```
public class Sliding_Window_Two_Pointer {
    // Longest Substring Without Repeating Characters
    class Solution {
        int longestUniqueSubsttr(String S) {
            int start = 0, end = 0, unique = 0, repeat = 0, ans = 0;
            int freq[] = new int[256];
            int n = S.length();
            while (end < n) {
                freq[S.charAt(end)]++;
                if (freq[S.charAt(end)] > 1) {
                    repeat++;
                }
                end++;
                while (start < end && repeat > 0) {
                    if (freq[S.charAt(start)] > 1) {
                        repeat--;
                    }
                    freq[S.charAt(start)]--;
                    start++;
                }
                if (repeat == 0) {
                    ans = Math.max(end - start, ans);
                }
            }
            return ans;
        }
    }

    // Max Consecutive Ones III
    class Solution {
        public int longestOnes(int[] nums, int k) {
            int start = 0, end = 0, flips = 0;
            int n = nums.length;
            int ans = -1;
            while (end < n) {
                if (nums[end] == 0) {
                    flips++;
                }
                end++;
                while (start < end && flips > k) {
                    if (nums[start] == 0) {
                        flips--;
                    }
                    start++;
                }
                if (flips <= k) {
                    ans = Math.max(end - start, ans);
                }
            }
```

```
        return ans;
    }
}

// Fruit Into Baskets
class Solution {
    public int totalFruit(int[] nums) {
        int start = 0, end = 0;
        int n = nums.length;
        int ans = -1;
        Map<Integer, Integer> mp = new HashMap<>();
        while (end < n) {
            if (mp.get(nums[end]) == null) {
                mp.put(nums[end], 1);
            } else
                mp.put(nums[end], mp.get(nums[end]) + 1);
            end++;

            while (start < end && mp.size() > 2) {
                int key = nums[start];
                if (mp.get(key) == 1) {
                    mp.remove(key);
                } else {
                    mp.put(nums[start], mp.get(nums[start]) - 1);
                }
                start++;
            }
            if (mp.size() <= 2) {
                ans = Math.max(end - start, ans);
            }
        }
        return ans;
    }
}

// Longest repeating character replacement
class Solution {
    public int characterReplacement(String s, int k) {
        int ans = Integer.MIN_VALUE;
        for (char ch = 'A'; ch <= 'Z'; ch++) {
            ans = Math.max(ans, solve(s, k, ch));
        }
        return ans;
    }

    public int solve(String s, int k, char ch) {
        int start = 0, end = 0, changes = 0, ans = 0, n = s.length();
        while (end < n) {
            if (s.charAt(end) != ch) {
```

```
                changes++;
            }
            end++;
            while (start < end && changes > k) {
                if (s.charAt(start) != ch) {
                    changes--;
                }
                start++;
            }
            ans = Math.max(ans, end - start);
        }
        return ans;
    }

}

// Binary subarray with sum
class Solution {
    public int numSubarraysWithSum(int[] nums, int k) {
        return countSubArray(nums, k) - countSubArray(nums, k - 1);
    }

    public int countSubArray(int nums[], int k) {
        int start = 0, end = 0, n = nums.length, sum = 0, ans = 0;
        while (end < n) {
            sum += nums[end];
            end++;
            while (start < end && sum > k) {
                sum -= nums[start];
                start++;
            }
            ans += end - start;
        }
        return ans;
    }
}

// Count number of nice subarrays
class Solution {
    public int countSubArray(int nums[], int k) {
        int start = 0, end = 0, n = nums.length, noOfOdd = 0, ans = 0;
        while (end < n) {
            if (nums[end] % 2 == 1) {
                noOfOdd++;
            }
            end++;
            while (start < end && noOfOdd > k) {
                if (nums[start] % 2 == 1) {
                    noOfOdd--;
```

```
                }
                start++;
            }
            ans += end - start;
        }
        return ans;
    }

    public int numberOfSubarrays(int[] nums, int k) {
        return countSubArray(nums, k) - countSubArray(nums, k - 1);
    }
}

// Number of substring containing all three characters
class Solution {
    public int numberOfSubstrings(String s) {
        int n = s.length();
        int start = 0;
        int end = 0;
        int ans = 0;
        Map<Character, Integer> map = new HashMap<>();
        while (end < n) {
            map.put(s.charAt(end), map.getOrDefault(s.charAt(end), 0) + 1);
            if (!map.containsKey('a') || !map.containsKey('b') || !map.containsKey('c')) {
                end++;
            }

            else {
                while (map.containsKey('a') && map.containsKey('b') && map.
containsKey('c')) {
                    ans += n - end;
                    char ch = s.charAt(start);
                    map.put(ch, map.getOrDefault(ch, 0) - 1);
                    if (map.get(ch) == 0) {
                        map.remove(ch);
                    }
                    start++;
                }
                end++;
            }

        }
        return ans;
    }
}

// Maximum point you can obtain from cards
class Solution {
    public int maxScore(int[] cardPoints, int k) {
```

```
        int n = cardPoints.length;
        int dp[][] = new int[n + 1][n + 1];
        for (int rows[] : dp) {
            Arrays.fill(rows, 0);
        }
        return f(0, n - 1, 0, k, cardPoints, dp);
    }

    public int f(int start, int end, int ct, int k, int cardPoints[], int dp[][]) {
        if (ct == k) {
            return 0;
        }
        if (dp[start][end] != -1) {
            return dp[start][end];
        }
        int takeFront = cardPoints[start] + f(start + 1, end, ct + 1, k, cardPoints, dp);
        int takeBack = cardPoints[end] + f(start, end - 1, ct + 1, k, cardPoints, dp);
        return dp[start][end] = Math.max(takeFront, takeBack);
    }

    class Solution {
        public int maxScore(int[] cardPoints, int k) {
            int sum = 0;
            int n = cardPoints.length;
            for (int i = 0; i < k; i++) {
                sum += cardPoints[i];
            }
            int maxSum = sum;
            for (int i = k - 1, j = n - 1; i >= 0; i--, j--) {
                sum -= cardPoints[i];
                sum += cardPoints[j];
                maxSum = Math.max(maxSum, sum);
            }
            return maxSum;
        }
    }
}

// Longest Substring with At Most K Unique characters
class Solution {
    public int longestkSubstr(String s, int k) {
        // code here
        int start = 0;
        int end = 0;
        int unique = 0;
        int ans = -1;
        int[] freq = new int[123];
        int n = s.length();
        while (end < n) {
```

```
            freq[s.charAt(end)]++;
            if (freq[s.charAt(end)] == 1)
                unique++;
            end++;

            while (start < end && unique > k) {
                freq[s.charAt(start)]--;
                if (freq[s.charAt(start)] == 0)
                    unique--;
                start++;
            }

            if (unique == k) {
                ans = Math.max(ans, end - start);
            }
        }
        return ans;
    }
}

// Subarray with k different integers
class Solution {
    public int subarraysWithKDistinct(int[] nums, int k) {
        return f(nums, k) - f(nums, k - 1);
    }

    public int f(int nums[], int k) {
        int start = 0, end = 0, n = nums.length, ans = 0, unique = 0;
        int freq[] = new int[10000000];
        while (end < n) {
            freq[nums[end]]++;
            if (freq[nums[end]] == 1)
                unique++;
            end++;
            while (start < end && unique > k) {
                freq[nums[start]]--;
                if (freq[nums[start]] == 0)
                    unique--;
                start++;
            }
            ans += end - start;
        }
        return ans;
    }
}

// Minimum Window Substring
class Solution{
    boolean isSatisfied(int[] sfreq, int[] tfreq) {
```

```
            for(int i = 0; i < 123; i++) {
            if(tfreq[i] > sfreq[i]) return false;
            }
            return true;
        }
        public String minWindow(String s, String t) {
            int n = s.length(), m = t.length();
            int[] tfreq = new int[123];
            for(int i = 0; i < m; i++) tfreq[t.charAt(i)]++;
            int[] sfreq = new int[123];
            int start = 0, end = 0;
            int ans = Integer.MAX_VALUE;
            int ansStart = -1, ansEnd = -1;
            while(end < n) {
                sfreq[s.charAt(end)]++;
                end++;
                while(start < end && isSatisfied(sfreq, tfreq)) {
                    if(ans > end - start) {
                    ans = end - start;
                    ansStart = start;
                    ansEnd = end;
                }
                sfreq[s.charAt(start)]--;
                start++;
            }
        }
        if(ansStart == -1) return "";
        return s.substring(ansStart, ansEnd);
    }
}
    // Minimum Window Subsequence
    public class Main {
        public static String DistinctWindow(String s) {
            int n = s.length();
            int cnt[] = new int[123];
            int distinct = 0;
            for (int i = 0; i < n; i++) {
                cnt[s.charAt(i)]++;
                if (cnt[s.charAt(i)] == 1)
                    distinct++;
            }
            return smallest(s, n, distinct);
        }

        public static String smallest(String s, int n, int k) {
            // write code here

            int ans = Integer.MAX_VALUE, start = 0, end = 0, unique = 0, ansStart = -1,
ansEnd = -1;
```

```
        int freq[] = new int[123];
        while (end < n) {
            char ch = s.charAt(end);
            freq[ch]++;
            if (freq[ch] == 1) {
                unique++;
            }
            end++;

            while (start < end && unique == k) {

                if (ans > end - start) {
                    ans = end - start;
                    ansStart = start;
                    ansEnd = end;
                }
                char g = s.charAt(start);
                freq[g]--;
                if (freq[g] == 0) {
                    unique--;
                }
                start++;
            }
        }
        if (ans == -1) {
            return "";
        }
        return s.substring(ansStart, ansEnd);
    }

    public static void main(String[] args) throws Throwable {
        Scanner sc = new Scanner(System.in);
        String s = sc.nextLine();
        String ans = DistinctWindow(s);
        System.out.println(ans);
    }
  }
}
```