791. Custom Sort String

Medium Hash Table String Sorting

Problem Description

The problem presents two strings - `order` and `s`. The string `order` contains unique characters that follow a custom sorted order. The task is to rearrange the characters in string `s` such that they follow the same sorting order as the characters in `order`. It is important to preserve the relative order of characters as they appear in `order`. If `s` contains characters that are not present in `order`, these characters should be placed at the end of the result string in any order. The problem requires us to output any valid permutation of `s` that meets this criterion.

Intuition

The intuition behind the solution is to respect the precedence of characters as given in `order`. Since `order` dictates the relative ordering of characters, the solution should build the resulting string by appending characters in the order they appear in `order`.

The first step is to count the occurrences of each character in `s`. We use this information to determine how many times each character should appear in the resulting string. We iterate over `order`, and for each character, we append it to the result as many times as it occurs in `s` (using the count we stored earlier).

We then set the count of those characters to zero to indicate that we have already placed those characters in the result. Finally, if there are any characters left in `s` that were not in `order`, we append them to the end of the result string. Since the order of these leftover characters doesn't matter, we simply append them in any order they occur.

By doing this, we ensure that the characters are sorted according to the order specified by `order`, and if `s` has extra characters not in `order`, they are just tacked on at the end.

Solution Approach

The solution approach involves two primary components to achieve the desired ordering of characters: counting occurrences and ordering as per `order`.

1. **Counting Occurrences**: We utilize the `Counter` class from Python's `collections` module to create a frequency map, `cnt`, which tracks the number of occurrences of each character in `s`. This gives us a dictionary where the keys are the characters from `s`, and the values are the number of times they appear in `s`.
2. **Ordering as Per `order`**: We initialize an empty list called `ans` which will hold the characters in the new order. We iterate through each character `c` in `order`:
   - Append the character `c` to `ans`, multiplied by its count from `cnt` (which is `cnt[c]`). This step ensures that if a character `c` is supposed to appear `n` times, it's added `n` times consecutively to `ans`.
   - Set `cnt[c]` to 0 to denote that we've accounted for all occurrences of `c` as specified by `order`.
3. **Handling Remaining Characters**: After processing all characters in `order`, there might be characters left in `s` that were not present in `order`. To handle such characters, we:
   - Iterate through the remaining items in `cnt`.
   - For every character `c` and its count `v` in `cnt`, append the character `c` to `ans`, multiplied by its count `v`. This ensures that characters not in `order` are placed at the end of `ans`.
4. **Building the Final String**: We use `''.join(ans)` to convert the list of characters `ans` into a string which is the final sorted string according to the custom order specified by `order`.

By following this process, we ensure that all characters in `order` take precedence and are arranged as per their ordering in `order`. Characters not found in `order` follow an ad-hoc order at the end of the sorted string. The solution abides by the problem constraints and provides a correctly ordered permutation of the string `s`.

Example Walkthrough

Let's consider a small example where `order = "cba"` and `s = "abcdabc"`. We want to sort the string `s` such that it follows the order given by `order` and any extra characters are placed at the end.

- Step 1: **Counting Occurrences**: We count the occurrences of each character in `s`. The count will look like this: `cnt = {'a': 2, 'b': 2, 'c': 2, 'd': 1}`.
- Step 2: **Ordering as Per `order`**: Initialize `ans` as an empty list, where we will append our characters.
  1. For character `c` in `order`, which is `'c'`, `cnt[c]` is 2. We append `'c'` twice to `ans`, making it `['c', 'c']`, and set `cnt['c']` to 0.
  2. Next is `'b'` in `order`. `cnt['b']` is 2. We append `'b'` twice to `ans`, which becomes `['c', 'c', 'b', 'b']`, and set `cnt['b']` to 0.
  3. Then for `'a'` in `order`, `cnt['a']` is 2. We append `'a'` twice to `ans`, now `ans` is `['c', 'c', 'b', 'b', 'a', 'a']`, and set `cnt['a']` to 0.
- Step 3: **Handling Remaining Characters**: We have `'d'` left in `cnt` with a count of 1, which was not in `order`. We append `'d'` once to `ans`, resulting in `['c', 'c', 'b', 'b', 'a', 'a', 'd']`.
- Step 4: **Building the Final String**: Join the characters in `ans` to get the final sorted string: `'ccbbaad'`.

The result `'ccbbaad'` is one valid permutation of `s` where the order `'cba'` is followed, and the character `'d'` (which is not in `order`) is placed at the end.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def customSortString(self, order: str, s: str) -> str:
        # Count the occurrences of each character in the string s
        char_count = Counter(s)

        # Initialize the answer as an empty list; it will store the sorted characters
        sorted_characters = []

        # Add characters to the sorted characters list following the order specified.
        # Multiply the character by its count to add it that many times.
        for char in order:
            sorted_characters.append(char * char_count[char])
            # Set the count for this character to 0 since it's been handled
            char_count[char] = 0

        # Add remaining characters that were not in 'order' to the list.
        # These are added at the end in their original order of occurrence.
        for char, count in char_count.items():
            sorted_characters.append(char * count)

        # Join all the characters in the sorted_characters list to form the sorted string
        return ''.join(sorted_characters)
```

Java

```java
class Solution {
    public String customSortString(String order, String str) {
        // Create an array to count each character's frequency in 'str'.
        int[] frequency = new int[26];

        // Iterate through the 'str' to populate the character frequencies.
        for (int i = 0; i < str.length(); ++i) {
            frequency[str.charAt(i) - 'a']++;
        }

        // Use StringBuilder to build the sorted string for efficient string manipulation.
        StringBuilder sortedStringBuilder = new StringBuilder();

        // Iterate through the 'order' string.
        for (int i = 0; i < order.length(); ++i) {
            char currentChar = order.charAt(i);

            // Append the current character from 'order' to the sorted string
            // as many times as it appears in 'str'.
            while (frequency[currentChar - 'a'] > 0) {
                sortedStringBuilder.append(currentChar);
                frequency[currentChar - 'a']--;
            }
        }

        // Append the remaining characters that were not in 'order' to the sorted string.
        for (int i = 0; i < 26; ++i) {
            while (frequency[i] > 0) {
                sortedStringBuilder.append((char) ('a' + i));
                frequency[i]--;
            }
        }

        // Return the final sorted string.
        return sortedStringBuilder.toString();
    }
}
```

C++

```cpp
class Solution {
public:
    string customSortString(string order, string str) {
        // Create an array to keep count of each character's occurrence in str
        int charCounts[26] = {0};

        // Fill the array with the counts of each character
        for (char& c : str) {
            charCounts[c - 'a']++;
        }

        // This string will store the result
        string sortedStr;

        // Iterate over the 'order' string and append the characters to 'sortedStr' in the order they appear
        for (char& c : order) {
            while (charCounts[c - 'a']-- > 0) {
                sortedStr += c;
            }
        }

        // Append characters that did not appear in 'order' to the end of 'sortedStr', in their original order
        for (int i = 0; i < 26; ++i) {
            // Check if the character is present in 'str' and was not included via 'order'
            if (charCounts[i] > 0) {
                // Append the character (i + 'a') 'charCounts[i]' times to the 'sortedStr'
                sortedStr += string(charCounts[i], i + 'a');
            }
        }

        // Return the custom sorted string
        return sortedStr;
    }
};
```

TypeScript

```typescript
function customSortString(order: string, str: string): string {
    // Function to convert a character to its alphabet index (0 for 'a', 1 for 'b', etc.)
    const charToIndex = (char: string) => char.charCodeAt(0) - 'a'.charCodeAt(0);

    // Initialize an array to keep track of the count of each character in str
    const charCount = new Array(26).fill(0);
    // Populate the character count array
    for (const char of str) {
        charCount[charToIndex(char)]++;
    }

    // Initialize an array to construct the final sorted string
    const sortedChars: string[] = [];

    // Add characters to the sortedChars array in the order specified by 'order'
    for (const char of order) {
        const index = charToIndex(char);
        sortedChars.push(char.repeat(charCount[index])); // Add ordered characters
        charCount[index] = 0; // Reset the count since these characters are now processed
    }

    // Add the remaining characters that were not specified in 'order', in lexicographical order
    for (let i = 0; i < 26; i++) {
        if (charCount[i] === 0) continue; // Skip characters with a count of zero
        sortedChars.push(
            String.fromCharCode('a'.charCodeAt(0) + i).repeat(charCount[i])
        );
    }

    // Join the array of sorted characters into a single string and return it
    return sortedChars.join("");
}
```

```python
from collections import Counter

class Solution:
    def customSortString(self, order: str, s: str) -> str:
        # Count the occurrences of each character in the string s
        char_count = Counter(s)

        # Initialize the answer as an empty list; it will store the sorted characters
        sorted_characters = []

        # Add characters to the sorted characters list following the order specified.
        # Multiply the character by its count to add it that many times.
        for char in order:
            sorted_characters.append(char * char_count[char])
            # Set the count for this character to 0 since it's been handled
            char_count[char] = 0

        # Add remaining characters that were not in 'order' to the list.
        # These are added at the end in their original order of occurrence.
        for char, count in char_count.items():
            sorted_characters.append(char * count)

        # Join all the characters in the sorted_characters list to form the sorted string
        return ''.join(sorted_characters)
```

Time and Space Complexity

Time Complexity

The time complexity of the function can be analyzed in two main steps involved in the `customSortString` method:

1. Counting occurrences of all characters in the input string `s`. This is done with the `Counter` collection from Python's `collections` module, which will take $O(N)$ time, where N is the length of the string `s`.
2. Constructing the sorted string:
   - For each character in `order`, it appends that character to `ans` as many times as it occurs in `s`. This process takes $O(M)$ time, where M is the length of the `order` string, assuming that appending to the list and setting the count to zero are constant time operations.
   - Furthermore, it iterates over the items in `cnt` to append the remaining characters. This iteration will, again, take at most $O(N)$ time since the number of keys in `cnt` cannot exceed the number of characters in `s`.

Putting this together, the total time complexity is $O(N) + O(M) + O(N)$, which simplifies to $O(N + M)$.

Space Complexity

The space complexity of the given code is influenced by the following:

1. The `Counter` object `cnt`, which stores a frequency count of characters in `s`. In the worst case, if all characters in `s` are unique, the space taken would be $O(N)$.
2. The `ans` list, which collects the sorted characters. In the worst case, when no characters are duplicated in `s`, this would again require $O(N)$ space.

Therefore, the maximum space requirement is for storing the `Counter` object and the output list, which adds up to $O(N)$ space complexity.