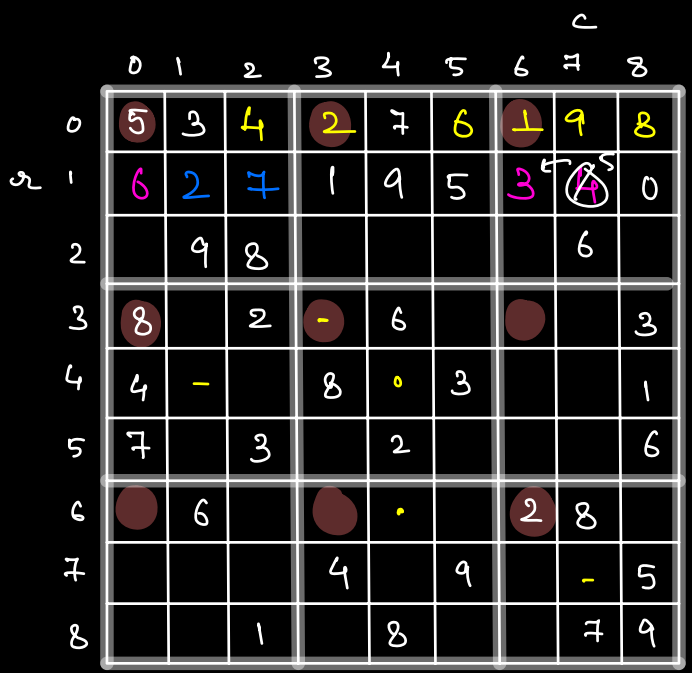
Q.1 SUDOKU



⇒ mat[9][9], partially filled, fill this Sudoku.
[1-9]

Rules:

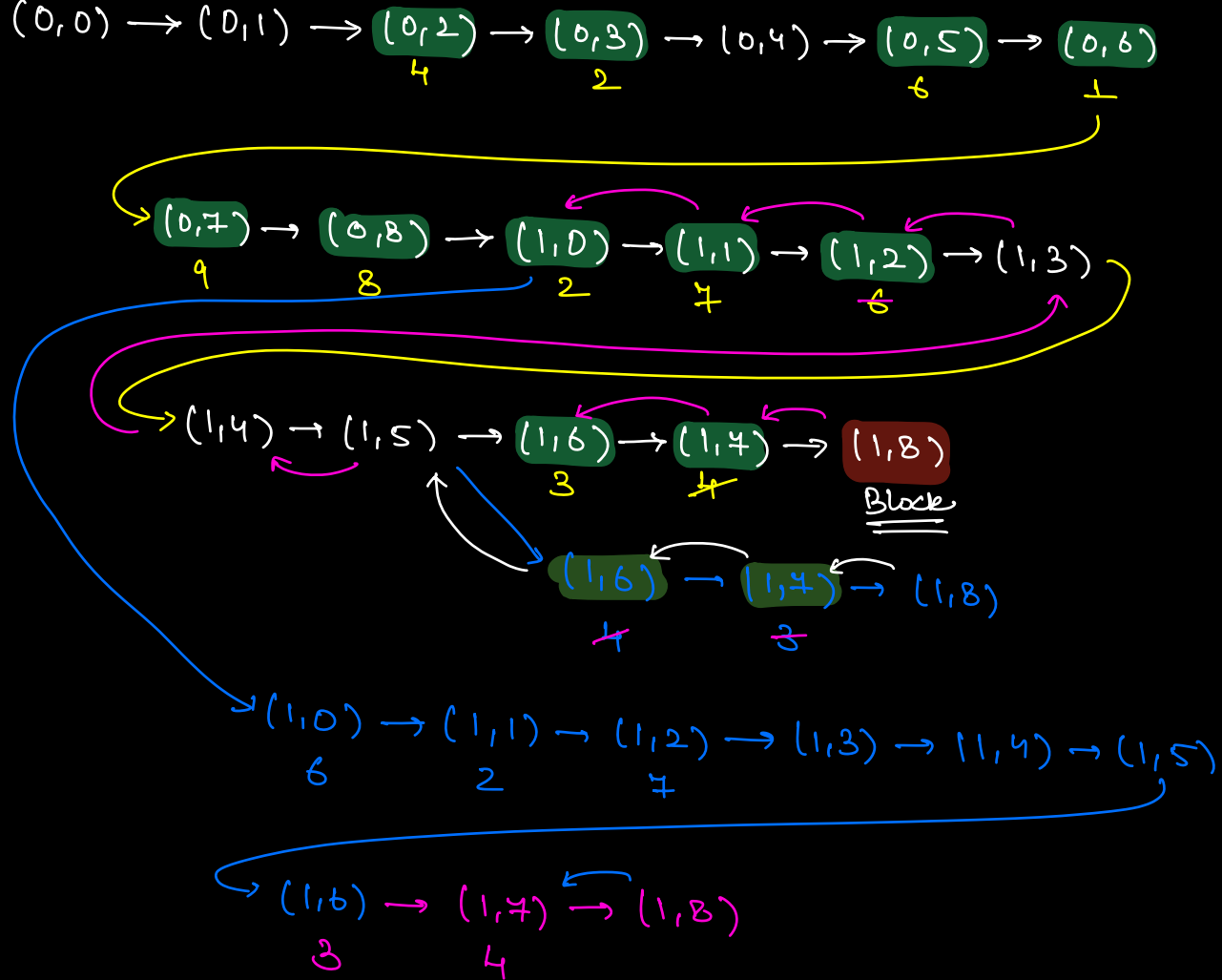
1: In a row, number can't repeat.
2: In a col, number can't repeat.
3: In a 3×3 box, number can't repeat.

Start index of 3×3 box

4,4 → 3,3
5,2 → 3,0
2,7 → 0,6
7,7 → 6,6
4,1 → 3,0

Nearest multiple of 3 <= i.

r → r − r%3
c → c − c%3



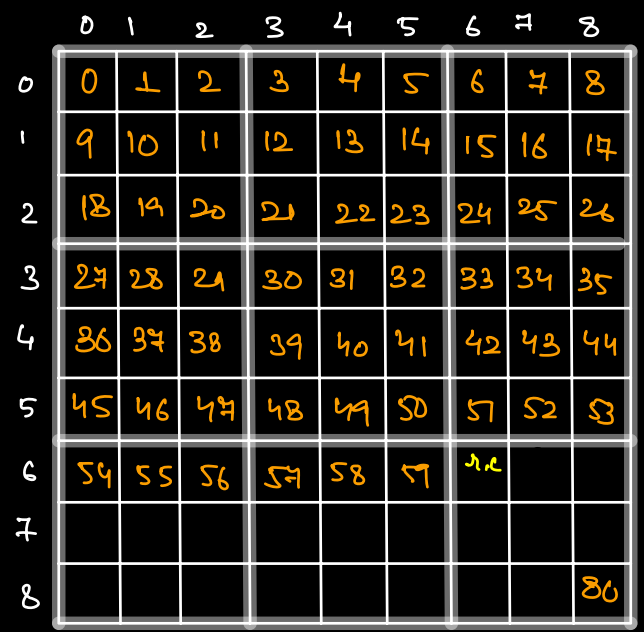
⇒ Backtracking

↳ Try all the possibilities.

⇒ Recursion + Backtracking

⇒ function

mat[][], (n)



	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16	17
2	18	19	20	21	22	23	24	25	26
3	27	28	29	30	31	32	33	34	35
4	36	37	38	39	40	41	42	43	44
5	45	46	47	48	49	50	51	52	53
6	54	55	56	57	58	59	60		
7									
8									80

9x9

$x = 24$ → $r = 24 \mid 9$, $c = 24 \cdot 1 \cdot 9$

$2 \times 9 + 6$

$x = 50 \Rightarrow r = 50 \mid 9 = 5$

$c = 50 \cdot 1 \cdot 9 = 5$

$x \longrightarrow r = x \mid 9$

$c = x \cdot 1 \cdot 9$

```
void SudokuSolver(mat[][], x) {
    if (x == 81) {
        print(mat[][])
        return;
    }
    r = x/9, c = x%9;
    if (mat[r][c] != 0) {
        // filled cell.
        SudokuSolver(mat, x+1);
    }
    else { // mat[r][c] = 0 ⇒ Blank cell
        for (i = 1; i <= 9; i++) {
            if (isValid(mat, r, c, i)) {
                mat[r][c] = i;
                ⇒SudokuSolver(mat, x+1);
                mat[r][c] = 0
            }
        }
    }
}
```

```
bool isValid(mat[][], r, c, d){
    // this fun checks if it is possible to put
    // i at index r,c.
    // row & col
    for(j=0; j<9; j++){
        if(mat[r][j]==d) return false;
        if(mat[j][c]==d) return false;
    }
    // Start index of 3x3 Box
    x = r - r%3
    y = c - c%3
    for(i=x; i<x+3; i++){
        for(j=y; j<y+3; j++){
            if(mat[i][j]==d){
                return false;
            }
        }
    }
    return true;
}
```

TC: $9 \times 9 \times 9 \times 9 \cdots 9 \Rightarrow 9^{81}$ (81 times)

↓

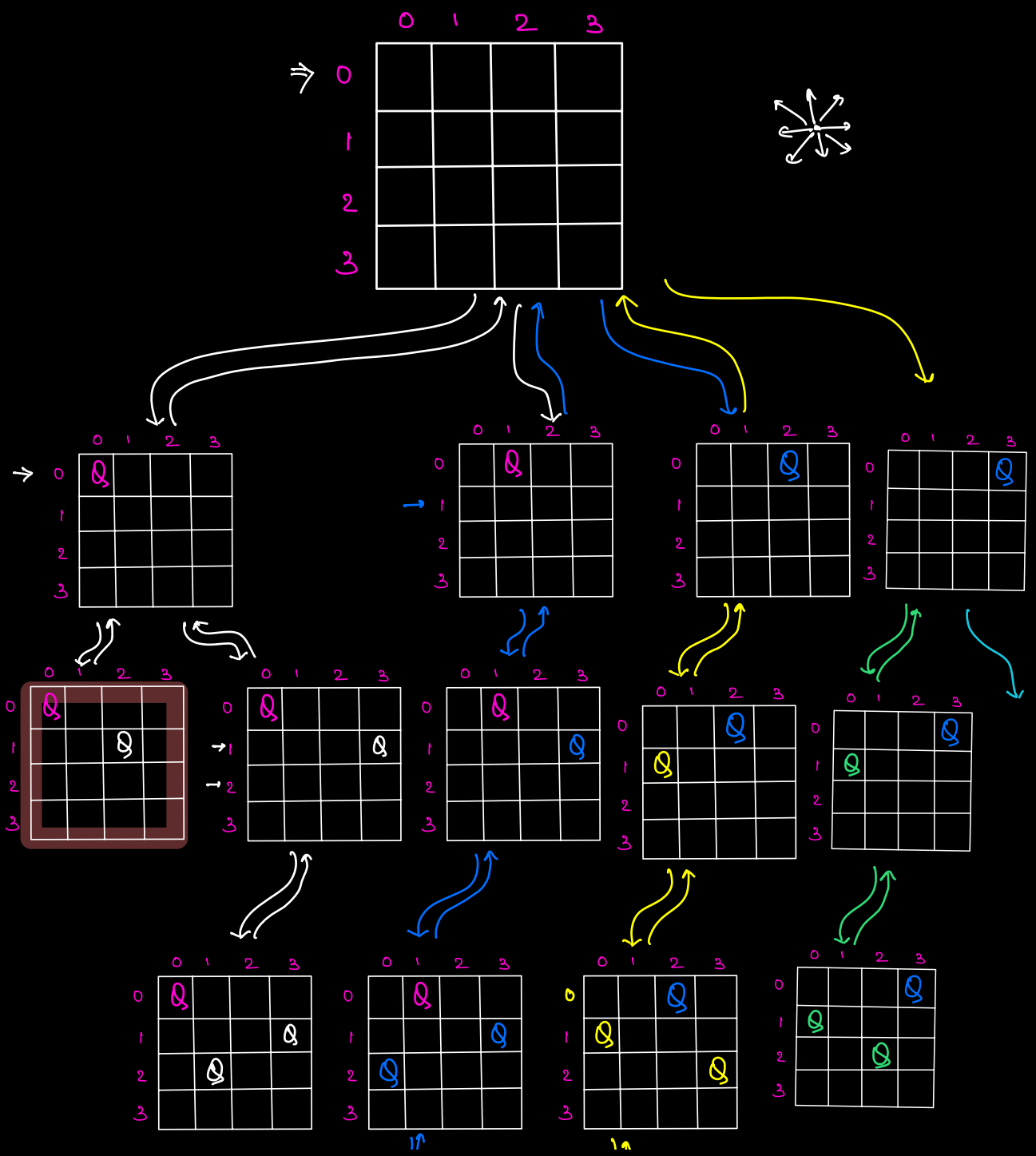
Upper Bound

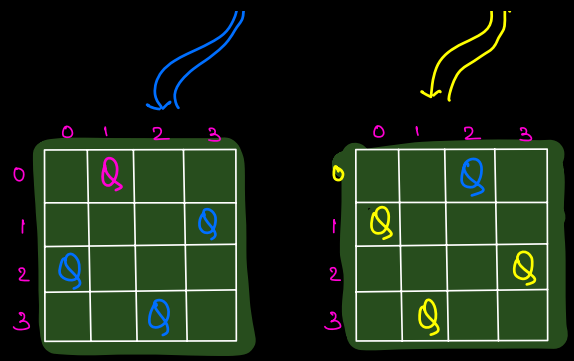
SC: 9×9

Q. N-Queen.

Given NxN matrix, print all valid possibilities of placing N queens in the given matrix.

⇒ 1 row can have only 1 queen.

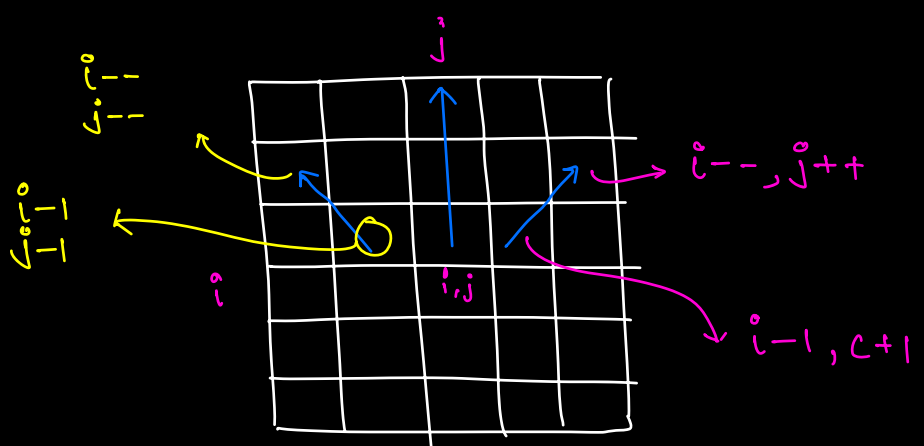





```
void nQueens(mat[][], N, i) {
    if(i == N) {
        print(mat)
        return;
    }
    for(j = 0; j < N; j++) {
        // Check if we can place a Queen i,j
        if(check(mat, i, j)) {
            mat[i][j] = 1;
            nQueens(mat, i+1);
            mat[i][j] = 0;
        }
    }
}
```

i → row index

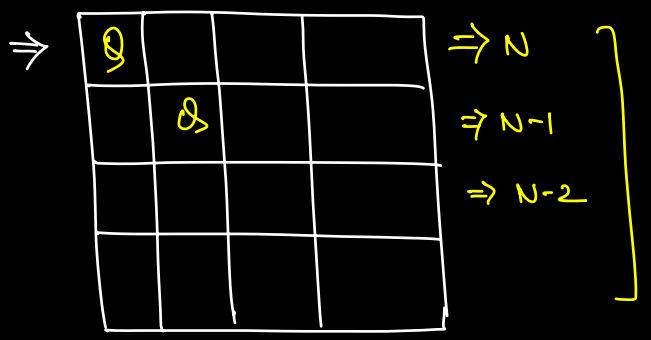
i,j



```
bool check(mat[][], i, j)  → O(N)
{
    for(r=0; r<i; r++){
        if(mat[r][j] == 1)
            return false;
    }                                   O(N)

    r = i-1, c = j+1
    while(r>=0 && c<N){
        if(mat[r][c] == 1)
            return false;
        r--
        c++                             O(N)
    }

    r = i-1, c = j-1
    while(r>=0 && c>=0){
        if(mat[r][c] == 1)
            return false;
        r--
        c--                             O(N)
    }

    return true;
}
```

HW

→ Check TC of N-Queens.



Check fun^

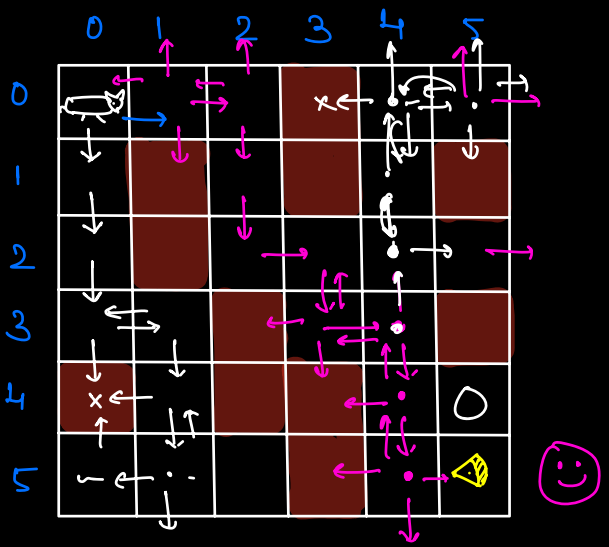
↓

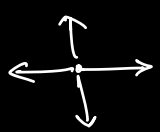
$\Rightarrow N! \times N$

(Upper bound.)

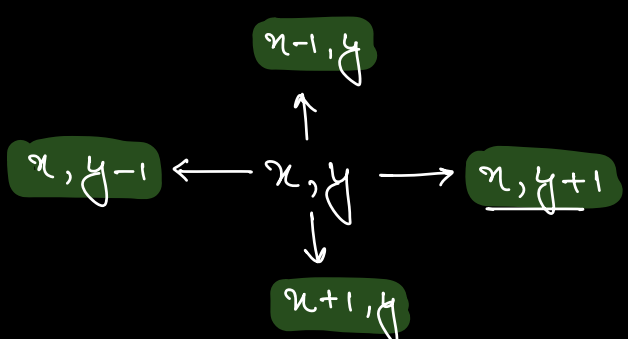
Q.3 Rat in a Maze

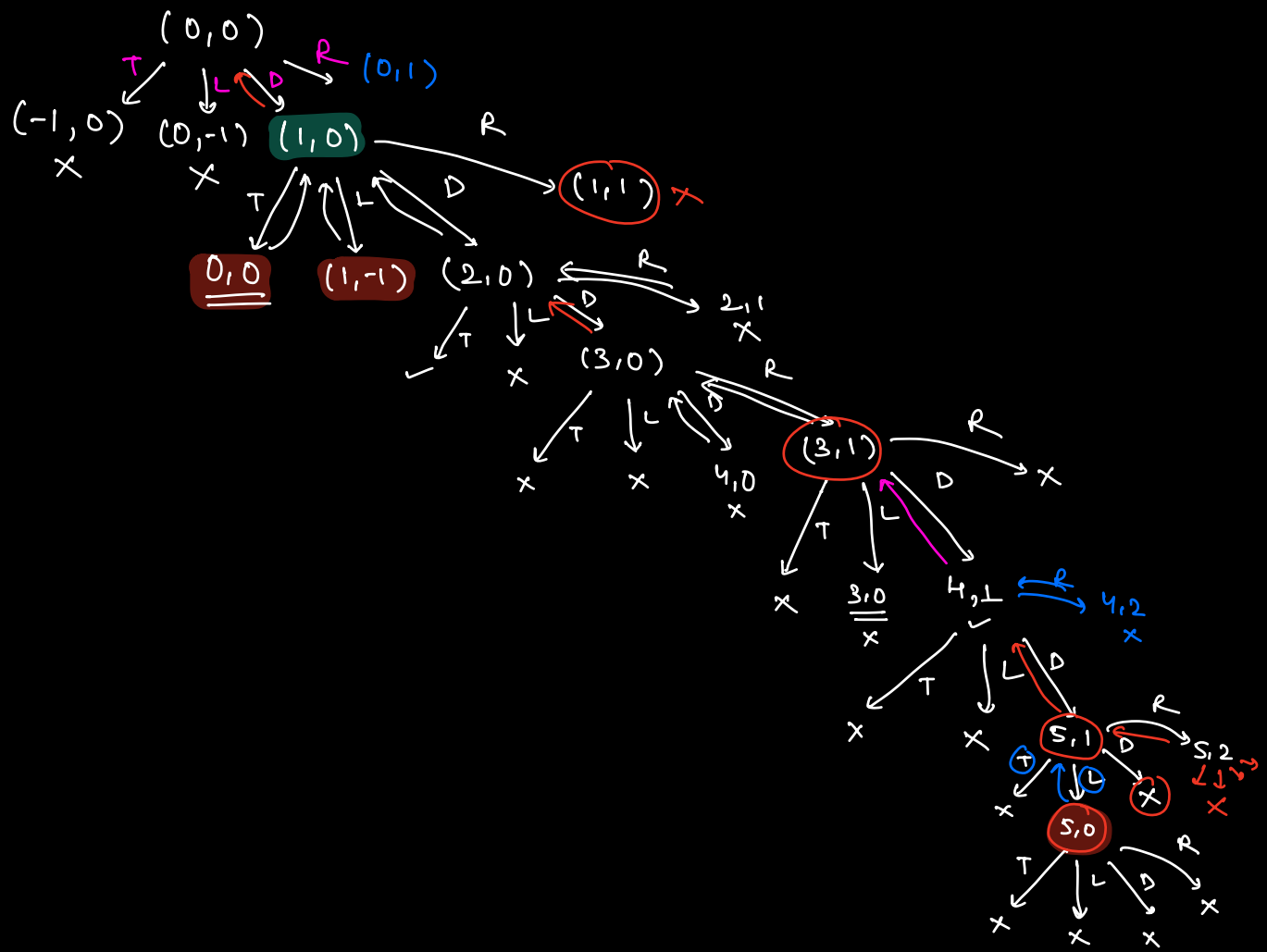
N × M





→ One cell can't be visited more than once.





mat[i][j] = 0 ⇒ free
1 ⇒ visited
2 ⇒ Blocked.

List<pair<int, int>>

```
Void maze(mat[][], N, M, i, j, list<pair<int,int>> path)
{
    if(i<0 || i>=N || j<0 || j>=M) {
        return;
    }
    if(mat[i][j]==1 || mat[i][j]==2)
        return;
    if(i==N-1 && j==M-1){
        print(path)
        return;
    }
    if(mat[i][j]==0) {
        mat[i][j]=1;
        path.add({i,j});
        maze(mat, N, M, i-1, j, path);
        maze(mat, N, M, i, j-1, path)
        maze(mat, N, M, i+1, j, path)
        maze(mat, N, M, i, j+1, path)
        mat[i][j]=0;
        path.remove();
    }
}
```

TC : O(N×M)

↳ Each cell is visited exactly once.

SC : O(NM)

O(NM) means maximum recursive function calls covering all cells