Project Cover Page

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

1. Tim Zook

2. Alan Achtenberg

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

1. Coding Shell and Radix sort, doing theoretical analysis

2. Coding Insertion, Selection, and Bubble sort, doing experimental analysis

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: Tim Zook Date: 9/20/2013

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: Alan Achtenberg Date: 9/20/2013

CSCE 221 Programming Assignment 2

Programs due September 22nd by 11:59pm
Reports due on the first lab of the week of September 22nd

1 Introduction

In this assignment, we implement five sorting algorithms: selection sort, insertion sort, bubble sort, shell sort and radix sort in C++. We tested our code using varied input cases, timed the implementation of sorts, recorded the number of comparisons performed in the sorts, and compared these computational results with the running time of implemented algorithms using Big-O asymptotic notation. We also analyze these results to compare them to the theoretical performance of the algorithms that we have learned from our book.

You can build the program using the

```
make
```

command in the code folder. You can then view the instructions for running the program by typing

```
./sort -h
```

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
```

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
```

2 Program Structure

The program uses polymorphism and inheretence to simplify the problem of having multiple different algorithms that all solve the same problem. It defines an abstract SORT class which serves as an interface for implementing each of the algorithms, then instantiates a concrete subclass which represents the actual sorting algorithm at runtime based on the command line input. This allows the program to call and measure all of the sorting algorithms in the same way without having different code paths for the different algorithms.

3 Algorithms

Selection sort is the most conceptually simple algorithm of the five. It's method is to loop through the array n times, looking first for the smallest element, then the next smallest element, then then next, etc. Due to the fact that it does not reduce the number of iterations if the list is already sorted this sort performs very poorly when the list is already sorted.

Bubble sort is another relatively simple algorithm that works by looping through the array until it is sorted, and each time it "bubbles" all the elements closer to their correct positions by comparing each element with the next element in the list and swapping them if the first is larger.

Insertion sort is similar to Bubble sort, except that instead of sorting the elements over multiple passes it sorts everything in one pass. It can do this because instead of "bubbling" each element once per pass like bubble sort it moves an element all the way to it's proper position as soon as it finds it. Thus the name of the algorithm, because in a way it "inserts" each element where it should go one by one. Note that this differs from selection sort because selection sort trys to find a particular element for a given slot, whereas insertion sort trys to find a particular spot for a given element.

Shell sort is basically Insertion sort repeated several times, where each time it uses a different "gap" between elements that it compares. Normal selection sort has a "gap" of one, where it always compares adjacent elements, whereas shell sort uses logarithmically scaling gap sizes one after another. So for a gap size of, for example, 4, it would compare elements 4 spaces apart. This generally makes it much faster than Insertion sort because it can move elements large distances around the list without having to compare each element in between.

Radix sort is a non-comparison based sort which allows it to run in linear time in all cases. It sorts the numbers digit by digit using counting sort, which is a somewhat complex algorithm based on using the number to be sorted as the index of a second array in order to count up how many of each number there is and then summing up all numbers that are lower than that number, and using this information to rearrange the final array. As a downside it's runtime also scales with the range of the numbers in the array, as it needs to run through the final range array in order to sum the numbers up. However it is still very fast in almost all real-world situations because of how well it scales with the size of input compared to the comparison based algorithms.

4 Theoretical Analysis

Selection sort is the most obviously slow algorithm out of all of them since it has an $O(n^2)$ best case wheras all the others have an $O(n)$ or $O(nlog(n))$ best case. In fact it doesn't really even have a "best case" or "worst case" in general because it always does the same amount of comparisons (although not always the same amount of swaps), and does not adapt to patterns in the data. It's a very poor choice to use for any real-world sorting application.

Insertion sort and Bubble sort are very similar from an analysis point of view, because they both perform the optimal $O(n)$ number of comparisons if the data is sorted, but have to do extra compares and swaps if the data is not sorted which makes their average and worst case complexities $O(n^2)$. In both of these sorting algorithms the best case is when the data is already sorted so the algorithm need only take one linear loop through the data to verify it's sortedness, whereas the worst case is when the data is reverse-sorted where every element will need to be swapped all the way to the opposite end of the array.

Shell sort is difficult to properly classify because it's time complexity can vary quite a bit based on what "gap function" you use to select the gaps used. In our implementation we used the original $\lfloor n/2^k \rfloor$ gap function used by Donald Shell in his original implementation, which is not the most optimal gap function to use but one of the easiest to implement. The average case for Shell sort is particularly hard to determine and very few formal proofs for average complexity exist for any gap functions. Much of the most significant work relating to Shell sort, such as the paper "Best Increments for the Average Case of Shellsort" by Ciura in 2001 is done experimentally rather than mathematically. The best case for our implementation was $O(nlog(n))$, but this can be increased to $O(n)$ in some implementations by using a constant number of gaps, instead of a gap function that depends on the size of the array. Unlike other sorting methods the worst case for shell sort is actually when the even and odd elements are sorted in opposite directions. In this case it will need to iterate all the way down to the gap size 1 before it can do any sorting at all, making it $O(n^2)$ just like a single selection sort in this case. The best case is when the array is already sorted just like with the other algorithms.

Because Radix sort is not a comparison based sort, it is able to run in $O(n)$ time in the worst case. However it also scales with the range of the numbers in the array, but as long as the range is roughly proportional to the value of n it runs in linear time. Generally the constant multiplier on this $O(n)$ is much higher than the other algorithms because it needs to loop through the data several times, however in the end since it does not scale exponentially it can be much faster than other algorithms for very large data sets that can be represented as unsigned integers. Radix sort also does not have a general "best case" or "worst case" scenario similar to Selection sort because it's complexity is $O(n)$ in all cases.

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	$O(nlog(n))$	$O(n^2)$	$O(n^2)$
Radix Sort	$O(n)$	$O(n)$	$O(n)$

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	$O(nlog(n))$	$O(n^2)$	$O(n^2)$
Radix Sort	$O(n)$	$O(n)$	$O(n)$

5 Experiments.

We verified the correctness of our sorting algorithms and measured their performance characteristics by running many tests with different types of input data. The input data varied in how many elements there were, and how they were sorted, resulting in 12 total tests for each sorting algorithm. We varied the size in order to see how each algorithm's running time and comparisons changed based on the value of n, and we varied the sorting method in order to see the worst, best, and average cases for each algorithm.

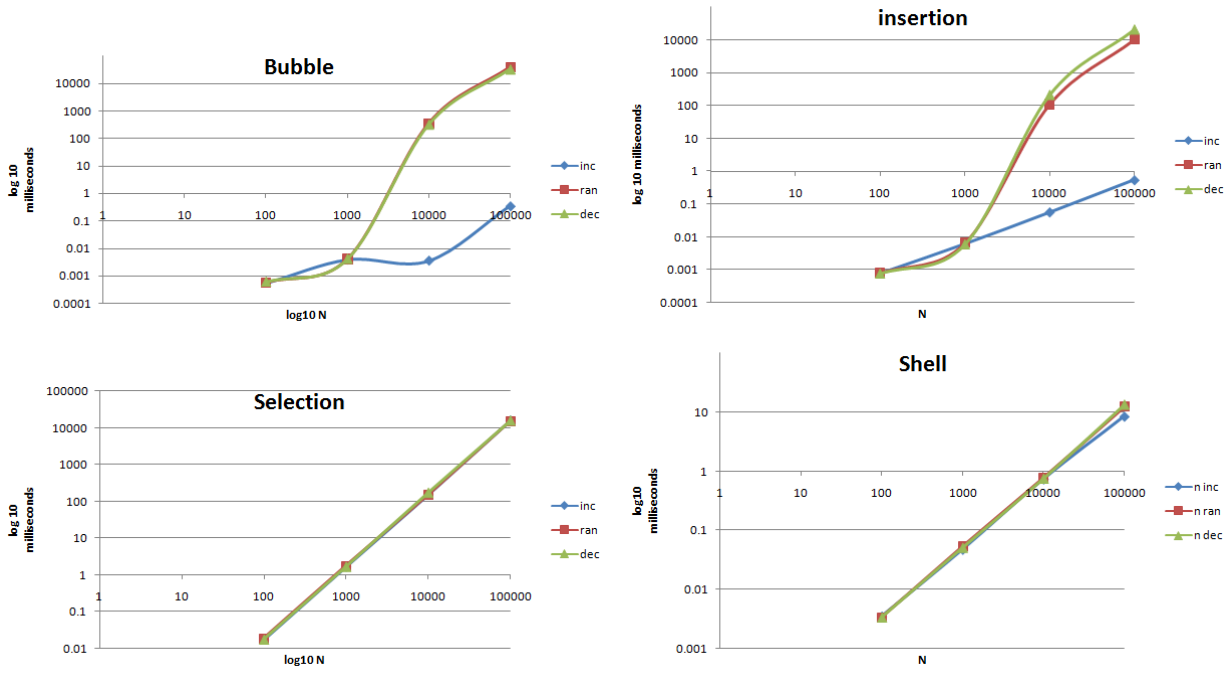
RT	Selection Sort			Insertion Sort			Bubble Sort			Shell Sort		
n	inc	ran	dec	inc	ran	dec	inc	ran	dec	inc	ran	dec
100	.017	.0185	.0177	.0008	.0008	.0007	.00056	.0005	.0006	.00350	.0033	.0034
10^3	1.641	1.746	1.66	.0063	.0065	.006	.0039	.0041	.0042	.0472	.053	.051
10^4	150	150	170	.0568	110	210	.00358	360	330	.733	.774	.753
10^5	15460	15140	15350	.55	10300	20310	0.348	39010	33710	8.46	12.2	13.3

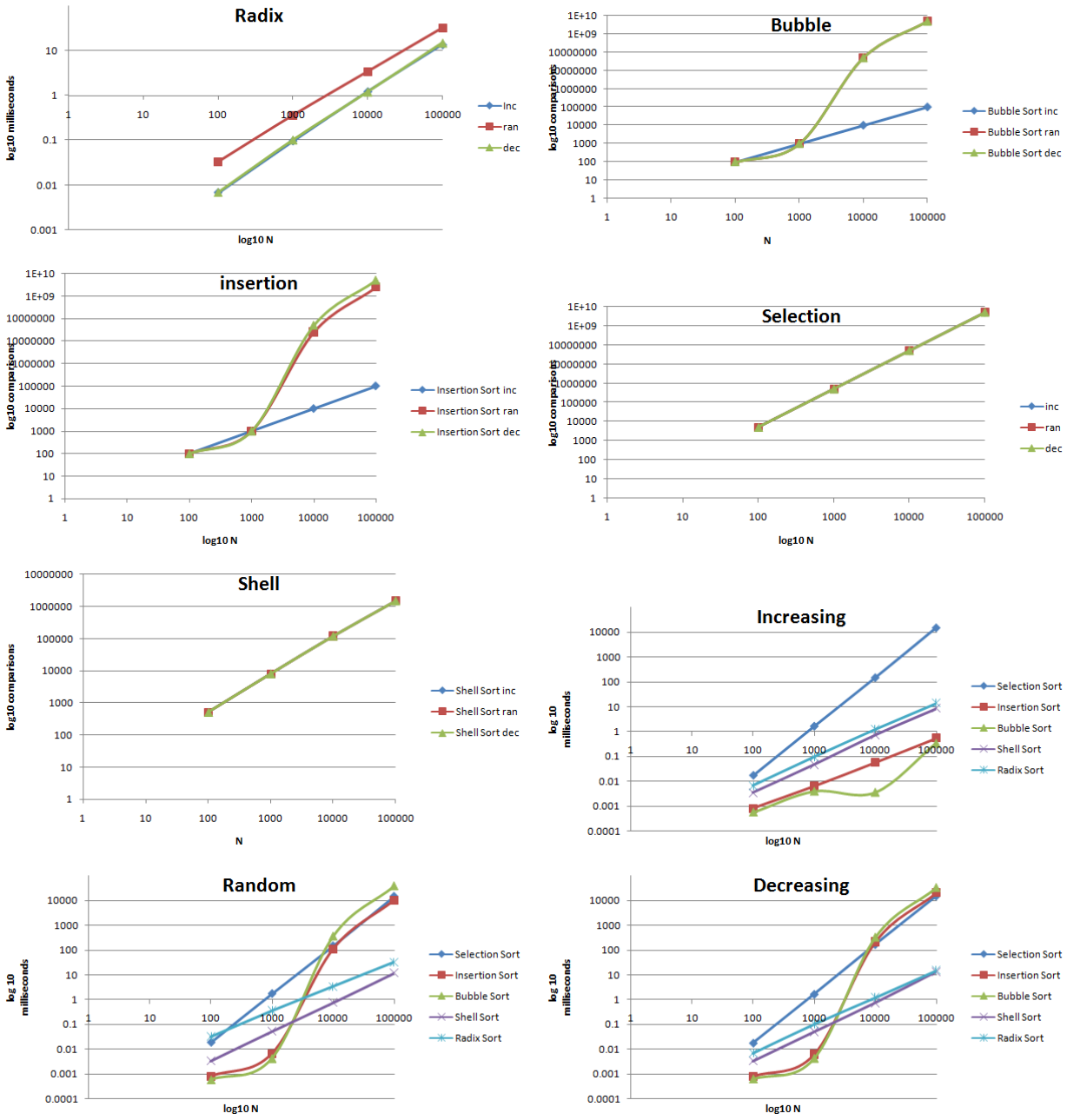
Radix Sort		
inc	ran	dec
.006	.032	.006
.096	.361	.102
1.23	3.43	1.20
13.7	32.5	15

#COMP	Selection Sort			Insertion Sort			Bubble Sort		
n	inc	ran	dec	inc	ran	dec	inc	ran	dec
100	4950	4950	4950	99	99	99	99	99	99
10^3	499500	499500	499500	999	999	999	999	999	999
10^4	49995000	49995000	49995000	9999	25044713	49995000	9999	49971995	49995000
10^5	4.9×10^9	4.9×10^9	4.9×10^9	99999	4.5×10^9	4.9×10^9	99999	4.9×10^9	4.9×10^9

Shell Sort		
inc	ran	dec
503	503	503
8006	8006	8006
120005	120005	120005
1500006	1500006	1500006

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





6 Discussion and Conclusions

We can see that for the most part, our theoretical analysis matched up very well with our actual results although there are a few suprises. For the best case (increasing) scenarios, insertion and bubble sort run the fastest, because they only have to do one simple pass through the data to determine it's sortedness. Shell and insertion sort are next, in Shell sort due to it's $O(nlog(n))$ nature and radix sort due to the fact that it has to loop through an array many times even though it's performance overall scales in a linear $O(n)$ fashion. Then as we predicted Selection sort performs very poorly with $O(n^2)$ complexity.

The worst case and average case results (random and decreasing) are so similar as to be virtually identical, at least from an analytical point of view. In both cases we can see predictably that Selection, Insertion, and Bubble sort scale very similarly with an $O(n^2)$ complexity. However it is interesting to note that for $n \leq 1000$ Bubble sort and Insertion sort are actually faster than Radix and Shell sort even in the worst case scenario. This shows how important it is to remember when working with Big-Oh notation that just because one algorithm has a better asymptotic complexity it doesn't nessesarly have better absolute running time for all practical values of n in your application.

Another interesting thing to note is that for almost all tests that we ran, the running time for Radix sort and Shell sort were very close. There may be implementation details such as memory caching patterns at work here or it may be that our tests simply did not use big enough values of n for the difference between $O(n)$ and $O(nlog(n))$ to really become apparent. With both Radix and Shell sort we can see that they follow a roughly $O(n)$ pattern, where when n increases by 10 the running time increases by roughly 10, where for the other algorithms during the worst cases when the n increases by 10 their running time increases by 100 or more due to their $O(n^2)$ pattern.

While in general our experimental results matched up reasonably well with our theoretical analysis, our results were a good demonstration of why we put so much weight on the theoretical analysis in computer science. When done on an actual computer our results can have many inaccuracies and variances due to many very complex factors (for example, on our increasing input bubble sort data, the running time for $n = 10^3$ is inexplicably longer than for $n = 10^4$). When doing theoretical analysis we can get much more usable and provable conclusions about our algorithms because they are not affected by variances in the computer or differences between different computers.

Overall we can see that Selection sort is a very poor choice of sorting algorithm in almost any situation. Insertion sort and Bubble sort are both very competative sorting algorithms if n is always relatively small, or if the list is usually close to being already sorted. Shell sort and Radix sort are both very comptetative for very large values of n, as they scale much better than the other algorithms, especially if the data is not already close to being sorted. Therefore we can see that as in many cases in computer science, it is important to fully understand the problem you are trying to solve when you chose an algorithm: there is no one perfect tool for all cases, and you should take care to choose the algorithm that best fits your needs.