```
                            CS 61B Lab 12
                          April 17-18, 2014
```

 Goal: This lab wil introduce you to several sorting algorithms and allow you
 to compare their running times in practice.

 Copy the Lab 12 directory by starting from your home directory and typing:

   cp -r ~cs61b/lab/lab12 .

 Part I: Identifying Sort Algorithms (1 point)

 To help build your intuition of how these sorting algorithms work, take a look
 at three algorithms in action. Run a web browser, make sure its has Java
 enabled, and go to the Web page

    http://www.cs.berkeley.edu/~jrs/61b/lab/MysterySort

 Each histogram shown illustrates an array of ints. The length of each bar is
 proportional to the integer stored at that position in the array. When you
 click on a histogram, an animated algorithm sorts the array.

 The histograms are numbered 1, 2, 3, and 4, each number corresponding to a
 different sorting algorithm you have seen in class. Identify each algorithm.
 How can you tell which is which? Which is fastest on the random data? What
 about the presorted data and the reverse sorted data?

 One of the algorithms is quicksort. Which element does it choose for the pivot
 in a partition? (The pivot is not randomly chosen.) How can you tell?

 Part II: Timing Sort Algorithms (2 points)

 Part I gave you a rough idea of the relative speeds of the three sorting
 algorithms. For greater accuracy, we will now time several algorithms.

 SortPerf.java runs a sorting algorithm (chosen from a selection of several) on
 a randomly generated array of specified size, and outputs the data to a file.
 Compile and run SortPerf.

   javac -g SortPerf.java
   java SortPerf select 5 175 200 select.dat

 The first argument is the sorting algorithm ("select", "merge", "insert", or
 "quick"). The second is both the size of the smallest array to sort and the
 increment between array sizes. The third is the size of the largest array to
 sort. The fourth is the number of trials for each size of array. Because
 timings are easily perturbed by other processes running on the same
 workstation, you must run each test for a few hundred trials to get accurate
 timings. The last argument is the name of the file in which SortPerf reports
 the total running time, in milliseconds, of all trials for each array size.

 The sample command above runs selection sort on arrays of size 5, 10, 15, ...,
 175, each 200 times, placing the timing results in the file select.dat. If you
 notice that the timings look a little bumpy, i.e., the running time does not
 appear to be strictly increasing with input size, close any unnecessary
 applications you have running, then try running SortPerf several times until
 you get a fairly smooth set of numbers. When SortPerf is running, be careful
 not to touch the mouse or keyboard, as any user input increases the running
 time.

 Now run SortPerf for the same sizes using mergesort, insertion sort, and
 quicksort. Save these data to different files--say, merge.dat, insert.dat, and
 quick.dat. To make this easier, we have also provided a file called runrace,
 which you can execute to run all four sorting algorithms. Read runrace to see

  how it works.

 Use the program "gnuplot" to plot the timing results. At the prompt in an
 xterm (not emacs), type

   gnuplot

 You will see a ">" prompt. Plot the files using a command like

   gnuplot> plot "select.dat" with linesp 1, "merge.dat" with linesp 2,
             "insert.dat" with linesp 3, "quick.dat" with linesp 4

   [Type all this on just one line, though.]

 You should observe that the algorithms behave asymptotically as expected, but
 it is unclear which algorithm is faster for small array sizes. At what values
 of n do you observe cross-overs in execution time between the three sorting
 algorithms? To help see these cross-overs better, edit the runrace file to use
 the command

   java SortPerf <sort> 1 50 1000 <sort>.dat

 for each sorting algorithm. This gives a clearer picture of what happens for
 array sizes less than 50. Again run the script and plot the results with
 gnuplot.

 PART III: Building a Better Sort (1 point)

 Based on your observations in Part II, write a sorting algorithm that works
 well on the random data for all sizes of n. It won't be a completely new
 sorting algorithm, but should combine the sorting algorithms described here in
 a very simple way. Implement your solution in YourSort.java. Run your
 algorithm from SortPerf by calling the sorting method "best". Modify runrace
 to generate timings for your new algorithm, and plot the results.

 Check-off
 ----
 1 point: What are sorting algorithms 1, 2, and 3 in Part I? How do you know?
            How is the pivot element chosen? How do you know?
 3 points: Show the plotted performance of the sorting algorithms. The third
            point is for including your algorithm in the plot, and demonstrating
            that it's roughly as good as any of the others for any array size.