1828. Queries on Number of Points Inside a Circle

Medium Geometry Array Math

Leetcode Link

Problem Description

This problem presents a geometric challenge involving both points on a plane and circles described by a center and a radius. You are provided with two arrays; the first, `points`, contains coordinates of various points on a 2D plane, and the second, `queries`, contains the specifications for several circles. Each entry in `queries` provides the central coordinates of a circle and its radius.

The task is to determine how many points from the `points` array fall within or on the boundary of each circle described in `queries`. To fall within or on the boundary, a point's distance from the center of the circle must be less than or equal to the circle's radius. For each circle in `queries`, the output should be the count of such points, and these counts are to be returned as an array.

The key aspect to consider here is the definition of a point being inside a circle. Geometrically, a point (x, y) is inside or on the boundary of a circle with center (xc, yc) and radius r if the distance from (x, y) to (xc, yc) is less than or equal to r. The distance between the two points is calculated using the Pythagorean theorem, which in this case does not require the square root calculation because we can compare the squares of the distances directly to the square of the radius.

Intuition

To find the intuitive solution to this problem, think about the standard way of measuring distance between two points on a plane - through the Pythagorean theorem. Usually, the formula √((x2 - x1)^2 + (y2 - y1)^2) is used, where (x1, y1) and (x2, y2) are coordinates of two points. In the context of a circle, a point lies inside or on the circle if this distance is less than or equal to the radius of the circle.

However, since comparing distances can be done without extracting the square root, the formula simplifies to (x2 - x1)^2 + (y2 - y1)^2 <= r^2. This squared comparison avoids unnecessary computation of square roots, makes the program run faster, and is helpful for counting points within the radius.

We arrive at our solution by iterating over each circle described in `queries`, and for each circle, we go through every point in the `points` array. For each point, we calculate this squared distance from the point to the circle's center and compare it to the square of the radius. If the squared distance is less than or equal to the squared radius, we increment a count. After checking all points for a particular circle, we append the total count to our answer array, and then proceed to the next circle.

The solution, therefore, follows a straight-forward brute-force approach. Its efficiency could, however, be improved by eliminating the need to calculate the distance for all points with respect to all circles, possibly by pre-sorting or partitioning the points.

Solution Approach

The solution provided in Python makes use of a direct implementation of the brute-force approach discussed in the intuition. It utilizes basic data structures from Python's standard library, namely lists, to store the sequence of points and queries and to accumulate the answer. The algorithm follows two nested loops to compare each point with every circle.

Here's a step by step explanation of the algorithm:

1. Initialize an empty list named `ans` which will store the number of points inside each circle.

```
ans = []
```

2. The outer loop iterates over every circle query in the `queries` list. For each circle, the loop retrieves the center coordinates (`x`, `y`) and the radius `r`.

```
for x, y, r in queries:
```

3. Inside the outer loop, we initialize a counter `cnt` to zero for counting how many points are within the current circle including on its boundary. This counter will be reset for each circle.

```
cnt = 0
```

4. A nested inner loop runs through every point in `points`, where `i` and `j` are the x and y coordinates of the current point being checked.

```
for i, j in points:
```

5. For each point, the solution calculates the squared distance from the point to the circle's center using the difference in x coordinates `dx = i - x`, the difference in y coordinates `dy = j - y`, and then summing their squares `dx * dx + dy * dy`.

```
dx, dy = i - x, j - y
```

6. It then checks if this squared distance is less than or equal to the square of the radius `r * r`. If this condition is true, it means the point is inside or on the boundary of the circle, so it increments the counter `cnt`.

```
cnt += dx * dx + dy * dy <= r * r
```

7. After the inner loop has finished checking all points, the inner loop ends, and the code appends the count for the current circle to the `ans` array.

```
ans.append(cnt)
```

8. Finally, once all queries have been checked, the function returns the list `ans` which contains the count of points inside or on the boundary of each circle.

There are no additional data structures or sophisticated patterns employed in this solution. It relies on the fact that distance checking using squares avoids the need for math library calls, which improves computation time, but still, the solution has a time complexity of O(n * m), where n is the number of points and m is the number of queries, which is not efficient for larger datasets.

Example Walkthrough

Let's consider a simple example to illustrate the solution approach. Suppose we have the following points and queries:

- `points` = [(1, 3), (3, 3), (5, 3)]
- `queries` = [(2, 3), 2], [(4, 3), 1]]

We're supposed to find out how many points fall within or on the boundary of each circle described by the queries.

Query 1

- Center (x, y) = (2, 3), Radius (r) = 2

For each point, calculate the squared distance to the center and compare it to the squared radius (r^2 = 4):

1. Point (1, 3): dx = 1 - 2 = -1, dy = 3 - 3 = 0
   - Squared distance = (-1)^2 + 0^2 = 1
   - 1 <= 4 (True), so this point is inside the circle.
2. Point (3, 3): dx = 3 - 2 = 1, dy = 3 - 3 = 0
   - Squared distance = 1^2 + 0^2 = 1
   - 1 <= 4 (True), so this point is also inside the circle.
3. Point (5, 3): dx = 5 - 2 = 3, dy = 3 - 3 = 0
   - Squared distance = 3^2 + 0^2 = 9
   - 9 <= 4 (False), so this point is outside the circle.

There are 2 points inside or on the boundary of the first circle.

Query 2

- Center (x, y) = (4, 3), Radius (r) = 1

Perform the same steps with r^2 = 1:

1. Point (1, 3): dx = 1 - 4 = -3, dy = 3 - 3 = 0
   - Squared distance = (-3)^2 + 0^2 = 9
   - 9 <= 1 (False), so this point is outside the circle.
2. Point (3, 3): dx = 3 - 4 = -1, dy = 3 - 3 = 0
   - Squared distance = (-1)^2 + 0^2 = 1
   - 1 <= 1 (True), so this point is on the boundary of the circle.
3. Point (5, 3): dx = 5 - 4 = 1, dy = 3 - 3 = 0
   - Squared distance = 1^2 + 0^2 = 1
   - 1 <= 1 (True), so this point is also on the boundary of the circle.

There is 1 point inside or on the boundary of the second circle.

In summary, the counts for each query are [2, 1]. Thus, the final returned list of counts would be `[2, 1]`, indicating that two points are within or on the boundary of the first described circle, and one point is within or on the boundary of the second described circle.

Python Solution

```
from typing import List

class Solution:
    def countPoints(self, points: List[List[int]], queries: List[List[int]]) -> List[int]:
        # Initialize an empty list for storing the answer
        answer = []

        # Iterate over each query which consists of a circle defined by (x, y, r)
        for center_x, center_y, radius in queries:
            # Initialize the count of points inside the current circle
            count = 0

            # Check each point to see if it lies within the circle
            for point_x, point_y in points:
                # Calculate the horizontal (dx) and vertical (dy) distance of the point from the circle's center
                dx, dy = point_x - center_x, point_y - center_y

                # Check if the point is inside the circle using the equation of a circle
                # If dx * dx + dy * dy <= radius * radius
                # If the point is inside the circle, increment the count
                count += dx * dx + dy * dy <= radius * radius

            # After checking all points, append the count of the current circle to the answer list
            answer.append(count)

        # Return the list containing the count of points within each circle
        return answer
```

Java Solution

```
class Solution {

    public int[] countPoints(int[][] points, int[][] queries) {
        // Determine the number of queries to process
        int queryCount = queries.length;

        // Prepare an array to store the results for each query
        int[] answer = new int[queryCount];

        // Loop through each query
        for (int k = 0; k < queryCount; ++k) {
            // Retrieve the center and radius of the current query circle
            int centerX = queries[k][0];
            int centerY = queries[k][1];
            int radius = queries[k][2];

            // Loop through each point to check if it is inside the query circle
            for (int[] point : points) {
                // Extract the coordinates of the point
                int pointX = point[0];
                int pointY = point[1];

                // Calculate the distance from the point to the center of the circle
                int distanceX = pointX - centerX;
                int distanceY = pointY - centerY;

                // Check if the point is within the circle by comparing the squares of the distances
                if (distanceX * distanceX + distanceY * distanceY <= radius * radius) {
                    // Increment the counter for this query if the point is inside the circle
                    ++answer[k];
                }
            }
        }

        // Return the array containing the count of points within each circle
        return answer;
    }
}
```

C++ Solution

```
#include <vector>
using std::vector;

class Solution {
public:
    // Function to count points that are within each circular query region
    vector<int> countPoints(vector<vector<int>>& points, vector<vector<int>>& queries) {
        vector<int> results; // This will hold the final count of points for each query

        // Loop over each query, which defines a circle
        for (auto& query : queries) {
            int centerX = query[0]; // X coordinate of the circle's center
            int centerY = query[1]; // Y coordinate of the circle's center
            int radius = query[2]; // Radius of the circle
            int count = 0; // Counter for points inside the circle

            // Compare each point with the current query circle
            for (auto& point : points) {
                int pointX = point[0]; // X coordinate of the point
                int pointY = point[1]; // Y coordinate of the point

                // Calculate squared distance from the point to the center of the circle
                int dx = pointX - centerX;
                int dy = pointY - centerY;
                // If the distance is less than or equal to the radius squared, increment count
                if (dx * dx + dy * dy <= radius * radius) {
                    count++;
                }
            }

            // Store the count of points in the result vector
            results.push_back(count);
        }

        // Return the vector with counts for each query
        return results;
    }
};
```

Typescript Solution

```
// function to count points within each circular query region.
// points: an array of arrays where each sub-array contains 2 integers representing the x and y coordinates of a point
// queries: an array of arrays where each sub-array represents a circle with a center at (x, y) and radius r
// Returns an array of integers where each integer represents the number of points inside a corresponding query circle.
function countPoints(points: number[][], queries: number[][]): number[] {
    // Map through each query and calculate the number of points within the circle defined by the query
    return queries.map(query => {
        // Destructure the query into center's x, center's y, and radius
        const [centerX, centerY, radius] = query;

        // Initialize counter for the number of points within the current circle
        let count = 0;

        // Iterate through each point
        for (const [pointX, pointY] of points) {
            // Calculate the distance from the point to the center of the circle
            const distance = Math.sqrt((centerX - pointX) ** 2 + (centerY - pointY) ** 2);

            // If the distance is less than or equal to the radius, the point is inside the circle
            if (distance <= radius) {
                count++;
            }
        }

        // Return the count of points within the circle for the current query
        return count;
    });
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n * m)`, where `n` is the number of points and `m` is the number of queries. This is because there are two nested loops: the outer loop iterates over each query (which is `m` in number), and the inner loop iterates over each point (which is `n` in number). In the inner loop, we are doing a constant-time computation to check if a point is within the radius of the query circle.

Space Complexity

The space complexity of the code is `O(m)`, where `m` is the number of queries. This is due to the fact that we are only using an additional list `ans` to store the results for each query. The size of this list grows linearly with the number of queries, therefore the space complexity is directly proportional to the number of queries.