CS 3305
INDIVIDUAL TAKE-HOME PORTION OF FINAL EXAM
Version 1.0 as of 11:38 AM 04/23/20

OBJECTIVE:

This project is designed to afford students an opportunity to tweak design of an algorithm and its implementation.

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.

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`

DETAILS:

Understand that this project requires you implement at least two alternate versions of the optimized quick sort – one that uses the STL stack and one that uses an internal array of structs that mimics the stack ADT. You must also implement insertion sort.

You must make some modifications to the testing program to capture timing for all the alternate sort routines you implement – at least the two versions of quick sort and the insertion sort.

Create a README.docx (using LibreOffice Write or MS Word) file in your working directory in which you document everything you did toward this project. Also, include some discussion about your observations. For example, (and be quantitatively precise) how much improvement did you see in your various implementations? Run the program several times (five or more) and calculate an average timing for various number of digits in values as well as the number of values to be sorted. Which sorts consistently perform best, which worst? Are the timings consistent with the theoretically expected performance of the various sorts?

Also, be sure your various sorts cause 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). Of course this working directory should be backed up with a "remote" repository. Specifically, your remote should exist in the `~/repos/CS3305/finalTH` directory. **This is critical as a script will be written to extract your work by using a git clone. Thus, the directory path is case-sensitive.**

GRADING:

This portion of the final exam is weighted 1/3 of the final exam grade. The in-class portion of the final will be weighted 2/3 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 and Hunter Chambers. Please refer to the Code of Student Life document www.wtamu.edu/codeofstudentlife/. In particular, make note of page 25 Section A: Academic Integrity Code, Item 1 parts a, b, c and d. 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.