<https://leetcode.com/problems/minimum-deletions-to-make-character-frequencies-unique/description/>

A string s is called good if there are no two different characters in s that have the same frequency.

Given a string s, return the minimum number of characters you need to delete to make s good.

The frequency of a character in a string is the number of times it appears in the string. For example, in the string "aab", the frequency of 'a' is 2, while the frequency of 'b' is 1.

**Example 1:**

**Input**: s = "aab"

**Output**: 0

**Explanation**: s is already good.

**Example 2:**

**Input:** s = "aaabbbcc"

**Output:** 2

**Explanation:** You can delete two 'b's resulting in the good string "aaabcc".Another way it to delete one 'b' and one 'c' resulting in the good string "aaabbc".

**Example 3:**

**Input:** s = "ceabaacb"

**Output:** 2

**Explanation:** You can delete both 'c's resulting in the good string "eabaab".Note that we only care about characters that are still in the string at the end (i.e. frequency of 0 is ignored).

**Constraints:**

1 <= s.length <= 10^5

s contains only lowercase English letters.

**Attempt 1: 2024-12-27**

**Solution 1: Greedy + Hash Table (60 min)**

class Solution {

    public int minDeletions(String s) {

        int[] freq = new int[26];

        for(char c : s.toCharArray()) {

            freq[c - 'a']++;

        }

        // For each 26 characters, check if it's count is already used.

        // If so, delete characters until you find unused count, or reach zero.

        int count = 0;

        Set<Integer> used = new HashSet<>();

        for(int i = 0; i < 26; i++) {

            while(freq[i] > 0 && !used.add(freq[i])) {

                freq[i]--;

                count++;

            }

        }

        return count;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/minimum-deletions-to-make-character-frequencies-unique/solutions/927549/c-java-python-3-greedy/>

Greedy works since we can only delete characters (we cannot add or replace characters).

So, count each character first. For each 26 characters, check if it's count is already used. If so, delete characters until you find unused count, or reach zero.

public int minDeletions(String s) {

int cnt[] = new int[26], res = 0;

Set<Integer> used = new HashSet<>();

for (int i = 0; i < s.length(); ++i)

++cnt[s.charAt(i) - 'a'];

for (int i = 0; i < 26; ++i) {

while (cnt[i] > 0 && !used.add(cnt[i])) {

--cnt[i];

++res;

}

}

return res;

}

**Refer to chatGPT**

**Solution Explanation:**

Count Character Frequencies:

Use an array or HashMap to count the frequency of each character.

Ensure Unique Frequencies:

Use a Set or similar structure to track used frequencies.

**If a frequency is already used, decrement it until it becomes unique or reaches zero (remove the character entirely).**

Count Deletions:

Keep track of how many deletions were made while reducing frequencies.

public class Solution {

public int minDeletions(String s) {

// Count the frequency of each character

int[] freq = new int[26];

for (char c : s.toCharArray()) {

freq[c - 'a']++;

}

// Track used frequencies

HashSet<Integer> usedFreq = new HashSet<>();

int deletions = 0;

for (int count : freq) {

while (count > 0 && usedFreq.contains(count)) {

count--; // Reduce the frequency

deletions++; // Increment deletions

}

if (count > 0) {

usedFreq.add(count); // Mark this frequency as used

}

}

return deletions;

}

}

**Key Points:**

Character Frequency Calculation:

freq[c - 'a']++: Maps each character to an index (0–25) for efficient counting.

Unique Frequency Tracking:

A HashSet is used to ensure no two characters share the same frequency.

Decrement Until Unique:

If a frequency already exists in usedFreq, decrement the count and increment the deletion counter.

Final Output:

The total deletions required to make all frequencies unique is stored in deletions.

**Complexity:**

Time Complexity:

O(n+k^2), where nnn is the length of the string and kkk is the maximum frequency of any character. The k2k^2k2 comes from potentially decrementing frequencies repeatedly.

In practice, kkk is much smaller than nnn (bounded by the length of the string).

Space Complexity:

O(k), where kkk is the number of distinct character frequencies (size of the HashSet).

**Refer to**

[L1423.Maximum Points You Can Obtain from Cards (Ref.L1658)](note://WEBb70d72cb469c39161ffaeb2d08c64a44)

[L2091.Removing Minimum and Maximum From Array](note://WEB76fa55ddd614833fd1c0727f35e2377e)

[L2423.Remove Letter To Equalize Frequency](note://WEB47fa1ee932a223505bea1cda4058949f)