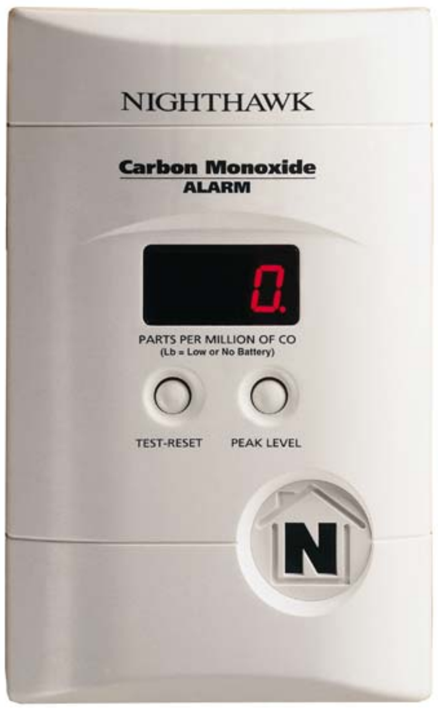
CSCI 135 Programming Exam #1
Fundamentals of Computer Science I
Fall 2014

This part of the exam is like a mini-programming assignment. You will create a program, compile it, and debug it as necessary. This part of the exam is open book and open web. You may use code from the course web site or from your past assignments. When you are done, submit all your Java source files to the Moodle exam #1 dropbox. Please ***double check you have submitted all the required files***.

You will have 100 minutes. No communication with any non-staff members is allowed. This includes all forms of real-world and electronic communication.

Grading. Your program will be graded on correctness and to a lesser degree on clarity (including comments) and efficiency. Partial credit is possible, so strive to provide a solution that demonstrates you know how to do as many parts of the problem as possible (even if there are bug(s) tripping up the total solution).



Overview. You are writing software to analyze and visualize the data logged by a carbon monoxide sensor. The sensor periodically records the carbon monoxide concentration in parts-per-million (ppm). The sensor hardware you are using detects carbon monoxide concentrations from 0 to 2000 ppm (inclusive of 0 and 2000). However, your meter sometimes logs spurious samples outside the range [0, 2000]. You will first develop a library of static methods that performs various calculations on a sequence of samples recorded by the detector. You will then build a client program that reads in sample data from standard input and graphically displays a bar chart of the valid data.

To get started, create an empty Eclipse project. When you create the project, we recommend you select the "Use project folder as root for sources and class files" option. Also, if using a lab machine, be sure to select "Java 1.7" as the execution JRE. Extract the contents of this zip file into your project directory (using the Windows explorer or Mac OS finder): http://katie.mtech.edu/classes/csci135/sample.zip

Part 1: Sample. This class provides a library of static methods that computes various things about a sequence of samples recorded by the detector. Each individual sensor sample is a integer representing the carbon monoxide concentration. A sample sequence is stored as an `int` array. You need to implement the methods in the following API:

```
class Sample
-----------------------------------------------------------------------
    boolean valid(int sample)
     double lengthInSeconds(int [] samples, int samplesPerSecond)
        int numValid(int [] samples)
     double avgValid(int [] samples)
     int [] allValidSamples(int [] samples)
    boolean alarmAt(int [] samples, int pos, int maxLevel)
```

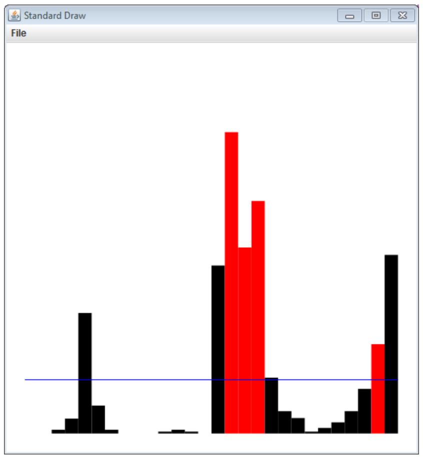
We have provided a stub version of `Sample.java` that includes comments describing exactly what each method does. You can assume arrays passed in as arguments are not `null`, although they may be of length 0. We have provided a placeholder `return` statement in each method so the class compiles. These placeholder `return` statements can be replaced as you implement each method.

Note: You may want to develop your methods in the order listed above. This will help you avoid repeated code; the earlier methods are often useful for implementing the latter methods.

We have provided a test `main()` method in the stub version of `Sample.java`. Here is our test output:

```
% java Sample

valid(-1)   = false
valid(0)    = true
valid(20)   = true
valid(2000) = true
valid(2001) = false

lengthInSeconds(samplesA, 1)  = 7.000
lengthInSeconds(samplesA, 3)  = 2.333
lengthInSeconds(samplesC, 3)  = 0.333
lengthInSeconds(samplesD, 3)  = 0.000

numValid(samplesA) = 7
numValid(samplesB) = 5
numValid(samplesC) = 0
numValid(samplesD) = 0

avgValid(samplesA) = 20.000
avgValid(samplesB) = 210.000
avgValid(samplesC) = 0.000
avgValid(samplesD) = 0.000

allValidSamples(samplesA) = [0, 0, 10, 50, 80, 0, 0]
allValidSamples(samplesB) = [200, 0, 500, 250, 100]
allValidSamples(samplesC) = []
allValidSamples(samplesD) = []

alarmAt(samplesA,  2, 50) = false
alarmAt(samplesA,  3, 50) = false
alarmAt(samplesA,  4, 50) = false
alarmAt(samplesA,  3, 5)  = true
alarmAt(samplesA, -1, 5)  = false
alarmAt(samplesA,  9, 5)  = false

alarmAt(samplesB, 0, 5)  = false
alarmAt(samplesB, 1, 5)  = false
alarmAt(samplesB, 2, 5)  = false
alarmAt(samplesB, 3, 5)  = false
alarmAt(samplesB, 4, 5)  = true
alarmAt(samplesB, 5, 5)  = false
alarmAt(samplesB, 6, 5)  = false
alarmAt(samplesB, 7, 5)  = false
```

Part 2: GraphSample. This client program makes use of your `Sample` library as well as `StdDraw` and `StdIn` to draw a bar chart of all the valid samples in a sequence:



This program reads the time series data from standard input. The input file starts with an integer specifying the number of samples followed by the sequence of integer sample values. Here is the file `samples-b.txt`:

```
% more samples-b.txt
8
200 0 -10 500 250 100 2010 92822
```

The program should meet the following requirements:

- Takes a single ***optional*** integer command-line argument specifying a maximum sample level for purposes of determining which samples cause an alarm. If no command-line argument is given, use a default maximum sample level of 200.
- Print to the console the number of sample points, the number of valid samples, and the average value of the valid samples, ***rounded to one decimal*** place. Example output for `samples-b.txt`:
```
      total = 8, valid = 5, average = 210.0
```
- Valid samples in the input are visualized as vertical bars. Invalid samples are ***not*** included in the plot.
- Samples are drawn in order from left to right with the first sample on the left-hand side.
- The height of each bar is directly proportional to the sample value. A sample value of 2000 should occupy the entire vertical drawing area (excluding the buffer area added automatically by `StdDraw`).
- Bars should be drawn in black with the exception of samples that would cause an alarm based on the maximum sample level (the optional command line argument). Bars representing samples that would cause an alarm should be drawn in red. *Note:* For determining which samples are alarms, ***ignore invalid samples*** (i.e. call `alarmtAt()` with an array containing only valid samples).
- On top of the bar chart, the average of the valid samples should be drawn as a horizontal blue line.

You should obviously use the static methods in your `Sample` library to help graph the data. For full credit, you should avoid repeating logic in `GraphSample` that could be done by making use of the `Sample` library.

Depending on the details of your implementation, you will also need some or all of the following `StdDraw` methods:

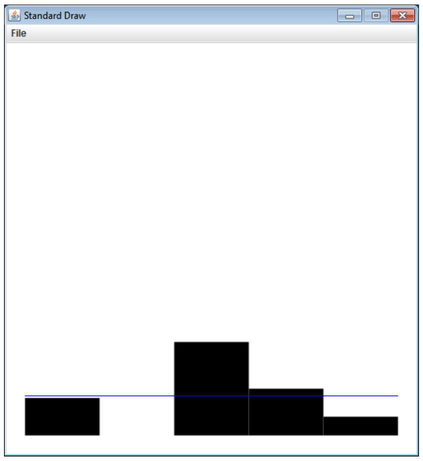
```
public class StdDraw
```

```
void setXscale(double min, double max)
void setYscale(double min, double max)
void setPenColor(Color color)
void line(double x0, double y0, double x1, double y1)
void filledRectangle(double x, double y, double halfWidth, double halfHeight)
```

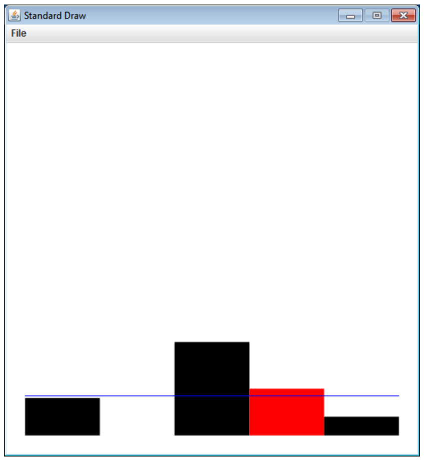
The next page provides some example runs on our provided data files.

Submission. Submit your source files: `Sample.java` and `GraphSample.java` to the Moodle dropbox.

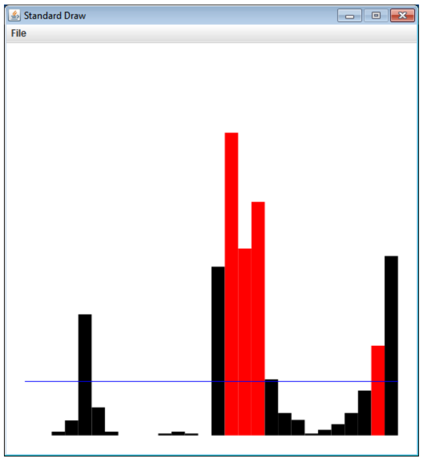
```
% java GraphSample < samples-b.txt
total = 8, valid = 5, average = 210.0
```



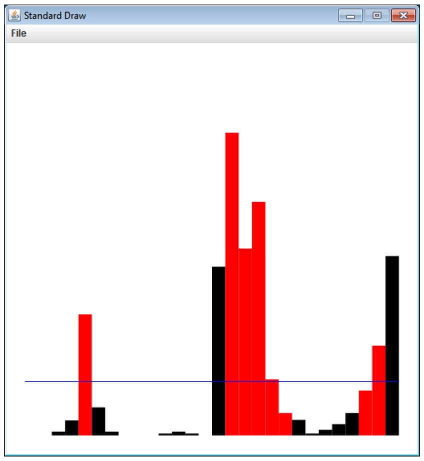
```
% java GraphSample 50 < samples-b.txt
total = 8, valid = 5, average = 210.0
```



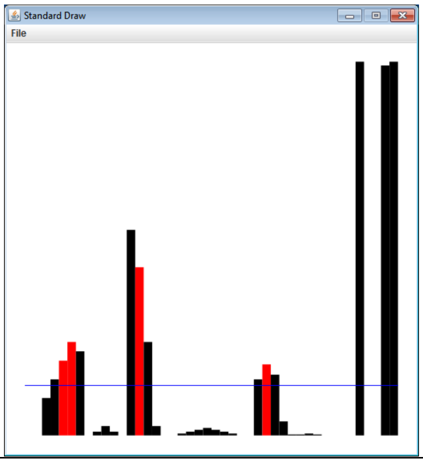
```
% java GraphSample < samples-30.txt
total = 30, valid = 28, average = 290.5
```



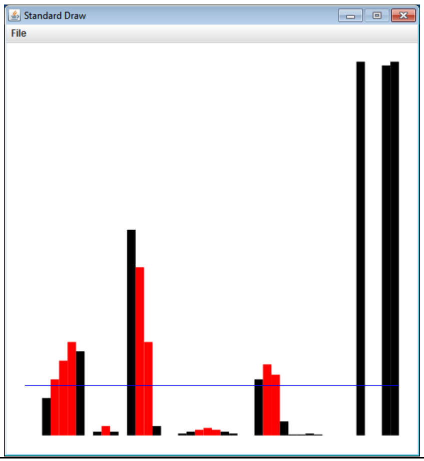
```
% java GraphSample 79 < samples-30.txt
total = 30, valid = 28, average = 290.5
```



```
% java GraphSample < samples-50.txt
total = 50, valid = 44, average = 266.7
```



```
% java GraphSample 10 < samples-50.txt
total = 50, valid = 44, average = 266.7
```


Standard Draw
File