Name – Arnav Gupta

Branch - CSE(AIML)

Sec - A

Roll No. – 48

Project Name

N Queens Problem

Problem Statement: N-Queens Problem

The **N-Queens problem** is a classic combinatorial problem in which we must place **N queens** on an **N × N chessboard** such that:

1. **No two queens are in the same row**
2. **No two queens are in the same column**
3. **No two queens are on the same diagonal**

A queen in chess can attack **horizontally, vertically, and diagonally**, so the challenge is to place all `N` queens in a way that they do not threaten each other.

Example for N = 4 (4-Queens Problem)

For `N = 4`, we must place **4 queens** on a **4 × 4** board. One possible valid solution is:

```
. Q . .    (Queen in row 0, column 1)
. . . Q    (Queen in row 1, column 3)
Q . . .    (Queen in row 2, column 0)
. . Q .    (Queen in row 3, column 2)
```

Here, no two queens attack each other.

Input and Output

- **Input:**
  - A single integer `N`, representing the size of the board (`N × N`) and the number of queens to place.
  - Example: `N = 4`
- **Output:**
  - A list of all valid ways to place `N` queens. Each solution is displayed as an `N × N` board with **"Q"** for queens and **"."** for empty spaces.
  - Example output for `N = 4`:

```
. Q . .
. . . Q
Q . . .
. . Q .
```

```
. . Q .
Q . . .
. . . Q
. Q . .
```

- The program may print multiple solutions because there can be **more than one valid arrangement**.

Constraints

- The problem has **no solution** for `N = 2` and `N = 3`, as it's impossible to place queens without them attacking each other.
- The smallest valid case is `N = 4`.

Methodology

The **N-Queens problem** is solved using a **backtracking algorithm**, which systematically explores all possible ways to place `N` queens on an `N × N` chessboard while ensuring that no two queens attack each other. The approach follows these steps:

1. **Problem Representation**:
   - The chessboard is represented using a **1D array (`board`)**, where `board[row] = col` indicates that a queen is placed in the given row and column.
2. **Backtracking Approach**:
   - The algorithm starts placing queens **row by row**, beginning with the first row.
   - For each row, it iterates over all columns to find a **valid position** for the queen.
3. **Validation of Queen Placement**:
   - A function (`is_valid`) checks if placing a queen at `(row, col)` is safe by verifying:
     - No other queen is in the **same column**.
     - No other queen is on the **same diagonal**.
   - If a position is valid, the queen is placed, and the algorithm moves to the next row.
4. **Recursive Exploration**:
   - If all `N` queens are successfully placed (`row == N`), a valid solution is found and stored.
   - If no valid column exists for a queen in a row, **backtracking** is applied:
     - The last placed queen is removed, and the algorithm retries the next column in the previous row.
5. **Solution Storage and Output**:
   - The algorithm finds all possible valid placements of `N` queens and stores them as formatted chessboard representations.
   - The final output consists of all unique solutions.

Complexity Considerations

- The backtracking approach ensures an efficient search through the solution space, but the worst-case **time complexity** is **O(N!)**, as each queen has multiple placement choices.

This method guarantees finding all possible solutions while minimizing unnecessary computations through backtracking.

Code For N Queens :

```
def solve_n_queens(n):
    # Ensure n is at least 4, as there are no solutions for n < 4.
    if n < 4:
        raise ValueError("n must be at least 4.")

    def is_valid(board, row, col):
        # Check previous rows for conflicts in the same column or diagonals
        for i in range(row):
            # Same column check
            if board[i] == col:
                return False
            # Check left diagonal
            if board[i] - i == col - row:
                return False
            # Check right diagonal
            if board[i] + i == col + row:
                return False
        return True

    def backtrack(row, board, solutions):
        # If all queens are placed successfully
        if row == n:
            # Convert board representation to a readable format
            solution = []
            for i in range(n):
                line = ['.'] * n  # Create an empty row
                line[board[i]] = 'Q'  # Place queen in the correct column
                solution.append("".join(line))  # Convert list to string
```

```
        solutions.append(solution)  # Store the valid solution
        return

    # Try placing a queen in each column of the current row
    for col in range(n):
        if is_valid(board, row, col):  # Check if it's safe to place a queen
            board[row] = col  # Place the queen at (row, col)
            backtrack(row + 1, board, solutions)  # Move to the next row

  solutions = []  # List to store all valid solutions
  board = [-1] * n  # Array to store queen positions (column index for each row)
  backtrack(0, board, solutions)  # Start backtracking from the first row
  return solutions  # Return all valid solutions

# Take input from the user
try:
  user_input = input("Enter the value of n (n must be at least 4): ")  # Prompt user input
  n = int(user_input)  # Convert input to integer
  results = solve_n_queens(n)  # Call the function to solve N-Queens
  print("Number of solutions:", len(results))  # Print total solutions
  for solution in results:
    for line in solution:
        print(line)  # Print each row of the solution
    print("")  # Print a blank line between solutions
except ValueError as ve:
  print("Error:", ve)  # Handle invalid input
```

<u>Output</u> :

For N=5

```
Enter the value of n (n must be at least 4): 5
Number of solutions: 10
Q....
..Q..
....Q
.Q...
...Q.

Q....
...Q.
.Q...
....Q
..Q..

.Q...
...Q.
Q....
..Q..
....Q

.Q...
....Q
..Q..
Q....
...Q.

..Q..
Q....
...Q.
.Q...
....Q

..Q..
....Q
.Q...
...Q.
Q....
```

Special Thanks:

Bikki Gupta Sir