Recursion Reading

Tim "Dr. T" Chamillard

What's Recursion Anyway?

Let's start by defining recursion. In its most basic form, recursion occurs when have a method call itself. Although it's possible to actually get recursion indirectly (method A calls method B, which calls method A), it's much more commonly used in the basic form.

There are two characteristics of problems that are well suited to a recursive solution. First, we need to be able to solve the problem by using the solution to the same problem that's a smaller size. Basically, the method will call itself with a smaller version of the same problem, then use that result in some way to generate the problem solution. Second, the problem has to have a *termination condition* or *base case*. In other words, at some point the problem needs to get small enough that we can just solve the problem without trying to solve a smaller version of the problem. Perhaps an example will make this clearer.

The Standard Example

The standard example people typically use to introduce recursion is calculating the factorial of a number. The factorial of a number n can be defined as the product of all positive integers less than or equal to n. For example, 5! = 5*4*3*2*1. For our purposes, though, it's more useful to observe that n! can also be defined as n * (n-1)!; in other words, 5! = 5 * 4!. That certainly looks like it meets our requirement that we can solve the problem using a solution to a smaller version of the problem, but what's the termination condition? It turns out that the mathematicians have (for good reason) defined 0! to equal 1. That means when we need to calculate 0!, we know the answer is 1 without having to solve anything else. Notice that factorial is not defined for negative numbers.

Let's look at a method that recursively calculates the factorial of a number that's 0 or greater.

```
int factorial(int n)
{
    // check for termination condition
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * factorial(n - 1);
    }
}
```

The first thing we do is check for the termination condition, that we're calculating 0!. If we are, we simply return 1. Otherwise, we calculate n * (n-1)!. Let's discuss some of the underlying mechanics behind how this works, then we'll revisit calling this method with a number other than 0.

When a method is called in C#, information about the method is pushed onto the *call stack*. We've already talked about stacks, but as a brief reminder think of a stack as a pile of plates. Using stack terminology, we can push a plate onto the top of the stack and pop a plate from the top of the stack. The stack therefore keeps track of everything we've added to and removed from it, with the special property that the order of the things we pop from the stack is exactly opposite of the order in which we pushed things onto the stack. This is particularly useful for method calls (and many other things!), because we call methods in a program in a particular order and we need to return from the last method we called first, then the next to last method, and so on.

Remember that we can use a class to instantiate as many objects of that class as we want. All those objects have the same attributes and behaviors as the class, but they're unique instances of the class. Although it's not perfectly accurate, think of a method as a special kind of class, where the method defines the attributes (local variables) and behaviors (the statements in the method). A method call to that method creates a new instance of the method and pushes it onto the call stack. With that idea, we can return to our factorial problem and walk through a more interesting example.

Let's call the factorial method with 3. Here's what happens:

1. The call to factorial(3) is pushed on the call stack (let's call this temporary method instance Call1)
2. Call1 needs to calculate 3*2!, so it calls factorial(2).
3. The call to factorial(2) is pushed on the call stack (let's call it Call2)
4. Call2 needs to calculate 2*1!, so it calls factorial(1).
5. The call to factorial(1) is pushed on the call stack (let's call it Call3)
6. Call3 needs to calculate 1*0!, so it calls factorial(0).
7. The call to factorial(0) is pushed on the call stack (let's call it Call4)
8. Call4 can simply return 1, so it does (and is popped from the call stack)
9. Call3 can now return 1 (1*0! = 1*1), so it does (and is popped from the call stack)
10. Call2 can now return 2 (2*1! = 2*1), so it does (and is popped from the call stack)
11. Call1 can now return 6 (3*2! = 3*2), so it does (and is popped from the call stack)

See how we've used the method calling itself with smaller versions of the factorial to solve the problem? That's what recursion is all about!

Algorithmic Complexity

In the Big-O Reading, we said "We regularly find ourselves looking at numbers of loop iterations (or recursions, which we'll see later) to determine algorithm complexity." For recursive methods, we need to analyze how many times we recursively call the method to analyze the algorithmic complexity of the implemented algorithm.

For our recursive factorial method, we make n recursive calls to calculate (n – 1)! down to 0!. That means our factorial algorithm is O(n).

Why Not Use a Loop?

It's actually the case that any problem we can solve with recursion can also be solved with a loop (though we might need to also use a separate stack in our loop solution). We can certainly solve our factorial problem using a loop:

```
int loopFactorial(int n)
{
    int factorial = 1;
    for (int i = 1; i <= n; i++)
    {
        factorial *= i;
    }
    return factorial;
}
```

So how do we know when to pick recursion over a loop? Unfortunately, this is really a judgment call rather than a set rule. The loop version would be faster for factorials because we save both the time required to execute each method call and the space on the call stack for each method call. On the other hand, once you get used to the ideas behind recursion the recursive solution to some problems – like this one – seems more intuitive based on the structure of the problem.

Algorithmic Complexity

It's easy to see that the loop version of our factorial calculation is also O(n) since the loop iterates n times. In fact, loop and recursive implementations of the same algorithm will always have the same time complexity. We note, however, that the loop implementation has O(1) space complexity while the recursive implementation has O(n) space complexity (because of the n method calls on the call stack).

Infinite Recursion

We know that we'll get an infinite loop if we write a while loop for which the Boolean expression never evaluates to false. Similarly, if we call methods recursively and the termination condition is never met, we end up with *infinite recursion*. It's exactly the same idea – we never realize we should stop.

The most common causes of infinite recursion is forgetting to check or incorrectly checking the termination condition in the recursive method. You should obviously be pretty careful about this!

Tail Recursion

There's actually a special type of recursion called *tail recursion*. In tail recursion, the very last operation in a method (at least for some control flow through the method) is the recursive call.

Why do we care about tail recursion? Because tail-recursive methods are easy for a good compiler to optimize so we don't actually use up all that call stack space we discussed above; less importantly, tail-recursive methods are very easy to turn into loops.

You should note that our factorial method is NOT tail-recursive, even though the recursive call falls at the end of the last statement for the non-termination case. It's not tail-recursive because we need to multiply n by the value returned by the recursive call AFTER we get that value, so there's still more work we do in the method after the recursion returns to us.

Recursive Binary Search

Recall that the `IndexOf` method in our `OrderedDynamicArray` class used binary search to find a particular element in the array (or to discover that the element wasn't in the array). Although we used a loop for that method, binary search is very well suited to a recursive solution. Consider the recursive method below:

```
private int BinarySearch(int searchValue, int[] searchArray,
      int lowerBound, int upperBound)
{
    // check exhausted array termination condition
    if (lowerBound > upperBound)
    {
        return -1;
    }

    // check found value termination condition
    int middleLocation = lowerBound + (upperBound - lowerBound) / 2;
    int middleValue = searchArray[middleLocation];
    if (middleValue == searchValue)
    {
        return middleLocation;
    }

    // do recursive call on appropriate part of array
    if (middleValue > searchValue)
    {
        return BinarySearch(searchValue, searchArray, lowerBound,
            middleLocation - 1);
    }
    else
    {
        return BinarySearch(searchValue, searchArray, middleLocation + 1,
            upperBound);
    }
}
```

This recursive method actually has two termination conditions: the element isn't in the array and the element is found at the middle location of the current range. We check those termination conditions with the first and second if statements, respectively.

If neither termination condition is met, we recursively call the method with a smaller range to search in the array. If the middle value we're checking is greater than the value we're searching for, we want to search the lower half of our current search range, so we change the upper bound for the recursive call to be just below the middle location we just checked. Otherwise, we need to search the upper half of our current search range, so we change the lower bound for the recursive call to be just above the middle location we just checked.

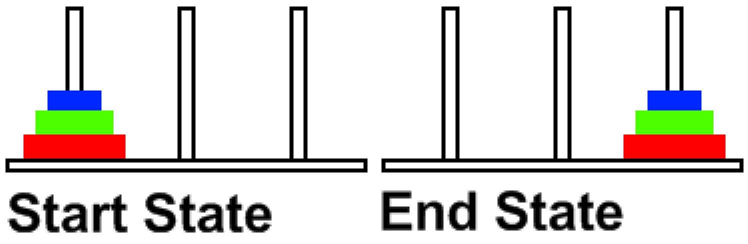
Algorithmic Complexity

This recursive implementation has the same O(log n) time complexity as the loop implementation we used earlier. It does, however, also have O(log n) space complexity as compared to O(1) space complexity for the loop version.

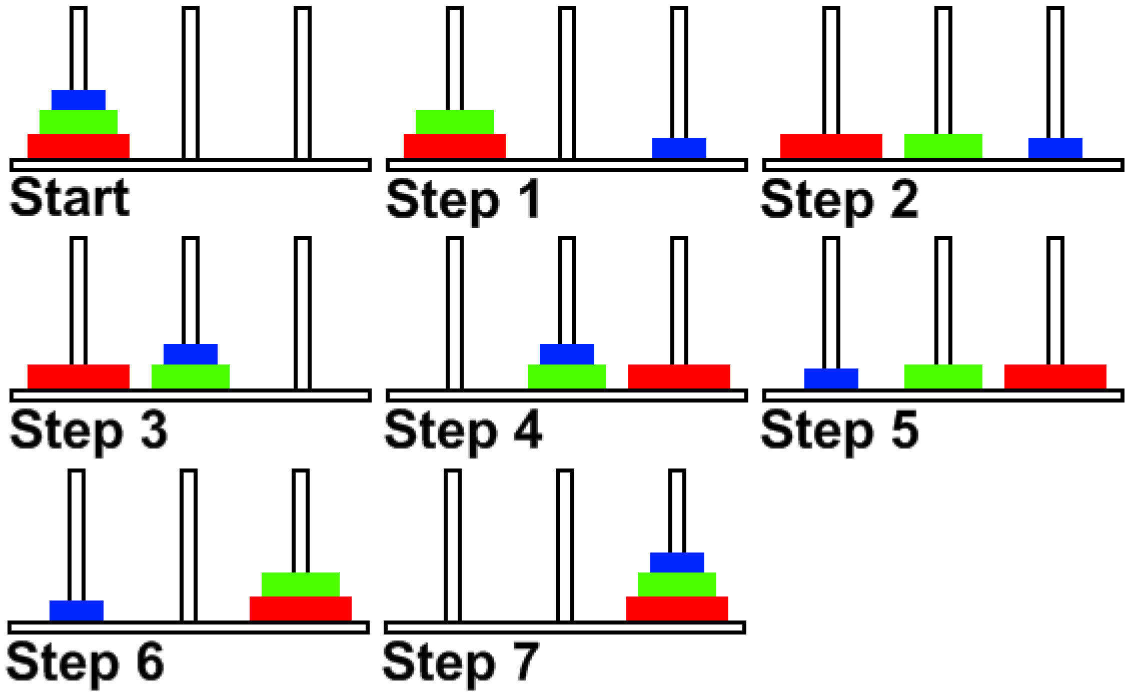
The Towers of Hanoi

The last recursion example in this reading is a problem called the Towers of Hanoi. This problem is interesting for a couple reasons: there's no obvious solution to the problem using loops and it's an O(2^n) algorithm, which is much worse than we'll typically deal with.

Here's how the Towers of Hanoi game works. The goal is to move a stack of disks from one peg to another peg without ever putting a larger disk on top of a smaller one. An example of the starting and ending states for a 3-disk game are shown below.



So how do we actually get from the start state to the end state? Here's the optimal solution for the 3-disk problem:



Notice that it takes us $2^3 - 1$ moves to do this; this is definitely an $O(2^n)$ algorithm!

So, how do we solve this problem using a method? Specifically, how do we break this problem into smaller versions of the same problem (we'll worry about the termination condition in a while)? The solution is very hard to see if you look at each step individually, but consider the following description of the steps:

1. Move the top n-1 disks from start peg to the open peg (Steps 1-3 above move the top two disks to the middle peg)
2. Move the nth disk to the destination peg (Step 4 above does this)
3. Move the n-1 disks from the open peg to the destination peg (Steps 5-7 above do this)

So what's our termination condition? We only do the above three steps if the number of disks we need to move is 1 or greater, so we terminate when the number of disks to move is 0.

Here's a recursive method that solves the Towers of Hanoi problem:

```
private void Hanoi(int n, int start, int destination, int open)
{
    // only solve if not termination condition
    if (n != 0)
    {
        // move n - 1 disks to open peg
        Hanoi(n - 1, start, open, destination);
        // move n disk to destination peg (note it just prints a message)
        Console.WriteLine("Moving disk " + n + " from peg " + start +
            " to peg " + destination);

        // move n - 1 disks to destination peg
        Hanoi(n - 1, open, destination, start);
    }
}
```

Pretty slick, huh? There are just a couple of subtle details to pay attention to. When we call the method, we specify the start peg, the destination peg, and the open peg for the n disks. Our initial call to the method will use 0, 1, and 2 for these peg values. For the first recursive call, we treat peg 2 as the open peg and peg 1 as the destination peg (look at the order of the arguments in the method call). If you look at the 3-disk solution above, you'll see that we moved the top two disks to peg 1 (the middle peg) for this step. After that recursive call returns and we move the n^{th} disk, the n-1 pegs are sitting on the original open peg and the n^{th} disk is sitting on the original destination peg. For the second recursive call, we have to move the n-1 pegs from the original open peg to the original destination peg (on top of the n^{th} disk). Again, look at the order of the arguments for the second method call to see how we do this.

Conclusion

Recursion is actually a very elegant way to solve a large number of problems, and it's regularly used for Artificial Intelligence to traverse search trees. Although we do need to be careful to check for termination conditions and to reduce the size of the problem we're solving on each recursive call, recursion is a very powerful technique that you need to understand to program games successfully.