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

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.

**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 <= 10^5

s consists of only uppercase English letters.

0 <= k <= s.length

**Attempt 1: 2022-09-10 (30min, no update of 'maxRepeat' when substring shrinks is tricky)**

class Solution {

    // Equal to find longest substring has

    // (length of substring - max frequency char in substring) <= k

    public int characterReplacement(String s, int k) {

        int maxLen = 0;

        // 'i' is left end pointer, 'j' is right end pointer

        int i = 0;

        int len = s.length();

        // s consists of only uppercase English letters

        int[] freq = new int[26];

        int maxRepeat = 0;

        for(int j = 0; j < len; j++) {

            char c = s.charAt(j);

            freq[c - 'A']++;

            maxRepeat = Math.max(maxRepeat, freq[c - 'A']);

            // When (length of substring - max frequency char in substring) > k

            // we start to shrink the left end

            if(j - i + 1 - maxRepeat > k) {

                char c1 = s.charAt(i);

                freq[c1 - 'A']--;

                // Below 'maxRepeat' update is not necessary, because when

                // the sliding window shrinks, the frequency counts array

                // won't get larger. So basically 'maxRepeat' never be updated

                // in this loop. The only potential update for 'maxRepeat' is

                // when substring expand on right end

                //for(int m = 0; m < 26; m++) {

                //    maxRepeat = Math.max(maxRepeat, freq[c1 - 'A']);

                //}

                i++;

            }

            maxLen = Math.max(maxLen, j - i + 1);

        }

        return maxLen;

    }

}

Space Complexity: O(n)

Time Complexity: O(n)

**Refer to chatGPT**

**Approach:**

This is a **sliding window** problem where we maintain a window and check its validity by ensuring the condition:

**Window Length − Max Frequency of Any Character in Window ≤ k**

If the condition holds, the window is valid; otherwise, shrink the window.

**Key Idea:**

Sliding Window:

Use two pointers to define the window: start and end.

Expand the window by moving end.

Shrink the window from start if the window becomes invalid.

Character Frequency:

Keep track of the frequency of each character in the window.

class Solution {

public int characterReplacement(String s, int k) {

int[] freq = new int[26]; // Frequency array for characters A-Z

int maxFreq = 0; // Maximum frequency of any single character in the window

int start = 0; // Start of the sliding window

int maxLength = 0; // Maximum length of valid substring

for (int end = 0; end < s.length(); end++) {

// Update frequency of the current character

freq[s.charAt(end) - 'A']++;

// Update the max frequency in the current window

maxFreq = Math.max(maxFreq, freq[s.charAt(end) - 'A']);

// Check if the window is valid

int windowLength = end - start + 1;

if (windowLength - maxFreq > k) {

// Shrink the window

freq[s.charAt(start) - 'A']--;

start++;

}

// Update the maximum length of the valid window

maxLength = Math.max(maxLength, end - start + 1);

}

return maxLength;

}

}

**Explanation:**

Window Validity:

If Window Length − Max Frequency ≤ k, the window is valid because the rest of the characters can be replaced to form a repeating substring.

Sliding Window:

Expand the window by moving end.

If the window becomes invalid, move start to shrink the window until it's valid again.

Frequency Tracking:

Use an array freq of size 26 to store the frequency of each character in the current window.

Max Frequency:

Track the maximum frequency of a single character in the window to avoid recalculating it every time.

**Refer to**

<https://leetcode.com/problems/longest-repeating-character-replacement/solutions/91271/java-12-lines-o-n-sliding-window-solution-with-explanation/>

In case anyone is confused by this solution, here's another way of explaining it:

**end - start + 1** = size of the current window

**maxCount** = largest count of a single, unique character in the current window

The main equation is: **end - start + 1 - maxCount**

When **end - start + 1 - maxCount == 0**, then the window is filled with only one character

When **end - start + 1 - maxCount > 0**, then we have characters in the window that are NOT the character that occurs the most. **end - start + 1 - maxCount** is equal to exactly the # of characters that are NOT the character that occurs the most in that window. Example: For a window "xxxyz", **end - start + 1 - maxCount** would equal 2. (maxCount is 3 and there are 2 characters here, "y" and "z" that are not "x" in the window.)

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.

If we have window with "xxxy" and k = 1, that's fine 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".

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.

public int characterReplacement(String s, int k) {

int len = s.length();

int[] count = new int[26];

int start = 0, maxCount = 0, maxLength = 0;

for (int end = 0; end < len; end++) {

maxCount = Math.max(maxCount, ++count[s.charAt(end) - 'A']);

while (end - start + 1 - maxCount > k) {

count[s.charAt(start) - 'A']--;

start++;

}

maxLength = Math.max(maxLength, end - start + 1);

}

return maxLength;

}

**Same code but another explain**

**Refer to**

<https://leetcode.com/problems/longest-repeating-character-replacement/solutions/91271/java-12-lines-o-n-sliding-window-solution-with-explanation/comments/95833>

This solution is great, best so far. However, it requires a bit more explanation.

Since we are only interested in the longest valid substring, our sliding windows need NOT shrink, even if a window may cover an invalid substring.

We either grow the window by appending one char on the right, or shift the whole window to the right by one.

And we only grow the window when the count of the new char exceeds the historical max count (from a previous window that covers a valid substring).

That is, we do not need the accurate max count of the current window; we only care if the max count exceeds the historical max count; and that can only happen because of the new char.

Here's my implementation that's a bit shorter

class Solution {

    public int characterReplacement(String s, int k) {

        int len = s.length();

        int[] count = new int[26];

        int maxCount = 0;

        int start = 0;

        for(int end = 0; end < len; end++) {

            maxCount = Math.max(maxCount, ++count[s.charAt(end) - 'A']);

            if(end - start + 1 - maxCount > k) {

                count[s.charAt(start) - 'A']--;

                start++;

            }

        }

        return len - start;

    }

}

**Refer to**

[L340.P2.3.Longest Substring with At Most K Distinct Characters (Ref.L424)](note://4E7F61484E324A0D83B155FFCC1A1D0B)