Sherlock and the Valid String Problem

Isaac DeJager

March 15, 2023

Click here to view the HackerRank page

1 Problem Description

Sherlock considers a string to be valid if all characters of the string appear the same number of times. It is also valid if he can remove just 1 character at 1 index in the string, and the remaining characters will occur the same number of times. Given a string s, determine if it is valid. If so, return **YES**, otherwise return **NO**.

Example
$s = abc$
This is a valid string because frequencies are $\{a : 1, b : 1, c : 1\}$

$s = abcc$
This is a valid string because we can remove one c and have 1 of each character in the remaining string.

$s = abccc$
This string is not valid as we can only remove 1 occurrence of c. That leaves character frequencies of $\{a : 1, b : 1, c : 2\}$.

Function Description
Complete the isValid function in the editor below. isValid has the following parameter:

- string s: a string

Returns

- string: either **YES** or **NO**

Input Format
A single string s.

Constraints

- $1 \leq |s| \leq 10^5$
- Each character $s[i] \in ascii[a-z]$

Sample Input 0
aabbcd

Sample Output 0
NO

Explanation 0
Given s ="aabbcd", we would need to remove two characters, both "c" and "d" $\rightarrow$ "aabb" OR "a" and "b" $\rightarrow$ "abcd" to make it valid. We are limited to removing only one character, so s is invalid.

Sample Input 1
aabbccddeefghi

Sample Output 1
NO

Explanation 1
Frequency counts for the letters are as follows:
{'a' : 2, 'b' : 2, 'c' : 2, 'd' : 2, 'e' : 2, 'f' : 1, 'g' : 1, 'h' : 1, 'i' : 1}
There are two ways to make the valid string:

- Remove 4 characters with frequency 1: {f, g, h, i}
- Remove 5 characters with frequency 2: {a, b, c, d, e }

Neither of these is an option.

Sample Input 2
abcdefghgfedecba

Sample Output 2
YES

Explanation 2
All characters occur twice except for e which occurs 3 times. We can delete one instance of e to have a valid string.

2 Solution

```
string isValid(string s) {

    int arr[26] = { 0 };

    for (int i = 0; i < s.length(); i++) {
        arr[s.at(i) - 97]++;
    }

    sort(arr, arr + 26);

    //Find the index of the first nonzero element
    int j = 0;
    for (int i = 0; arr[i] == 0; i++) {
        j++;
    }

    //Scenario 1
    if (j == 25 || arr[j] == arr[25]) {
        return "YES";
    }

    //Scenario 2
    if (arr[j] == 1 && arr[j + 1] == arr[25]) {
        return "YES";
    }

    //Scenario 3
    if (arr[25] - arr[j] == 1 && arr[24] == arr[j]) {
        return "YES";
    }

    return "NO";

}
```

3 Analysis

The program begins by creating and integer array of size 26. This array represents the number of each occurring character in the string s, where index 0 counts the number of "a"s, index 1 counts the number of "bs", and so on. The array is initialized so all values are 0. Then, the string is looped through each character, and for each character, the corresponding index is incremented. This loop occurs in $O(n)$ time where n is the length of the string.

Next, the program uses a built in C++ sorting function to sort the new array in ascending order. This function performs in $O(n * log(n))$ time, so this step occurs in $O(26 * log(26))$ time since the size of the array is 26.

The next step is to find the first non-zero element in the sorted array. This is simply done by looping through the array until a non-zero element is found. The integer variable j represents the index of the first non-zero element. The is done in $O(26)$ time.

Finally, its time to evaluate this array to determine if the passed string s is valid. There 3 scenarios where s may be valid.

Scenario 1

Every character occurs the same number of times. So the array reads:

$$[0, 0, 0, ..., X, X, X]$$

Where X is the number of times each character appears. If the string s matches this description, then the first non-zero element in the array will equal the last element of the array. And if that is the case, then all values between these two indices will also match because the array is sorted. First though, the program considers the scenario where the string s consists of only 1 character, so the array reads all 0s except for the last element. If that is the case, then the string is valid. So, if the index of the first non-zero element is 25, OR if the first non-zero element equals the last element then the string is valid because it passes scenario 1, and the function returns "YES".

Scenario 2

Every character occurs the same number of times, except for one character which only occurs once. So the array reads:

$$[0, 0, 0, ..., 1, X, X, X]$$

Since scenario 1 is checked first, we know that the index of the first non-zero element is not 25. This means there are at least two distinct characters in s. So, to check scenario 2, first the program sees if the first non-zero element in the array is 1. If it is, then, in order to pass scenario 2, all other elements after must be the same value. And since the array is sorted, that must mean that the elements at indexes $j + 1$ and 25 are the same, where j is the index of the first non-zero element. Essentially, if scenario 1 doesn't pass, then we check if the first non-zero element is 1. If it is, then we recheck scenario 1 after that element. And since scenario 1 didn't pass, $j < 25$, so accessing the $j + 1$ index

of the array is sure not to break array boundaries. And if s happens to read something like "abbbbbbbb", so the corresponding array reads $[0, 0, 0, ..., 1, 8]$, this algorithm still works. Because $j = 24$, and so you will be checking the $j+1$ element against the 25 element, which just means you are checking if the last element equals itself.

Scenario 3

Every character occurs the same number of times, except for one charater which occurs one more time. So the array reads:

$$[0, 0, 0, ..., X, X, X + 1]$$

If the string s produces this sorted array, then the element at index j must be one less than the last element. Additionally, the last element has to be unique in the array because it has to be the largest (and by only 1, but that was checked). So the program checks to see if the array at index j equals the array at index 24, which is the second to last element in the array. If it does, then all values between those indices are also the same because the array is sorted. And based on the first check, the last value has to be one greater. And in the scenario where s reads something like "aabbb" so that the corresponding array reads: $[0, 0, 0, ..., 2, 3]$, this algorithm still works. Because again, $j = 24$. The first checker clearly passes because 3 is one greater than 2. And the second checker passes because you check to see if the value at index j equals the value at index 24. But in this scenario, $j = 24$, so you check to see if the value at index j equals itself.

Each of these scenarios are checked in $O(1)$ time. Thus, the total time for the algorithm is:

$$O(n) + O(26 * log(26)) + O(26) + 3 * O(1) = O(n) + O(1)$$
$$O(n) + O(1) = O(n)$$

The only space required for this algorithm is one integer and an integer array of size 26.