CS&E A311, Spring 2020
Assignment 3
26 points
Due 28 Feb 2020 at 11:59pm to *cse2*

This assignment will give you practice writing Python and C++ code for implementing and empirically testing the performance of sorting algorithms. Be sure to submit your files to the grading directory on cse2: `/usr/local/class/csce311s20/a3/username/`. The latest file timestamp in your directory will be used to determine the assignment submission time.

1. (14 pts) In Python, implement both HeapSort and QuickSort that sorts an array of random integers with values ranging from 0 to 100,000. They may be in the same or separate python files, but should run on cse2 by typing something like, `python3 HeapSort.py`. Implement these algorithms by referring to the pseudocode given in class or the Cormen textbook. Use function names and variables that are similar to the pseudo-code. To make it easier to match the pseudocode, ignore index 0 in your list. For each algorithm, print out the step-by-step sorting operation (e.g. unsorted, after build-heap, after top-level heapify, after partition) for a 10-element list to verify that your algorithm works properly.

Both algorithms have theoretical runtimes of O(n lg n). But based on the observed runtime, which algorithm appears to run faster?

To answer this question, make plots of the mean of 10 runtimes for n = 10000, 20000, 40000, 80000, 160000, 320000, 640000, 1280000 integers for HeapSort & QuickSort (execution times should be <1 minute, so check your code if they are taking much longer to run). One way to time your code in Python is by using the `time` package and calling `time.time()`. Add to this plot the function c*$n$lg$n$ where c is a constant that places the function $n$lg$n$ just above the HeapSort & QuickSort runtime lines. To keep the y-axis scale reasonable, you may choose a c such that c*nlgn is an upper bound for all n > n_0. For instance, c may only work for n_0 = ~100,000. Plots may be created in any tool of your choosing (e.g. Excel, matplotlib in Python, R). Be sure to include a title and label the axes. Submit this plot as a .png or .jpg file in your a3 directory.

2. (4 pts) Modify your QuickSort code to experiment with different choice of pivots. Try at least 2 methods: random element and median of first, middle, and last elements. See Cormen et al. section 7.3 for an easy way to implement the random element version. ***Before*** you implement this, write down your hypothesis as to what the effect will be. It's OK to be wrong here. For instance, if your code takes 10 seconds to sort 1 million numbers using the last-element pivot, do you think the random element or median element will speed up or slow down your code? By how much? A couple of seconds, or negligible? After you run your code, explain why you were right or wrong. Include this discussion as a long comment in your code.

3. (8 pts) In c++, compare the runtime of bubble sort to an optimized bubble sort that exits early when no swap happens in the inner loop. For each implementation, print out the sorting algorithm effects (outer loop only) on a 10 element array to show that it works properly. There are different ways to time your code, but be sure the way you use works on the cse2 server. I used `<ctime>` with a couple of calls to `clock()`.

Under what conditions does this change improve the performance of the algorithm? Generate 30 random sets of numbers (choose a size that takes a few seconds to run, ~20000), run both algorithms, and test the results. Report the mean and standard deviation of each runtime and draw a conclusion about the effectiveness of the optimization (i.e. how much does the optimization help?).