1897. Redistribute Characters to Make All Strings Equal

String

Counting

Problem Description

Hash Table

Easy

The LeetCode problem presents us with a scenario where we're provided an array of strings and we're asked to determine if it's possible to make all strings within the array equal by performing a series of operations. Here, the operation consists of moving any character from one string to any position in another string, keeping in mind that the strings are non-empty and the indices chosen for each operation should be distinct. The goal is to see if we can use these operations to transform the array so all strings in the array are identical.

For example, if the input array is ["abc", "aabc", "bc"], we can perform operations to move characters around until all strings become "abc." Thus, in this case, the answer would be true.

If the operation can be successfully performed to make every string equal using any number of operations, we return true. If not,

we return false.

characters freely between strings, what really matters is whether the total count of each distinct character in the input array is divisible by the number of strings. If that's true for every character, then we can distribute the characters evenly among all

all strings.

ntuition

strings, making them all equal.

To arrive at the solution approach, consider the following points: We can only move characters between strings, so the total number of each character must remain the same. To make all strings equal, each string must have an identical character count for every individual character.

The intuition behind the solution is based on the frequency of each character across all strings. Since we are allowed to move

 The length of all strings will be equal if we can make them identical, and this length must be a multiple of the number of strings. • If the total count of any character isn't a multiple of the number of strings in the array, it's impossible to distribute that character evenly across

for counting hashable objects. Here's how the algorithm flows:

- The provided Python solution implements this logic by:
- 1. Creating a counter to tally the frequency of each character across all strings in the words.
- 2. Iterating over each string in the words array, and then over each character in these strings, updating the counter for each character. 3. Checking if each character's count is divisible by the number of strings n. This is achieved with the all function, which iteratively applies the modulo operation to each count value and returns True if all results are zero; otherwise False.

We initialize the Counter object with no elements.

counter = Counter()

for word in words:

array.

for c in word:

counter[c] += 1

element in the counter's values.

of the allowed operations.

Example Walkthrough

Solution Approach

We then iterate over the list of words, and for each word in words, we iterate over each character to update our counter.

Here, counter[c] += 1 is incrementing the count for the character c each time it is encountered. This step essentially builds a

frequency map where the key is the character and the value is the total number of times it appears across all strings in the

The last step is to determine if it is possible to make all strings equal by checking that each character count is divisible by n.

each character is a multiple of the number of strings. The all function then checks whether this condition is True for every

If every character can be evenly distributed among the strings, all will return True, meaning that we can make all strings equal by

The implementation of the solution utilizes a Counter from the Python collections module, which is a specialized dictionary used

After populating the Counter, we obtain the total number of strings n in the original words array. n = len(words)

```
return all(count % n == 0 for count in counter.values())
  This line uses a generator expression inside the all function. The expression count % n == 0 checks whether the count of
```

make all strings equal using any number of the specified operations. This approach works effectively for this problem because it abstracts away all specifics regarding actual character positions and movements, focusing only on the overall character counts, which is the core aspect of the problem given the unrestricted nature

the defined operations. If even one character cannot be evenly distributed, all will return False, which means it's impossible to

Let's walk through a small example to illustrate the solution approach described above. Consider the input array ["axx", "xay", "yaa"]. The goal is to determine if it's possible to make all these strings equal by moving characters between strings.

Following the suggested steps: We first initialize an empty Counter object that will count the frequency of each character across all the strings. counter = Counter()

We then iterate over each word in our array (["axx","xay","yaa"]) and, for each word, we iterate over each character to

counter = {'a': 1, 'x': 2} # After processing "xay" counter += {'x': 1, 'a': 1, 'y': 1}

Final counter

update our counter. Thus we get:

counter = {'a': 4, 'x': 3, 'y': 2}

After processing "axx"

After processing "yaa"

counter += {'y': 1, 'a': 2}

see if there is any remainder.

The all function will check:

the input array ["axx", "xay", "yaa"].

from collections import Counter

char_counter = Counter()

for char in word:

for word in words:

def makeEqual(self, words: List[str]) -> bool:

char_counter[char] += 1

Initialize a Counter to keep track of the frequency of each character.

Iterate over each word in the list and count the characters.

return True # All characters are evenly distributed across words.

rules.

Python

Java

class Solution {

/**

class Solution:

We then determine the total number of words in the array, which is 3 in our case. n = len(words) # n = 3

The last step is to check if every character count is divisible by n, the number of strings. We apply the modulus operation to

```
Given that not every character can be evenly distributed across the three strings (since 4 % 3 and 2 % 3 do not yield a zero
remainder), the all function will return False. This means it's impossible to make all strings equal using the allowed operations for
```

4 % 3 == 0 # False for character 'a'

2 % 3 == 0 # False for character 'y'

3 % 3 == 0 # True for character 'x'

return all(count % n == 0 for count in counter.values())

```
Solution Implementation
```

by the number of strings, allows us to efficiently determine whether all strings can be made equal according to the problem's

In conclusion, the solution applies a frequency count logic which, when coupled with the modulo operation to check for divisibility

Calculate the number of words to check if characters can be evenly distributed. num_words = len(words) # If each character's count is divisible by the number of words, # then it is possible to rearrange characters to make all words equal. for count in char_counter.values(): if count % num_words != 0:

return False # If not divisible by num_words, can't make all words equal.

```
* Checks if characters from 'words' can be redistributed equally.
* @param words Array of strings to be evaluated.
* @return boolean True if characters can be redistributed equally, False otherwise.
*/
```

public boolean makeEqual(String[] words) {

// Loop over each word in the array.

charCounts[c - 'a']++;

for (char c : word.toCharArray()) {

int[] charCounts = new int[26];

// Number of words in the array.

int numWords = words.length;

for (String word : words) {

// Array to count the occurrences of each character.

// Increment the count for each character in the word.

```
// Check that each character's count is divisible by the number of words.
        for (int count : charCounts) {
            if (count % numWords != 0) {
                // If a character's count isn't divisible by numWords,
                // equal redistribution isn't possible.
                return false;
       // If all characters could be redistributed equally, return true.
       return true;
C++
class Solution {
public:
    // Method to check if the characters of the given words can be rearranged
   // to make all the words equal
    bool makeEqual(vector<string>& words) {
       // Initialize a counter vector to count the occurrences of each letter
       vector<int> letterCounter(26, 0);
       // Loop over each word in the vector
        for (const string& word : words) {
            // Count the occurrences of each character in the word
            for (char c : word) {
                ++letterCounter[c - 'a']; // Increment the count for the character
        int numWords = words.size(); // Store the total number of words
       // Loop over the counter and check if each character's total count
       // is divisible by the number of words, otherwise return false
        for (int count : letterCounter) {
            if (count % numWords != 0) {
                return false; // If not divisible, we can't make all words equal
```

```
function makeEqual(words: string[]): boolean {
   // Get the number of words to determine if letters can be distributed equally
    let wordCount = words.length;
   // Initialize an array for the 26 letters of the English alphabet, all starting at 0
    let letterCounts = new Array(26).fill(0);
   // Iterate through each word in the array
```

// Increment the count of this letter in the `letterCounts` array

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

// If all counts are divisible, return true

// Iterate through each letter in the word

return true;

for (let word of words) {

for (let char of word) {

if count % num_words != 0:

Time and Space Complexity

};

TypeScript

```
// Check if each letter's count can be evenly divided by the number of words
      for (let count of letterCounts) {
          if (count % wordCount !== 0) {
              // If a letter cannot be evenly divided, return false
              return false;
      // If all letters can be evenly divided, return true
      return true;
from collections import Counter
class Solution:
   def makeEqual(self, words: List[str]) -> bool:
       # Initialize a Counter to keep track of the frequency of each character.
        char counter = Counter()
       # Iterate over each word in the list and count the characters.
        for word in words:
            for char in word:
                char counter[char] += 1
       # Calculate the number of words to check if characters can be evenly distributed.
       num_words = len(words)
       # If each character's count is divisible by the number of words,
       # then it is possible to rearrange characters to make all words equal.
        for count in char_counter.values():
```

return False # If not divisible by num_words, can't make all words equal.

return True # All characters are evenly distributed across words.

the words. This is because the code consists of a nested loop where the outer loop iterates over each word in words, and the inner loop iterates over each character in the word. Each character is then added to the counter, which is an operation that takes

Time Complexity

0(1) time on average. Because every character in every word is processed once, the total time taken is proportional to the total number of characters in all words combined. **Space Complexity** The space complexity of the code is O(U), where U is the number of unique characters across all words. This is due to the use of

The time complexity of the code is O(N * M), where N is the number of words in the input list words and M is the average length of

the counter which stores a count for each distinct character. Since the number of unique characters is limited by the character set being used (in this case, usually the lowercase English letters, which would be at most 26), the space used by the counter could be considered fixed for practical purposes. However, in the most general case where any characters could appear, the space complexity is bounded by the number of unique characters U.