1477. Find Two Non-overlapping Sub-arrays Each With Target Sum

Medium · Array · Hash Table · Binary Search · Dynamic Programming · Sliding Window · Leetcode Link

Problem Description

The problem requires finding two non-overlapping sub-arrays within a given array of integers, `arr`, where each sub-array sums up to a specific value, `target`. To qualify, neither of the sub-arrays should share any elements with the other, effectively meaning they cannot overlap. The goal is to find such pair of sub-arrays where the combined length of both sub-arrays is as small as possible. If it's impossible to find such pair of sub-arrays, the function should return `-1`.

Intuition

The solution to this problem involves dynamic programming and the use of a hashmap to track the prefix sums.

1. The intuition behind the dynamic programming approach is that at any point in the array, we want to know the length of the smallest sub-array ending at that point which sums to `target`. This can help us efficiently update the answer when we find the next sub-array that sums to `target`.
2. To efficiently find if a sub-array sums to `target`, we can use a hashmap to store the sum of all elements up to the current index (`s`), as the key, with the value being the index itself. If `s - target` is in the hashmap, it means there is a valid sub-array ending at the current index which sums to `target`.
3. While iterating through the array, we keep updating our hashmap with the current sum and index, and we also keep track of the smallest sub-array found so far that sums to `target` using an array `f`.
4. Whenever a new sub-array is found (checking if `s - target` exists in the hashmap), we calculate the minimum length of a sub-array that sums to `target` that ends at our current index. Simultaneously, we try to update the global answer, which is the sum of lengths of two such sub-arrays.

The key realization is that we do not need to store the actual sub-arrays. Instead, storing their lengths and endpoints suffices to solve the problem. By maintaining a running minimum length sub-array and utilizing the hashmap to efficiently query the existence of a previous sub-array sum, we can determine the optimal pair of sub-arrays that fulfill the conditions.

Solution Approach

The implementation utilizes a hashmap and a dynamic programming (DP) table. Here is a step-by-step breakdown:

1. **Hashmap for Prefix Sums**:
   - A hashmap `d` is used to store the sum of all elements up to the current index as keys (`s` presents the current sum) and their corresponding indices as values. This helps quickly check if there is a sub-array ending at the current index that sums up to `target`.
2. **Initializing Variables**:
   - `s` is the prefix sum initialized to 0.
   - `n` is the total number of elements in the array `arr`.
   - `f` is an array of size `n+1` initialized to `inf` (infinity). `f[i]` will hold the length of the smallest sub-array ending before or at index `i`, which has a sum of `target`.
   - `ans` is initialized to `inf` and will be used to store the minimum sum of the lengths of the two non-overlapping sub-arrays found so far.
3. **Iterating Through the Array**:
   - Using a for loop, we iterate through the array starting from index 1 (since `f` is 1-indexed to simplify calculations).
   - We update the prefix sum `s` by adding the current element `v`.
4. **Dynamic Programming Table Update**:
   - At each index `i`, the current value of `f[i]` is set to `f[i - 1]`, ensuring that we carry forward the length of the smallest sub-array found so far up to the previous index.
5. **Finding and Updating Sub-arrays**:
   - At each step of the iteration, we check if `s - target` is in the hashmap `d`.
   - If it is, that means we have encountered a sub-array, ending at the current index, which sums up to `target`.
   - We retrieve this sub-array's starting index `j` from the hashmap, and we calculate its length `i - j`.
   - Then we update `f[i]` as the minimum of the current value and the length of this new sub-array, which reflects the smallest sub-array summing to `target` up to index `i`.
6. **Updating the Answer**:
   - While a new valid sub-array is found, we calculate the sum of its length with the smallest length of a sub-array found before it, i.e., `f[j] + i - j`.
   - If this sum is smaller than our current answer, we update the answer `ans`.
7. **Returning the Answer**:
   - After iterating through the entire array, it is possible that no such pair of sub-arrays was found. We check this by comparing `ans` with `n`.
   - If `ans` is still `inf` or greater than `n`, it means no two non-overlapping sub-arrays with the sum `target` were found, and we return `-1`.
   - Otherwise, we return `ans`, which is the minimum sum of the lengths of the required two sub-arrays.

By the end of the loop, we either find the minimum sum of lengths of two non-overlapping sub-arrays that sum up to `target`, or determine that it's not possible to find such sub-arrays and return `-1`.

The use of the DP approach with a hashmap allows the algorithm to run efficiently by preventing repeated scanning of the array to check for sub-arrays that sum to `target`.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the array `arr = [1, 2, 1, 2, 1, 2, 1]` with `target = 3`.

1. **Hashmap for Prefix Sums**: We will use a hashmap `d` to keep track of the prefix sums. Initially, `d` is empty.
2. **Initializing Variables**:
   - `s` starts at 0.
   - `n` is `7`, the total number of elements in `arr`.
   - `f` is an array `[inf, inf, inf, inf, inf, inf, inf, inf]` (1-indexed for a total of `n+1` entries).
   - `ans` is initialized to `inf`.
3. **Iterating Through the Array**: We iterate `i` from 1 to 7, and for each element `v` in `arr`, we update `s`.
4. **Dynamic Programming Table Update**: At each index, `f[i]` is updated to be the minimum length of a valid sub-array up to that point.
5. **Finding and Updating Sub-arrays**:
   - For each `v`, we calculate `s` and check if `s - target` exists in `d`.
   - If it does, we find the length of the current sub-array and update `f[i]` with the minimum.
6. **Updating the Answer**:
   - Each time we find a valid sub-array, we calculate if it can contribute to a smaller `ans`.
7. **Returning the Answer**: At the end, we return `ans` or `-1` if not found.

Now, let's walk through with our array:

- Initial step: `d = {0: 0}` to account for starting from a sum of `0` at index `0`.
- `i = 1`, `arr[1] = 1`. `s = 1`. `f = [inf, inf, inf, inf, inf, inf, inf, inf]`, `ans = inf`. `d = {0: 0, 1: 1}`.
- `i = 2`, `arr[2] = 2`, `s = 3`. We see `s - target = 0` in `d`. Sub-array `[1, 2]` has sum `3`. Update `f[2] = min(inf, 2 - 0) = 2`. `d = {0: 0, 1: 1, 3: 2}`.
- `i = 3`, `arr[3] = 1`. `s = 4`. No change since `s - target = 1` is not a prefix sum we've seen that ended at an index before `i`. `d = {0: 0, 1: 1, 3: 2, 4: 3}`.
- `i = 4`, `arr[4] = 2`. `s = 6`. We find `s-target = 3` in `d`. The sub-array `[1, 2]` happens again. Update `f[4] = min(inf, 4 - 2) = 2`. `ans = min(inf, f[2] + 2) = min(inf, 2 + 2) = 4`. `d = {0: 0, 1: 1, 3: 2, 4: 3, 6: 4}`.
- `i = 5`, `arr[5] = 1`.`s = 7`. We find `s-target = 4` in `d`. Update `f[5] = min(inf, 5 - 3) = 2`. Ans does not update because `f[3]` was `inf`. `d = {0: 0, 1: 1, 3: 2, 4: 3, 6: 4, 7: 5}`.
- `i = 6`, `arr[6] = 2`. `s = 9`. We find `s-target = 6` in `d`. Update `f[6] = min(inf, 6 - 4) = 2`. `ans = min(4, f[4] + 2) = min(4, 2 + 2) = 4`. `d = {0: 0, 1: 1, 3: 2, 4: 3, 6: 4, 7: 5, 9: 6}`.
- `i = 7`, `arr[7] = 1`. `s = 10`. We find `s-target = 7` in `d`. Update `f[7] = min(inf, 7 - 5) = 2`. `ans = min(4, f[5] + 2) = min(4, 2 + 2) = 4`. `d = {0: 0, 1: 1, 3: 2, 4: 3, 6: 4, 7: 5, 9: 6, 10: 7}`.

After completing the iteration, we have `ans = 4`, which corresponds to two non-overlapping sub-arrays `[1, 2]` and `[1, 2]`, both sum up to `3`, with a combined minimum length of `4`. Hence, the function would return `4`.

Python Solution

```python
class Solution:
    def minSumOfLengths(self, arr: List[int], target: int) -> int:
        # Create a dictionary to remember sum of all elements till index i (0-indexed)
        sum_to_index = {0: -1}  # Adjusted for 0-index, starting with sum 0 at index -1
        current_sum = 0  # Current sum of elements
        final_result = float('inf')  # Initialize the final_result with infinity

        # Minimum length subarray ending at i that sums to target
        min_len_subarray = [float('inf')] * len(arr)

        for i, value in enumerate(arr):
            current_sum += value  # Update the current_sum with the current value

            # Update the min_len_subarray for the current position
            if i > 0:
                min_len_subarray[i] = min_len_subarray[i - 1]

            # If the current_sum minus target sum is in sum_to_index...
            if current_sum - target in sum_to_index:
                # Get the start index of subarray ending at current index i
                start_index = sum_to_index[current_sum - target]
                # Calculate the length of this subarray
                current_len = i - start_index
                # Update the minimum length subarray for the current position
                min_len_subarray[i] = min(min_len_subarray[i], current_len)
                # If this is not the first element, consider previous subarrays and update final_result
                if start_index >= 0:
                    final_result = min(final_result, min_len_subarray[start_index] + current_len)

            # Update the sum_to_index with the current_sum at current index i
            sum_to_index[current_sum] = i

        # If final_result was updated and is not infinity, return it, else return -1
        return final_result if final_result != float('inf') else -1

```

Java Solution

```java
class Solution {
    public int minSumOfLengths(int[] arr, int target) {
        // Hash map to store the cumulative sum up to the current index with the index itself
        Map<Integer, Integer> cumSumToIndex = new HashMap<>();
        cumSumToIndex.put(0, 0); // Initialization with sum 0 at index 0

        int n = arr.length; // Length of the array
        int[] minLengths = new int[n + 1]; // Array to store the minimum subarray length ending at i that sums up to target
        final int infinity = 1 << 30; // A very large number treated as infinity
        minLengths[0] = infinity; // Initialize with infinity since there's no subarray ending at index 0

        int currentSum = 0; // Sum of the current subarray
        int answer = infinity; // Initialize the answer with a large number representing infinity

        // Loop through the array to find minimum subarrays
        for (int i = 1; i <= n; ++i) {
            int value = arr[i - 1]; // Value at current index in the array
            currentSum += value; // Update the current sum

            // Copy the minimum subarray length found so far to current position
            minLengths[i] = minLengths[i - 1];

            // If the (currentSum - target) is found before, update the minimum length and answer
            if (cumSumToIndex.containsKey(currentSum - target)) {
                int j = cumSumToIndex.get(currentSum - target); // Previous index where the cumulative sum was currentSum - target
                minLengths[i] = Math.min(minLengths[i], i - j); // Update the minimum length for current index
                answer = Math.min(answer, minLengths[j] + i - j); // Calculate the combined length and update the answer
            }

            // Store the current cumulative sum and corresponding index
            cumSumToIndex.put(currentSum, i);
        }

        // If the answer is still infinity, no such subarrays are found; return -1. Else, return the found answer
        return answer > n ? -1 : answer;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function finds two non-overlapping subarrays which sum to the target and returns
    // the minimum sum of their lengths or -1 if there are no such subarrays.
    int minSumOfLengths(vector<int>& arr, int target) {
        unordered_map<int, int> prefixSumToIndex;
        prefixSumToIndex[0] = -1; // Initialization with a base case

        int prefixSum = 0, arraySize = arr.size();

        // Initialize the array to store the minimum length of a subarray ending at each index i that sums to target.
        vector<int> minLenToEndAt(arraySize + 1, INT_MAX);
        minLenToEndAt[0] = INT_MAX; // No subarray ends before the array starts.

        int minTotalLen = INT_MAX; // This will store the result.

        for (int i = 0; i < arraySize; ++i) {
            prefixSum += arr[i]; // Add the current element to the prefix sum.

            // Update the minimum length for a subarray ending at the current index.
            if (i > 0) {
                minLenToEndAt[i + 1] = minLenToEndAt[i];
            }

            // If the prefix sum needed to achieve the target exists...
            if (prefixSumToIndex.count(prefixSum - target)) {
                int startIndex = prefixSumToIndex[prefixSum - target];
                // Update the minimum length for this index.
                minLenToEndAt[i + 1] = min(minLenToEndAt[i + 1], i - startIndex);

                // If the previous subarray (ending at startIndex) is valid...
                if (minLenToEndAt[startIndex + 1] < INT_MAX) {
                    // Update the minimum total length.
                    minTotalLen = min(minTotalLen, minLenToEndAt[startIndex + 1] + i - startIndex);
                }
            }

            // Update or add the current prefix sum and its ending index to the map.
            prefixSumToIndex[prefixSum] = i;
        }

        // If minTotalLen is not updated, return -1 to indicate no such subarrays exist.
        return minTotalLen == INT_MAX ? -1 : minTotalLen;
    }
};

```

Typescript Solution

```typescript
// Initialize a helper function to find the minimum of two numbers
function min(a: number, b: number): number {
    return a < b ? a : b;
}

// This function finds two non-overlapping subarrays which sum to the target
// and returns the minimum sum of their lengths, or -1 if there are no such
// subarrays.
function minSumOfLengths(arr: number[], target: number): number {
    const prefixSumToIndex: Map<number, number> = new Map();
    prefixSumToIndex.set(0, -1); // Initialization with a base case

    let prefixSum: number = 0;
    const arraySize: number = arr.length;

    // Initialize the array to store the minimum length of a subarray ending at each index i that sums to target.
    const minLenToEndAt: number[] = new Array(arraySize + 1).fill(Number.MAX_SAFE_INTEGER);
    minLenToEndAt[0] = Number.MAX_SAFE_INTEGER; // No subarray ends before the array starts.

    let minTotalLen: number = Number.MAX_SAFE_INTEGER; // This will store the result.

    for (let i = 0; i < arraySize; ++i) {
        prefixSum += arr[i]; // Add the current element to the prefix sum.

        // Update the minimum length for a subarray ending at the current index.
        if (i > 0) {
            minLenToEndAt[i + 1] = minLenToEndAt[i];
        }

        // If the prefix sum needed to achieve the target exists...
        const neededSum = prefixSum - target;
        if (prefixSumToIndex.has(neededSum)) {
            const startIndex = prefixSumToIndex.get(neededSum);
            // Update the minimum length for this index.
            minLenToEndAt[i + 1] = min(minLenToEndAt[i + 1], i - startIndex!);

            // If the previous subarray (ending at startIndex) is valid...
            if (minLenToEndAt[startIndex! + 1] < Number.MAX_SAFE_INTEGER) {
                // Update the minimum total length.
                minTotalLen = min(minTotalLen, minLenToEndAt[startIndex! + 1] + i - startIndex!);
            }
        }

        // Update or add the current prefix sum and its ending index to the map.
        prefixSumToIndex.set(prefixSum, i);
    }

    // If minTotalLen is not updated, return -1 to indicate no such subarrays exist.
    return minTotalLen === Number.MAX_SAFE_INTEGER ? -1 : minTotalLen;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(N)`, where `N` is the number of elements in the array `arr`. This is because the code iterates through the array once, with constant-time operations performed for each element (such as updating a dictionary, arithmetic operations, and comparison operations).

More specifically:

1. We have a single `for` loop iterating over `arr`, so we visit each element of `arr` once.
2. For each element in the `for` loop, we are performing a dictionary lookup (or insertion) for operation `d[s] = i` and `d[s - target]` which is typically `O(1)` on average for a hash table.
3. Assignments and comparisons like `f[i] = f[i - 1]` and `ans = min(ans, f[j] + i - j)` are `O(1)` operations. Therefore, combining these constant-time operations within a single loop over `n` elements, the overall time complexity is linear, represented by `O(N)`.

Space Complexity

The space complexity of the code is `O(N)` as well. This can be attributed to two data structures that scale with the input size:

1. The dictionary `d` which, in the worst case, could contain an entry for each prefix sum.
2. The list `f` which contains `n + 1` elements (as it keeps a record of the minimum length of a subarray for every index up to `n`).

Since both `d` and `f` only grow linearly with the input array's size, the space complexity is linear, represented by `O(N)`.