795. Number of Subarrays with Bounded Maximum

Medium Array Two Pointers

Problem Description

Given an array of integers called `nums`, and two other integers known as `left` and `right`, the problem asks us to calculate how many non-empty subarrays (continuous parts of the array) have a maximum element that falls within the range `[left, right]`. To clarify, for a subarray to be counted, its largest number must be greater than or equal to `left` and less than or equal to `right`. Essentially, we're tasked with identifying portions of the array where if you pick out the biggest number, it wouldn't be smaller than `left` or larger than `right`.

Intuition

To solve this problem, we can use the concept of counting subarrays with elements less than or equal to a given maximum. The provided solution uses a helper function `f(x)` which counts subarrays where all elements are less than or equal to `x`. The idea is that we can count all subarrays that have elements under or equal to `right`, which captures all subarrays that could potentially meet our condition. Then, we subtract the count of subarrays with elements under or equal to `left - 1`, which represents subarrays that don't meet our condition (since their maximum would be too small).

The reasoning here is that when we have the count of subarrays with maximums not surpassing `right`, we include both valid subarrays (those we are looking for) and subarrays that have maximums that are too small. We don't want the latter, so we find out how many of them there are (using the same count technique but with a maximum of `left - 1`) and remove them from our total. The difference gives us exactly the number of valid subarrays we're seeking.

The `f(x)` function works by scanning through the array and using a temporary count `t` to keep track of valid subarrays ending at the current position. If a value is encountered that is not greater than `x`, it can be added to existing valid subarrays to form new ones; hence, `t` is incremented by 1. If a value is too large, `t` is reset to 0, as no new valid subarrays can end at this position. The variable `cnt` accumulates the count of all valid subarrays as we iterate through `nums`.

Solution Approach

The approach taken in the given solution involves a clever use of prefix sums, a common technique in which we construct an array of sums that represent the sum of elements up to a certain index. However, in this specific implementation, rather than directly computing a sum, we are counting the possible subarrays up to a certain index that respect the maximum value condition.

Let's break down the `f(x)` function, which is at the heart of the approach:

1. Initialize `cnt` and `t` to 0. `cnt` will hold the total count of subarrays that do not exceed the maximum allowed element `x`, and `t` will count such subarrays ending at the current index in `nums`.
2. Iterate over each value `v` in the `nums` array.
3. If `v` is greater than `x`, then any subarray including `v` cannot be part of our solution, so we reset `t` to 0. This essentially says "you can't start or continue a valid subarray from this point."
4. If `v` is less than or equal to `x`, then we can extend every subarray counted by `t` with this new value `v` plus one extra: the subarray that only includes `v` itself. Thus, we increment `t` by 1.
5. For every iteration, we add `t` to `cnt` which accumulates our count.

The outer function `numSubarrayBoundedMax` makes use of `f(x)` to apply our subarray counting method twice:

- First, we count all valid subarrays with maximums not more than `right`, which gives us `f(right)`.
- Then, we count the subarrays that are strictly smaller than `left`, which is `f(left - 1)`.

The subarrays we want to exclude from our count are the ones where the maximum element value is strictly less than 'left'. By counting these with `f(left - 1)`, we get the count of subarrays with maximum values in the range `[0, left-1]`. Thus, when we subtract `f(left - 1)` from `f(right)`, we remove these invalid subarrays from our total, leaving us with the number of valid subarrays that have their maximum element between `left` and `right`, inclusive.

This elegant subtraction of prefix counts helps to efficiently compute the final desired count without having to evaluate each subarray independently, thereby reducing the time complexity.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Suppose we have the array `nums = [2, 1, 4, 3]` and the integers `left = 2` and `right = 3`. We want to count the number of subarrays where the maximum is at least `left` and at most `right`.

First, let's consider the `f(x)` function that counts the subarrays with maximum elements less than or equal to `x`:

- Using `f(right)`, which is `f(3)`:

  1. Initialize `cnt = 0` and `t = 0`.
  2. For `v = 2` (first value), `t` is incremented to 1 (`t += 1`), so `cnt = 1`.
  3. For `v = 1` (second value), `t` is incremented to 2 (`t += 1`), so `cnt = 3`.
  4. For `v = 4` (third value), `t` is reset to 0, as 4 is greater than 3.
  5. For `v = 3` (fourth value), `t` is incremented to 1 (`t += 1`), so `cnt = 4`.

  `f(3)` gives us a count of 4 subarrays.

- Using `f(left - 1)`, which is `f(1)`:

  1. Initialize `cnt = 0` and `t = 0`.
  2. For `v = 2` (first value), `t` is reset to 0, since 2 is greater than 1.
  3. For `v = 1` (second value), `t` is incremented to 1 (`t += 1`), so `cnt = 1`.
  4. For `v = 4` (third value), `t` is reset to 0, as it's greater than 1.
  5. For `v = 3` (fourth value), `t` is reset again to 0, since 3 is also greater than 1.

  `f(1)` gives us a count of 1 subarray.

Now, we calculate the number of subarrays we are looking for by subtracting `f(left - 1)` from `f(right)`:

- Count of subarrays we are seeking = `f(right) - f(left - 1) = 4 - 1 = 3`.

So, there are 3 subarrays where the maximum element is between 2 and 3, inclusive. These subarrays are `[2]`, `[2, 1]`, and `[3]`. This method allows us to efficiently calculate the desired count without having to examine each subarray individually.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def numSubarrayBoundedMax(self, nums: List[int], left: int, right: int) -> int:
        # Define a helper function that counts the number of subarrays
        # with element values less than or equal to 'bound'.
        def count_subarrays(bound: int) -> int:
            count = temp_count = 0
            for value in nums:
                # Reset temporary count if value is greater than the bound,
                # or increase it if value is within the subarray criteria.
                temp_count = 0 if value > bound else temp_count + 1
                # Add the current temporary count to the total count.
                count += temp_count
            return count

        # Calculate the number of subarrays where the maximum element is
        # between 'left' and 'right' by taking the count of subarrays bounded
        # by 'right' and subtracting those strictly bounded by 'left - 1'.
        # This ensures that we only include subarrays where the maximum
        # value is at least 'left'.
        return count_subarrays(right) - count_subarrays(left - 1)
```

Java

```java
class Solution {
    // Method to count the number of subarrays with elements bounded by left and right
    public int numSubarrayBoundedMax(int[] nums, int left, int right) {
        // Calculate the number of subarrays where the maximum element is at most right
        // and subtract the count of subarrays where the maximum element is less than left
        return countSubarraysAtMost(nums, right) - countSubarraysAtMost(nums, left - 1);
    }

    // Helper method to count the number of subarrays with elements at most x
    private int countSubarraysAtMost(int[] nums, int bound) {
        int count = 0; // Count of subarrays meeting the condition
        int currentSubarrayCount = 0; // Temporary count for current valid subarray sequence

        // Iterate through each element of the array
        for (int num : nums) {
            // If the current element is not greater than bound, extend the current subarray sequence
            if (num <= bound) {
                currentSubarrayCount += 1; // Include this number in the current subarray
            } else {
                // If current element is greater than bound, reset the temporary count
                currentSubarrayCount = 0;
            }

            // Add the count of valid subarrays up to the current index
            count += currentSubarrayCount;
        }

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

C++

```cpp
class Solution {
public:
    // Function finds the number of subarrays with elements bounded by left and right.
    int numSubarrayBoundedMax(vector<int>& nums, int left, int right) {
        // Lambda function to count subarrays where the max element is less than or equal to a given bound.
        auto countSubarrays = [&](int bound) {
            int count = 0; // Number of valid subarrays
            int currentLength = 0; // Current length of the subarray satisfying the condition

            // Iterate through the elements of the nums vector
            for (int value : nums) {
                // If the current value is greater than the bound, reset currentLength to 0
                // If not, increase currentLength by 1 (extend the valid subarray by the current element)
                currentLength = value > bound ? 0 : currentLength + 1;

                // Add the length of the current valid subarray to count
                // This adds all subarray ends with this element and bounded by the value
                count += currentLength;
            }
            return count;
        };

        // Number of subarrays where the max element is less than or equal to right
        // and subtracting those where the max element is less than left (to exclude them)
        return countSubarrays(right) - countSubarrays(left - 1);
    }
};
```

TypeScript

```typescript
// The type definition for a list of numbers
type NumberList = number[];

// Function finds the number of subarrays with elements bounded by 'left' and 'right'.
// Params:
// nums: The input array of numbers.
// left: The lower bound for the maximum of the subarrays.
// right: The upper bound for the maximum of the subarrays.
// Returns: The count of subarrays meeting the criteria.
function numSubarrayBoundedMax(nums: NumberList, left: number, right: number): number {
    // Lambda function to count subarrays where the max element is less than or equal to a given bound.
    // Params:
    // bound: The bound to check for in subarrays.
    // Returns: The count of subarrays where the maximum element is less than or equal to the bound.
    const countSubarrays = (bound: number): number => {
        let count = number = 0; // Number of valid subarrays
        let currentLength: number = 0; // Current length of the subarray satisfying the condition

        // Iterate through the elements of the nums array
        for (let value of nums) {
            // If the current value is greater than the bound, reset currentLength to 0
            // If not, increase currentLength by 1 (extend the valid subarray by the current element)
            currentLength = value > bound ? 0 : currentLength + 1;

            // Add the length of the current valid subarray to count
            // This adds all subarray ends with this element and bounded by the value
            count += currentLength;
        }
        return count;
    };

    // Number of subarrays where the max element is less than or equal to right
    // and subtracting those where the max element is less than left (to exclude them)
    return countSubarrays(right) - countSubarrays(left - 1);
}

// Example usage:
// const result = numSubarrayBoundedMax([2, 1, 4, 3], 2, 3);
// console.log(result); // Outputs the count of subarrays meeting the criteria
```

Python

```python
from typing import List

class Solution:
    def numSubarrayBoundedMax(self, nums: List[int], left: int, right: int) -> int:
        # Define a helper function that counts the number of subarrays
        # with element values less than or equal to 'bound'.
        def count_subarrays(bound: int) -> int:
            count = temp_count = 0
            for value in nums:
                # Reset temporary count if value is greater than the bound,
                # or increase it if value is within the subarray criteria.
                temp_count = 0 if value > bound else temp_count + 1
                # Add the current temporary count to the total count.
                count += temp_count
            return count

        # Calculate the number of subarrays where the maximum element is
        # between 'left' and 'right' by taking the count of subarrays bounded
        # by 'right' and subtracting those strictly bounded by 'left - 1'.
        # This ensures that we only include subarrays where the maximum
        # value is at least 'left'.
        return count_subarrays(right) - count_subarrays(left - 1)
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is determined by the function `f(x)` which is called twice within the `numSubarrayBoundedMax` method. This function iterates through each element of the `nums` array exactly once. The number of elements in the `nums` array is represented as `n`.

Given that there are no nested loops and each operation inside the loop takes constant time, the time taken by `f(x)` is proportional to the number of elements in the array, which is $O(n)$. Since `f(x)` is called twice, the total time complexity is $O(2n)$, which simplifies to $O(n)$.

Space Complexity

The space complexity of the code can be analyzed by looking at the space taken by variables that could scale with the input. The function `f(x)` uses two variables `cnt` and `t` that do not scale with the size of the input array and are thus constant extra space ($O(1)$).

Since there are no additional data structures utilized that grow proportionally with the input size, the space complexity remains constant, irrespective of the size of the input array `nums`. Therefore, the space complexity of the code is $O(1)$.