Name _____

CSC 321: Data Structures
Fall 2015

Midterm Exam

1. True or False?	(20 pts) :	
2. Algorithm Analysis	(20 pts) :	
3. Searching & Sorting	(16 pts) :	
4. Lists & Efficiency	(24 pts) :	
5. Stacks & Queues I	(11 pts) :	
6. Stacks & Queues II	(13 pts):	
TOTAL	**(104 pts) :**	

I pledge that I have neither given nor received unauthorized aid on this test.

signed _____

1. True or False? (20 points)

_____ Every Java class that implements the `Collection` interface is required to provide an `iterator` method, which returns an *iterator* over that collection.

_____ If an algorithm is O(N), then technically it must also be $O(N^2)$.

_____ When searching for an item in a list, the *best-case performances* for both sequential search and binary search occur when the item is at the front of the list.

_____ The number of times you can repeatedly *halve* (and round down) a number N before it reaches 1 is roughly $\log_2 N$.

_____ Since *merge sort* is an O(N log N) algorithm and *selection sort* is $O(N^2)$, merge sort will sort any list in less time than selection sort.

_____ Using the Java `LinkedList` class, it is possible to add and remove elements at either end in O(1) time.

_____ In a *highly cohesive* object-oriented system, each class maps to a single, well-defined entity – encapsulating all of its internal state and external behaviors.

_____ A stack is known as a *first-in-first-out (FIFO)* data structure.

_____ In Java, a class that contains more than one private field is known as a *wrapper class*.

_____ The efficiency of merge sort can be described using the *recurrence relation*: $Cost(N) = 2Cost(N/2) + C_1N + C_2$, where N is the number of items in the list and C_1 and C_2 are a constants.

2. Algorithm Analysis (20 points)

A. Recall the formal definition of big-Oh: an algorithm is O(f(N)) if there exists a positive constant C & non-negative integer T such that for all $N \geq T$, the number of steps required to solve a problem of size N is $\leq C*f(N)$. Suppose you are analyzing an algorithm where the number of steps is defined by the cost function: $Cost(N) = 6N^3 - \frac{1}{2}N^2 + 5N - 1$. *Identify constants C and T* that show this algorithm to be $O(N^3)$, and provide a justification for your constants.

B. Identify the Big-Oh complexity of the following segments in terms of N, the number of items in `list`.

```
for (int i = 0; i < N; i += 2) {
    list[i] = 0;
}
```

```
int sum = 0;
for (int i = 1; i < N; i *= 2) {
    sum += list[i];
}
```

```
for (int i = 1; i < N; i++) {
    for (int j = N; j > 0; j--) {
        list[i] = list[i] + j;
    }
}
```

C. Consider a recursive algorithm for finding the maximum value in a (non-empty) list of numbers.

- Clearly, if the list has only one number, that number is the maximum.
- Otherwise, divide the list in half, use recursion to find the maximum value in each of the halves, then take the larger of those two to be the overall maximum.

Characterize the cost of this algorithm in the form of a recurrence relation (with N denoting the list size):

```
Cost(N) =
```

3. Searching & Sorting (16 points)

A. Assume that the following timings were taken of three algorithms as they processed lists of integers.

	time to process 2000 integers	time to process 4000 integers	time to process 8000 integers
Algorithm 1	0.1431 sec	0.5722 sec	2.2989 sec
Algorithm 2	0.8011 sec	1.4300 sec	2.4512 sec
Algorithm 3	0.0132 sec	0.0304 sec	0.0634 sec

- Which of these algorithms is most likely to be *selection sort*? What characteristics of the sort (and its performance) led you to your choice?

- Which of these algorithms is most likely to be *merge sort*? What characteristics of the sort (and its performance) led you to your choice?

B. In general, sorting a list of N comparable items requires an O(N log N) algorithm. However, we discussed two specialized sorts, frequency lists and radix sort, that can do better but only if the data has certain properties. *Describe one (1)* of these specialized sorts and the properties the data must have in order to allow for improved efficiency.

4. Lists & Efficiency (24 points)

A. We will say that two numbers are *close* if they differ by at most 0.5. Thus, 1.2 and 1.6 are close, but 1.2 and 0.6 are not.

Complete the definition of the following static method, which counts how many numbers in a list are close to a particular value. For example, if `numbers` contained [3.5, 8.0, 1.6, 4.3, 7.5, 8.3], then `numClose(4.0, numbers)` would return 2 (since 3.5 and 4.3 are both within 0.5 of 4.0).

```
/**
 * Finds the number of values in a list that are close to a given number.
 *   @param x the number to be compared with
 *   @param numList a list of numbers
 *   @return the number of values in numList that are close to x (within 0.5)
 */
public static int numClose(double x, List<Double> numList) {









}
```

B. What is the (worst-case) big-Oh complexity of your `numClose` method when called with an `ArrayList` of N items as input?

C. What is the (worst-case) big-Oh complexity of your `numClose` method when called with a `LinkedList` of N items as input?

D. Suppose we wanted to find the number in a list that has the most other numbers close to it (i.e., within 0.5). We will say this is the *most popular* number in the list. For example, 8.0 is the most popular number in the list [3.5, 8.0, 1.6, 4.6, 7.5, 8.3] since two other numbers in the list (7.5 and 8.3) are close to it, while each of the other numbers has at most one other close number.

Consider the following static method, which attempts to identify the most popular number in the list.

```
public static double mostPopular(ArrayList<Double> nums) {
    double most = nums.get(0);
    for (int i = 1; i < nums.size(); i++) {
        double next = nums.get(i);
        if (numClose(next, nums) > numClose(most, nums)) {
            most = next;
        }
    }
    return most;
}
```

Assuming your `numClose` works as described and the list size is non-empty, will this method correctly identify the most popular number? If not, provide an example where it returns an incorrect answer. If so, what is the (worst-case) Big-Oh complexity of this method (relative to list size N)? Explain your answers.

5. Stacks & Queues I (11 points)

A. Consider the following Java declarations/initializations.

```
Stack<Integer> numStack = new Stack<Integer>();
Queue<Integer> numQueue = new LinkedList<Integer>();
```

Why does the initialization of the queue specify `LinkedList` on the right-hand side (as opposed to `Queue`)? What is different about `Stack` and `Queue` in Java that requires such different initializations?

B. The following code adds numbers to `numStack` and `numQueue` then displays their contents. Note that a stack is printed as a list with the **top at the right**; a queue is printed with the **back at the right**.

```
for (int i = 1; i <= 4; i++) {
    numStack.push(i);
    numQueue.add(10+i);
}

System.out.println(numStack);  // outputs [1, 2, 3, 4]
System.out.println(numQueue);  // outputs [11, 12, 13, 14]
```

Fill in the boxes below to show the contents of the stack or queue at each print statement:

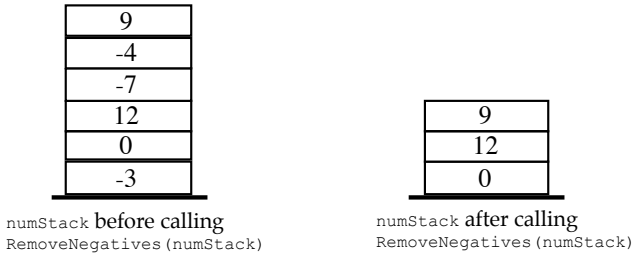
```
for (int j = 1; j <= 2; j++) {
    int num = numQueue.remove();
    numStack.push(num);
    numQueue.add(num);
}

System.out.println(numStack);
```

```
System.out.println(numQueue);
```

```
Stack<Integer> copy = new Stack<Integer>();
while (!numStack.empty()) {
    copy.push(numStack.pop());
}

System.out.println(copy);

while (!copy.empty()) {
    numStack.push(copy.pop());
}

System.out.println(numStack);
```

6. Stacks & Queues II (13 points)

A. Complete the definition of the staticmethod `removeNegatives` below, which removes all of the negative numbers from a stack of numbers. The remaining numbers should be left in the same relative order. For example:



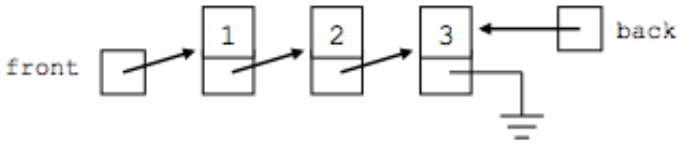
`numStack` before calling `RemoveNegatives(numStack)`

`numStack` after calling `RemoveNegatives(numStack)`

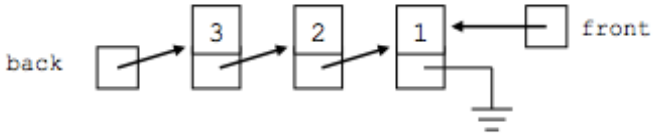
Note you may only use the standard Stack methods (push, pop, peek, empty) to manipulate the Stack. You may declare and initialize an additional Stack or Queue for temporary storage if needed.

```
/** Removes all negative numbers from a stack, leaving the remaining
 *  numbers in the same relative order.
 *    @param numStack the stack of numbers to be processed
 */
public static void removeNegatives(Stack<Integer> numStack) {
```

B. In lectures, we studied an implementation of a queue using a singly-linked list with references to the front and back. For example, adding 1, 2, and 3 (in that order) to an empty queue would yield:



Utilizing this structure, all of the queue operations (peek, add and remove) were O(1). Suppose that we had chosen to assign the `front` and `back` references to the opposite ends of the linked structure. That is:



For each of the main queue operations, does the reversal of the linked structure affect its efficiency? If not, explain why not. If so, explain why and identify the new Big-Oh complexity.

- peek (at front)

- add (at back)

- remove (from front)