CSS 342 rcs Lab 4: Sorting performance (not big O)- Mergesort * 2 and Quicksort

Detailed Grading Guide: Use the rubric and checklist to ensure that your work earns the best score.

This is a paired programming exercise. You must include full names for both authors at the top of each piece of work submitted. See Canvas for due date. You will need to study these instructions and design your solution and test cases before starting to write code.

Purpose: Although the merge sort algorithm has the same average case execution complexity as quick sort, (i.e., *O(n log n)*), in many languages and implementations, its practical performance is much slower than quick sort due to array copying operations at each recursive call. In this programming assignment, you must improve the performance of merge sort by implementing a non-recursive, semi-in-place version of the merge-sort algorithm. Semi-in-place means you must do so by using only one temporary allocation of memory beyond that holding the data to be sorted.

In-Place Sorting: In-place sorting does not use additional arrays. It saves space and time by making fewer copies. For instance, quicksort performs partitioning operations by simply repeating a swapping operation on two data items in a given array, thus it needs no extra arrays. See (http://algs4.cs.princeton.edu/23quicksort/)

A textbook, recursive version of mergesort allocates a temporary array, sorts partial data items in that array, and copies them back to the orignal array at each recursive call. Due to this repetitive array-copying operations, mergesort is slower than quicksort although both have an average performance of *O(n log n)*. Researchers have worked to develop in-place mergesort algorithms, almost all of which are impractically complicated. You can improve the performance of mergesort by adding the following two restrictions:

1. You must use a **non-recursive method in your mergesort**
2. You must use **only one additional array**, (this is a semi-in-place method).

To do so, you must first merge data from the original into an equal sized temporary array at the very bottom stage. Thereafter, your merge must alternate between merging from the temporary into the original array and from the original into the temporary.
Requirement: You must not copy intermediate results to the original array at the end of each stage for the purpose of always merging data from the original to the temporary array; you must design your code to alternate the source and destination of the merge after each repetition. You must only copy from the temporary to the original once, if needed, just before returning the fully sorted array. You will need to move while merging data items between the original and temporary arrays , many times.

Statement of Work

Design and implement a non-recursive, semi-in-place version of the merge-sort algorithm.
Requirement: Your ***iterativemergesort( )*** **function must not call itself or any other recursive functions**. Furthermore, you must base the algorithm on the same divide-and-conquer approach in our textbook.
You can start with the **lab4driver.cpp** file to help you create the **lab4performance.cpp** to evaluate the performance of all sort methods. You must build another program, **lab4test.cpp,** to verify your non-recursive, semi-in-place mergesort program and other modules. As with all work you submit, you must clean up and make this **lab4driver.cpp** starter code meet all rubric checklist requirements. You must name your iterative program "**iterativemergesort.cpp**".
You must use a proper version of the conventional, recursive mergsort and recursive quicksort (with insertion sort) provided by your textbook author to **compare time based performance**.
Requirement: take performance data printed by your program and type it into a spreadsheet to produce a graph that **clearly** shows the relative performance of your iterative mergesort, the quicksort, the recursive mergesort. For the graph, use **at least** the following increasing array sizes: 1000, and 10,000, 100,000. Use the size as the random seed to ensure all sorts get the same data.
Produce a brief report listing timing data and explaining your results.

Detailed Grading Guide: Use the rubric and checklist to ensure that your work earns the best score.

Requirement: You must print clearly labeled time performance results for each sort.

You need not provide complete doxygen comments for each test function; but, for **all other submitted work, use full class and function doxygen style comments.** See Appendix I, Listing 1-1 and http://www.doxygen.org for examples. At the top of each file submitted, **you must list the author(s) and describe its purpose**. You must precede each function/method implementation in the .cpp file by a complete doxygen style comment. Clearly state any assumptions you make in the appropriate comment block. For example, if you assume data is to be of a given format, document this as a pre-condition in functions that input data. You must use informative variable names in all signatures/prototypes and put signatures in .h files or above main(…).

What to Turn in

Turn in **electronic copies of all these** items and a **paper copy of items 1-6**:

1. first typescript file on the linux system showing your output from **lab4test.cpp** of your improved mergesort program when size = 34 . **NOTE: you must design your .h and .cpp files so that the compile commands below work exactly as typed below**.

   ```
   script
   g++  -g -Wall lab4test.cpp …
   valgrind ./a.out …
   Ctrl+D
   ```

2. second typescript files on the Linux system showing valgrind assessment of memory leaks for your **lab4performance.cpp** (turn off all output except timing performance data unless DEBUG is the last thing on command line)

   ```
   script
   g++  -g -Wall lab4performance.cpp …
   valgrind ./a.out …..
   Ctrl+D
   ```

3. A brief report with clear printed graph that clearly compares the performance among the recursive quicksort, the recursive mergesort, and your iterative mergesort
4. your nonrecursive, semi-in-place mergesort program, as prescribed, **you must use the specified function name and file names for all sort methods so that you (and I) can test them !**
5. a file named **lab4test.cpp** that includes all your unit tests, integration tests and full system tests and must print the before and after sort, first and last 9 values for each sort algorithm.
6. files named **iterativemergesort.cpp** and **.h** that you have fully tested
7. files named **lab4performance.cpp** containing a main that uses the same set of data for each of the sort implementations: **recursivemergsort.cpp** and **.h**, **quicksort .cpp** and **.h** that you have fully tested. **lab4performance.cpp** creates, in one program run, all the data you need to produce the required comparison graph and report
8. in the unlikely case that you need them, any additional data or header files required
9. additional/clarified requirements may be updated until one week before the assignment is due

Available **in this order**: For up to 10% extra credit, add a heapsort to your comparison data and report.
For up to 10% extra credit, add a mergesort for linked lists using only O(1) extra space to your comparison data and report.
For up to 10% extra credit, add an optimized (see http://en.wikipedia.org/wiki/Merge_sort#Variants) implementation of the mergesort that uses a simple bubble sort when the size of the data becomes relatively small. Run with switch to bubble at 5 different sizes (5,7,10,15,20). Present clear graphic evidence to show which size performs best over experiments where each size gets the same data with at least 34 repetitions (use the first 34 unique Fibonacci numbers as the seed).