PYTHON TASK - 7

47. Count Inversions

Objective: Count the number of inversions in an array, where an inversion is when for a[i]>a[j] for i<j

Code:

```
# Function to count inversions using Merge Sort

def merge_and_count(arr, temp_arr, left, mid, right):

    i = left    # Left subarray index

    j = mid + 1 # Right subarray index

    k = left    # Merged subarray index

    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, left, right):
    if left >= right:
        return 0
    mid = (left + right) // 2
    temp_arr = arr.copy()
    left_count = count_inversions(arr, left, mid)
    right_count = count_inversions(arr, mid + 1, right)
    merge_count = merge_and_count(arr, temp_arr, left, mid, right)
    return left_count + right_count + merge_count
arr = list(map(int, input("Enter numbers separated by spaces: ").split()))
print(f"Number of inversions: {count_inversions(arr, 0, len(arr) - 1)}")
```

Explanation:

- Uses a modified merge sort to count inversions while sorting the array.
- Each inversion is counted when merging two halves of the array.

---

48. Find the Longest Palindromic Substring

Objective: Find the longest palindromic substring in a given string.

Code:

```
def longest_palindrome(s):
    if len(s) == 0:
        return ""
    start, max_length = 0, 1
    for i in range(len(s)):
        for j in range(i, len(s)):
            if s[i:j+1] == s[i:j+1][::-1] and (j - i + 1) > max_length:
                start, max_length = i, j - i + 1
    return s[start:start + max_length]
s = input("Enter a string: ")
print(f"Longest palindromic substring: {longest_palindrome(s)}")
```

Explanation:

- Uses brute force to check all substrings and identify the longest palindrome.
- Could be optimized using dynamic programming or the expand-around-center approach.

---

49. Traveling Salesman Problem (TSP)

Objective: Find the shortest possible route that visits each city once and returns to the origin city.

Code:

```
from itertools import permutations

def tsp(graph, start):
    vertices = list(graph.keys())
    vertices.remove(start)
    min_path = float('inf')
    for perm in permutations(vertices):
        current_path_weight = 0
        k = start
        for j in perm:
            current_path_weight += graph[k][j]
            k = j
        current_path_weight += graph[k][start]
        min_path = min(min_path, current_path_weight)
    return min_path
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}
}
print(f"Shortest TSP path cost: {tsp(graph, 'A')}")
```

Explanation:

- Uses brute-force permutation to find the shortest cycle.
- Can be optimized using dynamic programming (Held-Karp algorithm).

---

50. Graph Cycle Detection

Objective: Detect whether a graph contains a cycle.

Code:

```
def dfs_cycle(graph, node, visited, parent):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            if dfs_cycle(graph, neighbor, visited, node):
                return True
        elif neighbor != parent:
            return True
    return False
def has_cycle(graph):
    visited = {node: False for node in graph}
    for node in graph:
        if not visited[node]:
            if dfs_cycle(graph, node, visited, None):
                return True
    return False
```

```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D'],
    'D': ['B', 'C']
}
print(f"Graph contains cycle: {has_cycle(graph)}")
```

Explanation:

- Uses DFS with a recursion stack to detect cycles in an undirected graph.
- A back edge indicates a cycle.

---

51. Longest Substring Without Repeating Characters

Objective: Given a string, find the length of the longest substring without repeating characters.

Code:

```
def length_of_longest_substring(s):
    char_index = {}
    left = 0
    max_length = 0
    for right, char in enumerate(s):
        if char in char_index and char_index[char] >= left:
```

```
            left = char_index[char] + 1
        char_index[char] = right
        max_length = max(max_length, right - left + 1)
    return max_length
s = input("Enter a string: ")
print(f"Length of longest substring without repeating characters: {length_of_longest_substring(s)}")
```

Explanation:

- Uses a sliding window with a dictionary to track characters.
- Adjusts the left boundary whenever a duplicate character is encountered.

---

52. Find All Valid Parentheses Combinations

Objective: Generate all possible valid combinations of parentheses.

Code:

```
def generate_parentheses(n, open_count=0, close_count=0, current_str="", result=None):
    if result is None:
        result = []
    if open_count == n and close_count == n:
        result.append(current_str)
        return result
    if open_count < n:
```

```
        generate_parentheses(n, open_count + 1, close_count, current_str + "(", result)
    if close_count < open_count:
        generate_parentheses(n, open_count, close_count + 1, current_str + ")", result)
    return result
n = int(input("Enter number of pairs of parentheses: "))
print("Valid Parentheses Combinations:", generate_parentheses(n))
```

Explanation:

- Uses recursion and backtracking to generate valid combinations.
- Ensures that an opening bracket always precedes a closing one.

---

53. Zigzag Level Order Traversal of Binary Tree

Objective: Traverse a binary tree in a zigzag level order.

Code:

```
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, left_to_right = [], deque([root]), True
    while queue:
        level_size = len(queue)
        level_nodes = deque()
        for _ in range(level_size):
            node = queue.popleft()
            if left_to_right:
                level_nodes.append(node.val)
            else:
                level_nodes.appendleft(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(list(level_nodes))
        left_to_right = not left_to_right
    return result
#Example usage:
#root = TreeNode(1, TreeNode(2), TreeNode(3))
#print(zigzag_level_order(root))
```

Explanation:

- Uses a deque to efficiently add elements at both ends.
- Alternates between left-to-right and right-to-left traversal at each level.

---

54. Palindrome Partitioning

Objective: Partition a string such that every substring is a palindrome.

Code:

```
def is_palindrome(s):
    return s == s[::-1]
def partition_helper(s, start, path, result):
    if start == len(s):
        result.append(path[:])
        return
    for end in range(start + 1, len(s) + 1):
        if is_palindrome(s[start:end]):
            path.append(s[start:end])
            partition_helper(s, end, path, result)
            path.pop()
def palindrome_partition(s):
    result = []
    partition_helper(s, 0, [], result)
    return result
s = input("Enter a string: ")
print("Palindrome Partitions:", palindrome_partition(s))
```

Explanation:

- Uses backtracking to explore all possible partitions.
- Checks each substring to determine if it is a palindrome.

---

7. Personal Budget Advisor

Objective: Build a program to track expenses and income, analyze spending patterns, and provide suggestions for saving money.

Code:

```
def add_transaction(transactions, category, amount, transaction_type):
    transactions.append({"category": category, "amount": amount, "type": transaction_type})
def calculate_summary(transactions):
    summary = {}
    for transaction in transactions:
        category = transaction["category"]
        amount = transaction["amount"]
        if transaction["type"] == "expense":
            summary[category] = summary.get(category, 0) + amount
    return summary
def provide_suggestions(income, expenses):
    savings = income - sum(expenses.values())
    print(f"Total Income: {income}")
    print(f"Total Expenses: {sum(expenses.values())}")
    print(f"Savings: {savings}")
    if savings < 0:
        print("Warning: You are overspending! Try reducing expenses in high-spending categories.")
    else:
        print("Good job! Consider investing or saving more.")
def main():
    transactions = []
    income = float(input("Enter your monthly income: "))
```

```
    while True:
        action = input("Enter transaction (category amount type[expense/income]) or 'done': ")
        if action.lower() == 'done':
            break
        try:
            category, amount, transaction_type = action.split()
            amount = float(amount)
            if transaction_type not in ["expense", "income"]:
                raise ValueError("Transaction type must be 'expense' or 'income'")
            add_transaction(transactions, category, amount, transaction_type)
        except ValueError as e:
            print(f"Invalid input: {e}")
    expenses = calculate_summary(transactions)
    provide_suggestions(income, expenses)
if __name__ == "__main__":
    main()
```

Explanation:

- Users can input income and transactions (expenses or income).
- The program tracks expenses by category and provides financial advice.
- Basic validation ensures correct input format.
