Design Technique: Backtracking Algorithms

Backtracking Algorithms

A backtracking algorithm is a problem-solving algorithm that uses a **brute force approach** to find the desired output.

The Brute force approach tries out all the possible solutions and choose the desired/best solutions. So, its complexity will be very high. While, in backtracking, only a few solutions are explored.

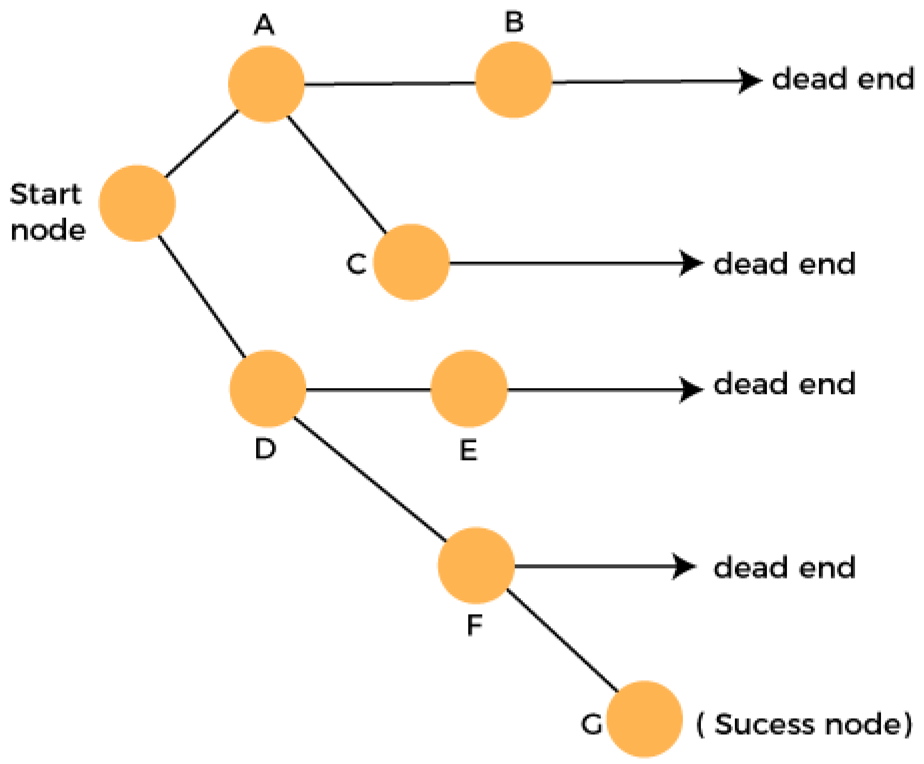
The term backtracking suggests if the current solution is not suitable, then backtrack and try other solutions.

Backtracking is used to solve problems in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion.

Backtracking Algorithms

Backtracking is a modified depth-first search of a tree.

It is the procedure whereby, after determining that a node can lead to nothing but dead nodes, we go back ("backtrack") to the node's parent and proceed with the search on the next child.



Backtracking Algorithms

- Based on depth-first recursive search

Approach

```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
```

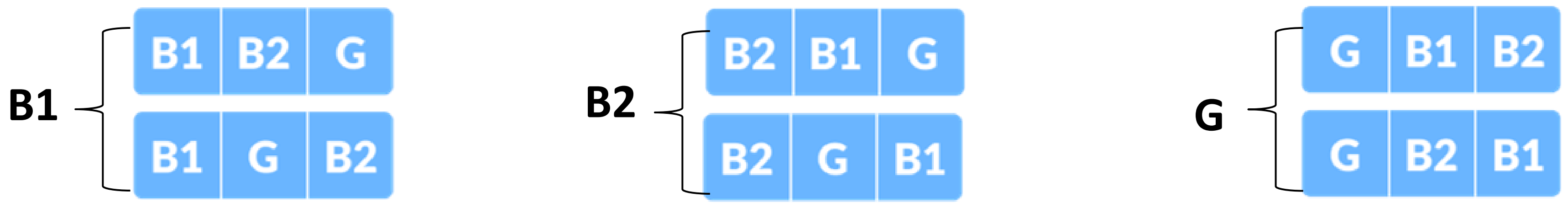
Backtracking Algorithms

Problem: We want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches. **Constraint: Girl should not be on the middle bench.**

Brute-Force Approach

Problem: We want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches. **Constraint: Girl should not be on the middle bench.**

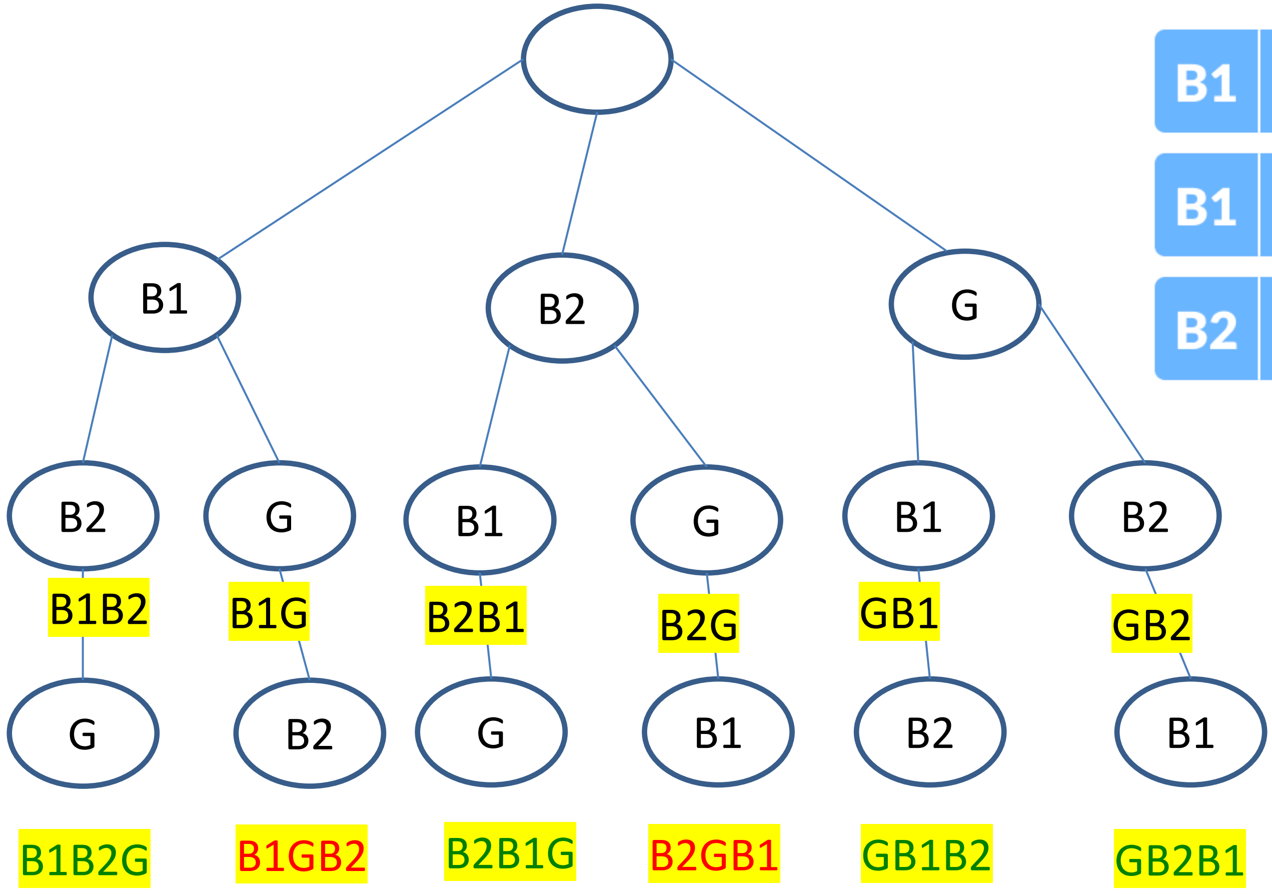
There are a total of 3! = 6 possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.



For better understanding, a state-space tree can be constructed.

Brute-Force Approach

Problem: We want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches. **Constraint: Girl should not be on the middle bench.**

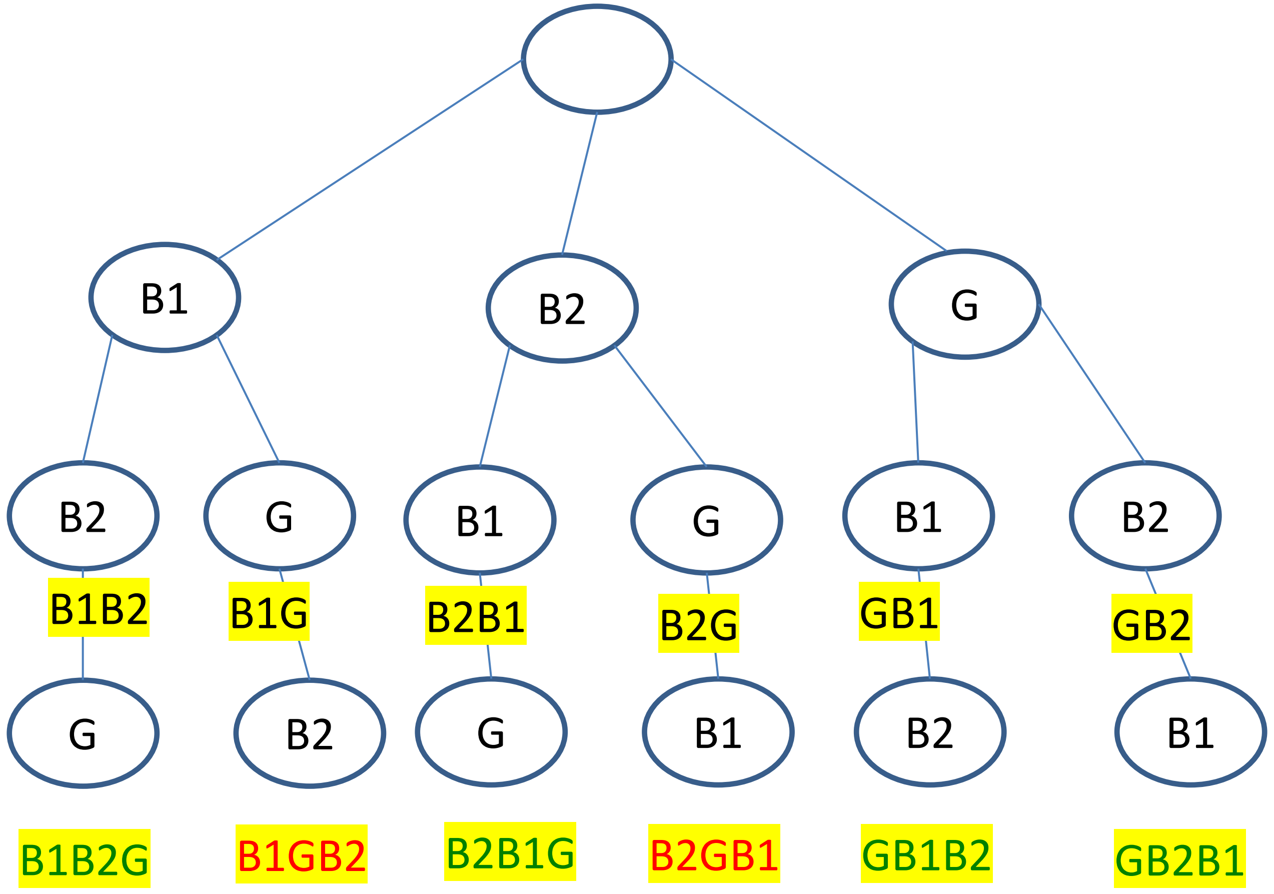


The state space tree

B1	B2	G	B2	G	B1
B1	G	B2	G	B1	B2
B2	B1	G	G	B2	B1

Brute-Force Approach

Problem: We want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches. **Constraint: Girl should not be on the middle bench.**



The state space tree

Time Complexity

Total possibility for n boys and girls = n!

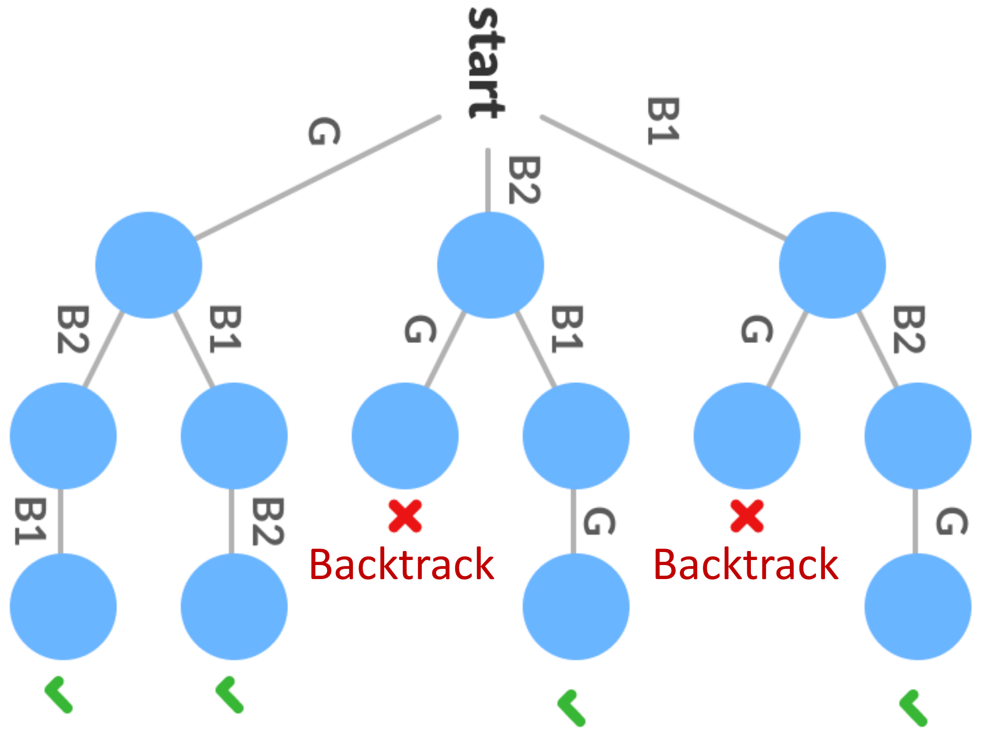
Time taken to find the solution that satisfy the constraint= n*n!

Total time= O(n*n!)

Backtracking Algorithms

Problem: We want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches. **Constraint: Girl should not be on the middle bench.**

In backtracking, if a generated sub solutions do not satisfy the constraint, then backtrack and try other possible solutions. No need to generate all possible solutions.



The N-Queens Problem

Problem: To place **n - queens** in such a manner on an **n x n chessboard** that no queens **attack each other** by being in the **same row, column or diagonal**.

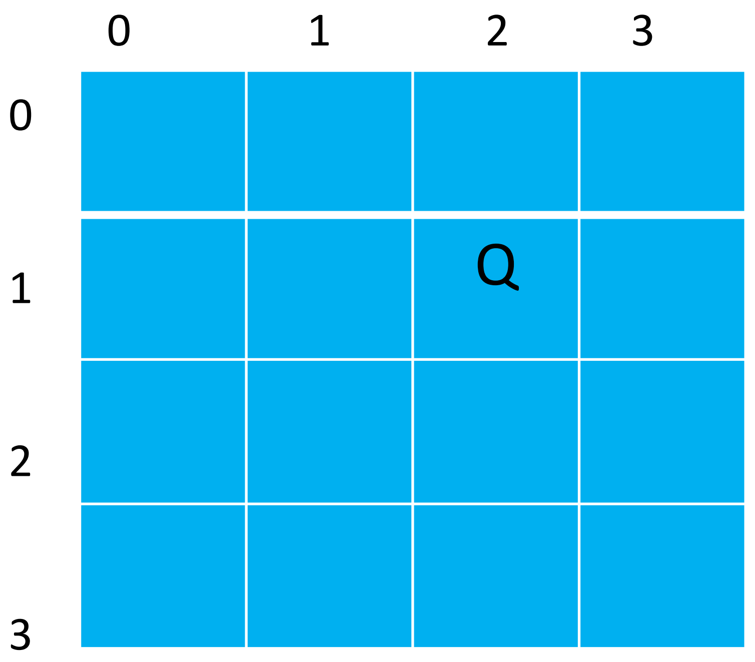
Constraint: Queens cannot be placed in the same row, column or diagonal.

For the given queen placement, following locations in the chess board will be under attack.

Row	**Col.**	**Dig.1**	**Diag.2**
(1,0)	(0, 2)	(0,1)	(3,0)
(1,1)	(2,2)	(2,3)	(2,1)
(1,3)	(3,2)		(0,3)

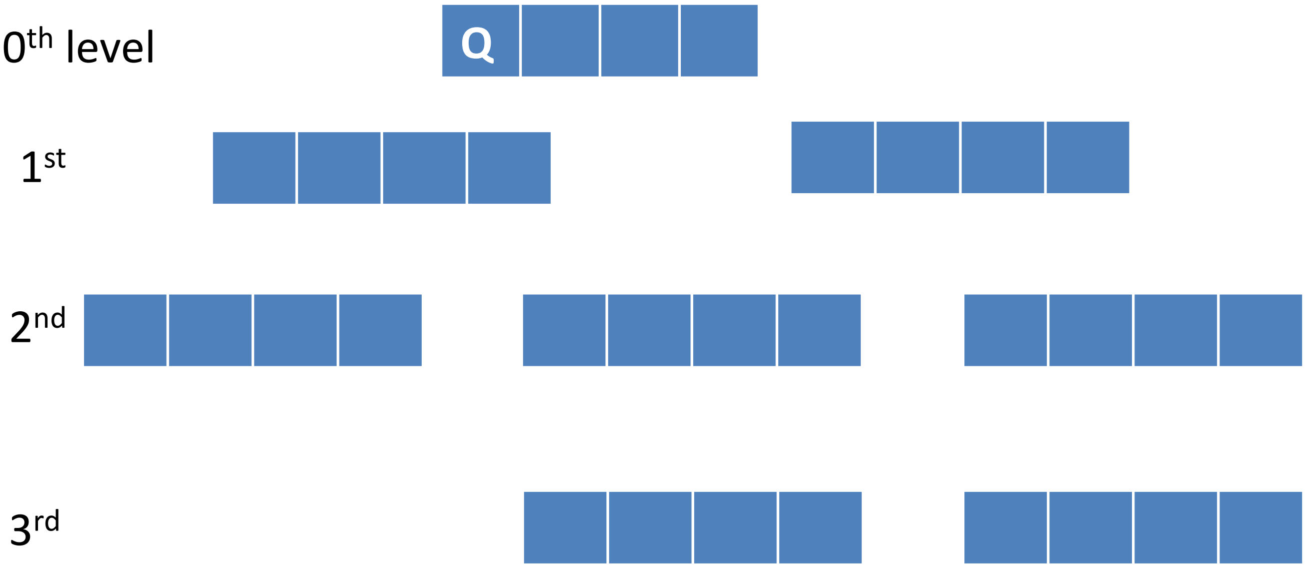
Dig. 1 attack positions = 0-1 = 2-3 => -1

Dig. 2 attack positions= 3+0= 2+1= 0+3



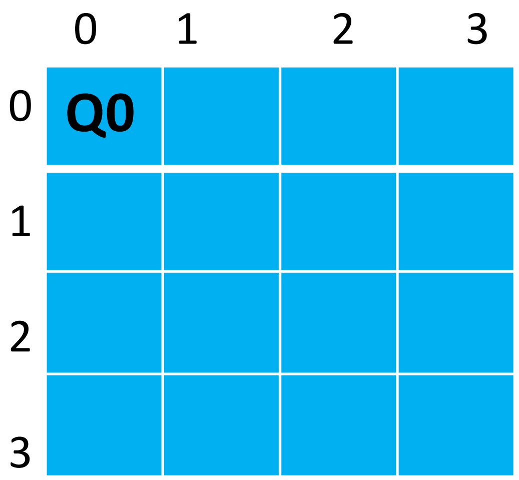
The N-Queens Problem

Place 0th queen at 0th column of 0th row.



Positions

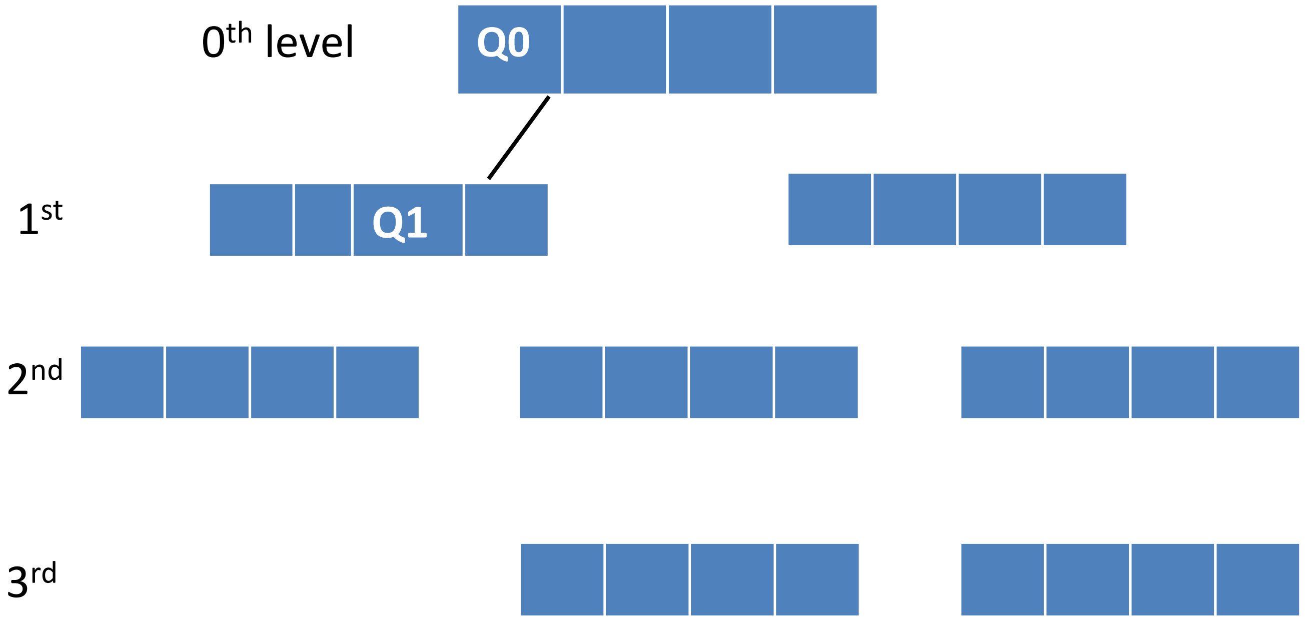
(0,0)



The N-Queens Problem

Call recursive function and place 1st queen

It will be placed at 2nd column of row 1, since other positions can be attacked by the Q0.



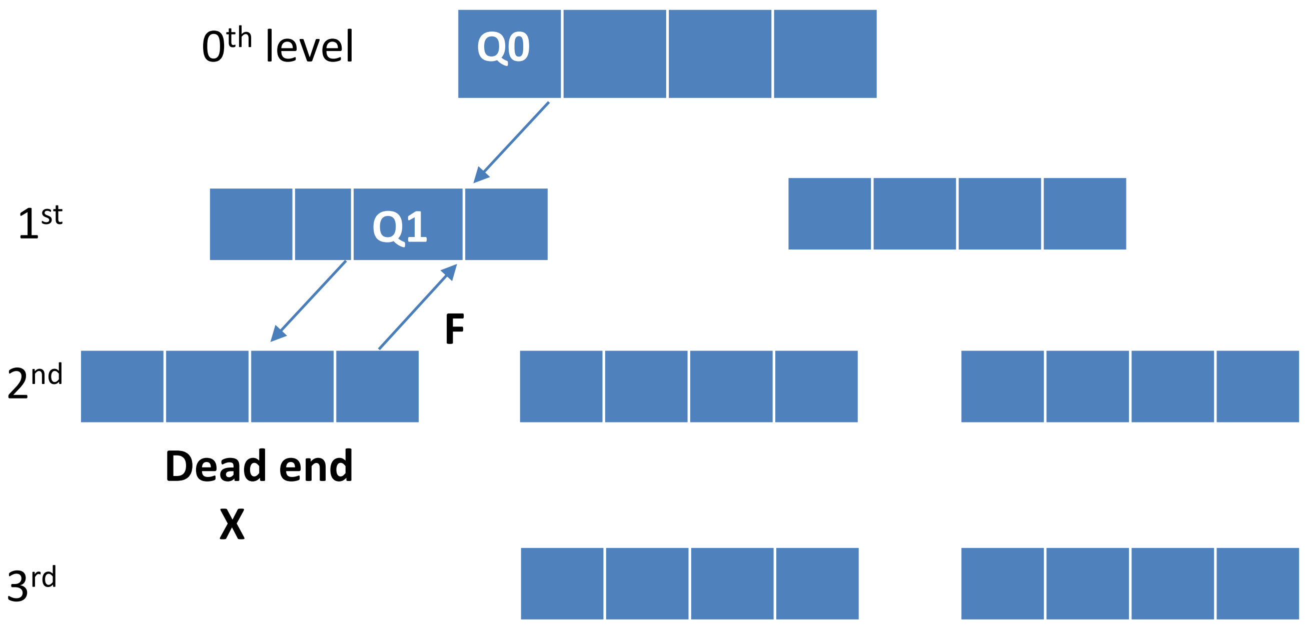
Positions
(0,0)
(1,2)

	0	1	2	3
0	**Q0**			
1			Q1	
2				
3				

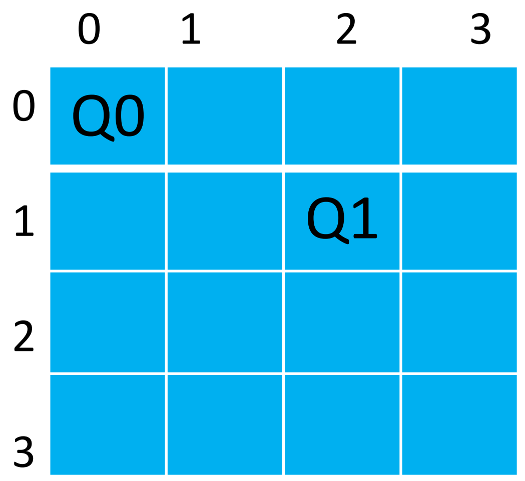
The N-Queens Problem

Call recursive function and place 2nd queen

It cannot be placed at any column of row 2, since all positions can be attacked by other queens.

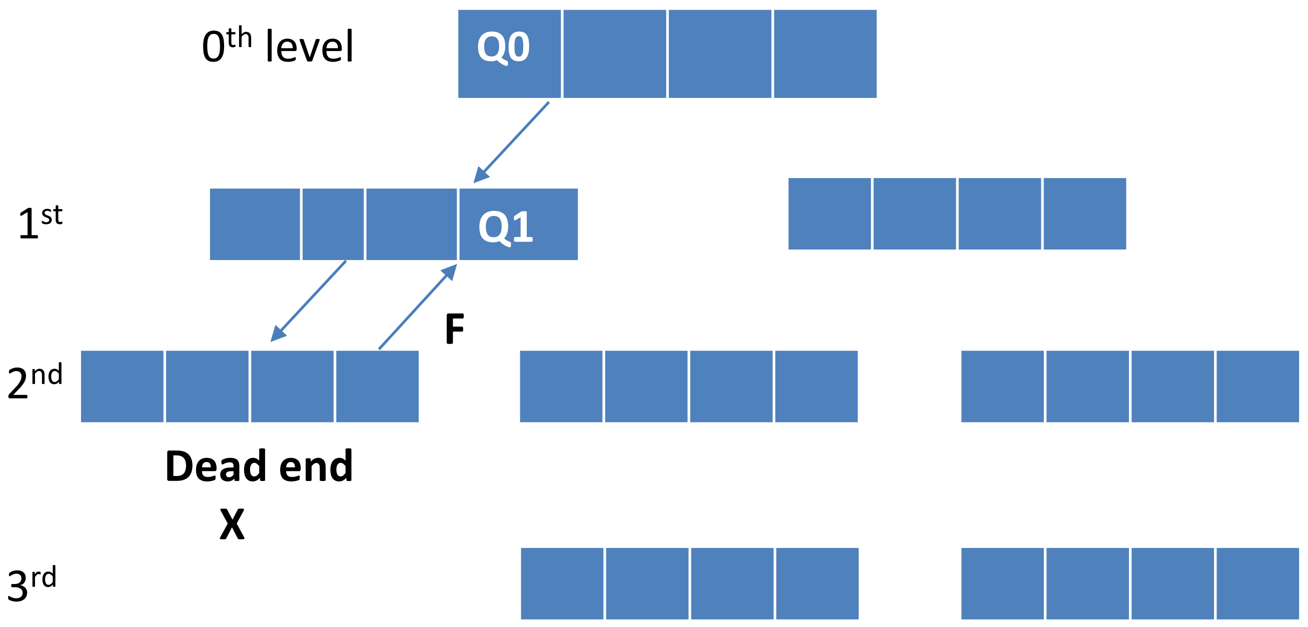


Positions
(0,0)
(1,2)

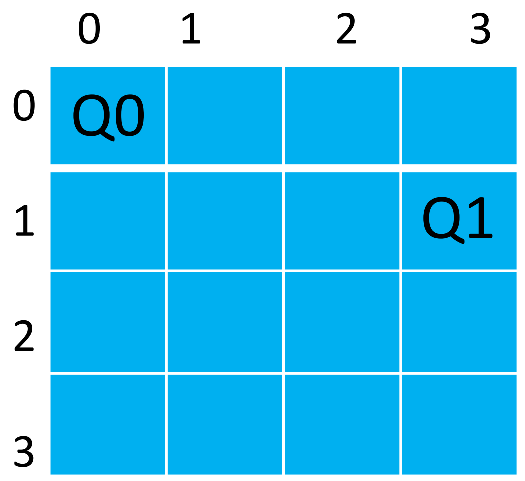


The N-Queens Problem

Return false to the calling function and shift queen at **level 1** to one position right.



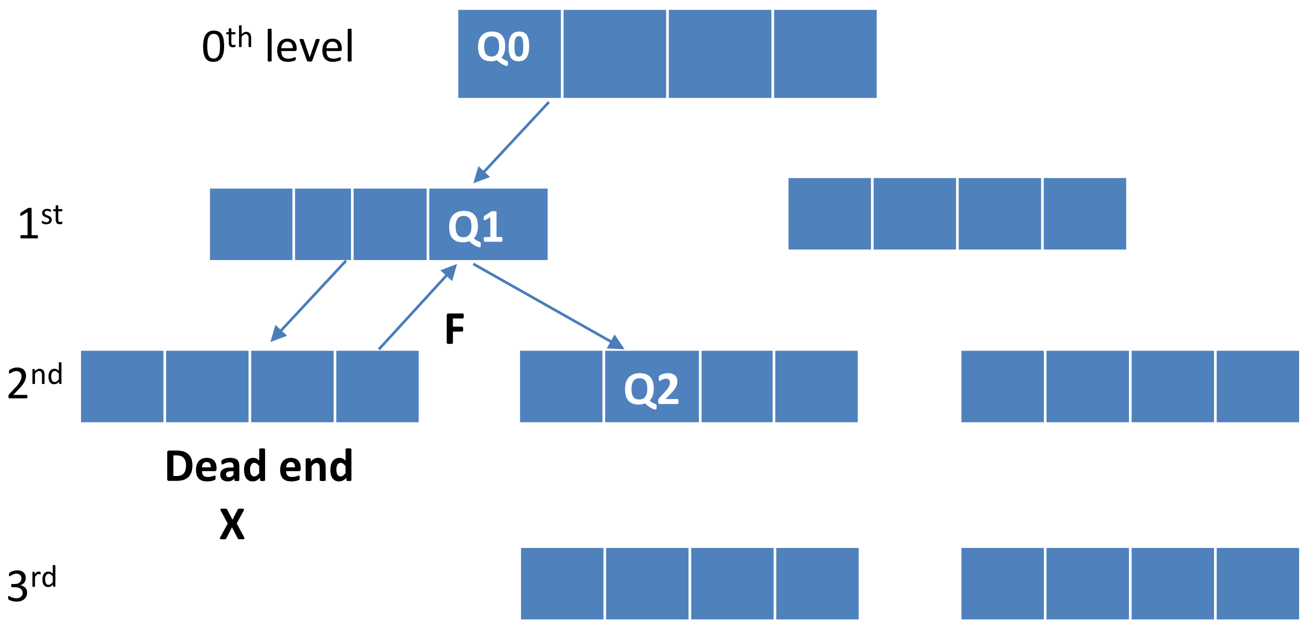
Positions
(0,0)
(1,3)



The N-Queens Problem

Call recursive function and place 2nd queen

It can be placed at 1st column of row 2.



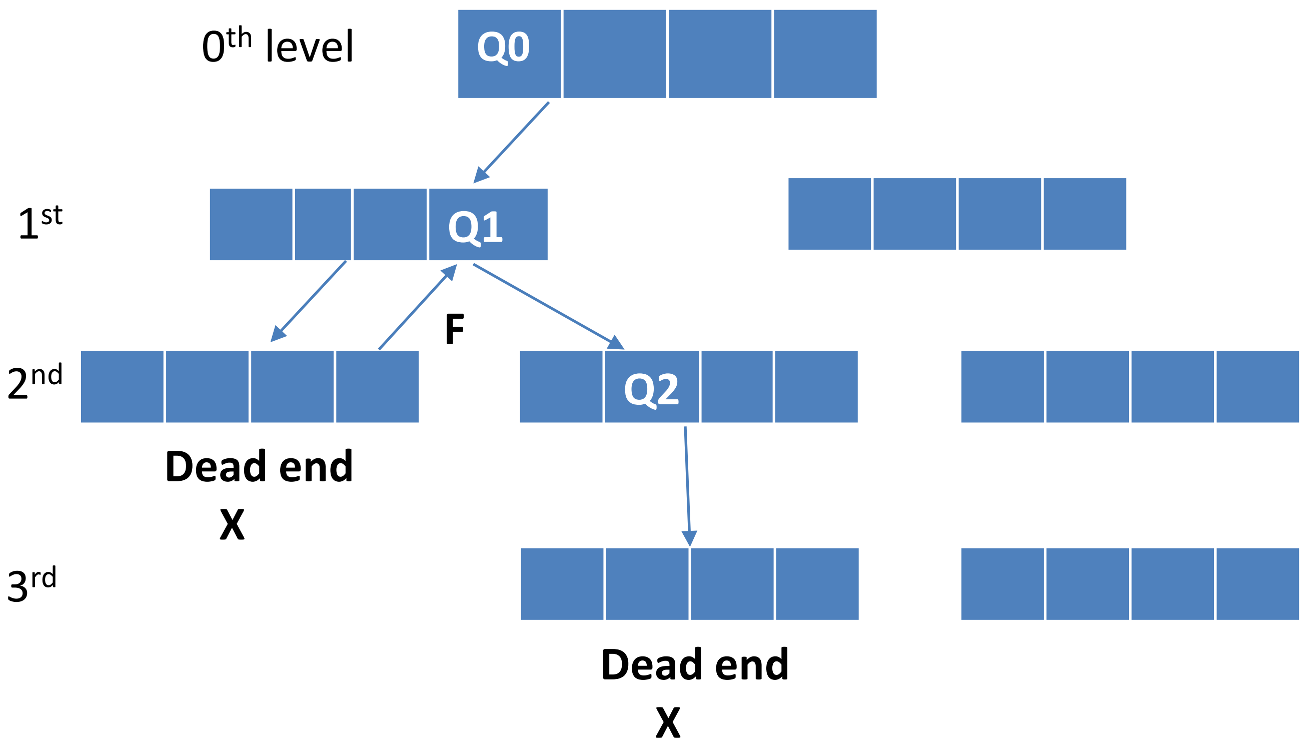
Positions
(0,0)
(1,3)
(2,1)

	0	1	2	3
0	Q0			
1				Q1
2		Q2		
3				

The N-Queens Problem

Call recursive function and place 3rd queen

It cannot be placed at any column of row 3, since all positions can be attacked by other queens.

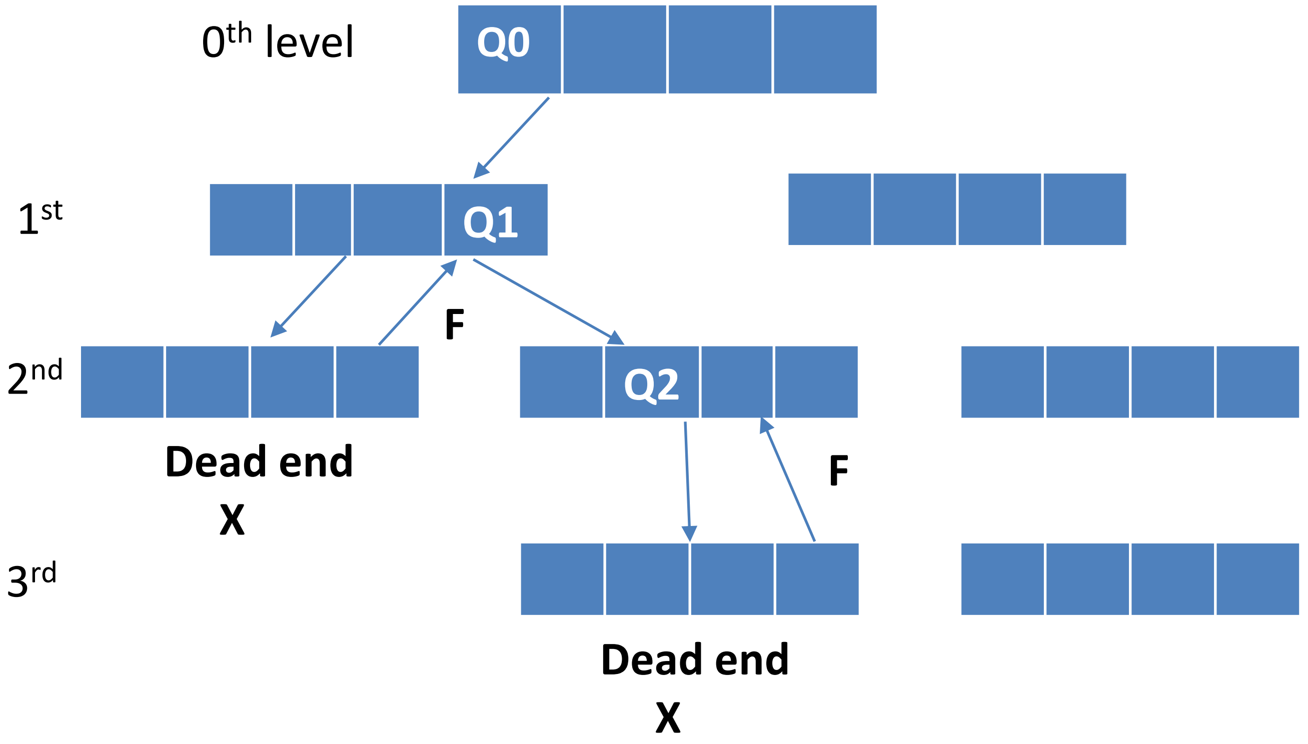


Positions
(0,0)
(1,3)
(2,1)

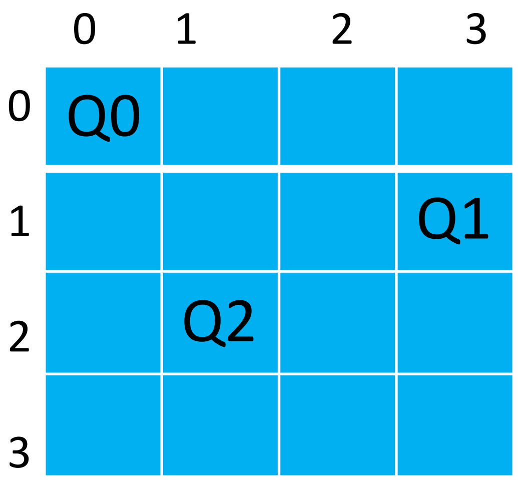
	0	1	2	3
0	Q0			
1				Q1
2		Q2		
3				

The N-Queens Problem

Return false to the calling function and shift **queen at level 2** to **one position right** and **check for the attack**. If **positions** can be **attack keep shifting** until **all positions are explored.**

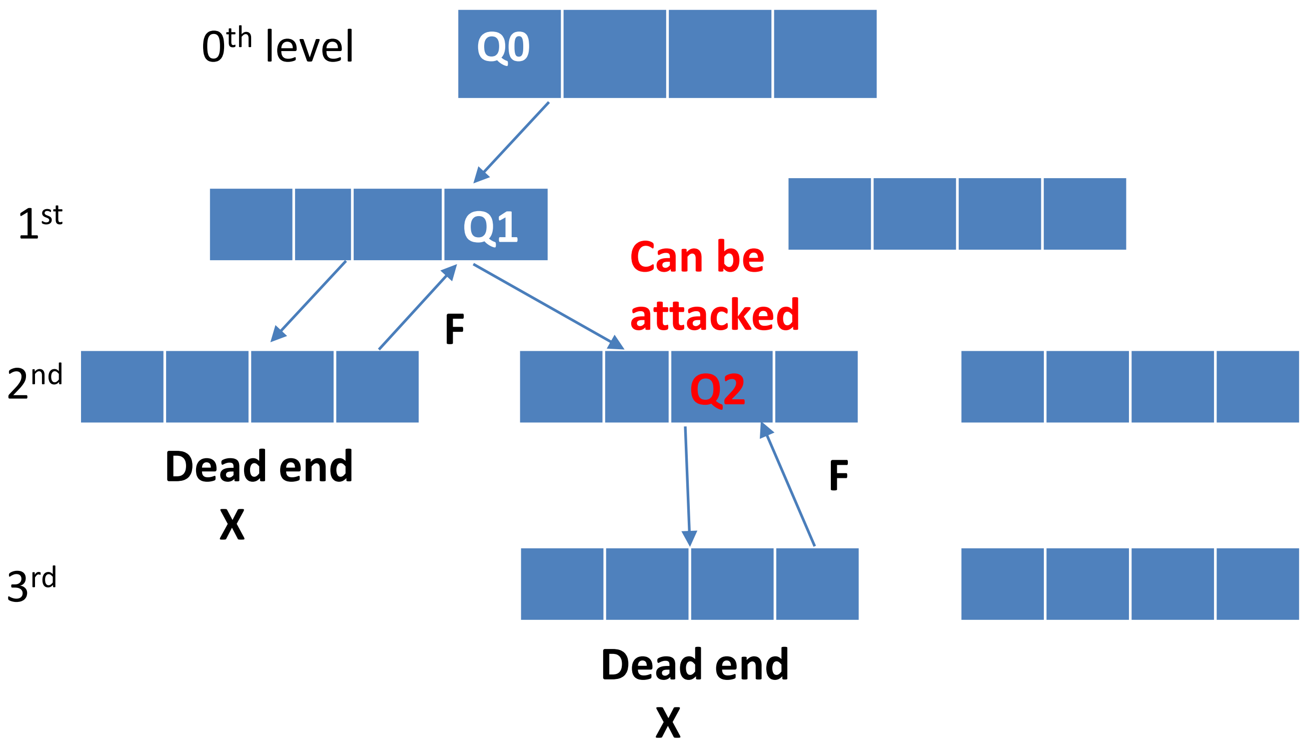


Positions
(0,0)
(1,3)
(2,1)

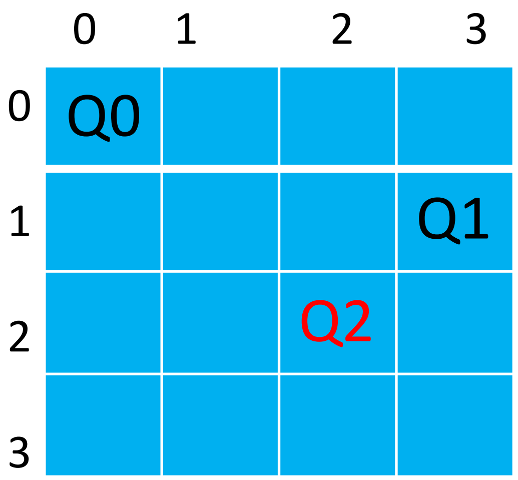


The N-Queens Problem

Shift **queen** at **level 2** to **one position right** and **check for the attack**. If **positions** can be **attack keep shifting** until **all positions** are **explored**.



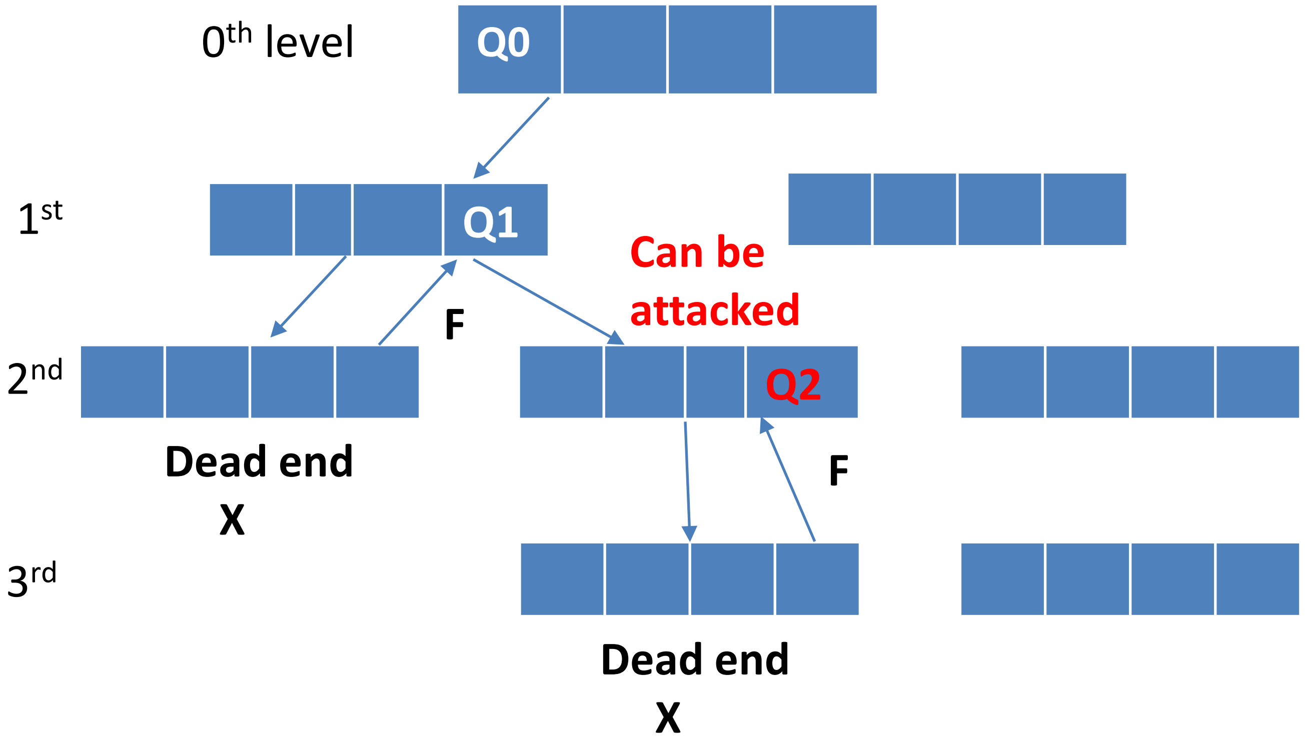
Positions
(0,0)
(1,3)
(2,2)



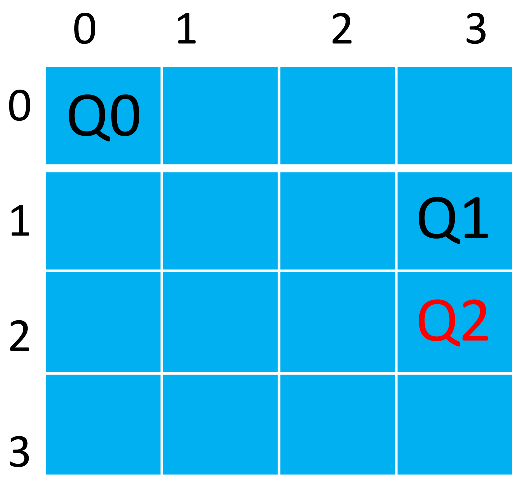
	0	1	2	3
0	Q0			
1				Q1
2			Q2	
3				

The N-Queens Problem

Shift **queen** at **level 2** to **one position right** and **check for the attack**. If **positions** can be **attack keep shifting** until **all positions** are **explored**.

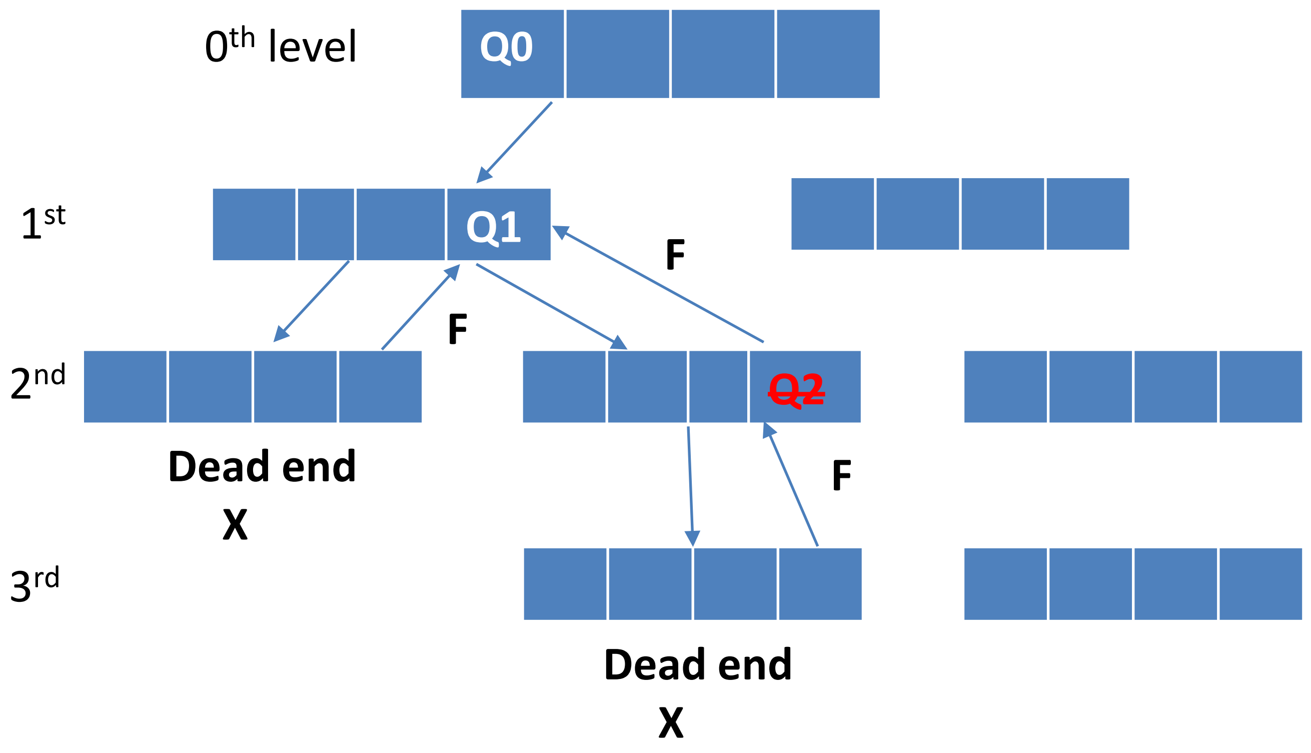


Positions
(0,0)
(1,3)
(2,3)



The N-Queens Problem

Since **all positions** at **level 2** can be **attacked** so **remove Q2** from **its position** and **return false** to the **calling function** (one level up).

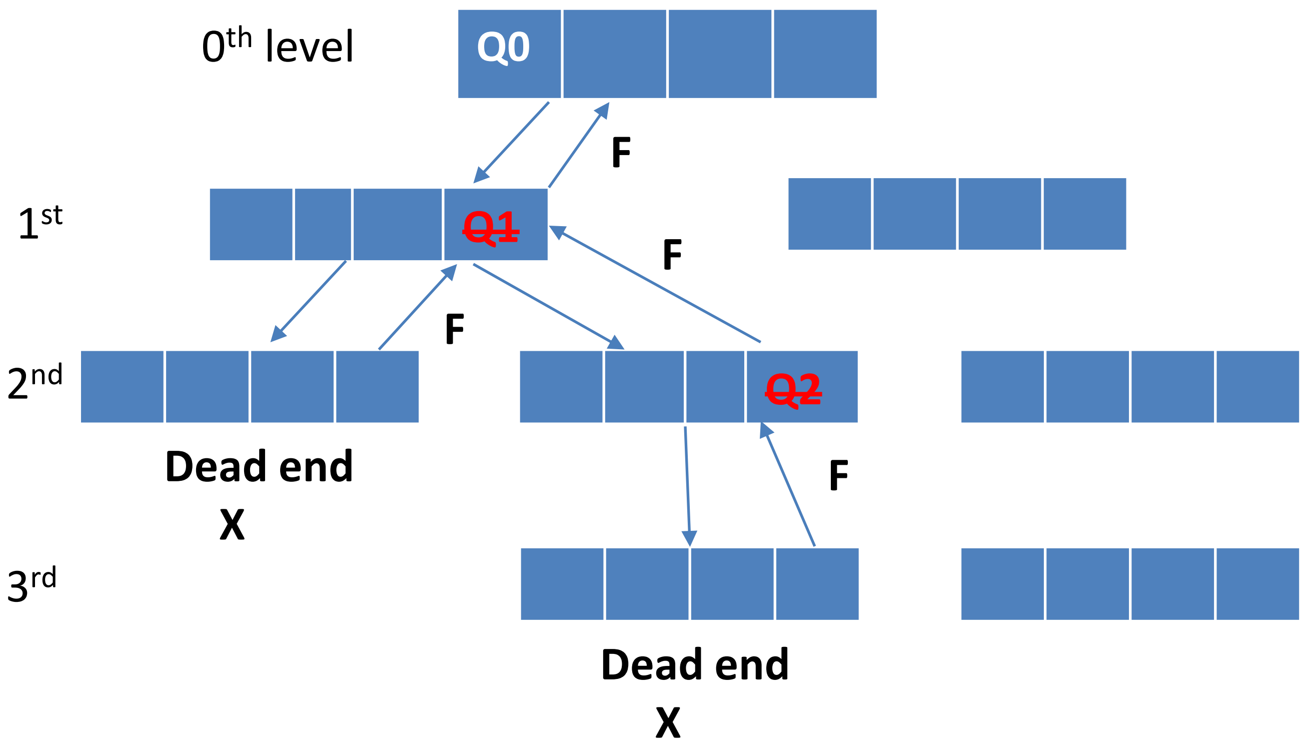


Positions
(0,0)
(1,3)
~~(2,3)~~

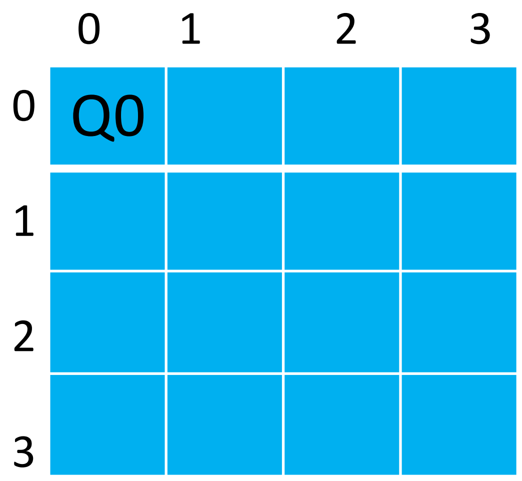
	0	1	2	3
0	Q0			
1				Q1
2				
3				

The N-Queens Problem

Shift queen at **level 1** to **one position right**, if **any block is available**, otherwise **return false to the calling function** (one level up).

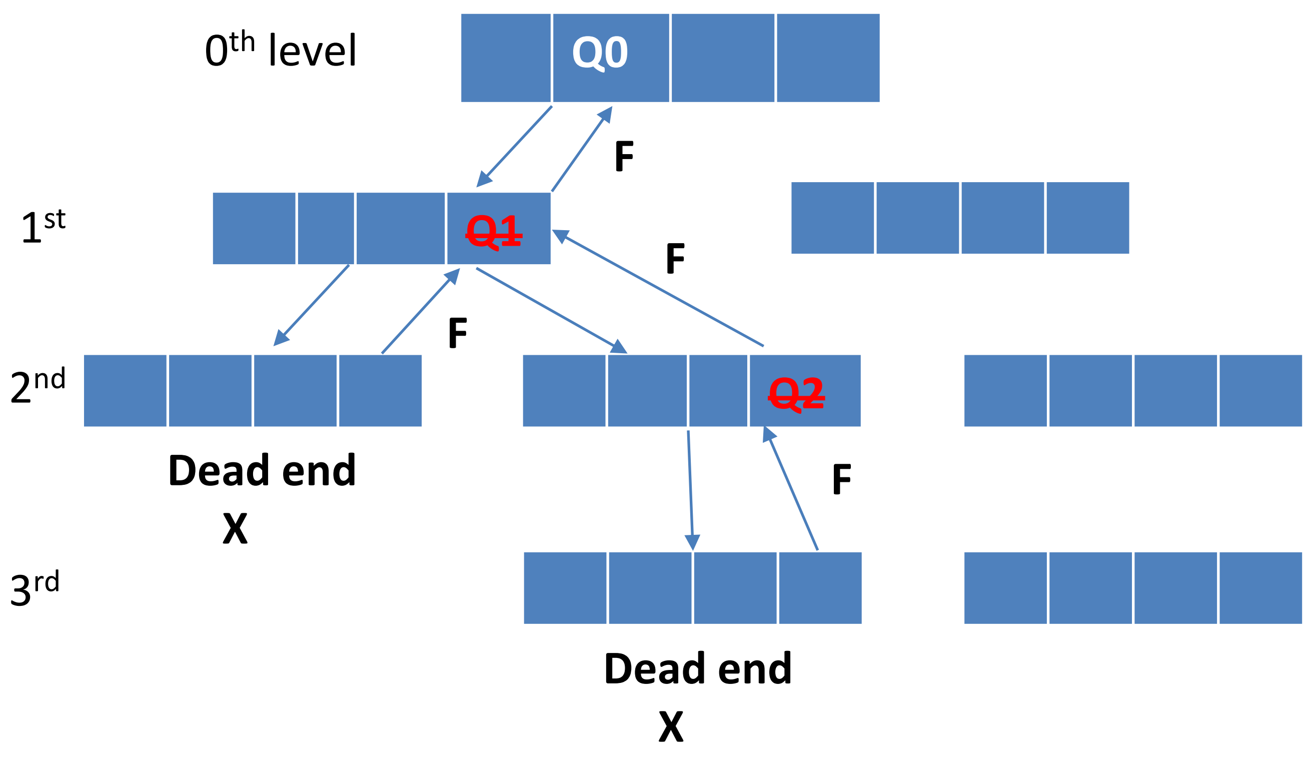


Positions
(0,0)
~~**(1,3)**~~
~~**(2,3)**~~

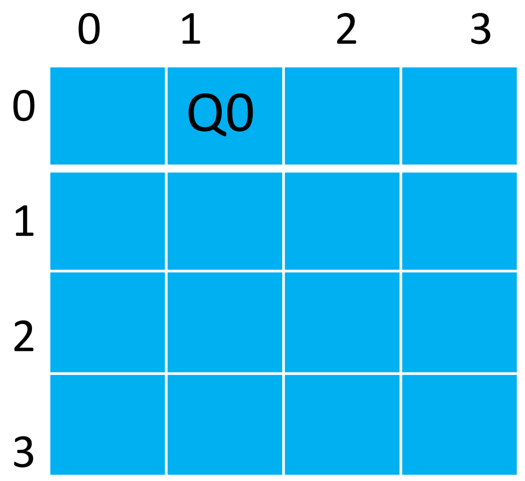


The N-Queens Problem

Shift **queen** at **level 0** to **one position right** and **update** the **position list**.



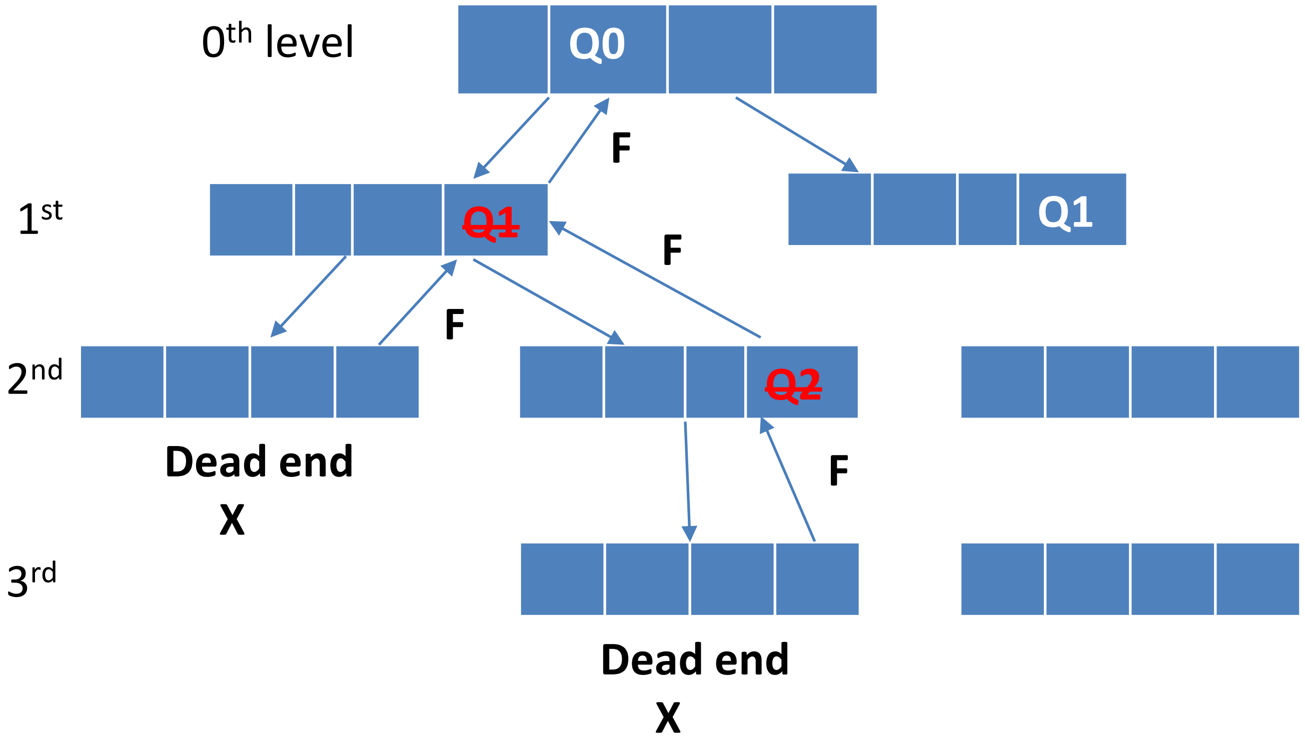
Positions
(0,1)
~~**(1,3)**~~
~~**(2,3)**~~



The N-Queens Problem

Call **recursive function and place 1st queen**

It will be placed at **3rd column** of **row 1**, since **other positions** can be **attacked** by the **Q0.**



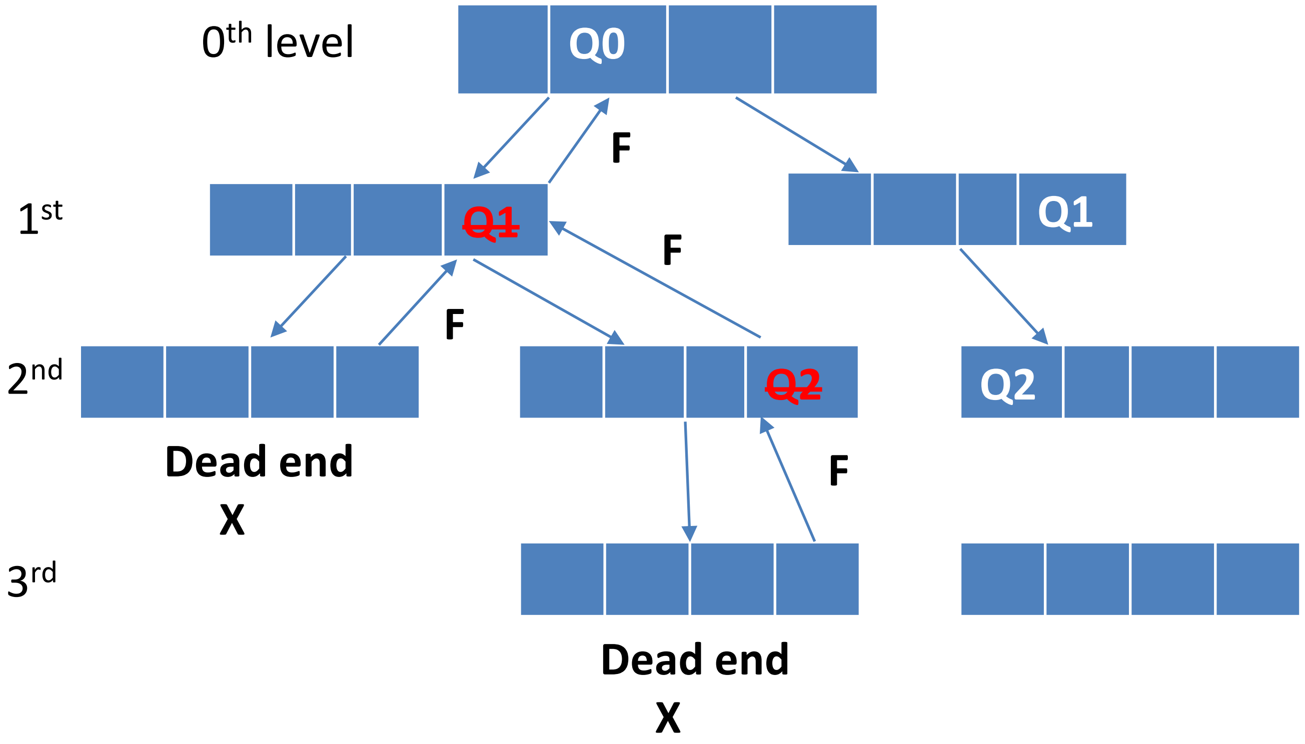
Positions
(0,1)
(1,3)

	0	1	2	3
0		Q0		
1				Q1
2				
3				

The N-Queens Problem

Call recursive function and place 2nd queen

It will be **placed** at **0th column** of **row 2**, since **other positions** can be **attacked** by the **Q0**.



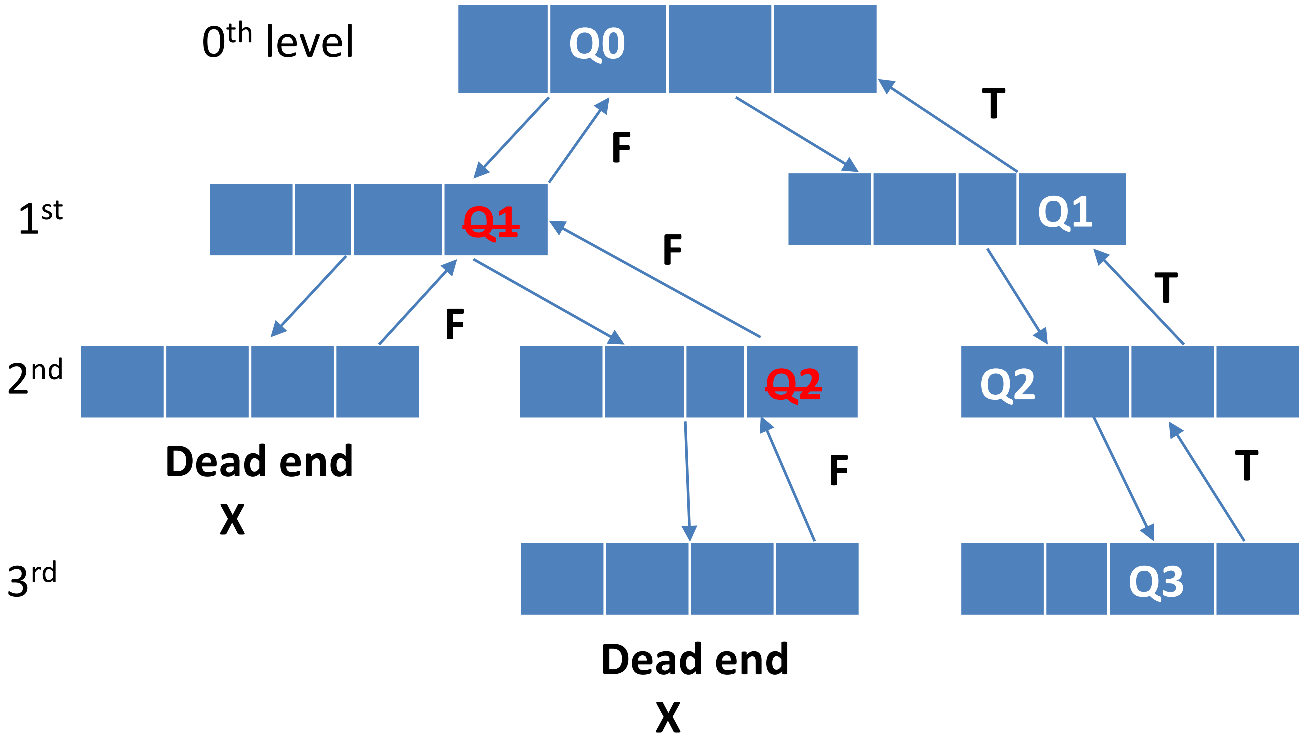
Positions
(0,1)
(1,3)
(2,0)

	0	1	2	3
0		Q0		
1				Q1
2	Q2			
3				

The N-Queens Problem

Call recursive function and **place 3rd queen**

It will be placed at **3rd column of row 3**, since **other positions** can be **attacked** by the **Q0.**



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

	0	1	2	3
0		Q0		
1				Q1
2	Q2			
3			Q3	

The N-Queens Problem

```
bool isSafe(int arr[][], int x, int y, int n)
{
    for (int row=0;row<x;row++)
    {
         if(arr[row][y]==1)
        {
            return false;
        }
    }
int row=x, int col=y;
while(row>=0 && col>=0)
   {
        if(arr[row][col]==1)
        {
        return false;
          }
      row--, col--;
  }
```

```
row=x, col=y;
while(row>=0 && col<n)
  {
if(arr[row][col]==1)
         {
         return false;
          }
         row--, col++;
  }
return true;
}
```

The N-Queens Problem

```
bool NQ(int arr[][], int row, int N)
{
        if (row> == n)
         return true;
for (int col = 0; col < N; col++)
    {
         if (isSafe(arr, row, col, N))
         {
             arr[row][col] = 1;
         if (NQ(arr, row + 1, N)){
                  return true;
                 }
        arr[row][col] = 0; //backtracking
         }
    }
    return false;
}
```

The N-Queens Problem

Time Complexity : O(N!), Since we have N choices in the first row, then N-1 choices in the second row and so on so the overall complexity become **O(N!)**

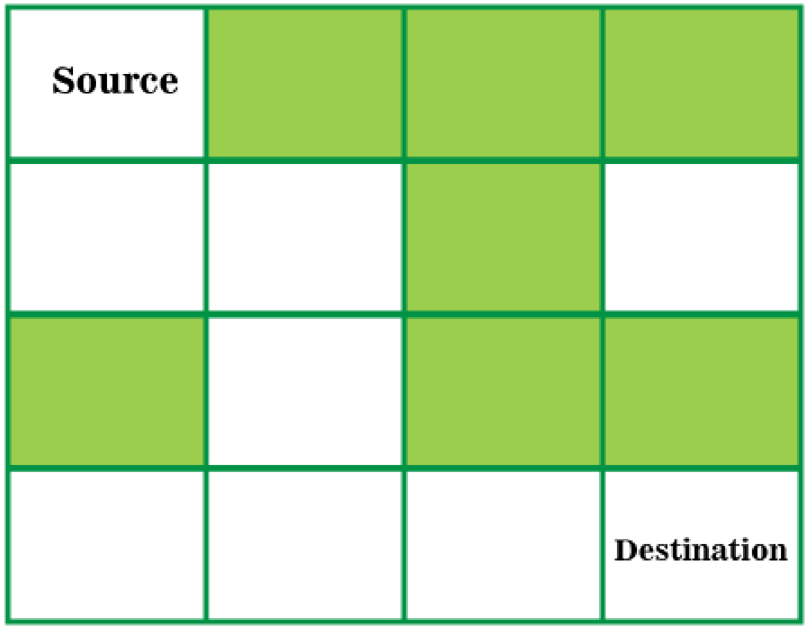
Space Complexity: O(N*N)

Rat in a Maze problem

Problem: Given a maze[][] of n*n matrix, a rat has to find path from source to destination. The top corner maze [0][0] is the source, and the right bottom corner [n-1][n-1] is the destination. The rat can move in two directions- right and down. In the maze, 1 represent free space and 0 means it is blocked/ dead end.

Use the concept of backtracking and print the path from (0,0) to (n-1, n-1) if exists else print -1;

Rat in a Maze problem



Maze

The following is the binary representation of the mentioned maze.

1.{1, 0, 0, 0}

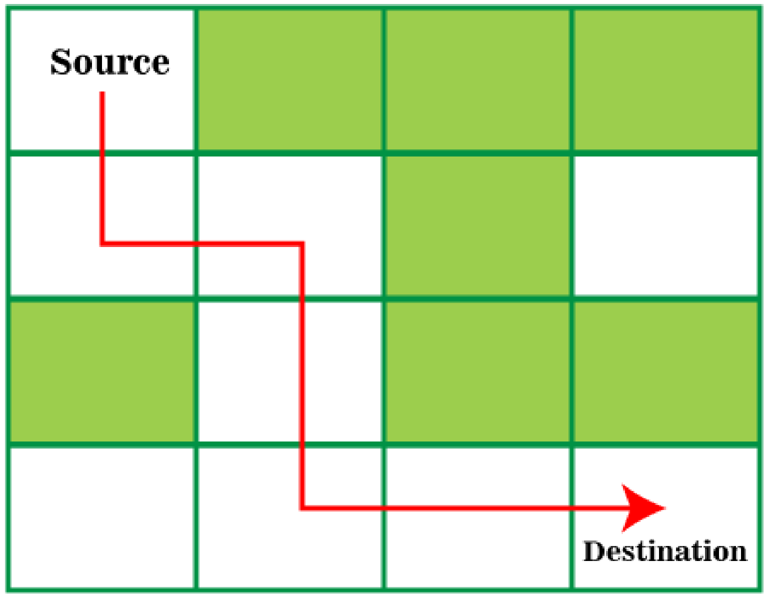
2.{1, 1, 0, 1}

3.{0, 1, 0, 0}

4.{1, 1, 1, 1}

Here, 1 means that the cell is open, and the rat can enter. 0 means the cell is blocked, and the rat cannot enter.

Rat in a Maze problem



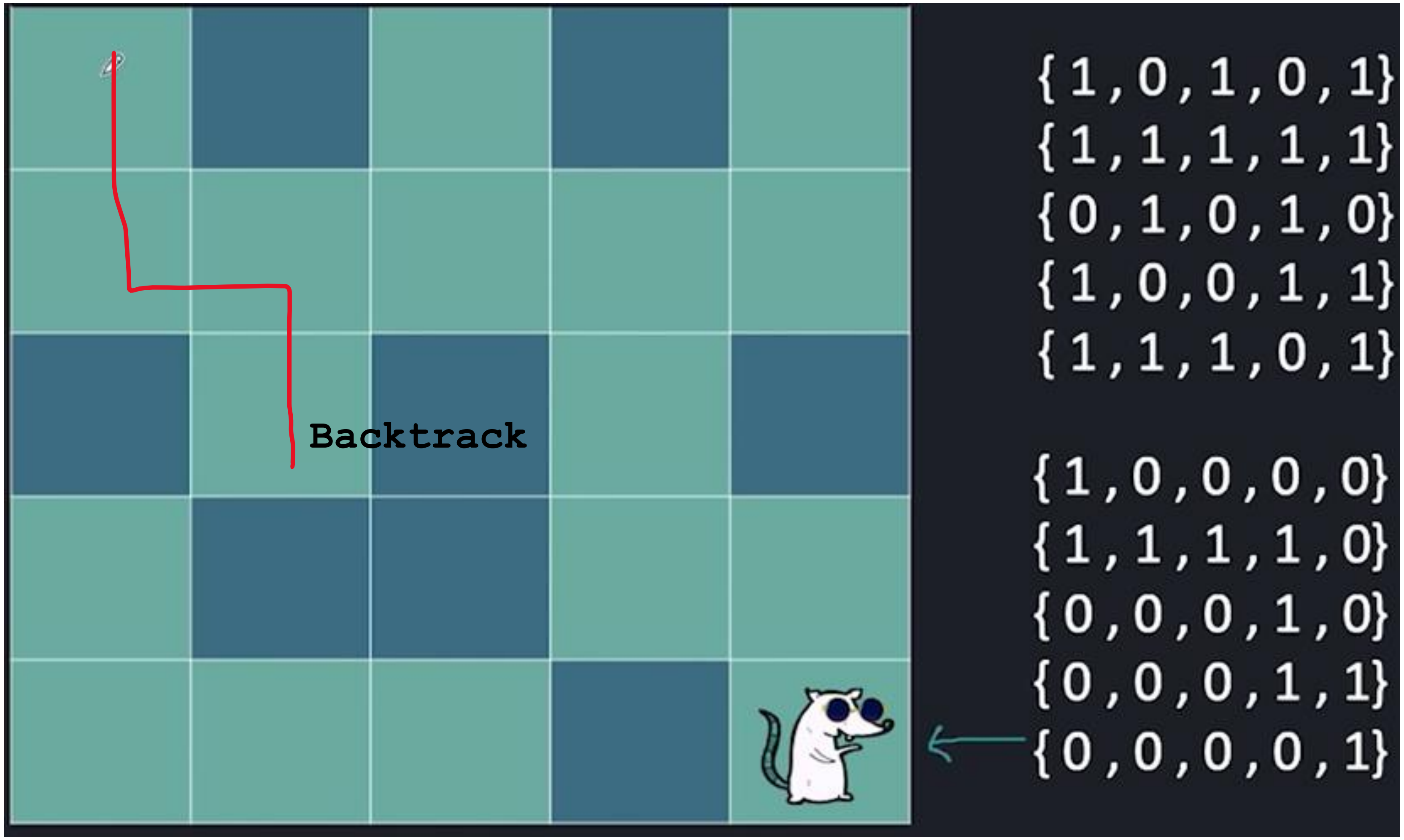
Maze

The following is the binary representation of the path shown in the above diagram.

1.{1, 0, 0, 0}
2.{1, 1, 0, 0}
3.{0, 1, 0, 0}
4.{0, 1, 1, 1}

Only the entries marked using the number 1 represent the path.

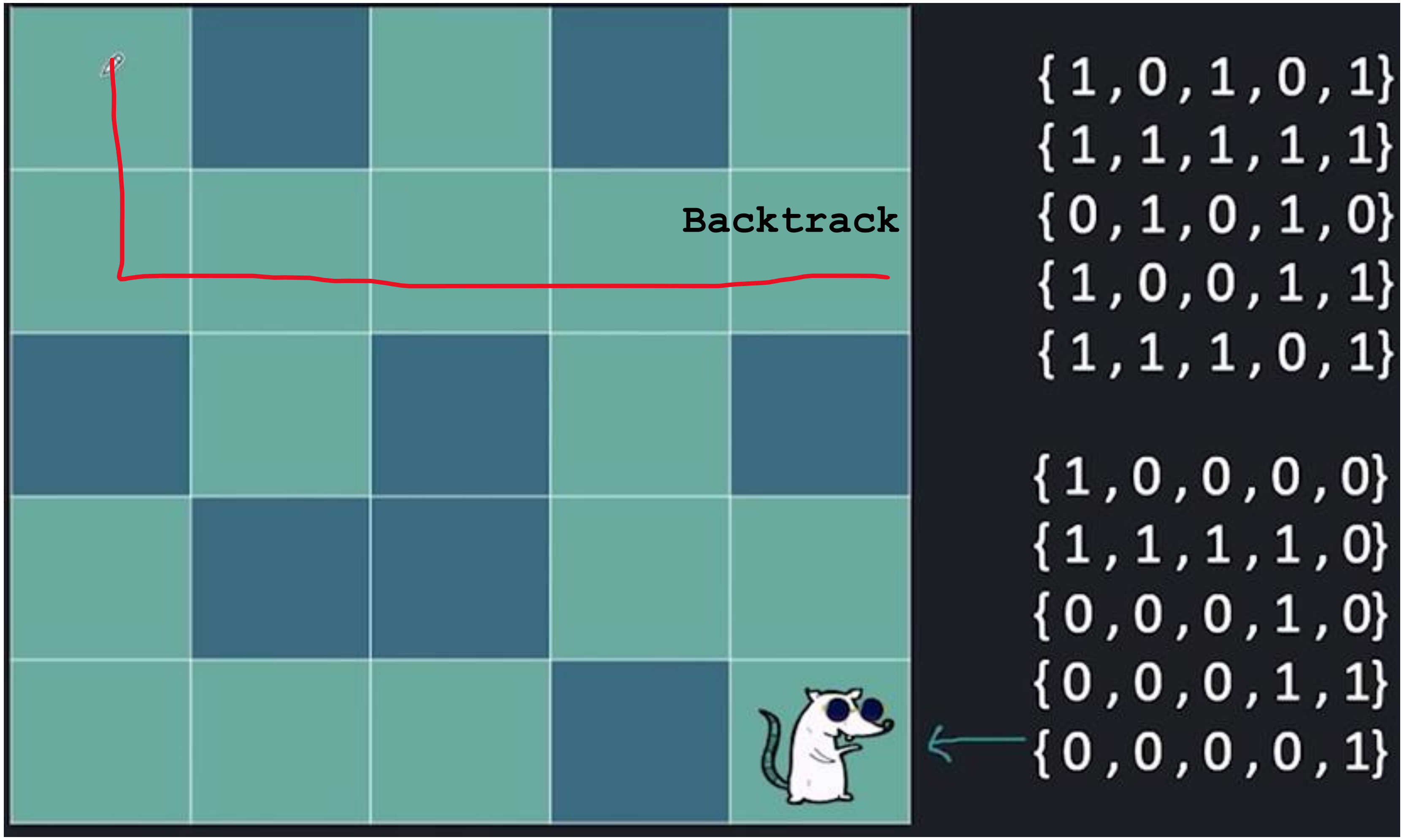
Rat in a Maze problem



```
{1,0,1,0,1}
{1,1,1,1,1}
{0,1,0,1,0}
{1,0,0,1,1}
{1,1,1,0,1}

{1,0,0,0,0}
{1,1,1,1,0}
{0,0,0,1,0}
{0,0,0,1,1}
{0,0,0,0,1}
```

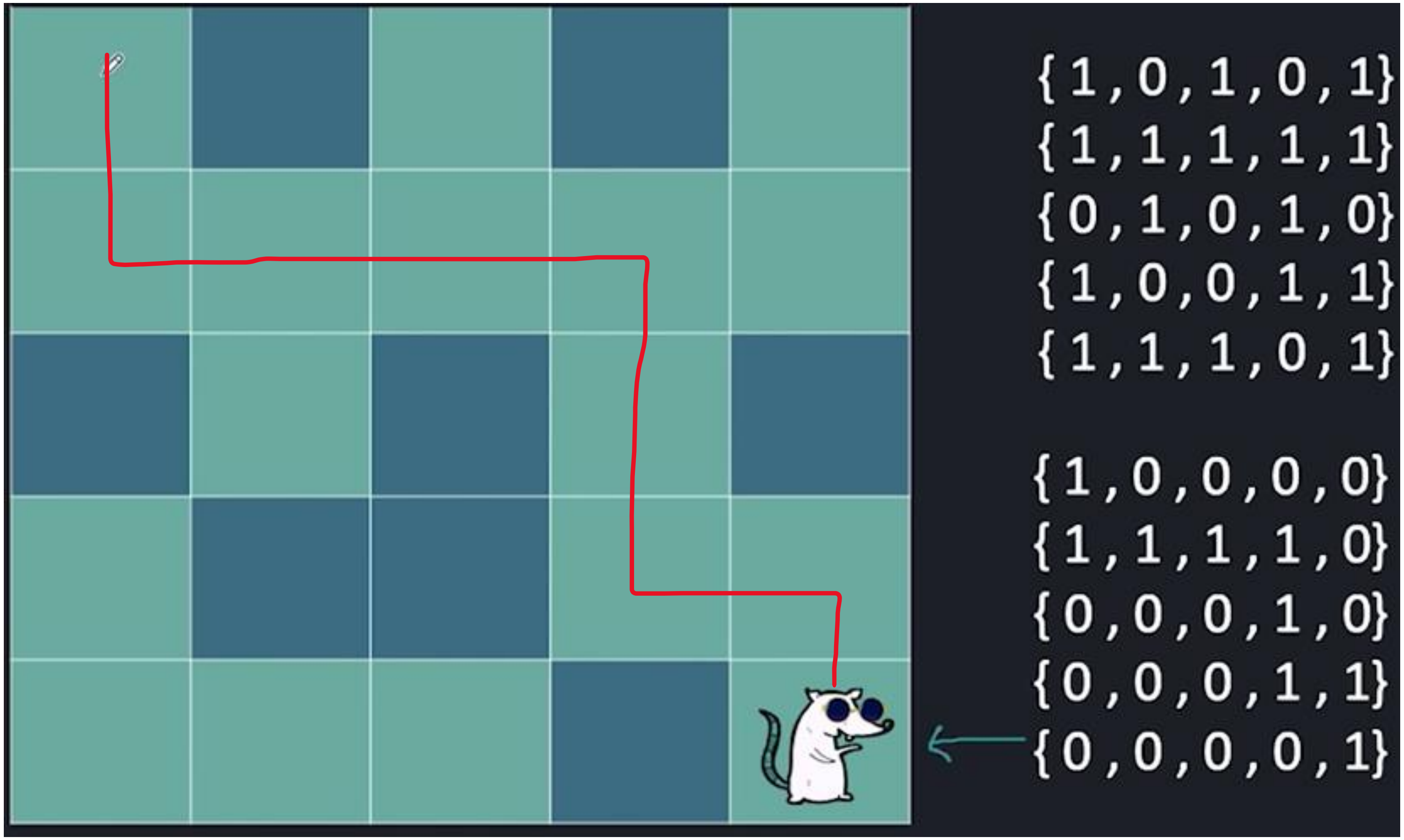
Rat in a Maze problem



{1,0,1,0,1}
{1,1,1,1,1}
{0,1,0,1,0}
{1,0,0,1,1}
{1,1,1,0,1}

{1,0,0,0,0}
{1,1,1,1,0}
{0,0,0,1,0}
{0,0,0,1,1}
{0,0,0,0,1}

Rat in a Maze problem



```
{1,0,1,0,1}
{1,1,1,1,1}
{0,1,0,1,0}
{1,0,0,1,1}
{1,1,1,0,1}

{1,0,0,0,0}
{1,1,1,1,0}
{0,0,0,1,0}
{0,0,0,1,1}
{0,0,0,0,1}
```

Reached to destination

Rat in a Maze problem

```
bool isSafe(int arr[][], int x, y, int n) {
        if(x<n && y<n && arr[x][y]==1){
                return true;
            }
        return false;
}
bool ratinMaze(int arr[][], int x, int y, int n int sol[][]){
        if(x==n-1 && y==n-1){
                 sol[x][y]=1;
                 return true;  }

if(isSafe(arr, x, y, n) {
             sol[x][y]=1;
        if(ratinMaze(arr, x+1, y, n, sol)){
                return true;}
        if(ratinMaze(arr, x, y+1, n, sol)){
                return true; }
        sol[x][y]=0;  // Backtracking
        return false;
      }
return false;
}
```

`ratinMaze(arr, 0,0, n, sol)`

The N-Queens Problem

Time Complexity-

The number of moves possible for rat is two. The algorithm can go down and go right and then try out every possibility to fill the lower square. Time complexity for the backtracking solution = *$O(2^{n\text{^}2})$* because we need to consider two different paths at every position.

Space Complexity – $O(n^2)$, since to keep the input and solution, 2-D array is used.