1941. Check if All Characters Have Equal Number of Occurrences

Easy Hash Table String Counting

Problem Description

The problem requires us to determine if a given string `s` is a "good" string. A string is considered "good" if every character that appears in the string occurs with the same frequency. That means, each character must appear the same number of times as every other character in the string. For example, the string "aabb" is good because both 'a' and 'b' appear twice. On the other hand, the string "aabc" is not good because the frequency of 'a' is 2, the frequency of 'b' is 1, and the frequency of 'c' is 1. Our function must return `true` if the string is good, or `false` otherwise.

Intuition

To solve this problem, we need to count the occurrences of each character in the string. A Python dictionary can be used to map each character to the number of times it appears in the string. To facilitate this, we use the `Counter` class from Python's `collections` module, which automatically creates a dictionary where the keys are the characters from the string and the values are the counts for those characters.

Solution Approach

The solution utilizes a counter and a set to solve the problem efficiently. Here is a step-by-step explanation of the algorithm, utilizing the Python `collections.Counter` and Python `set`:

1. **Initialize Counter**: The `Counter` class from Python's `collections` module is used to create a counter object. When the `Counter` is instantiated with a string `s`, it creates a dictionary-like object. Each key in this object is a unique character from the string, and the corresponding value is the number of times that character appears.

   ```
   cnt = Counter(s)
   ```

2. **Create a Set of Occurrences**: We then use the `values()` method of the `Counter` object to get a list of all the frequencies of characters in the string. These values are then turned into a set to eliminate duplicate counts.

   ```
   set(cnt.values())
   ```

   If all characters occur with the same frequency, this set will only contain one element (that frequency).

3. **Compare Set Size**: To determine if the string is good, we check the size of the set. If its size is exactly one, this means that all characters in the string occurred an equal number of times.

   ```
   len(set(cnt.values())) == 1
   ```

   The expression above will be `True` for a good string and `False` for a string that is not good.

4. **Return Result**: The result of the comparison mentioned in step 3 is the output of the method `areOccurrencesEqual`. This method returns `True` if the string is good and `False` otherwise.

   ```
   return len(set(cnt.values())) == 1
   ```

The `Counter` is a hash table-based data structure, and it helps us count the frequency of each character in `O(n)` time complexity, with `n` being the length of the string. The set conversion and length checking are `O(k)` operations where `k` is the number of unique characters in the string, which is at most the length of the string and in practice often much less. Therefore, the overall time complexity of the algorithm is `O(n)` and is pretty efficient.

The solution elegantly uses the properties of the Python `Counter` and `set` to determine the "goodness" of the string in a concise and readable manner.

Example Walkthrough

Let's apply the solution approach to a small example to illustrate how it works. Suppose we have the string `s = "ccaaabbb"`. We want to determine if this string is "good" according to the problem description.

1. **Initialize Counter**: First, we create a counter from the string:

   ```
   from collections import Counter
   cnt = Counter("ccaaabbb")
   ```

   The `Counter` object `cnt` will look like this: `{'c': 3, 'a': 3, 'b': 3}`. Each key is a unique character, and each value is the number of times that character appears in the string.

2. **Create a Set of Occurrences**: We extract the frequencies of each character and create a set:

   ```
   frequencies = set(cnt.values())
   ```

   For our example, the set of frequencies will be `{3}` because each character ('c', 'a', and 'b') appears three times in the string.

3. **Compare Set Size**: Now we check if the size of this set is exactly 1:

   ```
   is_good_string = len(frequencies) == 1
   ```

   Since `len({3})` is indeed 1, `is_good_string` will be `True`.

4. **Return Result**: Lastly, we output the result. Since `is_good_string` is `True`, our method `areOccurrencesEqual` would return `True` for the string "ccaaabbb".

Following the above steps, we have determined that "ccaaabbb" is a "good" string according to the given definition because every character in the string has the same frequency of 3. Thus, when the `areOccurrencesEqual` function is applied to "ccaaabbb", it returns `True`.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def areOccurrencesEqual(self, s: str) -> bool:
        # Create a counter to store the frequency of each character in the string
        char_count = Counter(s)

        # Get a set of all the frequency values from the counter
        unique_frequencies = set(char_count.values())

        # Check if all characters have the same frequency, which means
        # there should only be one unique frequency in the set.
        # Return True if yes, otherwise return False.
        return len(unique_frequencies) == 1
```

Java

```java
class Solution {
    public boolean areOccurrencesEqual(String s) {
        // Create an array to hold the count of each letter in the string
        int[] letterCounts = new int[26];

        // Iterate over the characters in the string and increment the count
        // for each character in the letterCounts array
        for (int i = 0; i < s.length(); i++) {
            letterCounts[s.charAt(i) - 'a']++;
        }

        // Variable to keep track of the count of the first character
        int targetCount = 0;

        // Iterate over the counts of each letter
        for (int count : letterCounts) {
            // If the count is positive (i.e., the letter is present in the string)
            if (count > 0) {
                // If it's the first non-zero count we've seen, set it as the target count
                if (targetCount == 0) {
                    targetCount = count;
                } else if (targetCount != count) {
                    // If the current count doesn't match the target count,
                    // not all characters occur the same number of times
                    return false;
                }
            }
        }

        // If we've gotten through the entire letterCounts array and all counts are equal,
        // return true (all characters occur the same number of times)
        return true;
    }
}
```

C++

```cpp
#include <string>
using namespace std;

class Solution {
public:
    bool areOccurrencesEqual(string s) {
        // Initialize a count array for all 26 letters to 0
        int letterCount[26] = {0};

        // Count the occurrence of each letter in the string
        for (char& c : s) {
            ++letterCount[c - 'a']; // Increment the count for the appropriate letter
        }

        // The 'occurrenceValue' will hold the number of times a letter should occur
        int occurrenceValue = 0;
        // Loop through the count array to determine if all non-zero counts are equal
        for (int count : letterCount) {
            if (count) { // If the current letter has occurred at least once
                if (occurrenceValue == 0) { // If it's the first non-zero count encountered
                    occurrenceValue = count; // Set the 'occurrenceValue' to this count
                } else if (occurrenceValue != count) { // If the current count doesn't match the 'occurrenceValue'
                    return false; // Not all occurrences are equal, return false
                }
            }
        }

        return true; // All non-zero occurrences are equal, return true
    }
};
```

TypeScript

```typescript
function areOccurrencesEqual(inputString: string): boolean {
    // Initialize a count array of length 26 to store the occurrence of each alphabet letter,
    // corresponding to the lowercase English alphabet letters a-z.
    const charCounts: number[] = new Array(26).fill(0);

    // Iterate over each character in the input string.
    for (const char of inoutString) {
        // Increment the count for this character in the charCounts array.
        // The character code of the current character minus the character code of 'a'
        // gives the index in the array.
        charCounts[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Find the first non-zero count in the array to establish a reference count.
    const referenceCount = charCounts.find(count => count > 0);

    // Return true if every count in the array is either zero (unused character)
    // or equal to the referenceCount found above. This means all used characters occur
    // the same number of times in the input string.
    // If there's any count that is different from referenceCount (and not zero), return false.
    return charCounts.every(count => count === 0 || count === referenceCount);
}
```

```python
from collections import Counter

class Solution:
    def areOccurrencesEqual(self, s: str) -> bool:
        # Create a counter to store the frequency of each character in the string
        char_count = Counter(s)

        # Get a set of all the frequency values from the counter
        unique_frequencies = set(char_count.values())

        # Check if all characters have the same frequency, which means
        # there should only be one unique frequency in the set.
        # Return True if yes, otherwise return False.
        return len(unique_frequencies) == 1
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` represents the length of string `s`. This is because creating the counter object `cnt` requires iterating over all characters in the string once, which is a linear operation.

The space complexity is also `O(n)` for the counter object `cnt`, which stores a count for each unique character in the string. In the worst case, if all characters are unique, the space required to store the counts is proportional to the number of characters in the string.