Understanding Recursion

Brian L. Stuart

February 23, 2015

It has been suggested that the single most original contribution that the field of Computer Science has made to the tapestry of human intellect is recursion. Although some aspects of the concept of recursion can be found in the mathematics of the era leading up to the computer, it is in the mechanization of recursion and the application of recursion to computational problems where we see something truly different. When seen from this perspective, one can hardly say that they have a significant understanding of Computer Science if they do not understand recursion. In the paragraphs that follow, I will make an attempt to help you, the reader, develop your understanding of recursion.

As you study this material, you should try to reach the highest level of understanding you can. In particular, there are three stages that you will likely pass through as your understanding deepens:

1. Understanding the terminology and the correspondence between recursive algorithms and the code that implements them
2. Understanding what happens in the computer when recursive code is run
3. Understanding how to construct a recursive solution to a problem

When taking an introductory programming class, the expectation will not generally go beyond the first two stages. That is you might be given a recursive algorithm and asked to turn it into code or given a recursive function and asked to trace what happens when it runs. In later courses, you will be expected to reach the third stage of understanding.

The truth is that to really understand recursion, there are three aspects of it with which we must come to grips. On the one hand, we need to understand it as a mathematical construct. Like much of mathematics we want to understand recursive constructions as static statements about relationships and definitions. On the other hand, we need to understand how recursion is implemented in a computer. What actually happens when we run a program that includes recursive functions? On the other...other hand, we want to understand how to go about constructing recursive solutions to problems. In the discussion that follows, I don't attempt to present these three perspectives in a rigid catagorical way. However, I do hope that you will find aspects of all three of these perspectives that will help you better understand recursion.

I do have to caution you that this is not a simple topic and this discussion is not brief. However, I have attempted to write it in an order such that the information that it's most important for you to understand first is presented first.

1 An Introductory Example

Although not my usually preferred pedagogical mode, I'll start with a very simple introductory example of recursion. You can think of this as the "Hello World" of recursion. It's the factorial function.

1.1 How Factorial Looks Mathematically

The first time you saw the factorial function you probably saw it defined something like:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Generally, this isn't a *bad* definition, but having an ellipsis (the ...) in a definition makes the formal mathematician part of us twitch. It's not as well-defined as we would like. It's certainly not elegant, and it just feels like it there should be a better defined way of saying it.

If we apply a little mathematician's trick of putting a set of brackets in a strategic place, you may notice something interesting:

$$n! = [1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1)] \cdot n$$

Look at the part inside the brackets that we just added. It's exactly how we would write out $(n-1)!$. In other words, we could even more easily say:

$$n! = (n-1)! \cdot n$$

or swapping it around to make it look nicer:

$$n! = n(n-1)!$$

The result is that we've defined factorial in terms of multiplication, subtraction, and factorial! We've defined it in terms of itself. That's the essence of recursion. If you're not seeing how this revised definition corresponds to the original one, now is a good time to pause and think over it. Make sure you understand how all of these ways of expressing factorial are saying the same thing.

However, if you look at this form, there's one little problem. What happens if you try to compute 1!? Suddenly, you're faced with $1 \cdot 0!$ and by this definition, $0! = 0 \cdot -1!$, etc. To make matters worse, the answer (if we ever found a place we could stop) is 0. In fact, if you think about it, an attempt to compute the factorial of any number using this definition will go on forever and give us the answer of 0. Does this mean that this whole recursion thing was a fool's errand? Not at all. We just have to go back and pick up a little part of the factorial

definition that we often forget to mention when describing factorial with the string of multiplications. In particular, part of the definition says that $0! = 1$. One way to look at it is that we were a little sloppy in our first definition, but taking a recursive view forced us to acknowledge that sloppiness and correct it. So in the first case, we should have said:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n & \text{otherwise} \end{cases}$$

or recursively,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

Having a case like this where we don't recurse is called a *base case.* It's important to recognize that every recursive technique has to have a base case. Furthermore, notice how our recursive calls always move us closer to the base case. Again, that's important for all recursive applications. It also points to an important way to think about recursion. Saying that we define some operation in terms of itself is true enough, but it's more precise to say we define it in terms of a simpler instance of itself. In other words, if we assume we have a solution that works with an instance of the problem of size n, then we show how to construct a solution that works for size $n + 1$. We don't have to know how the solution for n works yet; we just have to know how to build the solution for $n + 1$ using it.

1.2 How Factorial Looks in Code

If we reorder the notation in the algorithm a little, it is exactly what we want to say to implement factorial recursively in C or C++ as in Figure 1. Before moving further, make sure you understand how this code is an implementation of the mathematical definition in the last subsection.

```
int fact(int n)
{
    if(n == 0)
        return 1;
    else
        return n * fact(n-1);
}
```

Figure 1: Recursive Factorial Function

However, computers and computer programs aren't static entities that assert relationships. They are dynamic with events taking place in a serial sequence. How does this translation of a mathematical definition into the language of C/C++ behave dynamically as it runs? To help you get a picture of this, I've

```
int fact(int n)
{
    int factn;

    cout << "Entering fact(" << n << ")\n";
    if(n == 0) {
        cout << "n is 0; returning 1 from fact(0)\n";
        return 1;
    }
    else {
        cout << "n is not 0; calling fact(" << n-1 << ")\n";
        factn = n * fact(n-1);
        cout << "back from fact("  << n-1 << "); returning "
            << factn << " from fact(" << n << ")\n";
        return factn;
    }
}
```

Figure 2: Recursive Factorial with Tracing

added some output statements in Figure 2 that show the sequence of events when we call this function.

When we call this function from a main program with a statement like `fact(3);`, we get the output:

```
Entering fact(3)
n is not 0; calling fact(2)
Entering fact(2)
n is not 0; calling fact(1)
Entering fact(1)
n is not 0; calling fact(0)
Entering fact(0)
n is 0; returning 1 from fact(0)
back from fact(0); returning 1 from fact(1)
back from fact(1); returning 2 from fact(2)
back from fact(2); returning 6 from fact(3)
```

Now let's take a look at exactly what happens when this code was run. As you read through the following sequence of steps, follow along in the code and in the output to make sure you understand what's happening.

1. In the main program, we encounter the statement `fact(3);`. As with all function calls, the first thing we do is evaluate the argument, which is especialy easy in this case since it's a constant. We find a memory location to put the value of the argument and transfer control to the beginning of `fact`.

2. In `fact`, we take that memory location where we just stored the argument value 3, and call it `n`.

3. Comparing `n` to 0, we find that it's not and we now have to evaluate the expression `n * fact(n-1)`. To do that, we have to call `fact(n-1)`.

   (a) Just like we did before, we evaluate the argument `n-1`. The memory location we have labeled `n` has 3 in it, so we get the argument value 2. That goes into a memory location that's different from the one we have called `n` and we transfer control to the beginning of `fact`.

   (b) In this instance of `fact` we again give the label `n` to the location where we just put the 2. Note that this is a different `n` than the one we saw in the previous instance of `fact`. That `n` still labels a location containing 3, but we can't see it from within this instance of `fact`.

   (c) Comparing `n` to 0, we find that it's not and we now have to evaluate the expression `n * fact(n-1)`. To do that, we have to call `fact(n-1)`.

      i. Just like we did before, we evaluate the argument `n-1`. The memory location we have labeled `n` has 2 in it, so we get the argument value 1. That goes into a memory location that's different from the one we have called `n` and we transfer control to the beginning of `fact`.

      ii. In this instance of `fact` we again give the label `n` to the location where we just put the 1.

      iii. Comparing `n` to 0, we find that it's not and we now have to evaluate the expression `n * fact(n-1)`. To do that, we have to call `fact(n-1)`.

         A. Just like we did before, we evaluate the argument `n-1`. The memory location we have labeled `n` has 1 in it, so we get the argument value 0. That goes into a memory location that's different from the one we have called `n` and we transfer control to the beginning of `fact`.

         B. In this instance of `fact` we again give the label `n` to the location where we just put the 0.

         C. This time when we compare `n` to 0, we find that it is and we return 1 as the result of this function call.

      iv. Now that we have a result from evaluating `fact(0)`, we can resume with the expression `n * fact(n-1)` in the instance of `fact` where `n` is 1. Carrying out that multiplcation gives us 1, which we return as the result of this function call.

   (d) The same thing happens now in the instance of `fact` where `n` is 2 and we just got back 1 from the call to `fact(1)`. Multiplying them together gives us 2, which we return as the result.

4. Now we're finally back to the orignal instance of `fact` where `n` is 3 and we just got back 2 from `fact(2)`. The last step then is to return the product 6 back to the main program whence we were called.

For most people, the natural reaction, is "Wow, that's a lot. How am I supposed to remember that much detail for every time I use recursion?" The truth is you're not really expected to. What you should be able to do is two-fold. First, you should be able to understand how this sequence of steps arises as a result of running this code. Second, you should feel confident that if the occasion arises where you need to follow a recursive call in that level of detail you'd be able to. The good news is that you won't usually need to, but being able to is a good test of how thoroughly you understand it.

1.3 How Factorial Looks in Memory

There have been several times when I've said something like "we find a memory location to put the value." You'd be right to be asking yourself, "Just where and how is this 'finding' being done?" The answer to that is best described with reference to Figure 3. Every time a function gets called (recursive or not), the program puts a collection of data into the next available area of a section of memory reserved for function calls and similar memory allocations. Those collections of data are represented by the larger boxes in the figure. The values of the arguments are some of the data that are included in those boxes, as illustrated by the smaller boxes labeled `n`. Any local variables in a function are also included there. That's how we have a different value for `n` in every instance of a recursive call. When a function returns, the program frees up the associated box making it available for the next function call.

Because factorial is the "Hello World" of recursion, now is a good time for you to take a break and think about what we have covered so far. Make sure you feel comfortable with the recursive definition of factorial, with the implementation of that definition in code, and with what happens when that code runs. The remaining examples don't get easier, so it's a really good idea to go back over the factorial example until you are confident with your understanding of it.

2 A Second Example

2.1 Parenthesis Precedence

We've talked in class about the fact that the compiler arranges for the part of an expression in parentheses to be evaluated before the rest of the expression. So, for example, in the expression `d * (a + b)`, `a` is added to `b` before we multiply by `d`. However, we haven't said anything about *how* the compiler does that. For this second example, I'm going to show you a very much simplified version of a technique for giving precedence to what's in parentheses. Basically, instead of whole expressions, we're going to use single letters to represent what needs to be done, and we're going to say that something in parentheses can at most

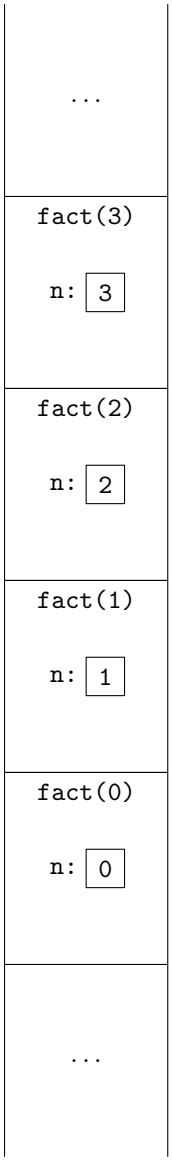

Figure 3: Memory Usage in Recursive Factorial

include one such letter and another expression in parentheses. Rather than bore you with a formal definition of these artificial expressions, I'll just give you a few examples.

Expression	Order of operations
(a (b))	b a
((c (d)) a)	d c a
(a)	a
(a (b (c (d (e)))))	e d c b a

As with the factorial case, I'll present the steps to create a function that solves this problem. It is going to read the parenthesized expression and print out the order of operations.

2.2 The Big Picture

Before I can design the details, I have to establish the big picture. The function is going to get called right after an open parenthesis is read, it will read characters from the input and print out operations in the right order and return when it reads the closing parenthesis that matches the one that made it get called. If we walk though how this will work on the first example in the table:

1. The main program will read the opening parenthesis and call our parenthesis parser.
2. The recursive function will then read the `a` and remember it.
3. The next character it reads is an opening parenthesis. So it will then recursively call itself.
4. The second instance of the parser will then read `b` and remember it.
5. Next, it reads the closing parenthesis. That's the signal to print the `b` it had remembered and to return.
6. The first instance of the parser now gets control back and reads a closing parenthesis. So it prints the `a` it had remembered and returns back to the main program.

Before moving on to the next subsection, stop and see if you can turn the example into an algorithm that will handle all the examples in the table.

2.3 The Algorithm

The example in the previous subsection really does give us a pretty complete picture of the algorithm we need. Here's one way to specify the algorithm so that we can translate it directly into code easily:
Parse parentheses:

- Repeat until we return

1. Read a character c
2. Based on the character, take the following actions:

   (Parse parentheses

) Print the saved character if any and return.

   otherwise Save the character c.

2.4 The Code

One way we can turn this algorithm into code is as in Figure 4. As you can see, it really is a very direct translation of the algorithm into code.

```
void
paren_parse(void)
{
    char c;
    char saveop;
    bool issaved;

    issaved = false;
    while(1) {
        cin >> c;
        switch(c) {
        case '(':
            paren_parse();
            break;
        case ')':
            if(issaved)
                cout << saveop << " ";
            return;
        default:
            saveop = c;
            issaved = true;
        }
    }
}
```

Figure 4: Recursive Parenthesis Parser

Now you may be thinking to yourself, "This doesn't look like the other recursion cases we've seen. There are no parameters, and it doesn't return anything. Besides, I'm not sure there's even a base case. Is he cheating?" I'm not cheating, but I am letting the external input and output streams play the roles of arguments and return values. Doing so allows me to illustrate these thechniques without using language features we haven't covered yet.

3 A More Involved List Example

Here's another example for you to consider. Don't worry, I'm not going into as much detail on how this example operates. Instead we're going to focus more on how we get to the recursive solution. In this example, we are going to work with lists. For instance we might have the list $x = (1\ 2\ 3\ 4\ 5)$. We only have three operations available to us for lists:

1. *first*(l) will result in the first element of the list, l. For the list x, *first*(x) will result in 1.

2. *rest*(l) will result in a list that's everything left over in l after removing the first element. So *rest*$(x) = (2\ 3\ 4\ 5)$.

3. *build*(e, l) results in a new list which is the same as adding e as a new first element in list l. Continuing our example, *build*$(42, x) = (42\ 1\ 2\ 3\ 4\ 5)$.

The last piece of the puzzle for these lists is a way to represent the empty list. You may be thinking, "wouldn't that just be ()?" You'd be right. That's how we're going to denote it. Although this limited set of operations may seem artificial, it turns out to be very natural for some types of list representations and it is also sufficient to build anything else we want to.

3.1 Appending Lists

Given this set of operations available to us, let's try to develop a way to append one list onto the end of another. In particular we want *append*(l_1, l_2) to result in a list where all the elements of l_1 are followed by all the elements of l_2. For example, if $l_1 = (1\ 2\ 3)$ and $l_2 = (a\ b\ c)$, then *append*$(l_1, l_2) = (1\ 2\ 3\ a\ b\ c)$.

Often, our first step in developing a recursive technique is to identify the base case. For appending, we can easily imagine two possible base cases. First, we can take *append*$(\,(), l) = l$ as the base case. Alternatively, we can take *append*$(l, (\,)\,) = l$ as the base case. I'll invoke foreknowledge for you and say that it turns out the first base case will be more useful for us. From what we observed earlier, we know that our recursive case will have to be designed in such a way that we move toward the base case. In other words, the first list needs to get smaller with each recursion.

Our first idea for how to structure the recursive step might be to shift one element from the first list to the second and recursively append the two new lists. Still thinking about $l_1 = (1\ 2\ 3)$ and $l_2 = (a\ b\ c)$, we might imagine recursively appending $(1\ 2)$ to $(3\ a\ b\ c)$. If we had a good way to get access to the last element of a list and to cut it off that would work well. However, it's not the last element that we can do that with; it's the first element. Before going on to the next paragraph, pause to think about this and how you might make use of the *first, rest,* and *build* operations to recursively do this.

Welcome back. You did pause to think about it, right? Well, here's what we've got to work with: the element 1 from *first*(l_1), the list $(2\ 3)$ from *rest*(l_1),

Figure 5: Towers of Hanoi Puzzle

and $l_2 = (a\ b\ c)$. While you were thinking about it, hopefully you noticed that we could append (2 3) to $(a\ b\ c)$ and use *build* to put the 1 at the beginning. So in English, our recursive step would be:

> Use *append* on $rest(l_1)$ and l_2, and use *build* to put $first(l_1)$ onto the beginning.

This can be described more like mathematical notation as:

$$append(l_1, l_2) = \begin{cases} l_2 & \text{if } l_1 = () \\ build(first(l_1), append(rest(l_1), l_2)) & \text{otherwise} \end{cases}$$

4 The Towers of Hanoi

Our last example is the classic puzzle called "The Towers of Hanoi." It consists of three posts and an assortment of different sized rings. The starting point for the puzzle has all of the rings stacked on one post where each ring is on top of a larger one. The objective is to move all of the rings from the starting post to a destination post subject to the following two rules:

1. You can move only one ring at a time.
2. You cannot place a larger ring on top of a smaller ring.

Figure 5 is a picture of such a puzzle with six rings.

The Towers of Hanoi puzzle turns out to be extremely easy to solve recursively. Our strategy is to follow what was suggested earlier about assuming we have a way to solve it for a smaller version of the problem and building a way to solve the larger one from that. Looking at the six-ring puzzle in the picture, suppose I tell you that I can move five rings. Now you want to move six rings from the left post to the right post using the middle post as a temporary post. By taking advantage of what I can do, how would you solve the puzzle? Take a moment to think about it before moving on.

Hopefully you were able to see that you could solve the puzzle as follows:

```
void tower(int n, string start, string dest, string temp)
{
    if(n > 0) {
        tower(n-1, start, temp, dest);
        cout << "move a ring from the " << start << " tower to the "
             << dest << " tower" << endl;
        tower(n-1, temp, dest, start);
    }
}
```

Figure 6: Recursive Towers of Hanoi Code

1. Ask me to move the top five rings from the left (starting) post to the middle (temporary) post.
2. You move the one remaining ring from the left (starting) post to the right (destination) post.
3. Ask me to move the five rings from the middle (temporary) post to the right (destination) post.

If we recognize that there's nothing to be done if there are zero rings and that makes a great base case, then we can generalize the technique into this algorithm: To move n rings from a start tower to a destination tower with a temporary tower:

1. If $n > 0$:
   (a) Recursively move $n - 1$ rings from the start tower to the temporary tower using the destination tower as a temporary for that part of it.
   (b) More one ring from the start tower to the destination tower.
   (c) Recursively move $n - 1$ rings from the temporary tower to the destination tower using the start tower as a temporary for that part of it.

In addition to being another classic example of recursion, the Towers puzzle is interesting in that each instance of the algorithm makes not one, but two different recursive calls of itself. However, as we will see, that doesn't really affect how we turn it into code or how we can understand what it does while it runs (except for taking longer to run). Turning this algorithm into C++ code is pretty simple as you can see in Figure 6. If we call this function from `main` with a line like:

```
tower(3, "left", "right", "middle");
```

we get the following output:

```
move a ring from the left post to the right post
move a ring from the left post to the middle post
move a ring from the right post to the middle post
move a ring from the left post to the right post
move a ring from the middle post to the left post
move a ring from the middle post to the right post
move a ring from the left post to the right post
```

You can find a wide variety of animations showing solutions to the puzzle. One animation on YouTube is:

`https://www.youtube.com/watch?v=BMkOBNZHcIs`

As a last step, let's return to thinking about the memory usage again. Just as with the factorial case, every call results in the allocation of one of those blocks of memory and returning frees it back up. In the case of our `tower` function, though, something interesting happens. Because we have two recursive calls, the second one just ends up reusing the same block the first one did. At any point in time these blocks are lined up like in Figure 3. However, we can also try to draw out all of the blocks that ever exist while the program is running using a sort of tree-like structure. Figure 7 shows how that looks. Each box in the figure is one of those memory allocations discussed earlier. Inside each box, I've listed the four arguments to the function call, but for space, I've abbreviated the posts as 'L', 'M', and 'R'. To help you follow the sequence of events, the curvy thin arrow shows the order in which the individual instances of `fact` print out the one ring they move.

5 Conclusion

Recursion is never easy the first time you see it. I remember the first time I saw references to it, it didn't just seem confusing, it seemed like someone was trying to pull a fast one. It was the computer equivalent of a magician pulling a rabbit out of a hat. How could the mechanism for solving a problem use the method for solving that problem? But one day it all sort of fell into place. I realized that not only was no one pulling a fast one, recursion is actually quite simple, and very elegant. The key is to stick with it and to look at it from multiple points of view.

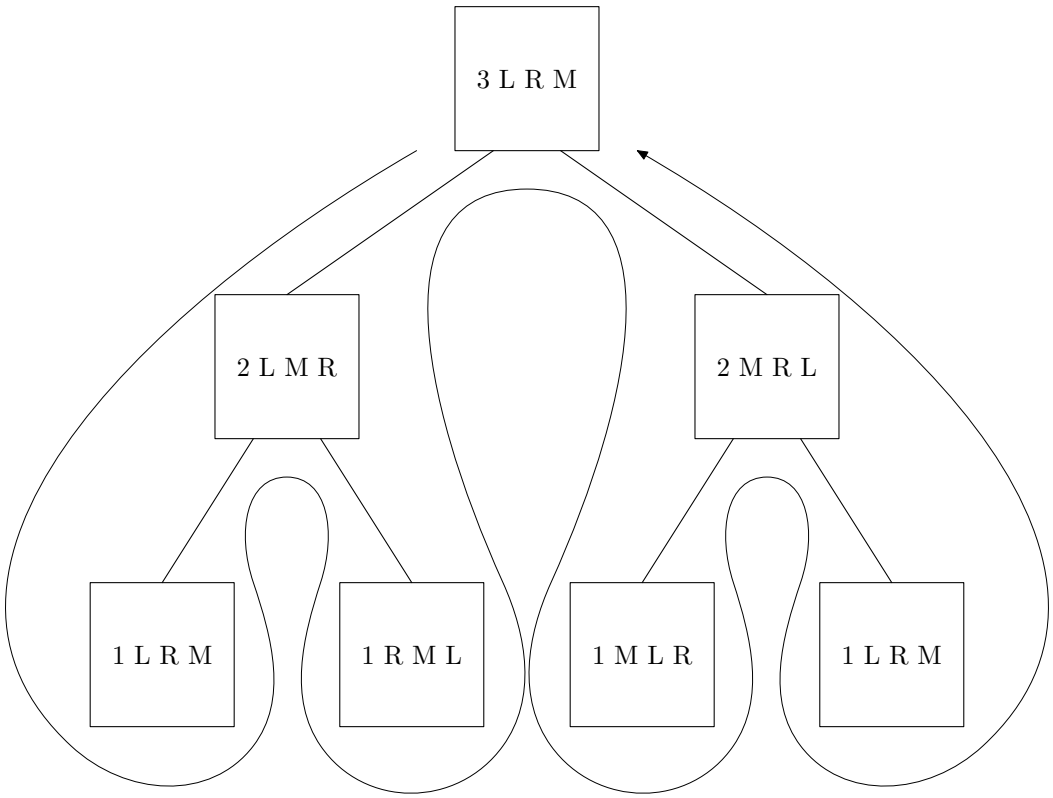

Figure 7: Memory Blocks for All `tower` Calls