CSE 143 Sample Midterm Exam #1

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

```
public static void mystery1(ArrayList<Integer> list) {
    for (int i = 0; i < list.size(); i += 2) {
        int element = list.get(i);
        list.remove(i);
        list.add(element);
    }
    System.out.println(list);
}
```

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

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

2. **`ArrayList` Programming**. Write a method `stretch` that accepts an `ArrayList` of strings and an integer "stretch factor" *k* as parameters and that replaces each element with *k* copies of that element. For example, if a variable called `list` stores the elements `["hi", "how are", "you?"]`, then the call of `stretch(list, 3);` would change `list` to store `["hi", "hi", "hi", "how are", "how are", "how are", "you?", "you?", "you?"]`. If a stretch factor of 0 or less is passed, the list is made empty. If an empty list is passed in, regardless of the stretch factor, 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 `compressDuplicates` that accepts a stack of integers as a parameter and that replaces each sequence of duplicates with a pair of values: a count of the number of duplicates, followed by the actual duplicated number. For example, suppose a variable called `s` stores the following sequence of values (duplicates underlined):

   `bottom [2, 2, 2, 2, 2, -5, -5, 3, 3, 3, 3, 4, 4, 1, 0, 17, 17] top`

   and we make the call of `compressDuplicates(s);`, after the call `s` should store the following values:

   `bottom [5, 2, 2, -5, 4, 3, 2, 4, 1, 1, 1, 0, 2, 17] top`

   This new stack indicates that the original had 5 occurrences of 2 at the bottom of the stack followed by 2 occurrences of -5 followed by 4 occurrences of 3, and so on. This process works best when there are many duplicates in a row. For example, if the stack instead had stored:

   `bottom [10, 20, 10, 20, 20, 10] top`

   Then the resulting stack after the call ends up being longer than the original:

   `bottom [1, 10, 1, 20, 1, 10, 2, 20, 1, 10] top`

   If the stack is empty, your method should not change it. 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 `countInAreaCode` that accepts two parameters, a `Map` from names (strings) to phone numbers (strings) and an area code (as a string), and returns how many unique phone numbers in the map use that area code. For example, if a map `m` contains these pairs:

```
{Marty=206-685-2181, Rick=520-206-6126, Beekto=206-685-2181,
 Jenny=253-867-5309, Stuart=206-685-9138, DirecTV=800-494-4388,
 Bob=206-685-9138, Benson=206-616-1246, Hottline=900-520-2767}
```

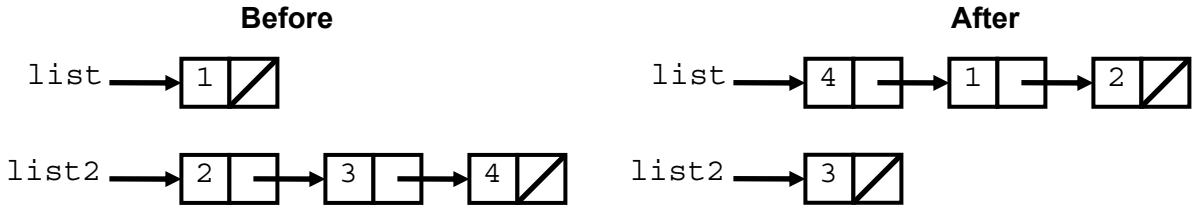
The call of `countInAreaCode(m, "206")` should return 3, because there are 3 unique phone numbers that use the 206 area code: Marty/Beekto's number of `"206-685-2181"`, Stuart/Bob's number of `"206-685-9138"`, and Benson's number of `"206-616-1246"`.

You may assume that the map passed is not `null`, that no key or value in it is `null`, that every phone number value string in the map will begin with a 3-digit numeric area code, and that the area code string passed will be a non-`null` numeric string exactly 3 characters in length. If the map is empty or contains no phone numbers with the given area code, your method should return 0.

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 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 `isSortedBy` to be added to the `LinkedIntList` class that accepts an integer *n* as a parameter and that returns `true` if the list of integers is sorted in nondecreasing order when examined "by *n*" and `false` otherwise. When examining a list "by *n*," you pick any element of the list and consider the sublist formed by that element followed by the element that comes *n* later, followed by the element that comes 2*n* later, followed by the element that comes 3*n* later, and so on. For example, suppose that a variable list stores the following sequence of numbers:

   `[1, 3, 2, 5, 8, 6, 12, 7, 20]`

   This list would normally not be considered to be sorted, which means the call of `list.isSortedBy(1)` should return `false`. But when examining elements by 2, we get two sorted sublists:

   `[1, 2, 8, 12, 20]` and `[3, 5, 6, 7]`

   The call of `list.isSortedBy(2)` should return `true`. Notice that duplicates are allowed in the sublists. The call of `list.isSortedBy(3)` returns `false` because one of the resulting sublists is not sorted:

   `[1, 5, 12]` (sorted), `[3, 8, 7]` (NOT sorted), and `[2, 6, 20]` (sorted)

   By definition, an empty list and a list of one element are considered to be sorted. The method should return true whenever *n* is greater than or equal to the length of the list, because in that case all of the resulting sublists would be of length 1. Your method should throw an `IllegalArgumentException` if passed an *n* that is less than or equal to 0.

   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. **`Comparable` Programming**. Suppose you have a pre-existing class `BankAccount` that represents users' accounts and money deposited at a given bank. The class has the following data and behavior:

Field/Constructor/Method	Description
`private String name`	name of the person using the account
`private int id`	ID number of this account
`private double balance`	amount of money currently in the account
`public BankAccount(String name, int id)`	makes account with given name/ID and $0.00
`public void deposit(double amount)`	adds given amount to account's balance
`public int getID()`	returns the account's ID number
`public String getName()`	returns the account's name
`public double getBalance()`	returns the account's balance
`public String toString()`	returns `String` such as `"Ed Smith: $12.56"`
`public void withdraw(double amount)`	subtracts given amount from account's balance

Make `BankAccount` 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.

The bank wants to sort by who has the most money, so accounts are compared by balance. One with a lower balance is considered to be "less than" one with a higher balance. If two accounts have the same balance, they are compared by ID. The one with the lower ID is considered to be "less than" the one with the higher ID. If two accounts have the same balance and ID, they are considered to be "equal." Your method should not modify any account's state. You may assume the parameter passed is not `null`.

```
public class BankAccount {
    ...

    // 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  14
int[] numbers = {-1,  3,  5,  8, 15, 18, 22, 39, 40, 42, 50, 57, 71, 73, 74};

// search for the value 42
int index = binarySearch(numbers, 42);
```

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 = {51, 41, 21, 71, 11, 81, 61, 31};
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 = {51, 41, 21, 71, 11, 81, 61, 31};
mergeSort(numbers);
```

9. **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 < 10) {
        return (n + 1) % 10;
    } else {
        return 10 * mystery(n / 10) + (n + 1) % 10;
    }
}
```

Call	Value Returned
`mystery(7)`	
`mystery(42)`	
`mystery(385)`	
`mystery(-790)`	
`mystery(89294)`	

10. **Recursive Programming**. Write a recursive method `digitMatch` that accepts two non-negative integers as parameters and that returns the number of digits that match between them. Two digits match if they are equal and have the same position relative to the end of the number (i.e., starting with the ones digit). In other words, the method should compare the last digits of each number, the second-to-last digits of each number, the third-to-last digits of each number, and so forth, counting how many pairs match. For example, for the call of `digitMatch(1072503891, 62530841)`, the method would compare as follows:

```
1 0 7 2 5 0 3 8 9 1
      | | | | | | |
      6 2 5 3 0 8 4 1
```

The method should return 4 in this case because 4 of these pairs match (2-2, 5-5, 8-8, and 1-1). Below are more examples:

Call	Value Returned
`digitMatch(38, 34)`	1
`digitMatch(5, 5552)`	0
`digitMatch(892, 892)`	3
`digitMatch(298892, 7892)`	3
`digitMatch(380, 0)`	1
`digitMatch(123456, 654321)`	0
`digitMatch(1234567, 67)`	2

Your method should throw an `IllegalArgumentException` if either of the two parameters is negative. 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.