1647. Minimum Deletions to Make Character Frequencies Unique

Medium Greedy Hash Table String Sorting Leetcode Link

Problem Description

The task is to create a "good" string by removing characters from the given string `s`. A "good" string is defined as one in which no two distinct characters have the same number of occurrences (or frequency). The goal is to find and return the minimum number of character deletions required to achieve a "good" string.

Frequency is simply how many times a character appears in the string. As an example, in the string `aab`, the character `a` has a frequency of 2, while `b` has a frequency of 1. The varying frequencies of each character play a crucial role in determining what constitutes a "good" string.

Intuition

To solve this problem, we need to adjust the frequency of characters so that no two characters have the same count. To minimize the number of deletions, we should try to decrease the frequencies of the more common characters as opposed to the less common ones, since generally this will lead to fewer total deletions.

The solution approach involves these steps:

1. First, we count the frequency of each character in the string using a counter data structure.
2. Then, we sort the frequencies in descending order so we can address the highest frequencies first.
3. We initialize a variable `pre` to `inf` which represents the previously encountered frequency that has been ensured to be unique by performing the necessary deletions.
4. We iterate over each sorted frequency value:
   - If `pre` is `0`, indicating that we can't have any more characters without a frequency, we must add all of the current frequency to the deletion count since having a frequency of `0` for all subsequent characters is the only way to ensure uniqueness.
   - If the current frequency is greater than or equal to `pre`, we must delete enough of this character to make its frequency one less than `pre` and update `pre` to be this new value.
   - If the current frequency is less than `pre`, we simply update `pre` to this frequency as no deletions are needed, it's already unique.

Throughout this process, we keep track of the total number of deletions we had to perform. Our goal is to maintain the property of having unique frequencies as we consider each frequency from high to low. By considering frequencies in descending order, we ensure that we are minimizing the number of deletions needed by prioritizing making more frequent characters less frequent.

In the solution code, the `Counter` from the `collections` module is used to count the frequencies, `sorted()` gets the frequencies in descending order, and a for loop is used to apply the described logic, updating the `ans` variable to store the total number of deletions required. `inf` is used as a placeholder for comparison in the loop to handle the highest frequency case on the first iteration.

Solution Approach

The solution involves implementing a greedy algorithm which operates with the data structure of a counter to count letter frequencies and a sorted list to process those frequencies. The pattern used here is to always delete characters from the most frequent down to the least, ensuring no two characters have the same frequency. Here's how the implementation unfolds:

1. **Count Frequencies:** The `Counter` from Python's `collections` module is used to create a frequency map for each character in the string. The `Counter(s)` invocation creates a dictionary-like object where keys are the characters, and values are the count of those characters.

2. **Sort Frequencies:** These frequency values are then extracted and sorted in descending order: `sorted(cnt.values(), reverse=True)`. The sorting ensures that we process characters by starting from the highest frequency.

3. **Initialize Deletion Counter and Previous Frequency:** An integer `ans` is initialized to count the deletions needed and `pre` is set to `inf` to ensure that on the first iteration the condition `v >= pre` will be false.

4. **Process Frequencies:**

   - Iterate over the sorted frequency list.
   - If `pre` has been decremented to `0`, it means we can no longer have characters with non-zero frequency (as we cannot have frequencies less than 0). Thus, for the current frequency `v`, all characters must be deleted, hence `ans += v`.
   - If the current frequency `v` is greater than or equal to `pre`, we decrement `v` to one less than `pre` to maintain frequency uniqueness, which makes `pre` the new current frequency minus 1, and increment `ans` by the number of deletions made, `v - pre + 1`.
   - If `v` is less than `pre`, it's already unique, so update `pre` to `v` and continue to the next iteration.
5. **Return Deletions:** After processing all character frequencies, the sum of deletions stored in `ans` is returned, which is the minimum number of deletions required to make the string `s` "good".

In terms of complexity, the most time-consuming operation is sorting the frequencies, which takes O(n log n) time. Counting frequencies and the final iteration take linear time, O(n), making the overall time complexity O(n log n) due to the sort. The space complexity is O(n) for storing the character frequencies and the sorted list of frequencies.

By implementing this greedy approach, we ensure that the process is efficient and that the least number of deletions are performed to reach a "good" string.

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Suppose we have the string `s = "aabbccddd"`. We want to create a "good" string by removing characters so that no two characters have the same frequency. Let's apply the steps outlined in the solution approach:

1. **Count Frequencies:** First, we use a `Counter` to get the frequencies of each character in `s`. The counter reveals the frequencies as follows: `{'a': 2, 'b': 2, 'c': 2, 'd': 3}`.

2. **Sort Frequencies:** We sort these values in descending order, which gives us `[3, 2, 2, 2]`.

3. **Initialize Deletion Counter and Previous Frequency:** We initialize `ans = 0` for counting deletions and `pre = inf` as the previous frequency.

4. **Process Frequencies:** Now, we iterate over the sorted list and apply the logic:

   - For the first frequency `3`, since `pre` is `inf`, we don't need to delete anything. We update `pre` to `3`.
   - Next, we look at the frequency `2`. Since `pre` is `3`, we can keep it as is and update `pre` to `2`.
   - For the next frequency `2`, it's equal to `pre`, so we need to delete one character to make it `1` (one less than the current `pre`). We increment `ans` by `1` and update `pre` to `1`.
   - For the last frequency `2`, we again need to make it less than `pre`, so we delete two characters this time, making it `0`. We increment `ans` by `2` and since `pre` is already `1`, we note that we can't reduce it further and any additional characters would need to be fully deleted.
5. **Return Deletions:** We've finished processing and have made `3` deletions in total (`ans = 3`). The result is that the minimum number of deletions required to make `s` a "good" string is `3`.

After these steps, the initial string `aabbccddd` has been transformed into a "good" string `aabbcd` by deleting two 'd's and one 'c'. Each remaining character (`a`, `b`, `c`, `d`) has a unique frequency (2, 2, 1, 1 respectively).

And following the time complexity analysis, most of our time expense was in the sorting step, with the rest of the process performed in linear time relative to the length of the string.

Python Solution

```python
from collections import Counter
from math import inf

class Solution:
    def minDeletions(self, string: str) -> int:
        # Count the frequency of each character in the string
        frequency_counter = Counter(string)

        # Initialize the number of deletions to 0 and 'previous frequency' to infinity
        deletions = 0
        previous_frequency = inf

        # Iterate over the frequencies in descending order
        for frequency in sorted(frequency_counter.values(), reverse=True):
            # If the previous frequency is 0, we must delete all occurrences of this character
            if previous_frequency == 0:
                deletions += frequency
            # If frequency is not less than the previous frequency,
            # decrease it to the previous frequency minus one and update deletions
            elif frequency >= previous_frequency:
                deletions += frequency - (previous_frequency - 1)
                previous_frequency -= 1
            else:
                # If frequency is less than the previous frequency, update the previous frequency
                previous_frequency = frequency

        # Return the total number of deletions required
        return deletions

# Example usage:
# sol = Solution()
# result = sol.minDeletions("aab")
# print(result) # Expected output would be 0 since no deletions are required for unique character frequencies.

```

Java Solution

```java
class Solution {

    // Function to find the minimum number of character deletions required
    // to make each character frequency in the string unique
    public int minDeletions(String s) {
        // Array to store the frequency of each character in the string
        int[] characterFrequency = new int[26];

        // Fill the array with the frequency of each character
        for (int i = 0; i < s.length(); ++i) {
            characterFrequency[s.charAt(i) - 'a']++;
        }

        // Sort the frequencies in ascending order
        Arrays.sort(characterFrequency);

        // Variable to keep track of the total deletions required
        int totalDeletions = 0;
        // Variable to keep track of the previous frequency value
        // Initialized to a large value that will not be exceeded by any frequency
        int previousFrequency = Integer.MAX_VALUE;

        // Go through each frequency starting from the highest
        for (int i = 25; i >= 0; --i) {
            int currentFrequency = characterFrequency[i];

            // If the previous frequency is 0, then all frequencies of this character must be deleted
            if (previousFrequency == 0) {
                totalDeletions += currentFrequency;
            } else if (currentFrequency >= previousFrequency) {
                // If the current frequency is greater than or equal to the previous frequency,
                // We need to decrease it to one less than the previous frequency
                totalDeletions += currentFrequency - previousFrequency + 1;
                previousFrequency--;
            } else {
                // Update the previous frequency to be the current frequency for the next iteration
                previousFrequency = currentFrequency;
            }
        }

        // Return the total deletions required to make each character frequency unique
        return totalDeletions;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>

class Solution {
public:
    // This function computes the minimum number of deletions required to make
    // each character in the string appear a unique number of times
    int minDeletions(string s) {
        // Count the frequency of each character in the string
        vector<int> frequencyCount(26, 0);
        for (char& c : s) {
            ++frequencyCount[c - 'a'];
        }

        // Sort the frequencies in descending order
        sort(frequencyCount.rbegin(), frequencyCount.rend());

        int deletions = 0; // Holds the number of deletions made

        // Loop through the frequency count starting from the second most frequent character
        for (int i = 1; i < 26; ++i) {
            // If the current frequency is not less than the previous (to ensure uniqueness)
            // and is also greater than 0, we decrement the current frequency to
            // make it unique and count the deletion made
            while (frequencyCount[i] >= frequencyCount[i - 1] && frequencyCount[i] > 0) {
                --frequencyCount[i]; // Decrement the frequency to make it unique
                ++deletions;         // Increment the number of deletions
            }
        }

        // Return the total number of deletions made to achieve unique character frequencies
        return deletions;
    }
};

```

Typescript Solution

```typescript
function minDeletions(s: string): number {
    // Create a frequency map for the characters in the string
    const frequencyMap: { [key: string]: number } = {};
    for (const char of s) {
        frequencyMap[char] = (frequencyMap[char] || 0) + 1;
    }

    // Initialize the variable for counting the number of deletions
    let deletionsCount = 0;

    // Extract the array of all frequencies
    const frequencies: number[] = Object.values(frequencyMap);

    // Sort the frequencies array in ascending order
    frequencies.sort((a, b) => a - b);

    // Iterate over the sorted frequencies
    for (let i = 1; i < frequencies.length; ++i) {
        // Continue reducing the frequency of the current element until
        // it becomes unique or reaches zero
        while (frequencies[i] > 0 && frequencies.indexOf(frequencies[i]) !== i) {
            // Decrement the frequency of the current character
            --frequencies[i];

            // Increment the deletions count
            ++deletionsCount;
        }
    }

    // Return the total number of deletions made to make all character
    // frequencies unique
    return deletionsCount;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code mainly consists of three parts:

1. Counting the frequency of each character in the string `s` which takes `O(n)` time, where `n` is the length of the string `s`.

2. Sorting the counts which take `O(k log k)` time, where `k` is the number of unique characters in the string `s`. In the worst case, `k` can be up to `n` if all characters are unique.

3. Iterating over the sorted counts to determine the minimum number of deletions which takes `O(k)` time.

Thus, the overall time complexity is `O(n + k log k + k)`, which simplifies to `O(n + k log k)` because `n` is at least as large as `k`.

Space Complexity

The space complexity of the code mainly comes from two parts:

1. Storing the character counts which require `O(k)` space, where `k` is the number of unique characters in `s`.

2. The sorted list of counts which also requires `O(k)` space.

Thus, the space complexity is `O(k)`.