1762. Buildings With an Ocean View

Medium | Stack | Array | Monotonic Stack

Leetcode Link

Problem Description

In this problem, we are given an array `heights` that represents the heights of buildings in a line, with the ocean situated to the right of the last building. The goal is to find out which buildings have an unobstructed view of the ocean. A building has an ocean view if all buildings to its right are shorter than it. We need to return a list of the indices of the buildings that can see the ocean. The list of indices should be sorted in increasing order and needs to be 0-indexed, which means the first building in the line has an index of 0.

Intuition

The solution involves iterating through the list of building heights in reverse (starting from the building farthest from the ocean). As we iterate, we maintain a running maximum height (`mx`). When the current building's height exceeds this running maximum, it means this building has an ocean view, because there are no taller buildings to its right blocking its view of the ocean.

The steps for the solution are as follows:

- Initialize an empty list `ans` to store indices of buildings with ocean views.
- Set a variable `mx` to 0 to keep track of the maximum height seen so far as we iterate backwards.
- Loop through the heights array in reverse order (from last to first):
  - Check if the current building's height is greater than `mx`, which is the tallest building observed to the right side of the current building.
  - If it is, the current building has an ocean view. We then:
    - Append the index `i` of this building to the `ans` list.
    - Update the `mx` to the new maximum height.
- Since we filled `ans` in reverse order, we reverse it once more before returning to give the correct order of indices (from left to right).

By iterating from the end towards the beginning, we ensure that the `mx` reflects the tallest building's height to the right of the current building. This way, each building is checked against only the relevant taller buildings that could potentially block its ocean view.

Solution Approach

The solution approach can be broken down into the following steps:

- We initialize an empty list `ans` that will eventually contain the indices of the buildings with an ocean view.
- Define a variable `mx` to keep track of the maximum height encountered as we iterate through the list of buildings in reverse. This is initially set to `0`.
- We start a loop that iterates through the `heights` array from the last element (furthest from the ocean) towards the first element (closest to the ocean).
- During each step of the loop, we compare the current building's height to `mx`. If the current height is greater, it means that this building has an unobstructed ocean view since all buildings to its right are shorter.
- If the building has an ocean view, we append its index to the `ans` list and update `mx` to the current building's height, ensuring that `mx` always represents the tallest building encountered so far.
- After iterating through all buildings, we reverse the `ans` list since we appended indices starting from the building closest to the ocean to get them sorted in the correct (increasing) order. The reasoning for this post-reversal is that indices are collected in decreasing order during the iteration from last to first.

This approach takes advantage of a simple **greedy algorithm** pattern, which, in this case, means that we always update our 'greedy' choice (`mx`) to reflect the tallest encountered building so far. The `mx` effectively "forgets" shorter buildings since they don't impact future comparisons.

In terms of **data structures**, only a simple list is used to keep track of the indices of buildings with an ocean view. The use of a list also allows us to efficiently reverse its contents at the end of our iteration.

Algorithm Complexity: The time complexity of this solution is `O(n)`, where `n` is the number of buildings. This is because we have to iterate through all the buildings at least once. The space complexity is also `O(n)` in the worst case, which occurs when all buildings have an ocean view, and we need a list of the same length as the input to store the indices.

The solution code utilizing this approach looks like this:

```python
class Solution:
    def findBuildings(self, heights: List[int]) -> List[int]:
        ans = []
        mx = 0
        for i in range(len(heights) - 1, -1, -1):
            if heights[i] > mx:
                ans.append(i)
                mx = heights[i]
        return ans[::-1]
```

This code translates the approach described above into Python, taking advantage of the language's list and loop constructs to efficiently solve the problem.

Example Walkthrough

Let's apply the solution approach to a small example. Consider the following array of building heights:

```python
heights = [4, 2, 3, 1]
```

We want to find out which buildings have an unobstructed view of the ocean. Our ocean is to the right, so we need to see which buildings are not blocked by buildings of greater height to their right.

We start by initializing an empty list `ans` to store indices and a variable `mx` with a value of 0 to keep track of the maximum height we've seen so far.

Then we iterate backwards through the array:

1. We start with the last building (index 3), height is 1. `mx` is 0. Since 1 > 0, we add index 3 to `ans` and update `mx` to 1.
2. Move to the next building (index 2), height is 3. Now, 3 > 1 (current `mx`), so we add this index (2) to `ans` and update `mx` to 3.
3. The next building's height (index 1) is 2. Because 2 is not greater than 3 (current `mx`), we don't do anything.
4. We now reach the first building (index 0), height is 4. Since 4 > 3, we add index 0 to `ans` and update `mx` to 4.

After iterating through the array, `ans` is `[0, 2, 3]`. However, this is in reverse order, so we reverse it to `[3, 2, 0]`.

Buildings at indices 3, 2, and 0 have an unobstructed view of the ocean. The result is sorted in increasing order, as required by the problem statement.

Python Solution

```python
class Solution:
    def findBuildings(self, heights: List[int]) -> List[int]:
        # Initialize an empty list to hold the indices of buildings with an ocean view
        ocean_view_indices = []

        # Initialize the maximum height found so far to 0
        max_height = 0

        # Iterate from the last building back to the first
        for i in range(len(heights) - 1, -1, -1):
            # Compare the current building's height with the max height found so far
            if heights[i] > max_height:
                # If the current building is taller, it has an ocean view
                # So we add its index to our list
                ocean_view_indices.append(i)
                # Update the max_height to the current building's height
                max_height = heights[i]

        # The resulting list is in reverse order, so we reverse it before returning
        return ocean_view_indices[::-1]

```

Java Solution

```java
class Solution {
    // This method finds the buildings that have an ocean view, given the heights of the buildings.
    // A building has an ocean view if all buildings to its right have a smaller height.
    public int[] findBuildings(int[] heights) {
        // Number of buildings
        int numberOfBuildings = heights.length;

        // List to store indices of buildings with an ocean view
        List<Integer> buildingsWithView = new ArrayList<>();

        // Keep track of the maximum height seen so far from the right
        int maxRightHeight = 0;

        // Iterate buildings from right to left to check for ocean view
        for (int i = numberOfBuildings - 1; i >= 0; --i) {
            // If the current building is taller than the max height seen so far to its right,
            // it has an ocean view.
            if (heights[i] > maxRightHeight) {
                // Add the current building's index to the list
                buildingsWithView.add(i);
                // Update the max height seen so far to the height of the current building
                maxRightHeight = heights[i];
            }
        }

        // Since we traversed from right to left, reverse the list to maintain the original order
        Collections.reverse(buildingsWithView);

        // Convert the List<Integer> to int[] for the final answer
        return buildingsWithView.stream().mapToInt(Integer::intValue).toArray();
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to find the buildings that can see the ocean
    vector<int> findBuildings(vector<int>& heights) {
        // The answer vector to store indices of the ocean-view buildings
        vector<int> oceanViewBuildings;
        // Variable to keep track of the maximum height observed as we scan from right to left
        int max_height_so_far = 0;

        // Iterate over the input 'heights' vector from right to left
        for (int i = heights.size() - 1; i >= 0; --i) {
            // If the current building's height is greater than the maximum height observed so far
            if (heights[i] > max_height_so_far) {
                // Add the index of this building to the ocean-view list
                oceanViewBuildings.push_back(i);
                // Update the maximum height to the height of the current building
                max_height_so_far = heights[i];
            }
        }

        // Since we added the indices in reverse order, we need to reverse the oceanViewBuildings vector
        reverse(oceanViewBuildings.begin(), oceanViewBuildings.end());

        // Return the final list of ocean-view building indices
        return oceanViewBuildings;
    }
};

```

Typescript Solution

```typescript
function findBuildings(heights: number[]): number[] {
    // Create an array to store the indices of the buildings with an ocean view.
    const buildingsWithViews: number[] = [];
    // Initialize a variable to keep track of the maximum height found so far as we iterate from right to left.
    let maxHeight = 0;

    // Start iterating from the last building towards the first.
    for (let i = heights.length - 1; i >= 0; --i) {
        // Check if the current building height is greater than the maximum height found.
        if (heights[i] > maxHeight) {
            // If so, add the index of this building to our result array.
            buildingsWithViews.push(i);
            // Update maxHeight to the height of the current building.
            maxHeight = heights[i];
        }
    }

    // Since we traversed the buildings from right to left, the resulting array is in reverse order.
    // Reverse the array to return the indices in the correct order, from left to right.
    return buildingsWithViews.reverse();
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be determined by analyzing the loop that iterates in reverse through the list of `heights`. Since the loop runs exactly once for each element in the `heights` array, which has `n` elements, this gives us a time complexity of `O(n)` where `n` is the length of the `heights` array.

Space Complexity

The space complexity of the code is mainly due to the `ans` list which stores the indices of buildings with an unobstructed view. In the worst case, all buildings can see the ocean, so the space taken by the `ans` list can be `n` in the worst case. Thus, the space complexity is `O(n)`. Also, note that this does not account for the space used to store the input `heights` as that is not part of the space used by the algorithm itself.