2609. Find the Longest Balanced Substring of a Binary String

Easy String

Leetcode Link

Problem Description

Given a binary string `s` which contains only the characters '0' and '1', the task is to find the length of the longest substring where two conditions are satisfied:

1. All the zeros in the substring must come before any ones.
2. The quantity of zeros must equal the quantity of ones within the substring.

These conditions define a "balanced" substring. This includes the consideration that an empty substring (one with no characters at all) is also balanced. A substring is simply any sequence of consecutive characters from the original string.

To summarize: you need to find a contiguous sequence within the given binary string where there is an equal number of zeros and ones, and all zeros precede the ones.

Intuition

Approaching this problem, one could initially think of a brute force strategy, trying every possible substring of `s` to check if it is balanced and keeping track of the longest one. However, this method proves inefficient with a time complexity of O(n^3), where n is the length of the string `s`.

An optimized strategy is to traverse the string while tracking the number of continuous zeros and ones. For every new character, there are certain things to be done based on whether it is a zero or a one:

- If it's a '0', and if we have already encountered a '1', both the count of zeros and ones need to be reset since a balanced substring cannot have zeros after ones. If no '1' has been encountered (one is zero), simply increment the count of zeros, since we might be extending a balanced substring.
- If it's a '1', increment the count of ones. Since a balanced substring must have equal numbers of zeros and ones, we update the maximum length of a balanced substring using the minimum of the current counts of zeros and ones, multiplied by 2 (to account for both zeros and ones).

This way, we can traverse the string only once, updating the count of zeros and ones and the maximum balanced substring length as we go. This improved method significantly reduces the time complexity to O(n) with a constant space complexity, as there are no additional data structures used to keep track of potential substrings.

Solution Approach

The provided solution efficiently finds the longest balanced substring using the following approach:

1. Initialize three variables: `ans` to keep track of the maximum length discovered so far, `zero` to count the consecutive zeros, and `one` to count the consecutive ones. All are initially set to 0.
2. Traverse the string `s` character by character using a for loop. For each character `c` encountered:

   a. If `c` is '0': - Check if the count of ones is greater than 0. This indicates that we've previously encountered a '1', and since a balanced substring can't have a '0' after a '1', we must reset both `zero` and `one`. - After the check, or if no '1' has been encountered yet, increment the count of zeros (`zero`).

   b. If `c` is '1': - Simply increment the count of ones (`one`). - Calculate the length of potential balanced substring as 2 times the minimum of `zero` and `one` counts. The multiplication by 2 is necessary to account for both zeros and ones in the matching counts. - Update the `ans` with the higher value between its current value and the potential balanced substring length calculated.
3. Continue the process until the whole string has been traversed.
4. Return `ans` as the length of the longest balanced substring found.

By only using counters and traversing the string once, this solution effectively employs a single-pass algorithm. Since it avoids nested loops or extensive substring operations, it significantly optimizes the time taken compared to brute-force methods. No additional data structures beyond simple variables are used, offering the benefit of constant space complexity.

The method hinges on understanding the problem's constraints and recognizing that a balanced substring can always be identified by pairing zeros and ones as long as they are in sequence and in equal number.

Example Walkthrough

Let's consider a small binary string `s = "00110"` to illustrate the solution approach.

Step 1: Initialize `ans = 0`, `zero = 0`, `one = 0`.

Step 2: Traverse the string character by character:

- Index 0: Encounter '0'
  - `zero` becomes 1 (`zero = 1`), since `one = 0` we continue.
  - `ans` remains 0.
- Index 1: Encounter another '0'
  - `zero` increments to 2 (`zero = 2`).
  - `ans` remains 0.
- Index 2: Encounter a '1'
  - `one` increments to 1 (`one = 1`).
  - Possible balanced substring length here is `min(zero, one) * 2 = 2`, so `ans` updates to 2.
- Index 3: Encounter another '1'
  - `one` increments to 2 (`one = 2`).
  - Now we have an equal count of zeros and ones.
  - Possible balanced substring length is `min(zero, one) * 2 = 4`, so `ans` updates to 4.
- Index 4: Encounter a '0'
  - Since we've previously encountered ones and now we see a '0', we reset both `zero` and `one` as this '0' cannot be part of a balanced substring following the encountered '1's.
  - `zero` resets to 1 (`zero = 1`), and `one` resets to 0 (`one = 0`).
  - `ans` remains 4, as no new balanced substring is found.

Step 3: Having traversed the string, we've completed our single pass.

Step 4: The final `ans` is 4. Hence, the longest balanced substring has a length of 4. In the given string "00110," the substring "0011" satisfies the conditions: equal number of zeros and ones, and all zeros come before any ones.

This walkthrough demonstrates how the algorithm processes each character of the input string, updating counters and maintaining the maximum length of a balanced substring as it progresses through the string. By doing so in a single pass, it achieves an efficient O(n) time complexity.

Python Solution

```python
class Solution:
    def find_the_longest_balanced_substring(self, s: str) -> int:
        # Initialize the longest balanced length, count of zeros and ones to zero
        longest_balanced_length = zero_count = one_count = 0

        # Iterate over each character in the string
        for char in s:
            if char == '0':
                # If we encounter a zero and there is an existing one count,
                # reset both counts as we no longer have a balanced substring
                if one_count:
                    zero_count = one_count = 0
                # Increment the zero count otherwise
                zero_count += 1
            else:  # Otherwise, char is '1'
                # Increment the one count
                one_count += 1
                # Update the longest balanced length with the minimum count of zeros and ones
                # multiplied by 2 (to count both zeros and ones)
                longest_balanced_length = max(longest_balanced_length, 2 * min(one_count, zero_count))

        # Return the length of the longest balanced substring
        return longest_balanced_length
```

Java Solution

```java
class Solution {
    // This method finds and returns the length of the longest balanced substring containing equal numbers of '0's and '1's.
    public int findTheLongestBalancedSubstring(String s) {
        int countZero = 0; // Count of '0's seen so far
        int countOne = 0;  // Count of '1's seen so far
        int maxLength = 0; // Length of the longest balanced substring found so far
        int n = s.length(); // Length of the input string

        // Loop through each character in the string
        for (int i = 0; i < n; ++i) {
            // If the current character is a '0'
            if (s.charAt(i) == '0') {
                // If there were any '1's seen without a corresponding '0', reset both counts
                if (countOne > 0) {
                    countZero = 0;
                    countOne = 0;
                }
                // Increment the count of '0's
                ++countZero;
            } else {
                // If the current character is a '1', we attempt to form a balanced substring.
                // Update maxLength to be the greater of its current value and twice the minimum of countZero and countOne+1.
                // The increment on countOne is done inline within the comparison.
                maxLength = Math.max(maxLength, 2 * Math.min(countZero, ++countOne));
            }
        }
        return maxLength; // Return the length of the longest balanced substring
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int findTheLongestBalancedSubstring(string s) {
        int countZero = 0; // Initialize counter for '0's
        int countOne = 0;  // Initialize counter for '1's
        int maxLength = 0; // Store the maximum length of a balanced substring

        // Iterate through the string character by character
        for (char& c : s) {
            if (c == '0') {
                // If we encounter a '0', then there can't be a balanced substring
                // that starts before this point, so we reset the counters
                if (countOne > 0) {
                    countZero = 0;
                    countOne = 0;
                }
                // Increment the counter for '0's
                ++countZero;
            } else { // c == '1'
                // Increment the counter for '1's and update the maximum length
                // The maximum length for a balanced substring is twice the minimum
                // of countZero and countOne (since a balanced substring contains equal '0's and '1's)
                maxLength = max(maxLength, 2 * min(countZero, ++countOne));
            }
        }

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

Typescript Solution

```typescript
function findTheLongestBalancedSubstring(s: string): number {
    // Initialize counters for consecutive zeros and ones
    let zeroCount = 0;
    let oneCount = 0;

    // Initialize the answer to store the maximum length of a balanced substring
    let maxLength = 0;

    // Iterate over each character in the string
    for (const char of s) {
        if (char === '0') {
            // If a '0' is found and there are pending ones, reset the counts
            if (oneCount > 0) {
                zeroCount = 0;
                oneCount = 0;
            }
            // Increment the count for zeros
            ++zeroCount;
        } else {
            // On finding a '1', calculate the potential balanced substring length
            // and update the maximum length if necessary
            maxLength = Math.max(maxLength, 2 * Math.min(zeroCount, ++oneCount));
        }
    }

    // Return the maximum balanced substring length found
    return maxLength;
}
```

Time and Space Complexity

The time complexity of the given code is `O(n)` because it consists of a single loop that iterates over each character in the input string `s` exactly once. The size of the input string is denoted by `n`, thus resulting in a linear relationship between the input size and the number of operations performed.

The space complexity is `O(1)` as the code uses only a constant amount of additional space that does not scale with the input size. The variables `ans`, `zero`, and `one` occupy a fixed amount of space regardless of the length of the string.