DUKE UNIVERSITY
Department of Computer Science

Test 1 Review: CompSci 201

During the test, you may consult your four (4) sheets of notes and no other resources. You may not use any computers, calculators, cell phones, or other human beings. Your answers may refer to any program text supplied in lectures or assignments.

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: (Short Ones)

A. Give two distinct advantages of using an ArrayList (e.g. ArrayList<String> list) versus an array (e.g. String[] a).

1. an ArrayList can grow and shrink as needed
2. an ArrayList has lots of useful methods (e.g., contains, indexOf)

B. What is printed as a result of the following code excerpt?

```java
int j = 12;
int i = j;
i = i - 4;

System.out.println(i);
System.out.println(j);
```

8
12

C. What is printed as a result of the following code excerpt?

```java
ArrayList<String> list1 = new ArrayList<String>();
ArrayList<String> list2 = list1;
list1.add("Baldwin");
list1.add("pass");

list2.add("luck");

System.out.println(list1);
System.out.println(list2);
```

[Baldwin, pass, luck]
[Baldwin, pass, luck]
D. Consider the following program.

```java
public class Pattern {
    public static void print(int n) {
        for (int i = -n; i <= n; i++) {
            for (int j = -n; j <= n; j++) {
                if (i == j)
                    System.out.print("A ");
                else if (i == -j)
                    System.out.print("B ");
                else
                    System.out.print(" . ");
            }
            System.out.println();
        }
    }

    public static void main(String [] args) {
        // Call print method with 1 as the argument
        print(1); // PART I
    }
}
```

I. Add a line to the `main` method above to call the `print` method with 1 as the argument.

II. What is the result of calling `print` above?
- A . B
- . A .
- B . A

III. What is the result of calling `print` with 2 as the argument?
- A  .  B
- . A  . B
- .  A  .
- . B  . A
- B  .  A

E. For each of the following object-oriented programming terms, summarize the distinction between the two terms. Your answer should be brief.

I. Comparing objects via `==` vs. `.equals`

   The `==` checks whether the references to objects are equal (i.e., are they labeling the same box). The `.equals` method checks whether the contents of an object are the same (do the boxes have the same contents).

II. Classes that implement `Comparable` vs. `Comparator`

   An object that is `Comparable` has a natural ordering defined by a `compareTo` method. A `Comparator` is an object itself whose purpose is to compare two other objects.

III. `class` vs. `object`

   A `class` represents an object in an abstract way, specifying the common attributes of all objects of a specific kind. An `object` is a specific instantiation of a class whose attributes have specific values.

IV. The constructor vs. any other class method

   A constructor is a method that is called only when an object is constructed. It should initialize all instance variables to a reasonable state so that any method can be called safely.
F. (4) Suppose that \( b[] \) is an array of 100 elements, with all entries initialized to 0, and \( a[] \) is an array of \( N \) elements, each of which is an integer between 0 and 99. What is the effect of the following loop? (select all that apply)

\[
\text{for (int } j = 0; j < N; j++)
\]

\[b[a[j]]++;\]

*Sets \( b[0] \) to the number of 0s in \( a[] \), \( b[1] \) to the number of 1s in \( a[] \), etc.*

G. What is printed as a result of the following code excerpts?

I. ```java
ArrayList<String> names = new ArrayList<String>();
TreeSet<String> namesSet = new TreeSet<String>();
names.add("Mary");
names.add("Jack");
names.add("Mary");
names.add("Jack");
namesSet.addAll(names);
System.out.println(names);
System.out.println(namesSet);
```

[Mary, Jack, Mary, Jack]

[Jack, Mary]

II. ```java
Map<Integer,String> mappings = new TreeMap<Integer,String>();
mappings.put(7, "D");
mappings.put(3, "C");
mappings.put(6, "B");
mappings.put(6, "A");
for (Integer num: mappings.keySet())
{
    System.out.println(num + " " + mappings.get(num));
}
```

3 C
6 A
7 D

H. Give a case where you would use a `HashSet` over a `TreeSet` and vice versa.

*A HashSet has faster \( O(1) \) vs. \( O(\log n) \) average case add and contains methods. A TreeSet keeps the elements in sorted order.*

I. In the Markov assignment, the `WordNgram` class includes a `equals` method similar to the one below.

```java
public class WordNgram {
    private String[] myWords; // other methods omitted
    public boolean equals( Object o ) {
        WordNgram other = (WordNgram) o;
        for( int k=0; k < myWords.length; k++ ) {
            if ( !myWords[k].equals( other.myWords[k] ) )
                return false;
        }
        return true;
    }
}
```
Why is the parameter, \(o\), of type \texttt{Object} and why is it cast to a \texttt{WordNGram}?

\textit{o is of type Object because the equals method is defined in and inherited from the Object class. In order to access the instance variables such as myWords, o needs to be case to a WordNGram object.}

\textbf{PROBLEM 2 : (Maximum impact: 8 points)}

\textbf{A.} Complete the function \texttt{allDifferent} that returns true only if all three of the given values have different values. For example, a call to \texttt{allDifferent(3,128,255)} should return \texttt{true}; while the values \texttt{allDifferent(128, 256, 128)} should return \texttt{false}. Write \texttt{allDifferent} below - including the method header.

\begin{verbatim}
boolean allDifferent(int a, int b, int c) {
    return (a!=b) && (b!=c) && (a !=c);
}
\end{verbatim}

\textbf{B.} Complete the function \texttt{max3} that takes three numbers and returns the one with the largest value. Your implementation may not use any conditionals or loops and \textbf{must} call the built-in Java library method \texttt{Math.max} described below at least once and use its result in determining the result of this function.

\begin{verbatim}
public class Math {
    /**
     * Returns the greater of two int values.
     */
    public static int max(int a, int b) {
        // Implementation omitted
    }
    // Rest of class omitted

    public int max3(int a, int b, int c) {
        return max(a, max(b,c));
    }
}
\end{verbatim}

\textbf{PROBLEM 3 : (Digits! (8 points))}

Write a method \texttt{oddDigits} below that returns whether a given integer has any digits that have odd values. Examples are below.

- \texttt{oddDigits(2468)} \(\rightarrow\) \texttt{false} because 2, 4, 6, & 8 are even
- \texttt{oddDigits(111)} \(\rightarrow\) \texttt{true} because 1 is odd
- \texttt{oddDigits(4096)} \(\rightarrow\) \texttt{true} because 9 is odd

\begin{verbatim}
boolean oddDigits(int n) {
    while (n != 0) {
        int lastDigit = n%10;
        if (lastDigit % 2 == 1)
            return true;
        n = n/10;
    }
    return false;
}
\end{verbatim}
PROBLEM 4:  \textit{(Close to Home (10 points))}

In this problem, you will write a method \texttt{closestToOrigin} that given arrays of x-coordinates and y-coordinates corresponding to points, returns the point that is the minimum distance away from the origin (0,0).

For example, given the arrays

\begin{verbatim}
closestToOrigin({1, 0, 6, 3, 5}, {10, -12, 4, 4, 5})
\end{verbatim}

should return the Point (3,4). The class Point is defined as follows:

\begin{verbatim}
public class Point {
    public int x;
    public int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public double distanceFrom(Point p) {
        return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
    }
}
\end{verbatim}

Complete \texttt{closestToOrigin} below.

\begin{verbatim}
public static Point closestToOrigin (int[] xVals, int[] yVals) {
    double min = Double.MAX_VALUE;
    Point origin = new Point(0, 0);
    Point closest = null;
    for (int k = 0; k < xVals.length && k < yVals.length; k += 1) {
        Point curr = new Point(xVals[k], yVals[k]);
        if (origin.distanceFrom(curr) < min) {
            closest = curr;
            min = origin.distanceFrom(curr);
        }
    }
    return closest;
}
\end{verbatim}

PROBLEM 5:  \textit{(Comparisons)}

The ABC organization has many chapters in every state of the United States. In order to keep track of all the member information an ABCChapter class has been created. An ABCChapter has three kinds of information, an array of all its member names, the state and the region of the state. We would like to be able to compare ABCChapters and thus to implement Comparable. Some of its methods are shown below and on the next page.

Assume that each member is a string with ?Lastname Firstname? and that the members in each chapter are stored in \texttt{myMembers} in alphabetical order. For example one ABC Chapter with four members might be:
State = "NC"
Region = "Eastern"
MemberList = "Astrachan Owen", "Duvall Robert" "Forbes Jeff" "Rodger Susan"

Below is the class.

```java
public class ABCChapter implements Comparable<ABCChapter>{
    private String[] myMembers;
    private String myState;
    private String myRegion;

    public ABCChapter(String[] list, String state, String region) {
        myMembers = new String[list.length];
        System.arraycopy(list, 0, myMembers, 0, list.length);
        myState = state;
        myRegion = region;
    }

    /**
     * Return value that meets criteria of compareTo conventions.
     * @param chap is the ABCChapter to which this is compared
     * @return appropriate value less than zero, zero, or greater than zero
     */
    public int compareTo(ABCChapter chap) { // TODO implement this method
    }

    /**
     * Return true if this ABCChapter is the same as the parameter
     * @param o is the ABCChapter to which this one is compared
     * @return true if o is equal to this ABCChapter
     */
    public boolean equals(Object o) {
        ABCChapter chap = (ABCChapter) o;
        // TODO return correct value
    }

    /**
     * Return a good value for this ABCChapter to be used in hashing.
     */
    public int hashCode() { // TODO return a hash value
    }

    A. Implement the equals method for the ABCChapter class. Two ABCChapters are equal if they have
    the same state, the same region, and all the members are the same. Assume the members are stored
    in alphabetical order in myMembers.

    public boolean equals( ) {
        ABCChapter chap = (ABCChapter) o;
        if (chap == null) return false;
        if (myState.equals(chap.myState) && myRegion.equals(chap.myRegion) &&
            myMembers.length == chap.myMembers.length) {
            for (int k = 0; k < myMembers.length; k++)
                if (!myMembers[k].equals(chap.myMembers[k]))
                    return false;
            return true;
        }
        return false;
    }
```
B. Implement the `compareTo` method for the `ABCChapter` class. An `ABCChapter` is less than another `ABCChapter` if they have fewer members. If both chapters have the same number of members, then a chapter is less than another chapter if its state comes before it in alphabetical order. If the states are the same then a chapter is less than another chapter if its region comes before the other region in alphabetical order. If both the regions and the state are the same, then one chapter is less than another chapter if the list of member names comes before the list of the other chapter member names in alphabetical order.

```java
public int compareTo(ABCChapter chap) {
    if (myMembers.length != chap.myMembers.length)
        return myMembers.length - chap.myMembers.length;
    if (!myState.equals(chap.myState))
        return myState.compareTo(chap.myState);  
    if (!myRegion.equals(chap.myRegion))
        return myRegion.compareTo(chap.myRegion);
    for (int k = 0; k < myMembers.length; k++)
        if (!myMembers[k].equals(chap.myMembers[k]))
            return myMembers[k].compareTo(chap.myMembers[k]);
    return 0;
}
```

C. Implement the `hashCode` method for the `ABCChapter` class and explain why you think it is a good `hashCode` function.

```java
public int hashCode(){
    int h = 0;
    h += myState.hashCode() + myRegion.hashCode()*3;
    for (int k=0, fact=9; k < myMembers.length; k++, fact *= 3) {
        h += myMembers[k].hashCode()*fact;
    return h;
}
```

*This hashcode encapsulates all of the instance variables of an `ABCChapter` and by raising each successive factor to another power of 3, we ensure that order matters. For example, a state named Georgia and a member named Geogia will make different contributions to the hashCode.*

PROBLEM 6 :  (*Credit Crunch*)

Credit card numbers can be validated using what’s called a *checksum algorithm*. The algorithm basically works as follows, where the left-most digit has index 0 (as it does for strings) and the algorithm works from left to right examining each digit of the credit card.

Accumulate a sum based on adding a value obtained from each digit of the credit card as follows, where each $k^{th}$ digit is examined starting with $k = 0$.

- If $k$ is even, add the $k^{th}$ digit to the sum.
- If $k$ is odd, multiply the $k^{th}$ digit by two. If the result is $\geq 10$, subtract 9. Add this computed value to the sum.

If the resulting sum is divisible by 10, the credit card number passes the checksum test, otherwise the card number isn’t valid. Credit card numbers are often 16-digits long, but they can be of any length.

Here’s an example for the card number 2543-2109-8765-4321 showing that it is a valid card number.
Since this is valid, the call `isValid("2543210987654321")` evaluates to true. However, the call `isValid("1234")` evaluates to false since the computed sum is 1 + 4 + 3 + 8 = 16 which isn’t divisible by 10.

Write the boolean-valued function `isValid` whose header is given on the next page.

In writing `isValid` you might want to use the function `Integer.parseInt` that converts a String to its int equivalent, e.g., `Integer.parseInt("9")` evaluates to 9 and `Integer.parseInt("17")` evaluates to 17.

```java
/**
 * Returns true if credit-card number’s checksum is valid
 * @param ccard contains only characters '0', '1', ... '8', '9'
 * @return true if ccard’s checksum is valid, otherwise returns false
 */
public boolean isValid(String ccard)
{
    int sum = 0;
    for (int k=0; k < ccard.length(); k+=1)
    {
        int digit = Integer.parseInt(ccard.substring(k,k+1));
        if (k % 2 == 0) { // even index
            sum += digit;
        } else { // odd index
            sum += digit*2;
            if (digit * 2 >= 10)
                sum -= 9;
        }
    }
    return sum % 10 == 0;
}
```

PROBLEM 7 : (Like MemberCheck)
A health club chain allows its members to visit any of its many health club locations an unlimited number of times per day. The only constraining rule is, a customer can only visit one health club location per day, even though he or she may return to that location an unlimited number of times for the rest of that day.

Although the honor system has always worked quite well, the club wants to run some tests to see how many people really follow the rules. You are to write a method that takes the entrance log files from all of the different clubs (all logging the same day) and return a sorted list of the people who are not honest and went to more than one health club location in the same day.

The entrances for each club are listed on a line. For example in the following file, there are three clubs.

<table>
<thead>
<tr>
<th>JOHN</th>
<th>JOHN</th>
<th>FRED</th>
<th>PEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEG</td>
<td>GEORGE</td>
<td>GEORGE</td>
<td>DAVID</td>
</tr>
</tbody>
</table>

In Club 0, John visited twice, while Fred and Peg visited once. Peg and George each visited Club 1 once. George also visited Club 2 twice, and David visited Club 2 once.

A. Complete the method, `attendeeList` below so that it returns a `List` of `Sets` corresponding to the members who visited each club.
For the above file, the three element list below should be returned.

{[FRED, JOHN, PEG], [GEORGE, PEG], [DAVID, GEORGE]}

List<Set<String>> attendeeList(Scanner in) {
    ArrayList<Set<String>> result = new ArrayList<Set<String>>();
    while (in.hasNext()) {
        TreeSet<String> words = new TreeSet<String>();
        Scanner line = new Scanner(in.nextLine());
        while (line.hasNext())
            words.add(line.next());
        result.add(words);
    }
    return result;
}

B. Write a method, whosDishonest that takes the entrance log files data structure from attendeeList the previous part and return a sorted list of the people who are not honest and went to more than one health club location in the same day.

For the file on the previous page, the two element list {GEORGE, PEG} should be returned.

List<String> whosDishonest(List<Set<String>> attendees) {
    TreeSet<String> dishonest = new TreeSet<String>();
    for (int i = 0; i < attendees.size(); i++)
        for (int j = i+1; j < attendees.size(); j++) {
            TreeSet<String> common = new TreeSet<String>(attendees.get(i));
            common.retainAll(attendees.get(j));
            dishonest.addAll(common);
        }
    return new ArrayList<String>(dishonest);
}

C. Briefly describe the big-Oh of the following call to your methods in terms of n, the number of clubs, and m, the number of people who visit each club per day. Briefly justify your answer

Scanner in = getScanner(); // open log file
List<String> list = whosDishonest(attendeeList(in)); // Give big-Oh for this line

attendeeList is $O(nm \log m)$ and whosDishonest is $O(n^2m \log m)$ so overall is $O(n^2m \log m)$

PROBLEM 8 : (Sets)

You are given the data from Dook University of the clubs and members of each club in the following format. The data for each club is on a line. Each line has the name of all the members in the club, with members separated by a colon.

The sample data below shows the members of three clubs.

Jeff Forbes:Hillary Rodham Clinton:Mary Lou Retton
Susan Rodger:Oprah Winfrey:Cay Horstmann:Mary Lou Retton:Owen Astrachan
Owen Astrachan:Oprah Winfrey:Mary Lou Retton
A. Complete the Clubs method `allMembers` that given a `Scanner` as an argument that is initialized to read from a valid data file, returns an `ArrayList` of all the people from Dook University in a club. For example, using the datafile given above, the `ArrayList` would contain: Jeff Forbes, Hillary Rodham Clinton, Mary Lou Retton, Susan Rodger, Oprah Winfrey, Cay Horstmann, and Owen Astrachan. You may write and use helper functions if you find it helpful.

```java
public class Clubs {
    public ArrayList<String> allMembers(Scanner in) {
        HashSet<String> members = new HashSet<String>();
        while (in.hasNext()) {
            String line = in.nextLine();
            String names = line.split(":");
            for (String n: names)
                members.add(n);
        }
        ArrayList<String> list = new ArrayList<String();
        for (String n: members)
            list.add(n);
        return list;
    }
}
```

B. Describe how you would change `allMembers` to return an `ArrayList` of all the people who are in more than one club.
In the previous example, you would return the list containing: Mary Lou Retton, Owen Astrachan, and Oprah Winfrey.
You should be specific as possible in describing your changes. For example, you could specify exactly the lines of code that would change and what the new lines would be.

*I would add a check to see if the new name I was adding was already in the set Members. If it was already in the set (members.contains(n)), then I would add it to a new set multiMembers.*

PROBLEM 9: (Map)

```java
public class FreqCompare implements Comparator<String> {
    private HashMap<String, Integer> myMap;

    public FreqCompare(String[] a) {
        // To be filled in..
    }

    public int compare(String a, String b) {
        int afreq = myMap.get(a);
        int bfreq = myMap.get(b);
        int diff = bfreq - afreq;
        if (diff != 0)
            return diff;
        return a.compareTo(b);
    }
}
```
A. Complete the constructor for FreqCompare below, so that the map is correctly filled in. That is, myMap should map from entries in the original array to the number of times they occur.

```java
public FreqCompare(String[] a) {
    myMap = new HashMap<String, Integer>();
    for (String s: a) {
        if (myMap.containsKey(s))
            myMap.put(s, myMap.get(s) + 1);
        else
            myMap.put(s, 1);
    }
}
```

B. Describe how the ordering done by the Comparator will change if the line

```
int diff = bfreq-afreq;
```

is changed to the following. Be brief and precise, not thorough.

```
int diff = afreq-bfreq;
```

That is, given the following code:

```
String a[] = // OMITTED
Arrays.sort(a, new FreqCompare(a));
```

how would the contents of a change.

*a would be sorted from least frequently occurring to most frequently occurring.*

PROBLEM 10: *(How many fit in the pool?)*

Write a method uniqueNames that has one Scanner parameter. This method reads names from a file and returns an array of the unique last names in alphabetical order.

The data is a list of names, one name per line. The last name is the last word on the line. There is exactly one space between words on a line. Here is a sample data file:

John Kevin Smith
Greg Smith
Fred El Mira Gumption
Chantal LaFruit
Sarah Ann Gumption

For this data file bound to the Scanner input, the call `uniqueNames(input)` would return the array 
{"Gumption", "LaFruit", "Smith"}.

Write the method header and body for uniqueNames below.

```java
public class Pool {
    public String[] uniqueNames(Scanner in) {
        TreeSet<String> names = new TreeSet<String>();
        while (in.hasNext()) {
            String[] lineElem = in.nextLine().split(" ");
            names.add(lineElem[lineElem.length-1]);
        }
        return names.toArray(new String[names.size()]);
    }
}
```
return names.toArray(new String[0]);

More straightforward conversion from Set to array:
String[] result = new String[names.size()];
int index = 0;
for (String elem: names)
  result[index++] = elem;
return result;

*/

PROBLEM 11 : *(The biggest course)*

Complete the method `emailsLargestCourse` that has one `ArrayList` parameter named `all`. Each string in `all` has three pieces of information separated by colons: the course name, the name of a student and the student’s email. This method returns an `ArrayList` of the emails of all students registering for the course with the largest enrollment. Assume there is just one course with the largest enrollment.

An example of the `Strings` in an `ArrayList` called `dukeSpring2011` is below.

```
CompSci 100: Fred Jack Smith: fjs@duke.edu
History 117: Fred Jack Smith: fjs@duke.edu
CompSci 102: Arielle Marie Johnson: amj@duke.edu
CompSci 100: Arielle Marie Johnson: amj@duke.edu
CompSci 006: Bertha White: bw@duke.edu
Econ 051: Bertha White: bw@duke.edu
English 112: Harry Potter: hp@duke.edu
CompSci 100: Harry Potter: hp@duke.edu
```

The call `emailsLargestCourse(dukeSpring2011)` would return the emails of "CompSci 100" (the largest course enrollment) in sorted order: ["amj@duke.edu", "fjs@duke.edu", "hp@duke.edu"]

```java
public ArrayList<String> emailsLargestCourse(ArrayList<String> all)
{
  ArrayList<String> result = new ArrayList<String>();
  Map<String,ArrayList<String>> mp = new HashMap<String, ArrayList<String>>();

  // build map and keep track of max size and key for max
  int max = 0;
  String maxKey = "";
  for (String str: all)
  {
    String[] items = str.split(":");
    String course = items[0];
    String email = items[2];
    ArrayList<String> emails = mp.get(course);
    if (emails == null)
    {
      emails = new ArrayList<String>();
      mp.put(course,emails);
    }
  }
  return result;
}
```
emails.add(email);
if (emails.size() > max)
{
    max = emails.size();
    maxKey = course;
}
}

result = mp.get(maxKey);
Collections.sort(result);
return result;

PROBLEM 12:  (Duke should be first)

A. The teamPoints map should map from team name to the total number of points scored by that team. A constructor should initialize any instance variables. Fill in the constructor for the BasketballPoints class below, creating a new instance of the map.

```java
/**
 * Initializes the teamPoints map
 */
public BasketballPoints() {
    teamPoints = new HashMap<String, Integer>();
}
```

B. The addResult map should incorporate the results from a single game for a team and update the teamPoints map. The method is passed the team name and an array indicating the number of one-point shots made (free throws), the number of two-point shots made, and the number of three-point shots made.

For example, `addResult("Duke", {19, 22, 9})` should add $1 \times 19 + 2 \times 22 + 3 \times 9 = 90$ points to Duke's overall point total stored in the map.

Complete the addResult method below. Be sure to read the comments for any hints.

```java
/**
 * Adds additional statistics to the map about a specific team.
 * Converts the shots array into the proper amount of points.
 * The team may or may not already be included in the map.
 * @param name The team name
 * @param shots An array in which the first element is the number of free throws made (1-pointers), the second is the number of two-pointers made, and the third is the number of three-pointers made.
 * For example, if shots = [3, 2, 7], then the point total to add is 28.
 */
public void addResult(String name, int[] shots) {
    int newTotal = 1 * shots[0] + 2 * shots[1] + 3 * shots[2];
```
teamPoints.put(name, teamPoints.get(name) + newTotal);
}

C. You decide to rank teams by total score, so that teams with a higher total points scored appear before teams with a lower points score. If each team has scored the same number of points, you should order the teams lexicographically (i.e., alphabetically).

Use the teamPoints map to complete the compare method of the PointsComp comparator. Read the comments to make sure you return the proper values.

```
/**
 * Comparator used to help sort the team names based
 * on their point totals stored in the map teamPoints.
 */
class PointsComp implements Comparator<String> {
    /**
     * a and b represent team names.
     * This method should return a negative number if a’s point total
     * is greater than b’s, a positive number if a’s
     * point total is less than b’s. If their point totals are equal, it should return a negative number
     * if a is before b alphabetically, a positive number if b is
     * before a, and 0 if both the points and team names are equal.
     */
    public int compare(String a, String b)
    {
        int pointDiff = teamPoints.get(b) - teamPoints.get(a);
        if (pointDiff != 0)
            return pointDiff;
        else
            return a.compareTo(b);
    }
}
```

D. Finally, complete the getBest method below, using the PointsComp comparator. Note that code has already been provided for you to convert the team names (the keys of teamPoints) into an ArrayList.

```
/**
 * Uses the PointsComp class to sort the team names based on
 * total points, and returns the team name in the map which
 * has the highest point total.
 *
 * @return a String representing the team’s name
 */
public String getBest() {
    //names is an ArrayList of all
    // the team names in the map.
    List<String> names = new ArrayList<String>(teamPoints.keySet());
    Collections.sort(names, new PointsComp());
    return names.get(0);
}
```

PROBLEM 13: (Over hill, Over dale, Overlap)
In this problem there is a lengthy explanation of the problem followed by three questions. You are modeling requests for one classroom. The requests are specified by the start-time and end-time for each class meeting 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.

We 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.

\[
(1,2) \quad (1,3) \quad (3,4) , \quad (1,4) , \quad (4,5) , \quad (3,5) , \quad (2,5) , \quad (6,7) \\
\]

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

\[
(1,2) \quad (2,5) \quad (6,7) \\
\]

or we could schedule these classes:

\[
(1,3) \quad (3,5) \quad (6,7) \\
\]

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

\[
(1,2) \quad (3,4) \quad (4,5) \quad (6,7) \\
\]

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

You’ll write code to manipulate the interval requests. You’ll also 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). The lowest end time comes first in the sorted array — see the sorted list above that starts with \((1,2)\) and ends with \((6,7)\) for an example.

2. Schedule the first (least) interval by adding it to an `ArrayList` being built that will contain the answer.

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` (it’s now the last-added interval).

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.

```
private static class Interval implements Comparable<Interval>{
    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(Interval o) {
    int diff = myEnd - o.myEnd;
    // complete code here in Part B
}

/**
 * Return time from the end of this interval to the beginning
 * of Interval i. If this interval doesn’t overlap with i
 * then return the time between when this ends and when i starts.
 * @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 Interval i.
 * @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 + "]";
}

A. 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, and intervals don’t overlap
 * 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){
    int sum = 0;
    for(int k=0; k < list.length-1; k++){
        sum += list[k].timeUntil(list[k+1]);
    }
    return sum;
}

B. 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.

private static class Interval implements Comparable<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(Interval o) {
        int diff = myEnd - o.myEnd;
        if (diff != 0) return diff;
        return o.myStart - myStart;
    }
}

C. 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 (so the sort call below works). 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 from list
     * @param list is an array of intervals
     * @return array containing maximal number of non-overlapping intervals
     */
    public Interval[] mostBusy(Interval[] list){
        ArrayList<Interval> maximal = new ArrayList<Interval>();
        Arrays.sort(list);
        maximal.add(list[0]);
        Interval last = list[0];
        for(int k=1; k < list.length; k++){
            Interval current = list[k];
            if (!current.overLaps(last)){
                maximal.add(current);
                last = current;
            }
        }
        return maximal.toArray(new Interval[maximal.size()]);
    }
return (Interval[]) maximal.toArray(new Interval[0]);