Project 1

In this project, you will implement insertion sort and mergesort and analyze their performance. Additionally, an implementation of timsort will be provided for you as a basis for comparison.

Implementation

The implementations that you are to develop should be able to take 3 command line arguments:

- the input file containing the unsorted numbers
- the number of elements to be sorted
- the output file to which the sorted list will be printed

There are a total of 32 files of increasing size that you will be required to sort, evenly spaced between 31250 and 1000000 (inclusive). The unsorted numbers will be provided for you, and the md5 hash of the correctly sorted 1000000 file will be made available so you can verify that your code works on that dataset. To do so, if "1000000.sorted" is the name of the file containing the result of your program, run "md5sum 1000000.sorted" and check to see if it matches the provided corresponding key which will be in the provided file "1000000.hash". You can print the contents of a file using "cat yourfilenamehere". Lastly, to be considered fully "working", your programs must be free of errors and memory leaks.

Analysis

You will be timing your implementations against the varying datasets. To do so use the unix "time" command, and use the "wall time" as your metric. Additionally, you will compute the total number of comparisons that your implementation performs during sorting. Only count comparisons that occur between elements of the array that you are sorting (i.e. you don't have to include comparisons performed naturally by a for loop checking the end condition). Once you have compiled your data, please create two graphs for each algorithm. One graph should contain the time of execution versus the size of the input, and the other should contain the number of comparisons versus the size of the input. Then for both algorithms, use your knowledge of their respective runtimes to derive two mathematical functions that approximate the data you have graphed (one function to approximate the runtime, and one function to approximate the number of comparisons. You may use matlab for this).

Speed

You will provide a second implementation of mergesort that has a faster wall time than timsort on the 1000000 dataset. You can achieve this however you like, as long as the numbers are correctly sorted, although we recommend you consider implementing a type of parallel mergesort. You do not need to provide analysis of this implementation.

Submission

You should submit all of your source files and documents containing your analysis to elms. Please do not upload the sorted or unsorted files, just the c files and your analysis. In total you need:

- insertion sort C file
- merge sort C file
- insertion sort analysis containing:

  - analysis of wall time
  - analysis of comparisons
- merge sort analysis containing:
  - analysis of wall time
  - analysis of comparisons
- faster merge sort C file

Grading

Your programs must run on the glue servers. Points will be distributed as follows:

- 10%: Providing a working implementation of insertion sort
- 10%: Providing a working implementation of merge sort
- 30%: Providing analysis for insertion sort
- 30%: Providing analysis for merge sort
- 20%: Providing an implementation of merge sort that is faster than timsort

Academic Integrity

This project is to be done individually. Any implementation ideas that are not purely your own should be cited (for an example see the timsort.h file). Please do not collaborate with other students while developing your programs, and please abide by the student honor code.