907. Sum of Subarray Minimums

Medium · Stack · Array · Dynamic Programming · Monotonic Stack

Problem Description

The problem presents a challenge where you're given an array of integers, `arr`, and you need to calculate the sum of the minimum element for every possible contiguous subarray within `arr`. Due to potentially large numbers, the result should be returned modulo `10^9 + 7`. A contiguous subarray is a sequence of elements from the array that are consecutive with no gaps. For example, in the array `[3, 1, 2, 4]`, `[1, 2, 4]` is a contiguous subarray, but `[3, 2]` is not.

Intuition

Solving this problem efficiently requires an understanding that each element of the array will be the minimum in some number of subarrays. So, rather than considering every subarray explicitly, we conceptualize the problem around each element of the array and determine how many subarrays it is the minimum element of.

To arrive at the solution, we must track two things for each element `arr[i]`:

1. `left[i]`: the index of the first smaller element to the left of `arr[i]`
2. `right[i]`: the index of the first element that is less than or equal to `arr[i]` to the right

With `left[i]` and `right[i]` determined, the number of subarrays in which `arr[i]` is the minimum can be calculated by `(i - left[i]) * (right[i] - i)`.

Notice that we are specifically looking for the first element to the right that is less than or equal to `arr[i]` to prevent double-counting certain subarrays. If we used a strictly less than condition, subarrays with same minimum values at different positions could be considered twice.

To maintain these `left` and `right` indices, we use a monotonic stack, which is a stack that keeps elements in either increasing or decreasing order. The monotonic stack is traversed twice: once from left to right to find each `left[i]`, and once from right to left to find each `right[i]`. Each element in the stack represents the next greater element for elements not yet encountered or processed.

The product of `(i - left[i])` and `(right[i] - i)` gives us the count of subarrays where `arr[i]` is the minimum. This count is then multiplied by `arr[i]` to get the contribution to the sum from `arr[i]`. Finally, we sum up the contributions from all elements and return it modulo `10^9 + 7` to handle the large number possibility.

Solution Approach

The implementation leverages two main concepts: the "Monotonic Stack" pattern and the "Prefix Sum" pattern, to efficiently solve the problem without having to evaluate every subarray explicitly. Here's the walk-through of the implemented solution, step-by-step:

1. Initialize two arrays, `left` and `right`, of the same length as `arr` to `n`, with `-1` and `n` respectively. These arrays will hold for each element the index of the previous smaller element (`left`) and the next smaller or equal element (`right`).

2. Initialize a stack `stk` which we'll use to iterate over the array to find the `left` and `right` indices. The stack approach efficiently maintains a decreasing order of elements and their indices.

3. Iterate through the elements of `arr` from left to right. For each element, while the stack is not empty and the top element of the stack is greater than or equal to the current element, pop elements from the stack. This process is maintaining the stack in a strictly decreasing order.

4. After elements larger than the current one are popped off `stk`, if the stack is not empty, set `left[i]` to the index of the top element of `stk`, which is the closest previous element smaller than `arr[i]`. Then, push the current index `i` onto the stack.

5. Clear the stack and then iterate through the elements of `arr` from right to left to similarly identify `right[i]` for each element. The process mirrors step 3 and 4, but in the reversed direction and with the condition that any equal value element could also terminate the loop, maintaining strict decreasing order up to equal values.

6. Once both `left` and `right` arrays are filled with proper indices, calculate the sum. By iterating over all indices `i`, find the product of the count of subarrays where `arr[i]` is the minimum `((i - left[i]) * (right[i] - i))` and `arr[i]`. This represents the sum of `arr[i]` for all `i` in its valid subarrays.

7. Sum these products for all `i`. As the final sum might be very large, each addition is taken modulo `10^9 + 7` to prevent integer overflow.

By combining the Monotonic Stack to find bounds for each element and the Prefix Sum pattern to calculate each element's contribution to the total sum, the algorithm achieves an efficient solution that operates in O(n) time complexity, where n is the size of the input array `arr`.

Example Walkthrough

Let's consider the array `arr = [3, 1, 2, 4]` and walk through the solution method to understand how the implemented algorithm works step-by-step:

1. First, initialize two arrays, `left` as `[-1, -1, -1, -1]` and `right` as `[4, 4, 4, 4]`. The `-1` indicates that we didn't find a previous smaller element yet, and `4` is used because it's the size of `arr`, indicating we haven't found the next smaller or equal element yet.

2. Initialize an empty stack `stk` to keep track of the indices of the elements we browse through.

3. Iterate through `arr` from left to right:

   - i = 0, `arr[i] = 3`, stack is empty, so push `0` onto the stack.
   - i = 1, `arr[i] = 1`, stack top element is `3 (> 1)`, so pop `0`. `left[1]` becomes the previous element, `-1`. Stack is empty, push `1`.
   - i = 2, `arr[i] = 2`, stack top element is `1 (< 2)`, do nothing. Push `2` to the stack.
   - i = 3, `arr[i] = 4`, stack top element is `2 (< 4)`, do nothing. Push `3` to the stack.

   After this loop, `left` becomes `[-1, -1, 1, 2]`, and stack `stk` contains `[1, 2, 3]`.

4. Clear the stack for the next phase.

5. Iterate through `arr` from right to left to fill the `right` array:

   - i = 3, `arr[i] = 4`, stack is empty, push `3` onto the stack.
   - i = 2, `arr[i] = 2`, stack top element is `4 (> 2)`, so pop `3`. Push `2` onto the stack.
   - i = 1, `arr[i] = 1`, stack top element is `2 (> 1)`, pop `2`, `right[1]` is `2`. Stack is now empty again, push `1`.
   - i = 0, `arr[i] = 3`, stack top element is `1 (< 3)`, so do nothing. Push `0` onto the stack.

   After this loop, `right` becomes `[4, 2, 4, 4]`, and stack `stk` contains `[1, 0]`.

6. Now, the sum is calculated. Iterate over the indices `i` and for each:

   - i = 0, `(i - left[i]) * (right[i] - i)` → `(0 - (-1)) * (4 - 0)` → `1 * 4 = 4` → `4 * 3 = 12`.
   - i = 1, `(i - left[i]) * (right[i] - i)` → `(1 - (-1)) * (2 - 1)` → `2 * 1 = 2` → `2 * 1 = 2`.
   - i = 2, `(i - left[i]) * (right[i] - i)` → `(2 - 1) * (4 - 2)` → `1 * 2 = 2` → `2 * 2 = 4`.
   - i = 3, `(i - left[i]) * (right[i] - i)` → `(3 - 2) * (4 - 3)` → `1 * 1 = 1` → `1 * 4 = 4`.

   The total sum is `12 + 2 + 4 + 4 = 22`.

7. The result would be the sum `22` modulo `10^9 + 7`, which is simply `22`, as it's less than the modulo value.

By following the steps outlined in the solution, we efficiently find the sum without ever calculating each subarray's minimum explicitly. This example demonstrates how both Monotonic Stack and Prefix Sum patterns can be harnessed to solve a seemingly complex array problem with an elegant and efficient algorithm.

Python Solution

```python
# The Solution class contains a method to find the sum of minimums of all subarrays in a given array.
class Solution:
    def sumSubarrayMins(self, arr: List[int]) -> int:
        n = len(arr)  # Get the length of the input array
        left = [-1] * n  # Store the index of previous less element for each element in the array
        right = [n] * n  # Store the index of next less element for each element in the array
        stack = []  # Initialize an empty stack for indices

        # Calculate the previous less element for each element in the array
        for i, value in enumerate(arr):
            while stack and arr[stack[-1]] >= value:  # Ensure that the top of the stack is < current element value
                stack.pop()  # Pop elements from stack while the current element is smaller or equal
            if stack:
                left[i] = stack[-1]  # Update left index if stack is not empty
            stack.append(i)  # Push the current index onto the stack

        stack = []  # Reset the stack for the next loop

        # Calculate the next less element for each element in the array, going backwards
        for i in range(n - 1, -1, -1):  # Start from end of the array and move backwards
            while stack and arr[stack[-1]] > arr[i]:  # Similar stack operation but with strict inequality
                stack.pop()  # Pop elements while current element is smaller
            if stack:
                right[i] = stack[-1]  # Update right index if stack is not empty
            stack.append(i)  # Push current index to the stack

        mod = 10**9 + 7  # Define modulus for the final result

        # Calculate the sum of all minimum subarray values with their respective frequencies
        # The frequency is the product of the lengths of subarrays to the left and right
        result = sum((i - left[i]) * (right[i] - i) * value for i, value in enumerate(arr)) % mod

        return result  # Return the result sum, modulo 10^9 + 7
```

Java Solution

```java
class Solution {
    public int sumSubarrayMins(int[] arr) {
        int length = arr.length;
        // Arrays to keep track of previous smaller and next smaller elements
        int[] left = new int[length];
        int[] right = new int[length];

        // Initialize left array with -1 indicating the start of array
        Arrays.fill(left, -1);
        // Initialize right array with length of array indicating the end of array
        Arrays.fill(right, length);

        // Stack to keep track of elements while traversing
        Deque<Integer> stack = new ArrayDeque<>();

        // Calculate previous smaller elements for each element in the array
        for (int i = 0; i < length; ++i) {
            // Popping all elements which are greater than the current element
            while (!stack.isEmpty() && arr[stack.peek()] >= arr[i]) {
                stack.pop();
            }
            // The current top of the stack indicates the previous smaller element
            if (!stack.isEmpty()) {
                left[i] = stack.peek();
            }
            // Push the current index into the stack
            stack.push(i);
        }

        // Clear the stack for next traversal
        stack.clear();

        // Calculate next smaller elements for each element in the array in reverse order
        for (int i = length - 1; i >= 0; --i) {
            // Popping all elements which are greater than or equal to the current element
            while (!stack.isEmpty() && arr[stack.peek()] > arr[i]) {
                stack.pop();
            }
            // The current top of the stack indicates the next smaller element
            if (!stack.isEmpty()) {
                right[i] = stack.peek();
            }
            // Push the current index into the stack
            stack.push(i);
        }

        // The mod value for big integer operations to prevent overflow
        int mod = (int) 1e9 + 7;
        // The result variable to keep track of the sum of subarray minimums
        long answer = 0;

        // Calculate the contribution of each element as a minimum in its possible subarrays
        for (int i = 0; i < length; ++i) {
            // Total count of subarrays where arr[i] is min is (i - left[i]) * (right[i] - i)
            // Multiply the count by the value itself and apply modulo operation
            answer += (long) (i - left[i]) * (right[i] - i) % mod * arr[i] % mod;
            // Ensure the running sum doesn't overflow
            answer %= mod;
        }

        // Cast the long result to int before returning as per method return type
        return (int) answer;
    }
}
```

C++ Solution

```cpp
using ll = long long;  // Define 'll' as an alias for 'long long' type
const int MOD = 1e9 + 7;  // The modulo value

class Solution {
public:
    // Function to calculate the sum of minimum elements in every subarray
    int sumSubarrayMins(vector<int>& nums) {
        int length = nums.size();  // Get the number of elements in the array
        vector<int> left(length, -1);  // Create a vector to store indices of previous less element
        vector<int> right(length, length);  // Create a vector to store indices of next less element
        stack<int> stk;  // Stack to help find previous and next less elements

        // Finding previous less element for each index
        for (int i = 0; i < length; ++i) {
            while (!stk.empty() && nums[stk.top()] >= nums[i]) {
                stk.pop();  // Pop elements that are greater or equal to current element
            }
            if (!stk.empty()) {
                left[i] = stk.top();  // Store the index of the previous less element
            }
            stk.push(i);  // Push the current index onto the stack
        }

        // Clear stack for reuse
        stk = stack<int>();

        // Finding next less element for each index
        for (int i = length - 1; i >= 0; --i) {
            while (!stk.empty() && nums[stk.top()] > nums[i]) {
                stk.pop();  // Pop elements that are strictly greater than current element
            }
            if (!stk.empty()) {
                right[i] = stk.top();  // Store the index of the next less element
            }
            stk.push(i);  // Push the current index onto the stack
        }

        ll sum = 0;  // Initialize the sum of minimum elements in all subarrays

        // Calculating the contribution of each element to the overall sum
        for (int i = 0; i < length; ++i) {
            sum += static_cast<ll>(i - left[i]) * (right[i] - i) * nums[i] % MOD;
            sum %= MOD;  // Apply modulus to keep the sum within range
        }

        return sum;  // Return the sum of minimums of all subarrays
    }
};
```

Typescript Solution

```typescript
function sumSubarrayMins(arr: number[]): number {
    const n = arr.length; // Get the length of the array

    // Helper function to get the element at index i or return a sentinel value for boundaries
    function getElement(i: number): number {
        if (i === -1 || i === n) return Number.MIN_SAFE_INTEGER; // Using safe min value for bounds
        return arr[i];
    }

    let answer = 0; // Initialize accumulator for the answer
    const MOD = 1e9 + 7; // The modulo value to prevent integer overflow
    let stack: number[] = []; // Initialize an empty stack to store indices

    // Iterate through all elements including boundaries
    for (let i = -1; i <= n; i++) {
        // While there are elements on the stack and the current element is smaller than the last
        // on the stack, process the stack and update the answer.
        while (stack.length && getElement(stack[0]) > getElement(i)) {
            const index = stack.shift()!; // Remove the top element of the stack
            // Calculate the contribution of the subarrays where arr[index] is the minimum
            // and add it to the answer
            answer = (answer + arr[index] * (index - stack[0]) * (i - index)) % MOD;
        }
        // Push the current index onto the stack
        stack.unshift(i);
    }

    return answer; // Return the final answer
}

// Note: The stack is being used to maintain a list of indices in non-decreasing order.
// By ensuring this ordering, we can efficiently find the previous and next smaller elements
// for every element in the array, which is essential for finding the minimum of each subarray.
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, which corresponds to the reference answer. Here's a breakdown of the main operations and their complexities:

1. Initializing the `left` and `right` arrays takes `O(n)` time as it fills them with default values based on the length of the array `arr`.
2. The first for-loop to populate the `left` array processes each element in `arr` once, resulting in `O(n)` time complexity. Even though there's a while-loop inside, it won't lead to a complexity higher than `O(n)` because elements are only added to and removed from the stack once.
3. The second for-loop to populate the `right` array also runs in `O(n)` time for the same reasons as the first loop.
4. Finally, the sum computation with a list comprehension operates over each index once, yielding a time complexity of `O(n)`.

Combining these operations, which all run sequentially and independently, the total time complexity remains `O(n)`.

Space Complexity

The space complexity of the code is `O(n)`:

1. Two additional arrays `left` and `right` of size `n` are created, contributing `2n` to the space complexity.
2. A stack is used, which, in the worst case, could also store up to `n` elements. However, the stack is reused and not stored in memory all at once. Each element is pushed and popped once.
3. The list comprehension does not create a new list; it merely iterates over the array to compute the sum, so it does not add to the space complexity.

As none of the auxiliary data structures' sizes exceed `n`, the total space complexity is `O(n)`.