```
class Solution {
    public int[] twoSum(int[] nums, int target){
        for(int i = 0; i < nums.length; i++) {
            for(int j=i+1; j<nums.length; j++){
                if (nums[i]+ nums[j]==target) {
                    return new int[]{i,j};
                }
            }
        }
        return null;
    }
}
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummyHead = new ListNode(0);
        ListNode tail = dummyHead;
        int carry = 0;

        while (l1 != null || l2 != null || carry != 0) {
            int digit1 = (l1 != null) ? l1.val : 0;
            int digit2 = (l2 != null) ? l2.val : 0;

            int sum = digit1 + digit2 + carry;
            int digit = sum % 10;
            carry = sum / 10;

            ListNode newNode = new ListNode(digit);
```

```
            tail.next = newNode;
            tail = tail.next;

            l1 = (l1 != null) ? l1.next : null;
            l2 = (l2 != null) ? l2.next : null;
        }

        ListNode result = dummyHead.next;
        dummyHead.next = null;
        return result;
    }
}
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        int maxLength = 0;
        Set<Character> charSet = new HashSet<>();
        int left = 0;

        for (int right = 0; right < n; right++) {
            if (!charSet.contains(s.charAt(right))) {
                charSet.add(s.charAt(right));
                maxLength = Math.max(maxLength, right - left + 1);
            } else {
                while (charSet.contains(s.charAt(right))) {
                    charSet.remove(s.charAt(left));
                    left++;
```

```
            }
            charSet.add(s.charAt(right));
          }
        }

        return maxLength;
    }
}
class Solution {
    int maxLen = 0;
    int lo = 0;
    public String longestPalindrome(String s) {
        char[] input = s.toCharArray();
        if(s.length() < 2) {
            return s;
        }

        for(int i = 0; i<input.length; i++) {
            expandPalindrome(input, i, i);
            expandPalindrome(input, i, i+1);
        }
        return s.substring(lo, lo+maxLen);
    }

    public void expandPalindrome(char[] s, int j, int k) {
        while(j >= 0 && k < s.length && s[j] == s[k]) {
            j--;
```

```
            k++;
        }
        if(maxLen < k - j - 1) {
            maxLen = k - j - 1;
            lo = j+1;
        }
    }
}
public class Solution {
    public boolean isPalindrome(int x) {
        if (x < 0 || (x % 10 == 0 && x != 0)) {
            return false;
        }
        int rev = 0;
        while (x > rev) {
            int digit = x % 10;
            rev = rev * 10 + digit;
            x = x / 10;
        }
        return x == rev || x == rev / 10;
    }
}
import java.util.HashMap;


public class Solution {
    public int romanToInt(String s) {
        HashMap<Character, Integer> romanMap = new HashMap<>();
```

```
        romanMap.put('I', 1);
        romanMap.put('V', 5);
        romanMap.put('X', 10);
        romanMap.put('L', 50);
        romanMap.put('C', 100);
        romanMap.put('D', 500);
        romanMap.put('M', 1000);

        int total = 0;
        for (int i = 0; i < s.length(); i++) {
            int currentVal = romanMap.get(s.charAt(i));

            if (i < s.length() - 1 && currentVal < romanMap.get(s.charAt(i + 1))) {
                total -= currentVal;
            } else {
                total += currentVal;
            }
        }

        return total;
    }
}
public class Solution {
    public String longestCommonPrefix(String[] strs) {
        if (strs == null || strs.length == 0) {
            return "";
        }
```

```
        String prefix = strs[0];
        for (int i = 1; i < strs.length; i++) {
            while (strs[i].indexOf(prefix) != 0) {
                prefix = prefix.substring(0, prefix.length() - 1);
                if (prefix.isEmpty()) {
                    return "";
                }
            }
        }

        return prefix;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        String[] strs1 = {"flower", "flow", "flight"};
        System.out.println(solution.longestCommonPrefix(strs1));
        String[] strs2 = {"dog", "racecar", "car"};
        System.out.println(solution.longestCommonPrefix(strs2));
    }
}
import java.util.Stack;

public class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        for (int i = 0; i < s.length(); i++) {
```

```
        char c = s.charAt(i);
        if (c == '(' || c == '{' || c == '[') {
            stack.push(c);
        }
        else {
            if (stack.isEmpty()) {
                return false;
            }
            char top = stack.pop();
            if (c == ')' && top != '(') {
                return false;
            }
            if (c == '}' && top != '{') {
                return false;
            }
            if (c == ']' && top != '[') {
                return false;
            }
        }
    }
    return stack.isEmpty();
}

public static void main(String[] args) {
    Solution solution = new Solution();
    String s1 = "()";
    System.out.println(solution.isValid(s1));
```

```
        String s2 = "()[]{}";
        System.out.println(solution.isValid(s2));
        String s3 = "(]";
        System.out.println(solution.isValid(s3));
        String s4 = "([)]";
        System.out.println(solution.isValid(s4));
    }
}
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
// Definition for singly-linked list
class ListNode {
    int val;
    ListNode next;
    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }

    // Deserialize a string into a linked list
```

```
public static ListNode deserialize(String s) {
    // Remove the square brackets and split by commas
    s = s.substring(1, s.length() - 1); // Remove the '[' and ']'
    if (s.isEmpty()) return null; // If the string is empty, return null

    String[] values = s.split(",");
    ListNode dummy = new ListNode(0); // Dummy node to simplify logic
    ListNode current = dummy;

    // Iterate over the string values and create the linked list
    for (String value : values) {
        ListNode newNode = new ListNode(Integer.parseInt(value.trim()));
        current.next = newNode;
        current = current.next;
    }

    return dummy.next; // Return the head of the linked list
}

// Helper function to print the list (for testing)
public static void printList(ListNode head) {
    ListNode current = head;
    while (current != null) {
        System.out.print(current.val + " ");
        current = current.next;
    }
    System.out.println();
```

```
    }
}

public class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        // Create a dummy node to help simplify the merging process
        ListNode dummy = new ListNode(-1);
        ListNode current = dummy;

        // Iterate through both lists
        while (list1 != null && list2 != null) {
            // Compare the values of the two lists and append the smaller one
            if (list1.val <= list2.val) {
                current.next = list1;
                list1 = list1.next;  // Move the pointer in list1
            } else {
                current.next = list2;
                list2 = list2.next;  // Move the pointer in list2
            }
            current = current.next;  // Move the current pointer to the next node
        }

        // If one of the lists is not empty, append the remaining nodes
        if (list1 != null) {
            current.next = list1;
        } else if (list2 != null) {
            current.next = list2;
```

```
    }

    // Return the merged list, which starts at dummy.next
    return dummy.next;
}

public static void main(String[] args) {
    Solution solution = new Solution();

    // Example 1: list1 = [1, 2, 4], list2 = [1, 3, 4]
    String list1Str = "[1, 2, 4]";
    String list2Str = "[1, 3, 4]";
    ListNode list1 = ListNode.deserialize(list1Str);
    ListNode list2 = ListNode.deserialize(list2Str);
    ListNode mergedList = solution.mergeTwoLists(list1, list2);
    ListNode.printList(mergedList);  // Output: [1, 1, 2, 3, 4, 4]

    // Example 2: list1 = [], list2 = []
    String list3Str = "[]";
    String list4Str = "[]";
    ListNode list3 = ListNode.deserialize(list3Str);
    ListNode list4 = ListNode.deserialize(list4Str);
    ListNode mergedList2 = solution.mergeTwoLists(list3, list4);
    ListNode.printList(mergedList2);  // Output: []

    // Example 3: list1 = [], list2 = [0]
    String list5Str = "[]";
```

```
        String list6Str = "[0]";
        ListNode list5 = ListNode.deserialize(list5Str);
        ListNode list6 = ListNode.deserialize(list6Str);
        ListNode mergedList3 = solution.mergeTwoLists(list5, list6);
        ListNode.printList(mergedList3);  // Output: [0]
    }
}
public class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0; // No elements to process
        }

        // Pointer k will track the position to place the next unique element
        int k = 1;

        // Iterate through the array starting from the second element
        for (int i = 1; i < nums.length; i++) {
            // When a new unique element is found
            if (nums[i] != nums[i - 1]) {
                nums[k] = nums[i]; // Move it to the next available position
                k++; // Increment the position for the next unique element
            }
        }

        // Return the number of unique elements
        return k;
```

```
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        // Example 1
        int[] nums1 = {1, 1, 2};
        int k1 = solution.removeDuplicates(nums1);
        System.out.println("Number of unique elements: " + k1); // Output: 2
        System.out.print("Modified array: ");
        for (int i = 0; i < k1; i++) {
            System.out.print(nums1[i] + " ");
        }
        System.out.println();

        // Example 2
        int[] nums2 = {0,0,1,1,1,2,2,3};
        int k2 = solution.removeDuplicates(nums2);
        System.out.println("Number of unique elements: " + k2); // Output: 4
        System.out.print("Modified array: ");
        for (int i = 0; i < k2; i++) {
            System.out.print(nums2[i] + " ");
        }
        System.out.println();
    }
}
public class Solution {
```

```
public int strStr(String haystack, String needle) {
    // Check if the needle is empty, return 0 as per the problem description
    if (needle.isEmpty()) {
        return 0;
    }

    // Use indexOf to find the first occurrence of needle in haystack
    return haystack.indexOf(needle);
}

public static void main(String[] args) {
    Solution solution = new Solution();

    // Example 1
    String haystack1 = "sadbutsad";
    String needle1 = "sad";
    System.out.println("First occurrence index: " + solution.strStr(haystack1, needle1)); // Output: 0

    // Example 2
    String haystack2 = "hello";
    String needle2 = "ll";
    System.out.println("First occurrence index: " + solution.strStr(haystack2, needle2)); // Output: 2

    // Example 3
    String haystack3 = "aaaaa";
    String needle3 = "bba";
    System.out.println("First occurrence index: " + solution.strStr(haystack3, needle3)); // Output: -1
```

```
    }
}
public class Solution {
    public int searchInsert(int[] nums, int target) {
        int low = 0;
        int high = nums.length - 1;

        // Binary search loop
        while (low <= high) {
            int mid = low + (high - low) / 2;

            // Check if target is at mid
            if (nums[mid] == target) {
                return mid;
            }
            // If target is smaller, move the high pointer
            else if (nums[mid] > target) {
                high = mid - 1;
            }
            // If target is larger, move the low pointer
            else {
                low = mid + 1;
            }
        }

        // If target is not found, low will be the position to insert
        return low;
```

```
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        // Example 1
        int[] nums1 = {1, 3, 5, 6};
        int target1 = 5;
        System.out.println("Insert position: " + solution.searchInsert(nums1, target1)); // Output: 2

        // Example 2
        int[] nums2 = {1, 3, 5, 6};
        int target2 = 2;
        System.out.println("Insert position: " + solution.searchInsert(nums2, target2)); // Output: 1

        // Example 3
        int[] nums3 = {1, 3, 5, 6};
        int target3 = 7;
        System.out.println("Insert position: " + solution.searchInsert(nums3, target3)); // Output: 4

        // Example 4
        int[] nums4 = {1, 3, 5, 6};
        int target4 = 0;
        System.out.println("Insert position: " + solution.searchInsert(nums4, target4)); // Output: 0
    }
}
public class Solution {
```

```
    public int lengthOfLastWord(String s) {
        s = s.trim();
        String[] words = s.split(" ");
        return words[words.length - 1].length();
    }
}
public class Solution {
    public int[] plusOne(int[] digits) {
        int n = digits.length;
        for (int i = n - 1; i >= 0; i--) {
            if (digits[i] < 9) {
                digits[i]++;
                return digits;
            }
            digits[i] = 0;
        }
        int[] result = new int[n + 1];
        result[0] = 1;
        for (int i = 0; i < n; i++) {
            result[i + 1] = digits[i];
        }

        return result;
    }
}
public class Solution {
    public String addBinary(String a, String b) {
```

```
        StringBuilder result = new StringBuilder();
        int i = a.length() - 1;
        int j = b.length() - 1;
        int carry = 0;
        while (i >= 0 || j >= 0 || carry != 0) {
            int sum = carry;
            if (i >= 0) {
                sum += a.charAt(i) - '0';
                i--;
            }
            if (j >= 0) {
                sum += b.charAt(j) - '0';
                j--;
            }
            result.append(sum % 2);
            carry = sum / 2;
        }
        return result.reverse().toString();
    }
}
public class Solution {
    public int climbStairs(int n) {
        if (n == 1) {
            return 1;  // Only one way to reach the top if there's just one step
        }

        int first = 1;  // dp[0], number of ways to stay at the ground level
```

```
        int second = 2; // dp[1], number of ways to reach the first step

        for (int i = 3; i <= n; i++) {
            int current = first + second; // Current number of ways to reach the i-th step
            first = second;  // Move first to the previous second
            second = current; // Move second to the current value
        }

        return second; // After the loop, second will store the number of ways to reach step n
    }
}
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        // If the list is empty or contains only one node, there are no duplicates to remove
        if (head == null) {
            return head;
        }
```

```
        // Use a pointer to traverse the list
        ListNode current = head;

        while (current != null && current.next != null) {
            // If the current node's value is the same as the next node's value, skip the next node
            if (current.val == current.next.val) {
                current.next = current.next.next;
            } else {
                // Otherwise, move to the next node
                current = current.next;
            }
        }
        return head;  // Return the modified head of the list
    }
}
```