CS 202, Fall 2021

Homework #1 – Algorithm Efficiency and Sorting

Due Date: October 26, 2021

Important Notes

Please do not start the assignment before reading these notes.

- Before 23:55, October 26, 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 contains 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).

- 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 Fatma Kahveci. Contact her or your TA Nogay Evirgen directly for any homework related questions.

Attention: For this assignment, you are allowed to use the codes given in our text- book 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

- [*10 points*] Find the asymptotic running times in big O notation of the following recurrence equations by using the repeated substitution method. Show your steps in detail.
    - $T(n) = 3T(n/3) + n$, where T(1) = 1 and n is an exact power of 3.
    - $T(n) = 2T(n-1) + n^2$, where T(1) = 1.
    - $T(n) = 3T(n/4) + n.logn$, where T(1) = 1 and n is an exact power of 4.
    - $T(n) = 3T(n/2) + 1$, where T(1) = 1 and n is an exact power of 2.
- [*10 points*] Trace the following sorting algorithms to sort the array [5, 6, 8, 4, 10, 2, 9, 1, 3, 7] in ascending order. Use the array implementation of the algorithms as described in the slides and show all major steps.
    - Bubble Sort
    - Selection Sort
- [*5 points*] Write the recurrence relation of quick sort algorithm for the worst case, and solve it. Show all the steps clearly.

Question 2 – 60 points

Implement the following methods in the sorting.cpp file:

(a) [*40 points*] Implement the insertion sort, merge sort, quick sort and radix sort algorithms. Your functions should take an array of integers and the size of that array or the first and last index of the 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. Your functions should have the following prototypes:

```
void insertionSort(int *arr, const int size, int &compCount, int &moveCount);
void mergeSort(int *arr, const int size, int &compCount, int &moveCount);
void quickSort(int *arr, const int size, int &compCount, int &moveCount);
void radixSort(int *arr, const 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) [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 creates four identical arrays of 2000 random integers. Use one of the arrays for the insertion sort, second for merge sort, third for the quick sort, and the last one for the radix sort algorithm. Output the elapsed time (in milliseconds), the number of key comparisons and the number of data moves. Number of key comparisons and data moves are not required for the radix sort. Also, implement a function that prints the elements in an array.

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

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 a – Time Analysis of Insertion Sort
Array Size          Time Elapsed        compCount       moveCount
2000                x ms                x               x
4000                x ms                x               x
...
-------------------------------------------------
Part b – Time Analysis of Merge Sort
Array Size          Time Elapsed        compCount       moveCount
2000                x ms                x               x
4000                x ms                x               x
...
-------------------------------------------------
Part c – Time Analysis of Quick Sort
Array Size          Time Elapsed        compCount       moveCount
2000                x ms                x               x
4000                x ms                x               x
...
-------------------------------------------------
```

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

- Creates 4 identical arrays of the following numbers: {7,3,6,12,13,4,1,9,15,0,11,14,2,8,10,5}
- Calls the insertionSort method and the printArray() method
- Calls the mergeSort method and the printArray() method
- Calls the quickSort 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 this tutorial for writing a simple makefile:

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Please make sure that your Makefile works properly, otherwise you will not get any points from Question 2.

Important: Run your executable and add the screenshot of the console to the solution of Question 2 in the pdf submission. Also, while running radix sort for big array sizes you might run out of memory which will cause your program to fail. If that happens, you can decrease the array sizes for radix sort.

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 2c. With the help of a spreadsheet program (Microsoft Excel, Matlab, Python 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.

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 increasing numbers instead of randomly generated numbers?