

# CS 4102, Algorithms: Homework 1, Fall 2021

## 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 sort and quicksort functions (methods) and associated code to test and analyze these sorts on various lists of data. 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 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 unsorted lists, study the time taken by your sorting algorithm, and answer some questions. You’ll submit a PDF file with the answers to those questions. But you will also submit your code to GradeScope where your sort functions will be called and 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 ArrayList or a C++ vector that contains integer values. The function will not return anything (return type of *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 `list.size()-1` then sorting is done to the last element of the list.
- 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*  $\leq$  1 or the size of the sublist is  $>$  *minsize*, then insertion sort is not used on the small sub-list and sorting is done with a recursive call to `quicksort()`.

Observe that passing the function a *minsize* value  $\leq$  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 algorithm (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 randomization (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.)
- We will provide to you a simple driver program that checks `insertionSort()` and `quickSort()` one at a time and reports any errors. The code includes a method (implemented for you) called `checkSorted()` which checks your sorted list for a few errors (e.g., not having the same elements as you began with, that the list is actually sorted, etc.). This driver will help you quickly test your code to ensure your sorting algorithms are working.

## The Exercises

The following explains what we want you to do with your code once it is written and working (according to the provided driver). 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. Answers for each of the five experiments 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 by tweaking your implementation to ensure you still get positive results from the provided driver (e.g., you might break your implementation on purpose to ensure the driver now crashes, etc.). 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. Once you are convinced your code is working correctly, do each of the following experiments. **NOTE: FOR SOME OF THESE EXPERIMENTS, YOU’LL NEED TO TWEAK THE DRIVER PROGRAM. YOU MIGHT WANT TO HAVE A BACKUP COPY AVAILABLE IN CASE YOU NEED TO REVERT.**
- **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 it  $X$ ), 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 ( $3X$ ). The value of  $X$  that works best for you may depend on your computer and what language you code in, so you’ll have to experiment a little.

For this and the following 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 (i.e., have partition use the element at one end of the sublist to partition around instead). 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 for you, and explain how this compares to the result you found in Experiment 1 for “pure” quicksort.

## What to Submit

You will submit one source file (Sorting.java/cpp/py), the Makefile (provided to you), 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. **NOTE: REMEMBER TO REMOVE ALL OUTPUT FROM YOUR CODE EXCEPT WHAT THE PROVIDED DRIVER REPORTS. DO NOT SUBMIT THE DRIVER PROGRAM AS GRADESCOPE AUTOMATICALLY COPIES IT IN FOR YOU WHEN GRADING.**

## Grading Guidelines

You will pass this homework if your submission contains the following:

- Your code passes the Gradescope test cases
- Glancing at your code reveals that you implemented the correct sorting algorithms (insertion sort and quickSort w/ partition).
- All five experiments were done and reported on.
- All five experiments contain clear numeric reports on timings and clear well-written explanations of the results.