CS110 assignment 1

Instead of a mini-project, which you will encounter in many of the upcoming CS110 assignments, in this first assignment you will be solving three independent problems. Use this opportunity to start your work early, finishing a question every week or so. Ideally, in the last week before the deadline, you will only have Q3 left to complete.

Feel free to add more cells to the ones always provided in each question to expand your answers, as needed. Make sure to refer to the CS110 course guide:

- on the grading guidelines, namely how many HC identifications and applications you are expected to include in each assignment.
- on the resources you are expected to submit with each assignment submission.

If you have any questions, do not hesitate to reach out to the TAs in the Slack channel "#cs110-algo", or come to one of your instructors' OHs.

Setting up notes:

1. Before you turn this problem in, make sure everything runs as expected. Click on `Kernel` $\rightarrow$ `Restart and Run All`.
2. Do not change the names of the functions provided in the skeleton codes.
3. Make sure you fill in any place that says `###YOUR CODE HERE` or "YOUR ANSWER HERE". Below, you will identify your name and collaborators.

NAME = "Steven H. Yang"

COLLABORATORS = "Yuehan Zeng"

Iteration vs. recursion

A Fibonacci Sequence is a list of numbers, in which each number is the sum of the two previous ones (starting with 0 and 1). Mathematically, we can write it as:

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2) for n larger than 1.

Your task is to compute the **n**th number in the sequence, where **n** is an input to the function.

(a) Use the code skeletons below to **provide two solutions** to the problem: an **iterative**, and a **recursive** one. For the recursive solution, feel free to use a helper function or add keyword arguments to the given function.

(b) In roughly 150 words:

- **explain** how your solutions follow the iterative and recursive paradigms, respectively.
- **discuss the pros and cons** of each approach as applied to this problem, and state which of your solutions you think is better, and why. Feel free to complement your answer with plots, if appropriate.
- provide a word count.

```
step_i = 0
step_r = 0
def fibonacci_iterative(n):
    ### YOUR CODE HERE
    """
    This function returns the fibonacci sequence's nth term.
    This function uses iterative method, which means the function does NOT recall i
    ----------
    n : int
    integer n
    Returns
    -------
    first_val: fibonacci sequence's nth term number.
    """
    global step_i
    first_val = 0 #since we know the first two values of the fibonacci sequence, we
                  #calculation benefits!
    second_val = 1
    if n == 0: #when n=0, there's no way that we can get first two terms of the fib
                  #Therefore, I put this if statement here to respond to the case whe
        step_i += 1
        return first_val
    else:
        step_i += 1
        for x in range(0,n): #since python starts counting from 0!
            step_i += 1
            temp_val = first_val #storing current first value as temp_val as it wil
            first_val = first_val + second_val #Fibonacci Calculation
            second_val = temp_val #in case the code have to move on to the one more
        return first_val


def fibonacci_recursive(n):
    ### YOUR CODE HERE
    """
    This function returns the fibonacci sequence's nth term.
    The function will countinuously recall itself until it gets the numerical value
    When we get the numerical values, function will create another numerical values
    ----------
    n : int
    integer n
    Returns
    -------
    fibonacci sequence's nth term number.
    """
    global step_r
    step_r += 1
    if n == 0:
        return 0
    elif n == 1 or n == 2: #since recursive function cannot return fibonacci_recurs
                        # I here define three basic returns to use them for recursi
        return 1
    else:
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2) #it executes the


#assert function belows checks whether the functions above work as expected.
#if functions work as expected, it won't show anything. If not, it will print error
#Commenting the test cases below as I confirmed they are working and not to affect
#assert fibonacci_iterative(0) == 0
#assert fibonacci_recursive(0) == 0
#assert fibonacci_iterative(4) == 3
#assert fibonacci_recursive(4) == 3


import matplotlib.pyplot as plt
#Open empty lists for storing the values
steps_i = []
steps_r = []
```

```
input_i = []
input_r = []
for x in range(0,11):
    fibonacci_iterative(x)
    input_i.append(x) #collect the input
    steps_i.append(step_i) #collect the number of steps
    step_i = 0 #reset the step to zero for the next iteration
    print(input_i,steps_i)
for x in range(0,11):
    fibonacci_recursive(x)
    input_r.append(x) #collect the input
    steps_r.append(step_r) #collect the number of steps
    step_r = 0 #reset the step to zero for the next iteration
    print(input_r,steps_r)

#plot the graph
plt.plot(input_i,steps_i,color='green',label='Iterative')
plt.plot(input_r,steps_r,color='red',label='Recursive')
plt.legend()
plt.xlabel('n')
plt.ylabel('Number of Steps')
txt="Fig 1. Change in the number of steps over the size of the input. \n This graph
plt.figtext(0.5, -0.2, txt, wrap=True, horizontalalignment='center', fontsize=12)
plt.show()
```

```
[0] [1]
[0, 1] [1, 2]
[0, 1, 2] [1, 2, 3]
[0, 1, 2, 3] [1, 2, 3, 4]
[0, 1, 2, 3, 4] [1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5] [1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6] [1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7] [1, 2, 3, 4, 5, 6, 7, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8] [1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
[0] [1]
[0, 1] [1, 1]
[0, 1, 2] [1, 1, 1]
[0, 1, 2, 3] [1, 1, 1, 3]
[0, 1, 2, 3, 4] [1, 1, 1, 3, 5]
[0, 1, 2, 3, 4, 5] [1, 1, 1, 3, 5, 9]
[0, 1, 2, 3, 4, 5, 6] [1, 1, 1, 3, 5, 9, 15]
[0, 1, 2, 3, 4, 5, 6, 7] [1, 1, 1, 3, 5, 9, 15, 25]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8] [1, 1, 1, 3, 5, 9, 15, 25, 41]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] [1, 1, 1, 3, 5, 9, 15, 25, 41, 67]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] [1, 1, 1, 3, 5, 9, 15, 25, 41, 67, 109]
```

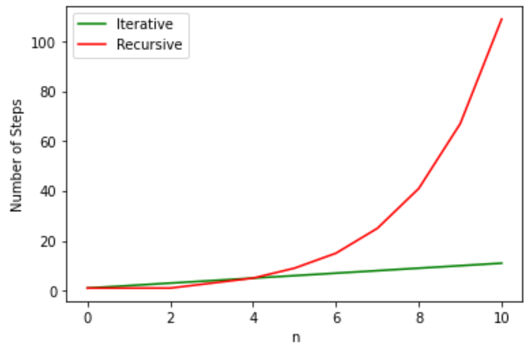

Fig 1. Change in the number of steps over the size of the input.
This graph plots both iterative and recursive method of calculating fibonacci sequence.
Until the input size is 4, both method perform similarly.
However, once the input size crosses 4, recursive function requires additional steps as it recalls it self a lot of times, which makes the algorithm more complicated.

The iterative code has for loop inside, and it computes the calculation until the for loop terminates while the recursive function recalls the function itself for further calculation if needed.

According to the graph, the iterative function increases its step much slower than how the recursive function grows its steps. This is because the recursive method requires recalling the function itself more and more as n grows. When we think about the algorithm tree, it keeps splits into another branch and branch. This is the critical reason why the recursive function grows its step much faster than how iterative functions grow its step.

This will result in a longer running time for recursive function, especially when n gets bigger and bigger.

Recursive function's code is much simpler and easy to write; however, this is not the best algorithm for the computer and uses as both want a faster computation. (149 words)

Understanding and documenting code

Imagine that you land your dream software engineering job, and among the first things you encounter is a previously written, poorly documented, and commented code.

Asking others how it works proved fruitless, as the original developer left. You are left with no choice but to understand the code's inner mechanisms, and document it properly for both yourself and others. The previous developer also left behind several tests that show the code working correctly, but you have a hunch that there might be some problems there, too.

Your tasks are listed below. Here is the code (please do not edit the following cell!):

```python
def my_sort(array):
    """
    YOUR DOCSTRING HERE
    """

    # ...
    for i in range(len(array)):

        # ...
        item = array[i]

        # ...
        intended_position = 0
        for num in array:
            if num < item:
                intended_position += 1

        # ...
        if i == intended_position:
            continue

        # ...
        array[intended_position], item = item, array[intended_position]

        # ...
        while i != intended_position:

            # ...
            intended_position = 0
            for num in array:
                if num < item:
                    intended_position += 1

            # ...
            array[intended_position], item = item, array[intended_position]

    return array
```

(a) Explain, in your own words, what the code is doing (it's sorting an array, yes, but **how**?). Feel free to use diagrams, play around with the code in other cells, print test cases or partially sorted arrays, draw step by step images. In the end, you should produce an approximately 150-word write-up of how the code is processing the input array.

This code will explore every item inside of the array. Every iteration, it will find its intended position(IP). It continuously moves the key by comparing when the elements on the left are smaller than the item that it tries to find its IP. Once it's on the IP, the code puts the item into the array on the IP. However, since it has to find the location of an item in the IP already, it takes the item that was in the array already to the item it tries to find its new IP. This is where the while loop helps. Inside the while loop, it repeats what it did here to find where the new item is supposed to go. Bug: after the while loop, it has a new item. It doesn't find IP for the new item. It goes back to the first for loop and gets the new-new item. (152words)

(b) Explain the difference between docstrings and comments. Add **both** a proper docstring and in-line comments to the code. You can follow the empty comments to guide you, but you can deviate, within reason.

Please keep in mind:

- Anyone from your section should be able to understand the code from your documentation. - Remember, however, that brevity is also a desirable feature.
- Please do not modify the code above. Include your docstrings and in-line comments in the cell below instead. This is important—you want to keep track of your changes!

```
def my_sort(array):
    """
    This code finds the sorted array of the input array.
    This code will continuously switch the item that we look for and the item that
    WARNING: This code is unfinished as it has a bug after while loop: new item has
    ----------
    array : list of elements
    array
    Returns
    -------
    sorted array
    """
    #The line below assigns each round repeatedly.
    for i in range(len(array)):

        #The line below chooses which specific item will be analyzed in a single rc
        item = array[i]

        #The lines below put the item for the very beginning of the array as defaul
        intended_position = 0
        for num in array:
            if num < item:
                intended_position += 1

        #This line will let loop to skip the iteration when current item is already
        if i == intended_position:
            continue

        #This line will be executed when loop is NOT skipped. It swaps the item and
        array[intended_position], item = item, array[intended_position]

        #Now we are repeating the process we did for the original item that we did
        while i != intended_position:

            #Resets the intended position. And repeat the things above.
            intended_position = 0
            for num in array:
                if num < item:
                    intended_position += 1

            #Once we find the intended position, we put the element we took out alr
            array[intended_position], item = item, array[intended_position]

    return array
```

© The previous developer included the following test cases with their code:

```
(1) assert my_sort([8, 5, 7]) == [5, 7, 8]

(2) assert my_sort([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]) == [1, 2, 3, 4, 5, 6, 7, 8, 9,

(3) input_array = [43, 99, 85, 45, 21, 58, 24, 12, 14, 64,
19, 94, 56, 13, 51, 2, 37, 11, 8, 66, 3, 95, 93, 53, 35,
81, 97, 9, 47, 78, 27, 50, 82, 71, 62, 59, 57, 42, 69, 72,
30, 63, 18, 31, 32, 88, 92, 73, 10, 74, 41, 22, 1, 80, 5,
60, 76, 52, 49, 77, 54, 44, 15, 7, 28, 84, 33, 83, 16, 91,
67, 23, 87, 25, 79, 89, 34, 4, 38, 48, 6, 96, 39, 40, 68,
55, 20, 36, 29, 65, 86, 70, 26, 98, 46, 90, 17, 0, 61, 75]
assert my_sort(input_array) == sorted(input_array)
```

They are *not* sufficient though. Explain why not and fix the code in the cell below.

Explanation:

The test inputs above will pass. However, the tests cases don't include and duplicated numbers. Based on the code above, it will be landed on the infinite loop because it will not change the index and looping inside the while loop. One example that can break this code is: [7,-7,7,-7].

After the first swap happens, the variable item will be replacing what's already in the list. However, if that was a duplicated value, after swapping, the new item variable has nowhere to go – returns an error at the end.

When the duplicates are inside of the input, the code above will keep counting by swapping because the termination conditions don't meet. However this kep counts up which makes the program goes out of the index.

Another example that can break this code is: [4,7,1].

```
def my_sort(array):
    """
    This code finds the sorted array of the input array.
    This code will continuously switch the item that we look for and the item that
    It takes the item after putting it into the array and find a new place for take
    ----------
    array : list of elements
    array
    Returns
    -------
    sorted array
    """
    #The line below assigns each round repeatedly.
    #len(array)-1 because we don't really have to repeat the last round since it sh
    for i in range(0, len(array)-1):

        #The line below chooses which specific item will be analyzed in a single ro
        item = array[i]

        #These lines find where to put the item that we currently have.
        intended_position = i
        #The for loops starts finding new intended position
        for num in range(intended_position +1, len(array)):
            if array[num] < item:
                intended_position += 1

        #This line will let loop to skip the iteration when current item is already
        if i == intended_position:
            continue

        #If we face duplicate, put the item right to the duplicate.
        #There might be multiple duplicates, we use while loop here.
        while item == array[intended_position]:
            intended_position += 1
        array[intended_position], item = item, array[intended_position]

        #Now we are repeating the process we did for the original item that we did
        #Since the new item is the element that was in array, we have to find where
        while i != intended_position:

            #Find where to put from unsorted part.
            intended_position = i
            for num in range(intended_position + 1, len(array)):
                if array[num] < item:
                    intended_position += 1
            #If we face duplicate, put the item right to the duplicate.
            #There mighe be multiple duplicates, we use while loop here.
            while item == array[intended_position]:
                intended_position += 1
            #Once we find the intended position, we put the element we took out alr
            array[intended_position], item = item, array[intended_position]

    return array


assert my_sort([7,-7,7,-7]) == sorted([7,-7,7,-7])
assert my_sort([4,-5,4,-5]) == sorted([4,-5,4,-5])
assert my_sort([5,9,1]) == sorted([5,9,1])

#Silen shared this input in the CS110 group chat for everyone wihtout a solution tc
#I modified the test input a little bit.
#It has decimals, negatives, zeros, multiiple identical numbers.
assert my_sort([1,3,1,324,2,34,23,423,-4,-324,-3.4,-324,2.34,32.4,-231,1,1,1,1,1,-3
```

New and mixed sorting approaches

In this question, you will implement and critique a previously unseen sorting algorithm. You will then combine it with another, known sorting algorithm, to see whether you can reach better behavior.

(a) Use the following pseudocode to implement `merge_3_sort()` . It is similar to merge sort—only that instead of splitting the initial array into halves, you will now split it into thirds, call the function recursively on each sublist, and then merge the triplets together. You might want to refer to this beautiful resource written by Prof. Drummond for details about the regular merge sort algorithm.

```python
def merge_3_sort(array, p, q):
    """
    Sorts array[p] to array[q] in place.
    E.g., to sort an array A, we will run
    `merge_3_sort(A, 0, len(A)-1)`.


    Parameters
    ----------
    array : Python list or numpy array
    p : int
        index of array element to start sorting from
    q : int
        index of last array element to be sorted

    Returns
    -------
    array: a sorted Python list

    """
    ### YOUR CODE HERE
    #The line below checks whether all items in array is ready to be sorted.
    input_validation = [False for i in array if (type(i)!=int and type(i)!=float)]
    if len(input_validation) > 0:
        return 'Your array includes non-numerical value. Please check your array'
    #If the array has one item, there's no need to sort.
    if len(array) == 0:
        return 'Your list is empty. Please fill the list'
    elif len(array) < 2 and len(array) > 0:
        return array
    #If the array has two elements, we can sort in this way by comparing two number
    elif len(array) == 2:
        if array[0] <= array[1]:
            return array
        else:
            array[0], array[1] = array[1], array[0]
            return array
    #If none of the conditions above don't meet, we can run merge_3_sort.
    else:
        #We split the array into three.
        if p < q:
            one_third = p + int((q-p)/3)
            two_third = p + int((q-p)/3)*2
            merge_3_sort(array,p,one_third) #Sort inside of the sub arraies
            merge_3_sort(array,one_third+1,two_third+1)
            merge_3_sort(array,two_third+2,q)
            merge_3(array,p,one_third,two_third,q) #Put them all together

    return array


def merge_3(array, p, q, r, s):
    """
    Merges 3 sorted sublists
    (array[p] to array[q], array[q+1] to array[r] and array[r+1] to array[s])
    in place.

    Parameters
    ----------
    array : Python list or numpy array
    p : int
        index of first element of first sublist
    q : int
        index of last element of first sublist
    r : int
        index of last element of second sublist
    s : int
        index of last element of third sublist

    """
    ### YOUR CODE HERE
    #Create three sub arraies
    L = array[p:q+1]
    MID = array[q+1:r+2]
    R = array[r+2:s+1]
    sentinal = float('inf')
    #Adding the sentinal(infinity) to the end of each array.
    L.append(sentinal)
    MID.append(sentinal)
    R.append(sentinal)

    #Set the indexes for the sub arraies
    indexL = 0
    indexMID = 0
    indexR = 0
    for i in range(p, s+1):
        #Take the minimum value of the index of each array that we look for. And ge
        minVal = min((L[indexL],MID[indexMID],R[indexR]))
        #if we find the minimum value from one certain sub array, we can increase t
        if minVal == L[indexL]:
            array[i] = L[indexL]
            indexL += 1
        elif minVal == MID[indexMID]:
            array[i] = MID[indexMID]
            indexMID += 1
        else:
            array[i] = R[indexR]
            indexR += 1
    return array
```

(b) Run at least 5 assert statements, which showcase that your code works as intended. In a few sentences, justify why your set of tests is appropriate and possibly sufficient.

```
array1 = [15,14,13,12,11,10,9,8,7,6,5,4,3,2,1] #Reversely Sorted
array2 = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] #Already Sorted
array3 = [11,12,13,14,15,1,2,3,4,5,6,7,8,9,10] #Sub-Arrays are sorted
array4 = [1,-1,1,-1,1,-1,1] #Osicilating Sequence
array5 = [1] #List with less than three items
array6 = [2,2,4,0,0,0,0,0,2.2,0.0,2,5,344444,2,-9,2,2,0,0,0] #Array with duplicated
array7 = [7,3] #List with less than three items
array8 = [] #List with nothing
array9 = [3,6,3.56,'Steven',3,49274,'Prof. Richard'] #List with non-numerical value
assert merge_3_sort(array1, 0, len(array1)-1) == sorted(array1)
assert merge_3_sort(array2, 0, len(array2)-1) == sorted(array2)
assert merge_3_sort(array3, 0, len(array3)-1) == sorted(array3)
assert merge_3_sort(array4, 0, len(array4)-1) == sorted(array4)
assert merge_3_sort(array5, 0, len(array5)-1) == sorted(array5)
assert merge_3_sort(array6, 0, len(array6)-1) == sorted(array6)
assert merge_3_sort(array7, 0, len(array7)-1) == sorted(array7)
assert merge_3_sort(array8, 0, len(array8)-1) == 'Your list is empty. Please fill t
assert merge_3_sort(array9, 0, len(array9)-1) == 'Your array includes non-numerical
```

I provided 9 test cases which inlcudes all different scenarios. By providing multiple scenarios, I can validate the input and find if my code is not working for certain types of inputs.

When I created those test cases, I kept thought about the conditions of algorithms and used pyhton visualizer to see how each test case breaks my code.

By having multiple test cases with different scenarios, I was able to find 4 mistakes in my code and all are resolved above.

© The algorithm becomes unnecessarily complicated when it tries to sort a really short piece of the original array, continuing the splits into single-element arrays. To work around this:

1. add a condition so that the algorithm uses selection sort (instead of continuing to recurse) if the input sublist length is below a certain threshold (which you should identify).
2. justify on the basis of analytical **and** experimental arguments what might be the optimal threshold for switching to selection sort.
3. include at least 5 assert statements that offer evidence that your code is correctly implemented.

To ensure you won't break your old code, first copy it to the cell below, and then create the new version in the code cell provided below.

```
def merge_3_sort_threshold(array, p, q, threshold):
    ### YOUR CODE HERE
    """
    Sorts array[p] to array[q] in place.


    Parameters
    ----------
    array : Python list or numpy array
    p : int
        index of array element to start sorting from
    q : int
```

```
            index of last array element to be sorted
        threshold : int
            threshold value of input array as one wishes

        Returns
        -------
        array: a sorted Python list

        """
        #Setting checkpoints for one third, two third of the input array.
        one_third = p + int((q-p)/3)
        two_third = p + int((q-p)/3)*2
        #Validate whether all inputs are able to be sorted
        input_validation = [False for i in array if (type(i)!=int and type(i)!=float)]
        if len(input_validation) > 0:
            return 'Your array includes non-numerical value. Please check your array'
        #If the array has one item, there's no need to sort.
        if len(array) == 0:
            return 'Your list is empty. Please fill the list'
        elif len(array) < 2 and len(array) > 0:
            return array
        #If array is smaller than threshold size, we can compute selection_sort here.
        if len(array[p:q]) <= threshold:
            selection_sort(array, p, q)
            return array
        else: #If array is NOT smaller than threshold size, we recursively call this fu
            merge_3_sort_threshold(array,p,one_third, threshold)
            merge_3_sort_threshold(array,one_third+1,two_third, threshold)
            merge_3_sort_threshold(array,two_third+1,q, threshold)
            merge_3(array,p,one_third,two_third,q)
            return array
    def merge_3_sort(array, p, q):
        """
        Sorts array[p] to array[q] in place.
        E.g., to sort an array A, we will run
        `merge_3_sort(A, 0, len(A)-1)`.



        Parameters
        ----------
        array : Python list or numpy array
        p : int
            index of array element to start sorting from
        q : int
            index of last array element to be sorted

        Returns
        -------
        array: a sorted Python list

        """
        ### YOUR CODE HERE
        #The line below checks whether all items in array is ready to be sorted.
        #input_validation = [False for i in array if (type(i)!=int and type(i)!=float)]
        #if len(input_validation) > 0:
        #    return 'Your array includes non-numerical value. Please check your array'
        #If the array has one item, there's no need to sort.
        if len(array) == 0:
            return 'Your list is empty. Please fill the list'
        elif len(array) < 2 and len(array) > 0:
            return array
        #If the array has two elements, we can sort in this way by comparing two number
        elif len(array) == 2:
            if array[0] <= array[1]:
                return array
            else:
                array[0], array[1] = array[1], array[0]
                return array
        else:
            #We split the array into three.
            if p < q:
                one_third = p + int((q-p)/3)
                two_third = p + int((q-p)/3)*2
                merge_3_sort(array,p,one_third) #Sort inside of the sub arraies
                merge_3_sort(array,one_third+1,two_third+1)
                merge_3_sort(array,two_third+2,q)
                merge_3(array,p,one_third,two_third,q) #Put them all together
            return array

    def merge_3(array, p, q, r, s):
        """
        Merges 3 sorted sublists
        (array[p] to array[q], array[q+1] to array[r] and array[r+1] to array[s])
        in place.

        Parameters
        ----------
        array : Python list or numpy array
```

```
        p : int
            index of first element of first sublist
        q : int
            index of last element of first sublist
        r : int
            index of last element of second sublist
        s : int
            index of last element of third sublist

        """
        ### YOUR CODE HERE
        #Create three sub arraies
        L = array[p:q+1]
        MID = array[q+1:r+1]
        R = array[r+1:s+1]
        sentinal = float('inf')
        #Adding the sentinal to the end of each array
        L.append(sentinal)
        MID.append(sentinal)
        R.append(sentinal)

        #Set the indexes for the sub arraies
        indexL = 0
        indexMID = 0
        indexR = 0
        for i in range(p, s+1):
            #Take the minimum value of the index of each array that we look for. And ge
            minVal = min((L[indexL],MID[indexMID],R[indexR]))
            #if we find the minimum value from one certain sub array, we can increase t
            if minVal == L[indexL]:
                array[i] = L[indexL]
                indexL += 1
            elif minVal == MID[indexMID]:
                array[i] = MID[indexMID]
                indexMID += 1
            else:
                array[i] = R[indexR]
                indexR += 1
        return array

    def selection_sort(array, p, q):
        """
        sorts the input array with selection sort method
        Parameters
        ----------
        array : Python list or numpy array
        p : int
            index of first element of list
        q : int
            index of last element of list
        """
        for i in range(p, q+1): #note that the first argument of for loop is inclusive
            minidx = i #minidx means minimum index
            for j in range(i+1, q+1):
                if array[j] < array[minidx]:
                    minidx = j
            array[i], array[minidx] = array[minidx], array[i] #swap method was adapted:
        return array


    array0 = [1,3,1,324,2,34,23,423,-4,-324,-3.4,2.34,32.4,-231,1,1,1,1,1,1,1,1,0,-1,2,
    array1 = [15,14,13,12,11,10,9,8,7,6,5,4,3,2,1] #Reversely Sorted
    array2 = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] #Already Sorted
    array3 = [11,12,13,14,15,1,2,3,4,5,6,7,8,9,10] #Sub-Arrays are sorted
    array4 = [1,-1,1,-1,1,-1,1] #Osicilating Sequence
    array5 = [1] #List with less than three items
    array6 = [2,2,4,0,0,0,0,0,2.2,0.0,2,5,344444,2,-9,2,2,0,0,0] #Array with duplicated
    array7 = [7,3] #List with less than three items
    array8 = [] #List with nothing
    array9 = [3,6,3.56,'Steven',3,49274,'Prof. Richard'] #List with non-numerical value

    assert merge_3_sort_threshold(array0,0,len(array0)-1,16) == sorted(array0)
    assert merge_3_sort_threshold(array1,0,len(array1)-1,16) == sorted(array1)
    assert merge_3_sort_threshold(array2,0,len(array2)-1,16) == sorted(array2)
    assert merge_3_sort_threshold(array3,0,len(array3)-1,16) == sorted(array3)
    assert merge_3_sort_threshold(array4,0,len(array4)-1,16) == sorted(array4)
    assert merge_3_sort_threshold(array5,0,len(array5)-1,16) == sorted(array5)
    assert merge_3_sort_threshold(array6,0,len(array6)-1,16) == sorted(array6)
    assert merge_3_sort_threshold(array7,0,len(array7)-1,1) == sorted(array7)
    assert merge_3_sort_threshold(array8, 0, len(array8)-1,16) == 'Your list is empty.
    assert merge_3_sort_threshold(array9, 0, len(array9)-1,16) == 'Your array includes


    import random #import random to create random list
    import timeit #timeit is a libraray that Viktor Tsvil recommended me to use. He did
    #The official documentation: https://docs.python.org/3/library/timeit.html#timeit.T
    import numpy as np
    #create a random list
```

```
random_list = [random.randrange(1,400) for i in range(500)]

#open a list to store the average running time
avg_running_time = []
#open a list to store threshold value that we used for testing
threshold_val = []
for threshold in range(5,50):
    #Since we choose the value of threshold to test, we append this to the list
    threshold_val.append(threshold)
    #open a temp list here because we want to store each calculated running time ar
    temp = []
    for i in range(500):
        start1 = timeit.default_timer() #Start the timer
        merge_3_sort_threshold(random_list,0,len(random_list)-1,threshold)
        stop1 = timeit.default_timer() #End the timer
        time1 = stop1-start1 #Time spent

        temp.append(time1) #append it to temp

    avg_running_time.append(np.mean(temp)) #Take the mean of temp and attach to avg

print(avg_running_time)
```

```
[0.006247239897999961, 0.004005568415999967, 0.003957210462000162, 0.00391245445000
```

```
import matplotlib.pyplot as plt
plt.plot(threshold_val,avg_running_time,color='green')
plt.title('Running time of Hybrid Function by Threshold')
plt.xlabel('Size of Thresholds')
plt.ylabel('Running Time (s)')
txt="Fig 2. Running time changes over the size of threshold. \n Around 16, we see t
plt.figtext(0.5, -0.15, txt, wrap=True, horizontalalignment='center', fontsize=12)
plt.show()

#since from 16 ish, using hybrid makes much more sense
```

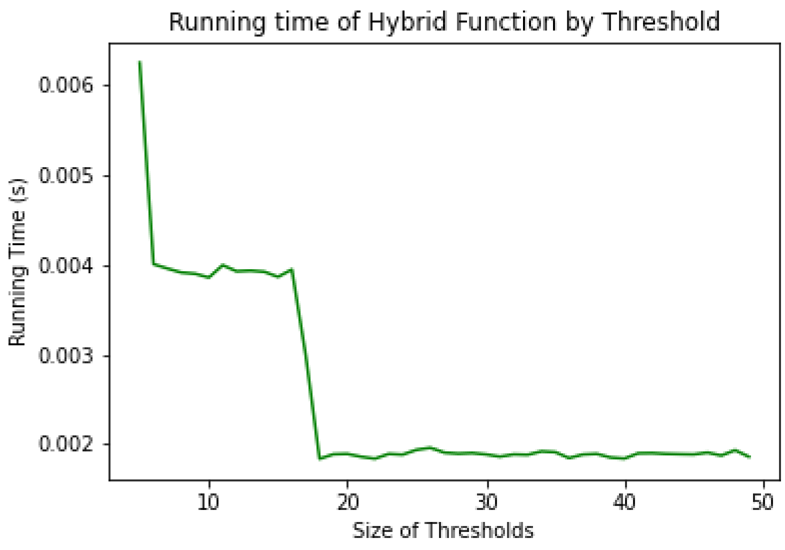

Fig 2. Running time changes over the size of threshold.
Around 16, we see that the running time drops down.
This graphically proves us that using hybrid function after threshold = 16 will benefit us in running time way.

Since I want to find the size of threshold that will benefit me in terms of running time, I want to find a point where selection sort and merge_3_sort share the equal running time. Once I find this, I can tell which until when it's better to use threshold. Selection sort's running time can be represented as (n^2+n)/2. Since I am taking a look at small sizes of input, that's why it's not n^2 here. For the same reason, merge_3_sort has (3n*(log n/log 3) *n +n) as its running time. When I let both equation to be equal and calculate the n value with the calculator, I get n = 16.215.

As we can see in Fig 2., We see that the running time drops down where the size of thresholds is around 16. This is an experimental approach that backs up the analytical reasoning above. This shows that the current code will be much more beneficial to recurse when the threshold is equal or greater than 16. However, it's still better to use only selection sort if the threshold is lower than 16.

(d) Finally, assess taking the hybrid approach (with the threshold) *versus* a strictly recursive algorithm (until the dividing step of the algorithm can no longer occur):

1. make this comparison as complete as possible (the analysis should include no less than 100 words, and at most 300 words).
2. your analysis must include both an analytical BigO complexity analysis, as well as graphical experimental evidence of your assertions.
3. provide a word count.

BigO of the pure merge sort is n*log n. However, BigO of the hybrid sort changes after the threshold. BigO of the hybrid sort is n^2 until the threshold, and it changes to n*log n. n^2 is more complex than n log n; however, it still works better for the hybrid algorithm because of the constant.

For the pure algorithm, let bigO is c1*nlog n and for the hybrid search, that's c2*n^2. In this case, n is such a small number that is equal to or lower than the threshold. Therefore, the existence of the constant does matter here. Since c2*n^2 < c1*nlog n satisfies, the hybrid search will perform better running time.

If I plot this with a graphical experiment, both functions logarithmically grow, but the pure one has a higher slope. This means that it is c1 > c2.

Here, the reasoning behind this is simple: in hybrid search, it uses less recursive structure than the normal merge sort. Since it keeps dividing the input array into its sub-arrays to use selection sort – avoid recalling the function itself as much as possible.

If one thinks this with the tree method, one will observe that the branch is spread in merge sort until we reach the base case – all sub-arrays are sorted already. The depth of the tree will be a huge number, in this case, depends on the input size. However, if selection sort comes when it's better to use than just merge sort, it will cut off expanding tree and make the tree itself shorter.

(260 words)

```python
def merge_3_sort_threshold(array, p, q, threshold):
    ### YOUR CODE HERE
    one_third = p + int((q-p)/3)
    two_third = p + int((q-p)/3)*2
# Commenting input_validation part as this is not necessary for analyzing BigO and
#     input_validation = [False for i in array if (type(i)!=int and type(i)!=float)
#     if len(input_validation) > 0:
#         return 'Your array includes non-numerical value. Please check your array'
# #     If the array has one item, there's no need to sort.
#     if len(array) == 0:
#         return 'Your list is empty. Please fill the list'
#     elif len(array) < 2 and len(array) > 0:
#         return array
    if len(array[p:q]) <= threshold:
        selection_sort(array, p, q)
        return array
    else:
        merge_3_sort_threshold(array,p,one_third, threshold)
        merge_3_sort_threshold(array,one_third+1,two_third, threshold)
        merge_3_sort_threshold(array,two_third+1,q, threshold)
        merge_3(array,p,one_third,two_third,q)
        return array
```

```
### YOUR CODE HERE
'''
ALERT: RUN THIS CELL FOR MULTIPLES TIMES TO SEE IF THE EXPERIMENTAL RESULTS ARE REF
- Steven Yang
'''


import random
import timeit
import numpy as np
#open lists to hold measured running time
runningtime_pure = []
runningtime_hybrid = []
#open a list to store input values
Input = []
#increasing the input size by 100 continuously to run this program quicker
for input_size in range(1, 15000, 100):
    #save the current input size
    Input.append(input_size)
    #create a random test list
    test_list = [random.randrange(1,500) for i in range(input_size)]
    for i in range(1):
        start1 = timeit.default_timer() #start timer
        merge_3_sort(test_list,0,len(test_list)-1)
        stop1 = timeit.default_timer() #end timer
        time1 = stop1-start1
        runningtime_pure.append(time1) #attach this to pure time
    for j in range(1):
        start2 = timeit.default_timer() #start timer
        merge_3_sort_threshold(test_list,0,len(test_list)-1,16)
        stop2 = timeit.default_timer() #end timer
        time2 = stop2-start2
        runningtime_hybrid.append(time2) #attach this to hybrid time

#plot the graph
import matplotlib.pyplot as plt
plt.plot(Input,runningtime_pure,color='green',label='Without Treshold')
plt.plot(Input,runningtime_hybrid,color='red',label='With Treshold')
plt.title('Compexity comparision with running time of merge and hybrid over input s
plt.legend()
plt.xlabel('Size of the inputs')
plt.ylabel('Running Time(s)')
txt="Fig 3. The graphs interpret the complexity of each function with runing time.
plt.figtext(0.5, -0.15, txt, wrap=True, horizontalalignment='center', fontsize=12)
plt.show()
```

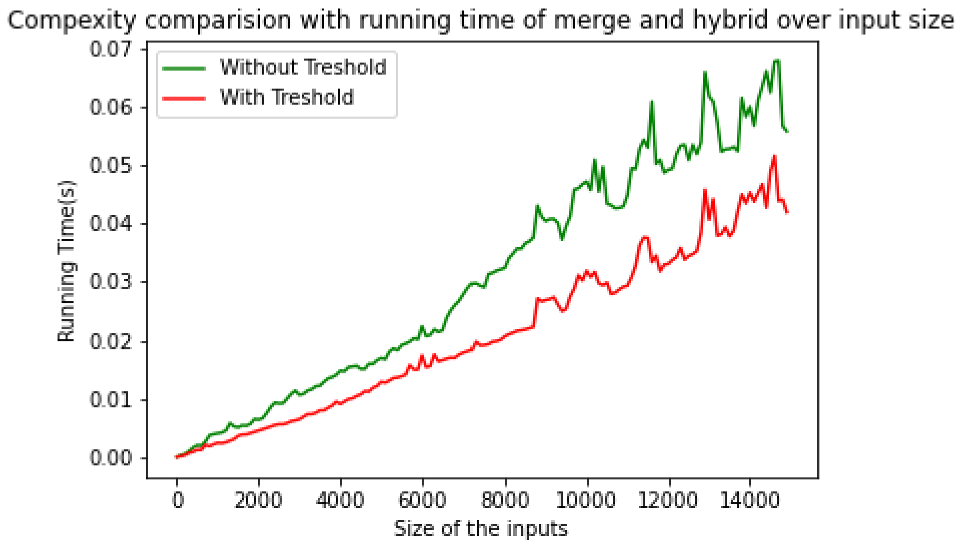

Fig 3. The graphs interpret the complexity of each function with runing time.
Both functions grow with the similar trend: n log n.
Since pure method has higher constant, it grows faster than the threshold method.

#HC APPENDIX

#DataViz: I created three graphs to back up my analytical analysis. Each figure is clearly labeled with x and y axes. The way to interpret the graph is also clearly noted as a caption. By using data visualization, I was able to justify my analysis.

#Optimization: I effectively applied this HC to address the threshold problem. Understanding that I want to minimize the algorithm's running time, I set this as my optimization goal. I found that the more recursive means, the more running time. By understanding the threshold meanings, I found that selection_sort can be added to the current algorithm to avoid numbers of recurses.

#Sampling: I generated multiple test cases and sample inputs. First, I generated multiple samples representing each different situation to see if the code was working in several ways. In the end, I also generated an array of the random list to mitigate potential biases: choosing numbers that I like or repetitions. By applying this HC effectively, I was able to find some errors in my code and was able to debug that successfully.

#Algorithms: I effectively used this HC to find algorithmic strategies and explained bigO and running time to explain why merge_3_sort_threshold could be a better option. I commented code effectively in case future CS110ers can read this code. I continuously drew pseudocode on my notebook and used pythontutor.com to understand what was happening in my code in every line.

#Professionalism: Even though I took longer than I expected, I tried my best to solve the problem and used official python documentation and some libraries. I drew pseudocode and used pythontutor.com to visualize how my python is being executed. I checked my grammar with Grammarly.