University of Washington Bothell

CSS 340: Applied Algorithmics

Program 5: Sorting

Purpose

This lab will serve two purposes. First, it will provide hands-on experience for utilizing many of the sorting algorithms we will introduce in the class. Second, it will viscerally demonstrate the cost of $O(n^2)$ v. O(n *log*n) algorithms. It will also clearly show that algorithms with the same complexity may have different running times.

Problem:

You will write a program which implements the following sorts and compares the performance for operations on lists of integers of growing sizes 10, 100, 1000, 5000, 10000, 25000, etc.... You will graph the performance of the different sorts as a function of the size of the list.

1) BubbleSort
2) InsertionSort
3) MergeSort
4) Non-Recursive, one extra list MergeSort (We'll call this improved version, IterativeMergeSort from here on out in this homework)
5) QuickSort
6) ShellSort

Details:

IterativeMergeSort

In-place sorting refers to sorting that does not require extra memory. For example, QuickSort performs partitioning operations by repeating a swapping operation on two data items in a given list. This does not require an extra list.

MergeSort as shown in class and the book allocates a temporary list at each recursive call. Due to this MergeSort is slower than QuickSort even though their running time is upper-bounded to O(n *log* n).

We can improve the performance of MergeSort by utilizing a non-recursive method and using only one additional list (instead of one list on each recursive call). In this improved version of MergeSort, *IterativeMergeSort*, one would merge data from the original list into the additional list and ***alternatively* copy back and forth between the original and the additional temporary list.** Please re-read the last sentence as it is critical to the grading of the lab.

For the IterativeMergeSort we still need to allow data items to be copied between the original and this additional list as many times as O(*log* n). However, given the iterative nature we are not building up state on the stack.

Other Sorts

BubbleSort, InsertionSort, MergeSort, ShellSort and QuickSort are well documented and you should implement them with the aid of examples in the Miller book and the slide decks.

Runtime Details

Your program, called Sorter, will take in up to three arguments:

1) sort type as a string of characters
2) the number of integers in the list
3) a string (PRINT) which will print the list pre-sort and post-sort. This argument is optional.

Your program will create and sort an integer list of the size with the specified sort: MergeSort, BubbleSort, InsertionSort, QuickSort, ShellSort or IterativeMergeSort.

Examples:

python Sorter MergeSort 100 (creates and sorts a of 100 using MergeSort)

python Sorter QuickSort 1000 (creates and sorts a list of 1000 using QuickSort)

python Sorter IterativeMergeSort 10000 (creates and sorts a list of 10000 using the newly implemented non-recursive semi-in-place MergeSort)

python Sorter BubbleSort 25 PRINT (creates a list of 25 random integers and sorts using the BubbleSort. The list is printed before and after the call)

What to turn in:

Turn in, in a .zip (not gz, etc..):

(1) sort.py which is a python module with sorts implemented

(2) Sorter.py which the driver function
(3) A separate report in word or pdf which includes: Graphs that compares the performance among the different sorting algorithms with increasing data size. You should increase the data size to clearly show the difference in performance of the different sorts.