CS110_assignment1

September 25, 2021

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`→`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 = "Helen Prykhodko"

COLLABORATORS = "None"

1 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.

[1]:
```python
def fibonacci_iterative(n):
    """
    This function returns the nth number of Fibonacci sequence iteratively.

    Parameters
    ----------
    n : int
        The non-negative input to the function.

    Returns
    -------
    int
        The nth number in the sequence, if n is non-negative.

    """

    # Check whether n is a correct input.
    if n < 0:
        print("Please, enter a correct input of")

    # Initialize n1 and n2.
    n1 = 0
    n2 = 1

    # Looping to find the nth number in Fibonacci sequence.
    for i in range(0, n):
        n1, n2 = n2, n1 + n2
    return n1


def fibonacci_recursive(n):
    """
    This function returns the nth number of Fibonacci sequence recursively.

    Parameters
    ----------
    n : int
        The non-negative input to the function.

    Returns
    -------
    int
        The nth number in the sequence, if n is non-negative.2
    """

    # Check whether n is a correct input.
    if n < 0:
        print("Please, enter a correct input")

    # Base cases.
    elif n == 0:
        return 0
    elif n == 1 or n == 2:
        return 1

    # Enter the recursion.
    else:
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)

# Testcases.
assert fibonacci_iterative(0) == 0
assert fibonacci_recursive(0) == 0
assert fibonacci_iterative(4) == 3
assert fibonacci_recursive(4) == 3
```

Functions Analysis The fibonacci_iterative function (1) follows the iterative approach since it includes a for loop to find the nth number in the Fibonacci sequence. The recursive approach (2) breaks down the problem into smaller chunks, checks the base cases, and calls the function itself to solve the problem.

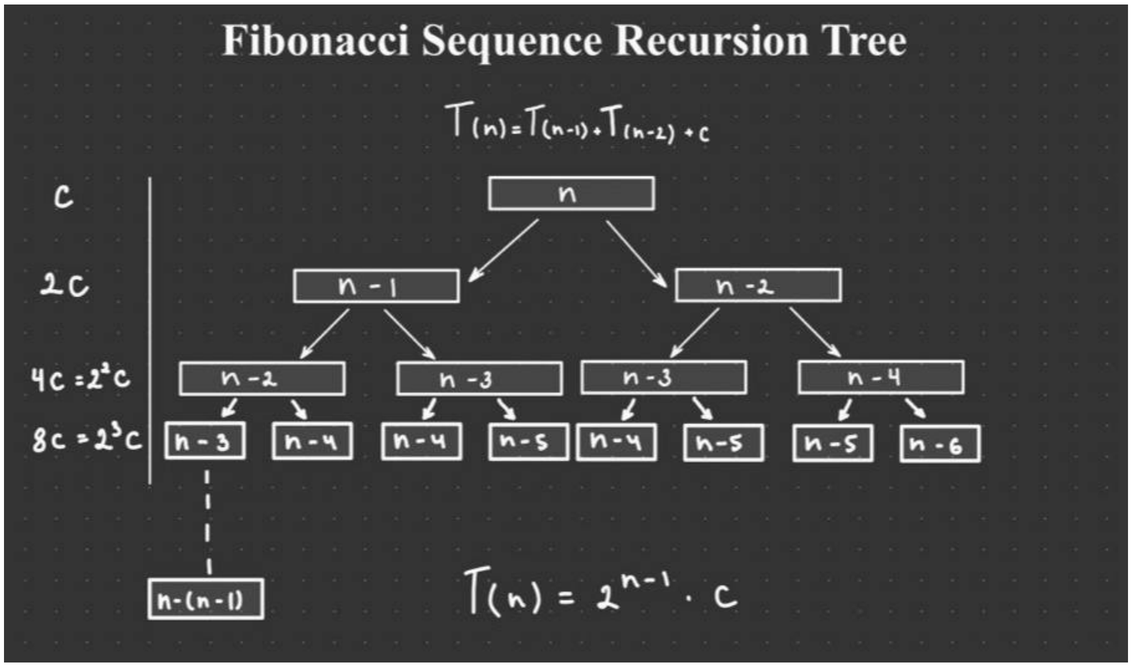
- Time Complexity: In (1), the loop runs from 1st number to (n+1)th which is a linear -> O(n). Plus, we make one comparison for n to be non-negative, this corresponds to a constant O(1). Therefore, T1(n) = O(n)+O(1). In (2), time complexity is less straightforward: T(n) = O(n-1)+O(n-2)+O(1), where O(1) is a constant c time it takes for comparisons and addition. As we see on recursion tree below, the time complexity of this algorithm is T(n) = 2^(n-1)*c ~ O(n) = 2^n.
- Space Complexity: In (1), the required space is the same for input of 100 and 1000 because n1 and n2 values are replaced through iteration requiring no additional storage. In (2), the space required is proportional to the depth of the recursion tree. We need much more storage to remember the solutions of the smaller chunks of the problem.

Therefore, algorithm (2) is less efficient than (1) based on running time and space complexity.

Word Count: 195 words.

[2]:
```
from IPython.display import Image
Image("Fibonacci Recursive.jpg")
```

[2]:



2 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!):

[3]:
```
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
```

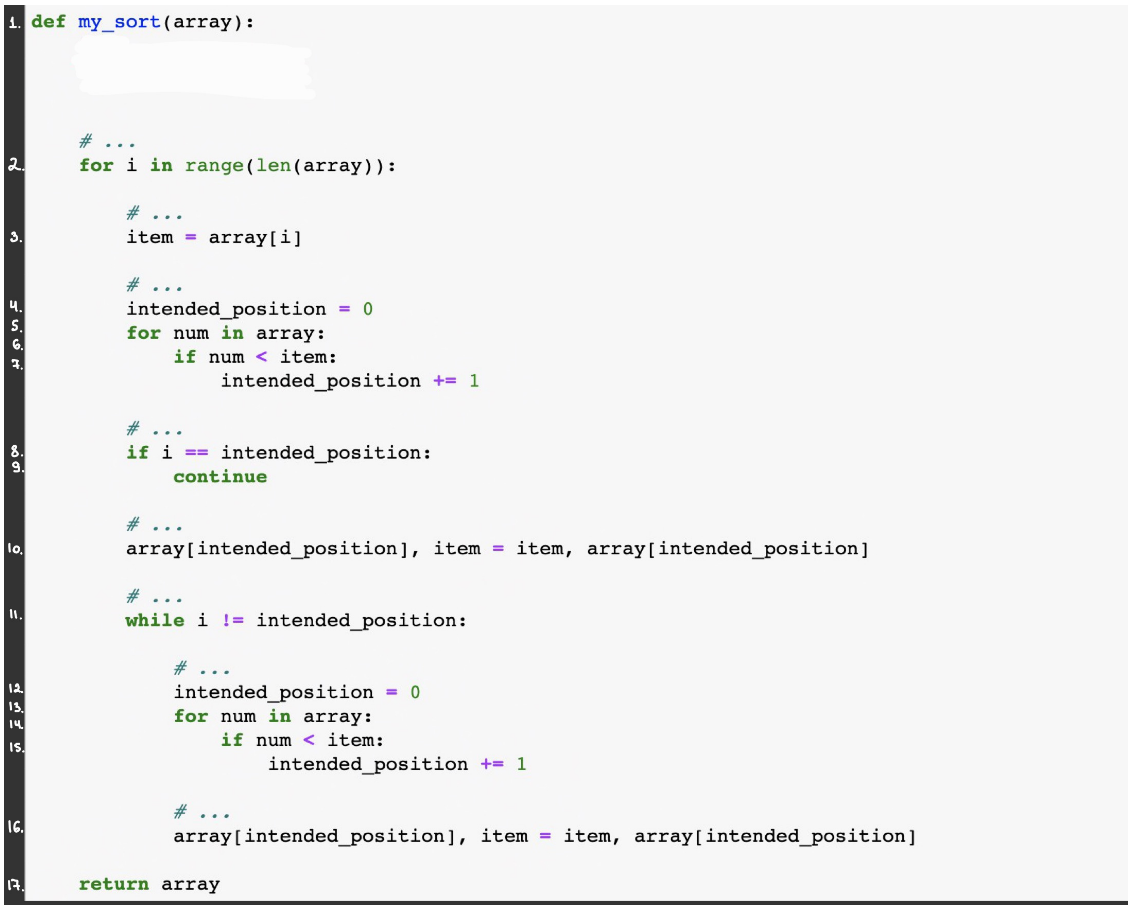
(a) Explain, in your own words, what the code is doing (it's sorting an array, yes, but how?). Feel free to use diagrams, play around with the code in other cells, print test cases or partially sorted arrays, draw step by step images. In the end, you should produce an approximately 150-word write-up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

The algorithm's steps are outlined in the figure below. The function takes input as an array to sort it in ascending order. **i** is the index of numbers in the array, **item** is a current number we are comparing to others, **intended position** is the place in the array we want the number to be at, and **num** is the number we are comparing the item to. The first loop compares the item with the number and increases the intended position if the item is greater. If after the loop i = intended position (the item is less than other numbers in array), we continue with i+=1. If not, we assign the value of a number in the intended position the item's value. We enter the while loop to change the value of the item and change the intended position for later comparisons. We then repeat the algorithm for the length of the array times.

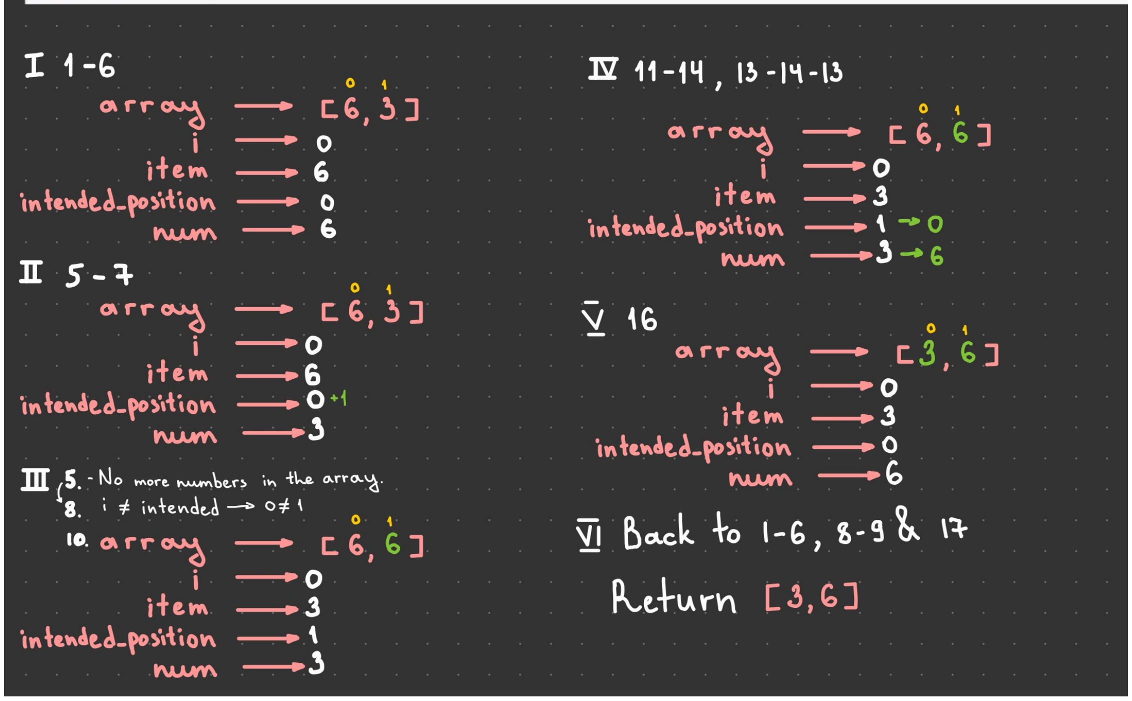
However, the algorithm has a bug. If there are two identical numbers in the array, the while loop will run forever because there is no base case to stop it. Ex.: for array [8,8,1], i is always not equal to the intended position by the end of for loop. The algorithm is unable to recognize the same numbers to switch places with the 1.

[4]:
```python
from IPython.display import Image
Image("My_Sort.jpg")
```

[4]:



```
1.  def my_sort(array):



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

            # ...
3.          item = array[i]

            # ...
4.          intended_position = 0
5.          for num in array:
6.              if num < item:
7.                  intended_position += 1

            # ...
8.          if i == intended_position:
9.              continue

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

            # ...
11.         while i != intended_position:

                # ...
12.             intended_position = 0
13.             for num in array:
14.                 if num < item:
15.                     intended_position += 1

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

17.     return array
```



I 1-6

```
                      0  1
          array  →  [6, 3]
              i  →  0
           item  →  6
intended_position →  0
            num  →  6
```

II 5-7

```
                      0  1
          array  →  [6, 3]
              i  →  0
           item  →  6
intended_position →  0 +1
            num  →  3
```

III 5. - No more numbers in the array.
8. i ≠ intended → 0≠1

```
                          0  1
          10. array  →  [6, 6]
                  i  →  0
               item  →  3
  intended_position  →  1
                num  →  3
```

IV 11-14, 13-14-13

```
                      0  1
          array  →  [6, 6]
              i  →  0
           item  →  3
intended_position →  1 → 0
            num  →  3 → 6
```

V 16

```
                      0  1
          array  →  [3, 6]
              i  →  0
           item  →  3
intended_position →  0
            num  →  6
```

VI Back to 1-6, 8-9 & 17

Return [3,6]

(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!

Docstrings VS Comments

Docstrings are string constants that explain the python object (e.g. class, funtion) or module. We write it as multiline comments with numerous strings. It starts and ends with triple single/double quotes. The first line should be a short description of what the object does. Then, we describe parameters and what the function returns.

Comments are a well-defined clear explanations of seprate code structures. We should add comments where English explanation would be useful to understand the code to either yourself or someone else who is going to read the code. The comments start with # sign.

[5]:
```python
def my_sort(array):
    """
    This function returns a sorted array in ascending order.

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

    Returns
    -------
    array: a sorted array in ascending order

    """

    # Looping through the array to find numbers to switch.
    for i in range(len(array)):

        # Assign item a value of number with index i.
        item = array[i]

        # Look where to put the item.
        intended_position = 0
        for num in array:
            if num < item:
                intended_position += 1
        # If the item is the smallest number
        # in array, continue the for loop.
            elif i == intended_position:
                continue

        # Else, put the item on number's place.
        array[intended_position], item = item, array[intended_position]

        # Loop through the rest of the cycle
        # to check through the rest of the numbers.
        while i != intended_position:

            # Assign intended position to 0.
            intended_position = 0
            for num in array:
                if num < item:
                    intended_position += 1

            # Change the number under intended_position on item.
            array[intended_position], item = item, array[intended_position]

    return array
```

(c) 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, 10]

```
(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.

The test cases do not include arrays with repetitive numbers with which the algorithm does not sort correctly. The while loop runs infintely when repetitve numbers in the array appear.

[6]:
```
def my_sort(array):
    """
    This function returns a sorted array in ascending order.
    Parameters
    ----------
    array : Python list or numpy array

    Returns
    -------
    array: a sorted array in ascending order

    """
    # Looping through the array to find numbers to switch.
    for i in range(len(array)-1):

        # Assign item a value of number with index i.
        item = array[i]

        # Look where to put the item.
        intended_position = i
        for num in range(i+1, len(array)):
            if array[num] < item:
                intended_position += 1

        # If the item is the smallest number
        # in array, continue the for loop.
        if i == intended_position:
            continue

        # Loop to change the intended position to avoid
        # looping forever lately if numbers are equal.
        while item == array[intended_position]:
            intended_position += 1

        # Put the item on number's place.
        array[intended_position], item = item, array[intended_position]

        # Looping to check for the rest of numbers to switch.
        while i != intended_position:

            # Find a position to place the item.
            intended_position = i
            for num in range(i+1, len(array)):
                if array[num] < item:
                    intended_position += 1

            # Loop to change the intended position to avoid
            # looping forever lately if numbers are equal.
            while item == array[intended_position]:
                intended_position += 1
            # Put the item on number's place.
            array[intended_position], item = item, array[intended_position]

    return array
```

[7]:
```
# Test cases.
assert(my_sort([]) == [])
assert(my_sort([5]) == [5])
assert(my_sort([5,1]) == [1,5])
assert(my_sort([10,9,8,7,6,5,4,3,2,1]) == [1,2,3,4,5,6,7,8,9,10])
assert(my_sort([5,2,7,-2,2,9,5]) == [-2,2,2,5,5,7,9])
```

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.

[8]:
```
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

    """

    if (q - p) < 2:
        return array
    else:
        r = p+((q-p)//3)
        s = p + 2 * ((q-p) // 3) + 1

        # Use recursion to sort subarrays
        # and merge an original array.
        merge_3_sort(array, p, r)
        merge_3_sort(array, r, s)
        merge_3_sort(array, s, q)
        merge_3(array, p, r, s, q)
    return array


def merge_3(array, p, r, s, q):
    """
    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
    r : int
        index of last element of first sublist
    s : int
        index of last element of second sublist
    q : int
        index of last element of third sublist

    """


    # Create subarrays to
    # divide the array.
    left = array[p : r]
    mid = array[r : s]
    right = array[s : q]

    # Set initial indeces
    # for each subarray.
    ind_left = 0
    ind_mid = 0
    ind_right = 0
    array_idx = p

    # Looping to check all possible conditions for sorting the arrays.

    while ind_left < len(left) and ind_mid < len(mid) and ind_right <␣
↪len(right):
        if left[ind_left] < mid[ind_mid]:
            if left[ind_left] < right[ind_right]:
                array[array_idx] = left[ind_left]
                ind_left += 1
            else:
                array[array_idx] = right[ind_right]
                ind_right += 1
            array_idx += 1
        else:
            if mid[ind_mid] < right[ind_right]:
                array[array_idx] = mid[ind_mid]
                ind_mid += 1
            else:
                array[array_idx] = right[ind_right]
                ind_right += 1
            array_idx += 1

    while ind_left < len(left) and ind_mid < len(mid):
        if left[ind_left] < mid[ind_mid]:
            array[array_idx] = left[ind_left]
            ind_left += 1
        else:
            array[array_idx] = mid[ind_mid]
            ind_mid += 1
        array_idx += 1

    while ind_left < len(left) and ind_right < len(right):
        if left[ind_left] < right[ind_right]:
            array[array_idx] = left[ind_left]
            ind_left += 1
        else:
            array[array_idx] = mid[ind_right]
            ind_right += 1
        array_idx += 1

    while ind_mid < len(mid) and ind_right < len(right):
        if mid[ind_mid] < right[ind_right]:
            array[array_idx] = mid[ind_mid]
            ind_mid += 1
        else:
            array[array_idx] = right[ind_right]
            ind_right += 1
        array_idx += 1

    while ind_left < len(left):
        array[array_idx] = left[ind_left]
        ind_left += 1
        array_idx += 1

    while ind_mid < len(mid):
        array[array_idx] = mid[ind_mid]
        ind_mid += 1
        array_idx += 1

    while ind_right < len(right):
        array[array_idx] = right[ind_right]
        ind_right += 1
        array_idx += 1
```

(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.

I try 5 different test cases to check whether the algorithm works. I look whether base cases work for arrays with length 0, 1, and 2. Then I look at the randomized array with negative, positive and repetitive numbers, and a worst-case scenario with an array in descending order. The algorithm works for all test cases. This is sufficient because I check every possible input that can happen.

[9]:
```
# Test cases.
array1 = []
array2 = [9]
array3 = [5,1]
array4 = [10,9,8,7,6,5,4,3,2,1]
array5 = [100,-20,5,2,85,43,2,23,67]

assert(merge_3_sort(array1,0,len(array1)) == [])
assert(merge_3_sort(array2,0,len(array2)) == [9])
assert(merge_3_sort(array3,0,len(array3)) == [1,5])
assert(merge_3_sort(array4,0,len(array4)) == [1,2,3,4,5,6,7,8,9,10])
assert(merge_3_sort(array5,0,len(array5)) == [-20,2,2,5,23,43,67,85,100])
```

(c) 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:

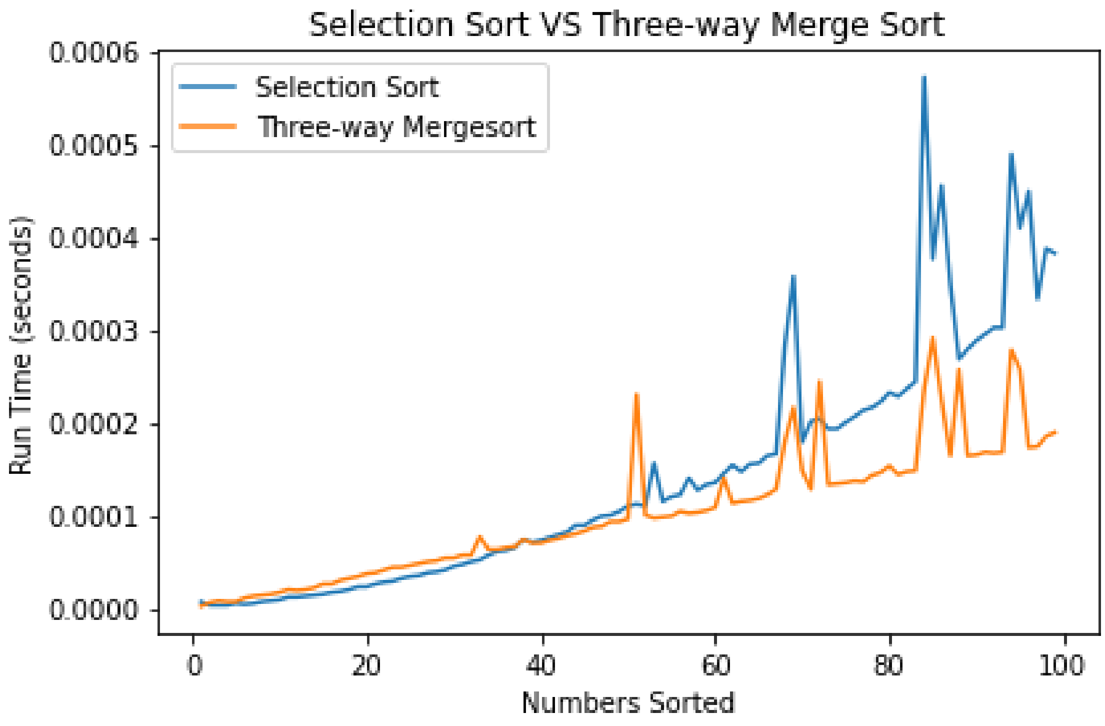
1. add a condition so that the algorithm uses selection sort (instead of continuing to recurse) if the input sublist length is below a certain threshold (which you should identify).

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.

[28]:
```
def selection_sort(array, p, q):
    for i in range (p, q+1):
        min_idx = i
        for j in range(i+1, q+1):
            if array[j]<array[min_idx]:
                min_idx = j
        array[i], array[min_idx] = array[min_idx],array[i]
    return array
```

[29]:
```
# Import necessary libraries.
import matplotlib.pyplot as plt
import time
import random

def graph_merge3sort_and_selectionsort():
    """
    This function produces a graph plotting running
    time of selection sort and three-way merge sort.

    """

    # Create lists to add time values.
    selection_sort_time = []
    merge_3_sort_time = []

    # Looping through random test cases running times.
    for i in range(1,100):
        testing = random.sample(range(-100,100),i)
        t1 = time.process_time()
        selection_sort(testing, 0, len(testing) - 1)
        t2 = time.process_time()
        merge_3_sort(testing, 0, len(testing))
        t3 = time.process_time()

        # Add running times to the lists.
        selection_sort_time.append((t2-t1))
        merge_3_sort_time.append((t3-t2))

    # Define x range for sorted numbers.
    x = range(1,100)
    # Add graph attributes.
    plt.plot(x, selection_sort_time)
    plt.plot(x, merge_3_sort_time)
    plt.title('Selection Sort VS Three-way Merge Sort')
    plt.xlabel('Numbers Sorted')
    plt.ylabel('Run Time (seconds)')
    plt.legend(['Selection Sort','Three-way Mergesort'], loc = 'upper left')
    plt.tight_layout()
    plt.show()

graph_merge3sort_and_selectionsort()
```



[34]:
```
def merge_3_sort_threshold(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

    """

    if len(array[p:q]) <= 40:
        selection_sort(array, p, q-1)

    else:
        r = p+((q-p)//3)
        s = p + 2 * ((q-p) // 3) + 1

        # Use recursion to sort subarrays
        # and merge an original array.
        merge_3_sort_threshold(array, p, r)
        merge_3_sort_threshold(array, r, s)
        merge_3_sort_threshold(array, s, q)
        merge_3(array, p, r, s, q)
    return array


def merge_3(array, p, r, s, q):
    """
    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
    r : int
        index of last element of first sublist
    s : int
        index of last element of second sublist
    q : int
        index of last element of third sublist

    """
    # Create subarrays to
    # divide the array.
    left = array[p : r]
    mid = array[r : s]
    right = array[s : q]

    # Set initial indeces
    # for each subarray.
    ind_left = 0
    ind_mid = 0
    ind_right = 0

    array_idx = p

    # Looping to check all possible conditions for sorting the arrays.

    while ind_left < len(left) and ind_mid < len(mid) and ind_right <␣
↪len(right):
        if left[ind_left] < mid[ind_mid]:
            if left[ind_left] < right[ind_right]:
                array[array_idx] = left[ind_left]
                ind_left += 1
            else:
                array[array_idx] = right[ind_right]
                ind_right += 1
            array_idx += 1
        else:
            if mid[ind_mid] < right[ind_right]:
                array[array_idx] = mid[ind_mid]
                ind_mid += 1
            else:
                array[array_idx] = right[ind_right]
                ind_right += 1
            array_idx += 1

    while ind_left < len(left) and ind_mid < len(mid):
        if left[ind_left] < mid[ind_mid]:
            array[array_idx] = left[ind_left]
            ind_left += 1
        else:
            array[array_idx] = mid[ind_mid]
            ind_mid += 1
        array_idx += 1

    while ind_left < len(left) and ind_right < len(right):
        if left[ind_left] < right[ind_right]:
            array[array_idx] = left[ind_left]
            ind_left += 1
        else:
            array[array_idx] = mid[ind_right]
            ind_right += 1
        array_idx += 1

    while ind_mid < len(mid) and ind_right < len(right):
        if mid[ind_mid] < right[ind_right]:
            array[array_idx] = mid[ind_mid]
            ind_mid += 1
        else:
            array[array_idx] = right[ind_right]
            ind_right += 1
        array_idx += 1

    while ind_left < len(left):
        array[array_idx] = left[ind_left]
        ind_left += 1
        array_idx += 1

    while ind_mid < len(mid):
        array[array_idx] = mid[ind_mid]
        ind_mid += 1
        array_idx += 1

    while ind_right < len(right):
        array[array_idx] = right[ind_right]
        ind_right += 1
        array_idx += 1
```

[36]:
```
# Test cases.
array1 = []
array2 = [9]
array3 = [5,1]
array4 = [10,9,8,7,6,5,4,3,2,1]
array5 = [100,-20,5,2,85,43,2,23,67]

assert(merge_3_sort_threshold(array1,0,len(array1)) == [])
assert(merge_3_sort_threshold(array2,0,len(array2)) == [9])
assert(merge_3_sort_threshold(array3,0,len(array3)) == [1,5])
assert(merge_3_sort_threshold(array4,0,len(array4)) == [1,2,3,4,5,6,7,8,9,10])
assert(merge_3_sort_threshold(array5,0,len(array5)) ==␣
 ↪[-20,2,2,5,23,43,67,85,100])
```

Theoretically, the threshold should be the number when selection sort performs faster for n items in the array than the three-way merge sort. Because time complexities are different, there is a point when selection sort outperforms merge sort. Though merge sort has a lower rate of growth - n×log3(n) - it is inefficient for arrays with small amount of items. For smaller inputs, higher rates are more efficient. So, if selection sort's order of growth is n^2, then there should be a value for input n when n^2<n×log3(n).

My reasoning behind the threshold of 40 is the graph result (``Selection Sort VS Three-way Merge Sort''). Running it multiple times to randomize the arrays, we observe that selection sort performs faster when the number of items in the array is around 40 or less.

(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.

Three-Way Merge Sort

This algorithm works as a regular merge sort, except it divides the array into three equal parts rather than two (see the recursion tree below). The recursion equation for this algorithm will be ( ) = 3 (/3) + Θ( ) + Θ(1). Solving the equation, we get the time complexity of O(n×log3(n)). The three-way merge sort requires more space complexity because we divide the subarray into three rather than two parts.

Three-Way Merge Sort with Threshold

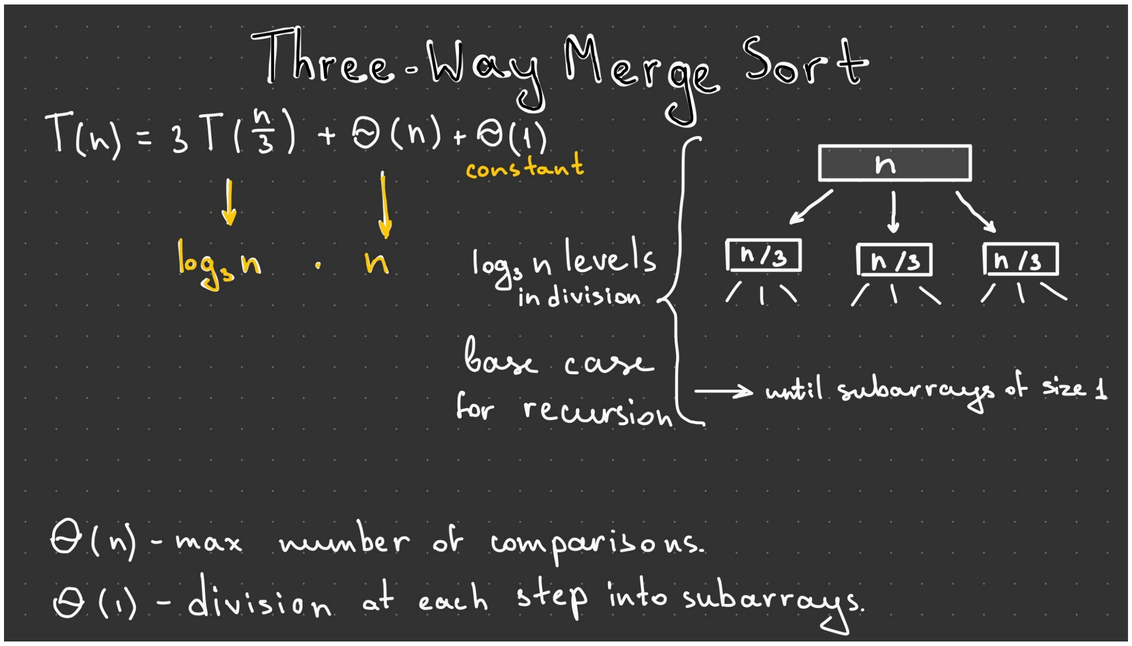
Because the merge sort with threshold uses selection sort for smaller values of n, the algorithm runs faster. When we divide the array into subarrays so that it reaches the threshold value of 40, we use selection sort, which is much faster for smaller inputs with time complexity: - for the best case: O(n^2). - for the worst case: O(n^2).

Here, selection sort time complexity is O(40^2) in the worst-case scenario. So, as n increases, the computational complexity stays constant. Instead of going through recursions with O(n×log3(n)), we are saving time with threshold value. Combining two algorithms, we get higher efficiency. This can also be seen on the graph below with the running time of both algorithms.

Word Count: 194 words.

```
[45]: from IPython.display import Image
      Image("Three-Way Merge Sort Complexity.jpg")
```

[45]:



Three-Way Merge Sort

$T(n) = 3T(\frac{n}{3}) + \Theta(n) + \underset{\text{constant}}{\Theta(1)}$

$\log_3 n \cdot n$

$\log_3 n$ levels in division

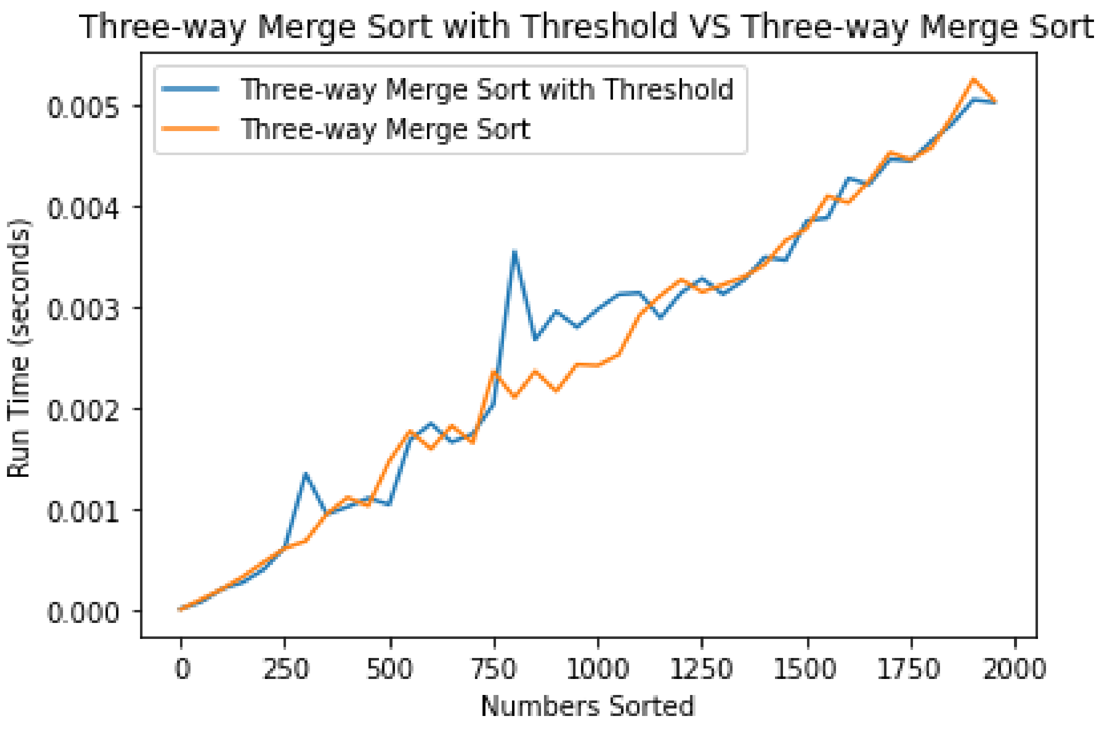
base case for recursion $\rightarrow$ until subarrays of size 1

$\Theta(n)$ - max number of comparisons.

$\Theta(1)$ - division at each step into subarrays.

```
[44]: # Import necessary libraries.
      import matplotlib.pyplot as plt
      import time
      import random

      def graph_merge3sort_and_threshold():
          """
          This function produces a graph plotting running time of three-way
          merge sort with threshold sort and three-way merge sort.

          """

          # Create lists to add time values.
          merge_3_sort_threshold_time = []
          merge_3_sort_time = []

          # Looping through random test cases running times.
          for i in range(1,2000,50):
              testing = random.sample(range(-2000,2000),i)
              t1 = time.process_time()
              merge_3_sort_threshold(testing, 0, len(testing))
              t2 = time.process_time()
              merge_3_sort(testing, 0, len(testing))
              t3 = time.process_time()

              # Add running times to the lists.
```

```
        merge_3_sort_threshold_time.append((t2-t1))
        merge_3_sort_time.append((t3-t2))

    # Define x range for sorted numbers.
    x = range(1,2000,50)

    # Add graph attributes.
    plt.plot(x, merge_3_sort_threshold_time)
    plt.plot(x, merge_3_sort_time)
    plt.title('Three-way Merge Sort with Threshold VS Three-way Merge Sort')
    plt.xlabel('Numbers Sorted')
    plt.ylabel('Run Time (seconds)')
    plt.legend(['Three-way Merge Sort with Threshold','Three-way Merge Sort'],␣
 ↪loc = 'upper left')
    plt.tight_layout()
    plt.show()

graph_merge3sort_and_threshold()
```



HCs Appendix #algorithms: I identified appropriate algorithmic strategies for every function. I implemented a well-defined, clear, and efficient code as well as explained my steps with clear docstrings and comments. Also, I was using appropriate variables so that others could understand my code easily. I explained the complexity analysis for running time and space needed when appropriate and justified my choice with graphs and figures.

#dataviz: I created detailed highly relevant data visualizations for the data and analyzed why it is appropriate and correct in the context. For instance, on the last visualization where I compare the three-way merge sort with threshold function with the three-way merge sort function, I explain why the running time of a threshold function is better than of the other one. I put two line graphs on the plot with different colors so that anyone could easily differentiate them to compare. Also, I provided relevant names for y-axis and x-axis.

#breakitdown: In merge sort, I break down the problem of sorting the algorithm: instead of sorting the whole array I break it into three subarrays so that the decomposition affects the algorithm's efficiency. I also explain how such decomposition changes the time complexity.I accurately combine component solutions into a full one which is clearly detailed and explained.I acknowledge that decomposition into three subarrays requires more storage than the regular merge sort.