GCIS-123
Class Activity & Problem Solving #03
Sort Time Complexity

Goals of the Assignment

For this assignment, you will be creating a new sort based on the previous sorts we have discussed. You will start by examining the runtime complexity for all the sorts using various data sets. Next, you must create a new sort using two of the existing sorts. The goal is to create a sort that runs better, on average, for all the data than any of the sorts we have created so far.

Activities (20% for each step)

1. Create a new Python module "`sort_times.py`".
    a. Write a function called `sort_function_timer` that declares parameters for a `sort_function` and `an_array`. When you call this function, you will pass one of your sort functions as an argument. Time how long it takes for the `sort_function` to sort `an_array`, then return that time.
    b. Create a global array called `SIZES` containing `200`, `500`, `800`, `1100`, `1400`, `1700`, and `2000` in order. The `SIZES` array will determine the size of arrays you will use to measure the time required to sort them.
    Hint: use the step value in the `range_array` function or similar
    c. Create a function called "`plot_sort_time_using_random_arrays`" that declares a parameter for a `sort_function`. Use the `sort_function_timer` function to time how long it takes for the `sort_function` to sort random arrays of various sizes in the `SIZES` array. This function should simply add data points to the plot. Make sure to call the `plotter.new_series` function before adding data points. Use the following as a code stub:

```
def plot_sort_time_using_random_arrays(sort_function):
    print("timing", sort_function.__name__)
    plotter.new_series()
```

2. Add a `main` function to your `sort_times` module so that it displays a plot that shows the time for insertion_sort, and `merge_sort`, and `quick_sort` on the same plot. You will need to initialize the plot first with `plotter.init()`, then call your `plot_sort_time_using_random_arrays` function three times, passing in a different sort function each time. Remember to call `plotter.plot()` in main afterwards. To stop the plotter window from closing, use `input()` to wait for keyboard input.

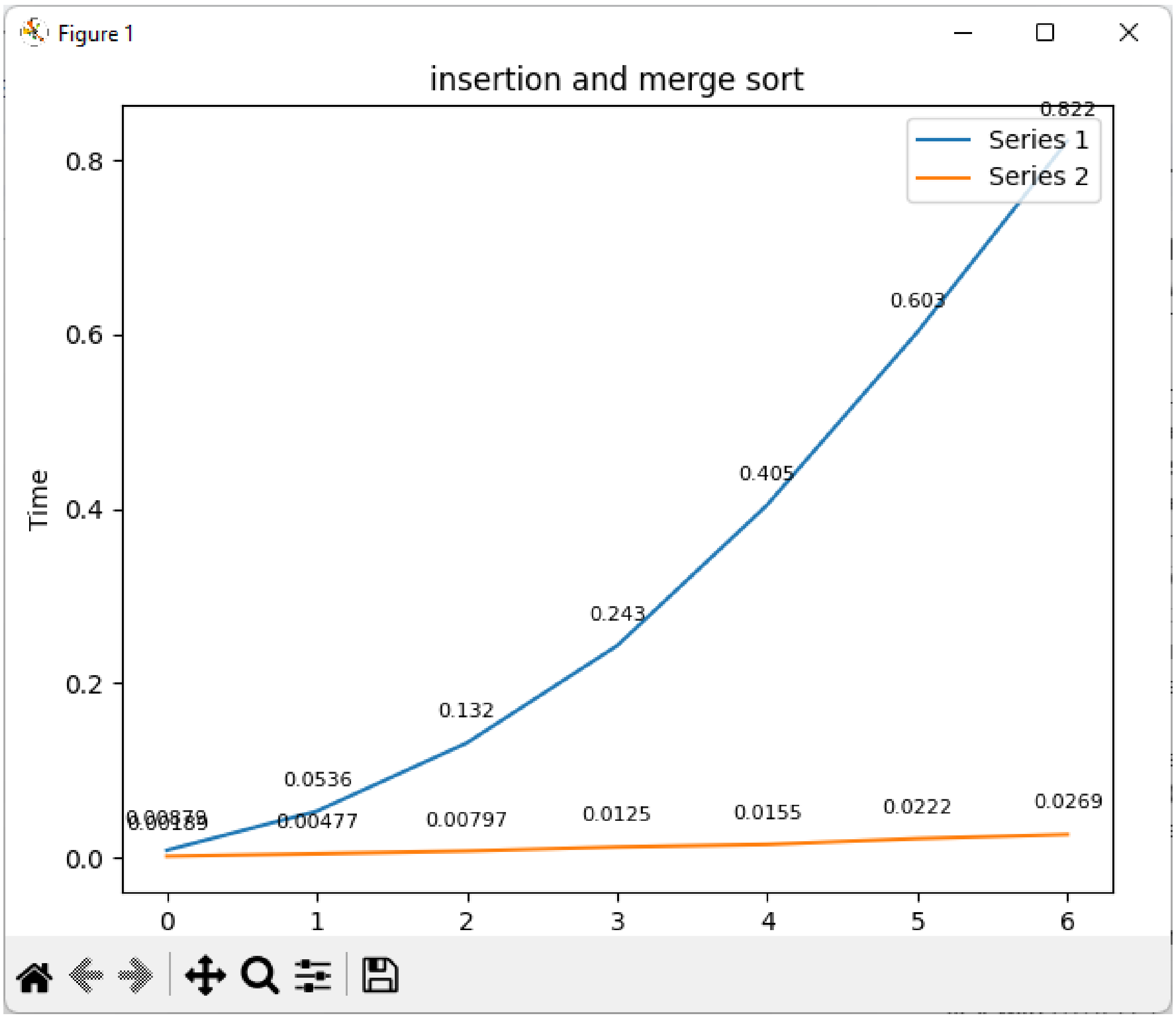

Figure 1: Example plotter output with insertion sort and merge sort

3. By now you know that merge sort and quicksort outperform insertion sort on average, we will compare them again using special arrays, i.e., sorted arrays. Add a new function to the `sort_times` module named `plot_sort_time_using_sorted_z`. This function works the same way as `plot_sort_time_using_random_arrays`, except that it uses sorted arrays. Call this from main. What happened and why? (Answer the question in a separate text file)

4. Let's focus on insertion sort and quick sort. Here is the summary you have observed so far.
   - Quicksort works well on average but it doesn't handle very well some special input arrays. In fact, no matter how you choose pivot, there will be always some input arrays that fail quicksort.
   - Insertion sort is slow in general but it runs very fast on special input, i.e., sorted arrays.

   Motivated with this observation, you will create a hybrid sort that combines the best of quicksort and insertion sort. Open the `sorts.py` file. Add a function named `quick_insertion_sort`. The overall structure of the new function will be the same as the `quick_sort` function, so you may copy-and-paste `quick_sort` and rename it. Here is a guide that helps you define the new sort function:
   - You will want to use an efficient sort as the starting sort.

- As it executes, your function should be able to sense that the quicksort approach hasn't performed well, probably because the array is sorted or nearly sorted. At this point, it will switch to insertion sort. This means that your new function will have another base case in addition to "`if len(an_array) < 2: return an_array`".

  HINT: There is no single best switching point you should use. However, there is a relationship you may use between the length of the input array and the recursion depth that holds for a well-behaving quick sort.

Don't forget to test the new function in `main` before moving to the next step.

5. Open `sort_times.py`. Compare the performance of `insertion_sort`, `merge_sort`, and `quick_insertion_sort` using random arrays. Which one works best?
   Comment out the test code and repeat the test using sorted arrays. Which one works best this time?

Submission Instructions

1. Include the appropriate internal-documentation (i.e. comments & docstring)

2. Upload **ALL the file** to the MyCourses Assignment box as (Activity03.zip) (only one team-member needs to submit to MyCourses)

3. Be sure that you have pushed **ALL the file** to your GitHub repository (**for each team-member**). Link to Assignment: [add link here]