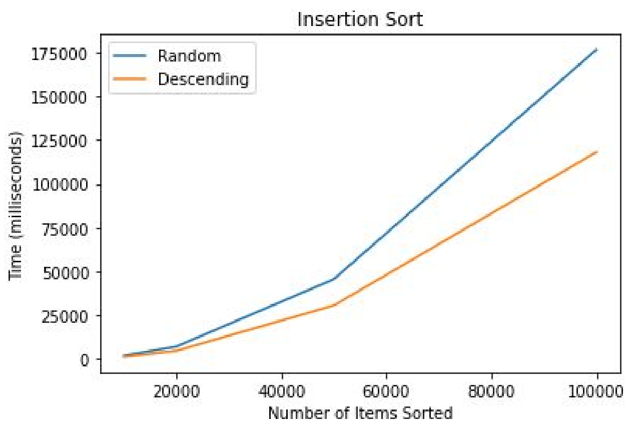
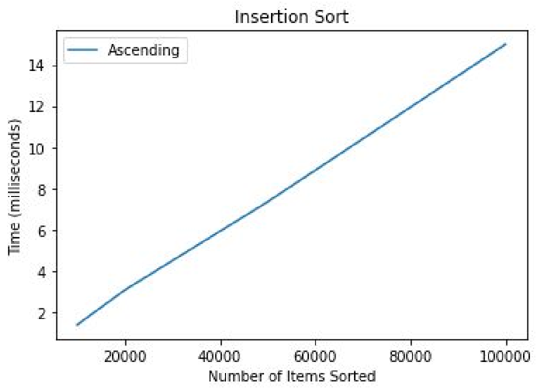
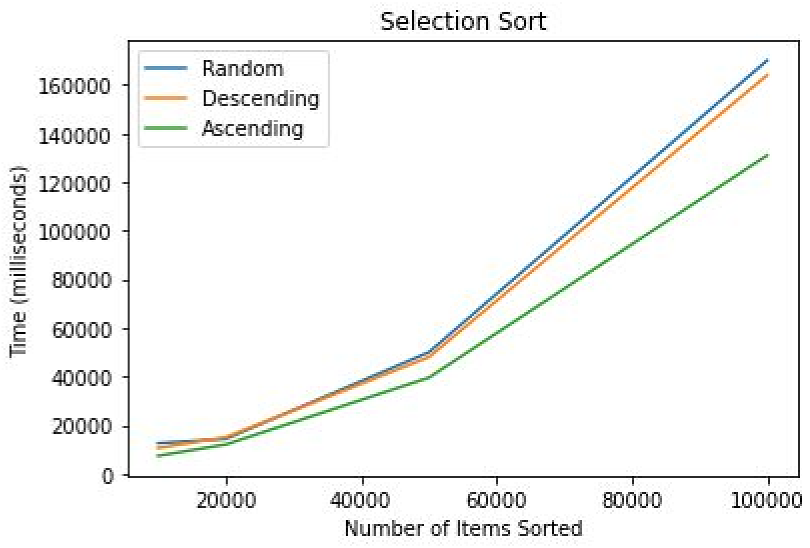
In my sorting analysis, merge sort, insertion sort, and selection sort were each tested for a total of 12 times, over four input files of sizes 10000, 20000, 50000, and 100000 integers. Each file size had three versions: random order, descending order, and ascending order. I called each sorting method for each file size five times, taking the average of the time elapsed for each trial.

The merge sort algorithm performed significantly faster than both selection and insertion sort, especially for larger input array sizes. Even in the times of smaller-sized arrays merge sort still beats selection and insertion sort to time. Between selection and insertion sort, insertion sort finishes faster than selection sort for nearly all test cases, however they share the same computational complexity: O(n^2). We can see this trendline growing exponentially for both sorting algorithms as we introduce test cases of greater sizes. Unlike merge sort, whose average time complexity is O(nlog(n)), insertion and selection sort suffer from exponential time complexity in the worst case, so their techniques begin to slow down substantially as the input size increases.

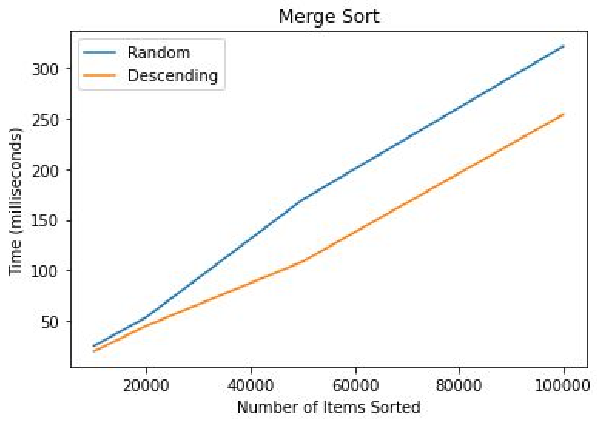


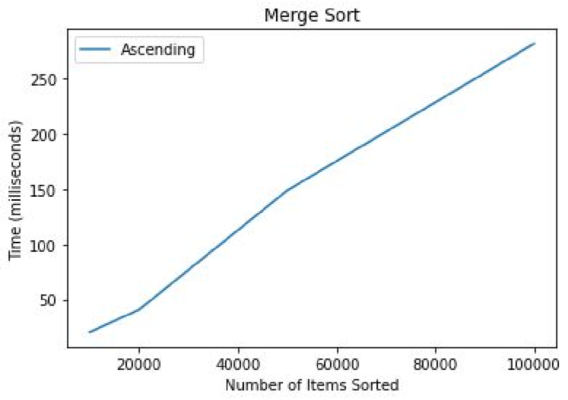


When we look at insertion sort, we can notice that the order of increasing computational complexity is as follows: ascending, descending, and random. Its best complexity is O(n), in the case that the numbers are in ascending order. This reduces time because no swaps are made between two adjacent numbers. In both random and descending order, the algorithm must check preceding numbers in the sorted section of the array to insert a number from the unsorted section, thus causing the O(n^2) complexity. We derive this from the array size, n, and the formula as follows: O(n + (n - 1) + (n - 2) + … + 1) = O(n(n+1)/2) = O((n^2)/2) = O(n^2). A larger file size results in an exponentially larger run time, as shown in the plots.



Looking at selection sort, we can notice that the order of increasing computational complexity is as follows: ascending, descending, and random. Descending order is slightly faster than random order because the algorithm finds fewer instances of a minimum value when comparing to the first item in the list. After it looking for the minimum at all n numbers, it repeats n - 1, n - 2, …, 1 times. Thus, the computational complexity for the best, average, and worst cases is O(n^2). A larger file size results in an exponentially larger run time, as shown in the plots. Ascending order must be the fastest case because the algorithm never finds a value smaller than the current value assigned at incremented indeces. No swaps take place, but the complexity is still O(n^2) due to the need to check values in the array about n^2 times. The algorithm can be faster by preventing any additional checks after the first searches fail to find a minimum value, thus proving the list to be pre-sorted. This case is rare, in that every subsequent item must be larger than the previous. Selection sort is not frequently used in computers because of this and the terrible computational complexity of O(n^2).





In the plots for merge sort, we can see that the order of increasing computational complexity is as follows: ascending, descending, and random. Descending order is faster than random order because within the algorithm, the time is takes to sort smaller subarrays is quicker when partitioning higher and lower integers. Merge sort is done recursively, breaking down the array into smaler subarrays. This divide and conquer method speeds up the process logarithmically. In making sure those subarrays themselves are sorted, the algorithm merges two unsorted lists, which takes time proportional to the number of elements in the array. Therefore, the time complexity combines these two to ends up with O(nlog(n)), visiting n items in the while loop and the maximum height of a binary tree being log(n). The cases for ascending order prevents swaps or sorting of arrays from happening, so they perform faster than the descending and random cases. In the end, the log-linear time for merge sort results in a much faster computation than both insertion and selection sort, no matter the size of the array inputted.