Project 2: Benchmarking Sorting Algorithms

Problem

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. Like Project 1, this will help train your intuition for using the more abstract Big-O notation to analyze algorithms. Expected runtimes assume that you implement good versions of the algorithms.

Sort Functions

Implement the following sort functions. 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**

quicksort and mergesort are typically recursive, and helper functions are often used internally. Any helper functions should be nested inside the function that uses it, NOT at module scope. Helper functions should not be directly callable outside the parent function.

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.

Grading

pylint will be run on sort.py. Expected minimum score is 8.5.

Score is the sum of:

- Percentage of test cases passed x 80 points
- min(Coding style score/8.5, 1) x 20 points
- Possible adjustment due to physical inspection of the code, to make sure you actually implemented things.

What to Submit

- sort.py (main() with conditional execution)