Project 2 - Sorting Algorithms

Purpose: One claim is that for sufficiently large lists, some sorting algorithms are faster than others. For this project, you will demonstrate the speed difference between sorting algorithms listed below. You will implement the first 4 sorts yourself, and time the results. You will need to use lists large enough to show a difference in speed of 2 significant digits.

- Quicksort
- Mergesort
- Insertion Sort
- Selection Sort
- Timsort (builtin to Python)

sufficiently large means *it does not take too long and you have 2 significant digits of output* for each search algorithm.

This project is about important theoretical and practical algorithms, rather than abstract data types. Sorting and searching are at the heart of many ideas and algorithms in Computing as a Science. The timing/speed values we expect assume that you implement good versions of the algorithms.

Sort Functions

Implement the following sort functions. Verify that the parameter is a list. Assume the list contains only integers, and that the list being sorted is randomized. Each function should return the list, even though most of these are in-place, destructive sorts. Use of "lyst" as an identifier is NOT a typo since "list" is a Python type.

- quicksort(lyst): implement quicksort, return the sorted list
- mergesort(lyst): implement mergesort, return the sorted list
- selection_sort(lyst): implement selection sort, return the sorted list
- insertion_sort(lyst): implement insertion sort, return the sorted list
- is_sorted(lyst): predicate function returns True if lyst is sorted, False otherwise. In addition to verifying that lyst is a list, this should also verify that every element is an integer

You must also **use** the builtin timsort--**don't write this one yourself**

There are two sort functions which use recursion (quicksort and mergesort). You can add any additional “helper” functions needed. However, any function which is called recursively MUST have a create a RecursionCounter object at the beginning of the call:

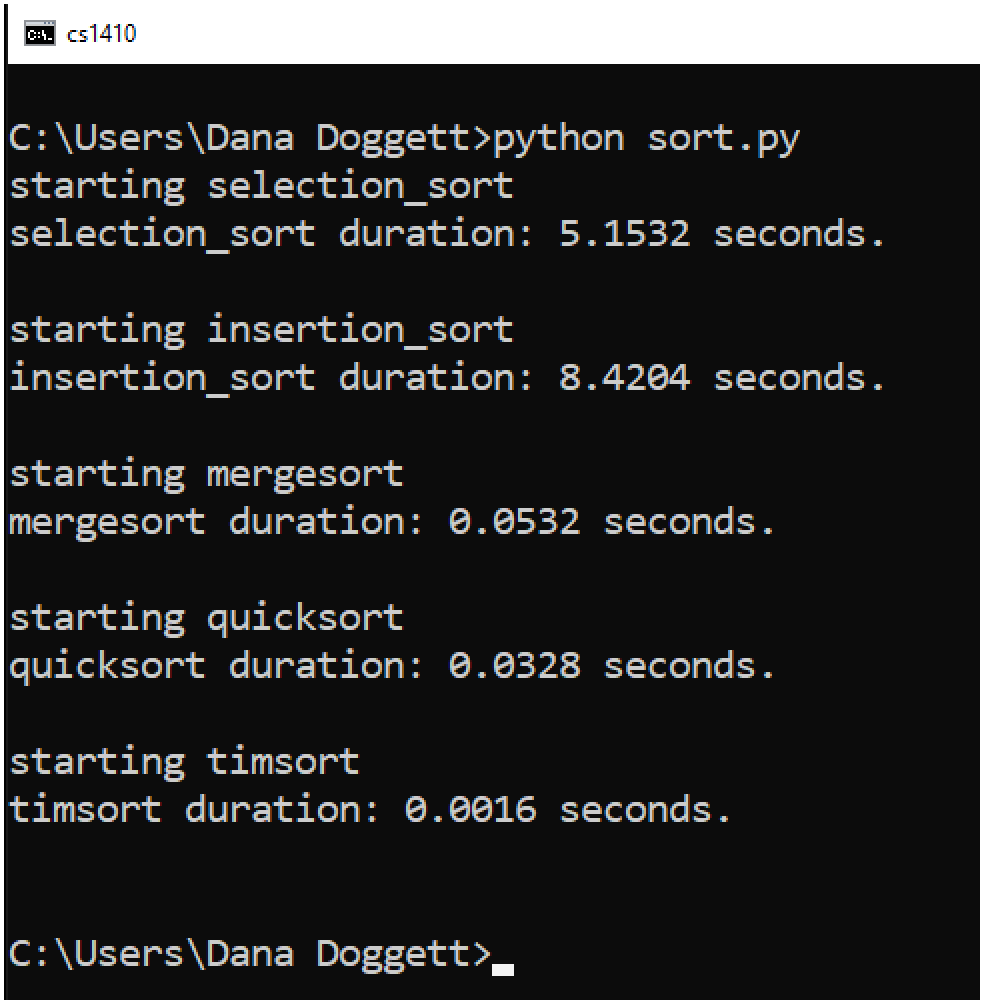
```
from recursioncounter import RecursionCounter     # needed once at the top of the module

def quicksort_helper(low, high, lyst):
    ''' the recursive part of quicksort '''
    RecursionCounter()  # needed for unittest
```

Main Program: Benchmarking

Create a **non-interactive** main function that runs each each sort, times each run, and reports the results to the console. **Be sure to randomize the list between calling successive sort functions.**

- Use large array size, typically in the range 10,000 – 50,000 values
- Your timing results should be to at least 2 significant digits.
- Report the duration for each sort routine as shown in Figure 1.
- **Do not do any printing inside the sorts as this will give incorrect timing results.**



```
C:\Users\Dana Doggett>python sort.py
starting selection_sort
selection_sort duration: 5.1532 seconds.

starting insertion_sort
insertion_sort duration: 8.4204 seconds.

starting mergesort
mergesort duration: 0.0532 seconds.

starting quicksort
quicksort duration: 0.0328 seconds.

starting timsort
timsort duration: 0.0016 seconds.


C:\Users\Dana Doggett>_
```

Figure 1. Example timing report for sorting routines. Actual sorting routines and times will be different.

Randomize the List

Use the functions in the random module for generating lists of numbers.

- random.seed(seed_value): The seed can be any integer value, but should be the same each time so that you can duplicate results for testing and debugging.
- random.sample(*population*, *k*): generates *k*-length random sequence drawn from *population* without replacement. Example: sample(range(10000000), k=60)
- be careful to not send a sorted list to the sort functions. A good way to do this is to make one unsorted list and then make a copy of that list each time you call a sort function:

  ```
  DATA_SIZE = 100000
  seed(0)
  DATA = sample(range(DATA_SIZE * 3), k=DATA_SIZE)
  ```

```
    test = DATA.copy()   # don't sort DATA, sort a copy of DATA
    print("starting insertion_sort")
    start = perf_counter()
    test = insertion_sort(test)
```

Timing functions

Use the Python 3 time.perf_count() to calculate runtimes.

Test Cases

Test Times

With a random list of 1000 integers, all four of your sort functions should returns a sorted list. The O(N^2) functions should be slower than the O(NlgN). The list's .sort() function should be faster than your O(NlgN) sort functions.

Test is_sorted()

create a random list of 1000 integers. is_sorted() should return a False. After sorting that list is_sorted() should return True.

Test Parameter Types

All sorting functions should raise a ValueError exception if their parameter is not a list. is_sorted should raise a similar exception. In addition, is_sorted() should raise a ValueError exception if it encounters an element of the list which is not an integer.

Test Coding Standards

PyLint should return a score of 8.5 or higher on your .py file.

Test Program Output

Run program Capture console output assert program output is similar to Figure 1

Grading (100 points)

scores are derived from the results of the Unit Test

• timing (relative)	20
• is_sorted function	20
• sorting	30

• parameter type validation	15
• coding standards	15

File to turn in through Canvas

sort.py (includes the driver in main())