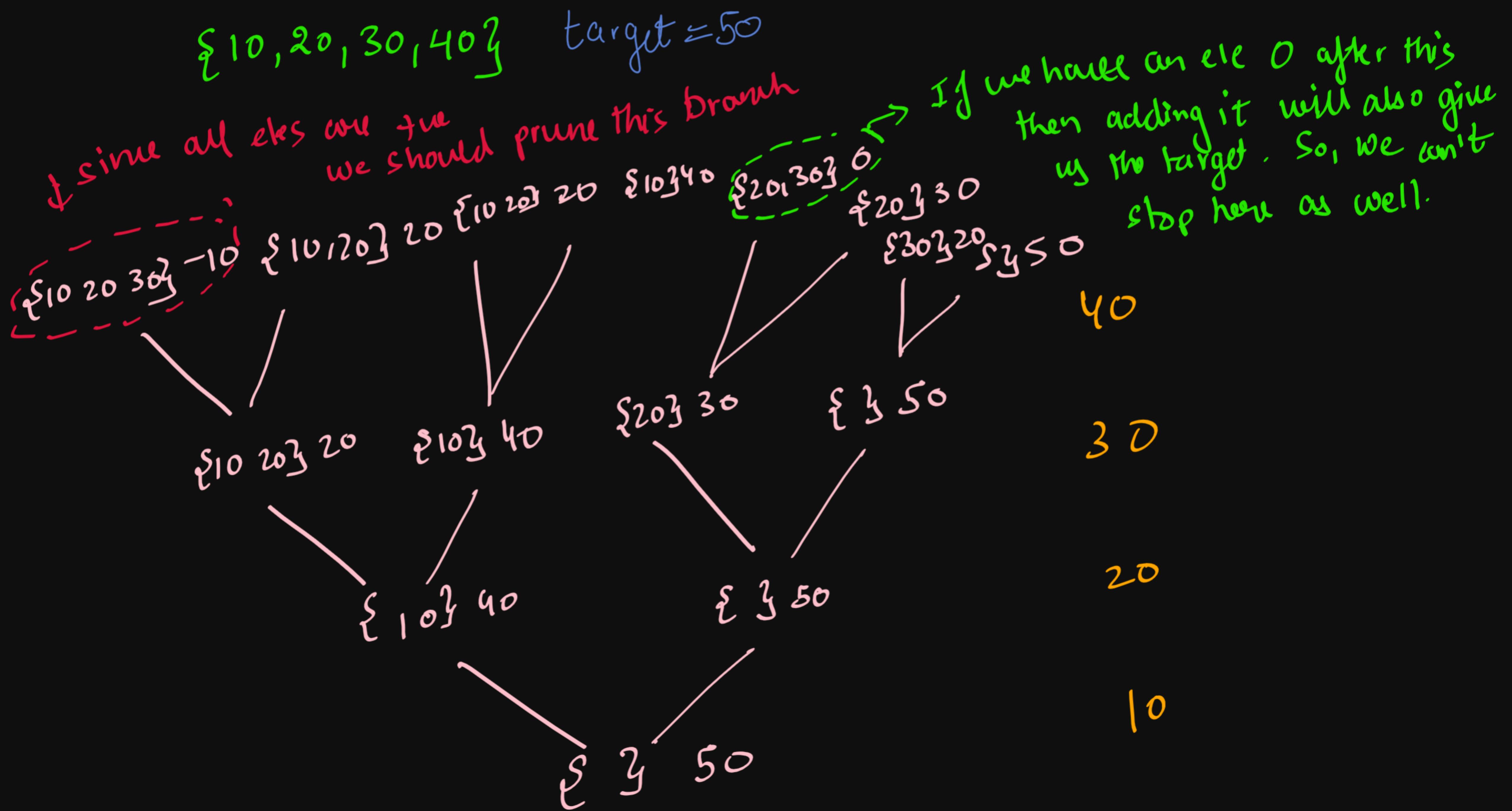


Recursion & Backtracking

Target Sum Subset (Not a backtracking problem)
Simple Rec Problem



Pruning is very imp in this ques-

→ Negative base case & pruning is diff

↓
if we don't take
care of this then
stack overflow will
occur as recursion
will keep on going

↓
if we don't take
care of this then recursion
is eventually going to stop
but a lot of extra calls
might be made-

```
public static void printTargetSumSubsets(int[] arr, int idx, String set, int remTarget, int tar) {  
  
    if(idx == arr.length) {  
        if(remTarget == 0) {  
            System.out.println(set + ".");  
        }  
  
        return;  
    }  
  
    //pruning  
    if(remTarget < 0) {  
        return;  
    }  
  
    printTargetSumSubsets(arr, idx+1, set + arr[idx] + ", ", remTarget - arr[idx], tar);  
    printTargetSumSubsets(arr, idx+1, set, remTarget, tar);  
}
```

flood fill

1	1	1	1	1
0	0	0	0	1
1	0	1	0	1
1	0	0	0	1
1	0	1	0	0

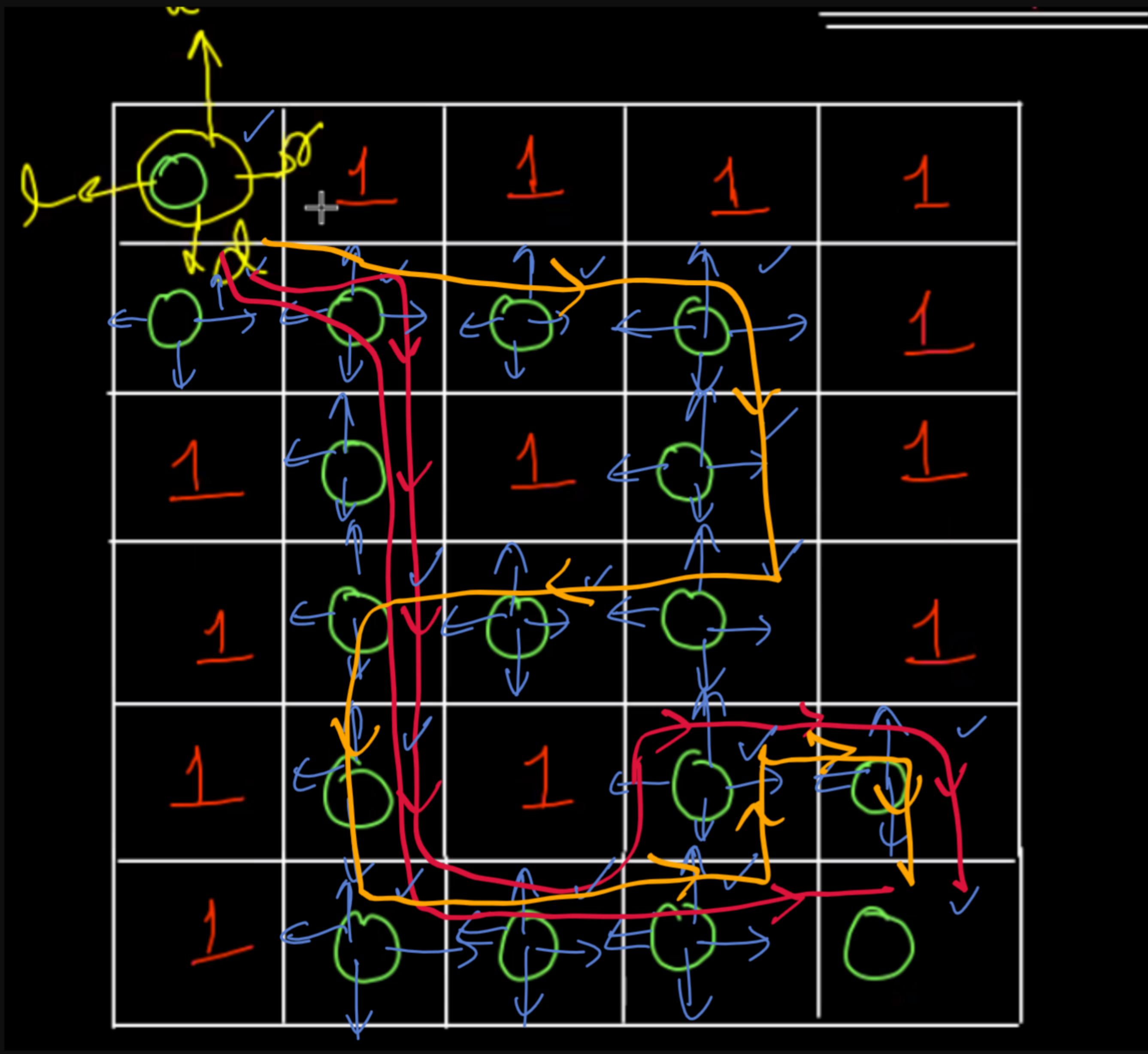
↓ This code will cause stack overflow as we keep on visiting same node in our endless cycle -

```
// asf -> answer so far
public static void floodfill(int[][] maze, int sr, int sc, String asf) {
    if(sr > dr || sc > dc || sr < 0 || sc < 0 || maze[sr][sc] == 1){
        // negative base case
        return;
    }

    if(sr == dr && sc == dc){
        // positive base case
        System.out.println(asf);
        return;
    }

    floodfill(sr - 1, sc, dr, dc, asf + "t"); // top
    floodfill(sr, sc - 1, dr, dc, asf + "l"); // left
    floodfill(sr + 1, sc, dr, dc, asf + "d"); // down
    floodfill(sr, sc + 1, dr, dc, asf + "r"); // right
}
```

So, we need backtracking now.



tl;dr → order of calls

```

public static void floodfill(int[][] maze, int sr, int sc, String psf, boolean[][] visited){
    int dr = maze.length-1;
    int dc = maze[0].length-1;

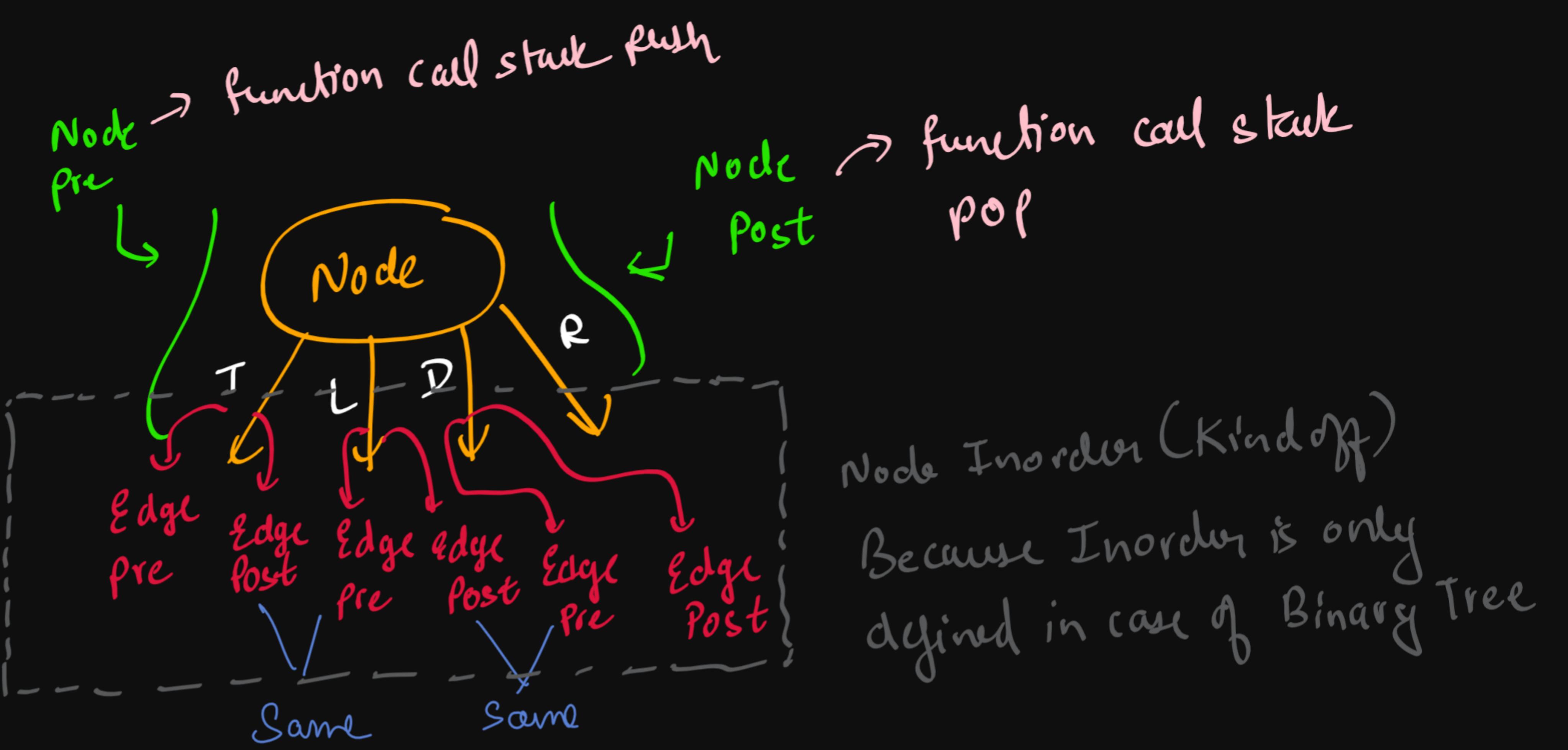
    if(sr > dr || sc > dc || sr < 0 || sc < 0 || maze[sr][sc] == 1 || visited[sr][sc] == true) {
        return;
    }
    visited[sr][sc] = true; → Node Pre

    if(sr == dr && sc == dc) {
        System.out.println(psf);
        visited[sr][sc] = false;
        return;
    }

    floodfill(maze,sr-1,sc,psf + "t",visited);
    floodfill(maze,sr,sc-1,psf + "l",visited);
    floodfill(maze,sr+1,sc,psf + "d",visited);
    floodfill(maze,sr,sc+1,psf + "r",visited);

    visited[sr][sc] = false; → Node Post
}

```



When can we apply Recursion & Backtracking (when NOT to apply DP)

- Print all paths / configs | P&C
- Count all paths / configs | P&C (DP / Greedy) → in 90% cases

Rec & Back Solns are generally exponential time soln.
So, small constraints like $N \leq 20 \text{ to } 30$

N Queens

1. You are given a number n , the size of a chess board.
2. You are required to place n number of queens in the $n * n$ cells of board such that no queen can kill another.
Note - Queens kill at distance in all 8 directions
3. Complete the body of printNQueens function - without changing signature - to calculate and print all safe configurations of n -queens.
Use sample input and output to get more idea.

Note -> The online judge can't force you to write the function recursively but that is what the spirit of question is. Write recursive and not iterative logic. The purpose of the question is to aid learning recursion and not test you.

2 possible configurations for 4×4 Matrix

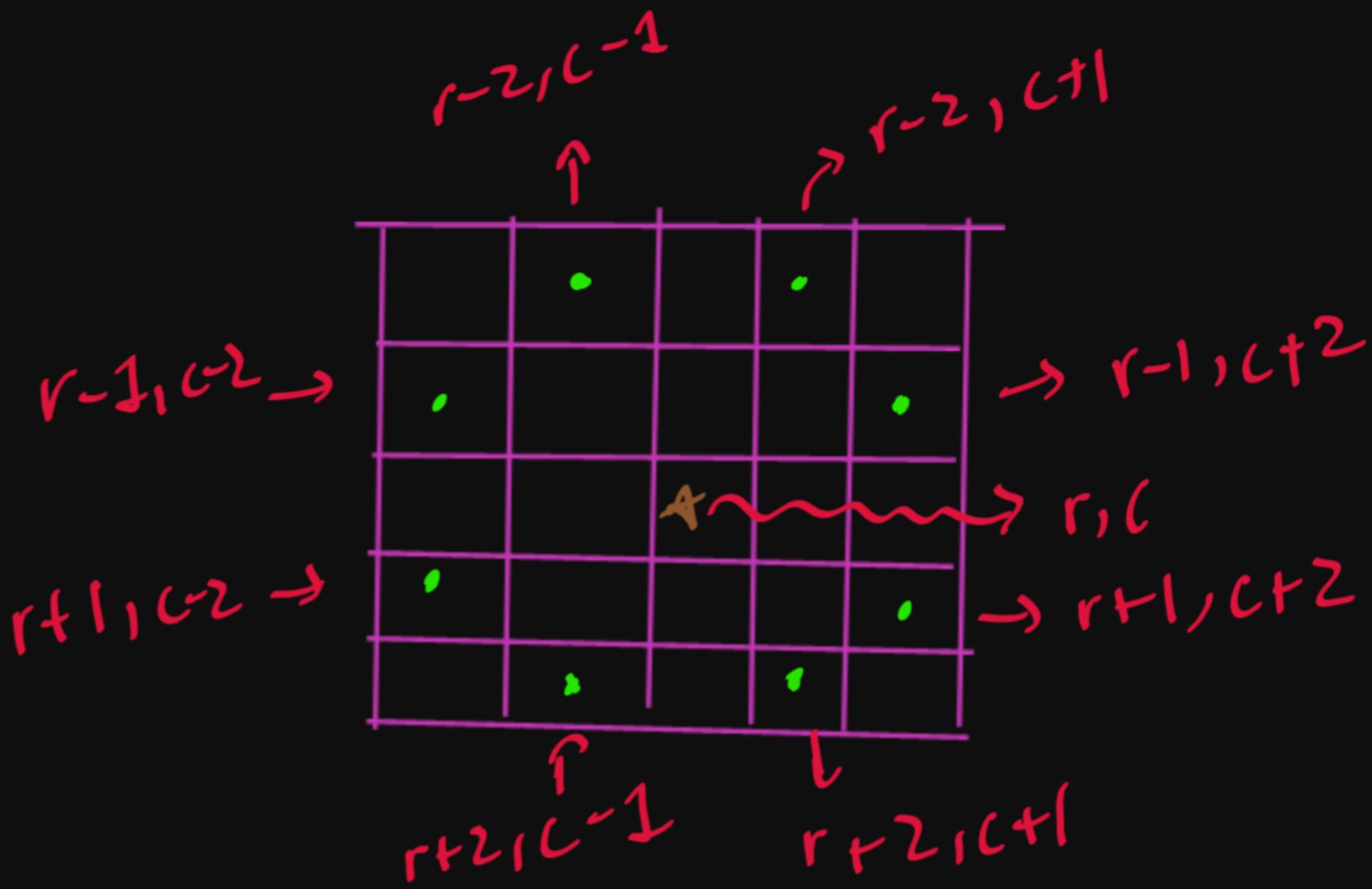
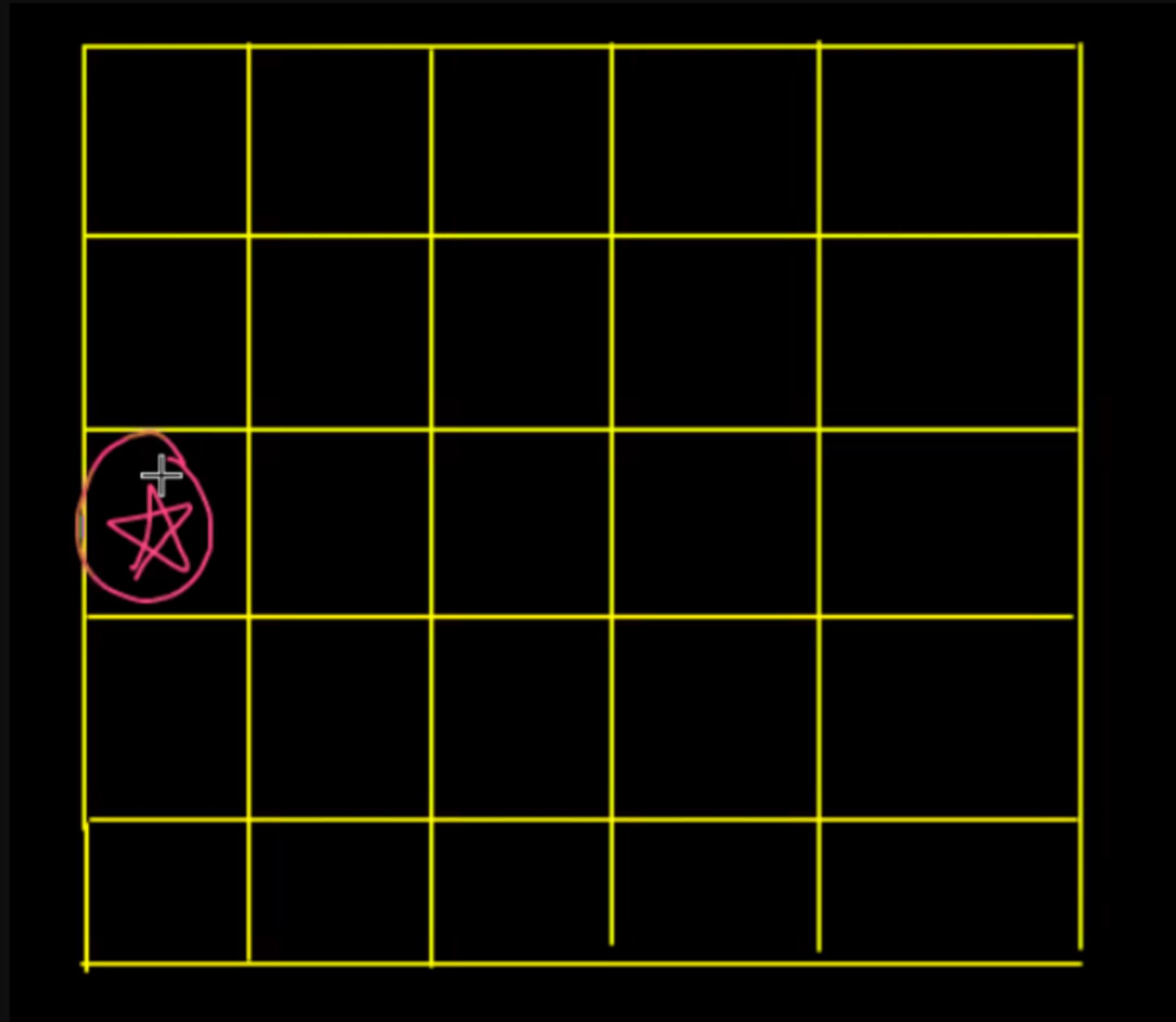
0	1	2	3
0	*		
1			*
2	*		
3		*	

0	1	2	3
0			*
1	*		
2			
3		*	

Level (parameter)
option (call/faith)

N Queens Dry Run

Knights' Tour



```
public static void printKnightsTour(int[][] chess, int r, int c, int upcomingMove) {  
    if(r < 0 || c < 0 || r>= chess.length || c>= chess.length || chess[r][c] > 0) return;  
  
    chess[r][c] = upcomingMove;  
    if(upcomingMove == chess.length * chess.length) {  
        displayBoard(chess);  
        chess[r][c] = 0;  
        return;  
    }  
  
    printKnightsTour(chess,r-2,c+1,upcomingMove + 1);  
    printKnightsTour(chess,r-1,c+2,upcomingMove + 1);  
    printKnightsTour(chess,r+1,c+2,upcomingMove + 1);  
    printKnightsTour(chess,r+2,c+1,upcomingMove + 1);  
    printKnightsTour(chess,r+2,c-1,upcomingMove + 1);  
    printKnightsTour(chess,r+1,c-2,upcomingMove + 1);  
    printKnightsTour(chess,r-1,c-2,upcomingMove + 1);  
    printKnightsTour(chess,r-2,c-1,upcomingMove + 1);  
    chess[r][c] = 0;  
}
```