A Thorough Look at Recursion

Val Chapman
Eli Devine
Stefan Kraus
Sammy Sherman

Introduction:

Recursion, in computer science, is the process a function goes through when one of the steps of the function involves invoking the function itself. When a function call or class is defined by referencing itself, it is recognized as being recursive. Formally, any recursive function must have at least one case that does not result in a call to itself, known as the base case, and any other cases must follow rules that will eventually reduce to this base case.

As a Software Engineer, you are introduced to recursion in your first 200 level programming class. In order to be successful, you need a deep understanding of recursion and implementation. This document goes indepth into recursion; explaining what it does and how to implement it.

The two types of recursion that are discussed in this document are: Tail Recursion and Augmented Recursion.

Background :

The Call Stack

The call stack, in computer science, is a stack data structure that holds the data and information for currently running subroutines. Understanding the basics of how this stack works is essential to understanding the Stack Overflow Error, a common issue encountered when using recursive functions improperly.

A stack data structure is conceptually basic: there is a list of elements that you add to on one end only; colloquially called “pushing” to the “top” of the stack, and remove from the same end; called “popping” the “top” of the stack. It can be envisioned metaphorically as a stack of papers on a table. One can easily take papers off the top, or place more on the top, but it would be impossible to add to the middle or bottom without upsetting the stack. Then, the execution of a program could be thought of as reading through these papers from top to bottom.

When a computer program runs, it pushes data blocks onto the call stack for execution. When a function is called, the data for that function must be added to the stack and executed, and then the program can return to the instructions it was executing. In this example, the papers might be thought of as instructions for your program. If you were reading through these papers and one said “now read the paper titled ‘fibonacci’” you would then place that paper on the stack, read it, then take it off and continue reading.

When a recursive call is made and the recursive step is reached during execution, the function data is pushed onto the stack again. In our metaphor, it would be as if that paper titled ‘fibonacci’ instructed you to read itself, which then of course instructed you to read itself again. This cycle would continue and you would be adding papers to the stack forever! This is what happens to the call stack when a recursive call is not properly reduced to its base case, or if the base case doesn’t exist. Since the call stack in computers is finite, it eventually has no space to store the next function call and the program exits with an error (in modern programming languages at least).

Discussion:

The following article explains the basic types of recursion and provides real life examples.

General discussion of types, implementation, process

Recursive Types

Tail Recursion

A recursive function whose ‘tail call’ is the function itself. A Tail Call means what is called in the ‘return’ statement and has no pending operations that will be performed on that return statement of the call stack. This means a return statement like the one in the example shown below.

Ex:

```
Some function foo(int n){
      return foo(n-1);   ← Here is the tail call, notice it calls
      itself.
}
```

Each recursive call is pushed to the stack. The stack space is reserved until the return statement executes in the function. This means that once the function is next to be popped off the stack, the return statement executes, and this function is no longer reserved on the stack. This recursive process only continues until a *base case* is reached. If no base case is implemented, the function will endlessly call itself until it causes a Stack Overflow. A base case is when a condition statement is called inside the recursive function. Given the example above, it does not contain a base case. Here is what a base case could look like:

```
Some function foo(int n){
      if(n == 1){       ← Here begins the base case.
            return 1;   ← It says, once the value of n equals 1, return 1.
      }                 ← Now we begin popping off the stack.
      return foo(n-1);
}
```

Having a base case is important because it prevents stack overflow, if implemented correctly, and it allows the process of popping from the stack. Let's look at a real example of tail recursion.

```
public static void main(String[] args){
      foo(3);
}

public static void foo(int n){
      if(n == 1){
            System.out.println(1);
      }
      else{
            foo(n-1);
            System.out.println(n);
      }
}
```

The return values would be:

1

2

3

This type of recursion is useful for implementing sorting algorithms such as merge sort, which is covered in the application section. The implementation section will cover more on how recursion can be used to make methods and functions cleaner, more efficient, and more readable.

Augmented Recursion

A recursive function is considered to be "augmented" recursive if the return value includes another call to the function with a pending operation. In augmented recursion, a value cannot be returned until all the function calls have been completed and the base case has been met. Each call to the function is pushed onto the stack. In the factorial example below, if the base case is not met, the function returns `n*factorial(n-1)`. Once the base case of *n*==*0* is met, the function returns a value of 1 instead of another function call. Because there are no more calls to the function, the deferred operations are performed starting at the top of the stack.

Ex:

```
public static int factorial(int n){
      if(n == 0){       ← The base case is 0.
            return 1;   ← 0 factorial is 1.
      }
      else{             ← The return relies on another call to the function
            return n * factorial(n-1);
      }
}
```

Calling the factorial method with an argument of 5 would execute as:

Call Order	Function
1	`factorial(5) ⇒ 5 * factorial(4)`
2	`factorial(4) ⇒ 4 * factorial(3)`
3	`factorial(3) ⇒ 3 * factorial(2)`
4	`factorial(2) ⇒ 2 * factorial(1)`

5	`factorial(1) ⇒ 1 * factorial(0)`
6	`factorial(0) ⇒ 1`

0 is the base case, which returns one. There are no more calls to factorial, now the multiplication can be performed starting from the top of the stack.

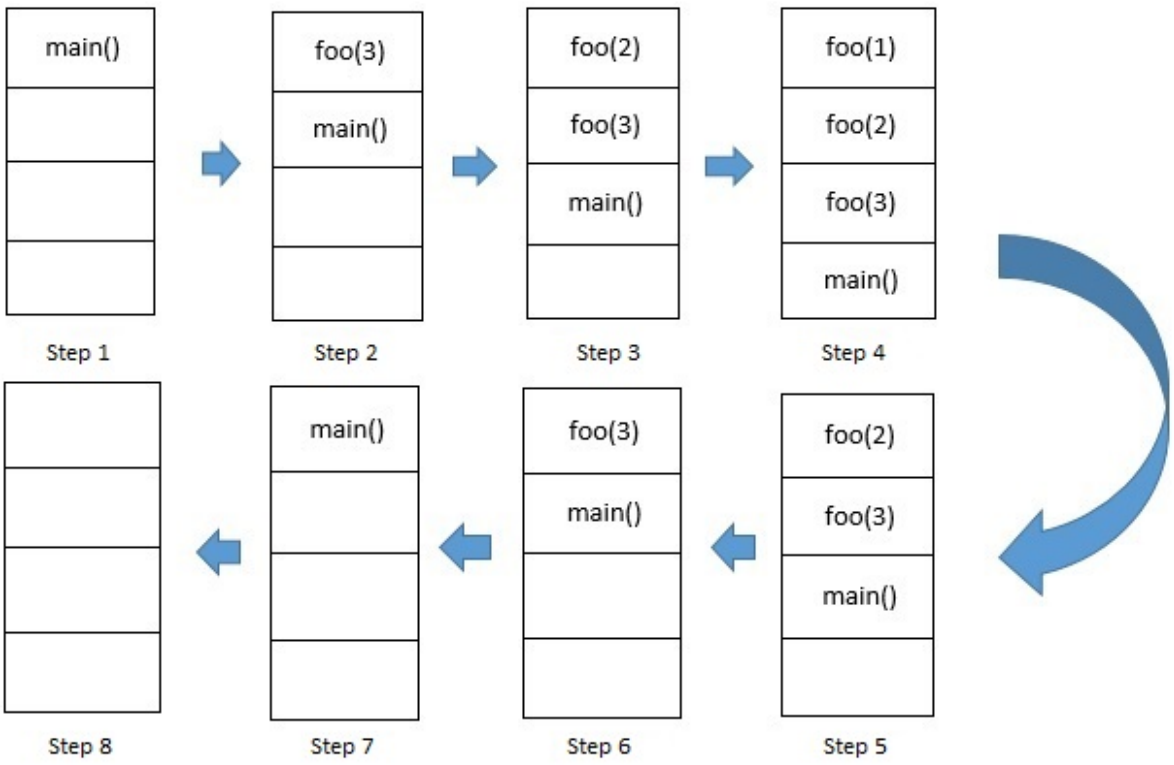
5	`factorial(1) ⇒ 1 * 1 ⇒ 1`
4	`factorial(2) ⇒ 2 * 1 ⇒ 2`
3	`factorial(3) ⇒ 3 * 2 ⇒ 6`
2	`factorial(4) ⇒ 4 * 6 ⇒ 24`
1	`factorial(5) ⇒ 5 * 24 ⇒ 120`
Return	**120**

As shown in the above example, the function cannot return a value until the base case has been met and the operations have been performed. When we call `factorial(5)`, the function evaluates the conditional statement, *n==0,* to determine if the base case has been satisfied. The value of n is 5, 5 does not equal 0, so the conditional statement is false and the else portion of the conditional is executed. In the else block, the function returns the value of n multiplied by a call to the function with n-1 as the argument. So, `factorial(5)` returns 5 * `factorial(4)`. Now the value of `factorial(4)` must be calculated before the value of `factorial(5)` can be determined. The entire procedure is repeated with 4 as the value of n; `factorial(4)` returns 4 * `factorial(3)`. Another function call means the process must be repeated again. This cycle will continue until the base case of 0 is reached when `factorial(0)` is called. When the base case is reached, the function calls can start returning values starting at the top of the stack. `Factorial(0)` returns 1, so `factorial(1)` returns 1*1. As a result, `factorial(2)` returns 2 * 1 (the 1 comes from `factorial(1)`). This continues until we get our result for `factorial(5)`, which is 5 * `factorial(4)`. The value of 120 is returned and the function is finished.

Process

Every time a function is called, it is pushed onto the call stack. Once it runs through and ends, it is popped from the stack and returns any data (if the function has any data to return). When an element is pushed to the stack, it means that each element already existing in the stack is moved one spot lower on the stack and the new element is pushed at the top of the stack. When an element is popped from the stack, the element at the top of the stack call is popped (removed) and all other existing elements are then moved up one spot in the stack.

Going over the example code that includes the `main() method` in the tail recursion, we can see how the process is done, as shown in Figure 1 below.



As seen in Step 1, once the program begins to execute we start in the main method, this function is pushed to the stack. Moving into the function we see the function call `foo(3)`, this is now pushed to the stack as shown in Step 2. The same thing is done for Step 3 and Step 4 for each n-1 call until the base case condition is met. The base case is met at `foo(1)` and the program prints 1, the print statement (which can be thought as the return statement) after being

executed then pops the function call off the stack, this can be seen in Step 5. Now that `foo(1)` is off the stack, we go back to `foo(2)` and continue where it left off (i.e., after the `foo(n-1)` call). This leads to the print call which prints 2, then pops `foo(2)` off the stack. Finally we get to the Step 6 and Step 7 where it prints 3, and pops `foo(3)` off the stack. Looking back into the main method after `foo(3)` pops from the stack we continue from where we left off at and we hit the end of the main method at "`}`". This pops `main()` off the stack and leaves the call stack empty, shown in Step 8.

Implementation

Although recursion can not be defined in a step-by-step guaranteed process, to work there are a few steps you can take to see if recursion can work for your problem. This is a walk through of these steps to calculate the value of x to the nth power(x^n) or x to the n exponent. We will call this function `exponential(int x, int n)` in both cases. The first example is the solution of the problem using iteration.

```
public static int exponential(int x, int n){
      int total = 1;
      if(n == 0){
            return 1;
      }
      for(int i = 0; i < n; i++){
            total = total * x;
      }

      return total;
}
```

Now that you know how to solve the problem iteratively, lets see how to solve this problem recursively.

First, you need to break your problem into the simplest parts possible.

x^(5) can be broken down into five simple parts: [([([1 * x] * x) * x] * x) * x]. Or x^(n) can be broken down into 1 * x(1) * x(2) * x(3)….. * x(n). This is as simple as the parts can be broken down to.

Then, you need to figure out the base cases that will start the return process.

The base case for this problem can be found by doing a simple calculation of 5^2. You will start by multiplying 5 * 1. But, if it's x^0 we actually need to have the answer be 1 because anything raised to the 0 power is 1. Our two base cases will be,

1. If n == 0 return 1;
2. Once n == 1 return x or x * 1;

Finally, you need to combine your base cases with your simplified steps.

1. Write out your base cases that are in pseudo code above.

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

2. Write out your simplest step in a recursive format, calling the function itself with different parameters.

```
if(n > 1){
      return x * exponential(x, n - 1);
}
```

3. Finally, combine these two steps and add the function definition.

```
private static int exponential(int x, int n){
      if(n == 0){
            return 1;
```

```
        }
        if(n == 1){
                return x * 1;
        }
        if(n > 1){
                return x * exponential(x, n - 1);
        }

}
```

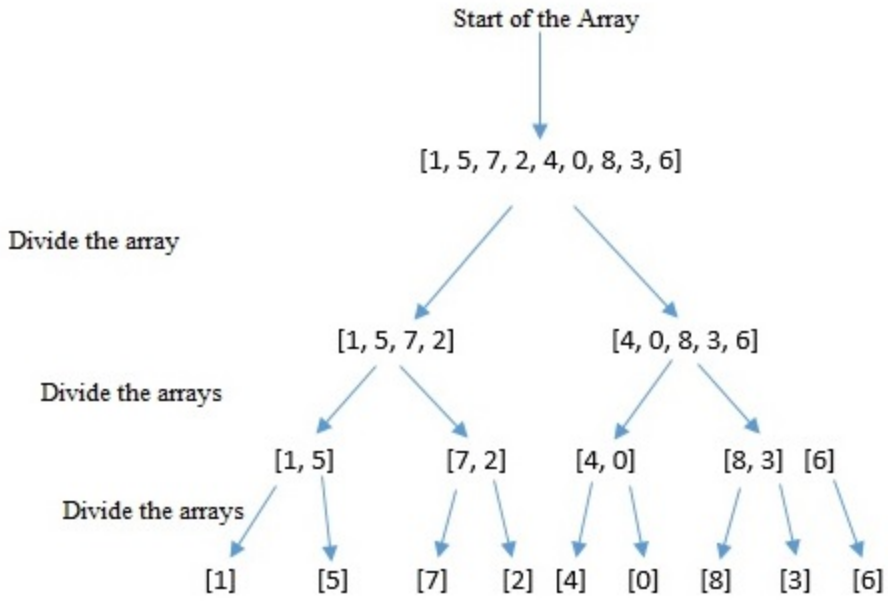
Application : Merge Sort

One of the most common applications of recursion is the Merge Sort Algorithm. This is a general method of sorting an array of comparable elements, for example placing an array of integers into numerical order. This algorithm is very often used because it has great performance in both time to execute and space that is required for execution.

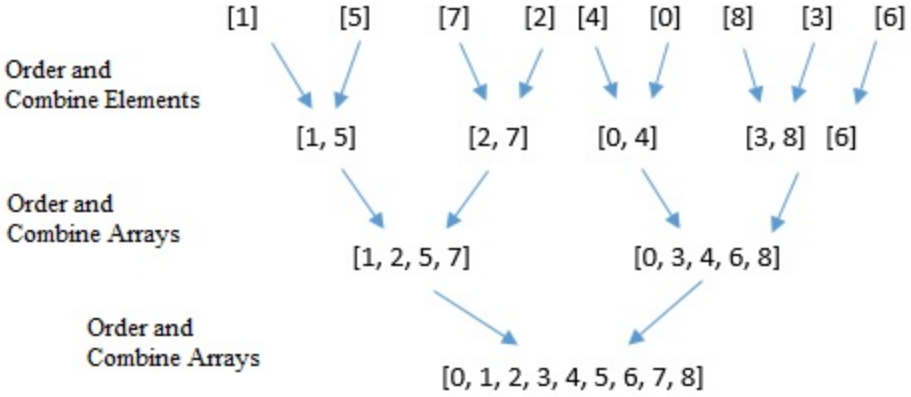
Conceptually, the algorithm has two parts:
1. Divide the given array into two equally-sized or off-by-one sized sub-arrays and sort them using merge sort.
2. Combine the two sorted arrays into one sorted array.

With the base case of the algorithm being the given array is of size one as naturally an array with only one element is sorted.

Step through the algorithm with the following array as the input:
[1, 5, 7, 2, 4, 0, 8, 3, 6]
Divide: => [1, 5, 7, 2] , [4, 0, 8, 3, 6]
Divide: => [1, 5], [7, 2], [4, 0] , [8, 3], [6]
Divide: => [1] [5] [7] [2] [4] [0] [8] [3] [6]



Here, the base case is reached and the last-called recursive step can continue execution.
Combine: => [1, 5] [2, 7] [0, 4] [3, 8] [6]
Combine: => [1, 2, 5, 7] [0, 3, 4, 6, 8]
Combine => [0, 1, 2, 3, 4, 5, 6, 7, 8]



Conclusion:

Recursion is a process of a function calling calling itself to complete tasks and reach a certain goal. Tail recursion does this process by calling itself in the return function. Augmented recursion makes its recursive call in the main portion of the function. Each call to a recursive function performs a push to the stack to store its variables. Once the final call is made on each return the desired output is returned to the previous function until the last function on the stack is returned.