1. Introduction
The purpose of this laboratory is to design and implement a timer class for which timer objects can start, stop, and report time periods to measure the times taken by two sorting algorithms to see which one executes the fastest.

2. Before You Begin
Programs contain implementations of various algorithms (such as sort, search, transform, control, and other algorithms) to efficiently process data or control external events and activities. Since the time the program takes to process data is usually dependent upon how fast the algorithms execute, it is important to choose the fastest algorithms and to implement them correctly. In this lab assignment we are interested in measuring how much time two different sorting algorithms take to sort n random integers.

Fortunately, we have some excellent estimates of how long it takes for various algorithms to execute. For example, it takes $\Theta(n^2)$ for the Bubble sorting method to sort $n$ numbers, that is, the time to execute the algorithm is proportional to $n^2$. For example, if the algorithm implementation takes 10 ms to sort 10,000 integers, it will take 40 us to sort twice as many integers. On the other hand, the QuickSort method sorts numbers in $\Theta(n \log n)$ time which is nearly linear. Your task is to measure actual run times of these two algorithms and verify that QuickSort sorts faster than Bubble sort.

To measure the elapsed execution times of the sorting algorithms, you need to create a timer class from which timer objects can be declared. Each timer object has a number of functions – start time, stop time, clear the timer, display elapsed time, and so forth – that can be used to determine the elapsed time a section of code takes to execute.

The example shown below is a program fragment that calculates the average of $n = 100$ random numbers (all code in non-bold font). Assuming we have a Timer class with the aforementioned functions, then we can declare a Timer object, start a timer (encapsulated in the object) before the random numbers are calculated, stop the timer after the numbers have been calculated, and report the elapsed execution time that (bold-font code).

```cpp
Timer randomTime; // Declare an object, randomTime, which is of type Timer
int n = 100;

// Set the random seed from the current cpu time. Needs #include <ctime>
srand( time(NULL) );

// Calculate average of n random numbers. Each number is uniformly
// distributed between 0 and 1.0
double average;
double sum = 0.0;
randomTime.start(); // Start timer
for (int i=0; i<n; i++)
    sum += rand()/static_cast<double>(RAND_MAX));
average = sum/n;
randomTime.stop(); // Stop timer; The execution interval is calculated within
                    // the object
    cout << average << endl;

    // Display "Random Average Time: 0.014534 ms elapsed time"
    randomTime.display("Random Average Time");
}
```

**Task 1.** Create a Timer class and use it to time the elapsed time for each of two different sorting methods take to sort list of randomly integers. The approach is to declare within the program doing the sorting two timer objects, one to time how long the bubble sort methods takes to sort $n$ integers, and the other to determine the time the
heapsort method takes to sort the same list of integers. Given the length of the largest list (maxLength) as a command line argument, display the execution times for both methods to sort ten lists of integers where the shortest list has length = maxLength/10, the next shortest list has length = 2*maxLength/10, …, and the longest list has length = maxLength. The program to be timed is provided to you (See appendix B). All you need to do is add the object timing functions that you create per this Task.

The Timer class should at a minimum have the following data and function members:

**Data Members (private)**
- currentTime – cpu time (obtained using the system `clock()` function – see description at end of this assignment)
- elapsedTime – current cpu time (clock()) minus time since most recent execution of the start() function
- timerOn – start() sets timeOn = true while stop() sets timeOn = false. Useful for detecting if stop() called without first executing start(), or display() called while timer still on.

**Function Members (public)**
- `start()` - If timerOn is true, then a previous call to start() is still active (ie, stop() has not been called since start() called). Ignore this condition and simply return. Otherwise, sample the cpu time, store it in currentTime, and set timerOn = true.
- `stop()` - If timerOn is false, then a previous call to stop() is still active (ie, start has not been called yet). Ignore this condition and simply return. Otherwise, sample the cpu time, calculate and store the elapsed time, and set timerOn = false.
- `display(string)` – If timerOn is true, then stop() has not been executed since the most recent start() was executed. Display an error msg and return. Otherwise, display “<string>: <elapsedTime> ms elapsed time <cr>”

Once the Timer class is coded and verified correct, insert Timer objects and calls to start(), stop(), and display() functions for each Timer object in the program available on Blackboard to correctly measure the number of milliseconds to execute two different sorting methods. The program is also given at the end of this assignment. When executed and timed, the program output should have the format (for maxLength = 10000):

```
Plot and annotate the two curves of sort time vs list length on the same plot. Use MATLAB, Excel, or some other plotting program to generate the plot. Do your plotted curves match the time complexity (i.e. \( \Theta() \)) of the sorting algorithms?

**Code Packaging**
```
Code the Timer interface in a file called Timer.cpp. Code the member function definitions for the Timer class in Timer.h.

**Reporting**
If your section requires hardcopy submission of your lab results, submit
1. All source files for this task that implement a complete and correct program,
2. A printout of the annotated plot, and
3. Your short answer to the question asked above
all in hardcopy form to your instructor or TA.

If instead your section uses an electronic archive to submit programs for grading, please submit
1. All source files for this task that implement a complete and correct program,
2. An image (.jpeg, .jpg, .gif, .bpm, .png, or .pdf) of the plot, and
3. Your short answer to the question asked above
as attached files to your email submission. Note: if you forget to attach the image file to your submission msg, you will need to resend all of the source files again along with the image file. The archive completely removes all previously submitted files whenever a new submission is received.

**Appendix A – The clock() function**
You will need to use the clock() function to obtain value of the CPU timer. Clock() returns the number of milliseconds the program has elapsed at the time clock() is called.

```c
#include <ctime>

// Returns the number of clock ticks elapsed since the program start. On failure, the function returns a value of -1.

clock_t clock ( void )

Returns the number of clock ticks elapsed since the program was launched. The macro constant expression
CLOCKS_PER_SEC specifies the relation between a clock tick and a second (clock ticks per second). The initial
moment of reference used by clock as the beginning of the program execution may vary between platforms. To
calculate the actual processing times of a program, the value returned by clock should be compared to a value
returned by an initial call to clock.

**Parameters (none)**

**Return Value** The number of clock ticks elapsed since the program start. On failure, the function returns a value of −1. clock_t is a type defined in <ctime> to some type capable of representing clock tick counts and support arithmetical operations (generally a long integer).

**Example**
```
void main ()
{
    int n;
    printf (*Starting countdown...
*);
    for (n=10; n>0; n--)
    {
        printf (*%d
*,n);
        wait (1);
    }
    printf (*FIRE!!!
*);
    return OK;
}

Output:
Starting countdown...
10
9
8
7
6
5
4
3
2
1
FIRE!!!

Appendix B. The design and code of the sorting program that is to be timed.
This program is also available in electronic form on Blackboard.

 ifndef MAIN_H_INCLUDED
#define MAIN_H_INCLUDED
// ********************************************************************
//  File:  main.h
//  
//  l8t1 - Sorts a list of random numbers using two sorting methods:
//        an inefficient 0(n^2) method (bubble sort) and an
// efficient O(nlogn) method (heap sort).
//
// The two sorting algorithms are internally timed to provide a data for a plot showing their execution times as a function of the length of the list.

#include <iostream>
#include <cstdlib>
#include <iomanip>
#include "Timer.h"
using namespace std;

const int MAXLENGTH = 1000000;

void bubbleSort(int[], int);
void heapsort(int[], unsigned int);

#include <cstdlib>
#include <ctime>

int main()
{
    int list[MAXLENGTH];
    // Populate the list with random numbers
    //... 

    // Call the bubble sort function
    bubbleSort(list, MAXLENGTH);
    //... 

    // Call the heap sort function
    heapsort(list, MAXLENGTH);
    //... 

    //... 
    return 0;
}
// OUTPUTS: 1. Display the length of the list
// 2. Display of the list of unsorted and unsorted numbers for
// each sorting method. The list displayed is truncated to
// 100 for longer lists.
// 3. Display of the time for each sorting method to sort the
// list.
//
// CONSTRAINTS: The integer values in the list to be sorted must be
// positive
//
// *********************************************************************************
// DESIGN
// 1. Display banner
// 2. Get maximum list length from command line
//   a. Ensure max list length is on the command line
//   b. Convert command line option string to number
//   c. Ensure max list length is in range 0-1,000,000 (TBR)
// 3. For list lengths of (max list length)/10, 2*(max list length)/10,
//    ..., (max list length), find the times to sort each list using
// bubbleSort and heapsort
//   a. Create list of integers to be sorted
//   b. Setup a timer for each sorting method
//   c. Sort list of integers twice and get execution time to do each sort
//      1) Sort list using the bubble sort method
//      2) Sort list using the heapsort method
// 4. Display sorted lists and execution times
// Format (example):
//                          Bubble Sort       HeapSort
//               List     Execution Time    Execution Time
//                -------     *******     *******
//                1000       0.010         0.000
//                2000       0.040         0.000
//                3000       0.160         0.020
//                ...        ...          ...
//               10000      17.866        0.250
//
// Note: the bubble sort and heapsort functions for use by this program
// are supplied in the laboratory assignment.

#include "main.h"

using namespace std;

int main(int nargs, char *argv[]) {

// 1. Display banner
//

// 2. Get maximum list length from command line
// a. Ensure max list length is on the command line
//
if (nargs != 2) {
    cout << "Max list length missing on command line, or there";
    cout << " are too many arguments on the command line. Aborting...\n\n";
    exit(EXIT_FAILURE);
}

// b. Convert command line option string to number
//
int maxLength = atoi(argv[1]);

// c. Ensure max list length is in range 0-1,000,000 (TBR)
//
if (maxLength < 2 || maxLength > MAXLENGTH) {
    cerr << "Maximum list length is either less than 2 or greater";
    cerr << " than " << MAXLENGTH << ". Aborting...\n\n";
    exit(EXIT_FAILURE);
}

// 3. For list lengths of (max list length)/10, 2*(max list length)/10,
//  ..., (max list length), find the times to sort each list using
//  bubbleSort and heapsort
//
Before starting the loop, display the header for the output table
cout << "                   Bubble Sort         HeapSort \n";
cout << "         List     Execution Time    Execution Time \n";
cout << "        Leng     (Seconds)          (Seconds) \n";
cout << "        th      \n";
int list[maxLength];
for (int length=maxLength/10; length<=maxLength; length+=maxLength/10) {
// a. Create list of integers to be sorted
//
srand(time(NULL)+length);
for (int i=0; i<length; i++) {
    list[i] = rand() % maxLength;
b. Setup a timer for each sorting method

// c. Sort list of integers twice and get execution time to do each sort
//  1) Sort list using the bubble sort method
//     a) Start bubble sort timer

bubbleSort(list, length);

//     c) Stop bubble sort timer and record elapsed time

//  2) Sort list using the heapsort method
//     a) Start heapsort timer

heapsort(list, length);

//     c) Stop heapsort timer and record elapsed time

//  4. Display sorted lists and execution times
//     Format (example):
//                          Bubble Sort       HeapSort
//               List     Execution Time    Execution Time
cout << "       " << setw(6) << right << length;  
cout << fixed << setprecision(3);  
cout << "       " << setw(6) << right;  
<----- elapsed time to sort <length> integers by bubble sort goes here>  
cout << "       " << setw(6) << right;  
<----- elapsed time to sort <length> integers by heapsort sort goes here>  
cout << endl;  
} // end for (int length= )

return EXIT_SUCCESS;

}  // bubbleSort()
void heapsort(int arr[], unsigned int N) {
// From http://codecodex.com/wiki/Heapsort#C.2B.2B

unsigned int n = N, i = n/2, parent, child;
int t;

for (;;) { /* Loops until arr is sorted */
    if (i > 0) { /* First stage - Sorting the heap */
        i--;           /* Save its index to i */
        t = arr[i];    /* Save parent value to t */
    } else { /* Second stage - Extracting elements in-place */
        n--;           /* Make the new heap smaller */
        if (n == 0) return; /* When the heap is empty, we are done */
        t = arr[n];    /* Save last value (it will be overwritten) */
        arr[n] = arr[0]; /* Save largest value at the end of arr */
    }

    parent = i; /* We will start pushing down t from parent */
    child = i*2 + 1; /* parent's left child */

    /* Sift operation - pushing the value of t down the heap */
    while (child < n) {
        if (child + 1 < n && arr[child + 1] > arr[child]) {
            child++; /* Choose the largest child */
        }
        if (arr[child] > t) { /* If any child is bigger than the parent */
            arr[parent] = arr[child]; /* Move the largest child up */
            parent = child; /* Move parent pointer to this child */
            child = parent*2 + 1; /* Find the next child */
        } else {
            break; /* t's place is found */
        }
    }

    arr[parent] = t; /* We save t in the heap */
}