Write a Program to Implement the following using Python.

1.Breadth First Search

Code:

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            print(vertex, end=" ")
            visited.add(vertex)
            for neighbor in graph.get(vertex, []):
                if neighbor not in visited:
                    queue.append(neighbor)

# Get number of nodes and edges
n = int(input("Enter number of nodes: "))
e = int(input("Enter number of edges: "))

graph = {}

# Input edges
print("Enter edges (one pair per line, e.g., A B):")
for _ in range(e):
    u, v = input().split()
    if u not in graph:
        graph[u] = []
    if v not in graph:
        graph[v] = []
    graph[u].append(v)
    # If the graph is undirected, also add the reverse edge:
    # graph[v].append(u)

# Display the graph
print("\nGraph:")
for node, neighbors in graph.items():
    print(f"{node} -> {', '.join(neighbors)}")

# Input start node for BFS
start_node = input("\nEnter start node for BFS: ")
print("\nBFS traversal:")
bfs(graph, start_node)
```

Output:

```
Enter number of nodes: 6
Enter number of edges: 7
Enter edges:
A B
A C
B D
B E
C F
E F
D E
Enter start node for BFS: A
```

Graph:
A -> B, C
B -> D, E
C -> F
D -> E
E -> F
F ->

BFS traversal:
A B C D E F

2. Depth First Search
Code:
```
def dfs(graph, start):
    visited = set()
    stack = [start]

    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            print(vertex, end=" ")
            visited.add(vertex)
            # Add neighbors in reverse order for correct order of traversal
            for neighbor in reversed(graph.get(vertex, [])):
                if neighbor not in visited:
                    stack.append(neighbor)

# Get number of nodes and edges
n = int(input("Enter number of nodes: "))
e = int(input("Enter number of edges: "))

graph = {}

# Input edges
print("Enter edges (one pair per line, e.g., A B):")
for _ in range(e):
    u, v = input().split()
    if u not in graph:
        graph[u] = []
    if v not in graph:
        graph[v] = []
    graph[u].append(v)
    # For undirected graph, uncomment:
    # graph[v].append(u)

# Display the graph
print("\nGraph:")
for node, neighbors in graph.items():
    print(f"{node} -> {', '.join(neighbors)}")

# Input start node for DFS
start_node = input("\nEnter start node for DFS: ")
print("\nDFS traversal:")
dfs(graph, start_node)
```

Output:
Enter number of nodes: 6
Enter number of edges: 7
A B
A C
B D
B E
C F
E F
D E
Enter start node for DFS: A
Graph:
A -> B, C
B -> D, E
C -> F
D -> E
E -> F
F ->

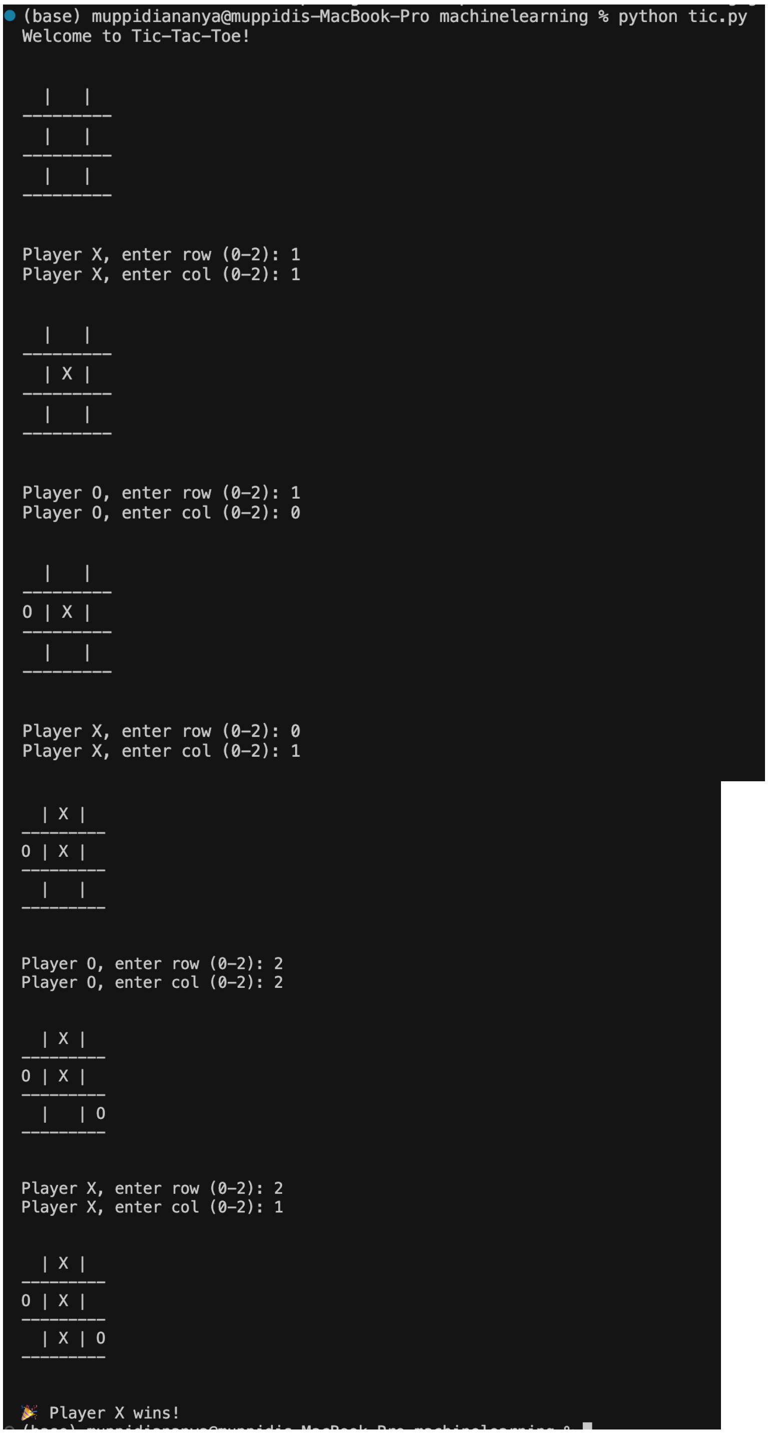
DFS traversal:
A C F B E D

3. Tic-Tac-Toe game

Code:
```
def print_board(board):
    print("\n")
    for row in board:
        print(" | ".join(row))
        print("-" * 9)
    print("\n")

def check_winner(board, player):
    # Check rows, columns, diagonals
    for row in board:
        if all(cell == player for cell in row):
            return True

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

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

def tic_tac_toe():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    current_player = 'X'

    print("Welcome to Tic-Tac-Toe!")
    print_board(board)

    while True:
        try:
            row = int(input(f"Player {current_player}, enter row (0-2): "))
            col = int(input(f"Player {current_player}, enter col (0-2): "))

            if not (0 <= row < 3 and 0 <= col < 3):
                print("❗ Invalid input. Row and col must be between 0 and 2.")
                continue

            if board[row][col] != ' ':
                print("❗ Cell already taken. Try another.")
                continue

            board[row][col] = current_player
            print_board(board)

            if check_winner(board, current_player):
                print(f"🎉 Player {current_player} wins!")
                break

            if is_draw(board):
                print("🤝 It's a draw!")
```

```
                break

            current_player = 'O' if current_player == 'X' else 'X'
        except ValueError:
            print("❗ Invalid input. Please enter numeric values.")

# Run the game
tic_tac_toe()
```

Output:



4. **8-Puzzle problem**
Code:
```
from collections import deque

# Directions: up, down, left, right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def is_valid(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def get_neighbors(state):
    neighbors = []
    zero_index = state.index('0')
    x, y = divmod(zero_index, 3)

    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy
        if is_valid(new_x, new_y):
            new_index = new_x * 3 + new_y
            state_list = list(state)
            # Swap '0' with the adjacent number
            state_list[zero_index], state_list[new_index] = state_list[new_index], state_list[zero_index]
            neighbors.append("".join(state_list))
    return neighbors

def bfs(start, goal):
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        current, path = queue.popleft()
        if current == goal:
            return path

        visited.add(current)
        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))
    return None

def print_state(state):
    for i in range(0, 9, 3):
        print(" ".join(state[i:i+3]))
    print()

# Take user input
start = input("Enter start state (e.g., 123456780): ").strip()
goal = input("Enter goal state (e.g., 123456780): ").strip()

# Validate
if len(start) != 9 or len(goal) != 9 or set(start) != set("012345678"):
    print("Invalid input! Use all digits from 0-8 exactly once.")
else:
    result = bfs(start, goal)
    if result:
        print("\nSolution found in", len(result) - 1, "moves:")
        for step in result:
            print_state(step)
    else:
        print("No solution found.")
```

Output:
```
Enter start state: 123456708
Enter goal state: 123456780
Solution found in 1 moves:
1 2 3
4 5 6
7   8

1 2 3
4 5 6
7 8
```

5. **Water-Jug problem**
Code:

```
from collections import deque

def is_goal(state, target):
    return state[0] == target or state[1] == target

def water_jug_bfs(cap_a, cap_b, target):
    visited = set()
    queue = deque()

    # Each state is (a, b, path)
    queue.append((0, 0, []))
    visited.add((0, 0))

    while queue:
        a, b, path = queue.popleft()

        # Goal check
        if is_goal((a, b), target):
            path.append((a, b))
            return path

        next_states = []

        # Fill Jug A or Jug B
        next_states.append((cap_a, b))
        next_states.append((a, cap_b))

        # Empty Jug A or Jug B
        next_states.append((0, b))
        next_states.append((a, 0))

        # Pour A → B
        pour = min(a, cap_b - b)
        next_states.append((a - pour, b + pour))

        # Pour B → A
        pour = min(b, cap_a - a)
        next_states.append((a + pour, b - pour))

        for new_a, new_b in next_states:
            if (new_a, new_b) not in visited:
                visited.add((new_a, new_b))
                queue.append((new_a, new_b, path + [(a, b)]))

    return None

# Inputs
cap_a = 4
cap_b = 3
target = 2

solution = water_jug_bfs(cap_a, cap_b, target)

if solution:
    print(f"\n✅ Found solution in {len(solution) - 1} steps:")
    for step in solution:
        print(f"Jug A: {step[0]}L, Jug B: {step[1]}L")
else:
```

```
    print("❌ No solution found.")
```

Output:

```
✅ Found solution in 6 steps:
Jug A: 0L, Jug B: 0L
Jug A: 4L, Jug B: 0L
Jug A: 1L, Jug B: 3L
Jug A: 1L, Jug B: 0L
Jug A: 0L, Jug B: 1L
Jug A: 4L, Jug B: 1L
Jug A: 2L, Jug B: 3L
```

6. **Travelling Salesman Problem**
Code:
```
from itertools import permutations

def get_distance_matrix(n):
    print(f"\nEnter the distance matrix (use space-separated values):")
    matrix = []
    for i in range(n):
        row = list(map(int, input(f"Row {i+1}: ").split()))
        if len(row) != n:
            raise ValueError("Each row must have exactly", n, "values.")
        matrix.append(row)
    return matrix

def tsp_brute_force(dist):
    n = len(dist)
    cities = list(range(n))
    min_cost = float('inf')
    best_path = []

    for perm in permutations(cities[1:]):  # Fix city 0 as start
        path = [0] + list(perm)
        cost = sum(dist[path[i]][path[i+1]] for i in range(n - 1))
        cost += dist[path[-1]][0]  # Return to start

        if cost < min_cost:
            min_cost = cost
            best_path = path

    return best_path, min_cost

# Main execution
try:
    n = int(input("Enter number of cities (e.g. 4): "))
    if n < 2:
        print("At least 2 cities required.")
    else:
        dist_matrix = get_distance_matrix(n)
        path, cost = tsp_brute_force(dist_matrix)

        # Optional: use letters for cities
        labels = [chr(65 + i) for i in range(n)]  # ['A', 'B', 'C', ...]
        named_path = [labels[i] for i in path]
        print("\n✅ Shortest Path:", " → ".join(named_path) + f" → {named_path[0]}")
        print("🧮 Minimum Cost:", cost)
except Exception as e:
    print("❌ Error:", e)
```
Output:
```
Enter number of cities (e.g. 4): 4

Enter the distance matrix (use space-separated values):
Row 1: 0 10 15 20
Row 2: 10 0 35 25
Row 3: 15 35 0 30
Row 4: 20 25 30 0

✅ Shortest Path: A → B → D → C → A
🧮 Minimum Cost: 80
```

7. **Tower of Hanoi**

Code:

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

# Main
try:
    n = int(input("Enter number of disks: "))
    if n <= 0:
        print("Please enter a positive number.")
    else:
        print(f"\nSteps to solve Tower of Hanoi with {n} disks:\n")
        tower_of_hanoi(n, 'A', 'B', 'C')  # A = source, B = auxiliary, C = target
except ValueError:
    print("Invalid input. Please enter an integer.")
```

Output:

```
Enter number of disks: 3

Steps to solve Tower of Hanoi with 3 disks:

Move disk 1 from A → C
Move disk 2 from A → B
Move disk 1 from C → B
Move disk 3 from A → C
Move disk 1 from B → A
Move disk 2 from B → C
Move disk 1 from A → C
```

8. **Monkey Banana Problem**
Code:
```
from collections import deque

# Positions
POSITIONS = ['door', 'window']

# Initial state: (monkey_pos, box_pos, has_bananas)
initial_state = ('door', 'door', False)
goal_state = ('window', 'window', True)  # monkey on window, box on window, has bananas

def is_goal(state):
    return state[2] == True

def get_next_states(state):
    monkey, box, has_bananas = state
    next_states = []

    # Actions:

    # 1. Monkey moves alone
    for pos in POSITIONS:
        if pos != monkey:
            next_states.append((pos, box, has_bananas))

    # 2. Monkey pushes box (only if monkey and box in same place)
    for pos in POSITIONS:
        if pos != box and monkey == box:
            next_states.append((pos, pos, has_bananas))

    # 3. Monkey climbs box (only if monkey and box same place)
    # Represented as monkey position 'box' meaning monkey is on box at box position
    if monkey == box and monkey != 'box':
        next_states.append(('box', box, has_bananas))

    # 4. Monkey grabs bananas (only if monkey on box at window)
    if monkey == 'box' and box == 'window' and not has_bananas:
        next_states.append((monkey, box, True))

    return next_states

def bfs():
    queue = deque()
    queue.append((initial_state, []))
    visited = set()
    visited.add(initial_state)

    while queue:
        current_state, path = queue.popleft()
        if is_goal(current_state):
            return path + [current_state]

        for next_state in get_next_states(current_state):
            if next_state not in visited:
                visited.add(next_state)
                queue.append((next_state, path + [current_state]))

    return None

def print_solution(solution):
```

```python
    if not solution:
        print("No solution found.")
        return
    print("Solution steps:")
    for idx, state in enumerate(solution):
        monkey, box, bananas = state
        monkey_pos = monkey if monkey != 'box' else f"on box at {box}"
        print(f"Step {idx}: Monkey at {monkey_pos}, Box at {box}, Has bananas: {bananas}")

if __name__ == "__main__":
    solution = bfs()
    print_solution(solution)
```

Output:

Step 0: Monkey at door, Box at door, Has bananas: False
Step 1: Monkey at window, Box at door, Has bananas: False
Step 2: Monkey at door, Box at door, Has bananas: False
Step 3: Monkey at door, Box at window, Has bananas: False
Step 4: Monkey at window, Box at window, Has bananas: False
Step 5: Monkey on box at window, Box at window, Has bananas: False
Step 6: Monkey on box at window, Box at window, Has bananas: True

9. **Alpha-Beta Pruning**
Code:

```
def build_tree(leaves, branching_factor):
    """
    Build a balanced tree from a list of leaves.
    Example: For branching_factor=3 and leaves length 9,
    creates a 2-level tree with 3 children each having 3 leaves.
    """
    if len(leaves) == 1:
        return leaves[0]

    tree = []
    for i in range(0, len(leaves), branching_factor):
        subtree = build_tree(leaves[i:i+branching_factor], branching_factor)
        tree.append(subtree)
    return tree

def alpha_beta(node, depth, alpha, beta, maximizing_player):
    if depth == 0 or isinstance(node, int):
        return node

    if maximizing_player:
        max_eval = float('-inf')
        for child in node:
            eval = alpha_beta(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta cutoff
        return max_eval
    else:
        min_eval = float('inf')
        for child in node:
            eval = alpha_beta(child, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Alpha cutoff
        return min_eval

def main():
    try:
        depth = int(input("Enter the depth of the tree (e.g. 2): "))
        branching_factor = int(input("Enter the branching factor (e.g. 3): "))
        num_leaves = branching_factor ** depth
        print(f"You need to enter {num_leaves} leaf node values (space separated):")
        leaves = list(map(int, input().split()))
        if len(leaves) != num_leaves:
            print(f"Error: You must enter exactly {num_leaves} integers.")
            return

        tree = build_tree(leaves, branching_factor)
        result = alpha_beta(tree, depth, float('-inf'), float('inf'), True)
        print("\nOptimal value using Alpha-Beta Pruning:", result)
    except Exception as e:
        print("Error:", e)

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

Output:
Enter the depth of the tree (e.g. 2): 2
Enter the branching factor (e.g. 3): 3
You need to enter 9 leaf node values (space separated):
3 5 6 3 2 9 0 1 5

Optimal value using Alpha-Beta Pruning: 5

10. **8-Queens Problem**

Code:

```
def is_safe(board, row, col, n):
    # Check this column on upper side
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check upper left diagonal
    i, j = row - 1, col - 1
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
            return False
        i -= 1
        j -= 1

    # Check upper right diagonal
    i, j = row - 1, col + 1
    while i >= 0 and j < n:
        if board[i][j] == 1:
            return False
        i -= 1
        j += 1

    return True

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

    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = 1
            if solve_n_queens_util(board, row + 1, n):
                return True
            board[row][col] = 0  # backtrack

    return False

def print_board(board, n):
    for row in range(n):
        for col in range(n):
            print("Q" if board[row][col] == 1 else ".", end=" ")
        print()
    print()

def main():
    try:
        n = int(input("Enter the number of queens (N): "))
        if n <= 0:
            print("Please enter a positive integer.")
            return

        board = [[0]*n for _ in range(n)]

        if solve_n_queens_util(board, 0, n):
            print(f"\nOne solution to the {n}-Queens problem:")
            print_board(board, n)
        else:
            print("No solution exists.")
```

```
    except ValueError:
        print("Invalid input. Please enter an integer.")

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

Output:

```
Enter the number of queens (N): 8

One solution to the 8-Queens problem:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```