Recursion Notes

Saturday, October 8, 2022 8:22 AM

RECURSION

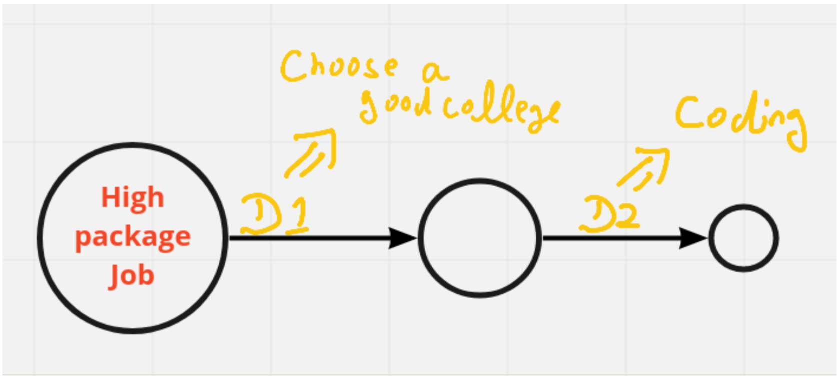
INTRODUCTION
Recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem.

What we gonna learn:-

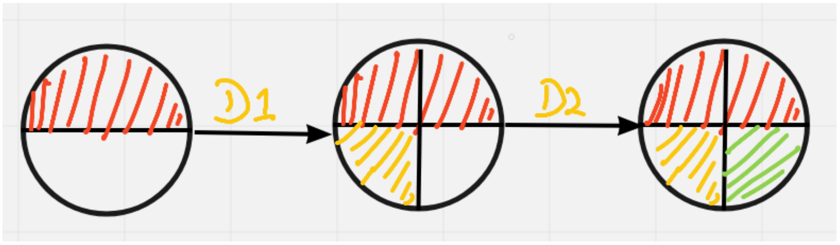
1. Make Input Smaller! But why?
2. Recursion - Decision Space
3. Recursive Tree - Soul of Recursion [Important]
4. 2 steps to solve any Recursive problem

- **Make Input Smaller! But Why?**
  We don't deliberately smaller the input. The input becomes smaller automatically!. That's the power of recursion

Let's understand it with an example. How the input is getting smaller.
Let's say you want to get a higher package job. For that you have to take some decision's. **The 1st decision [D1]** you take is to get into a good college. By doing that your problem becomes small. Now **2nd decision [D2]** you took is to learn coding [Data Structure & Algorithm]. Now your problem had become more smaller. Similarly upon taking several decision's you will land a high package job!



Similar thing happen's with our Iutput. Our Input is bigger in the starting but taking few decision's it becomes smaller & we finally get our result!



So, what you learn from above is, our **decision** has to be a **primary goal** & on the basis of that. Input becomes smaller automatically

- **Recusion - Decision Space**
  Whenever we have to think about space. Recusion is a good choice.

Now how can we identify that whether a problem statement use recursion or not.

For that we have to look at the problem, that we have given some Choice's & some Decision. If that's the case then it's a recursion problem.

```
-----------------------------
     Choices + Decision
-----------------------------
```

- **Recursion Tree**

Let's undrestand with an example,
-> Get subset of "abc". Now what is subset : "subset is a set which contains all the elements of given set"
In our "abc", abc subset will be :-

```
" "     a        ab        abc
    b        bc
c        ac
```

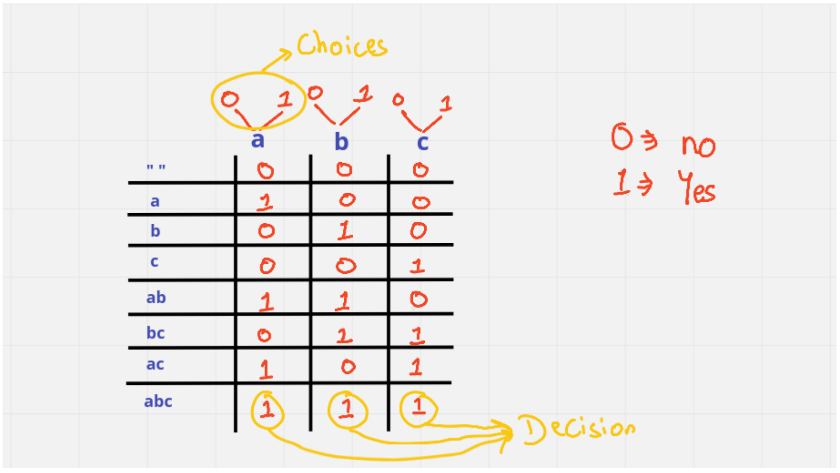
Don't worry I'm not teaching you right now, how to code & print subset. But later on we will learn.

So, why I'm teaching you subset so that you can learn how we are solving this problem just by taking some **choices & decision.**

So, In this example what choices can we have for **"abc"** for a we have two choices whether to consider or not.

```
      a              b               c
  /  \           /   \           /   \
Yes   No       Yes    No       Yes    No
```

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



	a	b	c
" "	0	0	0
a	1	0	0
b	0	1	0
c	0	0	1
ab	1	1	0
bc	0	1	1
ac	1	0	1
abc	1	1	1

So, if you see we are taking choices whether to **have or not. [0 or 1]** And by these choices we are making decision's. And because of that. Our Input is getting smaller.

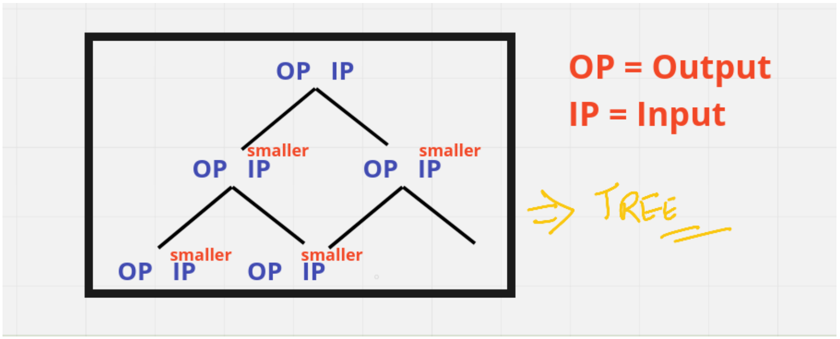
Now let's understand, what is **Recursion tree?**

Again, let's understand with example we have to get subset of "ab"

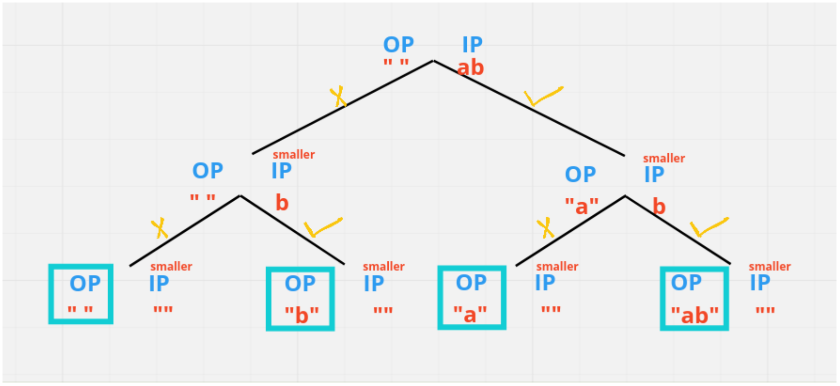
```
|  " "  |   a -> 0 b -> 0
|  a    |   a -> 1 b -> 0
|  b    |   a -> 0 b -> 1
|  ab   |   a -> 1 b -> 1
```

So, instead of representing in **a->0b->0** 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"**



Let's understand this tree with an example as well. Get **subset of "ab"**



When **smaller"IP"** becomes empty **return the tree & get's the answer**

- **2 steps to solve Recursion Problem**

1. Design a recursive tree
2. F8ck the problem a.k.a **Write the code**

```
RECUSRION is every-where
```

We use recursion in most of the places.

Data Structure :-

- Array/String
- Heap
- Stack
- Tree
- Graph [DFS]
- Linked List

Recursion is backbone of :- **DP / BackTracking / Divide n Conquer**

Question's we gonna solve :-

```
-> Print 1 to N
-> Print N to 1
-> Height of a Binary Tree
-> Sort an array

-> Sort an stack
-> Delete Middle Element from A Stack
-> Reverse A Stack
-> Kth Symbol In Grammar

-> Tower Of Hanoi
-> Print Subsets
-> Print Unique Subsets & Variation
-> Permutation with Spaces

-> Permutation with Case Change
-> Letter Case Permutation
-> Generate All Balanced Parenthesis
Hypothesis - Induction - Base Condition
```

Okay, now you will ask what is this, **Hypothesis-Induction-Base Condition**, well I call it as **IBH method**
I will get on to your point, but before that let's understand, in recursion there are several approaches to solve a Recursive problem
But we will learn 2 ways to solve a problem.

- Recursive Tree -> IP/OP method **[work's only if you know Decision]**
- Base Condition-Induction-Hypothesis -> [IBH] **[only work when you don't have given choices & make IP smaller]**

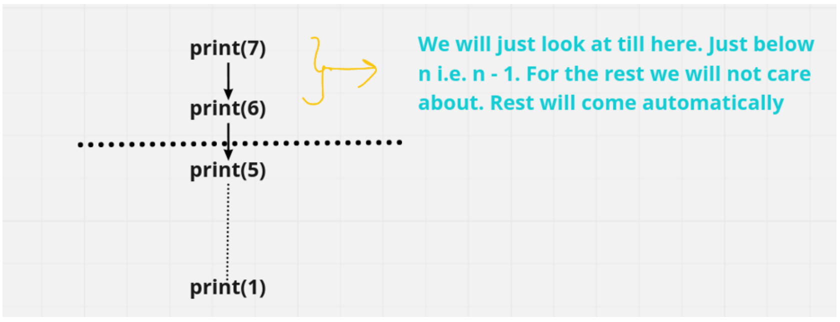
```
Let's understand what are Hypothesis, Induction & Base Condition
```

- **Hypothesis :- Work's like recursion tree**
- **Induction :- let's say you have to print 1 to n all numbers, so the magic happen's called induction**
- **Base Condition :- Work's only if you have "smallest valid Input" OR "smallest invalid iput"**

Let's understand our IBH method with one question

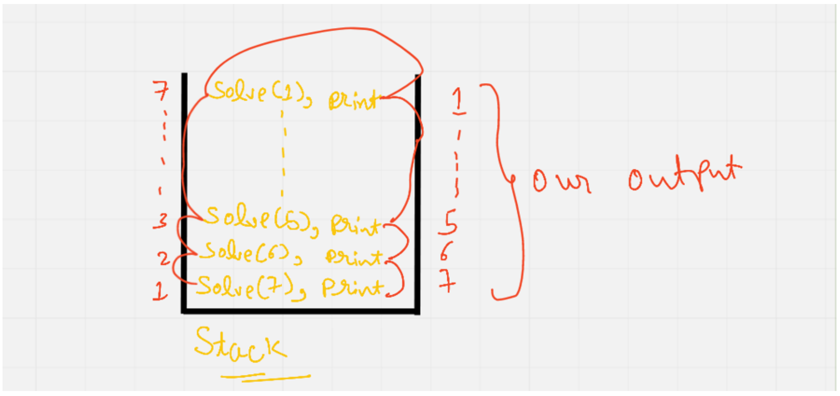
Que. Print 1 to n

Problem Statement :-
In this let say we have input 7 and we want to print all possible no.s from **1 to 7**. And that will be 1,2,3,4,5,6,7


print(7)
print(6)
print(5)
print(1)
We will just look at till here. Just below n i.e. n - 1. For the rest we will not care about. Rest will come automatically

In this problem our **hypothesis** is to look at the no just below 7 i.e. 6 & work wtill here. Rest will be come up automatically..
Now, you will ask automatically rest will come but **how?**. Okay let me explain then, so whenever we are making call to our let say **solve function**. So, it will store the value in the stack becuase firstly we are making function call of **solve(n - 1)** which is our **hypothesis** & then we are printing them out.

Once it make a call till 0. So, 0 is our smaller valid input possible & this will be our **base condition**. Once our function hit's the base condition, we will start printing it back & that will be our **induction**



So, if you carefully observe we are just making our input smaller each n every time by the help of **Hypothesis** that we had created. And once input become so smaller and hit our **base condition**, we will get our answer and that magic is **induction**.

`So, we are only worried about making our input 1 step smaller, for the rest input it will work magically. And that magic is recursion`

I hope now you have understand, that how IBH method helps in making Input smaller.

Now let's see how we code it up,

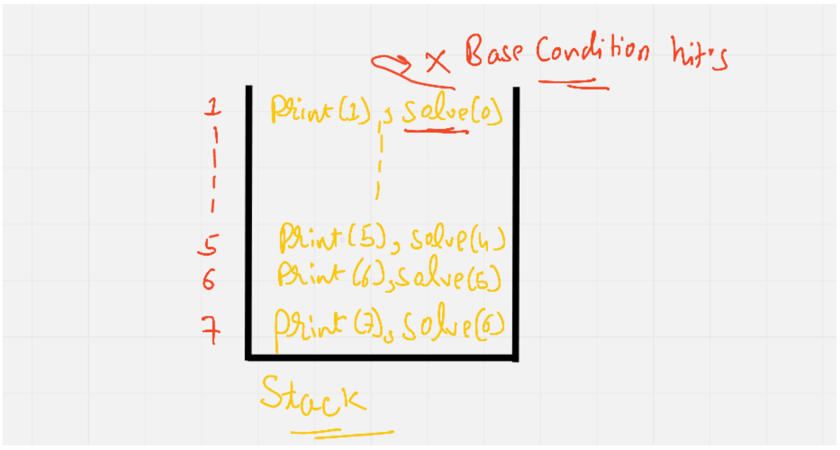
```
import java.util.*;
class Main {
  public static void solve(int n){
    // Base condition
    if(n == 0) return;
    solve(n - 1); // Hypothesis
    System.out.println(n); // Induction
  }
}
```

Que. Print n to 1

Problem Statement :-
In this let say we have input 7 and we want to print all possible no.s from **7 to 1**. And that will be 7,6,5,4,3,2,1

Okay this question is almost similar to the previous question we have done. In this one we have to print all of them in reverse order.

So, what difference you can see over here is in the previous question, first we are writing **Hypothesis** then making call to our **indcution** to print it. But in this question we dont store all of them in our stack. First we will make a call to print it, then we will move a call to our **solve function (n - 1)**. And once we hit the **base condition**, we will return our output.



Now let's see how we code this:

```
import java.util.*;
class Main {
  public static void solve(int n){
    // Base condition
    if(n == 0) return;
    System.out.println(n); // Induction
    solve(n - 1); // Hypothesis
  }
}
```

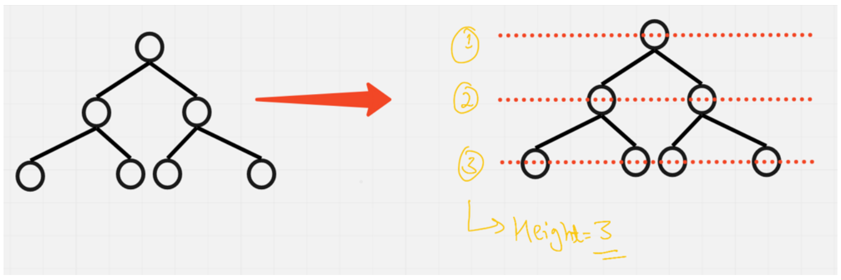
Alright guy's now you get a good understanding of how **IBH method** works. Now let's practice more question's and do some rock n roll

Que. Height Of A Binary Tree

Problem Statement:-
Given the root of a binary tree, return its maximum depth. In input we have given tree node values & in **output** we have to return it's maximum height.

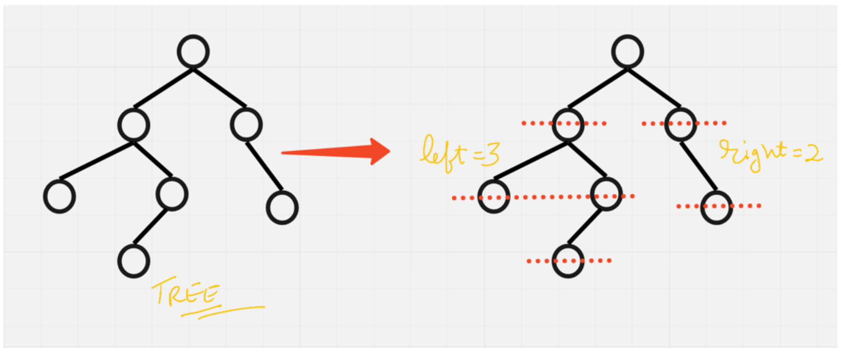
Let's understand this with an tree example.

Say we have given something like this :-



To calculate it's height, we draw line and found out that it's **height is 3**

Now, by `using recursion how can we find it`. So, we know the root & it's child will be find by **Hypothesis**. So we go deep inside to it's left, then to it's right & from there we will get the deepest root to child value of it's left & right. We will compare both of them and that will be our **Induction** and which one has the greatest value will be added to 1, as root will count as well. Now in this the smallest valid Input could be if root is null & that;s our **Base condition**. `Let's understand visually`



In the above diagram, max height we get is 3. But we have to add root as well. So, it will be **3 + 1 = 4**

Now let's code it up :-

```
class Solution {
    public int maxDepth(TreeNode root) {
        // Base Condition
        if(root == null) return 0;
        // Hypothesis
        int left = maxDepth(root.left);
        int right = maxDepth(root.right);
        // Induction
        return Math.max(left, right) + 1;
    }
}
```

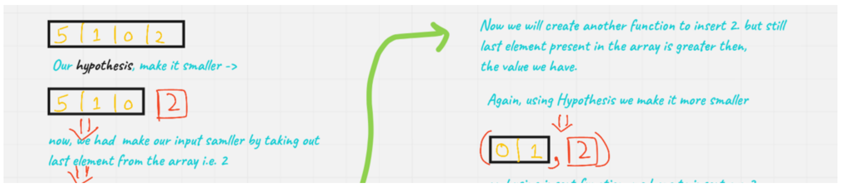
Que. Sort An Array Using Recursion

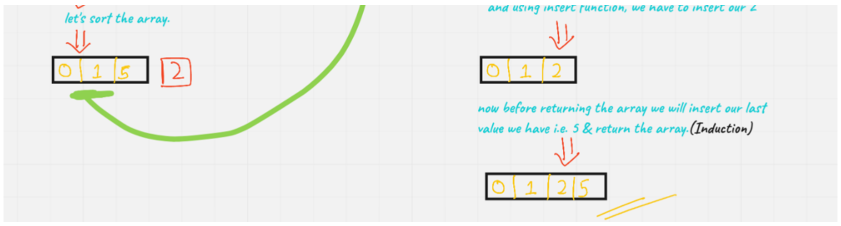
Problem Statement :-
We have given an array list, we have to sort the array in asscending order.
Input : [5,1,0,2] & **Output :** [0,1,2,5]

Firstly, your thought is. **Why we are doing this with recusrion** as, we can do that using **Sorting or Iterative way**.

`I say, you will learn from this a lot.`

arr = [5,1,0,2] sort this array
`Understand it visually:`


5 | 1 | 0 | 2
Our hypothesis, make it smaller ->
5 | 1 | 0
2
now, we had make our input samller by taking out last element from the array i.e. 2
Now we will create another function to insert 2. but still last element present in the array is greater then, the value we have.
Again, using Hypothesis we make it more smaller
0 | 1
2


and using insert function, we have to insert our 2
let's sort the array.
0 1 5
2
0 1 2
now before returning the array we will insert our last value we have i.e. 5 & return the array.(Induction)
0 1 2 5

Okay, so we have figure out our **Hypothesis & Induction**
Base Condition left. Our smallest valid input could be if **Array size is 1**, then we simply **return Array**

Now let's code it up :-

```
import java.util.*;
class Main { // Main function call
    public static void main(String[] args) {
        ArrayList arr = new ArrayList(
            Arrays.asList(5,1,0,2)
        );
        arr = sort(arr);
        System.out.println(arr);

    }
// Actual work start's from here
    public static ArrayList sort(ArrayList<Integer> arr){ // sort
function
// Base Condition
        if(arr.size()==1){
            return arr;
        }
// Hypothesis
        int temp = arr.get(arr.size()-1); // getting 2 out
        arr.remove(arr.size()-1); // removing it from array
        sort(arr); // sorting the array from [5,1,0] -> [0,1,5]

        insert(arr,temp);
        return arr;
    }
    public static ArrayList insert(ArrayList<Integer> arr,int temp){
// insert function
// Base Condition
        if(arr.size() == 0 || temp >= arr.get(arr.size() - 1)){ //
checking if let say we have 6 in temp & 6 is greater then 5
          arr.add(temp); // we will simply add it into our array
          return arr; // and return it
        }
// Hypothesis
        int val = arr.get(arr.size() - 1); // getting 5 out
        arr.remove(arr.size() - 1); // removing 5 from array -> [0,1]
        insert(arr, temp); // insrting 2 to [0,1] which becomes ->
[0,1,2]

// Induction
        arr.add(val); // adding 5 to [0,1,2] which becomes ->
[0,1,2,5]
        return arr;
    }
}
```

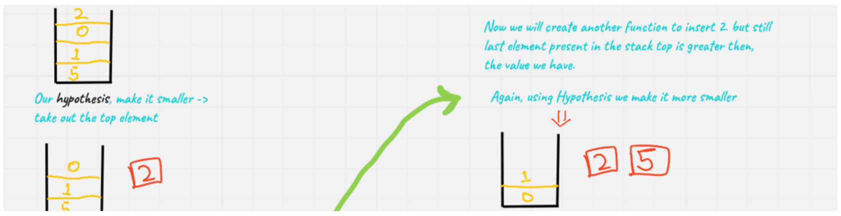
Que. Sort A Stack

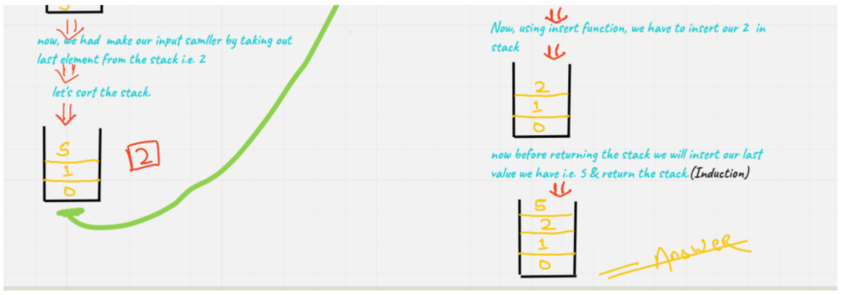
Problem Statement
Given a stack, sort it in asscending order

```
Input:            Output:
2  <--- Top              5  <--- Top
0                        2
1                        1
5                        0
```

Alright, Guy's this problem is much similar to "Sort an Array". `I mean it's almost to that`

Let's understand it visually,


2
0
1
5
Our hypothesis, make it smaller ->
take out the top element
0
1
5
2
Now we will create another function to insert 2. but still last element present in the stack top is greater then, the value we have.
Again, using Hypothesis we make it more smaller
1
0
2
5


now, we had make our input samller by taking out last element from the stack i.e. 2
let's sort the stack.
5
1
0
2
Now, using insert function, we have to insert our 2 in stack
2
1
0
now before returning the stack we will insert our last value we have i.e. 5 & return the stack.(Induction)
5
2
1
0
Answer

Okay, so we have figure out our **Hypothesis & Induction**
Base Condition left. Our smallest valid input could be if **Stack size is 1**, then we simply **return stack**

Now let's code it up :-

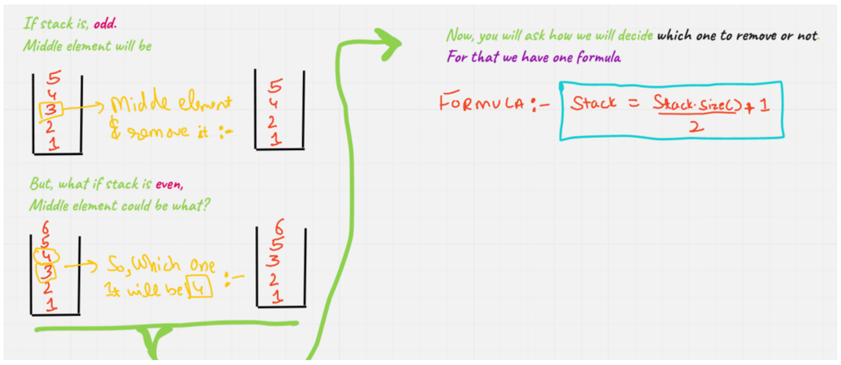
```
class Main {
  // Main function call
  public static void main(String[] args) {
    Stack<Integer> st = new Stack<>();
    st.push(5);
    st.push(1);
    st.push(0);
    st.push(2);

    System.out.println("Stack before sorting:");
    for (Integer s : st) {
      System.out.print(s + " ");
    }

    System.out.println("\nStack after sorting:");
    Stack<Integer> sorted = sortStack(st);
    for (Integer i : sorted) {
      System.out.print(i + " ");
    }
  }
  // Actual work start's from here
  public static Stack<Integer> sortStack(Stack<Integer> st){
    if(st.size() == 1) return st; // Base Condition
    // Hypothesis
    int temp = st.pop(); // getting 2 out
    sortStack(st); // sorting the stack from [5,1,0] -> [0,1,5]

    return insertElementAtStack(st, temp);
  }
  public static Stack<Integer> insertElementAtStack(Stack<Integer> st, int
temp){
    // Base Condition
    if(st.size() == 0 || temp >= st.peek()){ // checking if let say we have 6
in temp & 6 is greater then 5
      st.push(temp); // we will simply add it into our stack
      return st;
    }
    // Hypothesis
    int val = st.pop(); // getting 5 out stack becomes -> [0,1]
    insertElementAtStack(st, temp); // insrting 2 to [0,1] which becomes ->
[0,1,2]
    // Induction
    st.push(val); // adding 5 to [0,1,2] which becomes -> [0,1,2,5]
    return st;
  }
}
```

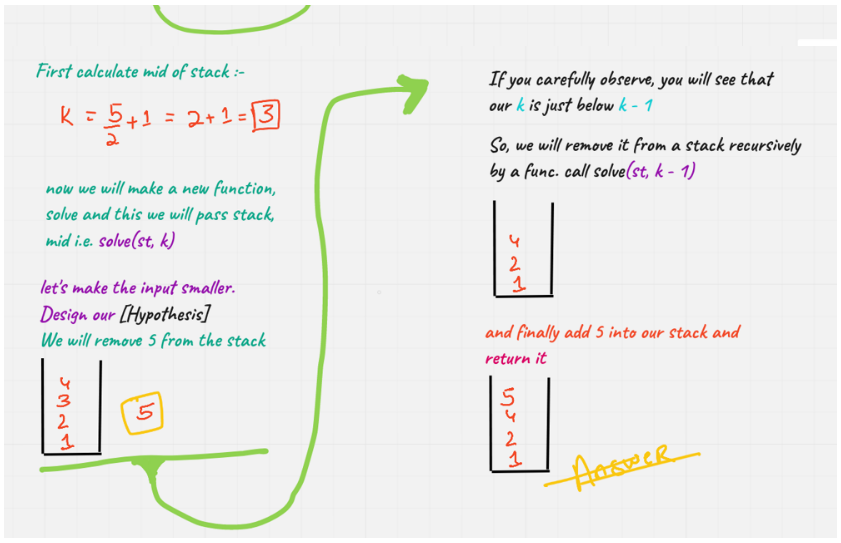
Que. Delete Middle Element Of a Stack

Problem Statement :-
Input : Stack[] = [1, 2, 3, 4, 5]
Output : Stack[] = [1, 2, 4, 5]

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

Let's understand it's visually :


If stack is, odd.
Middle element will be
5
4
3
2
1
Middle element & remove it :-
5
4
2
1
But, what if stack is even,
Middle element could be what?
6
5
4
3
2
1
So, Which one It will be 4 :-
6
5
3
2
1
Now, you will ask how we will decide which one to remove or not.
For that we have one formula
FORMULA :-
Stack = Stack.Size() + 1 / 2



First calculate mid of stack :-
K = 5/2 + 1 = 2 + 1 = 3
now we will make a new function, solve and this we will pass stack, mid i.e. solve(st, k)
let's make the input smaller.
Design our [Hypothesis]
We will remove 5 from the stack
4 3 2 1
5
If you carefully observe, you will see that our k is just below k - 1
So, we will remove it from a stack recursively by a func. call solve(st, k - 1)
4 2 1
and finally add 5 into our stack and return it
5 4 2 1
Answer

Okay, so we have figure out our **Hypothesis & Induction** [*Induction = return stack.push(temp)*]
Base Condition left. Our smallest valid input could be if **Stack size is 0**, then we simply **return stack**

Now let's code it up :-

```
class Solution {
    public static void midDel(Stack<Integer> st) {
        // Base Condition
        if(st.size() == 0) return st;

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

        // Hypothesis
        int temp = st.pop(); // removing 5 from stack
        solve(st, k - 1);

// Induction
        return st.push(temp);
    }
}
```

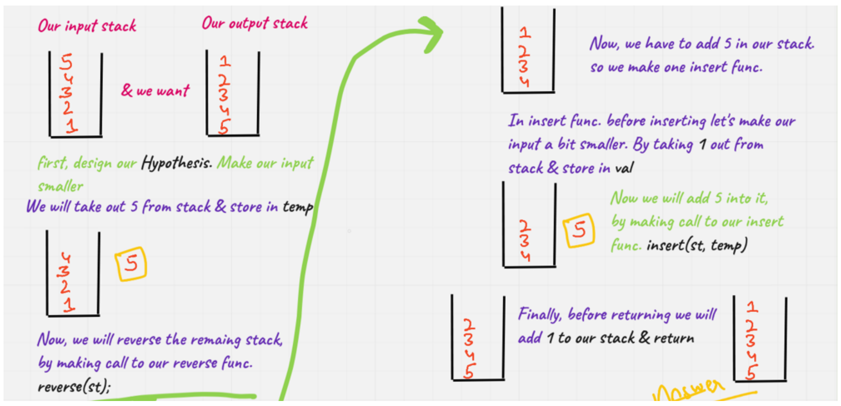
Que. Reverse A Stack

Problem Statement

```
Input :-             Output :-
5  <-- top            1  <-- top
    4                     2
    3                     3
    2                     4
1                     5
```

So, we have given a stack & we have to reverse it.

Let's understand this visually:-



Our input stack
Our output stack
5 4 3 2 1
& we want
1 2 3 4 5
First, design our Hypothesis. Make our input smaller
We will take out 5 from stack & store in temp
4 3 2 1
5
Now, we will reverse the remaing stack, by making call to our reverse func. reverse(st);
1 2 3 4
Now, we have to add 5 in our stack so we make one insert func.
In insert func. before inserting let's make our input a bit smaller. By taking 1 out from stack & store in val
2 3 4
5
Now we will add 5 into it, by making call to our insert func. insert(st, temp)
2 3 4 5
Finally, before returning we will add 1 to our stack & return
1 2 3 4 5
Answer

Okay, so we have figure out our **Hypothesis & Induction** [*Induction = return stack.push(val)*]
Base Condition left. Our smallest valid input could be if **Stack size is 1**, then we simply **return stack**

Now let's code it up :-

```
class Solution {
    public static void reverse(Stack<Integer> st) {
        // Base Condition
        if(st.size() == 1) return st;

        // Hypothesis
        int temp = st.pop() // removing 5 from stack
reverse(st); // reversing the stack
        insert(st, temp); // func. call
    }
    public static Stack<Integer> solve(Stack<Integer> st, int temp) {
        // Base Condition
        if(st.size() == 0){
            st.push(temp);
            return st;
        }

        // Hypothesis
        int val = st.pop(); // removing 1 from stack
        insert(st, temp); // inserting 5 to stack;

// Induction
st.push(val); // inserting 1 to stack
        return st;
    }
}
```