1.) Time Complexity

a.) Bubble Sort

i.) Bubble sort is a simple algorithm, but its simplicity comes with a cost. The best time complexity for bubble sort is O(n), and that's if the array is already sorted so it only loops once to notice that everything is sorted. The average and worst time complexity of bubble sort is O(n^2) since it consists of two nested loops. The constant of bubble sort depends on the length of the array. For smaller arrays with few elements, such as less than 20, the constant wouldn't impact the speed in which the algorithm sorts as much. However, if we have an array with much larger elements and a bigger length, bubble sort would perform really badly since the constant would be the number of elements it must iterate over. Having a large constant and array impacts the speed of the algorithm. For instance, I tried to sort 10000 elements and it took over 10 minutes to sort them. I have an old Mac 2012, which influenced how fast it ran, but the number of elements took a while to get sorted compared to the other functions.

b.) Shell Sort

i.) Similar to shell sort, the best time complexity for shell sort is O(n) in the instance that the algorithm is already sorted. Shell sort's time complexity is a bit tricky to calculate. Depending on what type of method you use to calculate the gap, the average time complexity changes. Typically though, the best time complexity of shell sort is O(n^5/3). Shell sort is an okay algorithm. It is not the best, but it's also not the worst. The constant for shell sort is relatively small, which is why it doesn't influence the time complexity as much. However, if we have a large array we want to sort, it will be a little slower than other algorithms because sometimes shell sort does more comparisons than it needs to do. Not as slow as bubble sort, but enough to trigger the fans in my computer.

c.) Quick Sort

i.) Quicksort is a predictable algorithm we can always count on. The average time complexity of this algorithm is O(n log n). If we pick a good pivot, the algorithm can sort quicker. However, if the array is already sorted the worst time complexity will be O(n^2), this sorting algorithm doesn't perform as well as bubble's or shell's best case time complexity. The constant of quicksort doesn't affect the time complexity as much since it already has a pretty good average time complexity. Even really big constants don't slow down quicksort as it does to other sorting algorithms. While I was testing my quicksort, I definitely noticed how faster it gave me results versus shell or bubble sort.

d.) Heap Sort

i.) Heapsort is a pretty reliable sorting algorithm. The best time complexity for it O(n log n). Quick sort's worst time complexity is also O(n log n). The constant for heap sort doesn't really affect its performance. Heapsort performs very well with larger constants and large arrays. While I was comparing the different sorting algorithms, heap sort always ran quickly.

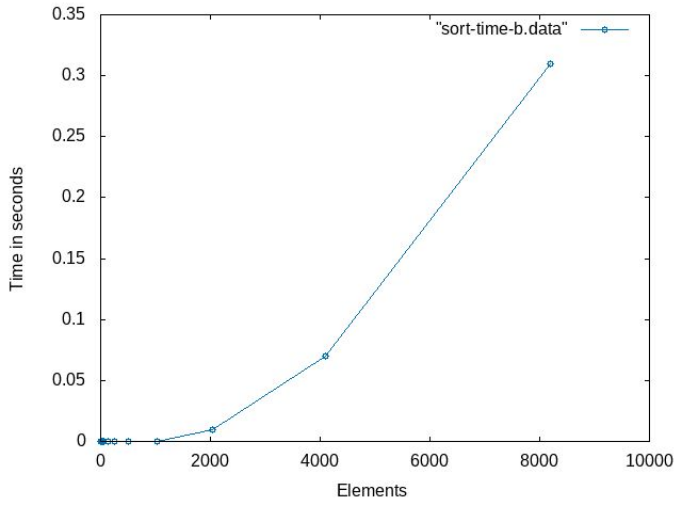
2.) What I Learned

a.) While I was implementing each algorithm, I learned a lot about time complexity as well as solidified my knowledge of indexing arrays and pointers. I learned that bubble sort is very easy to implement, but it will take a long time for it to do what I want it to do. If I want to sort an array with a small number of elements, I would typically call for bubble or shell sort. When I wanted consistent and quick results, I used quick sort and heap sort. Quicksort taught me that if I wanted consistency, quicksort would most likely always be consistent. Heap sort was a very hard concept for me to understand at first, but after working out a few examples at hand I was able to pick up on most of the logic. Heap sort taught me that if I want to sort large arrays, I should probably use heap sort because of its time complexity.

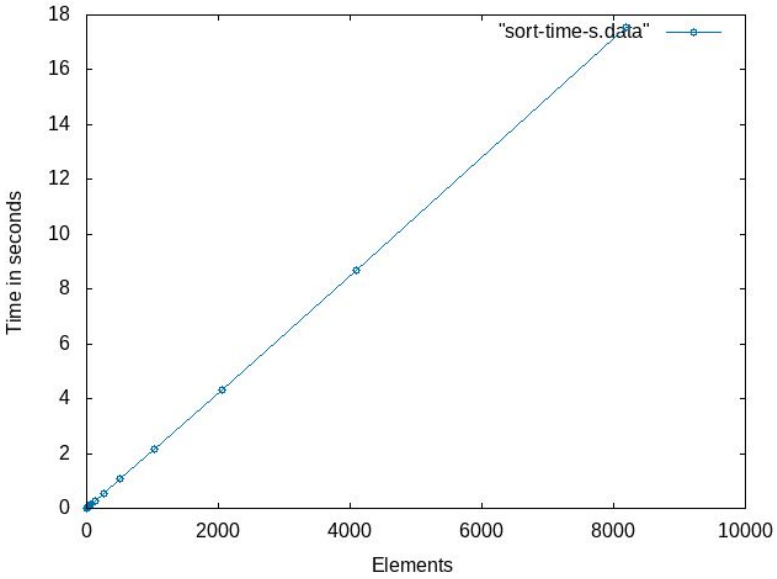
3.) How I experimented with the different sorting algorithms.

a.) To fully understand what the algorithms were doing, I first ran the python code and tried to follow that logic. I added a lot of print statements to my python code to see what each line was doing. Doing this really helped me visually see what was happening during each iteration of the algorithms. This also made debugging a lot easier because I was able to see where my code was inconsistent. Wherever I didn't know what my code was doing or why an algorithm was not sorting, I tried to do it by hand and followed each iteration and command by paper. I also experimented with the sorting algorithms by inserting different array sizes and different seed sizes. Overall, this was a fun code to play with and it was very rewarding once I was able to understand the coding algorithms.

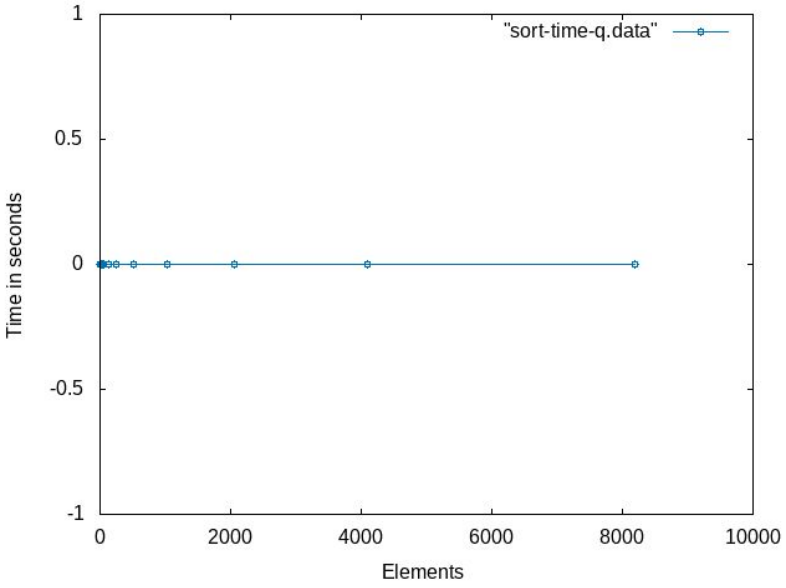
4.) Graphs



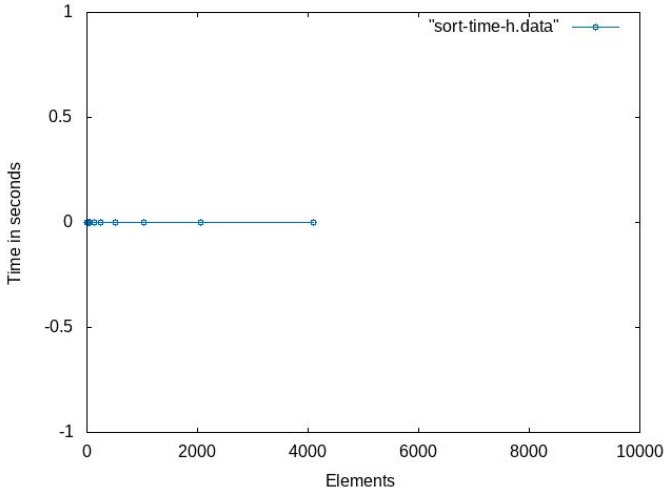
As this graph demonstrates, for a bubble sort algorithm the larger the constant and array are, the slower it sorts things. We can conclude from this graph that if we want to sort a large array, we would not use bubble sort as it would be inefficient to do so. I was only able to graph 10000 elements because my computer is very old and would crash whenever I tried to sort large numbers. Nonetheless, 10000 elements are enough to observe how changing the constant affects bubble sort. It took around 0.33 seconds to perform 10000 sorts.



As this graph demonstrates, for a shell sort algorithm the larger the constant and array are, the slower it sorts things. We can conclude from this graph that if we want to sort a large array, we would not use shell sort as it would be inefficient to do so. I was only able to graph 10000 elements because my computer is very old and would crash whenever I tried to sort large numbers. Nonetheless, 10000 elements are enough to observe how changing the array size affects shell sort. Unlike bubble sort, this is a pretty linear graph because the constant doesn't affect the shell's sort complexity as much. It took around 18 seconds to sort 10000 elements. It took longer to sort things with a shell sort algorithm than with a bubble sort algorithm, but that's probably because I did not use large numbers.



As this graph demonstrates, for a quicksort algorithm the larger the constant and array is, it doesn't have a huge effect on the performance of the algorithm. Quick sort was very consistent and took around 0 seconds to sort 10000 elements. We can conclude from this graph that if we want to sort a large array, we would want to use quick sort because it is very efficient. I was only able to graph 10000 elements because my computer is very old and would crash whenever I tried to sort large numbers. If I were to sort larger numbers I would see a slight change as I am bound to get the worst case complexity for quicksort at some point.



As this graph demonstrates, heap sort was a very fast sorting algorithm. Heap sort doesn't show that they sorted as many elements as the other algorithms because it uses different methods to sort. I was not able to graph a larger array because my computer would crash, but nonetheless from this graph we can conclude that heap sort is an efficient sorting algorithm. If I wanted to sort a large array, I would probably want to use a heap sort. This algorithm was able to sort 10000 elements in around 0 seconds. Thus demonstrating that the size of the array and constant will not have a huge effect on its time complexity.

Source for graphs: A student, Miles posted a plot .sh bash file we could run to achieve these graphs and I used that to create these graphs.

Although I really wanted to graph all the algorithms at the same time, I would have been super cool to see them all in one singular graph, I have a mid 2012 Macbook Pro that wouldn't let me do that. Nonetheless, from all the graphs we can see that heapsort performed the best, followed by quicksort, then it was shell sort, and lastly it was bubble sort.