Introduction

Let us start with Maths. In programming, I have observed a one-to-one mapping with Maths. All the mathematical operations and concepts I have known (such as: addition, multiplication, subtraction and many more) are incorporated in programming. Let's dive a bit deeper.

Relation With Mathematics

In mathematics, I had learnt a concept called functions. Let's try to revise functions.
As I can recall, functions were like a machine, there was some input and some output. Each input had one and only one output, though it was possible that two different inputs have similar outputs, but I cannot have two different outputs for a single input. Moreover, these functions had a definition, which defined how they are going to operate on input. For example: f(x)= x * x - 7, where x is input. To solve it for an input, let's say, x=5: f(5)= 5*5 - 7 = 25 - 7= 18. Now, Let's try figure out if there is a programming counterpart for it in mathematics.
In programming, we have a similar concept called functions. If I want to represent my running example of mathematical functions in programming, then I have to write it like this:

```
int f (int x) {
      return x * x - 7;
}
```

To solve it for input x = 5, I have to invoke this function. Invocation is written something like this:

```
f(5);
```

This proves that mathematical functions are one-to-one mapped with programming functions.

As I vaguely recall, there was another concept called "Principle of Mathematical Induction" in mathematics, let's try to figure if I have something in programming for this concept.

Before I dig deeper into it, let's revise. Principle of Mathematical Induction (PMI) is a tool for proving mathematical relations. It has a 3-step process. In step 1, I will prove f(n) for n=1 (here n is some whole number, sometimes I will use n=0). In following step 2, I will assume that this relation holds true for n=k. In the last step, using assumption step, I will prove that this relation holds true for n=k+1. After this 3-step process, I can say that this relation is valid for all whole numbers (or natural numbers). Do I have something related to it in programming?

Exercise for you: Try to prove the following expression, using PMI. $1 + 2 + 3 + \ldots + n = (n * (n+1))/2$

One-to-one mapped to this, there is a concept called recursion in programming. Recursion also has these 3 steps.

1. Step of base case: This is similar to step 1 of PMI, where I prove it for a whole number.
2. Step of Faith: This is similar to step 2 of PMI, where I assume that relation holds true for some n = k.
3. Step of deducing expectation from faith (already derived in step 2): This is similar to step 3 of PMI, where I prove it for n = k+1, by using assumption step.

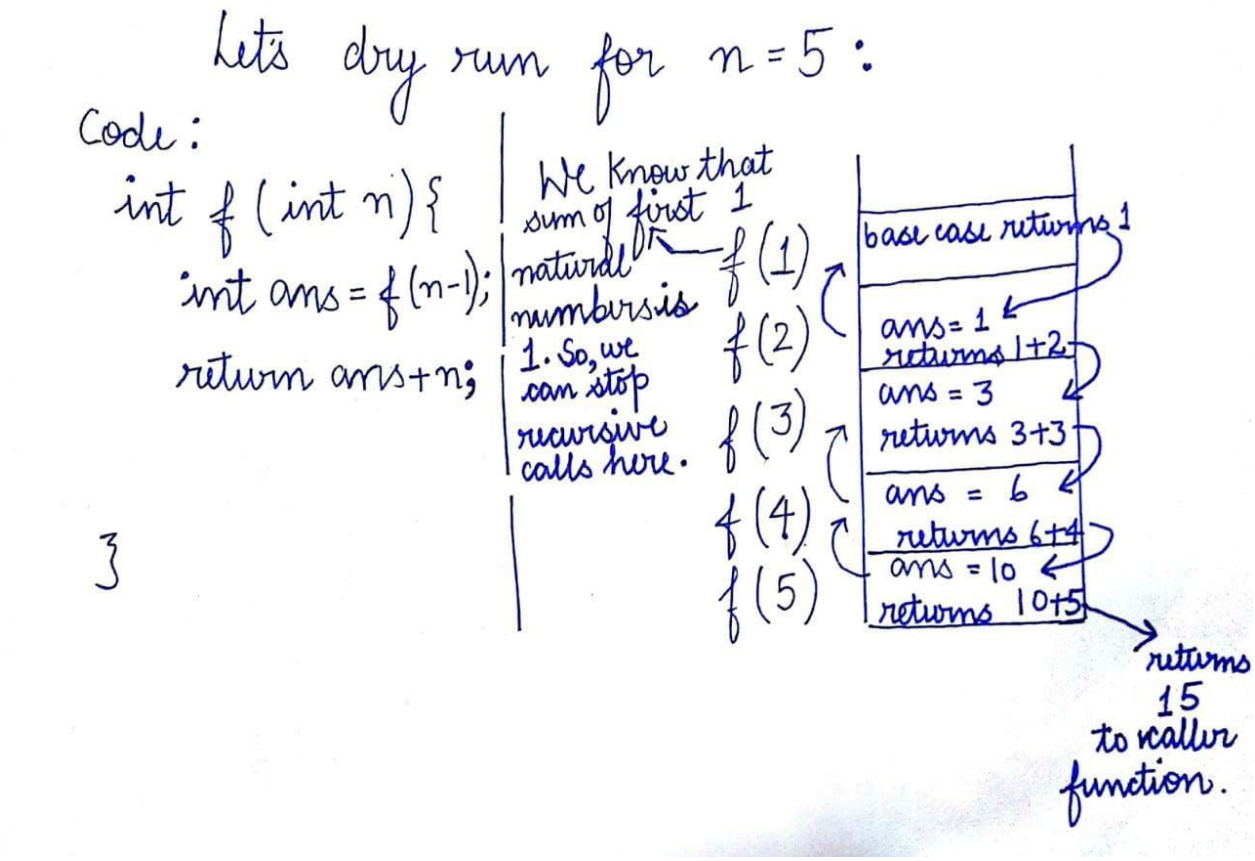
First Step into Recursive World

Let's dig deeper into 3 steps of Recursion. To solve a problem using recursion, follow these 3 steps:

1. Make an expectation (E) and faith (F): Here, expectation is the main problem which I am trying to solve using recursion, whereas faith is the sub-problem, using which I can achieve expectation. Let's define expectation and faith for our running example:
   Expectation: E(n): This gives us sum of first n natural numbers
   Faith: F(n-1): This gives us sum of first n-1 natural numbers.
2. Express expectation in terms of faith and write code: E = F + n
   Code:

```
int f(int n){
      int ans=f(n-1); //Faith step
      return ans + n; //Achieving expectation from faith
}
```

3. The code is still not complete. The missing part is base case. I will dry run to find the case where the recursion needs to stop.



So, our base case is decided. Let's write complete code for the same.
Code:

```
int f(int n){
      if(n==1)
```

```
            return 1;
        int ans = f(n-1);
        return ans + n;

    }
```

Summary till now:

You all know about iterative solutions. I introduced you with another technique called recursion, which is a powerful programming paradigm for solving problems. Recursive solutions are built on solutions of smaller subproblems, these sub-problems are further divided into sub-problems, till they are reduced to trivial sub-problem, whose answer I already know. This trivial sub-problem is called base-case. Generally, each recursive solution is built on following these 3 steps:

1. Defining expectation and faith.
2. Expressing expectation in terms of faith and writing “first-draft” of code.
3. Dry running “first-draft” to find the base case.

Flow of execution

Let us solve some more questions to understand recursion, better. I will solve problems with increasing complexity for the purpose. All these problems will have a return type of ‘void’ and will make only one recursive call.

Problem I: You are given a number N and you have to print numbers in the decreasing order, till 1. So, if N=5, then you need to print this to console output:

```
5
4
3
2
1
```

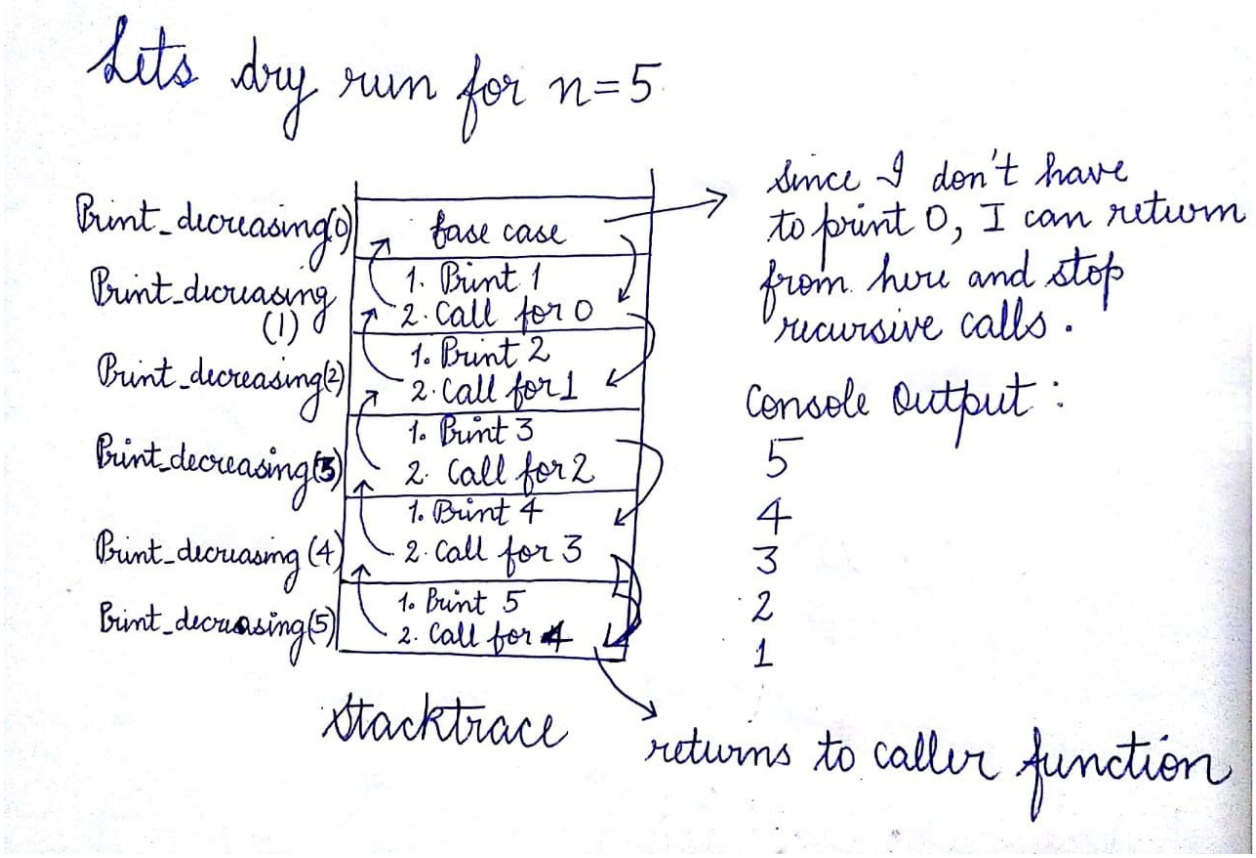
I will solve the problem using the same principle that I had discussed on Day 1 of Recursion.
The first step was to define expectation and faith.
Expectation on an input ‘n’ = E(n) =This will print numbers in the decreasing order, till 1, starting from n.
Faith on an input ‘n-1’ = F(n-1) =This will print numbers in the decreasing order, till 1, starting from n-1.

The second step was to define expectation in terms of faith and write “first draft” of code.
E(n)=Print n + F(n-1)
“First draft” Code:

```
public void Print_decreasing(int n) {
        System.out.println(n);
```

```
        Print_decreasing(n-1);
    }
```

The third and last step was to dry run this on a sample input to figure out base case.



After an exercise of dry run, I have identified base case as:

```
    if(n==0){
        return;
    }
```

Let us write complete code:

```
    public void Print_decreasing(int n){
        if(n==0){
            return;
        }
        System.out.println(n);
        Print_decresing(n-1);
    }
```

Problem II: You are given a number N and you have to print numbers in the increasing order, till N, starting from 1. So, if N=5, then you need to print this to console output:

```
1
2
3
4
5
```

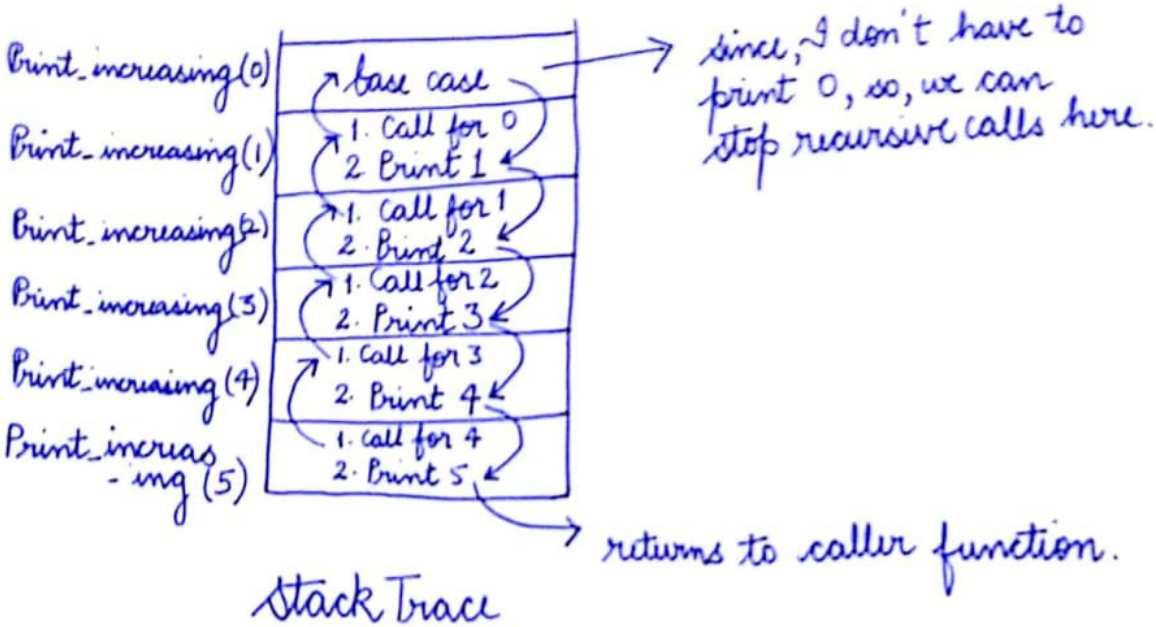
As in the previous problem, the first step was to define expectation and faith.
Expectation on an input ‘n’ = E(n) =This will print numbers in the increasing order, till N, starting from 1.
Faith on an input ‘n-1’ = F(n-1) =This will print numbers in the increasing order, till N-1, starting from 1.

The second step was to define expectation in terms of faith and write “first draft” of code.
E(n)=F(n-1) + Print n
“First draft” Code:

```
public void Print_increasing(int n) {
        Print_increasing(n-1);
        System.out.println(n);
}
```

The third and last step was to dry run this on a sample input to figure out base case.

Lets dry run for n = 5



Stack Trace

Console Output :

1
2
3
4
5

After an exercise of dry run, I have identified base case as:

```
if(n==0){
        return;
}
```

Let us write complete code:

```
public void Print_increasing(int n) {
        if(n==0){
                return;
        }
        Print_increasing(n-1);
        System.out.println(n);
}
```

Problem III: You are given a number N and you have to print numbers in the decreasing-increasing order. So, if N=5, then you need to print this to console output:

```
5
4
3
2
1
1
2
3
4
5
```

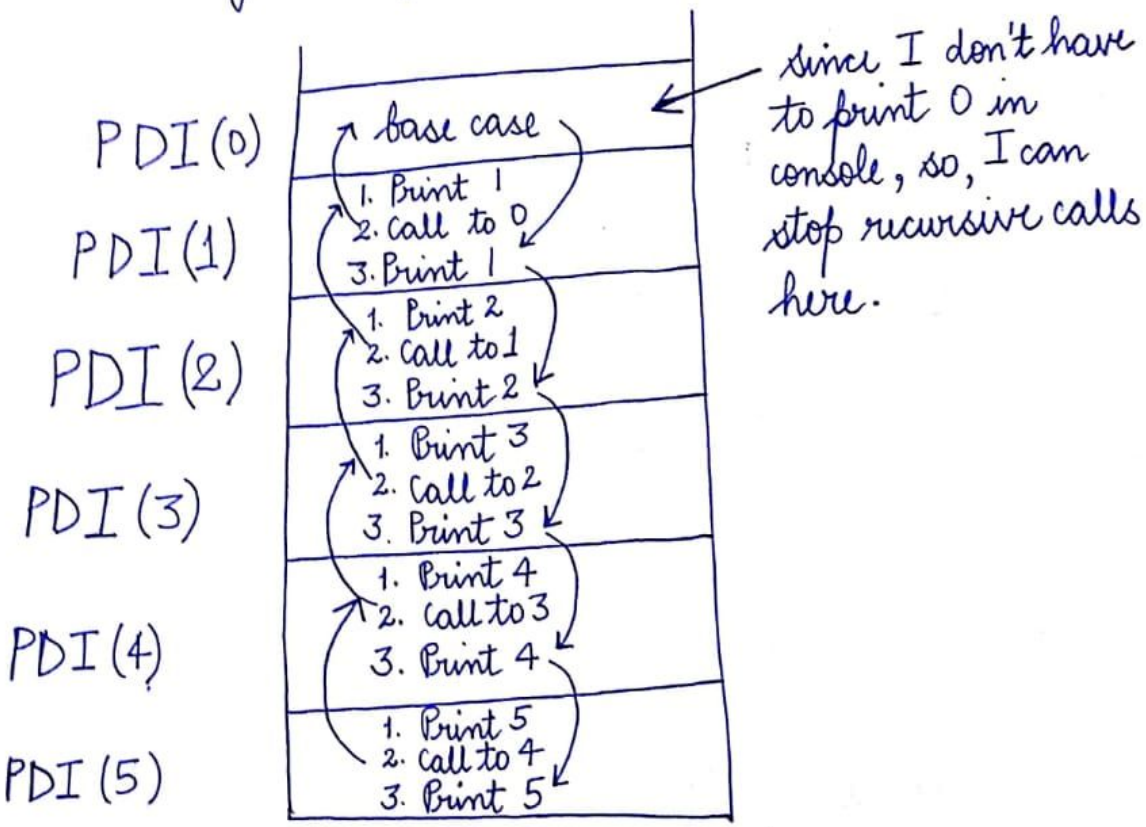
As in the previous problem, the first step was to define expectation and faith.
Expectation on an input ‘n’ = E(n) =This will print numbers in the decreasing-increasing order.
Faith on an input ‘n-1’ = F(n-1) =This will print numbers in the decreasing-increasing order.

The second step was to define expectation in terms of faith and write “first draft” of code.
E(n)=Print n + F(n-1) + Print n
“First draft” Code:

```
public void PDI(int n) {
        System.out.println(n);
        PDI(n-1);
        System.out.println(n);
}
```

The third and last step was to dry run this on a sample input to figure out base case.

Lets dry run for $n = 5$



Stack Trace

Console Output:

5
4
3
2
1
1
2
3
4
5

After an exercise of dry run, I have identified base case as:

```
if(n==0){
	return;
}
```

Let us write complete code:

```
public void PDI(int n) {
	if(n==0){
		return;
	}
	System.out.println(n);
	PDI(n-1);
	System.out.println(n);
}
```

Summary for this segment:
With the help of 3 problems of increasing complexity, I am sure you have been able to comprehend flow of execution of single recursive calls.

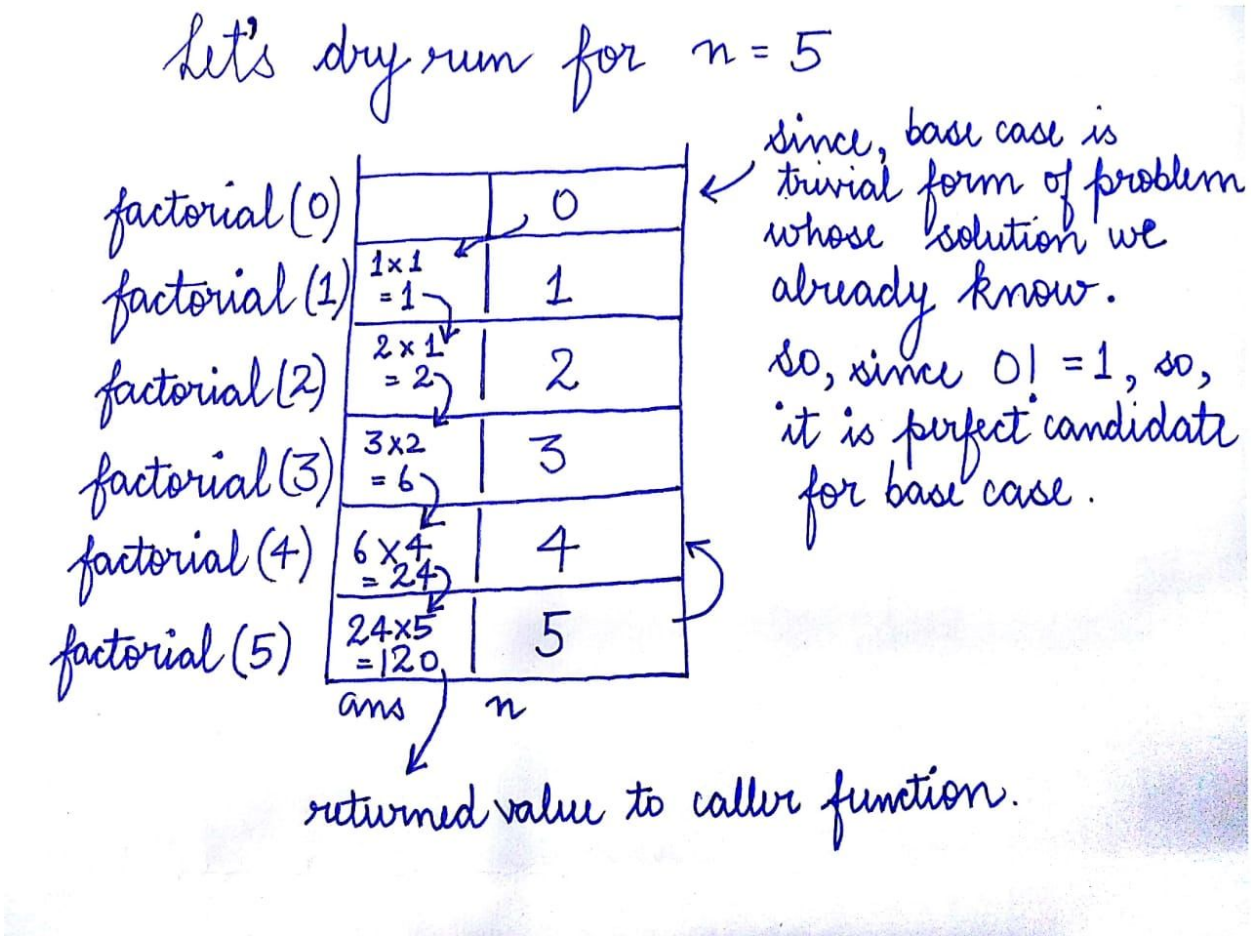
Recursive functions with non-void return types
The problem that I will be demonstrating is: You are given a natural number, n and you have to return factorial of that number, recursively.

I will solve this problem using the same principles that I have discussed so far. So, let's break this problem using steps that I have discussed so far. The first step was to define expectation and faith.
Expectation on an input of n = E = This will return factorial of n.
Faith on an input of n-1 = F = This will return factorial of n-1.

The second step was to express expectation in terms of faith. So, the question I am trying to answer is - Can I achieve factorial of n, if I have calculated factorial of n-1. Ofcourse, I can. Factorial of n = n * factorial of n-1. So, E = n * F.
Let's write "first-draft" of code using this information:

```
public int factorial(int n){
	int ans = n * factorial(n-1);
	return ans;
}
```

The third and last step is to dry run this code to find the point where I have to stop making recursive calls.



So, after an exercise of dry run, I have identified base case as:

```
if(n==0){
    return 1;
}
```

Let us write complete code:

```
public int factorial(int n){
    if(n==0){
        return 1;
    }
    int ans = n * factorial(n-1);
    return ans;
}
```

The take-away from this exercise is that each time I make a recursive call, a function is added in the stack trace and all the local variables of the function are initialized again.

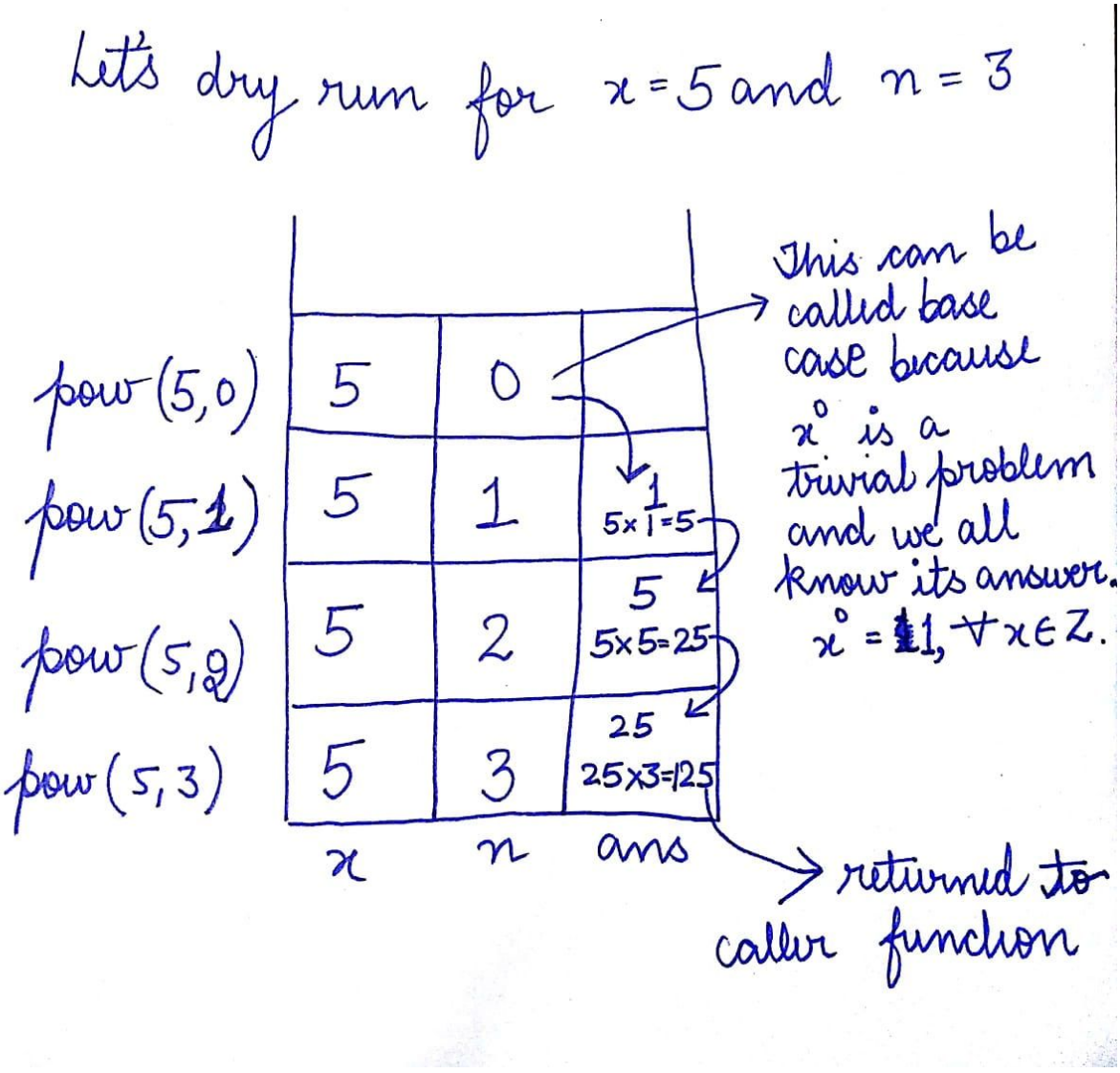
Moreover, I strongly feel that writing is a weak medium of expression, so, I am also giving you link for the video which explains the concept. You can access it here.

Problem: You are given two integers x and n. You have to recursively calculate and return x^n.
So, if x=5 and n=3, then the output must be 125.
I will solve the problem using those 3 steps, that I have discussed so far. The first step was to define expectation and faith.
Expectation from input: x and n= E= This will calculate and return x^n.
Faith from input: x and n-1= F= This will calculate and return x^{n-1}.

The second step was to define expectation in terms of faith and then write "first-draft" of code.
E = x * F
"First-draft" of code:

```
public int pow(int x, int n){
        int ans= x * pow(x, n-1);
        return ans;
}
```

The ultimate step was to dry run this code to find base case.



So, base case that I have figured out is:

```
if(n==0){
        return 1;
}
```

Complete Code:

```
public int pow(int x, int n){
        if(n==0){
                return 1;
        }
        int ans= x * pow(x, n-1);
        return ans;
}
```

This is O(n) solution. I have a O(log n) solution for this problem. The O(log n) solution is called Binary Exponentiation. Before I take you into this chasm called “Binary Exponentiation”, let us comprehend the concepts of complexity analysis of a recursive function. There are many methods, but in this write up, I will be discussing back-substitution method. In this back-substitution method, I will introduce a term called T(n) that denotes running time of an algorithm on an input of size n, and I will write equations that T(n) must satisfy. For example, for the recursive function pow(x, n),

`T(n) = T(n-1) + c` Equation (1)

Now, substituting n-1 in place n in equation (1) gives us:

`T(n-1) = T(n-2) + c` Equation (2)

So, equation (1) can be re-written as:

`T(n) = T(n-2) + 2c` Equation (3)

Similarly, T(n-2) can be expanded and equation (3) can be re-written as:

`T(n) = T(n-3) + 3c` Equation (4)

And in general, equation (2) and (4) can be re-written as:

`T(n-k) = T(n-(k+1)) + c` Equation (5)

`T(n) = T(n-k) + kc` Equation (6)

Now, I know that this recursive function invoking ends at n=0. So, the general equation (5) can be re-written as-

`T(n-k) = 1, if (n-k)=0` Equation (7.1)

`So, n=k.` Equation (7.2)

Hence, from equation (6) and (7.2),

`T(n) = T(0) + nc` , since we know that T(0)=1, so, T(n)= nc, which is O(n).

So, at the end of this write-up, I think you have tested the waters of recursion. Stay tuned for next write-up, which will be on Binary Exponentiation and it’s application.

Alternatively, I can solve this problem using the following recurrence relation:

x^n = 1, if n==0
$x^{(n/2)2}$, if n>0 and n is even
$x^{((n-1)/2)2} * x$, if n>0 and n is odd

So, as this code depends on n being even or odd, I have to use conditional statements in my recursive function. The function would look something like this:

```
public int pow(int x, int n){
    if(n==1){
        return 1;
    }

    int res = pow(x, n/2);
    if(n%2 == 0){
        return res*res;
    }
```

```
        else{
            return res*res*a;
        }
    }
```

Complexity Analysis

The recurrence relation can also be written as:

`T(n) = T(n/2) + c`

Complexity treatment for this recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + c \quad -①$$

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + c \quad -②$$

$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + c \quad -③$$

Putting ② in ①

$$T(n) = T\left(\frac{n}{4}\right) + 2c \quad -④$$

$$T(n) = T\left(\frac{n}{8}\right) + 3c \quad -⑤$$

in general

$$T(n) = T\left(\frac{n}{2^k}\right) + Kc \quad -⑥$$

Now, we know that:

$T(n) = 1$, if $n = 1$ or 0

so, $\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow \boxed{\log n = k}$

Putting result in equation 6, we get,

$$T(n) = T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n \times c$$

$$= T(1) + \log_2 n \times c$$

$T(n) \propto \log(n)$ or $T(n) = O(\log(n))$

Summary for this segment
I think I have discussed enough problems on function with single recursive calls. Going forward, I will discuss problems with multiple recursive calls.

Tower of Hanoi
Problem
You are given 3 rods, namely source, destination and auxiliary, and ‘n’ disks. In the beginning, all the disks are kept in the source rod with the largest disk at the bottom and the smallest disk at the top. Your task is to move all the disks from source rod to destination rod, using auxiliary disk. Your task is to print all the movements. The movements are to be made following these 2 rules:

1. One disk at a time can be moved.
2. Only smaller disks can be placed over larger disks.

I will solve the problem from expectation and faith principles. So, the first step was to define expectation and faith for the problem.
Expectation for an input of n disks = E = This will print all the instructions for moving n disks from source rod to destination rod, using auxiliary rod.
For the following problem, I will define two faiths.
Faith for an input of input of n-1 disks = F1 = This will print all the instructions for moving n-1 disks from source rod to auxiliary rod, using destination rod.
Faith for an input of input of n-1 disks = F2 = This will print all the instructions for moving n-1 disks from auxiliary rod to destination rod, using source rod.

The second step was to define expectation in terms of faith and then write “first-draft” of code.
E = F1 + Print instruction of moving nth disk from source rod to destination rod + F2
“First-draft” of code:

```
public void toh(String src, String dest, String aux, int n ){

        toh(src, aux, dest, n-1);
        System.out.println("Move disk from "+ src +" to "+dest );
        toh(aux, dest, src, n-1);
}
```

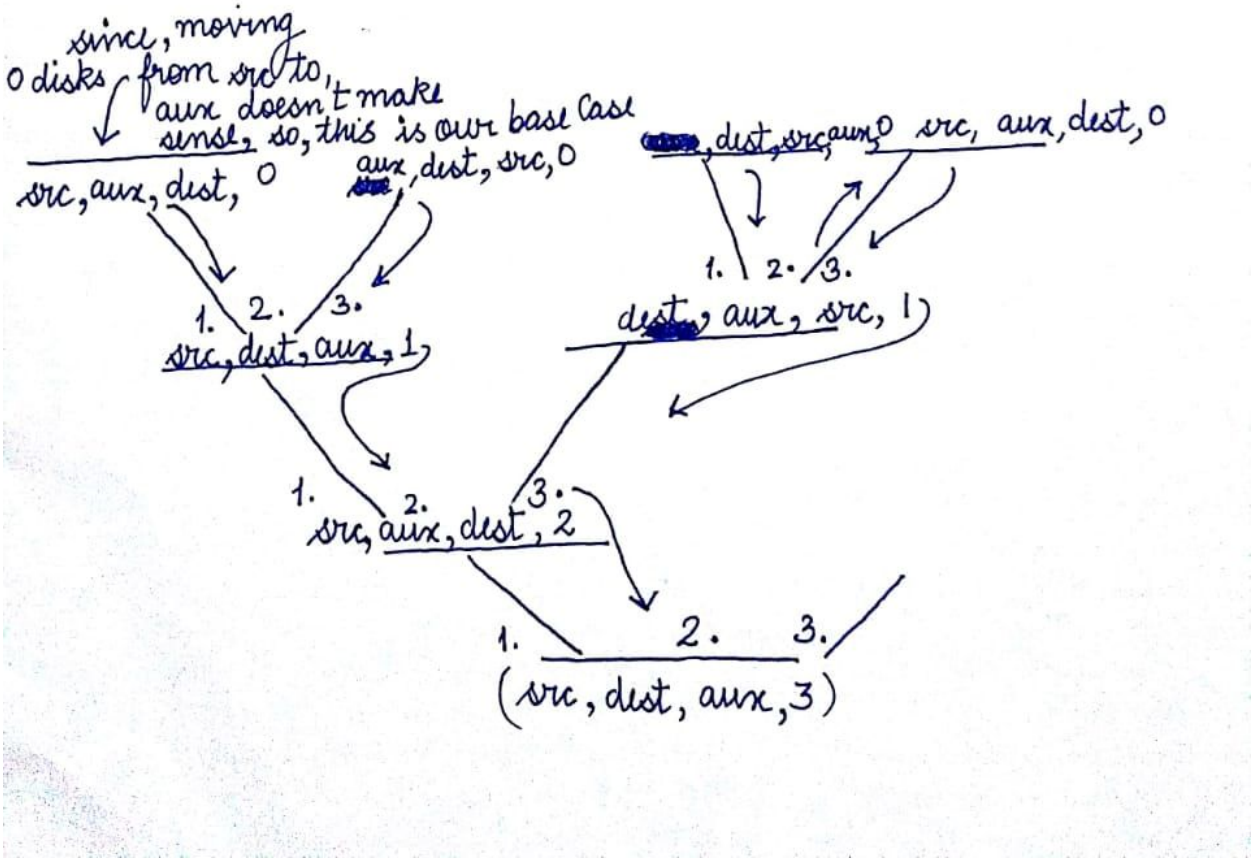
The third step was to dry run the first first-draft of code and identify the base case.

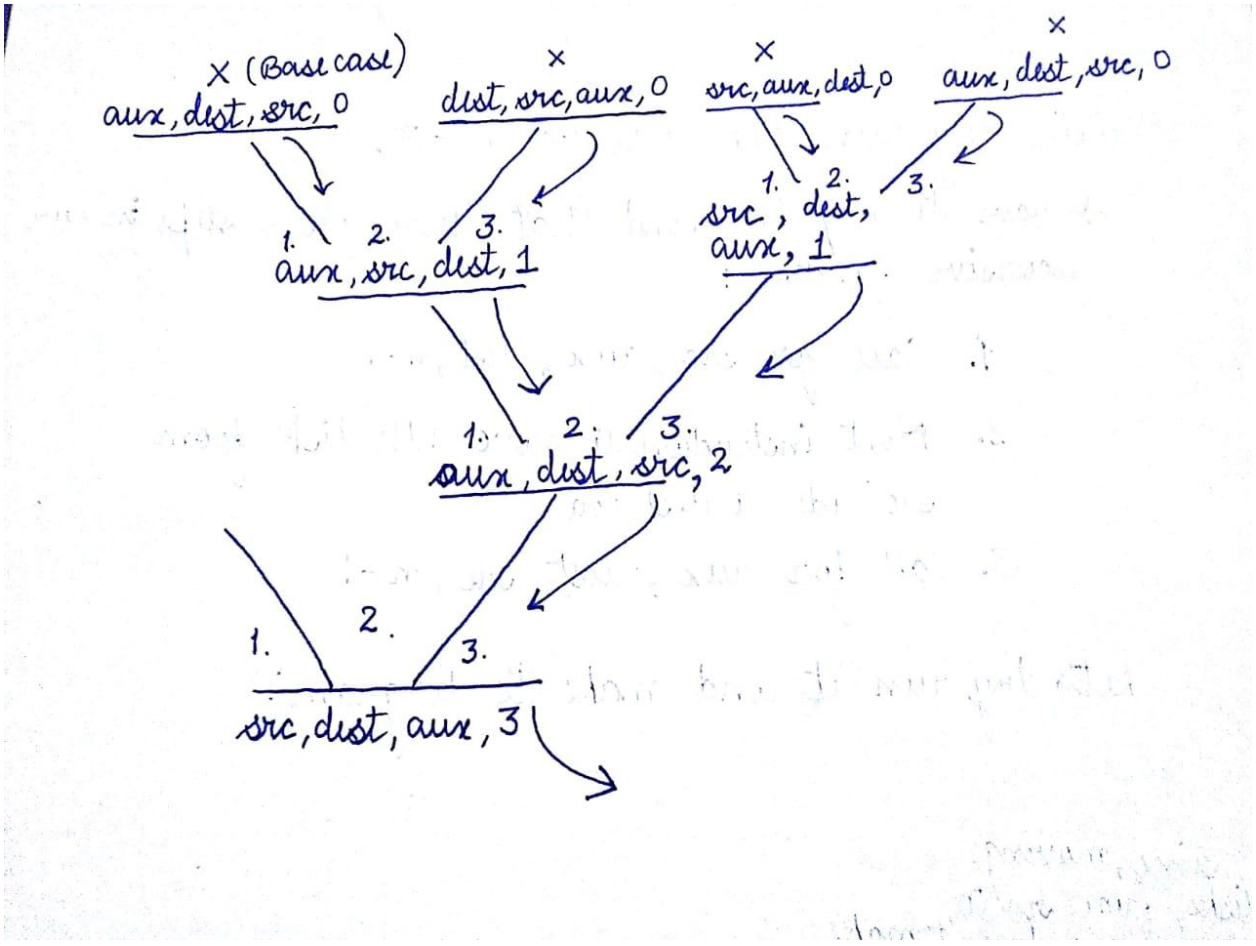
Let's dry run the code for $n = 3$.

We have to keep in mind that there are 3 steps in our recursive function:

1. Call for src, aux, dest, n-1
2. Print instruction to move nth disk from src rod to dest rod
3. Call for aux, dest, src, n-1

Let's dry run it and make it's diagram:





So, the identified base case is:

```
if(n==0){
        return;
}
```

So, the complete code is:

```
public void toh(String src, String dest, String aux, int n ){
        if(n==0){
                return;
        }
        toh(src, aux, dest, n-1);
        System.out.println("Move disk from "+ src +" to "+dest );
        toh(aux, dest, src, n-1);
}
```

Conclusion

I think, now, you are able to comprehend the execution of recursive calls in a better way. Going forward, I will discuss these recursive concepts on different data types and structures such as arrays, arraylists, strings and many more.