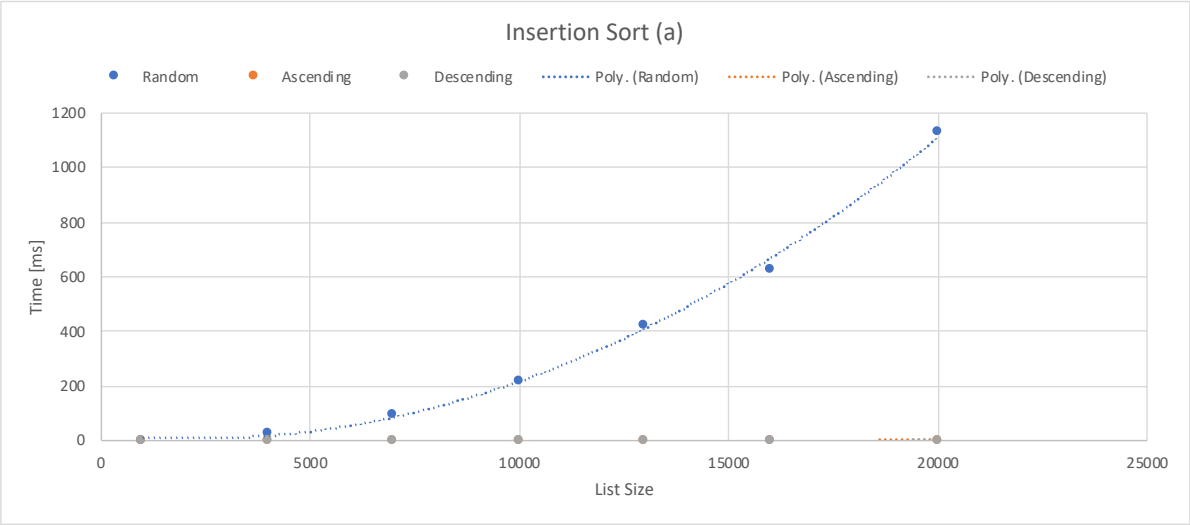
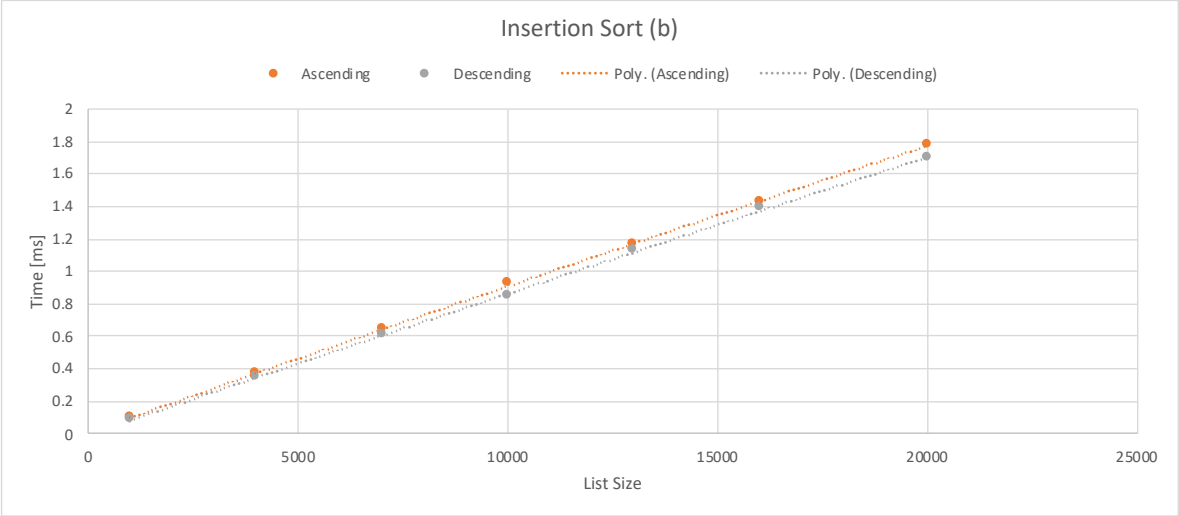
MP3 Test Log
Cameron L. Burroughs (burrou5)
Dr. Russel Hubbard
ECE 2230, Fall 2020

Performance Charts:

Graph 1a



Graph 1b



Graph 2

Graph 3

Graph 4

Evaluation:

- For lists that are initially random, explain the differences in running time for the sorting algorithms.

  My iterative and recursive selection sort algorithms had extremely similar run times throughout all list sizes and list types. For the random and ascending lists, their run times were nearly identical, as seen in the graphs above, while for the descending lists, iterative selection sort was slightly faster for larger lists. For all three list types, merge sort had significantly faster run times. While the other sort types had a run time of over one second at a list size of 20,000, merge sort didn't take longer than one second to run until it was given a list size of at least 500,000. The merge sort algorithm causes a dramatic increase in run time because it has to make far fewer comparisons to sort the lists than the other algorithms. With the merge sort algorithm, the initial list continues to be split in two until there is only one value in each sub list. As the smaller lists merge back into one large list, it is known that each smaller list is already sorted. To then merge each smaller list, the function must only compare the element until it finds the first spot in the list that element is larger/smaller than (depending on sort order). With the other algorithms, i.e. insertion sort and the two selection sorts, each new element added/sorted must be compared to almost every other element in the list to ensure that there is not another element that should be further up the list.

- Describe which algorithm(s) show extremely fast performance if the list is already sorted and explain why.

  The insertion sort algorithm shows extremely fast performance with lists than are already sorted in either ascending or descending order. This is due to the use of the llist_insert_sorted function. In llist_insert_sorted, it first checks to see if the element belongs either before the first element in the list or after the last element in the list. Then, if it belongs somewhere in between, it goes through the long process of finding which position the element should be inserted into the list. Because the lists are already sorted, each new insertion will either be to the front or back of the list, depending on if the sorted order of the initial list is the same or opposite of the sorting order. Graph 1a shows the relationship between the insertion sort and different list types, where you can see that the random list runtime increases quickly to one second, whereas the ascending and descending list types have run times that remain close to zero. Graph 1b shows more precisely the run times and relationship between insertion sort and the ascending and descending list types.