CS 211, ALL SECTIONS

PROJECT 1

DUE SUNDAY, FEBRUARY 10TH AT MIDNIGHT

The objective of this project is to implement a set of data processing tools, exercising skills in writing programs which utilize conditions, loops, and arrays

---

OVERVIEW:

1. Create the Java class `DataProcessor`.
2. Implement the ten data processing routines described below.
3. Students in the honors section will have an additional routine to complete for full credit.
4. Download and use the tester module to ensure that your program is correct.
5. (Optional) Use the `main` method to perform your own tests.
6. Prepare the assignment for submission and submit it.

Imagine that you're building a toolkit of functions for data processing (perhaps because you are doing some data analysis work and foresee needing to call certain routines over and over again). To do this, you create a class (`DataProcessor`) with several static methods which can be called to compute the result of some function; these methods will be described below.

RULES

1. This project is an individual effort; the Honor Code applies
2. You may not import any extra functionality besides the default (i.e. System, Math)
3. The `main` method will not be tested; you may use it any way you want.

INSTRUCTIONS: Download the template source for `DataProcessor.java`:

- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/DataProcessor.java

This file contains template code for the project. Your task will be to fill in each of the methods as described. Whenever you implement a method, be sure to delete the line which throws an exception. The exceptions are placeholders which keeps the unit tester running correctly even when nothing is present.

METHOD 1:

```
public static boolean inOrder(int[] values)
```

This method should return `true` if and only if every element in the input array is in ascending order, meaning that none of the values in the array are smaller than the previous element, and `false` otherwise. For example, the array `{1, 2, 5, 5, 6, 7, 8, 8, 8}` would be in order. An empty array is considered in order as well.

METHOD 2:

```
public static int smallest(int[] values)
```

This method will return the smallest value found in the array. For example, for the array `{4, 2, 3, 1, 5, 1}`, the value `1` will be returned. If the array is empty, then the built-in constant `Integer.MAX_VALUE` may be returned.

METHOD 3:

```
public static int smallestGreaterThanN(int n, int[] values)
```

This method will return the smallest value found in the array which is larger than the input `n`. For example, for the input `n = 2` and the array `{4, 2, 3, 1, 5, 1}`, the value `3` will be returned. If no such value exists, then the method will return the value `n`.

METHOD 4:

```
public static int numDistinct(int[] values)
```

Return the total number of distinct values contained in the input array. For example, the array `{2, 2, 5, 1, 7, 4, 7}` has `5` distinct values (the numbers 1, 2, 4, 5, and 7). Each of those five values appear at least once (the numbers 2 and 7 appear twice each, but they are only counted once).

Hint: calls to `smallest()` and `smallestGreaterThanN()` may help you solve this.

METHOD 5:

```
public static int numOccurrences(int v, int[] values)
```

Determines the number of times element `v` occurs within the array of values. For example, in the array `{1, 2, 5, 5, 3, 2}`, the value `v = 2` occurs twice.

METHOD 6:

```
public static int numInRange(int a, int b, int[] values)
```

Determines the number of elements from the array of values which have a value within the range `[a, b)` (that is, between `a` and `b`, including `a` but not including `b`). For example, if the input array is `{1, 2, 3, 4, 2, 3, 4, 3, 4, 4}` with `a = 2` and `b = 4`, the result would be `5` (because there are two 2's and three 3's).

METHOD 7:

```
public static double mean(int[] values)
```

Finds the arithmetic mean (the average) of the list of numbers. For example, if the input is `{1, 2, 3, 4}`, then the mean is `2.5`.

Hint: the input values are integers, but the output most likely won't be. Do you need to account for that somehow?

METHOD 8:

```
public static double meanInRange(int a, int b, int[] list, double[] values)
```

The input of this method includes two arrays, which you may assume are the same size, and whose entries correspond to one another (for example, the first array may be a list of ages of several people, while the second array may be a list of the heights of the same people). The method will compute the average of values from the second array, but only when the values correspond to entries from the first list within the range `[a, b)`. For example, if the first array is a list of ages, `{19, 15, 20, 25, 30, 5, 32}`, the second array is a list of heights in feet, `{5.2, 4.9, 5.9, 6.1, 5.85, 3.2, 5.7}`, and the range is `a = 20`, `b = 30`, then it would give the average of people aged 20-29, which in this example would be `6.0`. The ages 20 and 25 in this example are in range, giving the mean `(5.9 + 6.1)/2 = 6.0`.

METHOD 9:

```
public static int[] cleanList(int a, int b, int[] list)
```

When performing data analysis, it is often necessary to first clean obvious outliers from the data set, because these may be the result of a clerical error and could throw off analysis. As an example, when studying medical data, sometimes you will see "zombies" (people with hospital visits occurring after their recorded date of death) or "vampires" (people who are hundreds of years old, because their date of birth was left blank and defaulted to zero when entered into the the system). This method will eliminate entries from the array of input values which are outside of the range `[a, b)`, and return the list of entries which remain. For example, for the input `a = 0`, `b = 125`, and `{10, -1, 20, 30, 25, 125, 40, 200}`, the array which is returned would be `{10, 20, 30, 25, 40}`.

Hint 1: you don't need to delete elements from the input array. You can create a whole new array to return instead.

Hint 2: it may help to know how big your output array is before you begin. Would any of the other methods you've written help here?

METHOD 10:

```
public static double[] nearestNeighbor(double v, double[] values)
```

Return the value of the nearest neighbor to `v`. If `v` is some value, look through the list of values to find the one which is closest in value to `v`. For example, if `v = 1.2` and the input array is `{3.2, 1.7, 2.4, 0.9, 4.4}`, then the method will return `0.9` because it is the value closest to `1.2` in the list. If there is a tie (for example, both `0.9` and `1.5` are the same distance to `1.2`), then return the one which occurs first in the list. If the list is empty, then return `v`.

HONORS SECTION:

If you are in the honors section, you must complete this part and it is worth 20 points of the project grade. If you are not in the honors section, you are welcome to attempt this but you do not need to complete it and it is not worth any points if you do.

```
public static double kNearestNeighbors(int k, double v, double[] values)
```

Find and return the `k` elements which are closest to `v` in value, in order of increasing distance from `v`. For example, if `v = 1.2`, `k = 3`, and the input array is `{3.2, 1.7, 2.4, 0.9, 4.4}`, then the method will return `{0.9, 1.7, 2.4}`. If `k` is larger than the length of the input array, then the output array should only have as many values as you have in the original array.

You will need to uncomment the lines related to the kNearestNeighbors tests in the tester file to be able to test this method.

TESTING:

Download the following (the `jar` file is not necessary if you are testing from within Dr Java):

- https://mason.gmu.edu/~iavramo2/classes/cs211/junit-cs211.jar
- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/P1tester.java

These include sample test cases. For grading, we may modify the set of test cases.

When you think your programs are ready (you can run them even without the tester if you write a main method), do the following from the command prompt to run the tests.
On Windows:

```
javac -cp .;junit-cs211.jar *.java
java -cp .;junit-cs211.jar P1tester
```

On Mac or Linux:

```
javac -cp .:junit-cs211.jar *.java
java -cp .:junit-cs211.jar P1tester
```

In Dr Java:

- open the files, including `P1tester.java`, and click the *Test* button.

GRADING:

The implementation of each method described above is worth 10 points of the total. Of those 10 points, half will be based on automatic testing (the percent of test cases which passed). 1 point will be granted for style (i.e. commenting code, not writing unreadable code). The remainder will be awarded based on manual inspection. Thus, partial credit is possible, but hard-coding to pass a certain set of test cases will receive an automatic zero on the manual inspection points. Students in the honors section will have an additional 20 points assessed based on the completion of the designated honors section method.

Programs which do not compile without modification will receive a zero in most cases.

SUBMISSION:

Submission instructions are as follows.

1. Let *xxx* be your lab section number, and let *yyyyyyyy* be your GMU userid. Create the directory *`xxx_yyyyyyyy_P1/`*
2. Place the file `DataProcessor.java` in the directory you've just created.
3. Create the file `ID.txt` in the format shown below, containing your name, userid, G#, lecture section and lab section, and add it to the directory.

   Full Name: Donald Knuth
   userID: dknuth
   G#: 00123456
   Lecture section: 004
   Lab section: 213

4. compress the folder and its contents into a .zip file, and upload the file to Blackboard.