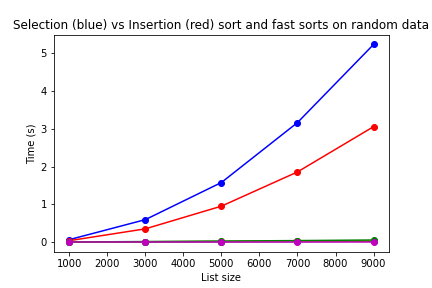
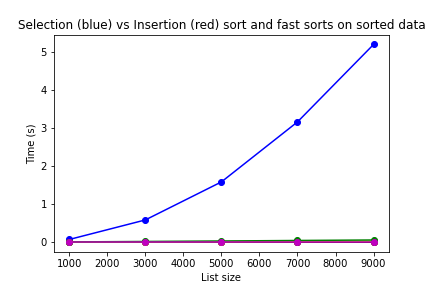
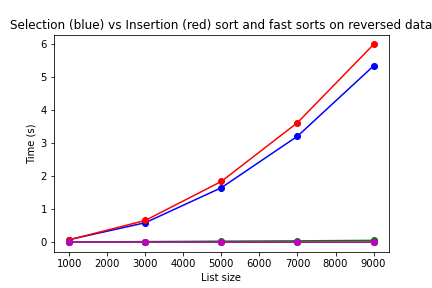
For these graphs, I used blue for Selection Sort, red for Insertion sort, black for Quick Sort, yellow for Merge Sort, green for Heap Sort, and magenta for the Built-In Sort.



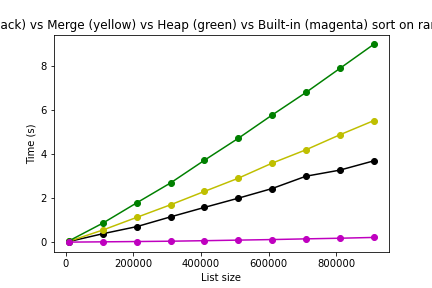
As we can see in this first graph, I am only able to use high enough values to show the slower sorts, Selection and Insertion. The other sorts are much faster, appearing as a jumbled horizontal line on the bottom of the graph. The reason I can’t use super high values for Selection and Insertion sort is because they are a O(n^2).



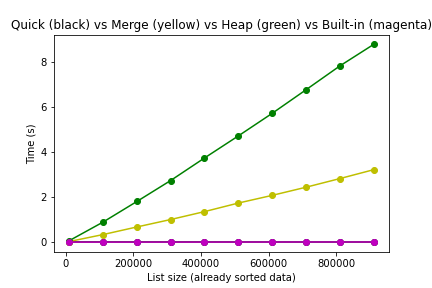
In this graph of sorting already sorted data, it looks like Insertion sort is magically gone from the graph. However, it is just in the jumbled horizontal line at the bottom with all of the other, much quicker sorts. If Insertion Sort is already sorted, it only has to look at each item once, so it becomes a O(n). Selection Sort, on the other hand, remains a O(n^2) because it has to move each item still. So, in this best-case scenario, Selection Sort still has to do its full work, while Insertion Sort is done instantly.



In this worst-case scenario, the list is reversed. Insertion Sort has returned and now takes longer to sort than Selection. This is because Insertion has to move each item every time, while Selection just moves the one it needs to its sorted place. As usual, the much quicker sorting functions make a jumbled horizontal line at the bottom of the graph, not even being phased in the worst-case scenario with such a small list.

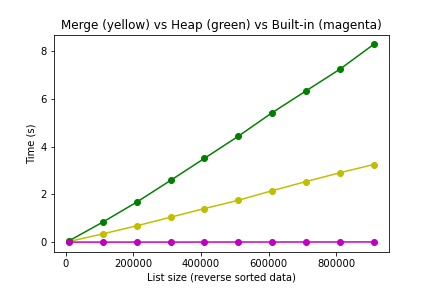


In this graph, the four faster sorts (Quick, Merge, Heap, and Built-In) are graphed in black (Quick), yellow (Merge), green (Heap), and magenta (built-in). The other two sorts (Insertion and Selection) were removed because it would take way too long on lists of this size. Heap, Merge, and Quick all have average running times of O(n log(n)). Heap sort takes the longest on the average case because it has to recursively sort basically twice in two different for loops. Quick and Merge are closer together, Quick sort seems to be slightly faster on the average. Python’s Built-in sort, however, takes the cake for the fastest sorting time. Even though it still has an average running time of O(n log(n)), it is considerably faster than the other sorts. The list size has to get up to the 3.5 million range (not shown in graph) to even hit a second to sort.



The first thing we see in this graph is that Quick sort is gone. It turns out if the data is already sorted, the running time for Quick sort turns into O(n^2), which could not possibly be run with values this high. In fact, it can’t run a list of over 1,000 items without exceeding the recursion limit.

Looking at the remaining sort functions, we can see that Heap sort takes the same amount of time to sort. Merge sort, on the other hand, only takes half the amount of time to finish ‘sorting’. The built-in sorting function is, of course, still the fastest. Upon further testing (not shown in graph), it only takes 0.02 seconds to sort a list of 3.5 million values using the Built-in sorting function if the list is already sorted, as opposed to a little over 1 second if the data wasn’t sorted.



In this graph, we have a list of the three fast sorting functions (excluding quick sort again) sorting completely reversed data, which is the worst-case scenario. Heap sort yet again takes exactly the same time to sort. Something interesting is happening with Merge sort, though. It takes less time on average for Merge sort to sort in its worst-case scenario (a reverse sorted list) than it does in a randomly sorted list (average case). The built-in sort function is, as usual, much faster than any of the other sort functions. It takes almost the same amount of time for the reverse sorted list to be sorted as it is to sort a list that is already sorted.

In conclusion, there are a lot of different sort functions for different scenarios. If your list isn’t that big, it’s probably best to use the Selection or Insertion sort methods. If you have to sort a fairly large list (that is completely random), Quick sort is probably your best bet, but Merge sort is close behind. If you want the same exact sorting time every time for a large list, then you should use Heap sort. Even though the times of Quick sort was not graphed, the function behaves very similarly to the Merge sort function. Instead of using the middle of the list every time like Merge, Quick finds a specific value to pivot around. If you have a ridiculously large list or just want to get the list sorted almost instantly, then Python’s built-in sort function (or Timsort) is what you want to use. It honestly was fun to see how the different sorting functions work and that they take different times to sort.