Problem 3027: Find the Number of Ways to Place People II

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are given a 2D array

points

of size

n x 2

representing integer coordinates of some points on a 2D-plane, where

points[i] = [x

i

, y

i

]

.

We define the

right

direction as positive x-axis (

increasing x-coordinate

) and the

left

direction as negative x-axis (

decreasing x-coordinate

). Similarly, we define the

up

direction as positive y-axis (

increasing y-coordinate

) and the

down

direction as negative y-axis (

decreasing y-coordinate

)

You have to place

n

people, including Alice and Bob, at these points such that there is

exactly one

person at every point. Alice wants to be alone with Bob, so Alice will build a rectangular fence with Alice's position as the

upper left corner

and Bob's position as the

lower right corner

of the fence (

Note

that the fence

might not

enclose any area, i.e. it can be a line). If any person other than Alice and Bob is either

inside

the fence or

on

the fence, Alice will be sad.

Return

the number of

pairs of points

where you can place Alice and Bob, such that Alice

does not

become sad on building the fence

.

Note

that Alice can only build a fence with Alice's position as the upper left corner, and Bob's position as the lower right corner. For example, Alice cannot build either of the fences in the picture below with four corners

(1, 1)

,

(1, 3)

,

(3, 1)

, and

(3, 3)

, because:

With Alice at

(3, 3)

and Bob at

(1, 1)

, Alice's position is not the upper left corner and Bob's position is not the lower right corner of the fence.

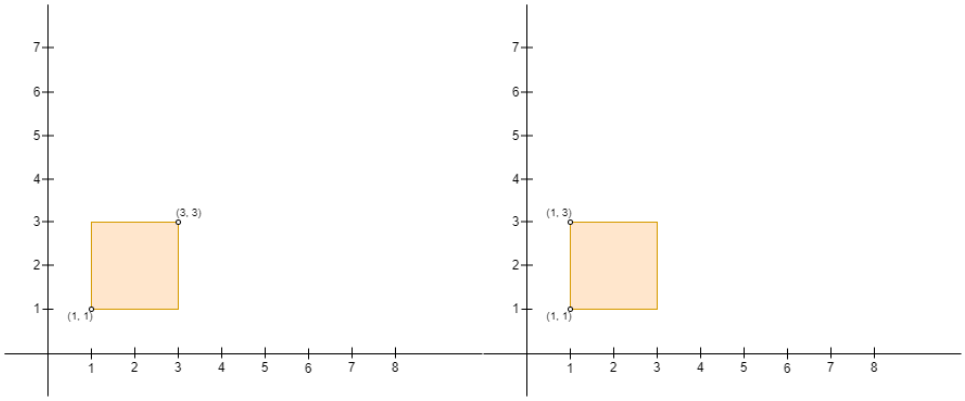
With Alice at

(1, 3)

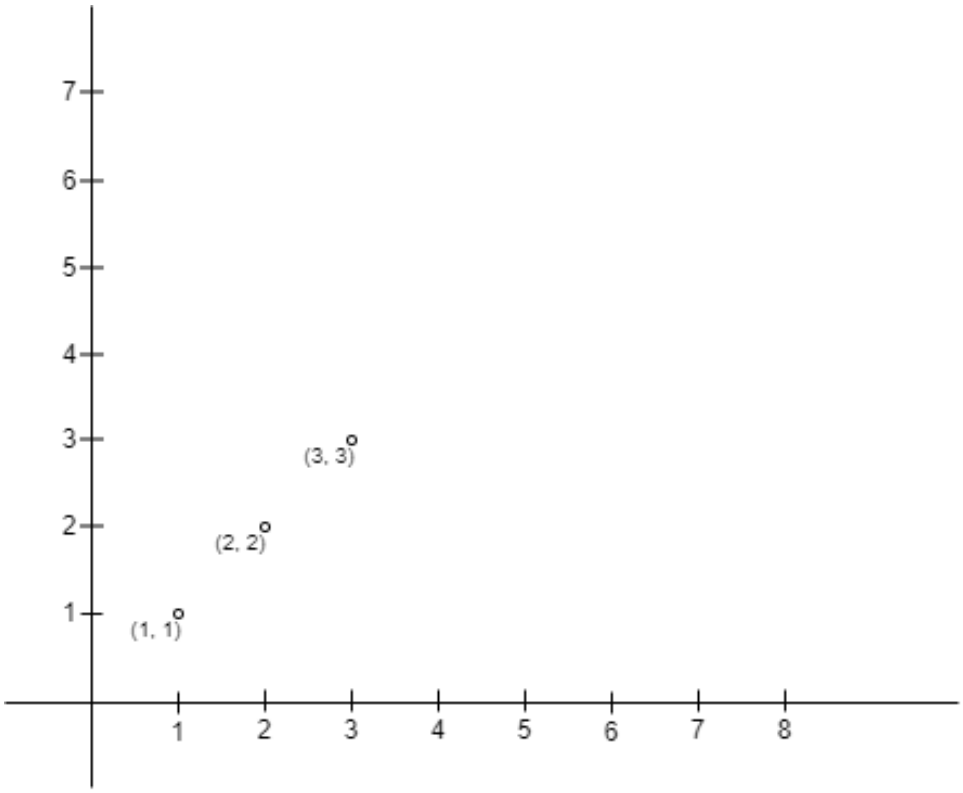
and Bob at

(1, 1)

(as the rectangle shown in the image instead of a line), Bob's position is not the lower right corner of the fence.



Example 1:



Input:

points = [[1,1],[2,2],[3,3]]

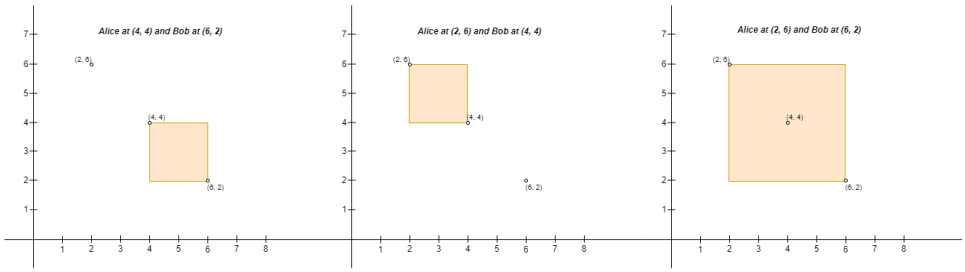
Output:

0

Explanation:

There is no way to place Alice and Bob such that Alice can build a fence with Alice's position as the upper left corner and Bob's position as the lower right corner. Hence we return 0.

Example 2:



Input:

points = [[6,2],[4,4],[2,6]]

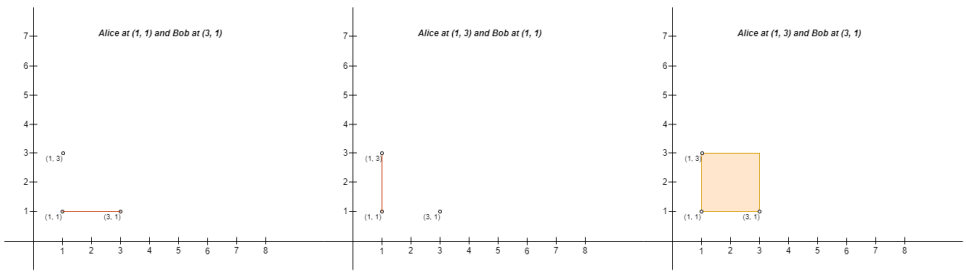
Output:

2

Explanation:

There are two ways to place Alice and Bob such that Alice will not be sad: - Place Alice at (4, 4) and Bob at (6, 2). - Place Alice at (2, 6) and Bob at (4, 4). You cannot place Alice at (2, 6) and Bob at (6, 2) because the person at (4, 4) will be inside the fence.

Example 3:



Input:

points = [[3,1],[1,3],[1,1]]

Output:

2

Explanation:

There are two ways to place Alice and Bob such that Alice will not be sad: - Place Alice at (1, 1) and Bob at (3, 1). - Place Alice at (1, 3) and Bob at (1, 1). You cannot place Alice at (1, 3) and Bob at (3, 1) because the person at (1, 1) will be on the fence. Note that it does not matter if the fence encloses any area, the first and second fences in the image are valid.

Constraints:

2 <= n <= 1000

points[i].length == 2

-10

9

<= points[i][0], points[i][1] <= 10

9

All

points[i]

are distinct.

Code Snippets

C++:

```
class Solution {
public:
int numberOfPairs(vector<vector<int>>& points) {

}
};
```

Java:

```
class Solution {
public int numberOfPairs(int[][] points) {

}
}
```

Python3:

```
class Solution:
def numberOfPairs(self, points: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def numberOfPairs(self, points):
"""
:type points: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[][]} points
* @return {number}
*/
var numberOfPairs = function(points) {

};
```

TypeScript:

```
function numberOfPairs(points: number[][]): number {

};
```

C#:

```
public class Solution {
public int NumberOfPairs(int[][] points) {

}
}
```

C:

```
int numberOfPairs(int** points, int pointsSize, int* pointsColSize) {

}
```

Go:

```
func numberOfPairs(points [][]int) int {

}
```

Kotlin:

```
class Solution {
fun numberOfPairs(points: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func numberOfPairs(_ points: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn number_of_pairs(points: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[][]} points
# @return {Integer}
def number_of_pairs(points)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $points
* @return Integer
*/
function numberOfPairs($points) {

}
}
```

Dart:

```
class Solution {
int numberOfPairs(List<List<int>> points) {

}
}
```

Scala:

```
object Solution {
def numberOfPairs(points: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec number_of_pairs(points :: [[integer]]) :: integer
def number_of_pairs(points) do

end
end
```

Erlang:

```
-spec number_of_pairs(Points :: [[integer()]]) -> integer().
number_of_pairs(Points) ->
.
```

Racket:

```
(define/contract (number-of-pairs points)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Find the Number of Ways to Place People II
* Difficulty: Hard
* Tags: array, math, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
int numberOfPairs(vector<vector<int>>& points) {

}
};
```

Java Solution:

```
/**
* Problem: Find the Number of Ways to Place People II
* Difficulty: Hard
* Tags: array, math, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int numberOfPairs(int[][] points) {

}
}
```

Python3 Solution:

```
"""
Problem: Find the Number of Ways to Place People II
Difficulty: Hard
Tags: array, math, sort

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def numberOfPairs(self, points: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def numberOfPairs(self, points):
"""
:type points: List[List[int]]
:rtype: int
```

```
"""
```

JavaScript Solution:

```
/**
* Problem: Find the Number of Ways to Place People II
* Difficulty: Hard
* Tags: array, math, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[][]} points
* @return {number}
*/
var numberOfPairs = function(points) {

};
```

TypeScript Solution:

```
/**
* Problem: Find the Number of Ways to Place People II
* Difficulty: Hard
* Tags: array, math, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function numberOfPairs(points: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Find the Number of Ways to Place People II
* Difficulty: Hard
* Tags: array, math, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int NumberOfPairs(int[][] points) {

}
}
```

C Solution:

```
/*
* Problem: Find the Number of Ways to Place People II
* Difficulty: Hard
* Tags: array, math, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

int numberOfPairs(int** points, int pointsSize, int* pointsColSize) {

}
```

Go Solution:

```
// Problem: Find the Number of Ways to Place People II
// Difficulty: Hard
// Tags: array, math, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach
```

```
func numberOfPairs(points [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun numberOfPairs(points: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func numberOfPairs(_ points: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Find the Number of Ways to Place People II
// Difficulty: Hard
// Tags: array, math, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn number_of_pairs(points: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} points
# @return {Integer}
def number_of_pairs(points)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $points
* @return Integer
*/
function numberOfPairs($points) {

}
}
```

Dart Solution:

```
class Solution {
int numberOfPairs(List<List<int>> points) {

}
}
```

Scala Solution:

```
object Solution {
def numberOfPairs(points: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec number_of_pairs(points :: [[integer]]) :: integer
def number_of_pairs(points) do

end
end
```

Erlang Solution:

```
-spec number_of_pairs(Points :: [[integer()]]) -> integer().
number_of_pairs(Points) ->
.
```

Racket Solution:

```
(define/contract (number-of-pairs points)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```