Computer Science 62
Lab 2

Wednesday, February 10, 2009

In this lab, we'll be playing with some of the sorting algorithms we've discusses in class. In addition, you'll get some familiarity with the `merge` method of `MergeSort`, which you'll be implementing an on-disk version of for the next assignment.

1 Getting started

Create a new project in `Eclipse` and then go to `Terminal` and copy over all the ".java" the files from:

```
/common/cs/cs062/labs/lab3
```

To tell the "cp" command that you only want ".java" files, append `/*.java` after the directory name. '*' is a wildcard that says match anything, so you will get all files in the above directory that end in ".java".

After you've copied the code, spend 5 minutes looking at the different classes. In particular,

- Look at the interface
- Look at how the `Quicksort` and `MergeSort` classes implement the interface
- Look at how the `SortTimer` class is able to print out data for an arbitrary number of `Sorter` classes (this is the benefit of using an interface!)

- Notice that the `SortTimer` class does a check for correctness after sorting. If you make a mistake in implementing your `merge` method, you'll get an error here.

2 Finish `MergeSort`

I've given you all of the code for this lab except the `merge` method, which you should now implement. Give it a good effort, but if you get stuck, I've provided a solution below. It will benefit you to figure it out in the lab, though, while you have help from me (i.e. without looking) since you will be implementing something similar for your assignment.

Once this is done, you should be able to run the `SortTimer` class.

3 Play with the timing

Run the `SortTimer` class. Do the data look like you'd expect? Which one is faster?

This should give you some confidence that `Quicksort` average case works as we expect. As an additional test, change the `printTimes` method to generate sorted data instead of random data. How does this change your timing data? Is this what you expected?

4 Playing with the sorting algorithms

Open a terminal window and type the following command:

```
/common/cs/cs062/labs/lab3/bin/coinsorter
```

You will see a window similar to the one for the Silver Dollar Game, except that all the squares are filled, and the coins have different sizes. Use the keystrokes below to shuffle and sort the coins. Experiment with several of the sorting algorithms.

`b`: sort the coins using bubble sort

`c`: sort the coins using a randomly-selected algorithm

`i`: sort the coins using insertion sort

`q`: sort the coins using quicksort

`r`: rearrange the coins into a random order

`s`: sort the coins using selection sort

`x`: exit the program

The program you are using has a few additional features. Typing `f` (for "freeze") stops the sorting; typing `t` (for "thaw") resumes the sorting. Typing `f` when the sorting is frozen advances the algorithm by one step. You can continue to type `f` to proceed step-by-step, or `t` to resume normal execution.

Typing `c` selects one of the sorting algorithms at random and executes it. Practice with the `c` command to develop your skill in identifying the algorithm from the pattern of comparisons and swaps.

5 If you still have time...

Implement a new class for one of the $O(n^2)$ running time sorting methods that `extends` our `Sorter` interface. Add this new class into the `SortTimer` class and compare its runtime to the other sorting methods.

An implementation of `merge`. Here is one implementation of the `merge` algorithm. It uses an extra ArrayList, and so mergesort does not sort "in place" as our other algorithms do.

```
public void merge(ArrayList<E> list, int low, int mid, int high){
   Object[] temp = new Object[high-low];

   int tempIndex = 0;
   int lowIndex = low;
   int midIndex = mid;

   while( lowIndex < mid &&
              midIndex < high ){

      if( list.get(lowIndex).compareTo(list.get(midIndex)) < 1 ){
         temp[tempIndex] = list.get(lowIndex);
         lowIndex++;
      }else{
         temp[tempIndex] = list.get(midIndex);
         midIndex++;
      }

      tempIndex++;
   }

   // copy over the remaining data on the low to mid side if there
   // is some remaining.
   while( lowIndex < mid ){
      temp[tempIndex] = list.get(lowIndex);
      tempIndex++;
      lowIndex++;
   }

   // copy over the remaining data on the mid to high side if there
   // is some remaining.  Only one of these two while loops should
   // actually execute
   while( midIndex < high ){
      temp[tempIndex] = list.get(midIndex);
      tempIndex++;
      midIndex++;
   }

   // copy the data back from temp to list
   for( int i = 0; i < temp.length; i++ ){
      list.set(i+low, (E)temp[i]);
   }
}
```