Recursion & Backtracking

- Karun Karthik

Contents →

D1

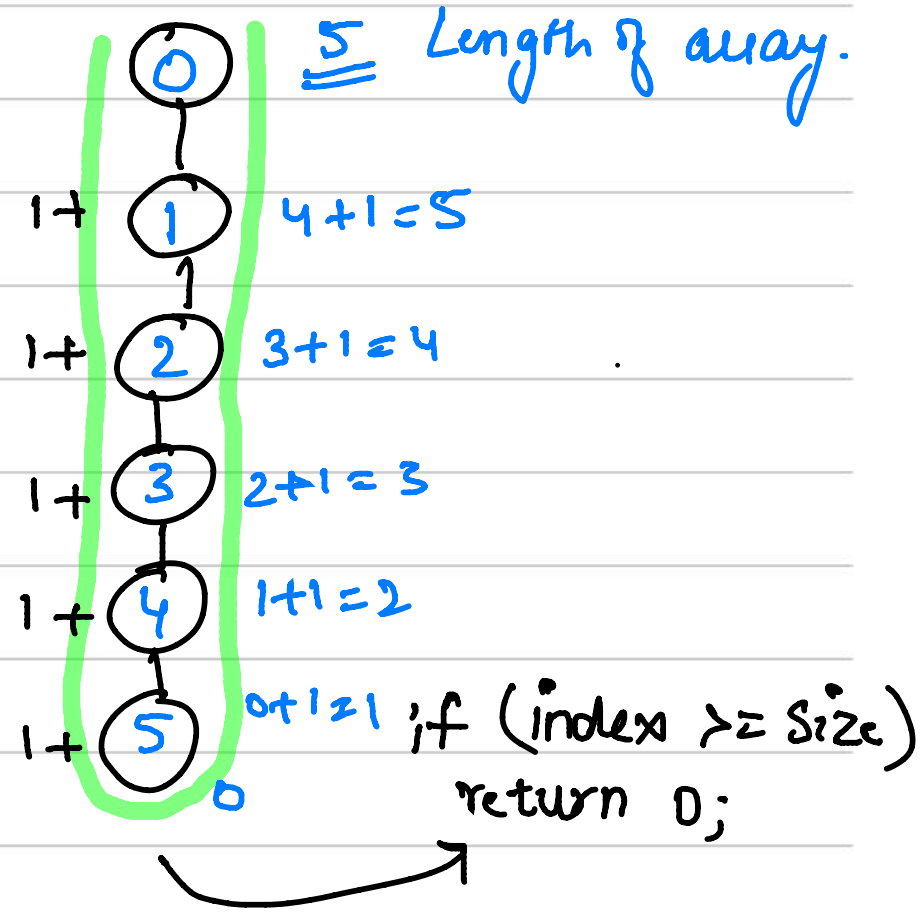
Recursion

① Length of an array

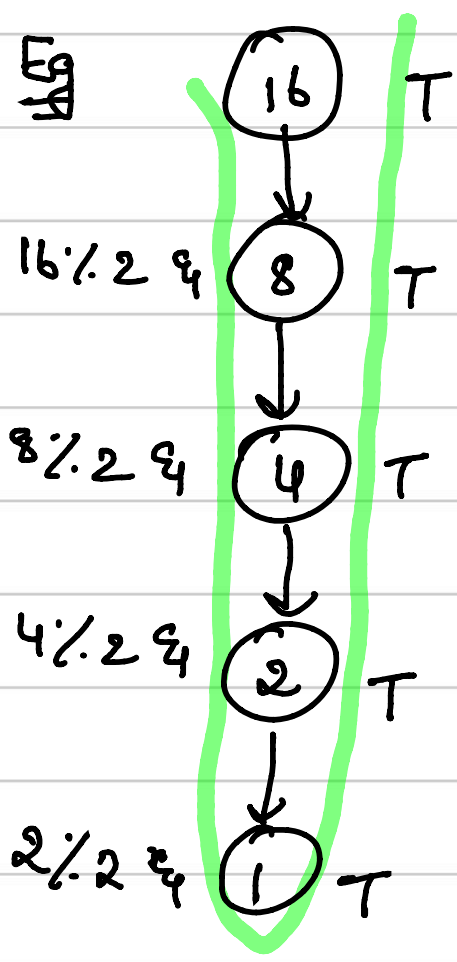
[20, 10, 40, 50, 30]
0 1 2 3 4

TC = O(n)

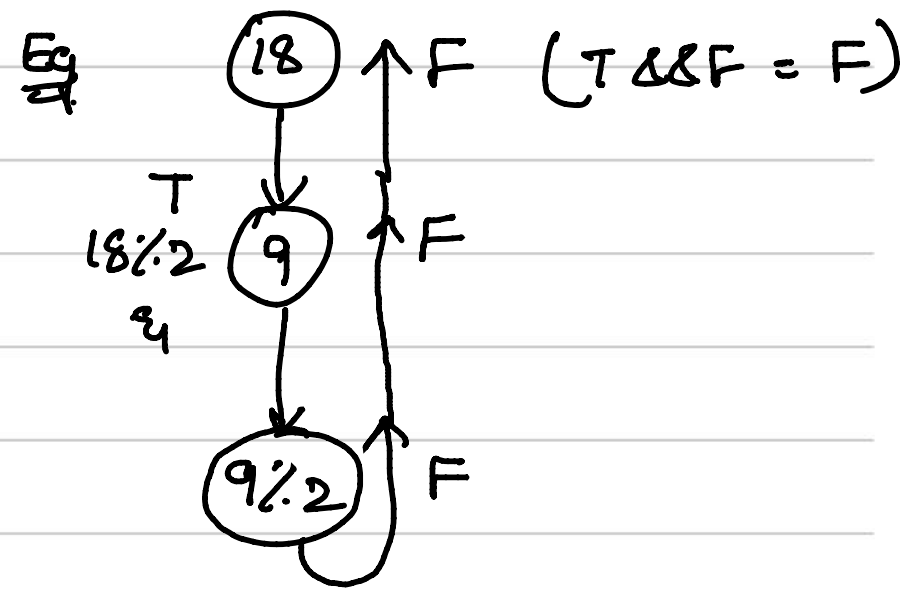
SC = O(n).



① Power of 2 $\longrightarrow 2^x = 2^0 \cdot 2^1 \cdot 2^2 \ldots 2^n$.



if 1 then
return true.



TC = O(log n)

Power of 2

```cpp
class Solution {
public:
    bool isPowerOfTwo(int n) {
        if(n==1) return true; //need to write it first else it might g
        if(n<=0 || n%2!=0) return false;
        return isPowerOfTwo(n/2);
    }
};
```

② Power of 3

```cpp
class Solution {
public:
    bool isPowerOfThree(int n) {
        if(n==1) return true; //need to write it first else it might
        if(n<=0 || n%3!=0) return false;
        return isPowerOfThree(n/3);
    }
};
```

③ Power of 4

```cpp
class Solution {
public:
    bool isPowerOfFour(int n) {
        if(n==1) return true; //need to write it first else it might g
        if(n<=0 || n%4!=0) return false;
        return isPowerOfFour(n/4);
    }
};
```

D2 Subsets

(4) Given an integer array nums, generate all the subsets. (subsequences)

If size = n then no. of subsets = 2^n.

Eg nums = [1, 2, 3]

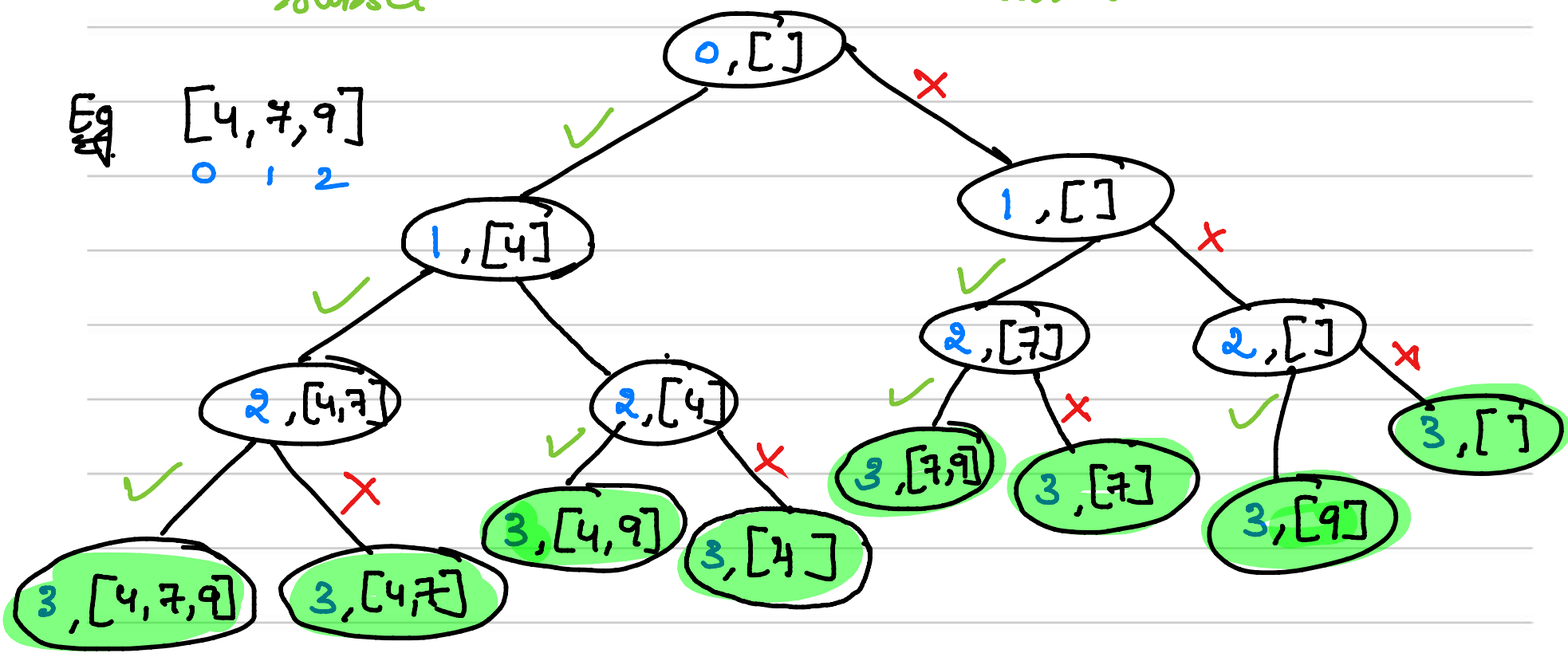
Sol = [[ ], [1], [1, 2], [1, 3], [1, 2, 3], [2], [2, 3], [3]]

For every, element

can be a part of subset

cannot be a part of subset.

Eg [4, 7, 9]
0 1 2



⇒ [[4, 7, 9], [4, 7], [4, 9], [4], [7, 9], [7], [9], [ ]].

* Once index is greater than or equal to size then store in result

Tc = O(2^n) → as there are 2 possibilities at every element.

Sc = O(2^n)

Code

```
class Solution {
public:
    void generateAllSubsets(vector<int>&nums, int currentIndex, vector<int>&res, vector<vector<int>> &powerSet){
        // base condition
        if(currentIndex >= nums.size()){
            powerSet.push_back(res);
            return;
        }
        int currentVal = nums[currentIndex];
        res.push_back(currentVal);
        generateAllSubsets(nums, currentIndex+1, res,powerSet);

        // remove the currentVal (not considering)
        res.pop_back();
        generateAllSubsets(nums, currentIndex+1, res,powerSet);
    }

    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> powerSet;
        vector<int> res;
        generateAllSubsets(nums, 0, res, powerSet);
        return powerSet;
    }
};
```

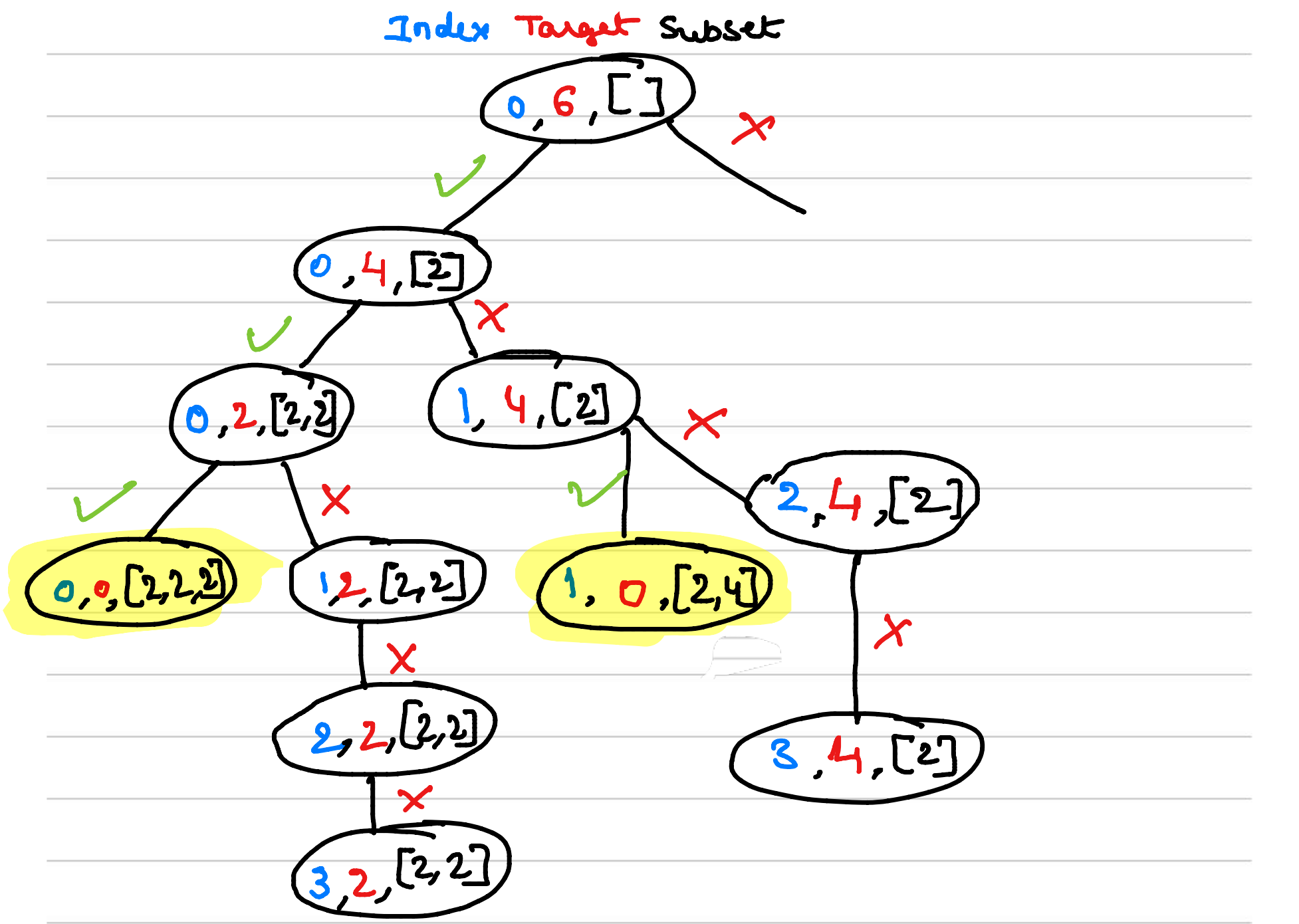
⑤ Combination Sum :-

nums = [2,3,5] (0, 1, 2) target = 8

Sol. [[2,2,2,2], [2,3,3], [3,5]]

Eg [2,4,5] (0,1,2)
target = 6

For every index ⇒ (CI, t) → (CI, t-nums[CI]) and (CI+1, t)



⇒ [[2,2,2], [2,4]].

* Store the result when target sum = 0

Code →

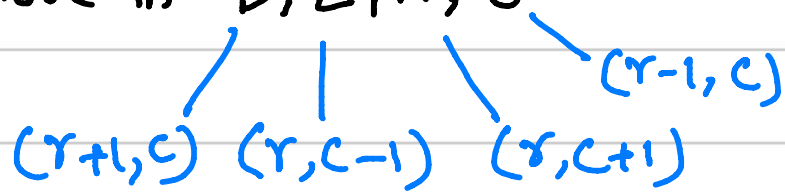
```
class Solution {
public:
    void totalWays(vector<int>&candidates, int target, int curr, vector<vector<int>>&res, vector<int>&aux ){
        if(curr==candidates.size()){
            if(target==0){
                res.push_back(aux);
            }
            return;
        }
        // feasible only if curr value is less than the target
        if(candidates[curr]<=target){
            aux.push_back(candidates[curr]);
            totalWays(candidates, target-candidates[curr], curr,res,aux);
            aux.pop_back();
        }
        // back-tracking
        totalWays(candidates, target, curr+1,res,aux);
    }

    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> res;
        vector<int> aux;
        totalWays(candidates, target, 0, res, aux);
        return res;
    }
};
```

D3

⑥ Rat in a maze

Generate all the ways to go from (0,0) to (n-1, n-1)

* At any cell (r,c) we can move in D, L, R, U

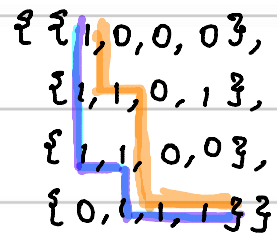


Eg: n = 3

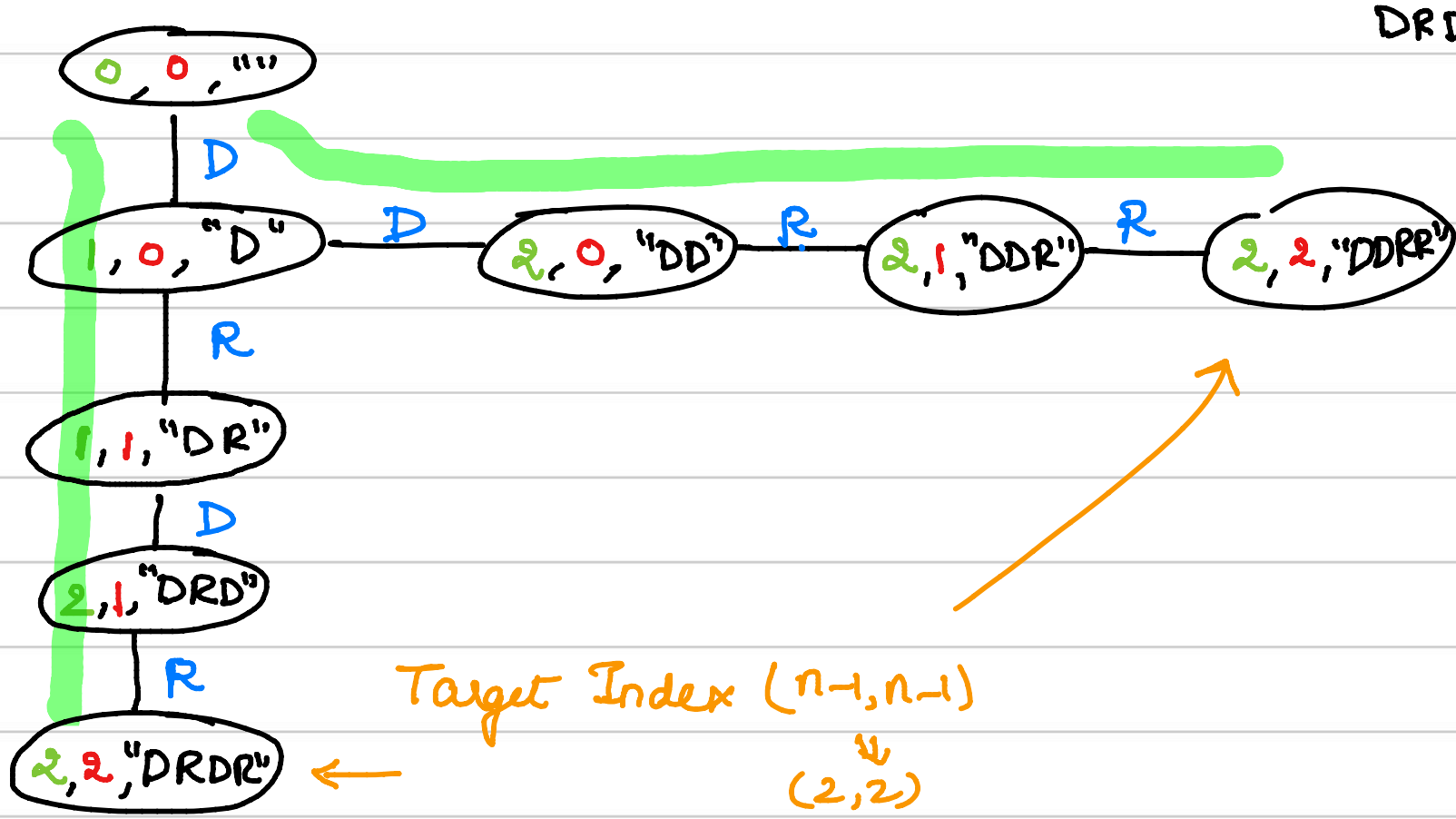
0 1 2
0 [1, 0, 0]
1 [1, 1, 0]
2 [1, 1, 1]

⇒ DRDR , DDRR

N = 4



[DDRDRR, DRDDRR]



* Before making any call from cell change its state

* while returning, UNDO the changes made (Backtracking)

Code →

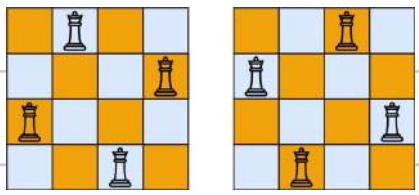
```cpp
class Solution{
    public:
    void allPaths(int row, int col, int n, vector<vector<int>>&m, string ans, vector<string>&res){

        if(row<0 || row>=n || col<0 || col>=n || m[row][col]==0){
            return;
        }

        if(row==n-1 && col==n-1){
            res.push_back(ans);
            return;
        }

        m[row][col]= 0;
        allPaths(row+1, col,n,m,ans+"D",res);
        allPaths(row, col-1,n,m,ans+"L",res);
        allPaths(row, col+1,n,m,ans+"R",res);
        allPaths(row-1, col,n,m,ans+"U",res);
        m[row][col] = 1;

        return;
    }

    vector<string> findPath(vector<vector<int>> &m, int n) {
        string ans = "";
        vector<string> res;
        allPaths(0,0,n,m,ans,res);
        sort(res.begin(), res.end());
        return res;
    }
};
```

D4 N-queens

↳ return all configurations

⑦ If given n, then we should place n-queens in N×N matrix, such that No 2 queens → share same row, column, diagonal

Eg n = 4



Initially

	0	1	2	3
0	.	.	.	.
1	.	.	.	.
2	.	.	.	.
3	.	.	.	.

X_R → Bad Row
X_C → Bad column
X_D → Bad diagonal
X_N → Not possible

Step → Start from (0,0)

① Column pos

	0	1	2	3
0	Q	.	.	.
1	.	.	.	.
2	.	.	.	.
3	.	.	.	.

Column pos: 0 ✓ | X_R X_D ✓ X_D

② Column pos

	0	1	2	3
0	Q	.	.	.
1	.	.	Q	.
2	.	.	.	.
3	.	.	.	.

Column pos: 0 ✓ | 1 X_R | 2 X_D | 3 ✓ X_D | 0 X_R | 1 X_D | 2 ✓ | 3 X_D | 0 X_R | 1 X_D | 2 X_C | 3 X_D

↳ this says (1,2) is a bad config & so is (0,0) ∴ Backtrack.

③ Column pos

	0	1	2	3
0	.	Q	.	.
1	.	.	.	.
2	.	.	.	.
3	.	.	.	.

Column pos: 0 X_N | 1 ✓ | 0 X_D | 1 X_C | 2 X_D | 3 ✓

④ Column pos

	0	1	2	3
0	.	Q	.	.
1	.	.	.	Q
2	.	.	.	.
3	.	.	.	.

Column pos: 0 X_N | 1 ✓ | 0 X_D | 1 X_C | 2 X_D | 3 ✓ | 0 ✓ | 1 X_C | 2 X_D | 3 X_C

⑤ Column pos

	0	1	2	3
0	.	Q	.	.
1	.	.	.	Q
2	Q	.	.	.
3	.	.	.	.

Column pos: 0 X_N | 1 ✓ | 0 X_D | 1 X_C | 2 X_D | 3 ✓ | 0 ✓ | 1 X_C | 2 X_D | 3 X_C | 0 X_C | 1 X_C | 2 ✓ | 3 X_C

⑥ Column pos

	0	1	2	3
0	.	Q	.	.
1	.	.	.	Q
2	Q	.	.	.
3	.	.	Q	.

Column pos: 0 X_N | 1 ✓ | 0 X_D | 1 X_C | 2 X_D | 3 ✓ | 0 ✓ | 1 X_C | 2 X_D | 3 X_C | 0 X_C | 1 X_C | 2 ✓ | 3 X_C

↑ final result ∴ store & backtrack for other config.

code →

```cpp
class Solution {
public:

    bool valid_row(int curr_row, vector<vector<char>>&grid, int n){
        for(int i = 0; i < n; i++){
            if(grid[curr_row][i]=='Q')
                return false;
        }
        return true;
    }

    bool valid_col(int curr_col, vector<vector<char>>&grid, int n){
        for(int i = 0; i < n; i++){
            if(grid[i][curr_col]=='Q')
                return false;
        }
        return true;
    }

    bool valid_diagonal(vector<vector<char>>&grid, int curr_row, int curr_col, int n){
        int i = curr_row;
        int j = curr_col;
        while(i>=0 && j>=0){        // Top-left diagonal
            if(grid[i][j]=='Q')
                return false;
            i--; j--;
        }

        i = curr_row;
        j = curr_col;
        while(i>=0 && j<n){         // Top-right diagonal
            if(grid[i][j]=='Q')
                return false;
            i--; j++;
        }

        i = curr_row;
        j = curr_col;
        while(i<n && j>=0){         // Bottom-left diagonal
            if(grid[i][j]=='Q')
                return false;
            i++; j--;
        }

        i = curr_row;
        j = curr_col;
        while(i<n && j<n){         // Bottom-right diagonal
            if(grid[i][j]=='Q')
                return false;
            i++; j++;
        }

        return true;
    }

```

```cpp
bool isValid(vector<vector<char>>&grid, int curr_row, int curr_col, int n){
    return valid_row(curr_row, grid, n) && valid_col(curr_col, grid, n) && valid_diagonal(grid, curr_row, curr_col, n);
}

// Function to convert grid char to strings
vector<string> populate(vector<vector<char>>&grid, int n){
    vector<string> result;
    for(int i = 0; i<n; i++){
        string temp = "";
        for(int j=0; j<n; j++){
            temp += grid[i][j];
        }
        result.push_back(temp);
    }
    return result;
}

void solve(vector<vector<char>>&grid, int curr_row, int n, vector<vector<string>>&ans){
    if(curr_row==n){
        vector<string> temp = populate(grid,n);
        ans.push_back(temp);
        return;
    }
    for(int curr_col=0; curr_col < n; curr_col++){
        if(isValid(grid, curr_row, curr_col,n)){
            grid[curr_row][curr_col] = 'Q';
            solve(grid, curr_row+1, n, ans);
            grid[curr_row][curr_col] = '.';
        }
    }
}

vector<vector<string>> solveNQueens(int n) {
    vector<vector<string>> ans;
    vector<vector<char>>grid(n, vector<char>(n,'.'));
    solve(grid, 0, n, ans);
    return ans;
}
};
```

(13) N-Queens II

↳ need to find the total number of possibilities

(*) everything is same as in N-Queens but return the no. of elements in the result.

D5 Sudoku Solver

(8) A sudoku solution must satisfy all of the following rules:

1. Each of the digits 1-9 must occur exactly once in each row.
2. Each of the digits 1-9 must occur exactly once in each column.
3. Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

Eg

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

→

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Algorithm

① Let (i, j) be an empty cell

② for i from 1 to 9:

if i is not in row, column, 3x3 sub-grid:

ⓐ grid(r, c) = i

ⓑ recursively fill remaining empty cells.

ⓒ if recursion is successful:

return True

ⓓ grid(r, c) = '.' (backtracking)

③ return false

Code

```cpp
class Solution {
public:
    bool valid_row(vector<vector<char>>&board, int currRow, int currVal){
        for(int i=0; i<9; i++){
            if(board[currRow][i]==currVal+'0'){
                return false;
            }
        }
        return true;
    }

    bool valid_col(vector<vector<char>>&board, int currCol, int currVal){
        for(int i=0; i<9; i++){
            if(board[i][currCol]==currVal+'0'){
                return false;
            }
        }
        return true;
    }

    bool valid_grid(vector<vector<char>>&board, int currRow, int currCol, int currVal){
        int x = 3*(currRow/3);
        int y = 3*(currCol/3);
        for(int i=0; i<3; i++){
            for(int j=0; j<3; j++){
                if(board[x+i][y+j]== currVal+'0'){
                    return false;
                }
            }
        }
        return true;
    }

    bool isValidCell(vector<vector<char>>&board, int currRow, int currCol, int currVal){
        return valid_row(board, currRow, currVal) && valid_col(board, currCol, currVal) &&
        valid_grid(board, currRow, currCol, currVal);
    }

```

```cpp

    bool sudokuSolver(vector<vector<char>>&board, int currRow, int currCol){
        if(currRow==9)
            return true;

        int nextRow = 0;
        int nextCol = 0;

        // find next possible row n column
        if(currCol==8){
            nextRow = currRow+1;
            nextCol = 0;
        } else {
            nextRow = currRow;
            nextCol = currCol+1;
        }

        // if not filled then call
        if(board[currRow][currCol]!='.'){
            return sudokuSolver(board, nextRow, nextCol);
        }

        // try all possibilities from 1 to 9 numbers
        for(int currVal=1; currVal<10; currVal++){

            // if valid then make the change
            if(isValidCell(board, currRow, currCol, currVal)){
                board[currRow][currCol] = '0'+currVal;

                // if already solved then return true directly
                if(sudokuSolver(board, nextRow, nextCol)==true)
                    return true;

                // backtracking
                board[currRow][currCol] = '.';
            }
        }

        return false;
    }
    void solveSudoku(vector<vector<char>>& board) {
        sudokuSolver(board, 0, 0);
    }
};
```

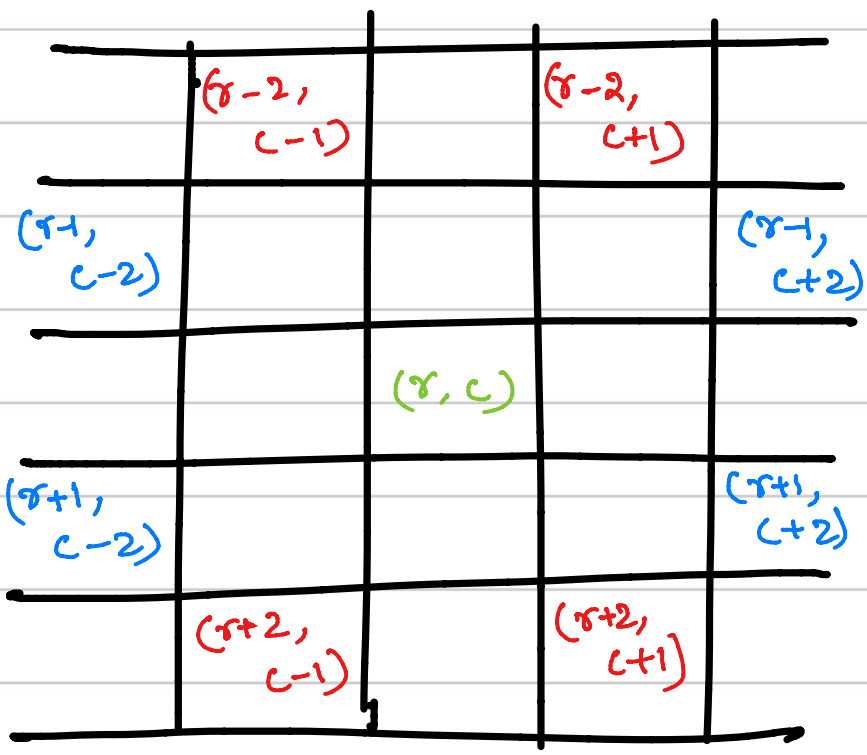
D6 Knight's tour problem.

(9) Given an n*n board, print the order of each cell in which they are visited. (n>=8)

For n = 8, the result is

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Sol) For every cell (r, c) we have 8 possibilities,



- (r-2, c-1)
- (r-2, c+1)
- (r+2, c-1)
- (r+2, c+1)
- (r-1, c-2)
- (r-1, c+2)
- (r+1, c-2)
- (r+1, c+2)

- The rest is similar to rat-in-a-maze problem except that the value will be incremented by 1.

Code →

```cpp
#include <bits/stdc++.h>
using namespace std;

void display(vector<vector<int>>&grid){
    for(auto i: grid){
        for(auto j:i){
            cout<<j<<" ";
        }
        cout<<"\n";
    }
}

void KnightTour(vector<vector<int>> &grid, int currRow, int currCol,
                                                int upcomingVal, int n){
    if(upcomingVal==n*n){
        display(grid);
        cout<<"\n";
        return;
    }

    if(currRow<0 || currRow>=n || currCol<0 || currCol>=n
                                || grid[currRow][currCol]!=0){
        return;
    }

    grid[currRow][currCol] = upcomingVal;

    KnightTour(grid, currRow-2, currCol-1, upcomingVal+1, n);
    KnightTour(grid, currRow-2, currCol+1, upcomingVal+1, n);
    KnightTour(grid, currRow+2, currCol-1, upcomingVal+1, n);
    KnightTour(grid, currRow+2, currCol+1, upcomingVal+1, n);
    KnightTour(grid, currRow-1, currCol-2, upcomingVal+1, n);
    KnightTour(grid, currRow-1, currCol+2, upcomingVal+1, n);
    KnightTour(grid, currRow+1, currCol-2, upcomingVal+1, n);
    KnightTour(grid, currRow+1, currCol+2, upcomingVal+1, n);

    grid[currRow][currCol] = 0;
    return;
}

int main() {
    int n;
    cin>>n;
    vector<vector<int>>grid(n,vector<int>(n,0));
    KnightTour(grid, 0, 0, 1, n);
    return 0;
}

```

(10) Letter combination of a phone number

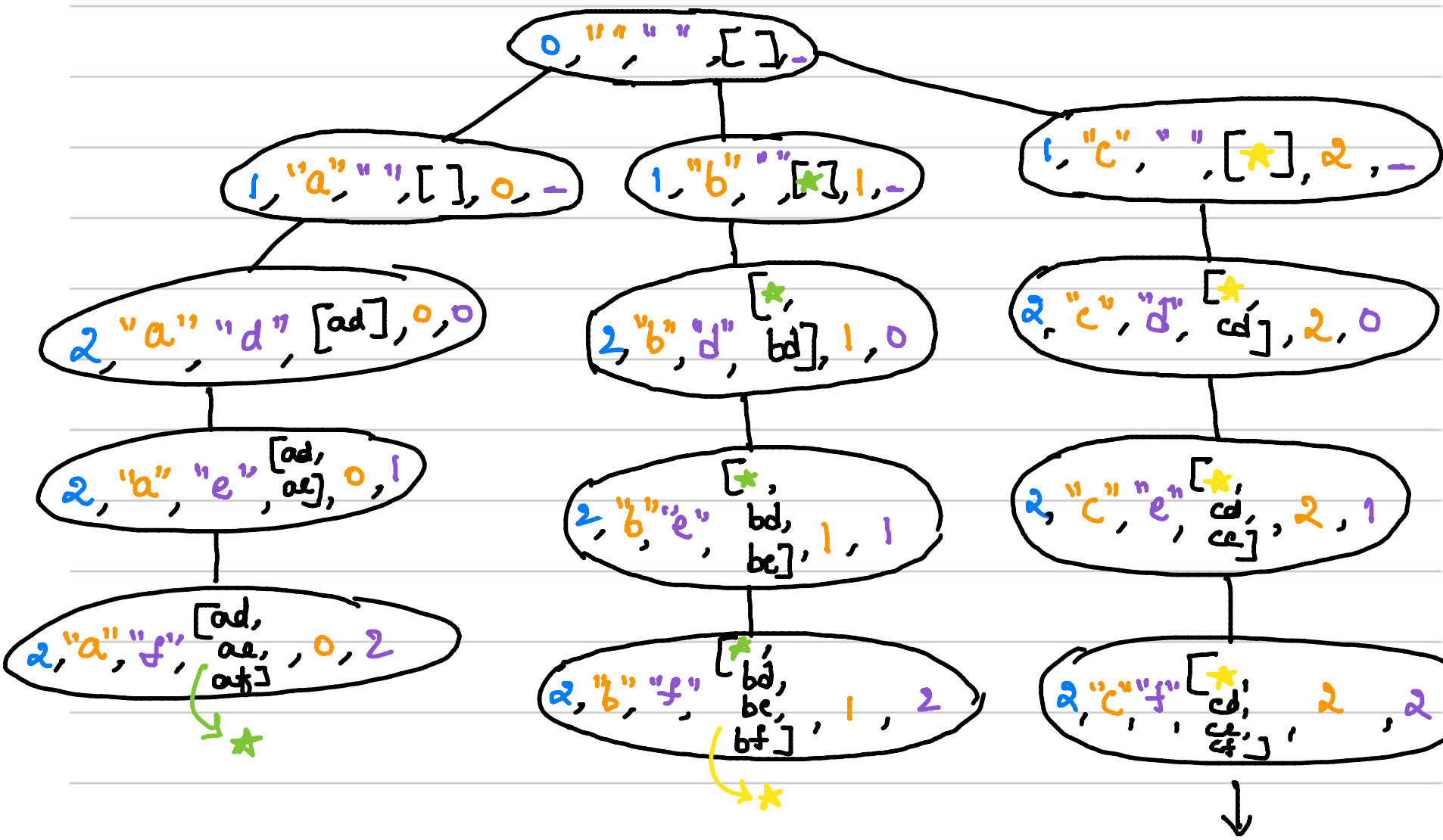
Eg digits = "23" → abc (0 1 2), def (0 1 2)

"2" → abc (0 1 2)

"3" → def (0 1 2)

* Initially create a map for numbers & their alphabets
* Then for each index in a string find all possibilities

//



final result.

[ad, ae, af, bd, be, bf, cd, ce, cf]

code →

```cpp
class Solution {
public:
    void findAll( map<char,string> &mapper, string digits,
    vector<string> &ans, string &s, int currentIndex){

        if(currentIndex>=digits.length()){
            ans.push_back(s);
            return;
        }

        char currNum = digits[currentIndex];
        string alpha = mapper[currNum];

        for(int i=0; i<alpha.size(); i++){
            s.push_back(alpha[i]);
            findAll(mapper, digits, ans, s, currentIndex+1);
            s.pop_back();
        }
        return;
    }

    vector<string> letterCombinations(string digits) {

        map<char,string>mapper{
            {'1', ""},
            {'2', "abc"},
            {'3', "def"},
            {'4', "ghi"},
            {'5', "jkl"},
            {'6', "mno"},
            {'7', "pqrs"},
            {'8', "tuv"},
            {'9', "wxyz"},
          };
        string s = "";
        vector<string> ans;

        // edge case
        if(digits.size()==0){
            return ans;
        }
        // else generate all possibilities
        findAll(mapper, digits, ans, s, 0);
        return ans;

    }
};
```

(11) Subsets II → Same as subsets but no duplicates.

(1) using set<int>

Code →

```cpp
class Solution {
public:
    void allsubs(vector<int>& nums,int curr,
    vector<int>&ds,set<vector<int>>&ans)
    {
        if(curr>=nums.size()){
            ans.insert(ds);
           return;
        }
        int currval = nums[curr];
        ds.push_back(currval);
        allsubs(nums,curr+1,ds,ans);

        // removing currentVal (not considering)
        ds.pop_back();
        allsubs(nums,curr+1,ds,ans);
    }

    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        set<vector<int>>ans;
        vector<int>vec;
        sort(nums.begin(),nums.end());
        allsubs(nums,0,vec,ans);
        vector<vector<int>> res{ans.begin(), ans.end()};
        return res;
    }
};
```

② without using sets

code →

```cpp
class Solution {
public:
    void allsubs(vector<int> &nums, int curr, vector<int> &ds,
                                        vector<vector<int>>& res){
        res.push_back(ds); // storing initial answers
        for(int i=curr; i<nums.size(); i++){
            if(i>curr && nums[i]==nums[i-1]) continue; // avoiding duplicates
            ds.push_back(nums[i]);
            allsubs(nums, i+1, ds, res);
            ds.pop_back();
        }
        return;
    }

    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        vector<vector<int>> res;
        vector<int> ds;
        sort(nums.begin(), nums.end());
        allsubs(nums, 0, ds, res);
        return res;
    }
};
```

(12) Combinational Sum - II

→ Same as combinational sum but no duplicates

Code →

```cpp
class Solution {
public:
    void findAll(vector<int>&candidates, int target, int idx,
                 vector<vector<int>> &ans, vector<int> &ds){

        if(target==0){
            ans.push_back(ds);
            return;
        }

        for(int i = idx; i<candidates.size(); i++){

            // avoid duplicates
            if(i>idx && candidates[i]==candidates[i-1]) continue;

            if(candidates[idx]<=target){
                ds.push_back(candidates[i]);
                findAll(candidates, target-candidates[i], i+1, ans, ds);
                ds.pop_back();
            }
        }
    }

    vector<vector<int>> combinationSum2(vector<int>& candidates,
                                        int target){
        vector<vector<int>> ans;
        sort(candidates.begin(), candidates.end());
        vector<int> ds;
        findAll(candidates, target, 0, ans, ds);
        return ans;
    }
};
```

(13) N-Queens II

↳ need to find the total number of possibilities

(*) everything is same as in N-Queens but return the no. of elements in the result.