Test 1: CPS 100

Owen Astrachan

October 1, 2004

Name: _____

Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	20 pts.	
Problem 2	30 pts.	
Problem 3	20 pts.	
Problem 4	20 pts.	
TOTAL:	90 pts.	

This test has 14 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper `import statements`. Assume that all libraries and packages we've discussed are imported in any code you write.

PROBLEM 1 : (***Miscellaneous Stuff (20 points)***)

Part A (4 points)

When the code fragment below executes, the output generated is `1234 != 1234`. In a sentence or two explain why.

```
String s = "1234";
Integer i = new Integer("1234");

if (s.equals(i)){
    System.out.println(s + " = "+ i);
}
else {
    System.out.println(s + " != "+ i);
}
```

Part B (8 points)

What is the value of each of the following postfix expressions. If an expression isn't valid postfix write *bad expression*, otherwise write the final value of the expression.

```
3 5 7 + * 4 6 * -
```

```
12 8 + 5 * 9 14 17 + + *
```

```
2 9 7 - 18 + * +
```

Find the exact value of the postfix expression diagrammed below

$$1\ 2\ 3\ \cdots\ n\ \underbrace{+\,+\,+\cdots+}_{\text{(n-1) times}}$$

Part C (8 points)

In this problem you'll implement a class to compare two arrays to see if one is less than the other (or equal to or greater than the other). In comparing arrays we compare corresponding elements and we only consider corresponding elements that exist in both arrays, i.e., elements with indexes `k` such that `0 <= k < Math.min(a.length,b.length)`. We consider the elements from least-index to greatest, i.e., starting with index zero. In comparing arrays, we compare corresponding elements until we can determine if $a < b$.

1. If the element of a is less than the corresponding element of b, we stop and we know $a < b$.
2. If the element of a is greater than the corresponding element of b, we stop and we know $a > b$.

If we compare all corresponding elements and they're all equal, then we make a determination by the length of the arrays. If the lengths are equal then we have $a == b$, otherwise the array with the shorter length is less than the other array.

The table below shows some results comparing a and b. We can't actually write `a.compareTo(b)` because array objects don't implement the `Comparable` interface.

a	**b**	`a.compareTo(b)`
(1,3,5)	(2,4,7)	returns value < 0
(1,3,5)	(2,4)	returns value < 0 since $1 < 2$
(1,3,5)	(2,4,5)	returns value < 0, since $1 < 3$
(2,3,5)	(1,4,7)	returns value > 0, since $2 < 1$
(1,3,8)	(1,2,7)	returns value > 0, since $3 > 2$
(1,2,3)	(1,2,3)	returns 0, arrays are equal
(1,2,3)	(1,2,3,4)	returns value < 0 since corresponding elements equal, and a has fewer elements
(1,2,3,4)	(1,2,3)	returns value > 0 since corresponding elements equal, and a has more elements

Complete the method `compare` below to implement the comparison algorithm described above. Assume the arguments to the method are arrays of `Comparable` objects.

```
import java.util.Comparator;

public class ArrayCompare implements Comparator
{
    public int compare(Object lhs, Object rhs){
        Comparable[] a = (Comparable[]) lhs;
        Comparable[] b = (Comparable[]) rhs;









    }
}
```

PROBLEM 2 : (*Uh-Oh (30 points)*)

As an example of how to think about some of the questions in this section, consider the method `stuff` below. The runtime complexity of this method is $O(n)$ and the value returned by the function is $O(n^2)$ for parameter n. As a concrete example of this, note than when $n = 100$ the loop executes 100 times doing an $O(1)$ operation each time. The value returned is $100 \times 100 = 100^2$.

```
public int stuff(int n){
    int sum = 0;
    for(int k=0; k < n; k++){
        sum += n;
    }
    return sum;
}
```

In all these problems n is a positive number. In each problem you should provide two big-Oh expressions: one for runtime and one for value returned.

Part A (6 points)

What is the runtime complexity and the value returned by method `calculate` below in terms of n? Use big-Oh and *justify your answer briefly* (in this problem the same big-Oh expression represents both runtime and value returned).

```
public int calculate(int n){

    int sum = 0;
    for(int k=0; k < n; k += 5){
        for(int j=0; j < n; j += 10){
            sum += 1;
        }
    }
    return sum;
}
```

Part B (6 points)

What is the runtime complexity and the value returned by method `evaluate` below in terms of n? Use big-Oh and *justify your answer briefly*

```
public int evaluate(int n){

    int sum = 0;
    for(int k=0; k < n; k++){
        sum += 1;
    }
    for(int k=0; k < n; k++){
        sum += 2;
    }
    for(int k=0; k < n; k++){
        sum += 3;
    }
    return sum;
}
```

Part C (6 points)

What is the runtime complexity and the value returned by method `value` below in terms of n? Use big-Oh and *justify your answer briefly*

```
public int value(int n){
    int result = 1;
    for(int k=0; k < n; k++){
        result *= 2;
    }
    return result;
}
```

Part D (6 points)

What is the runtime complexity and the value returned by method `mathize` below in terms of n? Use big-Oh and *justify your answer briefly*

```
public int mathize(int n){
    int sum = 0;
    int amount = n;
    while (n > 0){
        sum += amount;
        n = n/2;
    }
    return sum;
}
```

Part E (6 points)

Consider the method `stuff` from the beginning of this problem, reproduced below. Recall that the runtime complexity is $O(n)$ and the value returned is $O(n^2)$.

```
public int stuff(int n){
    int sum = 0;
    for(int k=0; k < n; k++){
        sum += n;
    }
    return sum;
}
```

Give big-Oh expressions for both the runtime complexity and the value returned for each of the expressions below. Justify your answer briefly.

```
int x = stuff(stuff(n));   // big-Oh for runtime and value returned


int y = stuff(stuff(n/4)); // big-Oh for runtime and value returned
```

PROBLEM 3 : (***Postscript: (20 points***)

Part A (8 points)

Postscript is a stack based language, one of the operations in postscript is *reverseN* which reverses the top N elements of a stack. For example, if the top of a stack is to the left, then `reverseN(s,4)` for the stack `s = (1, 2, 3, 4, 5, 6, 7, 8, 9)` changes the stack to `s = (4, 3, 2, 1, 5, 6, 7, 8, 9)`.

Write the function `reverseN` whose header is shown below. Assume the precondition holds, don't worry about stacks that contain fewer than N elements.

You can only use methods `push`, `pop`, `peek`, and `size` to change stack objects. However, you can use ArrayLists, arrays, and Queues, etc. in solving this problem.

```
/**
  * Reverse the order of the top n elements of a stack.
  * It is an error to call this method with a stack containing fewer than n elements.
  * @param s is the stack being reversed
  * @param n is the number of elements reversed
  */
public void reverseN(Stack s, int n){









}
```

Part B (2 points)

Give the big-Oh complexity of the method you wrote on the previous page in terms of N, the number of elements reversed, and S the number of elements on the stack. Your expression may involve only one of S and N or both. Justify your answer. If you didn't write a method, give a big-Oh expression with justification for an algorithm you'd implement if you could. All stack operations are $O(1)$.

Part C (2 points)

Give the big-Oh complexity of the method you wrote on the next page in terms of N, the number of elements reversed, and Q the number of elements on the queue. Your expression may involve only one of Q and N or both. Justify your answer. If you didn't write a method, give a big-Oh expression with justification for an algorithm you'd implement if you could. All queue operations are $O(1)$.

Part D (8 points)

Write `reverseN` for queues. The method should reverse the order of the first N elements of a queue. For example, if the front of a queue is to the left, then `reverseN(q,4)` for the queue `q = (1, 2, 3, 4, 5, 6, 7, 8, 9)` changes the queue to `q = (4, 3, 2, 1, 5, 6, 7, 8, 9)`. Write the function `reverseN` whose header is shown below. Assume the precondition holds, don't worry about queues that contain fewer than N elements.

You can only use methods `addLast`, `removefirst`, `getFirst`, and `size` to change queue objects. However, you can use ArrayLists, arrays, and Stacks, etc. in solving this problem.

```
/**
  * Reverse the order of the top n elements of a queue.
  * It is an error to call this method with a queue containing fewer than n elements.
  * @param s is the queue being reversed
  * @param n is the number of elements reversed
  */
public void reverseN(LinkedList q, int n){









}
```

PROBLEM 4 : (*Over hill, Over dale, Overlap (20 points)*)

Requests for a classroom are specified by the start-time and end-time for each class requesting to meet in the room, e.g., `1,3` means a two-hour class starting at one o'clock and `2,5` means a three-hour class starting at two o'clock (ending at five o'clock). In this problem all times are after noon and before midnight. Class requests are considered *intervals* with a start time and an end time.

Administrators want to keep rooms occupied and busy, but busy is the first priority. This means we want to schedule as many classes/requests as possible in the room. Consider the requests below.

```
(2,5), (1,3), (1,2), (1,4), (3,5), (4,5), (6,7), (3,4)
```

We could schedule the classes as follows. In both scenarios three classes are scheduled and the room is idle for one hour.

```
(1,2)  (2,5) (6,7)
```

Or we could schedule these classes:

```
(1,3)  (3,5) (6,7)
```

However, if we schedule the following classes we get four classes using the room although there are two idle hours.

```
(1,2) (3,4) (4,5) (6,7)
```

It's not possible to schedule more than four classes given the requests shown (though other schedules with four classes are possible too.)

In this problem you'll write code to produce the list of class requests that keeps the room most busy, i.e., that schedules the most classes in the room. To do this you'll implement the following algorithm in Java.

1. Sort the intervals by end time, breaking ties by the length of the interval (longer intervals are larger).
2. Schedule the first (least) interval by adding it to an ArrayList.
3. Now consider each of the remaining intervals in sorted order, call the interval being considered the *current* interval.
   (a) If the current interval overlaps with the last interval added to the ArrayList, skip it.
   (b) Otherwise, if the current interval doesn't overlap, add the current interval to the ArrayList.

The definition for the class `Interval` you'll use in this problem is provided on the next page. You'll implement several methods using this class.

```
public class Interval implements Comparable
{
    private int myStart;
    private int myEnd;

    public Interval(int start,int end){
        myStart = start;
        myEnd = end;
    }

    /**
     * Returns value < 0 if this interval less than o,
     * returns value > 0 if this interval greater than o,
     * and returns 0 if this interval is equal to o.
     */
    public int compareTo(Object o){
        Interval other = (Interval) o;
        int diff = myEnd - other.myEnd;
        // complete this method in Part B
    }

    /**
     * Return time from the end of this interval to the beginning
     * of another. If this interval ends after the other one starts, a
     * negative number will be returned, otherwise a non-negative number
     * is returned indicating the time between non-overlapping intervals.
     * @param i is the other interval
     * @return the time until interval i
     */
    public int timeUntil(Interval i){
        return i.myStart - myEnd;
    }

    /**
     * Returns true if and only if this interval overlaps
     * the interval passed to the method.
     * @param i is the interval considered for overlap with this one
     * @return true if and only if the intervals overlap
     */

    public boolean overLaps(Interval i){
        if (i.myEnd <= myStart || myEnd <= i.myStart){
            return false;
        }
        return true;
    }

    public String toString(){
        return "["+myStart+","+myEnd+"]";
    }
}
```

Part A (6 points)

Write the method `idleTime` which returns the total time between all intervals in a sorted, non-overlapping array of intervals. For example, consider these arrays:

```
{  (1,3)  (3,5) (6,7)  }
{  (1,2) (3,4) (4,5) (6,7) }
```

The first array has one hour of idle time between (3,5) and (6,7). The second array has two hours of idle time.

```
/**
  * Assuming list is sorted, returns the number of hours
  * not in any interval in list, only hours between
  * the start of list[0] and the end of list[list.length-1]
  * are considered.
  * @param list is the sorted list of non-overlapping intervals
  * @return the number of idle time, time not in any interval
  */
public int idleTime(Interval[] list){









}
```

Part B (10 points)

Write method `mostBusy` which returns an array of intervals which contains the maximal number of non-overlapping intervals from the parameter `list`. Assume the `Interval` class `compareTo` method is implemented correctly. Use the algorithm at the beginning of this problem to create a maximal array of non-overlapping intervals. The ArrayList that is part of the algorithmic description is initialized in the starter-code provided, and the ArrayList is converted to an array in the return statement. Add code as necessary to implement the algorithm.

```
/**
 * Returns array containing maximal number of non-overlapping
 * intervals from list
 * @param list is an array of intervals
 * @return array containing maximal number of non-overlapping intervals
 */
public Interval[] mostBusy(Interval[] list){
    ArrayList maximal = new ArrayList();
    Arrays.sort(list);









    return (Interval[]) maximal.toArray(new Interval[0]);
}
```

Part C (4 points)

Implement the `compareTo` method in the class `Interval` so that intervals are compared as specified earlier in this problem: an Interval a is less than an interval b if and only if a ends before b ends or they end at the same time and the length of interval a is less than the length of interval b.

```
public class Interval
{
      private int myStart;
      private int myEnd;

      /**
       * Returns value < 0 if this interval less than o,
       * returns value > 0 if this interval greater than o,
       * and returns 0 if this interval is equal to o.
       */
      public int compareTo(Object o){
          Interval other = (Interval) o;
          int diff = myEnd - other.myEnd;









      }
```