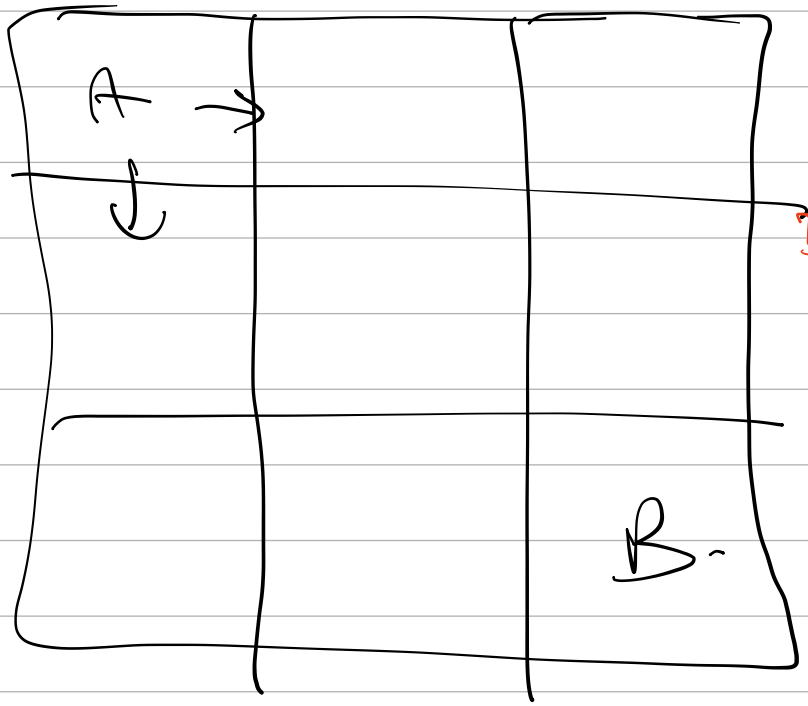
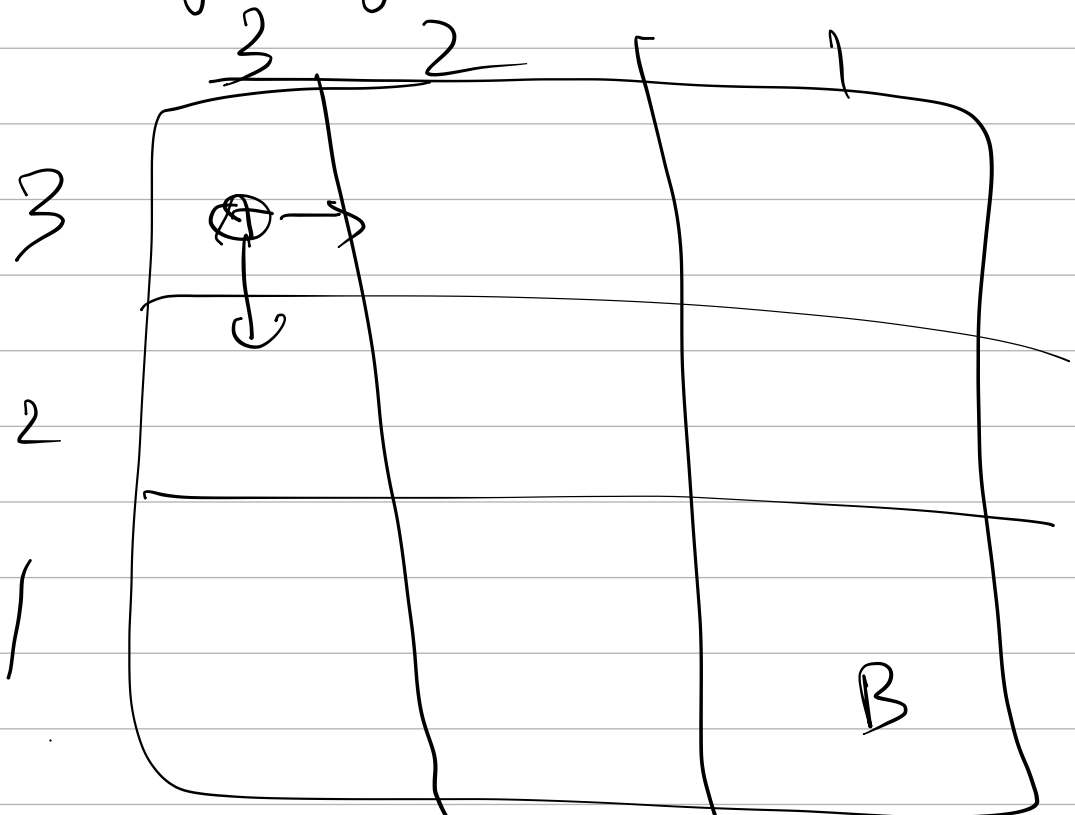
Maze Problems

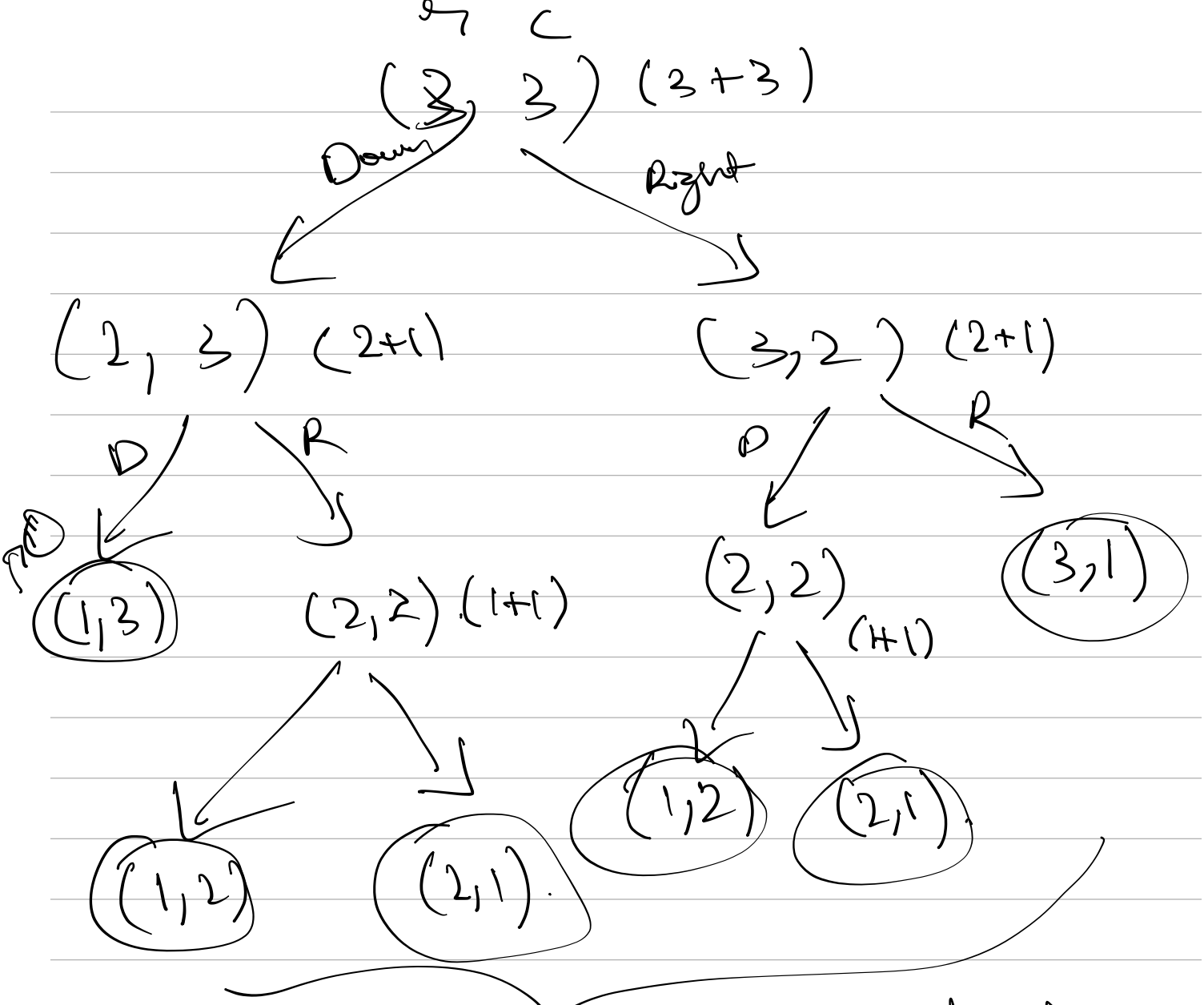


Reaching B from A. with restricted conditions

→ Move from A to B, by only moving Right or down in each call.



→ Find the no. of paths A can reach B.



All these return 1, when either row / column will be 1 (max)

→ As the values are returned, the upper nodes add 1 from each returned function call.

→ When row / column becomes 1, value 1 is returned because only one path will be available at each call when row / column reaches max.

Base condition:-

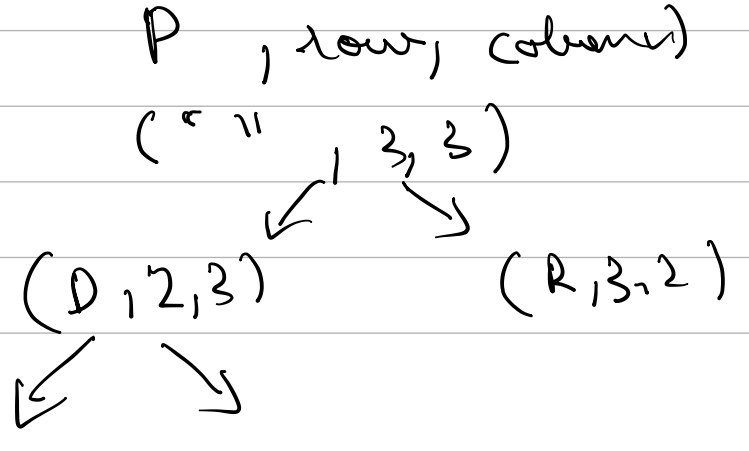
When (row == 1 || column == 1)
return 1;

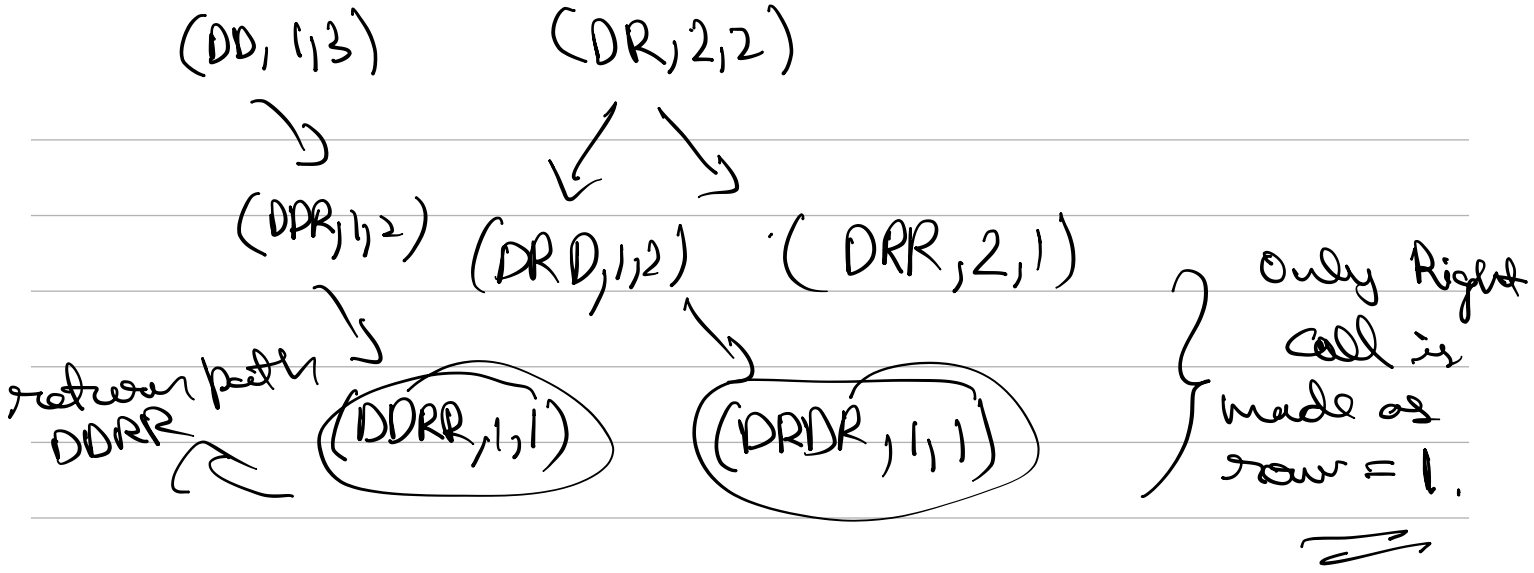
Code:-

```java
public class Maze {
    public static void main(String[] args) {

    }

    static int count(int r, int c) {
        if (r == 1 || c == 1) {
            return 1;
        }
        int left = count(r:r-1, c);
        int right = count(r, c:c-1);
        return left + right;
    }
}
```

2Q). Suppose we are printing paths from A to B.

Note:- Use a 'processed' string. to store path.





⇒

☆). Here the base condition is till both row and column becomes 1.

Return path.

Note:- This method is similar to subset question:- Processed and Unprocessed.

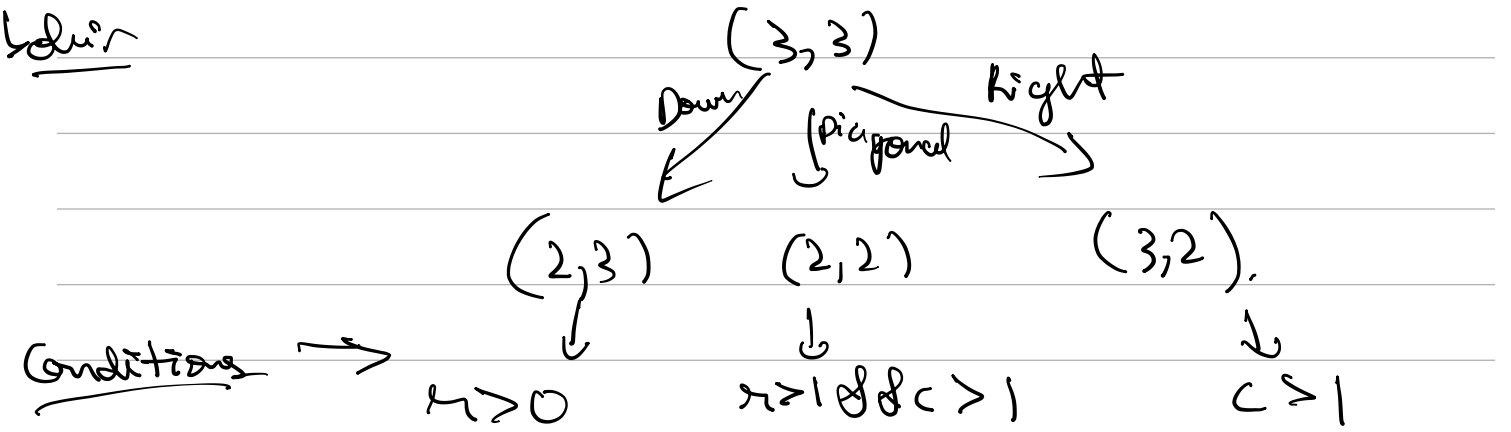
Code:-

```
public static void main(String[] args) {
    path(p:"", r:3, c:3);
}
```

```
static void path(String p, int r, int c) {
    if (r == 1 && c == 1) {
        System.out.println(p);
        return;
    }

    if (r > 1) {
        path(p:p + 'D', r:r-1, c);
    }

    if (c > 1) {
        path(p:p + 'R', r, c:c-1);
    }
}

}
```

39). Suppose to reach B from A, we can go Right, left and diagonal.

Solution



Code for this by returning an Arraylist consists of path:-

```
public class Maze {
    public static void main(String[] args) {
        System.out.println(pathRetDiagonal( p: "", r: 3, c: 3));
    }
```

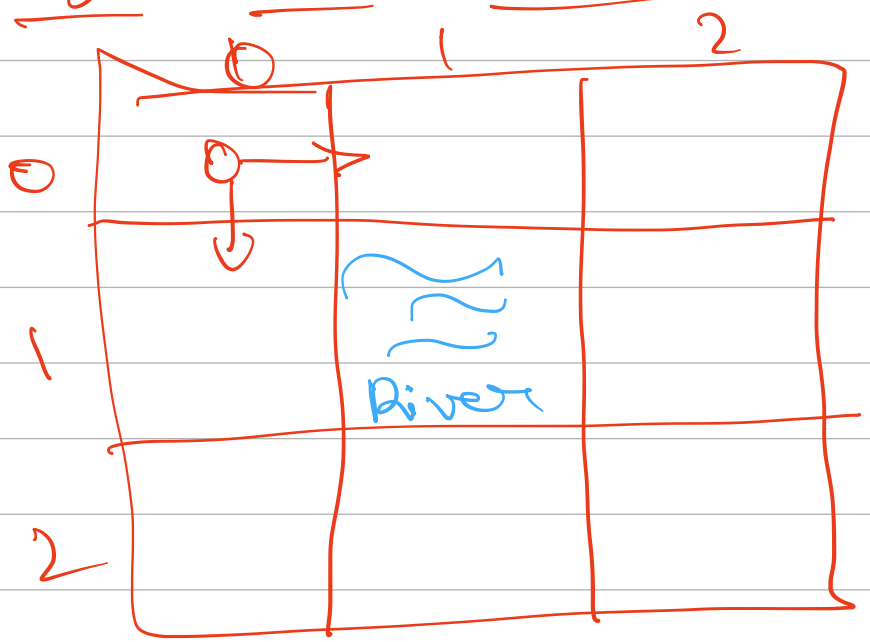
```
    static ArrayList<String> pathRetDiagonal(String p, int r, int c) {
        if (r == 1 && c == 1) {
            ArrayList<String> list = new ArrayList<>();
            list.add(p);
            return list;
        }

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

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

        if (r > 1) {
            list.addAll(pathRetDiagonal( p: p + 'V', r: r-1, c));
        }

        if (c > 1) {
            list.addAll(pathRetDiagonal( p: p + 'H', r, c: c-1));
        }

        return list;
    }

}
```

→ D means Diagonal.

→ V means Vertical movement (only Down)

→ H means Horizontal. (only Right)

Maze with Obstacles :-

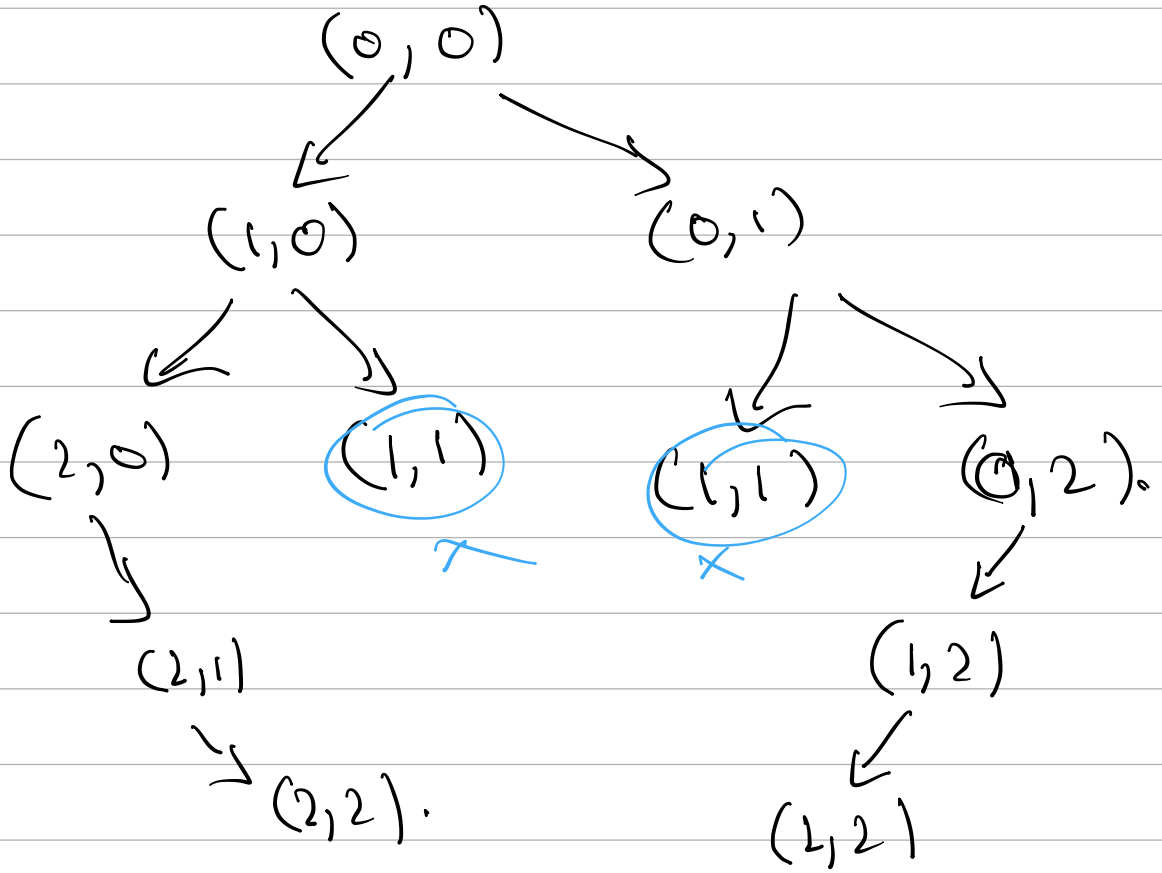


Boolean Matrix

False → River.

Note :- When you land on new cell check whether that is a river or not.

If you land on river stop recursion for that call.



Base condition:-

when (r == maze.length - 1 && c == maze[0].length - 1)

{ return path; }

Check condition:- → checking if false in case of river.

if (! maze[r][c])

{

return;

}

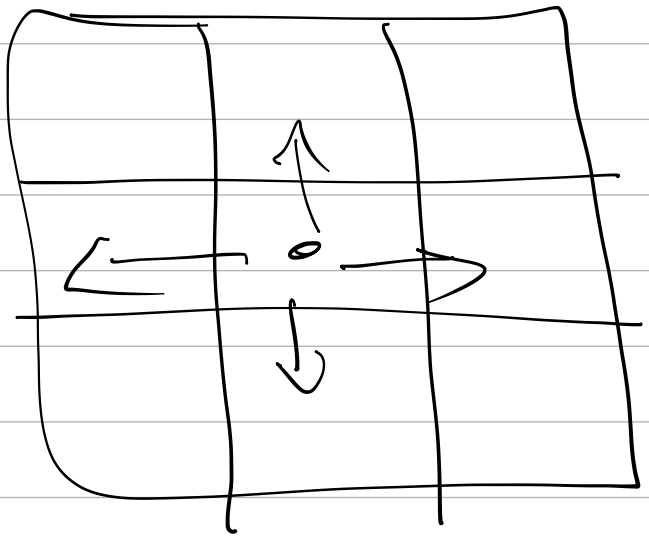
CODE:-

main () {

```
boolean[][] board = {
        {true, true, true},
        {true, false, true},
        {true, true, true}
};

pathRestrictions(p: "", board, r: 0, c: 0);
}
```

```
static 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]) {
        return;
    }

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

    if (c < maze[0].length - 1) {
        pathRestrictions(p: p + 'R', maze, r, c: c + 1);
    }
}
```

Q). Maze problem if all paths are allowed left, right, up, down allowed.



Sample code :-

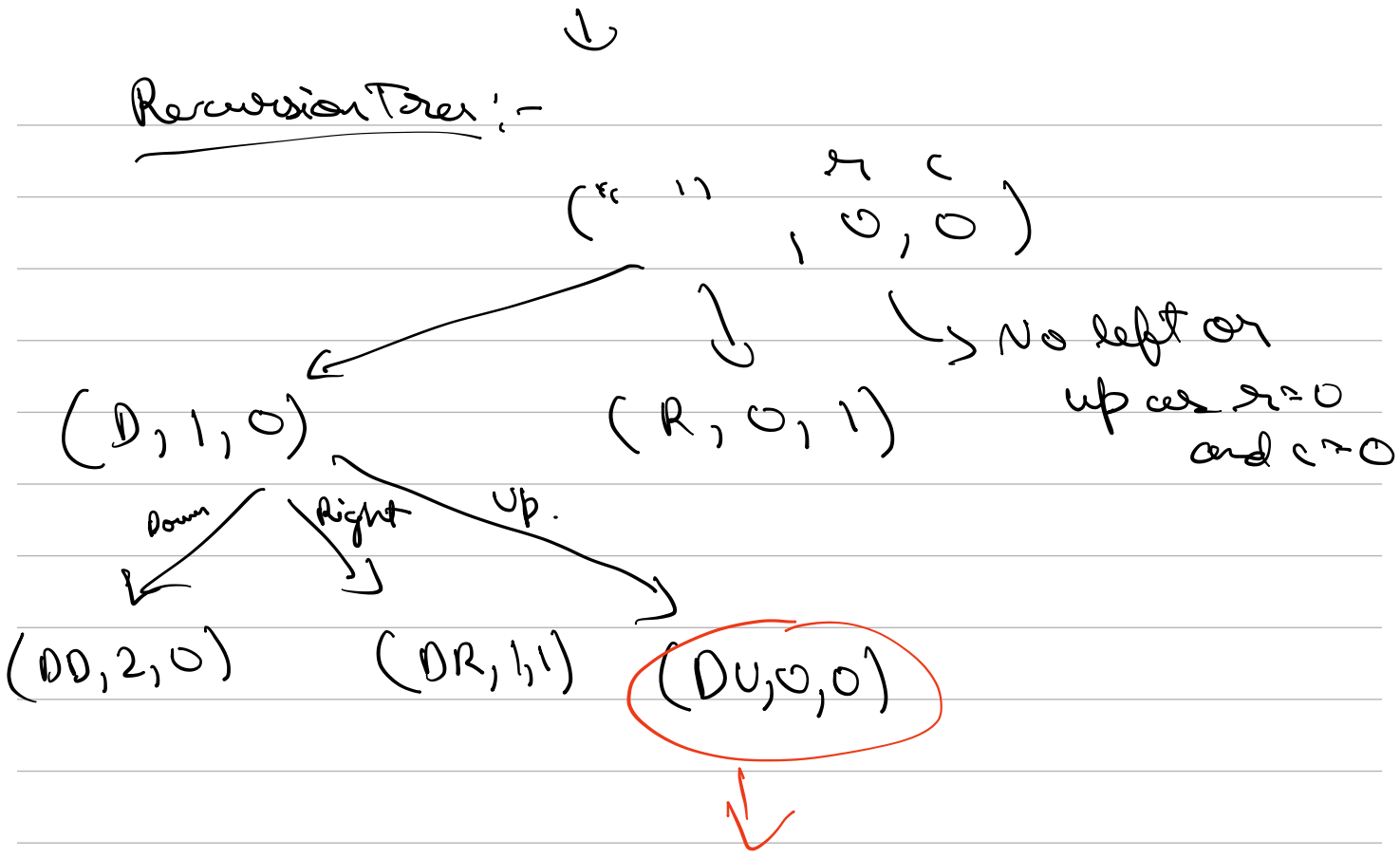
```
if (r < maze.length - 1) {
    allPath(p:p + 'D', maze, r:r+1, c);
}

if (c < maze[0].length - 1) {
    allPath(p:p + 'R', maze, r, c:c+1);
}

if (r > 0) {
    allPath(p:p + 'U', maze, r:r-1, c);
}

if (c > 0) {
    allPath(p:p + 'L', maze, r, c:c-1);
}
```

→ Moving down.
→ right
→ Up
→ Left.

* If you execute this :- It gives **error.**

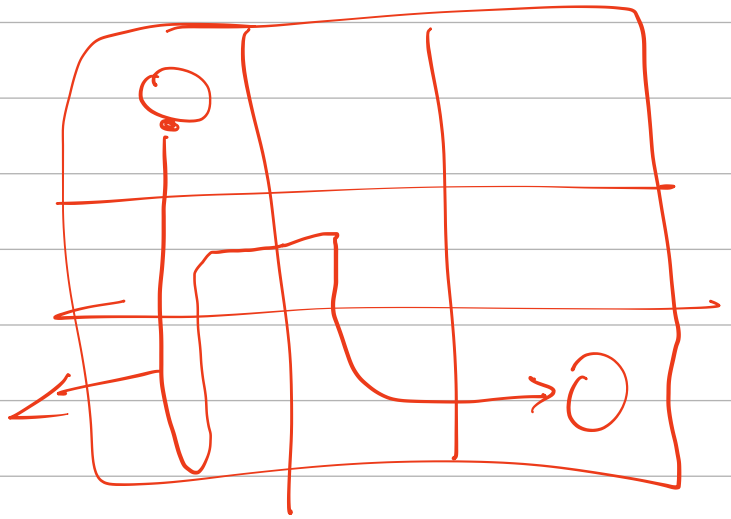
because Stack overflow, because no **backtracking** is there here, each cell is infinitely visited and path is traced.

Recursion Tree :-



Here again we came back to (0,0). so this recursion is never ending.

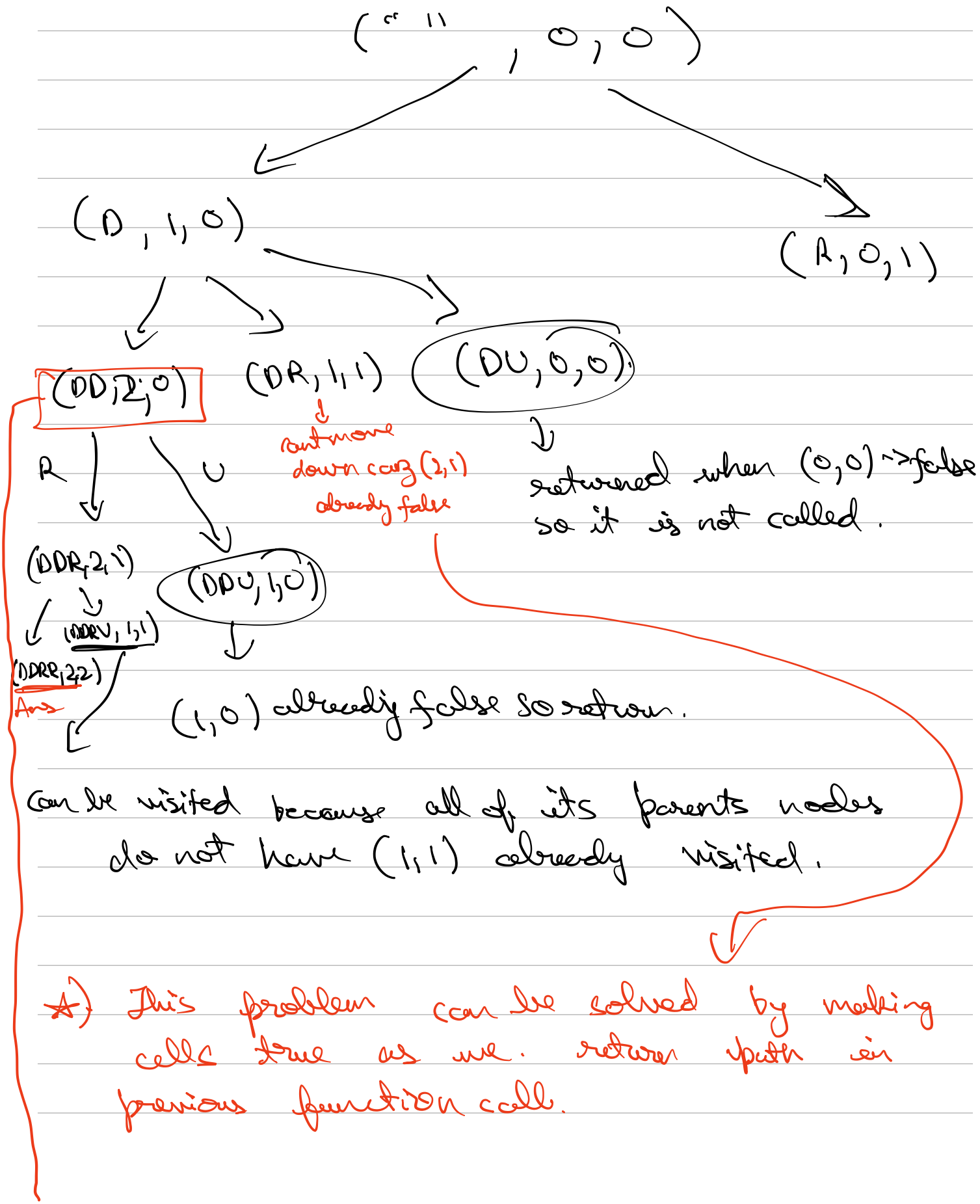
NOTE:- We cannot move back at same path.



This moving back is not allowed.

☆ Mark the visited cells as false

So that they are not visited again.



Can be visited because all of its parents nodes do not have (1,1) already visited.

☆) This problem can be solved by making cells true as we return path in previous function call.

i.e.

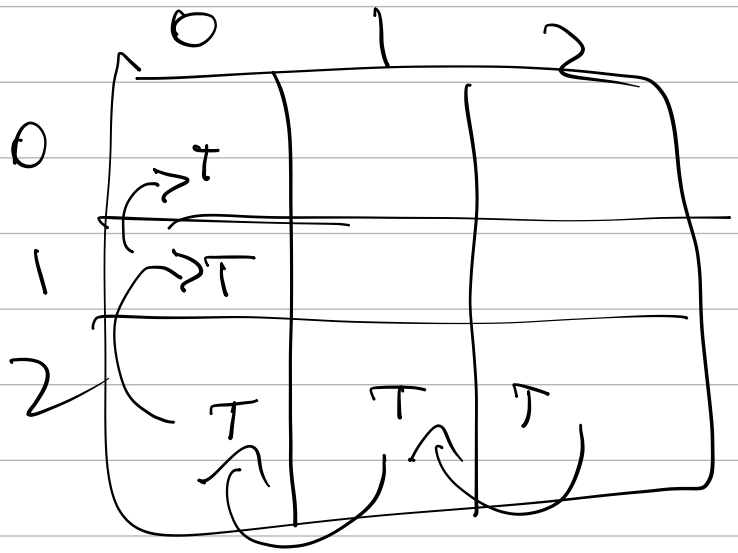
→ By the time the below function calls of this return, they make their respective cells as true.

Only the final answer (2,2) will print. path. (DDRR)

Above function calls. make their cell (r,c) = true if mat[r][c] == false

Example: If you are in (2,2) mark it as true return path. / (2,1) -> Mark true

(2,0) -> true



This is known as backtracking

→ When we go outside the recursion calls, change the changes we made previously.

Eg:- We were making cells false initially; while returning back making them true.

CODE:-

```java
public class AllPaths {
    public static void main(String[] args) {
        boolean[][] board = {
                {true, true, true},
                {true, true, true},
                {true, true, true}
        };

        allPath(p: "", board, r: 0, c: 0);
    }

    static 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) {
            allPath(p: p + 'D', maze, r: r+1, c);
        }

        if (c < maze[0].length - 1) {
            allPath(p: p + 'R', maze, r, c: c+1);
        }

        if (r > 0) {
            allPath(p: p + 'U', maze, r: r-1, c);
        }

        if (c > 0) {
            allPath(p: p + 'L', maze, r, c: c-1);
        }

        // this line is where the function will be over
        // so before the function gets removed, also remove the changes tha
        maze[r][c] = true;
    }
}
```

path printed only if (2,2).

returning if false (already visited)

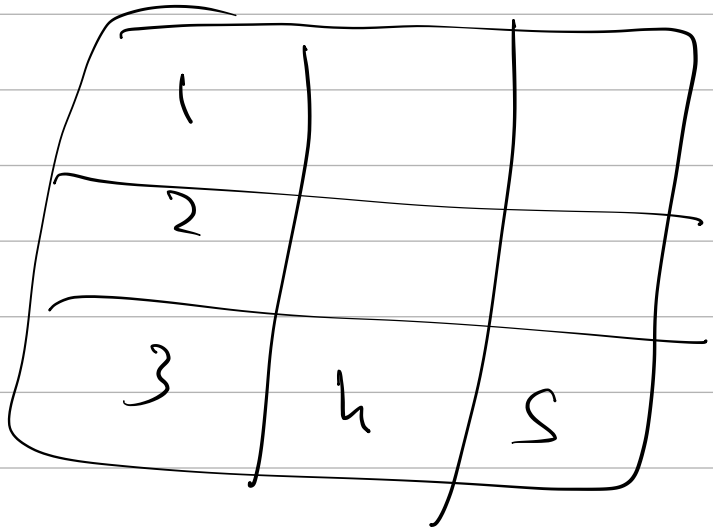
making it false initially.

making it true at end.
(this true is done after all the above function calls return).

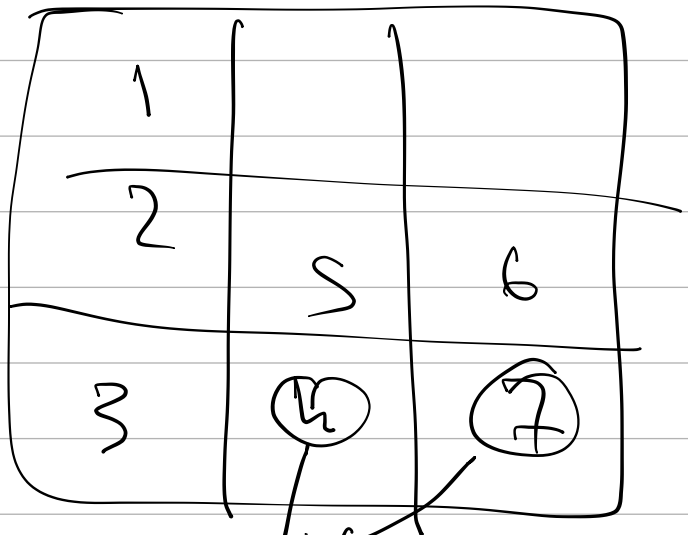
Above is the code for backtracking.

Q). Print the matrix and Path.

Sample:-



DDD R R



DDDRVR
D.

☆). Take a step variable.

☆). Update path array.

☆). Print the array in base condition.

☆). Backtrack.

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

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

    // i am considering this block in my path
    maze[r][c] = false;
    path[r][c] = step;
    if (r < maze.length - 1) {
        allPathPrint(p: p + 'D', maze, r: r+1, c, path, step: step+1);
    }

    if (c < maze[0].length - 1) {
        allPathPrint(p: p + 'R', maze, r, c: c+1, path, step: step+1);
    }

    if (r > 0) {
        allPathPrint(p: p + 'U', maze, r: r-1, c, path, step: step+1);
    }

    if (c > 0) {
        allPathPrint(p: p + 'L', maze, r, c: c-1, path, step: step+1);
    }

    // this line is where the function will be over
    // so before the function gets removed, also remove the changes tha
    maze[r][c] = true;
    path[r][c] = 0;
}
}
```

→ for keeping in track of visited cells.

→ the last cell also will be having step no.

→ Printing path.

→ for getting newline after each path matrix.

} making changes.

} reversing changes.

As we backtrack the step entered cell is set back to 0.

*). Here path[r][c] matrix contains path from A to B.

Main function.

```java
public class AllPaths {
    public static void main(String[] args) {
        boolean[][] board = {
                {true, true, true},
                {true, true, true},
                {true, true, true}
        };
        int[][] path = new int[board.length][board[0].length];
        allPathPrint( p: "", board, r: 0, c: 0, path, step: 1);
    }
```

Output :-

→ A.

```
[1, 0, 0]
[2, 0, 0]
[3, 4, 5]
DDRR

[1, 0, 0]
[2, 5, 6]
[3, 4, 7]
DDRURD

[1, 6, 7]
[2, 5, 8]
[3, 4, 9]
DDRUURDD

[1, 0, 0]
```

→ B