Problem Statement:

Design an `n x n` matrix with the first queen already placed. Use backtracking to place the remaining queens and generate the final n-queens matrix.

Problem Overview:

The **N-Queens problem** involves placing `n` queens on an `n x n` chessboard such that no two queens threaten each other. A queen can attack another queen if they share the same row, column, or diagonal. The goal is to place all `n` queens on the board such that none of them can attack each other.

For this problem, the first queen is already placed, and we need to find valid positions for the remaining queens using **backtracking**.

1. Backtracking Approach Overview:

Backtracking is a recursive method that builds a solution incrementally. In the context of the N-Queens problem, backtracking tries to place a queen in each row, one at a time, and checks if the current placement is valid. If a valid solution is found, it continues to the next row; otherwise, it backtracks to the previous row to try a different placement.

Steps to Solve the N-Queens Problem:

1. **Initialize the Chessboard**:
   - Create an `n x n` matrix (chessboard) and place the first queen at the specified position.
2. **Check Validity**:
   - For each row, try to place a queen in one of the columns.
   - A placement is valid if:
     - No queen is already placed in the same column.
     - No queen is already placed in the same diagonal (both left and right).
3. **Recursive Backtracking**:
   - Start placing queens from the second row onwards (since the first queen is already placed).
   - If a valid position is found, place the queen and move to the next row.
   - If no valid position is found in the current row, backtrack to the previous row and try a different column for the queen.
4. **Base Case**:
   - If all queens are placed successfully, print the solution (the board).

Procedure for the N-Queens Problem:

1. **Input**:
   - Size of the chessboard (`n`).
   - Initial position of the first queen (row and column).
2. **Place the First Queen**:

   - Manually place the first queen at the specified position on the chessboard.
3. **Backtracking Algorithm**:
   - Use the recursive backtracking function to place the remaining queens.
   - The function will attempt to place one queen per row, ensuring no two queens threaten each other.
4. **Base Condition**:
   - If all queens are successfully placed, the algorithm terminates with a valid configuration.
5. **Output**:
   - Print the `n x n` matrix with all queens placed such that no two queens can attack each other.

Backtracking Algorithm Pseudocode:

```
function solveNQueens(chessboard, row):
    if row >= n:
        print the chessboard (valid solution found)
        return true

    for each column in 0 to n-1:
        if isSafe(chessboard, row, column):
            place the queen at (row, column)
            if solveNQueens(chessboard, row + 1):
                return true  // found a valid solution
            remove the queen from (row, column)  // backtrack

    return false  // no valid position found in this row
```

Java Code for the N-Queens Problem:

```
java
public class NQueens {
    private static int N;  // size of the board
    private static int[][] board;  // n x n chessboard

    // Function to check if it's safe to place a queen at (row, col)
    static boolean isSafe(int row, int col) {
        // Check this column on upper side
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 1) {
                return false;
            }
        }
        // Check upper diagonal on left side
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }
        // Check upper diagonal on right side
        for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
            if (board[i][j] == 1) {
                return false;
            }
        }
        return true;
    }
    // Backtracking function to solve the N-Queens problem
    static boolean solveNQueens(int row) {
        // Base case: If all queens are placed
        if (row >= N) {
            printBoard();
            return true;  // found one solution
        }
        // Try placing the queen in all columns of this row
        for (int col = 0; col < N; col++) {
            if (isSafe(row, col)) {
                // Place the queen
                board[row][col] = 1;

                // Recur to place the rest of the queens
                if (solveNQueens(row + 1)) {
                    return true;  // found a valid solution
                }
                // If placing queen in this column doesn't lead to a
solution, backtrack
                board[row][col] = 0;
            }
        }
        return false;  // no valid solution for this row
    }
    // Function to print the chessboard
    static void printBoard() {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                System.out.print(board[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println();
    }
    public static void main(String[] args) {
        // Example for N = 4 and first queen placed at (0, 1)
        N = 4;
        board = new int[N][N];

        // Placing the first queen at (0, 1)
        board[0][1] = 1;
        // Start solving from the second row (since the first queen is
already placed)
        if (!solveNQueens(1)) {
            System.out.println("No solution exists");
        }
    }
}
```

Explanation of the Code:

isSafe(): Checks if it is safe to place a queen at position (row, col) by verifying that no other queens are present in the same column, left diagonal, or right diagonal.

- **`solveNQueens()`**:

    - A recursive function that attempts to place a queen in each column of the current row. If a valid placement is found, it moves to the next row.
    - If no valid placement is found for a row, it backtracks by removing the queen from the previous row and tries the next column.
- **`printBoard()`**: Prints the final chessboard configuration where all queens are safely placed.
- **Main Function**:
    - Initializes the board with size `N`.
    - Places the first queen at a predefined position, e.g., `(0, 1)`.
    - Calls the `solveNQueens()` function starting from the second row to place the remaining queens.

Sample Output:

```
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```

In this solution:

- The first queen is manually placed at position `(0, 1)`.
- The remaining queens are placed using backtracking to ensure no queens can attack each other.

Backtracking Strategy Recap:

- The backtracking algorithm places queens row by row.
- After placing a queen in a row, it checks for a valid position for the queen in the next row.
- If no valid position exists in the next row, the algorithm backtracks by removing the last placed queen and trying the next available column.

Time Complexity:

- The time complexity of the backtracking solution is `O(n!)`, where `n` is the size of the board. This is because for each queen, we attempt to place it in one of `n` columns, leading to `n` possibilities in the first row, `n-1` in the second, and so on.

Space Complexity:

The space complexity is `O(n^2)` due to the storage of the chessboard as an `n x n` matrix.

Conclusion:

This approach efficiently solves the N-Queens problem by incrementally placing queens and backtracking when conflicts arise. The first queen is placed at a predefined location, and backtracking is used to place the remaining queens without conflicts.