Name:	**Saad Wasi Ahmed Siddiqui**	**Slot:**	**L13+L14**
Professor:	**Dr. Indira B**	**Course Code:**	**BCSE204P**
Class Nbr:	**CH2025260101604**	**Semester:**	**Fall Semester 2025-26**
Reg. Nbr:	**23BRS1113**	**Course Title:**	**Design and Analysis of Algorithms Lab**

Lab Assignment 5: N-Queens Problem using the backtracking algorithm

Aim: The aim of this problem is to design and implement a backtracking algorithm that solves the classic N-Queens Problem. The task is to place N queens on an N × N chessboard such that no two queens attack each other, i.e., no two queens share the same row, column, or diagonal. The program should generate and display all possible solutions for given values of N (in this case, N=4 and N=8) and print the total number of solutions.

Algorithm (Backtracking Approach)

The N-Queens problem can be solved using backtracking, which is a depth-first search approach to systematically explore possible queen placements and backtrack whenever a conflict occurs.

Step-by-step Algorithm:

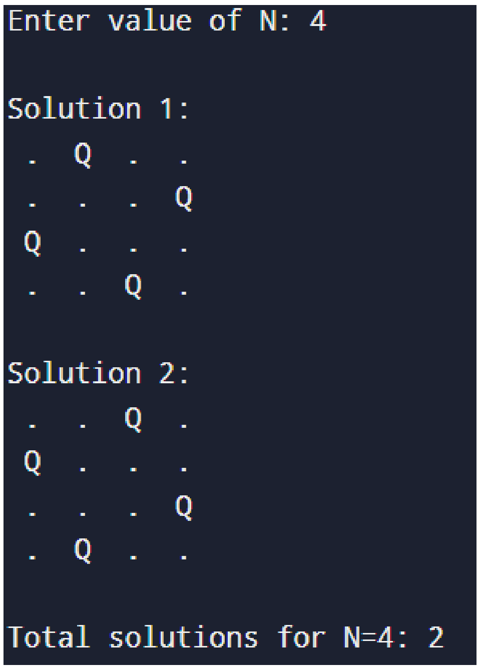
1. Start with an empty chessboard of size N × N.
2. Place queens row by row:
   - Start with the first row (row 0).
   - For each column in the current row, check if placing a queen is safe:
     - No other queen is already placed in the same column.
     - No other queen is already placed on the same left diagonal.
     - No other queen is already placed on the same right diagonal.

3. If it is safe, place the queen at that position and move to the next row recursively.
4. If a row has no valid column for placing a queen, backtrack:
    - Remove the last placed queen and try the next column in the previous row.
5. If all queens are placed successfully (row = N):
    - A valid solution is found. Store or print this solution.
6. Continue exploring until all possibilities are exhausted.
7. Print all solutions and the total count.

Source Code:

```c
#include <stdio.h>
#include <stdlib.h>

#define N 8

int board[N];
int solutionCount = 0;
int isSafe(int row, int col) {
    for (int i = 0; i < row; i++) {
        if (board[i] == col ||
            abs(board[i] - col) == abs(i - row)) {
            return 0;
        }
    }
    return 1;
}
void printSolution(int n) {
    printf("\nSolution %d:\n", ++solutionCount);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (board[i] == j)
                printf(" Q ");
            else
                printf(" . ");
        }
        printf("\n");
    }
}

void solveNQueens(int row, int n) {
    if (row == n) {
        printSolution(n);
        return;
    }
    for (int col = 0; col < n; col++) {
        if (isSafe(row, col)) {
            board[row] = col;
            solveNQueens(row + 1, n);
        }
    }
}

int main() {
    int n;
    printf("Enter value of N: ");
    scanf("%d", &n);

    if (n < 1 || n > N) {
        printf("Please enter a value of N between 1 and %d.\n", N);
        return 0;
    }

    solveNQueens(0, n);
    printf("\nTotal solutions for N=%d: %d\n", n, solutionCount);

    return 0;
}
```

Output:



```
Enter value of N: 4

Solution 1:
 .  Q  .  .
 .  .  .  Q
 Q  .  .  .
 .  .  Q  .

Solution 2:
 .  .  Q  .
 Q  .  .  .
 .  .  .  Q
 .  Q  .  .

Total solutions for N=4: 2
```

Time Complexity Analysis

- At each row, we try placing a queen in N possible columns.
- For each attempted placement, we must check if it is safe, which takes **O(N)** time in the worst case (checking previous rows and diagonals).
- In the worst case, we explore almost the entire search tree.

Thus, the time complexity is approximately: O(N!)