COMP 215 Exam #2

My name is:

- This is a 50 minute exam.
- The exam is open book and open notes. You are free to bring whatever materials you want to the exam. However, the rules are that (a) you cannot use an internet connection during the exam, and (b) you cannot use a Java compiler to help you on the exam (that is, you can't compile or execute any Java code during the exam).
- Write your answers in the spaces provided. Use extra paper to determine your solutions, if needed, then neatly transcribe them onto these sheets.
- Make sure you clearly write your name. Further, please sign the pledge at the bottom of this page.

Pledge: On my honor, I have neither given nor received any unauthorized aid on this exam.

Signed:

Problem 1 [20 points]

(a) On A5 and A6, it seems that most people developed their test code after first writing the assignment code. Many people wrote the tests in an iterative process where they first wrote a few test cases, then fixed their code, which suggested some other test cases to write, and so on. Is this best described as "black box" or "white box" testing? Breifly justify yor answer.

(b) Consider the following method:

```
// Return the maximum of the three input arguments.
public int maxOfThree (int x, int y, int z) {
      if (x > y) {
            if (x > z) {
                  return x;
            } else {
                  return z;
            }
      } else {
            if (y > z) {
                  return y;
            } else {
                  return z;
            }
      }
}
```

Suppose you run four tests on this method with the following (x, y, z) inputs: (3, 2, 1), (2, 1, 3), (2, 3, 1), (1, 2, 3). If they all pass, is this testing sufficient for you to have high confidence that this method works correctly? Why or why not?

Problem 2 [20 Points]

Many of the recursive methods we've seen have the following structure:

```
public class RecursionShell {
     public void recurse (ArrayList <Integer> intList) {
          if (intList.size () == 0) {
               System.out.println ("end of list");
          } else {
               Integer temp = intList.remove (0);
               System.out.println ("found a " + temp);
               recurse (intList);
               intList.add (0, temp);
          }
     }
}
```

(a) A proponent of functional programming would not like this code. What specifically would a "function head" object to?

(b) Imagine that we instead had a purely functional version of `ArrayList`, called `FuncArrayList`. `FuncArrayList` has a `remove ()` method that returns a `RemoveResult <T>` object (where `T` is the same generic type parameter used by the `FuncArrayList`). `RemoveResult` has two methods: `getRemovedItem ()` and `getResultingList ()`, both of which take no arguments and return the value implied by their names. With `intList` as a `FuncArrayList`, re-write the `else` clause:

```
     public void recurse (FuncArrayList <Integer> intList) {
          if (intList.size () == 0) {
               System.out.println ("end of list");
          } else {









          }
     }
```

Problem 3 [20 Points]

Consider the following class:

```
public class MyLinkedList<Data> {
     // A linked list holding elements of generic type "Data"

     private ANode head;

     private abstract class ANode {
          // An abstract list node
          ...
          private abstract int length ();
     }

     private class ListNode extends ANode {
          // A list node with an element of generic type "Data"
          private Data myData;
          private ANode next;
          ...
     }

     private class EndNode extends ANode {
          // A list node at the end of a list with no element
          ...
     }

     public int size () {
          return head.length ();
     }
}
```

Your task is to get the `size ()` method on the `MyLinkedList` type to work, by giving code for `ListNode.length ()` and `EndNode.length ()`.

Problem 4 [40 Points]

In this problem, your task is to write a recursive method that solves the famous "0-1 knapsack problem." In this problem, a thief has a knapsack (backpack) that can hold `capacity` pounds of jewels, and the goal is to compute the maximum value of jewels that can be stolen, subject to the constraint that the total weight cannot exceed `capacity`, and subject to the constraint that each jewel can either be taken or not (this is the "0-1" part).

In addition to `capacity`, your method will take as input an `ArrayList <Jewel>` object, where `Jewel` is defined as:

```
public interface Jewel {
  int getWeight ();  // return weight in pounds
  int getValue ();   // return value in dollars
}
```

Given this setup, complete the class definition below by writing a recursive method `getMaxValue` to compute the maximum value of jewels that the knapsack can hold. Note that you can write additional "helper" methods if needed. Also, do not try to be efficient—no one knows of a solution to this problem that is any faster than $O(2^n)$—so any brute-force solution is fine, as long as it works.

```
public class Knapsack {

  public int getMaxValue (ArrayList <Jewel> itemsToSteal, int capacity) {

  /* finish up this class! */
```