2280. Minimum Lines to Represent a Line Chart

Medium Geometry Array Math Number Theory Sorting Leetcode Link

Problem Description

We have a 2D integer array named `stockPrices`, where each element consists of two integers representing a day and the corresponding stock price on that day. The task is to determine the minimum number of straight lines required to connect all the points plotted on an XY plane, with the day as the X-axis and the price as the Y-axis, in the given order after sorting the data by the day. Essentially, we want to find the fewest lines that connect each adjacent point (each day to the next) to form a line chart.

Intuition

To solve this problem, the solution leverages the concept of slopes of lines. When connecting points to draw the line chart, we can continue using the same line as long as consecutive segments share the same slope. When the slopes differ, we need a new line.

To find the slope of the line segment between two points (x, y) and (x1, y1), we calculate `(y1 - y) / (x1 - x)`, which gives us the rate of change in price with respect to days. Now, if we compare this slope with the next segment's slope between the points (x1, y1) and (x2, y2), we would normally compute `(y2 - y1) / (x2 - x1)` and compare. To avoid division and possible precision issues with floating-point numbers, we instead compare cross-products: `(y1 - y) * (x2 - x1)` and `(y2 - y1) * (x1 - x)`. If these two products are not equal, the slope has changed, and we need an additional line.

The approach first sorts the `stockPrices` based on the day to ensure the points are in the correct order. Then, iterating through each pair of adjacent points, we calculate the aforementioned cross-product to decide whether the current segment can be connected with the prior one or if a new line has to start. The variable `ans` keeps track of the number of lines needed.

Remember, we increment the number of lines when a change in slope is detected between segments, and after calculating the slopes, we update the `dx` and `dy` to be the differences between the current and next points to be ready for the next iteration.

Solution Approach

The implementation given in the reference solution approach follows these steps:

1. **Sort the Input**: We start by sorting the `stockPrices` array. This ensures that the stock prices are ordered by day, which is essential as we need to connect points from one day to the next.
2. **Initialize Variables**: Two variables `dx` and `dy` are initialized to store the differences in the `x` (day) and `y` (price) coordinates of successive points. Additionally, `ans` is initialized to count the number of lines required.
3. **Iterate Through Points**: We use a loop to go through each pair of adjacent points in the sorted array. To do this efficiently, the code leverages the `pairwise` utility, which is not a standard Python function but likely a user-defined or library function that yields pairs of elements from the input list.
4. **Calculate and Compare Slopes**: For each pair `(x, y)` and `(x1, y1)`, we calculate the change in `x` as `dx1 = x1 - x` and the change in `y` as `dy1 = y1 - y`. We then compare the products of cross-multiplying these with the previous segment's `dx` and `dy` to check if the slopes match. If `dy * dx1` does not equal `dx * dy1`, the slope has changed, indicating a need for a new line.
5. **Increment Line Count**: When a slope change is detected, we increment `ans` by 1.
6. **Update Differences**: Whether a new line is added or not, we update `dx` and `dy` with the differences `dx1` and `dy1`, to use them in the next iteration for slope comparison.
7. **Return Result**: After all points have been processed, the variable `ans` is returned. This gives us the minimum number of lines needed to represent the chart.

Here is the explanation of the algorithm, data structure, or pattern used in each step:

- **Sorting**: It's the first step and is critical for comparing the slopes of segments in the correct order.
- **Looping and Iteration**: The use of a loop along with pairwise comparison is a common pattern when we need to check adjacent elements in a sequence.
- **Slope Comparison**: This step avoids division by comparing slopes using cross-multiplication, which is a mathematical technique to avoid floating-point precision errors and ensure that comparisons are accurate when checking for collinearity between points.

By systematically comparing each pair of points, the solution efficiently calculates the number of lines needed to form the chart without redundant calculations or comparisons.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have the following 2D array `stockPrices`:

```
stockPrices = [[1, 100], [2, 200], [3, 300], [4, 200], [5, 100], [6, 200]]
```

1. **Sort the Input**:
    - Since `stockPrices` is already sorted by the day (the first element of each sub-array), we don't need to sort it again.
2. **Initialize Variables**:
    - We set `dx` and `dy` to 0 as there is no previous point to compare with yet.
    - `ans` is initialized to 1 because we need at least one line to connect the first point to the next one.
3. **Iterate Through Points**:
    - Imagine comparing the first point `[1, 100]` with the second `[2, 200]`. This sets our initial `dx` and `dy` to `[2-1, 200-100] = [1, 100]`.
4. **Calculate and Compare Slopes**:
    - On the next iteration, we compare the point `[2, 200]` with `[3, 300]`. We calculate the changes `dx1 = 3 - 2` and `dy1 = 300 - 200`, resulting in `[1, 100]`.
    - Now we compare the slopes. Since `dy * dx1` = `dx * dy1` (100 * 1 = 1 * 100), the slope hasn't changed, and we stay on the same line.
5. **Increment Line Count**:
    - We don't increment `ans` as the line didn't change.
6. **Update Differences**:
    - We set `dx` and `dy` to `dx1` and `dy1` respectively, for the next comparison.

Continue this process for the remaining points:

7. Comparing `[3, 300]` with `[4, 200]`, we calculate `dx1 = 4 - 3` and `dy1 = 200 - 300`, resulting in `[1, -100]`.
    - Slopes differ (`dy * dx1` = `100 * 1` != `-1 * 100` = `dx * dy1`), so we increment `ans` to 2 and update `dx` and `dy`.
8. Comparing `[4, 200]` with `[5, 100]`, `dx1 = 5 - 4` and `dy1 = 100 - 200`, resulting in `[1, -100]`.
    - Slopes match (as they're both `[1, -100]`), so we don't increment `ans` and just update `dx` and `dy`.
9. Finally, comparing `[5, 100]` with `[6, 200]`, `dx1 = 6 - 5` and `dy1 = 200 - 100`, resulting in `[1, 100]`.
    - Slopes differ again (`dy * dx1` = `-100 * 1` != `1 * 100` = `dx * dy1`), so we increment `ans` to 3.
10. **Return Result**:
    - After all points have been processed, we conclude that we need `ans` = 3 lines to connect all the points.

With this hands-on example, we've illustrated how the solution approach can be applied to determine the minimum number of lines required to connect a series of points in a line chart.

Python Solution

```python
from itertools import pairwise
from typing import List

class Solution:
    def minimumLines(self, stock_prices: List[List[int]]) -> int:
        # Sort the stock prices based on the days
        stock_prices.sort()

        # Initialize the previous differences in x and y to 0 and 1 respectively as placeholders.
        prev_diff_x, prev_diff_y = 0, 1

        # Initialize the count of lines needed to 0
        num_lines = 0

        # Iterate over each pair of adjacent stock prices
        for (current_x, current_y), (next_x, next_y) in pairwise(stock_prices):
            # Compute the differences in x and y for the current pair
            diff_x, diff_y = next_x - current_x, next_y - current_y

            # If the cross product of the differences does not equal to zero,
            # then the points are not on the same line. Increment the number of lines.
            if prev_diff_y * diff_x != prev_diff_x * diff_y:
                num_lines += 1

            # Update the previous differences for the next iteration
            prev_diff_x, prev_diff_y = diff_x, diff_y

        # Return the total number of lines needed
        return num_lines

```

Java Solution

```java
class Solution {
    public int minimumLines(int[][] stockPrices) {
        // Sort the stock prices array based on the day (the 0th element of each sub-array);
        // if the days are equal, sort based on the price (the 1st element of each sub-array)
        Arrays.sort(stockPrices, (a, b) -> Integer.compare(a[0], b[0]));

        // Initialize variables to keep track of previous slope components
        int prevDeltaX = 0, prevDeltaY = 1; // Initiated to 0 and 1 to handle the starting calculation
        int lineCount = 0; // Counter for the minimum lines needed

        // Iterate over the stock prices starting from the second price
        for (int i = 1; i < stockPrices.length; ++i) {
            // Get the previous day's stock price
            int previousX = stockPrices[i - 1][0], previousY = stockPrices[i - 1][1];
            // Get the current day's stock price
            int currentX = stockPrices[i][0], currentY = stockPrices[i][1];

            // Calculate the change in x and y (i.e., the slope components) for the current line
            int deltaX = currentX - previousX, deltaY = currentY - previousY;

            // Check if the current line has a different slope than the previous one
            // by comparing cross products of the slope components (to avoid floating-point division)
            if (prevDeltaY * deltaX != prevDeltaX * deltaY) {
                ++lineCount; // A different slope means a new line is needed
            }

            // Update previous slope components for the next iteration
            prevDeltaX = deltaX;
            prevDeltaY = deltaY;
        }

        // Return the total number of lines needed to connect all points with straight lines
        return lineCount;
    }
}

```

C++ Solution

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

class Solution {
public:
    int minimumLines(std::vector<std::vector<int>>& stockPrices) {
        // Sort stockPrices by their x-value (date)
        std::sort(stockPrices.begin(), stockPrices.end());

        // Initialize previous difference in x and y to compare slopes.
        // Starting with an impossible value for the very first comparison.
        int prevDeltaX = 0, prevDeltaY = 1;

        // Begin with no lines drawn
        int numLines = 0;

        // Iterate through stockPrices starting from the second point
        for (int i = 1; i < stockPrices.size(); ++i) {
            // Get x and y of the current and previous points
            int currX = stockPrices[i - 1][0], currY = stockPrices[i - 1][1];
            int nextX = stockPrices[i][0], nextY = stockPrices[i][1];

            // Calculate differences in x and y for the current segment
            int deltaX = nextX - currX, deltaY = nextY - currY;

            // If the cross product of the two segments is not zero,
            // they are not collinear, so the slope has changed and a new line is needed.
            if (static_cast<long long>(prevDeltaY) * deltaX != static_cast<long long>(prevDeltaX) * deltaY) {
                ++numLines;
                // Update previous differences to current one for next iteration comparison.
                prevDeltaX = deltaX;
                prevDeltaY = deltaY;
            }
        }

        // Return the total number of lines needed
        return numLines;
    }
};

```

Typescript Solution

```typescript
function minimumLines(stockPrices: number[][]): number {
    // Get the number of stock price entries
    const stockPricesCount = stockPrices.length;

    // Sort the stock prices based on the first value of each pair (the dates are sorted)
    stockPrices.sort((a, b) => a[0] - b[0]);

    // Initialize the number of lines needed to connect all points
    let linesRequired = 0;

    // Initialize a variable to store the previous slope as a pair of BigInts for accurate calculation
    let previousSlope: [BigInt, BigInt] = [BigInt(0), BigInt(0)];

    // Start from the second point and compare slopes of consecutive points
    for (let i = 1; i < stockPricesCount; i++) {
        // Get the coordinates of the current point and the previous point
        const [previousX, previousY] = stockPrices[i - 1];
        const [currentX, currentY] = stockPrices[i];

        // Calculate the difference in x and y using BigInt for accuracy (to handle very large numbers)
        const differenceX = BigInt(currentX - previousX);
        const differenceY = BigInt(currentY - previousY);

        // If it's the first line or the slope differs from the previous slope, increment the line count
        if (i == 1 || differenceX * previousSlope[1] !== differenceY * previousSlope[0]) {
            linesRequired++;
        }

        // Update the previous slope to the current slope for the next iteration
        previousSlope = [differenceX, differenceY];
    }

    // Return the total number of lines required
    return linesRequired;
}

```

Time and Space Complexity

Time Complexity

The given Python code first sorts the `stockPrices`. Sorting takes `O(n log n)` time where `n` is the number of `stockPrices`.

Then, it iterates through the sorted prices to determine the number of lines required. This iteration involves a single pass through the `stockPrices` list, which takes `O(n)` time.

Inside the loop, it performs constant-time arithmetic operations to determine if a new line segment is required. Therefore, these operations do not increase the overall complexity beyond `O(n)`.

Since sorting is the most time-consuming operation and is followed by a linear pass, the total time complexity of this algorithm is `O(n log n) + O(n)`, which simplifies to `O(n log n)`.

Space Complexity

The space complexity is determined by the additional space required by the algorithm besides the input.

Here, the algorithm uses a fixed amount of space to store temporary values like `dx`, `dy`, `dx1`, and `dy1`, so it uses `O(1)` additional space.

Sorting is typically done in-place (especially if the Python `sort()` method is used), therefore, the space complexity remains `O(1)`.

In summary, the space complexity of the algorithm is `O(1)`.