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 consumes an iterator and produces either AListOfCities, and we have seen that we can use the binary search tree to sort the given items it contains. 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.

Theabstract class *ASortAlgo* provides a uniform *wrapper* for all sorting algorithm. The *initData* method consumes the given iterator for the data to be sorted and saves the given data in a data structure appropriate for this algorithm. So, for example, the *initData* method in the *ArrSortSelection* class that defines a mutating selection sort that works with *ArrayList* data copies the data into an *ArrayList* that will be later sorted.

The abstract class *ASortAlgo* is defined as follows:

```
import java.util.Comparator;
abstract class ASortAlgo {

  public Comparator<City> comp;

  // initialize a data set with the data generated by the traversal
  abstract public void initData(Traversal<City> tr);

  // sort the data set with respect to the given comparator
  // produce a traversal for the sorted data
  abstract public Traversal<City> sort();
}
```

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

Here is a summary of the algorithms you will implement. Please, use the names given below:

- *ArrSortSelection*
- *ArrSortInsertion*
- *AListSortInsertion*
- *ABinaryTreeSort*
- *AListSortQuickSort*
- *ArrSortQuickSort*

10.1 Problem

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

10.2 Problem

Design the class *ArrSortInsertion* that that extends the *ASortAlgo* class. It performs the insertion sort on an *ArrayList*. The *ArrayList* is initialized from the data supplied by the Traversal 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. There are example of this technique in nearly all files provided with this homework. ∎

10.3 Problem

Design the class *AListSortInsertion* that that extends the *ASortAlgo* class. It performs the insertion sort by consuming a *Traversal* and producing an *AListOfCities*. Here you may not need to copy the data first, because the new *AListOfCities* is generated as we traverse over the original data.

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 *ABinaryTreeSort* that that extends the *ASortAlgo* class. It performs the binary tree sort on the data supplied by the *Traversal* iterator.

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.

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 *AListSortQuickSort* that performs the recursively defined quicksort on the data supplied by the *Traversal* iterator and producing an *AListOfCities* data structure. You will need a helper method to append two lists together.

HtDP has a good explanation of quicksort.

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 *ArrSortQuickSort* that that extends the *ASortAlgo* class. It performs the quicksort sort on an *ArrayList*. The *ArrayList* is initialized from the data supplied by the Traversal iterator.

You may use any textbook or the web to find an implementation of this algorithm, but you are responsible for the correctness of your implementation.

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. There are example of this technique in nearly all files provided with this homework. ∎

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. It is possible to estimate the amount of time an algorithm should take in comparison to others. 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).

Test Data

The class *DataSet* represents one set of data to be sorted. It knows the size of the data set, whether it is a sequential subset of the original data or a randomly selected set of data. It provides an iterator that generates for the sorting algorithm all elements in this data set.

The class *TestData* generates all *DataSet*s we will use, so that we do not have to repeat this process, and also to make sure that all algorithms will use sort the same data. This way we can conduct 'controlled' experiments — comparing outcomes when solving the same problem.

Timing Tests

The class *TimerTests* provides a framework for conducting timing experiments. It contains a field that is an instance of *TestData* so we do not have to read the file **citiesdb.txt** of 29470 items for every test.

Finally, the method *runOneTest* runs one test of a sorting algorithm. It consumes a sorting algorithm (an instance of *ASortAlgo*) and an instance of *DataSet*. These two pieces of data determine what is the data to be sorted, how large it is, whether it is random or sequential, which algorithm is used, and which comparator is used. 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** *ComparatorByZip*) and by longitude (**class** *ComparatorByLongitude*).∎

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 instance of *TestData*, 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 the data in the*TestData* class, and 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.