52.

```
def generate_parentheses(n):
    def backtrack(s, left, right):
        if len(s) == 2 * n:
            result.append(s)
            return
        if left < n:
            backtrack(s + "(", left + 1, right)
        if right < left:
            backtrack(s + ")", left, right + 1)

    result = []
    backtrack("", 0, 0)
    return result

# Example Input
n = 3

# Example Output
print(generate_parentheses(n))
```

```
['((()))', '(()())', '(())()', '()(())', '()()()']
```

53.

```
from collections import deque

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

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

    result, queue, reverse = [], deque([root]), False

    while queue:
        level = deque()
        for _ in range(len(queue)):
            node = queue.popleft()
            if reverse:
                level.appendleft(node.val)
            else:
                level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(list(level))
        reverse = not reverse

    return result

# Example Input
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

# Example Output
print(zigzag_level_order(root))
```

```
[[1], [3, 2], [4, 5, 6, 7]]
```

54.

```
def is_palindrome(s):
    return s == s[::-1]
```

```
def partition_palindromes(s):
    def backtrack(start, path):
        if start == len(s):
            result.append(path[:])
            return
        for end in range(start + 1, len(s) + 1):
            if is_palindrome(s[start:end]):
                backtrack(end, path + [s[start:end]])

    result = []
    backtrack(0, [])
    return result

# Example Input
s = "aab"

# Example Output
print(partition_palindromes(s))
```

```
[['a', 'a', 'b'], ['aa', 'b']]
```

47.

```
def merge_and_count(arr, temp_arr, left, mid, right):
    i, j, k = left, mid + 1, left
    inv_count = 0

    while i <= mid and j <= right:
        if arr[i] <= arr[j]:
            temp_arr[k] = arr[i]
            i += 1
        else:
            temp_arr[k] = arr[j]
            inv_count += (mid - i + 1)
            j += 1
        k += 1

    while i <= mid:
        temp_arr[k] = arr[i]
        i += 1
        k += 1

    while j <= right:
        temp_arr[k] = arr[j]
        j += 1
        k += 1

    for i in range(left, right + 1):
        arr[i] = temp_arr[i]

    return inv_count

def count_inversions(arr):
    def merge_sort(arr, temp_arr, left, right):
        if left >= right:
            return 0
        mid = (left + right) // 2
        inv_count = merge_sort(arr, temp_arr, left, mid)
        inv_count += merge_sort(arr, temp_arr, mid + 1, right)
        inv_count += merge_and_count(arr, temp_arr, left, mid, right)
        return inv_count

    return merge_sort(arr, [0] * len(arr), 0, len(arr) - 1)

# Example Input
arr = [2, 4, 1, 3, 5]

# Example Output
print(count_inversions(arr))
```

```
3
```

48.

```
def longest_palindromic_substring(s):
    if not s:
        return ""
```

```
    start, max_length = 0, 1

    def expand_around_center(left, right):
        nonlocal start, max_length
        while left >= 0 and right < len(s) and s[left] == s[right]:
            if right - left + 1 > max_length:
                start, max_length = left, right - left + 1
            left -= 1
            right += 1

    for i in range(len(s)):
        expand_around_center(i, i)       # Odd-length palindromes
        expand_around_center(i, i + 1)   # Even-length palindromes

    return s[start:start + max_length]

# Example Input
s = "babad"

# Example Output
print(longest_palindromic_substring(s))
```

```
bab
```

49.

```
from itertools import permutations

def traveling_salesman(graph):
    cities = list(graph.keys())
    min_path = float('inf')
    best_route = []

    for perm in permutations(cities):
        current_distance = 0
        valid_path = True

        for i in range(len(perm) - 1):
            if perm[i + 1] in graph[perm[i]]:
                current_distance += graph[perm[i]][perm[i + 1]]
            else:
                valid_path = False
                break

        if valid_path and perm[-1] in graph[perm[0]]:
            current_distance += graph[perm[-1]][perm[0]]

            if current_distance < min_path:
                min_path = current_distance
                best_route = perm

    return best_route, min_path

# Example Input
graph = {
    'A': {'B': 10, 'C': 15, 'D': 20},
    'B': {'A': 10, 'C': 35, 'D': 25},
    'C': {'A': 15, 'B': 35, 'D': 30},
    'D': {'A': 20, 'B': 25, 'C': 30}
}

# Example Output
route, distance = traveling_salesman(graph)
print("Best Route:", route)
print("Total Distance:", distance)
```

```
Best Route: ('A', 'B', 'D', 'C')
Total Distance: 80
```

50.

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
```

```
    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)  # Since the graph is undirected

    def is_cyclic_util(self, v, visited, parent):
        visited[v] = True

        for neighbor in self.graph[v]:
            if not visited[neighbor]:
                if self.is_cyclic_util(neighbor, visited, v):
                    return True
            elif parent != neighbor:
                return True

        return False

    def is_cyclic(self):
        visited = {node: False for node in self.graph}

        for node in self.graph:
            if not visited[node]:
                if self.is_cyclic_util(node, visited, -1):
                    return True

        return False

# Example Input
g = Graph()
g.add_edge(0, 1)
g.add_edge(1, 2)
g.add_edge(2, 0)  # This creates a cycle

# Example Output
print(g.is_cyclic())  # Output: True
```

```
True
```

51.

```
def longest_unique_substring(s):
    char_index = {}
    left = 0
    max_length = 0

    for right in range(len(s)):
        if s[right] in char_index and char_index[s[right]] >= left:
            left = char_index[s[right]] + 1  # Move left pointer

        char_index[s[right]] = right  # Store/update last index of character
        max_length = max(max_length, right - left + 1)

    return max_length

# Example Input
s = "abcabcbb"

# Example Output
print(longest_unique_substring(s))
```

```
3
```