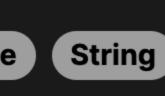
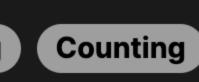
2068. Check Whether Two Strings are Almost Equivalent







Problem Description

equivalent if for each letter in the alphabet, the number of times it appears in one word is within three of the number of times it appears in the other word. We are asked to write a function that takes two strings of equal length and returns true if they are almost equivalent, and false if they are not. This problem is essentially about comparing the frequency of each character from 'a' to 'z' in the two strings and checking if that frequency difference is at most 3.

The problem presents us with the concept of two strings, word1 and word2, being almost equivalent. Two strings are almost

Intuition

strings. If the difference in the counts of any letter is more than 3, the strings are not almost equivalent.

The intuition behind the solution is to use a Counter data structure to store the frequencies of each letter from word1 and then

To solve this problem, the initial step is to understand that we need to count and compare the frequency of each letter in both

decrement those frequencies based on the letters from word2. This way, we directly get the difference in frequencies between word1 and word2. By using the built-in Counter from Python's collections module, we can efficiently count the letter frequencies and easily compute the differences.

After processing both strings, we then check if all frequency differences are within the acceptable range (at most 3). The all()

function is suitable for this check as it ensures that every letter satisfies the almost equivalent condition. If any letter's absolute frequency difference is greater than 3, all() immediately returns false, indicating that the words are not almost equivalent. Only if all frequency differences are within the allowed limit will all() return true, confirming that the strings are indeed almost equivalent.

Solution Approach

counting hashable objects (in this case, the characters of the strings). It internally maintains a dictionary where keys are the

elements being counted and the values are their respective counts.

Here's a breakdown of the steps in the algorithm:

Ne initialize the Counter with word1. This gives us a dictionary where each key is a unique character from word1, and its

In this solution, we're using the Counter data structure from Python's collections module, which is particularly useful for

2 We iterate through each character c in word2:

- 2. We iterate through each character c in word2 and decrement the count of c in our Counter by 1. If c wasn't in word1, its count in the Counter will be -1, accurately reflecting the difference in frequency between word1 and word2.
- After processing both strings, our Counter now represents the net frequency differences of each character.
 The final step is to examine if *all* values (net differences) in the Counter are less than or equal to 3 in absolute value. This is
- achieved by using the all() function combined with a generator expression. The expression $abs(x) \ll 3$ is evaluated for

corresponding value is the number of times the character appears in word1.

every value x in the Counter. If they all satisfy this condition, it means that the frequency of no character differs by more than 3 between the strings, hence word1 and word2 are almost equivalent, and the function returns true. If even one such condition fails, the function returns false as the strings are not almost equivalent.

In terms of complexity:

The time complexity of the solution is O(N), where N is the length of the input strings. This is because we iterate over each

• The space complexity is O(1) despite using the Counter, as the size of the Counter is bounded by the number of unique

string once.

- characters in word1, which is at most 26 in the English alphabet. Hence, it is a constant space overhead.

 This approach is efficient and exploits the constant upper bound on the number of unique characters. It elegantly simplifies the
- Example Walkthrough

Let's consider word1 = "abc" and word2 = "bcd". We shall walk through the solution approach using these two words.

Step 1: Initialize the Counter with word1. This gives us a Counter object where each key is a character from word1, and the

Counter for word1: {'a': 1, 'b': 1, 'c': 1}

Step 2: Iterate through word2, decrementing the counts in the Counter by 1 for each character in word2.

• After seeing 'b' in word2, decrement 'b' count by 1.

```
    After seeing 'c' in word2, decrement 'c' count by 1.
```

Counter: {'a': 1, 'b': 0, 'c': 0, 'd': -1}

For character 'a': abs(1) <= 3 is True.

For character 'd': abs(-1) <= 3 is True.

Counter: {'a': 1, 'b': 0, 'c': 1}

problem down to a single Counter check at the end.

corresponding value is the frequency of that character in word1.

• After seeing 'd' in word2, since 'd' is not in the Counter, its count becomes -1.

def checkAlmostEquivalent(self, word1: str, word2: str) -> bool:

char_counter[character] -= 1

result = sol.checkAlmostEquivalent("abc", "pqr")

Create a counter object to count the frequency of each character in word1

Check if the absolute difference of each character's frequency is at most 3

print(result) # Output: True or False based on the words being almost equivalent

// If all characters' frequency differences are 3 or less, return true

// Two words are almost equivalent if the absolute difference in the frequencies

++letterCounts[c - 'a']; // Increment the count for this letter.

--letterCounts[c - 'a']; // Decrement the count for this letter.

// This function checks if two words are almost equivalent.

bool checkAlmostEquivalent(string word1, string word2) {

// Count the frequency of each letter in word1.

// Decrement the frequency count for each letter in word2.

// of each letter in the words is at most 3.

for (char& c : word1) {

for (char& c : word2) {

If it is within [-3, 3] inclusive for every character, the words are almost equivalent

Counter: {'a': 1, 'b': 0, 'c': 0}

```
Step 4: Use the all() function with a generator expression to verify that all frequency differences are within 3. We check if abs(x) \ll 3 for every value x in the Counter.
```

For character 'b': abs(0) <= 3 is True.
For character 'c': abs(0) <= 3 is True.

Step 3: The Counter object now accurately reflects the difference in frequency for each character between word1 and word2.

This example clearly illustrates the solution approach – character counts are compared directly, and the process is both simple and efficient.

Since all conditions are True, all() returns True, hence word1 and word2 are almost equivalent and the function will return True.

char_counter = Counter(word1) # Decrement the count in the counter for each character in word2 for character in word2:

return true;

Solution Implementation

from collections import Counter

```
return all(abs(frequency) <= 3 for frequency in char_counter.values())
# Example usage:
# sol = Solution()</pre>
```

Python

class Solution:

```
Java
class Solution {
   // Function to check if two strings are almost equivalent
    public boolean checkAlmostEquivalent(String word1, String word2) {
       // Array to keep count of the frequency difference of each character
        int[] charFrequencyDifference = new int[26];
       // Increment the count for each character in word1
        for (int i = 0; i < word1.length(); ++i) {</pre>
            charFrequencyDifference[word1.charAt(i) - 'a']++;
       // Decrement the count for each character in word2
        for (int i = 0; i < word2.length(); ++i) {</pre>
            charFrequencyDifference[word2.charAt(i) - 'a']--;
       // Check if any character's frequency difference is greater than 3
        for (int frequencyDifference : charFrequencyDifference) {
            // If the absolute frequency difference is greater than 3, return false
            if (Math.abs(frequencyDifference) > 3) {
                return false;
```

int letterCounts[26] = {0}; // Initialize an array to count letters, one entry for each letter of the alphabet.

C++

public:

class Solution {

```
// Check all the counters to ensure none of the frequencies' absolute differences are greater than 3.
        for (int i = 0; i < 26; ++i) {
            if (abs(letterCounts[i]) > 3) {
                // If the absolute difference in frequencies of any letter is more than 3, words are not almost equivalent.
                return false;
       // If the function did not return false in the loop, the words are almost equivalent.
       return true;
};
TypeScript
function checkAlmostEquivalent(word1: string, word2: string): boolean {
    // Create an array of size 26 to represent each letter of the alphabet and initialize with 0
    const letterCounts: number[] = new Array(26).fill(0);
    // Iterate over each character in wordl and increment the count of the corresponding letter
    for (const char of word1) {
        letterCounts[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    // Iterate over each character in word2 and decrement the count of the corresponding letter
   for (const char of word2) {
        letterCounts[char.charCodeAt(0) - 'a'.charCodeAt(0)]--;
    // Check that the absolute difference in counts of each letter does not exceed 3
    return letterCounts.every(count => Math.abs(count) <= 3);</pre>
```

```
# Example usage:
# sol = Solution()
# result = sol.checkAlmostEquivalent("abc", "pqr")
# print(result) # Output: True or False based on the words being almost equivalent
```

Time and Space Complexity

char_counter = Counter(word1)

char counter[character] -= 1

for character in word2:

def checkAlmostEquivalent(self, word1: str, word2: str) -> bool:

Decrement the count in the counter for each character in word2

Create a counter object to count the frequency of each character in word1

Check if the absolute difference of each character's frequency is at most 3

return all(abs(frequency) <= 3 for frequency in char counter.values())</pre>

If it is within [-3, 3] inclusive for every character, the words are almost equivalent

from collections import Counter

class Solution:

The time complexity of the given code is O(n) where n is the combined length of word1 and word2. This is because the algorithm goes through each character of word1 to build the counter and then iterates over all characters of word2 to adjust the counts. Each operation of increment or decrement in the counter takes constant time, hence the overall complexity remains linear with respect to the input size.

The space complexity is O(C) where C is the size of the character set used in the counter. Since the problem deals with lowercase English letters, C is at most 26. This is the space required to store the count of each character. Since C is a constant number, we can also consider the space complexity to be O(1), as it does not grow with the input size but is bounded by a fixed limit.