# Project Cover Page

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

1. Oneal Abdulrahim, onealabdul@tamu.edu

2.Garret Sifuentez, gsifuentez8867@tamu.edu

3.Adalberto Gonzalez, aj20g@tamu.edu

Please write how each member of the group participated in the project. (All members worked on the report)

1. Oneal - Bubble Sort / Radix Sort

2.Garret - Selection Sort / Shell Sort

3.Adalberto Gonzalez - Insertion Sort & Testing

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 | Lab TA's |
| Web Material (give URL) | bigocheatsheet.com<br>cplusplus.com |
| Printed Material | Textbook for sort references and strategies |
| Other Sources | Lecture Slides<br>Listed in source code |

I certify that I have listed all the sources that I used to develop solutions to the submitted project report and code.
Your signature Oneal M. Abdulrahim, 23 September 2016

I certify that I have listed all the sources that I used to develop a solution to the submitted project and code.
Your signature Adalberto Gonzalez, 23 September 2016

I certify that I have listed all the sources that I used to develop solution to the submitted project and code.
Your signature Garret Sifuentez, 23 September 2016

## Description and Purpose

In this programming assignment, we were tasked as a group to implement various integer sorting algorithms. By doing this, we gain a better understanding of time complexities for common sorts in computer science, as well as study and compare number of comparisons and general efficiency of each sort. For this assignment, a "base" had already been provided for us by the teaching assistants. This base allows the program to be run from the command line, using the following format (this is also explained in the source files):

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

Where the flags denote:

- -h Display 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, and
- -c Display number of element comparisons (excluding radix sort)

## C++ Features, Generic Programming, and General Approach

Splitting each search algorithm into their own class allows for a cleaner and easier to read program overall, and for all of them to inherit from one parent class. The sort class included methods to run each sort by feeding it input and receiving output. This way, each class or sort can be a subclass of sort which only implements one major method - the sort itself. More generally, we used a simple method and C++'s random library to generate random numbers each run, if required. To keep track of time, the time header was included in the teaching assistant's base discussed above. Then, using the user selection by the flags passed in from the command line, the proper output was created.

## Algorithms

The **Selection Sort** divides list into two sub-parts and inserts the smallest value from the unsorted sub-part into the sorted sub-part one element at a time. This sort must iterate through the entire array twice over in the best case.
The **Insertion Sort** chooses first element in unsorted region of list and inserts it into its location in the sorted subsection of the array. This sort is most similar to how a human would sort a deck of cards.
The **Bubble Sort** (arguably the worst sort in our set) swaps consecutive elements based on their value, until the list is sorted. This allows the higher values to act as a "bubble" and eventually flow to the higher values of the array.
The **Shell Sort** swaps elements between a certain gap, and consecutively decreases that gap until the list is sorted. This sort implements a more general version of selection or insertion sort, but uses the gaps to split the sorting to that it does not iterate through the entire array over again.
The **Radix Sort** sorts integers by grouping them together by the same significant position and value. It is the only non-comparative sort in this assignment. Since no comparisons are made, this sort should, in theory, be most efficient.

## Theoretical Analysis

Below is a table of expected time complexities (in Big-O notation). For the right table, some sorts have no "worst" case, such as in the case of Selection Sort, since at minimum, it must perform $n^2$ operations. Therefore, every case is its best, average, and worst case.

| **Complexity** | best | average | worst |
|---|---|---|---|
| Selection Sort | $\Omega(n^2)$ | $\Theta(n^2)$ | $O(n^2)$ |
| Insertion Sort | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ |
| Bubble Sort | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ |
| Shell Sort | $\Omega(nlog(n))$ | $\Theta(n(log(n))^2)$ | $O(n(log(n))^2)$ |
| Radix Sort | $\Omega(nk)$ | $\Theta(nk)$ | $O(nk)$ |

| **Complexity** | inc | ran | dec |
|---|---|---|---|
| Selection Sort | average | average | average |
| Insertion Sort | best | average | average |
| Bubble Sort | best | average | worst |
| Shell Sort | best | average | average |
| Radix Sort | average | average | average |

## Experiments

In this phase, we ran each sort with the same array for every different input size *n*and every increasing, random, and decreasing order of integers. Our implementation generated a new array for every run, so although increasing and decreasing input stayed constant, the random arrays were created every time.

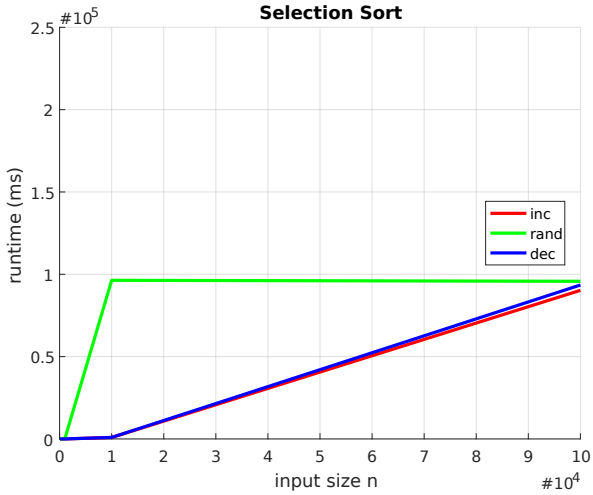
| **RT** | Selection Sort | | | Insertion Sort | | | Bubble Sort | | |
|---|---|---|---|---|---|---|---|---|---|
| *n* | inc | ran | dec | inc | ran | dec | inc | ran | dec |
| 100 | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms |
| $10^3$ | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms | 10 ms | 20 ms |
| $10^4$ | 890 ms | 96340 ms | 910 ms | 10 ms | 47220 ms | 890 ms | 0 ms | 212400 ms | 2300 ms |
| $10^5$ | 90250 ms | 95750 ms | 93490 ms | 0 ms | 47700 ms | 91260 ms | 0 ms | 216560 ms | 232120 ms |

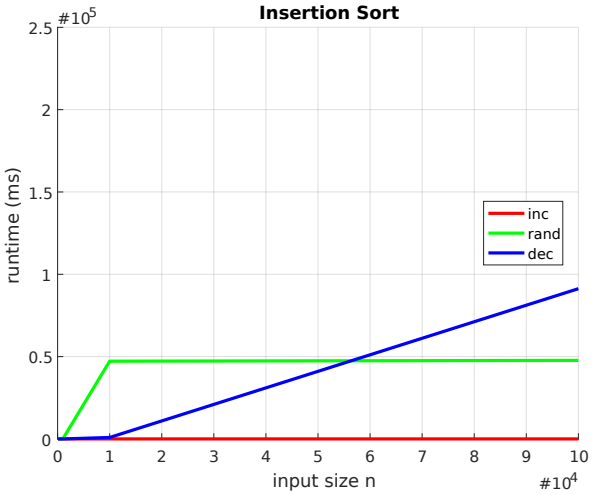
| **RT** | Shell Sort | | | Radix Sort | | |
|---|---|---|---|---|---|---|
| *n* | inc | ran | dec | inc | ran | dec |
| 100 | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms |
| $10^3$ | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms | 0 ms |
| $10^4$ | 0 ms | 180 ms | 10 ms | 0 ms | 100 ms | 10 ms |
| $10^5$ | 40 ms | 180 ms | 70 ms | 100 ms | 100 ms | 130 ms |

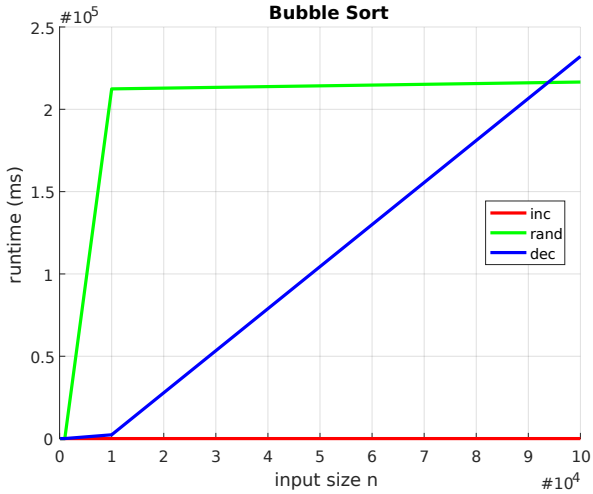
| **#COMP** | Selection Sort | | | Insertion Sort | | |
|---|---|---|---|---|---|---|
| *n* | inc | ran | dec | inc | ran | dec |
| 100 | 4950 | 4950 | 4950 | 100 | 100 | 100 |
| $10^3$ | 499500 | 499500 | 499500 | 1000 | 1000 | 1000 |
| $10^4$ | 49995000 | 704982704 | 49995000 | 10000 | 100000 | 10000 |
| $10^5$ | 704982704 | 704982704 | 704982704 | 100000 | 100000 | 100000 |

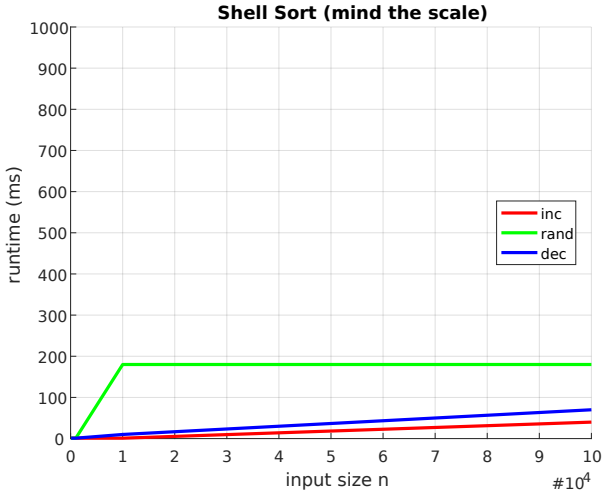
| **#COMP** | Bubble Sort | | | Shell Sort | | |
|---|---|---|---|---|---|---|
| *n* | inc | ran | dec | inc | ran | dec |
| 100 | 99 | 8968 | 10098 | 503 | 783 | 668 |
| $10^3$ | 999 | 987487 | 1000998 | 8006 | 15155 | 11716 |
| $10^4$ | 9999 | 1422253253 | 100009998 | 120005 | 4140224 | 172578 |
| $10^5$ | 99999 | 1386144801 | 1410165406 | 1500006 | 4207186 | 2244585 |

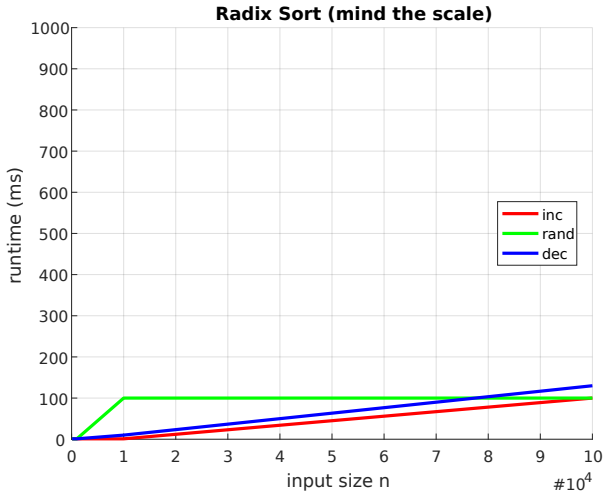
# Plots

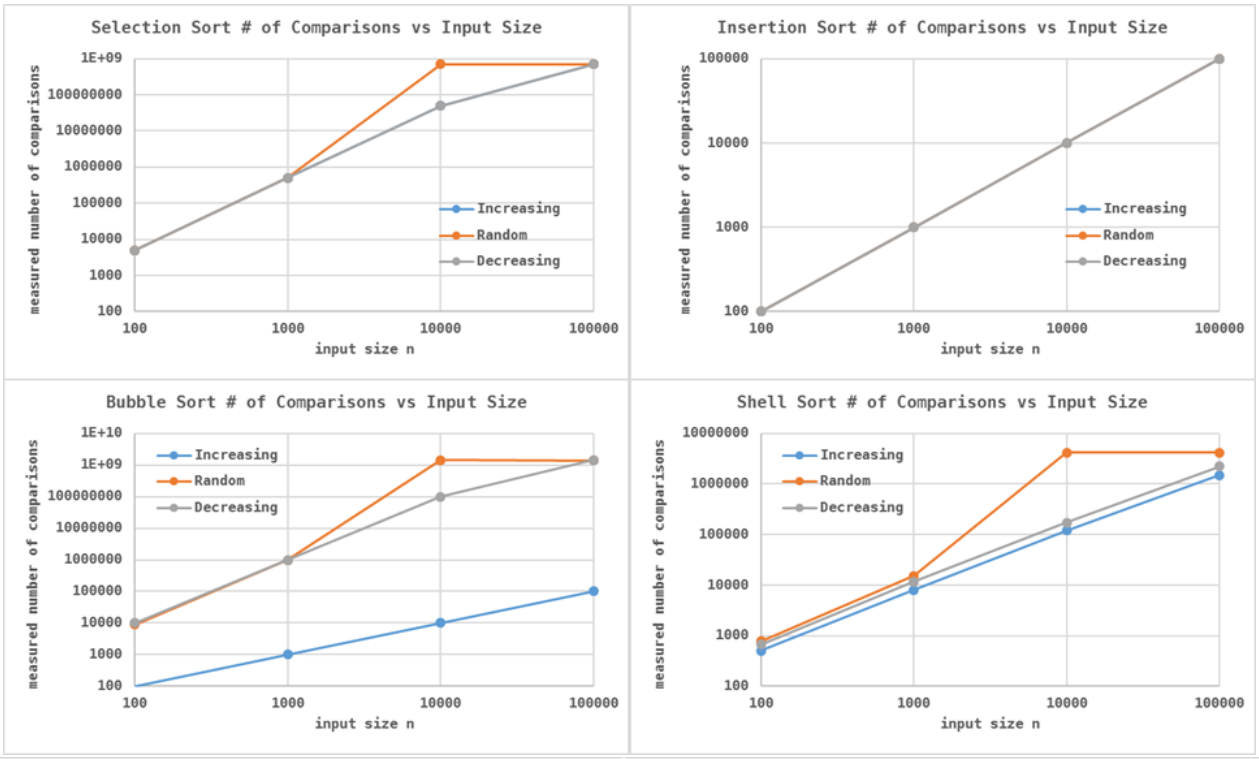

Selection Sort
#10^5
runtime (ms)
input size n
#10^4
inc
rand
dec

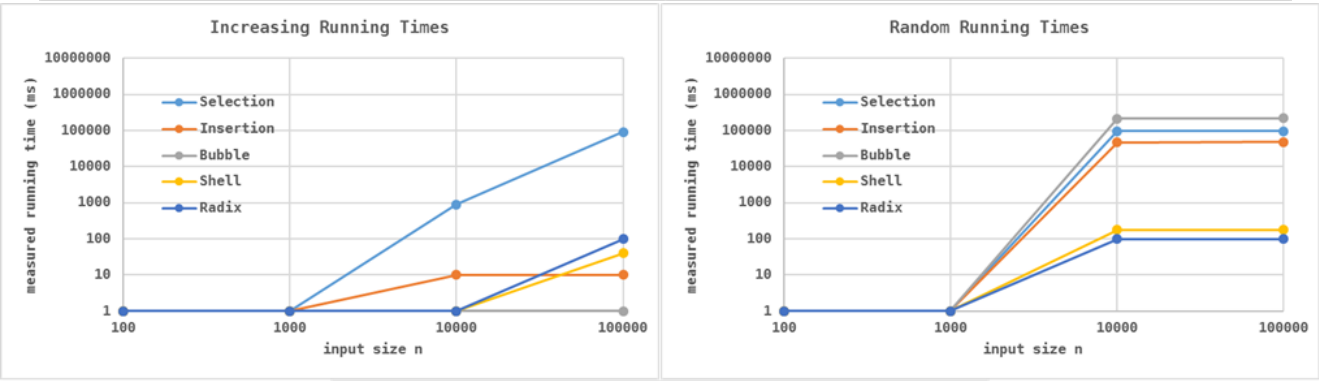

Insertion Sort
#10^5
runtime (ms)
input size n
#10^4
inc
rand
dec

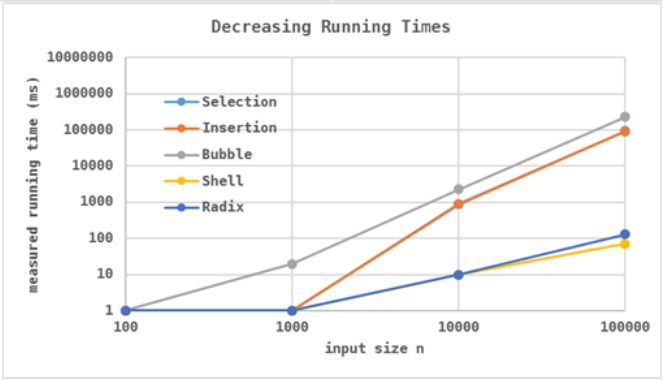

Bubble Sort
#10^5
runtime (ms)
input size n
#10^4
inc
rand
dec


Shell Sort (mind the scale)
runtime (ms)
input size n
#10^4
inc
rand
dec


Radix Sort (mind the scale)
runtime (ms)
input size n
#10^4
inc
rand
dec




## Discussion

All sorts had reasonable and expected running times as well as number of comparisons. For Bubble Sort especially, $n^2$ when $n = 10^5$ was a number within the order of $10^{10}$, which is to be expected. For smaller input sizes, the accuracy of the running time stayed around 0, which, although expected to be low, may have been imprecise. This extremely fast running time could be attributed to our compile flag, -Ofast using GCC. The only comparison based search algorithm to come close to radix in terms of running time was shell sort, with most of the other sorts taking much longer to sort the same input list.

## Conclusions

In general, it seems that for an array of integers, Radix sort performs the best, regardless of input size or order of integers. The predicted results from the earlier discussion agree with our experimental results. A worse sort than Bubble sort is Bogo sort, and we would certainly crash our Unix server if we tried to run that with $10^5$ input size. For other sorts, we observed the general behavior as input size increased. Again, our compiler, compiling strategies and flags, hardware, and implementation of code affect the results. We believe these results are sufficient for the purposes of this assignment, and demonstrate clearly enough the differences between the sorts given.