Experiment No. 9
To implement N -Queen problem
Name : Vaidehi D. Gadag
Branch/Div.: Comps-1 (C47)
Date of Performance: 28/03/2024
Date of Submission: 04/04/2024

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:

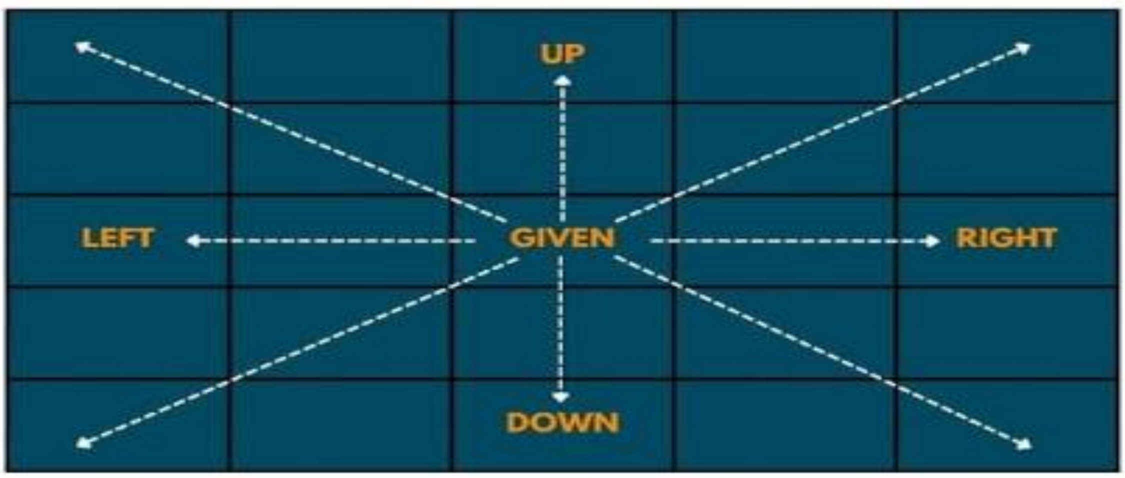
Backtracking is a problem-solving technique that involves recursively trying out different solutions to a problem, and backtracking or undoing previous choices when they don't lead to a valid solution. It is commonly used in algorithms that search for all possible solutions to a problem, such as the 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.

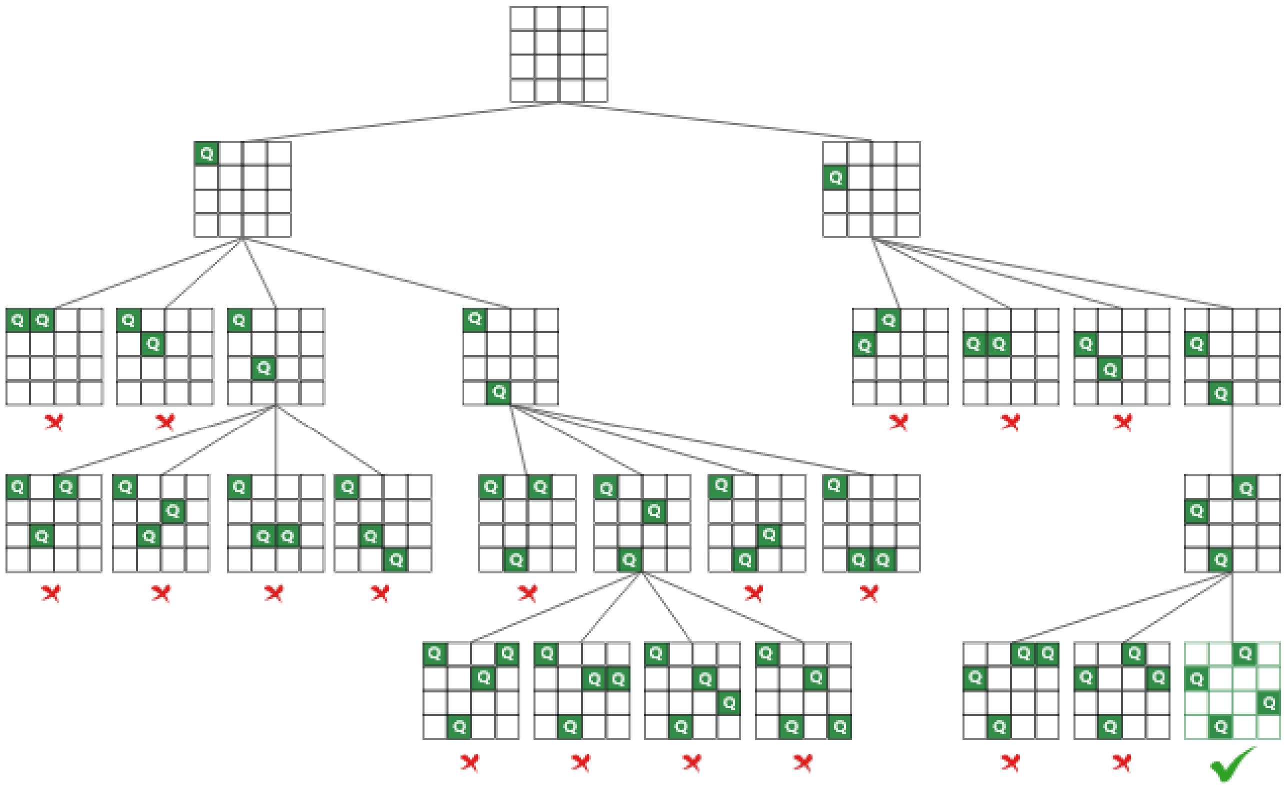


Example:

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**.

Solution to 4 Queen Problem



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!)

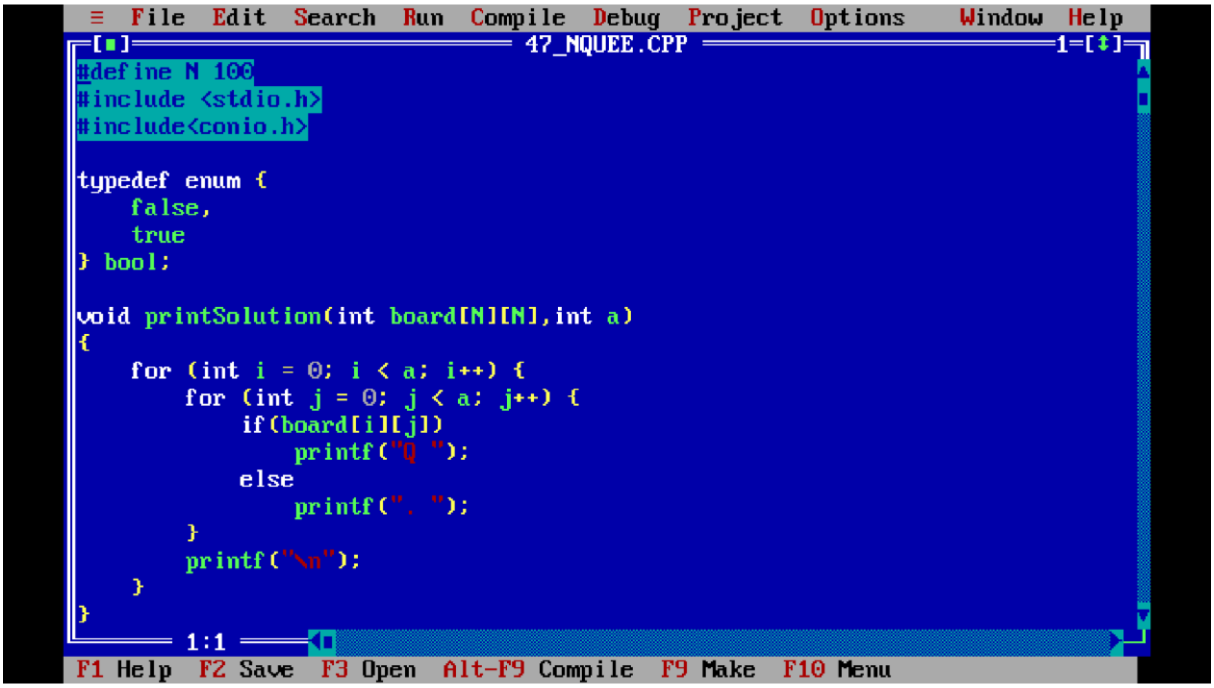
Space Complexity - O(N^2)

- 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).

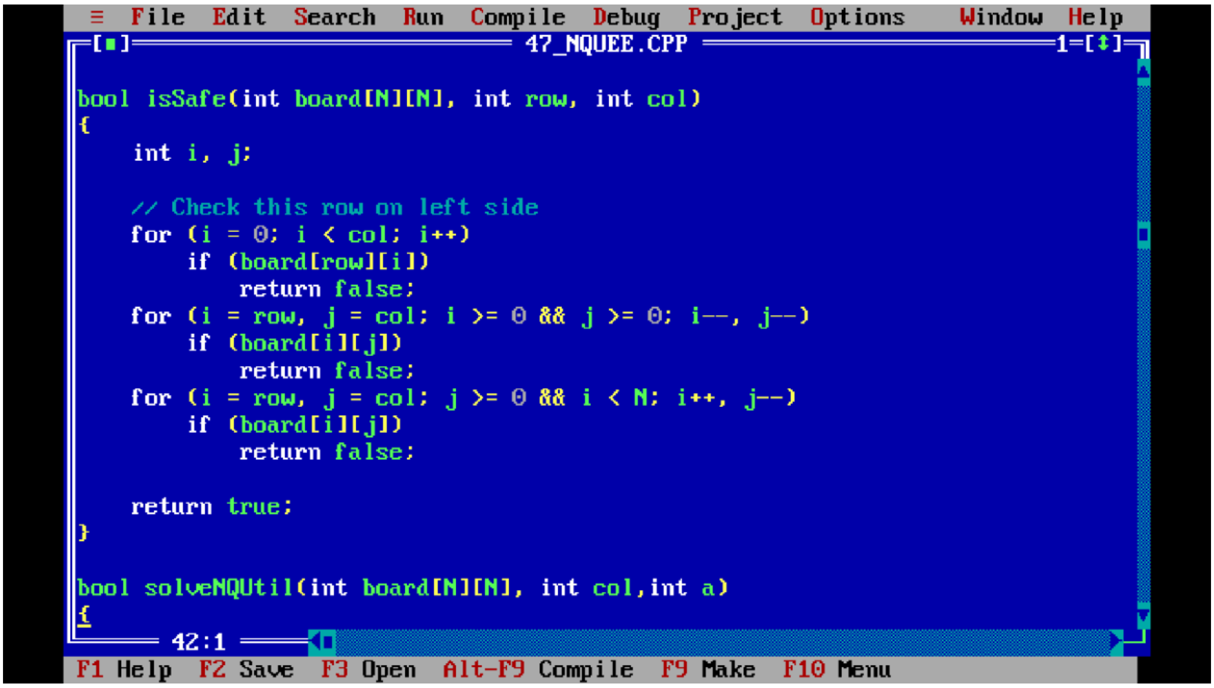
Program:

```
#define N 100
#include <stdio.h>
#include<conio.h>

typedef enum {
    false,
    true
} bool;

void printSolution(int board[N][N],int a)
{
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < a; j++) {
            if(board[i][j])
                printf("Q ");
            else
                printf(". ");
        }
        printf("\n");
    }
}

bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    // Check this row on left side
    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[N][N], int col,int a)
{
```

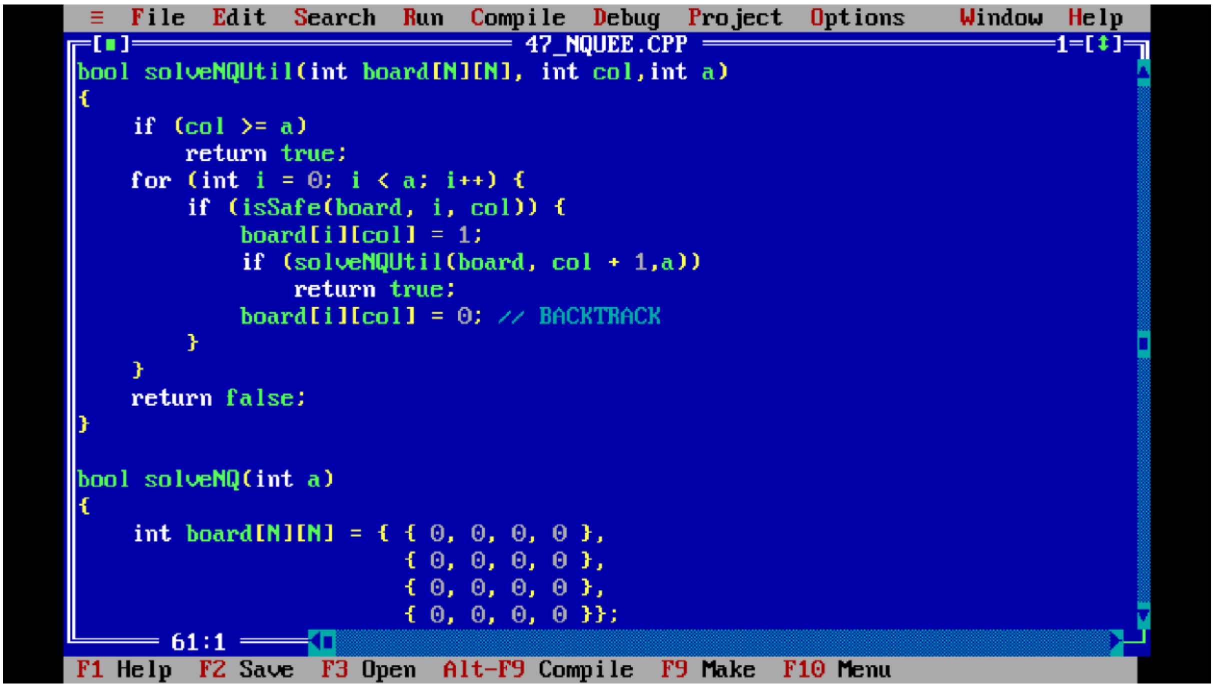
```
    if (col >= a)
        return true;
    for (int i = 0; i < a; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1,a))
                return true;
            board[i][col] = 0; // BACKTRACK
        }
    }
    return false;
}

bool solveNQ(int a)
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 }};

    if (solveNQUtil(board, 0, a) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board, a);
    return true;
}

int main()
{
    int a=0;
    printf("Enter number of queens :");
    scanf("%d",&a);
    solveNQ(a);
    getch();
    return 0;
}
```



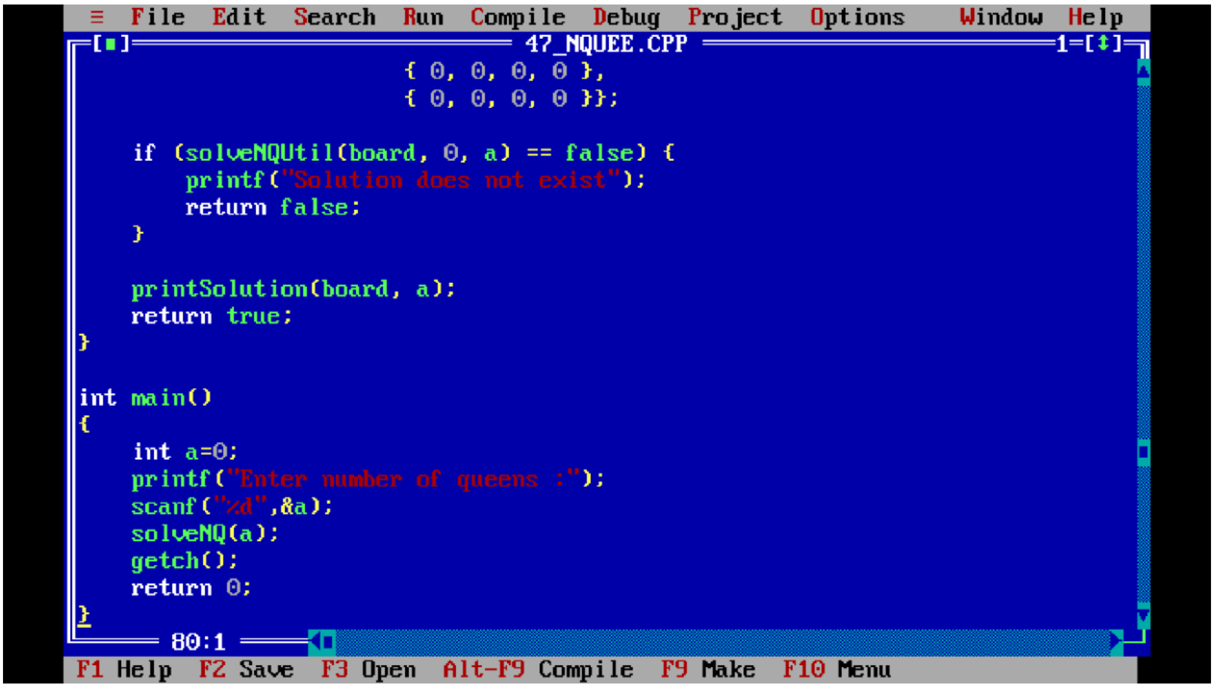

```
#define N 100
#include <stdio.h>
#include<conio.h>

typedef enum {
    false,
    true
} bool;

void printSolution(int board[N][N],int a)
{
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < a; j++) {
            if(board[i][j])
                printf("Q ");
            else
                printf(". ");
        }
        printf("\n");
    }
}
```




```
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    // Check this row on left side
    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[N][N], int col,int a)
{
```



```
≡ File Edit Search Run Compile Debug Project Options Window Help
[■] 47_NQUEE.CPP 1=[↕]
bool solveNQUtil(int board[N][N], int col,int a)
{
    if (col >= a)
        return true;
    for (int i = 0; i < a; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1,a))
                return true;
            board[i][col] = 0; // BACKTRACK
        }
    }
    return false;
}

bool solveNQ(int a)
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 }};
61:1
F1 Help  F2 Save  F3 Open  Alt-F9 Compile  F9 Make  F10 Menu
```

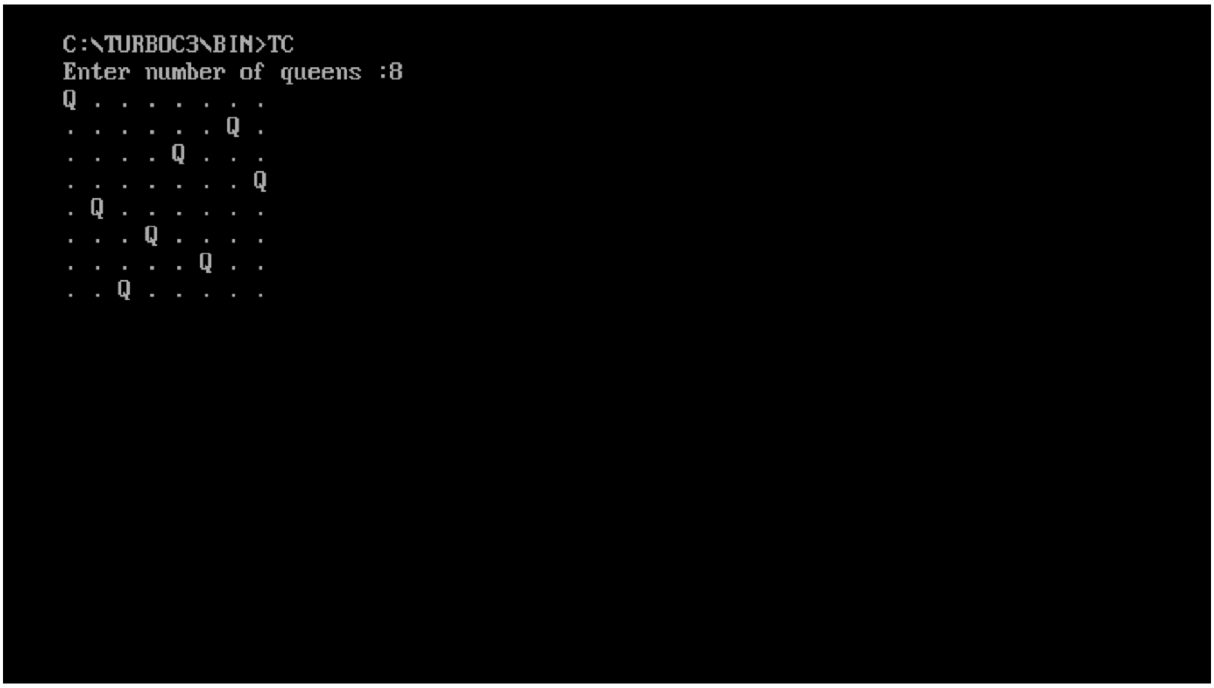


```
≡ File Edit Search Run Compile Debug Project Options Window Help
[■] 47_NQUEE.CPP 1=[↕]
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 }};

    if (solveNQUtil(board, 0, a) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board, a);
    return true;
}

int main()
{
    int a=0;
    printf("Enter number of queens :");
    scanf("%d",&a);
    solveNQ(a);
    getch();
    return 0;
}
80:1
F1 Help  F2 Save  F3 Open  Alt-F9 Compile  F9 Make  F10 Menu
```

Output:

Enter number of queens :8

```
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```



Enter number of queens :4

```
.  .  Q  .
Q  .  .  .
.  .  .  Q
.  Q  .  .
```

Conclusion:

In conclusion, the N Queen problem is a classic example of a constraint satisfaction problem that can be solved using backtracking. The goal is to place N queens on an N x N chessboard without any of them attacking each other. The backtracking algorithm explores different possibilities by placing queens one by one and checking for any conflicts. If a conflict is found, the algorithm backtracks and tries a different placement. This process continues until a valid solution is found or all possibilities have been exhausted. The N Queen problem highlights the power and versatility of backtracking as a problem-solving technique, and it has time complexity of O(N!)