CS 202, Spring 2019

Homework #1 – Algorithm Efficiency and Sorting

Due Date: March 9, 2019

Important Notes

Please do not start the assignment before reading these notes.

- Before 23:55, March 9, upload your solutions in a single **ZIP** archive using 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, 2 and 3.
  - `sorting.cpp, sorting.h, main.cpp` files which contain the C++ source codes, and the `Makefile`.
  - 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: 1
 * 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).
  - Use the exact algorithms shown in lectures.

- 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. Please make sure that you are aware of the homework grading policy that is explained in the rubric for homeworks.

- This homework will be graded by your TA, Mubashira Zaman. Thus, please contact her 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 $f(n) = 100n^3 + 8n^2 + 4n$ is $O(n^4)$ by specifying appropriate c and n_0 values in Big-O definition.

(b) [*5 points*] Find the asymptotic running time (in O notation, tight bound) of the recurrence equation $T(n) = 8T(n/2) + n^3$ where $T(n) = 1$ for all $n \leq 100$. Use the repeated substitution method and show your steps in detail.

(c) [*5 points*] Express the running time complexity (using asymptotic notation) of the following loop. Show all the steps clearly.

```
for (i = n; i > 0; i /= 2) {
    for (j = 1; j < n; j++) {
        for (k = 1; k < n; k += 2) {
            sum += (i + j * k); } } }
```

(d) [*10 points*] Trace the following sorting algorithms to sort the array [16, 6, 39, 21, 10, 21, 13, 7, 28, 19] in ascending order. Use the array implementation of the algorithms as described in the textbook and show all major steps.

- Selection sort
- Insertion sort

Question 2 – 60 points

Implement the following methods in the sorting.cpp file:

(a) [*40 points*] Implement the radix sort, bubble sort, quick sort, and merge sort algorithms. Your functions should take an array of integers and the size of that array and then sort it in the ascending order. Add two counters to count and return the number of key comparisons and the number of data moves for all sorting algorithms except the radix sort. For the quick sort algorithm, you are supposed to take the first element of the array as pivot. Your functions should have the following prototypes:

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

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 (e.g. a swap function mostly has 3 data moves).

(b) [*5 points*] Implement a function that prints the elements in an array.

```
void printArray(int arr, int size);
```

(c) [*10 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 creates four identical arrays of 2000 random integers. Use one of the arrays for the radix sort, second for bubble sort, third for the merge sort, and the last one for the quick sort algorithm. Output the elapsed time (in milliseconds), the number of key comparisons and the number of data moves. Number of key comparisons and

number of data moves are not required for the radix sort. Repeat the experiment for the following sizes: 6000, 10000, 14000, 18000, 22000, 26000, 30000.

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

Listing 2: Sample output

```
-----------------------------------------------------
Part c - Time analysis of Radix Sort
Array Size      Time Elapsed
2000                x ms
6000                x ms
...
-----------------------------------------------------
Part c - Time analysis of Bubble Sort
Array Size      Time Elapsed    compCount   moveCount
2000                x ms            x           x
6000                x ms            x           x
...
-----------------------------------------------------
Part c - Time analysis of Quick Sort
Array Size      Time Elapsed    compCount   moveCount
2000                x ms            x           x
6000                x ms            x           x
...
-----------------------------------------------------
Part c - Time analysis of Merge Sort
Array Size      Time Elapsed    compCount   moveCount
2000                x ms            x           x
6000                x ms            x           x
...
-----------------------------------------------------
```

(d) [*5 points, mandatory*] Create a `main.cpp` file which does the following in order:

- creates an array from the following numbers: $\{7, 3, 6, 12, 13, 4, 1, 9, 15, 0, 11, 14, 2, 8, 10, 5\}$
- calls the bubbleSort method and the printArray() method
- calls the quickSort method and the printArray() method
- calls the mergeSort method and the printArray() method
- calls the radixSort method and the printArray() method
- calls the performanceAnalysis() method

At the end, write a basic Makefile which compiles all of your code and creates an executable file named `hw1`. Check out these tutorials for writing a simple make file: tutorial 1, tutorial 2. Please make sure that your `Makefile` works properly, otherwise you will not get any points from Question 2.

Important note: Add the screenshot of the console output of Question 2 to the pdf submission.

Question 3 – 15 points

After running your programs, you are expected to prepare a single page report about the experimental results that you obtained in Question 2 c. With the help of a spreadsheet program (Microsoft Excel, Matlab or other tools), plot *elapsed time* versus *the size of array* for each sorting algorithm implemented in question 2. Combine the outputs of each sorting algorithm in a single graph. A sample figure is given in Figure 1 (*these values do not reflect real values*).

In your report, you need to discuss the following points:

- Interpret and compare your empirical results with the theoretical ones. Explain any differences between the empirical and theoretical results, if any.
- How would the time complexity of your program change if you applied the sorting algorithms to an array of decreasing numbers instead of randomly generated numbers?

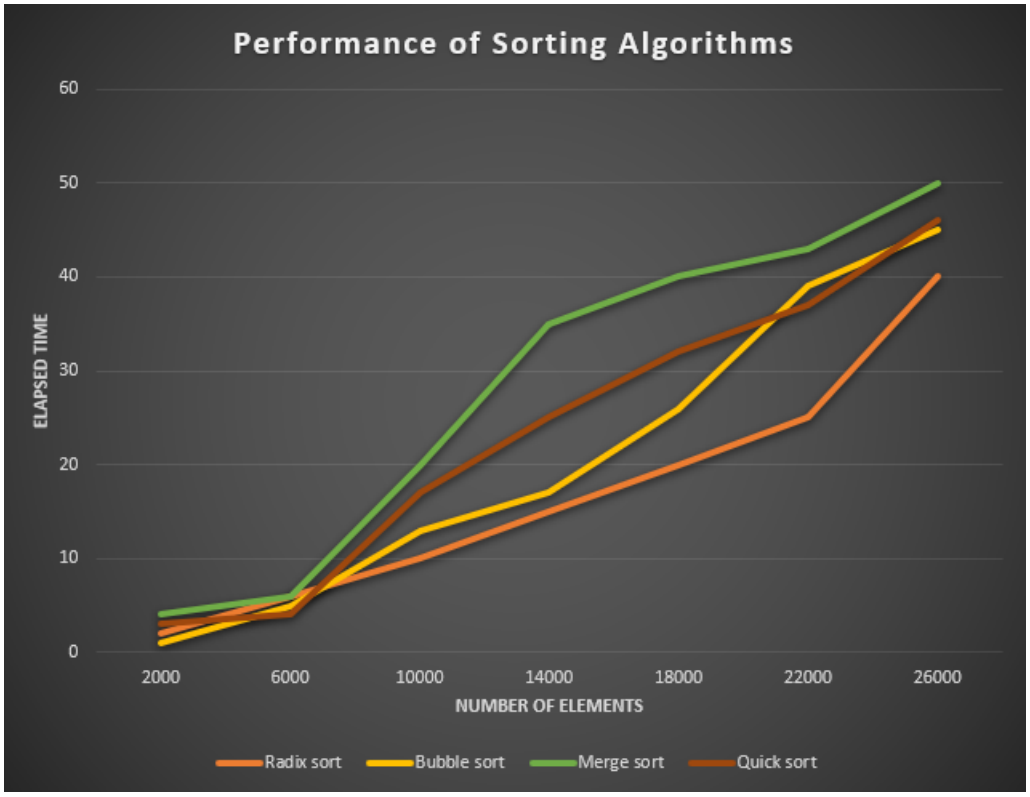

Figure 1: Sample figure for Sorting Performance Analysis