Longest Substring with Same Letters after Replacement (hard)

Problem Statement

Given a string with lowercase letters only, if you are allowed to **replace no more than 'k' letters** with any letter, find the **length of the longest substring having the same letters** after replacement.

Example 1:

```
Input: String="aabccbb", k=2
Output: 5
Explanation: Replace the two 'c' with 'b' to have a longest repeating substring "bbbbb".
```

Example 2:

```
Input: String="abbcb", k=1
Output: 4
Explanation: Replace the 'c' with 'b' to have a longest repeating substring "bbbb".
```

Example 3:

```
Input: String="abccde", k=1
Output: 3
Explanation: Replace the 'b' or 'd' with 'c' to have the longest repeating substring "ccc".
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
const length_of_longest_substring = function(str, k) {
  // TODO: Write your code here
  return -1;
};

```

TEST SAVE RESET

Solution

This problem follows the **Sliding Window** pattern and we can use a similar dynamic sliding window strategy as discussed in No-repeat Substring. We can use a HashMap to count the frequency of each letter.

We'll iterate through the string to add one letter at a time in the window. We'll also keep track of the count of the maximum repeating letter in any window (let's call it `maxRepeatLetterCount`). So at any time, we know that we can have a window which has one letter repeating `maxRepeatLetterCount` times, this means we should try to replace the remaining letters. If we have more than 'k' remaining letters, we should shrink the window as we are not allowed to replace more than 'k' letters.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
function length_of_longest_substring(str, k) {
  let windowStart = 0,
    maxLength = 0,
    maxRepeatLetterCount = 0,
    frequencyMap = {};

  // Try to extend the range [windowStart, windowEnd]
  for (let windowEnd = 0; windowEnd < str.length; windowEnd++) {
    const rightChar = str[windowEnd];
    if (!(rightChar in frequencyMap)) {
      frequencyMap[rightChar] = 0;
    }
    frequencyMap[rightChar] += 1;
    maxRepeatLetterCount = Math.max(maxRepeatLetterCount, frequencyMap[rightChar]);

    // Current window size is from windowStart to windowEnd, overall we have a letter which is
```

Mark Course as Completed

MW

Explore

Tracks

My Courses

Edpresso

Refer a Friend

Create

```
    // repeating 'maxRepeatLetterCount' times, this means we can have a window which has one letter
    // repeating 'maxRepeatLetterCount' times and the remaining letters we should replace.
    // if the remaining letters are more than 'k', it is the time to shrink the window as we
    // are not allowed to replace more than 'k' letters
    if ((windowEnd - windowStart + 1 - maxRepeatLetterCount) > k) {
      leftChar = str[windowStart];
      frequencyMap[leftChar] -= 1;
      windowStart += 1;
    }

    maxLength = Math.max(maxLength, windowEnd - windowStart + 1);
  }
```

RUN SAVE RESET

Time Complexity

The time complexity of the above algorithm will be $O(N)$ where 'N' is the number of letters in the input string.

Space Complexity

As we are expecting only the lower case letters in the input string, we can conclude that the space complexity will be $O(26)$, to store each letter's frequency in the **HashMap**, which is asymptotically equal to $O(1)$.

✓ MARK AS COMPLETED

← Back
No-repeat Substring (hard)

Next →
Longest Subarray with Ones after Rep...

Report an Issue