447. Number of Boomerangs

Medium | Array | Hash Table | Math

Problem Description

You are provided with a collection of `n` distinct points in a two-dimensional plane, represented by the coordinates `points[i] = [xi, yi]`. A **boomerang** is a set of three points `(i, j, k)` with the condition that the distance between point `i` and point `j` is the same as the distance between point `i` and point `k`. It's important to note that the order of the points in this tuple matters — which means `(i, j, k)` and `(i, k, j)` are considered different boomerangs if `j` and `k` are different points.

Your task is to find and return the total number of boomerangs among the given points.

Intuition

To solve this problem, we iterate through each point and consider it as the center point `i` of a potential boomerang. Then, for each of these center points, we compute the distance to every other point `j` and `k` in the list. If several points are at the same distance from the center point, these points can form multiple boomerangs, because they can be reordered while keeping the center point fixed.

One intuitive approach is to use a hash map (dictionary in Python) to store the number of points at a certain distance from the center point. For every distance, we can calculate the number of possible permutations for boomerangs with two identical legs (the segments `i-j` and `i-k`). Specifically, for `n` points at the same distance from `i`, there are `n * (n - 1)` possible boomerangs, because you can pick the first point in `n` ways and the second point in `(n - 1)` ways.

To implement this, we can use the `Counter` class from the Python Collections module, which essentially creates a frequency table of the distances. Then, we iterate through the distance frequency table, calculate the number of possible boomerangs for each distance, and add them up to get the result for the current center point. We do this for each point in the list and sum the results to get the final answer.

Solution Approach

The solution uses a combination of a brute-force approach and a hash map to efficiently count the number of boomerangs. Here's a step-by-step explanation of the implementation:

1. Initialize a variable `ans` to store the total number of boomerangs.
2. Loop through each point in the `points` list. This point will act as the point `i` in the potential boomerang `(i, j, k)`.
3. Inside the loop, create a `Counter` object called `counter`. This will be used to track the frequency of distances from the current point `i` to all other points.
4. Now, loop through each point again within the outer loop to check the distance from point `i` to every other point (let's call this point `q`).
5. Calculate the squared distance between points `i` (the current point in the outer loop) and `q` (the current point in the inner loop) by using the formula: `distance = (p[0] - q[0]) * (p[0] - q[0]) + (p[1] - q[1]) * (p[1] - q[1])`. Squared distances are used instead of actual distances to avoid floating-point comparison issues and the need for computation-intensive square root operations.
6. Increment the count for this distance in the `counter`. The key is the distance, and the value is the number of points at that distance from `i`.
7. After filling the `counter` with all distances for the current point `i`, iterate through the counter values, and for each distance value `val`, calculate the number of possible boomerangs using the formula `val * (val - 1)`. This represents selecting any two points (order matters) out of the `val` points that are equidistant from `i`.
8. Add the calculated potential boomerangs to the `ans` variable.
9. Continue the process until all points have been used as point `i`.
10. Finally, return the value stored in `ans`, which is the total number of boomerangs.

The algorithm essentially boils down to:

- For every point, count the number of points at each distinct distance.
- For each distance that has more than one point, calculate the possible arrangements of boomerang pairs and sum them up.

This method is O(n^2) in time complexity because it involves a nested loop over the points, with each loop being O(n). The space complexity is O(n) in the worst case, as the counter may contain up to n-1 distinct distance entries if all points are equidistant from point `i`.

Example Walkthrough

Consider an example where `points = [[0,0], [1,0], [2,0]]`. We will use the abovementioned solution approach to calculate the total number of boomerangs.

1. Initialize `ans = 0` as there are no boomerangs counted yet.
2. Start with the first point `[0,0]` and create an empty `Counter` object: `counter = Counter()`.
3. Loop through all points to compute distances from `[0,0]` to each point:
   - The distance from `[0,0]` to itself is `0`. `counter[0]` becomes 1.
   - The distance from `[0,0]` to `[1,0]` is `1^2 + 0^2 = 1`. `counter[1]` becomes 1.
   - The distance from `[0,0]` to `[2,0]` is `2^2 + 0^2 = 4`. `counter[4]` becomes 1.
4. Since there are no two points with the same distance from `[0,0]`, the total for this center point is `0`. No updates to `ans`.
5. Repeat step 2 with the second point `[1,0]`:
   - The distance from `[1,0]` to `[0,0]` is `1`. `counter[1]` becomes 1.
   - The distance from `[1,0]` to itself is `0`. `counter[0]` becomes 1.
   - The distance from `[1,0]` to `[2,0]` is `1`. `counter[1]` becomes 2 (since we already had one point at distance 1).
6. In this case, `counter` has `{1: 2, 0: 1}`. There are `counter[1] = 2` points at distance `1`, so we use the formula `2 * (2 - 1)` to find the number of boomerangs with `[1,0]` as the center which is `2`.
7. Update `ans` by adding the number of boomerangs calculated: `ans += 2`.
8. Repeat steps 2-7 for the third point `[2,0]`. This will result in the same count as with point `[1,0]` because it is the reflection of the situation with respect to the y-axis.
9. After processing all points, the `ans` is `2 + 2 = 4`, which means there are 4 boomerangs in this set of points.

Thus, the function should return `4` based on the example input.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def numberOfBoomerangs(self, points: List[List[int]]) -> int:
        # Initialize the count of boomerangs to zero
        boomerang_count = 0

        # Iterate over each point which will serve as the 'vertex' of the boomerang
        for vertex_point in points:
            # For this vertex, create a counter to keep track of occurrences of distances
            distance_counter = Counter()

            # Now go over all points to calculate the squared distance from the vertex
            for point in points:
                # Calculate squared distance to avoid floating point operations of sqrt
                squared_distance = (vertex_point[0] - point[0])**2 + \
                                   (vertex_point[1] - point[1])**2

                # Increment the count for this distance
                distance_counter[squared_distance] += 1

            # For each distance, calculate potential boomerangs.
            # A boomerang is a set of 2 points at the same distance from the vertex
            # (n choose 2) pairs for each distance which is simply n*(n-1)
            boomerang_count += sum(val * (val - 1) for val in distance_counter.values())

        # Return the total count of boomerangs found
        return boomerang_count
```

Java

```java
class Solution {

    // Function to find the number of boomerang tuples from the given list of points
    public int numberOfBoomerangs(int[][] points) {
        int countOfBoomerangs = 0; // This will hold the final count of boomerangs

        // Iterate over all points to consider each point as the vertex of the boomerang
        for (int[] currentPoint : points) {
            // Counter to store the number of points having the same distance from 'currentPoint'
            Map<Integer, Integer> distanceCounter = new HashMap<>();

            // Iterate over all points to compute distances from 'currentPoint' to others
            for (int[] otherPoint : points) {
                // Calculate squared Euclidean distance to avoid floating point operations
                int distanceSquared = (currentPoint[0] - otherPoint[0]) * (currentPoint[0] - otherPoint[0])
                                    + (currentPoint[1] - otherPoint[1]) * (currentPoint[1] - otherPoint[1]);

                // Increment count for this distance in the counter map
                distanceCounter.put(distanceSquared, distanceCounter.getOrDefault(distanceSquared, 0) + 1);
            }

            // Consider permutations of points with equal distances to form boomerangs
            for (int val : distanceCounter.values()) {
                // Each pair of points can form two boomerangs (i.e. (i, j) and (i, i)).
                // This can be calculated using permutation formula P(n, 2) = n * (n - 1)
                countOfBoomerangs += val * (val - 1);
            }
        }

        // Return the total count of boomerangs
        return countOfBoomerangs;
    }
}
```

C++

```cpp
#include <vector>
#include <unordered_map>

class Solution {
public:
    // Function to calculate the number of boomerangs (i.e., tuples of points that are equidistant from a central point)
    int numberOfBoomerangs(vector<vector<int>>& points) {
        int totalBoomerangs = 0; // Initialize the count of boomerangs to zero
        for (const auto& origin : points) { // Consider each point as the origin
            // Map to store the frequency of the squared distances from this origin
            unordered_map<int, int> distanceFreqMap;

            for (const auto& target : points) { // For each target point
                // Calculate the squared Euclidean distance to avoid dealing with floating point precision
                int distanceSquared = (origin[0] - target[0]) * (origin[0] - target[0])
                                    + (origin[1] - target[1]) * (origin[1] - target[1]);
                // Increment the frequency of this distance
                ++distanceFreqMap[distanceSquared];
            }

            for (const auto& [ , count] : distanceFreqMap) { // For each unique distance
                // If a distance occurs twice or more, it contributes to boomerangs
                // Here we count permutations of points that are equidistant from the origin,
                // which is count * (count - 1) since we need an ordered pair
                totalBoomerangs += count * (count - 1);
            }
        }
        return totalBoomerangs; // Return the total number of boomerangs found
    }
};
```

TypeScript

```typescript
function numberOfBoomerangs(points: number[][]): number {
    let count = 0; // Holds the total number of boomerangs detected

    // Loop over all points as the first point in the triplet
    for (let basePoint of points) {
        let distanceMap: Map<number, number> = new Map(); // Map to store the frequencies of distances

        // Loop over all other points to calculate distances from the basePoint
        for (let targetPoint of points) {
            // Calculate the Euclidean distance squared between basePoint and targetPoint
            const distanceSquared = (basePoint[0] - targetPoint[0]) ** 2 + (basePoint[1] - targetPoint[1]) ** 2;

            // Increment the frequency count for this distance
            const updatedFrequency = (distanceMap.get(distanceSquared) || 0) + 1;
            distanceMap.set(distanceSquared, updatedFrequency);
        }

        // Calculate the number of boomerangs that can be formed with each distance
        for (let [ , frequency] of distanceMap) {
            // If we have at least 2 points at this distance, they can form (frequency * (frequency - 1)) boomerangs.
            count += frequency * (frequency - 1);
        }
    }

    // Return the total number of boomerangs found
    return count;
}
```

```python
from collections import Counter

class Solution:
    def numberOfBoomerangs(self, points: List[List[int]]) -> int:
        # Initialize the count of boomerangs to zero
        boomerang_count = 0

        # Iterate over each point which will serve as the 'vertex' of the boomerang
        for vertex_point in points:
            # For this vertex, create a counter to keep track of occurrences of distances
            distance_counter = Counter()

            # Now go over all points to calculate the squared distance from the vertex
            for point in points:
                # Calculate squared distance to avoid floating point operations of sqrt
                squared_distance = (vertex_point[0] - point[0])**2 + \
                                   (vertex_point[1] - point[1])**2

                # Increment the count for this distance
                distance_counter[squared_distance] += 1

            # For each distance, calculate potential boomerangs.
            # A boomerang is a set of 2 points at the same distance from the vertex
            # (n choose 2) pairs for each distance which is simply n*(n-1)
            boomerang_count += sum(val * (val - 1) for val in distance_counter.values())

        # Return the total count of boomerangs found
        return boomerang_count
```

Time and Space Complexity

Time Complexity

The given code snippet involves two nested loops. The outer loop iterates over each point `p` in the list of points. For each point `p`, the inner loop compares it to every other point `q` in the list to calculate the distances and store them in a hash table (`counter`).

The outer loop runs `n` times, where `n` is the number of points. For each iteration of the outer loop, the inner loop also runs `n` times to calculate the distances from point `p` to every other point `q`. Therefore, the two nested loops result in an `O(n^2)` time complexity for the distance calculations.

After calculating distances, the code iterates over the values in the hash table, which in the worst case contains `n` entries (this is the case when all distances from point `p` to every other point `q` are unique). For each unique distance, an `O(1)` operation is performed to compute the combination of boomerangs. The sum of combinations for each distance is also `O(n)` since it depends on the number of entries in the counter.

Therefore, the total time complexity of the code is `O(n^2)` for the loops, plus `O(n)` for the sum of combinations. As the `O(n^2)` term dominates, the overall time complexity is `O(n^2)`.

Space Complexity

The space complexity of the code is primarily determined by the storage required for the hash table `counter`. In the worst case, if every distance calculated is unique, the counter will hold `n` entries, where `n` is the number of points. Therefore, the space complexity is `O(n)`.

In addition to the counter, a fixed amount of space is used for variables such as `ans` and `distance`, which does not depend on the number of points and thus contributes an `O(1)` term.

As a result, the total space complexity of the code is `O(n)` due to the hash table.