CSE 143 Sample Midterm Exam #4

(based on Summer 2009's midterm; thanks to Alyssa Harding)

1. **`ArrayList` Mystery**. Consider the following method:

```
public static void mystery4(ArrayList<Integer> list) {
    for (int i = list.size() - 2; i > 0; i--) {
        int a = list.get(i);
        int b = list.get(i + 1);
        list.set(i, a + b);
    }
    System.out.println(list);
}
```

Write the output produced by the method when passed each of the following `ArrayList`s:

List	**Output**
(a) `[72, 20]`	______________________________
(b) `[1, 2, 3, 4, 5, 6]`	______________________________
(c) `[10, 20, 30, 40]`	______________________________

2. **`ArrayList` Programming**. Write a method `removeShorterStrings` that takes an `ArrayList` of strings as a parameter and that removes from each successive pair of values the shorter string in the pair. For example, suppose that an `ArrayList` called "list" contains the following values:

```
["four", "score", "and", "seven", "years", "ago"]
```

In the first pair of strings (`"four"` and `"score"`) the shorter string is `"four"`. In the second pair of strings (`"and"` and `"seven"`) the shorter `String` is `"and"`. In the third pair of strings (`"years"` and `"ago"`) the shorter string is `"ago"`. Therefore, the call:

```
removeShorterStrings(list);
```

should remove these shorter strings, leaving the list with the following sequence of values after the method finishes executing:

```
["score", "seven", "years"]
```

If there is a tie (both strings have the same length), your method should remove the first string in the pair. If there is an odd number of strings in the list, the final value should be kept in the list. For example, if the list contains the following values:

```
["to", "be", "or", "not", "to", "be", "hamlet"]
```

After calling `removeShorterStrings`, it should contain the following:

```
["be", "not", "be", "hamlet"]
```

You may not use any other arrays, lists, or other data structures to help you solve this problem, though you can create as many simple variables as you like. You may assume that the list passed is not `null`.

3. **Stack and Queue Programming**. Write a method called `isSorted` that takes a stack of integers and returns `true` if the stack is sorted and `false` otherwise. A stack is considered sorted when its integers are in non-decreasing order (i.e. increasing order with duplicates allowed) when read from bottom to top.

   So, a sorted stack has its smallest integer on the bottom and its largest integer on the top. A stack that contains fewer than two integers is sorted by definition. For example, suppose that a variable called `s` stores the following sequence of values:

   ```
   bottom [-12, 0, 1, 8, 8, 8] top
   ```

   then a call on `isSorted(s)` should return `true`.

   If `s` had instead contained the following values:

   ```
   bottom [-9, 10, 43, 24, 97] top
   ```

   then a call on `isSorted(s)` should return `false`, because 24 is less than 43.

   You may use one queue as auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You may not use recursion to solve this problem. For full credit your code must run in O(*n*) time where *n* is the number of elements of the original stack. Use the `Queue` interface and `Stack`/`LinkedList` classes discussed in lecture.

   You have access to the following two methods and may call them as needed to help you solve the problem:

   ```
   public static void s2q(Stack<Integer> s, Queue<Integer> q) {
       while (!s.isEmpty()) {
           q.add(s.pop());          // Transfers the entire contents
       }                            // of stack s to queue q
   }

   public static void q2s(Queue<Integer> q, Stack<Integer> s) {
       while (!q.isEmpty()) {
           s.push(q.remove());      // Transfers the entire contents
       }                            // of queue q to stack s
   }
   ```

4. **Collections Programming**. Write a method `commonCustomers` that accepts two parameters, a `Map` from customer names (strings) to their television account numbers (integers) and a `Map` from customer names (strings) to their internet account numbers (integers), and returns a `Map` from customer names (strings) to their account number (integers) for each customer that has both a television account and a internet account. Assume that each customer with both types of accounts has the same account number for both accounts. For example, if a map `tvAccounts` contains these pairs:

```
{Marty=84321, David=23435, Stef=13266, Kim=62692, Lisa=25262}
```

and a map `internetAccounts` contains these pairs:

```
{Rob=93754, David=23435, Angela=42622, Kim=62692, Jason=36212}
```

The call of `commonCustomers(tvAccounts, internetAccounts)` should return a map containing these pairs:

```
{David=23435, Kim=62692}
```

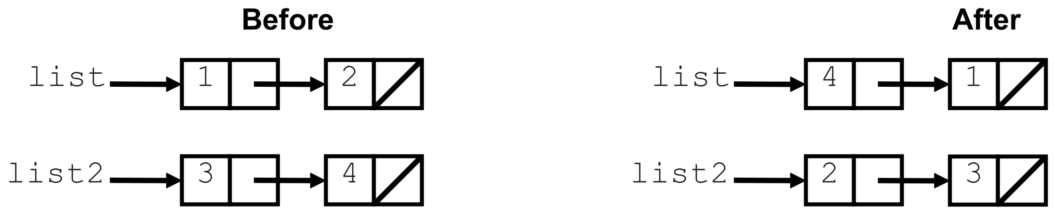
because David and Kim have both television accounts and internet accounts.

You may assume that the maps passed are not `null` and that no key or value in either is `null`. If either of the maps are empty or contain no common customers, your method should return an empty map.

You may create one collection of your choice as auxiliary storage to solve this problem. You can have as many simple variables as you like. You should not modify the contents of the maps passed to your method. For full credit your code must run in less than O(n^2) time where n is the number of pairs in the maps.

5. **Linked Nodes**. Write the code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown and/or creating new nodes as needed. There may be more than one way to write the code, but you are NOT allowed to change any existing node's `data` field value. You also should not create new `ListNode` objects unless necessary to add new values to the chain, but you may create a single `ListNode` variable to refer to any existing node if you like. If a variable does not appear in the "after" picture, it doesn't matter what value it has after the changes are made.

To help maximize partial credit in case you make mistakes, we suggest that you include optional comments with your code that describe the links you are trying to change, as shown in Section 7's solution code.



Assume that you are using the `ListNode` class as defined in lecture and section:

```
public class ListNode {
    public int data;       // data stored in this node
    public ListNode next;  // a link to the next node in the list

    public ListNode() { ... }
    public ListNode(int data) { ... }
    public ListNode(int data, ListNode next) { ... }
}
```

6. **Linked List Programming**. Write a method called `printPairsSwitched` that prints the elements of a list of integers so that the order of each pair of elements is switched. The method should separate each printed element with a single space. You are allowed to have a single extra space after the last element. Once all elements have been printed, the method should move to a new line. If the list contains an odd number of elements, the last element's order is unaffected. For example, if variables called `list1` and `list2` store the following values:

```
[1, 2, 3, 4]          // stored in list1
[5, 6, 7, 8, 9]       // stored in list2
```

then the calls:

```
list1.printPairsSwitched();
list2.printPairsSwitched();
```

should produce the following output:

```
2 1 4 3
6 5 8 7 9
```

Note that a call on the method should not change the structure of the list. That is, when the method is finished executing, the elements of the list should be in the exact same order as when it began.

Assume that we are adding this method to the `LinkedIntList` class as seen in lecture and as shown below. You may not call any other methods of the class to solve this problem and your method cannot change the contents of the list.

```
public class LinkedIntList {
    private ListNode front;

    methods
}
```

7. **Recursive Tracing**. For each call to the following method, indicate what value is returned:

```
public static int mystery(int n) {
    if (n < 0) {
        return -mystery(-n);
    } else if (n == 0) {
        return 0;
    } else {
        return mystery(n / 10) * 10 + 9 - (n % 10);
    }
}
```

Call	Value Returned
`mystery(0)`	
`mystery(5)`	
`mystery(13)`	
`mystery(297)`	
`mystery(-3456)`	

8. **Recursive Programming**. Write a recursive method called `digitsSorted` that takes an integer as a parameter and returns `true` if the digits of the integer are sorted and false otherwise. The digits must be sorted in non-decreasing order (i.e. increasing order with duplicate digits allowed) when read from left to right. An integer that consists of a single digit is sorted by definition. The method should be also able to handle negative numbers. Negative numbers are also considered sorted if their digits are in non-decreasing order. The following table shows several calls to your method and their expected return values

Call	Returns
`digitsSorted(0)`	`true`
`digitsSorted(2345)`	`true`
`digitsSorted(-2345)`	`true`
`digitsSorted(22334455)`	`true`
`digitsSorted(-5)`	`true`
`digitsSorted(4321)`	`false`
`digitsSorted(24378)`	`false`
`digitsSorted(21)`	`false`
`digitsSorted(-33331)`	`false`

You are not allowed to construct any structured objects other than strings (no array, `List`, `Scanner`, etc.) and you may not use any loops to solve this problem; you must use recursion.

Solution Key

1.

List	Output
(a) `[72, 20]`	`[72, 20]`
(b) `[1, 2, 3, 4, 5, 6]`	`[1, 20, 18, 15, 11, 6]`
(c) `[10, 20, 30, 40]`	`[10, 90, 70, 40]`

2. Two possible solutions are shown below.

```
public static void removeShorterStrings(ArrayList<String> list) {
    for (int i = 0; i < list.size() - 1; i++) {
        String first = list.get(i);
        String second = list.get(i + 1);
        if (first.length() <= second.length()) {
            list.remove(i);
        } else {
            list.remove(i + 1);
        }
    }
}

public static void removeShorterStrings(ArrayList<String> list) {
    for (int i = list.size() - 1 - (list.size() % 2); i > 0; i -= 2) {
        if (list.get(i - 1).length() <= list.get(i).length()) {
            list.remove(i - 1);
        } else {
            list.remove(i);
        }
    }
}
```

3.

```
public static boolean isSorted(Stack<Integer> s) {
    if (s.size() <= 1) {
        return true;
    }

    Queue<Integer> q = new LinkedList<Integer>();
    boolean isSorted = true;
    int prev = s.pop();
    q.add(prev);
    while (!s.isEmpty()) {
        int x = s.pop();
        if (x > prev) {
            isSorted = false;
        }
        q.add(x);
        prev = x;
    }

    while (!q.isEmpty()) {     // q2s(q, s);
        s.push(q.remove());
    }
    while (!s.isEmpty()) {     // s2q(s, q);
        q.add(s.pop());
    }
    while (!q.isEmpty()) {
        s.push(q.remove());    // q2s(q, s);
    }

    return isSorted;
}
```

4. Two possible solutions are shown below.

```
public static Map<String, Integer> commonCustomers(
        Map<String, Integer> tvAccounts,
        Map<String, Integer> internetAccounts) {
    Map<String, Integer> common = new HashMap<String, Integer>();
    for (String name : tvAccounts.keySet()) {
        if (internetAccounts.containsKey(name)) {
            common.put(name, tvAccounts.get(name));
        }
    }
    return common;
}

public static Map<String, Integer> commonCustomers(
        Map<String, Integer> m1, Map<String, Integer> m2) {
    Map<String, Integer> common = new HashMap<String, Integer>(m1);
    common.keySet().retainAll(m2.keySet());
    return common;
}
```

5.
```
list2.next.next = list;          // 4 -> 1
list.next.next = list2;          // 2 -> 3
list = list2.next;               // list -> 4
list2 = list.next.next;          // list2 -> 2
list.next.next = null;           // 1 /
list2.next.next = null;          // 3 /
```

6.
```
public void printPairsSwitched() {
    ListNode current = front;
    while (current != null && current.next != null) {
        System.out.print(current.next.data + " ");
        System.out.print(current.data + " ");
        current = current.next.next;
    }
    if (current != null) {
        System.out.print(current.data + " ");
    }
    System.out.println();
}
```

7.

Call	Value Returned
`mystery(0)`	`0`
`mystery(5)`	`4`
`mystery(13)`	`86`
`mystery(297)`	`702`
`mystery(-3456)`	`-6543`

8. Two solutions are shown below.

```
public static boolean isSorted(int x) {
    if (x < 0) {
        return isSorted(-x);
    } else if (x < 10) {
        return true;
    } else {
        int last = x % 10;
        int rest = x / 10;
        int secondToLast = rest % 10;
        return (secondToLast <= last && isSorted(rest));
    }
}

public static boolean isSorted(int x) {
    if (x < 0) {
        return isSorted(-x);
    } else {
        return x < 10 || (x / 10 % 10 <= x % 10 && isSorted(x / 10));
    }
}
```