CSC148, Lab #10
week of March 24th, 2014

This document contains the instructions for lab #10 in CSC148. To earn your lab mark, you must actively participate in the lab. We mark you in order to ensure a serious attempt at learning, NOT to make careful critical judgments on the results of your work. We will use the same general rules as for the first lab (including pair programming). See the instructions at the beginning of Lab #1 to refresh your memory.

overview

This lab has you compare implementations of several sorting algorithms. You will use the timeit module to estimate the total time taken. When two algorithms have very similar times, you may also use cProfile to investigate how the various functions spend their execution time.

Do not be surprised if sometimes the output of the profiler is not what you expect! Think about all the various factors that affect the profiler. Always keep in mind better ways of performing analysis of code performance. Feel free to tweak the code to try to come up with more sensible results, or write your own implementation of the algorithms to see if you can improve performance. You may also want to look at the documentation for **cProfile** or **timeit**:

`http://docs.python.org/3.3/library/profile.html`
`http://docs.python.org/3.3/library/timeit.html`

Warning! In order for the cProfile module to work correctly under Wing, you must execute your code in DEBUG mode — in other words, click on the **Debug** button instead of the **Run** button. This is due to some (in)compatibility issue. . .

sorting big-Oh

- For this part, student s1 drives and student s2 navigates.
- In CSC108 you may have seen several sorting algorithms in Python. Today, we would like you to run some sorting algorithms on inputs of various sizes, and record and compare their timing results. From these results we want you to determine which implementation of a sorting algorithm is the fastest in general, which is usually the slowest, and how these comparisons scale with the size of the list being sorted.

  Warning: We have set up the experiments with relatively small lists (multiples of 400 elements), but even so it takes a minute or so to run all the tests in Wing and get some output. If you increase the size of the lists, the running time will increase more than linearly!

- As you work through the various sorting algorithms described below, record your results on chart.xls, and plot the results on a graph.

  To graph your results, select all the rows for, say, the randomized sort. Then choose the **chart** tool from the toolbar, use the **wizard** to select points and lines as the chart type, and under **Data Range** select Data Series in rows, and first row and column as labels.

- You can find the necessary files here: sort.py, and test_sort.py.

Explain to your TA what you've discovered, and what you understand, about the various sorting algorithms' performance. In your explanation, you should consider the descriptions below of each algorithm.

sorting optimization

For this part, student s2 drives and student **s1** navigates.

Although the scaling of an algorithm on problem size **n** is determined by its big-Oh characteristic, there is often substantial time to be saved by tweaking different implementations of the same algorithm, or different algorithms in the same big-Oh class.

Several of the sorts have different variants. If you have time, you can uncomment the **profile_comparisons** code in **test_sort**, and run it under DEBUG, if you want to compare and tweak these variants.

selection sort

Selection sort works by repeatedly selecting the smallest remaining item and putting it where it belongs.

When you profile selection sort, you'll discover that a lot of time is spent calling **len**. Fix this by introducing a temporary variable in the main while loop and re-run the profiling. You'll notice that **len** is still being called a lot; find out where and use the same trick to avoid calling **len** so much. How much time did you just save?

insertion sort

Insertion sort works by repeatedly inserting the next item where it belongs in the sorted items at the front of the list. There are two versions: one manually moves items using a loop, and the other relies on Python's **del**. Why do you think Python's list code is so much faster? Of selection sort and insertion sort, which is faster? Why do you think this is?

bubblesort

Bubblesort works by repeatedly scanning the entire list and swapping items that are out of order. One consequence of bubblesort is that, on the first scan, the largest item will end up at the end of the list no matter where that item was before the first scan. Given what we've learned from timing selection and insertion sort, how do you think bubblesort will perform?

There are two versions of bubblesort. The second one has a check to see whether any items have been swapped on the last scan and, if not, stops early (in that case, no items were out of order). How much of a difference does it make to exit early? Is it noticeable? Once you've done the bubblesort timing, figure out which version is faster and why.

mergesort

Mergesort is different: it splits the list in half, sorts the two halves, and then merges the two sorted halves. There are two versions: the first one uses a helper function **_mergesort_1** that returns a new sorted list (and thus only replaces the items in the original list once, when the helper function exits), and the second one uses a helper function `_mergesort_2` that sorts the list between two indices and continually updates the original list. Which do you think is faster, and why?

quicksort

Quicksort works by partitioning the list into two halves: those items less than the first item, and those greater than or equal to the first item. For example, if the list is [5, 1, 7, 3, 9, 12], then the helper function **_partition** will rearrange the list into this: [3, 1, 5, 9, 12, 7] — notice that the 5 is now in the right place. Then the left and right sections are sorted using quicksort. How fast is this? Is quicksort faster on nearly-sorted lists or on random data? Why?

There are two versions of quicksort. The second one uses indices to sort the list in-place, without making copies of each sublist. How much difference does this make?

list.sort()

Compare Python's built-in sort to the other sorting algorithms. Why do you think the Python sort is so much faster?

nearly-sorted data, reversed data

The results hold for randomized input. Investigate whether there is an input that leads to fast or slow times: try both nearly-sorted input and reverse-sorted input. (See the comments in `test_sort.py` to find out how to generate these other kinds of inputs.)