CSCE 221 Programming Assignment 2 (200 points)
Robert Quan + Others

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 */
   ...
   ```

- **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 was to gain a better understanding of the relative efficiency of different sorting algorithms on a variety of input types, both in terms of number of comparisons and time taken. We are assigned to implement various sorting algorithms and experimentally observe their running times. We are supposed to implement 5 sorting algorithms: selection sort, insertion sort, bubble sort, shell sort, and radix sort. The program is run by the command ./sort [-a] [-f] [-o] [-h] [-d] [-p] [-t] [-c]. The command line arguments correspond to the desired algorithm, input file, output file, whether or not the user wants to display the help, display input, display output, display running time in milliseconds, and display number of comparisons, excluding radix sort because radix sort is a non-comparison-based sorting algorithm. The user should have the data in an input file, specified in the command line arguments passed to the main function in the program. Further, we learned the basic use of git and GitHub, as well as some experience in programming as part of a group.

  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*.

     We split the program into classes so that we could have a general virtual 'sort' class which could later be cast to one of a number of child classes. This allowed us to handle each preference of sorting type separately while allowing a common interface for the user. Generic programming features are the general algorithms executed by the various classes, as these are implementable in a similar fashion regardless of language. Object-oriented features include the use of a virtual class which is later cast to a child based upon command line inputs.

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

     Radix Sort- Radix sort uses a series of counting sorts from the least significant to the most significant places on a value, necessitating a stable counting sort algorithm, and taking in values up signed integers between $(-2^{15})$and $(2^{15}-1)$. It is capable of sorting negative numbers by supplying an offset to make all values positive, and then returning them to their original values with a commensurate negative offset at the end of the process. This particular implementation of radix sort utilizes a byte-level counting sort, which then conducts itself in the usual manner of counting sorts by using three arrays, scanning for indices, cascading addition in the vocabulary array, and then using inverted index referencing to determining the value and number of array cells to be filled. It can sort in linear time with the given constraints of data.

     Shell Sort- Shell sort uses a technique of segmenting normally distributed data across an interval into equal sub-intervals. The allocated into the correct sub-interval, and is sorted using insertion sort. Afterwards, the sub-intervals are stitched together, resulting in a sorted list. Because the sub-intervals allocation allows an insertion sort operation to be nearly sorted already, it can approach constant time given normally distributed data, and the other operations are linear.

     Bubble Sort- Bubble combs linearly from the first element in the array to the last element of interest, taking up the largest element that it finds and comparing it to its upper neighbor. After each iteration, it becomes uninterested in the next highest element of interest in the array, thereafter restricting its search to those elements that it considers unsorted. In this way it will always handle data in at least $O(n^2)$ time.

     Insertion Sort- Insertion sort, similarly to Bubble sort, combs linearly through the array. However, instead of carrying the largest element to the top, it instead will compare each element to its neighbors and carry small elements to the bottom. An element *x* at **[a]** will be compared to its neighbor *y* at **[a+1]**: if it is larger, then it will switch their indices. Then, *y* at **[a]** will be compared to the element at **[a - 1]** . This will continue until all elements are sorted, and it runs also in $O(n^2)$ time.

     Selection Sort- Selection sort will go through the element from smallest element to largest, and on each iteration compare the first element to every other in the list. If it finds that another element is smaller, then it will place that one at the front of the list. This algorithm has both a best and worst case of n^2time, and is horribly inefficient.

  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	$O(n \log n)$	$O(n(\log n))^2)$	$O(n(\log n))^2)$
Radix Sort	$O(n)$	$O(n)$	$O(n)$ or $O(k)$

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	$O(n \log n)$	$O(n(\log n))^2)$	$O(n(\log n))^2)$
Radix Sort	$O(n)$	$O(n)$ or $O(k)$	$O(n)$ or $O(k)$

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.
   For the experiments, we created lists of incrementing, random and decrementing values. We extensively tested each algorithm and took an average of the run times. This rigorous testing led us to the following results:

RT	Selection Sort (ms)			Insertion Sort (ms)			Bubble Sort (ms)		
n	inc	ran	dec	inc	ran	dec	inc	ran	dec
100	0.043	0.052	0.046	0.003	0.031	0.05	0.003	0.098	0.078
10^3	3.34	3.51	3.44	0.012	2.23	4.47	0.011	7.21	7.67
10^4	175	196	279	0.134	143	235	0.092	375	329
10^5	19332	21780	32388	0.713	11359	23823	0.598	44540	49056

RT	Shell Sort (ms)			Radix Sort (ms)		
n	inc	ran	dec	inc	ran	dec
100	0.007	0.028	0.12	0.062	0.056	0.049
10^3	0.107	0.428	0.185	0.33	0.328	0.404
10^4	1.143	4.88	2.16	2.28	1.09	1.91
10^5	10.205	55.9	19	20.43	20.56	20

#COMP	Selection Sort			Insertion Sort		
n	inc	ran	dec	**ran**	**inc**	dec
100	4950	4950	4950	2608	99	5049
10^3	499500	499500	499500	250035	999	500499
10^4	49995000	49995000	49995000	25222691	9999	49994956
10^5	4999950000	4999950000	4999950000	2506169125	99999	4999950203

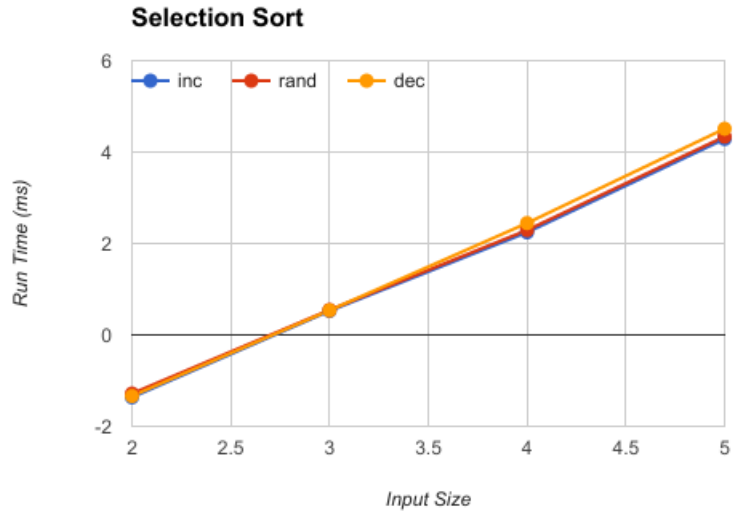
#COMP	Bubble Sort			Shell Sort		
n	**ran**	**inc**	dec	**ran**	**inc**	dec
100	4929	99	4950	902	503	668
10^3	499490	999	499500	15683	8006	11716
10^4	49991084	9999	49994999	258010	120005	169245
10^5	4999508670	99999	4999950000	4259397	1500006	2196626

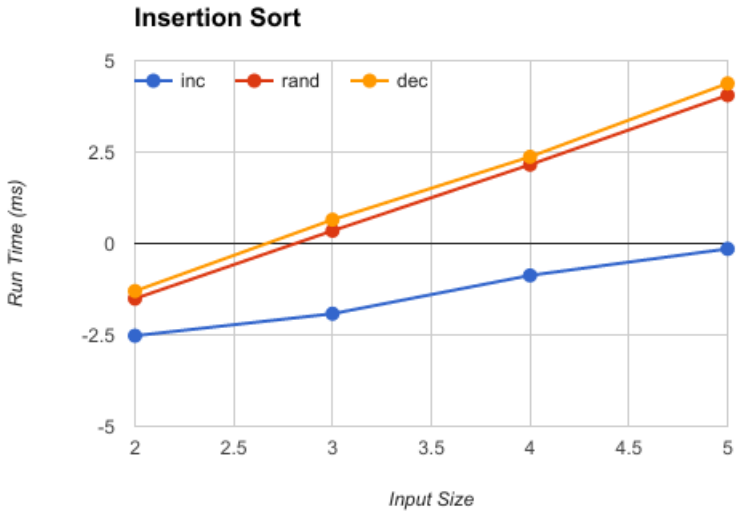
inc: increasing order; dec: decreasing order; ran: random order

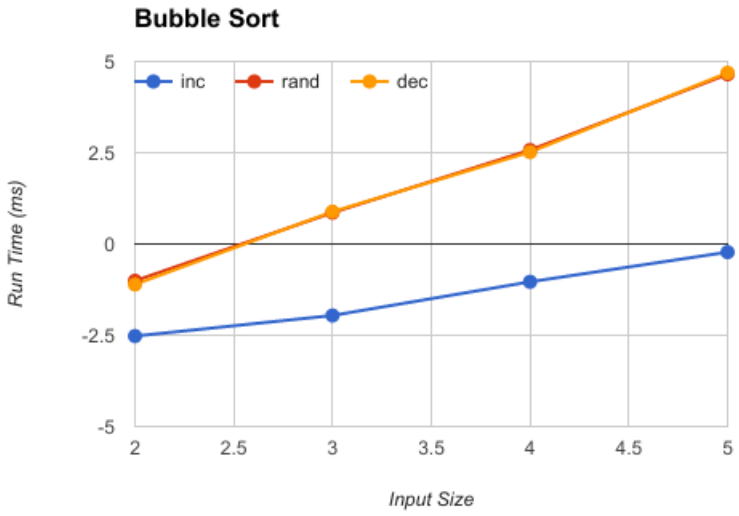
As we can see, the run time and number of comparisons do increase with number of elements on the list. The values for the number of comparison match the intended Big O notation.

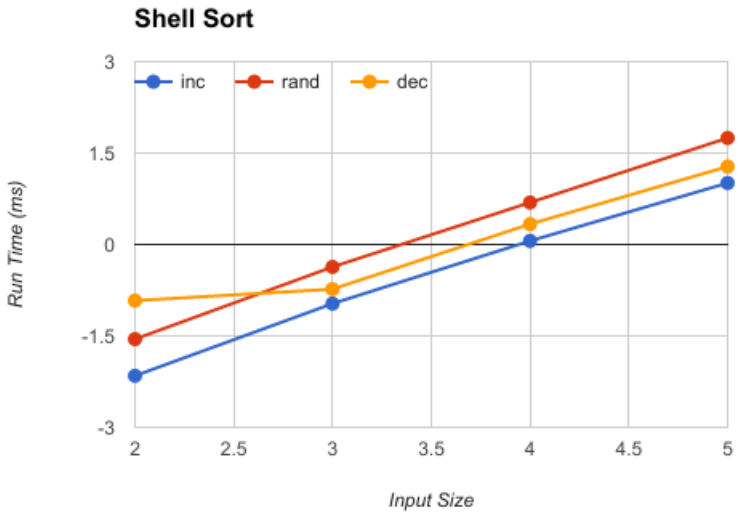
(a) For each of the five sort algorithms, graph the running times over the three input cases (inc, ran, dec) versus the input sizes (n); and for each of the first four algorithms graph the numbers of comparisons versus the input sizes, totaling in 9 graphs.

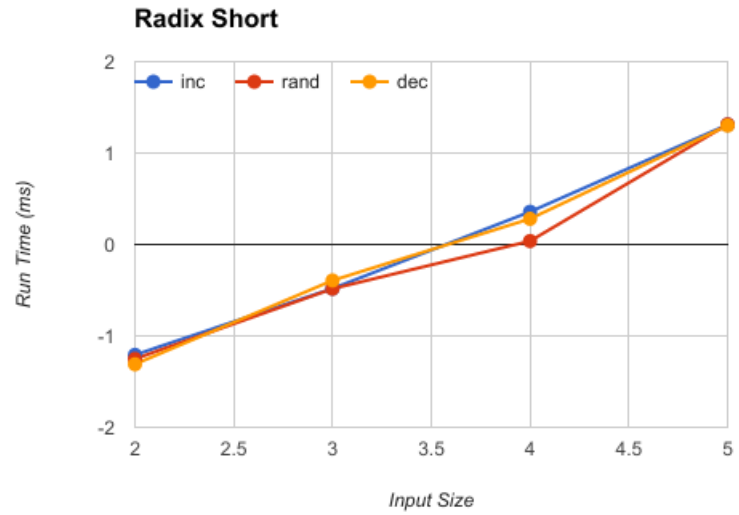
i. All of these graphs have been plotted in Log base 10 on both values.

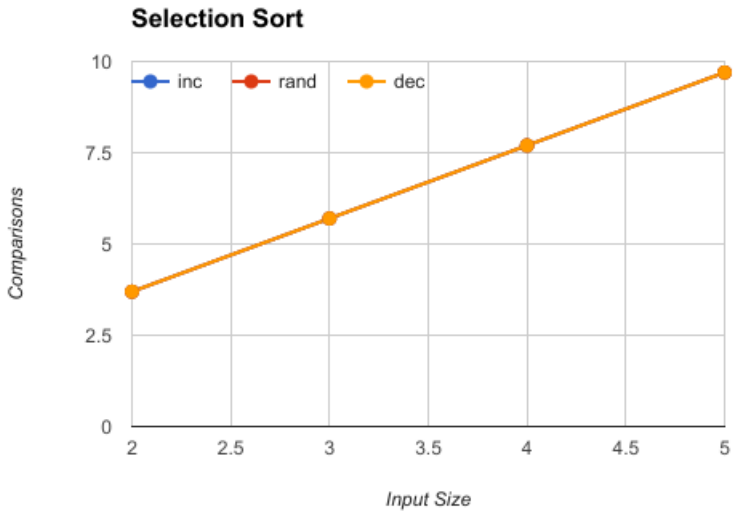


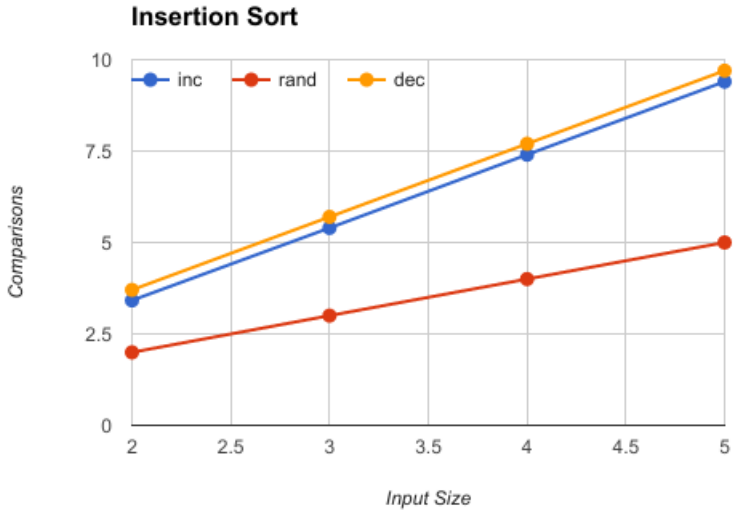


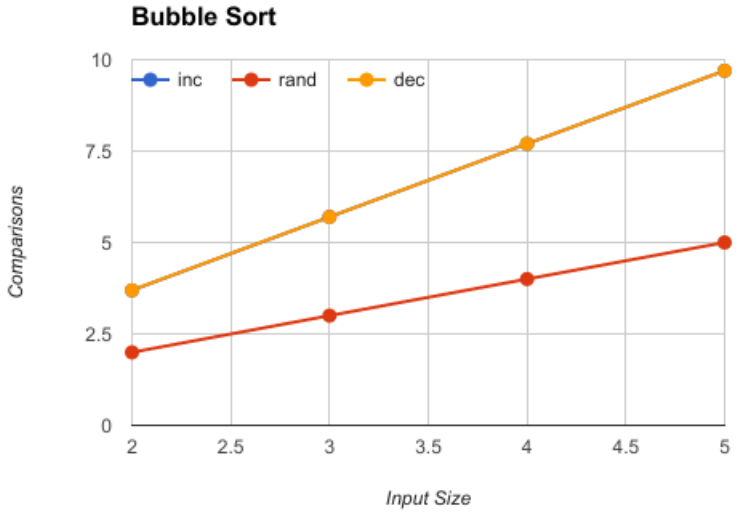

Bubble Sort
inc
rand
dec
5
2.5
0
-2.5
-5
2
2.5
3
3.5
4
4.5
5
Run Time (ms)
Input Size

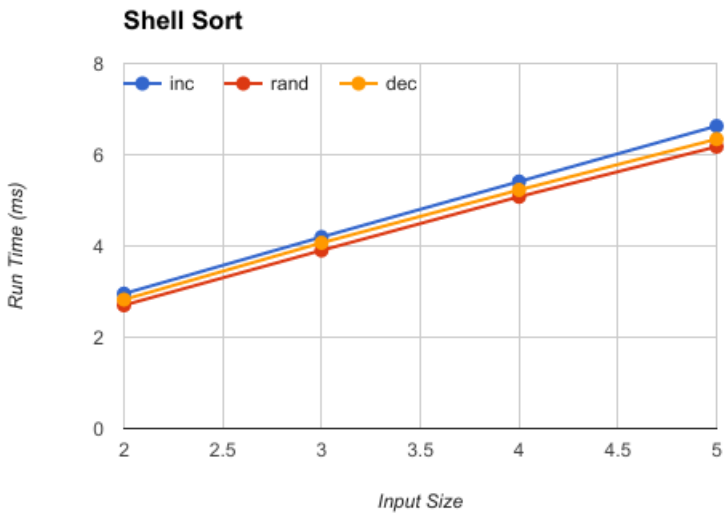

Shell Sort
inc
rand
dec
3
1.5
0
-1.5
-3
2
2.5
3
3.5
4
4.5
5
Run Time (ms)
Input Size


Radix Short
inc
rand
dec
2
1
0
-1
-2
2
2.5
3
3.5
4
4.5
5
Run Time (ms)
Input Size


Selection Sort
inc
rand
dec
10
7.5
5
2.5
0
Comparisons
2
2.5
3
3.5
4
4.5
5
Input Size

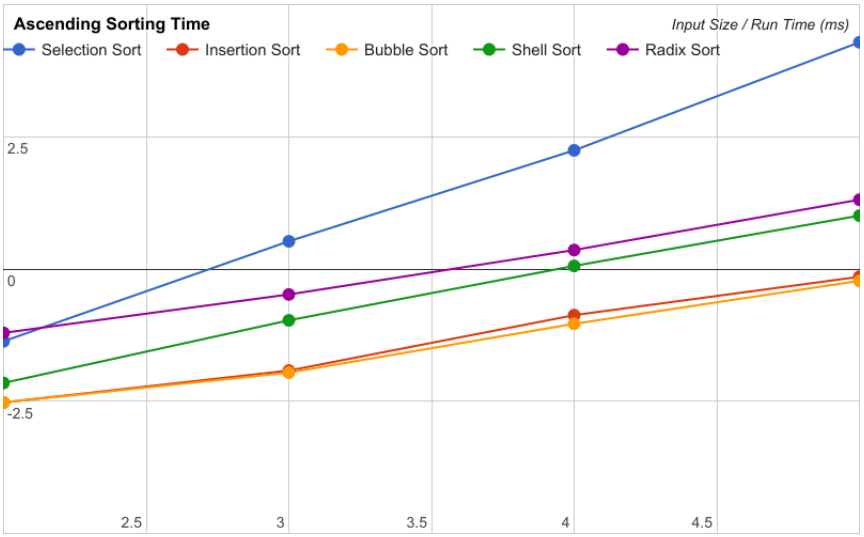

Insertion Sort
inc
rand
dec
10
7.5
5
2.5
0
Comparisons
2
2.5
3
3.5
4
4.5
5
Input Size

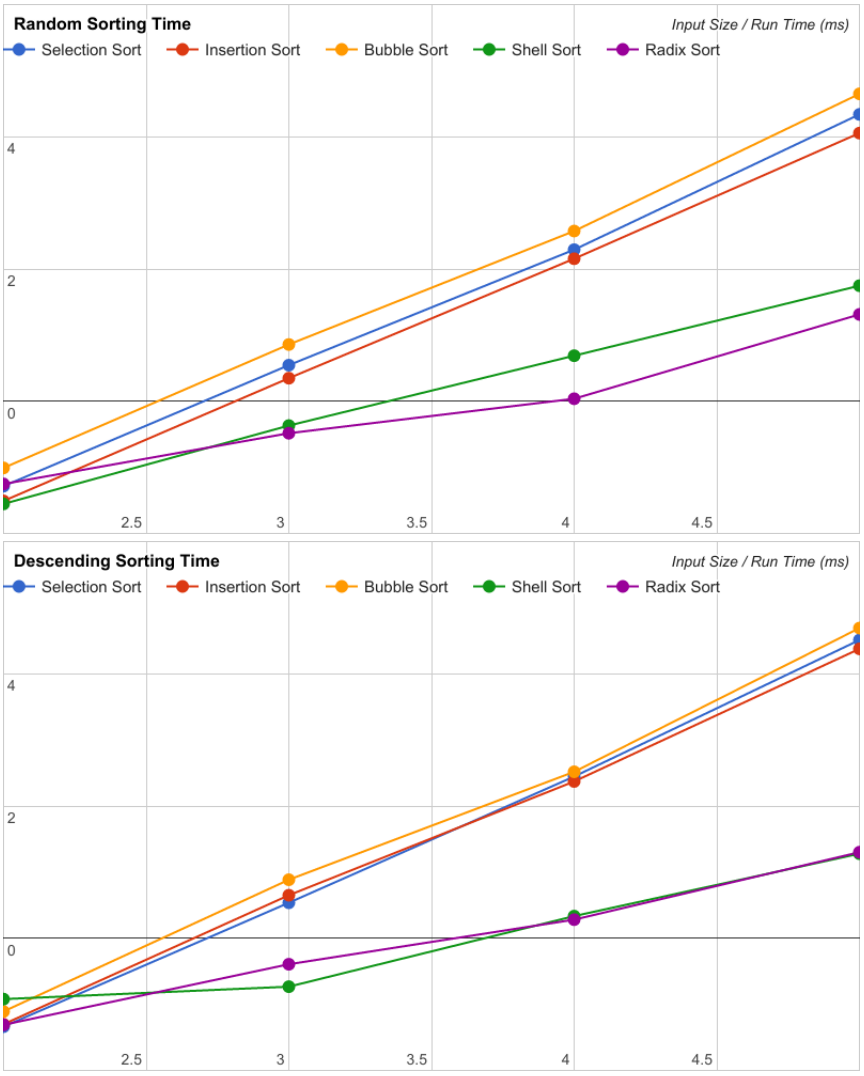

Bubble Sort
inc
rand
dec
10
7.5
5
2.5
0
Comparisons
2
2.5
3
3.5
4
4.5
5
Input Size



HINT: To get a better view of the plots, *use logarithmic scales* for both x and y axes.
Just to clarify, the last graph (Shell Sort) should be (Comparisons) on the y-axis.

(a) 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.

1. (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.

   We can see a correlation for between the theoretical and experimental results. For **Selection Sort** we notice that the BigO asymptotic function is n^2. From our results we can see that we have 4950 comparisons done on to sort our algorithm in all three cases. Even though this seems like only half of the time our comparisons will be done, we know from mathematics that the BigO function must be asymptotically larger than our function, so our experimental value agrees with our results. Another algorithm we can analyze would be Bubble Sort. From the theoretical predictions we see that the BigO should be a case of O(n) for increasing lists and a O(n^2) for the average and worst case. Our experimental results do follow this mathematical correlation with the best case being a run time of 0.598ms for the 10^5inputs and 44540 and 49056 for the average and worst case.

   The **Radix** Runt Time is significantly lower than the three comparison based algorithms but similar to the **Shell Sort** run times. For the average case, we do see that the **Radix** perform better than the **Shell.**

2. (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 the case of small input size (10^2 and 10^3), the **Shell Sort and Radix Sort** seems to be the best algorithms for the least number of comparisons. The Shell Sort performs at a run time of milliseconds for small input

sizes and the Radix sort performs on the millisecond range also. This agrees with our analysis of the BigO for OO(n^2(logn)) for the average and worst cases for Shell Sort and O(n) for all the cases in Radix sort. When we begin to use larger data sets (10^4 and 10^5), we see that the **Radix Sort** has the smallest values in Run Time for our inputs. This agrees with our theoretical analysis that states the worst case should be O(n). The **factors** that could be affecting our experimental results could be the fact that our **Radix Sort** was implemented with operations on the byte level, and so would be faster to compute than other arithmetic operations regardless of the inherent efficiency of the algorithm. Other factors could be varied computational load on the servers at the times that these algorithms were tested, or even the uncertainty that identical processors are handling our algorithms, given that the server is something of a black box. One last factor could be the fact that we ran most of the comparisons on a laptop and not TAMU's Unix servers. These discrepaciencs gave us different runtimes due to the hardware integrated into the seperate machines.