2516. Take K of Each Character From Left and Right

Problem Description

You are presented with a string `s` that contains only the characters 'a', 'b', and 'c', and a non-negative integer `k`. The task is to determine the minimum number of minutes required to take at least `k` instances of each character from the string. You can only remove one character per minute, and it must be either the first (leftmost) or the last (rightmost) character of the string. If it's not possible to remove enough characters to meet the condition (at least `k` of each character), then the function should return `-1`.

Intuition

To solve this problem using an efficient approach, we can apply the sliding window technique. The main idea behind the solution is to find the smallest window in the original string, which, when removed, still leaves at least `k` instances of each character 'a', 'b', and 'c'. Then, the number of minutes required would be equal to the length of the original string minus the size of this smallest window.

The process of devising this solution includes:

1. **Character Counting**: Initially, we count the occurrences of each character ('a', 'b', 'c') in the string. If any character does not have at least `k` instances, it's immediately impossible to accomplish the task, and we return `-1`.
2. **Finding the Smallest Window**: Next, we iterate through the string to determine the smallest window that can be removed. As we iterate with a pointer `i`, we decrease the count of the current character, reflecting that we've taken one instance of it from either end. We use another pointer `j` to represent the start of the window we want to remove. If after removing a character at `i`, we find that any character's count is below `k`, we move the pointer `j` forward to add characters back to the window, until each character's count is at least `k` again.
3. **Tracking the Minimum Minutes**: After we find such a window where the character counts outside the window are sufficient, we calculate its size and update the minimum number of minutes if this window is smaller than the previous ones found. The maximum size of the window we want to remove is noted.

At the end of the iteration, the length of the string minus the maximum size of the window we plan to remove will give us the minimum number of minutes needed. We return this value.

Solution Approach

The solution follows a sliding window approach and uses Python's `Counter` class from the `collections` module to keep track of the frequency of each character in the string `s`. This data structure is essential for efficiently updating and querying the count of the letters 'a', 'b', and 'c'.

Algorithm:

1. **Initialize the Counter**: Create a Counter object to store the occurrences of each character by passing the string `s` to it.
2. **Check Possibility**: Loop through the character counts. If the count of any of 'a', 'b', or 'c' is less than `k`, return `-1` since it's impossible to satisfy the condition.
3. **Initialize Pointers and Answer**: Set up two pointers, `i` and `j`, to iterate through the string. `i` will go from the start to the end of the string, while `j` will mark the beginning of the sliding window. Initialize `ans` to 0, which will keep track of the maximum window size found that maintains at least `k` occurrences of each character outside the window.
4. **Iterate and Find Window**: Loop through the string using the pointer `i`:
   - Decrease the count of the character `s[i]` from the `Counter`, as removing the character `s[i]` from either end means it's no longer part of the string outside the window.
   - If the count of the current character `s[i]` after decreasing becomes less than `k`, it means we need to move the other pointer `j` forwards to expand the sliding window and add characters back to the count, until all counts are again at least `k`.
   - During this adjustment, if `s[j]` is added back to the window, increase its count in the `Counter` and move `j` forward by 1.
   - Ensure that through these adjustments, the count of every character outside the window remains at least `k`.
5. **Calculate Maximum Window**: After each iteration, update `ans` with the size of the current window if it is larger than the previous `ans`. This window represents the maximum size that left at least `k` occurrences of each character outside.
6. **Result**: After completing the loop, the answer will be the total length of the string minus the maximum window size that we can remove while leaving at least `k` occurrences of each character. This gives us the minimum number of minutes needed, which we return.

The solution's effectiveness stems from its `O(n)` complexity where `n` is the length of the string, since each character is processed at most twice: once when increasing the window size (`j` moves forward) and once when decreasing (`i` moves forward).

Example Walkthrough

Let's use the problem approach to solve a small example where we have the string `s = "abcbab"` and `k = 2`.

1. **Initialize the Counter**:
   - Create a Counter from the string `s`. The Counter will look like this:
     ```
     {'a': 2, 'b': 3, 'c': 1}
     ```
2. **Check Possibility**:
   - Since `'c'` only has one instance and `k` is 2, it's not possible to have at least `k` instances of `'c'` remaining after any removals. Therefore, we immediately return `-1`.

Now let's consider a different string, `s = "abbcabc"` with `k = 2`. Starting over with the steps:

1. **Initialize the Counter**:
   - The Counter will look like this:
     ```
     {'a': 2, 'b': 3, 'c': 2}
     ```
2. **Check Possibility**:
   - Every character `'a'`, `'b'`, and `'c'` occurs at least `k` times, so it's possible to proceed.
3. **Initialize Pointers and Answer**:
   - `i = 0, j = 0, ans = 0` (maximum window size to remove is 0 so far).
4. **Iterate and Find Window**:
   - As we iterate with `i` from 0 to the end of the string, we decrease the count from the Counter when removing `s[i]` from consideration.
   - Here's what happens as `i` progresses:
     - `i = 0`: Remove `'a'`, Counter becomes `{'a': 1, 'b': 3, 'c': 2}`. No need to adjust `j`, as all counts are still at least `k`.
     - `i = 1`: Remove `'b'`, Counter becomes `{'a': 1, 'b': 2, 'c': 2}`. No need to adjust `j`, only when a letter goes below `k` do we adjust `j`.
     - `i = 2`: Remove `'b'`, Counter becomes `{'a': 1, 'b': 1, 'c': 2}`.
     - Now, removing the next character at `i` will make the count of `'b'` go below `k`, so we start moving `j` forward to find the maximum window we can remove:
       - `j = 0`: Adding `'a'` back into the window, Counter becomes `{'a': 2, 'b': 1, 'c': 2}`.
       - `j = 1`: We cannot move `j` forward as it would decrease the count of `'b'` which is already at minimum allowed for `k`.
     - By `i = 3`, the count of `'b'` can't be allowed to decrease any further without also decreasing `j`.
5. **Calculate Maximum Window**:
   - The window size with `i = 3` and `j = 1` is `3 - 1 = 2`. Update `ans` to 2.
6. **Result**:
   - Continue iterating with the same strategy till the end. After completing the loop, we find that the maximum window size to remove is `ans = 3` (which happens to be the window `"bca"` from index 2 to 4). The string length is 7, the answer is `7 - 3 = 4`. Thus, it takes a minimum of 4 minutes to remove characters such that at least `k` instances of each remain.

The process shows that by using the sliding window method, we managed to find the minimum number of minutes required to remove characters from the string `s` while maintaining at least `k` instances of each character.

Python Solution

```python
from collections import Counter

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

        # Check if the count of any character 'a', 'b', or 'c' is less than k
        if any(char_count[char] < k for char in "abc"):
            return -1  # If so, return -1 as we cannot fulfill the requirement

        # Initialize answer and the start index to 0
        max_length = start_index = 0

        # Iterate over the string with index i and character c
        for i, c in enumerate(s):
            char_count[c] -= 1  # Decrement the count of the current character

            # If count of current character is less than k, move the start index forward
            # to find a valid substring
            while char_count[c] < k:
                char_count[s[start_index]] += 1  # Increment the character at start index
                start_index += 1  # Move the start index forward

            # Update max_length to the maximum length found so far
            max_length = max(max_length, i - start_index + 1)

        # Return the length of the string minus the maximum length of a valid substring
        return len(s) - max_length
```

Java Solution

```java
class Solution {
    /**
     * This method calculates the minimum number of characters to remove from the string so that each character
     * appears at least 'k' times in the remaining string.
     *
     * @param s The input string.
     * @param k The minimum number of times each character should appear.
     * @return The minimum number of characters to remove, or -1 if it's not possible.
     */
    public int takeCharacters(String s, int k) {
        int[] charCounts = new int[3]; // Array to store the count of each character 'a', 'b', and 'c'.
        int length = s.length(); // The length of the input string.

        // Count occurrences of each character ('a', 'b', 'c').
        for (int i = 0; i < length; ++i) {
            // Decrement the count for the current character.
            charCounts[s.charAt(i) - 'a']--;
        }

        // Check if any character appears less than 'k' times.
        for (int count : charCounts) {
            if (count < k) {
                return -1; // It's not possible to fulfill the condition.
            }
        }

        int maxSubstringLength = 0; // Will hold the length of the longest valid substring.
        int leftPointer = 0; // Points to the start of the current substring.

        // Iterate over the characters of the string.
        for (int rightPointer = 0; rightPointer < length; ++rightPointer) {
            int currentCharIndex = s.charAt(rightPointer) - 'a'; // Index of the current character.
            // Decrement the count of the current character since we're moving rightPointer.
            charCounts[currentCharIndex]--;

            // If count of the current character falls below 'k', readjust the leftPointer.
            while (charCounts[currentCharIndex] < k) {
                // Increment count of the character at leftPointer as it's no longer in current substring.
                charCounts[s.charAt(leftPointer) - 'a']++;
                // Move leftPointer to the right.
                leftPointer++;
            }

            // Update the length of the longest valid substring found so far.
            maxSubstringLength = Math.max(maxSubstringLength, rightPointer - leftPointer + 1);
        }

        // Return the minimum number of characters to remove, i.e., total length minus length of longest valid substring.
        return length - maxSubstringLength;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int takeCharacters(string s, int k) {
        int charCounts[3] = {0}; // Array to store counts of 'a', 'b', and 'c'

        // Count the number of occurrences of each character
        for (char& c : s) {
            ++charCounts[c - 'a'];
        }

        // If any character appears less than k times, we cannot proceed
        if (charCounts[0] < k || charCounts[1] < k || charCounts[2] < k) {
            return -1;
        }

        int maxSubstringLength = 0; // To keep track of the maximum valid substring length
        int windowEnd = 0; // Right boundary of the current sliding window
        int stringSize = s.size(); // Size of input string

        // Iterate over each character in the string
        for (int windowStart = 0; windowEnd < stringSize; ++windowEnd) {
            int currentCharIndex = s[windowEnd] - 'a'; // Get the index of current character (0, 1, or 2)
            --charCounts[currentCharIndex]; // Count down the current character

            // If the current character's count is below k, slide the window
            while (charCounts[currentCharIndex] < k) {
                ++charCounts[s[windowStart++] - 'a']; // Increment the count of the char getting out of the window
            }

            // Calculate the maximum valid substring length seen so far
            maxSubstringLength = max(maxSubstringLength, windowEnd - windowStart + 1);
        }

        // The result is the size of the string minus the maximum valid substring length
        return stringSize - maxSubstringLength;
    }
};
```

Typescript Solution

```typescript
/**
 * Calculates the minimum number of characters that must be removed from a string
 * so that every character that appears in the string does so both k times.
 *
 * @param {string} str - The input string.
 * @param {number} k - The target frequency for each character.
 * @return {number} - The minimum number of characters to remove or -1 if not possible.
 */
function takeCharacters(str: string, k: number): number {
    // Helper function to convert a character to an index (0 for 'a', 1 for 'b', etc.).
    const getIndex = (char: string) => char.charCodeAt(0) - 'a'.charCodeAt(0);
    // Array to count the frequency of each character, assuming only lowercase letters.
    const charFrequency = new Array(26).fill(0);
    // Populate the frequency count array with the occurrences of each character in the input string.
    for (const char of str) {
        ++frequencyCount[getIndex(char)];
    }

    // If any character frequency is less than k, it's impossible to satisfy the condition, return -1.
    if (frequencyCount.some(value => value < k)) {
        return -1;
    }

    // Length of the input string.
    const stringLength = str.length;
    // Variable to store the maximum substring length satisfying the condition).
    let maxSubstringLength = 0;
    // Two pointers i and j to apply the sliding window technique.
    for (let i = 0, j = 0; i < stringLength; ++i) {
        // Decrease the frequency count of the j-th character as it's now outside the window.
        --frequencyCount[getIndex(str[i])];
        // Move the pointer j forward while the current character's frequency is below k.
        while (frequencyCount[getIndex(str[i])] < k) {
            // Increase the frequency count of the i-th character as it's now included in the window.
            ++frequencyCount[getIndex(str[j++])];
        }
        // Calculate the maximum substring length by comparing with the current window size.
        maxSubstringLength = Math.max(maxSubstringLength, j - i + 1);
    }
    // The answer is the difference between the string's length and the maximum substring length.
    return stringLength - maxSubstringLength;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is dictated by the `for` loop, which iterates through the string `s`; this operation is `O(n)`, where `n` is the length of the string. Inside the loop, the code performs constant-time operations, such as dictionary access, update, and comparison. However, the inner `while` loop might seem to increase time complexity because it moves the index `j` forward until a certain condition is met. But notice that `j` can never be moved more than `n` times over the entire execution of the outer loop, since each character is considered exactly once by both the outer loop and the inner `while` loop across the entire runtime. Therefore, the time complexity remains `O(n)` overall.

Space Complexity

The space complexity is governed by the `Counter` object `cnt` that stores the frequency of each character in the string. Since the number of distinct characters is constrained by the size of the alphabet in the conditional check `if any(cnt[c] < k for c in "abc")`, the size of the counter will be `O(1)`, as the alphabet size does not grow with the size of the input string. Hence, the overall space complexity remains constant, `O(1)`.