Homework 3: Plan Composition

Due **Tuesday November 15 11:00pm** via InstructAssist.

This assignment builds on this week's material on different ways to structure solutions to the same problem. It also gives you practice identifying the tasks within programming problems.

For each of the following problems, write **two** solutions, where each solution solves the problem using a different approach. Approaches count as different if they cluster at least some subtasks of the problems differently (like we saw for the Rainfall solutions in lecture); mere syntactic differences, such as replacing an element-based for-loop with an index-based one, or moving code into a helper without changing the order of the underlying computation, don't count as different.

As part of submitting the assignment, fill out this form, which requests answers to the following questions. Even if you work with a partner on the coding, submit your own individual answers to the form questions.

- For each problem, what are the (sub)tasks that make up that problem? For Rainfall, we'd be looking for a sequence of phrases such as "summing elements, counting elements, ignoring negative elements, stopping at the -999". This wording doesn't need to be exact (it'll be graded by humans) – they just need to convey the core computational tasks within the problem.

- For each problem, which of your solution structures do you prefer and why?

- Can you describe the general approach that each solution took (i.e., you cleaned out the data first, etc)?

Submitting the form is required. Your answers regarding identifying the sub-tasks will be graded for correctness. Your preferences and approach will not be graded for correctness as long as you make an honest attempt to answer those questions.

Please download and use this set of starter classes. The starter files contain all the classes needed for this assignment, as well as the classes in which to put your answers (to aid us in grading). You may add whatever methods you wish to these classes, but please don't rename any classes, fields, or methods. The section below on Using the Starter Files explains the starter files and where you should edit them with your solutions.

1 Programming Problems

A personal health record (PHR) contains four pieces of information on a patient: their name, height (in meters), weight (in kilograms), and last recorded heart rate (as beats-per-minute). A doctor's office maintains a list of the personal health records of all its patients.

Several problems will refer to these PHRs. The starter files contain a class for PHRs.

1.1 The BMI Sorter

Body mass index (BMI) is a measure that attempts to quantify an individual's tissue mass. It is commonly collected during annual checkups or clinic visits. It is defined as:

```
BMI = weight / ( height * height )
```

A simplified BMI scale classifies a value below 18.5 as "underweight", a value at least 18.5 but under 25 as "healthy", a value at least 25 but under 30 as "overweight", and a value at least 30 as "obese".

Design a program called `bmiReport` that consumes a list of personal health records (defined above) and produces a report containing a list of names (not the entire records) of patients in each BMI classification category. The names within each list in the report should be in the same order as in the original list of health records (you may assume that no two people have the same name). Use the `BMISummary` class in the starter files for the report.

The starter files provide a concrete test case for this method.

1.2 Data Smoothing

In data analysis, *smoothing* a data set means approximating it to capture important patterns in the data while eliding noise or other fine-scale structures and phenomena. One simple smoothing technique is to replace each (internal) element of a sequence of values with the average of that element and its predecessor and successor. Assuming that extreme outlier values are an abberation caused, perhaps, through poor measurement, this averaging process replaces them with a more plausible value in the context of that sequence.

For example, consider this sequence of `heartRate` values taken from a list of personal health records (defined above):

```
95 102 98 88 105
```

The resulting smoothed sequence should be

```
95 98.33333 96 97 105
```

where:

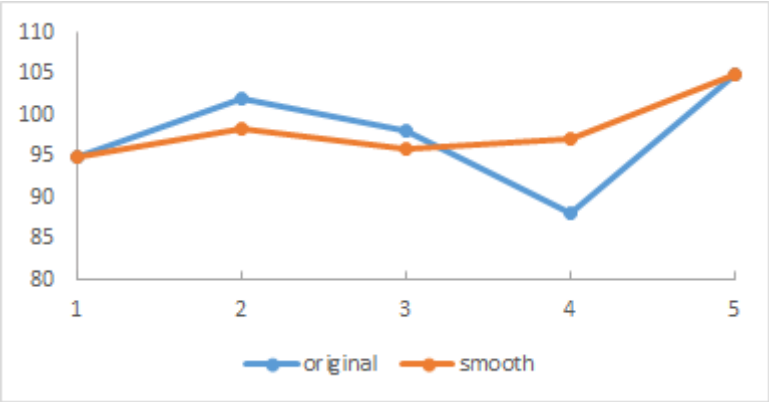
- `102` was substituted by `98.33333: (95 + 102 + 98) / 3`
- `98` was substituted by `96: (102 + 98 + 88) / 3`
- `88` was substituted by `97: (98 + 88 + 105) / 3`

(numbers such as 98.3333 do NOT need to be rounded or truncated).

Clarification added 11/10 at 7pm: Within the code, the result of data smoothing will be a list of doubles, so in an actual test case, each of the integer values in the sample output would need to have `.0` after them, as in

```
95.0 98.33333 96.0 97.0 105.0
```

This information can be plotted in a graph such as below, with the smoothed graph superimposed over the original values.



Design a program `dataSmooth` that consumes a list of PHRs and produces a list of the smoothed `heartRate` values (not the entire records).

Clarification added 11/10 at 7pm The heart rates in the `PHR` are of type int. When you average three ints, you don't always get a whole number, so the result of this function is a list of `Double`. We need to tell Java explicitly that you want the result of the division to be `double`. The easiest way to do this is to write your averaging expression along the following lines:

```
(double)(num1 + num2 + num3) / 3.0
```

(where you replace `num1`, `num2`, and `num3` with expressions for the numbers to average from your code.) This tells Java to sum the numbers, convert the sum to a `double`, and then do division with doubles (the use of 3.0 instead of 3).

Post to the forum if you have questions on this.

1.3 Earthquake Monitoring

Geologists want to monitor a local mountain for potential earthquake activity. They have installed a sensor to track seismic (vibration of the earth) activity. The sensor sends measurements one at a time over the network to a computer at a research lab. The sensor inserts markers among the measurements to indicate the date of the measurement. The sequence of values coming from the sensor looks as follows:

```
  20151004 200 150 175 20151005 0.002 0.03 20151007 130 0.54 20151101
78 ...
```

The 8-digit numbers are dates (in year-month-day format). Numbers between 0 and 500 are vibration frequencies (in Hz). This example shows readings of 200, 150, and 175 on October 4th, 2015 and readings of 0.002 and 0.03 on October 5th, 2015. There are no data for October 6th (sometimes there are problems with the network, so data go missing). Assume that the data are in order by dates (so a later date never appears before an earlier one in the sequence) and that all data are from the same year. The dates will always be 8-digit numbers in the format show above (and starting with a non-0 digit).

You may also assume that every date is followed by at least one frequency (in other words, every date has at least one measurement).

Design a program `dailyMaxForMonth` that consumes a list of sensor data (doubles) and a month (represented by a number between `1` and `12`) and produces a list of reports

(`maxHzReport`) indicating the highest frequency reading for each day in that month. Only include entries for dates that are part of the data provided (so don't report anything for October 6th in the example shown). Ignore data for months other than the given one. Each entry in your report should be an instance of the `maxHzReport` class in the starter files.

For example, given the sequence of values above and the month 10 (for October), the resulting list should be:

```
[maxHzReport(20151004, 200),
 maxHzReport(20151005, 0.03),
 maxHzReport(20151007, 130)]
```

2 Using the Starter Files

The starter files zip has a lot of files. Here's a guide to what you will find in there.

For each problem, there are at least three files:

- Files named `<Problem>1.java` and `<Problem>2.java`. Put one of your solutions in each of these files. The headers are already there, but the methods currently return `null` so that things will compile. Replace the method bodies with your methods. The `Earthquake` files also provide two helper functions for breaking down dates.

- A file named `<Problem>Examples.java`. Your test cases and data for the problem go in here, as usual. We have provided one simple test for each problem, to make sure you are computing what we expect you to. You need to add your own tests atop these.

- Auxiliary files with the classes for reports, PHRs, etc. The ones that are used in the output of methods have `equals` and `toString` methods, to aid checking your work.

Please post to the forum if you need help understanding the files. We set these up in advance to (a) help autograding, and (b) save you from having to do this setup yourselves.

3 Submission Guidelines

Edit the starter files, then zip them up and submit that zip to InstructAssist. There is a separate `Examples` class for each problem, which will assist us in autograding. Each test in your `Examples` classes should test only the single method for that problem.

Remember to fill in this form with free-response answers as well.

4 Grading and Testing Expectations

To aid us in grading, **submit tests written against the version 1 class (`BMI1`, `DataSmooth1`, `Earthquake1`)**. Follow the setup of tests shown in the starter files. If you want to test your version 2, simply change which version's class you create at the top of the `Examples` class. But please reset these to version 1 before submitting.

Your `Examples` classes do NOT need to have separate copies of the tests for both versions. We will assume you ran the same tests against both versions.

In grading this assignment, we will check for

- Whether your methods produce the right answers
- Whether your two solutions to the same problem differ in how they organize the computation
- Whether you are producing clean (well organized and documented) code
- Whether you have a good set of tests for these problems, where good means that your tests catch routine errors that a solution might make.
- Whether you can accurate identify the (sub)tasks involved in each problem.