CS 2500 Programming Project 2
Multiple Sorts Analysis

Monday, October 18, 2021

Be sure to read the Programming Requirements and Grading document before turning in your assignment. **Project 2 must be submitted on Canvas with all code included as Appendix A.** This is an individual assignment; you must work alone.

We will use the actual machine to validate our run-time complexity analysis and asymptotic run-time complexity analysis by comparing the input in reversed order, in sorted order, and random input case run times for sorts. We will compare Merge Sort, Quick Sort, and Heap Sort.

To complete this assignment (in addition to the standard programming requirements for all programs)

- Determine, *a priori*[1], which algorithm will have the biggest "leading constant" in its run time and validate your claim with the project.
- You should implement all three algorithms, then plot their time behavior for each case as a function of n and compare this with plots of the expected behavior.
- Determine n_0 for each algorithm as well as the leading constant.
- What inputs are required to generate "best case and worst case"? Are the inputs the same for all algorithms? If not, is this a fair comparison? Make a strong argument to support your position.
- Be sure to answer in your report: under what conditions, if any, would you choose to use each of the three sorts.

To be effective in this assignment you should read and understand Chapters 2; 3; 4.3, 4.4; and 6 thoroughly It is possible to download all this code from sites on the web, however, it's much more difficult to debug and/or insert assert() statements into somebody else's code so it makes the project much harder! If you feel you must download the code from the web, its vital to provide the source.

Grading will follow the same rubric as Project 1.

Extra Credit: For extra credit you may include in your analysis one of the following:

- Modified Quicksort to attempt to minimize the odds of a worst case. We discussed a number of ways to do this.
- Merge/Insertion sort: When the sub-problem to sort is smaller than a predetermined size it is sorted with Insertion sort rather than split into two sub-problems which are then sorted with merge sort.
- Merge Up Sort: Instead of splitting the problem into smaller and smaller sub-problems, it is possible to simply merge the items starting with sub-problems of size 1 to give sub-problems of size 2, then merge to size 4 and so on.

[1]Before the fact: State your educated guess *before* any coding which will have the best behavior for random input and large n. There is no shame in being wrong here. Remember to discuss why you made your guess and why it was right or wrong.

- In-place merge: Instead of creating two arrays (left and right) at each split to merge together, it is possible to write the first half to an axillary array and then merge the axillary array with the right half of the original array. This significantly lowers the space complexity of the merge sort.
- If you choose to include a fourth sort, you can earn up to 25 points extra credit. You must include the pseudo code for your fourth sort as well as a $O(g(n))$ analysis. I strongly urge you to get your program working for three sorts and then add the fourth.

NOTE: If you do not attach your actual code as an Appendix, I cannot effectively grade your work. If you attach your code in a poor format, it will lower your grade. I suggest using a text editor to print part of your code (a small section) to insure that the format is readable before you print the entire program. Any files, such as class files, needed to understand and run your program should be include in the Appendix. LaTeX users can include sources within the `verbatim` environment.