Test 1: Compsci 100

Owen Astrachan

February 15, 2006

Name: _____ (2 points)

Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	15 pts.	
Problem 2	12 pts.	
Problem 3	25 pts.	
Problem 4	21 pts.	
TOTAL:	75 pts.	

This test has 11 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper `import statements`. Assume that all libraries and packages we've discussed are imported in any code you write.

PROBLEM 1 : (*Mad Max: Beyond Blunderdome* 15 points)

Consider two algorithms for finding the string that occurs most frequently in an `ArrayList`. These are described below and code that implements the algorithms follows. You're asked four questions about these algorithms.

- Sort the array. Scan the sorted array looking for runs of equal strings. Each time a new string is found, update the current maximal-length run and the string that caused the run. This is the maximally occurring string. The method below implementing this is `sortMax`
- Create a set of the unique elements in the array. Count how many times each of the unique elements occurs in the array and keep track of the maximal value and the string that has this value. This is the maximally occurring string. The method below implementing this is `setMax`.

Both methods below correctly determine and print a maximally-occurring string.

```
public static void sortMax(ArrayList<String> list){
    Collections.sort(list);
    int count = 1;
    int max = 0;
    String smax = null;
    for (int k = 1; k < list.size(); k++) {
        if (list.get(k).equals(list.get(k-1))){
            count++;
        }
        else {
            if (count > max){
                max = count;
                smax = list.get(k-1);
            }
            count = 1;
        }
    }
    if (count > max){
        max = count;
        smax = list.get(list.size()-1);
    }
    System.out.println("max "+smax+" # occurrences = "+max);
}

public static void setMax(ArrayList<String> list){
    Set<String> set = new TreeSet<String>(list);
    int max = 0;
    String smax = null;
    for(String s : set){
        int count = Collections.frequency(list,s);
        if (count > max){
            max = count;
            smax = s;
        }
    }
    System.out.println("max "+smax+" # occurrences = "+max);
}
```

Part A (3 points)

Based on the code in `setMax`, describe the value that static method `Collections.frequency` returns.

Part B (4 points)

Why is the if-statement *after* the for-loop in `sortMax` necessary? Describe a situation such that when that if-statement is removed the incorrect maximal string will be printed.

Part C (4 points)

In general, and in the worst-case (though not always) `sortMax` runs more quickly than `setMax` (this is true empirically and theoretically). Explain why `sortMax` is, in general, faster than `setMax`.

Part D (4 points)

Describe an array/data that will make `setMax` run more quickly than `sortMax` and why you think this array/data will make `setMax` run more quickly.

PROBLEM 2 : (*Apollo O-Notation* 12 points)

For each of the methods below indicate the running time of the method, in terms of n, using O-notation. You must justify your answers for credit. In all examples assume n is positive.

Part A (3 points)

```
public int calc(int n){
    int sum = 0;
    for(int k=0; k < n; k++){
        sum++;
    }
    for(int k=0; k < n; k++){
        sum++;
    }
    return sum;
}
```

Part B (3 points)

```
public int clunk(int n){
    int sum = 0;
    for(int j=0; j < n; j++){
        for(int k=0; k < j; k++){
            sum++;
        }
    }
    return sum;
}
```

Part C (3 points)

```
public int goduke(int n){
    int sum = 0;
    for(int k=1; k <= n; k = k * 2){
        sum++;
    }
    return sum;
}
```

Part D (3 points)

```
public int stuff(int n){
    int p = 0;
    while (p*p < n){
        p++;
    }
    return p;
}
```

PROBLEM 3 : (*Listing to Starboard* 25 points)

Formally, a *one-list* is `[1]`, and an *N-list* for $N \geq 1$ is an *(N-1) list* followed by N occurrences of N. For example, a four-list is `[1,2,2,3,3,3,4,4,4,4]` which is a three-list with four 4's added to the end. There are 10 elements in a four-list. There are 15 elements in a five-list.

Part A (2 points)

Exactly how many elements are there in a six list? Exactly how many elements are there in a 100-list?

Part B (4 points)

Using O-notation, the number of elements in an *N-list* is $O(N^2)$. Removing all occurrences of N from an *N-list* yields a list with $O(N^2)$ elements. Justify these statements.

Part C (4 points)

Using O-notation, how many elements are left in an *N-list* from which every element greater than N/2 is removed. Justify your answer.

Part D (9 points)

On the next page are three methods: `makeListX`, `makeListY`, and `makeListZ`. Each method returns either an N-list or a reverse N-list. In other words, the call `makeListX(4)` returns either `[1,2,2,3,3,3,4,4,4,4]` or `[4,4,4,4,3,3,3,2,2,1]` and similarly for the other methods. Next to each method indicate whether it returns a regular N-list or a reverse N-list. You must explain your reasoning for credit.

Part E (6 points)

What is the big-Oh complexity of `makeListX` and `makeListZ` from the next page. Justify both answers. Write your response below (don't forget to write Part D on the next page.)

```
public class List
{
    public static class Node{
        int info;
        Node next;
        Node (int x, Node link){
            info = x;
            next = link;
        }
    }

    public static Node makeListZ(int n){
        if (n == 1) return new Node(1,null);
        Node front = makeListZ(n-1);
        Node list = null;
        for(int k=0; k < n; k++){
            list = new Node(n,list);
        }
        Node last = front;
        while (last.next != null){
            last = last.next;
        }
        last.next = list;
        return front;
    }

    public static Node makeListY(int n){
        if (n == 0) return null;
        Node first = new Node(n,null);
        Node last = first;
        for(int k=0; k < n-1; k++){
            last.next = new Node(n,null);
            last = last.next;
        }
        last.next = makeListY(n-1);
        return first;
    }

    public static Node makeListX(int n){
        if (n == 1){
            return new Node(1,null);
        }
        Node front = new Node(n,makeListX(n-1));
        for(int k=0; k < n-1; k++){
            front = new Node(n,front);
        }
        return front;
    }
```

PROBLEM 4 : (*ListStuff* 21 points)

The method `listToArray` below returns an array containing the elements in a linked-list; the array elements are in the same order as the elements in the linked list.

```
public class TestStuff
{
    public static class Node {
        String info;
        Node next;
        public Node(String s, Node link) {
            info = s;
            next = link;
        }
    }

    /**
     * Return an array containing elements of list
     * in same order as elements in list.
     */
    public String[] listToArray(Node list){
        int count = 0;
        Node temp = list;
        while (temp != null){
            count++;
            temp = temp.next;
        }
        String[] array = new String[count];
        count = 0;
        while (list != null){
            array[count] = list.info;
            list = list.next;
            count++;
        }
        return array;
    }
```

Part A (4 points)

The complexity of `listToArray` is $O(n)$ for an n-element list. Justify this statement.

(continued next page)

Part B (3 points)

Describe an alternative/modification of `listToArray` that avoids iterating over the linked list twice. This alternative will likely make the method execute faster, perhaps twice as fast, but the method will still be $O(n)$ for an n-element list.

Part C (8 points)

Write method `arrayToList` that returns a reference/pointer to the first Node of a linked-list containing the same elements, in the same order as those in the parameter `array`.

```
public Node arrayToList(String[] array){

}
```

Part D (6 points)

The method `readAndaddInOrder` returns a pointer to the first node of a *sorted* linked-list containing the words read from a `Scanner`. The method works correctly, it calls `addInOrder` which adds a string to an already-sorted linked-list so that the list remains sorted. Method `addInOrder` is recursive. You'll write an iterative version of this method. Complete it on the next page.

```
public class TestStuff
{
    public static class Node {
        String info;
        Node next;
        public Node(String s, Node link) {
            info = s;
            next = link;
        }
    }

    /**
      * Add a string to a sorted linked-list so the list
      * remains sorted; return the modified and still sorted list.
      */
    public Node addInOrder(Node list, String s) {

        // check to see if s should be first node of list

        if (list == null || s.compareTo(list.info) <= 0) {
            return new Node(s, list);
        }

        // not first node, add after list

        list.next = addInOrder(list.next, s);
        return list;
    }

    public Node readAndAddInOrder(Scanner s) {
        Node list = null;
        while (s.hasNext()) {
            list = addInOrder(list, s.next());
        }
        return list;
    }
```

Complete `addInOrder` below. The method should be iterative, not recursive.

```
/**
  * Add a node containing s to the sorted list so that list remains sorted.
  * @param list is the sorted linked list to which a node is added
  * @param s is the string to be added to list
  * @return a sorted list containing a new node with s
  */
public static Node addInOrder(Node list, String s){

    if (list == null) return new Node(s,null);

    // list has at least one node in it









}
```