Assignment 2 Report

February 19, 2017

Purpose The purpose of this assignment was to implement and analyze various sorting algorithms in order to sort lists of integers and to further our understanding of running-time functions of algorithms with respect to algorithm efficiency regarding asymptotic notation. The program constructed implements various sorting algorithms such as selection, bubble, insertion, shell, and radix sort which are used to sort a list of integers from either standard input or reading from an input text file and output the sorted integer data to either the screen or to a output text file recording the running time, number of comparisons, as well as specifying the algorithm used. The primary data structure used in this assignment is the array data structure, which in this case is a collection of fixed-size sequential elements of the integer data type in order to store, process, and sort our integer lists. In order to use the sorting program, navigate to the source directory via the cd command and type **make** or **make all** to build the program. After this done, typing **./sort –h** will execute the program and provide helpful user information regarding the user command line arguments and their descriptions and purpose. The other program, **gen**, was implemented as a helper program in order to generate the integer list input text files needed for testing the sorting algorithms in the main sort program. The text file generated by **gen** is organized by the first line containing the number of elements in the list, followed by each element value afterwards line-by-line. There are 12 input testing text files in the source directory, named by their respective power of 10, and then their order, **"i"** for increasing, **"d"** for decreasing, and **"r"** for random. For example, **10^2d.txt** means it has 100 elements and is in decreasing order. Other input files can be generated via the **gen** program. Navigate to the source directory, use this command to compile **c++ -std=c++11 gen.cpp -o gen** followed by **./gen** to execute the program, which will ask you to input a power of ten and input the order. The program will then create the respective input.txt file.

C++ Constructs This program utilizes two main classes called **Sort** which contains each of the sorting algorithm functions, and **Option**, which handles the various user-input arguments handled at the command-line. When a user inputs arguments in the command-line, an instance of the Option class is initialized and all the command line arguments are processed and the information is used to determine which processes are going to be handled by the program. For instance, if the user compiles and executes the program with the following command-line parameters: **./sort –a B –f input.txt –d –p –c –t** The program will process the arguments and sort the list found in the input.txt file using the bubble sort algorithm and will print both input unsorted list and the output sorted list, number of comparisons, and the running time of the sorting algorithm specified by the user. The actual sorting is done in the program by creating an instance of the parent Sort class and then by initializing an instance of the respective sorting algorithm's class. For example, after the command-line arguments are processed, an instance of the Sort class is initialized, and then an instance of the requested sorting algorithm class is initialized which calls the sort class member function to process the list of integers for sorting. There are elements of both inheritance and polymorphism in the program, as each individual child sorting class inherits both protected and public members from the parent class, Sort, such as the virtual Sort public function and the protected class member, num_cmps, which keeps track of the number of comparisons executed in runtime for each sorting algorithm.

Algorithms The program provides the implementation of four comparison-based sorting algorithms, Selection, Insertion, Bubble, Shell, and one non-comparison based sorting algorithm, Radix sort.

1. **Selection sort** – The selection sort starts at index zero and traverses the array comparing each value with the current index; if the value is smaller than the current index, the index is saved. After the array has been traversed and if a smaller value than the current index is found then a swap occurs between the two. The current index is then incremented and the algorithm reiterates until the array is sorted.

2. **Insertion sort** – The insertion sort selects the current element in the array and compares it to each element in the array until it finds a smaller value, inserting this smaller value back in the array until it finds a smaller element and places it at the correct index.

3. **Bubble sort** – The bubble sort selects two sequential elements in an array and compares them, swapping them to maintain order, moving on to the next two sequential elements, 1 and 2, then 2 and 3, then 3 and 4, etc., until the array is sorted.

4. **Shell sort** – The shell sort is one of the more efficient sorts compared to the other comparison-based sorts with the runtime depending on the gap size. The shell sort swaps elements that are far apart and then steadily sorts closer elements. Shell sort is efficient for sorting arrays where size (n) is large.

5. **Radix sort** – The radix sort is a non-comparison based sort and is different than the other four sorts because it does not compare elements against each other. The Radix sort is efficient compared to other sorts for lists of large size. The radix sort separates the least significant digit elements and sorts them into different ranges called "buckets". The buckets are then reinserted back into the array in order of increasing significant digits, the process repeats in increasing the significant digits where array becomes sorted in increasing order.

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

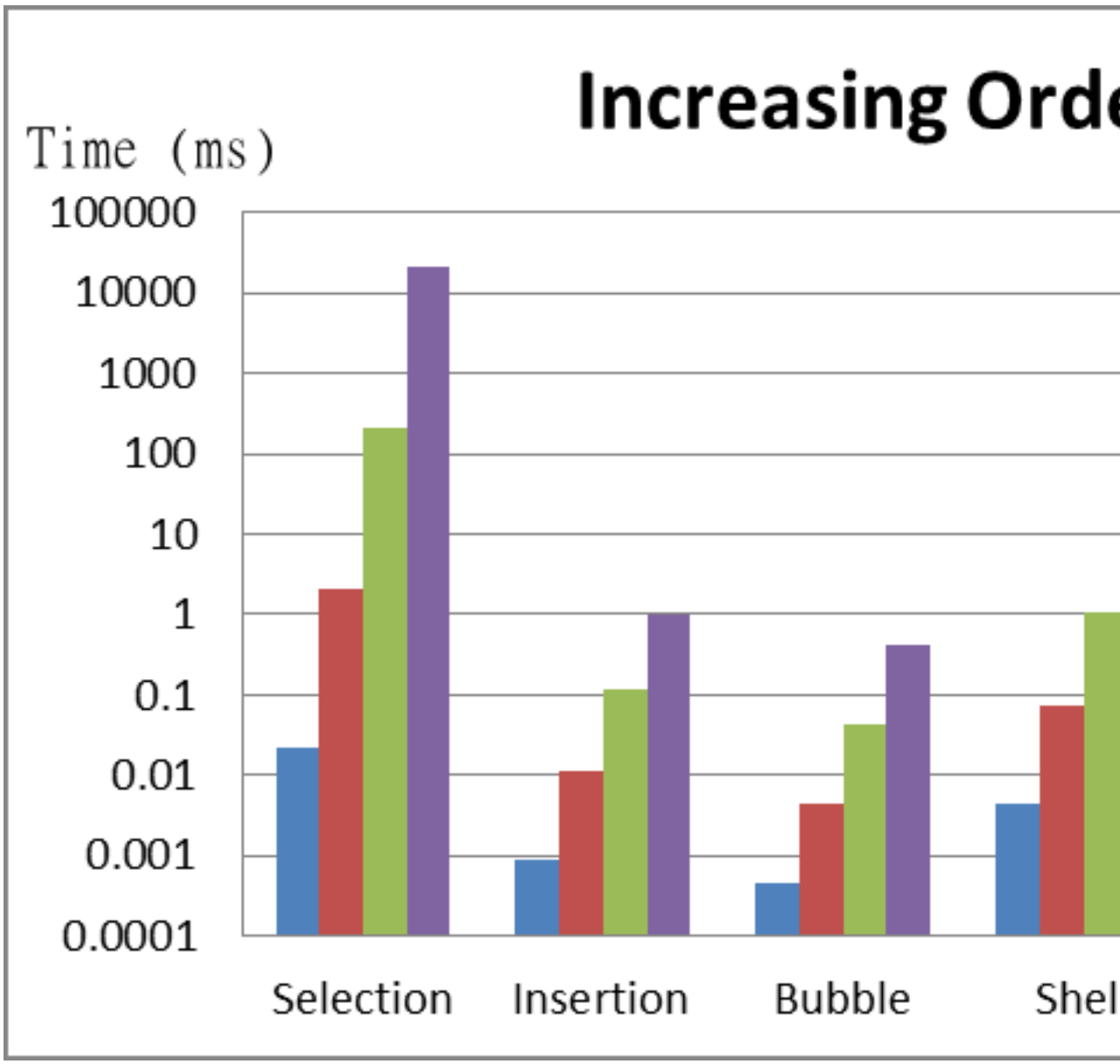
Experiments

RT	Selection Sort			Insertion Sort			Bubble	
n	inc	ran	dec	inc	ran	dec	inc	ran
100	0.022 ms	0.02207 ms	0.0222 ms	0.00092 ms	0.00093 ms	0.00088 ms	0.00045 ms	0.0004
10^3	2.089 ms	2.085 ms	2.091 ms	0.01132 ms	0.0099 ms	0.01014 ms	0.00442 ms	0.0044
10^4	210 ms	210 ms	220 ms	0.121 ms	0.131 ms	0.141 ms	0.04452 ms	0.0446
10^5	20990 ms	21010 ms	21970 ms	1.014 ms	2.488 ms	4.142 ms	0.407 ms	13950

#COMP	Selection Sort			Insertion Sort			Bub	
n	inc	ran	dec	inc	ran	dec	inc	ra
100	4950	4950	4950	99	2702	4950	99	46
10^3	499500	499500	499500	999	255504	499500	999	497
10^4	499500	49995000	49995000	9999	24882940	49995000	9999	4997
10^5	4999950000	4999950000	4999950000	99999	2501588846	4999950000	99999	49997


Selection Sort
increasing
random
decreasing
# of comparisons
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
100
1000
10000
100000
Size
increasing
100


Selection Sort
increasing
random
decreasing
100
1000
10000
100000


Increasing Orde
Time (ms)
100000
10000
1000
100
10
1
0.1
0.01
0.001
0.0001
Selection
Insertion
Bubble
Shel

Discussion The computational results produced in the experiment are similar to the theoretical results learned about in lecture. The Shell and Radix sort run times deemed to be much more efficient than the run times of the other comparison-based algorithms, and both shell and radix are also by far more efficient for sorting lists with large values of n compared to utilizing other comparison-based sorts. Bubble and Insertion sort seem to be only efficient when the size of the input is relatively small or the order is increasing, which would be the best case scenario. The number of comparisons per size (n) also match up with the asymptotic big-Oh notation for each sorting algorithm in the graph above. It is also worth noting the discrepancies with the random order for the radix sort, as the randomized seed used in the experiment generates and fills a list in random order with numbers of random length, and radix performs faster with numbers that have less significant digits, as d-passes are made for sorting d-digit numbers; this could deviate run times if the numbers generated were all small or all large for d-digits, the length of each integer generated is random.

Conclusion It seems from the experimental data produced that the shell sort is the fastest sort of the comparison-based sorts, which matches what was learned in lecture. Shell sort essientially out-performed every other sort in every case, with radix coming in as a close second as far as running time is considered. As far as comparisons go, bubble and insertion sort are efficient when n $\leq$100, or particularly in the increasing order case, but this occurance would be rare, as a list that is already sorted is the best case scenario. In the instance of comparing algorithm efficiency, hardware and system environments always have an effect on the running times of algorithms but overall the results produced from the experiment were indicative of the theoretical aspects of running times and asymptotic behavior learned in lecture.