→ Recursion

↳ function calls itself to solve smaller instance of a problem.

↳ key-concept:

① Recursive-case:

↳ part of function that calls itself to solve a smaller version of the original problem.

② Base-case

↳ stopping condition/s for recursion. This prevents the function from calling itself indefinitely.

③ stackoverflow error

↳ without base case, recursion leads to infinite recursion, causing a stackoverflow.

→ Types of recursion

↳ ① Direct Recursion: function calls itself directly i.e f() → f()

② Indirect-recursion: function A calls f(x) B, which then calls f(x) A

↳ i.e A() → B() → A()

③ Tail recursion: recursive call is the last operation in the function

↳ i.e f(n, acc=1) {
if(n==0) return acc;
return f(n-1, acc*n);
}

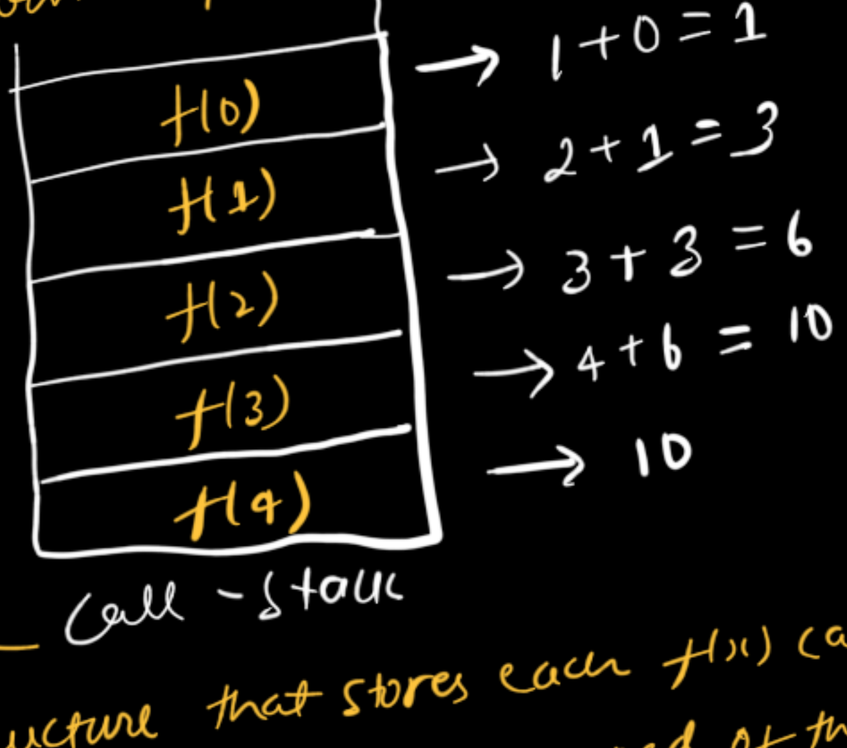
↳ last f(x) call

→ Find sum from 1 to n

```
f(int n){
  if(n==0) return 0;
  return n + f(n-1);
}
```

→

f(4) → 10
↓
4 + f(3) 4+6=10
↓
3 + f(2) 3+3=6
↓
2 + f(1) = 2+1=3
↓
1 + f(0) → 0



A memory structure that stores each f(x) call. when a f(x) returns, it's popped of the stack

→ Questions!

① Reverse Array & String
② Check if str is palindrome
③ digit sum & fibonacci
④ gcd & factorial
↳ TC: log(min(a,b))

→ Subsequence

↳ contigious / non-contigious, but it must preserve the original order of elements.

3	1	2

↳ Subsequence: { [3], [1], [2], [3,1], [3,2], [1,2], [3,1,2], [] }

pick = ✓

notpick = ✗

3	1	2

✓ ✗ ✓ → {3,2}

✓ ✓ ✗ → {3,1}

✗ ✓ ✓ → {1,2}

✗ ✗ ✓ → {2}

✓ ✗ ✗ → {3}

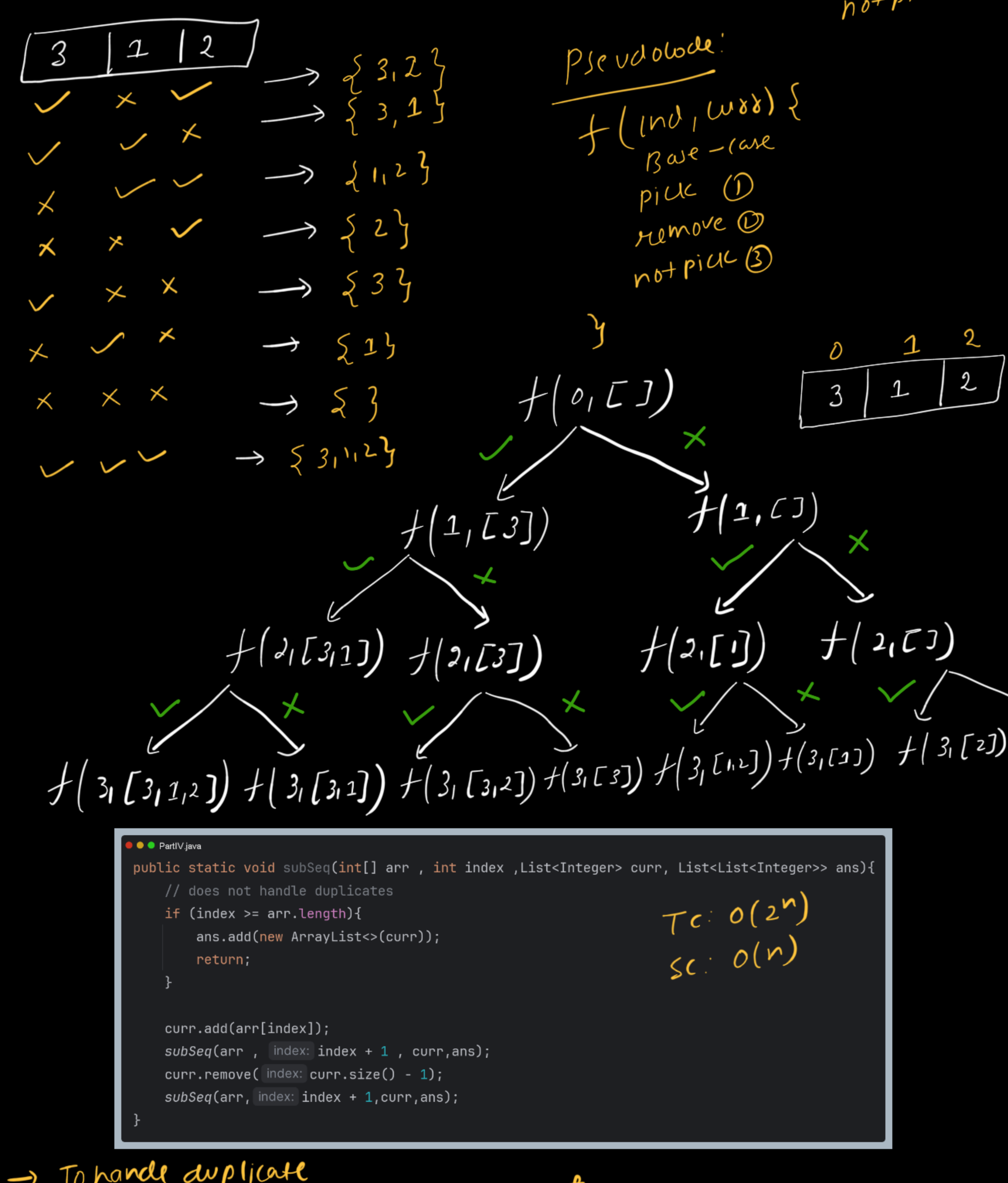
✗ ✓ ✗ → {1}

✗ ✗ ✗ → {}

✓ ✓ ✓ → {3,1,2}

Pseudocode:

```
f(ind, ans) {
  Base-case
  pick (1)
  remove (2)
  not pick (3)
}
```



PartIV.java

```java
public static void subSeq(int[] arr , int index ,List<Integer> curr, List<List<Integer>> ans){
    // does not handle duplicates
    if (index >= arr.length){
        ans.add(new ArrayList<>(curr));
        return;
    }

    curr.add(arr[index]);
    subSeq(arr , index: index + 1 , curr,ans);
    curr.remove( index: curr.size() - 1);
    subSeq(arr, index: index + 1,curr,ans);
}
```

Tc: $O(2^n)$

Sc: $O(n)$

→ To handle duplicate

↳ Sort the array, & before not picking up the element.

PartIV.java

```java
// skip all same adjacent element
int nextInd = index + 1;
while (nextInd < arr.length && arr[index] == arr[nextInd]) nextInd++;
```

→ Subset Sum Equal k

↳ Returns subsets whose sum Equal k

1	2	3

$K = 3$

↳ Ans = [[1,2], [3]]

→ When to return the result of a recursive call & when not to?

↳ Return recursive call?

① Yes ✓

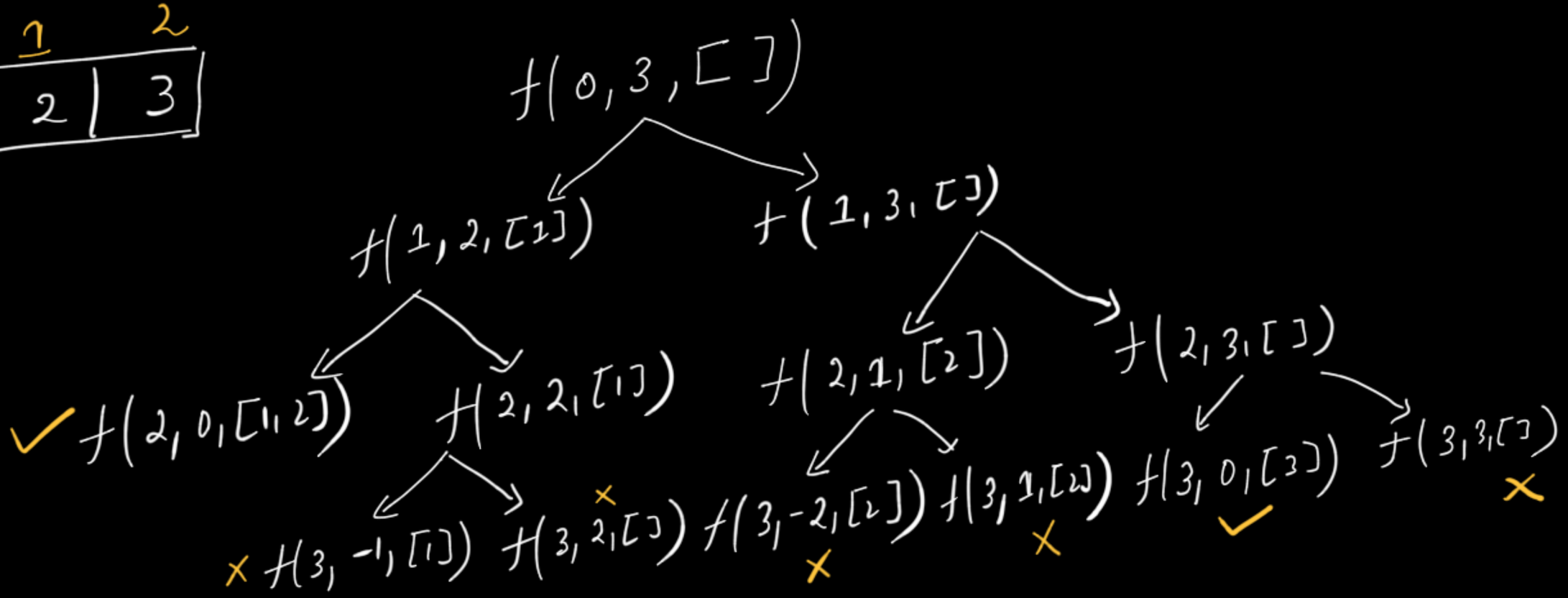
② No ✗

→ when the recursive call produces a value that need to be used, stored or returned

→ when the recursive call only perform actions (eg: modifying a list), & we're using shared mutable state.

when to use

0	1	2
1	2	3



PartV.java

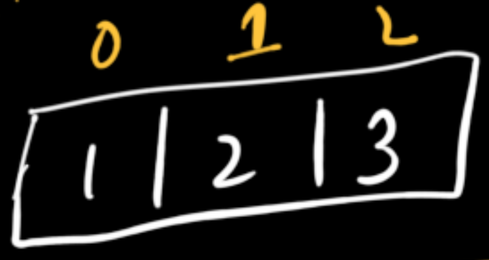
```java
public static void  3 usages  new *
subSumEqK(int[] arr, int index, int sum, List<Integer> curr, List<List<Integer>> ans) {
    if (sum == 0) {
        ans.add(new ArrayList<>(curr));
        return;
    }
    if (index >= arr.length || sum < 0){
        return ;
    }

    curr.add(arr[index]);
    subSumEqK(arr, index: index + 1, sum: sum - arr[index], curr, ans);
    curr.remove( index: curr.size() - 1);
    subSumEqK(arr, index: index + 1 , sum , curr , ans);
}
```

TC: $O(2^n) * n$ ↳ creating copy of the list

SC: $O(2^n * n)$

→ Count Subset Sum Equal K

↳ same question but instead of returning subset, only return count.



K = 3

index ← f(0, 3) → sum

2


```java
PartV.java
public static int subSumEqKCount(int[] arr, int index, int sum) {  3 usages  n
    if (sum == 0) {
        return 1;
    }
    if (index >= arr.length){   || sum < 0
        return 0;
    }

    int left = subSumEqKCount(arr, index: index + 1, sum: sum - arr[index]);
    int right = subSumEqKCount(arr, index: index + 1 , sum);

    return left + right;
}
```

TC: $O(2^n)$

SC: $O(n)$

↓

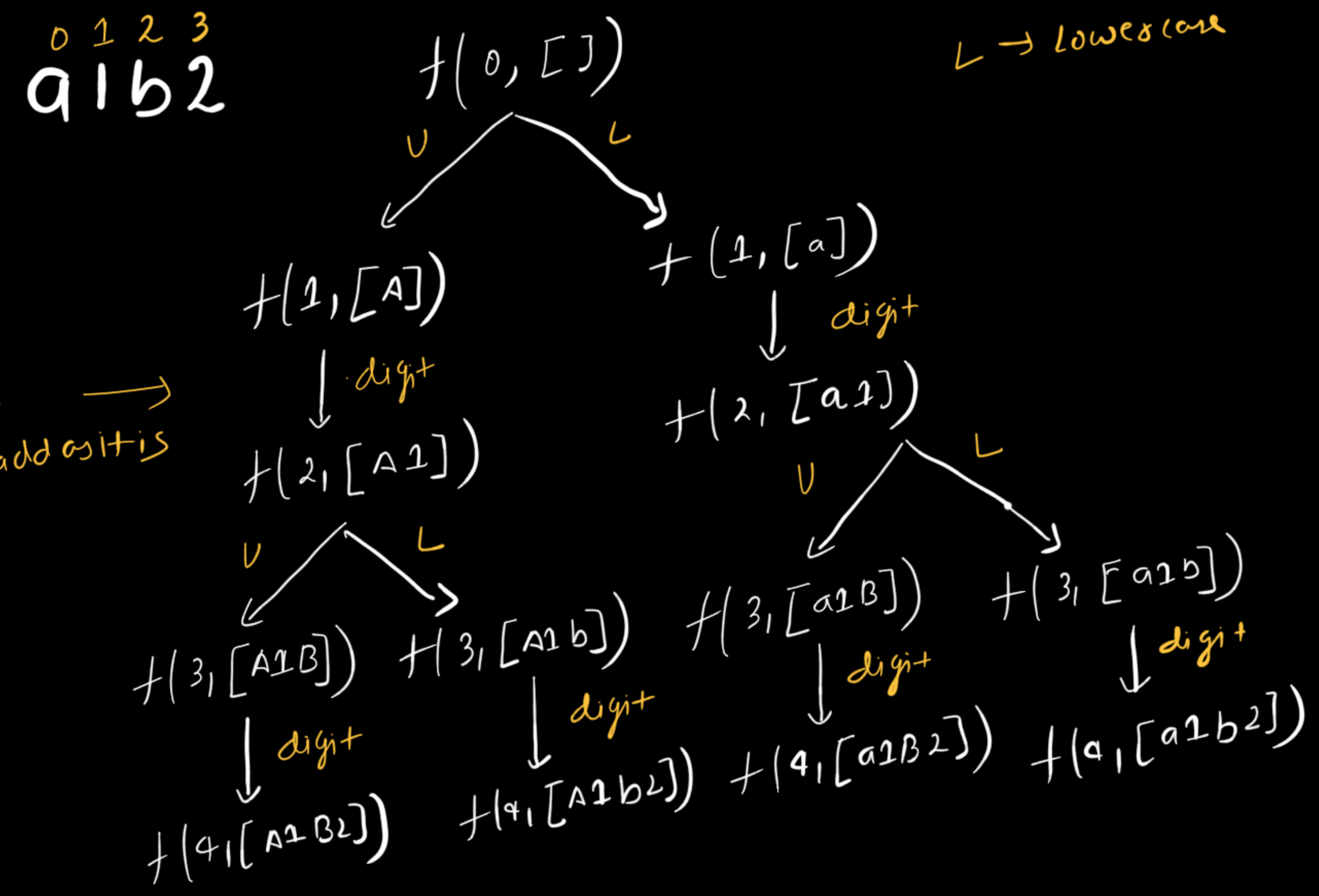
stack space

→ Letter Case Permutation (Uber | Oracle | Microsoft)

↳ Given a string s, we can transform every letter individually to be lowercase or uppercase to create another string.

s = a1b2 → [a1b2, A1b2, a1B2, A1B2]

U → uppercase
L → lowercase



PartVI.java

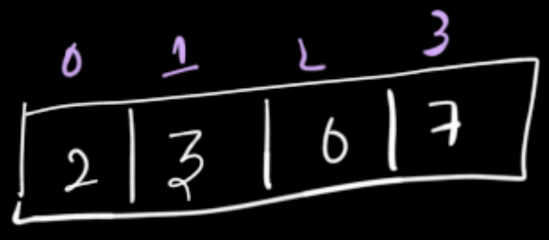
```java
private static void letterCasePermutation(String s, int index, String curr , List<String> ans) {
    if (index >= s.length()){
        ans.add(curr);
        return;
    }
    char ch = s.charAt(index);
    if (Character.isDigit(ch)){
        letterCasePermutation(s, index: index + 1 , curr: curr + ch , ans);
    }else {
        // lowercase
        char lo = Character.toLowerCase(ch);
        letterCasePermutation(s, index: index + 1 , curr: curr + lo , ans);

        //uppercase
        char up = Character.toUpperCase(ch);
        letterCasePermutation(s, index: index + 1 , curr: curr + up , ans);

    }
}
```

TC: $O(2^n)$
SC: $O(N) + O(N)$
↳ For storing result

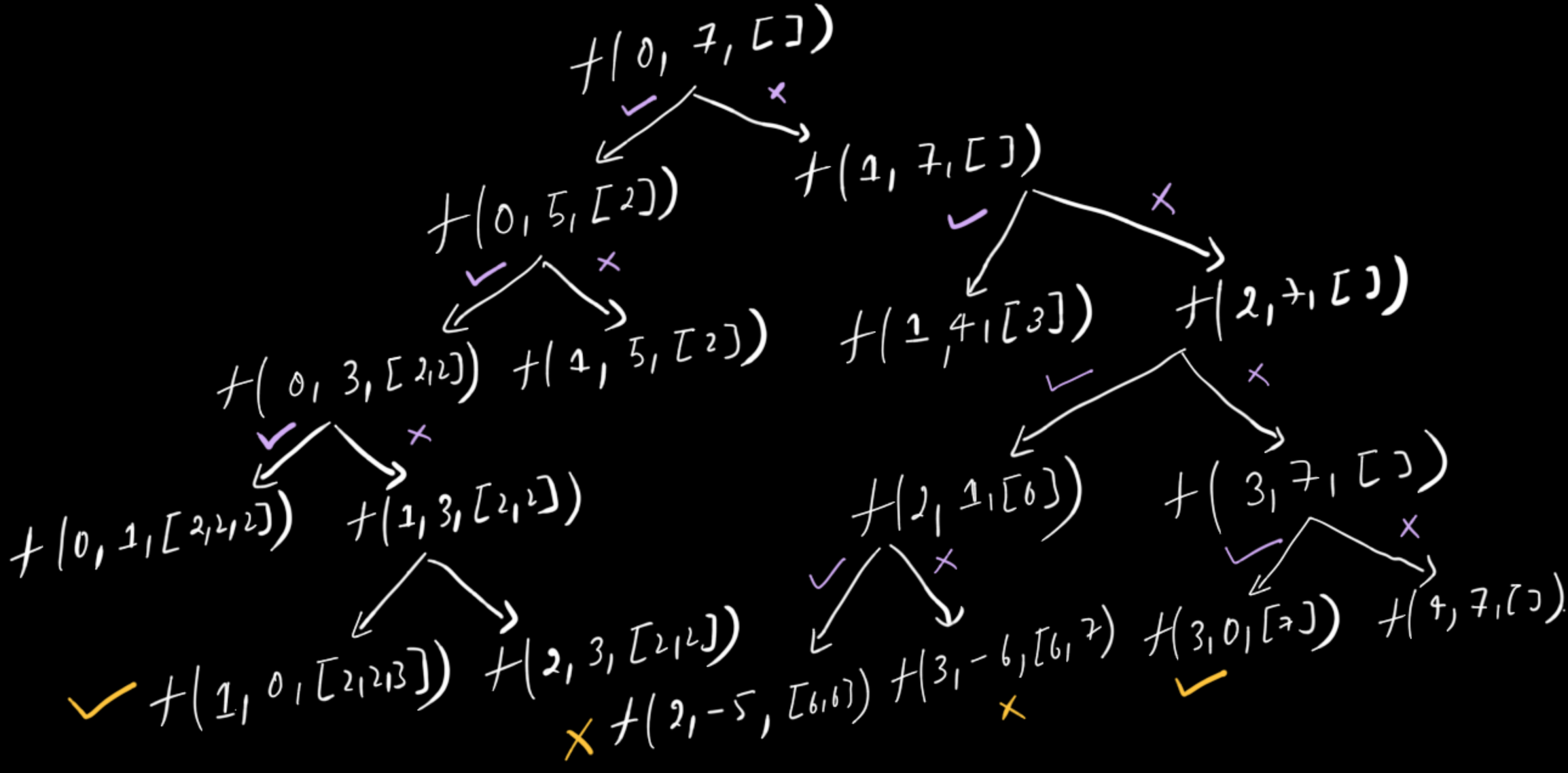
→ Combination Sum

↳ Return a list of all unique combinations, where chosen numbers sum up to target.

- The same number can be chosen unlimited number of times



target = 7



→ How do we achieve unique combination?

① Sort the array, so the duplicate element come adjacent to each other.

② Whenever we stumble upon similar element while not-pick option we just keep in-crementing the index i


```
//not-pick
int nextIndex = index+1;
while(nextIndex < arr.length &&
      arr[nextIndex] == arr[index]){
    nextIndex++;
}
combSum(arr, nextIndex, target, curr, ans);
```

PartVII.java

```java
public static void combSum(int[] arr, int index, int target, List<Integer> curr, List<List<Integer>> ans) {
    if (target == 0) {
        ans.add(new ArrayList<>(curr));
        return;
    }

    if (index >= arr.length || target < 0) return;

    // pick
    if (arr[index] <= target) {
        curr.add(arr[index]);
        combSum(arr, index, target: target - arr[index], curr, ans);
        curr.remove( index: curr.size() - 1);
    }

    // remove duplicates
    int nInd = index + 1;
    while (nInd < arr.length && arr[index] == arr[nInd]) nInd++;

    // not pick
    combSum(arr, nInd, target, curr, ans);
}
```

→ Combination Sum II

↳ similar question but, we can only choose a number once.

↳ solution

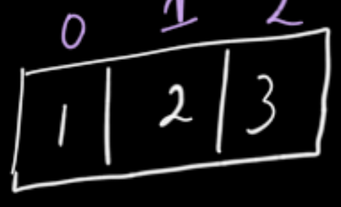
↳ do index+1 while picking an element.

TC: $O(2^n \times n) + n \log n$ ↘ sorting

SC: $O(n) + O(\text{ans size})$

→ Subset - sum

↳ Return sum of each derived subset in a list.



→

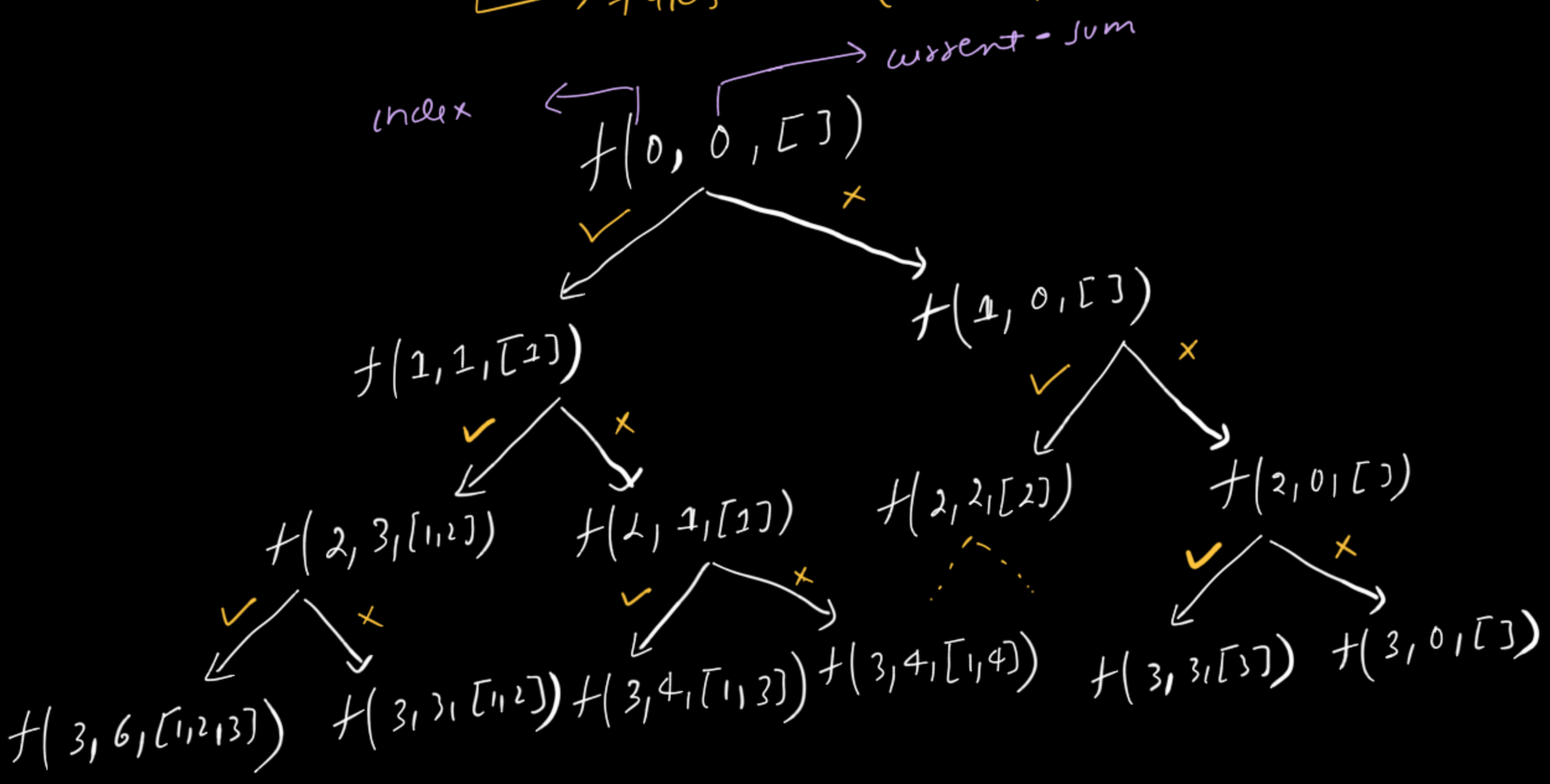
Gen. subset { } {1} {2} {3} {1,2} {2,3} {1,3} {1,2,3}

Sum 0 1 2 3 3 5 4 6

→ Brute force - approach

↳ Generate all power-set & then sum all the element individually.

↳ takes → $O(2^n \times n)$ → Time complexity.



PartVIII.java

```java
public static void subsetSum(int[] arr , int index , int currSum , List<Integer> ans){
    if (index >= arr.length){
        ans.add(currSum);
        return;
    }

    // pick
    subsetSum(arr, index: index + 1 , currSum: currSum + arr[index] , ans);

    // not pick
    subsetSum(arr, index: index + 1 , currSum , ans);
}
```

TC: $O(2^n)$

SC: $O(n) + O(2^n)$

↙ stack-space

↘ ans list size

→ Permutation

↳ generate all permutation of array / string.

| 1 | 2 | 3 | → [[1,2,3], [2,3,1], [3,1,2], [3,2,1], [2,1,3], [1,3,2]]

↳ all permutation.

→ Approach - 1

0	1	2
1	2	3

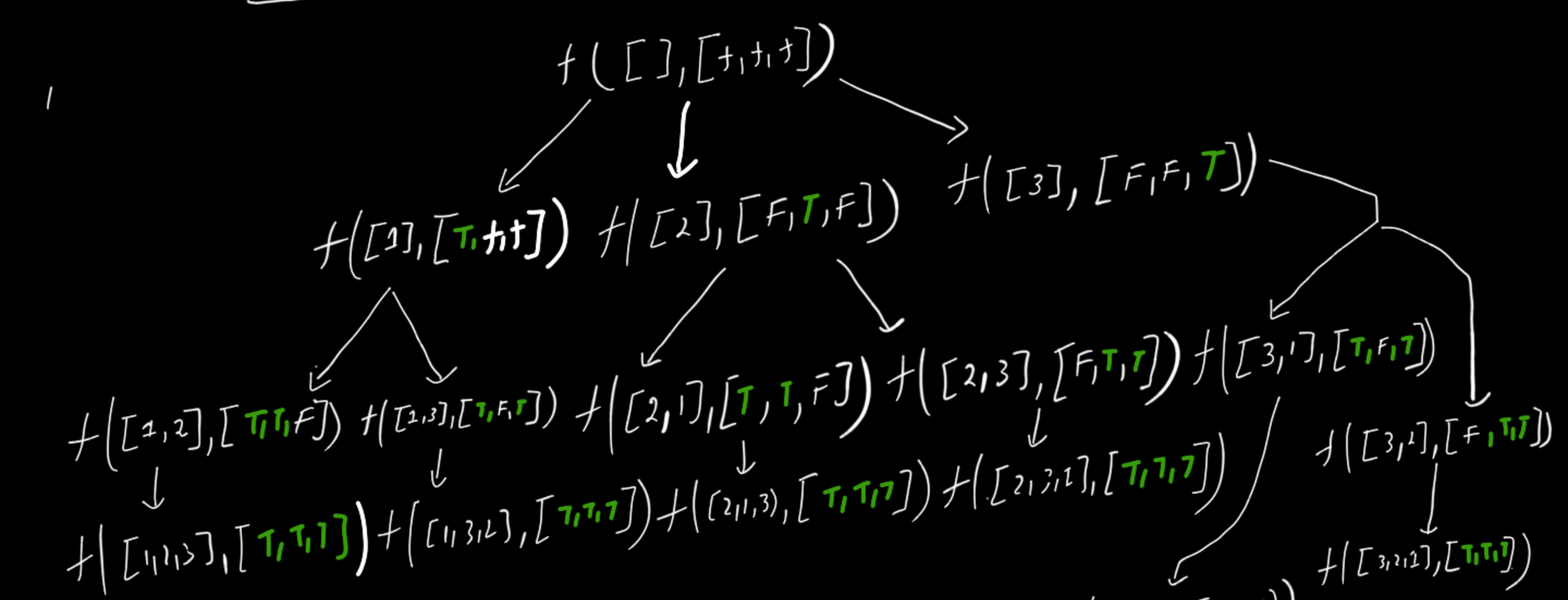
list → Visited array

○ Initially, we can take any element from the array

↓

n - choice

↳ array - length



Base - Case

① list become equal to array length

↳ we have one of our permutation.

PartIX.java

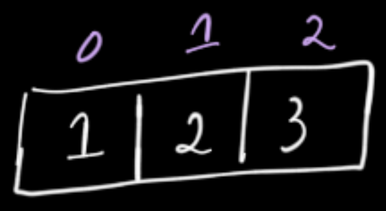
```java
public static void permute(int[] arr, boolean[] vis, List<Integer> curr, List<List<Integer>> ans) {
    if (arr.length == curr.size()) {
        ans.add(new ArrayList<>(curr));
        return;
    }

    for (int i = 0; i < arr.length; i++) {

        if (!vis[i]) {
            vis[i] = true;
            curr.add(arr[i]);
            permute(arr, vis, curr, ans);
            curr.remove( index: curr.size() - 1);
            vis[i] = false;
        }
    }

}
```

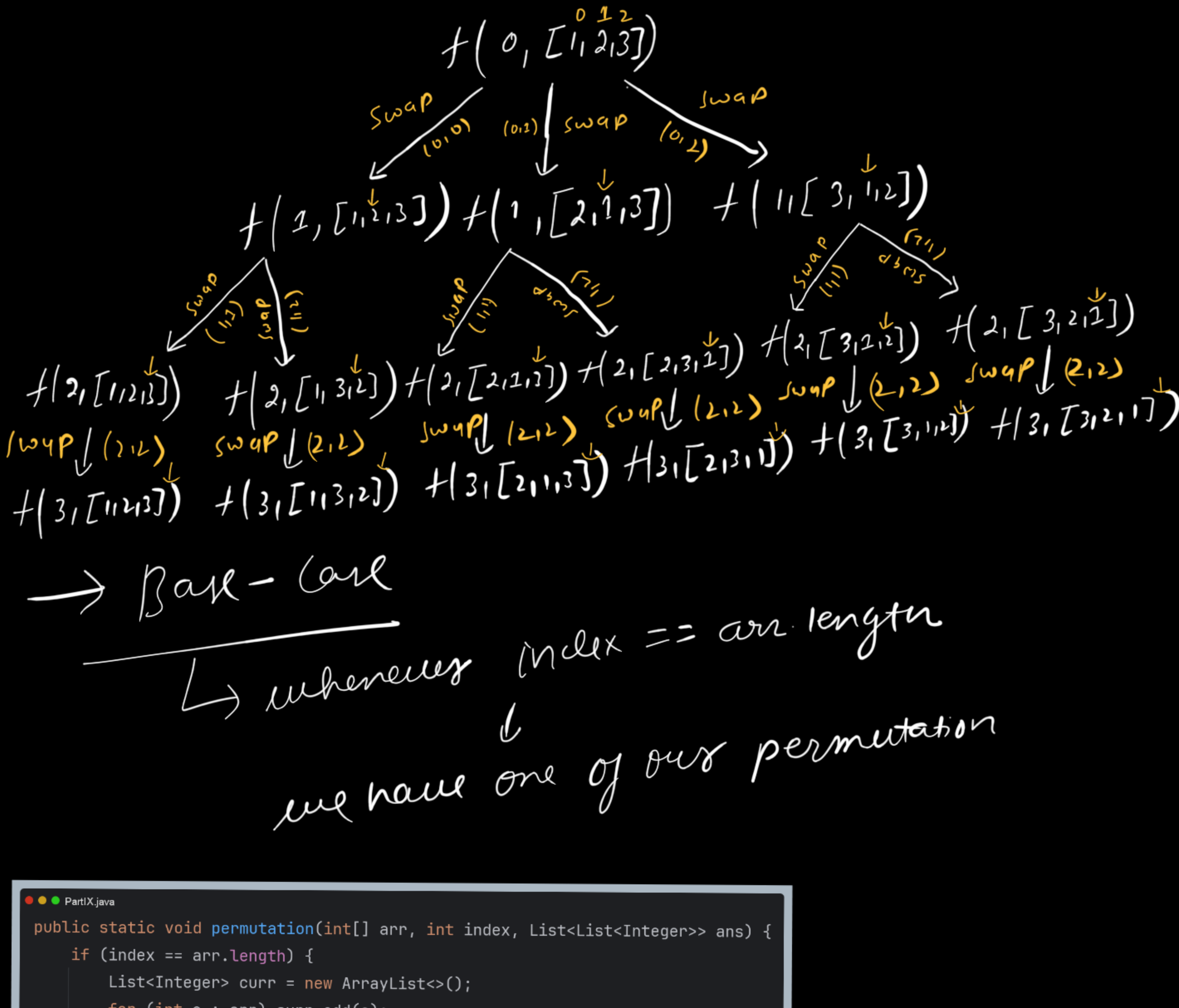
Tc: $O(n! * n)$

Sc: $O(n! * n)$

→ Approach - 2

↳ without extra visited array → swapping the element





→ Base - case

↳ whenever index == arr.length

we have one of our permutation

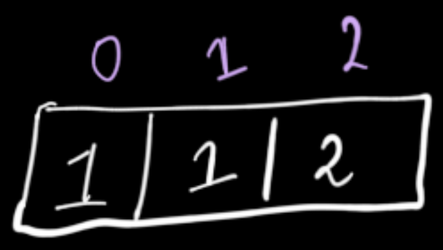
```java
PartIX.java
public static void permutation(int[] arr, int index, List<List<Integer>> ans) {
    if (index == arr.length) {
        List<Integer> curr = new ArrayList<>();
        for (int e : arr) curr.add(e);
        ans.add(curr);
        return;
    }

    for (int i = index; i < arr.length; i++) {
        swap(arr, i, index);
        permutation(arr, index: index + 1, ans);
        swap(arr, i, index);
    }

}

private static void swap(int[] arr, int f, int s) {  2 usages  new *
    int t = arr[f];
    arr[f] = arr[s];
    arr[s] = t;
}
```

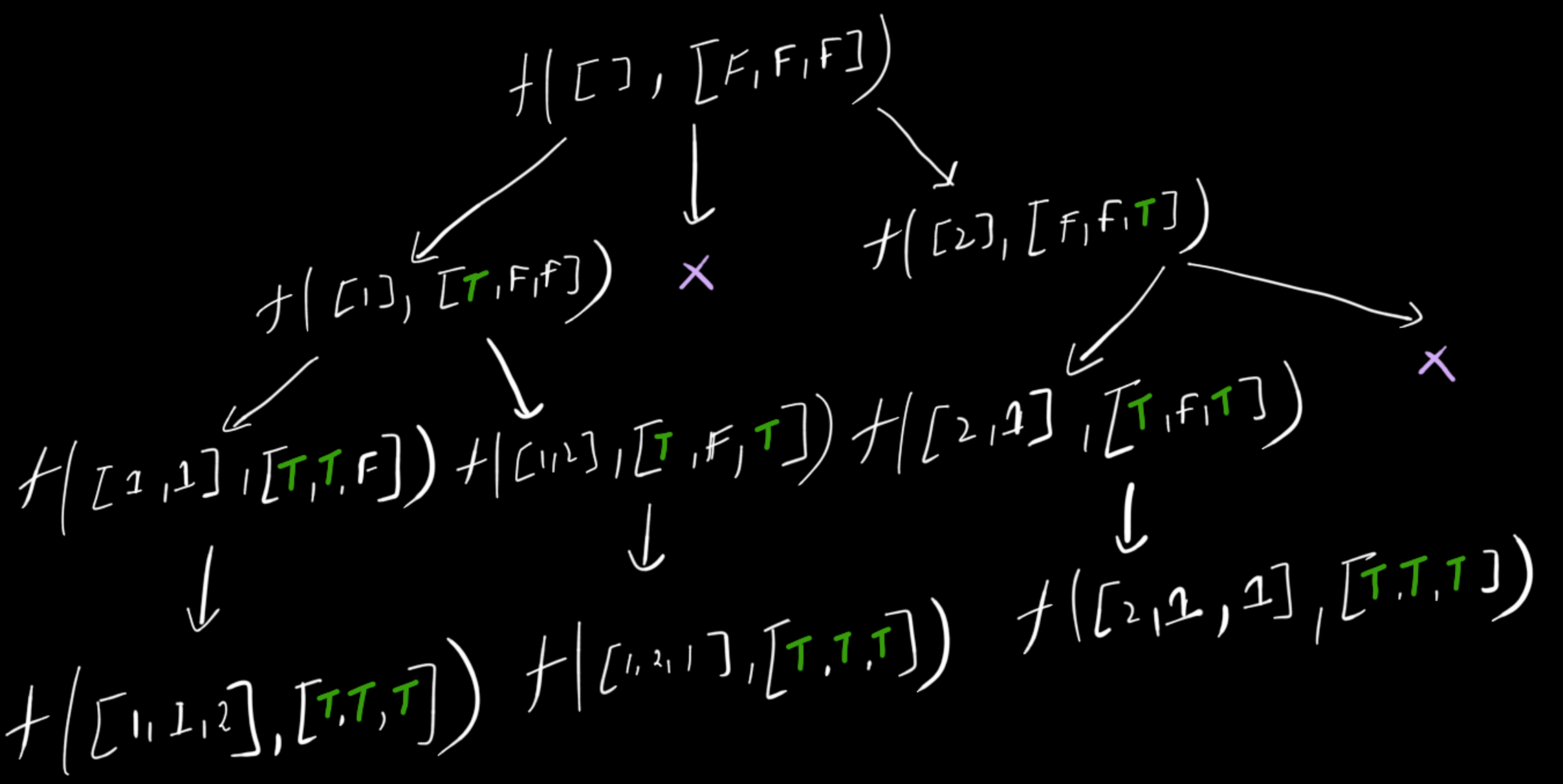
TC: O(n×n!)

SC: O(n×n!)

→ permutation II

↳ generate unique permutation, duplicates are not allowed





→ To skip an element

(1) We are checking if arr[i] == curr[i-1] ?

(2) Check if the previous same element hasn't been used in the current recursion path

!vis[i-1] ?

↳ if these both condition are satisfied, then we can safely skip the duplicate.

→ To check if adjacent element are similar or not we need to sort it before passing it to the f(x).

↓

O(n log n)

```java
PartX.java
public static void permuteUnique(int[] arr, boolean[] vis, List<Integer> curr, List<List<Integer>> ans) {
    if (arr.length == curr.size()) {
        ans.add(new ArrayList<>(curr));
        return;
    }

    for (int i = 0; i < arr.length; i++) {
        // remove duplicate
        if (i > 0 && arr[i] == arr[i - 1] && !vis[i - 1]) continue;

        if (!vis[i]) {
            vis[i] = true;
            curr.add(arr[i]);
            permuteUnique(arr, vis, curr, ans);
            curr.remove( index: curr.size() - 1);
            vis[i] = false;
        }
    }
```

→ duplicate handling

Tc: $O(n! * n)$

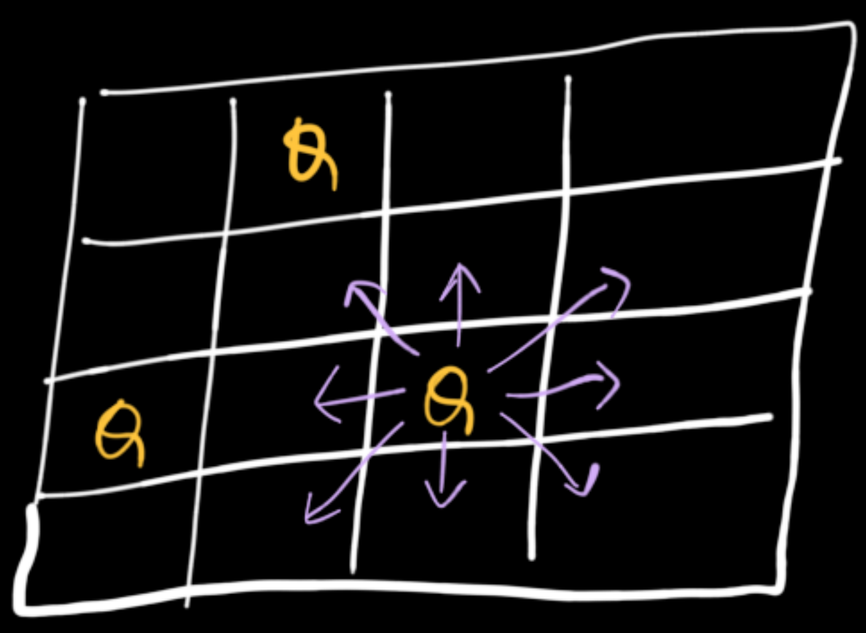
Sc: $O\left(\frac{n!}{t_1! \times t_2! \cdots t_n!}\right)$

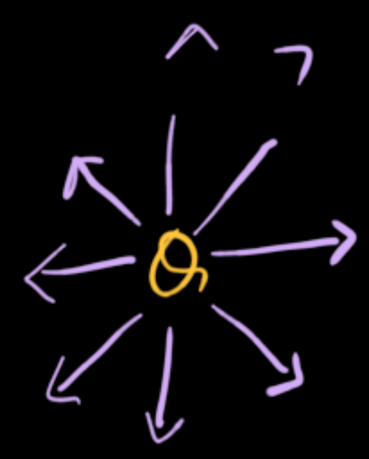
t_i → frequency of each element.

→ N-Queens

↳ place queens on n×n chessboard, such that their path do not intersect either diagonally / left, right, up or down

→ How do we check if placing a queen is safe?



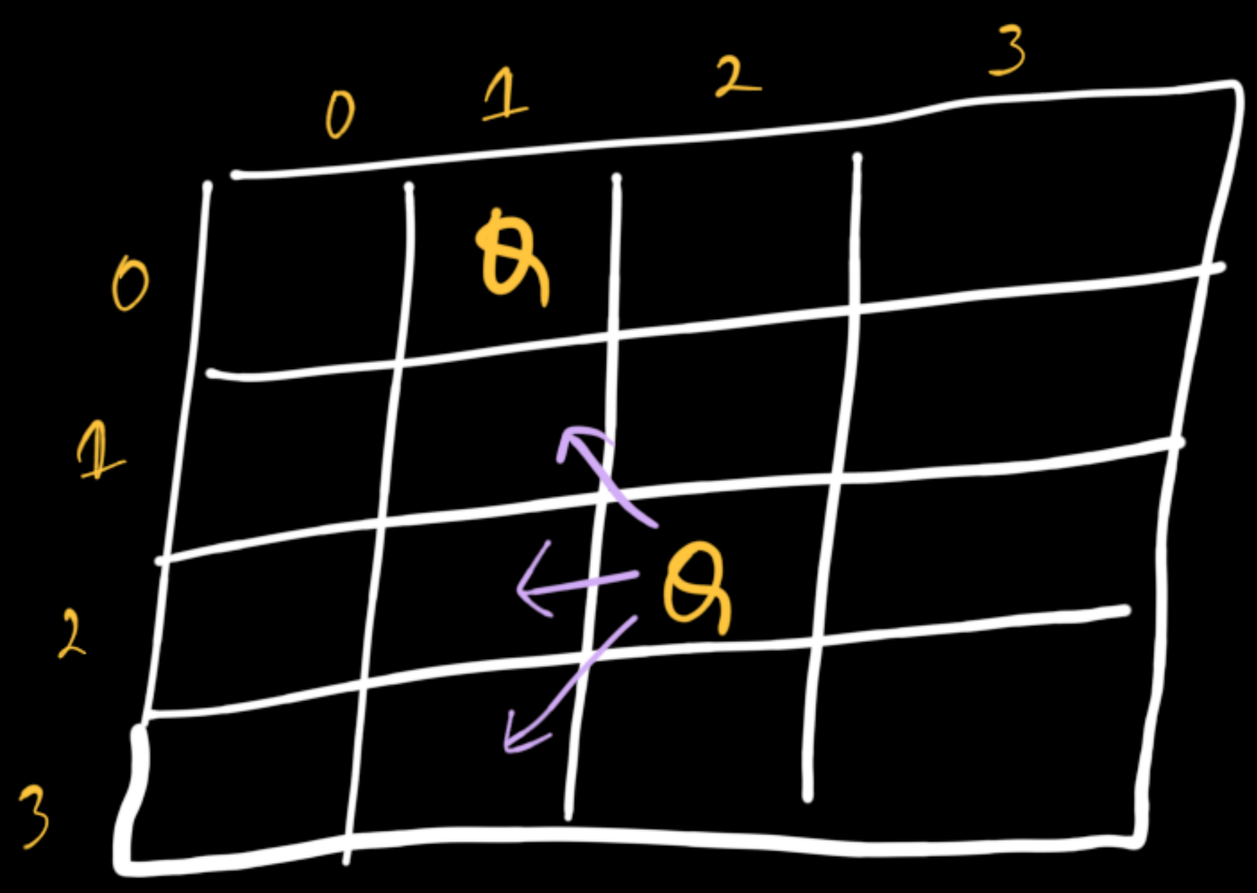


Queen can attack in 8 directions

- We already know that, there will only be one queen per column.
  ↳ i.e no need to check up & down

- We also know that, there is no need to check right, because we for sure know that any queen that threatns the current position must be in a previous (left) column.
  ↳ i.e we only need to check for

→ i) left
ii) upper-left diagonal
iii) lower-left diagonal



→ For upper-left diagonal

↳ Initially: Q(2,2)
↓ after moving
Q(1,1)

Both row & column is decreasing

→ For lower left diagonal

↳ Initially: Q(2,2)
↓ after moving
Q(3,1)

- row increases
- column decreases.

→ For left-side

↳ Initially: Q(2,2)
↓ after moving
Q(2,1)

Only column changes

Tc: $O(n! \times n)$

Sc: $O(n) + O(N^2) + O(S \times N^2)$

↙ stack space

↓ Board size

↓

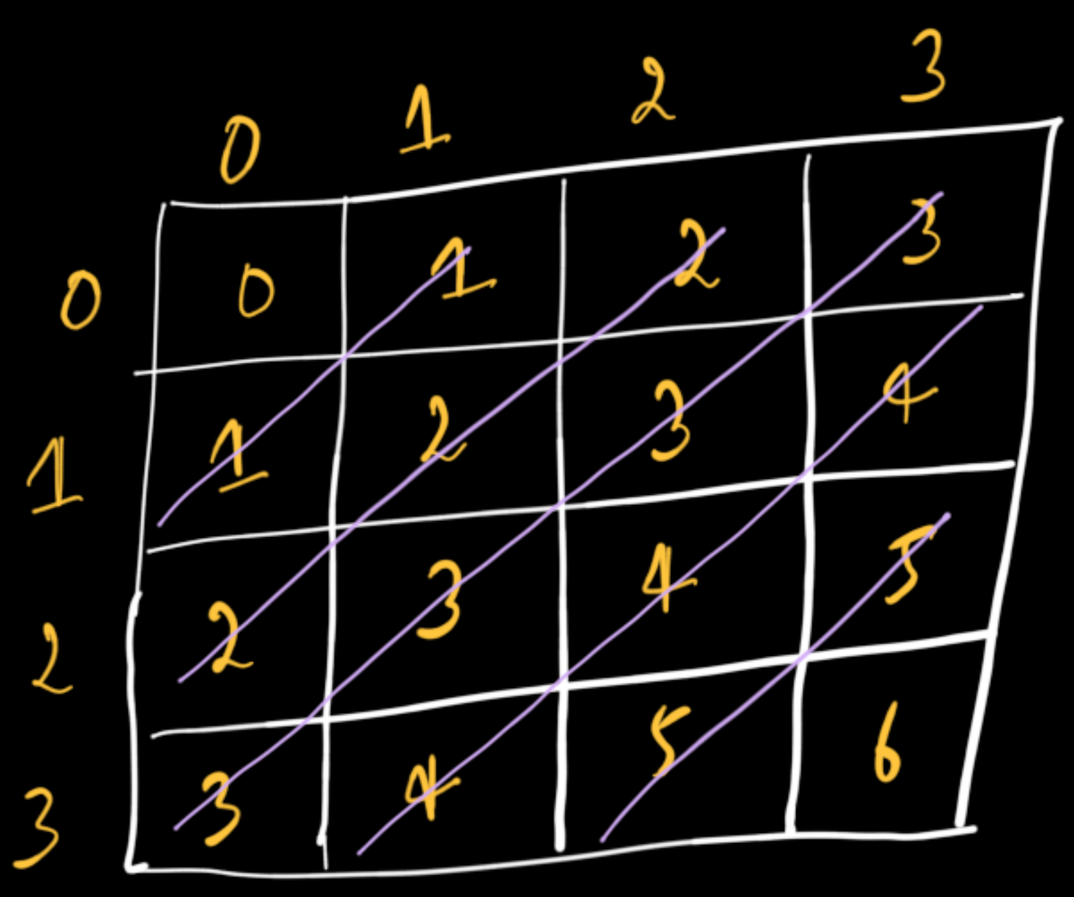
S: No of valid solution

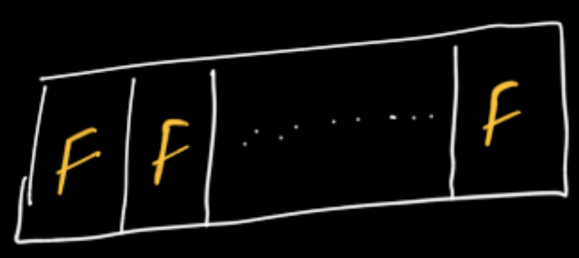
N^2: Each solution has n, each of length n

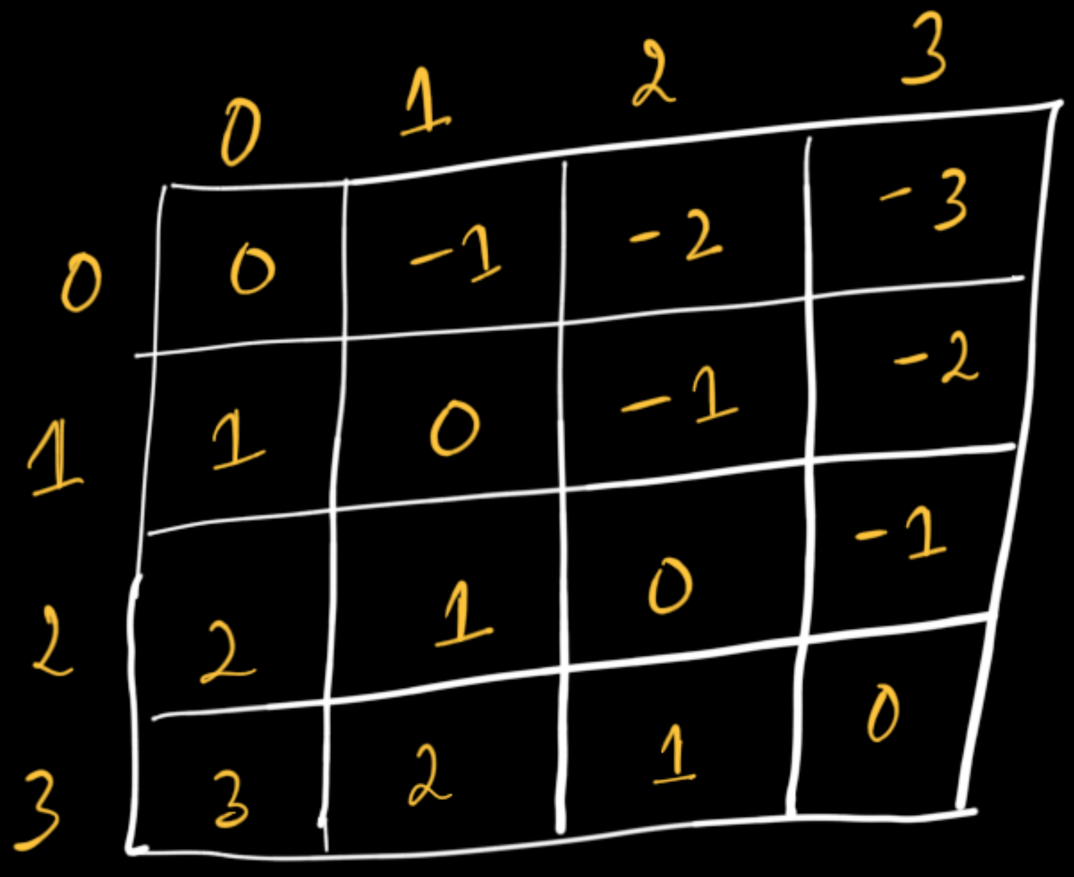
PartXI.java

```java
public static List<List<String>> solve(int n) {  1 usage  new *
    List<List<String>> ans = new ArrayList<>();
    Character[][] board = buildBoard(n);
    nQueen(board,  col: 0, ans);
    return ans;
}

private static void nQueen(Character[][] board, int col, List<List<String>> ans) {
    if (col == board[0].length) {
        ans.add(boardToStringList(board));
        return;
    }

    for (int row = 0; row < board.length; row++) {
        if (canWePlaceSafely(board, row, col)) {
            board[row][col] = 'Q';
            nQueen(board,  col: col + 1, ans);
            board[row][col] = '.';
        }
    }
}

private static boolean canWePlaceSafely(Character[][] board, int row, int col) {
    int dupRow = row;
    int dupCol = col;

    // check upper-left
    while (row >= 0 && col >= 0) {
        if (board[row][col] == 'Q') return false;
        row--;
        col--;
    }

    // check left
    row = dupRow;
    col = dupCol;
    while (col >= 0) {
        if (board[row][col] == 'Q') return false;
        col--;
    }

    // check lower-left
    col = dupCol;
    while (row < board.length && col >= 0) {
        if (board[row][col] == 'Q') return false;
        row++;
        col--;
    }

    return true;
}

private static Character[][] buildBoard(int n) {  1 usage  new *
    Character[][] board = new Character[n][n];
    for (int i = 0; i < n; i++) {
        board[i] = new Character[n];
        Arrays.fill(board[i],  val: '.');
    }
    return board;
}

private static List<String> boardToStringList(Character[][] board) {  1 usage  new *
    List<String> res = new ArrayList<>();
    for (Character[] row : board) {
        StringBuilder sb = new StringBuilder();
        for (Character cell : row) {
            sb.append(cell);
        }
        res.add(sb.toString());
    }
    return res;
}
```

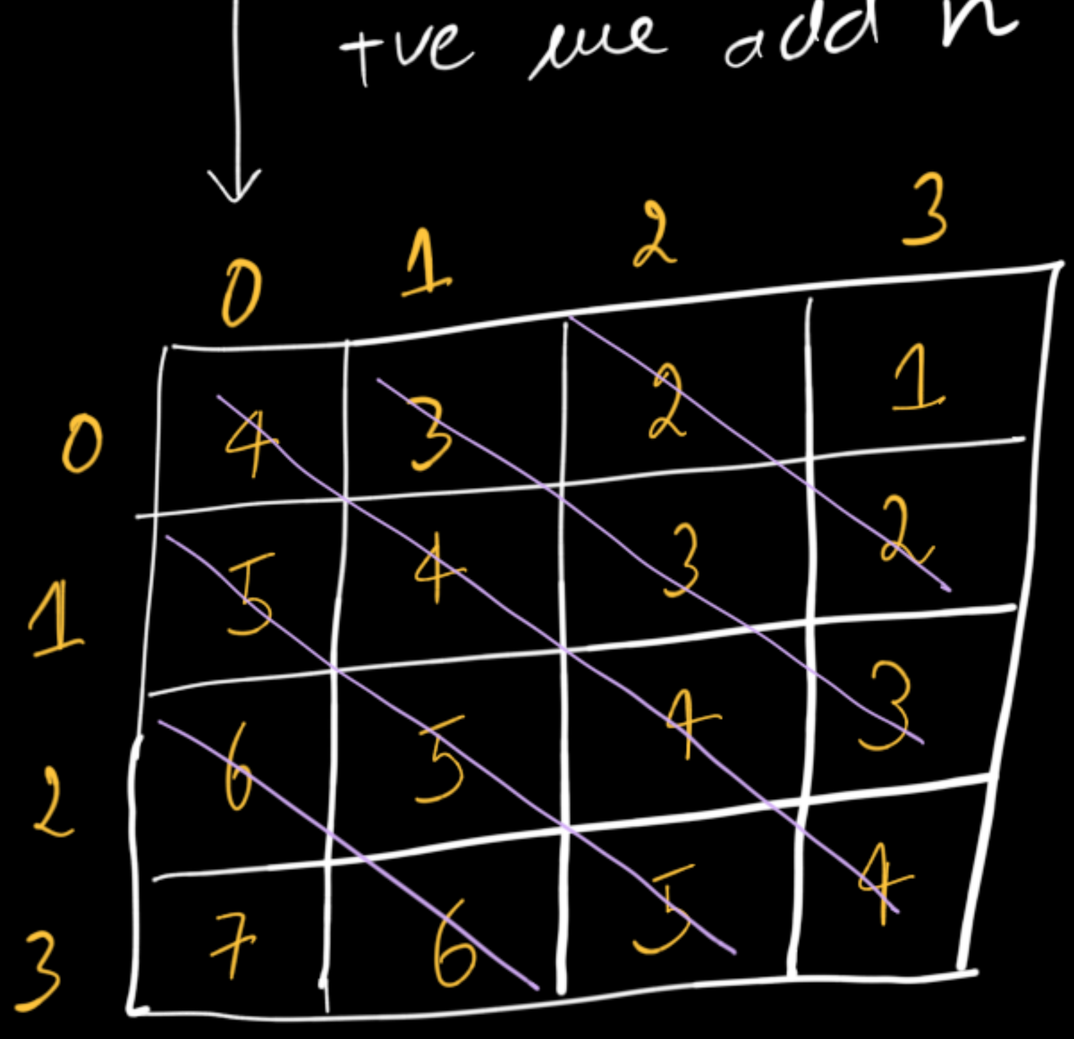
→ How can we optimize it?



boolean[] =  (2n) ←



we can't use negative index, so to make it +ve we add n



From Top-left to bottom right

① we notice that all diagonal have same sum of (row + col) from top-left to bottom right.

↳ How much space do we need to store all diagonal?

↳ Range of index

↳ min: (0,0)

max: (n-1) + (n-1) = 2n-2

- All unique diagonal is

↳ (max - min) + 1 = (2n-2-0) + 1

↳ **space = 2n-1**

→ From top-right to Bottom left diagonal

Range of Index

↳ min: -(n-1) + n = 1

max: (n-1) + n = 2n-1

space needs is

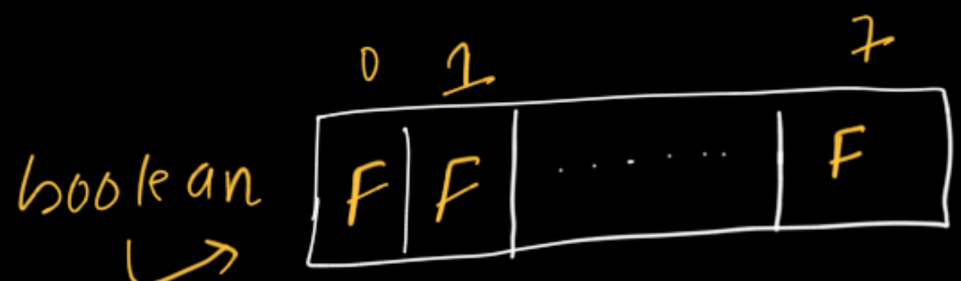
space = (max - min) + 1

= [(n-1) - [-(n-1)]] + 1

= [n-1 + n-1] + 1

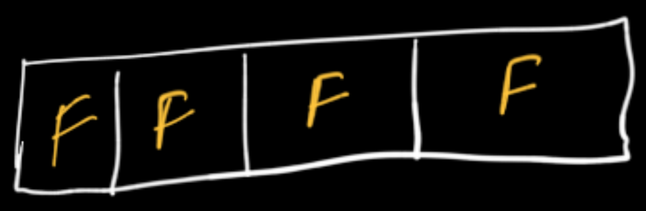
= [2n-2] + 1

= 2n-1

boolean ↳ 

→ How do we check for rows ?

↳ create boolean array of size rows [board.length].

boolean[] = 

PartXI.java

```java
public static List<List<String>> solve(int n) {
    List<List<String>> ans = new ArrayList<>();
    Character[][] board = buildBoard(n);
    boolean[] r = new boolean[board.length];
    boolean[] tlbr = new boolean[2 * n];
    boolean[] trbl = new boolean[2 * n];
    nQueenOpt(board, col: 0, r, tlbr, trbl, ans);
    return ans;
}
```

boolean[] r = for checking rows

boolean[] tlbr = for checking top-left to bottom right diagonal.

boolean[] trbl = for checking top-right to bottom left diagonal.

PartXI.java

```java
public static void nQueenOpt(Character[][] board, int col, boolean[] r, boolean[] tlbr, boolean[] trbl, List<List<String>> ans) {

    if (board.length == col) {
        ans.add(boardToStringList(board));
        return;
    }

    for (int row = 0; row < board.length; row++) {
        int topLeftBR = row + col;
        int topRightBL = row - col + board.length;

        if (r[row] || tlbr[topLeftBR] || trbl[topRightBL]) continue;

        board[row][col] = 'Q';
        r[row] = true;
        tlbr[topLeftBR] = true;
        trbl[topRightBL] = true;

        nQueenOpt(board, col: col + 1, r, tlbr, trbl, ans);
        board[row][col] = '.';
        r[row] = false;
        tlbr[topLeftBR] = false;
        trbl[topRightBL] = false;
    }

}
```

→ while visiting mark true.

→ while backtracking mark false.

TC: O(n!)

SC: O(n) + O(2n) + O(2n) + O(n) + O(5 x N2)

(n) → static space; (2n) → tlbr; + (2n) → trbl; (n) → row; (5 x N2) → ans list ← No. of solution.

→ Sudoku - solver

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

Rule:

1) Each of the digit 1-9 must occur exactly once in each row & column
2) Each of the digit 1-9 must occur exactly once in each of the 9 → 3×3 sub-boxes of the grid.

↓ solution

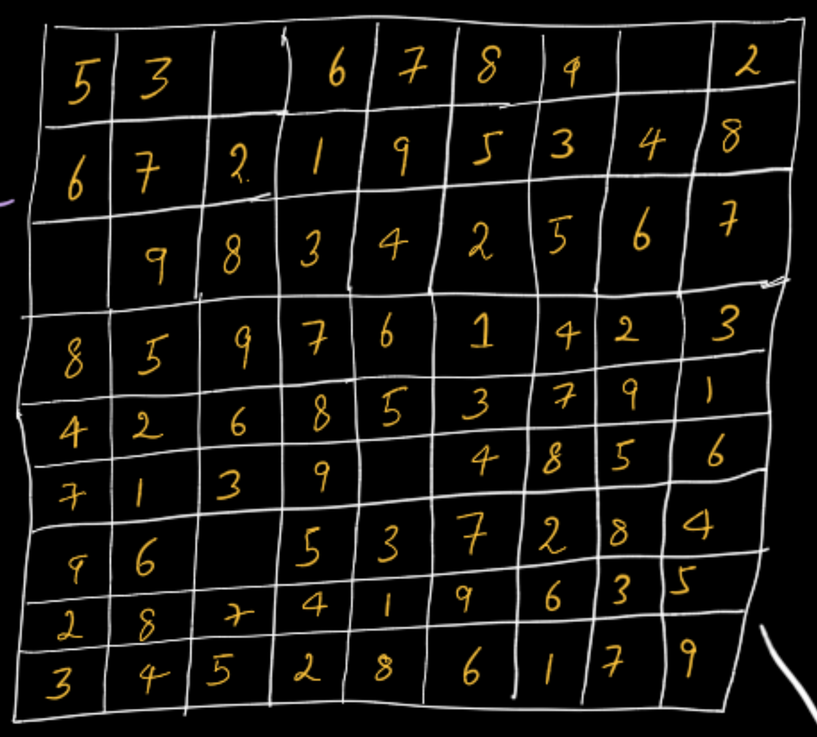
5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

↳ Each of the row & column have unique numbers

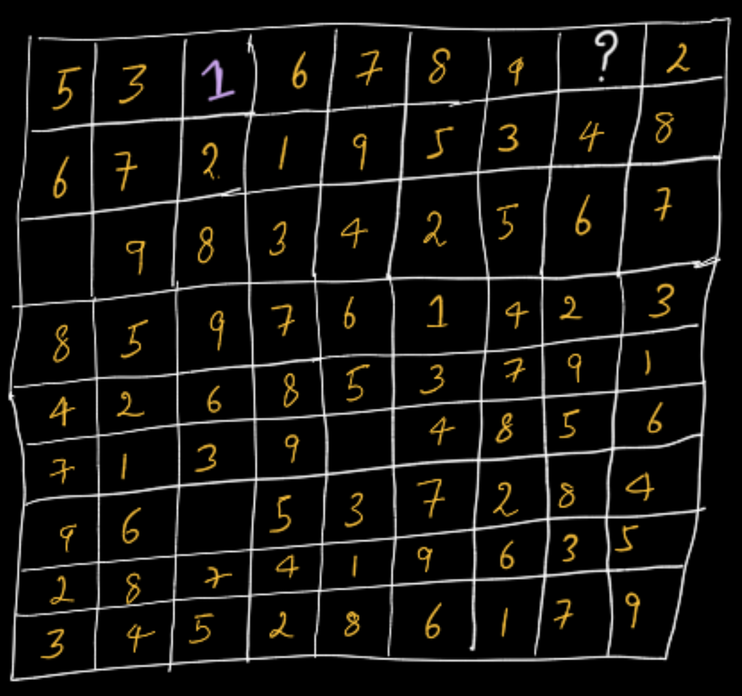
- Each of the 3×3 sub-boxes have unique numbers.

- Since we need to try all possible values of placing 1-9 digit safely. we can use recursion to try out all possible combination.

① Can we fill up a number 1-9 which is safe to place.



false



↓

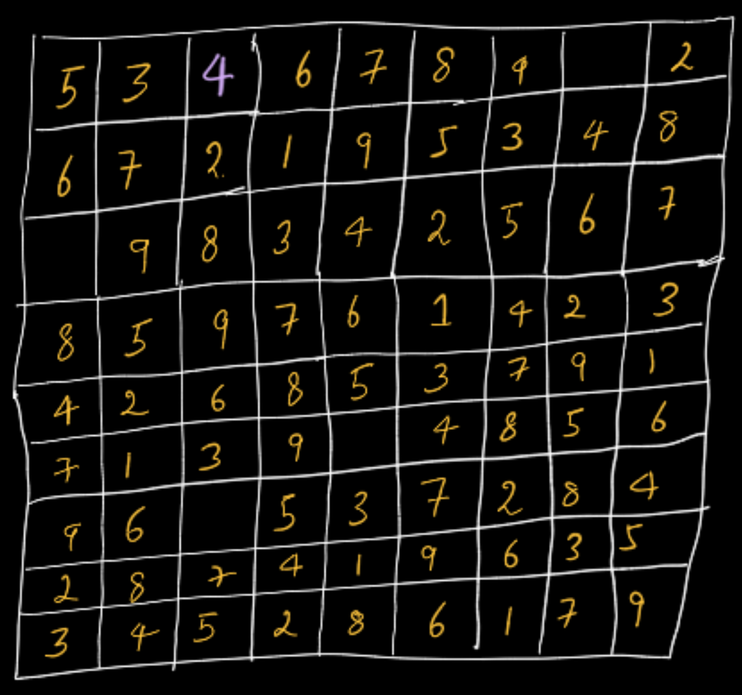
Can we place 1-9 at ? ×

↓

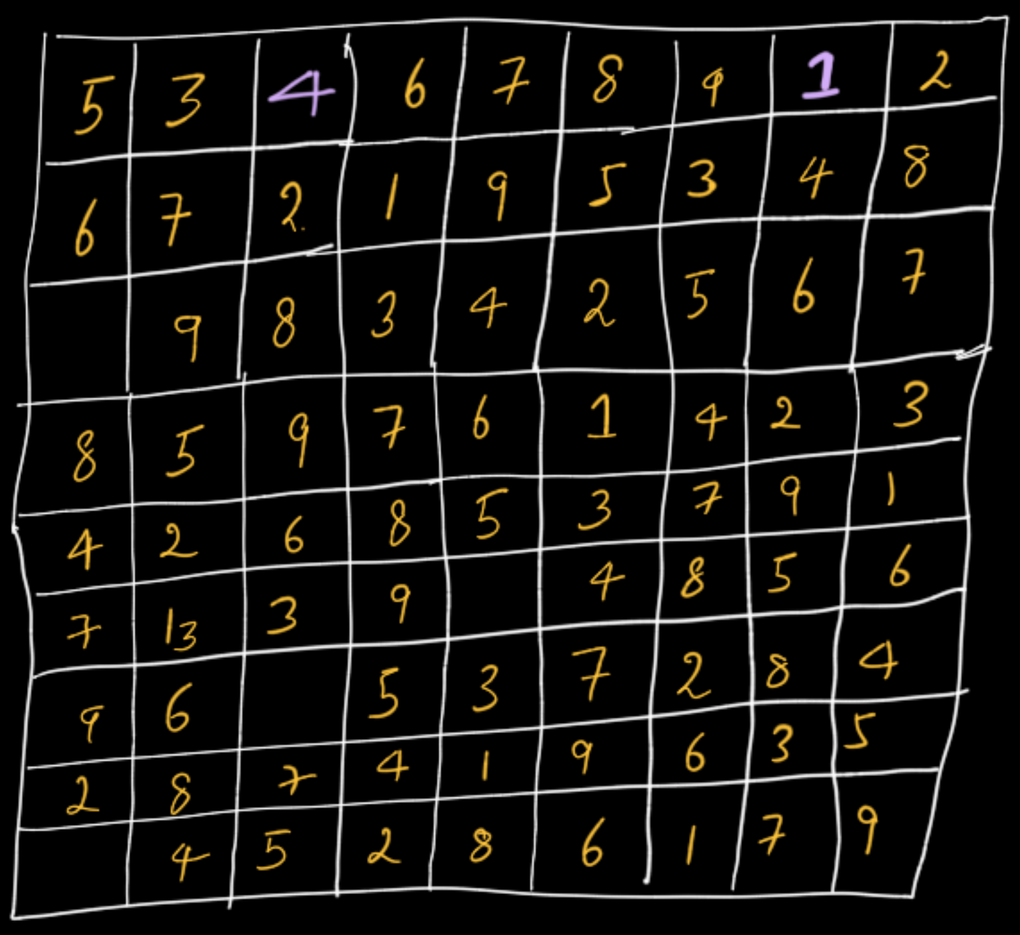
return false

↓

This recursion call will not have our answer

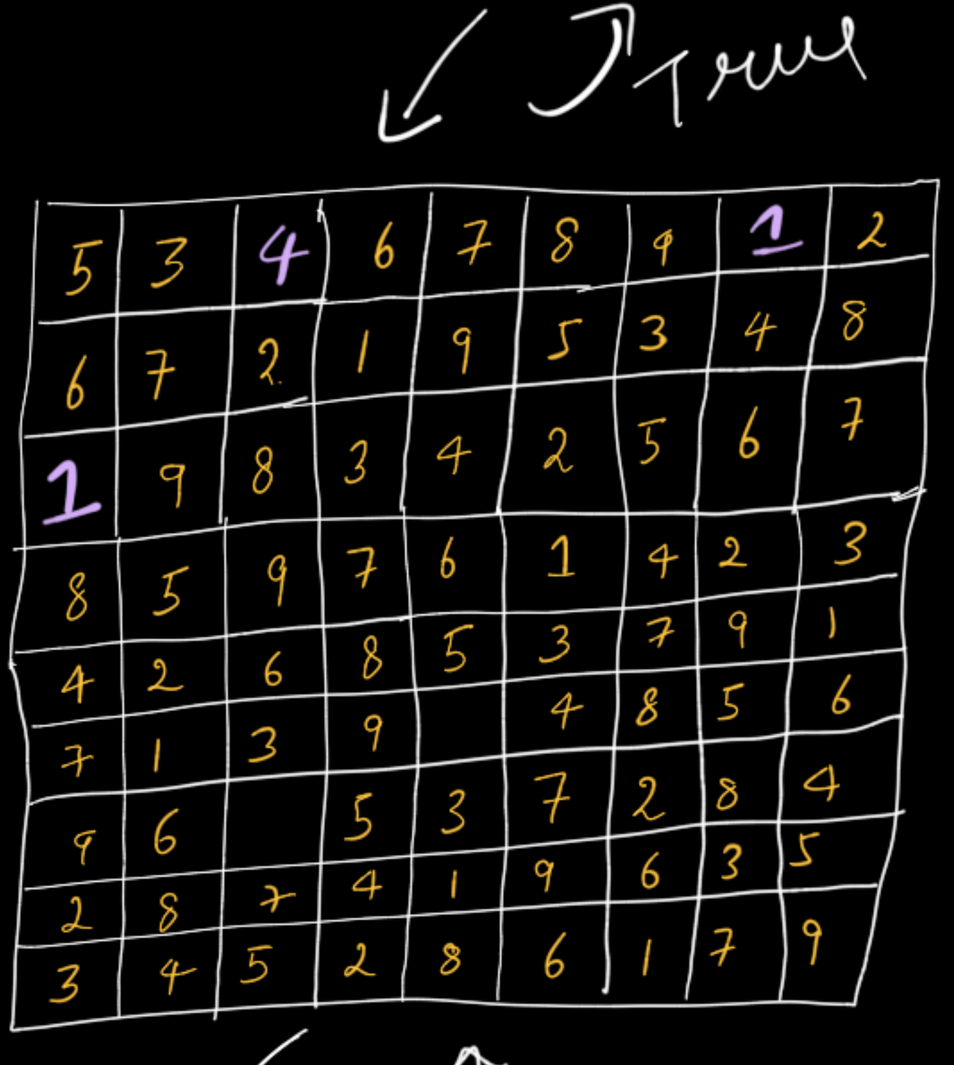


true



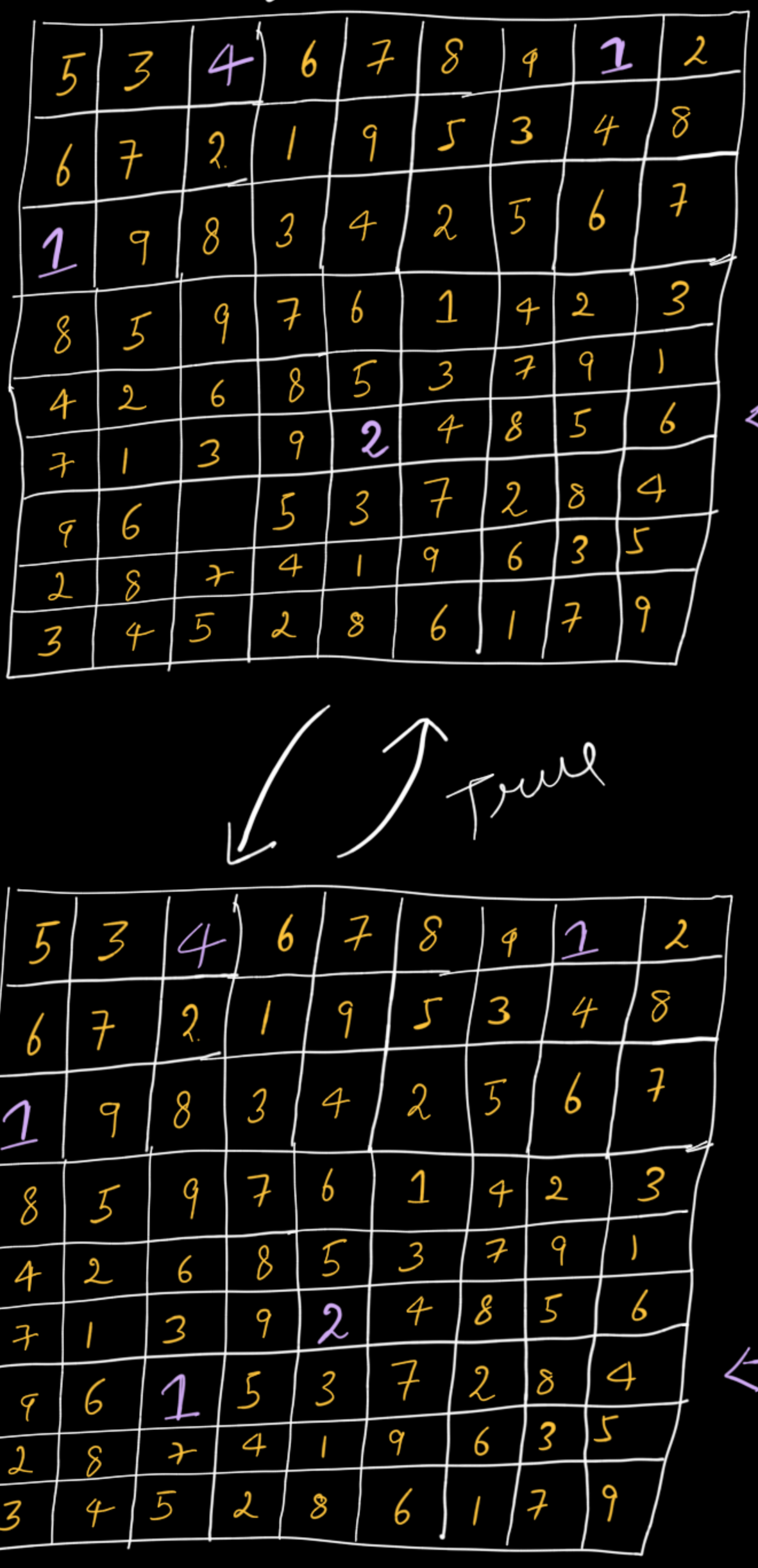
Try 1-9

true



Try 1-9

true



→ How do we check if putting a new value is safe or not?

→ For rows & column, just iterate from index 0 - 8 & check

board[row][i] == num || board[i][col] == num

→ For 3×3 grid

	0	1	2	3	4	5	6	7	8
0	5	3	4	6	7	8	9	1	2
1	6	7	2	1	9	5	3	4	8
2	1	9	8	3	4	2	5	6	7
3	8	5	9	7	6	1	4	2	3
4	4	2	6	8	5	3	7	9	1
5	7	1	3	9	2	4	8	5	6
6	9	6	1	5	3	7	2	8	4
7	2	8	7	4	1	9	6	3	5
8	3	4	5	2	8	6	1	7	9

Let's suppose we are filling a safe number from 1-9 at index (6, 2) ?

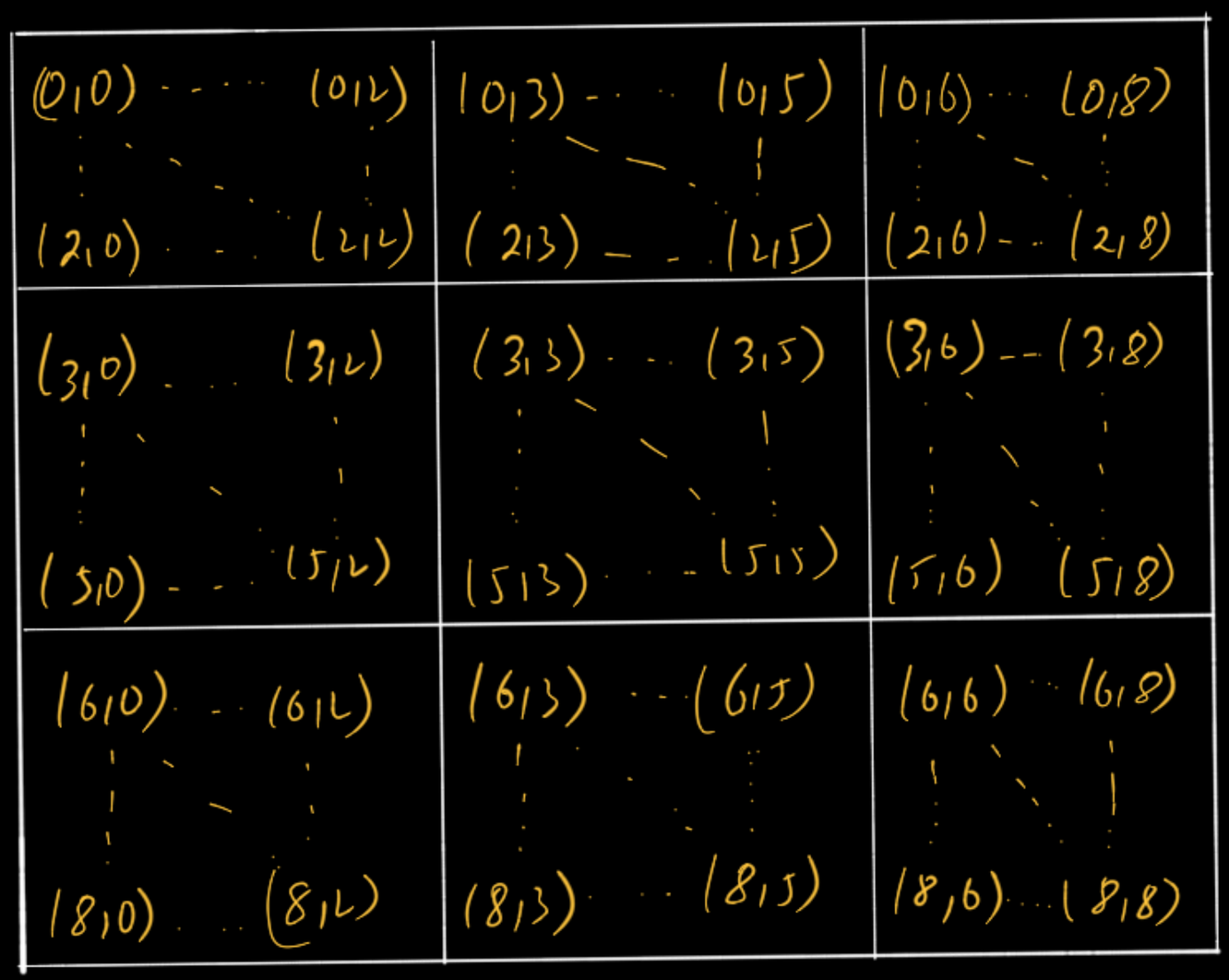
↓

How do we check the 3×3 grid?

↳ How do we know which to check or from which (row, col) index?

For (6, 2) index?

↳ we need to know from where to start our 3×3 grid search?



Blocks of 3?

$(r, c) \rightarrow (6, 2)$

$S_R = r/3$

$S_C = c/3$

$(S_R, S_C) \Rightarrow \left(\frac{r}{3}, \frac{c}{3}\right) = \left(\frac{6}{3}, \frac{2}{3}\right) = (2, 0)$

No if we multiply it by 3, we give get our start row & start column

$3(S_R, S_C) = 3(2, 0)$

$(S_R, S_C) = (6, 0)$

→ $S_R = 6$, $S_C = 0$

(6,0)	(6,1)	(6,2)
(7,0)	(7,1)	(7,2)
(8,0)	(8,1)	(8,2)

	0	1	2
0	9	6	1
1	2	8	7
2	3	4	5

→ How do we access each element? from (6,0) to (8,2)?

↳ Flatten the 3×3 grid in 1D

[9, 6, 1, 2, 8, 7, 3, 4, 5]
0 1 2 3 4 5 6 7 8

1-D → 2D index

$i \rightarrow (i/3, i\%3)$

0 → $(0/3, 0\%3) = (0,0)$

1 → $(1/3, 1\%3) = (0,1)$

2 → $(2/3, 2\%3) = (0,2)$

3 → $(3/3, 3/3) = (1,0)$

→ So, to scan in single for loop?

$sr = 3 * r/3 + i/3;$

$sc = 3 * c/3 + i\%3;$

PartXII.java

```java
public static boolean sudoku(char[][] board) {  3 usages  new *

    for (int row = 0; row < board.length; row++) {
        for (int col = 0; col < board[0].length; col++) {
            if (board[row][col] == '.') {

                for (char num = '1'; num <= '9'; num++) {
                    if (isValid(board, row, col, num)) {
                        board[row][col] = num;
                        if (sudoku(board)) return true;
                        board[row][col] = '.';
                    }
                }

                return false;
            }
        }
    }
    return true;

}

private static boolean isValid(char[][] board, int row, int col, char num) {

    for (int i = 0; i < 9; i++) {
        if (board[i][col] == num || board[row][i] == num) return false;

        int sr = 3 * (row / 3) + i / 3;
        int sc = 3 * (col / 3) + i % 3;

        if (board[sr][sc] == num) return false;
    }
    return true;
}
```

Tc: $O(9^{81})$

Sc: $O(n)$ → Static space

↳ for rows & columns

→ For 3×3 grid.

→ Find Unique Binary string

↳ Given an array of strings nums containing n-unique binary string each of length n, return a binary string of length n that does not appear in nums. Find any one.

nums = | "111" | "011" | "001" |

↳ if we generate all binary string of length 3 → {100, 000, 010, 110, 101}
↳ any one can be our answer.

set = [nums]

f("", 3, set)

f("0", 3, set) f("1", 3, set)

f("00", 3, set) f("01", 3, set) f("10", 3, set) f("11", 3, set)

f("000", 3, set) f("001", 3, set) f("100", 3, set) f("101", 3, set)

Base-case

① if length == 3, ss the generated answer is not in set,
↓
our answer.

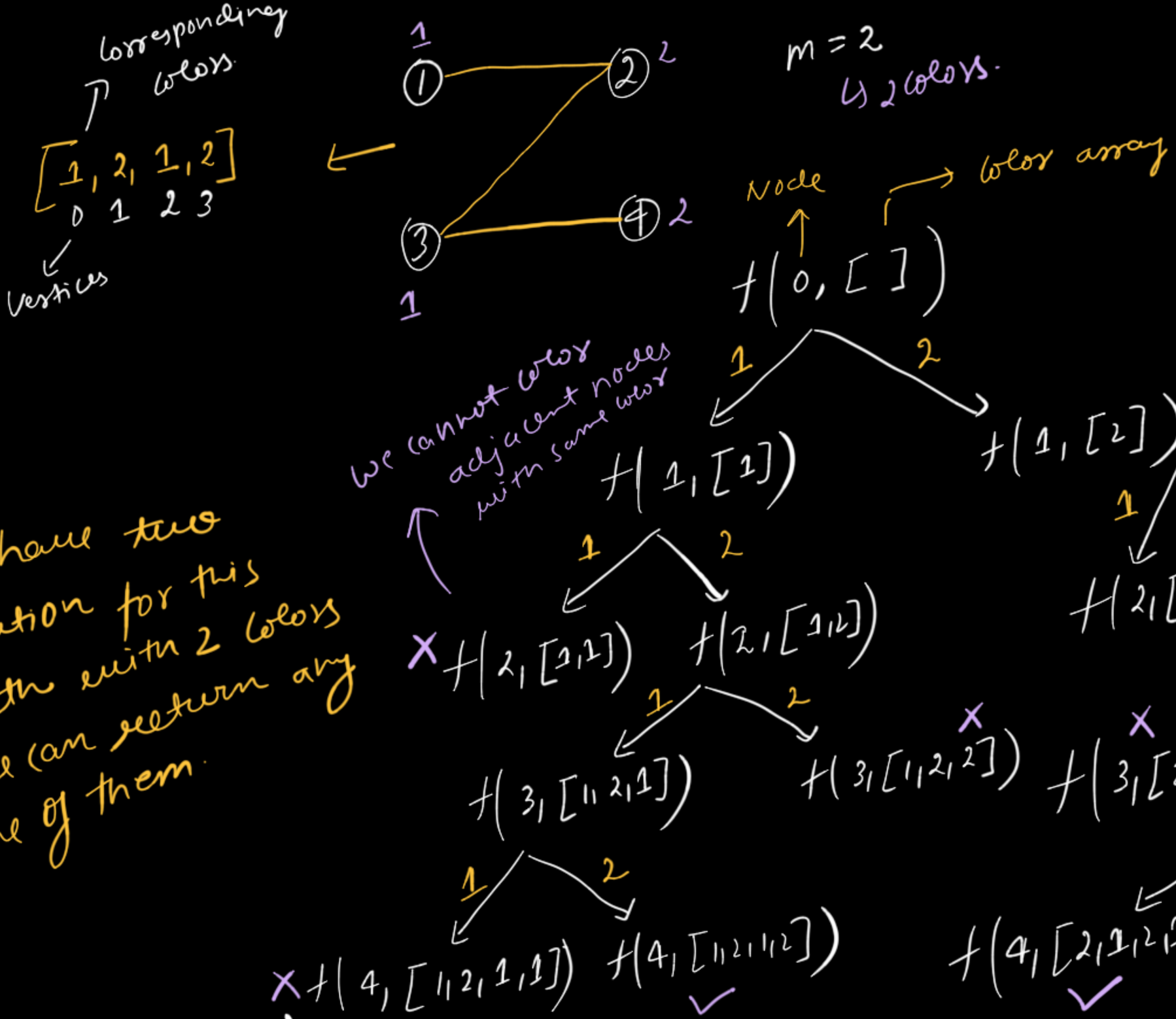
Tc: $O(2^n)$
Sc: $O(N)$

PartXIII.java

```java
public static String findDifferentBinaryString(String[] nums) {  1 usag
    Set<String> set = new HashSet<>();
    Collections.addAll(set, nums);
    return uniq( ans: "", nums.length, set);

}

private static String uniq(String ans, int length, Set<String> set) {
    if (ans.length() == length) {
        if (set.contains(ans)) {
            return null;
        }
        return ans;
    }
    // zero try
    String zt = uniq( ans: ans + '0', length, set);
    if (zt != null) return zt;

    // one try
    return uniq( ans: ans + '1', length, set);

}
```

→ M-Coloring Problem

↳ Given an undirected graph with N vertices & an Integer M, determine if its possible to color the graph using at most M colors such that no two adjacent vertices share the same color.



Tc: $O(N^M)$

Sc: $O(N) + O(N \times N)$

↓ for stack space

↓ for building graph of vertices n.

→ **Base-Case**

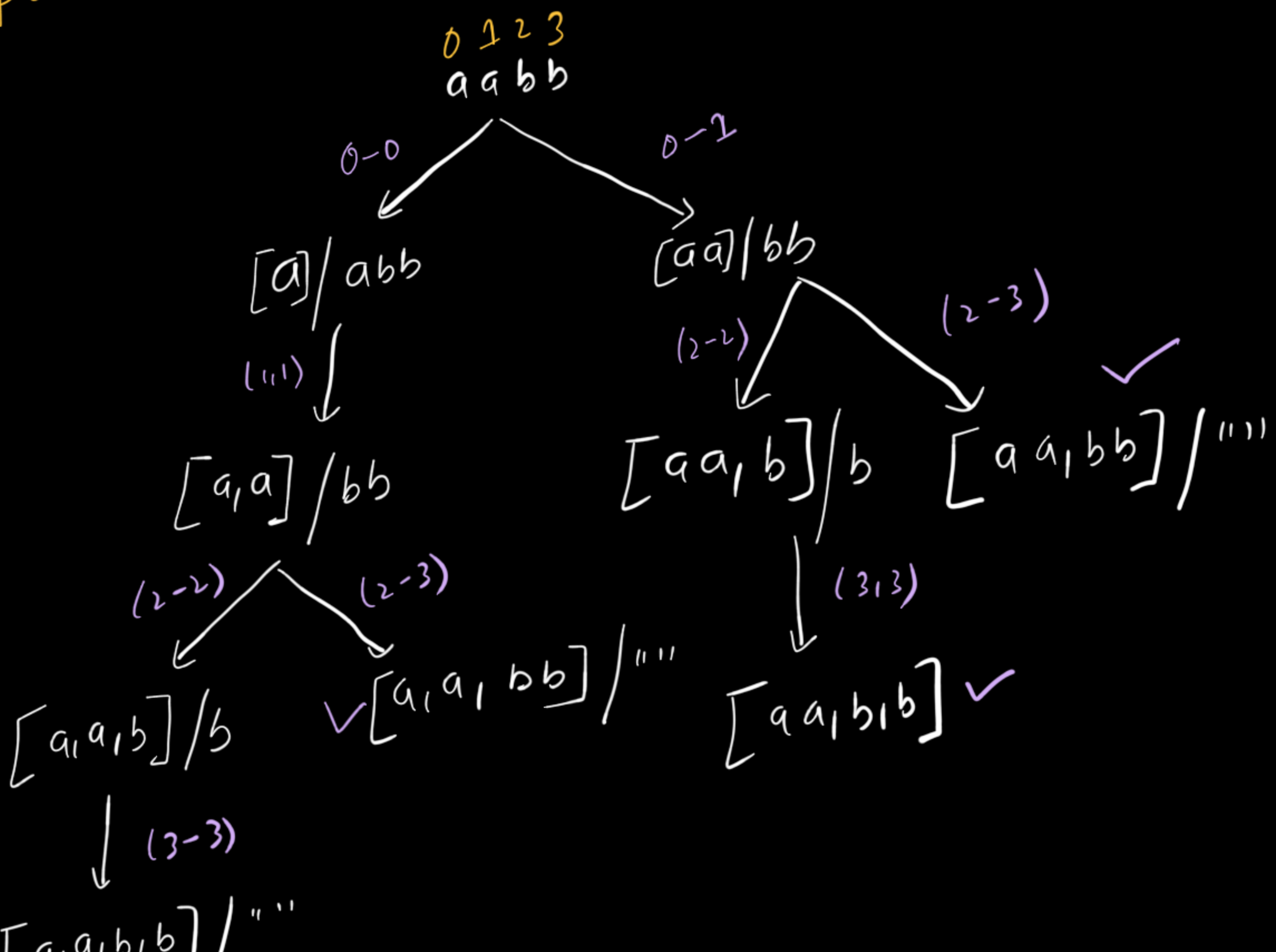
↳ when node == no. of vertices, we have one our solution.

★ Same question as Leetcode Q. 1042

```java
PartXIV.java
public static boolean solve(int vertices, int m, int[][] path) {  1 usage  new *
    boolean[][] graph = buildGraph(vertices, path);
    int[] color = new int[vertices];

    boolean result = mColoring( node: 0, graph, color, m, vertices);

    System.out.println("Coloring: " + Arrays.toString(color));
    return result;
}

private static boolean mColoring(int node, boolean[][] graph, int[] color, int m, int vertices) {
    if (node == vertices) return true; // base case: all nodes colored

    for (int c = 1; c <= m; c++) {
        if (isSafe(node, graph, color, c)) {
            color[node] = c;
            if (mColoring( node: node + 1, graph, color, m, vertices)) return true;
            color[node] = 0; // backtrack
        }
    }

    return false;
}

private static boolean isSafe(int node, boolean[][] graph, int[] color, int c) {  1 usage  new *
    for (int i = 0; i < graph.length; i++) {
        if (graph[node][i] && color[i] == c) return false;
    }
    return true;
}

private static boolean[][] buildGraph(int n, int[][] edges) {  1 usage  new *
    boolean[][] graph = new boolean[n][n];
    for (int[] e : edges) {
        int u = e[0] - 1;
        int v = e[1] - 1;
        graph[u][v] = true;
        graph[v][u] = true;
    }
    return graph;
}
```

→ Palindrome Partitioning

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



T.C: $O(N * 2^N)$

S.C: $O(N) + O(P)$

Stack space

Storing no. of partition

```java
PartXV.java
public static void partition(String s, int index, List<String> curr, List<List<String>> ans) {
    if (index == s.length()) {
        ans.add(new ArrayList<>(curr));
        return;
    }

    for (int i = index; i < s.length(); i++) {
        if (isPalindrome(s, index, i)) {
            curr.add(s.substring(index, i + 1));
            partition(s, index: i + 1, curr, ans);
            curr.remove( index: curr.size() - 1);
        }
    }

}

private static boolean isPalindrome(String str, int f, int s) {  1 usage  new *
    while (f <= s) {
        if (str.charAt(f) != str.charAt(s)) {
            return false;
        }
        f++;
        s--;
    }
    return true;
}
```

O(N) → for copying the arraylist.

→ Rat in a Maze problem

↳ Given a m×n grid, rat start at (0,0) & want to reach the bottom-right cell (n-1, n-1). The rat can move in four direction $\{U, D, L, R\}$. Find all possible path.

- True: path is free to go
- False: path is blocked

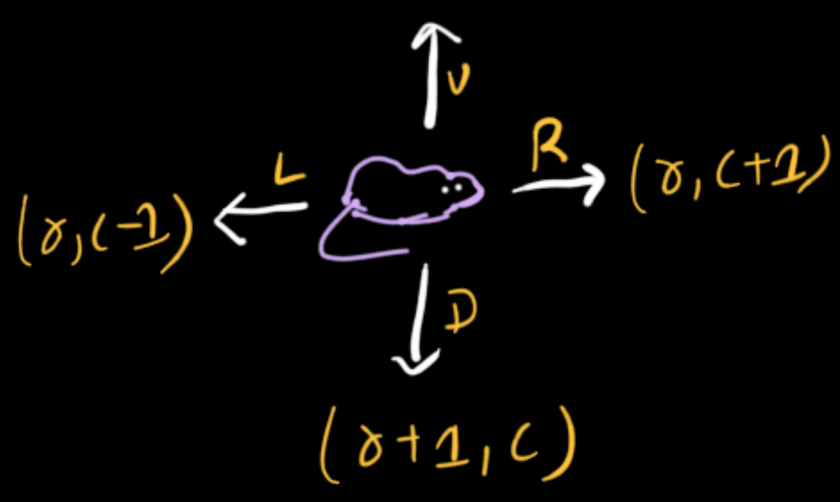
- A visited cell cannot be visited twice.

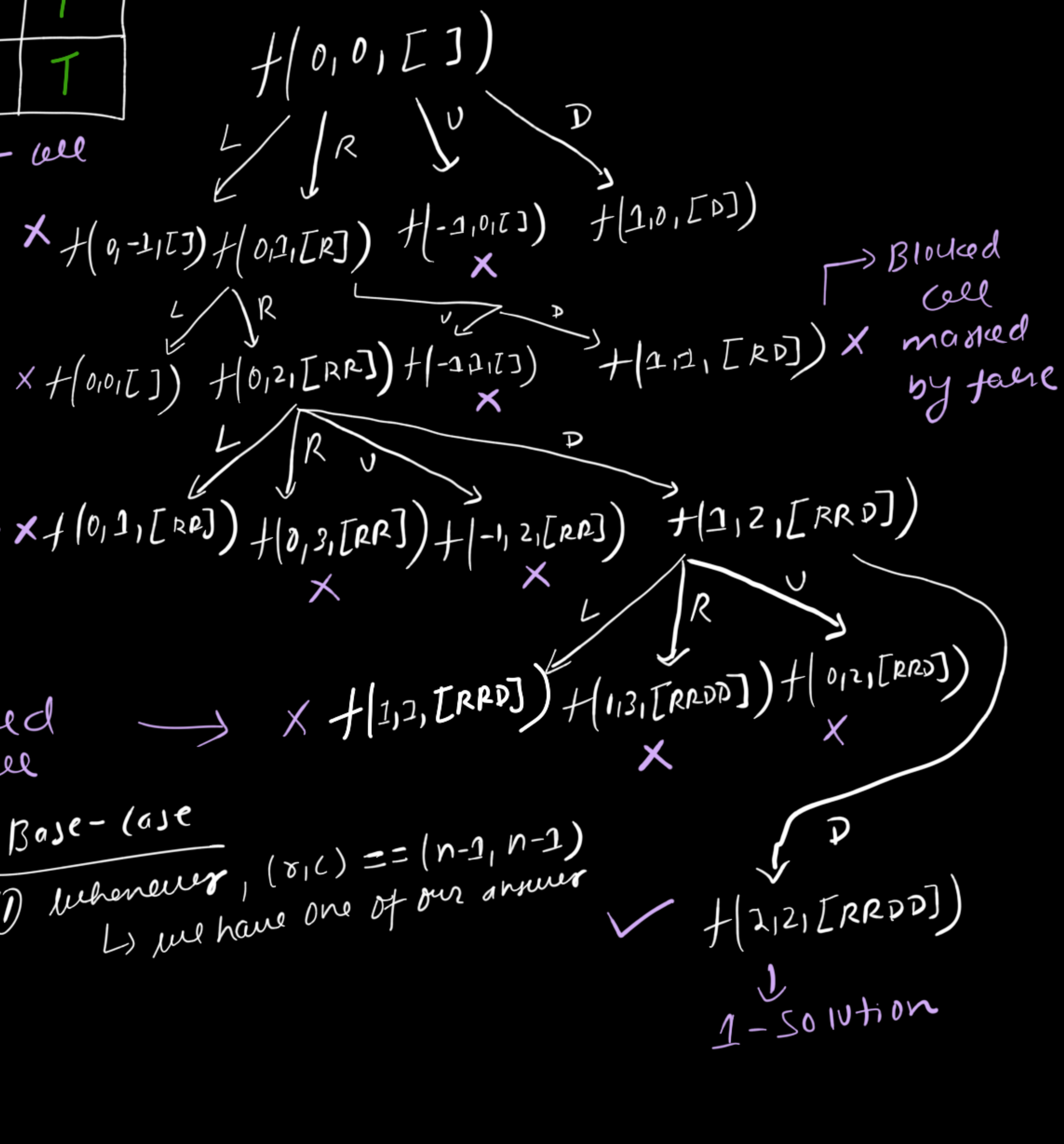
- Initially rat is at (0,0)

	0	1	2
0	T	T	T
1	F	F	T
2	F	F	T

Visited - cell

	0	1	2
0	🐀	T	T
1	T	F	T
2	T	T	T





→ Base-case

① whenever, $(r, c) == (n-1, n-1)$
↳ we have one of our answer

PartXVI.java

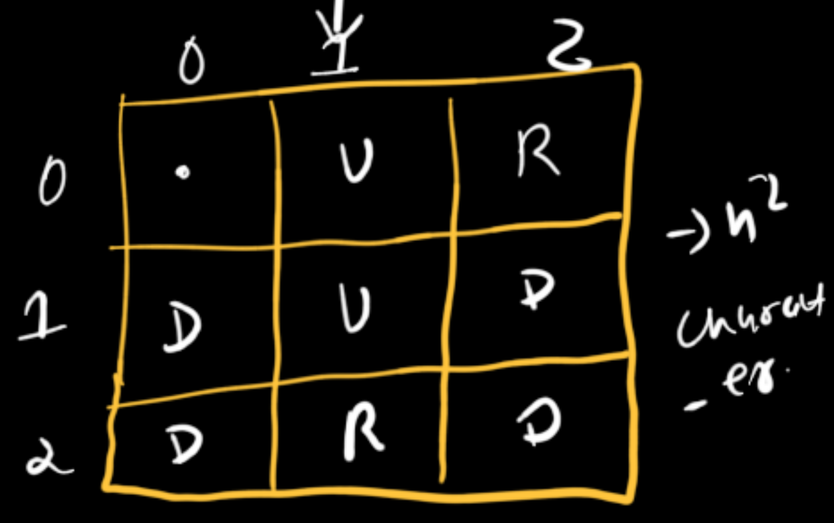
```java
public static List<String> ratMaze() {  1 usage  new *
    List<String> ans = new ArrayList<>();
    boolean[][] maze = new boolean[][]{
            {true, true, true},
            {true, false, true},
            {false, true, true}
    };

    int n = maze.length;
    boolean[][] visited = new boolean[n][n];
    ratMaze( row: 0,  col: 0,  path: "", maze, visited, ans);
    return ans;
}

private static void ratMaze(int row, int col, String path, boolean[][] maze, boolean[][] visited, List<String> ans) {
    int n = maze.length;

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

    // Boundary, wall or visited check
    if (row < 0 || col < 0 || row >= n || col >= n) return;
    if (!maze[row][col] || visited[row][col]) return;

    visited[row][col] = true;

    // Try all directions - {L,R,U,D}
    ratMaze(row,  col: col - 1,  path: path + 'L', maze, visited, ans); // left
    ratMaze(row,  col: col + 1,  path: path + 'R', maze, visited, ans); // right
    ratMaze( row: row - 1, col,  path: path + 'U', maze, visited, ans); // up
    ratMaze( row: row + 1, col,  path: path + 'D', maze, visited, ans); // down

    visited[row][col] = false; // backtrack
}
```

$TC: O(4^{n^2})$ → 4 direction to explore, a rat can visit every cell, there are n^2 cell

$SC: O(n^2 + k \cdot n^2)$

- Recursion stack space. → at each recursive call we explore n^2 cell in worst case
- Number of valid path
- in worst case, each of the valid path can consist of n^2 cell

For Example



→ K^{th} - permutation sequence

↳ Given n & K, return the K^{th} permutation of the sequence $[1,2,3,\ldots,n]$.

Approach - 1

For example:

↳ $n=3,\ K=3$

$[123, 213, 321, 132, 231, 312]$

↓ sort

$[123, 132, 213, 231, 312, 321]$

1 2 3 4 5 6

$k^{th}\_perm = 213$

TC: $O(n! \times n) + (n \log n)$

SC: $O(n! \times n)$

PartXVII.java

```java
public static String getPermutation(int n, int k) {  1 usage  new *
    List<String> ans = new ArrayList<>();
    StringBuilder sb = new StringBuilder();
    for (int i = 1; i <= n; i++) sb.append(i);

    permute(sb.toString().toCharArray(), index: 0, ans);
    return ans.get(k - 1);
}

private static void permute(char[] arr, int index, List<String> ans) {
    if (index == arr.length) {
        ans.add(new String(arr));
        return;
    }

    for (int i = index; i < arr.length; i++) {
        swap(arr, i, index);
        permute(arr, index: index + 1, ans);
        swap(arr, i, index); // backtrack
    }
}

private static void swap(char[] arr, int i, int j) {  2 usages  new *
    char temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

↳ Sort the list then access $(k-1)$

→ Approach - 2

↳ n = 4, K = 17

↳ For n = 4, the set contain [1, 2, 3, 4] → 4! = 24 perm

arr =

0	1	2	3
1	2	3	4

① Initially, we can make these permutation:

↳ For K = 17,

1 + {2,3,4} 3! = 6 (0-5)

2 + {1,3,4} 3! = 6 (6-11)

3 + {1,2,4} 3! = 6 (12-17)

4 + {1,2,3} 3! = 6 (18-23)

——

24

K = K - 1 = 16

our answer will lie in this range

↳ • We definetely know, 3 - - - will be part of our permutation

• To take 3 in our answer, we can do :-

↳ arr[K/6] = arr[16/6] = arr[2] = 3

• out of range (12-17) we need 16th seq -vence, so out of 6 permutation in this range we need,

↳ K = K % 6 = 16 % 6 = 4

② Now, we have selected one numbers (3) from our permutation, remaining number are :-

{1,2,4} → 1 + {2,4} 2! = 2 (0-1)

2 + {1,4} 2! = 2 (1-2)

4 + {1,4} 2! = 2 (2-3)

——

4

again we do same process → K = 4

• arr[K/2] = arr[4/2] = arr[2] = 4

• K = K - 1 - 2 = 0

3 4 - -

③ Now, we have selected two numbers (3,4) from our permutation, remaining numbers are :-

$\{1, 2\}$ (indices 0, 1) → $1 + \{2\}$ $1! = 1 = (0-0)$

↳ For $k=0$ → $2 + \{1\}$ $1! = 1 = (1-1)$

2

3 4 1 _

- $arr[k/1] = arr[0/1] = arr[0] = 1$
- $k = 0 \% 1 = 0$

④ We have selected four numbers (3,4,1) from our permutation, remaining numbers are :-

$\{2\}$ → This will be our final number.

3 4 1 2 → 16th perm

TC: O(n×n)

SC: O(n)

PartXVII.java

```java
public static String getPermutationOpt(int n, int k) {
    List<Integer> list = new ArrayList<>();  // O(N)
    int fact = 1;
    for (int i = 1; i < n; i++) {
        fact = fact * i;
        list.add(i);
    }
    list.add(n);
    StringBuilder ans = new StringBuilder();  // O(N)
    k = k - 1;

    while (true) {  // O(n)
        ans.append(list.get(k / fact));
        list.remove( index: k / fact);  // O(n)

        if (list.isEmpty()) break;

        k = k % fact;
        fact = fact / list.size();
    }
    return ans.toString();
}
```