Design and Analysis of Algorithms
Programming Assignment # 1
Due Date: 27 March 2017

Introduction:

Sorting routines are among the most widely used and studied algorithms. Every student should know how to implement several different kinds of sorts, and should have idea about how they perform, both theoretically and practically. This programming project is designed to give the student practice in implementing and observing the behavior of four sorts: Insertion Sort, Merge Sort, Heap Sort, and Quick Sort.

Resources:

The algorithms for Insertion Sort and Merge Sort are given in Chapter 2 of your textbook; the algorithm for Heap Sort is given in Chapter 6; and the algorithm for Quick Sort is given in Chapter 7.

Programs must be written in standard C, C++, or Java.

On the class web page you will find links to 12 data files for this project. These files all contain shuffled lists of integers, with one integer listed per line. The files are:

Filename	# items	lowest	highest	Description
Num8.txt	2^3	1	8	no omissions, no duplicates
Num16.txt	2^4	1	16	no omissions, no duplicates
Num32.txt	2^5	1	32	no omissions, no duplicates
Num64.txt	2^6	1	64	no omissions, no duplicates
Num128.txt	2^7	1	128	omissions/duplicates possible
Num256.txt	2^8	1	256	omissions/duplicates possible
Num512.txt	2^9	1	512	omissions/duplicates possible
Num1024.txt	2^{10}	1	1024	omissions/duplicates possible
Num2048.txt	2^{11}	1	2048	omissions/duplicates possible
Num4096.txt	2^{12}	1	4096	omissions/duplicates possible
Num8192.txt	2^{13}	1	8192	omissions/duplicates possible
Num16284.txt	2^{14}	1	16384	omissions/duplicates possible

Description:

You will write C, C++, or Java code that implements the textbook algorithms for the three sorting routines mentioned above. As part of your code, you will include counters that iterate whenever a specific line of the algorithm is executed.

Some lines in an algorithm may have a higher *cost* than other lines. For example, the function call in line 5 in the Merge Sort algorithm is executed only 7 times for an array with 8 elements, but the body of the Merge function which is being called has many lines, some of which are executed more than once. So the cost of line 5 in the Merge sort algorithm is higher than the other 4 lines. We can use the cost of the highest-cost line as an *indicator* of the cost of the algorithm as a whole.

Insertion Sort:

Here is the pseudocode for Insertion Sort, modified to include a counter:

```
count ← 0
Insertion_Sort (A)
1     for j ← 2 to length(A) do
2           key ← A[j]
3           // Insert A[j] into the sorted sequence A[1..j - 1]
4           i ← j – 1
5           while i > 0 and A[i] > key do
5.5               count ← count + 1
6                 A[i + 1] ← A[i]
7                 i ← i – 1
8           A[i + 1] ← key
```

Your code for Insertion Sort should have a line in it that is equivalent to line 5.5 in the Insertion_Sort pseudocode above. The global variable *count* will keep a running total of the number of times this line is executed. When you exit from the call to the Insertion Sort function, you should print out the values of *n* (the length of the array) and *count* as an indicator of the cost of the function.

Merge Sort:

Here is the pseudocode for Merge Sort, modified to include a counter:

```
count ← 0
Merge_Sort(A, p, r)
1     if p < r
2           then  q ← ⌊(p + r)/2⌋
3                 Merge-Sort (A, p, q)
4                 Merge-Sort (A, q+1, r)
5                 Merge (A, p, q, r)
```

And here is the modified algorithm for the Merge function used by Merge Sort:

```
Merge (A, p, q, r)
1     n1 ← (q – p) + 1
2     n2 ← (r – q)
3     create arrays L[1..n1+1] and R[1..n2+1]
4     for i ← 1 to n1 do
5           L[i] ← A[(p + i) -1]
6     for j ← 1 to n2 do
7           R[j] ← A[q + j]
8     L[n1 + 1] ← ∞
9     R[n2 + 1] ← ∞
10    i ← 1
11    j ← 1
12    for k ← p to r do
12.5        count ← count + 1
13          if L[I] <= R[j]
14                then  A[k] ← L[i]
15                      i ←i + 1
16                else A[k] ← R[j]
17                      j ← j + 1
```

Your code for Merge Sort should have a line of code in it that is equivalent to line 12.5 in the Merge pseudocode above. The global variable *count* will keep a running total of the number of times this line is executed. When you exit from the call to the Insertion Sort function, you should print out the values of *n* (the length of the array) and *count* as an indicator of the cost of the function.

Heap Sort:

Here is the pseudocode for Heap Sort, modified to include a counter:

```
count ← 0
HeapSort(A)
1     Build_Max_Heap(A)
2     for i ← length[A] downto 2 do
3           exchange A[1] ↔ A[i]
4           heap-size[A] ← heap-size[A] - 1
5           Max_Heapify(A, 1)
```

And here is the algorithm for the Max_Heapify function used by Heap Sort:

```
Max_Heapify(A, i)
1     l ← LEFT(i)
2     r ← RIGHT(i)
3     if l ≤ heap-size[A] and A[l] > A[i]
4           then largest ← l
5           else largest ← i
6     if r ≤ heap-size[A] and A[r] > A[largest]
7           then largest ← r
8     if largest ≠ i
9           then  exchange A[i] ↔ A[largest]
9.5               count ← count + 1
10                Max_Heapify (A, largest)
```

And here is the algorithm for the Build_Max_Heap function used by Heap Sort:

```
Build_Max_Heap(A)
1     heap-size[A] ← length[A]
2     for i ← floor(length[A]/2) downto 1 do
3           Max_Heapify (A, i)
```

Your program for Heap Sort should have a line of code in it that is equivalent to line 10 in the Max_Heapify pseudocode above. In your program, a global counter should keep track of the number of times this line is executed.

Quick Sort:

Here is the pseudocode for Quick Sort, modified to include a counter

```
QuickSort(A)
1 Count ← 0
2 QUICKSORT(A,1, length[A])
```

Here is the pseudocode for the QUICKSORT function used by Quick Sort:

```
QUICKSORT(A,p,r)
1 if p < r
2     then q ← PARTITION(A,p,r)
3     QUICKSORT(A,p,q-1)
```

```
4     QUICKSORT(A,q+1,r)
```

Here is the pseudocode for the PARTITION function used by QUICKSORT():

```
PARTITION(A,p,r)
1 x ← A[r]
2 i ← p - 1
3 for j ← p to r-1
3.5   count ← count + 1
4     do if A[j] ≤ x
5           then i ← i + 1
6                 exchange A[i] ↔ A[j]
7 exchange A[i+1] ↔ A[r]
8 return i+1
```

Program Execution:

Your program should read in data from Num8.txt, run Insertion Sort on it, and store its results; read in data from Num16.txt, run Insertion Sort on it, and store its results, etc., up through file Num16284.

Next it should repeat the process, using Merge Sort, Heap Sort, and Quick Sort as the sorting routines.

When your program terminates, you should have 36 sets of results. Each set of results should contain:

(1) the value of *count* immediately prior the termination of the algorithm,
(2) the array after having been sorted by your sort routine

Deliverables:

1. Email your instructor your program files.

By the due date, submit your source code as well as an instruction how to compile and run your code to the instructor via email at changhe.yuan@qc.cuny.edu.

If you are submitting multiple files, compress into and send a single zip file. Send only files which are readable by ordinary text editors.

Include in the body of your main email message instructions on how to compile and run your code.

2. Submit a printed lab report.

All lab reports are due in class on the due date. This lab report should consist of:

a) Title page: include Class name/number, assignment number, your name, due date, and actual date you are turning in the lab report

b) Printouts:

1) Source code: include source code files, header files, and make files, if any. The code of the printed lab report must be identical to the code supplied by email.

2) Test case output: include a print out of the results of running your program with your test cases. Include the *whole* sorted file for files Num8.txt, Num16.txt, Num32.txt, and Num64.txt. For each of the other files include a printout of the sorted file consisting of array items with index values of 51 through 100 *only*. Print the specified output for each of the 36 files onto a separate sheet, for a total of 36 sheets of paper. (You probably will have to use 8 point type.) Each sheet should have a header at the top explaining what it is (you can write this by hand), and then the list of array items.

3) Test case summary: The test case summary should be presented in the form of a table. Each entry in the table should be the value of *count* for one of the sorts after sorting one of the data sets. The table should have the following form:

Test Case Summary

	Insertion sort	*Merge sort*	*Heapsort*	*Quicksort*
Num8.txt				
Num16.txt				
Num32.txt				
Num64.txt				
Num128.txt				
Num256.txt				
Num512.txt				
Num1024.txt				
Num2048.txt				
Num4096.txt				
Num8192.txt				
Num16284.txt				

4) Analysis: Analyze and discuss the results of your experiment. You may want to plot your results using Excel or a graphing program to help you visualize the results better. At the very least answer the following questions:

a) Discuss what your results mean regarding the theoretical run-time of the different algorithms.
b) Do the sorts really take $O(n^2)$ and O(n lg n) steps to run?
c) Explain how you got your answer to this question.
d) Which of the sorts takes the most steps?
e) Which of the two O(n lg n) sorts takes the most steps?
f) Why?
g) Under what circumstances might you prefer to use one of the sorts versus others?
h) In general, which sort seems preferable?
i) Why?

Academic Honesty:
Please do your own work on this assignment. No collaboration in coding or writing the laboratory report is allowed. Downloading code from the web is also NOT permitted for this assignment.