CS 202, Fall 2019
Homework #1 – Algorithm Efficiency and Sorting
Due Date: October 21, 2019

Important Notes

Please do not start the assignment before reading these notes.

- Before 23:55, October 21, 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) [*10 points*] Sort the functions below in the increasing order of their asymptotic complexity: $f_1(n) = 10^n$, $f_2(n) = n^{log(logn)}$, $f_3(n) = n!$, $f_4(n) = logn$, $f_5(n) = n^{1/logn}$, $f_6(n) = nlogn$, $f_7(n) = e^n$, $f_8(n) = n^3$, $f_9(n) = log(n!)$, $f_{10}(n) = \sqrt{n}$

(b) [*5 points*] Determine the average processing time T(n) of the following algorithm assuming that the random function takes one time unit to return a random value between 0 and n.

```
int test(int n) {
    if (n<=0)
        return 0;
    else {
        int i = random(n);
        return (test(i) + test(n-1-i)); }    }
```

(c) [*10 points*] Trace the following sorting algorithms to sort the array [607, 1896, 1165, 2217, 675, 2492, 2706, 894, 743, 568] in ascending order. Use the array implementation of the algorithms as described in the textbook and show all major steps.

- Bubble sort
- Radix sort

Question 2 – 60 points

Implement the following methods in the sorting.cpp file:

(a) [*40 points*] Implement the quick sort, insertion sort and hybrid 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 quickSort(int *arr, int f ,int l, int &compCount, int &moveCount);
void insertionSort(int *arr, int size, int &compCount, int &moveCount);
void hybridSort(int *arr, int size, int &compCount, int &moveCount);
```

The hybrid sort algorithm starts with the quick sort (take the first element of the array as pivot), but when the partition size becomes less than or equal to 10, it sorts that partition with the insertion sort. Please see a sample implementation at the end of this document.

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 by writing a function named `performanceAnalysis`. This function will create **three identical arrays** of 1500 random integers. Use the first array for the quick sort, second for the insertion sort and the last one for the hybrid sort algorithm.

Listing 2: Sample output

```
-----------------------------------------------------
Part a - Time analysis of Quick Sort
Array Size      Time Elapsed    compCount   moveCount
1500                x ms            x           x
3000                x ms            x           x
...
-----------------------------------------------------
Part b - Time analysis of Insertion Sort
Array Size      Time Elapsed    compCount   moveCount
1500                x ms            x           x
3000                x ms            x           x
...
-----------------------------------------------------
Part c - Time analysis of Hybrid Sort
Array Size      Time Elapsed    compCount   moveCount
1500                x ms            x           x
3000                x ms            x           x
...
-----------------------------------------------------
```

Output the elapsed time (in milliseconds), the number of key comparisons and the number of data moves. Repeat the experiment for the following sizes: 3000, 4500, 6000, 7500, 9000, 10500, 12000, 13500, 15000. When the `performanceAnalysis` function is called, it needs to produce an output similar to Listing 2.

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

- Creates 3 identical arrays of the following numbers: $\{22, 11, 6, 7, 30, 2, 27, 24, 9, 1, 20, 17\}$
- On the 1st array, calls the quickSort method and the printArray() method
- On the 2nd array, calls the insertionSort method and the printArray() method
- On the 3rd array, calls the hybridSort 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 *the size of array* versus *elapsed time* 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 2 (*these values do not reflect real values*).

In your report, you need to interpret and compare your empirical results with the theoretical ones. How does combining two different sorting algorithms affect the efficiency of the sorting?

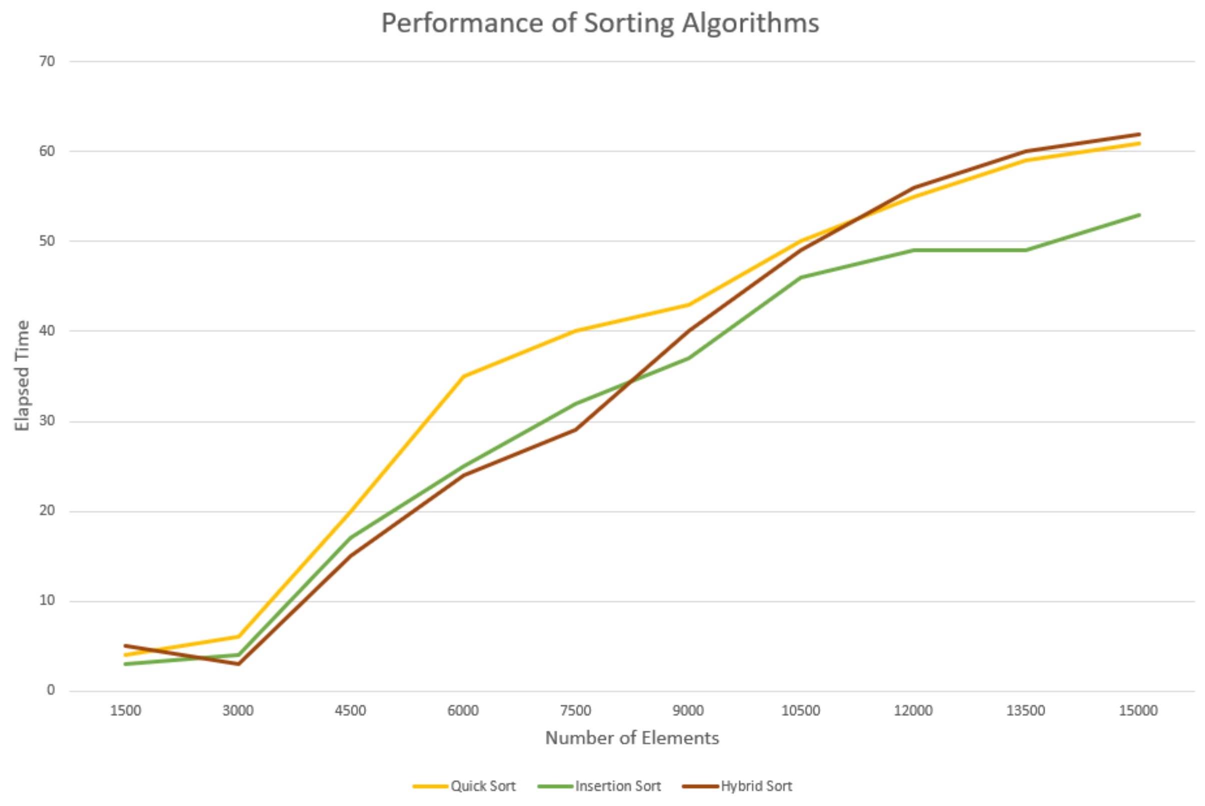
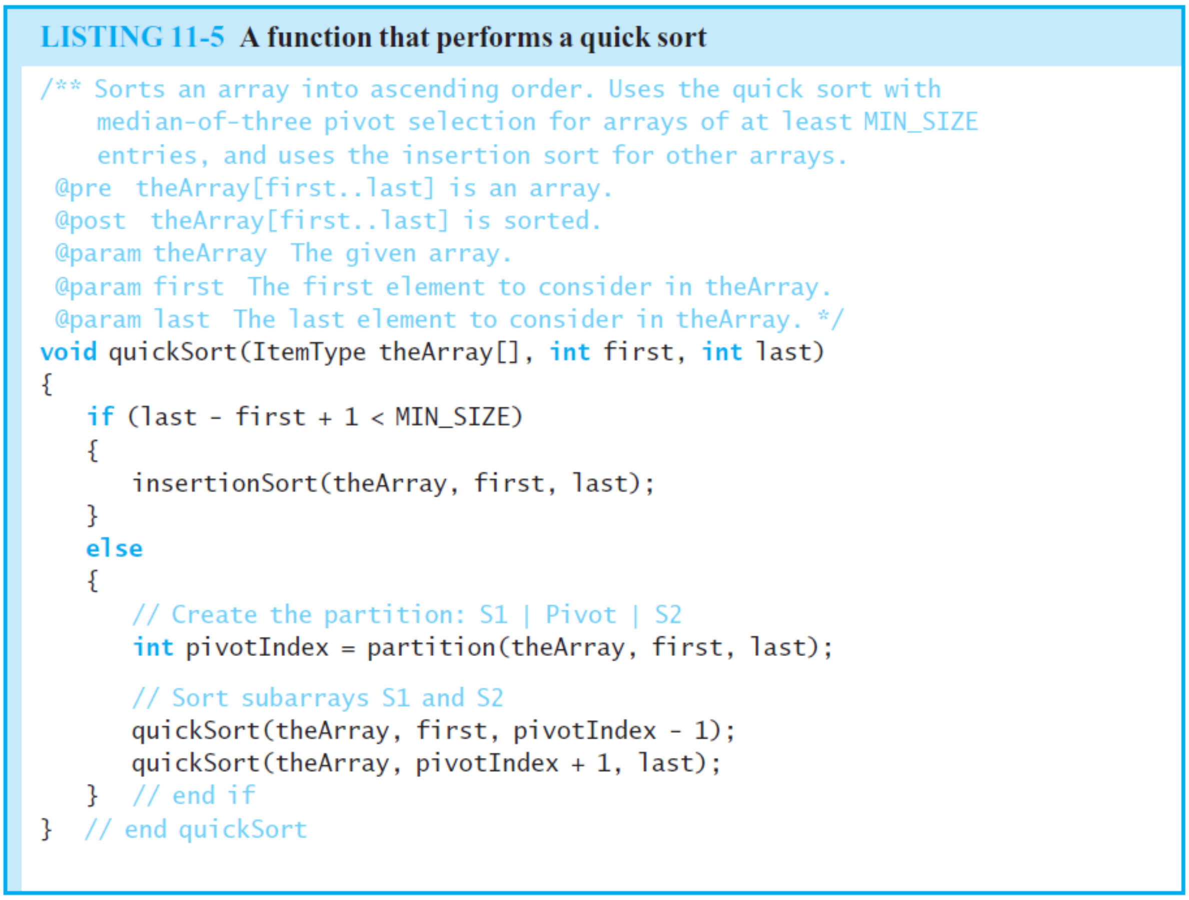


Figure 1: Sample figure for Sorting Performance Analysis



LISTING 11-5 A function that performs a quick sort

```
/** Sorts an array into ascending order. Uses the quick sort with
    median-of-three pivot selection for arrays of at least MIN_SIZE
    entries, and uses the insertion sort for other arrays.
 @pre  theArray[first..last] is an array.
 @post  theArray[first..last] is sorted.
 @param theArray  The given array.
 @param first  The first element to consider in theArray.
 @param last  The last element to consider in theArray. */
void quickSort(ItemType theArray[], int first, int last)
{
   if (last - first + 1 < MIN_SIZE)
   {
      insertionSort(theArray, first, last);
   }
   else
   {
      // Create the partition: S1 | Pivot | S2
      int pivotIndex = partition(theArray, first, last);

      // Sort subarrays S1 and S2
      quickSort(theArray, first, pivotIndex - 1);
      quickSort(theArray, pivotIndex + 1, last);
   }  // end if
}  // end quickSort
```

Figure 2: Sample code for Hybrid Sort (Carrano 6th edition, pg. 324)