TITLE PAGE

Project Name: N-Queens Problem

Student Name: Rituraj kumar

Roll No: 202401100300203

Institution: KIET Group of Institutions

Date: 11 March 2025

INTRODUCTION

The N-Queens problem is a well-known combinatorial puzzle that requires placing N queens on an N×N chessboard in such a way that no two queens can attack each other. This means that:

- **No two queens share the same row.**
- **No two queens share the same column.**
- **No two queens share the same diagonal.**

This report explores a Python-based approach using the Backtracking Algorithm to solve the problem efficiently.

METHODOLOGY

The solution follows a backtracking approach:

1. **Initialize an empty chessboard: The board is represented as a list where each index represents a row, and the value at each index represents the column position of a queen.**
2. **Place Queens Row by Row: The algorithm attempts to place a queen in each row.**
3. **Check for Safe Placement: Before placing a queen in a column, it checks if the placement is safe.**
4. **Recursive Exploration: If safe, it moves to the next row and repeats the process.**
5. **Backtracking: If a row has no valid placements, the algorithm backtracks to the previous row and tries a different column.**
6. **Complete Solution: Once all queens are placed safely, the board is printed as output.**

CODE:

```
def print_solution(board):
    """Prints the chessboard with queens represented as 'Q' and empty spaces as '.'"""
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
    print("\n")


def is_safe(board, row, col, n):
    """Checks if a queen can be placed at board[row][col] without being attacked."""

    # Check column for any existing queen
    for i in range(row):
        if board[i][col]:
            return False

    # Check upper-left diagonal for any existing queen
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j]:
            return False

    # Check upper-right diagonal for any existing queen
    for i, j in zip(range(row, -1, -1), range(col, n)):
```

```
        if board[i][j]:
            return False
    return True

def solve_n_queens(board, row, n, solutions):
    """Uses backtracking to place queens safely on the board."""

    # Base case: If all queens are placed, add solution to list
    if row == n:
        solutions.append([row[:] for row in board])
        print_solution(board)  # Print the valid board configuration
        return

    # Try placing a queen in each column of the current row
    for col in range(n):
        if is_safe(board, row, col, n):  # Check if placement is safe
            board[row][col] = 1  # Place the queen
            solve_n_queens(board, row + 1, n, solutions)  # Recur to place the next
queen
            board[row][col] = 0  # Backtrack: Remove the queen and try next
column

def n_queens(n):
    """Initializes the chessboard and starts the backtracking algorithm."""
    board = [[0] * n for _ in range(n)]  # Create an n x n board initialized with 0
    solutions = []  # List to store all valid solutions
```

```
    solve_n_queens(board, 0, n, solutions)  # Start placing queens from the first row
    return solutions


if __name__ == "__main__":
    n = 4  # Change this value to solve for different board sizes
    solutions = n_queens(n)  # Solve the N-Queens problem
    print(f"Total solutions for {n}-Queens: {len(solutions)}")  # Print the total number of solutions
```

OUTPUT/RESULT:

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


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


Total solutions for 4-Queens: 2
```

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


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


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


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


Total solutions for 6-Queens: 4
```

REFERENCES/CREDITS:

- **Python Official Documentation: https://docs.python.org**
- **N-Queens Problem Explanation: GeeksforGeeks**