1160. Find Words That Can Be Formed by Characters

Easy Array Hash Table String

Leetcode Link

Problem Description

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`. 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, 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

The solution approach can be divided into the following steps:

1. **Count Characters in `chars`:** We first count the frequency of each character in the string `chars`. This helps us know how many times we can use a particular character while forming words.

2. **Iterate Over `words`:** Next, we loop through each string in `words` and count the frequency of each character in the current string (`w`).

3. **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".

4. **Calculate and Sum Lengths:** For each "good" string found, we add its length to a running total, `ans`.

5. **Return the Result:** Once all words have been checked, return the accumulated sum of lengths.

The beauty of this approach lies in the efficient use of character counting, which allows us to verify if a word can be formed without having to repeatedly search for characters in `chars`. By using a Python Counter object, which is essentially a specialized dictionary for counting hashable objects, we perform the needed comparisons succinctly and efficiently. The `all` function in Python is then used to ensure that all character counts in the word meet the availability requirement in `chars`, which is a very intuitive way to check the condition for every character simultaneously.

Solution Approach

The implementation of the solution uses a `Counter` from Python's `collections` module, which is a specialized dictionary designed for counting hashable objects. The `Counter` data structure is ideal for tracking the frequency of elements in an iterable.

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

1. **Create a Counter for `chars`:** First, a `Counter` is created for the string `chars`. This `Counter` object, named `cnt`, will provide a dictionary-like structure where each key is a character from `chars` and its corresponding value is the number of occurrences of that character.

   ```
   cnt = Counter(chars)
   ```

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

   ```
   ans = 0
   ```

3. **Loop Through Each Word in `words`:** We iterate over each word `w` in the `words` array. For each word, a new `Counter` is created to count the occurrences of characters in that word.

   ```
   for w in words:
       wc = Counter(w)
   ```

4. **Check if the Word is "Good":** Using the `all` function, we check if every character `c` in the current word has a frequency count `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()):
       ans += len(w)
   ```

   - 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`.
   - 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 just use its values to check the `wc` counts.

5. **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 (`O(N)` where `N` is the total number of characters) since we are counting characters for each word and iterating over each character count. The space complexity is `O(U)` where `U` is the total number of unique characters in `chars` and all words since `Counter` objects need to keep track of each unique character and its count.

Example Walkthrough

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

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

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

   We have enough characters to potentially make the words "hello", "world", and "loop".

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

   ```
   ans = 0
   ```

3. **Step 3: Iterate Over `words`:** We iterate over each word:

   - For "hello":

     - Count characters: `h:1, e:1, l:2, o:1`
     - 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
       ```

   - For "world":

     - Count characters: `w:1, o:1, r:1, l:1, d:1`
     - All characters are present in our `chars` count. "world" is also a "good" word, so we add its length (5) to `ans`:
       ```
       ans += 5   # ans = 10
       ```

   - For "loop":

     - Count characters: `l:1, o:2, p:1`
     - 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.

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

   ```
   return ans   # ans = 10
   ```

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

Python Solution

```python
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'
        char_count = Counter(chars)
        # Initialize answer to hold the total length of all words that can be formed
        total_length = 0

        # Iterate through each word in the list of words
        for word in words:
            # Count the frequency of each character in the current word
            word_count = Counter(word)
            # Check if the word can be formed by the chars in 'chars'
            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

        # Return the total length of all words that can be formed
        return total_length

```

Java Solution

```java
class Solution {
    public int countCharacters(String[] words, String chars) {
        // Array to store the frequency of each character in 'chars'
        int[] charFrequency = new int[26];

        // Count the frequency of each character in 'chars'
        for (int i = 0; i < chars.length(); ++i) {
            charFrequency[chars.charAt(i) - 'a']++;
        }

        // Variable to hold the total length of all words that can be formed
        int totalLength = 0;

        // Iterate over each word in the array 'words'
        for (String word : words) {
            // Array to store the frequency of each character in the current word
            int[] wordFrequency = new int[26];
            // Flag to check if the current word can be formed
            boolean canBeFormed = true;

            // Count the frequency of each character in the current word
            for (int i = 0; i < word.length(); ++i) {
                int index = word.charAt(i) - 'a';

                // 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++ Solution

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

class Solution {
public:
    // Determines the sum of lengths of all words that can be formed by characters in 'chars'
    int countCharacters(vector<string>& words, string chars) {
        // Frequency array for characters in 'chars'
        int charCount[26] = {0};
        // Fill the frequency array
        for (char& ch : chars) {
            ++charCount[ch - 'a'];
        }

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

        // Iterate over each word in the list
        for (auto& word : words) {
            // Frequency array for characters in the current word
            int wordCount[26] = {0};
            bool canFormWord = true; // Flag to check if the word can be formed

            // Check if each character in the word can be formed by the characters in 'chars'
            for (char& ch : word) {
                int index = ch - 'a';
                // If the current character exceeds the frequency in 'chars', the word can't be formed
                if (++wordCount[index] > charCount[index]) {
                    canFormWord = false;
                    break;
                }
            }

            // If the word can be formed, add its length to the totalLength
            if (canFormWord) {
                totalLength += word.size();
            }
        }

        return totalLength; // Return the total sum of lengths of all words that can be formed
    }
};

```

Typescript Solution

```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;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed as follows:

1. The creation of the `cnt` counter from the `chars` string: this operation takes `O(m)`, where `m` is the length of the `chars` string since we must count the frequency of each character in `chars`.

2. 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`.

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

4. 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 `O(n*k)`.

Summing these up, the total time complexity is `O(m + n*k)`.

Space Complexity

The space complexity can be analyzed as follows:

1. The `cnt` counter for `chars` utilizes `O(u)` space, where `u` is the unique number of characters in `chars`.

2. 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.

3. However, since `wc` 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)`.