Chapter 7 | Functions and Recursion

Java AP

What is Recursion?

Recursion is simply the process of a function calling itself. This may seem a bit weird at first, if a function just calls itself wouldn't that lead to an infinite loop? There is a quip along the lines of “To understand recursion see recursion”. There is no need to be afraid of recursion, if used correctly, it is a powerful tool in the programmer’s toolbox. The more you practice the concept the clearer and easier recursion will become.

To explain recursion we are going to start with an example of an algorithm that can be done in a recursive manner, computing a number's factorial. Factorial is defined for all positive integers as the product of all positive integers less than or equal to n:

$$5! = 5 * 4 * 3 * 2 * 1$$
$$5! = 120$$

Note: 0! Is defined to be 1

Iterative Factorial Algorithm

First, let’s look at the algorithm implemented in an iterative fashion. In software when we refer to iterative we simply remain repeating the same chunk of code multiple times. What does this remind you of? Loops! Here is the factorial algorithm in an iterative (loop) based manner:

```
public class Main
{
	public static int factorialIterative(int number)
	{
		int output = 1;
		while(number > 0)
		{
			output *= number;
			-- number;
		}

		return output;
	}

	public static void main(String[] args)
	{
		System.out.println("5! = " +  factorialIterative(5));
	}
}
```

If we run this program, we see that we do indeed get the expected output ("5! = 120"). So, let us do a sample run to see how the algorithm is working.

- ***number*** is the variable that holds the number we want to get the factorial of. In our case ***number*** is equal to 5.
- First, we create an integer variable that will store the final result and initialize it to 1

int output = 1;

- Now, we want to multiply the ***output*** variable with all the numbers between the value of ***number*** and 0 (0 not included). A simple while loop will do the trick.
- Since we will not need the original value stored in *number* in the remainder of our function, we can change its value as we please.
- Our loop will be:
  - multiplying ***output*** with the current content of number
  - decrement ***number*** by 1
  - repeat until the content of ***number*** is equal to zero

Iteration number	Value in *output* at the end of the iteration	Value in *number* at the end of the iteration
1	5	4
2	20	3
3	60	2
4	120	1
5	120	0

Note: As you can see, the last iteration is not very useful since we already had the factorial value. If you want to optimize your code, you can change the while loop's condition to (*number > 1*)

- After the loop is done, *output* will contain the factorial value which we will return.

Recursive Factorial Algorithm

Before we look at the recursive version in code let's look at a different manner to define n!

For n = 5

$$5! = 5 * 4 * 3 * 2 * 1$$
$$5! = 5 * (4 * 3 * 2 * 1)$$
$$5! = 5 * 4!$$

In general form, this gives us: $\boldsymbol{n}! = \boldsymbol{n} * (\boldsymbol{n} - \mathbf{1})!$ We use this fact to make our recursive function.

```
public class Main
{
	public static int factorialRecursive(int number)
	{
		if(number == 1) /*because 1! is equal to  1*/
		{
			return 1;
		}
		else
		{
			/* number! = number * (number - 1)!
			return number * factorialRecursive(number - 1);
		}
	}

	public static void main(String[] args)
	{
		System.out.println("5! = " +  factorialRecursive(5));
	}
}
```

If we run this program, we see that we do indeed get the expected output ("5! = 120") but now we are using a recursive factorial function. So, let us do a sample run to see how the algorithm is working.

- when we first call the ***factorialRecursive*** function, ***number*** will contain the value that we want to get the factorial of. In our case ***number*** is equal to 5.

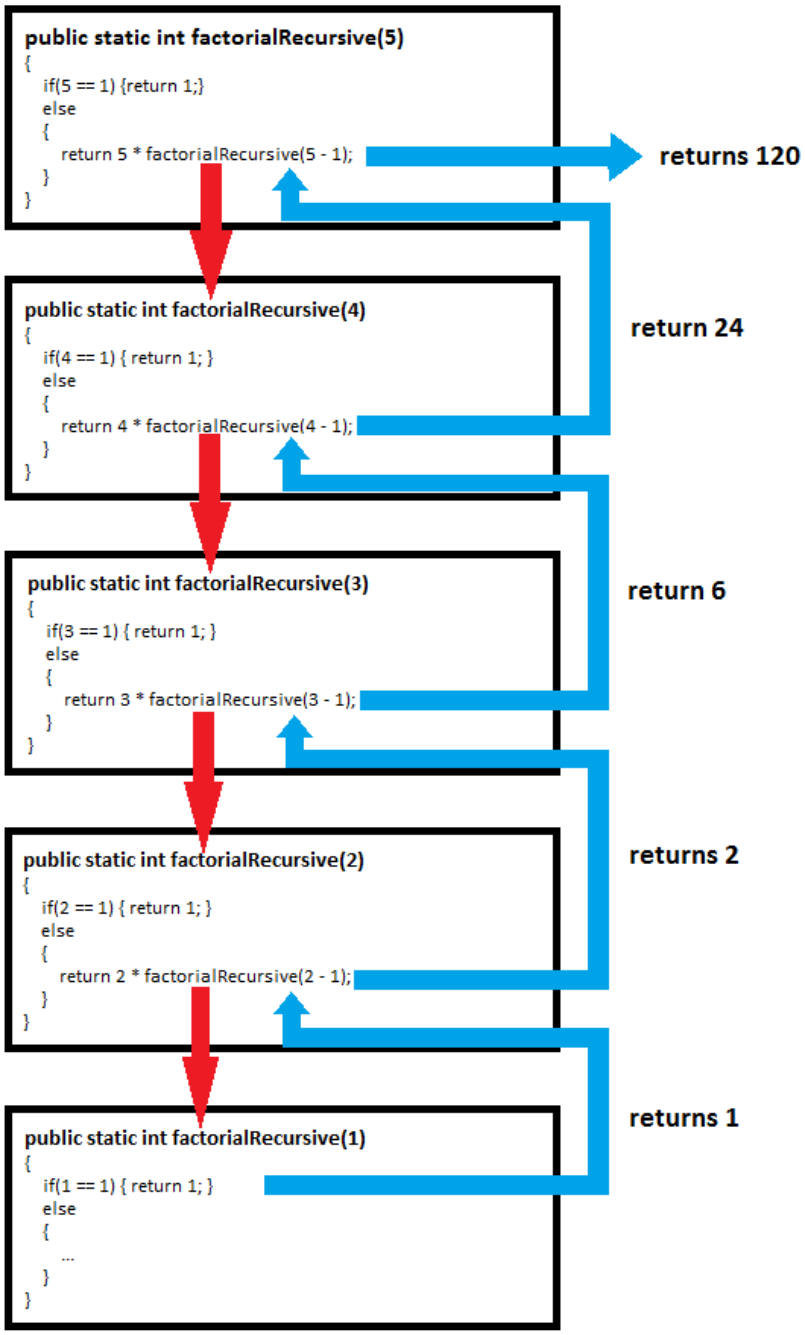
- since ***number*** isn't equal to 1, the code goes to the else clause which is returning the value of: ***number * factorialRecursive(number - 1)***
  - Most important thing to know is that the value will not be returned until the new ***factorialRecursive*** call is done

- The new ***factorialRecursive*** call has the value 4 in its ***number*** variable which will also lead to a new call for a ***factorialRecursive*** function with 3 in its ***number*** variable, etc...
- The recursion will keep going until we reach a ***factorialRecursive*** function with a 1 in its *number* variable. That will make the ***number == 1*** condition be true and by that the function returns a 1 and

stops the recursion (in other words, doesn't call the ***factorialRecursive*** function again with a new value).

Note: "*if(number == 1)*" is known as a termination condition

- After we reach the termination condition, the code does something called bubbling up the return value of each function called from the last function till the first.
  Bubbling is better explained in a diagram:



More Recursive Algorithms

A second algorithm we can look at is power (exponentiation), for example:

b^n which means raising b to the power n.

Another way to look at raising b to the power n is multiplying b by itself n times, for example:

b^5 is the same as b*b*b*b*b
3^5 = 3 * 3 * 3 * 3 * 3 = 243

Note: By definition any value raised to the 0^{th} is defined to be 1, example: 5^0 is defined to be 1.

As far as coding goes, this algorithm can be achieved by using a loop that repeats the multiplication of b by itself n number of times. Below is the code:

Example:

```
public class Main
{
	public static int powIterative(int b, int n)
	{
		int output = 1;
		for(int i = 0; i < n; ++i)
		{
			output *= b;
		}
		return output;
	}

	public static void main(String[] args)
	{
		System.out.println("Iterative 3 to power 5 = " +  powIterative(3,5));
	}
}
```

Output:

```
Iterative 3 to power 5 = 243
```

Indeed we got the correct value of 243. Just like with the factorial we can rewrite b^n in a recursive form. Let's look at 3^4. If we expand the expression we get 3 * 3 * 3 *3 which can be written as 3 * (3 * 3 * 3). What do you notice about this? What is (3 * 3 * 3) equivalent to? (3 * 3 * 3) = 3^3, therefore we can rewrite 3^4 as 3 * (3^3). Extending this to the general form of the equation we get b^n = b * (b^{n-1}). The new form helps us write the following recursive solution:

```
public static int powRecursive(int b, int n)
{
        if(n == 1)
        {
                /* Any number raised to the 1st power is itself*/
                return b;
        }
        else
        {
                /* b^n = b * b^(n-1) */
                return b * powRecursive(b,n-1);
        }
}
```

Note: We could use the relation $n^0 = 1$ here, however that would require one more unnecessary function call.

Efficiency of Recursive Functions

How does the performance of a recursive function compare with that of an iterative function? To understand this, we need to look at what each functions is doing. First let's look at the iterative solution to power:

Function	public static int powIterative(int b, int n)
	{
	int output = 1;
	for(int i = 0; i < n; ++i)
	{
	output *= b;
	}
	return output;
	}
		public static int powRecursive(int b, int n)
		{
		if(n == 1)
		{
		return b;
		}
		else
		{
		return b * powRecursive(b,n-1);
		}
		}

Memory	When called, the function will create 4 integers (b, n, output and i) which totals to 16 bytes of memory. ***PS: This is constant for any value of b and n.***	At first look, the powRecursive function seems better since it only creates 2 integers (b and n). However, since it is recursive, the 2 integers will be created for every powRecursive call. For example, powRecursive(3,5) will end up calling the powRecursive function 4 extra times which leads to 10 integers created in total. Another thing to keep in mind, the recursive

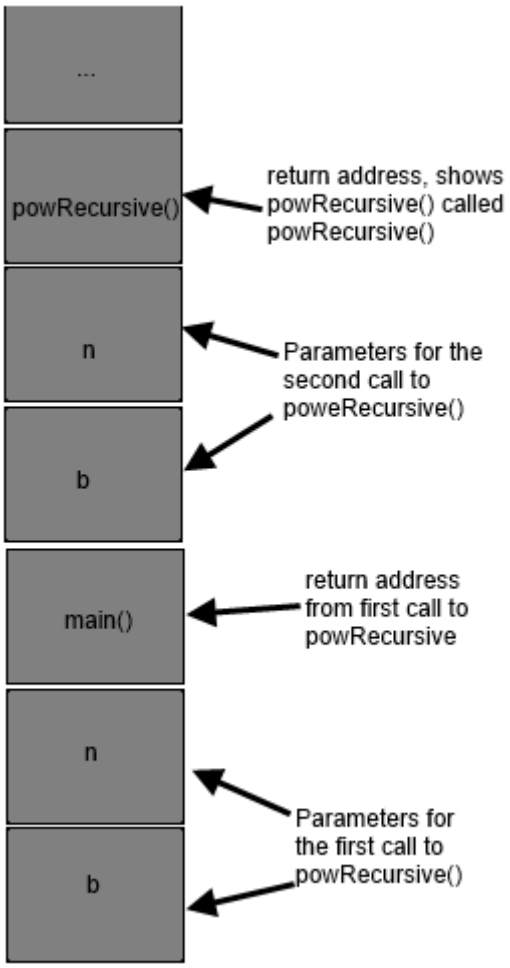
		function's memory intake changes depending on the parameters sent to it, for example 5^8 consumes more memory than 5^2 since more calls will be made during the recursion.
Benchmark (run time)	**1000000 calls:** b = 3, n = 5 in 5 milliseconds b = 3, n = 15 in 5 milliseconds **1000000000 calls:** b = 3, n = 5 in 5 milliseconds b = 3, n = 55 in 5 milliseconds b = 3, n = 155 in 7 milliseconds	**1000000 calls:** b = 3, n = 5 in 10 milliseconds b = 3, n = 15 in 22 milliseconds **1000000000 calls:** b = 3, n = 5 in 5448 milliseconds b = 3, n = 55 in 68099 milliseconds

In conclusion, the iterative function is faster and consumes less memory than the recursive one.
If iterative function are so much better than recursive one and we can write every recursive function in an iterative way, why would we ever use recursion? The only reason to use recursion is the elegance and simplicity of the recursive code.

Note: The examples shown in this chapter are easy to write in both iterative and recursive way. Once you start dealing with trees and more complicated concepts the difference between the two will be significant and you will see that a lot of times the recursive code is much cleaner and easier to write.

Call Stack and Stack Overflow

So what is a call stack? Simply put, the call stack is a piece of memory that stores information about the active subroutines of a computer program. In other words, for every function call, the stack stores all created local variables until the function ends.



Note: This diagram is merely illustrative, the stack may be laid out in many different manners.

If you let your recursion run unchecked you can exhaust the memory available on the stack (a.k.a. filling up the stack). When this happens you get a stack overflow error and your application will crash.

PS: Pushing to and popping from the stack introduces a slight performance penalty.