**Assignment 6**

**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

**Title:**

Assignment Based on Backtracking to Implement N Queen Problem (Determine Time and space complexity)

**Theory:**

Here is the full assignment based on Backtracking to implement the N Queen Problem, including theory, objective, explanation, pseudocode, Java code with user input and print statements, example output, and time/space complexity analysis.

## Objective

To implement the N Queen problem using the backtracking technique and determine its time and space complexity.

## Theory and Explanation

The N Queen problem is a classic combinatorial problem where the goal is to place N queens on an N×NN \times NN×N chessboard so that no two queens threaten each other. This means that no two queens can be placed in the same row, the same column, or on the same diagonal.

Backtracking is a systematic way to iterate through all possible configurations and reject those that violate constraints early. It tries to place a queen in a valid position in one column, then moves to the next column, and so on recursively. If a conflict arises, it backtracks by removing a queen and trying another position.

The problem exemplifies how backtracking helps in constraint satisfaction problems by searching and pruning the solution space.

## Important Points

* No two queens can share the same row, column, or diagonal.
* Attempt to place one queen per column moving left to right recursively.
* If no position in a column is safe, backtrack to previous column.
* Use a 2D board to mark queen placements (1 for queen, 0 for empty).
* Continue until all queens are placed or no solution exists.

## Pseudocode

text

function solveNQueen(board, col, N):

if col >= N:

return true

for row from 0 to N-1:

if isSafe(board, row, col, N):

board[row][col] = 1

if solveNQueen(board, col + 1, N):

return true

board[row][col] = 0 // backtrack

return false

function isSafe(board, row, col, N):

for i from 0 to col:

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

return false

for i, j = row, col to 0,0 (diagonal left-up):

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

return false

for i, j = row, col to N-1,0 (diagonal left-down):

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

return false

return true

## Example Output

Input:

text

Enter the size of the board (N):

4

Output:

text

Solution exists:

0 1 0 0

0 0 0 1

1 0 0 0

0 0 1 0

**Code:**

import java.util.Scanner;

public class NQueen {

    public static boolean isSafe(int[][] board, int row, int col, int N) {

        for (int i = 0; i < col; i++)

            if (board[row][i] == 1)

                return false;

        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

            if (board[i][j] == 1)

                return false;

        for (int i = row, j = col; i < N && j >= 0; i++, j--)

            if (board[i][j] == 1)

                return false;

        return true;

    }

    public static boolean solveNQueen(int[][] board, int col, int N) {

        if (col >= N)

            return true;

        for (int i = 0; i < N; i++) {

            if (isSafe(board, i, col, N)) {

                board[i][col] = 1;

                if (solveNQueen(board, col + 1, N))

                    return true;

                board[i][col] = 0;

            }

        }

        return false;

    }

    public static void printBoard(int[][] board, int N) {

        for (int i = 0; i < N; i++) {

            for (int j = 0; j < N; j++) {

                System.out.print(board[i][j] + " ");

            }

            System.out.println();

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of the board (N):");

        int N = sc.nextInt();

        int[][] board = new int[N][N];

        if (solveNQueen(board, 0, N)) {

            System.out.println("Solution exists:");

            printBoard(board, N);

        } else {

            System.out.println("Solution does not exist");

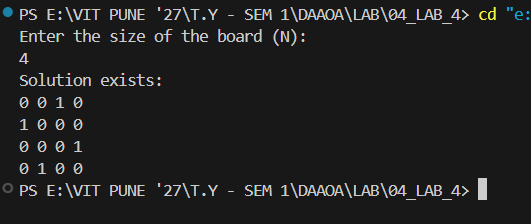
        }

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

## Time Complexity Analysis

* In worst case, the algorithm tries to place queens in all rows for each of the N columns.
* Possible configurations to explore: NNN^NNN.
* Pruning reduces number of checks, but worst case remains O(N!)O(N!)O(N!) due to permutations of placing queens per row without conflicts.
* Hence time complexity is approximately **O(N!)O(N!)O(N!)**.

## Space Complexity Analysis

* Uses an N×NN \times NN×N board to store queen positions.
* Recursion stack depth can go up to N (one for each column).
* Total space complexity is **O(N2)O(N^2)O(N2)** for the board plus recursion call stack, which is **O(N)O(N)O(N)**.
* Overall space complexity is **O(N2)O(N^2)O(N2)**.

**Pseudocode with Complexity Comments**

text

FUNCTION knapsack(W, wt, val, n)

DECLARE 2D array K of size (n+1) x (W+1) // Space: +(n+1)\*(W+1) = O(n\*W)

FOR i FROM 0 TO n // Time: +n+1

FOR w FROM 0 TO W // Time: +(W+1) per i; Total: \*n\*W

IF i == 0 OR w == 0 // Time: +1 per iteration

K[i][w] ← 0 // Time: +1

ELSE IF wt[i-1] <= w // Time: +1

K[i][w] ← MAX(val[i-1] + K[i-1][w - wt[i-1]], K[i-1][w]) // Time: +1 (max and addition)

ELSE

K[i][w] ← K[i-1][w] // Time: +1

ENDIF

ENDFOR

ENDFOR

RETURN K[n][W] // Time: +1 (return)

ENDFUNCTION

FUNCTION main

DECLARE scanner // Space: +1

PRINT "Enter number of items:" // Time: +1

INPUT n // Time: +1

DECLARE arrays val[n], wt[n] // Space: +n each = +2n total

PRINT "Enter value and weight of each item:" // Time: +1

FOR i FROM 0 TO n-1 // Time: +n

INPUT val[i], wt[i] // Time: +1 per read

ENDFOR

PRINT "Enter the capacity of the knapsack:" // Time: +1

INPUT W // Time: +1

maxProfit ← knapsack(W, wt, val, n) // Time: O(n\*W), Space: O(n\*W)

PRINT "Maximum profit that can be obtained = " + maxProfit // Time: +1

CLOSE scanner // Time: +1

ENDFUNCTION

**Complexity Explanation**

* **Time Complexity:** The nested loops iterate over each item (n) and capacity (W), so overall the time complexity is O(n×W)O(n \times W)O(n×W).
* **Space Complexity:** The 2D DP table KKK requires O(n×W)O(n \times W)O(n×W) space to store intermediate results.
* Input and output operations take linear time and constant extra space outside the storage arrays.
* All constant time operations (+1) occur within nested loops to build the solution table.

**Conclusion**

In this lab exercise, we learned how to implement 0-1 Knapsack problem using Dynamic Programming.