**CSCE 2014 – Programming Project 2**

**Midpoint Due Date – 6/15/17 at 11:59 PM**
**Final Due Date – 6/22/17 at 11:59 PM**

## 1. Problem Statement:

Quicksort is one of the most important sorting algorithms in computer science because it can efficiently work with large datasets with minimal overhead, but specific implementation details can **significantly** affect its performance in practice. For example, using the default partitioning scheme, quicksort has an $O(n\ lg_2 n)$ runtime on random data, but that performance can degrade to $O(n^2)$ for data that is already in sorted order. The default partitioning scheme will also cause significant performance losses if there are few unique values to be sorted (as might be the case when sorting by gender or birth year). In order to address these limitations, several potential improvements have been proposed:

- **Use a 3-way partition instead of a 2-way partition**. Elements are partitioned into 3 groups: < pivot, = pivot, and > pivot. If this partition scheme is implemented efficiently, performance should become significantly better when there are few unique values to be sorted.
- **Use a more intelligent pivoting scheme**. Instead of always choosing the first element or the last element in the array slice as the pivot, we can use the median value of: 1) the first element, 2) the last element, and 3) the center element. This makes it far less likely that the partition will be unbalanced, which should improve performance when values are already in sorted order.
- **Switch to a simpler sorting algorithm (such as insertion sort) when the input size is small enough**. This will reduce the overhead caused by the recursive calls considerably, which should improve performance for large datasets. You will need to decide on what "small enough" means.

For this assignment, you will perform a rigorous scientific study to determine which of these variables (or combinations of them) serve to improve the performance the most in several different environments, including: **data that is completely unsorted, data that is already sorted, and data for which there are few unique values**. In particular, you will be asked to measure the performance of each combination on one of three different datasets (that will be provided for you) in order to fill in this table:

| Method / Dataset | Unsorted | Sorted | Few Unique Values |
|---|---|---|---|
| **Baseline** | | | |
| **3-way** | | | |
| **Median** | | | |
| **Hybrid** | | | |
| **3-way + Median** | | | |
| **3-way + Hybrid** | | | |
| **Median + Hybrid** | | | |
| **3-way + Median + Hybrid** | | | |

Each cell of the table should represent the time required to execute the given algorithm variation on the appropriate dataset. The **Baseline** row represents the time required with the unmodified quicksort that was discussed in class. **Your final submission must include this table and a paragraph summarizing the results of your experiments**. When presenting results, it is often helpful to use a visual aid of some sort, so it is **highly recommended** that you include a graph of the results to make it exceedingly clear which improvements yielded the best results.

## Input

Three text files will be provided for your testing: `unsorted.txt`, `sorted.txt`, and `small.txt`. The first contains unique words in a random order. The second contains the same unique words, but in sorted order. The last file only has 5 unique values, but they are in random order. Each file corresponds to one column of the table above. The first two files contain slightly more than 45,000 words. The last one contains 50,000 words.

Each file uses the same format. One word is placed on each line. For example, this is what the first 10 lines of `unsorted.txt` looks like:

```
cessna
exercised
sanborn
slain
shearer
phototypesetters
augustly
suffolk
cruel
intimations
…
```

Your program will read a file like this into an array, perform one of the experiments, and then report the final results.

**NOTE:** You should be able to test your code by limiting the array size. For example, if you limit the array to a size of 20, you can easily see if your code is working correctly. You might also need to limit the array size if your computer is unable to allocate an array large enough to work with the entire file. If you do limit the array size, make sure to note that in your report. You should run your experiments with the entire file if it is possible to do so to get the most accurate results possible.

## Extra Credit Opportunities

C++ is a fairly unique language in that it is possible for the compiler to perform several significant optimizations to your code *during the compilation process*. These optimizations often have a drastic affect on run time performance, so they are often used in production software. Enabling compiler optimizations is very easy to do with most compilers. In g++, for example, you simply add the **–O1**, **-O2**, or **–O3** flags to the compilation command. **O1** is the lowest level of optimization, and **O3** is the highest. As the optimization level increases, so does both compilation time and performance (normally). This is an example command using the **O2** optimization level:

```
g++ -Wall –O2 main.cpp –o main
```

For a maximum of **5** points of extra credit, repeat your tests using at least one optimization level. For each optimization level you apply, you will have a separate table of results (and a corresponding conclusion paragraph/ graph). What conclusions can you make about the effect of compiler optimizations on the run time performance of your program?

### 2. Design:

You will be given more flexibility in the design of this program than in the last assignment. No particular code structure is required, but your program **MUST** meet the following objectives:

- Your data will be provided in text files. Each line of the file will contain one value to be sorted. We will only work with files that contain words in this assignment. You should **NOT** assume that every file has the same number of lines.
- Your code **MUST** include a function that reads such a file and stores the contents in a 1D array (as with the previous assignment). Your code needs to be robust to reasonable error conditions, such as a nonexistent file or an array overflow.
- You **MUST** write one or more quicksort variations in order to test all three performance enhancements. The exact means of doing so is up to you.
- Your main program should ask the user for the name of a test file and the type of test to run. It should perform the test and report the execution time to the console (e.g. in seconds or milliseconds). If you wish to write a single suite that performs all tests for a single file (e.g. fills in an entire column of the table at a time), that is also acceptable.
- Your file reading function and all sorting functions must be **templated** so they can be reused in other projects if necessary. We will only work with strings in this assignment, though.

### 3. Implementation:

You should write the necessary code to load a file into an array first. It would be wise to write a function to print the contents of the array as well to verify your file reading code works properly. Make sure to test your code thoroughly. In particular, make sure to test your code with a nonexistent file and with an array that is too small to store the entire contents of the file.

Next, you should write your main program. Remember, the main program should ask the user for the name of the file and the type of test to perform, run that test, and then report the results to the user. Start with a single test option, to perform a baseline test (using the unmodified quicksort algorithm below). Once you feel that is working correctly, you can start adding variations to the list of available options.

```
void partition(int data[], int low, int high, int &mid)
{
    // Select pivot value
    int pivot = data[high];
    int left  = low;
    int right = high;

    // Partition array into two parts
    while (left < right)
    {
        // Scan left to right
        while ((left < right) && (data[left] < pivot))
            left++;

        // Scan right to left
        while ((left < right) && (data[right] >= pivot))
            right--;

        // Swap data values
        swap(data[left], data[right]);
    }

    // Swap pivot to mid
    mid        = left;
    data[high] = data[mid];
    data[mid]  = pivot;
}

void quicksort(int data[], int low, int high)
{
    // Check terminating condition
    if (low < high)
    {
        // Partition data into two parts
        int mid = 0;
        partition(data, low, high, mid);

        // Recursive calls to sort array
        quicksort(data, low, mid - 1);
        quicksort(data, mid + 1, high);
    }
}
```

It is **EXTREMELY** important that you test each variation thoroughly to make sure you haven't accidentally broken the algorithm. All variations should give you exactly the same results. The only variable should be the run time performance of your code.

Do everything you can to make your code "bullet-proof". Try providing inputs that are egregiously wrong to see what your code does, for example. Stress tests like these are helpful for finding weaknesses and vulnerabilities. By eliminating them from the start, your code will be more robust and reliable.

**4. Style**

Make sure your code adheres to the guidelines provided in the Style Guide (available on Moodle). Your goal is to create code that is concise, descriptive, and easy for other humans to read. Avoid typos, spelling mistakes, or anything else that degrades the aesthetic of your code. Your final submission should be work that you are proud to call your own.

**5. Testing:**

Test your program to check that it operates correctly for all of the requirements listed above. Also check for the error handling capabilities of the code. Try your program with several input values, and save your testing output in text files for inclusion in your project report.

**6. Documentation:**

When you have completed your C++ program, write a short report using the project report template describing what the objectives were, what you did, and the status of the program. Does it work properly for all test cases? Are there any known problems? Save this report to be submitted electronically.

Your documentation MUST include the complete table listed above, as well as a paragraph detailing your conclusions from the experiments. It would also be a good idea to include a graph to better demonstrate which improvements were the most successful.

**7. Project Submission:**

In this class, we will be using electronic project submission to make sure that all students hand their programming projects and labs on time, and to perform automatic plagiarism analysis of all programs that are submitted.

When you have completed the tasks above go to Moodle to upload your documentation (a single **.pdf** file), and all C++ program files (**.h** and **.cpp**). Make sure your proof of testing is included in the documentation or is submitted as a separate file. Do NOT upload an executable version of your program.

The dates on your electronic submission will be used to verify that you met the due date above. Late projects will receive **NO** credit. You will receive partial credit for all programs that compile even if they do not meet all program requirements, so make sure to submit something before the due date, even if the project is incomplete.

**8. Academic Honesty Statement:**

Students are expected to submit their own work on all programming projects, unless group projects have been explicitly assigned. Students are NOT allowed to distribute code to each other, or copy code from another individual or website. Students ARE allowed to use any materials on the class website, or in the textbook, or ask the instructor for assistance.

This course will be using highly effective program comparison software to calculate the similarity of all programs to each other, and to homework assignments from previous semesters. Please do not be tempted to plagiarize from another student.

Violations of the policies above will be reported to the Provost's office and may result in a **ZERO** on the programming project, an **F** in the class, or suspension from the university, depending on the severity of the violation and any history of prior violations.