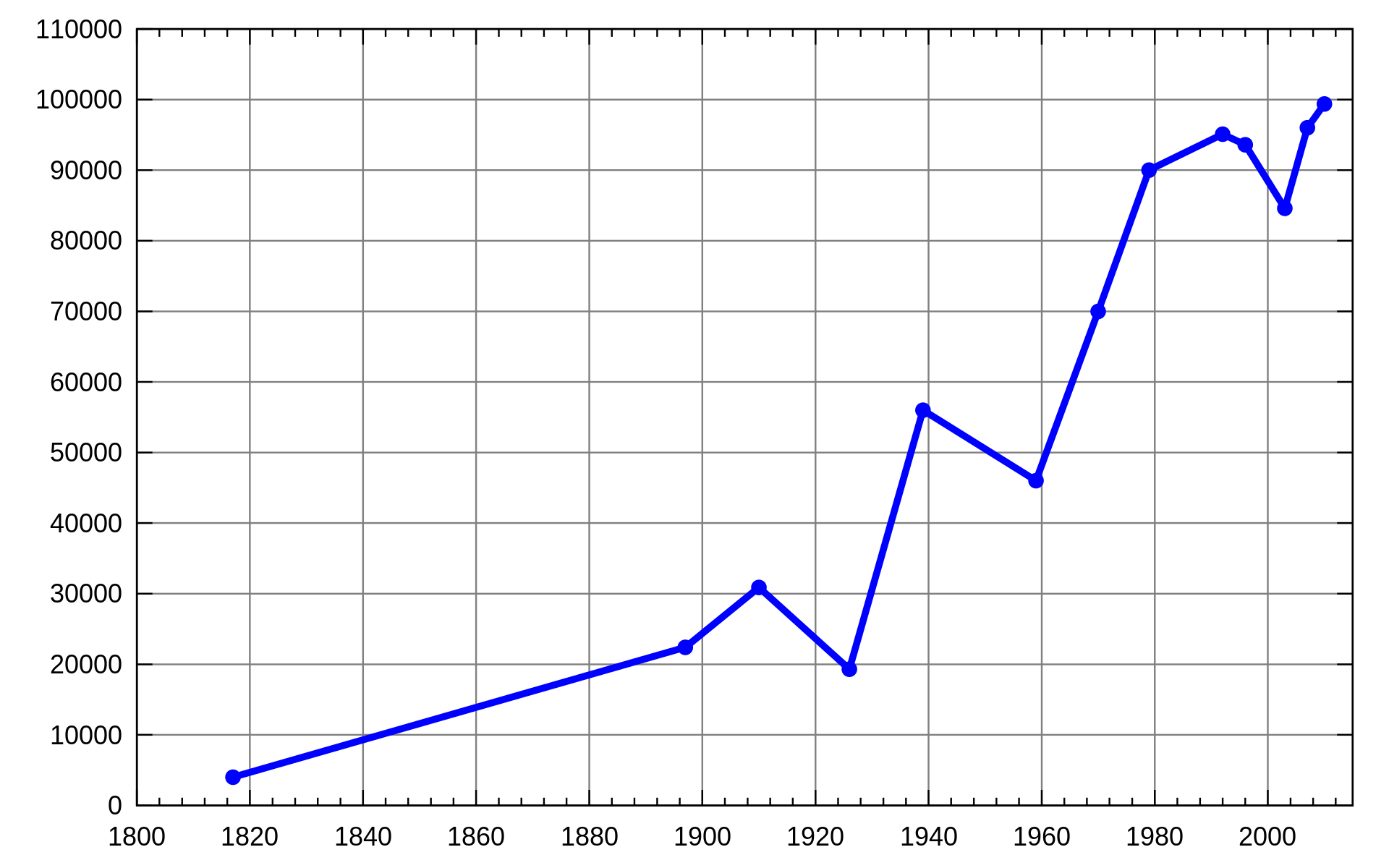
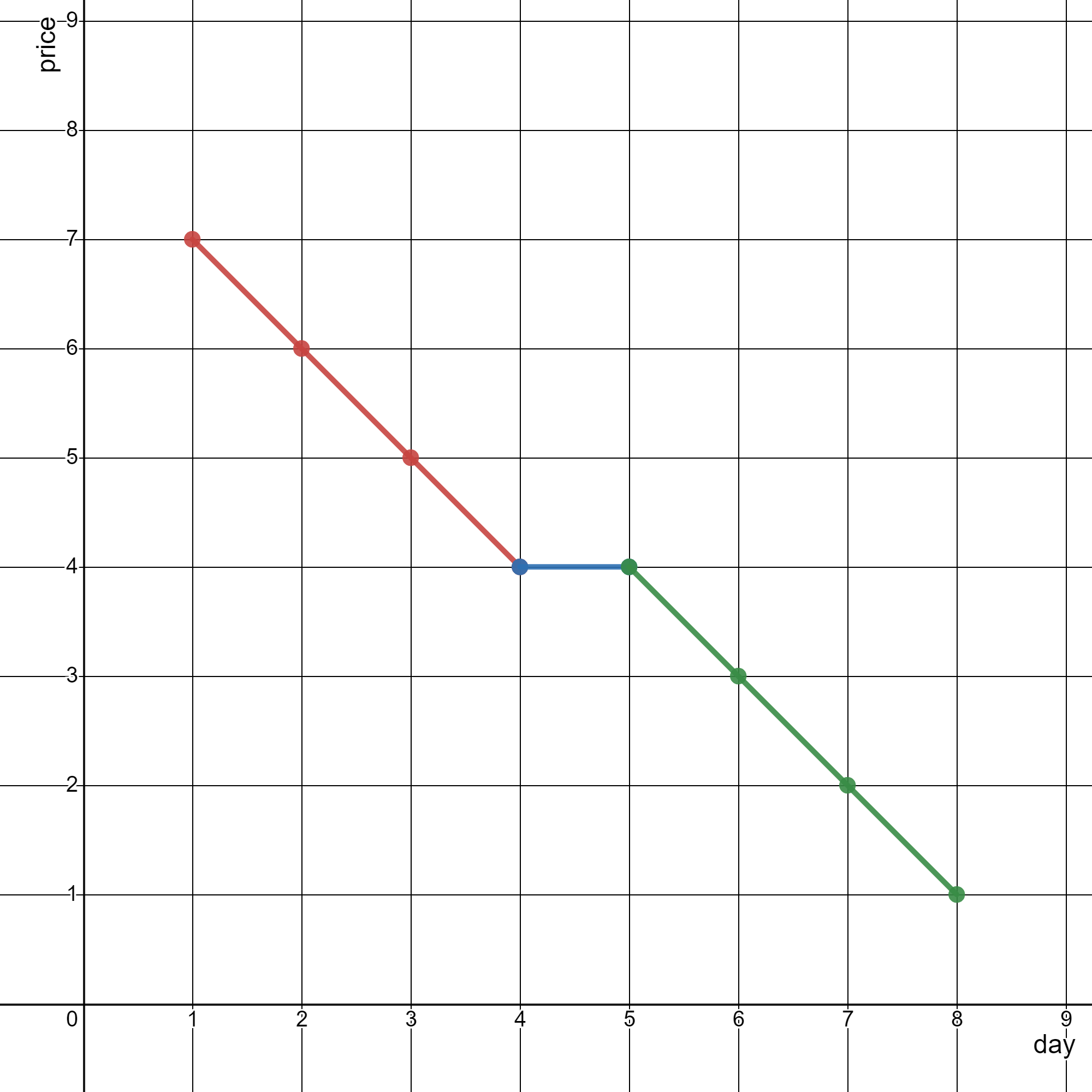
<https://leetcode.com/problems/minimum-lines-to-represent-a-line-chart/description/>

You are given a 2D integer array stockPrices where stockPrices[i] = [dayi, pricei] indicates the price of the stock on day dayi is pricei. A **line chart** is created from the array by plotting the points on an XY plane with the X-axis representing the day and the Y-axis representing the price and connecting adjacent points. One such example is shown below:



Return *the****minimum number of lines****needed to represent the line chart*.

**Example 1:**

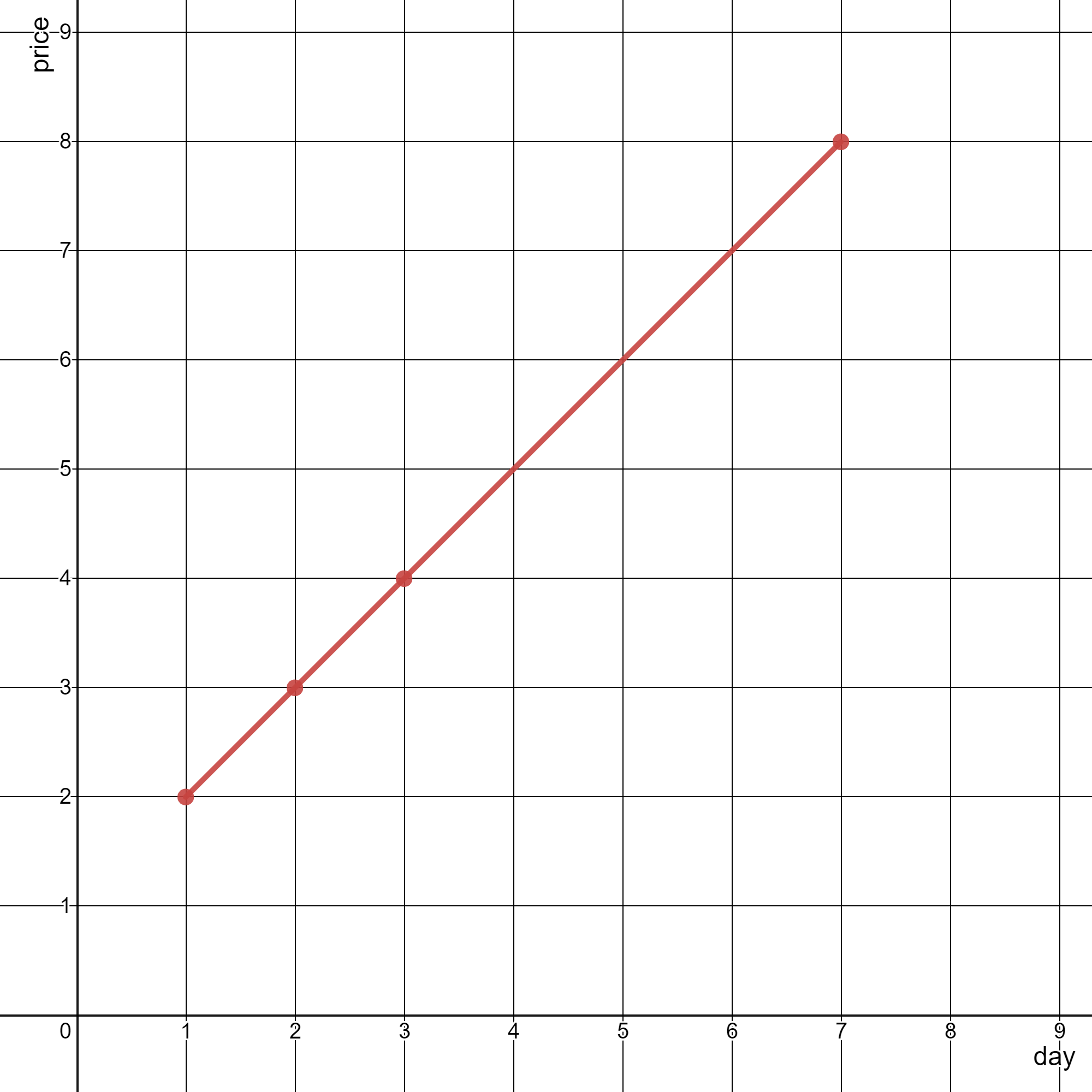


**Input:** stockPrices = [[1,7],[2,6],[3,5],[4,4],[5,4],[6,3],[7,2],[8,1]]

**Output:** 3

**Explanation:**The diagram above represents the input, with the X-axis representing the day and Y-axis representing the price.The following 3 lines can be drawn to represent the line chart:- Line 1 (in red) from (1,7) to (4,4) passing through (1,7), (2,6), (3,5), and (4,4).- Line 2 (in blue) from (4,4) to (5,4).- Line 3 (in green) from (5,4) to (8,1) passing through (5,4), (6,3), (7,2), and (8,1).It can be shown that it is not possible to represent the line chart using less than 3 lines.

**Example 2:**



**Input:** stockPrices = [[3,4],[1,2],[7,8],[2,3]]

**Output:** 1

**Explanation:**As shown in the diagram above, the line chart can be represented with a single line.

**Constraints:**

1 <= stockPrices.length <= 10^5

stockPrices[i].length == 2

1 <= dayi, pricei <= 10^9

All dayi are **distinct**.

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

**Solution 1: Math + Sorting (10min)**

class Solution {

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

        int n = stockPrices.length;

        if(n == 1) {

            return 0;

        }

        if(n == 2) {

            return 1;

        }

        int lines = 1;

        // Sort the points based on x-coordinate

        Arrays.sort(stockPrices, (a, b) -> a[0] - b[0]);

        // Calculate the initial slope between the first two points

        // To avoid division and floating point precision issues, use cross multiplication

        // Slope between points i and i+1: (y2 - y1)/(x2 - x1)

        // Compare slopes using cross multiplication: (y2 - y1)\*(x3 - x2) == (y3 - y2)\*(x2 - x1)

        long x1 = stockPrices[0][0];

        long y1 = stockPrices[0][1];

        long x2 = stockPrices[1][0];

        long y2 = stockPrices[1][1];

        long prevDeltaX = x2 - x1;

        long prevDeltaY = y2 - y1;

        for(int i = 2; i < n; i++) {

            long x3 = stockPrices[i][0];

            long y3 = stockPrices[i][1];

            long deltaX = x3 - x2;

            long deltaY = y3 - y2;

            if(prevDeltaY \* deltaX != deltaY \* prevDeltaX) {

                lines++;

                // Only need to update previous deltaX or Y

                // when the slope changed

                prevDeltaX = deltaX;

                prevDeltaY = deltaY;

            }

            // No need update x1, y1 since that's only for

            // initial point, but in the for loop no initial

            // point involved

            x2 = x3;

            y2 = y3;

        }

        return lines;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Refer to Deepseek**

To solve this problem, we need to determine the minimum number of line segments required to represent a given line chart. The line chart is defined by a series of points, and we need to connect these points with the fewest possible straight lines such that each line segment covers as many consecutive points as possible that lie on the same straight line.

**Approach**

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

**Sort the Points: Ensure the points are sorted by their x-coordinates to process them in order.**

**Calculate Slopes Between Consecutive Points: For each pair of consecutive points, calculate the slope. If the slope between the current pair of points is the same as the slope between the previous pair, they can be part of the same line segment.**

Count Line Segments: Iterate through the points, comparing the current slope with the previous slope. If they differ, increment the line segment count.

**Solution Code**

class Solution {

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

int n = stockPrices.length;

if (n <= 2) {

return n <= 1 ? 0 : 1;

}

// Sort the points based on x-coordinate

Arrays.sort(stockPrices, Comparator.comparingInt(a -> a[0]));

int lines = 1;

// Calculate the initial slope between the first two points

// To avoid division and floating point precision issues, use cross multiplication

// Slope between points i and i+1: (y2 - y1)/(x2 - x1)

// Compare slopes using cross multiplication: (y2 - y1)\*(x3 - x2) == (y3 - y2)\*(x2 - x1)

long x1 = stockPrices[0][0];

long y1 = stockPrices[0][1];

long x2 = stockPrices[1][0];

long y2 = stockPrices[1][1];

long prevDeltaX = x2 - x1;

long prevDeltaY = y2 - y1;

for (int i = 2; i < n; i++) {

long x3 = stockPrices[i][0];

long y3 = stockPrices[i][1];

long currDeltaX = x3 - x2;

long currDeltaY = y3 - y2;

// Check if the slopes are the same: (y2 - y1)\*(x3 - x2) == (y3 - y2)\*(x2 - x1)

if (prevDeltaY \* currDeltaX != prevDeltaX \* currDeltaY) {

lines++;

prevDeltaX = currDeltaX;

prevDeltaY = currDeltaY;

}

x2 = x3;

y2 = y3;

}

return lines;

}

}

**Explanation**

Sorting: The points are sorted by their x-coordinates to ensure they are processed in order from left to right.

Initial Setup: For the first two points, the initial slope is calculated using differences in their coordinates. This slope is stored for comparison with subsequent slopes.

Slope Comparison: For each subsequent point, the slope between the current point and the previous point is compared with the stored slope. If they differ, it indicates a new line segment is needed, and the line count is incremented. The stored slope is then updated to the new slope.

Cross Multiplication: To avoid floating-point precision issues, cross multiplication is used to compare slopes. This ensures accurate comparisons without division.

**Time Complexity**

This approach efficiently determines the minimum number of line segments by leveraging slope comparisons and cross multiplication to handle precision, ensuring optimal performance and correctness.

The time complexity of the provided solution is O(n log n), where n is the number of points in the stockPrices array. Here's the breakdown:

Sorting the Points:

Sorting the array of points based on their x-coordinates takes O(n log n) time. This is because Java's Arrays.sort() uses a variant of the TimSort algorithm, which has an average and worst-case time complexity of O(n log n).

Iterating Through Points:

After sorting, the algorithm iterates through the sorted points once to compute the minimum number of lines. This iteration runs in O(n) time since each point is processed exactly once.

Slope Comparisons:

Each slope comparison between consecutive points is done in O(1) time using cross multiplication to avoid floating-point precision issues.

**Space Complexity**

The space complexity is O(1) (constant space) if we disregard the input storage, as the algorithm only uses a few extra variables (prevDeltaX, prevDeltaY, etc.) and does not use additional data structures that grow with the input size.

**Summary**

Time Complexity: O(n log n) (due to sorting).

Space Complexity: O(1) (constant extra space).

This approach efficiently minimizes the number of line segments by leveraging sorting and linear traversal, making it optimal for the given problem constraints.

**Refer to**

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

[L356.Line Reflection (Ref.L149)](note://WEB8d586c79a665a9079d29d00042cf331c)

[L2152.Minimum Number of Lines to Cover Points (Ref.L149)](note://WEB86d93c200721692c35251879f14ef4e5)