Name:________________________________ **Partner:** __________________________

Python Activity 47: Sorting

More efficient searching & sorting algorithms, means more resourceful software!

Learning Objectives
Students will be able to:
Content:

- Identify ***best case*** and ***worst case*** scenarios for sorting algorithms
- Apply ***Big-O notation*** to measure the efficiency of additional algorithms
- Describe the ***selection*** and ***merge sort*** algorithms for sorting data
- Compare the timed run-times of algorithms

Process:

- Write code that implements ***selection sort*** and ***merge sort***

Prior Knowledge

- Python concepts: Big-O notation, searching algorithms, recursion, str.format()

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 position Repeat!	How would you sort a deck of cards?

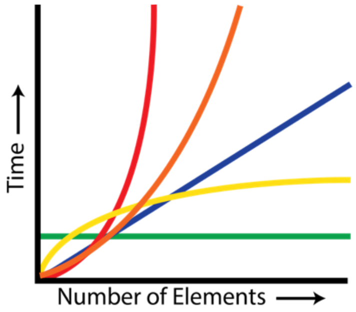
a. What might be the *best case* for the approach on the left? ________________________________
What might be the *worst case* for the approach on the left? __________________________

b. Is the approach on the left how you typically sort a deck of cards? ____________
What is your typical approach?
____
____
Is your approach more efficient than the one described on the left? ____________
What might be the *best case* for your approach? ________________________________
What might be the *worst case* for your approach? __________________________

c. What might be the run-time for the *first* sorting algorithm?
____
What might be the run-time for *your* sorting algorithm?
____

CM2. Let's think about the relationship between operations' number of elements and run-time:

a. Label the following graph with the run-times they represent:



Run-times:
O(1)
O(n)
O(log n)
O(n log n)
$O(n^2)$

b. Which of these run-times are new to us? ____________________

FYI: *Recall* that $O(\log_2 n)$ (***O(log n)*** to computer scientists), ***logarithmic run-times,*** indicate that if we double the number of elements, it only increases the number of operations by 1. Or, in other words, logarithms describe the number of times we can divide n by 2 until we get down to 1.

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

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

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

b. Which sorting algorithm is this most similar to from CM1? ____________________

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

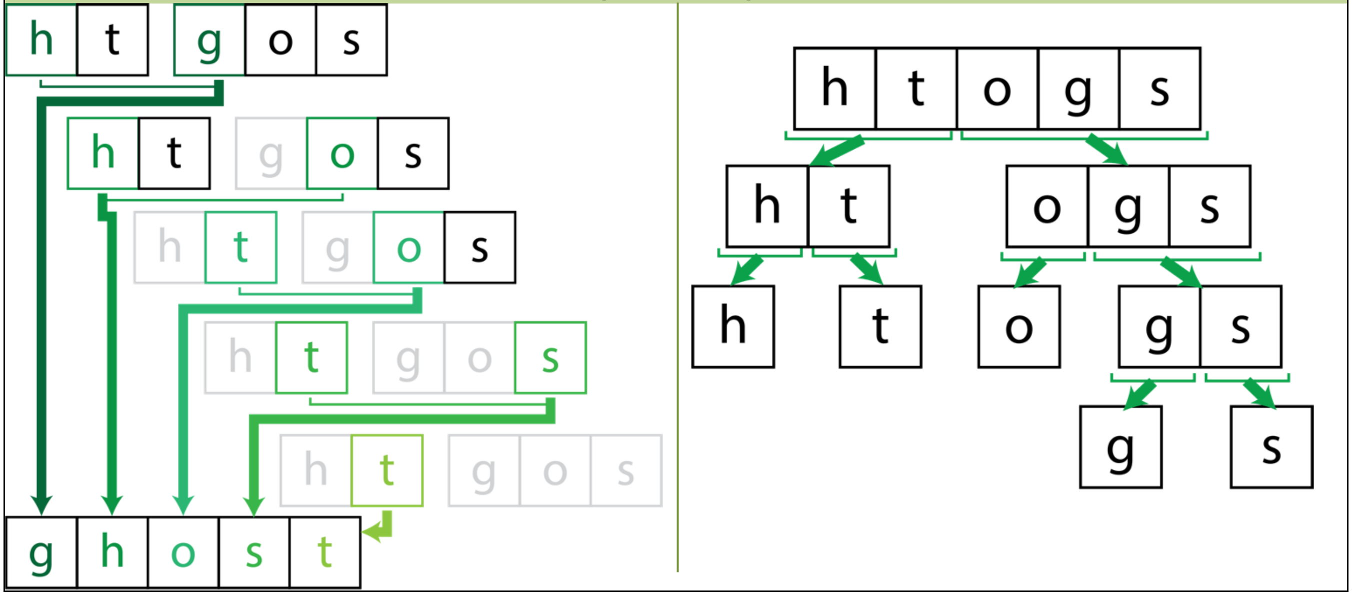
What is the ***Big-O notation*** run-time of this algorithm in the *best case*? O(________)

d. What is the *worst* case scenario for this algorithm? __________________________________

What is the ***Big-O notation*** run-time of this algorithm in the *worst case*? O(________)

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

Merge Sort Diagram



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

___

___

b. What is the *basic idea* of the diagram on the left? What is it trying to accomplish?

___

___

c. How do we sort a list of one element?

___

d. What is the *run-time* of the diagram on the right if there are ***n*** elements in the list?

___

What is the *run-time* of the diagram on the left if there are ***n//2*** elements in each list?

___

e. How might we combine these two algorithms, ***merge*** on the left and ***mergesort*** on the right in order to sort a given list of elements?

___

___

___

f. What is the *run-time* of this new algorithm? ___

3. Step through the code for `merge()` below, and explain what the following sections do:

`def merge(a, b):`	
`i, j, k = 0, 0, 0`	
`len_a, len_b = len(a), len(b)`	
`c = []`	
`while i < len_a and j < len_b:`	

`if a[i] <= b[j]:`
`c[k] = a[i]`
`i += 1`	

`else:`
`c[k] = b[j]`
`j += 1`	

`k += 1`	

`if i < len_a:`
`c.extend(a[i:])`	

`elif j < len_b:`
`c.extend(b[j:])`	

`return c`	

4. Examine the following partially complete code for *mergesorting* a list:

mergesort.py

```
def merge_sort(lst):
    n = len(lst)

    # Comment:
    if n == 0 or n == 1:
        return lst

    else:
        m = n//2 # Comment:

        # Recurse on left & right half
        # b. What should go here?
        # c. What should go here?

        # Comment:
        return merge(sort_lt, sort_rt)
```

a. Add in-line comments to the above code, describing what the commands below them do.

b. For the comment above that says `# b.` write a line of code to mergesort the left half of `lst`.

c. For the comment above that says `# c.` write a line of code to mergesort the right half of `lst`.

Application Questions.

1. We can compare the run-times of these two algorithms, ***selection sort*** and ***mergesort,*** with some timing code and a lot of data:

timed_sorting.py

```
def timed_sorting(word_lst):
   start = time()
   sorted_word_lst = selection_sort(word_lst)
   end = time()
   print("Selection sort takes {} secs".format(end - start))
   start = time()
   sorted_word_lst = merge_sort(word_lst)
   end = time()
   print("Merge sort takes {} secs".format(end - start))
```

a. Fill in the "How many times faster" column in the table below with your *guess* about how many times faster the fastest sorting algorithm will be, as compared to the slower algorithm:

`len(word_lst)`	Algorithm	Run-time	How many times faster	Timed
500	`selection_sort`	$O(n^2)$		s
	`merge_sort`	O(n log n)		s
7,000	`selection_sort`	$O(n^2)$		s
	`merge_sort`	O(n log n)		s
122,089	`selection_sort`	$O(n^2)$		s
	`merge_sort`	O(n log n)		s

b. Run the code to compare the time required by our two sorting algorithms (or watch your instructor do so), and fill in the "Timed" column for the actual timed running of the algorithms.

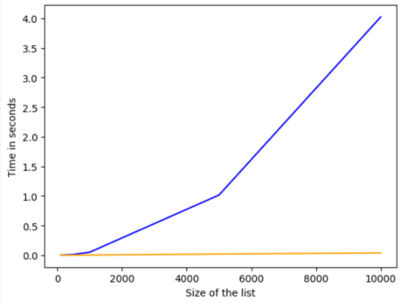
c. Circle the guesses you had that were correct (or close!). In the cases that you weren't, explain how you could have made a better prediction:

____

____

____

d. If we plot additional timed runnings of these two algorithms with `matplotlib`, we observe the following plot:



Label the lines on the plot with the algorithm they represent.

Why might the blue line have a sudden pivot near n=5000?

____

____

Why might the orange line look flat?

____

____