• Recursion & Backtracking •

Principle of Mathematical Induction (PMI)

$$\sum_{1}^{n} i = \frac{n \times (n+1)}{2}$$

① prove $T(n)$

② Assume $T(n-1)$ as true

$$\sum_{1}^{n-1} i = \frac{(n-1) \times n}{2}$$

③ Corner cases (trivial cases)

$$T(1) = \frac{1 \times (1+1)}{2} = \frac{1 \times 2}{2} = 1$$

$$T(0) = \frac{0 \times (0+1)}{2} = 0$$

$$T(n) = 1+2+\cdots n$$
$$= (1+2+\cdots n-1)+n$$
$$= \frac{n \times (n-1)}{2} + n$$
$$= \frac{n(n+1)}{2}$$

H.P

$\forall n$ valid

iterative statements (loops)

- for
- while

recursive functions

function calling itself in a finite no of times

```
fun(){
    fun();
}
```

→ Infinite recursion

Print Decreasing

Given integer N, print all natural nos from N to 1.

eg) N = 5

5 | 4 3 2 1

N-1 = 4

4 3 2 1

Loop ⇒

Time ⇒ O(n)

Space ⇒ O(1)

High Level

(1) Expectation (input/output)

parameters, return type

```
public static void printNos(int n)
```

(2) Faith (recursive call on smaller input)

↳ printNos(n-1); [4,3,2,1]

(3) remaining work

↳ Syso(n); (5)

(4) base case ↝ if(n==0) return;

```
void printNos(int N) {
    // base case
0.  if(N == 0) return;

    // remaining work (preorder)
1.  System.out.print(N + " ");→O(1)

    // faith: recursive call
2.  printNos(N - 1);→O(1)
} return;
```

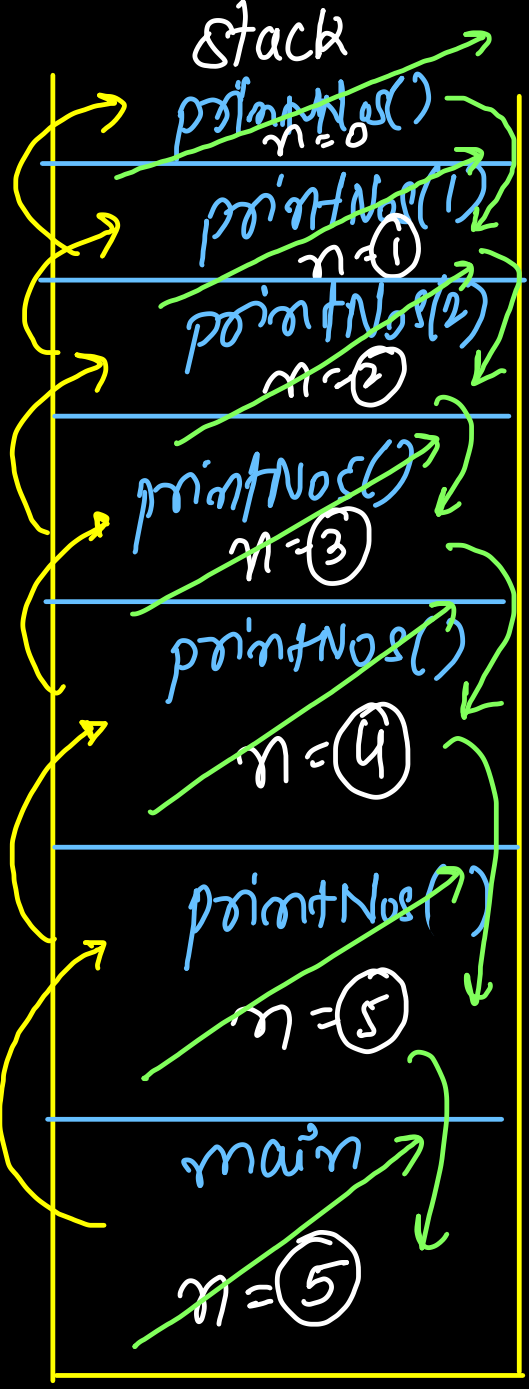
p.s.v. main(){

1. int n=5;
2. printNos(n);

}

(Read recursion)

Low-level



Output console

5 4 3 2 1

lot of [overhead]

Time ↓

O(n)

Space ↓

~~Input~~ ~~Output~~

Extra ✓

Recursion call stack ✓

O(n)

```java
void printNos(int N) {
    // remaining work
    System.out.println(N);

    // faith: recursive call
    printNos(N - 1);
}
```

due to infinite recursion
↳ runtime error (RT)
or
Stack overflow exception

⇓
Correctn (base case)
⇓

```java
void printNos(int N) {
    // base case
    if(N == 0) return;

    // remaining work
    System.out.print(N + " ");

    // faith: recursive call
    printNos(N - 1);
}
```

② Print Increasing

"Given integer n, print all nos from 1 to n w/o loop."

eg n = 5

output: 1 2 3 4 5

n-1 = 4

output: 1 2 3 4

① expectation
p.s. void printNos(int n)

② faith (recursive call)
↳ printNos(n-1);

③ remaining work
↳ Syso(n);

④ base case
↳ if(n==0) return;

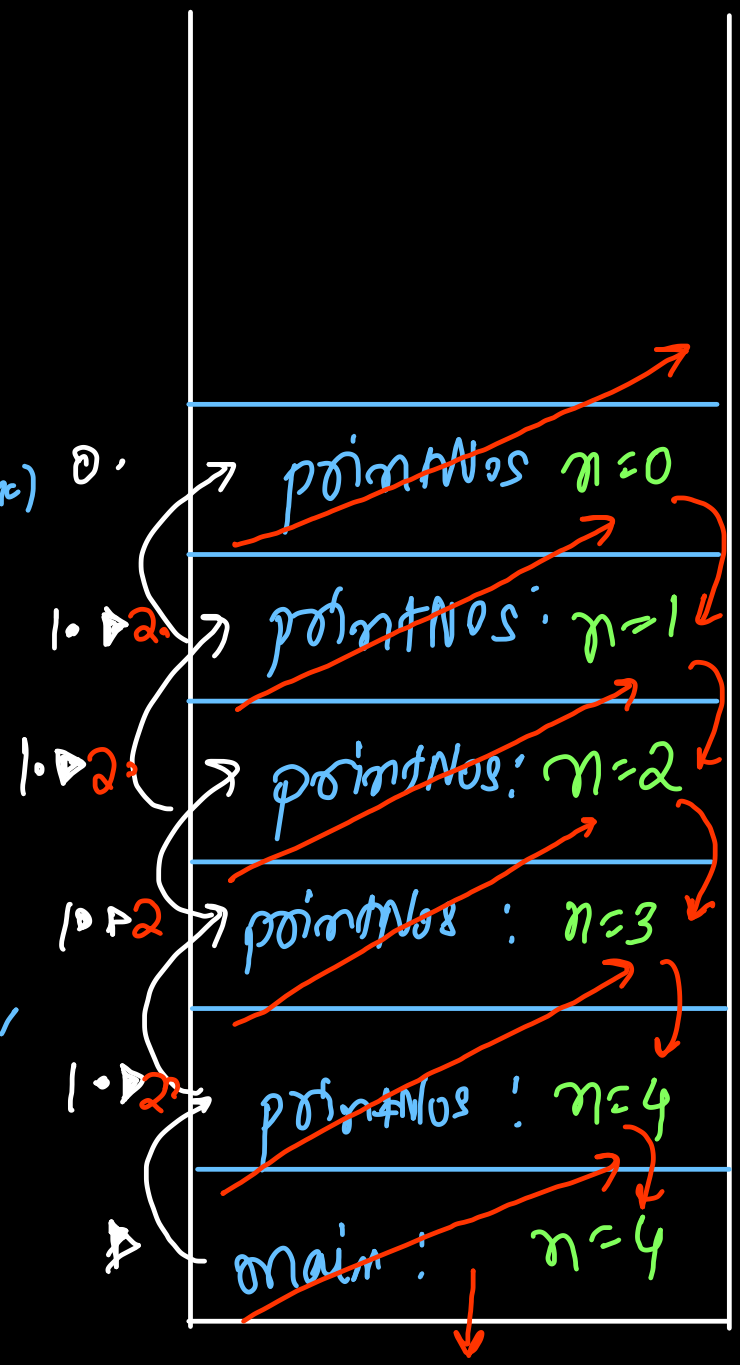
GFG

```java
public void printNos(int n)
  {
      // base case
0.    if(n == 0) return;

      // faith (recursive call)
1.    printNos(n - 1);

      // remaining work (postwork)
2.    System.out.print(n + " ");
  }
```

Time = O(n)

Space = O(n)

"Tail recursion"



Output console

1 2 3 4

Print Decreasing Increasing

n = 5

output : 5 4 3 2 1 1 2 3 4 5

n = 4

output: 4 3 2 1 1 2 3 4

Constraints

1. do not use iteration / loops
2. do not use 2 recursive fn
3. cannot pass multiple parameters

1. expectation
   p.s. void printNos(int n)
2. faith (recursive call)
   ↳ printNos(n-1);
3. remaining work
   Sysv(n); ↙ pre, ↘ post
4. base case.
   if (n == 0) return;

Factorial (Recursive fn returning values)

$n! = n \times (n-1)!$ ⇒ factorial(n) = n × factorial(n-1)

integer input
long output

$n! = n \times (n-1) \times (n-2) \times \cdots \cdots \quad 3 \times 2 \times 1$

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

$4! = 4 \times 3 \times 2 \times 1 = 24$

$3! = 3 \times 2 \times 1 = 6$

$2! = 2 \times 1 = 2$

$1! = 1 = 1$

$0! = 1$

① expectation

```
long factorial (int n)
    factorial(5) -> return 5! = 120
```

② faith(recursive call)

```
long sans = factorial(n-1);
```

③ remaining work

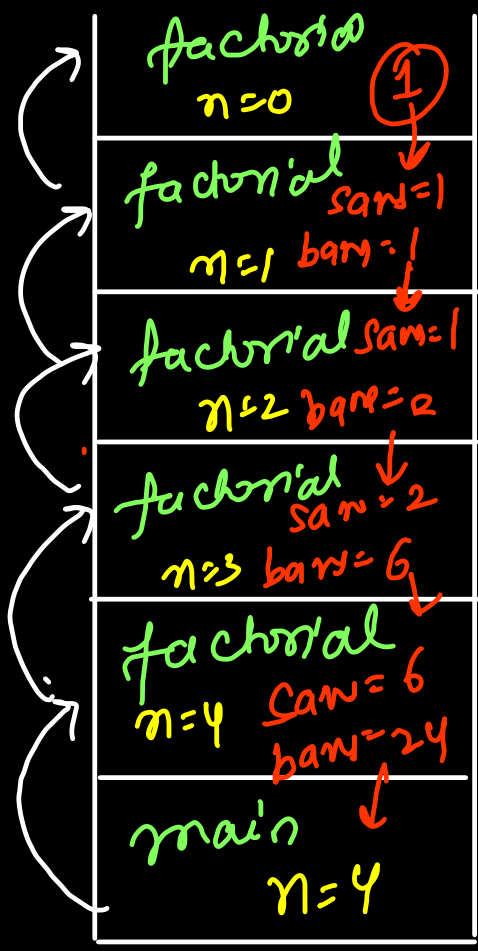
```
long bans = n * sans;
return bans;
```

④ base case `if(n==0) return 1;`

Recursion call stack

```java
static long factorial(int N){
    if(N == 0) return 1;
    // base case : 0! = 1

    long sans = factorial(N - 1);
    // recursive call or faith

    long bans = sans * N;
    return bans;
    // remaining work (postorder)
}

// or

static long factorialShort(int N){
    return (N == 0) ? 1 : factorialShort(N - 1) * N;
}
```

```
main(---){
    int n=4;
    Syso(factorial(n));
}
```



post = O(1)
pre = O(1)
calls = 1
depth = N

$(1+1) \times N + 1^N$

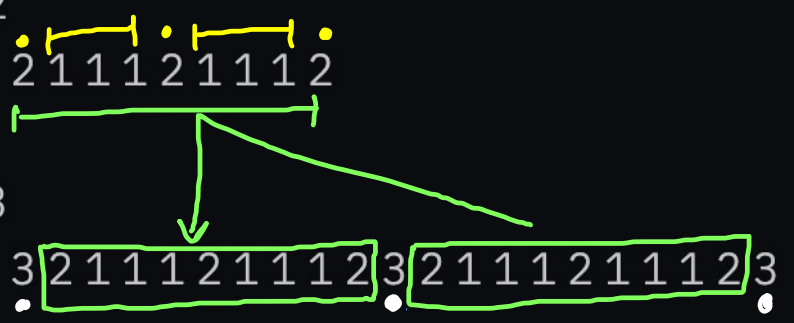
$O(2N+1) \Rightarrow O(N)$ time

Print ZigZag

Input1 -> 1
Output1 -> 1 1 1

Input2 -> 2
Output2 -> 2 1 1 1 2 1 1 1 2

Input2 -> 3
Output3 -> 3 2 1 1 1 2 1 1 1 2 3 2 1 1 1 2 1 1 1 2 3



1. expectation
2. faith
3. remaining work
4. base case

```
void printZigZag(int n) {
    if(n == 0) return;
    preorder  → Syso(n);
    left      ← printZigZag(n-1);
    inorder   → Syso(n);
    right     ← printZigZag(n-1);
    postorder → Syso(n);
}
```

pre/in/post ⇒ O(1)
calls ⇒ 2
depth ⇒ N

$1 \times N + 2^N$

$\Rightarrow O(N + 2^N) \Rightarrow O(2^N)$

linear exp^n exp^n

n = single / double digit only ⭐

```
void printZigZag (int n) {
    if(n == 0) return;
1. preorder  → Syso(n);
2. left      ← printZigZag (n-1);
3. inorder   → Syso(n);
4. right     ← printZigZag (n-1);
5. postorder → Syso(n);
}
```

output:

3

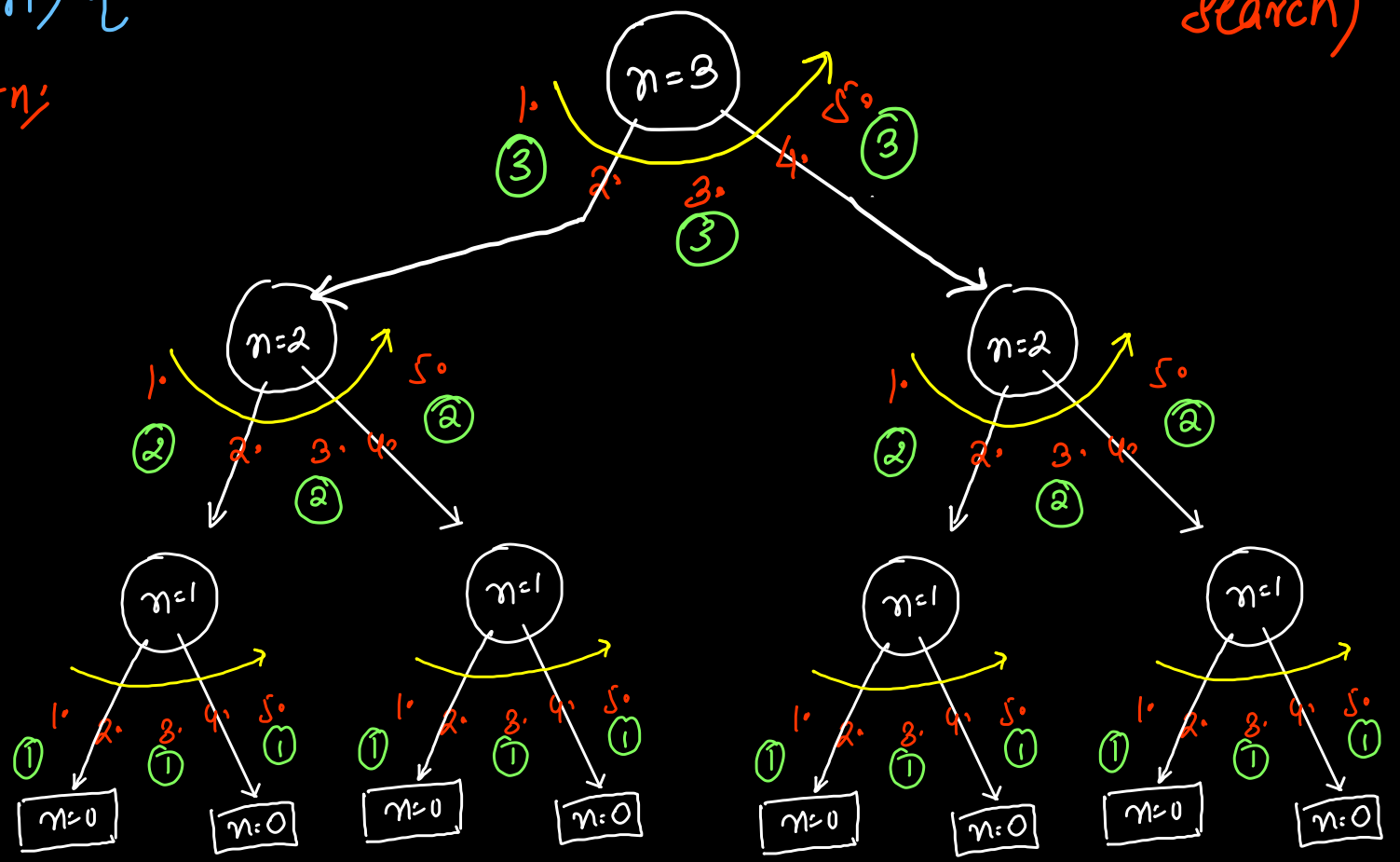
2 1 1 1 2 1 1 1 2

3

2 1 1 1 2 1 1 1

3

recursion tree (depth first search)



Space ⇒ O(n) depth (tree)

Time ⇒ O(2^N) exponential

Time Complexity ⇒ Total no of fn calls

If multiple recursive calls ⇒ total no of base case hits

(usually exponential)

Space Complexity ⇒ Max no of fns in stack at any given pt of time

(usually depth)

Fibonacci (TC $O(n)$)

integer N $\Longrightarrow$ return N^{th} term of fibonacci series

Series 0 1 1 2 3 5 8

F(0) F(1) F(2) F(3) F(4) F(5) F(6)

$F(n) = F(n-1) + F(n-2)$ $n > 1$ (recurrence relation)

$F(0) = 0$
$F(1) = 1$
} fixed → base case

Iterative Soln (using Three Pointers)

{ a=b ; b=c; c=a+b; }

↳ Dynamic Programming + Space optimization

Time: O(n)

Space: O(1)

Recursion

↳ inplace X

extraspace ✓

① expectation

int fib(int n)

n → input

n=5

F(n) → return / output

F(5) = 5

② faith

① prev1 = fib(n-1); F(4) = 3

② prev2 = fib(n-2); F(3) = 2

③ remaining work

F(5) = F(4) + F(3) = 3+2 = 5

curr = prev1 + prev2

return curr;

④ base case

if(n ≤ 1)

return n;

⇓

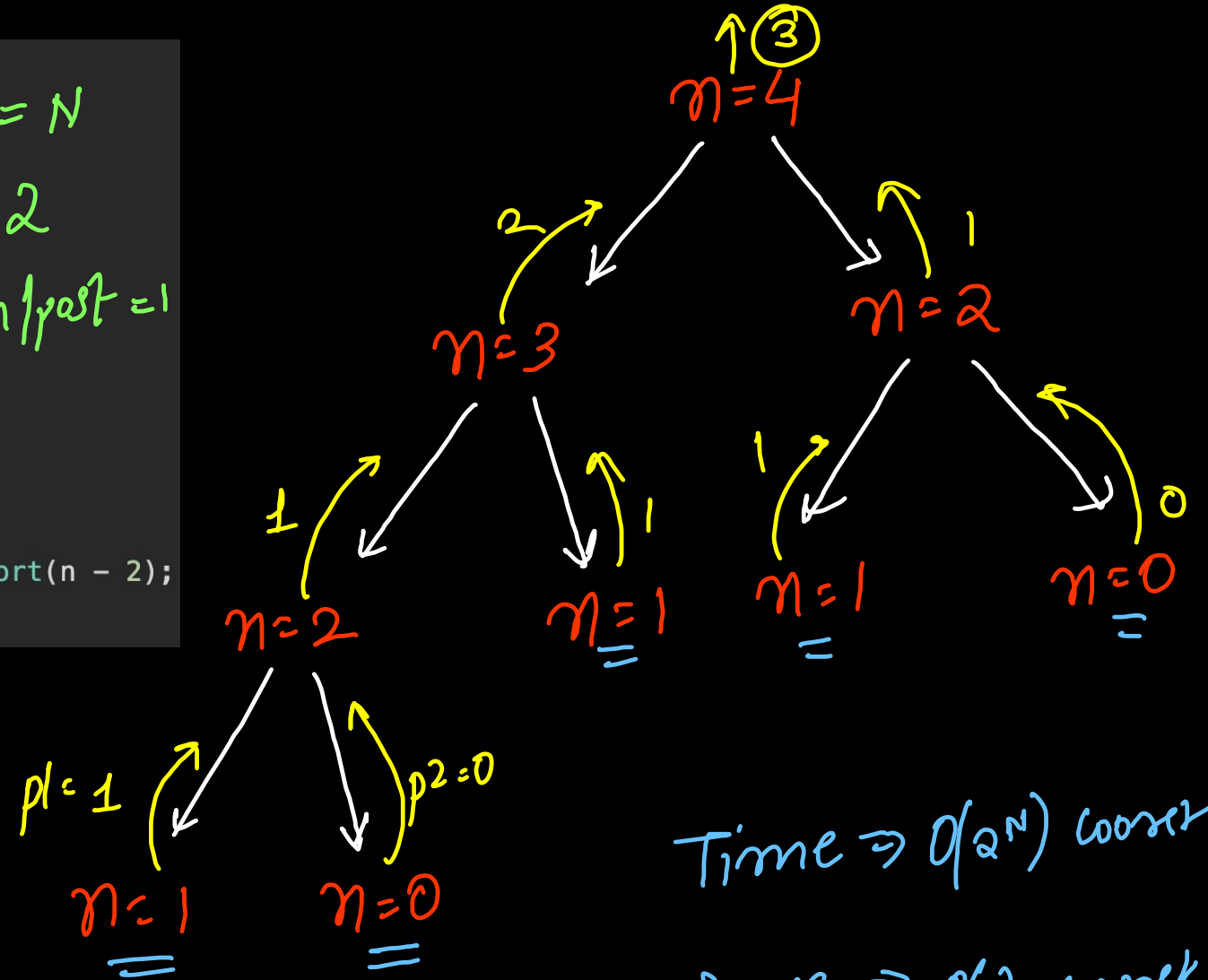
if(n == 0) return 0;

if(n == 1) return 1;

```java
public int fib(int n) {
    if(n == 0) return 0;
    if(n == 1) return 1;

    int prev1 = fib(n - 1);
    int prev2 = fib(n - 2);

    int curr = prev1 + prev2;
    return curr;
}

public int fibShort(int n){
    return (n <= 1) ? n : fibShort(n - 1) + fibShort(n - 2);
}
```

Left
Right
work

depth = N
Calls = 2
pre/in/post = 1



Time ⇒ O(2^N) worst

Space ⇒ O(n) worst

Average case ⇒

$O(\phi^N)$

↳ $\left(\frac{\sqrt{5}+1}{2}\right)^N = 1.4^N$

time = $1 \times N + 2^N = 2^N + 1$

⇒ $O(2^N)$

$T(n) = 2T(n-1) + O(1)$

$2 \times (T(n-1) = 2T(n-2) + O(1))$

$2^2 \times (T(n-2) = 2T(n-3) + O(1))$

$2^3 \times$ ⋮

$2^N (T(2) = 2T(1) + O(1))$

$O(1)$

$T(n) = 2^N T(1) + (2 + 2^2 + 2^3 + \cdots 2^N)$

$= 2^N + 2^N = 2^N \Rightarrow$ exponential ★

Time Complexity

Shortcut Formula (90% correctness)

$$Time = (Preorder + Postorder) * depth + (calls) \wedge depth$$

Exponentiation

Pow(x, n) LC 50

eg1 double int

$x = 2.0, \quad n = 5$

$x^n = (2.0)^5 = 2.0 * 2.0 * 2.0 * 2.0 * 2.0 = 32.0$ double

eg2

$x = 3.0, \quad n = -2$

$x^n = (3.0)^{-2} = \frac{1}{(3.0)^2} = \frac{1}{3.0 \times 3.0} = \frac{1}{9.0}$

eg3

$x = -1.0, \quad n = 10$

$x^n = (-1.0)^{10} = 1.0$

Brute force

(1) Expectation → double pow(double x, int n)

pow(2.0, 5) ⇒ 32.0

(2) Faith (recursive call) → double sans = pow(x, n-1);

pow(2.0, 4) ⇒ 16.0

(3) Remaining work → double bans = sans * x;

$x^n = x^{n-1} * x$

return bans;

recurrence relation

(4) base case

↳ if (n==0) return 1.0;

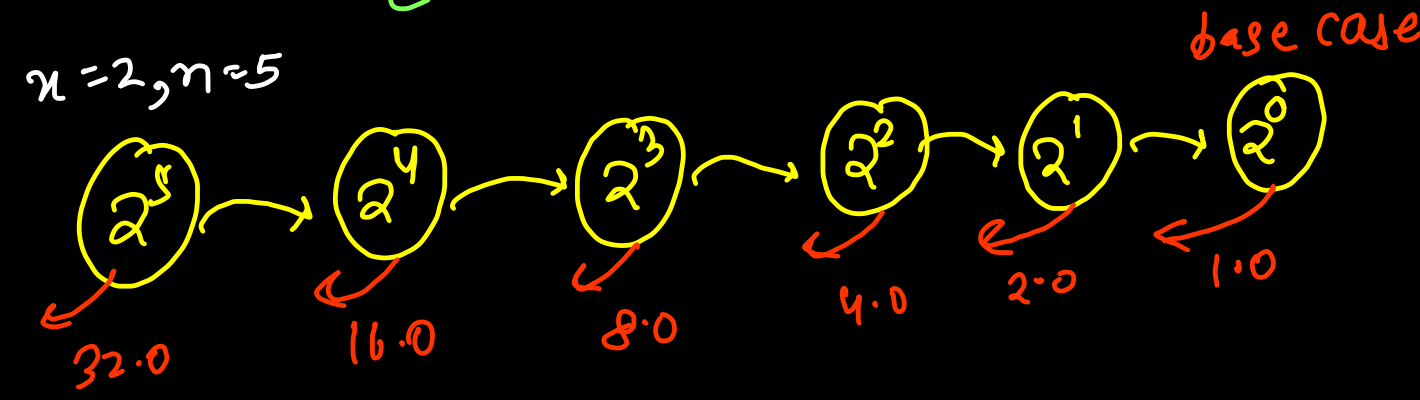
$x^0 = 1.0$

approach ①

```java
public double pow(double x, long n){
    if(n == 0) return 1.0;
    // base case

    double sans = pow(x, n - 1);
    // recursive call / faith

    double bans = sans * x;
    // remaining work
    return bans;
}

public double myPow(double x, int n) {
    if(n < 0){
        return 1.0 / pow(x, -1l * n);
    }

    return pow(x, n);
}
```

recursion ⟶ space ↑ ⟶ 10^4-10^5

{ Stack overflow error }



time = 1^n + $1*n$ = $1+n$ ⇒ $O(n)$

linear

space = $O(n)$ recursion call stack

$n = 10^9$ (2^{31})

approach ②

$x^n = x^{n-1} * x$

$2^5 = 2^4 * 2$

$2^{16} = 2^{15} * 2$

better

$2^{16} \rightarrow 2^{15} \rightarrow 2^{14} \rightarrow 2^{13} \rightarrow 2^{12} \rightarrow 2^{11} \rightarrow \cdots$

logarithmic

$x^n = x^{n/2} * x^{n/2} \Rightarrow n = \text{even}$

$2^{16} = 2^8 * 2^8$ } even

$2^8 = 2^4 * 2^4$

$x^n = x^{n-1} * x \Rightarrow n = \text{odd}$

$2^{15} = 2^{14} * 2$ } odd

$2^7 = 2^6 * 2$

linear

approach (2)

```java
public double pow(double x, long n){
    if(n == 0) return 1.0;
    // base case

    if(n % 2 == 1){
        // x ^ n = x ^ n - 1 * x
        double sans = pow(x, n - 1);
        double bans = sans * x;
        return bans;
    }
    // x ^ n = x ^ n / 2 * x ^ n / 2
    double sans = pow(x, n / 2);
    double bans = sans * sans;
    return bans;
}

public double myPow(double x, int n) {
    if(n < 0){
        return 1.0 / pow(x, -1l * n);
    }

    return pow(x, n);
}
```

n odd

logarithmic relation

n even

long

$n = -\infty \Rightarrow -n = -(-\infty) = +\infty$

$a = 2,\ n = 20$

Logarithmic

n

$n/2$

$n/4$

$n/8$

$\vdots$

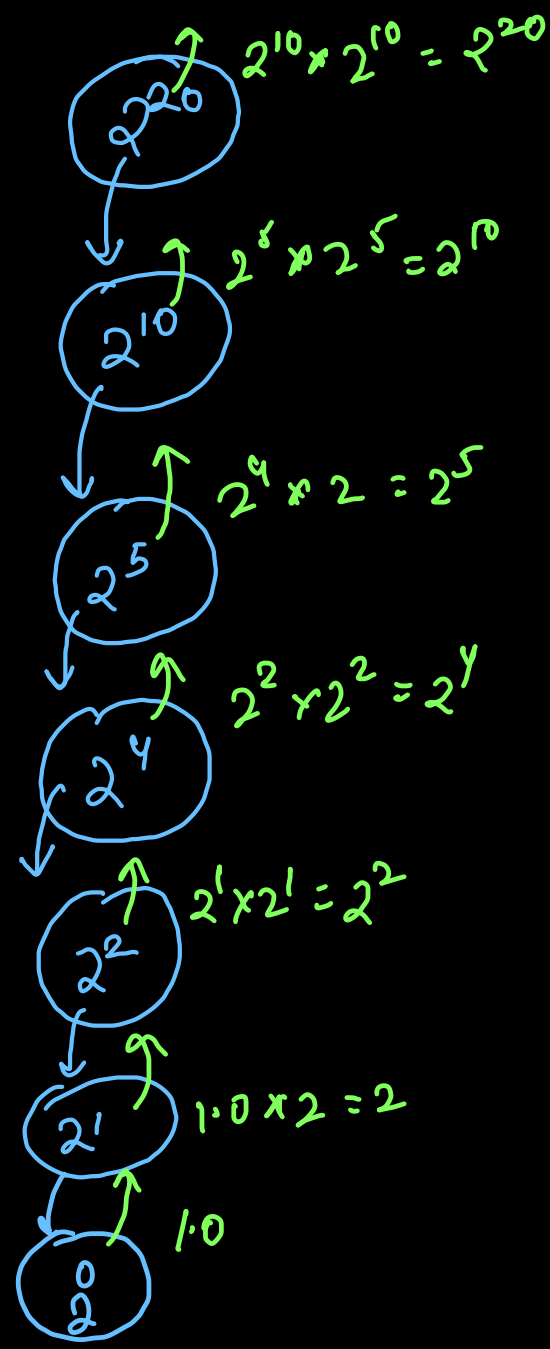
1

$O(\log_2 n)$

Time

Space

(accepted)



better approach exists?

Yes → Binary Exponentiation

↓

Bit manipulation (Iterative/Loops)

Time : $O(\log_2 n)$, Space => $O(1)$

Common mistake :→

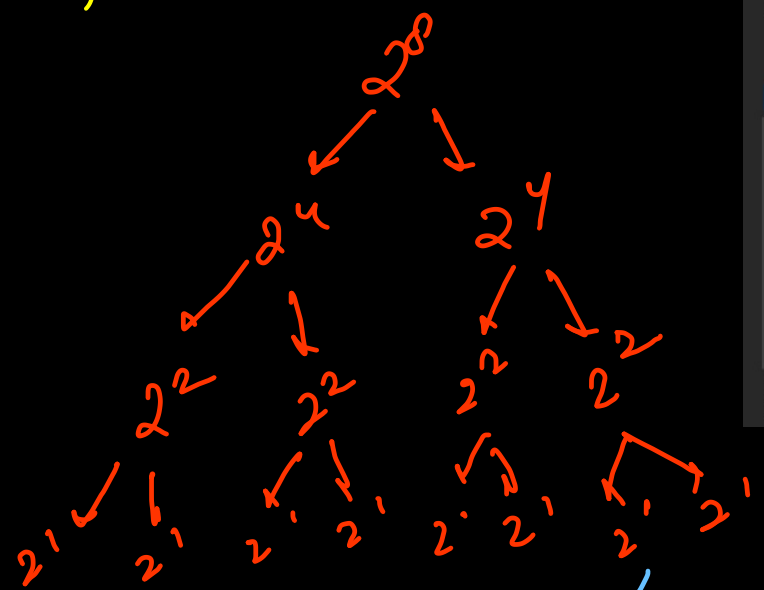
$$x^n = x^{n/2} * x^{n/2}$$

return pow(x, n/2) * pow(x, n/2);

calls = 2

depth = $\log_2 n$

pre/post = O(1)



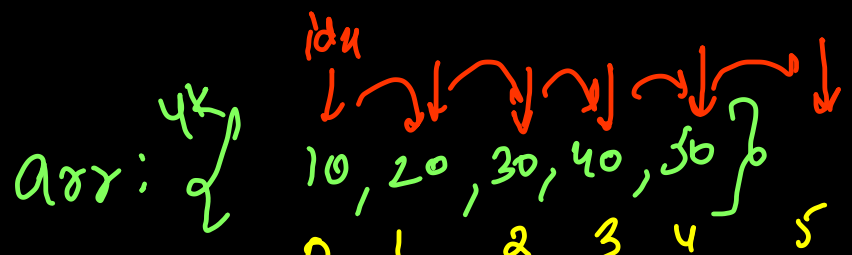
time = $1 * \log_2 n + 2^{\log_2 n} = \log n + n \Rightarrow O(n)$ linear

```java
public double pow(double x, long n){
    if(n == 0) return 1.0;

    if(n % 2 == 1)
        return pow(x, n - 1) * x;

    return pow(x, n / 2) * pow(x, n / 2);
}

public double myPow(double x, int n) {
    if(n < 0){
        return 1.0 / pow(x, -1l * n);
    }

    return pow(x, n);
}
```

1st call

2nd call

Print All Elements of Array



① expectation

② faith

③ remaining work

④ base case

```
void printArray(int[] arr, int idx)
```

⇒ printArray(4k, 0) = 10 20 30 40 50

```
printArray(arr, idx+1);
```

⇒ printArray(4k, 1) = 20 30 40 50

```
Syso(arr[idx]);
```

preorder ⇒ L to R
postorder ⇒ R to L

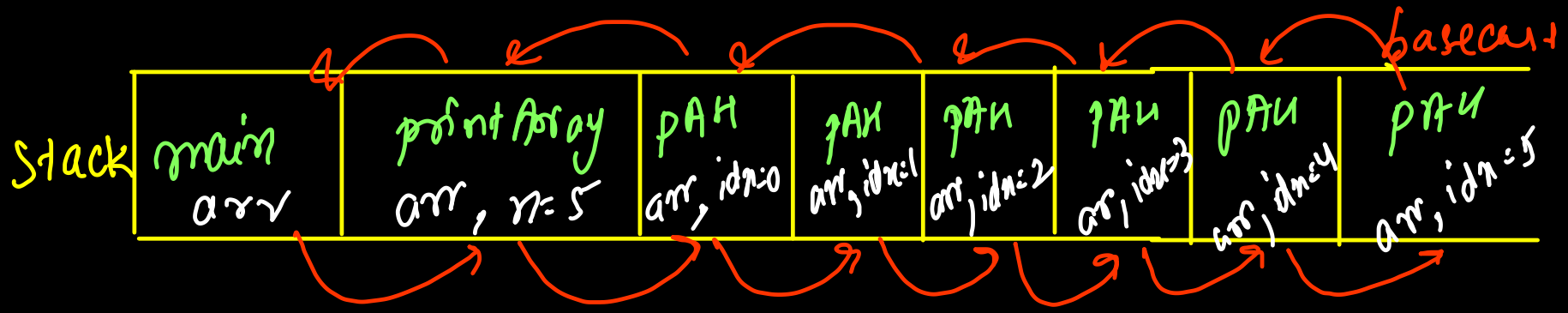
```
if(idx == arr.length) return;
```

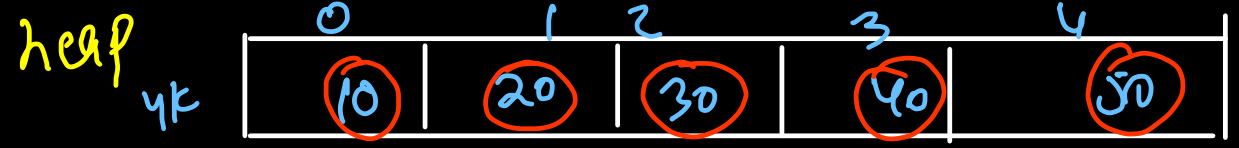
```java
void printArrayHelper(int[] arr, int idx){
    if(idx == arr.length) return;

    System.out.print(arr[idx] + " ");
    printArrayHelper(arr, idx + 1);
}

void printArray(int arr[], int n) {
    printArrayHelper(arr, 0);
}
```

Time ↓ O(n) $n = 10^8$ op/sec ,

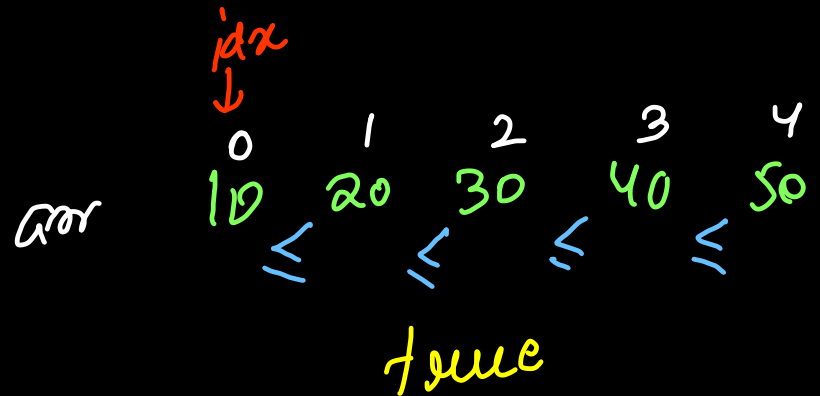
Space ↓ O(n) for call stack $n = 2 \times 10^4$ stack size

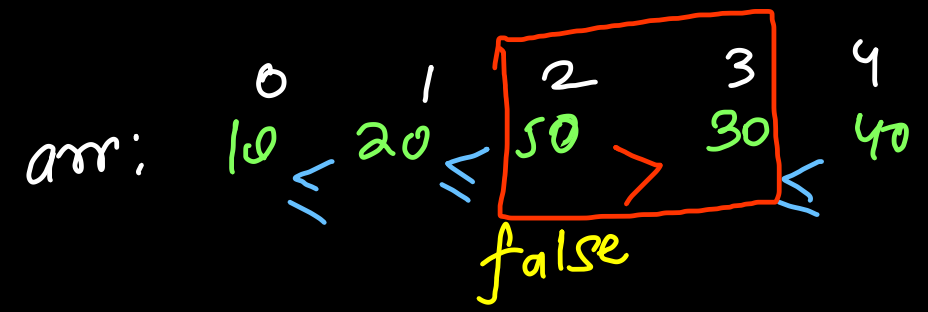
$n = 10^5 \Rightarrow$ stack overflow (in Java)





Check Array Is Sorted





Recursion

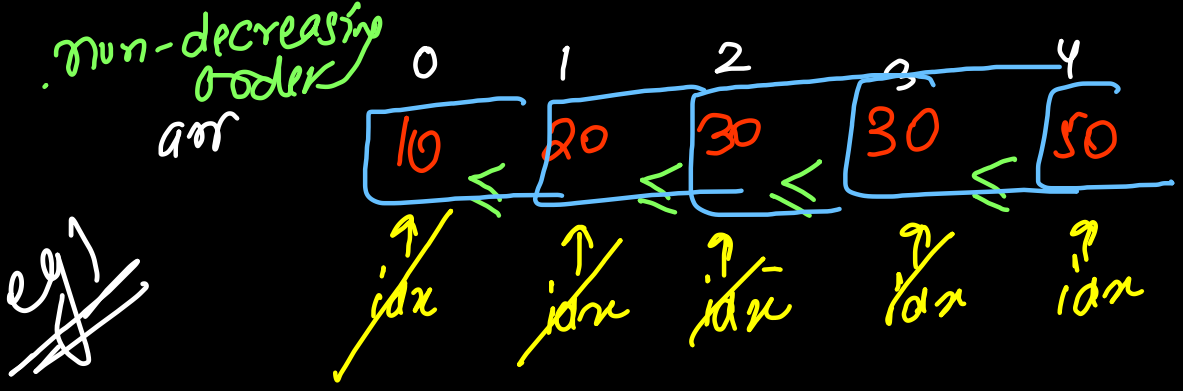
base case ⟶ `boolean checkSorted(int[] arr, int idx){`
`if(idx >= arr.length-1) return true;`

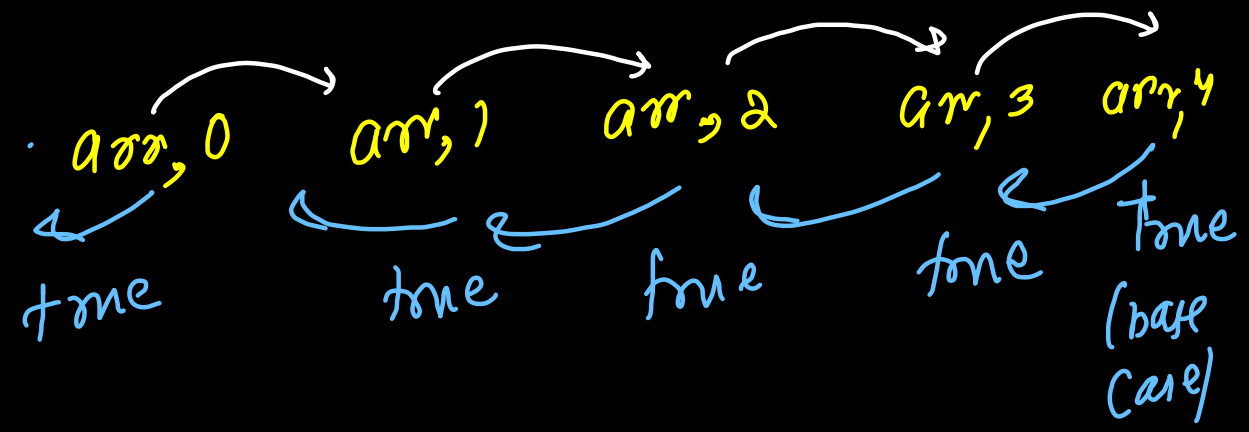
remaining work ⟶ `if(arr[idx] > arr[idx+1])`
`return false;`

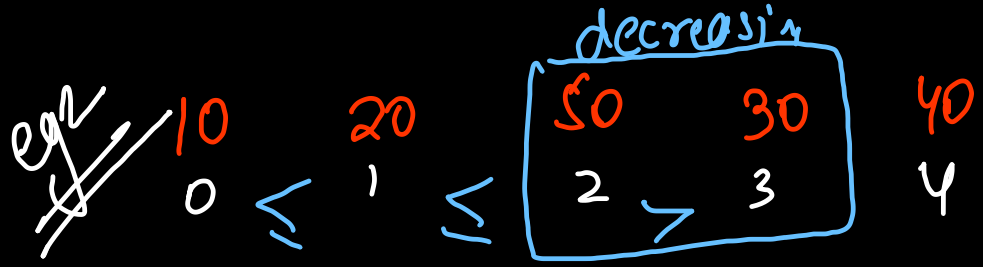
fair ⟶ `return checkSorted(arr, idx+1);`

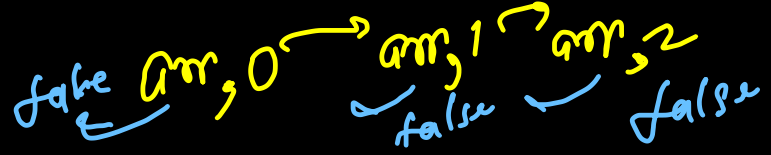
}

```java
boolean checkSorted(int[] arr, int idx){
    if(idx >= arr.length - 1){
        // 0 elements or 1 element
        return true; // already sorted
    }

    if(arr[idx] > arr[idx + 1]){
        // decreasing pair (remaining work)
        return false;
    }

    // faith (recursive call)
    return checkSorted(arr, idx + 1);
}

boolean arraySortedOrNot(int[] arr, int n) {
    return checkSorted(arr, 0);
}
```





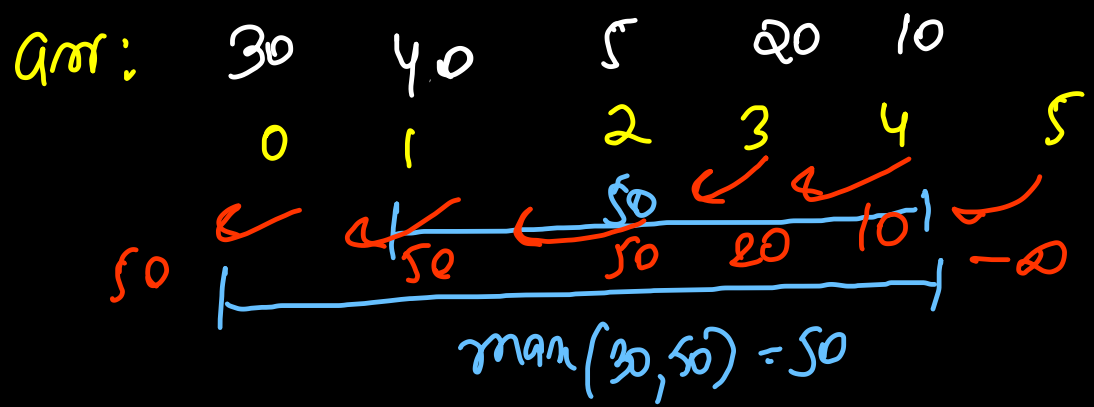




Time = O(n)
Space = O(n)
} linear

GfG — Largest No in Array (max value)

basecase
① empty array
② 1/y element

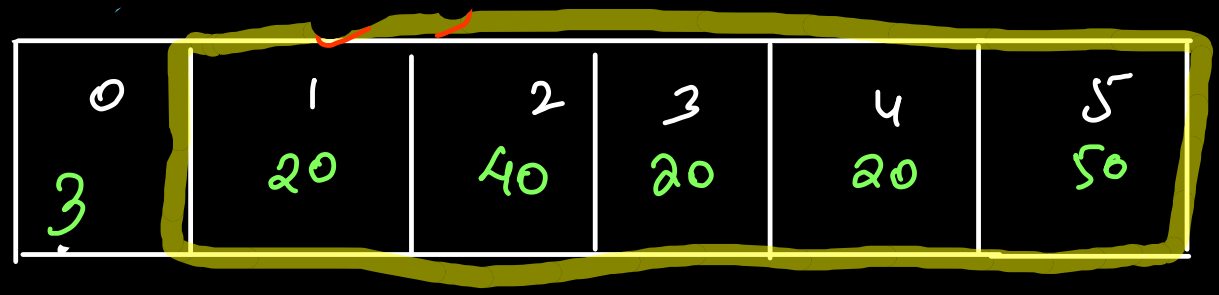



```
int max(int[]arr, int idx){
  if(idx==n) return -inf;
          (Integer.MIN_VALUE)

  int sans = max(arr, idx+1);

  int bans = Math.max
          (arr[idx], sans);

  return bans;
}
```

Time :→ O(n)

Space :→ O(n)

Array (First & Last Occurence)



target = 20 → [1, 4]
fi li

first index => leftmost occurence

last index => rightmost occurence

target = 40 → [2, 2]
fi, li

target = 10 → [-1, -1]
fi li

recursive linear Search

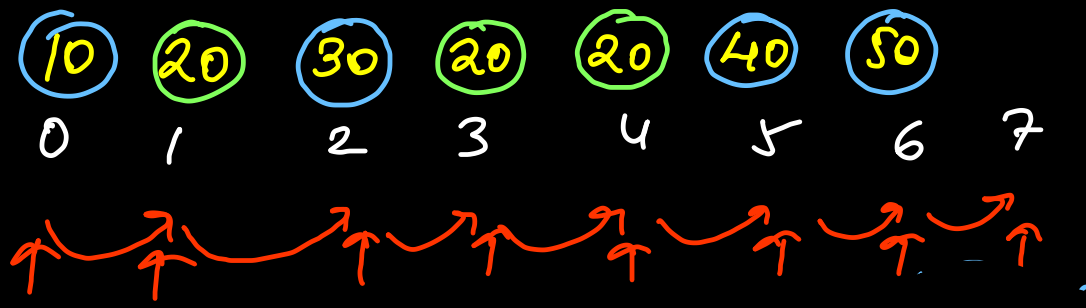
```
public int[] linearSearch(int[] arr, int target, int idx){
    if(idx==n) return new int[]{-1,-1};
    int[] ans = linearSearch(arr, target, idx+1);

    if(arr[idx]!=target) return ans;

    if(ans[0]!=-1){ ans[0]=idx; return ans;}
    else{ ans[0]=ans[1]=idx; return ans;}
}
```

```java
static int[] linearSearch(int[] arr, int target, int idx){
    if(idx == arr.length){
        // base case: emptry array: target not found
        return new int[]{-1, -1};
    }

    int[] ans = linearSearch(arr, target, idx + 1);

    if(arr[idx] == target){
        if(ans[0] == -1) {
            // first and last occurence
            ans[0] = ans[1] = idx;
        } else {
            // first occurence
            ans[0] = idx;
        }
    }

    return ans; // current element is not equal to target
}

static int[] findIndex(int arr[], int N, int target) {
    return linearSearch(arr, target, 0);
}
```

"faith"

remaining work

target = 20



fi	1	1	3	3	4	-1	-1	-1
li	4	4	4	4	4	-1	-1	-1

time = O(n) linear

Space = O(n) recursion call Stack

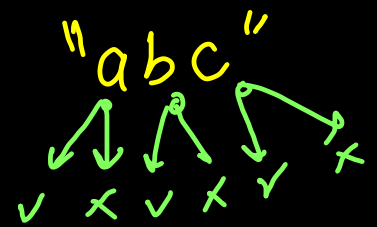
Power Set

all subsequences → part of string or subset of characters but in the same order of inpu

removing 0 or some or all of characters of input string will form a subsequence

"archit"

	substring	subsequence
arc	✓	✓
chit	✓	✓
aci	✗	✓
cra	✗	✗



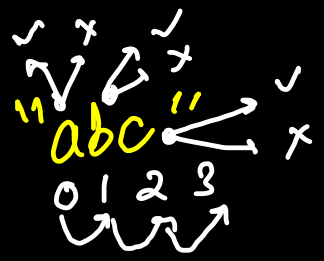
$2 \times 2 \times 2$

$= 2^3 = 8$

N string length

powerset : 2^N

"All substrings will be subsequence but vice-versa is not true"



0 1 2

a x b x c x → " "

a ✓ b x c x → "a"

a x b ✓ c x → "b"

a x b x c ✓ → "c"

a ✓ b ✓ c x → "ab"

a ✓ b x c ✓ → "ac"

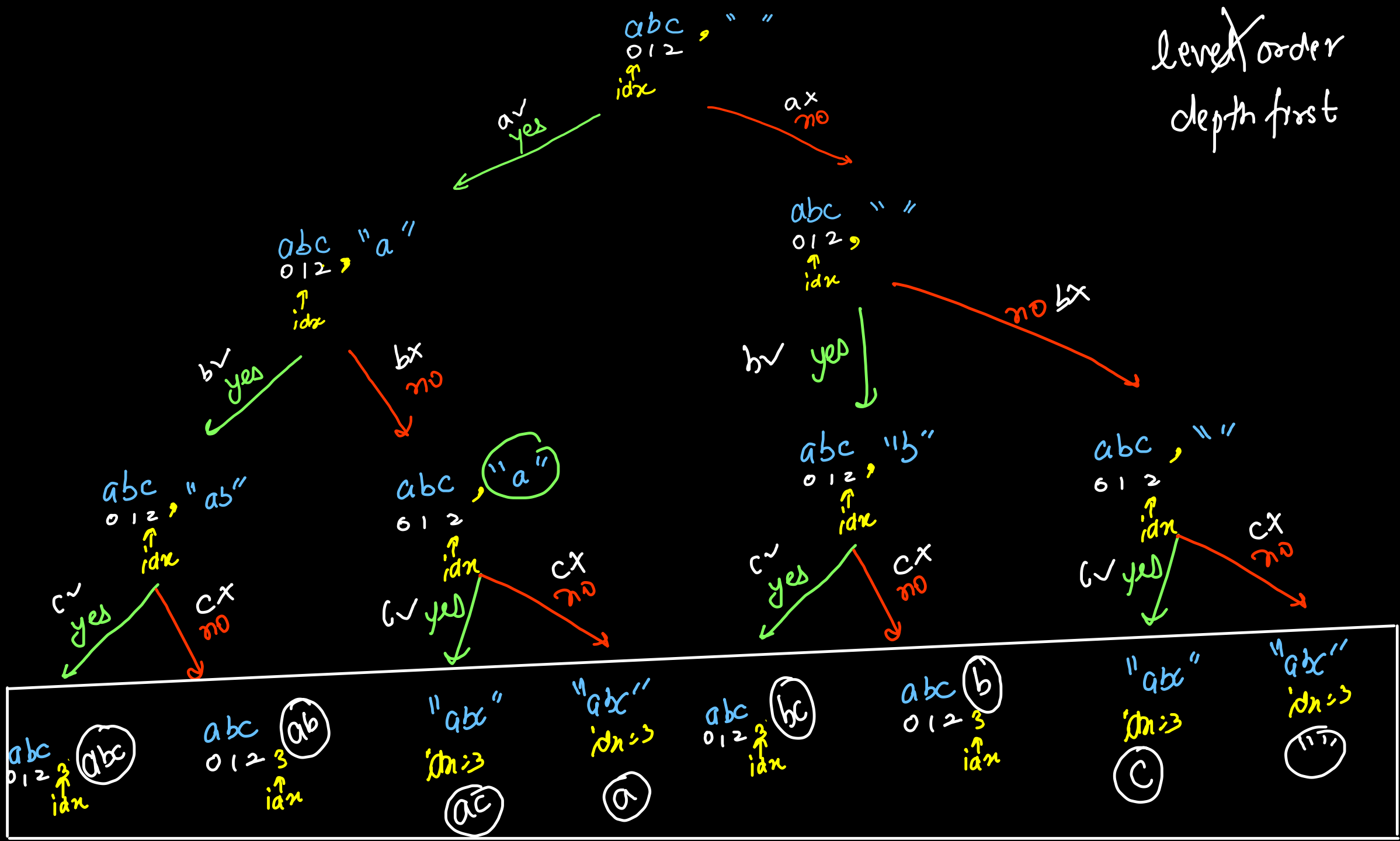
a x b ✓ c ✓ → "bc"

a ✓ b ✓ c ✓ → "abc"

recursion (print subsequences)

str, 0, " "

```
void printSubsequences(String input, int idx,
                       String output)
{
    if(idx == str.length()) {
        Syso(output);
        return;
    }
yes printSubsequences(input, idx+1, output +
                      input[idx])
no  printSubsequences(input, idx+1, output);
}
```


level order
depth first
abc , " "
idx
a ✓ yes
a ✗ no
abc , "a"
b ✓ yes
b ✗ no
abc , " "
b ✓ yes
no b ✗
abc , "ab"
c ✓ yes
c ✗ no
abc , "a"
c ✓ yes
c ✗ no
abc , "b"
c ✓ yes
c ✗ no
abc , " "
c ✓ yes
c ✗ no
"abc"
idx=3
"abc"
idx=3
"abc"
idx=3
abc
ab
abc
ac
a
bc
b
"abc"
c

app①

```java
List<String> result;

public void printSubsequences(String input, int idx, String output){
    if(idx == input.length()){
        if(output.length() == 0) return; // empty subsequence
        result.add(output);
        return;
    }

    // yes recursive call
    printSubsequences(input, idx + 1, output + input.charAt(idx));

    // no recursive call
    printSubsequences(input, idx + 1, output);
}

public List<String> AllPossibleStrings(String s)
{
    result = new ArrayList<>();
    printSubsequences(s, 0, "");
    Collections.sort(result);
    return result;
}
```

calls

concat(java) → new string

print

depth = N
calls = 2
pre/post = N

$N * N + 2^N$

$N^2 + 2^N$

$\Rightarrow O(2^N)$ time exponential

Time = $O(2^N)$ exponential

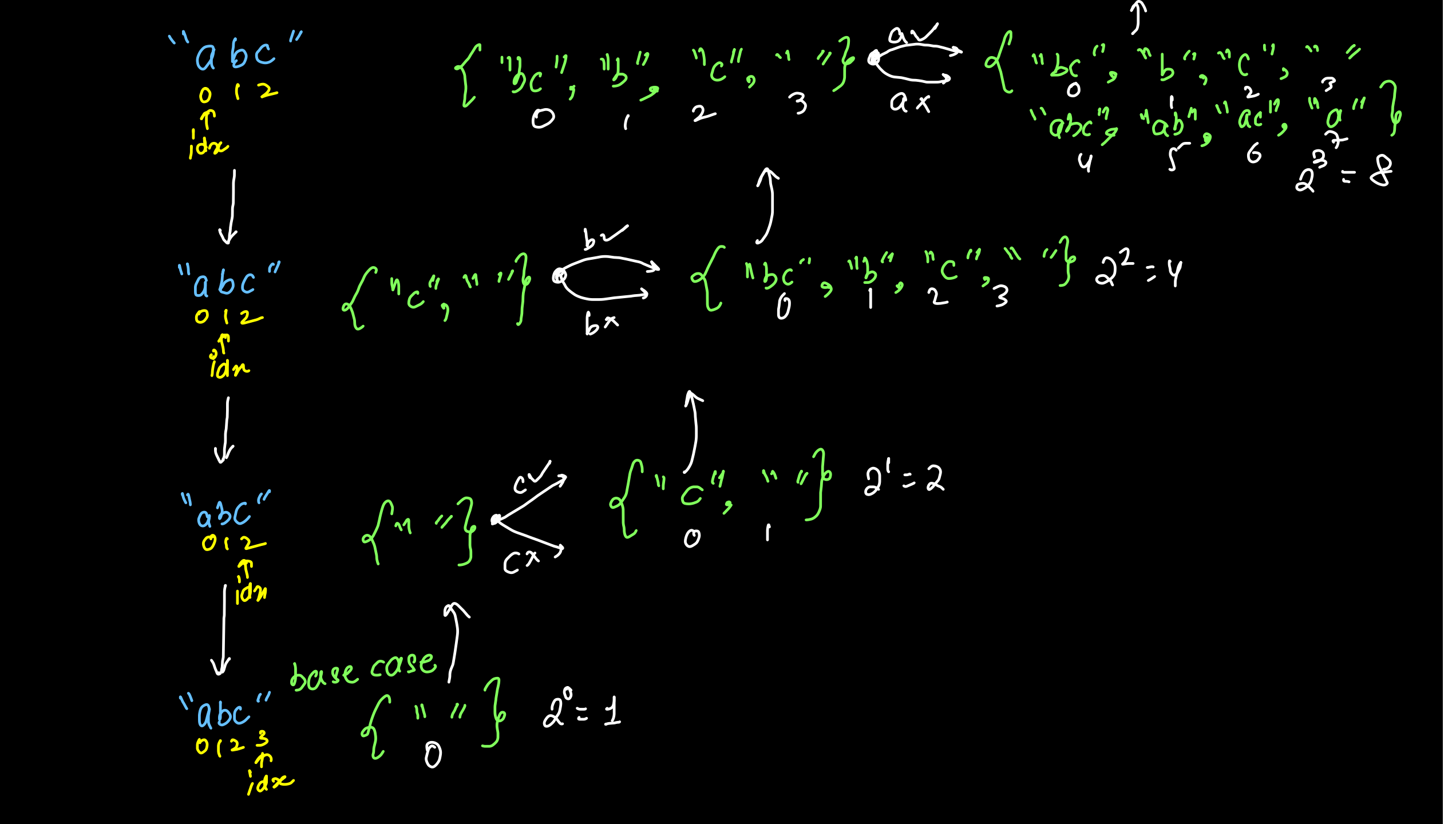
Space = $O(N)$ depth

app②

```java
public void printSubsequences(String input, int idx, String output, List<String> result){
    if(idx == input.length()){
        if(output.length() == 0) return; // empty subsequence ignore
        result.add(output);
        return;
    }

    // yes recursive call
    printSubsequences(input, idx + 1, output + input.charAt(idx), result);

    // no recursive call
    printSubsequences(input, idx + 1, output, result);
}

public List<String> AllPossibleStrings(String s)
{
    List<String> result = new ArrayList<>();
    printSubsequences(s, 0, "", result);
    Collections.sort(result);
    return result;
}
```

Get Subsequences (Power Set)

```
List<String> getSubsequences (String input, int idx) {

    if (idx == input.length()) {
        List<String> result = new ArrayList<>();
        result.add("");
        return result;
    }

    List<String> result = getSubsequences(input, idx+1);

    //Todo: remaining work

}
```


"abc"
0 1 2
idx
"abc"
0 1 2
idx
"abc"
0 1 2
idx
"abc"
0 1 2 3
idx
{ "bc", "b", "c", "" }
0 1 2 3
ax
ax
{ "bc", "b", "c", "" }
0 1 2 3
{ "c", "" }
bx
bx
{ "bc", "b", "c", "" } 2^2 = 4
0 1 2 3
{ "" }
cx
cx
{ "c", "" } 2^1 = 2
0 1
base case
{ "" } 2^0 = 1
0
{ "bc", "b", "c", "" }
0 1 2 3
"abc" "ab" "ac" "a"
4 5 6 7
2^3 = 8

```java
public List<String> AllPossibleStrings(String s){
    List<String> result = getSubsequences(s, 0);
    Collections.sort(result); // lexicographically sorted
    result.remove(0); // empty subsequence ignored
    return result;
}

public List<String> getSubsequences(String input, int idx){
    if(idx == input.length()){
        // base case
        List<String> result = new ArrayList<>();
        result.add(""); // 2 ^ 0 = 1
        return result;
    }

    // faith or recursive call
    List<String> result = getSubsequences(input, idx + 1);

    // remaining work
    int n = result.size();
    for(int i = 0; i < n; i++){
        String res = input.charAt(idx) + result.get(i);
        result.add(res);
    }
    return result;
}
```

"" ⇒ {""}

postorder

pow(allsubsequences) ⇒ O(2^N)

calls = 1

depth = N

pre/post = 2^N

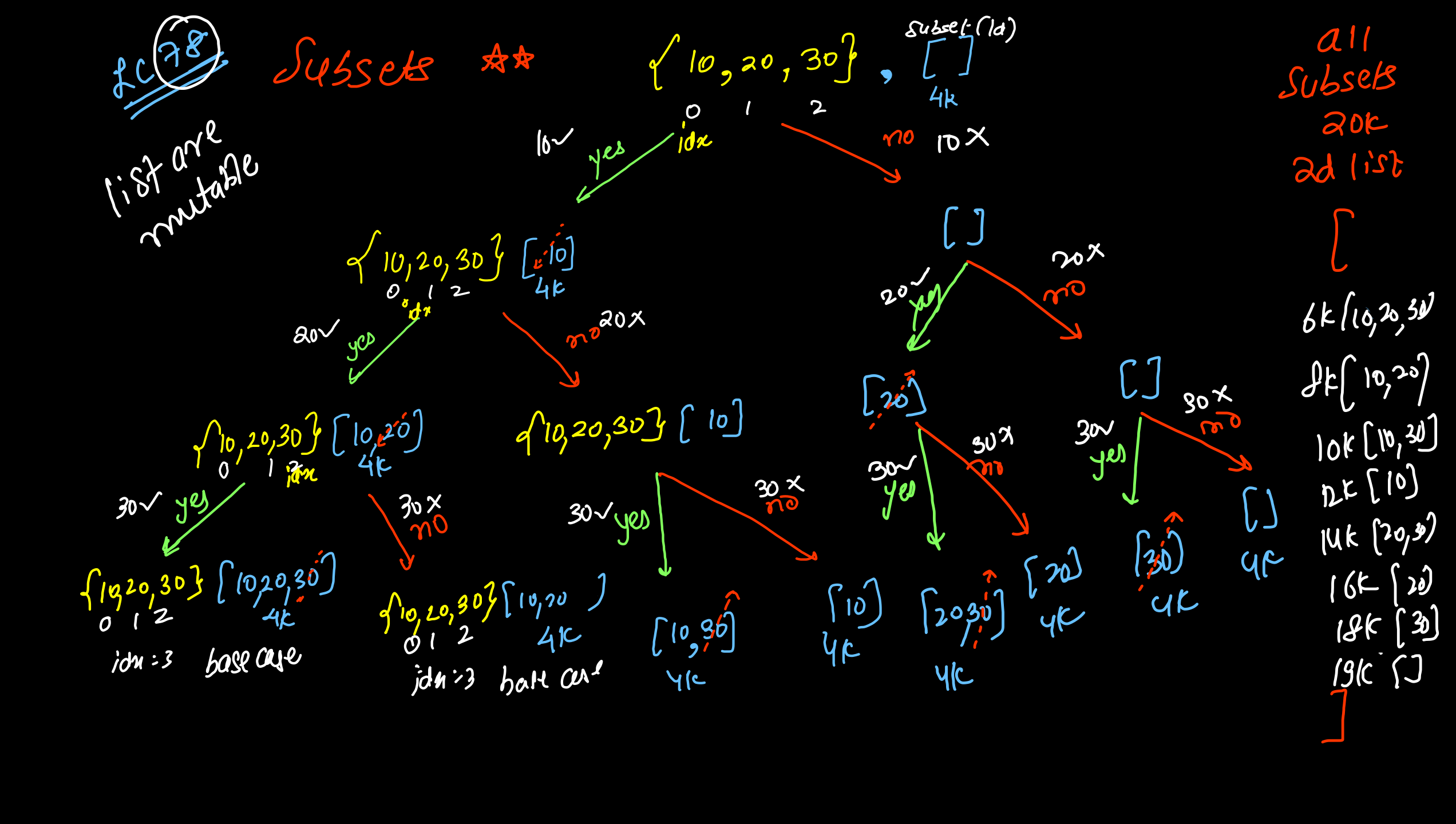
time = 2^N * N + 1^N

= O(2^N * N) ⇝ Nmax = 15

"exponential"

space = O(N) ⇝ Nmax = 10^4

"linear" Stack overflow


LC 78
Subsets **
list are mutable
{10,20,30}
subset = [ ]
4k
0 1 2
idx
10 ✓ yes
no 10 ✗
{10,20,30} [10]
20 ✓ yes
no 20 ✗
{10,20,30} [10,20]
30 ✓ yes
30 ✗ no
idx = 3 base case
{10,20,30} [10,20]
30 ✓ yes
30 ✗ no
idx = 3 base case
{10,20,30} [10]
30 ✓ yes
30 ✗ no
[10,30]
[10]
[ ]
20 ✓ yes
20 ✗ no
[20]
[20,30]
[20]
[ ]
30 ✓ yes
30 ✗ no
[30]
[ ]
all subsets 2D list
[
0k [10,20,30]
1k [10,20]
2k [10]
3k [10,30]
4k [20]
5k [20,30]
6k [30]
7k [ ]
]

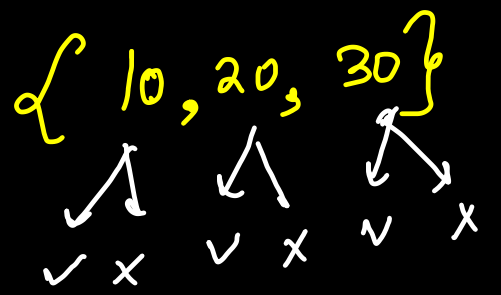
```java
List<List<Integer>> result;

public void printSubsets(int[] input, int idx, List<Integer> output){
    if(idx == input.length){
        // result.add(output); // shallow copy
        result.add(new ArrayList<>(output)); // deep copy
        return;
    }

    // yes call
    output.add(input[idx]); // insert
    printSubsets(input, idx + 1, output);
    output.remove(output.size() - 1); // backtracking (remove)

    // no call
    printSubsets(input, idx + 1, output);
}

public List<List<Integer>> subsets(int[] nums) {
    result = new ArrayList<>();
    List<Integer> output = new ArrayList<>();
    printSubsets(nums, 0, output);
    return result;
}
```

Time = O(2^N) exponential

Space = O(N) linear

↳ undo the step performed before the fn call during postorder

Subsets - II LC 90



2^3 = 8 subsets

"Unique Subsets"

{ 10, 20, 20 }

2^3 = 8 subsets ×

{ }

{ 10, 20, 20 }

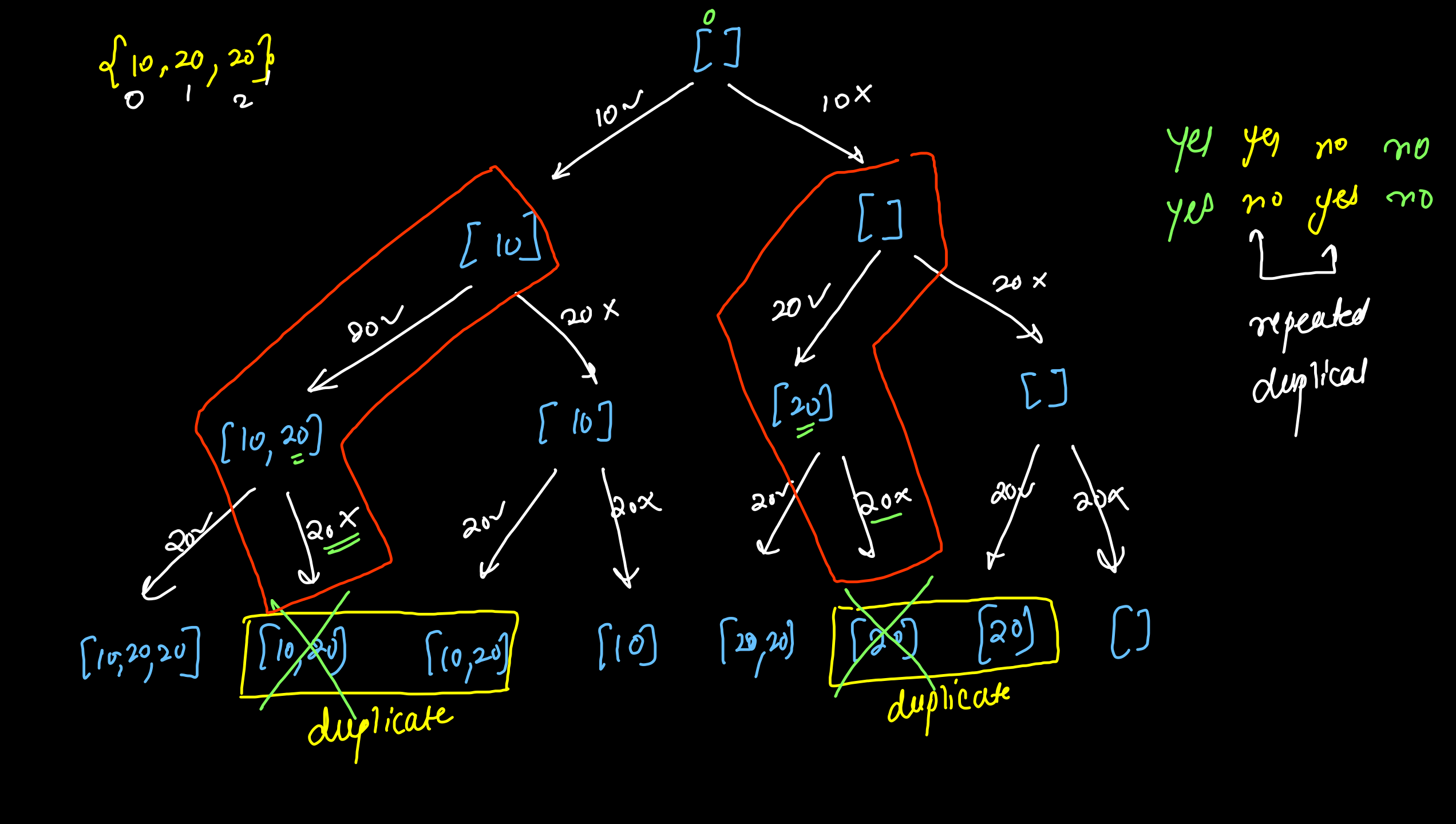
{ 10 }

{ 20 }

{ 10, 20 }

{ 20, 20 }

6 subsets


{10, 20, 20}
0 1 2
[]
10 ✓
10 ×
[10]
[]
20 ✓
20 ×
[10, 20]
[10]
[20]
[]
20 ✓
20 ×
20 ✓
20 ×
20 ✓
20 ×
20 ✓
20 ×
[10,20,20]
[10,20] [10,20]
duplicate
[10]
[20,20]
[20] [20]
duplicate
[]
yes yes no no
yes no yes no
repeated
duplical

```java
List<List<Integer>> result;

public void printSubsets(int[] input, int idx, List<Integer> output){
    if(idx == input.length){
        // result.add(output); // shallow copy
        result.add(new ArrayList<>(output)); // deep copy
        return;
    }

    // yes call
    output.add(input[idx]); // insert
    printSubsets(input, idx + 1, output);
    output.remove(output.size() - 1); // backtracking (remove)

    // no call
    if(output.size() > 0 && output.get(output.size() - 1) == input[idx])
        return;  // yes -> no on duplicates will be restiricted
    printSubsets(input, idx + 1, output);
}

public List<List<Integer>> subsetsWithDup(int[] nums) {
    Arrays.sort(nums);
    result = new ArrayList<>();
    List<Integer> output = new ArrayList<>();
    printSubsets(nums, 0, output);
    return result;
}
```

last index

* → remove duplicates (pruning)

→ * to apply pruning (adj. duplicates)

Order was not mandatory

Time: $O(2^N)$ exponential

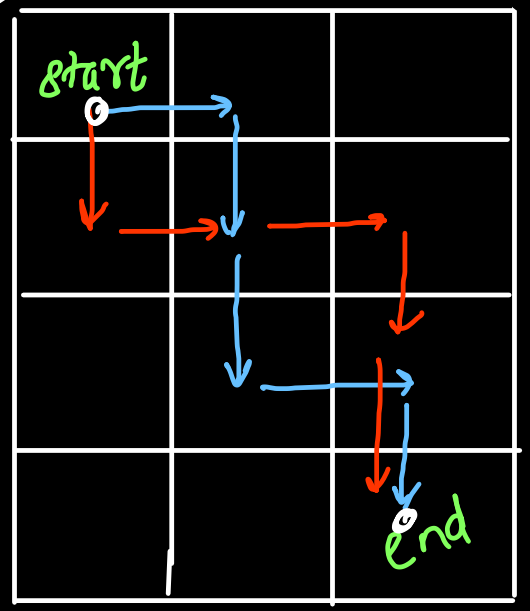
Space: $O(N)$ depth

Recursion on Maze / Matrix

↓

bhool bhulaiya

top left



bottom right

4 x 3
m x n
rows x cols

(1) constraint

choices → right move (H / R)
choices → down move (V / D)

all paths

eg) "RDDRD" or "HVVHV"

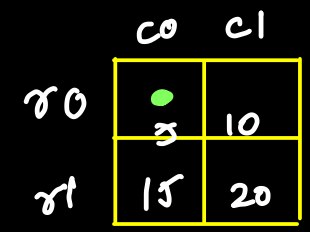
eg) "DRRDD" or "VHHVV"

no of steps

$= \frac{5!}{2!\,3!}$

$= \frac{(m+n-2)!}{(m-1)!\,(n-1)!}$

output:
[[5,10,20],[5,15,20]]

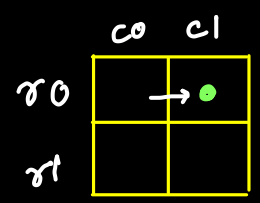


Time = $O(2^{N+M})$

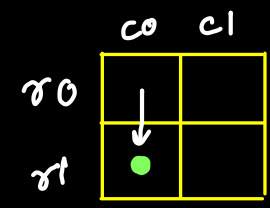
Space = $O(N+M)$

right (col++)

down (row++)



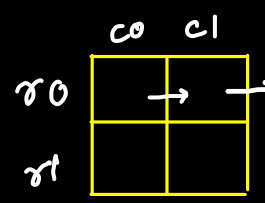
"R"



"D"

right

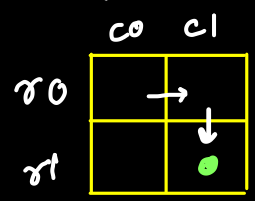
down



c==n

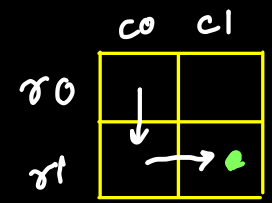
"RR"

base case (negative)



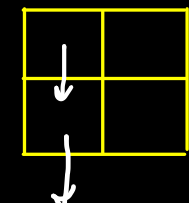
"RD"

base case (positive)



"DR"

base case (positive)



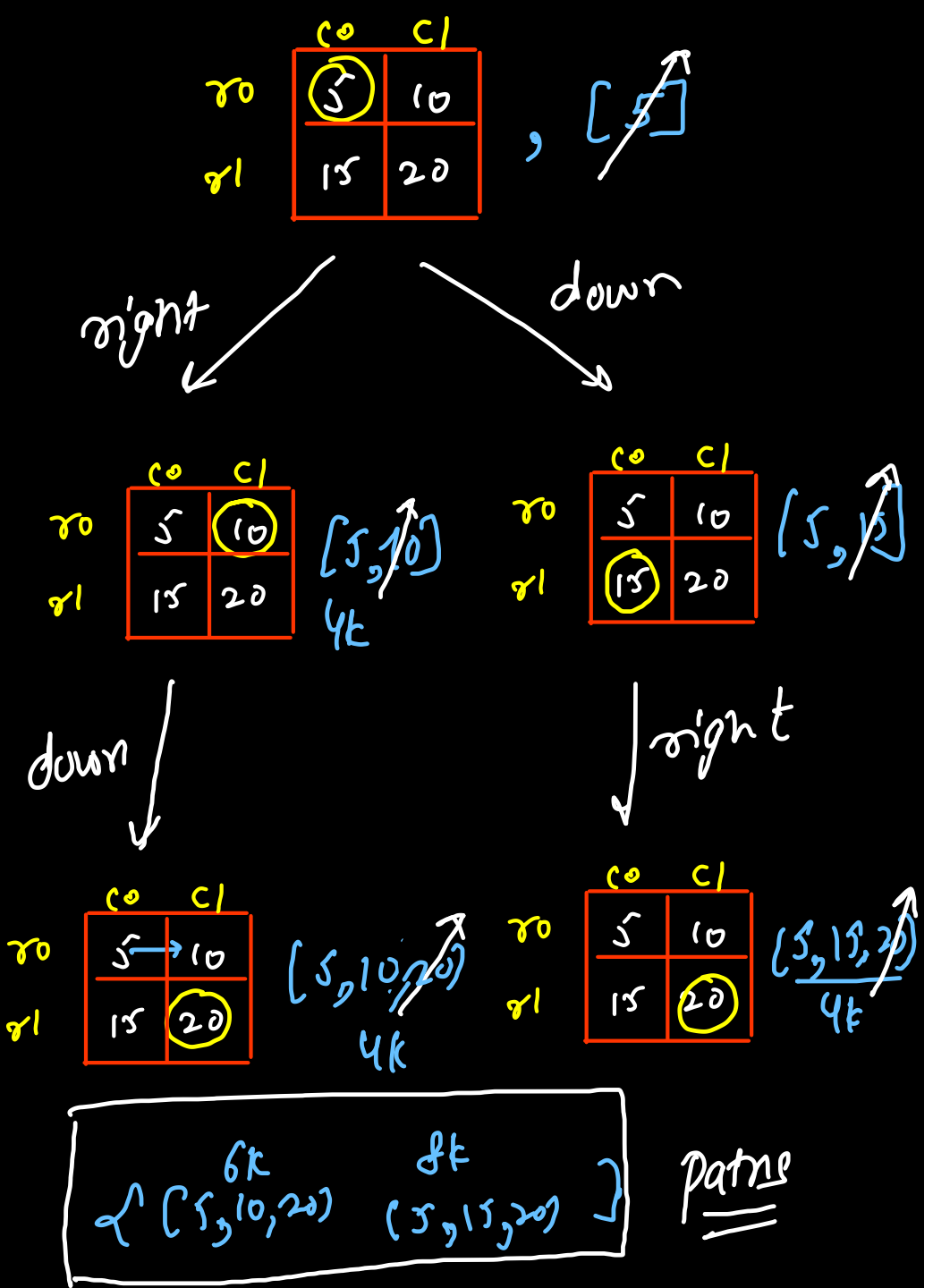
"DD"

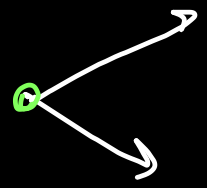
r==m

base case (negative)

```java
static List<List<Integer>> paths;

0 references
public static void printPaths(int[][] mat, int row, int col, List<Integer> path){
    if(row == mat.length || col == mat[0].length){
        // negative base case (out of matrix)
        return;
    }

    if(row == mat.length - 1 && col == mat[0].length - 1){
        // positive base case (bottom right)
        path.add(mat[row][col]); // work
        paths.add(new ArrayList<>(path)); // deep copy
        path.remove(path.size() - 1); // backtracking (undo)
        return;
    }

    path.add(mat[row][col]); // work

    // faith recursive calls
    printPaths(mat, row, col + 1, path); // right or horizontal
    printPaths(mat, row + 1, col, path); // down or vertical

    path.remove(path.size() - 1); // backtracking (undo)
}

0 references
public static int[][] printAllPaths(int[][] mat, int m, int n) {
    paths = new ArrayList<>();
    List<Integer> path = new ArrayList<>();
    printPaths(mat, 0, 0, path);

    int[][] res = new int[paths.size()][paths.get(0).size()];

    for(int r = 0; r < res.length; r++){
        for(int c = 0; c < res[0].length; c++){
            res[r][c] = paths.get(r).get(c);
        }
    }

    return res;
}
```

Convert 2D arrayList into 2D array





Tuesday Thursday 10 PM - 12 AM (2hr)

Saturday Sunday 10 AM - 1 PM

Temp (1-2 weeks)

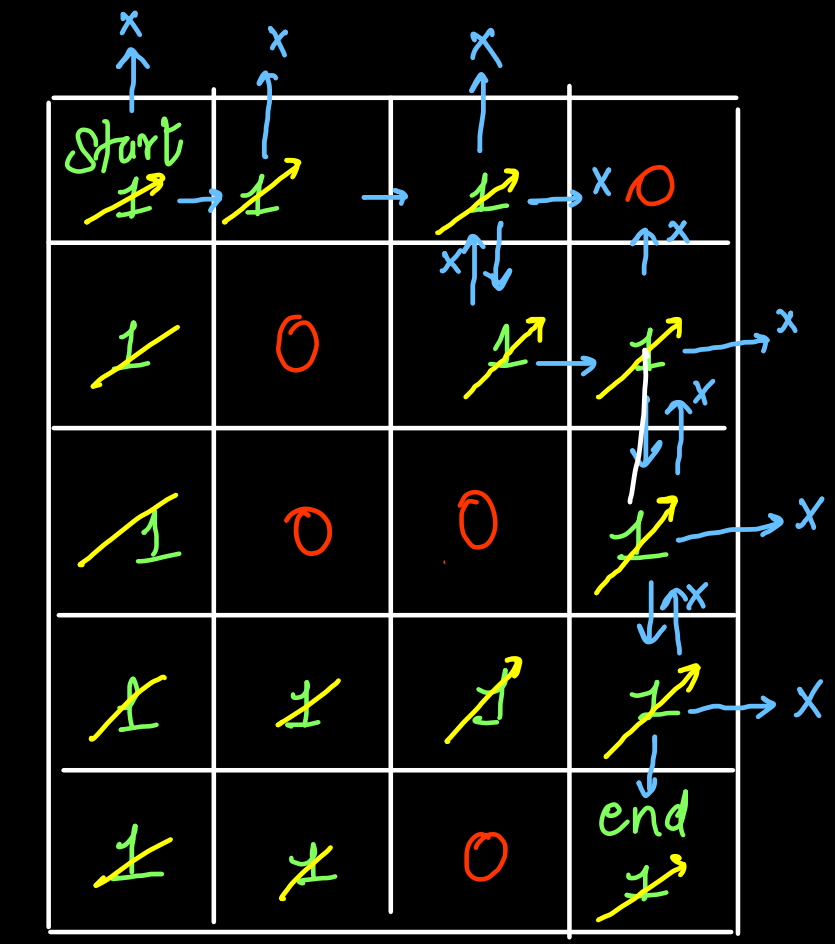
1) Tuesday - Thursday → 3 hr class (8 PM to 11 PM)

2) Tuesday - Wed - Thursday → 9 to 11 PM

3) no change

Rat in a Maze (Backtracking)

#Calls without marking visited ⇒ infinite loop



"RRDRDDD"

calls :→

1) top
2) right
3) bottom
4) left

Valid path

① source → dest
② no blockage
③ each cell can be visited atmost once.

positive base case → cell == dest

negative base case → blocked cell
→ out of matrix
→ visited cell

```java
static ArrayList<String> paths;

public static void findPath(int[][] mat, int row, int col, String path){
    if(row < 0 || col < 0 || row == mat.length || col == mat[0].length){
        // index out of bound -> negative base case
        return;
    }
    if(mat[row][col] == -1 || mat[row][col] == 0){
        // visited call or blocked cell -> negative base case
        return;
    }
    if(row == mat.length - 1 && col == mat[0].length - 1){
        // destination -> positive base case
        paths.add(path);
        return;
    }

    mat[row][col] = -1; // visited mark

    // top or up
    findPath(mat, row - 1, col, path + "U");

    // down
    findPath(mat, row + 1, col, path + "D");

    // left
    findPath(mat, row, col - 1, path + "L");

    // right
    findPath(mat, row, col + 1, path + "R");

}

public static ArrayList<String> findPath(int[][] mat, int n) {
    paths = new ArrayList<>();
    findPath(mat, 0, 0, "");
    return paths;
}
```

"Print Any One Path"

→ all paths

mark visited

never be visited again in any path

never be visited again in same path

returning (path rejected)

⇒ backtracking

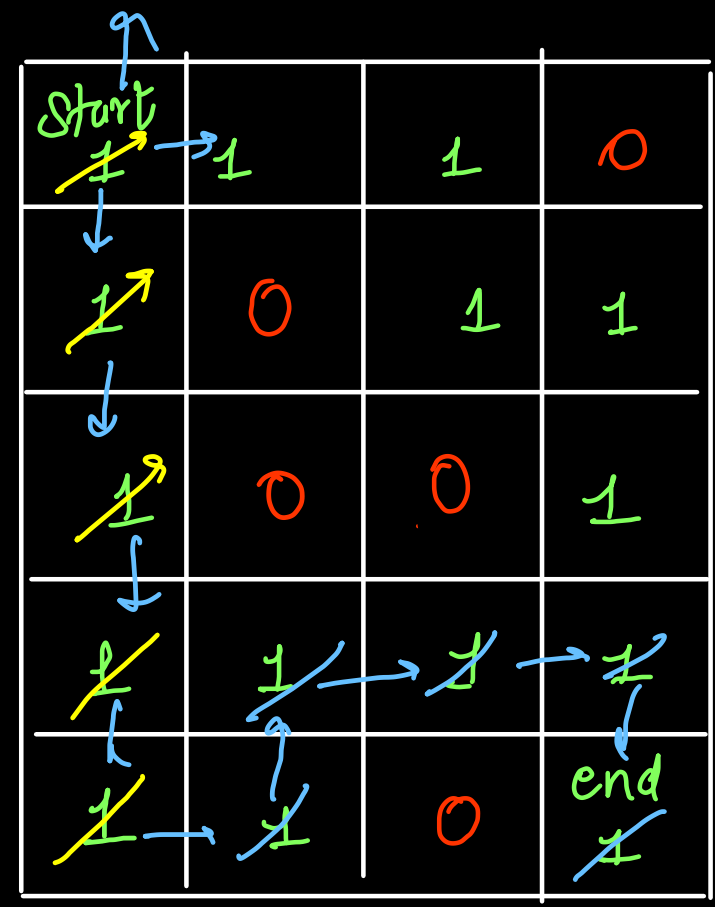
⇓

Unvisited mark

```java
public static void findPath(int[][] mat, int row, int col, String path){
    if(row < 0 || col < 0 || row == mat.length || col == mat[0].length){
        // index out of bound -> negative base case
        return;
    }
    if(mat[row][col] == -1 || mat[row][col] == 0){
        // visited call or blocked cell -> negative base case
        return;
    }
    if(row == mat.length - 1 && col == mat[0].length - 1){
        // destination -> positive base case
        paths.add(path);
        return;
    }

    mat[row][col] = -1; // visited mark

    // top or up
    findPath(mat, row - 1, col, path + "U");

    // down
    findPath(mat, row + 1, col, path + "D");

    // left
    findPath(mat, row, col - 1, path + "L");

    // right
    findPath(mat, row, col + 1, path + "R");

    mat[row][col] = 1; // unvisited mark (backtracking)
}

public static ArrayList<String> findPath(int[][] mat, int n) {
    paths = new ArrayList<>();
    findPath(mat, 0, 0, "");
    return paths;
}
```

diff paths can have same cell

"print all paths"

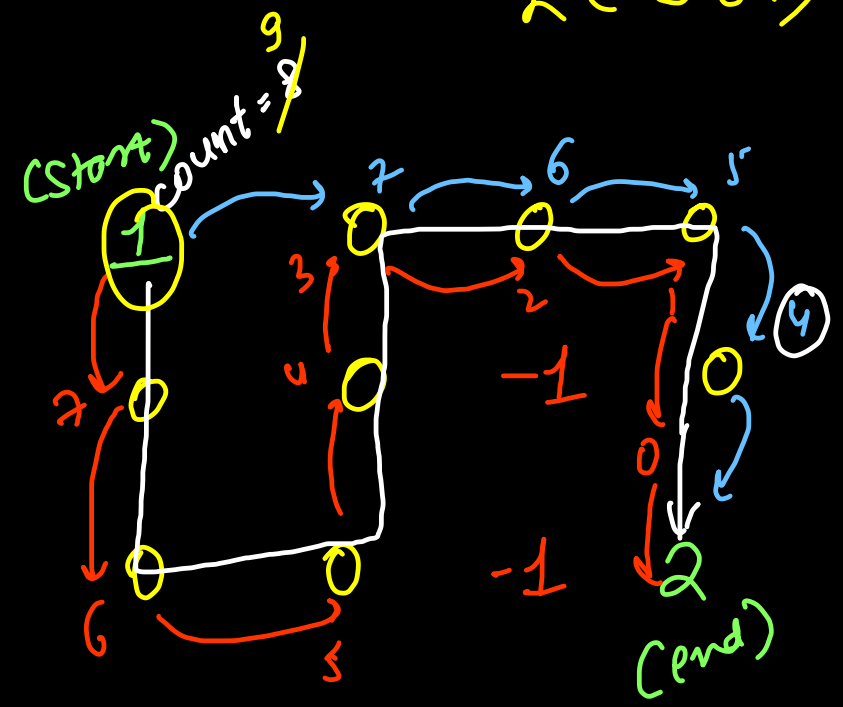


```java
static ArrayList<String> paths;

static int[] x = {-1, +1, 0, 0};
static int[] y = {0, 0, -1, +1};
static char[] moves = {'U', 'D', 'L', 'R'};

public static void findPath(int[][] mat, int row, int col, String path){
    if(row < 0 || col < 0 || row == mat.length || col == mat[0].length){
        // index out of bound -> negative base case
        return;
    }
    if(mat[row][col] == -1 || mat[row][col] == 0){
        // visited call or blocked cell -> negative base case
        return;
    }
    if(row == mat.length - 1 && col == mat[0].length - 1){
        // destination -> positive base case
        paths.add(path);
        return;
    }

    mat[row][col] = -1; // visited mark

    for(int idx = 0; idx < 4; idx++){
        int dx = x[idx], dy = y[idx];
        char move = moves[idx];
        findPath(mat, row + dx, col + dy, path + move);
    }

    mat[row][col] = 1; // unvisited mark (backtracking)
}

public static ArrayList<String> findPath(int[][] mat, int n) {
    paths = new ArrayList<>();
    findPath(mat, 0, 0, "");
    return paths;
}
```

Short syntax using xdy arrays

Rat in Maze - Followup (counts)

LC 980) Unique Paths - III



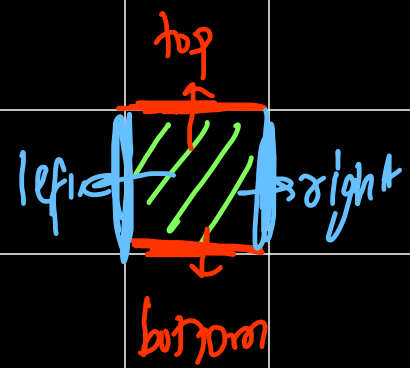
① start → end

② every free → exactly one

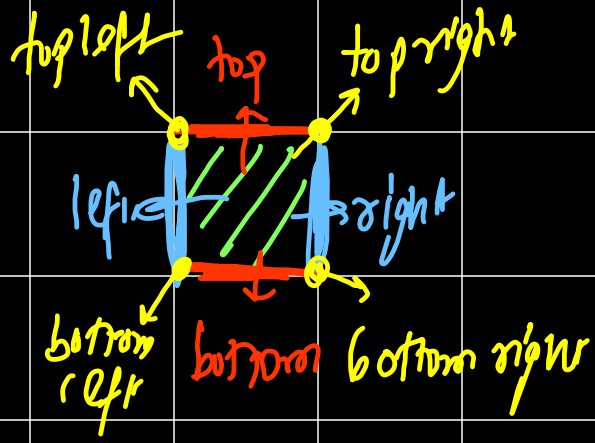
0 → free cell

-1 → blocked cell

4 directional neighbours (edge)



8 directional neighbours (edge + corner)



```java
int startRow = -1, startCol = -1;
int endRow = -1, endCol = -1;

public int countPaths(int[][] mat, int row, int col, int freeCells){
    if(row < 0 || col < 0 || row == mat.length || col == mat[0].length){
        // index out of bound -> negative base case
        return 0;
    }
    if(mat[row][col] == -2 || mat[row][col] == -1){
        // visited call or blocked cell -> negative base case
        return 0;
    }
    if(row == endRow && col == endCol){
        // destination -> positive base case
        if(freeCells == 0) return 1;
        else return 0;
    }

    mat[row][col] = -2; // visited mark

    int paths = 0;
    // top or up
    paths += countPaths(mat, row - 1, col, freeCells - 1);

    // down
    paths += countPaths(mat, row + 1, col, freeCells - 1);

    // left
    paths += countPaths(mat, row, col - 1, freeCells - 1);

    // right
    paths += countPaths(mat, row, col + 1, freeCells - 1);

    mat[row][col] = 0; // unvisited mark (backtracking)
    return paths;
}
```

$O(4^{mn})$ exponential

depth $\Rightarrow$ $m * n$

Calls $\Rightarrow$ 4

pre/post $\Rightarrow$ $O(1)$

```java
public int uniquePathsIII(int[][] grid) {
    int freeCells = 1; → starting cell == free cell

    for(int row = 0; row < grid.length; row++){
        for(int col = 0; col < grid[0].length; col++){
            if(grid[row][col] == 0){
                freeCells++;
            }
            else if(grid[row][col] == 1){
                startRow = row; startCol = col;
            }
            else if(grid[row][col] == 2){
                endRow = row; endCol = col;
            }
        }
    }

    return countPaths(grid, startRow, startCol, freeCells);
}
```

→ all free cells should be visited.

} starting & ending can be any cell

"Hard → Backtracking Problem"

$m \times n = 20 \times 20$

Combinations LC (77)

$n = 5$ $\quad \{1, 2, 3, 4, 5\}$

$k = 2$ $\quad {}^5C_2$

$\{\{1,2\}$ $\{2,3\}$ $\{3,4\}$ $\{4,5\}$

$\{1,3\}$ $\{2,4\}$ $\{3,5\}$

$\{1,4\}$ $\{2,5\}$

$\{1,5\}\}$

$k = 3$ $\quad {}^5C_3 = 10$

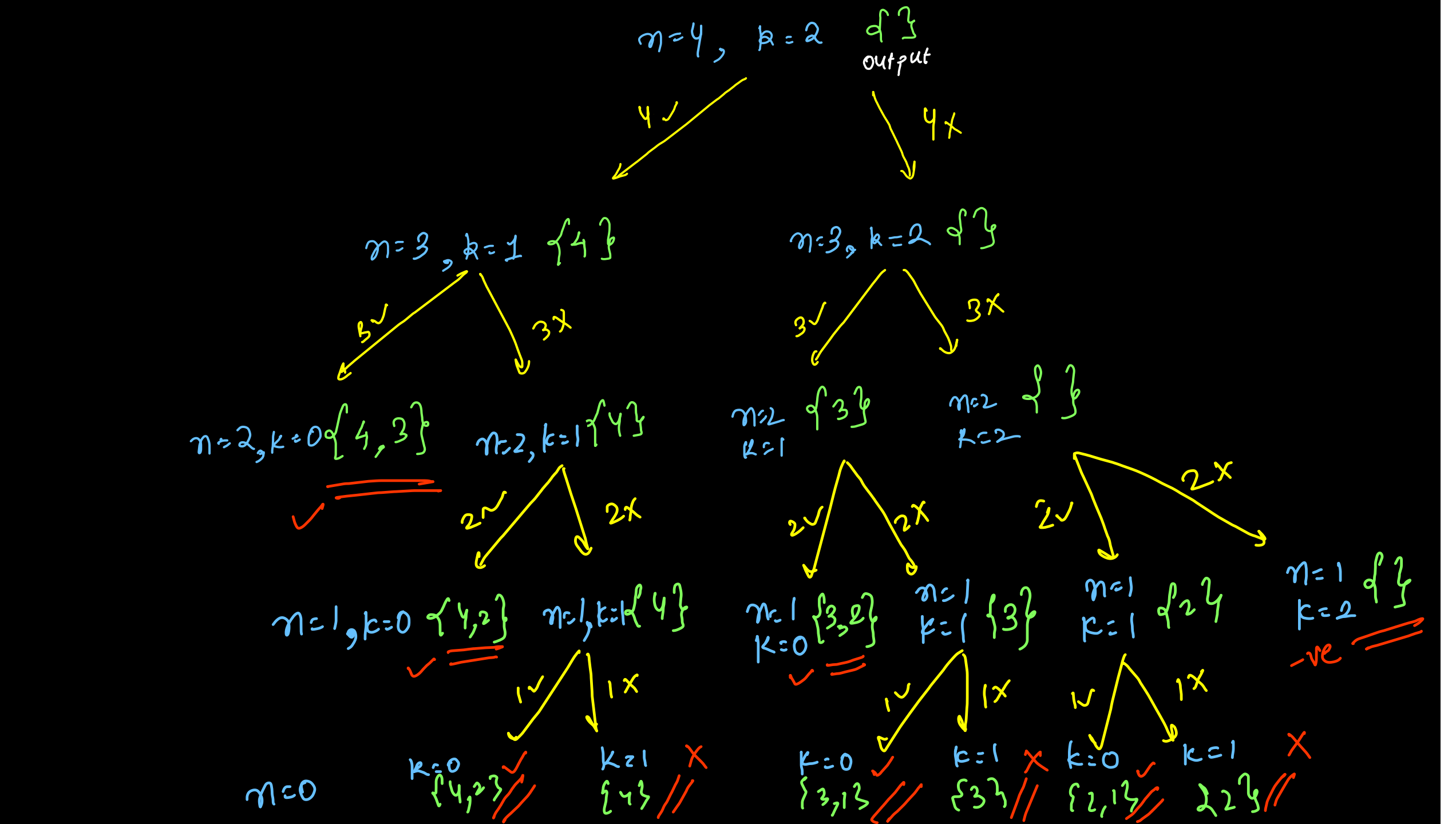
$\{\{1,2,3\}$ $\{1,3,4\}$

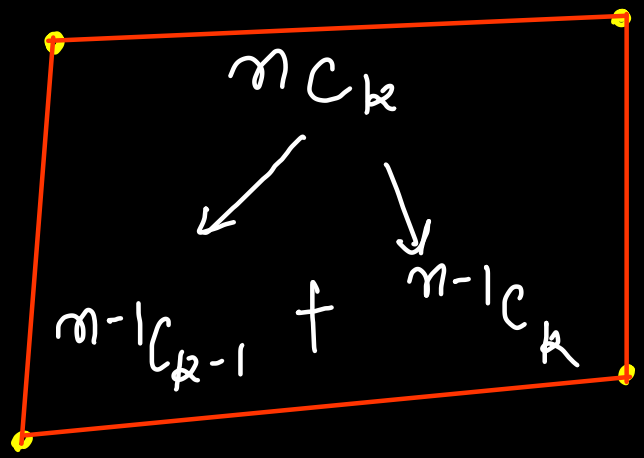
$\{1,2,4\}$ $\{1,3,5\}$

$\{1,2,5\}$ $\{1,4,5\}$

$\{2,3,4\}$ $\{3,4,5\}$

$\{2,3,5\}\}$


n=4, k=2 { }
output
4✓
4✗
n=3, k=1 {4}
n=3, k=2 { }
3✓
3✗
3✓
3✗
n=2, k=0 {4,3}
n=2, k=1 {4}
n=2 k=1 {3}
n=2 k=2 { }
2✓
2✗
2✓
2✗
2✓
2✗
n=1, k=0 {4,2}
n=1, k=1 {4}
n=1 k=0 {3,2}
n=1 k=1 {3}
n=1 k=1 {2}
n=1 k=2 { }
-ve
1✓
1✗
1✓
1✗
1✓
1✗
n=0
k=0 {4,2}
k=1 {4}
k=0 {3,1}
k=1 {3}
k=0 {2,1}
k=1 {2}
✗
✗
✗



calls = 2

depth = n

pre/post = n

$2^n + n * n$

Calls^{depth} + Work^{depth}

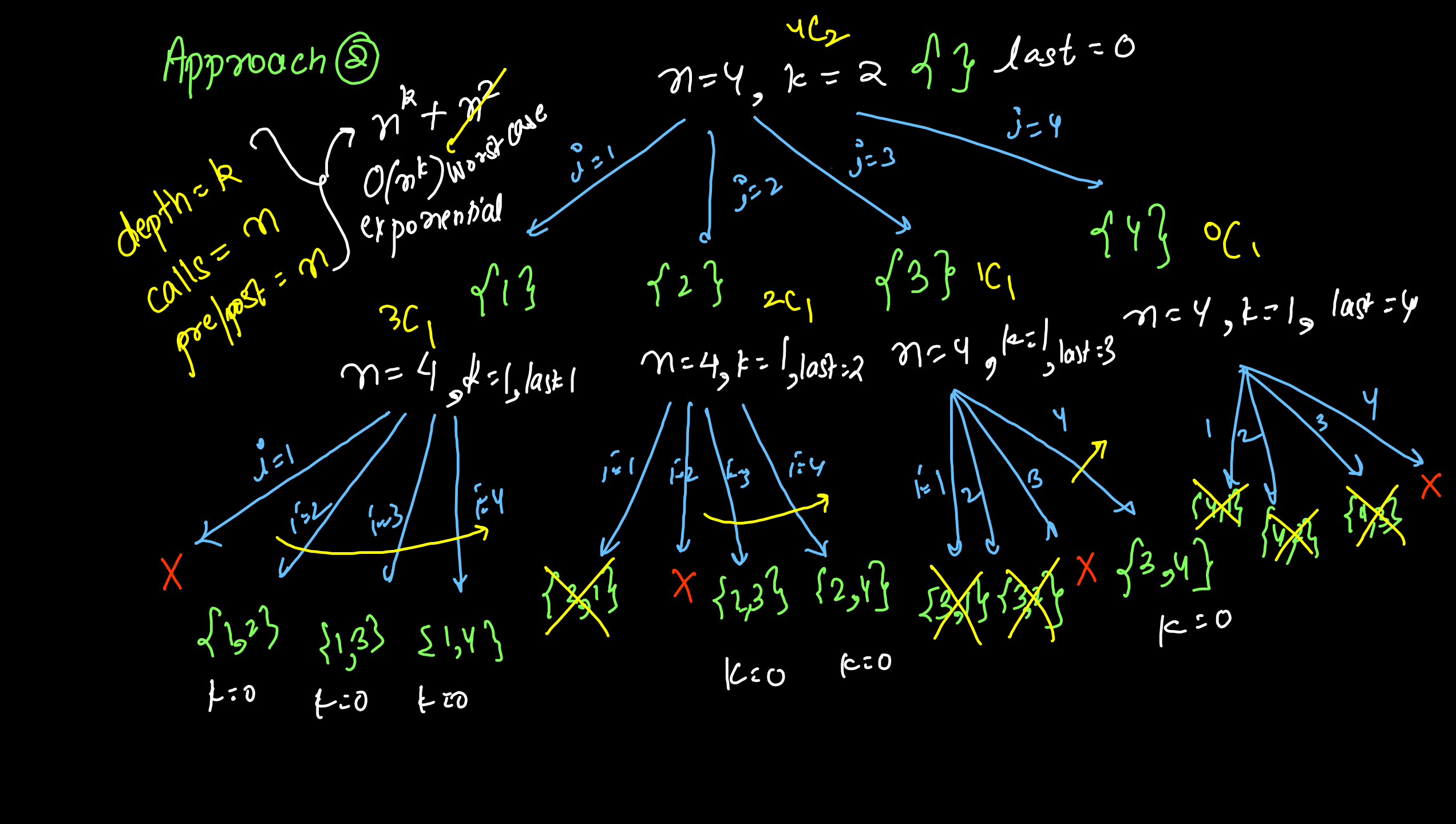
$O(2^n)$ exponential ⇒ time

$O(n)$ linear ⇒ space (recursion call stack)

```java
List<List<Integer>> ways;

public void printPaths(int n, int k, List<Integer> output){
    if(k == 0){
        ways.add(new ArrayList<>(output)); // deep copy
        return; // positive base case
    }
    if(n == 0 || k > n){
        return; // negative base case
    }

    // yes choice
    output.add(n);
    printPaths(n - 1, k - 1, output);
    output.remove(output.size() - 1); // backtracking

    // no choice
    printPaths(n - 1, k, output);
}

public List<List<Integer>> combine(int n, int k) {
    ways = new ArrayList<>();
    List<Integer> output = new ArrayList<>();
    printPaths(n, k, output);
    return ways;
}
```

deep copy ⇒ O(n)

App (1)


Approach ②
depth = k
calls = n
pre/post = n
$n^k + n^2$
$O(n^k)$ Worst case
exponential
$n=4, k=2$ { } last = 0
$^4C_2$
j=1
j=2
j=3
j=4
{1} $^3C_1$
{2} $^2C_1$
{3} $^1C_1$
{4} $^0C_1$
$n=4, k=1$, last=1
$n=4, k=1$, last=2
$n=4, k=1$, last=3
$n=4, k=1$, last=4
i=1
i=2
i=3
i=4
X
{1,2}
{1,3}
{1,4}
{2,3}
{2,4}
{3,4}
k=0

```java
List<List<Integer>> ways;

public void printPaths(int n, int k, List<Integer> output, int last){
    if(k == 0){
        ways.add(new ArrayList<>(output)); // deep copy
        return; // positive base case
    }

    for(int item = last + 1; item <= n; item++){
        output.add(item);
        printPaths(n, k - 1, output, item);
        output.remove(output.size() - 1); // backtracking
    }
}

public List<List<Integer>> combine(int n, int k) {
    ways = new ArrayList<>();
    List<Integer> output = new ArrayList<>();
    printPaths(n, k, output, 0);
    return ways;
}
```

→ loop for choices (which item to choose)

time = $O(n^k)$

Space = $O(k)$

$${}^nC_k = {}^1C_{k-1} + {}^2C_{k-1} + {}^3C_{k-1} + \cdots + {}^nC_{k-1}$$

Permutations LC 46

input: {1, 2, 3}

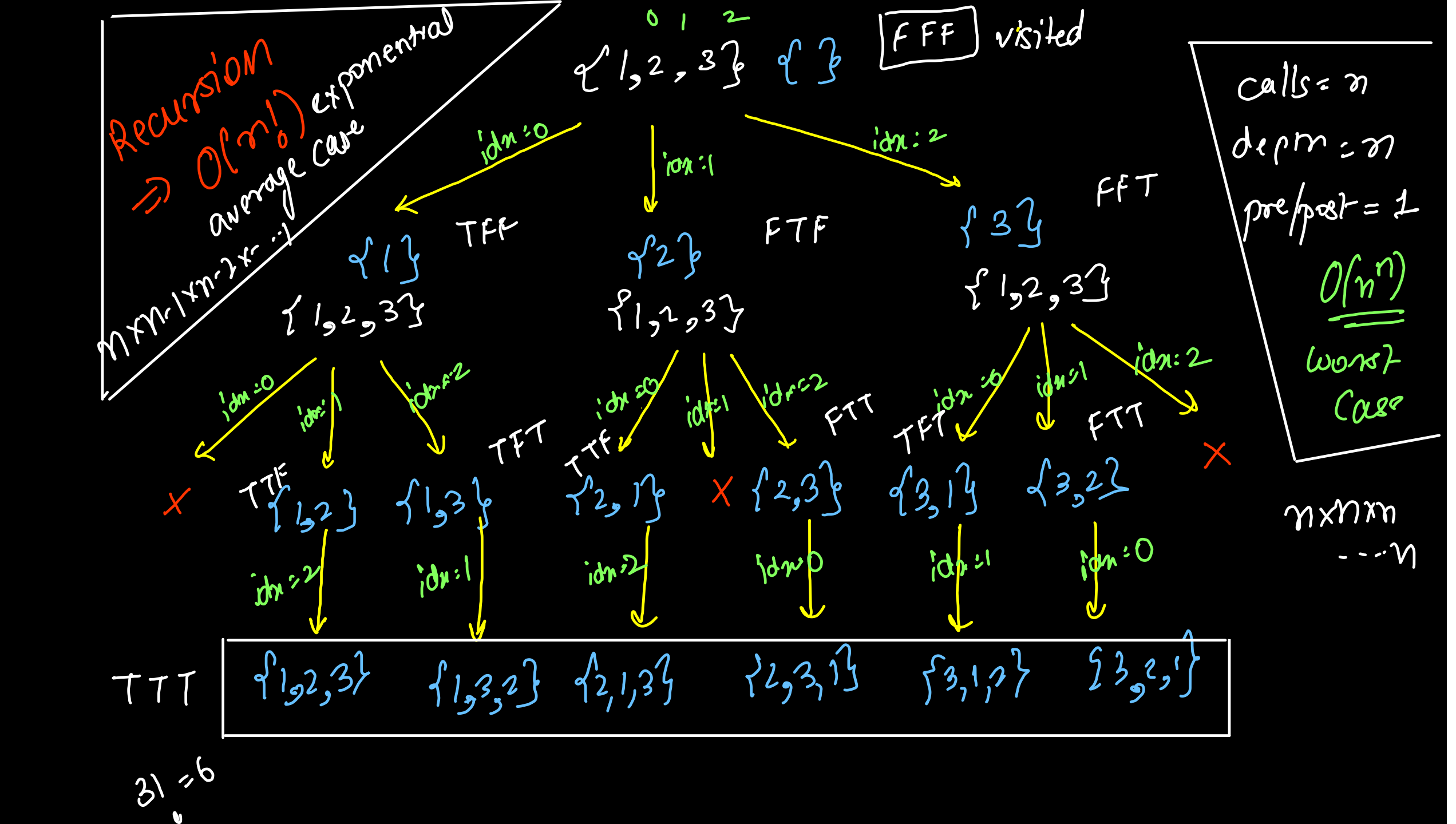
output: {
{1, 2, 3}
{1, 3, 2}
{2, 1, 3}
{2, 3, 1}
{3, 1, 2}
{3, 2, 1}
}

3! = 6

input: {0, 1}

output = {
{0, 1},
{1, 0}
}

2! = 2


Recursion
⇒ O(n!) exponential
average case
n x n-1 x n-2 x ... 1
0 1 2
{1,2,3} { } FFF visited
idx:0
idx:1
idx:2
{1} TFF {1,2,3}
{2} FTF {1,2,3}
{3} FFT {1,2,3}
TTF {1,2}
TFT {1,3}
TTF {2,1}
FTT {2,3}
TFT {3,1}
FTT {3,2}
TTT {1,2,3} {1,3,2} {2,1,3} {2,3,1} {3,1,2} {3,2,1}
calls = n
depth = n
pre/post = 1
O(n^n)
Worst Case
n x n x n ... n
31 = 6

```java
List<List<Integer>> ways;

public void printPaths(int[] nums, List<Integer> output, boolean[] vis){
    if(output.size() == nums.length){
        ways.add(new ArrayList<>(output)); // deep copy
        return; // positive base case
    }

    for(int idx = 0; idx < nums.length; idx++){
        if(vis[idx] == true) continue; // duplicate items

        output.add(nums[idx]);
        vis[idx] = true;

        printPaths(nums, output, vis);

        output.remove(output.size() - 1); // backtracking
        vis[idx] = false;
    }
}

public List<List<Integer>> permute(int[] nums) {
    boolean[] vis = new boolean[nums.length];
    ways = new ArrayList<>();
    List<Integer> output = new ArrayList<>();

    printPaths(nums, output, vis);
    return ways;
}
```

Code 1

#approach ①

Which item to chose?

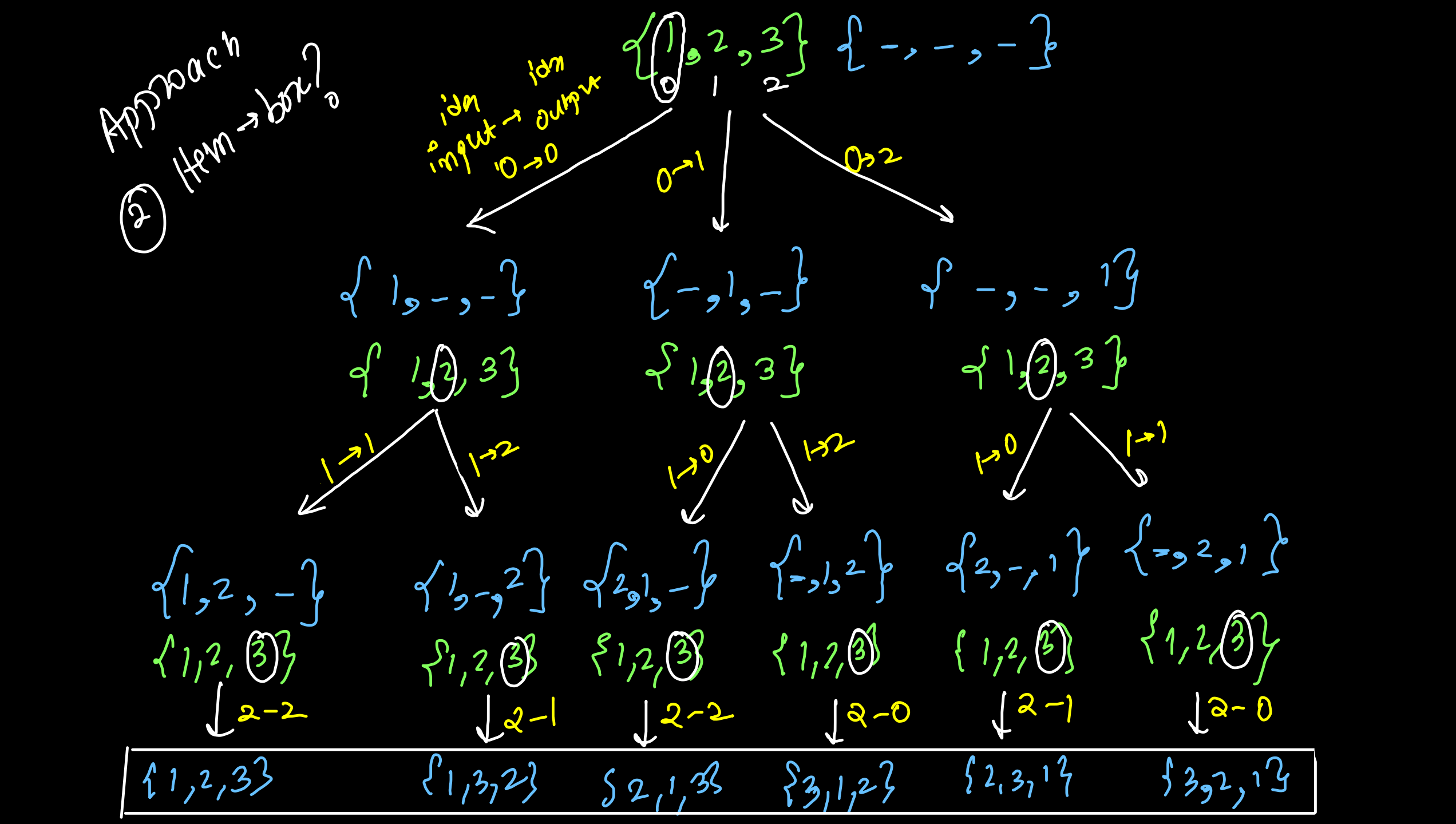
Time ⇒ O(n!) {expo}

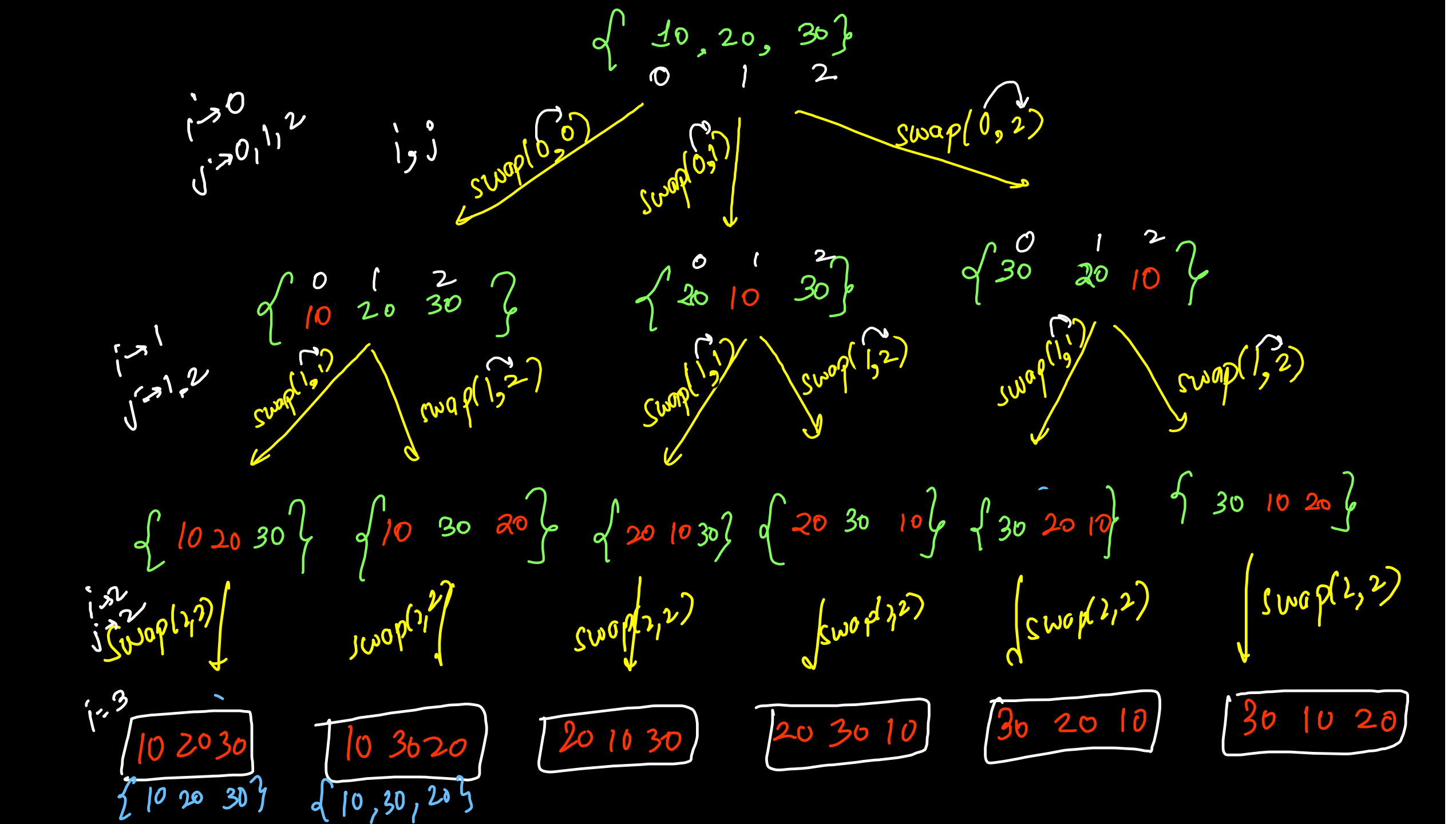
Space ⇒ O(n) {linear}

Code 2

```java
List<List<Integer>> ways;

public void printPaths(int[] nums, List<Integer> output){
    if(output.size() == nums.length){
        ways.add(new ArrayList<>(output)); // deep copy
        return; // positive base case
    }

    for(int idx = 0; idx < nums.length; idx++){
        if(output.contains(nums[idx]) == true) continue;
        output.add(nums[idx]);
        printPaths(nums, output);
        output.remove(output.size() - 1); // backtracking
    }
}

public List<List<Integer>> permute(int[] nums) {
    ways = new ArrayList<>();
    List<Integer> output = new ArrayList<>();
    printPaths(nums, output);
    return ways;
}
```

linear search


Approach
② Perm → box
idx
input → output
0 → 0
0 → 1
0 → 2
{1,2,3}
{-,-,-}
{1,-,-}
{-,1,-}
{-,-,1}
1 → 1
1 → 2
1 → 0
1 → 2
1 → 0
1 → 1
{1,2,-}
{1,-,2}
{2,1,-}
{-,1,2}
{2,-,1}
{-,2,1}
2-2
2-1
2-2
2-0
2-1
2-0
{1,2,3}
{1,3,2}
{2,1,3}
{3,1,2}
{2,3,1}
{3,2,1}

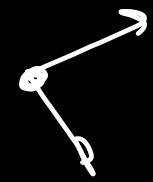

{ 10, 20, 30 }
0 1 2
i -> 0
j -> 0, 1, 2
i, j
swap(0,0)
swap(0,1)
swap(0,2)
{ 10 20 30 }
{ 20 10 30 }
{ 30 20 10 }
i -> 1
j -> 1, 2
swap(1,1)
swap(1,2)
{ 10 20 30 }
{ 10 30 20 }
{ 20 10 30 }
{ 20 30 10 }
{ 30 20 10 }
{ 30 10 20 }
i -> 2
j -> 2
swap(2,2)
i -> 3
10 20 30
10 30 20
20 10 30
20 30 10
30 20 10
30 10 20
{ 10 20 30 }
{ 10, 30, 20 }

```java
List<List<Integer>> ways;

public void swap(int[] nums, int i, int j){
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}

public void printPaths(int[] nums, int i){
    if(i == nums.length){
        List<Integer> output = new ArrayList<>();
        for(int val: nums) output.add(val);
        ways.add(output);
        return;
    }

    for(int j = i; j < nums.length; j++){
        swap(nums, i, j);
        printPaths(nums, i + 1);
        swap(nums, i, j); // backtracking
    }
}

public List<List<Integer>> permute(int[] nums) {
    ways = new ArrayList<>();
    printPaths(nums, 0);
    return ways;
}
```

(input) current idx

array → AL

desired idx (output - idx)



Time ⇒ O(n!) avg

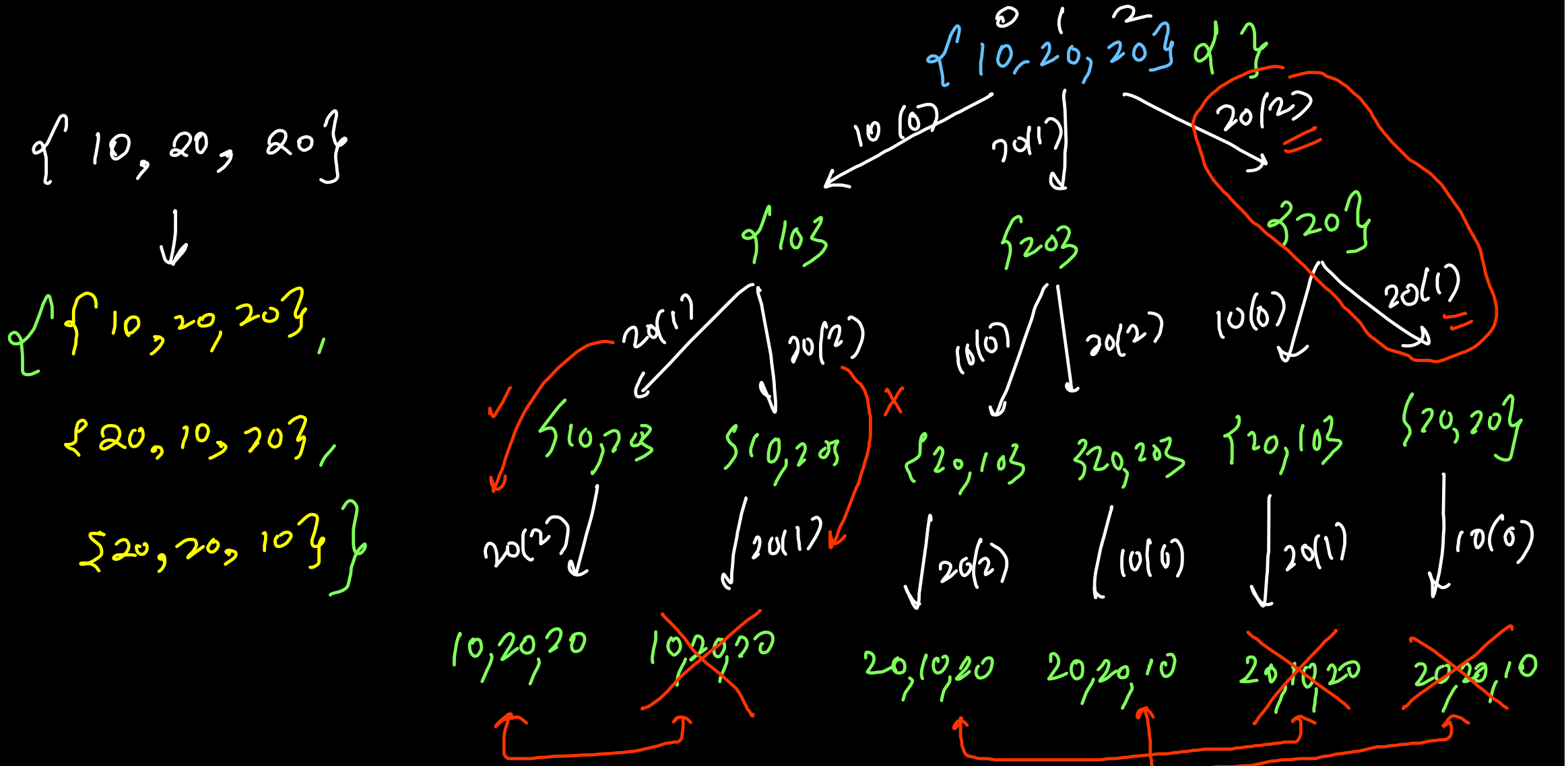
Space ⇒ O(n) linear

approach ②

Item → Which idx will it go at?

#HW # Distinct Permutations

LC 47



Combination Sum / Coin change

lc 40 { Finite Supply }

distinct combination

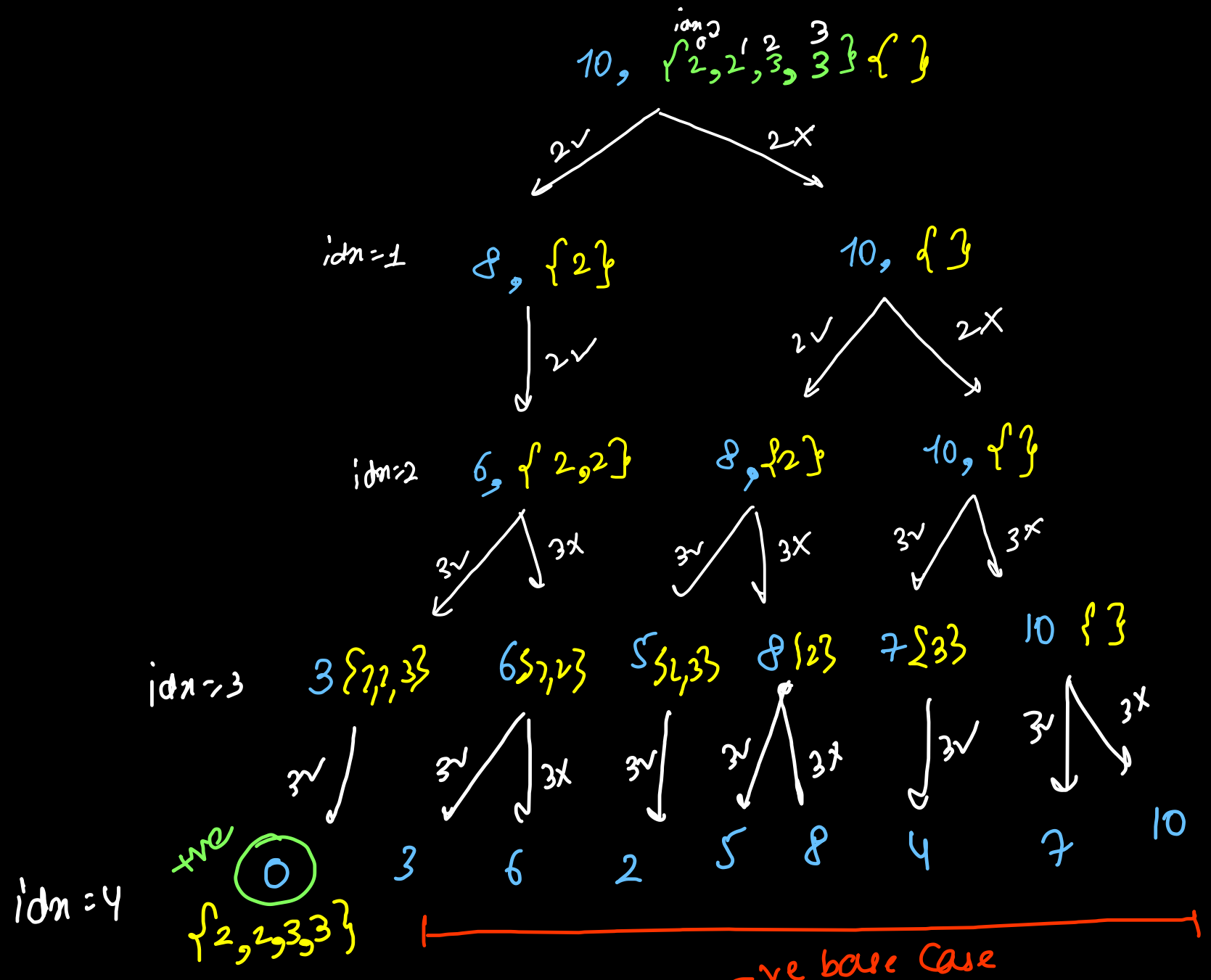
target = 10

coins = { 2, 2, 3, 3, 5, 5 }

1) { 2, 3, 5 }

2) { 5, 5 }

3) { 2, 2, 3, 3 }


idx=0
10, {2,2,3,3} {}
2✓
2✗
idx=1
8, {2}
10, {}
2✓
2✓
2✗
idx=2
6, {2,2}
8, {2}
10, {}
3✓
3✗
3✓
3✗
3✓
3✗
idx=3
3 {2,2,3}
6 {2,2}
5 {2,3}
8 {2}
7 {3}
10 {}
3✓
3✓
3✗
3✓
3✗
3✓
3✓
3✗
idx=4
+ve
0
{2,2,3,3}
3
6
2
5
8
4
7
10
-ve base case

```java
List<List<Integer>> ways;

public void printWays(int[] coins, int idx, int target, List<Integer> output){
    if(idx == coins.length){
        if(target == 0)
            ways.add(new ArrayList<>(output));
        return;
    }

    // yes
    output.add(coins[idx]);
    printWays(coins, idx + 1, target - coins[idx], output);
    output.remove(output.size() - 1);

    // no
    if(output.size() > 0 && output.get(output.size() - 1) == coins[idx])
        return; // duplicate combinations discarded
    printWays(coins, idx + 1, target, output);
}

public List<List<Integer>> combinationSum2(int[] coins, int target) {
    Arrays.sort(coins); // to remove duplicate combinations
    ways = new ArrayList<>();
    printWays(coins, 0, target, new ArrayList<>());
    return ways;
}
```

calls = 2

depth = n

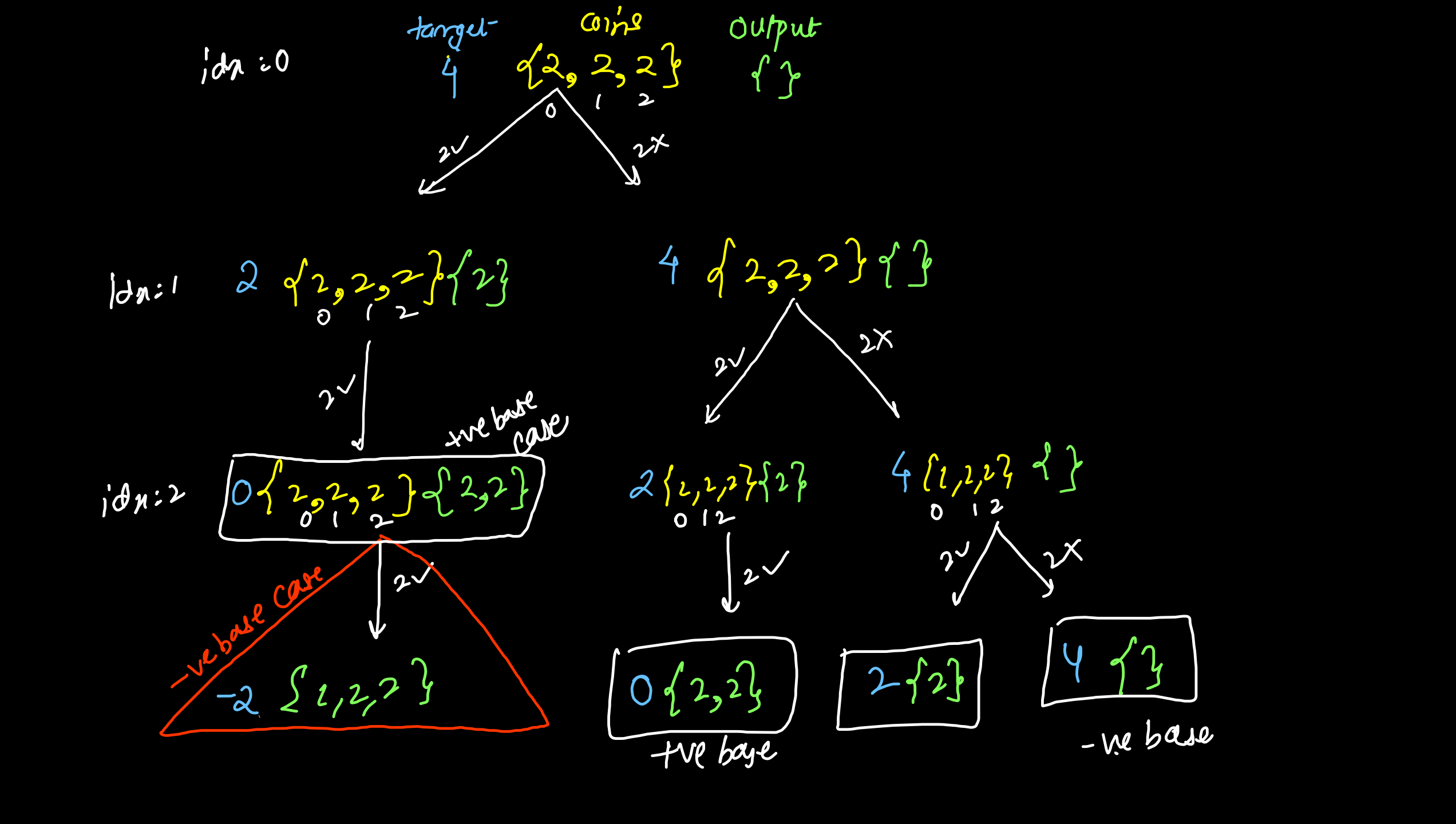
pre/post = 1

time : $2^n + 1 \times n \Rightarrow O(2^n)$

exponential

$N_{max} : 30$

w/o pruning $\Rightarrow$ TLE


idx :0
target-
4
coins
{2, 2, 2}
0 1 2
output
{}
2✓
2x
idx:1
2 {2, 2, 2}{2}
0 1 2
2✓
+ve base case
idx:2
0{2, 2, 2}{2, 2}
0 1 2
-ve base case
2✓
-2 {1, 2, 2}
4 {2, 2, 2}{}
2✓
2x
2{1, 2, 2}{2}
0 1 2
2✓
0{2, 2}
+ve base
4{1, 2, 2}{}
0 1 2
2✓
2x
2{2}
4{}
-ve base

```java
List<List<Integer>> ways;

public void printWays(int[] coins, int idx, int target, List<Integer> output){
    if(target < 0) return; // negative base case: pruning
    if(idx == coins.length){
        if(target == 0)
            ways.add(new ArrayList<>(output));
        return;
    }

    // yes
    output.add(coins[idx]);
    printWays(coins, idx + 1, target - coins[idx], output);
    output.remove(output.size() - 1);

    // no
    if(output.size() > 0 && output.get(output.size() - 1) == coins[idx])
        return; // duplicate combinations discarded
    printWays(coins, idx + 1, target, output);
}

public List<List<Integer>> combinationSum2(int[] coins, int target) {
    Arrays.sort(coins); // to remove duplicate combinations
    ways = new ArrayList<>();
    printWays(coins, 0, target, new ArrayList<>());
    return ways;
}
```

= * (accepted)

Time = 2^N

Space = N

→ duplicate adjacent

Combination Sum - I

Coin change (Infinite Supply)

coins = {2, 3, 6, 7} target = 7

answer : 1) {2, 2, 3}

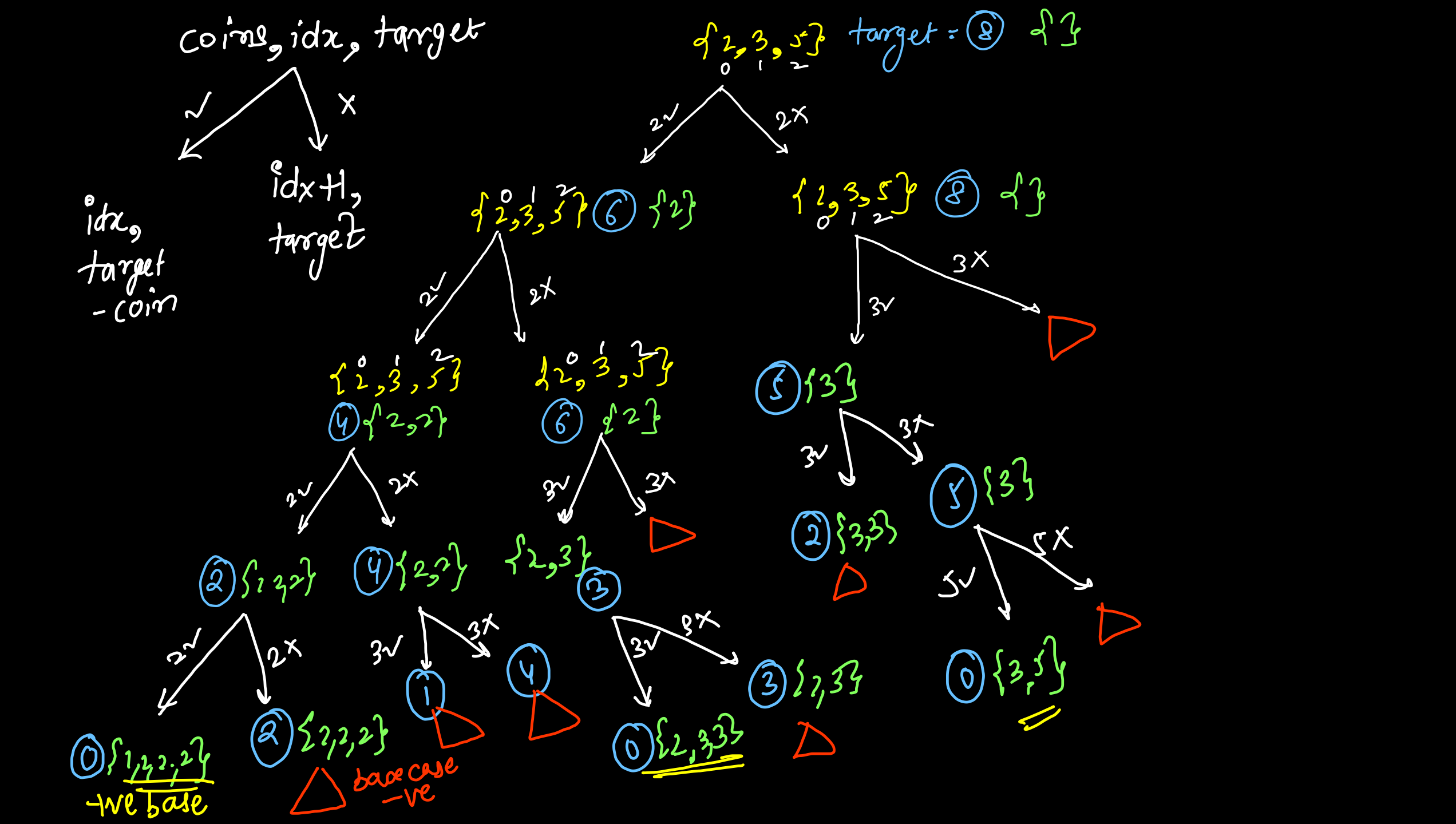
2) {7}

coins = {2, 3, 5} target = 8

answer = 1) {2, 3, 3}

2) {2, 2, 2, 2}

3) {3, 5}


coins, idx, target
✓
x
idx+1, target
idx, target - coin
{2,3,5} target = 8 {}
2✓
2x
{2,3,5} 6 {2}
{1,3,5} 8 {}
3✓
3x
{2,3,5} {2,2}
{2,3,5} 6 {2}
5 {3}
2 {3,3}
1 {3}
{2,3}
3 {1,3}
0 {3,5}
0 {1,2,2,2}
-ve base
2 {2,2,2}
base case -ve
0 {2,3,3}

```java
List<List<Integer>> ways;

public void printWays(int[] coins, int idx, int target, List<Integer> output){
    if(target < 0) return; // negative base case: pruning
    if(idx == coins.length){
        if(target == 0)
            ways.add(new ArrayList<>(output));
        return;
    }

    // yes
    output.add(coins[idx]);
    printWays(coins, idx, target - coins[idx], output);
    output.remove(output.size() - 1);

    // no
    printWays(coins, idx + 1, target, output);
}

public List<List<Integer>> combinationSum(int[] coins, int target) {
    ways = new ArrayList<>();
    printWays(coins, 0, target, new ArrayList<>());
    return ways;
}
```

If yes choice, then again same idx (Infinite Supply)

Time = $O(2^N)$

Space = $O(N)$

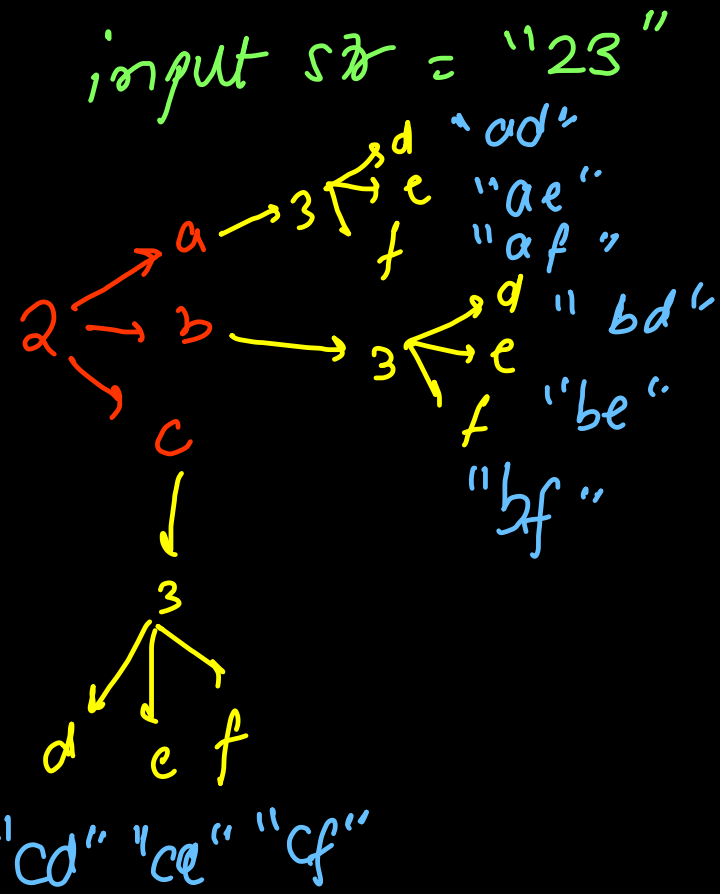
Recursion on Puzzles

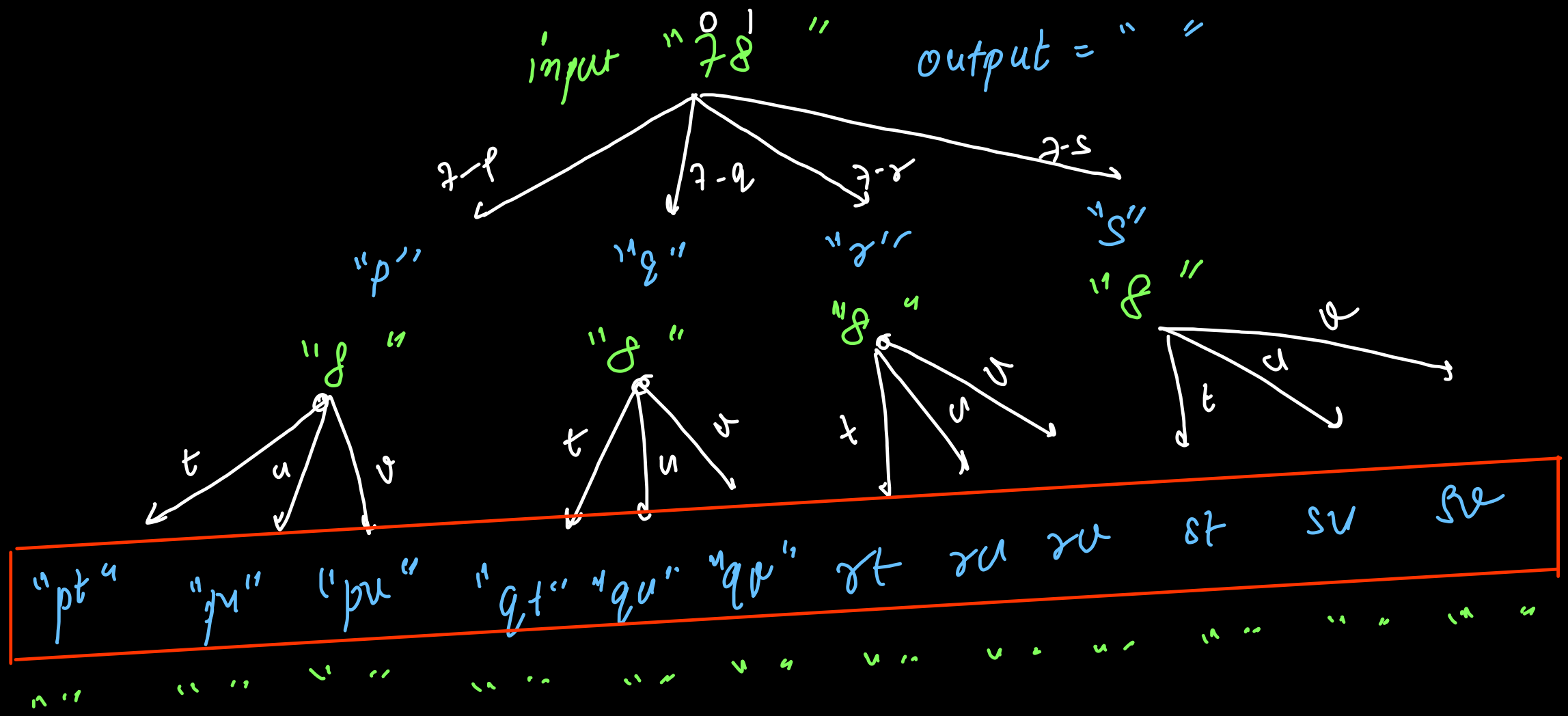
Keypad Problems

LC 17) Letter Combinations

mapping

input str = "23"





(48)

'0' → 0

-'0' (48)

'1' → 1

-'0' (48)

'7' = 7

-'0' (48)

```java
String[] map = {"", "", "abc", "def", "ghi",
    "jkl", "mno", "pqrs", "tuv", "wxyz"};
List<String> ways;

public void printPaths(String input, int idx, String output){
    if(idx == input.length()){
        ways.add(output);
        return;
    }

    int digit = input.charAt(idx) - '0';

    for(char letter : map[digit].toCharArray())
        printPaths(input, idx + 1, output + letter);
}

public List<String> letterCombinations(String digits) {
    ways = new ArrayList<>();
    if(digits.length() == 0) return ways;

    printPaths(digits, 0, "");
    return ways;
}
```

0 1 2 3 4

5 6 7 8 9

poe = n

calls = 4

depth: n

$O(4^N + n \times n)$

expo

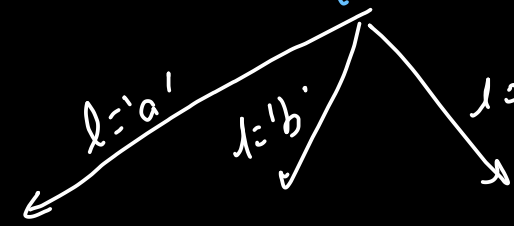
input "23" output " "

idx

digit = '2' - '0' = 50 - 48 = 2

map[2] = "abc"

=> {'a', 'b', 'c'}

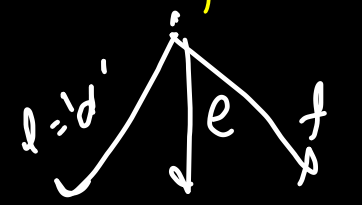


"23", "a"

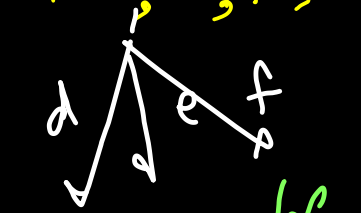
idx

digit = '3' - '0' = 3

map(3) = "def" => {'d', 'e', 'f'}



"23", "b"



"23", "c"

list of "ad" "ae" af bd be bf cd ce cf

Decode Ways {LC 91}

(65) ← +64
'A' → "1"
(66) ← +64
'B' → "2"
(67) ← +64
'C' → "3"

⋮

'X' → "24"
'Y' → "25"
'Z' → "26"
(90) ← +64

"2216"

22|1|6
{"VAF"

2|21|6
"BUF"

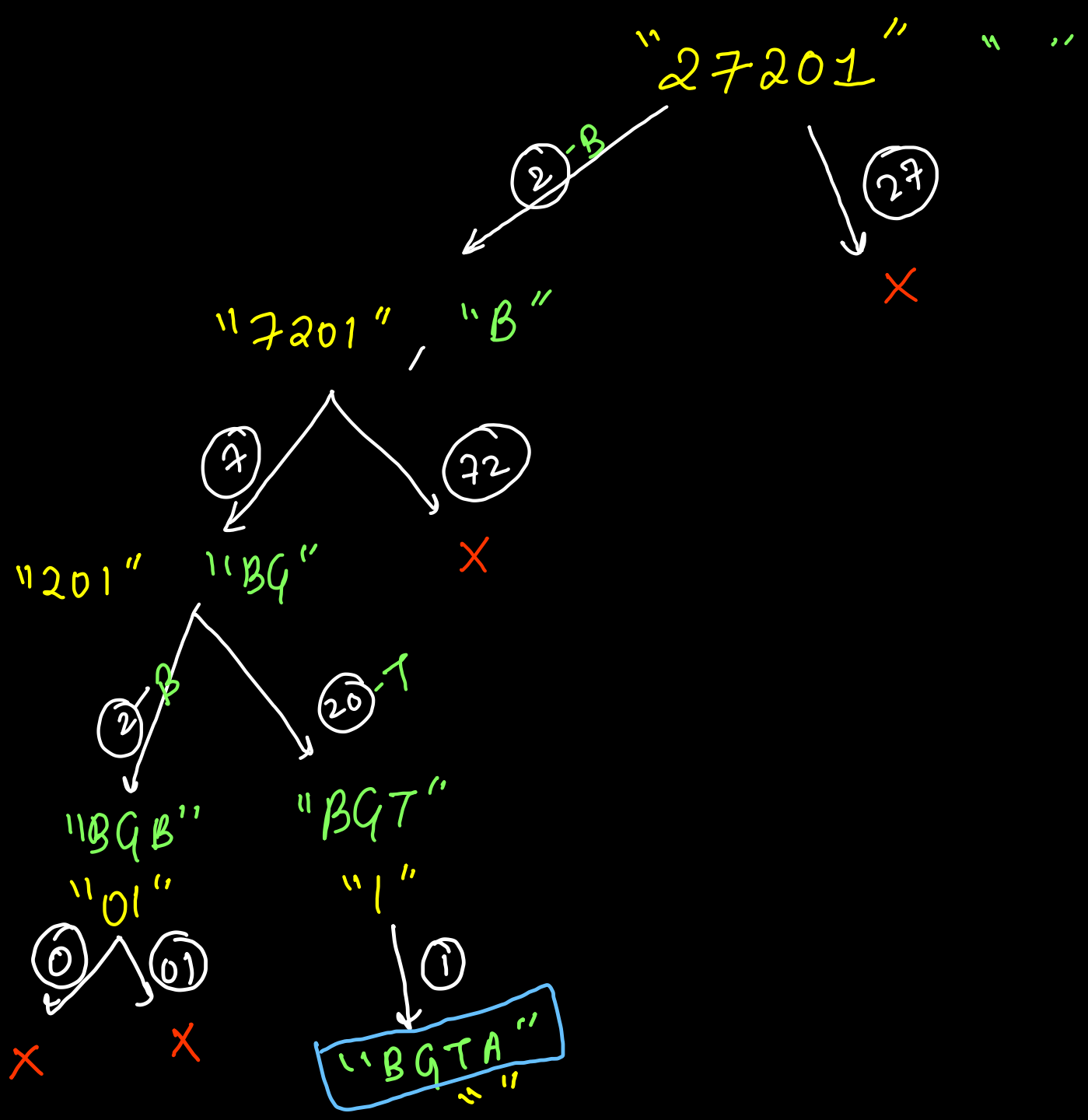
2|2|1|6
"BBAF"

22|16
"VP"}

2|2|16
"BBF"

A B C D E F G H I J K L M
1 2 3 4 5 6 7 8 9 10 11 12 13

N O P Q R S T U V W X Y Z
14 15 16 17 18 19 20 21 22 23 24 25 26


"27201" " "
2 - B
27
"7201" , "B"
7
72
"201" "BG"
2 - B
20 - T
"BGB"
"01"
"BGT"
"1"
0
01
1
"BGTA"
" "

```java
public int printWays(String input, int idx, String output){
    if(idx == input.length()){
        System.out.println(output);
        return 1;
    }

    int digit = input.charAt(idx) - '0';
    char letter = (char)(digit + 64);
    if(digit == 0) return 0;

    int ans = printWays(input, idx + 1, output + letter);

    if(idx + 1 < input.length()){
        digit = (input.charAt(idx) - '0') * 10 + (input.charAt(idx + 1) - '0');
        letter = (char)(digit + 64);
        if(digit >= 10 && digit <= 26)
            ans += printWays(input, idx + 2, output + letter);
    }

    return ans;
}

public int numDecodings(String s) {
    return printWays(s, 0, "");
}
```

'1' - '9'

'10' - '26'

Time => $O(2^N)$

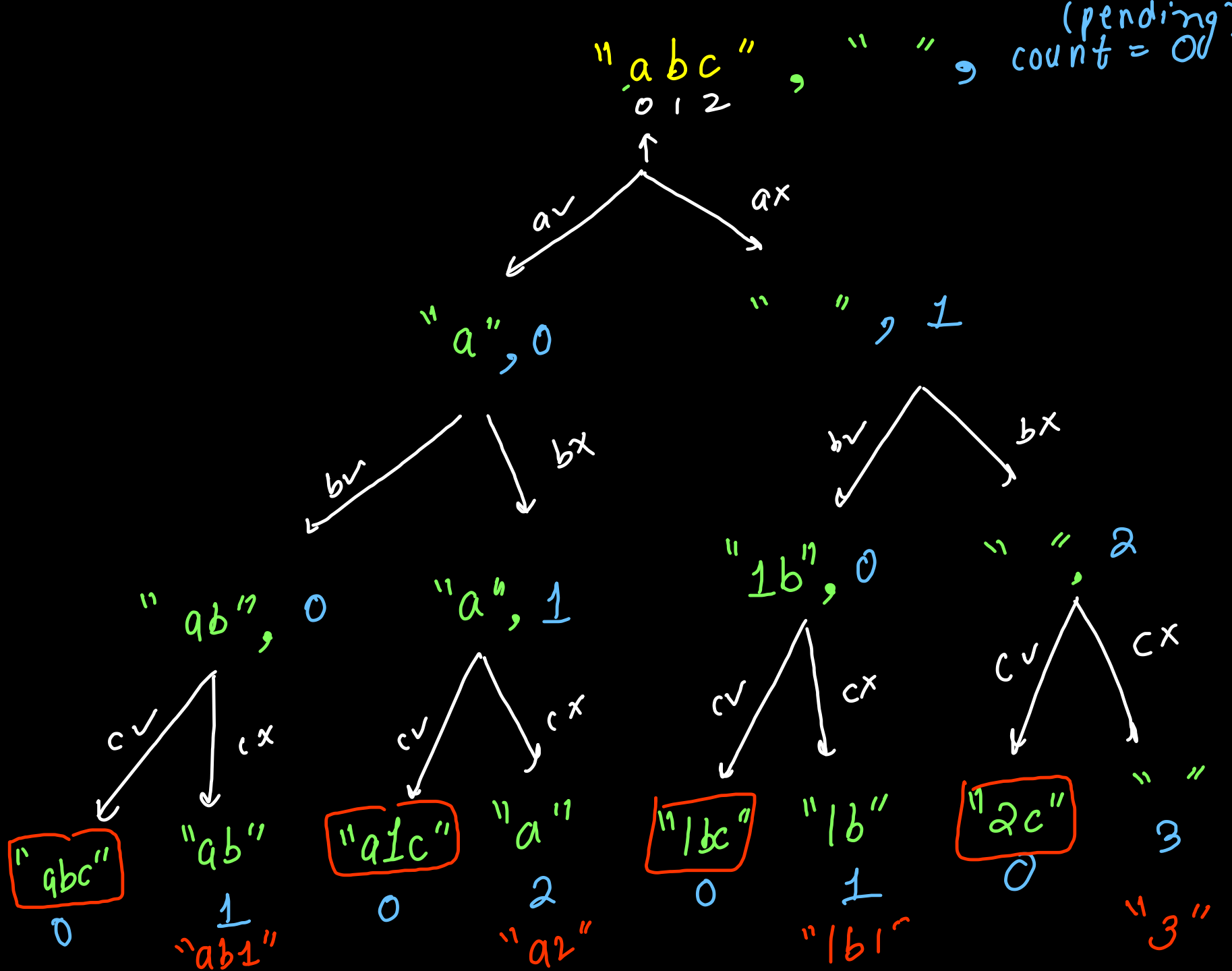
Space => $O(N)$

decode ways on leetcode (TLE)

Generalized Abbreviation.

"abc"

{"abc", "1bc", "a1c", "ab1"
"2c", "a2", "3", "1b1"}


"abc", "", count = 0
(pending)
0 1 2
av
ax
"a", 0
"", 1
bv
bx
bv
bx
"ab", 0
"a", 1
"1b", 0
"", 2
cv
cx
cv
cx
cv
cx
cv
cx
"abc"
0
"ab"
1
"ab1"
"a1c"
0
"a"
2
"a2"
"1bc"
0
"1b"
1
"1b1"
"2c"
0
""
3
"3"

LC locked (Google)

Generalized Abbreviations

```
static ArrayList<String> ways;

0 references
public static void printPaths(String input, int idx, String output, int count){
    if(idx == input.length()){
        // base case
        output += (count > 0) ? ("" + count) : "";
        ways.add(output);
        return;
    }

    // no
    printPaths(input, idx + 1, output, count + 1);

    // yes
    char ch = input.charAt(idx);
    output += ((count > 0) ? ("" + count) : "") + ch;
    printPaths(input, idx + 1, output, 0);
}
0 references
public static ArrayList < String > findAbbr(String str) {
    ways = new ArrayList<>();
    printPaths(str, 0, "", 0);
    Collections.sort(ways);
    return ways;
}
```

calls = 2

depth = N

pre/post = N

calls^{depth} + work x depth

time : $O(2^N + N \times N)$

=> exponential

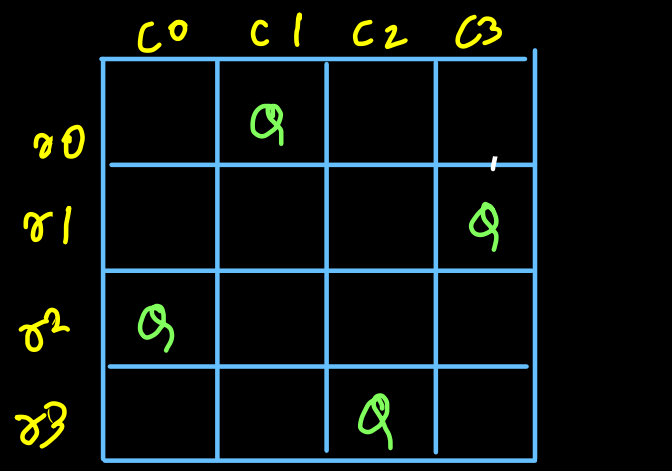
space = recursion call stack

$O(N)$

N Queens (VVIMP)

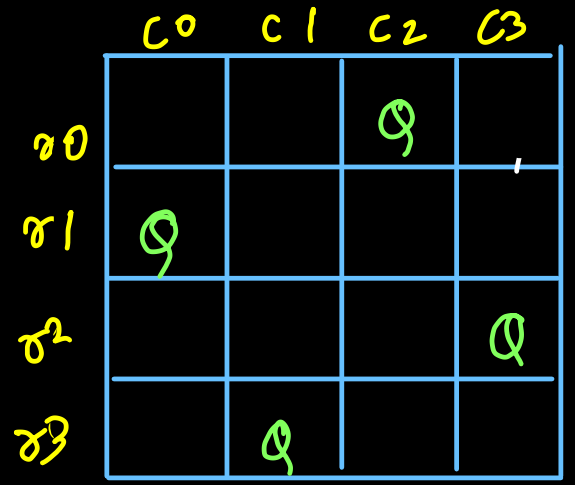
chess (queen)

n queens ⇒ such that no two queens can kill each other.

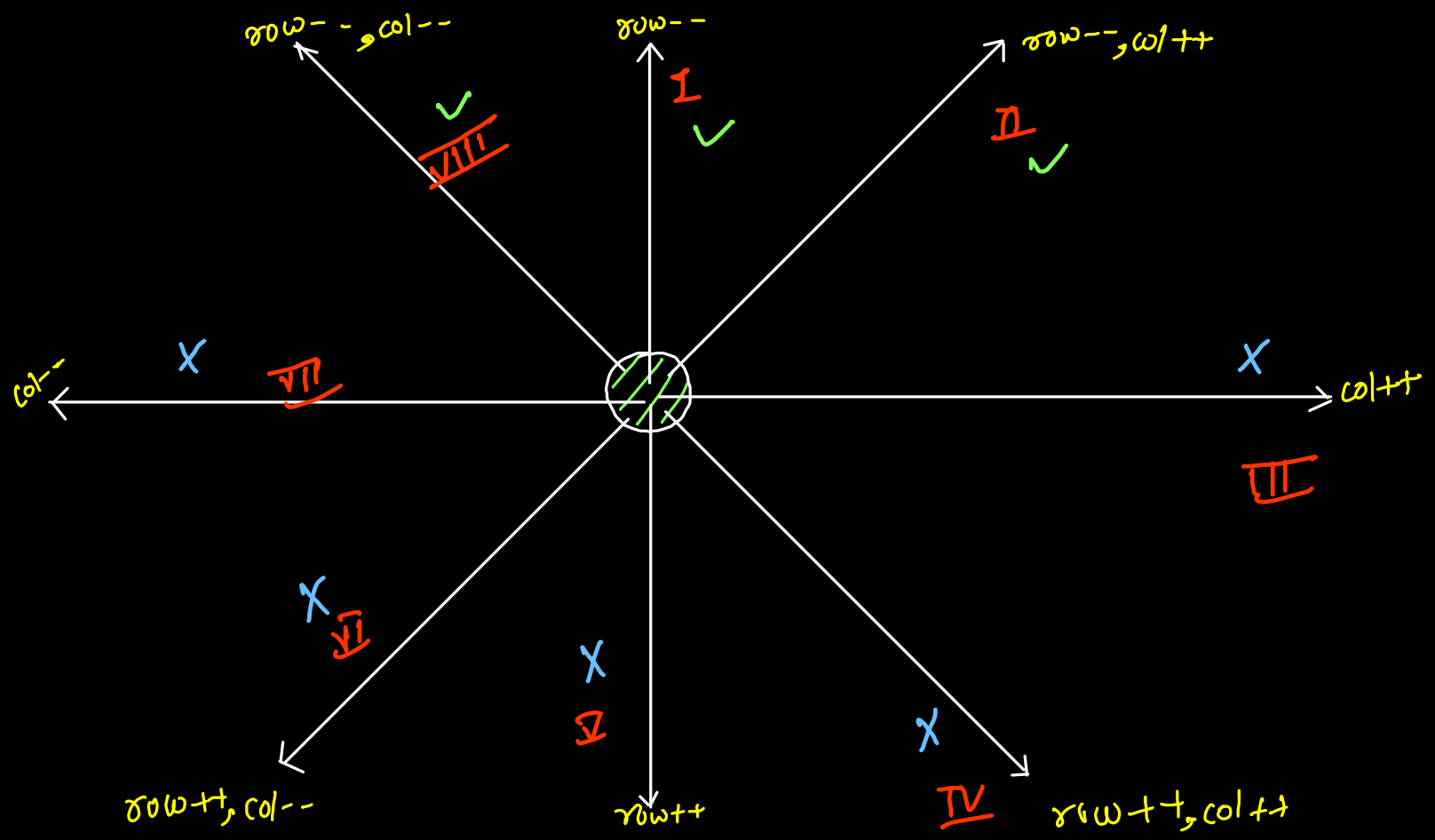


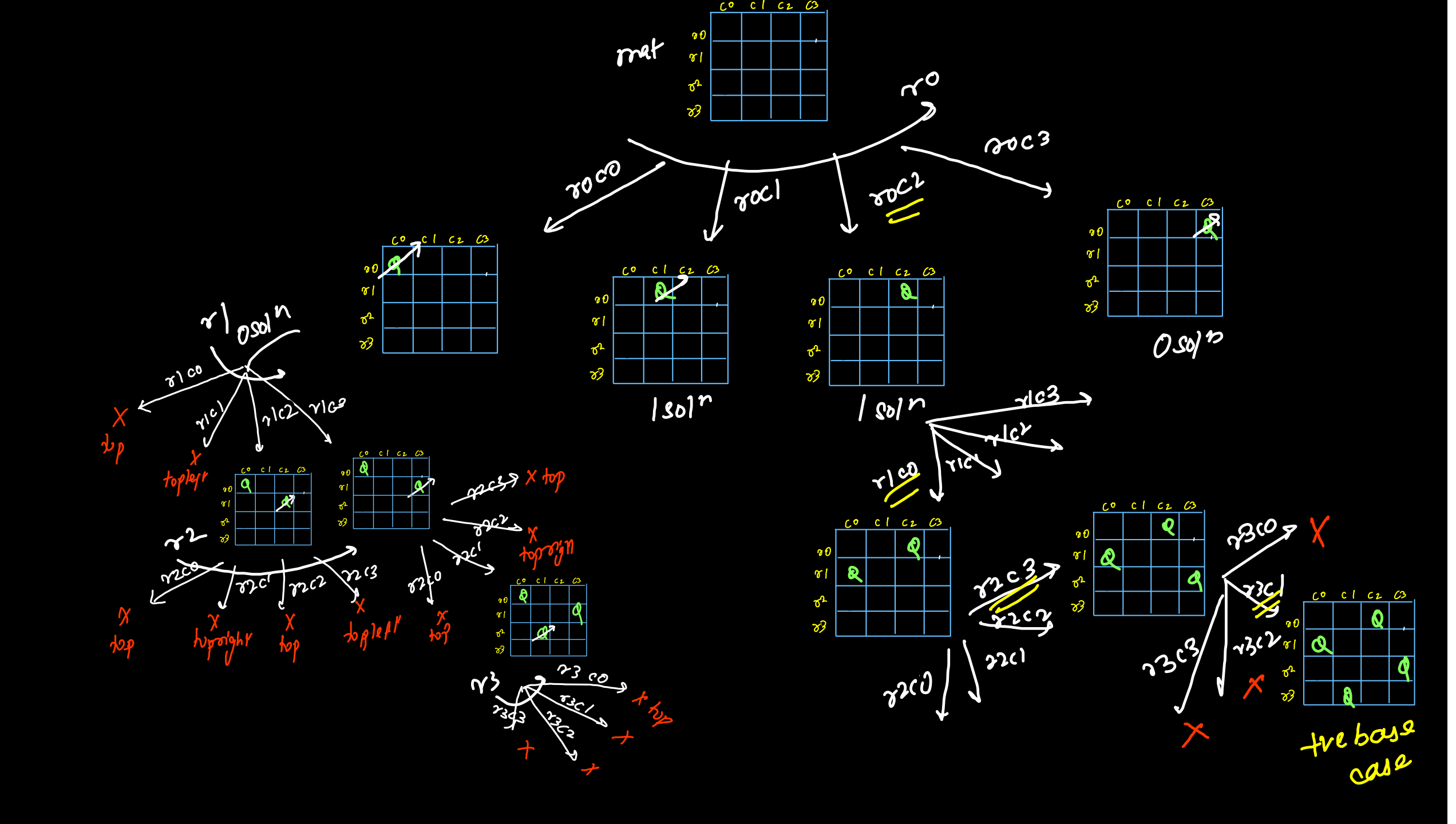
4x4

n=4

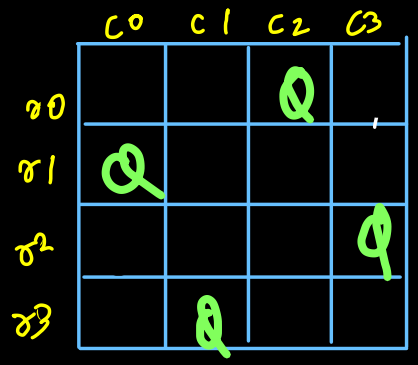


4x4


row--, col--
row--
row--, col++
col--
col++
row++, col--
row++
row++, col++
I
II
III
IV
V
VI
VII
VIII


mat
r0
r0c0
r0c1
r0c2
r0c3
0 soln
1 soln
1 soln
r1
1 soln
r1c0
r1c1
r1c2
r1c3
X
top
X
topleft
r2
r2c0
r2c1
r2c2
r2c3
X top
X
topright
X
top
X
topleft
X
top
r3
r3c0
r3c1
r3c2
r3c3
+
top
+
+
+
r1c3
r1c2
r1c1
r1c0
r2c3
r2c2
r2c1
r2c0
r3c0
X
r3c1
r3c2
X
r3c3
X
+ve base case

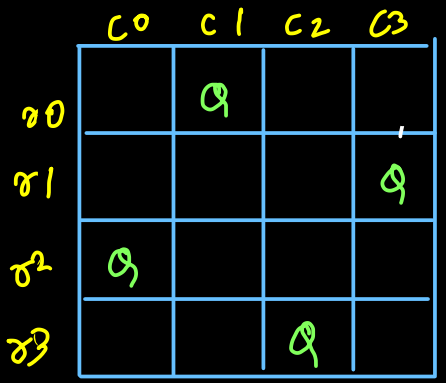
2d list → [



① ⟹ ["c0 c1 c2 c3 . . Q . " , "c0 c1 c2 c3 Q . . . " , "c0 c1 c2 c3 . . . Q " , "c0 c1 c2 c3 . Q . . "]

r0 r1 r2 r3

,



② ⟹ [" . Q . . " , " . . . Q " , " Q . . . " , " . . Q . "]

4 x 4

]

calls = N (cols), depth = N (rows), pre/post = N

time = calls^depth + work × depth

= $O(N^N)$ exponential

$n_{max} = 9$

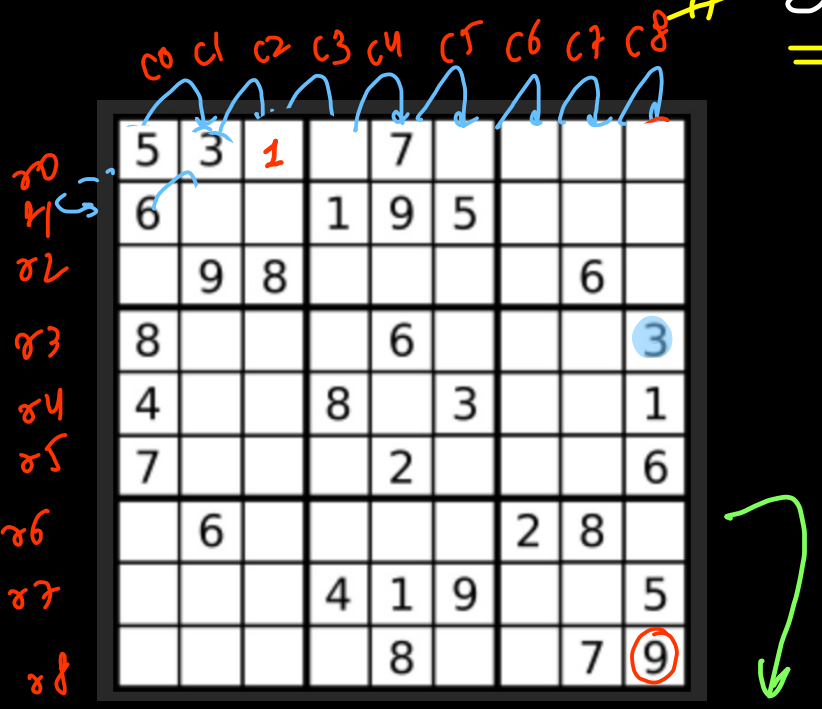
```java
List<List<String>> ways;

public void nQueens(boolean[][] chess, int row, int n){
    if(row == n){
        addOutput(chess); // boolean[][] -> List<String>
        return;
    }

    // choices or recursive call
    for(int col = 0; col < n; col++){
        if(isQueenSafe(chess, row, col) == false) continue; // pruning (invalid calls)

        chess[row][col] = true; // visited
        nQueens(chess, row + 1, n);
        chess[row][col] = false; // unvisited
    }
}

public List<List<String>> solveNQueens(int n) {
    ways = new ArrayList<>();
    boolean[][] chess = new boolean[n][n]; // -> false
    nQueens(chess, 0, n);
    return ways;
}
```

```java
public void addOutput(boolean[][] chess, int n){
    List<String> output = new ArrayList<>();

    for(int row = 0; row < n; row++){
        String str = "";
        for(int col = 0; col < n; col++){
            if(chess[row][col] == true){
                str += 'Q';
            } else {
                str += '.';
            }
        }
        output.add(str);
    }

    ways.add(output);
}
```

```java
public boolean isQueenSafe(boolean[][] chess, int row, int col, int n){
    // top
    for(int r = row; r >= 0; r--){
        if(chess[r][col] == true) return false; // not safe
    }

    // top left
    for(int r = row, c = col; r >= 0 && c >= 0; r--, c--){
        if(chess[r][c] == true) return false; // not safe
    }

    // top right
    for(int r = row, c = col; r >= 0 && c < n; r--, c++){
        if(chess[r][c] == true) return false; // not safe
    }

    return true; // safe
}
```

O(N)

Sudoku Solver (LC 37)



9x9

cell → 1 to 9

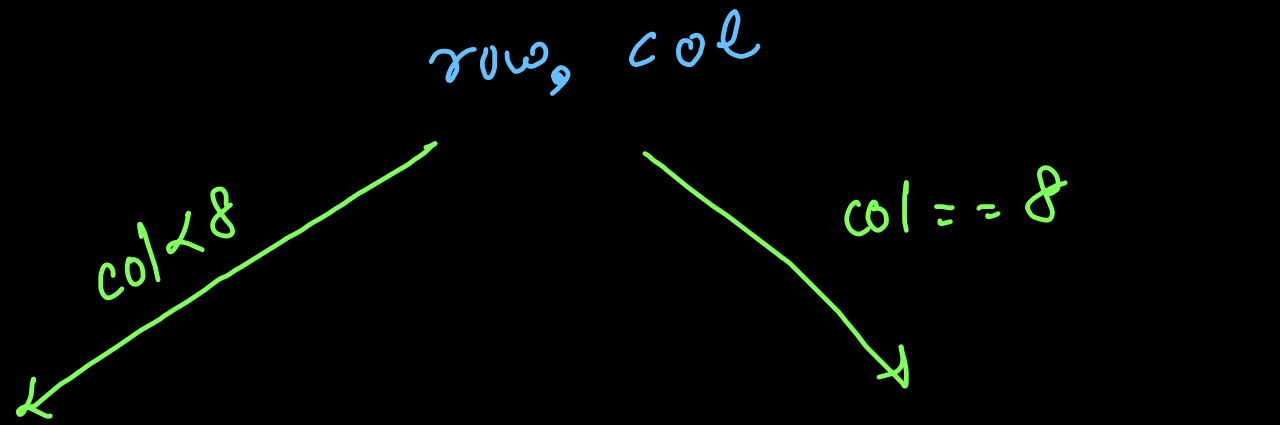
Constraint

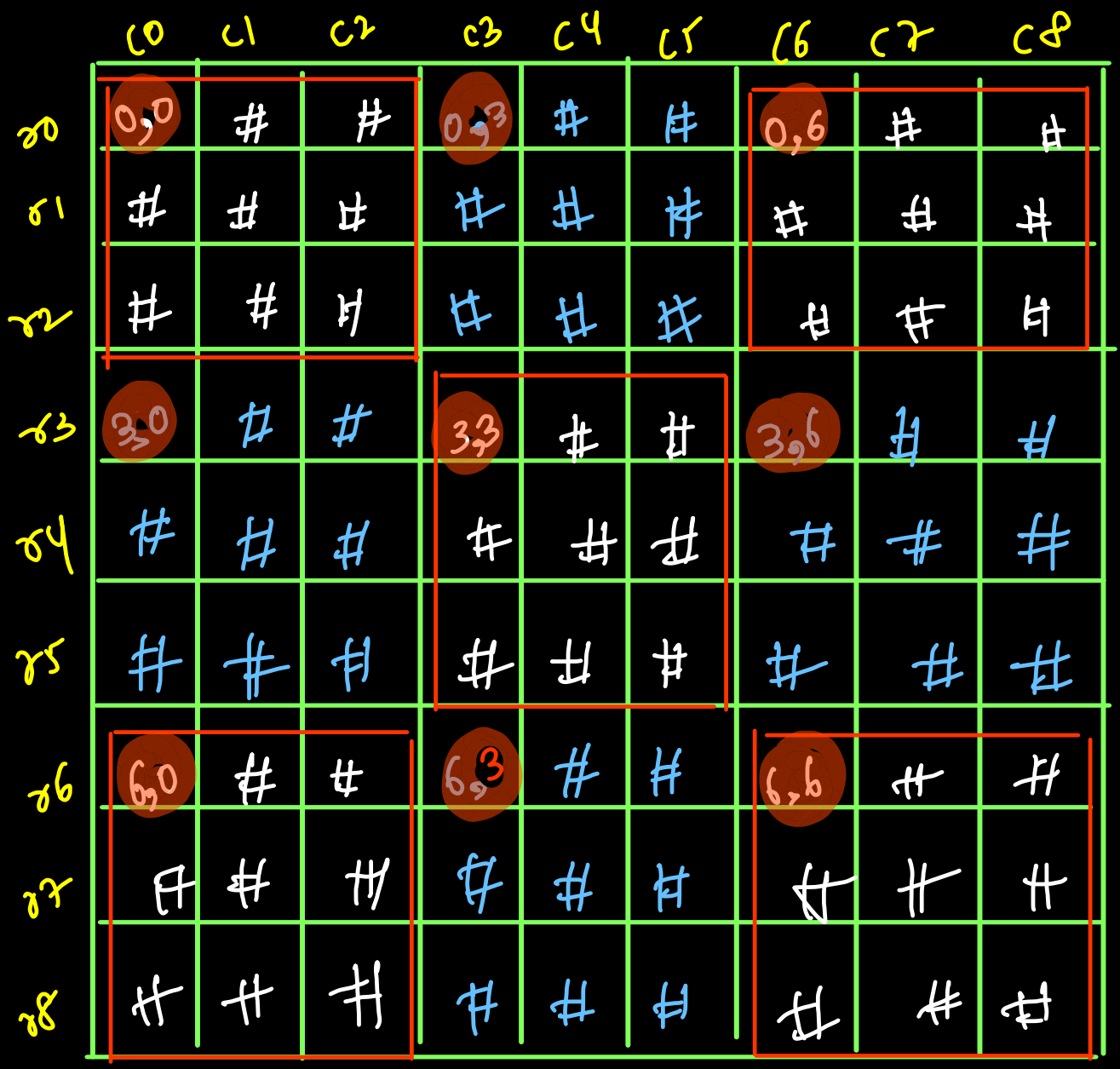
(1) no should not repeat in a same row

(2) no should not repeat in a same col

(3) no should not repeat in a subgrid.

cell by cell recursion





Print → 1, 1

(row, col)

top left corner of subgrid?

= ((row/3)*3, (col/3)*3)

```
char[][] result;

public boolean isSafe(char[][] board, int row, int col, char ch){
    // row
    for(int c = 0; c < 9; c++){
        if(board[row][c] == ch) return false;
    }

    // col
    for(int r = 0; r < 9; r++){
        if(board[r][col] == ch) return false;
    }

    // subgrid
    row = (row / 3) * 3;
    col = (col / 3) * 3;
    for(int r = 0; r < 3; r++){
        for(int c = 0; c < 3; c++){
            if(board[row + r][col + c] == ch)
                return false;
        }
    }

    return true;
}
```

```
public void sudoku(char[][] board, int row, int col){
    if(row == 9) {
        result = new char[9][9];
        for(int r = 0; r < 9; r++){
            for(int c = 0; c < 9; c++){
                result[r][c] = board[r][c];
            }
        }
        return;
    }

    // cell is already filled
    if(board[row][col] != '.'){
        sudoku(board, (col < 8) ? row : row + 1, (col < 8) ? col + 1 : 0);
        return;
    }

    // empty cell -> choices 1 to 9
    for(char ch = '1'; ch <= '9'; ch++){
        if(isSafe(board, row, col, ch) == false) continue;

        board[row][col] = ch;
        sudoku(board, (col < 8) ? row : row + 1, (col < 8) ? col + 1 : 0);
        board[row][col] = '.'; // backtracking
    }
}

public void solveSudoku(char[][] board) {
    sudoku(board, 0, 0);
    board = result;
}
```

$n \times n = 9 \times 9$

$depth = n^2$

$calls = n$

$O(n^{n^2}) = O(9^{81})$ Worst case

alternative code

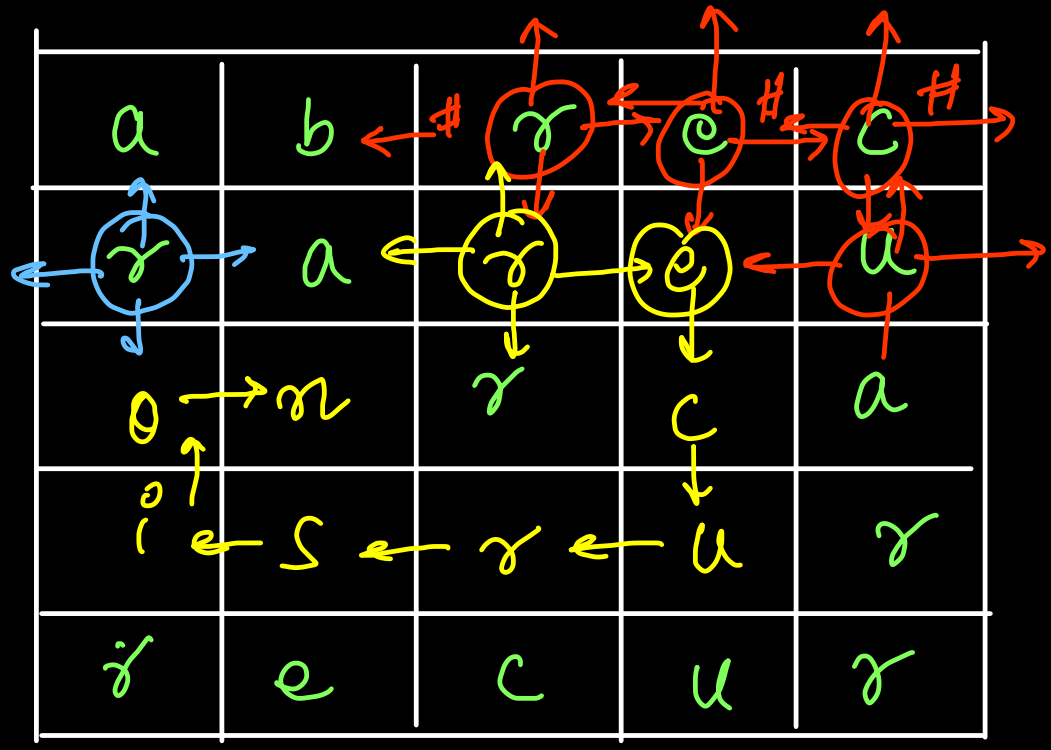
int nextrow = row, nextcol = col;

if (col == 8) { nextrow++; nextcol = 0; }
else nextcol++

```java
public boolean isSafe(char[][] board, int row, int col, char ch){
    // row
    for(int c = 0; c < 9; c++){
        if(board[row][c] == ch) return false;
    }

    // col
    for(int r = 0; r < 9; r++){
        if(board[r][col] == ch) return false;
    }

    // subgrid
    row = (row / 3) * 3;
    col = (col / 3) * 3;
    for(int r = 0; r < 3; r++){
        for(int c = 0; c < 3; c++){
            if(board[row + r][col + c] == ch)
                return false;
        }
    }

    return true;
}
```

```java
public boolean sudoku(char[][] board, int row, int col){
    if(col == 9){ row++; col = 0; }
    if(row == 9) return true;

    // cell is already filled
    if(board[row][col] != '.'){
        return sudoku(board, row, col + 1);
    }

    // empty cell -> choices 1 to 9
    for(char ch = '1'; ch <= '9'; ch++){
        if(isSafe(board, row, col, ch) == false) continue;

        board[row][col] = ch;
        boolean ans = sudoku(board, row, col + 1);
        if(ans == true) return true;
        board[row][col] = '.'; // backtracking
    }

    return false;
}

public void solveSudoku(char[][] board) {
    sudoku(board, 0, 0);
}
```

Word Search (RC 79)



→ neighbouring cell

→ same cell can be used almost once

"recursion"

true

target

"xyz"

false

```java
// {top, bottom, left, right}
int[] dr = {-1, +1, 0, 0};
int[] dc = {0, 0, -1, +1};

public boolean search(char[][] board, String word, int row, int col, int idx){
    if(idx == word.length()) return true;

    if(row < 0 || col < 0 || row == board.length || col == board[0].length){
        // index out of bound
        return false;
    }

    if(board[row][col] != word.charAt(idx)){
        // character mismatch
        return false;
    }

    for(int d = 0; d < 4; d++){
        board[row][col] = '#';
        boolean ans = search(board, word,  row + dr[d], col + dc[d], idx + 1);
        if(ans == true) return true;
        board[row][col] = word.charAt(idx); // backtracking
    }

    return false;
}


public boolean exist(char[][] board, String word) {
    for(int r = 0; r < board.length; r++){
        for(int c = 0; c < board[0].length; c++){
            boolean ans = search(board, word, r, c, 0);
            if(ans == true) return true;
        }
    }

    return false;
}
```

T B L R TL TR BL BR

dr = {-1, +1, 0, 0, -1, -1, +1, +1}

dc = {0, 0, -1, +1, -1, +1, -1, +1}

Time → O(4^Length)

Space → O(Length)

board[r][c] != '#'

↳ visited cell ⇒ return false

calls = 4

depth = L

O(4^L) ⇒ exponential

4 string will not contain '#'