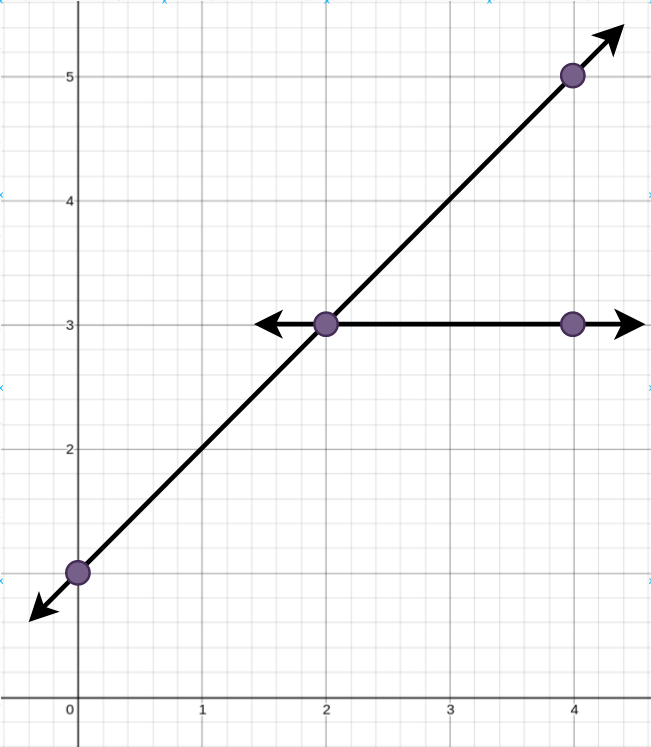
<https://github.com/doocs/leetcode/blob/main/solution/2100-2199/2152.Minimum%20Number%20of%20Lines%20to%20Cover%20Points/README_EN.md>

You are given an array points where points[i] = [xi, yi] represents a point on an X-Y plane.

**Straight lines** are going to be added to the **X-Y** plane, such that every point is covered by at **least**one line.

Return *the****minimum****number of****straight lines****needed to cover all the points*.

**Example 1:**



**Input:** points = [[0,1],[2,3],[4,5],[4,3]]

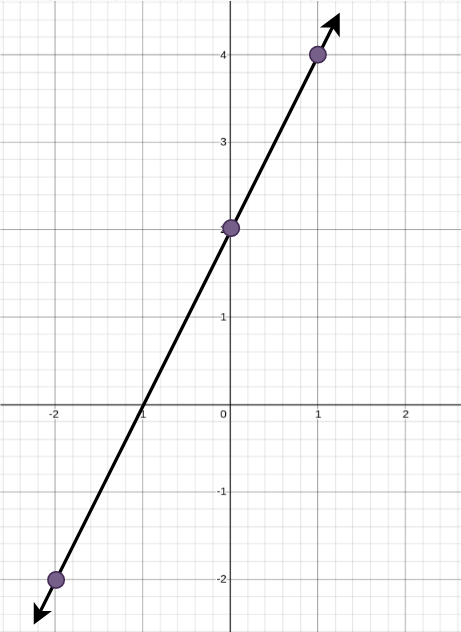
**Output:** 2

**Explanation:** The minimum number of straight lines needed is two. One possible solution is to add:

- One line connecting the point at (0, 1) to the point at (4, 5).

- Another line connecting the point at (2, 3) to the point at (4, 3).

**Example 2:**



**Input:** points = [[0,2],[-2,-2],[1,4]]

**Output:** 1

**Explanation:** The minimum number of straight lines needed is one. The only solution is to add:

- One line connecting the point at (-2, -2) to the point at (1, 4).

**Constraints:**

1 <= points.length <= 10

points[i].length == 2

-100 <= xi, yi <= 100

All the points are **unique**.

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

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

**Refer to Deepseek**

To solve this problem, we need to determine the minimum number of straight lines required to cover all given points on a 2D plane. The key insight is that a line can cover multiple points if they are collinear, so the goal is to find the smallest set of lines such that every point lies on at least one line in the set.

**Approach**

Check for Trivial Cases: If there are 0 or 1 points, no lines are needed. If there are 2 points, exactly one line is needed.

Generate All Possible Lines: For each pair of distinct points, generate the line that passes through them. A line can be uniquely represented by its coefficients (A, B, C) in the general form (Ax + By + C = 0).

Normalize Line Representation: To avoid duplicate representations of the same line, normalize the coefficients (A, B, C) by dividing by their greatest common divisor (GCD) and ensuring the leading coefficient is positive.

Count Covered Points: For each unique line, count how many points lie on it. Track the maximum number of points any single line can cover.

Greedy Selection: Use a greedy approach to select lines that cover the most uncovered points until all points are covered. This involves repeatedly selecting the line that covers the maximum number of remaining points and marking those points as covered.

**Solution Code**

class Solution {

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

int n = points.length;

if (n <= 2) {

return n == 0 ? 0 : 1;

}

List<Line> lines = new ArrayList<>();

// Generate all possible lines from pairs of points

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

for (int j = i + 1; j < n; j++) {

int x1 = points[i][0], y1 = points[i][1];

int x2 = points[j][0], y2 = points[j][1];

// Line equation: (y2 - y1)(x - x1) - (x2 - x1)(y - y1) = 0

// General form: A x + B y + C = 0

int A = y2 - y1;

int B = x1 - x2;

int C = x2 \* y1 - x1 \* y2;

// Normalize the line equation

int gcd = gcd(gcd(Math.abs(A), Math.abs(B)), Math.abs(C));

A /= gcd;

B /= gcd;

C /= gcd;

// Ensure the leading coefficient is positive

if (A < 0 || (A == 0 && B < 0)) {

A = -A;

B = -B;

C = -C;

}

lines.add(new Line(A, B, C));

}

}

// Remove duplicate lines

Set<Line> uniqueLines = new HashSet<>(lines);

lines = new ArrayList<>(uniqueLines);

// For each line, count how many points lie on it

Map<Line, Set<Integer>> linePoints = new HashMap<>();

for (Line line : lines) {

Set<Integer> covered = new HashSet<>();

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

int x = points[k][0], y = points[k][1];

if (line.A \* x + line.B \* y + line.C == 0) {

covered.add(k);

}

}

linePoints.put(line, covered);

}

// Greedily select lines that cover the most points

Set<Integer> remainingPoints = new HashSet<>();

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

remainingPoints.add(i);

}

int lineCount = 0;

while (!remainingPoints.isEmpty()) {

Line bestLine = null;

int maxCovered = 0;

for (Line line : linePoints.keySet()) {

Set<Integer> covered = linePoints.get(line);

// Retains only the elements in this set that are contained in the specified collection

covered.retainAll(remainingPoints);

if (covered.size() > maxCovered) {

maxCovered = covered.size();

bestLine = line;

}

}

if (bestLine == null) {

break; // No line covers any remaining points (shouldn't happen)

}

remainingPoints.removeAll(linePoints.get(bestLine));

lineCount++;

}

return lineCount;

}

private int gcd(int a, int b) {

while (b != 0) {

int temp = b;

b = a % b;

a = temp;

}

return a;

}

class Line {

int A, B, C;

Line(int A, int B, int C) {

this.A = A;

this.B = B;

this.C = C;

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Line line = (Line) obj;

return A == line.A && B == line.B && C == line.C;

}

@Override

public int hashCode() {

return Objects.hash(A, B, C);

}

}

}

**Explanation**

Trivial Cases Handling: Directly return 0 or 1 if there are 0 or 1-2 points respectively, as they can be covered by zero or one line.

Line Generation: For each pair of points, compute the line equation in the general form (Ax + By + C = 0). Normalize the coefficients by their GCD to ensure consistent representation.

Unique Lines: Use a set to eliminate duplicate lines that might have been generated from different point pairs but represent the same line.

Point Coverage: For each unique line, determine which points lie on it by checking if they satisfy the line equation.

Greedy Selection: Repeatedly select the line that covers the most remaining points, remove those points from consideration, and increment the line count until all points are covered.

This approach efficiently minimizes the number of lines by leveraging collinearity and greedy selection, ensuring optimal coverage with the least number of lines.

**Need explain on below ?**

**Line equation: (y2 - y1)(x - x1) - (x2 - x1)(y - y1) = 0**

**General form: A x + B y + C = 0**

The line equation ((y2 - y1)(x - x1) - (x2 - x1)(y - y1) = 0) is derived from the two-point form of a line equation. Let's break it down step by step to understand how it transforms into the general form (A x + B y + C = 0).

**Two-Point Form of a Line Equation**

Given two points ((x1, y1)) and ((x2, y2)), the slope (m) of the line passing through them is:

m = (y2 - y1) / (x2 - x1)

Using the point-slope form, the equation of the line can be written as:

y - y1 = m (x - x1)

Substituting the slope (m):

y - y1 = (y2 - y1) / (x2 - x1) \* (x - x1)

**Rearranging to Standard Form**

To eliminate the fraction, multiply both sides by (x2 - x1):

(x2 - x1)(y - y1) = (y2 - y1)(x - x1)

Now, bring all terms to one side to set the equation to zero:

(y2 - y1)(x - x1) - (x2 - x1)(y - y1) = 0

**Expanding the Equation**

Let's expand the terms:

(y2 - y1) \* x - (y2 - y1) \* x1 - (x2 - x1) \* y + (x2 - x1) \* y1 = 0

Combine like terms:

(y2 - y1) \* x - (x2 - x1) \* y + [ - (y2 - y1) \* x1 + (x2 - x1) \* y1 = 0

**Identifying Coefficients**

Now, let's identify the coefficients A, B, and C in the general form A x + B y + C = 0:

A = (y2 - y1)

B = -(x2 - x1) = (x1 - x2)

C = - (y2 - y1) x1 + (x2 - x1) y1 = x2 y1 - x1 y2 (after simplifying)

So, the general form becomes:

(y2 - y1) x + (x1 - x2) y + (x2 y1 - x1 y2) = 0

**Example**

Let's take two points (1, 2) and (3, 4):

A = 4 - 2 = 2

B = 1 - 3 = -2

C = 3 \* 2 - 1 \* 4 = 6 - 4 = 2

Thus, the line equation is:

2x - 2y + 2 = 0

which can be simplified to:

x - y + 1 = 0

**Verification**

Let's verify if both points satisfy this equation:

For ((1, 2)):

[ 1 - 2 + 1 = 0 ] ✓

For ((3, 4)):

[ 3 - 4 + 1 = 0 ] ✓

**Why Use This Form?**

The general form (A x + B y + C = 0) is advantageous because:

Unified Representation: It can represent any line, including vertical lines (where (B = 0)) and horizontal lines (where (A = 0)).

Easy to Compute: Coefficients (A), (B), and (C) can be directly derived from two points, making it straightforward to implement in code.

Normalization: By dividing by the GCD of (A), (B), and (C), we can ensure a unique representation for each line, avoiding duplicates.

**Conclusion**

The line equation ((y2 - y1)(x - x1) - (x2 - x1)(y - y1) = 0) is a convenient way to derive the general form (A x + B y + C = 0) from two points. This form is essential for algorithms that need to handle lines in a standardized manner, such as checking collinearity or finding the minimum number of lines to cover a set of points.

**Refer to**

[L149.Max Points on a Line](note://WEB9621fb1c6a6da80594279dfd17bf65a3)