451. Sort Characters By Frequency Counting Sorting Heap (Priority Queue) Medium **Hash Table Bucket Sort** 

String )

## **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

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.

The intuitive approach to this problem involves counting how often each character appears in the string, then sorting the

- 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

**Solution Approach** 

sign in the sort key -x[1].

their frequencies.

characters based on these counts. This is typically a two-step process:

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

• The sorted items are then concatenated to create the final string through a string join operation, which combines the characters multiplied by

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)

order of frequency.

fast.

class Solution:

def frequencySort(self, s: str) -> str:

# cnt is now Counter({'t': 1, 'r': 1, 'e': 2})

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

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.

**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

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:

concatenate all these strings together without any separator (''), creating the final sorted string.

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

while less frequent characters will follow, adhering to the problem's constraints.

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

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

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

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,

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

Let's run through a small example to illustrate how the solution approach works. Suppose our input string is s = "tree". **Count the occurrences of each character:** We use the **Counter** class to count the characters. from collections import Counter cnt = Counter("tree")

Sort characters by frequency: We then use the sorted() function to sort these characters by their frequency in descending

# sorted\_characters is now [('e', 2), ('t', 1), ('r', 1)]

order.

class Solution:

**Python** 

**Example Walkthrough** 

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

Construct the new string based on frequency: Finally, we iterate over the sorted character-count pairs and repeat 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:

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

frequency\_sorted\_string = ''.join(character \* frequency for character, frequency in sorted\_characters)

Since 'e' has the highest frequency, it comes first. 't' and 'r' have the same frequency, so their order with respect to each

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

may result in one of the possible outcomes such as "eett", "eetr", "tree", or "ttee".

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

def frequencySort(self, s: str) -> str:

By applying this method to our example:

cnt = Counter(s)

Solution Implementation

from collections import Counter

return frequency\_sorted\_string

for (int i = 0; i < s.length(); ++i) {</pre>

frequencyMap.merge(s.charAt(i), 1, Integer::sum);

// Use StringBuilder to build the result string

// Create a vector to store unique characters

uniqueChars.push\_back(keyValue.first);

for (auto& keyValue : frequencyMap) {

// Populate the vector with the keys from the frequencyMap

// Sort the unique characters based on their frequency

return frequencyMap[a] > frequencyMap[b];

result += string(frequencyMap[ch], ch);

sort(uniqueChars.begin(), uniqueChars.end(), [&](char a, char b) {

// Create a result string to store the sorted characters by frequency

vector<char> uniqueChars;

for (char ch : uniqueChars) {

const sortedArray: string[] = [];

return sortedArray.join('');

def frequencySort(self, s: str) -> str:

return frequency\_sorted\_string

char\_frequency = Counter(s)

from collections import Counter

for (const char of sortedCharacters) {

});

);

class Solution:

# Example usage:

# sol = Solution()

**Time Complexity:** 

string result;

// Create a list to store the characters (for sorting purposes)

List<Character> characters = new ArrayList<>(frequencyMap.keySet());

characters.sort((a, b) -> frequencyMap.get(b) - frequencyMap.get(a));

// Sort the character list based on their frequencies in descending order

# result = sol.frequencySort("tree")

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

# print(result) # Outputs a string with characters sorted by frequency, e.g. "eetr"

// Merge the current character into the map, increasing its count by 1

```
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
```

# Example usage:

Java

# sol = Solution()

import java.util.\*;

class Solution {

```
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++
#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];
```

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

// Build a string for each character, repeated by its frequency.

// Join the array of strings into a single string and return it.

# Count the frequency of each character in the input string

# Sort the characters based on frequency in descending order

# Create a string with characters repeated by their frequency

sorted\_characters = sorted(char\_frequency.items(), key=lambda item: -item[1])

frequency\_sorted\_string = ''.join(character \* frequency for character, frequency in sorted\_characters)

sortedArray.push(char.repeat(charFrequencyMap.get(char)!));

// Go through each character and append it to the result string, multiplied by its frequency

```
# result = sol.frequencySort("tree")
# print(result) # Outputs a string with characters sorted by frequency, e.g. "eetr"
Time and Space Complexity
```

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

## 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

string s. This space usage is O(m).

complexity of O(m log m), where m is the number of unique characters in the string s.

The space complexity of the code is analyzed as follows:

Overall, the dominating factor is the sort operation, so the total time complexity is 0(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

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.

- scenarios. **Space Complexity:**
- 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.

Output string - The output string is formed by joining individual characters multiplied by their frequency. The length of the

Counter dictionary - The Counter constructs a dictionary with m entries, where m is the number of unique characters in the

resulting string is equal to n, the length of the input string. Hence, the space required for the output string is 0(n). Since m 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.