```python
#47.Count Inversions
def merge_and_count(arr, temp_arr, left, mid, right):
    # Initialize pointers and inversion count
    i, j, k = left, mid + 1, left
    inv_count = 0

    # Merge process with inversion count
    while i <= mid and j <= right:
        if arr[i] <= arr[j]:  # No inversion
            temp_arr[k] = arr[i]
            i += 1
        else:  # Inversion occurs
            temp_arr[k] = arr[j]
            inv_count += (mid - i + 1)  # Count inversions
            j += 1
        k += 1

    # Copy remaining elements from left subarray
    while i <= mid:
        temp_arr[k] = arr[i]
        i += 1
        k += 1

    # Copy remaining elements from right subarray
    while j <= right:
        temp_arr[k] = arr[j]  # Fixed incorrect addition operator
        j += 1
        k += 1

    # Copy sorted elements back to original array
    for i in range(left, right + 1):
        arr[i] = temp_arr[i]

    return inv_count

def merge_sort(arr, temp_arr, left, right):
    inv_count = 0
    if left < right:
        mid = (left + right) // 2

        # Recursively count inversions in left and right subarrays
        inv_count += merge_sort(arr, temp_arr, left, mid)
        inv_count += merge_sort(arr, temp_arr, mid + 1, right)

        # Count split inversions and merge
        inv_count += merge_and_count(arr, temp_arr, left, mid, right)

    return inv_count
```

```
def count_inversions(arr):
    temp_arr = [0] * len(arr)  # Temporary array for merging
    return merge_sort(arr, temp_arr, 0, len(arr) - 1)

# Example array
arr = [4, 45, 54, 36]
print("Number of inversions:", count_inversions(arr))
```

```
Number of inversions: 2
```

```
#48.Find the Longest Palndromic substring
def longest_palindromic_substring(s):
    # Helper function to expand around the center
    def expand_around_center(left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return s[left + 1:right]  # Return the palindromic substring

    longest = ""

    # Iterate over each character in the string
    for i in range(len(s)):
        # Find the longest odd-length palindrome centered at i
        odd_palindrome = expand_around_center(i, i)
        # Find the longest even-length palindrome centered between i
and i+1
        even_palindrome = expand_around_center(i, i + 1)
        # Update the longest palindrome found
        longest = max(longest, odd_palindrome, even_palindrome,
key=len)

    return longest

# Test case
print(longest_palindromic_substring("mainstream"))
```

```
m
```

```
#49.Travellling Salsemen problem(TSP)
from itertools import permutations

def tsp(graph, start):
    n = len(graph)

    # Get all vertices except the start node
    vertices = [i for i in range(n) if i != start]

    min_path = float("inf")

    # Generate all permutations of the remaining cities
```

```
    for perm in permutations(vertices):
        current_path_weight = 0
        k = start

        # Calculate path cost for current permutation
        for j in perm:
            current_path_weight += graph[k][j]
            k = j

        # Complete the cycle by returning to the start node
        current_path_weight += graph[k][start]

        # Update the minimum path cost
        min_path = min(min_path, current_path_weight)

    return min_path

# Example adjacency matrix (graph)
graph = [
    [0, 6, 7, 8],
    [7, 0, 43, 88],
    [75, 74, 0, 86],
    [65, 46, 76, 0]
]

# Start node set to 1
print("Shortest TSP route cost:", tsp(graph, 1))
```

```
Shortest TSP route cost: 146
```

```
#50.Graph Cycle Detection
def dfs(graph, v, visited, parent):
    visited[v] = True  # Mark the node as visited

    for neighbor in graph[v]:
        if not visited[neighbor]:  # If neighbor not visited, continue
DFS
            if dfs(graph, neighbor, visited, v):
                return True
        elif neighbor != parent:  # If visited and not the parent,
cycle detected
            return True

    return False  # No cycle found

def contains_cycle(graph):
    visited = {node: False for node in graph}  # Initialize visited
dictionary

    for node in graph:
```

```
        if not visited[node]:  # Start DFS for each unvisited
component
            if dfs(graph, node, visited, -1):
                return True

    return False  # No cycles detected

# Example adjacency list representation of an undirected graph
graph = { 0: [1, 2],  1: [0, 3],  2: [0, 3], 3: [1, 2] } # Fixed
missing bidirectional connection

print("Cycle present:", contains_cycle(graph))
```

```
Cycle present: True
```

```
#51.Longest substring without repeating characters
def longest_unique_substring(s):
    char_map = {}  # Dictionary to store the last seen index of
characters
    left = max_length = 0  # Left pointer and maximum length tracker

    for right, char in enumerate(s):  # Iterate through the string
with index
        if char in char_map and char_map[char] >= left:
            # Move the left pointer to avoid repeating characters
            left = char_map[char] + 1

        # Update the last seen index of the current character
        char_map[char] = right

        # Update max_length to track the longest substring
        max_length = max(max_length, right - left + 1)

    return max_length  # Return the length of the longest substring
without repeating characters

# Test case
print(longest_unique_substring("helloworld"))  # Expected output: 5
("world")
```

```
5
```

```
#52.Generate All valid parameters combinations
def generate_parentheses(n):
    def backtrack(s, open_count, close_count):
        # Base case: If the string reaches the maximum length (2 * n),
add to result
        if len(s) == 2 * n:
            result.append(s)
            return
```

```
        # If open brackets are still available, add '(' and recurse
        if open_count < n:
            backtrack(s + "(", open_count + 1, close_count)

        # If there are more open brackets than closed, add ')' and
recurse
        if close_count < open_count:
            backtrack(s + ")", open_count, close_count + 1)

    result = []  # Store valid combinations
    backtrack("", 0, 0)  # Start backtracking with an empty string
    return result

# Example usage
n = 4
print("Valid Parenthesis combinations:", generate_parentheses(n))
```

```
Valid Parenthesis combinations: ['(((())))', '((()()))', '((())())',
'((()))()', '(()(()))', '(()()())', '(()())()', '(())(())', '(())()
()', '()((()))', '()(()())', '()(())()', '()()(())', '()()()()']
```

```
#53.Zigzag level Order Traversal of a Binary Tree
from collections import deque

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

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

    result = []  # Stores the final zigzag level order traversal
    queue = deque([root])  # Initialize queue with root node
    left_to_right = True  # Flag to track traversal direction

    while queue:
        level_size = len(queue)  # Number of nodes at the current
level
        level = deque()  # Deque to store nodes in correct order

        for _ in range(level_size):
            node = queue.popleft()  # Pop node from the queue

            # Append based on traversal direction
            if left_to_right:
                level.append(node.val)
            else:
```

```
                level.appendleft(node.val)

            # Add child nodes to queue for next level
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(list(level))  # Convert deque to list and add to
result
        left_to_right = not left_to_right  # Flip the traversal
direction

    return result

# Constructing the binary tree for testing
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)

# Output the zigzag traversal
print("Zigzag Level Order Traversal:", zigzag_level_order(root))
```

```
Zigzag Level Order Traversal: [[1], [3, 2], [4, 5, 6, 7]]
```

```
#54.Palindrome Partitioning
def palindrome_partitioning(s):
    # Helper function to check if a substring is a palindrome
    def is_palindrome(sub):
        return sub == sub[::-1]

    # Backtracking function to generate palindrome partitions
    def backtrack(start, path):
        # If we reach the end of the string, add the current partition
to the result
        if start == len(s):
            result.append(path[:])  # Copy the current path
            return

        # Explore all possible partitions
        for end in range(start + 1, len(s) + 1):
            if is_palindrome(s[start:end]):  # Check if the substring
is a palindrome
                backtrack(end, path + [s[start:end]])  # Recur with
the next partition
```

```
    result = []
    backtrack(0, [])  # Start backtracking from index 0
    return result

# Test case
s = "hello"
print("Palindrome Partitions:", palindrome_partitioning(s))
```

```
Palindrome Partitions: [['h', 'e', 'l', 'l', 'o'], ['h', 'e', 'll',
'o']]
```

```
#7.Personal Budget advisor (Project)
class BudgetAdvisor:
    def __init__(self, income):
        self.income = income  # Store total income
        self.expenses = {}  # Dictionary to track expenses by category

    def add_expense(self, category, amount):
        """Adds or updates an expense category with the given
amount."""
        if category in self.expenses:
            self.expenses[category] += amount
        else:
            self.expenses[category] = amount

    def get_summary(self):
        """Prints a summary of income, expenses, savings, and category
breakdown."""
        total_expenses = sum(self.expenses.values())  # Calculate
total expenses
        savings = self.income - total_expenses  # Calculate remaining
savings

        print(f"Total Income: ${self.income}")
        print(f"Total Expenses: ${total_expenses}")
        print(f"Savings: ${savings}")

        print("\nExpense Breakdown:")
        for category, amount in self.expenses.items():
            percentage = (amount / self.income) * 100
            print(f"{category}: ${amount} ({percentage:.2f}%)")

        # Warning for overspending
        if savings < 0:
            print("\n⚠ Warning: You are overspending! Consider
reducing expenses.")

# Example usage
advisor = BudgetAdvisor(3000)
advisor.add_expense("Rent", 1200)
```

```
advisor.add_expense("Food", 500)
advisor.add_expense("Entertainment", 300)  # Fixed typo
advisor.get_summary()
```

```
Total Income: $3000
Total Expenses: $2000
Savings: $1000

Expense Breakdown:
Rent: $1200 (40.00%)
Food: $500 (16.67%)
Entertainment: $300 (10.00%)
```