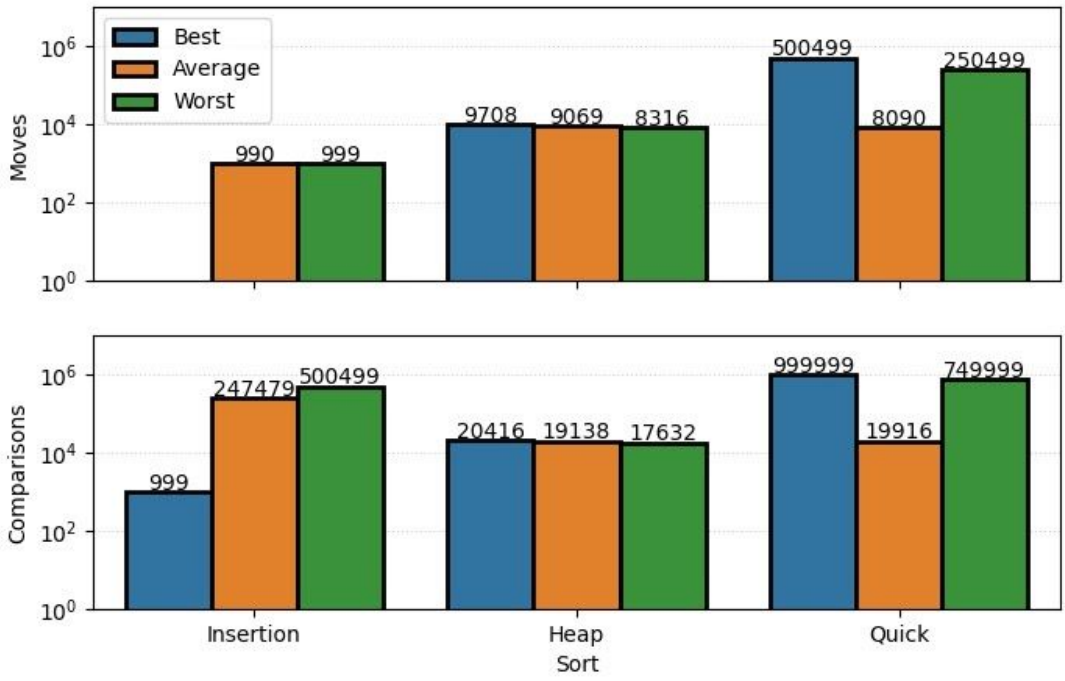
First, I implemented the three sorts in C++. I identified where the algorithm moves and compares elements in the data structure and added a count to track that. Next, I generated the best, average, and worst case array to sort. I simply ran all three sorts on all three arrays, which gave me 9 results for move counts and 9 results for comparison counts. After each run, I did a simple check to ensure that the sort algorithms ran properly and sorted the arrays correctly. The following figures show the plots that are the moves and comparisons that I counted as I ran the "experiment" with my implementation. I used Python libraries to plot, as I run Ubuntu on my laptop and do not have access to Excel nor do I know how to use it. I will attach my Python code for these plots along with the code to implement the "experiment".
***Note**: the y-axis is log scale.*



i. The insertion sort algorithm results in zero moves when sorting the best case scenario because the algorithm only moves unsorted values to the correct position, which is untrue for all the values in a fully sorted array.

ii. The insertion sort algorithm results in 999 (or n-1, n being the length of the array) comparisons when sorting an already sorted array because comparing the key to the elements prior only needs to be done once each iteration to show the element does not need to be moved and it will continue to next element as key.

iii. Based on my two plots and our two metrics, Heap Sort is the most consistent in performance and insertion is best for a mostly sorted array. Heap Sort is the most consistent as the moves and comparisons do not significantly between the three scenarios. Insertion Sort performs better as the array is more sorted previously. Quick Sort performs best for the average case, but worse for the best and worse cases.