|  |
| --- |
| **Experiment No. 9** |
| **To implement N -Queen problem** |
| Date of Performance: |
| Date of Submission: |

## Experiment No. 9

**Title**: To implement N -Queen problem

**Aim**: To study, implement and Analyze N queen Problem.

**Objective:** To introduce the N queen Problem and analyzing algorithms

**Theory**:

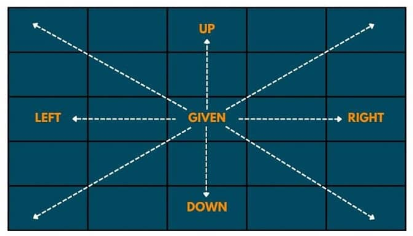
Backtracking is a problem-solving technique that involves recursively trying out different solutions to a problem, and backtracking or undoing previous choices when they don’t lead to a valid solution. It is commonly used in algorithms that search for all possible solutions to a problem, such as the famous eight-queens puzzle. Backtracking is a powerful and versatile technique that can be used to solve a wide range of problems.

The N Queen problem demands us to place N queens on a N x N chessboard so that no queen can attack any other queen directly.

**Problem Statement:**

Find out all the possible arrangements in which N queens can be seated in each row and each column so that all queens are safe.

The queen moves in 8 directions and can directly attack in these 8 directions only.



#### Example:

**4 - Queen Problem:**

* This problem demands us to put 4 queens on 4 X 4 chessboard in such a way that no 2 or more queens can be placed in the same diagonal or row or column.
* The idea is to place queens one by one in different columns, starting from the leftmost column.
* When we place a queen in a column, we check for clashes with already placed queens.
* In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution.
* If we do not find such a row due to clashes, then we backtrack and return **false**.

**A diagram of a network

Description automatically generatedSolution to 4 Queen Problem**

#### Algorithm and Complexity:

**Code:**

**Algorithm:**

Step 1: Start in the leftmost column.

Step 2: If all queens are placed return true.

Step 3: Try all rows in the current column. Do the following for every row.

Step 3.1: If the queen can be placed safely in this row.

Step 3.1.1: Then mark this [row, column] as part of the solution and recursively

check if placing queen here leads to a solution.

Step 3.1.2: If placing the queen in [row, column] leads to a solution then

return true.

Step 3.1.3: If placing queen doesn’t lead to a solution then unmark this [row,

column] then backtrack and try other rows.

Step 4: If all rows have been tried and valid solution is not found return false to trigger

backtracking.

**Time Complexity -  O(N!)**

* For the first row, we check N columns; for the second row, we check the N - 1 column and so on. Hence, the time complexity will be N \* (N-1) \* (N-2) …. i.e. O(N!)

**Space Complexity - O(N^2)**

* O(N^2), where ‘N’ is the number of queens.
* We are using a 2-D array of size N rows and N columns, and also, because of Recursion, the recursive stack will have a linear space here. So, the overall space complexity will be O(N^2).

**Program:**

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

void printSolution(int \*\*board, int N) {

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

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

if (board[i][j])

printf("Q ");

else

printf(". ");

}

printf("\n");

}

}

bool isSafe(int \*\*board, int row, int col, int N) {

int i, j;

for (i = 0; i < col; i++)

if (board[row][i])

return false;

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

if (board[i][j])

return false;

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

if (board[i][j])

return false;

return true;

}

bool solveNQUtil(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 (solveNQUtil(board, col + 1, N))

return true;

board[i][col] = 0;

}

}

return false;

}

bool solveNQ(int N) {

int \*\*board = (int \*\*)malloc(N \* sizeof(int \*));

if (board == NULL) {

printf("Memory allocation failed\n");

return false;

}

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

board[i] = (int \*)malloc(N \* sizeof(int));

if (board[i] == NULL) {

printf("Memory allocation failed\n");

// Free memory allocated so far

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

free(board[j]);

free(board);

return false;

}

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

board[i][j] = 0; // Initialize board to 0

}

if (solveNQUtil(board, 0, N) == false) {

printf("Solution does not exist");

// Free memory

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

free(board[i]);

free(board);

return false;

}

printSolution(board, N);

// Free memory

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

free(board[i]);

free(board);

return true;

}

int main() {

int N;

printf("Enter the size of the board: ");

scanf("%d", &N);

if (N <= 0) {

printf("Invalid board size\n");

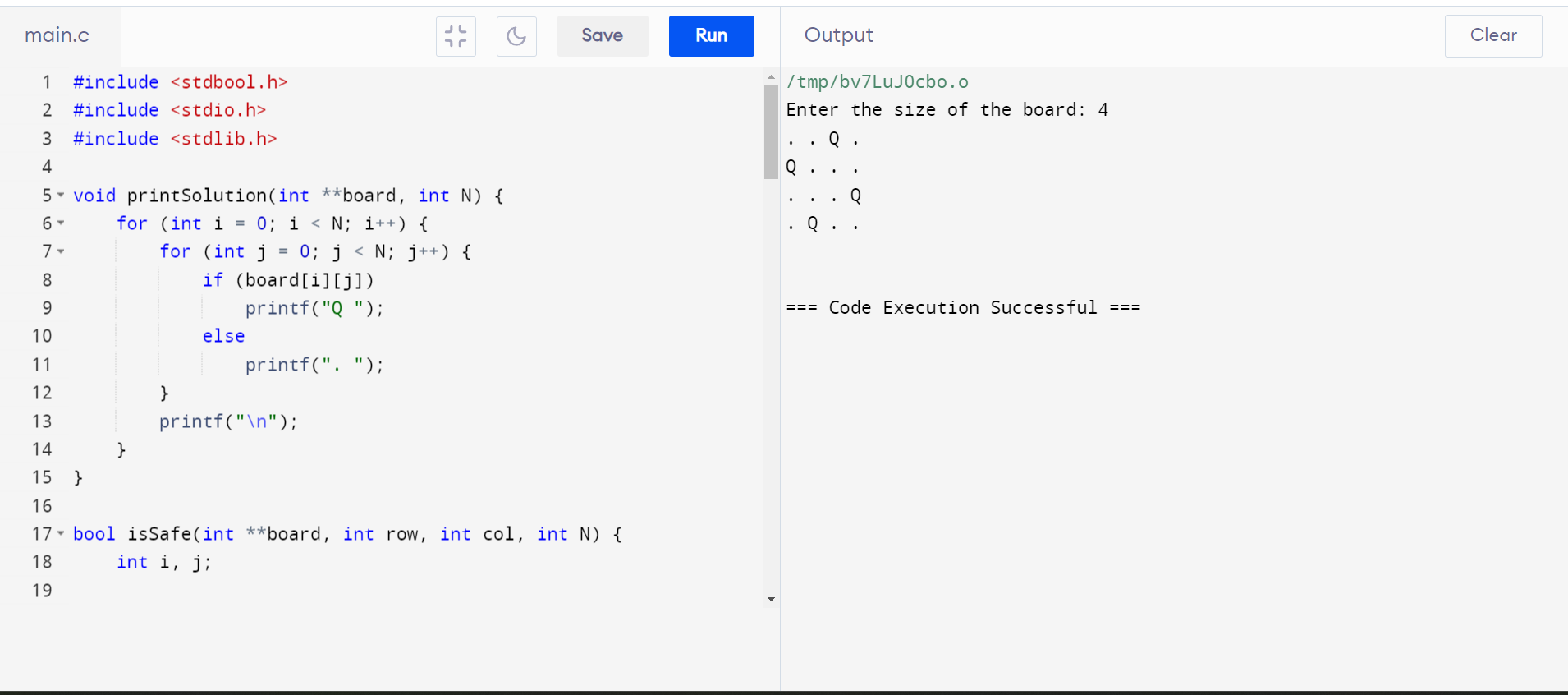
return 1;

}

solveNQ(N);

return 0;

}

**Output:**

**Conclusion:**

**The N-Queens problem challenges placing N chess queens on an N×N board without them threatening each other. It's computationally complex, with solutions found using algorithms like backtracking or genetic algorithms. While abstract, it's valuable for teaching problem-solving and has applications in computer science and AI.**