451. Sort Characters By Frequency

Medium Hash Table String Bucket Sort Counting Sorting Heap (Priority Queue)

Leetcode Link

Problem Description

The problem requires us to tackle a string sorting task based on a non-standard criterion: the frequency of each character in the string. Specifically, the goal is to reorder the string so that the characters that occur most frequently are placed first. If two characters have the same frequency, they can be in any order with respect to each other. The final output must be a string where the sorted order reflects these frequency criteria.

Intuition

The intuitive approach to this problem involves counting how often each character appears in the string, then sorting the characters based on these counts. This is typically a two-step process:

1. **Count the occurrences:** We need to go through the string and count the occurrences of each character. This can be done efficiently by using a hash table or a counter data structure where each character is a key, and its count is the value.
2. **Sort based on counts:** Once we have the counts, the next step is to sort the characters by these counts in descending order. We want the characters with higher counts to come first.

With these counts, we can construct a new string. We do this by iterating over each unique character, repeating the character by its count (since sorting by frequency means if a character appears (n) times, it should also appear (n) times consecutively in the final string), and concatenating these repetitions to form the final string.

In the provided solution:

- The `Counter` from the collections module is used to count occurrences of each character.
- The `sorted()` function sorts the items in the counter by their counts (values), with the sort being in descending order because of the negative sign in the sort key `-x[1]`.
- The sorted items are then concatenated to create the final string through a string join operation, which combines the characters multiplied by their frequencies.

This method is consistent with the requirements and efficiently achieves the sorting based on frequency.

Solution Approach

The solution makes use of a few key concepts in Python to address the problem:

- **Counter Data Structure:** The `Counter` class from the `collections` module is perfect for counting the frequency of characters because it automatically builds a hash map (dictionary) where characters are keys and their counts are values.

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

  Here, `s` is the input string, and `cnt` becomes a `Counter` object holding counts of each character.

- **Sorting by Frequency:** The `sorted()` function is used to sort the characters based on their frequency.

```
sorted(cnt.items(), key=lambda x: -x[1])
```

  `cnt.items()` provides a sequence of `(character, count)` pairs. The `key` argument to `sorted()` specifies that sorting should be based on the count, which is the second item in each pair (`x[1]`). The negative sign ensures that the sorting is in decreasing order of frequency.

- **String Joining and Character Multiplication:** Python's expressive syntax allows us to multiply a string by an integer to repeat it, and the `join()` method of a string allows us to concatenate an iterable of strings:

```
return ''.join(c * v for c, v in sorted(cnt.items(), key=lambda x: -x[1]))
```

  For each character `c` and its count `v`, `c * v` creates a string where `c` is repeated `v` times. The `join()` method is used to concatenate all these strings together without any separator (`''`), creating the final sorted string.

These steps are combined into a concise one-liner inside the `frequencySort` method of the `Solution` class. This is efficient because it leverages Python's built-in data structures and functions that are implemented in C under the hood, thus being quite fast.

```
class Solution:
    def frequencySort(self, s: str) -> str:
        cnt = Counter(s)
        return ''.join(c * v for c, v in sorted(cnt.items(), key=lambda x: -x[1]))
```

The approach works well and guarantees that the most frequent characters will be placed at the beginning of the resulting string, while less frequent characters will follow, adhering to the problem's constraints.

Example Walkthrough

Let's run through a small example to illustrate how the solution approach works. Suppose our input string is `s = "tree"`.

1. **Count the occurrences of each character:** We use the `Counter` class to count the characters.

```
from collections import Counter
cnt = Counter("tree")
# cnt is now Counter({'t': 1, 'r': 1, 'e': 2})
```

  In the string "tree", the character 'e' appears twice, while 't' and 'r' each appear once.

2. **Sort characters by frequency:** We then use the `sorted()` function to sort these characters by their frequency in descending order.

```
sorted_characters = sorted(cnt.items(), key=lambda x: -x[1])
# sorted_characters is now [('e', 2), ('t', 1), ('r', 1)]
```

  Here, we use a lambda function as the key to sort by the counts—the negative sign ensures it is in descending order.

3. **Construct the new string based on frequency:** Finally, we iterate over the sorted character-count pairs and repeat each character by its frequency, joining them to form the final result.

```
result_string = ''.join(c * v for c, v in sorted_characters)
# result_string is "eett" or "eetr" or "tree" or "ttee", etc.
```

  Since 'e' has the highest frequency, it comes first. 't' and 'r' have the same frequency, so their order with respect to each other does not matter in the final output. The result can be "eett", "eetr", "tree", or "ttee" because the order of characters with the same frequency is not specified.

Putting all this within the class method `frequencySort` would look like this:

```
class Solution:
    def frequencySort(self, s: str) -> str:
        cnt = Counter(s)
        return ''.join(c * v for c, v in sorted(cnt.items(), key=lambda x: -x[1]))
```

By applying this method to our example:

```
solution = Solution()
print(solution.frequencySort("tree"))
```

We will get a string that has 'e' characters first because they have the highest frequency, followed by 't' and 'r' in any order, which may result in one of the possible outcomes such as "eett", "eetr", "tree", or "ttee".

Python Solution

```
from collections import Counter

class Solution:
    def frequencySort(self, s: str) -> str:
        # Count the frequency of each character in the input string
        char_frequency = Counter(s)

        # Sort the characters based on frequency in descending order
        sorted_characters = sorted(char_frequency.items(), key=lambda item: -item[1])

        # Create a string with characters repeated by their frequency
        frequency_sorted_string = ''.join(character * frequency for character, frequency in sorted_characters)

        return frequency_sorted_string

# Example usage:
# sol = Solution()
# result = sol.frequencySort("tree")
# print(result)  # Outputs a string with characters sorted by frequency, e.g. "eetr"
```

Java Solution

```
import java.util.*;

class Solution {
    public String frequencySort(String s) {
        // Initialize a hash map to store frequency of each character
        Map<Character, Integer> frequencyMap = new HashMap<>(52);

        // Loop through all the characters in the string to fill the frequency map
        for (int i = 0; i < s.length(); ++i) {
            // Merge the current character into the map, increasing its count by 1
            frequencyMap.merge(s.charAt(i), 1, Integer::sum);
        }

        // Create a list to store the characters (for sorting purposes)
        List<Character> characters = new ArrayList<>(frequencyMap.keySet());

        // Sort the character list based on their frequencies in descending order
        characters.sort((a, b) -> frequencyMap.get(b) - frequencyMap.get(a));

        // Use StringBuilder to build the result string
        StringBuilder sortedString = new StringBuilder();

        // Loop through the sorted list of characters
        for (char c : characters) {
            // Append each character to the StringBuilder based on its frequency
            for (int frequency = frequencyMap.get(c); frequency > 0; --frequency) {
                sortedString.append(c);
            }
        }

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

C++ Solution

```
#include <string>
#include <unordered_map>
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to sort characters by frequency of appearance in a string
    string frequencySort(string s) {
        // Create a hash map to store the frequency of appearance of each character
        unordered_map<char, int> frequencyMap;
        // Calculate the frequency of each character in the string
        for (char ch : s) {
            ++frequencyMap[ch];
        }

        // Create a vector to store unique characters
        vector<char> uniqueChars;
        // Populate the vector with the keys from the frequencyMap
        for (auto& keyValue : frequencyMap) {
            uniqueChars.push_back(keyValue.first);
        }

        // Sort the unique characters based on their frequency
        sort(uniqueChars.begin(), uniqueChars.end(), [&](char a, char b) {
            return frequencyMap[a] > frequencyMap[b];
        });

        // Create a result string to store the sorted characters by frequency
        string result;
        // Go through each character and append it to the result string, multiplied by its frequency
        for (char ch : uniqueChars) {
            result += string(frequencyMap[ch], ch);
        }

        // Return the result string
        return result;
    }
};
```

Typescript Solution

```
// Function to sort the characters in a string by frequency of appearance in descending order.
function frequencySort(s: string): string {
    // Create a map to hold character frequencies.
    const charFrequencyMap: Map<string, number> = new Map();

    // Iterate over each character in the input string.
    for (const char of s) {
        // Update the frequency count for each character.
        charFrequencyMap.set(char, (charFrequencyMap.get(char) || 0) + 1);
    }

    // Convert map keys to an array, sort the array by frequency in descending order.
    const sortedCharacters = Array.from(charFrequencyMap.keys()).sort(
        (charA, charB) => charFrequencyMap.get(charB)! - charFrequencyMap.get(charA)!
    );

    // Initialize an array to hold the sorted characters by frequency.
    const sortedArray: string[] = [];

    // Build a string for each character, repeated by its frequency.
    for (const char of sortedCharacters) {
        sortedArray.push(char.repeat(charFrequencyMap.get(char)!));
    }

    // Join the array of strings into a single string and return it.
    return sortedArray.join('');
}
```

Time and Space Complexity

Time Complexity:

The time complexity of the provided code can be analyzed as follows:

1. Counting the frequency of each character - The `Counter` from the `collections` module iterates through the string `s` once to count the frequency of each character. This operation has a time complexity of $O(n)$, where n is the length of the string `s`.
2. Sorting the counted characters - The `sorted` function is used to sort the items in the counter based on their frequency (the value in the key-value pair). Sorting in python is typically implemented with the Timsort algorithm, which has a time complexity of $O(m \log m)$, where m is the number of unique characters in the string `s`.

Overall, the dominating factor is the sort operation, so the total time complexity is $O(m \log m + n)$. However, since m can be at most n in cases where all characters are unique, the time complexity is often described as $O(n \log n)$ for practical worst-case scenarios.

Space Complexity:

The space complexity of the code is analyzed as follows:

1. Counter dictionary - The `Counter` constructs a dictionary with m entries, where m is the number of unique characters in the string `s`. This space usage is $O(m)$.
2. Sorted list - The `sorted` function generates a list of tuples which is essentially the items of the counter sorted based on their frequency. This also takes $O(m)$ space.
3. Output string - The output string is formed by joining individual characters multiplied by their frequency. The length of the resulting string is equal to `s`, the length of the input string. Hence, the space required for the output string is $O(n)$.

Since n can be at most n, the overall space complexity is $O(n)$ considering the storage for the output string and the data structures used for counting and sorting.