525. Contiguous Array

Medium Array Hash Table Prefix Sum Leetcode Link

Problem Description

The problem provides an array `nums` consisting of only binary digits (0s and 1s). The task is to find a contiguous subarray where the number of 0s and 1s are equal, and this subarray should be the longest of any such subarrays in the given array. The length of this subarray is what we need to return.

Contiguous means that the elements are consecutive, without any interruptions from elements that are not part of the subarray.

For example, if the input array is `[0,1,0]`, the longest contiguous subarray with equal number of 0s and 1s would be `[0, 1]` or `[1, 0]`, and the length would be 2.

Intuition

The intuition behind the solution comes from the understanding that if we were to maintain a running difference between the number of 0s and 1s, a subarray with an equal number of 0s and 1s would correspond to a situation where the difference becomes zero. However, since we want to find the longest such subarray, we need a way to record the occurrences of the running difference.

Here's how we arrive at the solution:

1. We use a variable to keep a running count of the number of 1s and 0s we've seen so far. Let's increment this count by 1 for every 1 we see and decrement it by 1 for every 0 we see in the array.
2. We use a hashmap to keep track of the earliest index where each running count value has appeared.
3. As we iterate through the array:
   - We update the running count based on the current element.
   - We check if the running count is already in the hashmap. If it's not, we add it to the hashmap with the current index.
   - If the running count is already in the hashmap, it means we've found a subarray where the difference between the number of 0s and 1s is the same as some previous index, which implies that we have an equal number of 0s and 1s in between these two indexes.
   - We calculate the length of this subarray by subtracting the previous index of this running count value from the current index and check if it's greater than the length of the previous longest subarray we've found. If it is, we update our answer.
4. The maximum length found in this manner is the length of the longest contiguous subarray with equal numbers of 0s and 1s.

By following this approach, the algorithm elegantly handles the issue without having to compare every possible subarray, thereby reducing the complexity from O(n^2) to O(n).

Solution Approach

The implementation of the solution utilizes a hashmap (dictionary in Python) and a single pass through the array, taking advantage of the prefix sum concept and storing indices where each sum first occurs. Here's a step-by-step breakdown of the code used:

1. Initialize a variable `s` to 0, which will be used to keep track of the running count (prefix sum) of the number of 1s and 0s seen so far in the array. When a `1` is encountered in the array, `s` is incremented by 1, and for a `0`, `s` is decremented by 1.

2. Initialize another variable `ans` to 0, which will store the maximum length of the subarray found during the process.

3. Create a hashmap `mp` with one initial key-value pair `{0: -1}`, which signifies that before we start processing the array, the running count is 0 and its index is -1 (this helps handle cases where the subarray starts from index 0).

4. Iterate through the array using a loop, and for each element at index `i`:

   - Update the running count `s`. If the value `v` is 1, increment `s` by 1. If not, decrement `s` by 1 (`s += 1 if v == 1 else -1`).
   - Check if the running count `s` is already in the hashmap `mp`.
     - If it is, compute the length of the subarray (`i - mp[s]`) which represents the distance from the first occurrence of this running count to the current index. Compare and update the answer `ans` to be the maximum length found so far (`ans = max(ans, i - mp[s])`).
     - If it is not, add this running count `s` along with the current index `i` to the hashmap (`mp[s] = i`), marking the first appearance of this running sum.
5. After the loop concludes, the variable `ans` will hold the maximum length of the longest contiguous subarray with an equal number of 0s and 1s, which the function then returns.

In essence, the algorithm relies on the prefix sum pattern where changes in the running sum are monitored and recorded. When a running sum reoccurs, it indicates a potential subarray of interest. The hashmap acts as an efficient way to keep track of the indices where sums first occur, enabling the calculation of subarray lengths in constant time.

Example Walkthrough

Let's walk through a small example using the given solution approach with the input array `nums = [0, 1, 0, 1, 1, 0, 0]`.

The array is `[0, 1, 0, 1, 1, 0, 0]`. We initialize `s = 0` for the running count, `ans = 0` for the maximum length of subarray, and a hashmap `mp` with `{0: -1}`.

Now, we iterate over the array and update our running count `s`, the hashmap `mp`, and our answer `ans`.

- **Step 1:** `i = 0`, `nums[i] = 0`. `s = -1` (decremented from 0). This `-1` is not in `mp`, so `mp` is now `{0: -1, -1: 0}`. `ans` remains `0`.
- **Step 2:** `i = 1`, `nums[i] = 1`. `s = 0` (incrementing from -1). `s` is in `mp` at index `-1`, so `ans = 1 - (-1) = 2`.
- **Step 3:** `i = 2`, `nums[i] = 0`. `s = -1` (decremented from 0). `s` is in `mp`, so the length of this subarray is `2 - 0 = 2`. `ans = max(2, 2) = 2`.
- **Step 4:** `i = 3`, `nums[i] = 1`. `s = 0` (incrementing from -1). `s` is in `mp`, so the length of this subarray is `3 - (-1) = 4`. `ans = max(2, 4) = 4`.
- **Step 5:** `i = 4`, `nums[i] = 1`. `s = 1` (incremented from 0). This `1` is not in `mp`, so `mp` is now `{0: -1, -1: 0, 1: 4}`. `ans` remains `4`.
- **Step 6:** `i = 5`, `nums[i] = 0`. `s = 0` (decremented from 1). `s` is in `mp`, so the length of this subarray is `5 - (-1) = 6`. `ans = max(4, 6) = 6`.
- **Step 7:** `i = 6`, `nums[i] = 0`. `s = -1` (decremented from 0). `s` is in `mp`, so the length of this subarray is `6 - 0 = 6`. `ans` remains `6` as it's equal to the current maximum.

After going through the entire array, the longest contiguous subarray with an equal number of 0s and 1s is `[1, 0, 1, 1, 0, 0]` starting from index 1 to index 6 and has a length of `6`. This is the final answer and is the value of `ans` after iterating through the array.

Python Solution

```python
class Solution:
    def findMaxLength(self, nums: List[int]) -> int:
        # Initialize the current balance between 0s and 1s and the maximum length found
        balance = max_length = 0
        # Create a hash map to store the first occurrence of each balance
        balance_map = {0: -1}

        # Iterate over the elements in the array
        for index, value in enumerate(nums):
            # Update the balance (+1 for 1, -1 for 0)
            balance += 1 if value == 1 else -1

            # Check if the same balance has been seen before
            if balance in balance_map:
                # If we have seen this balance before, calculate the length of the subarray
                # between the previous index and the current index
                max_length = max(max_length, index - balance_map[balance])
            else:
                # If this is the first time we've seen this balance,
                # store the index for this balance in the hash map
                balance_map[balance] = index

        # Return the maximum length of the subarray with equal number of 0s and 1s
        return max_length

```

Java Solution

```java
class Solution {

    // Function to find the maximum length of a contiguous subarray with equal number of 0s and 1s.
    public int findMaxLength(int[] nums) {
        // Create a hash map to store the sum so far (key) and its index (value).
        Map<Integer, Integer> sumToIndexMap = new HashMap<>();

        // Initialize the sum of elements to 0 and the answer to the max length to 0.
        sumToIndexMap.put(0, -1); // Sum of 0 is considered to appear before the array starts.
        int sum = 0, maxLength = 0;

        // Iterate through the array.
        for (int i = 0; i < nums.length; i++) {
            // Update sum: +1 for '1', -1 for '0'. This helps in finding equal numbers of 1s and 0s.
            sum += nums[i] == 1 ? 1 : -1;

            // If the sum has been seen before, it means the subarray between the two indices has
            // equal number of 0s and 1s.
            if (sumToIndexMap.containsKey(sum)) {
                // Update maxLength with the larger value between current maxLength and the distance
                // between current index and the first index where this sum appeared.
                maxLength = Math.max(maxLength, i - sumToIndexMap.get(sum));
            } else {
                // If sum is not found in the map, add it with its index.
                sumToIndexMap.put(sum, i);
            }
        }

        // Return the found maximum length.
        return maxLength;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to find the maximum length of a contiguous subarray with an equal number of 0 and 1.
    int findMaxLength(vector<int>& nums) {
        unordered_map<int, int> prefixSumToIndex; // Map to store prefix sum to index mapping.
        int currentSum = 0; // Current prefix sum.
        int maxLength = 0; // Maximum length of the subarray found so far.
        prefixSumToIndex[0] = -1; // Initialize with the base case prefix sum before any element is processed.

        // Iterate through the array.
        for (int i = 0; i < nums.size(); ++i) {
            // Update the current sum: increment by 1 for a '1', decrement by 1 for a '0'
            currentSum += nums[i] == 1 ? 1 : -1;

            // Check if the current sum has been seen before.
            if (prefixSumToIndex.count(currentSum)) {
                // If yes, calculate the length of the subarray and update the maximum length.
                maxLength = max(maxLength, i - prefixSumToIndex[currentSum]);
            } else {
                // If not, add the current sum and index to the map.
                prefixSumToIndex[currentSum] = i;
            }
        }
        return maxLength; // Return the maximum length found.
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the maximum length of a contiguous subarray with an equal number of 0 and 1.
 * @param {number[]} nums - The input array containing 0s and 1s.
 * @return {number} - The maximum length of a contiguous subarray with equal number of 0s and 1s.
 */
function findMaxLength(nums: number[]): number {
    // Create a map to store the sum at each index. The map will help us find when we've seen a sum before.
    const sumIndexMap = new Map<number, number>();
    // Initialize sum for index -1 (before the start of the array) to 0.
    sumIndexMap.set(0, -1);
    let sum = 0; // This will hold our cumulative sum.
    let maxLength = 0; // This will store the maximum length we find.

    for (let i = 0; i < nums.length; ++i) {
        // Update sum: decrement for 0 and increment for 1.
        sum += nums[i] == 0 ? -1 : 1;

        // If the sum has been seen before, we have found a subarray with equal number of 0s and 1s.
        if (sumIndexMap.has(sum)) {
            // The length is the current index - the previous index where we've seen the same sum.
            maxLength = Math.max(maxLength, i - sumIndexMap.get(sum)!);
        } else {
            // If the sum hasn't been seen before, set the current index as the value for this sum.
            sumIndexMap.set(sum, i);
        }
    }

    // Return the maximum length found.
    return maxLength;
}

```

Time and Space Complexity

The time complexity of the code is `O(n)`, as it iterates through the `nums` list once, performing a constant amount of work for each element (checking and updating a dictionary, and updating a running sum and maximum length).

The space complexity of the code is `O(n)`, in the worst case, where `n` is the number of elements in the input list `nums`. This is because the dictionary `mp` can potentially store an entry for each unique sum encountered, which corresponds to each index in the list in the worst-case scenario.