Final Exam Sample (Part 2) With Solution Key*

CSS 162—Computer Programming II

Directions

This part of the exam is closed book and closed notes. No books, notes, electronic devices, or any material or information is allowed. Write neatly, use proper syntax, and clearly provide the problem number. There is no need to provide comments to code unless explicitly asked for. Partial credit may be awarded. It is recommended that you write supplementary explanations if you cannot produce a correct final answer.

Each question is worth 10 points. (Questions are those that are itemized by arabic numerals.) Please answer the question on a separate sheet(s) of paper; use the provided answer sheets, **one question per sheet of paper.** Please turn in this exam sheet with your answer sheet(s).

Questions

1. (a) Write a static method that accepts an `ArrayList<Integer>` parameter and returns a new `ArrayList<Integer>` in which each element is the cumulative sum of the original. For example, if the input to your function is [0, 1, 2, 3, −2, 4], the output `ArrayList` will be [0, 1, 3, 6, 4, 8].

   Solution:

```
public static ArrayList<Integer>
                        cumsum(ArrayList<Integer> input) {
    int sum = 0;
    ArrayList<Integer> output = new ArrayList<Integer>();
    for (int i = 0; i < input.size(); i++) {
        sum = sum + input.get(i);
        output.add(sum);
    }
    return output;
}
```

   (b) What is the running time, as expressed in terms of big-$\mathcal{O}$ notation, of your algorithm in part (a) for a list of N integers?

   Solution: $\mathcal{O}(N)$

*As of June 1, 2017. Acknowledgment: Some/all portions are copied from previous UWB CSS exams.

2. Assume that the following classes have been defined:

```
public class X<T extends Number> extends Z {
    public T value;
    public void doThis(T in) {
        value = in;
        System.out.println(value.doubleValue() * data.doubleValue());
        data = new Integer(data.intValue() + in.intValue());
    }
}

public class Y {
    public Integer data = new Integer(1);
    public void seeThis() {
        System.out.println(data.doubleValue() + 2.0);
    }
}

public class Z extends Y {
    public void printThis() {
        data = new Integer(4);
        System.out.println(data.doubleValue() * -2.0);
    }
}
```

Given the classes above, what output is produced by the following code?

```
X<Integer> a = new X<>();
Y b = new Y();
Z c = new Z();
a.doThis(new Integer(4));
b.seeThis();
c.printThis();
a.printThis();
a.seeThis();
c.seeThis();
a.doThis(new Integer(2));
a.doThis(new Integer(2));
```

Solution:

```
4.0
3.0
-8.0
-8.0
6.0
```

```
6.0
8.0
12.0
```

3. Describe how you replace arbitrary elements in each of the following abstract data types (structures). That is to say, you are told to replace a given element (e.g., "the 45th element" or "this element here") and until you were told which element it was, it could have been any one of the elements in the structure: how do you do this replacement? At the end of your replacement, the rest of the abstract data type has to be "back the way you found it."

   In your answer, note that:

   - You may use auxiliary structures, but they must be the same kind as the structures note. Thus, for instance, in your answer for stacks, you can use other stack(s) as auxiliary structures, but not other kinds of structures.
   - Your answer to the question can only make use of methods available to the structure in question. Thus, for instance, you cannot use queue methods in answering the question for stacks.

   Do not write any code; your description should be in sentences.

   (a) An array.

   Solution: Via assignment to the new element using the index of the element you are replacing.

   (b) A stack.

   Solution: Pop off the top until you get to the element you want to replace. Pop that element and discard it, then push on the new element, and then push back all the elements you had popped off to get to the replacement location.

   (c) A queue.

   Solution: Dequeue from the front until you get to the element you want to replace. Dequeue that element and discard it, enqueue the new element, and then enqueue all the elements you had dequeued to get to the replacement location. You have to enqueue, however, until you get back to the first element you dequeued. (You are dequeuing and enqueuing in a circular fashion.)

   (d) A linked list.

   Solution: Traverse until you find the node you want to replace. Add in the replacement node by linking the node prior to the replacement node to your new node and linking your new node to the node after your replacement node.

   Another acceptable answer: Traverse until you find the node whose instance variable data values (not link values) you want to replace. Once you find that node, replace those data values in the node via variable assignment.

4. In Java, implement an **insertion sort** as a static method with the signature given below that takes an array as input and returns a memory-independent copy of this array in ascending order (i.e., smallest to largest). You may assume the existence of the following methods (which you may use but don't have to use if you don't want to):

   - `minOfArray(int[] inArray)`: Returns the minimum value of an array `inArray`.
   - `minOfArrayIndex(int[] inArray)`: Returns the index of the minimum value of an array `inArray`.
   - `arrayRemoveAtIndex(int i, int[] inArray)`: Returns the array with the element given by index `i` in `inArray` removed. Thus, the array returned is one less in size than `inArray`. If `inArray` is one element in length, the return is a zero length array (which is legitimate in Java).
   - `arrayInsertAtIndex(int i, int[] inArray, int newElement)`: Returns the array with the element `newElement` inserted in `inArray` at the index `i` and with all elements originally at index `i` and greater moved to the right by one index. Thus, the array returned is one more in size than `inArray`. Inserting at an index `i` whose value is greater than or equal to the length of `inArray` results in appending to `inArray`.
   - `indexToInsertAscending(int[] inArray, int newElement)`: Returns the index `i` in `inArray` where `newElement` would go (assuming all elements with indices greater than or equal to index `i` would move to the right by one index) to keep `inArray` sorted in ascending order. The searching algorithm starts at the end of `inArray` and works backwards until the correct insertion point is found. `newElement` is assumed to be less than the last element and greater than the first element in `inArray`.

   Here is the signature of the method you're writing:

   ```
   public static int[] mySort(int[] toSort)
   ```

   You can assume the array to be sorted is at least two elements long. You don't have to test that `toSort` is a certain minimum length.

Solution: Here's on solution using the methods I provided:

```
public static int[] mySort(int[] toSort) {
    int[] sorted = new int[1];
    sorted[0] = toSort[0];

    // Insertion sort algorithm:
    for (int i = 1; i < toSort.length; i++) {
        if (sorted[sorted.length-1] < toSort[i]) {
            sorted = arrayInsertAtIndex(sorted.length, sorted, toSort[i]);
        } else {
            int idx = indexToInsertAscending(sorted, toSort[i]);
            sorted = arrayInsertAtIndex(idx, sorted, toSort[i]);
        }
    }

    return sorted;
}
```

and another solution that doesn't use those methods:

```
public static int[] mySort(int[] toSort) {
    int[] sorted = new int[toSort.length];
    for (int i = 0; i < toSort.length; i++) {
        sorted[i] = toSort[i];
    }

    // Insertion sort algorithm:
    for (int i = 1; i < sorted.length; i++) {
        int insertValue = sorted[i];
        int j = i-1;
        boolean isInserted = false;
        while (j >= 0 && !isInserted) {
            if (insertValue < sorted[j]) {
                sorted[j] = sorted[j+1];
                sorted[j+1] = insertValue;
            } else {
                isInserted = true;
            }
            j--;
        }
    }

    return sorted;
}
```