1684. Count the Number of Consistent Strings

Easy Bit Manipulation Array Hash Table String

Leetcode Link

Problem Description

The problem provides us with a string `allowed` which is made up of distinct characters, meaning no character is repeated in the string `allowed`. Additionally, we are given an array of strings called `words`. Our task is to determine how many strings in the array `words` are "consistent."

A string is defined as consistent if every character in the string appears in the `allowed` string. In other words, if there's even one character in a word that is not present in `allowed`, then that word is not considered consistent.

The goal is to return the number of consistent strings in the array `words`.

Intuition

To solve this problem, the intuitive approach is to check each word in `words` and ensure all of its characters are contained within the `allowed` string. To optimize this process, we convert the `allowed` string to a set. Sets in Python are collections that provide O(1) time complexity for checking if an element is present, which is faster than if `allowed` were a list or a string, where the time complexity would be O(n).

Once we have the set of allowed characters, we iterate over each word in `words`. For each word, we check whether every character is in our set of allowed characters. We use the `all()` function which returns True if all elements of the iterable (the characters in the word) are True (or if the iterable is empty).

The expression `(c in s for c in w)` is a generator that yields True or False for each character `c` in the word `w` depending on whether `c` is in the set `s` or not. The `all()` function takes this generator as input and evaluates to True only if every character in the word is in the set of allowed characters.

Finally, we use the `sum()` function to count how many words are consistent by adding up 1 for every True result from the `all()` check.

The result is the number of consistent strings in the `words` array, which is what we return.

Solution Approach

The solution provided uses Python's set and comprehension features to perform the task efficiently.

1. The first step is to convert the `allowed` string into a set:

```
s = set(allowed)
```

   Converting to a set is significant because set operations in Python are usually faster than list or string operations. Specifically, checking for membership using the `in` operator is very fast for sets, taking O(1) time on average.

2. Next, we use a list comprehension to iterate over each word in `words`:

```
sum(all(c in s for c in w) for w in words)
```

   This comprehensive line does several things:

   - `for w in words`: We start by going over each word in the `words` list.
   - `all(c in s for c in w)`: For each word, we create a generator expression that yields True for each character `c` that is in the set `s`. The `all()` function checks if all values provided by this generator are True, meaning every character in the word is in the `allowed` set.
   - The entire `all()` expression will be True if the word is consistent and False otherwise.
   - `sum(...)`: We are then using `sum()` to add up all the True values. Since True is equivalent to 1 in Python, `sum()` effectively counts the number of consistent strings.

This algorithm is very concise and takes advantage of Python's high-level operations to solve the problem with both simplicity and efficiency. The use of `all()` combined with a generator expression and `sum()` allows us to perform the entire operation in a single, readable line of code.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose the string `allowed` contains the characters `"abc"` and we have an array `words` containing the words `["ab", "ac", "de", "abc", "xy"]`.

Now, let's apply the solution step by step:

1. We convert the `allowed` string into a set to expedite membership checks:

```
s = set('abc')  # This creates the set {'a', 'b', 'c'}
```

2. Next, we iterate over each word in `words` and use the `all()` function to check if every character of a word is in the set `s`:

```
result = sum(all(c in s for c in w) for w in words)
```

Let's break this down for each word:

- For the word `"ab"`: the characters are `'a'` and `'b'`, both of which are in the set `s`. So `all(c in s for c in 'ab')` will yield True for both characters, and `all()` will return True.
- For the word `"ac"`: the characters are `'a'` and `'c'`, both of which are in the set `s`. The `all()` function will also return True.
- For the word `"de"`: the character `'d'` is not in the set `s`, so `all(c in s for c in 'de')` will yield False when checking `'d'`, and `all()` will return False.
- For the word `"abc"`: all the characters `'a'`, `'b'`, and `'c'` are in the set `s`, and `all()` will return True.
- For the word `"xy"`: neither `'x'` nor `'y'` is in the set `s`, so `all(c in s for c in 'xy')` will yield False immediately when checking `'x'`, and `all()` will return False.

So, we get the following results for our `words` array:

- `"ab"`: Consistent
- `"ac"`: Consistent
- `"de"`: Not consistent
- `"abc"`: Consistent
- `"xy"`: Not consistent

Finally, the `sum()` function would add up all the True values (represented as 1 in Python):

```
result = sum([True, True, False, True, False])  # This equals 3
```

Therefore, the output would be `3`, as there are three consistent strings in the array `words`.

Python Solution

```
class Solution:
    def countConsistentStrings(self, allowed: str, words: list[str]) -> int:
        # Convert the allowed string into a set for O(1) lookup times
        allowed_chars = set(allowed)

        # Count the words in which all the characters are in the allowed set
        # Summing up the boolean values where True is counted as 1, False as 0
        consistent_words_count = sum(all(char in allowed_chars for char in word) for word in words)

        return consistent_words_count  # Return the total count of consistent words

```

Java Solution

```
class Solution {
    // Method to count number of consistent strings
    public int countConsistentStrings(String allowed, String[] words) {
        // Array to keep track of allowed characters
        boolean[] isAllowed = new boolean[26];

        // Populate the isAllowed array with characters from the 'allowed' string
        for (char c : allowed.toCharArray()) {
            isAllowed[c - 'a'] = true;
        }

        // Initialize count for consistent strings
        int count = 0;

        // Iterate through each word in the array 'words'
        for (String word : words) {
            // If the word is consistent, increment the count
            if (isConsistent(word, isAllowed)) {
                count++;
            }
        }

        // Return the total count of consistent strings
        return count;
    }

    // Helper method to check if a word is consistent
    private boolean isConsistent(String word, boolean[] isAllowed) {
        // Iterate through each character in the word
        for (int i = 0; i < word.length(); i++) {
            // If the character is not in the allowed list, return false
            if (!isAllowed[word.charAt(i) - 'a']) {
                return false;
            }
        }

        // Return true if all characters of the word are allowed
        return true;
    }
}

```

C++ Solution

```
#include <vector>
#include <string>
#include <bitset>

class Solution {
public:
    int countConsistentStrings(std::string allowed, std::vector<std::string>& words) {
        // Initialize a bitset to store whether each letter in the alphabet is allowed
        std::bitset<26> allowedLetters;
        for (char ch : allowed) {
            allowedLetters.set(ch - 'a');
        }

        // Counter for the number of consistent strings
        int consistentCount = 0;

        // Lambda function to check if all characters in a word are allowed
        auto isConsistent = [&](std::string& word) {
            for (char ch : word) {
                // If any character is not allowed, return false immediately
                if (!allowedLetters.test(ch - 'a')) return false;
            }
            // All characters in the word are allowed
            return true;
        };

        // Iterate through each word and increment the consistent count if the word is consistent
        for (std::string& word : words) {
            if (isConsistent(word)) {
                ++consistentCount;
            }
        }

        // Return the final count of consistent strings
        return consistentCount;
    }
};

```

Typescript Solution

```
// Counts the number of words from the 'words' array that contain only characters from the 'allowed' string.
// @param allowed - A string consisting of distinct lowercase English letters, representing the allowed characters.
// @param words - An array of strings, where each string consists only of lowercase English letters.
// @return The count of words from the 'words' array that are "consistent" - contain only characters from 'allowed'.
function countConsistentStrings(allowed: string, words: string[]): number {
    // Converts a string to a bitmask where each bit set represents the presence of a character in the string.
    // @param str - A string to convert to a bitmask.
    // @return A bitmask representing the characters of the input string.
    const convertToBitmask = (str: string): number => {
        let bitmask = 0;
        for (const char of str) {
            bitmask |= 1 << (char.charCodeAt(0) - 'a'.charCodeAt(0));
        }
        return bitmask;
    };

    // Create a bitmask for the allowed characters.
    const allowedMask = convertToBitmask(allowed);
    let consistentCount = 0; // Initialize the count of consistent strings.

    // Loop through each word in the array.
    for (const word of words) {
        // If the bitmask of the word OR'd with the allowedMask equals the allowedMask,
        // it means the word contains only characters from 'allowed'.
        if ((allowedMask | convertToBitmask(word)) === allowedMask) {
            consistentCount++; // Increment count as the word is consistent.
        }
    }

    // Return the total count of consistent strings.
    return consistentCount;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function depends on two factors: the length of the `words` list meaning the number of words it contains (let's denote this number as `n`) and the average length of the words (let's denote it as `k`). The function iterates over each word and then over each character in the word to check if it is in the set `s`. Checking membership in a set is an `O(1)` operation on average. Therefore, the time complexity for checking one word is `O(k)`, and for all the words it's `O(n * k)`.

Space Complexity

The space complexity is `O(s)` where `s` is the number of unique characters in the `allowed` string because these are stored in a set. The other variable that could contribute to space complexity is `w` in the generator expression, but since the space required for `w` is reused as the iteration proceeds and since the strings in words are input to the function and not duplicated by it, this does not add to the space complexity of the solution itself. The space complexity for the output of the sum function is `O(1)` since it's accumulating the result into a single integer. Therefore, the overall space complexity is `O(s) + O(1)` which simplifies to `O(s)`.