Group A

Experiment No: 5

Title of the Assignment: Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen's matrix.

Aim: Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen's matrix.

Objective of the Assignment: Students should be able to understand and solve n-Queen Problem, and understand basics of Backtracking

Prerequisite:

1. Basic of Python or Java Programming

2. Concept of backtracking method

3. N-Queen Problem

Contents for Theory:

1. Introduction to Backtracking

2. N-Queen Problem

Introduction to Backtracking

● Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.

Suppose we have to make a series of decisions among various choices, where

● We don't have enough information to know what to choose

● Each decision leads to a new set of choices.

● Some sequence of choices (more than one choices) may be a solution to your problem.

What is backtracking?

Backtracking is finding the solution of a problem whereby the solution depends on the previous steps taken. For example, in a maze problem, the solution depends on all the steps you take one-by-one. If any of those steps is wrong, then it will not lead us to the solution. In a maze problem, we first choose a path and continue moving along it. But once we understand that the particular path is incorrect, then we just come back and change it. This is what backtracking basically is.

In backtracking, we first take a step and then we see if this step taken is correct or not i.e., whether it will give a correct answer or not. And if it doesn't, then we just come back and change our first step. In general, this is accomplished by recursion. Thus, in backtracking, we first start with a partial sub-solution of the problem (which may or may not lead us to the solution) and then check if we can proceed further with this sub-solution or not. If not, then we just come back and change it.

Thus, the general steps of backtracking are:

- start with a sub-solution
- check if this sub-solution will lead to the solution or not
- If not, then come back and change the sub-solution and continue again

Applications of Backtracking:

- N Queens Problem
- Sum of subsets problem
- Graph coloring
- Hamiltonian cycles.

N queens on NxN chessboard

One of the most common examples of the backtracking is to arrange N queens on an NxN chessboard such that no queen can strike down any other queen. A queen can attack horizontally, vertically, or diagonally. The solution to this problem is also attempted in a similar way. We first place the first queen anywhere arbitrarily and then place the next queen in any of the safe places. We continue this process until the number of unplaced queens becomes zero (a solution is found) or no safe place is left. If no safe place is left, then we change the position of the previously placed queen.

N-Queens Problem:

A classic combinational problem is to place n queens on a n*n chess board so that no two attack, i.,e no two queens are on the same row, column or diagonal.

What is the N Queen Problem?

N Queen problem is the classical Example of backtracking. N-Queen problem is defined as, "given N x N chess board, arrange N queens in such a way that no two queens attack each other by being in the same row, column or diagonal".

● For N = 1, this is a trivial case. For N = 2 and N = 3, a solution is not possible. So we start with N = 4 and we will generalize it for N queens.

If we take n=4then the problem is called the 4 queens problem. If we take n=8 then the problem is called the 8 queens problem.

Algorithm

1) Start in the leftmost column

2) If all queens are place return true

3) Try all rows in the current column.

Do following for every tried row.

a) 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.

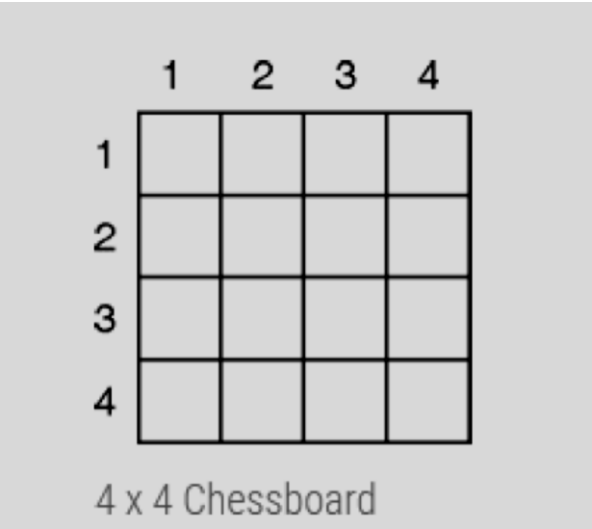
b) If placing the queen in [row, column] leads to a solution then return true.

c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.

4) If all rows have been tried and nothing worked,return false to trigger backtracking.

4- Queen Problem

Problem 1 : Given 4 x 4 chessboard, arrange four queens in a way, such that no two queens attack each other. That is, no two queens are placed in the same row, column, or diagonal.



We have to arrange four queens, Q1, Q2, Q3 and Q4 in 4 x 4 chess board. We will put with queen in ith row. Let us start with position (1, 1). Q1 is the only queen, so there is no issue. partial solution is <1>

● We cannot place Q2 at positions (2, 1) or (2, 2). Position (2, 3) is acceptable. the partial solution is <1, 3>.

● Next, Q3 cannot be placed in position (3, 1) as Q1 attacks her. And it cannot be placed at (3, 2), (3, 3) or (3, 4) as Q2 attacks her. There is no way to put Q3 in the third row. Hence, the algorithm backtracks and goes back to the previous solution and readjusts the position of queen Q2. Q2 is moved from positions (2, 3) to (2, 4). Partial solution is <1, 4>

Now, Q3 can be placed at position (3, 2). Partial solution is <1, 4, 3>.

● Queen Q4 cannot be placed anywhere in row four. So again, backtrack to the previous solution and readjust the position of Q3. Q3 cannot be placed on (3, 3) or(3, 4). So the algorithm backtracks even further.

● All possible choices for Q2 are already explored, hence the algorithm goes back to partial solution <1> and moves the queen Q1 from (1, 1) to (1, 2). And this process continues until a solution is found.

All possible solutions for 4-queen are shown in fig (a) & fig. (b)

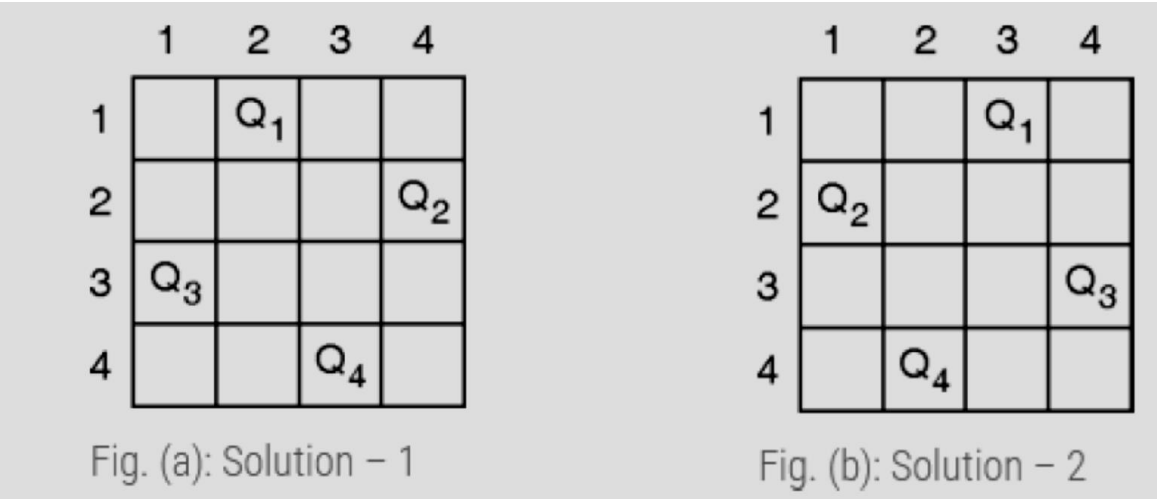
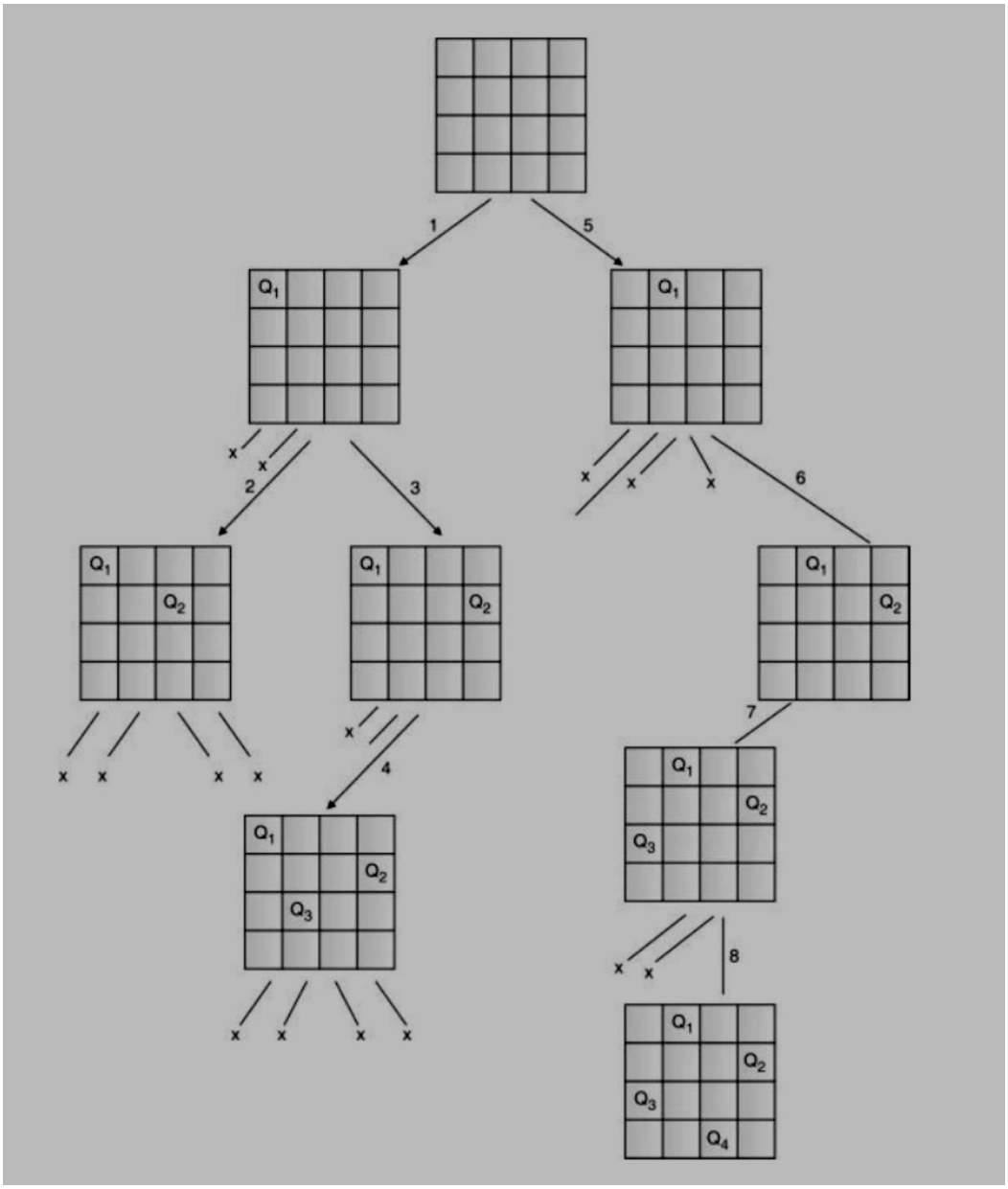


Fig. (a): Solution – 1

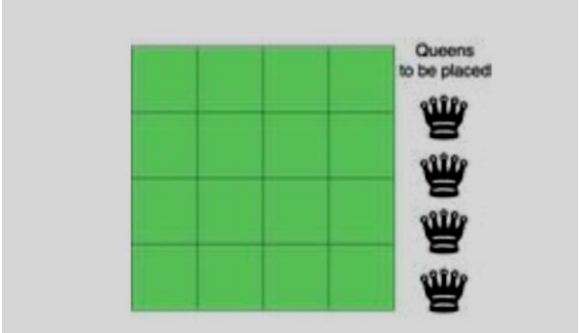
Fig. (b): Solution – 2

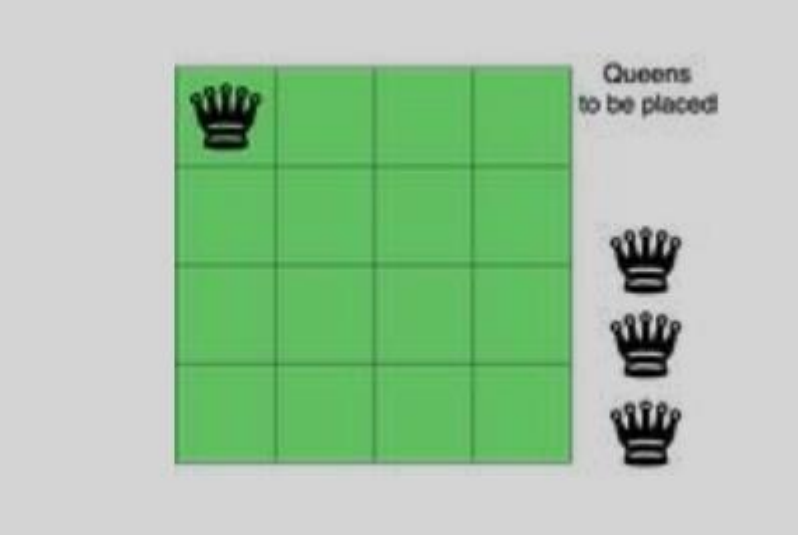
Fig. (d) describes the backtracking sequence for the 4-queen problem



The solution of the 4-queen problem can be seen as four tuples (x1, x2, x3, x4), where xi represents the column number of queen Qi. Two possible solutions for the 4-queen problem are (2, 4, 1, 3) and (3, 1, 4, 2).

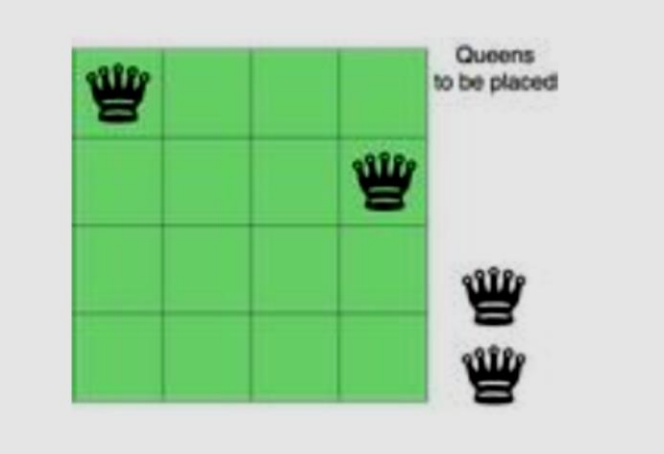
Explanation:



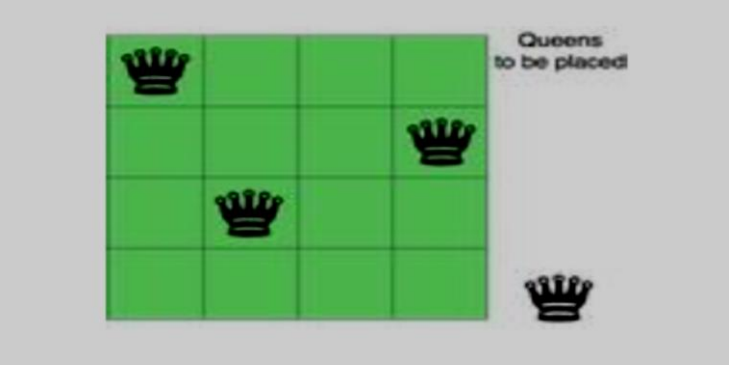


Now, the second step is to place the second queen in a safe position and then the third queen. Now, you can see that there is no safe place where we can put the last queen. So, we will just change the position of the previous queen. And this is backtracking.

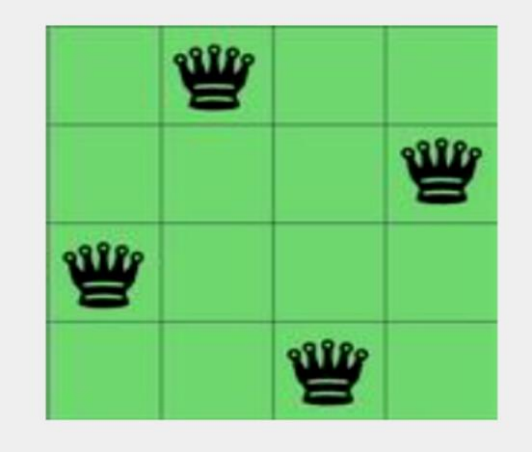
Also, there is no other position where we can place the third queen so we will go back one more step and change the position of the second queen.



And now we will place the third queen again in a safe position until we find a solution.

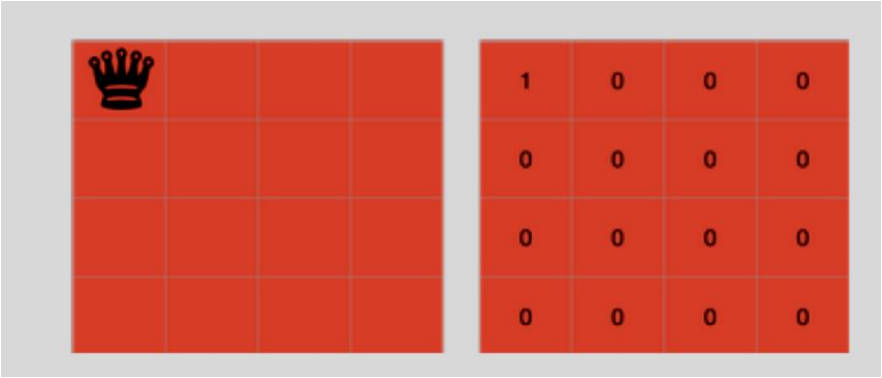


We will continue this process and finally, we will get the solution as shown below.

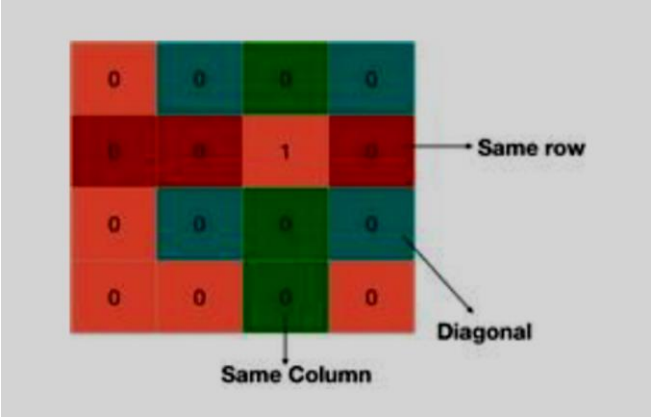


We need to check if a cell (i, j) is under attack or not. For that, we will pass these two in our function along with the chessboard and its size - IS-ATTACK(i, j, board, N).

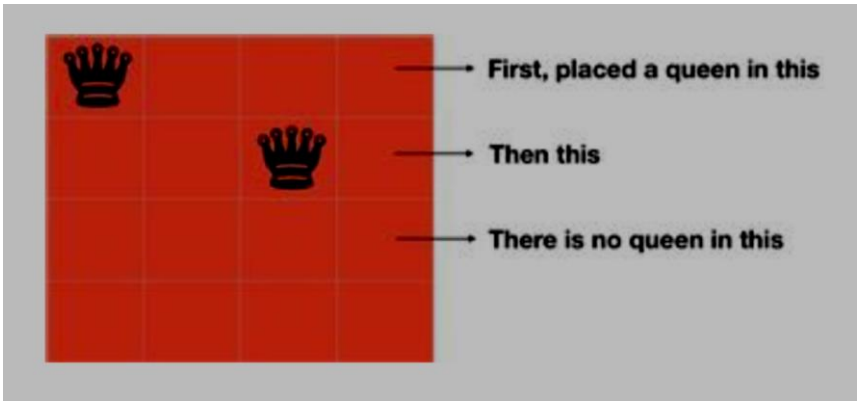
If there is a queen in a cell of the chessboard, then its value will be 1, otherwise, 0.



The cell (i,j) will be under attack in three condition - if there is any other queen in row i, if there is any other queen in the column j or if there is any queen in the diagonals.



We are already proceeding row-wise, so we know that all the rows above the current row(i) are filled but not the current row and thus, there is no need to check for row i.

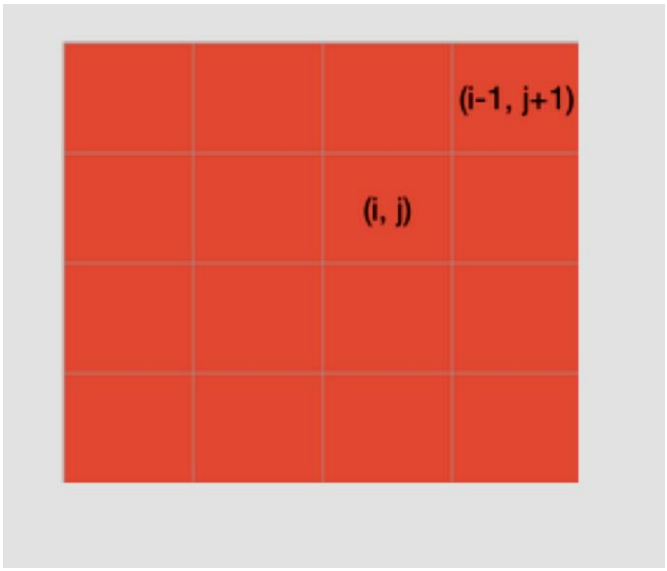


We can check for the column j by changing k from 1 to i-1 in board[k][j] because only the rows from 1 to i-1 are filled.

for k in 1 to i-1

if board[k][j]==1 return TRUE

Now, we need to check for the diagonal. We know that all the rows below the row i are empty, so we need to check only for the diagonal elements which above the row i. If we are on the cell (i, j), then decreasing the value of i and increasing the value of j will make us traverse over the diagonal on the right side, above the row i.

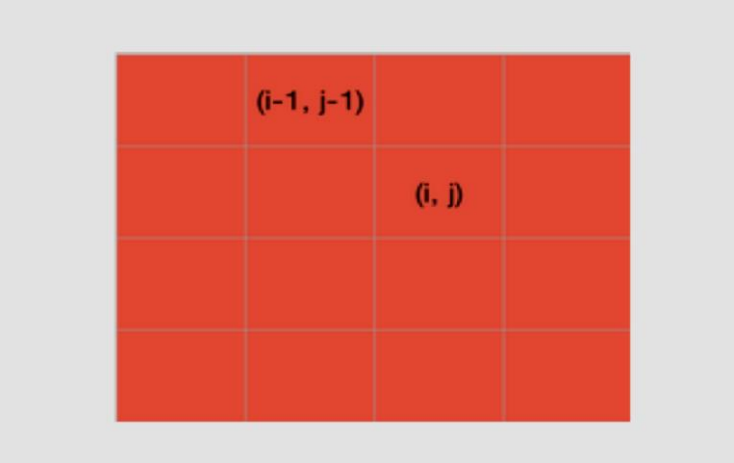


k = i-1 l = j+1

while k>=1 and l<=N if board[k][l] == 1 return TRUE

k=k-1 l=l+1

Also if we reduce both the values of i and j of cell (i, j) by 1, we will traverse over the left diagonal, above the row i.



k = i-1 l = j-1

while k>=1 and l>=1 if board[k][l] == 1 return TRUE

k=k-1 l=l-1

At last, we will return false as it will be return true is not returned by the above statements and the cell (i,j) is safe.

We can write the entire code as:

IS-ATTACK(i, j, board, N)

// checking in the column j

for k in 1 to i-1

if board[k][j]==1

return TRUE

// checking upper right diagonal

k = i-1

l = j+1

while k>=1 and l<=N

if board[k][l] == 1

return TRUE

k=k+1

l=l+1

// checking upper left diagonal

k = i-1

l = j-1

while k>=1 and l>=1

if board[k][l] == 1

return TRUE

k=k-1

l=l-1

return FALSE

Now, let's write the real code involving backtracking to solve the N Queen problem.

Our function will take the row, number of queens, size of the board and the board itself - N-QUEEN(row, n, N, board).

If the number of queens is 0, then we have already placed all the queens. if n==0

return TRUE

Otherwise, we will iterate over each cell of the board in the row passed to the function and for each cell, we will check if we can place the queen in that cell or not. We can't place the queen in a cell if it is under attack.

for j in 1 to N

if !IS-ATTACK(row, j, board, N) board[row][j] = 1

After placing the queen in the cell, we will check if we are able to place the next queen with this arrangement or not. If not, then we will choose a different position for the current queen.

for j in 1 to N

if N-QUEEN(row+1, n-1, N, board) return TRUE

board[row][j] = 0

if N-QUEEN(row+1, n-1, N, board) - We are placing the rest of the queens with the current arrangement. Also, since all the rows up to 'row' are occupied, so we will start from 'row+1'. If this returns true, then we are successful in placing all the queen, if not, then we have to change the position of our current queen. So, we are leaving the current cell board[row][j] = 0 and then iteration will find another place for the queen and this is backtracking.

Take a note that we have already covered the base case - if n==0 → return TRUE. It means when all queens will be placed correctly, then N-QUEEN(row, 0, N, board) will be called and this will return true.

At last, if true is not returned, then we didn't find any way, so we will return false. N-QUEEN(row, n, N, board)

...

return FALSE

N-QUEEN(row, n, N, board)

if n==0

return TRUE

for j in 1 to N

if !IS-ATTACK(row, j, board, N)

board[row][j] = 1

if N-QUEEN(row+1, n-1, N, board)

return TRUE

board[row][j] = 0 //backtracking, changing current decision

return FALSE

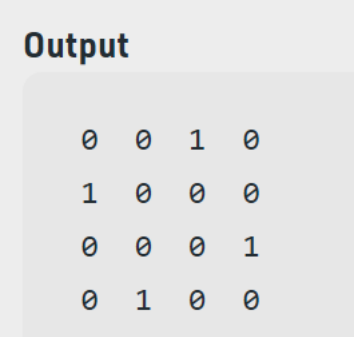
//Code:

```
# 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):
			print(board[i][j], end = " ")
		print()

# 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 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 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
solveNQ()
```

//OUTPUT



Conclusion- In this way we have explored Concept of Backtracking method and solve n-Queen problem using backtracking method.