1. Write a Program to Implement 8-Queens Problem using Python.

```
    def is_safe(board, row, col, n):
    # Check if there is a queen in the same row on the left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on the left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on the left side
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens_util(board, col, n):
    if col >= n:
        return True

    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1

            if solve_n_queens_util(board, col + 1, n):
                return True

            board[i][col] = 0

    return False

def solve_n_queens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]

    if not solve_n_queens_util(board, 0, n):
        print("Solution does not exist")
        return

    print_board(board)
```

<u>Output:</u>

```
"D:\Program Files\Python\Python310\python.exe" D:\Desktop\A.I\A.I\01.py
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0

Process finished with exit code 0
```

2. Write a Program to Implement Breadth First Search using Python.

```
from collections import deque
class Graph:
```

```
    def __init__(self):
        self.graph = {}

    def add_edge(self, vertex, neighbors):
        self.graph[vertex] = neighbors

   def bfs(self, start):
        visited = set()
        queue = deque([start])
        visited.add(start)

        while queue:
            current_vertex = queue.popleft()
            print(current_vertex, end=" ")

            for neighbor in self.graph[current_vertex]:
                if neighbor not in visited:
                    queue.append(neighbor)
                    visited.add(neighbor)

# Example usage:
if __name__ == "__main__":
    # Create a sample graph
    g = Graph()
    g.add_edge('A', ['B', 'C'])
    g.add_edge('B', ['A', 'D', 'E'])
    g.add_edge('C', ['A', 'F', 'G'])
    g.add_edge('D', ['B'])
    g.add_edge('E', ['B', 'H'])
    g.add_edge('F', ['C'])
    g.add_edge('G', ['C'])
    g.add_edge('H', ['E'])

    # Perform BFS starting from vertex 'A'
    print("Breadth-First Search:")
    g.bfs('A')
```

<u>Output:</u>

```
 "D:\Program Files\Python\Python310\python.exe" D:\Desktop\A.I\A.I\02.py
Breadth-First Search:
A B C D E F G H
Process finished with exit code 0
```

3. Write a Program to Implement Depth First Search using Python.

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, vertex, neighbors):
        self.graph[vertex] = neighbors

    def dfs(self, start, visited=None):
        if visited is None:
            visited = set()
```

```
        print(start, end=" ")
        visited.add(start)

        for neighbor in self.graph[start]:
            if neighbor not in visited:
                self.dfs(neighbor, visited)

# Example usage:
if __name__ == "__main__":
    # Create a sample graph
    g = Graph()
    g.add_edge('A', ['B', 'C'])
    g.add_edge('B', ['A', 'D', 'E'])
    g.add_edge('C', ['A', 'F', 'G'])
    g.add_edge('D', ['B'])
    g.add_edge('E', ['B', 'H'])
    g.add_edge('F', ['C'])
    g.add_edge('G', ['C'])
    g.add_edge('H', ['E'])

    # Perform DFS starting from vertex 'A'
    print("Depth-First Search:")
    g.dfs('A')
```

Output:

```
"D:\Program Files\Python\Python310\python.exe" D:\Desktop\A.I\A.I\03.py
Depth-First Search:
A B D E H C F G
Process finished with exit code 0
```

4. Write a Program to Implement a Tic-Tac-Toe game using Python.

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)
def check_winner(board, player):
    # Check rows and columns
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or all(board[j][i] == player for
j in range(3)):
            return True
 # Check diagonals
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for
i in range(3)):
        return True
    return False
def is_board_full(board):
    return all(board[i][j] != ' ' for i in range(3) for j in range(3))
def tic_tac_toe():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    current_player = 'X'
    while True:
        print_board(board)
        # Get player move
        row = int(input(f"Player {current_player}, enter row (0-2): "))
        col = int(input(f"Player {current_player}, enter column (0-2): "))
        # Check if the chosen cell is empty
```

```
        if board[row][col] == ' ':
            board[row][col] = current_player
            # Check for a winner
            if check_winner(board, current_player):
                print_board(board)
                print(f"Player {current_player} wins!")
                break
            # Check for a tie
            if is_board_full(board):
                print_board(board)
                print("It's a tie!")
                break
            # Switch to the other player
            current_player = 'O' if current_player == 'X' else 'X'
        else:
            print("Cell already taken. Try again.")

if __name__ == "__main__":
    tic_tac_toe()
```

Output:

```
"D:\Program Files\Python\Python310\python.exe" D:\Desktop\A.I\A.I\04.py

  |   |

---------

  |   |

---------

  |   |

---------

Player X, enter row (0-2):
```

5. Write a Program to Implement an 8-Puzzle problem using Python.

```
import heapq
import copy

class PuzzleNode:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost
        self.heuristic = self.calculate_heuristic()

    def calculate_heuristic(self):
        h = 0
        goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
        for i in range(3):
            for j in range(3):
                if self.state[i][j] != goal_state[i][j]:
                    h += 1
        return h

    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)
```

```
def get_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def get_possible_moves(blank_i, blank_j):
    moves = []
    if blank_i > 0:
        moves.append(('up', 1, 0))
    if blank_i < 2:
        moves.append(('down', -1, 0))
    if blank_j > 0:
        moves.append(('left', 0, 1))
    if blank_j < 2:
        moves.append(('right', 0, -1))
    return moves

def apply_move(state, move):
    blank_i, blank_j = get_blank_position(state)
    new_state = copy.deepcopy(state)
    new_i, new_j = blank_i + move[1], blank_j + move[2]
    new_state[blank_i][blank_j], new_state[new_i][new_j] = new_state[new_i][new_j],
new_state[blank_i][blank_j]
    return new_state

def print_puzzle(state):
    for row in state:
        print(" ".join(map(str, row)))

def solve_8_puzzle(initial_state):
    start_node = PuzzleNode(initial_state)
    frontier = [start_node]
    explored = set()

    while frontier:
        current_node = heapq.heappop(frontier)
        if current_node.state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]:
            print("Solution found:")
            path = []
            while current_node:
                path.append((current_node.move, current_node.state))
                current_node = current_node.parent
            for move, state in reversed(path):
                print(f"Move {move}:")
                print_puzzle(state)
                print()
            return

        explored.add(tuple(map(tuple, current_node.state)))

        blank_i, blank_j = get_blank_position(current_node.state)
        possible_moves = get_possible_moves(blank_i, blank_j)

        for move in possible_moves:
            new_state = apply_move(current_node.state, move)
            if tuple(map(tuple, new_state)) not in explored:
                new_node = PuzzleNode(new_state, current_node, move[0], current_node.cost
+ 1)
                heapq.heappush(frontier, new_node)

    print("No solution found.")

# Example usage:
```

```
if __name__ == "__main__":
    initial_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    solve_8_puzzle(initial_state)
```

6. Write a Program to Implement Traveling Salesman Problem using Python.

```
from itertools import permutations

def calculate_total_distance(tour, distances):
    total_distance = 0
    for i in range(len(tour) - 1):
        total_distance += distances[tour[i]][tour[i + 1]]
    total_distance += distances[tour[-1]][tour[0]]  # Return to the starting city
    return total_distance

def traveling_salesman_bruteforce(distances):
    num_cities = len(distances)
    cities = list(range(num_cities))

    min_distance = float('inf')
    optimal_tour = None

    for perm in permutations(cities):
        tour = list(perm)
        distance = calculate_total_distance(tour, distances)

        if distance < min_distance:
            min_distance = distance
            optimal_tour = tour

    return optimal_tour, min_distance

# Example usage:
if __name__ == "__main__":
    # Example distances between cities (replace with your own distances)
    distances = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]

    optimal_tour, min_distance = traveling_salesman_bruteforce(distances)

    print("Optimal Tour:", optimal_tour)
    print("Minimum Distance:", min_distance)
```

Output:

```
"D:\Program Files\Python\Python310\python.exe" D:\Desktop\A.I\A.I\06.py
Optimal Tour: [0, 1, 3, 2]
Minimum Distance: 80

Process finished with exit code 0
```

7. Write a Program to Implement Alpha-Beta Pruning using Python.

```
import math

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

```
def is_winner(board, player):
    # Check rows
    for row in board:
        if all(cell == player for cell in row):
            return True

    # Check columns
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True

    # Check diagonals
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for
i in range(3)):
        return True

    return False

def is_full(board):
    return all(cell != ' ' for row in board for cell in row)

def is_terminal(board):
    return is_winner(board, 'X') or is_winner(board, 'O') or is_full(board)

def evaluate(board):
    if is_winner(board, 'X'):
        return 1
    elif is_winner(board, 'O'):
        return -1
    else:
        return 0

def get_empty_cells(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def alpha_beta_pruning(board, depth, alpha, beta, maximizing_player):
    if depth == 0 or is_terminal(board):
        return evaluate(board)

    empty_cells = get_empty_cells(board)

    if maximizing_player:
        value = -math.inf
        for cell in empty_cells:
            i, j = cell
            board[i][j] = 'X'
            value = max(value, alpha_beta_pruning(board, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            board[i][j] = ' '  # Undo the move

            if beta <= alpha:
                break  # Beta cutoff
        return value
    else:
        value = math.inf
        for cell in empty_cells:
            i, j = cell
            board[i][j] = 'O'
            value = min(value, alpha_beta_pruning(board, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            board[i][j] = ' '  # Undo the move

            if beta <= alpha:
```

```
                break  # Alpha cutoff
        return value

def get_best_move(board):
    best_value = -math.inf
    best_move = None

    for cell in get_empty_cells(board):
        i, j = cell
        board[i][j] = 'X'
        move_value = alpha_beta_pruning(board, 10, -math.inf, math.inf, False)
        board[i][j] = ' '  # Undo the move

        if move_value > best_value:
            best_value = move_value
            best_move = (i, j)

    return best_move

def play_game():
    board = [[' ' for _ in range(3)] for _ in range(3)]

    while not is_terminal(board):
        print_board(board)
        player_move = tuple(map(int, input("Enter your move (row and column separated by
space): ").split()))
        if board[player_move[0]][player_move[1]] == ' ':
            board[player_move[0]][player_move[1]] = 'O'
        else:
            print("Invalid move. Cell already occupied. Try again.")
            continue

        if is_terminal(board):
            break

        print("AI is making a move...")
        ai_move = get_best_move(board)
        board[ai_move[0]][ai_move[1]] = 'X'

    print_board(board)
    winner = evaluate(board)
    if winner == 1:
        print("You lose!")
    elif winner == -1:
        print("You win!")
    else:
        print("It's a tie!")

if __name__ == "__main__":
    play_game()
```

8. Write a Program to Implement Water-Jug problem using Python.

```
from queue import Queue

class State:
    def __init__(self, jug1, jug2):
        self.jug1 = jug1
        self.jug2 = jug2

    def __eq__(self, other):
        return self.jug1 == other.jug1 and self.jug2 == other.jug2
```

```
    def __hash__(self):
        return hash((self.jug1, self.jug2))

    def __str__(self):
        return f"({self.jug1}, {self.jug2})"

def water_jug_bfs(capacity_jug1, capacity_jug2, target):
    start_state = State(0, 0)
    visited = set()
    queue = Queue()
    queue.put(start_state)
    visited.add(start_state)

    while not queue.empty():
        current_state = queue.get()
        print(f"Current State: {current_state}")

        if current_state.jug1 == target or current_state.jug2 == target:
            print("Target reached!")
            return

        # Fill Jug 1
        next_state = State(capacity_jug1, current_state.jug2)
        if next_state not in visited:
            queue.put(next_state)
            visited.add(next_state)

        # Fill Jug 2
        next_state = State(current_state.jug1, capacity_jug2)
        if next_state not in visited:
            queue.put(next_state)
            visited.add(next_state)

        # Empty Jug 1
        next_state = State(0, current_state.jug2)
        if next_state not in visited:
            queue.put(next_state)
            visited.add(next_state)

        # Empty Jug 2
        next_state = State(current_state.jug1, 0)
        if next_state not in visited:
            queue.put(next_state)
            visited.add(next_state)

        # Pour water from Jug 1 to Jug 2
        pour = min(current_state.jug1, capacity_jug2 - current_state.jug2)
        next_state = State(current_state.jug1 - pour, current_state.jug2 + pour)
        if next_state not in visited:
            queue.put(next_state)
            visited.add(next_state)

        # Pour water from Jug 2 to Jug 1
        pour = min(current_state.jug2, capacity_jug1 - current_state.jug1)
        next_state = State(current_state.jug1 + pour, current_state.jug2 - pour)
        if next_state not in visited:
            queue.put(next_state)
            visited.add(next_state)

if __name__ == "__main__":
    jug1_capacity = 4
    jug2_capacity = 3
    target_amount = 2
    water_jug_bfs(jug1_capacity, jug2_capacity, target_amount)
```

9. Write a Program to Implement Monkey Banana Problem using Python.

```
from queue import Queue

class State:
    def __init__(self, x, y, bananas_collected):
        self.x = x
        self.y = y
        self.bananas_collected = bananas_collected

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y and self.bananas_collected ==
other.bananas_collected

    def __hash__(self):
        return hash((self.x, self.y, self.bananas_collected))

    def __str__(self):
        return f"({self.x}, {self.y}, {self.bananas_collected})"

def is_valid_move(x, y, grid):
    return 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x][y] != 'X'

def monkey_banana_bfs(grid):
    start_state = State(0, 0, 0)
    visited = set()
    queue = Queue()
    queue.put(start_state)
    visited.add(start_state)

    while not queue.empty():
        current_state = queue.get()
        print(f"Current State: {current_state}")

        if grid[current_state.x][current_state.y] == 'B':
            current_state.bananas_collected += 1
            grid[current_state.x][current_state.y] = '.'  # Mark the banana as collected

        if current_state.bananas_collected == 2:  # Assuming there are 2 bananas to
collect
            print("Bananas collected!")
            return

        # Move right
        next_x, next_y = current_state.x, current_state.y + 1
        if is_valid_move(next_x, next_y, grid):
            next_state = State(next_x, next_y, current_state.bananas_collected)
            if next_state not in visited:
                queue.put(next_state)
                visited.add(next_state)

        # Move down
        next_x, next_y = current_state.x + 1, current_state.y
        if is_valid_move(next_x, next_y, grid):
            next_state = State(next_x, next_y, current_state.bananas_collected)
            if next_state not in visited:
                queue.put(next_state)
                visited.add(next_state)

if __name__ == "__main__":
    # Define the grid where 'X' represents obstacles and 'B' represents bananas
    grid = [
```

```
        ['.', '.', '.', 'X', '.'],
        ['.', 'X', '.', 'B', '.'],
        ['.', '.', '.', '.', '.'],
        ['B', '.', 'X', '.', '.'],
        ['.', '.', '.', 'X', 'B']
    ]

    monkey_banana_bfs(grid)
```

10. Write a Program to Implement Tower of Hanoi using Python.

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

if __name__ == "__main__":
    num_discs = 4  # Change this to the number of discs you want in the Tower of Hanoi

    tower_of_hanoi(num_discs, 'A', 'B', 'C')
```

__output:__

```
"D:\Program Files\Python\Python310\python.exe" D:\Desktop\A.I\A.I\10.py
Move disc 1 from A to B
Move disc 2 from A to C
Move disc 1 from B to C
Move disc 3 from A to B
Move disc 1 from C to A
Move disc 2 from C to B
Move disc 1 from A to B
Move disc 4 from A to C
Move disc 1 from B to C
Move disc 2 from B to A
Move disc 1 from C to A
Move disc 3 from B to C
Move disc 1 from A to B
Move disc 2 from A to C
Move disc 1 from B to C

Process finished with exit code 0
```