1063. Number of Valid Subarrays

Hard Biweek Array Monotonic Stack

Leetcode Link

Problem Description

The problem asks us to count the number of subarrays within a given array `nums`, where each subarray must adhere to the condition that its leftmost element is not larger than any of the other elements within the same subarray. A subarray is defined as a contiguous segment of an array.

Essentially, starting at any element within `nums`, we can form a valid subarray by extending it to the right as long as we encounter elements that are greater than or equal to this starting element. The count of all such possible subarrays is the required answer.

Intuition

To solve this problem efficiently, we can use a stack to keep track of elements and their indices. The core idea is to iterate through the array from right to left, adding elements to the stack as we go. For each element, we consider it as the potential leftmost element of a subarray and try to find out how many subarrays can be formed with this condition.

The stack will maintain the indices of elements in decreasing order leftward from the current element. The constraints of the problem ensure that for each element, any valid subarray can only extend as far as the element immediately preceding a smaller element in the array. This is because the leftmost element must be smaller than or equal to all other elements in the subarray.

During each iteration, we remove elements from the stack that are smaller than the current element because they cannot be the end of a valid subarray starting with the current element; their presence in the stack would violate our condition.

The difference in indices between the current element's index and the index of the element on top of the stack after popping (or the array's length if the stack is empty) gives us the number of valid subarrays with the current element as the leftmost item. This way, we can calculate the total number of valid subarrays efficiently without having to explicitly enumerate them.

Solution Approach

The provided Python solution utilizes a Monotonic Stack, which is a stack where the elements are in a single non-increasing or non-decreasing order. This pattern is particularly useful for problems involving subarray intervals and next greater elements.

Here's a step-by-step explanation of the approach:

1. **Initialize Variables**: We start by initializing a list called `stk` (which will serve as our stack), and a variable `ans` which will accumulate the count of valid subarrays.
2. **Iterate Through the Array**: We iterate through the array `nums` in reverse (from the last element to the first). This is done to treat each element as the potential starting point of several subarrays.
3. **Maintain Monotonic Stack**:
   - While there are elements in the stack and the element at the index specified by the top of the stack is greater than or equal to the current element `nums[i]`, we pop elements from the stack.
   - This ensures that the stack will always be in non-decreasing order and stores indices of elements that comply with our requirement of valid subarray start points.
4. **Calculate Subarray Count for Current Element**:
   - After popping all of the elements greater than the current element, the difference between the index on the top of the stack (if not empty) or the length of the array `n` (if the stack is empty) and the current index `i` gives the number of subarrays that can be formed with the current element as the leftmost one. We add this number to our `ans`.
5. **Push the Current Index onto the Stack**: The current index `i` is then added to the stack since it could be the start of a valid subarray with respect to the future elements as we continue iterating backward.
6. **Return the Result**: Once the iteration is complete, the total count of valid subarrays accumulated in `ans` is returned.

The stack is used here as a clever way to remember which elements have not yet found a smaller preceding element, and therefore from which points valid subarrays can still be extended. The stack maintenance ensures the next smaller element's index is always available for the calculations, minimizing the operations needed to get the subarray counts.

Using this approach, we can run through the array a single time with our operations on the stack being constant time on average due to the properties of a stack. This allows the algorithm to achieve a time complexity of O(n), where n is the length of the array.

Example Walkthrough

Let's say we have an example array:

```
nums = [3, 1, 2, 4]
```

We want to count the subarrays where the leftmost element is not larger than the other elements in the subarray.

Here's how the solution approach can be applied to this example:

1. **Initialize Variables**:
   - `stk = []` (our stack)
   - `ans = 0` (answer to accumulate subarray counts)
2. **Iterate Through the Array**:
   - We will iterate from right to left, which means we'll look at the elements in the following order: 4, 2, 1, 3.
3. **Maintain Monotonic Stack**:
   - When we start at index 3 (`nums[3] = 4`), the stack is empty so we proceed.
   - At index 2 (`nums[2] = 2`), the stack contains index 3 and `nums[3] >= nums[2]`. No popping required.
   - At index 1 (`nums[1] = 1`), the stack contains indices 3 and 2. `nums[3]` and `nums[2]` are both greater than `nums[1]`, so we pop both from the stack.
   - At index 0 (`nums[0] = 3`), the stack is empty as both prev elements were smaller.
4. **Calculate Subarray Count for Current Element**:
   - For index 3, stack is empty so `n - i + 4 - 3 = 1` subarray possible which is `[4]`.
   - For index 2, after stack operation top of stack is index 3, `3 - 2 = 1` subarray is possible which is `[2, 4]`.
   - For index 1, stack is empty after popping, `n - i + 4 - 1 = 3` subarrays possible: `[1, 2, 4]`, `[1, 2]`, and `[1]`.
   - For index 0, stack is empty so `n - i + 4 - 0 = 4` subarrays are possible: `[3, 1, 2, 4]`, `[3, 1, 2]`, `[3, 1]`, and `[3]`.
   - Add all the counts: `ans = 1 + 1 + 3 + 4 = 9`.
5. **Push the Current Index onto the Stack**:
   - After processing each element, we push its index onto the stack.
   - End of index 3: `stk = [3]`
   - End of index 2: `stk = [3, 2]`
   - End of index 1: `stk = [3]`
   - End of index 0: `stk = [0]`
6. **Return the Result**:
   - By the end of the iteration, `ans = 9`, which means there are 9 subarrays in `nums` where the leftmost element is not larger than the other elements.

Thus, we have found the number of subarrays that adhere to the given condition for the array `[3, 1, 2, 4]` using the solution approach described, with a final result of `9`.

Python Solution

```python
# Import the typing module to specify the type of the List
from typing import List

class Solution:
    def validSubarrays(self, nums: List[int]) -> int:
        # Get the length of the input list
        n = len(nums)

        # Initialize a stack to keep track of indices for valid subarrays
        stack = []

        # Initialize the counter for valid subarrays
        valid_subarray_count = 0

        # Iterate over the input list in reverse order
        for i in range(n - 1, -1, -1):
            # Pop elements from the stack while the current element is less than
            # the element at the index on top of the stack, which ensures that
            # we only count valid subarrays (non-decreasing order)
            while stack and nums[stack[-1]] >= nums[i]:
                stack.pop()

            # Calculate the number of valid subarrays including the current element.
            # If the stack is not empty, subtract the current index from the index on
            # top of the stack. If the stack is empty, it means the current element
            # can form valid subarrays with all subsequent elements.
            valid_subarray_count += (stack[-1] if stack else n) - i

            # Push the current index onto the stack
            stack.append(i)

        # Return the total count of valid subarrays
        return valid_subarray_count
```

Java Solution

```java
class Solution {
    public int validSubarrays(int[] nums) {
        // Get the total number of elements in the array.
        int totalElements = nums.length;
        // Initialize a stack that will help keep track of the indices.
        Deque<Integer> indexStack = new ArrayDeque<>();
        // This variable will store the count of valid subarrays.
        int validSubarrayCount = 0;

        // Iterate over the elements starting from the end of the array.
        for (int i = totalElements - 1; i >= 0; --i) {
            // If the stack is not empty and the last element in the stack
            // is greater than or equal to the current element, pop the stack.
            while (!indexStack.isEmpty() && nums[indexStack.peek()] >= nums[i]) {
                indexStack.pop();
            }

            // Calculate the number of valid subarrays ending with nums[i].
            // If the stack is empty, we can form subarrays with all the following elements.
            // If the stack is not empty, we form subarrays up to the last smaller element found.
            validSubarrayCount += (indexStack.isEmpty() ? totalElements : indexStack.peek()) - i;

            // Push the current index on to the stack.
            indexStack.push(i);
        }

        // Return the count of valid subarrays.
        return validSubarrayCount;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int validSubarrays(vector<int>& nums) {
        int n = nums.size(); // Get the number of elements in the nums vector.
        stack<int> indexStack; // This stack will store the indices of elements in nums.
        int answer = 0; // This will hold the count of valid subarrays.

        // Start from the end of the nums vector and move backwards.
        for (int i = n - 1; i >= 0; --i) {
            // Pop elements from the stack if they are greater than or equal to nums[i].
            // We want to find the next smaller element in the remaining array.
            while (!indexStack.empty() && nums[indexStack.top()] >= nums[i]) {
                indexStack.pop();
            }

            // If stack is not empty, then the current valid subarray size is (index at top of stack - current index).
            // If the stack is empty, all elements to the right are greater, hence (n - current index).
            answer += indexStack.empty() ? n - i : indexStack.top() - i;

            // Push the current index into the stack.
            indexStack.push(i);
        }

        // Return the final count of valid subarrays.
        return answer;
    }
};
```

Typescript Solution

```typescript
function validSubarrays(nums: number[]): number {
    const arrayLength = nums.length; // Total number of elements in the input array.
    const stack: number[] = []; // Stack to hold indices of elements.
    let countValidSubarrays = 0; // Stores the count of valid subarrays.

    // Iterate over the array in reverse.
    for (let index = arrayLength - 1; index >= 0; --index) {
        // Pop elements from the stack while the current element is less than
        // or equal to the element corresponding to the top index in the stack.
        while (stack.length && nums[stack[stack.length - 1]] >= nums[index]) {
            stack.pop();
        }

        // Increment the count of valid subarrays by the difference of
        // top stack index (representing the next greater element's index) and current index.
        // If stack is empty, use arrayLength as all elements to the right are valid.
        countValidSubarrays += (stack.length ? stack[stack.length - 1] : arrayLength) - index;

        // Push the current index onto the stack.
        stack.push(index);
    }

    // Return the total count of valid subarrays.
    return countValidSubarrays;
}
```

Time and Space Complexity

The provided code snippet defines a function `validSubarrays` that calculates the number of valid subarrays in a given list of integers, `nums`. A valid subarray here is defined such that for every element in the subarray, there is no smaller element to its right in the subarray.

Time Complexity

The time complexity of the function primarily depends on the loop and the while loop used for comparison and popping the stack.

The outer loop runs in reverse, starting from the last element of the array (`n - 1`) to the first one (index `0`) exactly once, thus iterating `n` times, where `n` is the length of the array. For each element, the inner `while` loop runs only as long as there are elements in the stack and the condition `nums[stk[-1]] >= nums[i]` holds true.

In the worst-case scenario, each element will be pushed and popped from the stack exactly once, since an element would only be popped if a smaller element is encountered, ensuring that each pair of elements is compared at most once during the iteration. This leads to the conclusion that the complexity of both loops combined still remains O(n), because the inner while loop operations are distributing the cost of `n` pop operations across `n` iterations, which is an amortized analysis.

Hence, the time complexity of this function is $O(n)$.

Space Complexity

The space complexity is mainly due to the additional data structure used - the stack `stk`. In the worst-case scenario, the stack will contain all `n` elements from the list `nums` when the list elements are in non-decreasing order. In this case, all elements will be pushed to the stack, and none will be popped until the loop is completely finished.

Therefore, the space complexity of this function is $O(n)$.