3009. Maximum Number of Intersections on the Chart

Hard Binary Indexed Tree Geometry Array Math

Problem Description

The problem presents a line chart with `n` points, each point corresponding to a given y-coordinate from an integer array `y`. Each point on the chart has the coordinates `(k, y[k])` where `k` is a 1-indexed position denoting the x-coordinate and `y[k]` is the y-coordinate of the k-th point. All points are connected by line segments to form a polyline, and it is mentioned that there are no horizontal line segments, meaning that the y-coordinates of every two consecutive points are different.

The task is to calculate the maximum number of intersection points that a horizontal line, which extends infinitely in both directions, can have with the given chart. The output should be the maximum number of times any such horizontal line crosses the line chart.

Intuition

When looking for the maximum number of intersections, we should consider that the number of intersections for a continuous polyline will increase or decrease as we move the horizontal line up or down.

The insight into solving this problem is to find all potential y-values where intersection count changes, which happen at or between the y-values of the given points. Instead of using a brute force approach and checking all possible y-values, we can optimize by only considering the y-values where changes occur.

Each segment between two points on the chart can intersect with a horizontal line at most once. If two consecutive points have y-coordinates such that `y[i] < y[i + 1]`, the segment between them will intersect with any horizontal line with y-coordinate between `y[i]` and `y[i + 1]` - and similarly, if `y[i] > y[i + 1]`, between `y[i + 1]` and `y[i]`.

To keep track of the changes, we can use a `TreeMap` to simulate the process. The keys of the `TreeMap` will be significant y-values multiplied by 2 (to handle the situation where the start and end y-values could be half values, since points can intersect at mid-points between integer y-values). The corresponding values will be the change in the intersection count happening at that y-coordinate.

We iterate through the segments and update the TreeMap where the start of the segment increments the intersection count and the end of the segment decrements it. While updating the TreeMap, the merge function is used to add or subtract counts for specific y-values. After processing all segments, iterate through the TreeMap to find the maximum number of intersections by summing changes to the intersection count and keeping track of the maximum value encountered.

The solution code implements this method and finds the maximum intersection count efficiently.

Solution Approach

The solution approach leverages the TreeMap data structure, which is a Red-Black tree-based NavigableMap implementation. It is used here because of its ability to maintain sorted keys, which allows for quick insertions and deletions while also providing an efficient means of iteratively processing in a sorted order.

The main algorithm proceeds as follows:

1. Initialize a TreeMap called `line` and two integer variables: `ans` to track the maximum number of intersections and `intersectionCount` to keep a cumulative count of intersections as we iterate through the changes.
2. Iterate over each segment in the line chart, looking at the pairs of points (i.e., `(y[i-1], y[i])` for `i` ranging from 1 to `n-1`), since the points are 1-indexed.
3. For each segment, calculate the starting and ending coordinates by doubling the y-values (`2 * y[i-1]` and `2 * y[i]`). Doubling is done to handle cases where the intersection occurs at exact halves between integers.
4. Conditionally adjust the ending coordinate by adding or subtracting 1 to prevent overlapping counts on segment borders (except for the last point).
5. Update the TreeMap using the `merge` method, which either adds a new key with the value or updates the existing key with the provided remapping function (`Integer::sum` in this case). Increment the intersection count at the start of the segment and decrement it after the end of the segment.
6. After populating the TreeMap with all the potential y-values and their corresponding changes, iterate through the values of the TreeMap.
7. As we traverse the TreeMap, add each value (change in intersection count) to `intersectionCount`. If `intersectionCount` exceeds `ans` at any point, update `ans` with the value of `intersectionCount`, thus keeping track of the maximum intersection count at any y-coordinate.
8. After completing the TreeMap traversal, return `ans` as the maximum number of intersections.

The use of TreeMap is crucial because it allows us to abstract away the process of sorting and efficiently computing the cumulative changes while iterating over the y-values. By simulating this process, we avoid calculating the intersection count for every possible y-value and instead only focus on those that actually contribute to an increase or decrease in the intersection count, thus optimizing the solution.

Example Walkthrough

Let's assume we have a line chart with 5 points (n=5) and the corresponding y-coordinates are as follows: [2, 3, 1, 4, 2]. These points translate to the coordinates: (1, 2), (2, 3), (3, 1), (4, 4), (5, 2) when considering their positions in our 1-indexed chart.

Following the solution approach, here's a step-by-step illustration of how to find the maximum number of intersections a horizontal line would have with the polyline formed by these points:

1. Initialize `TreeMap line` and integer variables `ans = 0` and `intersectionCount = 0`.
2. We have 4 segments in the line chart: between points (1, 2), (2, 3); (2, 3), (3, 1); (3, 1), (4, 4); and (4, 4), (5, 2). Iterate over these segments to determine start and end coordinates and update the TreeMap accordingly.
3. For the first segment from (1, 2) to (2, 3), we double the y-values and get (4, 6) as our starting and ending coordinates. Update the TreeMap:
   - `line.merge(4, 1, Integer::sum)` → Starting y-value; increments intersection count.
   - `line.merge(6, -1, Integer::sum)` → Ending y-value; decrements intersection count.
4. For the second segment from (2, 3) to (3, 1), doubled y-values are (6, 2). Since we decrement after the end, adjust the endpoint (1) to not overlap, making it (6, 1):
   - `line.merge(6, 1, Integer::sum)` → Overlaps with the endpoint of the first segment.
   - `line.merge(1, -1, Integer::sum)` → Adjusted starting y-value; decrements intersection count.
5. Repeat the above step for the remaining segments.
6. The TreeMap now stores all the points where the intersection count potentially changes with keys representing y-values (doubled and adjusted) and values representing the change in the intersection count at those points.
7. Iterate through the TreeMap, while adding each value to `intersectionCount` and updating `ans` if `intersectionCount` exceeds the current `ans`:
   - `intersectionCount += line.getKey();`
   - `ans = Math.max(ans, intersectionCount).`
8. After the TreeMap is fully traversed, `ans` will hold the maximum number of intersections, which is the answer we seek to find.

Using this method, let's calculate the number of intersections based on the values we have populated in our TreeMap:

Starting with `intersectionCount = 0` and considering TreeMap entries as (y-value, change):

- At y-value 1: `intersectionCount += -1` → `intersectionCount = -1`.
- At y-value 4: `intersectionCount += 1` → `intersectionCount = 0`.
- At y-value 6: `intersectionCount += 1 -1` → `intersectionCount = 0`.
- At y-value 8: `intersectionCount += 1` → `intersectionCount = 1` (update `ans` to 1).
- At y-value 14: `intersectionCount += -1` → `intersectionCount = 0`.

Therefore, the maximum number of intersections a horizontal line can have with this polyline is `ans = 1`.

Solution Implementation

Python

```python
class Solution:
    def max_intersection_count(self, v):
        n = len(v)  # Number of elements in the list v
        max_intersections = 0  # Holds the maximum count of intersections
        current_intersection_count = 0  # Tracks the current count of intersections
        sweep_line = {}  # Dictionary to simulate the sweep line algorithm

        # Iterate over elements of the array to populate the dictionary
        for i in range(1, n):
            # Calculate the starting point of the line segment
            start = 2 * v[i - 1]
            # Calculate the ending point of the line segment
            # Adjust for the last point or if the current v is greater than the previous v
            end = 2 * v[i] + (0 if i == n - 1 else (-1 if v[i] > v[i - 1] else 1))
            # Merge the start points into dictionary, incrementing the count for intersections
            sweep_line[start, end] = sweep_line.get((start, end), 0) + 1
            # Merge the end points into dictionary, decrementing the count for intersections
            sweep_line[end + 1] = sweep_line.get(end + 1, 0) - 1

        # Sort the keys of the dictionary to simulate the behavior of TreeMap
        sorted_keys = sorted(sweep_line)

        # Iterate through the sorted keys
        for key in sorted_keys:
            # Update the current intersection count
            current_intersection_count += sweep_line[key]
            # Update the maximum intersection count found so far
            max_intersections = max(max_intersections, current_intersection_count)

        return max_intersections  # Return the maximum number of intersections found
```

Java

```java
class Solution {
    public int maxIntersectionCount(int[] v) {
        final int n = v.length; // Number of elements in the array v
        int maxIntersections = 0; // Holds the maximum intersection count
        int currentIntersectionCount = 0; // Tracks the current count of intersections
        TreeMap<Integer, Integer> sweepLine = new TreeMap<>(); // TreeMap to simulate the sweep line algorithm

        // Iterate over the elements of the array to populate the TreeMap
        for (int i = 1; i < n; ++i) {
            // Calculate the starting point of the line segment
            final int start = 2 * v[i - 1];
            // Calculate the ending point of the line segment
            // Adjust for the last point or if the current v is more than the previous v
            final int end = 2 * v[i] + (i == n - 1 ? 0 : (v[i] > v[i - 1] ? -1 : 1));
            // Add or update the starting point in the map
            sweepLine.merge(start, 1, Integer::sum);
            // Merge the end points into TreeMap, decrementing the count for intersections
            sweepLine.merge(Math.max(start, end) + 1, -1, Integer::sum);
        }

        // Iterate through the TreeMap
        for (final int count : sweepLine.values()) {
            // Update the current intersection count
            currentIntersectionCount += count;
            // Update the maximum intersection count found so far
            maxIntersections = Math.max(maxIntersections, currentIntersectionCount);
        }

        return maxIntersections; // Return the max number of intersections found
    }
}
```

C++

```cpp
#include <map>

class Solution {
public:
    int maxIntersectionCount(std::vector<int>& v) {
        const int n = v.size(); // Number of elements in the vector v
        int maxIntersections = 0; // Holds the maximum intersection count
        int currentIntersectionCount = 0; // Tracks the current count of intersections
        std::map<int, int> sweepLine; // Map to simulate the sweep line algorithm

        // Iterate over the elements of the vector to populate the map
        for (int i = 1; i < n; ++i) {
            // Calculate the starting point of the line segment
            const int start = 2 * v[i - 1];
            // Calculate the ending point of the line segment
            // Adjust for the last point or if the current v is more than the previous v
            const int end = 2 * v[i] + (i == n - 1 ? 0 : (v[i] > v[i - 1] ? -1 : 1));
            // Add or update the starting point in the map
            ++sweepLine[std::min(start, end)];
            startCount++;
            // Add or update the ending point in the map
            --sweepLine[std::max(start, end) + 1];
            endCount--;
        }

        // Iterate through the map
        for (auto it = sweepLine.begin(); it != sweepLine.end(); ++it) {
            // Update the current intersection count
            currentIntersectionCount += it->second;
            // Update the maximum intersection count found so far
            maxIntersections = std::max(maxIntersections, currentIntersectionCount);
        }

        // Return the max number of intersections found
        return maxIntersections;
    }
};
```

TypeScript

```typescript
// Defining a type alias for clarity
type TreeMap = Map<number, number>;

function maxIntersectionCount(v: number[]): number {
    const n: number = v.length; // Number of elements in the array v
    let maxIntersections: number = 0; // Holds the maximum intersection count
    let currentIntersectionCount: number = 0; // Tracks the current count of intersections
    let sweepLine: TreeMap = new Map<number, number>(); // TreeMap to simulate the sweep line algorithm

    // Iterate over the elements of the array to populate the TreeMap
    for (let i: number = 1; i < n; ++i) {
        // Calculate the starting point of the line segment
        const start: number = 2 * v[i - 1];
        // Calculate the ending point of the line segment
        // Adjust for the last point or if the current v is more than the previous v
        const end: number = 2 * v[i] + (i === n - 1 ? 0 : (v[i] > v[i - 1] ? -1 : 1));

        // Merge the start points into TreeMap, incrementing the count for intersections
        sweepLine.set(Math.min(start, end), (sweepLine.get(Math.min(start, end)) || 0) + 1);
        // Merge the end points into TreeMap, decrementing the count for intersections
        sweepLine.set(Math.max(start, end) + 1, (sweepLine.get(Math.max(start, end) + 1) || 0) - 1);
    }

    // Iterate through the TreeMap
    sweepLine.forEach((count: number) => {
        // Update the current intersection count
        currentIntersectionCount += count;
        // Update the maximum intersection count found so far
        maxIntersections = Math.max(maxIntersections, currentIntersectionCount);
    });

    return maxIntersections; // Return the max number of intersections found
}
```

```python
class Solution:
    def max_intersection_count(self, v):
        n = len(v)  # Number of elements in the list v
        max_intersections = 0  # Holds the maximum count of intersections
        current_intersection_count = 0  # Tracks the current count of intersections
        sweep_line = {}  # Dictionary to simulate the sweep line algorithm

        # Iterate over elements of the array to populate the dictionary
        for i in range(1, n):
            # Calculate the starting point of the line segment
            start = 2 * v[i - 1]
            # Calculate the ending point of the line segment
            # Adjust for the last point or if the current v is greater than the previous v
            end = 2 * v[i] + (0 if i == n - 1 else (-1 if v[i] > v[i - 1] else 1))
            # Merge the start points into dictionary, incrementing the count for intersections
            sweep_line[start, end] = sweep_line.get((start, end), 0) + 1
            # Merge the end points into dictionary, decrementing the count for intersections
            sweep_line[end + 1] = sweep_line.get(end + 1, 0) - 1

        # Sort the keys of the dictionary to simulate the behavior of TreeMap
        sorted_keys = sorted(sweep_line)

        # Iterate through the sorted keys
        for key in sorted_keys:
            # Update the current intersection count
            current_intersection_count += sweep_line[key]
            # Update the maximum intersection count found so far
            max_intersections = max(max_intersections, current_intersection_count)

        return max_intersections  # Return the maximum number of intersections found
```

Time and Space Complexity

The time complexity of the code is determined by several loops and operations on the TreeMap `line`.

- The first `for` loop runs `(n - 1)` times, where `n` is the length of the array `v`.
- Inside the loop, there are two `merge` operations on the TreeMap. TreeMap operations typically take `O(log m)` time where `m` is the number of keys in the map because TreeMap is implemented with a Red-Black tree, which maintains a sorted order of its keys. As there are 2 merge operations for each `i` in the loop, they contribute `O(2*(n-1)*log m)` to the time complexity.
- The second `for` loop iterates through the values of the TreeMap `line`. In the worst case, the TreeMap can have `2*(n-1)` keys if all start and end points are different. Iterating through the values therefore takes `O(m)` time, where `m` can be as large as `2*(n-1)` in the worst case scenario.

Putting it all together, the total time complexity is `O((n-1)*log m) + O(m)`. Since `m` is at most `2*(n-1)`, we can consider the complexity to be `O(n log n)` in the worst case.

The space complexity of the code is primarily dictated by the size of the TreeMap `line`.

- In the worst case scenario, `line` may contain `2*(n-1)` key-value pairs if each segment has a unique starting and ending point.
- Since the TreeMap stores all the starting and ending points, its size could be `O(2*(n-1))`, which simplifies to `O(n)`.

Therefore, the space complexity of the code is `O(n)`.