Q1.Write a Python program toimplement depth first search algorithm.Refer the following graphs as an input for the program.[Initial node=1,Goal node=8].

```
class Graph:
   def _init_(self):
      self.graph = {}
  def add_edge(self, u, v):
      if u not in self.graph:
         self.graph[u] = []
      self.graph[u].append(v)
 def dfs(self, current, goal, visited=None, path=None):
      if visited is None:
         visited = set()
      if path is None:
         path = []
  visited.add(current)
      path.append(current)

      if current == goal:
         print("Path found:”,path)
         return
         for neighbour in self.graph.get(current,[]):
          if neighbor not in visited:
            self.dfs(neighbor, goal, visited, path.copy())
# Example usage:
graph = Graph()
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(2, 4)
graph.add_edge(2, 5)

graph.add_edge(3, 6)

graph.add_edge(3, 7)

graph.add_edge(4, 8)

graph.add_edge(5, 8)

graph.add_edge(6, 8)

graph.add_edge(7, 8)

start_node=1

goal_node=8

print(f”DFS from{start_node}to{goal_node}:”)

graph.dfs(start_node,goal_node)
```

Q2.Write a Python Program to implement simple chatbot.

```
def simple_chatbot(user_input):

   user_input = user_input.lower()

 if "hello" in user_input:

      return "Hi there! How can I help you?"

 elif "your name" in user_input:

      return "I'm a simple chatbot."

 elif "how are you" in user_input:

      return "I'm just a program, but thanks for asking!"

 elif "bye" in user_input: return "Goodbye! Have a great day."

 else:

      return "I'm not sure how to respond to that. Ask me something else."

# Main loop for interacting with the chatbot

while True:

 user_message = input("You:”)
 if user_message.lower() == "exit": print("Chatbot: Goodbye!")
  break
bot_response=simple_chatbot(user_message)
print(“Chatbot:”,bot_response)
```

Q3.Write a Python program to solve a tic tac toe problem.

```
def print_board(board):
    for row in board:
        print(" ".join(row))


def check_winner(board):
    # Check rows, columns, and diagonals for a win
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != ' ':
            return board[i][0]  # Check rows
        if board[0][i] == board[1][i] == board[2][i] != ' ':
            return board[0][i]  # Check columns
if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0]  # Check diagonal from top-left to bottom-right


    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2]  # Check diagonal from top-right to bottom-left


    return None  # No winner yet def is_board_full(board):
    for row in board:
        if ' ' in row:
            return False
```

```
 return True
def play_tic_tac_toe():
  board = [[' ' for _ in range(3)] for _ in range(3)]
  current_player = 'X'
 while True:
    print_board(board)
    row = int(input(f"Player {current_player}, enter the row (0, 1, or 2): "))
    col = int(input(f"Player {current_player}, enter the column (0, 1, or 2): "))
 if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == ' ':
      board[row][col] = current_player
      winner = check_winner(board)
 if winner:
        print_board(board)
        print(f"Player {winner} wins!")
        break
      elif is_board_full(board):
        print_board(board)
        print("It's a tie!")
        break
 current_player = 'O' if current_player == 'X' else 'X'
    else:
      print("Invalid move. Try again.")


if _name_ == "_main_":
  play_tic_tac_toe()
```

Q4.Write a Python program to solve a water juck problem. 2 jug with capacity 5 gallon and 7 gallon are given with unlimited water respectively.The target to achieve is 4 gallon ofwater insecond jug.

```
def water_jug_problem(capacity_jug1, capacity_jug2, target):
   jug1 = 0
   jug2 = 0
 while jug2 != target:
      print(f"Jug 1: {jug1} gallons, Jug 2: {jug2} gallons")
 # Fill Jug 2
      jug2 = capacity_jug2
 if jug2 == target:
         break
  # Pour water from Jug 2 to Jug 1
      pour = min(jug2, capacity_jug1 - jug1)
      jug2 = jug2 - pour
      jug1 = jug1 + pour
 if jug2 == target:
         break
 # Empty Jug 1
      jug1 = 0
 print(f"Target of {target} gallons reached in Jug 2.")
if _name_ == "_main_":
   capacity_jug1 = 5
  capacity_jug2=7
  target=4
water_jug_problem(capacity_jug1,capacity_jug2,target)
```

Q5.Write a Python Program to simulate a 4-Queens Problem.

```
def is_safe(board, row, col):
    # Check if there is a queen in the same row
    if any(board[row]):
        return False
 # Check if there is a queen in the same column
    if any(board[i][col] for i in range(len(board))):
        return False
 # Check if there is a queen in the same diagonal (upper left to lower right)
    if any(board[i][j] for i, j in zip(range(row, -1, -1), range(col, -1, -1))):
        return False
 # Check if there is a queen in the same diagonal (upper right to lower left)
    if any(board[i][j] for i, j in zip(range(row, -1, -1), range(col, len(board)))):
        return False
 return True
def solve_n_queens(n):
    def backtrack(row):
        if row == n:
            solutions.append([r[:] for r in board])
            return
for col in range(n):
            if is_safe(board, row, col):
                board[row][col] = 1
                backtrack(row + 1)
                board[row][col] = 0
 board = [[0] * n for _ in range(n)]
    solutions = []
 backtrack(0)
    return solutions


def print_solution(solution):
    for row in solution:
        print(" ".join("Q" if col else "." for col in row))
    print()
if _name_ == "_main_":
    n_queens_solutions = solve_n_queens(4)
 print(f"Total solutions for 4-Queens: {len(n_queens_solutions)}\n")
 for index, solution in enumerate(n_queens_solutions, start=1):
        print(f"Solution {index}:\n")
        print_solution(solution)
```

Q6.Write a Python Program to implement Tower of Hanoi using Python.

```
def tower_of_hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n - 1, source, auxiliary, target)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n - 1, auxiliary, target, source)


if _name_ == "_main_":
    num_disks = int(input("Enter the number of disks: "))
    tower_of_hanoi(num_disks, 'A', 'C', 'B’)
```

Q7.Write a Python Program for the following Cryptarithmetic problems.

GO+TO=OUT

```
from itertools import permutations
def is_valid_solution(mapping, word):
    return int("".join(str(mapping[ch]) for ch in word))
def solve_cryptarithmetic():
    for perm in permutations(range(10), 8):
        mapping = {'G': perm[0], 'O': perm[1], 'T': perm[2], 'U': perm[3]}
        go = is_valid_solution(mapping, 'GO')
        to = is_valid_solution(mapping, 'TO')
        out = is_valid_solution(mapping, 'OUT')
if go + to == out:
            print(f"Solution found: G={perm[0]}, O={perm[1]}, T={perm[2]}, U={perm[3]}")
            print(f"GO = {go}, TO = {to}, OUT = {out}")
            return
 print("No solution found.")
if _name_ == "_main_":
    solve_cryptarithmetic()
```

Q8.Write a Python Program to sort the sentence in alphabetical order.

```
def sort_sentence_alphabetically(sentence):
    words = sentence.split()
    sorted_words = sorted(words)
    sorted_sentence = ' '.join(sorted_words)
    return sorted_sentence
if _name_ == "_main_":
    input_sentence = input("Enter a sentence: ")
    result = sort_sentence_alphabetically(input_sentence)
  print("Sorted Sentence:", result)
```

Q9.Write a Python Program to implement a Breadth First Search algorithm.Refer the Following Graph as an input for the program.[initial node =1, Goal node=8].

```
from collections import deque
class Graph:
    def _init_(self):
        self.graph = {}
  def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)
 def bfs(self, start, goal):
        visited = set()while queue:
            current = queue.popleft()
            visited.add(current)


            if current == goal:
                print("Goal node reached!"
 return
 for neighbor in self.graph.get(current, []):
            if neighbor not in visited and neighbor not in queue:
              queue.append(neighbor)
 print("Goal node not reached.")
# Example usage:
graph = Graph()
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(2, 4)
graph.add_edge(2, 5)
graph.add_edge(3, 6)
graph.add_edge(3, 7)
graph.add_edge(4, 8)
graph.add_edge(5, 8)
graph.add_edge(6, 8)
graph.add_edge(7, 8) start_node = 1
goal_node = 8
print(f"BFS from {start_node} to {goal_node}:")
graph.bfs(start_node, goal_node
```

Q10.Write a Python Program to remove punctuations from the given string.

```
import string
def remove_punctuation(input_string):
  # Create a translation table with None for all punctuation characters
  translation_table = str.maketrans("", "", string.punctuation)
    # Use translate method to remove punctuations
  result_string = input_string.translate(translation_table)
```

```
return result_string

if _name_ == "_main_":

    input_string = input("Enter a string with punctuations: ")

    result = remove_punctuation(input_string)

    print("String without punctuations:", result)
```