1637. Widest Vertical Area Between Two Points Containing No Points

Medium Array Sorting Leetcode Link

Problem Description

The given problem requires us to determine the widest vertical gap between any two given points on a 2D plane, with the condition that no points should be inside the area. The input is a list of `n` points, where each point is represented as a coordinate pair `[x_i, y_i]`. The goal is to find the maximum width between two lines that are parallel to the y-axis such that the lines do not pass through any of the given points.

Intuition

To solve this problem, the main insight is to focus solely on the x-coordinates, since the vertical area extends infinitely along the y-axis. Essentially, we want to find two x-coordinates that have the greatest distance between them without any other x-coordinates lying in between.

The initial step is straightforward: we extract the x-coordinates from the provided points. With this list of x-coordinates alone, we want to identify the largest gap. A brute force approach could involve comparing every pair of x-coordinates to calculate the gap, but this is inefficient as the number of comparisons grows quickly with the number of points (`O(n^2)` comparisons).

A key realization is that we can find the largest gap more efficiently. If the x-coordinates were sorted, we could simply iterate through them and measure the gaps between consecutive x-coordinates, because the largest gap must be between two consecutive points in a sorted list. Sorting reduces the problem to a linear pass through the points, leading to a `O(n log n)` solution due to the sorting step.

However, the given solution uses a bucketing approach, which is more complex. Bucketing can be a clever optimization when the points are distributed over a large range and we expect to find an even wider vertical area. The idea is to divide the range of x-coordinates into equal-width intervals, or buckets, then place the x-coordinates into the corresponding buckets. We can then find the maximum gap by looking only at the maximum and minimum values in each bucket, and observing the gaps between the max of one bucket and the min of the next non-empty bucket. This approach can achieve better than `O(n log n)` performance under certain distributions of input. The complexity of the bucket approach in the solution provided however remains `O(n)` because it consists of creating the buckets and then iterating through them, which both happen in linear time.

In the given solution, a bucket size is calculated, and then each x-coordinate is assigned to a bucket. After that, the maximum and minimum x-values in each bucket are used to find the largest gap between consecutive non-empty buckets.

Solution Approach

In the provided solution, we seek to find the widest gap between the x-coordinates of the given points without any points lying in that area. Here's how the solution is implemented:

- We start by extracting all the x-coordinates from the given points into a list named `nums`.
- The range of x-coordinates is computed by finding the minimum `mi` and maximum `mx` values in `nums`.
- Instead of sorting all the x-coordinates, we use bucketing to group the values. The bucket size is chosen as the maximum of 1 and the result of the integer division of the range `(mx - mi)` by `(n - 1)`, where `n` is the number of points. This determines how many values will be placed in each bucket on average.
- The buckets are initialized as a list of pairs `[inf, -inf]` where we will keep the minimum (first element) and maximum (second element) x-coordinate that falls into each bucket.
- We iterate over `nums` and assign each x-coordinate to the appropriate bucket. An x-coordinate `x` is placed in the bucket at index `(x - mi) // bucket_size`. Inside the bucket, we update the minimum or maximum value as necessary.
- Once the buckets are populated, we iterate through them to find the maximal gap. We use two variables: `prev`, which keeps track of the rightmost x-coordinate we've seen so far (initialized to `inf`), and `ans`, which will store the solution.
- During the bucket iteration, we only consider non-empty buckets, determined by checking if `curmin <= curmax`. For each non-empty bucket, we calculate the gap as `curmin - prev`, which is the distance between the current bucket's leftmost point and the previous bucket's rightmost point. We update `ans` with the maximum of itself and the current gap.
- `prev` is then updated to `curmax`, handling the next iteration's calculations.
- After going through all buckets, `ans` holds the largest gap found, which is our final answer.

This approach cleverly exploits bucketing to essentially perform a sort of "coarse" sorting, allowing us to bypass the need to sort all the x-coordinates and still find the widest vertical area. It systematically reduces the problem into smaller manageable subproblems that can be solved quickly in linear time.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we are given the following points on a 2D plane: `[(4, 10), (2, 6), (12, 8), (8, 5)]`.

1. First, we extract the x-coordinates from the given points: `nums = [4, 2, 12, 8]`.
2. We calculate the range by finding the minimum and maximum values in `nums`, which are `mi = 2` and `mx = 12`.
3. Next, we determine the bucket size. The range `(mx - mi) = (12 - 2) = 10`, and we have `n = 4` points. So, the bucket size is `max(1, 10 // (4 - 1)) = max(1, 3) = 3`.
4. We initialize four buckets with `[inf, -inf]`, because we have `(mx - mi) // bucket_size + 1 = (10 // 3) + 1 = 4` buckets.
   - Buckets: `[[inf, -inf], [inf, -inf], [inf, -inf], [inf, -inf]]`
5. We iterate over `nums` and assign each x-coordinate to the correct bucket:
   - `4` goes into bucket `0` (since `(4 - 2) // 3 = 0`)
   - `2` goes into bucket `0` (since `(2 - 2) // 3 = 0`)
   - `12` goes into bucket `3` (since `(12 - 2) // 3 = 3`)
   - `8` goes into bucket `2` (since `(8 - 2) // 3 = 2`)

   After bucketing, we update the buckets with the min and max values:
   - Buckets: `[[2, 4], [inf, -inf], [8, 8], [12, 12]]`
6. To find the maximum gap, we initialize `prev = 2` (the leftmost x-coordinate) and `ans = 0`. We iterate through the buckets:
   - Skip bucket `1` because it is empty (`[inf, -inf]`).
   - At bucket `2`, `curmin = 8`, `curmax = 8`. The gap is `8 - 4 = 4`. Therefore, `ans = 4` and `prev = 8`.
   - Skip bucket `3` because it is right next to the previous non-empty bucket.
7. After iterating through the buckets, `ans = 4` is the widest vertical gap we can have between two lines that are parallel to the y-axis without any points lying in that region.

This example aligns with the solution's logic, where bucketing greatly simplifies the problem and allows us to avoid sorting the entire list of x-coordinates while still finding the maximum gap efficiently.

Python Solution

```python
from typing import List
from math import inf

class Solution:
    def maxWidthOfVerticalArea(self, points: List[List[int]]) -> int:
        # Extract the x-coordinates from the points
        x_coords = [x for x, _ in points]

        # Get the number of points
        num_points = len(x_coords)

        # Find the minimum and maximum x-coordinates
        min_coord, max_coord = min(x_coords), max(x_coords)

        # Calculate the bucket size based on the range of x-coordinates and number of points
        bucket_size = max(1, (max_coord - min_coord) // (num_points - 1))

        # Determine the number of buckets required
        bucket_count = (max_coord - min_coord) // bucket_size + 1

        # Initialize buckets, each will hold the minimum and maximum x-coordinates for that bucket
        buckets = [[inf, -inf] for _ in range(bucket_count)]

        # Distribute x-coordinates into buckets
        for x in x_coords:
            index = (x - min_coord) // bucket_size
            buckets[index][0] = min(buckets[index][0], x)  # Update min for the bucket
            buckets[index][1] = max(buckets[index][1], x)  # Update max for the bucket

        # Variable to keep track of maximum width of vertical area between points
        max_width = 0

        # Variable to keep track of the previous bucket's max x-coordinate
        prev_max = inf

        # Calculate the maximum width by considering the gaps between the buckets
        for bucket_min, bucket_max in buckets:
            # Ignore empty buckets
            if bucket_min > bucket_max:
                continue
            # Update maximum width if current gap is larger than what we have seen before
            max_width = max(max_width, bucket_min - prev_max)
            # Update previous bucket's max for the next iteration
            prev_max = bucket_max

        # Return the maximum width of vertical area found
        return max_width

# Example usage:
# sol = Solution()
# print(sol.maxWidthOfVerticalArea([[8,7], [9,9], [7,4], [9,7]]))  # Should output 1
```

Java Solution

```java
class Solution {
    public int maxWidthOfVerticalArea(int[][] points) {
        int numPoints = points.length;     // The total number of points provided
        int[] xCoords = new int[numPoints];   // Array to hold the x-coordinates of the points

        // Extract the x-coordinates from each point and store them in xCoords
        for (int i = 0; i < numPoints; ++i) {
            xCoords[i] = points[i][0];
        }

        // Define infinity as a large number
        final int infinity = 1 << 30;
        // Initialize variables to store the minimum and maximum x-coordinates
        int minX = infinity, maxX = -infinity;

        // Find the minimum and maximum x-coordinates among all points
        for (int x : xCoords) {
            minX = Math.min(minX, x);
            maxX = Math.max(maxX, x);
        }

        // Calculate the size of each bucket for the bucket sort algorithm
        int bucketSize = Math.max(1, (maxX - minX) / (numPoints - 1));
        int bucketCount = (maxX - minX) / bucketSize + 1;  // The number of buckets

        // Create a 2D array to store the minimum and maximum values in each bucket
        int[][] buckets = new int[bucketCount][2];
        for (int[] bucket : buckets) {
            bucket[0] = infinity;
            bucket[1] = -infinity;
        }

        // Place each x-coordinate into the correct bucket
        for (int x : xCoords) {
            int index = (x - minX) / bucketSize;
            buckets[index][0] = Math.min(buckets[index][0], x);
            buckets[index][1] = Math.max(buckets[index][1], x);
        }

        // Initialize the variable for the previous maximum x-coordinate
        int prevMaxX = infinity;
        // Initialize the answer to zero, which represents the maximum width found
        int maxGap = 0;

        // Iterate through each bucket and calculate the gap between adjacent non-empty buckets
        for (int[] bucket : buckets) {
            if (bucket[0] > bucket[1]) {  // Skip the empty bucket
                continue;
            }
            // Update the maximum gap found so far
            maxGap = Math.max(maxGap, bucket[0] - prevMaxX);
            // Update the previous maximum x-coordinate
            prevMaxX = bucket[1];
        }
        // Return the maximum gap (maximum width of vertical area)
        return maxGap;
    }
}
```

C++ Solution

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

using namespace std;

class Solution {
public:
    int maxWidthOfVerticalArea(vector<vector<int>>& points) {
        int n = points.size();
        // Create a vector to hold the x-coordinates
        vector<int> xCoords;
        for (const auto& point : points) {
            xCoords.push_back(point[0]);
        }

        // Set initial maximum and minimum values for coordinates
        const int INF = 1 << 30;
        int minX = INF, maxX = -INF;

        // Find minimum and maximum x-coordinates
        for (int x : xCoords) {
            minX = min(minX, x);
            maxX = max(maxX, x);
        }

        // Calculate the size of each bucket
        int bucketSize = max(1, (maxX - minX) / (n - 1));
        int bucketCount = (maxX - minX) / bucketSize + 1;

        // Initialize buckets to hold minimum and maximum values
        vector<pair<int, int>> buckets(bucketCount, {INF, -INF});

        // Distribute x-coordinates into buckets
        for (int x : xCoords) {
            int index = (x - minX) / bucketSize;
            buckets[index].first = min(buckets[index].first, x);
            buckets[index].second = max(buckets[index].second, x);
        }

        int maxWidth = 0;
        int prevMax = INF;

        // Calculate the maximum width between adjacent vertical lines
        for (const auto& [bucketMin, bucketMax] : buckets) {
            if (bucketMin > bucketMax) continue; // Skip empty buckets
            maxWidth = max(maxWidth, bucketMin - prevMax);
            prevMax = bucketMax;
        }

        return maxWidth;
    }
};
```

Typescript Solution

```typescript
function maxWidthOfVerticalArea(points: number[][]): number {
    // Extract the x-coordinates from the points
    const xCoordinates: number[] = points.map(point => point[0]);
    const infinity = 1 << 30; // A large value to represent infinity
    const numPoints = xCoordinates.length;
    let minCoordinate = infinity; // Initialize minimum coordinate to infinity
    let maxCoordinate = -infinity; // Initialize maximum coordinate to negative infinity

    // Find the minimum and maximum x-coordinates
    for (const x of xCoordinates) {
        minCoordinate = Math.min(minCoordinate, x);
        maxCoordinate = Math.max(maxCoordinate, x);
    }

    // Calculate the size of each bucket
    const bucketSize = Math.max(1, Math.floor((maxCoordinate - minCoordinate) / (numPoints - 1)));
    const bucketCount = Math.floor((maxCoordinate - minCoordinate) / bucketSize) + 1;
    // Initialize buckets for the bucket sort
    const buckets = new Array(bucketCount).fill(0).map(() => [infinity, -infinity]);

    // Distribute the x-coordinates into buckets
    for (const x of xCoordinates) {
        const bucketIndex = Math.floor((x - minCoordinate) / bucketSize);
        buckets[bucketIndex][0] = Math.min(buckets[bucketIndex][0], x);
        buckets[bucketIndex][1] = Math.max(buckets[bucketIndex][1], x);
    }

    let previousMax = infinity; // Tracking the maximum value of the previous non-empty bucket
    let maxDistance = 0; // Initialize the maximum distance to 0

    // Iterate through the buckets to find the maximum distance between non-empty consecutive buckets
    for (const [leftBoundary, rightBoundary] of buckets) {
        if (leftBoundary > rightBoundary) {
            // This means the bucket is empty, move to the next one
            continue;
        }
        maxDistance = Math.max(maxDistance, leftBoundary - previousMax); // Update the maximum distance
        previousMax = rightBoundary; // Set the previous max to the right boundary of the current bucket
    }

    // Return the maximum distance found
    return maxDistance;
}
```

Time and Space Complexity

The given solution is implementing a form of bucket sort algorithm to find the maximum width of a vertical area between the given points. Let's analyze the time and space complexity:

Time Complexity

- **Sorting the X-coordinates**: The initial step of extracting the x-coordinates does not sort the values, it merely extracts the x-coordinates into a list, which would take `O(N)` time, where `N` is the number of points.
- **Finding Min and Max**: Finding the minimum and maximum x-coordinate takes `O(N)` since it iterates over all x-coordinates once.
- **Bucket Calculations**: The next several lines calculate the number of buckets and allocate them, which is `O(B)`, where `B` is the number of buckets. `B` is calculated based on the range divided by the bucket size, which is `(mx - mi) // (n - 1)`. Therefore, `B` is at most `N`, since the maximum amount of buckets would be when every unique x-coordinate gets its own bucket.
- **Buckets Filling**: Iterating over the x-coordinates again to fill the buckets also takes `O(N)`, doing constant-time operations to update the bucket min and max.
- **Determining the Maximum Width**: Finally, the last loop to determine the max width goes through each bucket, which takes `O(B)` time. However, the loop skips over the empty buckets effectively, so it processes only the buckets that have been filled. Since the number of meaningful buckets is at most `N - 1` (accounting for the case where all points have unique x-coordinates), the complexity of this step is also `O(N)`.

So the total time complexity is `O(N)` for all the steps since they all operate linearly with respect to the number of points.

Space Complexity

- **Storage for x-coordinates**: The list of x-coordinates takes `O(N)` space.
- **Buckets Array**: The buckets contribute an additional `O(B)` space complexity. As mentioned earlier, in the worst case, `B` can be `N` for unique x-coordinates.

Therefore, the total space complexity is also `O(N)`.

In conclusion, the time complexity is `O(N)` and the space complexity is `O(N)`.