Project Cover Page

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

1. Fawcett, davidgfawcett@gmail.com

2. Bevolo, justinbevolo@tamu.edu

3. Zeng, zhiyangzeng@tamu.edu

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

1. David Fawcett: Took the lead role in converting the graphs and tables into a .tex file, as well as formatting the report. David was in charge of calculating the running times and comparisons of the Shell Sort and Bubble Sort. He wrote the code for the Random number generator, as well as code for both the Shell Sort and Bubble Sort. David made the graphs for the Bubble Sort Comparisons, Shell Sort Comparisons, Bubble Sort Running Time, Shell Sort Running Time, and the Increasing, Decreasing, and Random Running times. David made the tables for the Bubble Sort and Shell Sort comparisons and running times. On the report David included the writing for the: Introduction, Object Oriented Focus, Algorithms, and the information about the Experiments.

2. Justin Bevolo: Was in charge of calculating the comparisons and running times for the Selection Sort and Insertion Sort. He wrote code for the Insertion Sort and Selection Sort. He made the graphs for the Selection Comparisons vs Input, Insertion Comparisons vs Input, Selection Sort Running Time, and Insertion Sort Running Time. He made the tables for the time complexity, and the Insertion and Selection Sort comparisons and running times. On the report Justin included the writing for the: Theoretical Analysis, Discussion, and Conclusion.

3. Derek Zeng: Was in charge of calculating the comparisons and running times for the Radix Sort. He wrote the code for the Radix Sort, and also made it to where the Radix Sort code worked for both negative and positive numbers. He made the graphs for: Radix Sort Running Time, and the table for the Radix Sort running time.

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	None outside our group
Web material (give URL)	Data Structures and Algorithms in C++ pages: 110, 153-177, 259, 335, and 529
Printed Material	Linear Sort Algorithms and Introduction to Analysis of Algorithms PowerPoint slides
Other Sources	$http//bigocheatsheet.com/$, $http//www.algolist.net/Algorithms/Sorting/Selection_sort$, $http//www.sorting - algorithms.com/$

I certify I have listed all the source that I used to develop solutions to the submitted project report and code.
Your signature: Typed Name: David Fawcett Date: 9/18/15

I certify I have listed all the source that I used to develop solutions to the submitted project report and code.
Your signature: Typed Name: Justin Bevolo Date: 9/18/15

I certify I have listed all the source that I used to develop solutions to the submitted project report and code.
Your signature: Typed Name: Derek Zeng Date: 9/18/15

1. Introduction:

The purpose of this assignment was to implement several sorting algorithms, selection sort, insertion sort, bubble sort, shell sort, and radix sort, and analyze their running time complexity .Their running time complexity was investigated using theoretical analysis and experimentation. To run the sorting algorithms, use the make command in the terminal and run the ./sort file. You can either input numbers from the command line or from a file and you can output information about the performance of the algorithm and the sorted list of numbers to the terminal or to a file.

2. Object oriented focus:

Our program makes heavy use of C++ object oriented features in its organization. Each sorting algorithm is its own class which inherits from the sort class. The sort class is declared in the sort.h file. The sort class has one member, num_cmp, which is used in each sorting function to keep track of the number of comparisons preformed. It also has a virtual function called sort() which takes in an array of integers and the size of the array. This virtual function sort() is overridden by each sorting class and defined according to how that algorithm sorts numbers. Each of these classes are defined in .cpp files. The sort class contains several other functions defined in sort.cpp. This file also contains a main function to begin the program. This hierarchy tremendously reduces the amount of code by allowing the reuse of many functions throughout the program.

3. Algorithms:

Selection Sort:

Selection sort begins by finding the smallest element between the first index and the end of the sequence. This element is exchanged with the element at the first index. (Note: if they are the same then the element is merely assigned to itself). At this point the beginning of the list is guaranteed to have the smallest element. The process is repeated with the index being increased each time until the second to last index is reach and the sequence is sorted.

Insertion Sort:

Insertion sort starts with the second element of the sequence as compares it to the element preceding it in the sequence. If it is smaller, then the elements are ?switched?. (The element being ?inserted? is not inserted until the final step so the elements aren?t technically ?switched.?) This process continues until the front of the array is reached or the element is not smaller. Thus the first part of the sequence is always sorted and the following element is ?inserted? into the proper position of the sequence.

Bubble Sort:

Bubble Sort sorts a sequence of elements by making a series of passes through the elements. The elements are examined from the first index to the last and each element is compared to its neighbor. If the preceding element is larger that the succeeding one the elements are switched. In the first pass the last element is guaranteed to be the largest element. Therefore, subsequent passes do not need to proceed to the end of the array. In fact, the i^{th} pass only needs to compare the first $n - i$ positions. This process is repeated until no elements are swapped during a pass, thus ensuring the sequence is sorted.

Shell Sort:

The array being sorted in a shell sort algorithm is divided into segments whose elements are a certain distance apart. The array is thus divided into sub arrays, the number of which are equal to the distance separating the elements. These partitions are then sorted using insertion sort and the elements are returned to same position in the array as the elements they replaced in the sub array. The distance is reduced until its' value reaches one and insertion sort is applied to the entire array.

Radix Sort:

Radix sort is a non-comparison based sorting technique, which involves the use of counting sort. Counting sort first finds the number of elements less than or equal to each element being sorted. The element is then placed in the at that index in the output array. Thus two additional arrays are used: an array to hold the value of the number of elements less or equal to the elements and an output array. Radix reduces the size of the first additional array by sorting large numbers one place at a time. It sorts the least significant digit first and repeats the process based upon the number of digits in the largest number.

4. Theoretical Analysis:

Complexity	Best	Average	Worst
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$
Shell	$O(n)$	$O((nlog(n))^2)$	$O((nlog(n))^2)$
Radix	$O(n)$	$O(nk)$	$O(n^2)$

Complexity	Increasing	Random	Decreasing
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$
Shell	$O(n)$	$O((nlog(n))^2)$	$O((nlog(n))^2)$
Radix	$O(nk)$	$O(nk)$	$O(nk)$

For the Insertion Sort the best case scenario is when the sequence is already sorted in increasing order. With an already sorted sequence the sort will perform (n-1) comparisons. In this scenario the Insertion Sort has a linear running time, O(n). The worst case scenario is when the sequence is sorted in decreasing order. With a sequence in reverse order the Insertion Sort has to compare and swap every term, which requires (n(n-1))/2 comparisons. The time complexity of the Insertion Sort worst case scenario is $O(n^2)$. When considering the time complexity of the Insertion Sort average case it is necessary to determine what defines the average case. When you have an unsorted sequence there is a greater chance that the array is in a random order than already sorted, so the time complexity is $O(n^2)$.

During a Selection Sort the function will always make (n(n-1))/2 comparisons no matter the order of the elements in the sequence. This means that the best, average, and worst case scenarios of Selection Sort are all $O(n^2)$.

Bubble Sort has a best case running time when the elements are sorted in increasing order. In this case the bubble sort makes (n-1) comparisons and has a O(n) time complexity. The worst case of a Bubble Sort occurs when the elements of the sequence are in decreasing order. In this scenario the sort makes (n(n-1))/2 comparisons which gives the Bubble Sort a $O(n^2)$ running time in the worst case. On the average case the sequence is in a random order, which has a time complexity of $O(n^2)$.

Shell Sort in the best case, when the array is already sorted, has a time complexity of O(n), and similarly to Bubble and Insertion Sort, the average and worst case scenarios have the same time complexity. In Shell Sort the average and worst case time complexities are both $O((nlog(n))^2)$.

Unlike the previous 4 sorting algorithms, Radix sort is not comparison based. Radix Sort considers each digit of the numbers being sorted and forms a stable sort based on each digit. This is a linear sorting algorithm so in the best case it has a time complexity of O(n). The average time complexity of a Radix Sort is O(nk) where k represents the number of digits an a number. The worst case time complexity results when k is not a constant, so O(nk) becomes $O(n^2)$.

5. Experiments:

To test the theoretical analysis of the algorithms we calculated the number of comparisons and the average time each algorithm took when sorting 100, 10^3, 10^4 and 10^5 elements that were in increasing, decreasing, and random order. The results of the experiment are presented in the tables and graphs below.

a.

Comp	Insertion Sort			Selection Sort		
n	inc	dec	rand	inc	dec	rand
100	99	4950	2732	4950	4950	4950
10^3	999	499500	250321	499500	499500	499500
10^4	9999	49995000	24855826	49995000	49995000	49995000
10^5	99999	4999950000	2499996319	4999950000	4999950000	4999950000

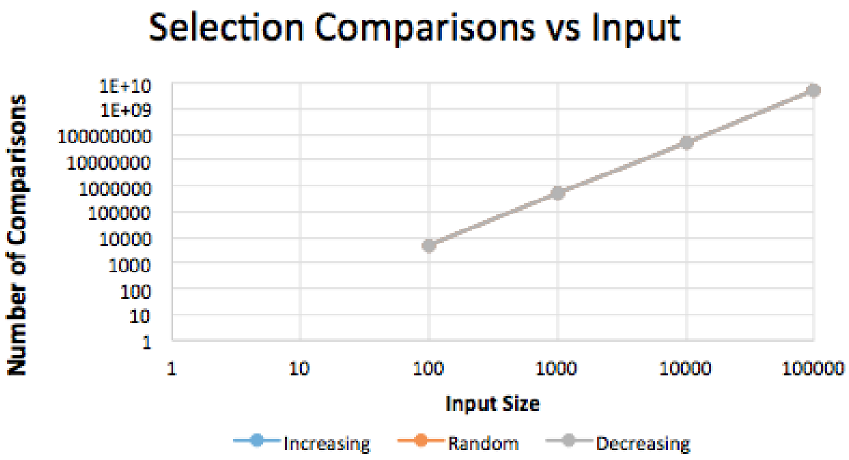
Comp	Bubble Sort			Shell Sort		
n	inc	dec	rand	inc	dec	rand
100	99	4950	4859	413	564	739
10^3	999	499500	499347	7090	9620	13035
10^4	9999	49995000	49993569	100844	137004	197198
10^5	99999	4999950000	4999809285	1308346	177315	2558725

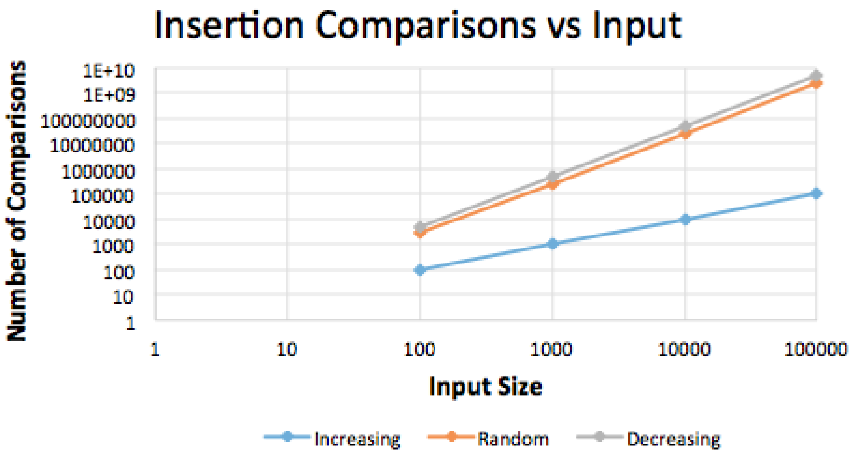
RT	Insertion Sort			Selection Sort		
n	inc	dec	rand	inc	dec	rand
100	0.00001	0.000527	0.000279	0.000306666	0.0003102	0.000318
10^3	0.000089	0.052634	0.026983	0.0301117	0.0300906	0.030124
10^4	0.000891	5.49271	2.75369	3.22915	3.22498	3.23996
10^5	0.008925	553.242	277.016	326.248	326.242	325.893

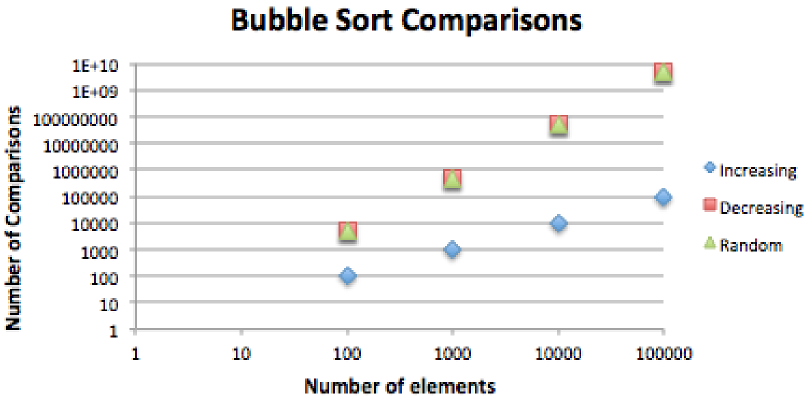
RT	Bubble Sort			Shell Sort		
n	inc	dec	rand	inc	dec	rand
100	0.0000003	0.0000353	0.000319	0.000003	0.0000046	0.0000073
10^3	0.000003	0.0026492	0.0027594	0.0000455	0.0000683	0.0001909
10^4	0.0000221	0.246681	0.288777	0.0006594	0.0008043	0.0024028
10^5	0.000219	25.0339	32.7316	0.0065728	0.0096712	0.0291891

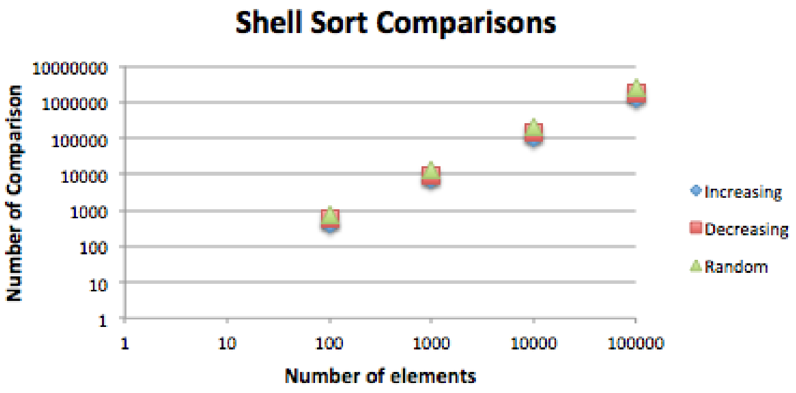
RT	Radix Sort		
n	inc	dec	rand
100	0.000214	0.00017	0.000182
10^3	0.000828	0.000831	0.000836
10^4	0.006392	0.00633	0.006664
10^5	0.062499	0.062357	0.06351

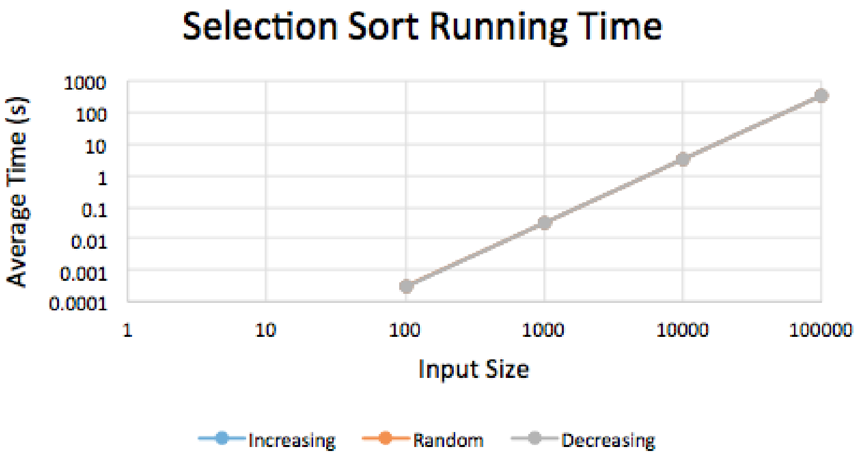
b.

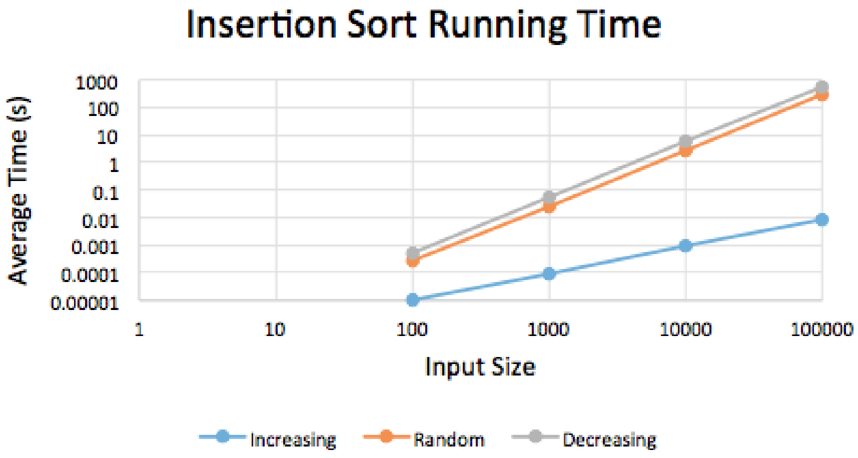

Selection Comparisons vs Input
Number of Comparisons
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
Input Size
Increasing
Random
Decreasing

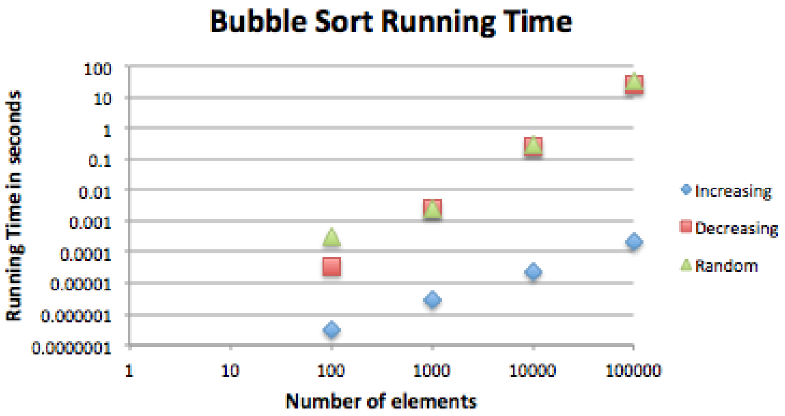

Insertion Comparisons vs Input
Number of Comparisons
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
Input Size
Increasing
Random
Decreasing

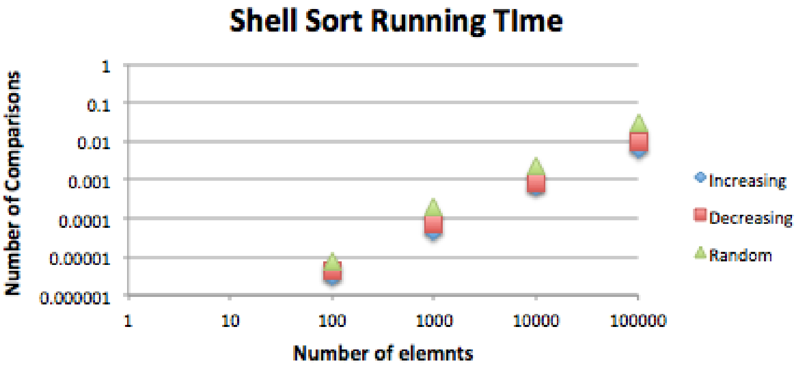

Bubble Sort Comparisons
Number of Comparisons
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
Number of elements
Increasing
Decreasing
Random

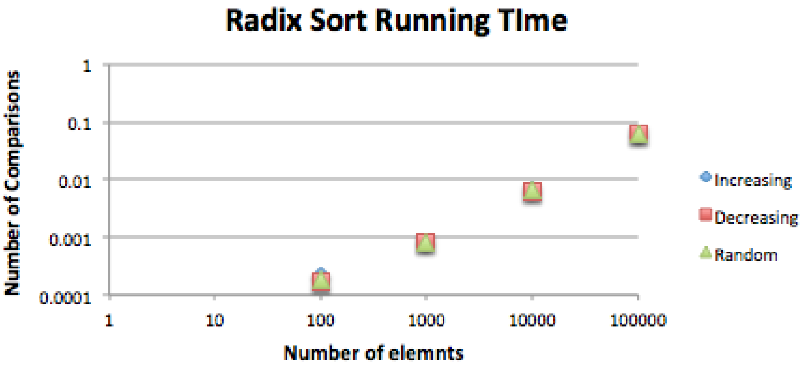

Shell Sort Comparisons
Number of Comparison
10000000
1000000
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
Number of elements
Increasing
Decreasing
Random


Selection Sort Running Time
Average Time (s)
1000
100
10
1
0.1
0.01
0.001
0.0001
1
10
100
1000
10000
100000
Input Size
Increasing
Random
Decreasing

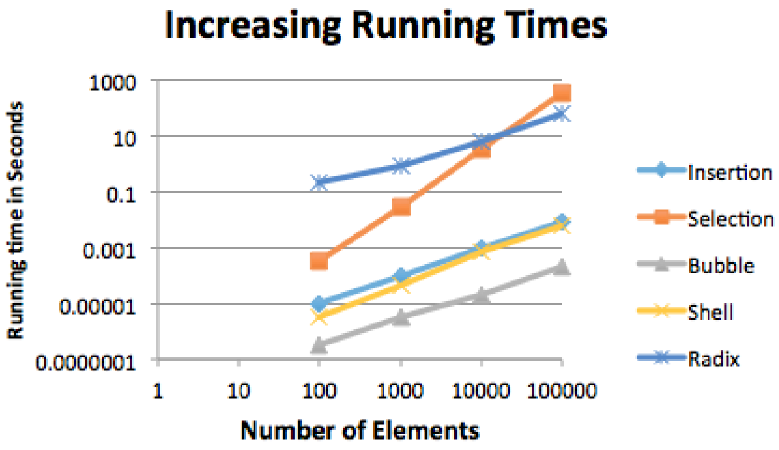

Insertion Sort Running Time
Average Time (s)
1000
100
10
1
0.1
0.01
0.001
0.0001
0.00001
1
10
100
1000
10000
100000
Input Size
Increasing
Random
Decreasing

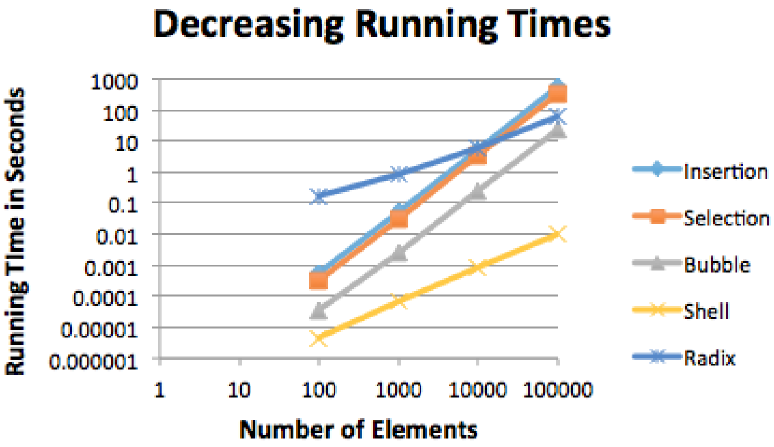

Bubble Sort Running Time

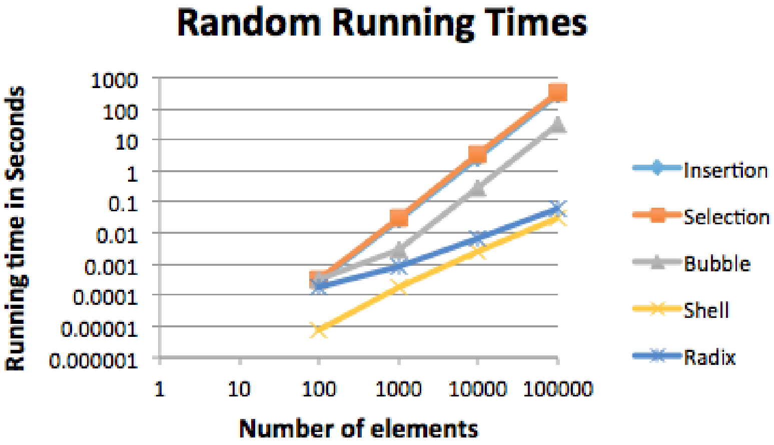

Shell Sort Running TIme


Radix Sort Running TIme

c.


Increasing Running Times
Running time in Seconds
1000
10
0.1
0.001
0.00001
0.0000001
1
10
100
1000
10000
100000
Number of Elements
Insertion
Selection
Bubble
Shell
Radix


Decreasing Running Times
Running TIme in Seconds
1000
100
10
1
0.1
0.01
0.001
0.0001
0.00001
0.000001
1
10
100
1000
10000
100000
Number of Elements
Insertion
Selection
Bubble
Shell
Radix


Random Running Times
Running time in Seconds
1000
100
10
1
0.1
0.01
0.001
0.0001
0.00001
0.000001
1
10
100
1000
10000
100000
Number of elements
Insertion
Selection
Bubble
Shell
Radix

6. Discussion:

Our experimental results line up very accurately with our theoretical analysis. The running times and comparisons for Bubble Sort, Insertion Sort, and Selection Sort line up exactly to how they were projected in the theoretical analysis. Shell Sort ran as intended during the best case scenario, but because of Shell Sort?s technique of separating the sequence into subsequences it is difficult to get consistent average and worst case results. Radix Sort?s running time, compared to the four comparison based algorithms, is significantly faster over the random, increasing, and decreasing tests. During the tests, because k was kept as a constant, the Radix Sort performs the sort in $O(n)$ time regardless of the order the numbers are in before the sort. Our computational results and the information we learned from the text and from the slides in class all line up.

7. Conclusion:

From this project it is clear that it is important to research the different types of algorithms before choosing one for a large assignment. For example if it was necessary to choose a sorting algorithm that sorts $100,000$ elements $10,000,000$ times choosing an algorithm with a $O(n^2)$ time complexity would not be practical. If you were presented with a set of 100000 elements that are already sorted then it would be practical to use any sorting algorithm except for Selection Sort. If you were presented with a large number of elements in backwards order, where k is a constant, then it would be practical to use a Radix Sort. It is clear that it is necessary to consider input sizes and theoretical sequence orders when choosing algorithms for a project. There is one small factor that can affect experimental results of this assignment: the sorts were divided and tested on three separate computers, which could have a slight effect on the running time of each test.