Lab 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. $a \le a$ for all a in the set 2. if $a \le b$ and $b \le c$ then $a \le c$ (transitivity) 3. if $a \le b$ and $b \le a$ then $a=b$ 4. for all a and b, either $a \le b$ or $b \le a$ // 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 $a \le b$ and $b \le a$; 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 Ω (n*log n).

Fill out and handin a table similar to the table below as well as answers to the questions below.

Selection Sort

List Size	Comparisons	Time (sec)
1000 observed	499500	0.15782762
2000 observed	1999000	0.6942358
4000 observed	7998000	2.450435
8000 observed	31996000	13.050435
16000 observed	127992000	47.46907411
32000 observed	511984000	232.2006376
100000 estimated	2.5E+10	1013.678992
500000 estimated	1E+12	3042.899582
1000000 estimated	1E+14	17266.3361
10000000 estimated	1E+16	21389.12834

Insertion Sort

List Size	Comparisons	Time (sec)
1000 observed	247986	0.1682024
2000 observed	973899	0.379494
4000 observed	3995264	4.292607784
8000 observed	16112194	15.47210193
16000 observed	64667449	67.49353596
32000 observed	257507119	90.1134555
100000 estimated	2600000000	280.1008105
500000 estimated	64600000000	1211.1199
1000000 estimated	2.57E+14	3287.07789
10000000 estimated	2.57E+16	19873.23498

1. Which sort do you think is better? Why?
Selection Sort is better for more compact data and in some cases large data because when it iterates through the entirety of the passed list, comparing the current minimum value to the next passed value. When it encounters a value smaller than the current minimum value, it swaps the places of those two values taking the least minimum value to the left and the greater value to the right. The algorithm still checks all the items in the list to make sure that no lower minimum value exists in the list, so if a list is passed already sorted, it will still iterate through the entire list to make sure no value is lower than the current minimum value it is set to.

2. Which sort is better when sorting a list that is already sorted (or mostly sorted)? Why?
Insertion sort is better because it only checks the next value being passed, so once it sees that the next value is greater than the current value it is comparing to, it will only compare those two values. If the list isn't sorted, it will compare the current minimum value to the next value, and iterates the left side of the list in reverse traversal by shifting the value in between where it's supposed to go.

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.)
If the list is already sorted, the comparisons cut down in half to when using an insertion algorithm will ignore swapping the two values and will continue to the next iteration instead of checking the rest of the list like the selection algorithm does. The time it takes to complete the sorting is longer for insertion sort because if the list is in random order it takes much more swapping around all the previous items and hence takes more time to complete, as where in selection sort checks all the items in the list but will only proceed to make one swap at a time and hence will take shorter time. Overall, insertion sort is handy for already sorted or semi-sorted lists, whereas selection sort is more efficient in time when sorting through random lists.