3025. Find the Number of Ways to Place People I

Medium Geometry Array Math Enumeration Sorting

Problem Description

The goal of the given problem is to determine the number of ways Alice and Bob can be positioned at two points on a 2D plane, without making Alice sad. We are supplied with a 2D array `points` which provides the coordinate pair for each point on the plane. The rules for positioning Alice and Bob are as follows:

- Alice wants to build a rectangular fence, with her position being the upper left corner and Bob's position being the lower right corner.
- No other person can be inside or on the fence, as this will make Alice sad.
- There must be exactly one person at each point, and Alice can only be happy if there are no other people apart from her and Bob either inside or on the borders of the fence that she builds.

It is crucial to understand that Alice can build a fence only if her position is the upper left corner and Bob's is the lower right corner, which means Alice must be at a point with lower x and y coordinates than Bob.

To solve this problem, we need to find all combinations of points where Alice and Bob can be placed that allow a fence to be built without enclosing or bordering any other people.

Intuition

To devise a solution to this problem, we must first understand the constraints imposed by the positions of Alice and Bob:

- Since Alice must be on the upper left and Bob on the lower right, for any point where Alice is placed, valid positions for Bob are those with greater x and smaller y coordinates.

Given that constraint, one possible solution progression is:

1. Sort the array of `points` by the x-coordinate and then by the negative y-coordinate in descending order. By doing this, we ensure that for a fixed point (Alice's position), all potential positions for Bob come later in the array, and we only have to look "forward" from each point to find valid Bob positions.
2. Starting from each point (Alice's position), we check every other point that comes after it (possible Bob positions).
3. We initialize a variable `max_y`, which keeps track of the highest y-coordinate of all previously encountered points in the "Bob" loop. If we find a point with a y-coordinate between `max_y` and Alice's y-coordinate, we can place Bob there and increment our answer (since no point within the fence has been encountered yet).
4. Each time we place Bob on a point within these constraints, we update `max_y` to be that y-coordinate, which guarantees that any subsequent points encountered cannot be within the fence for the current iteration of Alice's position.

By repeating this process for each point in the sorted array, we find all valid point pairs for Alice and Bob and count them, resulting in the desired solution.

Solution Approach

The solution approach involves a few key steps capitalized on by the `numberOfPairs` function. The algorithm, the usage of data structures, and the design pattern applied are explained step by step as follows:

1. **Sorting of Points**: The first action taken by the solution is to sort the `points` array. Sorting is done primarily by x-coordinates and secondarily by y-coordinates in descending order (hence the negative sign in the lambda function). This is achieved through the line `points.sort(key=lambda x: (x[0], -x[1]))`. Sorting allows us to apply a linear search pattern later, which reduces the problem complexity significantly.
2. **Two-Pointer Technique**: After sorting, the function employs the two-pointer technique through two nested loops. The outer loop variable `i` goes through each point, which we imagine as the potential position of Alice. The inner loop is used to check potential positions for Bob relative to Alice's position.
3. **Max Y-Coordinate and Counting Valid Pairs**: We introduce a variable `max_y`, initially set to negative infinity `-inf`. This variable is pivotal; it marks the y-coordinate of the highest point found so far that can potentially be inside or on the boundary of the fence being considered. We iterate through the points (for potential Bob positions) using the inner loop `for _, y2 in points[i + 1 :]:`. We check if the current point (`y2`) lies between `max_y` and `y1` (Alice's y-coordinate). If `y2` lies in this range, we have found a higher yet valid y-coordinate for Bob, update `max_y`, and increment `ans`, as this represents another valid pair. This continues until all points have been considered for the current Alice.
4. **Return Final Count**: Once all pairs have been evaluated, the final answer `ans`, which accumulates the number of valid pairs, is returned.

By utilizing sorting and the two-pointer technique, the algorithm ensures a time-efficient exploration of all possible Alice and Bob pairings. No additional data structures are required save for the input array itself, and the pattern avoids unnecessary checks for points that cannot be potential Bob positions based on the sorting conditions. This approach elegantly reduces the problem into a simpler form that can be solved in a single pass after the initial sort, leveraging the sorted property to skip invalid scenarios.

Example Walkthrough

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

Suppose we are given the following points: `[(1,3), (2,2), (3,1)]`. From these points, we need to determine the number of ways Alice and Bob can be positioned according to the given constraints.

1. **Sorting of Points**: First, we sort the points by the x-coordinate (increasing) and y-coordinate (decreasing). Our points are already sorted by the x-coordinates, and since the y-coordinates are in decreasing order when we move through the list, no further sorting is needed. After sorting, our array remains `[(1,3), (2,2), (3,1)]`.
2. **Two-Pointer Technique**: We employ the two-pointer technique where the first pointer `i` is fixed at the start. Let's walk through all points one by one considering them as Alice's position, looking for valid Bob positions.
   - When `i = 0`, Alice is at `(1,3)`. We begin the inner loop to find potential positions for Bob:
     - Bob can be placed at `(2,2)`.
     - Bob can also be placed at `(3,1)`.
   - Moving to `i = 1`, Alice at `(2,2)` and checking for Bob:
     - Bob can be positioned at `(3,1)`.
   - At `i = 2`, Alice is at `(3,1)`, but since there are no more points to the right, Bob cannot be placed anywhere.
3. **Max Y-Coordinate and Counting Valid Pairs**: As we check each potential pairing, we update `max_y` if a valid Bob's position is found. In this case, `max_y` starts off as negative infinity.
   - For `(1,3)` as Alice's position, when Bob is at `(2,2)`, `max_y` becomes `2`. As we check the next point, `(3,1)` also is valid because `1` is still greater than `max_y` (which is `2`). Therefore, `max_y` updates to `1` and we count two valid pairs for Alice's position `(1,3)`.
   - Moving on to Alice's position `(2,2)`, `max_y` is reset to negative infinity. Bob placed at `(3,1)` makes `max_y` updated to `1`. One valid pair is found for Alice's position `(2,2)`.
4. **Return Final Count**: We have found a total of three valid pairs: Alice at `(1,3)` with Bob at `(2,2)`, Alice at `(1,3)` with Bob at `(3,1)`, and Alice at `(2,2)` with Bob at `(3,1)`. We return `3` as the final count of the valid positioning combinations according to the constraints provided.

Using this example, we have demonstrated the solution approach by walking through the steps of sorting, iterating with two pointers, updating `max_y` for valid positionings, and then counting all valid ways Alice and Bob can be placed on the given 2D plane.

Solution Implementation

Python

```python
from math import inf

class Solution:
    def number_of_pairs(self, points):
        # Sort the points by x-coordinate in ascending order and by y-coordinate in descending order in case of tie
        points.sort(key=lambda point: (point[0], -point[1]))

        answer = 0  # Initialize the number of pairs to zero

        # Iterate through each point
        for i, (_, y1) in enumerate(points):
            max_y = -inf  # Initialize the maximum y as negative infinity

            # Compare with other points that come after the current one
            for _, y2 in points[i + 1:]:
                # If there is a point with a y-coordinate less than or equal to y1 but greater than max_y
                if max_y < y2 <= y1:
                    max_y = y2  # Update the max_y
                    answer += 1  # Increment the number of pairs

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

Java

```java
import java.util.Arrays;

class Solution {
    public int numberOfPairs(int[][] points) {
        // Sort points by X coordinate in ascending order, if X coordinates are equal, then by Y in descending order
        Arrays.sort(points, (point1, point2) -> point1[0] == point2[0] ? point2[1] - point1[1] : point1[0] - point2[0]);

        int pairCount = 0; // Initialize the count for the number of valid pairs
        int n = points.length; // Number of points available
        final int INF = 1 << 30; // A very large number to represent the lower bound of maximum Y

        // Iterate through each point to calculate valid pairs
        for (int i = 0; i < n; ++i) {
            int y1 = points[i][1]; // Y coordinate of the current point
            int maxY = -INF; // Initialize maxY to track the max Y value of the valid pair seen so far

            // Iterate through the points that lie after the current point
            for (int j = i + 1; j < n; ++j) {
                int y2 = points[j][1]; // Y coordinate of the subsequent point

                // Check if the current Y2 is a potential candidate for forming a valid pair with Y1
                if (maxY < y2 && y2 <= y1) {
                    maxY = y2; // Update the maximum Y found so far
                    ++pairCount; // Increment the pair count
                }
            }
        }
        return pairCount; // Return the total number of valid pairs found
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to count the number of "good pairs" in a set of points.
    // A "good pair" is defined by the problem statement criteria.
    int numberOfPairs(vector<vector<int>>& points) {
        // First, sort the points based on the x-coordinate, and then y-coordinate in descending order if x-coordinates are equal
        sort(points.begin(), points.end(), [](const vector<int>& a, const vector<int>& b) {
            return a[0] < b[0] || (a[0] == b[0] && b[1] > a[1]);
        });

        int totalPoints = points.size(); // Store the number of points for ease of reference.
        int pairCount = 0; // Initialize a counter for the number of "good pairs".

        // Iterate through each point to consider it as the first member of the pair.
        for (int i = 0; i < totalPoints; ++i) {
            int currentY = points[i][1]; // Get the y-coordinate of the current point.
            int maxY = INT_MIN; // Initialize a variable to keep track of the maximum y-value found so far that's less than curre

            // Iterate through the other points to see if they can form a "good pair" with the current point.
            for (int j = i + 1; j < totalPoints; ++j) {
                int comparisonY = points[j][1]; // Get the y-coordinate of the comparison point.

                // If a larger y-coordinate is found that is still not greater than the current point's y-coordinate, update maxY
                // This ensures that we always have the largest possible y-coordinate that can form a "good pair" with the curre
                if (maxY < comparisonY && comparisonY <= currentY) {
                    maxY = comparisonY;
                    pairCount++; // Increment the count of "good pairs."
                }
            }
        }

        return pairCount; // Return the final count of "good pairs."
    }
};
```

TypeScript

```typescript
function numberOfPairs(points: number[][]): number {
    // Sort points primarily by x-coordinate in increasing order
    // and secondarily by y-coordinate in decreasing order
    points.sort((a, b) => a[0] === b[0] ? b[1] - a[1] : a[0] - b[0]);

    const numPoints = points.length; // Store the number of points
    let pairCount = 0; // Initialize counter for the number of pairs

    // Iterate through all points as the first point in the pair
    for (let i = 0; i < numPoints; ++i) {
        const y1 = points[i][1]; // Y-coordinate of the current first point
        let maxY = -Infinity; // Track the maximum Y of the second point seen so far

        // Iterate through possible second points to form a pair
        for (let j = i + 1; j < numPoints; ++j) {
            const y2 = points[j][1]; // Y-coordinate of the current candidate for the second point

            // Check if the current second point's Y-coordinate is a new maximum
            // and is less than or equal to the first point's Y-coordinate
            if (maxY < y2 && y2 <= y1) {
                maxY = y2; // Update the maximum Y-coordinate
                pairCount++; // Increment the number of valid pairs
            }
        }
    }

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

```python
from math import inf

class Solution:
    def number_of_pairs(self, points):
        # Sort the points by x-coordinate in ascending order and by y-coordinate in descending order in case of tie
        points.sort(key=lambda point: (point[0], -point[1]))

        answer = 0  # Initialize the number of pairs to zero

        # Iterate through each point
        for i, (_, y1) in enumerate(points):
            max_y = -inf  # Initialize the maximum y as negative infinity

            # Compare with other points that come after the current one
            for _, y2 in points[i + 1:]:
                # If there is a point with a y-coordinate less than or equal to y1 but greater than max_y
                if max_y < y2 <= y1:
                    max_y = y2  # Update the max_y
                    answer += 1  # Increment the number of pairs

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

Time and Space Complexity

The given Python code is a method that is designed to count the number of pairs of points where one point is strictly above another point (higher on the Y-axis) and to the right (further along the X-axis). It starts by sorting the points by their X-coordinate, and in the case of a tie, by the negative Y-coordinate. Then, it checks pairs of points and counts the valid pairs. The time complexity and space complexity are analyzed below:

The time complexity of this algorithm is not O(1). The initial sorting of the `points` list takes $O(n \log n)$ time, where n is the number of points in the list. The first loop runs n times, and for each iteration, it runs a nested loop that can iterate up to n times in the worst case. Therefore, the worst case time complexity is $O(n^2)$ because of the nested loops after sorting.

As for the space complexity, typical sorting algorithms like Timsort used in Python's `sort` function have a space complexity of $O(n)$. However, since we don't use any additional data structures that scale with the input size, outside of the sort, the additional space used by the algorithm (for variables and constants) is indeed $O(1)$. The reference answer's claim that the space complexity is $O(1)$ is correct in the context of additional space used, but not for the total space, which accounts for the input data.