

Submitted to: Sir Rasikh Ali

Submitted by: Dua Saeed

Subject: PAI (Lab)

Section: BSAI-4A

Roll number: 042

N-Queens Problem (Dynamic):

**Introduction**

The **N-Queens problem** is a classic combinatorial problem that involves placing **N queens** on an **N × N chessboard** such that no two queens attack each other. This means that:

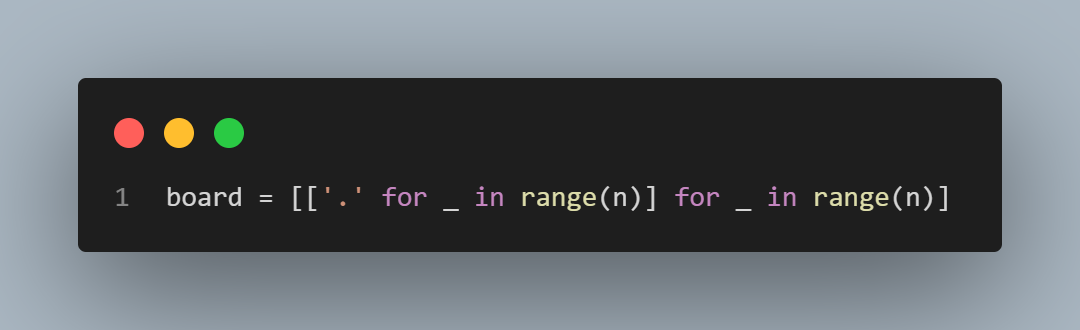
* No two queens can be in the same **row**.
* No two queens can be in the same **column**.
* No two queens can be in the same **diagonal**.

This report explains the provided Python implementation, how it works, and the underlying principles.

**Code Explanation**

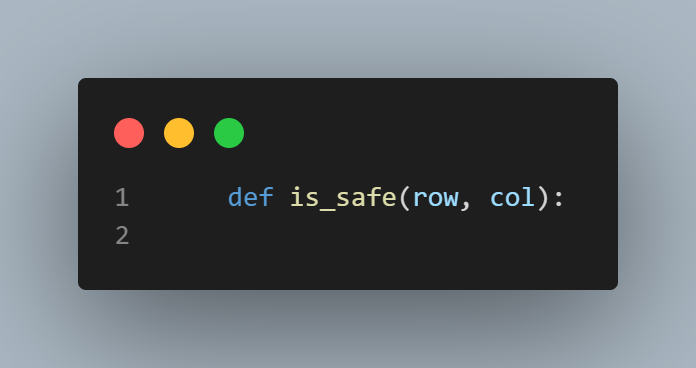
The given Python code is a **backtracking solution** to the N-Queens problem. It constructs a board and recursively places queens while ensuring that they follow the constraints.

1. **Initializing the Chessboard**



* Creates an N × N grid initialized with '.' (empty spaces).
* The board is represented as a **list of lists**.

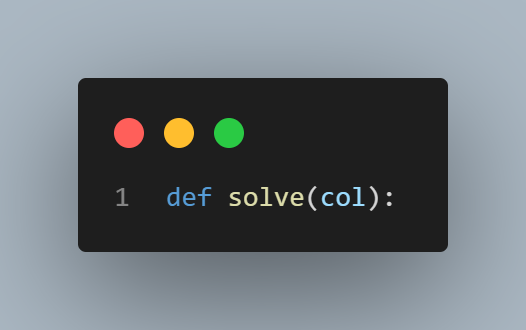
1. **Safety Check (is\_safe function)**

****

This function checks whether placing a queen at board[row][col] is safe:

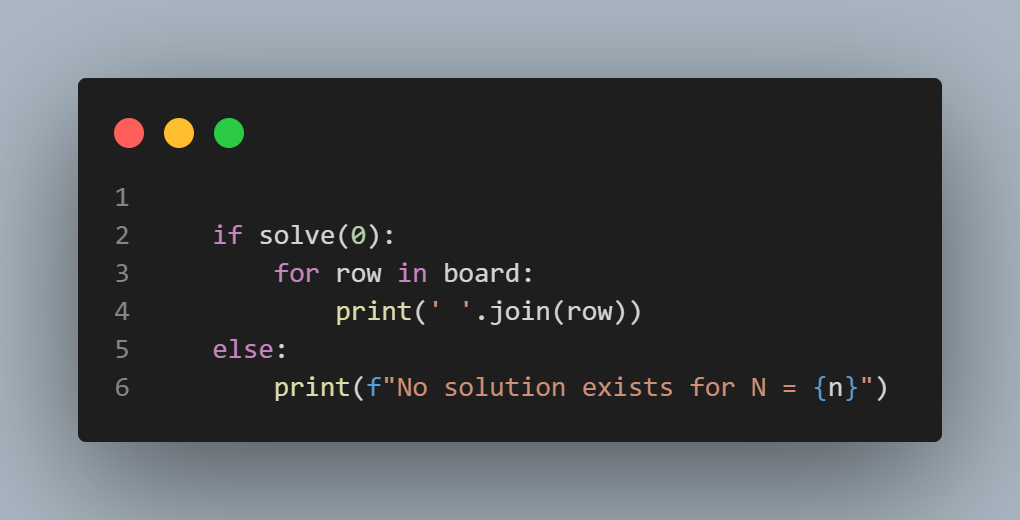
* **Left-side check**: Ensures no queen exists in the same row to the left.
* **Upper-left diagonal check**: Ensures no queen exists diagonally in the top-left direction.
* **Lower-left diagonal check**: Ensures no queen exists diagonally in the bottom-left direction.

1. **Solving the Problem (solve function)**

****

This function attempts to place queens column by column using recursion:

1. **Base Case**: If all columns are filled (col >= n), the board is successfully solved.
2. **Recursive Step**: For each row in the current column:
   * Check if placing a queen is safe using is\_safe(i, col).
   * Place the queen if safe (board[i][col] = 'Q').
   * Recur for the next column (solve(col + 1)).
   * If placing the queen does not lead to a solution, backtrack (board[i][col] = '.').
3. **If no placement works, return False** (backtracking).
4. **Execution and Output**

****

* If solve(0) finds a solution, the chessboard configuration is printed.
* If no solution exists, an appropriate message is displayed.

**Complexity Analysis**

* **Time Complexity**: O(N!) (Factorial time complexity)
  + Since every queen placement leads to recursive calls with reducing choices, the worst case results in exponential time.
* **Space Complexity**: O(N^2) (Chessboard storage)
  + However, by using a 1D array to track queen positions, this can be reduced to O(N).

**Why Backtracking is Used?**

* **Efficient pruning**: If a queen placement is unsafe, the algorithm backtracks immediately.
* **Recursive exploration**: Allows systematically exploring valid queen positions.
* **Space-efficient**: Compared to brute-force approaches that store all solutions.

**Output:**

