Project Cover Page

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

1.Alexander Kaiser

2.Zantis Moye

3.

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

1. Wrote code and performed experiments.

2.Researched theoretical analysis and recorded graphing.

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	Data Structures and Algorithms in C++ M.T. Goodrich, R. Tamassia and D. Mount
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-AlexanderKaiser Date: 9/19/16

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-Zantis Moye Date: 9/19/16

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)

Program and Reports due September 25th by 11:59pm

- **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.
     Purpose
     The purpose of this assignment is to compare run time and total number of comparisons between five different sorting algorithms.
     Description
     Each of the different sorting algorithms will be tested with 12 separate test cases. These include sets of numbers that are in random order, increasing order and decreasing order with size 10^2, 10^3, 10^4, and 10^5.
     HowtoRun
     Compile the program using make in the directory Kaiser-Alexander-A2. Run the program using ./sort and the corresponding keywords to display the preferred output (Given in -h, the help menu).
     InputandOutput
     The program inputs a .txt file containing a vector. The first element of the vector contains the size of the vector and the rest of the elements correspond to whether the vector is to be increasing, decreasing or random. After the initial sorting of the vector, the contents of the vector are resorted into increasing, decreasing or random elements to be sorted again. The vector is sorted then unsorted 10000000 times and the average runtime, the number of comparisons and the sorted vector are output onto the console.

  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*.
     The program is heavily dependent upon the C++ inheritance feature. Each sorting algorithm is placed in its own subclass of the .cpp file sort. This class heirarchy allows for multiple computations to be made to the data without the user changing the member functions of the base class. On top of inheritance, the code also uses polymorphism in the sorting algorithms. A call to the member function will cause a different function to be executed depending upon the type of object that calls the function.

  3. (5 points) **Algorithms.** Briefly describe the features of each of the five sorting algorithms.
     Selection
     Selection sort operates by scanning the entire set of elements and holds the smallest one passed. This element is moved to the first spot and locked in after having scanned the entire set. This process is repeated for the entire set until the set is sorted from lowest to highest.
     Insertion
     Insertion sort operates by scanning each individual element of the set and comparing it with the elements beside it. If the element is less than the element to its left, the two elements are swapped. This process repeats until the set is ordered from smallest to largest.
     Bubble
     Bubble sort operates opposite to the Insertion sort. Each element is compared to the two elements beside it and if the element to the right is less than the element, it is moved up one location until it reaches an element that it is not greater than. This process repeats for all elements until the set is ordered from smallest to largest.
     Shell
     Shell sort operates a lot like insertion sort however one extra step is added. The set is divided into subgroups and two elements from each subgroup are compared. If the element on the left is greater than the element on the right the two elements are swapped. This continues until the subgroups have been scanned. The set is then divided into smaller subgroups and the task is performed once more. This process repeats until the subgroups become the size of each individual element and all elements are organized from smallest to largest.
     Radix
     Radix sort operates by dividing the integers into columns of the digits in the ones, tenths, hundredths,

thousandths, etc. place and compares the elements starting from the ones place. The set is organized from least to greatest integers in the ones place, then promptly moves on to the next column of digits and repeats the process. This process is repeated until all columns have been sorted, which in turn sorts the elements from least to greatest.

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	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell Sort	$O(nlogn)$	$O(n(logn)^2)$	$O(n(logn)^2)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$

Complexity	inc	ran	dec
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell Sort	$O(nlogn)$	$O(n(logn)^2)$	$O(n(logn)^2)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$

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.

   Each experiment was performed with the corresponding .txt file containing n values and the vector was reorganized after each sort using a for loop.

RT	Selection Sort			Insertion Sort			Bubble Sort		
n	inc	ran	dec	inc	ran	dec	inc	ran	dec
100	.045596 ms	.051159 ms	.047211 ms	.003462 ms	.028871 ms	.053005 ms	.003461 ms	.084693 ms	.096771 ms
10^3	4.098721 ms	4.646834 ms	4.111782 ms	.034292 ms	2.572432 ms	5.063783 ms	.035428 ms	9.107398 ms	9.572376 ms
10^4	417.643964 ms	449.287634 ms	438.844896 ms	.385432 ms	261.367324 ms	511.945376 ms	.352321 ms	905.739342 ms	956.387452 ms
10^5	41405.349876 ms	44019.756354 ms	41545.134378 ms	4.647635 ms	26033.283974 ms	50024.65489 ms	3.532421 ms	91025.328973 ms	95645.390745 ms

RT	Shell Sort			Radix Sort		
n	inc	ran	dec	inc	ran	dec
100	.003675 ms	.087104 ms	.130194 ms	.262089 ms	.261985 ms	.263436 ms
10^3	.036398 ms	9.582873 ms	9.634375 ms	2.584428 ms	2.591856 ms	2.585432 ms
10^4	.364587 ms	943.187365 ms	957.643871 ms	26.156735 ms	26.125693 ms	26.376598 ms
10^5	3.637893 ms	94335.437862 ms	97644.975638 ms	259.134874 ms	260.745873 ms	265.867580 ms

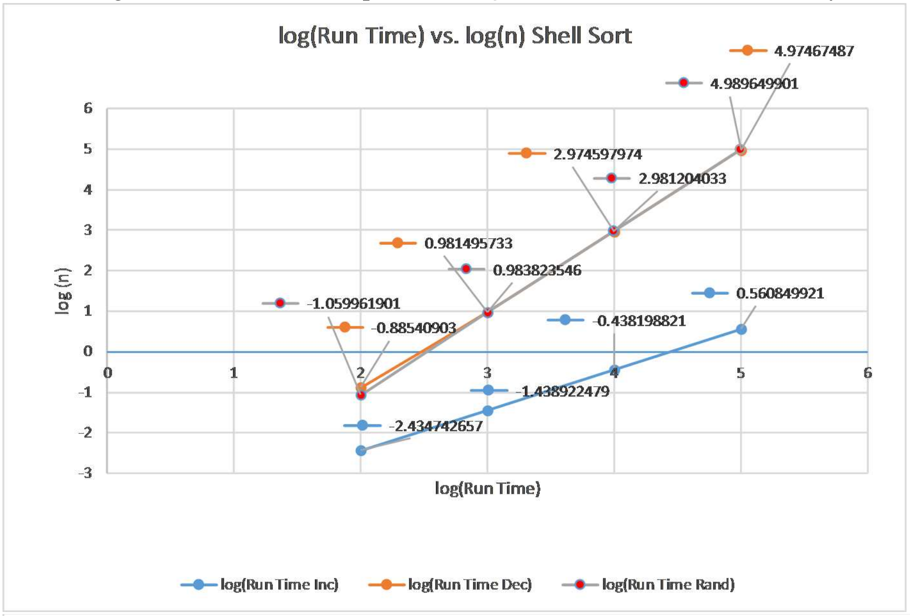
#COMP	Selection Sort			Insertion Sort		
n	inc	ran	dec	inc	ran	dec
100	4950	4950	4950	99	2464	4950
10^3	499500	499500	499500	999	259173	499500
10^4	49995000	49995000	49995000	9999	25114855	49995000
10^5	4999950000	4999950000	4999950000	99999	2491908607	4999950000

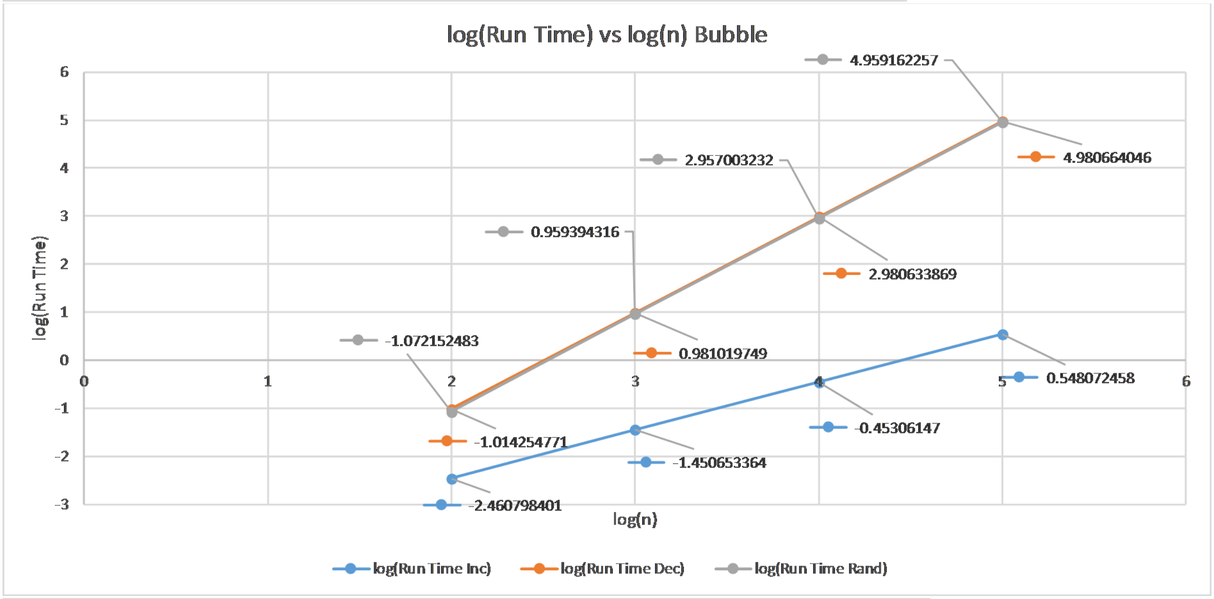
#COMP	Bubble Sort			Shell Sort		
n	inc	ran	dec	inc	ran	dec
100	197	4914	4950	197	4914	4950
10^3	1997	497355	499500	1997	497355	499500
10^4	19997	49986744	49995000	19997	49986744	49995000
10^5	199997	4999853420	4999950000	199997	4999853420	4999950000

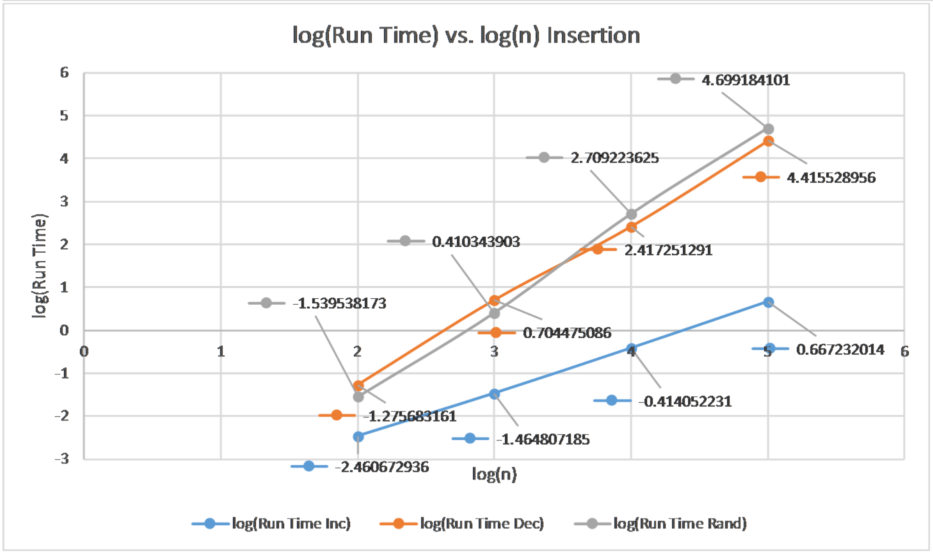
inc: increasing order; dec: decreasing order; ran: random order

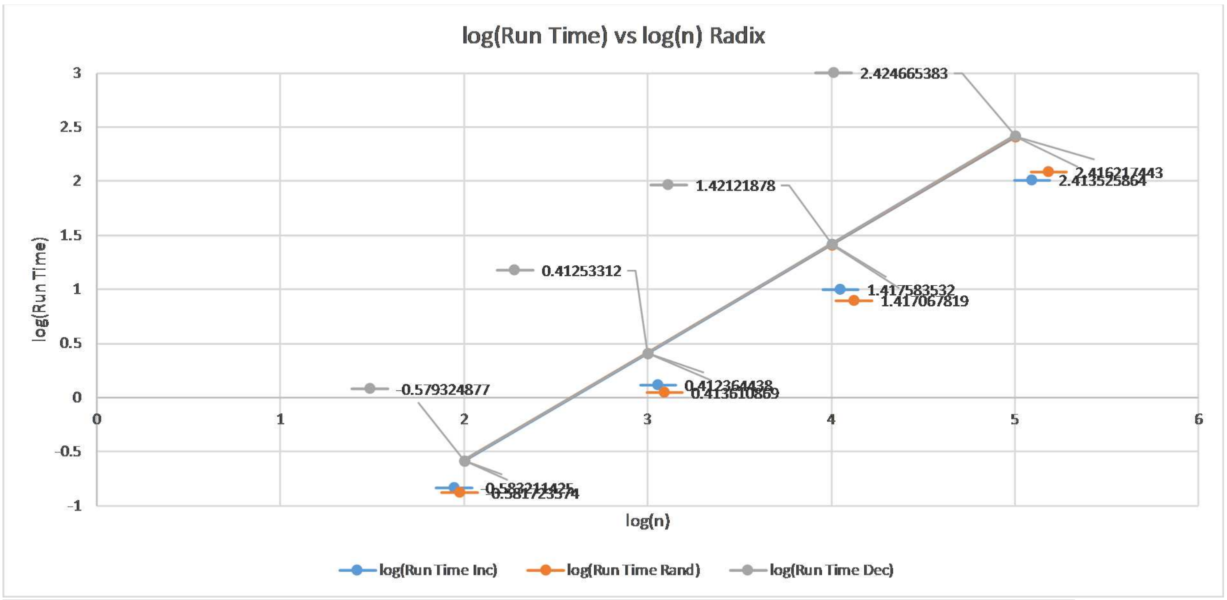
   (b) For each of the five sort algorithms, graph the running times over the three input cases (inc, ran, dec) versus the input sizes (n); and for each of the first four algorithms graph the numbers of comparisons versus the input sizes, totaling in 9 graphs.

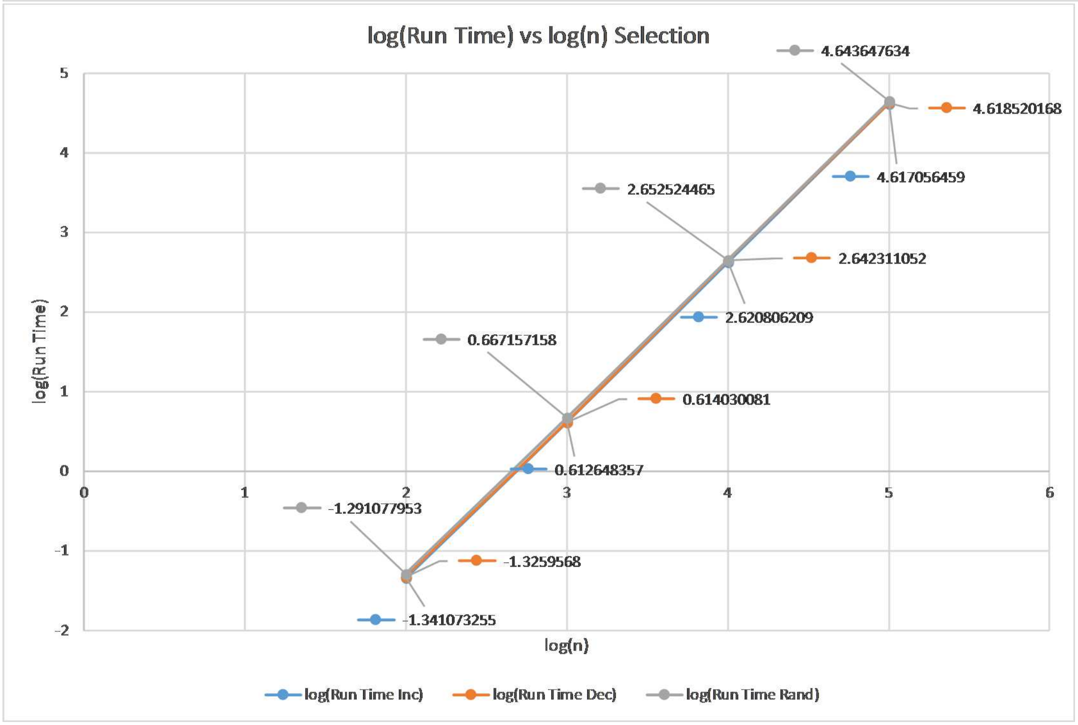
HINT: To get a better view of the plots, *use logarithmic scales* for both x and y axes.

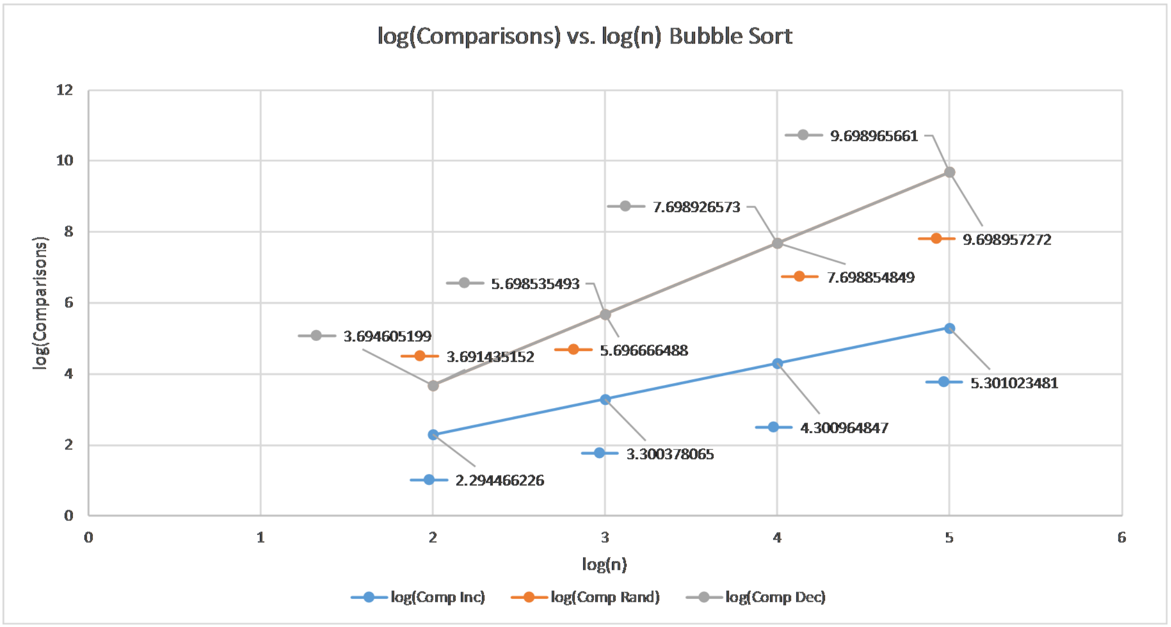


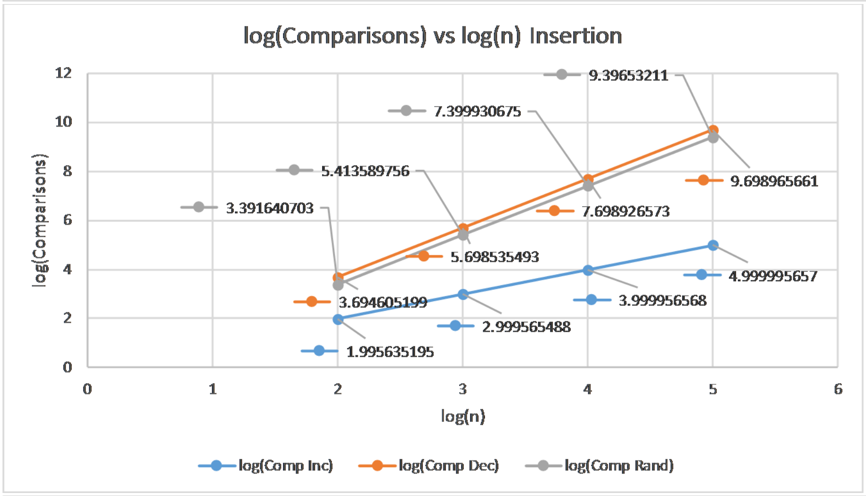


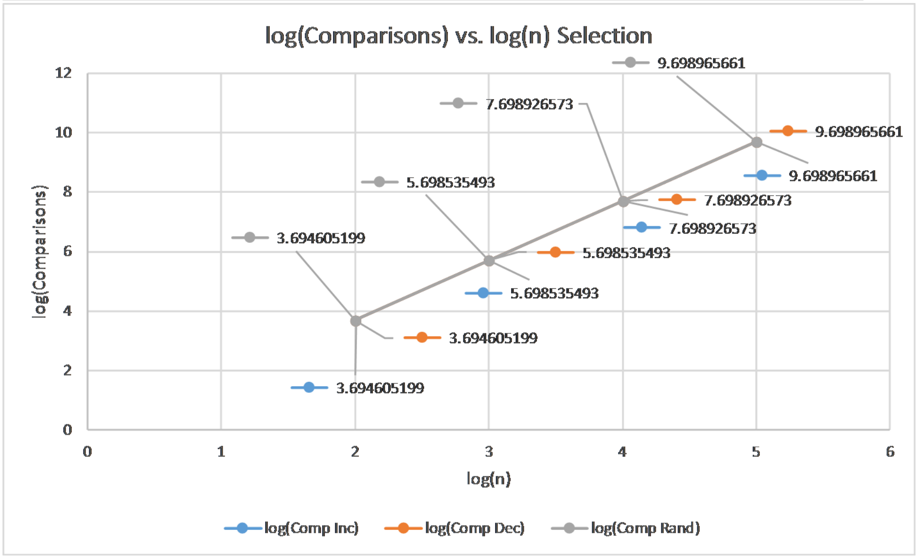


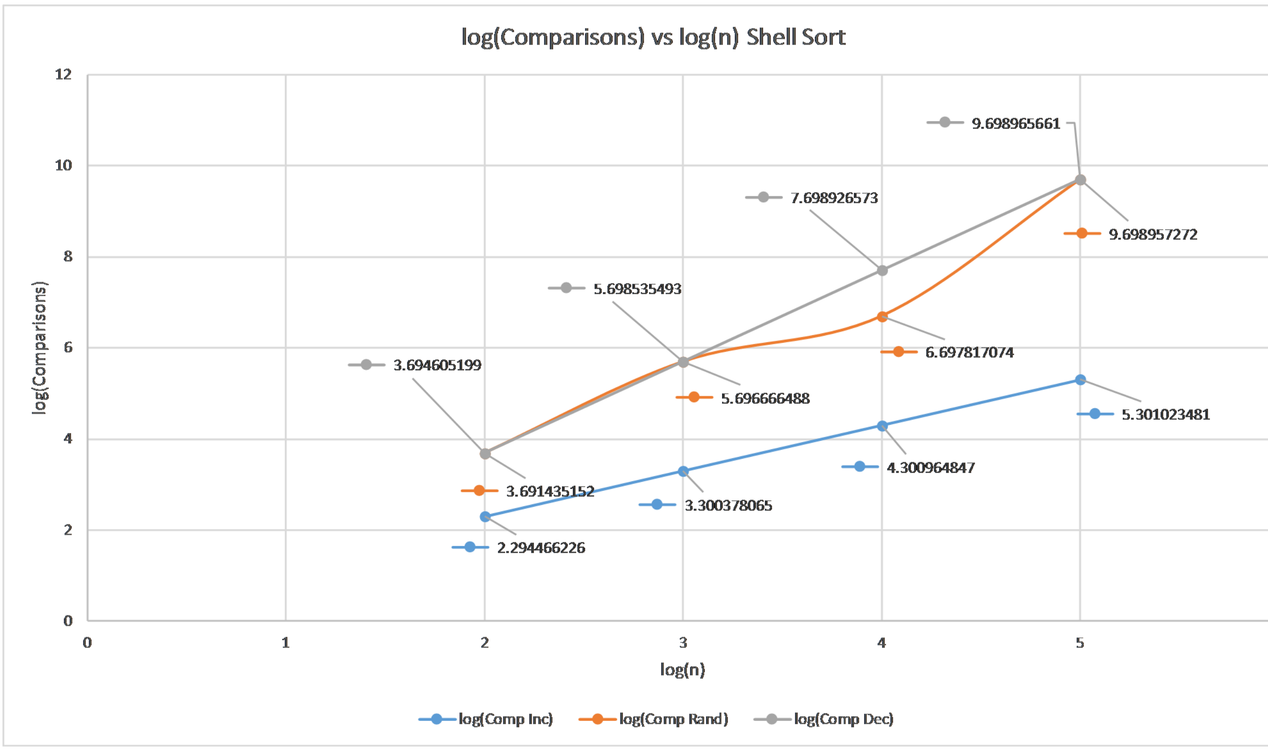

log(Run Time) vs log(n) Radix
2.424665383
2.416217443
2.413525864
1.42121878
1.417583532
1.417067819
0.41253312
0.412364438
0.413610869
-0.579324877
-0.583211425
-0.581723374
log(Run Time)
log(n)
3
2.5
2
1.5
1
0.5
0
-0.5
-1
0
1
2
3
4
5
6
log(Run Time Inc)
log(Run Time Rand)
log(Run Time Dec)


log(Run Time) vs log(n) Selection
4.643647634
4.618520168
4.617056459
2.652524465
2.642311052
2.620806209
0.667157158
0.614030081
0.612648357
-1.291077953
-1.3259568
-1.341073255
log(Run Time)
log(n)
5
4
3
2
1
0
-1
-2
0
1
2
3
4
5
6
log(Run Time Inc)
log(Run Time Dec)
log(Run Time Rand)

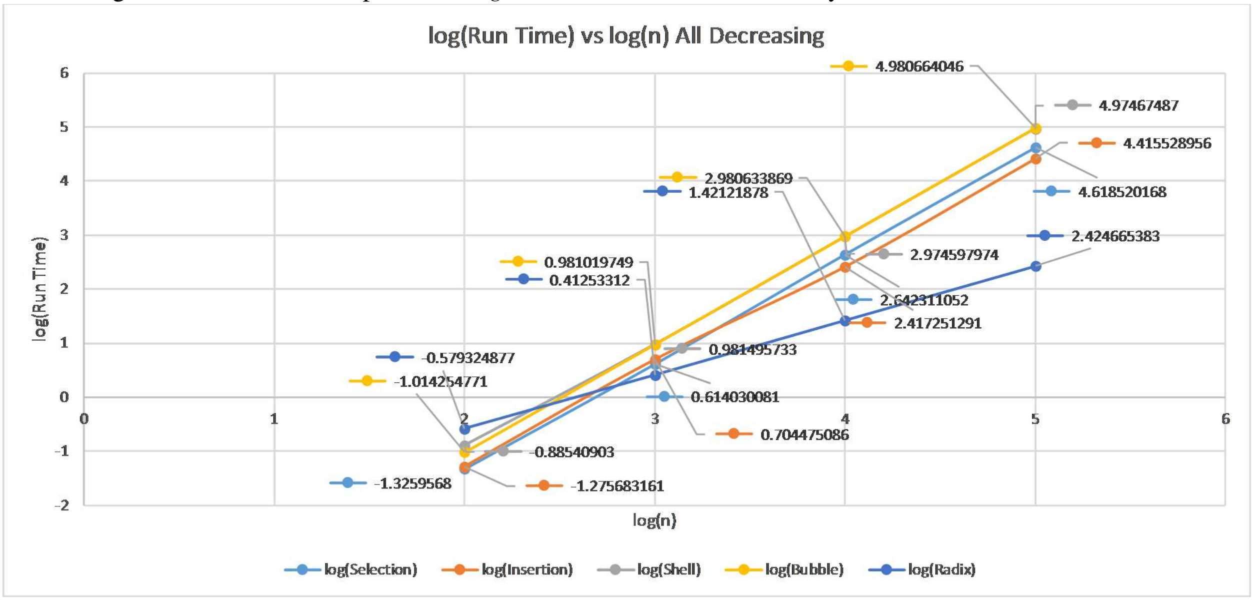

log(Comparisons) vs. log(n) Bubble Sort
log(Comparisons)
log(n)
0
2
4
6
8
10
12
0
1
2
3
4
5
6
9.698965661
7.698926573
9.698957272
7.698854849
5.698535493
5.696666488
3.694605199
3.691435152
5.301023481
4.300964847
3.300378065
2.294466226
log(Comp Inc)
log(Comp Rand)
log(Comp Dec)

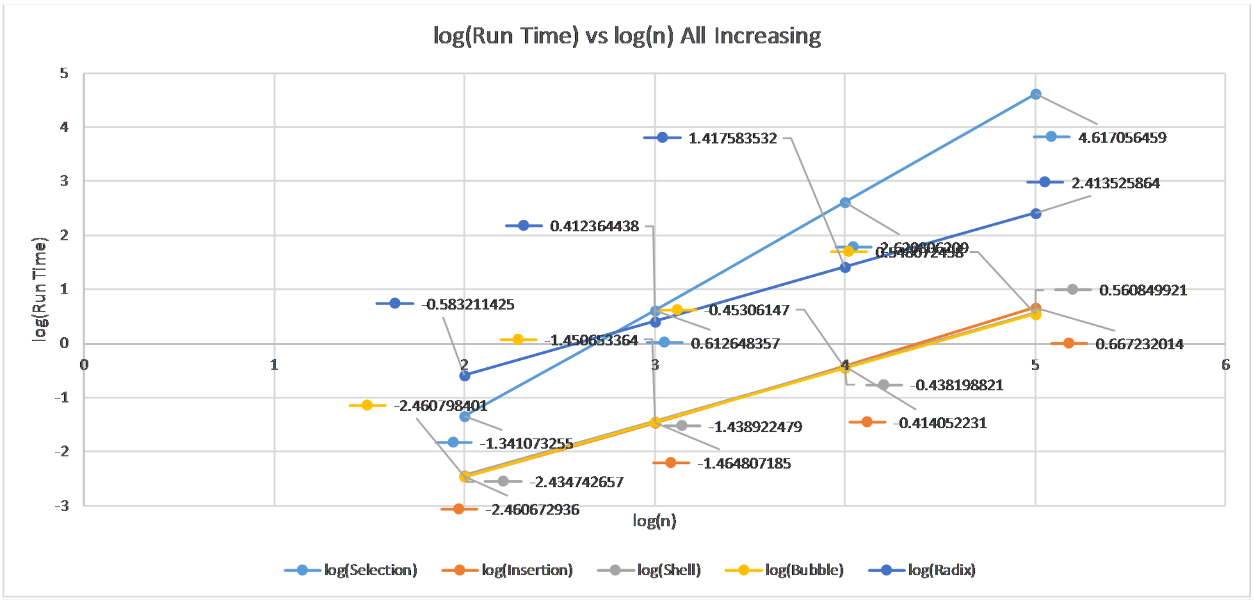

log(Comparisons) vs log(n) Insertion
log(Comparisons)
log(n)
0
2
4
6
8
10
12
0
1
2
3
4
5
6
9.39653211
7.399930675
5.413589756
3.391640703
9.698965661
7.698926573
5.698535493
3.694605199
4.999995657
3.999956568
2.999565488
1.995635195
log(Comp Inc)
log(Comp Dec)
log(Comp Rand)

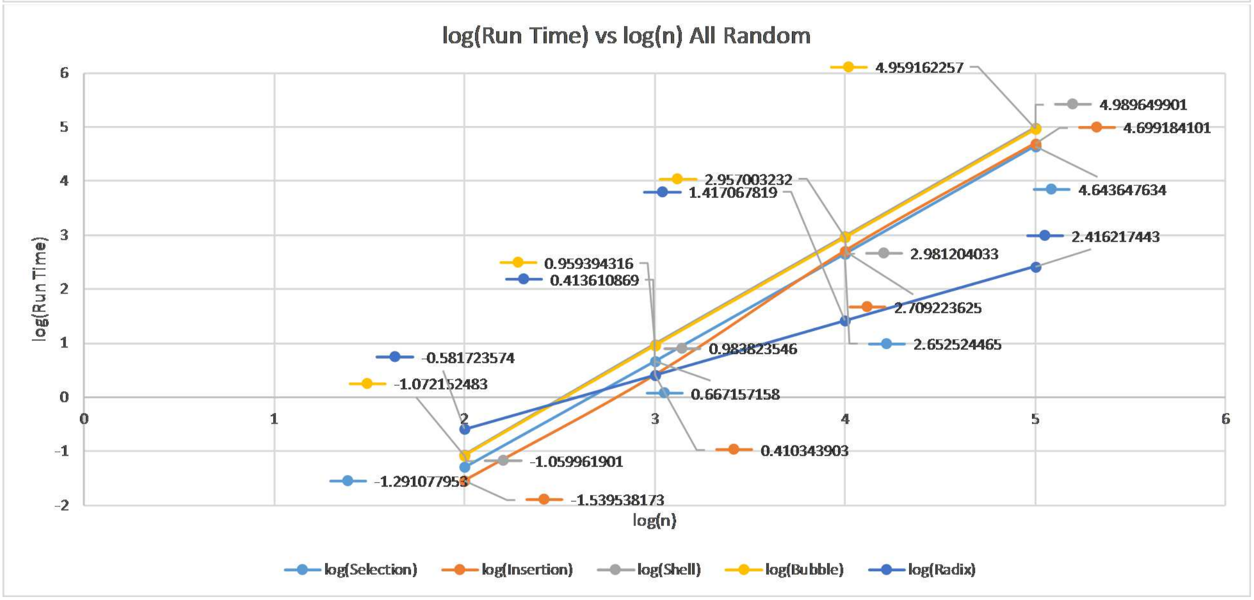

log(Comparisons) vs. log(n) Selection
log(Comparisons)
log(n)
0
2
4
6
8
10
12
0
1
2
3
4
5
6
9.698965661
7.698926573
9.698965661
9.698965661
5.698535493
7.698926573
7.698926573
3.694605199
5.698535493
5.698535493
3.694605199
3.694605199
log(Comp Inc)
log(Comp Dec)
log(Comp Rand)



(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.
Most all of the sorting algorithms follow in the footsteps of their theoretical counterpart. The fastest algorithms should be Bubble, Insertion and Radix when sorting an increasing set of numbers. This holds true. Typically, the randomly sorted and decreasing sets take the longest and have run times close to one another. The run times are symmetrical with how many comparisons are done. The more comparisons, the longer the run time. The algorithms with the same big O notation for their best, average and worst case all have around the same run times as well. As for radix sort, the run times are significantly better for larger sets of numbers, however the opposing algorithms are triumphant in the smaller sets of numbers.

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?
As stated in the discussion, the algorithms follow the theoretical analysis. The most efficient sorting algorithm for increasing sorted values are the Bubble, Insertion and Radix algorithms whereas for larger sets of numbers the radix sort has a significantly lower run time than the other algorithms with the selection sort coming in at second. The Bubble and Shell sort algorithms are incredibly inefficent for randomly sorted

and decreasing values and the Selection sort is incredibly inefficient for increasingly sorted values. Aside from the number of values being sorted and the arrangement of the values being sorted, the computers clocking speed as well as any unnecessary coding/computations in the sorting files can cause the data to have a longer run time than expected.