340. Longest Substring with at Most K Distinct Characters

Feb. 21, 2019 | 39.9K views

Average Rating: 4.52 (25 votes)

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

Example 1:

```
Input: s = "eceba", k = 2
Output: 3
Explanation: T is "ece" which its length is 3.
```

Example 2:

```
Input: s = "aa", k = 1
Output: 2
Explanation: T is "aa" which its length is 2.
```

Solution

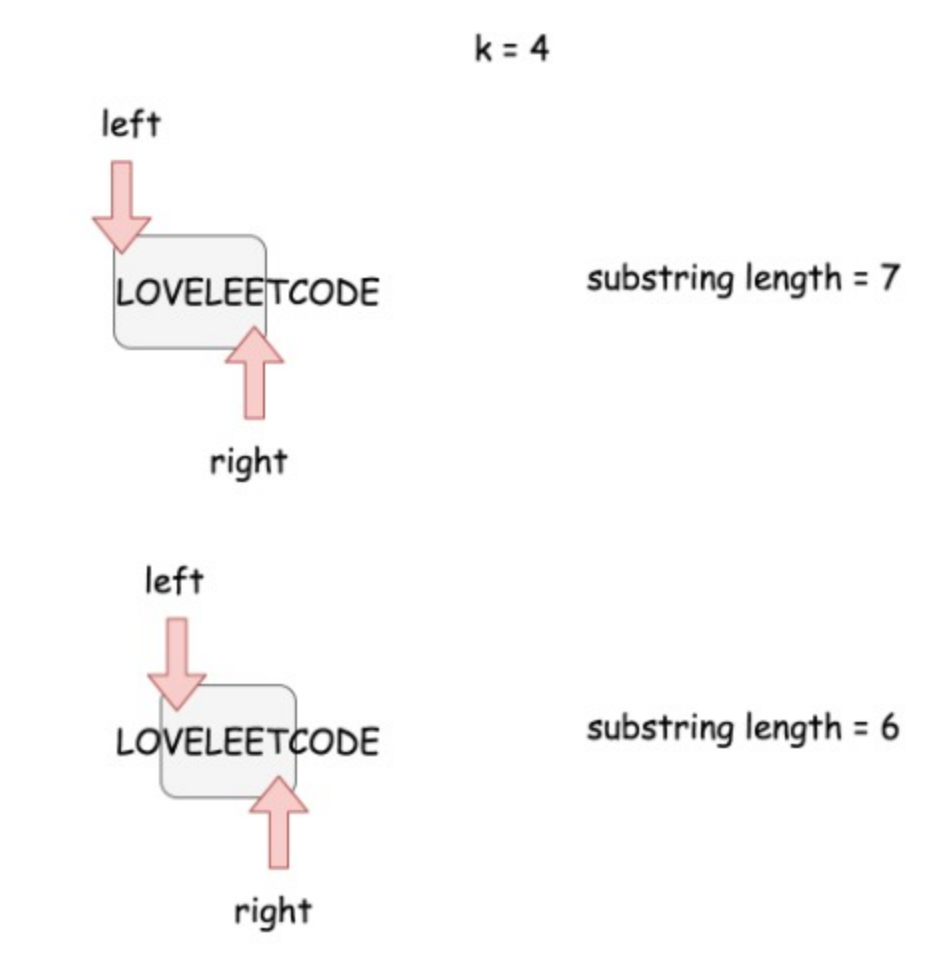
Approach 1: Sliding Window + Hashmap.

Intuition

Let's use here the logic from the more simple problem with at most two distinct characters.

To solve the problem in one pass let's use here *sliding window* approach with two set pointers `left` and `right` serving as the window boundaries.

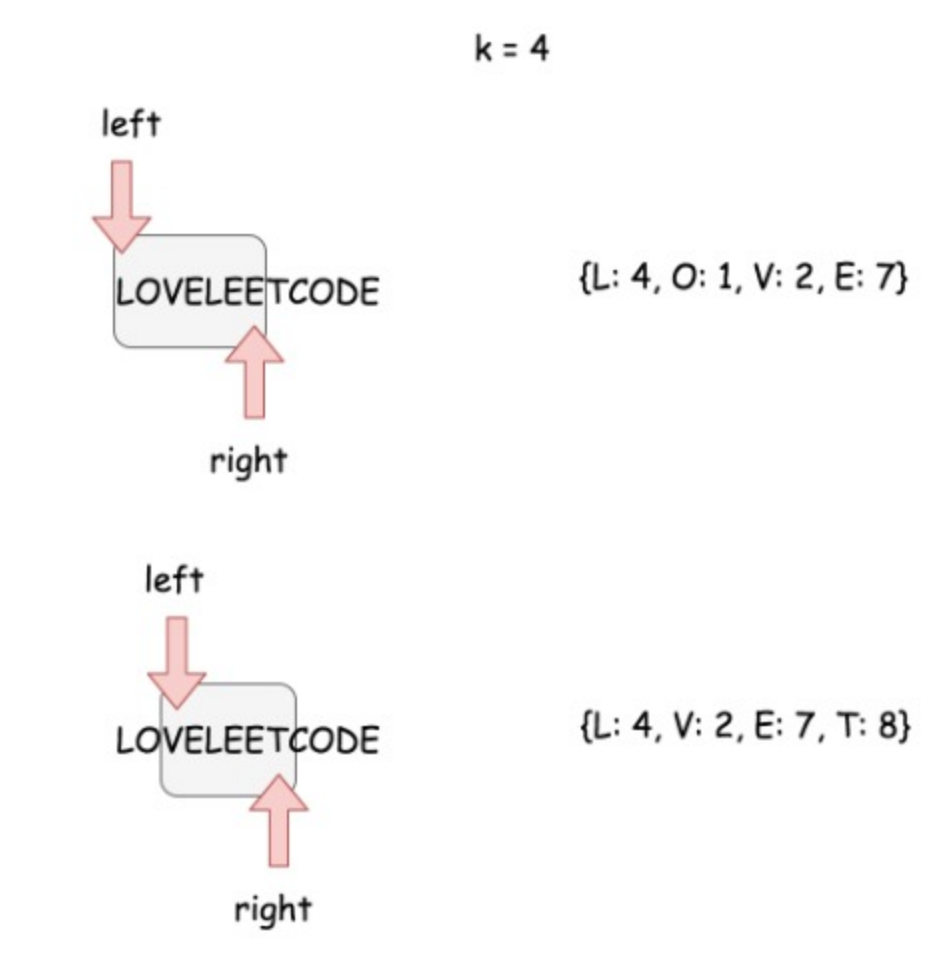
The idea is to set both pointers in the position `0` and then move `right` pointer to the right while the window contains not more than `k` distinct characters. If at some point we've got `k + 1` distinct characters, let's move `left` pointer to keep not more than `k + 1` distinct characters in the window.



Basically that's the algorithm : to move sliding window along the string, to keep not more than `k` distinct characters in the window, and to update max substring length at each step.

> There is just one more question to reply - how to move the left pointer to keep only `k` distinct characters in the string?

Let's use for this purpose hashmap containing all characters in the sliding window as keys and their rightmost positions as values. At each moment, this hashmap could contain not more than `k + 1` elements.



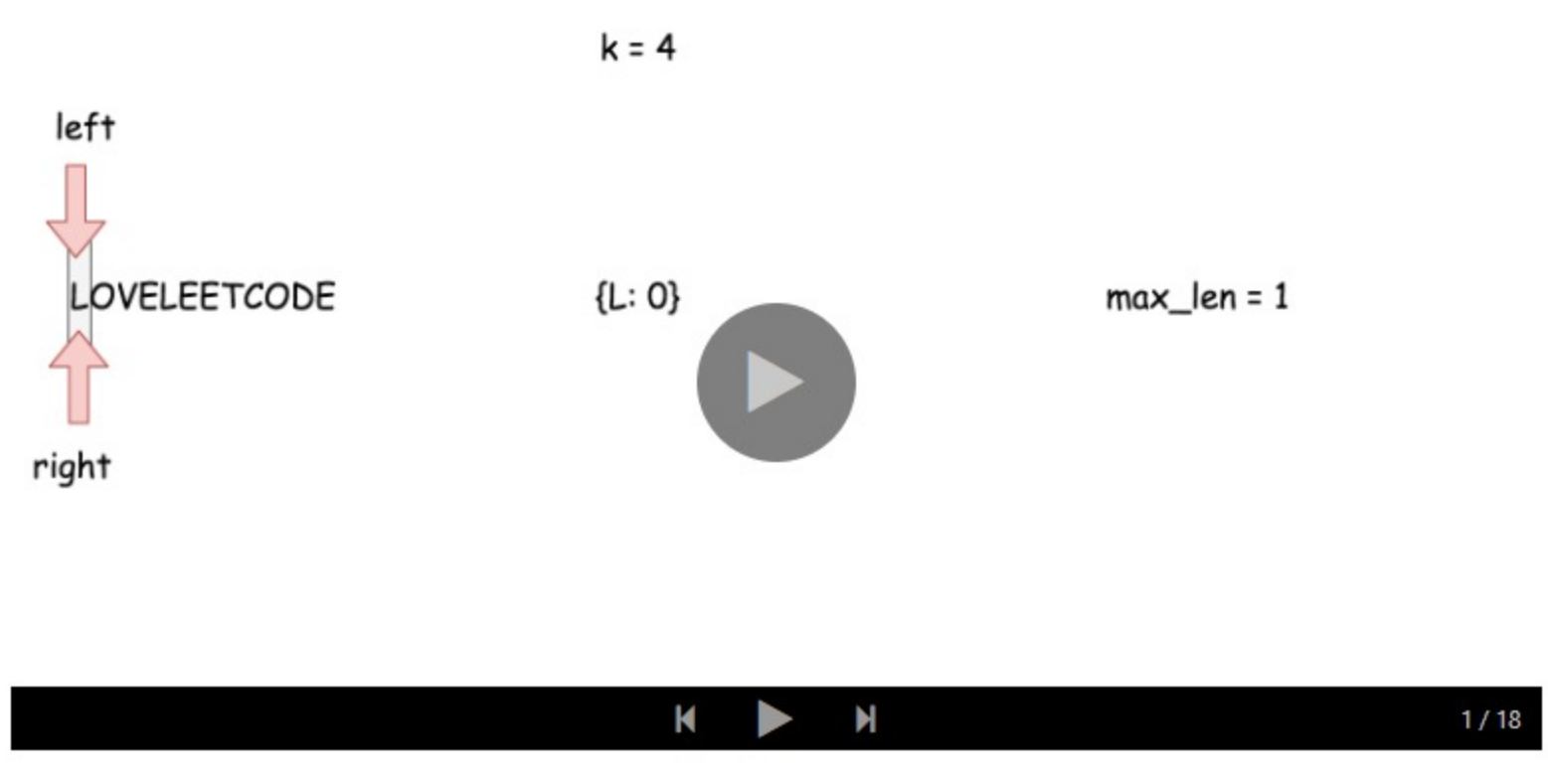
For example, using this hashmap one knows that the rightmost position of character `O` in `"LOVELEE"` window is `1` and so one has to move `left` pointer in the position `1 + 1 = 2` to exclude the character `O` from the sliding window.

Algorithm

Now one could write down the algortihm.

- Return `0` if the string is empty or `k` is equal to zero.
- Set both set pointers in the beginning of the string `left = 0` and `right = 0` and init max substring length `max_len = 1`.
- While `right` pointer is less than `N`:
  - Add the current character `s[right]` in the hashmap and move `right` pointer to the right.
  - If hashmap contains `k + 1` distinct characters, remove the leftmost character from the hashmap and move the `left` pointer so that sliding window contains again `k` distinct characters only.
  - Update `max_len`.

Implementation



Java | Python | Copy

```python
from collections import defaultdict
class Solution:
    def lengthOfLongestSubstringKDistinct(self, s: 'str', k: 'int') -> 'int':
        n = len(s)
        if k == 0 or n == 0:
            return 0

        # sliding window left and right pointers
        left, right = 0, 0
        # hashmap character -> its rightmost position
        # in the sliding window
        hashmap = defaultdict()

        max_len = 1

        while right < n:
            # add new character and move right pointer
            hashmap[s[right]] = right
            right += 1

            # slidewindow contains 3 characters
            if len(hashmap) == k + 1:
                # delete the leftmost character
                del_idx = min(hashmap.values())
                del hashmap[s[del_idx]]
                # move left pointer of the slidewindow
                left = del_idx + 1
```

Complexity Analysis

> Do we have here the best possible time complexity $\mathcal{O}(N)$ as it was for more simple problem with at most two distinct characters?

For the best case when input string contains not more than `k` distinct characters the answer is *yes*. It's the only one pass along the string with `N` characters and the time complexity is $\mathcal{O}(N)$.

For the worst case when the input string contains `n` distinct characters, the answer is *no*. In that case at each step one uses $\mathcal{O}(k)$ time to find a minimum value in the hashmap with `k` elements and so the overall time complexity is $\mathcal{O}(Nk)$.

- Time complexity : $\mathcal{O}(N)$ in the best case of `k` distinct characters in the string and $\mathcal{O}(Nk)$ in the worst case of `N` distinct characters in the string.
- Space complexity : $\mathcal{O}(k)$ since additional space is used only for a hashmap with at most `k + 1` elements.

Approach 2: Sliding Window + Ordered Dictionary.

How to achieve $\mathcal{O}(N)$ time complexity

Approach 1 with a standard hashmap couldn't ensure $\mathcal{O}(N)$ time complexity.

To have $\mathcal{O}(N)$ algorithm performance, one would need a structure, which provides four operations in $\mathcal{O}(1)$ time :

- Insert the key
- Get the key / Check if the key exists
- Delete the key
- Return the first / or the last added key/value

The first three operations in $\mathcal{O}(1)$ time are provided by the standard hashmap, and the forth one - by linked list.

> There is a structure called *ordered dictionary*, it combines behind both hashmap and linked list. In Python this structure is called *OrderedDict* and in Java *LinkedHashMap*.

Ordered dictionary is quite popular for the interviews, for example, check to implement LRU cache question by Google.

Algorithm

Let's use ordered dictionary instead of standard hashmap to trim the algorithm from the approach 1 :

- Return `0` if the string is empty or `k` is equal to zero.
- Set both set pointers in the beginning of the string `left = 0` and `right = 0` and init max substring length `max_len = 1`.
- While `right` pointer is less than `N`:
  - If the current character `s[right]` is already in the ordered dictionary `hashmap` -- delete it, to ensure that the first key in `hashmap` is the leftmost character.
  - Add the current character `s[right]` in the ordered dictionary and move `right` pointer to the right.
  - If ordered dictionary `hashmap` contains `k + 1` distinct characters, remove the leftmost one and move the `left` pointer so that sliding window contains again `k` distinct characters only.
  - Update `max_len`.

Implementation

Java | Python | Copy

```python
1  from collections import OrderedDict
2  class Solution:
3      def lengthOfLongestSubstringKDistinct(self, s: 'str', k: 'int') -> 'int':
4          n = len(s)
5          if k == 0 or n == 0:
6              return 0
7
8          # sliding window left and right pointers
9          left, right = 0, 0
10         # hashmap character -> its rightmost position
11         # in the sliding window
12         hashmap = OrderedDict()
13
14         max_len = 1
15
16         while right < n:
17             character = s[right]
18             # if character is already in the hashmap -
19             # delete it, so that after insert it becomes
20             # the rightmost element in the hashmap
21             if character in hashmap:
22                 del hashmap[character]
23             hashmap[character] = right
24             right += 1
25
26             # slidewindow contains k + 1 characters
27             if len(hashmap) == k + 1:
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N)$ since all operations with ordered dictionary : `insert/get/delete/popitem` (`put/containsKey/remove`) are done in a constant time.
- Space complexity : $\mathcal{O}(k)$ since additional space is used only for an ordered dictionary with at most `k + 1` elements.

Analysis written by @liaison and @andvary

Rate this article:

Comments: 25

Sort By

Type comment here... (Markdown is supported)

Preview Post

sanyam3 ★ 75 June 24, 2019 1:40 AM

Why is this question hard again? O(N) solution can be reached using a simple hashmap no need for the overly complicated second approach!

```
class Solution {
```

Read More

37 Share Reply

SHOW 6 REPLIES

jack103 ★ 28 February 25, 2019 8:10 AM

I don't get why solution 1's time complexity is not just O(N). You have 2 pointers. Right pointer takes n steps. Left pointer can take at most n steps because it's always trailing right. You can argue the time complexity is O(2N), which is O(N), but you cannot say it's O(Nk).

15 Share Reply

SHOW 5 REPLIES

zan00 ★ 10 February 22, 2019 4:18 PM

Great explanation, Thanks.

6 Share Reply

playboy9817 ★ 329 October 29, 2019 7:46 PM

don't store the index, instead, store the count of each character, once right point sees a new char (count==0), we increase the number of distinct character by 1, left pointer keeps decreasing the count, once count reaches 0, we decrease it. ordered dict is totally unnecessary

Read More

5 Share Reply

ts1992 ★ 50 May 2, 2019 6:08 PM

For solution 2, if you used access-ordered LInkedHashMap, it could save you some time from reinserting entries every time (line 17 -21). The constructor looks like this: `Map<Character, Integer> map = new LinkedHashMap<>(k + 1, 1, true);`

3 Share Reply

YuhanWu ★ 160 March 16, 2019 8:50 AM

You can definitely reach O(N) complexity with sliding window with two pointers, please refer to the following code.

```
    public int lengthOfLongestSubstringKDistinct(String s, int k) {
```

Read More

5 Share Reply

SHOW 2 REPLIES

milinthosani ★ 23 October 21, 2019 1:30 PM

I think my solution is O(n) in any worst case. Let me know if you think it is not. Will appreciate constructive criticism.

```
class Solution(object):
```

Read More

1 Share Reply

migeater ★ 80 March 3, 2019 5:01 PM

Why do we need defaultdict? It seems you hasn't use its default value.
And after python3.7

> the insertion-order preservation nature of dict objects has been declared to be an official part of the Python language spec.

1 Share Reply

ellisa_khoja ★ 41 July 9, 2019 6:48 AM

Awesome article. Very intuitive ;)

1 Share Reply

ganeyev ★ 4 March 4, 2019 11:23 PM

We can tune approach 1 to make it O(n) (actually, it will become Theta(2N), but time complexity will remain the same).

We keep right pointer and left pointers. Every time we store in dictionary more than allowed (k +1), we increase left pointer.

Read More

1 Share Reply

SHOW 1 REPLY