CS61B, Fall 2011 **HW #6** **P. N. Hilfinger**

Due: Mon., 17 October 2011

Create a directory to hold your answers. There is a skeleton for your solutions in the repository under the URL `$STAFFREPOS/hw6`, and also in the directory `~cs61b/code/hw6`. Put non-program answers in a file hw6.txt. Use the usual command sequence to copy your final solution to a `hw6-`N entry in your tags repository directory.

1. At first glance, it seems that the most reasonable way to estimate the time an algorithm takes is to measure it. Each computer has an internal clock that keeps track of time (usually the number of milliseconds or microseconds that have elapsed since a given base date) and language libraries provide access to the clock. The Java method that accesses the clock is System.currentTimeMillis. We've provided a ucb.util.Stopwatch class as part of the standard class setup in the lab. (Those of you at home should download `~cs61b/lib/ucb.jar` and put it in your Java system's classpath. The sources for the ucb package is in `~cs61b/src/java/classes/ucb`. Both of these are in `$STAFFREPOS/software`.) Read the on-line documentation for `System.currentTimeMillis` and for `ucb.util.StopWatch` (see `Tool Documentation`⇒`UCB Package Documentation` on the class homepage.)

The file Sorter.java in the directory contains a version of the insertion sort algorithm, which sorts a sequence of items by moving each one in turn backwards through the sequence until it is in the right position relative to previously sorted items. Its main method uses a command-line argument to determine how many items to sort, and to provide an optional seed with which to vary the contents of the random array it sorts (the same seed gives the same sequence). It fills an array of the specified size with randomly generated values, starts a timer, sorts the array, and prints the elapsed time when the sorting is finished. For example, the shell command sequence

```
java Sorter 100000
```

generates 100000 random doubles (in the range 0–1) and sorts them, printing the wallclock time elapsed during sorting.

Compile Sorter and try it for a variety of sizes. The file hw6.txt, included in the hw6 directory, contains some tables asking for approximate parameters in formulae for computing approximate running times. The tables assumes that the time cost of Sorter is a function of N, the size of the array sorted, and has the form

$$aN^2 + bN + c$$

for some constants a, b, and c. Try to come up with approximate values for these constants (there are "official" ways to do this sort of fitting, but we'll be happy with some rough numbers). Put the results in the "Sorter" line of Table I.

Now try the same exercise with the C function csort (use the command '`gmake`' to compile csort from csort.c; it will use the gcc compiler to do so). Put the results in the csort (-O2) line of Table I.

Do the same with the program Sorter2.java, which is the same as Sorter, but uses an ArrayList instead of an array. Add the results to Table I.

The Makefile compiles csort with an optimization option (-O2) that asks the compiler to try to compile for speed. To see the effect of this parameter, first erase the csort program (rm csort) and then recompile with

```
gmake CFLAGS=
```

(just as written). Now determine its parameters and fill in the last line of Table I.

2. Repeat the same procedure, but on a machine with an entirely different architecture. Be sure you have recorded which architecture you were using for Table I in the space provided. Login (via ssh would be most convenient) to a machine with a completely different architecture and repeat the previous experiments again, filling in Table II. The Unix command 'uname -m' will tell you what kind of machine you are running on. In the instructional labs, this will be either i86pc (an Intel architecture), sun4u (Sparc), or x86_64 (a 64-bit Intel architecture, such as the 'hive' servers). See the List of Instructional Login Servers on the class home page to see what machines are available in each architecture. You must recompile the C program before running on the new machine. The two commands

```
gmake clean
gmake
```

will do so.

3. Now fill in Table III in hw6.txt, but this time, use commands of the form

```
java Sorter <SIZE> B
./csort <SIZE> B
etc.
```

The 'B' parameter causes the programs to use a different method to generate the data to be sorted.

Try looking at the algorithm and see if you can explain the differences between the results in Table III and those in Table I.

4. Now consider the program `Sorter3.java` in hw6 files. It's the same as `Sorter2.java`, except that it uses a LinkedList rather than an ArrayList. What effect do you expect this to have on the running time of `Sorter3` and why? (Try to answer this question without empirical measurement, giving a $\Theta(\cdot)$ estimate.)

5. You can speed the `Sorter3.java` program up considerably by using `ListIterator` and its operations, rather than the `.get` and `.set` methods on `LinkedLists`. Indeed, you should be able to make its speed comparable to that of `Sorter2`. Modify `Sorter3.java` to accomplish this.

6. [Goodrich & Tamassia] Suppose that A is an $n \times n$ array of 1's and 0's with the property that all the 1's in a row come before all the 0's in that row. The array is huge ($n > 500000$), but instead of actually being stored as a Java array, it is represented by a `BitMatrix` object with a method `.get(`i, j`)`, which returns A_{ij} (i is the row, j the column). Fill in the method `mostOnes(A)` in the template file `BigMat.java` so that it returns the index of the row of A that contains the most 1's. When several rows contain the largest number of 1's, return the smaller index. Your method must operate in $O(n)$ time (*not* $O(n^2)$ time). Your program will be given a time limit that requires it to operate in better than $O(n^2)$ time.