longest repeating character replacement

Problem Statement: Given an integer k and a string s, any character in the string can be selected and changed to any other uppercase English character. This operation can be performed up to k times. After completing these steps, return the length of the longest substring that contains the same letter.

Examples

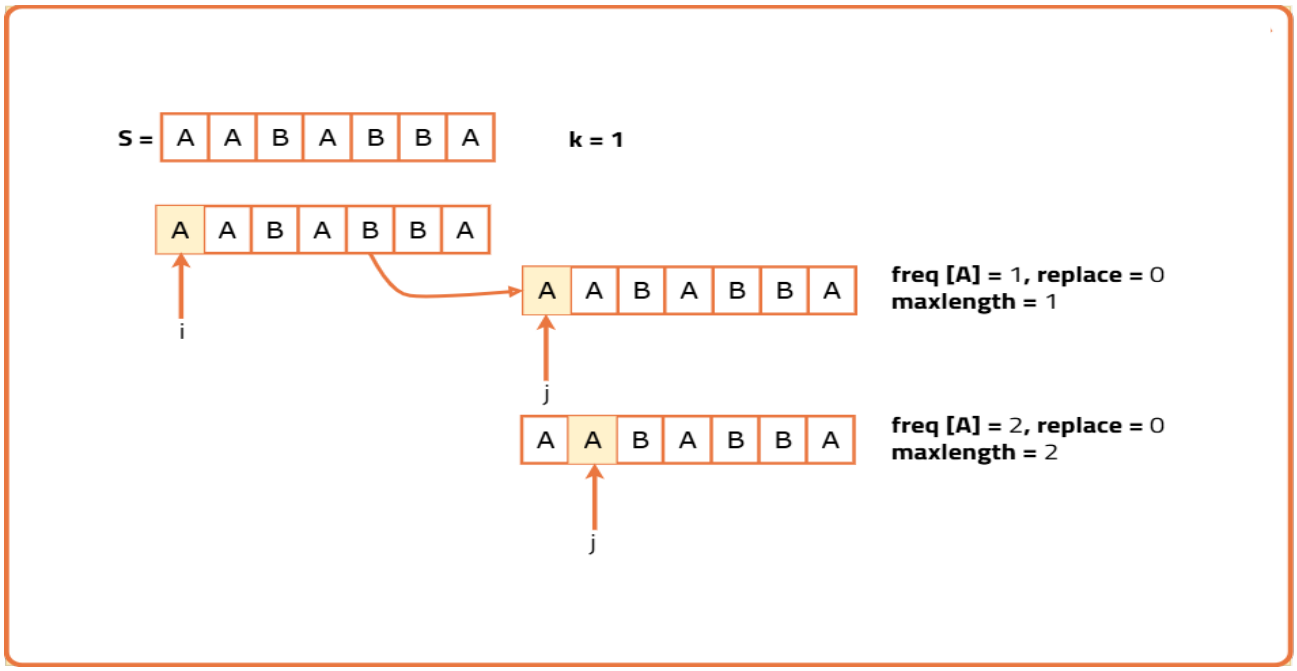
```
Input: s = "BAABAABBBAAA", k = 2
Output: 6
Explanation: We can change the B at index 0 and 3 (0-based indexing) to A. The
new string becomes "AAAAAABBBAAA". The substring "AAAAAA" is the longest
substring with the same letter, and its length is 6.

Input: s = "AABABBA", k = 1
Output: 4
Explanation: We can change one character to get the new string "AABBBBA". The
substring "BBBB" is the longest with the same character. There are other ways to
achieve this result as well.
```

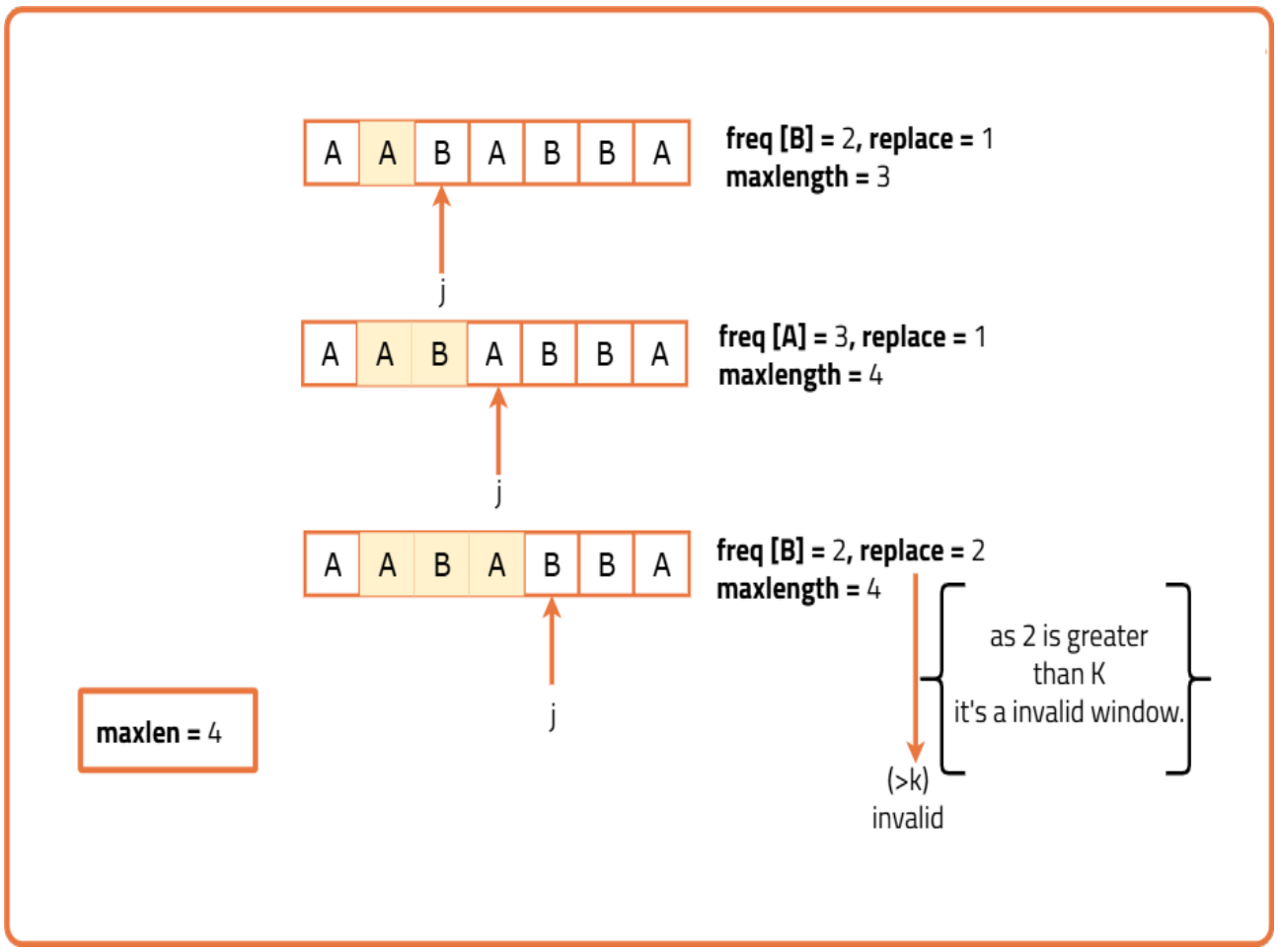
Brute force Approach

Algorithm

We are given a string and an integer k, and we can change at most k characters to make the string contain all the same characters in any chosen window. So the brute-force way would be to check every possible substring, and for each one, calculate how many characters we need to replace to make all characters in that substring the same. If that number is ≤ k, we update our answer.

- Loop over each possible starting position of the substring.
- From each starting point, expand the end position one by one to form all substrings.
- Track the frequency of characters in the current substring.
- Find the most frequent character in this substring.
- Calculate how many characters need to be changed to make all characters the same.
- If the number of required changes is within the allowed limit, update the maximum length.
- Repeat this for all starting positions.





Code

```
import java.util.*;

class Solution {
    // Function to find the length of longest substring that can be converted
    // into a substring of all same characters by replacing at most k characters
    public int characterReplacement(String s, int k) {

        // Variable to track the maximum valid substring length
        int maxLength = 0;

        // Outer loop to iterate through all starting indices
        for (int i = 0; i < s.length(); i++) {

            // Frequency array to store counts of each uppercase letter
            int[] freq = new int[26];

            // Variable to track the max frequency character in the current
window
            int maxFreq = 0;

            // Inner loop to check substrings starting at i
            for (int j = i; j < s.length(); j++) {

                // Increase frequency of current character
                freq[s.charAt(j) - 'A']++;

                // Update most frequent character count in window
                maxFreq = Math.max(maxFreq, freq[s.charAt(j) - 'A']);
```

```
                // Current window size
                int windowSize = j - i + 1;

                // Calculate replacements needed to make all characters same
                int replacements = windowSize - maxFreq;

                // If replacements are within k, update maxLength
                if (replacements <= k) {
                    maxLength = Math.max(maxLength, windowSize);
                }
            }
        }

        return maxLength;
    }
}

// Driver code
class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();
        String s = "AABABBA";
        int k = 1;
        System.out.println(sol.characterReplacement(s, k));
    }
}
```

Complexity Analysis

Time Complexity: O($n^2 \times 26$), where n is the length of the input string.This is because for every possible substring (which takes O(n^2) time), we compute the frequency of each character (which takes O(26) = O(1) time since there are only 26 uppercase English letters). So total time complexity becomes O($n^2 \times 26$), which simplifies to O(n^2).

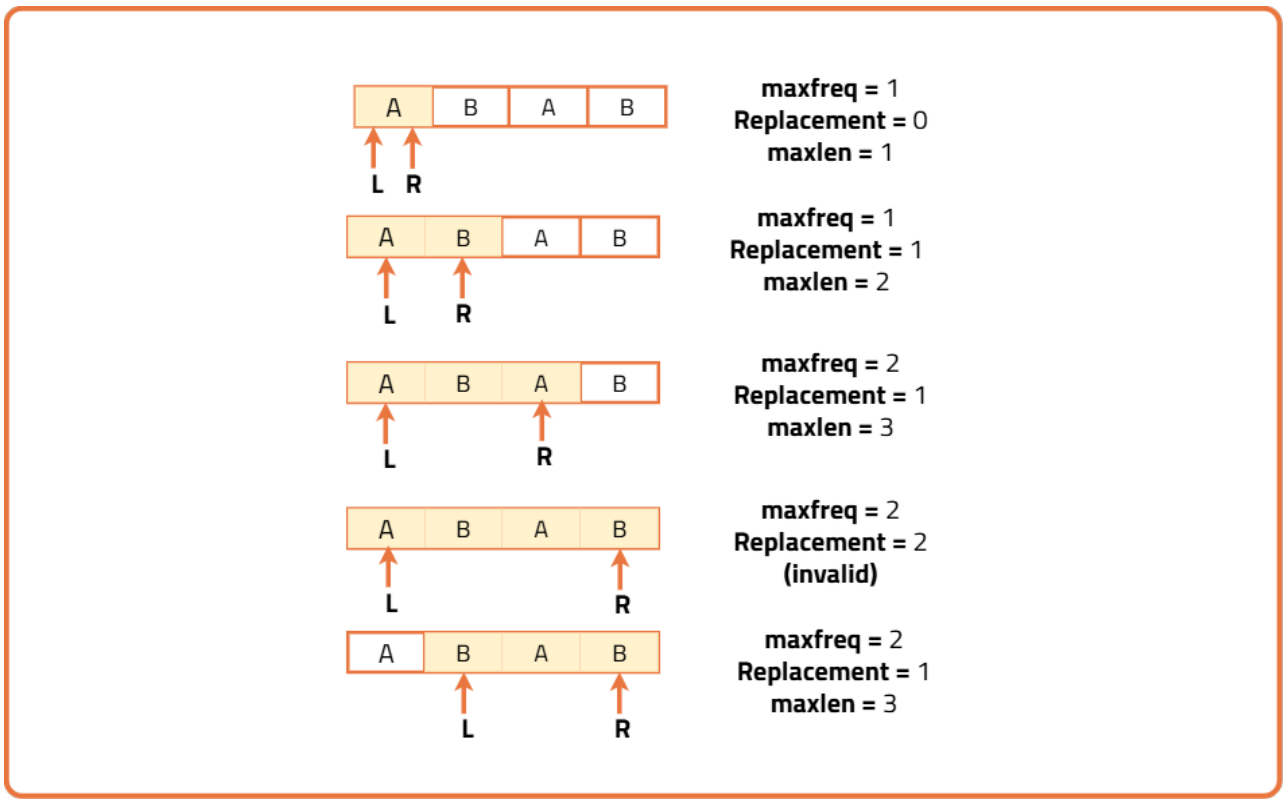
Space Complexity: O(1), constant space.We use a fixed-size array of size 26 to store character frequencies for each substring. No additional space is used that grows with input size.

Better Approach

Algorithm

Instead of checking all substrings, we use a sliding window technique. In a valid window, the number of characters we need to change is window length - count of most frequent character. If this number exceeds k, the window is invalid, and we shrink it from the left. This helps us maintain a valid window of maximum size.

- Initialize a sliding window using two pointers.
- Maintain a hashmap to store frequency of each character in the window.
- Track the count of the most frequent character in the current window.
- If the number of characters to change (window size - max frequency) exceeds allowed limit, move the left pointer forward to shrink the window.
- At each step, update the maximum window size found so far that satisfies the condition.



Code

```
import java.util.*;

class Solution {
    // Function to return the length of the longest substring that can be made
of repeating characters
    // by replacing at most k characters
    public int characterReplacement(String s, int k) {

        // Array to count frequency of characters in window
        int[] freq = new int[26];

        // Left pointer of sliding window
        int left = 0;

        // Tracks the highest frequency in the window
        int maxFreq = 0;

        // Stores result
        int maxLen = 0;

        // Traverse the string with right pointer
        for (int right = 0; right < s.length(); right++) {

            // Increment count of current character
            freq[s.charAt(right) - 'A']++;

            // Update max frequency in current window
            maxFreq = Math.max(maxFreq, freq[s.charAt(right) - 'A']);

            // If number of changes exceeds k, shrink window
            while ((right - left + 1) - maxFreq > k) {
                freq[s.charAt(left) - 'A']--;
                left++;
            }
```

```
            // Update result with valid window length
            maxLen = Math.max(maxLen, right - left + 1);
        }

        return maxLen;
    }
}

// Driver code
class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();
        String s = "AABABBA";
        int k = 1;
        System.out.println(sol.characterReplacement(s, k));
    }
}
```

Complexity Analysis

Time Complexity: O(N),We iterate through the entire string once using a sliding window. Each character is added and removed from the window at most once, resulting in linear time complexity relative to the length of the string (N).

Space Complexity: O(26) ,We use a fixed-size frequency array or hashmap to store counts of uppercase English letters (which are 26 in total), so the space used remains constant regardless of the input size.

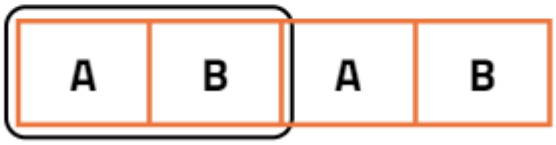
Optimal Approach
Algorithm
The key optimization over the previous sliding window method is to avoid shrinking the window. Instead of always adjusting the window size based on the replacement condition, we keep expanding the window as long as the current window satisfies the condition. We don't need to shrink the window manually because the maximum window size that satisfies the constraint can be tracked directly, as we are only asked for the maximum length, not the actual substring.

hence, as we slide the window, we always try to increase the window size. We maintain the count of the most frequent character in the current window. If the difference between the window size and this count exceeds k, it means we need more than k replacements but instead of shrinking, we just continue checking and computing the maximum valid length encountered.

This approach avoids recalculating the maximum frequency on every iteration, making it faster than the better approach.

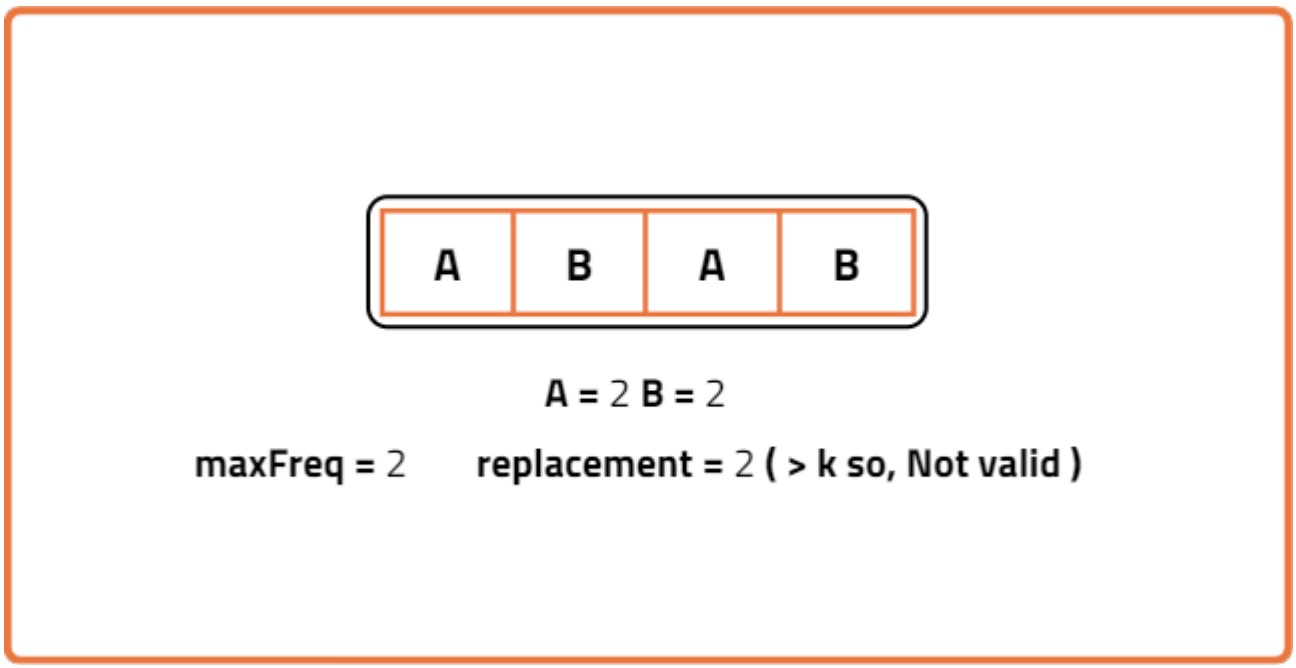
- Initialize a frequency array or hashmap to store character frequencies in the current window.
- Track the count of the most frequent character seen so far in the window.
- For every character, expand the window by moving the right pointer.
- If (current window length - max frequency) exceeds k, it means more than k replacements are needed, so we move the left pointer forward.
- Throughout the loop, update the max window length that satisfies the constraint.

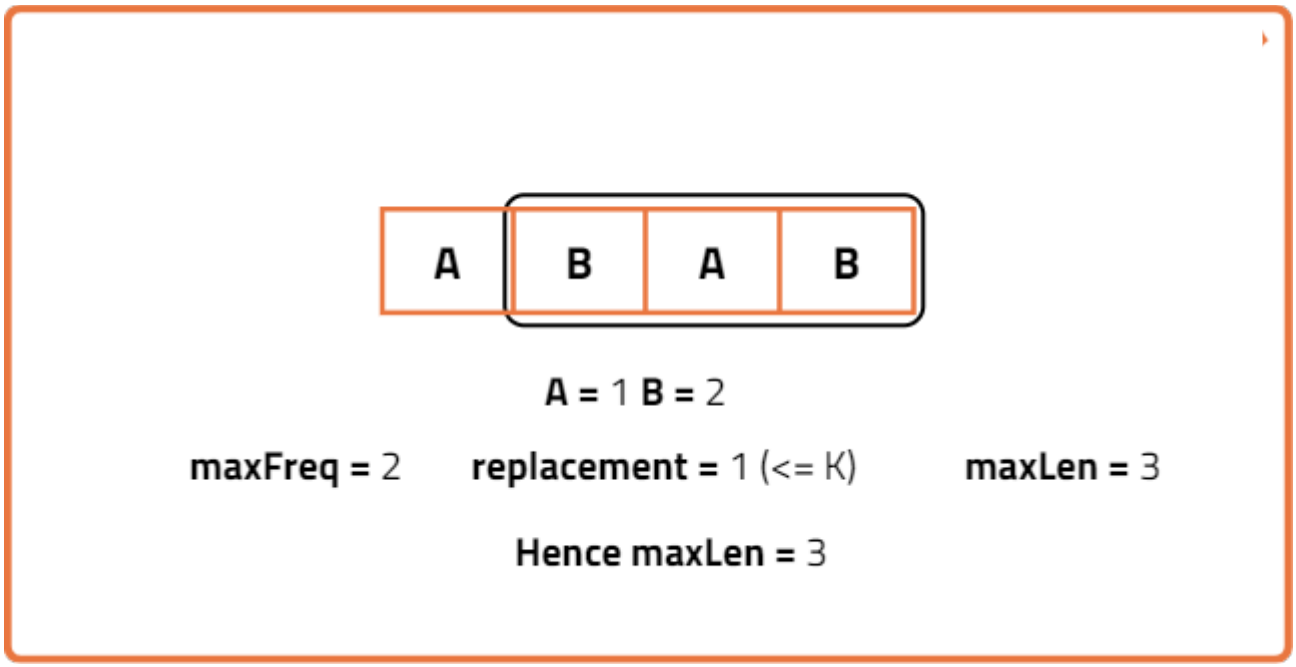
maxFreq = 1 **replacement =** 0 **maxLen =** 1



maxFreq = 1 **replacement =** 1 **maxLen =** 2

maxFreq = 2 **replacement =** 1 **maxLen =** 3





Code

```
import java.util.*;

class Solution {
    // Function to return the length of the longest substring that can be made
of repeating characters
    // by replacing at most k characters
    public int characterReplacement(String s, int k) {
        // Frequency array for A-Z
        int[] freq = new int[26];

        // Left and right pointers of sliding window
        int left = 0, right = 0;

        // Tracks the count of the most frequent character in current window
        int maxCount = 0;

        // Stores the maximum length of valid window
```

```
        int maxLength = 0;

        // Iterate through the string with right pointer
        while (right < s.length()) {

            // Increment the frequency of current character
            freq[s.charAt(right) - 'A']++;

            // Update maxCount with the max frequency seen so far
            maxCount = Math.max(maxCount, freq[s.charAt(right) - 'A']);

            // If the current window needs more than k replacements, move left
            while ((right - left + 1) - maxCount > k) {
                freq[s.charAt(left) - 'A']--;
                left++;
            }

            // Update the maximum window length
            maxLength = Math.max(maxLength, right - left + 1);

            // Move right pointer forward
            right++;
        }

        // Return the maximum valid window length
        return maxLength;
    }
}

// Driver class
class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();
        String s = "AABABBA";
        int k = 1;
        // Output: 4
        System.out.println(sol.characterReplacement(s, k));
    }
}
```

Complexity Analysis

Time Complexity: O(n), where n is the length of the string,each character is processed at most twice once by the right pointer, once by the left. All operations inside the loop run in constant time.

Space Complexity: O(1), constant space .Only a fixed-size frequency array (26 letters) is used, regardless of input size.