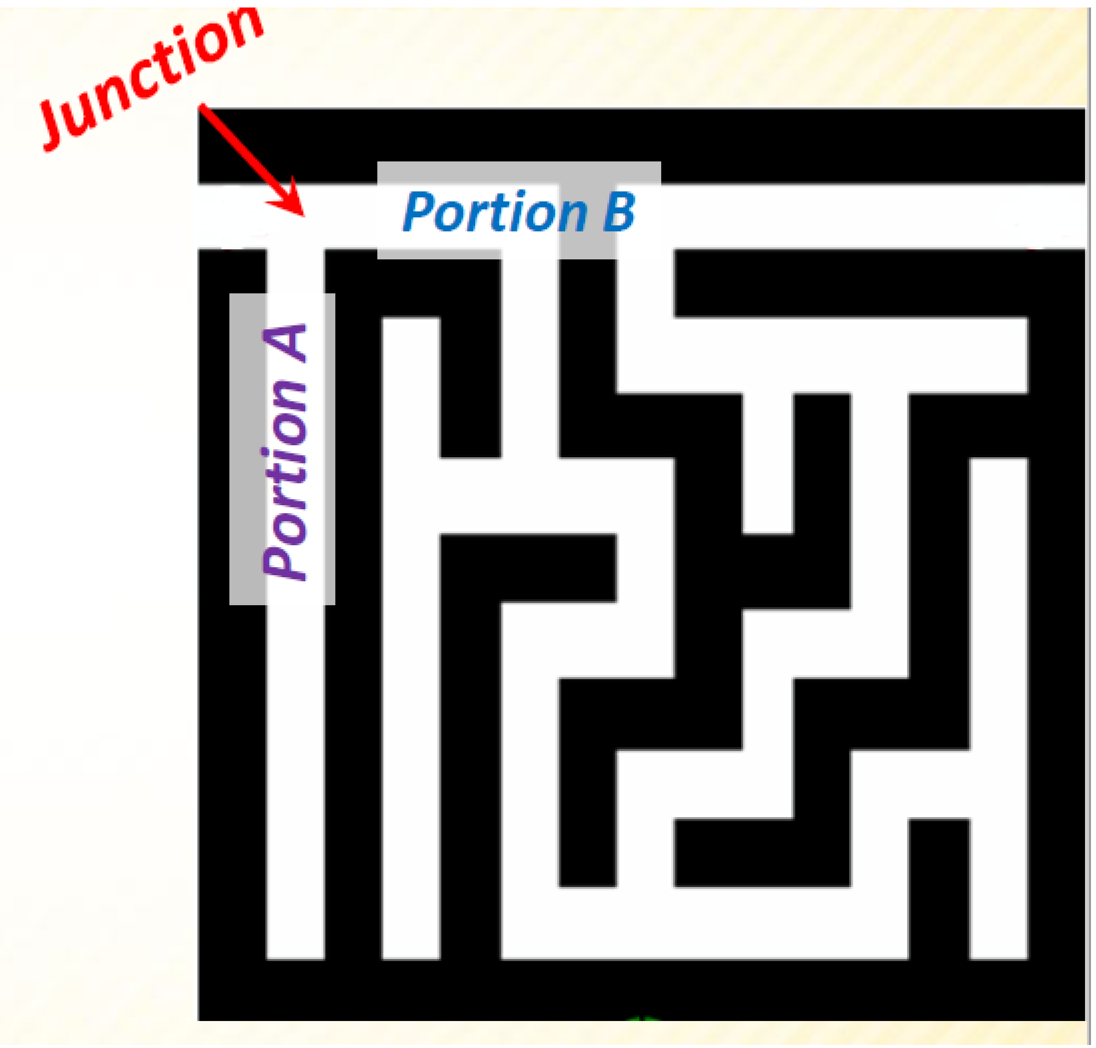
Backtracking

Backtracking

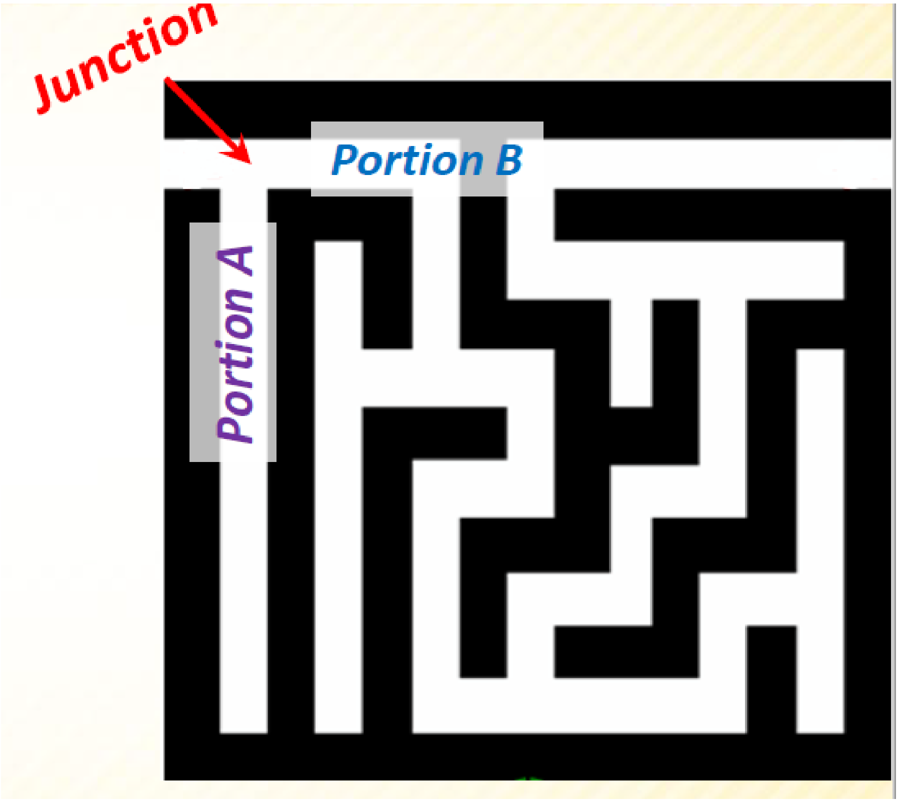
- Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.
- A standard example of backtracking would be going through a maze.
  - At some point in a maze, you might have two options of which direction to go:



Backtracking

- One strategy would be to try going through **Portion A** of the maze.
  - If you get stuck before you find your way out, then you ***"backtrack"*** to the junction.

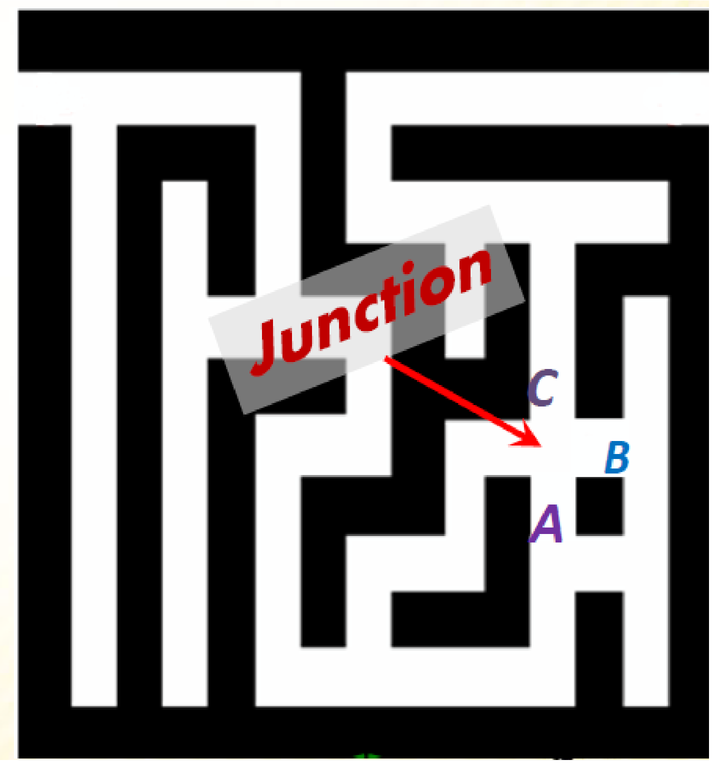
- At this point in time you know that **Portion A** will ***NOT*** lead you out of the maze,
  - so you then start searching in **Portion B**



BackTracking

- Clearly, at a single junction you could have even more than 2 choices.
- The backtracking strategy says to try each choice, one after the other,
  - if you ever get stuck, ***"backtrack"*** to the junction and try the next choice.
- If you try all choices and never found a way out, then there IS no solution to the maze.

- Well now, can you think how can you represent a maze in your program?
  - How can you create obstacles in the path?



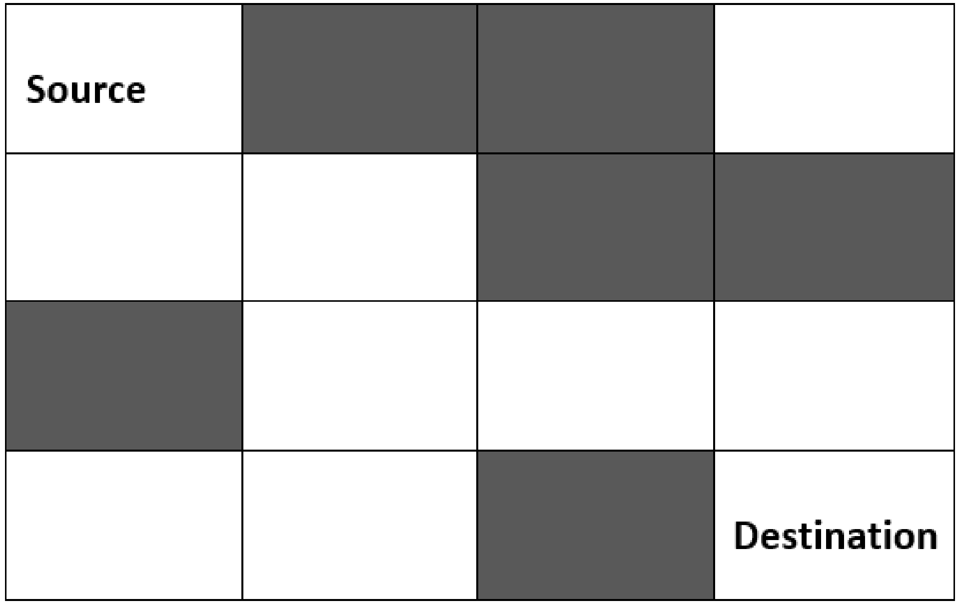
Rat in a Maze (Simple version)

- A Maze is given as N*N binary matrix of blocks where,
  - source 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 the destination. The rat can move only in two directions:
  - forward and down.
- In the maze matrix,
  - 0 means the block is a dead end
  - and 1 means the block can be used in the path from source to destination.
- Note that this is a simple version of the typical Maze problem.
  - For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Rat in a Maze

- Following is an example of maze
- Gray block are dead ends (value = 0)
- White blocks are accessible (value = 1)
- So, the matrix representation of this maze is like this:

```
M[4][4]=
{ { 1, 0, 0, 1 },
  { 1, 1, 0, 0 },
  { 0, 1, 1, 1 },
 { 1, 1, 0, 1 }
};
```

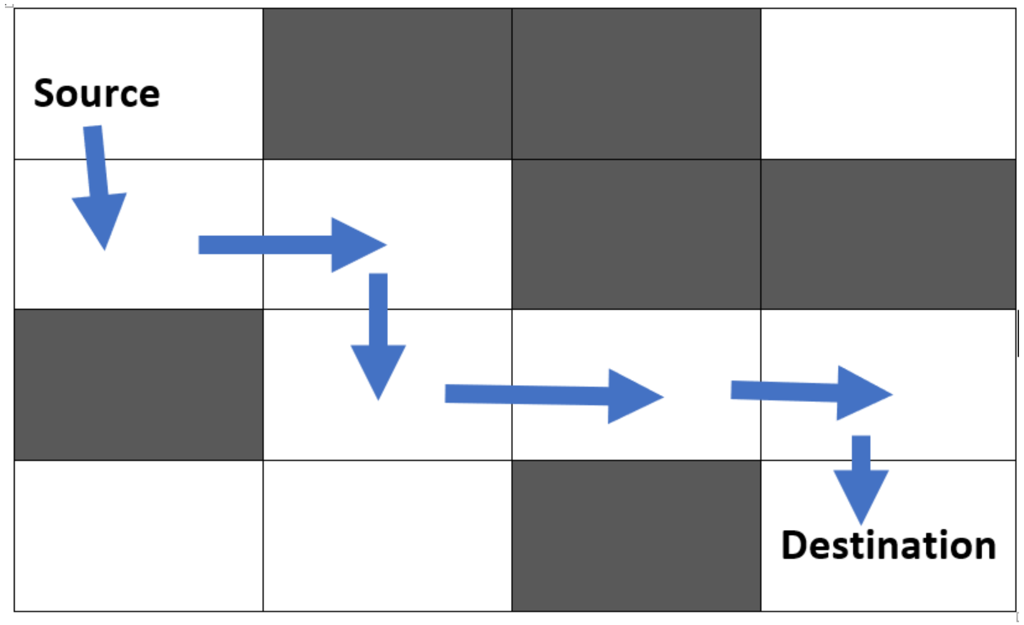


Rat in a Maze

- Following is a maze with highlighted solution path.
- The output solution matrix of this maze should look like this:

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

- All entries in solution path are marked as 1.



Rat in a Maze

- Naïve Algorithm
  - The Naïve Algorithm is to generate all paths from source to destination and one by one check if the generated path satisfies the constraints.

```
while there are untried paths
{
   generate the next path
   if this path has all blocks as 1
   {
      print this path;
   }
}
```

Rat in a Maze

- Walking through matrix

- Can you tell me if you are at maze[i][j], how can you go to the right, left, down, and up side?
  - Right: Maze[i][j+1]
  - Left: Maze[i][j-1]
  - Down: Maze[i+1][j]
  - Up: Maze[i-1][j]
- Do, you need to check anything before writing such statement to move to a particular direction?
  - Yes, you should check that you are going out of the matrix or not
- But note: We are mainly considering only 2 direction movement in this example. [forward and down]

	column 0	column 1	column 2	column 3
row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Let's say, i=1, j=1

Rat in a Maze

- Backtracking Algorithm

If destination is reached
    print the solution matrix
Else
   a) Mark current cell in solution matrix as 1.
   b) Move forward in the horizontal direction and recursively check if this move leads to a solution.
   c) 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.
   d) If none of the above solutions works then unmark this cell as 0 (BACKTRACK) and return false.

Implementing Rat in a Maze

- The main, Solve maze, and print function

```
int 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) == 0) {
    printf("Solution doesn't exist");
    return 0;
  }

  printSolution(sol);
  return 1;
}
```

```
// driver program to test above function
int main()
{
  int maze[N][N] = {
              { 1, 0, 0, 1 },
              { 1, 1, 0, 0 },
              { 0, 1, 1, 1 },
              { 1, 1, 0, 1 }
            };

  solveMaze(maze);
  return 0;
}
```

```
/* A utility function to print solution matrix
sol[N][N] */
void printSolution(int sol[N][N])
{
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++)
      printf(" %d ", sol[i][j]);

     printf("\n");
  }
}
```

```c
/* A recursive utility function to solve Maze problem */
int 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 1;
  }
  // Check if maze[x][y] is valid
  if (isSafe(maze, x, y) == 1) {
    // mark x, y as part of solution path
    sol[x][y] = 1;

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

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

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

    return 0;
  }
  return 0;
}
```

```c
/* A utility function to check if x, y is valid index for
N*N maze */
int isSafe(int maze[N][N], int x, int y)
{
  // if (x, y outside maze) return false
  if (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)
    return 1;

  return 0;
}
```

We will draw a recursion tree to solve this in the class

Output

```
1 0 0 0
1 1 0 0
0 1 1 1
0 0 0 1
```

Our maze was this

```c
int maze[N][N] = {
              { 1, 0, 0, 1 },
              { 1, 1, 0, 0 },
              { 0, 1, 1, 1 },
              { 1, 1, 0, 1 }
          };
```

Confused about these recursions? I have added three printf that should give you idea which recursion is being called

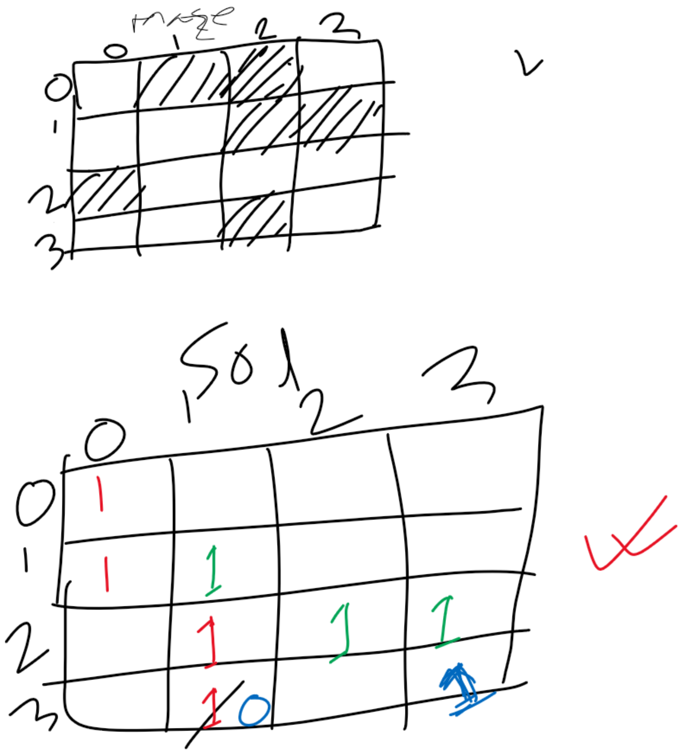
```c
/* A recursive utility function to solve Maze problem */
int solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])
{
  // if (x, y is goal) return true
  if (x == N - 1 && y == N - 1) {
    sol[x][y] = 1;
    return 1;
  }
  // Check if maze[x][y] is valid
  if (isSafe(maze, x, y) == 1) {
    // mark x, y as part of solution path
    sol[x][y] = 1;
    printf("Trace: before x s(%d, %d) mat = %d\n", x+1, y, sol[x][y]);
    /* Move forward in x direction (next row) */
    if (solveMazeUtil(maze, x + 1, y, sol) == 1)
      return 1;

    /* If moving in x direction doesn't give solution then
    Move down in y direction */
    printf("Trace: before y s(%d, %d) mat = %d\n", x, y+1, sol[x][y]);
    if (solveMazeUtil(maze, x, y + 1, sol) == 1)
      return 1;

    /* If none of the above movements work then BACKTRACK:
      unmark x, y as part of solution path */
    sol[x][y] = 0;
    printf("Trace: after x,y s(%d, %d) mat = %d\n", x, y, sol[x][y]);

    return 0;
  }
  return 0;
}
```

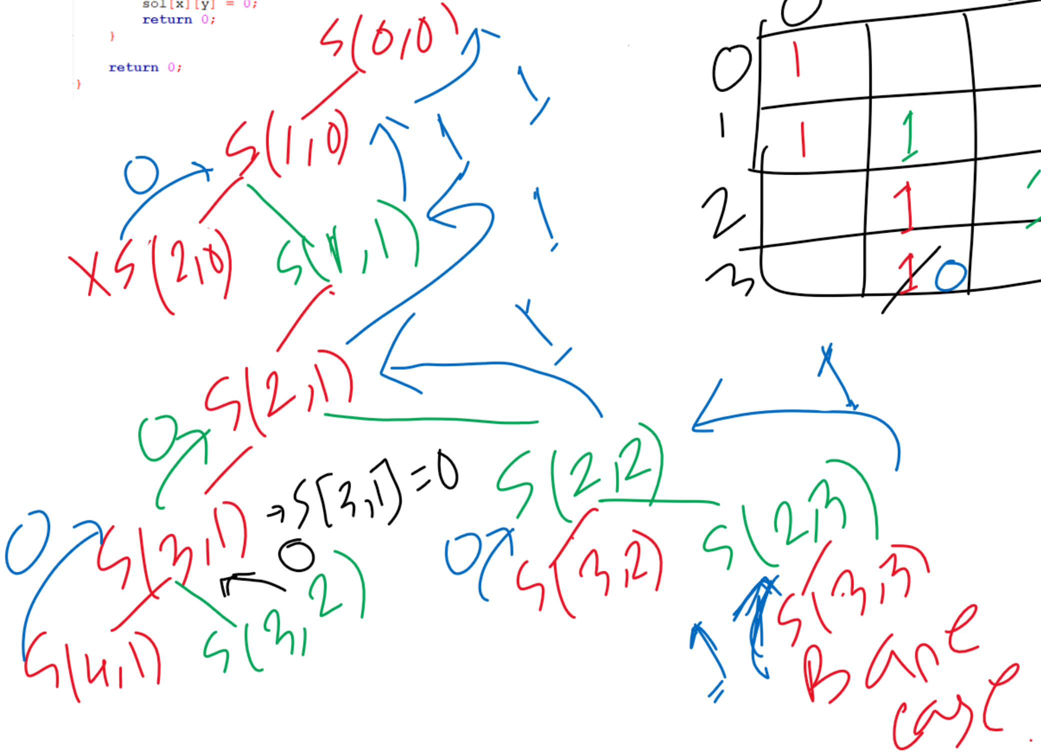
```
Trace: before x s(1, 0) mat = 1
Trace: before x s(2, 0) mat = 1
Trace: before y s(1, 1) mat = 1
Trace: before x s(2, 1) mat = 1
Trace: before x s(3, 1) mat = 1
Trace: before x s(4, 1) mat = 1
Trace: before y s(3, 2) mat = 1
Trace: after x,y s(3, 1) mat = 0
Trace: before y s(2, 2) mat = 1
Trace: before x s(3, 2) mat = 1
Trace: before y s(2, 3) mat = 1
Trace: before x s(3, 3) mat = 1
 1  0  0  0
 1  1  0  0
 0  1  1  1
 0  0  0  1
```

The code reached to this line only once. Should not it be in the program call stack?
-No! Because the recursion is happening in a condition! If both of the above conditions are false, then only the code is reaching here!

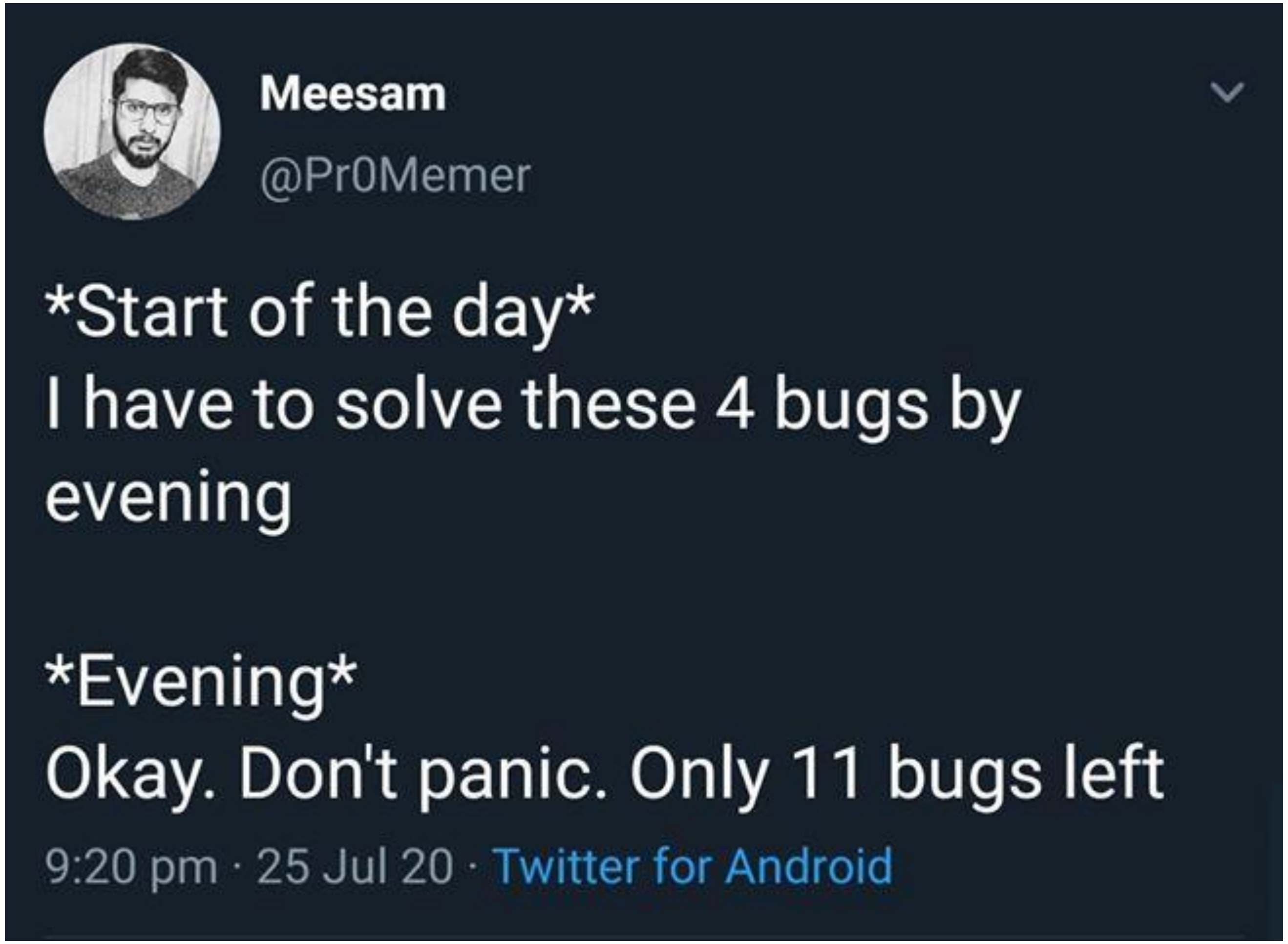
Here is the recursion tree of the maze problem for our input matrix

```c
/* A recursive utility function to solve Maze problem */
int 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 1;
    }

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

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

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

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

    return 0;
}
```





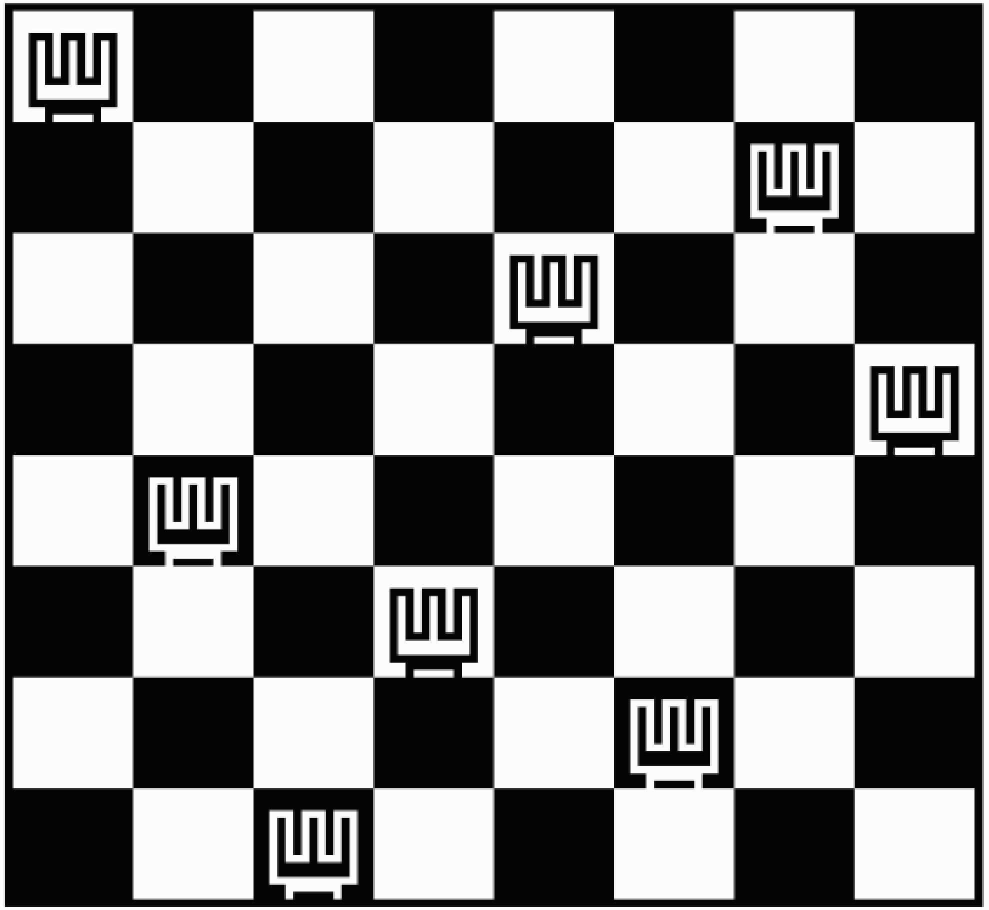
Fun! Life of a programmer



Courtesy: Facebook page of “I am a programmer, I have no life”

Eight Queens Problem

- **Find an arrangement of eight queens on a single chess board such that no two queens are attacking one another.**
- **In chess, queens can move (so long as no pieces are in the way).**
  1) **all the way down any row,**
  2) **All the way down any column**
  3) **All the way diagonal**
- **Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.**
- This is also called the N Queens problem since we can solve this problem for any NxN board with N Queens as well.

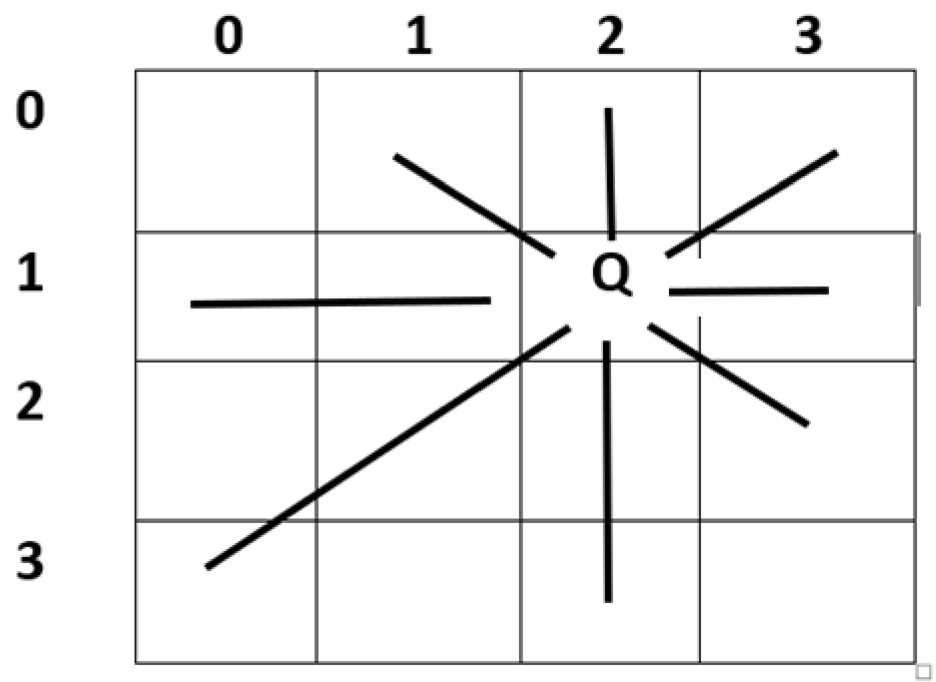


Eight Queens Problem

- The backtracking strategy is as follows:
- 1) Place a queen on the first available square in row 1.
- 2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
- 3) Continue in this fashion until either
  - (a) you have solved the problem, or
  - (b) you get stuck.
    - When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.

Now let us find out what cells are under attack if we place a queen at (1,2) in a 4-Queen problem

- All the cells will be under attack as shown in the picture.
- Who are those cells:
  - Same row as our queen’s place (1)
    - (1,0), (1,1), (1,3)
  - Same column as our queen’s place (2)
    - (0,2), (2,2), (3,2)
  - Top left to bottom right:
    - (0,1), (2,3)
    - Formula: Queen’s row-col = 1-2 = -1 //in our example
    - So, any cell, if row-col = -1 would be it’s diagonal and would be under attack.
    - Example: 0-1 = -1, 2-3 = -1
  - Top right to bottom left
    - (3,0), (2,1), (0,3)
    - Formula: Queen’s row+col = 1 + 2 = 3 //in our example
    - So, any cell, if row+column = 3 would be it’s another side’s diagonal and would be under attack
    - Example: 0+3 = 3, 2+1 = 3, 3+0 = 3



Eight Queens Problem

- **When we carry out backtracking, an easy way to visualize what is going on is a tree that shows all the different possibilities that have been tried.**

- **Now we will go through 4 Queen example with recursion in the class.**
- **Also will look through the code**
- **The example is based on this video for your reference and you can look it at home if you fail to take notes during the lecture:** https://www.youtube.com/watch?v=xouin83ebxE

More reading

- The codes for Maze and the Nqueen is available in webcourses
- Prof. Arup's Notes: http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/lec/Backtracking.doc
- Arup's EightQueen code:
  - http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/sampleprogs/eightqueens.c