<https://www.lintcode.com/problem/386/>

<https://leetcode.com/problems/longest-substring-with-at-most-k-distinct-characters/>

Given a string *S*, find the length of the longest substring *T* that contains at most k distinct characters.

**Example 1:**

Input: S = "eceba" and k = 3

Output: 4

Explanation: T = "eceb"

**Example 2:**

Input: S = "WORLD" and k = 4

Output: 4

Explanation: T = "WORL" or "ORLD"

**Constraints:**

1 <= s.length <= 5 \* 10^4

0 <= k <= 50

**Follow up:**

O(n) time solution possible ?

**Attempt 1: 2022-09-07**

**Solution 1: Not fixed length Sliding Window + HashMap (30min)**

public class Solution {

    /\*\*

    \* @param s: A string

    \* @param k: An integer

    \* @return: An integer

    \*/

    public int lengthOfLongestSubstringKDistinct(String s, int k) {

        // write your code here

        Map<Character, Integer> freq = new HashMap<Character, Integer>();

        int maxLen = Integer.MIN\_VALUE;

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

        int i = 0;

        int len = s.length();

        // Before over k distinct characters try to expand right end as much as possible

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

            char c = s.charAt(j);

            freq.put(c, freq.getOrDefault(c, 0) + 1);

            int size = freq.size();

            // When distinct character number > k, start shrink left end, use while loop

            // in case current string section (left end indicate by 'i') start with same

            // character, e.g "bbeceba" start with two 'b's when j = 3, k = 2 and i = 0,

            // currently substring indicated as "bbec", we have to remove two 'b' to match

            // k = 2, while loop run two times to increase i from 0 to 2, which means

            // truncate two 'b's and keep "ec", distinct characters drop back from 3 to 2

            while(size > k) {

                char c1 = s.charAt(i);

                freq.put(c1, freq.get(c1) - 1);

                if(freq.get(c1) == 0) {

                    freq.remove(c1);

                    size--;

                }

                i++;

            }

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

        }

        return maxLen == Integer.MIN\_VALUE ? 0 : maxLen;

    }

}

Space Complexity: O(n)

Time Complexity: O(n)

https://leetcode.com/problems/minimum-size-subarray-sum/discuss/59078/Accepted-clean-Java-O(n)-solution-(two-pointers)/924580

For those trying to figure out how is it O(n):

Here we have defined 2 index i & j,

In case of O(n^2) for each outer loop, inner loop runs some n or m number of times to make it O(nm),

that means, as soon as the outer loop finishes one iteration, inner loop resets itself.

In case of O(n2), as in this case, we are not resetting the inner inner variable i, it's just

incrementing each time. It is like 2 loops one after another and both runs n number of time.

**Solution 2: Not fixed length Sliding Window + HashMap (30min)**

public class Solution {

    /\*\*

    \* @param s: A string

    \* @param k: An integer

    \* @return: An integer

    \*/

    public int lengthOfLongestSubstringKDistinct(String s, int k) {

        // write your code here

        Map<Character, Integer> char\_index\_map = new HashMap<Character, Integer>();

        int maxLen = Integer.MIN\_VALUE;

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

        int i = 0;

        int len = s.length();

        // Before over k distinct characters try to expand right end as much as possible

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

            // Instead of store {char, frequency} mapping, we store {char, most recent index},

            // the most recent index of character means the latest happening position in String s

            // e.g 'b' in "bbeceba" when j = 3, the most recent index of 'b' is 1 (update from 0)

            char\_index\_map.put(s.charAt(j), j);

            int size = char\_index\_map.size();

            // When distinct character number > k, start shrink left end, use while loop

            // in case current string section (left end indicate by 'i') start with same

            // character, e.g "bbeceba" start with two 'b's when j = 3, k = 2 and i = 0,

            // currently substring indicated as "bbec", we have to remove two continuous

            // 'b' to match k = 2, the while loop does the truncate job, it runs one time

            // to increase i from 0 to 1, which means skip all continuous 'b' and reach to

            // the most recent one, the 2nd 'b' is the most recent one since its index = 1

            // match the stored most recent index = 1 when j = 3 (if we have continuous 'b'

            // at beginning, it will skip more, e.g "bbbbeceba" it will skip 3 times), now

            // we can remove distinct character 'b' from map directly by just increase i once,

            // since the next new character 'e' will be the new start of window (i++ after

            // size-- indicate to new character 'e'), distinct characters drop back from 3 to 2

            if(size > k) {

                while(char\_index\_map.get(s.charAt(i)) != i) {

                    i++;

                }

                char\_index\_map.remove(s.charAt(i));

                size--;

                i++;

            }

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

        }

        return maxLen == Integer.MIN\_VALUE ? 0 : maxLen;

    }

}

Space Complexity: O(n)

Time Complexity: O(n)

**Refer to chatGPT**

**Approach:**

This is a **sliding window with a hash map** problem. The sliding window keeps track of valid substrings, and a hash map (or frequency counter) maintains the count of characters in the current window.

**Key Idea:**

Sliding Window:

Expand the window by moving the end pointer.

Shrink the window by moving the start pointer when the number of distinct characters exceeds k.

HashMap for Character Frequency:

Use a hash map to store the frequency of characters in the current window.

When the size of the map exceeds k, shrink the window by decrementing the frequency of the character at the start pointer.

class Solution {

public int lengthOfLongestSubstringKDistinct(String s, int k) {

if (k == 0 || s == null || s.isEmpty()) {

return 0;

}

HashMap<Character, Integer> charCount = new HashMap<>();

int start = 0, maxLength = 0;

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

char currentChar = s.charAt(end);

charCount.put(currentChar, charCount.getOrDefault(currentChar, 0) + 1);

// Shrink the window if there are more than k distinct characters

while (charCount.size() > k) {

char startChar = s.charAt(start);

charCount.put(startChar, charCount.get(startChar) - 1);

if (charCount.get(startChar) == 0) {

charCount.remove(startChar);

}

start++;

}

// Update the maximum length

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

}

return maxLength;

}

}

**Explanation:**

Sliding Window:

Expand the window by including characters as end pointer moves.

Shrink the window by moving start if the number of distinct characters exceeds k.

Hash Map:

Tracks the count of each character in the current window.

Remove characters from the map when their count reaches 0 after shrinking the window.

Update Maximum Length:

For every valid window (with at most k distinct characters), update the maximum length.

**Refer to**

[L424.P2.6.Longest Repeating Character Replacement (Ref.L340)](note://1581773836A946879EB5237C85BFF6C2)