CrsA Post-Assessment Problems

For each problem, write two solutions, where each solution solves the problem using a different approach. You should also determine which solution structure you prefer. Specify your preference with a brief discussion of why.

Approaches count as different if they cluster at least some subtasks of the problems differently (like we saw for the Rainfall solutions); merely syntactic differences, such as replacing an element-based for-loop with an index-based one, don't count as different. It has to be a different decomposition of the tasks (i.e., compositions of different plans).

In the end, if after racking your brain you simply can't think of two truly different ways of doing one of these problems, submit the two most different versions you can.

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.

```
data PHR:
   | phr(name :: String,
         height :: Number,
         weight :: Number,
         heart-rate :: Number)
end
```

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 function called `bmi-report`—

```
fun bmi-report(phrs :: List<PHR>) -> Report
```

—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 can be in any order. Use the following datatype for the report:

```
data Report:
   | bmi-summary(under :: List<String>,
                 healthy :: List<String>,
                 over :: List<String>,
                 obese :: List<String>)
end
```

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 aberration 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 `heart-rate` values taken from a list of personal health records (defined above):

```
95 102 98 88 105
```

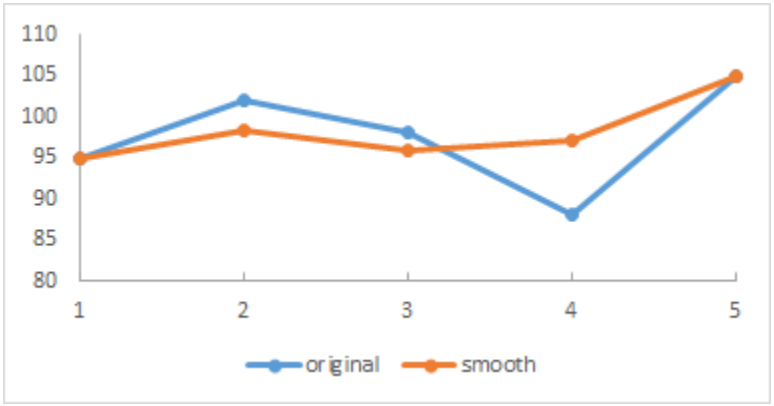
The resulting smoothed sequence should be

```
95 98.33 96 97 105
```

where:

- `102` was substituted by `98.33: (95 + 102 + 98) / 3`
- `98` was substituted by `96: (102 + 98 + 88) / 3`
- `88` was substituted by `97: (98 + 88 + 105) / 3`

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



Design a function `data-smooth`—

```
fun data-smooth(phrs :: List<PHR>) -> List<Number>
```

—that consumes a list of PHRs and produces a list of the smoothed `heart-rate` values (not the entire records).

1.3 Most Frequent Words

Given a list of strings, design a function `frequent-words`—

```
fun frequent-words(words :: List<String>) -> List<String>
```

—that produces a list containing the three strings that occur most frequently in the input list. The output list should contain the most frequent word first, followed by the second most frequent, then the third most frequent. Break ties by putting the shorter word (by length in characters) first. You may assume that the three most frequent words will have different length. You may also assume that the input will have at least three different words.

1.4 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 ...
```

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. Also, assume that every date reported has at least one measurement.

Design a function `daily-max-for-month`—

```
fun daily-max-for-month(sensor-data :: List<Number>, month :: Number) -> List<Report>
```

—that consumes a list of sensor data and a month (represented by a number between 1 and 12) and produces a list of reports 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 following datatype:

```
data Report:
   | max-hz(day :: Number, max-reading :: Number)
end
```