1156. Swap For Longest Repeated Character Substring

Medium Hash Table String Sliding Window

Problem Description

In this problem, you have a string `text` which consists of various characters. Your goal is to determine the length of the longest substring where all the characters are the same. Substring, in this context, refers to a sequence of characters that appear consecutively in the string.

However, there's an added twist to the problem. You are permitted to perform one operation, which is to swap any two characters in the string. This could potentially increase the length of the longest homogenous substring if it brings another matching character adjacent to it.

You need to calculate the length of the longest substring of repeating characters after performing at most one such swap.

Intuition

To solve this problem, we keep track of the frequency of each character in the original string, since it will be important to know if we have extra characters that we can bring to our current substring after a swap.

The main idea is to iterate through the string, and for each character, find the length of the maximum substring ending with that character which we might extend via a swap. We do this by checking the immediate substring made up of the same character, then look ahead to see if there's another block of the same character after a different one (`k` steps away where `k`>1).

For instance, in a string like `aabaa`, the immediate substring is `aa`, and the block after the next different character is `aa`. If we include one of the latter `a` characters by swapping, we can extend our substring by 1 character.

To ensure we don't consider an unavailable character for a swap, we take the minimum of our potentially extended substring length and the total count of the current character. Ultimately, we want to find the maximum length from all the possible substrings that could be formed after doing such operations.

Solution Approach

The implementation uses the following steps and data structures:

1. **Counter Data Structure**: First, we use the `Counter` class from Python's `collections` module to keep track of the frequency of each character in the input string `text`. This helps us to know the total occurrences of any character in the string which is essential for determining the maximum possible length of the substring after a swap.
2. **Two Pointer Approach**: The code then uses a two-pointer technique to iterate through the characters of the string. The index `i` represents the start of a sequence of identical characters, and `j` is used to find the end of this sequence.
3. **Finding Substrings**: For every new character that pointer `i` encounters, pointer `j` moves forward to find where the sequence of the same character ends. The length of the immediate sequence is `l = j - i`.
4. **Looking Ahead**: After `j` has found the end of the immediate sequence, the code looks ahead to see if there are other sequences of the same character separated by a different character. The index `k` is used to find the end of the next sequence of the same character, and we calculate the length of this secondary sequence as `r = k - j - 1`.
5. **Calculating Potential Swaps**: The total length of such two sequences combined potentially could be `l + r + 1`, counting in the swap. If there's an additional character available in the text of the same type, we would be able to make this longer substring continuous by swapping in the extra character.
6. **Ensuring Valid Swaps**: Since we can only swap in a character if an extra one is available, we take the minimum of `l + r + 1` and `cnt[text[i]]` (the count of the current character), to update the `ans` which keeps track of the longest valid substring we can form.
7. **Updating Answer**: The variable `ans` is updated with the maximum value between what it previously was and the length we just computed.
8. **Moving to Next Sequence**: Finally, the pointer `i` is set to the end of the immediate sequence, marked by `j`, to start checking for the next sequence in the string.

By using the above steps, we eventually return the maximum length of a homogenous substring that can be achieved with or without one swap, which is stored in `ans`.

Example Walkthrough

Let's consider a simple example with the string `text = "aabbaa"` to illustrate the solution approach step by step:

1. **Counter Data Structure**: We use a `Counter` to count the occurrences of each character.
   - `Counter({'a': 4, 'b': 2})` - There are four 'a' characters and two 'b' characters.
2. **Two Pointer Approach**: Set two pointers initially at the start of the string, `i = 0` and `j = 0`.
3. **Finding Substrings**: The first character is 'a', so we move `j` ahead to find all consecutive 'a's.
   - `i = 0` and `j = 1` - Found the substring "aa".
4. **Looking Ahead**: Now we skip the different characters 'b' to find the next sequence of 'a's.
   - `k = 4` - `k` is now at the start of the second sequence of 'a's.
5. **Calculating Potential Swaps**: The concatenated length by adding one 'a' from the next block would be `l + r + 1` = `2 (from "aa") + 2 (from "aa" after 'b's) + 1` = `5`.
6. **Ensuring Valid Swaps**: We have four 'a's available (`Counter('a') = 4`), so we can do this swap. The maximum length for 'a' becomes `min(5, 4) = 4`.
7. **Updating Answer**: `ans` is set to `4` as it's the longest substring recorded so far.
8. **Moving to Next Sequence**: Advance `i` past the 'b's, to the start of the next 'a' sequence.

Repeating the steps for the 'b' characters:

1. `i = 2` - Pointing to the first 'b' now.
2. `j = 3` - We have one sequence of 'b's, "bb".
3. There's no additional sequence of 'b's ahead, so we don't move `k`.
4. **Calculating Potential Swaps**: For 'b', the maximum length by swapping an extra 'b' would be `l + 1` = `2 (from "bb") + 1` = `3`.
5. **Ensuring Valid Swaps**: We check with `Counter('b') = 2`, but we have no extra 'b' to swap. So the length remains `2`.
6. `ans` remains `4`, as no longer sequence was found.
7. There are no more new sequences to explore.

Thus, the final answer is `4`, which corresponds to the longest possible substring of repeating 'a's after performing at most one swap.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def maxRepOpt1(self, text: str) -> int:
        char_count = Counter(text)  # Count the frequency of each character in the given text
        text_length = len(text)     # Length of the given text
        max_length = current_index = 0  # Initialize max_length and current_index to zero

        # Iterate through the text to find the maximum length of a repeating character substring
        while current_index < text_length:
            start_index = current_index
            # Find the end of the current sequence of same characters
            while start_index < text_length and text[start_index] == text[current_index]:
                start_index += 1
            # Calculate the length of this sequence
            left_sequence_length = start_index - current_index

            # Find the next sequence of the same character after a different one
            next_index = start_index + 1
            while next_index < text_length and text[next_index] == text[current_index]:
                next_index += 1
            # Calculate the length of the second sequence
            right_sequence_length = next_index - start_index - 1

            # Calculate the maximum length by using the sequences found and the overall character count
            # We can insert at most one character from other parts of the string
            total_length = min(left_sequence_length + right_sequence_length + 1, char_count[text[current_index]])

            # Update the max_length if the current total_length is greater
            max_length = max(max_length, total_length)

            # Move the current_index to the end of the first sequence
            current_index = start_index

        return max_length  # Return the maximum length found
```

Java

```java
class Solution {

    // Method to find the maximum length of a substring where one character can be replaced to maximize the length
    public int maxRepOpt1(String text) {
        int[] charCount = new int[26]; // Array to store the count of each character in the text
        int textLength = text.length();

        // Count the occurrences of each character
        for (int i = 0; i < textLength; ++i) {
            charCount[text.charAt(i) - 'a']++;
        }

        int maxLen = 0; // Variable to store the maximum length found
        int index = 0; // Index to iterate over the text

        // Iterate over the text to find the maximum length sequence
        while (index < textLength) {
            int endIndex = index;

            // Find the end index of the current sequence of the same character
            while (endIndex < textLength && text.charAt(endIndex) == text.charAt(index)) {
                ++endIndex;
            }

            int sequenceLength = endIndex - index; // Length of the continuous sequence
            int nextIndex = endIndex + 1;

            // Skip one different character and continue with the same character if possible
            while (nextIndex < textLength && text.charAt(nextIndex) == text.charAt(index)) {
                ++nextIndex;
            }

            int nextSequenceLength = nextIndex - endIndex - 1; // Length of the next sequence of the same character

            // Update maxLen with the higher value between current maxLen and the possible maximum length
            // by replacing one character from the sequence
            maxLen = Math.max(maxLen, Math.min(sequenceLength + nextSequenceLength + 1, charCount[text.charAt(index) - 'a']));

            index = endIndex; // Move the index to the end of the current sequence
        }

        return maxLen; // Return the maximum length found
    }
}
```

C++

```cpp
class Solution {
public:
    int maxRepOpt1(string text) {
        // Initialize an array to store the frequency of each character 'a' to 'z' in the text
        int charFreq[26] = {0};
        // Count the frequency of each character
        for (char c : text) {
            ++charFreq[c - 'a'];
        }

        int textLength = text.size();
        int maxRepeat = 0; // To store the maximum repeat length
        int index = 0; // Index to iterate over the string

        // Loop through each character in the text
        while (index < textLength) {
            // Find the sequence length of same characters starting at index 'i'
            int sameCharEndIndex = index;
            while (sameCharEndIndex < textLength && text[sameCharEndIndex] == text[index]) {
                ++sameCharEndIndex;
            }
            // Length of the sequence of the same characters
            int sequenceLength = sameCharEndIndex - index;
            // Try finding the next sequence of the same character after a different one
            int nextCharIndex = sameCharEndIndex + 1;
            while (nextCharIndex < textLength && text[nextCharIndex] == text[index]) {
                ++nextCharIndex;
            }
            // Length of the next sequence of the same character
            int nextSequenceLength = nextCharIndex - sameCharEndIndex - 1;
            // Calculate the max repeated length considering swapping one different char between sequences
            // Also, ensure that we do not count more than the total occurrences of the character in the text
            maxRepeat = max(maxRepeat, min(sequenceLength + nextSequenceLength + 1, charFreq[text[index] - 'a']));
            // Move to the next different character
            index = sameCharEndIndex;
        }
        // Return the maximum repeat length found
        return maxRepeat;
    }
};
```

TypeScript

```typescript
function maxRepOpt1(text: string): number {
    // Function to get the index of the character in the alphabet (0-based)
    const getIndex = (char: string) => char.charCodeAt(0) - 'a'.charCodeAt(0);

    // Initialize an array to hold the count of each character in the text
    const charCount: number[] = new Array(26).fill(0);

    // Counting occurrences of each character
    for (const char of text) {
        charCount[getIndex(char)]++;
    }

    let maxRepeat = 0; // Variable to store the maximum repeat length
    let i = 0; // Start index of the current sequence
    const textLength = text.length;

    // Iterate over the text to find repeat sequences
    while (i < textLength) {
        let j = i; // End index of the current sequence
        // Expand the sequence while the character is the same
        while (j < textLength && text[j] === text[i]) {
            ++j;
        }
        const currentLength = j - i; // Calculate the length of the current sequence

        // Look ahead for the next sequence of the same character
        let k = j + 1;
        while (k < textLength && text[k] === text[i]) {
            ++k;
        }
        const nextLength = k - j - 1; // Calculate the length of the next sequence

        // Calculate the maximum possible length by combining the two sequences
        // and possibly one character change if available
        maxRepeat = Math.max(maxRepeat, Math.min(charCount[getIndex(text[i])], currentLength + nextLength + 1));
        i = j; // Move to the next sequence
    }

    return maxRepeat; // Return the maximum repeat length
}
```


```python
from collections import Counter

class Solution:
    def maxRepOpt1(self, text: str) -> int:
        char_count = Counter(text)  # Count the frequency of each character in the given text
        text_length = len(text)     # Length of the given text
        max_length = current_index = 0  # Initialize max_length and current_index to zero

        # Iterate through the text to find the maximum length of a repeating character substring
        while current_index < text_length:
            start_index = current_index
            # Find the end of the current sequence of same characters
            while start_index < text_length and text[start_index] == text[current_index]:
                start_index += 1
            # Calculate the length of this sequence
            left_sequence_length = start_index - current_index

            # Find the next sequence of the same character after a different one
            next_index = start_index + 1
            while next_index < text_length and text[next_index] == text[current_index]:
                next_index += 1
            # Calculate the length of the second sequence
            right_sequence_length = next_index - start_index - 1

            # Calculate the maximum length by using the sequences found and the overall character count
            # We can insert at most one character from other parts of the string
            total_length = min(left_sequence_length + right_sequence_length + 1, char_count[text[current_index]])

            # Update the max_length if the current total_length is greater
            max_length = max(max_length, total_length)

            # Move the current_index to the end of the first sequence
            current_index = start_index

        return max_length  # Return the maximum length found
```


Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input string text. This is because the main while loop iterates over each character of the string at most twice – when counting the consecutive occurrences (`j` loop) and when checking for a single separated character (from `j` to `k`). No nested iterations with dependence on `n` are present that would increase the time complexity to `O(n^2)` or higher.

The space complexity of the code is `O(1)` or `O(min(n, 26))` to be more specific, considering that `Counter(text)` creates a counter collection for the distinct characters. In the worst case for English alphabet input, that would be 26 letters, which is a constant and does not scale with `n`. Therefore, we typically consider this as constant space complexity.