767. Reorganize String

Medium Topics Hash Table String Counting Sorting Heap (Priority Queue) Leetcode Link

Problem Description

The problem states that we have a string `s`, and we need to rearrange its characters so that no two adjacent characters are the same. If it's possible to arrange the string to satisfy this condition, we should return any one of the valid rearrangements. If it is not possible, we should return an empty string `""`.

Intuition

The key insight to solve this problem stems from the observation that if a character appears more than half of the string's length (rounded up), it is impossible to rearrange the string so that no two adjacent characters are the same, because there would be insufficient gaps between instances of this character to place other characters.

Considering this, we can use a greedy approach with the following logic:

1. **Count Frequencies**: Count how many times each character appears in the given string. This will help us identify the most frequent characters which potentially could be a problem if they exceed the allowed limit.
2. **Check for Impossible Cases**: If the most frequent character occurs more than half the length of the string (rounded up), then it is impossible to rearrange the string in the required manner. In such a case, we immediately return an empty string.
3. **Construct the Solution**: If we can rearrange the string, we then fill the even indexes first (0, 2, 4 ...) with the most frequent characters. This ensures these characters are separated. If we reach the end of the string (going beyond the last index) in this process, we switch to the odd indexes (1, 3, 5 ...).
4. **Building the Output String**: Starting with the most common character, fill the string's indices as described. After placing all instances of the most common character, move to the next most common, and so forth, until the string is completely filled.

This algorithm efficiently ensures that for every character placed, it will not be adjacent to the same character, fulfilling the given problem constraint.

Solution Approach

The Reference Solution Approach uses a hashmap and a sorting technique to tackle the problem. The detailed steps of implementing the solution are as follows:

1. **Using a Counter**: The `Counter` class from Python's `collections` module is utilized to count occurrences of each character in the string. This creates a hashmap (a dictionary in Python) where keys are the characters and values are their counts.
2. **Determining the Maximum Frequency Character**: By finding the maximum value in the `Counter`, we determine if there is a character that appears more often than `(n + 1) // 2` times (`n` being the length of the string). If such a character exists, we return an empty string `""` since it's impossible to rearrange the string per the problem's condition.
3. **Creating the Answer Array**: An array `ans` is initialized having the same length as the string `s`. This will contain the rearranged characters and initially filled with `None`.
4. **Sorting by Frequency and Populating the Answer Array**: Using `most_common()` on the `Counter` object, we retrieve characters and their counts sorted by frequency in descending order. We then iterate over these key-value pairs.
5. **Placing Characters at Even Indices First**: We start filling the `ans` array at index `0` initialized to 0, which targets even indices. For each character `k`, we decrement its count `v` by 1 each time it's placed in the array, and increment `i` by 2 to move to the next even index.
6. **Switching to Odd Indices**: If `i` becomes equal to or greater than `n`, it means we've run out of even indices. Thus, we reset `i` to 1 to start filling odd indices.
7. **Building the Final String**: When the loop ends, all characters are distributed in the `ans` array in a way where no two identical characters are adjacent. We use `''.join(ans)` to convert the array back into a string and return that as our solution.

These steps ensure that the solution is both efficient and satisfies the problem's constraints, resulting in either a valid string rearrangement or an empty string if it's not possible.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Suppose we are given the string `s = "aabbcc"`. The string is 6 characters long, so no character should appear more than `6 / 2 = 3` times for us to be able to rearrange the characters as required.

Here's how we would apply our solution approach to this example:

1. **Using a Counter**: We utilize the `Counter` class to get the count of each character in the string `s`. The result will be a hashmap like this: `{'a': 2, 'b': 2, 'c': 2}`.
2. **Determining the Maximum Frequency Character**: The maximum count in our example is 2, which does not exceed `6 / 2 = 3`. This means it's possible to rearrange the string, so we don't need to return an empty string `""`.
3. **Creating the Answer Array**: We initialize an array `ans` of length 6 (since the string `s` has 6 characters), filled with `None`: `[None, None, None, None, None, None]`.
4. **Sorting by Frequency and Populating the Answer Array**: We sort the characters by their frequency. In our case, the counts are equal, but we proceeded with the available order: `['a', 'b', 'c']`.
5. **Placing Characters at Even Indices First**: Starting with `a` which has a count of 2, we place 'a' at index 0 and index 2: `[a, None, a, None, None, None]`.
6. **Switching to Odd Indices**: After we fill the even indices with `a`, we move to `b` and place it at indices 4 and then 1 because index 4 is still even and available: `[a, b, a, None, b, None]`.
7. **Continuing the Pattern**: Now we place `c` at the remaining indices, index 5 (which is the last even index) and then index 3: `[a, b, a, c, b, c]`.
8. **Building the Final String**: The `ans` array is now `[a, b, a, c, b, c]`, and no two identical characters are adjacent. Finally, we join the array into a string to get our result: `'abacbc'`.

Hence, the output is a valid rearrangement of the string `s` where no two adjacent characters are the same, demonstrating the effectiveness of the solution approach.

Python Solution

```python
from collections import Counter

class Solution:
    def reorganizeString(self, s: str) -> str:
        # Calculate the length of the string
        string_length = len(s)

        # Count the frequency of each character in the string
        char_count = Counter(s)

        # Find the maximum frequency of any character
        max_freq = max(char_count.values())

        # If the max frequency is more than half of the string length, round up,
        # then the task is impossible as that character would need to be adjacent to itself.
        if max_freq > (string_length + 1) // 2:
            return ''

        # Initialize index for placing characters
        index = 0

        # Create a list to store the reorganized string
        reorganized = [None] * string_length

        # Fill in the characters, starting with the most common
        for char, freq in char_count.most_common():
            while freq > 0:
                # Place the character at the current index
                reorganized[index] = char
                # Decrease the frequency count
                freq -= 1
                # Move to the next even index or the first odd index if the end is reached
                index += 2
                if index >= string_length:
                    index = 1

        # Return the list as a string
        return ''.join(reorganized)

```

Java Solution

```java
class Solution {
    public String reorganizeString(String s) {
        // Array to count the frequency of each character.
        int[] charCount = new int[26];
        int maxCount = 0; // Keep track of the maximum character frequency

        // Count the frequency of each character in the string.
        for (char character : s.toCharArray()) {
            int index = character - 'a';
            charCount[index]++;
            // Update maxCount if current character's frequency is higher.
            maxCount = Math.max(maxCount, charCount[index]);
        }

        int length = s.length();
        // If the most frequent character is more than half of the length of the string,
        // it is impossible to reorganize.
        if (maxCount > (length + 1) / 2) {
            return "";
        }

        int distinctChars = 0;
        // Count the number of distinct characters.
        for (int count : charCount) {
            if (count > 0) {
                distinctChars++;
            }
        }

        // Create a matrix to store frequency and index of each character.
        int[][] charFrequency = new int[distinctChars][2];
        distinctChars = 0;
        for (int i = 0; i < 26; ++i) {
            if (charCount[i] > 0) {
                charFrequency[distinctChars++] = new int[] {charCount[i], i};
            }
        }

        // Sort the character frequency matrix by frequency in descending order.
        Arrays.sort(charFrequency, (a, b) -> b[0] - a[0]);

        // StringBuilder to build the result.
        StringBuilder result = new StringBuilder();
        int idx = 0; // Index used for inserting characters in result.

        // Fill the characters in the result string.
        for (int[] entry : charFrequency) {
            int freq = entry[0], charIndex = entry[1];
            while (freq-- > 0) {
                result.setCharAt(idx, (char) ('a' + charIndex));
                idx += 2;
                // Wrap around if index goes beyond string length.
                if (idx >= length) {
                    idx = 1;
                }
            }
        }

        return result.toString();
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to reorganize the string such that no two adjacent characters are the same
    string reorganizeString(string s) {
        vector<int> counts(26, 0); // Counts for each character in the alphabet
        // Calculate the counts for each character in the string
        for (char c : s) {
            ++counts[c - 'a'];
        }

        // Find the maximum occurrence of a character
        int maxCount = *max_element(counts.begin(), counts.end());
        int n = s.size();
        // If the maximum count is more than half the length of the string, reorganization is not possible
        if (maxCount > (n + 1) / 2) return "";

        // Pairing count of characters with their corresponding alphabet index
        vector<pair<int, int>> charCounts;
        for (int i = 0; i < 26; ++i) {
            if (counts[i]) {
                charCounts.push_back({counts[i], i});
            }
        }

        // Sort the character counts in ascending order
        sort(charCounts.begin(), charCounts.end());
        // Prepare the result string with the same length as the input
        reverse(charCounts.begin(), charCounts.end());

        // Prepare the result string with the same length as the input
        string result = s;
        int idx = 0; // Index to keep track of placement in result string

        // Loop through sorted character counts and distribute characters across the result string
        for (auto& entry : charCounts) {
            int count = entry.first, alphabetIndex = entry.second;
            while (count--) {
                // Place the character at the index, then skip one place for the next character
                result[idx] = 'a' + alphabetIndex;
                idx += 2; // Move to the next position skipping one

                // If we reach or pass the end of the string, start placing characters at the first odd index
                if (idx >= n) idx = 1;
            }
        }

        // Return the reorganized string
        return result;
    }
};
```

Typescript Solution

```typescript
// Function to reorganize the string so that no two adjacent characters are the same
function reorganizeString(s: string): string {
    const counts = new Array(26).fill(0); // Counts for each character in the alphabet
    // Calculate the counts for each character in the string
    for (let c of s) {
        counts[c.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Find the maximum occurrence of a character
    const maxCount = Math.max(...counts);
    const n = s.length;
    // If the maximum count is more than half the length of the string, reorganization is not possible
    if (maxCount > Math.floor((n + 1) / 2)) return "";

    // Pairing count of characters with their corresponding alphabet index
    const charCounts: [number, number][] = [];
    for (let i = 0; i < 26; ++i) {
        if (counts[i]) {
            charCounts.push([counts[i], i]);
        }
    }

    // Sort the character counts in descending order
    charCounts.sort((a, b) => b[0] - a[0]);

    // Prepare the result string with the same length as the input
    let result = s;
    let idx = 0; // Index to keep track of placement in the result string

    // Loop through sorted character counts and distribute characters across the result string
    for (let [count, alphabetIndex] of charCounts) {
        while (count--) {
            // Place the character at the index, then skip one place for the next character
            result = setCharAt(result, idx, String.fromCharCode('a'.charCodeAt(0) + alphabetIndex));
            idx += 2; // Move to the next position skipping one

            // If we reach or pass the end of the string, start placing characters at the first odd index
            if (idx >= n) idx = 1;
        }
    }

    // Return the reorganized string
    return result;
}

// Helper function to replace a character at a specific index in a string
function setCharAt(str: string, index: number, chr: string): string {
    if (index > str.length - 1) return str;
    return str.substring(0, index) + chr + str.substring(index + 1);
}
```

Time and Space Complexity

The time complexity of the code is `O(n + a log a)`. The `Counter(s)` initialization takes `O(n)` time to count frequencies of each character in the input string of length `n`. The `.most_common()` method sorts these counts, which takes `O(a log a)` time in the worst case when all characters are different. The while loop inside the for loop iterates over all `n` characters to construct the output string, resulting in `O(n)` time. Therefore, the most expensive operation is the sorting with `O(a log a)` time, and when added to the other `O(n)` time operations, the overall time complexity remains `O(n + a log a)`.

The space complexity of the code is `O(n)`. The space complexity comes from storing the count of each character using `Counter` which requires `O(a)` space in the worst case where all characters are unique. Additionally, the `ans` array is used to build the output string and has a length of `n`, contributing `O(n)` space. However, since these do not scale with `n` together (the `Counter` won't scale to `n` if `ans` is `n`, and vice versa), the total space complexity is still `O(n)`.