Hard Work beats Talent,

When Talent doesn't Work Hard!

RECURSION

INTRODUCTION :-

1.) Make Input Smaller! But Why?

2.) Recursion – Decision Space

3.) Recursive Tree – Soul of Recursion] Imp.

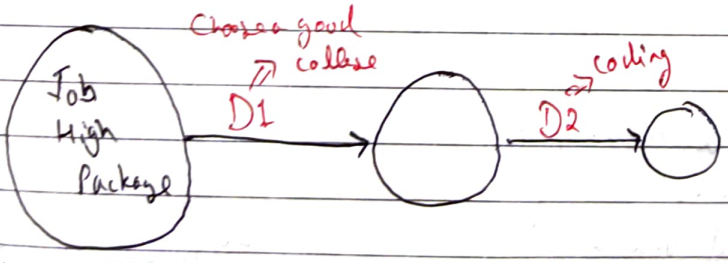
4.) 2 steps to solve any recursive Problem

1.) Make Input Smaller! But Why?

We don't deliberately smaller the input. The input becomes smaller automatically!

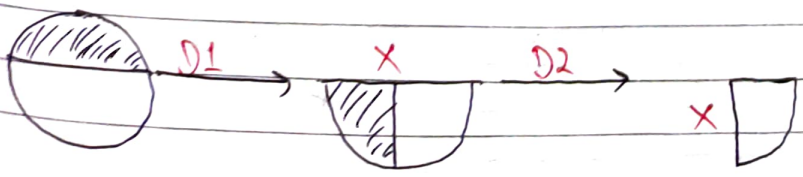
Example :-

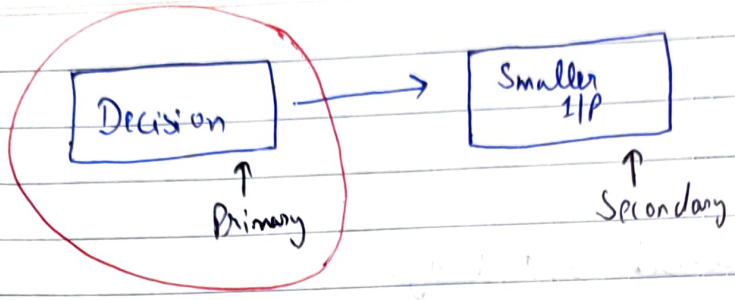
Let's understand it with an example. How the input is getting smaller.



To get a high package job. I have to take 1st Decision of good college. By doing that our problem becomes small. Now our 2nd Decision is to learn coding. Our problem becomes more smaller.

Similar thing with our Input.





Our decision has to be a primary goal & on the basis of that Input becomes smaller automatically

2) Recursion – Decision Space

Whenever we have to think about Space. Recursion is a good choice.
now, how can we identify that wether a problem statement use recursion or not.

⇒ For that we can see that, the problem has given choices & some decision. So, it's using recursion

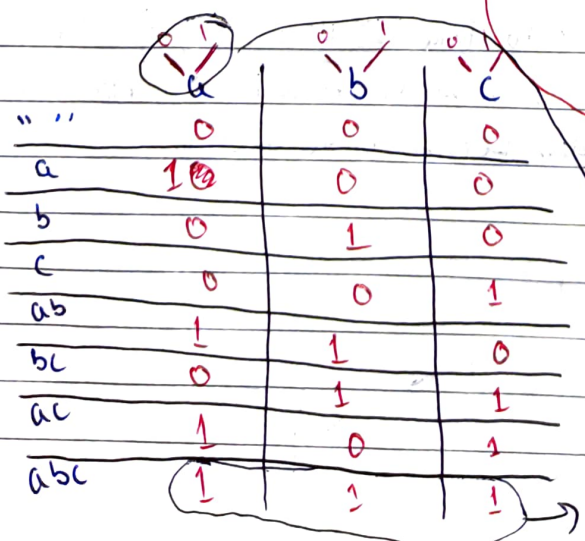
Choice + Decision

3) Recursion Tree

Let's understand with example.

→ Get subset of "abc"

So, it's subset will be "" a ab abc b bc c ac

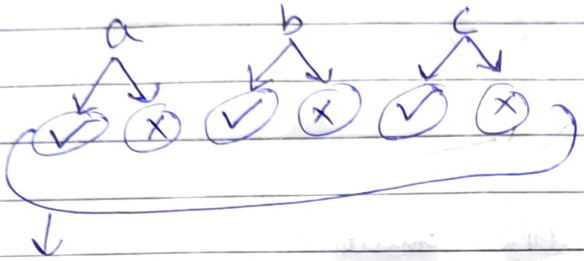


In recursion we have Choices & take decision

This is my choice +

These are my decision

⇒ So in this what choices do I have.
I should consider or not



So on these choices whatever the decision I take will make my subset!

⇒ So, if you all we talking choices wether to have or not. And by that choices we talking decision. And because of that our Input is becoming smaller

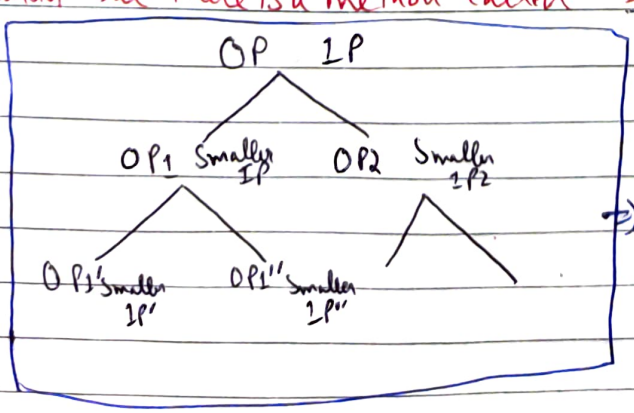
Now let's understands What is Recursive tree?

Again, lets understand with example we have to get subset of "ab"

→ "" ax bx
a a✓ bx
b ax b✓
ab a✓ b✓

So, instead of representing in axbx way. We want to represent in a good way. That way is called Recursive tree

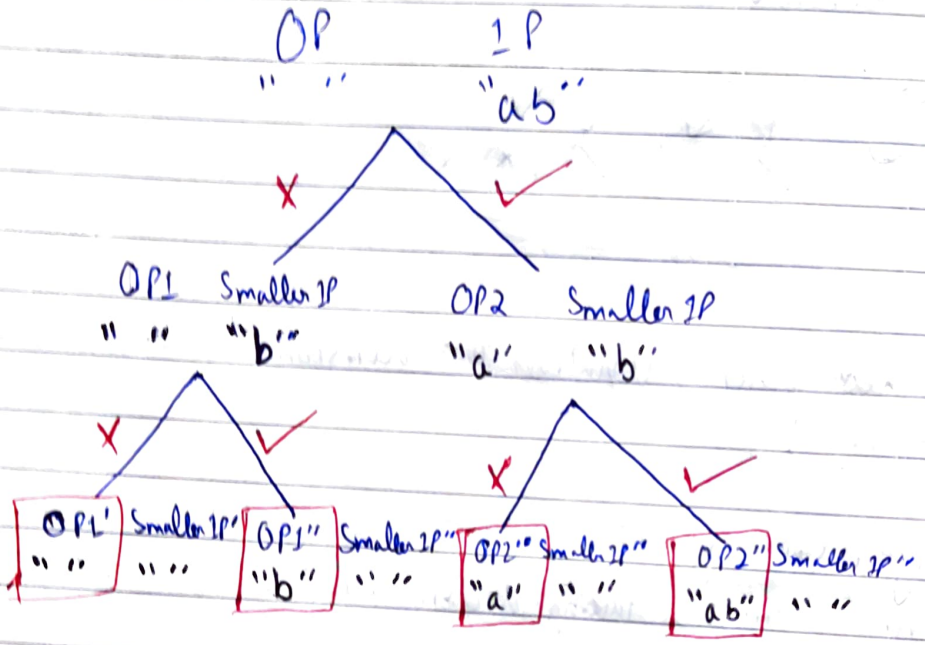
For Recursive Tree there is a method called "IP-OP method"



OP = Output
IP = Input

⇒ TREE

Let's understand this tree with an example.



- When Smaller "IP" becomes empty return the tree & gets the answer

4) 2 Steps to Solve Recursive Problem

1) Design a recursive tree

2) Fuck the Problem aka Write the code

RECURSION IS EVERYWHERE

We use recursion most of the places.

Data Structure :-

- ARRAY / String
- TREE
- HEAP
- GRAPH [DFS]
- STACK
- Linked List

Recursion is backbone of :- DP | BACKTRACKING | Divide Conquer

1.) Print 1 to n / N to 1
2.) Sort an array → Sort an Stack
3.) Delete middle element in a stack
4.) Remove Duplicate from a String
5.) Count no of occurence using Recursion

Easy

6.) Permutation :-
- Space Variation
- Case Change
- And many More

7.) Josephus Problem

Medium

8.) Print N binary numbers having more 1's than 0's
9.) Generate all Balanced Parentheses

Hard

Hypothesis - Induction - Base Condition

⇒ Print 1 to N / N to 1

Flow:- ✱ Methods for a Recursive Problem

✱ Why making I/P smaller is Popular/Imp.

✱ Print 1 to N

1) Methods:- There are 4 approaches to solve a Recursive problem:-

→ Recursive Tree - I/O Method [Works only if you know Decision]

→ B C - Induction - Hypothesis [making I/P smaller]

→ Choice Diagram (DP)

→

Recursion Problem ⟶ Hypothesis [work like recursion tree]

Recursion for ⇒ Solve(n) = 1 to n

Magic

Induction

Base Condition:- 1) Smallest valid I/P

2) Smallest Invalid I/P

```
void print(int n)
{
    if (n==1)
      return 1          ⇒ Base Condition

    print(n-1)          ⇒ Hypothesis

    S.O.Pln(n)          ⇒ Induction
}
```

Output:- 1
2
3
4
.
.
7

print(7)
↓
print(6)
↓
print(5)
.
.
print(1)

We will just look at till here. Till here we have to work

Cover up:-
- Easy by IBH
- Medium by Recursion tree
- Hard by Choice diagram

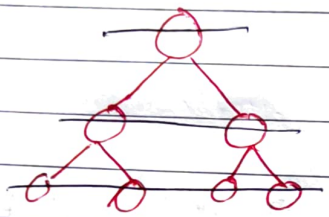
```
void print(int n)
{
    if (n==1) return;

    S.O.Pln(n);

    print(n-1);
}
```

Output ⇒ 7
6
5
4
.
.
1

Height Of A Binary Tree

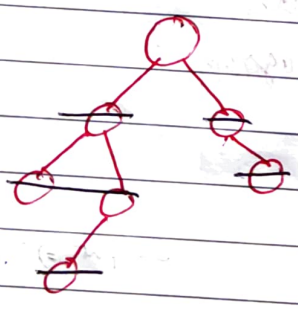
Let's understand this with an tree example.

Say, we have given something like this:-



To get his height, we draw line. That is "3"

Now, by recursion how we can do this. So we know the root & its child will be find by hypothesis. So we go deep inside the left, then right & from where we get the max. deep. That will be count as height with +1 +1 = bcz, root will count as well.



L = 3

R = 2

The code will be

```
int height(Node root){
    if(root == null) return 0; //Base condition

    int left = height(root.left);
    int right = height(root.right);          // hypothesis

    return Math.max(left, right)+1;          // Induction
}
```

Sort An Array Using Recursion

Firstly your thought is, why we are doing this with recursion. As, we can do that using Sorting or Iterative way.
I say, you will learn from this a lot. So, Now without any further due let's start.

PS:- I/P:- [2, 4, 3, 6, 9, 7]
O/P:- [2, 3, 4, 6, 7, 9]

I want you to understand with more basic example

arr = [0, 1, 5, 2] Sort this array

Our hypothesis will be make it smaller
So, Sort [0|1|5|~~2~~]
Sort ([0|1|5] [2])

Now in induction
insert([0|1|5] [2])
Again in hypothesis make it more smaller
insert ([0|1], [2]) → [0|1|2] [5]

Sort $\xrightarrow{\text{induction step}}$ insert $\xrightarrow{\text{induction step}}$ array.push

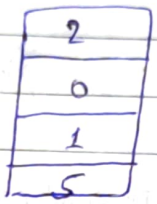
Code on next Page:-

```
Public void List<Integer> Sort( List<Integer> arr ){
    if(arr.size()==1) return arr; } Base Condition

Hypothesis {
    Integer temp= arr.get(arr.size()-1); //[0,1,5,2]
    arr.remove(arr.size()-1); //[0,1,5] [2]
    arr= sort(arr);   //[0,1,5]

Induction {
    return arr= insert(arr,temp); // [0,1,5] [2]
}

Public void List<Integer> insert( List<Integer> arr, int i){
    if (arr.size()==0 || i>= arr.get(arr.size()-1))
    {
        arr.add(i);      // let say in last we have 6
        return arr;      [0,1,5] [6] => [0,1,5,6]
    }

Hypothesis {
    int val= arr.get(arr.size()-1); // [0,1,5]
    arr.remove(arr.size()-1); // [0,1] [5]
    arr= insert(arr,i); //[0,1,2]

Induction {
    arr.add(val); //[0,1,2,5]
    return arr;
}
```

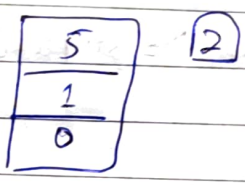
Sort A Stack

This is much similar to "sort an array".
I mean its almost to that

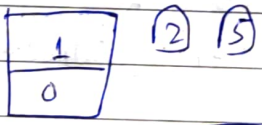
first what we do ; let's take an example:-

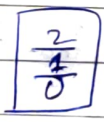


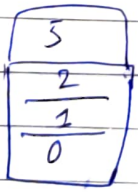
Take the top element & sort the stack!



Now insert 2 into stack but before that let's make our input more smaller



Now add 2 in stack =  & then finally add 5 in stack =



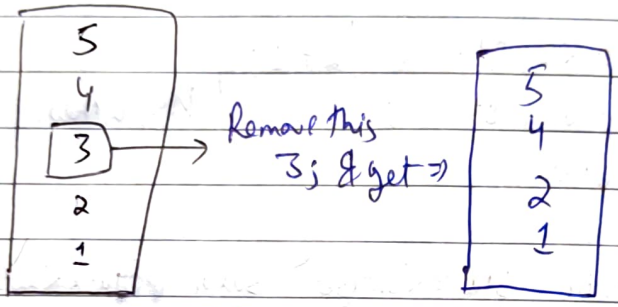
Let's Code it!

```
public void Stack<Integer> sortedStack(Stack<Integer> st){

    if(st.size()==1) return st;          } Base Condition

    Integer temp= st.pop();
    SortedStack(st);                     } Hypothesis

    return insertElementAt(st, temp);    } Induction
}
public void Stack<Integer> insertElementAt(Stack<Integer>st, Integer temp)
{
    if(st.size()==0 || temp >= st.peek()){   } Base Condition
        st.push(temp);
        return st;
    }

    Integer val= st.pop();
    insertElementAt(st, temp);           } Hypothesis

    st.push(val);                        } Induction
    return st;
}
```

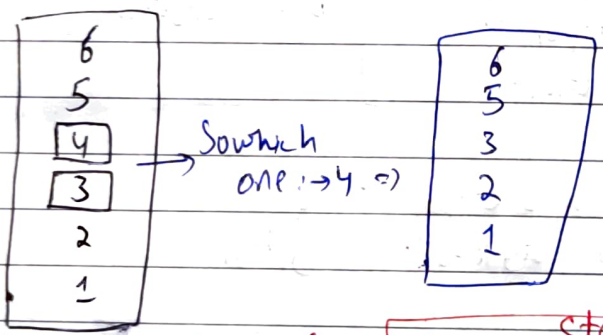
Delete Middle Element Of A Stack

So, in this we have to delete the middle element from the stack.

If Stack is odd, middle element will be



If Stack is even, middle element will be

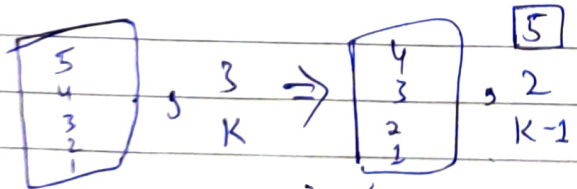


Now 4, we have formula;

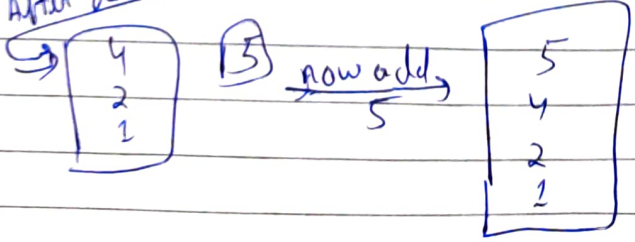
$$\text{Stack} = \frac{\text{Stack}}{2} + 1$$

So, how we do this we have choice means [1 BH], if 1 Can take decision then [Tree].

But I have ~~Extra~~ smaller Input



After deleting 3



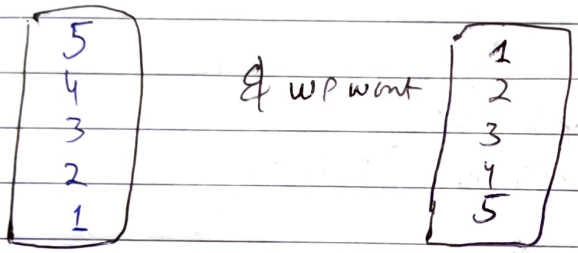
Let's Code it;

```
public static StomidDel (Stack<Integer> st) {

    if (st.size() == 0) return st; } Base Condition

    int k = st.size()/2 + 1;
    Solve (st, k);                  } Hypothesis
    return;
}
public static Stack<Integer> solve (Stack<Integer> st, int k) {
    if (k == 1) {
        st.pop();
        return;
    }                               } Base Condition
    int temp = st.pop();
    Solve (st, k-1);                } Hypothesis

    retwin st.push(temp); } Induction
}
```

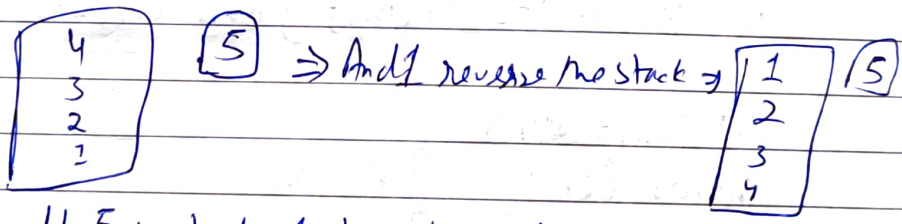
Reverse A Stack

So, we have given a stack & without using extra space we have to reverse the stack.

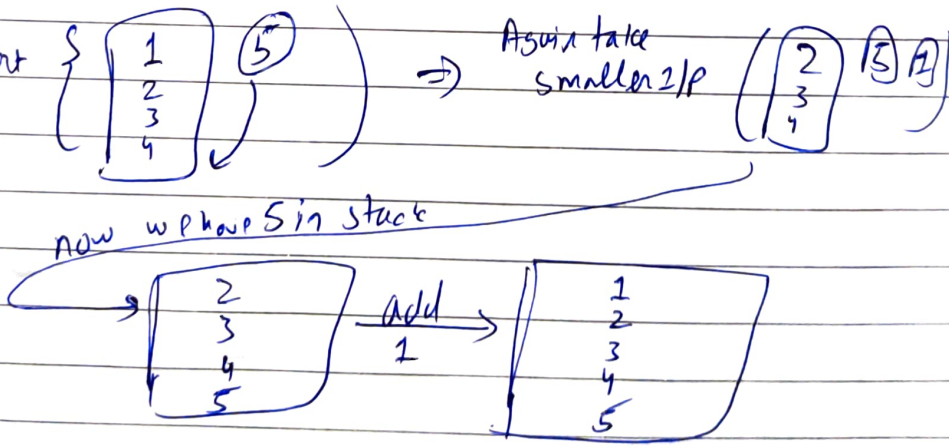
So, let's say we have something like this:-



first let me take a smaller Input that's my hypothesis.



Now to add 5 in stack. I have to empty my stack first & then add 5 & fill the stack after that.



Let's Code it

```
public void insert(Stack<Integer> st, int element){
    if(st.size()==0){
        st.push(element);
        return;
    }
    int temp= st.pop();
    insert(st, element);
    return st.push(temp);
}
```

Base Condition

Hypothesis

Induction

```
Public void reverse(Stack<Integer> st){
    if(st.size()==1) return;
    int temp= st.pop();
    reverse(st);
    insert(st, temp);
    return;
}
```

Base Condition

Hypothesis

Induction

K^{th} Symbol In Grammar

So, what it saying we have given two integer n & k. We have to return K^{th} (1-indexed) symbol in the n^{th} row of table n rows.

Example:- (n = 4, k = 3)

k=1

n=1 → 0

n=2 → 0 1

n=3 → 0 1 1 0

n=4 → 0 1 1 0 1 0 0 1

k= 1 2 ③

O/P

Now, this question is all about to check your observation skill!

So if we carefully look at, & we have to find induction:-

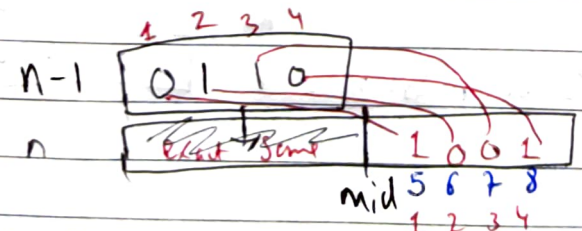
0

0 1

n-1 0 1 1 0

n 0 1 1 0 1 0 0 1

If we see then we find out that (n) 1st half is equals to (n-1). And the remaining half is just the complement of (n)



So for the complement part, how we can do is if we check the indexing of n-1 == k - mid

Let's get its induction part ⇒

```
if (k <= mid) { // to get values of k before mid
    solve(n,k) ⟶ solve(n-1,k);
}
else { // get value of k after mid
    solve(n,k) ⟶ solve(n-1, k - mid); // Induction
}
```

Hypothesis

Base condition

```
if (n==1 && k==1) return 0;
```

Grammer to get mid

$1 = 2^0 \leftarrow n=1$ 0

$2 = 2^1 \leftarrow n=2$ 0 1

$4 = 2^2 \leftarrow n=3$ 0 1 1 0

$8 = 2^3 \leftarrow n=4$ 0 1 1 0 1 0 0 1

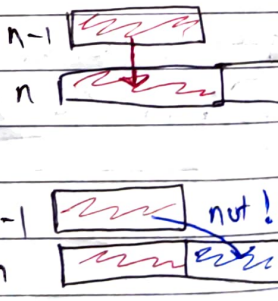
2^{n-1}

∴

```
int mid = (int)Math.pow(2, n-1) / 2;
```

Main logic

```
if (k <= mid)
{
    return & solve(n-1, k)
}
else
    return 1-solve(n-1, k-1)
```



OBSERVATION + IBH to solve this problm

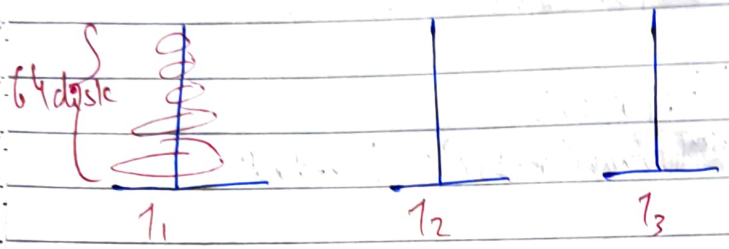
70% 30%

Let's Code it:-

```
public int KthGrammar(int n, int k){

    if(n==1 && k==1) return 0; //Base Condition

    int mid = (int)Math.pow(2, n-1)/2;

    if(k<=mid) // Hypothesis
        return KthGrammar(n-1, k); // Induction
    else
        return 1-KthGrammar(n-1, k-mid);
}
```

Tower Of Hanoi

Before solving the problem I have one Interesting story to share with you.
So, its original name is Tower of Brahma



Put on T_3 all 64 disk :-
But condition • 1 disk at a time
• And has to be in large to small order

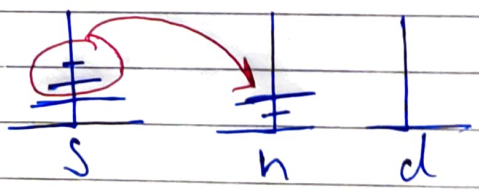
If 1 disk taking 1 sec then it will be $2^{64} \Rightarrow$ 585 billions of $[2^{30} = 1B] + [2^{30}] + [2^4 = 16] \Rightarrow$ [16BB seconds] age of universe

Problem :- Given 3 disk & 3 tower. Return the steps of putting from T_1 to T_3

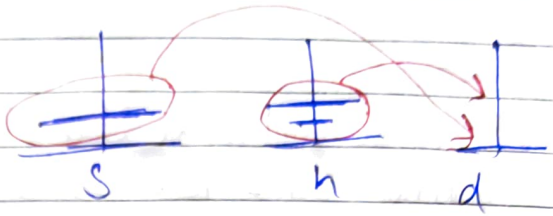
Hypothesis

solve(n, s → d, h)
Smaller

solve(n, s → h, d)

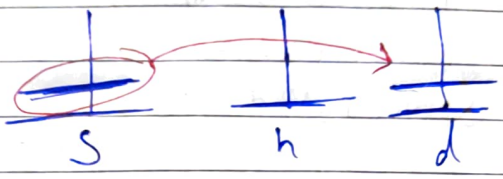


Induction



Base Condition

If there only 1 disk then simply put on destination



Pseudo Code

```
int main() {
    input(n);
    int s=1, h=2, d=3;
    solve(n, s, d, h);
}
void solve(int n, int s, d, h) {
    if (n==1) {
        print(moving plate n from s to d);
        return;
    }
    solve(n-1, s, h, d);
    Print(moving plate n from s to d);
    solve(n-1, h, d, s);
}
```

If ~~your~~ interviewer ask you to print no. of steps:-

Optional

Then in main funct. create
int count=0;
& in void funct. increment count
count++;
By doing that whenever func. call takes place count will increment.

Let's Code it:-

```
public static void main(String[] args){
    Scanner scn = new Scanner(System.in);
    int n = scn.nextInt();
    int s=1, d=2, h=3;
    solve(n,s,d,h);
}
```

} Main fn

```
public static void solve(int n, int s, int d, int h){
    if(n==1){
        S.o.Pln("Moving plate ~~from~~ ~~n~~ ~~to~~ from s to d)
        return;
    }
    solve(n-1,s,h,d);
    S.o.Pln(Moving plate n from s to d);
    Solve(n-1,h,d,s);
}
```

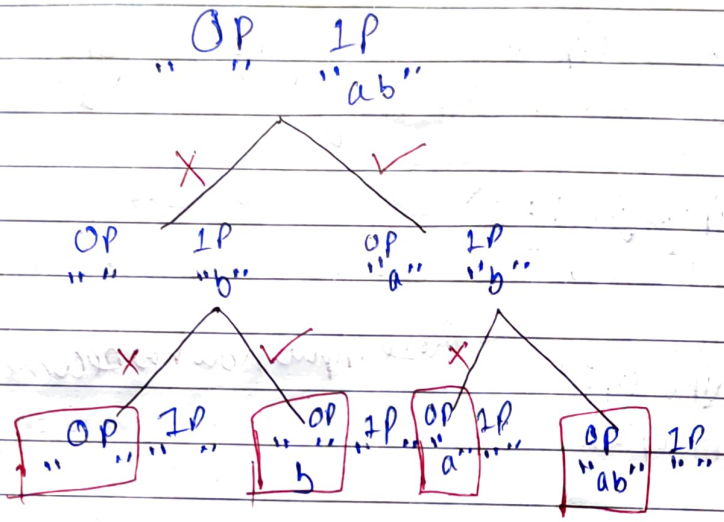
Base condition

Hypothesis

~~Hypothesis~~ Induction

PRINT SUBSETS

Let's first recap how we build a recursion tree of subsets.



First let's create main func.

PESUDO CODE

```
int main() {
    input=, str = ip;
    String op = " "
    solve(ip, op);
}
```

Let's call to our logic solve fnc.

```
void solve(String ip, String op){
if(ip.length()==0)
  {
    Print(op+" ");
    return;
  }
```

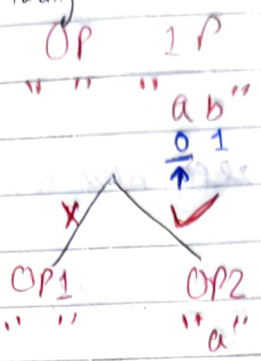
Base Condition

So, what this B.C is saying that, if input length becomes empty. Return output!

If you look at above diag. We are only returning when input becomes empty.

As we are not taking

String op1 = op // will be same as op.
String op2 = op // But for op2.
// We consider 0th index value.



op2.push_back(ip[0])
// adding 0th value to op2 string

ip.erase(ip.begin() + 0)
// we erase [remove] 0th value index for next consideration

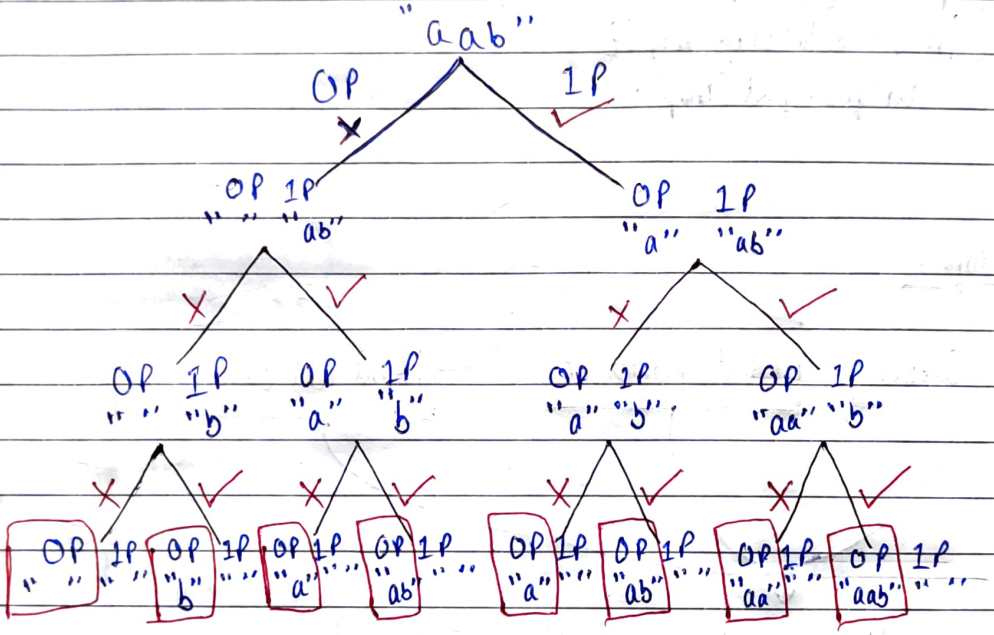
Solve (ip, op1) → These inputs now has become smaller
Solve (ip, op2)

Let's Code it:-

```
public static void main(String[] args) {
    Scanner scn = new Scanner(System.in);
    String ip = scn.nextLine();
    String op = "";
    solve(ip, op);
}

public static void solve(String ip, String op){
    if(ip.length() == 0) {
        S.o.Pln(op + "");
        return;
    }
    String op1 = op;
    String op2 = op;
    op2 += ip.charAt(0);
    ip = ip.substring(1);
    solve(ip, op1);
    solve(ip, op2);
}
```

Print Unique Subsets & Variation

As we already know how to print subsets, but what does print unique subsets means over here.
Let's see with an example:-



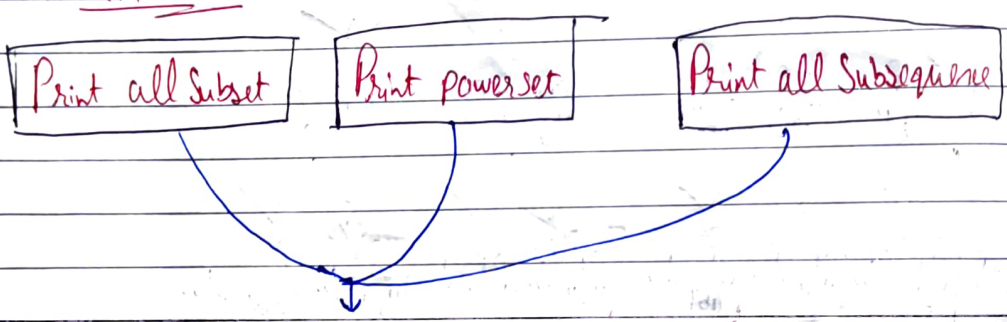
So from this we get subsets something like this,

""
b
a
ab
a
ab
aa
aab

If you carefully look there are some repeatance & we only want unique one. So, for that what we will do, instead of printing OP. We will create one Hashmap. And fill that Hashmap with all subsets.
We now create one more Hashmap & fill that one only with unique subsets.

```
if (ip.length() == 0){
    HashMap<String> map = new HashMap<>();
    return;
}
HashMap<String> unique = new HashMap<>();
if (!map.containsKey(map)){
    unique.put(map);
}
```

Variations



These 3 variations are not same. Print all subset is same as powerset. But subsequence is a diff. thing

Now, what is powerset?

⇒ We have given a set & we have to print it's all subset

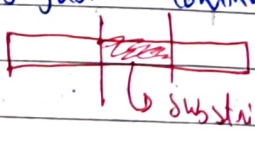
{a, b} ⟶ {a, b, "", ab}

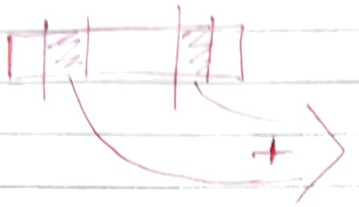
S ⟶ PS

Now, what is subsequence?

→ To understand this, we have to understand 1st substring.

Substring is just a continuous part of string



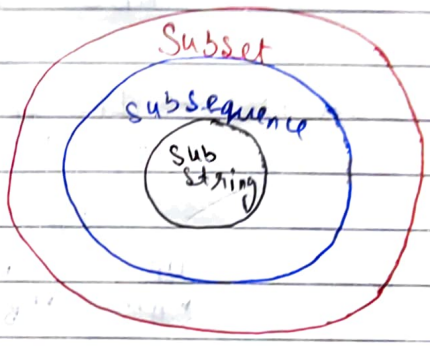


Is not a substring,
But it's a subsequence

"abc" → ac is a subsequence
but ca is not a subsequence.

Venn's Diagram

All substrings are → subsequence
All subsequence are → subsets



Sum-UP

→ If interviewer says to print Powerset, subset OR subsequence we have to print subset only.
Now here's 2 conditions
→ Have Duplicate
↳ Not a unique set
→ Print exactly tree
→ Print lexicographically

→ Not have Duplicate
↳ A unique set
→ Print exactly tree
→ Print lexicographically

→ For making unique use HashMap

→ For making lexicographical orders. Put into an Array, then sort it.

PERMUTATION WITH SPACES

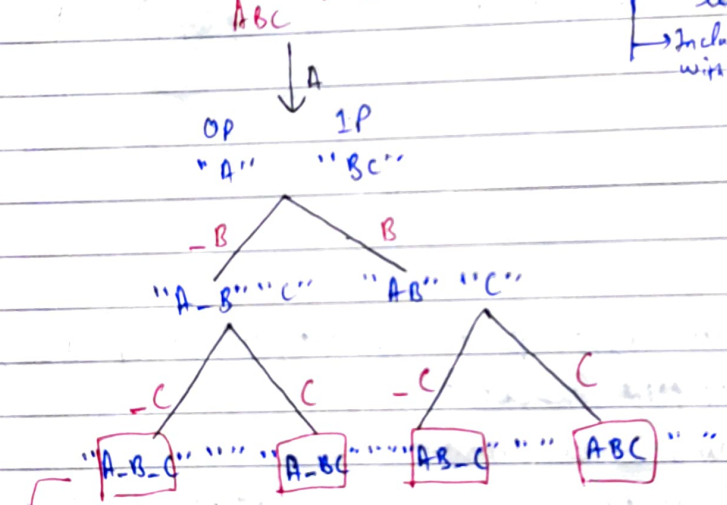
Problem Statement:-

Input:- ABC

Output:- A_B_C
A_BC
AB_C
ABC

Recursive Tree → IP OP

Choice
- → Include that letter with space
- → Include that letter without space



↳ Same as Output

A_B_C
A_BC
AB_C
ABC

PESUDO CODE

```
int main() {
    input ip= " ";
    String op= " ";
    ~~op= ip.charAt~~ op.pushback(ip[0]) //including A as it is
    ip.erase(ip.begin()+0); // removing A for ABC = BC
    Solve(ip, op)
}
```

```
void solve (String ip, String op){
    if (ip.length()==0){                    // Base Condition
        cout<<op<<" ";
        return;
    }
    String op1=op;
    String op2=op;
    op1.push_back(" _ ");
    op1.push_back(ip[0]);                   // Including with space
    op2.push_back(ip[0]);                   // Including without space
    ip.erase(ip.begin()+0);                 // Erasing from input to make smaller
    solve(ip,op1);
    solve(ip,op2);
    return;
}
```

3 Let's Code it:-

```
public static void main (String[] args) {
    Scanner scn = new Scanner(System.in);
    String ip= scn.nextLine();
    String op=" ";
    op+= ip.charAt(0);
    ip= ip.substring(1);
    solve(ip,op);
}
public static void solve (String ip, String op){
    if(ip.length()==0) {
        S.O.Pln(op);
        return;
    }

    String op1=op;
    String op2=op;
```

```
op1+="_"+ip.charAt(0);
op2+=ip.charAt(0);
ip=ip.substring(2);

solve(ip,op1);
solve(ip,op2);
}
```

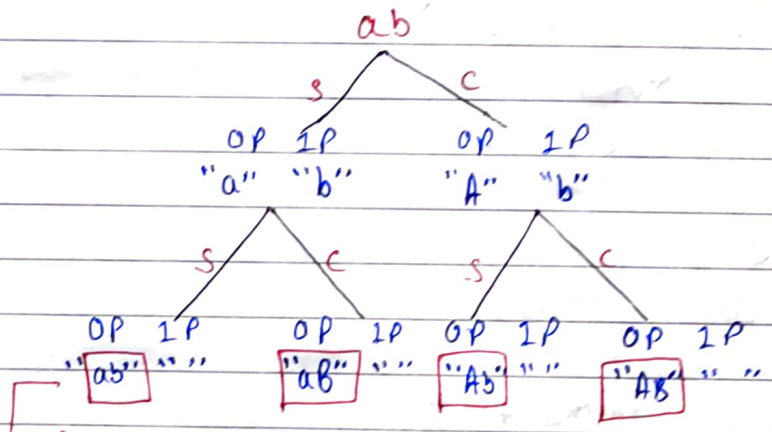
Permutation With Case Change

Problem statement:-

IP:- ab

OP:- ab
aB
Ab
AB

Recursive Tree → OP IP



ab
aB
Ab
AB

```
void solve (String ip, String op){
    if (ip.length()==0){
        print (op);
        return;
    }
    String op1=op;
    String op2=op;
    op1.push_back(ip[0]);
    op2.push_back(toupper(ip[0]));
    ip.erase(ip.begin()+0);
    solve (op1,op2);
    solve (ip,op2);
}
```

Let's Code it:-

```
public static void main(String[] args){
    Scanner scn = new Scanner(System.in);
    String ip = scn.nextLine();
    String op = "";
    solve(ip, op);
}
public static void solve(String ip, String op){
    if(ip.length()==0){
        System.out.println(op);
        return;
    }
    String op1 = op;
    String op2 = op;

    op1 += ip.charAt(0);
    op2 += Character.toUpperCase(ip.charAt(0));

    ip = ip.substring(1);

    solve(ip, op1);
    solve(ip, op2);
}
```

LETTER CASE PERMUTATION

Problem Statement:-

It's similar to previous question, but have some conditions in it.

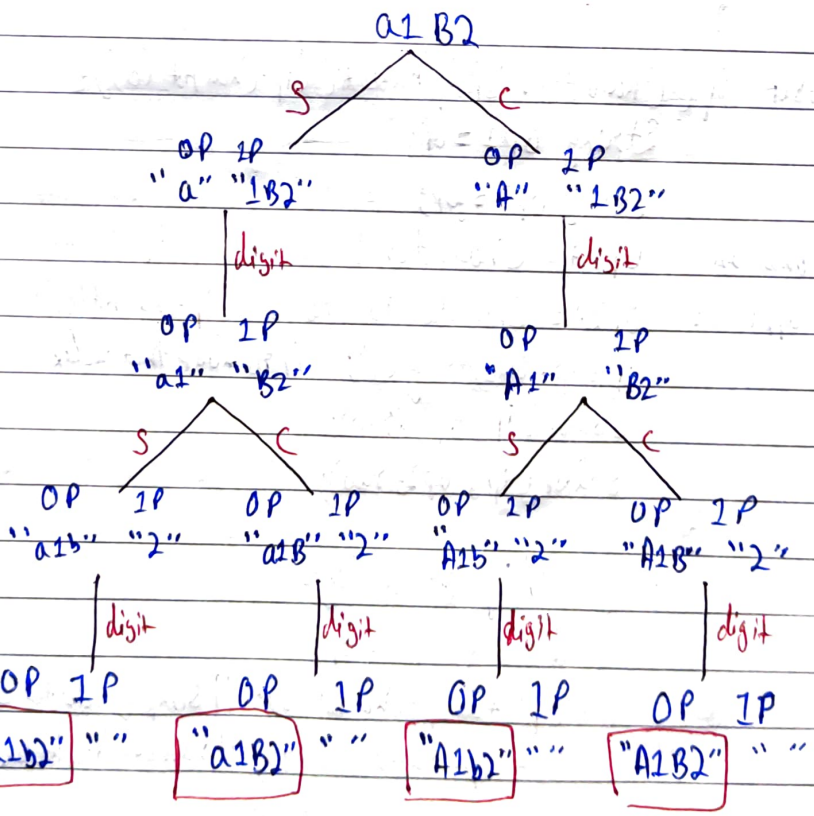
→ Letter could be smaller or upper

→ And has digit as well

IP:- a1B2

OP:- a1b2
a1B2
A1b2
A1B2

Recursive Tree → OP IP



→ Same as Output:- a1b2
a1B2
A1b2
A1B2

This is a Leet Code Problem,
Let's Code It:-

```
public List<String> letterCasePermutation(String s){
        String ip = s;
        String op = "";
        List<String> res = new ArrayList<>(); // result answer
        solve(ip, op, res); // func call
        return res;
}
public void solve(String ip, String op, List<String> res){
        if(ip.length()==0){ // Base Condition
            res.add(op); // add output to res
            return;
        }
// if its Alphabet if(Character.isAlphabetic(ip.charAt(0))){
            String op1 = op;
            String op2 = op;
// const. to lowercase op1 += Character.toLowerCase(ip.charAt(0));
// const. to uppercase op2 += Character.toUpperCase(ip.charAt(0));
            ip = ip.substring(1); // remove that index
            solve(ip, op1, res);
            solve(ip, op2, res);
        }
        else{ // if it's a digit
            String op1 = op;
            op1 += ip.charAt(0); // simply add to op1
            ip = ip.substring(1); // remove it's index
            solve(ip, op1, res);
        }
}
```

LETTER CASE PERMUTATION

Problem Statement:-

It's similar to previous question, but have some conditions in it.

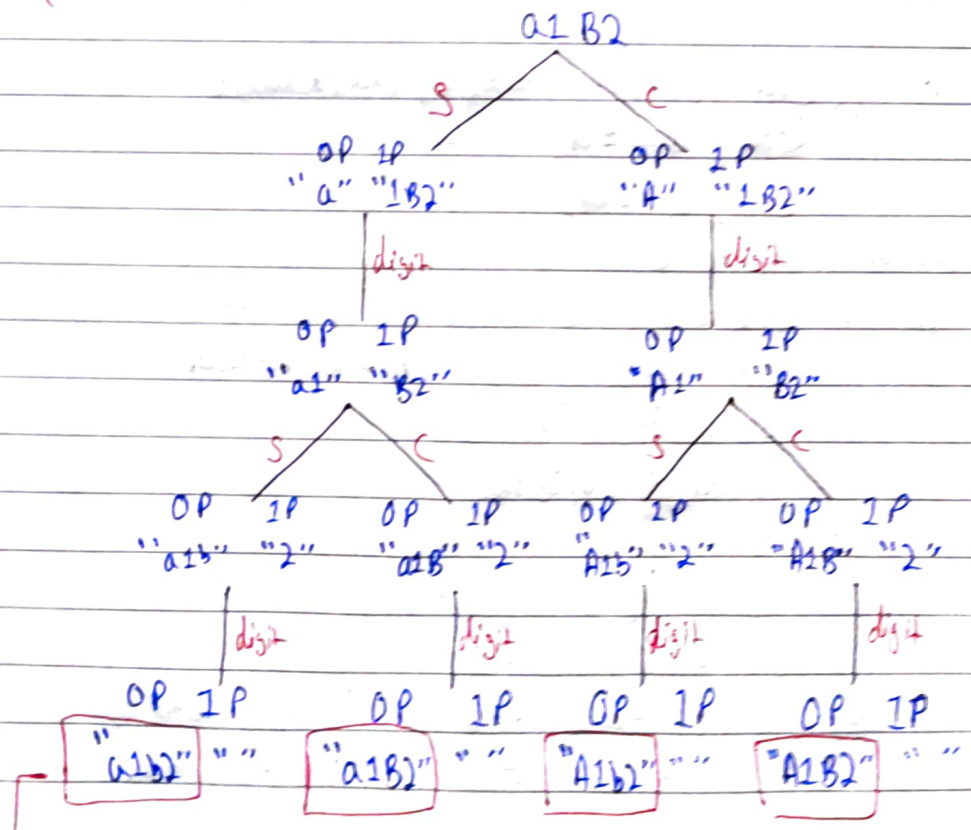
→ Letter could be smaller or upper

→ And has digit as well

IP:- a1B2

OP:- a1b2
a1B2
A1b2
A1B2

Recursive Tree → OP IP



→ Same as output:- a1b2
a1B2
A1b2
A1B2

This is a leetCode Problem,
Let's Code It:-

```
public List<String> letterCasePermutation(String s){
    String ip = s;
    String op = "";
    List<String> res = new ArrayList<>(); // result in vector
    solve(ip, op, res); // func call
    return res;
}
public void solve(String ip, String op, List<String> res){
    if(ip.length()==0){ // Base Condition
        res.add(op); // add output to res
        return;
    }
// if its Alphabet if(Character.isAlphabetic(ip.charAt(0))){
        String op1 = op;
        String op2 = op;
// const. to lower case op1 += Character.toLowerCase(ip.charAt(0));
// const. to upper case op2 += Character.toUpperCase(ip.charAt(0));
        ip = ip.substring(1); // remove that index
        solve(ip, op1, res);
        solve(ip, op2, res);
    }
    else{ // if it's a digit
        String op1 = op;
        op1 += ip.charAt(0); // simply add to op1
        ip = ip.substring(1); // remove it's index
        solve(ip, op1, res);
    }
}
```

Generate All Balanced Parentheses

Problem Statement :-

I/P :- n = 2 → Balanced Parentheses

↳ 2 open ((

2 close))

O/P :- ( ) ( ) (( ))

All possible balanced parentheses.

for n=2 boxes, case will be 4 :-

(( )) ← Valid parentheses

) ( ) (← Invalid parentheses

Let's take one more example, to understand it :-

n = 3

For n=3, possible balanced parentheses will be 5 :-

3 open (((

3 close)))

((( )))
(( ) ( ))
(( )) ( )
( ) (( ))
( ) ( ) ( )

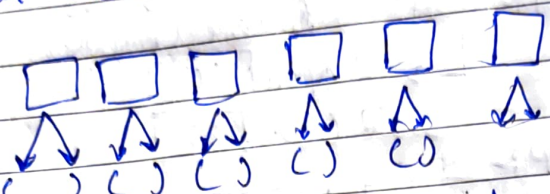
⇒ In the output we have to return Array of String, where every string is balanced

Now, Identify how this is recursion.

for that we have to think about **Choices + Decision**

To understand choices & Decision lets take n=3

for n=3, we have 6 boxes

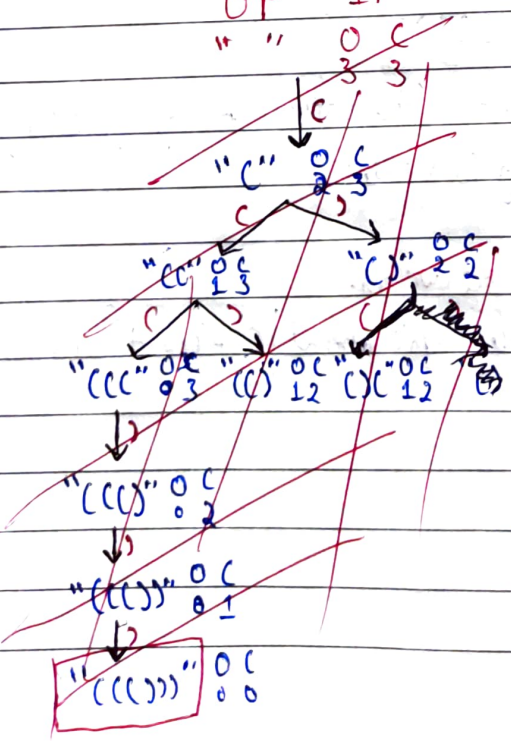


Now for every boxes we have 2 choices & we have to take decision accordingly.
So, our decision will give us balanced.
But we have to be carefully while taking decision.

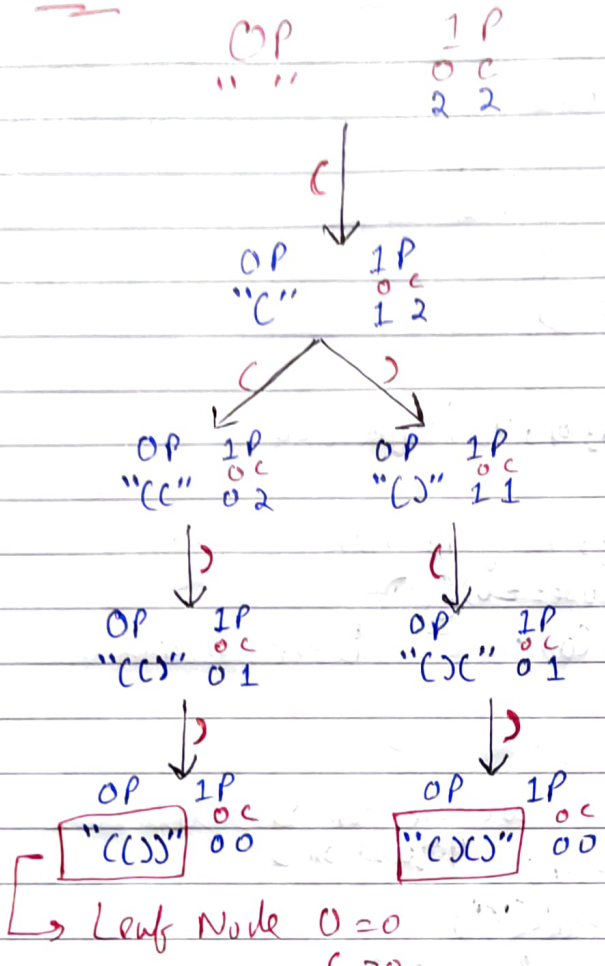
Let's Build Recursive Tree

To build it's recursive tree. If you see data types are not similar.

OP 1P
" " n=3
↓
String of Balance Parentheses



n=2



If you carefully look, sometimes we have 2 choices sometime 1 choice only.
What does it seems like we can only have closing bracket when, we have open bracket. Like opening brackets we have all time choice but for closing we get sometime or not!
So if open & close brackets are equal we will have 2 choices & if close brackets are more than open we will have closing choice.

If like)(→ use close bracket
But () so use open then when they are balance.
↑ ↑
Balance so that close become greater.

```
vector<string> bPar(int n){
    vector<string> v;
    int open=n;
    int close=n;
    string op="";
    solve(open, close, op, v) // func call
    return v;
}
void solve(int open, int close, string op, vector<string> v){
    if(open==0 && close==0){
        v.push_back(op);
        return;
    }
    if(open!=0){ // we have open still left
        string op1=op;
        op1.push_back('(')
        solve(open-1, close, op1, v);
    }
    if(close>open){ // we have close more than open
        string op2=op;
        op2.push_back(')')
        solve(open, close-1, op2, v)
    }
}
```

Initalize

Base Condition

It's an Leet Code Problem
Let's code it:-

```
Public List<String> generateParanthesis(int n){
    List<String> res = new ArrayList<>();
    int open = n;
    int close = n;
    String op = "";
    solve(open, close, op, res);
    return res;
}
public void solve(int open, int close, String op, List<String> res){
    if(open == 0 && close == 0){
        res.add(op);
    }
    if(open != 0){
        String op1 = op;
        op1 += '(';
        solve(open-1, close, op1, res);
    }
    if(close > open){
        String op2 = op;
        op2 += ')';
        solve(open, close-1, op2, res);
    }
}
```