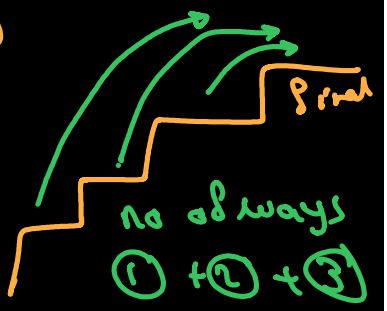
"Recursion"

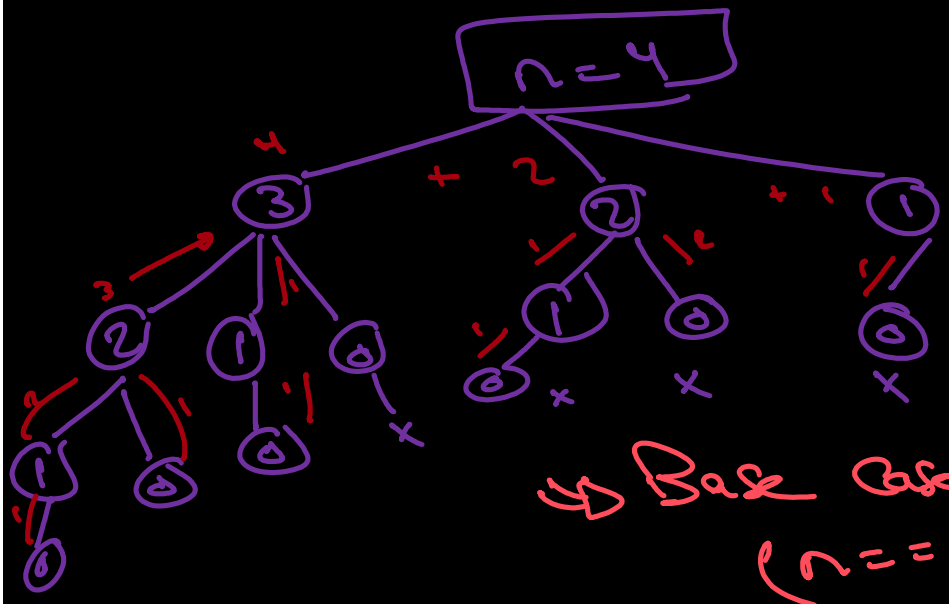
① climbing stairs:→

Suppose you have the ladder and you can climb it by one step or two or three at the same time. I want to find the number of ways to climb it?

$$F(n) = F(n-1) + F(n-2) + F(n-3)$$

⇒ if no of stairs = 7, By how many ways you can climb it?





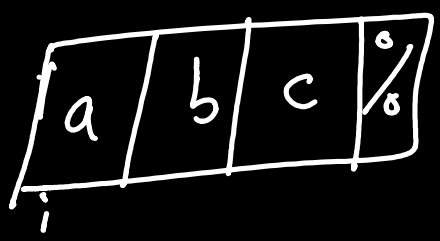
no of ways = 7

⇒ Base case will be if (n == 0) return 1;—

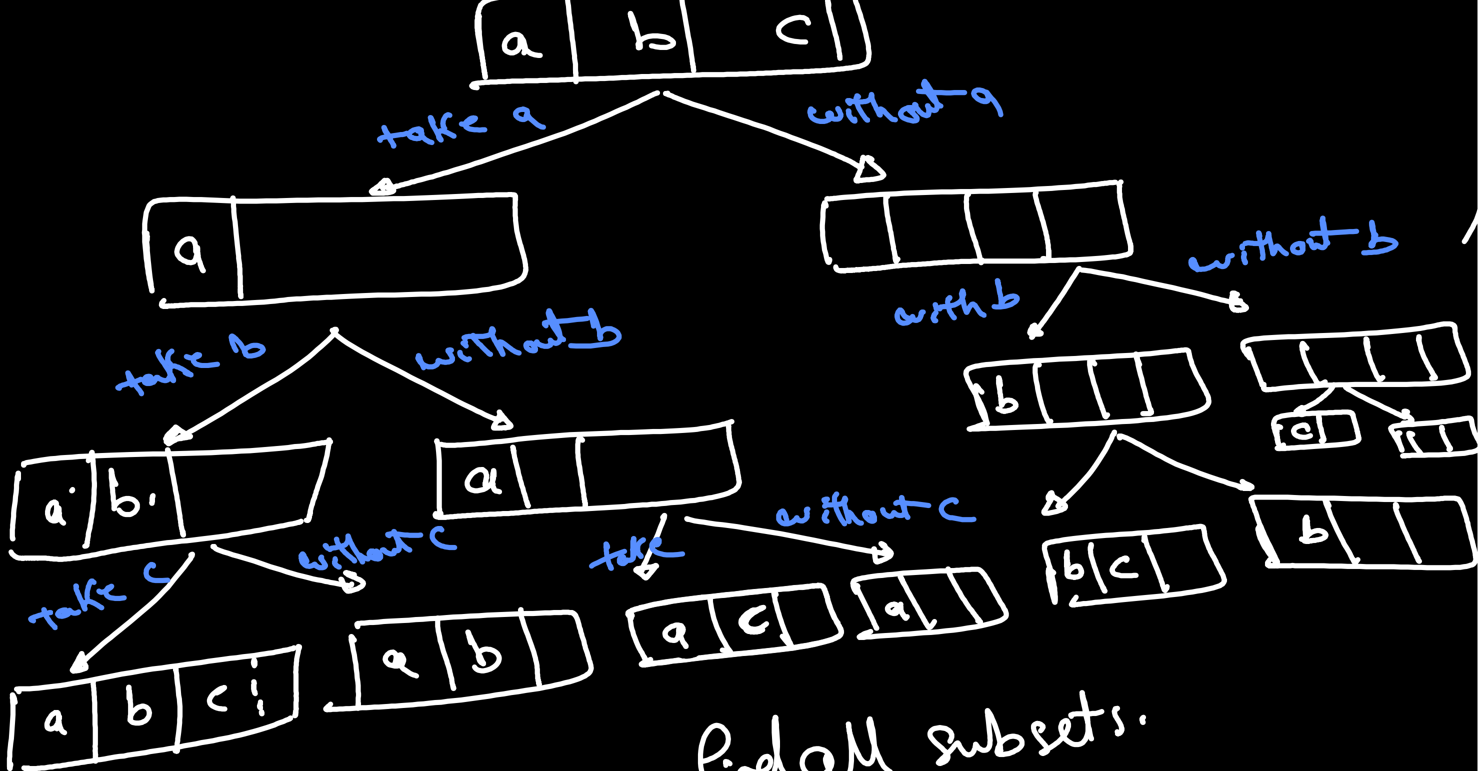
---

* Find all subsets:→

⇒ I want to find all subsets of the string "abc", How to approach that?



if I take the letter I will move it, i will move the it one ahead further.



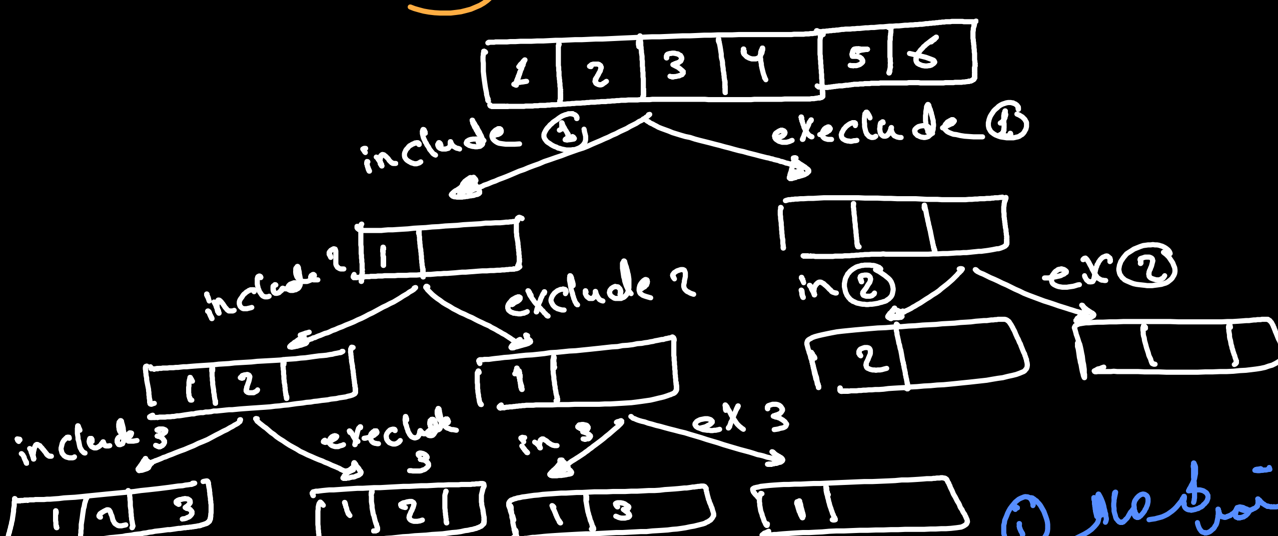
By this way we can find all subsets.

* Subsets Sum equal to x: →

Suppose I have array and target sum, I want to know How many subsets I can perform from the original one

if arr = {1, 2, 3, 4, 5} and the $x = 6$

* How many subsets which can be of sum x



وتستمر كده (i) لما توصل للـ arr.size ساعتها هتعمل check الـ sum بيساوي x ولا؟

```cpp
int getSubSetsEqualToSum(vector<int> arr, int sum,int i, int reminder) {
    // Base Case

    if(i == arr.size()) {
        if(reminder == 0)
            return 1;
        else return 0;
    }
    // Recursice Case
    // 1- Include The Current Element and Subtract its Value from the reminder
    int include = getSubSetsEqualToSum(arr, sum, i+1, reminder - arr[i]);
    // 1- Execlude The Current Element And There Is No Subtraction From the Reminder
    int execlude = getSubSetsEqualToSum(arr, sum , i+1, reminder);
    return include + execlude;
}
```

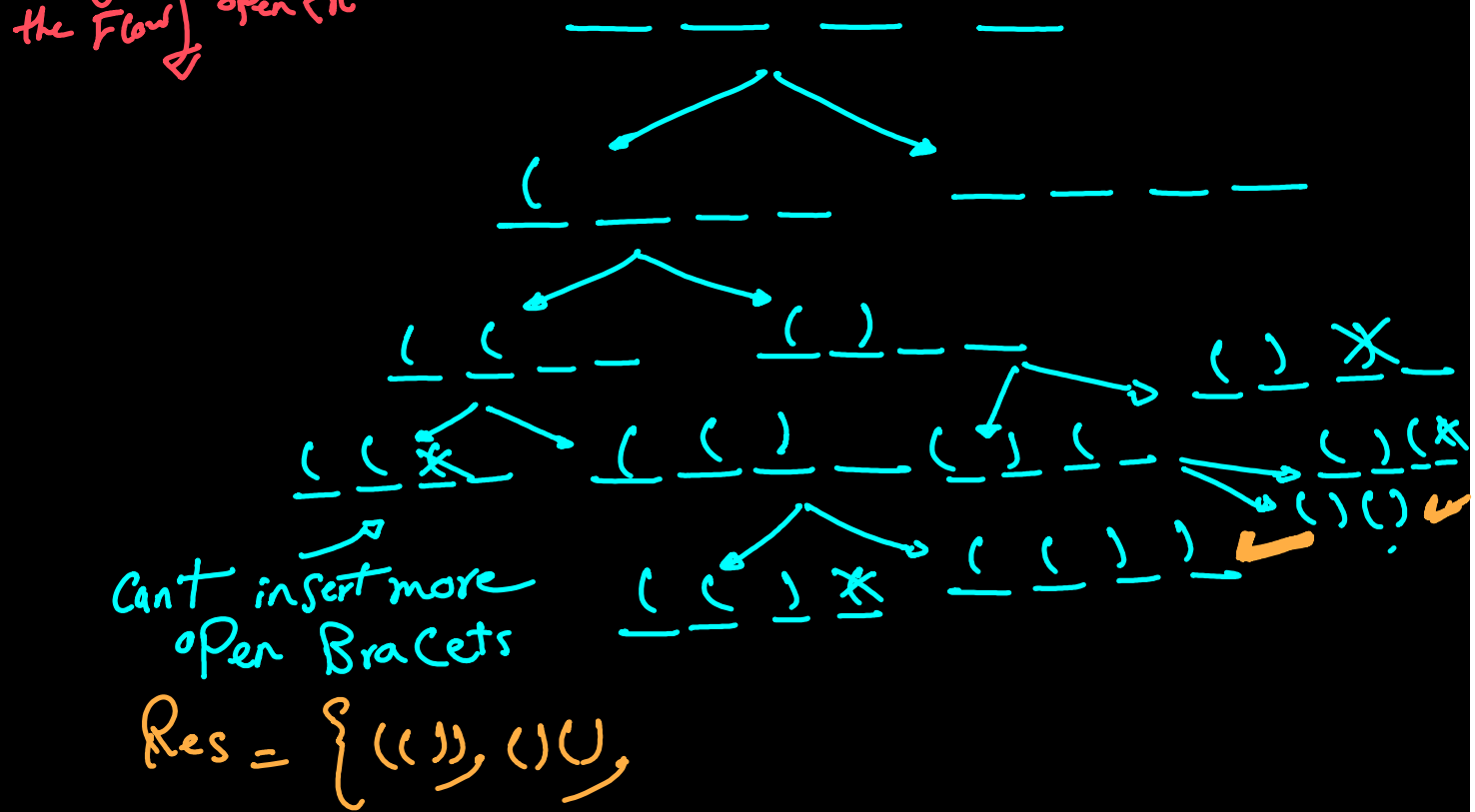
* Generate Bracets :=
↳ Generate all valid subsets of Bracets that can be formed

① Put an open one or closed and ask the recursion to do the rest of the work.

② to be valid it should → openBrackets ≤ closed ones
→ openBrackets < n

for example if n==2

during the Flow { open ≤ closed
open < n



Res = {(()), ()()}

```cpp
void getNoOfSubsets(vector<string> &finalRes, string res, int open, int closed, int n, int i)
{
    // Base Case
    if (i == n * 2)
    {
        finalRes.push_back(res);
        return;
    }

    // Recursive Case
    // 1- Forward Direction
    if (open < n)
    {
        res.push_back('(');
        getNoOfSubsets(finalRes, res, open + 1, closed, n, i + 1);
        res.pop_back();
    }
    // 2- BackWard Direction
    if (closed < open)
    {
        res.push_back(')');
        getNoOfSubsets(finalRes, res, open, closed + 1, n, i + 1);
        res.pop_back();
    }
}
```

---

* Smart key Board :→

For example if the input is 23 ?

2 → A, B, C

3 → D, E, F

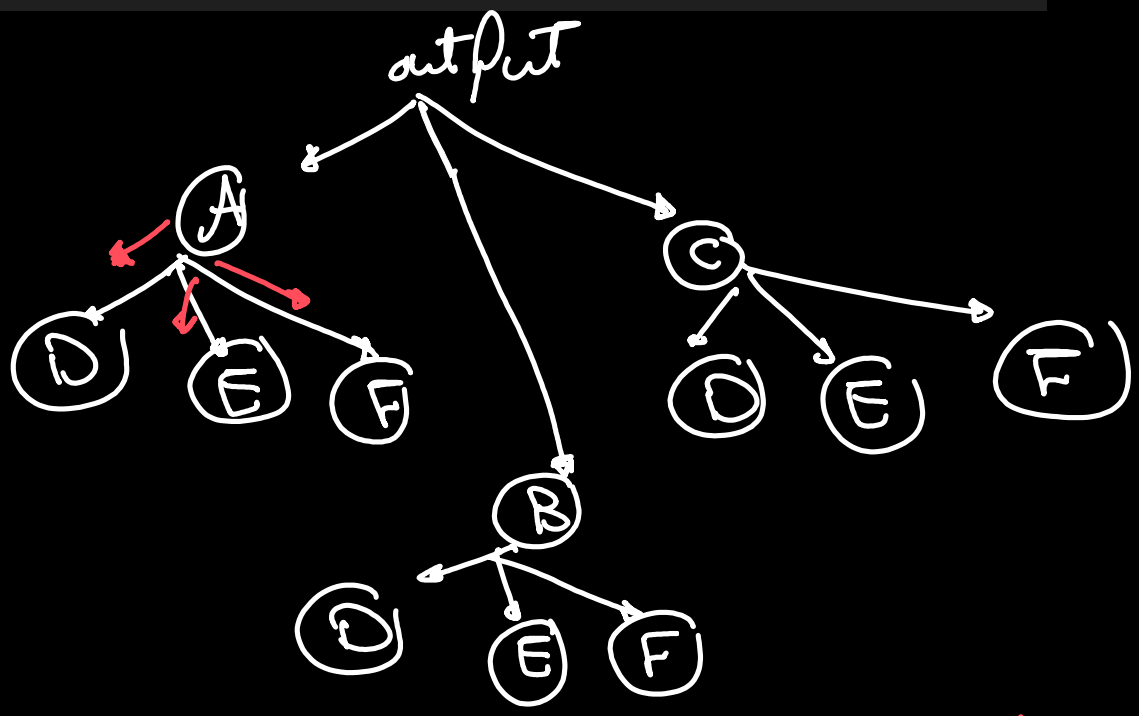
The Possible results are :→

AD	BD	CD
AE	BE	CE
AF	BF	CF

How to approach this?

→ Simply I will iterate throught the input string which is for example "23", first I will extract 2 and get associated letters which is abc then extract for 3 which is def

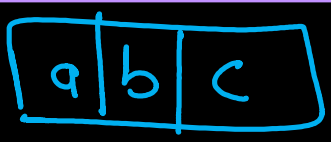
```cpp
vector<string> keybad = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

void getAllKeybad(string input, string output, int i, vector<string> &res)
{
    if (input[i] == '\0')
    {
        res.push_back(output);
        return;
    }

    int currDigit = input[i] - '0';
    if (currDigit == 0 || currDigit == 1)
        getAllKeybad(input, output, i + 1, res);

    for (int k = 0; k < keybad[currDigit].size(); k++)
    {
        getAllKeybad(input, output + keybad[currDigit][k], i + 1, res);
    }
    return;
}
```



The output will be ["AD", "AE", "AF", "BD" "BE", "BF", "CD", "CE", "CF"]

---

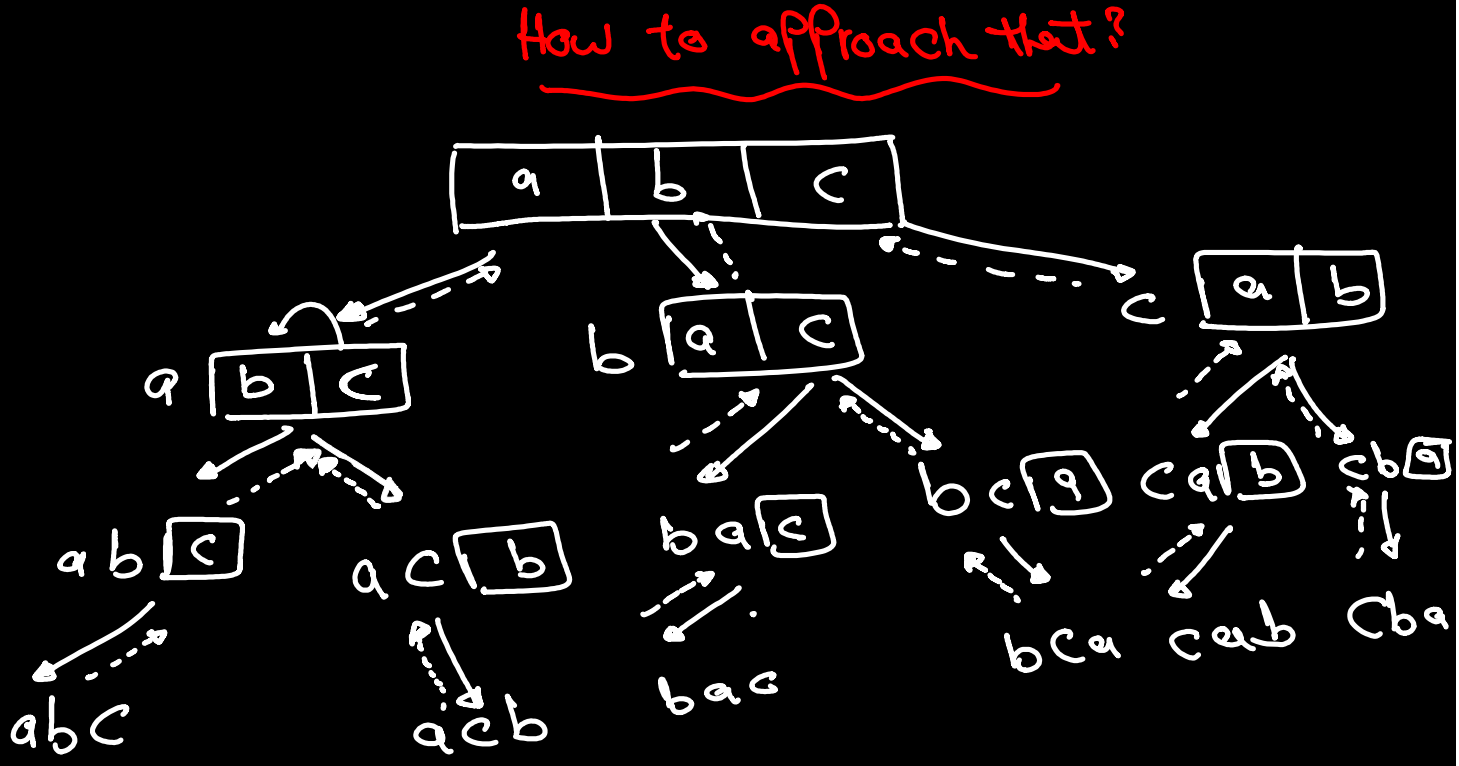
* Permutation:



* abc
* acb

* bac
* bca

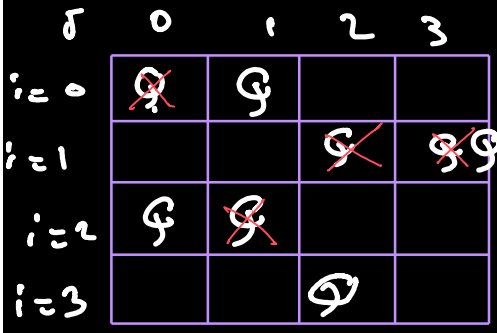
* cab
* cba




```cpp
class Solution {
public:
void swap(int &firstNum, int &secondNum) {
    int temp = firstNum;
    firstNum = secondNum;
    secondNum = temp;
}
    void getPermute(vector<vector<int>> &finalRes, vector<int> &res, int i) {
        if(i == res.size()) {
            finalRes.push_back(res);
            return;
        }

        for(int k = i; k < res.size(); k++) {
            swap(res[k], res[i]);
            getPermute(finalRes, res, i+1);
            swap(res[k], res[i]);
        }
    }
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> finalRes;
        getPermute(finalRes, nums, 0);
        return finalRes;
    }
};
```

* N Queen :→

➘ You will be given n x n Matrix and you want to find all Possible Matrices Formed By placing Queen, horizontal and vertical and diagonal not intercept.



* First I will place a Queen at j=0 and ask the recursion to Solve the Sub Problem Starting from i==1
* The Valid place is at j==2 again ask the Recursion Solve
* at i==2, we can't place the Queen at any Positi So I will return False to its Parent and ask him to modify its Queen Place
* The valid place For a Queen at i==2 will be at j==2
* Then ask the recursion to Solve the Rest subProbl but at i==4, there isn't any valid place For the Queen to be placed.
* So, I will return False to i==2, which will return false to i==1
* The first Row will move a queen one step ahead Then Ask the Recursion to Solve the rest and So on ------>

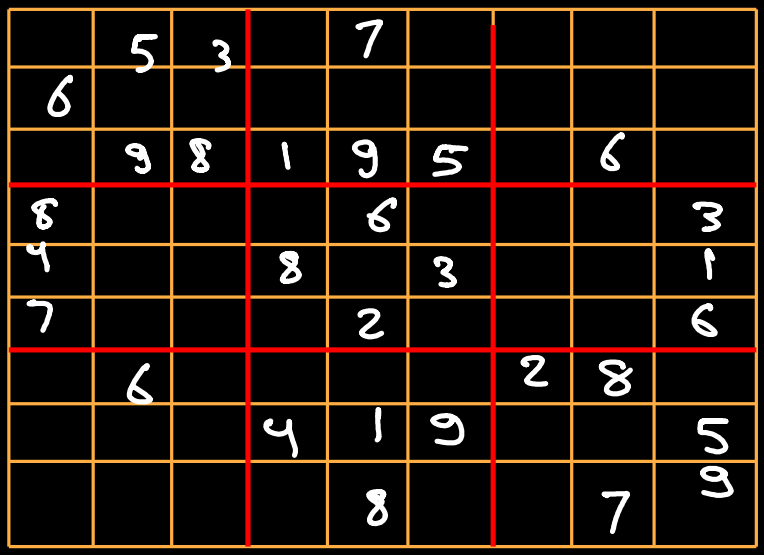
```cpp
bool solveNQueen(int n,int board[][20],int i){
    //base case
    if(i==n){
        //Print the board
        printBoard(n,board);
        return;
    }


    // rec case
    // try to place a queen in every row
    for(int j=0;j<n;j++){
        //whether the current i,j is safe or not
        if(canPlace(board,n,i,j)){
            board[i][j] = 1;
            bool success = solveNQueen(n,board,i+1);
            if(success){
                return true;
            }
            //backtrack
            board[i][j] = 0;
        }
    }
    return false;
}
```

---

* Sedoku Solver:→

Given a matrix of size 9x9 with some of Pre filled values, I want to fill the other cells in such a manner that :→

1. For every Sub Problem of size 3*3 there is no duplicate numbers.
2. the number doesn't exist in Row.
3. the number doesn't exist in Colum.



The unfilled cells is filled by Zero, and I want to change them to numbers From 1→9 fulfill the mentioned Conditions above.

```cpp
bool solveSudoku(int mat[][9],int i,int j,int n){
        //base case
        if(i==n){
                //print the solution later
                for(int i=0;i<9;i++){
                        for(int j=0;j<9;j++){
                                cout<<mat[i][j]<<" ";
                        }
                        cout<<endl;
                }

                return true;
        }

        //rec case
        if(j==n){
                return solveSudoku(mat,i+1,0,n);
        }

        //skip  the prefilled cell
        if(mat[i][j]!=0){
                return solveSudoku(mat,i,j+1,n);
        }

        //cell to be filled
        //try out all possibilites
        for(int no=1;no<=n;no++){
                //whether it is safe to place the number or not
                if(isSafe(mat,i,j,no)){

                        mat[i][j] = no;
                        bool solveSubproblem = solveSudoku(mat,i,j+1,n);
                        if(solveSubproblem==true){
                                return true;
                        }
                }

        }
        // if no option works, backtracking
        mat[i][j] = 0;
        return false;
}
```

Jump to next row

if it Contain avalue pre filled

Backtrack if the rest Can't be Solved