Name:______________________________ **Partner:** __________________________

Python Activity 51: Iterative Sorting

Ordered data makes finding data more efficient!

Learning Objectives
Students will be able to:
Content:
- Identify ***best case*** and ***worst case*** scenarios for sorting algorithms
- Describe the ***selection*** and ***insertion sort*** algorithms for sorting data
- Compare the timed run-times of algorithms

Process:
- Write code that implements ***selection sort*** and ***insertion sort***

Prior Knowledge
- Python concepts: searching algorithms

Concept Model:

CM1. The table below represents two approaches to *sorting* a deck of cards.

Sorting a Deck of Cards	
Look at each card & find the lowest: Swap it to the first unsorted position Repeat!	How would you sort a deck of cards?

a. What might be the *best case* for the approach on the left? ________________________

What might be the *average case* for the approach on the left? ________________________

What might be the *worst case* for the approach on the left? ________________________

b. Is the approach on the left how you typically sort a deck of cards? ____________

What is your typical approach?

____

____

What might be the *best case* for your approach? ______________________________

What might be the *average case* for your approach? ______________________________

What might be the *worst case* for your approach? __________________________

Is your approach more efficient than the one described on the left? ____________

c. When we add an additional n cards to our shuffled deck, how many additional operations might we have to complete to sort the deck with the approach on the left?

____

What if we add an additional n cards and use your sorting approach?

____

d. The sorting algorithm we describe on the left is called ***selection sort***. Why might that be an appropriate name for this algorithm?

____

Critical Thinking Questions:

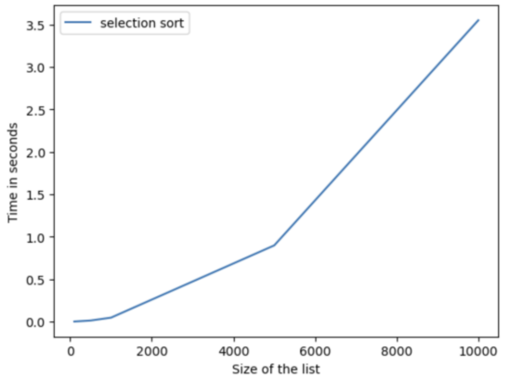
1. Examine the following code for *sorting* for an item in a list using ***selection sort*** (the algorithm on the left in CM1):

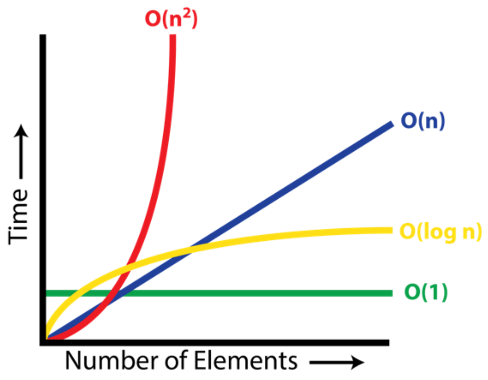
selection.py

```python
def selection_sort(target_list):
    # i. Comment?
    for i in range(len(target_list)):
        ii. Comment?
        min_index = i
        for j in range(i+1, len(target_list)):
            iii. Comment?
            if target_list[j] < target_list[min_index]:
                min_index = j
        iv. Comment?
        tmp = target_list[i]
        target_list[i] = target_list[min_index]
        target_list[min_index] = tmp
```

a. Add in-line comments to the code above where indicated (i. – iv.), explaining what the code beneath it is doing

b. Let's think about the relationship between number of elements and number of operations. Observe the plot below which shows the amount of time (y-axis) it takes to sort a shuffled list using ***selection sort***, as the number of items in that list increases (x-axis):

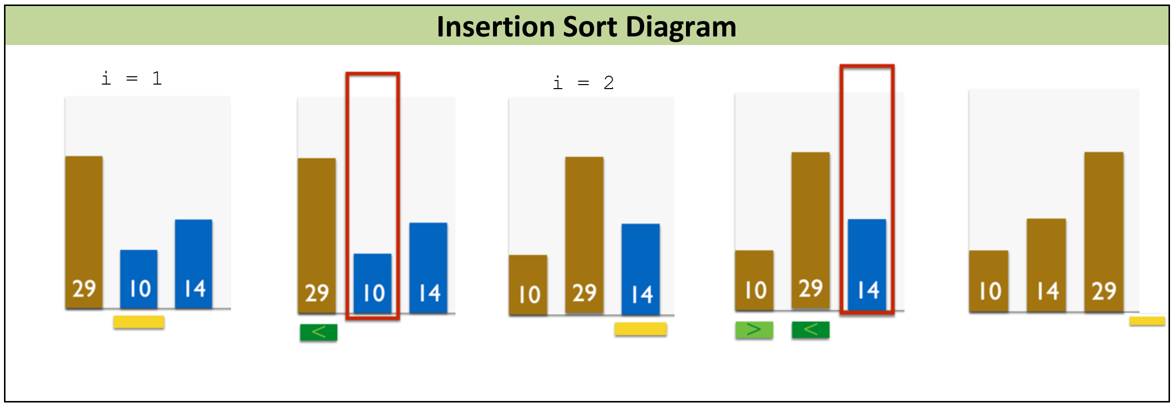




a. If you had to fit the empirical runtimes above to a more generalized runtime plot from the ones shown below, what would you you choose?

$O(n^2)$ or $O(n)$ or $O(\log n)$ or $O(1)$?

2. Examine the following diagrams of another approach to sorting a list, ***insertion sort***, with an example unsorted list of characters:



a. What is the *basic idea* of the diagram above? What is it trying to accomplish?

____

____

____

____

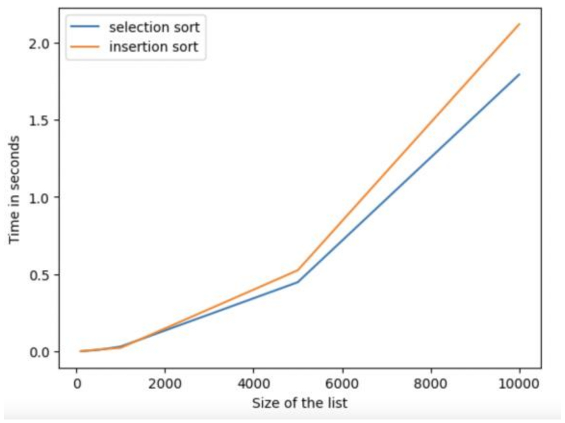
b. Step through the code, and explain what the following lines do:

`def insertion_sort(target_list):`	
`    for i in range(1, len(target_list)):`	
`        current_val = target_list[i]`	
`        j = i`	
`        while j > 0 and target_list[j-1] > current_val:`	
`            target_list[j] = target_list[j-1]`	
`            j = j - 1`	
`        target_list[j] = current_val`	

c. This algorithm is called ***insertion sort***. Why might that be?

____

3. Observe the plot below which shows the amount of time (y-axis) it takes to sort a *fully* shuffled list, as the number of items in that list increases (x-axis), for both ***insertion sort*** and ***selection sort***.



a. What might be the *best case* scenario for ***insertion sort***?

What might be the *worst case* scenario for ***insertion sort***?

b. When we add `n` additional elements to our list to be sorted, how many more operations/comparisons does ***insertion sort*** have to do?

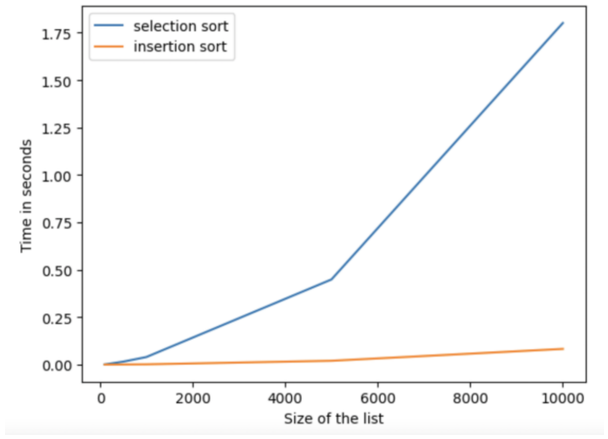
4. As in the previous question, ***insertion sort*** may appear to perform identically to ***selection sort***, but it has slightly more efficient behavior when the data has a certain property.

a. Under which scenario(s) might ***insertion sort*** perform better than ***selection sort***:
(*Hint: it might be helpful to think about the list* `[2, 5 , 1, 25, 50, 100]`)

When the data is in reverse sorted order
When the data is already sorted
When all the items are the same
When the data is only lightly shuffled

Why might ***insertion sort*** perform better in this situation?

b. Observe the plot below which shows the amount of time (y-axis) it takes to sort a *lightly* shuffled list, as the number of items in that list increases (x-axis), for both ***insertion sort*** and ***selection sort***.



Why might ***insertion sort*** perform so much better than ***selection sort*** for *lightly* shuffled lists, but not fully shuffled?

5. Observe the following session in interactive python below:

Interactive Python

```
>>> lst = [25, 17, 1, 39, 8]
>>> sorted(lst)
[1, 8, 17, 25, 39]
```

a. What might the `sorted(..)` built-in function do?

b. If we wanted to use the output from the `sorted(..)` built-in function, what would we have to do?

FYI: The python `sorted(..)` built-in function uses ***insertion sort*** for smaller lists (up to around length 50), and a faster sorting algorithm for larger lists.

Application Questions.

1. There's many more sorting algorithms, and we'll discuss some later in the semester. For now, consider the ***Bogo sorting algorithm**:* It reshuffles the list until it's sorted.

   a. Write some *pseudocode* that would implement this algorithm:

   b. What is the *best case* scenario for this algorithm?

   What is the *worst case* scenario for this algorithm?

   c. When you add `n` new items to the list to be sorted, how many additional operations/comparisons are required?

   d. Is this a "good" algorithm? Why/not?