CSE 143 Sample Midterm Exam #3

(based on Winter 2009's midterm)

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

```
public static void mystery3(ArrayList<Integer> list) {
    for (int i = list.size() - 1; i >= 0; i--) {
        if (i % 2 == 0) {
            list.add(list.get(i));
        } else {
            list.add(0, list.get(i));
        }
    }
    System.out.println(list);
}
```

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

<u>**List**</u>	<u>**Output**</u>
(a)	
`[10, 20, 30]`	____
(b)	
`[8, 2, 9, 7, 4]`	____
(c)	
`[-1, 3, 28, 17, 9, 33]`	____

2. **`ArrayList` Programming**. Write a method `removeBadPairs` that accepts an `ArrayList` of integers and removes any adjacent pair of integers in the list if the left element of the pair is larger than the right element of the pair. Every pair's left element is an even-numbered index in the list, and every pair's right element is an odd index in the list. For example, suppose a variable called `list` stores the following element values: `[3, 7, 9, 2, 5, 5, 8, 5, 6, 3, 4, 7, 3, 1]`. We can think of this list as a sequence of pairs: `[3, 7, `**`9, 2`**`, 5, 5, `**`8, 5`**`, `**`6, 3`**`, 4, 7, `**`3, 1`**`]`. The pairs 9-2, 8-5, 6-3, and 3-1 are "bad" because the left element is larger than the right one, so these pairs should be removed. So the call of `removeBadPairs(list);` would change the list to store `[3, 7, 5, 5, 4, 7]`. If the list has an odd length, the last element is not part of a pair and is also considered "bad;" it should therefore be removed by your method.

   If an empty list is passed in, the list should still be empty at the end of the call. You may assume that the list passed is not `null`. 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.

3. **Stack and Queue Programming**.

Write a method `mirrorHalves` that accepts a queue of integers as a parameter and replaces each half of that queue with itself plus a mirrored version of itself (the same elements in the opposite order). For example, suppose a variable `q` stores the following elements (each half is underlined for emphasis):

```
front [10, 50, 19, 54, 30, 67] back
```

After a call of `mirrorHalves(q);`, the queue would store the following elements (new elements in bold):

```
front [10, 50, 19, 19, 50, 10, 54, 30, 67, 67, 30, 54] back
```

If your method is passed an empty queue, the result should be an empty queue. If your method is passed a `null` queue or one whose size is not even, your method should throw an `IllegalArgumentException`.

You may use **one stack or one queue** (but not both) 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 queue. Use the `Queue` interface and `Stack`/`LinkedList` classes from 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 `rarestAge` that accepts as a parameter a map from students' names (strings) to their ages (integers), and returns the *least* frequently occurring age. Consider a map variable `m` containing the following key/value pairs:

```
{Alyssa=22, Char=25, Dan=25, Jeff=20, Kasey=20, Kim=20, Mogran=25, Ryan=25, Stef=22}
```

Three people are age 20 (Jeff, Kasey, and Kim), two people are age 22 (Alyssa and Stef), and four people are age 25 (Char, Dan, Mogran, and Ryan). So a call of `rarestAge(m)` returns `22` because only two people are that age.

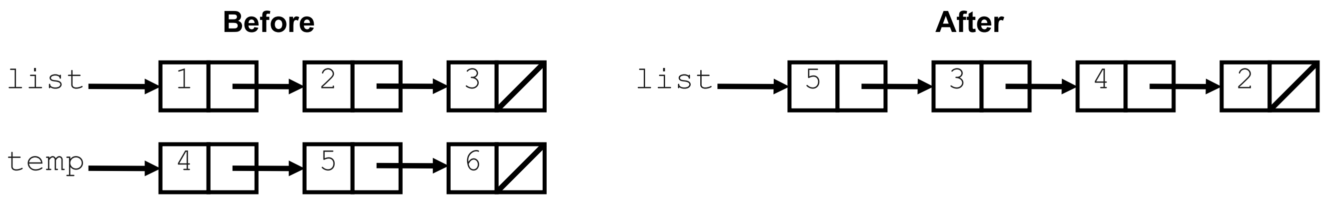
If there is a tie (two or more rarest ages that occur the same number of times), return the youngest age among them. For example, if we added another pair of `Kelly=22` to the map above, there would now be a tie of three people of age 20 (Jeff, Kasey, Kim) and three people of age 22 (Alyssa, Kelly, Stef). So a call of `rarestAge(m)` would now return `20` because 20 is the smaller of the rarest values.

If the map passed to your method is `null` or empty, your method should throw an `IllegalArgumentException`. You may assume that no key or value stored in the map is `null`. Otherwise you should not make any assumptions about the number of key/value pairs in the map or the range of possible ages that could be in the map.

You may create **one new set or map** 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 map 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 map.

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 compress that could be added to the LinkedIntList class, that accepts an integer *n* representing a "compression factor" and replaces every *n* elements with a single element whose data value is the sum of those *n* nodes. Suppose a LinkedIntList variable named list stores the following values:

```
[2, 4, 18, 1, 30, -4, 5, 58, 21, 13, 19, 27]
```

If you made the call of list.compress(2);, the list would replace every two elements with a single element (2 + 4 = **6**, 18 + 1 = **19**, 30 + (-4) = **26**, ...), storing the following elements:

```
[6, 19, 26, 63, 34, 46]
```

If you then followed this with a second call of list.compress(3);, the list would replace every three elements with a single element (6 + 19 + 26 = **51**, 63 + 34 + 46 = **143**), storing the following elements:

```
[51, 143]
```

If the list's size is not an even multiple of *n*, whatever elements are left over at the end are compressed into one node. For example, the original list on this page contains 12 elements, so if you made a call on it of list.compress(5);, the list would compress every five elements, (2 + 4 + 18 + 1 + 30 = **55**, -4 + 5 + 58 + 21 + 13 = **93**), with the last two leftover elements compressing into a final third element (19 + 27 = **46**), resulting in the following list:

```
[55, 93, 46]
```

If *n* is greater than or equal to the list size, the entire list compresses into a single element. If the list is empty, the result after the call is empty regardless of what factor *n* is passed. You may assume that the value passed for *n* is ≥ 1.

For full credit, you may not create any new ListNode objects, though you may have as many ListNode variables as you like. For full credit, your solution must also run in O(*n*) time. Assume that you are adding this method to the LinkedIntList class below. You may not call any other methods of the class.

```
public class LinkedIntList {
    private ListNode front;

    methods
}
```

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

```
public static void mystery(int n) {
    if (n < 0) {
        System.out.print("-");
        mystery(-n);
    } else if (n < 10) {
        System.out.println(n);
    } else {
        int two = n % 100;
        System.out.print(two / 10);
        System.out.print(two % 10);
        mystery(n / 100);
    }
}
```

Call	Output
`mystery(7);`	
`mystery(825);`	
`mystery(38947);`	
`mystery(612305);`	
`mystery(-12345678);`	

8. **Recursive Programming**.

Write a recursive method isReverse that accepts two strings as a parameter and returns true if the two strings contain the same sequence of characters as each other but in the opposite order (ignoring capitalization), and false otherwise. For example, the string "hello" backwards is "olleh", so a call of isReverse("hello", "olleh") would return true. Since the method is case-insensitive, you would also get a true result from a call of isReverse("Hello", "oLLEh"). The empty string, as well as any one-letter string, is considered to be its own reverse. The string could contain characters other than letters, such as numbers, spaces, or other punctuation; you should treat these like any other character. The key aspect is that the first string has the same sequence of characters as the second string, but in the opposite order, ignoring case. The table below shows more examples:

Call	Value Returned
`isReverse("CSE143", "341esc")`	`true`
`isReverse("Madam", "MaDAm")`	`true`
`isReverse("Q", "Q")`	`true`
`isReverse("", "")`	`true`
`isReverse("e via n", "N aIv E")`	`true`
`isReverse("Go! Go", "OG !OG")`	`true`
`isReverse("Obama", "McCain")`	`false`
`isReverse("banana", "nanaba")`	`false`
`isReverse("hello!!", "olleh")`	`false`
`isReverse("", "x")`	`false`
`isReverse("madam I", "i m adam")`	`false`
`isReverse("ok", "oko")`	`false`

You may assume that the strings passed are not null. 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. If you like, you may declare other methods to help you solve this problem, subject to the previous rules.

Solution Key

1.

List	**Output**
(a) `[10, 20, 30]`	`[20, 10, 20, 30, 30, 20]`
(b) `[8, 2, 9, 7, 4]`	`[8, 7, 8, 2, 9, 7, 4, 4, 2, 8]`
(c) `[-1, 3, 28, 17, 9, 33]`	`[33, 28, 33, -1, 3, 28, 17, 9, 33, 17, -1, 33]`

2. Two solutions are shown.

```
public static void removeBadPairs(ArrayList<Integer> list) {
    if (list.size() % 2 != 0) {
        list.remove(list.size() - 1);
    }

    for (int i = 0; i < list.size(); i += 2) {
        if (list.get(i) > list.get(i + 1)) {
            list.remove(i);
            list.remove(i);
            i -= 2;
        }
    }
}


public static void removeBadPairs(ArrayList<Integer> list) {
    if (list.size() % 2 != 0) {
        list.remove(list.size() - 1);
    }

    for (int i = list.size() - 1; i > 0; i--) {
        if (i % 2 != 0 && list.get(i - 1) > list.get(i)) {
            list.remove(i);
            list.remove(i - 1);
        }
    }
}
```

3. Two solutions are shown.

```
public static void mirrorHalves(Queue<Integer> q) {
   if (q == null || q.size() % 2 != 0) {
      throw new IllegalArgumentException();
   }

   Stack<Integer> s = new Stack<Integer>();
   int size = q.size();

   for (int i = 0; i < size / 2; i++) {
      int element = q.remove();
      s.push(element);
      q.add(element);
   }
   while (!s.isEmpty()) {
      q.add(s.pop());
   }

   for (int i = 0; i < size / 2; i++) {
      int element = q.remove();
      s.push(element);
      q.add(element);
   }
   while (!s.isEmpty()) {
      q.add(s.pop());
   }
}
```

```
public static void mirrorHalves(Queue<Integer> q) {
   if (q == null || q.size() % 2 != 0) {
      throw new IllegalArgumentException();
   }
   Stack<Integer> s = new Stack<Integer>();
   int size = q.size();
   for (int i = 1; i <= 2; i++) {
      while (s.size() < size / 2) {
         s.push(q.peek());
         q.add(q.remove());
      }
      s2q(s, q);
   }
}
```

4. Four solutions are shown.

```
public static int rarestAge(Map<String, Integer> m) {
   if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
   }
   Map<Integer, Integer> counts = new TreeMap<Integer, Integer>();
   for (String name : m.keySet()) {
      int age = m.get(name);
      if (counts.containsKey(age)) {
         counts.put(age, counts.get(age) + 1);
      } else {
         counts.put(age, 1);
      }
   }

   int minCount = m.size() + 1;
   int rareAge = -1;
   for (int age : counts.keySet()) {
      int count = counts.get(age);
      if (count < minCount) {
         minCount = count;
         rareAge = age;
      }
   }

   return rareAge;
}


public static int rarestAge(Map<String, Integer> m) {
   if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
   }
   Map<Integer, Integer> counts = new TreeMap<Integer, Integer>();
   for (int age: m.values()) {
      if (!counts.containsKey(age)) {
         counts.put(age, 0);
      }
      counts.put(age, counts.get(age) + 1);
   }

   int rareAge = -1;
   for (int age : counts.keySet()) {
      int count = counts.get(age);
      if (rareAge < 0 || counts.get(age) < counts.get(rareAge)) {
         rareAge = age;
      }
   }
   return rareAge;
}
```

```
public static int rarestAge(Map<String, Integer> m) {
   if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
   }
   Map<Integer, Integer> counts = new TreeMap<Integer, Integer>();
   for (String name: m.keySet()) {
      if (counts.containsKey(m.get(name))) {
         counts.put(m.get(name), counts.get(m.get(name)) + 1);
      } else {
         counts.put(m.get(name), 1);
      }
   }

   int minCount = 999999999;   // really big number to be overwritten
   for (int age : counts.keySet()) {
      minCount = Math.min(minCount, counts.get(age));
   }

   for (int age : counts.keySet()) {
      if (counts.get(age) == minCount) {
         return age;
      }
   }
   return -1;   // won't reach here
}


public static int rarestAge(Map<String, Integer> m) {
   if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
   }
   Map<Integer, Integer> counts = new HashMap<Integer, Integer>();
   for (String name: m.keySet()) {
      if (!counts.containsKey(m.get(name))) {
         counts.put(m.get(name), 0);
      }
      counts.put(m.get(name), counts.get(m.get(name)) + 1);
   }

   int minCount = 999999999;   // really big number to be overwritten
   for (int age : counts.keySet()) {
      minCount = Math.min(minCount, counts.get(age));
   }

   int rareAge = -1;
   for (int age : counts.keySet()) {
      if (counts.get(age) == minCount && (rareAge < 0 || age < rareAge)) {
         rareAge = age;
      }
   }
   return rareAge;
}
```

5. Four solutions are shown.

```
list.next.next.next = temp;         // 3 -> 4
temp.next.next = list.next.next;    // 5 -> 3
list.next.next = null;              // 2 /
ListNode temp2 = temp.next;         // temp2 -> 5
temp.next = list.next;              // 4 -> 2
list = temp2;                       // list -> 5
```

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

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

```
ListNode temp2 = list;               // temp2 -> 1
list = temp.next;                    // list -> 5
list.next = temp2.next.next;         // 5 -> 3
list.next.next = temp;               // 3 -> 4
list.next.next.next = temp2.next;    // 4 -> 2
list.next.next.next.next = null;     // 2 /
```

6. Four solutions are shown.

```
public void compress(int factor) {
   ListNode current = front;
   while (current != null) {
      int i = 1;
      ListNode current2 = current.next;
      while (current2 != null && i < factor) {
         current.data += current2.data;
         current.next = current.next.next;
         i++;
         current2 = current2.next;
      }
      current = current.next;
   }
}

public void compress(int factor) {
   ListNode current = front;
   while (current != null) {
      ListNode current2 = current.next;
      for (int i = 1; i < factor; i++) {
         if (current2 != null) {
            current.data += current2.data;
            current.next = current.next.next;
            current2 = current2.next;
         } else break;  // break is optional
         }
      current = current.next;
   }
}

public void compress(int n) {
   if (front != null) {
      ListNode current = front;
      int i = 1;
      while (current.next != null) {
         if (i == n) {
            current = current.next;
            i = 1;
         } else {
            current.data += current.next.data;
            current.next = current.next.next;
            i++;
         }
      }
   }
}

public void compress(int n) {
   ListNode current = front;
   while (current != null && current.next != null) {
      for (int i = 1; i < n; i++) {
         current.data += current.next.data;
         current.next = current.next.next;
         if (current.next == null) {
            break;
         }
      }
      current = current.next;
   }
}
```

7.

Call	Output
`mystery(7);`	`7`
`mystery(825);`	`258`
`mystery(38947);`	`47893`
`mystery(612305);`	`0523610`
`mystery(-12345678);`	`-785634120`

8. Four solutions are shown.

```
public static boolean isReverse(String s1, String s2) {
   if (s1.length() == 0 && s2.length() == 0) {
      return true;
   } else if (s1.length() == 0 || s2.length() == 0) {
      return false;  // not same length
   } else {
      String s1first = s1.substring(0, 1);
      String s2last = s2.substring(s2.length() - 1);
      return s1first.equalsIgnoreCase(s2last) &&
      isReverse(s1.substring(1), s2.substring(0, s2.length() - 1));
   }
}


public static boolean isReverse(String s1, String s2) {
   if (s1.length() != s2.length()) {
      return false;  // not same length
   } else if (s1.length() == 0 && s2.length() == 0) {
      return true;
   } else {
      s1 = s1.toLowerCase();
      s2 = s2.toLowerCase();
      return s1.charAt(0) == s2.charAt(s2.length() - 1) &&
             isReverse(s1.substring(1, s1.length()),
                       s2.substring(0, s2.length() - 1));
   }
}


public static boolean isReverse(String s1, String s2) {
   if (s1.length() == s2.length()) {
      return isReverse(s1.toLowerCase(), 0, s2.toLowerCase(), s2.length() - 1);
   } else {
      return false;   // not same length
   }
}
private static boolean isReverse(String s1, int i1, String s2, int i2) {
   if (i1 >= s1.length() && i2 < 0) {
      return true;
   } else {
      return s1.charAt(i1) == s2.charAt(i2) &&
             isReverse(s1, i1 + 1, s2, i2 - 1);
   }
}


public static boolean isReverse(String s1, String s2) {
   return reverse(s1.toLowerCase()).equals(s2.toLowerCase());
}
private static String reverse(String s) {
   if (s.length() == 0) {
      return s;
   } else {
      return reverse(s.substring(1)) + s.charAt(0);
   }
}
```