# CPSC 211 - Spring 2007
# Recursion Notes

created by:
Mark Crowley

Do not turn over, answer is on the backside

Recursion is simply the technique of writing a method that calls itself. In order to be useful and avoid infinite recursive calls, causing an out of memory error, a recursive method needs two components:

**A Base Case :** This is the simple case that can be solved immediately. No further recursive calls to the current method are made in the base case. This is where the recursion ends.

**One or more recursive cases :** Some work may be done on the input. Then the input is reduced or broken up in to smaller pieces and the current method is called one or several times. Each call to the current method must use a smaller input so that the recursion progresses towards the base case.

## Examples of Recursion

Recursion is confusing at first because it doesn't seem to show up in the real world. It requires the ability to make copies endlessly, here are some examples. Can you think of any others? :

**GNU unix tools** - The name of this popular open source unix tools (Gnome, gnuplot, g++, GIMP) is a recursive acronym. GNU stands for "GNU's Not Unix". But what does GNU stand for? Well, it stand for "GNU's Not Unix". But what does...

**Pyramid Schemes** - These are sort of recursive. They involve different people, but each person is asked to do the same thing. You tell 10 friends to all give you $100 and tell their friends to do the same to them. Then everyone give 10% of their money to the person who told them. Everybody wins! (well...really only the people at the beginning win, but in a computer program you don't care about all the other invocations :) )

**Chain Letters** - Send this email to 10 friends and your wishes will come true!

**Mathematical Induction** - Inductive proofs in math are recursion. The base case states what you assume to be true. The inductive case defines a pattern that must be true given the previous case is true, starting with the base case. QED.

## Loops and Recursion

Loops are very similar to a simple type of recursion called *tail recursion*. Consider the following very inefficient loop to sort a list:

```
uList = random List of numbers
sList = empty List
while  uList.size() > 0
   x = uList.popMin()
   sList.append(x)
end while
```

```
def whileR(uList, sList)
   if uList.size() <= 0
      return sList
   else
      x = uList.popMin()
      sList.append(x)
      sList = whileR(uList, sList)
      return sList
```

Here `popMin()` searches the list for the smallest element and removes it from the list, returning the value. The recursive method on the right does exactly the same thing the loop on the left is doing. It does its work then at the end calls itself with a reduced parameter (`uList`) and the running answer (`sList`). In a pure functional language, like LISP, all loops would be programmed this way. Most compilers automatically turn this kind of recursion into a loop because it is more efficient to run.

## Multiple Recursion

Recursion becomes more complicated (and much more powerful) when multiple recursive calls are made within the method. Assume our pseudocode List class also has a simple `get(index)` method as well as the following methods:

**list.frontHalf()** - return a list of the first half of the elements. $\lfloor n/2 \rfloor$ of them.

**list.backHalf()** - return a list of the last half of the elements. $\lceil n/2 \rceil$ of them.

**Question:** In pseudocode, write a recursive method called `recurseSort(uList)` that takes an unsorted list and returns a sorted list by splitting the list into halves to be sorted and recombining them. Don't worry about being efficient, just keep it simple. Make sure there is a base case that 'sorts' the smallest list, should it be size 0, 1 or 2?

## Homer Simpson's Shopping Algorithm

To get an intuition for multiple recursion, think of what Homer would do...
If Homer understood recursion when he was going grocery shopping for Marge, and had the ability to spawn copies of himself, he could make shopping much easier.

**Homer is given** a shopping list of some length.

**(Base Case)** If the shopping list had only 1 thing on it, Homer would just go get it and put it in his shopping cart. But its longer than that, and he's lazy, so...

**Homer spawns** two copies of himself complete with matching shopping carts

**Homer rips** the shopping list in half and gives one half to each Homer and tells them to simply follow Homer Simpson's Shopping Algorithm. Homer waits impatiently for them to return, tapping his foot madly and whistling.

**Two Homers** soon return with full shopping carts. Homer empties the two carts into his and the spawned Homers get a puzzled look on their faces and wink out of existence.

**Homer returns** his shopping cart to the Homer who spawned him and...doh!

---

**Answer:** Note, this algorithm is still very inefficient but is much better than the previous example. Many good sorting algorithms find ways to intelligently improve splitting and merging.

```
recurseSort(uList)
     if uList.size() == 1
          //Base Case
          return uList
     else
          //Recursive Case
          sList1 = recurseSort(uList.frontHalf())
          sList2 = recurseSort(uList.backHalf())

          //Now combine the Lists by adding the lowest elements in turn to a new List
          sList3 = new List
          done = false
          x,y = 0
          while not done{
               //If one list is empty, append the rest of the other list to the answer
               if sList1.size() == 0
                   sList3.append(sList2)
                   done = true
               else if sList2.size() == 0
                   sList3.append(sList1)
                   done = true
               else
                   //Add the lowest value to the answer and remove it
                   x = sList1.get(0)
                   y = sList2.get(0)
                   if x < y
                        sList3.append(x)
                        sList1.remove(0)
                   elsif y < x
                        sList3.append(y)
                        sList2.remove(0)
                   else
                        sList3.append(x)
                        sList3.append(y)
                        sList1.remove(0)
                        sList2.remove(0)
          }
          return sList3
```