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) Bubble Sort
2) Insertion Sort
3) Merge Sort
4) Non-Recursive, one extra list Merge Sort (We'll call this improved version, IterativeMergeSort from here on out in this homework)
5) Quick Sort
6) Shell Sort

Details:

Iterative Merge Sort

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

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

We can improve the performance of Merge Sort 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 Merge Sort, *Iterative Merge Sort*, 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 Iterative Merge Sort 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

Bubble Sort, Insertion Sort, Merge Sort, Shell Sort and Quick Sort 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 as specified below
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, **Sorter.py,** will create and sort an integer list of the size with the specified sort. The sort is specified by giving the name of the sort as the first argument to the program. Please use the following spellings:

bubble: Bubble Sort
insertion: Insertion Sort
merge: Recursive Merge Sort
imerge: Iterative Merge Sort
quick: Quick Sort
shell: Shell Sort

Examples:

python Sorter.py merge 100 (creates / sorts a of 100 using MergeSort)

python Sorter.py quick 1000 (creates / sorts a list of 1000 using QuickSort)

python Sorter.py imerge 10000 (creates and sorts a list of 10000 using the newly implemented non-recursive semi-in-place Merge Sort)

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

Your sorts should be implemented in a module called **sort** (sort.py). The signature of the sorts should follow the appropriate coding guidelines and take as input a list. You should implement the following in your sort module:

```
def bubbleSort(my_list):
def insertionSort(my_list):
def mergeSort(my_list):
def iterativeMergeSort(my_list):
def quickSort(my_list):
def shellSort(my_list):
```

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
    a. Graph or Graphs that clearly compares the performance among the different sorting algorithms with increasing data size.
    b. Short discussion / summary of what the graphs show and why
    c. Clear articulation of complexity of each sort type