Recursion

by

Striver

Recursion :-

When a function calls itself until a specified condition is met.

Example :-

```
void f(){
  print(1);
  f();
}

main(){
  f();
}
```

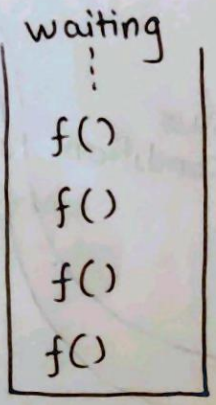
⇒ prints 1

```
void f()
{
  print(1);
  f();
}
```

```
void f(){
  print(1);
  f();
}
```

→ Infinite times

Stack



⇒ stack overflow

⇒ SEGMENTATION FAULT

Recursive function with base case :-

```
cnt=0
f(){
  if(cnt == 4) → false
    return;
  print(cnt);
  cnt++;
  f();
}

main(){
  f();
}
```

```
f(){
  if(cnt==4) → false
    return;
  print(cnt); → 1
  cnt++;
  f();
}
```

```
f(){
  if(cnt==4) ×
```

---→ like this

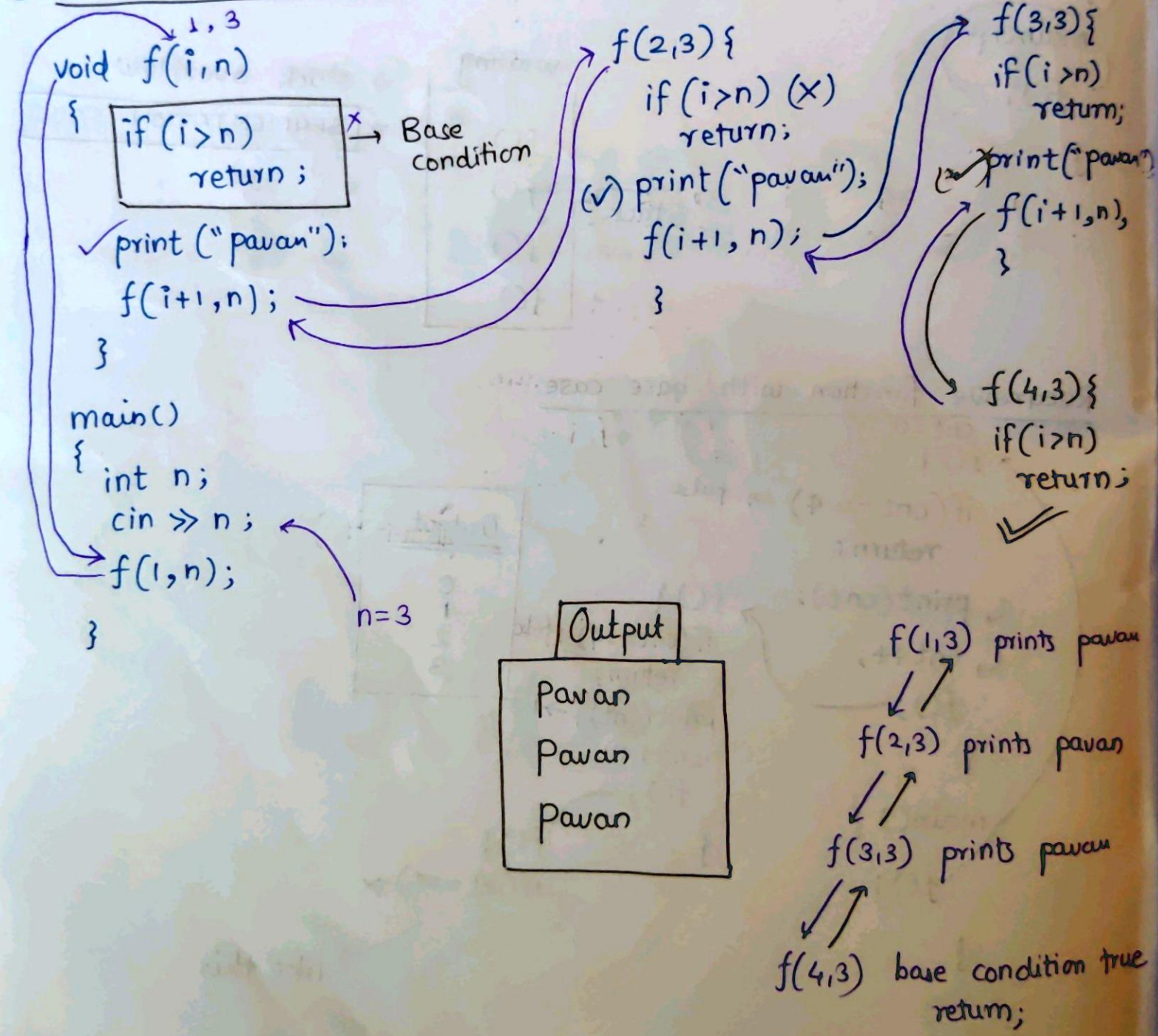
Output :-

0
1
2
3

Basic Recursion Problems :-

⇒ Print Name 5 times
⇒ Print linearly from 1 to n.
⇒ Print from n to 1.
⇒ print linearly from 1 to n (using backtracking)
⇒ print from n to 1 (using backtracking)

① Print name n times :-



Output

Pavan
Pavan
Pavan

f(1,3) prints pavan
f(2,3) prints pavan
f(3,3) prints pavan
f(4,3) base condition true
return;

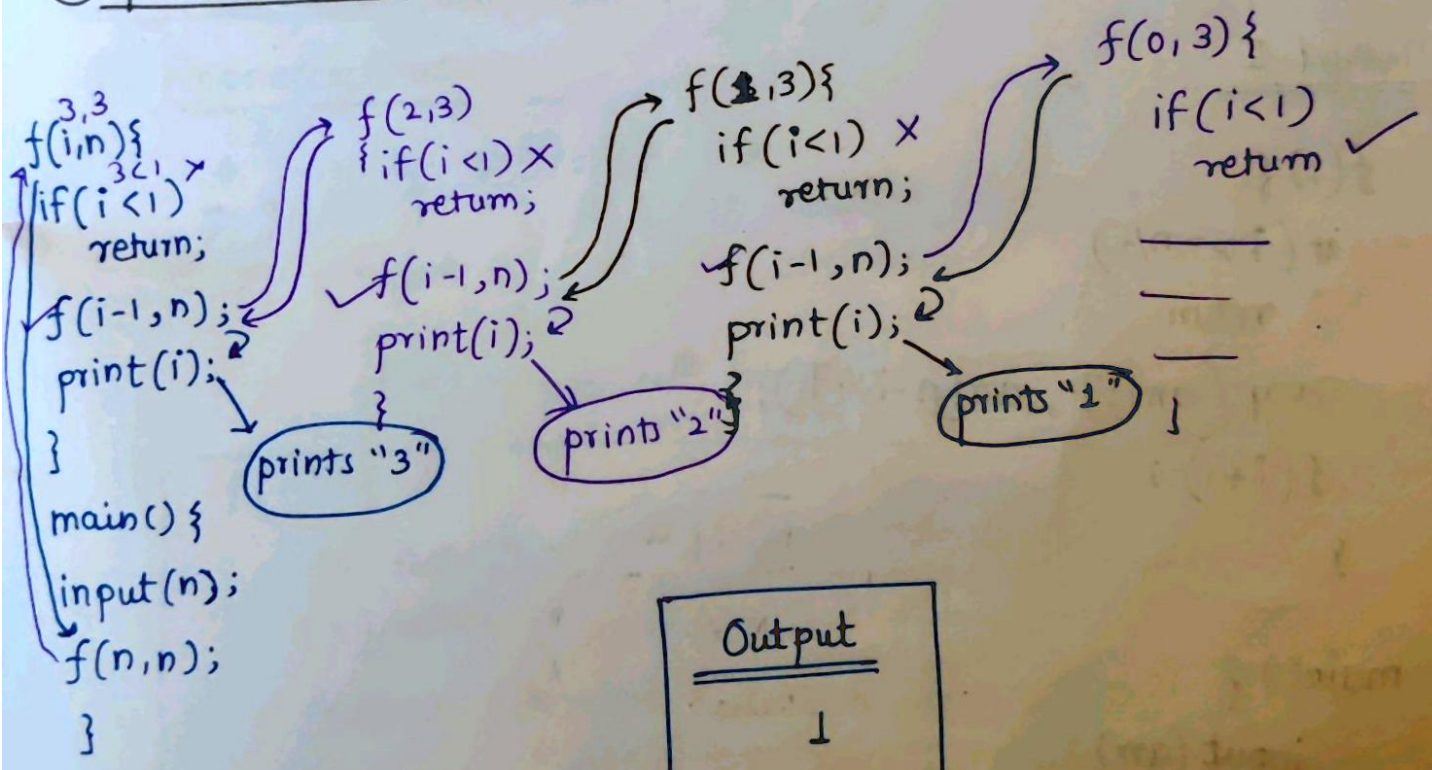
② print 1 to n :-

```
f(i,n)
{
  if (i>n)
     return;
  print(i);
  f(i+1,n);
}
main() {
  input(n);
  f(1,n);
}
```

③ print n to 1

```
f(i,n)
{  if(i<1)
     return;
   print(i);
   f(i-1,n);
}
main() {
  input(n);
  f(n,n);
}
```

④ print 1 to n using backtracking :-



Output
1
2
3

Reverse the array using recursion

① Method - I

```
f(l,r) {
    if(l>=r) return;

    swap( arr[l], arr[r])
     f(l+1, r-1);
   }
main() {
    input (arr) ;
    f(0, n-1);
  }
```

→ It will swap until (l>=r) this condition will get true.

1	2	4	5	3

② Method - II

```
f(i) {
 if( i >= n/2)
    return;
 swap( arr[i] , arr[n-i-1]);
  f(i+1) ;
 }

main() {
    input (arr) ;
    f(0);
}
```

check if a string is palindrome

"MADAM" —reverse→ "MADAM" ⇒ This is palindrome

```
f(i) {
    if(i>=n/2)
        return true;

    if(s[i] != s[n-i-1])
        return false;

return f(i+1);
}
```

Multiple Recursion calls

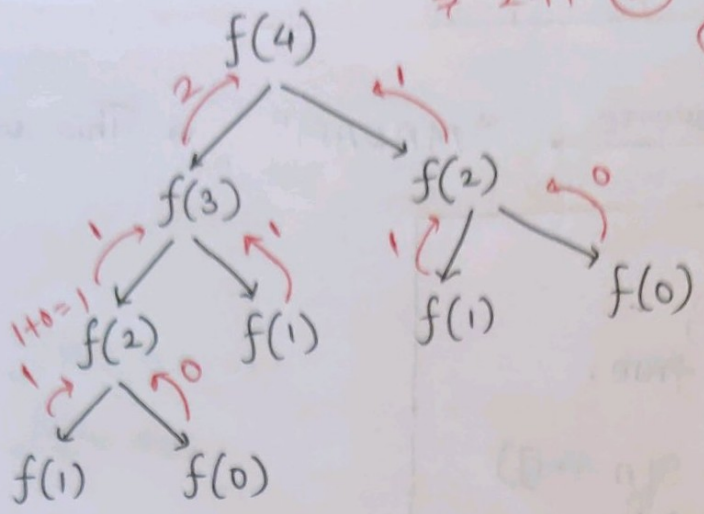
① Fibonacci Number

0 1 1 2 3 5 8 13

0th 1st 2nd 3rd

f(N) ⟶ N^{th} Fibonacci Number

```
f(n) {
    if(n<=1)
        return n;

    return f(n-1) + f(n-2);

}
```



Print all Subsequence :-

A contiguous / non-contigous sequence which follows the order

arr = [3, 1, 2]

[ ]
[3]
[1]
[2]
[3, 1]
[1, 2]
[3, 2]
[3, 1, 2]

} #. Subsequence = 8

#. Subsequence = 2^n

arr = | 3 | 1 | 2 |

Take | Non-take
↓
on indices

for [3,2]

	Take	X	Take
i ⇒	0	1	2
[1,2]	X	✓	✓
[3,1]	✓	✓	X
[3,1,2]	✓	✓	✓
{ }	X	X	X

V. Imp

arr | 3 | 1 | 2 |
0 1 2

```
f(ind, [])
{
  if(ind >= n)
      print([]);
      return;

  [].push_back(arr[i]);
   f(ind+1, []);  ——————→ Take
  [].pop_back(arr[i]);   (remove)
   f(ind+1, []);  ——————→ Not-take
}

main() {
  arr = {3,1,2}
   f(0,[]);
}
```

```
f(ind, []){
  if(ind >= n)
    print([]);
    return;
[3] [].push_back(arr[i]);
  f(ind+1, []);  → Take
  [].remove(arr[i]);
  f(ind+1, []);
}
```

f(1, [3])
{ if() ✗

[3,1] [3].pb(arr[i]);
f(2, [3,1]);

[3,1].remove(arr[1]);

f(2, [3,1])
{ if() ✗

[3,1].pb(arr[2]) ⇒ [3,1,2]

f(3, [3,1,2]);

[3,1,2].remove(arr[2]);

⇒ [3,1]

f(3, [3,1]);

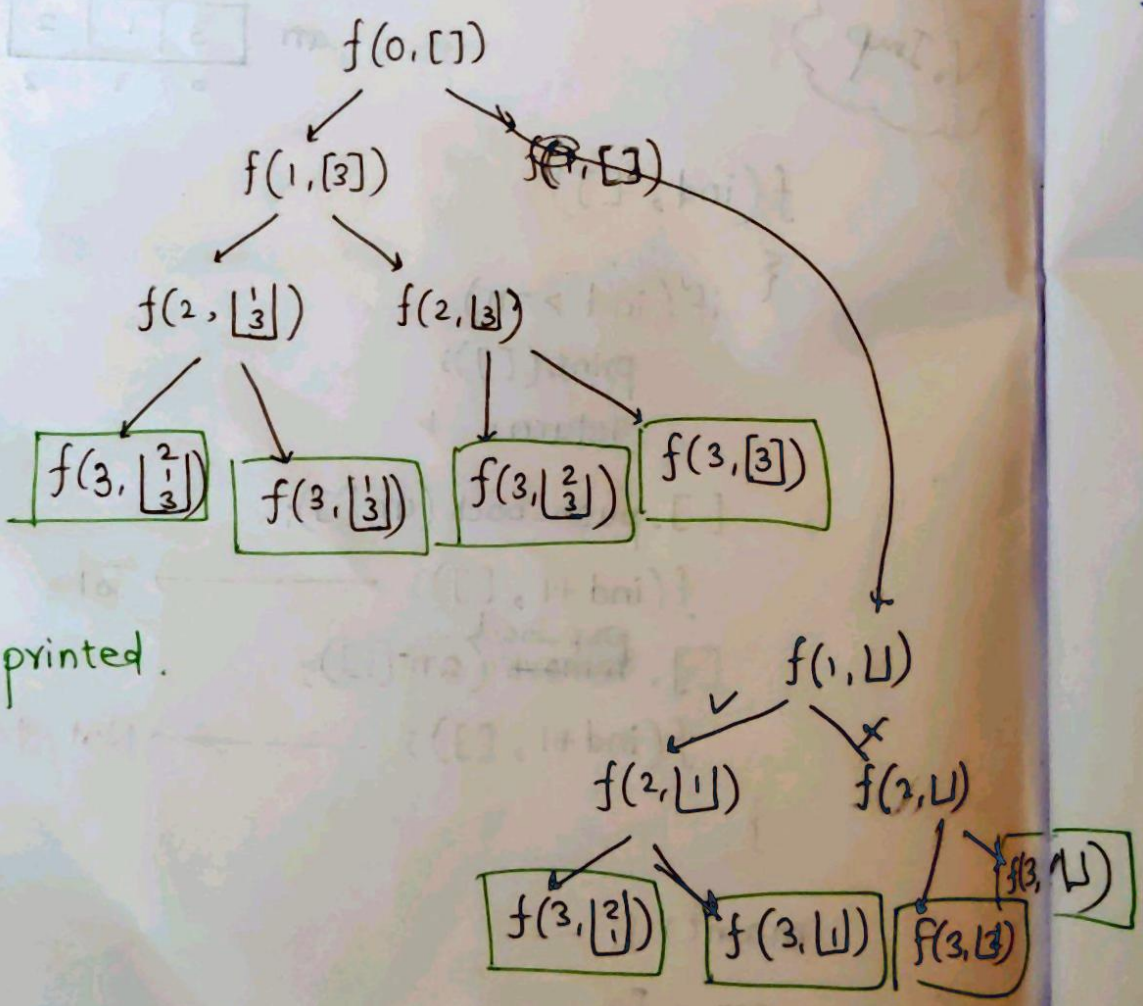
f(3, [3,1,2])

base condition ✓

f(3, [3,1]);

base Condition ✓

In this way the whole code is working



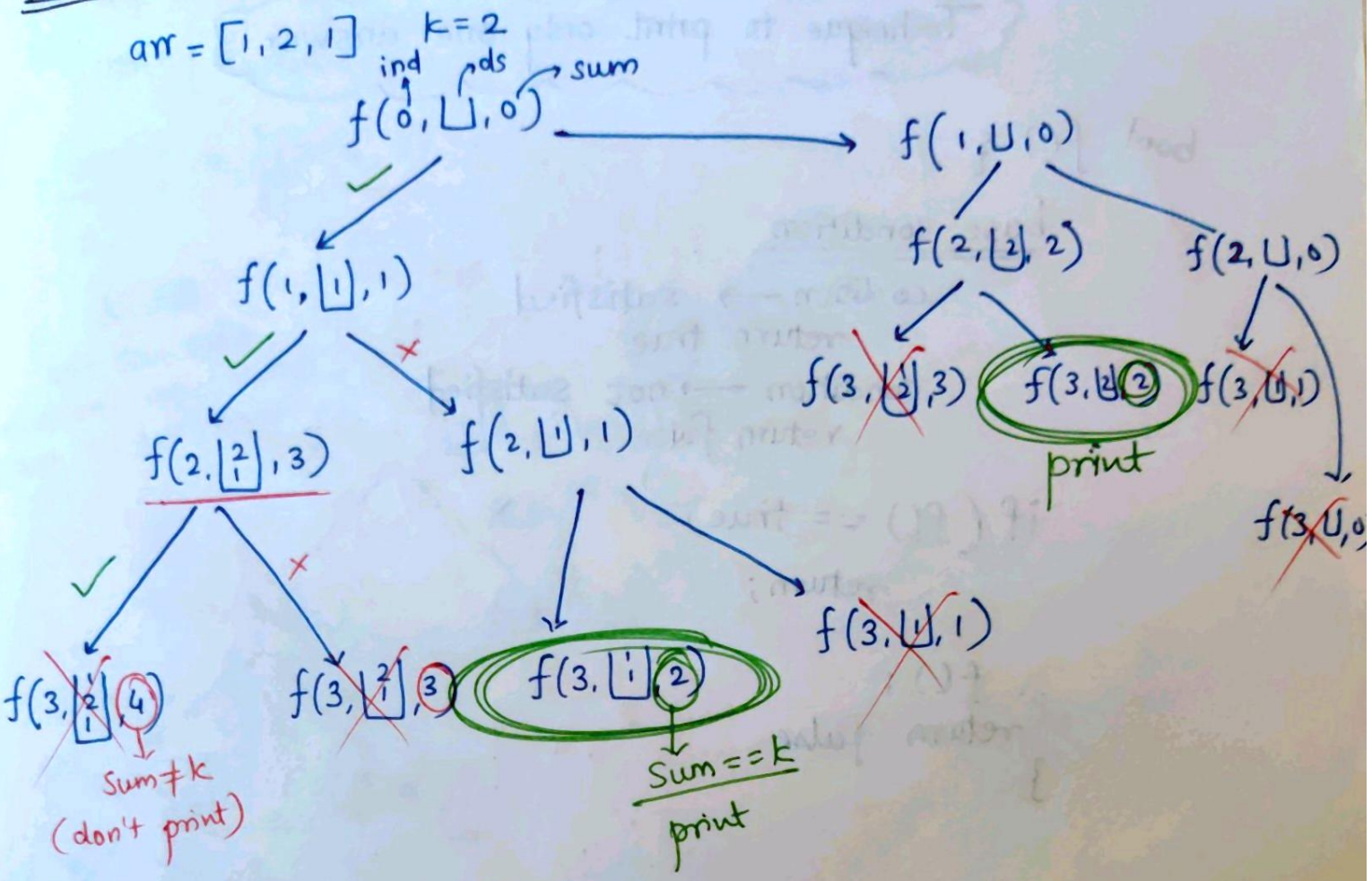
Output

[3,1,2]

[3,1]

⇒ printed.

Printing all subsequences whose sum is k




```
f(ind, ds, sum)
{
  if (i == n)
    if (sum == k)
      print(ds);
    return;

  ds.push_back(arr[ind]);
  sum += arr[ind];

  f(ind+1, ds, sum);

  ds.pop_back();
  sum -= arr[ind];

  f(ind+1, ds, sum);

}
```

- **Print any subsequence whose sum is k.**

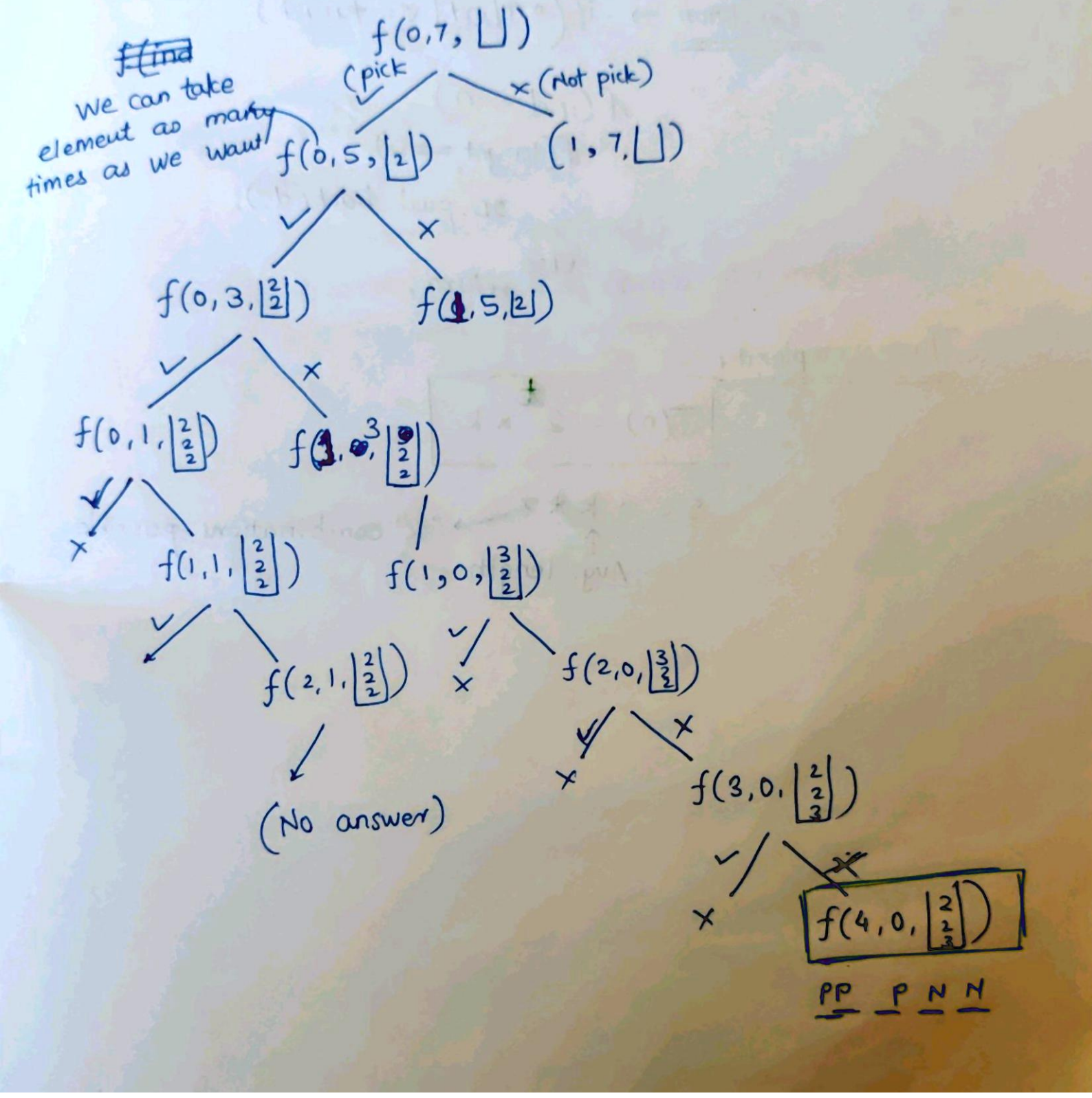
Technique to print only one answer

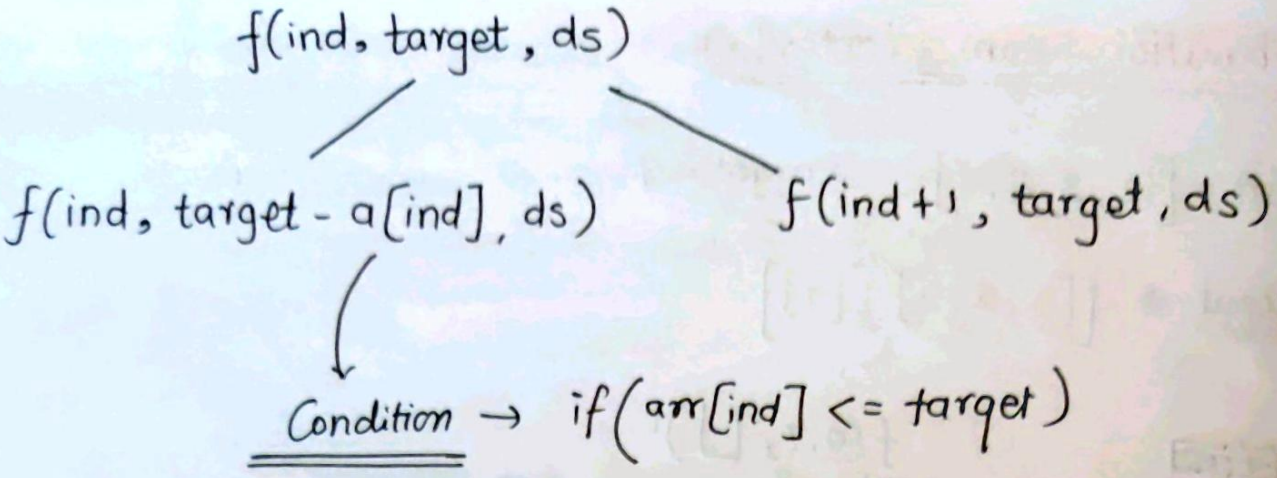
```
bool f() {
    base condition
        codition ⟶ satisfied
            return true
        condition ⟶ not satisfied
            return false

    if (f() == true)
        return;

    f();
    return false
}
```

Combination Sum (Leetcode)

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

Output ⇒ [[2, 2, 3], [7]]





Condition → if(arr[ind] <= target)

```
if (ind == n)
    if(target == 0)
        2D.push_back(ds);
else return;
```

Time Complexity :-

$$T(n) = 2^t \times k$$

S.C. = k * x → 'x' combinations possible

↑ Avg. length

Code:-

```
findCombination(int ind, target, arr, vector<vector<int>>&ans, ds)
    if(ind == arr.size())
        if(target == 0){
            ans.push_back(ds);
        }
        return;
    //pick the element
    if(arr[ind] <= target){
        ds.push_back(arr[ind])
        findCombination(ind, target -arr[ind], arr, ans, ds);
        ds.pop_back();
    }
    // not pick
    findCombination(ind+1, target, arr, ans, ds);
```

→ when upper recursion is totally executed and we have not got our answer so we have to empty our ds. (pop_back() that element)

Combination Sum - II (Leetcode)

arr[] and target will be given to us. Find all unique combinations in "arr[]" where array elements sums to target.

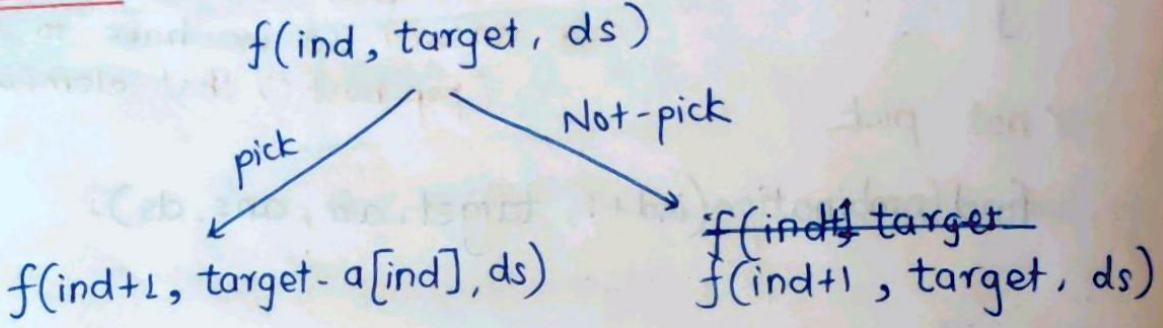
⇒ Each number can be taken only once.

⇒ The solution set must not contain duplicate combinations.

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

Combinations → [[1, 1, 2], [2, 2]]

Brute Force :-



⇒ If we pick the element then we have to move the "ind" by 1 cause in this problem we can take ~~any~~ element only once.

```
findCombination(ind, target, arr, set<vector<int>> &ans, ds)
    if (ind == arr.size()) {
        if (target == 0) {
            ans.insert(ds); }
        return;
    }
    if (arr[ind] <= target){
        ds.push_back(arr[ind]);
        findCombination(ind+1, target - arr[ind], arr, ans, ds);
        ds.pop_back();
    }
    findCombination(ind+1, target, arr, ans, ds);
```

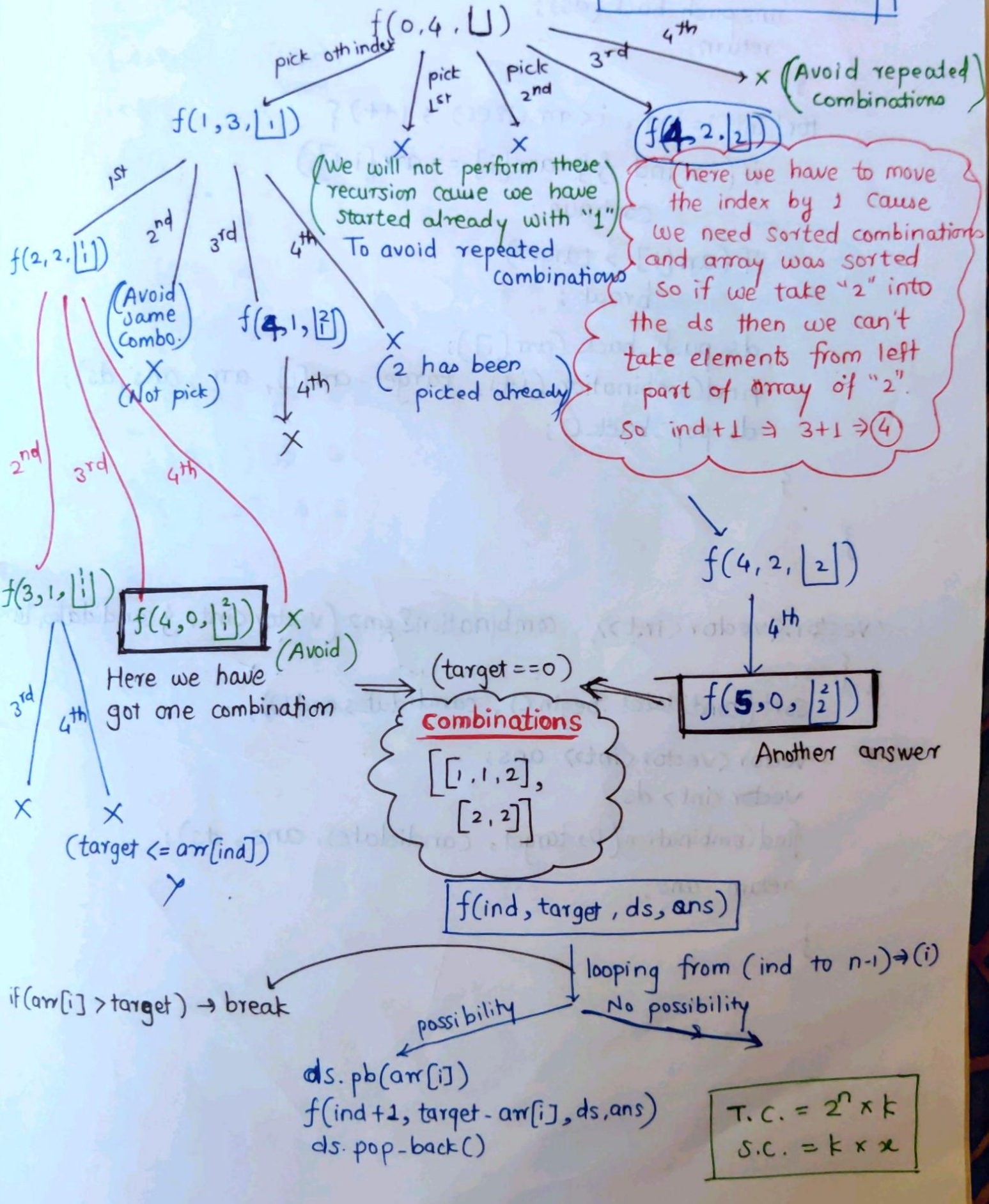
modified approach :-

Instead of pick and non-pick we will pick subsequences.

arr[] = [1, 1, 1, 2, 2]
0 1 2 3 4

target = 4

first sort the array



Code

```
void findCombination(ind, target, arr, vector<vector<int>>&ans, ds){
    if(target==0){
        ans.push_back(ds);
        return;
    }
    for(int i=ind; i<arr.size(); i++){
        if(i>ind && arr[i]==arr[i-1])
            continue;
        if(arr[i]>target)
            break;
        ds.push_back(arr[i]);
        findCombination(i+1, target-arr[i], arr, ans, ds);
        ds.pop_back();
    }
}

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

Subset-Sum - I

Given an array. print sums of all subsets in it.
Output should be printed in increasing order of sums.

Ex [3,1,2] n=3

	Sum
{ } ⇒	0
{3} ⇒	3
{1} ⇒	1
{2} ⇒	2
{3,1} ⇒	4
{1,2} ⇒	3
{3,2} ⇒	5
{3,1,2} ⇒	6

o/p

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

for any 'n'
The no. of subsets = 2^n

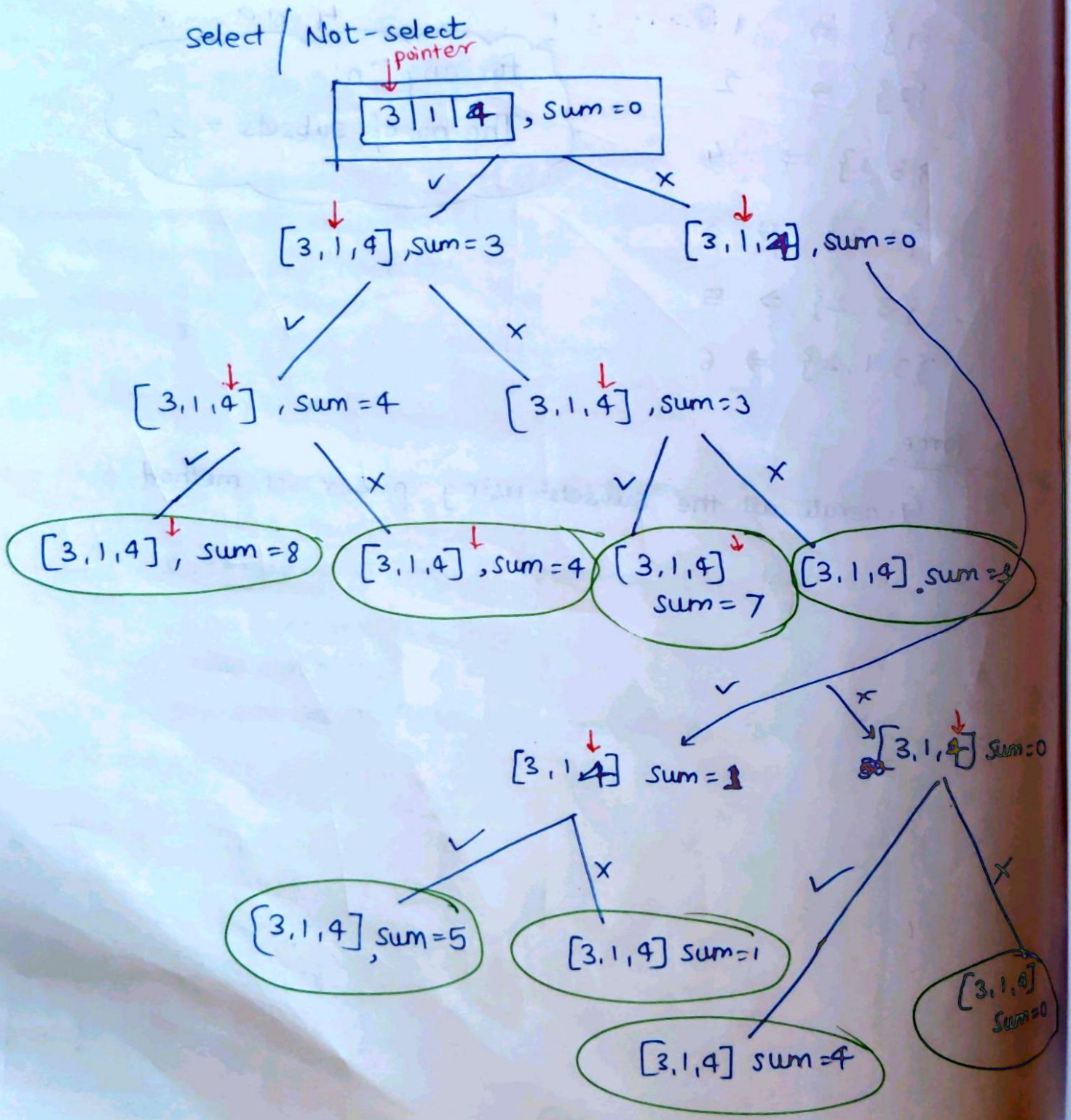
Brute force

Generate all the subsets using power set method

[3, 1, 4]

✓ (0) ✗ (1) ✓ (2) ⇒ [3, 4]

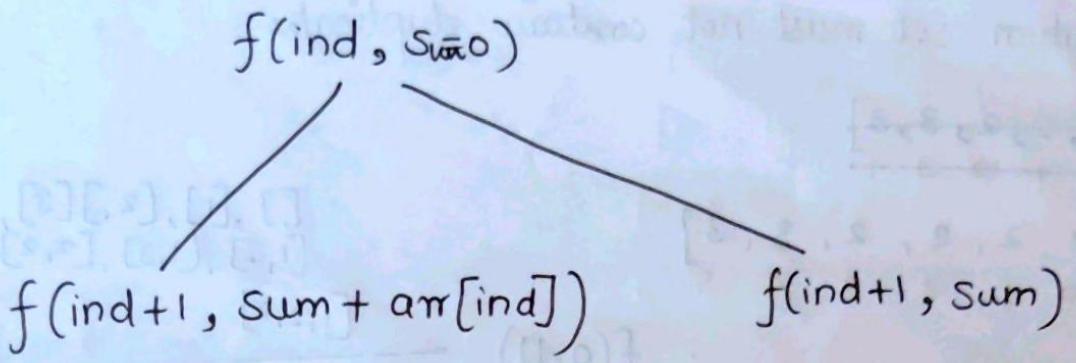
✓ (0) ✓ (1) ✗ (2) ⇒ [3, 1]



[8, 4, 7, 3, 5, 1, 4, 0]

Sort → [0, 1, 3, 4, 4, 5, 7, 8]

Recursive Calls



base case

```
if (ind == n)
    ds.push_back(sum);
```

T.C. = $2^n + 2^n \log(2^n)$

Subset Sum - II

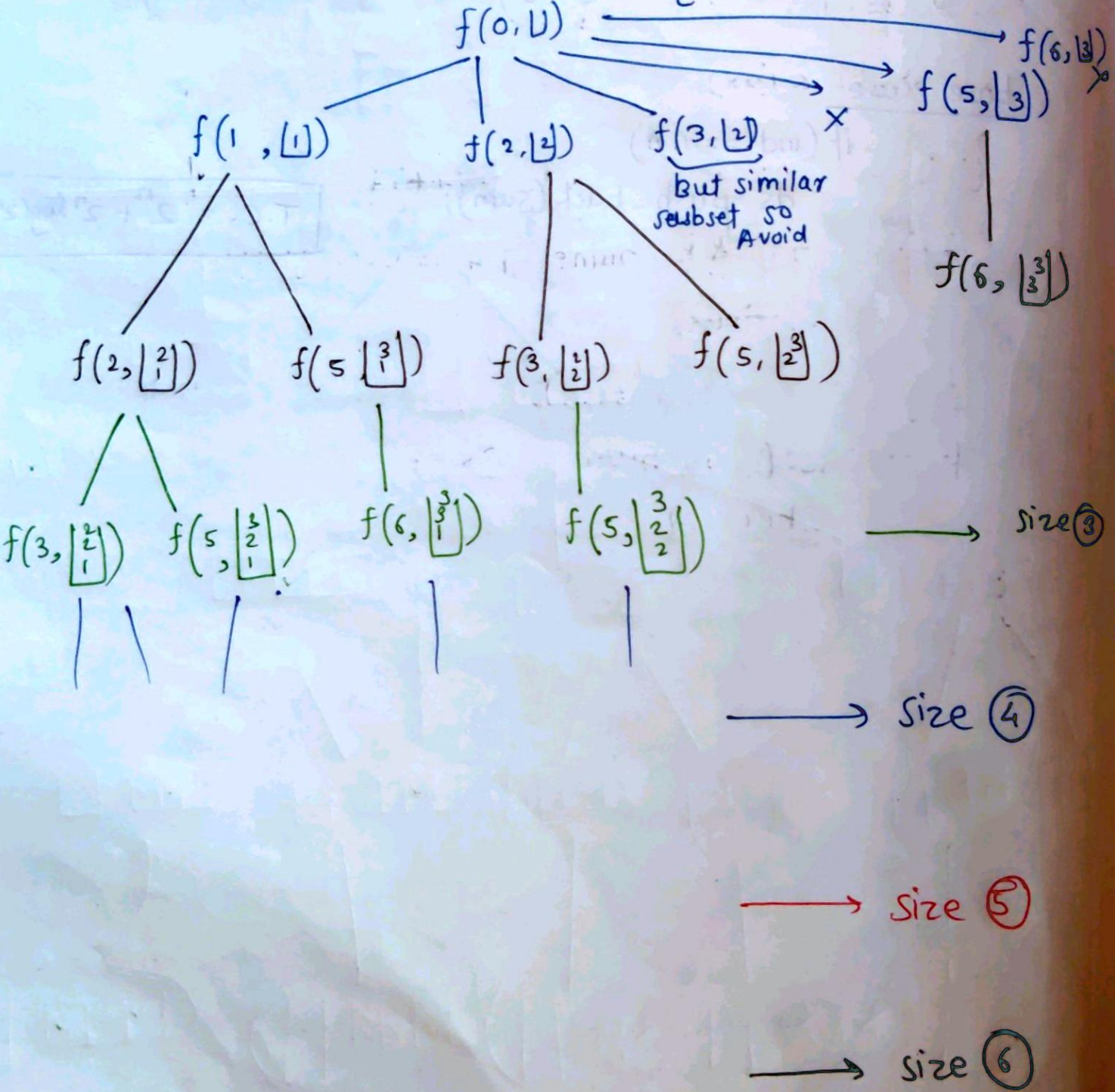
Given an integer array nums that may contains duplicates, return all possible subsets.

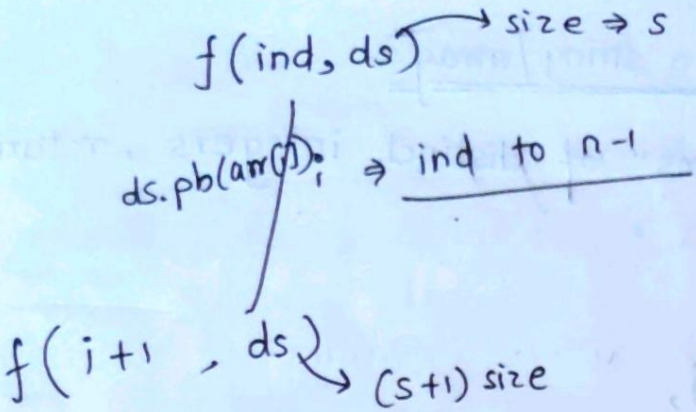
The solution set must not contain duplicates

~~arr = [1, 2, 2, 3, 3]~~
~~0 1 2 3 4~~

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

[], [1], [2], [3],
[1,2], [1,3], [2,2], [2,3], [3,3]





Code

```
void findSubsets(ind, nums, ds, ans)
    ans.push_back(ds);

    for(int i=ind ; i<nums.size() ; i++){

        if(i != ind && nums[i] == nums[i-1])
                continue;

        ds.push_back(nums[i]);
        findSubsets(i+1, nums, ds, ans);
        ds.pop_back();
    }
```

• Print all permutations of a string/array :-

Given an array "nums" of distinct integers, return all possible permutations.

Ex :- nums = [1, 2, 3]

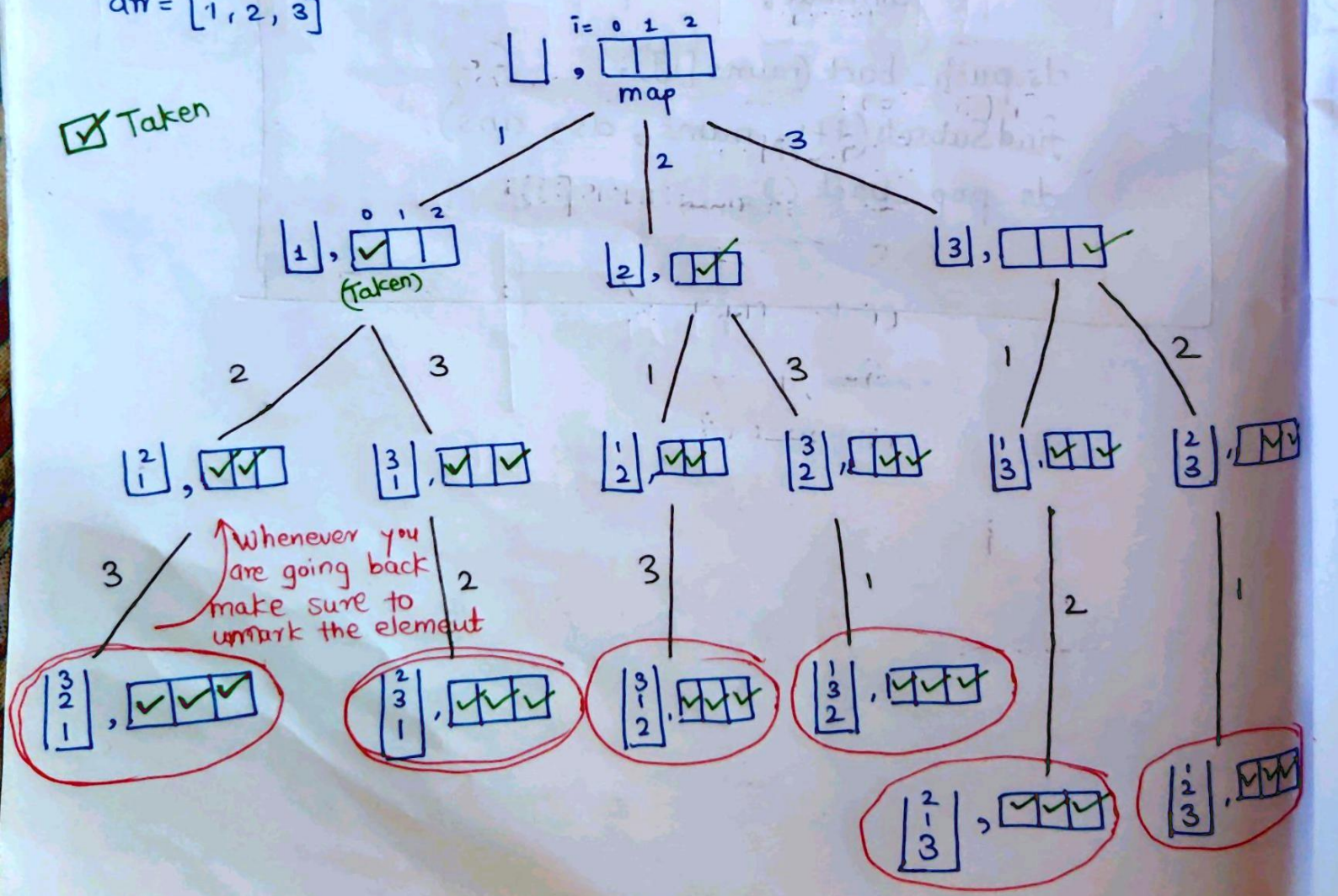
output ⇒ [[1,2,3],
[1,3,2],
[2,1,3],
[2,3,1],
[3,1,2],
[3,2,1]]

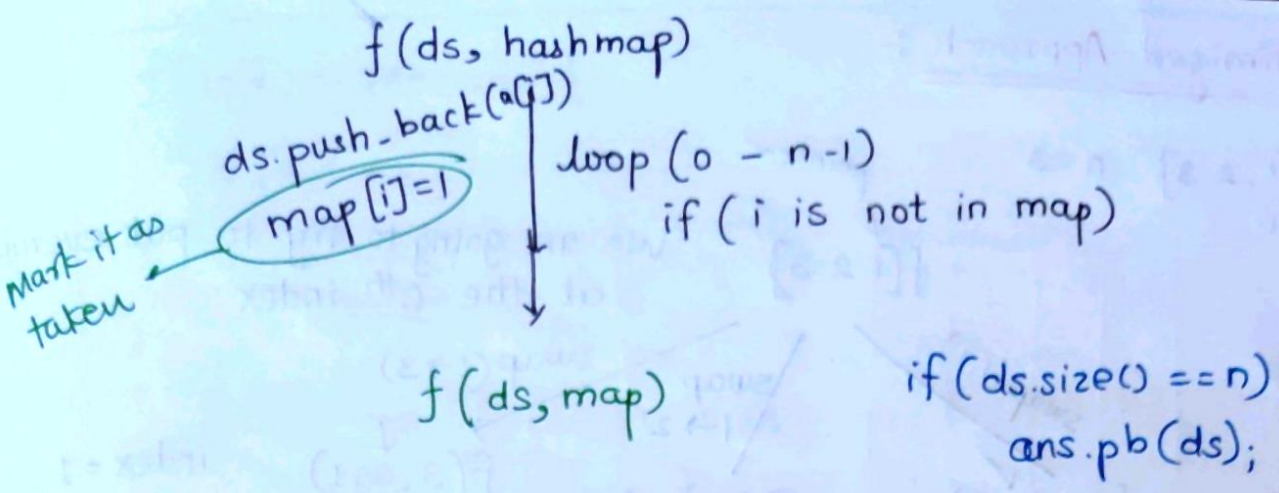
① Using Extra Space Complexity :-

If n elements is given then

No. of possible permutations = n!

arr = [1, 2, 3]





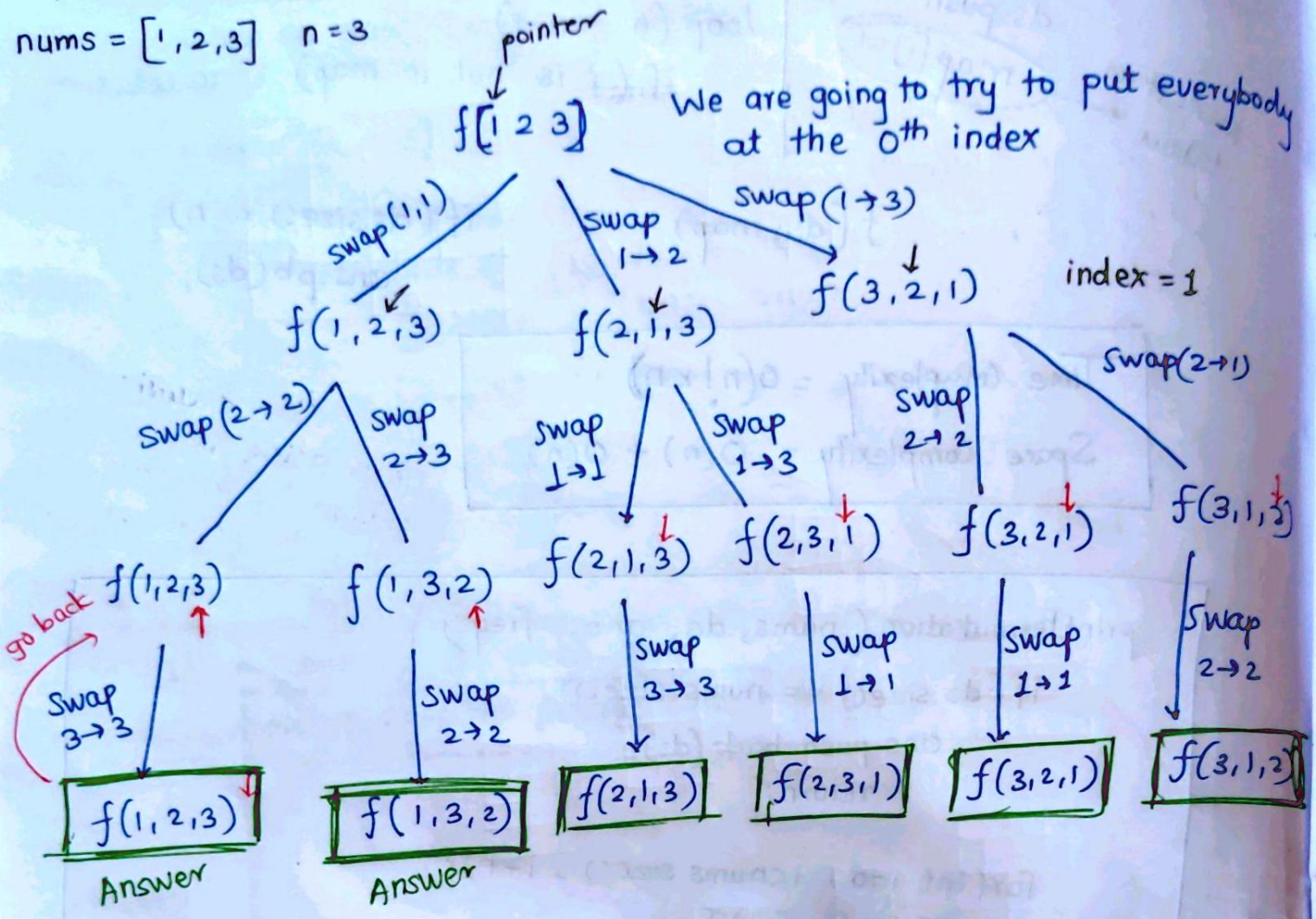
Time Complexity = $O(n! \times n)$

Space Complexity = $O(n) + O(n)$

```
printPermutation ( nums, ds, ans, freq ) {
    if( ds.size() == nums.size())
        ans.push_back (ds);
          return;

    for(int i=0 ; i<nums.size() ; i++){
        if ( !freq[i]) {
            ds.push_back(nums[i]);
            freq[i] = 1; ⟶ marked as picked
            printPermutation(nums , ds , ans , freq);
            freq[i] = 0; ⟶ unmark the position in freq
            ds.pop_back(); ⟶ Remove element from(ds);
        }
    }


main(){
    takeInput(nums);
    vector <int > ds;
    vector <vector <int>> ans;
    ~~for vector~~ int freq[nums.size()];
    for(int i=0 ; i<nums.size() ; i++)
          freq[i] = 0;
```

② Optimized Approach :-

nums = [1,2,3] n = 3




```
f(ind, arr)
    |
    |   i → (ind → n-1)
    |   swap(a[ind], a[i]);
    |
    v
f(ind+1, arr)
```

Base Case :-

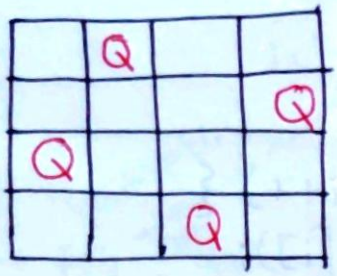
```
if(ind == n)
    ans.push_back(arr);
    return;
```

```
f(ind, nums, ans){
    if (ind == nums.size())
        ans.push_back(nums);
        return;

    for(int i=ind ; i<nums.size(); i++) {
        swap (nums[ind], nums [i]);

        f(ind+1, nums, ans);
        swap (nums[ind], nums [i]);
    }
```

→ After recursive call when going backwards we have to reswap the elements.

N-Queens

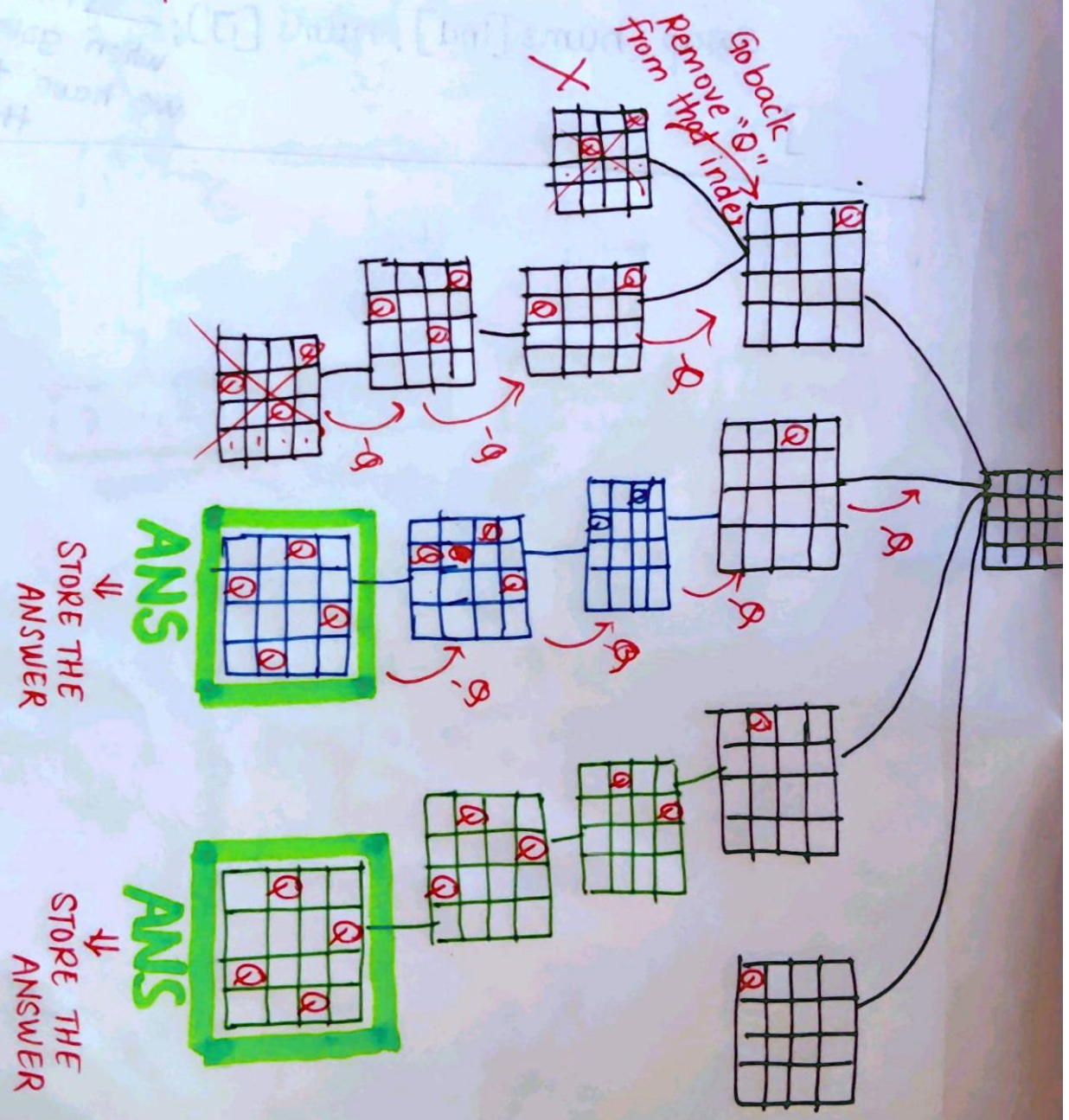
n=4

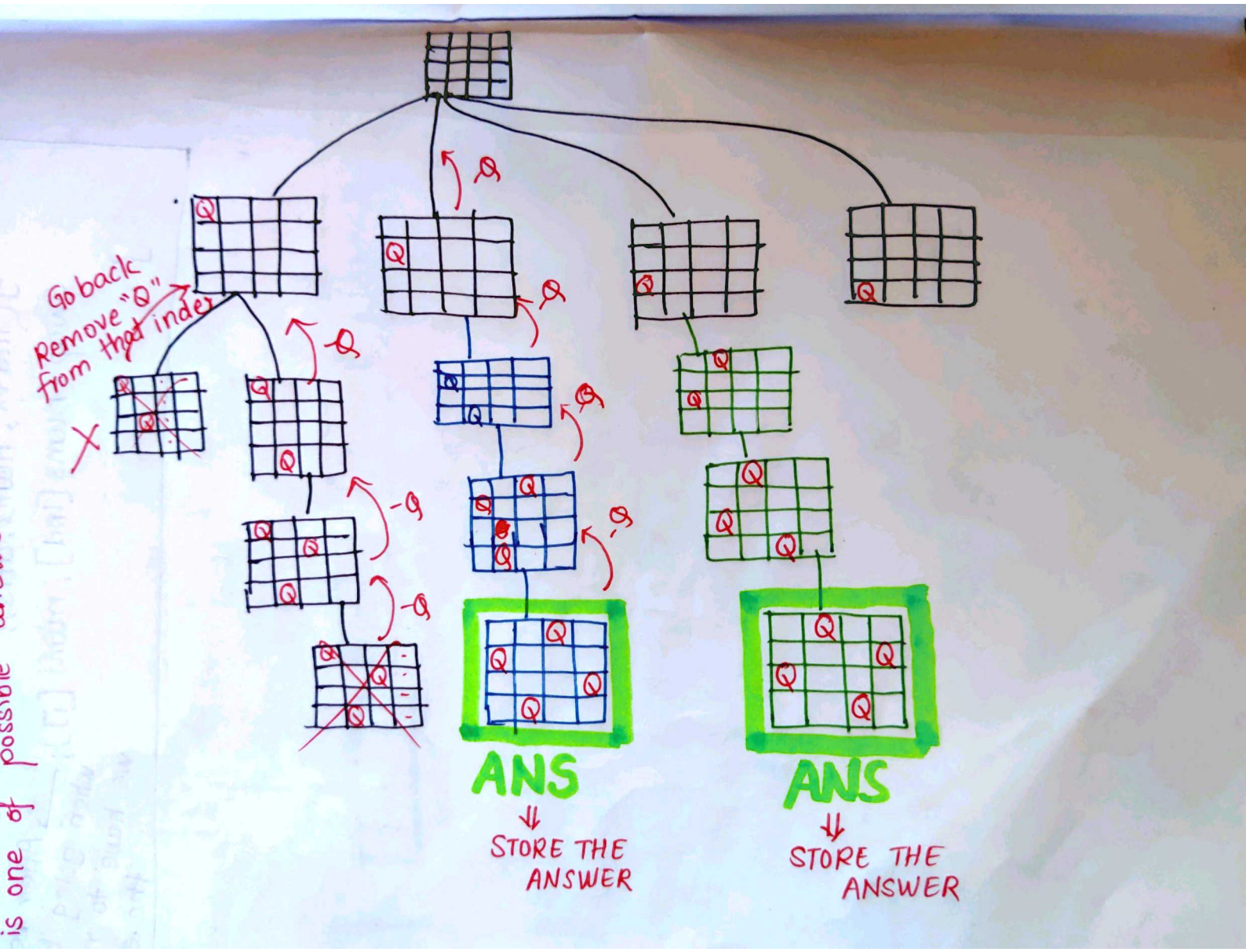


Rules

1. Every row should have a queen.
2. Every column should have a queen.
3. None of the Queen attack each other.

This is one of possible answer.





pseudo Code

```
if(col) {        → filling each column
    for(i=0 ; i<row ;i++){
        if(fill ✓){
            Board[row][col] = 'Q' :
            Solve(col+1);
            Board[row][col] = '.';
        }
    }
```

Code :-

```
bool isSafe(int row, int col, vector<string> board, int n){
    int duprow = row;
    int dupcol = col;
    while(row>=0 && col>=0){
        if(board[row][col] == 'Q') {
            return false;
        }
        row--;
        col--;
    }
    row = duprow;
    col = dupcol;
~~while(row<n &&~~
    while(col>=0) {
        if(board[row][col] == 'Q'){
            return false;
        }
        col--;
    }
```

```
    row = duprow;
    col = dupcol;
    while( row <n && col >= 0) {
        if (board[row][col] == 'Q') {
            return false;
        }
        row++;
        col--;
    }
    return true;
}

void Solve(int col, vector<string> &board,
                    vector<vector<string>> &ans, int n){
    if (col ==n) {
        ans.push_back(board);
        return ;
    }
    for(int row=0 ; row<n ; row++) {
        if( isSafe(row, col, board, n)) {
            board[row][col] = 'Q';
            Solve(col+1, board, ans, n);
            board[row][col] = '.';
        }
    }
}
```

```
vector<vector<string>> solveQueens(int n){
    vector<vector<string>> ans;
    vector<string> board(n);
    string s(n, '.');
    for (int i=0; i<n; i++){
        board[i] = s;
    }
    solve(0, board, ans, n);
    return ans;
}
```

Sudoku Solver

1) The digit 1-9 appears only once in a row.
2) The digit 1-9 appears only once in a column.
3) The digit 1-9 appears only once in 3x3 grid.

Code :-

```
void SolveSudoku(vector<vector<char>>& Board) {
        Solve(board);
    }

bool Solve(vector<vector<char>>& board){
  for(int i=0 ; i< board.size() ; i++){
    for(int 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) == true)
                        return true;
                    else
                        return false; board[i][j] = '.' ;
                }
            }
        return false;
        }
    }
  }
  return true;
}
```

```
bool isValid(vector<vector<char>>& board, int row, int col, char c) {
    for(int i=0 ; i<9 ; i++) {
        if(board[i][col] == c)
            return false;

        if (board[row][i] == c)
            return false;

        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 the graph can be colored with at most M colors such that no two adjacent vertices of the graph are colored with the same color. Here coloring of graph means the assignment of colors of all vertices. Print 1 if possible to colour all the vertices and 0 otherwise.

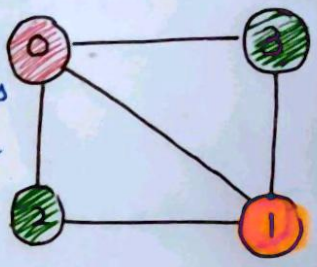
Ex:-

N = 4
M = 3
E = 5

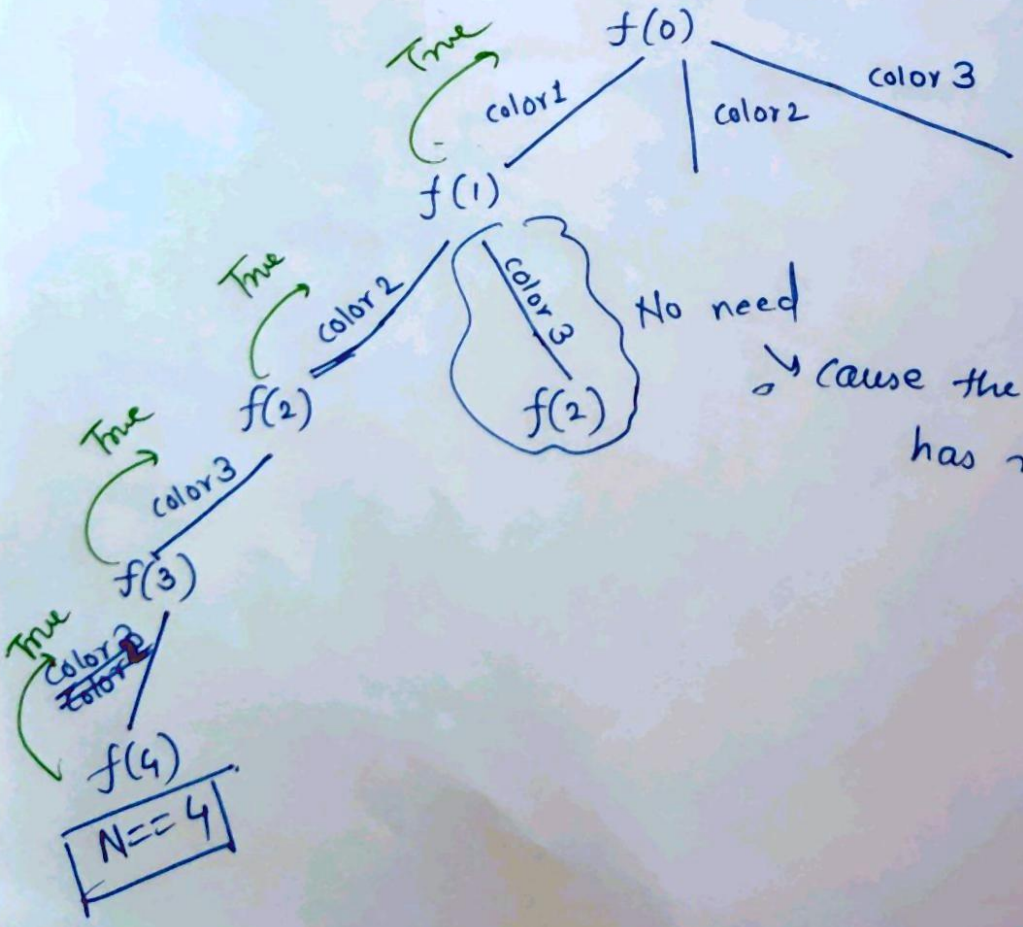
Edges[] = {(1,2), (2,3), (3,4), (4,1), (1,3)}

o/p = 1

None of the adjacent nodes have the same color Hence we can color the graph.



M = 3
↓
No. of colors



pseudo code :-

```
f(node) {
    if (node == N) ⟶ Base case
        return true

    for (color = 1 ⟶ M) {
        if (possibleColor(node)) {
            color[node] = col ;
            if(f(node + 1) == true) ⟶ if it giving ⇒ T
                return true ;          then we can return true
            color[node] = 0 ⟶ Remove the color
            }
        }
    return false ; ⟶ It's not possible to color the
                     graph.
}
```

- **Palindrome Partitioning** :-

Given a string s, partition 's' such that every substring of the partition is a palindrome. Return all possible palindrome partition of string 's'.

Ex :- s = "aab"

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

Ex :- ① s = "aabb"

→ if a|a|b|b → then partition { "a", "a", "b", "b" }

partions

a|a|b b → { "a", "a", "bb" }

a a|b|b → { "aa", "b", "b" }

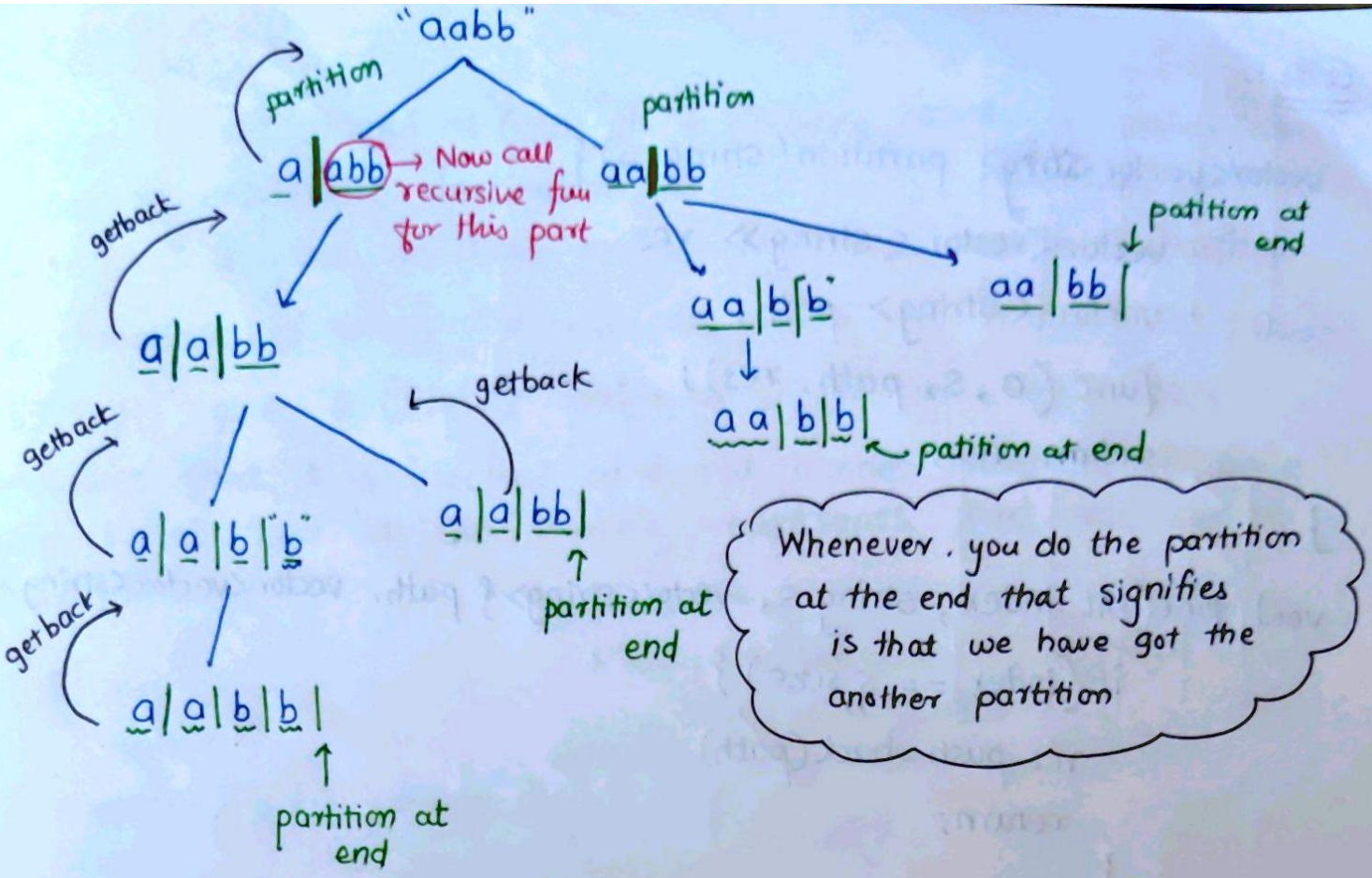
aa|bb → { "aa", "bb" }

Output :-

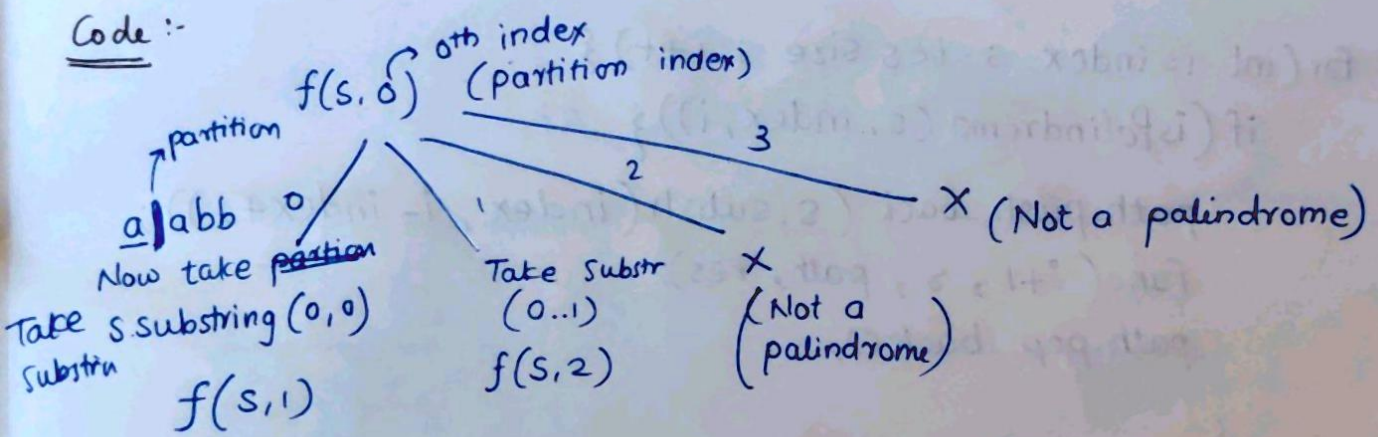
{ { "a", "a", "b", "b" },
{ "a", "a", "bb" },
{ "aa", "b", "b" },
{ "aa", "bb" } }

⇒ as 1 <= S.length() <= 16

so we can apply brute force.



Code :-



f(s, ind)

↓

loop | (ind → (n-1)) → i

if (ind ... i) till → substring is palindrome then only I can partition then only I can call the next recursion.

When (ind == n)

Code :-

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

• Rat in a maze :-

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

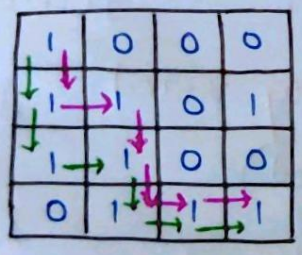
The answer should be printed in lexicographically

Input :-

n = 4

matrix[][] = { {1, 0, 0, 0},
{1, 1, 0, 1},
{1, 1, 0, 0},
{0, 1, 1, 1} }

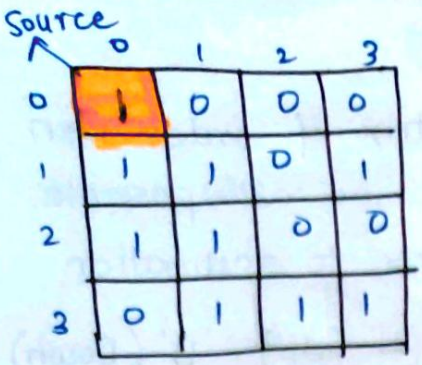
output ⇒ DDRDRR DRDDRR



DRDDRR
DDRDRD

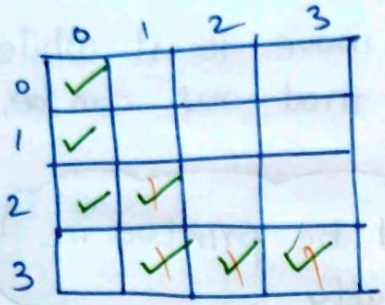
lexicographically →

DDRDRR ✗ DRDDRR



f(start, end, "")
f(0,0,"")

Lexicographically order of moves
D L R U



visited

f(0,0,"") (D,L,R,U)

f(1,0,"D") (D,L,R,U) → you cannot go upwards as upward cell is visited

f(2,0,"DD") (D,L,R,U)

f(2,1,"DDR") (D,L,R,U)

getback

f(1,1,"DDRU")

f(3,1,"DDRD") (D,L,R,U) (Already visited)

(Already visited)

f(3,2,"DDRDR") (D,L,R,U)

Now after returning, we have to mark (3,3) as unvisited → but we can't take any more!

f(3,3,"DDRDRR")

! We have reached to our destination!!

f(1, 0, "D")

Right move

f(1, 1, "DR") D L R U

f(2, 1, "DRD") D L R U

f(3, 1, "DRDD") D L R U

f(3, 2, "DRDDR") D L R U

f(3, 3, "DRDDRR");

Reached!

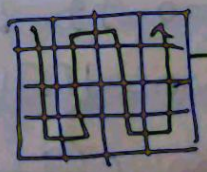
Pseudo Code

```
f() {
    Down
    f();
    Left
    f();
    Right
    f();
    Up
    f();
}
```

Time complexity = $O(4^{m \times n})$

Space Complexity = Depth of the maximum recursive tree

$O(m*n)$

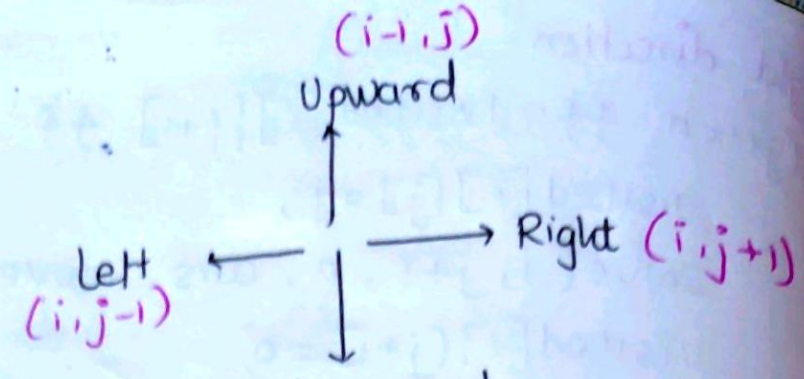


```
vector<string> findPath(vector<vector<int>>& m, int n) {
        vector<string> ans
        vector<vector<int>> visited(n, vector<int>(n,0));
        if(m[0][0] == 1)
            solve(0,0,m,n, ans, "", vis);
        return ans;
    }

void solve(int i, int j, vector<vector<int>>& a, int n, vector<string>& ans, string move,
                                                    vector<vector<int>>& visited)
    {
        if(i==n-1 && j==n-1)
            retur ans.push_back(move);
                return;

        // Downward direction
        if(i+1<n && !visited[i+1][j] && a[i+1][j]==1){
                visited[i][j] = 1;
                Solve(i+1, j, a, n, ans, move + 'D', visited);
                visited[i][j] = 0;
        }

        // Left direction
        if(j-1>=0 && !visited[i][j-1] && a[i][j-1]==1){
                visited[i][j] = 1;
                solve(i+1, j, a, n,
                solve(i, j-1, a, n, ans, move + 'L', visited);
                visited[i][j]=0;
        }
```

```
// Right direction
if (j+1<n && !visited[i][j+1] && a[i][j+1]==1){
    visited[i][j] = 1;
    Solve(i, j+1, n, ans, move + 'R', visited);
    visited[i][j+1] = 0;
}

// upward direction
if (i-1>=0 && !visited[i-1][j] && a[i-1][j]==1){
    visited[i][j] = 1;
    Solve(i-1, j, n, ans, move + 'U', visited);
    visited[i-1][j] = 0;
}
```

→ This is a bit lengthy code.

→ We can shorten this code by using for loop.



{D} {L} {R} {U}

di[] = {+1, +0, +0, -1}

dj[] = {+0, -1, +1, +0}

current index gets → +1
and 'j' gets → +0

So we can do

i + di[index]
j + dj[index]

→ Because of this you don't need to write 4 different if statements

```
void solve( int i, int j, vector<vector<int>> &a, int n, vector<string> ans,
        string move, vector<vector<int>> &visited, int di[], int dj[]){
    if(i==n-1 && j==n-1) {
        ans.push_back(move);
        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
         && !visited[nexti][nextj] && a[nexti][nextj]==1){
            visited[i][j] = 1;
            solve(nexti, nextj, a, n, ans, move + direction[index],
                                        visited, di, dj);
            visited[i][j] = 0;
    }
}
```

```
vector<string> findPath(vector<vector<int>>& m, int n) {
    vector<string> ans;
    vector<vector<int>> visited(n, vector<int>(n,0));
    int di[] = {1, 0, 0, -1};
    int dj[] = {0, -1, 1, 0};

    if(m[0][0] ==1)
        solve(0,0, m, n, ans, "", visited, di, dj);

    return ans;
}
```

k^{th} permutation Sequence

The set [1, 2, 3, ..., n] contains a total of "n!" unique permutations.

By listing and labeling all of the permutations in order, we get the following sequence for n = 3.

1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

Given 'n' & 'k' find the k^{th} permutation sequence.

Example 1 :-

Input :- n = 3, k = 3

output :- "213"

① Brute Force :-

1. Generate all the permutations by using recursion.
2. Store all the permutation into a data structure.
3. Sort the data structure.
4. Return the $(k-1)^{th}$ ~~perm~~ element from the ds.

Time Complexity :- $O(n! \times n)$

To generate all permutation

Store the permutation into a ds

② Optimal Solution :-

Example :- $n=4$, $k=17$

#. of permutations = $n!$
$= 4!$
$= 24$

[1, 2, 3, 4]

indexing $k=17$ → 16th permutation

(0-5) pick 1 + (2,3,4) ⟶ 6 permutations

(6-11) pick 2 + (1,3,4) ⟶ 6 permutations

(12-17) k will lie In this range pick 3 + (1,2,4) ⟶ 6 permutations

(18-23) pick 4 + (1,2,3) ⟶ 6 permutations

(0-based indexing)

1 2 3 4 ⟶ 0th permutation

⋮

4 3 2 1 ⟶ 23rd permutation

$k=17$ ⟶ 16th permutation

Each no. has 6 permutation

So $16/6 = 2$ index

0 1 2 3

[1, 2, 3, 4]

index means It will start from this number.

3 _ _ _

16 % 6 = 4th sequence out of

3 + (1, 2, 4)

this sequence

then our job will done.

[1, 2, 4] → Remaing numbers

and we want 4th permutation

So Now question is

[1, 2, 4] k = 4

Again Same process

(0-1) pick → 1 + {2, 4} → 2 permutations

(2-3) pick → 2 + {1, 4} → 2 permutations

(4-5) pick → 4 + {1, 2} → 2 permutation

kth permutation → 4/2 = (2)

[1, 2, 4] (0 1 2)

permutation is going to start with 4

3 4 _ _ ↘

4 % 2 = 0 → Now question boils down to

[1, 2] k = 0

Again Same process

[1, 2] k = 0

(0-0) pick → 1 + {2} → 1 permutations

(1-1) pick → 2 + {1} → 1 permutation

k^{th} permutation = 0/1 = 0 → [1, 2]

means permutation is going to start with 1.

3 4 1 _

↓

k = 0% 0 = 0 → [2] k = 0

of again same process

we will get

3412 ⇒ This is our answer!!

Summary :-

At first,

4 blocks { 0] 6, 1] 6, 2] (6), 3] 6 }

$6+6+6+6 = 24$

$4! = 24$

k^{th} → $16 \% 6 = \underline{4}$

3 blocks { 0] 2, 1] 2, 2] (2) }

$2+2+2 = 8$

$3! = 8$

2 blocks {] 1,] 1 }

$1+1 = 2$

$2! = \underline{2}$

In each pass, we are taking/picking 1 element out till (n) → so O(N)

like this.

Time Complexity : $O(N) * O(N) \Rightarrow O(N^2)$

n = 4 → we are looking for 4 numbers

so O(N) will take

Space Complexity = O(N)

Code

WHY? because initially we have made the to do the division in terms of blocks

if n=4 → (24) permutations but we had '4' blocks of [6] permutations

↑ This is the factorial

Hence (<n) is taken

for n = 4 → we have to compute 3!

```
string getPermutation(int n, int k){
    int fact = 1;
    vector<int> numbers;
    for(int i=1; i<n; i++){
        fact = fact * i;
        numbers.push_back(i);
    }
```

→ This will store [1, 2, 3]

```
    numbers.push_back(n);
```

→ This will add 'n' to last position

[1, 2, 3, 4]

```
    string ans = "";
    k = k-1;
```

→ (0-based) indexing hence we have reduced a value by 1.

```
    while(true){
        ans += to_string(numbers[k/fact]);
```

k = 16

↓ Initially we had 4 blocks of (6).

16/6 = we have done & we got '2' and then we have taken the 2nd number

[1, 2, 3, 4]

Taken ✓

Add it to ans.

```
        numbers.erase(numbers.begin()+(k/fact));
```

erase '3'

So 16/6 = 2

numbers.begin() + 2

0+2

(2)nd element get erased.

```
        if(numbers.size()==0) break;
        k = k % fact
```

→ After adding, our next 'k' value will be 16%6 = (4)

```
        fact = fact / numbers.size();
```

→ Now, block size will be '2' Hence we have to divide 6/3 = 2

This (3) is size after adding element to vec

```
    }
    return ans;
}
```