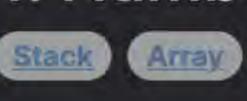
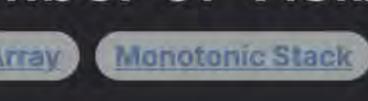
# Hard



Problem Description



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

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,

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
- 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).

additional person (the taller one) who then blocks the view further. Therefore, we increment the count by one more in this case.

reducing the time complexity of the operation compared to checking all pairs of persons for visibility. Solution Approach

By using the monotonic stack approach we avoid re-scanning parts of the array that we have already processed, hence significantly

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 [1] will be

direction of visibility (rightwards).

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

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

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.

person's view of anybody else to their right. Therefore, ans [i] is incremented by 1.

3. Pop Shorter People Off The Stack: Inside the loop, we have a while loop while stk and stk[-1] < heights[i]: which

- 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
- class Solution: def canSeePersonsCount(self, heights: List[int]) -> List[int]: n = len(heights) ans = [0] \* n

#### if stk: 10 11 stk.append(heights[i]) 12

13

stk = []

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.

blocks the view, so the stack becomes stk = [1].

again, and we push height 8, making stk = [8].

def canSeePersonsCount(self, heights: List[int]) -> List[int]:

# Iterate through the list of heights in reverse order.

for index in range(number\_of\_people - 1, -1, -1):

while stack and stack[-1] < heights[index]:</pre>

visible\_count[index] += 1

visible\_count[index] += 1

stack.append(heights[index])

return visible\_count

# Add the current height to the stack.

# Return the list of counts for each person.

number\_of\_people = len(heights)

# The number of people whose height can be seen by each person.

# 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.

The algorithm is compactly implemented as follows:

for i in range(n - 1, -1, -1):

ans[i] += 1

ans[i] += 1

stk.pop()

while stk and stk[-1] < heights[i]:

```
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
```

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].

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

7. Return the Answer: The filled answer array represents the count of visible people for each person in the original queue order:

we increment ans [0] again and don't pop it. We push height 5 onto the stack, resulting in stk = [8, 5].

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

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

- 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.
  - # 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 = []

```
stack.pop()
17
18
               # If the stack is not empty after the above while loop, it means the current person
19
20
               # can see at least one other person who is taller. Increment the count by 1.
21
               if stack:
```

ans = [2, 1, 1, 1, 0].

**Python Solution** 

class Solution:

8

9

10

11

12

13

14

15

16

22

23

24

25

26

27

28

29

```
Java Solution
   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];
 8
9
           // Initialize a stack that will keep track of the heights as we move backwards
10
           Deque<Integer> stack = new ArrayDeque<>();
11
           // Iterate over the heights array from end to start
12
13
           for (int i = n - 1; i >= 0; --i) {
14
               // Pop elements from the stack while the top element is less than the current height
15
               // because the current person can see over the shorter person(s) behind.
               while (!stack.isEmpty() && stack.peek() < heights[i]) {</pre>
16
17
                    stack.pop();
                    ++answer[i]; // Increment the count of people the current person can see
18
19
20
21
               // If there's still someone in the stack, increment the count by one
22
               // because the current person can see at least one person who is taller.
23
               if (!stack.isEmpty()) {
24
                   ++answer[i];
25
26
27
               // Push the current height onto the stack
28
               stack.push(heights[i]);
29
30
31
           // Return the populated answer array
32
           return answer;
33
```

## stack<int> heightStack; // Stack to keep track of the heights of people 10 12 13

C++ Solution

1 #include <vector>

class Solution {

public:

using namespace std;

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

2 #include <stack>

34 }

35

```
// Start from the end of the line to calculate visible counts for each person
           for (int i = count - 1; i >= 0; --i) {
14
               // Pop people from stack who are shorter than the current person
15
               // as current person can see over them
               while (!heightStack.empty() && heightStack.top() < heights[i]) {</pre>
16
                    ++visibleCounts[i]; // Increment count of people this person can see
17
                    heightStack.pop(); // Remove the shorter person from consideration
18
19
20
21
               // 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
22
               if (!heightStack.empty()) {
23
                   ++visibleCounts[i];
24
25
26
27
               // Push the current person's height into the stack
28
               heightStack.push(heights[i]);
29
30
31
            return visibleCounts; // Return the answer with counts of how many people each person can see
32
33 };
34
Typescript Solution
    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
 9
           while (stack.length && stack[stack.length - 1] < heights[i]) {</pre>
10
```

### 20 21 22 // Push the current height onto the stack 23 stack.push(heights[i]);

return answer;

answer[i]++;

stack.pop();

if (stack.length) {

answer[i]++;

Time and Space Complexity

### 13 14 15 // If the stack is not empty after popping smaller elements, it means there 16 // is at least one higher person that the current person can see, so increment // the count for the current position

12

19

24

25

26

28

27 }

**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).