Sorting Out Quicksort

sort /sôrt/
verb
gerund or present participle: sorting
1. arrange systematically in groups; separate according to type, class, etc.
"the mail was sorted"

2. (INFORMAL) resolve (a problem or difficulty).
"the problem with the engine was soon sorted"

This homework requires you to implement insertion sorting and quicksort functions (methods) and associated code to test and analyze the sorting routines on various data files. As always for our programming assignments, you can write your code in C++, Java, or Python 3. You will use system calls to print out the time required by sections of your code; we'll provide examples and guidance on how to do this. Your version of quicksort will use the "tuning" discussed in class where small lists are not sorted recursively with quicksort but instead are sorted with insertion sort.

Part of this homework will be like a lab exercise or experiment where you'll run your code on different data files, study the time taken by your sorting algorithm, and answer some questions. You'lll submit a PDF file with the answers to those questions. But you will also submit your code to GradeScope where it will be run against test-cases to show that your sorting methods work correctly. (GradeScope will not test for efficiency or time-complexity.)

What You Will Code

Here are instructions for what we want you to code.

- Write a function `insertionsort()` that takes 3 parameters: a list, a *start* index, and an *end* index. The function will sort the portion of the list from index *start* through *end* (inclusive) using insertion sort. The list parameter will be a Python list, a Java List or a C++ vector., and it will store integer values. The function will not return anything (return type if *void* or the equivalent). List elements will be sorted in non-descending order. Lists in these languages are zero-indexed, so if *start* is 0 then sorting starts at the beginning of the list. If *end* is -1 then sorting is done to the last element of the list.

- Write a function `is_sorted()` that you will use to to check your sorting method by verifying that the portion of the between index *start* and *end* are correctly sorted. It takes 3 parameters: a list, a *start* index, and an *end* index, that work as described above. It will print "yes" or "no" to the console to show if the list is sorted or not, and will return a boolen value of *true* if the list is sorted.

- Write a function `quicksort()` that takes 4 parameters: a list, a *start* index, an *end* index, and an int value *minsize.* This function will work like the insertion sort function except that it will use quicksort. (See next item for details about partitioning.) The first 3 parameters serve the same role as for the insertion sort function. The 4th parameter is used to control what quicksort does when it's processing a small list, that is one that has $\leq$ *minsize* elements. If *minsize* > 1 and the size of the sub-list between *start* and *end* is $\leq$ *minsize*, then that sub-list is sorted with your insertion sort function. If *minsize* ≤ 1 or the size of the sublist is $>$ *minsize*, then insertion sort is not used on the small sub-list.

Observe that passing the function a *minsize* value ≤ 1 means that your quicksort will not use insertion sort at all. We'll call this *"pure" quicksort*, and when insertion sort is used on small lists we'll call this *"hybrid" sorting.*

- Your quicksort function should call a separate `partition()` function. You may choose Lomuto's or Hoare's algortihm (both covered in the text book); put a comment at the start of your function stating which you're using. Your code should randomly choose an element in the sub-list and swap it into the right position for use as the partition element, as discussed in class. However, be prepared to comment out this randomizaton (or disable it somehow) for one of the experiments you'll do (more info below on this). (We do not want you to use any of the approaches we'll discuss after Module 1 in the course for finding the median or something close to it for partitioning.)

The Exercises

The following explains what we want you to do with your code. Some of these activities are experiments that require you to report on results. Your report must be a document (in PDF format) that is simple and easy to read. Each of the five experiements listed below must be clearly labeled, with the values clearly reported as required and short explanations given when we ask for that.

- Test your two sorting functions to convince yourself that they work. Do this in whatever way you find easiest. You could use a unit-testing framework you've learned (JUnit, pyunit, etc.) or you can write code in your driver program. Either way you should create small lists that are different test cases and exercise your code for different input possibilities or corner cases. For quicksort, test it with *minsize* ≤ 1 to make sure it works as "pure" quicksort, and then with that value set to some small value > 1 to make sure the "hybrid" sort works. Use your *is_sorted()* method for your testing. (That's why you wrote it!)
- **Experiment 1:** Run an experiment in which you fill a list with random values between 1 and 1,000,000. Time both insertion sort and "pure" quicksort. How big should this list be? You'll experiment with the size to find a size that is large enough that comparing the run-times of the algorithms shows a meaningful difference. Once you've found this size (let's call is X), do this two more times with a list that's 2 times the first size(2X), and then with a list that's 3 times the first size (3X).

  For this and the folllowing experiments, be sure you only time the execution of the sort itself (and not the creation of the list or anything else).

  In your report document the run-time for both insertion sort and "pure quicksort" for these three lists of size X, 2X, and 3X (where X is the initial size you chose). Explain if the times are what you expect to see based on your knowledge of the order-class of these sorts.
- **Experiment 2:** Pick one of the input sizes used in the previous experiment, and create a list in sorted order. If your *partition()* function randomly chose an element to partition around, comment that out or disable it for this experiment. Run both insertion sort and "pure" quicksort with this list and record the run-times. Explain if the times are what you expect to see based on your knowledge of these sorts.
- **Experiment 3:** Repeat the previous experiment but make the list be in reverse-order.
- **Experiment 4:** (If your *partition()* function randomly chose an element to partition around, you should now restore that ability if you disabled it for the previous two experiments.)

Create a list of the same size as in the last two experiments that is "almost" sorted. Here, by "almost sorted", we mean that each element in the unsorted list is about 5–10 positions away from its correct position, but almost all elements are not in their final position.

Run both insertion sort and "pure" quicksort with this list and record the run-times. Explain if the times are what you expect to see based on your knowledge of these sorts.

- **Experiment 5:** Here you'll experiment to see if our "hybrid" sorting (where quicksort calls insertion sort on small lists) makes much of an improvement. Try different values of *minsize* with quicksort when sorting a randomly-ordered list of the size used in Experiments 2–4. Try some values between, say, 5 and 50 for *minsize*, and try to find a value that results in the smallest run-time. Report the value that seems to work best fo ryou, and explain how this compares to the result you found in Experiment 1 for "pure" quicksort.

What to Submit

Submit your source file(s), the Makefile (see the course website), and your PDF report to GradeScope. We will run your code in GradeScope by calling your methods from our test-driver. A TA will look at your code to confirm you are really coding the sorts we've asked you to do, and a TA will look at your PDF report. Your source code does not have to include all the code used to run the experiments; we will just want see and test your sorting functions.