Documentation in Number of Boomerangs

Table of Contents

1. Problem Statement

We are given n distinct points in a 2D plane, where points[i] = [x_i, y_i]. A **boomerang** is defined as a tuple (i, j, k) such that:

- The distance between (i, j) is equal to the distance between (i, k).
- The **order** of (i, j, k) matters.

We need to count the total number of boomerangs possible.

Example 1

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

Output: 2

Explanation: The valid boomerangs are:

- [[1,0], [0,0], [2,0]]
- [[1,0], [2,0], [0,0]]

Example 2

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

Output: 2

2. Intuition

The key observation is that the distance between points is the only factor that matters. If a point i has m points at the same distance, then we can form m * (m - 1) boomerangs by permuting these points.

3. Key Observations

i. We only need to consider distances and not actual coordinates.

ii. Instead of storing actual distances (which involve square roots), we can store squared distances to avoid floating-point precision issues.

iii. Using a hashmap (dictionary) to store distances allows us to efficiently count possible boomerangs in $O(n^2)$.

4. Approach

We iterate through each point i and compute distances to all other points j.

Steps:

i. Use a hashmap to store the count of each squared distance from point i to all other points.

ii. For each unique distance d in the hashmap, if there are m points at this distance, they can form m * (m - 1) boomerangs.

iii. Sum up the total boomerangs.

5. Edge Cases

i. Single Point ($n = 1$)
   a. There are no other points to form a boomerang, so the output should be 0.

ii. All Points are Far Apart
   a. If no two points have the same distance from any given point, the output should be 0.

iii. Multiple Points at Same Distance
   a. If a point has multiple others at the same distance, permutations should be counted correctly.

6. Complexity Analysis

Time Complexity

- Outer loop runs $O(n)$.
- Inner loop runs $O(n)$, computing distances and storing them in a hashmap.
- Counting boomerangs takes $O(n)$.
- Overall Complexity: $O(n^2)$.

Space Complexity

- We use a hashmap to store distances, which in the worst case holds $O(n)$ entries.
- Overall Complexity: $O(n)$.

7. Alternative Approaches

- Brute Force ($O(n^3)$)
  - Try all (i, j, k) combinations and check distances.
  - Not feasible for $n = 500$.
- Optimized Sorting Approach ($O(n \log n)$)
  - Sorting points based on distance before counting.
  - Sorting would be $O(n \log n)$, but hashmap lookup is $O(1)$, so hashmap remains optimal.

8. Test Cases

Basic Cases

```
assert Solution().numberOfBoomerangs([[0,0],[1,0],[2,0]]) == 2
assert Solution().numberOfBoomerangs([[1,1],[2,2],[3,3]]) == 2
assert Solution().numberOfBoomerangs([[1,1]]) == 0
```

Edge Cases

```
# All points at the same location (should return 0)
assert Solution().numberOfBoomerangs([[0,0],[0,0],[0,0]]) == 0

# Large input
points = [[i, 0] for i in range(500)]
assert Solution().numberOfBoomerangs(points) >= 0  # Just to check it runs efficiently
```

9. Final Thoughts

- The hashmap approach provides an optimal solution in $O(n^2)$, which is necessary for large inputs (n = 500).
- Using squared distances avoids floating-point precision issues.
- This approach efficiently counts permutations using hashmaps and simple distance calculations.