CS 202, Fall 2018

Homework #1 – Algorithm Efficiency and Sorting

Due date: October 16, 2018

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 the Moodle submission form. Name the file as **`studentId_hw1.zip`**.

- Your ZIP archive should contain the following files:
  - ✓ **`hw1.pdf`**, the file containing the answers to Questions 1 and 3,
  - ✓ **`sorting.h`**, **`sorting.cpp`** and **`main.cpp`** files which contain the C++ source codes,
  - ✓ **`Makefile`**,
  - ✓ **`readme.txt`**, the file containing anything important on the compilation and the 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 the following to the beginning of each file:

```
/*
 * 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 on the **dijkstra** server (`dijkstra.ug.bcc.bilkent.edu.tr`). We will compile

and test your programs on that server. Thus, you will lose significant amount of points if your C++ code does not compile or execute on the **dijkstra** server.

- This homework will be graded by your TA, Hasan Balcı. 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) [5 points] Show that $f(n) = 4n^5 + 3n^2 + 1$ *is* $O(n^5)$ by specifying appropriate c and n_0 values in Big-O definition.

(b) [10 points] Find the asymptotic running times (in Θ notation, tight bound) of the following recurrence equations by using the repeated substitution method. Show your steps in detail.

➔ $T(n) = T(n - 1) + n^2$, $T(1) = 1$

➔ $T(n) = 2\ T(n/2) + n/2$, $T(1) = 1$

(c) [10 points] Trace the following sorting algorithms to sort the array *[8, 4, 5, 1, 9, 6, 2, 3]* in *ascending order*. Use the array implementation of the algorithms as described in the textbook and show all major steps (after each sort pass for instance).

➔ Selection sort

➔ Bubble sort

Question 2 – 55 points

You are asked to implement the following algorithms for *an array of integers* and then perform the measurements as detailed below:

- ➔ **insertion sort (10 points)**
- ➔ **merge sort (10 points)**
- ➔ **quick sort (10 points)**
- ➔ **hybrid sort (10 points):** This hybrid sort algorithm starts with the quick sort, but when the partition size becomes less than or equal to 10, sorts that partition with the insertion sort. Please see a sample implementation in LISTING 11-5 of the textbook.

- For each algorithm, implement the functions that take an array of integers, the index of the first and last element to consider in the array and then sort it in *non-descending (ascending) order*. Add two counters to count and return the number of key comparisons and the number of data moves during sorting.
- For key comparisons, you should count each comparison like "`k1 < k2`" as one comparison, where `k1` and `k2` 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. For example, the following swap function has three such assignments (and thus three data moves):

```
void swap(DataType &x, DataType &y) {
   DataType temp = x;
   x = y;
   y = temp;
}
```

- For the quick sort and hybrid sort algorithms, you are supposed to take **the last element** of the array as the pivot.

- After implementing the sorting algorithms, implement a function named **`performanceAnalysis` (15 points)** which takes the array size as parameter and does the following:
  1. Create four identical arrays, whose size taken from the parameter, with random integers using the random number generator function **`rand`**. Use one of the arrays for the insertion sort, another one for the merge sort, another one for the quick sort and the last one for the hybrid sort algorithm. Output the elapsed time in milliseconds, the number of key comparisons and the number of data moves (use clock from ctime for calculating elapsed time).
  2. Now, instead of creating arrays of random integers, create arrays with elements in ascending order and repeat the steps in part 1.
  3. Now, instead of creating arrays of random integers, create arrays with elements in descending order and repeat the steps in part 1.
  4. Run the experiment (parts 1-3) for the following array sizes: {1000, 7000, 14000, 21000}, given as input to **`performanceAnalysis`** (total of four different sizes).

For example, when the **`performanceAnalysis`** function is called with the array size 1000, it needs to produce an output similar to the following one:

```
Performance analysis for arrays of size 1000
---------------------------------------------------------------------------------------------
Random integers     Elapsed time     compCount     moveCount
Insertion sort
Merge sort
Quick sort
Hybrid sort
---------------------------------------------------------------------------------------------
Ascending integers  Elapsed time     compCount     moveCount
Insertion sort
Merge sort
Quick sort
Hybrid sort
---------------------------------------------------------------------------------------------
Descending integers Elapsed time     compCount     moveCount
Insertion sort
Merge sort
Quick sort
Hybrid sort
```

- Put the implementations of these functions in a file named **`sorting.cpp`**, and their interfaces in a file named **`sorting.h`**. Also write a main function separately inside a file named **`main.cpp`** that calls only the **`performanceAnalysis`** function with four different input sizes mentioned above.
- Although you will write your own main function to get the experimental results, we will also write our own main function to test whether or not your algorithms work correctly. In our main function, we will call your sorting algorithms with the following prototypes:

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

In all of these prototypes, **`arr`** is the array that the algorithm will sort, **`first`** is the index of the first element to consider in **`arr`**, **`last`** is the index of the last element to consider in **`arr`**, **`compCount`** is the number of key comparisons in sorting, and **`moveCount`** is the number of data moves in sorting. After returning from this function, **`arr`** should become sorted.

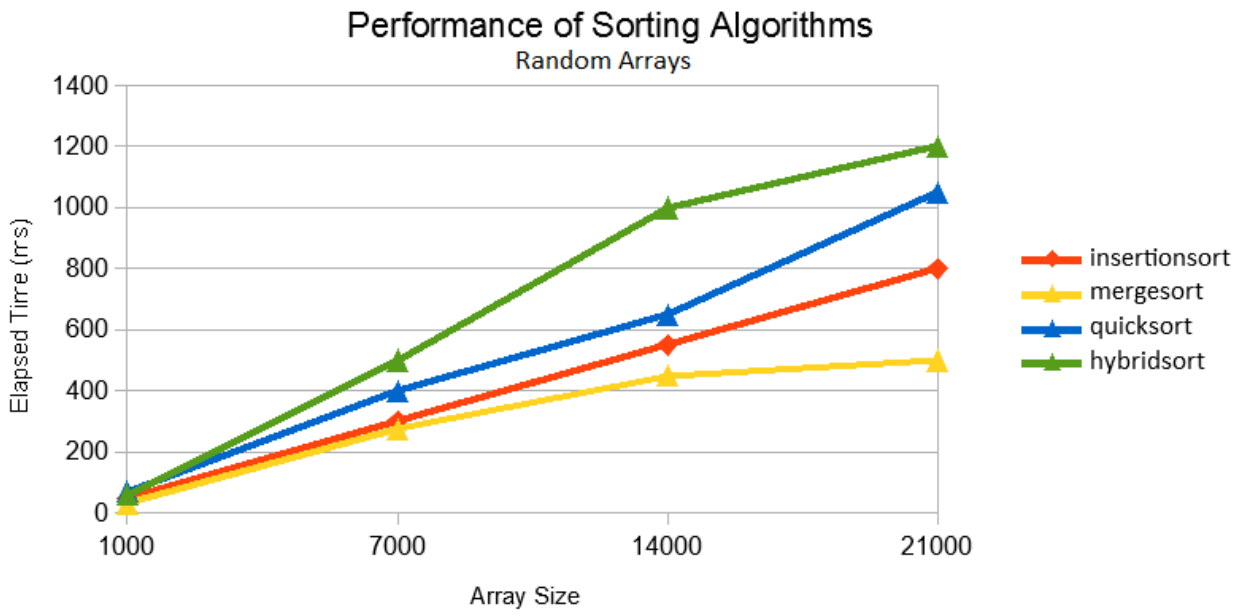
IMPORTANT: At the end, write a basic **`Makefile`** which compiles all your code and creates an executable file named **`hw1`**. Check out these tutorials for writing a simple make file:
http://mrbook.org/blog/tutorials/make/,
http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/.

Question 3 – 20 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. First, prepare tables for presenting the results for the number of key comparisons, the number of data moves, and the elapsed time. You should prepare a separate table for each required number. For each table, each row should include the type of the input (e.g., R1K - array with 1000 random integers, A1K - array with 1000 ascending integers, D1K - array with 1000 descending integers etc.) and the values obtained by insertion sort, merge sort, quick sort and hybrid sort in four separate columns. A sample table can be like:

Arrays	**Elapsed Time (in miliseconds)**			
	Insertion Sort	**Merge Sort**	**Quick Sort**	**Hybrid Sort**
R1K				
R7K				
R14K				
R21K				
A1K				
A7K				
A14K				
A21K				
D1K				
D7K				
D14K				
D21K				

Then, with the help of a spreadsheet program (Microsoft Excel, Matlab or other tools), plot *elapsed time* versus t*he 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):



Your report must also answer/explain the following points briefly:

- Interpret and compare your empirical results with the theoretical ones for each sorting algorithm. Explain any discrepancies between the empirical and theoretical results, if any.
- In general, when should insertion sort algorithm be preferred over merge sort and quick sort algorithms?
- In general, when should merge sort algorithm be preferred over quick sort algorithm?
- Does hybrid sort algorithm have any advantages/disadvantages over the quick sort algorithm? Explain the reasons, if any.