Q1

Program to print multiplication table for given no. b) Program to check whether the given no is prime or not. c) Program to find factorial of the given no and similar program

```
# Program to print the multiplication table of a given number
# Input from user

num = int(input("Enter a number: "))
# Printing multiplication table
print(f"Multiplication Table for {num}")
for i in range(1, 11):
    print(f"{num} x {i} = {num * i}")
```

```
Multiplication Table for 4
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36
4 x 10 = 40
```

```
# Function to check for prime number
def is_prime(num):
    if num <= 1:  # Numbers less than or equal to 1 are not prime
        return False
    for i in range(2, int(num ** 0.5) + 1):  # Check divisibility from 2 to sqrt(num)
        if num % i == 0:
            return False
    return True
# Input from user
num = int(input("Enter a number: "))
# Checking and printing the result
if is_prime(num):
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
```

```
58 is not a prime number.
```

Write a program to implement List Operations Nested list, Length, Concatenation, Membership ,Iteration ,Indexing and Slicing List Methods Add, Extend & Delete.

```
# 1. Nested List
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print("Nested List:", nested_list)
```

```
Nested List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
# 2. Length of a List
list1 = [1, 2, 3, 4, 5]
print("Length of list1:", len(list1))
```

```
Length of list1: 5
```

```
# 3. Concatenation of two lists
list2 = [6, 7, 8]
concatenated_list = list1 + list2
print("Concatenated List:", concatenated_list)
```

```
Concatenated List: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
# 4. Membership (Check if an item exists in a list)
print("Is 3 in list1?:", 3 in list1)
```

```
Is 3 in list1?: True
```

```
# 5. Iteration through a list
print("Iterating through list1:")
for item in list1:
    print(item, end=" ")
print()
```

```
Iterating through list1:
1 2 3 4 5
```

```
# 6. Indexing
print("Element at index 2 in list1:", list1[2])
```

```
Element at index 2 in list1: 3
```

```
# 7. Slicing (extracting parts of a list)
sliced_list = list1[1:4]  # From index 1 to 3 (4th index excluded)
print("Sliced List (index 1 to 3):", sliced_list)
```

```
Sliced List (index 1 to 3): [2, 3, 4]
```

```
# 1. Adding an element (using append)
list1.append(9)
print("List after appending 9:", list1)
```

```
List after appending 9: [1, 2, 3, 4, 5, 9]
```

```
# 2. Extending a list (adding multiple elements)
list1.extend([10, 11, 12])
print("List after extending with [10, 11, 12]:", list1)
```

```
List after extending with [10, 11, 12]: [1, 2, 3, 4, 5, 9, 10, 11, 12]
```

```
list1.remove(9)  # Removes the first occurrence of 9
print("List after removing 9:", list1)
```

```
List after removing 9: [1, 2, 3, 4, 5, 10, 11, 12]
```

Write a program to Illustrate Different Set Operations.

```
# Define two sets
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}
```

```
# 1. Union: Combines all elements from both sets (no duplicates)
union_set = set1.union(set2)
print("Union of set1 and set2:", union_set)
```

```
Union of set1 and set2: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
# 2. Intersection: Elements common to both sets
intersection_set = set1.intersection(set2)
print("Intersection of set1 and set2:", intersection_set)
```

```
Intersection of set1 and set2: {4, 5}
```

```
# 3. Difference: Elements in set1 but not in set2
difference_set = set1.difference(set2)
print("Difference of set1 - set2:", difference_set)
```

```
Difference of set1 - set2: {1, 2, 3}
```

```
# 4. Symmetric Difference: Elements in either set1 or set2, but not in both
symmetric_difference_set = set1.symmetric_difference(set2)
print("Symmetric Difference of set1 and set2:", symmetric_difference_set)
```

```
Symmetric Difference of set1 and set2: {1, 2, 3, 6, 7, 8}
```

```
# 5. Check if a set is a subset of another set
subset_check = {1, 2}.issubset(set1)
print("Is {1, 2} a subset of set1?:", subset_check)
```

```
Is {1, 2} a subset of set1?: True
```

```
# 6. Check if a set is a superset of another set
superset_check = set1.issuperset({1, 2})
print("Is set1 a superset of {1, 2}?:", superset_check)
```

```
Is set1 a superset of {1, 2}?: True
```

```
# 7. Adding an element to a set
set1.add(9)
print("Set1 after adding 9:", set1)
```

```
Set1 after adding 9: {1, 2, 3, 4, 5, 9}
```

```
# 8. Removing an element from a set
set1.remove(9)
print("Set1 after removing 9:", set1)
```

```
Set1 after removing 9: {1, 2, 3, 4, 5}
```

Write a program to implement Simple Chatbot

```
# Predefined responses for the chatbot
responses = {
    "hello": "Hello! How can I assist you today?",
    "hi": "Hi there! How can I help?",
    "how are you?": "I'm just a bot, but I'm doing great! How about you?",
    "what is your name?": "I'm a simple chatbot created to assist you.",
    "what can you do?": "I can answer basic questions and have small conversations. Try asking me something!",
    "bye": "Goodbye! Have a great day!"
}
def chatbot_response(user_input):
    # Convert user input to lowercase to make the chatbot case-insensitive
    user_input = user_input.lower()
    # Check if the user input matches any predefined questions
    if user_input in responses:
        return responses[user_input]
    else:
        # Fallback response for unrecognized input
        return "I'm sorry, I don't understand that. Could you please rephrase?"
def chatbot():
    print("Chatbot: Hello! I am your friendly chatbot. Type 'bye' to end the conversation.")
    while True:
        # Get input from the user
        user_input = input("You: ")
        # Exit the loop if the user says 'bye'
        if user_input.lower() == "bye":
            print("Chatbot:", responses["bye"])
            break
        # Get and print chatbot's response
        response = chatbot_response(user_input)
        print("Chatbot:", response)
# Run the chatbot
chatbot()
```

```
Chatbot: Hello! I am your friendly chatbot. Type 'bye' to end the conversation.
Chatbot: Hi there! How can I help?
Chatbot: I'm sorry, I don't understand that. Could you please rephrase?
Chatbot: I'm a simple chatbot created to assist you.
Chatbot: I can answer basic questions and have small conversations. Try asking me something!
Chatbot: Goodbye! Have a great day!
```

Write a program to implement Breadth First Search Traversal

```
from collections import deque
# BFS function to traverse the graph
def bfs(graph, start_node):
    # Initialize a set to keep track of visited nodes
    visited = set()
    # Create a queue for BFS and add the start node
    queue = deque([start_node])
    # Mark the start node as visited
    visited.add(start_node)
    # While there are nodes to process in the queue
    while queue:
        # Pop the front node from the queue
        node = queue.popleft()
        print(node, end=" ")  # Print the current node
        # Get all adjacent nodes of the popped node
        for neighbor in graph[node]:
            # If the neighbor hasn't been visited yet
            if neighbor not in visited:
                # Mark it as visited and add it to the queue
                visited.add(neighbor)
                queue.append(neighbor)
# Define the graph as an adjacency list (dictionary)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
# Run BFS starting from node 'A'
print("Breadth First Search (BFS) Traversal starting from node A:")
bfs(graph, 'A')
```

```
Breadth First Search (BFS) Traversal starting from node A:
A B C D E F
```

Write a program to implement Depth First Search Traversal

```
# DFS function to traverse the graph
def dfs(graph, node, visited):
    # Mark the node as visited
    visited.add(node)
    # Print the current node
    print(node, end=" ")
    # Recur for all the adjacent vertices of the current node
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
# Define the graph as an adjacency list (dictionary)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
# Run DFS starting from node 'A'
visited = set()  # A set to keep track of visited nodes
print("Depth First Search (DFS) Traversal starting from node A:")
dfs(graph, 'A', visited)
```

```
Depth First Search (DFS) Traversal starting from node A:
A B D E F C
```

Write a program to implement Water Jug Problem

```
from collections import deque
# Function to check if a state has been visited before
def is_visited(visited, state):
    return visited.get(state, False)
# Function to mark a state as visited
def mark_visited(visited, state):
    visited[state] = True
# Function to implement the BFS solution to the Water Jug Problem
def water_jug_bfs(jug1_capacity, jug2_capacity, target):
    # Queue to store the states (jug1, jug2) and the operations taken
    queue = deque()
    # Dictionary to keep track of visited states
    visited = {}
    # Initial state: both jugs are empty
    initial_state = (0, 0)
    queue.append((initial_state, []))
    mark_visited(visited, initial_state)
    while queue:
        # Get the current state and the list of operations to reach it
        (jug1, jug2), operations = queue.popleft()
        # If the target is reached in either jug, return the operations
        if jug1 == target or jug2 == target:
            return operations
        # Possible operations from the current state
        possible_operations = [
            ((jug1_capacity, jug2), "Fill Jug1"),  # Fill Jug1
            ((jug1, jug2_capacity), "Fill Jug2"),  # Fill Jug2
            ((0, jug2), "Empty Jug1"),  # Empty Jug1
            ((jug1, 0), "Empty Jug2"),  # Empty Jug2
            ((min(jug1 + jug2, jug1_capacity), max(0, jug2 - (jug1_capacity - jug1))), "Pour Jug2 to Jug1"),  # Pour Jug2 to Jug1
            ((max(0, jug1 - (jug2_capacity - jug2)), min(jug1 + jug2, jug2_capacity)), "Pour Jug1 to Jug2")   # Pour Jug1 to Jug2
        ]
        # Process each possible operation
        for new_state, operation in possible_operations:
            if not is_visited(visited, new_state):
                mark_visited(visited, new_state)
                queue.append((new_state, operations + [operation]))
    # If no solution is found, return None
    return None
# Driver code
def solve_water_jug_problem(jug1_capacity, jug2_capacity, target):
    result = water_jug_bfs(jug1_capacity, jug2_capacity, target)
    if result:
        print("Steps to achieve the target:")
        for step in result:
            print(step)
    else:
        print("No solution exists.")
# Example usage:
solve_water_jug_problem(4, 3, 2)
```

```
Steps to achieve the target:
Fill Jug2
Pour Jug2 to Jug1
Fill Jug2
Pour Jug2 to Jug1
```

Write a program to implement K Nearest Neighbor algorithm.

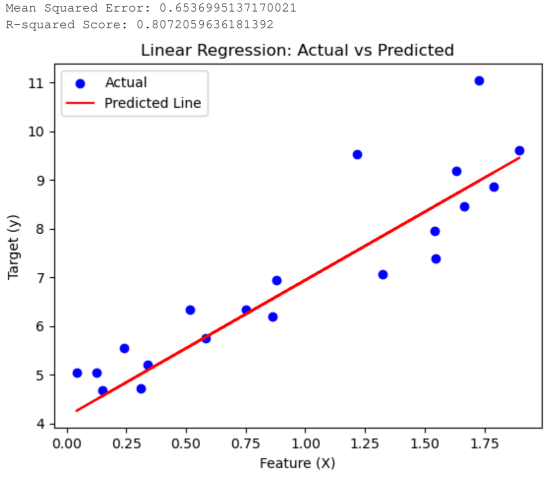
```
import numpy as np
from collections import Counter
 # Function to calculate Euclidean distance between two points
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))
      # Function to implement K-Nearest Neighbor algorithm
def k_nearest_neighbors(training_data, training_labels, test_point, k):
    # List to store distances and corresponding labels
    distances = []
    # Calculate distance from the test_point to all training data points
    for i in range(len(training_data)):
        distance = euclidean_distance(test_point, training_data[i])
        distances.append((distance, training_labels[i]))
    # Sort distances in ascending order and get the labels of the k nearest neighbors
    distances.sort(key=lambda x: x[0])
    k_nearest_labels = [label for _, label in distances[:k]]
    # Get the majority vote for classification
    most_common_label = Counter(k_nearest_labels).most_common(1)[0][0]
    return most_common_label
 # Driver code
if __name__ == "__main__":
    # Training data (features) - 2D array where each row is a data point
    training_data = np.array([
        [1, 2],
        [2, 3],
        [3, 1],
        [6, 5],
        [7, 7],
        [8, 6]
    ])
    # Labels (class labels corresponding to the training data)
    training_labels = np.array([0, 0, 0, 1, 1, 1])
    # Test point that we want to classify
    test_point = np.array([5, 5])
    # Set value of k (number of neighbors to consider)
    k = 3
    # Get the predicted label using K-NN
    predicted_label = k_nearest_neighbors(training_data, training_labels, test_point, k)
    print(f"The predicted label for test point {test_point} is: {predicted_label}")
```

```
The predicted label for test point [5 5] is: 1
```

Write a program to impliment regression algorithm

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
# Generating some example data (simulating a dataset)
# Features (X): Randomly generated values
# Target (y): A linear function of X with some noise
np.random.seed(42)
X = 2 * np.random.rand(100, 1)  # 100 random data points (feature)
y = 4 + 3 * X + np.random.randn(100, 1)  # y = 4 + 3X + noise
# Split the data into training and test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Create the Linear Regression model
model = LinearRegression()
# Train the model using the training data
model.fit(X_train, y_train)
# Predict the target values for the test data
y_pred = model.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)  # Mean Squared Error
r2 = r2_score(y_test, y_pred)  # R-squared score
# Display the results
print(f"Mean Squared Error: {mse}")
print(f"R-squared Score: {r2}")
# Plot the results: Actual vs Predicted
plt.scatter(X_test, y_test, color="blue", label="Actual")
plt.plot(X_test, y_pred, color="red", label="Predicted Line")
plt.xlabel("Feature (X)")
plt.ylabel("Target (y)")
plt.title("Linear Regression: Actual vs Predicted")
plt.legend()
plt.show()
```

```
Mean Squared Error: 0.6536995137170021
R-squared Score: 0.8072059636181392
```



Write a computer program to play tic-tac-toe game.

```
import random
 # Function to initialize the board
def initialize_board():
    return [' ' for _ in range(9)]  # A list of 9 spaces
 # Function to print the board
def print_board(board):
    print("\n")
    print(f" {board[0]} | {board[1]} | {board[2]} ")
    print("---|---|---")
    print(f" {board[3]} | {board[4]} | {board[5]} ")
    print("---|---|---")
    print(f" {board[6]} | {board[7]} | {board[8]} ")
    print("\n")
 # Function to check for a win
def check_winner(board, player):
    win_conditions = [
        (0, 1, 2), (3, 4, 5), (6, 7, 8),  # Rows
        (0, 3, 6), (1, 4, 7), (2, 5, 8),  # Columns
        (0, 4, 8), (2, 4, 6)               # Diagonals
    ]
    return any(all(board[i] == player for i in condition) for condition in win_conditions)
 # Function to check for a draw
def check_draw(board):
    return all(space != ' ' for space in board)
 # Function to get a list of available moves
def available_moves(board):
    return [i for i, space in enumerate(board) if space == ' ']
 # Function for the computer's move (random selection)
def computer_move(board):
    move = random.choice(available_moves(board))
    board[move] = 'O'
     # Function for the player's move
def player_move(board):
    while True:
        try:
            move = int(input("Enter your move (1-9): ")) - 1
            if move in available_moves(board):
                board[move] = 'X'
                break
            else:
                print("Invalid move. Try again.")
        except ValueError:
            print("Invalid input. Please enter a number between 1 and 9.")
 # Main game loop
def play_game():
    board = initialize_board()
    print_board(board)
    while True:
        # Player's move
        player_move(board)
        print_board(board)
        if check_winner(board, 'X'):
            print("Congratulations! You win!")
            break
        if check_draw(board):
            print("It's a draw!")
            break
        # Computer's move
        computer_move(board)
        print_board(board)
        if check_winner(board, 'O'):
            print("Computer wins! Better luck next time.")
            break
        if check_draw(board):
            print("It's a draw!")
            break
 # Start the game
if __name__ == "__main__":
    play_game()
```

```
   |   |
---|---|---
   |   |
---|---|---
   |   |



   |   |
---|---|---
 X |   |
---|---|---
   |   |



   |   |
---|---|---
 X |   |
---|---|---
   |   | O



   |   |
---|---|---
 X |   | X
---|---|---
   |   | O



   | O |
---|---|---
 X |   | X
---|---|---
   |   | O


Invalid move. Try again.

   | O | X
---|---|---
 X |   | X
---|---|---
   |   | O



   | O | X
---|---|---
 X |   | X
---|---|---
 O |   | O
```

In [ ]: