Lab task 4 (Report)

# Title: N-Queens Problem Solver

The code implements a solution to the classic N-Queens problem. This problem involves placing N chess queens on an N×N chessboard so that no two queens threaten each other. In other words, no two queens share the same row, column, or diagonal.

**Algorithm**

The code uses a backtracking algorithm to find all possible solutions to the N-Queens problem.

1. **Backtracking:** The core of the algorithm is the backtrack function. It explores the search space by placing queens on the board one by one. It starts with an empty board (represented by an empty list of queens) and tries to place a queen in each column of the current row. If a safe position is found, it recursively calls itself for the next row. If it reaches the last row and has successfully placed all queens, it adds the solution to a list of solutions. If it cannot find a safe position in the current row, it backtracks to the previous row and tries a different column.
2. **Constraint Checking:** The check function verifies if a queen can be placed at a specific position without conflicting with other queens already on the board. It checks if the current position shares a column or diagonal with any of the existing queens. If a conflict is detected, it returns False; otherwise, it returns True.
3. **Solution Collection:** When the backtrack function finds a valid placement of all N queens, it adds a copy of the current board configuration (represented by the queens list) to a list of solutions.

**Implementation**

1. nqueens(n): This function initializes the backtracking process. It takes the board size (n) as input and calls the backtrack function with an initial row of 0 and an empty list of queens. It returns the list of solutions found.
2. check(row, col, queens): This function checks if placing a queen at the given row and col is safe, considering the existing queens on the board. It returns True if the placement is safe, False otherwise.
3. backtrack(row, queens): This is the recursive function that explores the search space. It takes the current row and the list of queens already placed as input. It tries to place a queen in each column of the current row. If a safe position is found using the check function, it adds the queen to the queens list and recursively calls itself for the next row. If it reaches the last row (row == n), it means a solution is found, and it adds a copy of the solution to the list of solutions. If it cannot find a safe position in the current row, it backtracks by removing the last queen placed and tries a different column.
4. print\_solution(solution, n): This function takes a solution (a list of queen positions) and the board size (n) as input. It creates a visual representation of the board with queens placed according to the solution and prints it to the console.

**Usage**

The code demonstrates solving the 4-Queens problem. It calls nqueens with n = 4 to find solutions and then iterates through the solutions, printing each one using print\_solution.

**Output**

To see the output, run the code. The output will display the number of solutions found and a visual representation of each solution on the board. Queens are represented by 'Q', and empty squares by

