""" Design n-Queens matrix having first Queen placed. Use backtracking to place remaining

Queens to generate the final n-queen’s matrix. """

class NQueens:

def \_\_init\_\_(self):

self.size = int(input("Enter size of chessboard: "))

self.board = [[False] \* self.size for \_ in range(self.size)]

self.count = 0

def printBoard(self):

for row in self.board:

for ele in row:

print("Q" if ele else "X", end=" ")

print()

print()

def isSafe(self, row: int, col: int) -> bool:

# Check the column

for i in range(self.size):

if self.board[i][col]:

return False

# Check the diagonals

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if self.board[i][j]:

return False

for i, j in zip(range(row, -1, -1), range(col, self.size)):

if self.board[i][j]:

return False

for i, j in zip(range(row, self.size), range(col, -1, -1)):

if self.board[i][j]:

return False

for i, j in zip(range(row, self.size), range(col, self.size)):

if self.board[i][j]:

return False

return True

def set\_position\_first\_queen(self):

print("Enter coordinates of the first queen: ")

row = int(input(f"Enter row (1-{self.size}): ")) - 1

col = int(input(f"Enter column (1-{self.size}): ")) - 1

# Ensure the position is valid

if 0 <= row < self.size and 0 <= col < self.size:

self.board[row][col] = True

self.printBoard()

else:

print("Invalid coordinates. Please try again.")

self.set\_position\_first\_queen() # Retry input

def solve(self, row: int):

if row == self.size:

self.count += 1

self.printBoard()

return

# Skip rows that already have a queen

if any(self.board[row]):

self.solve(row + 1)

return

for col in range(self.size):

if self.isSafe(row, col):

self.board[row][col] = True

self.solve(row + 1)

self.board[row][col] = False # Backtrack

def displayMessage(self):

if self.count > 0:

print(f"Total solutions found: {self.count}")

else:

print("No solutions exist for the given position of the queen.")

if \_\_name\_\_ == "\_\_main\_\_":

solver = NQueens()

solver.set\_position\_first\_queen()

solver.solve(0)

solver.displayMessage()

"""

output:-

Enter size of chessboard: 4

Enter coordinates of the first queen:

Enter row (1-4): 1

Enter column (1-4): 2

X Q X X

X X X X

X X X X

X X X X

X Q X X

X X X Q

Q X X X

X X Q X

Total solutions found: 1

"""

QA

### 1. What is Backtracking? Give the General Procedure

**Backtracking** is a systematic method for solving constraint satisfaction problems where we incrementally build up solutions and abandon a partial solution as soon as it’s determined that this solution cannot possibly lead to a valid or optimal solution. Backtracking is widely used in combinatorial problems like puzzle solving, pathfinding, and arrangement-based problems such as the N-Queens problem.

The basic idea of backtracking is to try out each possibility, and if a possibility doesn’t lead to a solution, we backtrack to the previous step and try another path.

**General Procedure for Backtracking**:

1. Start by selecting an option or placing an item (e.g., placing a queen in a column).
2. If the current option leads to a solution, mark it as part of the solution.
3. Move to the next step in the sequence.
4. If the step fails to meet the constraints or doesn’t lead to a solution, backtrack by undoing the previous step and try an alternative option.
5. Repeat the process until all solutions are found or a valid solution is reached.

**Example Problems Solved by Backtracking**:

* **N-Queens Problem**
* **Sudoku Solving**
* **Graph Coloring**
* **Subset Sum Problem**

### 2. Problem Statement of the N-Queens Problem and Solution Explanation

The **N-Queens Problem** is a classic problem in combinatorial optimization where the goal is to place NNN queens on an N×NN \times NN×N chessboard so that no two queens threaten each other. This means:

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

**Problem Statement**: Given an N×NN \times NN×N chessboard, place NNN queens on the board so that:

1. No two queens share the same row.
2. No two queens share the same column.
3. No two queens share the same diagonal.

**Solution Explanation**: To solve the N-Queens problem using backtracking:

1. Place queens one by one in different columns starting from the first row.
2. For each queen placement, check if the position is safe by ensuring it is not attacked by any previously placed queen.
3. If a safe position is found in the current row, move to the next row to place the next queen.
4. If placing the queen in any column of a row doesn’t lead to a solution, backtrack and move the previous queen to the next column.
5. Repeat this process until all queens are placed, or all positions are exhausted.

This solution finds all possible arrangements for the queens, providing either one or multiple solutions based on the problem requirement.

### 3. Algorithm for N-Queens Problem using Backtracking

Below is the algorithm for solving the N-Queens problem using backtracking.

python

Copy code

def is\_safe(board, row, col, n):

# Check this row on the left side

for i in range(col):

if board[row][i] == 1:

return False

# Check the upper diagonal on the left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check the lower diagonal on the left side

for i, j in zip(range(row, n), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, col, n):

# If all queens are placed, return true

if col >= n:

return True

for i in range(n):

if is\_safe(board, i, col, n):

# Place queen in this position

board[i][col] = 1

# Recur to place rest of the queens

if solve\_n\_queens\_util(board, col + 1, n):

return True

# If placing queen in board[i][col] doesn't lead to a solution,

# then remove queen from board[i][col]

board[i][col] = 0 # Backtrack

return False

def solve\_n\_queens(n):

board = [[0] \* n for \_ in range(n)]

if not solve\_n\_queens\_util(board, 0, n):

print("Solution does not exist")

return

for row in board:

print(" ".join(str(cell) for cell in row))

# Example usage

solve\_n\_queens(4)

**Explanation**:

* **is\_safe**: Checks if placing a queen at a given position is safe by ensuring no other queens threaten the position in the row, upper diagonal, and lower diagonal.
* **solve\_n\_queens\_util**: Recursively tries placing queens in each column, backtracking when necessary.
* **solve\_n\_queens**: Initializes the board and calls the utility function, printing the board if a solution is found.

This algorithm works for any size NNN, though larger values may result in significantly longer computation times.

### 4. Why is it Applicable to N=4 and N=8 Only?

The N-Queens problem can actually be solved for any value of N≥1N \geq 1N≥1, but certain values like N=2N=2N=2 and N=3N=3N=3 do not have solutions. Specifically:

* For **N=2** and **N=3**, no arrangement allows the queens to avoid each other, so no solutions exist.
* For **N=4** and **N=8**, solutions do exist and are easier to compute. These values are often referenced because they are historically famous and small enough to be solved and visualized easily.

In general, the problem has solutions for any N≥4N \geq 4N≥4, and solutions become increasingly complex as NNN grows, requiring more computational power for larger values. For instance:

* **N=8** is popular in examples because it is the classic version of the N-Queens problem, also known as the "8-Queens Problem," and has 92 distinct solutions.