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 repeating substring "bbbbb".
```

Example 2:

```
Input: String="abbcb", k=1
Output: 4
Explanation: Replace the 'c' with 'b' to have the 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++

```java
class CharacterReplacement {
  public static int findLength(String str, int 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 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` (hence, it represents the maximum repeating count of ANY letter for ANY window). Why don't we need to update this count when we shrink the window? Since we have to replace all the remaining letters to get the longest substring having the same letter in any window, we can't get a better answer from any other window even though all occurrences of the letter with frequency `maxRepeatLetterCount` is not in the current window.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```java
import java.util.*;

class CharacterReplacement {
  public static int findLength(String str, int k) {
    int windowStart = 0, maxLength = 0, maxRepeatLetterCount = 0;
    Map<Character, Integer> letterFrequencyMap = new HashMap<>();
    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str.charAt(windowEnd);
      letterFrequencyMap.put(rightChar, letterFrequencyMap.getOrDefault(rightChar, 0) + 1);
      maxRepeatLetterCount = Math.max(maxRepeatLetterCount, letterFrequencyMap.get(rightChar));

      // current window size is from windowStart to windowEnd, overall we have a letter which is
      // 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) {
        char leftChar = str.charAt(windowStart);
        letterFrequencyMap.put(leftChar, letterFrequencyMap.get(leftChar) - 1);
        windowStart++;
      }

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

    return maxLength;
  }
```

Run | Save | Reset

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 $O(1)$.