Algorithm Complexity: Stress Tests

Download the provided zip file *Lab12-sorting.zip* and unzip it. Create a new *Eclipse* project named *Lab12-sorting*. Add the given code to the project. You should have the following Java files:

- class *Examples* defines and runs all the tests for lists of random integers.
- class *Algorithms* implements the imperative insertion sort and the quicksort using the *ArrayList<T>*, and the functional insertion sort using the *AList<T>* classes defined in the same file.
- class *IntComp* implements the *Comparator* for integers.
- class *StringComp* implements the *Comparator* for *String*s, based on their length.
- class *Sorter* is a wrapper that enables us to print the timing results neatly. The file includes two implementations, *Quick* and *Insert*. It also includes a separate but similar class *InsertList* that wraps the insertion sort for *AList* classes and includes two methods *fillList* for first converting the *List* data to *AList* and *result* that converts the sorting result saved as *AList* to an *ArrayList*.
- class *Timing* provides a simple way to interact with the system clock.
- class *StringExamples* defines and runs all the tests for lists of random *String*s.

For this section of the lab we are going to quickly explore the differences between $O(n^2)$ and average $O(nlogn)$ sorting algorithms.

Insertion Sort:

As mentioned in class, the running time of insertion sort is approximately $O((n*(n+1))/4) = O(n^2)$. This is because in order to insert each element into the sorted portion of the *List* we must compare $k/2$ items on average, where k is the size of the sorted portion.

In the *Algorithms* class you can see an implementation of *Insertion Sort* which sorts an *ArrayList<T>* in-place.

Quick Sort:

This algorithm is considered one of the best in-place sorting algorithms because it is easy to implement and runs pretty fast. Have a look at the implementation in the *Algorithms* class.

Your Task

If you try to run the *Examples* class you will notice there is a *RuntimeException* that's thrown. This is because there is a missing implementation. As further practice with *Comparator*s, you need to implement the *IntComp* class which compares two *Integers* using available functions.

You must then add a new instance of your class to the *Examples* main method (see where the *null* is?) so that the sorting tests will work.

Once you have implemented the class and created an instance, run the *Examples* class to see what it produces. **Check the output to see if it is indeed sorted... if not you will need to fix your comparator!**

When the sorts work correctly, run the *Examples* class again, but this time modify the source to run 3 or 4 timed sort tests by changing the variable loops appropriately. Note the loop which uses this variable.

Now run the similar code in the class *StringExamples*. It mimics the *Examples* class, but deals with lists of random *String*s. The sorting algorithms and their wrappers use generic types and require no changes.

Results

You should get some reasonable differences between the times of *Insertion* and *Quick Sort* even on these smaller *ArrayList*s.

Sketch a plot of your results on a piece of paper and observe the difference between the slopes of the plots for the two insertion sorts and the plot for the quicksort.

Look over the interesting portions of the supplied code:

- *static* and *Generic* methods in the *Algorithms* class
- The *fillData*(...) method in the *Examples* class... try to understand what's going on there
- The abstract class *Sorter* and its implementations that wrap calls to the *Algorithms* code (remember the Function Objects?) and the methods which use them in the *Example* class.

- And check out the *Timing* for a way to query the *System* for accurate time counts and what we can do with them.

Visitors

In this section you will get a chance to work with the code from yesterday's lecture on the *Visitor*s.

Download the provided zip file *Lab12-visitors.zip* and unzip it. Create a new *Eclipse* project named *Lab12-visitors*. Add the given code to the project. You should have the following Java files: *PieMan.java* and *ListMan.java*. The file *ListMan.java* contains a similar code, but written in the contest of our standard *AList* classes. You may find it easier to read that code first.

- The file *PieMan.java* defines the code we covered in class yesterday and adds test code. There are no comments anywhere. However, the class diagram may help. Read the code and add purpose statements to the classes and methods.
- Run the program, using the *tester* library for the testing support.
- Look at the definitions of the test cases and make sure you understand how the code is organized.
- The interface *IPieMan* has a method *boolean containsTop*(*Object o*) commented out. The purpose is to determine whether the pie contains the given topping. Make a list of all that you have to do to implement this method.
- Make examples of the use of this method in the *Examples* class.
- Add all methods and classes needed to implement this method correctly.
- If you have time left look at the implementation of the same methods in the context of a recursively defined *AList* class hierarchy and add the *contains* method there as well.