Length of Longest Substring without any Repeating Character

Problem Statement: Given a String, find the length of longest substring without any repeating character.

Examples:

Example 1:

Input: s = ”abcabcbb”

Output: 3

Explanation: The answer is abc with length of 3.

Example 2:

Input: s = ”bbbbb”

Output: 1

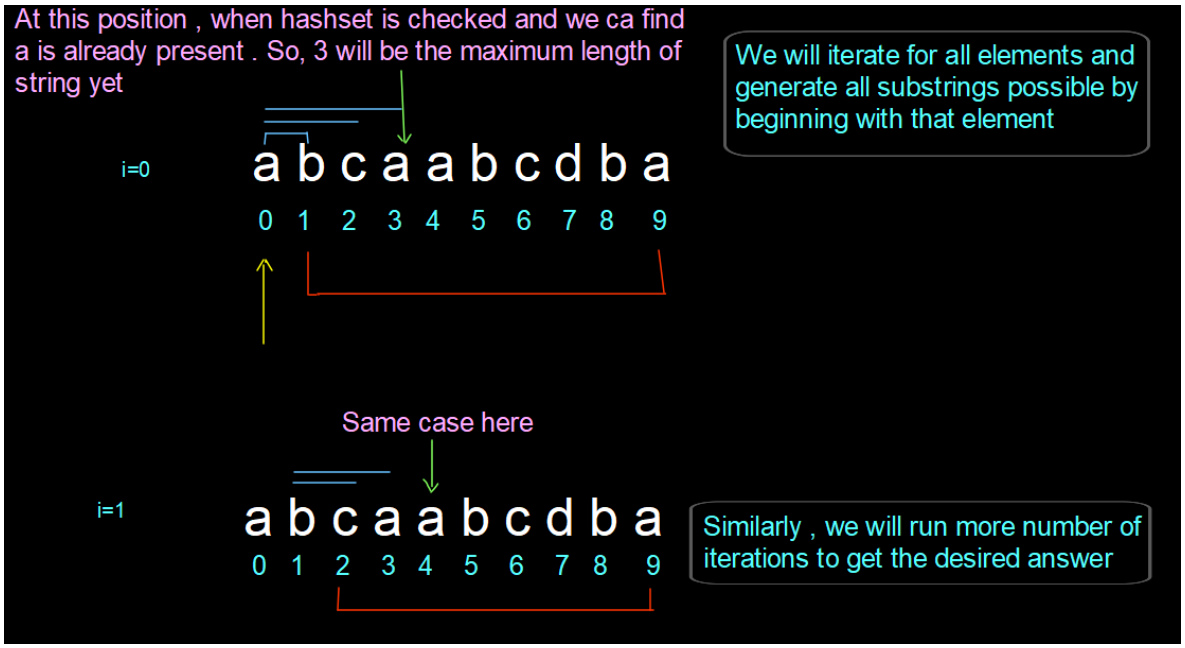
Explanation: The answer is b with length of 1 units.

Solution

Disclaimer: *Don’t jump directly to the solution, try it out yourself first.*

Solution 1: Brute force Approach

Approach: This approach consists of taking the two loops one for traversing the string and another loop – nested loop for finding different substrings and after that, we will check for all substrings one by one and check for each and every element if the element is not found then we will store that element in HashSet otherwise we will break from the loop and count it.



Source code:

```
static int solve(String str) {
    int maxans = Integer.MIN_VALUE;
    for (int i = 0; i < str.length(); i++) // outer loop for traversing the string
    {
        Set < Character > se = new HashSet < > ();
        for (int j = i; j < str.length(); j++) // nested loop for getting different
        string starting with str[i]
        {
            if (se.contains(str.charAt(j))) // if element if found so mark it as ans
            and break from the loop
            {
                maxans = Math.max(maxans, j - i);
                break;
            }
            se.add(str.charAt(j));
        }
```

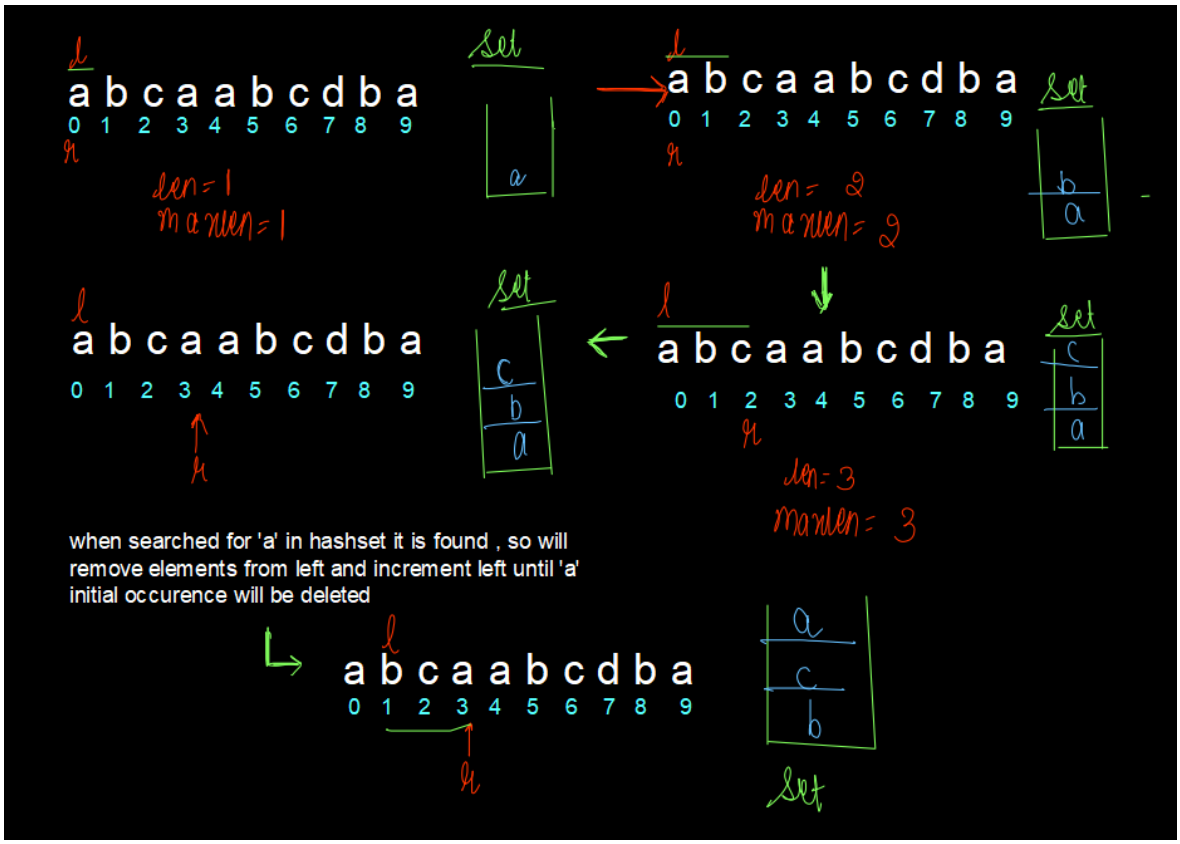
```
        }
        return maxans;
    }
```

Time Complexity: O(N^2)

Space Complexity: O(N) where N is the size of HashSet taken for storing the elements

Solution 2: Optimised Approach 1

Approach: We will have two pointers left and right. Pointer 'left' is used for maintaining the starting point of substring while 'right' will maintain the endpoint of the substring.' right' pointer will move forward and check for the duplicate occurrence of the current element if found then 'left' pointer will be shifted ahead so as to delete the duplicate elements.



Source Code:

```
public int lengthOfLongestSubstring(String s) {
        HashSet<Character> visitedSet=new HashSet<>();
```

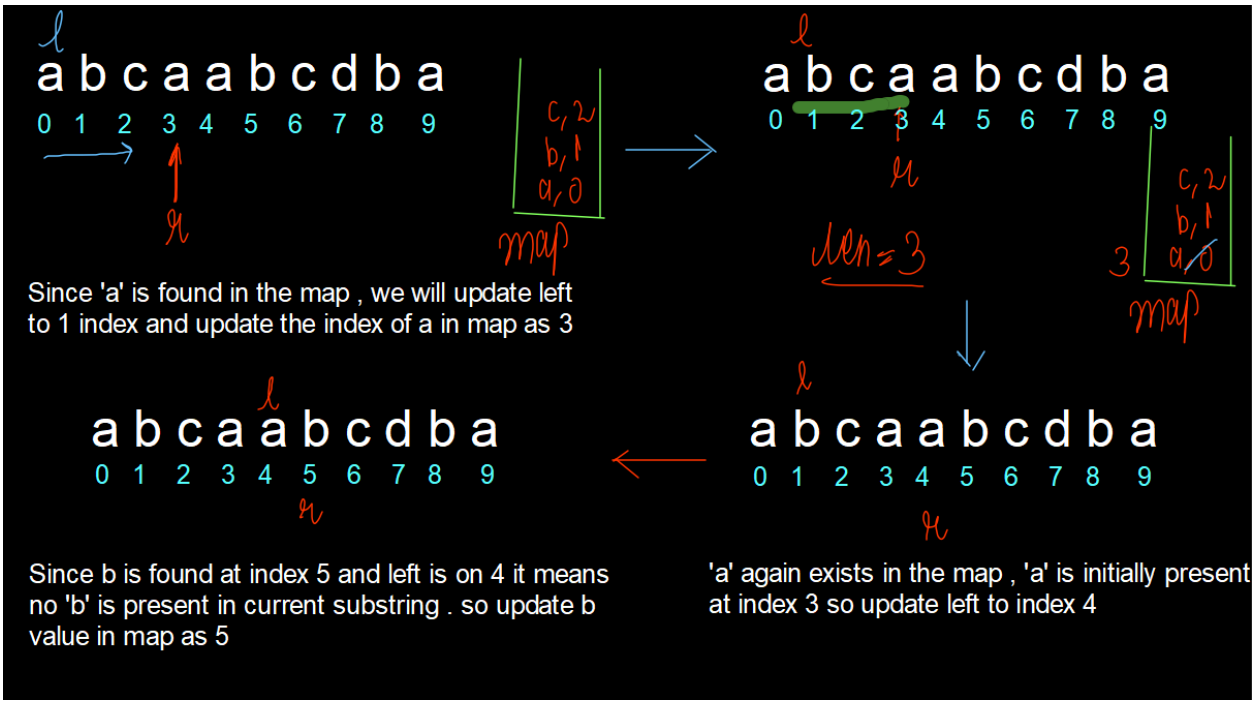
```
        int left=0,right=0,n=s.length();
        int count=0;
        while(right<n)
        {
            char c=s.charAt(right);
            while(!visitedSet.isEmpty()&&visitedSet.contains(c))
            {
                visitedSet.remove(s.charAt(left));
                left++;
            }
            visitedSet.add(c);
            count=Math.max(count,(right-left+1));
            right++;
        }
        return count;
    }
```

Time Complexity: O(2*N) (sometimes left and right both have to travel complete array)

Space Complexity: O(N) where N is the size of HashSet taken for storing the elements

Solution 3: Optimised Approach 2

Approach: In this approach, we will make a map that will take care of counting the elements and maintaining the frequency of each and every element as a unity by taking the latest index of every element.



Source Code:

```
public int lengthOfLongestSubstring(String s) {
        HashMap<Character,Integer> lastSeenMap=new HashMap<>();
        int left=0,right=0,n=s.length();
        int count=0;
        while(right<n)
        {
            char c=s.charAt(right);
            if(lastSeenMap.containsKey(c))
            {
                int lastSeen=lastSeenMap.get(c);
                if(lastSeen>=left)
                {
                    left=lastSeen+1;
                }
            }
            lastSeenMap.put(c,right);
```

```
            count=Math.max(count,(right-left+1));
            right++;
        }
        return count;
    }
```

Time Complexity: O(N)

Space Complexity: O(N) where N represents the size of HashSet where we are storing our elements