Lab 6

Selection Sort		
List Size	**Comparisons**	**Time (seconds)**
1,000 (observed)	499500	0.05937027931213379
2,000 (observed)	1999000	0.1435694694519043
4,000 (observed)	7998000	0.6412713527679443
8,000 (observed)	31996000	2.416534900665283
16,000 (observed)	127992000	11.144203186035156
32,000 (observed)	511984000	39.132041215896606
100,000 (estimated)	4,995,000,000	593.7027931213379
500,000 (estimated)	124,875,000,000	14,842.5698280334475
1,000,000 (estimated)	499,500,000,000	59,370.27931213379
10,000,000 (estimated)	49,950,000,000,000	5,937,027.931213379

Insertion Sort		
List Size	**Comparisons**	**Time (seconds)**
1,000 (observed)	247986	0.05340385437011719
2,000 (observed)	1018717	0.20047211647033691
4,000 (observed)	3995264	0.8332729339599609
8,000 (observed)	16112194	3.4565887451171875
16,000 (observed)	64667449	12.304732084274292
32,000 (observed)	257507119	58.21491360664368
100,000 (estimated)	2,479,860,000	534.0385437011719
500,000 (estimated)	61,996,500,000	13,350.9635925292975
1,000,000 (estimated)	247,986,000,000	53,403.85437011719
10,000,000 (estimated)	24,798,600,000,000	5,340,385.437011719

1. Which sort do you think is better? Why?

   For very large lists or very small lists, I think insertion sort is better because it took less time to analyze. For medium sized lists, I think selection sort is better because it takes less time to analyze those lists.

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 already or mostly sorted because the way it works is that it assumes that the first value is sorted and as it further analyzes the list, it assumes the next value is sorted. So, since the list is mostly or already sorted, it does not have to do much modifications.

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.)

    Selection sort works "backwards" when sorting. Meaning that it finds the largest value and puts it the back of the list and only has to interpret the rest of the list to find the next largest value and put that in the slot before the previously inserted number. This eliminates redundancy in analyzing the already "organized" section of the list. Insertion sort, however, has this redundancy. When it receives a new value, it must check the assumed already sorted section, to find where to place it. When it gets a new value, it repeats this process, causing redundancy in its analysis.

	Number of Quicksort Comparisons	
Starting List	pivot = first	pivot = median of 3
Ordered, ascending		
n = 100	4950	4950
n = 200	19900	19900
n = 400	79800	79800
n = 800	319600	319600
Random		
n = 100 (average 10 runs)	625.1	717.7
n = 200 (average 10 runs)	1587	1740.7
n = 400 (average 10 runs)	3560.8	3674
n = 800 (average 10 runs)	8339.2	8934.1
Observed Big O() behavior, ordered with pivot = first : O(n^2)		
Observed Big O() behavior, ordered with pivot = median of 3 : O(n^2)		
Observed Big O() behavior, random with pivot = first : O(nlogn)		
Observed Big O() behavior, random with pivot = median of 3 : O(nlogn)		
For random list, observation regarding using first vs. median of 3: For smaller lists, quicksort with pivot = first is better. For larger lists, pivot = median of 3 is better. While it doesn't show in our data, the number of comparisons does not have as large of a jump between each increment of numbers in the list. If we were to increase the number of items in the list, the amount of comparisons in pivot=first would surpass the pivot=median of 3.		