1546. Maximum Number of Non-Overlapping Subarrays With Sum Equals Target

Medium Greedy Array Hash Table Prefix Sum

Leetcode Link

Problem Description

Given an array called `nums` and an integer called `target`, the task is to find the maximum number of distinct, non-overlapping subarrays where each subarray adds up to the given `target`. A subarray is a contiguous part of the array, and it must be non-empty.

Intuition

The solution approach revolves around iterating through the array and keeping track of the cumulative sum of the elements. We want to know if at any point the cumulative sum minus the `target` has been previously seen. If it has, that means we have found a subarray that sums up to `target`.

The intuition is based on the following thought process:

1. Initialize a variable to store the cumulative sum `s` and a set `seen` to keep track of all the different sums we have encountered, initializing the set with a 0 to handle cases where a subarray starting from the first element meets the `target`.
2. Iterate over the array `nums`, adding each number to our cumulative sum `s`.
3. For each new sum, we check if `s - target` exists in the `seen` set. If it does, it means we've found a non-overlapping subarray that sums up to `target` because we had a subarray previous to this whose sum was `s - target`, making the sum between that point and the current index exactly `target`.
4. Every time we find such a subarray, we increment our answer `ans`, break the while loop to not to consider overlapping subarrays, and reset our set `seen` and sum `s` for the next iteration.
5. We continue this process for each element in `nums`.

This approach ensures that we're always looking at non-overlapping subarrays by resetting the set and cumulative sum after each found subarray. The counter `ans` is our final answer, representing the maximum number of subarrays summing up to `target`.

Solution Approach

The solution uses a `while` loop to iterate over the elements in the array `nums`, and a set named `seen` to keep track of the cumulative sums during the iteration of subarrays.

Here's the step-by-step breakdown of the algorithm:

1. Initialize two pointers `i` and `n`. Pointer `i` is used to traverse the array, and `n` holds the length of the array for bounds checking.
2. A counter `ans` is initialized to 0. This counter tracks the number of non-overlapping subarrays found that sum up to `target`.
3. The outer `while` loop continues as long as `i < n`, ensuring we go through each element.
4. Inside the outer loop, we initialize a sum `s` to 0 and a set `seen` with the initial element being 0. The sum `s` will keep track of the cumulative sum of the elements starting from index `i`, and the set `seen` keeps track of all previous cumulative sums.
5. The inner `while` loop also continues as long as `i < n`, which goes over the elements from the current starting point:
   - We add the current element `nums[i]` to the cumulative sum `s`.
   - Check if `s - target` is in the set `seen`. If it is, it means we have found a valid subarray because the difference between the current cumulative sum and the `target` is a sum we saw earlier. Since we ensure to add only non-overlapping sums to `seen`, finding `s - target` guarantees a non-overlapping subarray.
   - If we found a valid subarray, we increment `ans` by 1, which is our count for non-overlapping subarrays summing up to `target`. We then break out of the inner `while` loop to start looking for the next valid subarray, ensuring non-overlap.
   - If we did not find a valid subarray yet, we proceed to the next element by incrementing `i` and adding the new sum `s` to the set `seen`.
6. As soon as we exit the inner `while` loop (either due to finding a valid subarray or reaching the end of the array), we increment `i` to look for the next starting point of a potential subarray.

The process repeats until we have exhausted all elements in the array `nums`. Finally, the variable `ans` holds the maximum number of non-overlapping subarrays with a sum equal to `target`, which is returned from the function.

Using a set to track cumulative sums is a clever way to check for the presence of a sum in constant time, which keeps the solution efficient. Resetting `s` and `seen` after finding a subarray ensures we only count non-overlapping subarrays, adhering to the problem requirements.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following array and target:

```
nums = [1, 2, 3, 4, 5]
target = 5
```

We need to find the maximum number of distinct, non-overlapping subarrays where each subarray sums up to the target value of 5.

1. We initialize `i` to 0, `n` to the length of `nums` which is 5, and `ans` to 0.
2. We start our outer while loop with `i < n`. Since `i = 0` and `n = 5`, we enter the loop.
3. We initialize our cumulative sum `s` to 0 and our set `seen` with an initial element of 0.
4. Now, we enter the inner while loop. At `i = 0`, `nums[i] = 1`. We add this to our sum `s`, so `s = 1`. We then add `s` to `seen`, so `seen = {0, 1}`.
5. We move to the next element `i = 1`, `nums[i] = 2`. Our new sum `s = 3`. This is not in `seen` after subtracting `target`, so we add it to `seen`, which now becomes `{0, 1, 3}`.
6. Next up is `i = 2`, `nums[i] = 3`. Adding this to our sum gives us `s = 6`. Now, `s - target = 1`, which is in `seen`. That means we found a valid subarray `[1, 2, 3]` that adds up to our target `5`.
7. We increment `ans` by 1, to reflect the subarray we found. We break the inner while loop, reset our sum `s` to 0, and clear `seen` to `{0}` to look for further non-overlapping subarrays.
8. We increment `i` outside of the inner while loop to move to the next potential starting point of a subarray. Since we broke the inner loop at `i = 3`, which corresponds to the fourth element `nums[3] = 4`. We now start from there.
9. We repeat steps 4 to 7. Our cumulative sum `s` is incremented by `nums[3]`, so `s = 4`. And `seen` is updated to `{0, 4}`.
10. Moving to `i = 4`, `nums[i] = 5`. Now, `s = 9`. However, `s - target = 4` which is in the set `seen`. This means we've found another valid subarray `[4, 5]`.
11. We increment `ans` by 1 again and break the inner loop. Now, `ans = 2`, which reflects the two distinct non-overlapping subarrays `[1, 2, 3]` and `[4, 5]` that sum up to `target`.
12. There are no more elements to process, as we've reached the end of `nums`.

The final answer, held by `ans`, is 2, representing the maximum number of non-overlapping subarrays with a sum equal to `target` in the given `nums` array.

Python Solution

```python
class Solution:
    def maxNonOverlapping(self, nums: List[int], target: int) -> int:
        curr_index, nums_length = 0, len(nums)  # Initializing the current index and the total length of the array
        non_overlapping_count = 0  # To keep track of the count of non-overlapping subarrays

        # Iterate through the array until the current index is less than the length of the array
        while curr_index < nums_length:
            cumulative_sum = 0  # Initialize the cumulative sum for the current subarray
            seen_sum = {0}  # Set to store cumulative sums which are useful for identifying if a subarray with the target sum exists

            # Continue in the inner while-loop to find a subarray that sums to the target
            while curr_index < nums_length:
                cumulative_sum += nums[curr_index]  # Update the cumulative sum

                # If the difference between the current cumulative sum and the target is in seen_sums,
                # we have found a subarray that sums up to the target
                if cumulative_sum - target in seen_sums:
                    non_overlapping_count += 1  # Increment the count of non-overlapping subarrays
                    break  # Exit the inner while-loop to start looking for the next subarray

                # If we haven't found a valid subarray yet, update the current index and add the current sum to seen_sums
                curr_index += 1
                seen_sums.add(cumulative_sum)

            # Move to the next index to start a new subarray scan
            curr_index += 1

        # Return the total number of non-overlapping subarrays that sum up to the target
        return non_overlapping_count
```

Java Solution

```java
class Solution {
    public int maxNonOverlapping(int[] nums, int target) {
        int currentIndex = 0; // Initialize the current index to start from the beginning of the array.
        int totalSubarrays = 0; // This will keep track of the count of non-overlapping subarrays that sum up to 'target'.
        int arrayLength = nums.length; // Get the length of the input array 'nums'.

        // Iterate over the array until we reach the end.
        while (currentIndex < arrayLength) {
            int currentSum = 0; // Initialize the sum of the current subarray being evaluated.
            HashSet<Integer> seenSums = new HashSet<>(); // Use a HashSet to store the unique sums encountered.
            seenSums.add(0); // Add zero to handle the case when a subarray starts from the first element.

            // Keep scanning through the array until the end.
            while (currentIndex < arrayLength) {
                currentSum += nums[currentIndex]; // Add the current element to the current sum.

                // If the set contains the current sum minus the target, we've found a valid subarray.
                if (seenSums.contains(currentSum - target)) {
                    totalSubarrays++; // Increment the count of valid subarrays.
                    break; // Break to start looking for the next non-overlapping subarray.
                }
                seenSums.add(currentSum); // Add the current sum to the set of seen sums.
                currentIndex++; // Move to the next element in the array.
            }
            currentIndex++; // Increment to skip the start of the next subarray after finding a valid subarray.
        }

        return totalSubarrays; // Return the total number of non-overlapping subarrays with sum equal to 'target'.
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>
using namespace std;

class Solution {
public:
    // Function to find the maximum number of non-overlapping subarrays that sum to a target value
    int maxNonOverlapping(vector<int>& nums, int target) {
        int index = 0; // Start index for checking subarrays
        int n = nums.size(); // Length of the input array
        int answer = 0; // Initialization of count of maximum non-overlapping subarrays

        // Iterate over the array to find all possible non-overlapping subarrays
        while (index < n) {
            int currentSum = 0; // Initialize the sum of the current subarray
            unordered_set<int> seenSums; // Track all unique sums encountered within the current window
            seenSums.insert(0); // Insert 0 to handle cases where a subarray starts from the first element

            // Continue to expand the window until the end of the array is reached
            while (index < n) {
                currentSum += nums[index]; // Update current sum

                // If the sum minus the target has been seen before, we've found a target subarray
                if (seenSums.count(currentSum - target)) {
                    answer++; // Increment the count for the answer
                    break; // Start looking for the next subarray
                }

                // Insert the current sum into the set and move to the next element
                seenSums.insert(currentSum);
                index++;
            }

            // Skip the next index after a valid subarray is found to ensure non-overlapping
            index++;
        }

        // Return the total count of non-overlapping subarrays summing to the target
        return answer;
    }
};

int main() {
    // Example usage:
    Solution solution;
    vector<int> nums = {1,1,1,1,1};
    int target = 2;
    int maxSubarrays = solution.maxNonOverlapping(nums, target);
    // maxSubarrays should be 2 for this input
    return 0;
}
```

Typescript Solution

```typescript
function maxNonOverlapping(nums: number[], target: number): number {
    let index = 0; // Start index for checking subarrays
    const n = nums.length; // Length of the input array
    let answer = 0; // Initialization of count of maximum non-overlapping subarrays

    // Iterate over the array to find all possible non-overlapping subarrays
    while (index < n) {
        let currentSum = 0; // Initialize the sum of the current subarray
        const seenSums = new Set<number>(); // Track all unique sums encountered within the current window
        seenSums.add(0); // Insert 0 to handle cases where a subarray starts from the first element

        // Continue to expand the window until the end of the array is reached
        while (index < n) {
            currentSum += nums[index]; // Update current sum

            // If the sum minus the target has been seen before, we've found a target subarray
            if (seenSums.has(currentSum - target)) {
                answer++; // Increment the count for the answer
                break; // Start looking for the next subarray
            }

            // Insert the current sum into the set and move to the next element
            seenSums.add(currentSum);
            index++;
        }

        // Skip the next index after a valid subarray is found to ensure non-overlapping
        index++;
    }

    // Return the total count of non-overlapping subarrays summing to the target
    return answer;
}

// Example usage:
const nums = [1, 1, 1, 1, 1];
const target = 2;
const maxSubarrays = maxNonOverlapping(nums, target);
// maxSubarrays should be 2 for this input
```

Time and Space Complexity

Time Complexity

The time complexity of this code is $O(n)$, where n is the length of the `nums` array. This linear time complexity arises from the fact that the code iterates over the array elements at most twice. Once for the outer `while` loop, and at most once more within the inner `while` loop before a matching subarray sum is found and the loop is broken. Once the code finds a matching sum, it immediately breaks out of the inner loop and skips to the next index after the end of the current subarray. Thus, each element is touched at most twice during the iteration.

Space Complexity

The space complexity of the code is also $O(n)$. The primary contributing factor to the space complexity is the `seen` set, which in the worst-case scenario could store a cumulative sum for each element in the `nums` array if no sums match `s - target`. As a result, in the worst case, this set would store n unique sums, making the space complexity linear with respect to the length of `nums`.