Practical 8: Backtracking - N Queens

Aim

Solve the $\boldsymbol{n}$ queens' problem using backtracking. Here, the task is to place $\boldsymbol{n}$ chess queens on an n x n board so that no two queens attack each other.

Algorithm

1. Initialize an empty chessboard of size N x N.
2. Start with the leftmost column and place a queen in the first row of that column.
3. Move to the next column and place a queen in the first row of that column.
4. Repeat step 3 until either all N queens have been placed or it is impossible to place a queen in the current column without violating the rules of the problem.
5. If all N queens have been placed, print the solution.
6. If it is not possible to place a queen in the current column without violating the rules of the problem, backtrack to the previous column.
7. Remove the queen from the previous column and move it down one row.
8. Repeat steps 4-7 until all possible configurations have been tried.

Program

```
import java.util.Scanner;

public class NQueenProblem {
    static int N;

    // print the final solution matrix
    static void printSolution(int board[][]) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                System.out.print(" " + board[i][j] + " ");
            }
            System.out.println();
        }
    }

    // function to check whether the position is safe or not
    static boolean isSafe(int board[][], int row, int col) {
        int i, j;
        // Check for Same Row
        for (i = col - 1; i >= 0; i--) {
            if (board[row][i] == 1) {
                return false;
            }
        }
        // Check for Upper Diagonal
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }
        // Check for Lower Diagonal
        for (i = row, j = col; j >= 0 && i < N; i++, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }
```

```
        return true;
    }

    static boolean solveNQueen(int board[][], int col) {
        if (col >= N) {
            return true;
        }
        for (int i = 0; i < N; i++) {
            if (isSafe(board, i, col)) {
                board[i][col] = 1;

                if (solveNQueen(board, col + 1)) {
                    return true;
                }
                // Backtrack if the above condition is false
                board[i][col] = 0;
            }
        }
        return false;
    }

    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter size of board: ");
        N = sc.nextInt();
        sc.close();

        int[][] board = new int[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                board[i][j] = 0;
            }
        }

        if (!solveNQueen(board, 0)) {
            System.out.print("Solution does not exist");
            return;
        }

        printSolution(board);

    }
}
```

Output:

```
Enter size of board: 5
 1  0  0  0  0
 0  0  0  1  0
 0  1  0  0  0
 0  0  0  0  1
 0  0  1  0  0
```

Analysis of Algorithm

Time Complexity:

The time complexity of this code is **O(N!)** because for each column, we check all possible rows to place the queen, which means we have N choices in the first column, N-2 choices in the second column, N-4 choices in the third column, and so on. Therefore, the total number of possibilities will be **N * (N-2) * (N-4) * ... * 1**, which is equivalent to N!.

Space Complexity:

The space complexity of this code is **$O(N^2)$** because we are using a **two-dimensional array of size N x N** to represent the chessboard. Additionally, we are using recursive function calls that will occupy memory on the stack, and the maximum depth of the recursion tree will be N. Therefore, the space complexity will be $O(N^2 + N)$, which is equivalent to **$O(N^2)$**.