Longest Substring with Same Letters after Replacement (hard)

We'll cover the following

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 the longest repeatin
g substring "bbbbb".
```

Example 2:

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

Example 3:

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

Try it yourself#

Try solving this question here:

C++

```
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>

class CharacterReplacement {
 public:
  static int findLength(const string& str, int k) {
    int maxLength = 0;
    // TODO: Write your code here
    return maxLength;
  }
};
```

Solution#

This problem follows the **Sliding Window** pattern, and we can use a similar dynamic sliding window strategy as discussed in Longest Substring with Distinct Characters. We can use a HashMap to count the frequency of each letter.

- We will iterate through the string to add one letter at a time in the window.
- We will 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 do have a window with one letter repeating `maxRepeatLetterCount` times; this means we should try to replace the remaining letters.
    - If the remaining letters are less than or equal to `k`, we can replace them all.
    - If we have more than `k` remaining letters, we should shrink the window as we cannot replace more than `k` letters.

While shrinking the window, we don't need to update `maxRepeatLetterCount`. Since we are only interested in the longest valid substring, our sliding windows do not have to shrink, even if a window may cover an invalid substring. Either we expand the window by appending a character to the right or we shift the entire window to the right by one. We only expand the window when the frequency of the newly added character exceeds the historical maximum frequency (from a previous window that included a valid substring). In other words, we do not need to know the exact maximum count of the current window. The only thing we need to know is whether the maximum count exceeds the historical maximum count, and that can only happen because of the newly added char.

Code#

Here is what our algorithm will look like:

C++

```
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>

class CharacterReplacement {
public:
  static int findLength(const string &str, int k) {
    int windowStart = 0, maxLength = 0, maxRepeatLetterCount = 0;
    unordered_map<char, int> letterFrequencyMap;
    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      letterFrequencyMap[rightChar]++;

      // we don't need to place the maxRepeatLetterCount under the below 'if
      // explanation in the 'Solution' section above.
      maxRepeatLetterCount = max(maxRepeatLetterCount, letterFrequencyMap[ri

      // current window size is from windowStart to windowEnd, overall we ha
      // repeating 'maxRepeatLetterCount' times, this means we can have a wi
      // letter repeating 'maxRepeatLetterCount' times and the remaining let
      // if the remaining letters are more than 'k', it is the time to shrin
      // are not allowed to replace more than 'k' letters
      if (windowEnd - windowStart + 1 - maxRepeatLetterCount > k) {
        char leftChar = str[windowStart];
```

Time Complexity#

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

Space Complexity#

As we expect 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 $O(1)$.

Back

Longest Substring with Distinct Chara...

Next

Longest Subarray with Ones after Rep...

Mark as Completed

Report an Issue