2104. Sum of Subarray Ranges

Medium Stack Array Monotonic Stack Leetcode Link

Problem Description

The problem provides us with an integer array called `nums`. We need to calculate the sum of the ranges of all possible subarrays of this array. A subarray is defined as a contiguous non-empty sequence of elements within an array, and the range of a subarray is the difference between the largest (maximum) and smallest (minimum) element in the subarray. In other words, we have to find all the subarrays, determine the range for each one, and then sum these ranges to get the final result.

Intuition

To solve this problem efficiently, we need to think beyond the brute force approach, which would involve finding all possible subarrays and then calculating the range for each. Since the number of subarrays grows quadratically with the array size, this would lead to a time-consuming solution. Instead, we need an optimized approach.

The key insight is to count how many times each element of the array becomes a minimum and a maximum of a subarray. We can find this out by determining, for each element, the bounds (to the left and to the right) within which it is the minimum or maximum. These bounds can be found using the concept of "monotonic stacks", which are stacks that maintain elements in a either strictly increasing or decreasing order.

The provided solution defines a function `f(nums)`, which uses a monotonic stack to calculate the sum of the product of the number of subarrays, where each element of `nums` is the maximum. It initializes two lists, `left` and `right`, that store the indices of the next smaller elements to the left and right, respectively. When traversing the array, we maintain a stack that keeps track of the indices of elements in a non-decreasing order. If the current element is greater than the element at the top of the stack, we pop elements from the stack until we find an element smaller or the stack becomes empty. This process helps us locate the bounds for each element where it stands as the maximum.

Once we have the `left` and `right` arrays, we can calculate the sum of products of differences of indices and the value at each index.

For the minimum case, we invert the `nums` array by negating each element and pass it to the same function `f`. The inversion swaps the roles of maximum and minimum elements within the subarrays.

Finally, we sum up the maximums and minimums to get the total sum of subarray ranges. This approach is efficient because each element is pushed and popped from the stack only once, making it an O(n) solution, where n is the length of the array.

Solution Approach

The solution applies a technique using 'monotonic stacks' to calculate the sum of all subarray ranges efficiently. Here is how the implementation unfolds:

1. **Monotonic Stacks**: We use two monotonic stacks to find the bounds within which each element is the maximum and minimum, respectively. A monotonic stack is a stack that is either strictly increasing or decreasing.
2. **Finding Left and Right Bounds**:
   - For each element in `nums`, we find the nearest smaller element to the left and store its index in the `left` array. If there's no such element, we store `-1`.
   - Similarly, we find the nearest smaller element to the right for each element and store its index in the `right` array. If there's no such element, `n` (the length of `nums`) is stored.
3. **Calculating the Sum for Maximums**:
   - We iterate through `nums`, using the indices in `left` and `right` to determine the number of subarrays where each element is the maximum.
   - The sum for each element as the maximum is calculated by multiplying the element's value by the difference between its index and its `left` bound and by the difference between its `right` bound and its index. This product represents the sum of ranges for the subarrays where this element is the maximum.
4. **Calculating the Sum for Minimums**:
   - We negate each element in `nums` and repeat the above process. This effectively swaps the roles of the maximums and minimums without changing the range calculation logic.
5. **Combine Sums of Maximums and Minimums**:
   - The function `f(nums)` calculates the sum assuming `nums[i]` is the maximum in its subarrays, while `f([-v for v in nums])` calculates the sum assuming `nums[i]` is the minimum.
   - The overall solution is the sum of these two results, giving us the sum of all subarray ranges.
6. **Efficiency**:
   - The algorithm is efficient because each element is pushed onto and popped from the stack only once. Since each element is dealt with in constant time when it is at the top of the stack, the total time complexity for the algorithm is `O(n)`, where `n` is the number of elements in `nums`.

The code leverages the concept of calculating the contribution of each element to the sum of subarray ranges separately as a maximum and minimum, which allows for an elegant and efficient solution. It bypasses the need to enumerate every subarray explicitly, which would be computationally expensive for larger arrays.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach.

Consider the array `nums = [3, 1, 2, 4]`. We want to find the sum of the ranges of all possible subarrays.

1. **Monotonic Stacks**: We will use monotonic stacks to find bounds for each element where it acts as the maximum and minimum of subarrays.
2. **Finding Left and Right Bounds**:
   - For maximums, starting with an empty stack, we process each element in `nums` from left to right. Taking the first element `3`, there is no smaller element to the left, so `left[0] = -1`. Our stack will be [3].
   - Moving to `1`, it is smaller than `3`, so for the maximum case, `left[1] = -1`. We update our stack to [1].
   - For `2`, the nearest smaller element to the left is `1`, so `left[2] = 1`. Our stack is now [1, 2].
   - Lastly for `4`, there are no smaller elements to the left, so `left[3] = -1`, and our stack ends as [1, 2, 4].
   - To find the right bounds, we assume each element is the last element (`right[i] = n`, with `n = 4`). Hence, `right = [4, 4, 4, 4]` as no element in `nums` has a next smaller element on the right.
3. **Calculating the Sum for Maximums**:
   - For `3`, it is the maximum for subarrays using the elements `[3], [3, 1], [3, 1, 2], [3, 1, 2, 4]`. The sum of these subarray ranges is `3*1*4 = 12`.
   - For `1`, it is not the maximum for any subarrays; we can skip it for the maximum sum.
   - For `2`, it is the maximum for subarrays [2] and [1, 2], with corresponding indices [2, 3]. The sum is `2*1*1 = 2`.
   - For `4`, it is the maximum for subarrays [4], [2, 4], and [1, 2, 4], which is a sum of `4*3*1 = 12`.
4. **Calculating the Sum for Minimums**:
   - We negate each element of nums to get `[-3, -1, -2, -4]` and repeat steps 2 and 3. For minimums, `4` and `3` never become the minimum of any subarrays.
   - `-1` is the minimum for the subarrays `[-1], [-3, -1], [-1, -2], [-3, -1, -2], [-1, -2, -4], [-3, -1, -2, -4]`, with sum `-1*(1+2+1+2)*1 = -12`.
   - `-2` is the minimum for subarrays `[-2], [-1, -2], [-2, -4], [-1, -2, -4]`, with sum `-2*(1+2)*2 = -12`.
5. **Combine Sums of Maximums and Minimums**:
   - The sum of subarray ranges considering maximums from step 3 is `12 + 0 + 2 + 12 = 26`.
   - The sum of subarray ranges considering minimums from step 4 is `0 + (-12) + (-12) + 0 = -24`.
   - Adding these, we get `26 - 24 = 2`.
6. **Efficiency**:
   - During this example, each element was processed in constant time, showing how each element contributes to the sum with a single pass through the array without explicitly considering every subarray.

So the sum of the ranges of all possible subarrays of `nums = [3, 1, 2, 4]` is `2`.

Python Solution

```python
from typing import List

class Solution:
    def subArrayRanges(self, nums: List[int]) -> int:
        # Function to calculate the sum of maximum or minimum of all subarrays
        def calculate_subarray_sums(nums):
            # Initialize stack, left and right index arrays
            stack = []
            n = len(nums)
            # Left index for the nearest smaller number to the left of the current number
            left_indices = [-1] * n
            # Right index for the nearest smaller number to the right of the current number
            right_indices = [n] * n

            # Calculate the left indices
            for i, value in enumerate(nums):
                # If the stack is not empty and current element is greater
                # pop elements from the stack
                while stack and nums[stack[-1]] <= value:
                    stack.pop()
                # If stack is not empty assign the last element as the left limit
                if stack:
                    left_indices[i] = stack[-1]
                # Push the current index onto the stack
                stack.append(i)

            # Reset the stack for the right indices calculation
            stack = []

            # Calculate the right indices
            for i in range(n - 1, -1, -1):
                # If the stack is not empty and current element is greater, pop from stack
                while stack and nums[stack[-1]] < nums[i]:
                    stack.pop()
                # If stack is not empty, assign the last element as the right limit
                if stack:
                    right_indices[i] = stack[-1]
                # Push the current index onto the stack
                stack.append(i)

            # Calculate the final sum
            return sum((i - left_indices[i]) * (right_indices[i] - i) * value for i, value in enumerate(nums))

        # Calculate maximum values sum
        max_sum = calculate_subarray_sums(nums)
        # Calculate minimum values sum (inverting the values)
        min_sum = calculate_subarray_sums([-value for value in nums])
        # Return the total sum of all subarray ranges
        return max_sum + min_sum

# Example usage:
# solution = Solution()
# print(solution.subArrayRanges([1, 2, 3]))  # Output: 4

```

Java Solution

```java
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

class Solution {
    public long subArrayRanges(int[] nums) {
        // Calculate the sum of max elements in all subarrays
        long sumOfMax = calculateSubarrayValues(nums);

        // Invert all numbers in nums to find the sum of min elements using the same function
        for (int i = 0; i < nums.length; ++i) {
            nums[i] *= -1;
        }
        long sumOfMin = calculateSubarrayValues(nums);

        // The total sum of subarray ranges is the sum of max elements plus the (inverted) sum of min elements
        return sumOfMax + sumOfMin;
    }

    private long calculateSubarrayValues(int[] nums) {
        Deque<Integer> stack = new ArrayDeque<>();
        int n = nums.length;
        int[] left = new int[n];
        int[] right = new int[n];
        Arrays.fill(left, -1);  // Initialize left bounds for all elements
        Arrays.fill(right, n);  // Initialize right bounds for all elements

        // Calculate the left bounds for each element
        for (int i = 0; i < n; ++i) {
            while (!stack.isEmpty() && nums[stack.peek()] <= nums[i]) {
                stack.pop();
            }
            if (!stack.isEmpty()) {
                left[i] = stack.peek();
            }
            stack.push(i);
        }

        stack.clear(); // Clear the stack for the next phase of calculations

        // Calculate the right bounds for each element
        for (int i = n - 1; i >= 0; --i) {
            while (!stack.isEmpty() && nums[stack.peek()] < nums[i]) {
                stack.pop();
            }
            if (!stack.isEmpty()) {
                right[i] = stack.peek();
            }
            stack.push(i);
        }

        long sum = 0; // Initialize the sum to aggregate the subarray values
        // Calculate the sum of values using the left and right bounds
        for (int i = 0; i < n; ++i) {
            sum += (long) (i - left[i]) * (right[i] - i) * nums[i];
        }

        // Return the total sum of subarray values for either max or min based on the nums state
        return sum;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the sum of ranges of all subarrays
    long long subArrayRanges(vector<int>& nums) {
        long long maxSum = calculateSum(nums); // Sum of max elements of all subarrays

        // Negate all the elements to use the same function for minimum
        for (int i = 0; i < nums.size(); ++i) {
            nums[i] *= -1;
        }

        long long minSum = calculateSum(nums); // Sum of min elements of all subarrays
        return maxSum + minSum; // Sum of ranges (max-min) of all subarrays
    }

    // Helper function to calculate the sum of either max or min elements of all subarrays
    long long calculateSum(vector<int>& nums) {
        stack<int> st; // Stack to maintain indices
        int n = nums.size();
        vector<int> left(n, -1); // Indices of previous smaller or equal elements
        vector<int> right(n, n); // Indices of next smaller elements
        long long sum = 0; // Resulting sum

        // Fill left array with previous smaller or equal indices
        for (int i = 0; i < n; ++i) {
            while (!st.empty() && nums[st.top()] <= nums[i]) st.pop();
            if (!st.empty()) left[i] = st.top();
            st.push(i);
        }

        // Clear stack to reuse it for the right array
        st = stack<int>();

        // Fill right array with next smaller indices
        for (int i = n - 1; i >= 0; --i) {
            while (!st.empty() && nums[st.top()] < nums[i]) st.pop();
            if (!st.empty()) right[i] = st.top();
            st.push(i);
        }

        // Calculate the sum based on the left and right arrays
        for (int i = 0; i < n; ++i) {
            sum += (long long) (i - left[i]) * (right[i] - i) * nums[i];
        }

        return sum; // Return the total sum for the array
    }
};

```

Typescript Solution

```typescript
function subArrayRanges(nums: number[]): number {
    // Define the length of the nums array for later use.
    const length = nums.length;
    // This variable will store the cumulative sum of all subarray ranges.
    let totalRangeSum = 0;

    // Outer loop to consider starting point of subarrays.
    for (let start = 0; start < length - 1; start++) {
        // Initialize min and max with the first element of the current subarray.
        let currentMin = nums[start];
        let currentMax = nums[start];

        // Inner loop to consider different ending points of subarrays.
        for (let end = start + 1; end < length; end++) {
            // Update the current minimum and maximum of the subarray.
            currentMin = Math.min(currentMin, nums[end]);
            currentMax = Math.max(currentMax, nums[end]);
            // Add the range (max - min) of this subarray to the total sum.
            totalRangeSum += currentMax - currentMin;
        }
    }

    // Return the total sum of all ranges.
    return totalRangeSum;
}

```

Time and Space Complexity

The given code calculates the sum of the range of each subarray in an integer array. It defines a function `f` that computes the cumulative product of the value at each position, the distance to the closest smaller element on the left, and the distance to the closest smaller element on the right.

Time Complexity:

1. The function `f` loops through the array twice - once in the forward direction and once in the reverse direction. In each loop, every element is processed only once, and each element is inserted and removed from the stack at most once. This leads to a time complexity of `O(n)` for each loop, where `n` is the length of the `nums`.
2. The sum computation iterates over each element in `nums` once, executing a constant amount of work for each element. Thus, this also takes `O(n)` time.
3. Since the function `f` is called twice - once for the original `nums` array and once for the negated values of `nums`, the total time complexity is `O(2n)`, which simplifies to `O(n)`.

Therefore, the overall time complexity of the `subArrayRanges` method is `O(n)`.

Space Complexity:

1. There are two additional arrays created, `left` and `right`, each of the same length as `nums`, leading to a space complexity of `O(2n)` which simplifies to `O(n)`.
2. Additionally, a stack is used to keep track of indices while iterating through the array. In the worst case, this stack could hold all `n` elements at once if the array is strictly monotonic. This does not increase the overall space complexity since it remains `O(n)`.
3. There are no recursive calls or additional data structures that grow with the size of the input beyond what has already been accounted for.

Hence, the overall space complexity for the `subArrayRanges` method is `O(n)`.