The N-Queens problem is a puzzle that involves placing N-queens on an N × N chessboard in such a way that no two queens can attack each other. This code dynamically solves the N-Queens problem, ensuring that no two queens are placed in the same row, column, or diagonal.

**Code Overview:**

This code solves the N-Queen problem using backtracking. This code involves:

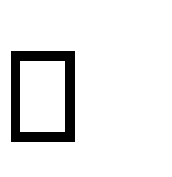
* Recursively placing queens row by row.
* Checking weather placing a queen is save or not.
* Backtracking when necessary.
* Printing all the possible solutions.

**Functions:**

* **def print\_board(board):**

printing the chessboard where “Q” represents the placed queen and “.” Represents the empty space.

**Parameters:**



borad: A 2-D array representing the chessboard.

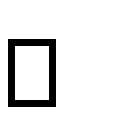
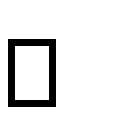
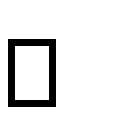
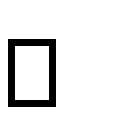
* **def placing\_safe(board, row, col, n):**

Checking weather placing a queen is save or not at the position.

* + Checking upper left diagonal
  + Checking queen in same column
  + Checking upper right diagonal

**Parameters:**

borad: Checking weather placing a queen is save or not.



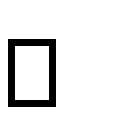
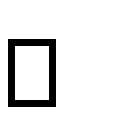
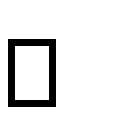
row : row index where the queen is being placed. col: column index where the queen is being placed.

n: size of chessboard (N\*N). returns **True** if the queen is placed safely otherwise **False**.

* **def solution\_Nqueen(board, row,n):**

Recursively trying to place queen row by row while ensuring no queen is attacking.

**Parameters:**  borad: Checking weather placing a queen is save or not. row : the current row is processing. n: size of chessboard (N\*N).



returns **True** if the solution is found otherwise **False**

* **def n\_queens():**

Tacking input from user and initializing the size of board.

Prints the solution if found, otherwise prints "Solution not found".

