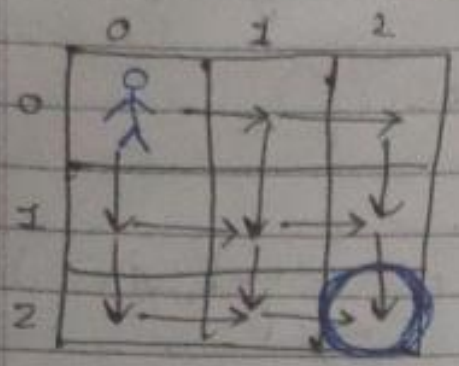
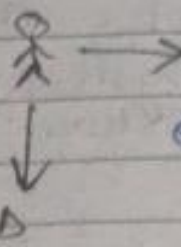
Mazes

Q





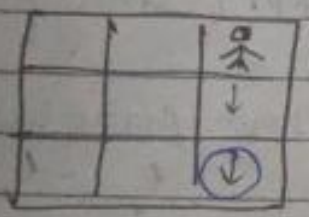
only right & down allowed

Ans: RRDD, DDRR, RDDR, RDRD .....

No. of ways from (0,0) → (2,2)

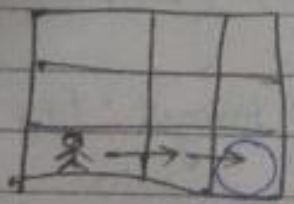
BASE CONDITION

⇒ When we are at the last ~~row~~ column

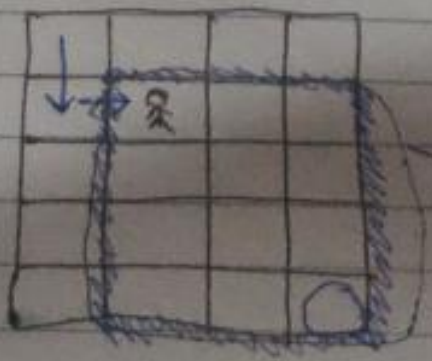


YOU CAN ONLY GO DOWNWARDS

⇒ When we are at the last row



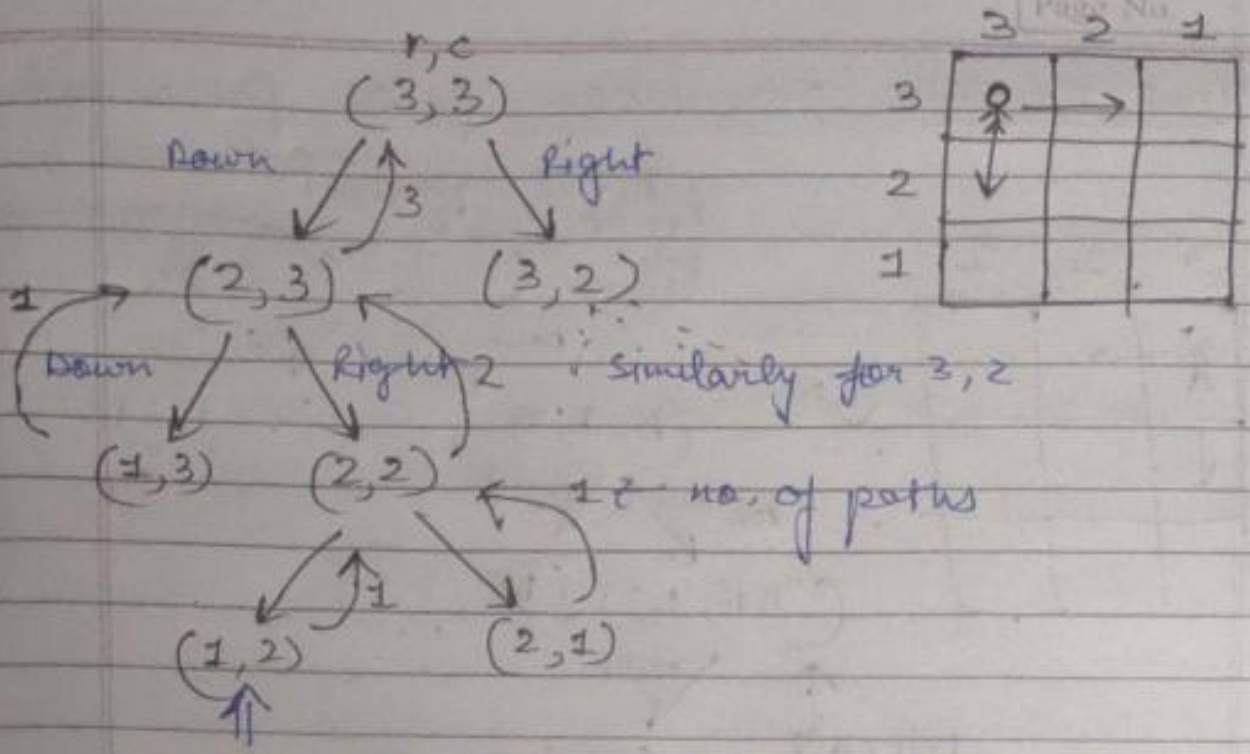
YOU CAN ONLY GO RIGHT



paths till that point

DR + (Answer of this part)

Remaining paths

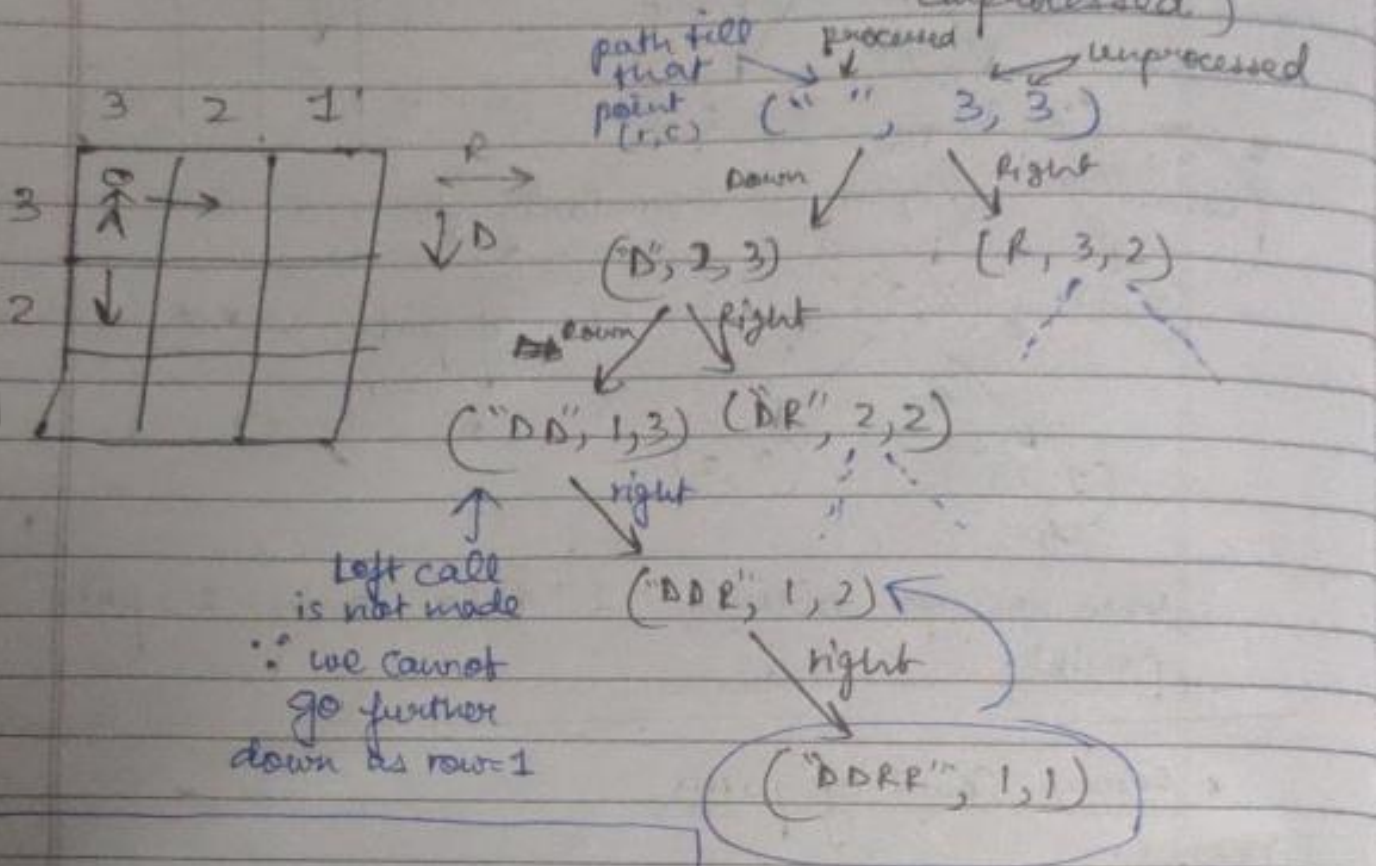


When row or column = 1 then only 1 path possible

* Return Count of paths

```
static int count(int row, int col){
  if(row == 1 || col == 1){
      return 1; //only 1 ans possible
  }
  int down = count(row-1, col);   // Get count when going downwards
  int right = count(row, col-1);  // Get count when going right
  return down + right;            // Add both answers & return
}
```

* Print the path (permutations, follow processed & unprocessed)



* Code

initially "" when called from main()

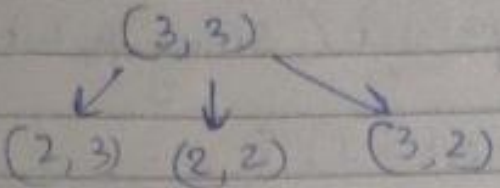
When both 1, 1 we get an answer. Hence, return the processed

```
void path(String p, int r, int c)
{
    if(r==1 && c==1){ <- r=1, c=1 means you reached the destination
        System.out.println(p); <- print the path & return
        return;
    }

    if(r>1){ <- if not at last row, then you can go further DOWN i.e. (r-1, c)
        path(p+'D', r-1, c);
    }
    if(c>1){ <- if not at last col then you may go further RIGHT i.e. (r, c-1)
        path(p+'R', r, c-1);
    }
}
```

* Add all paths in ArrayList

```
ArrayList<String> pathPet(String p, int r, int c){
    if(r==1 && c==1){
        ArrayList<String> list = new ArrayList<>();
        list.add(p);
        return;
    }

    ArrayList<String> list = new ArrayList<>();

    if(r>1){
        list.addAll(pathPet(p+'D', r-1, c));
    }
    if(c>1){
        list.addAll(pathPet(p+'R', r, c-1));
    }

    return list;
}
```

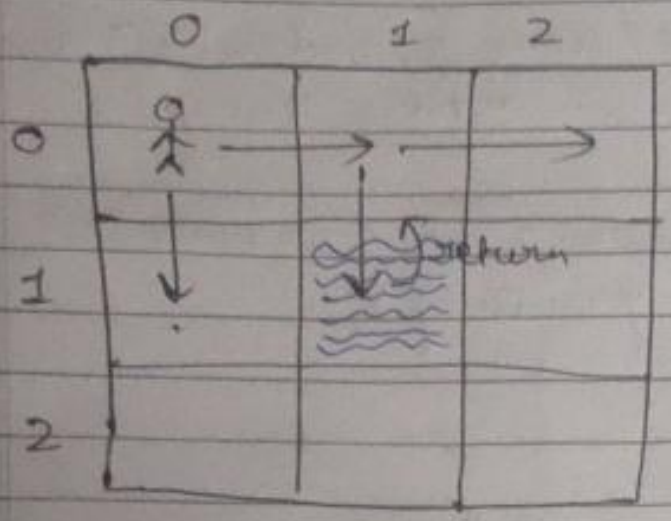
* With Diagonal, Right, Down allowed



⇒ Conditions for going :
Down ⇒ (r > 1)
Diagonal ⇒ (r > 1 && c > 1)
Right ⇒ (c > 1)

```
                                    Diagonal
ArrayList<String> pathRet(String p, int r, int c){
if(r == 1 && c == 1){
    ArrayList<String> list = new ArrayList<>();
    list.add(p);
    return;
}
ArrayList<String> list = new ArrayList<>();

if(r > 1 && c > 1){ //diagonal calls
    list.addAll(pathRetDiagonal(p+'D', r-1, c-1));
}

if(r > 1){ // vertical (Right) calls
    list.addAll(pathRetDiagonal(p+'V', r-1, c));
}
if(c > 1){ // Horizontal (Down) calls
    list.addAll(pathRetDiagonal(p+'H', r, c-1));
}

return list;
}
```

Q. Maze With Obstacles



Boolean Matrix

false ⇒ river
↖ can't cross it

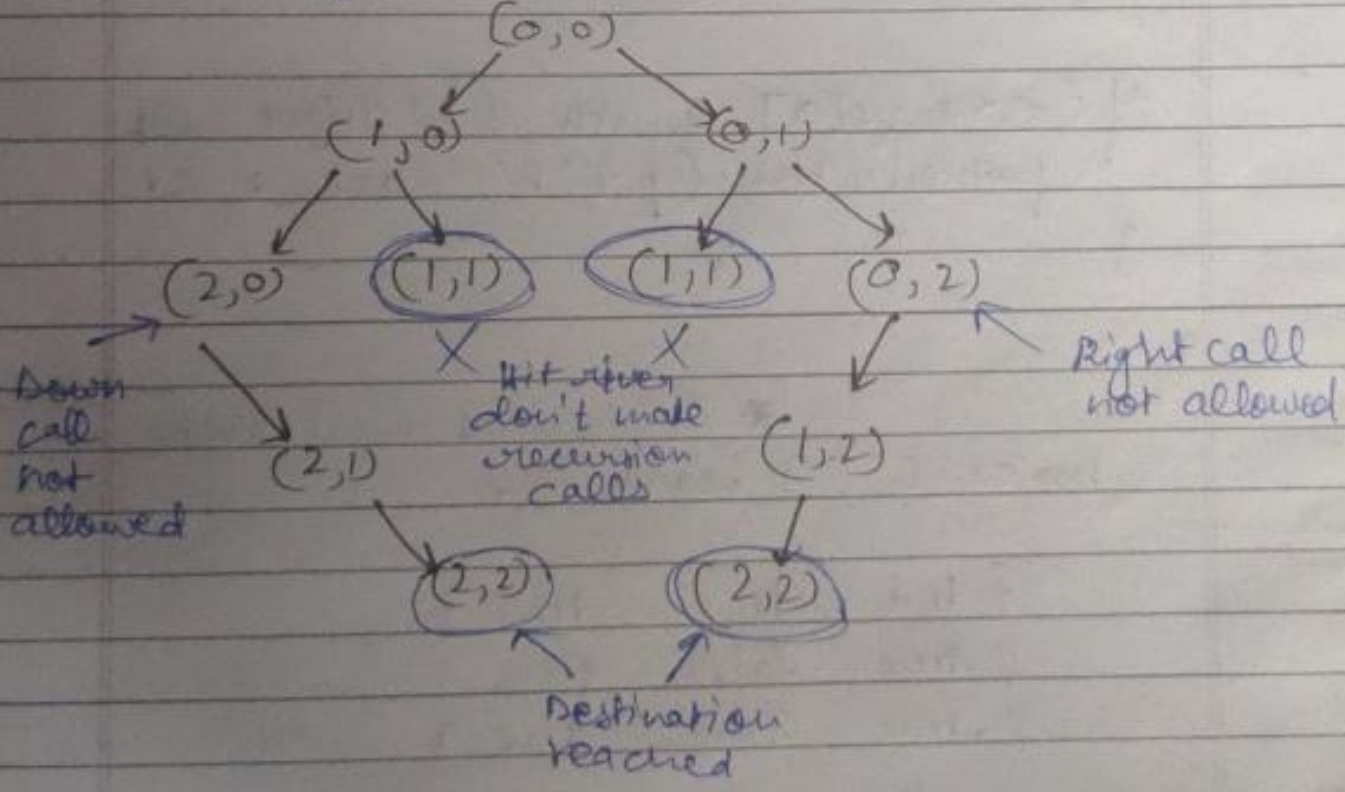
Allowed paths

→ R

↓ D

NOTE: When you land on a new call, check whether that is river or not.

If you land on river, stop recursion of that call



* Print all paths (starts from (0,0), boolean matrix)

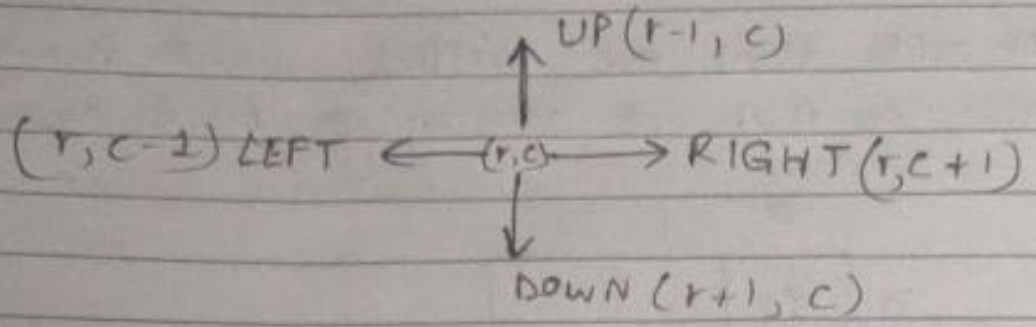
```
void pathRestrictions(String p, boolean[][] maze,
                      int r, int c) {

    if (r == maze.length-1 && c == maze[0].length-1)
    {  System.out.println(p);
       return;
    }

    if (!maze[r][c]){ // if (maze[r][c] = false)
        return;       // river encountered
    }                 // return without calling further

    if(r < maze.length-1) {//downward calls
        pathRestrictions(p+'D', maze, r+1, c);
    }

    if(c < maze[0].length-1){//right calls
        pathRestrictions(p+'R', maze, r, c+1
    }

}
```

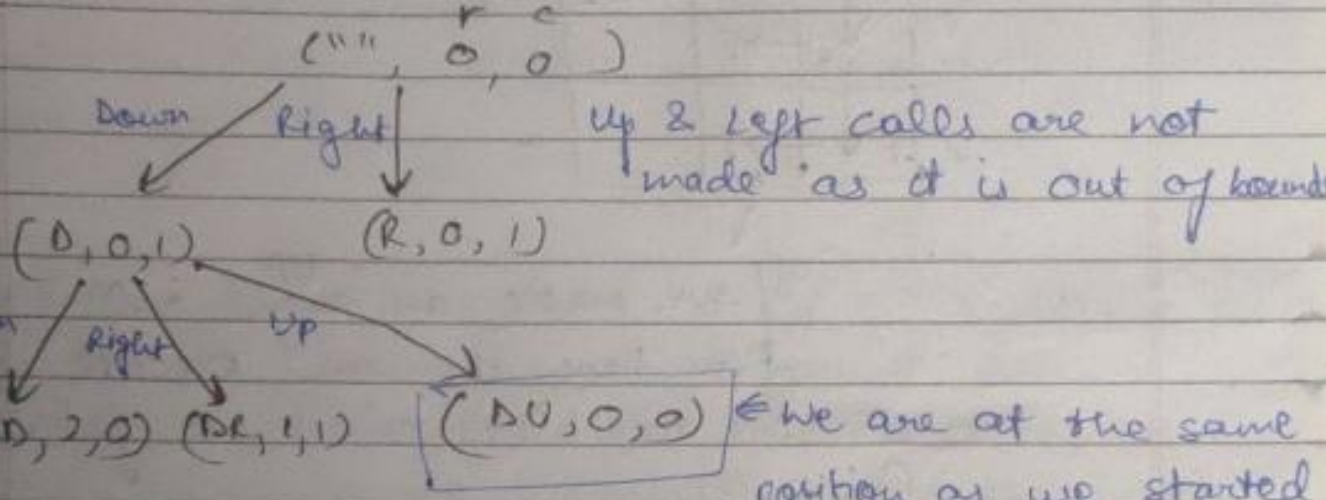
* This is what maze looks like

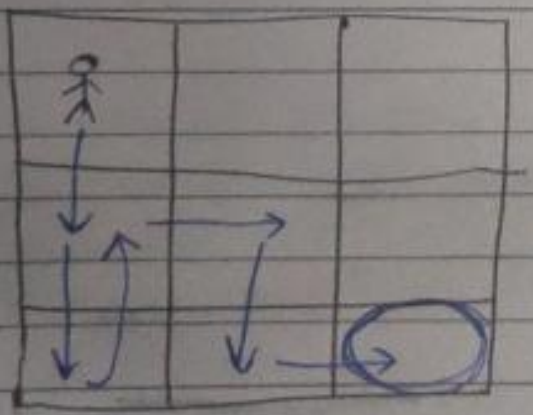
```
boolean[][] board = {

    { true, true, true },
    { true, false, true },
    { true, true, true }

};
```

Date: / /
Page No.

MAZE (All Directions allowed)







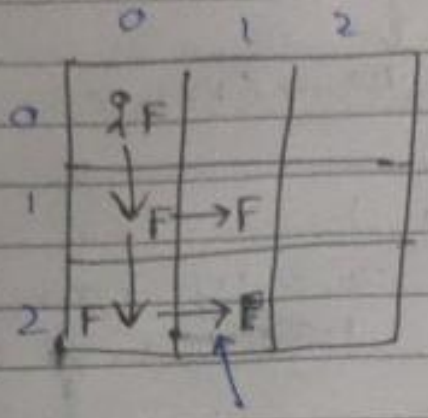
THIS **CANNOT** BE AN ANSWER

* Do not move back the same path

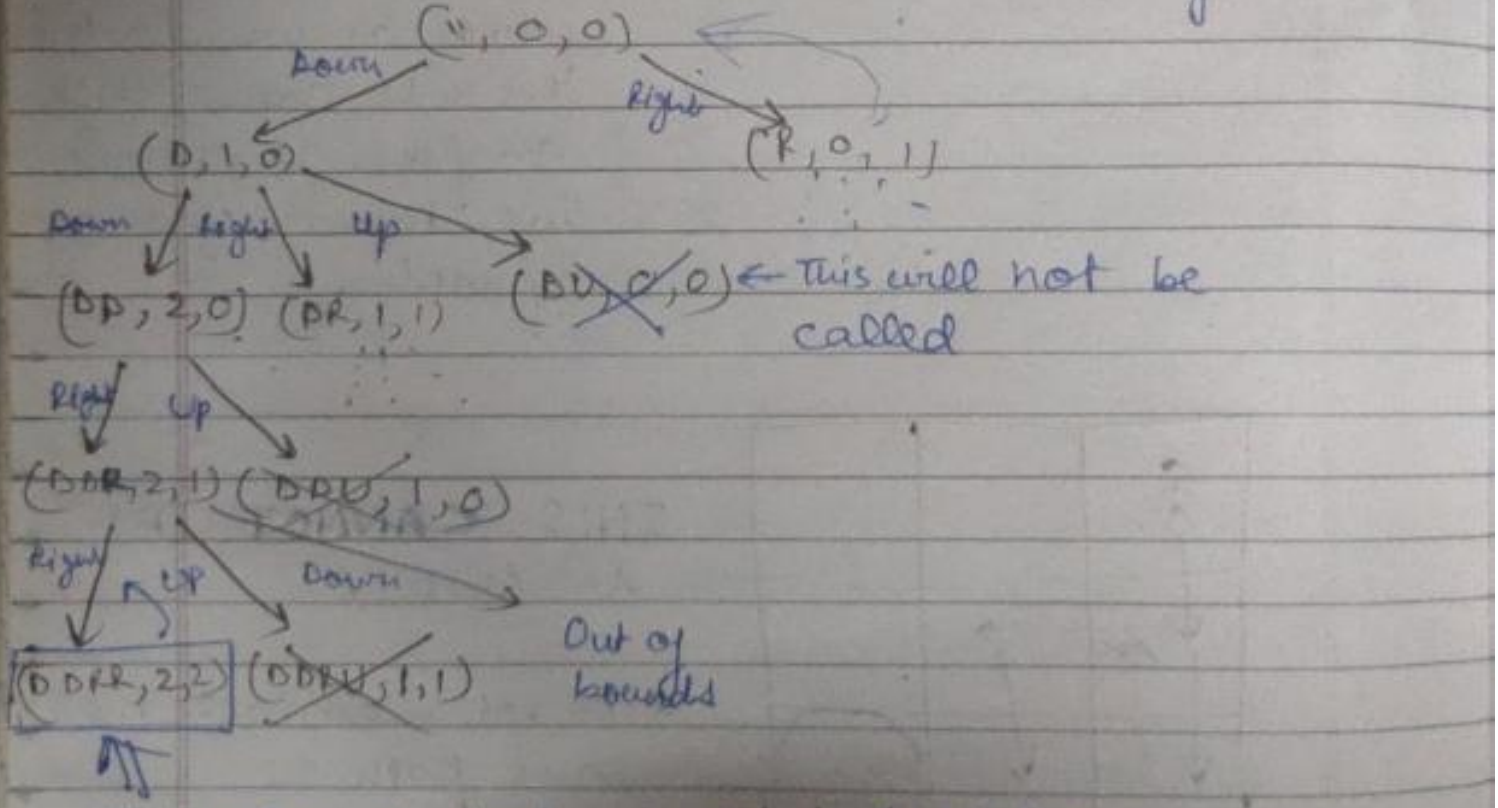
* This is Backtracking problem

* How to solve this problem

All cells that are visited, mark those as false, so that we cannot go back again.



If we make an ~~above~~ UP call from here, it won't execute as the block above is false.



∴ we are at last ∴ this is 1 of the answers

NOTE: Here is a problem is that original array is being modified

* While going back the ~~path~~ cell is marked visited which don't allow future answers to be formed.

Common Sense: Marking false == I have that cell in my current path.
So, when that path is over, e.g.: You are in another recursion call, those cells should not be false

Important

While you are moving back you restore the maze as it was.

* When do we go back?
When the function ~~returns~~ is returned.

⇒ When you come out of the recursive f^n → you are in above recursive call.
Hence, remark the cells as true.

This is known as backtracking

Date: / /
Page No

Backtracking

We are making some changes while going to ahead recursive calls.

What would the 4th scenario be, hadn't those changes be made?

Undoing changes made in current recursive call for restoring the state in previous recursive call is called BACKTRACKING.

Code (All 4 directions MAZE)

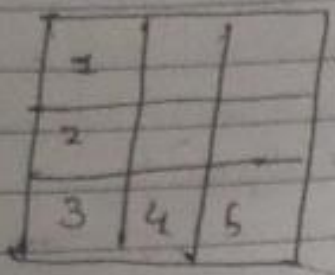
```
void allPath(String p, boolean[][] maze, int r, int c)
{
    if(r == maze.length-1 && c == maze[0].length-1){
        System.out.println(p);
        return;
    }

    if(!maze[r][c]){
        return;
    }

    // I am considering this block in my path
    maze[r][c] = false;

    if(r < maze.length-1){ // Down Call
        allPath(p + 'D', maze, r+1, c);
    }
```

```
if (c < maze[0].length-1) { //right call
    allpath(p+'R', maze, r+1, c);
}
if (r>0) { //up call
    allpath(p+'U', maze, r-1, c);
}
if (c>0) { //left call
    allpath(p+'L', maze, r, c-1);
}

// this line is where the fn will be over

// so before the fn gets removed, also remove the
changes that were made by that fn
maze[r][c] = true;
}

}
```

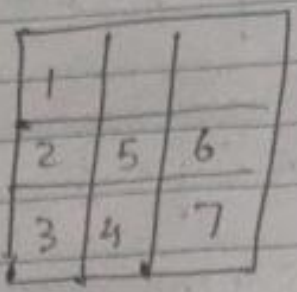
```
boolean[][] maze = {
        { true, true, true },
        { true, true, true },
        { true, true, true }
};
```

GIVEN MAZE MATRIX

* Print the paths with matrix



DDRR



DDRURD

→ Take a step variable
→ update the path array
→ print it in base condition
→ BACKTRACK (∵ All reference variables in diff f^n arguments point to same object)

Code

~~void allpathprint(~~
void main() {
boolean[][] board = {
{true, true, true},
{true, true, true},
{true, true, true}
};
int[][] path = new int[board.length][board[0].length];

allpathprint("", board, 0, 0, path, 1);

}

```
void allpathprint(String p, boolean[][] maze, int r, int c,
                  int[][] path, int step) {

if (r == maze.length-1 && c == maze[0].length-1){
 path[r][c] = step; // the destination block is also a step

 for(int[] arr: path){
     System.out.println(Arrays.toString(arr));
 }
  System.out.println(p);

  System.out.println();
  return;
}

if(!maze[r][c]) {
        return;
 }
// Considering this block in my path
 maze[r][c] = false;
// marking the step
 path[r][c] = step;

 if( r<maze.length-1){ // Down call
    allpathprint(p+'D', maze, r+1, c, path, step+1);
 }

 if( c<maze[0].length-1) { // Right call
    allpathprint( p+'R', maze, r, c+1, path, step+1);
 }
```

```
if(r>0){ // up call
    allpathprint(p+'U', maze, r-1, c, path, step+1);
}

if(c>0){// left call
    allpathprint(p+'L', maze, r, c-1, path, step+1);
}

// this line is where fn will be over

// so before the fn gets removed, also remove the
// changes that were made by that fn
maze[r][c] = true;
path[r][c] = 0;
}
```