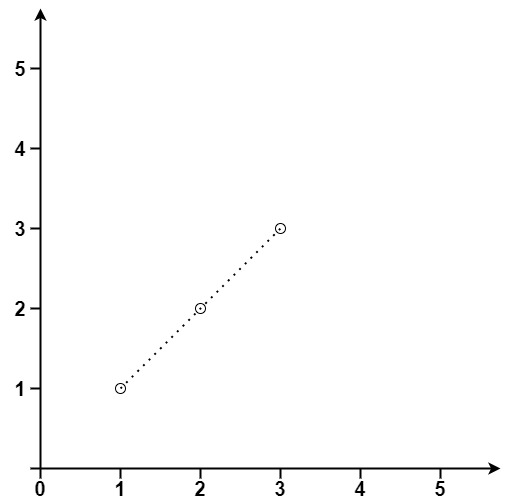
<https://leetcode.com/problems/max-points-on-a-line/description/>

Given an array of points where points[i] = [xi, yi] represents a point on the **X-Y** plane, return *the maximum number of points that lie on the same straight line*.

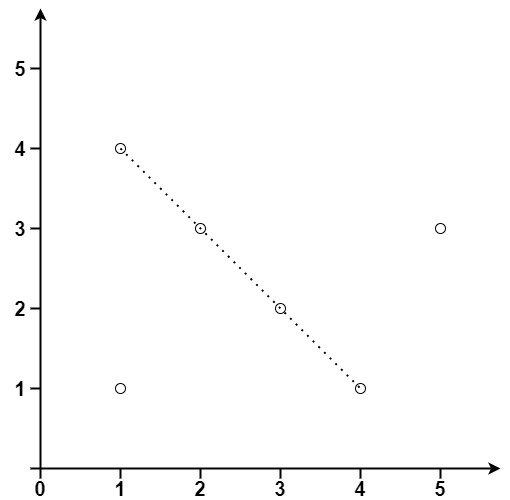
**Example 1:**



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

**Output:** 3

**Example 2:**



**Input:** points = [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]

**Output:** 4

**Constraints:**

1 <= points.length <= 300

points[i].length == 2

-10^4 <= xi, yi <= 10^4

All the points are **unique**.

**Attempt 1: 2025-03-27**

**Solution 1: Math (60 min)**

class Solution {

    public int maxPoints(int[][] points) {

        int n = points.length;

        int max = 0;

        // For each base point, we will calculate the slope relative to every other point.

        for(int i = 0; i < n; i++) {

            Map<String, Integer> slopeCount = new HashMap<>();

            int x = points[i][0];

            int y = points[i][1];

            int curMax = 0;

            // If no change on x and y, then its a duplicate point

            // Count points that are identical to the base point separately.

            int duplicates = 0;

            // For each pair of points, compute the change in x (deltaX)

            // and change in y (deltaY).

            for(int j = 0; j != i && j < n; j++) {

                int dx = points[j][0] - x;

                int dy = points[j][1] - y;

                if(dx == 0 && dy == 0) {

                    duplicates++;

                    continue;

                }

                // If deltaX is zero, the line is vertical. Otherwise, reduce

                // the fraction deltaY/deltaX using the GCD to form a unique

                // key for the slope.

                String key = "";

                if(dx == 0) {

                    key = "INF";

                } else {

                    int gcd = gcdHelper(Math.abs(dx), Math.abs(dy));

                    int reduceDx = dx / gcd;

                    int reduceDy = dy / gcd;

                    // Normalize negative slope

                    // If we don't normalize the negative sign, the key would

                    // be 2\_-1. However, the slope 2 / -1 is the same as -2 / 1

                    // (both represent the same line). To ensure consistency,

                    // we normalize the representation by moving the negative

                    // sign to the numerator (if the denominator is negative)

                    if(reduceDx < 0) {

                        reduceDx = -reduceDx;

                        reduceDy = -reduceDy;

                    }

                    key = reduceDx + "\_" + reduceDy;

                }

                slopeCount.put(key, slopeCount.getOrDefault(key, 0) + 1);

                curMax = Math.max(curMax, slopeCount.get(key));

            }

            // The '+ 1' means base point {x, y} itself

            max = Math.max(curMax + duplicates + 1, max);

        }

        return max;

    }

    // Find greatest common divisor

    private int gcdHelper(int a, int b) {

        while(b != 0) {

            int tmp = b;

            b = a % b;

            a = tmp;

        }

        return a;

    }

}

Time Complexity: O(n^2)

Space Complexity: O(1)

**Refer to**

**Refer to Deepseek**

To solve this problem, we need to determine the maximum number of points that lie on the same straight line given an array of points on a 2D plane.

**Approach**

The key insight is that for each point, we can consider it as the base point and calculate the slope of the line formed with every other point. Points that lie on the same line through the base point will share the same slope. To avoid precision issues with floating-point arithmetic, we represent the slope as a reduced fraction using the greatest common divisor (GCD). Additionally, we handle vertical lines separately by checking if the change in x (deltaX) is zero.

Iterate through each point as the base point: For each base point, we will calculate the slope relative to every other point.

Handle duplicates: Count points that are identical to the base point separately.

Slope calculation: For each pair of points, compute the change in x (deltaX) and change in y (deltaY). If deltaX is zero, the line is vertical. Otherwise, reduce the fraction deltaY/deltaX using the GCD to form a unique key for the slope.

Hash Map: Use a hash map to count occurrences of each slope (or vertical line) relative to the base point.

Max points calculation: The maximum number of points on a line through the base point is the highest count from the hash map plus the count of duplicates and the base point itself.

**Solution Code**

class Solution {

public int maxPoints(int[][] points) {

int n = points.length;

if (n <= 1) {

return n;

}

int max = 0;

for (int i = 0; i < n; i++) {

Map<String, Integer> slopeCount = new HashMap<>();

int duplicates = 0;

int currentMax = 0;

for (int j = 0; j < n; j++) {

if (i == j) {

continue;

}

int dx = points[j][0] - points[i][0];

int dy = points[j][1] - points[i][1];

if (dx == 0 && dy == 0) {

duplicates++;

continue;

}

String key;

if (dx == 0) {

key = "inf";

} else {

int gcd = gcd(Math.abs(dx), Math.abs(dy));

int reducedDx = dx / gcd;

int reducedDy = dy / gcd;

if (reducedDx < 0) {

reducedDx = -reducedDx;

reducedDy = -reducedDy;

}

key = reducedDy + "\_" + reducedDx;

}

slopeCount.put(key, slopeCount.getOrDefault(key, 0) + 1);

currentMax = Math.max(currentMax, slopeCount.get(key));

}

max = Math.max(max, currentMax + duplicates + 1);

}

return max;

}

private int gcd(int a, int b) {

while (b != 0) {

int temp = b;

b = a % b;

a = temp;

}

return a;

}

}

**Explanation**

Initialization: Check if the number of points is less than or equal to 1, in which case return the number of points directly.

Outer Loop: For each point, treat it as the base point.

Inner Loop: For every other point, calculate the slope relative to the base point.

Duplicate Handling: Count how many points are identical to the base point.

Slope Calculation: For non-vertical lines, compute the GCD of deltaX and deltaY to reduce the fraction and form a unique key. For vertical lines, use a special key "inf".

Hash Map Update: Track the count of each slope using a hash map.

Result Calculation: For each base point, determine the maximum points on a line by combining the highest slope count, duplicates, and the base point itself.

This approach efficiently handles all edge cases and ensures precision by using reduced fractions and special handling for vertical lines. The time complexity is O(n^2) due to nested loops, which is feasible for the problem constraints.

**Can you show an example for why we need below handling ?**

if (reducedDx < 0) {

reducedDx = -reducedDx;

reducedDy = -reducedDy;

}

is to normalize the representation of the slope to avoid considering different but equivalent slope representations as distinct keys in the hash map. This ensures that slopes that are mathematically the same are treated as the same key, which is crucial for accurately counting points on the same line.

**Example to Illustrate the Need for Normalization**

Consider two points A = (1, 1) and B = (2, 3). The slope between A and B is calculated as:

deltaX = 2 - 1 = 1

deltaY = 3 - 1 = 2

Slope fraction: deltaY / deltaX = 2 / 1

Now, consider another point C = (3, 5). The slope between A and C is:

deltaX = 3 - 1 = 2

deltaY = 5 - 1 = 4

Slope fraction: deltaY / deltaX = 4 / 2 = 2 / 1 (after reducing by GCD of 4 and 2, which is 2)

Both slopes 2/1 and 4/2 reduce to 2/1, so they should be considered the same. However, if we didn't normalize the representation, we might end up with different keys for the same slope, leading to incorrect counts.

**Another Example with Negative Slopes**

Now, let's consider points A = (1, 1) and D = (0, 3). The slope between A and D is:

deltaX = 0 - 1 = -1

deltaY = 3 - 1 = 2

Slope fraction: deltaY / deltaX = 2 / -1

If we don't normalize the negative sign, the key would be 2\_-1. However, the slope 2 / -1 is the same as -2 / 1 (both represent the same line). To ensure consistency, we normalize the representation by moving the negative sign to the numerator (if the denominator is negative):

Original reduced slope: 2 / -1

After normalization: -2 / 1

Now, the key becomes -2\_1. This ensures that slopes like 2 / -1 and -2 / 1 are treated as the same key.

**Why This Matters**

Without this normalization, the hash map might treat 2/1 and -2/-1 (which reduce to 2/1) as different keys, leading to incorrect counts of points on the same line. By ensuring the denominator is always positive (and adjusting the numerator accordingly), we maintain a consistent key representation for the same slope.

**Final Answer**

The normalization step ensures that equivalent slope representations (like 2/-1 and -2/1) are treated as the same key in the hash map. This is crucial for accurately counting the number of points lying on the same straight line, as it avoids duplicate keys for the same slope.

**Example Walkthrough**

Let's walk through an example with points A = (1, 1), B = (2, 3), C = (3, 5), and D = (0, 3).

Base Point A:

Slope with B: deltaX = 1, deltaY = 2 → key 2\_1

Slope with C: deltaX = 2, deltaY = 4 → reduced to 2\_1 (same as above)

Slope with D: deltaX = -1, deltaY = 2 → initially 2\_-1, normalized to -2\_1

Without normalization, the keys would be 2\_1, 2\_1, and 2\_-1, leading to counts of 2 and 1, respectively. With normalization, the keys are 2\_1 (count=2) and -2\_1 (count=1), but since 2\_1 and -2\_1 represent the same line, we need to ensure they are treated the same. However, in this case, 2\_1 and -2\_1 are actually different slopes (2/1 is positive, -2/1 is negative), so they should be treated separately.

Wait, this seems contradictory to the earlier explanation. Let me re-examine.

Actually, 2 / -1 is -2 / 1, so the normalized key should be -2\_1. But 2 / 1 is a different slope. So in this case, the lines AB and AC have slope 2/1, and AD has slope -2/1. These are different lines, so they should have different keys. The normalization ensures that 2 / -1 and -2 / 1 are the same key (-2\_1), but it doesn't merge 2/1 and -2/1.

So the initial explanation might have been slightly misleading. The normalization ensures that slopes like a/-b and -a/b are the same key, but slopes a/b and -a/b are different (as they should be, since they represent lines with opposite slopes).

Therefore, the normalization is necessary to handle cases where the slope fraction could be represented in multiple ways due to negative signs, but it doesn't merge positive and negative slopes.

For example, 2 / -1 and -2 / 1 are the same slope, so they should have the same key (-2\_1). Without normalization, one might be 2\_-1 and the other -2\_1, leading to separate counts.

So the normalization ensures that all representations of the same slope (differing only by sign) are consolidated into a single key.

In the example above, the slope between A and D is 2 / -1, which is the same as -2 / 1. The normalization converts 2 / -1 to -2 / 1, so the key is -2\_1.

Similarly, if another point E gave a slope of -4 / 2, this would reduce to -2 / 1, same key -2\_1.

This ensures that all representations of the same slope are counted together.

Without this, 2 / -1 and -4 / 2 might generate different keys (2\_-1 and -4\_2), leading to incorrect counts.

So the normalization is crucial for correctly identifying identical slopes despite different fractional representations.

**Conclusion**

The normalization step ensures that different fractional representations of the same slope (due to negative signs) are consolidated into a single key in the hash map. This is essential for accurately counting the number of points lying on the same straight line, as it prevents the same slope from being counted under different keys.

**Refer to**