<https://leetcode.com/problems/sort-characters-by-frequency/description/>

Given a string s, sort it in **decreasing order** based on the **frequency** of the characters. The **frequency** of a character is the number of times it appears in the string.

Return *the sorted string*. If there are multiple answers, return *any of them*.

**Example 1:**

Input: s = "tree"

Output: "eert"

Explanation: 'e' appears twice while 'r' and 't' both appear once.

So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.

**Example 2:**

Input: s = "cccaaa"

Output: "aaaccc"

Explanation: Both 'c' and 'a' appear three times, so both "cccaaa" and "aaaccc" are valid answers.

Note that "cacaca" is incorrect, as the same characters must be together.

**Example 3:**

Input: s = "Aabb"

Output: "bbAa"

Explanation: "bbaA" is also a valid answer, but "Aabb" is incorrect.

Note that 'A' and 'a' are treated as two different characters.

**Constraints:**

* 1 <= s.length <= 5 \* 105
* s consists of uppercase and lowercase English letters and digits.

**Attempt 1: 2023-11-10**

**Solution 1: Priority Queue (10 min)**

class Solution {

public String frequencySort(String s) {

Map<Character, Integer> freq = new HashMap<>();

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

freq.put(c, freq.getOrDefault(c, 0) + 1);

}

PriorityQueue<Map.Entry<Character, Integer>> maxPQ = new PriorityQueue<>((a, b) -> b.getValue() - a.getValue());

for(Map.Entry<Character, Integer> e : freq.entrySet()) {

maxPQ.offer(e);

}

StringBuilder sb = new StringBuilder();

while(!maxPQ.isEmpty()) {

Map.Entry<Character, Integer> e = maxPQ.poll();

for(int i = 0; i < e.getValue(); i++) {

sb.append(e.getKey());

}

}

return sb.toString();

}

}

Time Complexity: O(N\*logM), not O(N\*logN) but rather O(N\*logM) where M is number of distinct characters. It's expected that number of district characters are gonna be much less than length of the string. So assuming logm part to be constant is reasonable. The first solution takes O(N) additional space, which can be problematic if the input string is very large.

Space Complexity: O(N)

**Solution 2: Bucket Sort (10 min)**

class Solution {

public String frequencySort(String s) {

Map<Character, Integer> freq = new HashMap<>();

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

freq.put(c, freq.getOrDefault(c, 0) + 1);

}

List<Character>[] bucket = new List[s.length() + 1];

for(Map.Entry<Character, Integer> e : freq.entrySet()) {

int f = e.getValue();

if(bucket[f] == null) {

bucket[f] = new ArrayList<>();

}

bucket[f].add(e.getKey());

}

StringBuilder sb = new StringBuilder();

for(int i = bucket.length - 1; i >= 0; i--) {

if(bucket[i] != null) {

for(int j = 0; j < bucket[i].size(); j++) {

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

sb.append(bucket[i].get(j));

}

}

}

}

return sb.toString();

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Follow Up: Keep the original character order in given string if same frequency for two characters (10 min)**

class Solution {

public String frequencySort(String s) {

Map<Character, int[]> freq = new HashMap<>();

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

char c = s.charAt(i);

if(!freq.containsKey(c)) {

freq.put(c, new int[] {1, i});

} else {

int[] tmp = freq.get(c);

tmp[0]++;

freq.put(c, tmp);

}

}

// If characters have same frequency then sort based on original

// first occurrence index in original string

PriorityQueue<Map.Entry<Character, int[]>> maxPQ = new PriorityQueue<>((a, b) -> a.getValue()[0] == b.getValue()[0] ? a.getValue()[1] - b.getValue()[1] : b.getValue()[0] - a.getValue()[0]);

for(Map.Entry<Character, int[]> e : freq.entrySet()) {

maxPQ.offer(e);

}

StringBuilder sb = new StringBuilder();

while(!maxPQ.isEmpty()) {

Map.Entry<Character, int[]> e = maxPQ.poll();

for(int i = 0; i < e.getValue()[0]; i++) {

sb.append(e.getKey());

}

}

return sb.toString();

}

}

Time Complexity: O(N\*logM), not O(N\*logN) but rather O(N\*logM) where M is number of distinct characters. It's expected that number of district characters are gonna be much less than length of the string. So assuming logm part to be constant is reasonable. The first solution takes O(N) additional space, which can be problematic if the input string is very large.

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/sort-characters-by-frequency/solutions/93420/java-o-n-bucket-sort-solution-o-nlogm-priorityqueue-solution-easy-to-understand/>

The logic is very similar to NO.347 and here we just use a map a count and according to the frequency to put it into the right bucket. Then we go through the bucket to get the most frequently character and append that to the final stringbuilder.

public class Solution {

public String frequencySort(String s) {

Map<Character, Integer> map = new HashMap<>();

for (char c : s.toCharArray())

map.put(c, map.getOrDefault(c, 0) + 1);

List<Character> [] bucket = new List[s.length() + 1];

for (char key : map.keySet()) {

int frequency = map.get(key);

if (bucket[frequency] == null) bucket[frequency] = new ArrayList<>();

bucket[frequency].add(key);

}

StringBuilder sb = new StringBuilder();

for (int pos = bucket.length - 1; pos >= 0; pos--)

if (bucket[pos] != null)

for (char c : bucket[pos])

for (int i = 0; i < pos; i++)

sb.append(c);

return sb.toString();

}

}

And we have normal way using PriorityQueue as follows:

according to user "orxanb", O(n) ignore logm since m is the distinguish character, can be O(1) since only 26 letters. So the overall time complexity should be O(n), the same as the buck sort with less memory use.

public class Solution {

public String frequencySort(String s) {

Map<Character, Integer> map = new HashMap<>();

for (char c : s.toCharArray())

map.put(c, map.getOrDefault(c, 0) + 1);

PriorityQueue<Map.Entry<Character, Integer>> pq = new PriorityQueue<>((a, b) -> b.getValue() - a.getValue());

pq.addAll(map.entrySet());

StringBuilder sb = new StringBuilder();

while (!pq.isEmpty()) {

Map.Entry e = pq.poll();

for (int i = 0; i < (int)e.getValue(); i++)

sb.append(e.getKey());

}

return sb.toString();

}

}

There is a follow up if you are interested, when same frequency we need to maintain the same sequence as the character show in the original string, the solution is add a index as a secondary sort if the frequency is same, code as below:

public static String frequencySort(String s) {

Map<Character, int[]> map = new HashMap<>();

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

char c = s.charAt(i);

if (!map.containsKey(c)) map.put(c, new int[]{1, i});

else {

int[] freqAndSeq = map.get(c);

freqAndSeq[0]++;

map.put(c, freqAndSeq);

}

}

PriorityQueue<Map.Entry<Character, int[]>> pq = new PriorityQueue<>((a, b) ->

a.getValue()[0] == b.getValue()[0] ? a.getValue()[1] - b.getValue()[1] : b.getValue()[0] - a.getValue()[0]);

pq.addAll(map.entrySet());

StringBuilder sb = new StringBuilder();

while (!pq.isEmpty()) {

Map.Entry<Character, int[]> e = pq.poll();

for (int i = 0; i < e.getValue()[0]; i++)

sb.append(e.getKey());

}

return sb.toString();

}