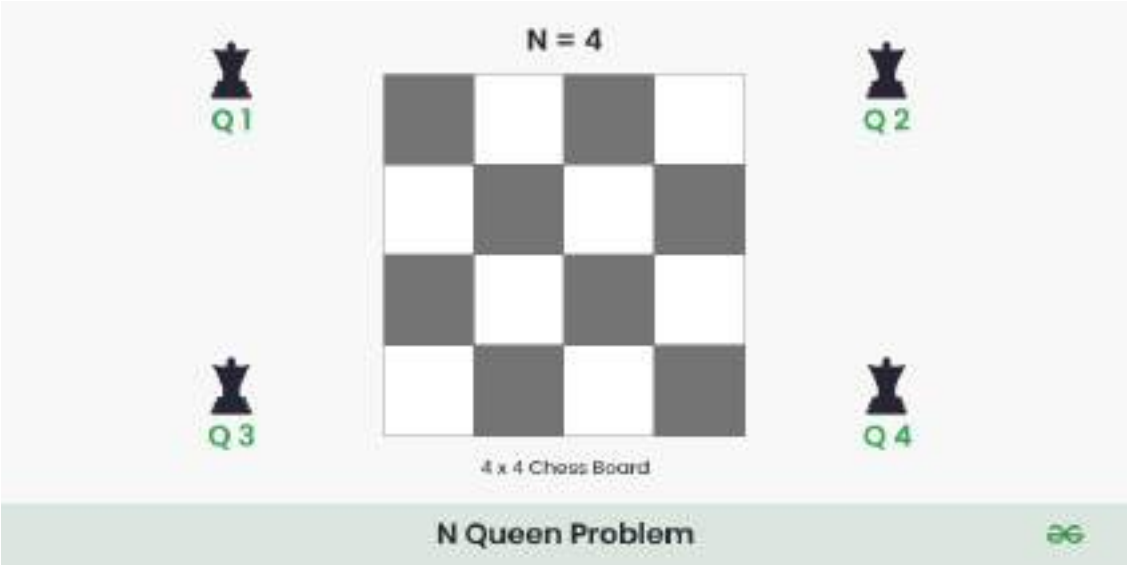
N Queen Problem

We have discussed **Knight's tour** and **Rat in a Maze** problem earlier as examples of Backtracking problems. Let us discuss N Queen as another example problem that can be solved using backtracking.

What is N-Queen problem?



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.

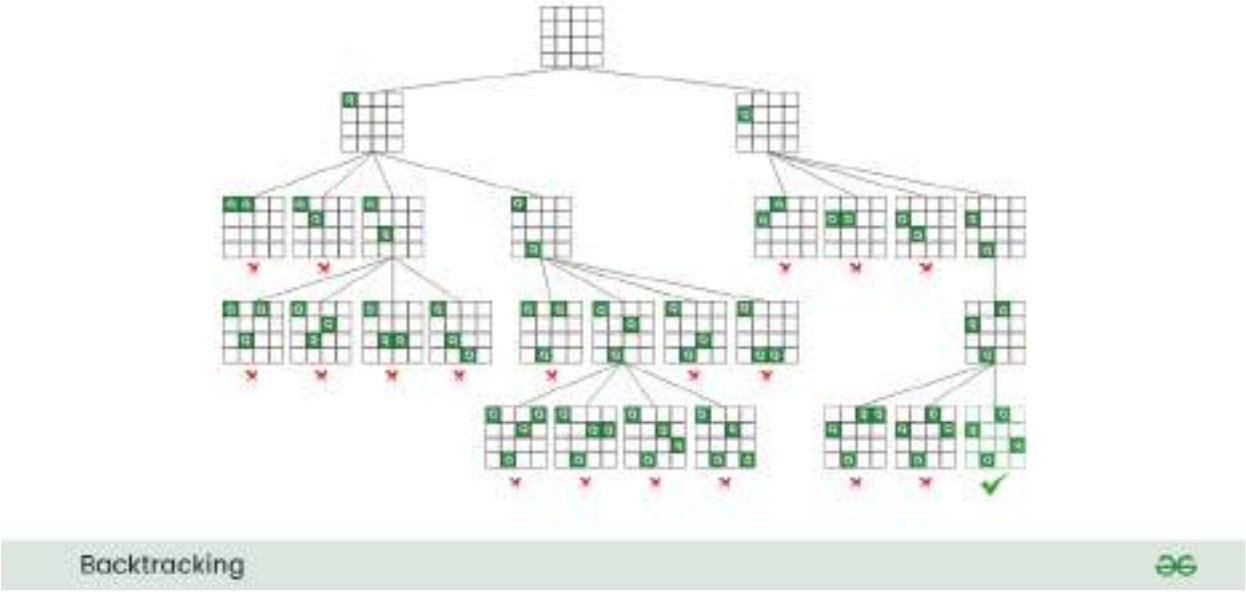
```
. Q . .
. . . Q
Q . . .
. . Q .
```

Recommended: Please solve it on "***PRACTICE***" first, before moving on to the solution.

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:



Recursive tree for N Queen problem

Follow the steps mentioned below to implement the idea:

- Start in the leftmost column

- 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

- 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.

For better visualisation of this backtracking approach, please refer 4 Queen problem.

Note: We can also solve this problem by placing queens in rows as well.

Below is the implementation of the above approach:

C++

```
// C++ program to solve N Queen Problem using backtracking

#include <bits/stdc++.h>
#define N 4
using namespace std;

// A utility function to print solution
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
           if(board[i][j])
            cout << "Q ";
           else cout<<". ";
        printf("\n");
    }
}

// A utility function to check if a queen can
// be placed on board[row][col]. Note that this
// function is called when "col" queens are
// already placed in columns from 0 to col -1.
// So we need to check only left side for
// attacking queens
bool isSafe(int board[N][N], int row, int col)
{
```

```
    // Check this row on left side
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

// A recursive utility function to solve N
// Queen problem
bool solveNQUtil(int board[N][N], int col)
{
    // base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on
        // board[i][col]
        if (isSafe(board, i, col)) {

            // Place this queen in board[i][col]
            board[i][col] = 1;

            // recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col]
            // doesn't lead to a solution, then
            // remove queen from board[i][col]
            board[i][col] = 0; // BACKTRACK
        }
```

```
    // If the queen cannot be placed in any row in
    // this column col  then return false
    return false;
}

// This function solves the N Queen problem using
// Backtracking. It mainly uses solveNQUtil() to
// solve the problem. It returns false if queens
// cannot be placed, otherwise, return true and
// prints placement of queens in the form of 1s.
// Please note that there may be more than one
// solutions, this function prints one  of the
// feasible solutions.
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

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

    printSolution(board);
    return true;
}

// Driver program to test above function
int main()
{
    solveNQ();
    return 0;
}

// This code is contributed by Aditya Kumar (adityakumar129)
```

C

```
// C program to solve N Queen Problem using backtracking

#define N 4
#include <stdbool.h>
#include <stdio.h>
```

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

// A utility function to check if a queen can
// be placed on board[row][col]. Note that this
// function is called when "col" queens are
// already placed in columns from 0 to col -1.
// So we need to check only left side for
// attacking queens
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;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

// A recursive utility function to solve N
// Queen problem
bool solveNQUtil(int board[N][N], int col)
{
    // Base case: If all queens are placed
    // then return true
    if (col >= N)
```

```
        // Consider this column and try placing
        // this queen in all rows one by one
        for (int i = 0; i < N; i++) {

            // Check if the queen can be placed on
            // board[i][col]
            if (isSafe(board, i, col)) {

                // Place this queen in board[i][col]
                board[i][col] = 1;

                // Recur to place rest of the queens
                if (solveNQUtil(board, col + 1))
                    return true;

                // If placing queen in board[i][col]
                // doesn't lead to a solution, then
                // remove queen from board[i][col]
                board[i][col] = 0; // BACKTRACK
            }
        }

        // If the queen cannot be placed in any row in
        // this column col  then return false
        return false;
    }

    // This function solves the N Queen problem using
    // Backtracking. It mainly uses solveNQUtil() to
    // solve the problem. It returns false if queens
    // cannot be placed, otherwise, return true and
    // prints placement of queens in the form of 1s.
    // Please note that there may be more than one
    // solutions, this function prints one  of the
    // feasible solutions.
    bool solveNQ()
    {
        int board[N][N] = { { 0, 0, 0, 0 },
                            { 0, 0, 0, 0 },
                            { 0, 0, 0, 0 },
                            { 0, 0, 0, 0 } };

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

```
}

// Driver program to test above function
int main()
{
    solveNQ();
    return 0;
}

// This code is contributed by Aditya Kumar (adityakumar129)
```

Java

```
// Java program to solve N Queen Problem using backtracking

public class NQueenProblem {
    final int N = 4;

    // A utility function to print solution
    void printSolution(int board[][])
    {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (board[i][j] == 1)
                    System.out.print("Q ");
                else
                    System.out.print(". ");
            }
            System.out.println();
        }
    }

    // A utility function to check if a queen can
    // be placed on board[row][col]. Note that this
    // function is called when "col" queens are already
    // placeed in columns from 0 to col -1. So we need
    // to check only left side for attacking queens
    boolean isSafe(int board[][], int row, int col)
    {
        int i, j;

        // Check this row on left side
        for (i = 0; i < col; i++)
            if (board[row][i] == 1)
                return false;
```

```
            if (board[i][j] == 1)
                return false;

        // Check lower diagonal on left side
        for (i = row, j = col; j >= 0 && i < N; i++, j--)
            if (board[i][j] == 1)
                return false;

        return true;
    }

    // A recursive utility function to solve N
    // Queen problem
    boolean solveNQUtil(int board[][], int col)
    {
        // Base case: If all queens are placed
        // then return true
        if (col >= N)
            return true;

        // Consider this column and try placing
        // this queen in all rows one by one
        for (int i = 0; i < N; i++) {

            // Check if the queen can be placed on
            // board[i][col]
            if (isSafe(board, i, col)) {

                // Place this queen in board[i][col]
                board[i][col] = 1;

                // Recur to place rest of the queens
                if (solveNQUtil(board, col + 1) == true)
                    return true;

                // If placing queen in board[i][col]
                // doesn't lead to a solution then
                // remove queen from board[i][col]
                board[i][col] = 0; // BACKTRACK
            }
        }

        // If the queen can not be placed in any row in
        // this column col, then return false
        return false;
    }
```

```
    // solve the problem. It returns false if queens
    // cannot be placed, otherwise, return true and
    // prints placement of queens in the form of 1s.
    // Please note that there may be more than one
    // solutions, this function prints one of the
    // feasible solutions.
    boolean solveNQ()
    {
        int board[][] = { { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 } };

        if (solveNQUtil(board, 0) == false) {
            System.out.print("Solution does not exist");
            return false;
        }

        printSolution(board);
        return true;
    }

    // Driver program to test above function
    public static void main(String args[])
    {
        NQueenProblem Queen = new NQueenProblem();
        Queen.solveNQ();
    }
}
// This code is contributed by Abhishek Shankhadhar
```

Python3

```
# Python3 program to solve N Queen
# Problem using backtracking

global N
N = 4


def printSolution(board):
    for i in range(N):
        for j in range(N):
            if board[i][j] == 1:
                print("Q",end=" ")
```

```
# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
def isSafe(board, row, col):

    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True


def solveNQUtil(board, col):

    # Base case: If all queens are placed
    # then return true
    if col >= N:
        return True

    # Consider this column and try placing
    # this queen in all rows one by one
    for i in range(N):

        if isSafe(board, i, col):

            # Place this queen in board[i][col]
            board[i][col] = 1

            # Recur to place rest of the queens
```

```
            # If placing queen in board[i][col
            # doesn't lead to a solution, then
            # queen from board[i][col]
            board[i][col] = 0

    # If the queen can not be placed in any row in
    # this column col then return false
    return False


# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
def solveNQ():
    board = [[0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0]]

    if solveNQUtil(board, 0) == False:
        print("Solution does not exist")
        return False

    printSolution(board)
    return True


# Driver Code
if __name__ == '__main__':
    solveNQ()

# This code is contributed by Divyanshu Mehta
```

C#

```
// C# program to solve N Queen Problem
// using backtracking
using System;
```

```
// A utility function to print solution
void printSolution(int [,]board)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if (board[i, j] == 1)
                Console.Write("Q ");
            else
                Console.Write(". ");
        }
        Console.WriteLine();
    }
}

// A utility function to check if a queen can
// be placed on board[row,col]. Note that this
// function is called when "col" queens are already
// placeed in columns from 0 to col -1. So we need
// to check only left side for attacking queens
bool isSafe(int [,]board, int row, int col)
{
    int i, j;

    // Check this row on left side
    for (i = 0; i < col; i++)
        if (board[row,i] == 1)
            return false;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 &&
         j >= 0; i--, j--)
        if (board[i,j] == 1)
            return false;

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 &&
                  i < N; i++, j--)
        if (board[i, j] == 1)
            return false;

    return true;
}

// A recursive utility function to solve N
```

```
{
    // Base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (int i = 0; i < N; i++)
    {
        // Check if the queen can be placed on
        // board[i,col]
        if (isSafe(board, i, col))
        {
            // Place this queen in board[i,col]
            board[i, col] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1) == true)
                return true;

            // If placing queen in board[i,col]
            // doesn't lead to a solution then
            // remove queen from board[i,col]
            board[i, col] = 0; // BACKTRACK
        }
    }

    // If the queen can not be placed in any row in
    // this column col, then return false
    return false;
}

// This function solves the N Queen problem using
// Backtracking. It mainly uses solveNQUtil () to
// solve the problem. It returns false if queens
// cannot be placed, otherwise, return true and
// prints placement of queens in the form of 1s.
// Please note that there may be more than one
// solutions, this function prints one of the
// feasible solutions.
bool solveNQ()
{
    int [,]board = {{ 0, 0, 0, 0 },
                    { 0, 0, 0, 0 },
                    { 0, 0, 0, 0 },
                    { 0, 0, 0, 0 }};
```

```
        {
            Console.Write("Solution does not exist");
            return false;
        }

        printSolution(board);
        return true;
    }

    // Driver Code
    public static void Main(String []args)
    {
        GFG Queen = new GFG();
        Queen.solveNQ();
    }
}

// This code is contributed by Princi Singh
```

Javascript

```
// JavaScript program to solve N Queen
// Problem using backtracking
const N = 4

function printSolution(board)
{
    for(let i = 0; i < N; i++)
    {
        for(let j = 0; j < N; j++)
        {
            if(board[i][j] == 1)
                document.write("Q ")
            else
                document.write(". ")
        }
        document.write("</br>")
    }
}

// A utility function to check if a queen can
// be placed on board[row][col]. Note that this
// function is called when "col" queens are
// already placed in columns from 0 to col -1.
// So we need to check only left side for
```

```
        // Check this row on left side
        for(let i = 0; i < col; i++){
            if(board[row][i] == 1)
                return false
        }

        // Check upper diagonal on left side
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
            if (board[i][j])
                return false

        // Check lower diagonal on left side
        for (i = row, j = col; j >= 0 && i < N; i++, j--)
            if (board[i][j])
                return false

        return true
    }

    function solveNQUtil(board, col){

        // base case: If all queens are placed
        // then return true
        if(col >= N)
            return true

        // Consider this column and try placing
        // this queen in all rows one by one
        for(let i=0;i<N;i++){

            if(isSafe(board, i, col)==true){

                // Place this queen in board[i][col]
                board[i][col] = 1

                // recur to place rest of the queens
                if(solveNQUtil(board, col + 1) == true)
                    return true

                // If placing queen in board[i][col
                // doesn't lead to a solution, then
                // queen from board[i][col]
                board[i][col] = 0
            }
        }
        // if the queen can not be placed in any row in
```

```
}

// This function solves the N Queen problem using
// Backtracking. It mainly uses solveNQUtil() to
// solve the problem. It returns false if queens
// cannot be placed, otherwise return true and
// placement of queens in the form of 1s.
// note that there may be more than one
// solutions, this function prints one of the
// feasible solutions.
function solveNQ(){
    let board = [ [0, 0, 0, 0],
                  [0, 0, 0, 0],
                  [0, 0, 0, 0],
                  [0, 0, 0, 0] ]

    if(solveNQUtil(board, 0) == false){
        document.write("Solution does not exist")
        return false
    }

    printSolution(board)
    return true
}

// Driver Code
solveNQ()

// This code is contributed by shinjanpatra
```


Output

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

Time Complexity: O(N!)
Auxiliary Space: O(N^2)

Further Optimization in is_safe() function:

The idea is not to check every element in the right and left diagonal, instead use the property of diagonals:

- *The sum of **i** and **j** is constant and unique for each right diagonal, where **i** is the row of elements and **j** is the column of elements.*
- *The difference between **i** and **j** is constant and unique for each left diagonal, where **i** and **j** are row and column of element respectively.*

Below is the implementation:

C++

```
// C++ program to solve N Queen Problem using backtracking

#include <bits/stdc++.h>
using namespace std;
#define N 4

// ld is an array where its indices indicate row-col+N-1
// (N-1) is for shifting the difference to store negative
// indices
int ld[30] = { 0 };

// rd is an array where its indices indicate row+col
// and used to check whether a queen can be placed on
// right diagonal or not
```

```
// Column array where its indices indicates column and
// used to check whether a queen can be placed in that
// row or not*/
int cl[30] = { 0 };

// A utility function to print solution
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << " " << (board[i][j]==1?"Q":".") << " ";
        cout << endl;
    }
}

// A recursive utility function to solve N
// Queen problem
bool solveNQUtil(int board[N][N], int col)
{
    // Base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on
        // board[i][col]

        // To check if a queen can be placed on
        // board[row][col].We just need to check
        // ld[row-col+n-1] and rd[row+coln] where
        // ld and rd are for left and right
        // diagonal respectively
        if ((ld[i - col + N - 1] != 1 && rd[i + col] != 1)
            && cl[i] != 1) {

            // Place this queen in board[i][col]
            board[i][col] = 1;
            ld[i - col + N - 1] = rd[i + col] = cl[i] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;
```

```
            // remove queen from board[i][col]
            board[i][col] = 0; // BACKTRACK
            ld[i - col + N - 1] = rd[i + col] = cl[i] = 0;
        }
    }

    // If the queen cannot be placed in any row in
    // this column col  then return false
    return false;
}

// This function solves the N Queen problem using
// Backtracking. It mainly uses solveNQUtil() to
// solve the problem. It returns false if queens
// cannot be placed, otherwise, return true and
// prints placement of queens in the form of 1s.
// Please note that there may be more than one
// solutions, this function prints one  of the
// feasible solutions.
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

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

    printSolution(board);
    return true;
}

// Driver program to test above function
int main()
{
    solveNQ();
    return 0;
}

// This code is contributed by Aditya Kumar (adityakumar129)
```

Java

```
import java.util.*;

class GFG {
    static int N = 4;

    // ld is an array where its indices indicate row-col+N-1
    // (N-1) is for shifting the difference to store
    // negative indices
    static int[] ld = new int[30];

    // rd is an array where its indices indicate row+col
    // and used to check whether a queen can be placed on
    // right diagonal or not
    static int[] rd = new int[30];

    // Column array where its indices indicates column and
    // used to check whether a queen can be placed in that
    // row or not
    static int[] cl = new int[30];

    // A utility function to print solution
    static void printSolution(int board[][])
    {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                System.out.printf(" %d ", board[i][j]);
            System.out.printf("\n");
        }
    }

    // A recursive utility function to solve N
    // Queen problem
    static boolean solveNQUtil(int board[][], int col)
    {
        // Base case: If all queens are placed
        // then return true
        if (col >= N)
            return true;

        // Consider this column and try placing
        // this queen in all rows one by one
        for (int i = 0; i < N; i++) {
            // Check if the queen can be placed on
            // board[i][col]

            // To check if a queen can be placed on
            // board[row][col] We just need to check
```

```
            // diagonal respectively
            if ((ld[i - col + N - 1] != 1
                 && rd[i + col] != 1)
                && cl[i] != 1) {
                // Place this queen in board[i][col]
                board[i][col] = 1;
                ld[i - col + N - 1] = rd[i + col] = cl[i]
                    = 1;

                // Recur to place rest of the queens
                if (solveNQUtil(board, col + 1))
                    return true;

                // If placing queen in board[i][col]
                // doesn't lead to a solution, then
                // remove queen from board[i][col]
                board[i][col] = 0; // BACKTRACK
                ld[i - col + N - 1] = rd[i + col] = cl[i]
                    = 0;
            }
        }

        // If the queen cannot be placed in any row in
        // this column col then return false
        return false;
    }

    // This function solves the N Queen problem using
    // Backtracking. It mainly uses solveNQUtil() to
    // solve the problem. It returns false if queens
    // cannot be placed, otherwise, return true and
    // prints placement of queens in the form of 1s.
    // Please note that there may be more than one
    // solutions, this function prints one of the
    // feasible solutions.
    static boolean solveNQ()
    {
        int board[][] = { { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 } };

        if (solveNQUtil(board, 0) == false) {
            System.out.printf("Solution does not exist");
            return false;
        }
```

```
    }

    // Driver Code
    public static void main(String[] args)
    {
        solveNQ();
    }
}

// This code is contributed by Princi Singh
```

Python3

```
# Python3 program to solve N Queen Problem using
# backtracking
N = 4

# ld is an array where its indices indicate row-col+N-1
# (N-1) is for shifting the difference to store negative
# indices
ld = [0] * 30

# rd is an array where its indices indicate row+col
# and used to check whether a queen can be placed on
# right diagonal or not
rd = [0] * 30

# Column array where its indices indicates column and
# used to check whether a queen can be placed in that
# row or not
cl = [0] * 30


# A utility function to print solution
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=" ")
        print()


# A recursive utility function to solve N
# Queen problem
def solveNQUtil(board, col):
```

```
            return True

        # Consider this column and try placing
        # this queen in all rows one by one
        for i in range(N):

            # Check if the queen can be placed on board[i][col]

            # To check if a queen can be placed on
            # board[row][col] We just need to check
            # ld[row-col+n-1] and rd[row+coln]
            # where ld and rd are for left and
            # right diagonal respectively
            if ((ld[i - col + N - 1] != 1 and
                 rd[i + col] != 1) and cl[i] != 1):

                # Place this queen in board[i][col]
                board[i][col] = 1
                ld[i - col + N - 1] = rd[i + col] = cl[i] = 1

                # Recur to place rest of the queens
                if (solveNQUtil(board, col + 1)):
                    return True

                # If placing queen in board[i][col]
                # doesn't lead to a solution,
                # then remove queen from board[i][col]
                board[i][col] = 0  # BACKTRACK
                ld[i - col + N - 1] = rd[i + col] = cl[i] = 0

                # If the queen cannot be placed in
                # any row in this column col then return False
        return False


# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns False if queens
# cannot be placed, otherwise, return True and
# prints placement of queens in the form of 1s.
# Please note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
def solveNQ():
    board = [[0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0]
```

```
            printf("Solution does not exist")
            return False
        printSolution(board)
        return True


# Driver Code
if __name__ == '__main__':
    solveNQ()

# This code is contributed by SHUBHAMSINGH10
```

C#

```
// C# program to solve N Queen Problem using backtracking

using System;

class GFG {
    static int N = 4;

    // ld is an array where its indices indicate row-col+N-1
    // (N-1) is for shifting the difference to store
    // negative indices
    static int[] ld = new int[30];

    // rd is an array where its indices indicate row+col
    // and used to check whether a queen can be placed on
    // right diagonal or not
    static int[] rd = new int[30];

    // Column array where its indices indicates column and
    // used to check whether a queen can be placed in that
    // row or not
    static int[] cl = new int[30];

    // A utility function to print solution
    static void printSolution(int[, ] board)
    {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                Console.Write(" {0} ", board[i, j]);
            Console.Write("\n");
        }
    }
```

```
static bool solveNQUtil(int[, ] board, int col)
{
    // Base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (int i = 0; i < N; i++) {
        // Check if the queen can be placed on
        // board[i,col]

        // To check if a queen can be placed on
        // board[row,col].We just need to check
        // ld[row-col+n-1] and rd[row+coln] where
        // ld and rd are for left and right
        // diagonal respectively
        if ((ld[i - col + N - 1] != 1
             && rd[i + col] != 1)
            && cl[i] != 1) {
            // Place this queen in board[i,col]
            board[i, col] = 1;
            ld[i - col + N - 1] = rd[i + col] = cl[i]
                = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i,col]
            // doesn't lead to a solution, then
            // remove queen from board[i,col]
            board[i, col] = 0; // BACKTRACK
            ld[i - col + N - 1] = rd[i + col] = cl[i]
                = 0;
        }
    }

    // If the queen cannot be placed in any row in
    // this column col then return false
    return false;
}

// This function solves the N Queen problem using
// Backtracking. It mainly uses solveNQUtil() to
// solve the problem. It returns false if queens
```

```
    // Please note that there may be more than one
    // solutions, this function prints one of the
    // feasible solutions.
    static bool solveNQ()
    {
        int[, ] board = { { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 } };

        if (solveNQUtil(board, 0) == false) {
            Console.Write("Solution does not exist");
            return false;
        }

        printSolution(board);
        return true;
    }

    // Driver Code
    public static void Main(String[] args)
    {
        solveNQ();
    }
}

// This code is contributed by Rajput-Ji
```

Javascript

```
    // JavaScript code to implement the approach

let N = 4;

// ld is an array where its indices indicate row-col+N-1
// (N-1) is for shifting the difference to store negative
// indices
let ld = new Array(30);

// rd is an array where its indices indicate row+col
// and used to check whether a queen can be placed on
// right diagonal or not
let rd = new Array(30);

// Column array where its indices indicates column and
```

```
// A utility function to print solution
function printSolution( board)
{
    for (let i = 0; i < N; i++)
    {
        for (let j = 0; j < N; j++)
            document.write(board[i][j] + " ");
        document.write("<br/>");
    }
}

// A recursive utility function to solve N
// Queen problem
function solveNQUtil(board, col)
{
    // Base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (let i = 0; i < N; i++)
    {

        // Check if the queen can be placed on
        // board[i][col]

        // To check if a queen can be placed on
        // board[row][col].We just need to check
        // ld[row-col+n-1] and rd[row+coln] where
        // ld and rd are for left and right
        // diagonal respectively
        if ((ld[i - col + N - 1] != 1 &&
             rd[i + col] != 1) && cl[i] != 1)
        {
            // Place this queen in board[i][col]
            board[i][col] = 1;
            ld[i - col + N - 1] =
            rd[i + col] = cl[i] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col]
```

```
                board[i][col] = 0; // BACKTRACK
                ld[i - col + N - 1] =
                rd[i + col] = cl[i] = 0;
            }
        }

        // If the queen cannot be placed in any row in
        // this column col then return false
        return false;
    }

    // This function solves the N Queen problem using
    // Backtracking. It mainly uses solveNQUtil() to
    // solve the problem. It returns false if queens
    // cannot be placed, otherwise, return true and
    // prints placement of queens in the form of 1s.
    // Please note that there may be more than one
    // solutions, this function prints one of the
    // feasible solutions.
    function solveNQ()
    {
        let board = [[ 0, 0, 0, 0 ],
                     [ 0, 0, 0, 0 ],
                     [ 0, 0, 0, 0 ],
                     [ 0, 0, 0, 0 ]];

        if (solveNQUtil(board, 0) == false)
        {
            document.write("Solution does not exist");
            return false;
        }

        printSolution(board);
        return true;
    }

    // Driver code
    solveNQ();

    // This code is contributed by sanjoy_62.
```


Output

```
. . Q .
```

. Q . .

Time Complexity: O(N!)
Auxiliary Space: O(N)

Related Articles:

- Printing all solutions in N-Queen Problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

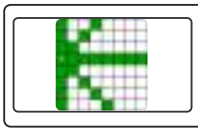
Last Updated : 03 Oct, 2023

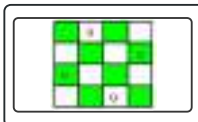
241

Similar Reads


Find position of Queen in chessboard from given attack lines of queen


8 queen problem


N Queen Problem using Branch And Bound

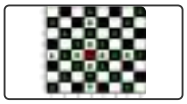

Printing all solutions in N-Queen Problem


N-Queen Problem | Local


N Queen in O(n) space


Check if a Queen can attack a given cell on chessboard


Count positions in a chessboard that can be visited by the Queen which…


Number of cells a queen can move with obstacles on the chessboard


Secretary Problem (A Optimal Stopping Problem)

Related Tutorials


Mathematical and Geometric Algorithms - Data Structure and Algorithm Tutorials


Learn Data Structures with Javascript | DSA Tutorial


Introduction to Max-Heap – Data Structure and Algorithm Tutorials


Introduction to Set – Data Structure and Algorithm Tutorials


Introduction to Map – Data Structure and Algorithm Tutorials

Next

Java Program for N Queen Problem | Backtracking-3

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Hard

Easy | Normal | Medium | Hard | Expert

Improved By : Shivam.Pradhan, Rajput-Ji, AniruddhaPandey, ankita_saini, Parimal7, sanjoy_62, harminder3027, princi singh, SHUBHAMSINGH10,

namananand891, shinjanpatra, meena13091999, surajrasr7277, user_7gr9iodclfx, prajwalkandekar123, laxmishinde5t82, jaykush9161, paras_tiwari_gfg

Article Tags : Accolite , Amazon , Amdocs , chessboard-problems , MAQ Software , Twitter , Visa , Backtracking , DSA

Practice Tags : Accolite, Amazon, Amdocs, MAQ Software, Twitter, Visa, Backtracking

Improve Article Report Issue

Digital Logic Design

Engineering Maths

Python Interview Question

Data Science & ML

Data Science With Python

Data Science For Beginner

Machine Learning Tutorial

Maths For Machine Learning

Pandas Tutorial

NumPy Tutorial

NLP Tutorial

Deep Learning Tutorial

DevOps

Git

AWS

Docker

Kubernetes

Azure

GCP

Competitive Programming

Top DSA for CP

Top 50 Tree Problems

Top 50 Graph Problems

Top 50 Array Problems

Top 50 String Problems

Top 50 DP Problems

Top 15 Websites for CP

System Design

What is System Design

Monolithic and Distributed SD

Scalability in SD

Databases in SD

High Level Design or HLD

Low Level Design or LLD

Crack System Design Round

System Design Interview Questions

Interview Corner

Company Wise Preparation

Preparation for SDE

Experienced Interviews

Internship Interviews

Competitive Programming

Aptitude Preparation

GfG School

CBSE Notes for Class 8

CBSE Notes for Class 9

CBSE Notes for Class 10

CBSE Notes for Class 11

CBSE Notes for Class 12

English Grammar

Commerce

Accountancy

UPSC

Polity Notes

Human Resource Management (HRM)

Management

Income Tax

Finance

Statistics for Economics

Science and Technology Notes

Economics Notes

Important Topics in Ethics

UPSC Previous Year Papers

SSC/ BANKING

SSC CGL Syllabus

SBI PO Syllabus

SBI Clerk Syllabus

IBPS PO Syllabus

IBPS Clerk Syllabus

Aptitude Questions

SSC CGL Practice Papers

Write & Earn

Write an Article

Improve an Article

Pick Topics to Write

Share your Experiences

Internships