Lab Week 6: Basic Comparison Sorts

A comparison sort is a type of sorting algorithm that compares elements in a list (array, file, etc) using a comparison operation that determines which of two elements should occur first in the final sorted list. The operator is almost always a **total order**:

1. $a \leq a$ for all a in the set
2. if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity)
3. if $a \leq b$ and $b \leq a$ then $a=b$
4. for all a and b, either $a \leq b$ or $b \leq a$ // any two items can be compared (makes it a total order)

In situations where three does not strictly hold then, it is possible that and b are in some way different and both $a \leq b$ and $b \leq a$; in this case either may come first in the sorted list. In a **stable sort**, the input order determines the sorted order in this case.

The following link is helpful.

- Comparison Sorting Visualizations: https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Your goal for this lab is to implement simple versions of Insertion Sort - **insert_sort(alist)**, Selection Sort - **select_sort(alist),** and Merge Sort Sort - **merge_sort(alist)** that will sort an array of integers and **count the number of comparisons**. Each function takes as input a list of integers, sorts the list counting the comparisons at the same time, and returns the number of comparisons. After the function completes the "**alist**" should be sorted.

The worst-case runtime complexity is $\Theta(n^2)$ for selection and insertion sort. Why? Write out the summation that represents the number of comparisons.

Note: There is a fundamental limit on how fast (on average) a comparison sort can be, namely $\Omega(n*\log n)$. We will discuss the reasons for this in class.

In addition to submitting your code file (**sorts.py**) and associated test file (**sorts_tests.py**) for **insert** and **selection** sorts, fill out and submit a table similar to the table below as well as answers to the questions below to Canvas. **Note: Do not submit Mergesort but be sure you can reproduce the algorithm and understand it in detail.**

Note: The list sizes designated as (**observed**) should be based on actual runs of your code. The list sizes designated as (**estimated**) should be estimates you make based on the behavior of the sorting algorithm and the times from actual runs with fewer elements.

Selection Sort		
List Size	**Comparisons**	**Time (seconds)**
1,000 (observed)		
2,000 (observed)		
4,000 (observed)		
8,000 (observed)		
16,000 (observed)		
32,000 (observed)		
100,000 (estimated)		
500,000 (estimated)		
1,000,000 (estimated)		
10,000,000 (estimated)		

Insertion Sort		
List Size	**Comparisons**	**Time (seconds)**
1,000 (observed)		
2,000 (observed)		
4,000 (observed)		
8,000 (observed)		
16,000 (observed)		
32,000 (observed)		
100,000 (estimated)		
500,000 (estimated)		
1,000,000 (estimated)		
10,000,000 (estimated)		

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

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

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 does more of compared to selection sort.)