2423. Remove Letter To Equalize Frequency

Counting

String

Problem Description

Hash Table

Easy

removing one letter, each of the remaining letters should appear the same number of times in the adjusted string. For example, consider the string "aabbcc". If we remove one 'c', we will have "aabb" left, where 'a' and 'b' each appear twice, meeting the condition for equal frequency.

In this problem, we are dealing with a string word that is made up of lowercase English letters. The objective is to find if it's

possible to remove one letter from this string such that the remaining letters all have the same frequency. In other words, after

Important points to note from the problem: The given string uses a 0-based index, meaning the first character has an index of 0.

- We need to check the frequencies of the letters present after the removal and are only concerned with those letters that would still exist in the
- string.

If no removal results in equal frequencies, return False.

essentially a specialized dictionary used to count hashable objects.

■ Construct a set comprehension set(v for v in cnt.values() if v) which:

We are required to remove exactly one letter, and doing nothing is not an option.

- Intuition To solve this problem, the intuition is to iterate through each letter in the word, simulate the removal of that letter, and check the

frequencies of the remaining letters to see if they all match. The solution is implemented as follows: Use a counter to track the frequency of each letter in the original word.

Iterate through each letter, decrement its count in the counter (simulating its removal), and check if all the other counts are the same. If after removing a letter, all other letters have the same count, it means it is possible to have equal frequency by removing

- one letter, so return True. If the condition is not met, restore the count of the removed letter back before moving onto the next letter.
- **Solution Approach** The implementation of the solution is straightforward, leveraging Python's Counter from the collections module, which is

Here's a step-by-step walk-through of the algorithm:

each iteration:

word).

its removal.

Example Walkthrough

We initialize a Counter object with the word as an argument to count the frequency of each letter present in the word. The function then enters a loop, iterating through the keys in the counter (which represent unique letters in the word), and for

The frequency of the current letter is decreased by one, simulating its removal. This is done using cnt[c] -= 1. We then check if the frequencies match for all the remaining letters. To do this concisely, we: 0

efficiently check for equal frequencies after each removal.

be {1, 2}, so the condition is not met and we restore 'b'.

value, implying all remaining letters have the same frequency.

if the frequencies of the remaining letters match.

def equalFrequency(self, word: str) -> bool:

char_count = Counter(word)

for char in char_count.keys():

char_count[char] -= 1

char_count[char] += 1

Create a counter for all characters in the word

Iterate through each character in the counter

Decrement the character's count by 1

Generate a set of all non-zero counts in the counter

Restore the original count for the character

// Count the frequency of each character in the word

// Iterate through each character in the alphabet

// If the current character is present in the word

// Decrease the frequency of the character by 1

if (targetFreq > 0 && v != targetFreq) {

int targetFreq = 0; // The target frequency all characters should have

continue; // Skip if the character is not in the word

// Check if after removing one character, the rest have the same frequency

isValid = false; // Frequencies differ, set flag to false

targetFreq = v; // Set the current frequency as the target for others to match

boolean isValid = true; // Flag to check if the current modification leads to equal frequencies

for (int i = 0; i < word.length(); ++i) {</pre>

freq[word.charAt(i) - 'a']++;

for (int v : freq) {

 $if (v == 0) {$

break;

target_frequency = count;

// Restore the count after checking

// assuming 'a' maps to index 0 and 'z' maps to index 25.

charFrequency[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;

const charFrequency: number[] = new Array(26).fill(0);

// Iterate through each character's frequency.

// If no equal frequency is possible by the removal of one character

// Initialize a count array of length 26 to store the frequency of each letter,

// Populate the charFrequency array with the count of each character in the word.

// Decrement the frequency of the current character to check if

// we can achieve equal frequency by removing one char occurrence.

// Iterate through the frequencies to check if they are all the same.

// Update commonFrequency with the current non-zero frequency.

if (commonFrequency && frequency !== commonFrequency) {

// Since we modified the original frequency, restore it back.

// If we did not find any instance where all non-zero frequencies

Generate a set of all non-zero counts in the counter

Check if all remaining character counts are the same

Restore the original count for the character

// were equal after removing one char occurrence, return false.

// Initialize a variable to store the frequency of the first non-zero character we encounter.

continue; // Skip if frequency is 0 as we are looking for non-zero frequencies.

// If commonFrequency is set and current frequency is different, set the equal flag to false.

if (can_equalize) {

++counts[i];

function equalFrequency(word: string): boolean {

return false;

for (const char of word) {

for (let i = 0; i < 26; ++i) {

if (charFrequency[i]) {

charFrequency[i]--;

let commonFrequency = 0;

break;

return true;

charFrequency[i]++;

for char in char_count.keys():

char_count[char] -= 1

return True

char_count[char] += 1

Time and Space Complexity

return False

Time Complexity

Space Complexity

if len(unique_counts) == 1:

let allFrequenciesEqual = true;

if (frequency === 0) {

for (const frequency of charFrequency) {

allFrequenciesEqual = false;

commonFrequency = frequency;

Iterate through each character in the counter

Decrement the character's count by 1

};

TypeScript

return true;

for (int i = 0; i < 26; ++i) {

if (freq[i] > 0) {

freq[i]--;

unique_counts = set(count for count in char_count.values() if count)

Return False if no condition satisfies the equal frequency requirement

Solution Implementation

from collections import Counter

class Solution:

 Loops through all frequency values in the counter. Includes a value in the set if it is non-zero (since a frequency dropping to zero implies the letter is effectively removed from the

Creates a set so that any duplicates are removed and only unique frequency values remain.

■ If the resulting set has only one element, it means all remaining letters have the same frequency.

- If this condition is met, we return True immediately, indicating success. If the condition is not met, we restore the frequency of the current letter before moving on to the next one with cnt[c] += 0
- equal frequency of letters. In this case, the function returns False.

This solution approach effectively employs the counter to test each possible single-letter removal and leverages Python's set to

If the loop completes and no return statement has been executed, this implies no single removal could achieve the desired

1. This step is crucial to ensure that the next iteration starts with the original frequencies, minus the next letter to simulate

Let's consider the string word = "aabbccd". We want to determine if it's possible to remove one letter from this string so that the remaining letters all have the same frequency.

First, we initialize a Counter object with the string. For word = "aabbccd", the counter would look like this: Counter({'a': 2,

We construct a set from the values of the counter, excluding any zeros: {1, 2}. Since the set has more than one unique value,

Next, we simulate removing one 'b', we get Counter({'a': 2, 'b': 1, 'c': 2, 'd': 1}). The set of frequencies would again

We then enter a loop to iterate through the keys in the counter. For the first iteration, we begin with the letter 'a', decreasing its count by one, resulting in Counter({'a': 1, 'b': 2, 'c': 2,

the condition for all letters to have the same frequency isn't met with the removal of 'a'.

'b': 2, 'c': 2, 'd': 1}). This means 'a' appears twice, 'b' appears twice, 'c' appears twice, and 'd' appears once.

We restore the count of 'a' back to its original value and move on to the next letter: Counter({'a': 2, 'b': 2, 'c': 2, 'd':

1}).

'd': 1}).

We proceed through the letters. When we decrement 'c', we get Counter({'a': 2, 'b': 2, 'c': 1, 'd': 1}) and a set of frequencies {1, 2}. We restore 'c' and continue.

Finally, we simulate removing 'd'. This results in Counter({'a': 2, 'b': 2, 'c': 2, 'd': 0}). The frequency set would be

{2}, since we exclude the zero frequency of 'd' (it is as if 'd' has been removed from the word). This set contains one unique

Since we've found a case where removing one letter results in equal frequencies for the remaining letters, we return True. It is indeed possible to remove one letter from "aabbccd" to make all remaining letters have the same frequency.

This walkthrough illustrates how the solution approach tests different scenarios by removing each letter one by one and checking

- **Python**
- # Check if all remaining character counts are the same if len(unique_counts) == 1: return True

class Solution { public boolean equalFrequency(String word) { // Frequency array to hold the count of each character in the word int[] freq = new int[26];

return False

Java

```
// If removing one occurrence of this character results in all other characters having the same frequency
               if (isValid) {
                    return true;
               // Undo the frequency change as we move on to the next character
                freq[i]++;
       // If no single removal leads to equal frequencies, return false
       return false;
class Solution {
public:
   bool equalFrequency(string word) {
        int counts[26] = {0}; // Initialize an array to store the frequency of each letter
       // Populate the frequency array with counts of each character in the word
       for (char& c : word) {
           ++counts[c - 'a'];
       // Iterate through the alphabet
        for (int i = 0; i < 26; ++i) {
            if (counts[i]) { // Check if the current letter has a non-zero frequency
               --counts[i]; // Decrementing the count to check if we can equalize frequency by removing one
               // Initialize a variable to store the frequency to compare against
               int target_frequency = 0;
                bool can_equalize = true; // Flag to check if equal frequency can be achieved
               // Iterate through the counts array to check the frequencies
                for (int count : counts) {
                    if (count == 0) {
                        continue; // Skip if the count is zero
                   // If we have already set the frequency to compare and current one doesn't match
                    if (target_frequency && count != target_frequency) {
                        can_equalize = false; // We cannot equalize the frequencies
                        break;
```

// If this is the first non-zero frequency we see, we set it as the target frequency

// Checking if we can equalize the frequency by the removal of one character

// If all non-zero frequencies are equal, return true. if (allFrequenciesEqual) {

```
return false;
from collections import Counter
class Solution:
   def equalFrequency(self, word: str) -> bool:
       # Create a counter for all characters in the word
        char_count = Counter(word)
```

unique_counts = set(count for count in char_count.values() if count)

Return False if no condition satisfies the equal frequency requirement

characters, this results in up to 0(k) iterations. Inside the loop, updating a count in the dictionary is an O(1) operation. 3.

O(n) operation where n is the length of the word.

The time complexity of the provided code is determined by several factors:

- set. The condition len(set(v for v in cnt.values() if v)) == 1 is essentially checking if all non-zero values in the cnt
- dictionary are equal, which takes up to 0(k) time to compute since it involves iteration over all values to form a set and then checking its length.
- Combining these factors, the overall time complexity is $0(n + k^2)$. In the worst case, if all characters are unique, k is equal to n, this simplifies to 0(n^2).
- For space complexity: The cnt dictionary stores counts of unique characters, so it takes up O(k) space where k is the number of unique characters.

The set and list comprehensions create temporary sets and lists that can contain up to k elements. However, these are

Building the cnt dictionary based on the Counter class, which requires iterating over every character in the word, results in a

The for loop in the code iterates over each character in the unique set of characters of the word. If k is the number of unique

The set and list comprehension iterates over the values of cnt, which is also 0(k), as it is done for each character in unique

temporary and not stored, so they don't add to the space complexity asymptotically. Summarizing, the overall space complexity is O(k). In the worst case scenario, k = n, which yields O(n) for the space complexity.