INF100 H17 Semester Assignment 2

Due on Friday, October 20, 2017

Albin Severinson, Dag Haugland

Abstract

For the second semester assignment, we will be doing forensic science. Specifically, we will use Newton's law of cooling together with Monte Carlo simulations to help the police figure out how long ago a murder took place. For each problem we will write one or two methods and the subsequent problems will depend on these methods. You should only hand in your final program. We will only grade this final program. The program is graded on a 0 to 100 points scale. If you have questions specific to this assignment you should send them to me at `albin.severinson@uib.no`.

Submission

1. Add your program file to a `.zip` archive named `sx2_firstname_lastname.zip`. You do this by right clicking on the file ending with `.java` that contains your program code and choosing compress or add to archive (exactly what it is called varies by operating system and version). You should only add your final program, i.e., the one for problem 6. Remember that the file name must match the class name and that you therefore can not rename the `.java` file! Ask your group leader if you are having issues with this. Note that handing in the wrong file may cause you to fail the assignment!
2. If you complete the voluntary getting ahead part of the assignment, it should be handed in separately. If you complete this part you thus need to hand in two files (the `.java` file for problem 6 and the `.java` file for the getting ahead part). Put both files into the same `.zip` archive.
3. Log in to `https://mitt.uib.no` and press the box marked with "INF100" to enter the page for this course.
4. Press "Oppgåver" on the left. Next, select "Semester Assignment 2".
5. Press "Lever oppgåve" and upload the `.zip` file you created in step 1.

Grading

- This assignment is worth a total of 100 points. Each problem is worth 20 points.
- We will deduct points for poor code. For example, lack of comments, poorly chosen variable names (please do not use non-ASCII characters such as Ø in variable names :)), incorrect indentation, and code repetition may be reason to deduct points.
- Have a look at the Google Java style guide if you are unsure on how to structure some part of your code: `https://google.github.io/styleguide/javaguide.html`. See the next section for comment guidelines.
- Make sure your solution actually does what the problem asks for. We may deduct points if you implement something other than what the problem asks for.

Comment Guidelines

These comment guidelines are adapted from the guidelines of David Eck at Hobart and William Smith Colleges (see `http://math.hws.edu/eck/cs124/f11/style_guide.html`).

- Every variable that has a non-trivial role in the program should have a comment that explains its purpose. For-loop variables and other local utility variables do not, in general, have to be commented.
- Comments can be included in the body of a method when they are needed to explain the logic of the code. In general, well-written code needs few comments.
- Comments should never be used to explain the Java language. A comment such as "declare an int variable named x" or "increment the variable ct" is worse than useless. Assume that your reader knows Java! (Note that such comments are sometimes used in programming textbooks or on the blackboard, but never in real programs.)

General Guidelines

These are general guidelines that will (hopefully) help you maintain your sanity through your programming endeavours.

- Have your computer indent your code for you. You do this in DrJava by selecting all of your code, right clicking, and selecting `indent lines`.
- You should make a copy of all your code after completing each problem so that you can roll back to a previous version when you inadvertently break something. Even the developers at Google sometimes have to roll back.
- Set aside time to clean up your code. The number that gets bandied around on the internet is that the best programmers spend roughly half their time cleaning up and rewriting their code, i.e., you are halfway done when you get it working.
- You save time by making sure you properly understand your current program before moving on.
- You should probably be spending a significant part of your time on stackoverflow.
- Google is your friend when you want to figure out how a standard class or method works (`Scanner`, say). For example, searching for “java Scanner” gives you its Java docs page as the first result.
- When in doubt, look at the Google Java style guide to figure out how to write something.

Body Temperature

The police recently found a body and have already apprehended several suspects. Furthermore, the police have a good idea of at what time each suspect was at the scene of the crime. Knowing the time of death would go a long way towards figuring out who the murderer is.

You are doing an internship as a forensic scientist at the police department. Your boss (the forensic scientist in charge of the case) has been tasked with figuring out the time of death. However, your boss has fallen ill and solving the case is now up to you! Fortunately, the police measured the temperature of the body immediately upon discovering it. Furthermore, your boss wrote the following piece of bode before falling ill (this code is also available on the assignment page at `http://mitt.uib.no`):

```
/**
 * Compute the time in hours required for a body to cool down to temperature
 * degrees. Gaussian noise is added to simulate parameter uncertainty.
 * @param temperature The temperature of the body when found.
 * @return the time in hours required for the body to cool down to
 * temperature degrees.
 */
public static double cooldown(double temperature) {
    // we need this object to generate Gaussian random variables
    // (remember to import java.util.Random)
    Random random = new Random();

    // the average body temperature of a (living) human
    double bodyTemperature = 37;

    // add noise to simulate that the body temperature of the victim at the
    // time of death is uncertain
    bodyTemperature += random.nextGaussian();

    // compute the time required for the body to cool down from
    // bodyTemperature to temperature using Newton's law of cooling.
    double cooldownTime = Math.log(bodyTemperature / temperature);
    cooldownTime *= 1 / bodyTemperature;

    // normalize this value such that cooling down from 37 to 32 degrees
    // takes 1 hour. we assume that we have measured this for the
    // environment that the body is found in. we add Gaussian noise to
    // simulate measurement uncertainty.
    cooldownTime *= 255 + random.nextGaussian();

    return cooldownTime;
}
```

The code makes use of Newton's law of cooling (see `https://en.wikipedia.org/wiki/Newton%27s_law_of_cooling`), to compute the time that has passed since death based on the current temperature of the body. Because we do not know the body temperature of the victim at the time of death exactly, Gaussian noise (see `https://en.wikipedia.org/wiki/Gaussian_noise`) is added to the function parameters. Due to this uncertainty we need to perform the computation many times. Furthermore, because there are several suspects it is not enough to only consider the average value! Instead, we need to compute the probability of the victim dying within the time frames that each suspect visited the scene of the crime. For this assignment we will produce this data. Specifically, we will produce what a statistician would call a histogram (see `https://www.mathsisfun.com/data/histograms.html`).

Problem 1 ($\frac{100}{6}$ points)

The first step is to call the `cooldown` method many times and store the resulting values. We refer to this as sampling. Write a method that returns an array of `numSamples` cooldown samples. Each element of the array should contain the result from a call to `cooldown(temperature)`. Print the resulting array from the main method (see the previous assignment for a primer on printing arrays). As you see there is a span of possible times at which the victim may have died.

```
public static double[] cooldownSamples(int temperature, int numSamples) {

}
```

Example

Example for parameters `temperature=27`, `numSamples=10`.

```
[2.0088799960771184, 2.121420889236832, 1.9396865921089017,
2.4044747294759574, 2.2430778650951178, 2.083040119880876,
2.0595035785038114, 2.1782979876210806, 1.8812817807415378,
2.232108837421659]
```

Problem 2 ($\frac{100}{6}$ points)

We will need some helper methods before moving on with the crime solving. Write methods that return the minimum and maximum values from an array. Specifically, write the following two methods:

```
public static double minFromArray(double[] array) {

}
```

```
public static double maxFromArray(double[] array) {

}
```

This very pragmatic way of naming functions is called Apps Hungarian (see `https://www.joelonsoftware.com/2005/05/11/making-wrong-code-look-wrong/`). It makes it very clear what each method expects as input and what it returns. Interestingly, the author of the above article (Joel Spolsky) is the co-founder of Stack Overflow! Joel Spolsky has also written several other excellent articles on management, software design and business.

Example

Running `minFromArray` and `maxFromArray` for `double[] array = {1.3, 2.2, 3.1};` should return `1.3` and `3.1` respectively. Call the `minFromArray` and `maxFromArray` methods with this input from your main method to verify that they work correctly before moving on.

Problem 3 ($\frac{100}{6}$ points)

Now that we have an array of samples we need some way of computing what samples occur most frequently. For example, if we divide the number of samples that fall in the range 2.0 to 2.1 by the total number of samples we get the probability of the victim dying in the range 2.0 to 2.1 hours ago. We would like to do this for many small ranges and compute the probability of the victim dying within each of the intervals. To do this we first need to split the range of values in the array returned from `cooldownSamples` into small ranges. Next, we need to count the number of samples that fall within each range. This is the signature of the method you should write:

```
public static double[] countsFromArray(double[] array, int numRanges) {
    double[] counts = new double[numRanges];

    // your code...

    return counts;
}
```

We first explain how this method should work for `double[] array = {1.0, 1.0, 2.0, 3.0, 5.0}` and `numRanges=5`. After this we will explain it more generally.

1. Declare the array `double[] counts = new double[numRanges]`.
2. Compute `max = 5`, `min = 1`.
3. Compute `rangeSize = (max - min) / (numRanges - 1) = (5 - 1) / (5 - 1) = 1`.
4. Iterate over the elements of `array`.
   (a) `(int) ((array[0] - min) / rangeSize) = 0`, so we increment `counts[0]` by 1.
   (b) `(int) ((array[1] - min) / rangeSize) = 0`, so we increment `counts[0]` by 1.
   (c) `(int) ((array[2] - min) / rangeSize) = 1`, so we increment `counts[1]` by 1.
   (d) `(int) ((array[3] - min) / rangeSize) = 2`, so we increment `counts[2]` by 1.
   (e) `(int) ((array[4] - min) / rangeSize) = 4`, so we increment `counts[4]` by 1.
5. We now have `counts = [2.0, 1.0, 1.0, 0.0, 1.0]`.
6. Return `counts`.

More generally, this method should split the range of values in `array` into `numRanges` equally sized ranges. First, use the `minFromArray` and `maxFromArray` methods to get the range of values in `array`. Next, divide this range into `numRanges` ranges of equal size `rangeSize = (max - min) / (numRanges - 1)` (this is an acceptable approximation that simplifies the code). Finally, iterate over `array` and count the number of samples that fall within each range. Use the `counts` array (defined in the above method) to count the number of samples that fall within each of the `numRanges` ranges. Specifically, the i-th element of `counts` should be the number of values within `array` that satisfies `rangeSize * i <= (value - min) < rangeSize * (i + 1)`, where `min` is the minimum value in `array`. Note that `i = (int) ((value - min) / rangeSize)`. We return an array of type `double` rather than `int` so that we can use the methods from the previous problem.

Example

Calling `countsFromArray` with `double[] array = {1.0, 1.0, 2.0, 3.0, 5.0};` and `numRanges=5` should return

```
[2.0, 1.0, 1.0, 0.0, 1.0]
```

Calling `countsFromArray` with the array from the example of problem 1 and `numRanges=10` should return

```
[1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 2.0, 0.0, 0.0, 1.0]
```

Problem 4 ($\frac{100}{6}$ points)

Next, we need a way of presenting this data nicely. Write a method

```
public static void printArray2d(String[][] array2d) {

}
```

This method should print each element of the two-dimensional input array and should print a newline between the elements of `array2d[0]`, `array2d[1]`, and so on.

Example

Calling `printArray2d` for `String[][] array2d = {{"#", "#", ""}, {"#", "", ""}, {"#", "#", "#"}};` should print

```
##
#
###
```

Problem 5 ($\frac{100}{6}$ points)

Now we have the data we are interested in (the array return from `countsFromArray`) and a nice way of printing 2D arrays. The next step is thus to convert the array returned from `countsFromArray` into something we can print with `printArray2d`. Define the method `array2dFromCounts`

```
public static String[][] array2dFromCounts(double[] counts) {
    final int PRINT_WIDTH = 50;
    String[][] array2d = new String[counts.length][PRINT_WIDTH];

    // your code...

    return array2d;
}
```

If the i-th element of `counts` has value 2.0 and the maximum value in `counts` is 3.0, then the first `(int) (2 * PRINT_WIDTH / 3)` elements of `array2d[i]` should be given the value "#". The remaining `PRINT_WIDTH - (int) (2 * PRINT_WIDTH / 3)` values should be given the value " ", i.e., a string with only a space. More generally, if the i-th value of `counts` is k, the first `(int) (k * PRINT_WIDTH / max)` elements of `2dArray[i]` should be given the value "#", where `max` is the maximum value in `counts`. Print this array with the `printArray2d` method.

Example

Put the following code in your main method.

```
double[] array = {2.0088799960771184, 2.121420889236832, 1.9396865921089017,
                  2.4044747294759574, 2.2430778650951178, 2.083040119880876,
                  2.0595035785038114, 2.1782979876210806, 1.8812817807415378,
                  2.232108837421659};
double[] counts = countsFromArray(array, 10);
String[][] array2d = array2dFromCounts(counts);
printArray2d(array2d);
```

It should print this text to the terminal

```
##########################
##########################
##########################
##################################################
##########################
##########################
##################################################


##########################
```

Problem 6 ($\frac{100}{6}$ points)

It is finally time to put it all together! We will write a method that prints `array2d` as well as some additional things. The method should have signature

```
public static void printReport(String[][] array2d,
                               double arrayMin, double arrayMax) {

}
```

This method should print a title, the minimum value of the array and the maximum value of the array. It should also print the time interval that each printed line corresponds to. Use `array2d.length` to get the size of `array2d`. Between the minimum and maximum values `array2d` should be printed using `printArray2d`.

Example

Put the following code in your main method.

```
double[] array = cooldownSamples(27, 100000);
double[] counts = countsFromArray(array, 20);
String[][] array2d = array2dFromCounts(counts);
printReport(array2d, minFromArray(array), maxFromArray(array));
```

It should print the following to the terminal

```
Time since death probability distribution
- Each line corresponds to 0.06 hours.
================================================
1.45 hours
5
10  #
    ###
    #######
    ##############
    ##########################
15  ######################################
    ################################################
    ################################################
    ########################################
    ########################
20  ##########
    ###
25  2.60 hours
    ================================================
```

In this case we know for sure that the murder took place between 1.45 and 2.60 hours ago. Furthermore, by counting the lines and the #'s (all of the #'s combined correspond to 100%), we can figure out the probability of the murder being committed within any time interval between 1.45 and 2.60 hours ago! Try increasing the number of ranges and `PRINT_WIDTH` to increase the resolution of the printout.

Getting Ahead

The chief of police is very pleased with your report. In fact, it proved to be a critical piece of evidence in the murder trial! We can make an interesting extension though. Implement a method that prints the same method to a file instead of to the terminal. Here is a primer on writing to files: `https://www.youtube.com/watch?v=Bws9aQuAcdg`. If you came this far you could also consider printing the report and hanging it on your fridge :)

This part is meant as preparation for the coming course material and will not be graded. However, we will still look at it and comment it! Furthermore, you never have to catch up if you are always getting ahead. This part should be handed in separately. If you complete this part you thus need to hand in two files (the `.java` file for problem 6 and the `.java` file for this part). Put both files into the same `.zip` archive.