CSE 143 Sample Midterm Exam #2

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

```
public static void mystery2(ArrayList<Integer> list) {
    for (int i = list.size() - 1; i > 0; i--) {
        if (list.get(i) < list.get(i - 1)) {
            int element = list.get(i);
            list.remove(i);
            list.add(0, element);
        }
    }
    System.out.println(list);
}
```

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

List	Output
(a)	
`[2, 6, 1, 8]`	_____
(b)	
`[30, 20, 10, 60, 50, 40]`	_____
(c)	
`[-4, 16, 9, 1, 64, 25, 36, 4, 49]`	_____

2. **`ArrayList` Programming**. Write a method `isConsecutive` that accepts an `ArrayList` of integers as a parameter and returns `true` if the list contains a sequence of consecutive integers and `false` otherwise. Consecutive integers are integers that come one after the other in ascending order, as in 5, 6, 7, 8, 9, etc. For example, if a variable called `list` stores the values `[3, 4, 5, 6, 7, 8, 9]`, then the call of `list.isConsecutive()` should return `true`. If the list instead stored `[3, 4, 5, 6, 7, 12, 13]` then the call should return false because the numbers 7 and 12 are not consecutive. The list `[3, 2, 1]` might seem to be consecutive, but the elements appear in reverse order, so the method would return `false` in that case. Any list with fewer than two values should be considered to be consecutive. You may assume that the list passed is not `null`.

3. **Stack and Queue Programming**. Write a method `interleave` that accepts a queue of integers as a parameter and rearranges the elements by alternating the elements from the first half of the queue with those from the second half of the queue. For example, suppose a variable `q` stores the following sequence of values:

```
front [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] back
```

and we make the call of `interleave(q);`, the queue should store the following values after the call:

```
front [1, 6, 2, 7, 3, 8, 4, 9, 5, 10] back
```

To understand the result, consider the two halves of this list. The first half is `[1, 2, 3, 4, 5]` and the second half is `[6, 7, 8, 9, 10]`. These are combined in an alternating fashion to form a sequence of interleave pairs: the first values from each half (1 and 6), then the second values from each half (2 and 7), then the third values from each half (3 and 8), and so on. In each pair, the value from the first half appears before the value from the second half.

The previous example uses sequential integers to make the interleaving more obvious, but the same process can be applied to any sequence of even length. For example, if `q` had instead stored these values:

```
front [2, 8, -5, 19, 7, 3, 24, 42] back
```

Then the method would have rearranged the list to become:

```
front [2, 7, 8, 3, -5, 24, 19, 42] back
```

Your method should throw an `IllegalArgumentException` if the queue does not have even size. You may use one stack 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 solution must run in O(*n*) time, where *n* represents the size of the queue. 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 `union` that accepts two maps (whose keys and values are both integers) as parameters, and returns a new map that represents a merged union of the two original maps. For example, if two maps `m1` and `m2` contain these pairs:

```
{7=1, 18=5, 42=3, 76=10, 98=2, 234=50}      m1
{7=2, 11=9, 42=-12, 98=4, 234=0, 9999=3}    m2
```

The call of `union(m1, m2)` should return a map that contains the following pairs:

```
{7=3, 11=9, 18=5, 42=-9, 76=10, 98=6, 234=50, 9999=3}
```

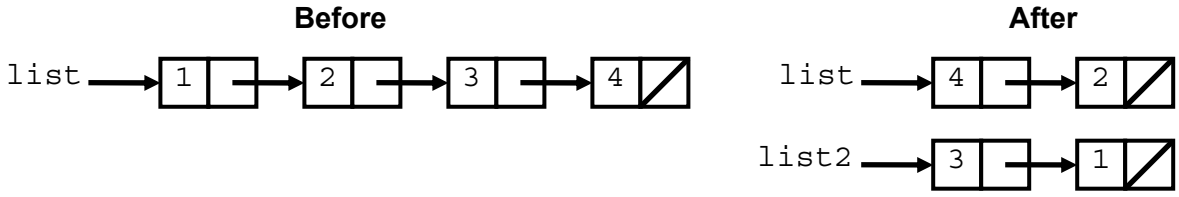
The "union" of two maps *m1* and *m2* is a new map that contains every key from *m1* and every key from *m2*. Each value stored in your "union" map should be the sum of the corresponding value(s) for that key in *m1* and *m2*, or if the key exists in only one of the two maps, that map's corresponding value should be used. For example, in the maps above, the key 98 exists in both maps, so the result contains the sum of its values from the two maps, 2 + 4 = 6. The key 9999 exists in only one of the two maps, so its sole value of 3 is stored as its value in the result map.

You may assume that the maps passed are not `null`, though either map (or both) could be empty. Though the pairs are shown in sorted order by key above, you should not assume that the maps passed to you store their keys in sorted order, and the map you return does not need to store its keys in any particular order.

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 combined number of pairs in the two maps.

5. **Linked Nodes**. Write the code that will turn the Before picture below 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 `removeLast` that could be added to the `LinkedIntList` class that removes the last occurrence (if any) of a given integer from the list of integers. For example, suppose that a variable named `list` stores this sequence of values:

```
[3, 2, 3, 3, 19, 8, 3, 43, 64, 1, 0, 3]
```

If we repeatedly make the call of `list.removeLast(3);`, then the list will take on the following sequence of values after each call:

```
after first call:  [3, 2, 3, 3, 19, 8, 3, 43, 64, 1, 0]
after second call: [3, 2, 3, 3, 19, 8, 43, 64, 1, 0]
after third call:  [3, 2, 3, 19, 8, 43, 64, 1, 0]
after fourth call: [3, 2, 19, 8, 43, 64, 1, 0]
after fifth call:  [2, 19, 8, 43, 64, 1, 0]
after sixth call:  [2, 19, 8, 43, 64, 1, 0]
```

Notice that once we reach a point where no more 3's occur in the list, calling the method has no effect.

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.

```
public class LinkedIntList {
    private ListNode front;

    methods
}
```

7. **`Comparable` Programming**. Suppose you have a pre-existing class `Rational` that represents fractions such as 1/2 or 5/8. The class has the following data and behavior:

Field/Constructor/Method	Description
`private int numerator`	value on the top of the fraction (numerator)
`private int denominator`	value on the bottom of the fraction (denominator)
`public Rational(int n, int d)`	makes a rational number to store *n*/*d*
`public int getNumerator()`	returns the fraction's top value (numerator)
`public int getDenominator()`	returns the fraction's bottom value (denominator)
`public double numericValue()`	returns the fraction's actual real number value, such as `0.5` for 1/2 or `0.375` for 3/8
`public void normalize()`	reduces numerator/denominator by greatest common factor; for example, converts 3/6 to 1/2
`public String toString()`	returns `String` such as `"5/8"`

Make `Rational` objects comparable to each other using the `Comparable` interface. Add any necessary code below, and/or make any changes to the existing code headings shown.

Rational numbers are compared by value. For example, 1/2 is a larger value than 3/8 so it is considered to be "greater" than it; and 4/17 is a smaller value than 2/3, so it is considered to be "less than" 2/3. If two fractions have the same numeric value, such as 1/2 and 4/8, break ties by considering the one with the smaller numerator and denominator to be "less." So in the previous example, 1/2 would be "less than" 4/8. If two rational numbers have exactly the same numerator and denominator, they are considered to be "equal." Your method should not modify either of the rational numbers' state. You may assume the parameter passed is not `null`.

```
public class Rational {
    ...

    // write any added code here









}
```

8. **Searching and Sorting**.

(a) Suppose we are performing a **binary search** on a sorted array called `numbers` initialized as follows:

```
// index           0   1   2   3   4   5   6   7   8   9  10  11  12  13
int[] numbers = {-30, -9, -6, -4, -2, -1,  0,  2,  4, 10, 12, 17, 22, 30};

// search for the value -5
int index = binarySearch(numbers, -5);
```

Write the indexes of the elements that would be examined by the binary search (the `mid` values in our algorithm's code) and write the value that would be returned from the search. Assume that we are using the binary search algorithm shown in lecture and section.

- Indexes examined: _____
- Value Returned: _____

(b) Write the state of the elements of the array below after each of the first 3 passes of the outermost loop of the **selection sort** algorithm.

```
int[] numbers = {100, 87, 15, 92, 45, 38, 61, 20};
selectionSort(numbers);
```

(c) Trace the complete execution of the **merge sort** algorithm when called on the array below, similarly to the example trace of merge sort shown in the lecture slides. Show the sub-arrays that are created by the algorithm and show the merging of sub-arrays into larger sorted arrays.

```
int[] numbers = {100, 87, 15, 92, 45, 38, 61, 20};
mergeSort(numbers);
```

9. **Recursive Tracing**. For each call to the following method, indicate what output is produced:

```
public static void mystery(int x, int y) {
    if (x > y) {
        System.out.print("*");
    } else if (x == y) {
        System.out.print("=" + y + "=");
    } else {
        System.out.print(y + " ");
        mystery(x + 1, y - 1);
        System.out.print(" " + x);
    }
}
```

Call	Output
`mystery(3, 3);`	
`mystery(5, 1);`	
`mystery(1, 5);`	
`mystery(2, 7);`	
`mystery(1, 8);`	

10. **Recursive Programming**. Write a recursive method `repeat` that accepts a string *s* and an integer *n* as parameters and that returns a string consisting of *n* copies of *s*. For example:

Call	Value Returned
`repeat("hello", 3)`	`"hellohellohello"`
`repeat("this is fun", 1)`	`"this is fun"`
`repeat("wow", 0)`	`""`
`repeat("hi ho!  ", 5)`	`"hi ho!  hi ho!  hi ho!  hi ho!  hi ho!  "`

You should solve this problem by concatenating strings using the `+` operator. String concatenation is an expensive operation, so it is best to minimize the number of concatenation operations you perform. For example, for the call `repeat("foo", 500)`, it would be inefficient to perform 500 different concatenation operations to obtain the result. Most of the credit will be awarded on the correctness of your solution independent of efficiency. The remaining credit will be awarded based on your ability to minimize the number of concatenation operations performed.

Your method should throw an `IllegalArgumentException` if passed a negative value for *n*. You are not allowed to construct any structured objects other than `String`s (no array, `List`, `Scanner`, etc.) and you may not use any loops to solve this problem; you must use recursion.