Example Midterm Questions

1. Here's a recursive function that operates on arrays of integers.

```
public static int mysteryFunc(int[] a, int first, int last) {
      if (first > last) return 0;
      if (first == last) return a[first];
      else {
            int mid = (first + last)/2;
            int result1 = mysteryFunc(a, first, mid);
            int result2 = mysteryFunc(a, mid+1, last);
            return result1 + result2;
      }
}
```

   A. Describe what this recursive function does. Hint: if you're a bit confused by this function, you could write out a boxtrace on the back of the paper for a small array of integers to help you figure it out.

The function adds together all the integers in an array.

   B. Given how it works, what function that you've implemented in a lab or assignment is it most similar to?

It works like *mergeSort()*, recursively breaking the array into halves until it gets to an individual number and then returns that number. The line with + is like the call to *merge()* in *mergeSort()*. Given the sum of the numbers in two sub-arrays, it adds the two sums together to get the sum of the bigger array.

2. Here's some code for `Object get(int index)` on a linked list:

```
public Object get(int index) {
      if (index < 0 || index >= size)
            throw new ListIndexOutOfBoundsException(i);
      else {
            Node current = head;
            for(int i = 0; i < index; i++) {
                  current = current.next;
            }
            return current.item;
      }
}
```

A. Write a recursive version of `Object get(int index)`. The outline of the code has been provided for you below:

```
public Object get(int index) {
      if (index < 0 || index >= size)
            throw new ListIndexOutOfBoundsException("Index error " +
      i);
      // Call getRecursive starting with index we're looking for
      // and the head of the list.
      else return getRecursive(index, head);
}

// getRecursive is the helper function that you need to implement
private Object getRecursive(int index, Node current) {
      // First have an if statement for the base case - when do
      // you know you're done and can directly return the answer?

      if (index == 0)
            return current.item;

      // Then handle the recursive case. Hint: How do you need to
      // change the arguments to getRecursive() in the recursive
call?

      else {
            index--;
            return getRecursive(index, current.next);
      }
}
```

B. In terms of computing time and memory requirements (on the call stack) which method is more efficient, the non-recursive one above or the recursive one? If you're not sure, section 3.5 of the textbook can help you figure it out.

The non-recursive version is more efficient. The recursive version creates incurs the overhead of a method call for every element in the list up to the index, as well as allocating a stack frame for every call.

3. Each of the parts of question 3 assume you have linked list and array list implementations that store items of type `Object` and implement `ListInterface`. There is something wrong with each of these code snippets. For each one, describe what is wrong and how you would fix it.

For the questions below assume you have:

```
// This is the stack from class
class IntegerStack implements IntegerStackInterace { … }

// A list implementation that stores items of type Object using
// Nodes
class MyLinkedList implements ListInterface { … }

// A list implementation that stores items of type Object using
// an array
class MyArrayList implements ListInterface { … }
```

3.1

```
IntegerStackInterface stack = new IntegerStack();
int i = 5;
stack.add(0, i);
```

add() is not defined on a stack.
Use stack.push(i) instead.

3.2

```
// This printList method is supposed to work with any kind of
// list you pass into it regardless of implementation
void printList(MyLinkedList l) { // code to print here }

MyLinkedList list1 = new MyLinkedList();
MyArrayList list2 = new MyArrayList();
printList(list1);
printList(list2);
```

The argument to printList can only be a MyLinkedList, so will get an error on printList(list2).
Change the argument type of printList to: printList(ListInterface l)

3.3.

```
ListInterface list = new ListInterface();
String s = "hello";
list.add(0, s);
```

You can't instantiate interfaces, only classes.
Change the first line to instantiate a list implementation, for example:
ListInterface list = new MyLinkedList();

3.4

```
MyLinkedList list = new MyLinkedList();
String s1 = "Hello world";
list.add(0, s);
String s2 = list.get(0);
```

Since MyLinkedList stores items of type Object, get() returns an Object. Object can't be automatically converted into a String.
Typecast the Object into a String: String s2 = (String)list.get(0);

3.5

```
ListInterface list = new MyArrayList();
list.add("green");
list.add("red");
list.add("blue");
for(int i = 0; i < list.size(); i++) {
   Node n = list.get(i);
   System.out.println(n);
}
```

Array lists don't use Nodes to store elements, they use an array. And even for linked lists that use Nodes, get() doesn't return Nodes, but rather the item stored in a Node.
Change the code inside the for loop to: System.out.println(list.get(i)) .

4. You have created your own class to store 2D data points:

```
class Point2D {
     double x;
     double y;

     Point2D(double x, double y) { this.x = x; this.y = y; }
}
```

In your program you have the following lines:

```
Point2D p = new Point2D(1.0, 1.0);
System.out.println(p);
```

But instead of printing out "(1.0, 1.0)" like you want, instead it prints out `Point2D@5c647e05`.

Write the method that you need to define on `Point2D` so that the `println` above properly prints the x,y coordinates in parentheses when it prints a `Point2D`.

```
public String toString() {
        return "(" + x + ", " + y + ")";
}
```

5. Assume you have an `IntegerList` implementation that stores lists of integers and which has the following operations defined on it: `isEmpty()`, `size()`, `get(int index)`, `add(int index, int item)`, `remove(int index)` and `removeAll()`. Write the methods described below using *only these six ADT operations*. *In other words you are writing methods belonging to a client program that is using the `IntegerList` ADT, so you do not have direct access to the internals of `IntegerList`*. For the purposes of this question you can assume that only valid indices will be passed to your methods.

   A. Write a `static void` method called `swap(IntegerList list, int i, int j)` that will interchange the items currently at positions `i` and `j` of list.

```
public static void swap(IntegerList list, int i, int j) {
        int firstInt = list.get(i);
        int secondInt = list.get(j);

        list.remove(i);
        list.add(i, secondInt);

        list.remove(j);
        list.add(j, firstInt);
}
```

   B. Write a `static int` method called `search(IntegerList list, int i)` that will perform a linear search of list for the target `i`. The `search()` method will return the list index where `i` was found, or it will return -1 if the item doesn't exist in the list.

```
public static int search(IntegerList list, int i) {
        for(int x = 0; x < list.size(); x++)
                if (i == list.get(x))
                        return x;

        return -1;
}
```

6. The Rule class for Tracery looks like:

```
class Rule {
      private static Random random;

      private String raw;

      // We're changing this to a ListInterface
      private String[] sections;
      …

      Rule(String raw) {
            this.raw = raw;
            sections = raw.split("#");
      }
}
```

Imagine that we changed the Rule class so that `sections` is a list instead of a `string[]`: `ListInterface sections;`

Write a new constructor for `Rule` that stores the sections in a list. Assume you're using a list implementation called `MyLinkedList`.

```
Rule(String raw) {
      this.raw = raw;
      sections = new MyLinkedList();
      String[] sectionStrings = raw.split("#");
      for(int i = 0; i < sectionStrings.length; i++) {
            sections.add(i, sectionStrings[i]);
      }
}
```

7. Assume you've implemented:
`getSmallestIndex(String[] words, String query, int startIndex, int endIndex)` and
`getLargestIndex(String[] words, String query, int startIndex, int endIndex)`
as you did in assignment 1. Use these two functions to write a function `printWordsBetween(String[] words, String query1, String query2)` that prints out all the words in the sorted `String[]` that are alphabetically after `query1` and alphabetically before `query2`. For example if the `String[]` was:

`String[] words = {"frankenstein", "frankenstein", "frankness", "frantic", "fraud", "free", "free"}`

and we called `printWordsBetween(words, "frankenstein", "free")`

the output should be:

```
frankness
frantic
fraud
```

If either of the query words doesn't appear in the `String[]`, don't print anything. Using the functions `getSmallestIndex()` and `getLargestIndex()`, this function can be written in only 5 lines.

```
public static void printWordsBetween(String[] sortedWords, String query1, String query2)
{
	int i1 = getLargestIndex(sortedWords, query1, 0, sortedWords.length - 1);
	int i2 = getSmallestIndex(sortedWords, query2, 0, sortedWords.length - 1);

	if (i1 != -1 && i2 != -1) {
		for(int i = i1 + 1; i < i2; i++)
			System.out.println(sortedWords[i]);
	}
}
```

8. In question 5 you implemented the the method `swap(IntegerList list, int i, int j)` using the public list interface. Now write a version of `swap(int i, int j)` as a method *on your list class*. This means that this version of `swap` can directly access the internal data structures *inside your list*. Write your function so it only has to iterate through the list once. Assume that the list is a linked list with the following structure:

```
class IntegerLinkedList implements IntegerListInterface {
	protected Node head;
	protected int size;

	protected class Node {
		int item;
		Node next;
		Node(int i) {
			this.i = i;
			next = null;
		}
	}
	… // rest of methods here
}
```

Here is an outline of the code you need to write:

```
public void swap(int index1, int index2) {
      if (index1 < 0 || index2 < 0 || index1 >= size || index2 >=
size)
            throw new ListIndexOutOfBoundsException("Index out of
bounds in swap");

      Node nodeToSwap1 = null; // the node at index1
      Node nodeToSwap2 = null; // the node at index2
      int i = 0; // an index counter for our loop
      Node currentNode = head; // the current node starts at the head

      // hint: need a while loop that keeps looping until it has
found
      // the two nodes to swap at index1 and index2

      while (nodeToSwap1 == null || nodeToSwap2 == null) {
            if (i == index1)
                  nodeToSwap1 = currentNode;
            if (i == index2)
                  nodeToSwap2 = currentNode;
            currentNode = currentNode.next;
            i++;
      }

      // Once you're here you have found the two nodes containing the
      // items to swap. Do the swap
      int temp = nodeToSwap1.item;
      nodeToSwap1.item = nodeToSwap2.item;
      nodeToSwap2.item = temp;
}
```

9. Imagine you are implementing an integer stack using an array rather than a list as the underlying data storage. In the first implementation we're assuming that the *top* of the stack is always at index 0 of the array.

```
class ArrayIntegerStack implements IntegerStackInterface {
   private int[] array;
   int size;

   public ArrayIntegerStack() {
      array = new int[50]; // This stack has a max size of 50
```

```
        size = 0;
    }

    // Buggy version of push!
    public void push(int i) {
        array[0] = i;
        size++;
    }

    // Buggy version of pop!
    public int pop(int i) {
        size--;
        return array[0];
    }
}
```

A. Describe what the problem is with `push()` and `pop()` and describe in words what you would have to do to fix the problem. Note that one of the problems with the `push()` and `pop()` above is that they don't check whether the array is full (in the case of `push()`) or empty (in the case of `pop()`). But that's not the problem we're looking for here.

Push() keeps on writing to index 0 without the items being moved down the array. So each push will overwrite the previous pushed item. Pop() keeps on readings from item 0 without items being moved up the array. So pop() keeps on returning the same value. Need to move items down the array (towards higher indices) for push() and up the array (towards lower indices for pop().

B. Write the correct code for `push()`. This involves: 1) fixing the problem you identified in part A and 2) throwing a `StackException("Stack full")` if the stack is full.

```
public void push(int i) {
        if (size == 50)
                throw new StackException("Stack full");

        // This is the same code we had for adding to an ArrayList except with the
        // index in the loop condition set to 0
        for(int pos = size - 1; pos >= 0; pos--)
                listArray[pos+1] = listArray[pos];

        array[0] = value;
        size++;
}
```

10. In this question we're adding a tail reference (pointer) to a linked list. A linked list with just a head reference has a pointer to the first node in the list, but no other pointer. Adding an element to the beginning of such a list is very efficient: you only need to change two references: the next reference in the new node you're adding (to point to the current head of the list) and the head reference (to point to the new node). But to add at the end requires chasing next pointers all the way down the list until you get to the end. A tail pointer makes adding at the end as efficient as adding at the front.

Here's a new `MyLinkedList` class with a tail pointer:

```
class MyLinkedList implements ListInterface {
   protected Node head;
   protected Node tail;
   protected int size;
   … // rest of code goes here

   // Constructor
   public MyLinkedList() {
      size = 0;
      head = null;
      tail = null;
   }
}
```

Below is code for the `add()` method. You need to add code for appropriately adjusting the `tail` reference. The two places you need to add code have been marked in comments with **`A.`** and **`B.`**

```
public void add(int index, Object o) {
      if (index >=0 && index <= size) {
            if (index == 0) {
                  // Adding to the front of the list
                  Node newNode = new Node(o);
                  newNode.next = head;
                  head = newNode;
                  // A. Need some new code here to adjust the tail
                  // Hint: You only need to adjust the tail if adding
                     to
                  // an empty list.

                  if (tail == null)
                     tail = newNode;
```

```
            }               }
            else if (index == size) {
                  // B. Adding to the end of the list

                  Node newNode = new Node(o);
                  tail.next = newNode;
                  tail = newNode;

            }
            else {
                  // Code for adding to the middle of the list goes
                  // here.
                  ...
            }
            size++;
      }
      else {
            throw new ListIndexOutOfBoundsException("Index out of
   bounds in add()");
      }
}
```

11. Imagine that you have a class called `MyQueue` that implements a queue using a linked list.

```
class MyQueue implements QueueInterface {
      protected MyLinkedList list;

      public MyQueue() {
            list = new MyLinkedList();
      }

      … // rest of the code here
}
```

In your program that is using `MyQueue`, you want to make a copy of of your queue.

```
MyQueue queue1 = new MyQueue();
queue1.enqueue("Hello");
queue1.enqueue("World");

MyQueue queue2 = queue1; // A. Making a copy of the queue

String s = queue2.dequeue();
```

However, when you run your code you find that removing an object (in this case a `String`) from `queue2` is also removing it from `queue1`.

A. What is wrong with the code above that is causing changes to `queue2` to also happen to `queue1`?

Queue2 is just a copy of the reference stored in queue1. So both variables are pointing at the same queue.

B. Write the code for the new kind of constructor you need to add to `MyQueue` to support making true copies of a `MyQueue`.

```
public MyQueue(MyQueue queue) {
        list = new MyLinkedList();
        for(int i = 0; i < queue.list.size(); i++) {
                list.add(i, queue.list.get(i));
}
```

C. Write the line of code to replace the line marked **A.** in the comment above so that it uses the new copy constructor to create a true copy of `MyQueue`.

```
MyQueue queue2 = new MyQueue(queue1);
```