**Name: Khushi Yaduka**

**Roll No: 21BCP417**

**DIV-6** **G-12**

**ASSIGNMENT-8**

**Aim:** Use Backtracking to solve N-Queens Problem

**Theory:**

The N Queens problem is a well-known problem in computer science and mathematics. It is a classic problem of placing N chess queens on an N x N chessboard so that no two queens attack each other.

The N Queens problem can be solved using different algorithms, including backtracking, recursive, and iterative algorithms. The backtracking algorithm is one of the most commonly used algorithms for solving the N Queens problem. It is an efficient algorithm that uses a depth-first search approach to explore all possible solutions of the problem. The backtracking algorithm works by placing queens in different cells of the chessboard and checking if they are in conflict with each other. If they are not in conflict, the algorithm moves to the next row and places the next queen. If all queens are placed on the board, the algorithm returns true. If a conflict is detected, the algorithm backtracks to the previous cell and tries a different position for the queen.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Function to display the chessboard with queens placed

void display\_board(int queens[], int n) {

//outer loop for rows

for (int i = 0; i < n; i++) {

//inner loop for columns

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

if (queens[i] == j) {

printf(" Q ");

} else {

printf(" - ");

}

}

printf("\n");

}

printf("\n");

}

// Function to check if the current queen placement is valid

bool is\_valid(int queens[], int row, int col) {

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

if (queens[i] == col || abs(queens[i] - col) == abs(i - row)) {

// A queen can attack another queen if they are in the same column or diagonal

return false;

}

}

// If no other queen can attack the current queen, then it is a valid placement

return true;

}

// Function to solve the N queens problem using backtracking

void solve\_n\_queens(int queens[], int row, int n, int\* count) {

if (row == n) {

// If all queens have been placed on the board, we have found a valid solution

(\*count)++;

printf("Solution %d:\n", \*count);

display\_board(queens, n);

return;

}

// Try placing the queen in each column of the current row

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

if (is\_valid(queens, row, col)) {

queens[row] = col;

// If the current queen placement is valid, move on to the next row

solve\_n\_queens(queens, row + 1, n, count);

queens[row] = -1;

// Backtrack and try the next column if the current placement does not lead to a solution

}

}

}

int main() {

int n, count = 0;

printf("Enter the number of queens: ");

scanf("%d", &n);

// The queens array stores the column number of the queen in each row

int queens[n];

for (int i = 0; i < n; i++) {

queens[i] = -1;

}

// Start placing queens from the first row

solve\_n\_queens(queens, 0, n, &count);

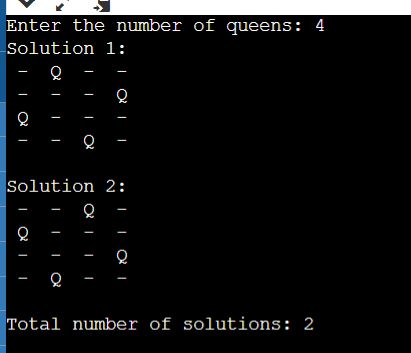
//displaying the total number of solutions

printf("Total number of solutions: %d\n", count);

return 0;

}

**OUTPUT:**



**TIME COMPLEXITY ANALYSIS:**

* The time complexity of the N-Queens problem using backtracking is O(N!), where N is the number of queens or the size of the chessboard.
* In the worst case scenario, the backtracking algorithm needs to try all possible configurations of the N queens on the N x N chessboard. At the first row, there are N possible choices for placing the queen, at the second row there are (N-1) choices remaining, at the third row there are (N-2) choices remaining, and so on. Therefore, the total number of configurations to be checked is N x (N-1) x (N-2) x ... x 1 = N!.
* Although this upper bound on the time complexity is very high, in practice the algorithm often finds a solution much more quickly due to pruning the search space using the is\_valid function. The is\_valid function reduces the number of possibilities by checking if the placement of the queen in the current row and column conflicts with any previously placed queens in the same column or diagonal. This pruning of the search space can significantly reduce the number of recursive calls that the algorithm has to make.