Coding Questions

1) Array Manipulation

- Given an array of integers, write a function to move all zeros to the end without changing the order of non-zero elements.

Solution:

```
def move_zeros_to_end(nums):
    non_zeros = [num for num in nums if num != 0]
    zeros = [0] * (len(nums) - len(non_zeros))
    return non_zeros + zeros
```

2) Stack Implementation using Queues

- Implement a stack using queues. The stack should support the following operations: push, pop, top, and empty.

Solution:

```
from collections import deque
class Stack:
    def __init__(self):
        self.queue = deque()

    def push(self, x):
        self.queue.append(x)
        for _ in range(len(self.queue) - 1):
            self.queue.append(self.queue.popleft())
    def pop(self):

        return self.queue.popleft()



    def top(self):

        return self.queue[0]



    def empty(self):

        return len(self.queue) == 0
```

3) Valid Parentheses

- Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

Solution:

```
def is_valid(s):

    stack = []

    mapping = {')': '(', '}': '{', ']': '['}



    for char in s:

        if char in mapping:

            top_element = stack.pop() if stack else '#'

            if mapping[char] != top_element:

                return False

        else:

            stack.append(char)
    return not stack
```

4) Merge Two Sorted Lists

- Merge two sorted linked lists and return it as a new sorted list. The new list should be made by splicing together the nodes of the first two lists.

Solution:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next


def merge_two_lists(l1, l2):
    dummy = ListNode()
    current = dummy


    while l1 and l2:
        if l1.val < l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next


    current.next = l1 if l1 else l2
    return dummy.next
```

5) Linked List Cycle Detection

- Determine whether a linked list has a cycle in it.

Solution:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next


def has_cycle(head):
    slow = fast = head


    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True


  return False
```

6) Binary Tree Level Order Traversal

- Given a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

Solution:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def level_order(root):
    if not root:
        return []

    result = []
    queue = [root]

    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.pop(0)
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result
```

7) Maximum Depth of Binary Tree

- Given a binary tree, find its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Solution:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def max_depth(root):
    if not root:
        return 0
    left_depth = max_depth(root.left)
    right_depth = max_depth(root.right)

    return max(left_depth, right_depth) + 1
```

8) Breadth-First Search (BFS)

- Given a graph represented as an adjacency list and a starting vertex, implement BFS to traverse the graph.

Solution:

```
def bfs(graph, start):
    visited = set()
    queue = [start]

    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            print(vertex, end=' ')
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
```

9) Depth-First Search (DFS)

- Implement DFS to traverse a graph given its adjacency list and a starting vertex.

Solution:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=' ')

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
```

10) **Validate BST**

- Given the root of a binary tree, determine if it is a valid binary search tree (BST).

Solution:

```
class TreeNode:
  def __init__(self, val=0, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right

def is_valid_bst(root, min_val=float('-inf'), max_val=float('inf')):
  if not root:
    return True
  if not min_val < root.val < max_val:
    return False

  return (is_valid_bst(root.left, min_val, root.val) and
      is_valid_bst(root.right, root.val, max_val))
```