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.

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 like the table below as well as answers to the questions below to Canvas as a separate pdf file **SortAnalysis.pdf**. **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– Random Data		
List Size	Comparisons	Time(seconds
1,000 (observed)	499500	0.0538566112518310
2.000 (observed)	1999000	0.2522966861724853
4,000 (observed)	7998000	0.8572187423706055
8,000 (observed)	31996000	3.4542062282562256
16,000 (observed)	127992000	13.185082197189331
32,000 (observed)	511984000	53.47231721878052
100,000 (estimated)	5x10^9	527
500,000 (estimated)	1.25x10^11	13400
1,000,000 (estimated)	5x10^11	54,020
10,00,000 (estimated)	5x10^13	5.5x10^6

Insertion Sort – Random Data		
List Size	Comparisons	Time(seconds
1,000 (observed)	248534	0.0987358093261718
2.000 (observed)	992765	0.4452888965606689
4,000 (observed)	4007011	1.7384116649627686
8,000 (observed)	16179647	6.43133544921875
16,000 (observed)	64770773	27.100253582000732
32,000 (observed)	255898692	108.54739809036255
100,000 (estimated)	2.4555x10^9	1,070
500,000 (estimated)	5.985x10^10	27,187
1,000,000 (estimated)	2.370x10^11	109,702
10,00,000 (estimated)	2.288x10^13	1.114x10^7

1. 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.

Selection sort:

For selection sort the worst case time complexity is n squared as it has to iterate over rest of the array every time to find the minimum or maximum value. Summation for insertion sort is as follows $\sum_{i=0}^{n-1} n - i - 1$

Insertion sort:

For insertion sort the worst case is when the list is in the opposite direction to the desired direction so it has to iterate over the sorted list portion fully for every iteration.

Summation is as follows $\sum_{i=1}^{n} i$

2. What is the worst case for insertion sort? What is the worst case for Selection Sort?

For selection sort there is no worst case it is always n^2 , and insertion sort the worst case is when the list is in reverse order.

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

Insertion sort is better as it will have less comparisons on all but the worst case.

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

Insertion sort is better for this because it will have to do significantly less comparisons. Closser to n comarisons

5. 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?

My selection sort uses the min() function witch is implemented in C witch is drastically faster than python. Also there are a differing amount of swaps and other bits of code that are run witch alter the loop times between the two algorithms. An optimal implementation in a compiled language would probably result in insertion sort being faster.