10 Sorting, Performance/Stress Tests

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.

Etudes: Sorting Algorithms

To practice working with the *ArrayList* start by implementing the ***selection sort***. Because we will work with several different sorting algorithms, yet want all of them to *look the same* the selection sort will be a method in the class that extends the abstract class *ASortAlgo*. A sorting algorithms needs to be given the data to sort, the *Comparator* to use to determine the ordering, and needs to provide an access to the data it produces. Additionally, the *ASort* class also includes a field that represents the name of the implementing sorting algorithm.

```
abstract class ASortAlgo<T> {
  /** the comparator that determines the ordering of the elements */
  public Comparator<T> comp;
  /** the name of this sorting algorithm */
  public String algoName;

  /**
   * Initialize a data set to be sorted
   * with the data generated by the traversal.
   * @param the given traversal
   */
  abstract public void initData(Traversal<T> tr);

  /**
   * Sort the data set with respect to the given Comparator
   * and produce a traversal for the sorted data.
   * @return the traversal for the sorted data
   */
  abstract public Traversal<T> sort();
}
```

1. Design the class *ArrSortSelection* that implements a selection sort and extends the class *ASortAlgo* using an *ArrayList* as the dataset to sort. The *ArrayList* becomes an additional field in this class.

   - The *initData* initializes the *ArrayList* by adding to it all the data generated by the given *Traversal*.
   - the new method *selectionSort* implements a mutating selection sort on this *ArrayList*. There are numerous lecture notes and labs from the past several years available online that guide you through the design of *selection sort*. The *third part* of you draft textbook has a section on selection sort on pages 102-112.
   - The *sort* method invokes a selection sort on this *ArrayList* and returns an instance of *ALT* the *Traversal* over the data in an *ArrayList*.
   - *Do this once you include the code in the main part of your assignment.*

     Include in the class a self test in the form of a method testSort() that provides a test for all methods in this class. There is an example of this technique in the *AListSortInsertion* class provided with this homework.

2. Design the class *ArrSortInsertion* that implements a mutating insertion sort for an *ArrayList* and extends the class *ASortAlgo* in that same way as was done for the selection sort.

You may work on this part with your partner if you wish — or get any help you need. However, you must make sure that at the end you understand every part of the code and how the tests were designed.

Sorting out Sorting

We have seen by now three different ways to sort a collection of data: The insertion sort, the quicksort, and the binary tree sort.

The binary tree sort was *hidden* — we first designed the method to insert data into a binary search tree, then we designed the traversal over the binary search tree that generated the elements in the desired order.

The algorithms all have the same purpose, yet they do not conform to the same interface. They also deal with the data structured in different ways. However, we would like to compare these algorithms for efficiency (speed).

ASortAlgo class: Reporting the results

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 a traversal for the sorted list.

Because the traversal of a binary search tree involes quite a lot of computation, to make to comparisons fari, we will conclude the binary tree sort by producing an sorted *ArrayList*, generated by the data produced by the traversal of the binary search tree we have constructed.

ASortAlgo class: Initializing the data

The abstract class *ASortAlgo* introduced in the *Etudes* provides a uniform *wrapper* for all sorting algorithms. The *initData* method consumes the given traversal for the data to be sorted and saves the given data in a data structure appropriate for this algorithm.

For the algorithms that are based on the *ArrayList* the *initData* method creates a new *ArrayList* and adds all elements generated by the given *Traversal*. For the *AListSortInsertion* class that implements the insertion sort for a recursively built *AList*, the *initData* method copied the given data into a new *AList*. For the quicksort algorithm that is based on recuesively build lists, this will not be necessary - as the algorithm traverses over the data and constructs two new lists - the upper and the lower partition, and does not use any properties of the lists. Finally, for the binary tree sort the task of constructing the binary tree comprises a substantial amount of work, and so should be done as a part of the *sort* method. Therefore, the *initData* method just copies the reference to the *Traversal*.

We provide an example of a class that implements the insertion sort algorithm for data saved as *AList<T>*.

ASortAlgo class: Running the timing tests

To measure the time each algorithm takes to complete its task, we will invoke each algorithm in three parts. First, we provide the data to sort and invoke the *initData* method. Next we start the timer, invoke the *sort* method and stop the timer when the method returns the result. Finally, we check that the data was indeed sorted and record the result.

We will test each of the several sorting algorithms on several datasets (varying the dataset size and whether the data is in the original order as received from the database, or is *randomized*). For each dataset we will use several different *Comparator*s — by name and state, by latitude, and by zip code.

The results will be recorded in a uniform way, so that we may then look for patterns of behavior.

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

- *ArrSortSelection* — done as a Part 1 of the Etudes
- *ArrSortInsertion* — done as Part 2 of the Etudes
- *AListSortInsertion* — provided
- *ABinaryTreeSort* — modify the solution to homework 5
- *AListSortQuickSort* — to be done (modify 8.1 part 9 of homework 8)
- *ArrSortQuickSort* — to be done

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*.

Later, you will need the same method in the class *TimerTests*.∎

10.2 Problem

Complete the design the classes *AListSortSelection* and *AListSortInsertion*.

Include in each class a self test in the form of a method *testSort*(*Tests tests*) that provides a test for all methods in this class.∎

10.3 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 each class a self test in the form of a method *testSort*(*Tests tests*) that provides a test for all methods in this class.∎

10.4 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 each class a self test in the form of a method *testSort*(*Tests tests*) that provides a test for all methods in this class.∎

10.5 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 each class a self test in the form of a method *testSort*(*Tests tests*) that provides a test for all methods in this class.

Note: If you are having problems with this algorithm, go on to the *Time Trials* and finish this only if you have time left.∎

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.

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.

Finally, the method *allTests* runs timing tests for all the selected combinations of sorting algorithms, comparators, and datasets and collects the results into an *ArrayList* of *Result*s.

10.6 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*).∎

10.7 Problem

Study the design of 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.

The method *runATest* in the class *TimerTests* modifies the method *runOneTest* to produce an instance of the *Result*.

Modify the method *toString* in the class *Result* to produce a *String* that represents the result formatted the way you would like to see the results. ∎

10.8 Problem

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 are produced as an *ArrayList* of *Result*s.

Run all possible tests. Make a note of those that fail. Choose a few at a time (for example all algorithms and comparators for a given dataset) and record the results into a text file or into a spreadsheet. ∎

10.9 Problem

Look at the results of the timing tests and see what you can learn about the different ways to sort 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. For example, you may want to see how your algorithm performs on sorted data.

Present both the numerical results, and your comments about the meaning of the 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.∎