CS2223, Project 1
Intro to Lua:
Measuring and Counting Sorting Behaviors*

Due midnight, Monday, 5 Nov 2012

Download and install Lua version 5.1 or 5.2 on some convenient machine. Pre-compiled binaries for various systems are available at `http://lua-users.org/wiki/LuaBinaries`. The most recent build of Lua 5.1 is build 4, called 5.1.4, and 5.2 is at build 1.

For documentation, see `http://www.lua.org/docs.html`. There is an online manual at `http://www.lua.org/manual/5.1/`, and I've bought three copies of the published book version to put on reserve in the library.

The book *Programming in Lua* is good, and not expensive in printed form. The free online version at `http://www.lua.org/pil/` describes Lua 5.0 rather than 5.1, but the differences are small. I think you will find it very usable.

In fact, for our purposes, the only really important addition in Lua 5.2 or 5.1 versus 5.0 is the operator `#`, which gives the length of an array without gaps. If `a` is an array with entries for all the numbers $1, \ldots, k$, and no entry for $k+1$, then `#a` returns the number k. 5.1 vs. 5.2 makes no difference for us.

Windows users may want to use the *Lua for Windows* install from `http://code.google.com/p/luaforwindows/`, which includes an editor (Scite) and a way to run files. Macintosh users may want to use the text editor TextWrangler, freely available at `http://www.barebones.com/products/TextWrangler/`. Under the menu item `#!`, it has a dialog box to start Lua applied to a file. It runs the commands in the file, and prints any output. If there's no output, it looks like nothing happened, so be sure to put some output-producing commands at the end of the file.

You can give Lua interactive commands by starting the program `lua`. A command `function_name(args)` will apply the function to its arguments. The command

```
print(function_name(args))
```

will apply it and then also print the result. The command

*Joshua Guttman, FL 137, `mailto:guttman@wpi.edu`. Include [cs2223] in the Subject: header of email messages.

```
dofile("filename.lua")
```

executes the declarations and commands contained within the file with that name in the current directory, adding the results inside the current Lua session.

You can also execute a file as a unit (at least on Unix-family systems) by inserting text such as

```
#!/usr/local/bin/lua -i
```

as the *first* line (substituting the path that leads to the Lua interpreter on your system if it is different). The `-i` flag at the end causes it to start an interactive session after executing the file contents.

Project goals. The goals of this homework are for you to familiarize yourself with Lua, and with three sorting methods—*insertion* sort, *bubble* sort, and *merge* sort—and to evaluate the amount of work they need to do to successfully sort arrays of numbers.

Work with the Lua code at URL `http://web.cs.wpi.edu/~cs2223/b12/proj1/proj1.zip`. Your job is to modify this code to evaluate what it does when it runs on different inputs, to do the experiments, and to graph and interpret the results.

The Lua constructs you will use include local variables; array access and array modification; if-then-else and for-loops; and very little else. It is designed to let you focus on numbers and arrays and the core of Lua that manipulates them.

Bug reports. Please email `cs2223-staff@cs.wpi.edu` with any bug reports on this code. A bug report requires a brief explanation of what part of the code caused the problem, and why it's doing the wrong thing. Be sure to report the test case that exercised the bug.

Bounty for bug-finders: The first student to report each bug that makes any part of this code compute a wrong result wins 10 points extra credit (on the 100 point per-homework score).

Utility File. In the zip archive there is a file named `utilities.lua` containing a lot of useful functions. Read through it. Some examples are:

`copy_table` is useful to make a copy of an array so that the original will not be lost when procedures modify the copy.

`print_array` prints out arrays, at least when the system knows how to print the individual array entries.

`random_int_array` selects numbers randomly and inserts them into an array. Its first argument is the length of the array to generate, and its second argument is the largest number to permit into the array. You can always use 1,000,000 if you would like.

`run_with_time` takes two arguments, a function and an argument for that function. It checks the time, then applies the function to its argument, checks the time when it terminates, and returns both the return value of the function and the elapsed time.

`run_with_time_and_collector` is similar, but it also takes a *collector* (see next item) as an argument. It also returns the count in the collector when it completes.

`make_collector` is an *object* with a field that keeps an integer counter. It reveals the value in this counter when its `reveal` method is called. It re-initializes its counter to 0 when its `reset` method is called. And every time its `inc` method is called, it increments the counter. The collector can be used to count events that are occurring in the code, by putting a call to `collector.inc()` next to every event to count in the code.

`collector.inc()` returns `true`, so you can use it to count evaluations of a boolean expression `e`, by writing `collector.inc() and e`.

Project activities. There are three main activities for this project (after getting Lua to a workable state in your system). The activities will be graded on the basis of 100 points. The activities are:

1. Write two small utility functions on arrays. One will check that array is sorted. The other will check that two arrays have the same entries.

   You can use the first to check that a sorting routine has in fact delivered sorted result. You can use the second to check that two sorting routines have given the same result when applied to the same argument.

   The sortedness checker should use a `for` loop to inspect each pair of adjacent entries. If any is in the wrong order, it should return false.

   The array entry equality checker will first check that its two arguments have the same length. If they do not, it returns false. Otherwise, it uses a `for` loop to inspect the array values of each array for each index. If the arrays differ for any index, it should return false. It does not have to report the index.

   Add your commented code to the file proj1.lua, together with the example arrays that you used to test that your functions gave the right answer. (20 points.)

2. You will count the number of operations that the different sort algorithms require, when applied to various arguments. In particular, we will count *comparisons* between array elements, to determine whether they are out of order.

   Insert code into `proj1.lua` to count how many times comparison operators are applied to *array elements*. Do not count other uses of $<, \leq$, etc., just the ones that are applied to values in the arrays being sorted. You may

use a collector as defined in `util.lua`, or else you may use global variables and assignment statements.

Write a test procedure that will generate 100 test arrays, of sizes increasing from 50 entries to 5,000 entries. This will take a few minutes to run of most computers. For each array, make a copy of it to sort using insertion sort; another copy to sort using bubble sort, and a copy to sort using merge sort. For each array and sorting algorithm, maintain two arrays. One array should have the elapsed times as entries. The i^{th} entry in the insertion sort timing array should be the time it took to run insertion sort on the i^{th} array. The i^{th} entry in the merge sort count array should be the number of comparisons that merge sort required to sort the i^{th} array. Etc.

You will use this information to answer four questions:

(a) For each algorithm, how stable is the relation between time used and number of comparisons? Do you have a fairly constant ratio of comparisons per unit time?

(b) For each of the three algorithms (bubble sort, insertion sort, and merge sort), how does the number of comparisons increase as the array size increases?

(c) For a particular (large) array size, what are the ratios of the numbers of comparisons that the different algorithms perform, as we look at the different pairs of the three algorithms?

(d) Are there specific kinds of inputs for which an algorithm will perform particularly efficiently? For instance, inputs that are already in the right order, or nearly? Inputs that are almost completely wrong, namely the opposite of the right order? Randomly chosen input? For this question, you should also generate some specially designed arrays, and record the statistics for those.

Real world sorting applications often have special kinds of inputs.

Insert into the file your re-definitions of the sort routines with extra code to do the comparison counting. Make sure that your test procedures call the procedures being tested repeatedly with different arrays as the test arguments. The test procedures should maintain arrays that record the numbers of comparisons for these different test arguments.

Include these output results, and also your conclusions, as comments in the code.

It's convenient for this that Lua has "block comments" that look like this:

```
--[[

    This is a block comment.  You can include your
    output from the test in this form.
```

```
--]]
```

3. Now compare the timings for the three versions, namely the three different algorithms. You can get good, accurate timings in Lua using `os.clock()`. That gives a floating point number that says how many seconds of CPU time Lua has used since being started. You can find how much work some activity requires using the code:

```
local start_time = os.clock()

... execute activity here ...

local end_time = os.clock()
return end_time-start_time
```

   Report the same kinds of information as for the comparison counting.