CSC148 Lab#8, winter 2018

learning goals

In this lab you will measure the performance of algorithms for sorting lists.

set-up

Download sort.py, test_sort.py, and chart.xls, saving them to a sub-directory called **lab8**. Notice that **test_sort.py** uses module timeit to measure total time used, and cProfile to find out which parts of your code are costing the most time.

Note: If you're using Wing, there is an (in)compatibility issue that means you must run the tests in debug mode.

big-Oh characteristics of sorting algorithms

In this section you will test various sorting algorithms to see, empirically, which complexity class they fall into. The key idea is to notice how quickly the running time grows as you increase the size of the problem — the number of elements in a list in this case.

Open **test_sort.py** in your IDE, and press the run button. Various results should print in the console.

Warning: We have tried to choose the size of the lists being sorted so that the experiment takes a reasonable amount of time. You are welcome to change these parameters, but notice that the running times may increase to take longer than you are comfortable with.

As you work through the various sorting algorithms, record your results on chart.xls. 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.

Suppose n denotes the length of a list being sorted. When you're done graphing, decide which sorting algorithms take time proportional to n^2 to run ($\mathcal{O}(n^2)$), and which take time proportional to $n \lg n$ ($\mathcal{O}(n \lg n)$).

less than big-Oh

The scaling behaviour of an algorithm — how it responds to the problem size being increased — is expressed in its big-Oh class. An algorithm that grows quickly as the problem size is increased may not be feasible, no matter how much tweaking you subject the code to. Developers think **first** about big-Oh, long before optimizing.

Sometimes there **is** some time to be saved by tweaking different implementations of the same algorithm, or different algorithms in the same big-Oh class. This sort of tweaking may be appropriate at late stages of development, when the code is not being changed a lot. Or perhaps not.

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 there are many calls to **len**. Change the **while** loop to a **for** loop to avoid re-evaluating **len**, 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? Was it worth it?

insertion sort

Insertion sort works by repeatedly inserting the next item where it belongs among 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 adjacent items that are out of order. One consequence of bubblesort is that, on the first scan, the largest item must 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 sub-list. 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? You may want to google **tim sort**.