

ECE 220 Computer Systems & Programming

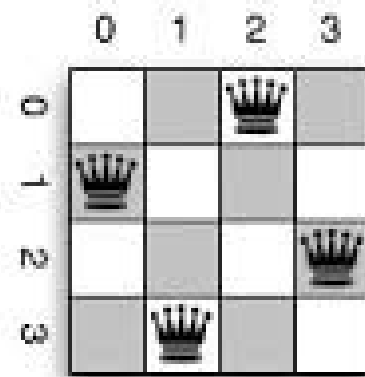
Lecture 15 – Recursion with Backtracking

March 7, 2019

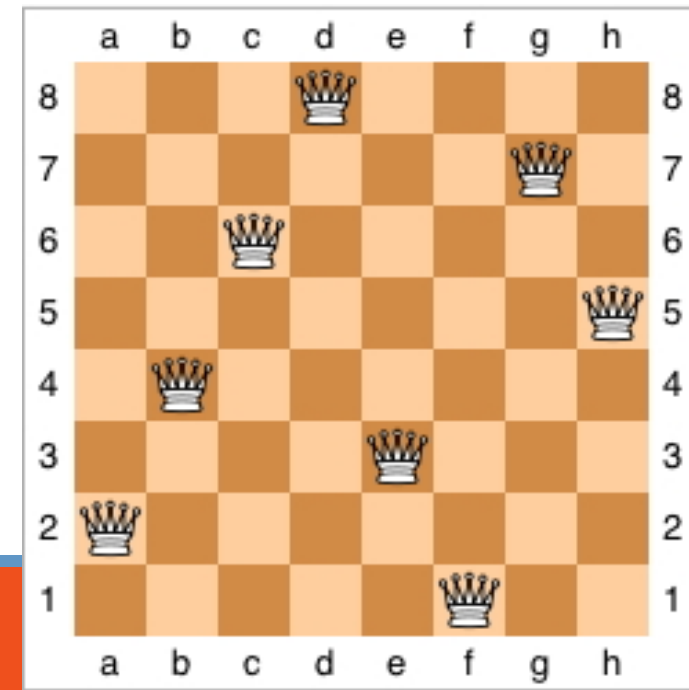


N queens problem using recursive Backtracking

- Place N queens on an NxN chessboard so that none of the queens are **under attack**;
- Brute force: total number of possible placements:
- $\sim N^2$ Choose N $\sim 4.4 \text{ B}$ (N=8)



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




N Queens using Backtracking


1. Start in leftmost column
2. If all queens are placed return true
3. For each row in the current column:
 - a. If queen can be safely placed in this row, then mark [row, col] as part of the solution and recursively check if this board leads to solution.
 - b. If it leads to solution, return true
 - c. Else, unmark[row, col] (backtrack) and go to next row (step 3)
4. If no rows work, then return false (triggering backtrack)

	0	1	2	3
Row 0				

	0	1	2	3
0				
1				
2				
3				


QueenPosition[]


	0	1	2	3
Row 0				

	0	1	2	3
0				
1				
2				
3				

QueenPosition[]

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

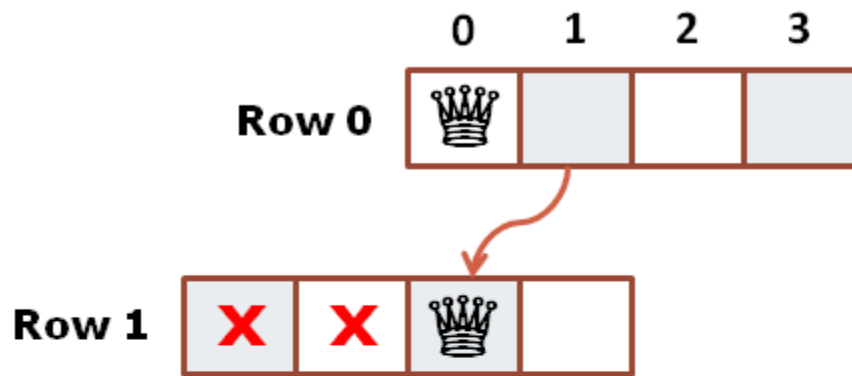
	0	1	2	3
Row 0				

	0	1	2	3
0				
1				
2				
3				

QueenPosition[]

Row 0 (0,0)

Add 0th Queen's position to position array



	0	1	2	3
0	♔			
1	X	X	♔	
2				
3				

QueenPosition[]

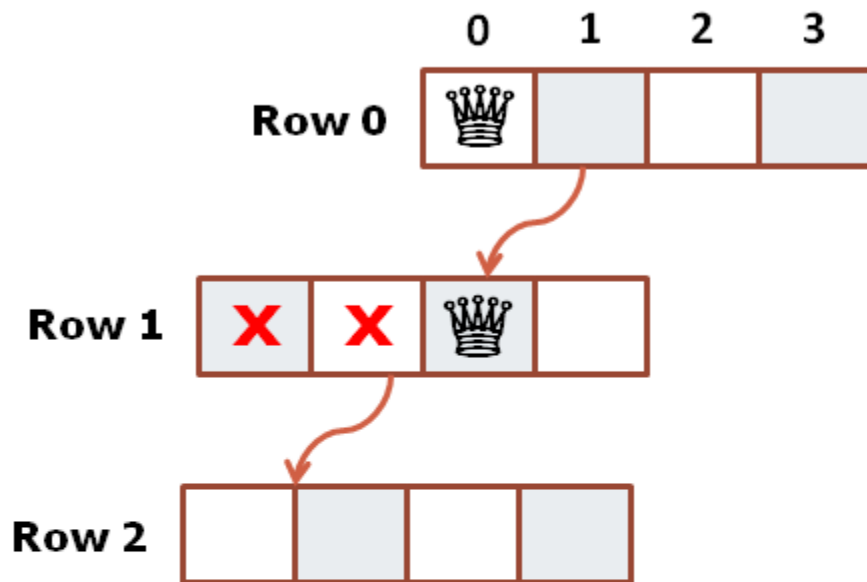
Row 0 (0,0)

Row 1 (1,2)

Go to the next level of recursion.

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





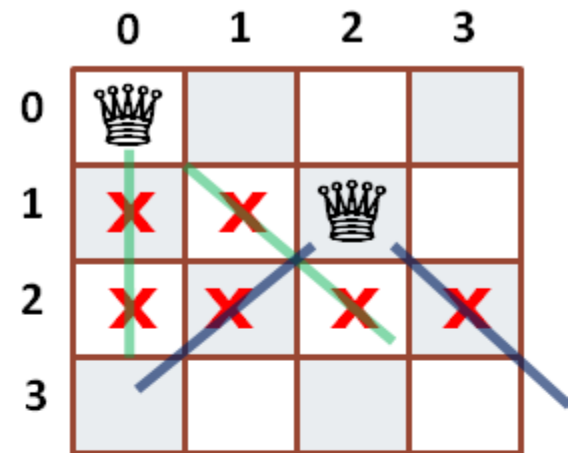
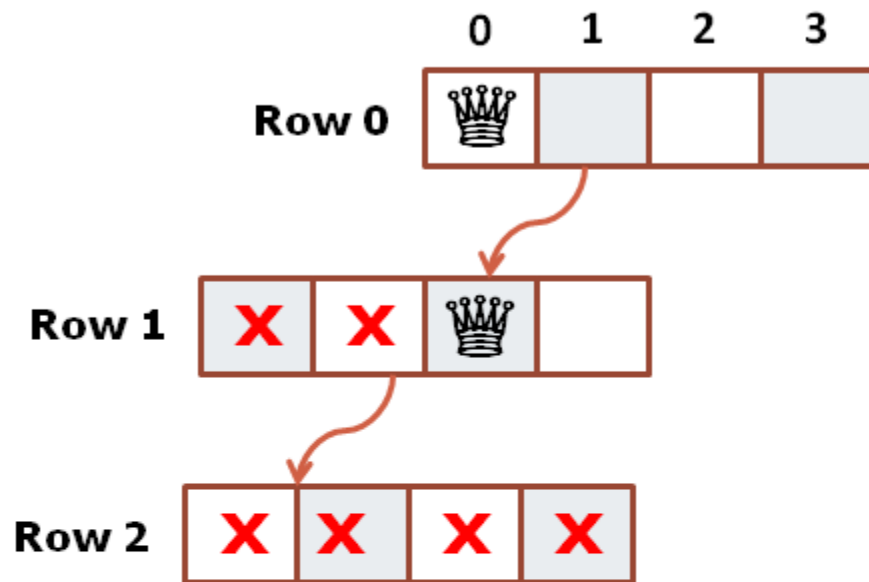
	0	1	2	3
0	♔			
1	✗	✗	♔	
2				
3				

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.

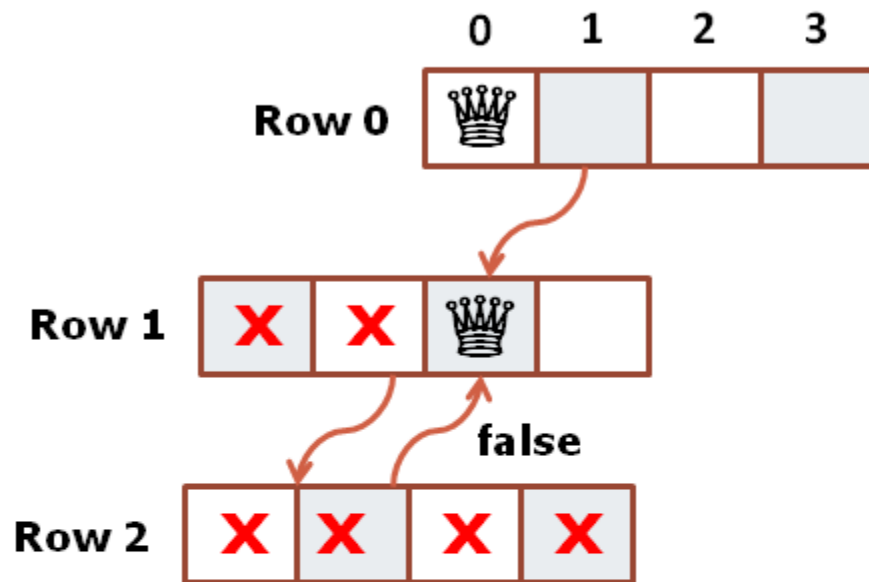


QueenPosition[]

Row 0 (0,0)

Row 1 (1,2)

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



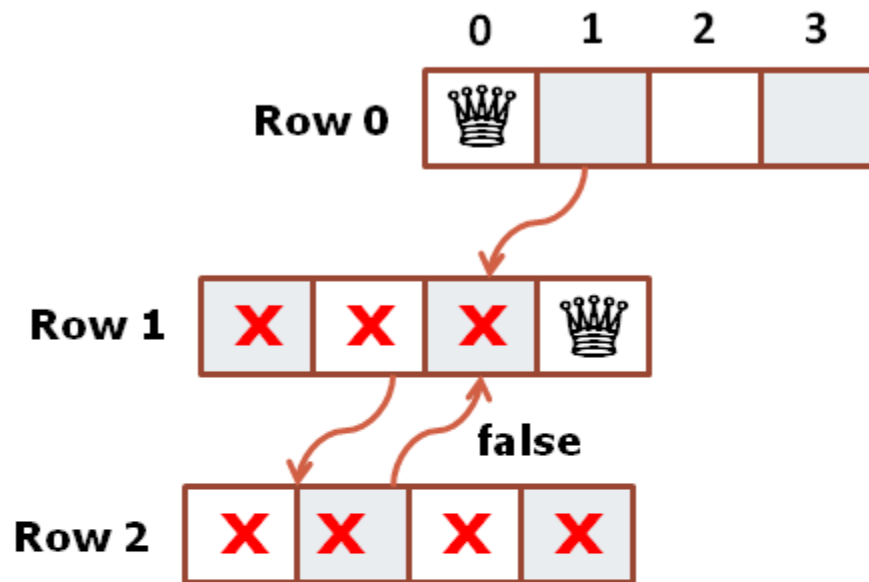
	0	1	2	3
0	♔			
1	✗	✗	♔	
2	✗	✗	✗	✗
3				

QueenPosition[]

Row 0 (0,0)

Row 1 (1,2)

So function will return false to the calling function.



	0	1	2	3
0	Queen			
1	X	X	X	Queen
2				
3				

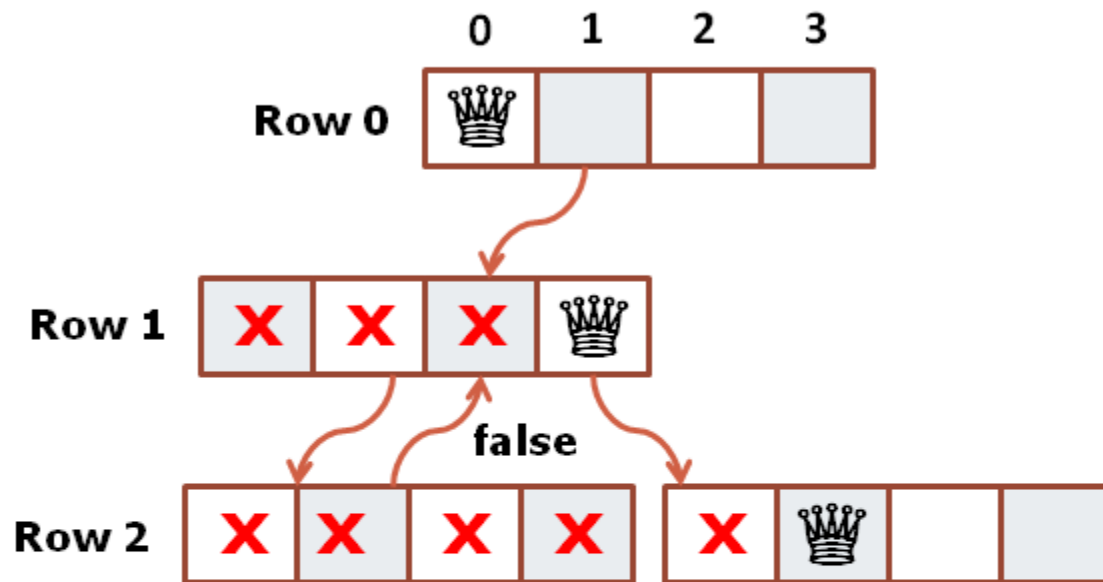
QueenPosition[]

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.



	0	1	2	3
0	Queen			
1	X	X	X	Queen
2	X	Queen		
3				

QueenPosition[]

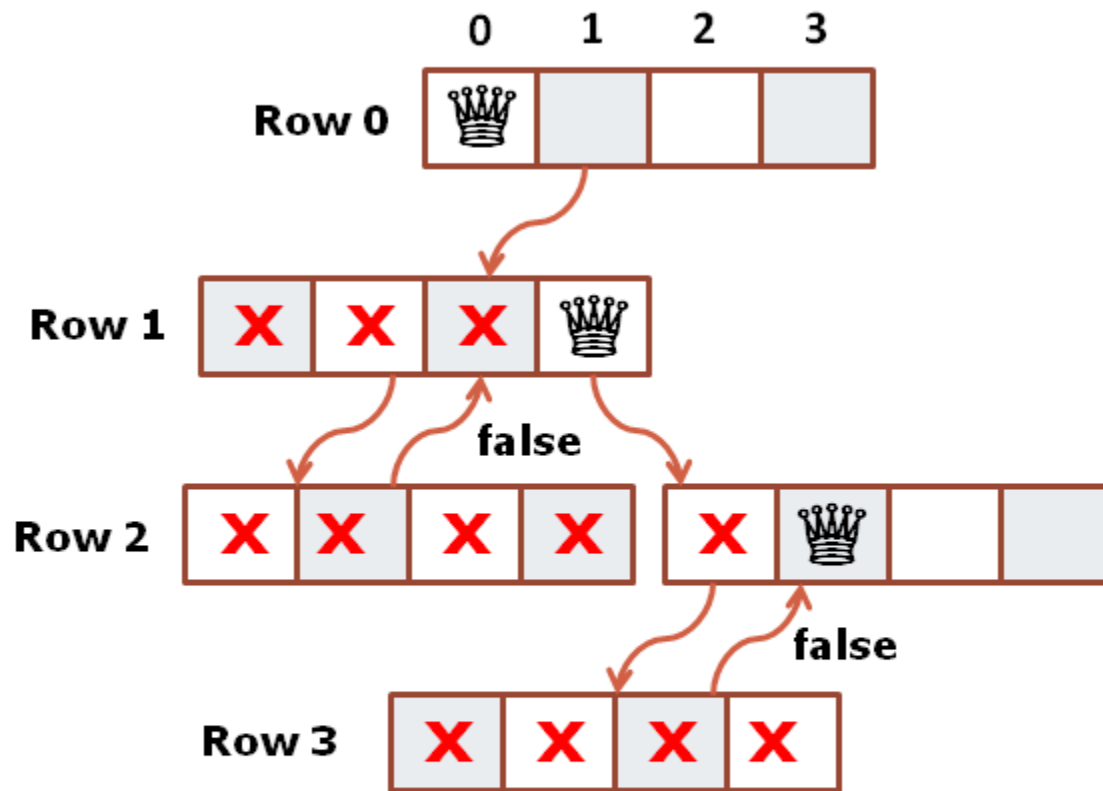
Row 0 (0,0)

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.



	0	1	2	3
0				
1				
2				
3				

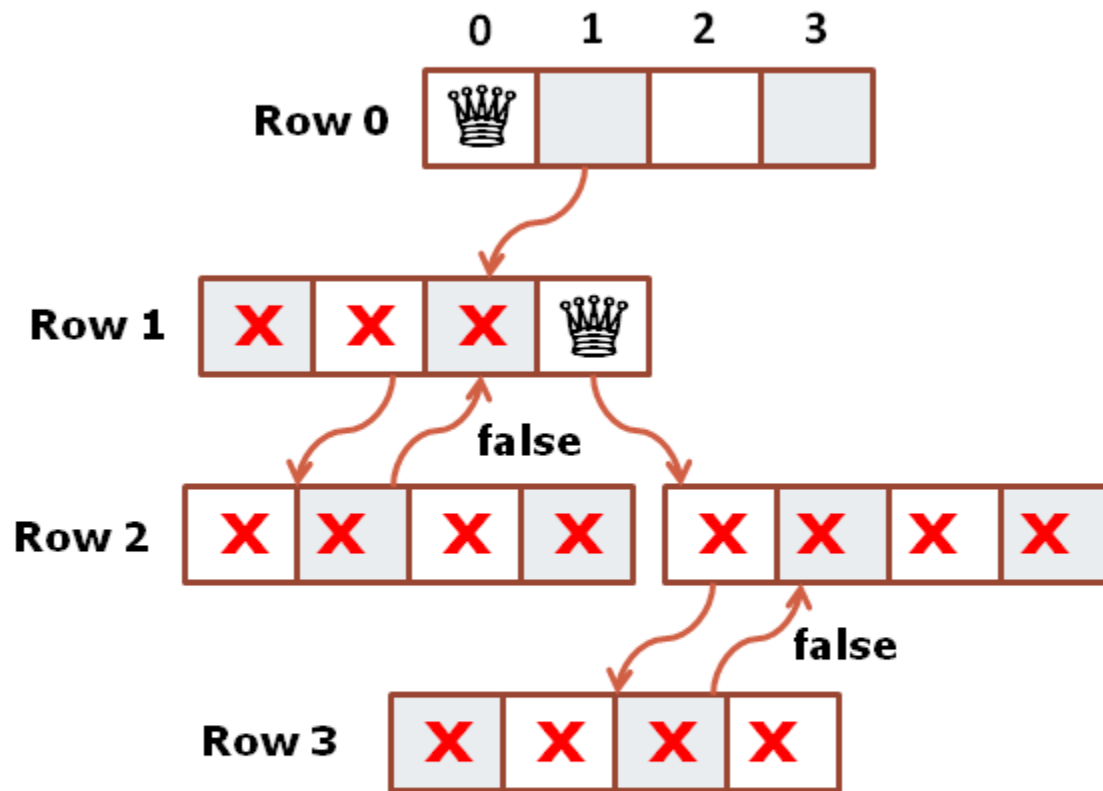
QueenPosition[]

Row 0 (0,0)

Row 1 (1,3)

Row 1 (2,1)

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



	0	1	2	3
0				
1	X	X	X	
2	X	X	X	X
3				

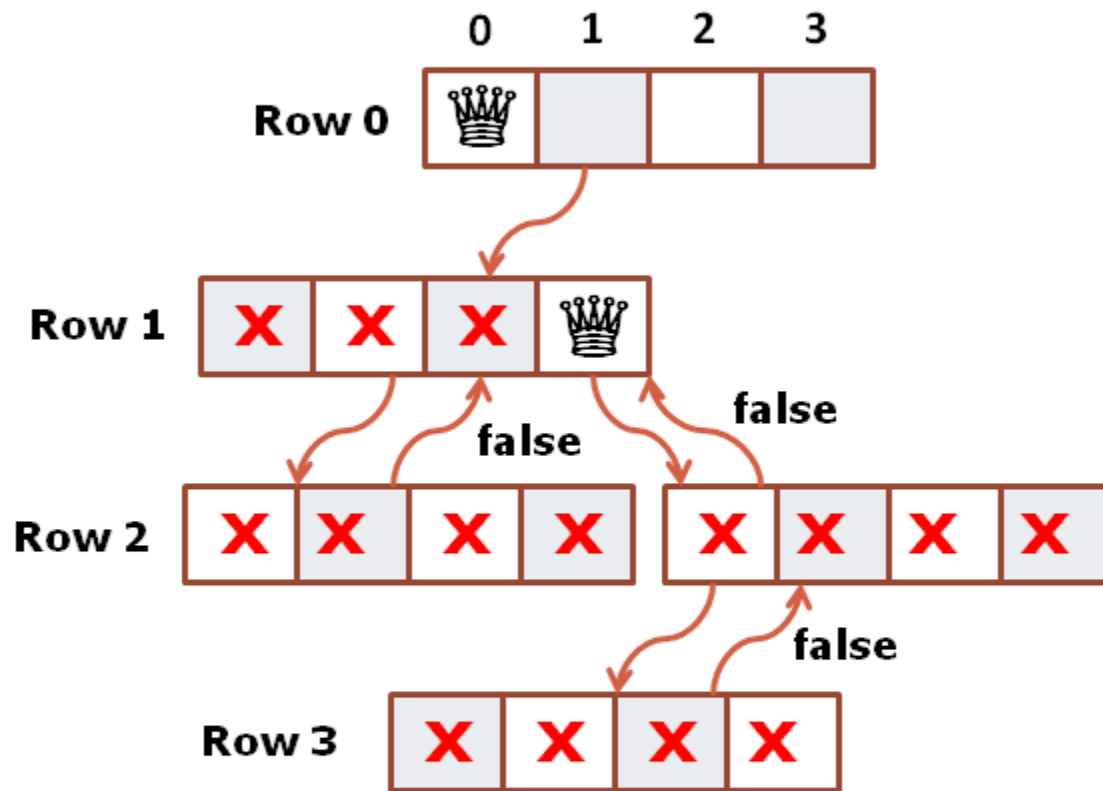
QueenPosition[]

Row 0 (0,0)

Row 1 (1,3)

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

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



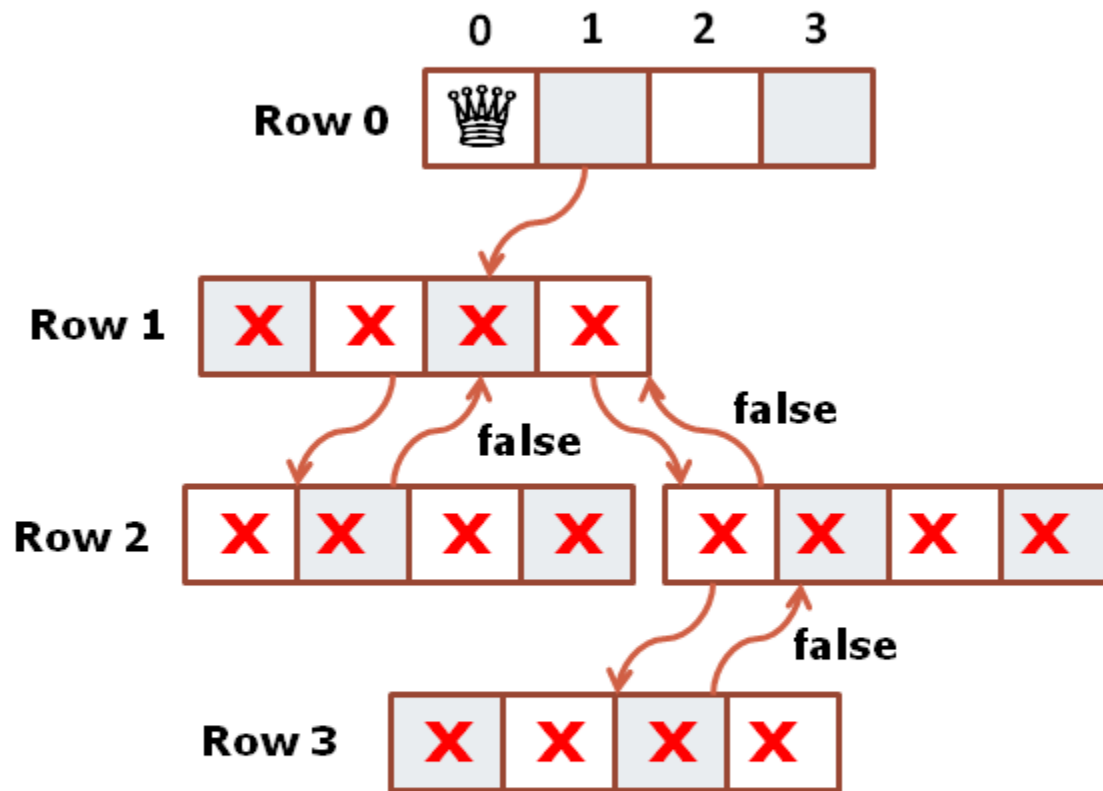
	0	1	2	3
0				
1	X	X	X	
2				
3				

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.



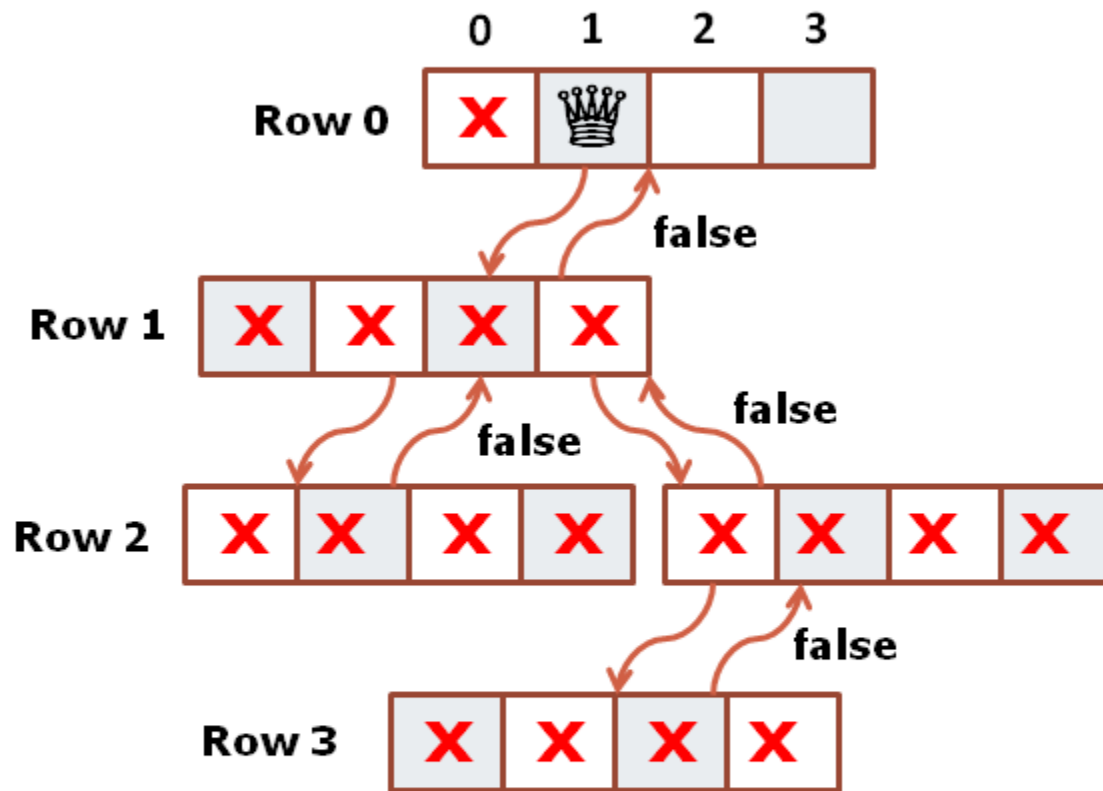
	0	1	2	3
0				
1	X	X	X	X
2				
3				

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 return false to
Its calling function.



	0	1	2	3
0	X	♔		
1				
2				
3				


QueenPosition[]

~~Row 0~~ (0,0)

Row 0 (0,1)

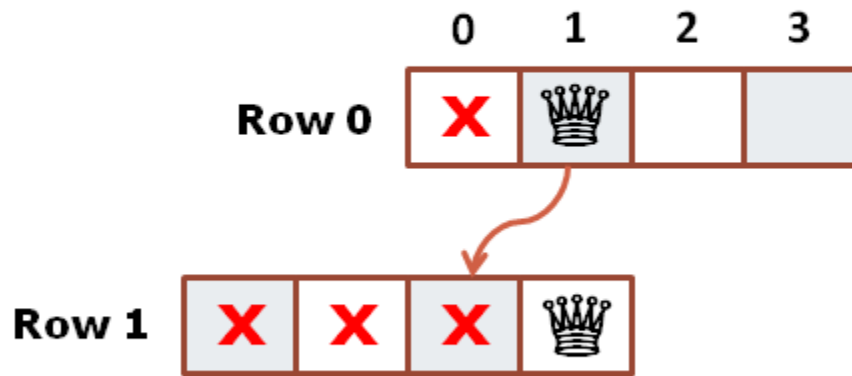
Queen at the **1st** row tries to find next safe cell.
Let us remove these failed recursion calls from the screen.

	0	1	2	3
Row 0	X			

	0	1	2	3
0	X			
1				
2				
3				

QueenPosition[]

Row 0 (0,1)

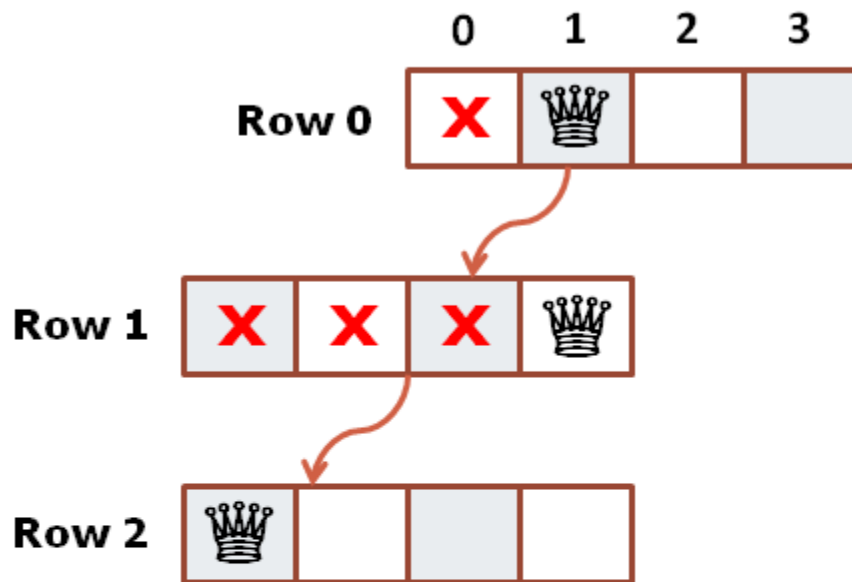


	0	1	2	3
0	X	Queen		
1	X	X	X	Queen
2				
3				

QueenPosition[]

Row 0 (0,1)

Row 1 (1,3)



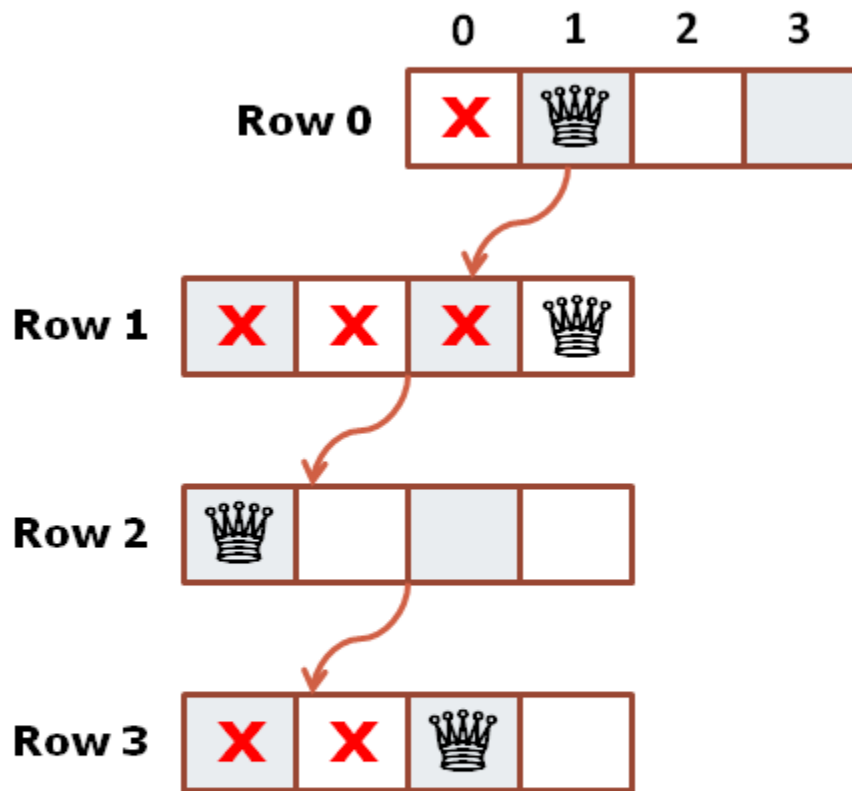
	0	1	2	3
0	X	Queen		
1	X	X	X	Queen
2	Queen			
3				

QueenPosition[]

Row 0 (0,1)

Row 1 (1,3)

Row 2 (2,0)



	0	1	2	3
0	X	Queen		
1	X	X	X	Queen
2	Queen			
3	X	X	Queen	

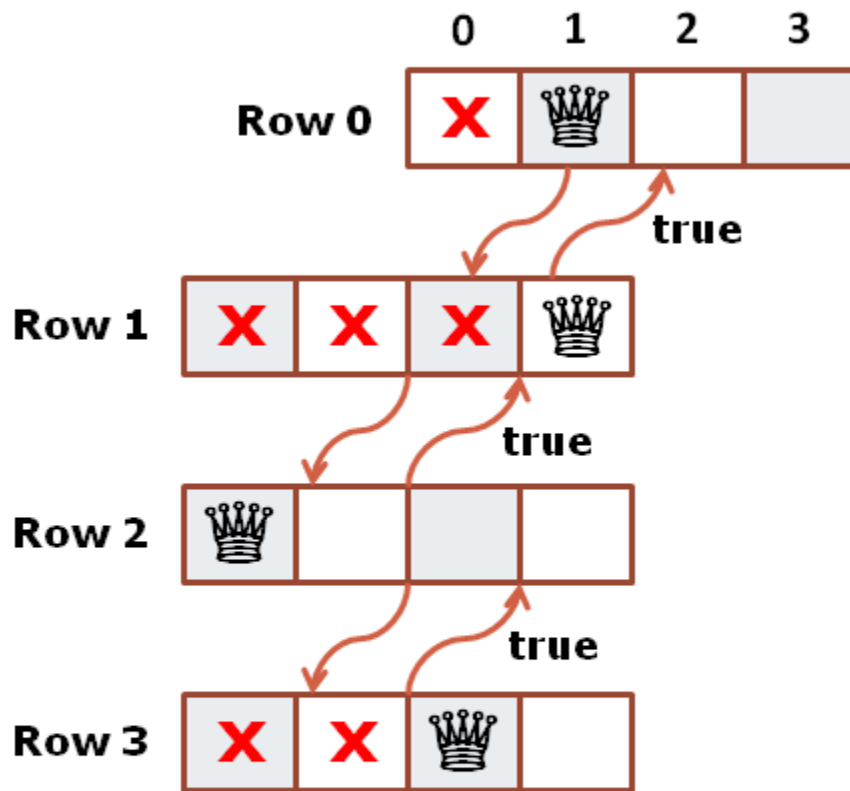
QueenPosition[]

Row 0 (0,1)

Row 1 (1,3)

Row 2 (2,0)

Row 3 (3,2)



	0	1	2	3
0	X	Queen		
1	X	X	X	Queen
2	Queen			
3	X	X	Queen	

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(board, 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 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

Warm up

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

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

- 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

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

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

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

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

```
1  #include <stdio.h>
2
3  //Solve 4x4 n Queen problem using recursion with backtracking
4
5  #define N 4
6  #define true 1
7  #define false 0
8
9  void printSolution(int board[N][N]);
10 int Solve(int board[N][N], int col);
11 int isSafe(int board[N][N], int row, int col);
12
13 int main()
14 {
15     int board[N][N] = {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
16
17     //game started at row 0
18     if(Solve(board,0) == false)
19     {
20         printf("Solution does not exist.\n");
21         return 1;
22     }
23
24     printf("Solution: \n");
25     printSolution(board);
26     return 0;
27 }
```

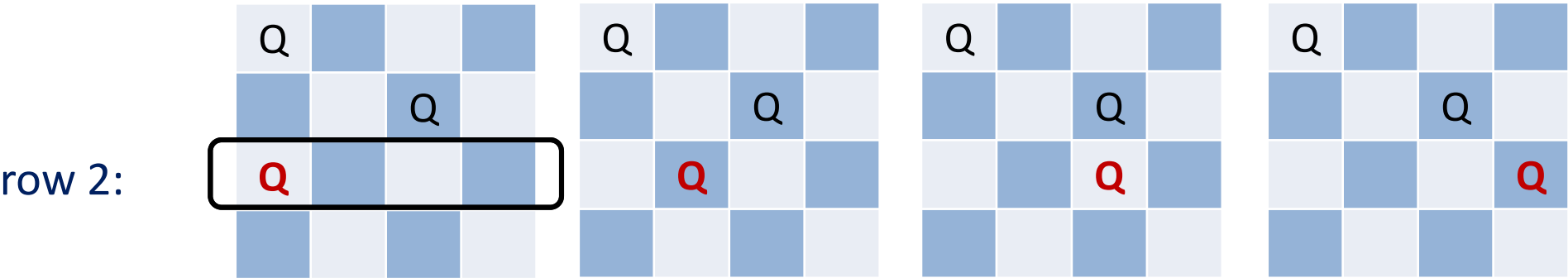
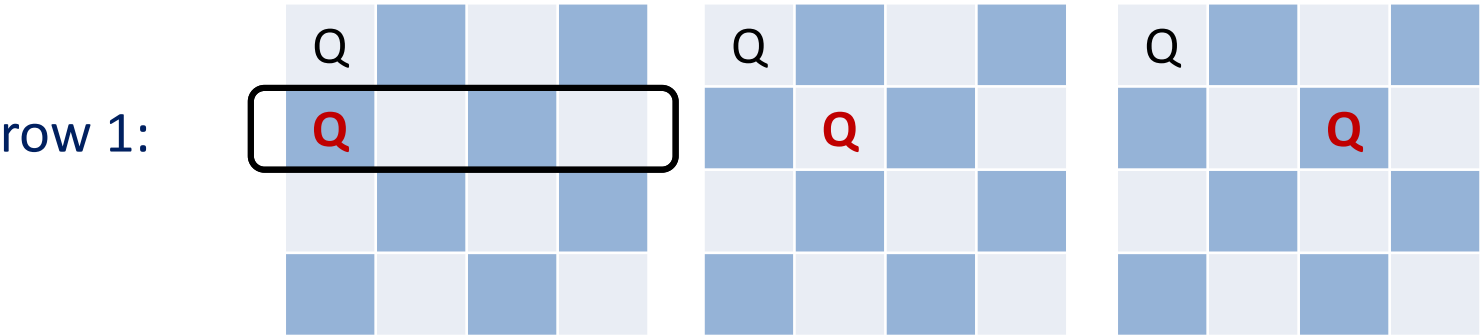
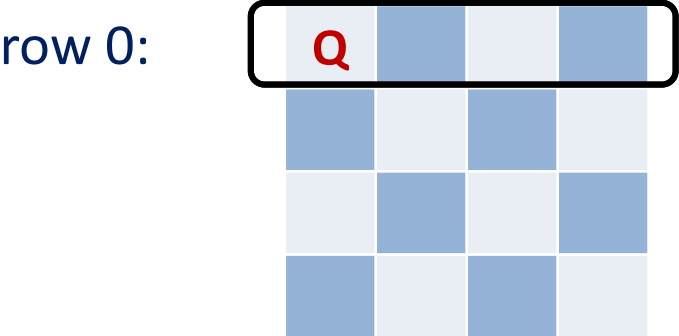
N-Queen (4x4) Backtracking – CODE (Solve function)

```
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 }
```

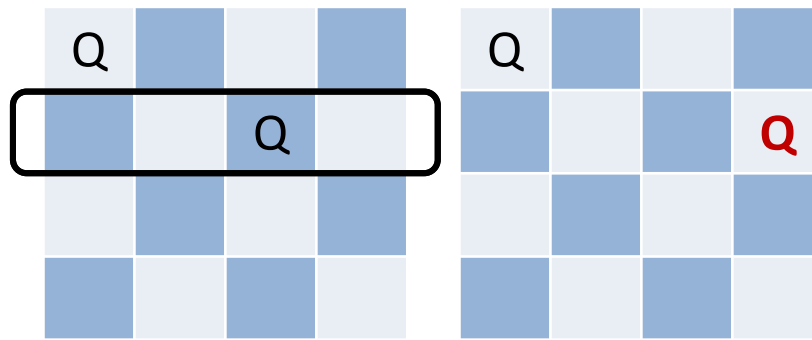
N-Queen (4x4) Backtracking – CODE (isSafe & PrintSolution functions)

```
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 }
```

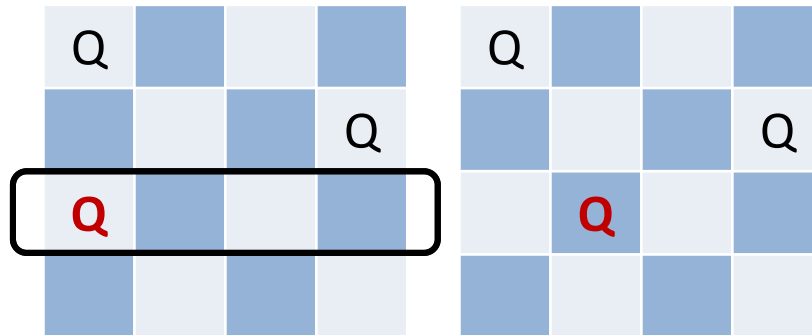
Example: 4x4 n-Queen



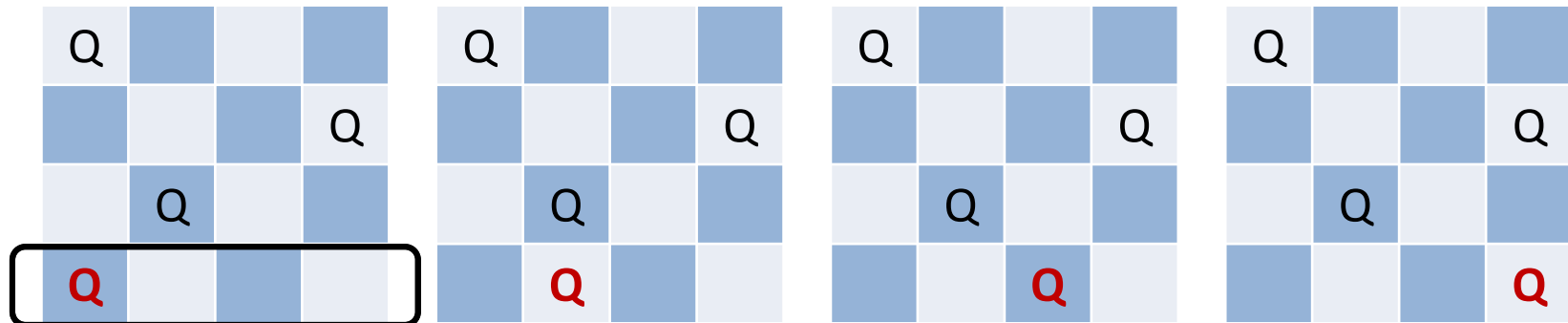
*Backtrack
to row 1 and
make a new
choice:*



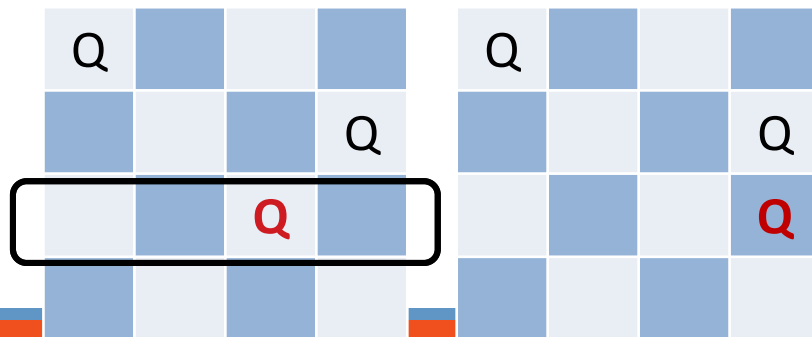
row 2:



row 3:

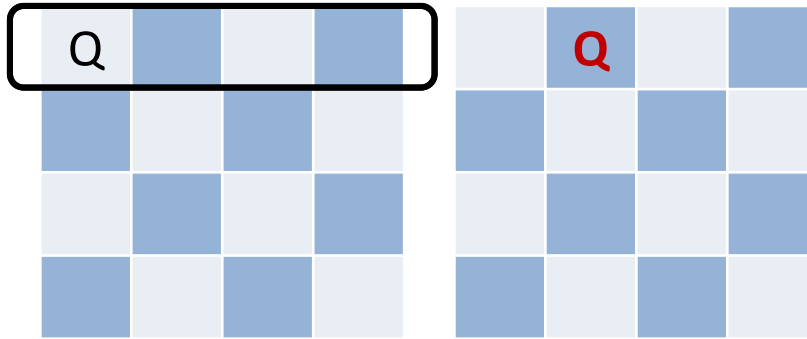


*Backtrack
to row 2 and
make a new
choice:*

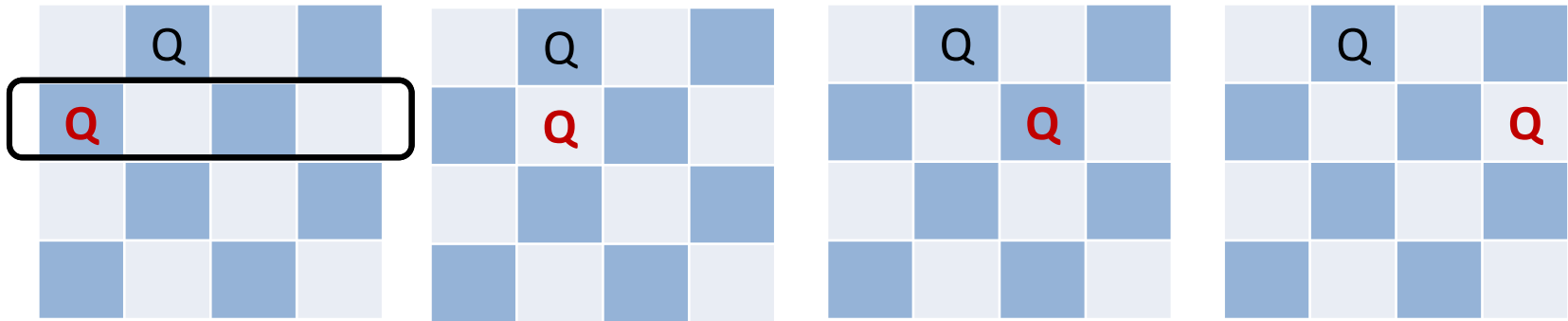


****Backtrack to row 1, but no columns left**

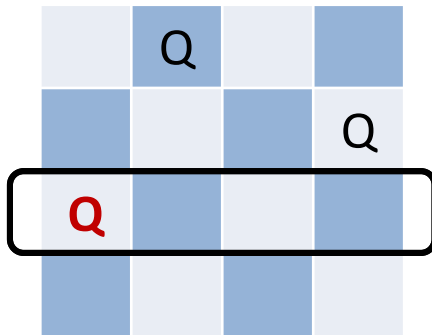
*Backtrack
to row 0 and
make a new
choice:*



row 1:



row 2:



row 3:

