424. Longest Repeating Character Replacement

<https://leetcode.com/problems/longest-repeating-character-replacement/>

1. **Listen**

**Problem Statement:**

You are given a string s and an integer k.

You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most k times.

Return the length of the longest substring containing the same letter you can get after performing the above operations.

**Input:**

1. string **s**
2. integer **k**

**Goal:**

You can choose any character of the string **s** and change it to any other uppercase English character. You can perform this operation at most **k** times.

**Return:**

return the length of the longest contiguous substring containing the same letter you can get after performing the above operations.

1. **Examples**

**Example 1:**

**Input:** s = "ABAB", k = 2

**Output:** 4

**Explanation:** Replace the two 'A's with two 'B's or vice versa.

**Example 2:**

**Input:** s = "AABABBA", k = 1

**Output:** 4

**Explanation:** Replace the one 'A' in the middle with 'B' and form "AABBBBA".

The substring "BBBB" has the longest repeating letters, which is 4.

**Constraints:**

* 1 <= s.length <= 105
* **s** consists of only uppercase English letters.
* 0 <= k <= s.length
* you can change up to **k** letters in **s**.

**Test Cases:**

* Entire string is same letter (“ABAB” k = 2)
* Merging two separate substrings (“AABCAA” k = 2)

**Edge Cases:**

* k is larger than s.length()

1. **Brute Force / Optimize / Walkthrough**

Solution 1: Time = O(S), Space = O(1)

**Technique**

* We can use the sliding window technique.
* We iterate over the string, and slide window over it to represent a substring. This way, we can find the longest substring containing the same letter.

**Identifying Problem Constraint**

How do we know which two characters in the string to change?

First, let’s establish some variables:

1. **end-start+1** = size of the current window.
2. **maxCount** = the current most common character in the window. In other words, the largest count of a single, unique character in the current window.
3. **k** = the number of characters in the window that are not the most common character in the window.

We can find the value of k in the current window by subtracting the character count of the most common character (maxCount) from the current size of the window.

Therefore, we can derive that

**(end-start+1) - maxCount == k**

Therefore, we can also derive that when

**(end-start+1) - maxCount == 0**, then the window is either filled with only one character, or the window consists of only a single unique character (i.e. “s” or “ttttttt”).

**(end-start+1) - maxCount > 0**, then we have characters in the window that are NOT the character that occurs the most.

For Example: For a window "xxxyz"

size of window = (end-start+1) = 5

maxCount = 3

Therefore, (end-start+1) – maxCount = 2

(maxCount is 3 and there are 2 other characters here that are not the most common character, specifically, "y" and "z" that are not "x" in the window.)

Therefore, we can also derive that when

**if (end-start+1) – maxCount == k**

and

**if k == the exact # of characters that are NOT the character that occurs the most in that window**

then

**if (end-start+1) – maxCount > k** then the # of characters that are NOT the character that occurs the most in that window is greater than the # of characters we can change.

We are allowed to have at most k replacements in the window, so when

**(end-start+1) - maxCount > k**

then there are more characters in the window than we can replace, and we need to **shrink the window**.

And we have therefore, found our problem constraint: (end-start+1) - maxCount > k

**Shrink the Window**

If we have window with "xxxy" and k = 1:

We don’t need to shrink the window (enter the loop) because (end-start+1) - maxCount = 1, which is not > k. maxLength gets updated to 4.

But if we then find a "z" after, like "xxxyz", then we need to shrink the window because now (end-start+1) - maxCount = 2, and 2 > 1. The window becomes "xxyz".

And thus, if we continue to run into characters like “z” that are not the most common character, “x”, then we will continue to expand and shrink the array, maintaining the size.

How do we actually shrink the window?

Once there are more than **k** characters in the window that are not the most common character, we need to **shrink** the window.

We can accomplish this by

1. decrementing the frequency array for the character at the front of the window since we are shrinking the window by 1.
2. increment the start of the window

max

“sssxyzysssss”

What about **maxCount**? Won’t it become invalid if the front character is the most common character?

**maxCount may be invalid at some points, but this doesn't matter, because it was valid earlier in the string, and all that matters is finding the max window that occurred *anywhere* in the string**. Additionally, it will expand ***if and only if*** enough repeating characters appear in the window to make it expand. So whenever it expands, it's a valid expansion.

**Runtime**

The runtime of this algorithm is worst-case O(2S) = O(S), where S is the length of the string s.

The space of this algorithm is O(1)

1. **Implement**

public int characterReplacement(String s, int k) {

if(k > s.length()) return s.length();

}

1. **Test**