Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
Department of Master of Computer Application

Experiment	8
Aim	To implement 8 Queens Problem
Objective	1) Learn N queen problem 2) Implement 8 queen problem 3) Derive the Time complexity 8 queen problem
Name	Vivek Tiwari
UCID	2023510059
Class	FY MCA
Batch	C
Date of Submission	02042024

Algorithm and Explanation of the technique used	**Algorithm:** 1. Initialize Variables: • Initialize a variable `size` to represent the size of the chessboard (N). • Initialize an empty list `solutions` to store the valid solutions. 2. Define `solveNQueens` Function: Create an empty chessboard (`emptyBoard`) of size `n x n`. Call the `backtrack` function with initial parameters: `row` = 0 (starting row) `diagonals` = empty set (to track diagonals) `antiDiagonals` = empty set (to track antidiagonals) `cols` = empty set (to track occupied columns) `emptyBoard` (initial state of the board) Return the list of solutions. 3. Define `backtrack` Function: `row`: current row `diagonals`: set of diagonals under attack `antiDiagonals`: set of antidiagonals under attack `cols`: set of occupied columns `state`: current state of the chessboard

	• If `row` equals `size`, add the current state of the board to `solutions` and return. • Iterate over each column (`col`) in the current row (`row`): • Calculate the diagonal and antidiagonal indices (`currDiagonal` and `currAntiDiagonal`) corresponding to `(row, col)`. • If `col` is in `cols` or `currDiagonal` is in `diagonals` or `currAntiDiagonal` is in `antiDiagonals`, continue to the next column. • Add `col` to `cols`, `currDiagonal` to `diagonals`, and `currAntiDiagonal` to `antiDiagonals`. • Place a queen at `(row, col)` in `state`. • Recur by calling `backtrack` with updated parameters for the next row. • Remove `col` from `cols`, `currDiagonal` from `diagonals`, and `currAntiDiagonal` from `antiDiagonals`. • Remove the queen from `(row, col)` in `state`.

N-Queens Problem

Step 1:

	0	1	2	3
0	Q_1	×	×	×
1	×	×		
2	×		×	
3	×			×

Step 2:

	0	1	2	3
0	Q_1	×	×	×
1	×	×	Q_2	×
2	×	×	×	×
3	×			×

Step 3:

At row 2, there are no cells safe for Q_3 so back track & remove Q_2 from (1,2).

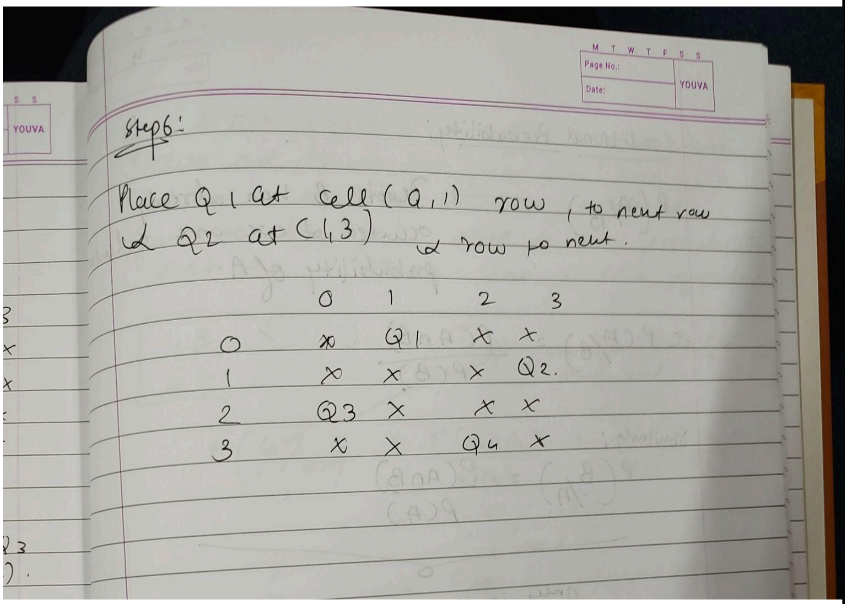
Step 4:

There is still a safe space in (1,3) for Q_2.

	0	1	2	3
0	Q_1	×	×	×
1	×	×	×	Q_2
2	×	Q_3	×	×
3	×	×	×	×

Step 5:

No cell to place Q_4 at Row 3, so back-track & remove Q_3 from row 2, again no place in R2 so remove Q_2 & then Q_1.



Program(Co de)

```
import java.util.*;

public class lab8queens {
    private int size;
  private List<List<String>> solutions = new ArrayList<>();

    public List<List<String>> solveNQueens(int n) {
        size = n;
        char emptyBoard[][] = new char[size][size];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                emptyBoard[i][j] = '.';
            }
        }

        backtrack(0, new HashSet<>(), new HashSet<>(),
new HashSet<>(), emptyBoard);
        return solutions;
    }

        private List<String> createBoard(char[][]
            state) { List<String> board = new
                    ArrayList<>();
        for (int row = 0; row < size; row++) {
          String current_row = new String(state[row]);
            board.add(current_row);
        }

        return board;
    }
```

```java
    private void backtrack(int row, Set<Integer> diagonals,
Set<Integer> antiDiagonals, Set<Integer> cols, char[][]
state)  {
        if (row == size) {
            solutions.add(createBoard(state));
            return;
        }

        for (int col = 0; col < size; col++) {
            int currDiagonal = row col;
            int currAntiDiagonal = row + col;
            if (cols.contains(col) ||
diagonals.contains(currDiagonal) ||
antiDiagonals.contains(currAntiDiagonal)) {
                continue;
            }

            cols.add(col);
            diagonals.add(currDiagonal);
            antiDiagonals.add(currAntiDiagonal);
            state[row][col] = 'Q';

            backtrack(row + 1, diagonals, antiDiagonals,
cols,  state);

            cols.remove(col);
            diagonals.remove(currDiagonal);
           antiDiagonals.remove(currAntiDiagonal);
            state[row][col] = '.';
        }
    }

    public static void main(String[] args) {
        lab8queens obj = new lab8queens();
        int n = 8; // Change this to your desired board
        size int count=0;
        List<List<String>> solutions =
        obj.solveNQueens(n); for (List<String> solution
        : solutions) {
            printBoard(solution);
            System.out.println("");
            System.out.println();
            count++;
        }
       System.out.println("Total number of solutions:
"+count);

    }
```

```java
    private static void printBoard(List<String>
        board) { int n = board.size();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.print(board.get(i).charAt(j) + "
");
            }
            System.out.println();
        }
    }
}
```

Output

```
----------

. . . . . . . Q
. . . Q . . . .
Q . . . . . . .
. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
. . . . Q . . .
----------

Total number of solutions: 92
```

Justification of the complexity calculated

1. Backtracking Function:
    - At each row, the algorithm considers placing a queen in one of the (n) columns.
    - However, it prunes branches by checking for conflicts before placing a queen in a particular column.
    - The number of choices for placing the queen decreases by one at each subsequent row.
    - Therefore, the total number of possibilities explored is at most (n!).
2. Time Complexity Analysis:
    - In the worst case, the algorithm explores all (n!) possible permutations of queen placements on the board.
    - This is because the algorithm systematically explores all possible configurations while adhering to the constraints of the problem. • Pruning is used to avoid exploring invalid configurations, but it does not change the overall time complexity.
    - As (n) increases, the number of possible configurations grows factorially ((n!)), resulting in exponential growth in the time taken to find solutions.

Therefore, the justified time complexity of the provided algorithm is (O(n!)). This means that as the size of the chessboard (n) increases, the time taken to find solutions grows factorially ((n!)).

Conclusion	***Applications:*** Chess Strategies: N-Queens models chessboard setups, informing strategies to place queens without attacking each other, relevant for chess algorithms and game analysis. Scheduling Challenges: It translates to scheduling tasks without conflicts, useful in educational timetabling, conference scheduling, and shift planning. VLSI Layout: Represents VLSI chip design, optimizing component placement for performance and manufacturing efficiency. Algorithm Evaluation: Acts as a benchmark for testing optimization algorithms like genetic algorithms. Educational Tool: Helps teach problem-solving techniques like backtracking and recursion due to its simplicity and relevance.