2067. Number of Equal Count Substrings

Medium String Counting Prefix Sum Leetcode Link

Problem Description

In the given LeetCode problem, we need to deal with a string `s` that is composed of lowercase English letters, and we are provided with an integer `count`. The problem's central concept is to identify "equal count substrings." An equal count substring is defined as a substring where each unique letter appears exactly `count` times within it.

Essentially, the task is to find all the substrings where the frequency of each distinct letter within the substring equals the given count. We then need to return the total count of such substrings.

Given the nature of strings and substrings, the problem might appear complex because there can be a lot of substrings to consider, especially as the length of the string increases. We are asked to calculate the number of substrings that meet the criteria, not to list them, which slightly simplifies the task.

Intuition

The intuition behind the solution is to leverage the sliding window technique combined with frequency counting for every unique letter. Here's a breakdown of how we can approach the problem:

1. We need to find substrings that contain each unique character `count` times. Since we are limited to lowercase English letters, there is a maximum of 26 unique characters that can appear in substrings.
2. We initialize a counter called `ans` that will keep track of the total number of equal count substrings found.
3. Begin by iterating over a range from 1 to 27, which corresponds to the possible number of unique characters in a substring (as there are 26 lowercase letters). The variable `x` represents this count of unique characters.
4. For each `x`, we calculate `m`, the minimum length a substring needs to be to contain `x` unique characters `count` times each. If `m` exceeds the string length, we can break out of the loop because it is no longer possible for any further substrings to meet the criteria.
5. We use a sliding window method in combination with a counter `cnt` to keep track of the occurrences of each character as we slide through the string.
6. By moving the right edge of the window, we add characters to the `cnt` and keep track of how many characters meet the `count` condition (`y`).
7. Once the window reaches the size `m`, we start moving the left edge to remove characters from the `cnt`. Adjust the counts for the character we remove and modify the `y` accordingly to reflect the current state of our window.
8. We increment `ans` each time the number of characters meeting the criterion `x` is the same as `y`, which implies we found an equal count substring.

By the end of this process, we will have iterated over all possible substrings that can have an equal count of all its unique characters. Each time the window slides and the conditions match, we would increment `ans`, and finally, we return the value of `ans` as the result.

Solution Approach

The solution employs a combination of the **sliding window technique** and a **frequency counter (Counter)** to efficiently identify all equal count substrings within the given string `s`.

Here's how the implementation works:

- A Python `Counter` from the `collections` module is utilized to maintain the frequency of each character within the current sliding window.
- The solution begins by iterating over all potential counts of unique characters a substring might have (`x`), ranging from `1 to 26`. This corresponds to the possible number of characters (considering the alphabet has 26 letters) that can fulfill the equal count condition.
- `m` is calculated as `count * x` and represents the minimum size the window needs to have for there to be `x` unique characters each appearing `count` times.
- The sliding window is iterated across the string `s` using a for-loop. For each character `c` encountered, the `Counter cnt` is updated to include the new character.
- `y` tracks how many characters within the current window meet the condition of appearing exactly `count` times.
- When the end of the window goes beyond index `i - m`, indicating that the window is larger than the size needed to fit `x` unique characters `count` times each, we start to shrink the window from the left.
- As we shrink the window, we decrease the frequency of the character that goes out of the window and update `y` to reflect whether the character going out still meets the `count` condition or not.
- If the current window exactly contains `x` characters that meet the `count` condition, then this window is a valid equal count substring, and `ans` is incremented.
- The above steps are repeated for each `x` until all potential substring lengths have been checked or until `m` exceeds the length of the string.

The Python `Counter` data structure speedily updates and keeps track of character frequencies, and the sliding window technique ensures that the algorithm considers each substring without redundant recalculations.

In code, this is represented as:

```python
class Solution:
    def equalCountSubstrings(self, s: str, count: int) -> int:
        ans = 0
        for x in range(1, 27):
            m = count * x
            if m > len(s):
                break
            cnt = Counter()
            y = 0
            for i, c in enumerate(s):
                cnt[c] += 1
                y += cnt[c] == count
                y -= cnt[c] == count + 1
                j = i - m
                if j >= 0:
                    cnt[s[j]] -= 1
                    y += cnt[s[j]] == count
                    y -= cnt[s[j]] == count - 1
                ans += x == y
        return ans
```

Each iteration adaptive to the character presence updates the answer `ans` when a valid window is found. The algorithm's complexity is primarily driven by the length of the input string `s` and the interaction between the count of unique characters and the sliding window.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach with a string `s = "aabbcc"` and `count = 2`.

We must find substrings where each distinct character appears exactly twice. Here's how the sliding window technique and frequency counting would be used:

1. We iterate from `x = 1` to `x = 26`, where `x` represents the number of unique characters that must each appear `count` times within a substring.
2. Let's start with `x = 1`. The minimum length `m` for a substring to have one unique character appearing twice is `m = count * x = 2 * 1 = 2`.
3. We initialize a counter `cnt` and a variable `y` to keep track of characters meeting the condition.
4. As we slide through the string, we will encounter "aa", "bb", and "cc" as substrings of length `m = 2`. Each time we slide and the condition matches (where `x = y`), we increment `ans`.
5. For our example, "aa", "bb", and "cc" are valid substrings when `x=1`. So, `ans = 3` after processing for `x=1`.
6. Next, take `x = 2`. Here, `m = count * x = 2 * 2 = 4`. We use the window size of 4, slide through the string, and find "aabb", "bbcc".
7. For `x = 2`, our window "aabb" and "bbcc" are valid, so `ans = ans + 2 = 3 + 2 = 5`.
8. We proceed with this process, increasing `x` until `m` exceeds the string length or until `x = 26`.
9. In this example, for `x = 3`, `m = count * x = 2 * 3 = 6`, which is equal to the length of the string. So "aabbcc" is also a valid substring, `ans = ans + 1 = 5 + 1 = 6`.
10. Since we've reached a window length equal to the entire string, we stop our iteration here as no longer substrings can be formed.

Through this process for each possible value of `x`, we have found the total number of equal count substrings to be `6`. This method aggregates counts for all possible window sizes that match the condition of having each unique character appear exactly `count` times.

Python Solution

```python
from collections import Counter

class Solution:
    def equalCountSubstrings(self, s: str, count: int) -> int:
        num_substrings = 0  # Initialize the number of valid substrings

        # Iterate through the possible numbers of unique characters (1 to 26 for the alphabet)
        for unique_chars in range(1, 27):
            # Calculate the length of the substring that must be considered given the count
            required_length = count * unique_chars
            # If the required length exceeds the string length, no further consideration is needed
            if required_length > len(s):
                break

            # Initialize a counter for the occurrence of characters
            char_count = Counter()
            # Initialize a variable to track the number of unique characters with exactly 'count' occurrences
            current_count_match = 0

            # Iterate through characters of the string
            for index, char in enumerate(s):
                char_count[char] += 1
                # If a character reaches the exact 'count', increment the current count match
                if char_count[char] == count:
                    current_count_match += 1
                # If a character exceeds the 'count', decrement the current count match
                elif char_count[char] == count + 1:
                    current_count_match -= 1

                # Calculate the start index of the current window
                start_index = index - required_length
                # If start index is valid, update the character count and current count match
                if start_index >= 0:
                    char_count[s[start_index]] -= 1
                    # If count of a character becomes 'count', increment the current count match
                    if char_count[s[start_index]] == count:
                        current_count_match += 1
                    # If count of a character falls below 'count', decrement the current count match
                    elif char_count[s[start_index]] == count - 1:
                        current_count_match -= 1

                # If the current count match is equal to the number of unique characters needed,
                # increment the result
                num_substrings += unique_chars == current_count_match

        # Return the total number of valid substrings
        return num_substrings

```

Java Solution

```java
class Solution {
    public int equalCountSubstrings(String s, int count) {
        int answer = 0; // This will contain the final count of valid substrings.
        int stringLength = s.length(); // The length of the input string.

        // Iterate over possible distinct character counts from 1 to 26 (inclusive),
        // but only if 'count * x' does not exceed the length of the string.
        for (int numDistinctChars = 1; numDistinctChars <= 26 && count * numDistinctChars <= stringLength; ++numDistinctChars) {
            // The length of the substring we're looking for based on 'numDistinctChars'.
            int substringLength = count * numDistinctChars;
            int[] charCount = new int[26]; // Count of each character in the current window.
            int validCharCount = 0; // Number of characters with exactly 'count' occurrences in the current window.

            // Iterate over the characters of the string.
            for (int i = 0; i < stringLength; ++i) {
                int currentCharIndex = s.charAt(i) - 'a'; // Convert char to index (0-25).
                ++charCount[currentCharIndex]; // Increment the count for this character.

                // Update validCharCount for the current character count.
                if (charCount[currentCharIndex] == count) {
                    ++validCharCount;
                }
                if (charCount[currentCharIndex] == count + 1) {
                    --validCharCount;
                }

                // Remove the character from the start of the window if it's outside the window.
                int startWindowIndex = i - substringLength;
                if (startWindowIndex >= 0) {
                    int startCharIndex = s.charAt(startWindowIndex) - 'a';
                    --charCount[startCharIndex]; // Decrement the count for this character.

                    // Update validCharCount after decrementing.
                    if (charCount[startCharIndex] == count) {
                        ++validCharCount;
                    }
                    if (charCount[startCharIndex] == count - 1) {
                        --validCharCount;
                    }
                }

                // If the number of valid characters matches the distinct character count,
                // we've found a valid substring, so increment the answer.
                if (numDistinctChars == validCharCount) {
                    ++answer;
                }
            }
        }
        return answer; // Return the total count of valid substrings.
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int equalCountSubstrings(string s, int count) {
        int totalSubstrings = 0; // This will hold the total count of substrings found
        int strLength = s.size(); // The length of the input string
        int charCount[26]; // Array to keep count of each character within a window

        // Iterate over possible lengths of substrings which are multiples of count
        for (int substringFactor = 1; substringFactor * count <= strLength; ++substringFactor) {
            int windowSize = count * substringFactor; // Calculate window size
            memset(charCount, 0, sizeof charCount); // Initialize character counts to 0
            int currentFactorCount = 0; // Tracks how many characters have reached the 'count' within the window

            // Iterate over each character in the input string
            for (int i = 0; i < strLength; ++i) {
                int charIndex = s[i] - 'a'; // Convert character to index (0-25)
                ++charCount[charIndex]; // Increment the count for this character

                // Increment currentFactorCount if count for this character has reached 'count'
                if (charCount[charIndex] == count) {
                    ++currentFactorCount;
                } else if (charCount[charIndex] == count + 1) {
                    --currentFactorCount; // Decrement if the count exceeds 'count'
                }

                // If we've moved past the first window, start removing characters that fall out
                if (i >= windowSize) {
                    int removeIndex = s[i - windowSize] - 'a'; // Character to remove from window
                    --charCount[removeIndex]; // Decrement the count for removed character

                    // Adjust currentFactorCount based on the new count of the removed character
                    if (charCount[removeIndex] == count) {
                        ++currentFactorCount;
                    } else if (charCount[removeIndex] == count - 1) {
                        --currentFactorCount;
                    }
                }

                // If currentFactorCount equals substringFactor, it means we have a valid substring
                totalSubstrings += (substringFactor == currentFactorCount);
            }
        }
        return totalSubstrings; // Return the total count of valid substrings
    }
};

```

Typescript Solution

```typescript
/**
 * Counts the number of substrings where each unique character occurs exactly `count` times.
 * @param {string} s - The input string.
 * @param {number} count - The exact number of times each character should repeat in a substring.
 * @return {number} The count of valid substrings.
 */
var equalCountSubstrings = function(s: string, count: number): number {
    let answer: number = 0;
    const strLength: number = s.length;

    // Iterate through possible substrings with unique character counts from 1 to 26
    for (let numOfUniqueChars = 1;
         numOfUniqueChars <= 26 && numOfUniqueChars * count <= strLength;
         ++numOfUniqueChars) {
        const windowSize: number = numOfUniqueChars * count;
        const charCount: number[] = new Array(26).fill(0);
        let validCount: number = 0;

        // Iterate through the string to count the occurrences of each character
        for (let i = 0; i < strLength; ++i) {
            const currentCharIndex: number = s.charCodeAt(i) - 'a'.charCodeAt(0);
            ++charCount[currentCharIndex];
            validCount += charCount[currentCharIndex] === count ? 1 : 0;
            validCount -= charCount[currentCharIndex] === count + 1 ? 1 : 0;

            const startIndex = i - windowSize;

            // Decrement the count for the character going out of the window
            if (startIndex >= 0) {
                const oldCharIndex: number = s.charCodeAt(startIndex) - 'a'.charCodeAt(0);
                --charCount[oldCharIndex];
                validCount += charCount[oldCharIndex] === count ? 1 : 0;
                validCount -= charCount[oldCharIndex] === count - 1 ? 1 : 0;
            }

            // If the number of valid characters equals the number of unique characters,
            // it is a valid substring.
            answer += numOfUniqueChars === validCount ? 1 : 0;
        }
    }

    return answer;
};

```

Time and Space Complexity

Time Complexity

The provided Python code involves nested loops where the outer loop iterates over a range determined by the input string length and the constant `count`. Specifically, the outer loop can run up to 26 times representing the number of unique letters in the English alphabet, and it breaks early if the count multiplied by the current index exceeds the length of the string `s`.

- The outer loop runs at most 26 times (for each possible unique character count `x`).
- The inner loop runs linearly with the length of the string `s`, processing each character exactly once.
- Within the inner loop, operations such as incrementing `Counter` values, conditional checks, and +/- operations are constant time.

The time complexity, therefore, would be O(26 * n) which simplifies to O(n), where `n` is the length of string `s`.

However, if we delve deeper, we may notice that the computations depend not only on `n` but also on `m` which is `count * x`. The inner loop performs work proportional to both `n` and the window size determined by `m`. So, even though we iterate over the string once, we must consider the cost of maintaining the sliding window. Therefore, the time complexity is more accurately represented as O(n * x), where `x` can be up to a maximum of `n / count`.

Space Complexity

- The `Counter` object holds at most 26 keys at any time (for each letter of the English alphabet), but the number of keys will actually correspond to the number of unique characters in each considered substring of s, bound by `m`.
- The variables `ans`, `y`, `x`, `m`, `i`, `j`, and `c` use constant space.

Thus, the space complexity is primarily determined by the `Counter` object, which is O(1) because the number of unique characters in Counter is capped by the alphabet size, which is a constant.

In conclusion, the code has a linear time complexity with respect to the size of the string 's' and a constant space complexity.