Project Cover Page

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

1. Jessica Fang, jessicafang@tamu.edu

2. Justin Gonzalez, pika411@tamu.edu

3. Juan Pablo Zambrano, jpz95@tamu.edu

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

1. Jessica
wrote, tested, and graphed shell sort and selection sort
increasing, decreasing, random graphs
wrote parts 3 and 4 of report

2. Justin
wrote, tested, and graphed bubble sort and insertion sort
wrote parts 1 and 2 of report

3. Juan
wrote, tested, and graphed radix sort
wrote parts 6 and 7 of report

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 Jessica Fang Date 02/18/2015

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 Juan Pablo Zambrano Date 02/22/2015

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 Justin Gonzalez Date 02/21/2015

CSCE 221 Programming Assignment 2 (200 points)

Programs due February 19th by 11:59pm
Reports due on the first lab of the week of 29th

- **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
     (c) Powerpoint slides: Tracing Sorting Algorithms
  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 comparisions 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 was to show understanding of sort construction and utilization with analysing of diffrent sorts. In this assignment we implemented the basic sorts used in C++: Bubble, Selection, Insertion, Shell and Radix. Each sort was tasked to sort an input of integers from a text file. Three input types were tested per Sort, decreasing integer input, increasing interger input, and random integer input. The Program allows for diffrent option of output such as including the following in the output file: unsorted list, sorted list run time and number of comparisons. Commands are given to define which sort to use , input filename, output file name, and output options. To run the program the user must go to the directory of the files and type "make" to compile the files. Afterwords typing ./sort B .... to sort using the bubbleSort, followed by arguments which correspond to what the user wants to do. For input the first line represents the number of elements to be sorted, each integer is on a seperate line.

  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*.

     C++ Constructs: The program consists of a main Sort class which controls the sorting commands. In addition to these a Option class is contained to return the instructions of the program to the user, aswell as handle arguments inputed by the user. The program begins by calling the option class, in which then the user inputs a command to be executed i.e "./sort -a B -f bubbleinput.txt -o Increasing2.txt -t -c -p " here the program uses the bubble sort algorithm and print out the sorted and unsorted list, along with runtime and number of comparisons. All sorting operations are called within the Sort class. Each Sort type is externally defined in its own cpp file and called using Sort.

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

     Bubble. Compares two adjacent elements and swaps if they are in the wrong order. Most practical when list is mostly sorted with a few elements out of order.

     Insertion. Takes an element, finds the location in the list where the element is supposed to go, and inserts it in the list. Not as efficient on large lists.

     Radix. Uses stable counting sort starting from the ones place and moving up (to the tens, hundreds, etc.). Radix sort is non-comparative (does not compare the elements to each other) and therefore efficient on large lists.

     Selection. Searches for the lowest element and puts it in the correct location (next index of list) by swapping. Is not efficient on large lists.

     Shell. Uses insertion sort to sort elements at every x number of elements (gap) and then reduces the gap and sorts agains. Runtime is dependent on gap size.

  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)	O(n log_2n) or O(n^2)	O(n^2)
Radix Sort	O(dn)	O(dn)	O(dn)

Complexity	inc	ran	
Selection Sort	Best:$O(n^2)$	Average:$O(n^2)$	
Insertion Sort	Best:$O(n)$	Average:$O(n^2)$	
Bubble Sort	Best:$O(n)$	Average:$O(n^2)$	
Shell Sort	Best:$O(n)$	Average: O(n log_2n)	W
Radix Sort	Best:$O(dn)$	Worst:$O(dn)$	

     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	0.02 ms	0.02 ms	0.01 ms	0.005 ms	0.034 ms	0.083 ms	0.004 ms	0.048 ms	0.069
10^3	1.94 ms	1.94 ms	1.94 ms	0.015 ms	2.632 ms	4.213 ms	0.010 ms	5.167 ms	5.303
10^4	190 ms	200 ms	200 ms	0.064 ms	180 ms	290 ms	0.059 ms	450 ms	460 m
10^5	19630 ms	19650 ms	20260 ms	0.634 ms	17290 ms	41940 ms	0.401 ms	45390 ms	45810

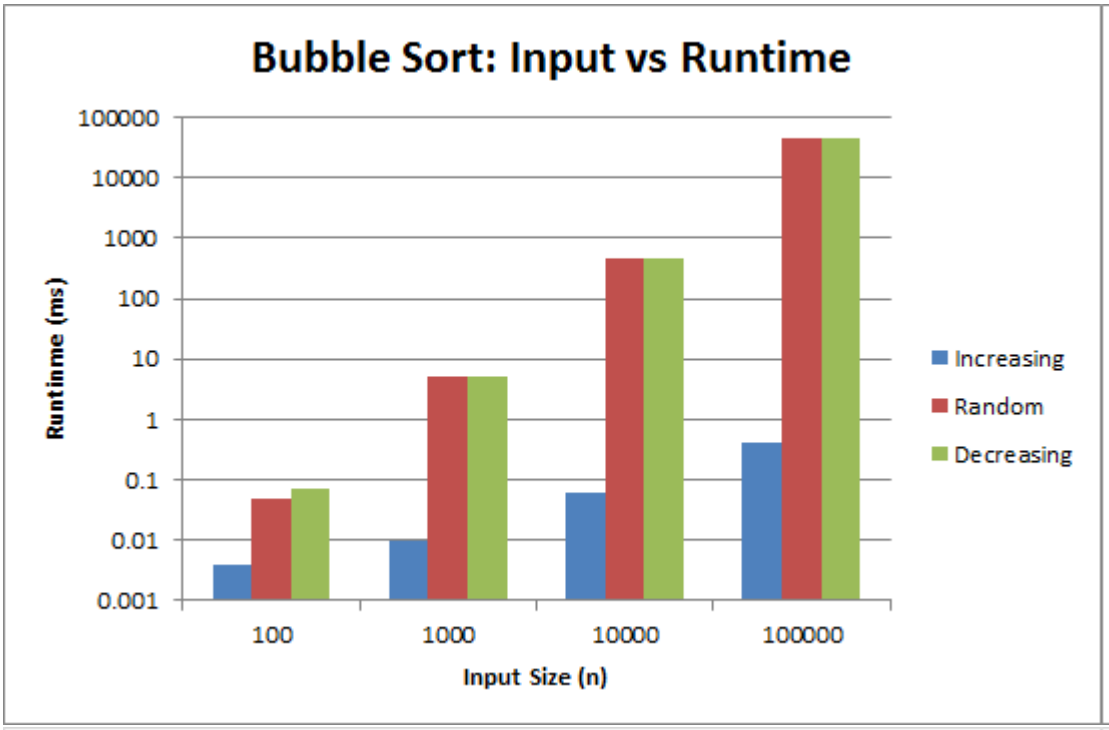
RT	Shell Sort			Radix Sort		
n	inc	ran	dec	inc	ran	dec
100	0.004 ms	0.003 ms	0.003 ms	0.20 ms	0.04 ms	0.10 ms
10^3	0.06 ms	0.06 ms	0.06 ms	0.42 ms	0.66 ms	1.02 ms
10^4	0.97 ms	0.98 ms	0.98 ms	10 ms	10 ms	10 ms
10^5	10 ms	30 ms	20 ms	100 ms	130 ms	100 ms

#COMP	Selection Sort			Insertion Sort		
n	inc	ran	dec	inc	ran	dec
100	4950	4950	4950	99	2440	4950
10^3	499500	499500	499500	999	262916	499500
10^4	49995000	49995000	49995000	9999	24925391	49995000
10^5	4999950000	4999950000	4999950000	99999	2505899060	4999950000

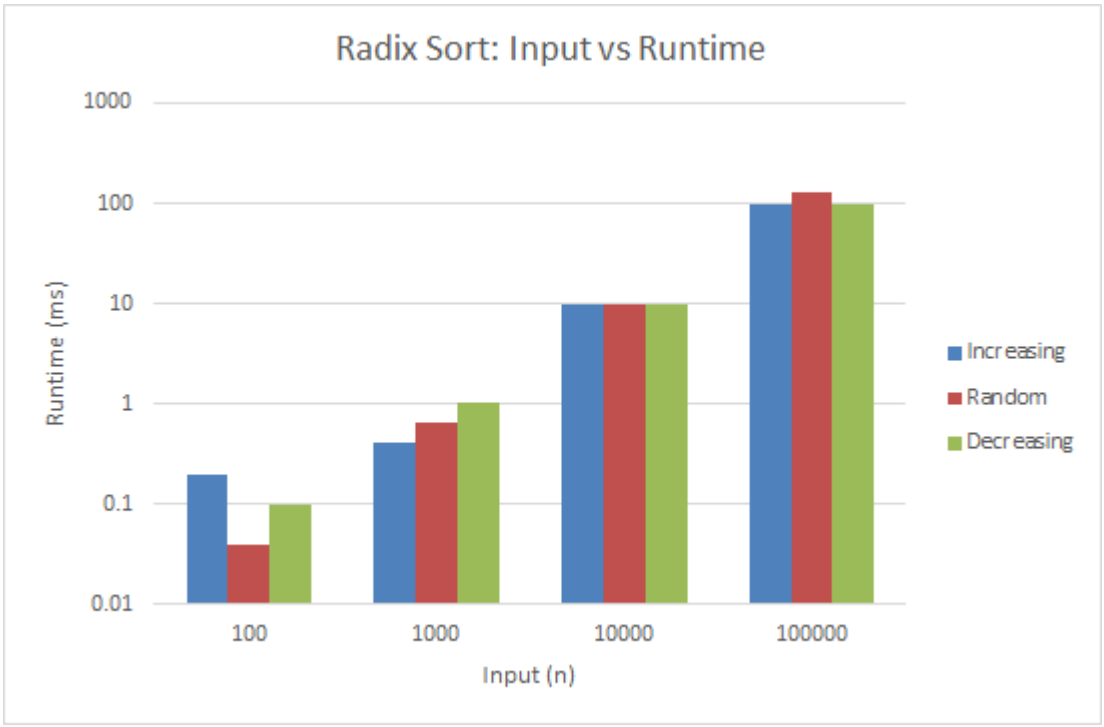
#COMP	Bubble Sort			Shell Sort		
n	inc	ran	dec	inc	ran	dec
100	99	4895	4950	413	659	437
10^3	999	498972	499500	7090	10129	7490
10^4	9999	49967739	49995000	100844	150639	104520
10^5	99999	4999764864	4999950000	1308346	1942296	1345378

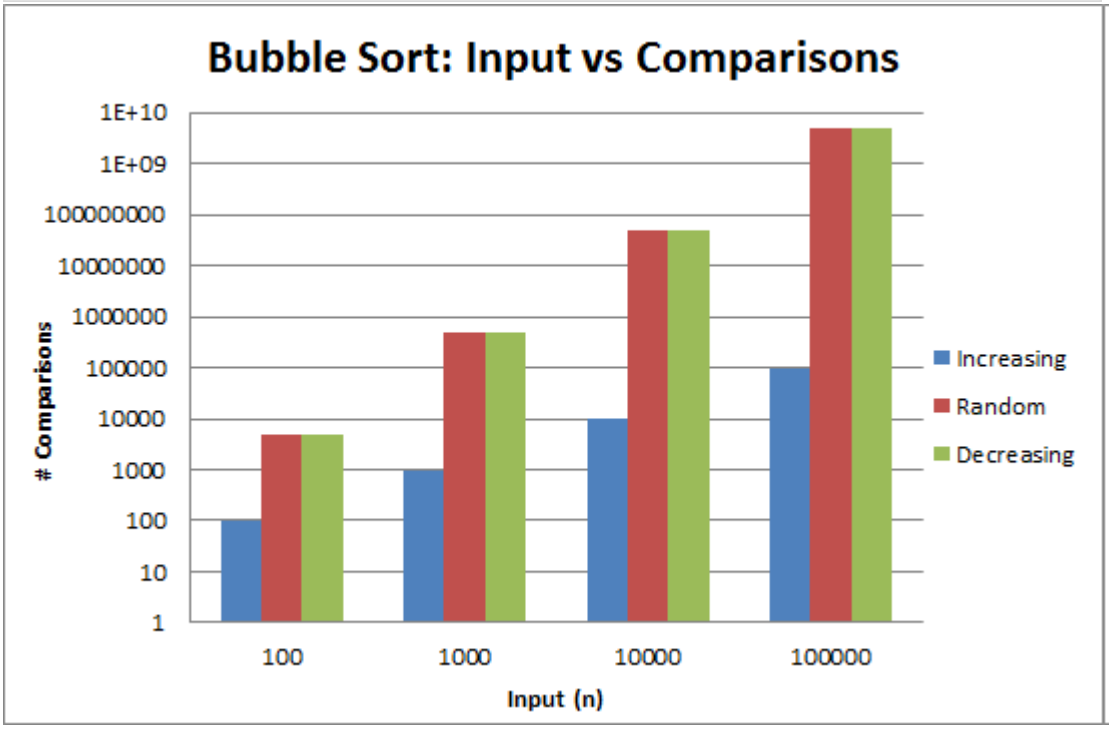
inc: increasing order; dec: decreasing order; ran: random order

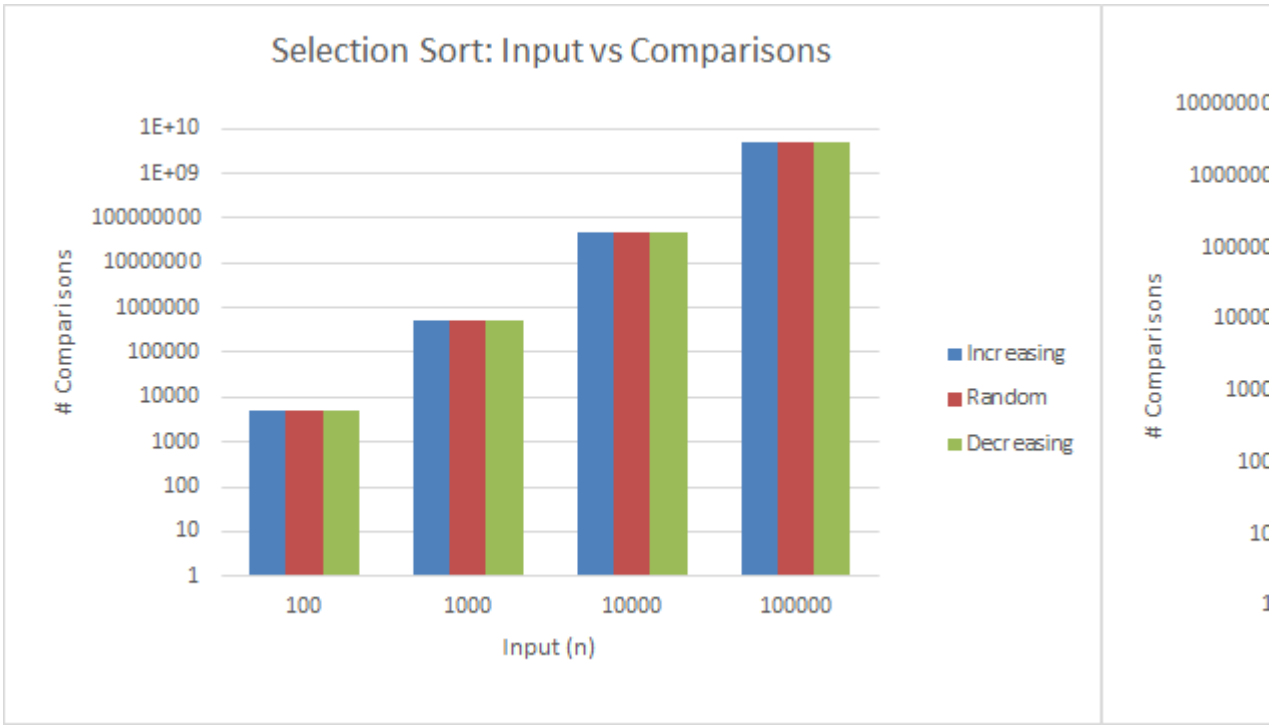
(b) For each of the five sort algorithms, graph the running times over the three input cases (inc, ran, dec) versus the input sizes (n); and for each of the first four algorithms graph the numbers of comparisons versus the input sizes, totaling in 9 graphs.
HINT: To get a better view of the plots, *use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


Bubble Sort: Input vs Runtime
Runtinme (ms)
100000
10000
1000
100
10
1
0.1
0.01
0.001
100
1000
10000
100000
Input Size (n)
Increasing
Random
Decreasing

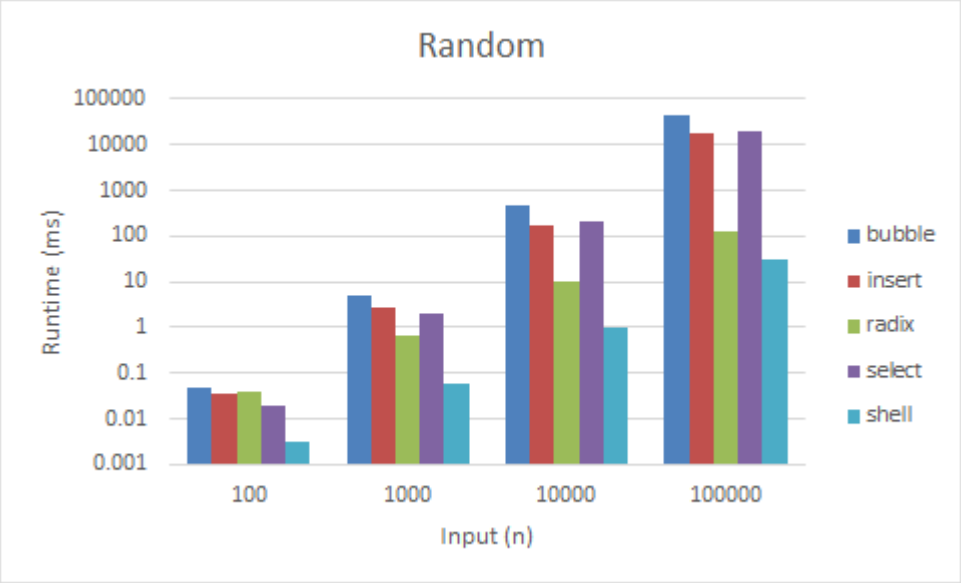

Selection Sort: Input vs Runtime
Runtime (ms)
100000
10000
1000
100
10
1
0.1
0.01
0.001
100
1000
10000
100000
Input Size (n)
Increasing
Random
Decreasing


Radix Sort: Input vs Runtime
1000
100
10
1
0.1
0.01
Runtime (ms)
100
1000
10000
100000
Input (n)
Increasing
Random
Decreasing


Bubble Sort: Input vs Comparisons
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
Comparisons
100
1000
10000
100000
Input (n)
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.

   Discussion: When compared side by side, the experimental results are a good reflection of the theoretical analysis we performed. Overall, shell sort seems to be the fastest sorting algorithm for all input sizes and input cases with radix sort close behind it. However, according to our theoretical analysis, radix sort should have been faster. This discrepancy could have been due to the extra checks for the largest and smallest integer value - which were needed to run the sort correctly and adjust for negative values, respectively - in the radix sort code. As the input sizes grew, more time was spent in these loops and might have affected its overall runtime. In addition, it is possible that there was more network traffic done while performing the radix sort experimental runtimes since it was performed during lab time, causing slower runtimes.

   Despite this, radix sort was still faster than bubble sort, insertion sort, and selection sort for larger input sizes. Although selection sort was by far the worst in the theoretical analysis since it compared each element for every iteration, its runtimes were suprisingly faster for the decreasing input case in comparison to insertion sort and bubble sort. On reflection, the experimental runtimes for these were done by different members at different times and might have led to different conditions while testing.

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?

   Conclusions: Given the data we collected, shell sort had the best runtimes overall, and radix sort had the second best. For the increasing input case, bubble sort was the fastest with insertion sort close behind, but since the input values were already sorted, it doesn't carry much weight. In theory, radix sort should have performed the best - especially for the random and decreasing input cases - and selection sort should have performed the worst. However, our data does not accurately portray that and so does not agree with the theoretical analysis learned in class. In order to have gotten more reliable results, the experiments should have been performed on the same machine and at the same time as it would have provided similar and fairer testing conditions for all of the algorithms. Nevertheless, this experiment still provided a solid foundation on the advantages and disadvantages different sorting algorithms.