2345. Finding the Number of Visible Mountains

Medium | Array | Math | Sorting | Monotonic Stack | Leetcode Link

Problem Description

The problem provides us with an array of `peaks`, where each peak is represented by its coordinates `[x_i, y_i]`. These coordinates define the peak of a mountain which is shaped as a right-angled isosceles triangle, with its base along the x-axis. The sides of the mountain follow lines with gradients +1 and -1 respectively. This means that the mountains expand equally both up and down from the peak. A mountain is said to be visible if its peak is not obscured by any other mountain.

To determine if a mountain is visible, we must check that no other peak exists at a higher y-value for any given x-value that lies within the base range of the mountain we are examining. The objective is to count how many mountains remain visible when considering this criterion.

Intuition

To solve this problem, we start by transforming the peak coordinates into the form `(left, right)`, representing the leftmost and rightmost points of the mountain's base. This is done by subtracting and adding the `y` value to the `x` value of the peak respectively since we know the slopes are 1 and -1. By converting the coordinates this way, we create an interval for each mountain, and our task becomes akin to finding how many non-overlapping intervals there are.

Next, we sort the transformed array `arr` based on the leftmost point, and in case of a tie, the one with a farther rightmost point comes first. We initialize `cur` to negative infinity to represent the most right position we've covered so far.

As we iterate over sorted intervals, we can ignore any interval that ends before or exactly at `cur` since it means this mountain is completely hidden by another. If we find an interval where the rightmost point extends beyond `cur`, that means we've encountered a visible mountain.

However, there is a subtle detail. If we have duplicate intervals (`Counter(arr)` keeps track), it means two mountains have the same base and exactly overlap. In this case, we cannot count both as visible since one obscures the other, so we ensure we count a mountain as visible only if its interval `(left, right)` is unique, which is validated by the condition `cnt[(l, r)] == 1`.

With this approach, we can accurately count the number of visible mountains and return that as our answer.

Solution Approach

The solution approach involves several key steps, each utilizing fundamental algorithms and data structures:

1. **Transformation of coordinates**: We first convert the `peaks` array into an `arr` of `(left, right)` tuples, where `left` and `right` establish the boundaries of the base of the mountains. Essentially, this is just an application of the formula `(x - y, x + y)` for each peak `(x, y)`. This part of the code uses list comprehension for transformation:

```
1 arr = [(x - y, x + y) for x, y in peaks]
```

2. **Counting duplicates**: Before sorting, we count how many times each `(left, right)` tuple appears using `Counter` from the `collections` module. This is essential to identifying mountains that share a base and thus cannot both be visible:

```
1 cnt = Counter(arr)
```

3. **Sorting**: We then sort the `arr` by the `left` value of each tuple and in case of ties, by the `right` value, in descending order. The reason for sorting by `right` in descending order is to ensure that, in case of overlapping bases, the mountain with the wider base will be considered first. This makes use of a custom sorting function using the `key` parameter of the sort method:

```
1 arr.sort(key=lambda x: (x[0], -x[1]))
```

4. **Iterating and counting visible mountains**: To count the mountains, we iterate through the sorted `arr`. We keep track of the `cur` maximum `right` value seen so far, starting from negative infinity. If the `right` value of the current interval is less than or equal to `cur`, we skip this mountain as it is not visible. If not, we update `cur` to the `right` value of this mountain and increment the `ans` counter if the `(left, right)` tuple is unique (`cnt[(l, r)] == 1`). The use of a `for` loop along with conditional statements can be seen here:

```
1 ans, cur = 0, -inf
2 for l, r in arr:
3     if r <= cur:
4         continue
5     cur = r
6     if cnt[(l, r)] == 1:
7         ans += 1
```

5. **Returning the result**: After iterating through all the mountains, the `ans` variable holds the count of visible mountains, so we simply return that value:

```
1 return ans
```

This approach cleverly transforms the problem into an interval overlapping problem with unique intervals, which is then solved with sorting and line sweeping.

Example Walkthrough

Let's consider a small example with an array of peak coordinates: `[ (2, 3), (6, 1), (5, 2) ]`.

1. **Transformation of coordinates**: For each peak `(x, y)` we'll convert it to `(left, right)` by applying the formula `(x - y, x + y)`.
   - For the peak `(2, 3)`, the transformed coordinates are `(-1, 5)`.
   - For the peak `(6, 1)`, the transformed coordinates are `(5, 7)`.
   - For the peak `(5, 2)`, the transformed coordinates are `(3, 7)`.

   So now our `arr` looks like this: `[ (-1, 5), (5, 7), (3, 7) ]`

2. **Counting duplicates**: We count how many times each `(left, right)` tuple appears since overlapping bases can obscure each other. Using the `Counter` function we get:

```
1 cnt = { (-1, 5): 1, (5, 7): 1, (3, 7): 1 }
```

   No duplicates are found in this case.

3. **Sorting**: We sort `arr` by the `left` value and then by the `right` value in descending order. After sorting, our `arr` looks like this:

```
1 sorted arr = [ (-1, 5), (3, 7), (5, 7) ]
```

   The array is already sorted by the `left` values, and there are no ties to consider for the `right` values in descending order.

4. **Iterating and counting visible mountains**: We now count the visible mountains:
   - We start with `ans = 0` and `cur = -inf`.
   - The first interval `(-1, 5)` has `right` value `5`, which is greater than `cur`, so it is visible. We set `cur = 5` and since it's unique, increment `ans` to `1`.
   - The second interval `(3, 7)` has `right` value `7`, which is greater than `cur`, so it is visible. We set `cur = 7` and since it's unique, increment `ans` to `2`.
   - The third interval `(5, 7)` starts at `5`, but its `right` value is not greater than `cur`, which means it's not visible because it's fully within the range of the second mountain. We do not increment `ans`.
5. **Returning the result**: We went through all the mountains and concluded that 2 are visible. So, the final answer returned is `2`.

Python Solution

```
1  from collections import Counter
2  from math import inf
3
4  class Solution:
5      def visibleMountains(self, peaks: List[List[int]]) -> int:
6          # Convert each peak to a representation of its visibility range (left and right points)
7          visibility_ranges = [x - y, x + y] for x, y in peaks]
8
9          # Count how many times each visibility range occurs
10         counts = Counter(visibility_ranges)
11
12         # Sort the visibility ranges by the left point, and then by the right point in descending order
13         visibility_ranges.sort(key=lambda point: (point[0], -point[1]))
14
15         # Initialize the answer as 0 and the marker for the furthest right point seen so far as -infinity
16         visible_mountains_count, furthest_right = 0, -inf
17
18         # Loop through the sorted visibility ranges
19         for left, right in visibility_ranges:
20             # If the right point of the current range is not the furthest right point seen
21             # it means this mountain is obscured by another, so we can continue
22             if right <= furthest_right:
23                 continue
24
25             # Update the furthest right point seen so far to the current range's right point
26             furthest_right = right
27
28             # If the current range only has one occurrence, it means the mountain is visible
29             if counts[(left, right)] == 1:
30                 visible_mountains_count += 1
31
32         # Return the total count of visible mountains
33         return visible_mountains_count
34
```

Java Solution

```
1  class Solution {
2      public int visibleMountains(int[][] peaks) {
3          // Initialize the number of peaks
4          int numberOfPeaks = peaks.length;
5
6          // Transform the array to store left and right coordinates of peaks
7          int[][] transformedPeaks = new int[numberOfPeaks][2];
8
9          // Hash map to count occurrences of pairs of left and right coordinates
10         Map<String, Integer> countMap = new HashMap<>();
11
12         // Transform the peaks into their left and right coordinates and populate the count map
13         for (int i = 0; i < numberOfPeaks; ++i) {
14             int xCoordinate = peaks[i][0];
15             int yCoordinate = peaks[i][1];
16             transformedPeaks[i] = new int[] {xCoordinate - yCoordinate, xCoordinate + yCoordinate};
17             String key = (xCoordinate - yCoordinate) + "" + (xCoordinate + yCoordinate);
18             countMap.merge(key, 1, Integer::sum);
19         }
20
21         // Sort the transformed peaks array by left coordinate; if tied, sort by right coordinate descendently
22         Arrays.sort(transformedPeaks, (a, b) -> a[0] == b[0] ? b[1] - a[1] : a[0] - b[0]);
23
24         // Initialize the answer to 0 and current max right coordinate to minimum integer value
25         int visiblePeaksCount = 0;
26         int currentMaxRight = Integer.MIN_VALUE;
27
28         // Iterate over sorted peaks and count the number of visible peaks
29         for (int[] peak : transformedPeaks) {
30             int leftCoordinate = peak[0];
31             int rightCoordinate = peak[1];
32
33             // Skip peaks that are not beyond the current max right coordinate
34             if (rightCoordinate <= currentMaxRight) {
35                 continue;
36             }
37
38             // Update the current max right coordinate
39             currentMaxRight = rightCoordinate;
40
41             // Check if a peak is visible (unique left and right combination)
42             if (countMap.get(leftCoordinate + "" + rightCoordinate) == 1) {
43                 visiblePeaksCount++;
44             }
45         }
46
47         // Return the number of visible peaks
48         return visiblePeaksCount;
49     }
50 }
51
```

C++ Solution

```
1  class Solution {
2  public:
3      int visibleMountains(vector<vector<int>>& peaks) {
4          // Create a vector of pairs to hold transformed coordinates
5          vector<pair<int, int>> transformedPeaks;
6
7          // Transform the peaks coordinates and add them to the vector
8          for (auto& peak : peaks) {
9              int x = peak[0], y = peak[1];
10             // Transforming the peak coordinates into a line that represents the visible edge
11             transformedPeaks.emplace_back(x - y, -(x + y));
12         }
13
14         // Sort the transformed peaks based on their left coordinate, then by right coordinate in decreasing order
15         sort(transformedPeaks.begin(), transformedPeaks.end());
16
17         // Initialize the number of visible peaks and the current rightmost coordinate
18         int visibleCount = 0, currentRightmost = INT_MIN;
19
20         // Iterate over the transformed peaks to count the number of visible peaks
21         for (int i = 0; i < transformedPeaks.size(); ++i) {
22             int left = transformedPeaks[i].first;
23             int right = transformedPeaks[i].second;
24
25             // If the right coordinate is less than or equal to the current rightmost,
26             // the peak is obscured and we skip it
27             if (right <= currentRightmost) {
28                 continue;
29             }
30
31             // Update the current rightmost coordinate
32             currentRightmost = right;
33
34             // Increment visibleCount if it is the last peak or if the next peak is different
35             visibleCount += (i == transformedPeaks.size() - 1 || i < transformedPeaks.size() - 1 && transformedPeaks[i] != transformedPeaks[i + 1]);
36         }
37
38         // Return the count of visible peaks
39         return visibleCount;
40     }
41 };
42
```

Typescript Solution

```
1  // Define a type alias for a peak as a tuple of two numbers
2  type Peak = [number, number];
3
4  // Define a type alias for a transformed peak as a tuple of two numbers
5  type TransformedPeak = [number, number];
6
7  // Function to compute the number of visible mountains from the array of peaks
8  function visibleMountains(peaks: Peak[]): number {
9      // Create an array to hold transformed coordinates of peaks
10     const transformedPeaks: TransformedPeak[] = [];
11
12     // Transform the peak coordinates into lines representing visible edges
13     peaks.forEach(([x, y]) => {
14         transformedPeaks.push([x - y, -(x + y)]);
15     });
16
17     // Sort the transformed peaks based on their left coordinate, then by the right coordinate in decreasing order
18     transformedPeaks.sort((a, b) => {
19         const diff = a[0] - b[0];
20         return diff !== 0 ? diff : b[1] - a[1];
21     });
22
23     // Initialize the number of visible peaks and the current rightmost coordinate
24     let visibleCount: number = 0,
25         currentRightmost: number = -Infinity;
26
27     // Iterate over the transformed peaks to count the number of visible peaks
28     for (let i = 0; i < transformedPeaks.length; ++i) {
29         const [left, rightLimit] = transformedPeaks[i];
30         const right = -rightLimit;
31
32         // Skip the peak if it is obscured by the current rightmost peak
33         if (right <= currentRightmost) {
34             continue;
35         }
36
37         // Update the current rightmost coordinate to the rightmost point of the current peak
38         currentRightmost = right;
39
40         // Increment visibleCount if it is the last peak or if the next peak is different
41         const isLastPeakOrUnique = (i === transformedPeaks.length - 1) || (transformedPeaks[i + 1] && transformedPeaks[i][0] !== transformedPeaks[i + 1][0]);
42         if (isLastPeakOrUnique) {
43             visibleCount++;
44         }
45     }
46
47     // Return the total count of visible peaks
48     return visibleCount;
49 }
50
51 // Example usage:
52 // const peaks: Peak[] = [[2, 1], [4, 1], [3, 2]];
53 // const result: number = visibleMountains(peaks);
54 // console.log(result); // This will output the number of visible mountains
55
```

Time and Space Complexity

The provided Python code defines a function `visibleMountains` that computes the number of visible mountains given an array of peaks. Each peak is represented as a list with two integers. The time complexity and space complexity for each part of the algorithm are:

- Transforming peaks to `arr`, where a tuple `(x - y, x + y)` is created for each peak `(x, y)`: takes $O(n)$, where n is the length of the `peaks` array. The space complexity is also $O(n)$ due to the storage of the transformed `arr` list.
- Counting elements with `Counter(arr)`: this operation has a time complexity of $O(n)$ because it iterates once over the `arr`. The space complexity is $O(n)$ for storing the count of each unique tuple in `arr`.
- Sorting `arr` using `arr.sort(key=lambda x: (x[0], -x[1]))`: the time complexity for sorting in this case is $O(n \log n)$ because it uses the TimSort algorithm (Python's built-in sorting algorithm), which has this time complexity on average. Since sorting is done in place, the space complexity remains $O(1)$.
- Iterating over the sorted `arr` and computing the visible mountains: the time complexity is $O(n)$, as it requires a single pass through the array. The variable `cur` is used to keep track of the highest right point seen so far, and `ans` is used to count the number of unique, visible mountains. This part does not require additional space, so the space complexity is $O(1)$.

Overall, the time complexity of the entire function is dominated by the sorting part, which is $O(n \log n)$. The space complexity is determined by the additional data structures, primarily the transformation list and the counter, both of which are $O(n)$. Thus, the total space complexity is $O(n)$.