Assignment 1

Tools & Techniques for Large-Scale Data Analytics (CT5105)

NUI Galway, Year 2015/2016, Semester 1

- ***Submission deadline (strict): Wednesday, 30th September, 23:59***
- *Put all your answers and code files into a single .zip archive with name "YourName_Assignment1.zip" and submit via Blackboard*
- *Include all source code files required to compile and run your code*
- *Unless specified otherwise in the question, use only plain Java 8 (without any external libraries or frameworks)*
- *Please note that all submissions will be checked for plagiarism*
- *Use comments to explain your source code. Missing or insufficient comments can lead to mark deductions*

Remarks: 1) Some parts of this assignment require knowledge from the next lecture, but you can already start working on it. 2) Even if questions are wordy, they normally don't require much coding.

Question 1 [30 marks]

You are given the following implementation of a sorting algorithm (Bucket Sort):

```java
public static int[] bucketSort(int[] numbers, int bucketCount) {

    if (numbers.length <= 1) return numbers;

    int maxVal = numbers[0];
    int minVal = numbers[0];

    for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] > maxVal) maxVal = numbers[i];
        if (numbers[i] < minVal) minVal = numbers[i];
    }

    double interval = ((double)(maxVal - minVal + 1)) / bucketCount; // range of bucket

    ArrayList<Integer> buckets[] = new ArrayList[bucketCount];

    for (int i = 0; i < bucketCount; i++)  // initialize buckets (initially empty)
        buckets[i] = new ArrayList<Integer>();

    for (int i = 0; i < numbers.length; i++)  // distribute numbers to buckets
        buckets[(int)((numbers[i] - minVal)/interval)].add(numbers[i]);

    int k = 0;

    for (int i = 0; i < buckets.length; i++) {

        Collections.sort(buckets[i]);  // calls Java's built-in merge sort

        for (int j = 0; j < buckets[i].size(); j++) { // update array with the bucket content
            numbers[k] = buckets[i].get(j);
            k++;
        }
    }

    return numbers;
}
```

PTO

a) Modify this code so that it makes use of multithreading (where appropriate). Use Java 7-style code for this. Also, create a `main()` method which invokes `bucketSort()` with some example array of numbers and prints the result.

b) The code above requires a “helper” sorting method. More precisely, it invokes merge sort from the Java API (the `Collections.sort()` call). Modify your code from a) so that method `bucketSort()` takes an additional parameter `sortFunction.` This parameter should represent the sorting function used in place of `Collections.sort`. In other words, parameter `sortFunction` should allow the caller of `bucketSort()` to specify any sorting method in place of `Collections.sort` (provided it has the same signature as `Collections.sort`). Again, use Java 7-style code for this question. Hint: Anonymous classes.
Also, extend your `main()` method so that it invokes `bucketSort()` with the new parameter (you don’t need to implement another sorting method, you can simply pass `Collections.sort` into the `sortFunction` parameter).

c) Modify your solution for b) so that a Java 8 lambda expression is used instead of an object of an anonymous class. That is, `bucketSort()` should accept argument `sortFunction` in form of a lambda expression.
You don’t need to implement another sorting method, you just need to wrap the call of `Collections.sort` in a lambda expression and pass it into `bucketSort()` where it is evaluated. Also, extend `main()` again in order to test your modification.

Question 2 [30 marks]

Suppose you want to analyze the data produced by a couple of weather stations (meteorological stations). Create a class `WeatherStation` with three attributes: the *location* of the station (simply a string with the name of the country in which the station is located), the station’s stream of measurements (a Java 8 stream of values of class `Measurement`), and a static field *`stations`* (a list of all existing weather stations). Class `Measurement` should have attributes *time* (an integer number, representing the time of the measurement) and *temperature* (a double number).

Add two methods to class `WeatherStation` which perform simple analytics tasks: a method `averageTemperature(startTime, endTime)` and a static method `medianTemperature(startTime, endTime, listOfLocations)`. The first method should return the average temperature measured by the weather station between `startTime` and `endTime`. The second method should return the *median* of all temperatures measured by all the weather stations with a location in `listOfLocations` (a list of country names) between `startTime` and `endTime`. Important: implement these methods using Java 8 stream operations, as far as possible.

Also add a `main()` method where you create a few example objects of class `WeatherStation` (stored in the list in static field *stations*), and invoke `averageTemperature()` and `medianTemperature()` with a few example values of your choice and print the results. For the streams of measurements, you can simply use a list with a few example `Measurement` objects (converted to a stream). Use only Java 8 for this question. Don’t use the MapReduce approach here (but see next question).

PTO

Question 3 [40 marks]

Add a further method `countTemperatures(t)` to class `WeatherStation` from the previous question. It should return the number of times where temperature `t` has been (approximately) measured so far by any of the weather stations in *`stations`* (“approximate” means `t` +/- 1). E.g., if there are two weather station objects in list *`stations`* and the first station has measured 20.0, 11.7, 5.4, 18.7 and the second one has measured 8.4, 19.2, 7.2, then `countTemperatures(19.0)` should return 3.
However, now use an “emulated” MapReduce approach to compute the result. Your program doesn’t need to perform real MapReduce (don’t use, e.g., Hadoop or multiple machines!), just try to create and apply MapReduce methods *map* and *reduce* so that your approach resembles the real MapReduce approach described in the lecture as closely as possible (including use of parallelism where appropriate). You can convert the stream of measurements into an ordinary list or array, if you like.

This question requires some knowledge from the MapReduce lecture next week.