CSC 221 Spring 2007  
Program 2: Sorting Algorithms Comparison  
Due Date: Wednesday, March 21st at 11:59pm

The purpose of the lab is to reinforce your understanding and ability to use sorting algorithms, to present Unix/C++ timing and random number functions, and to introduce how to turn on compiler optimizations.

Implement the insertion sort, mergesort, and quicksort algorithms as presented in class or in the textbook, updating them to work on arrays that are holding doubles (all examples in class were for integers).

Your functions should be able to be called by this interface:

```c
double insertionsort(double* anArray, int arraySize)
double mergesort(double* anArray, int arraySize)
double quicksort(double* anArray, int arraySize)
```

Accordingly, if your sort requires more parameters than this, you should write a second function called from this function which does the actual sort. Within each `double sortName(double* anArray, int arraySize)` function, incorporate timing functions around the call to the real sorting function. Don’t include the time to generate the mergesort temporary array in your timing. The double returned from your function should be the elapsed time computed by your timing calls. A section describing how to record the execution time for a function is included at the end of this document.

In your main program, you should dynamically generate (and later cleanup) arrays of sizes 10, 100, 1000, 10,000, 100,000, 1,000,000, and 10,000,000. For any array size, you should actually create three arrays (one to pass to each sorting function).

For each size array, fill in each array with the same set of random numbers. A section describing how to generate random numbers is included at the end of this document. Once you have all three arrays filled with the same random numbers, pass each respective array to the appropriate sorting algorithm and record the time required to sort. Repeat this (fill with random then sort) process 10 times and compute and print out the average time to sort the array with each sorting method over the 10 runs. Graph your results using Excel or your favorite graphing program with array size as the x-axis and sort time as the y-axis. *Feel free to not run insertion sort for 1 million and 10 million items unless you have a lot of free time on your hands!*

The C++ compiler can optimize your code for you by removing and rearranging assembly language instructions. Different levels of optimization are allowed with g++, each of which performs a larger set of optimizations: -O, -O1, -O2, -O3 (most highly optimized), -Os (optimizes for size). With higher levels of optimization, there is a corresponding increase in compile time as the compiler works harder trying to modify
and improve your code. Recompile your program by passing the flag –O3 to the compiler. This can be done by running the compiler with the following syntax:

```bash
g++ -O3 -o outputExecutableName programName.cpp
```

Rerun your program and record and graph your results. The compiler optimization can’t make big-O level changes to your sorts, but it can often cut the time by a significant amount.

Finally, try to optimize one of your functions by hand to improve execution speed. Think back over our discussions in class on finding a better pivot, reducing swaps, and so on. In your submission of the lab, indicate which function you tried to optimize. I will compare algorithms on my computer and award a small prize to the overall fastest algorithm.

---

**How to Measure Time**

You should use the `clock()` function in Unix to determine the amount of time required to sort the arrays. You will need to `#include <time.h>` to use this function.

A simplistic way of measuring time is to store the current time before an operation starts and the current time after the operation ends, and then take the difference (end – start). This difference is the elapsed time for the operations you are measuring. An example function using `clock()` in this manner is written below.

```c
double functionWithTimeMeasurement()
{
    clock_t start, end;
    double elapsed;

    start = clock();
    // code to measure goes here
    end = clock();
    elapsed = ((double) (end - start)) / CLOCKS_PER_SEC;

    return elapsed;
}
```

**How to Generate Random Numbers**

Use the following include statement, `#include <stdlib.h>`, to bring in the random number generation functions. At the start of your `main` function, call `srand48(time(NULL))` to seed the random number generator based on the current time (this ensures different values are generated on each program execution).

Calling the function `drand48()` will then return a double between 0.0 and 1.0.