ECE 220 Computer Systems & Programming

Lecture 13 – Recursive sorting and Recursion with Backtracking

ECE ILLINOIS

ILLINOIS

Binary Search (recursive)

```c
// C program to implement binary search using recursion
#include <stdio.h>

// A recursive binary search function. It returns location
// of x in given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    // checking if there are elements in the subarray
    if (r >= l) {

        // calculating mid point
        int mid = (l + r) / 2;

        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then it can only
        // be present in left subarray
        if (arr[mid] > x) {
            return binarySearch(arr, l, mid - 1, x);
        }

        // Else the element can only be present in right
        // subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}
```

Insertion Sort

1. Remove item from array, insert it at the proper location in the sorted part by shifting other items.
2. Repeat this process until the end of array is reached.


```
void insert_sort(int *a, int size)
{

    int sorted_ind,unsortedItem,unsorted_ind;

    for(unsorted_ind=1;unsorted_ind<size;unsorted_ind++)
    {
        unsortedItem=a[unsorted_ind];
        sorted_ind=unsorted_ind-1;
        while((sorted_ind>=0) && (unsortedItem<a[sorted_ind]))
        {
            a[sorted_ind+1]=a[sorted_ind];
            sorted_ind--;
        }

        a[sorted_ind+1]=unsortedItem;
    }

}
```

Recursive Insertion sort

```c
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    int unsorted_ind=1;
    insertsort_recur(arr, n, unsorted_ind);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
   printf("\n");
    return 0;
}
```

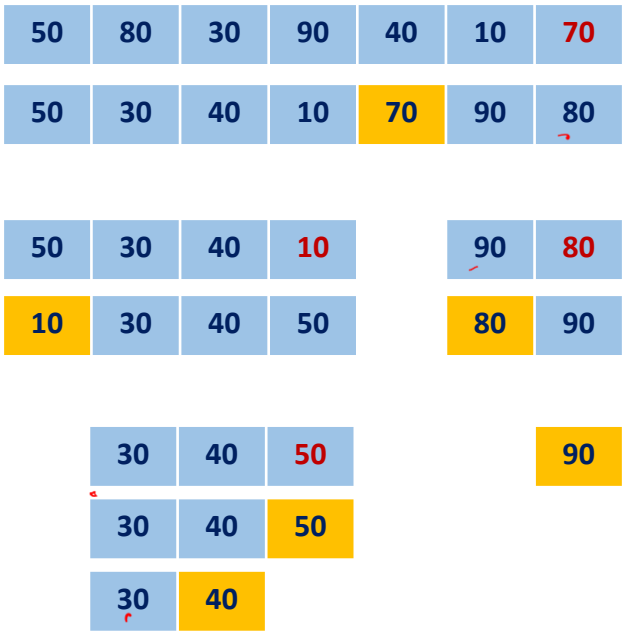
```c
#include<stdio.h>
void insertsort_recur(int *a, int size, int unsorted_ind)
{

        int sorted_ind,unsortedItem;
        if (unsorted_ind==size)
                return;

        else{
                unsortedItem=a[unsorted_ind];
                sorted_ind=unsorted_ind-1;
                while((sorted_ind>=0) && (unsortedItem<a[sorted_ind]))
                {
                        a[sorted_ind+1]=a[sorted_ind];
                        sorted_ind--;
                }

                a[sorted_ind+1]=unsortedItem;
        }
        unsorted_ind++;
        insertsort_recur(a,size,unsorted_ind);
}
```

```c
void insert_sort(int *a, int size)
{

    int sorted_ind,unsortedItem,unsorted_ind;

    for(unsorted_ind=1;unsorted_ind<size;unsorted_ind++)
    {
        unsortedItem=a[unsorted_ind];
        sorted_ind=unsorted_ind-1;
        while((sorted_ind>=0) && (unsortedItem<a[sorted_ind]))
        {
            a[sorted_ind+1]=a[sorted_ind];
            sorted_ind--;
        }

        a[sorted_ind+1]=unsortedItem;
    }

}
```

```c
#include<stdio.h>
void insertsort_recur(int *a, int size, int unsorted_ind)
{

        int sorted_ind,unsortedItem;
        if (unsorted_ind==size)
                return;

        else{
                unsortedItem=a[unsorted_ind];
                sorted_ind=unsorted_ind-1;
                while((sorted_ind>=0) && (unsortedItem<a[sorted_ind]))
                {
                        a[sorted_ind+1]=a[sorted_ind];
                        sorted_ind--;
                }

                a[sorted_ind+1]=unsortedItem;
        }
        unsorted_ind++;
        insertsort_recur(a,size,unsorted_ind);
}
```

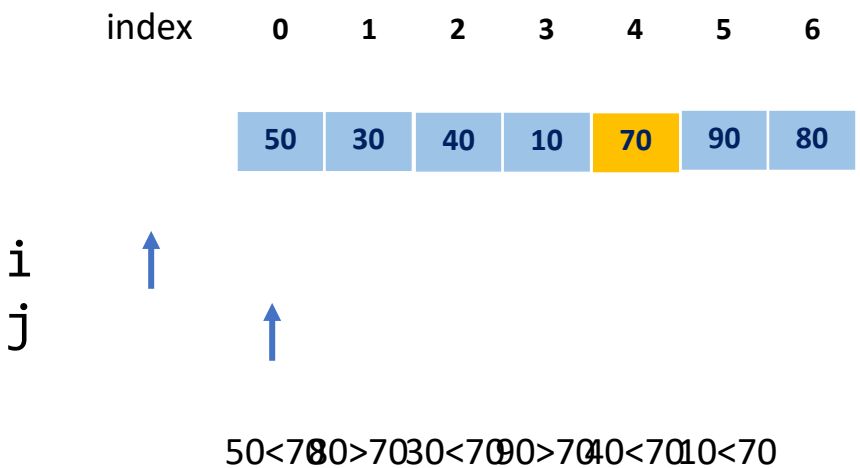
Quick Sort (divide-and-conquer)

1. Pick a pivot and partition array into 2 subarrays (smaller elements than the pivot in the left and greater elements in the right)
2. Sort the 2 subarrays using the same method



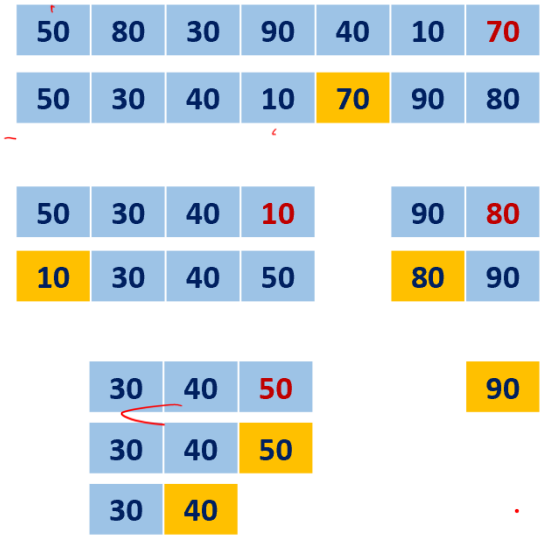
Quick Sort: Partition

```
i: index of smaller elements (i=low-1)
j: loop index (j=low) -> low to (high-1)
```



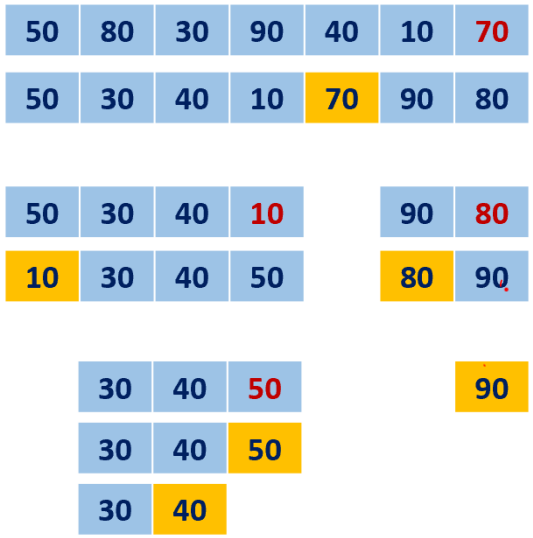

```
for j from low to high-1{
    if array[j] < pivot
        i = i + 1;
        swap array[i] and array[j]
}
swap array[i+1] and pivot
return i+1 as the pivot index
```

Using STACK in Quick Sort




```
39  void quickSortIterative(int arr[], int l, int h)
40  {
41      // Create an auxiliary stack
42      int stack[h - l + 1];
43      // initialize top of stack
44      int top = h-l+1;
45      int tos=h-l+1;
46      // push initial values of l and h to stack
47      stack[--top] = l;
48      stack[--top] = h;
49      // Keep popping from stack while is not empty
50      while (top < tos) {
51          // Pop h and l
52          h = stack[top++];
53          l = stack[top++];
54          // Set pivot element at its correct position
55          // in sorted array
56          int p = partition(arr, l, h);
57          // If there are elements on left side of pivot,
58          // then push left side to stack
59          if (p - 1 > l) {
60              stack[--top] = l;
61              stack[--top] = p-1;
62          }
63          // If there are elements on right side of pivot,
64          // then push right side to stack
65          if (p + 1 < h) {
66              stack[--top] = p+1;
67              stack[--top] =h;
68          }
69      }
70  }
```

Recursive Quick Sort




```
// The QuickSort function implementation
void quickSort(int arr[], int low, int high) {
    if (low < high) {

        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // Recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, pi + 1, high);
        quickSort(arr, low, pi - 1);
    }
}
```

```
void quickSortIterative(int arr[], int l, int h)
{
    // Create an auxiliary stack
    int stack[h - l + 1];
    // initialize top of stack
    int top = h-l+1;
    int tos=h-l+1;
    // push initial values of l and h to stack
    stack[--top] = l;
    stack[--top] = h;
    // Keep popping from stack while is not empty
    while (top < tos) {
        // Pop h and l
        h = stack[top++];
        l = stack[top++];
        // Set pivot element at its correct position
        // in sorted array
        int p = partition(arr, l, h);
        // If there are elements on left side of pivot,
        // then push left side to stack
        if (p - 1 > l) {
            stack[--top] = l;
            stack[--top] = p-1;
        }
        // If there are elements on right side of pivot,
        // then push right side to stack
        if (p + 1 < h) {
            stack[--top] = p+1;
            stack[--top] =h;
        }
    }
}
```

Activation Records Build up and Tear Down

```
// The QuickSort function implementation
void quickSort(int arr[], int low, int high) {
    if (low < high) {

        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // Recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, pi + 1, high);
        quickSort(arr, low, pi - 1);
    }
}
```

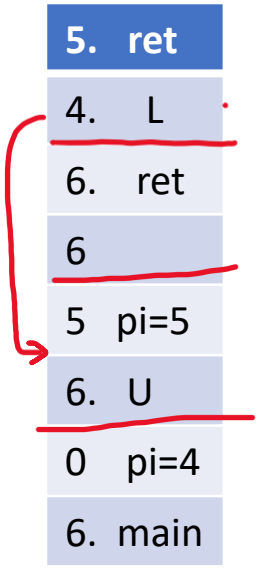
0	1	2	3	4	5	6
50	80	30	90	40	10	70
50	30	40	10	70	90	80

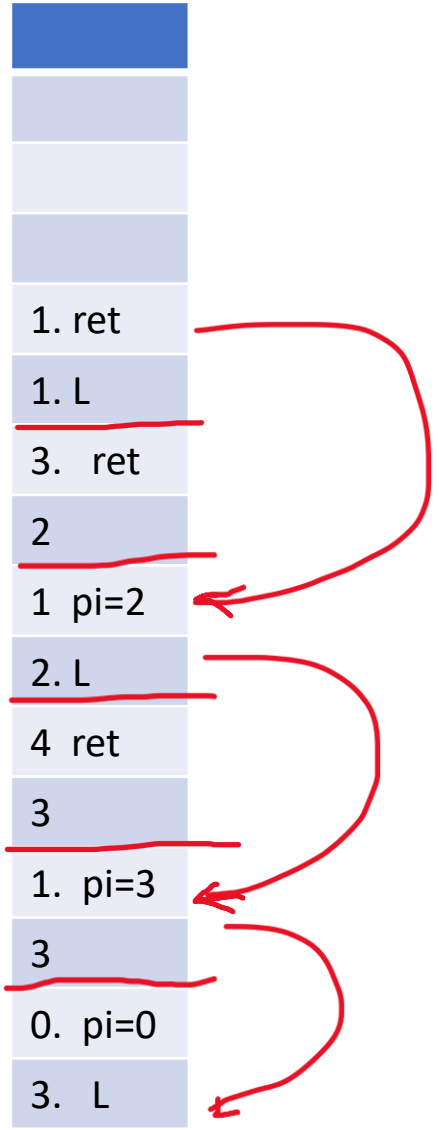
50	30	40	10		90	80
10	30	40	50		80	90

	30	40	50			90
	30	40	50			
	30	40				

When (**low<high**) is violated, it destroys caller activation records.

When returning to the **next line of L**, it tears down caller activation records

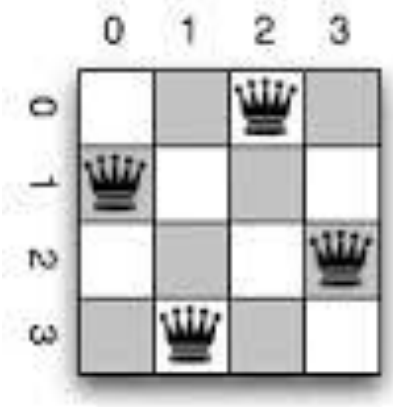




Recursive Backtracking:

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem statement.

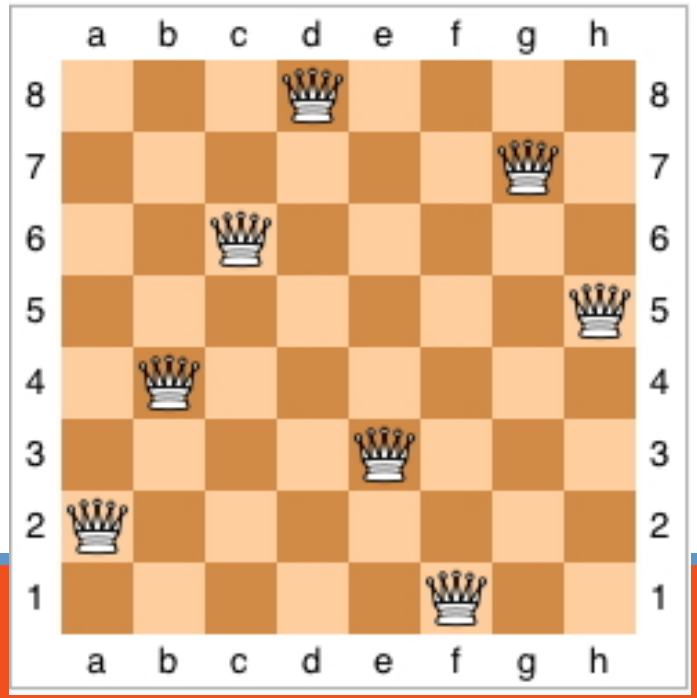
N queens problem using recursive Backtracking

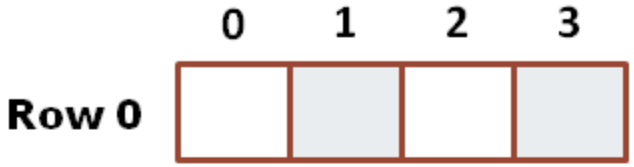


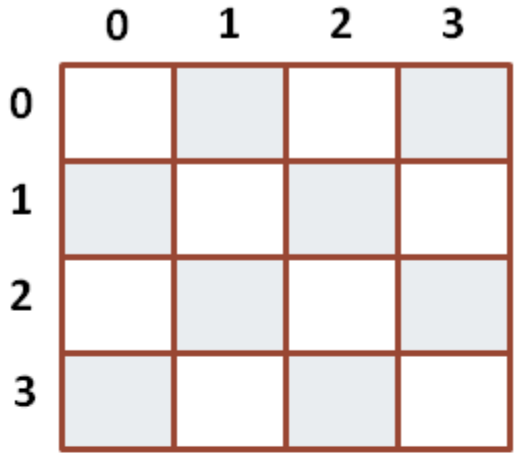
0	0	1	0
1	0	0	0
0	0	0	1
0	1	0	0

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

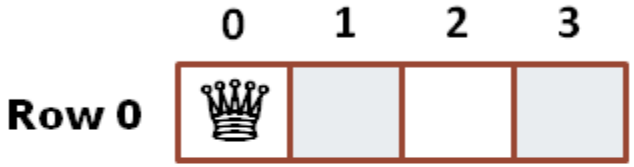
- Brute force: total number of possible placements:
- ~N^2 Choose N ~ 4.4 B (N=8)

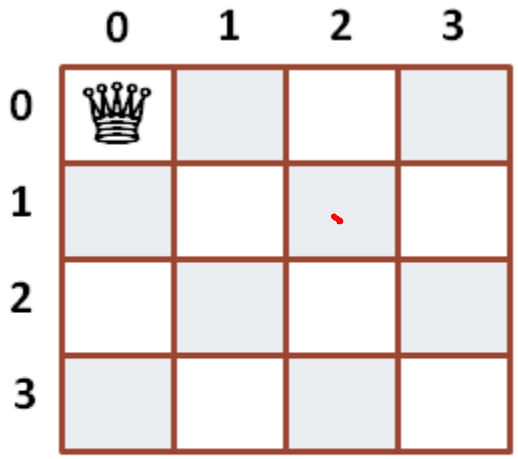






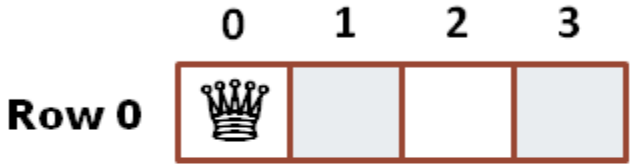
QueenPosition[ ]

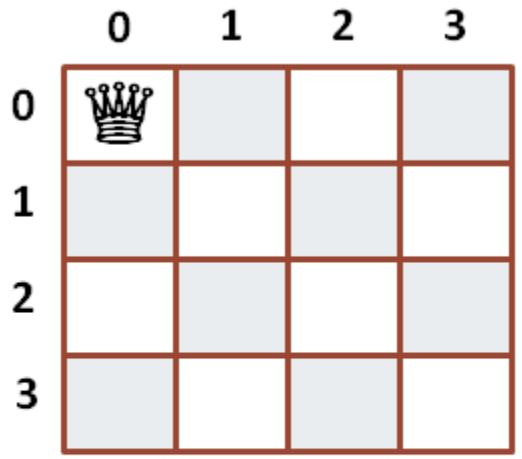




QueenPosition[ ]

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

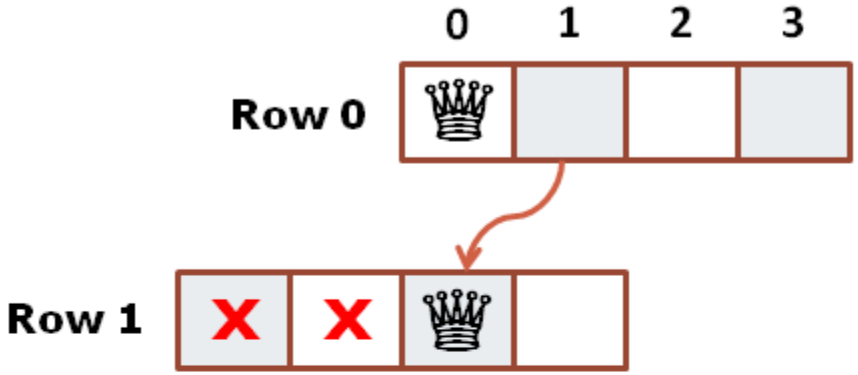


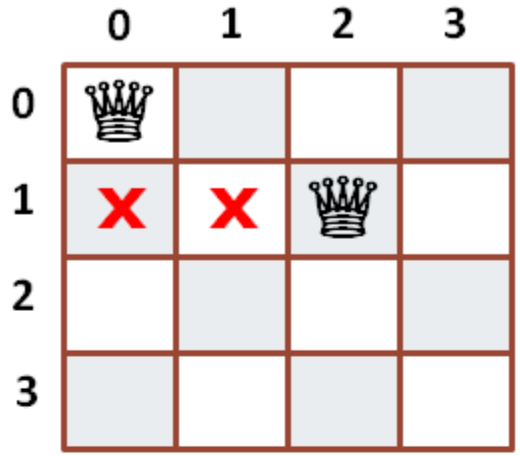


QueenPosition[ ]

Row 0 (0,0)

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





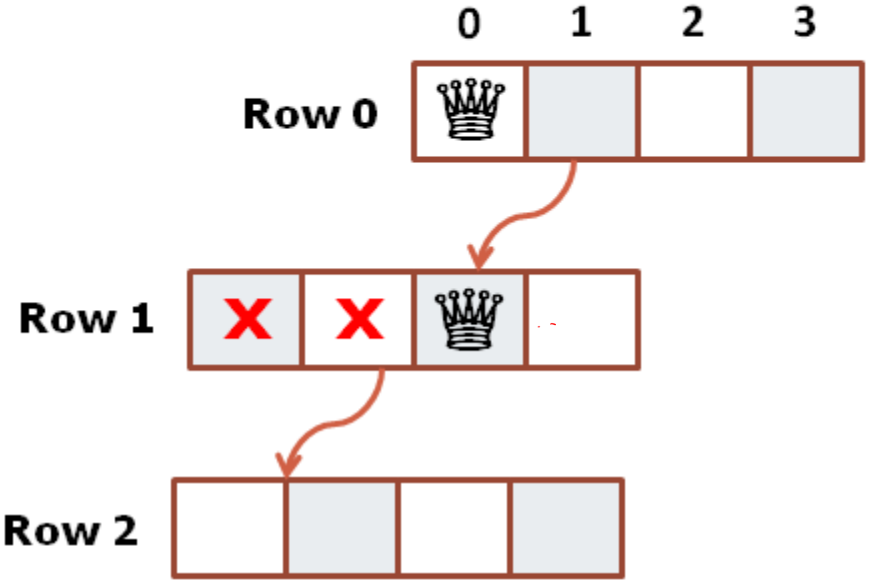
QueenPosition[ ]

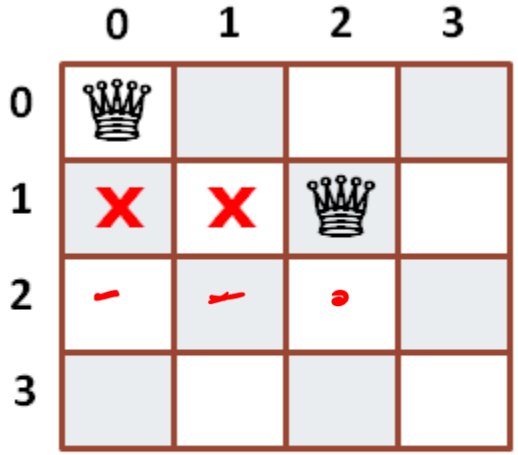
Row 0 (0,0)

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.



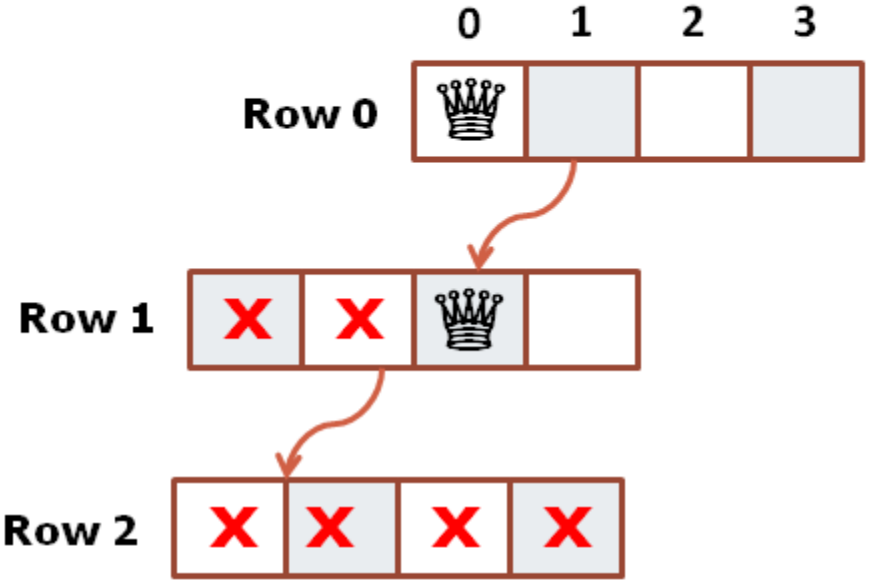


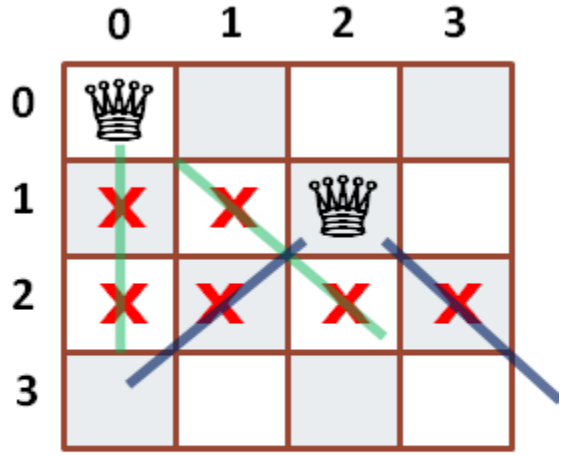
QueenPosition[ ]

Row 0 (0,0)

Row 1 (1,2)

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



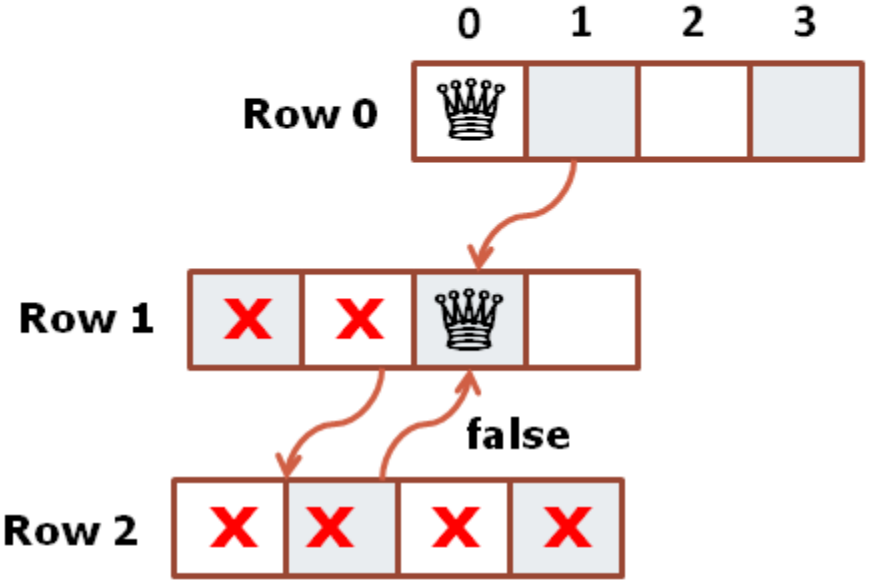


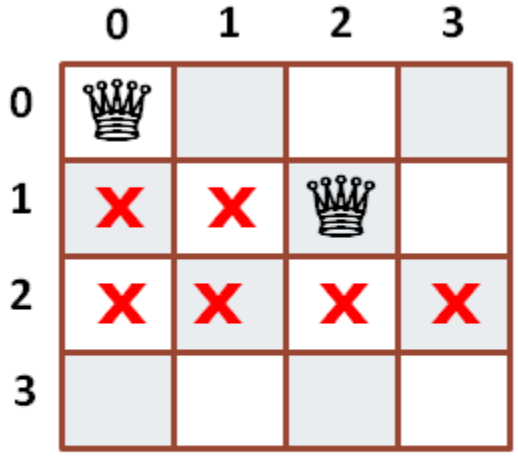
QueenPosition[ ]

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.



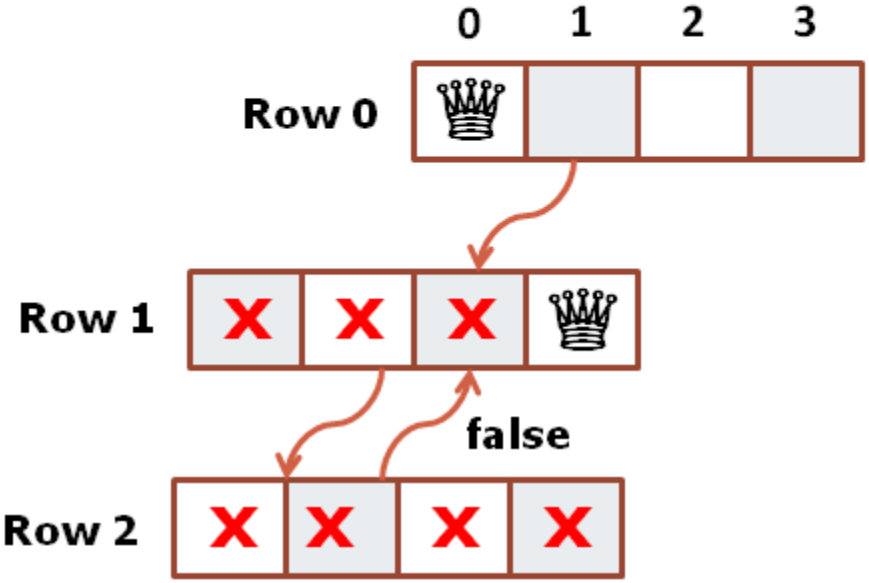


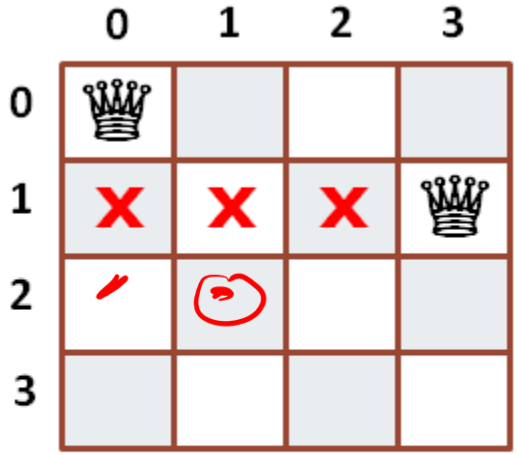
QueenPosition[ ]

Row 0 (0,0)

Row 1 (1,2)

So function will return false to the calling function.





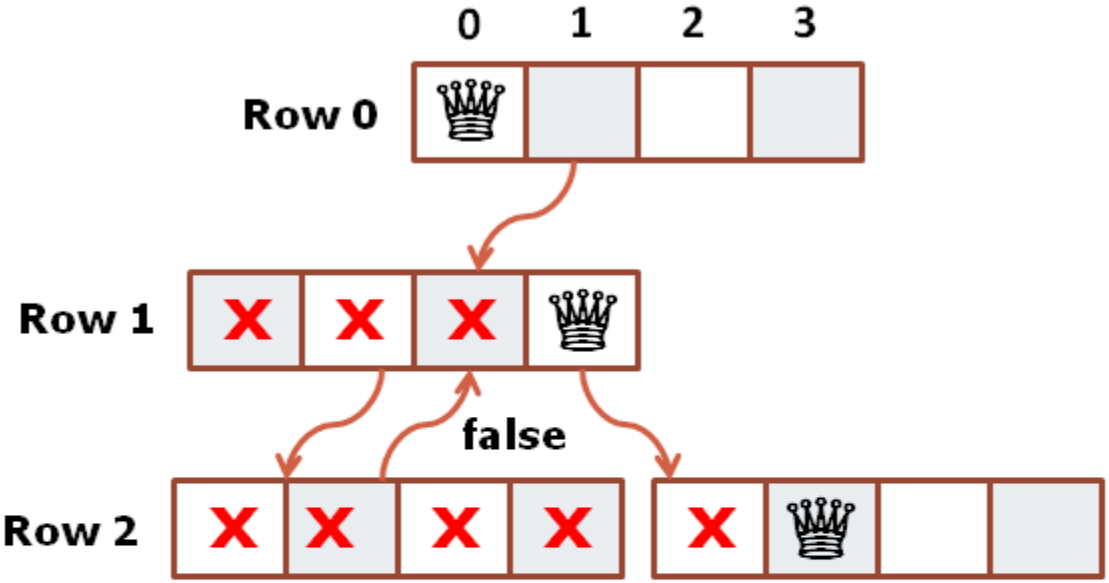
QueenPosition[ ]

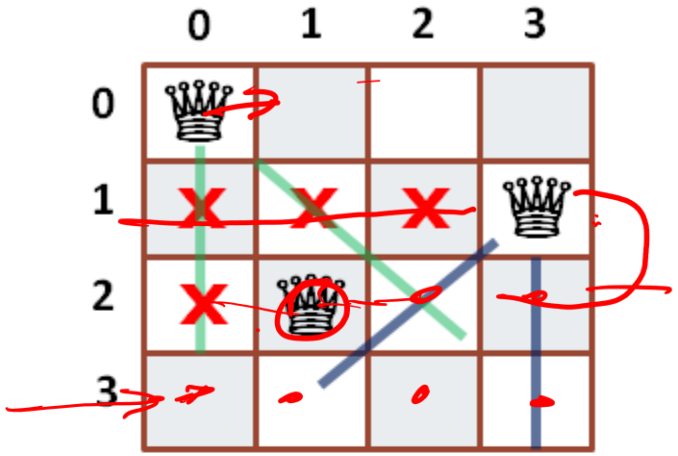
Row 0 (0,0)

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

Row 1 (1,3)

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



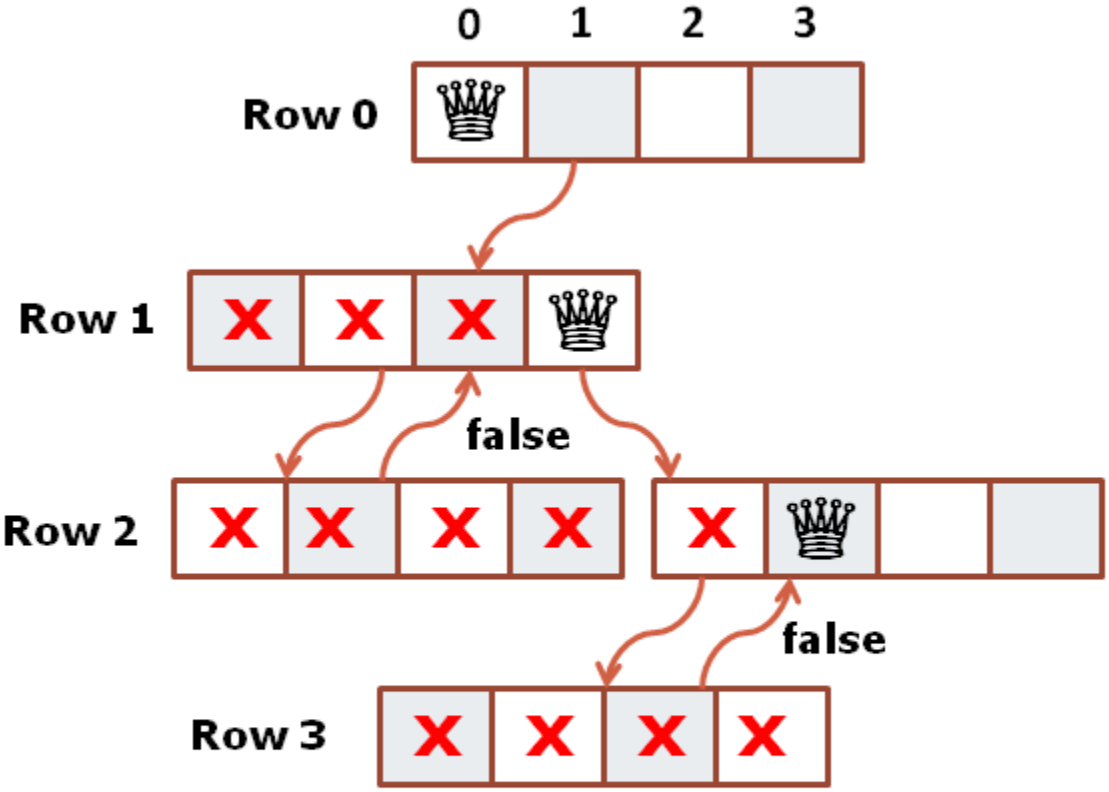


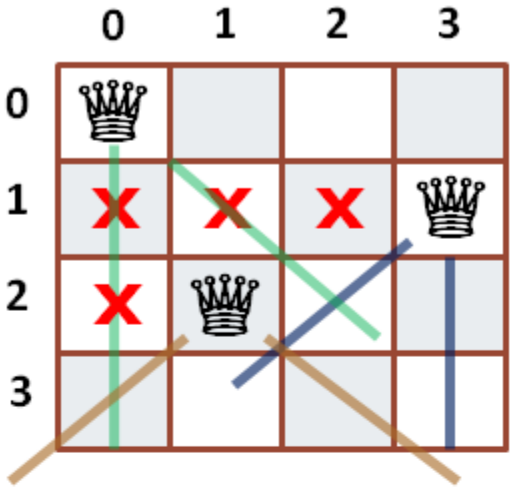
QueenPosition[ ]

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

Again find the cell on **2nd** 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.

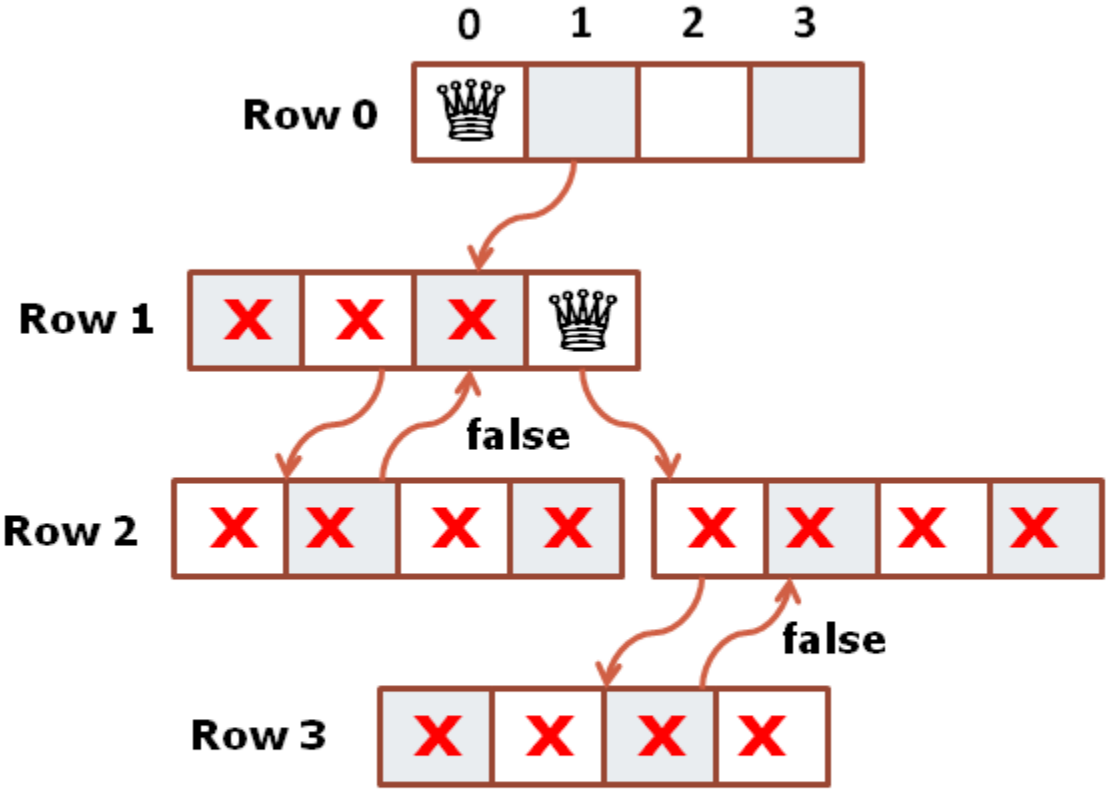


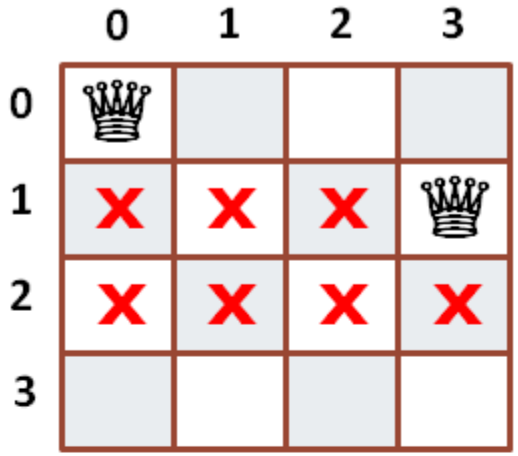


QueenPosition[ ]

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.





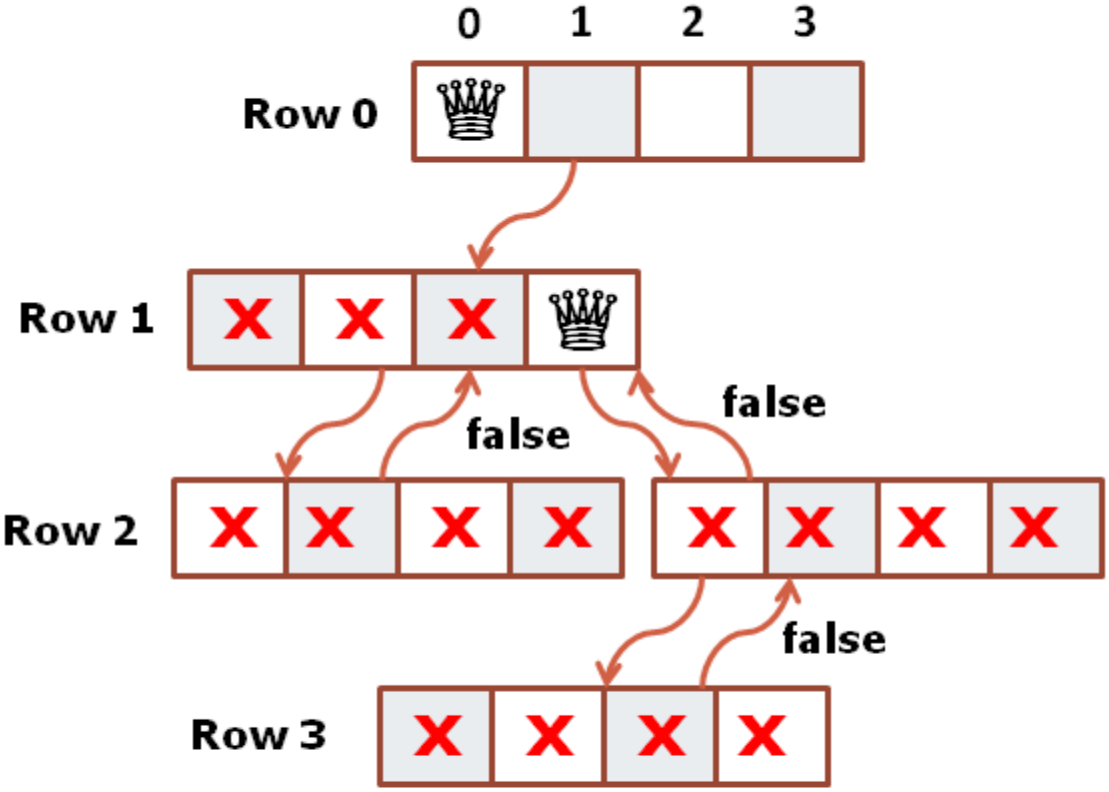
QueenPosition[ ]

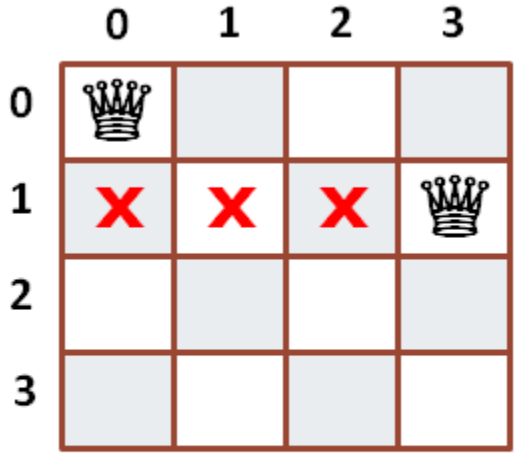
Row 0 (0,0)

Row 1 (1,3)

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

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

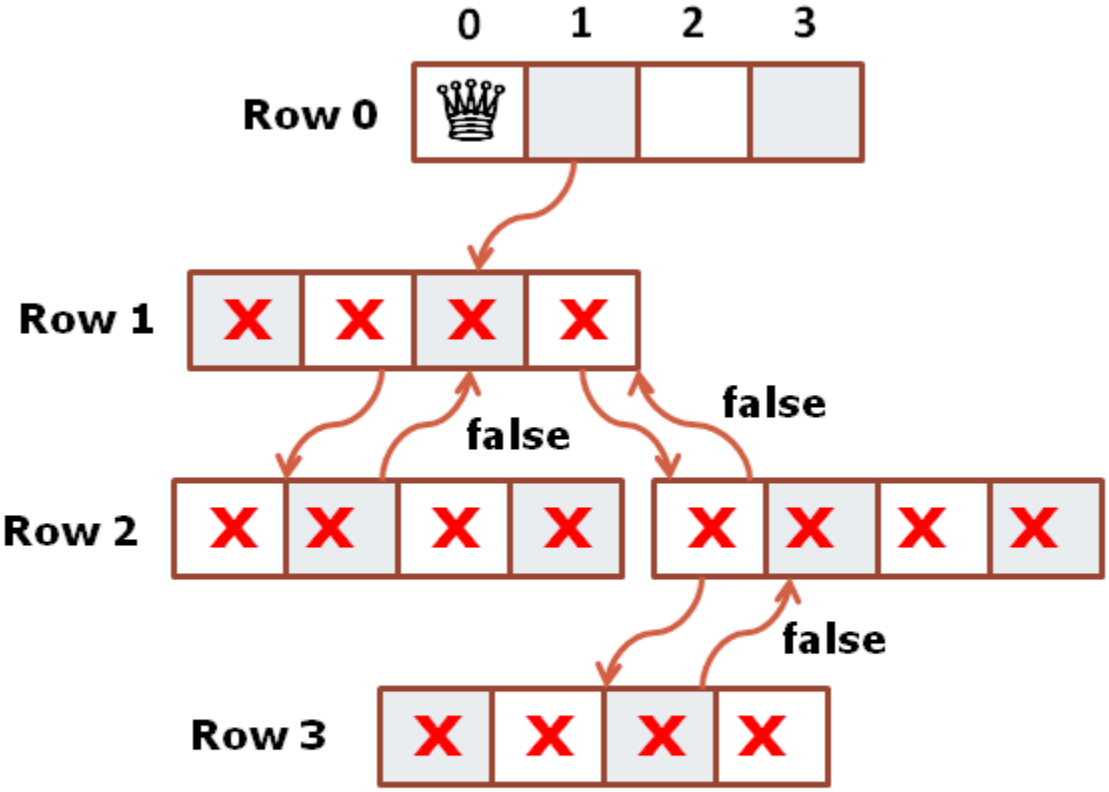


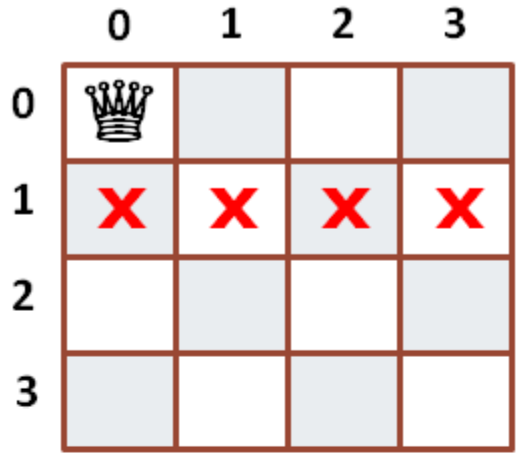


QueenPosition[ ]

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.



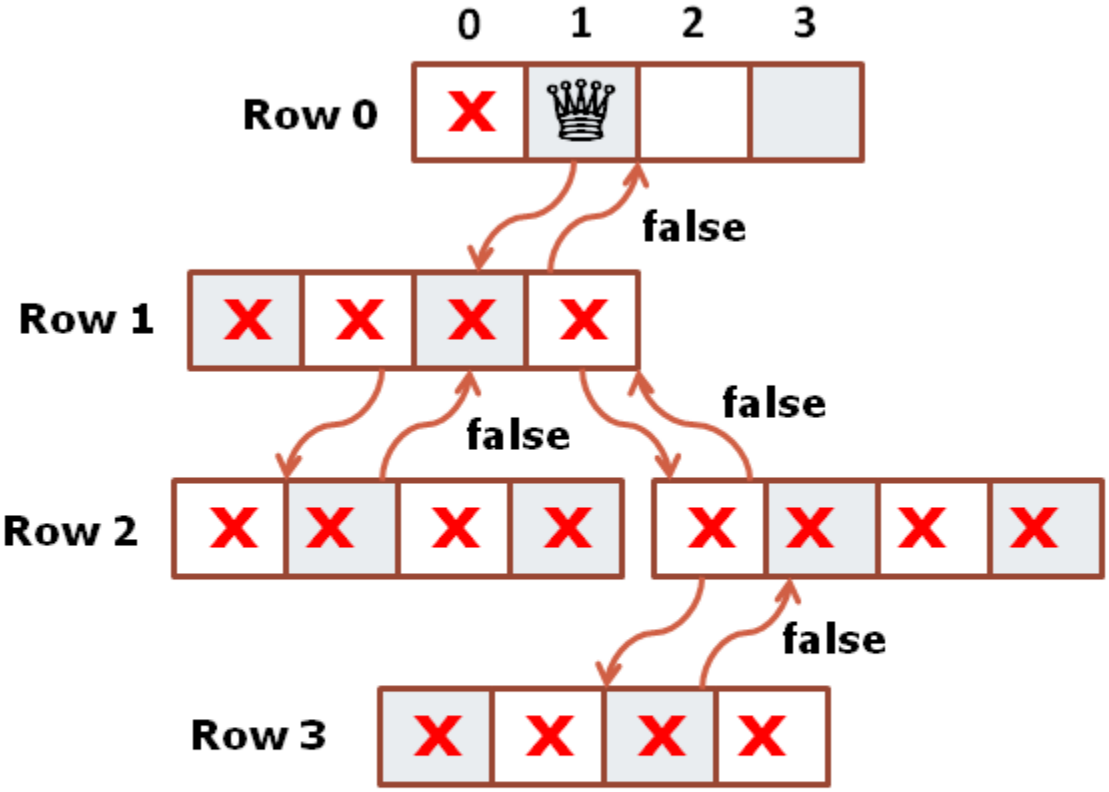


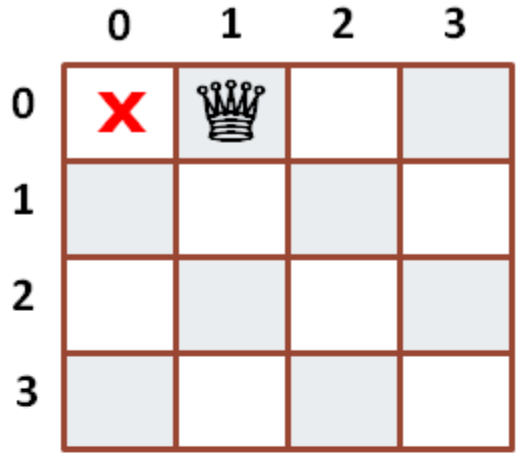
QueenPosition[ ]

Row 0 (0,0)

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

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



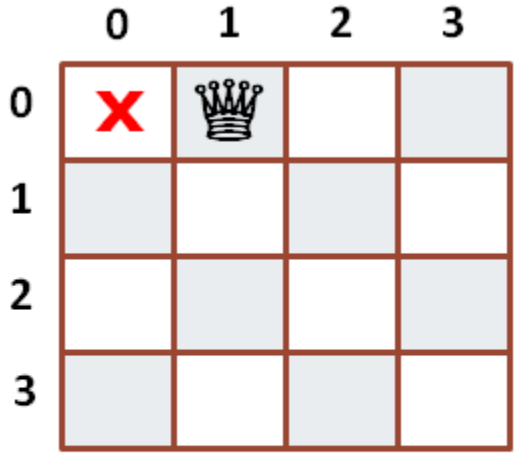


QueenPosition[ ]

~~**Row 0**~~ ~~(0,0)~~

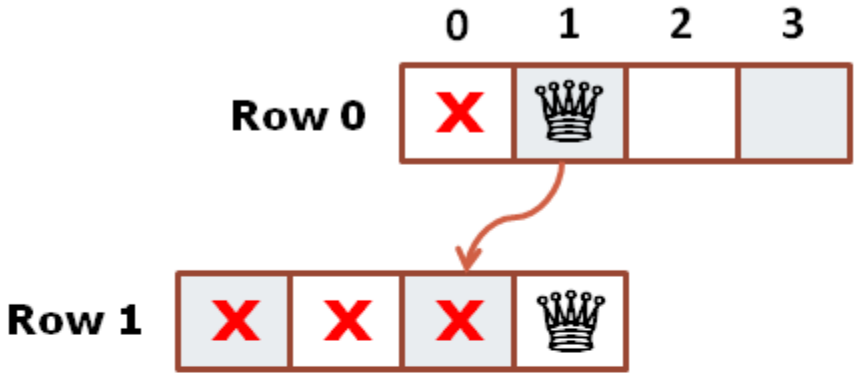
Row 0 (0,1)

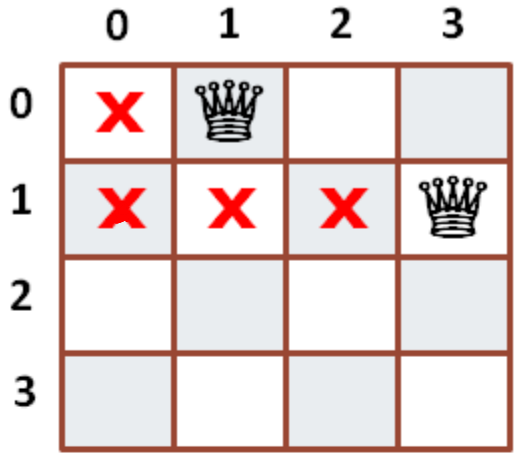
Queen at the **1st** row tries to find next safe cell.
Let us remove these failed recursion calls from the screen.



QueenPosition[ ]

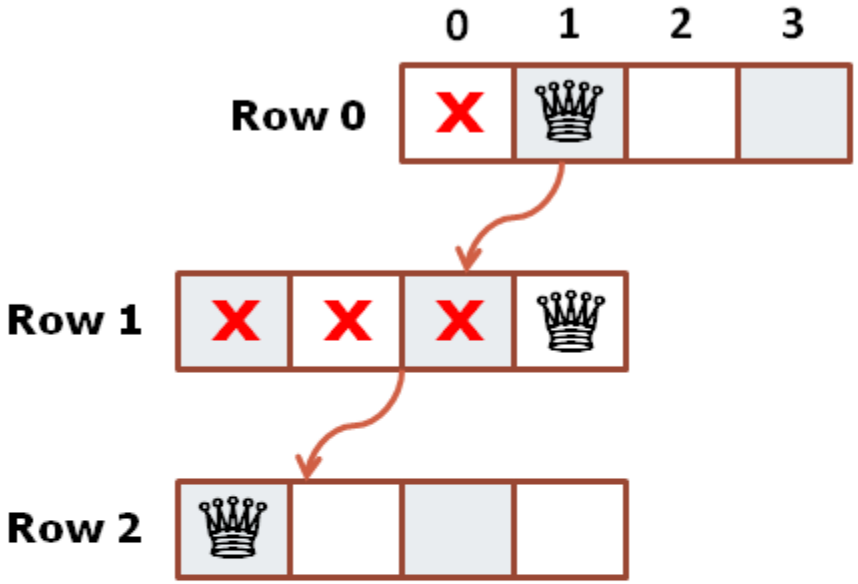
Row 0 (0,1)

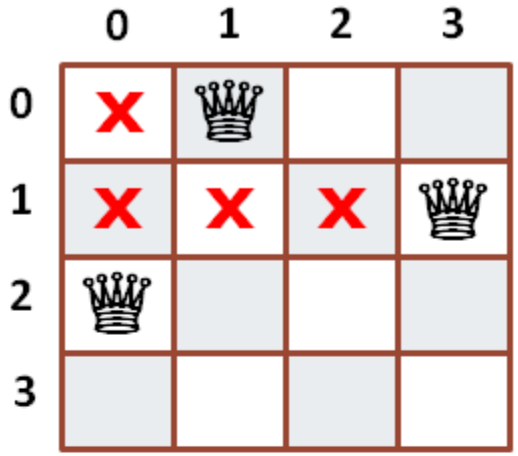




QueenPosition[ ]

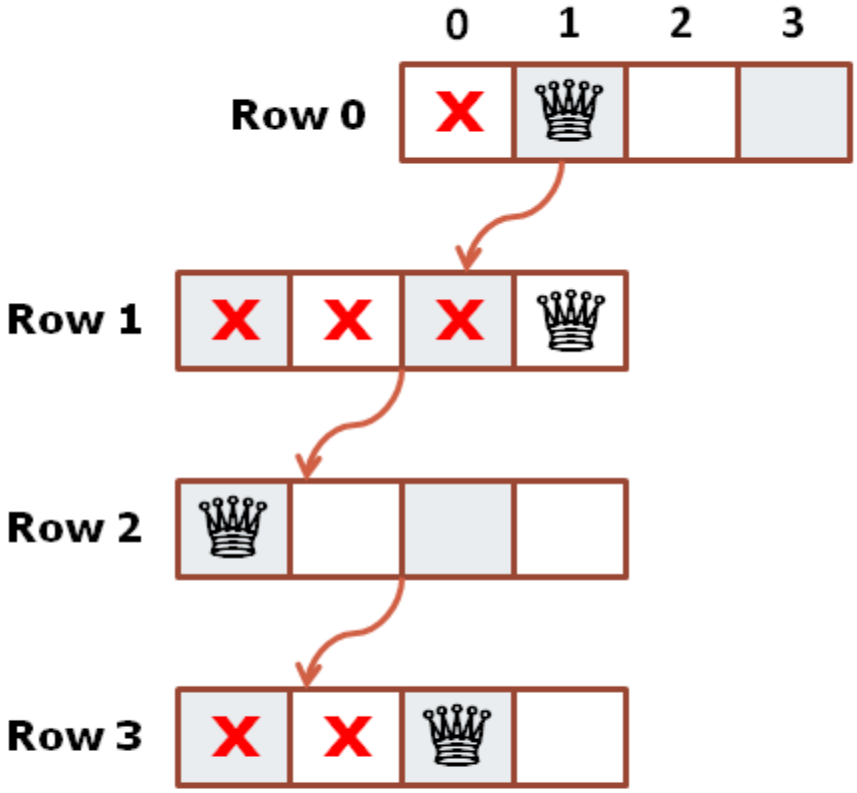
Row 0 (0,1)
Row 1 (1,3)

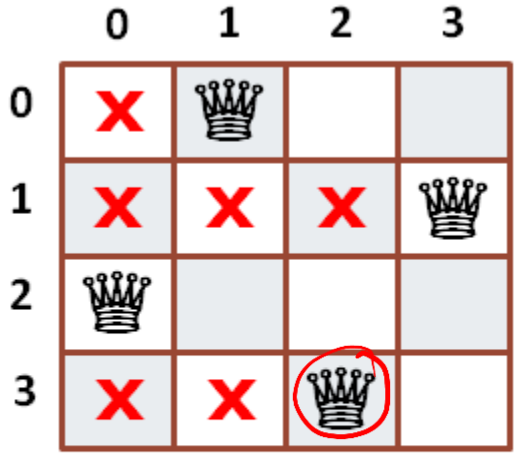




QueenPosition[ ]

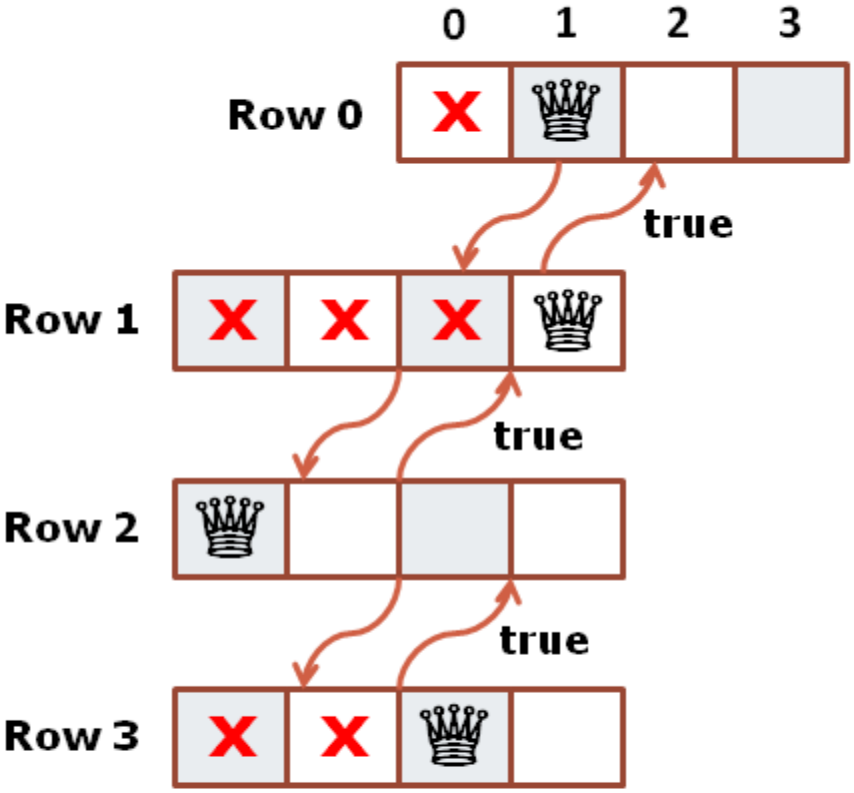
Row 0	(0,1)
Row 1	(1,3)
Row 2	(2,0)

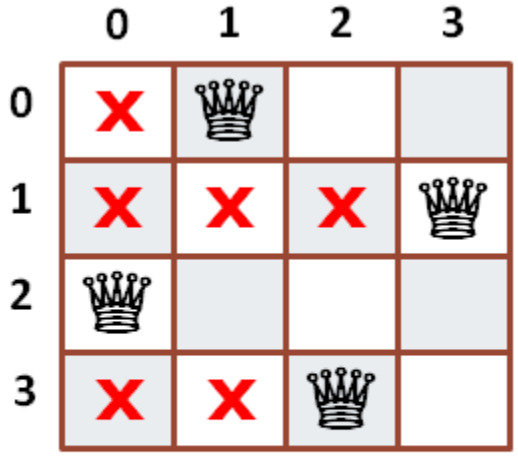




QueenPosition[ ]

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





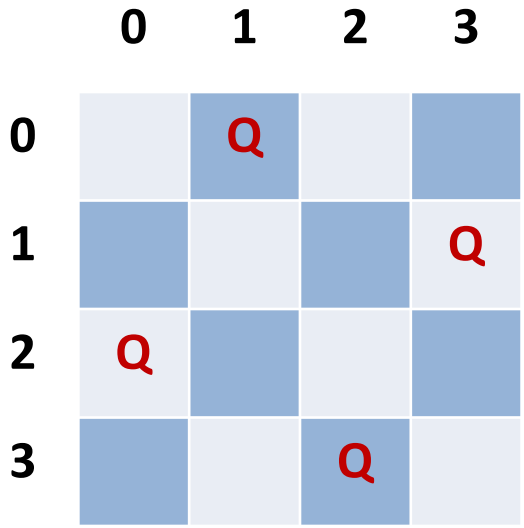
QueenPosition[ ]

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	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(board, row, col): returns 1 iff new queen can be placed at (row,col) in board
  - Solve(board, row): recursively attempts to place (N-row) queens; returns 0 iff it fails

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

Recursive with Backtracking

- N-Queen Problem by Backtracking
  1. **Decision**
     Place a queen at a safe place.
  2. **Recursion**
     Explore the solution for the next row.
  3. **Backtrack (Undo)**
     Remove the queen if no solution for the next row.
  4. **Base case**
     Reach the goal.

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

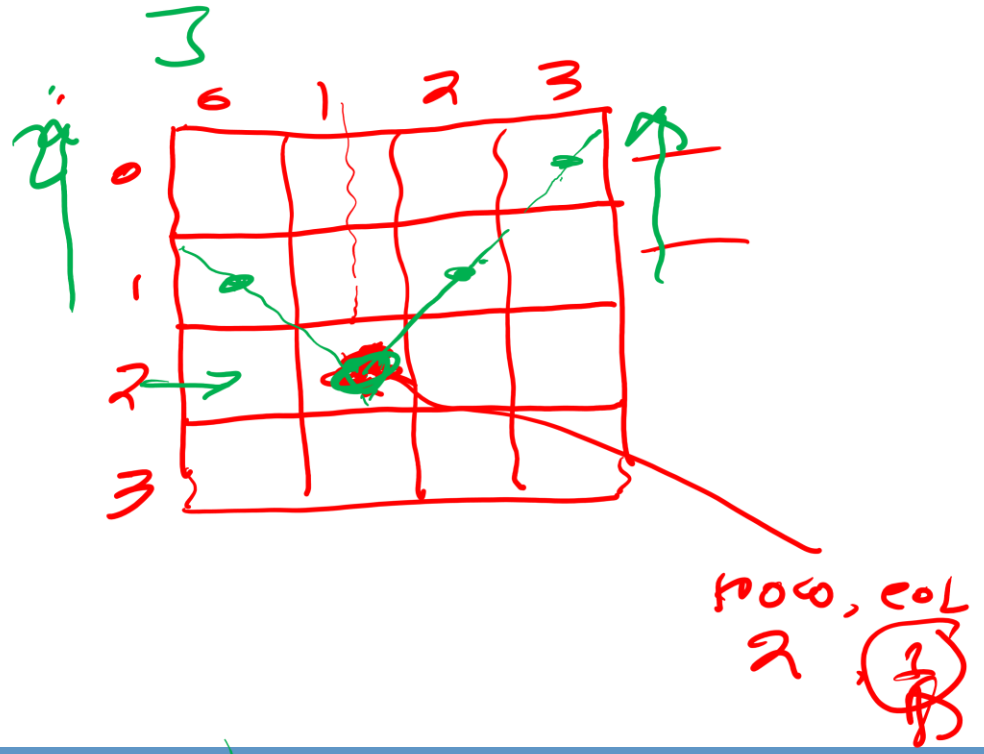
```c
#include <stdio.h>

//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)

```
29  int Solve(int board[N][N], int row)
30  {
31      //base case
32      if(row>=N)
33          return true;
34
35          //find a safe column(j) to place queen
36      int j;
37      for(j=0;j<N;j++)
38      {
39          //column j is safe, place queen here
40          if(isSafe(board, row, j) == true)
41          {
42              board[row][j]=1;
43              printf("Current Play: \n");
44                  printSolution(board);
45
46              //increment row to place the next queen
47              if(Solve(board, row+1) == true)
48                  return true;
49              //attempt to place queen at row+1 failed,-
50              //backtrack to row and remove queen
51              board[row][j]=0;
52              printf("Backtrack: \n");
53              printSolution(board);
54          }
55      }
56      return false;
57  }
```

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

```
59  int isSafe(int board[N][N], int row, int col)
60  {
61      int i, j;
62      for(i=0;i<row;i++)
63      {
64          for(j=0;j<N;j++)
65          {
66              //check whether there's a queen at the same column or the 2 diagonals
67              if(((j==col) || (i-j == row-col) || (i+j == row + col)) && (board[i][j]==1))
68                  return false;
69          }
70      }
71      return true;
72  }
73
74
75  void printSolution(int board[N][N])
76  {
77      int i,j;
78      for(i=0;i<N;i++)
79      {
80          for(j=0;j<N;j++)
81              printf(" %d ", board[i][j]);
82          printf("\n");
83      }
84  }
```

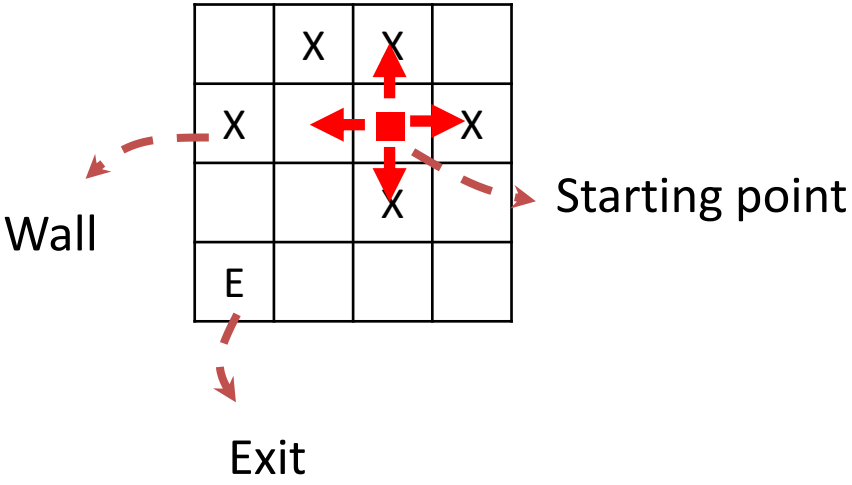


```
int Solve(int board[N][N], int row)
{
    //base case
    if(row>=N)
        return true;

    //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;
}
```

Maze Solver

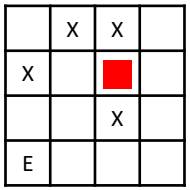
	X	X	
X	*	*	X
*	*	X	
E			

1 of solutions

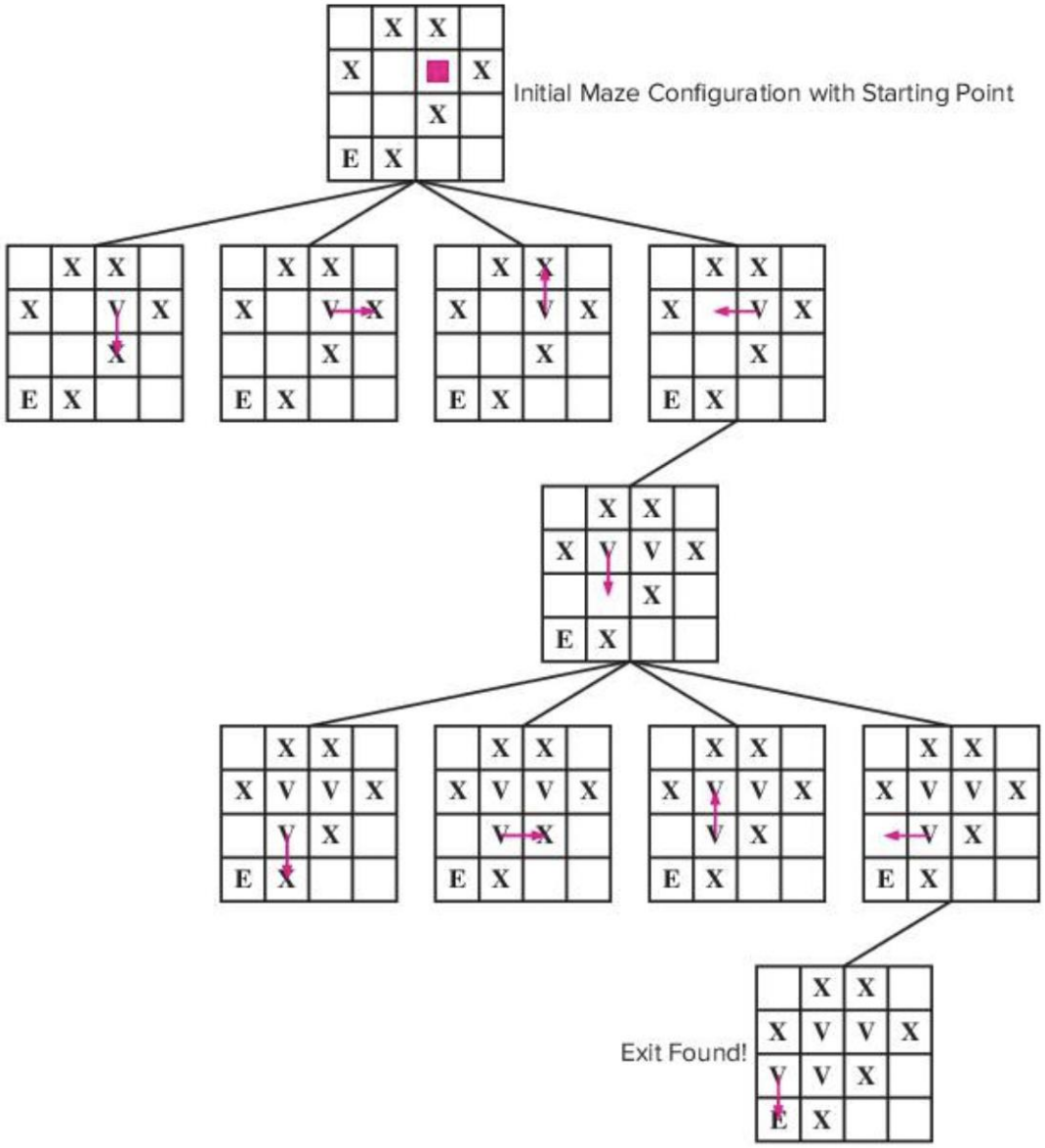


- GOAL
  - *if solve(..),* find a path from the starting point to the exit → Return 1 and mark '*'
  - One solution is enough (may not be the shortest path)

- IDEA
  - Try 1 of 4 moves (U/D/L/R)
  - Solve it from there (recursively!)
  - Mark 'V', if visited (avoid circling)



circle!


Initial Maze Configuration with Starting Point
Exit Found!

```c
#include <stdio.h>
/*
// maze 1
#define MAZE_H 4
#define MAZE_W 4
*/


// maze 2
#define MAZE_H 4
#define MAZE_W 6

void print_maze(char maze[][MAZE_W], int height, int width);
int solve(char maze[][MAZE_W], int xpos, int ypos);

int main(){
/*
        char maze[MAZE_H][MAZE_W] = {{' ', 'X', 'X', ' '},
                                     {'X', ' ', ' ', 'X'},
                                     {' ', ' ', 'X', ' '},
                                     {'E', 'X', ' ', ' '}};
*/

    char maze[MAZE_H][MAZE_W] = {{' ', 'X', 'X', 'X', ' ', 'X'},
                                 {' ', ' ', ' ', ' ', ' ', 'X'},
                                 {' ', 'X', 'X', 'X', ' ', ' '},
                                 {'E', 'X', ' ', 'X', ' ', ' '}};

    /*
    char maze[MAZE_H][MAZE_W] = {{' ', 'X', 'X', 'X', ' ', 'X'},
                                 {'X', ' ', ' ', ' ', ' ', 'X'},
                                 {' ', 'X', 'X', 'X', ' ', ' '},
                                 {'E', 'X', ' ', ' ', ' ', ' '}};
                                 */
    int x_start = 1, y_start = 2;
    printf("Starting point: (%d, %d)\n", x_start, y_start);
    print_maze(maze, MAZE_H, MAZE_W);
    int res = solve(maze, x_start, y_start);
```

```c
void print_maze(char maze[][MAZE_W], int height, int width){
    int i,j;
    printf(" ");
    for(j=0;j<width;j++)
        printf("%d", j);
    printf("\n");
    for(i=0;i<height;i++){
        printf("%d", i);
        for(j=0;j<width;j++){
            printf("%c", maze[i][j]);
        }
        printf("\n");
    }
}
```

```c
if(res)
    printf("\n\nFound a solution.\n");
else
    printf("\n\nNo solution exists.\n");
print_maze(maze, MAZE_H, MAZE_W);
                                    }
```

Recursive Function:

int solve(…….)

```
Starting point: (1, 2)
 012345
0 XXX X
1     X
2 XXX
3EX X


Found a solution.
 012345
0 XXXVX
1***VVX
2*XXXVV
3EX XVV
```

```
// if no path exists, return 0
// if a path exists, return 1
int solve(char maze[][MAZE_W], int xpos, int ypos){
    // current xy position is one of the following cases
    // 1. (B) out of bound
    if(xpos < 0 || xpos >= MAZE_H || ypos < 0 || ypos >= MAZE_W)
        return 0;
    // 2. (B) found exist!
    if(maze[xpos][ypos] == 'E')
        return 1;
    // 3. (B) hit a wall or visited path
    if(maze[xpos][ypos] == 'V' || maze[xpos][ypos] == 'X')
        return 0;

    // 4. (R) a new empty space (let's mark as visted)
    maze[xpos][ypos] = 'V';
    // move down
    if( solve(maze, xpos + 1, ypos) == 1 ){
        maze[xpos][ypos] = '*';
        return 1;
    }
    // move right
    if( solve(maze, xpos, ypos + 1) == 1 ){
        maze[xpos][ypos] = '*';
        return 1;
    }
    // move up
    if( solve(maze, xpos - 1, ypos) == 1 ){
        maze[xpos][ypos] = '*';
        return 1;
    }
    // move left
    if( solve(maze, xpos, ypos - 1) == 1 ){
        maze[xpos][ypos] = '*';
        return 1;
    }

    return 0;
}
```