2249. Count Lattice Points Inside a Circle

Medium Geometry Array Hash Table Math Enumeration

Leetcode Link

Problem Description

The problem presents a scenario where multiple circles are drawn on a grid. Each circle is defined by its center point coordinates (x_i, y_i) and its radius (r_i). The task is to determine how many lattice points exist that fall inside at least one of these circles. A lattice point is a point on the grid with integer coordinates, and points lying exactly on the circumference of a circle are counted as inside that circle.

Intuition

To solve this problem, one straightforward approach is to check every possible point in the grid and determine whether it falls inside any of the circles. We do this by:

1. Identifying the largest possible x and y coordinates that we might need to check. This is determined by finding the maximum x plus radius and the maximum y plus radius from all the circles since a circle could potentially cover points to the right and above its center up to its radius.
2. We iterate through each point in this bounded grid space.
3. For each point, we calculate its distance from the center of every given circle using the distance formula ((dx * dx + dy * dy)). Here (dx) and (dy) are the differences in the x and y coordinates of the point and the circle's center, respectively. We compare this distance squared to the radius squared of the circle (since (\sqrt{dx^2 + dy^2} <= r) is equivalent to (dx^2 + dy^2 <= r^2)). This squared comparison avoids the use of a square root which is more computationally expensive.
4. If a point is inside any circle (including on the edge), we count it exactly once and then move on to the next point. If a point is not inside the current circle, we continue to check the next circle. Once a point is found inside a circle, there is no need to check the remaining circles which optimizes the process slightly.
5. Continue this process for all points, and then the `ans` variable will contain the total count of lattice points inside at least one circle.

Solution Approach

The implementation uses a brute force algorithm to solve the problem. Given the simplicity of checking whether a point is within a circle, there's no need for complex data structures or advanced patterns. The key steps are as follows:

1. First, we identify the maximum x-coordinate value (`mx`) and y-coordinate value (`my`) that we might need to consider, which will be the bounds for our search space. This is accomplished using list comprehensions that iterate over all the circles and calculate the maximum x-coordinate plus the radius, and the maximum y-coordinate plus the radius, respectively.
2. Then, we use two nested loops to iterate over every possible lattice point within these bounds. The outer loop runs across the x-coordinate range (from 0 to `mx`), and the inner loop runs across the y-coordinate range (from 0 to `my`).
3. For each point `(i, j)` in our nested loops, the algorithm then iterates over every circle to check if the point lies within it. This is where the distance formula comes in. It computes the square of the distance from the point to the circle's center, given by the expression `dx * dx + dy * dy` where `dx = i - x` and `dy = j - y`, and compares this with the square of the radius of the circle `r * r`.
4. If at any point `dx * dx + dy * dy <= r * r` evaluates to true, we increment the count (stored in variable `ans`) by one and break out of the loop that iterates over the circles using `break`. This ensures we avoid double counting points that lie within multiple circles.
5. Finally, after all points have been checked, the algorithm returns `ans`, which contains the total number of lattice points inside at least one circle.

This approach does not use any specific data structure optimizations; it's a direct application of mathematical principles combined with iterative brute force checking. The efficiency could potentially be improved by reducing the search space or by using a geometric data structure to minimize the number of distance checks required, but the given solution is straightforward and relies on the computation being inexpensive enough to check all points within reasonable bounds.

Example Walkthrough

Let's consider a small example with two circles to demonstrate how the solution works:

1. Suppose we have two circles, (C_1) with center at ((x_1, y_1) = (2, 3)) and radius (r_1 = 2), and (C_2) with center at ((x_2, y_2) = (5, 5)) and radius (r_2 = 1).
2. Based on the circles' centers and radii, the maximum x-coordinates we'll need to check is (x_1 + r_1 = 2 + 2 = 4) and (x_2 + r_2 = 5 + 1 = 6). Therefore, the maximum x-coordinate (`mx`) to consider is 6. Similarly, for the y-coordinates, we have (y_1 + r_1 = 3 + 2 = 5) and (y_2 + r_2 = 5 + 1 = 6). Hence, the maximum y-coordinate (`my`) to check is 6.
3. Now, we'll iterate through all lattice points within the bounded space. We start with point (0,0) and end at point (6,6). For illustration, let's check some points to see if they fall inside any circle:
   - Take the point (2,2). For (C_1), we calculate (dx = 2 - 2 = 0) and (dy = 2 - 3 = -1). The distance squared is (dx^2 + dy^2 = 0^2 + (-1)^2 = 1), which is less than (r_1^2 = 4), so the point is inside (C_1). We count it and move to the next point.
   - Consider the point (5,4). For (C_1), (dx = 5 - 2 = 3) and (dy = 4 - 3 = 1) giving us (dx^2 + dy^2 = 3^2 + 1^2 = 10), which is greater than (r_1^2 = 4), so it's outside (C_1). For (C_2), (dx = 5 - 5 = 0) and (dy = 4 - 5 = -1), yielding (dx^2 + dy^2 = 0^2 + (-1)^2 = 1), which is less than (r_2^2 = 1), making the point inside (C_2). We count it.
4. We continue this process, incrementing our count each time we find a point within any circle.
5. After completing the iteration over all points, let's say our counter `ans` stands at 20. This means there are 20 lattice points that lie within at least one of the circles provided.

This example takes only a small portion of the grid and a couple of circles to illustrate the solution steps in an easily understandable manner. In practice, the iteration would cover a larger set of points and potentially more circles, but the principle remains exactly the same.

Python Solution

```python
from typing import List

class Solution:
    def countLatticePoints(self, circles: List[List[int]]) -> int:
        # Initialize a counter for lattice points
        count = 0

        # Calculate the maximum x and y coordinates to consider, so the iteration does not go beyond necessary
        max_x = max(x + r for x, _, r in circles)
        max_y = max(y + r for _, y, r in circles)

        # Iterate over all the points within the area defined by max_x and max_y
        for i in range(max_x + 1):
            for j in range(max_y + 1):
                # Check if the current point (i, j) lies within any of the given circles
                for x, y, radius in circles:
                    # Calculate the squared distance from the center of the circle (x, y) to the point (i, j)
                    dx, dy = i - x, j - y
                    if dx * dx + dy * dy <= radius * radius:
                        # If the point is within a circle, increment the counter and move to the next point
                        count += 1
                        break  # Important to avoid counting a point multiple times
        # Return the total count of lattice points within all circles
        return count

```

Java Solution

```java
class Solution {

    // This method counts all the lattice points that are inside or on the perimeter of the given circles.
    // Each circle is represented by an array with three integers: [x_center, y_center, radius].
    public int countLatticePoints(int[][] circles) {
        // 'maxX' and 'maxY' are used to determine the size of the grid to check for lattice points.
        int maxX = 0, maxY = 0;

        // Calculate the furthest x and y coordinates that we need to check, by looking at the circle
        // with the largest x + radius and y + radius.
        for (var circle : circles) {
            maxX = Math.max(maxX, circle[0] + circle[2]);
            maxY = Math.max(maxY, circle[1] + circle[2]);
        }

        // 'count' will store the total number of lattice points.
        int count = 0;

        // Check each point in the grid.
        for (int x = 0; x <= maxX; ++x) {
            for (int y = 0; y <= maxY; ++y) {
                // Check if this point is within or on any of the circles.
                for (var circle : circles) {
                    int deltaX = x - circle[0];
                    int deltaY = y - circle[1];
                    // Check if the point (x, y) is inside the circle by comparing the square of the distance
                    // from (x, y) to the circle's center with the square of the circle's radius.
                    if (deltaX * deltaX + deltaY * deltaY <= circle[2] * circle[2]) {
                        // If the point is in the circle, increment the count and move to the next point.
                        ++count;
                        break; // Move to the next point since we only need to count each point once.
                    }
                }
            }
        }

        // Return the total count of lattice points.
        return count;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    int countLatticePoints(vector<vector<int>>& circles) {
        int maxX = 0, maxY = 0;

        // Find the maximum x and y values to limit the search space
        for (auto& circle : circles) {
            maxX = max(maxX, circle[0] + circle[2]);
            maxY = max(maxY, circle[1] + circle[2]);
        }

        int answer = 0; // Initialize count of lattice points

        // Iterate over the bounded search area
        for (int x = 0; x <= maxX; ++x) {
            for (int y = 0; y <= maxY; ++y) {
                // Check every circle to see if the point is inside it
                for (auto& circle : circles) {
                    int deltaX = x - circle[0]; // Difference in x
                    int deltaY = y - circle[1]; // Difference in y

                    // Check if the point (x, y) is within the current circle
                    if (deltaX * deltaX + deltaY * deltaY <= circle[2] * circle[2]) {
                        ++answer; // Increment the count if inside any circle
                        break; // Only count once, even if within multiple circles
                    }
                }
            }
        }

        return answer; // Return the total count of lattice points
    }
};

```

Typescript Solution

```typescript
function countLatticePoints(circles: number[][]): number {
    // Initialize maximum x and y values to track the furthest points
    let maxX = 0;
    let maxY = 0;

    // Determine the maximum x and y values considering all circles
    for (const [cx, cy, radius] of circles) {
        maxX = Math.max(maxX, cx + radius);
        maxY = Math.max(maxY, cy + radius);
    }

    // Initialize the answer variable to count lattice points
    let latticePointCount = 0;

    // Iterate over all possible lattice points within the calculated bounds
    for (let x = 0; x <= maxX; ++x) {
        for (let y = 0; y <= maxY; ++y) {
            // Check if current point (x, y) lies within any of the circles
            for (const [cx, cy, radius] of circles) {
                const deltaX = x - cx;
                const deltaY = y - cy;
                // If point is within the circle, increment the count and break out of the loop
                if (deltaX * deltaX + deltaY * deltaY <= radius * radius) {
                    latticePointCount++;
                    break;
                }
            }
        }
    }

    // Return the total count of lattice points within all circles
    return latticePointCount;
}

```

Time and Space Complexity

The given code snippet is designed to count all the lattice points (points with integer coordinates) that lie within or on the boundary of given circles. Each circle is represented by its center coordinates and its radius. The code does the following:

1. Calculates the maximum x and y coordinates (`mx` and `my`) that need to be checked based on the sum of the x or y coordinate of a circle's center and its radius. This determines the bounding box within which lattice points could potentially lie inside a circle.
2. It then iterates over all integer points within this bounding box using two nested loops. For each point `(i, j)`, it checks if the point lies inside any of the circles by calculating the distance from the point to the center of each circle and comparing it with the radius of the circle.
3. If the point is inside a circle, the count (`ans`) is incremented by 1, and the innermost loop is broken to avoid counting a point more than once if it lies within multiple circles.

Time Complexity:

The time complexity is determined by the number of iterations in the nested loops.

- The outer loop runs `mx + 1` times and the second loop runs `my + 1` times, giving `(mx + 1) * (my + 1)` for the two combined.
- The innermost loop runs for each circle, so if there are `n` circles, it runs `n` times for each point.

Therefore, the time complexity is $O(n * (mx + 1) * (my + 1))$, where `n` is the number of circles, `mx` is the maximum x-coordinate to be considered, and `my` is the maximum y-coordinate to be considered.

Space Complexity:

The space complexity of the code is $O(1)$. The only extra space used is a handful of variables for keeping track of counts and iterating through loops, such as `ans`, `mx`, `my`, `dx`, `dy`. These require a fixed amount of space that doesn't change with the input size (number of circles, size of coordinates, or radii of the circles).