Back To Course Home

Grokking the Coding Interview: Patterns for Coding Questions

4% completed

Search Course

Pattern: Cyclic Sort

Introduction (/courses/grokking-the-coding-interview/YVjXo6J9xN9)

Cyclic Sort (easy) (/courses/grokking-the-coding-interview/B8qXVqVwDKY)

Find the Missing Number (easy) (/courses/grokking-the-coding-interview/JPnp17NYXE9)

Find all Missing Numbers (easy) (/courses/grokking-the-coding-interview/Y52qNM0IjWK)

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 a longest repeating substring
"bbbbb".
```

Example 2:

```
Input: String="abbcb", k=1
Output: 4
Explanation: Replace the 'c' with 'b' to have a longest repeating substring "bbb
b".
```

Example 3:

```
Input: String="abccde", k=1
Output: 3
Explanation: Replace the 'b' or 'd' with 'c' to have the longest repeating substr
ing "ccc".
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
class CharacterReplacement {
  public static int findLength(String str, int k) {
    // TODO: Write your code here
    int n = str.length();
    HashMap<Character,Integer> map = new HashMap<>(
    int start = 0 , end = 0 , ans =0 , maxCharCount

    while(end < n)
```

```
    {
      char temp = str.charAt(end);
      map.put(temp,map.getOrDefault(temp,0)+1);
      maxCharCount = Math.max(maxCharCount,map.get(
      if(end-start+1 - maxCharCount > k)
      {
        char temp2 = str.charAt(start);
        map.put(temp2,map.get(temp2)-1);
        start++;
      }
      ans = Math.max(ans, end - start +1);
      end++;
    }

    return ans;
  }
}
```

Test

Save *

Reset

Show Results | Show Console

3 of 3 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	findLength(aabccbb, 2)	5	5	Succeeded
✓	findLength(abbcb, 1)	4	4	Succeeded
✓	findLength(abccde, 1)	3	3	Succeeded

2.826s

Solution

This problem follows the **Sliding Window** pattern, and we can use a similar dynamic sliding window strategy as discussed in No-repeat Substring (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5485010335301632/). We can use a HashMap to count the frequency of each letter.

- We'll iterate through the string to add one letter at a time in the window.
- We'll 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 can have a window which has one letter repeating `maxRepeatLetterCount` times; this means we should try to replace the remaining letters.
- If we have more than 'k' remaining letters, we should shrink the window as we are not allowed to replace more than 'k' letters.

While shrinking the window, we don't need to update `maxRepeatLetterCount` (which makes it global count; hence, it is the maximum count for ANY window). Why don't we need to update this count when we shrink the window? The answer: In any window, since we have to replace all the remaining letters to get the longest substring having the same letter, 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

```
import java.util.*;

class CharacterReplacement {
  public static int findLength(String str, int k) {
    int windowStart = 0, maxLength = 0, maxRepeatLe
    Map<Character, Integer> letterFrequencyMap = ne
    // try to extend the range [windowStart, windov
    for (int windowEnd = 0; windowEnd < str.length(
      char rightChar = str.charAt(windowEnd);
      letterFrequencyMap.put(rightChar, letterFrequ
      maxRepeatLetterCount = Math.max(maxRepeatLeti

      // current window size is from windowStart to
      // repeating 'maxRepeatLetterCount' times, th
      // repeating 'maxRepeatLetterCount' times and
      // if the remaining letters are more than 'k'
      // are not allowed to replace more than 'k' l
      if (windowEnd - windowStart + 1 - maxRepeatLe
        char leftChar = str.charAt(windowStart);
        letterFrequencyMap.put(leftChar, letterFrec
        windowStart++;
      }

      maxLength = Math.max(maxLength, windowEnd - w
    }

    return maxLength;
  }

  public static void main(String[] args) {
    System.out.println(CharacterReplacement.findLer
```

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 $O(1)$.