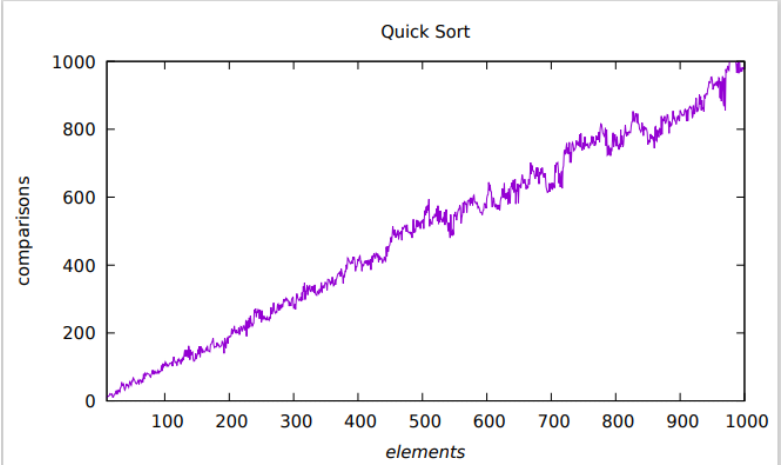
Assignment 4: Write-up, discussion and results

James Contini
10.23.22

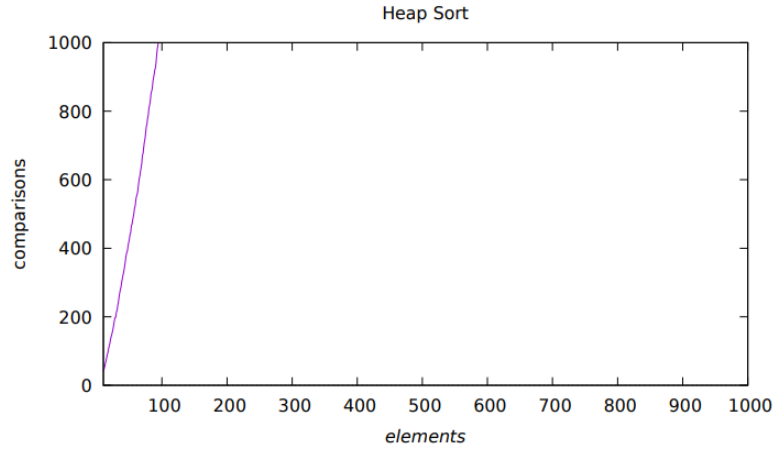
In this assignment, students were asked to investigate four sorting algorithms of various time complexities.

The first topic I want to discuss is how the perceived efficiency of Quicksort and Heapsort change by changing the definition of a comparison.

Quicksort is a divide and conquer sorting algorithm with an average time complexity of *nlog(n)* and a very good worst case complexity of *n2.* To visualize the algorithm's efficiency I used gnuplot to graph data from the mersenne twister random number generator. The plot shows the relationship between a set's number of comparisons until sorted and the amount of random numbers in the set.



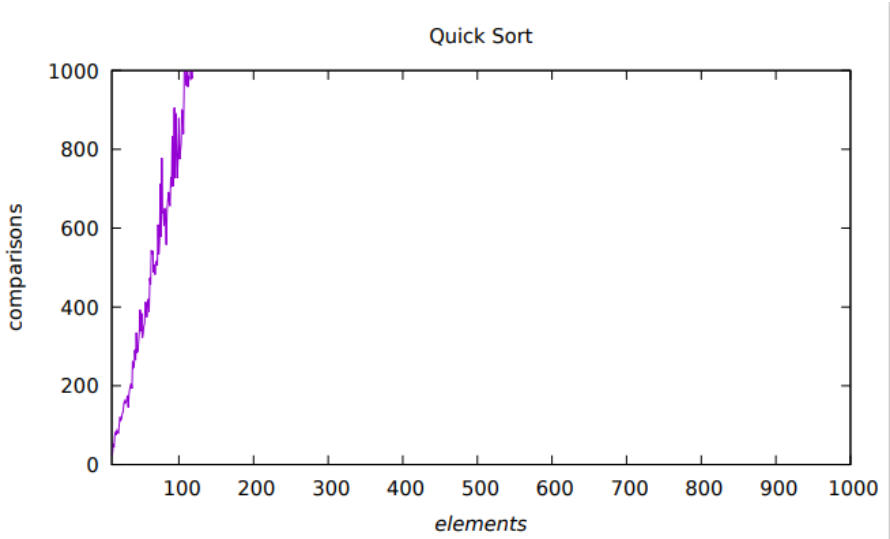
Just like Quicksort, Heapsort is of *nlog(n)* time complexity. But this graph didn't look like *nlog(n)* nor did it look like Heapsort below:



Strangely, my Quicksort looked like it ran at O(n), so after looking through my code and graphing various changes, I realized that I wasn't counting comparisons between a cached array value and an actively

assessed value. In my defense the reason I didn't is because, in CSE30 we had an assignment which tracked array accesses because we were told they were "expensive" .

- Including temp comparisons my Quicksort graph now looks like:

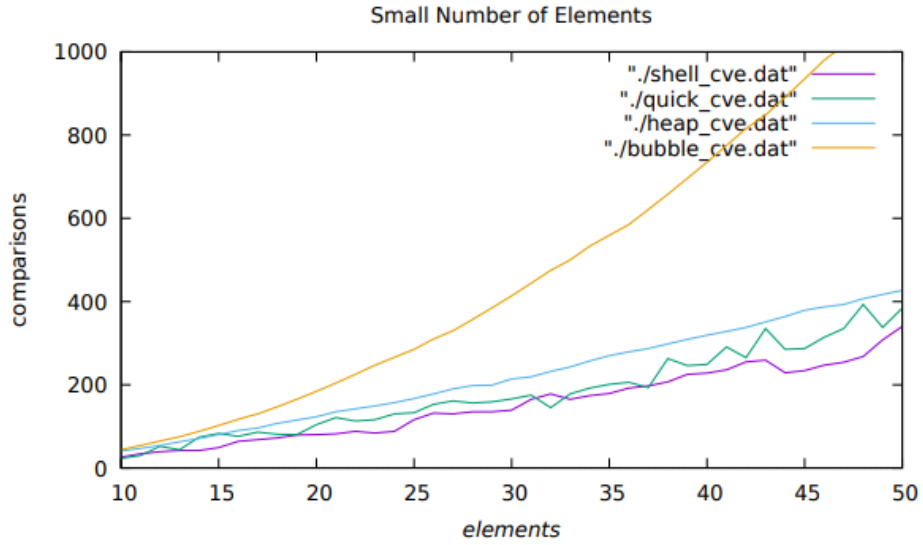


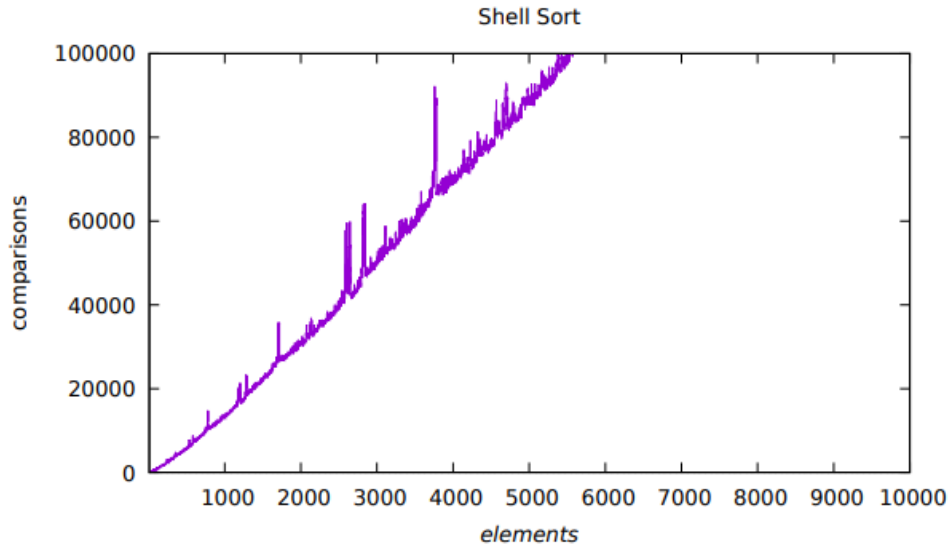
-which makes a lot more sense than the previous one graph.

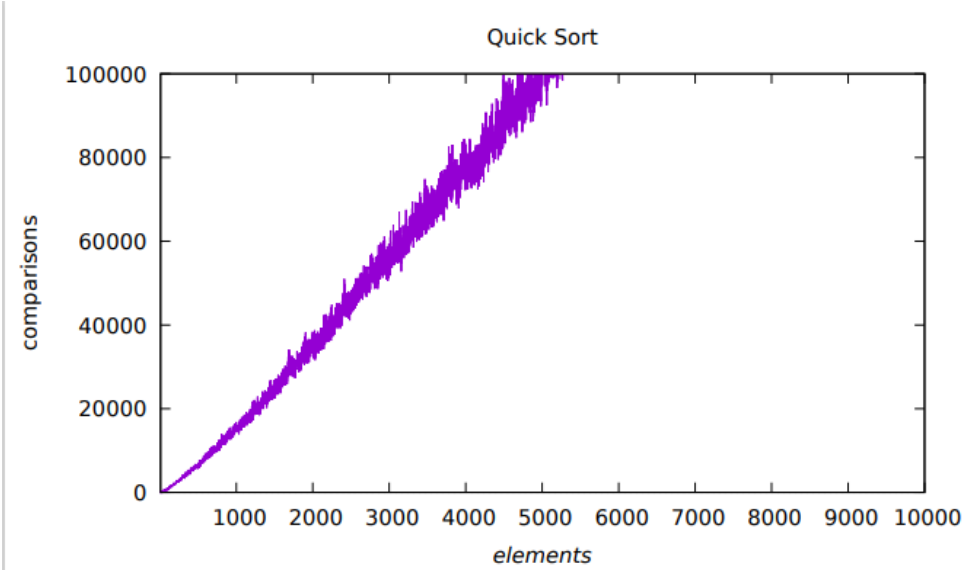
Referring back to caching efficiency, array access can actually be expensive if one is working in certain environments. According to Professor Miller, Python lists are actually a mix between a dictionary and an array, therefore in Python, there would be more of an incentive to cache array elements instead of actively accessing them.

Small amount of elements:

Similarly, we were also asked to investigate how well each algorithm would perform given a smaller amount of elements *n*. I predicted bubble sort would have nearly the same efficiency at the beginning as other sorting algorithms because the graphs of *nlog(n)* and *n^2* are very close at small values of *n*. And I was right about that being the case, but what I didn't expect to see was how quickly Bubble Sort became outperformed by all the other algorithms. I also didn't expect Shellsort to outperform Quicksort since in higher values of *n* Quicksort is on average better than Shellsort.







The linear model for Quicksort models looks like:

```
Linear Model:       y = 22.65 x - 9951
Slope:              22.65 +- 0.01947
Intercept:          -9951 +- 112.5
Correlation:        r = 0.9963
Sum xy:             7.054e+12
```

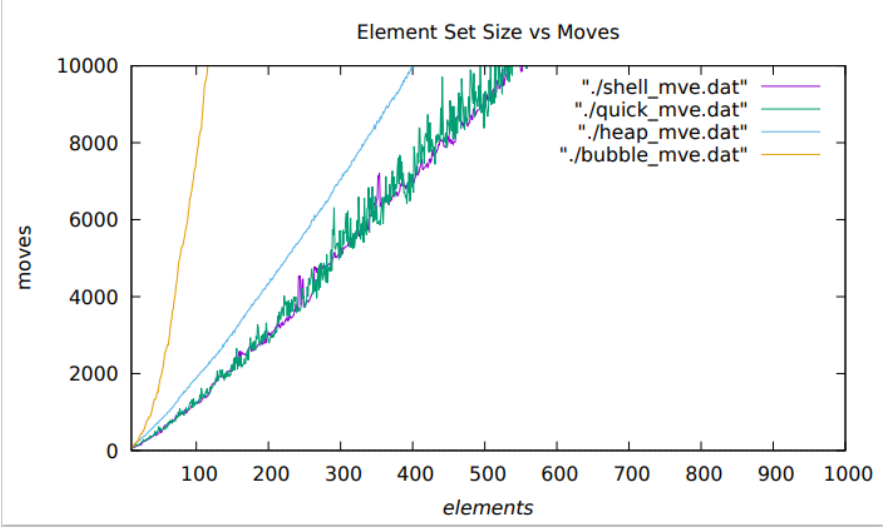
While Shellsort's linear model looks like:

```
Linear Model:       y = 20.59 x - 9138
Slope:              20.59 +- 0.02114
Intercept:          -9138 +- 122.1
Correlation:        r = 0.9948
Sum xy:             6.409e+12
```

Contrary to what I just said, my algorithm, the linear models of both curves, indicate that my implementation of Shellsort is slightly more efficient with large values of *n*.

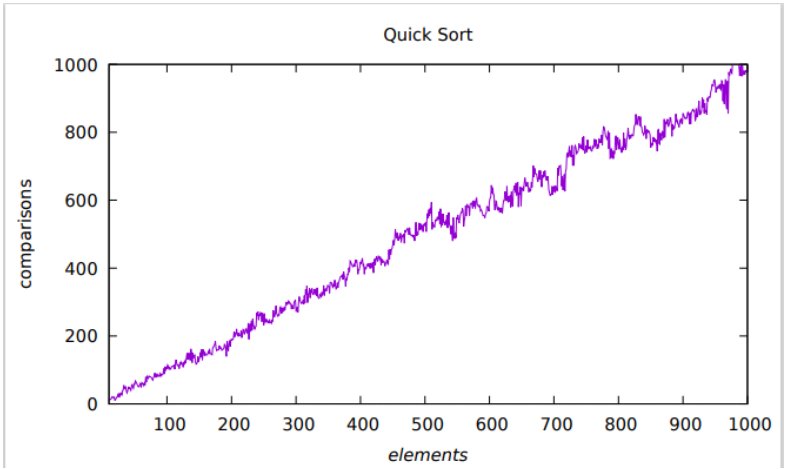
In addition, due to the large fluctuations in comparisons for Quicksort, Shellsort would be a more reliable algorithm for sorting completely random databases. However, I said earlier the cache efficiency of Quicksort and its near runtime performance to Shellsort makes it more efficient for certain applications where accessing data is costly.

Cache Costly:



This is a graph that plots the *mve* or "moves versus elements" curve for each algorithm. The x axis represents the number of random numbers in the test set and it goes from ten to one thousand. Comparing the number of moves that Quicksort and Shellsort use to sort their sets it can be seen that while Shell may have a small advantage over Quicksort, they are just about the same.

To reiterate, in a situation where accessing the value of an element can be runtime costly, Quicksort really is the way to go since it just about equals the amount of moves Shellsort takes. The only difference is how much faster Quicksort can be if accessing elements is an issue.



Here is the plot showcasing *O(n)* runtime if we ignore temp value comparisons. How cool.

In conclusion:

Bubble Sort becomes extremely inefficient extremely quickly especially once *n* becomes greater than fifteen. Heapsort, although I didn't talk too much about it, still remains as one of the faster algorithms for sorting at *nlog(n)* . Shellsort and Quicksort though, are more competitive than either other algorithm I discussed today. Shellsort should be recognized for its reliable curve and its overall greater total comparisons efficiency. But Quicksort's caching technique lends it to be a very efficient sorting algorithm for certain applications where accessing data is expensive.