Project Cover Page

This project is a group project. For each group member, please print first and last name and e-mail address.

1.

2.

3.

Please write how each member of the group participated in the project.

1.

2.

3.

Please list all sources: web pages, people, books or any printed material, which you used to prepare a report and implementation of algorithms for the project.

Type of sources:	
People	
Web Material (give URL)	
Printed Material	
Other Sources	

I certify that I have listed all the sources that I used to develop solutions to the submitted project report and code.
Your signature Typed Name Date

I certify that I have listed all the sources that I used to develop a solution to the submitted project and code.
Your signature Typed Name Date

I certify that I have listed all the sources that I used to develop solution to the submitted project and code.
Your signature Typed Name Date

CSCE 221 Programming Assignment 2 (200 points)

Programs due February 14th by 11:59pm
Reports due on the first lab of the week of 14th

- **Objective**

  In this assignment, you will implement five sorting algorithms: selection sort, insertion sort, bubble sort, shell sort and radix sort in C++. You will test your code using varied input cases, time the implementation of sorts, record number of comparisons performed in the sorts, and compare these computational results with the running time of implemented algorithms using Big-O asymptotic notation.

- **General Guidelines**

  1. This project can be done in groups of at most three students. Please use the cover sheet at the previous page for your hardcopy report.
  2. The supplementary program is packed in `221-A2-code.tar` which can be downloaded from the course website. You may untar the file using the following command on Unix or using 7-Zip software on Windows.

     ```
     tar xfv 221-A2-code.tar
     ```

  3. Make sure your code can be compiled using GNU C++ compiler before submission because your programs will be tested on a CSE Linux machine. Use `Makefile` provided with supplementary program by typing the following command on Linux

     ```
     make clean
     make
     ```

  4. When you run your program on the Linux server, use Ctrl+C to stop the program. Do NOT use Ctrl+Z, as it just suspends the program and does not kill it. We do not want to see the department server down because of this assignment.
  5. Supplementary reading
     (a) Lecture note: Introduction to Analysis of Algorithms
     (b) Lecture note: Sorting in Linear Time
  6. Submission guidelines
     (a) Electronic copy of all the code, the 15 types of input integer sequences, and reports in Lyx and PDF format.
     (b) Hardcopy report, the code of 5 sort functions, and the code that generates integer sequences.
  7. Your program will be tested on TA's input files.

- **Code**

  1. In this assignment, the sort program reads a sequence of integers either from the screen (standard input) or from a file, and outputs the sorted sequence to the screen (standard output) or to a file. The program can be configured to show total running time and/or total number of comparisons done in the sort.
  2. This program does not have a menu but takes arguments from the command line. The code for interface is completed in the template programs, so you only have to know how to execute the program using the command line.

The program usage is as follows. *Note that options do not need to be specified in a fixed order.*
Usage:

```
./sort [-a ALGORITHM] [-f INPUTFILE] [-o OUTPUTFILE] [-h] [-d] [-p] [-t] [-c]
```

Example:

```
./sort -h
./sort -a S -f input.txt -o output.txt -d -t -c -p
./sort -a I -t -c
./sort
```

Options:

```
-a ALGORITHM: Use ALGORITHM to sort.
   ALGORITHM is a single character representing an algorithm:
   S for selection sort
   B for bubble sort
   I for insertion sort
   H for shell sort
   R for radix sort
-f INPUTFILE: Obtain integers from INPUTFILE instead of STDIN
-o OUTPUTFILE: Place output data into OUTPUTFILE instead of STDOUT
-h: Display this help and exit
-d: Display input: unsorted integer sequence
-p: Display output: sorted integer sequence
-t: Display running time of the chosen algorithm in milliseconds
-c: Display number of comparisons (excluding radix sort)
```

3. **Format of the input data.** The first line of the input contains a number n which is the number of integers to sort. Subsequent n numbers are written one per line which are the numbers to sort. Here is an example of input data:

```
5 // this is the number of lines below = number of integers to sort
7
-8
4
0
-2
```

4. **Format of the output data.** The sorted integers are printed one per line in increasing order. Here is the output corresponding to the above input:

```
-8
-2
0
4
7
```

5. (*50 points*) Your tasks include implementing the following five sorting algorithms in corresponding cpp files.
   (a) selection sort in `selection-sort.cpp`
   (b) insertion sort in `insertion-sort.cpp`
   (c) bubble sort in `bubble-sort.cpp`
   (d) shell sort in `shell-sort.cpp`
   (e) radix sort in `radix-sort.cpp`
      i. Implement the radix sort algorithm that can sort 0 to $(2^{16}-1)$ but takes input -2^{15} to $(2^{15}-1)$.
      ii. About radix sort of negative numbers: "You can shift input to all positive numbers by adding a number which makes the smallest negative number zero. Apply radix sort and next make a reverse shift to get the initial input."

6. (*20 points*) Generate several sets of 10^2, 10^3, 10^4, and 10^5 integers in three different orders.
   (a) random order
   (b) increasing order
   (c) decreasing order

   HINT: The standard library `<cstdlib>` provides functions `srand()` and `rand()` to generate random numbers.

7. Measure the average number of comparisons (excluding radix sort) and average running times of each algorithms on the 15 integer sequences.
   (a) (*20 points*) Insert additional code into each sort (excluding radix sort) to count the number of ***comparisons performed on input integers***. The following tips should help you with determining how many comparisons are performed.
      i. You will measure 3 times for each algorithm on each sequence and take average
      ii. Insert the code that increases number of comparison `num_cmps++` typically in an `if` or a loop statement
      iii. Remember that C++ uses the shortcut rule for evaluating boolean expressions. A way to count comparisons accurately is to use comma expressions. For instance

      ```
      while (i < n && (num_cmps++, a[i] < b))
      ```

      HINT: If you modify `sort.cpp` and run several sorting algorithms subsequently, you have to call resetNumCmps() to reset number of comparisons between every two calls to `s->sort()`.
   (b) Modify the code in `sort.cpp` so that it repeatedly measures the running time of `s->sort()`.
      i. You will measure roughly 10^7 times for each algorithm on each sequence and take the average. You have to run for the same number of rounds for each algorithm on each sequence, and make sure that each result is not 0.
      ii. When you measure the running time of sorting algorithms, please reuse the input array but fill with different numbers. Do not allocate a new array every time, that will dramatically slower the program.
      iii. To time a certain part of the program, you may use functions `clock()` defined in header file `<ctime>`, or `gettimeofday()` defined in `<sys/time.h>`. Here are the examples of how to use these functions. The timing part is also completed in the template programs. However, you will apply these function to future assignments.

   The example using `clock()` in `<ctime>`:

   ```
   #include <ctime>
       ...
       clock_t t1, t2;
       t1 = clock(); // start timing
       ...
       /* operations you want to measure the running time */
       ...
       t2 = clock(); // end of timing
       double diff = (double)(t2 - t1)/CLOCKS_PER_SEC;
       cout << "The timing is " << diff << “ ms” << endl;
   ```

   The example using `gettimeofday()` in `<sys/time.h>`:

   ```
   #include <sys/time.h>
       ...
       struct timeval start, end;
       ...
       gettimeofday(&start,0); // start timing
       ...
       /* operations you want to measure the running time*/
       ...
       gettimeofday(&end,0); // end of timing
   ```

```
double diff = (end.tv_sec - start.tv_sec)
              + (double)(end.tv_usec - start.tv_usec)/1e6;
cout << "The timing is " << diff << " sec" << endl;
```

- **Report (110 points)**

  Write a report that includes all following elements in your report.

  1. (5 points) A brief description of assignment purpose, assignment description, how to run your programs, what to input and output.
  2. (5 points) Explanation of splitting the program into classes and *a description of C++ object oriented features or generic programming used in this assignment*.
  3. (5 points) **Algorithms.** Briefly describe the features of each of the five sorting algorithms.
  4. (20 points) **Theoretical Analysis.** Theoretically analyze the time complexity of the sorting algorithms with input integers in decreasing, random and increasing orders and fill the second table. Fill in the first table with the time complexity of the sorting algorithms when inputting the best case, average case and worst case. Some of the input orders are exactly the best case, average case and worst case of the sorting algorithms. State what input orders correspond to which cases. You should use big-O asymptotic notation when writing the time complexity (running time).

Complexity	best	average	worst
Selection Sort			
Insertion Sort			
Bubble Sort			
Shell Sort			
Radix Sort			

Complexity	inc	ran	dec
Selection Sort			
Insertion Sort			
Bubble Sort			
Shell Sort			
Radix Sort			

inc: increasing order; dec: decreasing order; ran: random order

  5. (65 points) **Experiments.**
     (a) Briefly describe the experiments. Present the experimental running times (**RT**) and number of comparisons (**#COMP**) performed on input data using the following tables.

RT	Selection Sort			Insertion Sort			Bubble Sort		
n	inc	ran	dec	inc	ran	dec	inc	ran	dec
100									
10^3									
10^4									
10^5									

RT	Shell Sort			Radix Sort		
n	inc	ran	dec	inc	ran	dec
100						
10^3						
10^4						
10^5						

#COMP	Selection Sort			Insertion Sort		
n	inc	ran	dec	inc	ran	dec
100						
10^3						
10^4						
10^5						

#COMP	Bubble Sort			Shell Sort		
n	inc	ran	dec	inc	ran	dec
100						
10^3						
10^4						
10^5						

inc: increasing order; dec: decreasing order; ran: random order

(b) For each of the five sort algorithms, graph the running times over the three input cases (inc, ran, dec) versus the input sizes (*n*); and for each of the first four algorithms graph the numbers of comparisons versus the input sizes, totaling in 9 graphs.
HINT: To get a better view of the plots, *use logarithmic scales* for both x and y axes.

(c) To compare performance of the sorting algorithms you need to have another 3 graphs to plot the results of all sorts for the running times for *each* of the input cases (inc, ran, dec) separately.
HINT: To get a better view of the plots, *use logarithmic scales* for both x and y axes.

6. (5 points) **Discussion.** Comment on how the experimental results relate to the theoretical analysis and explain any discrepancies you note. Is your computational results match the theoretical analysis you learned from class or textbook? Justify your answer. Also compare radix sort's running time with the running time of four comparison-based algorithms.

7. (5 points) **Conclusions.** Give your observations and conclusion. For instance, which sorting algorithm seems to perform better on which case? Do the experimental results agree with the theoretical analysis you learned from class or textbook? What factors can affect your experimental results?