**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

# Experiment Number: 04

**Title:** Assignment Based on Backtracking. (Implement N- Queen problem)

|  |  |  |  |
| --- | --- | --- | --- |
| **Title of Experimentation** | **CO**  **Mapping** | **CO-Statements** | **PO**  **Mapping** |
| Assignment Based on Backtracking. (Implement N- Queen problem) | CO1,CO2,CO3 | To establish NP-completeness of some decision problems, grasp the significance of the notion of NP-completeness and its relationship with intractability of the decision problems. |  |

### **Theory**

The **N-Queen problem** is a classical combinatorial problem:

* Place N queens on an N × N chessboard.
* Queens must be placed so that no two queens attack each other.
* A queen can attack another if they share the same **row, column, or diagonal**.

**Backtracking** is used:

* Place queens one by one in different rows.
* If placing a queen leads to a valid state, proceed to the next row.
* If a conflict occurs, backtrack and try the next column.

### **Input:**

* A single integer N → size of chessboard (and number of queens).

### **Output:**

* One or more valid configurations of N queens on the board.
* (Each solution shows positions where queens are placed safely.)

### **Objective of Experiment:**

To understand and implement **Backtracking** by solving the **N-Queen problem**, demonstrating how systematic trial and error with recursive backtracking helps solve constraint satisfaction problems.

### **Algorithm (Backtracking):**

1. Start with the first row.
2. Try placing a queen in each column of the current row.
3. If placing queen is **safe** (no other queen in same column/diagonal), place it.
4. Recurse to the next row.
5. If all queens are placed → print solution.
6. If no valid column exists in current row → backtrack (remove queen from previous row and try next possibility).

### **Pseudo Code:**

function solveNQueen(N):

create board[N][N] initialized to 0

if placeQueen(board, 0, N) == false:

print "No solution exists"

else:

print board

function placeQueen(board, row, N):

if row == N:

return true // all queens placed

for col = 0 to N-1:

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

board[row][col] = 1

if placeQueen(board, row+1, N):

return true

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

return false

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

check column above

check upper-left diagonal

check upper-right diagonal

if no conflicts → return true

else → return false

### **Java Implementation:**

import java.util.Scanner;

public class NQueen {

static int N;

// Function to print solution

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] == 1 ? "Q " : ". "));

}

System.out.println();

}

System.out.println();

}

// Check if a queen can be placed at board[row][col]

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

// Check column

for (int i = 0; i < row; i++)

if (board[i][col] == 1)

return false;

// Check upper-left diagonal

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

if (board[i][j] == 1)

return false;

// Check upper-right diagonal

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

if (board[i][j] == 1)

return false;

return true;

}

// Recursive function to solve N-Queen problem

static boolean solveNQUtil(int board[][], int row) {

if (row == N) {

printSolution(board);

return true;

}

boolean res = false;

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

if (isSafe(board, row, col)) {

board[row][col] = 1;

res = solveNQUtil(board, row + 1) || res;

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

}

}

return res;

}

static void solveNQ() {

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

if (!solveNQUtil(board, 0)) {

System.out.println("No solution exists");

}

}

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter value of N: ");

N = sc.nextInt();

solveNQ();

sc.close();

}

}

**Input:**

Enter value of N: 4

**Output:** (One possible solution)

. Q . .

. . . Q

Q . . .

. . Q .

. . Q .

Q . . .

. . . Q

. Q . .

### **Flowchart :**

* **Start**
* Input N
* Initialize empty board[N][N]
* Call recursive function placeQueen(row)
  + If row == N → print solution
  + Else try placing queen in each column:
    - If safe → place queen → recurse → if fails → backtrack
* Repeat until all solutions are found
* **Stop**

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

-

**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");

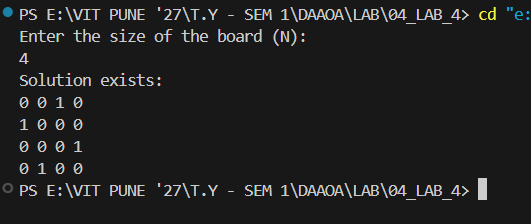
        }

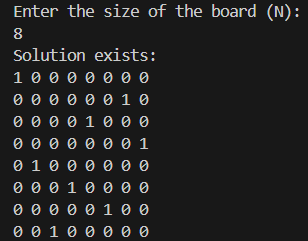
        sc.close();

    }

}

**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: N^N
* Pruning reduces number of checks, but worst case remains O(N!) due to permutations of placing queens per row without conflicts.
* Hence time complexity is approximately **O(N!)**

## Space Complexity Analysis

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

**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)
* **Space Complexity:** The 2D DP table K requires 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.