1180. Count Substrings with Only One Distinct Letter

Easy Math String

Leetcode Link

Problem Description

The problem requires counting the substrings within a given string `s` that meet a specific criterion: each substring must contain exactly one distinct letter. In other words, any given substring is a sequence of the same character, with no other different characters inside. To illustrate, if `s` is "aaa", we should count the individual substrings "a", "a", "a", the pairs "aa", "aa", and the entire string "aaa", which gives us a total of 6 such substrings.

Intuition

To solve this problem, we can observe patterns in the strings and how substrings forming consecutive identical characters contribute to the total count. Consider a substring with `n` identical characters; it will have `1 + 2 + 3 + ... + n` distinct substrings with one distinct letter, which is the sum of the first `n` natural numbers. We can use the formula for this sum, `n * (n + 1) / 2`, to find the number of valid substrings in a piece of the sequence without the need for an exhaustive search for all possible substrings.

With this knowledge, we can look for these consecutive substrings of identical characters. We iterate through the string with two pointers (or indices) `i` and `j`. We use `i` to start at the beginning of a substring with identical characters and `j` to find its end. For each substring we find, we calculate its contribution to the total count using our sum formula and add it to `ans`, which keeps track of the number of valid substrings. We repeat this process until we've considered every character in the string, at which point `ans` will contain the final count of substrings with one distinct letter.

Solution Approach

The solution utilizes a simple iterative approach with two pointers to traverse the string efficiently. The key idea is to find consecutive groups of the same character and then calculate the number of substrings that can be formed from these groups using a mathematical formula. This eliminates the need for nested loops to consider every possible substring explicitly. We'll break down the steps of the implementation:

1. Initialize two pointers, `i` and `j`. `i` will scan through the string, and `j` will be used to find the end of a consecutive group of identical characters starting at `i`.
2. Initialize `ans` to zero, which will be used to accumulate the total number of substrings with one distinct letter.
3. Use a `while` loop to iterate over the string with the condition `i < n`, ensuring we don't go past the string's end.
4. Within the loop, set `j` to `i` to mark the beginning of the new possible consecutive group.
5. Start a nested `while` loop to move `j` forward as long as `j < n` and `s[j]` is equal to `s[i]`. This loop determines the length of the consecutive group.
6. With the length of the consecutive group determined (`j - i`), calculate the number of substrings using the formula `(1 + j - i) * (j - i) // 2`. This formula represents the sum of the first `j - i` natural numbers.
7. Add this calculated number to `ans`, accumulating the count of valid substrings.
8. Update `i` to `j` to move the first pointer to the next group of identical characters.
9. Return `ans` once the entire string has been scanned.

The algorithm makes a single pass over the string, achieving linear time complexity, O(n), where n is the length of the input string. The space complexity is O(1) as it only uses a constant amount of extra space for the pointers and counter. This efficiency stems from the combination of the two pointers technique and the application of the arithmetic series sum formula.

Example Walkthrough

Let's apply the solution approach using the string `s = "aabbbc"` as an example.

Following the steps:

1. **Initialize two pointers, `i` and `j`.** We start by setting both `i = 0` and `j = 0`.
2. **Initialize `ans` to zero.** This will hold the count of valid substrings. So, `ans = 0`.
3. **Use a `while` loop to iterate over the string.** Since `i < n` where `n = 6` (length of "aabbbc"), the loop begins.
4. **Within the loop, set `j = i`.** This acknowledges the start of a new consecutive group (we expect `j` to find how long it goes).
5. **Start a nested `while` loop with `j < n` and `s[j] == s[i]`.**
   - For `i = 0`, `j` moves from `0` to `1` as `s[0]` (which is 'a') is the same as `s[1]`. `j` stops at `2` because `s[2]` is different ('b').
   - The group "aa" ends at index `1`, thus with length `j - i = 2`.
6. **Calculate the number of substrings:** `(1 + j - i) * (j - i) // 2 = (1 + 2 - 0) * (2 - 0) // 2 = 3 * 2 // 2 = 3`.
7. **Add to `ans`:** `ans = ans + 3` which becomes `ans = 3`.
8. **Update `i` to `j`:** set `i` to `2` where `j` had stopped.

Now we continue the steps for the next group of identical characters 'b':

9. **While loop continues with `i = 2`.** As before, we set `j = i`.
10. **Nested `while` loop identifies the group "bbb":** The loop starts at `j = 2` and increments `j` until it reaches `5`, right before 'c', as 'b's are consecutive until index `4`.
11. **Calculate for "bbb":** `(1 + j - i) * (j - i) // 2 = (1 + 5 - 2) * (5 - 2) // 2 = 4 * 3 // 2 = 6`.
12. **Add to `ans`:** `ans = ans + 6` which now becomes `ans = 9`.
13. **Update `i` to the new position `j = 5`,** skipping over the 'bbb' group.

The loop won't find any more groups longer than 1 character ('c' is a single character).

14. **Remaining single character** will be the group "c":
    - Since `j == n` and no further groups are possible, we simply add `1` for the solo 'c'.
    - `ans = ans + 1` which now becomes `ans = 10`.

The string has been fully scanned:

15. **Return `ans`.** The final count of substrings where each contains exactly one distinct letter is `10`.

We've just applied the solution approach using two-pointer technique to efficiently calculate the number of substrings within the string "aabbbc" that consist of the same character. The total count of valid substrings in this case is `10`, which is the sum of substrings from groups "aa", "bbb", and "c".

Python Solution

```python
class Solution:
    def countLetters(self, s: str) -> int:
        # Initialize the length of the string for easy reference
        string_length = len(s)

        # Initialize the index and answer variables
        index, total_count = 0, 0

        # Iterate over the string, using 'index' as the starting pointer
        while index < string_length:
            # Set 'current_char' as the character at the current index
            current_char = s[index]
            # Initialize 'span_length' which will count the span of identical characters
            span_length = 0

            # Count continuous span of identical characters starting from 'index'
            while index < string_length and s[index] == current_char:
                span_length += 1
                index += 1

            # Calculate the total substrings for the span and add to 'total_count'
            # The formula (span_length * (span_length + 1)) // 2 calculates the number of total
            # possible substrings in a string containing identical characters.
            total_count += (span_length * (span_length + 1)) // 2

        # Return the total count of all possible substrings
        return total_count
```

Java Solution

```java
class Solution {

    // This method counts all possible substrings which consist of the same character
    public int countLetters(String s) {
        int totalCount = 0; // Initialize total count of valid substrings

        // Iterate through the string starting from the first character
        for (int currentIndex = 0, stringLength = s.length(); currentIndex < stringLength;) {
            int nextIndex = currentIndex; // Index to find the end of a group of identical characters
            // Continue while we have the same character as at currentIndex
            while (nextIndex < stringLength && s.charAt(nextIndex) == s.charAt(currentIndex)) {
                nextIndex++;
            }

            // Calculate the number of substrings that can be formed with the same character
            // and add it to totalCount. It is based on the arithmetic series (n(n+1)/2),
            int sameCharCount = nextIndex - currentIndex;
            totalCount += (sameCharCount + 1) * sameCharCount / 2;

            // Skip to the next character group
            currentIndex = nextIndex;
        }

        // Return the total count of substrings
        return totalCount;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to count the total number of substrings that have all the same letters
    int countLetters(string s) {
        int totalCount = 0; // This will hold the final count of substrings
        int n = s.size(); // Get the size of the string to iterate over

        // Loop through the string
        for (int i = 0; i < n) {
            // Start of the current substring with the same character
            int start = i;

            // Find the end of the current group of the same character
            while (i < n && s[i] == s[start]) {
                ++i;
            }

            // Length of the group of the same character
            int length = i - start;

            // Add the count of substrings for this group to totalCount
            // Counts the number of substrings that can be formed with `length` characters,
            // which is the sum of the series 1 + 2 + ... + length.
            totalCount += (1 + length) * length / 2;

            // No need to set i = j, as i is already at the end of the current character group
        }

        return totalCount; // Return the total count of all such substrings
    }
};
```

Typescript Solution

```typescript
/**
 * Counts the total number of contiguous occurrences of each letter
 * in the string `s`. For each continuous group of the same character,
 * it adds up a series, where the nth character contributes n to the count
 * (e.g., for "aa" it adds 1 for the first 'a' and 2 for the second 'a', giving 3).
 *
 * @param {string} s - The string to analyze.
 * @return {number} - The total count of contiguous letters.
 */
function countLetters(s: string): number {
    let totalCount = 0; // Initialize total count
    const lengthOfS = s.length; // Cache the length of the string

    // Iterate over the string
    for (let index = 0; index < lengthOfS; ) {
        let currentIndex = index; // Index used to find contiguous characters
        let contiguousCount = 0; // Reset counter for contiguous characters

        // Count contiguous occurrences of the character
        while (currentIndex < lengthOfS && s[currentIndex] === s[index]) {
            ++currentIndex; // Move to the next character
            totalCount += ++contiguousCount; // Increment and add to total count
        }

        // Continue from where the last contiguous sequence ended
        index = currentIndex;
    }

    return totalCount; // Return the computed total count
}
```

Time and Space Complexity

Time Complexity

The given Python code for `countLetters` has two nested loops. However, the inner loop does not start from the beginning every time, but rather from the index where the outer loop left off. The inner loop only runs when it finds characters in string `s` that are the same as the character at index `i`, and once it finds a different character, it breaks and sets `i` to `j` (the next start position). This means each character in the string is visited exactly once by the inner loop, and thus the total number of iterations across both loops is `O(n)`, where `n` is the length of the string `s`.

Therefore, **time complexity is `O(n)`.**

Space Complexity

As for the space complexity, the code uses a fixed number of integer variables (`n`, `i`, `j`, `ans`) that do not depend on the size of the input string `s`. No additional data structures are used that would grow with the input size.

Therefore, **space complexity is `O(1)`.**