1944. Number of Visible People in a Queue

Hard Stack Array Monotonic Stack Leetcode Link

Problem Description

In this problem, we have `n` people standing in a queue, with each person having a *distinct* height. They are indexed from `0` to `n - 1`, starting from left to right. The goal is to determine for each person, how many other people in the queue they can see to their right.

The ability of one person to see another depends on the heights of others standing between them. Specifically, a person standing at position `i` can see a person at position `j` (where `j > i`) if everyone in between them is shorter than both person `i` and person `j`. In order words, the height of person `i` and person `j` must exceed the height of all individuals standing in positions `i+1` to `j-1`.

Our task is to return an array `answer`, where `answer[i]` is the number of people that person `i` can see to their right in the queue based on the described visibility condition.

Intuition

The key to solving this problem is to use a structure called a *monotonic stack*. This kind of stack preserves an order (either increasing or decreasing) as we process the elements.

Given that we want to find the number of people a person can see to the right before someone taller blocks the rest, the stack will help us keep track of potential candidates in decreasing height as we iterate from right to left through the queue.

Here's how we derive the solution:

1. We iterate from the end of the array `heights` to the beginning. This back-to-front iteration lets us approach the problem from the perspective of the person looking to the right, ensuring we've already handled all individuals potentially visible to them.
2. We use a stack to store the heights of people that are currently "visible" to the person at position `i`. Since we're looking for people that a person can see to the right, we keep removing shorter individuals from the stack until we find someone taller or the stack is empty.
3. As we remove these shorter individuals from the stack, we count them, as they contribute to the number of people the current person can see.
4. If there is someone taller on the stack after we've popped all shorter individuals, this means the current person can see one additional person (the taller one) who then blocks the view further. Therefore, we increment the count by one more in this case.
5. Finally, we have to add the current person's height to the stack because they may be visible to the next person we process (they are now the new "taller" individual that could potentially block others from view to the right).

By using the monotonic stack approach we avoid re-scanning parts of the array that we have already processed, hence significantly reducing the time complexity of the operation compared to checking all pairs of persons for visibility.

Solution Approach

The key to this problem is effectively tracking and incrementing our visibility count for each person as we iterate through the queue. Here's the step-by-step explanation of how the provided Python solution accomplishes this:

1. **Initialization**: We declare an empty list called `stk` meant to function as a stack. This stack will maintain indices of persons in a monotonically decreasing order according to their heights, meaning that each person's height on the stack is smaller than the one below them. We also create a list called `ans` of the same length as the `heights` array, initialized with zeroes. `ans[i]` will be used to store the number of people person `i` can see.
2. **Iterating Through The Queue**: The loop `for i in range(n - 1, -1, -1):` goes through each person starting from the end of the queue (`n - 1`) and moves towards the first person (`0`), backward one index at a time. This reverse iteration aligns with the direction of visibility (rightwards).
3. **Pop Shorter People Off The Stack**: Inside the loop, we have a *while* loop `while stk and stk[-1] < heights[i]:` which continues to pop elements from the stack if the top of the stack (the last element `stk[-1]`) is shorter than the current person's height (`heights[i]`). For each pop, we increment `ans[i]` because each of these popped persons is directly visible to the current person.
4. **Handle The Next Tallest Person**: After removing all shorter people, if the stack is not empty `if stk:`, it means that there is at least one person taller than the current person remaining on the stack. This taller person is the one who blocks the current person's view of anybody else to their right. Therefore, `ans[i]` is incremented by 1.
5. **Update The Stack**: Finally, we append the current person's height `heights[i]` to the stack. They will now be the potential "blocker" for people to their left, just as we are calculating for them with respect to those to their right.
6. **Return The Answer**: When the loop completes, we have filled `ans` with the visibility counts for all persons and we return this list.

Code Snippet

The algorithm is compactly implemented as follows:

```python
class Solution:
    def canSeePersonsCount(self, heights: List[int]) -> List[int]:
        n = len(heights)
        ans = [0] * n
        stk = []
        for i in range(n - 1, -1, -1):
            while stk and stk[-1] < heights[i]:
                ans[i] += 1
                stk.pop()
            if stk:
                ans[i] += 1
            stk.append(heights[i])
        return ans
```

This solution is efficient due to the monotonic stack which helps in keeping track of people that can potentially be seen without scanning the entire list for each person. As each person is processed, only a relevant subset of the people they can see is considered, thereby optimizing the visibility count update.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider a queue with 5 people having the following distinct heights: `[5, 3, 8, 3, 1]`. We'll go through the steps to figure out how many people each person can see to their right in the queue.

1. **Initialization**: Initialize an empty stack called `stk` and an answer array called `ans` with the same length as the `heights` array and all elements set to 0: `ans = [0, 0, 0, 0, 0]`.
2. **Iterating in reverse**: Start from the end of the queue with person 4 (height 1). Since the stack is empty, there is no one taller that blocks the view, so the stack becomes `stk = [1]`.
3. Move to person 3 (height 3). Since height 1 (on the stack) is less than height 3, we pop 1 from the stack and increment `ans[3]` by 1 (one person is visible). After popping, since the stack is empty, we don't increment `ans[3]` further and push height 3 onto the stack, so `stk = [3]`.
4. At person 2 (height 8), person 3 (on the stack, height 3) is shorter, so we pop it and increment `ans[2]`. The stack is now empty again, and we push height 8, making `stk = [8]`.
5. For person 1 with height 3, height 8 on the stack is taller, so we don't pop anyone from the stack, but increment `ans[1]` by 1 to indicate visibility of that one taller person blocking further view. We push height 3 onto the stack, so `stk = [8, 3]`.
6. Finally, for person 0 with height 5, since height 3 is shorter (on the stack), we pop it and increment `ans[0]`. Height 8 is taller, so we increment `ans[0]` again and don't pop it. We push height 5 onto the stack, resulting in `stk = [8, 5]`.
7. **Return the Answer**: The filled answer array represents the count of visible people for each person in the original queue order: `ans = [2, 1, 1, 1, 0]`.

To summarize, we processed each person from right to left, maintaining a stack that helped us quickly determine who each person can see by only considering those who are relevant (taller) for the visibility condition. The answer array now holds for each person the count of people they can see to their right.

Python Solution

```python
class Solution:
    def canSeePersonsCount(self, heights: List[int]) -> List[int]:
        # The number of people whose height can be seen by each person.
        number_of_people = len(heights)
        # Initialize an array to hold the count of people that can be seen for each person.
        visible_count = [0] * number_of_people
        # Initialize an empty list to use as a stack.
        stack = []

        # Iterate through the list of heights in reverse order.
        for index in range(number_of_people - 1, -1, -1):
            # While stack is not empty and the height at the top of the stack is less
            # than the current height, increment the count of visible people for
            # the current height and remove the top height from the stack.
            while stack and stack[-1] < heights[index]:
                visible_count[index] += 1
                stack.pop()

            # If the stack is not empty after the above while loop, it means the current person
            # can see at least one other person who is taller. Increment the count by 1.
            if stack:
                visible_count[index] += 1

            # Add the current height to the stack.
            stack.append(heights[index])

        # Return the list of counts for each person.
        return visible_count

```

Java Solution

```java
class Solution {
    public int[] canSeePersonsCount(int[] heights) {
        // Initialize the number of people in the array
        int n = heights.length;

        // Initialize the answer array where the result will be stored
        int[] answer = new int[n];

        // Initialize a stack that will keep track of the heights as we move backwards
        Deque<Integer> stack = new ArrayDeque<>();

        // Iterate over the heights array from end to start
        for (int i = n - 1; i >= 0; --i) {
            // Pop elements from the stack while the top element is less than the current height
            // because the current person can see over the shorter person(s) behind.
            while (!stack.isEmpty() && stack.peek() < heights[i]) {
                stack.pop();
                ++answer[i]; // Increment the count of people the current person can see
            }

            // If there's still someone in the stack, increment the count by one
            // because the current person can see at least one person who is taller.
            if (!stack.isEmpty()) {
                ++answer[i];
            }

            // Push the current height onto the stack
            stack.push(heights[i]);
        }

        // Return the populated answer array
        return answer;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <stack>
using namespace std;

class Solution {
public:
    vector<int> canSeePersonsCount(vector<int>& heights) {
        int count = heights.size();  // Get the number of people in the line
        vector<int> visibleCounts(count);  // Initialize a vector to store the counts of visible people
        stack<int> heightStack;  // Stack to keep track of the heights of people

        // Start from the end of the line to calculate visible counts for each person
        for (int i = count - 1; i >= 0; --i) {
            // Pop people from stack who are shorter than the current person
            // as current person can see over them
            while (!heightStack.empty() && heightStack.top() < heights[i]) {
                ++visibleCounts[i];  // Increment count of people this person can see
                heightStack.pop();  // Remove the shorter person from consideration
            }

            // If the stack is not empty after popping shorter people, increment count by 1
            // This means the current person can see the next taller person
            if (!heightStack.empty()) {
                ++visibleCounts[i];
            }

            // Push the current person's height into the stack
            heightStack.push(heights[i]);
        }

        return visibleCounts;  // Return the answer with counts of how many people each person can see
    }
};

```

Typescript Solution

```typescript
function canSeePersonsCount(heights: number[]): number[] {
    const numHeights = heights.length;
    const answer = new Array(numHeights).fill(0);
    const stack: number[] = [];

    // Iterate from the end of the heights array
    for (let i = numHeights - 1; i >= 0; --i) {
        // While there is a height in the stack smaller than the current height,
        // pop it and increment the count for the current position
        while (stack.length && stack[stack.length - 1] < heights[i]) {
            answer[i]++;
            stack.pop();
        }

        // If the stack is not empty after popping smaller elements, it means there
        // is at least one higher person that the current person can see, so increment
        // the count for the current position
        if (stack.length) {
            answer[i]++;
        }

        // Push the current height onto the stack
        stack.push(heights[i]);
    }

    return answer;
}

```

Time and Space Complexity

Time Complexity

The code iterates through the list of heights in reverse using a loop, resulting in `O(n)` where `n` is the length of the `heights` list. However, for each element, it potentially performs multiple comparisons and pop operations on the stack until a higher height is found or the stack is empty. In the worst-case scenario, each element will be pushed to and popped from the stack once, leading to an amortized time complexity of `O(n)`. Thus, the overall time complexity is `O(n)`.

Space Complexity

The space complexity is determined by the additional space used by the stack and the `ans` list. The `ans` list is the same size as the `heights` list, resulting in `O(n)` space complexity. The stack can also grow up to `n` in size in the worst case when heights are in ascending order, which also contributes `O(n)` space usage. The combined space complexity taking into account both the `ans` list and the stack is `O(n)`.