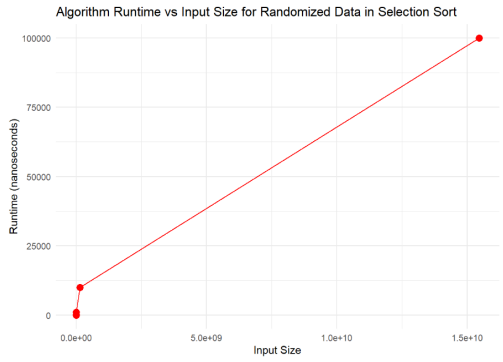
Colin Nguyen

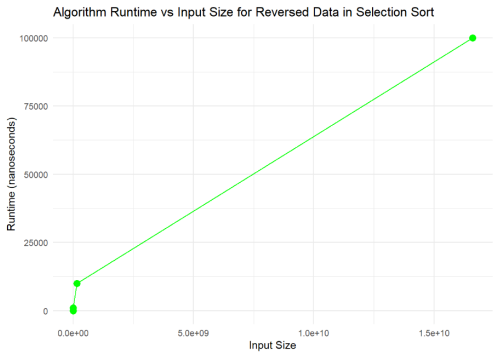
CS 271-01

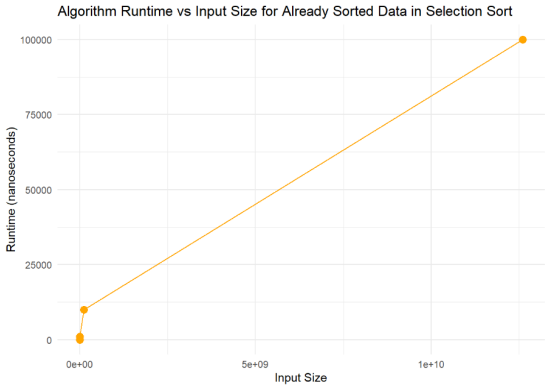
Dr. Currin

September 16, 2025

SELECTION SORT





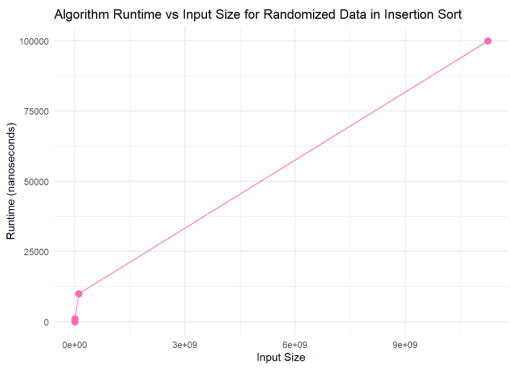


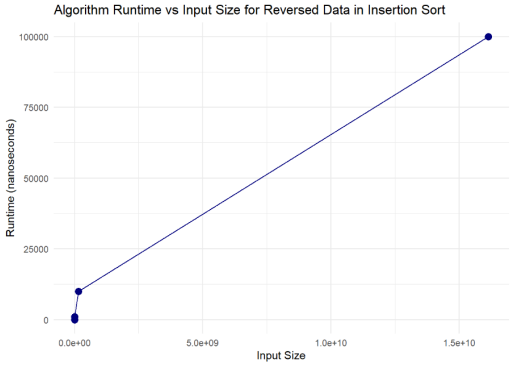
The way that I confirmed that it outputted the correct result was by starting with really small tests (maybe 8 single-digit integers) and then working up from there. For longer lists, I would compare my output to an output created from an online sorter (this was needed for the numbers with lots of digits, making it confusing to really determine if they were in order or not).

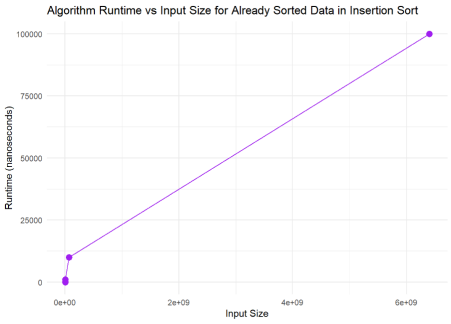
It didn't seem like the runtime changed at all depending on the size of the input, and it all looked like a really steep growth, with the noticeable outlier being the biggest input size. It seems to be less efficient that the $\Theta(n^2)$ for both worse and best case scenario for selection sort. My guess is because the approach I took to getting the pointers into their correct place was by starting at the head each time and then, with a for-loop, move the pointer to their correct node in the list. This wasn't done for all of the pointers, and it may not have been super present in selection sort, but it was definitely present and much more significant in the other algorithms. For me, even though I know that this isn't the best approach, it's just that it felt intuitive to me and so I decided to go with my own idea.

The way that I approached this algorithm and every subsequent algorithm was by drawing it all out on paper so that I could visualize what was going on. Since it all started out completely on paper, I wouldn't catch any mistakes until I plugged all the code in, and when I inevitably found errors or incorrect results, I would have to draw everything back out again to see where those errors were appearing. This took up a lot of paper, but it always allowed me to find out exactly where I was going wrong, and what I could possibly do to fix it. It was often an off-by-one error where I'd either go past the end of the list or I'd be one node short.

INSERTION SORT





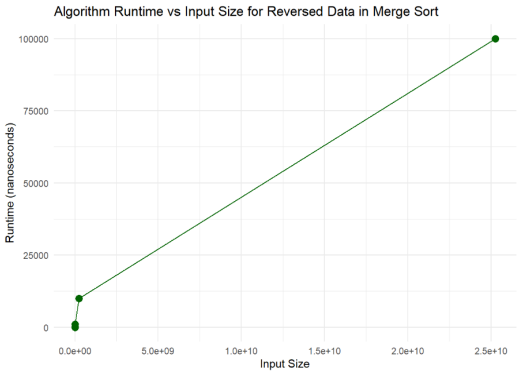


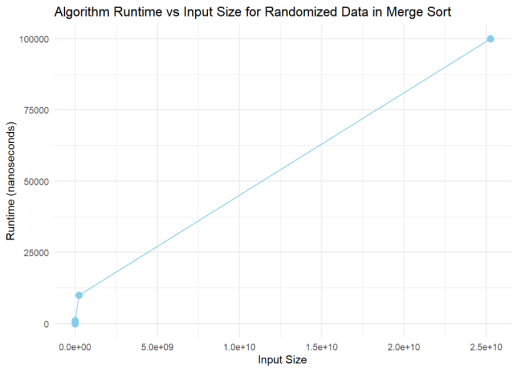
Like selection sort, I would compare the output with an output created by an online number sorter to check for correctness.

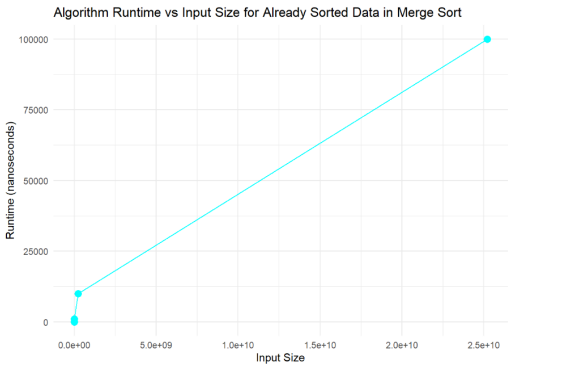
The runtime for insertion sort also looks like it just increases linearly, once again taking up a lot of time for significantly big inputs and taking on a really steep growth for smaller inputs. Like selection sort, I used a lot of for-loops to get the pointers to their correct place, so this isn't too surprising.

This one was easy to quickly get refreshed on the concept of, but for some reason this took longer to implement with a doubly-linked list than selection sort was. I had to draw out the algorithm on a whiteboard so many times to see where my pointer was- and if I got confused where I was at, I had to restart completely from the beginning of the algorithm (like the debugger). It helped me to see where some errors came from- sometimes I accidentally went past the end of the list, other times I would have some sort of off-by-one error.

MERGE SORT







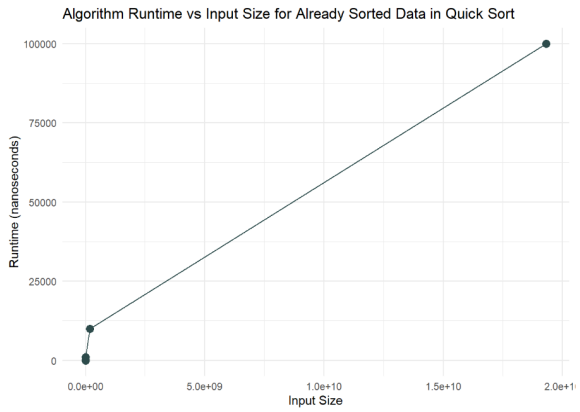
Like all other algorithms, I compared this output with the output of an online sorter (for the really long strings, I used another online application to make sure that both outputs are exactly identical.

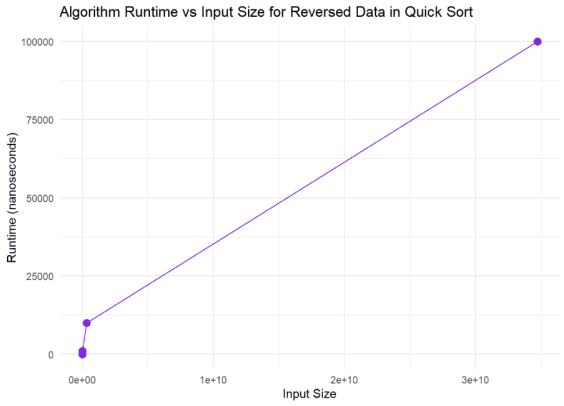
Merge sort runtimes for all types of inputs look exactly the same as selection and insertion sort, with a really steep time growth for smaller functions, so while it does work, it is not at all efficient. It does make sense that the best and worst cases are matching, as that's what it would be like for an array approach.

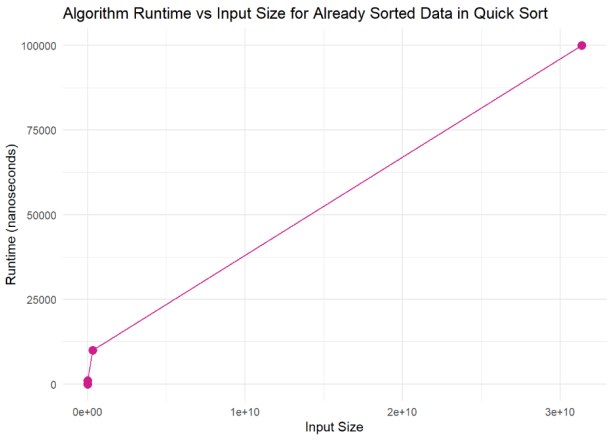
This algorithm was probably the hardest for me because I didn't know how to implement a recursive function that has no parameters or return value- what I did was just use the merge sort

function to call another merge function that actually takes in parameters. This was harder for me to conceptualize on the whiteboard, so I mostly looked to the handout from class and the textbook to make sure I was understanding the algorithm correctly.

QUICK SORT







This algorithm's runtimes look exactly the same as all other algorithms. I checked to make sure that the data and variables that I was using was the ones that I wanted to use and not just the same ones over and over, but they all turned out to be similar. While I did expect them to not look as efficient as the actual sorting algorithm runtimes due to the method that I used to get

pointers into their correct spots, I didn't expect this to cause them all to have the same runtime. There is no difference between the best and worst case scenarios, and I evaluated the correctness of their output all the same as the rest.

This algorithm was harder than selection and insertion sort, but easier than merge sort (primarily because I didn't have to create a copy of the list input to be passed in each recursive call). I only had to use the array-like for-loop approach twice to get my pointers into their correct place, but at least the partition algorithm felt much easier to grasp and implement, even if this took a long time to debug and analyze.