CS 3305
INDIVIDUAL TAKE-HOME PORTION OF FINAL EXAM
Version 1.0 as of 02:55 PM 04/15/19

OBJECTIVE:

This project is designed to afford students an opportunity to not only tweak design of an algorithm and its implementation but to also demonstrate an ability to conduct empirical analysis of the performance of algorithms. This project also affords students and opportunity to present a formal research report consisting of title page, table of contents, statement of problem, analysis, and conclusions.

RESOURCES:

There are two major resources for this task. The first is discussion in the C++ text Chapter 13 – Sorting. The second is the code archive to be git cloned from the . . ./CS3305/finalTH git remote repository located on the CS cloud server thor.

DISCUSSION OF CODE ARCHIVE:

The attached code archive is designed to test various sorting algorithms with random integer data and to provide non-trivial testing of these algorithms. The test program is designed to extract timing information for each of the actual sorts excluding the setup and verification efforts. The timing is based on availability of a high-resolution clock that can deliver timing information at the precision of nano-seconds. Thus, on Prof. Haiduk's VM, the time duration of the various sorts is measured at the precision of nano-seconds -- or 1.0e-9 seconds. Note that Prof. Haiduk's VM is configured to provide two CPUs and 4096 Gbyte of RAM. RAM is critical for some of the algorithms as the number of items to be sorted gets large.

The test program has three adjustable test parameters. These are:

- ARRAY_SIZE -- which defines the number (n) of integer items to be sorted -- which should take on values 256, 1024, 2048, 4096, 8192, 16384, 32768, and 65536 (and maybe 131072) through various runs of the test program. Note that the sizes are powers of two – this should be particularly helpful for analyses of algorithms whose complexity is represented by O(n log n).
- LIMIT -- this is a constant stating the maximum value of any of the random integers stored in the data array. Its value should be a sequence of 9s ranging from 999 to 999999 -- it is critical that this constant always be adjusted in synch with MAX_DIGITS – note handout is using hex digits which should range from 0xFFF through 0xFFFFFF.
- MAX_DIGITS -- this is a constant stating the number of digits in the constant LIMIT -- it is critical that this constant always be adjusted in synch with LIMIT.

Please note that timing information will NOT be the same from run to run due to OS overhead eating up some clock cycles. Thus, once we adjust the testing parameters, we should run the test program at least five to ten times and then take the average of those times for each sort. More on that to follow.

QUICKSORT OPTIMIZATION:

Quicksort clearly shines due to its O(n log n) performance curve. However, we may significantly reduce by some constant factor, the time required for the sort by eliminating recursive calls (or any extraneous calls for that matter). Implement a quickHelperI -- that is purely iterative -- it makes NO function calls other than call to the partition function. Make several runs to measure the improvement. Following is a pseudo-code algorithm that uses a "stack" -- please NOTE that the performance gain will occur if, and only if, you use an array of structs to hold the left and right and manage your "stack" without extraneous function calls.

```
void quickHelperI(int data[ ], int left, int right) {
    int pivotIndex;

    create the stack of appropriate fixed size;
    push(left, right, stack);

    while(!empty(stack)) {
        //take the top pair of values from the stack and set them to left and right
        pop(left, right, stack);
        if(left >= right) continue;

        if ( ( right - left) <= 10 ) {
             n = right - left + 1
             insertionsort( data+left, n )
        }
        else {
            pivotIndex = partition(data, le, ri);

            if(pivotIndex - left < right - pivotIndex) {#do small partition first
                //smaller chunk goes first
                push(left, pivotIndex - 1, stack);
                push(pivotIndex + 1, right, stack);
            }
            else {
                push(pivotIndex + 1, right, stack);
                push(left, pivotIndex - 1, stack);
            }
        }
    }
    delete stack;
}
```

Of course, you must come up with an appropriate size for your array that supports your "stack". HINT: implement this first using the STL stack and appropriate method calls. Once you have it working, then, and only then, convert it over to the internal array. Also, you must implement insertionsort. The motivated student will want to compare using the STL stack vs. using the array to see what, if any, difference in time occurs. Be aware that the STL stack requires creation of the stack and member function calls to invoke its behavior. Use Google search for C++17 STL stack. The motivated student might also want to time insertion sort.

Hint: the cleanest way to define the struct is to do so within a typedef. Thus, one might define a new type stackParms as: `typedef struct{ int left; int right; } stackParms`

ANALYSIS:

Based on the analysis, prepare a formal report that should include graphs, etc. that shows whether the performance observed matches the performance stated in text book and elsewhere. (Investigate curve fitting analyses and plots). Your report should exist in the file named report.odt (it is expected that you will use LibreOffice writer to prepare your report). Hint: You should also use LibreOffice calc to store your observations and to prepare graphs showing the relative performance. You might investigate statistical analyses for goodness of fit – do the observed points fit the expected curve with a high degree of statistical confidence?

Relative to each sort, the first thing one must do is calculate a TPE --time per element (the program can do this) for a given arraysize. Once this is calculated using the middle array size, one should be able to appropriately calculate the expected curve for smaller and larger array sizes as reference points on the curve to be plotted for the expected time. REMEMBER, only one of the algorithmic complexities is linear – O(n*k); the others are $O(N^2)$ and O(n log n) algorithmic complexities. Be sure to document the platform on which you derive your timing information. HINT: Use thor but be sure no one else is running sorttests lest your timing be spoiled by the other load on the system. The command:

```
ps axuw | grep "sorttests*"
```

will reveal whether any other user(s) is/are running the program (or editing it).

As an overall hint, let the program generate your data to be used for analysis.

Recall that you need to run your program several times for each array size and then calculate the average time for each array size. Use those times to generate a plot of the actual time as compared with the theoretically expected time.

Also, be sure your program causes NO memory leaks.

SUBMISSION:

It is expected that students will adhere to the git workflow documented previously and that their work be completed in the working directory finalTH (case sensitive) which has its remote on thor also named finalTH. For expediency, students can simply do all their work in the master branch for this project.

GRADING:

This portion of the final exam is weighted 2/3 of the final exam grade. The in-class portion of the final will be weighted 1/3 of the final exam grade. Once the in-class portion of the final exam is published, the number of points allocated to this will likely be adjusted to achieve that weighting.

ACADEMIC INTEGRITY:

This assignment is to be completed individually without help from anyone other than Professor Haiduk. Please refer to the Code of Student Life document www.wtamu.edu/codeofstudentlife/. In particular, make note of page 28 Section A: Academic Integrity Code, Item 1 parts a and b. Breach of this code will result in an automatic grade of 0 for this assignment and the grade of F for the course. Note that this applies to the acquisition or providing of information.