Documentation: Frequency Sort Algorithm

Table of Contents

1. **Problem Statement**
2. **Intuition**
3. **Key Observations**
4. **Approach**
5. **Edge Cases**
6. **Complexity Analysis**
    - Time Complexity
    - Space Complexity
7. **Alternative Approaches**
8. **Test Cases**
9. **Final Thoughts**

1. Problem Statement

You are given a string s. Your task is to sort the characters of the string in **decreasing order** based on their frequency of occurrence. The frequency of a character is the number of times it appears in the string. If there are multiple possible answers, any of them can be returned.

Input: A string s containing uppercase and lowercase English letters and digits.

Output: A string where characters are sorted by their frequency of occurrence in decreasing order.

2. Intuition

The problem can be intuitively solved by first counting the frequency of each character, sorting those characters based on their frequency in descending order, and then building a new string where each character appears as many times as its frequency. The sorting step ensures that the characters with the highest frequency appear first, and the construction step creates the final string.

3. **Key Observations**

- The frequency of each character is crucial for determining the order of characters in the final string.
- Sorting the characters by frequency in descending order requires comparing the frequency of each character.
- In cases where multiple characters have the same frequency, their order among themselves doesn't matter, so any order can be returned.

4. **Approach**

The approach to solving this problem involves the following steps:

- **Count the Frequency of Each Character**: Use a frequency counter (such as Python's Counter) to determine how many times each character appears in the string.
- **Sort Characters by Frequency**: Sort the characters based on their frequency in descending order. In Python, this can be done using the sorted() function, using the frequency as the key.
- **Construct the Resulting String**: Once sorted, create the final string by repeating each character according to its frequency.

5. **Edge Cases**

- **Single character string**: If the string contains only one character, the output will be the same as the input since no sorting is needed.
- **All characters have the same frequency**: If all characters appear with equal frequency, the output will be any permutation of the input characters.
- **Empty string**: An empty string should return an empty string.
- **String with mixed case**: Since the characters are treated as distinct based on their case, A and a are considered different characters.

6. **Complexity Analysis**

Time Complexity

- **Counting the frequency**: O(n) where n is the length of the string. The Counter function iterates over each character of the string once.
- **Sorting the frequency list**: O(k log k) where k is the number of unique characters in the string. In the worst case, k is 62 (for uppercase and lowercase letters, and digits), which is much smaller than n.
- **Building the result string**: O(n) where n is the length of the string, since we concatenate the characters repeatedly according to their frequency.

Thus, the total time complexity is O(n + k log k).

Space Complexity

- **Frequency map**: The space complexity for storing the frequency map is O(k) where k is the number of unique characters in the string.
- **Result string**: The space for the result string is O(n) since it stores the entire input string's characters repeated by their frequency.

Thus, the total space complexity is O(n + k).

7. **Alternative Approaches**

i. **Using a Max-Heap**:
   - Instead of sorting the characters directly, you can use a max-heap (priority queue) to store characters by frequency. This will help efficiently extract characters with the highest frequency.
   - The time complexity for building the heap would be O(n log k), where k is the number of unique characters.

ii. **Bucket Sort**:
   - If the frequency range of characters is bounded (such as only letters and digits), you can use a bucket sort approach. This approach might be more efficient when the frequency range is limited.

8. **Test Cases**

i. **Test Case 1:**

a. Input: "tree"

b. Output: "eert" or "eetr"

c. Explanation: 'e' appears twice, while 't' and 'r' appear once, so 'e' comes first.

ii. **Test Case 2:**

a. Input: "cccaaa"

b. Output: "aaaccc" or "cccaaa"

c. Explanation: Both 'c' and 'a' appear three times, so the result can be any order of them.

iii. **Test Case 3:**

a. Input: "Aabb"

b. Output: "bbAa" or "bbaA"

c. Explanation: 'b' appears twice, 'A' appears once, and 'a' appears once. The output could be any valid ordering.

iv. **Test Case 4:**

a. Input: "a"

b. Output: "a"

c. Explanation: The string contains only one character.

v. **Test Case 5:**

a. Input: "abc"

b. Output: "abc" or any permutation of 'abc'

c. Explanation: All characters appear once, so the order doesn't matter.

vi. **Test Case 6:**

a. Input: "" (empty string)

b. Output: ""

c. Explanation: An empty string should return an empty string.

9. Final Thoughts

- This algorithm is efficient for the given problem constraints. With a time complexity of O(n + k log k) and a space complexity of O(n + k), it should work well within the problem's limits.
- The approach leverages Python's built-in Counter for frequency counting and sorting, making the solution concise and easy to implement.
- The algorithm works with any string containing English letters and digits, and it handles edge cases such as single character strings and empty strings gracefully.