3027. Find the Number of Ways to Place People II

Hard Geometry Array Math Enumeration Sorting

Problem Description

In this problem, we're given a 2D array `points` representing coordinates of certain points on a plane. Each point is meant to be occupied by one individual, and there are two particular individuals of interest: Alice and Bob. The task is to figure out how many ways Alice can be positioned at a point that will serve as the upper left corner of a rectangular fence, and Bob can be positioned at a point that will serve as the lower right corner of the same fence, without any other person being inside or on the boundary of this fence.

The essence of this problem revolves around the concept that we can only place Alice and Bob in such a way that Alice is above and to the left of Bob (based on their x and y coordinates). Thus, no individual should have coordinates such that they fall within or on the boundary of the rectangle defined by Alice's and Bob's locations.

To summarize, the challenge is to find all pairs `(Alice, Bob)` that allow the fence to be built without enclosing or including any other individuals on its boundary or within its area.

Intuition

Approaching this problem begins with the understanding that if we have Alice's position fixed as the upper left corner, then Bob must be positioned diagonally across from her at the lower right corner. This means Bob's x-coordinate must be greater than Alice's x-coordinate, and his y-coordinate must be less than Alice's y-coordinate, to form a valid rectilinear fence.

The intuition behind the solution hinges on efficiently checking potential `Bob` points for every potential `Alice` point. We must avoid counting any point that would be inside or on the proposed fence boundary.

Sorting `points` by their x-coordinate distinctly helps in the forward search since it naturally orders Bobs to the right of Alice. The y-coordinates are sorted inversely to deal with ensuring that Alice's y-coordinate is more than Bob's. After sorting, iterate through each Alice candidate while keeping track of the maximal y-coordinate observed for Bob that is less than or equal to Alice's y-coordinate—the idea being that we don't want any point that sits on or inside the imaginary fence.

For each potential Alice, we search for potential Bobs. If we find a Bob with a y-coordinate less than Alice's but greater than any we've previously encountered, we know we've found a pair that can build a fence without including another individual, so we increment our pairs count. The process is repeated until all viable Alice and Bob pairs are accounted for.

Solution Approach

To implement the solution, the following steps are taken:

1. **Sort the Points**: Firstly, we sort the `points` array. This is done based on the x-coordinates in ascending order so that potential Bobs are to the right of Alice. For points with the same x-coordinate, we sort based on the y-coordinate in descending order so that potential Alices are above Bobs. This is accomplished with the line:

   ```
   points.sort(key=lambda x: (x[0], -x[1]))
   ```

   This step is crucial as it aligns all points to a common reference that respects the direction constraints (right and down) mentioned in the problem statement.

2. **Initialize Counter**: We have an `ans` variable set to 0, which will keep track of the number of valid pairs (Alice, Bob) found.

3. **Iterate Through Points**: We iterate through the sorted `points` list, treating each point as a potential Alice's position.

4. **Track Maximum y-coordinate**: As we iterate through potential Bob positions (to the right of Alice), we maintain a `max_y` variable set to negative infinity to track the highest y-coordinate of points encountered that could serve as a Bob's position and still not invalidate the fence.

5. **Check and Update Fence Conditions**: For each potential Bob, we check if any previous point's y-coordinate (greater than `max_y` but less than or equal to current Alice's y-coordinate) can be used as Bob's position. This is done with the following check:

   ```
   if max_y < y2 <= y1:
       max_y = y2
       ans += 1
   ```

   This checks whether the current point can be a Bob without any other point on the fence. If it can, we update `max_y` to this new value and increment `ans`.

6. **Return the Count**: After iterating over all points, the `ans` value will contain the total number of valid (Alice, Bob) pairs that can be placed such that no other point will be inside or on the fence. This value is returned as the final answer:

   ```
   return ans
   ```

Algorithm and Data Structures: The solution employs a simple sorting algorithm along with a linear iteration and tracking of maximum y-values. There was no need for complex data structures as the sorted list itself was sufficient to find all possible pairs.

Patterns Used: What stands out is the use of the two-pointer approach, in a way, where we keep a moving pointer for Alice and another moving pointer for Bob, within the sorted array. This pattern is efficient because it omits the need for nested iterations over the entire set of points for every Alice, reducing the potential time complexity.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Consider the following points representing coordinates on a plane: `points = [[1, 3], [3, 0], [3, 4], [2, 2]]`

Following the steps of the solution:

1. **Sort the Points**: After sorting based on the given key, we have the points array sorted as `[[1, 3], [2, 2], [3, 4], [3, 0]]`. Notice how the points are sorted left to right first, and then for the same x-coordinate, they are sorted from top to bottom.

2. **Initialize Counter**: We set `ans = 0`, ready to count valid pairs.

3. **Iterate Through Points**: We treat each point as potential Alice's position and move from left to right, top to bottom.

4. **Track Maximum y-coordinate**: We start with `max_y = -inf` as we haven't encountered any Bobs yet.

5. **Check and Update Fence Conditions**:
   - Firstly, we consider point `[1, 3]` as a potential Alice. The max_y is still `-inf`.
   - We move to the next point `[2, 2]`. Could this be Bob for Alice `[1, 3]`? Yes, because `max_y < 2 <= 3`, so we update `max_y = 2`, and increment `ans` to 1.
   - Then, we consider point `[3, 4]` - this cannot be Bob for the current Alice because its y-coordinate is not less than Alice's.
   - Lastly, we check point `[3, 0]`. This could potentially serve as Bob to Alice `[1, 3]`, as it satisfies the condition `max_y < 0 <= 3`. Therefore, we update the `max_y` to 0, and increment `ans` to 2.

   At this moment, if we try to find a Bob for Alice `[2, 2]`, no point to its right has a lesser y-coordinate. Similarly, points `[3, 4]` and `[3, 0]` serve as potential Alice positions but have no valid Bobs to the right.

6. **Return the Count**: Since we only found valid (Alice, Bob) configurations with the initial Alice `[1, 3]`, the final `ans` is 2.

The result tells us there are two distinct ways to position Alice and Bob for the creation of a fence that does not enclose or include others on its boundary. The pairs found were `([1, 3], [3, 0])` and `([1, 3], [2, 2])`.

Solution Implementation

Python

```python
from math import inf

class Solution:
    def number_of_pairs(self, points: List[List[int]]) -> int:
        # Sort the points by their x-coordinate, and then by their y-coordinate in descending order
        points.sort(key=lambda point: (point[0], -point[1]))
        ans = 0  # Initialize the number of valid pairs to zero

        # Iterate over each point
        for i, (_, y1) in enumerate(points):
            max_y = -inf  # Initialize the maximum y-value for the points considered so far

            # Compare with subsequent points to check for pairs
            for _, y2 in points[i + 1:]:
                # If the y-value of the current point is greater than max_y
                # and not greater than y1, this is a valid pair
                if max_y < y2 <= y1:
                    max_y = y2  # Update max_y to the current y-value
                    ans += 1  # Increment the count of valid pairs

        return ans  # Return the total number of valid pairs found
```

Java

```java
class Solution {
    public int numberOfPairs(int[][] points) {
        // Sort the array of points in ascending order by x-coordinates,
        // and in descending order by y-coordinates if x-coordinates are the same
        Arrays.sort(points, (point1, point2) -> point1[0] == point2[0] ? point2[1] - point1[1] : point1[0] - point2[0]);

        int count = 0; // Initialize the count of valid pairs
        int numberOfPoints = points.length; // Total number of points
        final int INFINITY = 1 << 30; // Representation of negative infinity

        // Iterate over each point to check for valid pairs
        for (int i = 0; i < numberOfPoints; ++i) {
            int y1 = points[i][1]; // Get the y-coordinate of the current point

            int maxY = -INFINITY; // Set maxY as negative infinity initially

            // Iterate over the points that come after the current point
            for (int j = i + 1; j < numberOfPoints; ++j) {
                int y2 = points[j][1]; // Get the y-coordinate of the next point

                // Check if the current maxY is less than y2 and y2 is less than or equal to y1
                // If so, this forms a valid pair and update maxY and increment count
                if (maxY < y2 && y2 <= y1) {
                    maxY = y2; // Update maxY
                    ++count; // Increment the number of valid pairs
                }
            }
        }

        return count; // Return the total number of valid pairs
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for sort
#include <climits> // Include climits header for INT_MIN

class Solution {
public:
    int numberOfPairs(std::vector<std::vector<int>>& points) {
        // Sort the points in non-decreasing order of their x-values,
        // in case of a tie sort by the y-values in decreasing order.
        std::sort(points.begin(), points.end(), [](const std::vector<int>& a, const std::vector<int>& b) {
            return a[0] < b[0] || (a[0] == b[0] && a[1] > a[1]);
        });

        int n = points.size(); // The total number of points
        int ans = 0; // Initialize the count of pairs

        for (int i = 0; i < n; ++i) {
            int y1 = points[i][1]; // Get the y-value of the current point
            int maxY = INT_MIN; // Initialize maxY with the smallest possible integer

            // Iterate through all points that come after the current point
            for (int j = i + 1; j < n; ++j) {
                int y2 = points[j][1]; // Get the y-value of the next point
                // If maxY is less than y2, and y2 is less than or equal to y1,
                // it means we found a pair where the second point can potentially
                // form a pair with the first, based on the y-values.
                if (maxY < y2 && y2 <= y1) {
                    maxY = y2; // Update maxY to the new found y-value
                    ++ans; // Increment the count of pairs
                }
            }
        }

        return ans; // Return the total number of pairs found
    }
};
```

TypeScript

```typescript
function numberOfPairs(points: number[][]): number {
    // Sort the points array. First by x-coordinate and then by y-coordinate in descending order if x is the same
    points.sort((pointA, pointB) => pointA[0] === pointB[0] ? pointB[1] - pointA[1] : pointA[0] - pointB[0]);

    const totalPoints = points.length; // Total number of points
    let pairCount = 0; // Initialize pairs count

    // Iterate over each point
    for (let i = 0; i < totalPoints; ++i) {
        const y1 = points[i][1]; // Get the y-coordinate of the current point
        let maxY = -Infinity; // Initialize the max Y seen so far for the pairs

        // Iterate over the points after the current one to find pairs
        for (let j = i + 1; j < totalPoints; ++j) {
            const y2 = points[j][1]; // Get the y-coordinate of the next point

            // Check if the current y-coordinate is within the required range and update maxY
            if (maxY < y2 && y2 <= y1) {
                maxY = y2;
                ++pairCount; // Increment the count of valid pairs
            }
        }
    }

    return pairCount; // Return the total number of valid pairs found
}
```

```python
from math import inf

class Solution:
    def number_of_pairs(self, points: List[List[int]]) -> int:
        # Sort the points by their x-coordinate, and then by their y-coordinate in descending order
        points.sort(key=lambda point: (point[0], -point[1]))
        ans = 0  # Initialize the number of valid pairs to zero

        # Iterate over each point
        for i, (_, y1) in enumerate(points):
            max_y = -inf  # Initialize the maximum y-value for the points considered so far

            # Compare with subsequent points to check for pairs
            for _, y2 in points[i + 1:]:
                # If the y-value of the current point is greater than max_y
                # and not greater than y1, this is a valid pair
                if max_y < y2 <= y1:
                    max_y = y2  # Update max_y to the current y-value
                    ans += 1  # Increment the count of valid pairs

        return ans  # Return the total number of valid pairs found
```

Time and Space Complexity

The time complexity of the code provided is not `O(1)`. It first sorts the list, which has `O(n log n)` complexity where `n` is the length of the `points` list. Then it uses a nested loop to iterate through the `points` list, leading to a worst-case scenario of `O(n^2)` complexity since for each point, it potentially compares it to all other points. Therefore, the overall time complexity is `O(n^2)` due to the nested loop after sorting.

The space complexity is `O(1)` (ignoring the space taken up by the input and the sort implementation), as the code only uses a fixed amount of additional space – a few variables like `ans` and `max_y`, which do not scale with the size of the input.