424. Longest Repeating Character Replacement

Medium Hash Table String Sliding Window

Problem Description

In this problem, you're provided with a string `s` and an integer `k`. You are allowed to perform at most `k` operations on the string. In each operation, you may choose any character in the string and change it to any other uppercase English letter. The objective is to find the length of the longest substring (that is a sequence of consecutive characters from the string) that contains the same letter after you have performed zero or more, up to `k`, operations.

For example, given the string `"AABABBA"` and `k = 1`, you are allowed to change at most one character. The best way is to change the middle 'A' to 'B', resulting in the string "AABBBBA". The longest substring with identical letters is "BBBB", which is 4 characters long.

Intuition

The intuition behind the solution involves a common technique called the "sliding window". The core idea is to maintain a window (or a subrange) of the string and keep track of certain attributes within this window. As the window expands or contracts, you adjust these attributes and check to see if you can achieve a new maximum.

The solution keeps track of:

1. **The frequency of each letter within the window**: This is kept in an array called `counter`. Each index of this array corresponds to a letter of the English alphabet.
2. **The count of the most frequent letter so far**: During each step, the code calculates the current window's most frequent letter. This is stored in the variable `maxCnt`.
3. **The indices `i` (end) and `j` (start) of the window**.

The approach is as follows:

- Expand the window by moving `i` to the right, incrementing the counter of the current character.
- Update the `maxCnt` with the maximum frequency of the current character.
- Check if the current window size is greater than the sum of `maxCnt` (the most frequent character count) and `k`. If it is, then that means the current window cannot be made into a substring of all identical letters with at most `k` operations. If this happens, decrease the count of the leftmost character and move the start of the window to the right (increment `j`).
- Repeat this process until you have processed the entire string.
- Since the window size only increases or remains unchanged over time (because we only move `i` to the right and increment `j` when necessary), the final value of `i - j` when `i` has reached the end of the string will be the size of the largest window we were able to create where at most `k` operations would result in a substring of identical letters.

By the time the window has moved past all characters in `s`, you've considered every possible substring and the maximum viable window size is the length of the longest substring satisfying the condition. Hence the answer is `i - j`. This is an application of the two-pointer technique.

Solution Approach

To implement the solution, the approach capitalizes on several important concepts and data structures:

- **Sliding Window Technique**: This technique involves maintaining a window of elements and slides it over the data to consider different subsets of the data.
- **Two Pointers**: `i` and `j` are pointers used to represent the current window in the string, where `i` is the end of the window and `j` is the beginning.
- **Array for Counting**: An array `counter` of size 26 is used to keep count of all uppercase English letters within the current sliding window. Since there are only 26 uppercase English letters, it's efficient regarding both space and time complexity.

Here's a step-by-step of what happens in the code:

1. **Initialization**: Set up an array `counter` with length 26 to zero for all elements, representing the count of each letter. Initialize two pointers `i` and `j` to `0`, and `maxCnt` to `0` which will store the maximum frequency of a single letter within the current window.
2. **Sliding Window Expansion**: Iterate over the string using the pointer `i` to expand the window to the right. For each character `s[i]`, increment the count in the `counter` array at the position corresponding to the letter (found by `ord(s[i]) - ord('A')` where `ord` is a function that gets the ASCII value of a character).
3. **Updating Maximum Letter Count**: After updating `counter` for the new character, update `maxCnt` to reflect the maximum frequency of any single character in the current window.
4. **Exceeding the Operation Limit**: At each iteration, check if the current window size (`i - j + 1`) is greater than allowed (`maxCnt + k`). If it is, this means more than `k` replacements are required to make all characters in the current window the same. Therefore, you need to shrink the window by incrementing `j`, and decreasing the count of the character at the start of the window.
5. **Continue Until the End**: Keep repeating steps 2 to 4 until the end of the string is reached. At this point, since the window only grew or remained the same throughout the process, the difference `i - j` will be the length of the longest substring that can be achieved with at most `k` changes.

Using this approach, as you can see in the Python code, the function `characterReplacement` operates on the string efficiently by using a fixed amount of memory (the `counter` array) and makes a single pass over the string, thus the time complexity is O(n), where n is the length of the string.

Example Walkthrough

Let's walk through the solution approach using a small example: `s = "ABAA"` and `k = 1`.

1. **Initialization**: We start by initializing our `counter` array of size 26 to zero and the pointers `i` and `j` are both set to `0`. `maxCnt` is also initialized to `0`.
2. **Sliding Window Expansion**: Begin iterating through the string.
   - For the first character, 'A', `counter[A]` (consider `counter[0]` since 'A' is the first letter) is incremented to `1`. Now `maxCnt` also becomes `1`, as the only character in the window is 'A'.
   - Move `i` to the right to point to the second character `s[1]` which is 'B'. Increment `counter[B]` (consider `counter[1]` since 'B' is the second letter). `maxCnt` remains `1` because the frequency of both 'A' and 'B' is the same (`1`) in the window.
3. **Updating Maximum Letter Count**: As we continue, we update `counter` and `maxCnt`:
   - Increment `i` to point to `s[2]`, which is 'A'. Now `counter[A]` is `2`. We then update `maxCnt` to `2`, as 'A' is now the most frequent letter within the window.
4. **Exceeding the Operation Limit**: Continue expanding the window by moving `i` to the right:
   - Increment `i` to point to `s[3]`, which is 'A'. `counter[A]` is now incremented to `3`. The window size is `4` (from `s[0]` to `s[3]`), and since `maxCnt` is `3` and `k` is `1`, we satisfy the condition `(window size) <= (maxCnt + k)`, which is `4 <= (3 + 1)`.
5. **Shrinking the Window**: At this point, since we're at the end of the string, we stop and observe that we did not have to shrink the window at any point. The largest window we could form went from index `0` to index `3` with one operation allowed to change a 'B' to an 'A', resulting in all 'A's.

The longest substring that can be formed from string `"ABAA"` by changing no more than `1` letter is `"AAAA"`, which has a length of `4`. So, our output is `4`, which is the length from pointer `j` to `i`.

Using this step-by-step walkthrough, it is evident that this approach is both systematic and efficient in determining the length of the longest substring where at most `k` changes result in a uniform string.

Python Solution

```python
class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        # Initialize the frequency counter for the 26 letters of the alphabet
        frequency_counter = [0] * 26
        # Initialize pointers for the sliding window
        left = right = 0
        # Variable to keep track of the count of the most frequent character
        max_frequency = 0

        # Iterate over the characters in the string
        while right < len(s):
            # Update the frequency of the current character
            frequency_counter[ord(s[right]) - ord('A')] += 1
            # Find the maximum frequency count so far
            max_frequency = max(max_frequency, frequency_counter[ord(s[right]) - ord('A')])

            # Calculate the window size and compare it with the maximum frequency count and allowed replacements (k)
            if (right - left + 1) > max_frequency + k:
                # If the condition is true, decrement the frequency of the leftmost character
                # as it will be excluded from the current window
                frequency_counter[ord(s[left]) - ord('A')] -= 1
                # Shrink the window by moving the left pointer forward
                left += 1

            # Move the right pointer forward to expand the window
            right += 1

        # Return the maximum length of the window
        return right - left

```

Java Solution

```java
class Solution {
    public int characterReplacement(String s, int k) {
        int[] letterCount = new int[26]; // Array to store the frequency count of each letter
        int windowStart = 0; // Start index of the sliding window
        int windowEnd = 0; // End index of the sliding window
        int maxCountInWindow = 0; // Variable to store the maximum count of a single character in the current window

        // Iterate over the string with windowEnd serving as the end pointer of the sliding window
        for (; windowEnd < s.length(); ++windowEnd) {
            char currentChar = s.charAt(windowEnd); // Current character in iteration
            letterCount[currentChar - 'A']++; // Increment the count for this character in the frequency array

            // Update the maxCountInWindow to be the max between itself and the count of the current character
            maxCountInWindow = Math.max(maxCountInWindow, letterCount[currentChar - 'A']);

            // Check if current window size minus max frequency count is greater than k
            // If it is, we need to slide the window ahead while decrementing the count of the char at windowStart
            if (windowEnd - windowStart + 1 - maxCountInWindow > k) {
                letterCount[s.charAt(windowStart) - 'A']--; // Decrement count of the start character of the window
                windowStart++; // Move the window's start index forward
            }
        }
        // The maximum length substring is the size of the window on loop exit
        return windowEnd - windowStart;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int characterReplacement(string s, int k) {
        vector<int> charCount(26, 0); // Counter for each letter's frequency within the sliding window
        int left = 0; // Left index of the sliding window
        int right = 0; // Right index of the sliding window
        int maxCharCount = 0; // Variable to keep track of the count of the most frequent character within the window

        // Iterate over the characters of the string
        for (right = 0; right < s.size(); ++right) {
            charCount[s[right] - 'A']++; // Increment the count for the current character

            // Update the max frequency character count seen so far in the current window
            maxCharCount = max(maxCharCount, charCount[s[right] - 'A']);

            // Check if the current window size minus the count of the max frequency character
            // is greater than k, if so, shrink the window from the left
            if (right - left + 1 - maxCharCount > k) {
                charCount[s[left] - 'A']--; // Decrement the count for the character at the left index as it's going out of the windc
                left++; // Shrink the window from the left
            }
        }

        // The length of the largest window compliant with the condition serves as the answer
        return right - left;
    }
};

```

Typescript Solution

```typescript
// Counter for each letter's frequency within the sliding window
const charCount: number[] = new Array(26).fill(0);
// Left index of the sliding window
let left: number = 0;
// Right index of the sliding window
let right: number = 0;
// Variable to keep track of the count of the most frequent character within the window
let maxCharCount: number = 0;

/**
 * Method to find the length of the longest substring which can be made
 * by replacing at most k characters with any letter.
 *
 * @param {string} s - The input string to be processed
 * @param {number} k - The maximum number of characters that can be replaced
 * @returns {number} The maximum length of the substring
 */
function characterReplacement(s: string, k: number): number {
    // Reset variables for a new call
    charCount.fill(0);
    left = 0;
    right = 0;
    maxCharCount = 0;

    // Iterate over the characters of the string
    for (right = 0; right < s.length; ++right) {
        // Increment the count for the current character
        charCount[s.charCodeAt(right) - 'A'.charCodeAt(0)]++;

        // Update the max frequency character count seen so far in the current window
        maxCharCount = Math.max(maxCharCount, charCount[s.charCodeAt(right) - 'A'.charCodeAt(0)]);

        // Check if the current window size minus the count of the max frequency character
        // is greater than k. If so, shrink the window from the left.
        if (right - left + 1 - maxCharCount > k) {
            // Decrement the count for the character that is exiting the window
            charCount[s.charCodeAt(left) - 'A'.charCodeAt(0)]--;
            // Move the left pointer to shrink the window
            left++;
        }
    }

    // The length of the largest window compliant with the condition serves as the answer
    return right - left;
}

```

Time and Space Complexity

The given code implements a sliding window algorithm to find the longest substring that can be created by replacing at most `k` characters in the input string `s`.

Time Complexity:

The time complexity of the code is `O(n)`, where `n` is the length of the input string `s`. This is because:

- The algorithm uses two pointers `i` (end of the window) and `j` (start of the window) that move through the string only once.
- Inside the `while` loop, the algorithm performs a constant number of operations for each character in the string: updating the `counter` array, computing `maxCnt`, comparing window size with `maxCnt + k`, and incrementing or decrementing the pointers and counter.
- Although there is a `max` operation inside the loop which compares `maxCnt` with the count of the current character. This comparison takes constant time because `maxCnt` is only updated with values coming from a fixed-size array (the `counter` array with 26 elements representing the count of each uppercase letter in the English alphabet).
- No nested loops are dependent on the size of `s`, so the complexity is linear with the length of `s`.

Space Complexity:

The space complexity of the code is `O(1)`:

- The `counter` array uses space for 26 integers, which is a constant size and does not depend on the length of the input string `s`.
- Only a fixed number of integer variables (`i`, `j`, `maxCnt`) are used, which also contributes to a constant amount of space.

In conclusion, the algorithm runs in linear time and uses a constant amount of additional space.