Due Week 7 Live Session.

Assignment #2: Performance Evaluation of Sorting Algorithms **version** 1.1

For this project, you are required to implement the following sorting algorithms and run a benchmark test. This will allow you to see and compare the actual running times of the algorithms we have studied and give you a deeper understanding of their inner workings. Note that you are not allowed to work in groups. All the code you submit should be your own work.

You are asked to implement the following sort algorithms: insertion, selection, bubble, bubble sort with flag, heapsort, mergesort, quicksort, radixsort, smoothsort and countingsort.

In order to evaluate all cases, you are required to read the inputs from a text file. Each line of the text file shall include 10 non-negative integers separated by a dash "-". You are required to run this analysis for three different cases and for each case you shall create a text file to read your inputs from. The input file for the first case, ascending.txt, will include integers that are already in ascending order. The input file for the second case, descending.txt, will include integers that are in descending order. And the file for the third case, random.txt, will include integers that are in random order. Your sort algorithms are expected to sort the input values in ascending order.

You are also required to run a step by step analysis, meaning you shall run sorting for only the first 1000 elements, then 2000, then 4000 and so on doubling the input size in every iteration until you sort the entire set of n numbers (n=the number of integers in the file). For each iteration of sort, e.g. 1000 elements, 2000 elements etc., you are required to measure the exact running time of each algorithm and plot your timings of each algorithm (you can use excel or Google charts for your plot). You may also consider running your algorithm three times for each iteration and taking the average runtime for that iteration.

Here is an interface you can use.

```
struct ISort {
  ISort() {}
  virtual ~ISort() {}
  virtual void sort(std::vector<int>& vector) = 0;
};
```

Each sorting algorithm should be a concrete class derived from ISort. You can use your own vector type or std::vector for this project.

Please make sure that, the time measures that you take are only for running the algorithms and do not consider the time for reading the files. You may use the attached WindowsStopwatch class to measure the time for an algorithm's runtime.

Due Week 7 Live Session.

WindowsStopWatch.h

```
#ifndef WINDOWS_STOPWATCH_H
#define WINDOWS_STOPWATCH_H
//
// Based on code from Dr. Fawcett:
//
http://www.lcs.syr.edu/faculty/fawcett/handouts/CSE687/code/HiResTimerNativeCpp/HiResT
imer.cpp
//
http://www.lcs.syr.edu/faculty/fawcett/handouts/CSE687/code/HiResTimerNativeCpp/HiResT
imer.h
//
#ifdef linux
#elif defined __APPLE_CC__
#else

#include <Windows.h>

class WindowsStopwatch {
public:
  WindowsStopwatch();
  void start();
  void stop();
  long getTime();
protected:
  __int64 a, b, f;
};

#endif

#endif
```

WindowsStopWatch.cpp

```
//
// revision history:
//
// version 1.0 - intial release
//
// version 1.1 - fix by Shawn Mueller to start the stopwatch in the constructor
//               to give similar behaviour to the unix stopwatch.
//
#include "WindowsStopwatch.h"
#include <exception>

#ifdef linux
#elif defined __APPLE_CC__
#else

WindowsStopwatch::WindowsStopwatch(){
  b = 0UL;
  if ( QueryPerformanceFrequency((LARGE_INTEGER*)&f) == 0)
    throw std::exception("no high resolution counter on this platform");

  QueryPerformanceCounter((LARGE_INTEGER*)&a);
}
```

Due Week 7 Live Session.

```
void WindowsStopwatch::start(){
  ::Sleep(0);
  QueryPerformanceCounter((LARGE_INTEGER*)&a);
}

void WindowsStopwatch::stop(){
  QueryPerformanceCounter((LARGE_INTEGER*)&b);
}

long WindowsStopwatch::getTime(){
  __int64 d = (b-a);
  __int64 ret_milliseconds;
  ret_milliseconds = (d*1000UL)/f;
  return ret_milliseconds;
}

#endif
```