Basic String Questions

Q). Remove a's from a String.

Eg:- baccad

Ans = bccd.

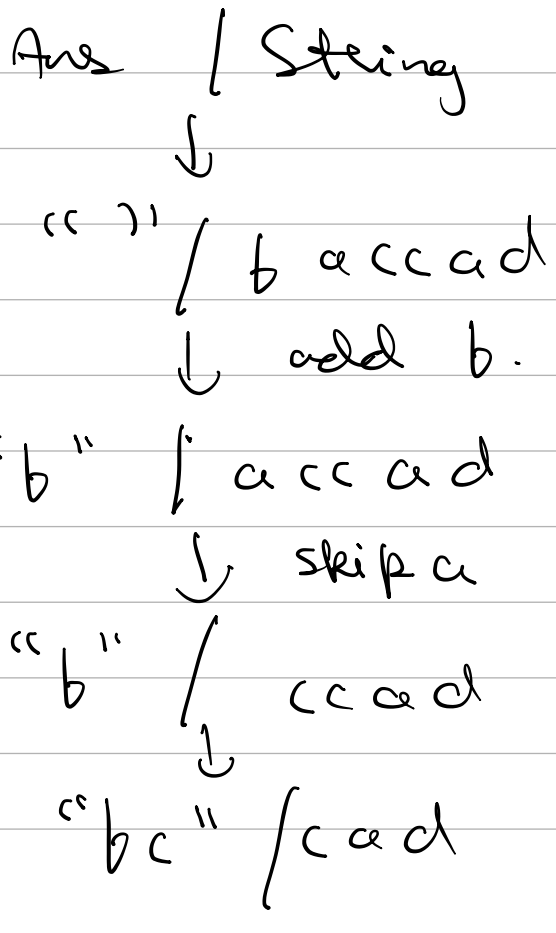
Soln:- ___ recursion

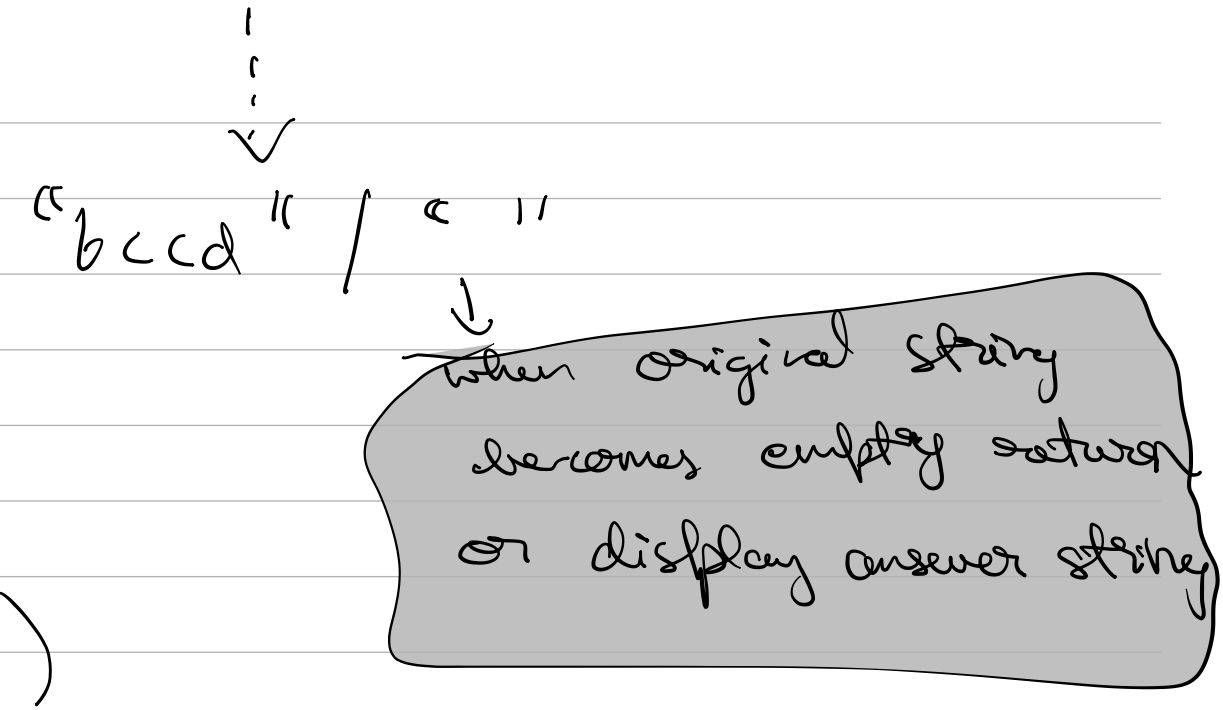
Used to pass answers to upper calls.

① Pass the ans string in argument.

② Create ans variable in function bod.

Logic:-





(OR)

Recursion Implementation:-

(b a ccad)

ch = b +

(accad)

[ch = "" +]

(ccad)

[ch = c +]

(cad)

[ch = c +]

(ad)

[ch = "" +]

(d)

[ch = 'a' + ...]

Terminating Condition

If string = empty. return. ch.

Note:- • substring doesn't change the original array, it creates new array starting from index.

Code:-

• substring(index) → creates a substring starting from 'index' number to end.

```
public class Stream {
    public static void main(String[] args) {
        skip( p: "", up: "baccdah");
    }
```

```
    static void skip(String p, String up) {
        if (up.isEmpty()) {
            System.out.println(p);
            return;
        }

        char ch = up.charAt(0);

        if (ch == 'a') {
            skip(p, up.substring(1));
        } else {
            skip( p: p + ch, up.substring(1));
        }
    }
}
```

p → processed string.
up → unprocessed string.

Another method:-

We return string here, instead of printing it in the function.

```
    }

    static String skip(String up) {
        if (up.isEmpty()) {
            return "";
        }

        char ch = up.charAt(0);

        if (ch == 'a') {
            return skip(up.substring(1));
        } else {
            return ch + skip(up.substring(1));
        }
    }
```

Q). Skip a given string.

Eg:- apple is good.

Skip = "appli"

Ans:- is good.

Eg:- 2]. is apple good.

skip: "apple"

Ans:- is good.

Logic:- is apple good.

↓

[i + ↓]

↓

sapplegood

↓

[s + ↓]

↓

applegood

engtnoapple.

[up.substring(5)] → apple good

(starts with apple so, skip).

[f + ↓]

↓

[g + ""]

↓

empty.

Code:-

```
static String skipApple(String up) {
    if (up.isEmpty()) {
        return "";
    }

    if (up.startsWith("apple")) {
        return skipApple(up.substring(5));
    } else {
        return up.charAt(0) + skipApple(up.substring(1));
    }
}
}
```

→ when 'apple' is encountered at beginning pass the 'up' as starting from 5th index.

Q]. Skip a string if it is not the required string:-

Logic:- is app good.

(i) Check for apple, if not present but only 'app' is present,

Skip 'app', as it is not required.

Code:-

```
static String skipAppNotApple(String up) {
    if (up.isEmpty()) {
        return "";
    }
    if (up.startsWith("app") && !up.startsWith("apple")) {
        return skipAppNotApple(up.substring(3));
    } else {
        return up.charAt(0) + skipAppNotApple(up.substring(1));
    }
}
```

→ 'app" length.

Skips 'app' if apple is not present

Subsets

☆ Permutation and combinations.

[3, 5, 9] → [3], [5, 9], [3, 9]

. . . — — —

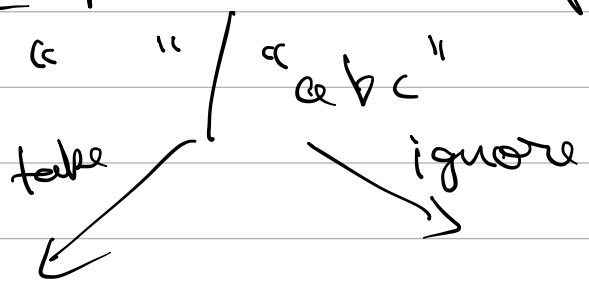
Recursion and Iteration methods to obtain subsets

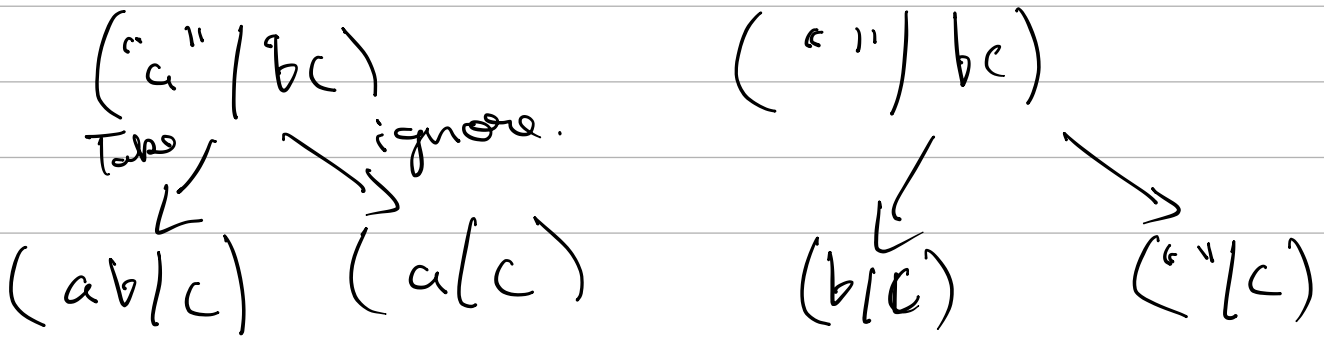
str = "abc"

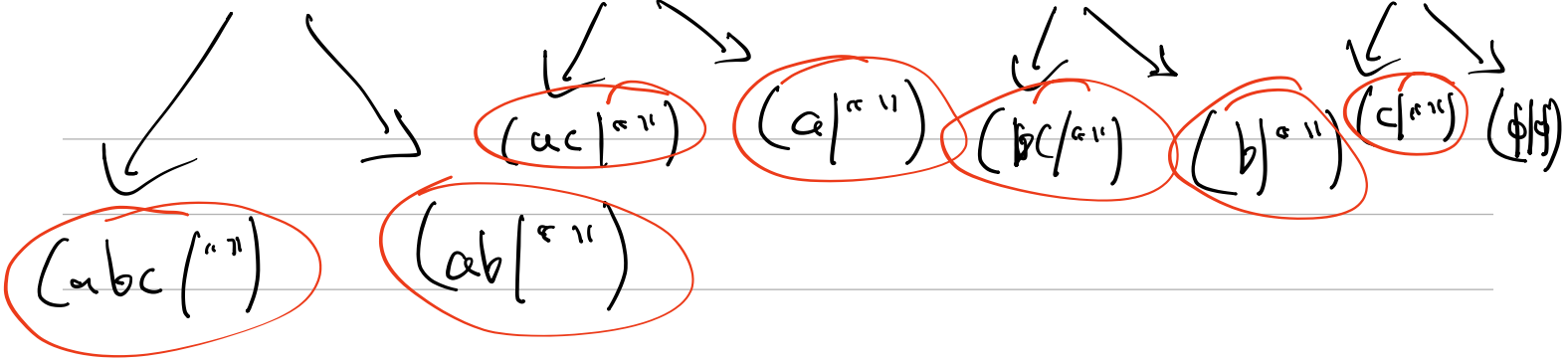
ans: [a, b, c, ab, ac, bc, abc].

☆). This pattern of taking some elements & removing some is known as subset pattern

Processed ← P UP → unprocessed.







Logic:-

*]. Whenever UP becomes empty, print 'P' and move ahead.

Code:-

```
public class SubSeq {
    public static void main(String[] args) {
        subseq( p: "", up: "abc");
    }
```

```
    static void subseq(String p, String up) {
        if (up.isEmpty()) {
            System.out.println(p);
            return;
        }

        char ch = up.charAt(0);

        subseq( p: p + ch, up.substring(1));
        subseq(p, up.substring(1));
```

take

ignore

}

Output:-

abc
ab
ac
a
bc
c

See recursion tree for output logic from left to right.

If we want to return an arraylist of substrings

Two methods we know:-

(1) One pass arraylist as argument and add output to it.

(2). Create new arraylist inside function call and return that. which contains answer.

Code:-

```
static ArrayList<String> subseqRet(String p, String up) {
    if (up.isEmpty()) {
        ArrayList<String> list = new ArrayList<>();
        list.add(p);
        return list;
    }
    char ch = up.charAt(0);
    ArrayList<String> left = subseqRet( p: p + ch, up.substring(1));
    ArrayList<String> right = subseqRet(p, up.substring(1));

    left.addAll(right);
    return left;
}
```

when UP is empty.

stores left side processed string. list

stores right side processed string. list

left contains final answer at last function call return.

For example :- left stores -> "abc"
right -> "ab"
we add left to right = [abc, ab]

This is returned at leftmost side

Simultaneously :- In other calls. left and right list will store the left and right outcomes of substrings and keeps adding them.
at last we get [abc, ab, ac, a bc, c].

See recursion tree for explanation !

We get [ac, a] after adding. left = "ac" and right = "a".

In above level, left = [abc, ab] right = [ac, a] adding we get.

[abc, ab, ac, a] - - - - -
This recursively goes on!

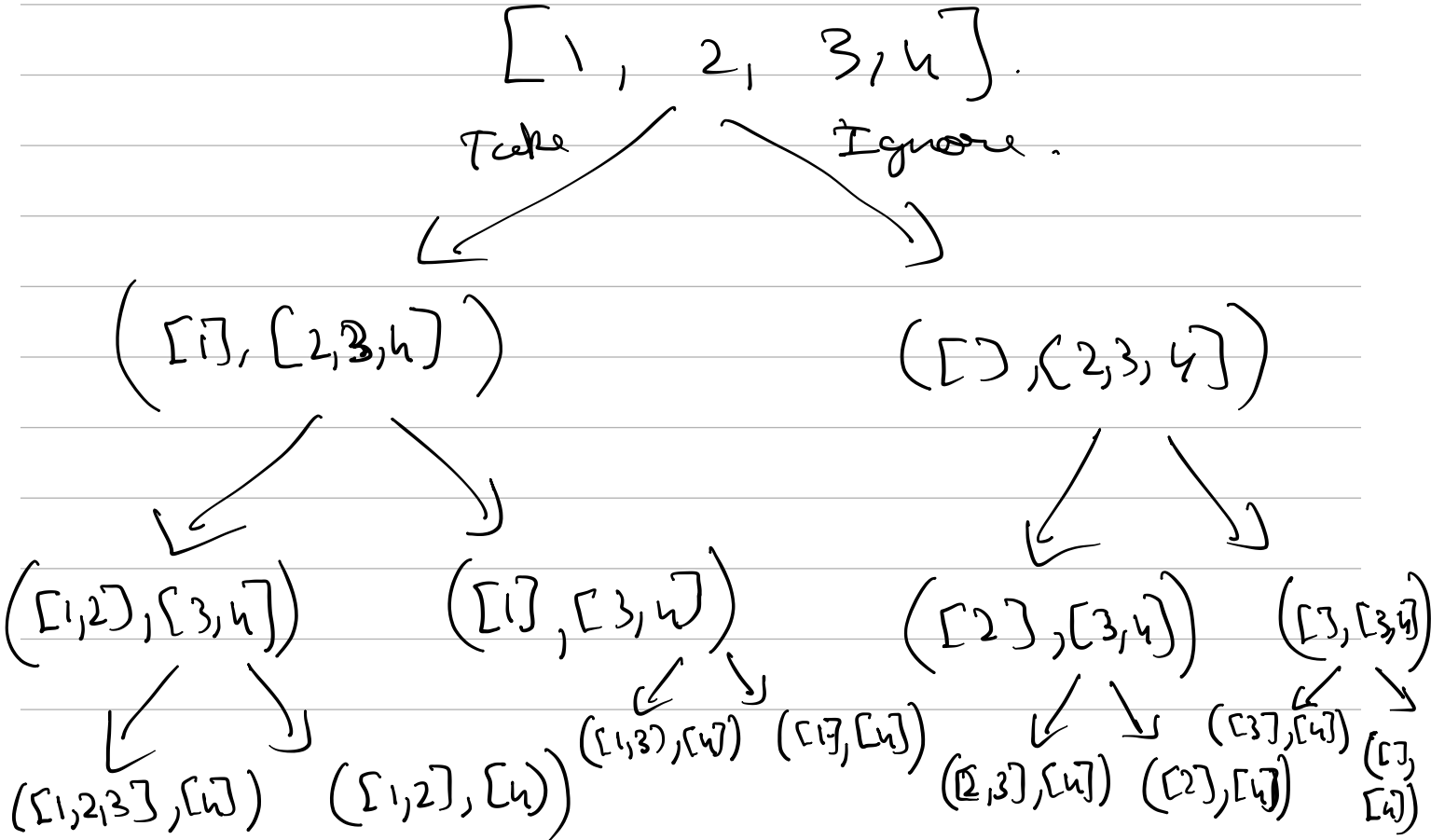
Print ASCII value of a character

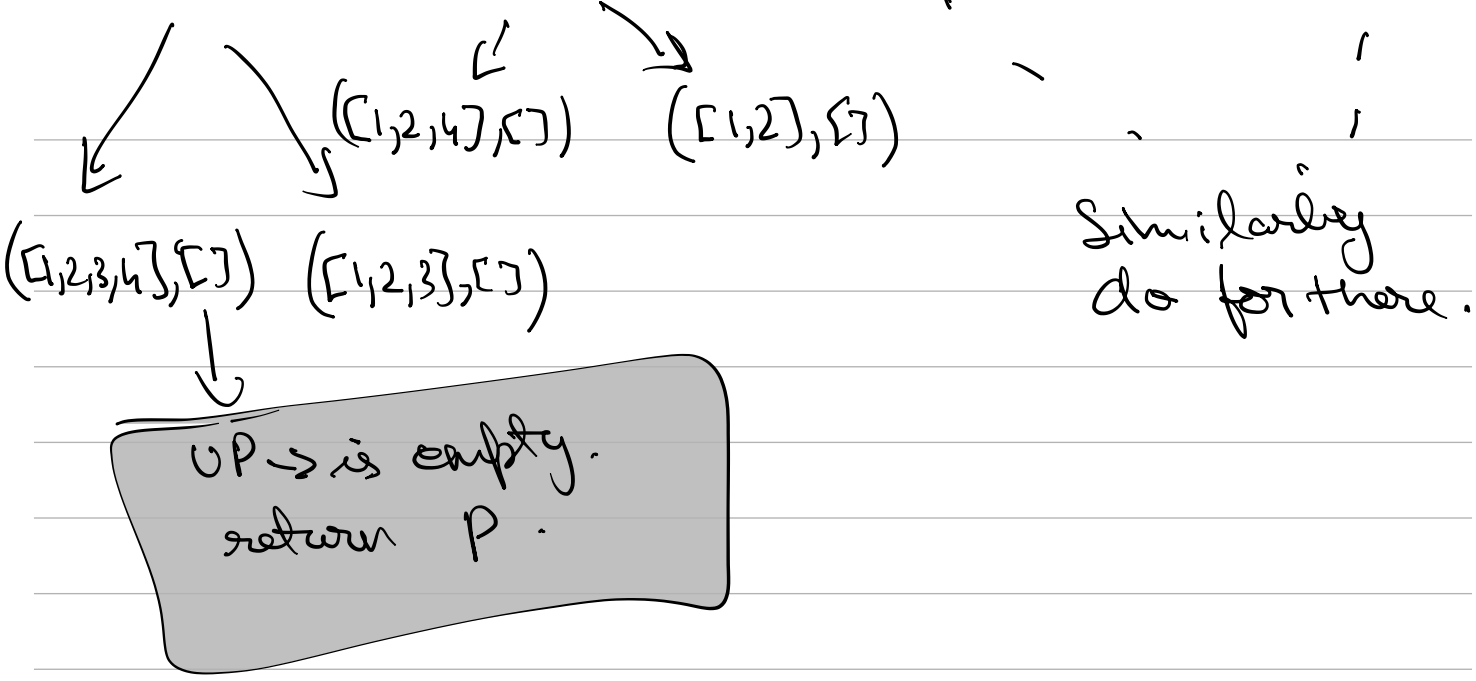
```
public class Ascii {
    public static void main(String[] args) {
        char ch = 'a';
        System.out.println(ch + 0);
    }
}
```

→ Zero is added to convert ch to integer else, only 'a' will be printed.

Output:- 97.

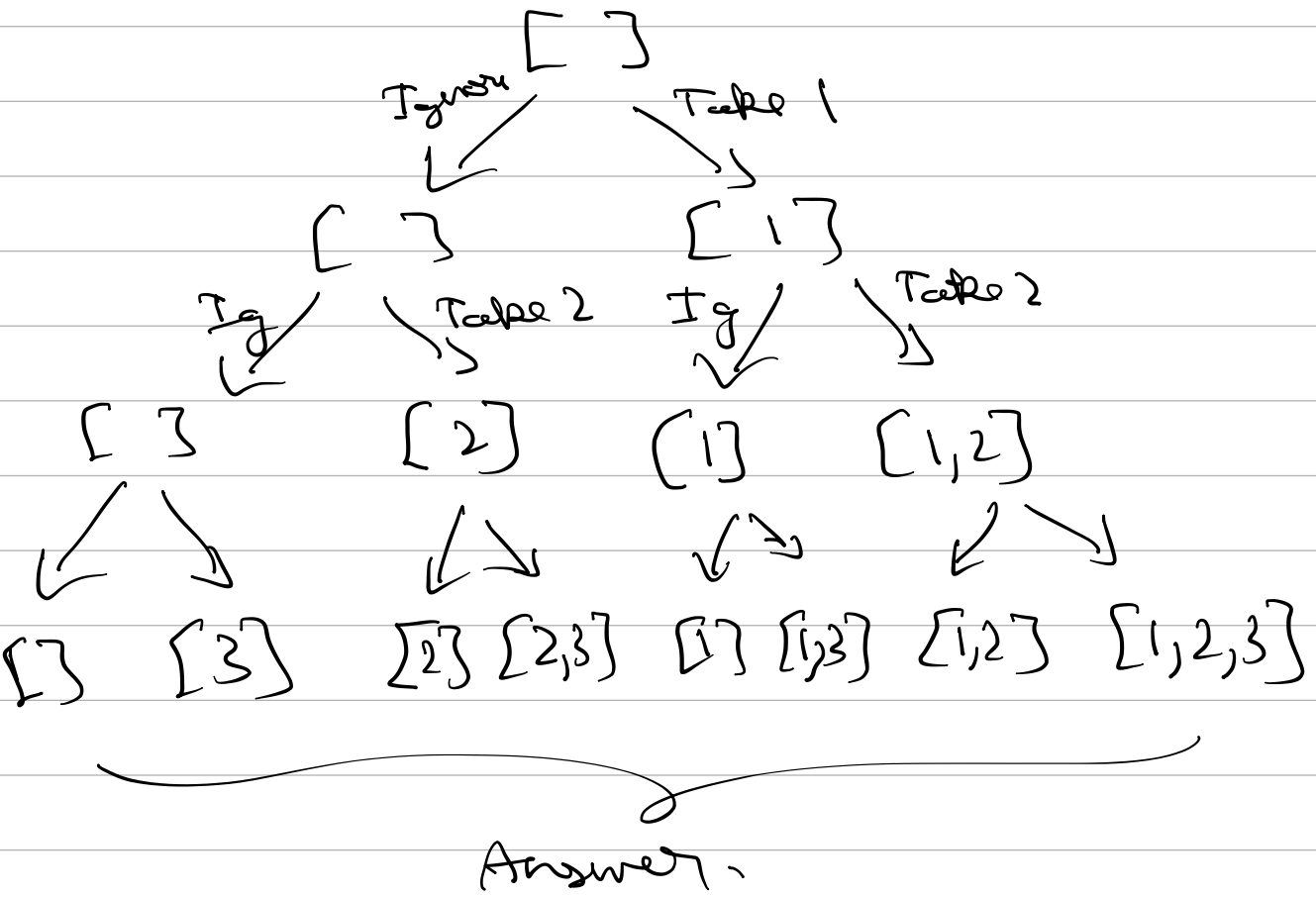
Obtaining subset in Array





Program Logic :-

Array = 1, 2, 3.



*). We observe that we need two array List → Internal and outer.

At first Iteration:-

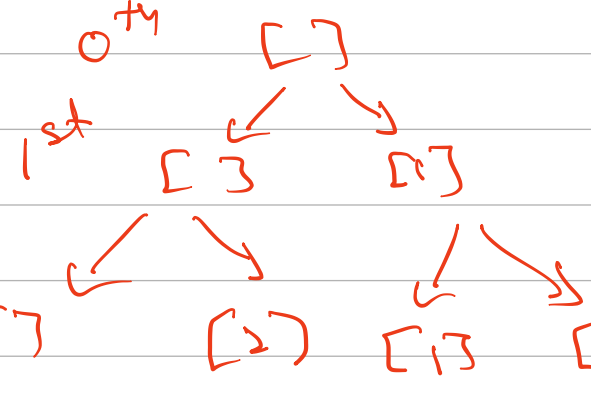
[ ], [1] → Individually are internal list

[[ ], [1]] → Outer list.

*). Observe at each iteration, the outer list size is getting doubled.

i.e, we are creating copy of old outer list in above iteration, before we add new element.

For example:-



Here [ ] [ ] are copied from 1st iteration to 2nd Iteration. and then new element '2'

is added to [ ] and [1] to get [2] and [1,2].

Code:-

```java
public class SubSet {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3};
        List<List<Integer>> ans = subset(arr);
        for (List<Integer> list : ans) {
            System.out.println(list);
        }
    }
```

```java
static List<List<Integer>> subset(int[] arr) {
    List<List<Integer>> outer = new ArrayList<>();

    outer.add(new ArrayList<>());

    for (int num : arr) {
        int n = outer.size();
        for (int i = 0; i < n; i++) {
            List<Integer> internal = new ArrayList<>(outer.get(i));
            internal.add(num);
            outer.add(internal);
        }
    }
    return outer;
}
```

→ initially outer list size 1.

arr = 1, 2

Iteration 1:.

num = 1

n = 1

i = 0 ; i < 1

Internal = [ ]

internal = 1 . (num = 1)

Outer = [[ ], [1]]

↓ outer already was added with [ ]

Iteration 2:- num = 2

n = 2

i = 0; i < 2 > 2 times (0, 1).

Internal = [] (we are setting internal to outer.get(0) i.e [])

Internal = [2]

outer = [[], [1], [2]]

2nd Time

Internal = [1] (we are setting it to outer.get(1) i.e. [1])

Internal = [1, 2].

Outer = [[], [1], [2], [1, 2]]

Like this Outer arraylist has all the subsets.

Output for code:-

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

Complexity Analysis

Basically time complexity is no. of levels × time taken at each level

No of levels :- N (For eg:- 1, 2. we have 2^i iterations)

time taken at each level :- observe at each iteration, the number of subsets is doubled.

2^N.

Time complexity :- $O(N \times 2^N)$ → no of subsets.

Space complexity :- = $O(2^N \times N)$

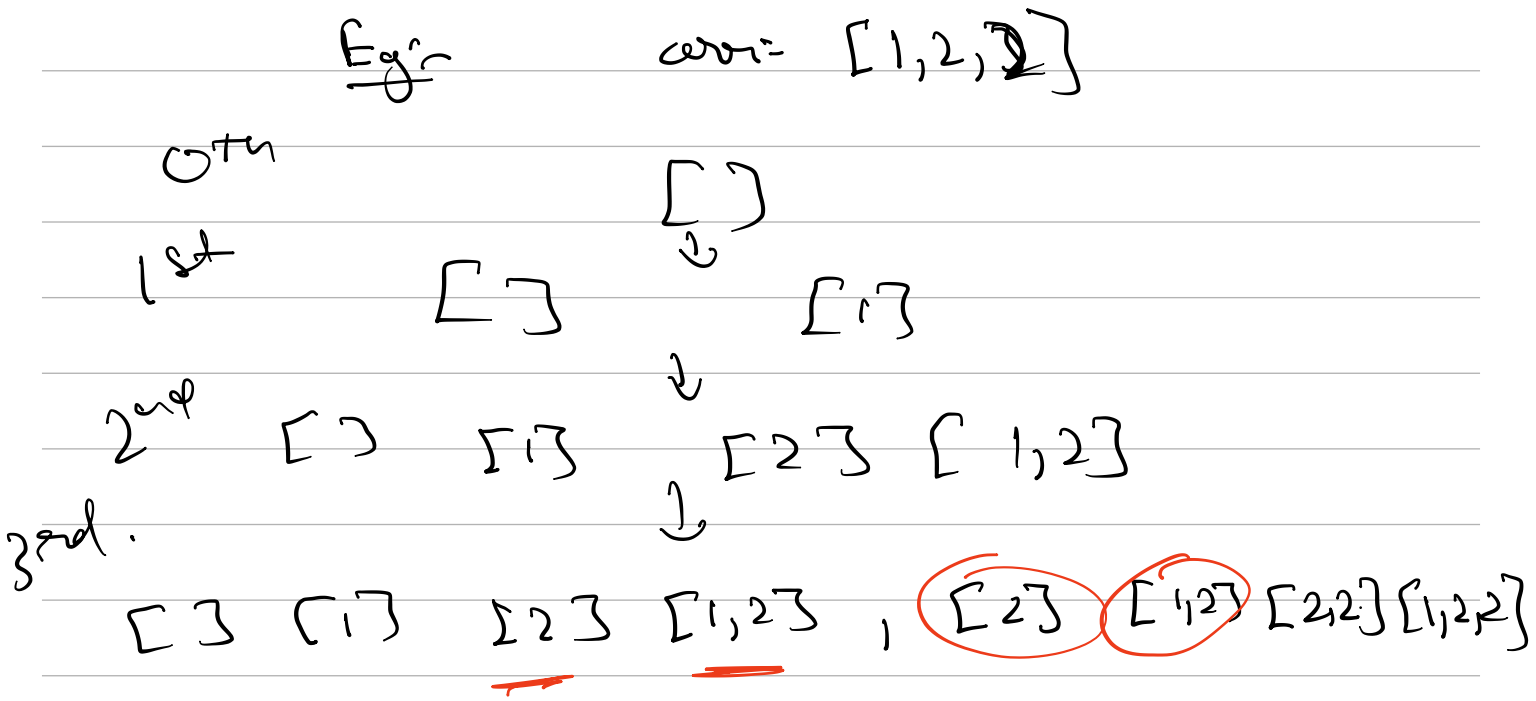
2^N → The answer has 2^N subsets.

N → each subset takes N space

[1] → takes 1 space

[1,2] → takes 2 space as N=2

What to do if duplicate elements are present



● -> Duplicate elements will be obtained in the outer arraylist if same logic is applied.

To solve this

☆). For this the array should be sorted, so that the duplicate elements appear adjacently.

*). Use a start and end pointer to point starting and ending internal list to choose proper internal list to add new array item.

Logic:- If (i > 0 && arr[i] == arr[i-1])
start = end + 1;

// at previous iteration end will be pointing to the outer.size() - 1.

Eg:- After 2nd iteration, end = 1 (ie [1])
↓
end will be assigned before [2] and [1,2] is added to outer, so end = 1 [sized 1st iteration]

So, start will be end+1 (i.e 2).
Outer[2] is [2].

So duplicate element 2, will be added to [2] to get [2,2] and [1,2] to get [1,2,2]

This avoids [2] and [1,2] to get printed again

```java
public class SubSet {
    public static void main(String[] args) {
        int[] arr = {1, 2, 2};
        List<List<Integer>> ans = subsetDuplicate(arr);
        for (List<Integer> list : ans) {
            System.out.println(list);
        }
    }

    static List<List<Integer>> subsetDuplicate(int[] arr) {
        Arrays.sort(arr);
        List<List<Integer>> outer = new ArrayList<>();
        outer.add(new ArrayList<>());
        int start = 0;
        int end = 0;
        for (int i = 0; i < arr.length; i++) {
            start = 0;
            // if current and previous element is same, s = e + 1
            if (i > 0 && arr[i] == arr[i-1]) {
                start = end + 1;
            }
            end = outer.size() - 1;
            int n = outer.size();
            for (int j = start; j < n; j++) {
                List<Integer> internal = new ArrayList<>(outer.get(j));
                internal.add(arr[i]);
                outer.add(internal);
            }
        }
        return outer;
    }
}
```

→ Check for duplicate element in sorted array.

↓ for each iteration of selecting internal list for adding array element start and end is initialized.

If duplicate is not present, start will be 0th list inside outer list always.

*). Only when duplicate is encountered, start will be end+1.