CSCE 156 – Computer Science II

Lab 13.0 - Sorting

Prior to Lab

1. Review this laboratory handout prior to lab.
2. Review insertion sort and quick sort from the class notes or from the following resources:
    - http://en.wikipedia.org/wiki/Insertion_sort
    - http://en.wikipedia.org/wiki/Quick_sort
3. Familiarize yourself with sorting algorithms using the following site: http://www.sorting-algorithms.com/

Lab Objectives & Topics

Following the lab, you should be able to:

- Sort a list of objects by using the `Comparable<T>` interface and various sorting methods including the `Collections.sort()` method.
- Be familiar with Insertion Sort and Quick Sort algorithms and be able to adapt them to sort comparable objects.
- Compare and contrast the performance of various sorting methods
- Empirically measure the relative performance of algorithms with respect to the input size.

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

Sorting

Being able to organize and retrieve information in large datasets is a big research area with numerous applications. At the heart of any data mining endeavor is the fundamental operation of sorting.

For this lab, we have provided a file containing United States geographical data (specifically, a comprehensive list of 80264 geographical locations identified by 5-digit zip code, city, state and latitude/longitude). Functionality has already been provided for processing this data file and creating an array of `Location` objects to hold this data. Each time you run the `main()` method in the `SortingPerformance` class, a random selection of locations is loaded. The parameter, `n` limits the number of locations loaded from the file so that you'll be able to easily run experiments on different input sizes.

It will be your task to determine how this geographical data should be sorted, develop several sorting algorithms to actually sort the data, and empirically evaluate the running time of each method.

Comparable Interface

Numerical and string data types have a natural ordering that is understood by a computer language. However, user defined types such as the `Location` objects do not have an obvious ordering; it may be possible to order locations by an individual field (zip code, state, city) or a combination of those fields. Java provides a means for you to define exactly how instances of your class should be ordered by implementing the `Comparable<T>` interface.

If a class implements the `Comparable<T>` interface, it must provide an implementation for the following method.

```
public int compareTo(T item)
```

The general contract of this method is as follows:

- It returns a negative number if this object should precede the item object
- It returns zero if this object is "equivalent" to the item object
- It returns a positive number if this object should come after the item object

Sorting Algorithms

You will be comparing four different sorting algorithms: selection sort, insertion sort, quick sort, and the sorting method provided by the JDK (Java Developer's Kit, specifically the `Arrays.sort(Object[])` method). The selection sort algorithm and the method to call the JDK's sorting algorithm have already been provided for you in the SortingAlgorithms class. Refer to the selection sort method especially if you are still unclear on how to use the `compareTo()` method. You will need to adapt the insertion sort and quick sort algorithms from your text to sort `Location` objects in the given array. Be sure to sort and return a copy of the array, not the array itself.

Comparing Running Times

To compare how each sorting algorithm performs, you will setup an experiment and calculate how much time it actually takes to sort the array of locations by using the `System.nanoTime()` method. This method returns the current system time, more-or-less precise to a nanosecond (1 second = 1,000 milliseconds = 1,000,000 microseconds = 1,000,000,000 nanoseconds). By taking a snapshot of the system time before and after a method call, you can compute (roughly) the total elapsed time. As an example, consider the following code snippet.

```java
long start = System.nanoTime();
Location sorted[] = SortingAlgorithms.insertionSort(sp.getLocations());
long end = System.nanoTime();
long elapsedTimeNs = (end - start);
```

Activities

Clone the starter code for this lab from GitHub using the following url: https://github.com/cbourke/CSCE156-Lab13.

1. Implement the `compareTo()` method in the `Location` class.
2. Implement the `insertionSort()` and `quickSortRecursive()` methods in the `SortingAlgorithms` class. Do this by adapting the algorithms as presented in your text book.
3. Debug your code and verify that each sorting algorithm is correctly sorting by using the `printLocationArray()` method provided to you to output the array contents.
4. Perform several timed experimental runs of each of your algorithms on various array sizes as outlined in the worksheet

Advanced Activities (Optional)

1. Examine the experimental data for each of the sorting algorithms on the various input sizes. Under the assumption that we have 1 million records, make a prediction, based on the observed running times on how long each algorithm would take to execute. Find a (or generate a fake) fake dataset of 1 million entries and run your experiment. Do your predictions match the actual running time?
2. Implementing the `Comparable` interface means that `Location` objects can only be "naturally ordered" in one way. It is often more flexible to instead use `Comparator` class to enable a user to order objects in any order that they want while reusing your methods. Rewrite each of the sorting methods to instead use a `Comparator<Location>` instance passed in to the method.