ECE 220 Computer Systems & Programming

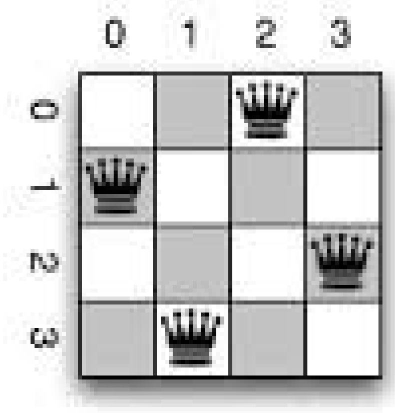
Lecture 15 – Recursion with Backtracking

March 7, 2019

ECE ILLINOIS

ILLINOIS

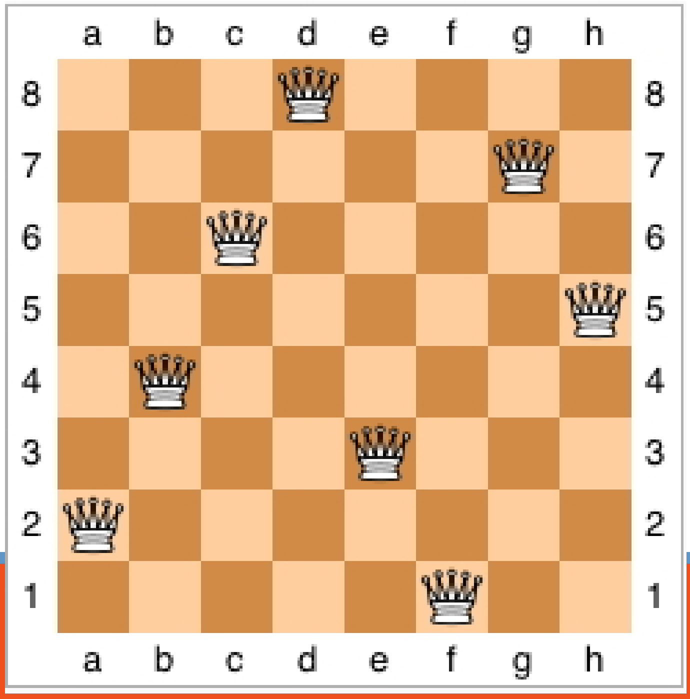
N queens problem using recursive Backtracking



0	0	1	0
1	0	0	0
0	0	0	1
0	1	0	0

- Place N queens on an NxN chessboard so that none of the queens are under attack;

- Brute force: total number of possible placements:
- ~N^2 Choose N ~ 4.4 B (N=8)



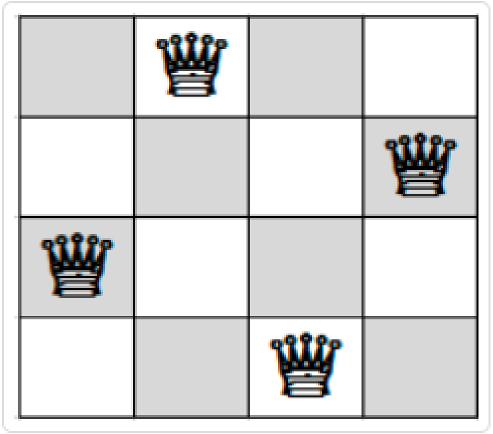
Ref: https://codepumpkin.com/n-queen-problem/

What Is Backtracking Algorithm ?

In backtracking algorithms you try to build a solution one step at a time. If at some step it becomes clear that the current path that you are on cannot lead to a solution, you go back to the previous step (backtrack) and choose a different path. Briefly, once you exhaust all your options at a certain step you go back.

Think of a labyrinth or maze - how do you find a way from an entrance to an exit? Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Backtracking is also known as Depth First Search.

Approach For Solving N Queen Problem Using Recursive Backtracking Algorithm



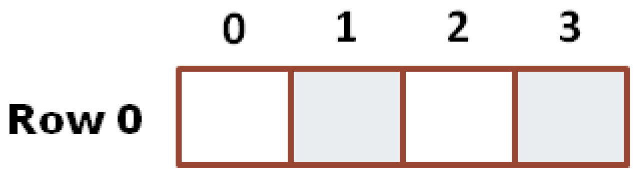
Since our board is 4×4, our recursion will be 4 level deep.

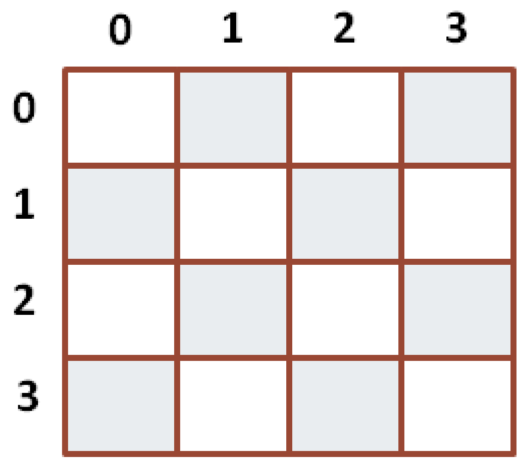
At the 0th level recursion, we place the 0th Queen on the 0th row.

At the 1st level of recursion, we place 1st queen on the 1st row such that she does not attack the 0th queen.

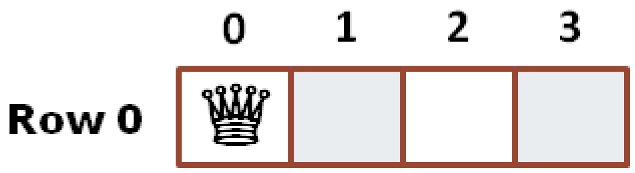
At the 2nd level of recursion, we place 2nd queen on the 2nd row such that she does not attack the 1st and 0th queen and so on.

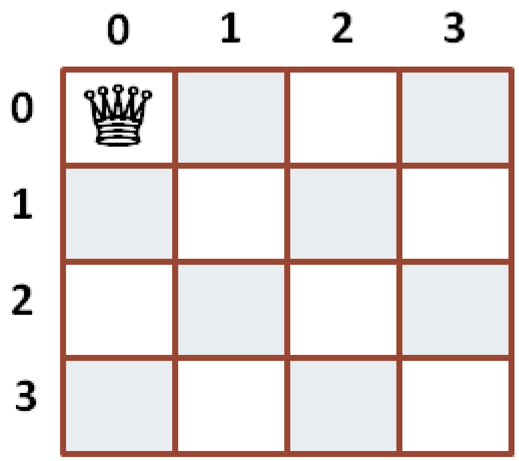
At any point of time if we cannot find a cell for a queen on that row, we return false to the calling funcion and that level of recursion will then try to place the queen on the next available cell of that row. If that doesn' t work out then that function itself is going to return false to the calling function above it and so on.





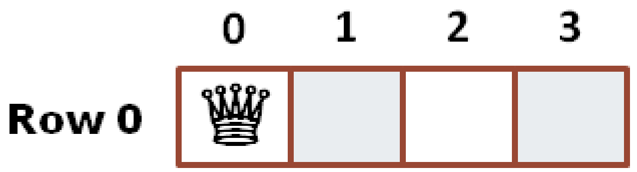
QueenPosition[ ]

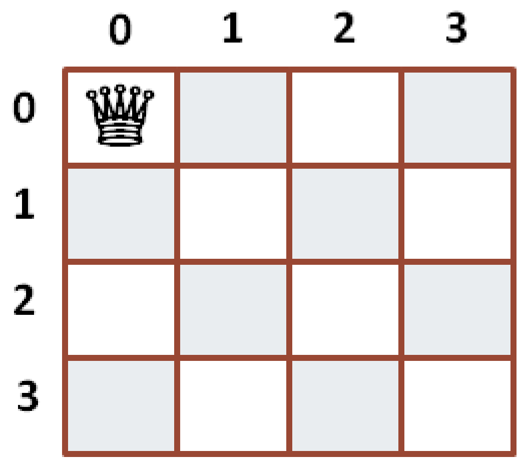




QueenPosition[ ]

Place **0**th Queen on the **0**th Column of **0**th Row

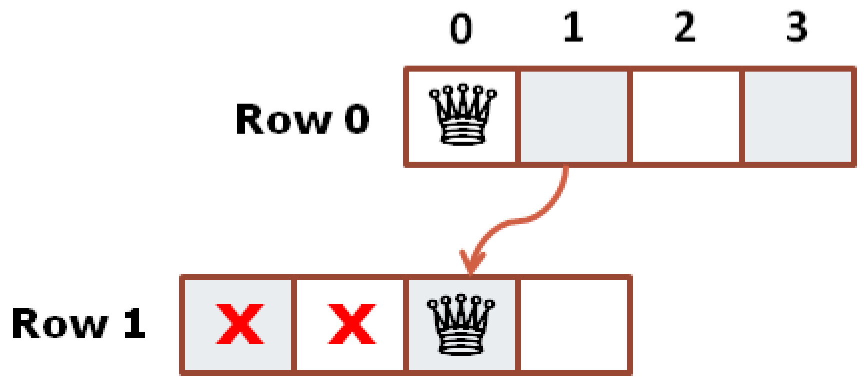


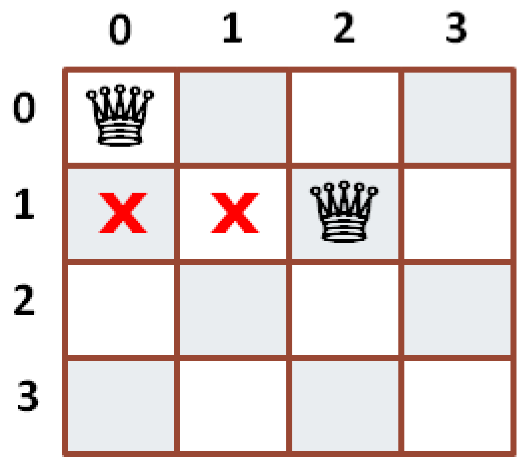


QueenPosition[ ]

Row 0	(0,0)

Add **0**th Queen's position to position array



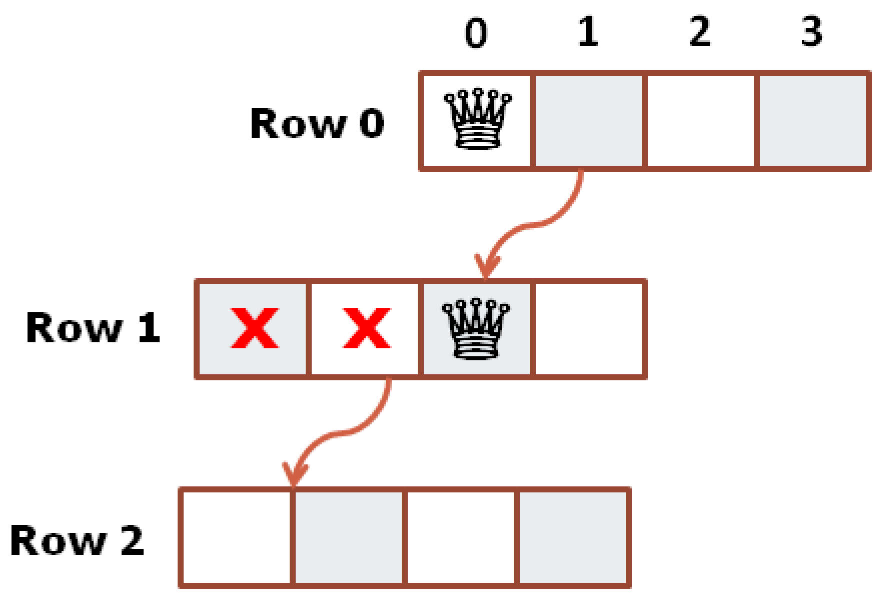


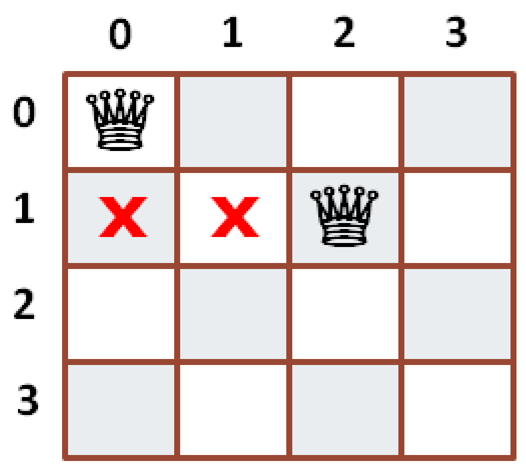
QueenPosition[ ]

Row 0	(0,0)
Row 1	(1,2)

Go to the next level of recursion.

Place the **1**st queen on the **1**st row such that she does not attack the **0**th queen and add that to Positions.

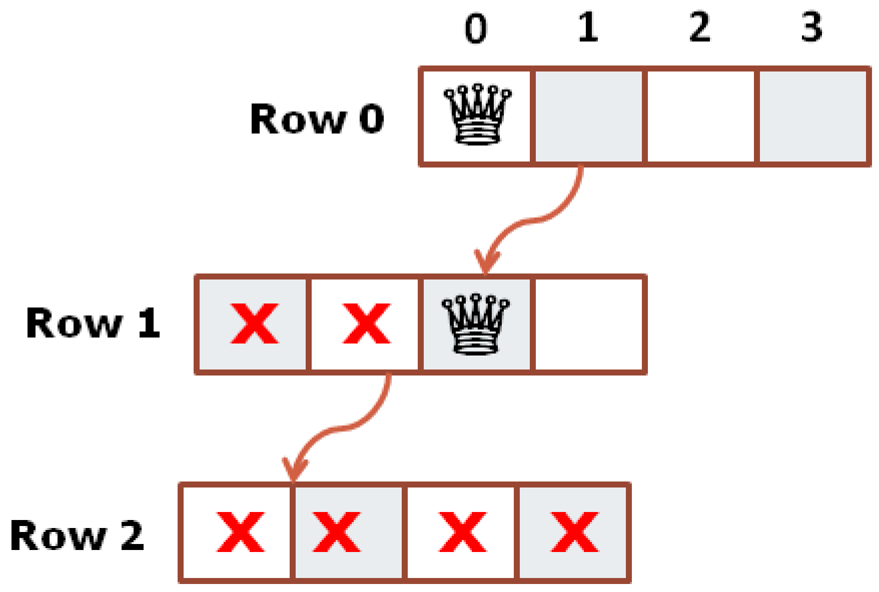


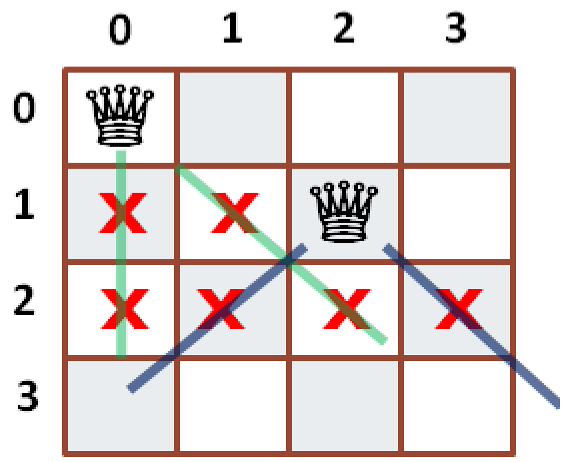


QueenPosition[ ]

Row 0	(0,0)
Row 1	(1,2)

In the next level of recursion, find the cell on **2**nd row such that it is not under attack from any of the available queens.

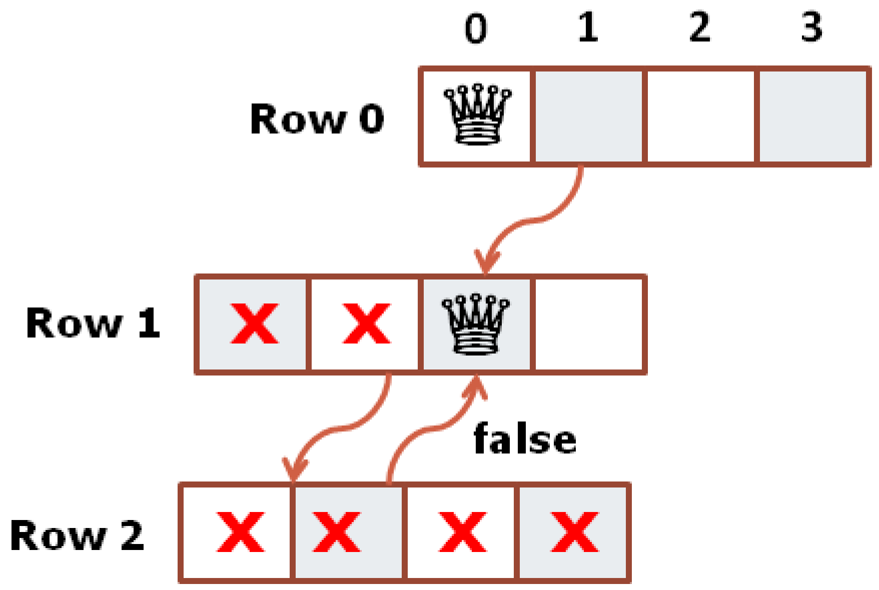


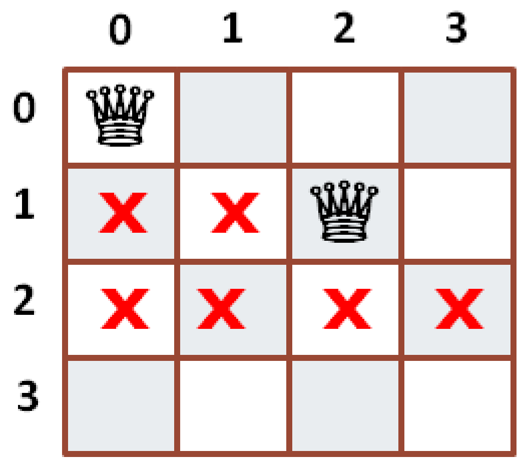


QueenPosition[ ]

Row 0	(0,0)
Row 1	(1,2)

But cell **(2,0)** and **(2,2)** are under attack from **0**th queen and cell **(2,1)** and **(2,3)** are under attack from **1**st queen.

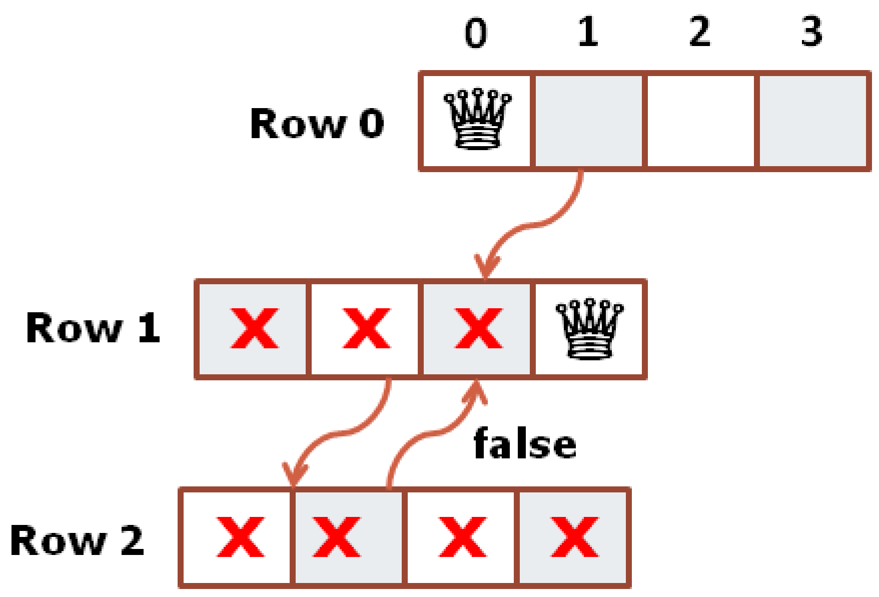


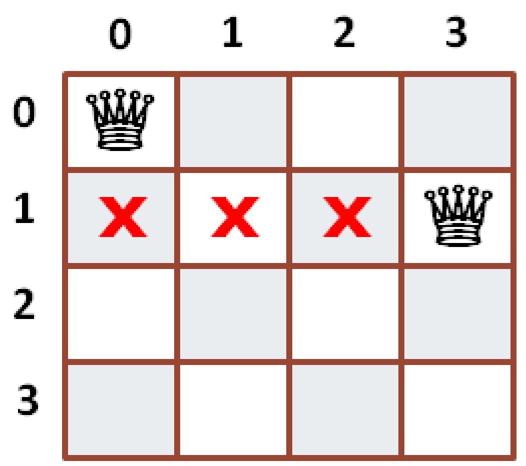


QueenPosition[ ]

Row 0	(0,0)
Row 1	(1,2)

So function will return false to the calling function.

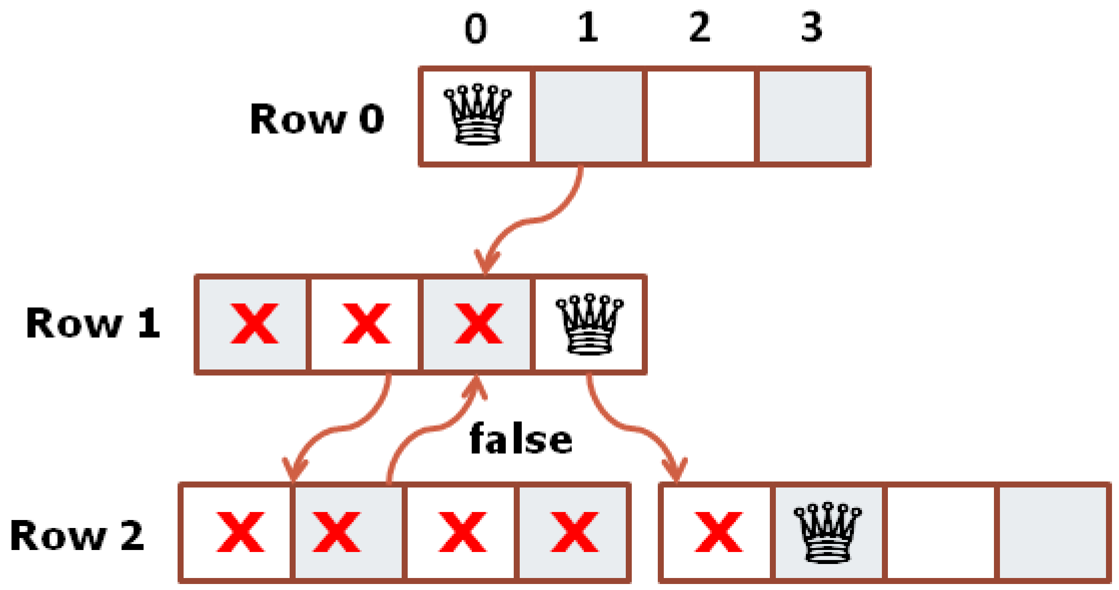


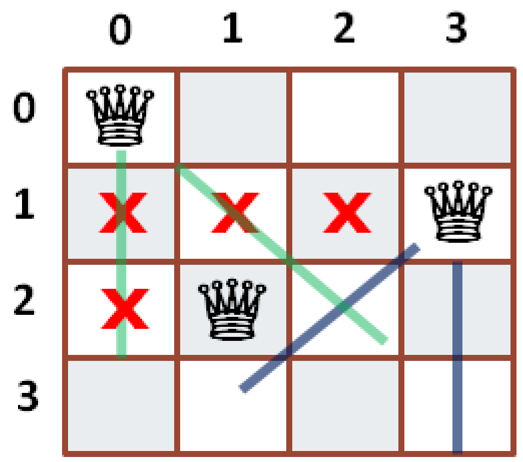


QueenPosition[ ]

Row 0	(0,0)
~~**Row 1**~~	~~(1,2)~~
Row 1	(1,3)

Calling function will try to find next possible place for the **1**st queen on **1**st row and update the queen position in position array.



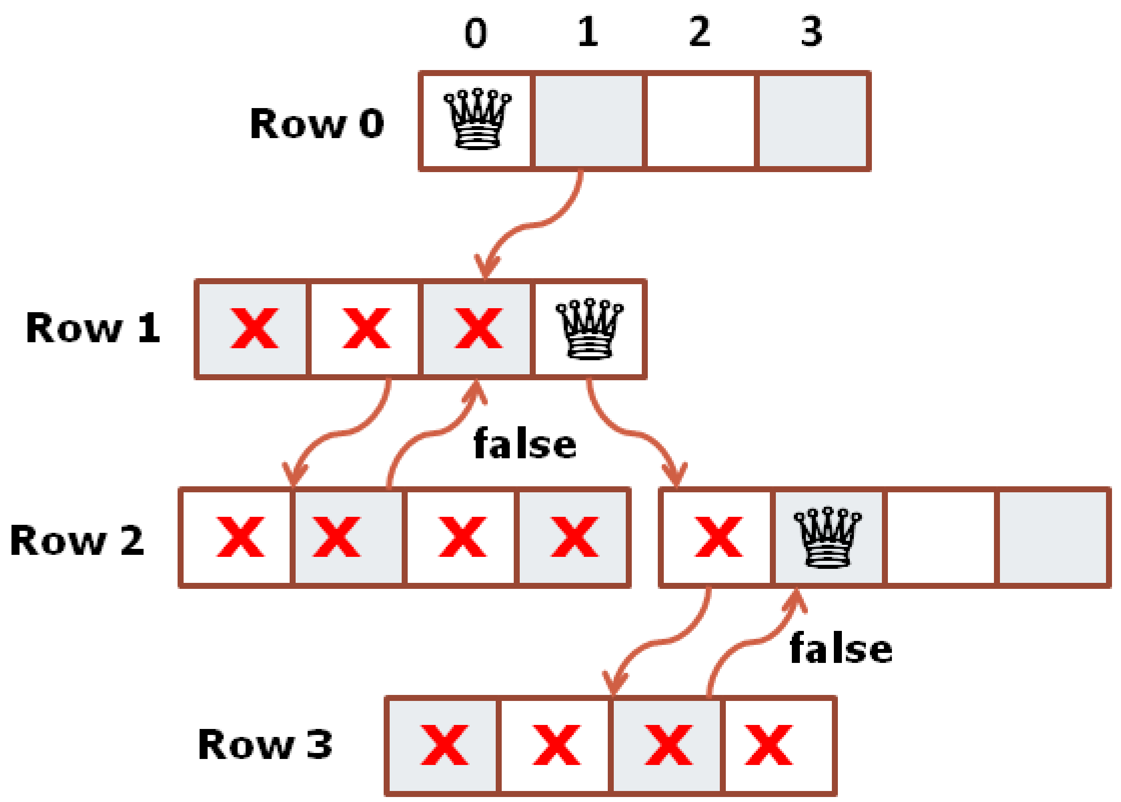


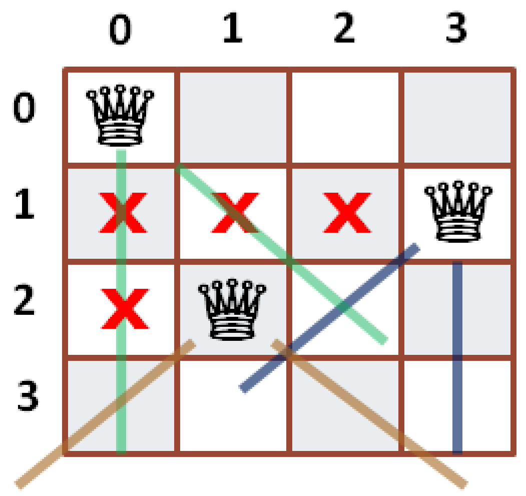
QueenPosition[ ]

Row 0	(0,0)
Row 1	(1,3)
Row 1	(2,1)

Again find the cell on **2**nd row such that it is not under attack from any of the available queens.

Placing the queen in cell **(2,1)** as it is not under attack from any of the queen.

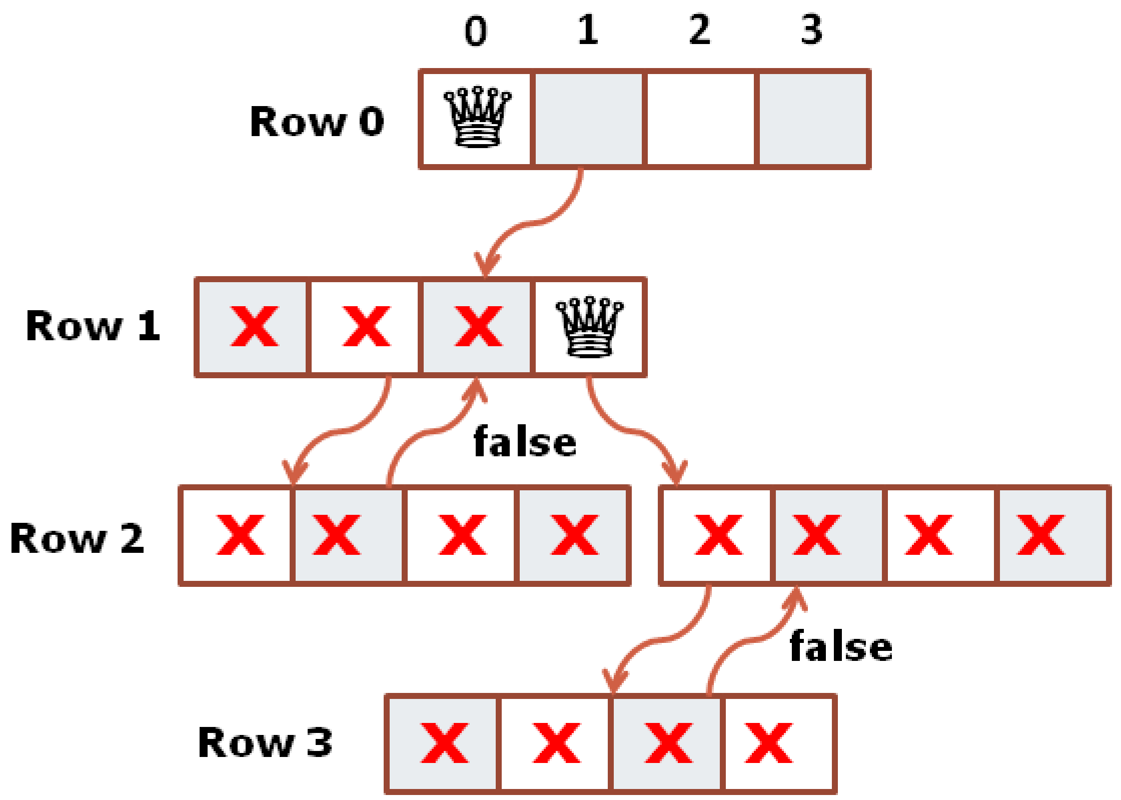


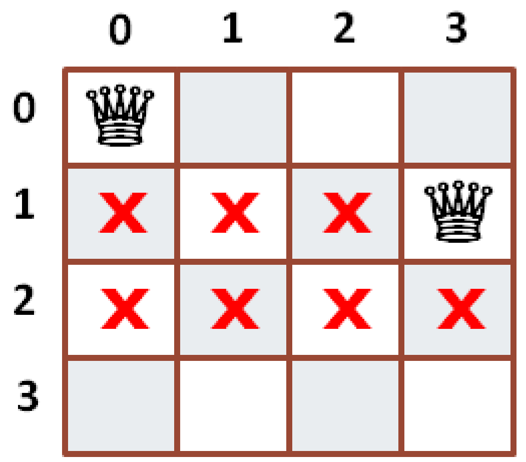


QueenPosition[ ]

Row 0	(0,0)
Row 1	(1,3)
Row 1	(2,1)

For **3**rd queen, no safe cell is available on **3**rd row.
So function will return false to calling function.

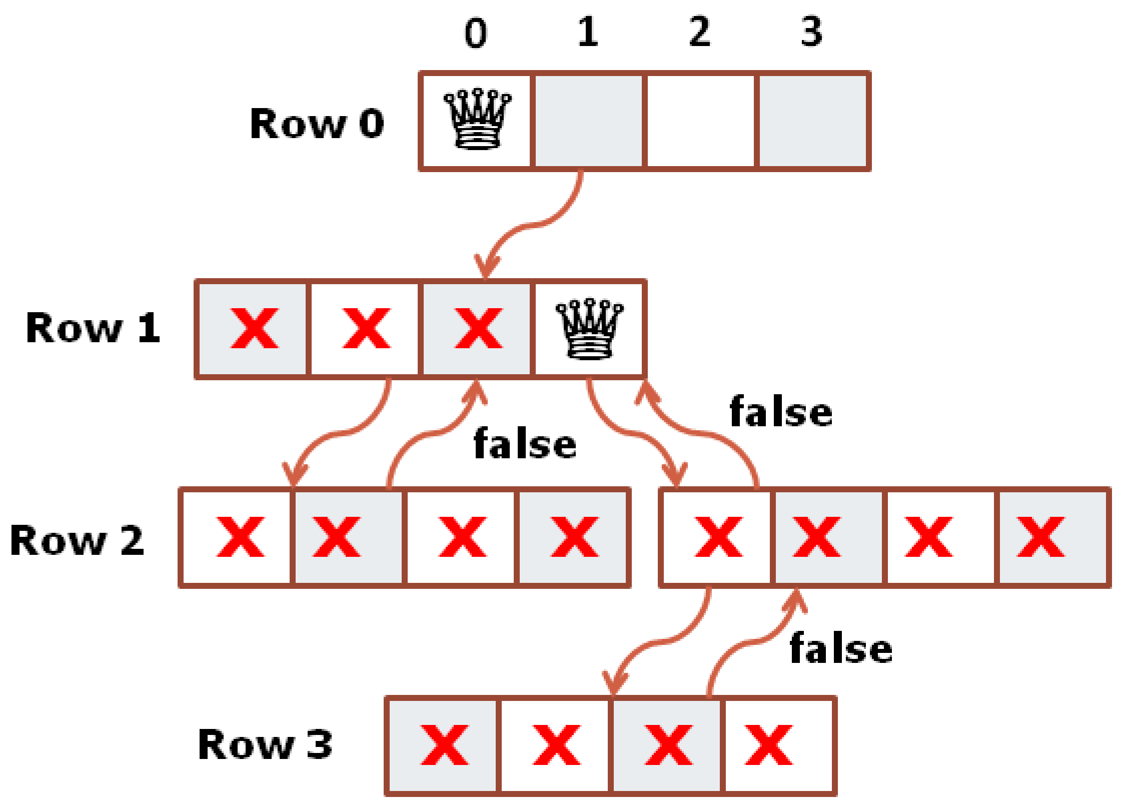


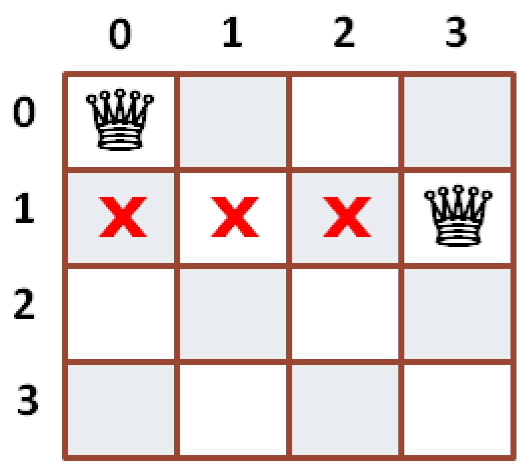


QueenPosition[ ]

Row 0	(0,0)
Row 1	(1,3)
~~**Row 1**~~	~~(2,1)~~

Queen at the **2**nd row tries to find next safe cell.

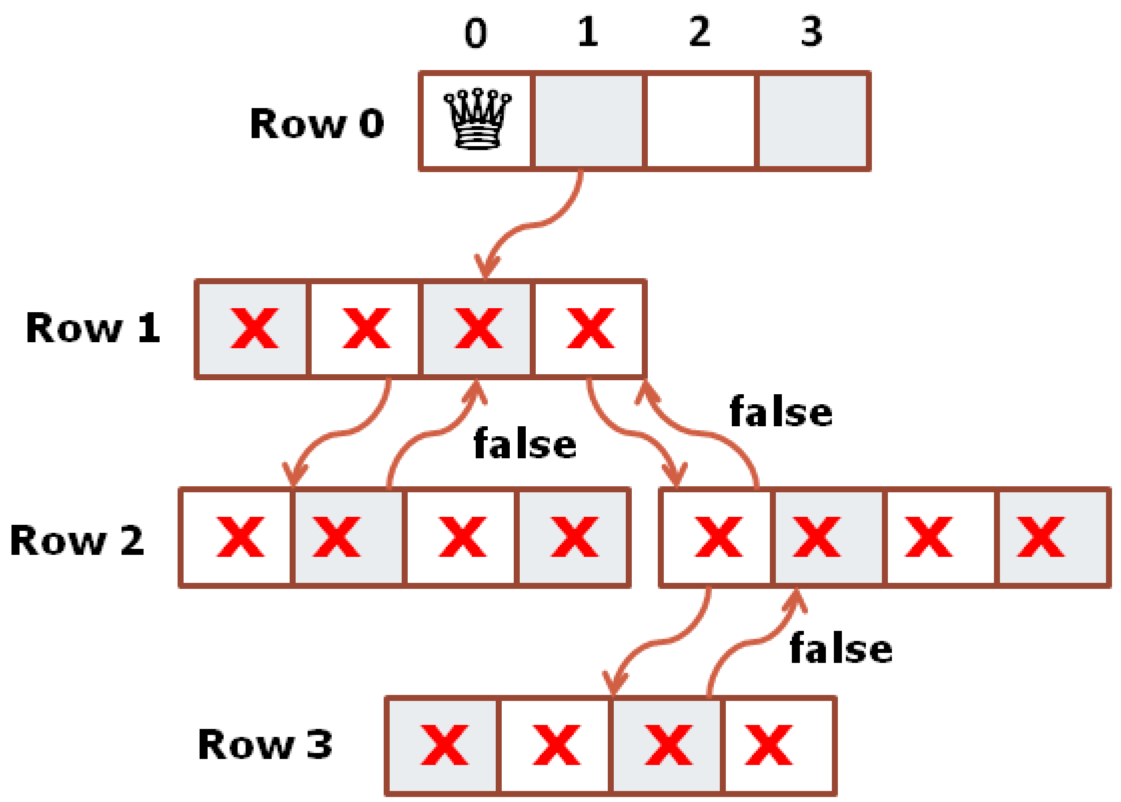


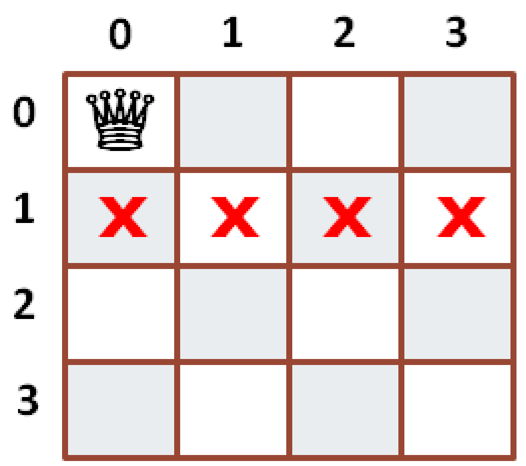


QueenPosition[ ]

Row 0	(0,0)
Row 1	(1,3)

But as both remaining cells are under attack from other queens, this function also returns false to its calling function.

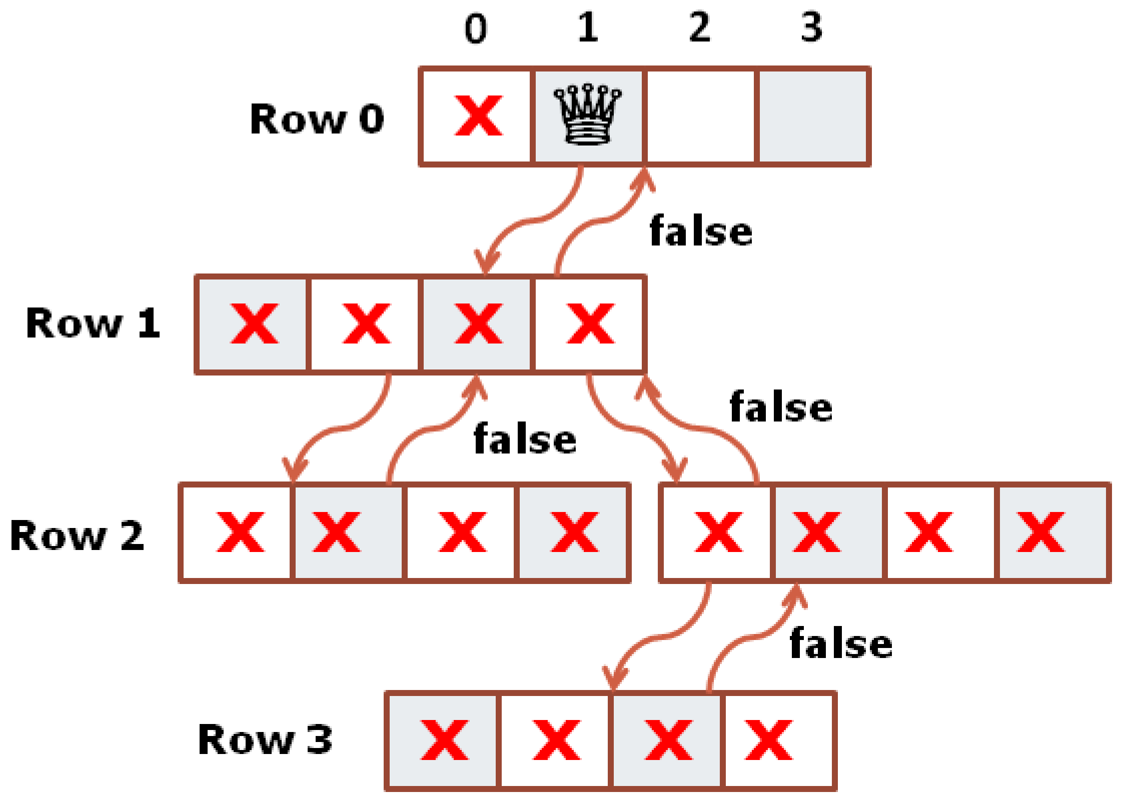


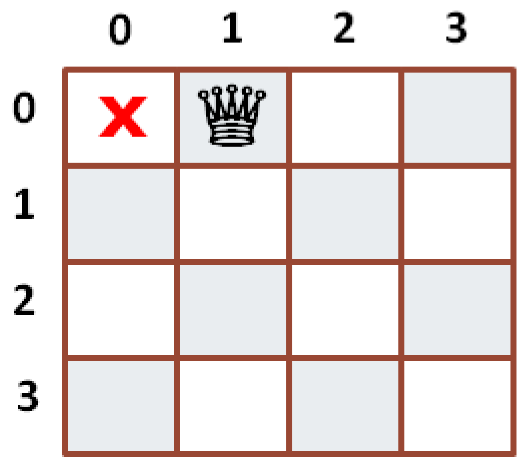


QueenPosition[ ]

Row 0	(0,0)
~~Row 1~~	~~(1,3)~~

Queen at the **1**st row tries to find next safe cell.
But as queen is in the last cell, it will retuen false to
Its calling function.

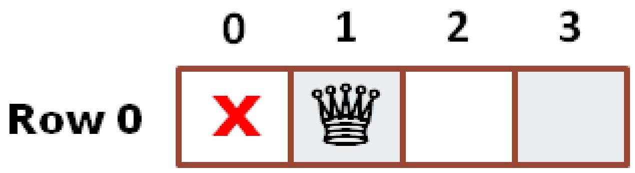


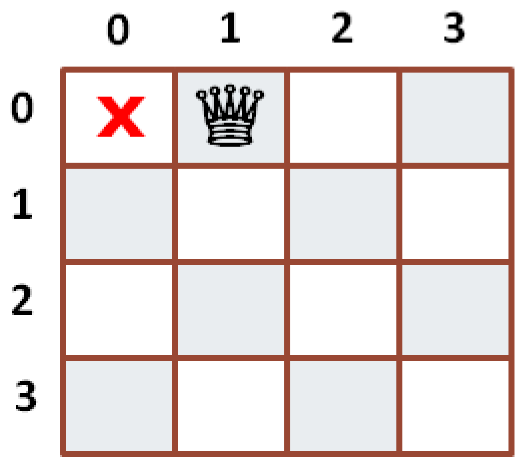


QueenPosition[ ]

~~Row 0~~	~~(0,0)~~
Row 0	(0,1)

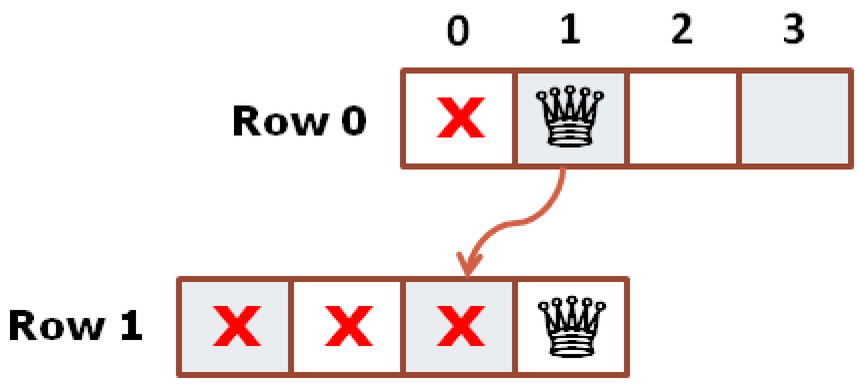
Queen at the **1**st row tries to find next safe cell.
Let us remove these failed recursion calls from the screen.

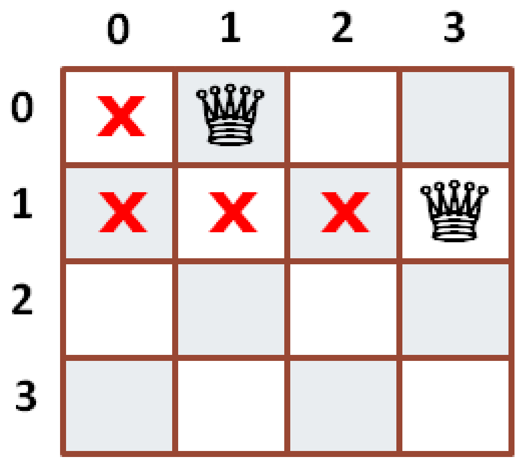




QueenPosition[ ]

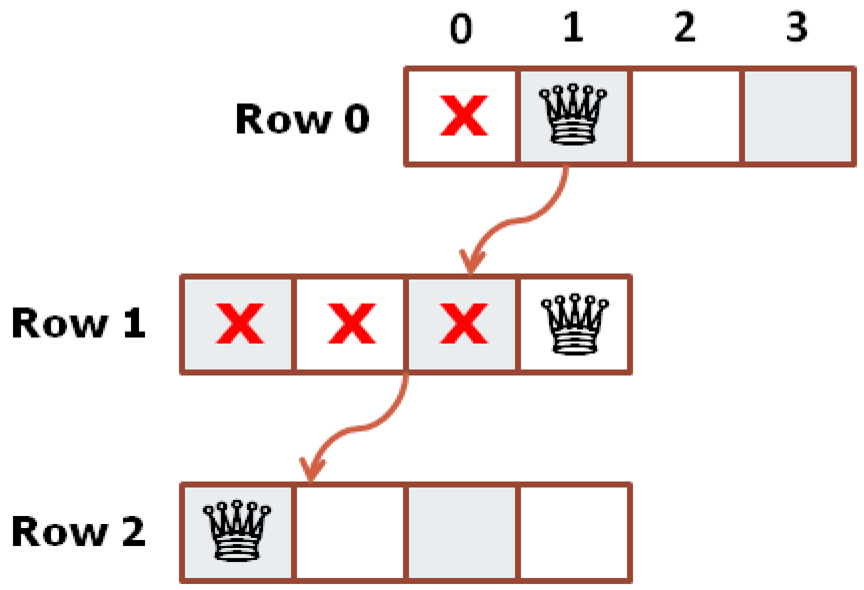
Row 0	(0,1)

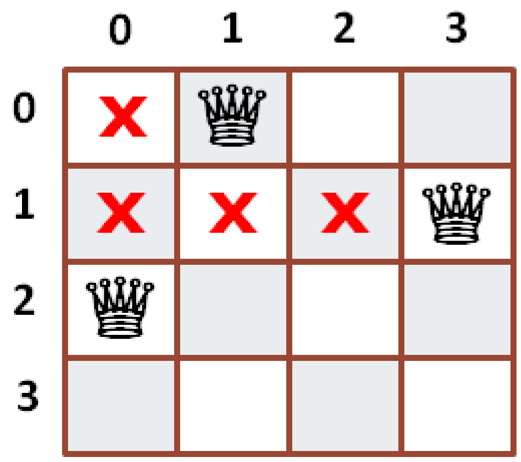




QueenPosition[ ]

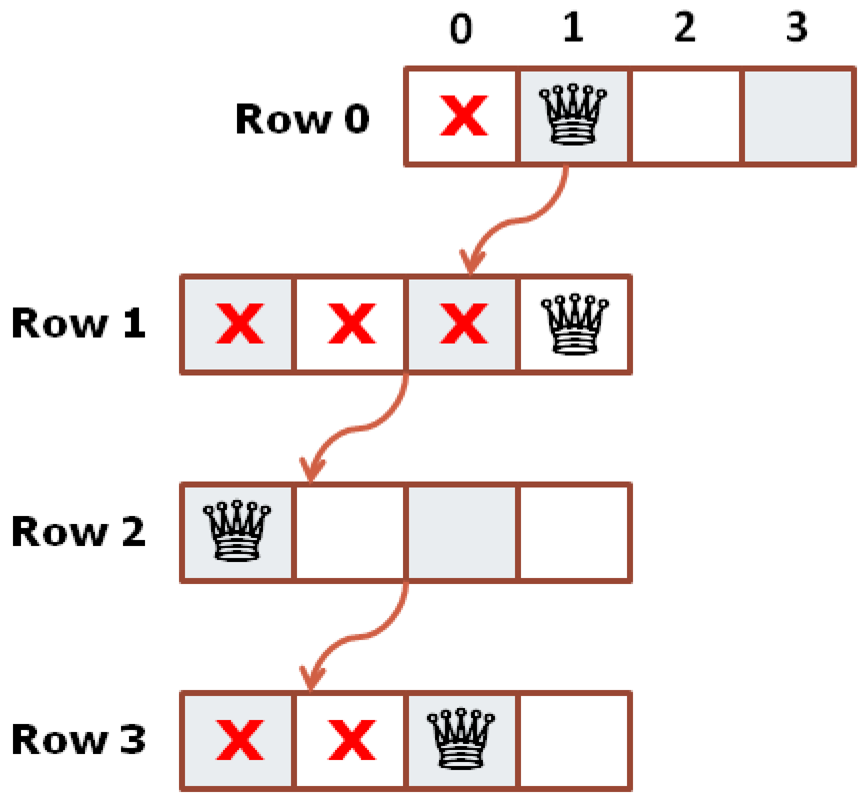
Row 0	(0,1)
Row 1	(1,3)

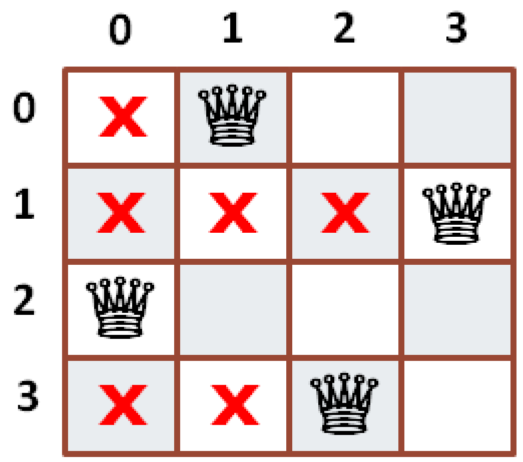




QueenPosition[ ]

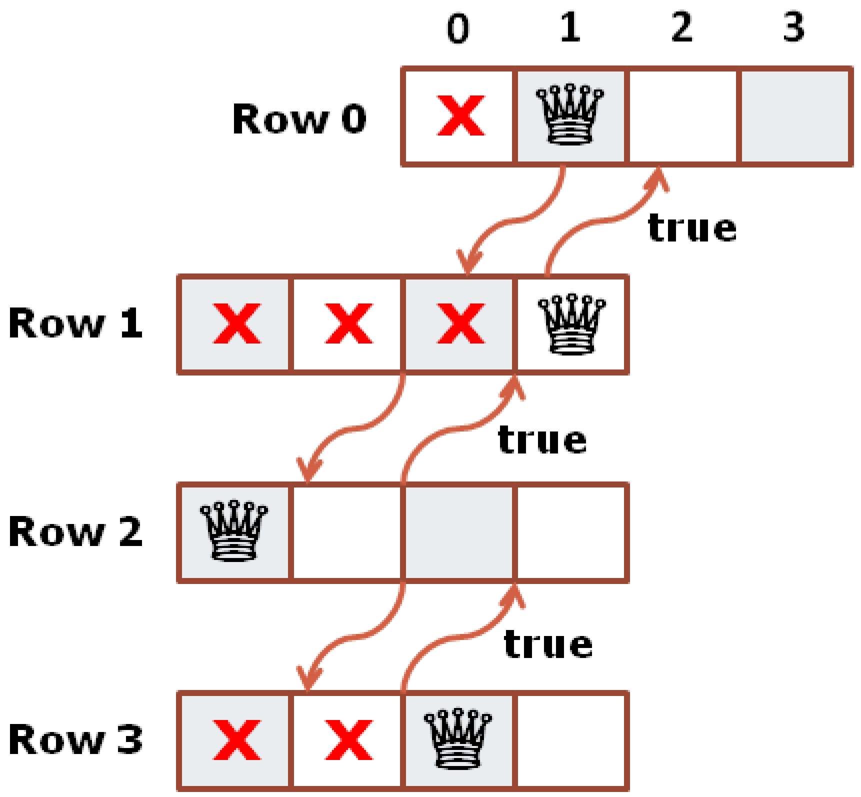
Row 0	(0,1)
Row 1	(1,3)
Row 2	(2,0)

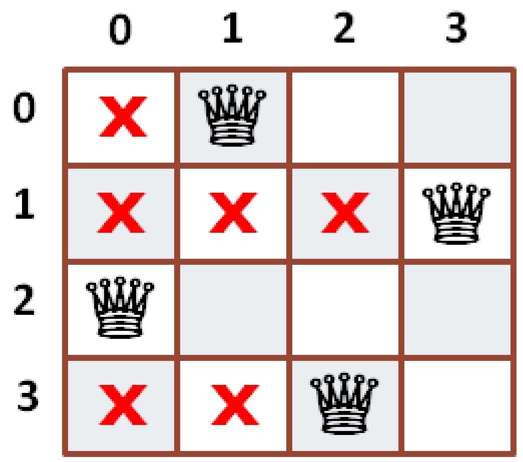




QueenPosition[ ]

Row 0	(0,1)
Row 1	(1,3)
Row 2	(2,0)
Row 3	(3,2)





QueenPosition[ ]

Row 0	(0,1)
Row 1	(1,3)
Row 2	(2,0)
Row 3	(3,2)

All functions will return true to their calling function.
It means all queens are placed on the board such that they are not attacking each other.

Recursion with Backtracking: n-Queen Problem

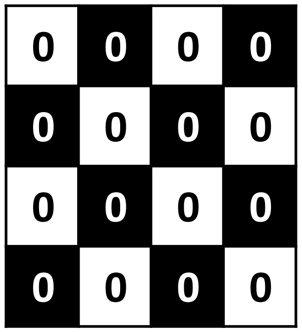
1. Find a safe column (from left to right) to place a queen, starting at the first row;
2. If we find a safe column, make recursive call to place a queen on the next row;
3. If we cannot find one, backtrack by returning from the recursive call to the previous row and find a different column.

	0	1	2	3
0		Q		
1				Q
2	Q			
3			Q	

0	1	0	0
0	0	0	1
1	0	0	0
0	0	1	0

N Queens with backtracking

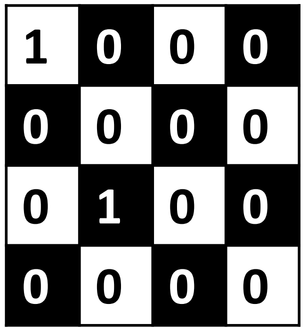
- int board[N][N] represents placement of queens
  - board[i][j] = 0: no queen at row i column j
  - board[i][j] = 1:queen at row i column j
- Initialize, for all i,j board[i][j] = 0
- Functions
  - PrintBoard(board): Prints board on the screen
  - IsSafe(borad, row, col): returns 1 iff new queen can be placed at (row,col) in board
  - Solve(board,col): recursively attempts to place (N-col) queens; returns 0 iff it failes



0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

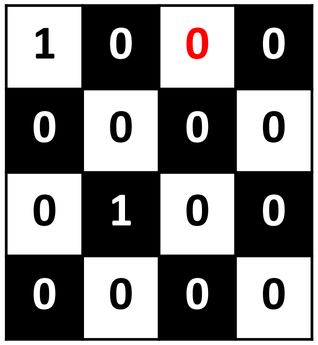
Initial board

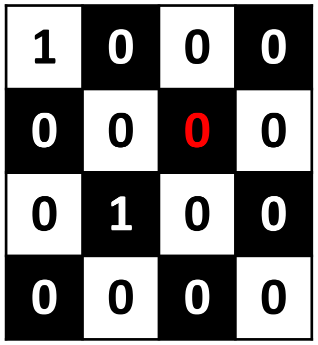
Solve(board,3) returns 0



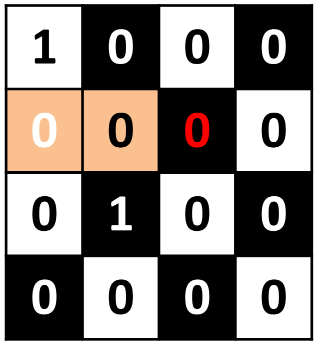
1	0	0	0
0	0	0	0
0	1	0	0
0	0	0	0

Warm up

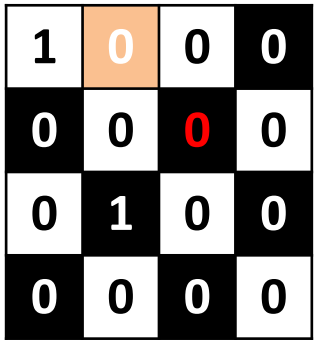


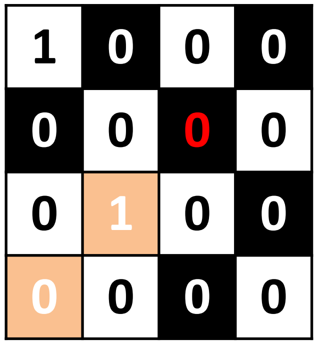


- PrintSolution(board): prints the board

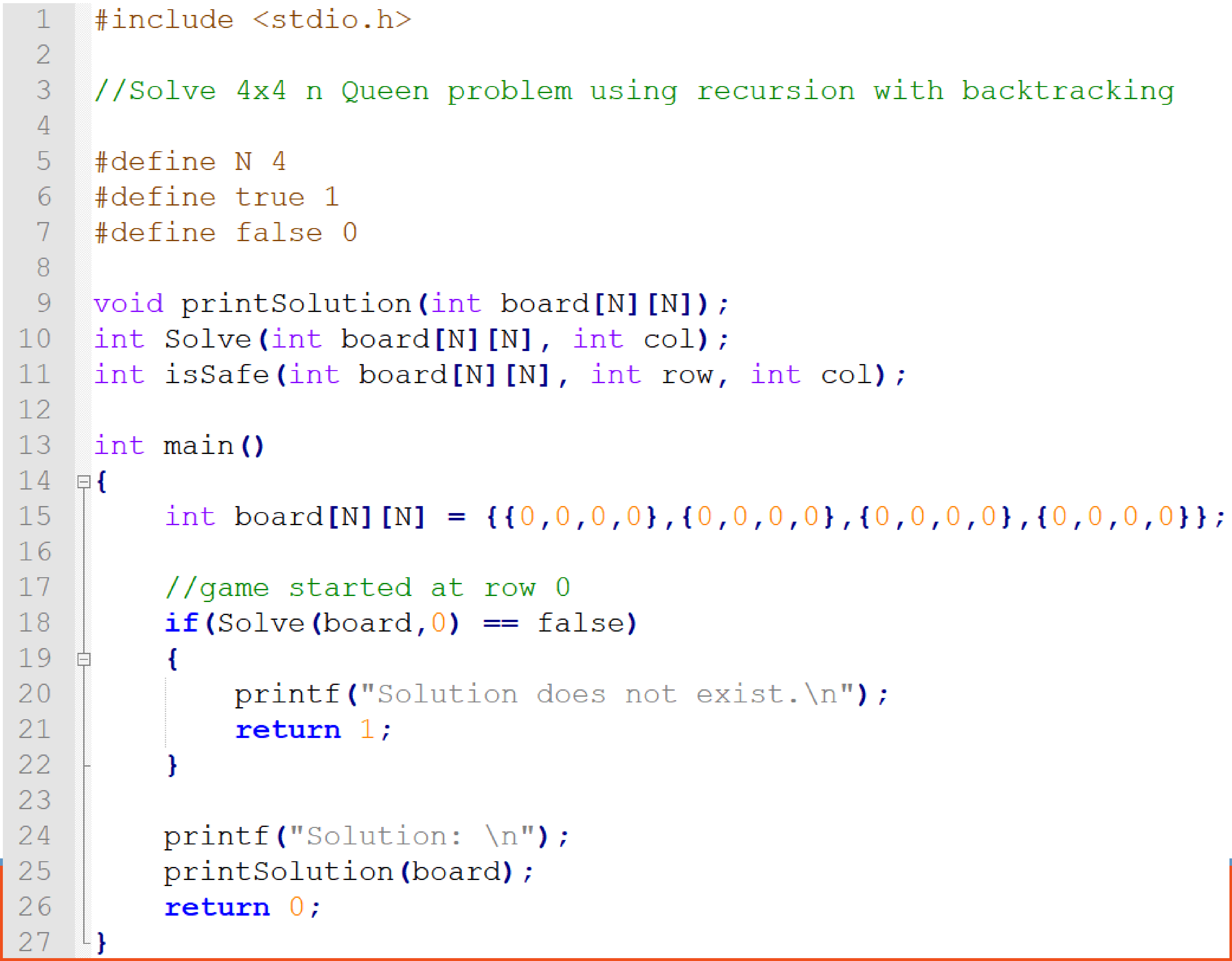


- int isSafe(board, row, col) checks if it is safe to place a queen at (row,col) within the given board.
  - Returns 1 if it can be placed
  - Returns 0 otherwise



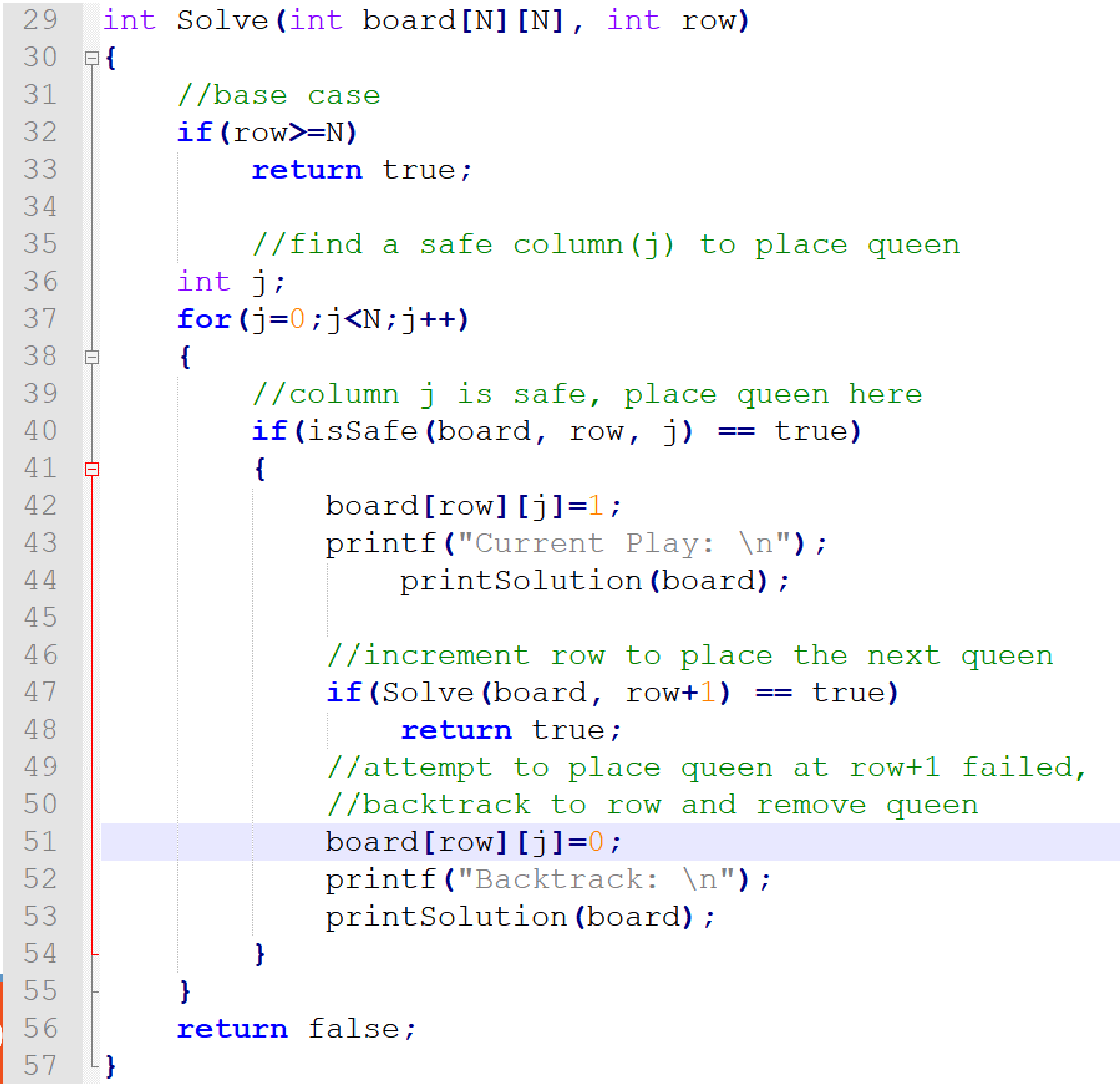


N-Queen (4x4) Backtracking – CODE (Main function)



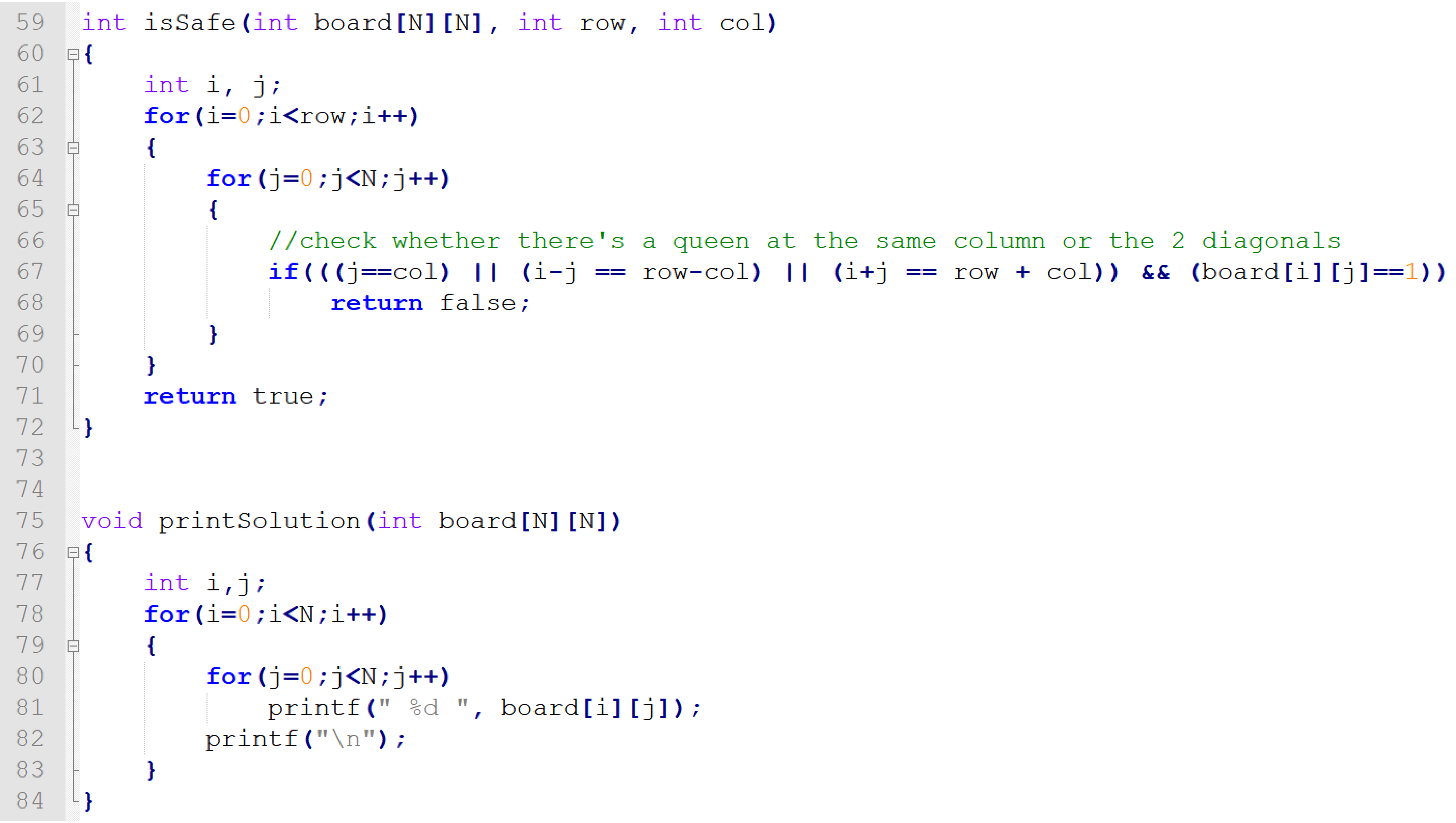
```c
#include <stdio.h>

//Solve 4x4 n Queen problem using recursion with backtracking

#define N 4
#define true 1
#define false 0

void printSolution(int board[N][N]);
int Solve(int board[N][N], int col);
int isSafe(int board[N][N], int row, int col);

int main()
{
    int board[N][N] = {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};

    //game started at row 0
    if(Solve(board,0) == false)
    {
        printf("Solution does not exist.\n");
        return 1;
    }

    printf("Solution: \n");
    printSolution(board);
    return 0;
}
```

N-Queen (4x4) Backtracking – CODE (Solve function)



```c
int Solve(int board[N][N], int row)
{
    //base case
    if(row>=N)
        return true;

        //find a safe column(j) to place queen
    int j;
    for(j=0;j<N;j++)
    {
        //column j is safe, place queen here
        if(isSafe(board, row, j) == true)
        {
            board[row][j]=1;
            printf("Current Play: \n");
                printSolution(board);

            //increment row to place the next queen
            if(Solve(board, row+1) == true)
                return true;
            //attempt to place queen at row+1 failed,
            //backtrack to row and remove queen
            board[row][j]=0;
            printf("Backtrack: \n");
            printSolution(board);
        }
    }
    return false;
}
```

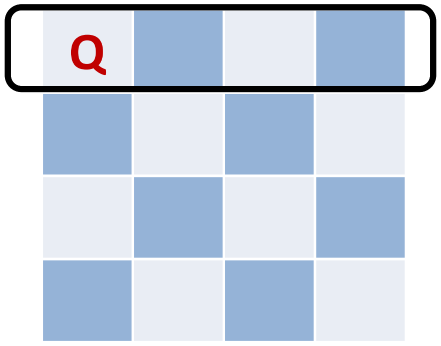
N-Queen (4x4) Backtracking – CODE (isSafe & PrintSolution functions)



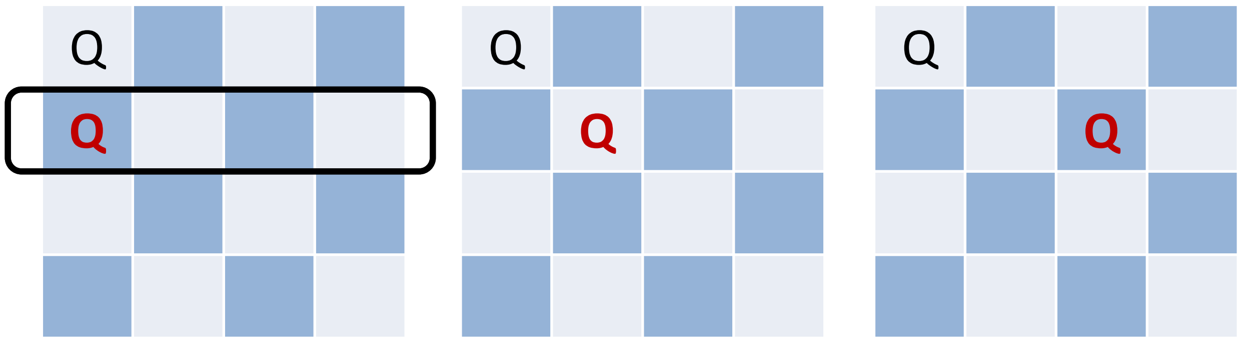
```c
int isSafe(int board[N][N], int row, int col)
{
    int i, j;
    for(i=0;i<row;i++)
    {
        for(j=0;j<N;j++)
        {
            //check whether there's a queen at the same column or the 2 diagonals
            if(((j==col) || (i-j == row-col) || (i+j == row + col)) && (board[i][j]==1))
                return false;
        }
    }
    return true;
}


void printSolution(int board[N][N])
{
    int i,j;
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}
```

Example: 4x4 n-Queen

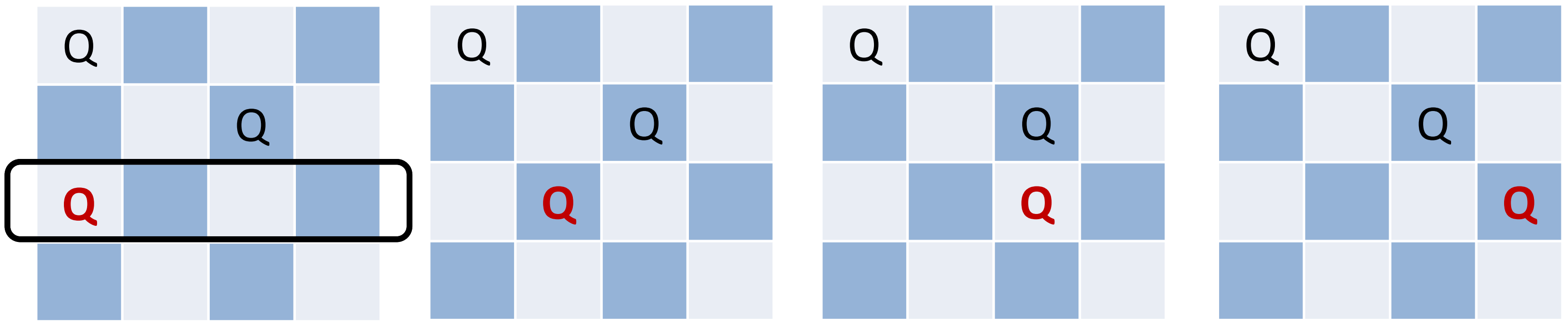
row 0:



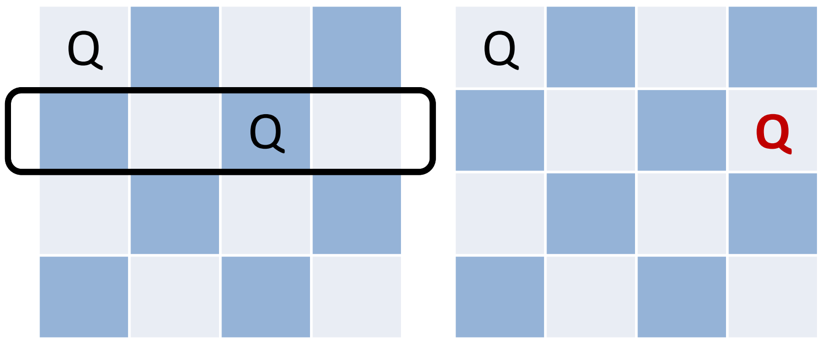
row 1:



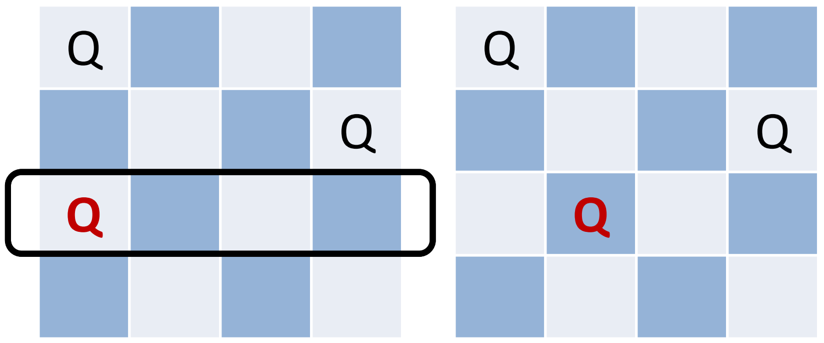
row 2:



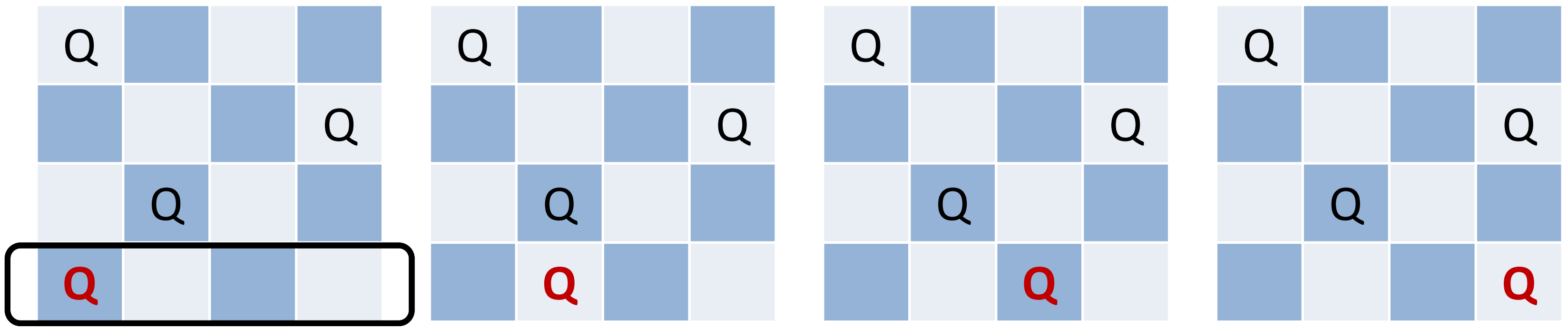
Backtrack to row 1 and make a new choice:



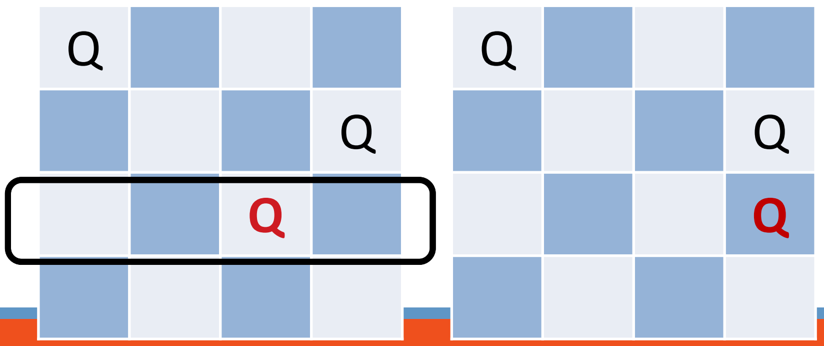
row 2:



row 3:

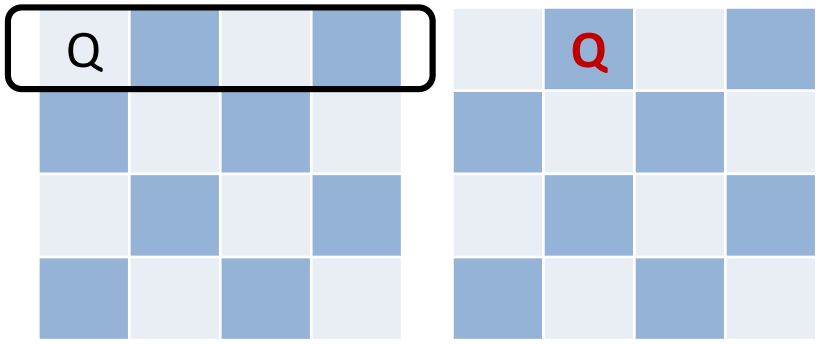


Backtrack to row 2 and make a new choice:

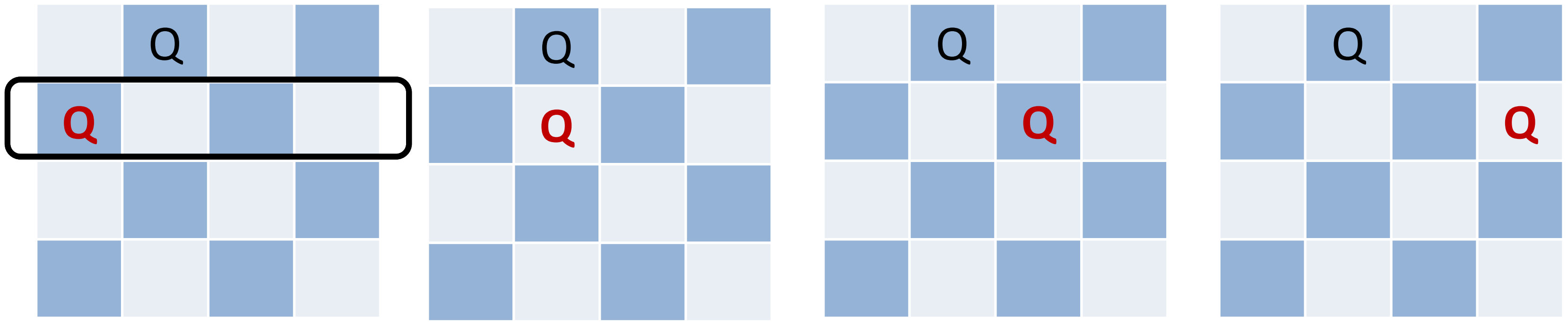


**Backtrack to row 1, but no columns left

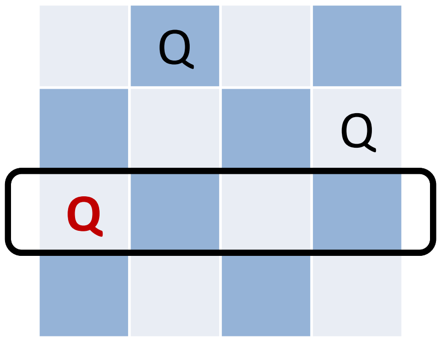
Backtrack to row 0 and make a new choice:



row 1:



row 2:



row 3: