1508. Range Sum of Sorted Subarray Sums

Medium Array Two Pointers Binary Search Sorting Leetcode Link

Problem Description

The given problem revolves around finding the sum of a specific subset of a sorted array that contains sums of all non-empty continuous subarrays of the given array `nums`. The problem requires three inputs:

1. `nums` - an array of `n` positive integers.
2. `left` - the starting index from which to sum (one-indexed).
3. `right` - the ending index up to which to sum (one-indexed).

Our objective is to compute the sum of all non-empty continuous subarrays, sort these sums, and then calculate the sum of the elements from the `left` to `right` indices in the sorted array. The final answer should be returned modulo `10^9 + 7` to handle very large numbers which could cause integer overflow situations.

To clarify:

- A *non-empty continuous subarray* is a sequence of one or more consecutive elements from the array.
- *Sorting* these sums means arranging them in non-decreasing order.
- *Summing from index `left` to index `right`* implies adding all elements of the sorted sums array starting at index `left-1` and ending at index `right-1` (since the problem statement indexes from 1, but arrays in most programming languages, including Python, are 0-indexed).

Intuition

The brute force approach to this problem is to generate all possible continuous subarrays of `nums`, calculate their sums, and then sort these sums. This can be done by using two nested loops:

1. The outer loop goes through each element of `nums` from which a subarray can start.
2. The inner loop extends the subarray from the starting element from the outer loop to the end of `nums`, calculating the sum for each extension.

This generates an array `arr` of sums for each subarray. Once we have this array, we sort it. With the sorted array, we now only need to sum the numbers from `left-1` to `right-1` since we're using Python's zero-indexing (the original problem indexes from 1).

This solution is intuitive but not optimized. It can solve the problem as long as the `nums` array is not too large since the time complexity would otherwise become prohibitive. For larger arrays, an optimized approach should be considered involving more advanced techniques such as prefix sums, heaps, or binary search. However, for the provided solution, we stick to the intuitive brute force method, being mindful of its limitations.

Solution Approach

The provided solution iterates through each possible subarray of the given array `nums` and maintains a running sum. To achieve this, the following steps are implemented:

1. Initialize an empty list `arr` that will hold the sums of all non-empty continuous subarrays.

2. Use a nested loop to iterate through all possible subarrays:

    - The outer loop, controlled by variable `i`, goes from `0` to `n-1`, where `i` represents the starting index of a subarray.
    - For each `i`, initialize a sum variable `s` to `0`, to calculate the sum of the subarray starting at index `i`.
    - The inner loop, controlled by variable `j`, goes from `i` to `n-1`. In each iteration of this loop, add the value `nums[j]` to `s`. This effectively extends the subarray by one element in each iteration.
    - After adding `nums[j]` to `s`, append the new sum to `arr`.

3. Once all sums are calculated, sort the list `arr` with the `.sort()` method. This arranges all the subarray sums in non-decreasing order.

4. Calculate the sum of the elements from index `left-1` to `right-1` in the sorted list `arr`. This is done with the slice notation `arr[left - 1 : right]`.

5. Compute the result modulo `10**9 + 7` to get the output within the specified range and handle any potential integer overflow. This is important since the sum of subarray sums could be very large.

6. Return the result.

In summary, the approach uses:

- **Brute force algorithm**: to generate all subarray sums.
- **Sorting**: to arrange the sums in non-decreasing order.
- **Prefix sum technique**: by maintaining a running sum `s` as we iterate through the array.
- **Modular arithmetic**: to ensure the final sum stays within the specified limits.

The time complexity of this approach is `O(n^2)` for generating the subarray sums and `O((n*(n+1)/2) * log(n*(n+1)/2))` for sorting, where `n` is the length of the `nums` array. The space complexity is `O(n*(n+1)/2)` since we are storing the sum of each possible subarray in `arr`.

Example Walkthrough

To illustrate the solution approach, let's consider a small example:

Suppose we have an array `nums = [1, 2, 3]`, with `left = 2` and `right = 3`. We need to find the sum of the sorted subarray sums from the `left` to `right` indices, inclusive.

1. **Initialize `arr`**: We start with an empty list `arr` to hold the sums of all non-empty continuous subarrays.

2. **Iterate using nested loops**:

    - The outer loop runs with `i` from `0 to 2` (the length of `nums` minus one).
    - The inner loop starts with `j` at `i`, with `s` initialized to `0` each time the outer loop starts.

Iteration breakdown:

- For `i = 0`:
    - `j = 0` : `s = 0 + nums[0]` (add `1`), `arr` becomes `[1]`.
    - `j = 1` : `s = 1 + nums[1]` (add `2`), `arr` becomes `[1, 3]`.
    - `j = 2` : `s = 3 + nums[2]` (add `3`), `arr` becomes `[1, 3, 6]`.
- For `i = 1`:
    - `j = 1` : `s = 0 + nums[1]` (add `2`), `arr` becomes `[1, 3, 6, 2]`.
    - `j = 2` : `s = 2 + nums[2]` (add `3`), `arr` becomes `[1, 3, 6, 2, 5]`.
- For `i = 2`:
    - `j = 2` : `s = 0 + nums[2]` (add `3`), `arr` becomes `[1, 3, 6, 2, 5, 3]`.

3. **Sorting `arr`**: We sort `arr`, resulting in `[1, 2, 3, 3, 5, 6]`.

4. **Calculate the result**: We sum the elements from `left-1` to `right-1`, which corresponds to the second (`index 1`) and third (`index 2`) elements in the sorted `arr`. This gives us the sum of `2 + 3 = 5`.

5. **Modular arithmetic**: The final step is to take the result modulo `10**9 + 7`. Since `5` is already less than `10**9 + 7`, the answer remains `5`.

The method returns the result `5`, which is the sum of the second and third smallest sums of non-empty continuous subarrays of `nums`. In this case, the sums of these subarrays are `2` for the subarray `[2]` and `3` for the subarrays `[3]` and `[1, 2]`. Since `[3]` appears twice, we only count it once in the specified range.

This walkthrough summarizes the steps in the provided solution to compute the sum of subarray sums in a given range after sorting them, thereby displaying the complete process from initialization through to obtaining the final result.

Python Solution

```python
from typing import List

class Solution:
    def rangeSum(self, nums: List[int], n: int, left: int, right: int) -> int:
        # Initialize an array to store the sum of contiguous subarrays
        subarray_sums = []

        # Calculate the sum of every contiguous subarray and store it into subarray_sums
        for i in range(n):
            current_sum = 0
            for j in range(i, n):
                current_sum += nums[j]  # Add the current element to the sum
                subarray_sums.append(current_sum)  # Append the current sum to the list

        # Sort the array of subarray sums in non-decreasing order
        subarray_sums.sort()

        # Define the modulus value to prevent integer overflow issues
        mod = 10**9 + 7

        # Compute the sum of the elements from the 'left' to 'right' indices
        # Note: The '-1' adjustment is required because list indices in Python are 0-based
        range_sum = sum(subarray_sums[left - 1 : right]) % mod

        # Return the computed sum modulo 10^9 + 7
        return range_sum

```

Java Solution

```java
import java.util.Arrays; // Import Arrays class for sorting

class Solution {

    // This method calculates the sum of values within a given range of subarray sums from nums array
    public int rangeSum(int[] nums, int n, int left, int right) {
        // Calculate the total number of subarray sums
        int totalSubarrays = n * (n + 1) / 2;
        // Initialize an array to store all possible subarray sums
        int[] subarraySums = new int[totalSubarrays];

        int index = 0; // Index to insert the next sum into subarraySums
        // Loop over nums array to define starting point of subarray
        for (int i = 0; i < n; ++i) {
            int currentSum = 0; // Holds the temporary sum of the current subarray
            // Loop over nums array to create subarrays starting at index i
            for (int j = i; j < n; ++j) {
                currentSum += nums[j]; // Add the current number to the current subarray sum
                subarraySums[index++] = currentSum; // Store the current subarray sum and increment index
            }
        }

        // Sort the array of subarray sums
        Arrays.sort(subarraySums);

        int result = 0; // Initialize the result to 0
        final int mod = (int) 1e9 + 7; // Modulo value to prevent integer overflow
        // Add the values from position "left" to "right" in the sorted subarray sums
        for (int i = left - 1; i < right; ++i) {
            result = (result + subarraySums[i]) % mod;
        }

        return result; // Return the computed sum
    }
}

```

C++ Solution

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

class Solution {
public:
    int rangeSum(vector<int>& nums, int n, int left, int right) {
        // Create an array to store the sums of all subarrays, with size based on the number of possible subarrays
        vector<int> subarraySums(n * (n + 1) / 2);
        int k = 0; // Index for inserting into subarraySums

        // Calculate the sum of all possible subarrays
        for (int start = 0; start < n; ++start) {
            int currentSum = 0; // Stores the sum of the current subarray
            for (int end = start; end < n; ++end) {
                currentSum += nums[end]; // Add the next element to the currentSum
                subarraySums[k++] = currentSum; // Store the sum of the subarray
            }
        }

        // Sort the sums of the subarrays
        sort(subarraySums.begin(), subarraySums.end());

        int answer = 0; // Variable to store the final answer
        const int mod = 1e9 + 7; // The modulo value

        // Calculate the sum of the subarray sums between indices left-1 and right-1 (inclusive)
        for (int i = left - 1; i < right; ++i) {
            answer = (answer + subarraySums[i]) % mod; // Aggregate the sum modulo mod
        }

        return answer; // Return the final calculated sum
    }
};

```

Typescript Solution

```typescript
function rangeSum(nums: number[], n: number, left: number, right: number): number {
    // Calculate the number of possible subarrays and initialize an array to store their sums
    let subarraySums: number[] = new Array(n * (n + 1) / 2);
    let index = 0; // Index for inserting into subarraySums

    // Calculate the sum of all possible subarrays
    for (let start = 0; start < n; ++start) {
        let currentSum = 0; // Stores the sum of the current subarray
        for (let end = start; end < n; ++end) {
            currentSum += nums[end]; // Add the next element to the current sum
            subarraySums[index++] = currentSum; // Store the sum of the subarray
        }
    }

    // Sort the sums of the subarrays
    subarraySums.sort((a, b) => a - b);

    let answer = 0; // Variable to store the final answer
    const MOD = 1e9 + 7; // The modulo value

    // Calculate the sum of the subarray sums between indices left-1 and right-1 (inclusive)
    for (let i = left - 1; i < right; ++i) {
        answer = (answer + subarraySums[i]) % MOD; // Aggregate the sum modulo MOD
    }

    return answer; // Return the final calculated sum
}

```

Time and Space Complexity

The given Python code computes the range sum of all possible contiguous subarrays sorted in a non-decreasing order and then returns the sum of the subarray values from the `left` to `right` indices (1-indexed). Here is the complexity analysis:

- **Time Complexity:**
    - The outer loop runs `n` times, where `n` is the length of the input list `nums`.
    - The inner loop runs `n - i` times for each iteration of the outer loop, summing elements and appending the sum to the array `arr`. When `i` is 0, the inner loop will run `n` times; when `i` is `n-1`, it will run 1 time. On average, it will run `n/2` times.
    - The `sort` operation on the array `arr` of size `n * (n + 1) / 2` (the total number of subarrays) has a time complexity of `O(n^2 * log(n^2))` which simplifies to `O(n^2 * log(n))`.
    - The `sum` operation over `arr[left - 1 : right]` can be considered `O(k)`, where `k` is the difference between `right` and `left`. However, this is negligible compared to the sorting complexity.

    Combining these, the overall time complexity is dominated by the sort, resulting in `O(n^2 * log(n))`.

- **Space Complexity:**
    - The space used by the array `arr`, which stores the `n * (n + 1) / 2` sum values, dominates the space complexity. This results in `O(n^2)`.
    - No other significant space-consuming structures or recursive calls that would impact the overall space complexity.

Therefore, the space complexity of the code is `O(n^2)` and the time complexity is `O(n^2 * log(n))`.