1637. Widest Vertical Area Between Two Points Containing No Points Medium Array Sorting Leetcode Link



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 yaxis. 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

this is inefficient as the number of comparisons grows quickly with the number of points $(0(n^2)$ comparisons).

minimum x-values in each bucket are used to find the largest gap between consecutive non-empty buckets.

want to identify the largest gap. A brute force approach could involve comparing every pair of x-coordinates to calculate the gap, but

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 0(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

 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

In the provided solution, we seek to find the widest gap between the x-coordinates of the given points without any points lying in

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)

that can be solved quickly in linear time.

 $\max(1, 10 // (4 - 1)) = \max(1, 3) = 3.$

8 goes into bucket 2 (since (8 - 2) // 3 = 2)

Skip bucket 1 because it is empty ([inf, -inf]).

list of x-coordinates while still finding the maximum gap efficiently.

Extract the x-coordinates from the points

Determine the number of buckets required

Distribute x-coordinates into buckets

Ignore empty buckets

prev_max = bucket_max

continue

if bucket_min > bucket_max:

x_coords = [x for x, _ in points]

Get the number of points

num_points = len(x_coords)

for x in x_coords:

max_width = 0

def maxWidthOfVerticalArea(self, points: List[List[int]]) -> int:

bucket_count = (max_coord - min_coord) // bucket_size + 1

max_width = max(max_width, bucket_min - prev_max)

Update previous bucket's max for the next iteration

// Initialize the variable for the previous maximum x-coordinate

if (bucket[0] > bucket[1]) { // Skip the empty bucket

maxGap = Math.max(maxGap, bucket[0] - prevMaxX);

// Return the maximum gap (maximum width of vertical area)

// Update the previous maximum x-coordinate

int maxWidthOfVerticalArea(vector<vector<int>>& points) {

// Set initial maximum and minimum values for coordinates

// Create a vector to hold the x-coordinates

for (const auto& point : points) {

xCoords.push_back(point[0]);

// Find minimum and maximum x-coordinates

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});

// Update the maximum gap found so far

// Initialize the answer to zero, which represents the maximum width found

// Iterate through each bucket and calculate the gap between adjacent non-empty buckets

int prevMaxX = infinity;

continue;

for (int[] bucket : buckets) {

prevMaxX = bucket[1];

int maxGap = 0;

return maxGap;

buckets = [[inf, -inf] for _ in range(bucket_count)]

index = (x - min_coord) // bucket_size

axis without any points lying in that region.

Example Walkthrough

Solution Approach

that area. Here's how the solution is implemented:

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

 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.

After going through all buckets, ans holds the largest gap found, which is our final answer.

prev is then updated to curmax, handling the next iteration's calculations.

Buckets: [[inf, -inf], [inf, -inf], [inf, -inf], [inf, -inf]]

5. We iterate over nums and assign each x-coordinate to the correct bucket:

After bucketing, we update the buckets with the min and max values:

(x - mi) // bucket_size. Inside the bucket, we update the minimum or maximum value as necessary.

- 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.
- 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
- 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].

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

4. We initialize four buckets with [inf, -inf], because we have (mx - mi) // bucket_size + 1 = (10 // 3) + 1 = 4 buckets.

2. We calculate the range by finding the minimum and maximum values in nums, which are mi = 2 and mx = 12.

\circ 4 goes into bucket 0 (since (4 - 2) // 3 = 0)

Python Solution

class Solution:

8

9

10

11

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

38

39

40

41

42

43

44

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

C++ Solution

#include <vector>

class Solution {

public:

9

10

11

12

13

14

15

16

17

18

19

20

27

28

29

30

31

32

#include <algorithm>

using namespace std;

int n = points.size();

const int INF = 1 << 30;

int minX = INF, maxX = -INF;

minX = min(minX, x);

maxX = max(maxX, x);

vector<int> xCoords;

from typing import List

from math import inf

 2 goes into bucket 0 (since (2 - 2) // 3 = 0) \circ 12 goes into bucket 3 (since (12 - 2) // 3 = 3)

- 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:
- 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-

This example aligns with the solution's logic, where bucketing greatly simplifies the problem and allows us to avoid sorting the entire

12 # Find the minimum and maximum x-coordinates 13 min_coord, max_coord = min(x_coords), max(x_coords) 14 # Calculate the bucket size based on the range of x-coordinates and number of points 15 bucket size = max(1, (max coord - min coord) // (num points - 1)) 16

Initialize buckets, each will hold the minimum and maximum x-coordinates for that bucket

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

Update maximum width if current gap is larger than what we have seen before

Variable to keep track of maximum width of vertical area between points

33 # Variable to keep track of the previous bucket's max x-coordinate 34 prev_max = inf 35 36 # Calculate the maximum width by considering the gaps between the buckets for bucket_min, bucket_max in buckets: 37

```
45
             # Return the maximum width of vertical area found
 46
 47
             return max_width
 48
 49 # Example usage:
 50 # sol = Solution()
 51 # print(sol.maxWidthOfVerticalArea([[8,7], [9,9], [7,4], [9,7]])) # Should output 1
 52
Java Solution
    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) {</pre>
                 xCoords[i] = points[i][0];
  8
  9
 10
             // Define infinity as a large number
 11
 12
             final int infinity = 1 << 30;
 13
             // Initialize variables to store the minimum and maximum x-coordinates
 14
             int minX = infinity, maxX = -infinity;
 15
 16
             // Find the minimum and maximum x-coordinates among all points
             for (int x : xCoords) {
 17
 18
                 minX = Math.min(minX, x);
 19
                 maxX = Math.max(maxX, x);
 20
 21
 22
             // Calculate the size of each bucket for the bucket sort algorithm
 23
             int bucketSize = Math.max(1, (maxX - minX) / (numPoints - 1));
 24
             int bucketCount = (maxX - minX) / bucketSize + 1; // The number of buckets
 25
 26
             // Create a 2D array to store the minimum and maximum values in each bucket
 27
             int[][] buckets = new int[bucketCount][2];
 28
             for (int[] bucket : buckets) {
 29
                 bucket[0] = infinity;
 30
                 bucket[1] = -infinity;
 31
 32
 33
             // Place each x-coordinate into the correct bucket
             for (int x : xCoords) {
 34
 35
                 int index = (x - minX) / bucketSize;
 36
                 buckets[index][0] = Math.min(buckets[index][0], x);
 37
                 buckets[index][1] = Math.max(buckets[index][1], x);
 38
```

for (int x : xCoords) { 21 22 23 24 25 26 // Calculate the size of each bucket

```
33
             // Distribute x-coordinates into buckets
 34
             for (int x : xCoords) {
 35
                 int index = (x - minX) / bucketSize;
 36
                 buckets[index].first = min(buckets[index].first, x);
 37
                 buckets[index].second = max(buckets[index].second, x);
 38
 39
 40
             int maxWidth = 0;
             int prevMax = INF;
 41
 42
 43
             // Calculate the maximum width between adjacent vertical lines
             for (const auto& [bucketMin, bucketMax] : buckets) {
 44
 45
                 if (bucketMin > bucketMax) continue; // Skip empty buckets
                 maxWidth = max(maxWidth, bucketMin - prevMax);
 46
 47
                 prevMax = bucketMax;
 48
 49
 50
             return maxWidth;
 51
 52
    };
 53
Typescript Solution
    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</pre>
         const numPoints = xCoordinates.length;
  5
         let minCoordinate = infinity; // Initialize minimum coordinate to infinity
  6
         let maxCoordinate = -infinity; // Initialize maximum coordinate to negative infinity
  8
         // Find the minimum and maximum x-coordinates
  9
 10
         for (const x of xCoordinates) {
 11
             minCoordinate = Math.min(minCoordinate, x);
 12
             maxCoordinate = Math.max(maxCoordinate, x);
 13
 14
 15
         // Calculate the size of each bucket
         const bucketSize = Math.max(1, Math.floor((maxCoordinate - minCoordinate) / (numPoints - 1)));
 16
 17
         const bucketCount = Math.floor((maxCoordinate - minCoordinate) / bucketSize) + 1;
 18
         // Initialize buckets for the bucket sort
 19
         const buckets = new Array(bucketCount).fill(0).map(() => [infinity, -infinity]);
 20
 21
         // Distribute the x-coordinates into buckets
 22
         for (const x of xCoordinates) {
 23
             const bucketIndex = Math.floor((x - minCoordinate) / bucketSize);
 24
             buckets[bucketIndex][0] = Math.min(buckets[bucketIndex][0], x);
 25
             buckets[bucketIndex][1] = Math.max(buckets[bucketIndex][1], x);
 26
 27
 28
         let previousMax = infinity; // Tracking the maximum value of the previous non-empty bucket
 29
         let maxDistance = 0; // Initialize the maximum distance to 0
 30
 31
         // Iterate through the buckets to find the maximum distance between non-empty consecutive buckets
 32
         for (const [leftBoundary, rightBoundary] of buckets) {
 33
             if (leftBoundary > rightBoundary) {
 34
                 // This means the bucket is empty, move to the next one
 35
                 continue;
```

Time and Space Complexity

return maxDistance;

// Return the maximum distance found

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-

• 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

coordinates into a list, which would take O(N) time, where N is the number of points.

The given solution is implementing a form of bucket sort algorithm to find the maximum width of a vertical area between the given

maxDistance = Math.max(maxDistance, leftBoundary - previousMax); // Update the maximum distance

previousMax = rightBoundary; // Set the previous max to the right boundary of the current bucket

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.

36

37

38

39

40

41

42

44

43 }

- 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)
- 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.

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

Space Complexity Storage for x-coordinates: The list of x-coordinates takes 0(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.
- In conclusion, the time complexity is O(N) and the space complexity is O(N).

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