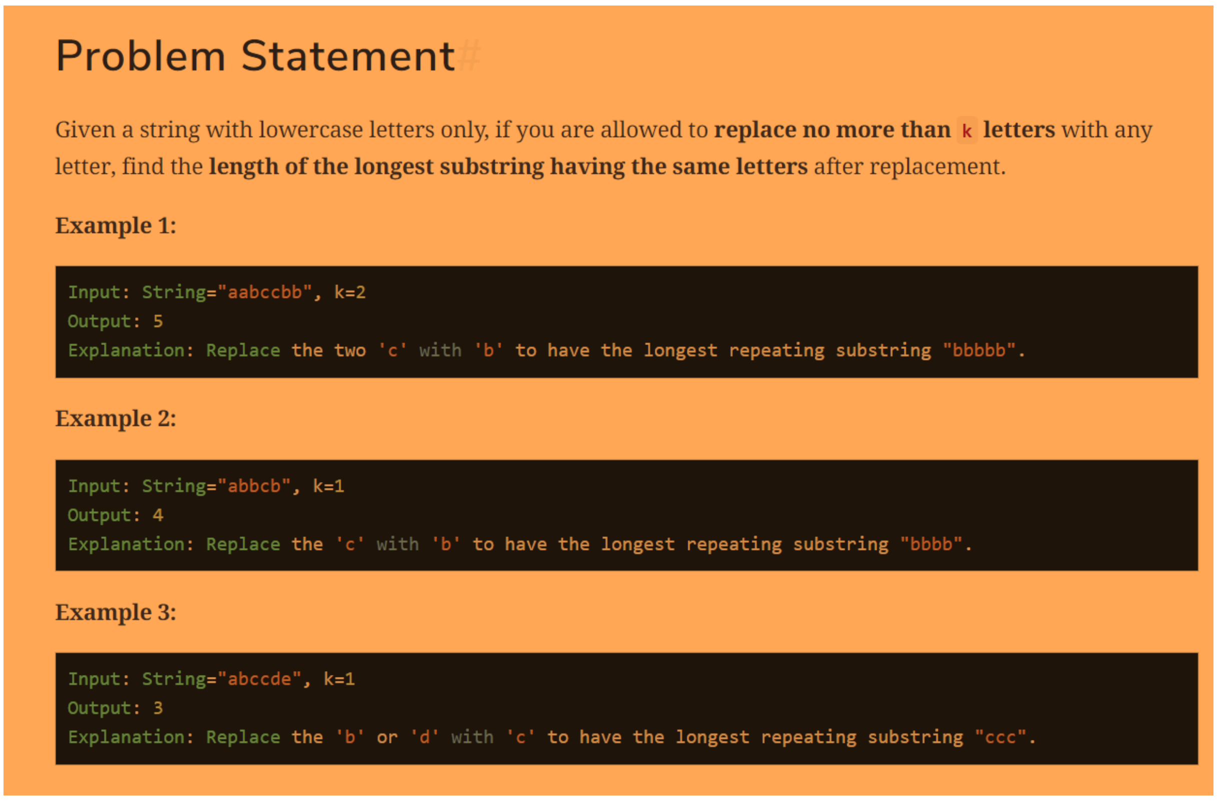
Longest Substring with Same Letters after Replacement

SR Score	1204
Link	https://www.educative.io/courses/grokking-the-coding-interview/R8DVgjq78yR
Last Reviewed	@April 9, 2022
Time	3
Score	2
DS	arrays
Algo	sliding window
Stated	hard
Perceived	hard
List	REPEAT
Needs Review	☑
Repeat Offender	□
Confident	□
C_Date	1
C_Solution	4
C_Time	300
Frequency	

▼ **Problem Statement**



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".
```

▼ **Intuition**

- setup for the sliding window is the same—windowStart/windowEnd pointers, updating max variable after the inner condition, a dict to store letter/character counts, **and a condition that is the crux of the logic in sliding window problems**
- so here's the **condition** of the problem:
    - we have a certain window size found by $(windowEnd - windowStart + 1)$ right?
    - so the condition is always based on finding when the window is valid and as follows, when it's no longer valid
        - in any given window of the string, we obv. want to replace as few characters as possible since we have a limited # of replacements, so we find the character which occurs most often in the window

      - so the # of replacements needed in the window = $(windowEnd - windowStart + 1) - maxOccurringCharacterInWindow$
      - now if the # of replacements we *have,* $k >= replacementsNeeded,$ then we have a “valid” window and can expand the window further
      - if $replacementsNeeded > k$, however, then we need to shrink the window until $replacementsNeeded <= k$
  - ▼ the $O(n)$ optimization
    - so we can frame the goal of this problem like this: we are trying to maximize the $longest$ variable
    - now go back to the all-important **condition**: $((windowEnd - windowStart + 1) - maxOccurringCharacterInWindow) <= k$
      - normally, $maxOccurringCharacterInWindow$ is actually $max(tracker.values())$, which itself is an $O(26)$ operation since we coud theoretically store all 26 letters of the alphabet in the $tracker$ dict
      - so tying together our reframed goal and condition, $longest$ is maximized when it is as large of a # as possible, but as that number grows, $maxOcurringCharacterInWindow$ has to grow as well to keep us within k without going over
      - as such, $longest$ can only be maximized when $maxOcurringCharacterInWindow$ is increasing
        - so when we are going thru the main for loop, if the $maxOcurringCharacterInWindow$ for the ***current window*** we’re in is not higher than what $maxOcurringCharacterInWindow$ was in another previous window, then there is no way we can get a new max $longest$, so we don’t need to check for a new $maxOcurringCharacterInWindow$ everytime we’re executing the **condition**, but only once as we process each letter in the original string itself: $max(maxOcurringCharacterInWindow, tracker[letter])$
        - this operation is $O(1)$ bc we are only accessing a dict key (which is constant on average) whereas scanning thru the entire $tracker$ dict each time is $O(26)$ in the worst case
- ▼ **Time & Space Considerations**
  - Time: O(26n) → O(n) optimal
    - main for loop goes thru all elements in string → O(n)
    - while loop will process each element max of 1 time → O(n)
    - in unoptimized version, $max(tracker.values())$ iterates thru hashmap (max of 26 keys (see Space: O(26) ~ O(1)) → O(26))
      - O(26(n + n)) = O(52n) ~ O(26n)
  - Space: O(26) ~ O(1)
    - problem says only lowercase letters, and there are only 26 of those in the case that every letter in the alphabet has to be stored in in the $tracker$ dict
- ▼ **Review Notes**
  - ▼ [Early March]
    - no clue, had to look at solution
  - ▼ [3/17/22]
    - had solution that had lot of the right parts, the key condition for shrinking the window was wrong
  - ▼ [4/9/22]
    - got a solution that passed GCI test cases, but had bugs bc failed on LC
    - Looked at conceptual explanation of initial solution in Resources, and coded O(26n) version in ~2-3 min
    - Looked at solution for O(n) optimal version in Resources completely to understand O(n) solution
    - regressed from attempt 2
- ▼ **Tracking**

Scores

Aa Attempt #	Date	# Time	# Score
3	@April 9, 2022	4	2
2	@March 17, 2022	2	3

Attempt #	Date	Time	Score
1	@March 1, 2022 → March 16, 2022	1	1

▼ Solutions

```
# attempt 3: 4/9/22
# had to peek, then look at both O(26n), O(n) solutions
def characterReplacement(self, s, k):
    tracker = dict()
    longest = maxRepeatingChar = windowStart = 0
    for windowEnd in range(len(s)):
        letter = s[windowEnd]
        tracker[letter] = tracker.get(letter, 0) + 1
        maxRepeatingChar = max(maxRepeatingChar, tracker[letter])
        while (windowEnd - windowStart + 1) - maxRepeatingChar > k:
            leftLetter = s[windowStart]
            tracker[leftLetter] -= 1
            windowStart += 1

        longest = max(longest, windowEnd - windowStart + 1)

    return longest

def main():
    print(length_of_longest_substring("aabccbb", 2))
    print(length_of_longest_substring("abbcb", 1))
    print(length_of_longest_substring("abccde", 1))

main()
# -------------------------------------------------------------------------------
# attempt 2: 3/17/22
# had a lot of the parts of the sliding window pattern, but couldn't get the main
# / condition that made it tricky
def length_of_longest_substring(str1, k):
    tracker = dict()
    longestSubstr = windowStart = maxAppearingLetter = 0
    for windowEnd in range(len(str1)):
        letter = str1[windowEnd]
        tracker[letter] = tracker.get(letter, 0) + 1
        maxAppearingLetter = max(maxAppearingLetter, tracker[letter])
        while maxAppearingLetter > k:
            leftmostLetter = str1[windowStart]
            tracker[leftmostLetter] -= 1
            if tracker[leftmostLetter] == 0:
                del tracker[leftmostLetter]
            windowStart += 1
        longestSubstr = max(longestSubstr, windowEnd - windowStart + 1)

    return longestSubstr

def main():
    print(length_of_longest_substring("aabccbb", 2))
    print(length_of_longest_substring("abbcb", 1))
    print(length_of_longest_substring("abccde", 1))

main()
# -------------------------------------------------------------------------------
# attempt 1: sometime in March before the 17th
# didn't get a solution, looked at Resources
def length_of_longest_substring(str1, k):
    letterTracker = dict()
    longestSubstr = windowStart = maxFrequency = 0
    for windowEnd in range(len(str1)):
        letter = str1[windowEnd]
        letterTracker[letter] = letterTracker.get(letter, 0) + 1
        maxFrequency = max(maxFrequency, letterTracker[letter])
        while (windowEnd - windowStart + 1) - maxFrequency > k:
            leftLetter = str1[windowStart]
            letterTracker[leftLetter] -= 1
            if letterTracker[leftLetter] == 0:
                del letterTracker[leftLetter]
            windowStart += 1
        longestSubstr = max(longestSubstr, windowEnd - windowStart + 1)

    return longestSubstr

def main():
    print(length_of_longest_substring("aabccbb", 2))
    print(length_of_longest_substring("abbcb", 1))
    print(length_of_longest_substring("abccde", 1))

main()
```

▼ Resources

Longest Repeating Character Replacement - Leetcode 424 - Python

🎓 Get 10% off EducativeIO today ▸ https://www.educative.io/neetcode🟣 Get 10% off AlgoMonster today ▸ https://bit.ly/3nYBVKS (Use code NEET at checkout for ...

https://www.youtube.com/watch?v=gqXU1UyA8pk

▼ GitHub

GCI/Pattern 1 - Sliding Window/Longest Substring with Same Letters after Replacement at main · psdev30/GCI

Contribute to psdev30/GCI development by creating an account on GitHub.

https://github.com/psdev30/GCI/tree/main/Pattern%201%20-%20Sliding%20Window/Longest%20Substring%20with%20Same%20Letters%20after%20Replacement