696. Count Binary Substrings

Easy Two Pointers String

Leetcode Link

Problem Description

The problem asks for the number of non-empty substrings in a given binary string `s`, where these substrings have an equal number of `0`'s and `1`'s and the `0`'s and `1`'s are grouped consecutively. That means in every valid substring, all the `0`'s will be together and all the `1`'s will be together, with no mixing or interleaving. For example, the substring "0011" is valid as it has two `0`'s followed by two `1`'s, but "0101" is not valid as here `0`'s and `1`'s are not grouped together. It's also necessary to count multiple occurrences of the same substrings separately if they appear in different positions in the string `s`.

Intuition

The intuition behind the solution approach is to break the problem into smaller, more manageable pieces that we can count efficiently. We want to count substrings where `0`'s and `1`'s are grouped together, which means we only need to focus on transitions from `0` to `1` or from `1` to `0`. Whenever such a transition is found, the number of possible substrings is the minimum of the number of `0`'s and `1`'s around this transition point, because the substring must end when a different character starts.

Here's the approach in more detail:

- We first iterate through string `s` to create a list `t` that stores the lengths of consecutive `0`'s or `1`'s (we can call each consecutive sequence a "block"). For example, if `s` is "00111011", the list will be `[2, 3, 1, 2]` which corresponds to "00", "111", "0", and "11".
- Next, we look for consecutive blocks where a `0` block is followed by a `1` block, or vice versa, and add the minimum length of these two blocks to our answer. This is because, at each transition, the number of valid substrings is limited by the shorter block. This step counts the valid substrings that occur around the transitions we identified.
- We iterate over the list `t` and add up the minimum values between adjacent blocks to get the total count of substrings.

The advantage of this method is that we convert the problem from having to consider all possible substrings (which would be very inefficient) to considering only the transitions between different characters, which greatly reduces the number of possibilities we need to examine.

Solution Approach

The implementation of the solution utilizes a simple array (list in Python) to keep track of the counts of consecutive characters and then uses a single pass to calculate the number of valid substrings.

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

1. Initialize an index `i` to start from the beginning of the string `s` and determine the length `n` of the string for boundary conditions. Also, prepare an empty list `t` to store the lengths of consecutive characters.
2. Use a while loop to iterate over the string:
   - The inner while loop counts consecutive characters that are the same and increments the count `cnt`. The condition `s[i + 1] == s[i]` checks for consecutive similar characters.
   - This count is then appended to the list `t` after the end of the consecutive characters is reached, and the outer loop is moved to the next character that is different.
3. After the loop ends, you have a list `t` that contains the counts of consecutive `0`'s or `1`'s. For the given string "00111011", `t` would be `[2, 3, 1, 2]`.
4. Now, initialize a variable `ans` to `0`. This will hold the final count of valid substrings.
5. Iterate over the list `t` using a for loop starting from index `1` since we want to compare each block with its previous block:
   - For each pair of consecutive blocks, add the minimum of the two block lengths to `ans`. The expression `min(t[i - 1], t[i])` does exactly that, considering the current and previous block count.
6. Finally, return `ans`, which now contains the total number of valid substrings.

The algorithm effectively uses a grouping-and-counting technique, converting the input string into a list of counts that represents the "compressibility" of the data. This allows for an efficient counting of the substrings in a second pass without the need for nested loops and without having to check each possible substring within the original string.

This code has a linear time complexity, O(n), because it only requires two passes over the input: one to create the count list and one to compute the result from the count list. The space complexity is also linear, O(n), because it requires additional space proportional to the size of the input to store the count list.

Example Walkthrough

Let's walk through an example to illustrate the solution approach with the binary string `s = "00110011"`.

1. **Initialize index and list**: We start with an index `i = 0`, and an empty list `t = []`.
2. **Count consecutive characters**:
   - The first two characters are `0`s. The inner while loop counts these two `0`s and we append `2` to the list `t`. Now, `t = [2]`.
   - The next three characters are `1`s. We count these and append `3` to `t`. Now, `t = [2, 3]`.
   - After that, we have two `0`s, so we append `2` to `t`. Now, `t = [2, 3, 2]`.
   - Finally, we count two `1`s and `t` becomes `[2, 3, 2, 2]`.
3. **List `t` represents blocks**: After this step, for the string "00110011", the list `t` is `[2, 3, 2, 2]`. Each number represents a block of consecutive characters.
4. **Initialize answer variable**: Set `ans = 0`.
5. **Iterate over the list `t` and count substrings**:
   - We compare the first and second blocks: `2` and `3`. The minimum is `2`, so we add `2` to `ans`.
   - Next, we compare the second and third blocks: `3` and `2`. The minimum is `2`, add `2` to `ans`.
   - Then we compare the third and fourth blocks: `2` and `2`. The minimum is `2`, add `2` to `ans`.
   - After iterating, `ans` is `2 + 2 + 2 = 6`.
6. **Result**: The total number of non-empty substrings that have an equal number of `0`s and `1`s and are grouped consecutively is `6`.

Valid substrings based on the given example are "0011", "01", "1100", "10", "0011", and "01" at different positions in the string. The approach successfully simplifies the problem by focusing on the transitions from `0` to `1` or from `1` to `0` and uses the concept of counting the minimum length of consecutive blocks to determine valid grouped substrings.

Python Solution

```python
class Solution:
    def countBinarySubstrings(self, s: str) -> int:
        # Initialize the index and get the length of the input string
        index, length = 0, len(s)
        # This list will store the counts of consecutive '0's or '1's
        group_counts = []

        # Loop through the input string to create group counts
        while index < length:
            count = 1
            # If the next character is the same, increment the count
            while index + 1 < length and s[index + 1] == s[index]:
                count += 1
                index += 1
            # Append the count of the group to the list
            group_counts.append(count)
            index += 1

        # Initialize the answer variable to store the total count
        answer = 0
        # Iterate over the group counts and add the minimum count of
        # adjacent groups to the answer since they can form binary substrings
        for i in range(1, len(group_counts)):
            answer += min(group_counts[i - 1], group_counts[i])

        # Return the total count of binary substrings
        return answer

```

Java Solution

```java
class Solution {
    public int countBinarySubstrings(String s) {
        // Initialize the current index 'currentIndex' which is used to traverse the string
        int currentIndex = 0;
        // The length of the input string 's'
        int length = s.length();
        // A list to store the consecutive character counts
        List<Integer> groups = new ArrayList<>();

        // Traverse the entire string
        while (currentIndex < length) {
            // Starting count is 1 since we're looking at one character initially
            int count = 1;
            // Check if the next character is the same as the current one
            while (currentIndex + 1 < length && s.charAt(currentIndex + 1) == s.charAt(currentIndex)) {
                currentIndex++;
                count++;
            }
            // Add the count to the list of group counts
            groups.add(count);
            // Move to the next character
            currentIndex++;
        }

        // Initialize 'totalSubstrings' to count the number of valid binary substrings
        int totalSubstrings = 0;
        // Iterate through the list of group counts
        for (int i = 1; i < groups.size(); i++) {
            // Add the minimum count of adjacent groups since that is the maximum number of
            // valid substrings we can get from those two groups
            totalSubstrings += Math.min(groups.get(i - 1), groups.get(i));
        }

        // Return the total number of valid binary substrings
        return totalSubstrings;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int countBinarySubstrings(string s) {
        int currentIndex = 0, stringSize = s.size(); // Initialize variables for the current index and the size of the string.
        vector<int> groupLengths; // This vector will hold the lengths of consecutive groups of '0's or '1's.

        // Process the input string to populate groupLengths with the lengths of consecutive groups.
        while (currentIndex < stringSize) {
            int count = 1; // Start with a count of 1 for the current character.
            // Count consecutive characters that are the same.
            while (currentIndex + 1 < stringSize && s[currentIndex + 1] == s[currentIndex]) {
                ++count;
                ++currentIndex;
            }
            groupLengths.push_back(count); // Add the count to the list of group lengths.
            ++currentIndex; // Move to the next character (or next group of characters).
        }

        int answer = 0; // This will hold the total count of valid binary substrings.

        // Calculate the number of valid binary substrings using the lengths of groups.
        for (int i = 1; i < groupLengths.size(); ++i) {
            // The number of valid substrings for a pair of adjacent groups is the
            // minimum of the lengths of those two groups.
            answer += min(groupLengths[i - 1], groupLengths[i]);
        }

        return answer; // Return the total count of valid binary substrings.
    }
};

```

Typescript Solution

```typescript
// Counts the number of binary substrings with equal number of consecutive 0s and 1s.
function countBinarySubstrings(s: string): number {
    let currentIndex = 0, stringSize = s.length; // Initialize variables for the current index and the size of the string.
    let groupLengths: number[] = []; // This array will hold the lengths of consecutive groups of '0's or '1's.

    // Process the input string to populate groupLengths with the lengths of consecutive groups.
    while (currentIndex < stringSize) {
        let count = 1; // Start with a count of 1 for the current character.
        // Count consecutive characters that are the same.
        while (currentIndex + 1 < stringSize && s[currentIndex + 1] === s[currentIndex]) {
            count++;
            currentIndex++;
        }
        groupLengths.push(count); // Add the count to the list of group lengths.
        currentIndex++; // Move to the next character (or next group of characters).
    }

    let answer = 0; // This will hold the total count of valid binary substrings.

    // Calculate the number of valid binary substrings using the lengths of groups.
    for (let i = 1; i < groupLengths.length; i++) {
        // The number of valid substrings for a pair of adjacent groups is the
        // minimum of the lengths of those two groups.
        answer += Math.min(groupLengths[i - 1], groupLengths[i]);
    }

    return answer; // Return the total count of valid binary substrings.
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by the two loops present in the algorithm. The first loop iterates over the string `s` to count consecutive characters and fill the array `t` with the lengths of these consecutive sequences. This loop has a complexity of `O(n)` where `n` is the length of the string `s`, because each character is visited at most twice.

The second loop goes through the array `t` (which has at most `n` elements if the string `s` is alternating between `0` and `1`) calculating the number of valid binary substrings, which is also done in `O(n)` time because it processes each group once.

Therefore, the overall time complexity of the code is `O(n) + O(n) = O(n)`.

Space Complexity

The space complexity is also determined by the array `t`. This array holds at most `n` elements in the worst-case scenario when the binary string alternates characters. Hence, the space complexity of the code is `O(n)` due to the storage requirements of the array `t`.