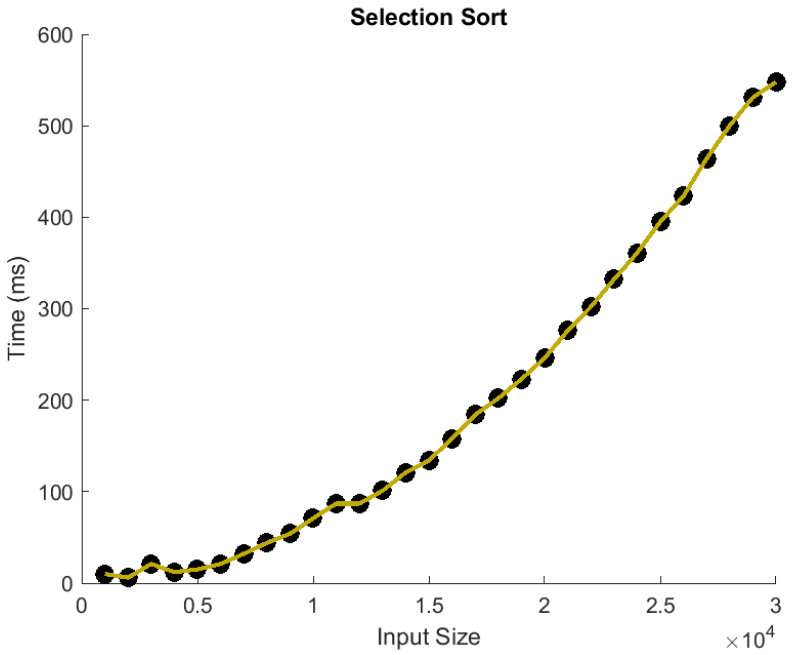
Sorting Methods Analysis

Allow me to preface the analysis by stating that diving into the details and implementing the various sorting algorithms by hand and getting them to work properly was a very satisfying experience. Programming them from scratch provided me with a much deeper level of understanding than I had from simply reading the theory and descriptions of the algorithms. In fact I wish we had done more sorting algorithms, and generally more implementations of data structures and algorithms, including sets, maps, and others.

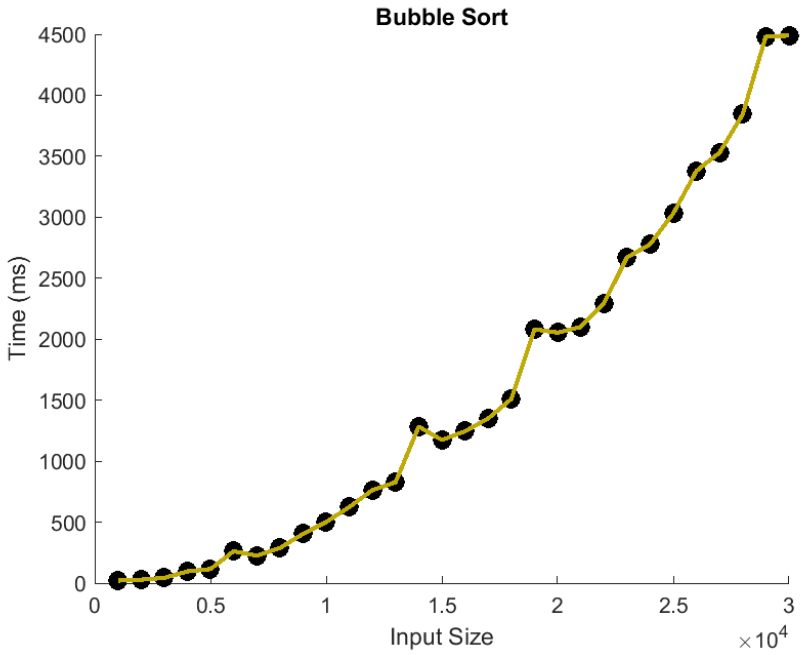
The seven sorting algorithms we implemented in this class consist of three that are $O(n^2)$, and four that are $O(n \log n)$. One of the $O(n^2)$ algorithms is Selection Sort, which I think is the most intuitive algorithm that most people would come up with to sort a list. If the list has length n, make $n-1$ passes of the list, and for each pass find the minimum value that is at or after the current position, and swap the minimum with the value at the current position. Since we have two nested loops each of which execute on the order of n times, the algorithm is $O(n^2)$.

All the figures in this analysis were generated by MATLAB, but the code is omitted because the emphasis is on the visualizations of the performances of the various algorithms. For the $O(n^2)$ algorithms, we used list sizes that ranged from 1,000 to 30,000 in increments of 1,000, while for the $O(n \log n)$ algorithms we used list sizes that ranged from 1,000 to 100,000 in increments of 1,000. For each sorting algorithm and for each input size k, we generated a list that contains k random integers in the interval [0, 100,000). The result of the Selection Sort simulation is shown in the following figure:

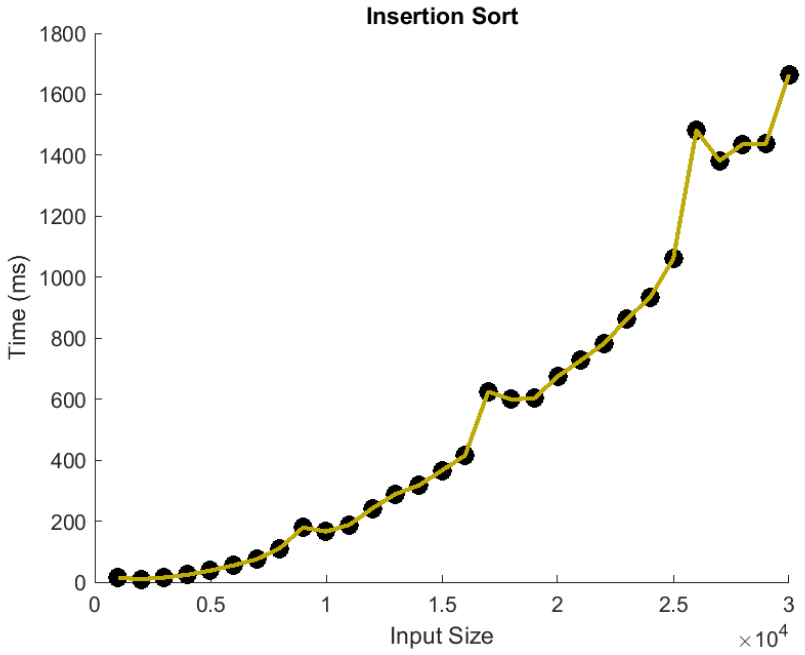


The next $O(n^2)$ algorithm we discuss is Bubble Sort. Again $n-1$ passes are made, and in each pass the largest element yet to be encountered "bubbles" to the top. Since we have nested loops each of which execute on the order of n times, this algorithm is also $O(n^2)$. One optimization that can be made is that at the start of each pass, a flag can be set to detect whether an exchange is made in that pass. If no exchanges were made, then the list must

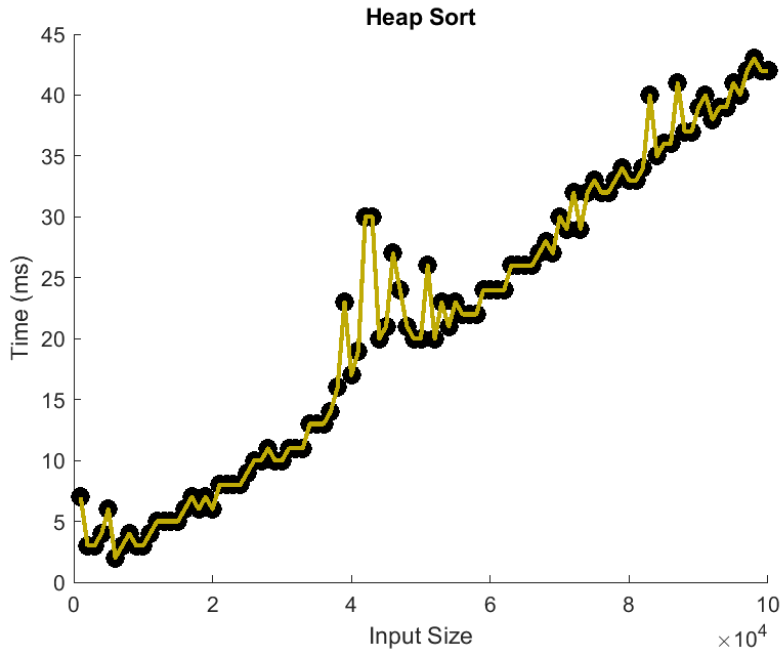
be sorted, and we need not continue with further passes. Despite this possibility, the algorithm's time complexity remains $O(n^2)$. Following is a visualization of the running time for Bubble Sort:



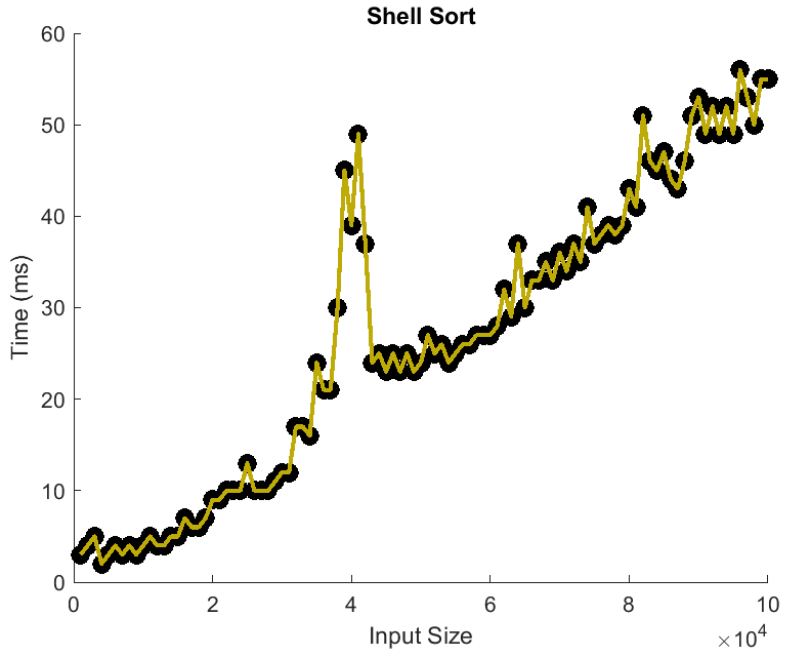
The last $O(n^2)$ algorithm in this analysis is Insertion Sort, apparently named after a popular method that people use to sort the playing cards in their hand. [Incidentally, I have spent many years playing various card games, including Texas Hold'em which allots two cards per player, Omaha which allots four, five, or six cards depending on the variant, and Seven Card Stud which allots up to a maximum of seven cards. Very rarely did I encounter players who actually resorted to a physical insertion sort.] Again $n - 1$ passes are made and in each pass the correct insertion position is determined for the element under consideration. A visualization of the running time for Insertion Sort follows:



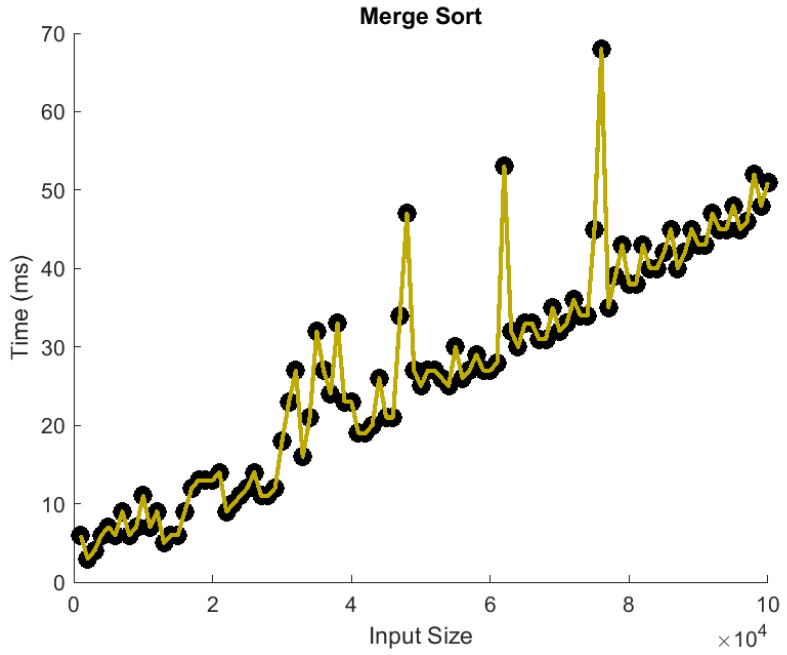
We move now the the $O(n \log n)$ algorithms. My favorite one of these, and probably of all the algorithms I have to date encountered, is Heap Sort. The sorting itself follows immediately once a min heap is available. Therefore if we have an unsorted list, we create a min heap from the list, and then one element at a time we remove elements from (the top) of the heap and add it to a different list. Since a min heap always maintains the minimum element in the heap at the top of the heap (this is accomplished by fixing the heap after each removal), when the heap is empty the resulting list is sorted. While we do not discuss heaps in more detail, we note that the algorithms for insertion into and removal from a heap are particularly pleasing. A visualization of the running time for Heap Sort follows:



The next algorithm in our analysis is Shell Sort, an algorithm that has maintained its status over the last 60 years, through the vast improvements in computer science theory and exponential increase in computing power. Shell Sort is essentially an improved version of Insertion Sort. Shell Sort's improvement is that it first sorts many smaller sublists of the original list using Insertion Sort, and merges these smaller lists together. In each iteration of the algorithm, a *gap* value is used, the details of which are unimportant here. However, we do note that a Shell Sort with an initial gap of 1 is exactly the same as the original Insertion Sort. A visualization of Shell Sort follows:



The next algorithm in our analysis is Merge Sort, an efficient algorithm that is rather intuitive. Merge Sort relies on recursion. Given an unsorted list, the idea is to split the list into two halves and recursively sort each half using Merge Sort. After this is accomplished, the two halves are merged together into a single list. If the two halves are themselves sorted, then the resulting list after the merge will also be sorted. Since this is a recursive algorithm, a terminating condition is needed, and this condition is that a list of size 1 or smaller is considered to be sorted. The splitting of the list into halves at each stage is an $O(\log n)$ operation, but the merging is $O(n)$. Hence the algorithm is $O(n \log n)$. A visualization follows:



The final algorithm in our analysis is Quick Sort, which is neat but not very intuitive. The idea is to take the unsorted

list and *partition the range*, which involves choosing a value x (known as the *pivot*) and placing it into a position such that all elements to the left of x are $\leq x$ and all values to the right of x are $\geq x$. Once this is accomplished, the sublists to the left and right of x are themselves recursively sorted using Quick Sort, which results in the entire original list being sorted. Once the pivot is established, the list is split into halves, but establishing the pivot is $O(n)$. Hence the algorithm is $O(n \log n)$. A visualization follows: