Lab 4

Task: N-Queens Problem (Dynamic)

Here's a step-by-step explanation of how the code works:

1. Function `is_safe(board, row, col, N)`

- **Purpose**: This function checks whether placing a queen at position `(row, col)` on the board is safe, considering the rules of the N-Queens problem.
- **Parameters**:
    - `board`: A list that keeps track of the column positions of the queens placed so far.
    - `row`: The current row where we are attempting to place a queen.
    - `col`: The column where we want to place the queen.
    - `N`: The size of the board (N x N).
- **Logic**:
    - **Check Column**: It iterates through all the previous rows (i.e., `for i in range(row)`) to check if there's already a queen placed in the same column (`board[i] == col`).
    - **Check Diagonals**: It checks if the current position is on the same diagonal as any previously placed queens. This is done by checking if `abs(board[i] - col) == row - i`, which ensures that the difference in columns is equal to the difference in rows, i.e., a diagonal conflict.
    - If either condition is violated, the function returns `False` (indicating the position is not safe). If no conflicts, it returns `True`.

2. Function `solve_n_queens_util(board, row, N, solutions)`

- **Purpose**: This is a **backtracking function** that tries to place queens row by row and recursively explores all possible placements.
- **Parameters**:
    - `board`: A list that holds the column positions of queens for each row.
    - `row`: The current row where we are attempting to place a queen.
    - `N`: The size of the board (N x N).
    - `solutions`: A list to store all valid solutions (complete configurations of queens on the board).
- **Logic**:
    - **Base Case**: If `row == N`, all queens have been placed successfully, and the current board configuration is added to the list of solutions (`solutions.append(board[:])`).
    - **Recursive Case**: For the current row, it tries placing a queen in each column (`for col in range(N)`).
        - If the position is safe (`is_safe(board, row, col, N)`), it places a queen by setting `board[row] = col` and then recursively calls

`solve_n_queens_util(board, row + 1, N, solutions)` to place queens in the next row.
        - After the recursive call, it **backtracks** by removing the queen (`board[row] = -1`) to explore other possibilities.

3. Function `solve_n_queens(N)`

- **Purpose**: This function initializes the board and starts the recursive backtracking process to solve the N-Queens problem.
- **Parameters**:
    - `N`: The size of the chessboard (N x N) and the number of queens.
- **Logic**:
    - **Board Initialization**: `board = [-1] * N` creates a list of size `N` initialized to `-1`, indicating that no queens have been placed.
    - **Solutions List**: `solutions = []` initializes an empty list to store all valid solutions.
    - **Backtracking Call**: It starts the recursive process by calling `solve_n_queens_util(board, 0, N, solutions)` starting from row 0.
    - **Display Solutions**: After the backtracking process finishes:
        - It prints the total number of solutions found (`print(f"Total solutions: {len(solutions)}")`).
        - It iterates through each solution and prints the board in a human-readable format where queens are represented by `'Q'` and empty spaces by `'.'`. The row is formatted such that the queen's position is marked on the board.

OUTPUT: