Project Cover Page

This project is a group project with up to three students per team. For each group member, please print first and last name and e-mail address.

1. Fang Wang : fangwang@tamu.edu

2.Alejandra Sandoval : alesandoval11@tamu.edu

3.Abraham Hinojosa: abrahamhino21@tamu.edu

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

1. Fang Wang: Created Github repository. Created BubbleSort and ShellSort class. Created input file generator to genearte input files for sorting and modified sort file to calculate running time and reset number of comparisons. Manage file permissions on repository and keep repository clean. Tested running time and number of comparisons for bubble sort and shell sort. Prepared report on description of assignment, program guideline, algorithms introduction, theoretical analysis. Participated in discussion and conclusion

2.Alejandra Sandoval: Created Insertion Sort and Selection Sort class. Tested running time and number of comparisons for insertion and selection sort.

3.Abraham Hinojosa: Coded radix sort. Tested running time for radix. Description and theoretical analysis radix sort

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)	Dr. Leyk's Class Lecture Notes <http://courses.cs.tamu.edu/teresa/csce221/pdf-lectures/algorithm-analysis_rev.pdf>
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 Fang Wang 02/18/2017
I certify that I have listed all the sources that I used to develop a solution to the submitted project and code.
Your signature Alejandra Sandoval 02/19/2017
I certify that I have listed all the sources that I used to develop solution to the submitted project and code.
Your signature Abraham Hinojosal 02/19/2017

CSCE 221 Programming Assignment 2 (200 points)

Program and Reports due February 19th 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. You will be required to manage this project with GitHub.

- **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 need to "untar" the file using the following command on linux:

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

     It will create the directory `221-A2-code`. You will be required to push the files in this directory to your team's repository. For more information on this, see the Git tutorial.
  3. Make sure that your code can be compiled using a C++ compiler running on a linux machine before submission because your programs will be tested on such a linux machine. Use `Makefile` provided in the directory to compile C++ files by typing the following command:

     ```
     make
     ```

     You may clean your directory with this command:

     ```
     make clean
     ```
  4. When you run your program on a linux machine, use Ctrl+C (press Ctrl with 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
     (c) Git tutorial
  6. Submission guidelines
     (a) Electronic copy of all your code, 15 types of input integer sequences, and reports in LyX and PDF format should be in the directory `221-A2-code`. This command typed in the directory `221-A2-code` will create the tar file (`221-A2-code-submit.tar`) for the submission to CSNet:

     ```
     make tar
     ```

     (b) 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 the sets of the sizes 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 12 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.

     The purpose of this assignment is to implement five sorting algorithms: selection sort, insertion sort, bubble sort, shell sort and radix sort in C++. We test the 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. The project is managed with GitHub.

  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 sorted into different classes. The main class is the sort class. Each sorting algorithms is built in a separate inheritance class. A switch option is built so that user can choose which sorting algorithms to use and input/output formats. Input is tested for validation. Exception will be thrown if input is invalid. Error will occur if the sequence is not sorted after sorting. Abstractions include funcitons like function to read input and function to show output.

  3. (5 points) **Algorithms.** Briefly describe the features of each of the five sorting algorithms.

     Selection Sortis a comparison based sorting algorithm that compares every element in the array until it finds the smallest number. After it finds the smallest number, it swaps with the element in the first position. After that it looks again for the smallest number in the unsorted array, and swaps it in the second position. This happens until the array is sorted.

     Insertion Sort is a comparison based sorting algorithm tha starts in the second index by comparing the elements to the left of the index and swaps them until they are sorted, then, it increases its index and finds a place for theat new element. That keeps on happening until the array is sorted.

     Bubble sort is comparison based sorting algorithm to compare and swap conjunctive integers until no swap is made. It sort one largest integer in one pass.

     Shell sort is comparison based sorting algorithm to divide array into segments and sort on the subarrays. The whole array is partially sorted after every partition and the gap is reduced until the whole array is sorted. Insertion sort is applied to each segment.

     Radix is a non comparison based algorithim that sorts numbers by paring integer keys and comparing them and arrangeing them from least to greatest it reapeats this on all the integers that conform the number starting in the once place and ending on the last digit. In other words it applies counting sort on each digit that makes up the number.

  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	$O(n\log n)$	depends	$O(n\log^2 n)$
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	$O(n\log n)$	depends	$O(n\log^2 n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$

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

Increasing order corresponds to best case. For shell sort, random order can be worst case. For selection sort and radix sort, the complexity is the same regardless of the order. For all other sorts, decreasing order corresponds to worst case and random order can be average case.

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.

   The input.cpp file will generate 3 sequences of integers, in increasing, random and decreasing numbers. Different sorting algorithms are then used to sort these sequences and running time and number of comparisons are then recorded. For each size n, 3 input sets are generated and the running time and number of comparisons are the average result.

RT	Selection Sort			Insertion Sort			Bubble Sort		
n	inc	ran	dec	inc	ran	dec	inc	ran	dec
100	0.0237	0.0267	0.0238	0.0013	0.0146	0.0292	0.001	0.0692	0.0858
10^3	2.094	2.149	2.175	0.01	1.809	3.702	0.01	7.39	8.55
10^4	210.2	210.8	219,40	0.0800	215.9	346.8	0.12	745	855.2
10^5	21080	21110	22080	0.83	16680	41830	2	75700	84270

RT	Shell Sort			Radix Sort		
n	inc	ran	dec	inc	ran	dec
100	0.0064	0.0153	0.0096	0.1542	0.1534	0.154
10^3	0.1	0.25	0.16	1.52	1.51	1.51
10^4	1.8	3.62	2.4	15.13	15.12	15.16
10^5	18	48	28	152.9	151.5	152.6

#COMP	Selection Sort			Insertion Sort		
n	inc	ran	dec	inc	ran	dec
100	4950	4950	4950	99	2572	4950
10^3	499500	499500	499500	999	245954	499500
10^4	49995000	49995000	49995000	9999	25038784	49995000
10^5	4999950000	4999950000	4999950000	99999	2505657173	4999950000

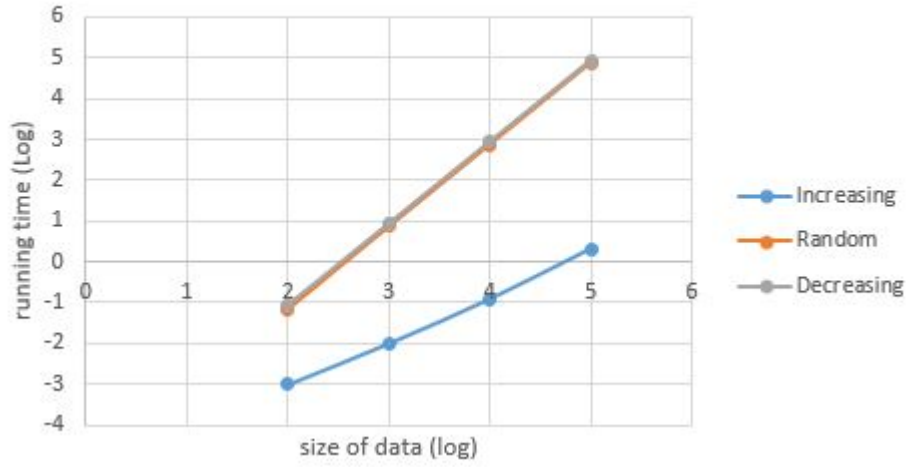
#COMP	Bubble Sort			Shell Sort		
n	inc	ran	dec	inc	ran	dec
100	99	4900	4950	418	893	654
10^3	999	497904	499500	7098	13767	10490
10^4	9999	49987125	49995000	100855	197583	137014
10^5	99999	4999909530	4999950000	1308360	2594278	1773328

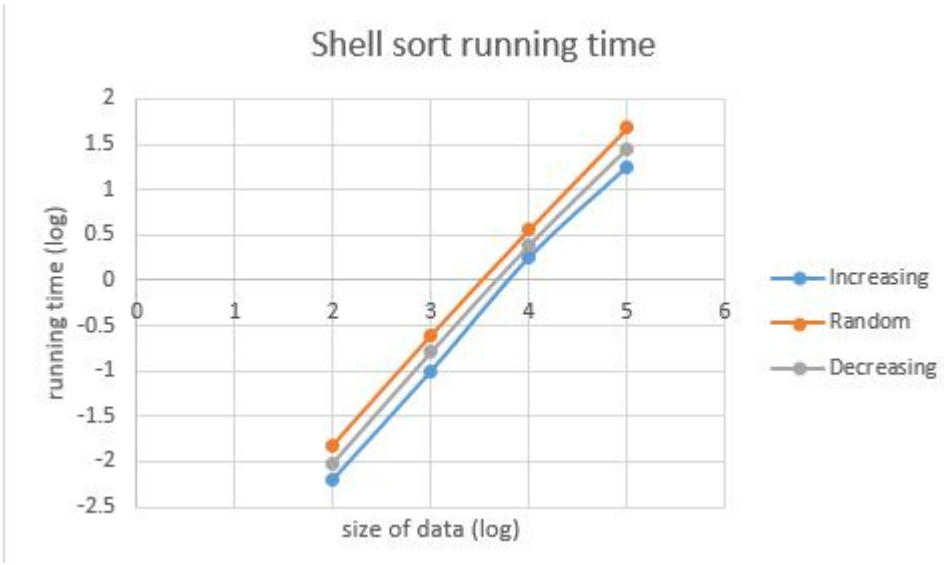
inc: increasing order; dec: decreasing order; ran: random order

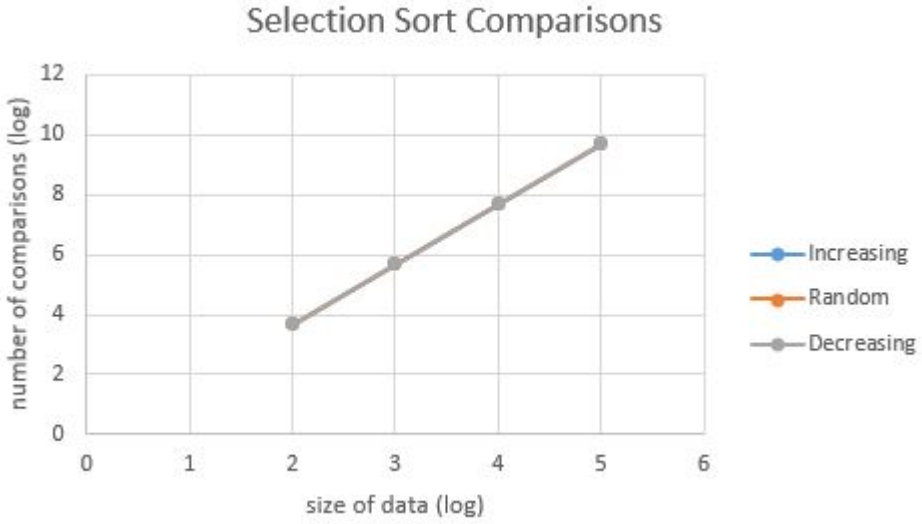
   (b) For each of the five sort algorithms, graph the running times over the three input cases (inc, ran, dec) versus the input sizes (n); and for each of the first four algorithms graph the numbers of comparisons versus the input sizes, totaling in 9 graphs.
   HINT: To get a better view of the plots,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


Selection Sort Running Time
running time (Log)
size of data (log)
Increasing
Random
Decreasing

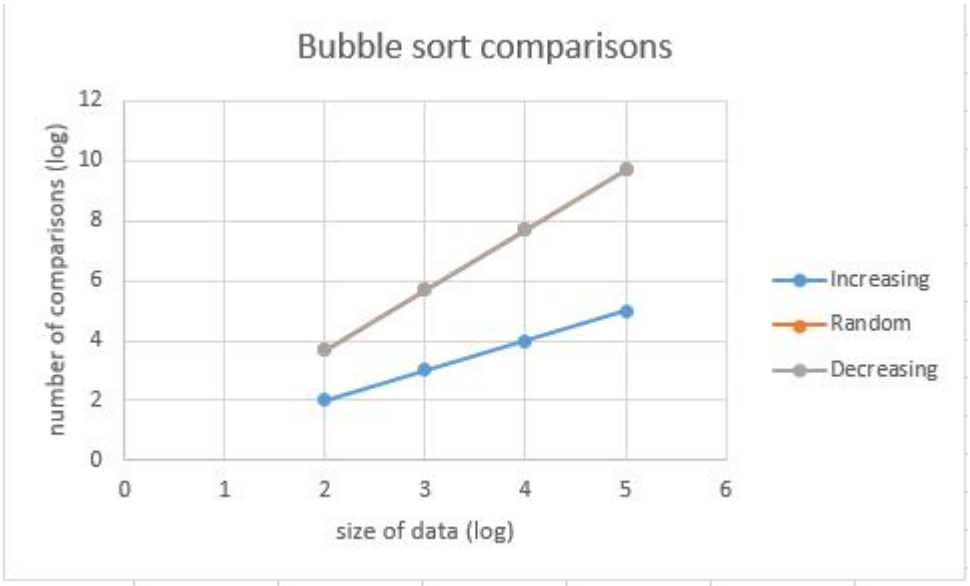

Insertion Sort Running Time
running time (log)
size of data (log)
Increasing
Random
Decreasing

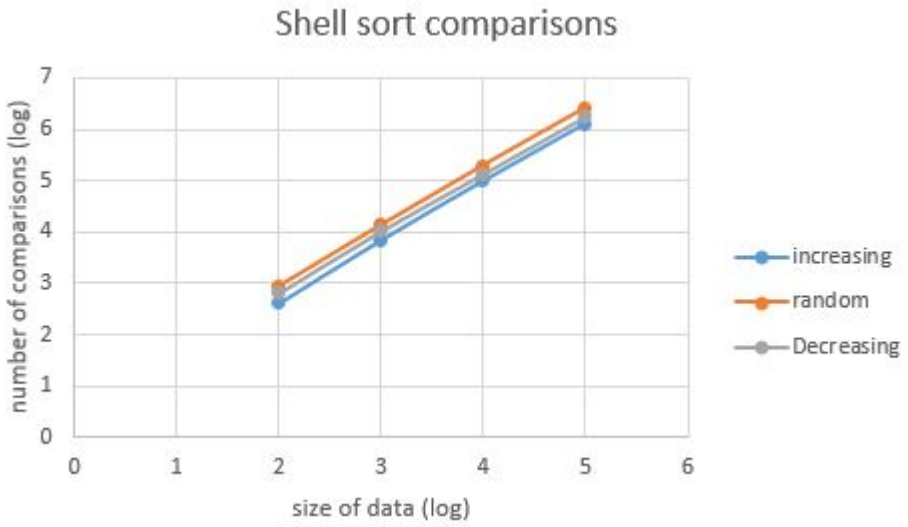

Bubble sort running time
running time (Log)
size of data (log)
Increasing
Random
Decreasing

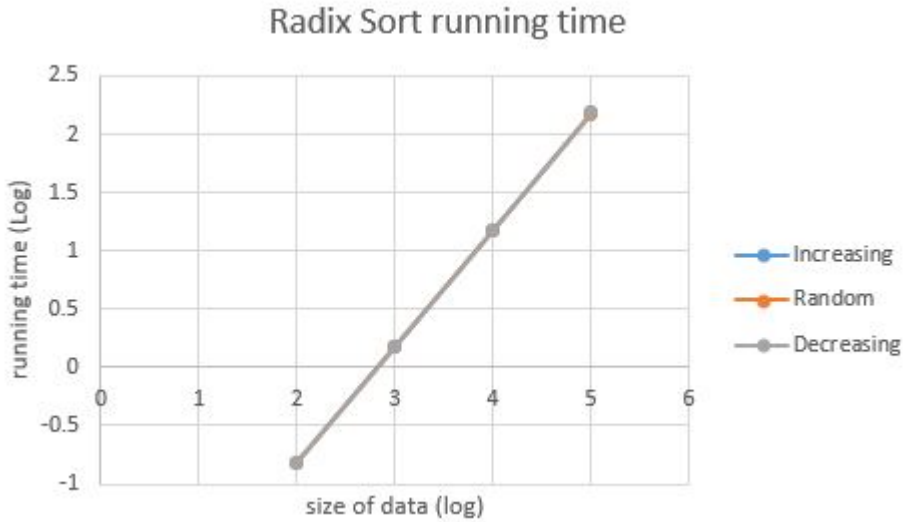

Shell sort running time
running time (log)
size of data (log)
Increasing
Random
Decreasing


Selection Sort Comparisons
number of comparisons (log)
size of data (log)
Increasing
Random
Decreasing


Insertion Sort Comparisons
number of comparisons (log)
size of data (log)
increasing
random
Decreasing







(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.

The experimental results correspond to the theoretical analysis. For bubble sort, the best case costs $n-1$ comparisons and the worst case costs $\frac{n^2-n}{2}$ comparisons. So the increasing case, the running time and number of comparisons are linear to size of data and the slope is 1. For worse case and average case, the running time and number of comparisions are linear to size of data and the slope is 2 because $\log n^2 = 2\log n$. So the result makes sense.

Shell Sort: the best case costs $O(n\log n)$ and the worst case costs $O(n(\log_2 n)^2)$ and the average case is between $O(n(\log_2 n)^2)$ and $O(n^{3/2})$.

Selection Sort: The experimental results relate to the theoretical analysis which is $O(n^2)$ because in the threoretical analysis it has the worst running time and the experimental results prove that. The computational results match what we have learned in class. This is comparing each of them everytime and therefore has the worst comparing case of all the sorts and the same comparison number and the same running time which we can see it in the graphs(they have almost identical lines.

Insertion Sort: The experimental results relate to the theoretical analysis because the number of comparison for the best case is O(n-1) which matches the number of comparisons by the experimental results. The worst case of insertion sort is $O(n^2)$ which is different from the best case, which in this case would be the decreasing and the random order. In the experiment that we conducted we can see that that is what happens, the numbers change drastically and the graphs show the the difference between the best and worst case.

The theoretical analysis would let us infer that radix sort would have the lowest runtime due to its capacity to perform without comparisions and its worst best and average case $O(nk)$ making the running time of the algorithm linear it increased it was a constant increment of one. The radix sort code was not as efficient as it should have been we assume it was due to the check for negative elements it slowed down furthemore we infer the code was written in an inefficient manner that affected the performance of the radix sort.

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?

   For increasing cases, the insertion sort and bubble sort work best and selection sort takes the longest time. Because insertion and bubble sort take $n-1$ comparisons in the best case and selection sort takes $O(n^2)$. The random case and decreasing case are pretty similar. Shell sort and radix sort work best for worst case in decreasing order. This makes sense, because the other three algorithms are comparisons based and the running time increases significantly as the size of data increases. Radix sort did not perform as expected. It did not have the fastest run time as predicted. We assume the code was written inefficiently and that the conditions to sort negative numbers also affected the readix sort performance.

- **(20 points) Use of Git**

  1. (10 points) Host the final working version of your project on your team's GitHub repository. **This repository must be private.**
  2. (10 points) Demonstrate knowledge of branching. Create branches for at least three of the six sorting algorithms. Once the sorting algorithm is working, merge the branch with the master branch but DO NOT delete the branch used to implement the algorithm.

  Each member of your team should have roughly equal amount of contributions. This would be approximately implementing two of the six sorting algorithms and roughly equal contributions to the final report. This information will be used to assign the final grade for this assignment.

  If you are unfamiliar with git or need a refresher, see the Git tutorial.

- `Quick Start Guide`

  1. Choose a team member from each group (max of three members) to create a **private** repository on `github.tamu.edu`.
  2. This same team member will add each team member as a collaborator to the repository.
  3. One of the team members should download the supplementary code from the website. `Untar` the supplementary code from the website.
  4. Inside the directory created by untaring the file, use the command '`git init`' to create a local repository in this directory.
  5. Use the command '`git remote add origin` [address to your repository]' to create the link between your local repository and the remote repository.
  6. Use the command '`git push origin master`' to push the supplementary code to your team's repository.
  7. Your team can begin to create branches for the sorting algorithms.
  8. See Git tutorial for more information.