149. Max Points on a Line

Hard Geometry Array Hash Table Math

Problem Description

The problem presents a set of points on a 2D coordinate system and asks us to find the maximum number of points that align straightly. In simple terms, we are to determine the largest group of points that lie on the same line. This is a geometric problem commonly encountered in computer graphics and spatial analysis and has significant implications in understanding patterns and relationships between data points in a 2D space.

Intuition

The intuition behind the solution begins with the realization that to find points on the same line, we need to consider the slope formed by pairs of points. In mathematics, the slope of a line is a number that describes both the direction and the steepness of the line.

The key idea for this solution is to iterate through each point and calculate slopes it forms with all other points. If two different lines share the same slope and have a common point, it means they are the same line. However, calculating slopes as floating-point numbers can cause precision issues. To circumvent this, we represent slopes as fractional tuples `(dy, dx)` where `dy` is the change in y (rise) and `dx` is the change in x (run).

Before creating this tuple, we need to reduce these values to their simplest form so that the slopes are uniformly represented. This is where the greatest common divisor (GCD) comes in: if we divide both `dy` and `dx` by their GCD, we get the smallest identical tuple for all points on the same line.

As we calculate these slope tuples, we store and count them using a hash structure, a `Counter` in Python, which keeps the count of each distinct slope seen from a specific starting point. The maximum count of these slopes, plus 1 (for the starting point itself), will indicate the maximum number of points on the same line from that point. We track the overall maximum as we loop through each point to solve the problem.

Solution Approach

The solution uses a brute-force approach along with some mathematical optimizations to reduce computational complexity. Below is the step-by-step breakdown of the implementation:

1. The `gcd` function is defined to find the greatest common divisor of two numbers. It's a classic algorithm known as Euclid's algorithm, which is a recursive approach to successively finding remainders until it lands a zero, thereby finding the GCD of the initial pair.
2. We start by initializing the `ans` to 1 which represents the default maximum number of points on a line that can always include the point itself.
3. We iterate over each point `points[i]` in the outer loop. This point will act as a reference or starting point to calculate slopes with every other point.
4. For a given reference point, we create a `Counter` object to keep track of all the slopes we calculate from the reference point to all other points.
5. The inner loop starts from `points[i + 1]` to the end of the array, ensuring we don't repeat calculations for pairs of points that we have already checked.
6. For every point `points[j]` in this inner loop, we calculate the differences `dx` and `dy`. These represent the horizontal and vertical distances between the points.
7. We then call our `gcd` function with `dx` and `dy` to get the greatest common divisor of the two differences, `g`.
8. With `g`, we normalize `dx` and `dy` by dividing both by `g`. This step ensures we are working with the simplest form of the ratio that defines the slope. We store this normalized pair as a tuple `k`.
9. The `Counter` object, `cnt`, updates or increments the count for the tuple `k`, effectively counting how many points have formed the exact same slope with the starting point.
10. As we increment the count, we also keep track of the maximum value so far with `ans = max(ans, cnt[k] + 1)`. We add 1 because the count in `cnt` does not include the starting point.
11. After considering all pairs starting from each point `points[i]`, the final value of `ans` will be the maximum number of points that lie on the same line.

This approach is exhaustive in that it compares all pairs, but it's efficient in ensuring the numerical stability and uniqueness of the slopes by using the GCD and normalizing the differences. Although the time complexity can be high for very large input arrays, this method is quite practical and straightforward when dealing with typical problem constraints seen in coding interviews.

Example Walkthrough

To illustrate the solution approach, let's consider a small example. Suppose we have the following set of points on a 2D coordinate system: `[(1,1), (2,2), (3,3), (2,3), (3,2)]`.

1. We begin by initializing `ans` to 1.
2. Starting with the first point, `(1,1)`, we create a `Counter` object called `cnt` to store the slopes of lines formed with this point and others.
3. We skip comparing `(1,1)` with itself and move on to calculate the slope with the second point `(2,2)`. The change is `dx = 2−1 = 1` and `dy = 2−1 = 1`. We find the GCD of `dx` and `dy`, which is 1, and normalize to get the slope tuple `(dy/dx) = (1/1)`.
4. We update the counter by setting `cnt[(1,1)]` to 1, indicating that there's one line with this slope from the start point.
5. Next, we calculate the slope between `(1,1)` and `(3,3)`. The change is `dx = 3−1 = 2` and `dy = 3−1 = 2`. The GCD is 2, and the normalized slope tuple is `(1,1)`. We increment `cnt[(1,1)]` to 2.
6. Now, we check the slope between `(1,1)` and `(2,3)`. Here, `dx = 2−1 = 1`, `dy = 3−1 = 2`. The GCD is 1, so the tuple becomes `(2,1)`. We store this new slope in the counter with `cnt[(2,1)] = 1`.
7. Finally for point `(1,1)`, we calculate the slope between `(1,1)` and `(3,2)`. The changes are `dx = 3−1 = 2` and `dy = 2−1 = 1`, with the GCD being 1, leading to a slope tuple `(1,2)`. We put `cnt[(1,2)] = 1`.
8. We now compare the counter values to `ans`. For slope `(1,1)`, we have 2 count, which makes `ans = max(1, 2+1)` (we add 1 to include the reference point).
9. Repeat the process from points `(2,2)`, `(3,3)`, `(2,3)`, and `(3,2)`. Each time we update `cnt` and `ans` appropriately.

Upon completing the loops, we find that the slope `(1,1)` has the highest count, thus the maximum number of points aligning straightly is `ans`, which now equates to 3, taken from the three collinear points `(1,1)`, `(2,2)`, and `(3,3)`.

This step-by-step iteration through each point and counting slope occurrences provides us with the required result. This approach will be applied to all points and all possible slopes to ensure we find the maximum number of collinear points.

Python Solution

```python
from collections import Counter

class Solution:
    def maxPoints(self, points):
        """
        Calculates the maximum number of points that lie on a straight line.

        :param points: A list of point coordinates [x, y].
        :type points: List[List[int]]
        :return: The maximum number of points on a straight line.
        :rtype: int
        """

        def calculate_gcd(a, b):
            """
            Helper function to calculate the Greatest Common Divisor (GCD) using recursion.

            :param a: First number
            :param b: Second number
            :return: GCD of a and b
            """
            return a if b == 0 else calculate_gcd(b, a % b)

        num_points = len(points)
        max_points_on_line = 1  # A single point is always on a line.

        for i in range(num_points):
            x1, y1 = points[i]
            slopes = Counter()  # Counter to track the number of points for each slope
            for j in range(i + 1, num_points):
                x2, y2 = points[j]
                delta_x = x2 - x1
                delta_y = y2 - y1
                gcd_value = calculate_gcd(delta_x, delta_y)

                # Reduce the slope to its simplest form to count all equivalent slopes together.
                slope = (delta_x // gcd_value, delta_y // gcd_value)

                # Increment the count for this slope and update the maximum if needed.
                slopes[slope] += 1
                max_points_on_line = max(max_points_on_line, slopes[slope] + 1)

        return max_points_on_line

# An example of using the class to calculate the maximum number of points on a line.
# Example usage:
# solution = Solution()
# result = solution.maxPoints([[1,1], [2,2], [3,3]])
# print(result)  # Output: 3

```

Java Solution

```java
class Solution {
    public int maxPoints(int[][] points) {
        // Number of points on the plane
        int numPoints = points.length;
        // At least one point will always form a line
        int maxPointsInLine = 1;

        // Iterate over all points as the starting point of a line
        for (int i = 0; i < numPoints; ++i) {
            int x1 = points[i][0], y1 = points[i][1];
            // A map to store the slope of lines and their counts
            Map<String, Integer> lineMap = new HashMap<>();

            // Try forming lines with every other point
            for (int j = i + 1; j < numPoints; ++j) {
                int x2 = points[j][0], y2 = points[j][1];
                // Calculate the deltas for the line
                int deltaX = x2 - x1;
                int deltaY = y2 - y1;
                // Compute the greatest common divisor to normalize the slope
                int gcd = gcd(deltaX, deltaY);
                // Create a unique string key for the slope after normalizing
                String slopeKey = (deltaX / gcd) + "." + (deltaY / gcd);
                // Increment the number of points that form the current line
                lineMap.put(slopeKey, lineMap.getOrDefault(slopeKey, 0) + 1);
                // Update the maximum number of points in a line if necessary
                maxPointsInLine = Math.max(maxPointsInLine, lineMap.get(slopeKey) + 1);
            }
        }
        // Return the maximum number of points found in a line
        return maxPointsInLine;
    }

    // Helper method to calculate the greatest common divisor of two numbers
    private int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }
}

```

C++ Solution

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

class Solution {
public:
    // Helper function to find the Greatest Common Divisor (GCD) of two numbers
    int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }

    // Function to find the maximum number of points that lie on a straight line
    int maxPoints(std::vector<std::vector<int>>& points) {
        int numPoints = points.size(); // Number of points
        int maxPointsOnLine = 1; // Initialize the maximum with 1 (a line requires at least 2 points)

        for (int i = 0; i < numPoints; ++i) {
            int x1 = points[i][0], y1 = points[i][1]; // Coordinates for the first point
            std::unordered_map<std::string, int> slopeCount; // Map to keep track of slopes (as string keys) and their counts

            for (int j = i + 1; j < numPoints; ++j) {
                int x2 = points[j][0], y2 = points[j][1]; // Coordinates for the second point
                int deltaX = x2 - x1, deltaY = y2 - y1; // Differences in x and y coordinates (components of the slope)
                int gcdSlope = gcd(deltaX, deltaY); // Calculate GCD to standardize the slope

                // Create a unique key for the slope by concatenating deltaX and deltaY divided by their GCD
                std::string slopeKey = std::to_string(deltaX / gcdSlope) + "." + std::to_string(deltaY / gcdSlope);

                // Increment the count of points for the current slope
                slopeCount[slopeKey]++;
                // Update maxPointsOnLine if the current slope has more points than the maximum recorded so far
                maxPointsOnLine = std::max(maxPointsOnLine, slopeCount[slopeKey] + 1);
            }
        }
        return maxPointsOnLine; // Return the maximum number of points on a line
    }
};

```

Typescript Solution

```typescript
// Import statements from TypeScript/JavaScript are not required here as we're defining global variables and methods.

// Helper function to find the Greatest Common Divisor (GCD) of two numbers
function gcd(a: number, b: number): number {
    // If b is 0, a is the GCD
    return b === 0 ? a : gcd(b, a % b);
}

// Function to find the maximum number of points that lie on a straight line
function maxPoints(points: number[][]): number {
    const numOfPoints = points.length; // Number of points
    let maxPointsOnLine = 1; // Initialize to 1 since a single point technically forms a line

    // Iterate through each point to use as a starting point
    for (let i = 0; i < numOfPoints; ++i) {
        const x1 = points[i][0], y1 = points[i][1]; // Coordinates for the current point
        let slopeCount: { [key: string]: number } = {}; // Map to keep track of slopes and their counts

        // Iterate through remaining points to form lines and calculate slopes
        for (let j = i + 1; j < numOfPoints; ++j) {
            const x2 = points[j][0], y2 = points[j][1]; // Coordinates for the next point
            let deltaX = x2 - x1, deltaY = y2 - y1; // Differences in x and y coordinates (components of the slope)
            const commonDivisor = gcd(deltaX, deltaY); // Calculate GCD to standardize the slope representation

            // Create a unique key for the slope by concatenating normalized deltaX and deltaY
            const slopeKey = `${deltaX / commonDivisor},${deltaY / commonDivisor}`;

            // Increment the count of points for the current normalized slope
            slopeCount[slopeKey] = (slopeCount[slopeKey] || 0) + 1;
            // Update maxPointsOnLine if the current slope has more points than the maximum recorded so far
            maxPointsOnLine = Math.max(maxPointsOnLine, slopeCount[slopeKey] + 1);
        }
    }

    return maxPointsOnLine; // Return the maximum number of points on a line
}

// Example usage:
// const result = maxPoints([[1,1],[2,2],[3,3]]);
// console.log(result); // Output would be 3

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by two nested loops iterating over the points and the calculation of the greatest common divisor (gcd) for each pair of points. The outer loop runs `n` times (where `n` is the number of points), and the inner loop runs progressively fewer times as `i` increases, leading to a sum of the series from `1` to `n−1`, which is (n-1)n/2. The gcd calculation has a time complexity that is generally O(log(min(a, b))), where `a` and `b` are the differences in the x and y coordinates of two points.

Overall, the time complexity is O(n^2 * log(min(dx, dy))), since the gcd computation is the most significant operation in the inner loop.

```
O(n^2 * log(min(dx, dy)))
```

Space Complexity

The space complexity of the code is largely influenced by the `Counter` dictionary storing each unique slope encountered. In the worst case, every pair of points could have a unique slope, leading to O(n^2) entries in the `Counter`. However, this is very unlikely in a standard dataset, and thus the space complexity would typically be much lower.

The recursive nature of the `gcd` function also adds to the space complexity, due to the call stack. However, because the depth of the recursion is O(log(min(a, b))), this does not significantly affect the overall space complexity.

Thus, the worst-case space complexity is:

```
O(n^2)
```

But on average, it would be significantly better, depending on how many points share the same slope.