**1. Write an algorithm to determine if a number n is happy.**

**A happy number is a number defined by the following process:**

- **Starting with any positive integer, replace the number by the sum of the squares of its digits.**
- **Repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1.**
- **Those numbers for which this process ends in 1 are happy.**

**Return true *if* n *is a happy number, and* false *if not.***

```
Sol: def is_happy(n: int) -> bool:
    def sum_of_squares(num):
        return sum(int(digit) ** 2 for digit in str(num))

    seen = set()

    while n != 1 and n not in seen:
        seen.add(n)
        n = sum_of_squares(n)

    return n == 1

# Example usage:
n = 19
print(is_happy(n))  # Output: True, because 19 is a happy number
```

**2. Given an integer x, return true *if* x *is a palindrome, and* false *otherwise.***

**Sol:**
```
class Solution {
    public boolean isPalindrome(int x) {
        if (x < 0 || (x > 0 && x % 10 == 0)) {
            return false;
        }
        int y = 0;
        for (; y < x; x /= 10) {
            y = y * 10 + x % 10;
        }
        return x == y || x == y / 10;
    }
}
```

**3. You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.**

**You may assume the two numbers do not contain any leading zero, except the number 0 itself.**

**Sol:**
```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode();
        ListNode *temp = dummy;
        int carry = 0;
        while( (l1 != NULL || l2 != NULL) || carry) {
            int sum = 0;
            if(l1 != NULL) {
                sum += l1->val;
                l1 = l1 -> next;
            }

            if(l2 != NULL) {
                sum += l2 -> val;
                l2 = l2 -> next;
            }

            sum += carry;
            carry = sum / 10;
            ListNode *node = new ListNode(sum % 10);
            temp -> next = node;
            temp = temp -> next;
        }
        return dummy -> next;
    }
};
```

**4. Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.**

**You may assume that each input would have *exactly* one solution, and you may not use the *same* element twice.**

**You can return the answer in any order.**

**Sol**:
```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> d = new HashMap<>();
        for (int i = 0;; ++i) {
            int x = nums[i];
            int y = target - x;
            if (d.containsKey(y)) {
                return new int[] {d.get(y), i};
            }
            d.put(x, i);
        }
    }
}
```

**5. Given the roots of two binary trees p and q, write a function to check if they are the same or not.**

**Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.**

**Sol:**
```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
```

```
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == q) return true;
        if (p == null || q == null || p.val != q.val) return false;
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
}
```

**6. You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.**

**Merge nums1 and nums2 into a single array sorted in non-decreasing order.**

**The final sorted array should not be returned by the function, but instead be *stored inside the array* nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.**

**Sol:** class Solution {

```
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        for (int i = m - 1, j = n - 1, k = m + n - 1; j >= 0; --k) {
            nums1[k] = i >= 0 && nums1[i] > nums2[j] ? nums1[i--] : nums2[j--];
        }
    }
}
```

**7. Given a signed 32-bit integer x, return x *with its digits reversed*. If reversing x causes the value to go outside the signed 32-bit integer range [$-2^{31}$, $2^{31}$ - 1], then return 0.**

**Assume the environment does not allow you to store 64-bit integers (signed or unsigned).**

**Sol:**

```
class Solution {
    public int reverse(int x) {
        int ans = 0;
        for (; x != 0; x /= 10) {
            if (ans < Integer.MIN_VALUE / 10 || ans > Integer.MAX_VALUE / 10) {
                return 0;
            }
            ans = ans * 10 + x % 10;
        }
        return ans;
    }
}
```