Name:________________________________ Partner: ______________________________

Python Activity 46: Searching

Understanding algorithmic efficiency is critical to computer science.

Learning Objectives
Students will be able to:
Content:

- Identify ***best case*** and ***worst case*** scenarios for searching algorithms
- Predict how changes in a ***searching*** algorithm impacts efficiency
- Define ***constant, linear, logarithmic,*** and ***quadratic*** run-times
- Explain how ***Big-O notation*** measures efficiency
- Describe the ***linear*** and ***binary searching*** algorithms for sorted vs. unsorted data

Process:

- Write code that implements ***binary search*** ***recursively***

Prior Knowledge

- Python concepts: computational thinking, recursion, lists, LinkedList

Concept Model:

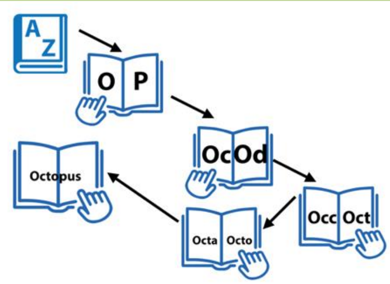
CM0. List examples of when you *search*: ____

____

What would happen if any of these search activities took twice as long as you expected?

____

CM1. The text and diagram below represent **two** approaches to finding the word "octopus" in a physical, paper dictionary (not a Python dictionary!).

Finding a Word in a Dictionary – Two Ways	
For each page in our dictionary book: Check to see if our word is on that page If it is, then we've found the word! If it isn't, then turn the page.	

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 find a word in a physical dictionary? ___________
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? ________________________

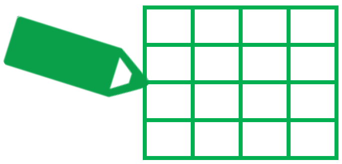
c. Which of these approaches would work better for finding a word in an *unsorted* order? Why?

____

FYI: A ***best case*** scenario is when the minimum number of operations is required (i.e., when an approach will take the fewest number of steps). A ***worst case*** scenario is when the maximum number of operations is required (i.e., most number of operations over all possible inputs). An ***average case*** scenario is when the average/typical number of operations is required.

CM2. We can explain the difference in efficiency in the above two algorithms with mathematical understanding. Let's consider another task: creating a grid on a piece of paper:

a. Complete the following table (don't overthink this one!):



Number of boxes we want on the paper	**Number of steps (number of boxes we draw)**
1	
2	
4	
8	
16	16

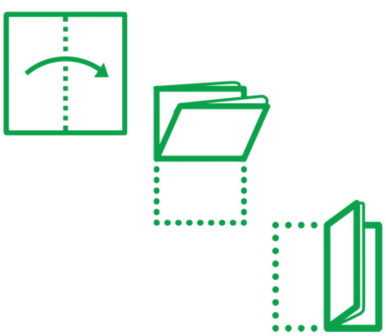
b. What is the mathematical relationship between the *number of boxes we want* and the *number of operations* if we draw each box's 4 sides (circle one):

constant linear logarithmic quadratic

FYI: ***Constant run-time*** occurs when an operation does not depend on the number of elements. ***Linear run-time*** is when an operation requires time proportional to the number of elements.

c. Is drawing each individual box (or each corner of each individual box) the only way to *create* a grid on a piece of paper? Is it the most operation-efficient way?

d. Complete the following table, where we do a different approach to creating a grid on a paper. **Highly* recommend actually folding a piece of paper!!!!*



Number of boxes we want on the paper	**Number of steps (number of times we fold the paper)**
1	
2	
4	
8	
16	

e. What is the mathematical relationship between the *number of boxes we want* and the *number of steps* if we fold the paper (circle one):

constant linear logarithmic quadratic

FYI: ***Logarithmic run-time*** occurs when an operation has a *inverse*-exponential relation to the number of elements. ***Quadratic run-time*** is when an operation requires time proportional to the number of elements *squared*.

f. What might be the run-time for the *first* dictionary searching algorithm in CM1?

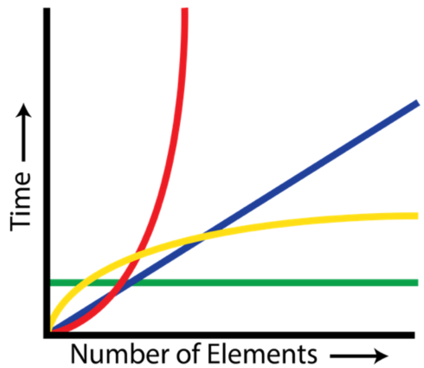
____

What might be the run-time for *your* dictionary searching algorithm in CM1?

____

CM3. 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:
Constant
Linear
Logarithmic
Quadratic

b. Provide an example algorithm or operation with the following run-times:

constant: ______________________________

linear: ______________________________

logarithmic: ______________________________

quadratic: ______________________________
(*what would result in a quadratic runtime?*)

> **FYI:** When computer scientists discuss run-times of algorithms or operations, we use something known as ***Big-O Notation*** which represents the run-time mathematical operations we've been discussing in this activity. Big-O notation drops less important details (such as the addition of constants), to focus mostly on the operation's *order of magnitude* based upon the number of elements (`n`).

c. Match the run-time on the left with its Big-O notation on the right:

Constant	$O(1)$
Linear	$O(n^2)$
Logarithmic	$O(2^n)$
Quadratic	$O(\log n)$
Exponential	$O(n!)$
Factorial	$O(n)$

d. Why might computer scientists use **Big-O notation** to describe the run-time efficiency of algorithms/operations, rather than simply timing the operation on a computer?

____

____

____

Critical Thinking Questions:

1. Examine the following partially complete code for *searching* for an item in a list:

linear.py

```
def linear_search(my_lst, item):
    # (i) for each item in our list


    # (ii) check to see if it's our item and…?


    # (iii) otherwise…
```

   a. Complete the code above where the comments scaffold a linear search of a list.

   b. Which searching 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 partially complete code for *searching* for an item in a *sorted* list:

binary.py

```
def binary_search(a_lst, item):
    """ Assume a_lst is sorted. If item is in a_lst, return True;
    else return False. """
    n = len(a_lst)
    mid = n // 2
    # Comment:
    if n == 0:
        return False
    # Comment:
    elif item == a_lst[mid]:
        return True
    # Comment:
    elif item < a_lst[mid]:
        return binary_search(a_lst[:mid], item)
    # Comment:
    else:
        # (iv). What should be done here?
```

FYI: *Sequence splicing* is an O(n) operation, which means this implementation of Binary Search is not as efficient as it could be! A faster Binary Search takes the `index_start` and `index_end` as arguments so that splicing isn't needed.

```
def binary_search_better(a_lst, item, index_start, index_end):
    n = index_end - index_start
    mid = (n // 2) + index_start
    if  n <= 0: return False
    elif item == a_lst[mid]: return True
    elif item < a_lst[mid]: return binary_search_better(a_lst, item, 0, mid)
    else: # (iv). What should be done here?
```

a. Step through the code, and explain what the following sections do:

`def binary_search(a_lst, item):`	
`n = len(a_lst)`	
`mid = n // 2`	
`if n == 0:` `return False`	
`elif item == a_lst[mid]:` `return True`	
`elif item < a_lst[mid]:` `return binary_search(a_lst[:mid], item)`	
`else:` `# (iv). What should be done here?`	

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

c. Write one line of code to complete the `(iv)` comment section:

____

____

d. 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(________)

e. What is the *worst* case scenario for this algorithm? ____________________________________

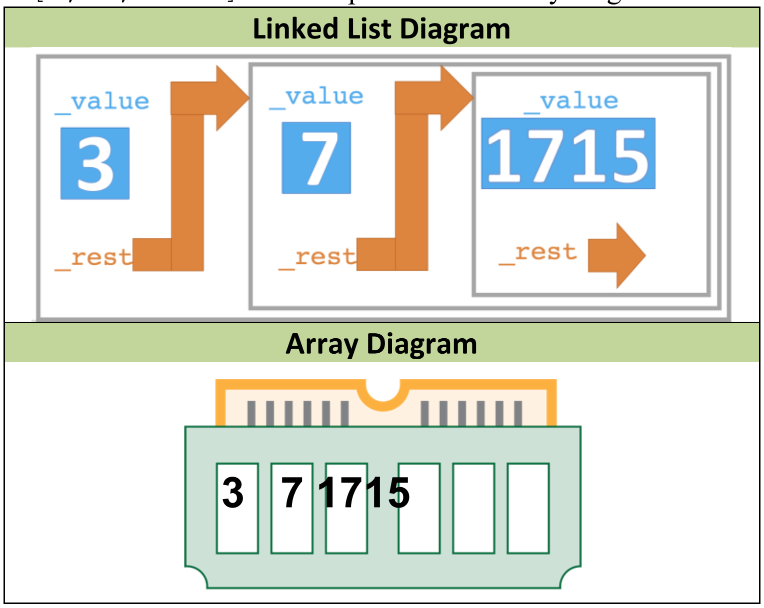
What is the ***Big-O notation*** run-time of this algorithm in the *worst case*? O(________)

f. Will this code work on an *unsorted* list? Why or why not?

____

Application Questions.

1. Recall our LinkedList class and its underlying structure, as in the diagram below for the list, `linklst = [3, 7, 1715]` and compare it to the array diagram below:



FYI: ***LinkedLists*** are a "pointer-based" data structure, and can grow & shrink on the fly, meaning we don't need to know how big they'll be when we create them. ***Arrays*** are a contiguous memory data structure, where all their data is stored contiguously in memory, so we need to know how much space to allocate for the array when we create it. These two data structures illustrate the ***time-space trade off***.

a. For the following operations, which data structure is better? Consider the ***run-time & space trade off***, and circle one!

When you know how big the list will be	LinkedList or Array
When you don't know how big the list will be	LinkedList or Array
Inserts at the beginning of the list	LinkedList or Array
Inserts at the end of the list	LinkedList or Array
Accessing an item by index	LinkedList or Array

b. Which data structure is *better*? Why?

____