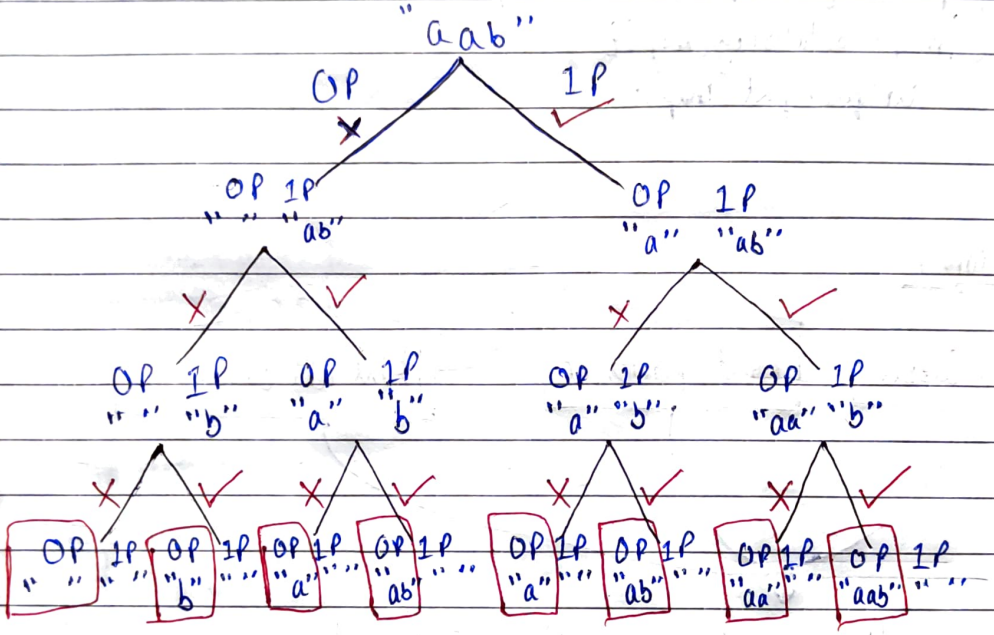
Print Unique Subsets & Variation

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



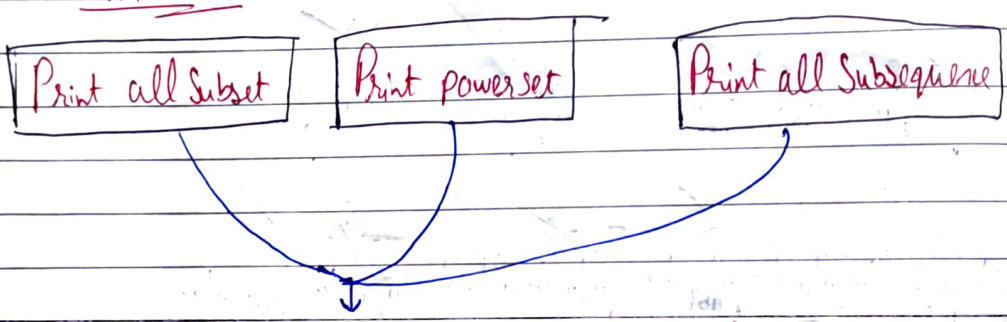
So from this we get subsets something like this,

""
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

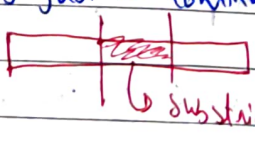
{a, b} ⟶ {a, b, "", ab}

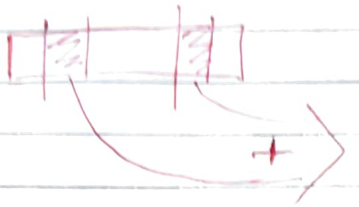
S ⟶ PS

Now, what is subsequence?

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

Substring is just a continuous part of string



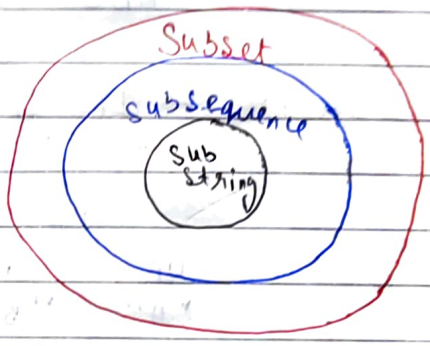


Is not a substring,
But it's a subsequence

"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

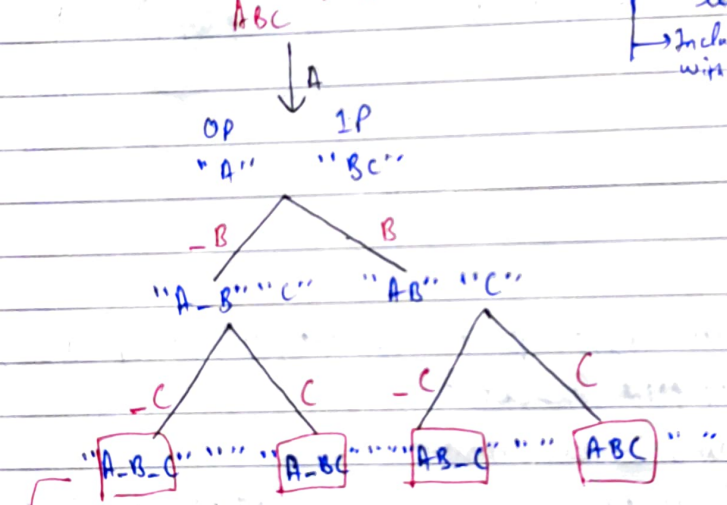
Problem Statement:-

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);
}
```

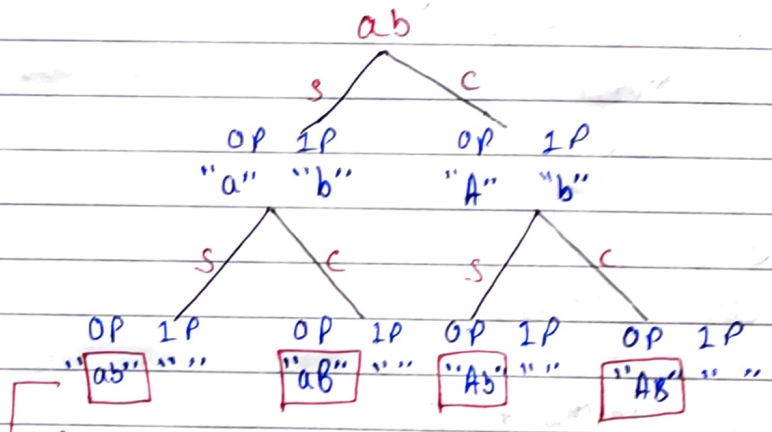
Permutation With Case Change

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.

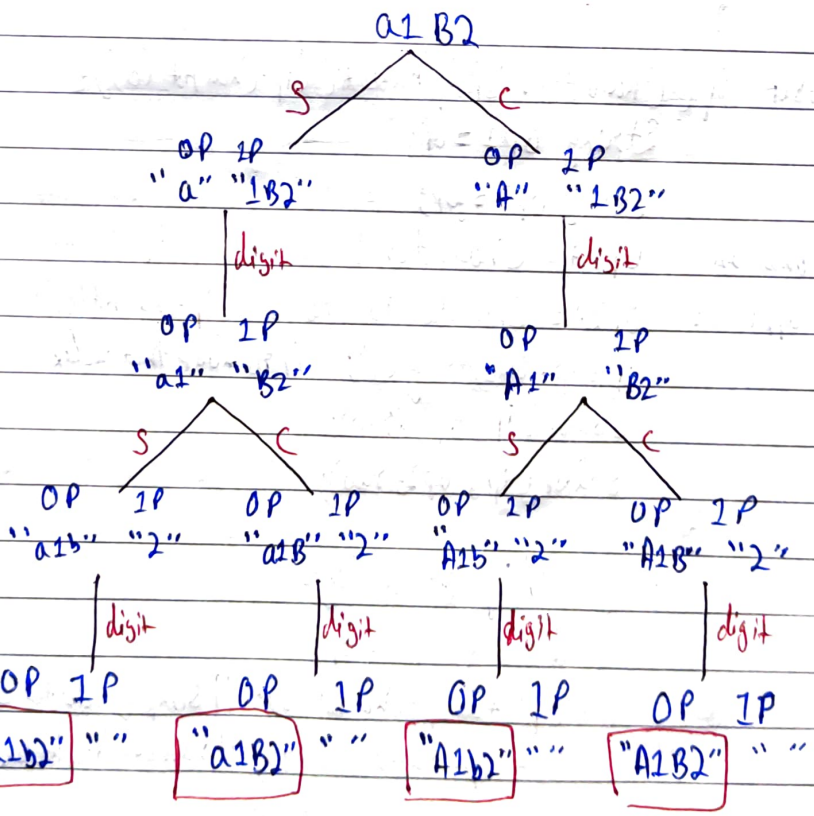
→ Letter could be smaller or upper

→ 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.

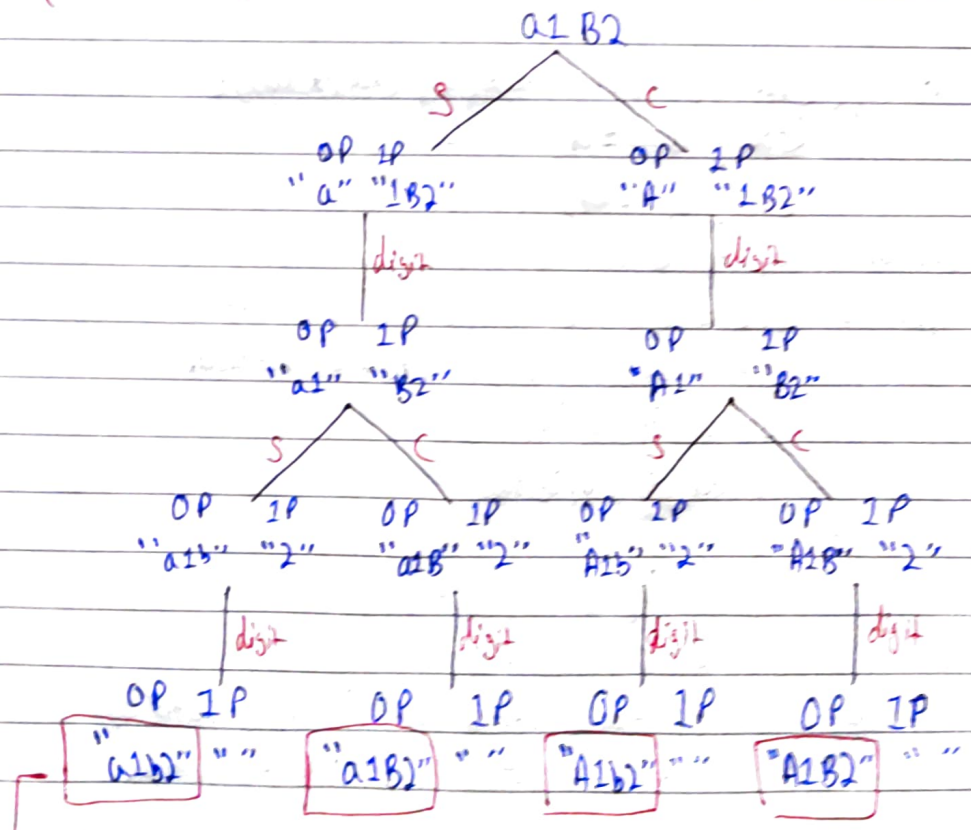
→ Letter could be smaller or upper

→ 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)))

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

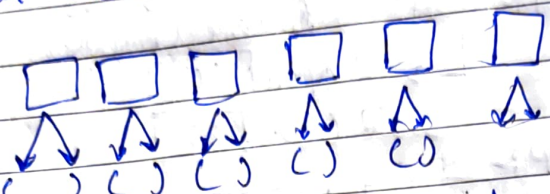
⇒ In the output we have to return Array of String, where every string is balanced

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

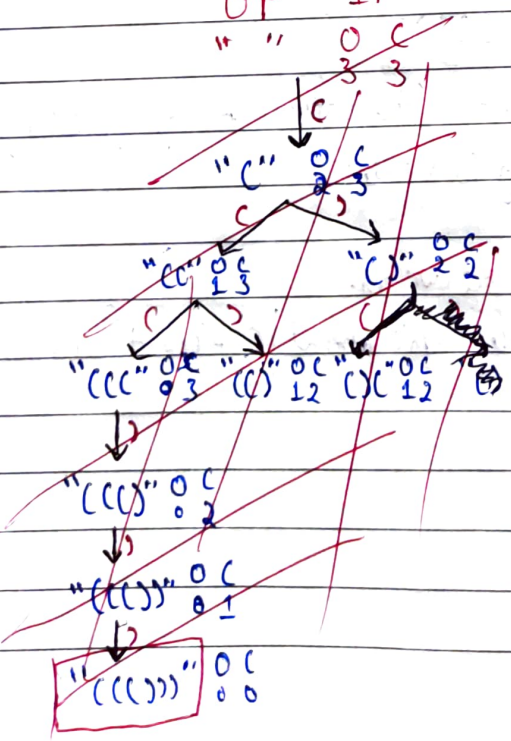


Now for every boxes we have 2 choices & we have to take decision accordingly.
So, our decision will give us balanced.
But we have to be carefully while taking decision.

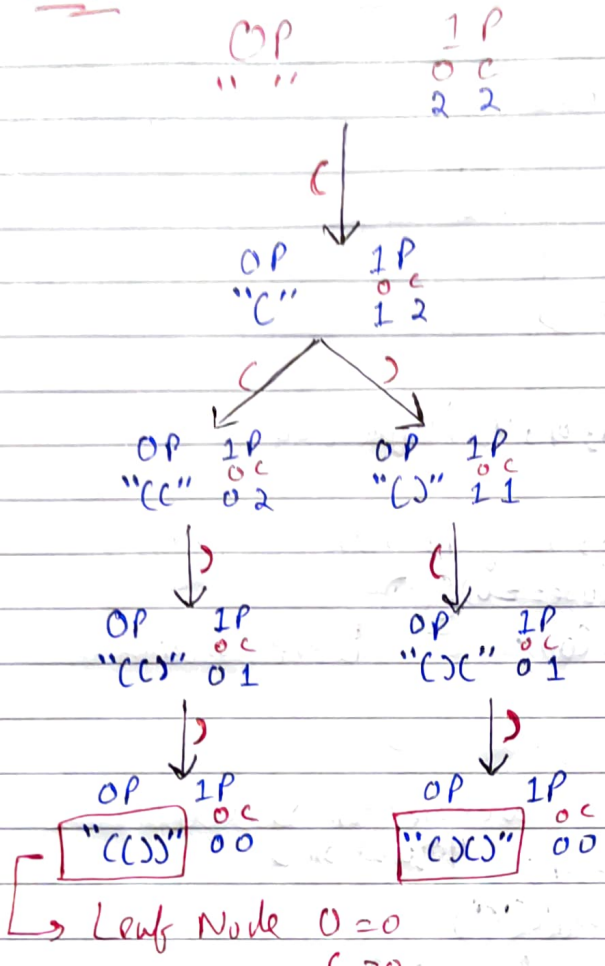
Let's Build Recursive Tree

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);
    }
}
```