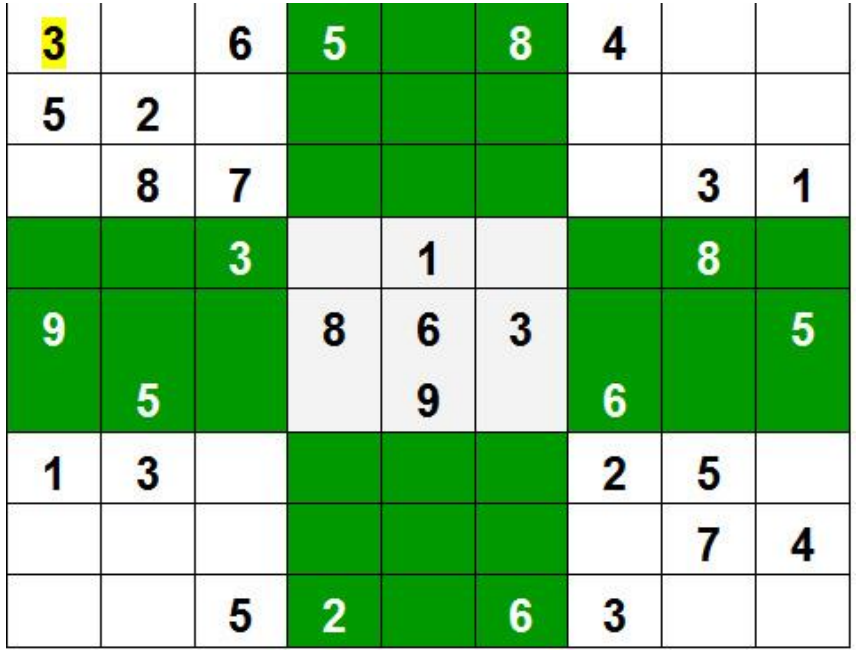
Backtracking

SWE2016-44

Introduction

Backtracking: 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 at any point of time.



Sudoku

Rat in a Maze

Problem Statement

A Maze is given as NxN binary matrix of blocks.

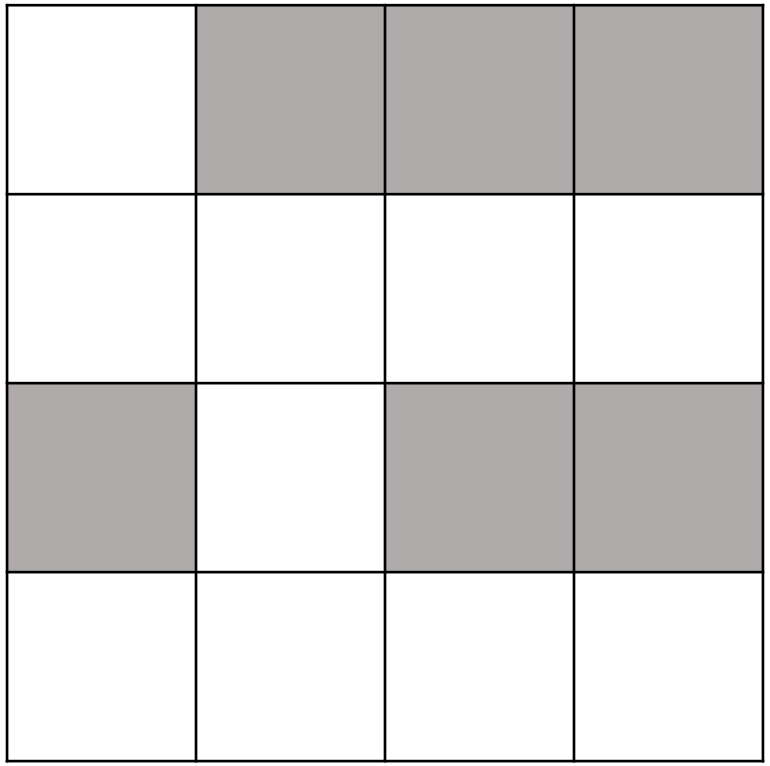
The source or start block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1].

A rat starts from source and has to reach destination.

The rat can move only in two directions: right and down in the matrix.

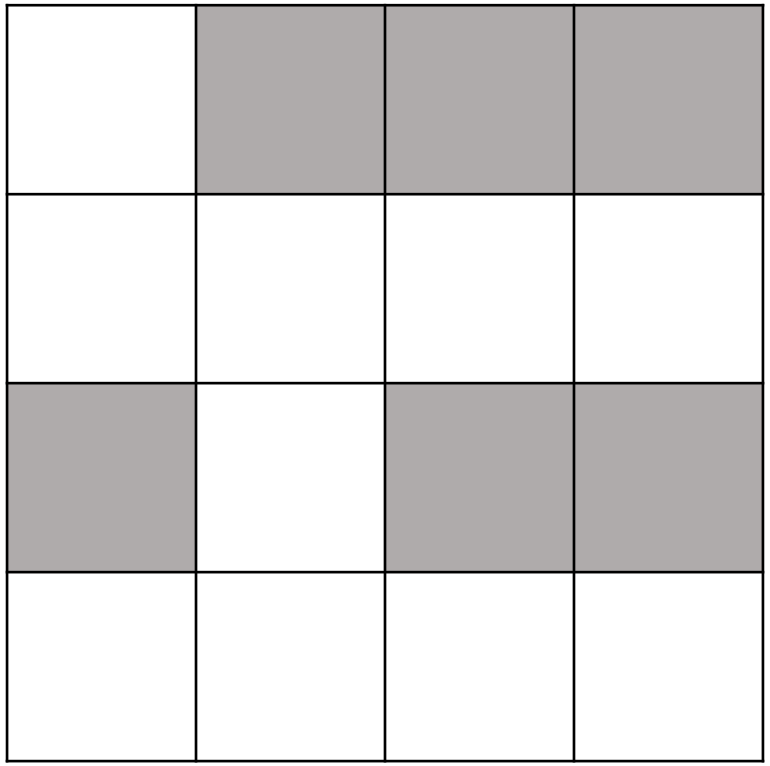
Solution

Let us solve our problem for the following maze



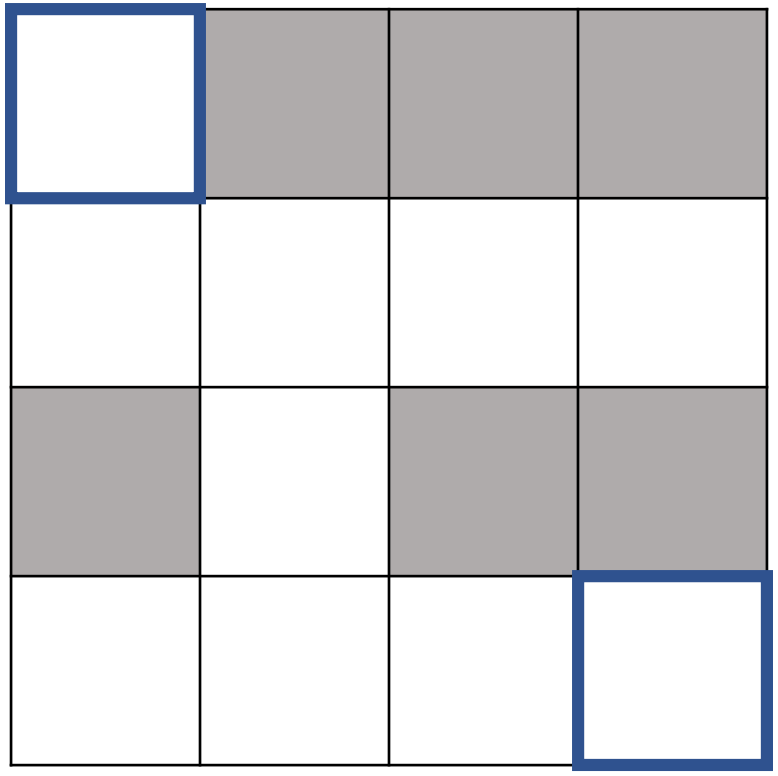
Solution

The rat can move on the white block only.



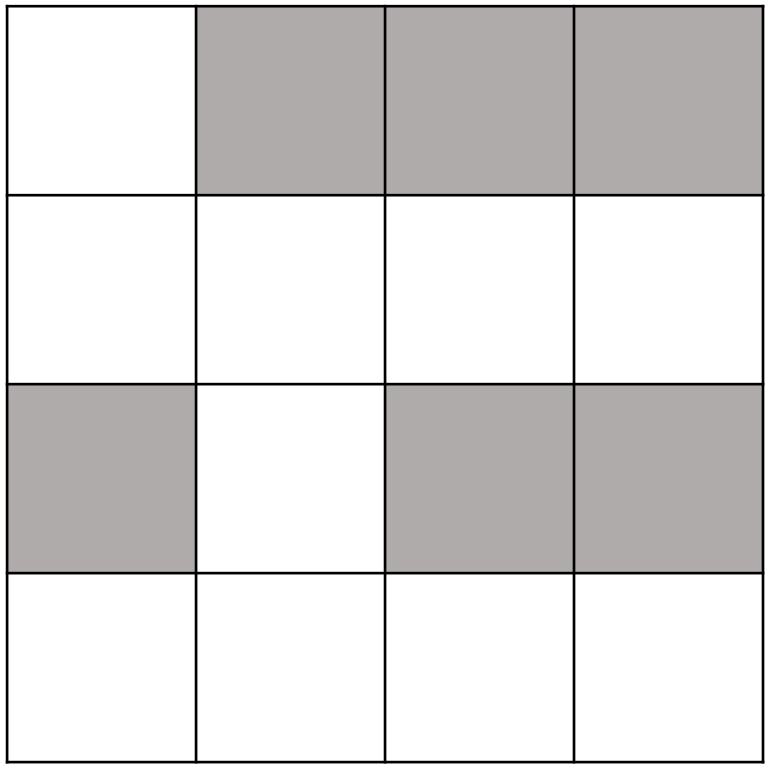
Solution

It starts from the top-left and has to go to the bottom right



Solution

In the maze matrix, 0 means the block is dead end and 1 means the block can be used in the path from source to destination.



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Solution

Let’s define a function solveMazeUtil(maze, x, y, sol)

- **maze is the matrix representing the blocked and unblocked bocks in the maze**
- **x, y is the x and y coordinates of the position of the Rat**
- **sol holds the path traversed by the rat so far, also in the form of a matrix**

Solution

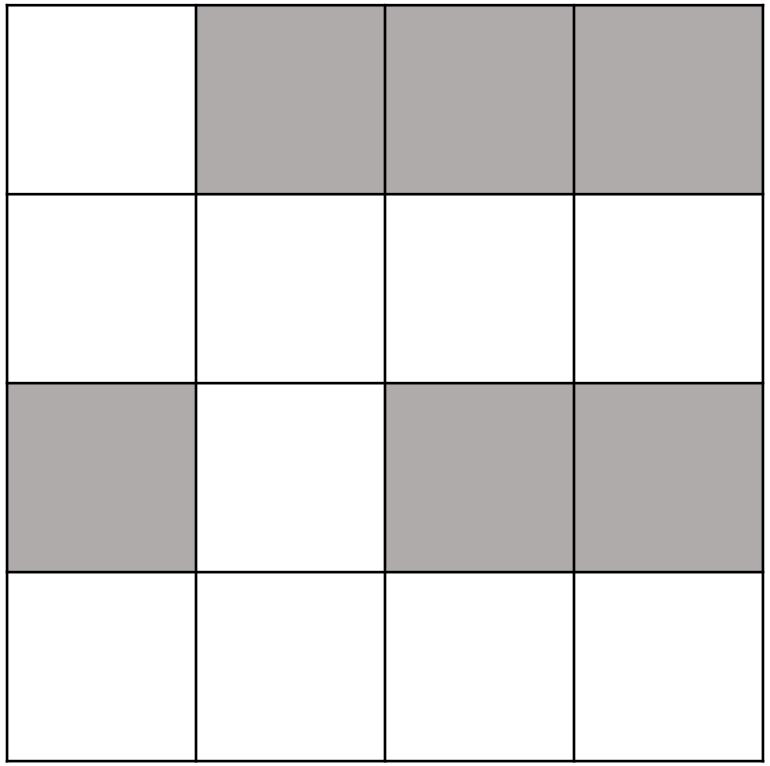
The function solveMazeUtil(maze, x, y, sol) will return false if there's no path possible to reach the destination and true if any path is found.

The function will modify the sol matrix with the required path from current position (x, y) to destination (N-1, N-1).

Solution

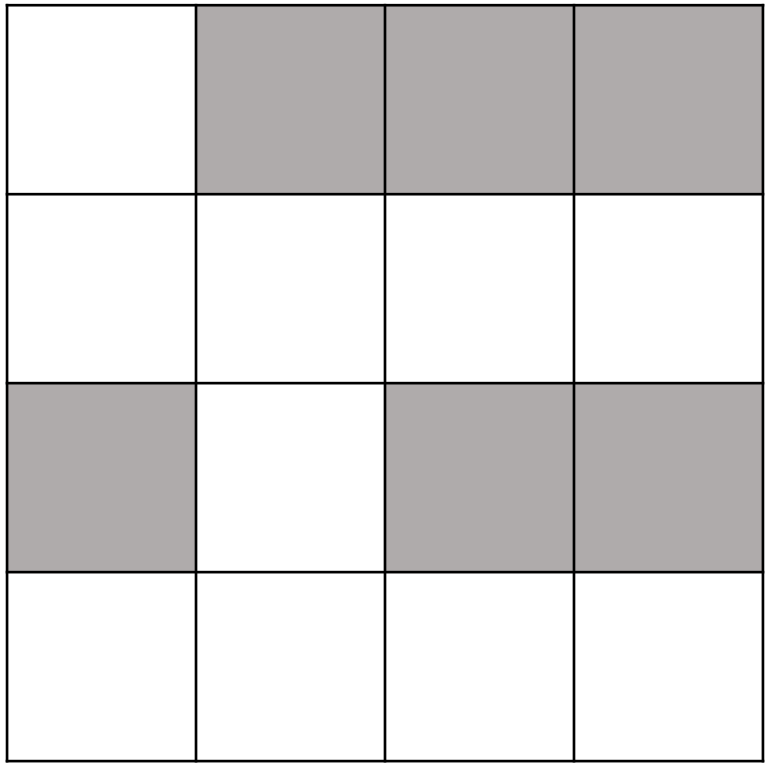
We will initiate the function at the start position and it will return the required path in the sol matrix.

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])



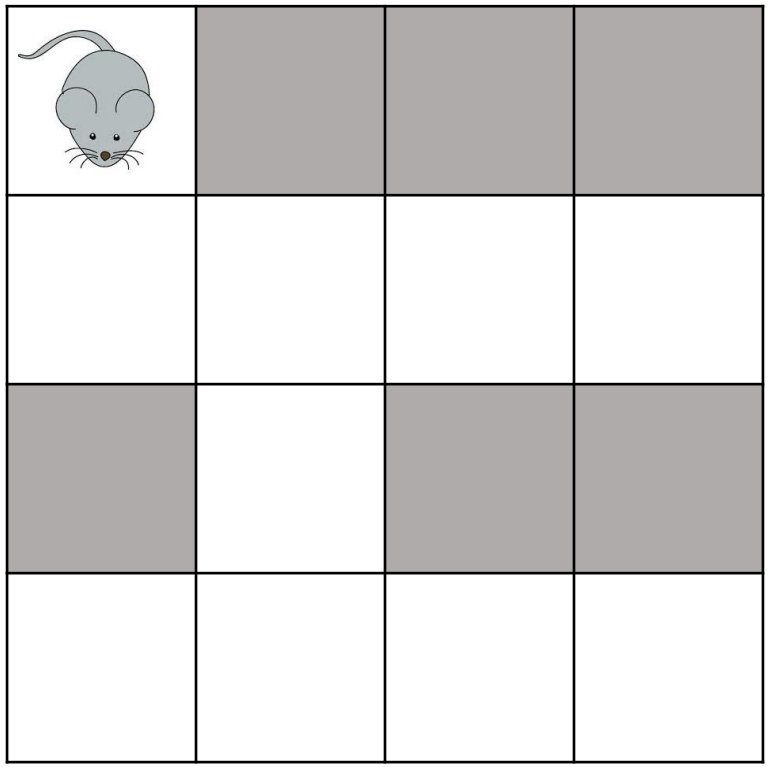
solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

Now, let's define the function so that it translates the maze using backtracking and finds the required path.



solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

For the current situation, let's place the Rat at the start position.



sol Matrix

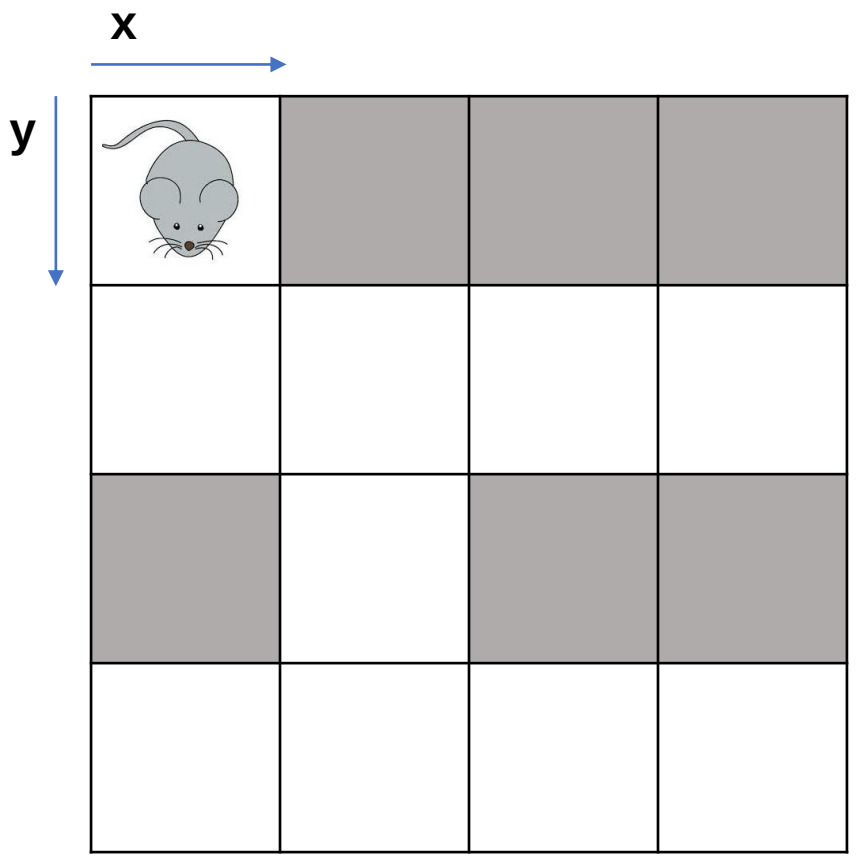
$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

check

if going in x direction i.e., solveMazeUtil(maze[4][4], 1, 0, sol[4][4]),

else go in y direction i.e., solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

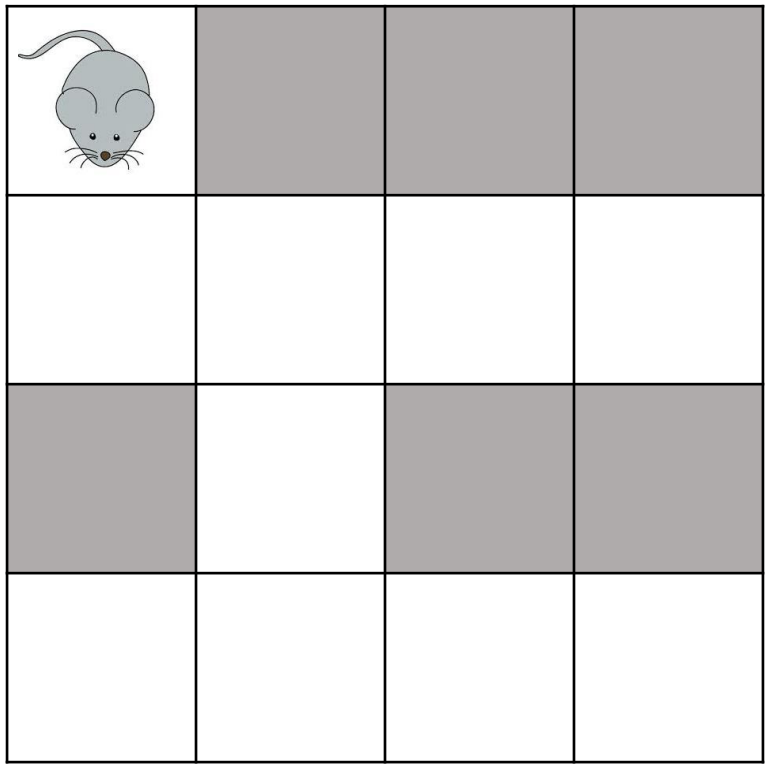


sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4]) solveMazeUtil(maze[4][4], 0, 1, sol[4][4])



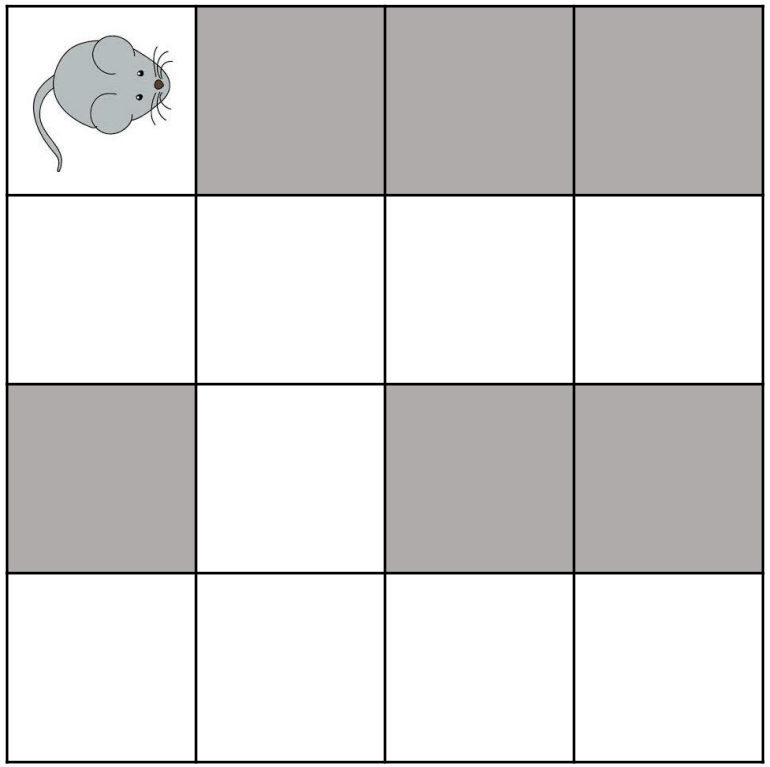
sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4]) solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

cannot go in x direction



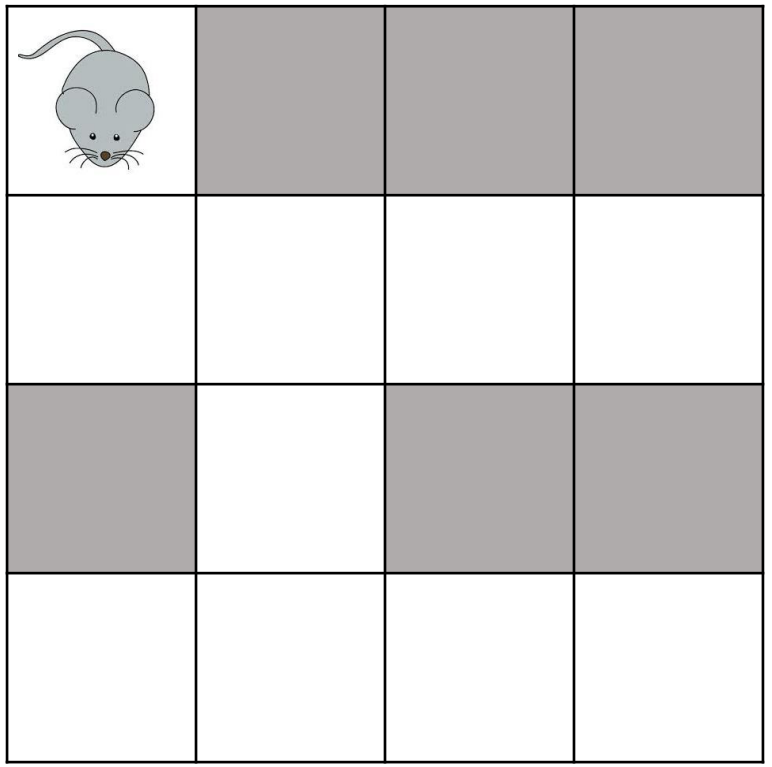
sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])



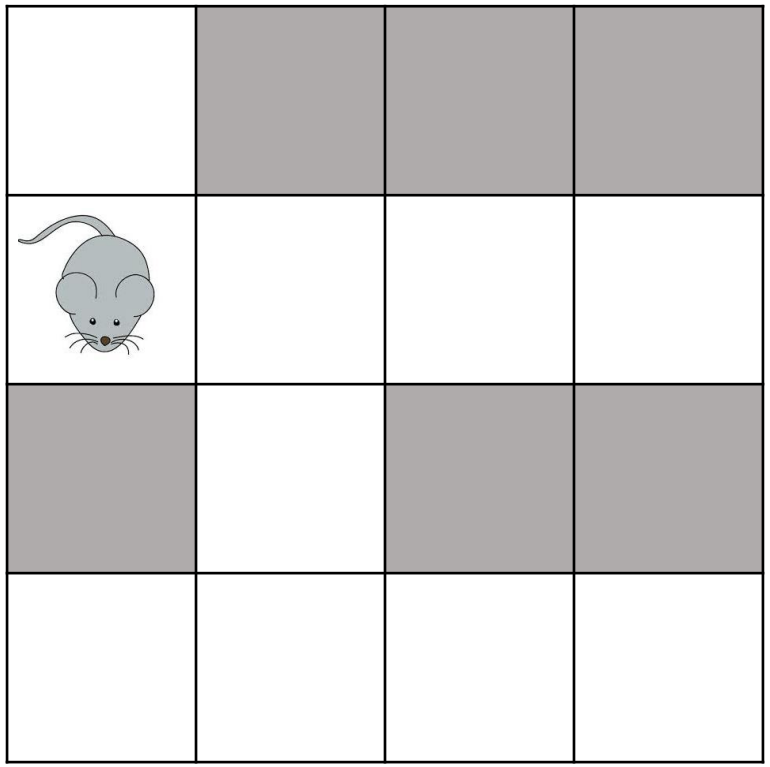
sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4]) **solveMazeUtil(maze[4][4], 0, 1, sol[4][4])**

solveMazeUtil(maze[4][4], 1, 1, sol[4][4]) **solveMazeUtil(maze[4][4], 0, 2, sol[4][4])**



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

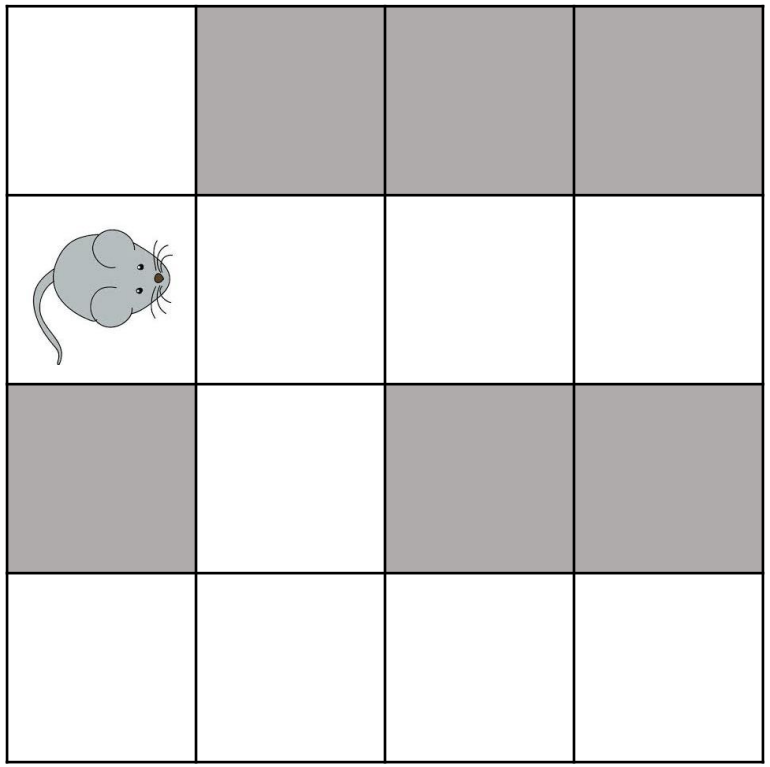
solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

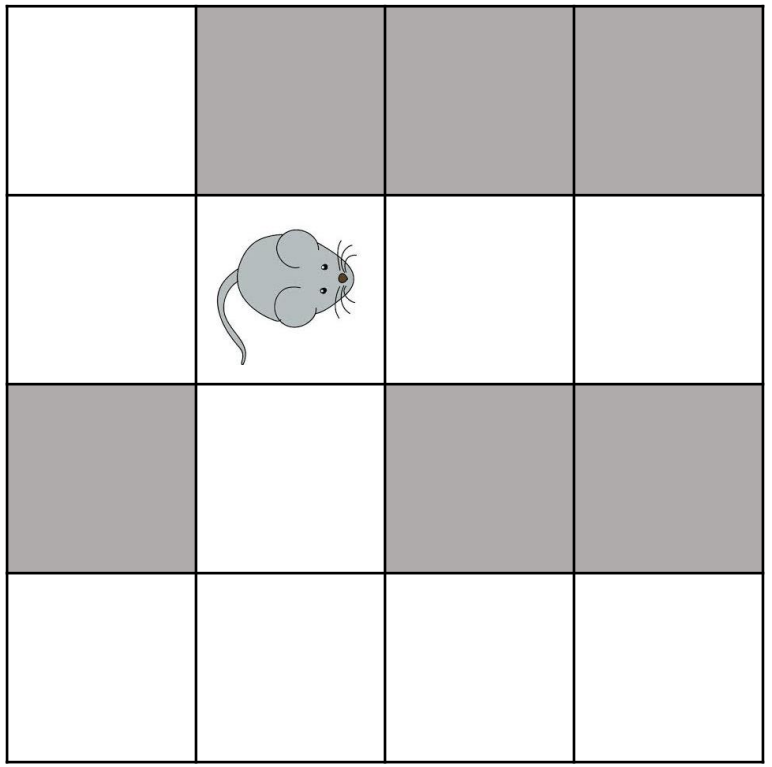
solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

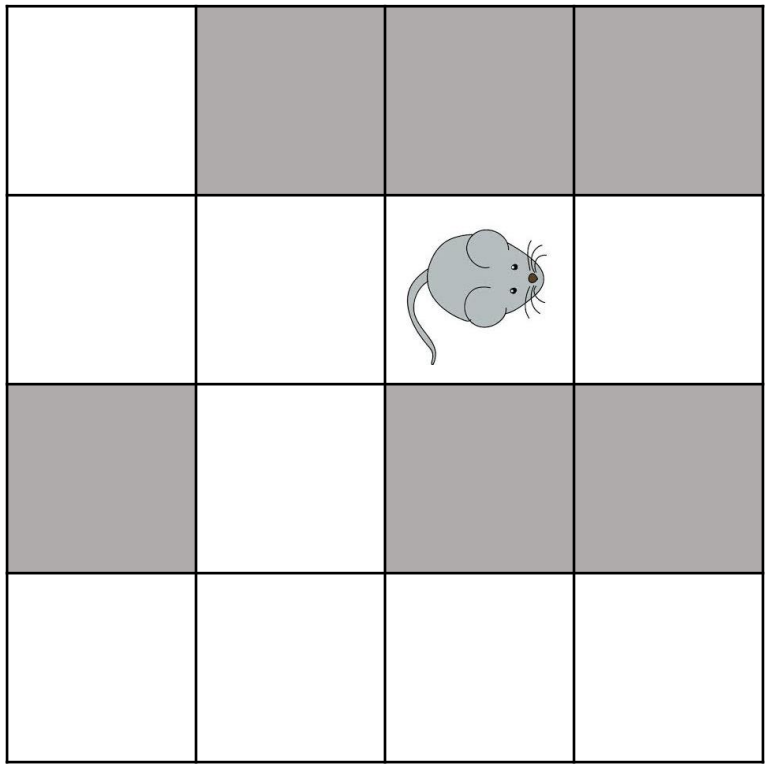
solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 3, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 2, sol[4][4])



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 3, 1, sol[4][4])

Not possible to go to (4,1)

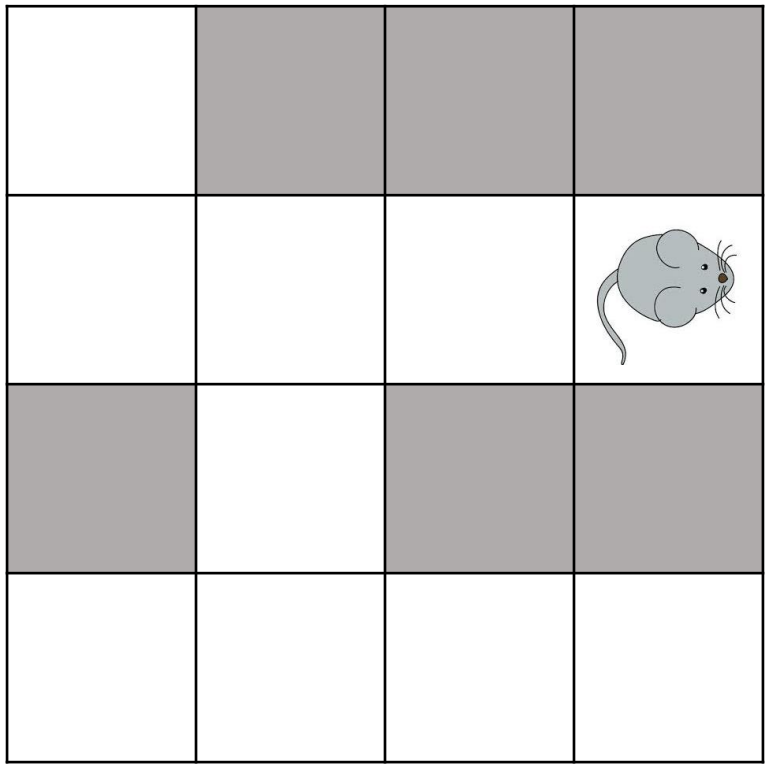
solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 2, 2, sol[4][4])

solveMazeUtil(maze[4][4], 3, 2, sol[4][4])



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 3, 1, sol[4][4])

Not possible to go to (4,1)

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

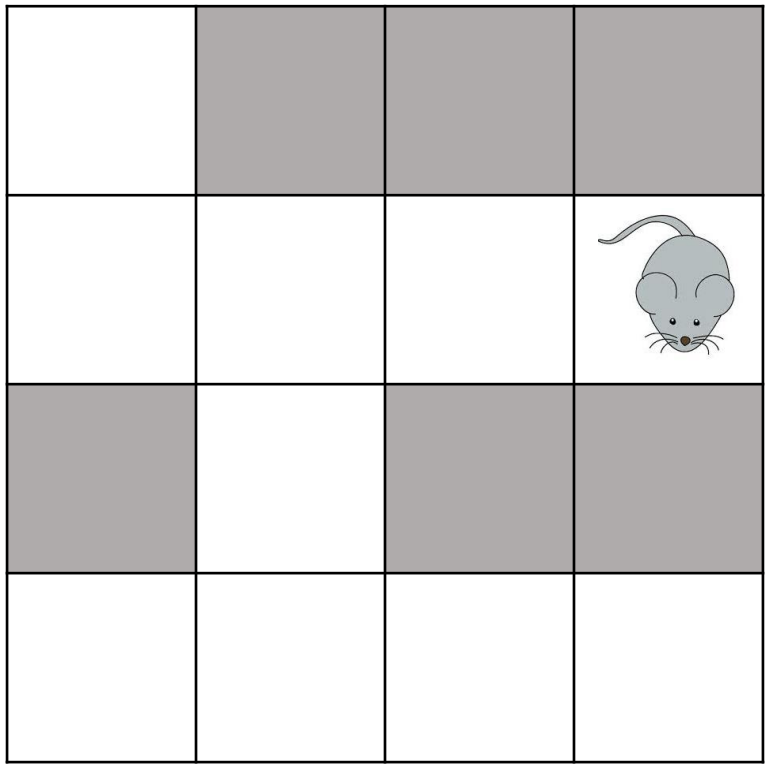
solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 2, 2, sol[4][4])

solveMazeUtil(maze[4][4], 3, 2, sol[4][4])

cannot go in y direction

So we backtrack



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

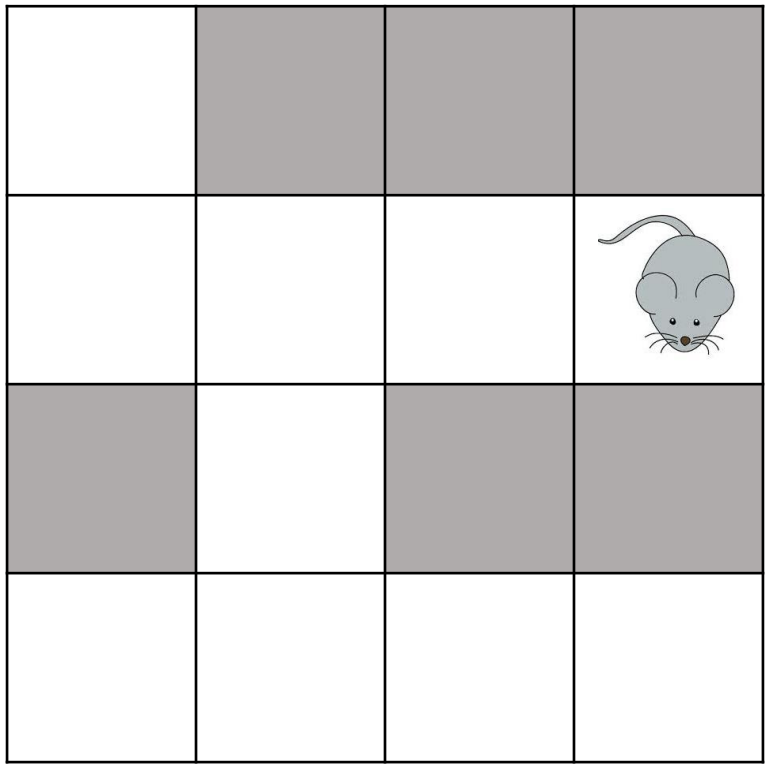
solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 3, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 2, sol[4][4])



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

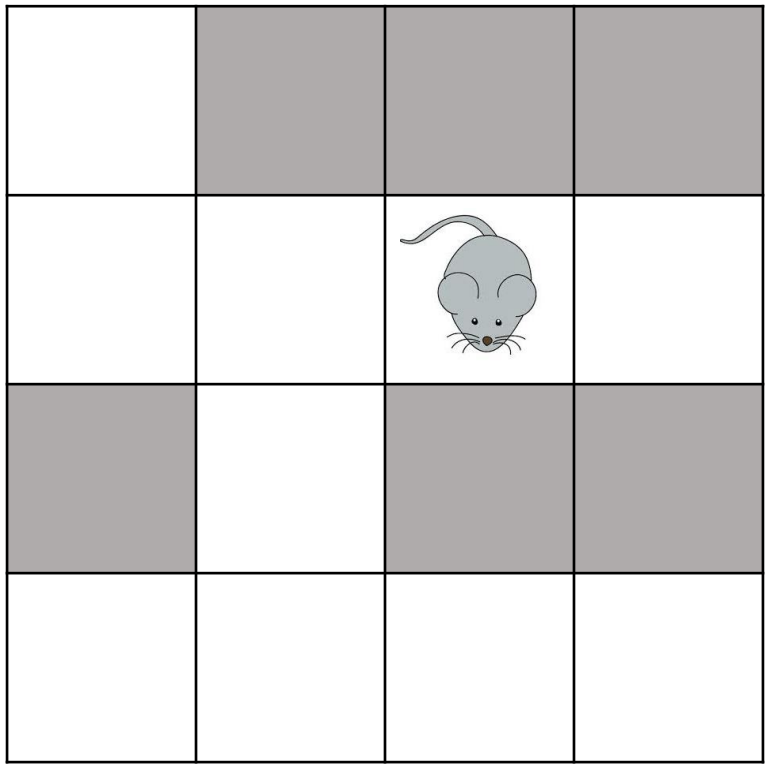
solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 3, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 2, sol[4][4])

cannot go in y direction



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

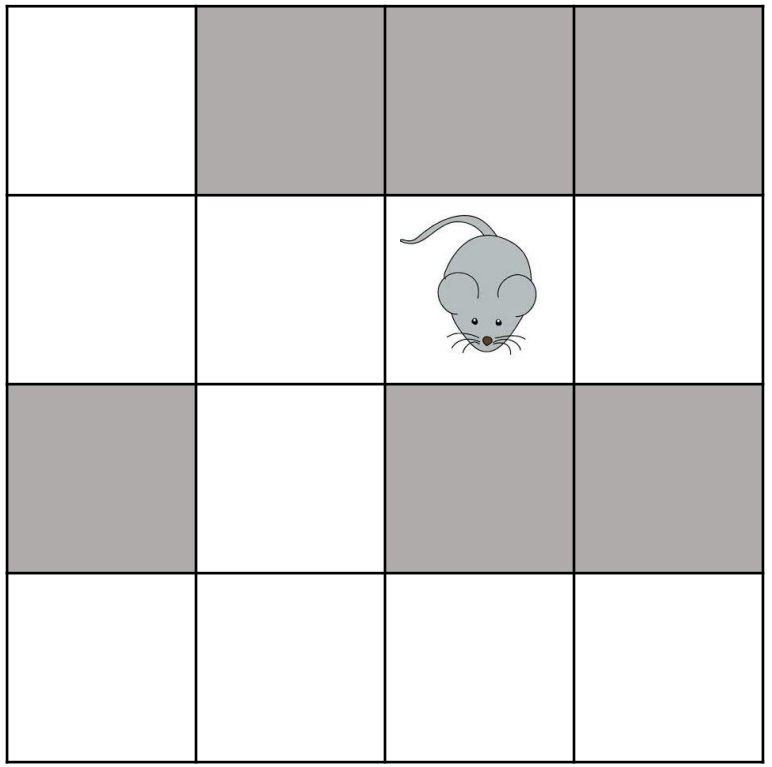
solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

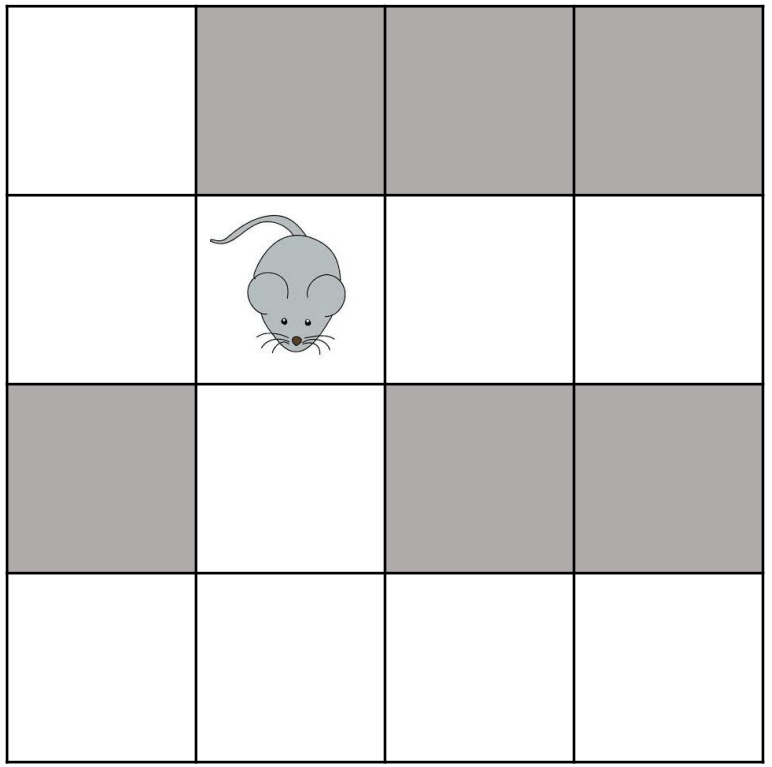
solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

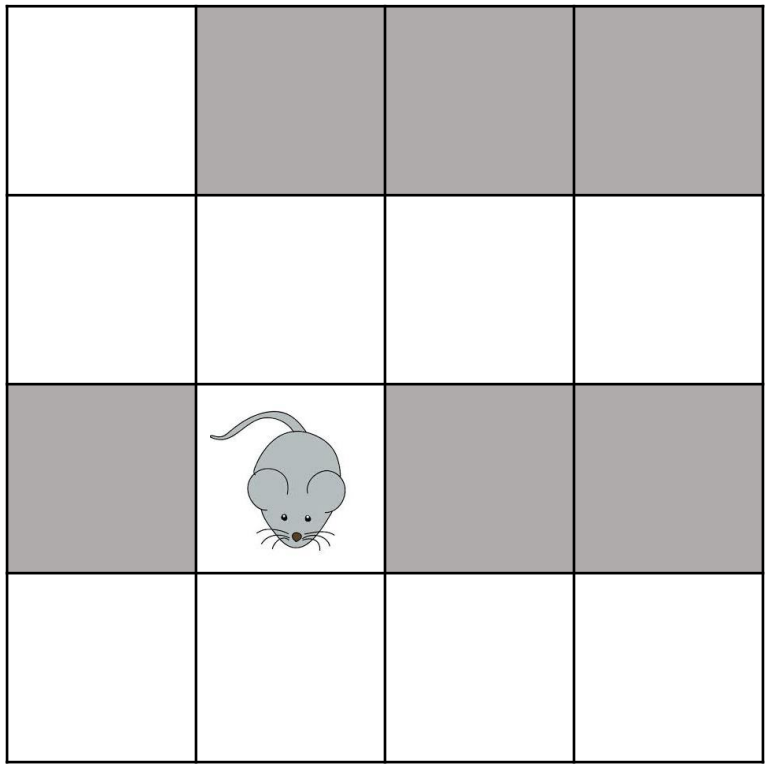
solveMazeUtil(maze[4][4], 2, 2, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 3, sol[4][4])



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

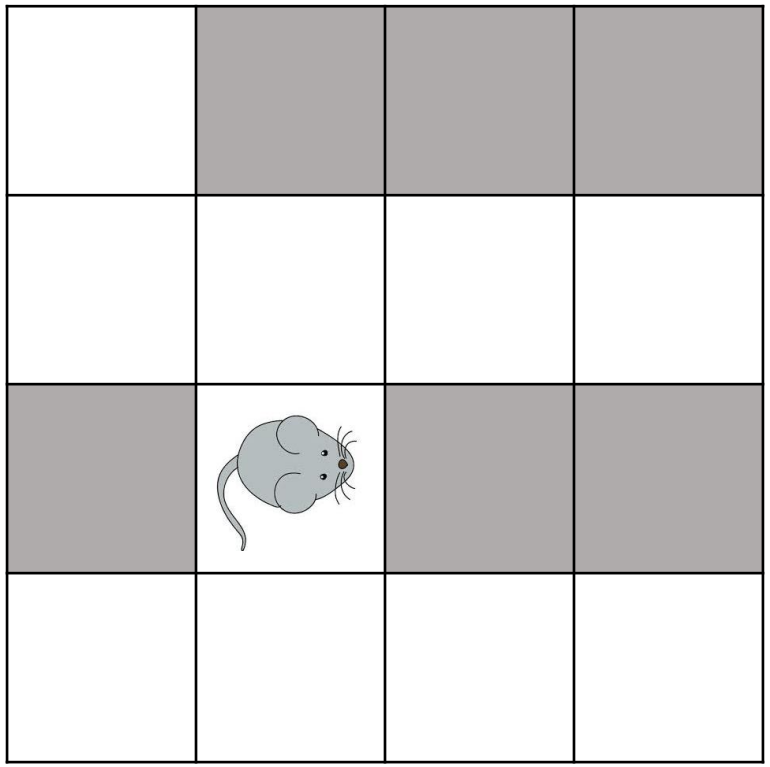
solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 2, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 3, sol[4][4])

cannot go in x direction



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

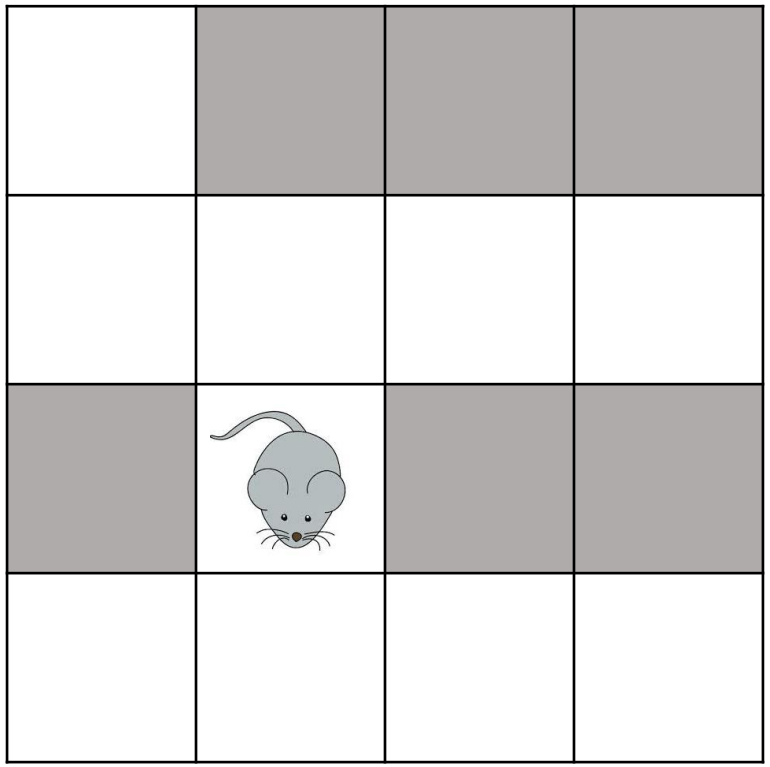
solveMazeUtil(maze[4][4], 2, 2, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 3, sol[4][4])



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 0, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 2, sol[4][4])

solveMazeUtil(maze[4][4], 2, 3, sol[4][4])

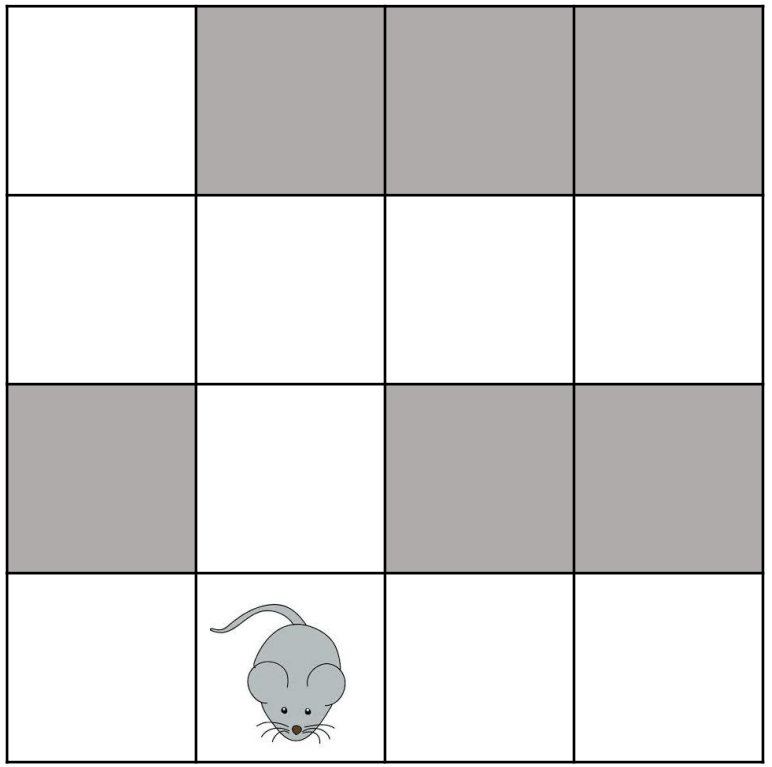
solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 3, sol[4][4])

Not possible to go to (1,4)



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 2, sol[4][4])

solveMazeUtil(maze[4][4], 2, 3, sol[4][4])

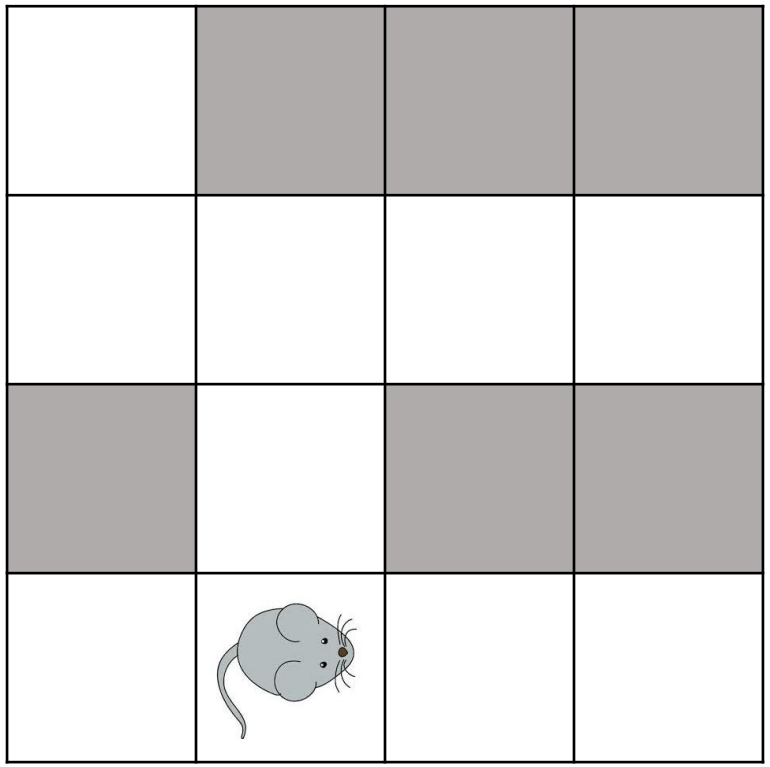
solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 3, sol[4][4])

Not possible to go to (1,4)



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 2, sol[4][4])

solveMazeUtil(maze[4][4], 2, 3, sol[4][4])

solveMazeUtil(maze[4][4], 3, 3, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

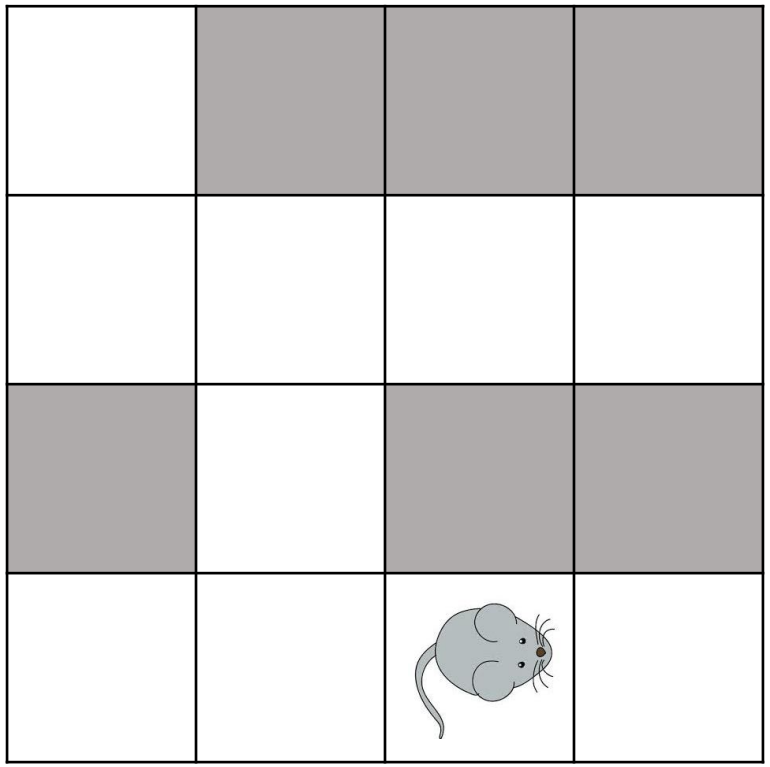
solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 3, sol[4][4])

Not possible to go to (1,4)

Not possible to go to (2,4)



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

solveMazeUtil(maze[4][4], 1, 0, sol[4][4])

solveMazeUtil(maze[4][4], 1, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 1, sol[4][4])

solveMazeUtil(maze[4][4], 2, 2, sol[4][4])

solveMazeUtil(maze[4][4], 2, 3, sol[4][4])

solveMazeUtil(maze[4][4], 3, 3, sol[4][4])

solveMazeUtil(maze[4][4], 0, 1, sol[4][4])

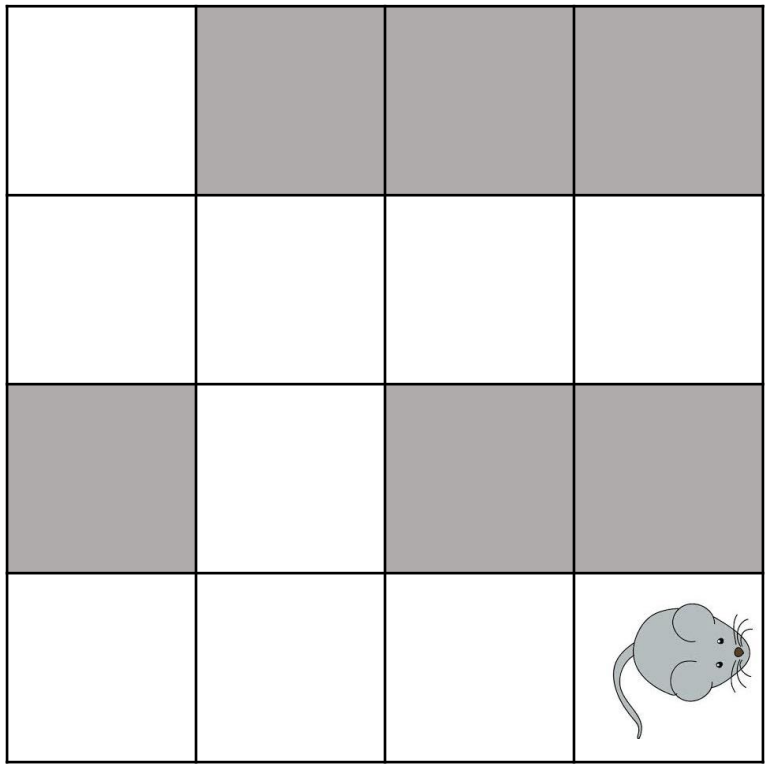
solveMazeUtil(maze[4][4], 0, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 2, sol[4][4])

solveMazeUtil(maze[4][4], 1, 3, sol[4][4])

Not possible to go to (1,4)

Not possible to go to (2,4)



sol Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Pseudo Code

If destination is reached

print the solution matrix

Else

1) **Mark current cell in solution matrix as 1.**
2) **Move forward in the horizontal direction and recursively check if this move leads to a solution.**
3) **If the move chosen in the above step doesn’t lead to a solution then move down and check if this move leads to a solution.**
4) **If none of the above solutions works then unmark this cell as 0 (BACKTRACK) and return false.**

Implementation

```
/* This function solves the Maze problem using Backtracking.  It mainly
   uses solveMazeUtil() to solve the problem. It returns false if no
   path is possible, otherwise return true and prints the path in the
   form of 1s. Please note that there may be more than one solutions,
   this function prints one of the feasible solutions.*/
bool solveMaze(int maze[N][N])
{
    int sol[N][N] = { { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 } };

    if (solveMazeUtil(maze, 0, 0, sol) == false) {
        printf("Solution doesn't exist");
        return false;
    }

    printSolution(sol);
    return true;
}
```

```
/* A recursive utility function to solve Maze problem */
bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])
{
    // if (x, y is goal) return true
    if (x == N - 1 && y == N - 1) {
        sol[x][y] = 1;
        return true;
    }

    // Check if maze[x][y] is valid
    if (isSafe(maze, x, y) == true) {
        // mark x, y as part of solution path
        sol[x][y] = 1;

        /* Move forward in x direction */
        if (solveMazeUtil(maze, x + 1, y, sol) == true)
            return true;

        /* If moving in x direction doesn't give solution then
           Move down in y direction  */
        if (solveMazeUtil(maze, x, y + 1, sol) == true)
            return true;

        /* If none of the above movements work then BACKTRACK:
            unmark x, y as part of solution path */
        sol[x][y] = 0;
        return false;
    }

    return false;
}
```

Time Complexity

O(N): N is the cell number.

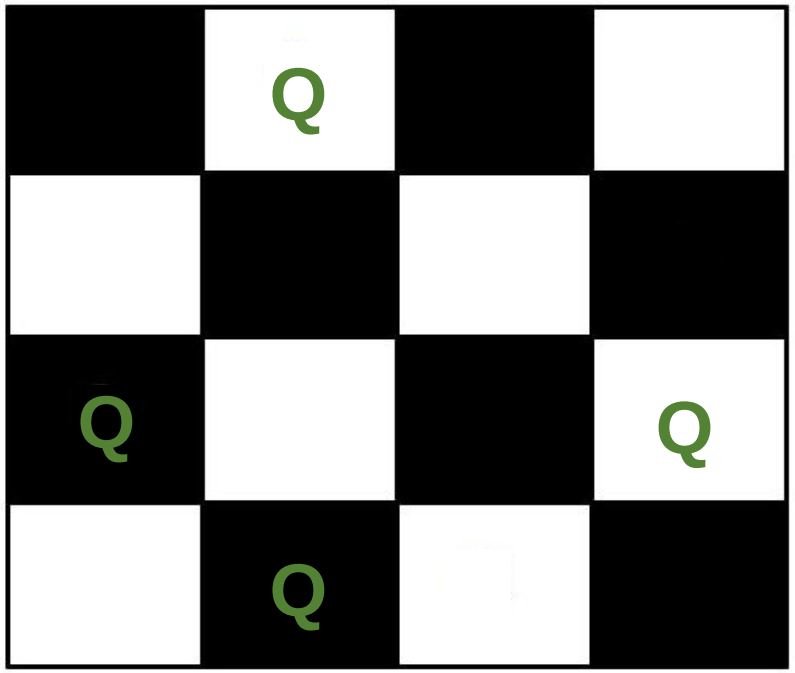
N Queen Problem

Problem Statement

N Queen is the problem of placing N class queens on an NxN chessboard so that no two queens attack each other.

Problem Statement

For example, the following is one of the solutions for 4 Queen problem.



Each queen is placed such that it does not attack any other queen on the board

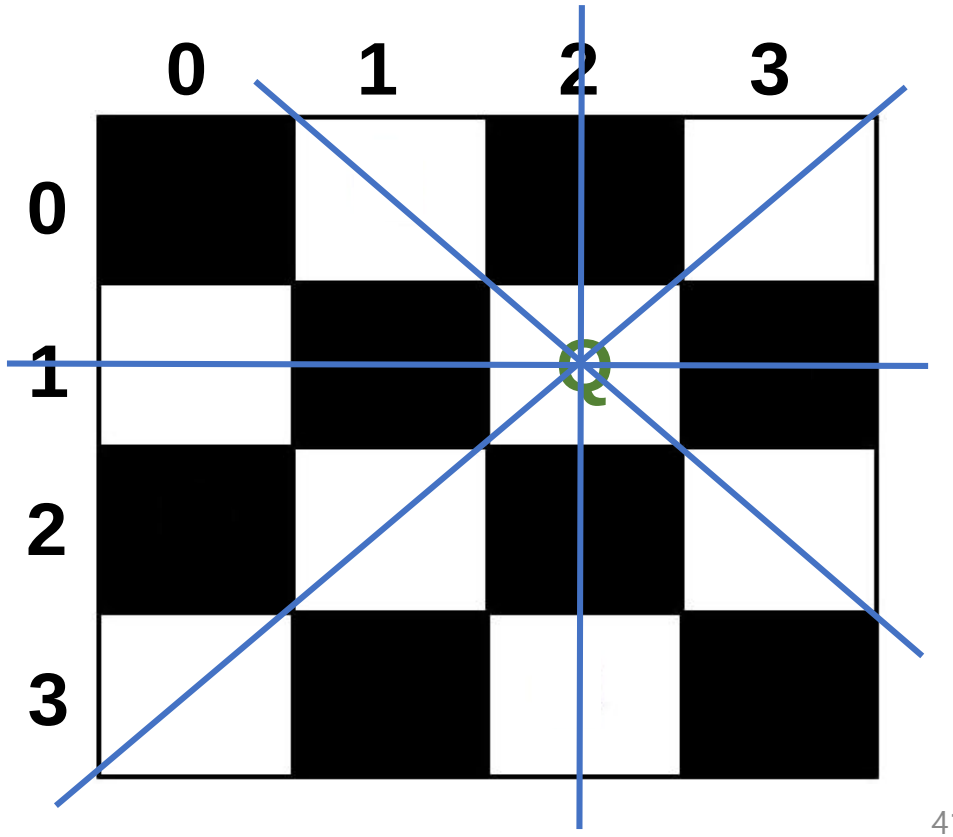
Problem Statement

Let suppose a Queen was placed at square (1, 2).

Attack Points for (1, 2)

row	col	diag1	diag2
(1, 0)	(0, 2)	(0, 1)	(3, 0)
(1, 2)	(2, 2)	(2, 3)	(2, 1)
(1, 3)	(3, 2)		(0, 3)
		row-col	row+col

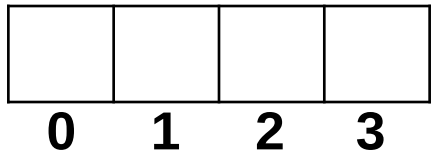
We should avoid the attack points as other queens' locations.



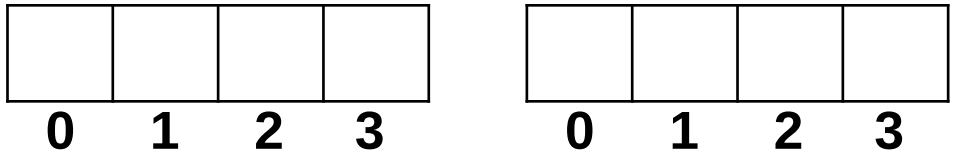
Solutions

Col

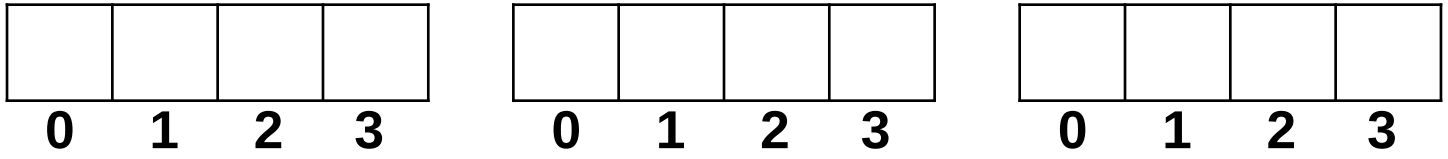
0



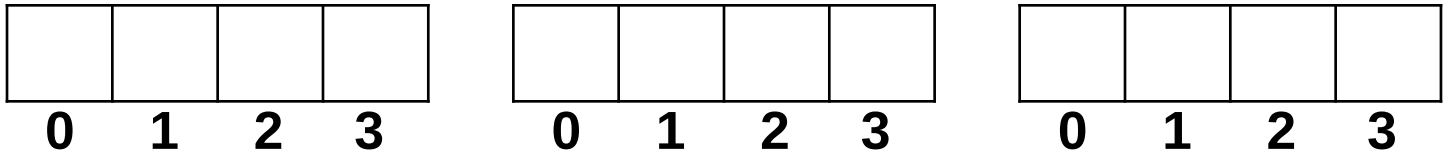
1



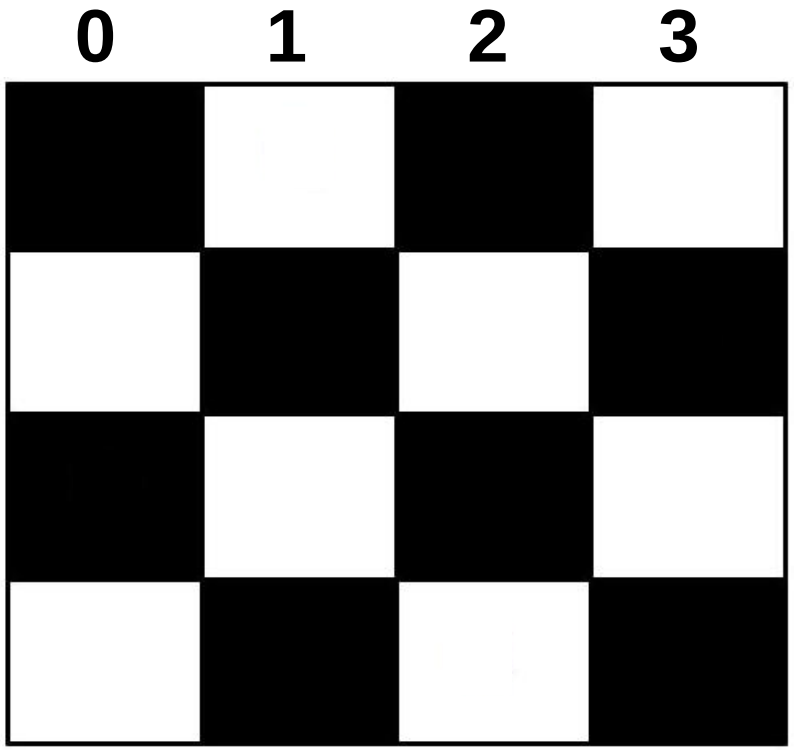
2



3



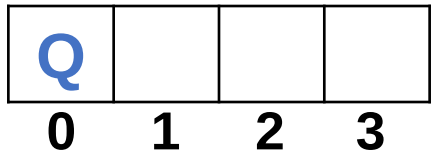
Positions:



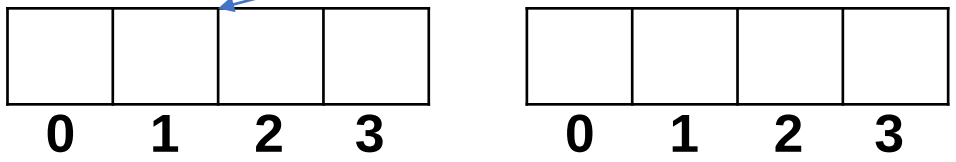
Solutions

Col

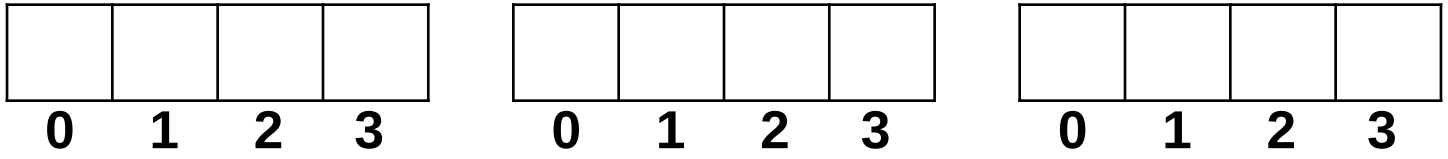
0



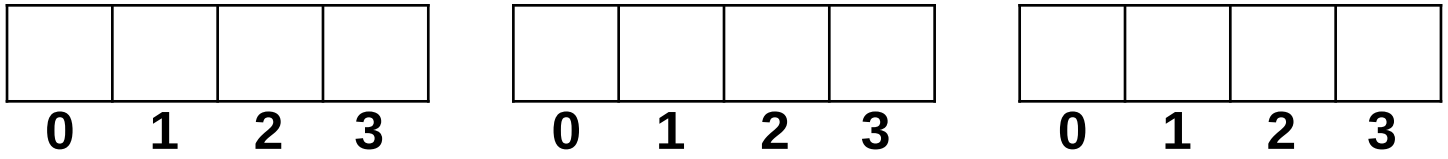
1



2

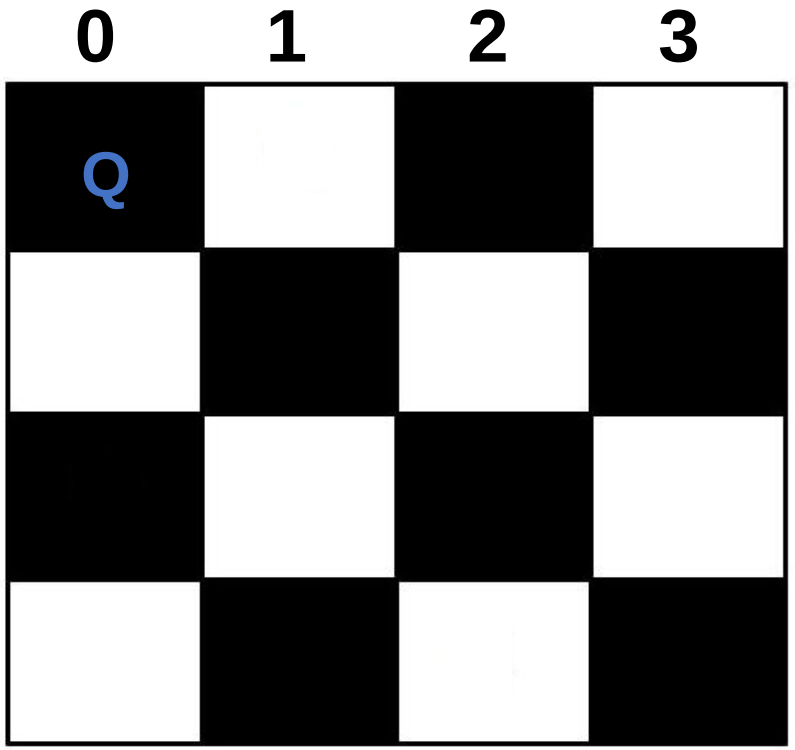


3

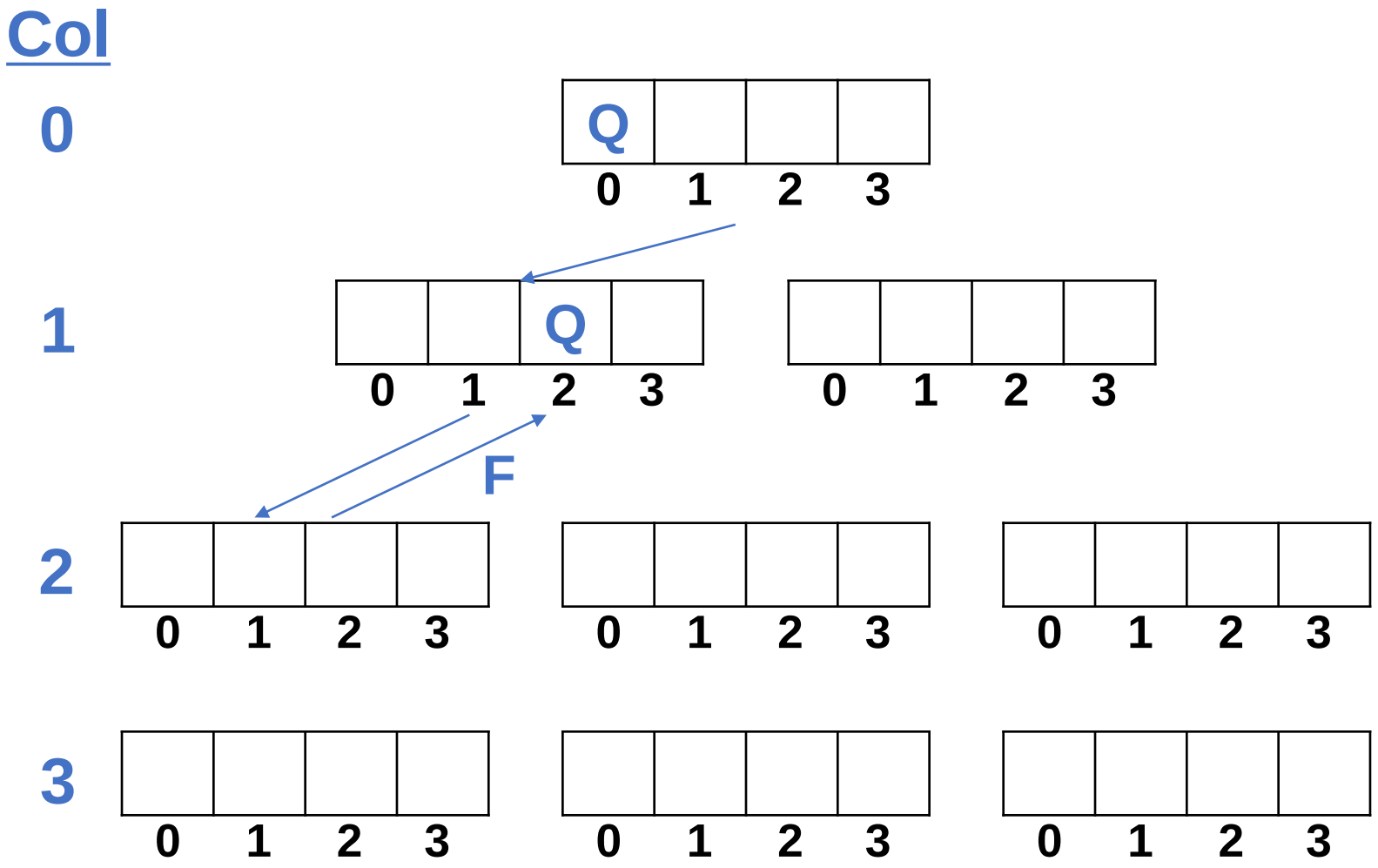


Positions:

(0, 0)

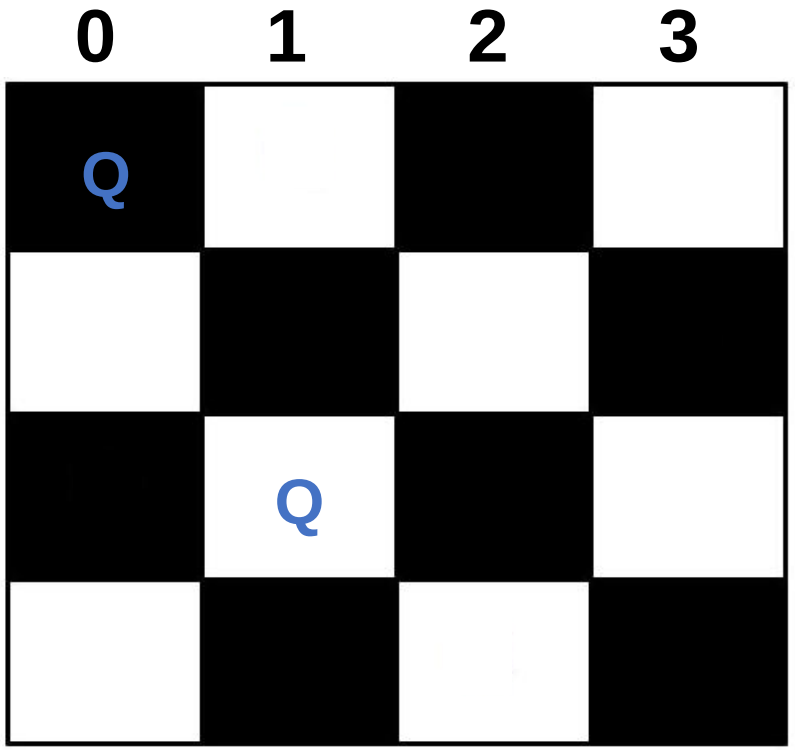


Solutions



Positions:

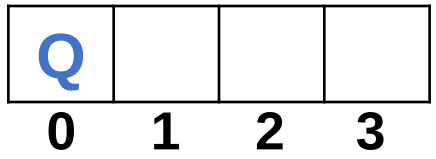
(0, 0)
(2, 1)



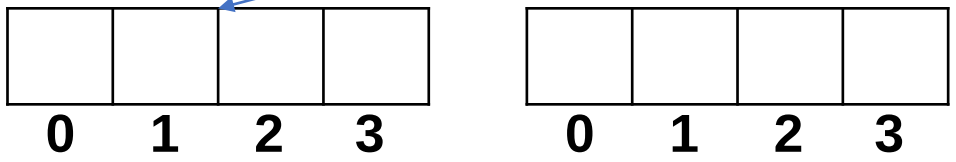
Solutions

Col

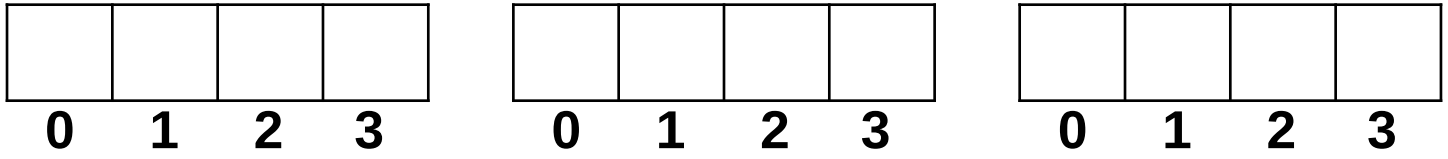
0



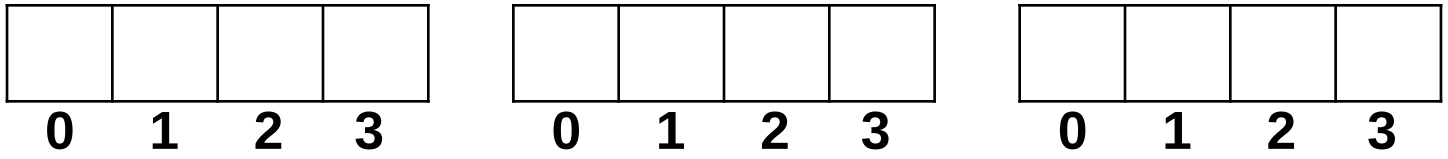
1



2

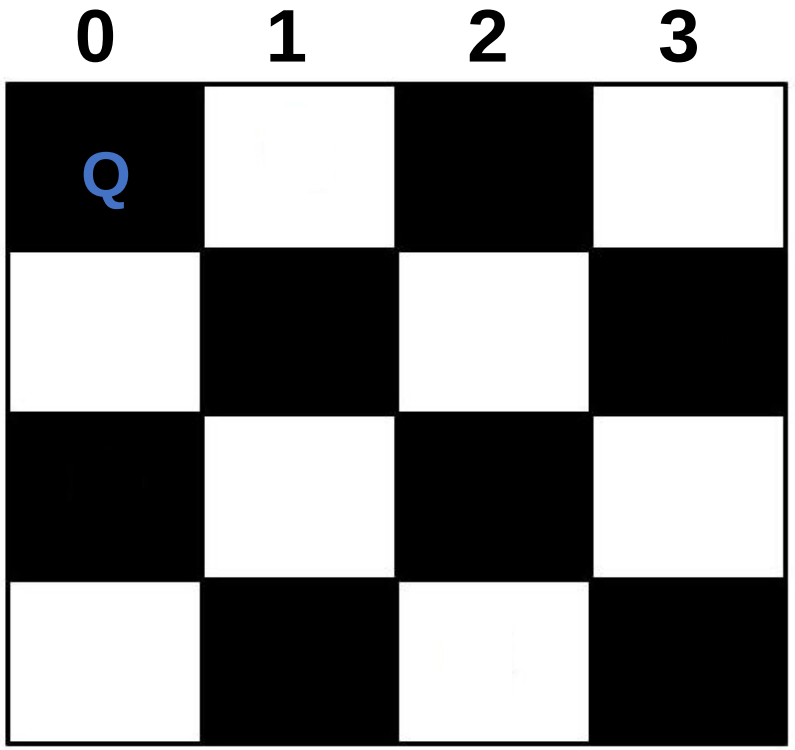


3

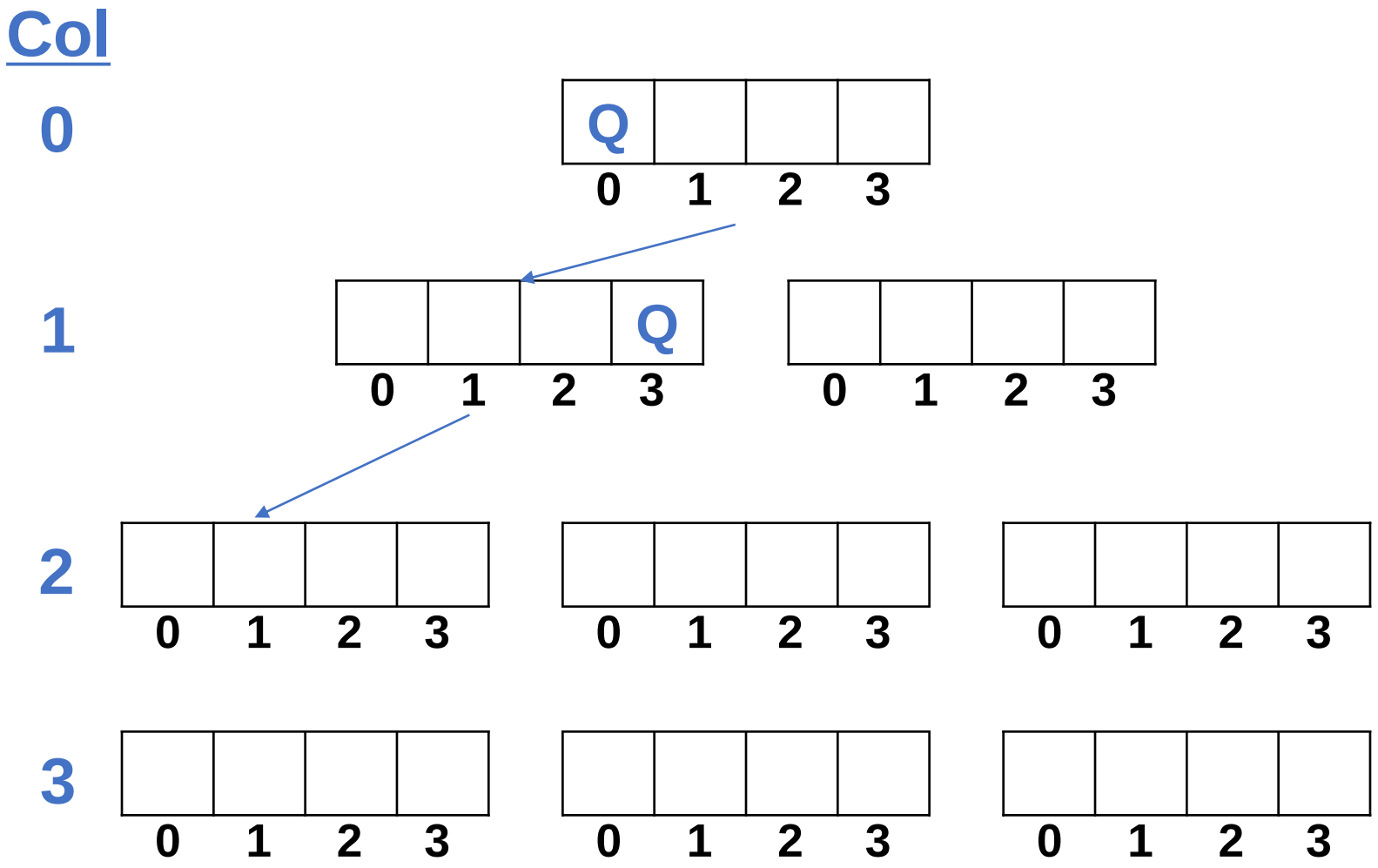


Positions:

(0, 0)

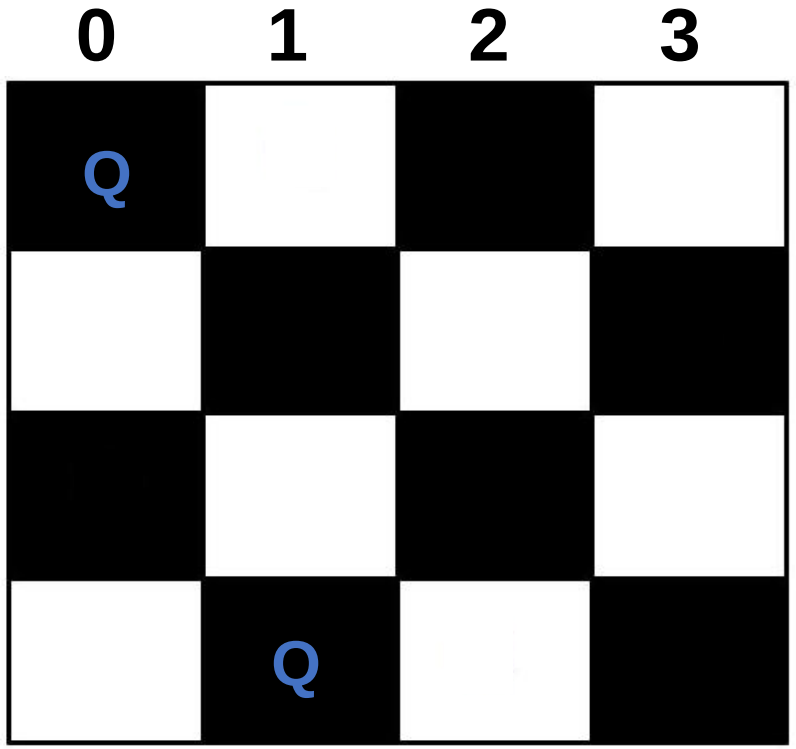


Solutions

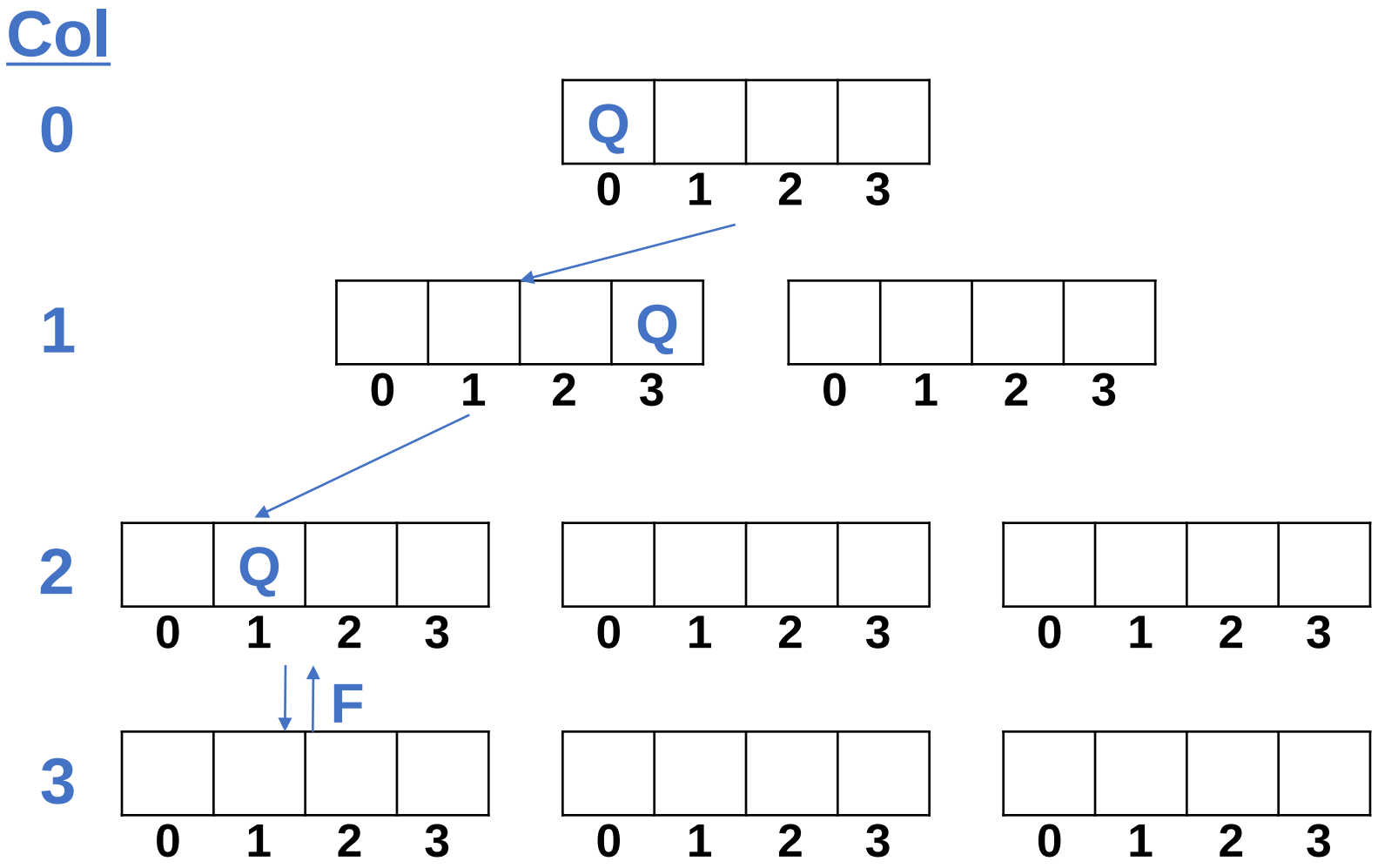


Positions:

(0, 0)
(3, 1)

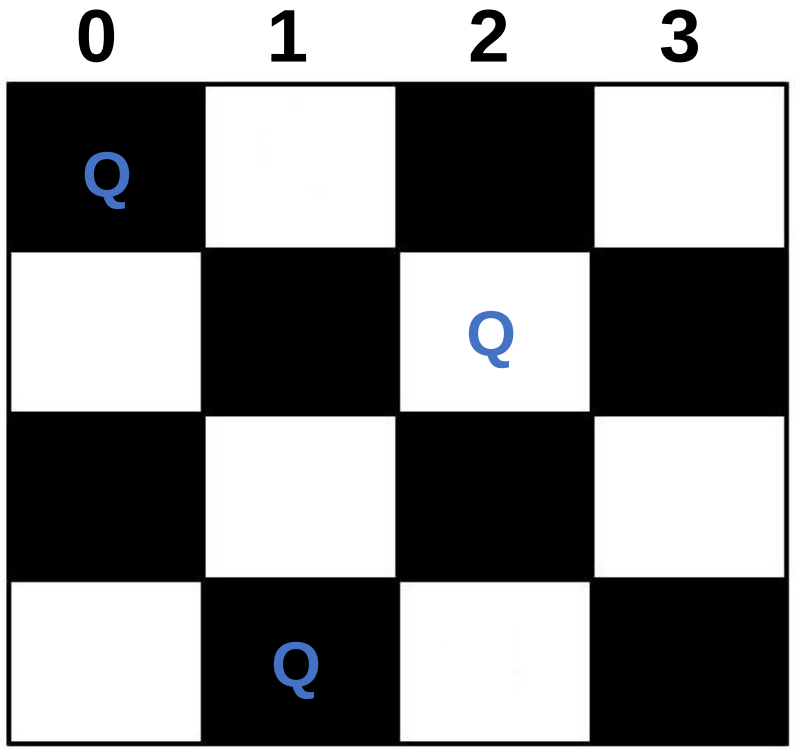


Solutions

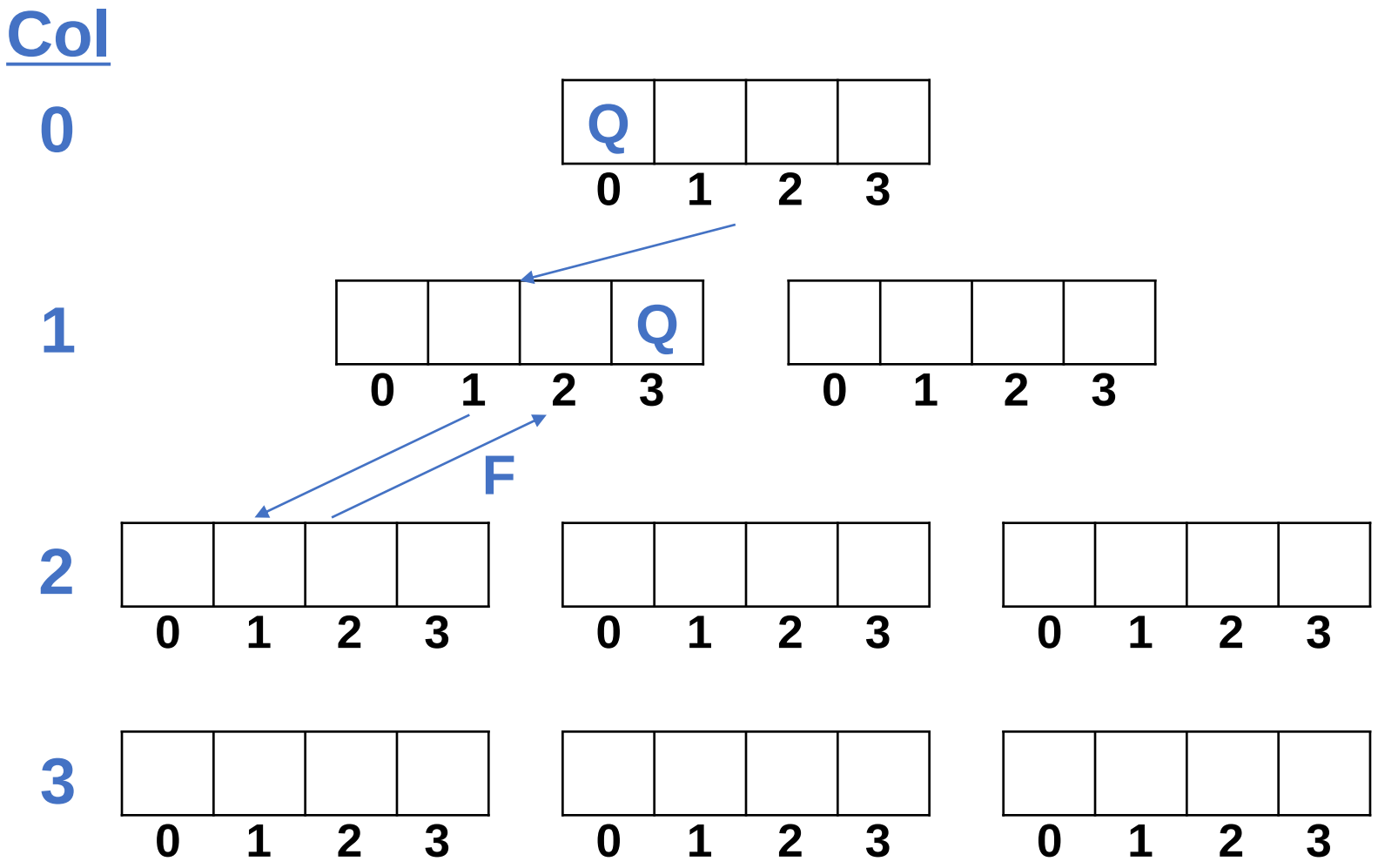


Positions:

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

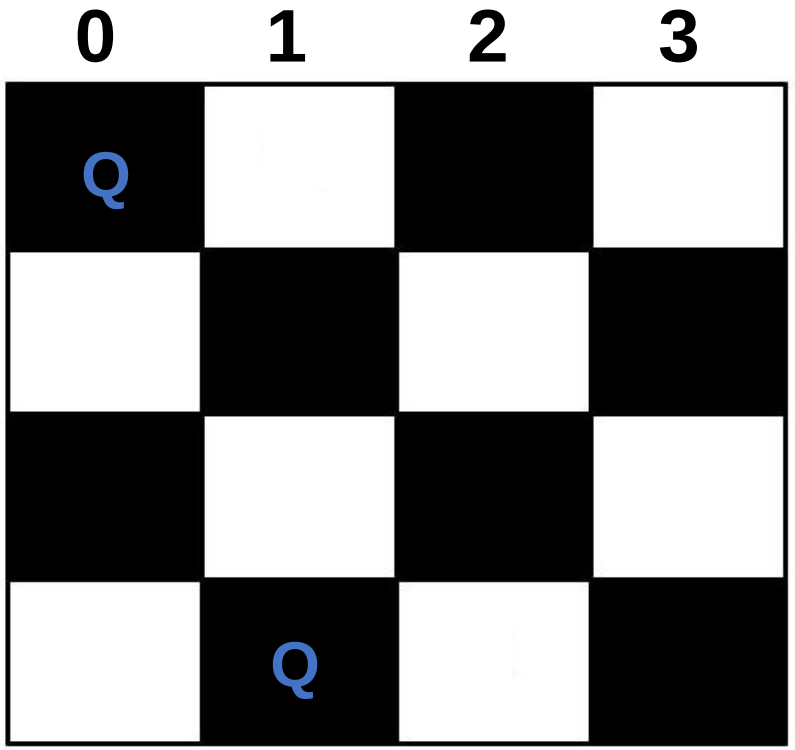


Solutions



Positions:

(0, 0)
(3, 1)



Solutions

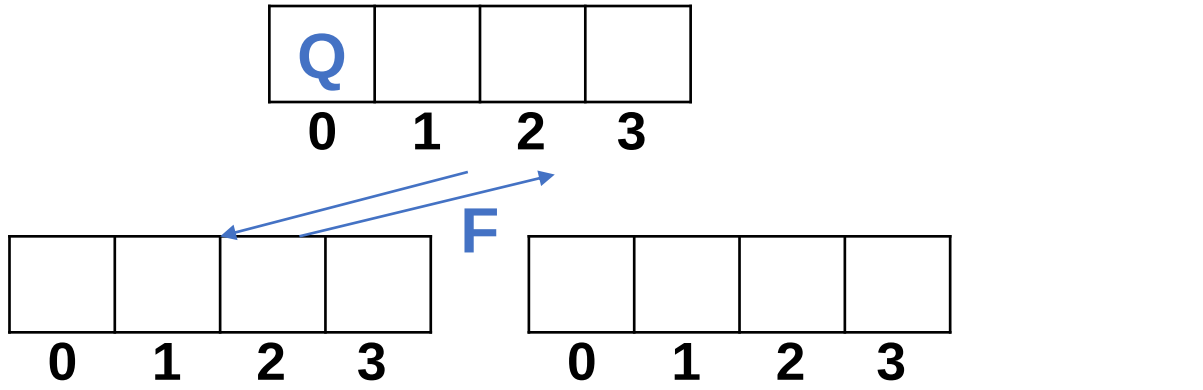
Col

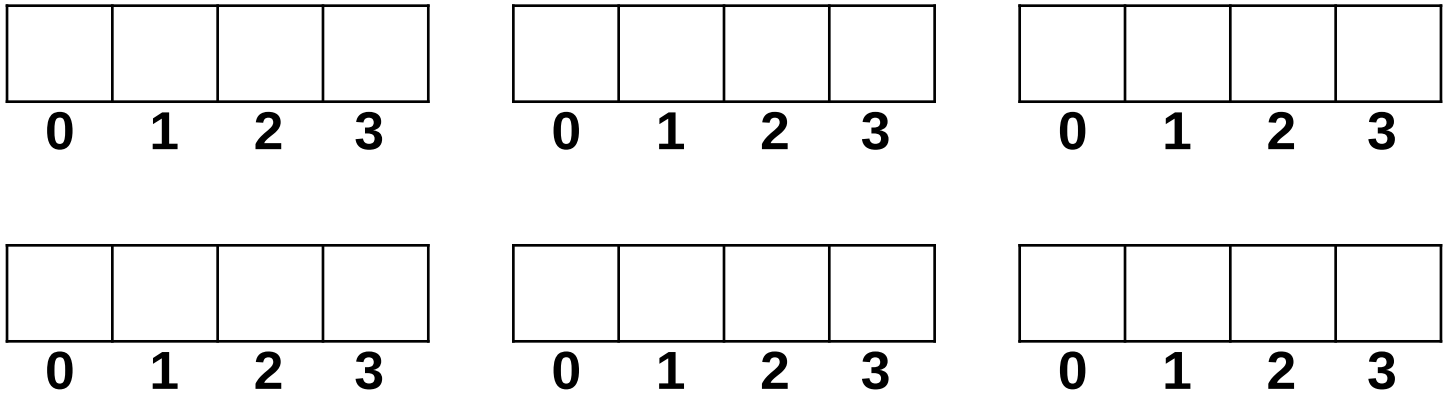
0

1

2

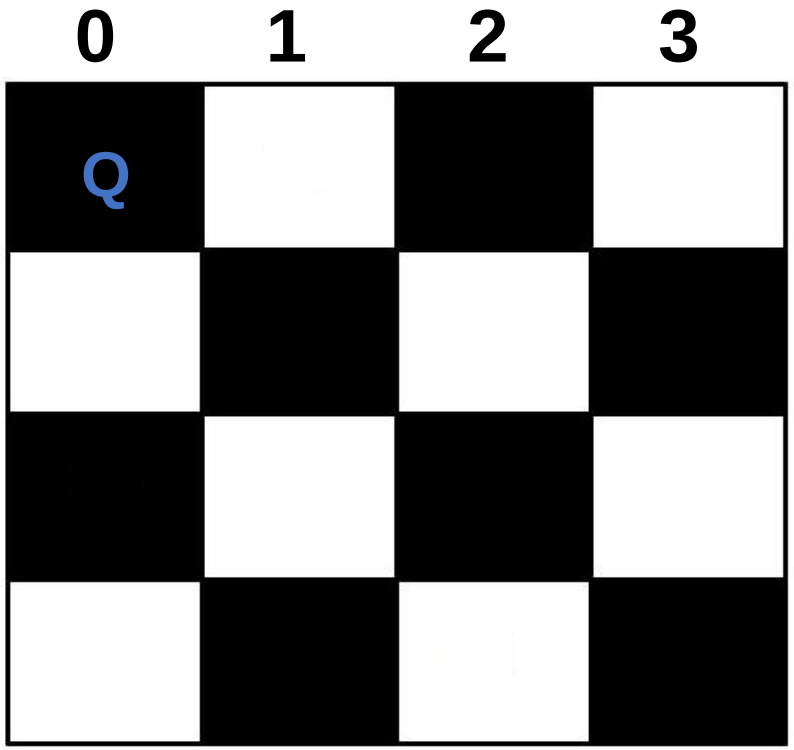
3





Positions:

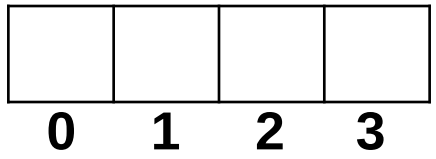
(0, 0)



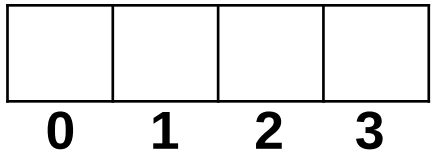
Solutions

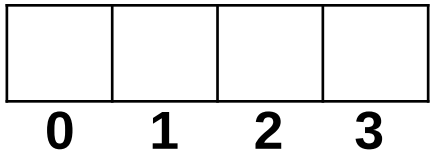
Col

0

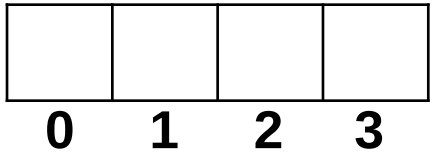


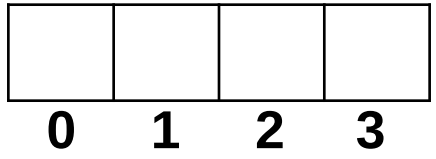
1

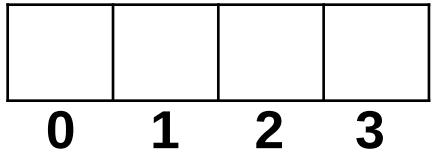




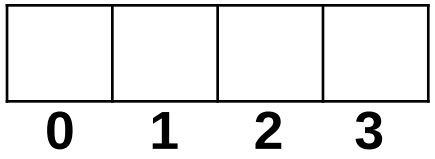
2

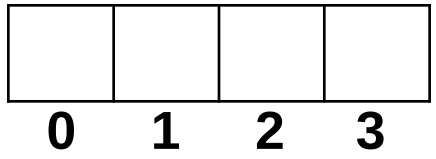


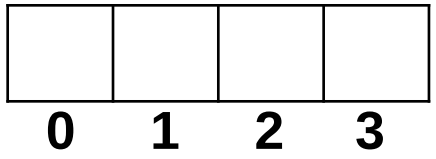




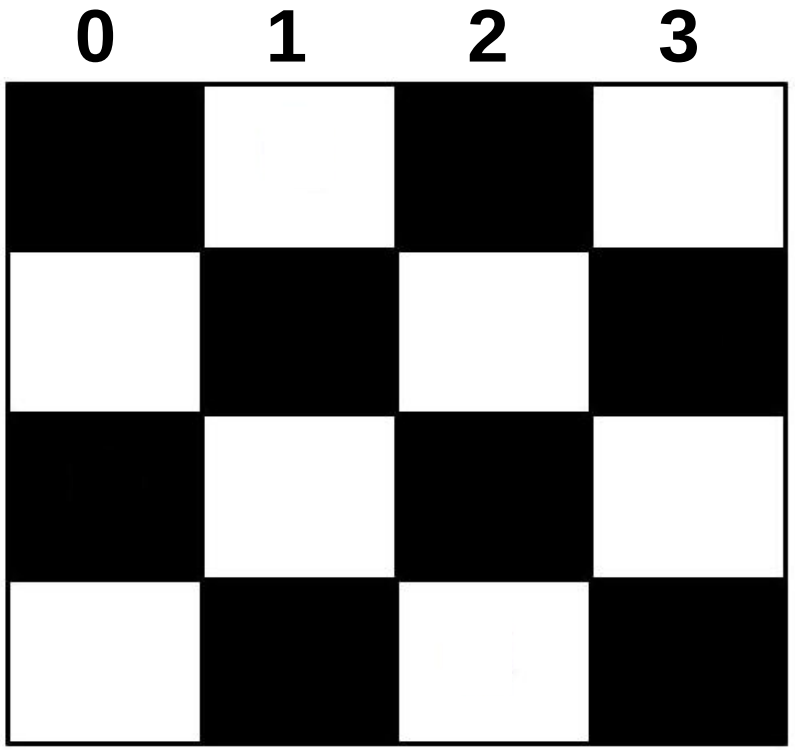
3







Positions:



Solutions

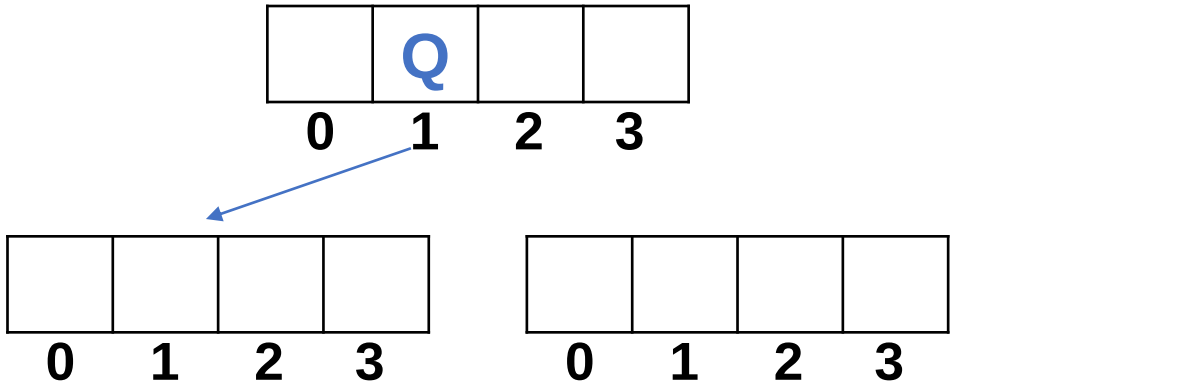
Col

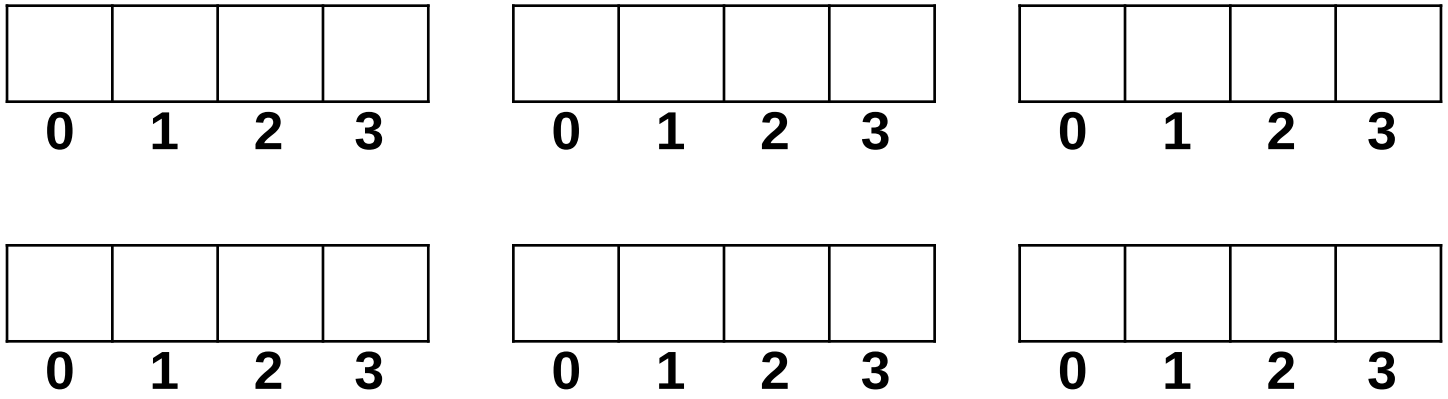
0

1

2

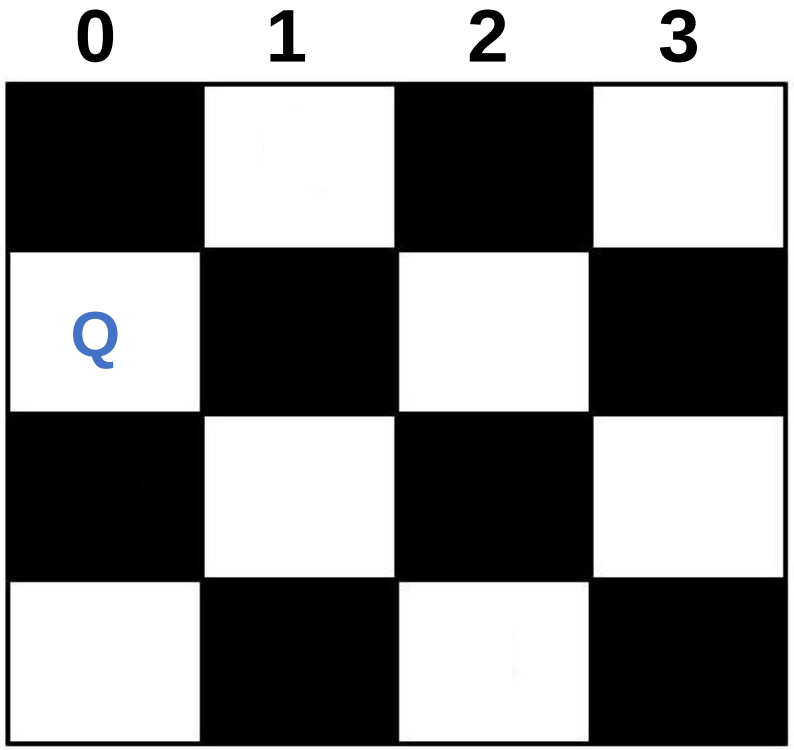
3



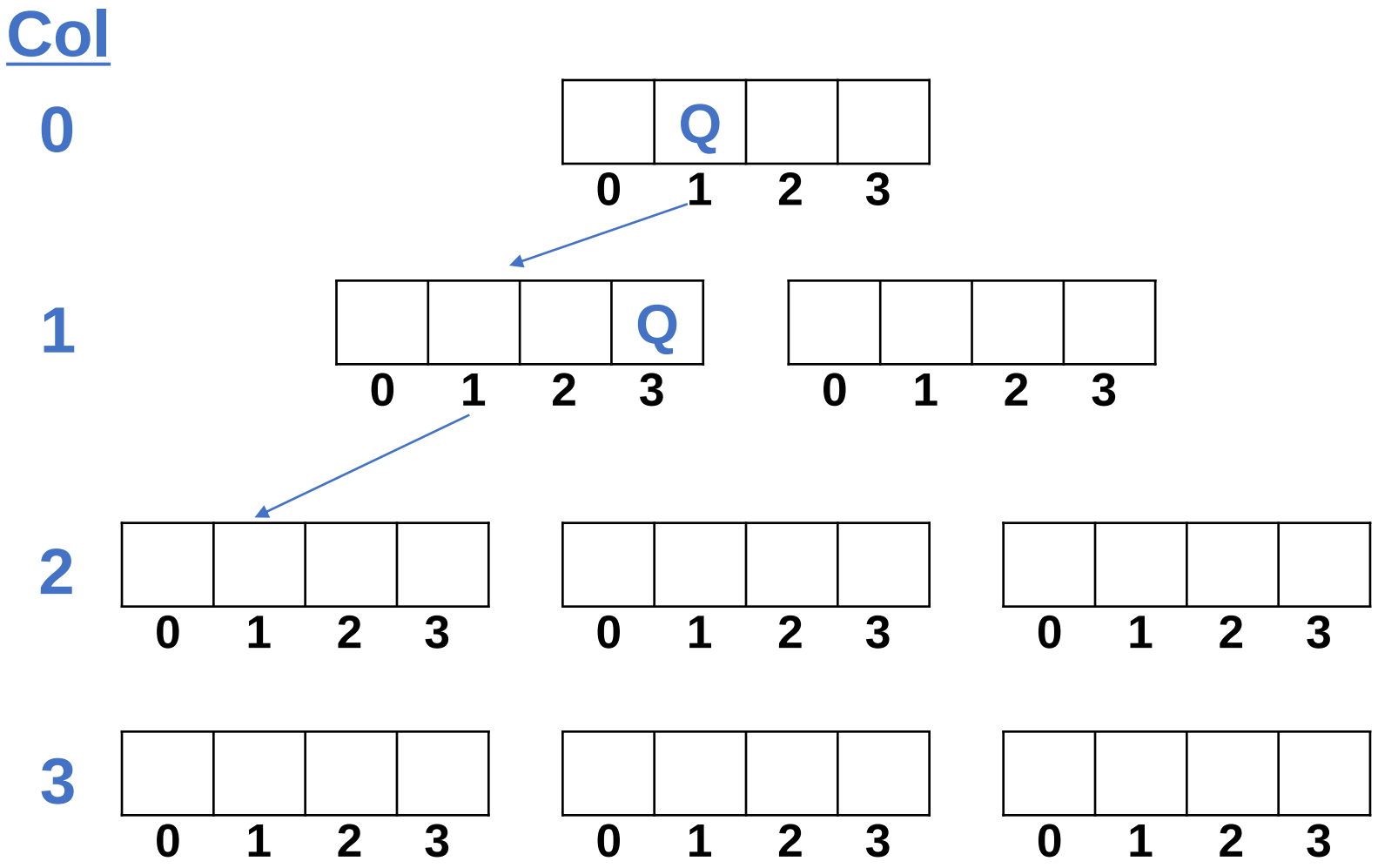


Positions:

(1, 0)

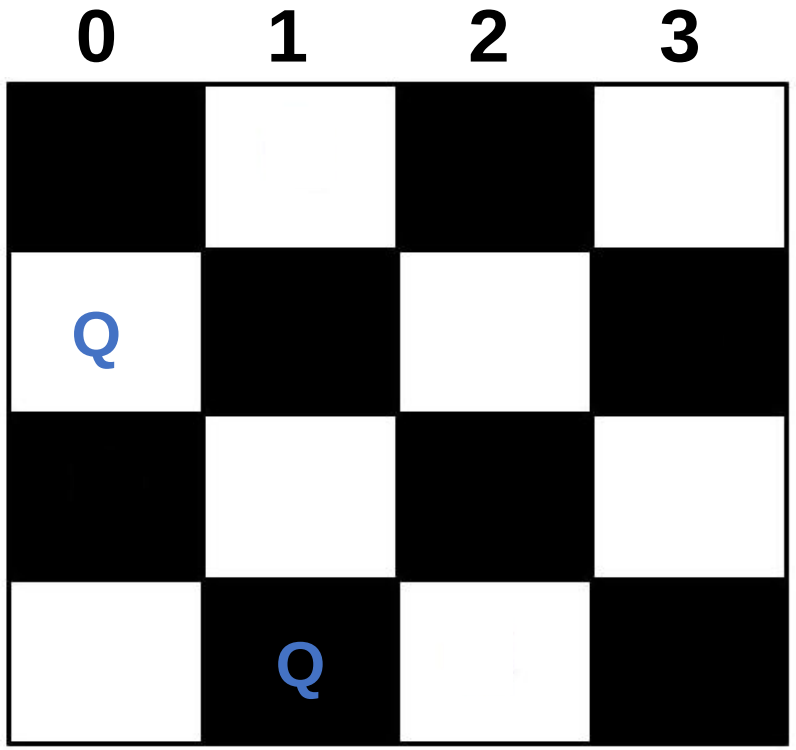


Solutions

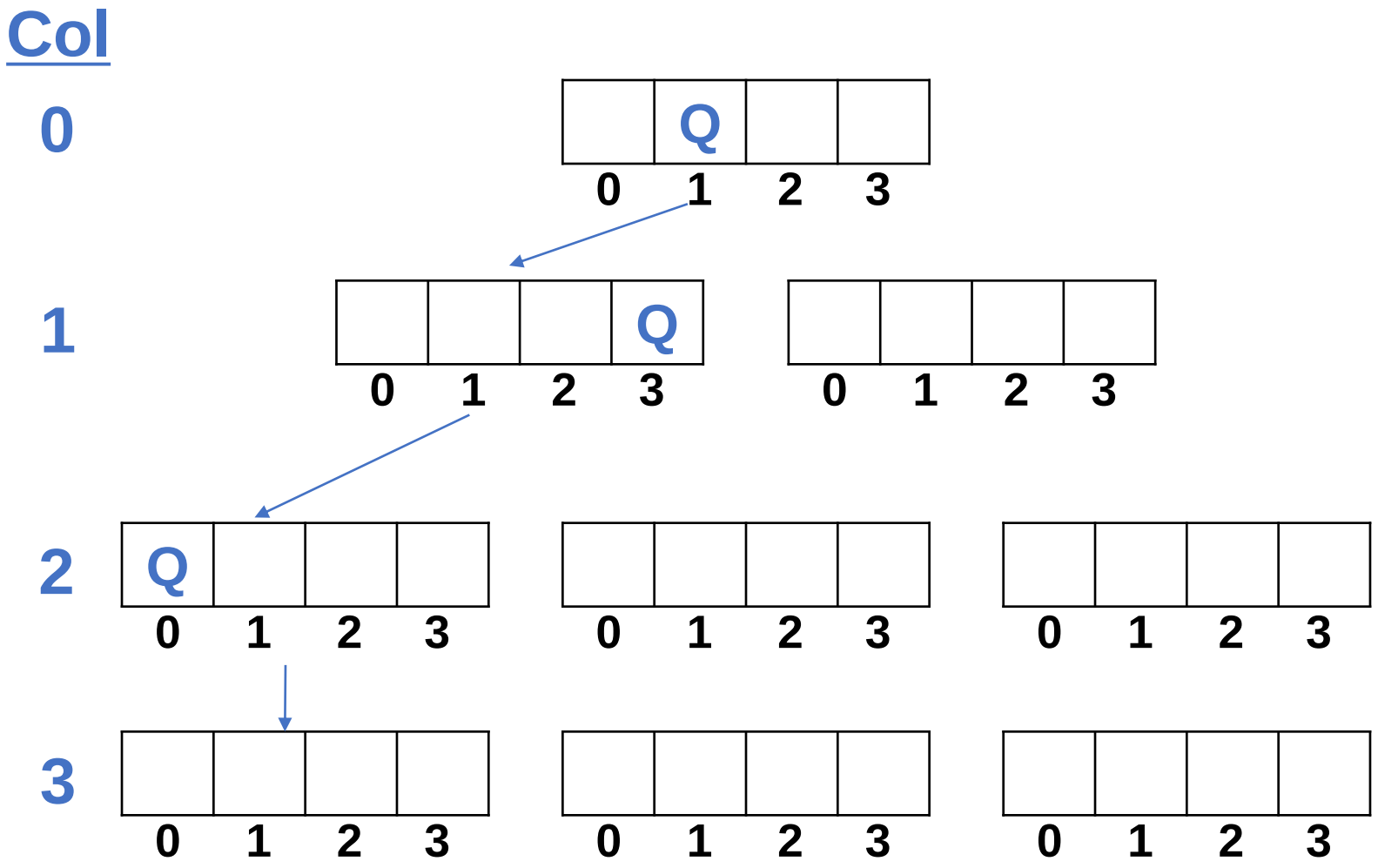


Positions:

(1, 0)
(3, 1)

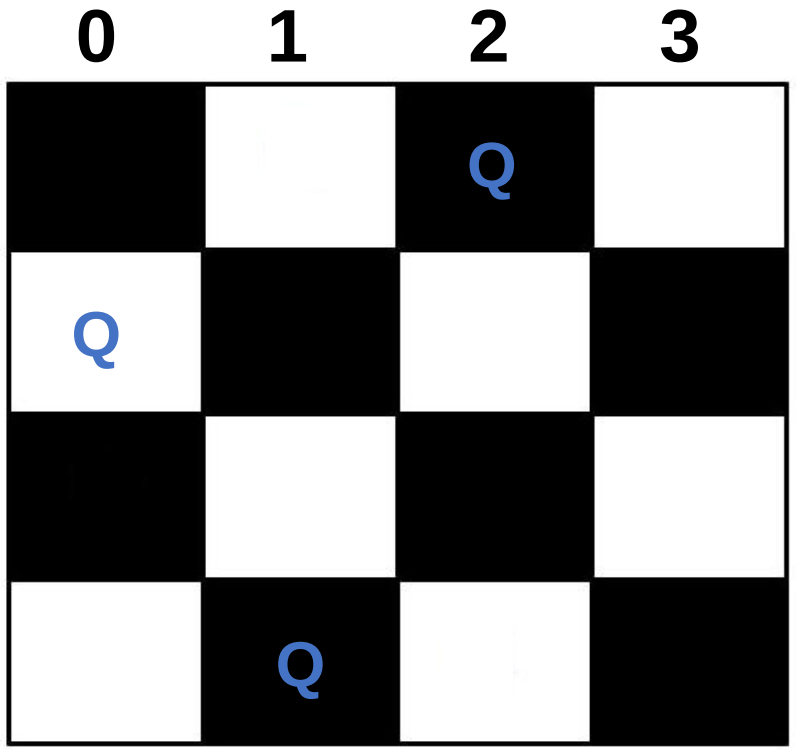


Solutions

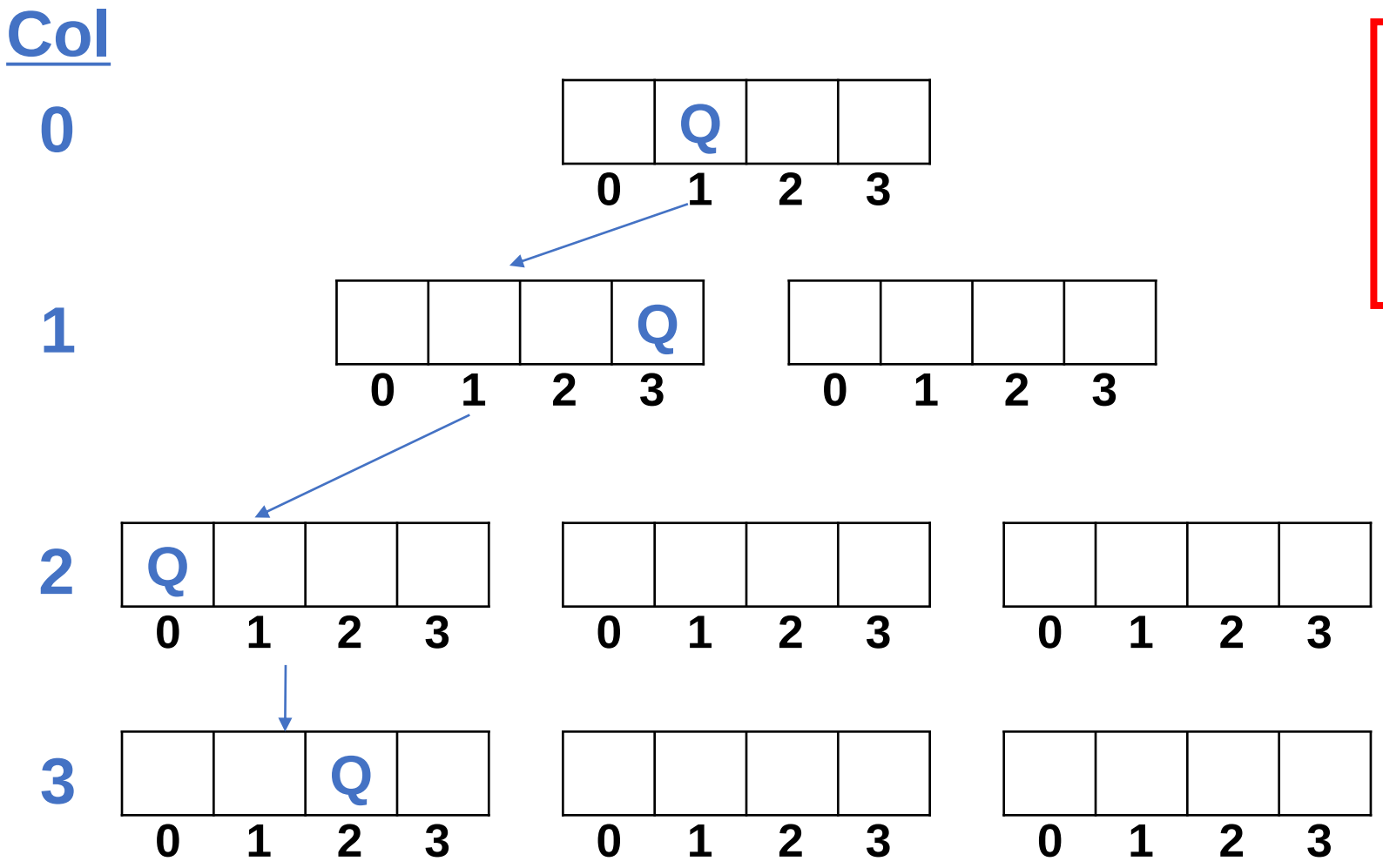


Positions:

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

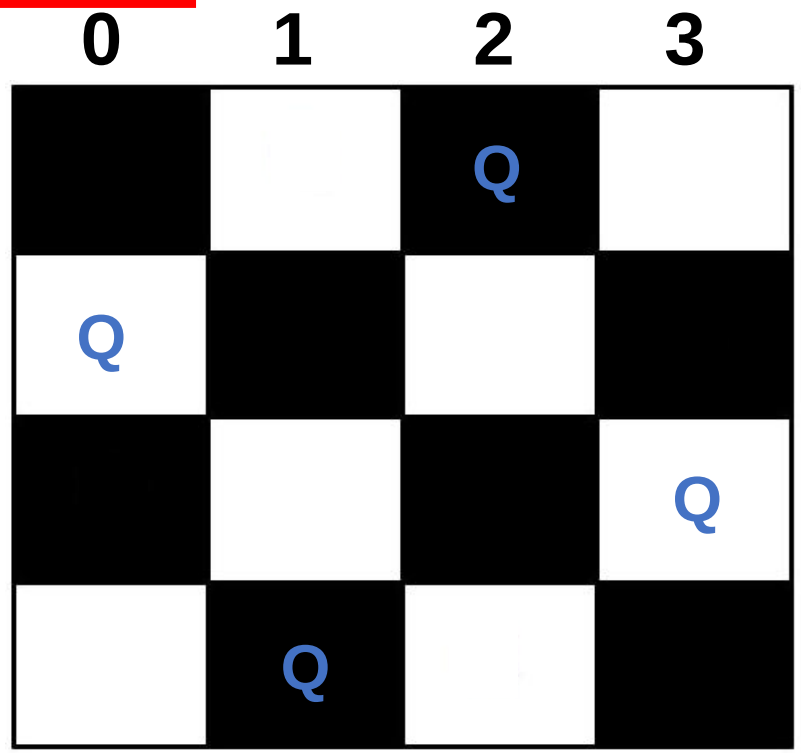


Solutions



Positions:

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



Implementation

```
/* 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)
        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 colum col  then return false */
    return false;
}
```

Time Complexity

$T(n) = n\,T(n-1) + O(n^2)$

→ $O(n!)$

Reference

- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms”,* September 29, 2004

- https://www.geeksforgeeks.org