NAME: Meet Raut

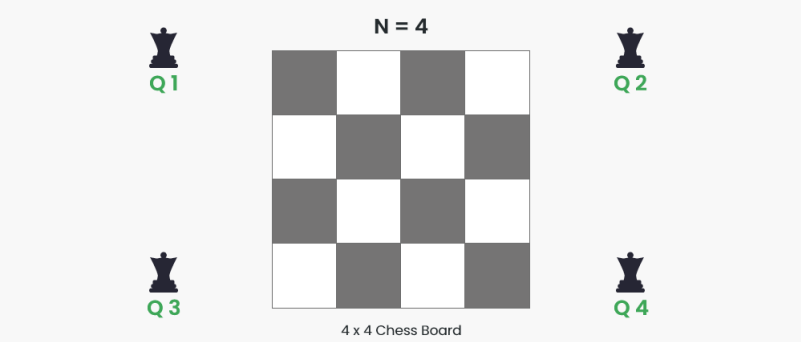
DIV: S2-1

ROLL.NO: 2201084

Experiment 11:

- **AIM:** **To study and implement N Queen Problem using backtracking approach.**

- **THEORY:**



The **N** Queen is the problem of placing **N** chess queens on an **N×N** chessboard so that no two queens attack each other.
For example, the following is a solution for the 4 Queen problem.

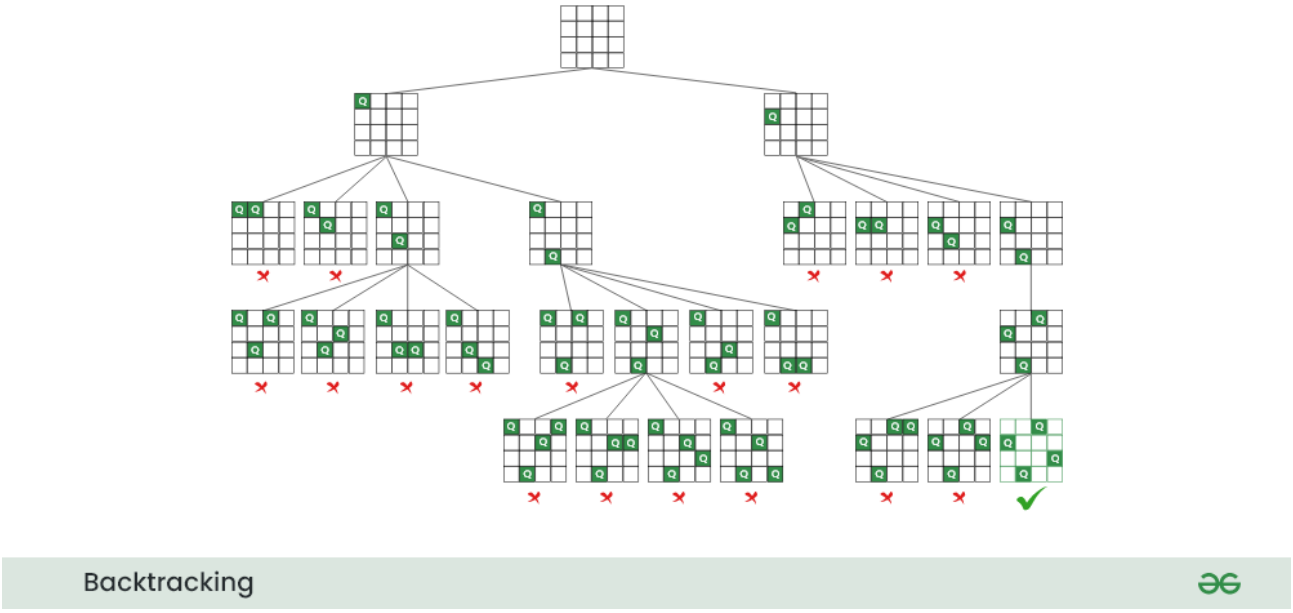
The expected output is in the form of a matrix that has '**Q**'s for the blocks where queens are placed and the empty spaces are represented by **'.'** . For example, the following is the output matrix for the above 4-Queen solution.

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

N Queen Problem using Backtracking:

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**.

Below is the recursive tree of the above approach:



Backtracking

Follow the steps mentioned below to implement the idea:

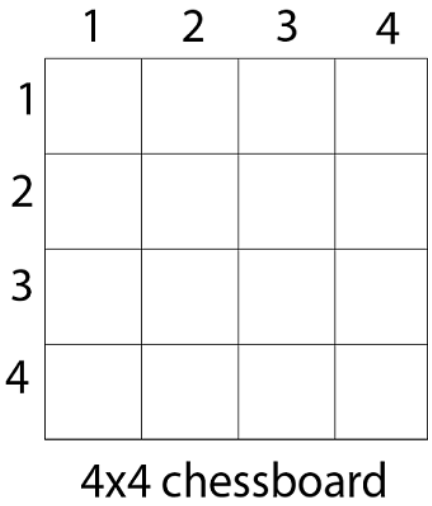
- Start in the leftmost column
- If all queens are placed return true
- Try all rows in the current column. Do the following for every row.
    - If the queen can be placed safely in this row
        - Then mark this **[row, column]** as part of the solution and recursively check if placing queen here leads to a solution.

        - If placing the queen in **[row, column]** leads to a solution then return **true**.
        - If placing queen doesn't lead to a solution then unmark this **[row, column]** then backtrack and try other rows.
    - If all rows have been tried and valid solution is not found return **false** to trigger backtracking.

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

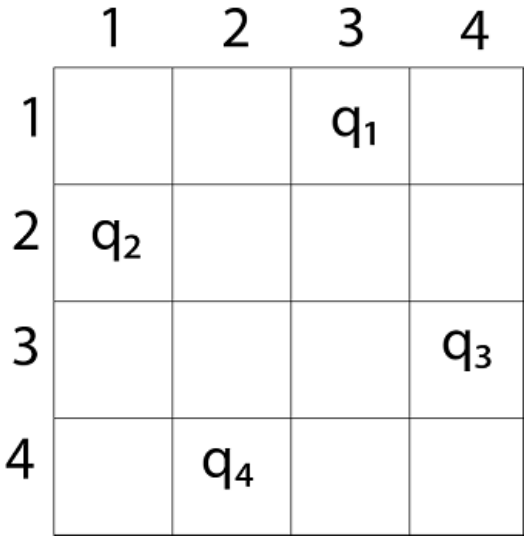
It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.



Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.



The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

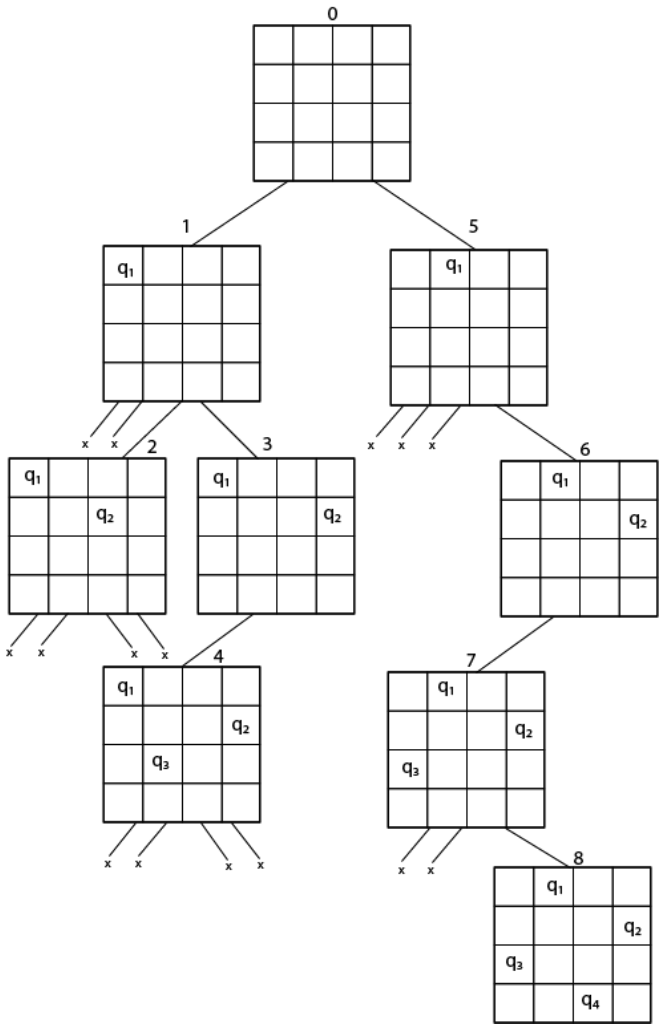
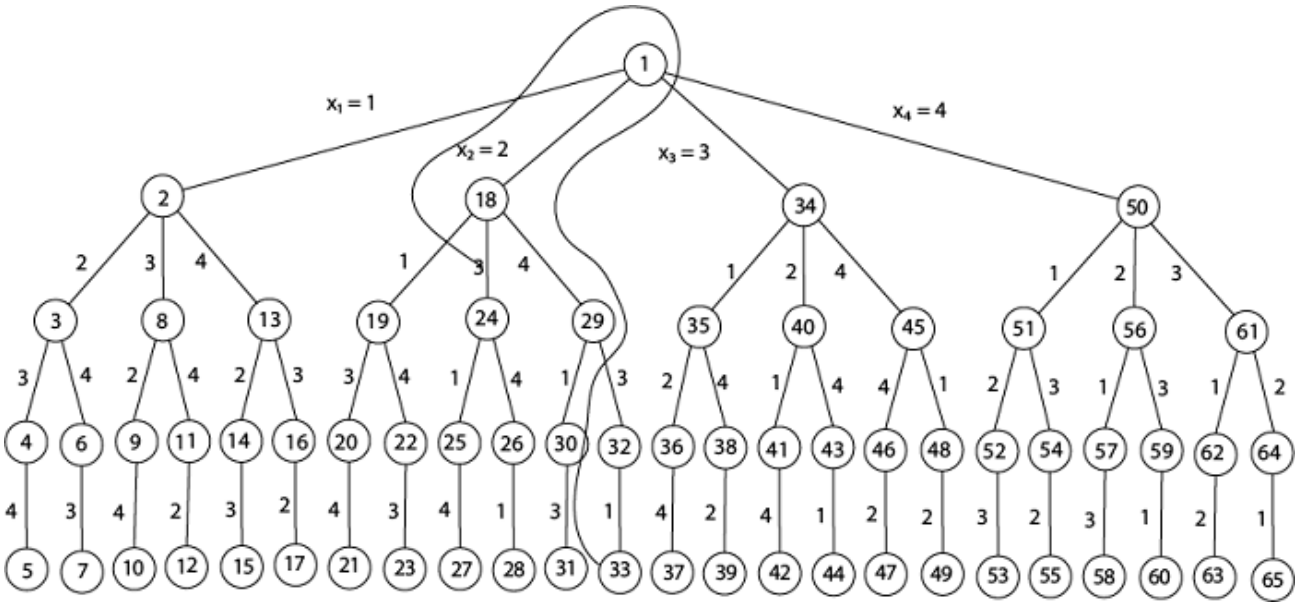


Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in fig:

	1	2	3	4	5	6	7	8
1				q_1				
2						q_2		
3								q_3
4		q_4						
5							q_5	
6	q_6							
7			q_7					
8					q_8			

Thus, the solution for 8 -queen problem for (4, 6, 8, 2, 7, 1, 3, 5).

If two queens are placed at position (i, j) and (k, l).

Then they are on same diagonal only if (i - j) = k - l or i + j = k + l.

The first equation implies that j - l = i - k.

The second equation implies that j - l = k - i.

Therefore, two queens lie on the duplicate diagonal if and only if |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to then n- queens problem.

```
Place (k, i)
  {
    For j ← 1 to k - 1
     do if (x [j] = i)
      or (Abs x [j]) - i) = (Abs (j - k))
   then return false;
    return true;
}
```

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

```
N - Queens (k, n)
{
  For i ← 1 to n
      do if Place (k, i) then
  {
    x [k] ← i;
    if (k ==n) then
      write (x [1....n));
    else
    N - Queens (k + 1, n);
  }
}
```

N Queen Problem Algorithm

1. We create a board of N x N size that stores characters. It will store 'Q' if the queen has been placed at that position else '.'
2. We will create a recursive function called "solve" that takes board and column and all Boards (that stores all the possible arrangements) as arguments. We will pass the column as 0 so that we can start exploring the arrangements from column 1.
3. In solve function we will go row by row for each column and will check if that particular cell is safe or not for the placement of the queen, we will do so with the help of isSafe() function.
4. For each possible cell where the queen is going to be placed, we will first check isSafe() function.
5. If the cell is safe, we put 'Q' in that row and column of the board and again call the solve function by incrementing the column by 1.
6. Whenever we reach a position where the column becomes equal to board length, this implies that all the columns and possible arrangements have been explored, and so we return.
7. Coming on to the boolean isSafe() function, we check if a queen is already present in that row/ column/upper left diagonal/lower left diagonal/upper right diagonal /lower right diagonal. If the queen is present in any of the directions, we return false. Else we put board[row][col] = 'Q' and return true.

Time & Space Complexity

As we got to know in the algorithm for each cell, to check if the queen can be placed there or not, we are iterating for N times. So the recurrence relation comes out to be:

T(N) = N * T(N-1) + N.

T(N-1) = N * T(N-2) + N

T(1) = 1

This totals T(N) = N* N!. therefore, the time complexity comes out to be O(N * N!).

And as we have used an extra board of characters of N x N Size, The space complexity comes out to be O(N *N).

- **C PROGRAM:**

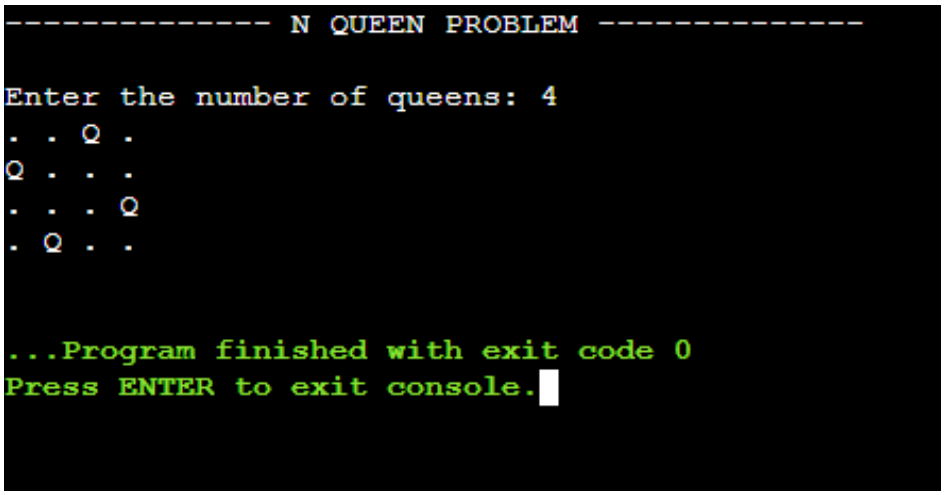
```
#include <stdio.h>
#include <stdlib.h>

int isSafe(int row, int col, char **board, int n) {
   int i, j;

   for (j = 0; j < col; j++) {
      if (board[row][j] == 'Q') {
         return 0;
      }
   }

   for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
      if (board[i][j] == 'Q') {
         return 0;
      }
```

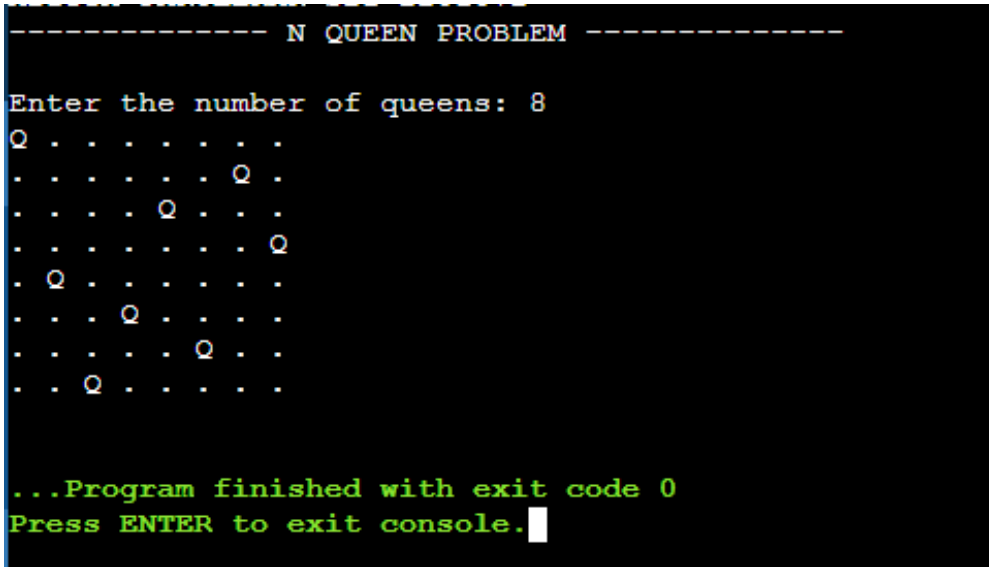
```
    }

    for (i = row, j = col; j >= 0 && i < n; i++, j--) {
        if (board[i][j] == 'Q') {
            return 0;
        }
    }

    return 1;
}

int solveNQueens(char **board, int col, int n) {
    if (col >= n) {
        return 1;
    }

    for (int i = 0; i < n; i++) {
        if (isSafe(i, col, board, n)) {
            board[i][col] = 'Q';

            if (solveNQueens(board, col + 1, n)) {
                return 1;
            }
            board[i][col] = '.';
        }
    }
    return 0;
}
```

```
void printBoard(char **board, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%c ", board[i][j]);
        }
        printf("\n");
    }
}

void solution(int n) {
    char **board = (char **)malloc(n * sizeof(char *));
    for (int i = 0; i < n; i++) {
        board[i] = (char *)malloc(n * sizeof(char));
        for (int j = 0; j < n; j++) {
            board[i][j] = '.';
        }
    }

    if (solveNQueens(board, 0, n) == 0) {
        printf("Solution does not exist");
        return;
    }

    printBoard(board, n);

    for (int i = 0; i < n; i++) {
        free(board[i]);
    }
    free(board);
```

```
}

int main() {
    int n;
    printf("---------- N QUEEN PROBLEM ----------\n\n");
    printf("Enter the number of queens: ");
    scanf("%d", &n);

    solution(n);
    return 0;
}
```

- **OUTPUT:**





- **<u>CONCLUSION:</u> Hence, we have successfully implemented N Queen Problem using backtracking approach; LO 1, LO 2.**