1935. Maximum Number of Words You Can Type

Easy Hash Table String

Problem Description

In the given problem, you are presented with a scenario where you have a keyboard with some malfunctioning letter keys. These keys are broken and do not work at all, while the rest of the keyboard is functioning properly. You're provided with a string called `text` that contains words separated by single spaces. There's no extra spacing at the beginning or end of this string. You're also given another string named `brokenLetters`, which contains unique letters that represent the broken keys on the keyboard.

Your task is to determine how many of the words in `text` you could type on this keyboard. In other words, a word can be typed if it doesn't contain any of the letters from `brokenLetters`. The goal is to return the count of such fully typable words from the original string `text`.

Intuition

To find a solution to the problem, the straightforward approach is to use a set to track all the broken keys. A set is chosen because it allows O(1) complexity for checking the existence of a character, which helps in efficiently determining whether a character in a word is broken.

The next step is to iterate over each word in the provided text. As we check each character in the word, we use the set to verify if that character corresponds to a broken key. If we don't find any broken letters in a word, it indicates that the word can be fully typed using the keyboard and therefore, we count that word as typable. Conversely, if even one character in the word matches a broken letter, the word can't be typed at all.

The solution code applies this logic to count the number of words that can be typed. It uses list comprehension paired with the `all()` function to efficiently test each word. Here's a breakdown of what the solution does:

- It first converts `brokenLetters` into a set for fast lookup.
- Then, it splits `text` into individual words.
- For each word, it checks whether all the characters are not in the set of broken letters using `all(c not in s for c in w)`.
- The `sum()` function accumulates the total count of words that satisfy the condition.

Solution Approach

The solution uses a simple but effective approach utilizing Python's built-in data structures and functions.

Step 1: Create a Set of Broken Letters

Firstly, the code creates a set from `brokenLetters`. A set is an appropriate data structure because it allows for constant time complexity, $O(1)$, for lookups.

```
s = set(brokenLetters)
```

Step 2: Split the Text into Words

The `text` string is then split into individual words using the `split()` method, which by default, separates words based on spaces.

```
text.split()
```

Step 3: Check Each Word

For each word in the text, we check if all characters `c` are not in the set `s`. This is done using the `all()` function alongside a generator expression. The `all()` function returns `True` if all elements of the iterable (in this case, the generator expression) are true. If any character `c` is found in the set `s` (broken letters), `all()` would return `False` for that word.

```
all(c not in s for c in w) for w in text.split()
```

Step 4: Count the Typable Words

We then use the `sum()` function to count the number of words for which the `all()` function returned `True`. This results in the total number of words from the `text` that can be typed using the keyboard despite the broken keys.

```
return sum(all(c not in s for c in w) for w in text.split())
```

Algorithm

The algorithm essentially iterates through each word and checks each character only once. For a text of n words and assuming the longest word has m characters, the algorithm would have a time complexity of $O(m * n)$, where $m * n$ represents the total number of characters we need to check. The space complexity of this algorithm is $O(k)$, where k is the number of broken letters because we are creating a set to hold these letters. However, since the number of letters in English is constant (26 letters), and `brokenLetters` cannot be longer than that, the set is of a constant size, and the space complexity can also be considered $O(1)$.

The solution is clean and efficient; it uses basic data structures in Python and leverages their properties, such as the set for fast lookups and the `all()` function to check the validity of each word.

Example Walkthrough

Let's take a simple example to walk through the solution approach. Suppose we have the text `text = "hello world program"` and let's say the `brokenLetters = "ad"`.

Step 1: Create a Set of Broken Letters

Create a set from `brokenLetters`. Our set `s` will look like this:

```
s = set('ad')
```

Now we have `s = {'a', 'd'}`.

Step 2: Split the Text into Words

Next, split `text` into individual words:

```
words = "hello world program".split()
```

After the split, `words` will be `['hello', 'world', 'program']`.

Step 3: Check Each Word

Now, check each word in `words` to see if any of the characters are in set `s`. For `'hello'`, applying `all(c not in s for c in 'hello')`:

- `'h'` is not in `s`, continue.
- `'e'` is not in `s`, continue.
- `'l'` is not in `s`, continue.
- `'l'` is not in `s`, continue.
- `'o'` is not in `s`, continue. Since all characters passed the check, `'hello'` can be typed.

For `'world'`, applying `all(c not in s for c in 'world')`:

- `'w'` is not in `s`, continue.
- `'o'` is not in `s`, continue.
- `'r'` is not in `s`, continue.
- `'l'` is not in `s`, continue.
- `'d'` is in `s`, stop. Since `'d'` is a broken letter, `'world'` cannot be typed.

For `'program'`, applying `all(c not in s for c in 'program')`:

- `'p'` is not in `s`, continue.
- `'r'` is not in `s`, continue.
- `'o'` is not in `s`, continue.
- `'g'` is not in `s`, continue.
- `'r'` is not in `s`, continue.
- `'a'` is in `s`, stop. Since `'a'` is a broken letter, `'program'` cannot be typed.

Step 4: Count the Typable Words

Finally, we count the typable words. Only the first word, `'hello'`, can be typed with the broken keys `'ad'`. So, using the `sum()` function:

```
return sum(all(c not in s for c in w) for w in words)
```

The code will evaluate to `1`, since only one word in the array can be typed.

Therefore, the final answer for the text `text = "hello world program"` with `brokenLetters = "ad"` is `1`, as only the word `"hello"` can be typed.

Solution Implementation

Python

```
class Solution:
    def canBeTypedWords(self, text: str, brokenLetters: str) -> int:
        # Convert the string of broken letters into a set for O(1) lookup times
        broken_letters_set = set(brokenLetters)

        # Split the input text by spaces to get individual words
        words = text.split()

        # Initialize a variable to count the number of words that can be typed
        count = 0

        # Iterate through each word in the list of words
        for word in words:
            # Check if all characters in the current word are not in the set of broken letters
            if all(char not in broken_letters_set for char in word):
                # If the word can be typed, increment the count
                count += 1

        # Return the final count of words that can be typed without using any broken letters
        return count
```

Java

```
class Solution {
    // Method to count how many words in a given text can be typed
    // using a keyboard with some broken letters.
    public int canBeTypedWords(String text, String brokenLetters) {
        // Array to keep track of which letters are broken.
        boolean[] isBroken = new boolean[26];

        // Populate the isBroken array; a 'true' value means the letter is broken.
        for (char letter : brokenLetters.toCharArray()) {
            isBroken[letter - 'a'] = true;
        }

        int count = 0; // This will store the number of words that can be typed.
        // Split the input text into words and iterate through them.
        for (String word : text.split(" ")) {
            boolean canTypeWord = true; // Flag to check if the current word can be typed.

            // Check each character in the word to see if it is broken.
            for (char letter : word.toCharArray()) {
                // If the letter is broken, set the flag to false and break out of the loop.
                if (isBroken[letter - 'a']) {
                    canTypeWord = false;
                    break;
                }
            }

            // If the word can be typed (none of its letters are broken), increase the count.
            if (canTypeWord) {
                count++;
            }
        }

        // Return the total count of words that can be typed.
        return count;
    }
}
```

C++

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

class Solution {
public:
    // Function to count the number of words that can be typed with broken letters
    int canBeTypedWords(std::string text, std::string brokenLetters) {
        // Initialize an array to mark the broken letters
        bool brokenStatus[26] = {false};
        // Mark the broken letters in the array
        for (char& letter : brokenLetters) {
            brokenStatus[letter - 'a'] = true;
        }

        // Variable to store the count of words that can be typed
        int count = 0;
        // Split the input text into words and process each word
        for (auto& word : split(text, ' ')) {
            bool canTypeWord = true;
            // Check each character in the word
            for (char& c : word) {
                // If the character is a broken letter, skip the word
                if (brokenStatus[c - 'a']) {
                    canTypeWord = false;
                    break;
                }
            }
            // If all characters are typable, increase the count
            if (canTypeWord) {
                count++;
            }
        }
        return count;
    }

    // Helper function to split a string into words based on a delimiter
    std::vector<std::string> split(const std::string& str, char delimiter) {
        std::vector<std::string> result;
        std::string currentWord;
        // Iterate over each character in the string
        for (char c : str) {
            // If the delimiter is encountered, add the current word to the result
            if (c == delimiter) {
                result.push_back(currentWord);
                currentWord.clear();
            } else {
                // Add the character to the current word
                currentWord.push_back(c);
            }
        }
        // Add the last word to the result
        result.push_back(currentWord);
        return result;
    }
};
```

TypeScript

```
function canBeTypedWords(text: string, brokenLetters: string): number {
    // Create an array to keep track of broken letters.
    const brokenStatus: boolean[] = new Array(26).fill(false);

    // Iterate through the list of broken letters and update their
    // status in the boolean array. 'true' means the letter is broken.
    for (const letter of brokenLetters) {
        const index = letter.charCodeAt(0) - 'a'.charCodeAt(0);
        brokenStatus[index] = true;
    }

    // Initialize a counter for the number of words that can be typed.
    let count = 0;

    // Split the text into words and iterate through each word.
    for (const word of text.split(' ')) {
        let canTypeWord = true; // Flag indicating if the word can be typed.

        // Check each character of the word.
        for (const char of word) {
            // If the character corresponds to a broken letter, mark the word as untypable.
            if (brokenStatus[char.charCodeAt(0) - 'a'.charCodeAt(0)]) {
                canTypeWord = false; // Cannot type this word, so set the flag to false.
                break; // No need to check other characters.
            }
        }

        // If the word can be typed (no broken letter found), increment the counter.
        if (canTypeWord) {
            count++;
        }
    }

    // Return the total number of words that can be typed.
    return count;
}
```

```
class Solution:
    def canBeTypedWords(self, text: str, brokenLetters: str) -> int:
        # Convert the string of broken letters into a set for O(1) lookup times
        broken_letters_set = set(brokenLetters)

        # Split the input text by spaces to get individual words
        words = text.split()

        # Initialize a variable to count the number of words that can be typed
        count = 0

        # Iterate through each word in the list of words
        for word in words:
            # Check if all characters in the current word are not in the set of broken letters
            if all(char not in broken_letters_set for char in word):
                # If the word can be typed, increment the count
                count += 1

        # Return the final count of words that can be typed without using any broken letters
        return count
```

Time and Space Complexity

The time complexity of the provided code is $O(n)$, where n is the length of the input string `text`. This is because the code iterates through each character of every word in `text` exactly once in the worst case, to check if any of the characters is in the set of broken letters.

The space complexity is $O(1)$ or $O(|E|)$ where $|E|$ represents the size of the set of unique letters, which in the context of the English alphabet is a constant 26. This space is used to store the set of broken letters. Since the size of this set is limited by the number of unique characters in the alphabet, and does not grow with the size of the input `text`, it is considered a constant space complexity.