387. First Unique Character in a String

Easy Queue Hash Table String Counting

Problem Description

The problem provides us with a string `s` and our task is to find the first character in this string that does not repeat anywhere else within that same string. In other words, we need to identify a character that appears only once in the string, and then return the position (index) of that character in the string. If our string doesn't have any non-repeating characters, we must return `-1`. An important detail is that we're interested in the *first* non-repeating character, so we must scan the string from the beginning, and as soon as we find a unique character, we know that is the answer.

Intuition

The intuition behind this solution starts with recognizing that in order to determine whether a character is repeated, we need to check the entire string. We can do this efficiently by counting how many times each character appears in the string. The `Counter` class from Python's `collections` module is a perfect fit for this, as it enables us to create a count of all characters with just a single pass through the string.

Once we have the counts of each character stored in the `cnt` dictionary, we make another pass through the string. This time, we examine each character and consult the count we've stored in `cnt` to see if it is `1` (which means the character is unique). The first time we find a character with a count of `1`, we know we've located the first non-repeating character, so we return its index.

If we get through the entire string without finding a non-repeating character, we return `-1`, which signals that there aren't any unique characters in the string.

Solution Approach

The provided solution utilizes two key components of the Python language: the `Counter` class from the `collections` module and the `enumerate` function.

Here's a breakdown of how the solution is implemented:

1. `cnt = Counter(s)`: The `Counter` class is used to create a dictionary-like object where each key is a character from the string `s`, and the corresponding value is the number of times that character appears in `s`. This is our first pass through the string and is done in `O(n)` time complexity, where `n` is the length of the string.

2. `for i, c in enumerate(s)`: We then use Python's built-in `enumerate` function to iterate over `s` again. This time, `enumerate` gives us both the index `i` and the character `c` for each position in the string. We do this to be able to return the index of the first unique character we encounter.

3. `if cnt[c] == 1`: Within the loop, we check the count of the current character `c` by accessing it in our counter `cnt`. If the count is `1`, it means `c` is a non-repeating character, and since we are scanning from the start of the string, it is the *first* non-repeating character.

4. `return i`: When we find a character that appears only once, we return the index `i`. This is our desired result according to the problem description.

5. `return -1`: This line is reached only if the loop completes without returning an index, indicating that there are no characters in the string that are non-repeating. Therefore, we return `-1` as specified.

The overall time complexity of the solution is `O(n)` as we traverse the string twice—once for counting the characters and once for checking their uniqueness. The space complexity is also `O(n)` because in the worst-case scenario, the `Counter` object may hold a distinct count for every character if they are all unique.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose the input string `s` is `"leetcode"`.

1. We use the `Counter` to count the occurrences of each character in `s`. The `Counter(s)` would look like this:

```
Counter({'l': 1, 'e': 3, 't': 1, 'c': 1, 'o': 1, 'd': 1})
```

2. We then iterate over the string `s` with the help of `enumerate` which gives us the index and the character at that index. Our first iteration would give us index `0` and character `'l'`.

3. We then check if the count of the current character in the `cnt` dictionary is `1`. For this example, `cnt['l']` would be `1`.

4. Since the count is `1`, we have found our first non-repeating character, which is `'l'`.

5. We return the index of `'l'`, which is `0`.

This works correctly for our example. If our string was `"lleetcode"`, where the first `'l'` is not unique, the steps would be as follows:

1. The `Counter` for this string would be:

```
Counter({'l': 2, 'e': 3, 't': 1, 'c': 1, 'o': 1, 'd': 1})
```

2. Iterating through the string, when we reach the character `'t'` at index `2`, we would find that `cnt['t']` is `1`, which means `'t'` is the first non-repeating character.

3. We would return `2`, which is the index of `'t'`.

Following these steps ensures we find the first unique character efficiently, or determine that there isn't one if we reach the end of the string without finding any character with a count of `1`.

Python Solution

```python
from collections import Counter

class Solution:
    def firstUniqChar(self, s: str) -> int:
        # Create a counter object to count the frequency of each character in the string
        char_count = Counter(s)

        # Iterate over the characters in the string with their indices
        for index, char in enumerate(s):
            # If the character's count is 1, it is unique
            if char_count[char] == 1:
                # Return the current index as the first unique character's position
                return index

        # If no unique character is found, return -1
        return -1

```

Java Solution

```java
class Solution {
    public int firstUniqChar(String s) {
        // Create an array to store the frequency of each character
        int[] charCounts = new int[26];

        // Get the length of the string for use in loops
        int stringLength = s.length();

        // Iterate over the string to count the frequency of each character
        for (int i = 0; i < stringLength; ++i) {
            charCounts[s.charAt(i) - 'a']++;
        }

        // Iterate over the string a second time to find the first unique character
        for (int i = 0; i < stringLength; ++i) {
            // If the count of the current character is 1, return its index
            if (charCounts[s.charAt(i) - 'a'] == 1) {
                return i;
            }
        }

        // If no unique character is found, return -1
        return -1;
    }
}
```

C++ Solution

```cpp
#include <string> // Include string library for string type

class Solution {
public:
    // Function to find the first non-repeating character in a string
    // and return its index. If it doesn't exist, return -1.
    int firstUniqChar(std::string s) {

        // Initialize a count array with 26 elements for each alphabet character
        int charCount[26] = {0};

        // Iterate over each character in the string and
        // increment the count in the corresponding array index
        for (char c : s) {
            ++charCount[c - 'a'];
        }

        // Get the size of the string
        int strLength = s.size();

        // Iterate over the string once again
        for (int i = 0; i < strLength; ++i) {
            // If the count at the position of the character is 1,
            // it means that it is a unique character, so return its index
            if (charCount[s[i] - 'a'] == 1) {
                return i;
            }
        }

        // If no unique character is found, return -1
        return -1;
    }
};

```

Typescript Solution

```typescript
function firstUniqChar(s: string): number {
    // Create an array to keep track of the frequency of each character.
    const charFrequency = new Array(26).fill(0);

    // Iterate through the string to fill the frequency array.
    for (const character of s) {
        // Increment the frequency of the character in the array.
        // 'a'.charCodeAt(0) gives us the char code for 'a', which we subtract
        // from the current char code to get the index (assuming lowercase ASCII).
        charFrequency[character.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Iterate through the string to find the first unique character.
    for (let index = 0; index < s.length; index++) {
        // Check if the character at 'index' has a frequency of 1,
        // which means it's unique.
        if (charFrequency[s.charCodeAt(index) - 'a'.charCodeAt(0)] === 1) {
            // If the character is unique, return its index.
            return index;
        }
    }

    // If there is no unique character in the string, return -1.
    return -1;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code has two main components: one for creating the counter dictionary, and the other for iterating through the string.

- `Counter(s)`: This function creates a hashmap (dictionary) of character frequencies in the string `s`. The time complexity of this operation is `O(n)` as it requires going through all the characters in the string once, where `n` is the length of the string.

- `for i, c in enumerate(s)`: This loop iterates over each character index pair in the string to check if the character count is `1`. The iteration itself is `O(n)`, since it goes over each character of the string once.

The combined time complexity is the sum of these two operations, which are both linear with respect to the length of the input string. Therefore, the overall time complexity is `O(n)`.

Space Complexity

The space complexity of the given code is primarily dependent on the size of the counter dictionary that gets created.

- `Counter(s)`: The space occupied by this dictionary is proportional to the number of distinct characters in the string `s`. In the worst-case scenario, where all characters in the string are unique, the space required will be `O(n)`. However, for a string with a fixed character set (like the ASCII characters), the space complexity would be `O(1)` with respect to the size of the character set since the counter dictionary cannot exceed the size of the character set irrespective of the length of the string.

Combining the above, the worst-case space complexity of the code is `O(n)` if the number of distinct characters is proportional to the length of the string. For strings in a fixed character set, the space complexity is `O(1)`.