

#Task 47

```
def merge_and_count(arr, temp, left, mid, right):
    i = left        # Left subarray index
    j = mid + 1     # Right subarray index
    k = left        # Merged subarray index
    inv_count = 0

    # Merge two sorted halves
    while i <= mid and j <= right:
        if arr[i] <= arr[j]:
            temp[k] = arr[i]
            i += 1
        else:
            temp[k] = arr[j]
            inv_count += (mid - i + 1) # Count inversions
            j += 1
        k += 1

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

    # Copy remaining elements of right subarray
    while j <= right:
        temp[k] = arr[j]
        j += 1
        k += 1

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

    return inv_count

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


        inv_count += merge_sort_and_count(arr, temp, left, mid)
        inv_count += merge_sort_and_count(arr, temp, mid + 1, right)
        inv_count += merge_and_count(arr, temp, left, mid, right)

    return inv_count

def count_inversions(arr):
    temp = arr.copy()
    return merge_sort_and_count(arr, temp, 0, len(arr) - 1)

# Example Usage
arr = [8, 4, 2, 1]
```

```
print("Number of Inversions:", count_inversions(arr))
```

 Number of Inversions: 6

```
#task 48
```

```
def expand_around_center(s, 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 longest palindrome found

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

    longest = ""
    for i in range(len(s)):
        # Odd-length palindrome
        odd_palindrome = expand_around_center(s, i, i)
        # Even-length palindrome
        even_palindrome = expand_around_center(s, i, i+1)

        # Update longest palindrome found
        longest = max(longest, odd_palindrome, even_palindrome, key=len)

    return longest

# Example Usage
s = "babad"
print("Longest Palindromic Substring:", longest_palindromic_substring(s))
```

 Longest Palindromic Substring: bab

```
#Task 49
```

```
from itertools import permutations

def tsp_bruteforce(graph):
    n = len(graph)
    min_path = float("inf")
    best_route = None

    cities = list(range(1, n)) # Exclude the starting city (0)

    for perm in permutations(cities):
        current_path = 0
        prev = 0 # Start from city 0
        for city in perm:
            current_path += graph[prev][city]
            prev = city
        current_path += graph[prev][0] # Return to start

        if current_path < min_path:
```

```

        min_path = current_path
        best_route = (0,) + perm + (0,)

    return best_route, min_path

# Example Graph (Distance Matrix)
graph = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

route, cost = tsp_bruteforce(graph)
print("Best Route:", route)
print("Minimum Cost:", cost)

```

➡ Best Route: (0, 1, 3, 2, 0)  
Minimum Cost: 80

```

from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u) # Since it's an undirected graph

    def is_cyclic_util(self, v, visited, parent):
        visited[v] = True
        for neighbor in self.graph[v]:
            if not visited[neighbor]: # If the neighbor is not visited, recurse
                if self.is_cyclic_util(neighbor, visited, v):
                    return True
            elif neighbor != parent: # If the neighbor is visited and not parent, cycle
                return True
        return False

    def is_cyclic(self):
        visited = [False] * self.V
        for i in range(self.V):
            if not visited[i]: # Check all components of the graph
                if self.is_cyclic_util(i, visited, -1):
                    return True
        return False

# Example usage:
g = Graph(4)
g.add_edge(0, 1)
g.add_edge(1, 2)

```

```

g.add_edge(2, 0)
g.add_edge(2, 3)

print(g.is_cyclic()) # Output: True (Cycle exists)

```

→ True

#Task 50

```

from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list) # Adjacency list
        self.V = vertices # Number of vertices

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u) # Since it's an undirected graph

    def is_cyclic_util(self, v, visited, parent):
        visited[v] = True # Mark the current node as visited

        for neighbor in self.graph[v]:
            if not visited[neighbor]: # If the neighbor is not visited, recurse
                if self.is_cyclic_util(neighbor, visited, v):
                    return True
            elif neighbor != parent: # If visited and not parent, cycle exists
                return True

        return False

    def is_cyclic(self):
        visited = [False] * self.V

        for i in range(self.V): # Check all components of the graph
            if not visited[i]:
                if self.is_cyclic_util(i, visited, -1):
                    return True

        return False

# Example usage:
g = Graph(5)
g.add_edge(0, 1)
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)
print(g.is_cyclic()) # Output: False (No cycle)

g.add_edge(4, 1) # Adding a cycle
print(g.is_cyclic()) # Output: True (Cycle exists)

```

→ False  
True

#Task 51

```
def length_of_longest_substring(s):
    char_set = set() # Store unique characters in the window
    left = 0 # Left pointer
    max_length = 0

    for right in range(len(s)):
        while s[right] in char_set: # If duplicate found, shrink window from left
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right]) # Add new character to the set
        max_length = max(max_length, right - left + 1) # Update max length

    return max_length

# Example usage:
s = "abcabcbb"
print(length_of_longest_substring(s)) # Output: 3 ("abc")

s = "bbbbbb"
print(length_of_longest_substring(s)) # Output: 1 ("b")

s = "pwwkew"
print(length_of_longest_substring(s)) # Output: 3 ("wke")
```

→ 3  
1  
3

#Task 52

```
def generate_parentheses(n):
    result = []

    def backtrack(current, open_count, close_count):
        if len(current) == 2 * n: # Base case: Valid combination found
            result.append(current)
            return

        if open_count < n: # Add '(' if possible
            backtrack(current + "(", open_count + 1, close_count)

        if close_count < open_count: # Add ')' if valid
            backtrack(current + ")", open_count, close_count + 1)

    backtrack("", 0, 0) # Start with an empty string
    return result

# Example usage:
n = 3
```

```
print(generate_parentheses(n))
```

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

#Task 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 = deque([root])
```

```
    left_to_right = True # Track direction
```

```
    while queue:
```

```
        level_size = len(queue)
```

```
        level_nodes = deque() # Store current level nodes
```

```
        for _ in range(level_size):
```

```
            node = queue.popleft()
```

```
            if left_to_right:
```

```
                level_nodes.append(node.val) # Normal order
```

```
            else:
```

```
                level_nodes.appendleft(node.val) # Reverse order
```

```
            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 # Toggle direction
```

```
    return result
```

```
# Example usage:
```

```
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)
```

```
print(zigzag_level_order(root))
```

→ `[[1], [3, 2], [4, 5, 6, 7]]`

#Task 54

```
def partition(s):
    result = []

    def is_palindrome(sub):
        return sub == sub[::-1]

    def backtrack(start, path):
        if start == len(s): # If end of string is reached, add partition
            result.append(path[:])
            return

        for end in range(start + 1, len(s) + 1):
            if is_palindrome(s[start:end]): # Check palindrome
                backtrack(end, path + [s[start:end]]) # Recur with new substring

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

# Example usage:
s = "aab"
print(partition(s))
```

→ `[['a', 'a', 'b'], ['aa', 'b']]`

#Task 07

import os

```
class BudgetAdvisor:
    def __init__(self, filename="budget_data.txt"):
        self.filename = filename
        self.data = {"income": 0, "expenses": {}}
        self.load_data()

    def load_data(self):
        """Load data from a file."""
        if not os.path.exists(self.filename):
            return # No data yet
        with open(self.filename, "r") as file:
            lines = file.readlines()
            for line in lines:
                key, value = line.strip().split(": ")
                if key == "income":
                    self.data["income"] = float(value)
                else:
                    category, amount = key.split(", ")
                    self.data["expenses"][category] = float(amount)
```

```

def save_data(self):
    """Save data to a file."""
    with open(self.filename, "w") as file:
        file.write(f"income: {self.data['income']}\n")
        for category, amount in self.data["expenses"].items():
            file.write(f"{category}, {amount}: {amount}\n")

def add_income(self, amount):
    """Add income to the budget."""
    try:
        amount = float(amount)
        self.data["income"] += amount
        self.save_data()
        print(f"Income updated: ${self.data['income']:.2f}")
    except ValueError:
        print("Invalid input! Please enter a numerical value.")

def add_expense(self, category, amount):
    """Add an expense to the budget."""
    try:
        amount = float(amount)
        if category in self.data["expenses"]:
            self.data["expenses"][category] += amount
        else:
            self.data["expenses"][category] = amount
        self.save_data()
        print(f"Expense added: {category} - ${amount:.2f}")
    except ValueError:
        print("Invalid input! Please enter a numerical value.")

def analyze_budget(self):
    """Analyze spending and give recommendations."""
    total_expenses = sum(self.data["expenses"].values())
    savings = self.data["income"] - total_expenses

    print("\n💎 **Budget Summary** 💎")
    print(f"Total Income: ${self.data['income']:.2f}")
    print(f"Total Expenses: ${total_expenses:.2f}")
    print(f"Savings: ${savings:.2f}\n")

    if total_expenses > self.data["income"]:
        print("⚠️ You are overspending! Consider reducing expenses.")
    elif savings < self.data["income"] * 0.2:
        print("💡 Try to save at least 20% of your income for better financial security")

    print("\n💎 **Spending Breakdown** 💎")
    for category, amount in self.data["expenses"].items():
        percentage = (amount / total_expenses) * 100 if total_expenses else 0
        print(f"{category}: ${amount:.2f} ({percentage:.2f}%)")
        if percentage > 30:
            print(f"⚠️ You are spending too much on {category}! Consider reducing.")

def menu(self):
    """Display the menu and handle user input."""

```



```

while True:
    print("\n◆ **Personal Budget Advisor** ◆")
    print("1. Add Income")
    print("2. Add Expense")
    print("3. View Budget Analysis")
    print("4. Exit")

    choice = input("Enter your choice: ")
    if choice == "1":
        amount = input("Enter income amount: ")
        self.add_income(amount)
    elif choice == "2":
        category = input("Enter expense category (e.g., Food, Rent): ")
        amount = input("Enter expense amount: ")
        self.add_expense(category, amount)
    elif choice == "3":
        self.analyze_budget()
    elif choice == "4":
        print("Goodbye! 🙋")
        break
    else:
        print("Invalid choice. Please try again.")

```

```

# Run the program
advisor = BudgetAdvisor()
advisor.menu()

```



◆ \*\*Personal Budget Advisor\*\* ◆

1. Add Income
2. Add Expense
3. View Budget Analysis
4. Exit

Enter your choice: 1

Enter income amount: 3000

Income updated: \$3000.00

◆ \*\*Personal Budget Advisor\*\* ◆

1. Add Income
2. Add Expense
3. View Budget Analysis
4. Exit

Enter your choice: 3

◆ \*\*Budget Summary\*\* ◆

Total Income: \$3000.00

Total Expenses: \$0.00

Savings: \$3000.00

◆ \*\*Spending Breakdown\*\* ◆

◆ \*\*Personal Budget Advisor\*\* ◆

1. Add Income
2. Add Expense
3. View Budget Analysis

```
4. Exit  
Enter your choice: 4  
Goodbye! 🙋
```

Start coding or generate with AI.