CS 202, Fall 2017

Homework #1 – Algorithm Efficiency and Sorting

Due Date: October 16, 2017

Important Notes

Please do not start the assignment before reading these notes.

- Before 23:55, October 16, upload your solutions in a single **ZIP** archive using Moodle submission form. Name the file as `student_id.zip`.
- Your ZIP archive should contain the following files:
  - `hw1.pdf`, the file containing the answers to Questions 1 and 3,
  - `sorting.h` and `sorting.cpp` files which contain the C++ source codes, and
  - `readme.txt`, the file containing anything important on the compilation and execution of your program in Question 2.
  - Do not forget to put your name, student id, and section number in all of these files. Well comment your implementation. Add a header as in Listing 1 to the beginning of each file:

Listing 1: Header style

```
/**
 * Title: Algorithm Efficiency and Sorting
 * Author: Name Surname
 * ID: 21000000
 * Section: 0
 * Assignment: 1
 * Description: description of your code
 */
```

  - Do not put any unnecessary files such as the auxiliary files generated from your favorite IDE. Be careful to avoid using any OS dependent utilities (for example to measure the time).
  - **You should prepare the answers of Questions 1 and 3 using a word processor (in other words, do not submit images of handwritten answers).**

- Although you may use any platform or any operating system to implement your algorithms and obtain your experimental results, your code should work in a Linux environment (specifically using the g++ compiler). We will compile your programs with the g++ compiler and test your codes in a Linux environment. Thus, you may lose significant amount of points if your C++ code does not compile or execute in a Linux environment.

- This homework will be graded by your TA, Ilkin Safarli. Thus, please contact him directly for any homework related questions.

Attention: For this assignment, you are allowed to use the codes given in our textbook and/or our lecture slides. However, you ARE NOT ALLOWED to use any codes from other sources (including the codes given in other textbooks, found on the Internet, belonging to your classmates, etc.). Furthermore, you ARE NOT ALLOWED to use any data structure or algorithm related function from the C++ standard template library (STL).

Do not forget that plagiarism and cheating will be heavily punished. Please do the homework yourself.

Question 1 – 25 points

(a) [*10 points*] Sort the functions below in the increasing order of their asymptotic complexity: $f_1(n) = 10^{\pi}$, $f_2(n) = n$, $f_3(n) = \sqrt{n}$, $f_4(n) = \log n$, $f_5(n) = n^{0.0001}$, $f_6(n) = n \log n$, $f_7(n) = 2^n$, $f_8(n) = n!$, $f_9(n) = \log(n!)$, $f_{10}(n) = n^n$

(b) [*10 points*] Express the running time complexity (using asymptotic notation) of each loop separately. Show all the steps clearly.

Listing 2: Loops

```
// Loop A - 3 points
for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j++)
        sum = sum + 1;

// Loop B - 3 points
i = 1;
while (i < n) {
    for (j = 1; j <= i; j++)
        sum = sum + 1;
    i *= 2;
}
```

```
// Loop C - 4 points
i = 1;
while (i < n) {
    j = 1;
    while (j < i*i) {
        for (k = 1; k <= n; k++)
            sum = sum + 1;
        j *= 2;
    }
    i *= 2;
}
```

(c) [*5 points*] Write the recurrence relation of merge sort and quick sort algorithms for the worst case, and solve them. Show all the steps clearly.

Question 2 – 50 points

(a) [*35 points*] You will implement insertion sort, merge sort, and quick sort algorithms. Your functions should take an array of integers and the size of that array and then sort it in increasing order. Add two counters to count the number of key comparisons and the number of data moves during sorting. For the quick sort algorithm, you are supposed to take *the last element* of the array as pivot. Your functions should have the following prototypes:

```
void insertionSort(int *arr, int size, int &compCount, int &moveCount);
void mergeSort(int *arr, int size, int &compCount, int &moveCount);
void quickSort(int *arr, int size, int &compCount, int &moveCount);
```

For key comparisons, you should count each comparison like $k_1 < k_2$ as one comparison, where k_1 and k_2 correspond to the value of an array entry (that is, they are either an array entry like `arr[i]` or a local variable that temporarily keeps the value of an array entry).

For data moves, you should count each assignment as one move, where either the right-hand side of this assignment or its left-hand side or both of its sides correspond to the value of an array entry.

(b) [*15 points*] In this part, you will analyze the performance of the sorting algorithms that you implemented in part a. Write a function named `performanceAnalysis` which does the followings:

1. Create three identical arrays with random 1000 integers (use `rand` from `cstdlib`). Use one of the arrays for the insertion sort, another one for the merge sort, and the last one for the quick sort algorithm. Output the elapsed time in milliseconds, the number of key comparisons, the number of data moves (use `clock` from `ctime` for calculating elapsed time). Repeat the experiment for the following sizes: $\{5000, 10000, 15000, 20000\}$

2. Now, instead of creating arrays of random integers, create arrays with elements in ascending order and repeat the steps in part b1.
3. Lastly, instead of creating arrays of random integers, create arrays with elements in descending order and repeat the steps in part b1.

When `performanceAnalysis` function is called, it needs to produce an output similar to the following one:

Listing 3: Sample output

```
Part b1 - Performance analysis of random integers array
---------------------------------------------------------
                    Elapsed Time  compCount  moveCount
Array Size: 1000
Insertion Sort
Merge Sort
Quick Sort
---------------------------------------------------------
                    Elapsed Time  compCount  moveCount
Array Size: 5000
Insertion Sort
Merge Sort
Quick Sort
...

Part b2 - Performance analysis of ascending integers array
---------------------------------------------------------
                    Elapsed Time  compCount  moveCount
Array Size: 1000
Insertion Sort
Merge Sort
Quick Sort
...
```

All of your code related to this question, goes into `sorting.cpp` file. You are free to write helper functions to accomplish the tasks required from the above functions. Use the given file names and function signatures during implementation. You will not submit a `main.cpp` file, instead we will use our `main.cpp` file during evaluation. Therefore, it is important to use given file names and function signatures.

A minimal `main.cpp` file and `sorting.h` header file are provided with the assignment. Please make sure that your solution works with this `main.cpp` file.

Question 3 – 25 points

After running your programs, you are expected to prepare a $3-4$ page report about the experimental results that you obtained in Question 2 b. First, create a table similar to Table 1 and fill it with the results you obtained. [1]

Table 1: Sorting Algorithms Performance Table

Array	**Elapsed Time (in milliseconds)**			**Number of Comparisons**			**Number of Data Moves**		
	Insertion Sort	*Merge Sort*	*Quick Sort*	*Insertion Sort*	*Merge Sort*	*Quick Sort*	*Insertion Sort*	*Merge Sort*	*Quick Sort*
R1K									
R10K									
R20K									
A1K									
A10K									
A20K									
D1K									
D10K									
D20K									

Then, with the help of a spreadsheet program (Microsoft Excel, Matlab or other tools), plot *elapsed time* versus *the size of array*. Note that you will need to plot 3 figures, one for each array type (ascending, descending and random). A sample figure is given below (*these values do not reflect real values*):

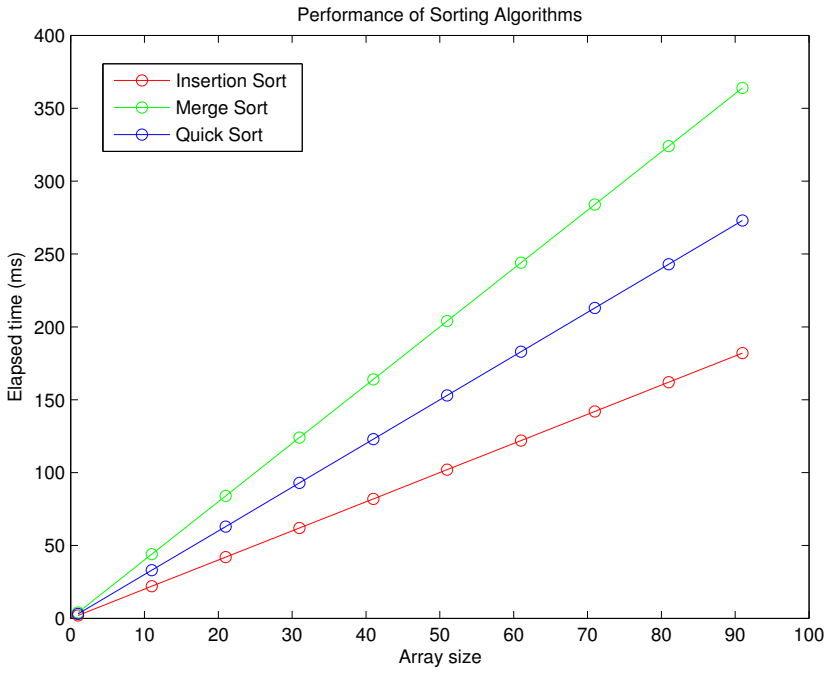

Figure 1: Performance analysis for array with random elements

[1] $R1K$ - array with 1000 random integers, $A1K$ - array of 1000 ascending integers, $D1K$ - array of 1000 descending integers.

Interpret and compare your empirical results with the theoretical ones for each sorting algorithm. Explain any differences between the empirical and theoretical results, if any. Finally, answer the following questions:

- When should insertion sort algorithm be preferred over merge sort and quick sort algorithms?
- When should merge sort algorithm be preferred over quick sort algorithm?