WRITEUP ASGN3

Cruzid: acristea

February 2023 - CSE13S

1 Creating Sorting Algorithms

- For every sorting algorithm, I used asgn3.pdf from our resources git to understand them and then write them in code. Using the stats.c and stats.h files provided, I broke the python pseudo-code down line by line and converted it into C. I utilized what I learned from Asgn 1 and 2 when it came to creating graphs and made a bash script allowing me to depict my data as shown below.

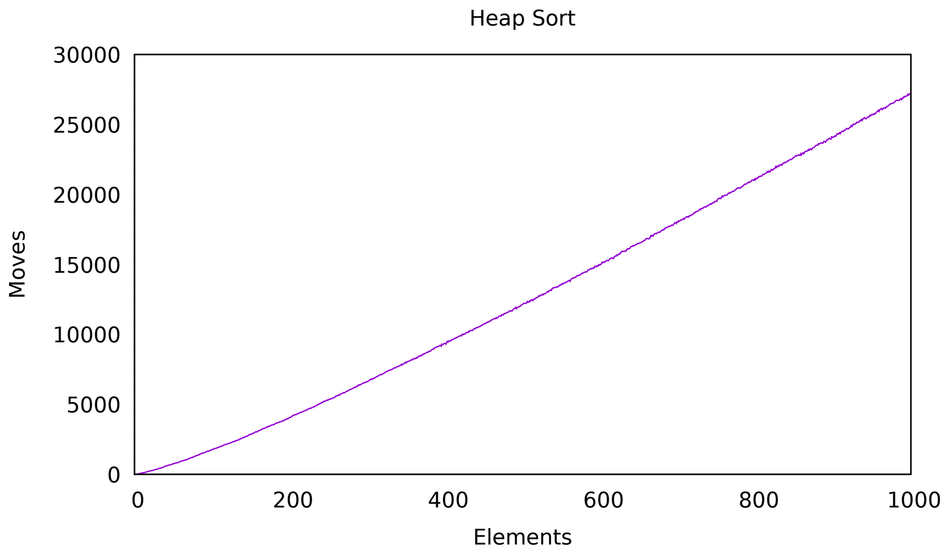
Heap Sort - Looking at the pseudocode from asgn3.pdf, it was obvious Heap Sort wasn't going to be the fastest out of the bunch, but it would be efficient as the time required to perform Heap sort increases logarithmically. I also realized its memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work. This does however slow it down, making it very costly.

Shell Sort - Using the pseudocode from asgn3.pdf, which was the shell sorting algorithm in python, I first figured out I had to get the gap. Because of this, Shell Sort needs an extra for loop, which makes it a lot more inefficient than other sorting algorithms. It's a quicker version than insertion sort, but still slow.

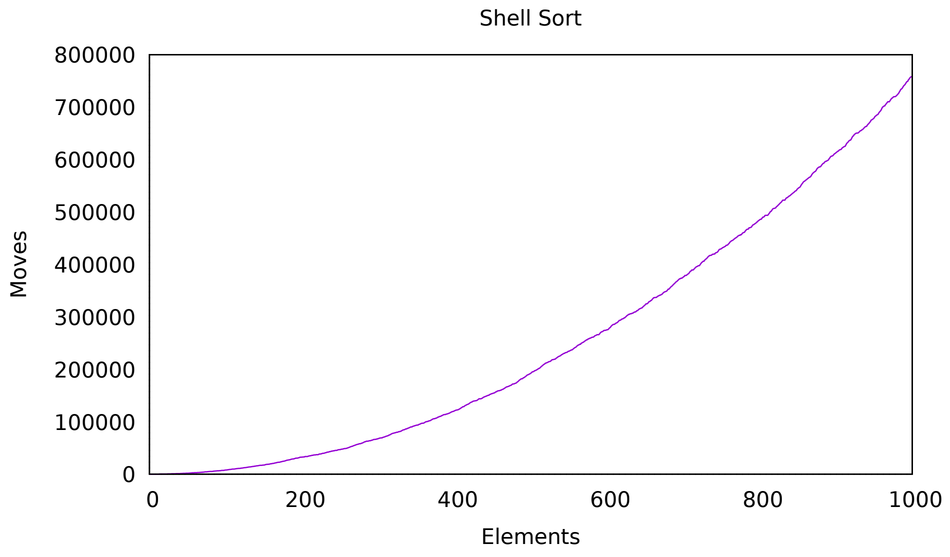
Batcher's Sort - Looking at the pseudocode from asgn3.pdf, I was really curious to see how Batcher would perform I had initially thought it would be the slowest. I was very surprised to see it keep under 25000 moves (explained more in the graphs).

Quick Sort - Like the name suggests, Quick Sort has the potential to be the fastest, but at its worst run time can be longer than wanted O(n*n). If the sorting algorithm gets lucky, and its pivot is well placed, Quick Sort can be very efficient, in fact, the most efficient from this group.

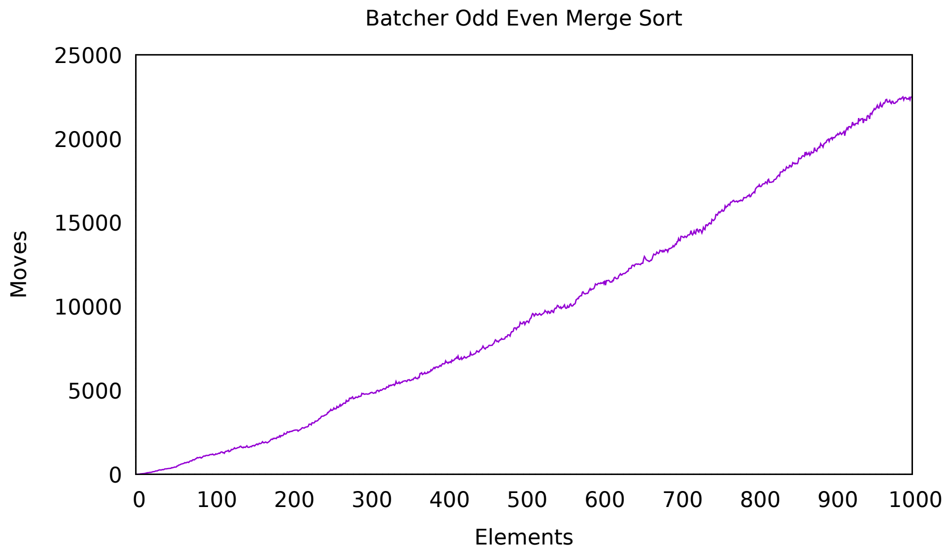
2 Comparison of Sorting Algorithms



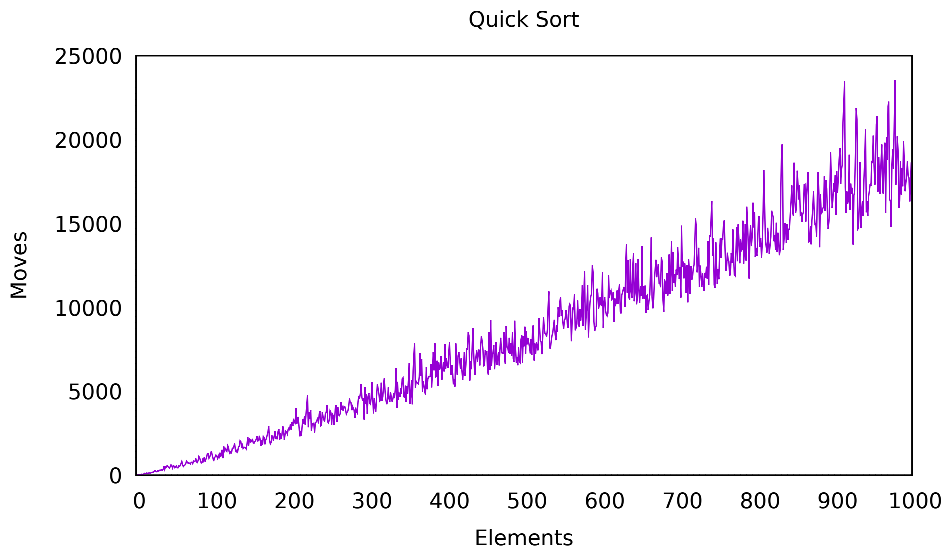
This plot shows our heap sorting algorithm over 1000 elements. It tracks the number of moves the algorithm took based on the size of the array. As you can see, the heap is linear meaning the number of moves increases proportionally when the elements are increased as well. Looking at the graph above, we can see that the slope of the correlation between moves and elements is under 1, meaning as the elements increase, the amount of required moves decreases. So, that being said, longer sequences seem to be more successful and more efficient when using Heap Sort. This doesn't show the heap's log potential and maximum efficiency which it perhaps why it's one of the slowest.



This plot shows our shell sorting algorithm over 1000 elements. It tracks the number of moves the algorithm took based on the size of the array. As you can see, the shell is by far the most inefficient. If we observe the graph above, we can see that the correlation between moves and elements is growing at an exponential rate. In short, we can say that if we were to utilize this sorting function for larger arrays, it wouldn't be very efficient and it would be very costly. However, if the list was short, Shell Sort would be more useful.



This plot shows our Batcher Even-Odd Merge Sort algorithm. If we observe the graph above, we can see that the correlation between moves and elements for this sorting function is slightly linear. I would say that because the slope of the line looks to be less than 1 in terms of the graph's dimensions, we can say that for longer arrays, this function would be more reliable and less costly than that of Shell Sort. We can compare this to the previous Heap Sort, however, the data drawn shows a weaker correlation, so Heap would still "take the cake".



This plot shows our Quick Sort algorithm. If we observe the graph above, we can see that the correlation between moves and elements fairly resembles that of a linear line. I say fairly because as the elements increase, the dissonance in the line grows bigger. I would also say that this dissonance is another sign that maybe Quick Sort isn't the most reliable, but can be fast under the best sort of conditions. I would assume that the data points that are lower represent those arrays that are almost sorted, or when the pivot is well placed and the higher of the points are the arrays that are less sorted to begin with or the pivot is not well placed.

3 Conclusion

In conclusion, I learned how to read python files and convert them into working C files. I learned how to use sets, use unregistered integers and allocate memory space. I also learned how to work with arrays in C and further increase my knowledge of sorting algorithms and time complexity.