2250. Count Number of Rectangles Containing Each Point

Medium Binary Indexed Tree Array Binary Search Sorting

Problem Description

bottom-left corner fixed at the origin (0, 0) and the top-right corner at coordinates (length, height). The points are given with their x and y coordinates.

Your task is to determine for each point how many rectangles out of the given set contain it. A point is considered contained within a

The problem presents us with a set of rectangles and a set of points. Each rectangle is defined by its length and height, with the

Leetcode Link

Your task is to determine for each point how many rectangles out of the given set contain it. A point is considered contained within a rectangle if it lies inside the rectangle or on its edges, meaning the point's x-coordinate is less than or equal to the rectangle's length and its y-coordinate is less than or equal to the rectangle's height.

x and y coordinates, respectively.

Intuition

The intuition behind the solution involves preprocessing the rectangles to make the query for each point efficient. Since we only care

To sum it up: for every point given, count how many rectangles have both their length and height greater than or equal to the point's

about whether a point is contained within a rectangle, we can organize our rectangles by their heights. For each possible height, we keep a list of lengths of rectangles that have at least that height.

keep a list of lengths of rectangles that have at least that height.

Once we have this structure, we can quickly determine how many rectangles contain a point with a given y-value. For a point with coordinates (x, y), we need to count the number of rectangles that have a height >= y and a width >= x.

This is where binary search comes into play. Because each list of lengths for a given height is sorted, we can use binary search to

efficiently find the index of the smallest length that is still >= x. All lengths to the right of this index will also be >= x, so the number

of such lengths is the total length of the list minus the index.

By summing up the counts for each height from y to the maximum possible height (which in this case is set to 100, the assumed maximum height), we get the total number of rectangles that contain the point. We repeat this process for each point and return the

accumulated counts as our answer.

Solution Approach

The implementation of the solution can be divided into the following steps:

1. First, we initialize a dictionary (defaultdict) named d that will hold lists of rectangle lengths indexed by their heights.

2. We then iterate through each rectangle in the rectangles list, add the length of each rectangle to the corresponding list in d

4. We then initialize an empty list ans which will hold our final counts for each point.

indexed by its height.

5. Now we iterate through each given point in points. For the current point, we start a counter at 0. Then we check every possible

total number of lengths (len(xs) - bisect_left(xs, x)).

• Finally, we have the answer [3,1].

For the rectangle [1,2], we add 1 to the list at d[2].

For the rectangle [2,3], we add 2 to the list at d[3].

For the rectangle [2,5], we add 2 to the list at d[5].

Step 3: We initialize ans as an empty list to hold the count for each point.

rectangle height starting from the point's y-coordinate up to the maximum height (which is set to 101 to include the maximum height of 100).

3. After storing all the rectangles, we sort each list of lengths in d. This allows us to apply binary search on these lists.

6. For each height h that is equal to or greater than the y-coordinate of the point, we get the list of rectangle lengths xs from the dictionary d.

7. We perform a binary search using the bisect_left function from Python's bisect module to find the index where the point's x-

8. To find the number of lengths that are greater than or equal to the point's x-coordinate, we subtract the found index from the

coordinate could be inserted into the xs list to maintain sorted order. This index also represents the number of lengths that are less than the point's x-coordinate.

9. We add this number to our counter cnt and, after checking all the possible heights, append cnt to our ans list.

Suppose we have rectangles = [[1,2], [2,3], [2,5]] and points = [[1,1], [1,4]].
Our dictionary d after processing rectangles will look like: {2: [1], 3: [2], 5: [2]}

• After sorting, it will be {2: [1], 3: [2], 5: [2]} (no change in this case because there's only one length for each height).

• Now for point [1,1], we look at d[1] upwards: d[1] doesn't exist, but d[2] does, and since 1 can be inserted at index 0 in [1],

there is 1 - 0 = 1 rectangle that contains it; looking at d[3] and d[5], both have lists [2] which would include the point since

the bisect_left([2], 1) would give 0 resulting again in one rectangle each. Hence cnt = 1+1+1.

• The process is repeated for the second point [1,4], where now only d[5] can include the point, resulting in cnt = 1.

This solution utilizes a combination of hashmap to collate per-height rectangle lengths, sorting to prepare for binary search, and

Example Walkthrough

binary search to effectively answer queries for each point on how many rectangles contain it.

10. Finally, after calculating the counts for all the points, we return the list ans as our output.

Let's go through an example to better understand the algorithm using its data structures and patterns:

rectangles = [[1,2], [2,3], [2,5]]
 points = [[1,1], [1,4]]
 Step 1: The dictionary d is initialized. After processing rectangles, d is filled as follows:

Let's walk through a small example to illustrate the approach provided in the solution. We have a set of rectangles and a set of points

5 }

3: [2],

5: [2]

The dictionary now looks like this:

efficiently apply binary search later on.

defined as follows:

At height h=2 (since d[1] doesn't exist), we find the list [1]. Since 1 can be inserted at index 0 to maintain order, it means there is 1 - 0 = 1 rectangle of height at least 2 that contains the point.
 At height h=3, bisect_left([2], 1) is 0, so there is 1 - 0 = 1 rectangle of height at least 3 that contains the point.

• At height h=5, bisect_left([2], 1) is again 0, so there is another rectangle here. The accumulated count cnt becomes 1 +

At height h=5, bisect_left([2], 1) gives 0, so there is 1 - 0 = 1 rectangle of height at least 5 that contains the point. No

other heights are considered since they are less than the y-coordinate of the point. The final count for this point is 1, which

Step 2: As each list only contains one element, the lists are already sorted. If there were multiple elements, we would sort them to

Step 4-9: We iterate through the points [1,1] and [1,4] to count how many rectangles contain these points:

• For point [1,1], we start with a counter cnt set to 0. We look at possible heights starting from y=1:

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

Create a dictionary to store the x-coordinates grouped by their y-coordinates.

Initialize the list to hold the number of rectangles covering each point.

Check each possible y-coordinate (height) at or above the point's y.

// Create a list of array lists to store the x-coordinates indexed by y-coordinates

• For point [1,4], cnt is again set to 0. Since we only have heights up to 5, we look at:

Python Solution

Go through each rectangle and group x-coordinates by y.

Sort each list of x-coordinates for binary search efficiency.

For each point, count the number of rectangles that can cover it.

Append the count to results list for the current point.

List<Integer>[] coordinatesByHeight = new List[maxYCoordinate];

// Fill the list with x-coordinates for corresponding y-coordinates

Arrays.setAll(coordinatesByHeight, k -> new ArrayList<>());

coordinatesByHeight[rectangle[1]].add(rectangle[0]);

// Sort each list of x-coordinates for binary search

for (List<Integer> list : coordinatesByHeight) {

3 rectangles and the second point [1,4] is contained within 1 rectangle.

In this example, we see the practical use of organizing rectangle dimensions by height to efficiently query with binary search the

for height in range(point_y, 101):

Retrieve the list of x-coordinates at the current height.

x_coordinates_at_height = y_to_xs[height]

Use binary search to find the starting position where rectangles at this height can cover the point.

count += len(x_coordinates_at_height) - bisect_left(x_coordinates_at_height, point_x)

Step 10: The final output [3,1] is obtained after processing both points, which indicates that the first point [1,1] is contained within 3 rectangles and the second point [1,4] is contained within 1 rectangle.

count of rectangles that include each point.

1 from collections import defaultdict

y_to_xs = defaultdict(list)

y_to_xs[y].append(x)

y_coordinates.sort()

for point_x, point_y in points:

results.append(count)

int maxYCoordinate = 101;

// Initialize the list of array lists

for (int[] rectangle : rectangles) {

Collections.sort(list);

for y_coordinates in y_to_xs.values():

for x, y in rectangles:

from bisect import bisect_left

results = []

count = 0

return results

class Solution:

10

11

12

13

14

15

16

17

18

20

21

22

23

29

32

33

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

we append to ans.

1 + 1 = 3. We append this count to ans.

Java Solution

1 class Solution {
2 public int[] countRectangles(int[][] rectangles, int[][] points) {
3 // The maximum possible y-coordinate of the rectangles, assuming it is <= 100</pre>

```
22
             // The length of the array containing query points
 23
             int numPoints = points.length;
 24
 25
             // An array to store the counts of rectangles that contain each point
 26
             int[] ans = new int[numPoints];
 27
             // Iterate over each point to calculate the count of rectangles containing the point
 28
 29
             for (int i = 0; i < numPoints; ++i) {</pre>
                 int x = points[i][0], y = points[i][1];
 30
                 int count = 0;
 32
 33
                 // Evaluate for rectangles having a height greater than or equal to the y-coordinate of the point
 34
                 for (int h = y; h < maxYCoordinate; ++h) {</pre>
 35
 36
                     // Get the list of x-coordinates for the specific y-coordinate
 37
                     List<Integer> xCoordinates = coordinatesByHeight[h];
 38
 39
                     // Perform binary search to find the index where x-coordinate of rectangles
                     // is just greater than or equal to the x-coordinate of the point
 40
                     int left = 0, right = xCoordinates.size();
 41
 42
                     while (left < right) {</pre>
 43
                         int mid = (left + right) >> 1;
                         if (xCoordinates.get(mid) >= x) {
 44
 45
                             right = mid;
 46
                         } else {
 47
                             left = mid + 1;
 48
 49
 50
                     // Since the list is sorted, all x-coordinates from 'left' to the end will have
 51
                     // x-coordinates greater than or equal to the point's x-coordinate
 52
 53
                     count += xCoordinates.size() - left;
 54
 55
 56
                 // Store the count in the answer array
 57
                 ans[i] = count;
 58
 59
             return ans;
 60
 61 }
 62
C++ Solution
  #include <vector>
 2 #include <algorithm>
   class Solution {
  public:
       // Method to count the number of rectangles that can encompass each of the points
6
       std::vector<int> countRectangles(std::vector<std::vector<int>>& rectangles, std::vector<std::vector<int>>& points) {
```

// 'maxHeight' is a constant representing the maximum height we are considering

// Iterate through each point and count the number of rectangles that contain the point

// The number of valid rectangles is the total number of widths at this height

count += widthsAtHeight.end() - std::lower_bound(widthsAtHeight.begin(), widthsAtHeight.end(), x);

// minus the number of widths that are smaller than the point's x-coordinate

// Declare a constant for the maximum dimension (101 here is chosen based on problem constraints)

// Array 'heightToWidths' will store vectors of widths at each height index

std::vector<std::vector<int>> heightToWidths(maxHeight);

// Vector 'answers' will store the result for each point

auto& widthsAtHeight = heightToWidths[h];

// Return the answer vector containing counts for each point

function countRectangles(rectangles: number[][], points: number[][]): number[] {

// Initialize an array to store x-coordinates grouped by y-coordinate

// Sort each group of x-coordinates to enable binary search

for (let xCoordinates of xMapsByHeight) {

heightToWidths[rect[1]].push_back(rect[0]);

std::sort(widths.begin(), widths.end());

// Store widths in the corresponding height index and sort them

// For heights from the point's y-coordinate to maxHeight,

// count widths that are equal or larger than the point's x-coordinate

const int maxHeight = 101;

std::vector<int> answers;

int count = 0;

return answers;

const MAX_DIMENSION = 101;

Typescript Solution

for (auto& point : points) {

for (auto& rect : rectangles) {

for (auto& widths : heightToWidths) {

int x = point[0], y = point[1];

for (int h = y; h < maxHeight; ++h) {</pre>

// Add the count to the result vector

answers.push_back(count);

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

43

46

11

13

45 };

let xMapsByHeight = Array.from({ length: MAX_DIMENSION }, () => []); // Populate the xMapsByHeight with the x-coordinates of the rectangles for (let [x, y] of rectangles) { xMapsByHeight[y].push(x); }

```
xCoordinates.sort((a, b) => a - b);
14
15
16
17
       // Initialize an array to store the final counts for each point
       let counts = [];
18
19
       // Iterate through each point to count the number of rectangles that can contain the point
20
21
       for (let [px, py] of points) {
           let count = 0;
           // Check at or above the point's y-coordinate (as rectangles extend upwards)
24
           for (let height = py; height < MAX_DIMENSION; height++) {</pre>
               const xCoordinates = xMapsByHeight[height];
               let left = 0;
26
27
               let right = xCoordinates.length;
28
               // Perform binary search to find the number of rectangles with x greater than or equal to point's x
               while (left < right) {</pre>
29
                   let mid = (left + right) >> 1; // Equivalent to Math.floor((left + right) / 2)
30
                   if (px > xCoordinates[mid]) {
31
                       left = mid + 1;
33
                   } else {
                       right = mid;
34
35
36
               // Count the rectangles by subtracting the index of first rectangle not less than the point's x
37
38
               count += xCoordinates.length - right;
39
           // Add the count to the counts array
40
           counts.push(count);
42
43
       // Return the array of counts for each point
44
       return counts;
45
46 }
47
Time and Space Complexity
Time Complexity
The time complexity of the code consists of multiple parts:
 1. Building the dictionary d: We iterate over each rectangle in rectangles, resulting in O(n) operations, where n is the number of
    rectangles. At most, we are storing 100 keys (because height h could be from 1 to 100 according to the problem constraints).
 2. Sorting the lists in the dictionary: For each key in the dictionary, which represents a unique height h, we sort the list of
    associated x coordinates. The sort operation has a time complexity of O(k log k) for each list of length k. Since there could be
    up to n \times n values per height, and there are at most 100 heights, the worst-case complexity of this part is 0(100 \times n \log n).
```

each point (x, y), we have an inner loop where we iterate potentially from y to 100, let's denote this range as R = 100 - y + 1. Inside this loop, we perform a binary search using bisect_left which has a complexity of $0(\log k)$ for each height h. The total time complexity for processing points translates to 0(m*R*log n).

Combining these complexities, the overall time complexity of the function is $0(n + 100 * n \log n + m*R*\log n)$. Since R and the number 100 are constants, and in the context of big O notation we drop constants and lower-order terms, the dominant term is $0(n + 100 * n \log n)$.

approximates to $O(n \log n)$; otherwise, it will be closer to $O((n + m) \log n)$.

- Space Complexity

 The space complexity is determined by:
- Since h is a constant (at most 100), we can simply say 0(n).

 2. The output list ans: It stores the result for each point, so it takes 0(m) space.

Thus, the final space complexity of the function is 0(n + m), for storing all the rectangles and output results. In the contexts where n is much larger than m, it simplifies to 0(n).

1. The dictionary d storage: The dictionary itself will take O(h) space, with h being the unique heights, which is at most 100. Each

list in the dictionary can collectively hold up to O(n) coordinates. Therefore, the space complexity for the dictionary is O(n + n).

3. Processing the points: We iterate over every point in points, resulting in O(m) operations, where m is the number of points. For

log n + m*log n). When n (the number of rectangles) is large compared to m (the number of points), the time complexity