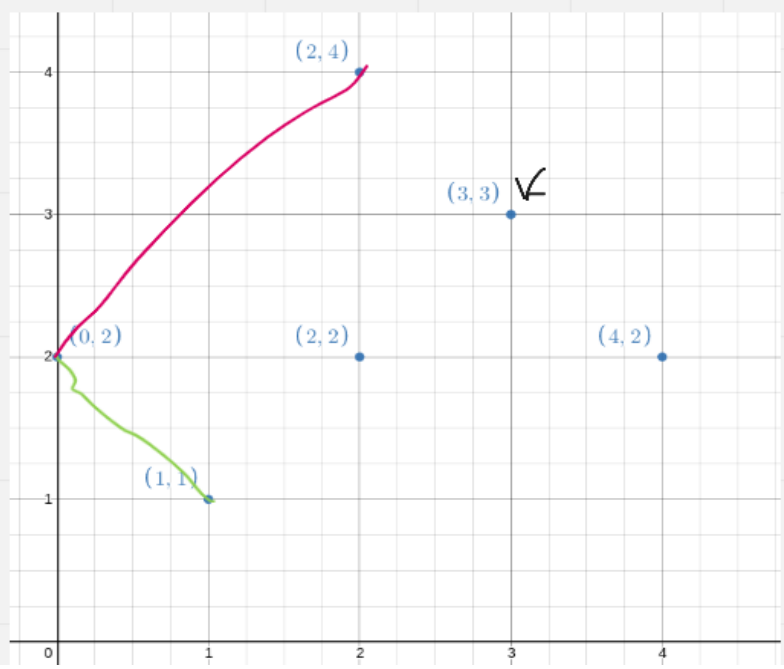
Can we build a upper convex hull with the point (2,4) from {(0,2),(2,2)}? No: delete (2,2)



Just one point is in the upper convex hull, so add (2,4)



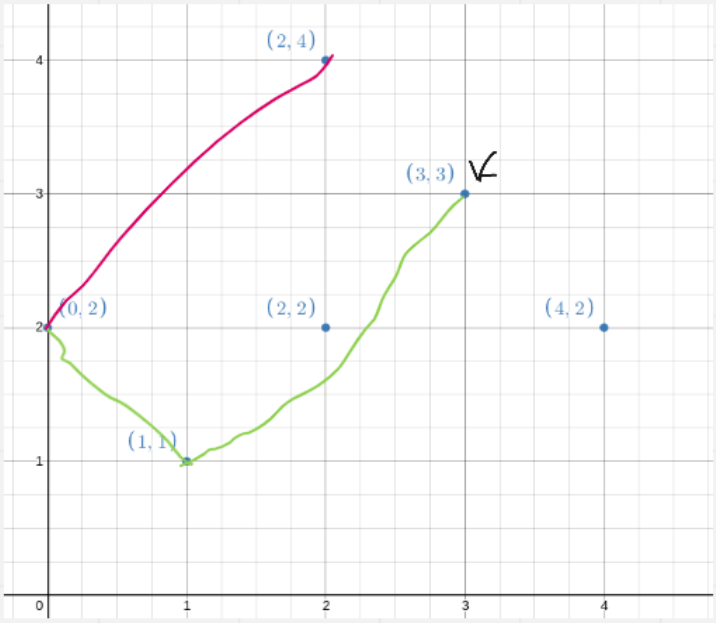
Left most point is (3,3).
Can we build a lower convex hull with the point (3,3) from {(1,1),(2,4)}? No: Delete (2,4)



lower: (0,2) , (1,1), (2,2) , (2,4)

upper: (0,2) , (1,1) , (2,2) , (2,4)

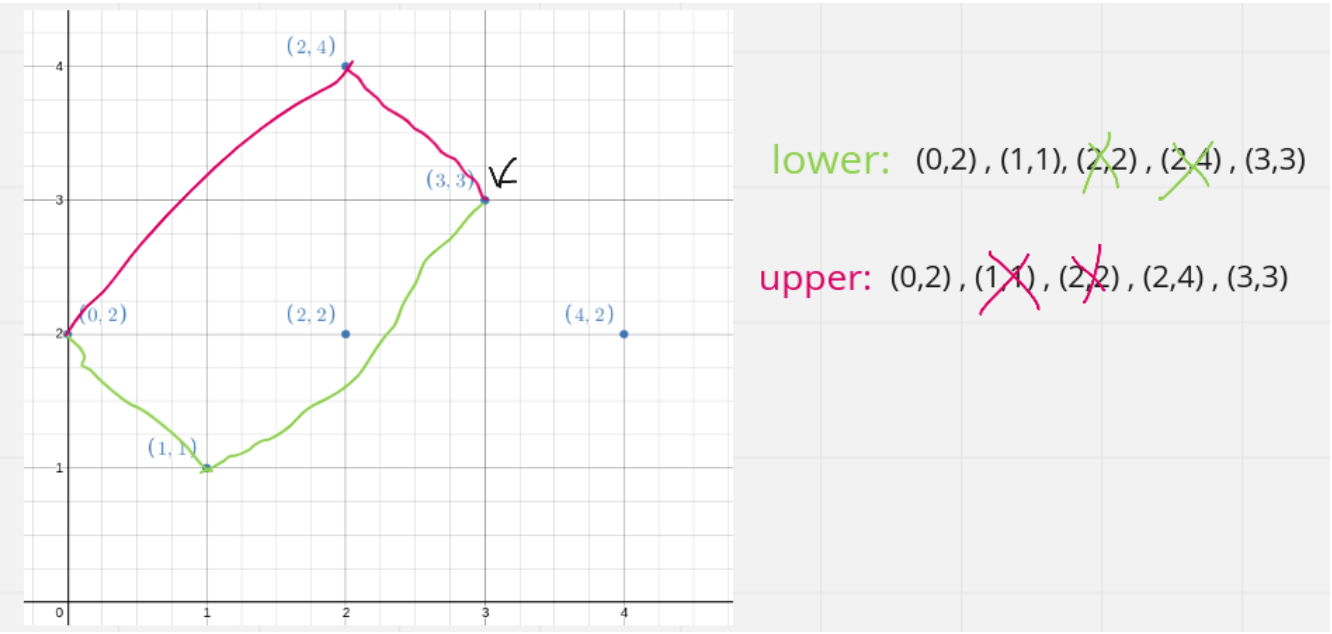
Can we build a lower convex hull with the point (3,3) from {(0,2) , (1,1)}? Yes: Add (3,3)



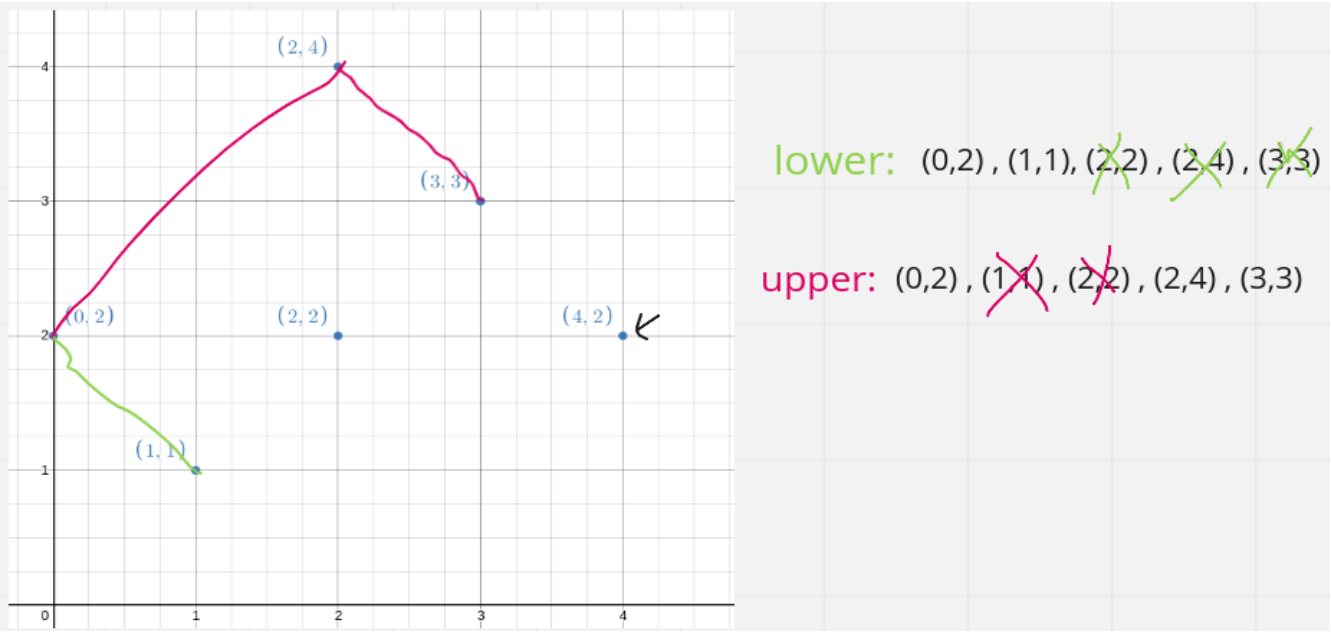
lower: (0,2) , (1,1), (2,2) , (2,4) , (3,3)

upper: (0,2) , (1,1) , (2,2) , (2,4)

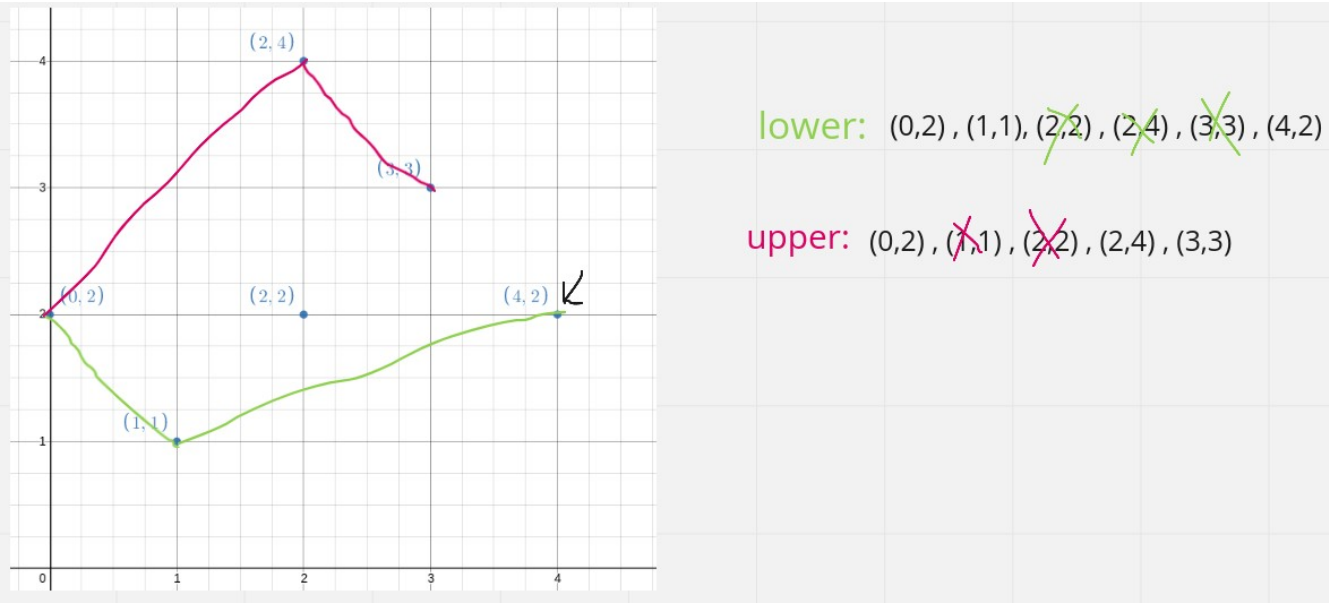
Can we build an upper convex hull with the point (3,3) from {(0,2) , (2,4)}? Yes: Add (3,3)



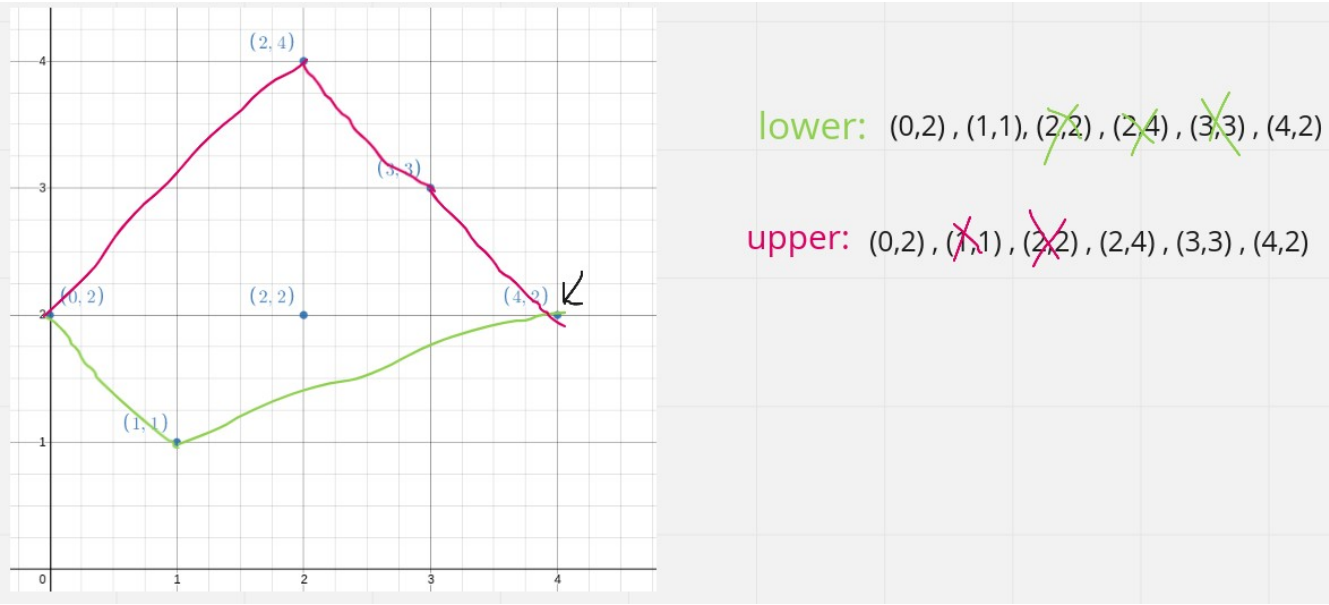
Left most point is (4,2).
Can we build a lower convex hull with the point (4,2) from {(1,1) , (3,3)}? No: Delete (3,3)



Can we build a lower convex hull with the point (4,2) from (1,1)? Yes: Add (4,2)



Can we build an upper convex hull with the point (4,2) from {(2,4) , (3,3)}? Yes: Add (4,2)



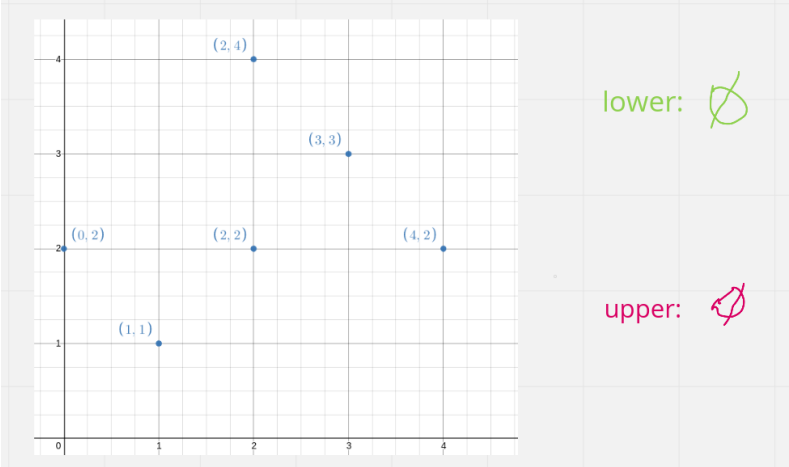
The entire convex hull is: {(0,2), (1,1), (4,2), (3,3) , (2,4)}

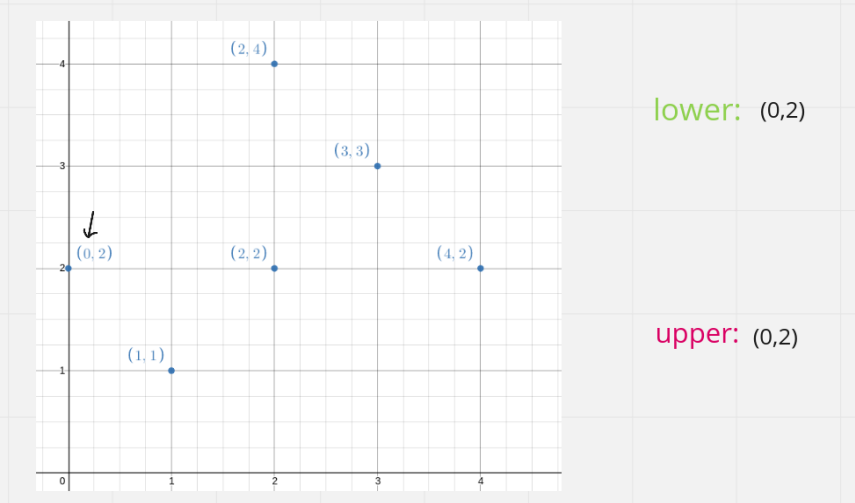
The problem now, how to know if a point is good for the current convex hull or not?

To answer the above question we have to introduce the notions of **cross product** and of **orientation**.

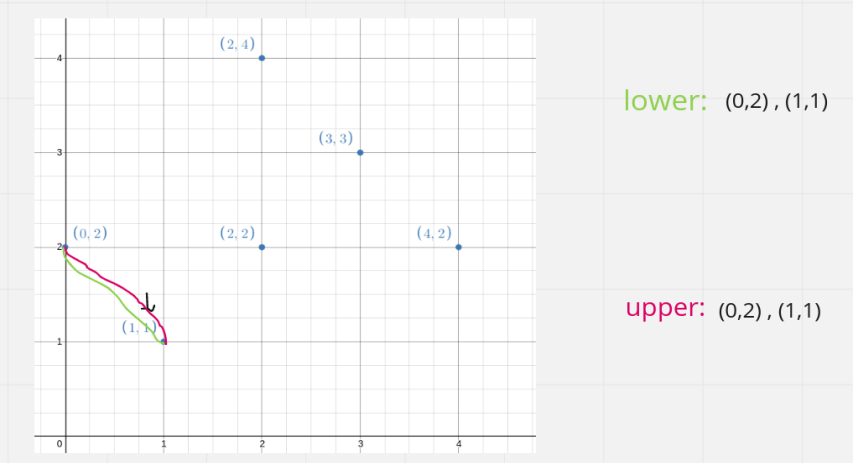
More detailed example explanation

Let's run the same example, but this time we gonna use the concept of orientation.



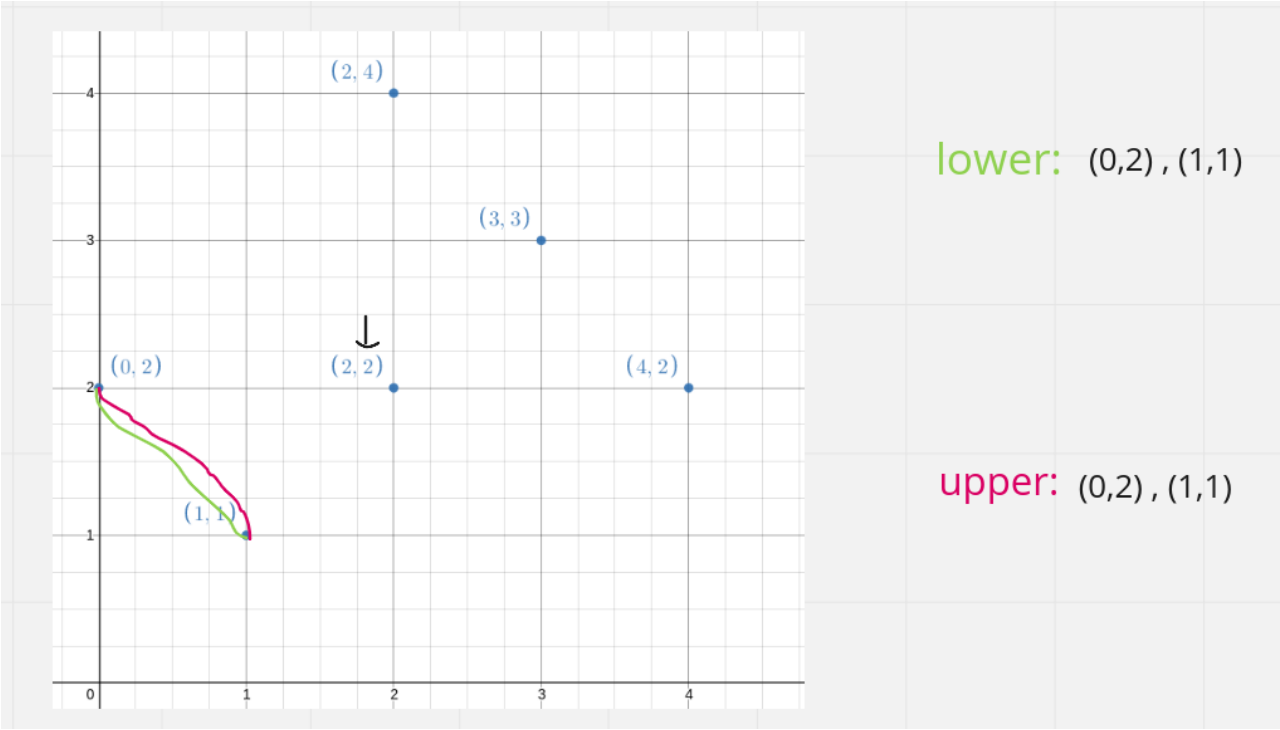


Left most point is (0,2). This the 1st point of both convex hulls

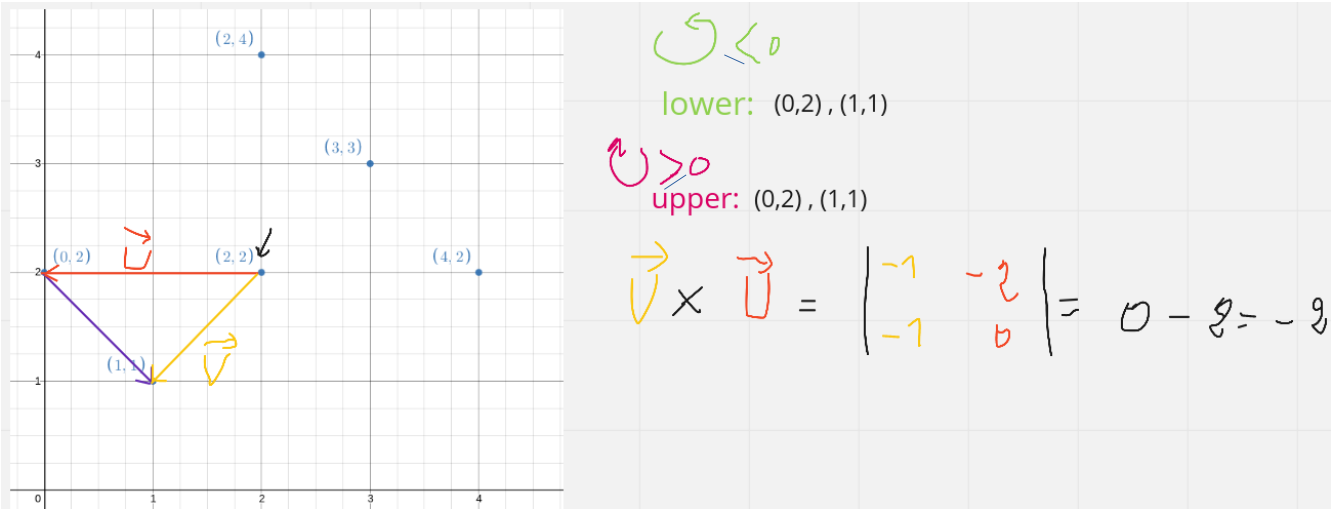


Left most point is (1,1). This the 2nd point of both convex hull

Left most point is (2,2).



Can we build a lower convex hull with the point (2,2) from {(0,2) , (1,1)}?



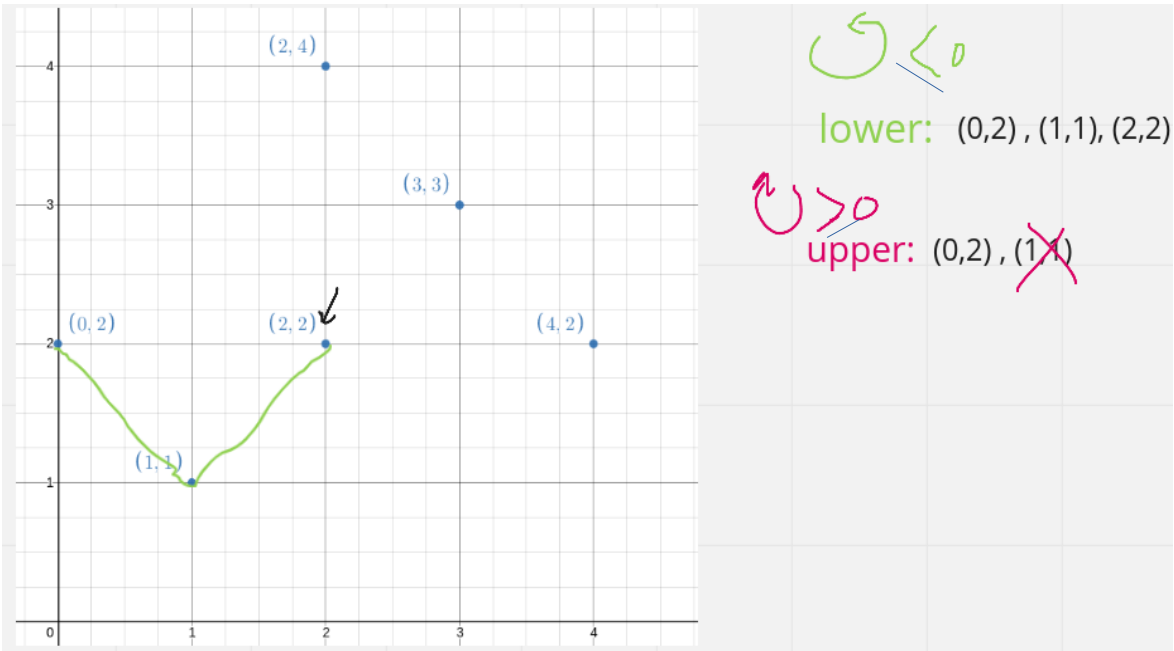
Cross product is negative:

- we can add (2,2) to lower hull,

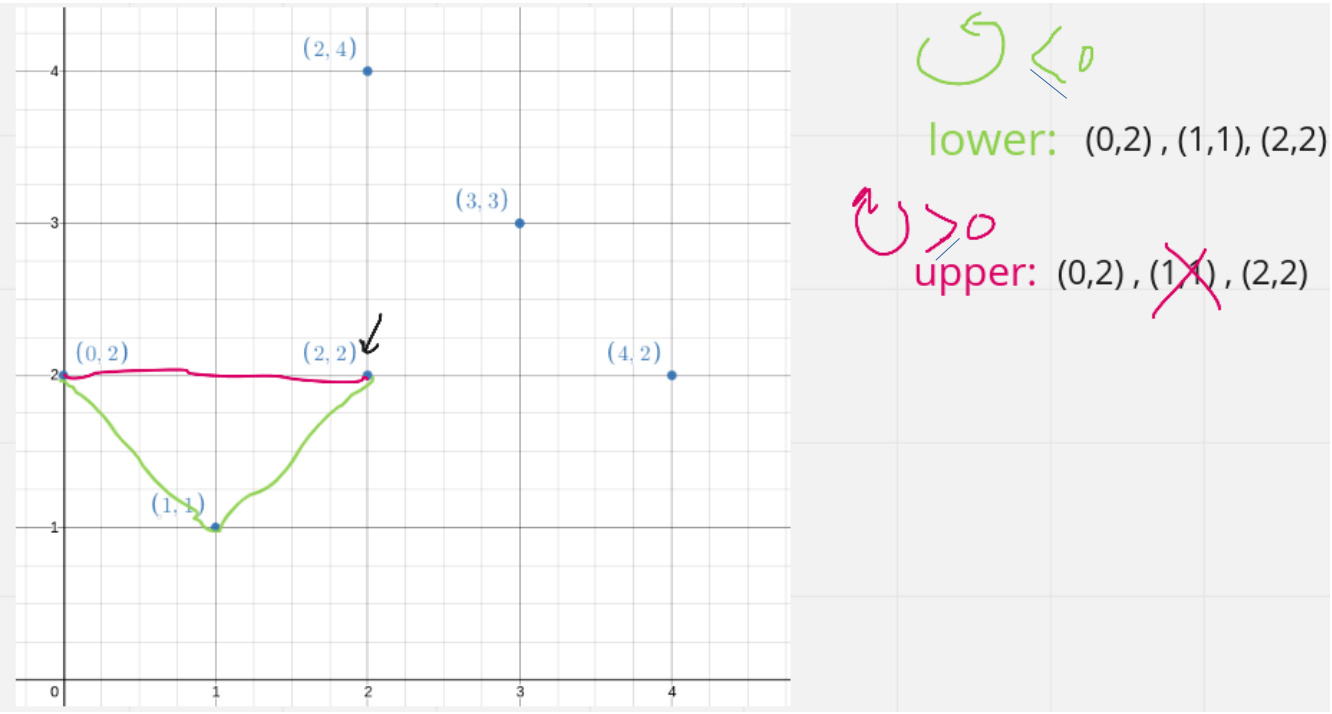
Can we build an upper convex hull with the point (2,2) from {(0,2) , (1,1)}

Cross product is negative:

- we can not add (2,2) to the upper hull, we have to delete (1,1)

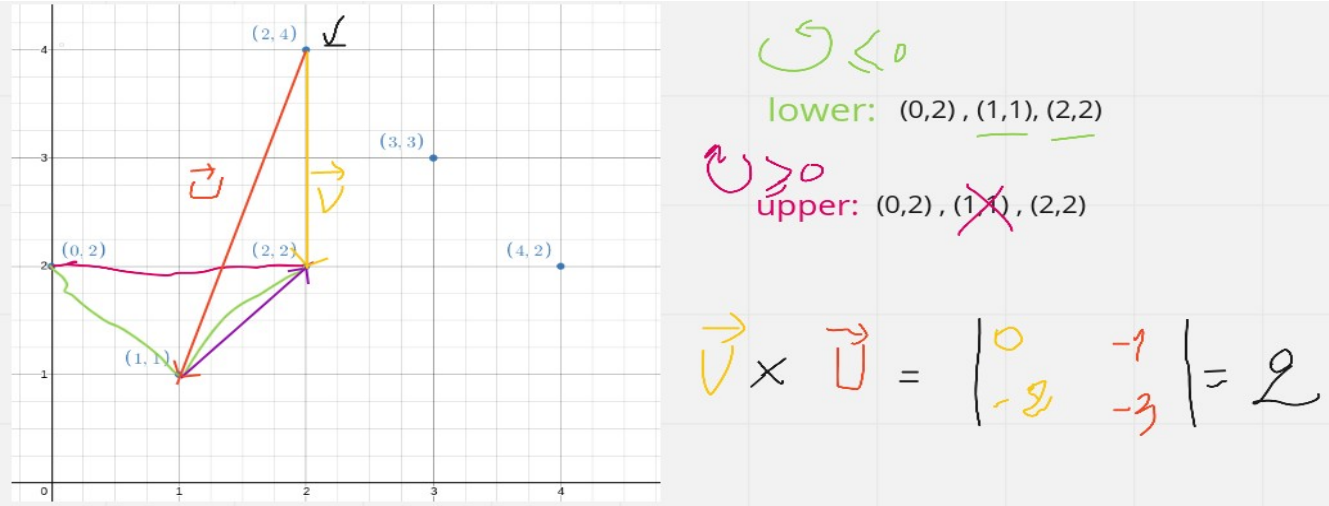


Just one point in upper hull, add (2,2):



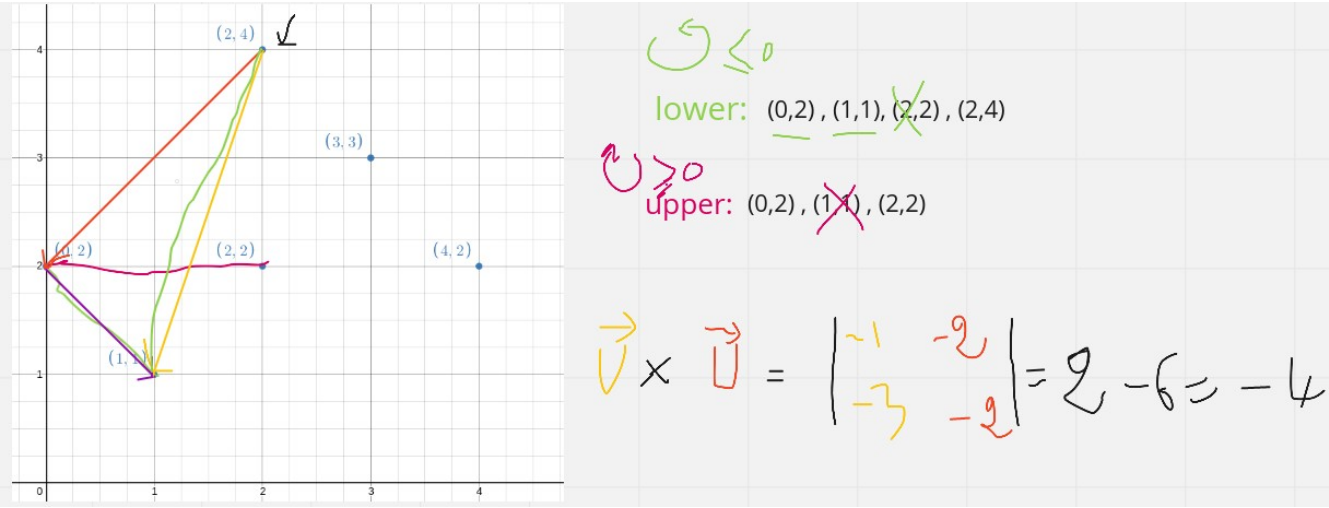
Left most point is (2,4)

Can we build a lower convex hull with the point (2,4) from {(1,1) , (2,2)}?



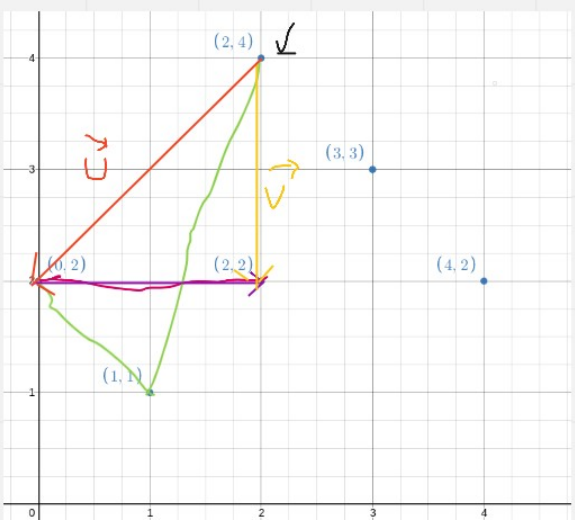
Cross product is positive, we can not add (2,4) to lower hull: delete (2,2)

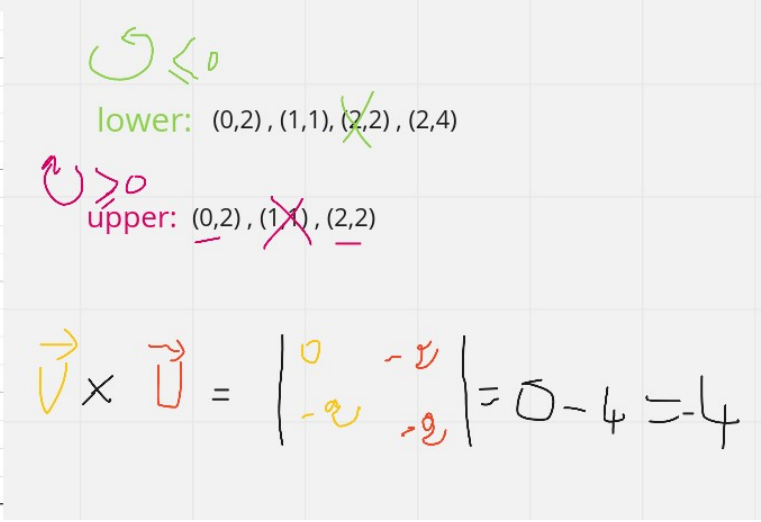
Can we build a lower convex hull with the point (2,4) from {(0,2) , (1,1)}?



Cross product is negative, we add (2,4) to lower hull

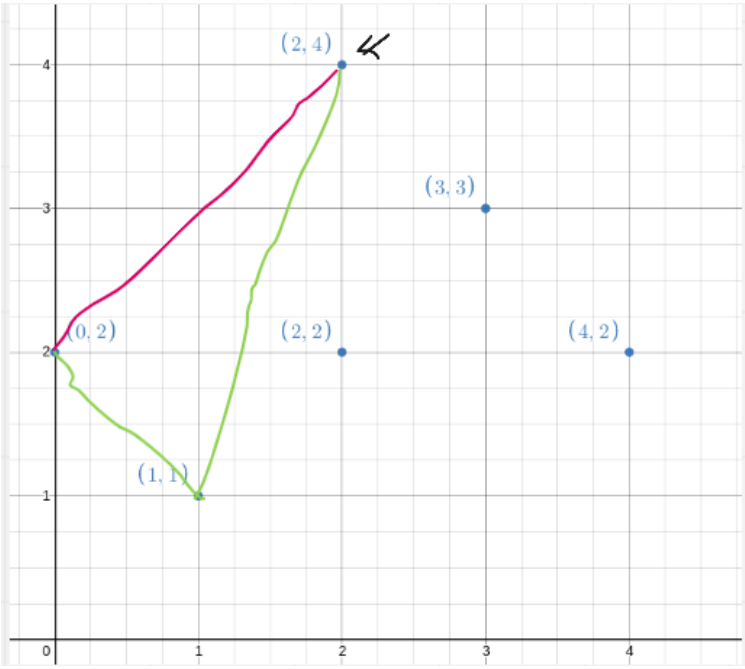
Can we build an upper convex hull with the point (2,4) from {(0,2) , (2,2)}?

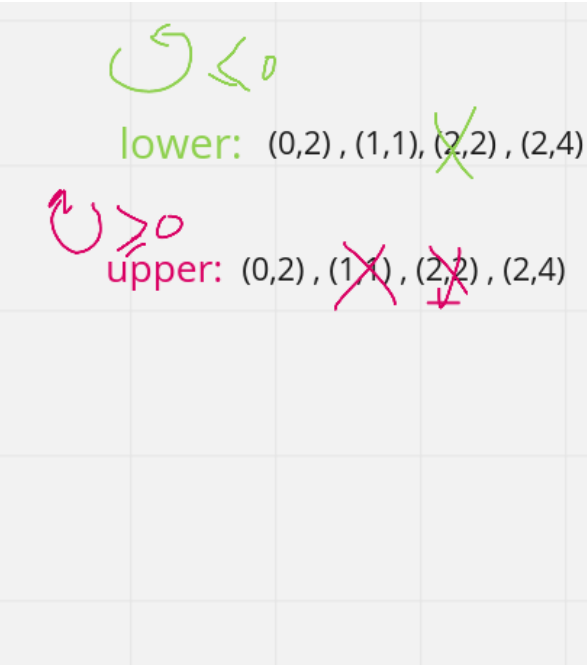




Cross product is negative, we can't add (2,4), delete (2,2)

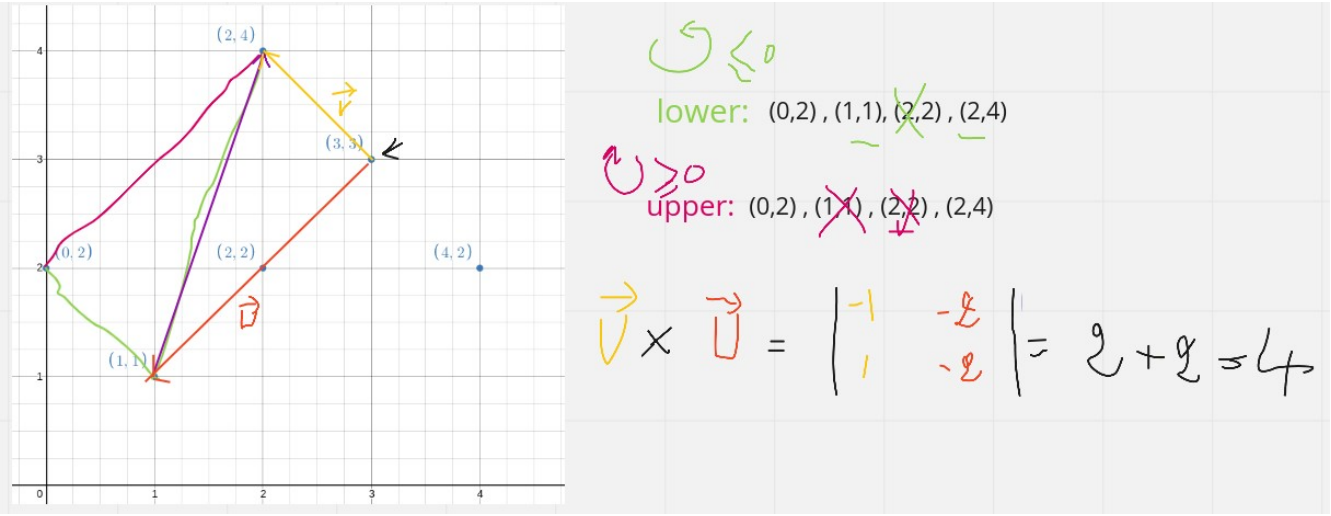
Just one point left in upper hull: add(2,4)





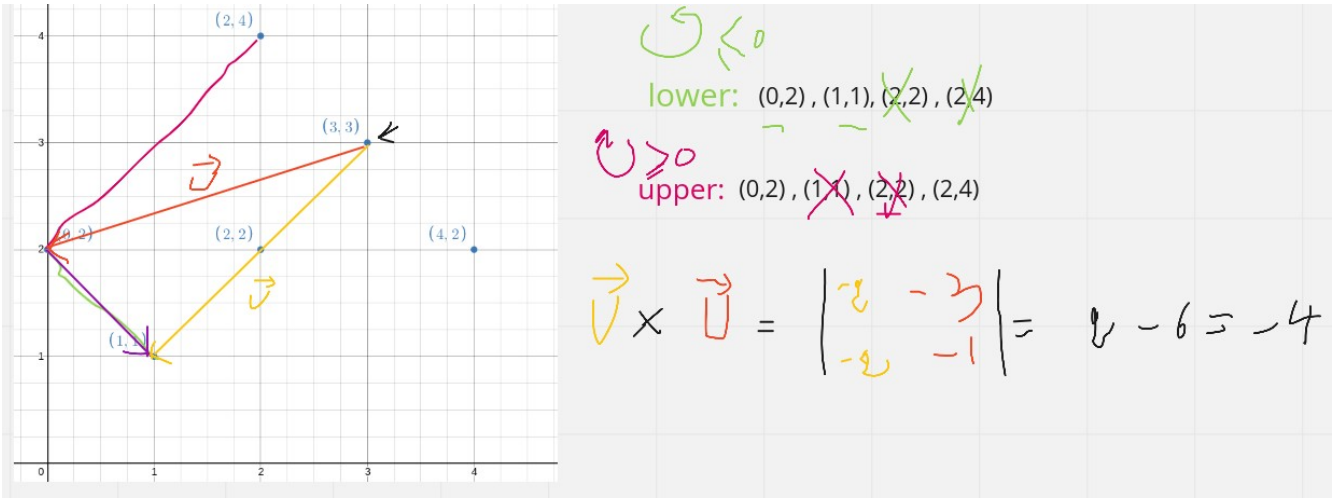
Left most point is (3,3)

Can we build a lower convex hull with the point (3,3) from {(1,1) , (2,4)}?



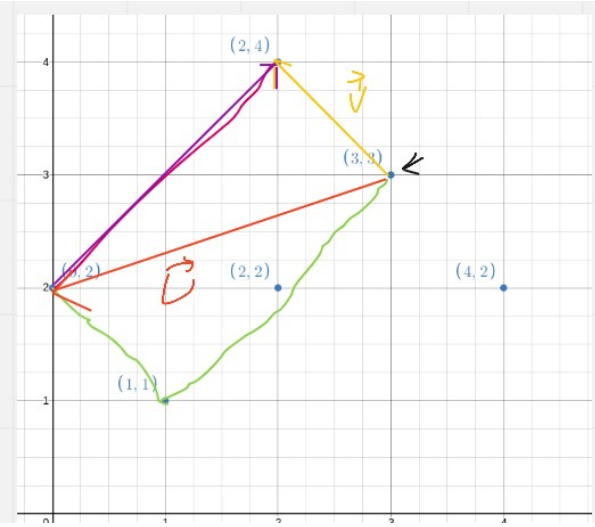
Cross product is positive, we can not add (3,3) to lower hull: delete (2,4)

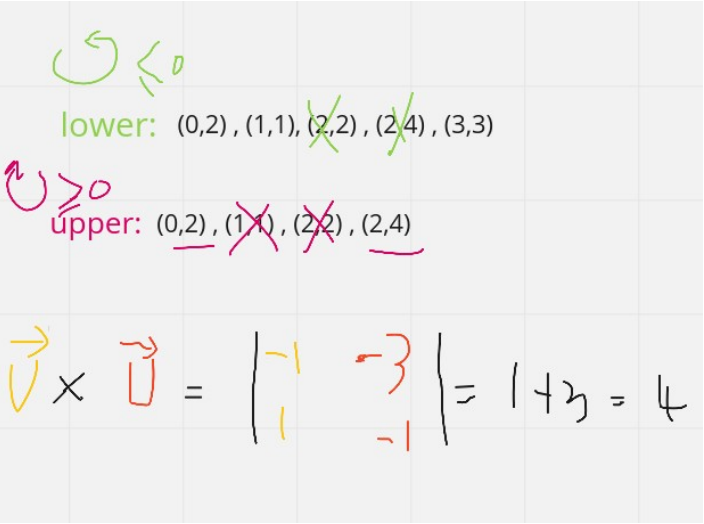
Can we build a lower convex hull with the point (3,3) from {(0,2) , (1,1)}?



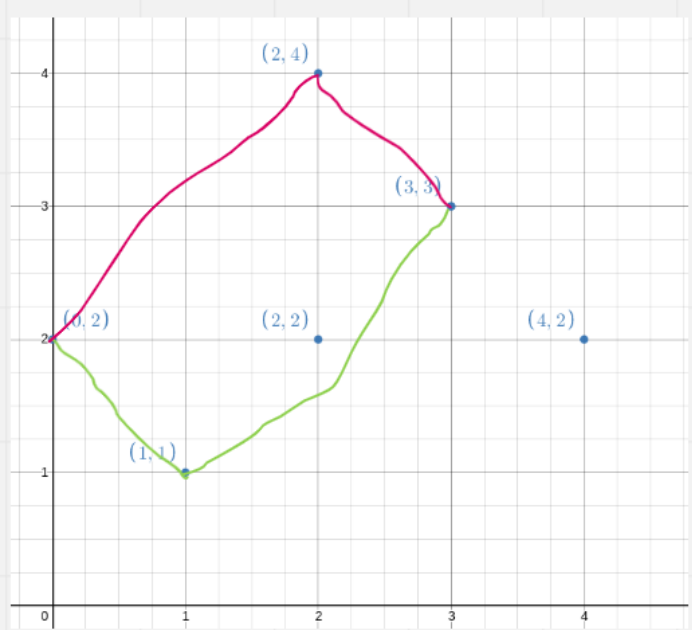
Cross product is negative, we add (3,3) to lower hull

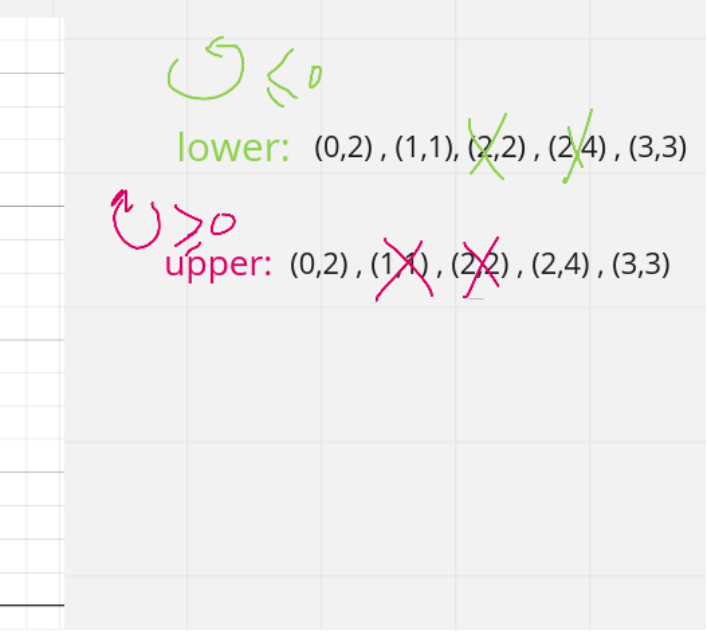
Can we build an upper convex hull with the point (3,3) from {(0,2) , (2,4)}?





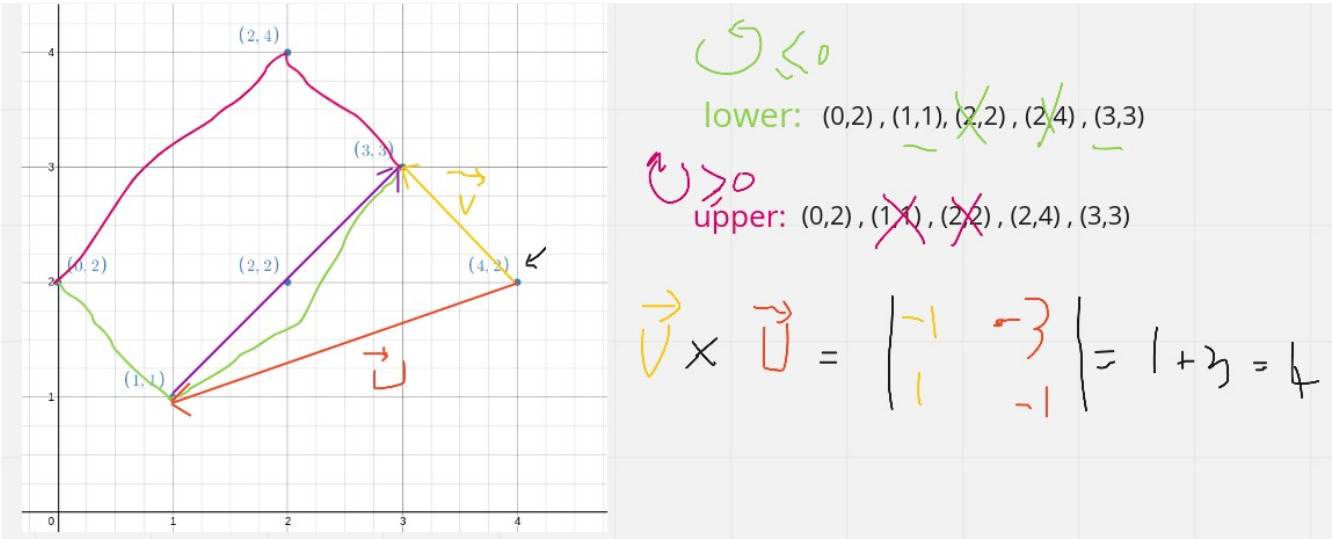
cross product is positive, add (3,3) to upper hull





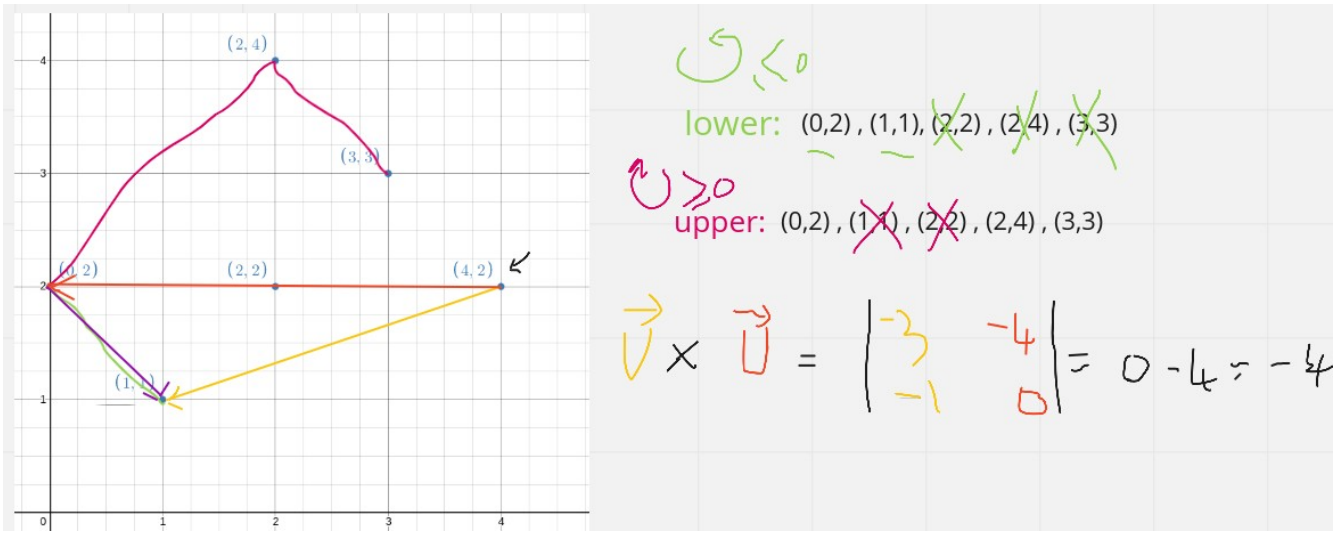
Left most point is (4,2)

Can we build a lower convex hull with the point (4,2) from {(1,1) , (3,3)}?



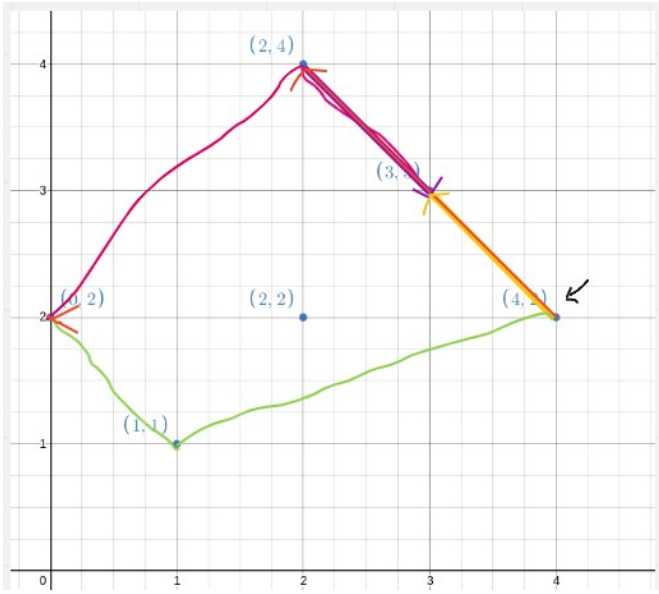
Cross product is positive, we can not add (4,2) to lower hull: delete(3,3)

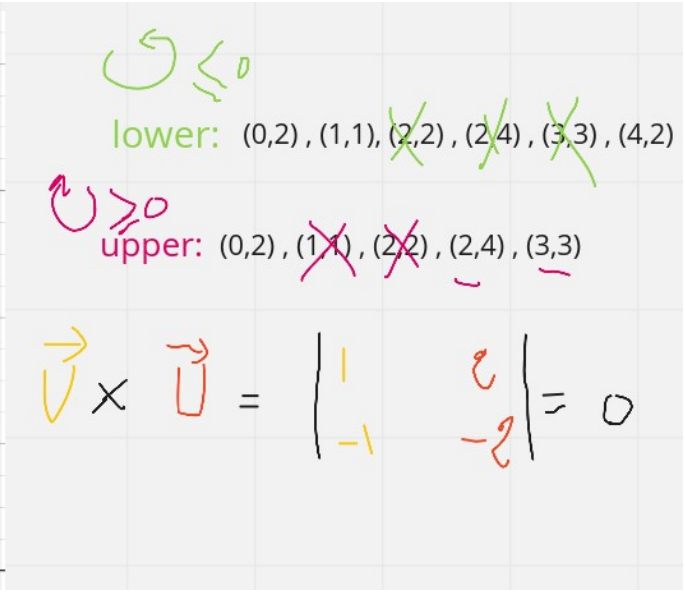
Can we build a lower convex hull with the point (4,2) from {(0,2) , (1,1)}?



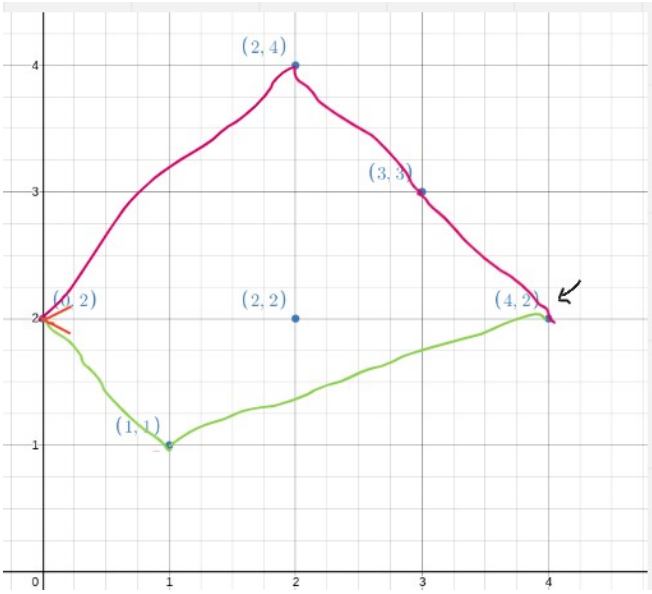
Cross product is negative, we can add (4,2) to lower hull

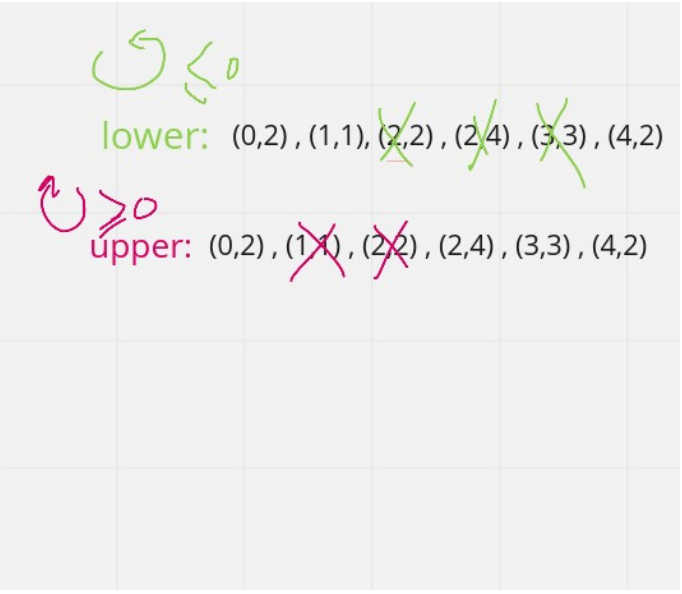
Can we build an upper convex hull with the point (4,2) from {(2,4) , (3,3)}?





Cross product is null, we can add (4,2) to upper hull

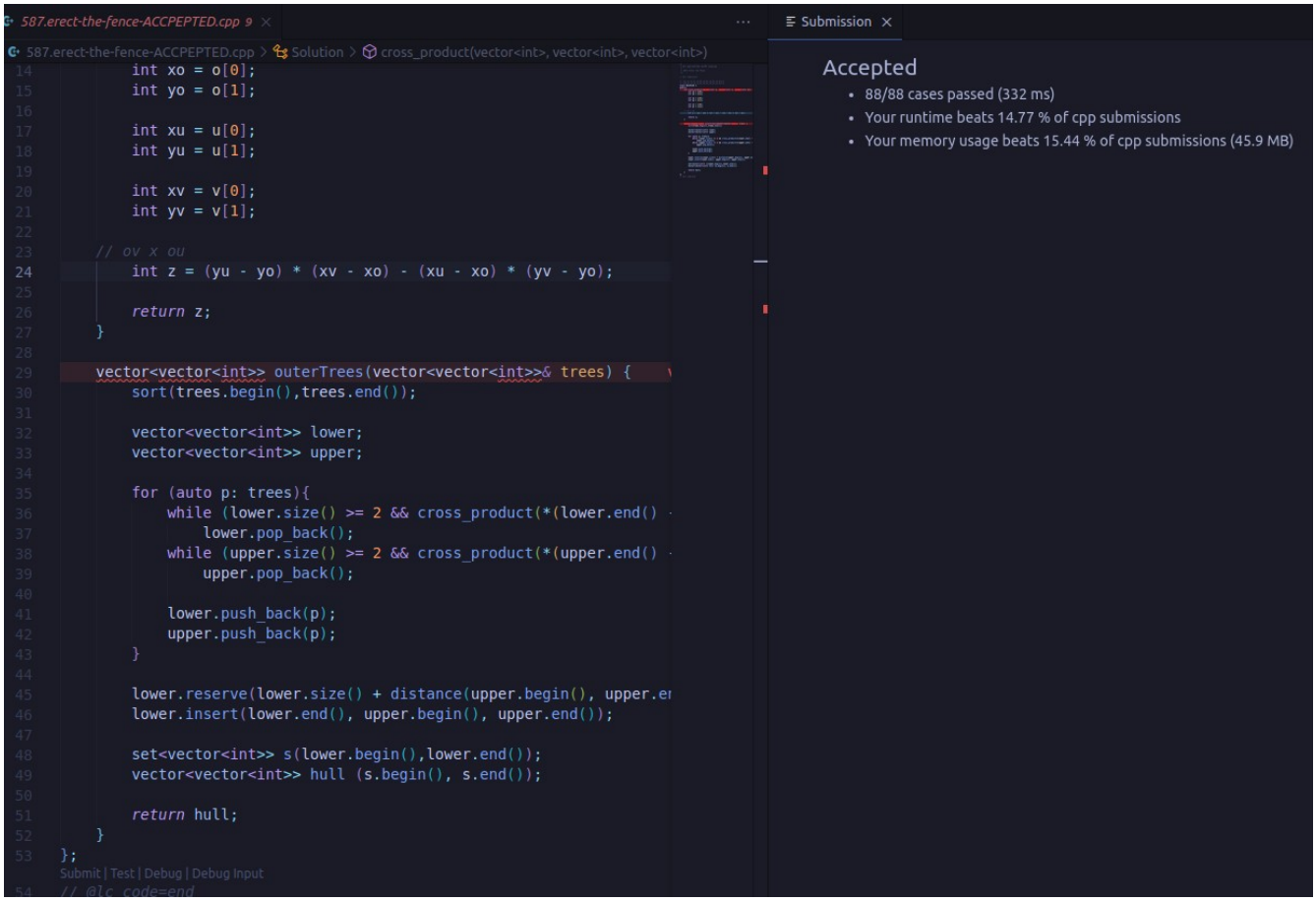




C++: Accepted: Amortizd O(n)

```
/*
 * @lc app=leetcode id=587 lang=cpp
 *
 * [587] Erect the Fence
 */

// @lc code=start
class Solution {
public:
    int cross_product(vector<int> u, vector<int> v, vector<int> o){
        int xo = o[0];
        int yo = o[1];

        int xu = u[0];
        int yu = u[1];

        int xv = v[0];
        int yv = v[1];

    // ov x ou
        int z = (yu - yo) * (xv - xo) - (xu - xo) * (yv - yo);

        return z;
    }
```

```
vector<vector<int>> outerTrees(vector<vector<int>>& trees) {
        sort(trees.begin(),trees.end());

        vector<vector<int>> lower;
        vector<vector<int>> upper;

        for (auto p: trees){
            while (lower.size() >= 2 && cross_product(*(lower.end() - 2), lower.back() , p) > 0)
                lower.pop_back();
            while (upper.size() >= 2 && cross_product(*(upper.end() - 2), upper.back() , p) < 0)
                upper.pop_back();

            lower.push_back(p);
            upper.push_back(p);
        }

        lower.reserve(lower.size() + distance(upper.begin(), upper.end()));
        lower.insert(lower.end(), upper.begin(), upper.end());

        set<vector<int>> s(lower.begin(),lower.end());
        vector<vector<int>> hull (s.begin(), s.end());

        return hull;
    }
};
// @lc code=end
```



587.erect-the-fence-ACCPEPTED.cpp 9

587.erect-the-fence-ACCPEPTED.cpp > Solution > cross_product(vector<int>, vector<int>, vector<int>)

```
        int xo = o[0];
        int yo = o[1];

        int xu = u[0];
        int yu = u[1];

        int xv = v[0];
        int yv = v[1];

    // ov x ou
        int z = (yu - yo) * (xv - xo) - (xu - xo) * (yv - yo);

        return z;
    }

    vector<vector<int>> outerTrees(vector<vector<int>>& trees) {
        sort(trees.begin(),trees.end());

        vector<vector<int>> lower;
        vector<vector<int>> upper;

        for (auto p: trees){
            while (lower.size() >= 2 && cross_product(*(lower.end()
                lower.pop_back();
            while (upper.size() >= 2 && cross_product(*(upper.end()
                upper.pop_back();

            lower.push_back(p);
            upper.push_back(p);
        }

        lower.reserve(lower.size() + distance(upper.begin(), upper.er
        lower.insert(lower.end(), upper.begin(), upper.end());

        set<vector<int>> s(lower.begin(),lower.end());
        vector<vector<int>> hull (s.begin(), s.end());

        return hull;
    }
};
Submit | Test | Debug | Debug Input
// @lc code=end
```

Submission

Accepted

- 88/88 cases passed (332 ms)
- Your runtime beats 14.77 % of cpp submissions
- Your memory usage beats 15.44 % of cpp submissions (45.9 MB)