Selection Sort		
List Size	**Comparisons**	**Time (seconds)**
1,000 (observed)	499500	0.03147482872009277s
2,000 (observed)	1999000	0.10160493850708008s
4,000 (observed)	7998000	0.4019458293914795s
8,000 (observed)	31996000	1.6131367683410645s
16,000 (observed)	127992000	6.483428239822388s
32,000 (observed)	511984000	25.686216831207275s
100,000 (estimated)	~4100000000	~205.488s
500,000 (estimated)	~98304000000	~4931.712s
1,000,000 (estimated)	~393216000000	~19726.848s
10,000,000 (estimated)	~33553593000000	~1682936.857s

Insertion Sort		
List Size	**Comparisons**	**Time (seconds)**
1,000 (observed)	247986	0.029107093811035156s
2,000 (observed)	1018717	0.10313916206359863s
4,000 (observed)	3995264	0.40640807151794434s
8,000 (observed)	16112194	1.6432170867919922s
16,000 (observed)	64667449	6.549669981002808s
32,000 (observed)	257507119	26.2252836227417s
100,000 (estimated)	~2050000000	~209.801s
500,000 (estimated)	~49152000000	~5035.216s
1,000,000 (estimated)	~196608000000	~20140.864s
10,000,000 (estimated)	~16776797000000	~1718257.389s

1. Which sort do you think is better? Why?
   I think it depends on what someone is looking for, but overall insertion sort is more stable and efficient. The benefit of insertion sort is that it makes less comparisons, so if the list wasn't completely random, then insertion would perform better. So in modern practice, insertion sort is probably better overall as it makes less comparisons, and there are rare cases where a list is completely random, else selection sort may be better when it's a completely unsorted, random list.

2. Which sort is better when sorting a list that is already sorted (or mostly sorted)? Why?

Insertion sort is better for sorting a list that is mostly sorter because insertion sort only compares elements that are necessary. So if the list is mostly sorted, insertion sort performs nearly at O(n).

3. You probably found that insertion sort had about half as many comparisons as selection sort. Why? Why are the times for insertion sort not half what they are for selection sort? (For part of the answer, think about what insertion sort has to do more of compared to selection sort.)

Insertion sort makes half as many comparisons as selection sort because when there are portions of a list sorted, insertion sort will not make comparisons once it is sorted as the unsorted part stays put, as it builds the sorted half. Insertion sort is not half the time of selection sort because insertion sort has to generally swap more than insertion sort. One advantage of selection sort is that the number of swaps is O(n), while in insertion sort it is O(n^2). Both have generally the same average time complexity of O(n^2), but insertion sort's best case can be O(n) if the list is sorted.

	Number of Quicksort Comparisons	
Starting List	pivot = first	pivot = median of 3
Ordered, ascending		
n = 100	4950	480
n = 200	19900	1153
n = 400	79800	2698
n = 800	319600	6187
Random		
n = 100 (average 10 runs)	618	543
n = 200 (average 10 runs)	1576	1326
n = 400 (average 10 runs)	3645	3181
n = 800 (average 10 runs)	8403	7115
Observed Big O() behavior, ordered with pivot = first : O(n^2)		
Observed Big O() behavior, ordered with pivot = median of 3 : O(nlog(n))		
Observed Big O() behavior, random with pivot = first : O(nlog(n))		
Observed Big O() behavior, random with pivot = median of 3 : O(nlog(n))		
For random list, observation regarding using first vs. median of 3: First pivot and median of 3 pivots have roughly the same time complexity for a random list, but first pivot makes more comparisons as there's a chance of choosing the smallest or largest indexed element as a pivot versus in a median of 3 where the smallest or largest indexed element can never be chosen. In the median of 3 its best and worst case		

will be O(nlog(n)) unlike the first pivot where its worst will be O(n^2) and best at O(nlog(n)).