Chapter 15
Recursion

Hello!

Recursion implies that something is defined in terms of itself. We will see in detail how **code** can be recursive (a method that calls itself), and we will also explore how **data** can be recursive. Let's start with an example of a recursive method: calculating the factorial of a number.

Recursive Methods

We will first take a look at how a method can be recursive, by exploring methods that use recursion to solve a programming task. The general approach is to keep making the problem smaller on each successive call.

Example: Factorials
The factorial of a non-negative integer can be defined in many ways. Here are two definitions that are not recursive:

Alternative text for the below formula: "n factorial is the product of 'i' for i = 0 through n".

$$n! = \prod_{i=1}^{n} i \qquad \text{n! = n*(n-1)*(n-2)*...*3*2*1}$$

We could also define it as:

Alternative text for the below formula: "n factorial is equal to 1 when n = 0, and n times ((n minus 1) factorial) when n > 1".

$$n! = \begin{cases} 1 & when\ n = 0 \\ n * (n-1)! & when\ n > 1 \end{cases}$$

The definition of factorial involves a factorial expression, namely (n-1)!. This definition is also pretty convenient, as it gives us a specific expression to use based on the specific value of n we are given. This tells us *what* the value of n! is, instead of showing us *how* to calculate it. Let's look at the code for recursively calculating the factorial of a number:

```
public static int factorial (int n) {

      //base case: 0! == 1. no recursion.
      if (n==0) return 1;

      //recursive case: n! == n*(n-1)!
      else      return n * factorial(n-1);
}
```

We see the recursive call: factorial(n-1) . This is what makes the method recursive, which just means that the method contains a call to itself in its body. There are a few key points to notice that will show up in any recursive method:

The Base Cases
Base cases represent the non-recursive uses of our method. When we calculate factorial for n==0, we already know the answer (1) and can simply return this result by rote. Base cases are our chance to 'escape' from the recursive calls, and thus must be checked first. It is very similar to the guard check of a while loop or for loop – before each iteration, we check if it's time to quit yet.

The Recursive Cases
Recursive cases represent the recursive uses of our method. When we calculate factorial for n>0, we are basically realizing that we don't have the answer memorized for this particular n, but we know how to make the problem smaller: think of n! = n*(n-1)!. We already know n, because it was given; (n-1)! is a "smaller" problem, because (n-1) is closer to our base case, when we just know the answer. So we go ahead and recursively call factorial(n-1), and then multiply the result by n to figure out what n! is.

If we only had recursive cases, then we'd always be calling the method again, and we'd never finish calling the method! Just as we can (accidentally?) write an infinite while-loop or infinite for-loop, we can also write an infinitely recursive method, and it's just as bad.

Calling the code
We can test our factorial code with the following testing harness, which accepts the value to test as a command-line argument. (You could instead create a java.util.Scanner and read from System.in).

```
public static void main (String[] args) {
      int n = Integer.parseInt(args[0]);
      int nfact = factorial (n);
      System.out.println("n="+n+"\nn!="+nfact);
}
```

Each Call is Distinct
When a method calls itself, the two calls are distinct: they were each invoked by passing particular parameters around; each one may create its own local data. The values of all this local data are stored in different locations in memory for each recursive call. It might help to think of each call as using a different copy of the code that just so happens to do the same thing:

```
public static int factorial  (int n) { if (n==0) return 1; else return n*factorial1(n-1); }
public static int factorial1 (int n) { if (n==0) return 1; else return n*factorial2(n-1); }
public static int factorial2 (int n) { if (n==0) return 1; else return n*factorial3(n-1); }
public static int factorial3 (int n) { if (n==0) return 1; else return n*factorial4(n-1); }
public static int factorial4 (int n) { if (n==0) return 1; else return n*factorial5(n-1); }
…
```

It is easy to understand that each call to a different method above has its own local data (including its parameters), and occupies its own space on the stack. If we wrote enough versions of factorialX, this would work and would not be recursion. We have enough extra copies to calculate factorial up to 4 in the above sample, but suppose we wanted to call factorial on just slightly larger numbers? There would never be enough copies to handle all possible inputs. This is the same reason we would write a while loop instead of a lot of nested cut-pasted if statements. It's not just cleaner and easier to write, it's actually more powerful.

Another thing to consider is that any particular method that calls another method basically pauses itself, lets the other method complete its task and return with a value, and then the caller gets to un-pause right where it was, and it continues its task. This same idea happens with a recursive call:

When a method calls itself recursively, the original call pauses, then the recursive call runs completely from start to finish; lastly, the original call finishes.

Okay, so we've got our single version, which calls itself:

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

Note: each call to factorial *still* has its own local data (like parameters). Whenever we have one method call another, it has to pause; wait for the other one to finish; and then it un-pauses and completes its task right where it left off, using the result of the called method as necessary. If we are paused waiting on a recursive call, and multiple other recursive calls occur as a result, we'll still just have to wait (stay paused) until they all finish and the call we made is able to return to us. So it makes sense that the same thing (pausing until the call completes) should happen when a method calls itself.

Why does recursion work?

When we use recursion, it's as if we get the solution for free: in order to solve a problem, we just call the solution we're writing to write the solution. How is this fair? It all comes down to that requirement that the recursive call must be a "smaller" problem, which we can also phrase as saying that we must make progress towards one of our base cases.

There's the notion of mathematical induction: If I want to show that some property P(n) holds for all positive integers n, let's try to show two things:

base case:	P(1) is true	the property is true for n==1
recursive case:	P(k) → P(k+1)	if I know P(k) is true, I can show why P(k+1) is true.

Now, if I wanted to show why P(4) is true, I can say that I know P(1) is true; then, we show why P(1) → P(2); why P(2) → P(3); and lastly, why P(3) → P(4). The beauty is that these two pieces of information are enough to show that P(n) is true for any positive integer n. We don't have to do an endless number of proofs!

Now, let's apply this notion to method recursion. Using the example property P above, we have to think of the goal as proving P(4), and thinking in reverse:

- If I only knew P(3), then I could show how P(3) → P(4).
- If I only knew that P(2) were true, I could show how P(2)→P(3)→P(4).
- If I only knew that P(1) were true, I could then show how P(1)→P(2)→P(3)→P(4).
- Well, hey, we happen to know that P(1) is true! So apply that information (the calculated results) to show that P(1)→P(2)→P(3)→P(4), and now we can see that P(4) is true.

Because each step got us closer to the base case (we always knew P(1)), we would eventually find the base case and be able to stop the recursive calling, completing all our calls and ending up with the answer.

So how can we check that we're getting closer to a base case? It's all about inputs (parameters). For many recursive methods, we have numerical variables as parameters. These same variables tend to be used to phrase our base cases. Whatever value we need to trigger the base case, and whatever value we currently have instead, will dictate the change in value that we'd need to see in order to know that we are getting closer to a base case. In our factorial example, the numerical parameter was named n. The base case is when n==0, and we would start with a larger positive number. Our recursive call used the value (n-1), so it is pretty clear that we're getting closer to our base case. If we were to plug in a negative number (say, -5), our code would still blindly call factorial on the next-lower number (-6). This would not be closer to our base case. This would recursively call itself infinitely, or as many times as possible before our computer ran out of memory to store the local data of each successive call (all those parameter values have to go somewhere while the recursive calls are busy!). In Java, you'd see a StackOverflowError.

Your Turn!

- Plug in a negative number to our recursive factorial definition.
- "fix" the factorial definition by treating all negative numbers as a base case: if the input is negative, just return -1 instead of performing any recursive calls.
- Let's write some simple definitions with recursion. True, there are easier ways to write these methods, but let's use recursion anyways, while the problems are simple.
    - We are comfortable writing the sum of all numbers from zero to n with a for-loop; now, try to write a method that sums all the values from zero to the single integer parameter that uses the following knowledge:
        - sum(0)=0
        - sum(n)=n+sum(n-1).
    - Suppose we don't have the % operator in our language of choice, and we'd like to use recursion to figure out if a number is divisible by 2. Use recursion to write an isEven method, given that:
        - isEven(0) = true
        - isEven(1) = false
        - isEven(n) = isEven(n-2)

Note: Recursive methods are far more common when our data are also recursive. The recursive structure of our values often beautifully lines up with the recursive calling structure of the methods that perform calculations or manipulations over those recursively-defined structures. We will see a few examples of these once we introduce data recursion later in this tutorial.

Mutual Recursion (Indirect Recursion)

Sometimes recursion happens in a roundabout way: suppose that method A sometimes calls method B, and that method B sometimes calls method A. If we only looked at method A's code, we might not realize that calling A can result in many more calls to method A! Here is a very small example that can showcase it:

```
public static boolean isEven(int n) {
        if (n==0) return true;
        else return (! isOdd(n-1));
}
public static boolean isOdd(int n) {
        if (n==0) return false;
```

```
        else return (! isEven(n-1));
    }
```

The isEven method calls isOdd as its 'recursive' case. isOdd calls isEven as its 'recursive' case. If we called isEven on 5, we'd work thus:

- isEven(5): let's figure out if 4 is odd.
- isOdd(4): let's figure out if 3 is even.
- isEven(3): let's figure out if 2 is odd.
- isOdd(2): let's figure out if 1 is even.
- isEven(1): let's figure out if 0 is odd.
- isOdd(0): false!
- return the false value through all those calls
  → 5 is not even after all.

Certain care must always be taken with recursive calls to ensure we're approaching a base case. When the recursion is indirect, it can be easy to forget that we still must make progress towards our base case. But it is just as necessary to successfully use recursion, be it direct or indirect.

Data Recursion

Recursion can also occur in our data representations, not just in our instructions. Data recursion happens whenever we are defining a new type of value, and we state that one of the fields (some state) is of the type we're defining. In this way, the structure of our values is defined in terms of what the structure of our values looks like.

We will explore one very simplified version of a data structure in order to see data recursion. When you take a data structures course (CS 310, here at George Mason), you will get to learn about far more complex and interesting data structures. It gives you a very powerful set of tools to approach your programming tasks! Data structures let you build upon decades of others' programming experience of the 'correct' way to solve problems, and the better you understand them, the far stronger programmer you will become.

Example - Lists

If we didn't have arrays in our language, we might want to create a list of integers by stating that:

- a list node is a single number followed by another list.
- a list is just a list node, which might be null.

Now, if we create a new class for a list node, and another class for a list (which will just be a list node and a bunch of methods), we might come up with the following definition:

```
public class IntList {
        public IntListNode head;
        public IntList() { head = null; }
}

public class IntListNode {
        public int val;
        public IntListNode next;
        public IntListNode(int val, IntListNode next) {
                this.val = val;
```

```
            this.next = next;
        }
}
```

Throughout, let's try not to get confused between the IntList class and the IntListNode class. An IntList represents the entire thing, and really just gives us a place to put methods that want to view the whole structure (it also lets us gracefully use null to represent an empty list, so that we don't have to check for null all over the place). An IntListNode represents just one link in the chain.

Your Turn!

- Let's get our code base started, which we will add to later. Starting with the above code, overload the IntListNode constructor to have one that only asks for the int val, and always sets the next field to null.

If we wanted to represent the three numbers 3,5,7 in an IntList, we could then do the following:

```
IntList ill = new IntList();
ill.head = new IntListNode(3);
ill.head.next = new IntListNode(5);
ill.head.next.next = new IntListNode(7);
```

We created each node (each spot in the list), and linked them together by using enough .next's to chain them together. Clearly we'd like a more convenient way to add items to our list. Let's add a method to our IntList class to do so:

```
// IntList add: drives the IntListNode add.
public void add(int v){
      if (head==null)
            head = new IntListNode(v);
      else
            head.addToNode(v);
}
//IntListNode addToNode. This method is recursive!
public void addToNode(int v){
      if (next==null)
            next = new IntListNode(v);
      else
            next.addToNode(v);
}
```

Now we can create our 3-element list a bit more simply:

```
IntList ilist = new IntList();
ilist.add(3);
ilist.add(5);
ilist.add(7);
```

Notice how we are writing code that deals with recursive data (the addToNode method), and the definition was recursive. This is a common occurrence. We could have defined a non-recursive version of add:

```
//IntList add_iterative
public void add_iterative(int v){
  // If the first node is empty(null), then add it specially.
  if (head==null){
    head = new IntListNode(v);
    return;
  }
```

```
    //keep stepping through nodes, as long as the next one also exists.
    IntListNode curr = head;
    while (curr.next != null){
      curr = curr.next;
    }
    // we know curr is the last existing node, so reassign its next field.
    curr.next = new IntListNode(v);
  }

  …
  // usage: pretty much like the recursive version.
  IntList ilist = new IntList();
  ilist.add_iterative(3);
  ilist.add_iterative(5);
  ilist.add_iterative(7);
```

There is always a tradeoff between iterative and recursive versions of a method. Iterative versions tend to be faster (whether it's imperceptible or severely apparent), but recursive definitions are often a more natural way to phrase a solution. The successive recursive calls are a natural place to 'store' temporary calculations. Note that there was no `current` node in the recursive version; it was implicitly tracked by calling `add(..)` on a different IntList object each time. The this of each successive recursive call effectively was the curr node.

This time, the recursive version should run in almost identical time. The only overhead is in the effort spent calling a method versus the effort spent going to the next while loop iteration, and that difference is not huge.

We can write more recursive definitions on our definition. Let's consider pretty-printing our list, in iterative fashion first. We'd like something such as "[3,5,7]" to be the result.

```
// IntList toString
public String toString(){
      String s = "IntList[ ";
      IntListNode curr = head;
      while (curr != null){
            s += curr.val+",";
            curr = curr.next;
      }
      return s+"]";
}
```

This task is perhaps slightly better suited to an iterative version (as above), because the very first time would be different from all successive recursive calls (it has to print square brackets). When the first time to call a method has a few extra details (or should do a few more/fewer things), we might want to write a "driver" method. This has nothing to do with recursion per se, though it does lend well to recursion:

```
// IntList toString, version 2 (with worker in IntListNode class).
public String toString(){
      // handle empty list case
      if (head==null){return "[]";}
      // just this first node must add [ ]'s.
      String s = "["+head.val;
      // all further nodes are handled normally
      s += head.next.toString_worker();
      s += "]";
      return s;
}
```

```
// IntListNode: helper method, should only be called by IntList's toString.
public String toString_worker(){
	// include this node in the String
	String t = ", "+this.val;
	// base case: no more nodes
	if (this.next == null) {
		return t;
	}
	// recursive case: get entire rest of list's String representation
	else {
		t += this.next.toString_worker();
		return t;
	}
}
```

We would call IntList's toString the driver method, because it is used to initiate (drive) the toString activities. All the interior calls happen recursively through IntListNode's toString_worker. Another common time to use a driver method is if we have a few parameters that are tweaked on each successive call, but the initial call we'd always know their starting values (perhaps zero, false, or someArray.length). The driver method can offer a shorter parameter list, and then call the interior worker-version with these known starting values directly, without forcing the user to remember what those initial values ought to be. In our case above, the only difference between toString and toString_worker was just the body of the method.

Your Turn!

- There is a bug in IntList's toString as shown above! ☹ It turns out that it does not work for length-one arrays, though it does work for length-zero and length 2+ lists. Figure out why, and fix it.

- The following methods can all be defined over lists. For each one, write either an iterative implementation in IntList, or else write a driver in IntList and a recursive worker in IntListNode.
  - Write the contains method, that accepts an int parameter and returns a boolean value describing whether this node or one it links to has that int value as its val.
  - Write the insert method, which accepts an index and a value to insert at that location. It should take over that index, and push all following values over one spot.
  - Write the remove method, which accepts an index, navigates to that spot, and removes that node.
  - write the addAll method, which accepts an int[] and adds each of its items to your list in the order given. (They all end up at the end of your list).
  - Write the method size, with no parameters : it returns the length of the list as an int.
  - Write a static version of size that accepts an IntList as a parameter.
  - Write the maxValue method, which has no parameters and returns the largest value in the list. It's okay if it crashes on empty lists.

Thoughts

If we didn't have the IntList class, and just used the IntListNode class, we could still effectively get all of our operations implemented. But the only way to represent an empty list would be with the whole thing being null, instead of an instance variable being null as in our implementation. Instead of all our if (head==null) checks happening inside methods, any code that wanted to use our IntListNode would be responsible for knowing that null was a possible value, and to first check for null before using it. This is error prone and laborious usage, so we chose to have the IntList class. It serves as a facilitator between the nodes on the inside representation, and the abstracted view that outsiders have of a list. It is also a

very convenient place to deal with the "first iteration is different" aspects of any of our operations, as it's truly a different place with different things available.

That's a common theme with Abstract Data Types such as lists: some inner recursive structure utilizes many objects to define a larger entity, and the whole thing is hidden behind a clean API. Java's Collections Framework provides many nice ADT's as interfaces (the API), and then the messy details are hidden inside class implementations (where the data recursion occurs).

Bonus Things

Some jokes and takes on recursive structures.

Comics	http://xkcd.com/878/
	http://www.buzzfeed.com/scott/sad-keanu-recursion
image searches	https://www.google.com/search?q=recursive+image&tbm=isch