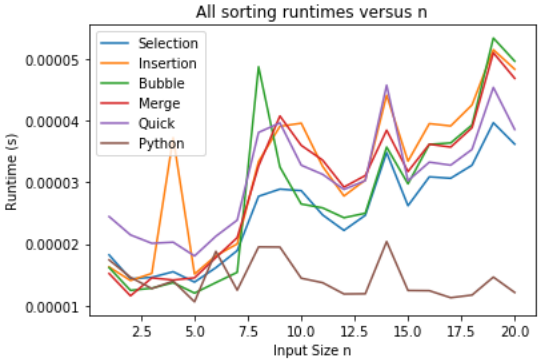
ECE 590
Walker Harrison (weh18) & Lisa Lebovici (lrl22)

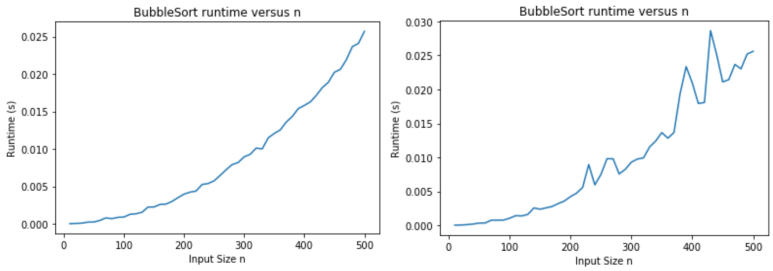
After writing code for sorting algorithms and checking that they can successively process an unsorted list, it's important to put them through rigorous testing that covers a wide scope of possible inputs, from short lists (small n) to enormous lists (large n), and from totally random arrays to completely sorted ones, and even those that are "somewhat" sorted.

Using large values of n, up to 500 or 1000 in this case, is the only way to accurately compare algorithms' run-time complexities. For small values of n, the constant effects or "overhead" represent enough of the total time that they obscure the asymptotic trends. While overhead is an important consideration, the point of Big-O notation is that it is free to focus on only the terms that grow the fastest as n increases. Otherwise, we're really just measuring the algorithm's implementation of book-keeping measures like checking the length of an array. For example, here is the performance comparison of our code for arrays of size 1 through 20. None of the trends are apparent, since they're overshadowed by the overhead:

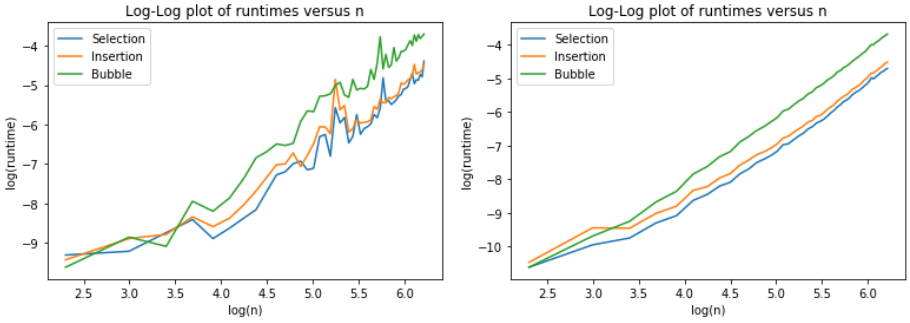


Having large n doesn't make your trials foolproof though. The timing mechanism we're using relies on wall-clock time, which is to say that the runtime is based on the difference between the real-world time at the beginning and end of the algorithm. This might be appropriate if our computers were dedicated entirely to running these algorithms, but of course a laptop's CPU is a resource shared between all of the computer's ongoing processes. So, if we time our algorithm while the computer is pre-occupied with other tasks, we won't get an accurate estimate.

As an example, on the left is our BubbleSort under normal computing conditions, while on the right is BubbleSort while also doing high-dimensional matrix multiplication in the background. The y-axes are nearly the same, so on average the algorithm is still performing as per usual. But on the right graph there are occasional large values that deviate wildly from the trend, which we might attribute to the intense background processes that we've imposed.

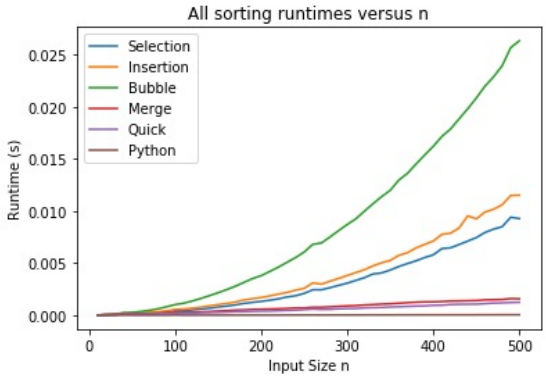


Such variation means that we should not only try to run our algorithms under comparable conditions (or better yet use CPU time instead of wall-clock time), but measure them many times and take the average. Beyond multiple ongoing processes, there are other random elements at play when measuring algorithms—in fact the inputs themselves are random, so taking the average over several dozen trials allows us to strip away some of the added noise and hone in on the actual underlying speed. To illustrate this concept, compare timings done with 1 trial on the left versus those done with 100 on the right. The same general trends are apparent but they're much more erratic and would be harder to pinpoint over shorter ranges:



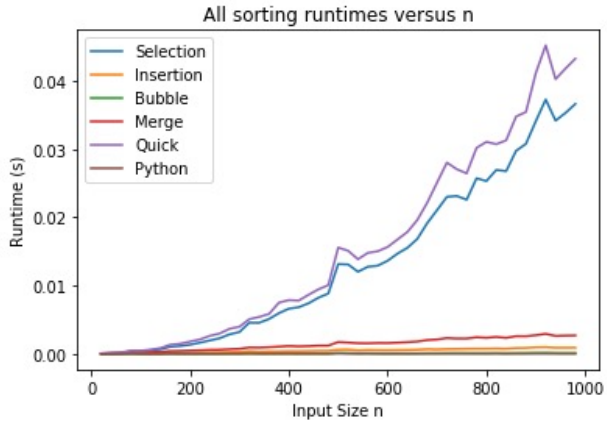
However, once we do run multiple trials under comparable conditions, we're able to observe the trends that are associated with these algorithms. The algorithms separate into two classes: those that run in $O(n^2)$ (SelectionSort, InsertionSort, and BubbleSort), and those that run in $O(nlogn)$ on average (MergeSort, QuickSort, and whatever is used by Python). The former three are starting to blow up as n gets into the hundreds, while the latter three have much more constrained growth, as we'd expect.

The worst algorithm is probably BubbleSort, which is slower than InsertionSort and SelectionSort even though they have the same time complexity. This is because the coefficient of n^2 is usually larger for BubbleSort, since it often has to execute many swaps to get an element to its final location, whereas in SelectionSort, for example, elements are moved directly into their sorted position.



You might think that the best (non-Python) algorithm is QuickSort based on this graph, since it looks slightly faster than MergeSort and begins to separate as n increases. However, looking at performance on sorted arrays tells a different story: suddenly QuickSort is as bad as SelectionSort and BubbleSort is arguably the best. We implemented QuickSort using the right-

most element as the pivot value, which makes no difference for random numbers. For a sorted array, however, it recurses on sub-arrays of size 1 and *n-1*, incurring worst-case run-time of $O(n^2)$. As a result, we'd argue that MergeSort is our best algorithm. BubbleSort, meanwhile, terminates after running through the list with 0 swaps, which is guaranteed for sorted arrays.



This analysis shows that in some cases, theoretical runtimes are more useful and in other cases, experimental runtimes are the practical choice. Theoretical runtimes circumvent issues that arise from discrepancies between computers. As we've shown, the available resources can have significant influence over timing results, so we wouldn't want to say an algorithm was better just because it ran at a different time or on a different machine. Plus, looking at theoretical complexity is better preparation the faster, more powerful machines of the future—many of the advances in engineering ignored the computing restrictions of their time.

However, experimental runtimes also have value. If I know which machine I'll be using and the type of data I'll be inputting, I don't really care how my performance compares to other computers'. I just want an estimate of how fast something might run for me, in which case experimental runtimes are useful.