Assignment 3 – Due Tuesday, October 25

Objectives: There are four objectives for this assignment:

- Learn how to create and use a Java `Comparator` class
- To explore different data structures and algorithms for finding the smallest elements in a dataset
- Compare experimental to theoretical running time results
- Write a brief experimental results report comparing algorithms

Background: Distributed with the initial code for this assignment is precinct-level voting data from the 2012 presidential election, covering 37 states and a total of 86,476 precincts, based on data from the Harvard Election Data Archive, `http://hdl.handle.net/1902.1/21919` (available under a Creative Commons license) that was cleaned up slightly for this assignment. The `vote2012.dat` file in the project's root directory contains this data, with each line listing a state, county, and precinct followed by the total number of votes for the Democratic candidate (Barack Obama) followed by the number of votes for the Republican candidate (Mitt Romney), with all data fields separated by semi-colons. The provided code reads in this file and creates an `ArrayList` of `PrecinctVotes` objects for you to work with. Note that not all lines have all fields, so some of the strings fields (like the "county" information) may be empty — this shouldn't affect your solution to this assignment.

Using n to represent the total number of precincts (so for this particular data, $n = 86,476$), your goal is to identify the m precincts (for some $m \leq n$ that is provided) with the narrowest margin of victory. You will test the following four algorithms for this problem, carefully comparing the performance of the algorithms — Algorithm 1 is implemented for you (it's just two lines after you create the necessary comparator!), but you'll need to implement the others:

Algorithm 1: Sorting all the data first. This algorithm sorts all n items, and then copies the smallest m items into a new list. This algorithm takes $\Theta(n \log n)$ time.

Algorithm 2: Keeping the smallest m items in an array. This option processes the data one element at a time, keeping track of the m smallest items in an array (you decide whether the data in the array should be kept in sorted order or not). This algorithm takes $\Theta(nm)$ time.

Algorithm 3: Keeping the smallest m items in a red-black tree. This algorithm uses a Java `TreeSet` to keep all items in a red-black tree. The first m items are simply added to the tree, and then once there are m items in the tree the algorithm can check in $O(\log m)$ time whether a new item should be inserted into the tree (removing the largest item so the size remains m) — to find the largest item in the tree (for comparison) you should use the `last()` method, and to remove the largest use `pollLast()`. This algorithm takes $\Theta(n \log m)$ time.

Algorithm 4: Keeping the smallest m items in a binary heap. For this algorithm, keep the data in a max heap so that you can quickly locate the maximum item out of the m values you are tracking. Use

the standard Java `PriorityQueue` class, but the `Comparator` should make a max heap! Once m items are put into a max heap, you can quickly locate the largest item (using method `peek()`) and remove it if necessary (using method `poll()`. This algorithm takes $\Theta(n \log m)$ time.

Every algorithm should return the m smallest values in a `List<PrecinctVotes>`, but the order in the list isn't important. Note that the asymptotic time used by the last two algorithms is the same, but the constants hidden by the asymptotic notation will be different. Which do you think will be the more efficient implementation?

What To Do: Start with the code in Bitbucket, as in previous assignments: fork the "Assign3" repository, rename it to include your username, grant read access to the class administrators, and then use NetBeans on your computer to clone it so you can work with it.

The first thing you should do is to create a `Comparator` class for `PrecinctVotes` objects. In order to sort `PrecinctVotes` objects or put them into any kind of ordered data structure, you will need to create a class that implements the `Comparator<PrecinctVotes>` interface. Implement this as a nested class inside the `PrecinctVotes` class so that the comparator can directly access the `PrecinctVotes` instance variables. You should implement this comparator in such a way that `PrecinctVotes` object `a` is less than `b` if the absolute value of the vote difference is smaller, or if the vote difference is the same and the state, county, and precinct (in that order) is smaller. Note that for one of the implementations, you'll actually need the reverse of this comparator — see the `reversed()` method in the `Comparator` class for an easy solution!

Next, implement the four algorithms described above, and test them thoroughly to make sure each one works correctly. Other than selecting the comparator *this should be completely generic* and independent of the `PrecinctVotes` class. Run your program with the profiler to time how long each implementation takes to process the precinct data, when the value of m is set to 100, 1000, 5000, 10000, and 20000. For each value of m, run it once in the profiler without recording times, and then run it three more times and average the times for each algorithm. Record the times in a table, like this:

	100	1000	5000	10,000	20,000
Array	21.1	101.1	511	1879	4574
Sort	88.7	91.2	84.5	85.6	81.6
Tree	38.0	59.3	76.2	89.2	102.4
Heap	27.4	37.9	50.2	60.3	66.1

Finally, write a brief report describing the results of your tests. Your report should have 3 sections: "Algorithms Tested" (in which you describe, in your own words, the algorithms that you implemented), "Experimental Results" (with the table of times and a brief explanation of what is in the table, such as time units), and finally "Comparison with Theoretical Model" (where you compare the time measured for $m = 20,000$ with an estimate you calculate from the asymptotic running time and the times with $m \leq 10000$). We will talk more about that last section in class.

Submission Instructions: For this assignment, you will have both code and an experiment report to turn in. For the code, do what you've done in previous assignments: commit all changes to your project and do a "push to upstream" to put the most up-to-date files on the Bitbucket server. You should turn in the report on paper in class.