Lab 6: Part A 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 a 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 shows visualization of some common sorting algorithms:
https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Your goal for this lab is to implement simple versions of Insertion Sort - **insertion_sort(alist)**, and Selection Sort - **selection_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 $\Omega(n*\log n)$.

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 the GitHub repository. Note: The list sizes with (observed) noted should be based on actual runs of your code. The list sizes with (estimated) noted 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 has to do more of compared to selection sort.)

Lab 6: Part B Quicksort

Quicksort is a “Divide and Conquer” algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of Quicksort that pick the pivot in different ways, including:

1. Always picking the first, middle or last element as pivot.
2. Picking a random element as pivot.
3. Picking a median (usually from 3 elements) as pivot.

The key process in Quicksort is partition(). When the partition() function is called with an input array, it will determine a pivot (in our case, either using the first element, or the median of 3). When partition() completes it work (in linear time), the elements will have been rearranged such that the pivot element is in the correct location, and all elements "less than" the pivot element are located at indices less than that of the pivot index, and all elements "greater than" the pivot element are located at indices greater than the pivot index.

You have been provided code that implements the Quicksort sorting algorithm using the first element as the pivot (when PIVOT_FIRST is True). You will need to write the code that will use the median of three method to select the pivot (set PIVOT_FIRST to False).

Submit your code (quicksort.py and quicksort_tests.py) and a pdf of the following table, with your observations recorded to the GitHub repository.

	Number of Quicksort Comparisons	
Starting List	pivot = first	pivot = median of 3
Ordered, ascending		
n = 100		
n = 200		
n = 400		
n = 800		
Random		
n = 100 (average 10 runs)		
n = 200 (average 10 runs)		
n = 400 (average 10 runs)		
n = 800 (average 10 runs)		
Observed Big O() behavior, ordered with pivot = first :		
Observed Big O() behavior, ordered with pivot = median of 3 :		
Observed Big O() behavior, random with pivot = first :		
Observed Big O() behavior, random with pivot = median of 3 :		

For random list, observation regarding using first vs. median of 3 :