```
Vector<vector<int>> combo(vector<int>& cond, int target)
{   vector<vector<int>> answer;
    vector<int> ds;
    sort(cond.begin(), cond.end());
    Solve(0, condidate, target, answer, ds);
    return answer;
}
```

Subset Sum

Given a list (array) of N integers, print sums of all subsets in it. Output should be printed in increasing order of sums.

Input : N = 2, Arr = [2, 3]

Subsets = [0], [2], [3], [2, 3]

Output = 0, 2, 3, 5

Approach

formula : Number of subset = 2^n where n = no. of elements

[3, 1, 2], N = 3

Subsets => { } → 0, {3} → 3, {1} → 1, {2} = 2, {3, 1} → 4

{3, 2} → 5, {1, 2} → 3, {3, 1, 2} → 6

Output = 0, 1, 2, 3, 3, 4, 5, 6

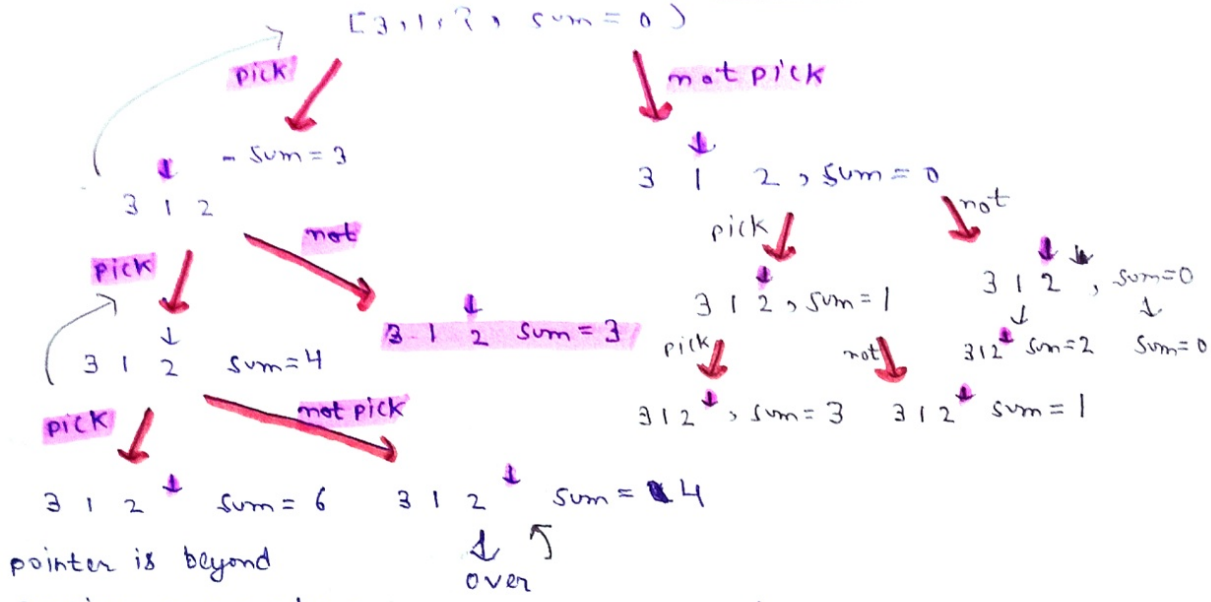
We can use Power set for subset generation using Bit Manipulation which take TC = $2^n \times N$ but this is Brute force soln

OPTIMISED || TC = $2^n + 2^N \log(2^N)$

2^N = size of output

|| SC = 2^N

arr = [3, 1, 2]



pointer is beyond arr size means we have taken our subset sum successfully

over

[3, 1, 2, sum = 3]

3 1 2, sum = 5

3 1 2, sum = 3

f(ind, s = 0, ans)

pick

not

f(ind+1, s = 0+arr[ind], ans)

f(ind+1, s = 0, ans)

```
// Base
if (ind == arr.size())
    {
        ds.push(subset);
    }
```

Subset - II

Given an integer array nums that may contain duplicates, return all possible subsets [the power set].

Solution must not contain duplicate subsets. Return the solution in any order.

Input: nums = [1,2,2]

Output: [[],[1],[1,2],[1,2,2],[2],[2,2]]

Input => nums = [0]

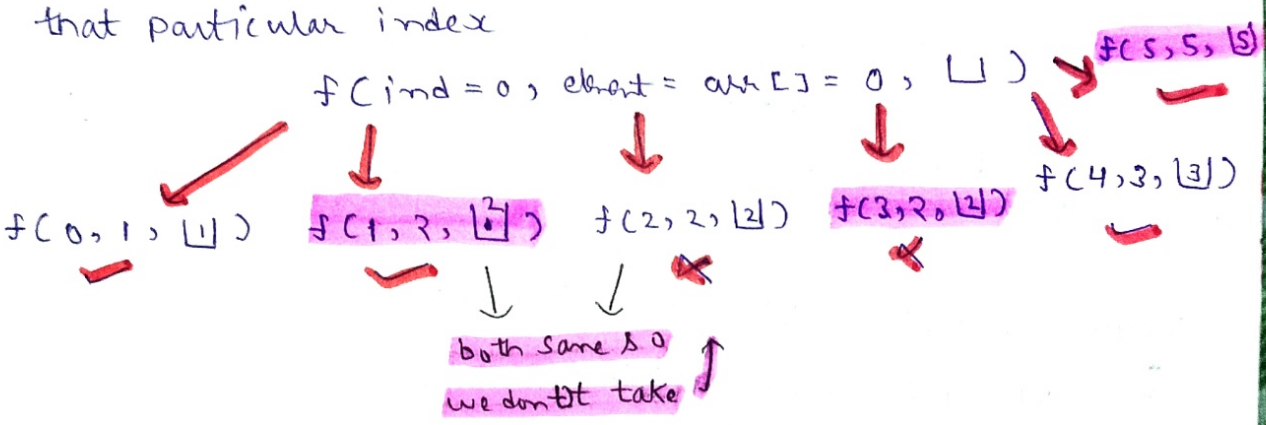
Output = [[],[0]]

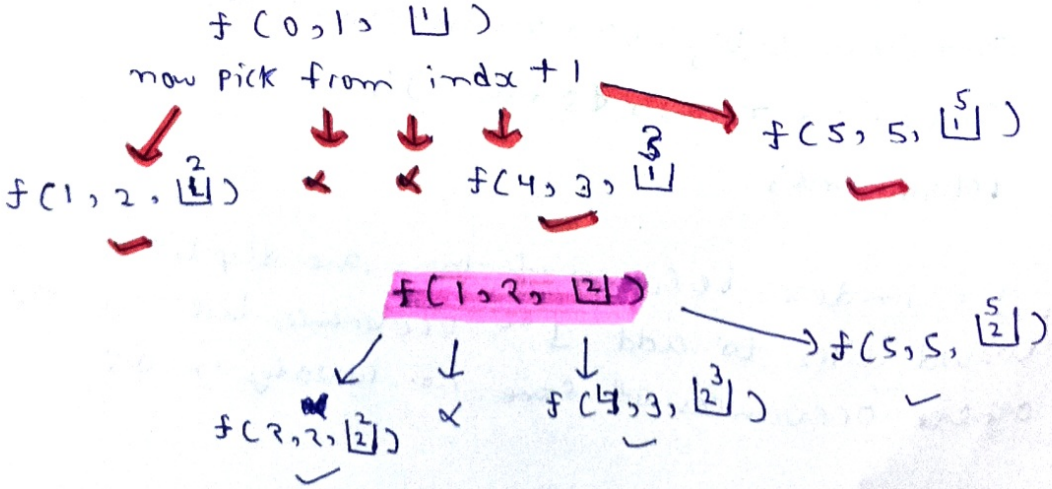
Approach

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

we can start from any index starting from a index = 0 and an empty DS just same as combination sum - 2

but if arr[ind] == arr[ind-1] we will not take that particular index





Code // TC = $2^N \times n$ // SC = $O(2^n) \times k$

↓ generating subset ↓ copy each subset to ans

k = length of subset

aux space = O(n)

```
// we do not have base condn because this loop below will
run till n-1 and it automatically gets over once array size
exceeded by index
// we use backtrack step of pop-back
void Solve(index, nums, ds, &ans)
{
    ans.push_back(ds);
    for(int i=index; i<nums.size(); i++)
    { if(i!=index && num[i]==nums[i-1]) continue;
      ds.push_back(nums[i]);
      Solve(i+1, nums, ds, ans);
      ds.pop_back();
    }
}
vector<vector<int>> subset(nums)
{
    vector<vector<int>> ans;
    vector<int> ds;
    Sort(nums.begin(), nums.end());
    Solve(0, nums, ds, ans);
    return ans;
}
// i!=index because if there are duplicates we
would like to add 1st occurence but not the
others occurences of same no. already in ds.
```

(1st March) Combination Sum

Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where chosen number sum to target. You may return combinations in any order. The same no. may be chosen from candidates an unlimited no. of times. Two combination are unique if frequency of atleast one of chosen no. is different. It is guaranted that no. of unique combinations that sum up to target is less than 150 combination of given input.

Input: candidates = [2,3,6,7] target = 7

Output = [2,2,3] = 7 and [7].

Input: candidates = [2,3,5] target = 8

Output = [2,2,2,2], [2,3,3], [3,5]

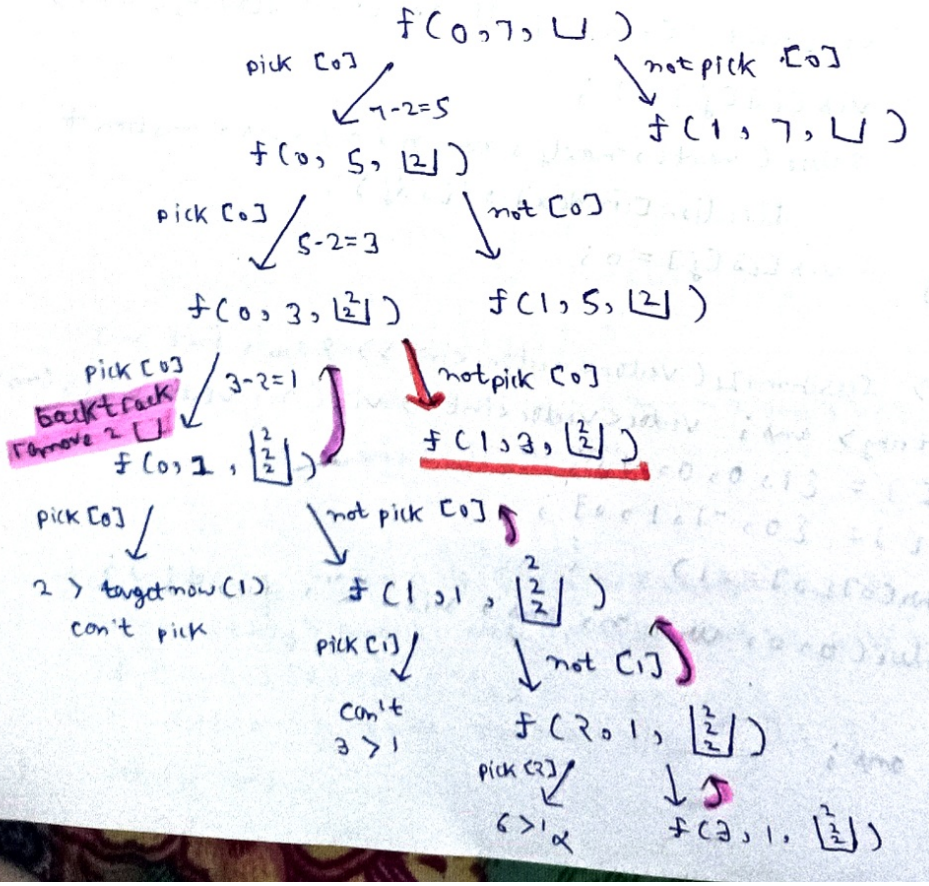
Approach ||TC = $2^{target} \times k$ || SC = k x x

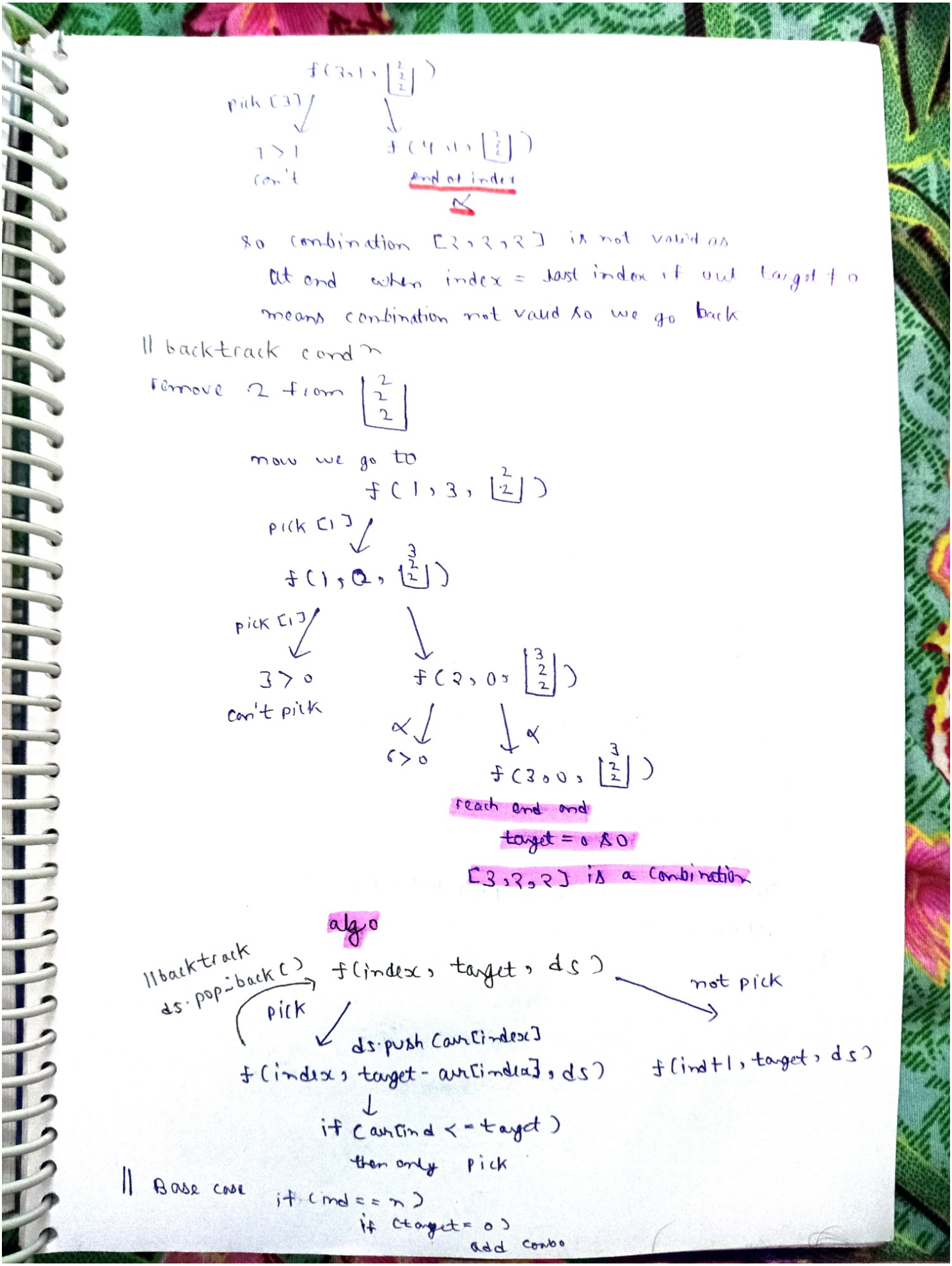
x = no. of combo

We will use pick / non pick Approach

We can pick one value multiple times so if we pick a index say (0) then again next time we still be at (0) and again we will choose whether we can pick 0 or not. we do till our arr[0] > target.

arr = [2, 3, 6, 7] . target = 7




f(3,1,[2,2,2])
pick [3]
3>1
can't
f(4,1,[2,2,2])
end of index
So combination [2,2,2] is not valid as
At end when index = last index if our target ≠ 0
means combination not valid so we go back
// backtrack cond n
remove 2 from [2,2,2]
now we go to
f(1,3,[2,2])
pick [1]
f(1,2,[3,2,2])
pick [1]
3>0
can't pick
f(2,0,[3,2,2])
<
f(3,0,[3,2,2])
reach end and
target = 0 & 0
[3,2,2] is a combination
algo
//backtrack
ds.pop_back()
f(index, target, ds)
pick
not pick
ds.push(arr[index])
f(index, target - arr[index], ds)
f(ind+1, target, ds)
if(arr[ind] <= target)
then only pick
// Base case
if(ind == n)
if(target = 0)
add combo

```
void Solve(int index, vector<int> arr, int target, vector<vector<int>> &ans, vector<int> &ds)
{
    // base case
    if(index == arr.size())
    {
        if(target == 0)
        {
            ans.push_back(ds);
        }
        return;
    }
    // recursive call
    if(arr[index] <= target)
    {
        // pick
        ds.push_back(arr[ind]);
        Solve(ind, arr, target - arr[ind], ans, ds);
        ds.pop_back();
    }

    // not pick
    Solve(ind+1, arr, target, ans, ds);
}
vector<vector<int>> combination(vector<int> &candidates, int target)
{
    vector<vector<int>> ans;
    vector<int> ds;
    Solve(0, candidates, target, ans, ds);
    return ans;
}
```

Combination Sum - II

Given candidates and target, find all unique combinations in candidates where candidate numbers sum to target. Each no. in candidates may only be used once in combination. The solution must not contain duplicates

Input ⇒ candidates = [10, 1, 2, 7, 6, 1, 5] target = 8

Output = [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]] , 1 is repeating because it appear 2 times in input

Note: Return answer in lexographical order. (sorted)

Approach

// TC = $2^n \times k \times \log(\text{set size})$

Brute force

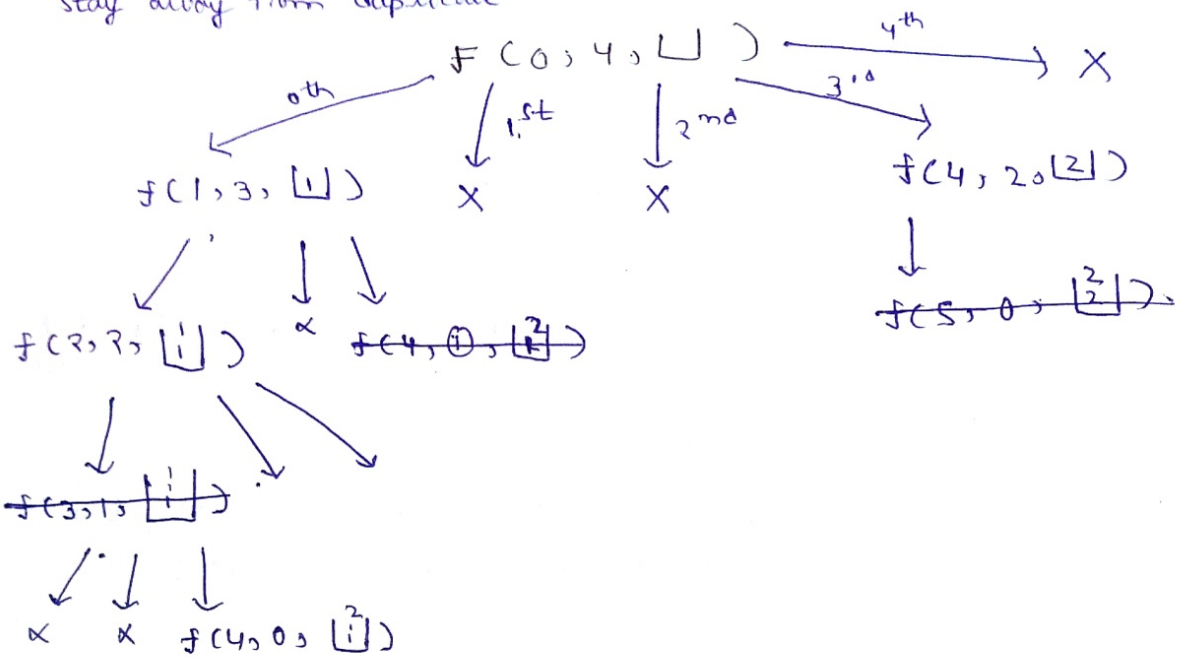
```
// same as combination 1
void solve(int index, vector<int> arr, int target,
           Set<vector<int>> &ans, vector<int> &ds)
{
    // same

    if (arr[ind] <= target)
    {
        // same
        solve(ind+1, arr, target- arr[ind], ans, ds);
        // same
    }

    Solve(ind+1, arr, target, ans, ds)
}
vector<vector<int>> comb(candidates, target)
{
    Set<vector<int>> ans;
    vector<vector<int>> answer;
    sort(candidates.begin(), candidates.end());
    for(auto it = ans.begin(); it != ans.end(); it++)
    {
        answer.push_back(*it);
    }
    return answer;
}
```

Approach 2

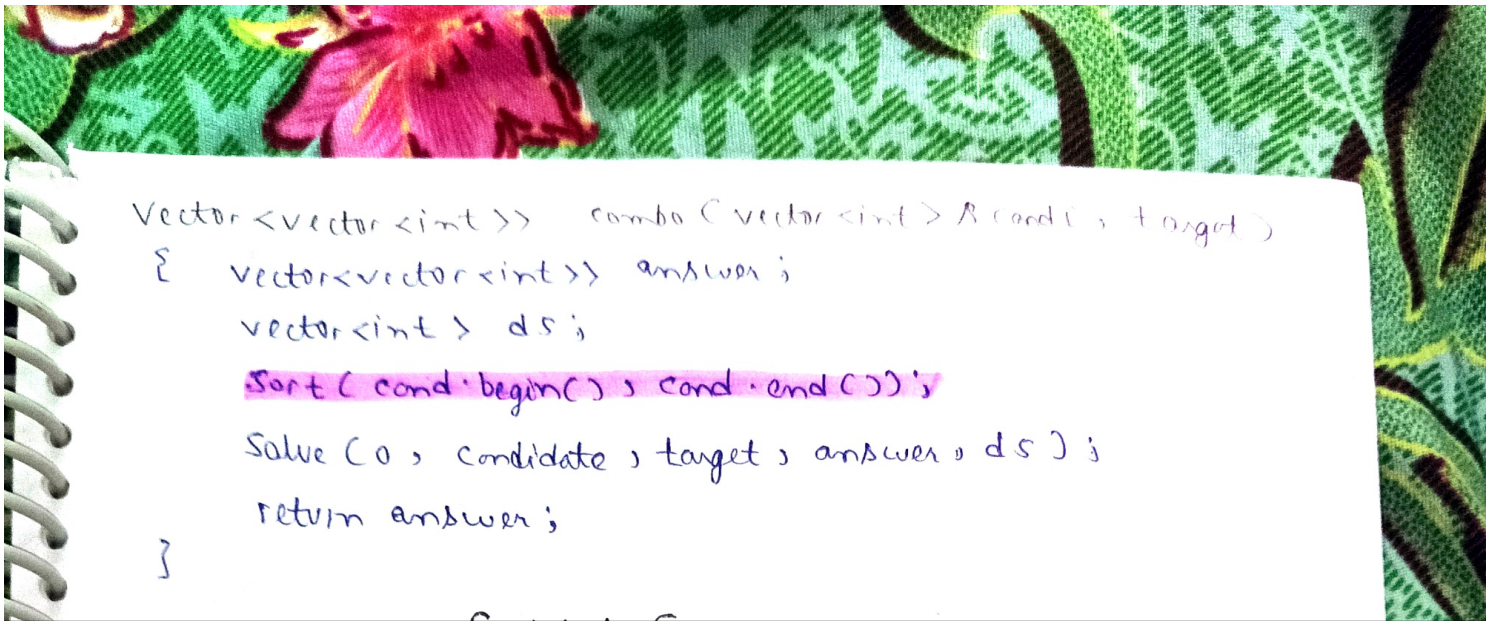
$TC = 2^n \times k$ | $SC = k \times x$

arr = [1, 1, 1, 2, 2] target = 4

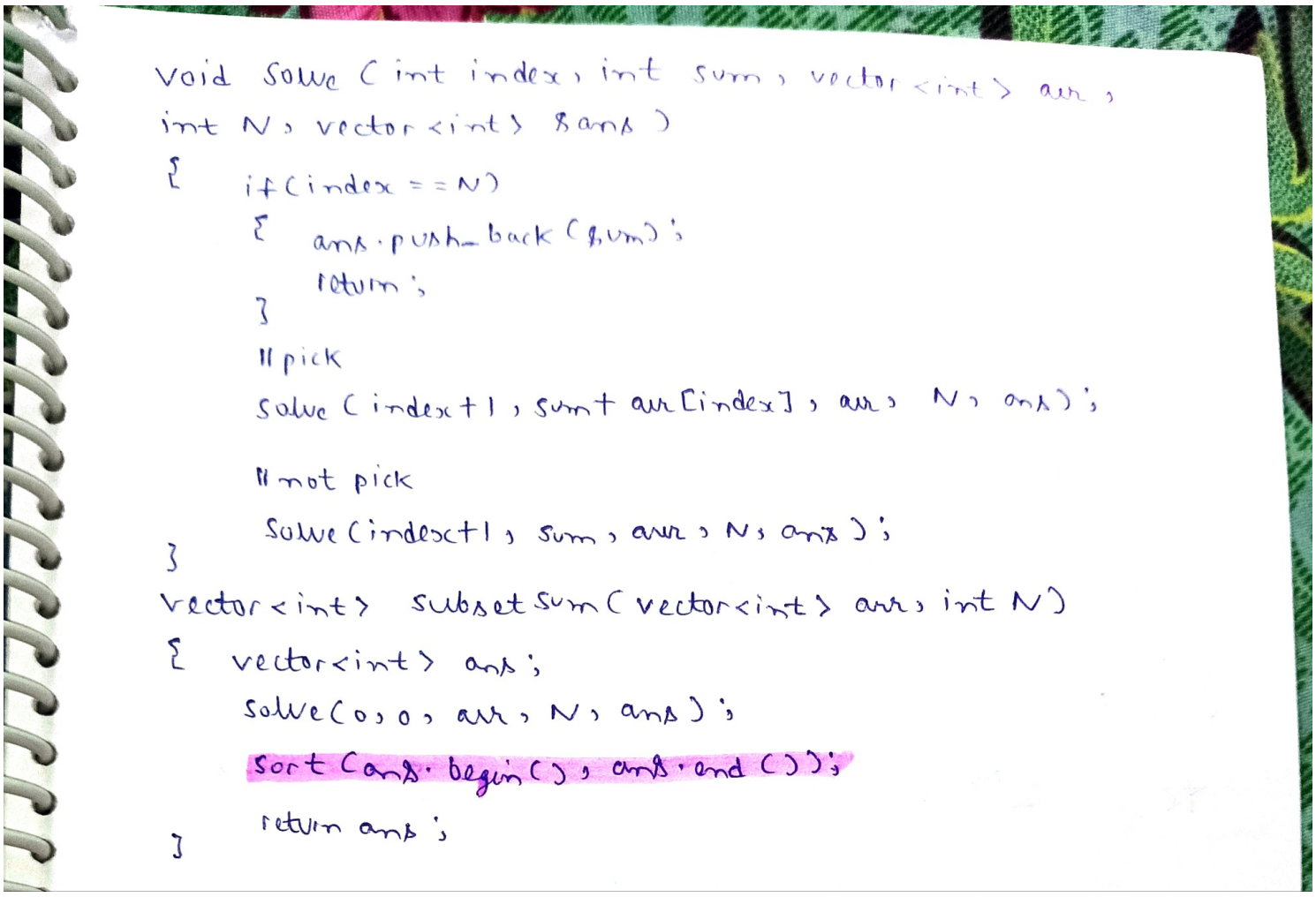
We can start with any of index but we want to stay away from duplicate




```
Void Solve(int index, vector<int> arr, int target, vector<
vector<int>> &ans, vector<int> &ds)
{   if(target==0)
    {   ans.push_back(ds);
        return;
    }
    for(int i = index; i < arr.size(); i++)
    {   if(i > index && arr[i] == arr[i-1]) continue;
        if(target < arr[i]) break;
        ds.push(arr[i]);
        Solve(i+1, arr, target - arr[i], ans, ds);
        ds.pop();
    }
}
```



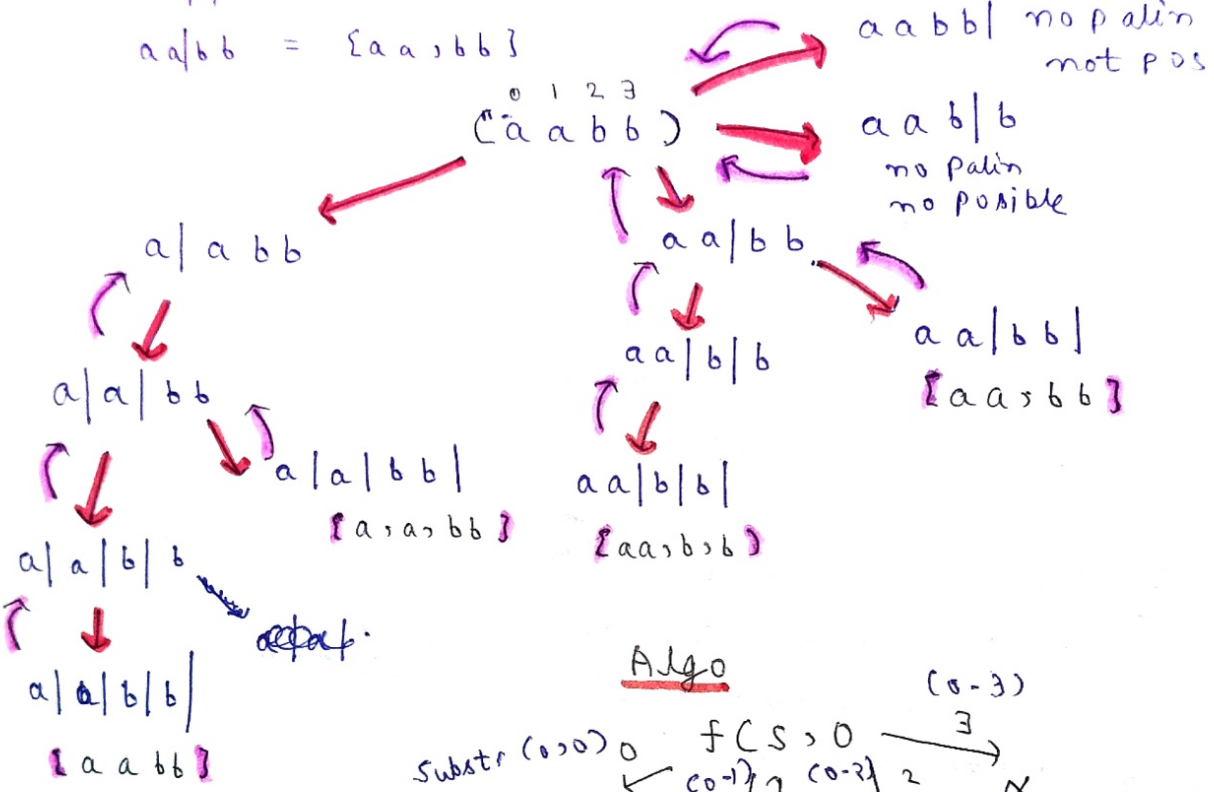
```
vector<vector<int>> combo(vector<int>& cand, int target)
{
    vector<vector<int>> answer;
    vector<int> ds;
    sort(cond.begin(), cond.end());
    Solve(0, candidate, target, answer, ds);
    return answer;
}
```



```
void Solve (int index, int sum, vector<int> arr,
int N, vector<int> &ans)
{
    if(index == N)
    {
        ans.push_back(sum);
        return;
    }
    // pick
    Solve(index+1, sum+arr[index], arr, N, ans);

    // not pick
    Solve(index+1, sum, arr, N, ans);
}
vector<int> subsetSum(vector<int> arr, int N)
{
    vector<int> ans;
    Solve(0, 0, arr, N, ans);
    sort(ans.begin(), ans.end());
    return ans;
}
```

Approach

"a|a|b|b" = { a , a , b , b }
a|a|bb = { a , a , bb }
aa|b|b = { aa , b , b }
aa|bb = { aa , bb }



Algo

(check if substring is palindrome or not)

```
for (ind → n-1)
{   is palind
}
call recursion

// base
if (ind == n)
{   store in ds
}
```

```
vector<vector<string>> partition(string s)
{ vector<vector<string>> res;
   vector<string> ans;
   solve(0, s, res, ans);
   return res;
}
void solve(index, string s, vector<vector<string>> &res,
  vector<string> ans)
{  if(index == s.size())
    {  res.push_back(ans);
       return;
    }
   for(i = index; i<s.size(); i++)
   {  if(isPalindrome(s, index, i))
      {  ans.pushback(s.substr(index, i-index+1));
         solve(i+1, s, vec, ans);
         ans.pop_back();
      }
   }
}
bool isPalindrome(string s, int start, int end)
{
    while(start <= end)
    {  if(s[start++] != s[end--])
       {
            return false;
       }
    }
    return true;
}
```

Palindrome Partitioning

Given a string s, partition s such that every substring of partition is a palindrome. Return all possible palindrome partitioning of s. A palindrome string is a string that reads same backward as forward.

Input: s = "aab"

output: [["a","a","b"],["aa","b"]]

Input: s = "a"

output = [["a"]]

Partition Sequence

The set [1, 2, 3, ..., n] consists total of n! unique permutations. By listing and labelling all permutations in order, we get the following sequence for n = 3

① "123" ④ 231
② 132 ⑤ 312
③ 213 ⑥ 321

Given n and k, return k^{th} permutation sequence

Input : n = 3, k = 3 output = "213"

Input : n = 4, k = 9 output = "2314"

Input : n = 3, k = 1 output = "123"

Approach

Brute force || TC = n! × n

① Generate all permutation using Recursion and store in a DS
② if the DS in which we've stored the permutations has 0 based indexing then our k^{th} perm = ds[k-1] after sorting DS

Optimal || TC = o(n) × o(n) to erase || SC = o(n)

n = 4, then no. of perm = n! = 24

k = 17

we can start the permo with

1 + (2, 3, 4)] 6 (0-5 permutations)
2 + (1, 3, 4)] 6 (6-11)
3 + (1, 2, 4)] 6 (12-17)
4 + (1, 2, 3)] 6 (18-23)

24 unique permutation

we need k = 17 means 17^{th} permutation and it will start with 3 as 3 has (12-17 perm) as there is 0 based indexing so

k = 17 means 16^{th}

each no. contain 6 permotation so

16 / 6 = 2

means index = 2 has 16^{th} perm starting for 0 based indexing

16 / 6 = 4th sequence from those 6 permutations given by starting with 3. So now question comes down to [1, 2, 4], k = 4 as we already know sequence start from 3.

we can say our permutation will start from

1 + (2, 4)] 2 permutations (0 - 1)

2 + (1, 4)] 2 (2 - 3)

4 + (1, 2)] 2 (4 - 5)

6 i.e 3! as n = 3 now

Since k = 4, 4 / 2 = 2 means index = 2 i.e 4

4 % 2 = 0 means perm start from 4 at 0th. question become [1, 2], k = 0

perm can start with

1 + (2)] 1 (0 - 0)

2 + (1)] 1 (1 - 1)

2 i.e 2!

0 / 1 = 0 i.e 0th index means perm start with 1

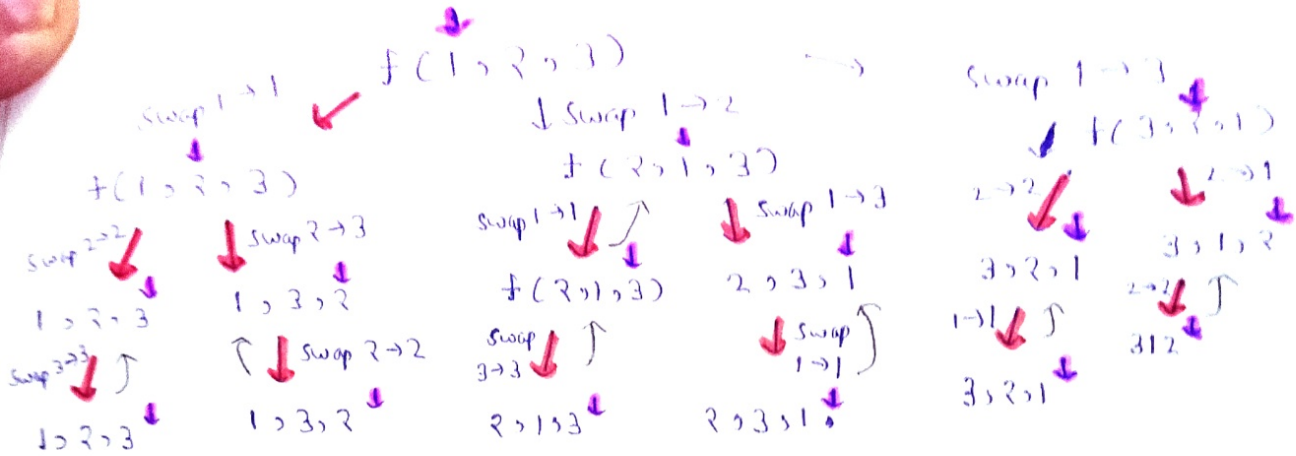
0 % 1 = 0 i.e k = 0

[2], k = 0

2 + (empty)] 1

so our Output 3 4 1 2

```
string getPermutation(int n, int k)
{   int fact = 1;
    vector<int> numbers;
    for(i=1; i<n; i++)
    {   fact = fact * i;
        numbers.push(i);
    }
    numbers.push(n);
    k = k-1;
    while(true);
    {
        ans += to_string(numbers[k/fact]);
        numbers.erase(numbers.begin() + (k/fact));
        if(numbers.size()==0)
        {   break;
        }
        k = k % fact;
        fact = fact / numbers.size();
    }
    return ans;
}
```



Permutations

Given an array nums of distinct integers, return all possible permutations. You can return answer in any order.

Input : nums = [1,2,3]

Output = [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]

Input : nums = [0,1]

Output = [[0,1], [1,0]]

Input : nums = [1]

Output = [[1]]

CODE

// TC - O(n!) X n
// SC - O(n!) X n

```
void solve( int index, vector<int> &nums, vector<vector
<int>> &ans )
{   if (index == nums.size()-1)
    {   ans.push_back (nums);
        return;
    }
    for (int j= index; j <nums.size(); j++)
    {   swap (nums[index], nums[j]);
        Solve (index+1, nums, ans);
        swap (nums[index], nums[j]);
    }
}
Vector<vector<int>> permute (vector<int> &nums)
{   vector<vector<int>> ans;
    Solve (0, nums, ans);
    return ans;
}
```

Power of three

if there exist an x such that n == 3x

```
bool isPowerofThree(int n)
{   // TC = O(log3(n))      SC = O(log3(n))
    if(n==1) return true;
    if(n<=0 || n%3 != 0) return false;
     return isPowerofThree(n/3);
}
```

Power of Four

```
bool isPowerofFour(int n)
{   // Recursively   TC = O(log4(n)) ; SC = O(log4(n))
    if(n==1) return true;
    if(n<=0 || n%4 != 0) return false;
     return isPowerofFour(n/4);
}

bool isPowerofFour(int n)
{   // Iterative  TC = O(log4(n))  ;  SC = O(1)
    if(n==0) return false;
    while(n%4==0)
    {   n=n/4;
    }
    return n==1;
}
```

Power of Two using Recursion

Given an integer no Return true if its power of two . otherwise return false
An integer is power of 2 if there exist an integer x such that $n = 2^x$

CODE

Divide no. by 2 until it reach 1 successfully if not div by 2 or -ve return false.

// TC = O(log2(n)) SC: O(log2(n))

~~if (n == 1) return ;~~

```
bool isPowerofTwo (int n)
{   if (n==1)  return true;
    if (n<=0 || n%2 != 0) return false;
    return isPowerofTwo(n/2);
}
```

51. N-Queens

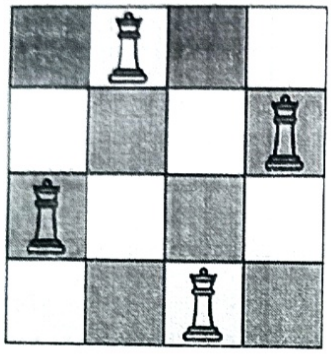
Hard 5264 145 Add to List Share

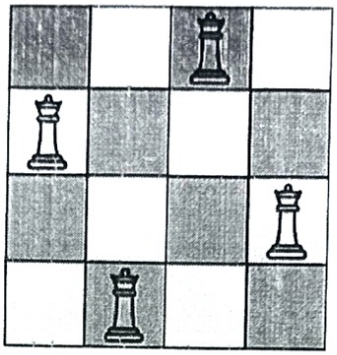
The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the* ***n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Example 1:



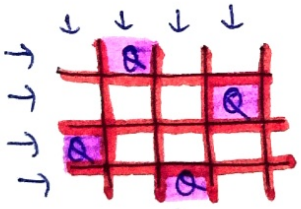


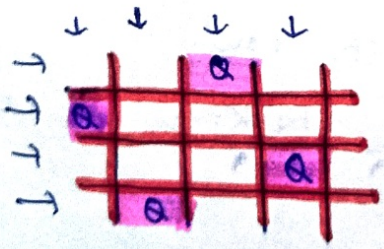
```
Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above
```

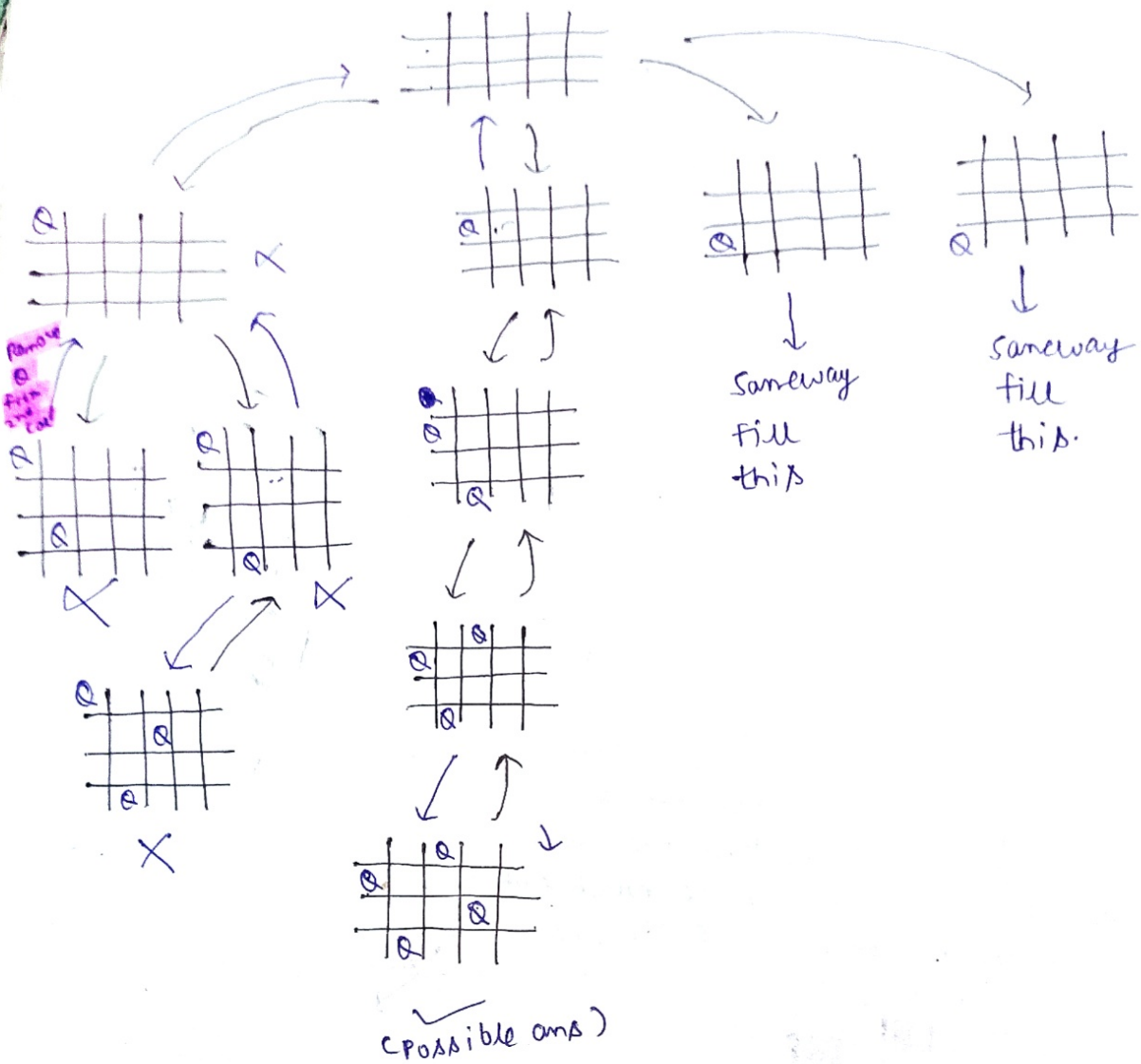
N Queens

1. every row should have one Queen
2. every col should have one Queen
3. None of Queen should attack each other

Queen can attack in 8 Directions

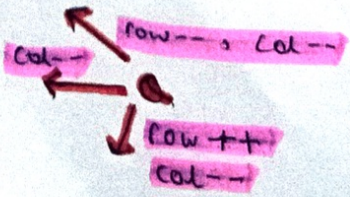






Algo

1. We are filling Queens in each row
2. Going inside each column and checking is it safe to place Queen there.
3. if we reach col end we return
4. while returning we make that box again empty.
5. while filling to check is it safe or not just check in 3 Direc^n



```
bool isSafe ( int row, int col , vec<string> board , int n)
{   // check in all 3 direc^n
    int duprow = row;
    int dupcol = col;
    // upward in left
    while (row >=0 && col >=0)
    {   if (board [row] [col] == 'Q' ) return false;
        row--; col--;
    }
    // straight on left
    row = duprow; col = dupcol;
    while (col >=0) {if(board [row] [col] =='Q') return false; col--;
    // Down ward on left
    row = Duprow; col = Dupcol;
    while (row < n && col >=0)
    { if (board [row] [col] == 'Q') return false; row++; col--; }
    return true;
}
void ways ( int col, vec<string> &ans, vec<vec<string>> &board , int n)
{   // base case
    if (col == n)
    { ans.push (board); return; }
    for (int row= 0; row <n ; row++)
    {   if (isSafe ( row, col, board , n) ;
        { board [row] [col] = 'Q' ;
          ways (col+1, ans, board, n);
          board [row] [col] = '.' ; // backtrack step
        }
    }
}
vector<vector <string>>  NQueen (int n) ;
{   vector<string> board (n);
    vector < vector <string> > ans (n);
    string s(n, '.'); // empty string
    for (i=0 ; i<n; i++)
    {   board [i] = s;
    }
    ways (0, ans, board , n);
    return ans;
}
```

37. Sudoku Solver

Hard 4682 145 Add to List Share

Write a program to solve a Sudoku puzzle by filling the empty cells.

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.

The `'.'` character indicates empty cells.

Example 1:

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

Example 1:

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

```
Input: board = [["5","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".","6","."],["8",".",".",".","6",".",".",".","3"],
["7",".",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".",".","8",".",".","7","9"]]
Output: [["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"],
["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"]]
Explanation: The input board is shown above and the only valid solution is shown below:
```

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

SUDOKU SOLVER

Rules of Sudoku

1. It is 9×9 board where there are 3×3 board which are 9 in number.
2. The digit 1-9 appears only once in each row
3. The digit 1-9 appears only once in each col
4. Digits 1-9 appears only once in any 3×3 board

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

1. We have to fill empty places in given sudoku keeping above cond^n in mind.
2. We will try to fill the free spot with every no. b/w 1-9 keeping in mind all above cond^n.
3. We check empty places in row-wise fashion and then try all possibility in it
4. if answer is not possible, we remove filled no. this is our backtrack step. & return false.
5. if we Do not get any empty spot left means all spot filled successfully return true
6. if we have successful ans, dont remove filled no. during backtrack because we just want to return one valid answer.

```
void solveSudoku(vector<vector<char>>& board)
{   solve(board); }

bool solve(vector<vector<char>>& board)
{   for(i=0; i<board.size(); i++)
    {   for(j=0; j<board[0].size(); j++)
        {   if(board[i][j]=='.')
            {   for(char c='1'; c<='9'; c++)
                {   if(isValid(board, i, j, c))
                    {   board[i][j]=c;
                        if(solve(board))
                        {   return true;
                        }
                        else board[i][j]='.';
                    }
                }
                return false
            }
        }
    }
}

bool isValid(vector<vector<char>>& board, int row, int col, char c)
{   for(i=0; i<9; i++)
    {   if(board[i][col]==c)
            return false;
        if(board[row][i]==c)
            return false
        // special formula to check in 3X3 matrix
        if(board[3*(row/3)+i/3][3*(col/3)+i%3]==c)
            return false
    }
    return true;
}
```

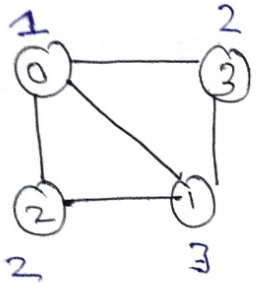
M - Coloring Problem

Given an undirected graph and an integer M, The task is to determine if graph can be colored with at most M colours such that no two adjacent vertices of graph are colored with same colour. Here colouring of graph means the assignment of colors to all vertices. Print 1 if it is possible to colour vertices and 0 otherwise.

Input: N = 4, M = 3, E = 5

Edges[] = {(1,2), (2,3), (3,4), (4,1), (1,3)};

Output = 1, it is possible to colour given graph using 3 colours

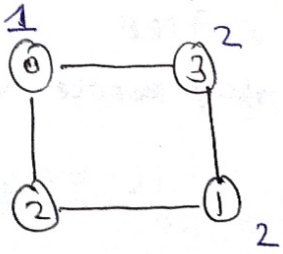


Approach

if M = 3

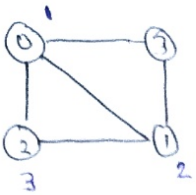
Return 1 as no 2 adjacent vertices have same colour

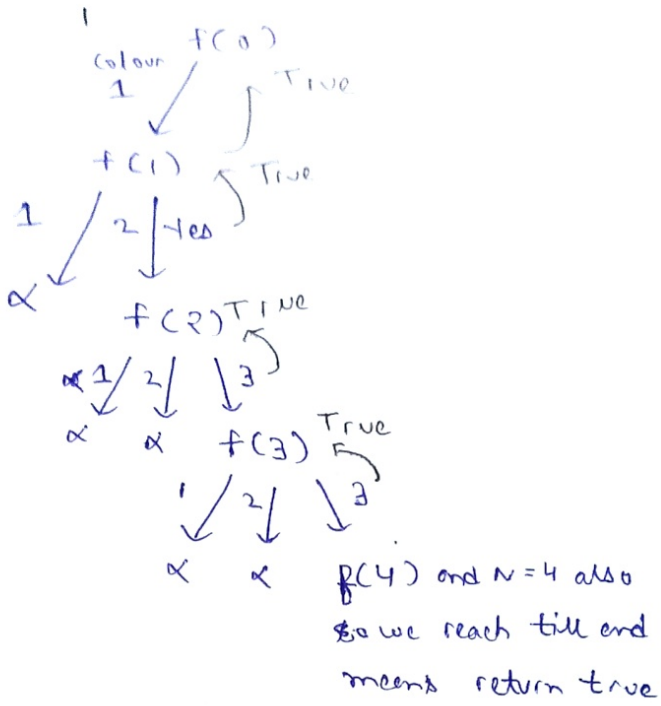
if M = 2



Return 0 as we can satisfy adjacent vertices condn using only 2 colour.

we will use recursion and will try out every colour of each vertice keeping in mind that two adjacent vertice do not have same colour.





how we can try f(1) with other colours also, f(0) with other colours also but we do not need to as we need to Return True only if its possible else return false.

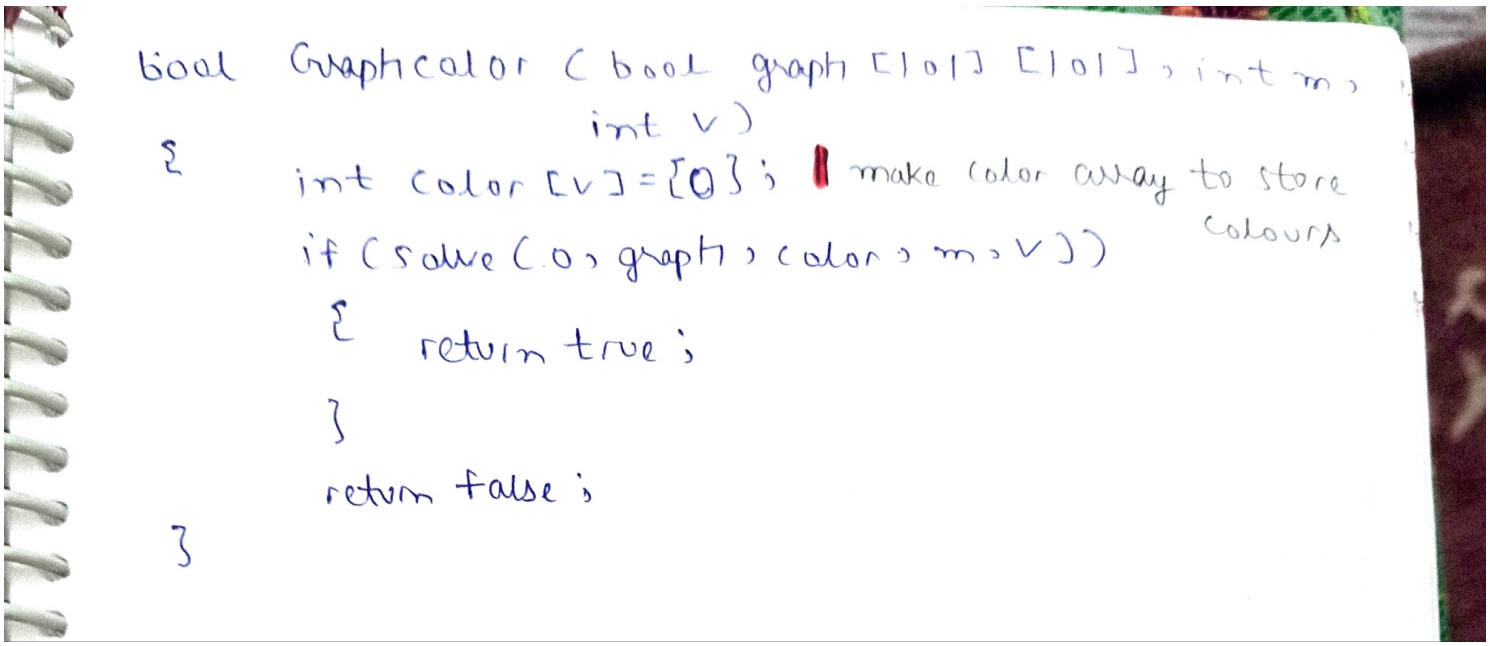
Pseudo code

T(→ N^m SC→ O(N) + O(N)

```
f(node)
{
    if (node == N) return True;
    for (col = 1 → m)
    {
        if (possible → v)
        {
            color[node] = col;
            if ( f(node + 1) == T)
                return T;
            colour[node] = 0; // backtrack if that colour does not work remove it
        }
    return False; // not possible to colour with M colors
}
```

code

```
bool isSafe(int node, int color[], bool graph[][], int m, int V, int i)
{
    for(int k=0; k<V; k++)          // to check if adjacent vertex has same colour or not
    {
        if(k != node && graph[k][node] == 1 &&
                color[k] == i)      // checks if [k][node]==1 means they are adjacent and if color[k] = i means we can't fill this colour in this node now.
        {
            return false;
        }
    }
    return true;
}

bool Solve(int node, bool graph[10][10], int color[], int m, int V)
{
    if(node == V)                   // base case
    {
        return true;
    }
    for(int i=1; i<=m; i++)         // to fill colors
    {
        if(isSafe(node, color, graph, m, V, i))
        {
            color[node] = i;        // if safe fill color
            if(Solve(node+1, graph, color, m, V))
            {
                return true;        // call for next node
            }
            color[node] = 0;        // if failed backtrack and make colour = 0
        }
    }
    return false;
}
```



```
bool GraphColor ( bool graph [ ] [ ] , int m ,
                  int V )
{
    int color [V] = {0} ;   make color array to store
                            colours
    if ( Solve ( 0 , graph , color , m , V ) )
    {
        return true ;
    }
    return false ;
}
```

Rat in a Maze 22nd Feb

Consider a rat placed at (0,0) in a square matrix of order N*N. It has to reach destination at (N-1, N-1). Find all possible paths that rat can take to reach from source to destination. The directions in which rat can move are 'U' (up), 'D' (down), 'L' (Left), 'R' (Right). Value 0 at a cell in matrix represents that it is blocked and rat cannot move to it while value 1 at cell in cell in matrix represent that rat can travel through it.

Note: In a path, no cell be visited more than one time.

Example 1:

Input:

N = 4

m[][] = {{1,0,0,0}, {1,1,0,1}, {1,1,0,0}, {0,1,1,1}};

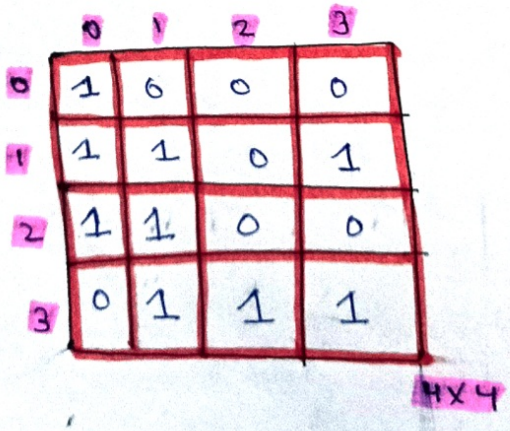
Output = DDRDRR DRDDRR

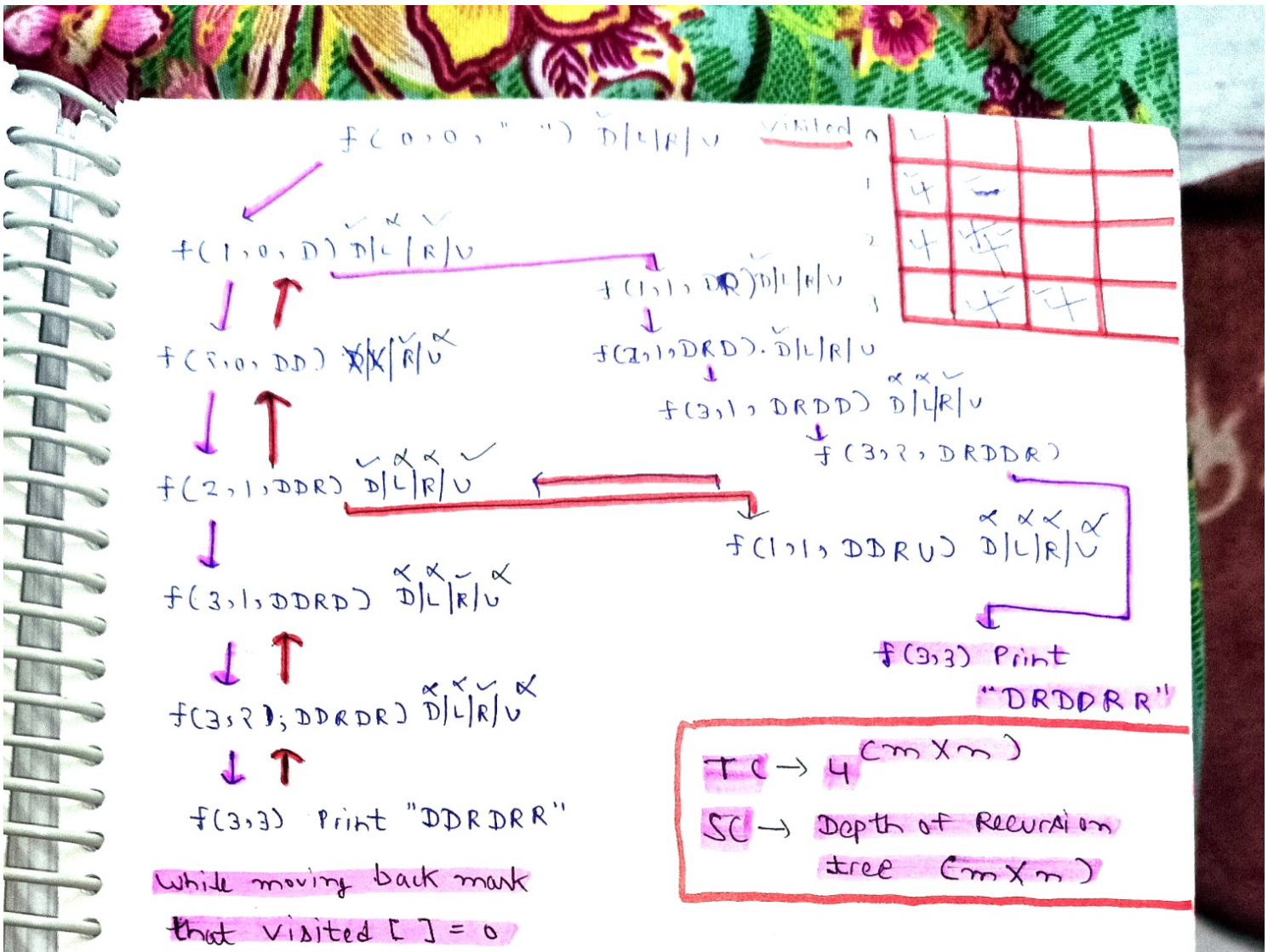
Approach

Note: Answer should follow Lexographical order

D < L < R < U

1. we take visited Array for each block which is visited
2. we travel all ways in all direc^n we can.
3. we will follow lexo order i.e D < L < R < U
4. whenever we take a path make visited[path] = 1 as we can visit one node only one time





Pseudo code

① We will write code for each Direcn D, L, R, U and Recursion will work in each one of it.

② While going
- Up → i - 1
- Down → i + 1
- Left → j - 1
- Right → j + 1

we check visited [U|D|L|R][i|j] each time if its 0 and a maze [U|D|L|R][i|j] == 1

then mark visited [ ] [ ] == 1

Recursion call

visted [ ] [ ] == 0 // backtrack step

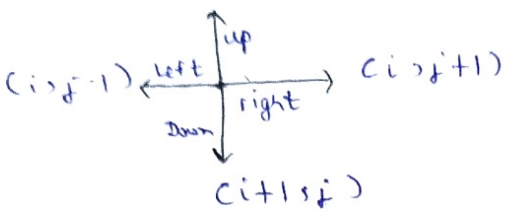
CODE

```
void solve(int i, j, vector<vector<int>> &m, int n, vector<string>
  &ans, vector<vector<int>> &vis, string motion)
{
    if(i==n-1 && j==n-1)
    {  ans.push_back(motion);
       return;
    }
    // Downward means i+1
    if(i+1<n && !vis[i+1][j] && m[i+1][j]==1)
    {   vis[i][j]=1;
        solve(i+1,j,m,n,ans,vis,motion+'D');
        vis[i][j]=0;
    }// left means j-1
    if(j-1>=0 && !vis[i][j-1] && m[i][j-1]==1)
    {   vis[i][j]=1;
        solve(i,j-1,m,n,ans,vis,motion+'L');
        vis[i][j]=0;
    }
    // Right means j+1
    if(j+1<n && !vis[i][j+1] && m[i][j+1]==1)
    {   vis[i][j]=1;
        solve(i,j+1,m,n,ans,vis,motion+'R');
        vis[i][j]=0;
    }// upward means i-1
    if(i-1>=0 && !vis[i-1][j] && m[i-1][j]==1)
    {   vis[i][j]=1
        solve(i-1,j,m,n,ans,vis,motion+'U');
        vis[i][j]=0;
    }
}
vector<string> findpath(vector<vector<int>> &m, int n)
{  vector<string> ans; vector<vector<int>> vis(n,vector<int>(n,0));
   if(m[0][0]==1)
   {  solve(0,0,m,n,ans,vis,""); }
   return ans;
}
```

Simpler Code

```
di[] = { 1, 0, 0, -1 }
dj[] = { 0, -1, 1, 0 }
```

	D	L	R	U
di[]	+1	+0	+0	-1
dj[]	0	-1	+1	+0



CODE

```
void solve( int i, int j, vector<vector<int>> arr, int n, vector<string>
&ans, vector<vector<int>> &vis, string motion, int di[], int dj[])
{   if(i==n-1 && j==n-1)
    {   ans.push_back(motion);
        return;
    }
    string direction = "DLRU";
    for(int index=0; index<4; index++)
    {   int nexti = i + di[index];
        int nextj = j + dj[index];
        if(nexti>=0 && nextj>=0 && nexti<n && nextj<n
        && !vis[nexti][nextj] && arr[nexti][nextj]==1)
        {   vis[i][j] = 1;
            solve(nexti, nextj, arr, n, ans, vis, motion +
                direction[index], di, dj);
        }   vis[i][j] = 0;
    }
}
vector<string> searchmaze(vector<vector<int>> &arr, int n)
{   vector<string> ans; vector<vector<int>> vis(n, vector<int>(n,0));
    int di[] = {1, 0, 0, -1};
    int dj[] = {0, -1, 1, 0};
    if(arr[0][0]==1)
    {   solve(0, 0, arr, n, ans, vis, "", di, dj);
    }
    return ans;
}
```