1031. Maximum Sum of Two Non-Overlapping Subarrays

Medium Array Dynamic Programming Sliding Window Leetcode Link

Problem Description

Given an array `nums` of integers and two distinct integer values `firstLen` and `secondLen`, the task is to find the maximum sum that can be achieved by taking two non-overlapping subarrays from `nums`, with one subarray having a length of `firstLen` and the other having a length of `secondLen`. A subarray is defined as a sequence of elements from the array that are contiguous (i.e., no gaps in between). It is important to note that the subarray with length `firstLen` can either come before or after the subarray with length `secondLen`, but they cannot overlap.

Intuition

To solve this problem, an effective idea is to utilize the concept of prefix sums to quickly calculate the sum of elements in any subarray of the given `nums` array. By using the prefix sums, you can determine the sum of elements in constant time, rather than recalculating it every time by adding up elements iteratively.

Here's the intuition broken down into steps:

1. **Calculate prefix sums:** First, we need to create an array of prefix sums `s` from the input array `nums`, which holds the sum of the elements from the start up to the current index.

2. **Initialize variables:** We then define two variables `ans` to store the maximum sum found so far and `t` to keep track of the maximum sum of subarrays of a particular length as we traverse the array.

3. **Find Maximum for each configuration:**

   - Start by considering subarrays of length `firstLen` and then move on to subarrays of length `secondLen`. As we iterate through the array, we calculate the maximum sum of a `firstLen` subarray ending at the current index and store it in `t`.
   - Then we use this to calculate and update `ans` by adding the sum of the following `secondLen` subarray.
   - We ensure that at each step, the chosen subarrays do not overlap by controlling the indices and lengths properly.
4. **Repeat the process in reverse:** To ensure we are not missing out on any configuration (since the `firstLen` subarray can appear before or after the `secondLen` subarray), we reverse the lengths and repeat the procedure.

5. **Return the result:** The maximum of all calculated sums is stored in `ans`, which we return as the final answer.

By iterating over each possible starting point for the `firstLen` and `secondLen` subarrays and efficiently calculating sums using the prefix array, we find the maximum sum of two non-overlapping subarrays of designated lengths.

Solution Approach

The solution is built around the efficient use of a prefix sum array and two traversal patterns to evaluate all possible configurations of the two required subarrays.

Here are the steps involved in the implementation:

1. **Prefix Sum Array:** A prefix sum array `s` is constructed from the input array `nums` using `list(accumulate(nums, initial=0))`. This function call essentially generates a new list where each element at index `i` represents the sum of the `nums` array up to that index.

2. **Traverse and Compute for `firstLen` and `secondLen`:** The algorithm starts off with two for loop constructs, each responsible for handling one of the two configurations:

   - The first for loop starts iterating after `firstLen` to leave room for the first subarray. Inside this loop, `t` is calculated as the maximum sum of the `firstLen` subarray ending at the current index i.
   - It immediately computes the sum of the next `secondLen` subarray and updates the answer `ans` if needed, by adding the sum of the current `firstLen` subarray (`t`) and the sum of the consecutive `secondLen` subarray.
3. **Variable `t` and `ans`:** The variable `t` tracks the maximum sum of a subarray of length `firstLen` found up to the current position in the iteration (essentially, it holds the best answer found so far for the left side). The variable `ans` accumulates the maximum combined sum of two non-overlapping subarrays, comparing the sum of the current subarray of `firstLen` plus the sum of the non-overlapping subarray of `secondLen`.

4. **Repeat the Process for Reversed Lengths:** After the first pass is completed, the same process is repeated, with the roles of `firstLen` and `secondLen` reversed. This ensures that all possible positions of `firstLen` and `secondLen` subarrays are evaluated.

5. **Checking for Overlapping:** While updating `ans`, care is taken to ensure that the subarrays do not overlap by controlling the sequence of index increments and subarray length considerations.

6. **Return the Maximum Sum:** After both traversals, the variable `ans` holds the maximum sum possible without overlap, which is then returned.

By separately handling the cases for which subarray comes first, the function ensures it examines all possible configurations while efficiently computing sums using the prefix sum array, thus arriving at the correct maximum sum of two non-overlapping subarrays of given lengths.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Consider the array `nums = [3, 5, 2, 1, 7, 3]`, with `firstLen = 2` and `secondLen = 3`.

Step 1: Prefix Sum Array

Construct a prefix sum array `s` from `nums`.

- Original `nums` array: [3, 5, 2, 1, 7, 3]
- Prefix sum array `s`: [0, 3, 8, 10, 11, 18, 21]

The element at index `i` in the prefix sum array represents the sum of all elements in `nums` up to index `i-1`.

Step 2: Traverse and Compute for firstLen, then secondLen

- Initialize `ans` to -infinity (or a very small number) and `t` to 0.
- Start the first for loop after `firstLen` (index is 2 in this case).

Step 3: Variable t and ans

- At index 2 (i = 2), we ignore because we cannot form both subarrays.
- At index 3 (i = 3), `t = max(t, s[3] - s[3 - firstLen])` which is `max(0, 10 - 3)` so `t = 7`.

We then calculate the sum of the next `secondLen` subarray: `s[i + secondLen] - s[i]` which is `s[6] - s[3]` so `18 - 10 = 8`. Since `8 (secondLen subarray sum) + 7 (firstLen subarray sum) = 15` is greater than `ans`, we update `ans` to 15.

Step 4: Repeat for Reversed Lengths

- We reset `t` to 0, and reverse the `firstLen` and `secondLen`.
- Start the next for loop after `secondLen` (index is 3 in this case).
- At index 3 (i = 3), we ignore because we cannot form both subarrays.

Checking for Overlapping

- At index 4 (i = 4), `t = max(t, s[4] - s[4 - secondLen])` which is `max(0, 11 - 3)` so `t = 8`.

We then compute the sum of the next `firstLen` subarray: `s[i + firstLen] - s[i]` which is `s[6] - s[4]` so `21 - 11 = 10`. We add `10 (firstLen subarray sum)` to `8 (secondLen subarray sum)` and get `10 + 8 = 18` which is greater than `ans`, so we update `ans` to 18.

Step 6: Return the Maximum Sum

We have finished evaluating both configurations. The variable `ans` now holds the value `18`, which is the maximum sum of two non-overlapping subarrays for the lengths provided. Thus, we return `18` as the final answer for this example.

Python Solution

```python
from itertools import accumulate

class Solution:
    def max_sum_two_no_overlap(self, nums: List[int], first_len: int, second_len: int) -> int:
        # Determine the total number of elements in nums
        n = len(nums)

        # Create a prefix sum array with an initial value of 0 for easier calculation
        prefix_sums = list(accumulate(nums, initial=0))

        # Initialize the answer and a temporary variable for tracking the max sum of the first array
        max_sum = max_sum_first_array = 0

        # Loop through nums to consider every possible second array starting from index first_len
        i = first_len
        while i + second_len - 1 < n:
            # Find the max sum of the first array ending before the start of the second array
            max_sum_first_array = max(max_sum_first_array, prefix_sums[i] - prefix_sums[i - first_len])
            # Update the max_sum with the best we've seen combining the two arrays so far
            max_sum = max(max_sum, max_sum_first_array + prefix_sums[i + second_len] - prefix_sums[i])
            i += 1

        # Reset the temporary variable for the max sum of first and second arrays
        max_sum_second_array = 0
        # Loop through nums to consider every possible first array starting from index second_len
        i = second_len
        while i + first_len - 1 < n:
            # Find the max sum of the second array ending before the start of the first array
            max_sum_second_array = max(max_sum_second_array, prefix_sums[i] - prefix_sums[i - second_len])
            # Update the max_sum with the best we've seen for the swapped sizes of the two arrays
            max_sum = max(max_sum, max_sum_second_array + prefix_sums[i + first_len] - prefix_sums[i])
            i += 1

        # Return the maximum sum found
        return max_sum

```

Java Solution

```java
class Solution {
    public int maxSumTwoNoOverlap(int[] numbers, int firstLength, int secondLength) {
        // Initialize the length of the array
        int arrayLength = numbers.length;
        // Create a prefix sum array with an additional 0 at the beginning
        int[] prefixSums = new int[arrayLength + 1];
        // Calculate prefix sums
        for (int i = 0; i < arrayLength; ++i) {
            prefixSums[i + 1] = prefixSums[i] + numbers[i];
        }

        // Initialize the answer to be the maximum sum we are looking for
        int maxSum = 0;

        // First scenario: firstLength subarray is before secondLength subarray
        // Loop from firstLength up to the point where a contiguous secondLength subarray can fit
        for (int i = firstLength, tempMax = 0; i + secondLength - 1 < arrayLength; ++i) {
            // Get the maximum sum of any firstLength subarray up to the current index
            tempMax = Math.max(tempMax, prefixSums[i] - prefixSums[i - firstLength]);
            // Update the maxSum with the sum of the maximum firstLength subarray and the contiguous secondLength subarray
            maxSum = Math.max(maxSum, tempMax + prefixSums[i + secondLength] - prefixSums[i]);
        }

        // Second scenario: secondLength subarray is before firstLength subarray
        // Loop from secondLength up to the point where a contiguous firstLength subarray can fit
        for (int i = secondLength, tempMax = 0; i + firstLength - 1 < arrayLength; ++i) {
            // Get the maximum sum of any secondLength subarray up to the current index
            tempMax = Math.max(tempMax, prefixSums[i] - prefixSums[i - secondLength]);
            // Update the maxSum with the sum of the maximum secondLength subarray and the contiguous firstLength subarray
            maxSum = Math.max(maxSum, tempMax + prefixSums[i + firstLength] - prefixSums[i]);
        }

        // Return the maximum sum found for both scenarios
        return maxSum;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For std::max

using std::vector;
using std::max;

class Solution {
public:
    int maxSumTwoNoOverlap(vector<int>& nums, int L, int M) {
        int n = nums.size();
        vector<int> prefixSum(n + 1, 0);
        // Calculate prefix sums
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }

        int maxSum = 0;
        int maxL = 0; // To store max sum of subarray with length L

        // Find max sum for two non-overlapping subarrays
        // where first subarray has length L and second has length M
        for (int i = L; i + M - 1 < n; ++i) {
            maxL = max(maxL, prefixSum[i] - prefixSum[i - L]);
            maxSum = max(maxSum, maxL + prefixSum[i + M] - prefixSum[i]);
        }

        int maxM = 0; // To store max sum of subarray with length M

        // Same as above, but first subarray has length M and second has length L
        for (int i = M; i + L - 1 < n; ++i) {
            maxM = max(maxM, prefixSum[i] - prefixSum[i - M]);
            maxSum = max(maxSum, maxM + prefixSum[i + L] - prefixSum[i]);
        }

        // Return the max possible sum of two non-overlapping subarrays
        return maxSum;
    }
};

```

Typescript Solution

```typescript
// The `nums` array stores the integers, `L` and `M` are the lengths of the subarrays
function maxSumTwoNoOverlap(nums: number[], L: number, M: number): number {
    const n: number = nums.length;
    const prefixSum: number[] = new Array(n + 1).fill(0);

    // Calculate prefix sums
    for (let i = 0; i < n; ++i) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    let maxSum: number = 0; // Max sum of two non-overlapping subarrays
    let maxL: number = 0; // Max sum of subarray with length L

    // First loop: fixing the first subarray with length L and finding optimal M
    for (let i = L; i + M <= n; ++i) {
        maxL = Math.max(maxL, prefixSum[i] - prefixSum[i - L]);
        maxSum = Math.max(maxSum, maxL + prefixSum[i + M] - prefixSum[i]);
    }

    let maxM: number = 0; // Max sum of subarray with length M

    // Second loop: fixing the first subarray with length M and finding optimal L
    for (let i = M; i + L <= n; ++i) {
        maxM = Math.max(maxM, prefixSum[i] - prefixSum[i - M]);
        maxSum = Math.max(maxSum, maxM + prefixSum[i + L] - prefixSum[i]);
    }

    // Return the max possible sum of two non-overlapping subarrays
    return maxSum;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `nums` list. This is because the code iterates over the list twice with while-loops. In each iteration, it performs a constant number of operations (addition, subtraction, and comparison).

The space complexity of the code is `O(n)`, due to the additional list `s` that is created with the `accumulate` function to store the prefix sums of the `nums` list. The size of the `s` list is directly proportional to the size of the `nums` list.