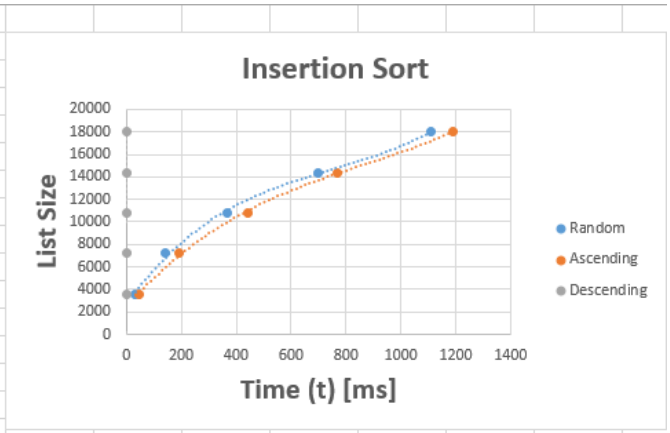
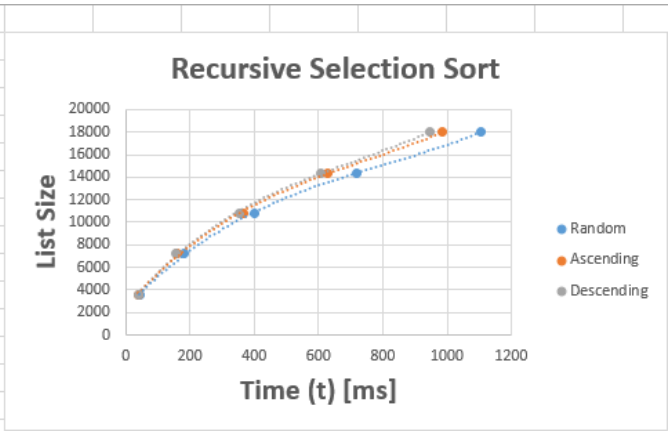
Ryan Barker
ECE 2230 Fall 2014

MP3 Test Log

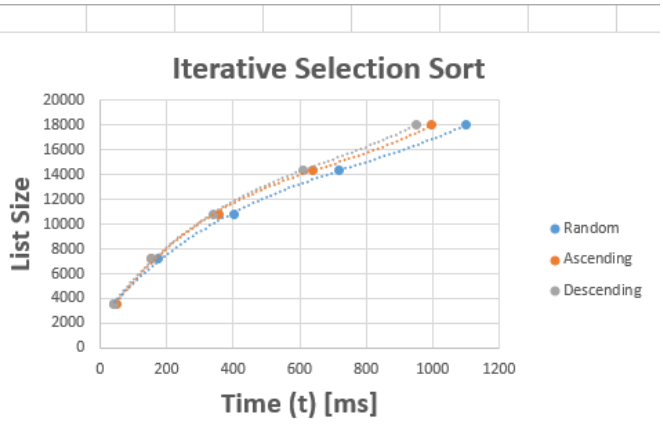
List Size	Random [ms]	Ascending [ms]	Descending [ms]
3600	32.351	47.626	0.135
7200	143.355	190.376	0.269
10800	368.012	441.84	0.574
14400	699.397	768.633	0.735
18000	1108.721	1190.363	0.662



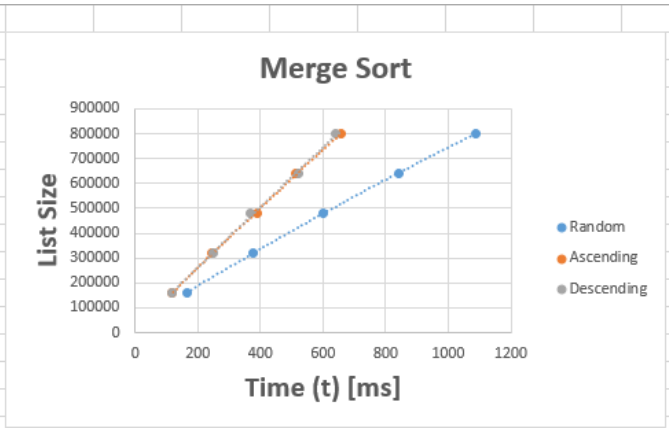
List Size	Random [ms]	Ascending [ms]	Descending [ms]
3600	42.064	39.114	38.578
7200	180.381	158.668	155.05
10800	399.89	366.396	353.286
14400	717.229	628.36	606.29
18000	1105.401	985.724	946.141



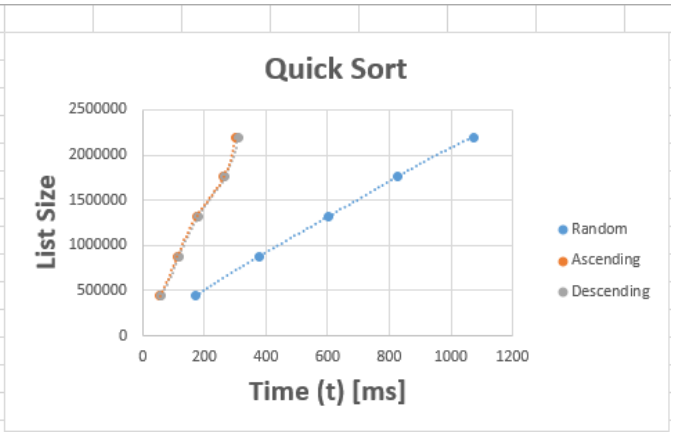
List Size	Random [ms]	Ascending [ms]	Descending [ms]
3600	41.725	49.542	40.284
7200	175.743	154.374	154.896
10800	401.918	355.398	342.068
14400	719.308	638.453	608.368
18000	1099.274	995.602	950.945



List Size	Random [ms]	Ascending [ms]	Descending [ms]
160000	163.915	118.809	117.139
320000	375.506	244.135	247.291
480000	598.893	388.649	369.636
640000	839.967	514.403	521.689
800000	1086.169	656.068	638.847



List Size	Random [ms]	Ascending [ms]	Descending [ms]
440000	171.348	55.351	57.916
880000	377.793	112.208	118.123
1320000	601.707	173.56	179.41
1760000	824.728	262.681	266.917
2200000	1072.214	303.282	310.372



Discussion:

a) Of the sorting algorithms, quick sort was the fasted, which makes sense, as it takes advantage of a C library function specifically for sorting data. It could sort approximately 2.2 million randomly ordered items in one second. The next fastest was merge sort, which was dramatically faster than the following algorithms because it broke down the problem of sorting the list into a smaller problem by halving the list, sorting the individual pieces, and combining the sorted lists back together. It clocked at 800,000 randomly ordered items in one second. All of the other algorithms clocked at about the same rate, which was dramatically slower than merge or quick sort (About 18,000 randomly ordered items in one second). Note that these algorithms include the iterative and recursive versions of selection sort, which clocked at about equal rates because the iterative version of the algorithm still has to compare each element in the list to everything proceeding it like the recursive version, so it does not save very much time or efficiency over the recursive version.

b) Some algorithms are very efficient when passed lists already sorted in ascending or descending order. These include:

   1. Insertion Sort with Descending Lists: Note that insertion sort works by removing the head of the list and inserting it into the new, sorted list with the sorting algorithm from

MP2. So, when a descending list is sorted, the first item, which is the largest item in the list, is removed and inserted into the sorted list. Then the next smallest item is removed and inserted, and so on: Since each item is guaranteed to be less than all of the items already in the sorted list, the sorting algorithm doesn't have to do any work. It can just keep inserting records in the head of the sorted list.

2. Merge Sort with lists in either order: As mentioned, merge sort works by splitting the input list in halves, sorting those halves, and merging those halves back together into a sorted list. When the list passed is already in ascending order, all Merge Sort has to do is take it apart and put it back together, which takes a reduced amount of time compared to a random list. When it is descending order, all Merge Sort has to do is flip the list around, but because it is already split in pieces by the algorithm, this takes about the same amount of time as sorting an ascending list.
3. Quick Sort with lists in either order: Despite being fast with any type of list, because it takes advantage of a C library function to sort the input list, Quick Sort is especially fast with a list that is already sorted. This is because the library function recognizes that a list already in ascending order (which it can find out by doing very fast comparisons between each item since they are in an array rather than a linked list) doesn't need to be sorted and a list in descending order simply needs to be flipped around.