325. Maximum Size Subarray Sum Equals k

Medium Array Hash Table Prefix Sum

Leetcode Link

Problem Description

The problem asks us to find the maximum length of a subarray from the given array `nums`, such that the sum of its elements is exactly equal to the given integer `k`. A subarray is a contiguous part of an array. If such a subarray doesn't exist, we should return 0.

To clarify with an example, let's say our array is `[1, -1, 5, -2, 3]` and `k` is `3`. The subarray `[1, -1, 5, -2]` sums to `3`, and its length is `4`, which would be the answer because there's no longer subarray that sums to `3`.

To approach this problem, we need to find a way to efficiently look up the sum of elements from the beginning of the array up to a certain point, and determine if there is a previous point where the sum was exactly `k` less than the current sum. If we can find such a place, then the elements between these two points form a subarray that sums to `k`.

Intuition

The intuition behind the solution lies in the use of a running sum and a hash table (or dictionary in Python). As we iterate through the array, we keep track of the cumulative sum of elements. The hash table stores the earliest index at which each cumulative sum appears.

Here's the reasoning:

- Start with a hash table `d` that maps a cumulative sum to the first index where it appears. Initialize it with the pair `{0: -1}`, which says the sum `0` is achieved before the first element.
- Initialize variables for the current sum (`s`) and the maximum length found (`ans`) to `0`.
- Iterate through `nums`, updating the running total `s` by adding each element `x` to it.
- Check if `s - k` has been seen before. If it has, we've found a subarray ending at the current index with a sum of `k`.
- Update `ans` with the length of this subarray if it is longer than the maximum found so far.
- Add the current sum to the hash table with its corresponding index, but only if this sum hasn't been seen before to maintain the earliest index.

The reason we only store the earliest occurrence of a sum is because we're looking for the longest subarray; any later occurrence of the same sum would yield a shorter subarray.

Solution Approach

The solution leverages a hashing strategy to efficiently track the sum of elements in the subarrays and their starting indices. Here's a detailed walkthrough of the implementation:

1. **Hash Table Initialization**: We initiate a hash table (dictionary in Python) named `d` with a key-value pair `{0: -1}`. This represents that the sum `0` is achieved before starting the iterations, essentially before the first element at index `-1`. This base case ensures that if the sum of elements from the beginning reaches `k`, the length can be calculated correctly.

2. **Preparing Variables**: Two variables are prepared: `ans` to store the maximum length of the subarray with sum `k` found so far and `s` to keep track of the running sum.

3. **Iteration & Running Sum**: We use a loop to iterate over the array using `enumerate(nums)` which gives us both the index `i` and the value `x`. We increment the running sum `s` by the value of `x`.

4. **Checking for a Matching Subarray**: During each iteration, after updating the running sum, we check if `s - k` is present in the hash table `d`. If it is, it means that from the earliest index `d[s - k]` to the current index `i`, the elements sum up exactly to `k`. We then compare `(i - d[s - k])` with `ans` and store the larger one in `ans`.

5. **Updating the Hash Table**: The hash table is updated with the running sum only if this running sum has not been recorded before. We do this because we are interested in the first occurrence of this running sum to ensure the longest possible subarray.

6. **Returning the Result**: After the loop ends, `ans` will be holding the length of the longest subarray that sums to `k`. This value is returned as the final result.

This algorithm effectively uses the hashing technique to reduce the time complexity otherwise inevitable with brute force solutions. By using a hash table to record the first occurrence of each sum, the solution approaches an `O(n)` time complexity, since it processes each element of `nums` just once. The space complexity is also `O(n)` in the worst case, as it might store each sum occurring in the array.

Example Walkthrough

Let's use the solution approach to walk through a small example. Consider the array `nums = [3, 4, -3, 2, 1]` and the target sum `k = 3`.

1. **Hash Table Initialization**: Initialize a hash table `d` with `{0: -1}`. This signifies that before we start, the cumulative sum of `0` is reached at an index before the array starts.

2. **Preparing Variables**: The variables `ans` (maximum subarray length) and `s` (running sum) are both set to `0`.

3. **Iteration & Running Sum**:

   - For the first element `3`, the running sum `s` becomes `3`. Since `s - k = 3 - 3 = 0` and the hash table contains `0` with index `-1`, we update `ans` to `1 - (-1) = 2`.
   - For the second element `4`, `s` becomes `7`. We look for `s - k = 4` in the hash table, but it's not there, so no change to `ans`.
   - For the third element `-3`, `s` is now `4`. We look for `s - k = 1` which is not in the hash table, non update to `ans`.
   - For the fourth element `2`, `s` becomes `6`. We look for `s - k = 3`, and since it's not present, no update to `ans`.
   - For the last element `1`, `s` increases to `7`. We look for `s - k = 4`. It has appeared before when `s` was `7` for the second element. The index then was `1`. Now, the index is `4`, so the length of the subarray is `4 - 1 = 3` which is greater than the current `ans`, so we update `ans` to `3`.

4. **Checking for a Matching Subarray**: Every iteration includes this check. We successfully find a matching sum during the first and last elements.

5. **Updating the Hash Table**: As we proceed, we update the hash table with the sums `3`, `7`, `4`, and `6` at their respective first occurrences with the indices `0`, `1`, `2`, and `3`.

6. **Returning the Result**: At the end of the iteration, `ans` holds the value `3`, which is the length of the longest subarray with sum `k = 3`. This is our final result.

The correct subarray that has the sum of `3` and the maximum length is `[4, -3, 2]`. The running sum at the start of this subarray was `4`, and by adding the subarray elements, it became `7`. The subarray has `3` elements, hence `ans = 3` is returned.

Python Solution

```python
class Solution:
    def maxSubArrayLen(self, nums: List[int], target: int) -> int:
        # Initialize a dictionary to store the cumulative sum up to all indices
        sum_to_index = {0: -1}

        # Initialize variables to store the maximum length of subarray and the cumulative sum
        max_length = cumulative_sum = 0

        # Iterate through the list of numbers
        for index, num in enumerate(nums):
            # Update the cumulative sum
            cumulative_sum += num

            # Check if there is a subarray whose sum equals 'target'
            if (cumulative_sum - target) in sum_to_index:
                # Update max_length with the larger of the previous max_length and the current subarray length
                max_length = max(max_length, index - sum_to_index[cumulative_sum - target])

            # If this cumulative sum has not been seen before, add it to the dictionary
            if cumulative_sum not in sum_to_index:
                sum_to_index[cumulative_sum] = index

        # Return the maximum length of subarray found that adds up to 'target'
        return max_length
```

Java Solution

```java
import java.util.HashMap;
import java.util.Map;

class Solution {
    public int maxSubArrayLen(int[] nums, int k) {
        // Create a hashmap to store the sum up to the current index as the key
        // and the index as the value.
        Map<Long, Integer> sumToIndexMap = new HashMap<>();
        // Initialize the map with base case: a sum of 0 before index -1
        sumToIndexMap.put(0L, -1);

        // This will hold the maximum length of the subarray found so far
        int maxLength = 0;

        // This will keep track of the running sum
        long sum = 0;

        // Loop through every element in the array
        for (int i = 0; i < nums.length; ++i) {
            // Update the running sum with the current element
            sum += nums[i];

            // If a subarray ending at index i has a sum of (sum - k),
            // then a subarray with sum k exists.
            if (sumToIndexMap.containsKey(sum - k)) {
                // Compare and store the maximum length found so far
                maxLength = Math.max(maxLength, i - sumToIndexMap.get(sum - k));
            }

            // If the current sum has not been seen before,
            // add it to the map with the corresponding index.
            sumToIndexMap.putIfAbsent(sum, i);
        }

        // Return the maximum length of the subarray found
        return maxLength;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <algorithm> // For using max function

class Solution {
public:
    int maxSubArrayLen(vector<int>& nums, int k) {
        // Create a hashmap to store the cumulative sum and its index.
        unordered_map<long long, int> indexByCumulativeSum{{0, -1}};
        // Initialize variables to store the cumulative sum 's' and max length 'maxLen'.
        int maxLen = 0;
        long long cumulativeSum = 0;

        // Loop through the array to compute the cumulative sum.
        for (int i = 0; i < nums.size(); ++i) {
            cumulativeSum += nums[i];

            // If the current cumulative sum minus 'k' is found in the map, we found a subarray.
            if (indexByCumulativeSum.count(cumulativeSum - k)) {
                maxLen = max(maxLen, i - indexByCumulativeSum[cumulativeSum - k]);
            }

            // Only add the current cumulative sum and its index to the map if it's not already there.
            // This ensures we keep the smallest index to get the longest subarray.
            if (!indexByCumulativeSum.count(cumulativeSum)) {
                indexByCumulativeSum[cumulativeSum] = i;
            }
        }

        // Return the maximum length found.
        return maxLen;
    }
};
```

Typescript Solution

```typescript
/**
 * Finds the maximum length of a subarray with a sum equal to k.
 * @param nums - The input array of numbers.
 * @param k - The target sum to look for.
 * @returns The length of the longest subarray which sums to k.
 */
function maxSubArrayLen(nums: number[], k: number): number {
    // Initialize a map to store the cumulative sum as the key and its index as the value.
    const cumSumIndexMap: Map<number, number> = new Map();
    cumSumIndexMap.set(0, -1); // Base case: set the sum of an empty subarray before the first element to -1.

    let maxLength = 0; // Initialize the maximum subarray length to zero.
    let cumulativeSum = 0; // Variable to store the cumulative sum of elements.

    // Iterate over the elements of the array.
    for (let i = 0; i < nums.length; ++i) {
        cumulativeSum += nums[i]; // Increment the cumulative sum with the current element.

        // If cumulativeSum - k exists in the map, there's a subarray with sum k ending at the current index.
        if (cumSumIndexMap.has(cumulativeSum - k)) {
            // Update maxLength with the higher value between the previous maxLength and the new subarray length.
            maxLength = Math.max(maxLength, i - cumSumIndexMap.get(cumulativeSum - k)!);
        }

        // If the current cumulativeSum isn't in the map, set it with the current index.
        if (!cumSumIndexMap.has(cumulativeSum)) {
            cumSumIndexMap.set(cumulativeSum, i);
        }
    }

    // Return the maxLength found.
    return maxLength;
}
```

Time and Space Complexity

The provided Python code finds the length of the longest subarray which sums up to `k`. It makes use of a hashmap (dictionary in Python terms) to store the cumulative sum at each index, which helps in finding subarrays that sum up to `k` in constant time.

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of elements in the input list `nums`. This is because the algorithm iterates through the list once, and for each element it performs a constant time operation of adding the element to the running sum, checking if `(s - k)` is in the dictionary `d` and updating the maximum length `ans`. All these operations are constant time operations: arithmetic operations, dictionary lookup and dictionary insertion (when `s` is not already in `d`).

Space Complexity

The space complexity of the code is `O(n)`. In the worst case, every running sum is unique, and hence, each sum (represented by `s` in the code) would be stored in the dictionary `d` along with its corresponding index. Since there are `n` such running sums in the worst case, the dictionary could contain up to `n` key-value pairs.