Analysis - Putting your affairs in order
James Gu
Fall 2021

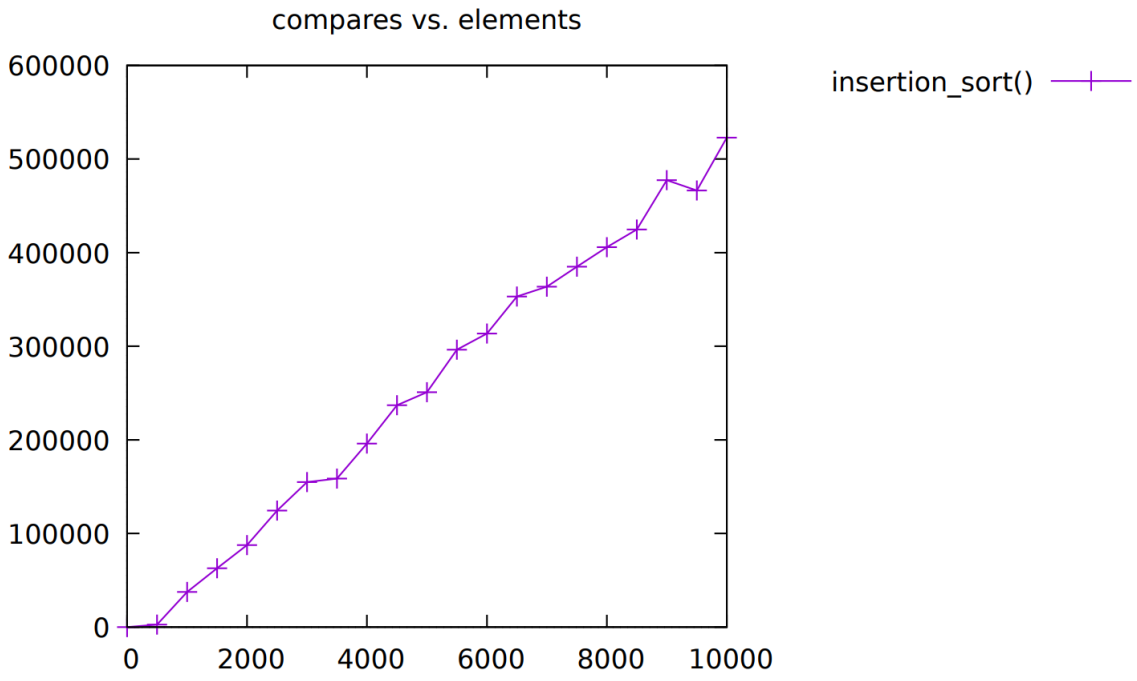
1. Introduction

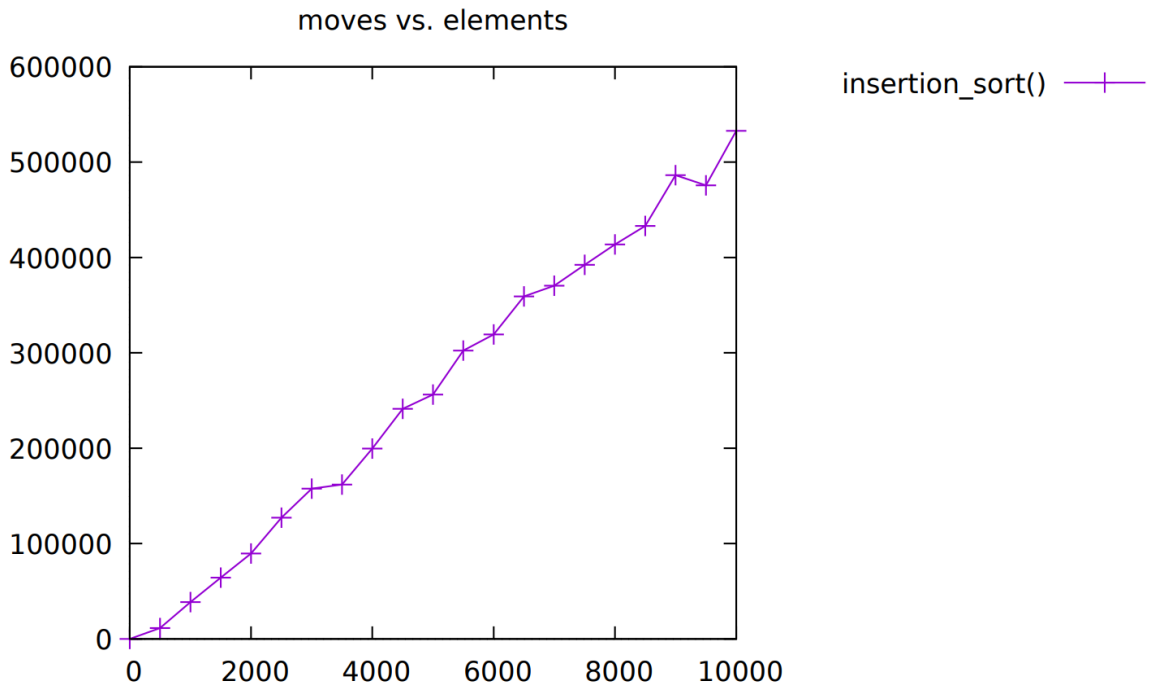
Sorting elements is a very common problem in Computer Science so this program is an implementation of a collection of various sorting algorithms. These include the following: Insertion Sort, Shell Sort, Heap Sort, and Quicksort. Insertion Sort goes through each element one by one and places them in the correct spot. Shell Sort is a variation of Insertion Sort, but first sorts pairs of elements that start with a gap and continues until the gap is one. Heap Sort can be visualized as a tree in which the top most node is the greatest number for max Heap Sort and the least for min Heap. This is called the parent node and its children nodes need to be less than the parent for max heap and greater than for min heap. If the child node doesn't obey the rule of the heap they are switched with the parent and this process is repeated until the entire tree obeys the rules. Each time in this process the top-most node is placed outside the tree since it no longer needs to be switched with anything and is sorted already. Quicksort involves a pivot number which has a partition: the left partition has numbers smaller than it and the right partition bigger than it. Then the sort organizes each partition using the same process as the original pivot.

2. Analysis

insert.c:

The file insert.c involved the most basic and intuitive way to sort an array. How the algorithm works is that the program goes through each element one by one so the growth of moves and comparisons should be linear as so in the following graphs:

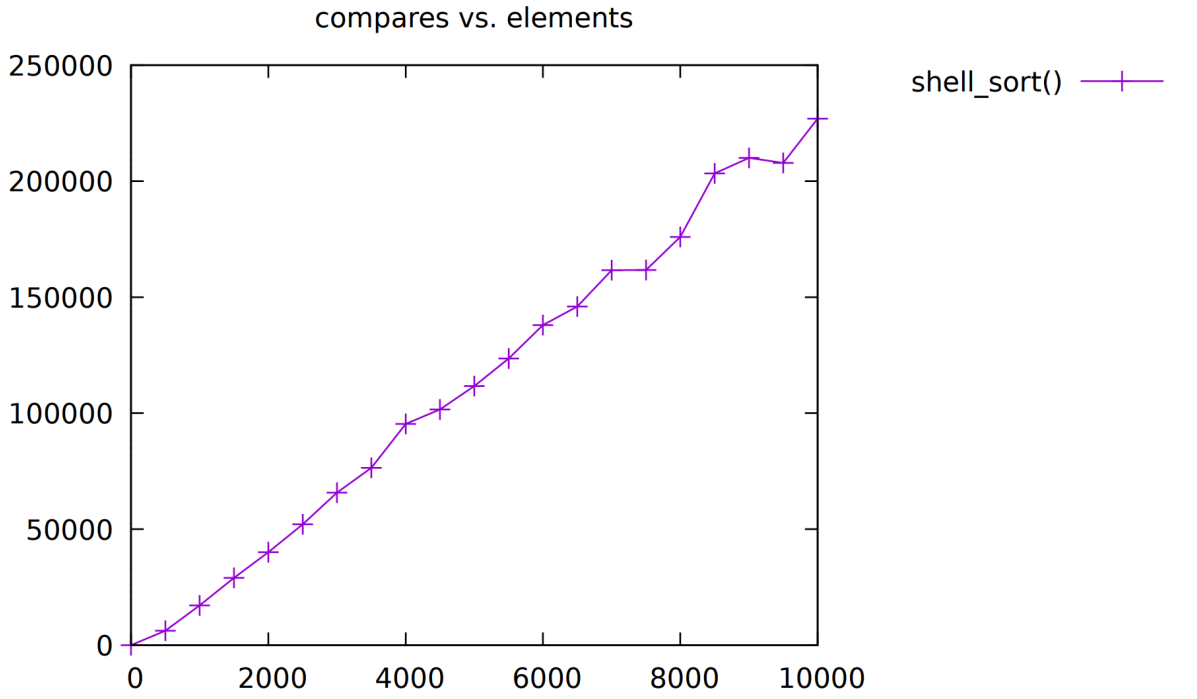


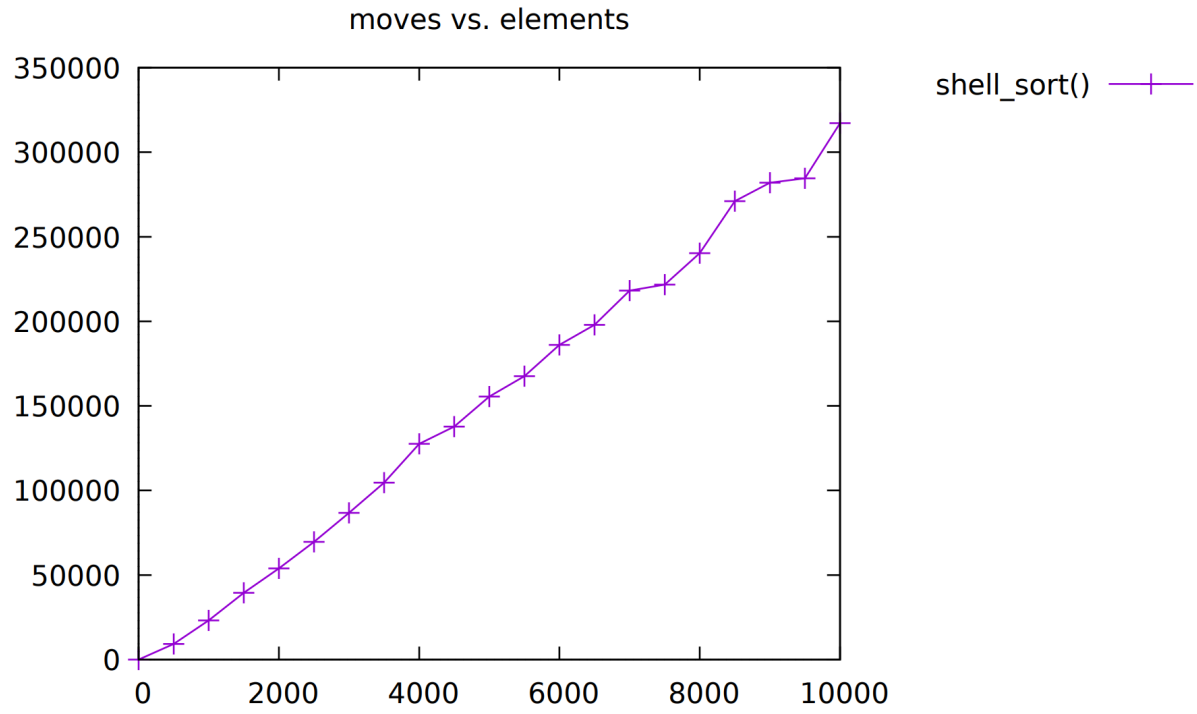


Both the comparison and moves graphs have a similar upward linear trajectory. This implementation had a longer runtime compared to the other sorts when it came to larger amounts of elements perhaps because the sort checks every single element and doesn't take any shortcuts. This is because the implementation is just one for loop going through all the elements.

shell.c:

Shell sort method is based on the above insertion sort. They are variations of each other. Shell uses a gap system which decreases at each loop until it reaches 1. A gap of 1 is the same as the insertion sort. Perhaps because they are variations of each other, the graph for shell also appeared pretty linear:



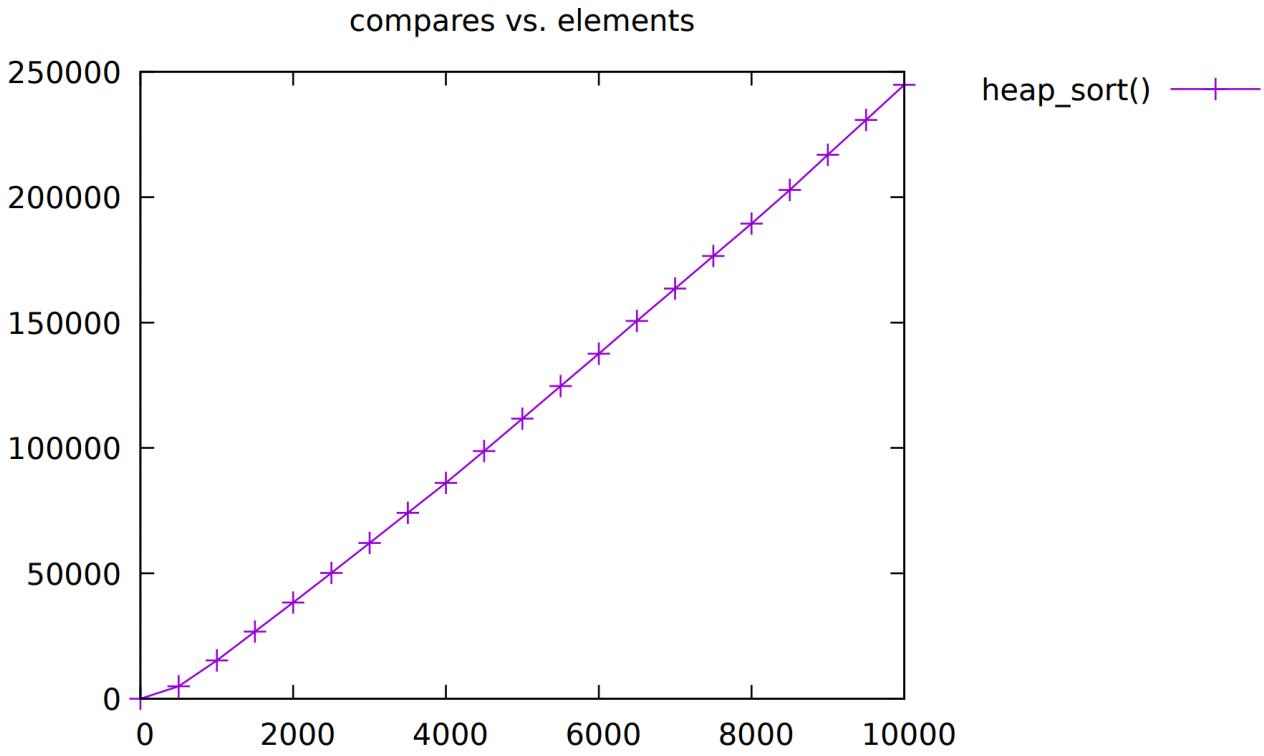


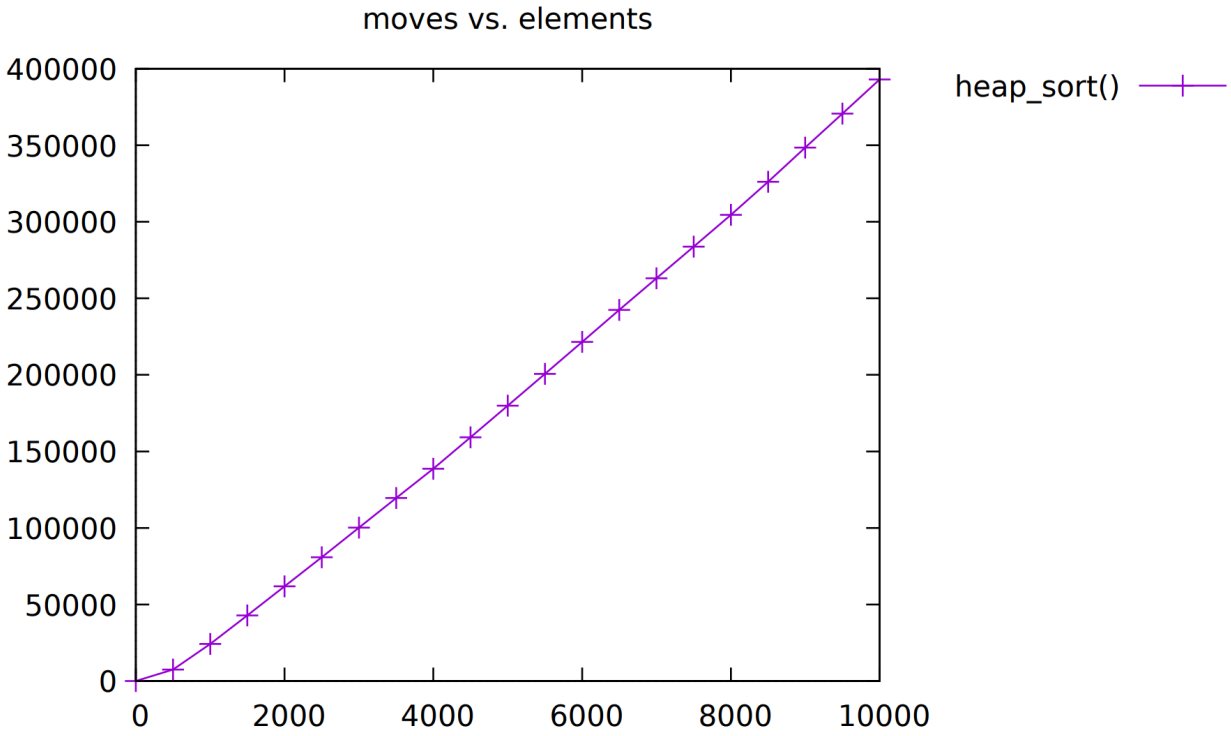
This graph was made using points spread across the results of sorting arrays in between 0 and 10000 in size. Near the 10000 mark the line seems to fluctuate a bit compared to the insertion sort. Even with these slight discrepancies, shell seems to almost double the efficiency of insertion sort near the end. The top end of moves for shell is 300000 moves while insertion is

600000 moves. The implementation of this sort involved a helper gap function that would return the current gap because it keeps decreasing. Inside the primary shell_sort() function it features a similar for loop to insertion perhaps because they are variations of each other.

heap.c:

The heap sort is on of the more interesting sort algorithms. The main visualization to help understand this sort is a tree. The numbers at the top of the tree have to be higher in value compared to the children branches. So this implementation involves creating this "tree". The function to do so is called build_heap() and it uses the function fix_heap() which uses max_child(). fix_heap() makes sure that the tree is obeying the rules of the heap and swaps the chidren and parent numbers that don't agree. max_child() returns which child is the greatest and might have to swap with the mother. The amount of compares and moves are as follows:

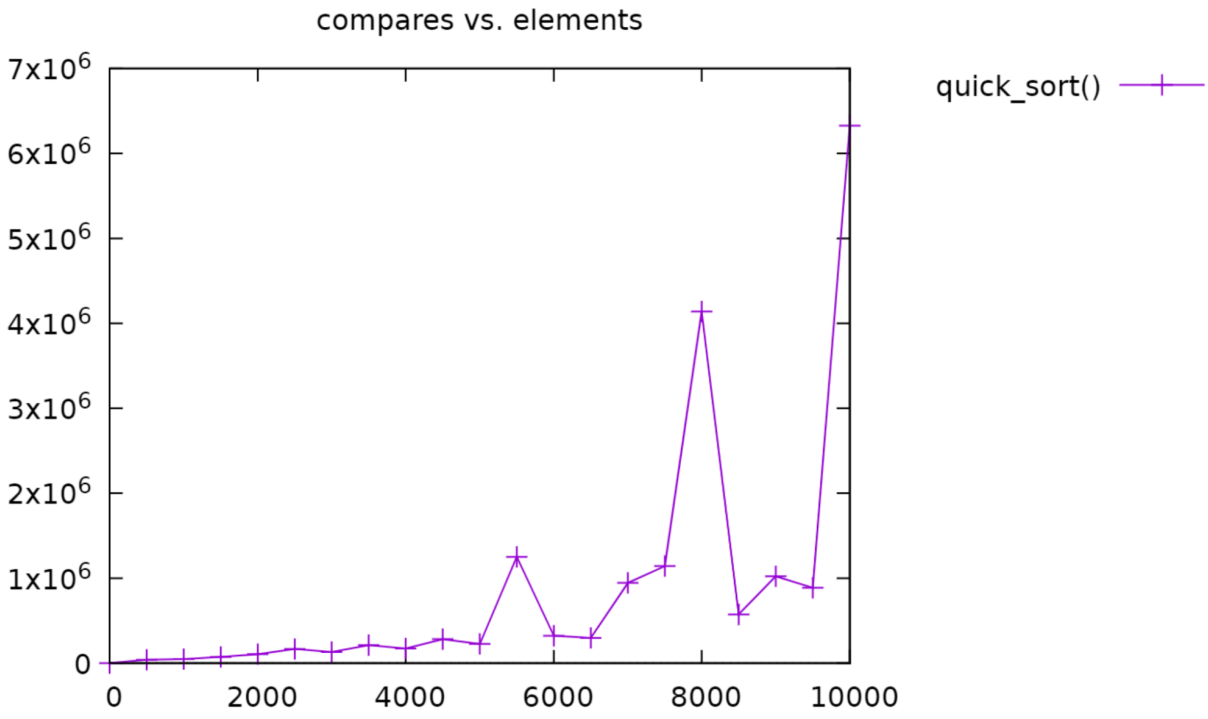


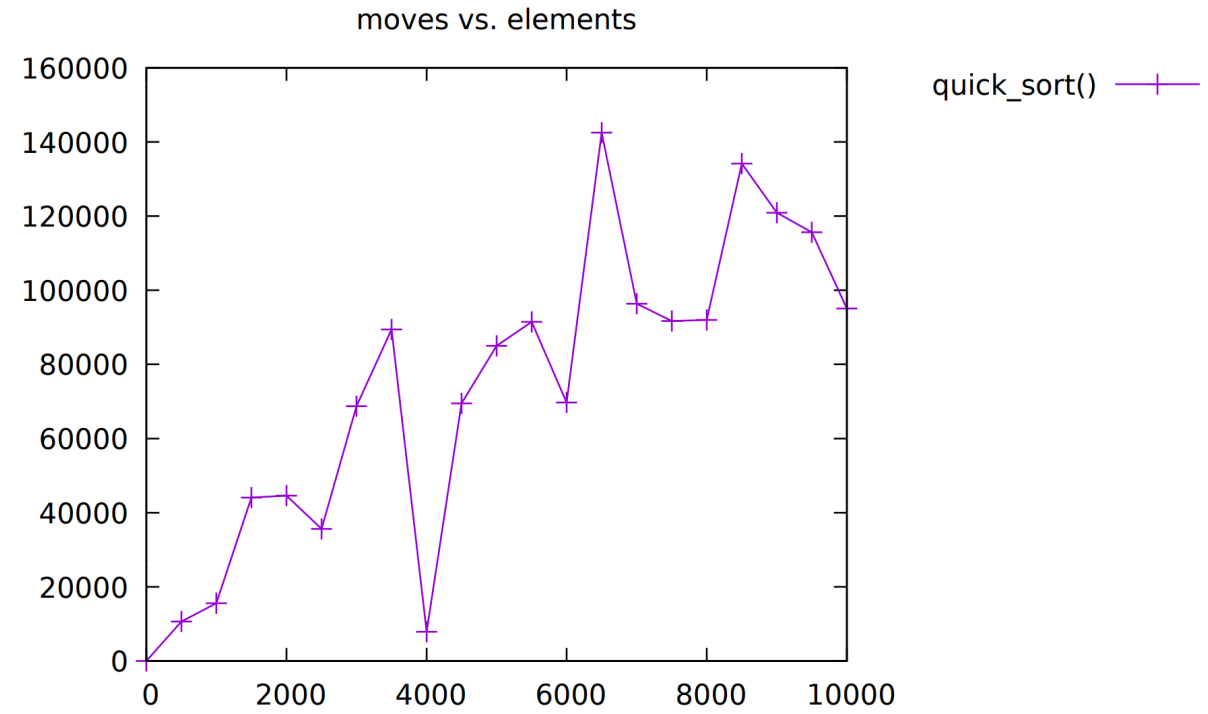


The values of this sort at the top end are similar to that of shell, but what is noticeable is that the curve straightens while shell doesn't. While the values are similar, it is perhaps because heap does so many moves and compares even tho it takes less time. It needs to compare children and parents frequently and swaps increasing the moving by a lot.

quick.c:

Quicksort is the last sort algorithm and it involves recursive calling itself. The theory behind it is that there is a pivot number that has a left and right side. The left of it has lower numbers while the right of it has higher numbers. Once all the numbers are organized it does the same ruling to both sides until every number is rightfully sorted. A new pivot is assigned to the left and right and new partitions are made. It's a quick way to sort but fluctuates a lot for compares and moves:





These graphs are probably the craziest of them all. My reasoning for these fluctuations is the recursiveness of the functions. quick_sorter() calls itself many times and the way numbers get partitioned can easily stack up compared to just linearly going through the array. On the other hand, when quicksort works the best is when it can divide and conquer and shortcut through

sorting. Worst case scenario quick has to go through n^2 times so some data points might have been unluckily seeded.