10 Sorting out Sorting

In this problem set you will examine the properties of the different algorithms we have seen as well as see and design new ones. The goal is to learn to understand the tradeoffs between different ways of writing the same program, and to learn some techniques that can be used to explore the algorithm behavior.

Part 1: Sorting Algorithms

We have seen so far several different sorting algorithms. We have implemented selection sort for *ArrayList*, an insertion sort that consumed an iterator and produces either AListOfCities or ArrayList, and we have seen in the Assignment 5 that we can use the binary search tree to sort the given items. The algorithms are similar, yet they do not conform to the same interface.

Our first task is to design *wrappers* for all these algorithms that will allows us to use them interchangeably to sort any collection of data supplied through an iterator. Of course, we want all of them to produce the data in a uniform format as well. Therefore, we want all of these algorithms to produce an iterator for the sorted list.

For each of the algorithms listed below design a class that implements the *SortAlgorithm* interface, using the specified sorting algorithm.

The *SortAlgorithm* interface is defined as follows:

```
import java.util.Comparator;
interface SortAlgorithm {


// initialize the data to be consumed by the sort,
// perform the sorting algorithm,
// produce an iterator for the result
public IRange sort(IRange it, Comparator comp);
}
```

We provide an example of a class that implements a quicksort algorithm. This implementation swaps the items within the *ArrayList* without using additional space.

10.1 Problem

Design the method in the *Examples* class that determines whether the data generated by the given *IRange* iterator is sorted, with regard to the given *Comparator*. ∎

10.2 Problem

Design the class *SelectionArrSort* that performs the selection sort on an *ArrayList*. The *ArrayList* is initialized from the data supplied by the IRange iterator.

Include in the class a self test in the form of a method testSort() that provides a test for all methods in this class. Include the *main* method that invokes this test and run the test as well. ∎

10.3 Problem

Design the class *InsertionArrSort* that performs the insertion sort on an *ArrayList*. The *ArrayList* is initialized from the data supplied by the IRange iterator.

Include in the class a self test in the form of a method testSort() that provides a test for all methods in this class. Include the *main* method that invokes this test and run the test as well. ∎

10.4 Problem

Design the class *InsertionListSort* that performs the insertion sort on an *AListOfCities*. The *AListOfCities* is initialized from the data supplied by the IRange iterator.

Include in the class a self test in the form of a method testSort() that provides a test for all methods in this class. Include the *main* method that invokes this test and run the test as well. ∎

10.5 Problem

Design the class *BinaryTreeSort* that performs the binary tree sort on the data supplied by the IRange iterator provided as an argument to the method .

The *sort* method first builds the binary search tree from the data provided by the iterator, then saves the data generated by the *inorder* traversal in an *ArrayList* or in an *AListOfCities* data structure. The code you wrote for the Assignment 5 can easily be adapted to solve this problem.

Include in the class a self test in the form of a method testSort() that provides a test for all methods in this class. Include the *main* method that invokes this test and run the test as well. ∎

10.6 Problem

Design the class *QuickListSort* that performs the quicksort on an *AListOfCities*. The *AListOfCities* is initialized from the data supplied by the IRange iterator.

Include in the class a self test in the form of a method testSort() that provides a test for all methods in this class. Include the *main* method that invokes this test and run the test as well. ∎

Part 2: Time Trials

All of the tests we designed as the part of our code sorted only very small collections of data. It is important to make sure that the programs work well for large amounts of data as well. We have learned about the analytical way to estimate the amount of time an algorithm should take. However, we would like to verify these results on real data, and learn in the process what other issues we need to take into consideration (for example, the space the algorithm uses, and whether the data is already sorted or nearly sorted).

The class *TimerTests* provides a framework for conducting timing experiments. The constructor consumes an *IRange* and initializes an *ArrayList* with this data, so we do not have to read the file of 29470 items for every test. For most of our work, this data will come from the file **citiesdb.txt** that contains data for 29470 cities throughout the USA.

The class *TimerTests* also contains two methods that generate a dataset of the desired size from this initial database. One of them, *buildData*, just selects a random contiguous subsequence of the original data (preserving the original ordering by states) while the second one, *buildRandomData*, selects the elements of the new data set randomly from the original one — with possible repetitions.

Finally, the method *runOneTest* runs one test of a sorting algorithm. It consumes a sorting algorithm (an instance of *SortAlgorithm*), a *Comparator*, the size of the data set we wish to sort, and a boolean value that specifies whether we want a sequential subset of the original data, or randomly selected elements. It runs the sorting algorithm with a stopwatch and produces the timing result.

10.7 Problem

Design the classes that implement the Java *Comparator* interface and allow us to compare two cities by their zip codes (**class** *ByZip*) and by longitude (**class** *ByLongitude*). ∎

10.8 Problem

Design the class *Result* that holds the results of the timing tests. For each test we want to remember that the name of the test (for example "Insertion sort with ArrayList"), the size of the data that we sorted, whether it was sequentially or randomly selected data, and the time it took to run the algorithm.

Modify the method *runOneTest* in the class *TimerTests* so it produces an instance of *Result*.

Include the method *toString* in the class *Result* that produces a nicely formatted String that represents the result. ∎

10.9 Problem

Design the method *runAllTests* that consumes an *ArrayList* of instances of *SortAlgorithm*, an *ArrayList* of instances of *Comparator*s, and the size of the data, and runs the timing tests for each algorithm, using each of the comparators, using both, sequential and random data. The results should be produced as an *ArrayList* of *Result*s. ∎

10.10 Problem

Use the method *runAllTests* to learn about all these sorting algorithms. Present your findings in a report that describes what you learned from running these experiments.

You should run all algorithms with all combinations of comparators on large data (10000 or more items - if possible), explore how the performance varies between random data and the sequentially selected data.

If one of the algorithms takes too much time or space, you may eliminate it from further trials on larger datasets. However, try to understand why that may be hapenning.

You may also modify the way the dataset is initialized. You may want to see how your algorithm performs on sorted data, or you may want to test several algorithms with identical data.

Produce your results in a professionally designed format — possibly with charts. We care both about the results and about the way you present them and explain what you learned from them.