"""

Assignment-B4 (N-Queen)

Problem Statement: Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.

"""

def placeQueens(i, cols, leftDiagonal, rightDiagonal, cur):

n = len(cols)

if i == n:

return True

for j in range(n):

if cols[j] or rightDiagonal[i + j] or leftDiagonal[i - j + n - 1]:

continue

cols[j] = 1

rightDiagonal[i + j] = 1

leftDiagonal[i - j + n - 1] = 1

cur.append(j)

if placeQueens(i + 1, cols, leftDiagonal, rightDiagonal, cur):

return True

cur.pop()

cols[j] = 0

rightDiagonal[i + j] = 0

leftDiagonal[i - j + n - 1] = 0

return False

def nQueen(n):

cols = [0] \* n

leftDiagonal = [0] \* (n \* 2)

rightDiagonal = [0] \* (n \* 2)

cur = []

board = [['.' for \_ in range(n)] for \_ in range(n)]

if placeQueens(0, cols, leftDiagonal, rightDiagonal, cur):

for i in range(n):

board[i][cur[i]] = 'Q'

return board

else:

return None

def printBoard(board):

if board:

for row in board:

print(" ".join(row))

else:

print("No solution exists.")

n = int(input("Enter the number of queens:\t"))

board = nQueen(n)

printBoard(board)

Here's a complete explanation of your \*\*Assignment-B4 (N-Queen Problem)\*\* code along with relevant \*\*viva questions and answers\*\*:

---

## 🧠 \*\*Explanation of the Code (N-Queens using Backtracking and Branch & Bound)\*\*

### ✅ \*\*Problem Statement\*\*:

Solve a \*\*Constraint Satisfaction Problem (CSP)\*\* using \*\*Backtracking\*\* and \*\*Branch and Bound\*\* for either the \*\*N-Queen problem\*\* or \*\*Graph Coloring\*\*.

You're solving the \*\*N-Queens Problem\*\*.

---

## ♛ \*\*What is the N-Queens Problem?\*\*

The task is to \*\*place `n` queens on an `n x n` chessboard\*\* so that \*\*no two queens attack each other\*\* — meaning:

- No two queens should share the \*\*same row\*\*, \*\*same column\*\*, or \*\*same diagonal\*\*.

---

## 🔍 \*\*Code Explanation\*\*

### ➤ Function: `placeQueens(...)`

This is the \*\*recursive backtracking\*\* function that tries to place a queen in each row:

\*\*Parameters\*\*:

- `i`: current row

- `cols`: tracks if a column is occupied

- `leftDiagonal`: tracks diagonals from top-left to bottom-right (i - j)

- `rightDiagonal`: tracks diagonals from top-right to bottom-left (i + j)

- `cur`: stores current column positions for queens in each row

\*\*Steps\*\*:

1. If `i == n`, it means all queens have been placed. Return `True`.

2. Loop through each column `j`:

- Check if the column or diagonals are already occupied.

- If safe, mark the column and diagonals as used.

- Recurse to next row (`i + 1`).

- If the next step fails, \*\*backtrack\*\* (undo the changes).

This is where \*\*branch and bound\*\* comes in — by pruning paths early (e.g., skipping columns/diagonals already blocked), you reduce the number of recursive calls.

---

### ➤ Function: `nQueen(n)`

- Initializes tracking arrays:

- `cols`, `leftDiagonal`, and `rightDiagonal` to mark threats

- Calls `placeQueens()` to solve the board.

- If solution exists, builds and returns the board with 'Q's.

- If no solution, returns `None`.

---

### ➤ Function: `printBoard(board)`

- Nicely prints the board if a solution exists.

- Otherwise, says "No solution exists."

---

### ➤ Example:

For `n = 4`, a valid board could be:

```

. Q . .

. . . Q

Q . . .

. . Q .

```

---

## 🎤 Viva Questions with Answers

### 🔹 \*\*N-Queens and Backtracking Basics\*\*

1. \*\*Q:\*\* What is the N-Queen problem?

\*\*A:\*\* It is placing `n` queens on an `n x n` chessboard such that no two queens threaten each other.

2. \*\*Q:\*\* What technique did you use to solve it?

\*\*A:\*\* I used \*\*Backtracking\*\* with \*\*Branch and Bound\*\* optimization.

3. \*\*Q:\*\* What is backtracking?

\*\*A:\*\* Backtracking is a technique to \*\*explore all possibilities\*\* and \*\*revert (backtrack)\*\* if a decision leads to a dead end.

4. \*\*Q:\*\* What is Branch and Bound?

\*\*A:\*\* It's an optimization technique that \*\*prunes branches\*\* of the search space using constraints (like column and diagonal checks in N-Queen).

---

### 🔹 \*\*Algorithm and Code Understanding\*\*

5. \*\*Q:\*\* Why do you use arrays for diagonals?

\*\*A:\*\* To \*\*track conflicts quickly\*\*:

- For any queen at (i, j), left diagonal is `i - j`

- Right diagonal is `i + j`

- These help detect diagonal attacks in \*\*O(1)\*\* time.

6. \*\*Q:\*\* What does the `cur` list store?

\*\*A:\*\* It stores the \*\*column index\*\* of the queen placed in each row.

7. \*\*Q:\*\* What is the base case in `placeQueens()` function?

\*\*A:\*\* If `i == n`, it means \*\*all queens have been placed\*\* successfully.

8. \*\*Q:\*\* What is the time complexity of this solution?

\*\*A:\*\* In worst case, it is \*\*O(N!)\*\*, since for each row we try all columns.

---

### 🔹 \*\*Input/Output and Edge Cases\*\*

9. \*\*Q:\*\* What happens when no solution exists?

\*\*A:\*\* The function returns `None`, and the message "No solution exists." is printed.

10. \*\*Q:\*\* Can the N-Queen problem be solved for all values of `n`?

\*\*A:\*\* No. There are \*\*no solutions for n = 2 and n = 3\*\*. For `n >= 4`, at least one solution exists.

Here are **viva questions and answers** related to your **Assignment-B4**: **N-Queen Problem using Branch and Bound and Backtracking**, suitable for explaining the **Constraint Satisfaction Problem (CSP)** concept and the solution you implemented:

**🔹 General Viva Questions on N-Queen and CSP:**

1. **Q: What is the N-Queen problem?**  
   **A:** The N-Queen problem is placing N queens on an N×N chessboard such that no two queens attack each other — meaning no two queens share the same row, column, or diagonal.
2. **Q: Why is N-Queen a Constraint Satisfaction Problem (CSP)?**  
   **A:** It is a CSP because it requires assigning positions (values) to N queens (variables) under specific constraints (no two queens attacking each other).
3. **Q: What is backtracking?**  
   **A:** Backtracking is a problem-solving technique where we build a solution incrementally and abandon a path ("backtrack") as soon as we determine it won't lead to a valid solution.
4. **Q: What is the role of Branch and Bound in your N-Queen solution?**  
   **A:** The solution uses branch and bound by pruning invalid positions early (branches where placing a queen leads to attack) using constraints (columns and diagonals) to reduce search space.

**🔹 Code-Specific Questions:**

1. **Q: What does the cols, leftDiagonal, and rightDiagonal arrays represent?**  
   **A:**
   * cols[j] = 1 means column j is occupied by a queen.
   * leftDiagonal[i - j + n - 1] tracks '' diagonals.
   * rightDiagonal[i + j] tracks '/' diagonals.
2. **Q: Why are diagonals represented as i + j and i - j + n - 1?**  
   **A:**
   * For the '/' diagonal, all cells where i + j is constant lie on the same diagonal.
   * For the '' diagonal, i - j is constant, but adjusted with + n - 1 to ensure positive indexing.
3. **Q: What is the base case in your recursive function placeQueens?**  
   **A:** The base case is if i == n, meaning all queens are placed safely — a valid solution is found.
4. **Q: Why do we reset the values (backtrack) after the recursive call?**  
   **A:** To explore other possibilities by undoing the current decision, allowing the algorithm to search for alternative valid solutions.
5. **Q: What does the cur list store?**  
   **A:** It stores the column positions where queens are placed for each row. cur[i] = j means row i has a queen at column j.
6. **Q: What does the final board matrix represent?**  
   **A:** It visually represents the chessboard with Q for queens and . for empty cells.

**🔹 Theoretical Questions:**

1. **Q: How many solutions exist for 8-Queen problem?**  
   **A:** There are **92 distinct solutions**, but only **12 unique solutions** if rotations and reflections are considered the same.
2. **Q: What is the time complexity of the N-Queen problem using backtracking?**  
   **A:** In the worst case, it is **O(N!)**, but pruning (via Branch and Bound) reduces this significantly in practice.
3. **Q: Is the N-Queen problem NP-Hard?**  
   **A:** Yes, in generalized form for arbitrary N and additional constraints, it is considered NP-Hard.
4. **Q: How does this implementation differ from brute-force?**  
   **A:** Instead of trying all permutations, it prunes branches early using constraints (greedy bounding), making it more efficient.