1160. Find Words That Can Be Formed by Characters

```
Problem Description
```

Hash Table

String

<u>Array</u>

Easy

Each character in chars may only be used once when forming a word. After identifying all the "good" strings, we must calculate the sum of their lengths and return that sum as the result. For example, if words = ["cat", "bt", "hat", "tree"] and chars = "atach", only "cat" and "hat" are "good" because they can be formed using the characters in chars without reusing a single character. The lengths of "cat" and "hat" are 3 and 3,

In this problem, we are given two inputs: an array of strings called words and a single string called chars. Our task is to determine

which strings in the words array are "good". A "good" string is one that can be completely formed using the characters in chars.

respectively, so the sum is 6. The goal of the problem is to implement a function that can do this calculation for any given words and chars. Intuition

Count Characters in chars: We first count the frequency of each character in the string chars. This helps us know how many

(w).

Iterate Over words: Next, we loop through each string in words and count the frequency of each character in the current string

The solution approach can be divided into the following steps:

times we can use a particular character while forming words.

- Check if a Word Can be Formed: To determine if a word is "good", we compare character frequencies of the current word with those in chars. If for each character in the word, the count in chars is greater than or equal to the count in the word, the
- word is "good". Calculate and Sum Lengths: For each "good" string found, we add its length to a running total, ans.
- Return the Result: Once all words have been checked, return the accumulated sum of lengths.

The implementation of the solution uses a Counter from Python's collections module, which is a specialized dictionary designed

Create a Counter for chars: First, a Counter is created for the string chars. This Counter object, named cnt, will provide a

for counting hashable objects. The Counter data structure is ideal for tracking the frequency of elements in an iterable.

dictionary-like structure where each key is a character from chars and its corresponding value is the number of occurrences of that character.

ans = 0

for w in words:

wc = Counter(w)

cnt = Counter(chars)

Solution Approach

Initialize an Answer Variable: An integer ans is initialized to zero, which will hold the sum of lengths of all "good" strings in

Loop Through Each Word in words: We iterate over each word win the words array. For each word, a new Counter is created to

Check if the Word is "Good": Using the all function, we check if every character c in the current word has a frequency count

the array words.

count the occurrences of characters in that word.

Here is a step-by-step breakdown of how the solution is implemented:

- v that is less than or equal to its count in cnt. This ensures that each character required to form the word w is available in the quantity needed in chars. if all(cnt[c] >= v for c, v in wc.items()):
- o If the condition is true for all characters, it means the word can be formed from the characters in chars, hence it is a "good" word. We add the length of the word to ans.

just use its values to check the wc counts.

h:1, l:2, o:2, e:1, d:1, r:1, w:1

ans += len(w)

Return the Total Length: After iterating through all the words, the total length of all "good" words is stored in ans, which we return. return ans

This algorithm has a time complexity that is dependent on the total number of characters in all words (0(N) where N is the total

complexity is O(U) where U is the total number of unique characters in chars and all words since Counter objects need to keep

number of characters) since we are counting characters for each word and iterating over each character count. The space

• We do not modify the original cnt Counter because we do not want to affect the counts for the subsequent iteration of words. Instead, we

- track of each unique character and its count. **Example Walkthrough**
- We have enough characters to potentially make the words "hello", "world", and "loop".

Consider a small example where we have words = ["hello", "world", "loop"] and chars = "hloeldorw".

Step 1: Count Characters in chars: We count the frequency of each character:

■ We check each character against our chars count and see that we can form "hello" with chars. Since "hello" is a "good" word, we add its length (5) to ans: ans += 5 # ans = 5

Count characters: w:1, o:1, r:1, l:1, d:1

ans += 5 # ans = 10

■ Count characters: l:1, o:2, p:1

■ Count characters: h:1, e:1, l:2, o:1

Step 3: Iterate Over words: We iterate over each word:

2. Step 2: Initialize an Answer Variable: We initialize ans to zero:

"good" words. We return this as the final answer:

Python

ans = 0

For "hello":

For "world":

For "loop":

return ans # ans = 10

Solution Implementation

char_count = Counter(chars)

word_count = Counter(word)

int[] charFrequency = new int[26];

Iterate through each word in the list of words

total_length = 0

for word in words:

return total_length

int totalLength = 0;

for (String word : words) {

■ We have only 2 'o's and 1 'l' in chars, not enough to form the word "loop", so we do not add anything to ans for this word. Step 4: Return the Total Length: After processing all the words, the value of ans is 10 (5+5). This is the sum of lengths of all

The sum of lengths of all "good" words that can be formed by the given chars is 10 in this example.

Initialize answer to hold the total length of all words that can be formed

if all(char_count[char] >= count for char, count in word_count.items()):

total_length += len(word) # If it can be formed, add the word's length to the total

Count the frequency of each character in the current word

Check if the word can be formed by the chars in 'chars'

Return the total length of all words that can be formed

// Array to store the frequency of each character in 'chars'

// Flag to check if the current word can be formed

// Variable to hold the total length of all words that can be formed

// Count the frequency of each character in the current word

// Array to store the frequency of each character in the current word

// Determines the sum of lengths of all words that can be formed by characters in 'chars'

int totalLength = 0; // This will hold the sum of lengths of words that can be formed

// Check if each character in the word can be formed by the characters in 'chars'

// If the current character exceeds the frequency in 'chars', the word can't be formed

bool canFormWord = true; // Flag to check if the word can be formed

// If the word can be formed, add its length to the totalLength

public int countCharacters(String[] words, String chars) {

// Count the frequency of each character in 'chars'

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

charFrequency[chars.charAt(i) - 'a']++;

// Iterate over each word in the array 'words'

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

int index = word.charAt(i) - 'a';

int countCharacters(vector<string>& words, string chars) {

// Frequency array for characters in the current word

if (++wordCount[index] > charCount[index]) {

// Frequency array for characters in 'chars'

int charCount[26] = {0};

for (char& ch : chars) {

for (auto& word : words) {

// Fill the frequency array

++charCount[ch - 'a'];

// Iterate over each word in the list

int wordCount[26] = {0};

for (char& ch : word) {

break;

int index = ch - 'a';

canFormWord = false;

int[] wordFrequency = new int[26];

boolean canBeFormed = true;

■ All characters are present in our chars count. "world" is also a "good" word, so we add its length (5) to ans:

- from collections import Counter # Import the Counter class from the collections module class Solution: def countCharacters(self, words: List[str], chars: str) -> int: # Count the frequency of each character in the given string 'chars'
- Java

class Solution {

public:

```
// If the character frequency exceeds that in 'chars', the word can't be formed
                if (++wordFrequency[index] > charFrequency[index]) {
                    canBeFormed = false;
                   break; // Break out of the loop as the current word can't be formed
            // If the word can be formed, add its length to the totalLength
            if (canBeFormed) {
                totalLength += word.length();
       // Return the total length of all words that can be formed
       return totalLength;
C++
#include <vector>
#include <string>
class Solution {
```

if (canFormWord) { totalLength += word.size();

```
return totalLength; // Return the total sum of lengths of all words that can be formed
  };
  TypeScript
  function countCharacters(words: string[], chars: string): number {
      // Function to get index of character in the alphabet array (0-25)
      const getIndex = (char: string): number => char.charCodeAt(0) - 'a'.charCodeAt(0);
      // Initialize an array to store the frequency of each character in 'chars'
      const charCount = new Array(26).fill(0);
      // Fill the charCount array with the frequency of each character in 'chars'
      for (const char of chars) {
          charCount[getIndex(char)]++;
      // Initialize a variable to keep track of the total length of all valid words
      let totalLength = 0;
      // Iterate over each word in the 'words' array to check if it can be formed
      for (const word of words) {
          // Initialize an array to store the frequency of each character in the current word
          const wordCount = new Array(26).fill(0);
          // Flag to check if the current word can be formed
          let canBeFormed = true;
          // Iterate over each character in the word to update wordCount and check if it can be formed
          for (const char of word) {
              wordCount[getIndex(char)]++;
              // If the character's frequency in word exceeds that in 'chars', set the flag to false
              if (wordCount[getIndex(char)] > charCount[getIndex(char)]) {
                  canBeFormed = false;
                  break;
          // If the word can be formed, add its length to the totalLength
          if (canBeFormed) {
              totalLength += word.length;
      // Return the total length of all words that can be formed using 'chars'
      return totalLength;
from collections import Counter # Import the Counter class from the collections module
class Solution:
```

The time complexity of the code can be analyzed as follows: The creation of the cnt counter from the chars string: this operation takes 0(m), where m is the length of the chars string since we must count the frequency of each character in chars.

def countCharacters(self, words: List[str], chars: str) -> int:

Iterate through each word in the list of words

char_count = Counter(chars)

word count = Counter(word)

total length = 0

for word in words:

return total_length

Time Complexity

Time and Space Complexity

Count the frequency of each character in the given string 'chars'

Count the frequency of each character in the current word

Check if the word can be formed by the chars in 'chars'

Return the total length of all words that can be formed

Initialize answer to hold the total length of all words that can be formed

if all(char_count[char] >= count for char, count in word_count.items()):

total_length += len(word) # If it can be formed, add the word's length to the total

- The all function checks if all characters in w have a count less or equal to their count in cnt. This operation is O(k) as it checks each character's count (up to the total character count of a word).
- Since steps 3 and 4 are within the loop iteration, they will run n times, which makes that part of the algorithm 0(n*k). Summing these up, the total time complexity is 0(m + n*k).

Inside the loop, a new counter wc is created for each word: this operation also has a complexity of O(k).

- **Space Complexity**
- The space complexity can be analyzed as follows: The cnt counter for chars utilizes O(u) space, where u is the unique number of characters in chars.
- The wc counter for each word similarly utilizes O(v) space in the worst case, where v is the unique number of characters in that word.

The for-loop iterates over each word in the words list. Let the length of the list be n and the average length of the words be k.

However, since we is constructed for one word at a time, O(v) space will be reused for each word, and v is bounded by the fixed size of the alphabet (u), so we can consider O(u) space for the counters. Given that space used by variables like ans and temporary variables in the iteration is negligible relative to the size of the input,

the overall space complexity is dominated by the space required for the counters, which is O(u) where u is the number of unique

letters in chars and is bounded by the size of the character set used (e.g., the English alphabet, which has a fixed size of 26).

Therefore, the space complexity is O(u).