Reminders and Announcements

- Project 8 spec now out
  - BigInt.java, Integer of arbitrary size.
  - Milestone: Implement addition. You may use loops if you want.
  - Final: Implement add, sub, mul, div, mod using recursion only. Each loop you use will cost you some points.
What is recursion?

- Syntactically, when a function contains a call to itself.
- Conceptually, it is a problem solving technique that allows us to solve a larger problem by breaking it up into sub problems.
Recursive functions

There are 3 important parts of a recursive function:

- The arguments: If you need to pass information across function calls, you will do so using the arguments.
- Base case: Similar to the termination condition of a loop. Without this, you will run into infinite recursion.
- Recursive case: This is where all the real work happens. Similar to the body of a loop.
Factorial

- Mathematically:
  - $\text{factorial}(n) = 1 \times 2 \times 3 \times \ldots \times n$.

- Recursively:
  - $\text{fact}(n) = n \times \text{fact}(n-1)$

- Note how we 'break' the larger problem into a sub problem and solve that first. This is a key ingredient of recursion.
public int factorial(int number) {

    if (number < 0) { //Invalid input
        return -1;
    } else if (number == 0) { //base case
        return 1;
    } else { //recursive call to a smaller problem
        return number * factorial(number - 1);
    }
}
How factorial works...

factorial(3) -> this goes to a recursive case that returns (3 * factorial(2))
factorial(2) -> this goes to a recursive case that returns (2 * factorial(1))
factorial(1) -> this goes to a recursive case that returns (1 * factorial(0))
factorial(0) -> this goes to a base case that returns 1

Going all the way up we see that:
factorial(3) = (3 * (2 * (1 * (1)))) = 6.
Infinite recursion

- Just like we can write infinite loops, we can also write infinite recursive functions.
- A missing base case could lead to infinite recursion.
- The wrong size argument in the recursive call could also lead to infinite recursion.
- Infinite recursion is not always a bad thing!
In java, infinite recursion will lead to *stack overflow*.  
Java reserves a fixed size stack for your program. Too many recursive calls will cause you to run out of stack space.  
Stack overflow can also be caused if you just have too many recursive calls. You may run into this while doing project8.
public int search(int nums[], int val, int start, int end) {
    for(int i=start;i<end;++i) {
        if(nums[i]== val) {
            return i;
        }
    }
    return -1;
}
public int search(int nums[], int val, int start, int end) {

    if(nums[start] == val) //base case
        return start;
    else if(start == end) //base case
        return -1;
    else //recursive case
        return search(nums, val, start+1, end);
}

Search using recursion
Merge Sort

- Once again, we break the larger problem into sub-problems.
- To sort a list of numbers, we break it into two halves, sort each and merge the result.
To traverse a tree, look at the 'root' element, then recursively traverse each 'branch'. Or try a different traversal order.
Notes on recursion

- Pay close attention to your arguments.
  - If you think you need more arguments than the header provides, create a helper method with extra arguments!
- Don't forget your base cases.
- Before you start coding, write pseudocode on a piece of paper.
Linked List

- Recall that an array’s capacity is fixed once it is created or initialized.
- The LinkedList class in java.util used a linked list to implement a variable size list of objects.
- How does this work?
- We will study it by creating our own version of the LinkedList class which can store a list of objects of any class in order.
The Node Class

class Node {
    private Node next;
    private Object content;

    public void Node() {
        next = null;
        content = null;
    }
    public Object getContent(){
        return content;
    }
    public void setContent(Object c){
        content = c;
    }
    public Node getNext(){
        return next;
    }
    public void setNext(Node nextNode){
        next = nextNode;
    }
}
The LinkedList class

class LinkedList {
    private Node head;
    private Node iterator;

    public void LinkedList() {
        head = null;
        iterator = null;
    }

    public void addToHead(Object c) {
        Node n = new Node();
        n.setContent(c);
        n.setNext(head);
        head = n;
    }

    public void deleteFromHead() throws Exception {
        if (head == null)
            throw new Exception("Empty List");
        else
            head = head.getNext();
    }

    public void startScan() throws Exception {
        if (head == null)
            throw new Exception("Empty List");
        else
            iterator = head;
    }

    public boolean hasMore() {
        if (iterator.next == null)
            return false;
        else
            return true;
    }

    public void moveAhead() {
        iterator = iterator.getNext();
    }

    public Object getCurrentItem() throws Exception {
        if (iterator == null)
            throw new Exception("No Current Item");
        return iterator.getContent();
    }
}

Example

```
LinkedList list;
list = new LinkedList();
String s;
s = "test1";
List.addToHead(s);
s = "test2";
list.addToHead(s);
s = "test3";
list.addToHead(s);
```
list.startScan();
s = (String)
    list.getCurrentItem();
System.out.println(s);
while(list.hasMore()){
    list.moveAhead();
    s = (String)
        list.getCurrentItem();
    System.out.println(s);
}
Deletion

```
... list.deleteFromHead();
```
int puzzle(int n) {
    if (n == 1)
        return 1;
    if (n % 2 == 0)
        return puzzle(n/2);
    else
        return puzzle(3*n+1);
}

What are the sequence of recursive calls when puzzle(6) is called?