1234. Replace the Substring for Balanced String

Medium String Sliding Window Leetcode Link

Problem Description

In this problem, we are given a string `s` with a length of `n`, where `n` is an integer and the string consists only of the characters 'Q', 'W', 'E', and 'R'. The task is to find the minimum length of a substring of `s` that we can replace with another string of the same length such that the resulting string is balanced. A string is considered balanced if each of the four characters appears exactly `n / 4` times, where `n` is the total length of the string. If the string is already balanced, we should return 0.

Intuition

The intuition behind the solution involves understanding that if the string is already balanced, the answer is 0, since we will not need to replace any substring. If the string is not balanced, we want to find the shortest substring which, if replaced, would balance the whole string. To approach this, we use what's called the two-pointer technique.

Here's the intuition step-by-step:

1. **Counting Characters**: We start by counting the occurrences of each character in the string using a Counter data structure.
2. **Checking for Already Balanced String**: Check if the string is already balanced by verifying if each character occurs `n / 4` times. If it is, return 0.
3. **Finding the Minimum Substring**: Use two pointers to find the minimum substring which, if replaced, leads to a balanced string. We start with the first pointer `j` at the beginning of the string, and the second pointer `i` traverses the string. As `i` moves forward, we keep decrementing the count of each character.
4. **Sliding Window**: While the current window (from `j` to `i`) contains extra characters beyond `n / 4`, we shrink this window from the left by incrementing `j` and updating the character counts accordingly.
5. **Calculating the Answer**: The length of the current window from `j` to `i` represents the length of a substring that we can replace to potentially balance the string. We keep track of the minimum such window as we iterate through the string.

By maintaining this sliding window and adjusting its size as we iterate through the entire string, we are able to isolate the minimum subset of characters that, if replaced, will balance the original string.

Solution Approach

The implementation of the provided solution follows this approach:

1. **Using a Counter**: A `Counter` from Python's `collections` module is used to tally the occurrences of each character in the string. The `Counter` data structure facilitates the efficient counting and updating of character frequencies.
2. **Early Return for Balanced Strings**: Before proceeding with the main algorithm, the solution checks if the string is already balanced by comparing each character's count with `n // 4`. If all counts are within this limit, the function returns `0`, as no replacement is needed.
3. **Two-Pointer Technique for Sliding Window**: The solution implements two pointers, `i` and `j`, to maintain a sliding window within the string. The `i` pointer iterates over the string in a for-loop, while `j` is controlled within a nested while-loop that acts as the left boundary of the sliding window.
4. **Decrementing Count When Moving `i`**: As the `i` pointer advances through the characters of the string, it decrements the count of each character in the `Counter`. This represents that characters within the window bounded by `i` and `j` could be part of the substring to be replaced to balance the string.
5. **Shrinking the Window**: Inside the for-loop, a while-loop checks if the character counts are within the balance threshold (`n // 4`). If they are, this means the current window can possibly be a candidate for replacement. To find the smallest such window, `j` is moved to the right, effectively shrinking the window, and the corresponding character's count is incremented.
6. **Updating Minimum Answer**: The length of each valid window is calculated with `i - j + 1`, and the minimum value is updated accordingly. This length represents the length of the shortest substring that can be replaced to balance the `s`.
7. **Returning the Result**: After the iterations are complete, the smallest length found is returned as the result.

The essence of this problem lies in the sliding window technique, combined with the efficient counting approach provided by `Counter`. This allows the solution to dynamically change the boundaries of the potential substrings and find the minimum length needed to balance the string.

Here is a relevant part of the code explaining this technique:

```python
# Setup of the Counter with character frequencies
cnt = Counter(s)
n = len(s)
# Early check for balance
if all(v <= n // 4 for v in cnt.values()):
    return 0
ans, j = n, 0
# Traverse the string with i pointer
for i, c in enumerate(s):
    cnt[c] -= 1  # Decrement count for current character
    # Inner loop to shrink the sliding window and update the minimum answer
    while j <= i and all(v <= n // 4 for v in cnt.values()):
        ans = min(ans, i - j + 1)
        cnt[s[j]] += 1  # Increment count as we leave the current character out of the window
        j += 1
# Return the smallest window length found
return ans
```

This algorithm runs in O(n) time where `n` is the length of the string, since each character is visited at most twice (once by `i` and once by `j`).

Example Walkthrough

Let's consider the string `s = "QWERQQQW"`. The length of `s` is `n = 8`, and a balanced string requires each character 'Q', 'W', 'E', 'R' to appear exactly `n / 4 = 2` times each.

1. After counting each character, we get the frequencies: {'Q': 4, 'W': 2, 'E': 1, 'R': 1}. Since 'Q' appears more than twice and 'E' and 'R' appear less than twice, the string is not balanced.
2. We will now use the two-pointer technique. We initialize the answer with the length of the string (`ans = 8`) and start with the first pointer `j` at index `0`.
3. We start iterating through the string with our second pointer `i`. As we move `i` from left to right, we decrement the counter for each character.
4. For each position of `i`, we check if we can move `j` to the right to shrink the window. We move `j` to the right as long as every character is within the balance limit (not needed more than twice).
5. While checking the window, we find that between index `j = 3` and `i = 7` ("ERQQQW"), we can replace 'Q' with 'E' and 'R', resulting in 'EW', which would balance the string. The length of this window is `5`.
6. However, we proceed and find a smaller window: When `i` is at the last 'W' and `j` has moved after the second 'Q', the substring "QQQW" from index `j = 4` to `i = 7` can be replaced with 'E', 'R', and 'Q' to balance the string as "ERQW". The length of this window is `4`, which is less than the previous found length.
7. After finishing traversing the string, the smallest window we discovered is of length `4`. There are no shorter substrings that, if replaced, would balance `s`, so the answer is `4`.

Therefore, the minimum length of the substring to replace is `4` to make the string balanced ("QWERERQW").

Python Solution

```python
from collections import Counter

class Solution:
    def balancedString(self, s: str) -> int:
        # Count the frequency of each character in string s.
        char_count = Counter(s)
        n = len(s)

        # Check if the string is already balanced.
        if all(count <= n // 4 for count in char_count.values()):
            return 0

        # Initialize the answer as the length of the string.
        min_len = n
        # Initialize the window's starting index.
        start_index = 0

        # Expand the window with end index 'i'.
        for i, char in enumerate(s):
            # Decrease the count of the current character.
            char_count[char] -= 1

            # Contract the window from the left (start_index),
            # ensuring all character counts are balanced (<= n // 4).
            while start_index <= i and all(count <= n // 4 for count in char_count.values()):
                # Update the minimum length if a smaller balanced window is found.
                min_len = min(min_len, i - start_index + 1)
                # Increase the count of the character's count when removing it from the window.
                char_count[s[start_index]] += 1
                # Move the window to the right.
                start_index += 1

        # Return the minimum length of the window we found.
        return min_len

```

Java Solution

```java
class Solution {
    public int balancedString(String s) {
        // Initialize the count array to keep track of the occurrences of 'Q', 'R', 'E', 'W'
        int[] count = new int[4];
        String characters = "QWER"; // This will be used to map characters to indices in count
        int stringLength = s.length(); // Store the length of the string

        // Count the occurrences of each character in the string
        for (int i = 0; i < stringLength; ++i) {
            count[characters.indexOf(s.charAt(i))]++;
        }

        // Calculate the expected count of each character if the string was balanced
        int expectedCount = stringLength / 4;

        // If all characters count are already equal to the expectedCount, the string is already balanced
        if (count[0] == expectedCount && count[1] == expectedCount && count[2] == expectedCount && count[3] == expectedCount) {
            return 0;
        }

        // Initialize the minimum length of the substring to be replaced
        int minSubstringLength = stringLength;

        // Use two pointers to find the minimum window that when replaced makes the string balanced
        for (int start = 0, end = 0; start < stringLength; ++start) {
            // Decrement the count of the current character as it is to be included in the window
            count[characters.indexOf(s.charAt(start))]--;

            // While the end pointer is before the start pointer and all characters are within or below the expected count
            while (end <= start && count[0] <= expectedCount && count[1] <= expectedCount && count[2] <= expectedCount && count[3] <=
                // Update the minimum substring length if the current window is smaller
                minSubstringLength = Math.min(minSubstringLength, start - end + 1);

                // Move the end pointer to the right and adjust counts as these characters are removed from the window
                count[characters.indexOf(s.charAt(end++))]++;
            }
        }

        // Return the minimum length of substring found that could be replaced to balance the string
        return minSubstringLength;
    }
}

```

C++ Solution

```cpp
#include <string>
#include <algorithm>
using namespace std;

class Solution {
public:
    int balancedString(string s) {
        int charCount[4] = {}; // array to keep count of Q, W, E, R characters
        string characters = "QWER"; // the characters to check in the string
        int stringLength = s.size(); // get the size of the string
        // Count occurrences of 'Q', 'W', 'E', and 'R'.
        for (char& ch : s) {
            charCount[characters.find(ch)]++;
        }

        int target = stringLength / 4; // calculate target count of each char for balance

        // Check if the string is already balanced
        if (charCount[0] == target && charCount[1] == target &&
            charCount[2] == target && charCount[3] == target) {
            return 0; // the string is already balanced
        }

        int answer = stringLength; // initialize answer to max possible length
        // Loop to find the minimum length substring that can be replaced to balance
        for (int start = 0, end = 0; start < stringLength; ++start) {
            charCount[characters.find(s[start])]--; // include s[start] in the sliding window

            // Loop to shrink the window size from left if balanced
            while (end <= start && charCount[0] <= target && charCount[1] <= target &&
                   charCount[2] <= target && charCount[3] <= target) {
                answer = min(answer, start - end + 1); // update the answer with smaller length
                // remove s[end] from sliding window and increase to narrow down the window
                charCount[characters.find(s[end++])]++;
            }
        }

        return answer; // return minimum length of substring to be replaced
    }
};

```

Typescript Solution

```typescript
// Import required items from the standard library
import { min } from 'lodash';

// Count occurrences of each character in the given string.
function countCharacters(s: string): number[] {
    // Initialize counts of 'Q', 'W', 'E', 'R' as an array.
    const charCount = [0, 0, 0, 0];
    const characters = 'QWER'; // Characters to check in the string
    // Loop through each character in the string to count them
    for (const ch of s) {
        charCount[characters.indexOf(ch)]++;
    }
    return charCount;
}

// Determine the minimum length of substring to replace to balance the string.
function balancedString(s: string): number {
    const charCount = countCharacters(s); // Get the count of each character
    const stringLength = s.length; // Get the size of the string
    const target = stringLength / 4; // Calculate target count of each character for balance

    // Check if the string is already balanced
    if (charCount.every(count => count === target)) {
        return 0; // The string is already balanced
    }

    let answer = stringLength; // Initialize answer to max possible length

    // Loop to find the minimum length substring that can be replaced to balance
    for (let start = 0, end = 0; start < stringLength; ++start) {
        // Include s[start] in the sliding window and decrement the count
        charCount[characters.indexOf(s[start])]--;

        // Shrink the window size from the left if balanced
        while (end <= start && charCount.every(count => count <= target)) {
            answer = min([answer, start - end + 1]); // Update the answer with smaller length
            // Remove s[end] from sliding window and increase to narrow down the window
            charCount[characters.indexOf(s[end++])]++;
        }
    }

    return answer; // Return the minimum length of substring to be replaced
}

```

Time and Space Complexity

The given Python code defines a method `balancedString` within a `Solution` class, which aims to find the minimum length of a substring to replace in order to make the frequency of each character in the string `s` no more than `n / 4` where `n` is the length of `s`. The code implements a sliding window algorithm alongside the `Counter` object to track character frequencies.

Time Complexity:

Let's consider the time complexity of the code:

- Initializing the `Counter` object `cnt` with string `s` has `O(n)` complexity where `n` is the length of the string since each character is visited once.
- Checking if all character frequencies are less than or equal to `n // 4` takes `O(1)` time since there are only 4 types of characters and this check is done in constant time.
- The outer for-loop runs `n` times as it loops over each character in `s`.
- The worst-case scenario for the inner while-loop is when it also runs `n` times, for instance, when all the characters are the same at the beginning of the string. However, it's worth noting that each character in the string `s` will be added and removed from the `cnt` exactly once due to the nature of the sliding window. Hence, the pair of loops has a combined complexity of `O(n)`, not `O(n^2)`.

So, the overall time complexity of the code is `O(n)`.

Space Complexity:

Analyzing the space complexity:

- The `Counter` object `cnt` stores at most 4 key-value pairs, one for each unique character, which is a constant `O(1)`.
- Additional variables like `ans`, `j`, and `i` use `O(1)` space as well.

This means the space complexity of the code is `O(1)`, as the space used does not scale with the size of the input string `s`.