CMSC 2123 Discrete Structures

Project 3

Due: See the due date on D2L.

1. Basic Requirements

We will combine the Insertion Sort and Binary Search that we learned.
The pseudocode for Insertion Sort:

```
procedure insertion sort (a_1,…,a_n: real numbers with n ≥ 2)
    for j := 2 to n
        i := 1
        while a_j > a_i
            i := i + 1
        m := a_j
        for k := 0 to j − i − 1
            a_{j-k} := a_{j-k-1}
        a_i := m     {Now a_1,…,a_n is in increasing order}
```

The pseudocode for Binary Search:

```
procedure binary search(x: integer, a_1,a_2,…, a_n: increasing
    integers)
    i := 1       {i is the left endpoint of interval}
    j := n       {j is right endpoint of interval}
    while i < j
        m := ⌊(i + j)/2⌋
        if x > a_m then i := m + 1
        else j := m
    if x = a_i then location := i
    else location := 0
    return location  {location is the subscript i of the term a_i
equal to x, or 0 if x is not found}
```

The current method in Insertion Sort to locate the insertion point is based on linear search. This is reflected by the inner while loop of the Insertion Sort (red colored code segment).

The idea of this project is to replace this part with Binary Search, since all the elements before the target element a_j are sorted. To achieve this target, you need to make some modification to the original Binary Search. We call this new combined algorithm as **Binary Insertion Sort**. **Do not use recursive implementation**.

The program framework is provided, you need to implement Insertion Sort and Binary Insertion Sort. And make a brief comparison (see Section 3 Submissions).

```
/*
 * Your implemention of the insertion sort.
 */
void MyInsertionSort(int* arr, int size) {
  /*** TODO: put your code here ***/
}

/*
 * Your implemention of the binary insertion sort.
 */
void MyImprovedSort(int* arr, int size) {
  /*** TODO: put your code here ***/
}
```

2. Program Structure

The program structure is provided, and you need to implement the TODO parts. The test arrays **are randomly generated** by the given program. And there could be **duplicated numbers** in the test arrays. Your task is to accept the replicas and sort these arrays.

2.1 Program Files

Project **3** consists of the file **p03.cpp.**

You can choose to use the provided source code, or implement everything by yourself. You can also make any changes to the provided source code **as long as the input and output are consistent**. There are some unimplemented parts in the given source code, and you need to finish those parts to make the program complete. They are marked as "**TODO**" in the comments.

2.2 Command Line

To compile your source code, you can use:

```
$ g++ -o p03 p03.cpp
```

To execute your program after a successful compilation, you can use:

```
$ ./p03 unsorted.dat sorted1.dat sorted2.dat
time1.dat time2.dat
```

This is the way how we use command line argument to get input parameters from the users. You can find further information regarding command line argument from:

http://www.cplusplus.com/articles/DEN36Up4/

This part is similar as Project 2.

Here there are 5 parameters: (1). name of the file for unsorted arrays, e.g. unsorted.data, (2). name of the file for sorted arrays using Insertion Sort, e.g. sorted1.dat, (3). name of the file for sorted arrays using Binary Insertion Sort, e.g. sorted2.dat, (4). name of the file for time consumed by algorithm 1 (Insertion Sort), e.g. time1.dat, (5). Name of the file for time consumed by algorithm 2 (Binary Insertion Sort), e.g. time2.dat**. How to read and write these input and output files has already been implemented, you just need to worry about the implementation of the algorithms.**

2.3 Input and Output Specification

The input and output parts have been implemented.

The arrays are randomly generated, and then referenced by your algorithms. Before you start sorting, these generated arrays are written to the file for unsorted arrays (unsorted.dat in the example above).

Later the sorted arrays are stored to different files for different algorithms (sorted1.dat and sorted2.dat in the example above).

Meanwhile the time consumed by each algorithm is also recorded and is saved to the file for time consumption (time.dat in the example above).

Please note:

In this implementation, there are **arrays of different sizes, and 20 different arrays for each size**. In unsorted.dat, each line is an array. You can easily figure out the size of the array (from the loop control in the source code or from the number of elements in each row of unsorted.dat). The idea of "20 different arrays for each size" is to get the **average execution time**: the input arrays are randomly generated, so there is also randomness in the performance data collected, and to better evaluate the performance we need to get the average performance for each specific array size.

Arrays in sorted1.dat and sorted2.dat follow the same format as the ones in unsorted.dat. Corresponding arrays have the same line number in these files, e.g. the array in

the first line of unsorted.dat is sorted and save as the array in the first line of sorted1.dat using Insertion Sort and as the array in the first line of sorted2.dat using Binary Insertion Sort.

Time1.dat and time2.dat contain the time used by each algorithm. The time is in **millisecond**. Each line is the time used by **a single run** of the corresponding algorithm. Please **pay attention**: to get the average execution time, you need to group multiple lines properly.

3. Submission

Please make your submission on D2L. This report should include your name(s) + email(s), instructions to compile and execute your program (e.g., information about your operating system, command for compilation, etc.), and the result of a sample execution.

For this project, you also need to include a **graph** for the comparison of these two algorithms. In this graph, please include two curves showing the time used for different sizes of the input array. For example, the horizontal axis is the size of the array, and the vertical axis is the time used. Please draw the two curves in the same graph. You can use any tool to draw the graph, e.g. use Microsoft Excel: https://www.youtube.com/watch?v=WaNMnDHbTI0 (maybe you can try "Scatters with smooth lines and markers")

The data in time1.dat and time2.dat can be used to generate the performance graphs of these two algorithms. Please pay attention: we have multiple runs of each algorithm for an input array with a certain size, please use the **average time** for that size in this graph. So you need to parse the time correctly (you can do this manually or write another program or by Microsoft Excel).

In your report, please also make a short analysis about the **performance** of the Binary Insertion Sort in term of **big-O notation or big-Theta notation**.

All the files need to be zipped as p03_group*.zip, where * means your group number, e.g., the submission from group 7 should be named as p03_group7.zip. Please organize the file as:

```
p03_group*.zip
+-----group* (a folder)
      +------ report.pdf
      |____ src (a folder for the source code)
```

This is a teamwork, you can have a partner and register to a group on D2L. Please check D2L course homepage -> "Communication" -> "Groups" -> "Project Groups", and enroll yourself in a group.

4. Evaluation

This project will be evaluated according to the correctness of the various tasks specified above, and secondarily according to the quality of your code. The rubric is as follows:

Categories		Weights	
Algorithm 1		30%	
	Translation of the pseudocode		20%
	Correctness		10%
Algorithm 2		40%	
	Translation of the pseudocode		20%
	Combination of two algorithms		10%
	Correctness		10%
Fairness of the comparison		10%	
Report		20%:	
	The comparison graph		10%
	Big-O analysis		5%
	The rest		5%
Bonus on coding style		+10%	

Notes:

1. To be considered on time, the program must be turned in before the due date.
2. Programs must reflect your knowledge and work, not others.
3. No points, zero (0), will be given to any program containing compilation errors.