1370. Increasing Decreasing String String

Counting

## **Problem Description**

Hash Table

Easy

the smallest and largest characters from the string according to given rules and appending them to form a new result string. The rules for picking characters are as follows: 1. Select the smallest character from s and append it to the result. 2. Choose the next-smallest character that is greater than the last appended one and append it.

The given problem requires us to reorder a given string s following a specific algorithm. The algorithm involves repeatedly picking

- 3. Continue with step 2 until no more characters can be picked.
- 4. Select the largest character from s and append it to the result.
- 5. Choose the next-largest character that is smaller than the last appended one and append it.
- 7. Repeat steps 1 to 6 until all characters from s have been picked and appended.

6. Continue with step 5 until no more characters can be picked.

- The algorithm allows for any occurrence of the smallest or largest character to be chosen if there are multiple.
- The solution to this problem uses a frequency counter to keep track of how many times each character appears in the string s.

We use an array counter of length 26 to represent the frequency of each lowercase English letter.

Intuition

Here's how the solution is constructed: 1. Count the frequency of each character in the string s and store it in counter. This allows us to know how many times we need to pick each character during the reordering process.

2. Initialize an empty list ans to build up the result string. 3. Use a while loop that continues until the length of ans matches the length of the input string s.

5. Then iterate over the counter from end to start to append the largest character (if available) to ans and decrease its count. 6. The process repeats, alternating between picking the smallest and largest character until all characters are used.

Character Frequency Counting: We start by creating a list of zeroes called counter to maintain a count of each letter in the

string. The length of the list is 26, one for each letter of the English alphabet. We then iterate over each character of the

string s, and for each character, we find its corresponding index (0 for 'a', 1 for 'b', etc.) by subtracting the ASCII value of 'a'

4. Inside the loop, iterate over the counter from start to end to append the smallest character (if available) to ans and decrease its count.

- 7. Convert the list ans to a string and return it as the final reordered string.
- **Solution Approach**
- The solution implementation utilizes a straightforward approach that involves counting, iterating, and string building. Here's a detailed walk-through of the implementation using Python:

## from the ASCII value of the character. We increment the count at this index in the counter list. counter = [0] \* 26

for i in range(26):

if counter[i]:

for i in range(25, -1, -1):

counter[i] -= 1

if counter[i]:

**Example Walkthrough** 

for c in s: counter[ord(c) - ord('a')] += 1

to the described algorithm. The loop will continue until the length of ans becomes equal to the length of the original string s, signaling that all characters have been chosen and appended. while len(ans) < len(s):</pre>

Picking the Smallest Character: Inside the loop, we iterate over the counter from the start (0) to the end (25) which

append the corresponding character to ans and decrement its count in counter.

Result String Assembly: We define a list ans to accumulate the characters in the order we choose them based on the

Main Loop - Building the Result: We use a while loop to repeat the process of picking characters from the string s according

corresponds to characters 'a' to 'z'. If the current character's counter is not zero, indicating that it is available to be picked, we

algorithm rules—the resulting string after the reordering will be formed by concatenating the characters in this list.

ans.append(chr(i + ord('a'))) counter[i] -= 1 Picking the Largest Character: We do the same for picking the largest character, but in reverse order, iterating over the counter from the end (25) to the start (0). Again, if the current character's counter is not zero, we append the corresponding

Returning the Result: Finally, after the while loop concludes, we join the list of characters in ans using ''.join(ans) to return

the final string, which is the original string s reordered according to the algorithm described. The above steps translate the problem's algorithm into Python code in a way that is both efficient (since the actions within the loop are simple and fast) and clean, leading to a solution that straightforwardly follows the rules laid out in the problem statement.

the counter becomes [1, 2, 1].

'c', 'a', 'b'] and counter is [0, 1, 0].

def sort\_string(self, s: str) -> str:

 $char_count = [0] * 26$ 

for char in s:

sorted\_chars = []

to ['a', 'c', 'a', 'b', 'b'] and counter to [0, 0, 0].

string. Therefore, the final reordered string is "acabb".

updates to [1, 2, 0].

character to ans and decrement its count.

ans.append(chr(i + ord('a')))

position represents 'a', 2 at the second position is for 'b', and 1 at the third position is for 'c'. Result String Assembly: We initialize an empty list ans to store the characters as we pick them following the algorithm.

Picking the Smallest Character: On the first iteration, we look for the smallest character, which is 'a'. We add 'a' to ans, and

Picking the Largest Character: We now pick the largest character, which is 'c'. After appending 'c' to ans, the counter list

Picking the Largest Character: As per our steps, we continue to pick the largest character left, which is still 'b', updating ans

For the string "bacab", the frequency counter counter would look like this after counting: [2, 2, 1], where 2 at the first

Let's walk through a small example to illustrate the solution approach. Suppose our input string s is "bacab".

Main Loop - Building the Result: We start the while loop since len(ans) < len(s), which is 5 in this case.

Character Frequency Counting: We count the frequency of every character in the string.

- Picking the Smallest Character: We pick 'a' again as it is the next available smallest character. The ans list becomes ['a', 'c', 'a'] and counter is [0, 2, 0]. Picking the Largest Character: We need to pick the largest character now, which is 'b'. After doing so, the ans list is ['a',
- **Returning the Result**: The while loop exits since len(ans) is now equal to len(s). We join the elements of ans to form the final
- the expected outcome. The method of counting characters, appending the smallest and largest in order, and decrementing their count ensures that the rules of the problem statement are adhered to at every step of the process.

By following the steps laid out in the solution approach, we successfully applied the algorithm to the example input and achieved

Solution Implementation **Python** 

# Initialize a list to keep track of the count of each character in the string

char\_count[i] -= 1 # Decrement the count of the added character

char\_count[i] -= 1 # Decrement the count of the added character

# Traverse the `char\_count` list from end to start and add each character once if it's present

# Count the occurrences of each character in the string

sorted\_chars.append(chr(i + ord('a')))

# Join the list of characters into a string and return it

char\_count[ord(char) - ord('a')] += 1

# Initialize a list to build the sorted string

for i in range(25, -1, -1):

if char\_count[i] > 0:

// Method to sort the string in a custom order

// Method to sort the string in a specific pattern

// Create a frequency counter for each letter in the alphabet

for (int i = 25; i >= 0; --i) { // Start from 'z'

++frequency[c - 'a']; // Increment the count of the current letter

// Keep building the result until its size matches the original string size

if (frequency[i] > 0) { // Check if the character is present

if (frequency[i] > 0) { // Check if the character is present

result += (i + 'a'); // Convert index to char and append

result += (i + 'a'); // Convert index to char and append

// Append characters from 'a' to 'z' to the result string if they are present

// Append characters from 'z' to 'a' to the result string if they are present

\* Sort the string based on the custom order: ascending characters followed by descending characters

--frequency[i]; // Decrement the frequency of the used character

--frequency[i]; // Decrement the frequency of the used character

string sortString(string s) {

for (char c : s) {

string result = "";

vector<int> frequency(26, 0);

// Initialize the answer string

while (result.size() < s.size()) {</pre>

// Return the sorted string

\* @return {string} - The sorted string.

function sortString(str: string): string {

let resultString: string = '';

\* @param {string} str - The original string to be sorted.

// Count the occurrences of each character in the string

const charMap: Map<string, number> = new Map();

return result;

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

# Continue until the sorted string's length equals the input string's length while len(sorted\_chars) < len(s):</pre> # Traverse the `char\_count` list from start to end and add each character once if it's present for i in range(26): if char\_count[i] > 0: sorted\_chars.append(chr(i + ord('a')))

```
return ''.join(sorted_chars)
Java
```

class Solution {

class Solution:

```
public String sortString(String s) {
   // Counter array to hold frequency of each character 'a'-'z'
   int[] frequency = new int[26];
   // Fill the frequency array with count of each character
    for (char ch : s.toCharArray()) {
        frequency[ch - 'a']++;
   // StringBuilder to hold the result
   StringBuilder sortedString = new StringBuilder();
   // Loop until the sortedString's length is less than the original string length
   while (sortedString.length() < s.length()) {</pre>
       // Loop from 'a' to 'z'
        for (int i = 0; i < 26; ++i) {
            // Check if the character is present
            if (frequency[i] > 0) {
                // Append the character to sortedString
                sortedString.append((char) ('a' + i));
                // Decrement the frequency of appended character
                frequency[i]--;
        // Loop from 'z' to 'a'
        for (int i = 25; i >= 0; --i) {
            // Check if the character is present
            if (frequency[i] > 0) {
                // Append the character to sortedString
                sortedString.append((char) ('a' + i));
                // Decrement the frequency of appended character
                frequency[i]--;
   // Return the resultant sorted string
   return sortedString.toString();
```

```
};
TypeScript
```

**/**\*\*

\*/

C++

public:

class Solution {

```
for (let char of str) {
    charMap.set(char, (charMap.get(char) || 0) + 1);
const keys: string[] = Array.from(charMap.keys());
keys.sort(); // Sort the keys (characters) in ascending order once
// Keep constructing the string until the resultString's length equals the input string's length
while (resultString.length < str.length) {</pre>
   // Append characters in ascending order to the result string
    for (let key of keys) {
        if (charMap.get(key)! > 0) { // Ensure the character count is not zero
            resultString += key;
            charMap.set(key, charMap.get(key)! - 1); // Decrement the count in the map
```

charMap.set(keys[i], charMap.get(keys[i])! - 1); // Decrement the count in the map

// Append characters in descending order to the result string

for (let  $i = keys.length - 1; i >= 0; i--) {$ 

# Count the occurrences of each character in the string

char\_count[ord(char) - ord('a')] += 1

# Initialize a list to build the sorted string

for i in range(25, -1, -1):

return ''.join(sorted\_chars)

Time and Space Complexity

**Time Complexity** 

(from 'a' to 'z').

time complexity.

if char\_count[i] > 0:

if (charMap.get(keys[i])! > 0) {

resultString += keys[i];

return resultString;

def sort\_string(self, s: str) -> str:

char count = [0] \* 26

for char in s:

sorted\_chars = []

class Solution:

```
# Continue until the sorted string's length equals the input string's length
while len(sorted_chars) < len(s):</pre>
    # Traverse the `char_count` list from start to end and add each character once if it's present
    for i in range(26):
        if char_count[i] > 0:
            sorted_chars.append(chr(i + ord('a')))
            char_count[i] -= 1 # Decrement the count of the added character
    # Traverse the `char_count` list from end to start and add each character once if it's present
```

char\_count[i] -= 1 # Decrement the count of the added character

sorted\_chars.append(chr(i + ord('a')))

# Join the list of characters into a string and return it

# Initialize a list to keep track of the count of each character in the string

After that, it enters a loop that continues until the length of ans matches that of s. Within this loop, there are two for-loops: the first iterates in ascending order, the second in descending order. Each of these for-loops iterates over the 26 possible characters

The provided Python function sortString starts with an initial counting pass over the input string s, incrementing values in

- For each character, if that character count is non-zero, it is appended to ans and the count decremented. Since each character in s is processed exactly once (each is appended and then decremented), and there are two passes for each character (one in ascending and one in descending order), the total count of operations inside the while-loop is 2n, leading to an additional 0(n)
- **Space Complexity**

Thus, the time complexity of the entire function is O(n).

counter which takes O(n) time, where n is the length of s.

## The space complexity includes:

- 2. The ans list that will eventually grow to be the same size as s to accommodate all characters, which is O(n). Therefore, the total space complexity is O(n), where n is the length of the input string s.

1. The counter array which is always 26 elements long, thus it is a constant space 0(1).