PART ONE

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 an 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 a zip file called NumFiles-PARTONE.zip containing 12 data files for this part of the project. These files all contain shuffled lists of integers, with one integer listed per line. The files are:

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

Description:

You will write C, C++, or Java code that implements the textbook algorithms for the 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:

```plaintext
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:

```plaintext
count ← 0
Merge_Sort(A, p, r)
1 if p < r
2   then q ← \left\lfloor (p + r)/2 \right\rfloor
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:

```plaintext
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]
```
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 \textit{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 \textit{n} (the length of the array) and \textit{count} as an indicator of the cost of the function.

\textbf{Heap Sort:}

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

\begin{verbatim}
count ← 0
HeapSort(A)
1       Build_Max_Heap(A)
2       for i ← length[A] downto 2 do
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)
4           then largest ← l
5           else largest ← i
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)
\end{verbatim}

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.

\textbf{Quick Sort:}

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

\begin{verbatim}
QuickSort(A)
1       Count ← 0
2       QUICKSORT(A, 1, length[A])
\end{verbatim}
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
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 48 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 lab report and 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.

2. Submit a printed lab report before class of the due date.

**About Lab report:**

All lab reports are due both electronically and 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 the 48 files (you can layout the results into fewer pages as long as you make separations clear). Each section should have a header at the top explaining what it is 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:

<table>
<thead>
<tr>
<th>Test Case Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Num8.txt</td>
</tr>
<tr>
<td>Num16.txt</td>
</tr>
<tr>
<td>Num32.txt</td>
</tr>
<tr>
<td>Num64.txt</td>
</tr>
<tr>
<td>Num128.txt</td>
</tr>
<tr>
<td>Num256.txt</td>
</tr>
<tr>
<td>Num512.txt</td>
</tr>
<tr>
<td>Num1024.txt</td>
</tr>
<tr>
<td>Num2048.txt</td>
</tr>
<tr>
<td>Num4096.txt</td>
</tr>
<tr>
<td>Num8192.txt</td>
</tr>
<tr>
<td>Num16284.txt</td>
</tr>
</tbody>
</table>

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 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.
PART TWO

Introduction:

Sorting routines are among the most widely used and studied algorithms. Every CS student should know how to implement several different kinds of sorts, and should have an 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 three sorts: Counting sort, Radix sort, and Bucket sort.

Resources:

The algorithms for these three linear-time sorts are given in Chapter 8 of your textbook.

Your programs must be written in standard C, C++, or Java. On the class web page you will find a zip file called NumFiles-PARTTWO.zip containing 6 data files for this part of the project. These files all contain shuffled lists of integers, with one integer listed per line. The files are:

<table>
<thead>
<tr>
<th>Filename</th>
<th># items</th>
<th>lowest</th>
<th>highest</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num8.txt</td>
<td>2^3</td>
<td>1</td>
<td>8</td>
<td>no omissions, no duplicates</td>
</tr>
<tr>
<td>Num16.txt</td>
<td>2^4</td>
<td>1</td>
<td>16</td>
<td>no omissions, no duplicates</td>
</tr>
<tr>
<td>Num32.txt</td>
<td>2^5</td>
<td>1</td>
<td>32</td>
<td>no omissions, no duplicates</td>
</tr>
<tr>
<td>Num64.txt</td>
<td>2^6</td>
<td>1</td>
<td>64</td>
<td>no omissions, no duplicates</td>
</tr>
<tr>
<td>Num128.txt</td>
<td>2^7</td>
<td>1</td>
<td>128</td>
<td>omissions/duplicates possible</td>
</tr>
<tr>
<td>Num256.txt</td>
<td>2^8</td>
<td>1</td>
<td>256</td>
<td>omissions/duplicates possible</td>
</tr>
</tbody>
</table>

Description:

Write C, C++, or Java code that implements the textbook algorithms for the three sorting routines mentioned above. As part of your code, 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. We can use the cost of the highest-cost line as an indicator of the cost of the algorithm as a whole.

Counting Sort:

Here is the pseudocode for Counting Sort in your textbook:

```
COUNTING-SORT (A, B, k)
1  for i ← 0 to k do
2    C[i] ← 0
3  for j ← 1 to length[A] do
4    C[A[j]] ← C[A[j]] + 1
5  // C[i] now contains the # of elements = to i.
6  for i ← 1 to k do
7    C[i] ← C[i] + C[i-1]
8  // C[i] now contains the # of elements ≤ to i.
9  for j ← length[A] downto 1 do
10    B[C[A[j]]] ← A[j]
11    C[A[j]] ← C[A[j]] - 1
```
Your program for Counting Sort should have a line of code in it that is equivalent to line 10 in the Counting-Sort pseudocode above. In your program, a counter should keep track of the total number of times this line is executed.

Radix Sort:

Here is the pseudocode for Radix Sort in your textbook:

RADIX-SORT (A, d)
1 for i ← 1 to d do
2 use a stable sort to sort array A on digit i

For the “stable sort” in line 2 above, use your Counting-Sort program. (You may rewrite Counting-Sort slightly to meet the needs of this assignment, and you may incorporate it into the Radix-Sort code instead of calling it as a separate function.) We need to determine the cost of line 2 in the Radix-Sort pseudocode above. That line calls Counting-Sort. Your program for Counting Sort should have a line of code in it that is equivalent to line 10 in the Counting-Sort pseudocode above. In your Radix Sort program, a counter should keep track of the total number of times this line is executed while executing line 2 of the Radix-Sort algorithm.

Bucket Sort:

Here is the pseudocode for Bucket Sort in your textbook:

BUCKET-SORT(A)
1 n ← length[A]
2 for i ← 1 to n do
3 insert A[i] into list B[LnA[i]]
4 for i ← 0 to n-1 do
5 sort list B[i] with Insertion Sort
6 concatenate lists B[0], B[1], ..., B[n-1] together in order

(If you need help working with linked lists, Section 10.2 in your textbook gives the algorithms for basic operations on linked lists. However, you do not have to use linked lists; you may use an array of arrays if you wish. That is, you may have an array of n buckets, where each bucket is a short 1-D array.)

Here is the pseudocode for Insertion Sort in your textbook:

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
6 A[i + 1] ← A[i]
7 i ← i - 1
8 A[i + 1] ← key

Your program for Insertion Sort should have a line of code in it that is equivalent to line 6 in the Insertion-Sort pseudocode above. In your Bucket-Sort program, a counter should keep track of the total number of times this line is executed while executing line 5 of the Bucket-Sort algorithm.
The Bucket Sort is designed to work with real-valued numbers in the interval \((0, 1]\). To convert a set of numbers that are not in this range, you can use the following conversion algorithm, which should work for all numbers, including negative numbers:

\[
\begin{align*}
x &= -(element_{\min}) \\
&\text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
&\quad element[i] = element[i] + x \quad // \text{now the value of } element_{\min} = 0 \\
y &= element_{\max} + 1 \\
&\text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
&\quad element[i] = element[i] / y \quad // \text{now all values are in the range } (0, 1]
\end{align*}
\]

At this point you can use the Bucket-Sort algorithm given in your book. Once the sort terminates, you can reverse the above process to recover the original numbers. Due to the round-off error caused by trying to represent floating-point numbers in a finite number of bits, if your original numbers were integers you may need to round the floating-point numbers up or down to recover the integer values.

**Program Execution:**

Your program should read in data from Num8.txt, run Counting Sort on it, and store its results in a file; read in data from Num16.txt, run Counting Sort on it, and store its results in a file, etc., up through file Num256.

Next it should repeat the process, using Radix Sort as the sorting routine.

Finally it should repeat the process, using Bucket Sort as the sorting routine.

When your program terminates, you should have 18 files. Each file should contain:
1. the count for the specific line of the algorithm you were asked to keep track of,
2. the array after having been sorted by your sort routine

**Deliverables:**

1. Email your instructor your program files.
   By the *beginning of class* on the due date, submit your source code to the instructor via email at:changhe.yuan@qc.cuny.edu.
   If you are submitting multiple files, attach each separately. Do not compress (zip, tar, rar, jar, etc.) or encode your files. 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 at the *beginning of 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) First page: specify whether you are submitting UNIX code or code for a PC, as well as what compiler should be used. It should also include any instructions necessary to compile and execute your code.
   c) 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 18 files (you can layout the results into fewer pages as long as you make separations clear). Each section should have a header at the top explaining what it is 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 total number of times the specified line is executed by one of the sorts while sorting one of the data sets. The table should have the following form:

<table>
<thead>
<tr>
<th></th>
<th>Counting sort</th>
<th>Radix sort</th>
<th>Bucket sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num8.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num16.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num32.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num64.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num128.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num256.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

   j) Discuss what your results mean regarding the theoretical run-time of the different algorithms.
   k) Do the sorts really take O(n) steps to run?
   l) Explain how you got your answer to this question.
   m) Which of the sorts takes the most steps?
   n) Why?
   o) Under what circumstances might you prefer to use a particular one of the three linear sorts?
   p) Why?
   q) In general, which sort seems preferable?
   r) Why?
   s) When might you prefer to use Quicksort instead of one of the linear-time sorts?
   t) When might Insertion sort be the best sort of all to use?

**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.