2287. Rearrange Characters to Make Target String

Easy Hash Table String Counting

Leetcode Link

Problem Description

The task is to determine how many copies of the given `target` string can be formed using characters from the string `s`. We are allowed to rearrange the characters taken from `s` to match the order of characters in `target`. Both strings `s` and `target` start indexing from 0. In essence, remember that you can only form a complete `target` if all the characters in `target` can be matched by characters from `s`, considering also the number of occurrences.

For example, if `s` is "abcab" and `target` is "ab", you can form two copies of `target` because `s` contains two 'a's and two 'b's, which are enough to form "ab" twice.

Intuition

The solution relies on a simple insight: For each unique character in `target`, count how many times we can find it within `s`. This is essentially a problem of matching supply with demand. The "supply" here is the number of times a character appears in `s`, and the "demand" is the number of times it appears in `target`.

To solve the problem, we perform the following steps:

1. Count the occurrences of each character in `s`. This gives us the supply for each character available in `s`.
2. Count the occurrences of each character in `target`. This represents the demand for each character to formulate a `target` string.
3. For each character in `target`, calculate how many times we can provide for that demand using our supply. This is done by dividing the supply of a character in `s` by the demand of that character in `target`.
4. The minimum of these quotients (rounded down) is the maximum number of times we can form the `target` string. This is because we cannot form a complete `target` string if even one character is in short supply.

For the final answer, we can use Python's `min` function to find this minimum quotient for all characters. The `Counter` class from the `collections` module is ideal for counting character occurrences efficiently.

Here is the description of the solution in Python code:

```
class Solution:
    def rearrangeCharacters(self, s: str, target: str) -> int:
        cnt1 = Counter(s)      # Count characters in s
        cnt2 = Counter(target)  # Count characters in target
        # Calculate min number of times we can form target character-wise
        return min(cnt1[c] // v for c, v in cnt2.items())
```

Using `Counter(s)` and `Counter(target)`, we create two dictionaries to store the counts of each character. We then use a generator expression inside the `min` function to calculate the minimum number of complete copies of `target` that can be formed. This generator expression iterates over each character and its count in `target`, using `//` for integer division to find how many times each character in `target` can be supplied by `s`. The `min` function then determines the maximum copies of `target` by finding the smallest of these values.

Solution Approach

The solution implements a straightforward approach using two algorithms/data structures:

- **Counters (Hash Tables)**: The Python `Counter` class from the `collections` module serves as a hash table or a dictionary that maps each unique character to its count. This data structure is used twice: once for counting occurrences of characters in `s` and once for `target`. This allows us to have immediate access to the count of each character required and available.
- **Integer Division**: The Python `//` operator is used for integer division. It provides the quotient of the division without the remainder, which is crucial since we're interested in the number of complete copies of `target` that can be formed.

Here's a breakdown of the implementation steps:

1. `cnt1 = Counter(s)`: We count each character in `s`. `Counter` iterates over `s`, and for each character, it increments the character's count in the hash table. For example, if `s = "bbaac"`, then `cnt1` will be `{ 'b': 2, 'a': 2, 'c': 1 }`.

2. `cnt2 = Counter(target)`: Similarly, we count the characters in `target`. If `target = "abc"`, then `cnt2` will be `{ 'a': 1, 'b': 1, 'c': 1 }`.

3. The generator expression `(cnt1[c] // v for c, v in cnt2.items())` iterates over the items of `cnt2`, which are tuples of (character, count) from `target`. For each character `c` in `target`, it calculates how many times that character can be taken from `s`. This is the supply we have (from `cnt1[c]`) divided by the demand (from `v`).

4. `min(...)`: We wrap the generator expression with `min` to find the character that is the limiting factor in forming the `target`. This step ensures that we consider the "least abundant" character in terms of how many complete `target` strings we can formulate. The minimum value is the maximum number of `target` strings we can form.

Here's the code section again for reference:

```
class Solution:
    def rearrangeCharacters(self, s: str, target: str) -> int:
        cnt1 = Counter(s)      # Count characters in s
        cnt2 = Counter(target)  # Count characters in target
        return min(cnt1[c] // v for c, v in cnt2.items())
```

This implementation successfully leverages the strength of hash tables for counting operations, combined with integer division to satisfy the requirements of the problem efficiently.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Suppose we are given the string `s = "aabbbccd"` and `target = "abc"`. Our task is to determine the maximum number of copies of `target` that can be formed from `s`.

Step 1: First, we use `Counter(s)` to count the occurrences of each character in `s`. This gives us a dictionary `cnt1` which represents our character supply:

- `cnt1 = Counter("aabbbccd")` results in `cnt1 = {'a': 2, 'b': 3, 'c': 2}`.

Step 2: Next, we count the occurrences of each character in `target`. This gives us another dictionary `cnt2` which represents our character demand:

- `cnt2 = Counter("abc")` results in `cnt2 = {'a': 1, 'b': 1, 'c': 1}`.

Step 3: We then iterate over the `cnt2` dictionary and for each character in `target`, we calculate how many times the character can be provided by the supply from `s`. This is done by dividing `cnt1[c]` by `cnt2[c]`:

- For character 'a': the supply is `cnt1['a'] = 2` and the demand is `cnt2['a'] = 1`, so `2 // 1 = 2`. We can form two 'a's.
- For character 'b': the supply is `cnt1['b'] = 3` and the demand is `cnt2['b'] = 1`, so `3 // 1 = 3`. We can form three 'b's.
- For character 'c': the supply is `cnt1['c'] = 2` and the demand is `cnt2['c'] = 1`, so `2 // 1 = 2`. We can form two 'c's.

Step 4: The final step is to find the minimum value among the quotients calculated because the minimum value represents the bottleneck for forming the `target` string fully:

- We take the minimum value of `[2, 3, 2]`, which corresponds to the counts of 'a, 'b,' and 'c' respectively.
- The minimum is `2`, so we can form the `target` string `abc` twice using characters from `s`.

Hence, the result of running our solution would be `2` as that is the maximum number of times we can completely form the string `abc` from the characters in `s = "aabbbccd"`.

Python Solution

```
from collections import Counter  # Import the Counter class from collections module

class Solution:
    def rearrangeCharacters(self, s: str, target: str) -> int:
        # Count the frequency of each character in the original string `s`
        char_count = Counter(s)
        # Count the frequency of each character in the `target` string
        char_count_target = Counter(target)

        # Calculate the minimum number of times the `target` can be formed
        # by dividing the frequency of each character in `s` by the
        # frequency of the same character in `target`, and taking the min.
        return min(char_count_original[char] // count for char, count in char_count_target.items())

# The function rearrangeCharacters takes two strings as parameters: `s` and `target`.
# It utilizes the Counter class to count occurrences of each character.
# Then, it determines the smallest quotient of the character counts from `s` divided by those from `target`,
# which represents the maximum number of times `target` can be formed from `s`.
```

Java Solution

```
class Solution {
    public int rearrangeCharacters(String s, String target) {
        // counts for characters in 's'
        int[] countS = new int[26];
        // counts for characters in 'target'
        int[] countTarget = new int[26];

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

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

        // Initialize the answer with a high value.
        // It represents the maximum number of times 'target' can be formed.
        int maxFormable = Integer.MAX_VALUE;

        // Calculate the number of times 'target' can be formed
        for (int i = 0; i < 26; ++i) {
            if (countTarget[i] > 0) {
                // Find the minimum number of times a character from 'target'
                // can be used based on its frequency in 's'.
                maxFormable = Math.min(maxFormable, countS[i] / countTarget[i]);
            }
        }

        // Return the maximum number of times 'target' can be formed
        return maxFormable;
    }
}
```

C++ Solution

```
#include <string>
#include <algorithm>

class Solution {
public:
    // This function calculates the maximum number of times the target string
    // can be formed using the characters from the string s.
    int rearrangeCharacters(string s, string target) {
        // Initialize character count arrays for s and target strings.
        int counts[26] = {0};
        int countTarget[26] = {0};

        // Count the frequency of each character in the string s.
        for (char ch : s) {
            ++counts[ch - 'a']; // Update the count of the character ch.
        }

        // Count the frequency of each character in the target string.
        for (char ch : target) {
            ++countTarget[ch - 'a']; // Update the count of the character ch.
        }

        // An arbitrary large value chosen as a starting minimum.
        int maxOccurrences = INT_MAX; // Can make the target string at least this many times.

        // Loop through each character from 'a' to 'z'.
        for (int i = 0; i < 26; ++i) {
            // If the current character is in the target string,
            // find the minimum number of times we can use it by comparing
            // the frequency of the character in s and target.
            if (countTarget[i] > 0) {
                maxOccurrences = std::min(maxOccurrences, counts[i] / countTarget[i]);
            }
        }

        // Return the maximum number of times we can form the target string.
        return maxOccurrences;
    }
};
```

Typescript Solution

```
function rearrangeCharacters(source: string, target: string): number {
    // Helper function to get the index using character code.
    // Assumes that the input characters will be lowercase English alphabets.
    const getIndex = (character: string) => character.charCodeAt(0) - 'a'.charCodeAt(0);

    // Counts of characters in the source string.
    const sourceCount = new Array(26).fill(0);
    // Counts of characters in the target string.
    const targetCount = new Array(26).fill(0);

    // Count occurrences of each character in the source string.
    for (const character of source) {
        ++sourceCount[getIndex(character)];
    }

    // Count occurrences of each character in the target string.
    for (const character of target) {
        ++targetCount[getIndex(character)];
    }

    // Initialize the answer to the highest possible number.
    // This value will eventually hold the maximum number of times
    // the 'target' can be formed from 'source'.
    let maxTargetCount = Infinity;

    // Loop through each letter's count in the target string
    // and calculate how many times the target can be created
    // from the source based on the minimum available(if of
    // required characters in the source string.
    for (let i = 0; i < 26; ++i) {
        if (targetCount[i]) {
            maxTargetCount = Math.min(maxTargetCount, Math.floor(sourceCount[i] / targetCount[i]));
        }
    }

    // Return the maximum number of times the target can be formed.
    // If the target cannot be formed even once, it will return 0.
    return maxTargetCount === Infinity ? 0 : maxTargetCount;
}
```

Time and Space Complexity

The given Python code snippet aims to find the maximum number of times the `target` string can be formed from the characters in the string `s`. To do this, the code employs two `Counter` objects from the `collections` module to count the frequency of each character in `s` and `target` and then calculates the minimum number of times the `target` can be formed by the available characters in `s`.

Time Complexity

The time complexity of the code mainly comes from the following parts:

1. The `Counter(s)` operation - Counting the frequency of each character in `s` has a time complexity of `O(n)`, where `n` is the length of `s`.
2. The `Counter(target)` operation - Similarly, counting the frequency of each character in `target` has a time complexity of `O(m)`, where `m` is the length of `target`.
3. The `min(cnt1[c] // v for c, v in cnt2.items())` operation - Iterating over each unique character in `target` and checking its availability in `s` has a time complexity of `O(u)`, where `u` is the number of unique characters in `target`.

Since these operations are sequential and not nested, the overall time complexity is `O(n + m + u)`. However, since `u` cannot exceed the size of the alphabet used (let's say `k` for a fixed-size alphabet), and `m` will always be less than or equal to `n` in the worst case when each character in `s` is part of `target`, we can simplify the time complexity to `O(n)` as `k` is a constant and can be considered negligible.

Space Complexity

The space complexity is determined by:

1. The space required to store the `Counter` objects for `s` and `target` - This would be `O(a + b)`, where `a` is the number of unique characters in `s` and `b` is the number of unique characters in `target`.
2. Since the size of the alphabet is fixed, and hence `a` and `b` will be at most `k` (the alphabet size), we can consider the space complexity to be `O(k)`.

Consequently, the overall space complexity of the code snippet is `O(k)` which is effectively a constant space because the alphabet size does not change with input size.