605.202 DATA STRUCUTRES

GUAN YUE WANG

JHU ID: gwang39

LAB 4 ANALYSIS

AUGUST 16, 2017

General Program Design

The program conducts seven sorting methods on data sets with different sizes and natures in order to compare their relative performances. Sorting methods include insertion sort, heap sort, and five shell sorts with different increments based on various sequences such as Knuth's sequence, Marcin Ciura's sequence, and Hibbard sequence.

The overall program is controlled by a while loop to read input text file number by number. After running this while loop for the first round to figure out the file size, the program runs the loop again to take each number and add them to array so they can be sorted later. When seven data sets prepared and ready for seven sorting methods, the program runs those sorts one by one and records sort time. At the end of each round, the sorted results and run time are output to corresponding output file based on output path provided.

The program consists of one main class and three other classes for each of the insertion sort, heap sort, and shell sort. All sorts are constructed using iterative approaches plus array implementation if needed. The main entry point can be ran with or without command line arguments ([input file] [output file]). If the user choose not to do so, the program console would prompt user to input I/O file locations.

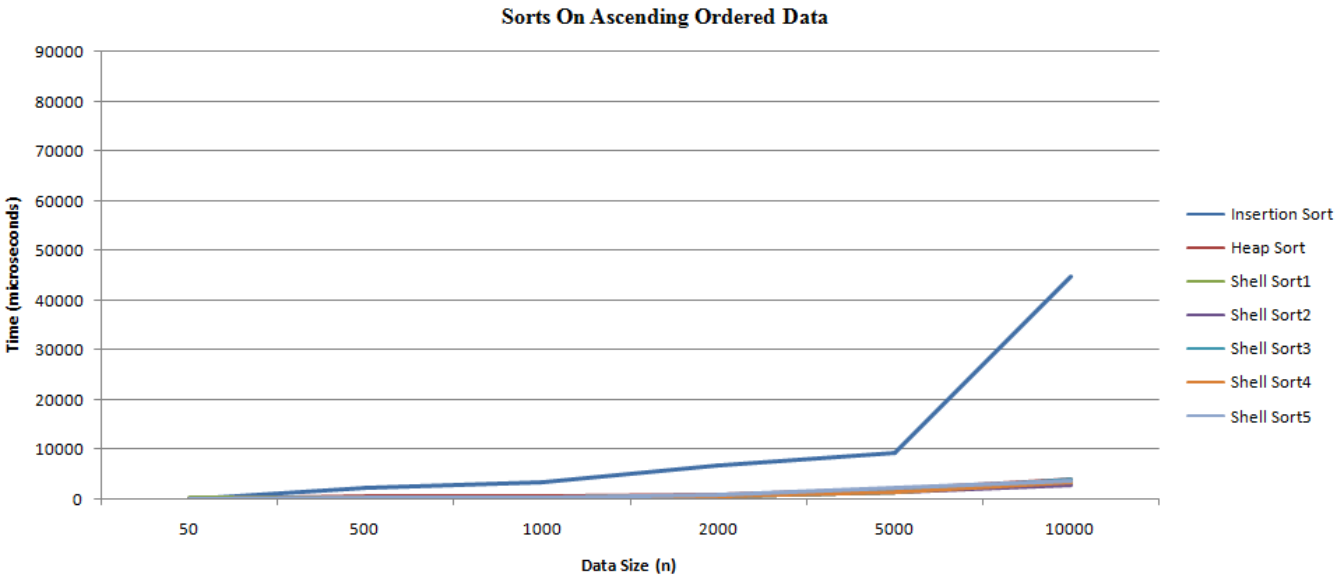
Data Structure Implementation And Appropriateness

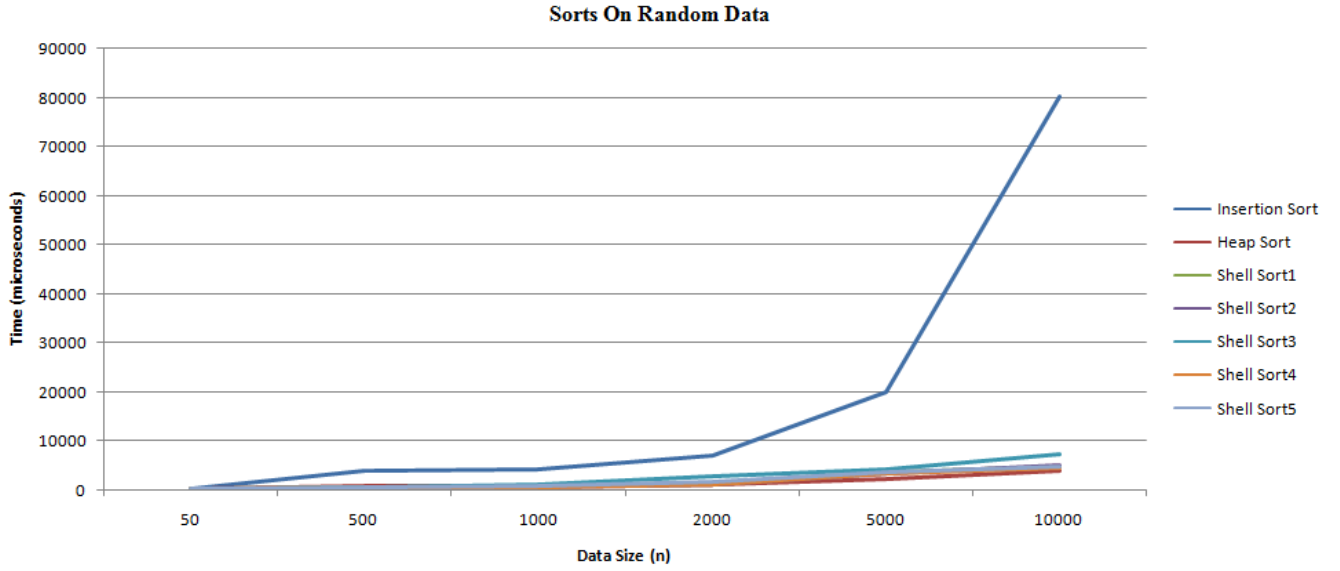
Array is used as the major data structure to store data and support sorting in the program. It is considered a good fit with our program based on several reasons. Firstly, the major part of our program is to use insertion sort or selection sort; therefore, array implementation gives us random access to the data and makes it easier for us to construct the sort and build iterative logics. To be specific, insertion sort requires iteration through the data and conduct comparison along the process whereas selection sort iteratively move data from unsorted region to sorted region so array implementation provides a better support for both algorithms. Furthermore, as we can easily figure out the size of the file, the memory allocation to the array can be accurately defined without any waste of resource.

Sorting Performances

Performance Overview

Overall, we can see heap sort and shell sorts are generally better in terms of efficiency. The only case insertion sort can be better than the rest is when data size is small which can be observed in our test case n = 50. This actually makes sense as insertion has a best case n but average and worse case n^2 whereas heap sort has a consistent complexity nlogn and shell sort has complexity ranging from nlogn (best case) to n(logn)^2 (average and worse case). In terms of space complexity, all sorts has memory requirement 1 as all of them involve in-place comparison so there are all equal. Tables and Graphs below shows overview of the performance of all sorts with respect to different data size and data nature (ascending, random, reversed).


Sorts On Ascending Ordered Data
Time (microseconds)
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
50
500
1000
2000
5000
10000
Data Size (n)
Insertion Sort
Heap Sort
Shell Sort1
Shell Sort2
Shell Sort3
Shell Sort4
Shell Sort5


Sorts On Random Data
Time (microseconds)
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
50
500
1000
2000
5000
10000
Data Size (n)
Insertion Sort
Heap Sort
Shell Sort1
Shell Sort2
Shell Sort3
Shell Sort4
Shell Sort5

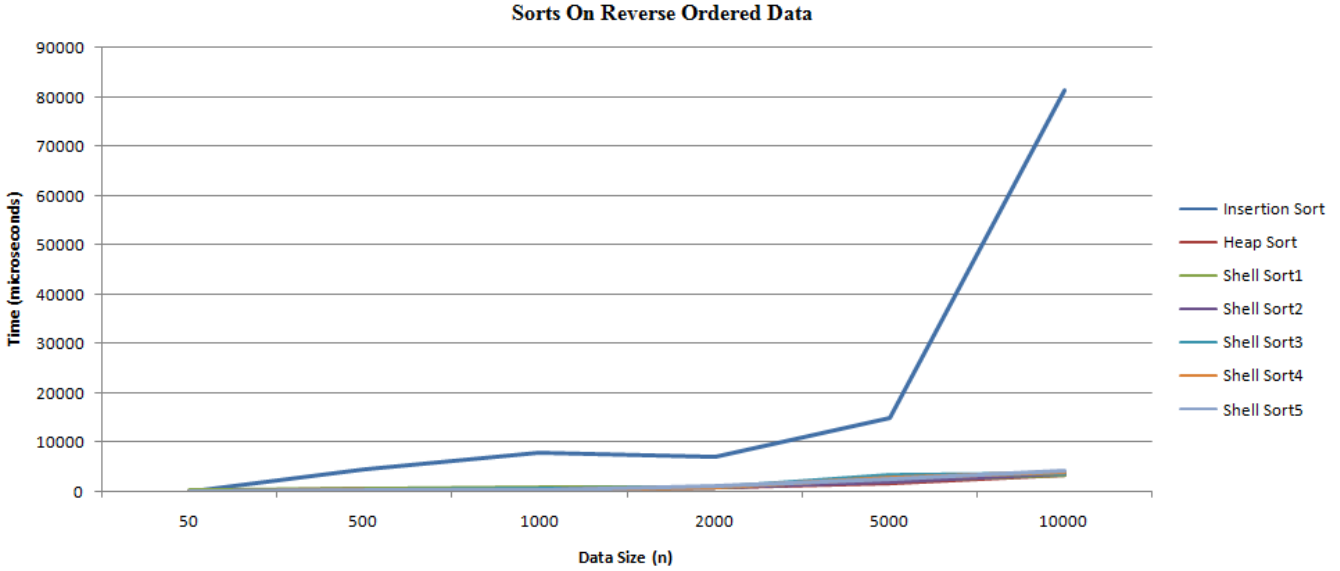
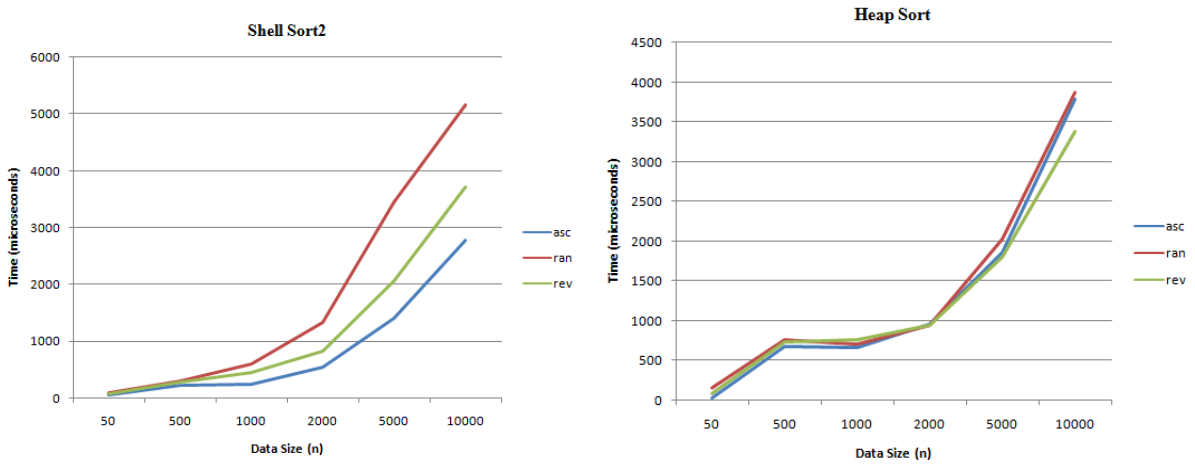

Sorts On Reverse Ordered Data
Time (microseconds)
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
50
500
1000
2000
5000
10000
Data Size (n)
Insertion Sort
Heap Sort
Shell Sort1
Shell Sort2
Shell Sort3
Shell Sort4
Shell Sort5

Chart - Sorting Time On Different Data (time measured in microseconds)

n	data	Insertion Sort	Heap Sort	Shell Sort1	Shell Sort2	Shell Sort3	Shell Sort4	Shell Sort5
50	asc	27	22	132	61	72	87	82
50	ran	129	154	155	90	87	103	118
50	rev	50	82	121	65	78	73	114
500	asc	2141	675	283	236	202	253	251
500	ran	3735	758	371	291	516	338	323
500	rev	4475	735	349	277	273	276	286
1000	asc	3454	653	326	254	268	336	355
1000	ran	4155	704	653	591	1111	579	630
1000	rev	7811	760	626	444	529	420	501
2000	asc	6731	951	572	551	540	618	868
2000	ran	6852	950	1332	1327	2974	1350	1477
2000	rev	6983	949	804	818	955	866	1094
5000	asc	9284	1864	1411	1421	1410	1411	2098
5000	ran	19909	2028	3572	3460	4303	3570	3574
5000	rev	14868	1805	2078	2061	3308	2735	2654
10000	asc	44628	3791	3387	2787	3828	3314	3715
10000	ran	80169	3877	4804	5166	7276	4550	4774
10000	rev	81284	3383	3304	3715	3726	3827	4214

Heap Sort vs. Shell Sort

As we can see insertion sort is much worse in terms of performance when n gets larger despite the nature of the data, we will focus our performance comparison and discussion around heap sort and shell sort.



From graph above, we can see heap sort provides a more consistent performance across all different kinds of data whereas shell sort shows a difference in efficiency in handling different type of data. To be

specific, while heap sort has almost the same performance on ascending, random, and descending data, Shell sort works the best on ascending data, then reverse ordered data, and performs the worst on random data. This actually aligns with the big O we learnt in class where heap sort always give nlogn performance whereas shell sort's performance can range from nlogn (best case) to n(logn)^2 (average and worst case). Based on the time we recorded, we can see heap sort performs better when the data is random; otherwise, shell sort provides a better run time on ascending and reversed data.

Sequence Used In Shell Sort

Based on project instruction and my search results, five different increments are used for shell sort as below

Shell Sort1- Knuth's sequence

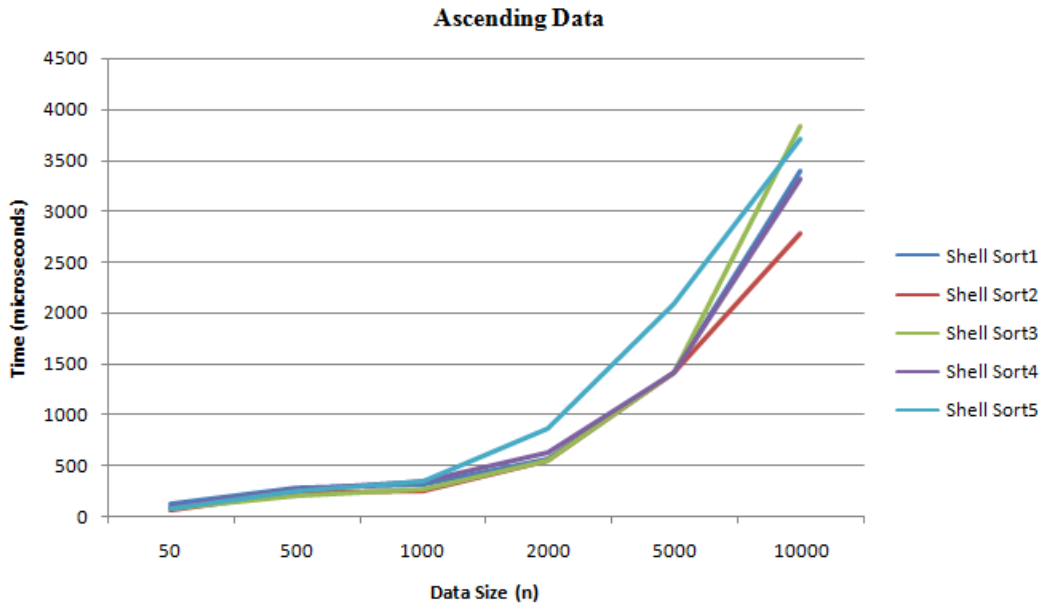
Shell Sort2 - Knuth's sequence to the nearest prime number

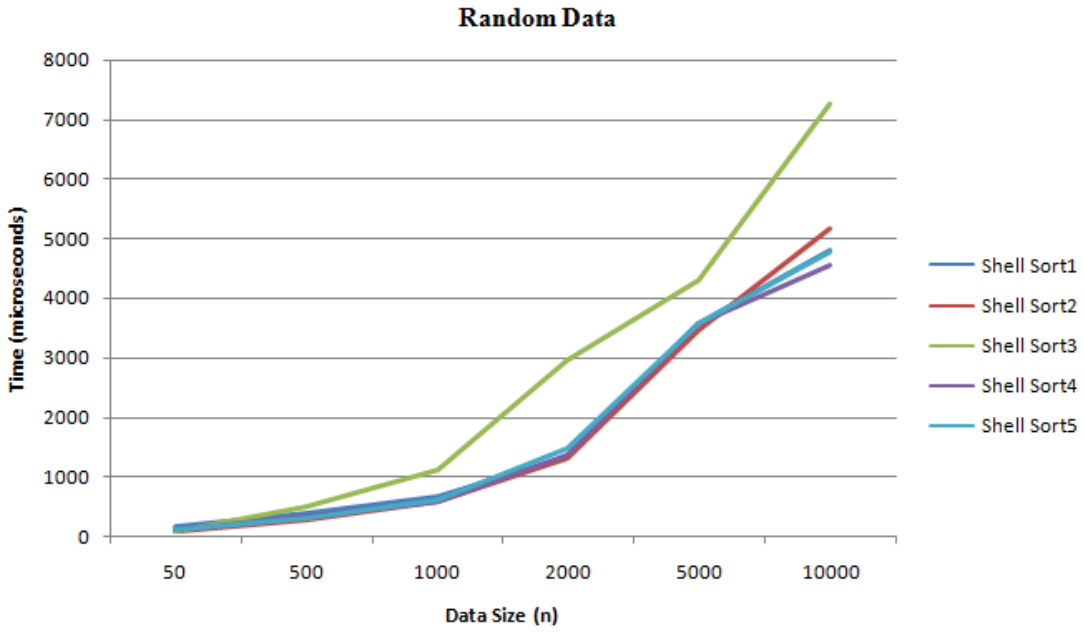
Shell Sort3 - Knuth's sequence to the nearest composite

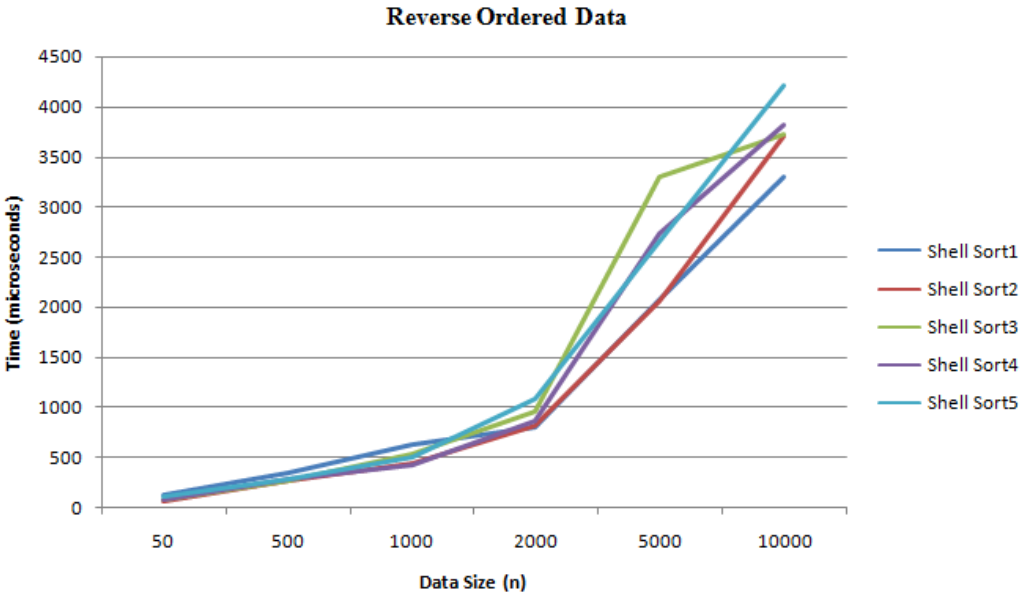
Shell Sort4 - Marcin Ciura's sequence

Shell Sort5 - Hibbard's sequence

Their performance can be graphed as below:







From our observation above, Hibbard's sequence tends to give a better performance for ascending and reverse ordered data whereas Knuth's sequence to the nearest composite seems to give a better performance for random data. After looking into the sequence, I find Hibbard's sequence tends to give more increments compared to other sequence which can be a potential reason leads to better performance. In summary, the performance of the shell sort is highly dependent on the gap sequence selected.

Alternative Approach

The program mainly utilizes array for data storage and sort implementation, it would be interesting to implement linked structure for all series of tasks we do in the program. By using array, we have a easier time to access data and conduct comparison. However, it's also important to see how sorting algorithm

work in a linked data structure. Furthermore, we can explore further by testing it on different kind of linked structure such as double linked list, multi-linked list, or linked list with headers.

Also, all the sorting algorithm is implemented using iterative approach because it's more convenient to test against large data set. Yet, it is worth trying to see how recursion can be used in each of the sorting algorithms and see how the performance differs compared to iterativon based implementations.

Time And Space Efficiency

As the program uses insertion sort, heap sort, and shell sort on datasets, the big O of the program would be O(n^2) driven by the worst case of the insertion sort. However, if we get rid of the insertion sort, the worst case of shell sort would then be the worst case of the program which is O(n(logn)^2).

From a space requirement, as we can determine the size of the data before we allocate any space, the program is fairly efficient in space efficiency with 7 * size n array initiated for each of the 7 sorting methods. Potentially we can also only keep 2 size n array where one array contains unsorted data and the other one resets and take unsorted data for sorting at each round of sorts.

Learning And Takeaway

There are many takeaways from this lab. One of the most important ones is how the selection of data structure and sorting algorithm can lead to huge impact on time and space complexity for the whole program. For example, by getting rid of insertion sort, the program's big O can change from O(n^2) to O(n(logn)^2). Also, different sequence and increments used for shell sort can also leads to significant different in time efficiency. Furthermore, space requirement of the sorting algotirhm is also a key factor should be taken into account when design the program

Enhancement

There are several enhancements in the program to improve the process flow in the aspect of sorting methods, test cases, user I/O, and program methods.

To begin with, insertion sort and two extra shell sorts are created in order to better compare relative performance. Also, extra data size is used to increase the test case and give more observations.

As a part of the design, there's also enhancement at user I/O level, it grants user the option to choose whether they want to provide input and output locations as command line arguments or not. To be more specific, they can state input and output file locations as command line arguments so the program gets the I/O information directly from there. Meanwhile, they can also run the program without providing any command line arguments and the program would prompt user for I/O locations in the console and read from there. In addition, seven output files are generated based on one output location user provided.

In addition to general design, there are also multiple methods created to enhance the program structure and functionality such as print array, return array contents as string, and copy one array to another one.