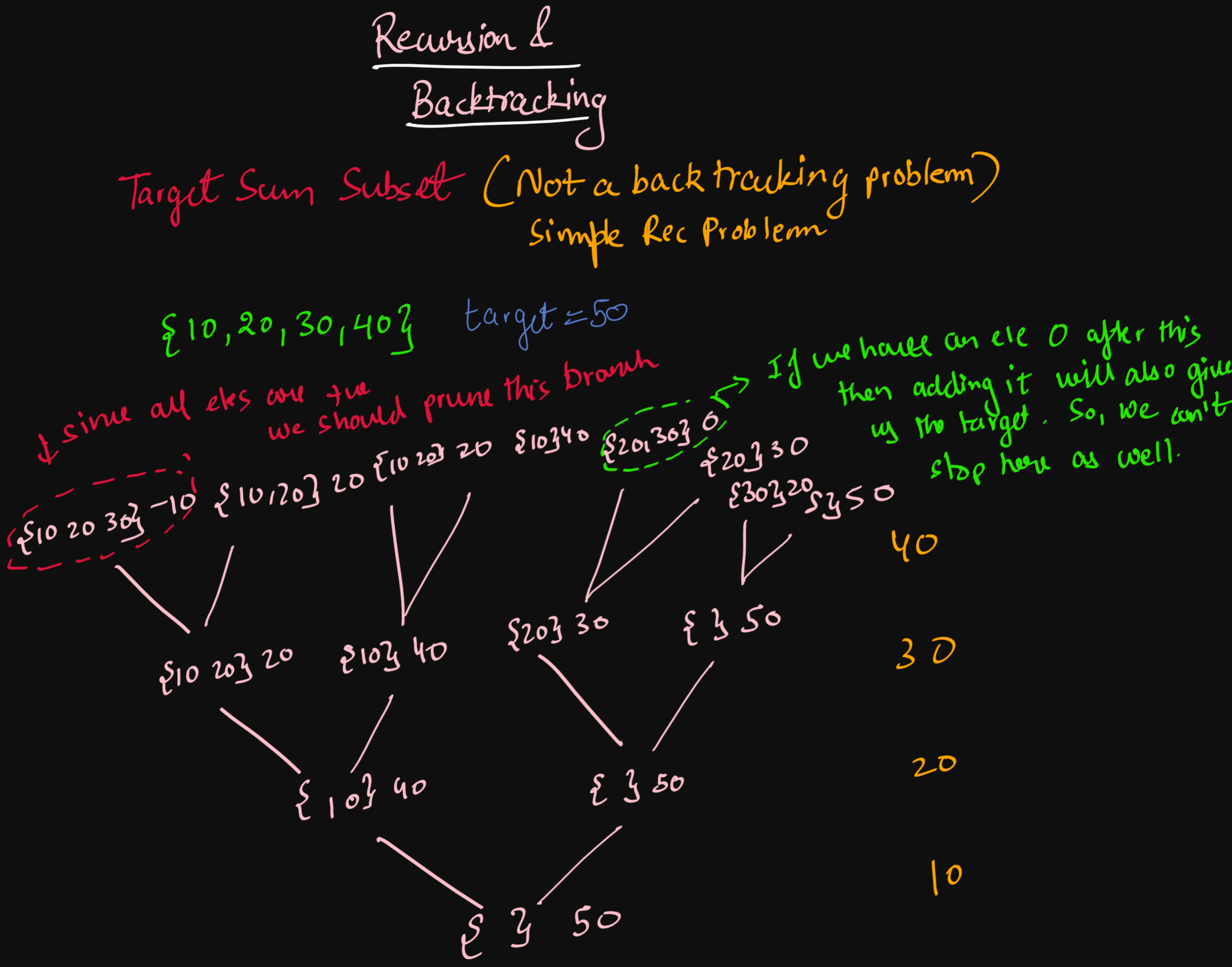
Recursion & Backtracking

Target Sum Subset (Not a backtracking problem)

Simple Rec Problem

{10, 20, 30, 40} target = 50



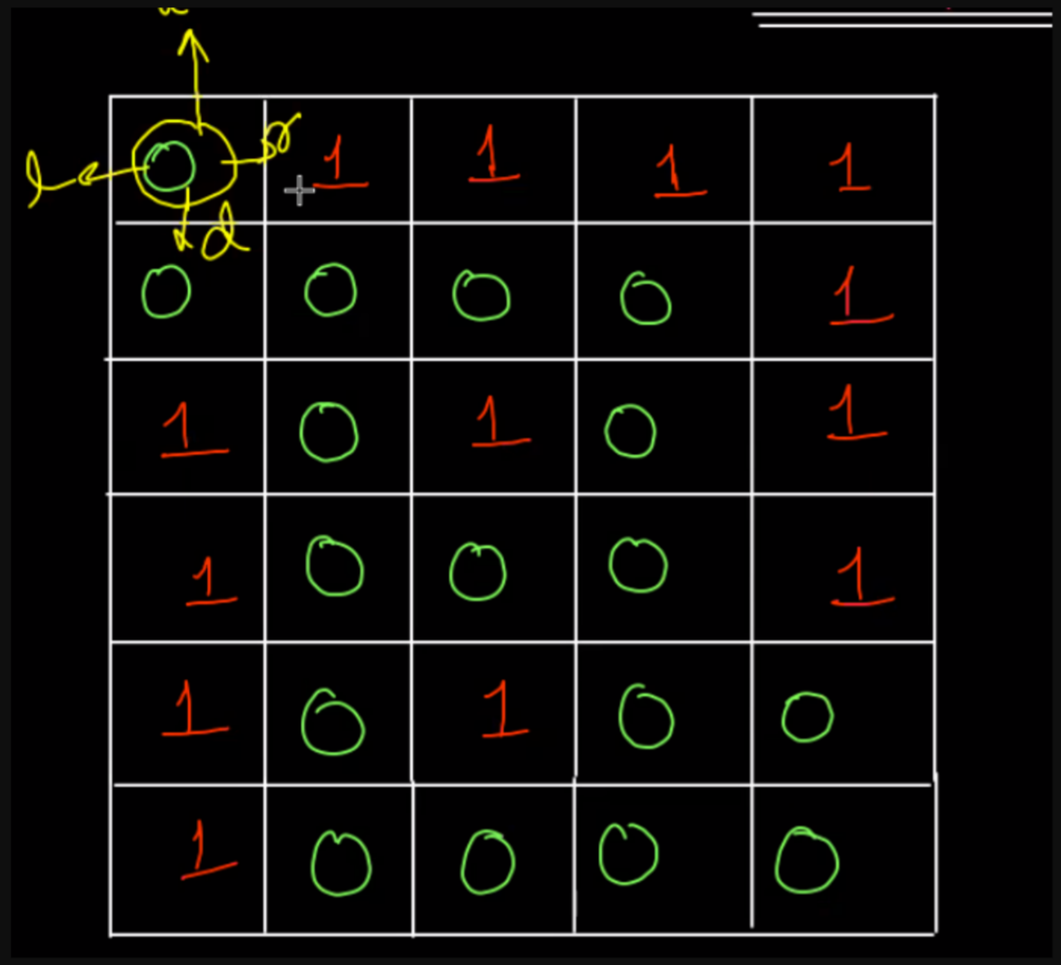
Pruning is very imp in this ques.

→ Negative base case & pruning is diff

Negative base case	Pruning
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.

```java
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);
}
```

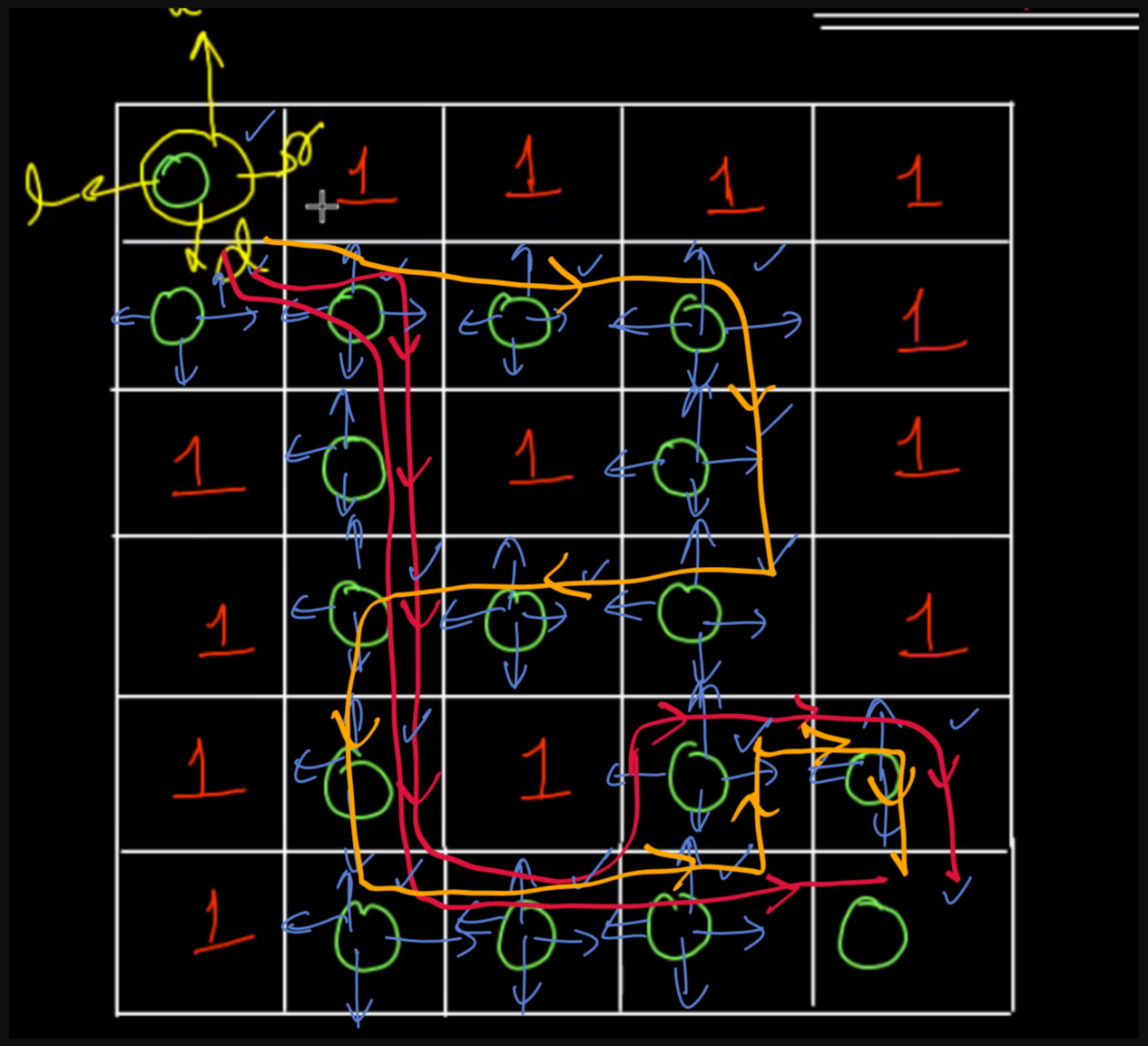
flood fill



↓ This code will cause stack overflow as we end up on visiting same node in an endless cycle.

```java
// 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(psf);
        return;
    }

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

}
```

So, we need backtracking now.

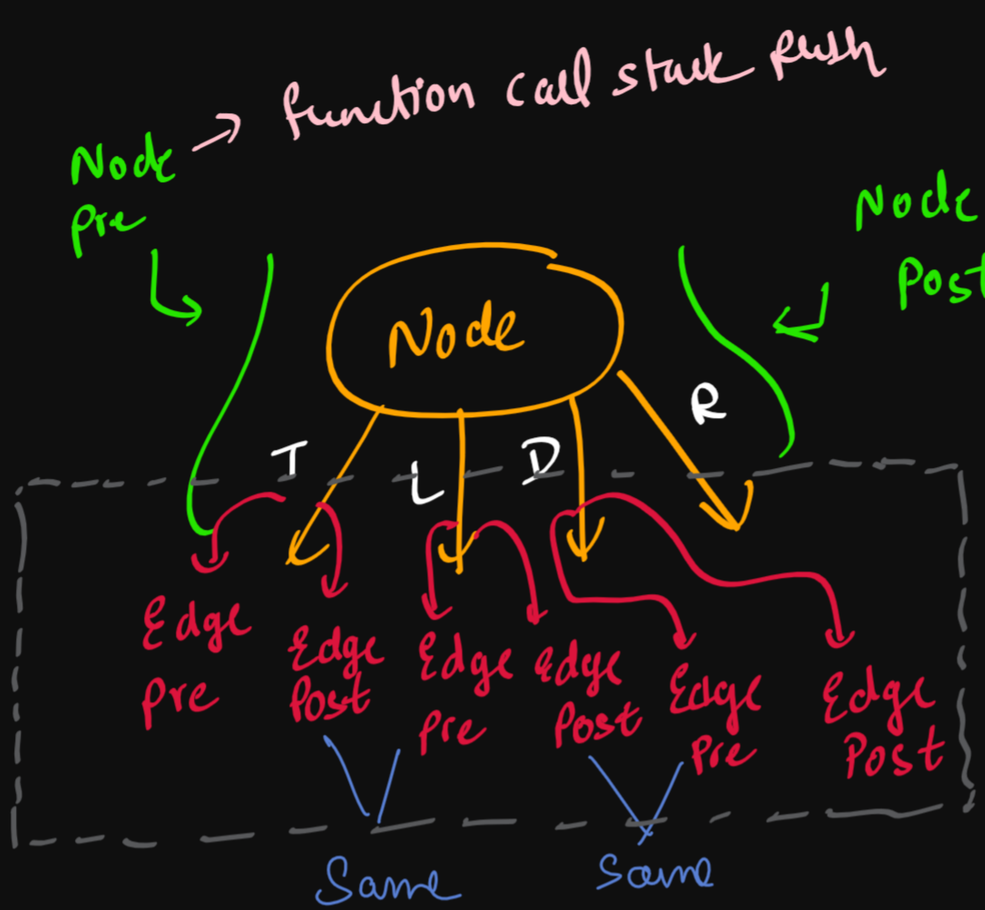


tldr → order of calls

```java
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;

    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 Pre

→ Node Post



Node Inorder (Kind of)
Because Inorder is only defined in case of Binary Tree

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

→ Print all paths / configs / PLC

→ Count all paths / configs / PLC (DP / Greedy) → in 90 % cases

Rec & Back Soln are generally exponential time soln.

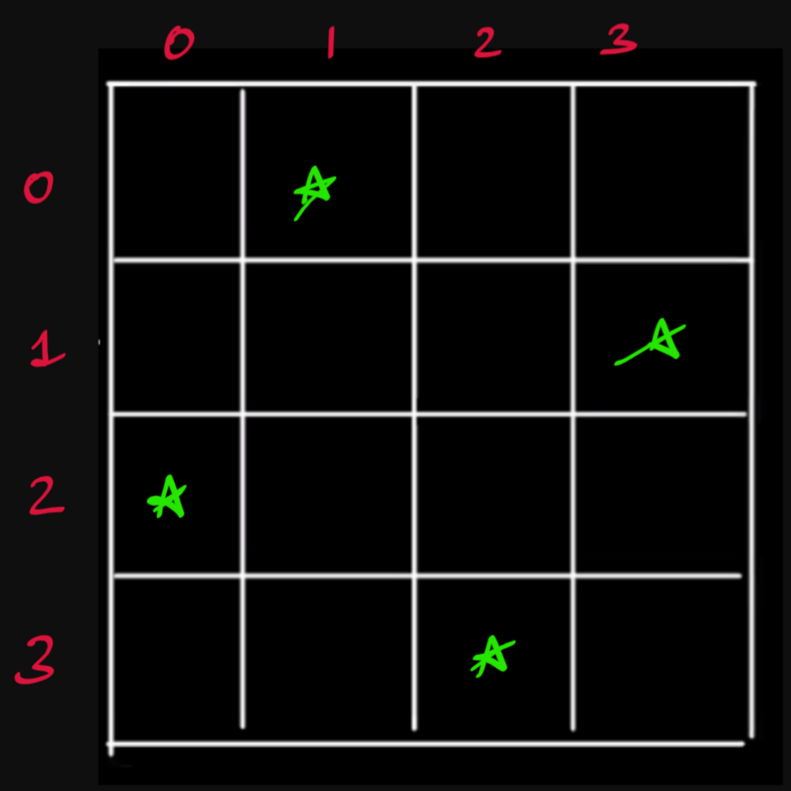
So, small constraints like $N \leq 20$ & 30

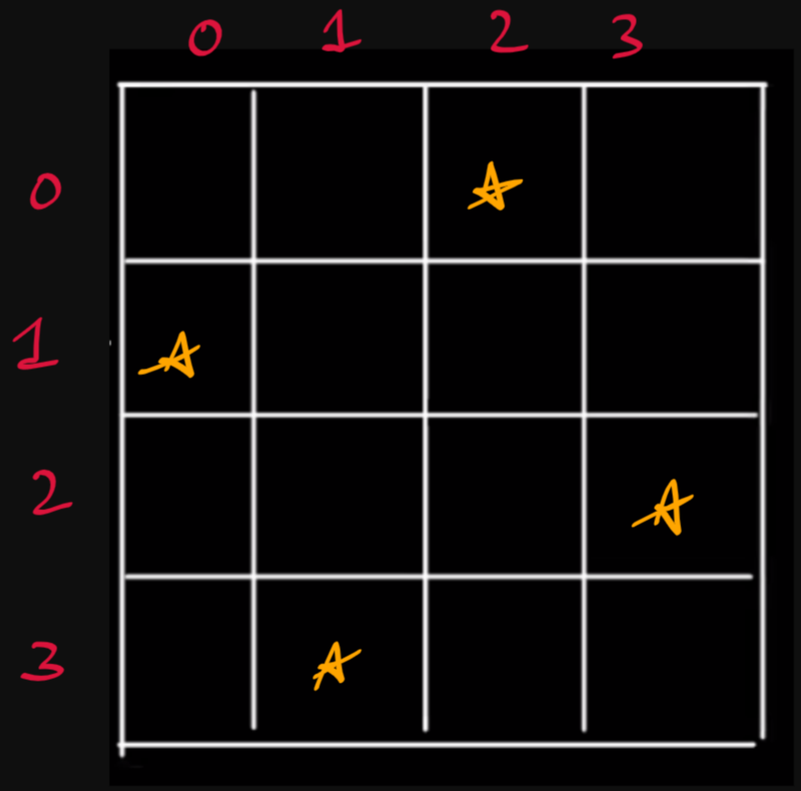
N Queens

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.

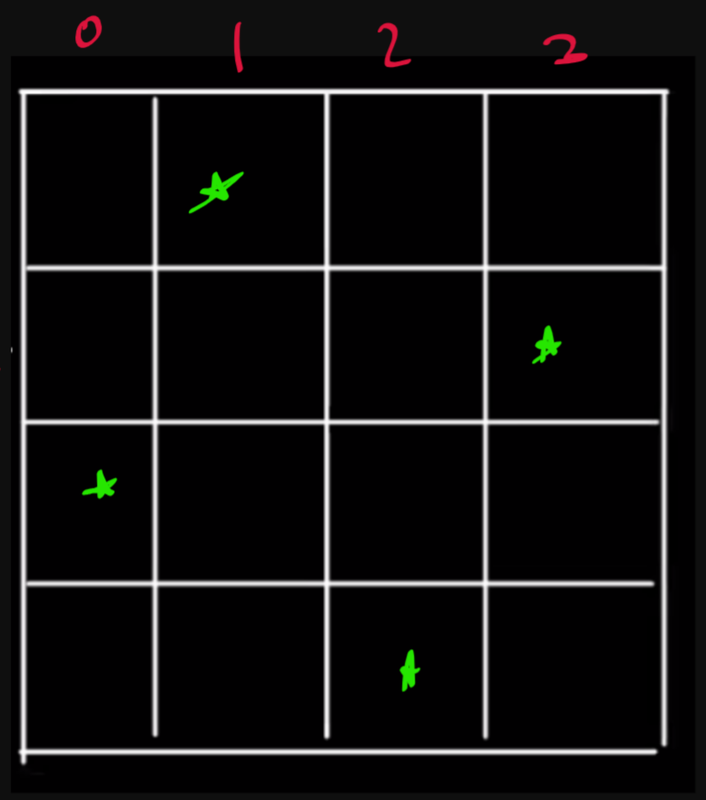
2 possible configurations for 4x4 Matrix

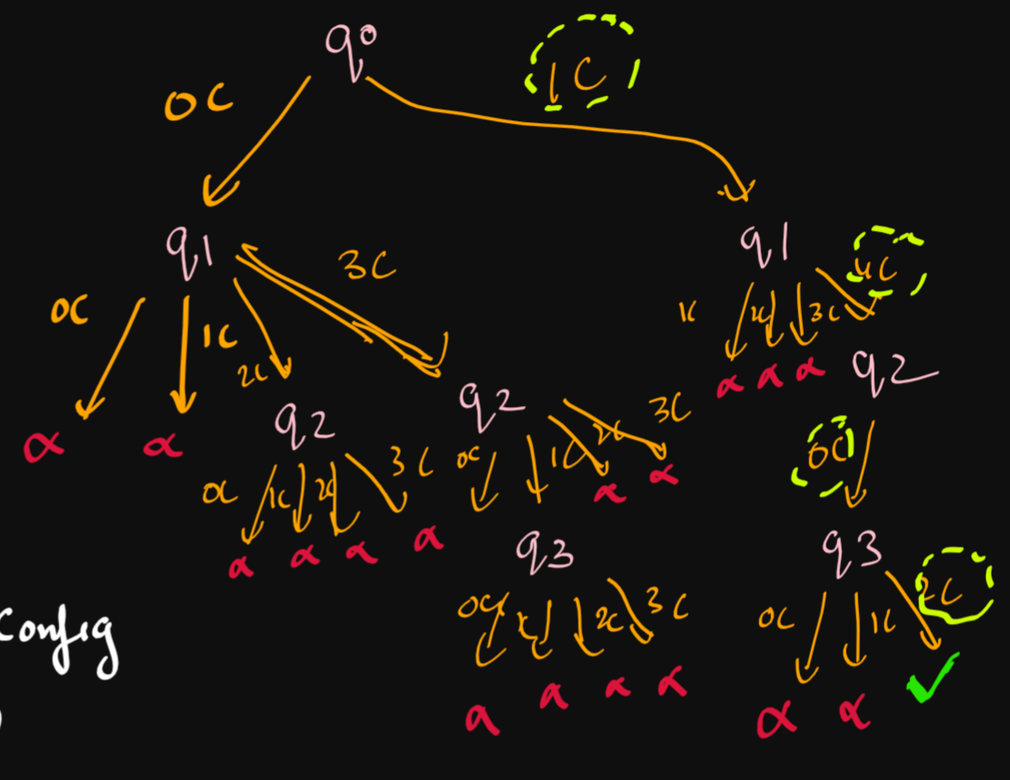




level (parameter)
option (calls/ faith)

N Queens Dry Run





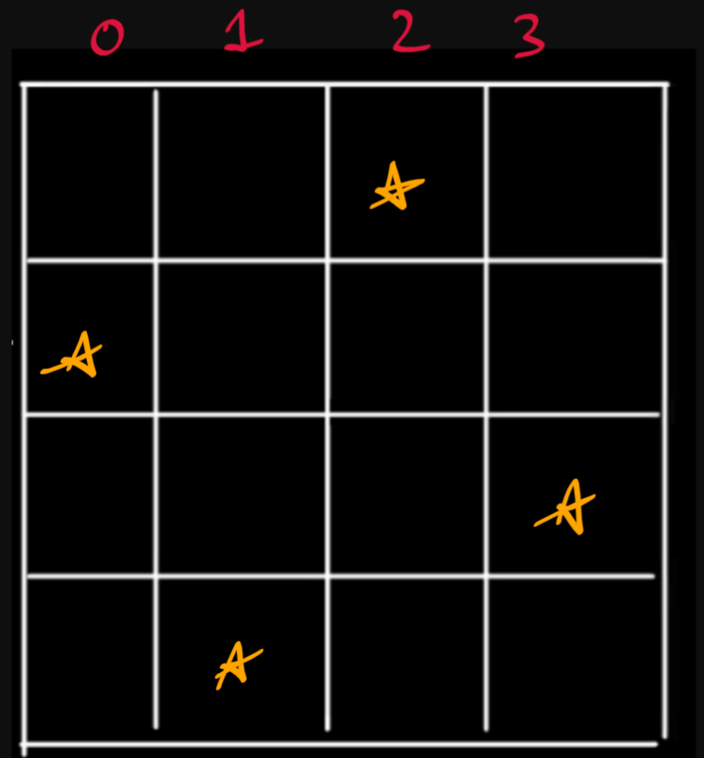
→ First Valid Config

$q_0 \rightarrow 1C$ (r_0)

$q_1 \rightarrow 4C$ (r_1)

$q_2 \rightarrow 0C$ (r_2)

$q_3 \rightarrow 2C$ (r_3)



Why we can find the second valid config

$q_0 \rightarrow 2C$

$q_1 \rightarrow 0C$

$q_2 \rightarrow 4C$

$q_3 \rightarrow 1C$

```java
public static void printNQueens(boolean[][] chess, String qsf, int row) {

    if(row == chess.length) {
        System.out.println(qsf + ".");
        return;
    }

    for(int col=0;col<chess[0].length;col++) {
        if(isQueenSafe(chess,row,col) == true) {
            chess[row][col] = true; //edge pre
            //call
            printNQueens(chess,qsf + row + "-" + col + ", ",row + 1);
            chess[row][col] = false; //edge post
        }
    }
```

The faith is that we will place the first queen in the first row ourselfs. Rest all the queens will be placed in the next rows themselves.

```java
public static boolean isQueenSafe(boolean[][] chess,int row, int col) {

    //upward col

    int i = 0;
    while(i < row) {
        if(chess[i][col] == true) return false;
        i++;
    }

    //upward left diagonal
    i = row;
    int j = col;
    while(i >= 0 && j >= 0) {
        if(chess[i][j] == true) return false;
        i--;
        j--;
    }

    //upward right diagonal
    i = row;
    j = col;
    while(i >= 0 && j< chess[0].length) {
        if(chess[i][j] == true) return false;
        i--;
        j++;
    }

    return true;
}
```

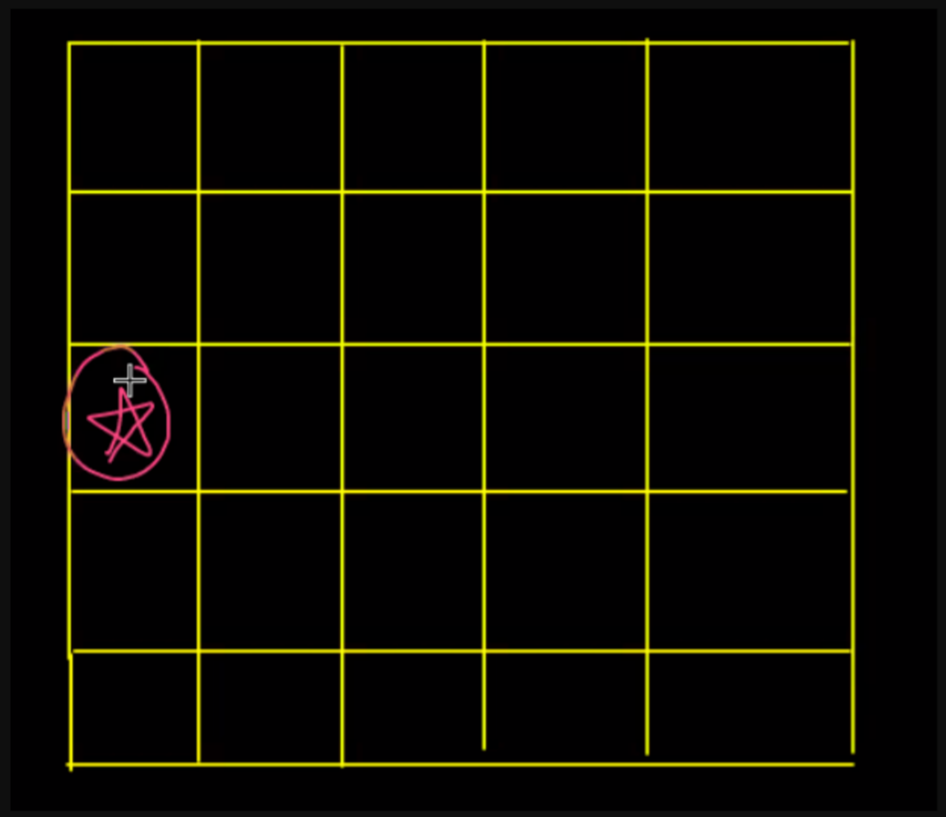
We are checking only in the above directions because we are placing the queens row-wise hence we are sure that that there is no queen below our current queen in the maze.

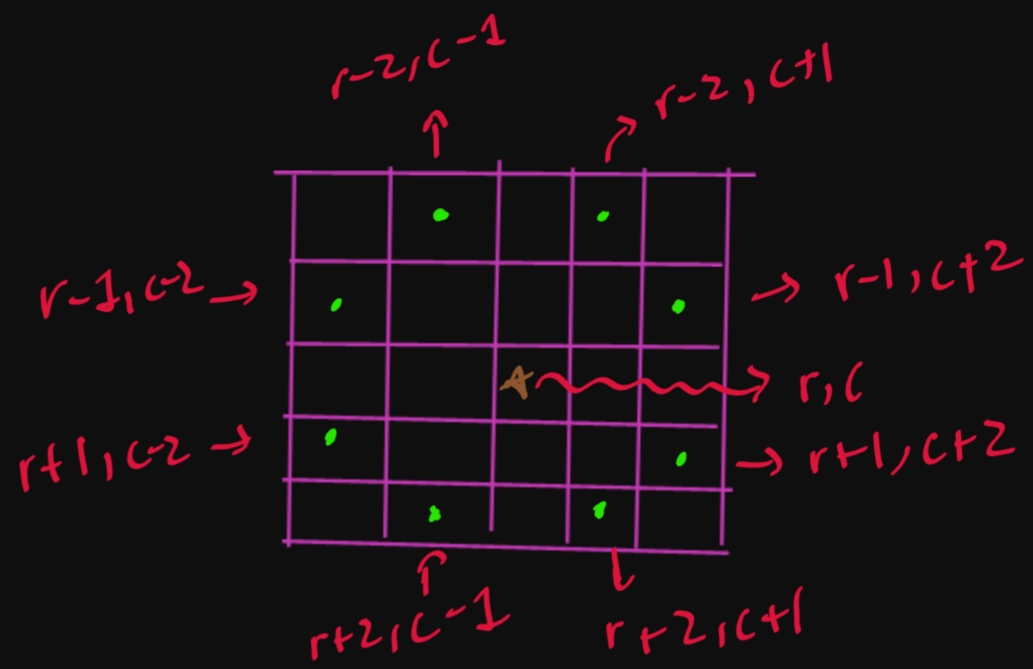
→ Each row will have exactly one queen

→ Each col will have exactly one queen

→ Out of all the left diagonals exactly N digonals will have a queen each in them & same applies for the right diagonals.

Knights Tour






```java
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;
}
```