Assignment 2 Report

February 15, 2013

Purpose The purpose of the assignment was to analyze various sorting algorithms. The program implements a number of sorting algorithms: selection, insertion, bubble, shell, and radix. These are used to sort a list of integers and output the result to either the screen or a file. There are options available to output the sorted and unsorted data, number of comparisons, and running time, as well as a method for specifying the algorithm to use. To run the program on Unix, one must first change the current directory to that which contains the source files by using the `cd` command. Then, type `make` to build the program. Once this is done, typing `./sort -h` will describe the various arguments to pass in. For the input, the first line contains a number representing the number of elements in the list, followed by a list of integers, all separated by carriage returns.

C++ Constructs This program uses one main class called `Sort` that handles all of the sorting operations, and another called `Option` that handles the various arguments that can be passed in by the user. It starts by initializing an `Option` object, and then uses the information passed in to `Option` to determine which actions to perform. For instance, if a user types `./sort -a R -f input.txt -p -c -t`, the program will sort the list given in input.txt using radix sort and print the sorted list, number of comparisons, and running time. All sorting operations are done within the `Sort` class, including counting the number of comparisons.

Algorithms Selection sorts divide the input list into two sublists, the sorted and nonsorted sections. The sorted sublist starts empty and is filled from "left to right." The smallest element is found in the unsorted part and added to the end of the sorted sublist until there are no more elements in the unsorted sublist. Insertion sort selects the current element of the list and compares it to each element until it finds one that is smaller. Then the insertion sort selects the new element and moves it back in the list until it finds a smaller element and places it at the proper index. The insertion sort is ineffiecient for large lists. A bubble sort selects 2 sequential elements in the list and compares them, if the second one is smaller it trades their indexes, it then compares the next two elements in the form index: 1 and 2, then 2 and 3 etc. The shell sort is one

of the more efficient comparison sorts. It is an in place comparison sort. The shell sort performs an insertion sort for every h-th element within a gap size of h. The runtime of a shell sort is dependent upon its gap size. It exchanges elements that are far apart to begin with and steadily sorts closer and closer elements. It is the fastest search of the sorts we tested where n is large. Also it's worst case run time is random order as opposed to decending order like the other comparison sorts we tested. The last sort in this experiment is the radix sort. The radix sort is different because it is a non comparison sort. It does not compare the elements to other elements in the list. The radix sort uses the least significant digit of every element to sort it into different "buckets." The buckets are then reinserted back into the list in order of increasing significant digits. The process is repeated for each digit of growing significance until it reaches the most significant digit where after sorting the list is now completely sorted in increasing order. The radix sort is very efficient for lists of large size.

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	$O(n)$	Between$O(n(log_2 n)^2)$ and $O(n^{\frac{3}{2}})$	Depends on gap sequence, best known is: $O(n$
Radix Sort	$O(n)$	$O(kn)$	$O(kn)$

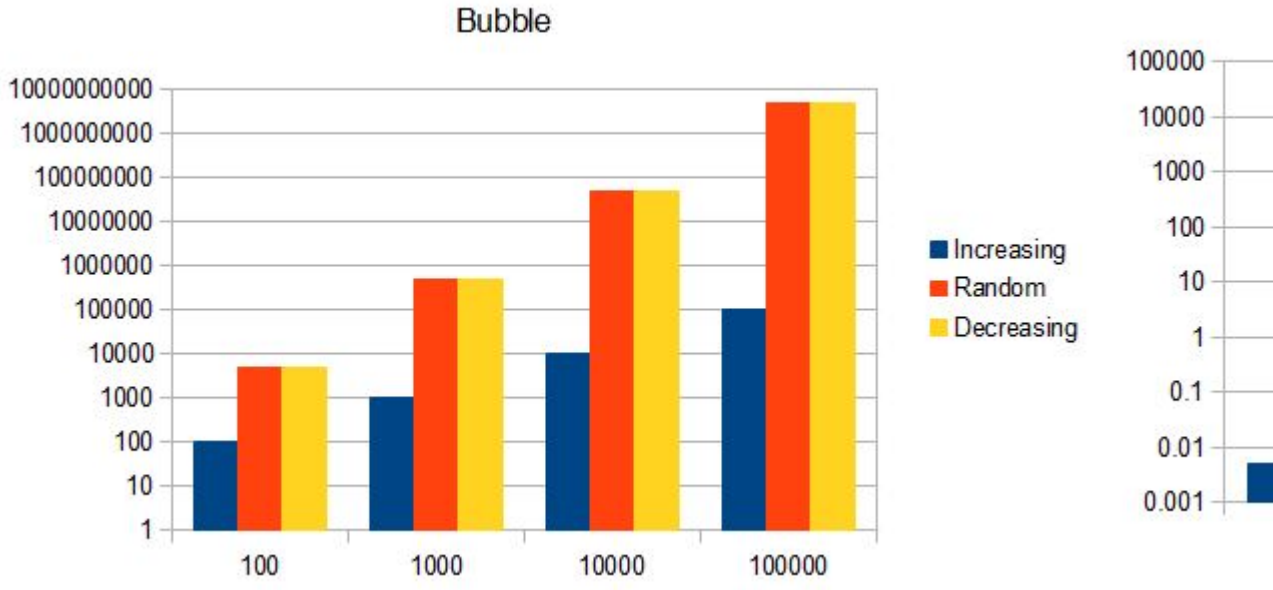
Complexity	inc	ran	dec
Selection Sort	Best:$O(n^2)$	Average:$O(n^2)$	Worst:$O(n^2)$
Insertion Sort	Best:$O(n)$	Average:$O(n^2)$	Worst:$O(n^2)$
Bubble Sort	Best:$O(n)$	Average:$O(n^2)$	Worst:$O(n^2)$
Shell Sort	Best:$O(n)$	Worst: Between$O(n(log_2 n)^2)$ and $O(n^{\frac{3}{2}})$	Average:$O(nlog^2 n)$
Radix Sort	Best:$O(n)$	Average:$O(kn)$	Worst:$O(kn)$

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

Experiments

RT	Selection Sort			Insertion Sort			Bubble	
n	inc	ran	dec	inc	ran	dec	inc	ran
100	0.043 ms	0.044 ms	0.04 ms	0.006 ms	0.029 ms	0.051 ms	0.005 ms	0.069 m
10^3	1.951 ms	2.839 ms	2.072 ms	0.012 ms	2.129 ms	3.861 ms	0.011 ms	5.953 m
10^4	204.833 ms	184.118 ms	192.724 ms	0.067 ms	133.849 ms	256.353 ms	0.061 ms	396.261 r
10^5	18860.2 ms	18780.1 ms	19655.2 ms	0.645 ms	12711.4 ms	26201.3 ms	0.407 ms	40669.1 r

#COMP	Selection Sort			Insertion Sort			Bub	
n	inc	ran	dec	inc	ran	dec	inc	ra
100	4950	4950	4950	99	2055	4950	99	494
10^3	499500	499500	499500	999	250351	499500	999	498
10^4	49995000	49995000	49995000	9999	25273883	49995000	9999	4997
10^5	4999950000	4999950000	4999950000	99999	2496355009	4999950000	99999	49999



Discussion Our experimental results were in line with the theoretical results. The comparison sorts were much slower than radix and shell sort for large values of n. Our Radix sort was reasonably comparable to Shell sort, but we also counted in the run time the time it took to determine the largest integer. This could have effected our run time all values of n as it adds a factor of into our formula so instead of the formula being Radix = n it was closer to Radix = n + n. This could have thrown off our data as shell sort was faster than radix over all ordered tests, but radix performed better in random ordered sets. The other comparison based algorithms, Bubble, Selection, and Insertion

were far outclassed by Shell and Radix. The comparison based algorithms were measured in seconds when n was 10^4. This compared to the times of Shell and Radix at 10^5 are miserable. Insertion and Bubble performed well when the set of numbers were already ordered but by that point, those algorithms running would be redundant. The single major discrepancy was the number of comparisons that the comparison sort functions made. Integer could not hold the iterator that kept up with the number of compariconsthat the functions made so we got overflow and strange numbers of comparisons. Once that was fixed our data lined up with theoretical numbers and the rest of the experiment went smoothly.

Conclusions The shell sort is the fastest of the comparison sorts we test. It out performed each the other sorts on every test. The most interesting thing about the shell sort is that it performs worse for sorts in random order than decending order. Because of this for real world uses a radix sort, the only non-comparison based sort we test, is more practical. The shell sort does out perform the radix sort in worst case senarios though. We saw that the radix sort seems to scale in near linear time, each input list that was 10 times larger produced a runtime that was close to 10 times longer. Our experitmental results all agree with our theoretical results on all but the 10^4 and10^5 runtimes for the bubble sort. The bubble sort took longer to sort the random order than the decending order which should be worst case scenario. Different things that would have affected the runtime of our sort algorithms was the actual hardware that we run them on. When running our experiments on the unix server some of the sorts, most notably bubble and selection sorts took several minutes while when run on the laptop we used to document all of our sorts the longest we waited for any sort to complete was 40 seconds.